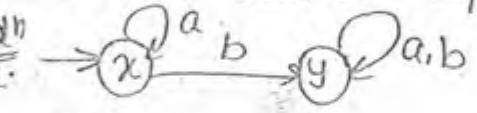
Dated 25.Oct.10

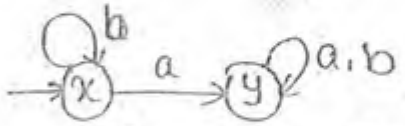
COMPILER

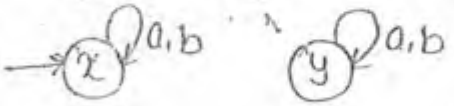
Day-1

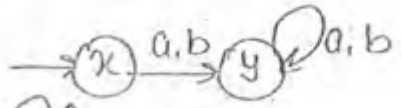
Q. How many possible finite automata's are there, where x is always initial state over the alphabet x and y or a and b.

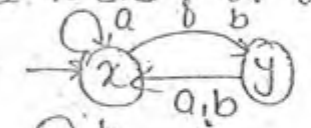
Sol"

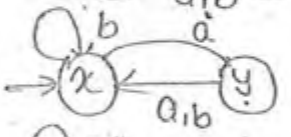
1. 

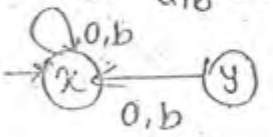
2. 

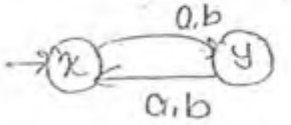
3. 

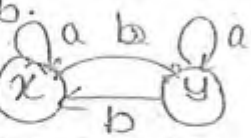
4. 

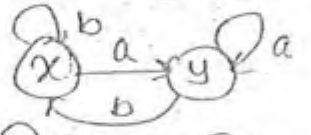
5. 

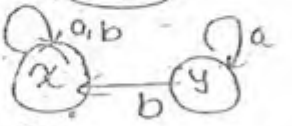
6. 

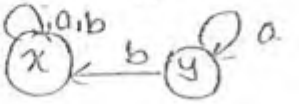
7. 

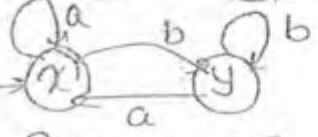
8. 

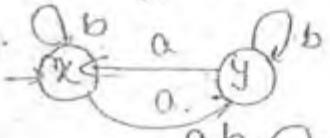
9. 

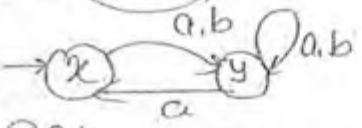
10. 

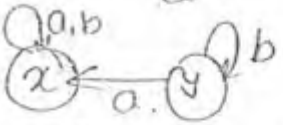
11. 

12. 

13. 

14. 

15. 

16. 

find - On the basis of No of final states

final 2 (x & y) = 16

1 (x (or) y) = 16 / 16 = 32

0 (no final) = 16

64 Ans

→ Consideration of No of possible finite automata's can be done as:-

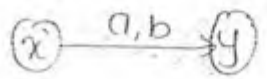
		a	b	
2	x	2	2	→ 4
2	y	2	2	→ 4
4				→ 16

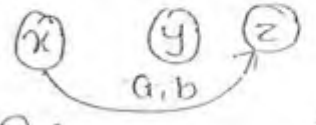
16 × 4 = 64 Ans

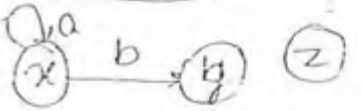
Q. How many possible finite automata's with three states x, y, z where x is always initial states over the alphabet a and b.

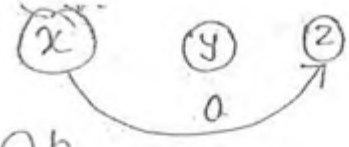
Sol"

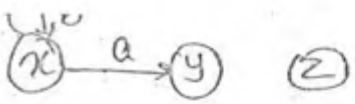
1. 

2. 

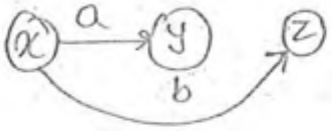
3. 

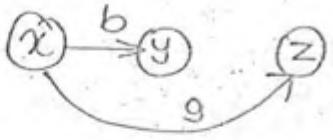
4. 

5. 

6. 

7. 

8. 

9. 

		a	b	
2	x	3	3 ⇒ 9	
2	y	3	3 ⇒ 9	729
2	z	3	3 ⇒ 9	
8				5832 Ans

final
0 → 1
2 → 3
1 → 3
3 → 3

QN How many possible finite automata's are there states x, y & z over the alphabet a and b

Sol" x is initial → 5832
y is initial → 5832
z is initial → 5832
= 3×5832
= 17496

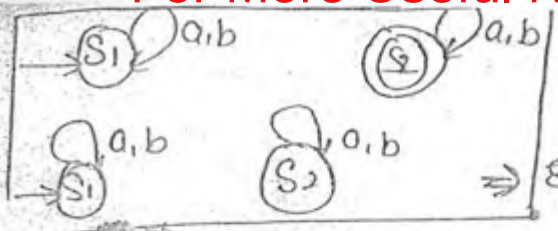
QN How many possible finite automatas are there with three states x, y and z, over the alphabet a, b and c, where x is both initial and final.

	a	b	c	
x	3	3	3	3^3 ⇒ 19,683 Ans
y	3	3	3	
z	3	3	3	

QN How many possible finite automatas are there with state x any, where x is always initial state over the alphabet a, b that accepts empty language.

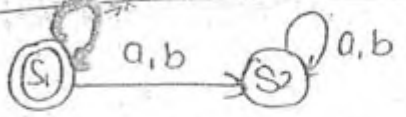
Sol"

		a	b	
2	x	2	2	2^4 ⇒ 16
2	y	2	2	
4				16×4 = 64



⇒ Empty language. (final state is not reachable)

⇒ Empty language (not any final state)



⇒ 1, not empty language.

possibilities of finding the automatas which accepts empty language.

→ No final state = 16 (Basic and easy)

→ final state are not reachable = 4

20 possibilities

If there are two states then-

2 — 16 (both are final) ✗

1 → 16 (x is initial) ✗

16 (4) ✓

0 → 16 (Empty language) ✓

QN How many possible finite automates are there with two states x and y, where x is always initial state with alphabet a and b, that accepts everything.

Soln 2 → 16 ✓

1 → x → 16 (4) ✓ ⇒ 20

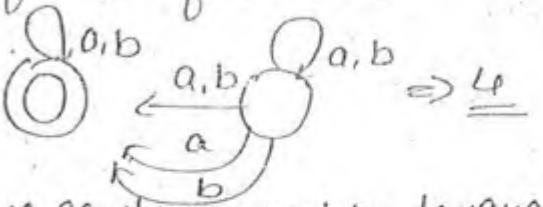
y → 16 ✗

0 → 16 ✗

If y is final state ⇒

(y not accepting ϵ). ✗

If x is final then-

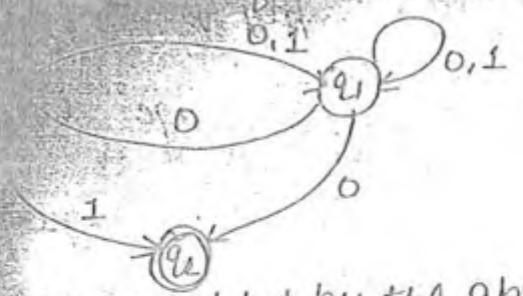


⇒ 4

Total possibilities = 20

Note:- If 20 DFA's are there which are accepting empty language then 20 DFA's should be there which will accepts everything, because of complementation rule.

finite automata →

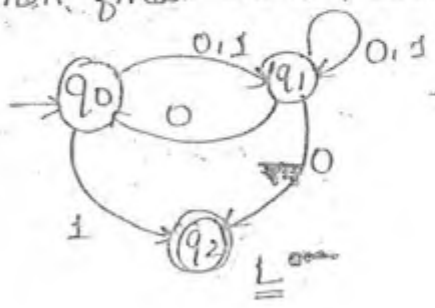


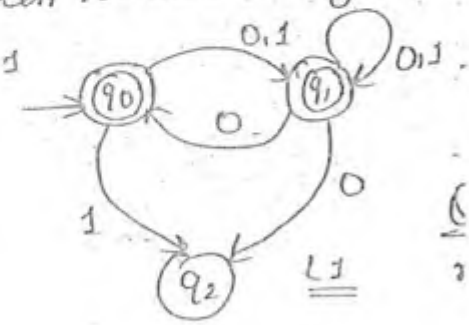
age accepted by the above finite automata and ge accepted by the above finite automata by al and non final states, then which one of the tent :-

1) $L = (0+1)^*$

$L_1 \subseteq (0+1)^*$

$L_1 = L$





If it is DFA then $\Rightarrow (0+1)^* - L$ (Remove L from Σ).

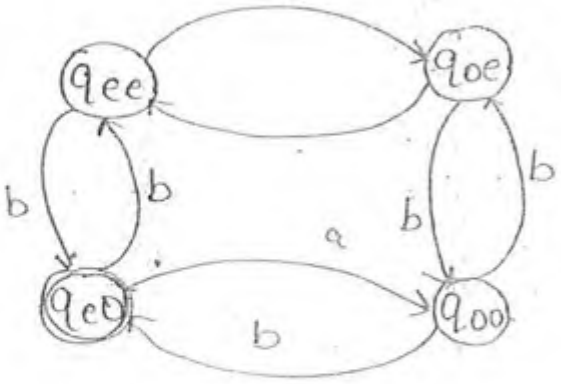
Note:- In the case of DFA we will always get complimented finite automata, but in the case of NFA, we will not always get complimented finite automata (manual checking is only solution).

QN Construct finite automata that accepts all strings of a's and b's where no of a's in given string is even and no of b's in the given string is odd.

Soln

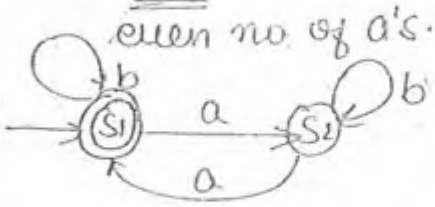
S

S_1 even a's S_2 odd b's

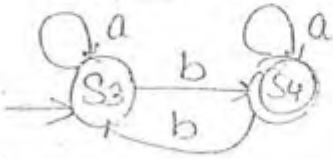


P1

even no of a's.



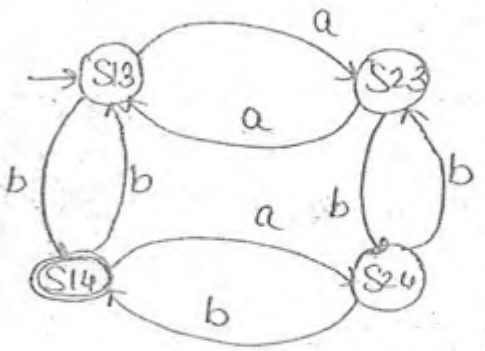
P2 (odd no of b's)



By P1 & P2

$S_{13} \xrightarrow{a} (S_1, a) \cup (S_3, a) = S_2 S_3$

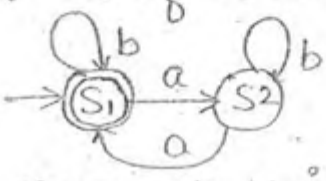
$S_{13} \xrightarrow{b} (S_1, b) \cup (S_3, b) = S_2 S_4$



final state ⇒ where both the finals are there.

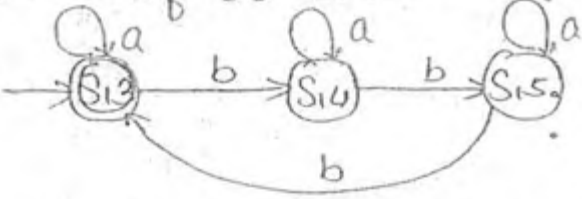
Q.11 Construct FA which accepts all strings of a's and b's in which no of a's are divisible by 2 and no of b's are divisible by 3.

P_1: no of a's divisible by 2.



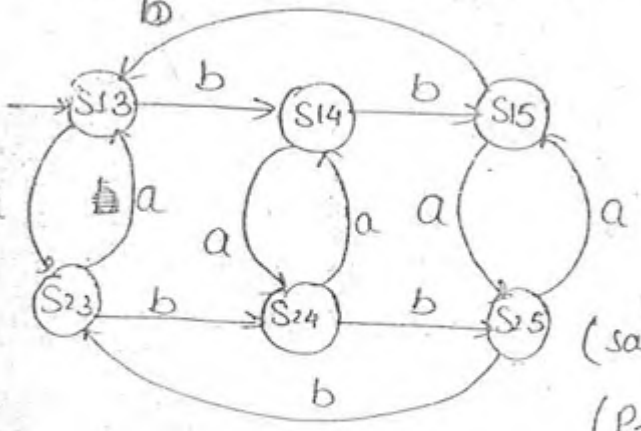
How many states
= 2×3 = 6 states.

P_2: no of b's divisible by 3.



$S_{13} \xrightarrow{a} S_{23}$ $S_{15} \xrightarrow{a} S_{25}$

$S_{13} \xrightarrow{b} S_{14}$ $S_{15} \xrightarrow{b} S_{13}$



Final states

$P_1 \wedge P_2 = S_{13}$

$P_1 \cup P_2 = S_{13}, S_{14}, S_{15}, S_{23}$

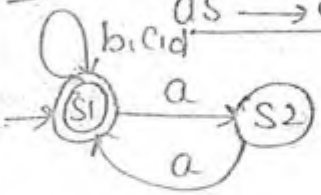
$P_1 - P_2 = S_{14}, S_{15}$
(satisfying 1st but not second one)

$(P_2 - P_1) = S_{23}, S_{24}, S_{25}$

Q4 Find the minimum no of state required to construct minimum dfa which will accept all strings of a's, b's and c's, d's where the no of a's divisible by 2, No of b's not divisible by 3, No of c's not divisible by 5, No of d's not divisible by 7. How many minimum no of states required.

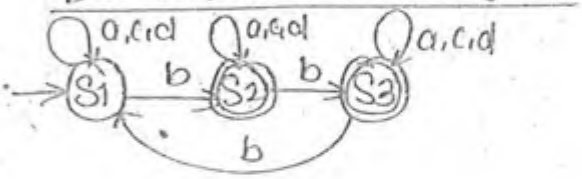
Soln P1

a's → divisible → 2



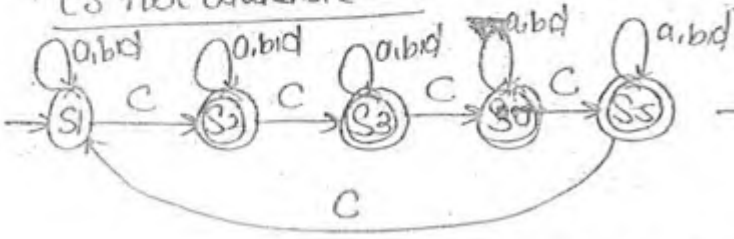
P2

b's → not divisible by 3



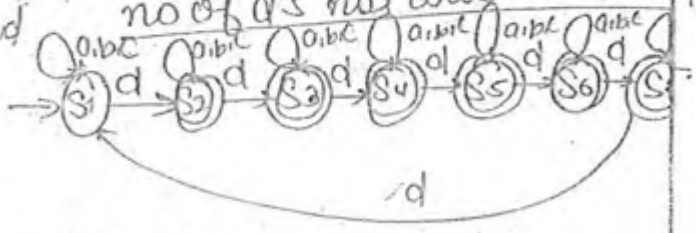
P3

c's not divisible - 5



P4

no of d's not divisible by 7

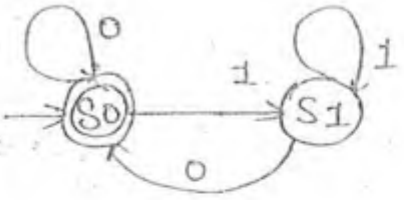


Minimum No of states = 2 * 3 * 5 * 7 = 210 Ans

Q4.2 Construct finite automata that accepts all binary number which are divisible by 2.

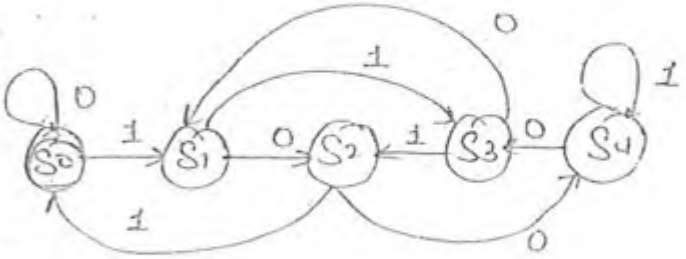
Soln If initial and final states are same then this method can be utilized ⇒

	0	1
→ S_0^*	S_0	S_1
S_1	S_0	S_1



⇒ all binary no divisible by 5 - (Remainders = 0, 1, 2, 3, 4)

	0	1
→ S_0^*	S_0	S_1
S_1	S_2	S_3
S_2	S_4	S_0
S_3	S_1	S_2
S_4	S_3	S_4



→ all ternary no divisible by 7 -

	0	1	2
→S*	S0	S1	S2
S1	S3	S4	S5
S2	S6	S0	S1
S3	S2	S3	S4
S4	S6	S6	S0
S5	S1	S2	S3
S6	S4	S5	S6

Binary or ternary no matters

always contain 7 states

Note:- The minimum no of states required to construct finite automata that accepts all base m numbers, which are divisible by n, Contain n-states.

→ all binary no's divisible by 5 and starting with 0.

	0	1
→S	S0	D ⇒ start with 0
* S0	S0	S1
S1	S2	S3
S2	S4	S0
S3	S1	S2
S4	S3	S4
D	D	D ⇒ start with 1

divisible by 5

⇒ 7 states

→ all binary no divisible by 9 and starts with 1

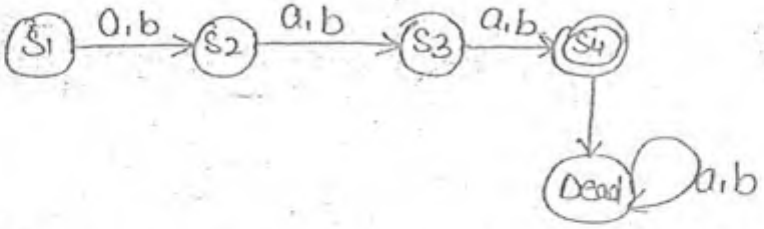
	0	1
→S	D	S1 ⇒ starting with 1
* S0	S0	S1
S1	S2	S3
S2	S4	S5
S3	S6	S7
S4	S8	S0
S5	S10	S2
S6	S3	S4
S7	S5	S6
S8	S7	S8
D	D	D ⇒ starting with 0

Divisible by 9

⇒ 9+2 = 11 states

QN Construct a FA that accepts all strings of a's and b's, where the length of the string of exactly 3.

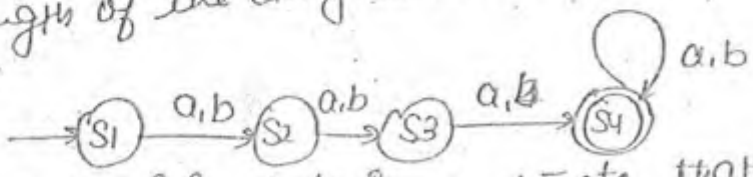
Soln



⇒ 3+2 = 5 states

Note:- The minimal finite automata that accepts all strings of a's and b's, where the length of string is 'n' exactly contains (n+2) states.

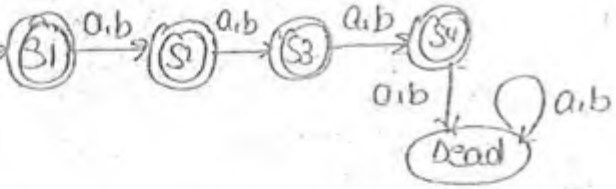
QN Construct minimal FA that accepts all strings of a's and b's where the length of the string is atleast 3:



Note:- The minimal finite automata that accepts all string's of a's and b's, where the length of string atleast n contains (n+1) states.

QN FA, accepts all strings of a's and b's where the length of string is almost three.

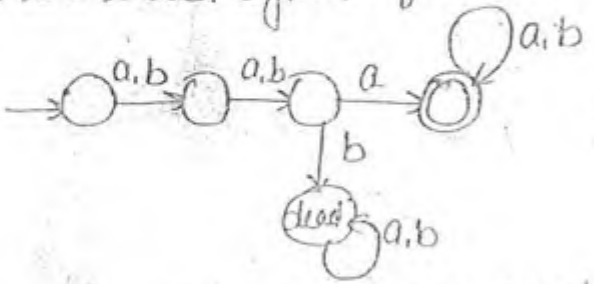
Soln



= 3+2 = 5 states

Note:- Where the length of strings are almost n, Contain n+2 state

QN FA, that accepts all strings of a's and b's, where each string conta 'a' as the third symbol from LHS.



⇒ 3+2 = 5 states

Note:- The minimal FA that accepts all strings of a's and b's where n'th symbol from L.H.S. Contain n+2 states.

Note:- The minimum FA which accepts all the strings of a's and b's, where nth symbol from the right hand side is b contains 2^n states.

dated 5.Oct.10

$L_1 = \{a^n \mid n \geq 1\}$ → Regular

$L_2 = \{a^n b^n \mid n \geq 1\}$ → CFL (Regular + 1 stack)

$L_3 = \{a^n b^n c^n \mid n \geq 1\}$ → CSL

$L_4 = \{a^m b^n \mid m \neq n,\ m, n \geq 1\}$ → CFL

$L_5 = \{a^l b^m c^n \mid l \neq m \text{ or } m \neq n\}$ → CFL

How can we say that what the language is given as:- ($l, m, n \geq 1$)

If we want to check that given language is regular or not) ⇒ Language

- Finite language (Regular language)
- Infinite language
  - Memory required (non-regular) (any comparision)
  - Memory not required
    - not in AP (not regular)
    - in AP (Regular)

Examples

$L = \{a^m b^m \mid m \leq 10,000\}$

Regular language (just because of finiteness)

$L = \{a^n \mid n \geq 1\}$ (Regular)

Here we are using the concept of generation of series.

$L = \{a^{n!} \mid n \geq 1\}$

Theorem:- For a given problem if you can construct the CFG, then it can surely be solved by T.M.

Non-regular (not in AP)

$L = \{a^{2^n} \mid n \geq 1\}$ = not regular (not in AP)

④ $L = \{a^{2n} \mid n \geq 1\}$ ⇒ even no of a's.
Regular language (in AP)

⑤ $L = \{a^n \mid n \geq 1\}$
⇒ not regular (not in AP)

⑥ $L = \{a^n b^n \mid n \geq 1\}$
⇒ not regular (memory required)

⑦ $L = \{w w^R \mid n \geq 1\}$
⇒ non regular (memory required)

⑧ $L = \{a^i b^j \mid gcd(i,j) = 1\}$
⇒ non regular (memory required).

ϕ Regular ∧ CFL ⇒ CFL
$(a+b)^* \wedge a^n b^n = a^n b^n$
(Intersection of lower and higher language will always go to higher language).

ϕ CFL ∧ CFL = not CFL (CSL) → (need not be)

$a^l b^l c^m \wedge a^m b^n c^n$

$a^l b^l c^l$ (not CFL)

ϕ $a^n b^n c^n$ = CSL ⇒ Compliment of CSL is need not be CSL.
⇓ ⇒ Compliment of CFL, need not be CFL, it can be CSL. (CFL's are not closed under compliment ation).
$(a^n b^n c^n)^l$
⇓
CFL

Note:- ① Insersection → CFL ⇒ need not be CFL or (CSL)
② Compliment of CFL ⇒ need not be CFL or (CSL)
③ Intersection of Regular and CFL is always CFL.

Q. Give the regular expression that derives all strings of 0's & 1's, where each string begin with a and end with b.

$R = \{a.(a+b)^*.b\}$ Ans ⇒ Concatenation order is important

$a+b = b+a$ (order can be changed)

QN Regular expression, where the first and last symbols are different

⇒ $a(a+b)^*b + b(a+b)^*a$

QN Regular expression, that derives all string's of 0's and 1's, where each string starting and ending symbols are same.

⇒ $a(a+b)^*a + b(a+b)^*b + \lambda + a + b$

QN Give the Regular expression that derives all strings of a's and b's, where all strings contains abb as substring.

⇒ $(a+b)^*abb(a+b)^*$

QN Regular expression, where the length of string is exactly three.

⇒ $(a+b)(a+b)(a+b) = (a+b)^3 \Rightarrow (a+b)^n$

QN Regular expression, where the length of string is atleast 3.

⇒ $(a+b)(a+b)(a+b)(a+b)^*$ ⇒ more effecient

⇓ (or)

$(a+b)^*(a+b)^3$ ⇒ more effecient

⇓ (or)

$(a+b)^*(a+b)^3(a+b)^*$ ⇒ more correct but not effecient

QN Regular expression, where the length of string is atmost 3.

$= \lambda + (a+b) + (a+b)(a+b) + (a+b)(a+b)(a+b) = (a+b+\lambda)^3$

QN Regular expression, where the length of string is even.

$= ((a+b)^2)^*$

QN. Odd length-

$(a+b)((a+b)^2)^*$ or $((a+b)^2)^*(a+b)$

QN Regular expression, that where each string starts with a and not having two consecutive b's.

$= (a+ab)^+$ (or) $a(a+ba)^*(\lambda+b)$

$(a+ab)^*$

QN Regular expression, where each string does not contain 2 consecutive a's and b's

Soln $= (b+\epsilon)(a+b)^*(a+\epsilon)$ or $(a+\epsilon)(ba)^*(b+\epsilon)$

QN Regular expression, where each string contain exactly 2 a's.

$= b^* a b^* a b^*$

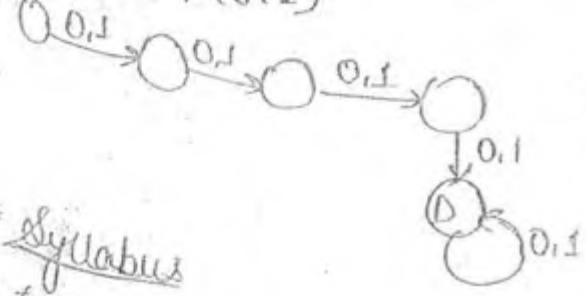
QN Regular expression, where each string contain atmost two a's.

$= b^* + b^* a b^* + b^* a b^* a b^*$

QN $b^*(a+\epsilon) b^* (a+\epsilon) b^*$

QN find the minimal states of dfa that accepts described by the R.E. $= (0+1)(0+1)(0+1)(0+1)$ ..... n-times

$(0+1)(0+1)(0+1)$



If $n=3$, $\Rightarrow$ 5 states $\Rightarrow (3+2)$

If $n=n$ then $(n+1)$ states

* Syllabus

1. Lexical analyser
2. Parsing
3. Syntax directed Translation
4. Intermediate code generation
5. Code Optimization

} $\Rightarrow$ Imp

References

→ Compiler Technique by Aho Ullmen & Ravisethi

INTRODUCTION

Compiler
⇓ HLL
Assembly level language

* Compiler is a Convertor, that can convert High level language into Assembly level language.
* In this conversion, we are using 6-phases, which are as follows:-

⇓ HLL
- Lexical Analyser
- Syntax Analyser
- Semantic Analyser
- Intermediate code generator
- Code optimization
- Target code generation

⇓ ALL

Symbol table

* C-Compiler knows everything about C-language.

⇓
the thing that are not known to C-Compiler, is stored in symbol table.

Error Handler ⇒ used to handle all the errors made by user.

Exp:- $x = a + b * 60.5$

⇓ Lexical Analyser

$(id_1, 1) = (id_2, 2) + (id_3, 5) * 60.5$

7 tokens

$id_1 = id_2 + id_3 * 60.5$

⇓ Syntax Analyser

Symbol table

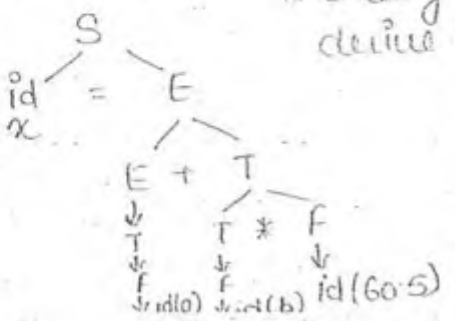
1	int	x
2	int	a
3	int	b

x, a, b = identifiers

* C language compiler, uses CFG, to define all the arithmetic expression

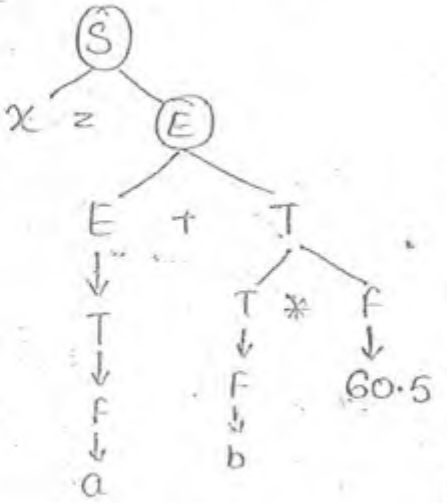
$a + b = c$ = Syntax error
⇓
CFG can't generate this

$S \rightarrow id = E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow id$



* Type checking will never be take by the syntax and will done by semantic analyser.

Semantic Analysis (Type checking)



- How can we multiply int to floating point number?
- for this semantic analyser uses the Implicit conversion by float to int (60.5) = 60

* Type mismatch type errors are given by semantic analysis.

---

x = a + b * 60 (CFG take care of priorities itself)

Intermediate Operations or code

$t_1 = b * 60$ (t_1, t_2) = temporary variables

$t_2 = a + t_1$

$x = t_2$

⇓

Code Optimization

⇓

$t_1 = b * 60$

$x = a + t_1$

⇓

Target Code

⇓

MOV b, R1 (b to R1).
MUL R1, 60 (R1 = R1 * 60)
MOV a, R2 (a to R2)
ADD R1, R2 (R1 = R1 + R2)
MOV R1, x (R1 to x)

L.A.
Syntax
Semantic
I.C.G.
} front end

Code Optimiz. ⇒ optional phase

Target code ⇒ Back end

Front end = Depends upon source language

Back end = Depends on processor

In order to achieve the portability, we separate the phases of analysis

for the same source code, we can generate different assembly language

sets; based upon type of processors.

Types of Compiler

1. Single pass Compiler
2. Multi pass Compiler

(1) Single pass Compiler:- all the 6-modules at a time in memory

Disadvantage :-
① Wastage of space.

Advantage
① No Divide and Conquer (less time required).

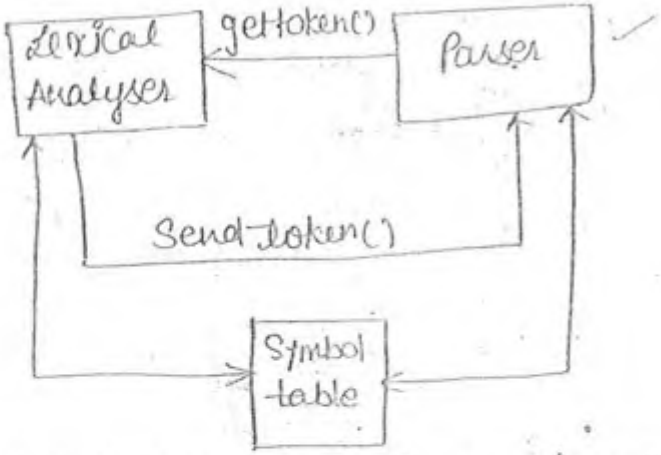
(2) Multi pass Compiler:- front end and back end are placing seprately in memory.

Disadvantage
① More time required (Divide and conquer)

Advantage
① Less space required.

Chapter No 1

Lexical Analysis



* Initially the I/P is given to parser. parser calls lexical analyser for tokens

Lexical Analyser :- ① It will read the given I/P strings and divides the string into some meaningful groups or words which are called as tokens
② It will remove the comment lines in the given C pgm.

3) It will eliminate white space characters, in the source co
White Space Characters → blank(Space), tab, newline characters

4) It will help to provide error messages. It scan each and eve
line of source code. The line number is also provided by the
lexical analyser.

OR

Q4 Find the no of tokens in the following C pgm:-

```
int max(i,j)
 int i, j;
 /* return max of i and j */
 {
   return i>j ? i : j;
 }
```

Sol^n

```
int | max | ( | i | , | j | ) |
int | i | , | j | ; |
{ |
return | i | > | j | ? | i | : | j | ; |
} |
```

= 23 tokens Ans

Q5 By not seeing the next symbol, we can say that is a token:-

a) ++

b) > ⇒ = may be, then (>=) ⇒ token

c) main ⇒ may be main(or some user defined variable

d) < ⇒ = may be, then (<=) ⇒ token

Q6 Find the no of tokens.

```
printf ("Hai x = %d", i);
```

Sol^n printf | (| " Hai x = %d " | , | i |) | ;

1 2 3 4 5 6 7

⇒ Inside " " not need to enter = 7 tokens Ans

Q7 Find the no of tokens

```
printf ("i = %d, &i = %x", i, &i);
```

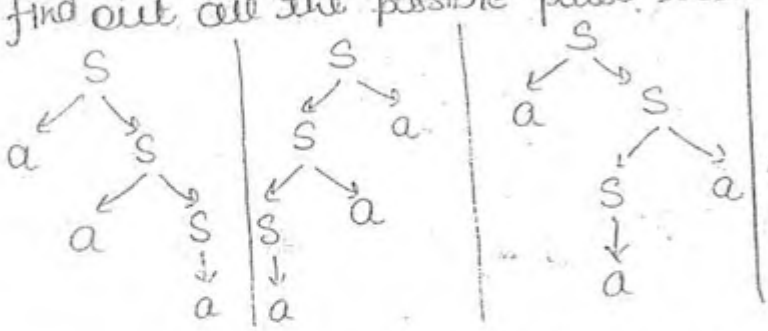
Sol^n printf | (| " i = %d, &i = %x " | , | i | , | &i |) | ; | = 10 tokens Ans

ϕ Grammar

Exp:- $S \to aS \mid Sa \mid a$

$w = aaa$

find out all the possible parse trees.



$S \to aS \mid Sa \mid a$

$\Downarrow \quad \Rightarrow a^+$

$S \to a \mid aS \Downarrow$

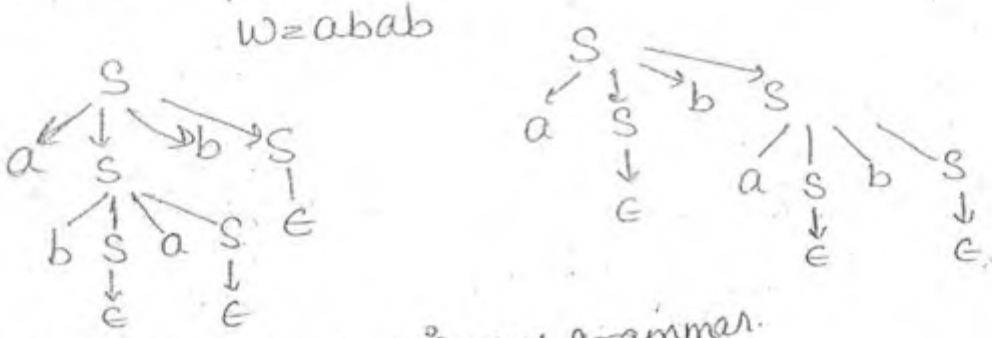
$\Rightarrow a^+$

The above grammar is ambiguous grammar, because more than 1-parse tree is available to derive a string $w = aaa$

ON check the following grammar is ambiguous or not.

$S \to aSbS \mid bSaS \mid \epsilon$

$w = abab$

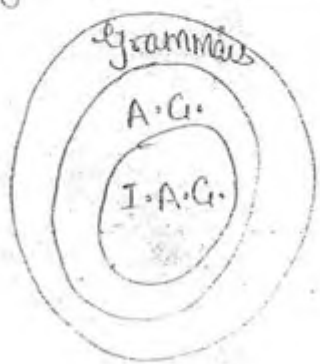


Given grammar is ambiguous grammar.

Note:- ① There is no algorithm to check whether the given grammar is ambiguous grammar or not so it is an undecidable problem.

② There is no algorithm, to convert the given ambiguous grammar into unambiguous grammar. It is also undecidable problem.

③ Those ambiguous grammars, from which we can not eliminate ambiguity, is called as inherently ambiguous grammars.



Grammar

CFG

Left Recursive Grammar

$A \to Aa \mid a$

$A \to A\alpha \mid \beta$

$\alpha, \beta \in (T \neq V)^*$

A = Single variable

Right recursive Grammar

$A \to aA \mid a$

$A \to \alpha A \mid \beta$

$\alpha, \beta \in (V+T)^*$

A = Single variable

Note ① For every Left Most Derivation Tree, Right Most Derivation Tree is also possible.

② If the given grammar is unambiguous grammar, LMDT, RMDT both are same.

③ Top Down Parser will always keep the LMRecursive grammar into infinite loop

Exp:- $A \to A\alpha \mid \beta$

$w = \beta\alpha^{100}$

A
↓
$A\alpha$
↓
$A\alpha$
↓
$A\alpha$
|
|

⇒ Infinite loop

repeated again & again

Exp:- Right Recursion (Top Down Parser)

→ No problem will occur.

$A \to \alpha A \mid \beta$

$w = \alpha^{100}\beta$

A
↓
αA
↓
αA
⋮
$\alpha^{100} A \Rightarrow \alpha^{100}\beta$
↓
β

Elimination of Left Recursion

Extended Backneous Normal form (EBNF)

Exp:- $E \to E+T \mid T$

$T \to T * F \mid F$

$F \to id$

Sol^n $E \to E+T \mid T$

$T(+T)^0$
$T(+T)^1$
$T(+T)^2$
$T(+T+T)^3$
$T+T+T+T+T+T+T$ ——

$E \to T\{+T\}$

$T \to T*F \mid F$

$F(*F)^0$
$F(*F)^1$
$F(*F*F)^2$
$F(*F*F*F)^3$
$F*F*F*F*$ ——

$T \to F\{*F\}$

$\Rightarrow$
$E \to T\{+T\}$
$T \to F\{*F\}$
$F \to id$

Dm

Drawback:- Equivalent grammar is not CFG.

Exp:- $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

$\Rightarrow$ Eliminate left recursion:

Recursion should be there but not left recursion.

Sol^n

$E \to E+T \mid T$ | $T \to T*F \mid F$ | $F \to id$
$E \to TE'$ | $T \to FT'$ |
$E' \to +TE' \mid \epsilon$ | $T' \to \epsilon \mid *FT'$ |

Dm

Exp:- Eliminate left recursion-

$S \to aBDh$
$B \to Bb \mid h$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

Sol^n

$B \to Bb \mid h$ | $S \to aBDh$
$B \to hB'$ | $D \to EF$
$B' \to \epsilon \mid bB'$ | $E \to g \mid \epsilon$
| $F \to f \mid \epsilon$

Dm

minate left recursion

(L) | a
L, S | S

S → (L) | a
L → SL'
L' → ∈ | , SL'

Eliminate left recursion

→ Aa | b
→ Ac | Sd | ∈

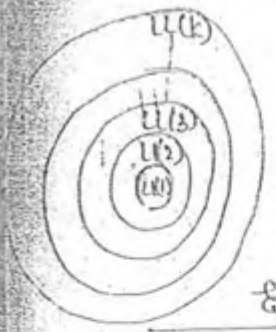
S → Aa | b
A → Ac | Aad | ∈ | bd

⇓

S → Aa | b
A → bdA' | A'
A' → cA' | adA' | ∈

Left factoring

Parser sees one symbol at a time, from left to right.



S → aα1 | aα2 | aα3

w = aα3

⇒ Parser is confused to choose out of these, bcoz all are giving 'a'. This is known as left factoring.

⇓ Elimination

S → aβ
β → α1 | α2 | α3

QN Eliminate left factoring from the following grammar :-

S → iEtS | iEtSeS | a
E → b

Sol^n S → iEtSS' | a
S' → ∈ | eS
E → b

QN Eliminate left factoring-

$S \to aAd \mid aB$

$A \to a \mid ab$

$B \to ccd \mid ddc$

Sol^n $S \to aS'$

$S' \to Ad \mid B$

$A \to aA'$

$A' \to \epsilon \mid b$

$B \to ccd \mid ddc$

QN $S \to abc \mid abd \mid aef$

Sol^n $S \to aS'$

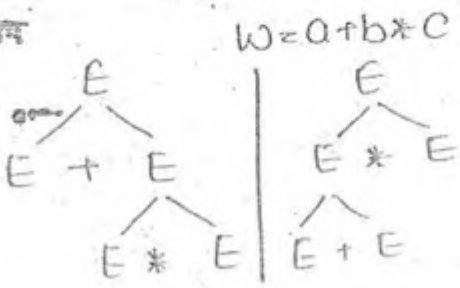
$S' \to bc \mid bd \mid ef$

$S' \to bB' \mid ef$

$B' \to c \mid d$

⇒ all having the same priorities

ϕ $E \to E+E \mid E*E \mid E/E \mid E-E \mid id$



ambiguous grammar

$E \to E+T \mid T$ ⇒ lower priority

$T \to T*F \mid F$ ⇒ higher priority

$F \to id$

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

$F \to id$

* In CFG, the thing that will be on lower level, will be evaluated first
* The operator which have lower priority, make that root

QN Convert the following ambigious into unambiguous grammar.

$R \to R+R \mid R \cdot R \mid R^* \mid a \mid b$

Sol^n $R \to R+T \mid T$

$T \to T \cdot F \mid F$

$F \to (G)^* \mid G$

$G \to a \mid b$

QN ambigious → unambigious

$Bexpr \to Bexp \text{ or } Bexp \mid Bexp \text{ and } Bexp \mid \text{not } Bexp \mid 0 \mid 1$

Sol^n $Bexpr \to Bexp \text{ or } T \mid T$

$T \to T \text{ and } F \mid F$

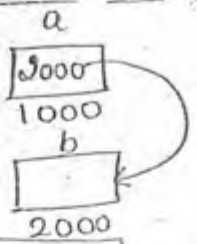
$F \to \text{not } G \mid G$

$G \to 0 \mid 1$

End

Some extra and important points (D.S.)

```
① int *a;
② int b;
③ a = &b
```



a = variable pointer

```
① int (*fp)()
② int fun();
③ fp = fun;
   (*fp)();
```

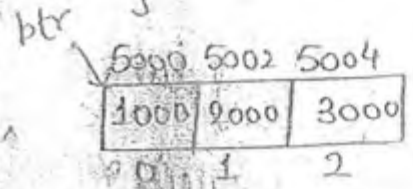
fp = functional pointer pointing to function that returns integer

Exp

```
main()
{
  int (*ptr[3])();
  ptr[0] = aaa;
  ptr[1] = bbb;
  ptr[2] = ccc;
  (*ptr[2])();
}
```

```
1000
aaa()
{
  printf("aaa");
}
2000
bbb()
{
  printf("bbb");
}
```

```
3000
ccc()
{
  printf("ccc");
}
```



O/P = ccc

CHAPTER No. 3 (Parsing)

DH.1 int (*f)(int, int)

Meaning:- A pointer to a function that takes two integers as I/P and returns O/P as integer.

DH. What does the following 'C' statement declare:-

char * (*(*a[N])()) ()

int (*)() ⇒ pointer to a function that returns integer.

⇒ Array of N pointers, pointing to function, that returns pointer to a function returning pointing to character or character pointer.

DH What does the following C-statement will do-

void (*abc)(int, void (*def)())

Sol"

abc is a pointer to a function, which will return void and which will take two parameter i-

(1) integer parameter

(2) A pointer def to a function, which will return void.

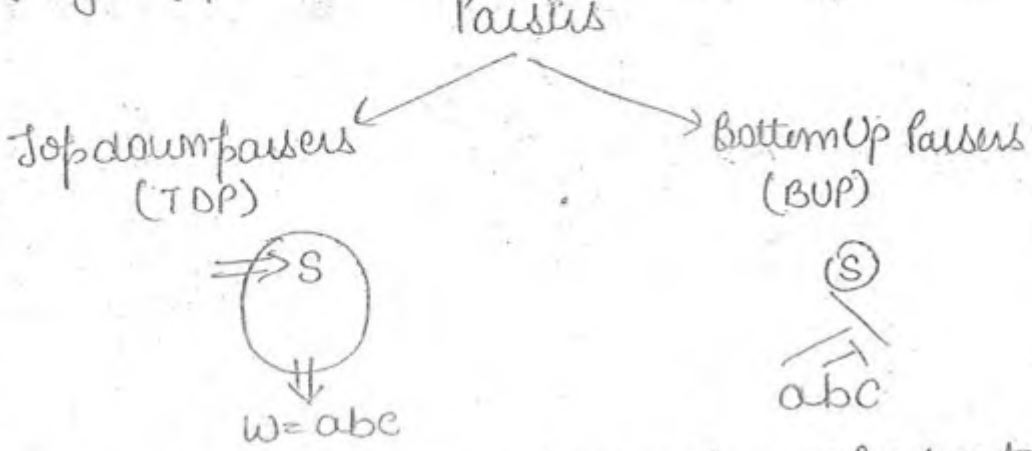
! (char *(*)()) (*ptr[N])()

Array of N pointers to functions, that will return pointer to a function which will return pointer to a character pointer.

* Chapter No 2 *

Parsing

ϕ Type of parsers



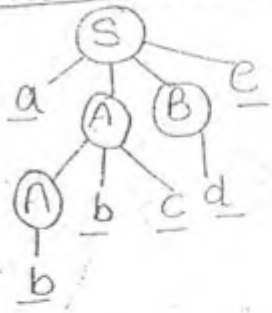
TDP :- In TDP, we will derive the given string by taking the start symbol.

BUP :- In BUP, we will take the string and finally we will get the start symbol.

Working of TDP ⇒ TDP follow the left most derivation.

Exp :- S → aABe
A → Abc/b
B → d

w = abbcde
↑ look ahead symbol.



TDP, only sees one symbol at a time.
If there are two possibilities, choosing the right one is difficult.

note1 :- In TDP we are using left most derivation.

note2 :- The difficulty in TDP, is that if a variable is having more than one possibility, choosing right production.

ϕ L

L =
L =
to

L

L =
R =

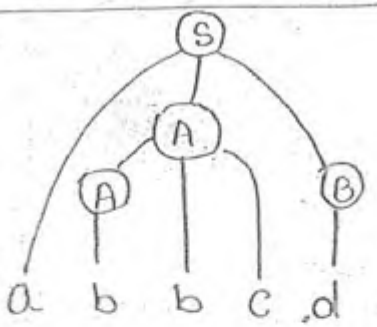
late
most
late
fanc
go t

B

Br
n

Working of Bottom-UP parser

Exp:-



$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$w = abbcde$

Note-1:- In Bottom up parsing, we are using reverse of Right most derivation to derive the string.

Note2:- The difficulty in bottom up parsing finding the substring (handle), which will give our required variable, so that we will go to start symbol.

Top Down Parsers

TDP (No-LR, No-LF)

- with Backtracking → Brute force method
- without backtracking → predictive parsers
  - Recursive Decent Parsers
  - Non-Recursive Decent Parsers (LL(1))

LL(1) Parser

L = left-right (reading the I/P symbol)

L = using left most derivation taking 1 symbol at a time

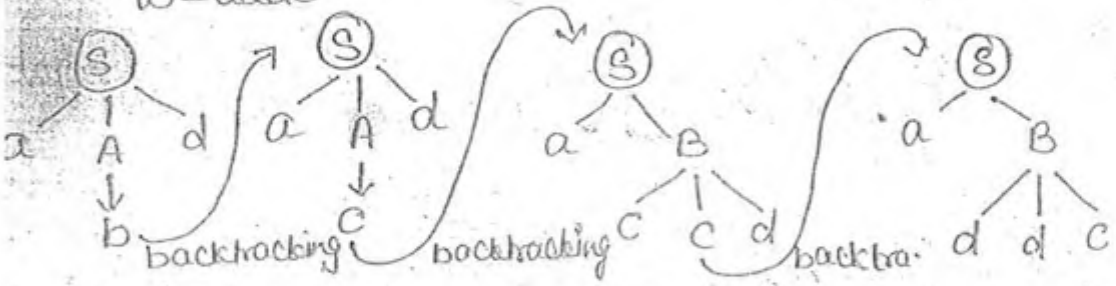
LR(1) Parser

L = left-right (reading the I/P symbol)

R = using right most derivation, taking one symbol at a time

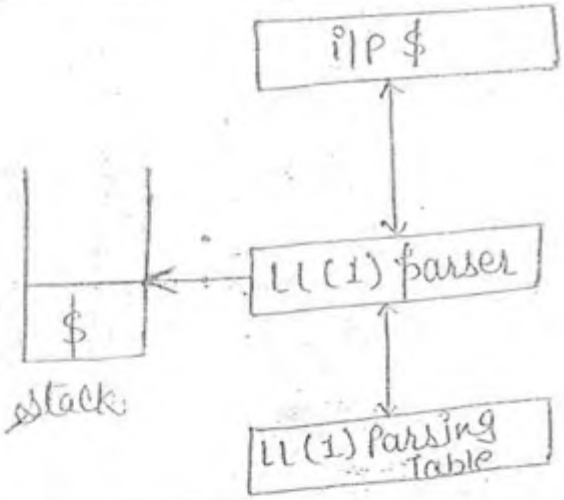
Method

S → aAd | aB
A → b | c
B → ccd | ddc
w = addc



Drawback:- It will take a lot of time in backtracking. It is not good for debugging.

2) Non-Recursive Decent Parsers:-



LL(1) Parsing Algo

If X is top of the stack & 'a' is a lookahead symbol-

1. If ((X == a) == $) then, successful completion.
2. If ((X == a) ≠ $), pop out from the stack, increment i/p pointe
3. If (X is non terminal) then use LL(1) parsing table entry M[X,
   If M[X,a], if X → uvw, replace X by uvw in the reverse order
   If M[X,a] = blank space, indicate error.

Exp:
E →
T →
F →
0 = 1
M
E
E'
T
T'
F
Ren
E -
E' -
T -
T' -
F →
IN
dol

Exp:—

$E \to E+T \mid T$
$T \to T*F \mid F$ ⇒ Exp(1)
$F \to id \mid (E)$

i/p = id + id * id $

LL(1) Parsing table

M	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

⇒ Size of LL(1) Parsing table

$m * (n+1)$

m = no. of variables in grammar
n = no. of terminals in grammar
* always we are doing guess, because we want left most derivation.

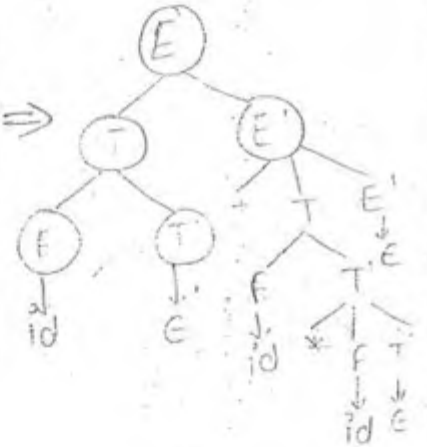
Removal of left recursion

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

⇒

E→TE'	T→FT'	T'→ϵ
T→FT'	F→id	E'→ϵ
F→id	T'→*FT'	
T'→ϵ	F→id	
E'→+TE'		

⇒



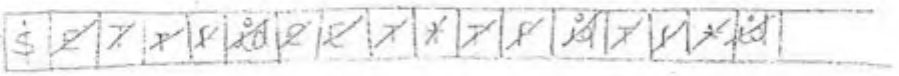
N. w = (id + id * id) $ ⇒ Exp(2)

Total productions =

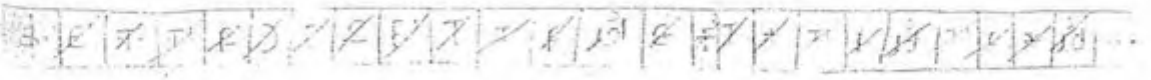
E→TE'	T'→ϵ	T'→ϵ
T→FT'	E'→+TE'	E'→ϵ
F→(E)	T→FT	T'→ϵ
E→TE'	F→id	E'→ϵ
T→FT'	T'→*FT	
F→id	F→id	

Ans

Exp 1 stack



Exp 2 stack



Q. $S \to (L) \mid a$

$L \to L,S \mid S$

$w = (a,a,a)$

	a	,	(	)	$
S	$S \to a$		$S \to (L)$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to ,SL'$		$L' \to \epsilon$	

Parsing table

Sol^n

Q^n $S \to (L) \mid a$

$L \to SL'$

$L' \to \epsilon \mid ,SL'$

$S \to (L)$	$L' \to ,SL'$	$L' \to \epsilon$
$L \to SL'$	$S \to a$	
$S \to a$	$L' \to ,SL'$	
	$S \to a$	

Construction of LL(1) Parsing table

- First(α), gives a set of terminals, that begin in strings derived from α.

Exp:- $\alpha \to abc \mid def \mid Ab$

$A \to e$

gives by α' $\Leftarrow \begin{bmatrix} a \Leftarrow abc \\ d \Leftarrow def \\ e \Leftarrow eb \end{bmatrix}$

Rule-1:- First(α) = α, if α is terminal

Rule-2:- First(ϵ) = ϵ

Rule-3:- $\alpha \to x_1 x_2 x_3$

First(α) = First($x_1 x_2 x_3$)

= First(x_1)

= First(x_1) - ϵ $\cup$ First(x_2, x_3)

only if, $x_1 \to \epsilon$

Exp:- $x_1 \to a \mid \epsilon$

$x_2 \to b \mid \epsilon$

$x_3 \to c \mid \epsilon$

First(x_1, x_2, x_3)

⇓

First(x_1)

a $\cup$ First(x_2, x_3)

⇓

First(x_2)

⇓

b $\cup$ First(x_3)

⇓

c, ϵ

N Find the first for the following grammar-

$E \to TE'$

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

$T \to FT'$

$T' \to \epsilon \mid *FT'$

$F \to id \mid (E)$

Solⁿ

	First()
E	id, C
E'	∈, +
T	id, C
T'	∈, *
F	id, C

IN. Find the first for the following grammar:-

S → aBDh First(S) = {a}
B → cC First(B) = {c}
C → bC | ∈ First(C) = {b, ∈}
D → EF First(D) = {g, f, ∈}
E → g | ∈ First(E) = {g, ∈}
F → f | ∈ First(F) = {f, ∈}

IN. Find the first for the following grammar:-

S → ACB | CbB | Ba First(S) = d, g, h, ∈, b, a
A → da | BC First(A) = d, g, h, ∈
B → g | ∈ First(B) = {g, ∈}
C → h | ∈ First(C) = {h, ∈}

IN. Find the first for the following grammar:-

S → AaAb | BbBa First(S) = c, a, d, b
A → c | ∈ First(A) = c, ∈
B → d | ∈ First(B) = d, ∈

Follow():→ Follow(A) ← variable.

Follow(A) gives set of all terminals, that may follow immediately to the right of A.

Rule 1:- If A a start symbol, then

Follow(A) = $

Rule 2:- If X → αAβ is in G:-

then, Follow(A) = First(β)

Rule-3: If $X \to \alpha A$ (or) $X \to \alpha A\beta$, $\beta \to \epsilon$

Follow(A) = Follow(X)

QN. Find first and follow for the following grammar:-

$X \to aABe$
$B \to c|d$
$A \to a$

	First()	Follow()
X	a	$
B	c, d	e
A	a	c, d

QN. Find first and follow for the following grammar:-

$E \to TE'$
$E' \to \epsilon | +TE'$
$T \to FT'$
$T' \to \epsilon | *FT'$
$F \to id | (E)$

	First	Follow
E	id, (	$,)
E'	ε, +	$,)
T	id, (	$,), +
T'	ε, *	+, $,)
F	id, (	*, +, $,)

QN Find the first and follow of the following grammar-

$S \to aBDh$
$B \to cC$
$C \to bC | \epsilon$
$D \to EF$
$E \to g | \epsilon$
$F \to f | \epsilon$

	First	follow
S	a	$
B	c	g, f, h
C	b, ε	g, f, h
D	g, f, ε	h
E	g, ε	f, h
F	f, ε	h

QN. $S \to (L) | a$
$L \to SL'$
$L' \to \epsilon | ,SL'$

	First	Follow
S	(, a	$, ,,)
L	(, a	),
L'	ε, ,	)

LL1 Table Construction Algo.

For each production, $A \to \alpha$
repeat following two steps:-

1) Add $A \to \alpha$, under $M[A, b]$
where, $b \in First(\alpha)$

2) If First(α) contain ϵ, then
add $A \to \alpha$, under $M[A, c]$
where, $c \in Follow(A)$.

IN Construct LL(1) parsing table for the following grammar,
$S \to (L) | a$
$L \to$

LL(1) table conversion algo

for each production, $A \to \alpha$, repeat, following two steps:-

1) Add $A \to \alpha$, under $M[A, b]$
where, $b \in First(\alpha)$

2) If, First(α) Contain ϵ, then
add $A \to \alpha$, under $M[A, c]$
where, $c \in Follow(A)$.

IN Construct LL(1) parsing table, for the following grammar:-

IN $S \to (L) | a$
$L \to SL'$
$L' \to \epsilon | , SL'$

$\Longrightarrow$

$S \to (L) | a$
$L \to L, S | S$

{ actual grammar }

after removing left Recursion)

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \epsilon$		$L' \to , SL'$	

Given grammar is LL(1), because each entry of LL(1) parsing

parsing table contains maximum one entry.

QN Construct LL(1) parsing table – (or) given grammar is LL1 or not

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

Eliminate left recursion –

$E \to TE' \mid T$

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

$T \to FT' \mid F$

$T' \to \epsilon \mid *FT'$

$F \to id \mid (E)$

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

QN Check the following grammar LL(1) is not –

$S \to A$

$A \to aB \mid Ad$

$B \to b$

$C \to g$

⇓

$S \to A$

$A \to aBA'$

$A' \to \epsilon \mid dA'$

$B \to b$

$C \to g$ ⇒ LL(1) grammar

	a	b	d	g	$
S					
A					
A'			$A' \to dA'$		$A' \to \epsilon$
B					
C					

QN Check the following grammar is LL(1) or not :–

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

	a	b	$
S	$S \to AaBb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

QN. Find the grammar is LL(1) or not-

$S \to AaAb \mid BbBa$

$A \to b$

$B \to a$

⇓

LL(1)

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

Note 1:-

$\to \alpha_1 \mid \alpha_2 \mid \alpha_3$ ⇒ If this is the given form of grammar, then that grammar is said to be LL(1) only if-

$$\begin{aligned} &First(\alpha_1) \wedge First(\alpha_2) = \phi \\ &First(\alpha_1) \wedge First(\alpha_3) = \phi \\ &First(\alpha_2) \wedge First(\alpha_3) = \phi \end{aligned}$$

⇒ pairwise mutually disjoint

Note 2 If the grammar is in the form of -

$A \to \alpha_1 \mid \alpha_2 \mid \in$, Grammar is called LL(1) only if-

$$\begin{aligned} &First(\alpha_1) \wedge First(\alpha_2) = \phi \\ &First(\alpha_1) \wedge Follow(A) = \phi \\ &First(\alpha_2) \wedge Follow(A) = \phi \end{aligned}$$

⇒ pairwise mutually disjoint

N. Check the following grammar is LL(1) or not - (GATE)

$\to E \mid a$

$\to a$

Ans S ⇒ first(E) ∧ first(a)

$a \wedge a = a$ not (LL1) Ans

N. Check the following grammar is LL1 or not-

$S \to aABb$

$A \to a \mid \in$

$B \to d \mid c$

A: first(a) ∧ follow(A) → (a, ∈) ∧

B: first(d) ∧ follow(B) → d ∧ c

Given grammar is LL(1) grammar. Ans

QN. $S \to aSA \mid \epsilon$
$A \to c \mid \epsilon$

Sol^n S

first(A) ∧ follow(A)
a ∧ c, $ = ϕ

A

first(A) ∧ follow(A)
c ∧ $, c = c

This grammar is not LL(1) grammar. Ans

QN. Check the following grammar is LL(1) or not,
$S \to AB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$

Sol^n A

first(A) ∧ follow(A)
a ∧ b, $ = ϕ

B

first(b) ∧ follow(b)
b ∧ $ = ϕ

This grammar is LL(1) grammar. Ans

QN. Given grammar is LL(1) or not-
$S \to (L) \mid a$
$L \to L,S \mid S$

Sol^n S

first(a) ∧ follow(S)
(∧ a = ϕ

Note:- Any left recursive grammar is not LL(1).

Note-2:- Any grammar which contain left factoring is not LL(1)

QN. Construct LL(1) parsing table for the following grammar:
begram → begin ① d Semi X end
X → d Semi X ② | S Y ③
Y → Semi S Y ④ | ϵ ⑤

	Semi	begin	d	end	S	$
program		①				
X			②		③	
Y	④			⑤		

⇒ LL(1) grammar

10. Find first and follow for the following grammar-

$E \to aA \mid (E)$

$A \to +E \mid *E \mid \epsilon$

Sol^n

	first	follow
E	a, (	), $
A	+, *, ε	), $

11. Which one of the following is true :-

$E \to E*F \mid E+F \mid F$

$F \to F-F \mid id$

1) * has higher precedance than +.

2) ✓ *

3) *, - has same precedance

4) + has higher precedance.

12. ϕ

Recursive Decent Parsers

Exp:- $E \to E+T \mid T$

$T \to T*F \mid F$ ⟹ Eliminate left Recursion

$F \to id$

$E \to TE'$

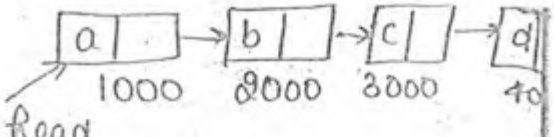
$E' \to \epsilon \mid +TE'$

$T \to FT'$

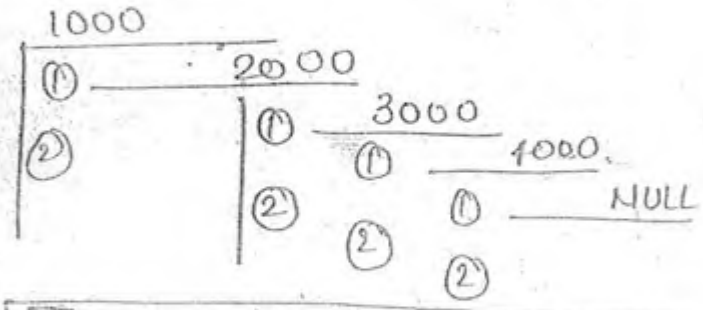
$T' \to \epsilon \mid *FT'$

$F \to id$

```
E()          E'()          T()          T'()
{            {             {            {
  E()          E'()          T()          T'()
}            }             }            }
```

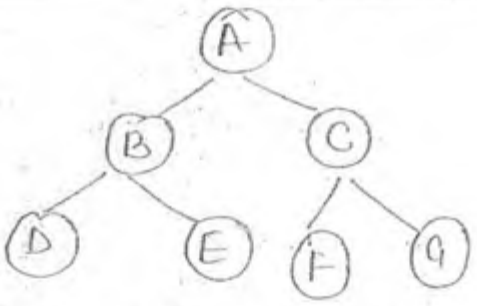



```
Read
Return(Head)
{
  if(Head==Null)
    return;
  else
  {
    (1) Reverse(Head->next)
    (2) printf(Head->data);
  }
}
```

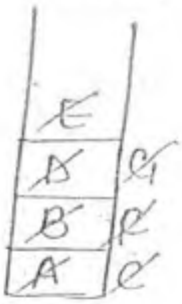


Non Recursive pgm → stack at the place of recursion

Preorder without Recursion



Stack



A, B, D, E, C, F, G

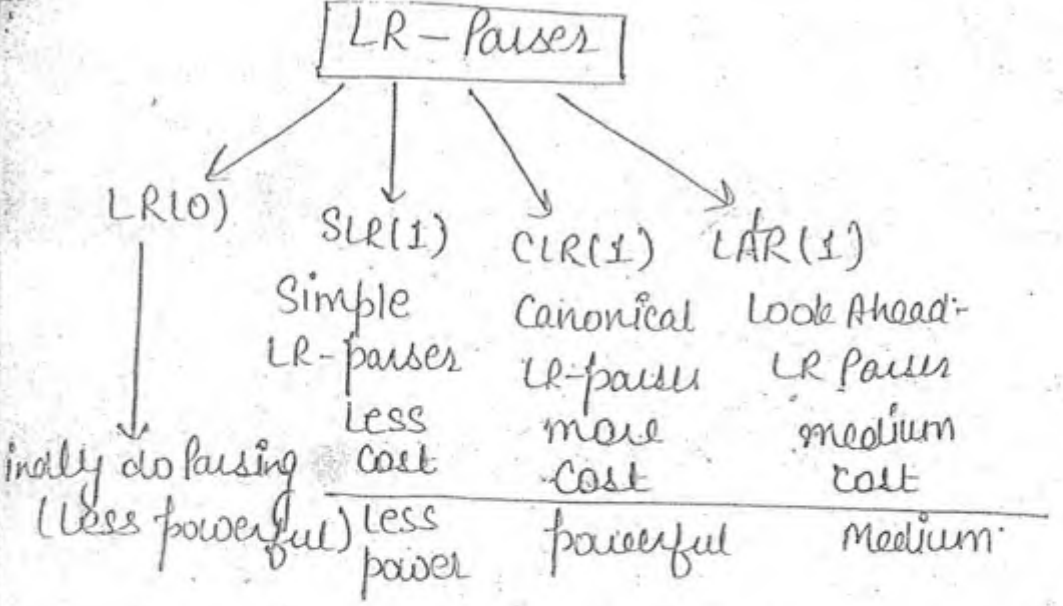
Note:- In recursive decent paser, we will write the recursive pgm. for every variable, such that if any variable contains more than one possibility, it will choose the correct productio

Bottom-Up Paser

Bottom-up Paser (Shift-Reduce Pasers)

- Operator Precedance Paser
  - Operator Grammar ⇐ Ambiguous, Unambiguous
- LR-Paser
  - applied only on unambiguous grammar

LR-Parsers



* Bottom-up parsers are much powerful than top-down parsers, but designing is more complex.

$$LL(K) \subseteq LR(K)$$

Operator Precedance Grammar (Parsers)

Operator Grammar :-

ambiguous , unambiguous

Operator Precedance Parser

Grammar is said to be a operator grammar, if it satisfies following two conditions-

- No null productions
- No two variables side by side on R.H.S. of production

Exp:- $E \to E+E \mid E*E \mid id$ ✓

Exp:- $E \to A+B$
$A \to a$
$B \to b \mid \in \to \alpha$

Exp:- $E \to AB$
$A \to a$ α
$B \to b$

Check the following grammar is operator grammar or not

$S \to SAS \mid a$

$A \to bSb \mid b$

This grammar is not operator grammar.

conversion into Operator grammar

1) $S \to SbSbS \mid SbS \mid a$ $\Rightarrow$ $S \to SbSbS \mid SbS \mid a$

$A \to bSb \mid b$ remove A, useless production

2) $P \to SR \mid S$ $\Rightarrow$ $P \to SbP \mid SbS \mid S$

$R \to bSR \mid bS$ $R \to bP \mid bS$

$S \to WbS \mid W$ $S \to WbS \mid W$

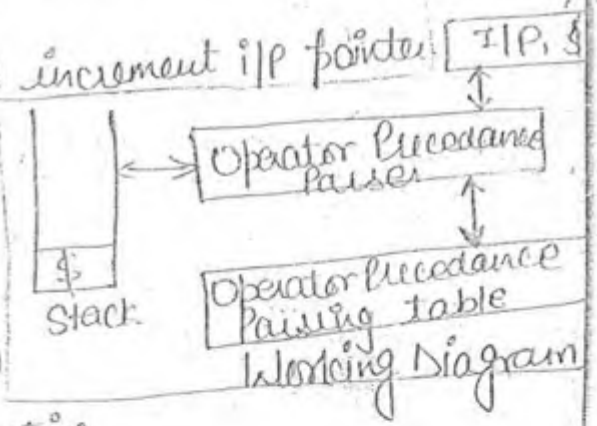
$W \to L*W \mid L$ $W \to L*W \mid L$

$L \to id$ $L \to id$

Operator Precedence Parser (Parsing algorithm):-

If a is the top of the stack, then b is the look ahead symbol then-

1) If $a < b$ or $a = b$, then shift b and increment i/p pointer
2) If $a > b$, repeat-
   {
   pop out from the stack
   } until
   (top < recently popped out)
3) If $a = b = \$$, then successful completion.



Ex:- Parsing table

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

① $w = id + id * id \$$

$ id + id * id

id id id * + A#

② $w = id * id * id \$$

$ id * id * id

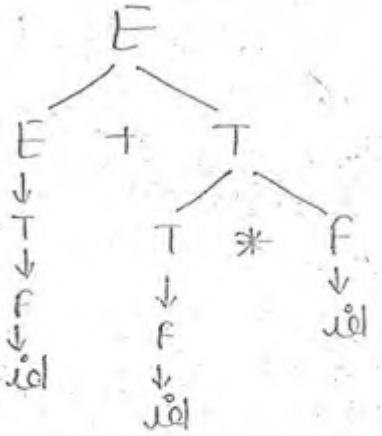
id id * id * A#

E → E+T | T
T → T*F | F
F → id

$ < id > + < id > * < id > $

+ < *
+ > +
* > *

= 5 Handle



$	id	*	id	*	id	

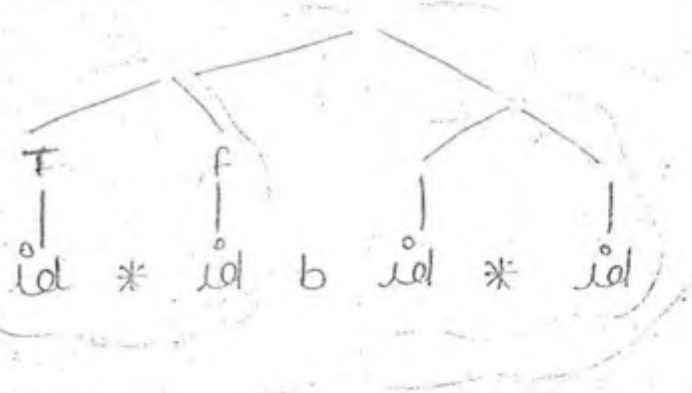
id → ① Handle * → 4 Handle
id → ② Handle + → 5 Handle
id → ③ Handle

Q. Define the Operator precedance parsing table for, w=

w = id * id b id * id

Operators ⇒ *, b and * > b b < b

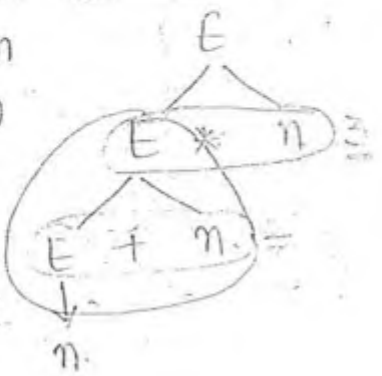
	id	*	b	$
id		>	>	>
*	<	>	>	>
b	<	<	<	>
$	<	<	<	



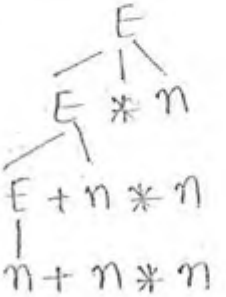
Q. Consider the grammar-
E → E+n | E*n | n, for the I/P string,
w = n+n*n, the handels are :-

1) n, E+n and E+n*n
2) n, E+n and E+E*n
3) n, n+n and n+n*n
4) n, E+n and E*n

Parse Tree



1st Handle = n



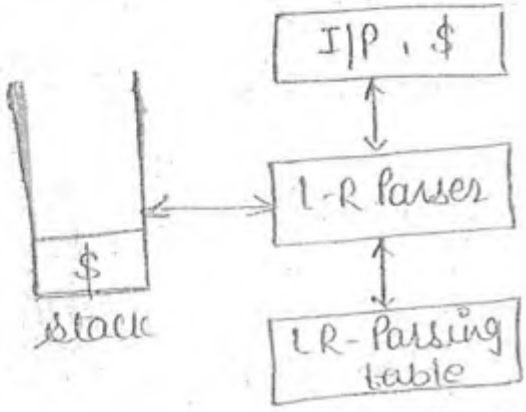
L-R Parsing Algo *

If S- state on the top of the stack and a- lookahead symbol, then-

1. If action [S, a] = Si, then shift a and i and increment the i/p pointer.
2. If action [S, a] = Rj, and if Rj is, $\alpha \rightarrow \beta$, then pop $2*|\beta|$ sym from the stack and replaced by α.

If Sm-1 is the state below α, then push Goto [Sm-1, α].

3. If action [S, a] = acc, then successful completion.



Note:- For all LR-parsers, parsing algorithm is same, parsing tables are different.

Exp:- S → AA ①
A → aA ② | b ③

w = aabb$

LR-(0) Parsing table

	a	b	$	S	A
	action			Goto	
0	S3	S4		1	2
1			acc		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

Rows = State No.
Columns = Action | Goto

$	0	a	3	a	3	b	4

a a b b $

⇒ acc = accepted

Ex: w = abab$

$	0	a	3	b	4	b	4

S 1 = acc = accepted

r_3 A → b, |b| = 1*2 = 2

r_2 A → |aA| = 2*5 = 4

r_1 S → AA = 2*2 = 4

r_3 A → b, |b| = 1*2 = 2

Problem with Shift → Reduce ⇒ Action part

↳ because these entries are only on action part

Another parsers other than LR(0), has some minimization in reduction entries, because they can predict the next symbol.

Conflict:- Shift and reduce together S_j/r_i

Ex: S → AA

A → aA | b

(added to know successful completion)

↓↓ Augmented Grammar

S' → S ⇒ S·S

S → AA

A → aA | b

$ Item

S → ·xyz (Item)

S → x·yz

S → xy·z

S → xyz· ⇒ (final production / Complete production / reduced production)

Closure of an item

↓↓

applied only on item

Given Grammar

Ex:- S → AA

A → aA | b

Now, S' → ·S ⇒ item (closure)

(i) S → A·A ⇒ closure (S)

① S' → ·S

② S → ·AA

③ A → ·aA

· b

S → A·A

A → ·aA

· b

ϕ Finding closure of a variable

Closure (I) =

① Add item I to closure(I)

② If $A \to \beta . B\alpha$ is I and $B \to \gamma$ in G, then add $B \to .\gamma$ to closure of

③ repeat previous two steps. for every newly added production

Exp:- $S \to AA$
$A \to aA | b$

① $S \to .S$

② $S \to .AA$

③ $A \to .aA$
$.b$

ϕ Finding Goto (I, X) :-

Find the Goto of (I, X) by following method :-

Write the same item (I) as it is, by moving (.) after X.

Apply closure function on the result of step 1.

ϕ LR(0) Parsing table Construction

Step-1 Find the augmented grammar.

Step-2 Find the closure of augmented production

Step-3 Using closure of augmented production construct the dfa.

Step-4 Reduce dfa into table.

Exp:- Construct LR(0) Parsing table, for the following grammar:-

$S \to AA$ ①

$A \to aA$ ② $| b$ ③

Sol"

① $S' \to S$
$S \to AA$
$A \to aA | b$

② I_0: $S' \to .S$, $S \to .AA$, $A \to .aA$, $A \to .b$

I_0 on S → I_1: $S' \to S.$

I_0 on A → I_2: $S \to A.A$, $A \to .aA$, $A \to .b$

I_2 on A → I_5: $S \to AA.$

I_0 on a → I_3: $A \to a.A$, $A \to .aA$, $A \to .b$

I_3 on a → I_3

I_3 on b → I_4: $A \to b.$

I_3 on A → I_6: $A \to aA.$

I_4 on a → I_4

I_0 on b → I_4

Construction of table

M	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			acc		
2	S3	S4			5
3	S3	S4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

* If in a Ij more than one production is completed, it will become Reduce-Reduce Conflict

* Shift-Reduce Conflict = Reduce-Shift Conflict

* No Shift-Shift Conflict ⇓ – because it is a d.f.a.

⇒ given grammar is LR(0) grammar, because no entry of this table contain more than one value AS

Note:- S-S Conflict is not possible, because given diagram is d.f.a.

Q. Check the following grammar is LR(0) or not-

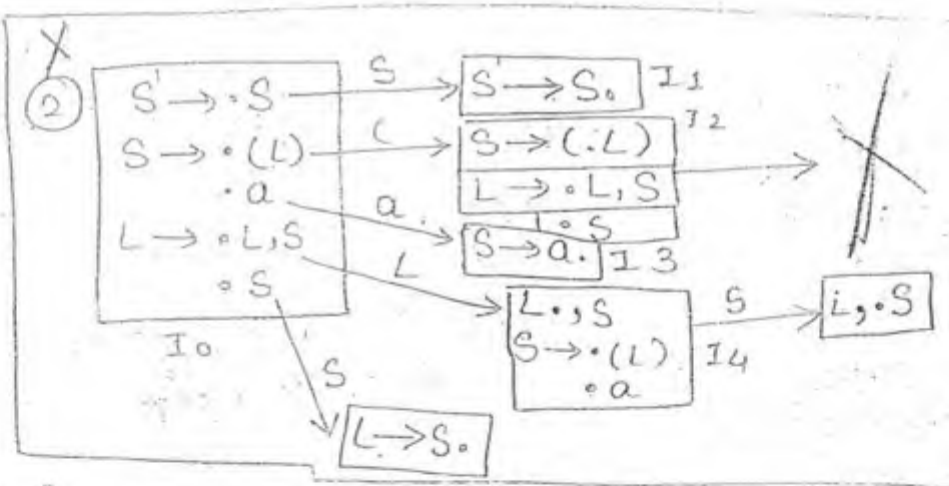
G → (L) | a

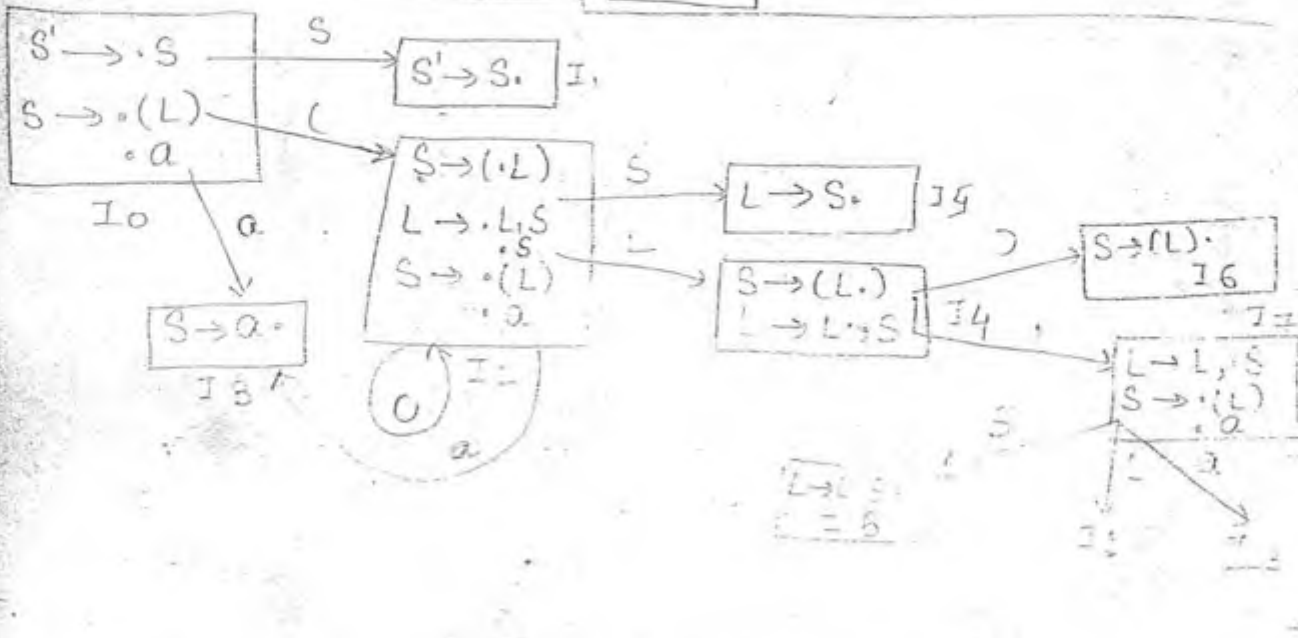
L → L, S | S

(1) S' → S

S → (L) | a

L → L, S | S





Parsing table

M	Action					Goto	
	a	(	)	,	$	S	L
0	S3	S2				1	
1					acc		
2	S3	S2				5	4
3	r2	r2	r2	r2	r2		
4				S6	S7		
5	r4	r4	r4	r4	r4		
6	r3	r1	r1	r1	r1		
7	S3	S2				8	
8	r3	r3	r3	r3	r3		

⇒ LR(0)

Q.N. Check the following grammar is LR or not:-

S → dA | aB

A → bA | c

B → bB | c

Sol^n

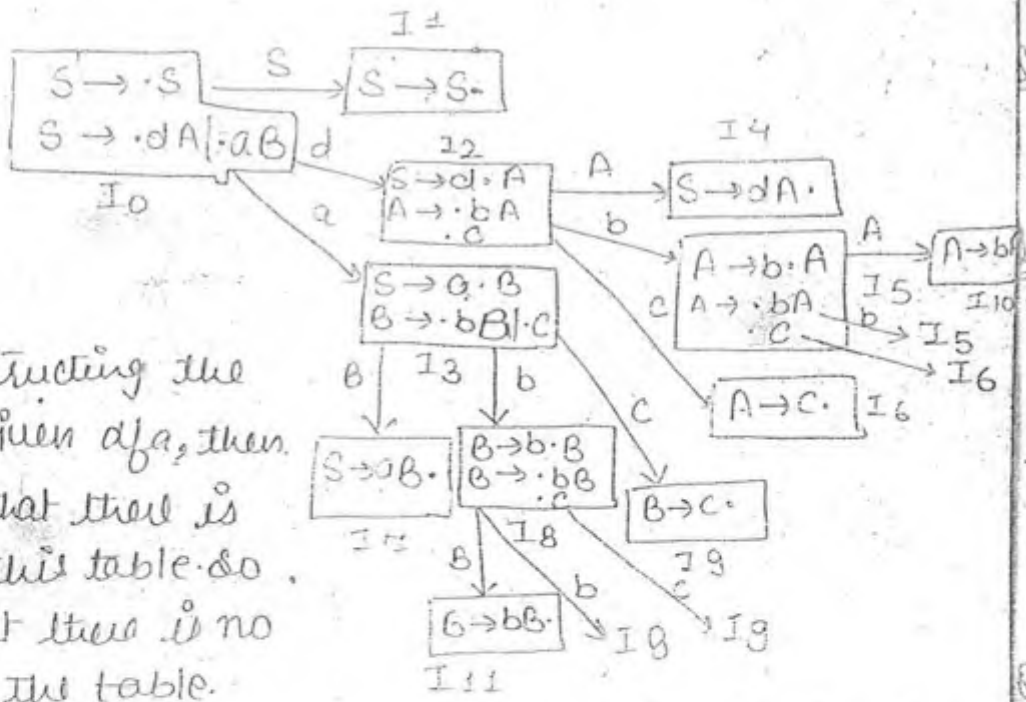
S' → S

S → dA | aB

A → bA | c

B → bB | c

⇓



If we are constructing the table, for the given dfa, then we will find, that there is no conflict in this table. So, it is clear, that there is no any conflict in the table.

⇒ There is no conflict in this given Grammar. So it is LR(0) Grammar.

States = 0 · 11

Action = a, b, c, $

Goto → S, A, B, $

Q. Check the following grammar is LR(0) or not:-

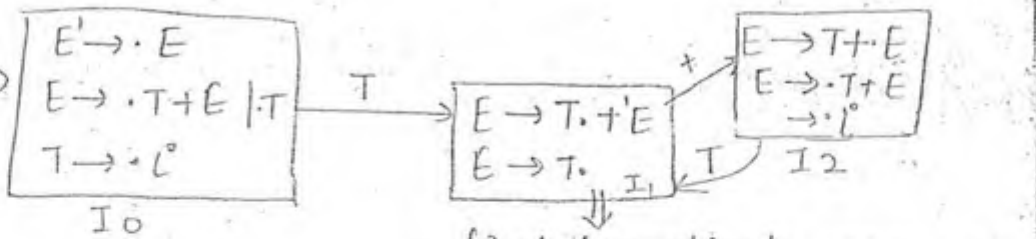
$E \to T+E \mid T$

$T \to i$

$E' \to E$

$E \to T+E \mid T$

$T \to i$



final & Non final ⇒ SR conflict

Not LR(0) Ans

Exp:- $S \to A \cdot B$, $A \to b \cdot$ ⇒ final and non final but still conflict not

bcoz $S \to A \cdot B$ (reduction part) not ⇓ Go to part

$S \to b \cdot$ (purely action part)

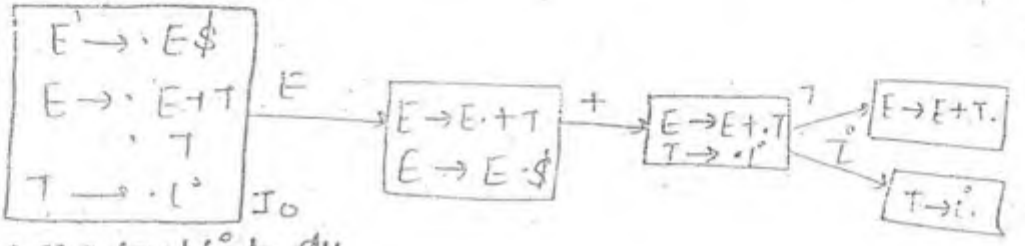
Q. $E \to E+T \mid T$

$T \to i$

$E' \to E\$$

$E \to E+T \mid T$

$T \to i$



It is LR(0) → no conflict Ans

Q. following grammar is LR(0) or not:-

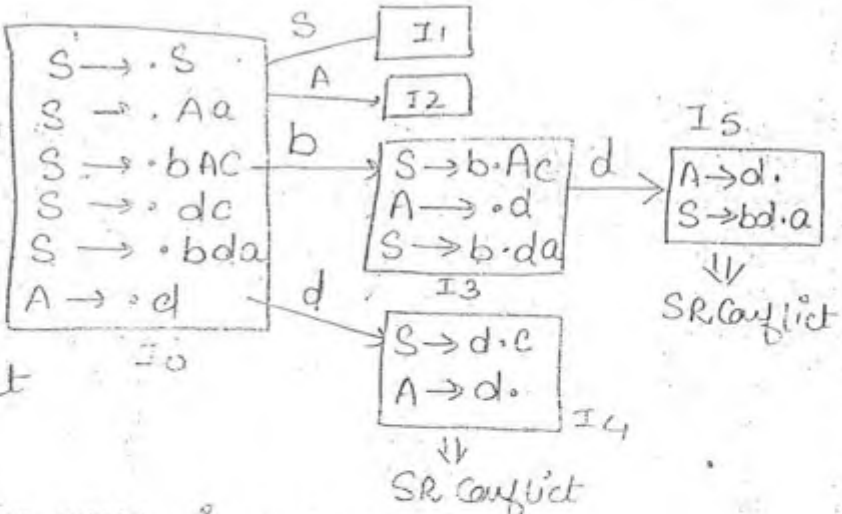
$S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$

Sol^n $S' \to S$

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

$A \to d$



This grammar is not LR(0) because two states are not acceptable.

Q. Check the following grammar is SLR(1) or not:-

Sol^n $S \to Aa \mid bAC \mid dc \mid bda$

$A \to d$

given grammar is not SLR(1) because, the conflicts which are there in LR(0) are not eliminated by SLR(1).

IN Check the following grammar is SLR(1) or not-

$E \to E+T \mid T$

$T \to TF \mid F$

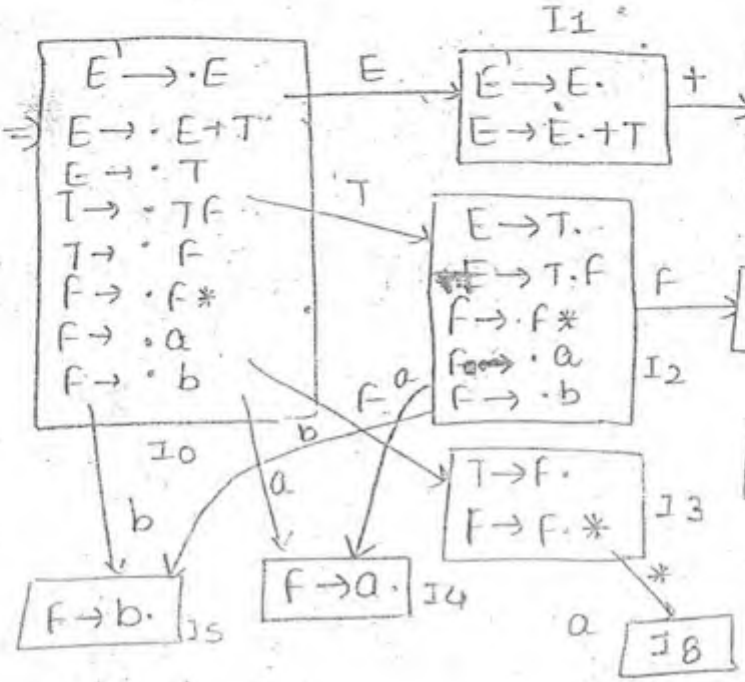
$F \to F* \mid a \mid b$

Sol" $E' \to E$

$E \to E+T \mid T$

$T \to TF \mid F$

$F \to F* \mid a \mid b$



	a	b	*	+	$
2	r2/s4	r2/s5	r2	r2	r2
3	r4	r4	r4/s8	r4	r4
9	r1/s4	r1/s5	r1	r1	r1
7	r3	r3	r3/s10	r3	r3

⇒ not LR(0), just bcoz of conflicts

Now removing conflicts, apply SLR(1). Find follows of the conflict entries and intersection with shift entries If we got ϕ = no conflict and LR(1)

SLR(1) table

	a	b	*	+	$
2	Su	S5	r2	r2	r2
3	r	r	S	r	r
9	S	S		r	r
7	r	r	S	r	r

I2 = R = + $ / S = a, b / ϕ

I3, R = + $ a, b / S = * / ϕ

I7 + $, a, b / * / ϕ

I9, + , $ / a, b / ?

= SLR(1) ⇒ all the conflicts of LR(0) are removed here.

1* Check the following grammar is SLR(1) or not:-

$S \to AaAb \mid BbBa$

$A \to \epsilon$

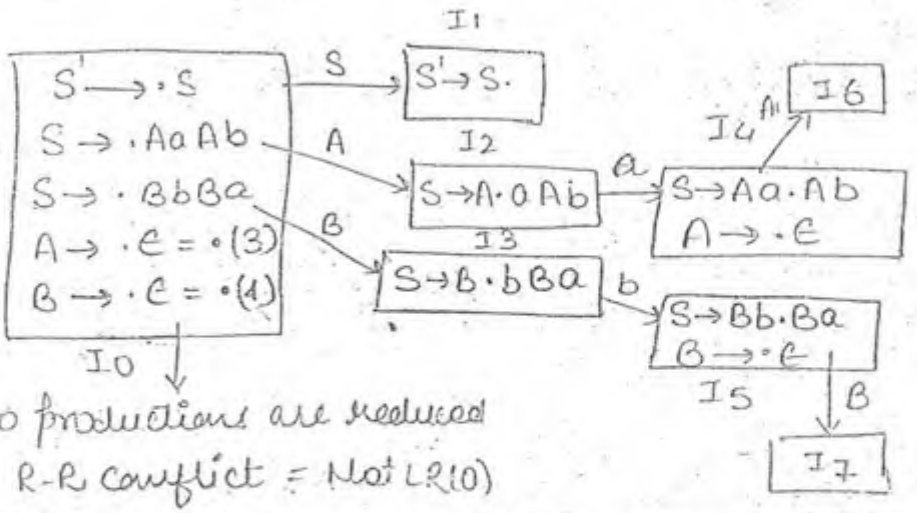
$B \to \epsilon$

Sol^n $S' \to S$

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$



Two productions are reduced so R-R conflict = Not LR(0)

Given grammar is not LR(0) because LR(0) is faulty state which will contain R-R conflict

→ Check for SLR(1)

follow(A) = b, a

follow(B) = a, b

	a	b
0	r3/r4	r3/r4

⇒ not SLR(1)

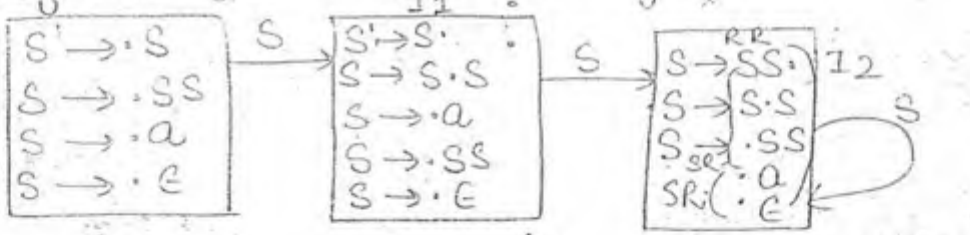
2 Consider the following grammar-

$S \to SS \mid a \mid \epsilon$

Find the number of inadequate state in LR(1) grammar

Sol^n $S' \to S$

$S \to SS \mid a \mid \epsilon$



I0 S-R Conflict

S-R Conflict

SR, RR → Conflict

	a	b
2	r1/r3	r1/r3
2	r1/s3	r1
2	r3/s3	r3

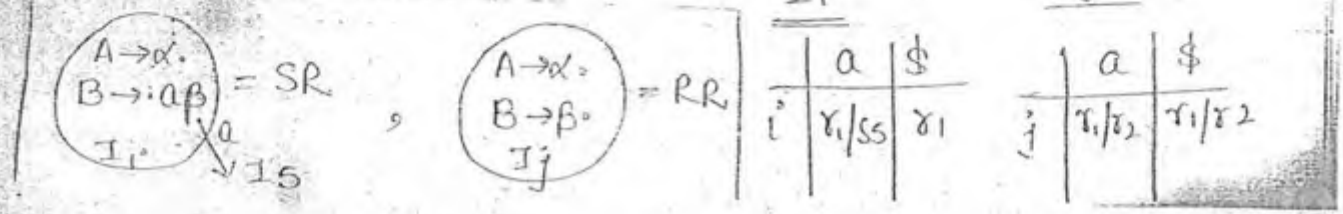
4 - SR

1 - RR

(3) → I.S.

Φ CLR(1) Grammar:-

Conflicts in LR(0)



I_1°

	a	$
i	r1/s5	r1

I_j°

	a	$
j	r1/r2	r1/r2

LR(0) → item

LR(0) | SLR(1)

⇓

LR(1) item

⇓

LR(0) item + 1 - look ahead symbol.

LR(1) item

CLR(1) | LALR(1)

DN. $S \to AA$

$A \to aA \mid b$

⇓

$S' \to .S, \$$

closure $(S' \to .S, \$)$ =

(1) $S' \to .S, \$$

(2) $S \to .AA, \$$

(3) $A \to .aA, a/b$

$.b, a/b$

Ⓟ Find the closure of LR(1) item

Closure of Item (I) =

1) Add the same LR(1) item. $(S' \to .S, \$)$

2) If $A \to \alpha.B(\beta, \$)$ is LR(1) item I,

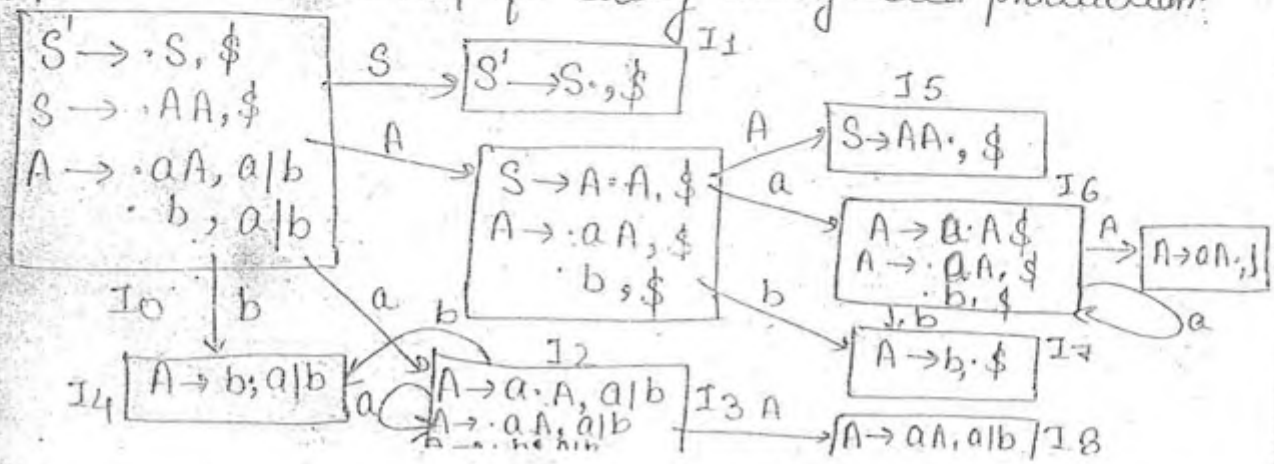
and $B \to C$ is in G.

then

add $B \to .C$, first $(\beta, \$)$ to closure of LR(1) item I.

3) Repeat the second step for every newly added production.

Soln



The given grammar is CLR(1), because no state contain conflicts.

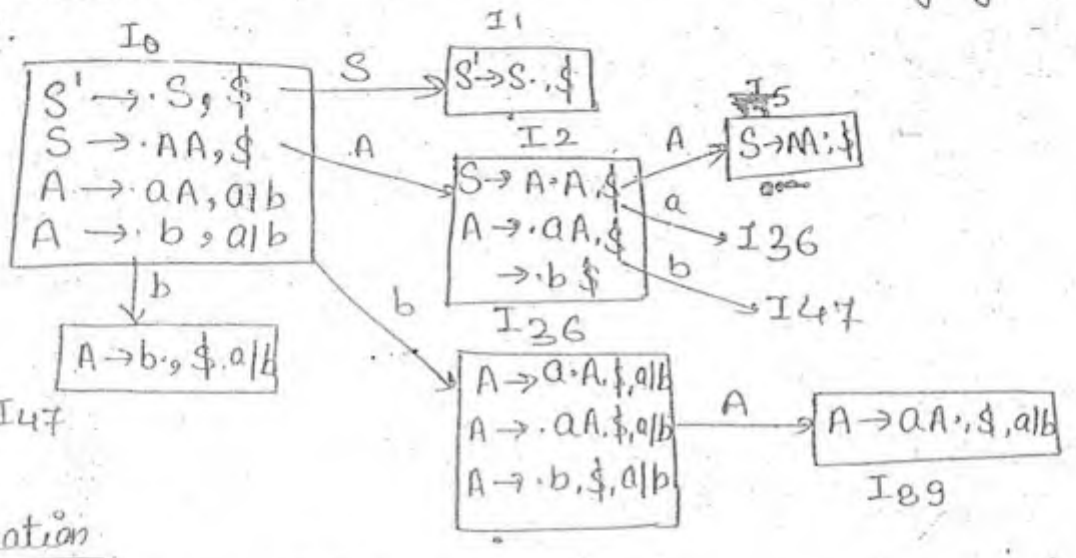
Relation b/w the states of LR(0), SLR(1), CLR(1)-

LR(0)	CLR(1)	SLR(1)	LALR(1)
⇓	⇓	⇓	⇓
n_1	n_3	n_2	n_4

$n_1 = n_2 \leq n_3$ $\qquad$ $n_1 = n_2 = n_4 \leq n_3$

LASLR(1) Parsers

It will combine the two states, which are differe only by lookahead symbol.



minimization

Q1. Check the following grammar is CLR(1) or not -

S → AaAb | BbBa

A → ∈

B → ∈

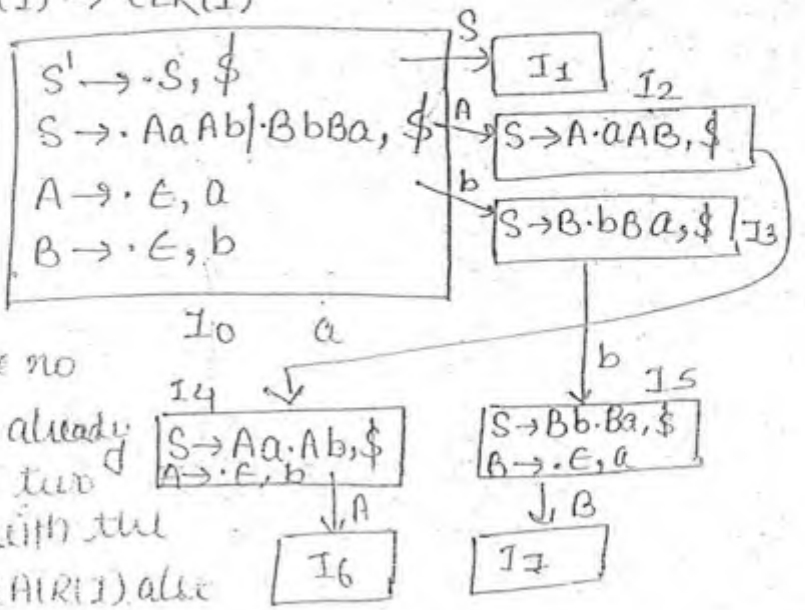
Soln LL(1) → LALR(1) → CLR(1)

S' → S

S → AaAb | BbBa ⇒

A → ∈

B → ∈



The given grammar is CLR(1) grammar because no conflicts. The given dfa is already minimized, because no two states are differed with the lookahead symbol (LALR(1)) also.

1) Check the following grammar is CLR(1) or not →

$S \to AA / bAC / dc / bda$

$A \to d$

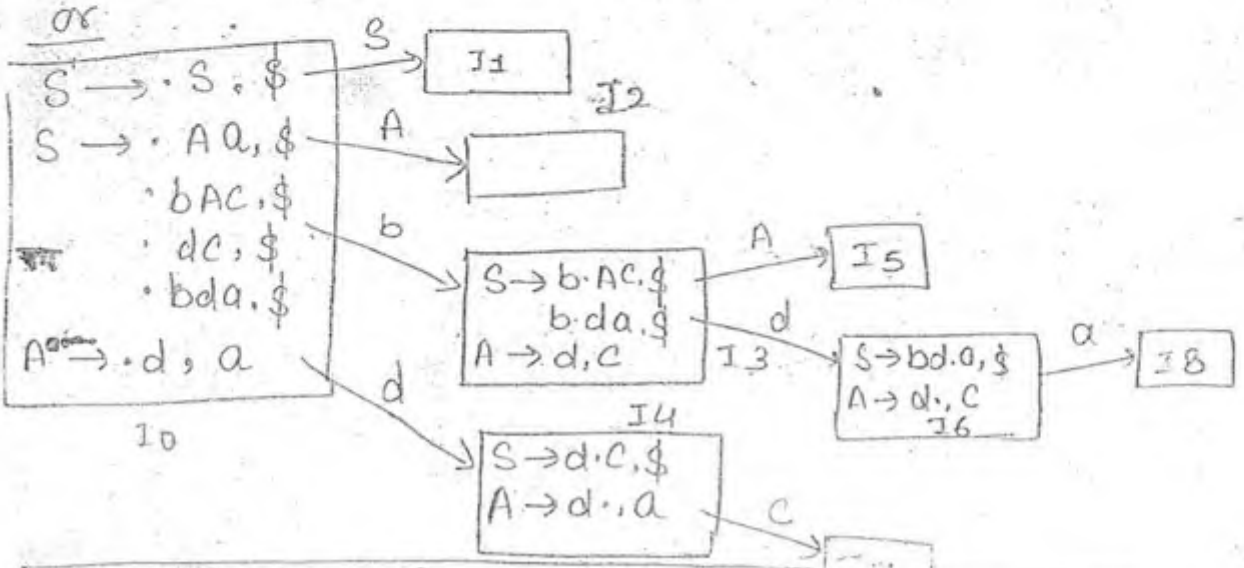
Sol: $S' \to S$

$\to AA / bAC / dc / bda \Rightarrow S' \to \cdot S, \$$

$S \to \cdot AA / \cdot bAC / \cdot dc / \cdot bda, \$$

$A \to \cdot d, d$

or



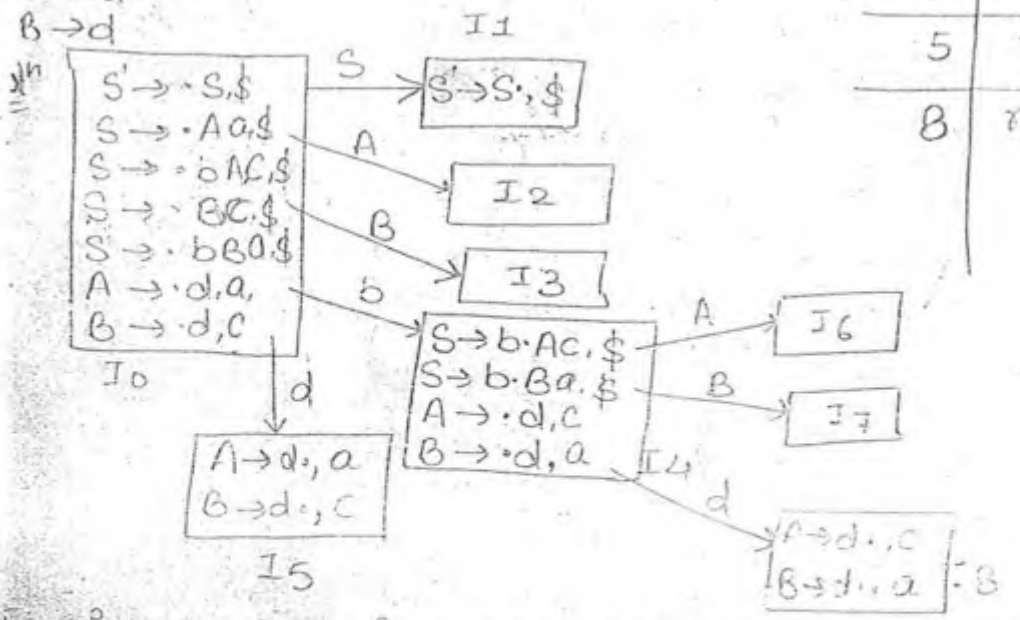
→ no conflict (CLR(1)) = LALR(1) = already minimized. du

2) Check the following grammar is LALR(1) or not.

$S \to Aa / bAc / Bc / bBa$

$A \to d$

$B \to d$



	a	c
5	r5	r6
8	r6	r5

LALR(1)

I58

A → d · , a/c

A → d · , a/c

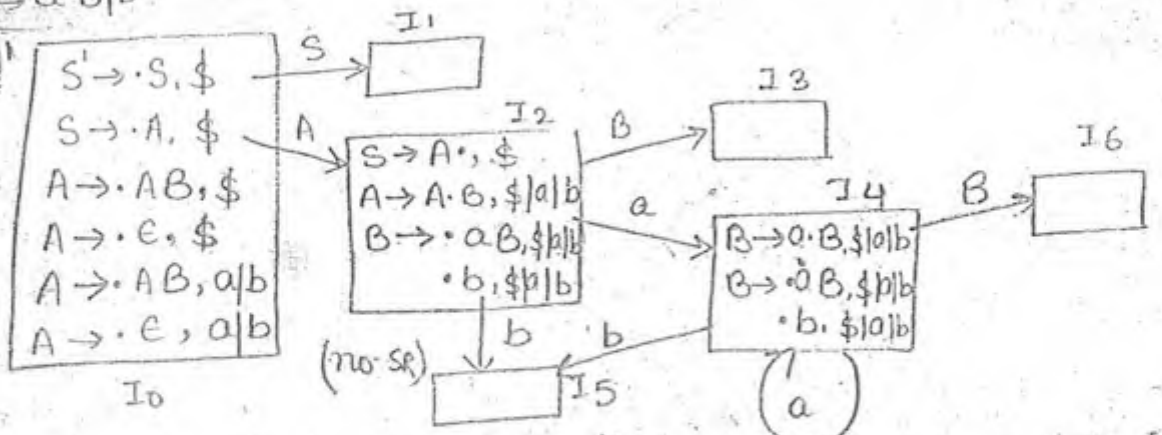
	a	c
58	r5 , r5	r5 , r6

The given grammar is CLR(1), because of no conflicts The given gram is not LALR(1), because after combining states 5 and 8 then is conflict

Q1 Check the following grammar is LALR(1) or not-

$S \to A$
$A \to AB \mid \epsilon$
$B \to aB \mid b$

Sol'n

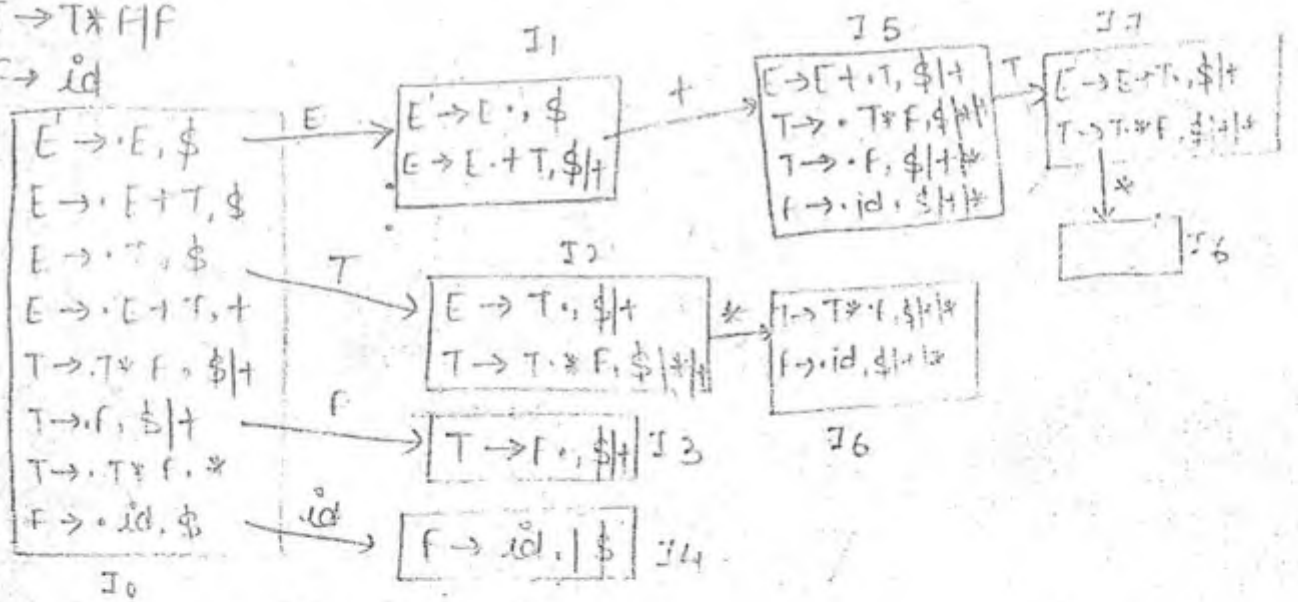


Given grammar is CLR(1). It is LALR(1) also, because it is already minimized.

Q2 Check the following grammar is LALR(1) or not-

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

Sol'n



It is CLR(1). It is LALR(1) also

GATE QUESTIONS

Q1 Consider SLR(1) and LALR(1) table for the given CFG:-
Which of the following is true-

(A) GO TO of both the tables may be different
(B) Shift entries are identical in both the tables
(C) Reduce entries in both the tables may be different
(D) None of above.

1.2 Let, SLR(1) parsing table will take n_1 rows, LALR(1) will take n_2 rows, relation b/w n_1 and n_2 :-

$$n_1 = = n_2$$ Ans

a) n_1
b) n_1
c) n
d) n_1

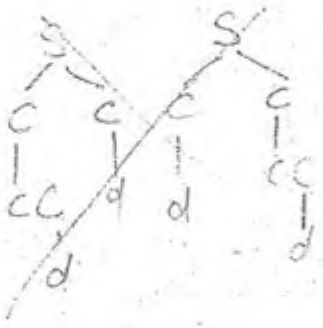
1.3 Consider the following CFG-

$S \to CC$
$C \to cC \mid d$

on which one of the following is true:-

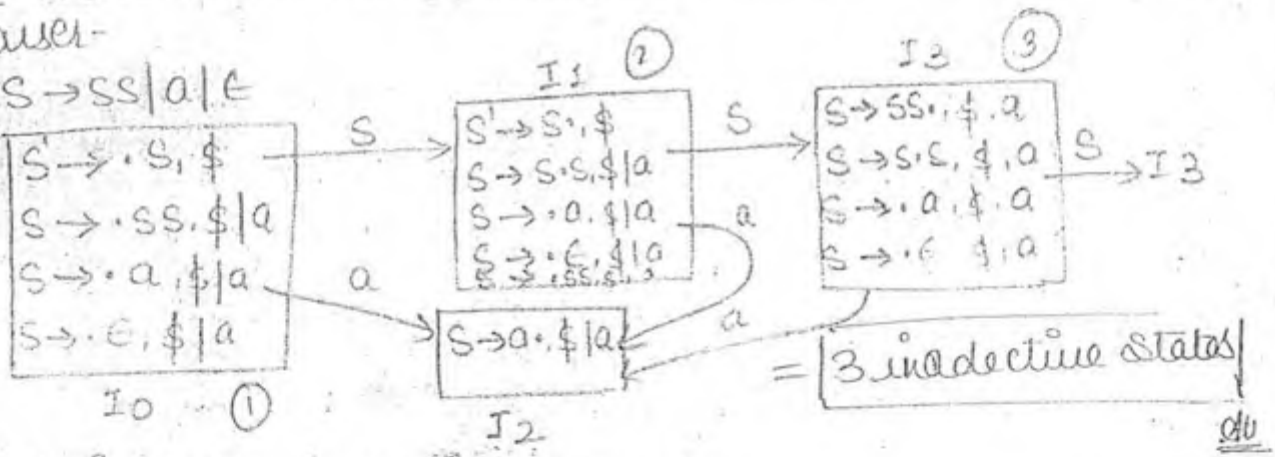
LL(1)
SLR(1) not LL(1)
LALR(1) not SLR(1)
CLR(1) not LALR(1).

LL(1) → surely LALR(1)
⇓
have to check for LALR(1)



1.4 Find the number of inadequate states while constructing CLR(1) parser-

$S \to SS \mid a \mid \epsilon$



1.5 Consider the following grammar:-

$A \to AA \mid (A) \mid \epsilon$ is not suitable for operator precedance parsing because

ambiguous
left recursion
right recursion
none of the above (operator Grammar)

1.6 Consider the following grammar-

$S \to (S) \mid a$

SLR(1) → n_1
LR(1) → n_2
LALR(1) → n_3

$$n_1 = n_3 \leq n_2$$

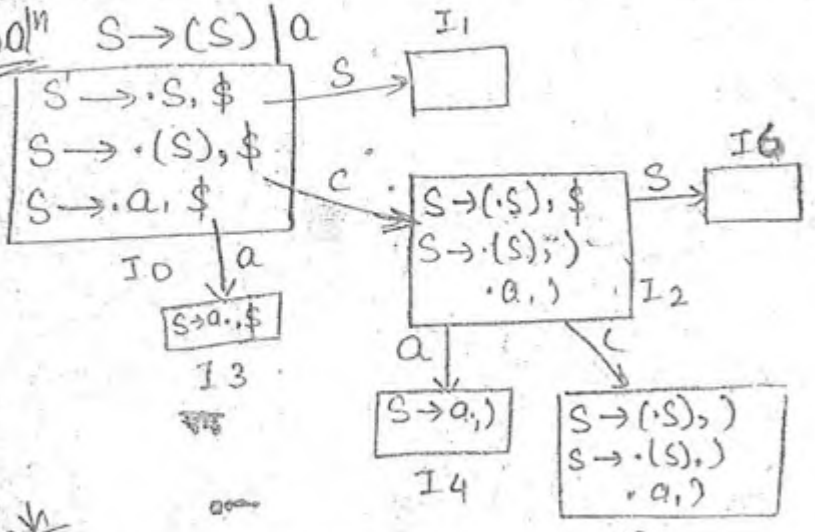
a) $n_1 < n_2 < n_3$

b) $n_1 = n_3 < n_2$

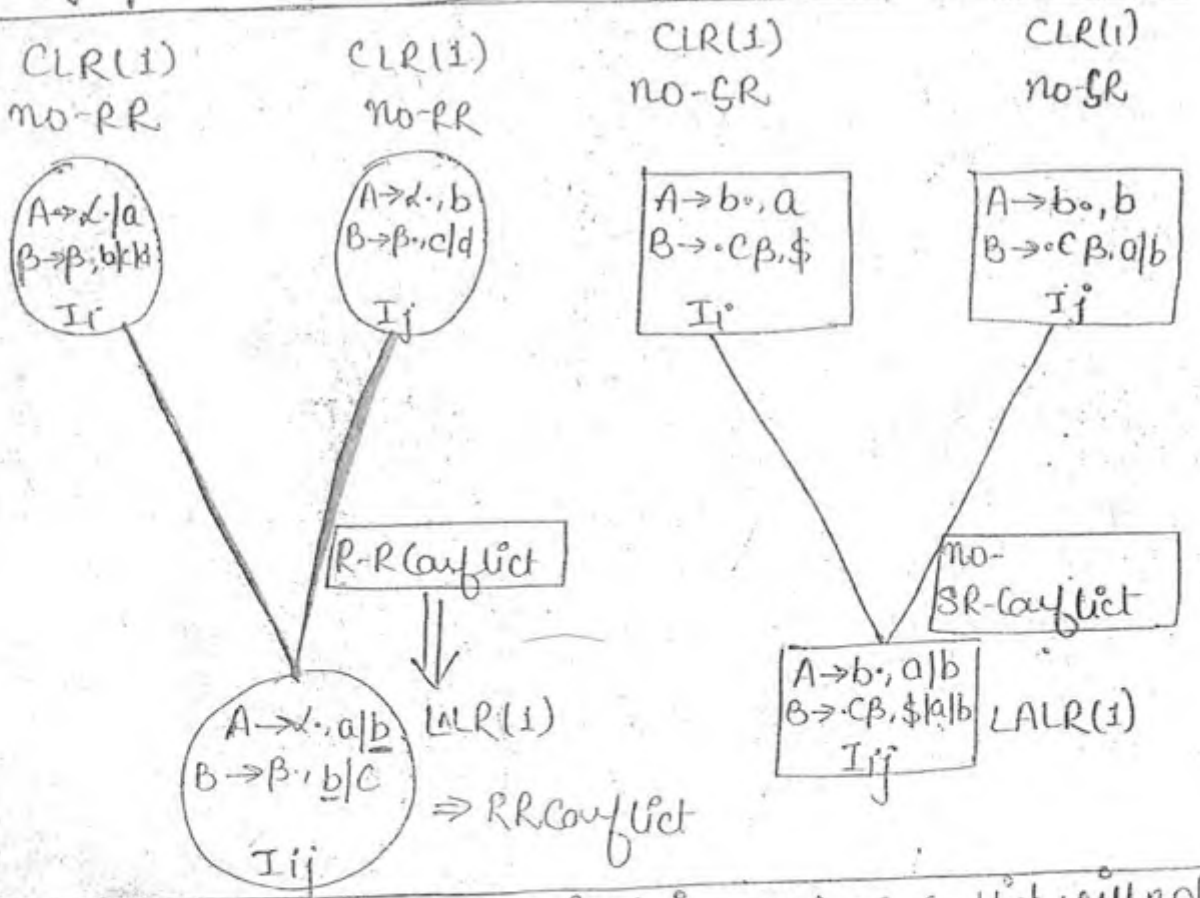
c) $n_1 = n_2 = n_3$

d) $n_1 \geq n_3 \geq n_2$

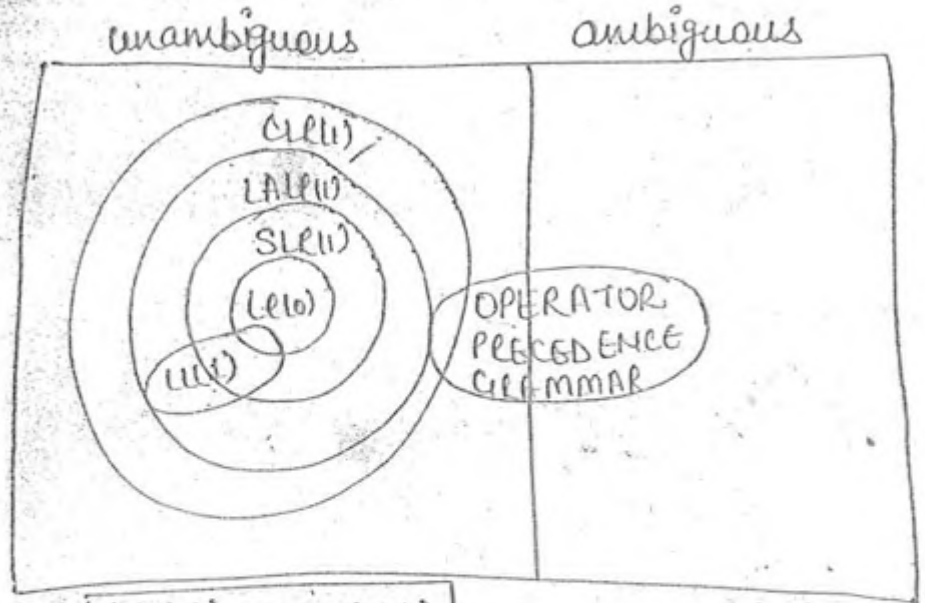
Soln $S \to (S) \mid a$



Note 1:- Even though there are no R-R conflict in CLR(1), still RR conflict may present in LALR(1).



Note 2:- If there are no SR-conflicts in CLR(1), SR-conflict will not present in LALR(1) also.



$LL(1) \subseteq LALR(1)$

$LL(1) \subsetneq LR(1)$

$LL(k) \subseteq LR(k)$

$S \to a/ab$: it is LALR(1) not LL(1)

Note-1

① Bottom up parsers are more complex to design as compared to T.D.P

② Bottom up parser is accepting more no of grammars comparing with the top down parser.

③ Size of Bottom Up Parser table = 2 * top down Parser table size

Dated 9.Dec.10

Chapter No.3

SYNTAX DIRECTED TRANSLATION (SDT)

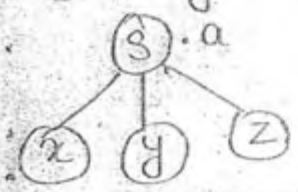
G + Semantic rules (or) Translation rules ⟹ SDT

Syntax tree or Parse tree

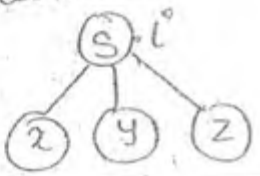
Attribute

- Synthesized attributes
- Inherited attribute

$S \to xyz$



$S.a = f(x.a | y.a | z.a)$



$y.i = f(x.i | S.i | z.i)$

Applications of syntax directed translations

* Converting the given infix expression to postfix expression.
* evaluating the given infix expression.
* Binary to decimal conversion.
* Creating syntax tree
* Creating directed acyclic graph
* To generate intermediate code
* Storing the data into symbol table

Construct Syntax Directed Translation (SDT) to convert the given infix expression to postfix expression

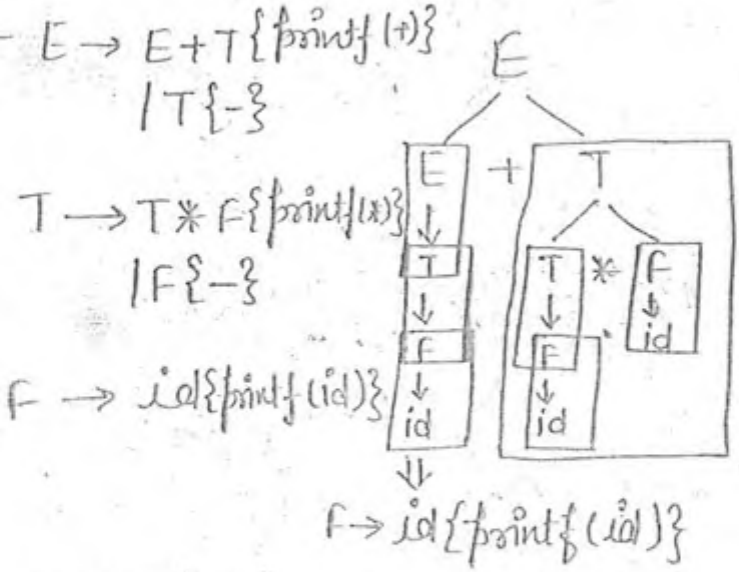
I/P: 2+3*4

O/P: 234*+

Grammar:— $E \rightarrow E+T$ {printf(+)}
$|T$ {-}

$T \rightarrow T*F$ {printf(*)}
$|F$ {-}

$F \rightarrow id$ {printf(id)}



$F \rightarrow id$ {printf(id)}

Q1 Construct SDT to find out no of reductions to evaluate the given infix expression:-

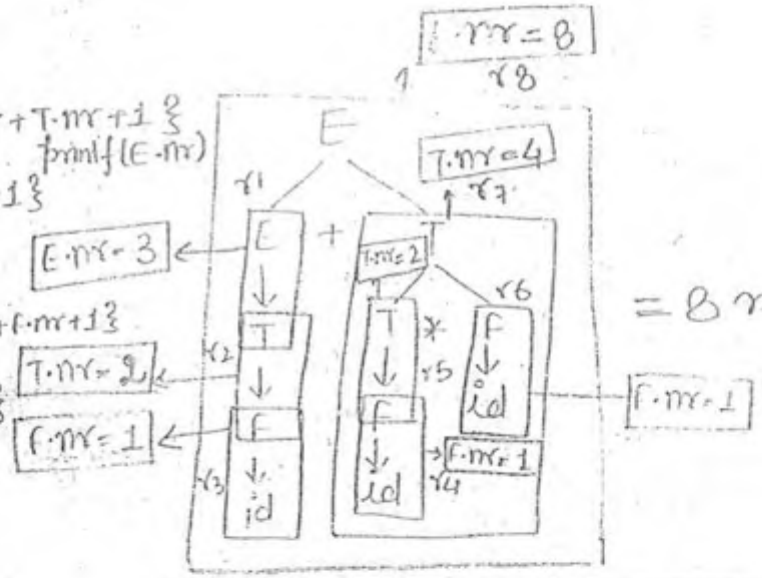
I/P = 2+3*4

O/P = 8

$E \rightarrow E+T$ {E.nr = E.nr + T.nr + 1 printf(E.nr)}
$|T$ {E.nr = T.nr + 1}

$T \rightarrow T*F$ {T.nr = T.nr + F.nr + 1}
$|F$ {T.nr = F.nr + 1}

$F \rightarrow id$ {F.nr = 1}



= 8 reductions of

attribute = nr = synthesized attribute

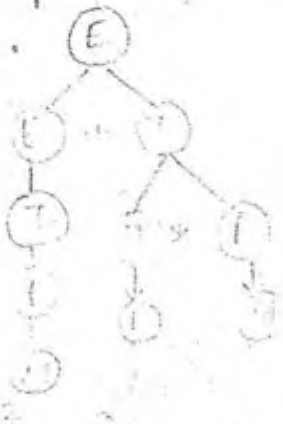
Q1 Construct SDT to evaluate the given infix expression-

I/P = 2+3*4

O/P = 14

$E \rightarrow E+T$ { printf(E.val + T.val) }
$|T$ { E.val = T.val }

$T \rightarrow T*F$ { T.val = T.val * F.val }
$|F$ { T.val = F.val }



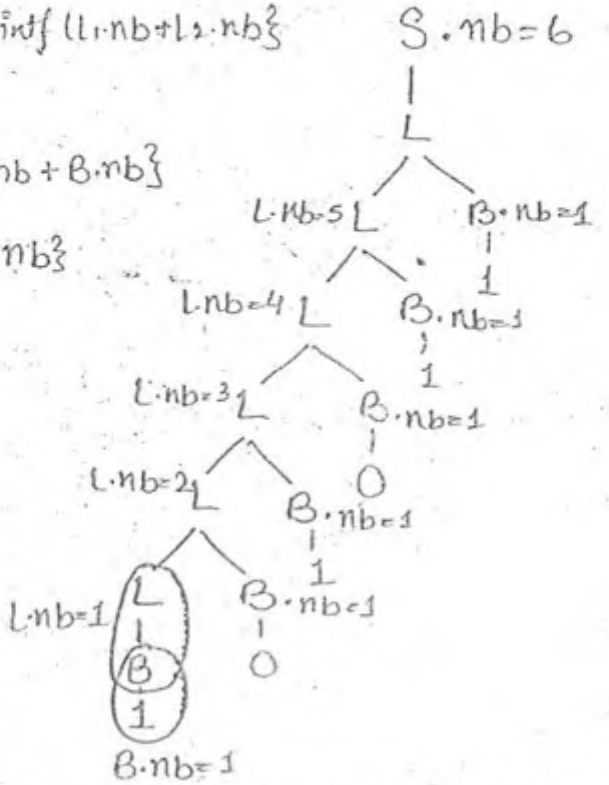
11. Construct SDT to find the dits in the given binary numbers.

i/p I/P: 101011 | 1011.111
O/P: 6 | O/P = 7

$S \rightarrow L \mid L.L \{printf(L_1.nb + L_2.nb)\}$
$\downarrow$ {printf(L.nb)}

$L \rightarrow L\,B \{L.nb = L.nb + B.nb\}$
$\mid B \{L.nb = B.nb\}$

$B \rightarrow 0 \{B.nb = 1\}$
$\mid 1 \{B.nb = 1\}$



ii/p

12. Construct a SDT to convert given decimal no. into binary number.

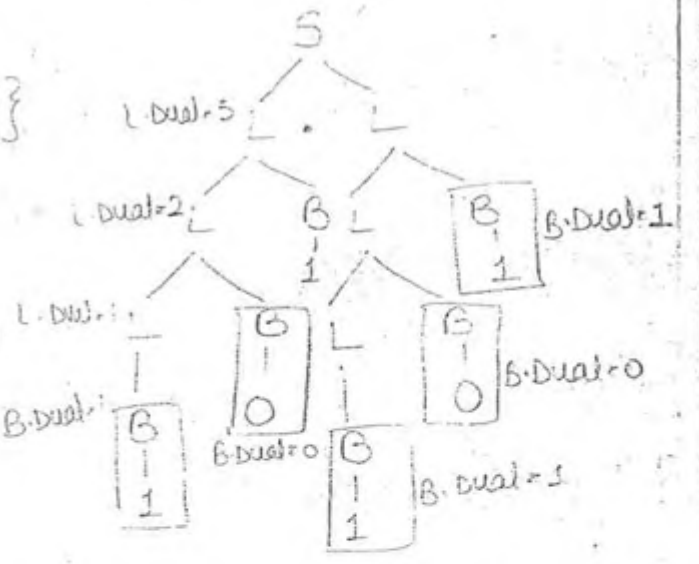
i/p I/P: 101.101

ii/p I/P: 5.625

iii/p $S \rightarrow L \mid L.L \Rightarrow \{printf(L_1.DV + \frac{L_2.DV}{2^{L_2.DV}})\}$
$\downarrow$
$printf\{(L.DV)\}$

$L \rightarrow L\,B \begin{cases} L.nb = L.nb + B.nb \\ L.DV = 2 * L.DV + B.DV \end{cases}$
$\mid B \begin{cases} L.DV = B.DV \\ L.nb = B.nb \end{cases}$

$B \rightarrow 0 \begin{cases} B.nb = 1 \\ B.Dual = 0 \end{cases}$
$\mid 1 \begin{cases} B.Dual = 1 \\ B.nb = 1 \end{cases}$



13. Construct SDT to convert the given infix expression into prefix expression:-

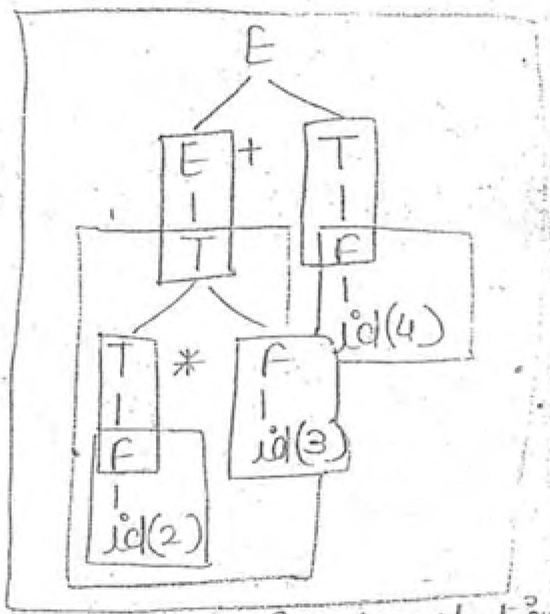
I/P:- 2*3+4

O/P:- +*234

Sol^n $E \to E+T \Rightarrow \{printf(+)\} E+T$
$\quad | T$

$T \to T*F \Rightarrow \{printf(*)\} T*F$
$\quad | F$

$F \to id \Rightarrow \{printf(id)\}$



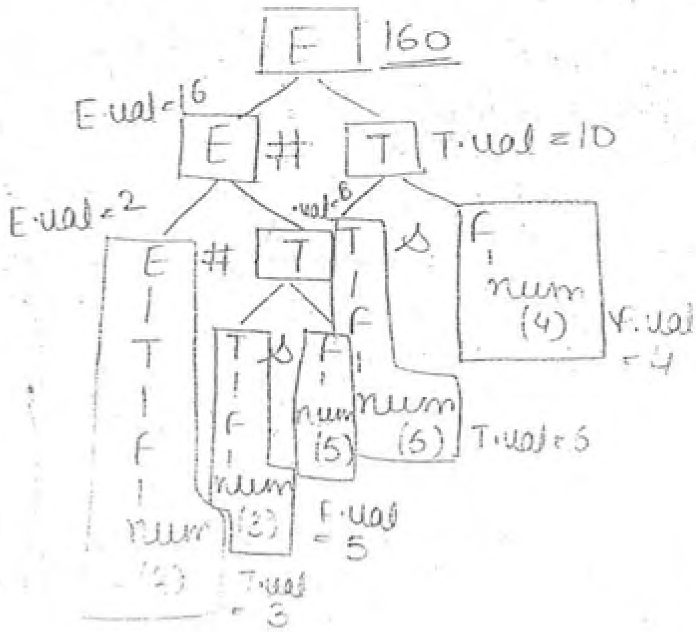
Qn. [GATE] Consider the grammar with the following translation rules:- $ E as the start symbol-

$E \to E_1 \# T$ { $E.val = E_1.val * T.val$ }
$\quad | T$ { $E.val = T.val$ }
$T \to T_1 \$ F$ { $T.val = T_1.val + F.val$ }
$\quad | F$ { $T.val = F.val$ }
$F \to num$ { $F.val = num$ }

Compute the E.val for the root of the parse tree for the expression-
2 # 3 $ 5 # 6 $ 4

Sol^n

E.val = 160 Ans



Qn. 512 a) b) c) d) Sol

4.8 E → E#T { E.val = E1.val * T.val } |
T { E.val = T.val }

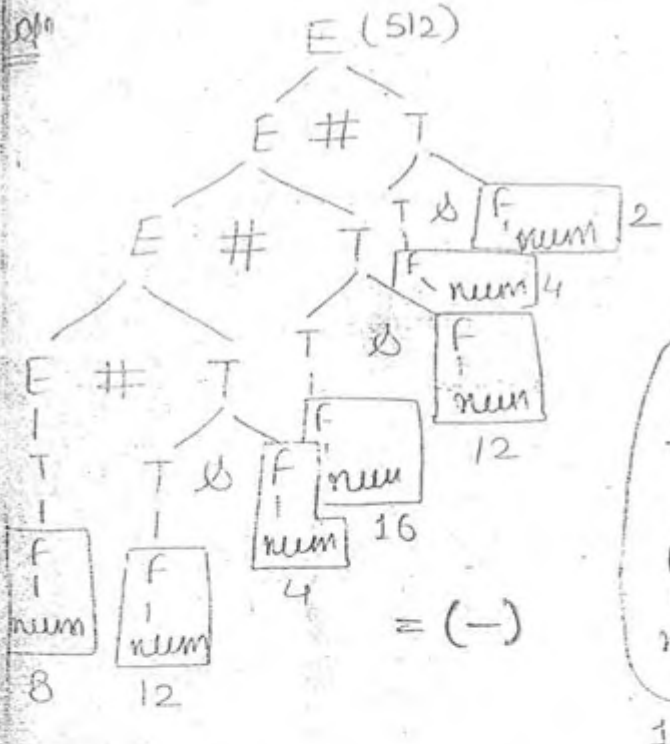
T → T$F { —— } T.val = T.val - F.val
| F { T.val = F.val }

F → num { F.val = num }

Ques 4.9) If the expression 8#12$4#16$12#4$2 is evaluated to 12, which one of following is correct-

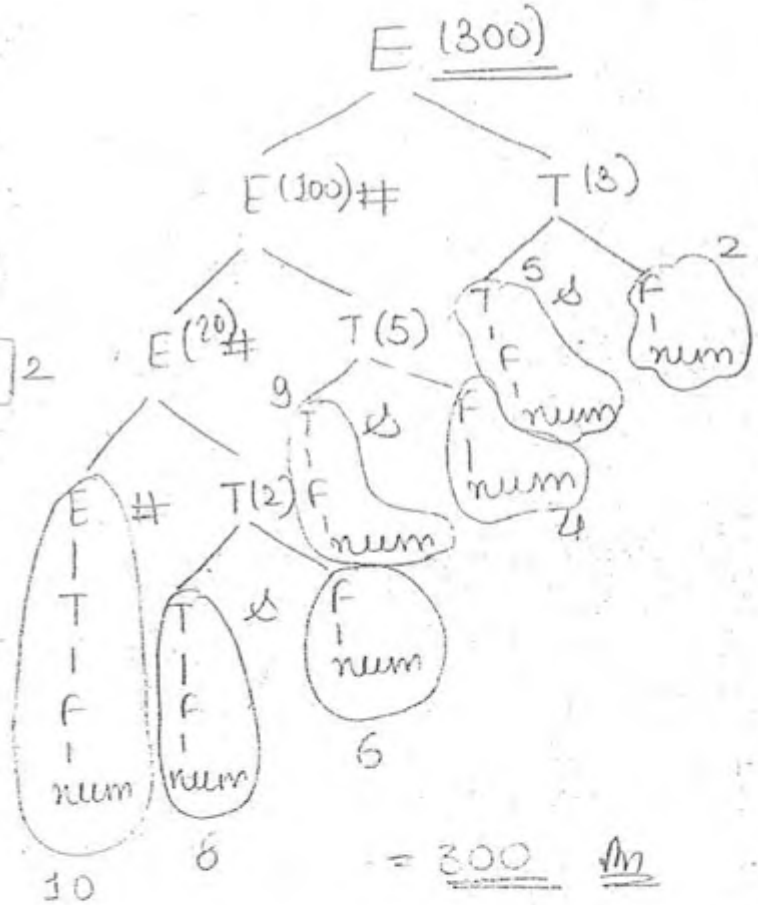
a) T.val = T1.val * F.val
b) T.val = T1.val + F.val
c) T.val = T1.val / F.val
d) none

Ans



= (—)

Ans(b) Compute 10#8$6#9$4#5$2



= 300 Ans

Ans If the given grammar is -

S → TR
R → # T { print(#) } R | ε
T → num { print(num) }

If the I/P is- 9+5+2, what will be the O/P-

a) 9+5+2

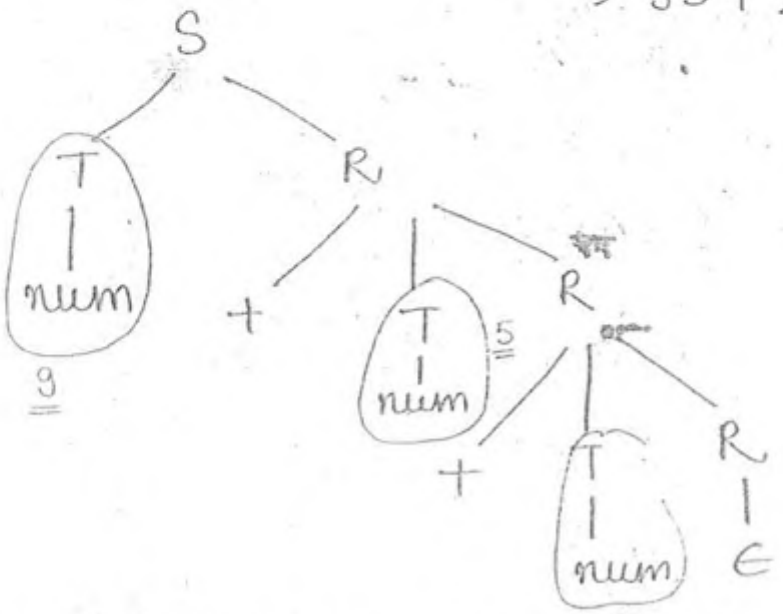
b) 95+2+

c) 952++

d) ++952

⇒ 95+'2+

Sol^n



Q Construct the SDT to store type info into symbol table.

I/P: int x, y, z;

O/P:

V.name	V.type
x	int
y	int
z	int

Sol^n

$D \rightarrow D_1, id$ { D.type = D_1.type; addtype(id, D_1.type) }

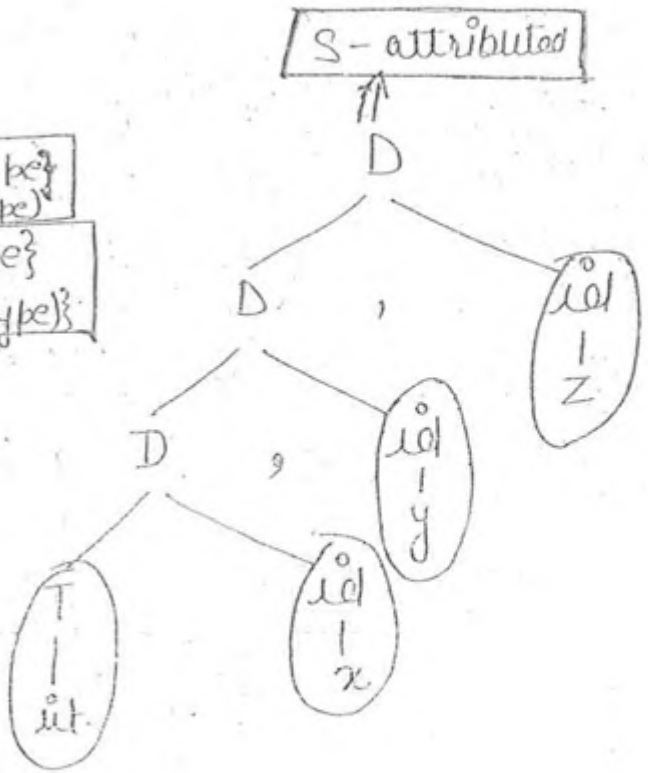
| T id { D.type = T.type; addtype(id, T.type) }

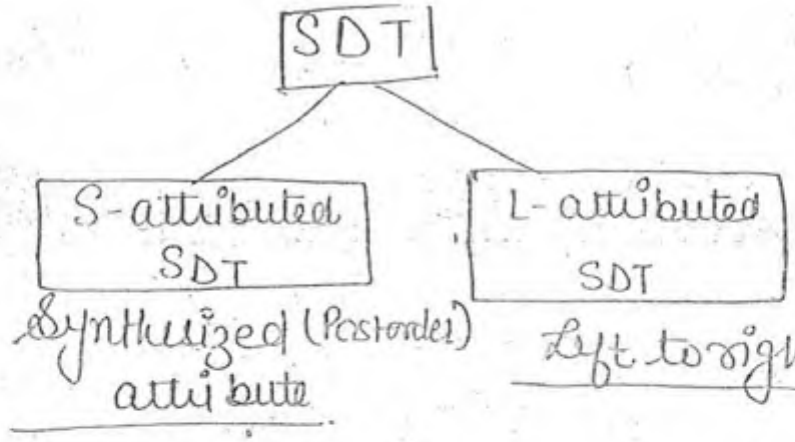
$T \rightarrow$ int { t.type = int }

| float { t.type = float }

| char { t.type = char }

$id \rightarrow a \mid b \mid c \ldots \mid z$





Problems using Inherited attributes (Preorder evaluation)

QH $S \rightarrow T\ I$ $\{I.i = T.type\}$

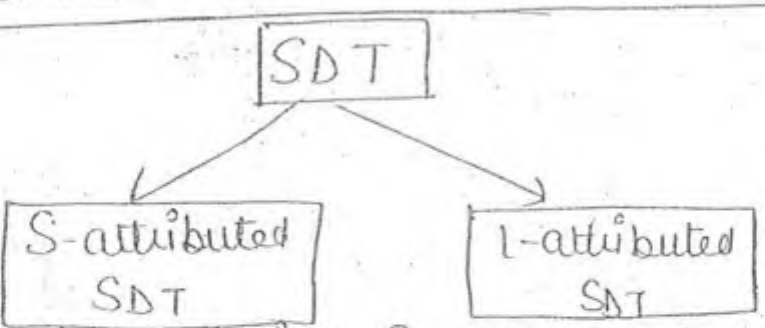
L-attributed ⇐ S

$T \rightarrow int$ $\{T.type = int\}$
| char
| float

$I \rightarrow I_1, id$ $\{AI_1.i = I.i\}$ $\{addtype(id, I_1.type)\}$.

$I \rightarrow id$ $\{addtype(id, I_1.type)\}$

If one required value is on L.H.S and other value is on R.H.S., no method other than inherited attribute.



we will use only synthesizer attribute

① we can use both synthesizer and inherited attribute But in the case of inherited attribute, only, values from root or the left siblings can be taken not from right sibling.

② we can place the semantic rule, anywhere at the right hand side of production

semantic rules will be placed after end of the production

evaluation is (3) The order of evaluation - Preorder.

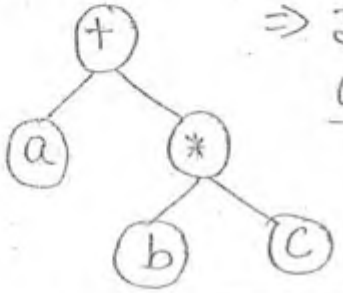
S-attributed definitions are also S-attributed definition

S-attribute ⊂ L-attribute

Q. Construct SDT to construct syntax tree for the given infix expression:-

I/P: a+b*c

O/P:



⇒ In the syntax tree leaf nodes are Operands and intermediate nodes are Operator.

⇒ makenode(lp, id1, rp) = Creating a node

⇓

will return the location, where the node is created.

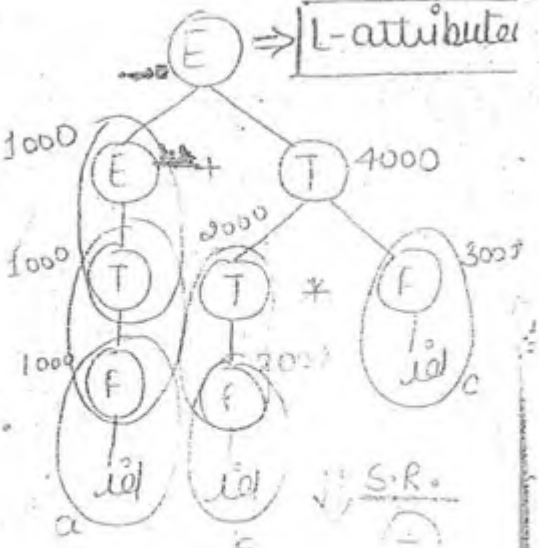
Grammar

S-attribute ⇑ L-attribute

$E \rightarrow E_1 + T \{E.ptr = makenode(E_1.ptr, +, T.ptr)\}$
$\quad | T \{E.ptr = T.ptr\}$

$T \rightarrow T_1 * F \{T.ptr = makenode(T_1.ptr, *, F.ptr)\}$
$\quad | F \{T.ptr = F.ptr\}$

$F \rightarrow id \{F.ptr = makenode(N, id, N)\}$



S.R.

Q1 Give a grammar to reverse the given infix expression-

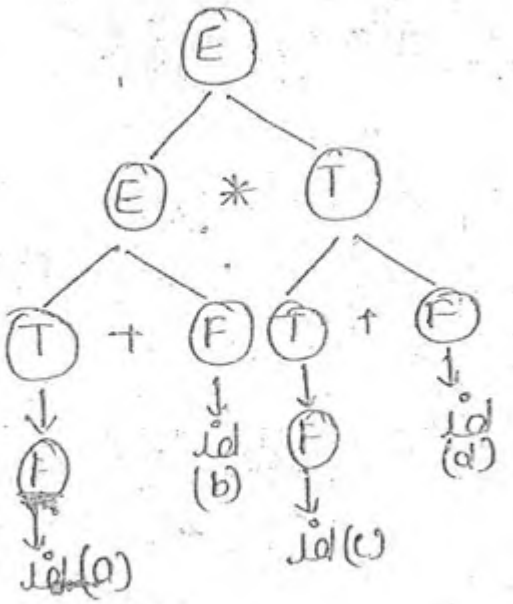
I/P: (a+b) * (c+d)

O/P: (d+c) * (b+a)

$E \rightarrow E * E \mid (T)$

$T \rightarrow T + F \mid F$

$F \rightarrow id$



Semantic rules

$E \rightarrow E * E$

$\rightarrow (T)$

$\rightarrow T + F$

$\mid F$

$F \rightarrow id \{F.val = id\}$

Q2. Construct SDT to generate Three address code (Intermediate code) for the given infix subexpression:-

I/P: $x = a + b * c$ → newtemp() = Create temporary variable

O/P: $t1 = b * c$

$t2 = a + t1$ → gen(t = b * c)

$x = t2$

Sol" $S \rightarrow id = E \{gen(id = E.val)\}$

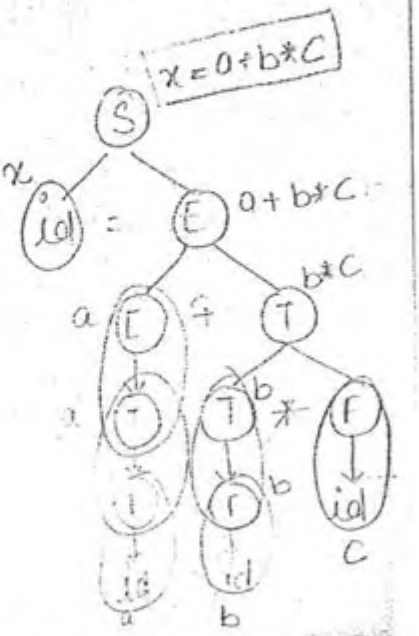
$E \rightarrow E + T \begin{cases} Q = newtemp() \\ gen(Q = E1.val + T.val) \\ E.val = Q \end{cases}$

$\mid T \{E.val = T.val\}$

$T \rightarrow T1 * F \begin{cases} P = newtemp(); \\ gen(P = T1.val * F.val \\ T.val = P \end{cases}$

$\mid F \{T.val = F.val\}$

$F \rightarrow id \{F.val = id\}$



QN GATE

Consider the SDT shown below:-

$S \rightarrow id = E \{ gen(id.place = E.place) \}$

$E \rightarrow E_1 + E_2 \{ t = newtemp(); \; gen(t = E_1.place + E_2.place); \; E.place = t; \}$

$E \rightarrow id \{ E.place = id \}$

Here gen is a function that generates the O/P code and newtemp(); is a function that returns the name of new temporary variable at every call. Assume that t_i are the new temporary variable name, generated by newtemp. For the statement $x = y + z$, the three address code generated by the above SDT is-

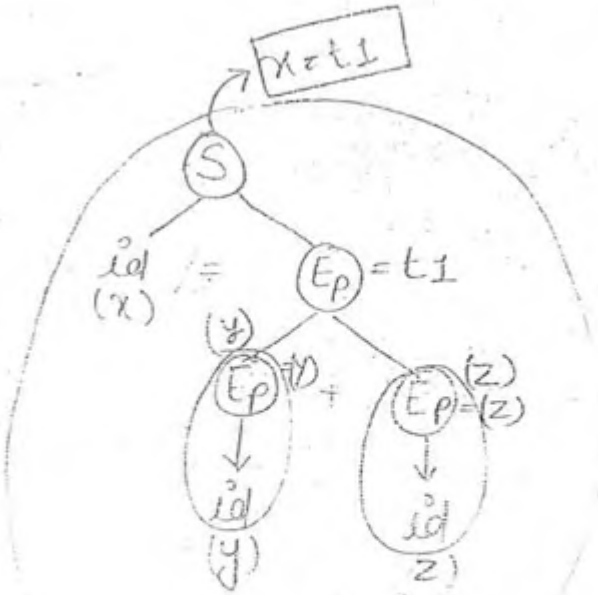
a) $x = y + z$

b) $t_1 = y$, $t_2 = t_1 + z$
$x = t_2$

c) $t_1 = y + z$
$x = t$

d) $t_1 = y$, $t_2 = z$
$t_3 = t_1 + t_2$, $x = t_3$.

Soln $t_1 = y + z$
$x = t_1$



4. Consider the following SDT:-

$E \rightarrow$ number $\{E.val = number\}$

$| E+E \{E.val = E_1.val + E_2.val\}$

$| E*E \{E.val = E_1.val * E_2.val\}$

YACC

↓

Yet Another Compiler Compiler

Sol" I/P: 3*2+1

YACC (Give more priority to shift (Push), rather than reduce (pop)

*	+		

3, (2, 1, +) *

QH a) The above grammar and semantic rule is given to YACC tool for parsing and evaluating arithmetic expressions, which one of the following is true, about the action of YACC for given grammar -

i) It detects recursion and eliminate

ii) It detects reduce-reduce conflicts and resolves

iii) It detects shift-reduce conflict and resolve the conflict and resolves in favour of shift over reduce.

iv) resolves favour of reduce over shift

b) Assume the conflict in QH(a), what will be the precedence and associativity for the expression - 3*2+1

i) equal precedence and left associative, evaluated to 7.

ii) Equal precedence and right associativity, evaluated to 9.

YACC tool = LALR(1) parser generator

↓ Parser → no multiple value entry

LL(1) or LR(1) ⇒ LL(1), Because in LR(1) there is YACC tool.

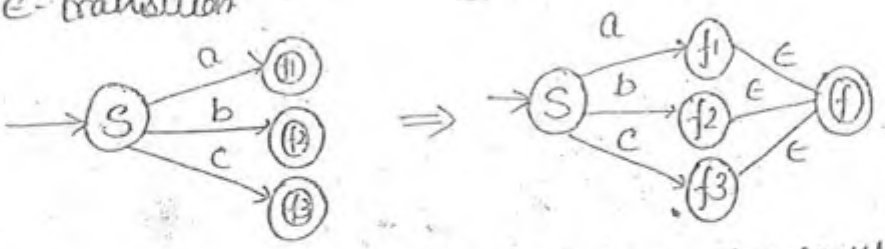
Dated 28.Dec.10

FA→RE

Steps

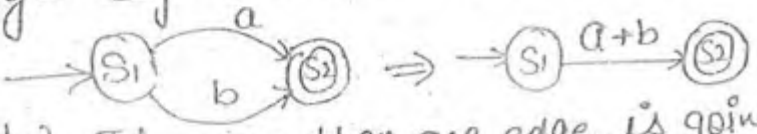
Step-1 If more than 1 final state is there, make it as single final by adding ε-transition

Exp:-



* A NFA with more than one final state can be converted into equivalent NFA with single final state, but it is not possible in case of DFA.

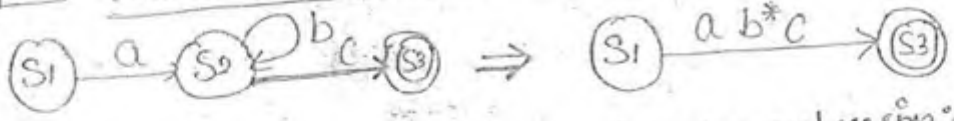
Step-2 If more than 1 edge going in same direction make it as single edge and label with union with symbol.



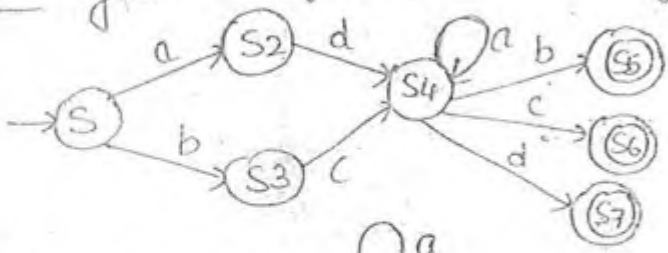
Step-3 If more than one edge is going in same direction one after another, making it as single edge, with the label of concatenation of symbols.

Exp:-

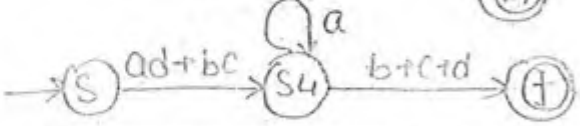
Step-4 State Elimination method



QN.1 Give the equivalent the regular expression:-

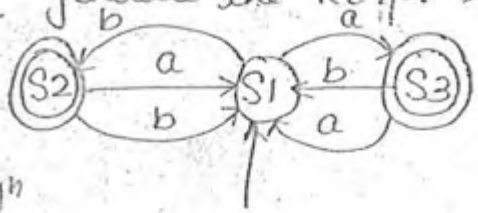


Sol"

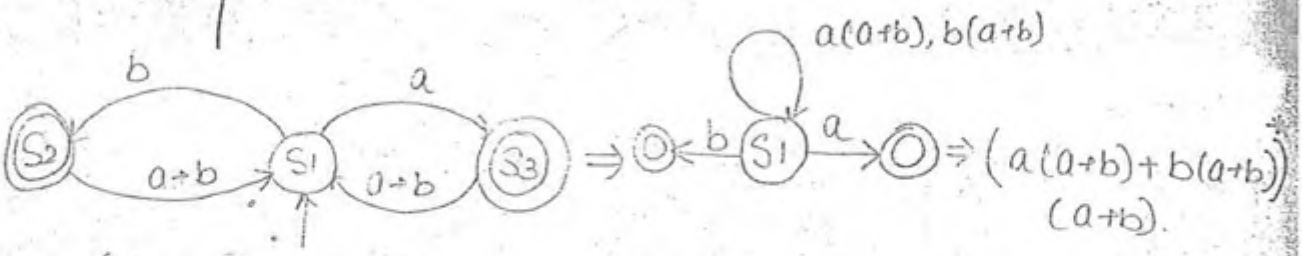


= $\boxed{(ad+bc)\,a^*\,(b+c+d)}$ Ans

ON Generate the RE for the following automata :-



Sol'n



$\Rightarrow (a(a+b)+b(a+b))^*(a+b)$.

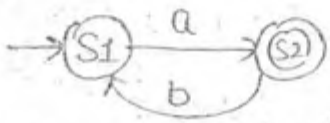
$= ((a+b)(a+b))^*(a+b)$

$= ((a+b)^2)^*(a+b)$

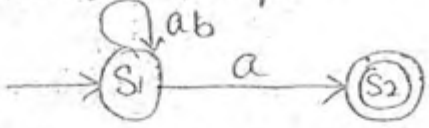
Even length string followed by a or b

Odd length string ans

ON Give the RE for the following finite automata :-

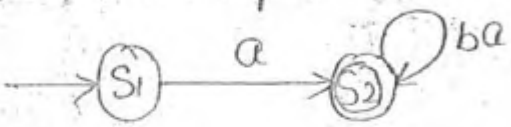


Maintain the loop at S1 :-



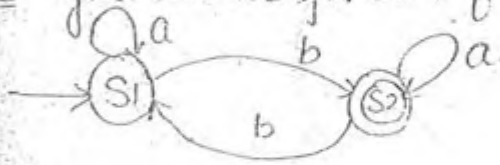
$\Rightarrow (ab)^*a$

Maintain the loop at S2 :-

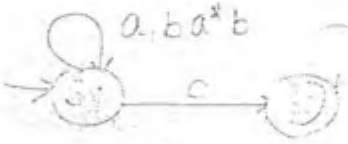


$\Rightarrow a(ba)^*$

ON Give the RE for the following finite automata :-



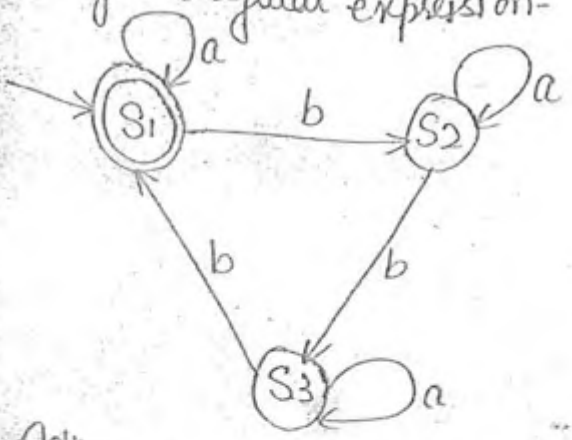
Sol'n



$= (a+ba^*b)^*ba^*$

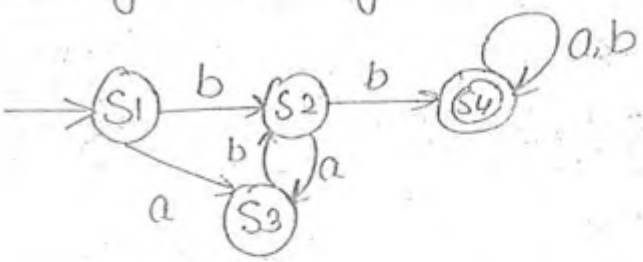
$= a^*c(a+ca^*c)^*$

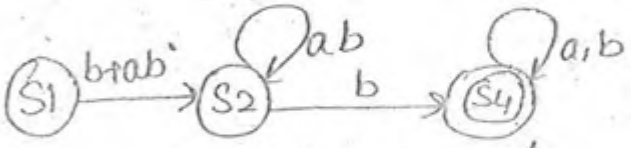
Q4 Give regular expression-



Ans $(a+(ba^*ba^*b))^*$

Q4 Give the regular expression:

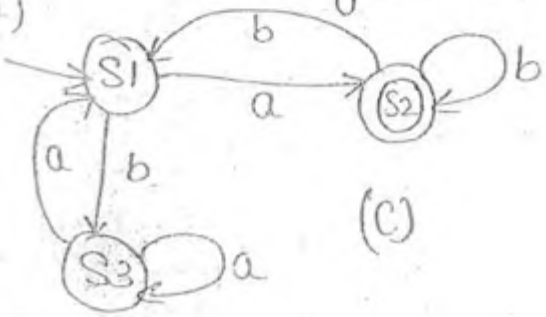




$= (b+ab)(ab)^* b(a+b)^*$

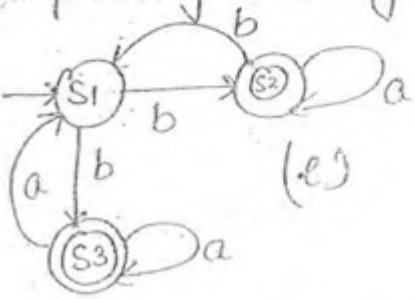
Q4 Match each of the NFA, with corresponding matching option-

1)



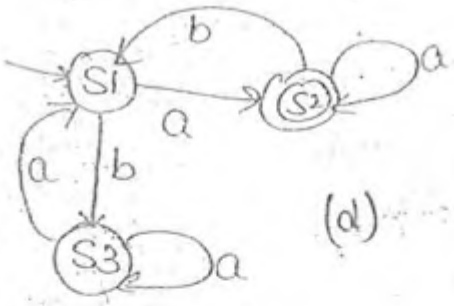
(c)

(5)



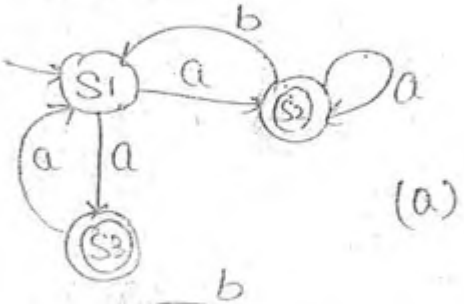
(e)

2)



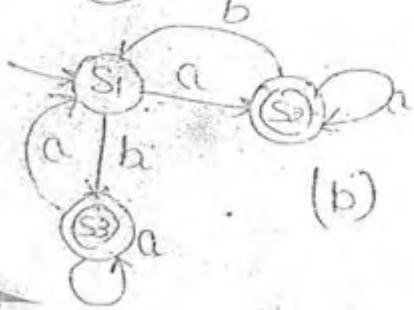
(d)

3)



(a)

4)



(b)

a) $(aa^*b+ba^*b)^* ba^*$

b) $(aa^*a+aa^*b)^* aa^*$

c) $(ba^*a+ab^*b)^* ab^*$

d) $(ba^*a+aa^*b)^* aa^*$

e) $(ba^*a+ba^*b)^* ba^*$

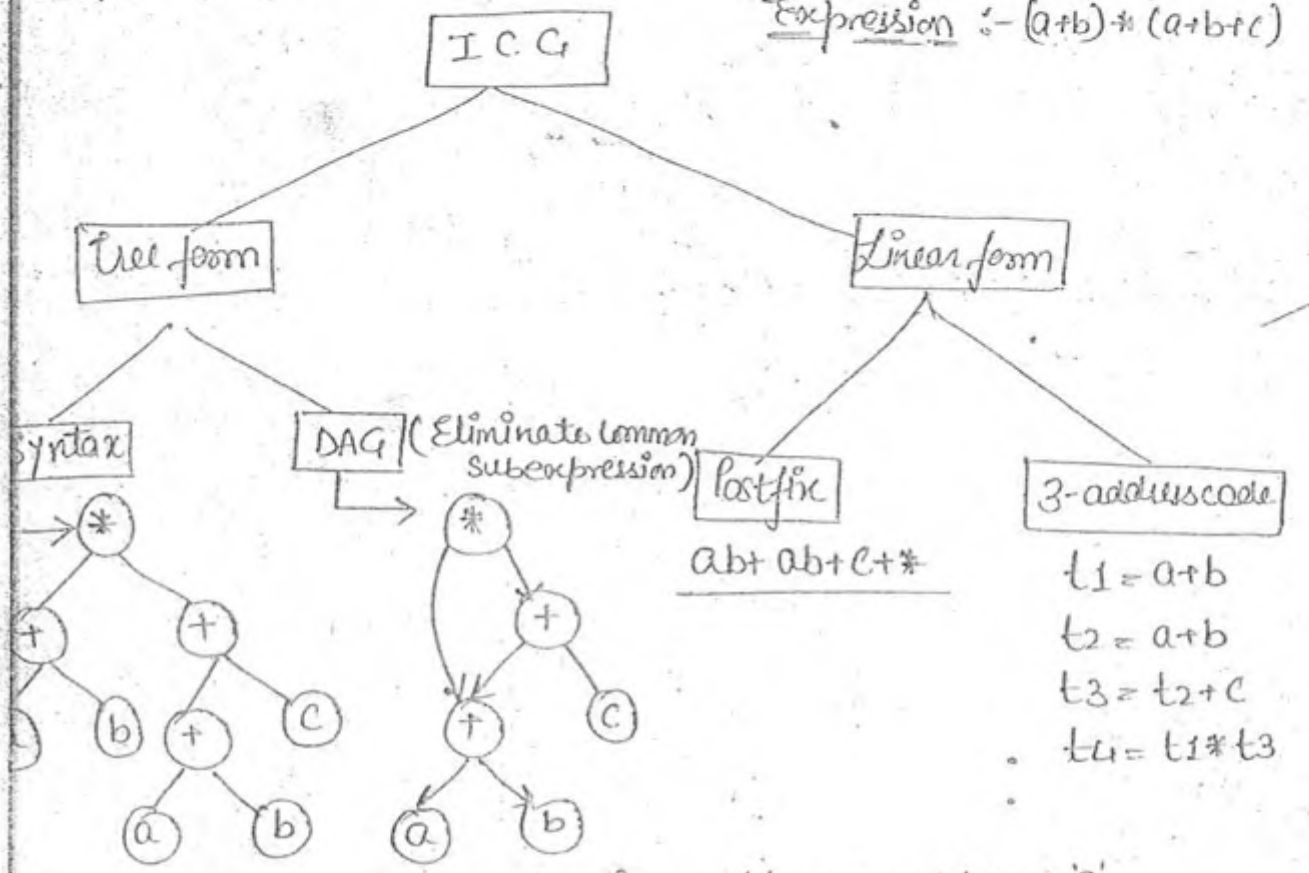
Dated 3. DEC. 10

Chapter No 4

Intermediate Code Generation

Representation of intermediate code generation

Expression :- (a+b) * (a+b+c)

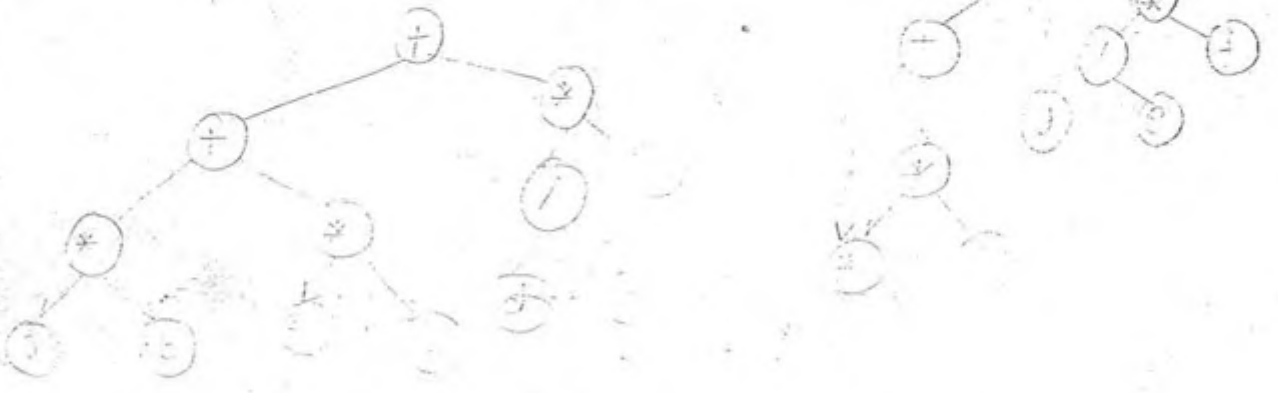


DAG :- atleast one node with indegree '0' and outdegree '0'.

Q.H.2 (a*b)+(a*b*c)+d/e*f

Sol" Syntax tree

DAG



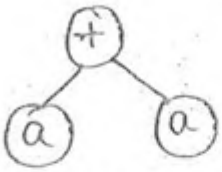
Postfix

ab* ab*c*+de/f*+

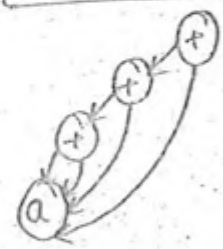
3-address code

t1 = a*b
t2 = a*b
t3 = t2*c
t4 = t1+t3
t5 = d/e
t6 = t5*f
t7 = t4+t6

DAG – (a+a)+(a+a)



a+a+a+a



(a+a)+(a+a+a)

Types of three address code

1) x = y op z
2) x = op y
3) x = y
4) x = *y
5) x = &y
6) x = a[i] ⇒ *(a+i)
7) a[i] = x
8) goto L (unconditioned jump)
9) if x<y goto L

No three addres code

- x = a[i,j]
- x = fun(a,b)

Q) Construct three address code for the following expression
if a<b then t=1 else e=0

Ans) It is not a three address code.

⇓ Conversion in three address code

i) if a<b goto i+3
(i+1) e=0
(i+2) goto i+4
(i+3) t=1
(i+4) —

Back patching (filling gaps)

Q) If a<b && c>d then t=1 else t=0

Ans) It is not in three address code.

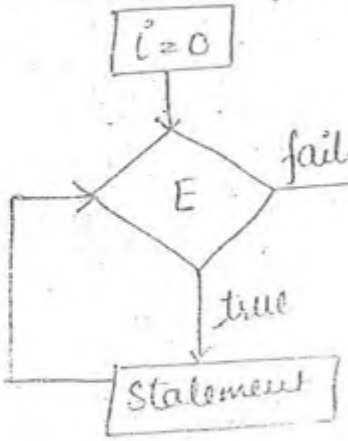
⇓ Conversion in three address code

i) if a<b goto i+1
(i+1) if c>d goto i+4
(i+2) e=0
(i+3) goto i+5
(i+4) t=1
(i+5) —

i) if a<b goto i+2
(i+1) goto i+3
(i+2) if c>d goto i+5
(i+3) e=0
(i+4) goto i+6
(i+5) t=1
(i+6) —

Q) Construct three address code for 'while' statement in C language

Ans)
```
i=0
while(i<10)
{
  x=a+b*c;
  i++;
}
```

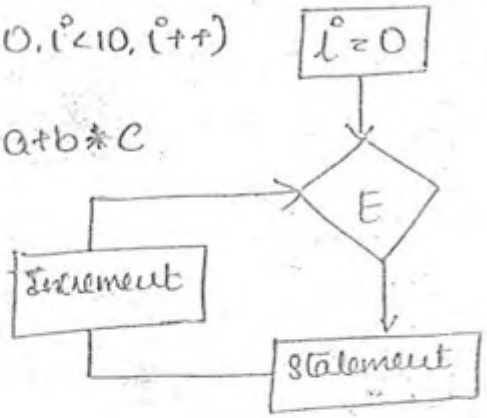


Three address code (condition)

i=0
S) if i<10 goto S+2 (if true)
S+1) goto S+7 else outside
S+2) t1=b*c
S+3) t2=a+t1
S+4) x=t2
S+5) i=i+1
S+6) goto S
S+7) —

QN Construct three address code for 'for' loop in C language.

Soln
```
for(i=0, i<10, i++)
{
  x = a+b*c
}
```




```
S+0) i=0          for(i=0; i<10; i++)
S+1) if i<10 goto S+2
S+1) goto S+7          a=b+c;
S+2) t1 = b*c
S+3) t2 = a+t1
S+4) x = t2
S+5) i = i+1
S+6) goto S
S+7) ——
```

QN Construct a three address code for switch statement in 'C' language

Soln
```
i=1
switch(i)
{
  case 1: x1 = a1+b1*c1
          break;
  case 2: x2 = a2+b2*c2
          break;
  default: x3 = a3+b3*c3
}
```

Three address code

```
     i=1
S)   if(i==1) goto case1
S+1) if(i==2) goto case2
S+2) t1 = b3*c3
S+3) t2 = a3+t1
S+4) x3 = t2
S+5) ——
case1:) t1 = b1*c1
        t2 = a1+t1
        x1 = t2
        goto S+5
case2:) t1 = b2*c2
        t2 = a2+t1
        x2 = t2
        goto S+5
```

Q14 Construct three address code for. $x = a[i][j]$, suppose $a[10][20]$

Sol^n It is not a three address code.

⇓ Conversion in three address code.

$x = a[i][j] = *(*(a+i)+j)$

$t1 = i * 20$

$t2 = t1 + j$

$x = a[t2]$

$a[5,10]$

$5 * 20 = 100$

$+ 10$

110

Representations of three address code

1) Quadruples
2) Triples
3) Indirect Triples

Expression :- $-(a+b) * (a+b*c)$

1) Quadruples → Advantage - Can move the result.
Disadvantage - More space

S.No.	OP	OP1	OP2	Result
1	+	a	b	t1
2	−	t1		t2
3	*	b	c	t3
4	+	a	t3	t4
5	*	t2	t4	t5

Memory (Quadruples)

	OP1	OP2	result
	a	b	t1
	t1		t2
	b	c	t3
	a	t3	t4
	t2	t4	t5

2) Triples

S.No.	OP	OP1	OP2
1	+	a	b
2	−	(1)	
3	*	b	c
4	+	a	(3)
5	*	(2)	(4)

Advantage :-

* Less space.
* Can't move the result at desired place. → disadvantage

3) Indirect tuples

→ If there is a requirement, then we can move the result to some another location by copying the same values.

Advantages

* less space is required.
* Results can be move.

Q4 (a+b) * (a+b+c) * d/e + f

Sol^n Quadruples

S.No.	OP	OP1	OP2	Result
1	+	a	b	t1
2	+	a	b	t2
3	+	t2	c	t3
4	*	t1	t3	t4
5	*	t4	d	t5
6	/	t5	e	t6
7	+	t6	f	t7

Triples

S.No.	OP	OP1	OP2
1	+	a	b
2	+	a	b
3	+	(2)	c
4	*	(1)	(3)
5	*	(4)	d
6	/	(5)	e
7	+	(6)	f

Indirect Triples

S.No.	OP	OP1	OP2	Copy
1	+	a	b	
2	+	a	b	
3	+	t2	c	
4	*	t1	t3	500
5	*	t4	d	
6	/	t5	e	
7	+	t6	f	

Dec-2010

Chapter No-5

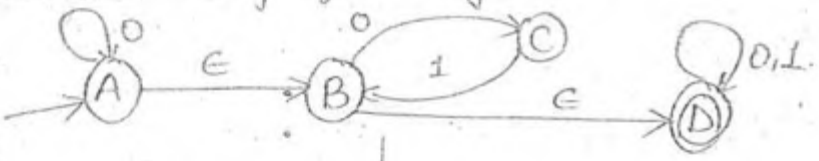
CODE OPTIMIZATION

ϕ ε-NFA ⇒ NFA

Soln Conversion

	0	1	2
→q0*	q0,q1,q2	q1,q2	q2
q1*	ϕ	q1,q2	q2
q2*	ϕ	ϕ	q2

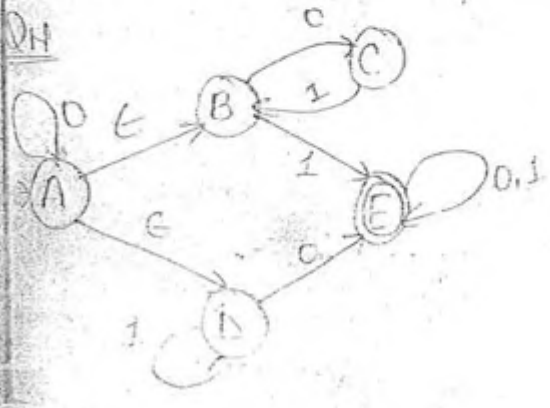
Q Construct NFA for following ε-NFA:-



Soln

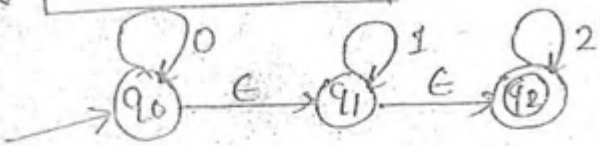
	0	1
→A*	A,B,C,D	D
B*	C,D	D
C	ϕ	B,D
*D	D	D

Q

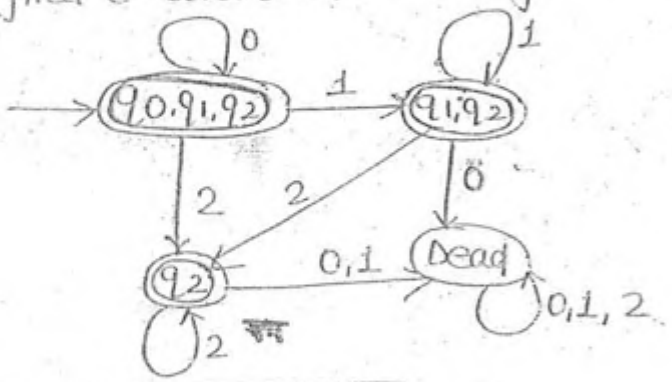


	0	1
→A*	A,B,C,D,E	D,E
B	C	E
C	ϕ	B
D	E	D
*E	E	E

ϕ ε-NFA → DFA



- Find ε-closure to starting state, then-



ϕ Quotient Operation

If L_1 is regular and L_2 is also regular, then L_1/L_2 is also regular.

$$L_1/L_2 = \{x \mid \forall\, xy \in L_1 \text{ and } y \in L_2\}.$$

Exp:- $L_1 = \{b^2, b^4, b^6, b^8, \dots\}$

$L_2 = \{b\}$

$L_1/L_2 = \{b, b^3, b^5, b^7, \dots\}$

$L_3 = \{a\}$.

$L_1/L_3 = \{\}$

$L_2/L_1 = \{\}$.

Exp:- $L_1 = \{101, 011, 0010, 00\}$.

$L_2 = \{0, 1\}$

$L_3 = \{00\}$.

$L_1/L_2 = \{001, 0, 10, 01\}$.

$L_1/L_3 = \{\epsilon\}$

$L_3/L_2 = \{0\}$.

Note:- In L_1/L_2, if L_2 contain ε, then $L_1/L_2 = L_1 \cup \{\}$.

Note-2 If L is non empty, then $\frac{\Sigma^*}{L} = \Sigma^*$ | If L is empty then $\frac{\Sigma^*}{L} = \{\}$. (Nomatching)

Note-3 If L is non-empty, then $\frac{L}{\Sigma^*}$ = all the prefixes of L.

$\frac{a}{(a+b)^*} = \epsilon, a$

$\frac{TOC}{(A+B+\cdots Z)^*} = \epsilon, T, TO, TOC$ = all the prefixes of L.

Code Optimization

* Loop Optimization
* Strength Reduction
* Redundancy Elimination
* Dead code elimination
* Constant Folding
* Copy propagation
* Algebric Simplification

Loop Optimization

(1) Loop invariant (code motion)

(2) Loop unrolling (Decreasing testcases)

(3) Loop jamming (Loop combine)

1 Loop Invariant

```
i=0
while(i<=10,00,000)
{
  x = Sin(A) * Cos(B) * i      ----> not varying (invariant)
  i = i+1;
}
```

⇓ take invariant code outside the while loop-

```
i=0
t= Sin(A) * Cos(B)
while (i<= 1,00,000)
 {
   x= t*i;
   i= i+1;
 }.
```

2 Loop Unrolling (Decreasing test cases).

```
while (i<= 10,00,000)
 {
   x[i]=i;
   i++;
 }.
```

10,00,000 test cases ⇒

```
while (i<= 10,00,000)
{
  x[i]=i;
  i++;
  x[i]=i;
  i++;
}
```

5,00,000 test cases.

3 Loop Jamming (Loop Combine).

```
i=1;
while (i<= 10,00,000)
 {
   x= a+b*i;
   i= i+1;
 }
```

10,00,000 Comp

```
i=1;
while (i<= 10,00,000)
 {
   y= c+d*i
   i++;
 }.
```

10,00,000 Comp

```
i=1
while (i<=10,00,000)
{
  x= a+b*i;
  y= c+d*i;
  i++;
}.
```

4 Strength Reduction :- Replacing costlier operation by less cost operation or replacing lower speed operator to the higher speed operator

Exp:- n ⇒ * ⇓ Left shift

2*n

i*j ⇒ * ⇓ (less time)

i+i+i+i

Constant Folding :-

Fold all the constants and give one equivalent value.

$a = b + \boxed{5+10+15+25}$

$\Downarrow$

$a = b + \boxed{55}$

Copy Propagation :- Unnecessarily don't propagate the constant by copying one by one into another variable.

Exp:-
$PI = 3.14$
$x = PI$
$y = x * 100$
$z = 100$
$a = y/z$

Redundancy Elimination :- Use DAG Data Structure

$A = b+C$
$B = 2+b+3+C$
$C = C+1+b$

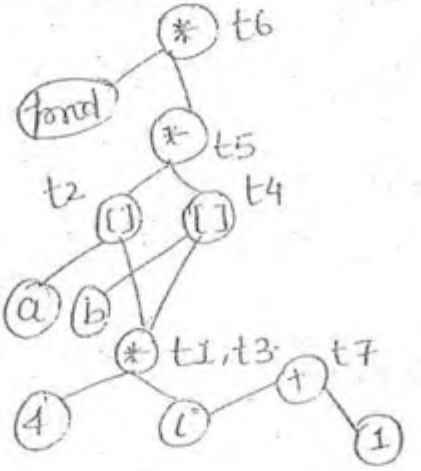
$\Downarrow$

$A = b+C$
$B = 5+A$
$C = A+1$

Exp-2

$t1 = 4*i$
$t2 = a[t1]$
$t3 = 4*i$
$t4 = b[t3]$
$t5 = t2*t4$
$t6 = prod*t5$
$t7 = i+1$

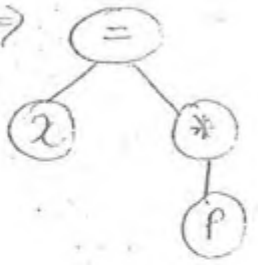
$\Rightarrow$



$\Downarrow$

$t3 = 4*i$
$t2 = a[t3]$
$t4 = b[t4]$
$t5 = t2*t4$
$t6 = prod*t5$
$t7 = i+1$

$x = *p \Rightarrow$



Dead Code Elimination :-

Exp

$x = t1$
$a[t1] = t2$
$b[t2] = a[t1]$
$printf(b[t2])$

$\Rightarrow$

$a[t1] = t2$
$b[t2] = a[t1]$
$printf(b[t2])$

$\Rightarrow$ x is not at all useful.

Algebric Simplification

A = A * 1
B = B + 0
} ⇒ don't use this type of operators

GATE Problems

Q1 Consider the following C pgm-

```
for(i=1; i<N; i++)
{
  for(j=1; j<N; j++)
  {
    if(i%2)
    {
      x += 4*j + 5*i;
      y += 7 + 4*j;
    }
  }
```

Then which one of the following is false:-

a) above pgm contain loop invariant
b) above pgm contain common subexpression elimination.
c) above code contain strength reduction
d) None of the above.

- Common subexpression = 4*j
- Strength Reduction = j+j+j+j
-

```
for(i=1; i<N; i++)
  if(i%2)
for(j=1; j<N; j++)
{
  x += 4*j + 5*i;
  y += 7 + 4*j;
}
```

Q2 multiplication of a positive integer by a power of 2, can be replaced by left shift, which executes faster on most of the processors. This is an example of:-

a) loop unrolling
b) strength Reduction
c) Dead code Reduction
d) None of above

Q4 $i=1, j=0;$ for the above pgm, including integers i, j, n, which one of the following is loop invariant:-

```
while (j<n)
{
  i = 2*i;
  j = j+1;
}
```

i) $i = j+1$

ii) $i = (j+1)^2$

iii) $j = 2^i$

iv) $i = 2^{j+1}$

$\Downarrow$

$i = (2)^{j+1}$

Q4.20
$S \to AB|CA$
$B \to BC|AB$
$A \to a$
$C \to aB|b$

Sol^n Reduced form

① Eliminate all the states or variables which are not reachable from start symbol.

↳ $S \to AB|CA$
$B \to BC|AB$
$A \to a$
$C \to aB|b$

② Eliminate those variables and productions, which are unnecessary

$S \to \cancel{AB}|CA$
$\cancel{B \to BC|AB}$ $\Rightarrow$ $S \to CA$
$A \to a$ $A \to a$
$C \to \cancel{BB}|b$ $C \to b$

ϕ LA
Syntax
Semantic
I.C.G.
C.O.
T.C.G.

Chapter No-5

RUN TIME ENVIRONMENT

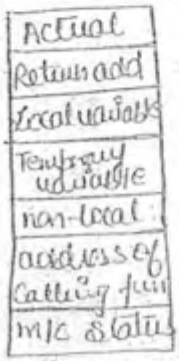
Environment (Binding)

a ⟶ 5000 memory location

⇒ variable will be allocated to the multiple locations at runtime. Variable will not changed.

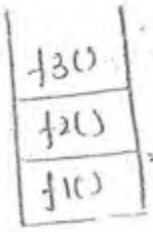
f1() → f2() → f3()

(Activation record)



⇒ Activation Record

Control stack



⇒ all the current active functions of the system in same order.

⇒ All d activation record first enter to the control stack.

(This are the information that should be to f1() before the control is going to f1() → f2()

Storage Allocation

i) Static Storage Allocation
ii) Stack Storage Allocation
iii) Heap storage allocation

→ memory created only once (static variable) = Compilation time memory allocation

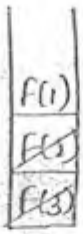
⇓

Can't be allocated at run time

* Static Storage Allocation

→ Memory is allocated at compilation time only
→ Bindings do not change at run time
→ One activation record for procedure
→ Recursion is not supported
→ Size of the object must be known as compile time itself (one time allocation of address)
→ Data structures can not be created dynamically (not pointers and dynamically allocated dynamically)

Exp:-



2 Stack Storage Allocation:-

→ Whenever is a function is called, activation record is created and pushed it into the stack.
→ Whenever a function ends, activation record is poped out from the stack.
→ At the time of running memory location will change.
→ Locals are bound to new activation record.

⇒ 6

Disadvantages:-
Locals can not retained when activation ends i.e function is over.

3 Heap Allocation

* Allocations and deallocation may be done in any order.

Code Optimization
Runtime Environment
} ⇒ Book

DBMS
① Nf
② Relational Algebra
③ Transaction

CN
① Data link layer
→ stop n wait
→ Gobackn
→ Selective reject
→ Tokening
→ Jamming
② N/W layer
→ Subnetting
→ Supernetting
③ Transport layer

OS
→ process magnt
→ Scheduling
→ Synchronization
→ Memory magnt
→ Page replacement
→ Disk Scheduling

Algo
NOTES

DS
NOTES

Compiler

Digital
Complete

CO
i) pipelining
ii) Addressing modes
iii) Memory magnt
iv) Floating point

Maths
→ Matrices
→ Graph
→ Relⁿ and fun
→ Lattices
→ group theory

Apti

④ Web technology
XML } W3school.com
HTML }

⑤ S/W Engg
→ Cyclomatic
→ Enigma model

COMPILER