Compiler

Done

Grammer in the automat (V, T, P, S)

- V — set of variables
- T — set of terminals
- P — Set of Production
- S — start symbol

ex

Production:

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

[// keep attention to capital 'A', 'a' and difference between more terminal and variables]

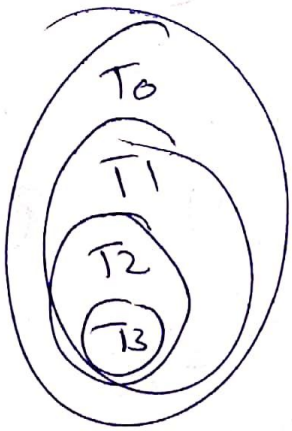
terminal = {a, b}

Variable = {A, B, S}

ε — not variable nor symbol.

$G(\{A, B, S\}, \{a, b\}, P, S)$

Types of Grammer According to Noam chomsky



Type 0: (unrestricted)

Type 1: (contax Sensitive)

Type 2: (contax free)

Type 3: (Regular Grammer)

Done

Type 0

$P: \alpha \rightarrow \beta$

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

$\in$ included (*)

404

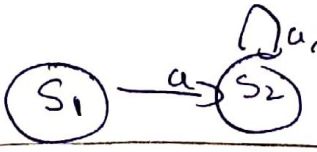
< All are >

$(a^*b)^*$ — b can be produced

$a+\varepsilon = a$ (or) $\in$

$a.\varepsilon = a$

(0) string accepting ε, $L=\{\varepsilon\}$, $L \neq \phi$ (~~Empty language~~ Empty String)



Language = ϕ (Empty language)

Type 1

$\alpha - \beta$

1) ~~$|\alpha| \leq |\beta|$~~ $(\alpha, \beta) \in (V+T)^*$

404

2) $|\alpha| - |\beta|$

< no contracting >
expansion

eg → $S \rightarrow AB$
→ $A \rightarrow a/B\varepsilon$ TOC
→ $B \rightarrow Bb$ Prodcc 4

done

Type 2

$A \rightarrow \beta$

$A \in V$

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

Condition

= To check type all left side should be single variable

eg $S \rightarrow AB$

$BA \rightarrow a$

$\beta \rightarrow b$

Still a single variable

Ans: If yes type 2

else Type 1 if $|\alpha| \leq \beta$

else type 0

Type 3

$A \rightarrow T^* B / T^*$

1 condition

2

variable on the right side right linear grammer.

$A \rightarrow BT^* / T^*$

Left linear grammer

// If both left an right not a regular gramer

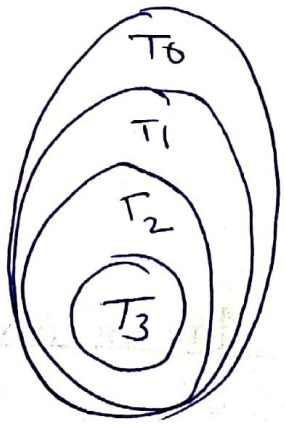
eg $S \rightarrow AB$ — not T_3 bcs of this

$A \rightarrow a$ — type 2

$B \rightarrow b$

eg $S \rightarrow aA/b$ T_3 (Right linear)

$A \rightarrow aA/b$



// First check left hand side for T_2 then go to type 3.

① Left side single variable T_2 confirmed

eg $S \rightarrow aA/b$

$A \rightarrow aA/\epsilon$

Type 1 X (null production)
Type 2 ✓
Type 3 ✓

Hierarchy not true for null Production

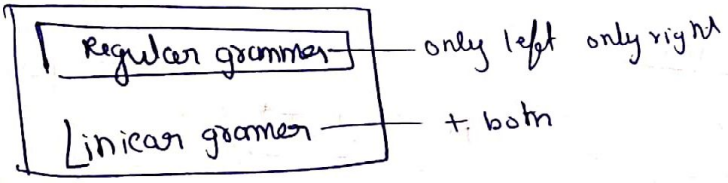
∴ special case

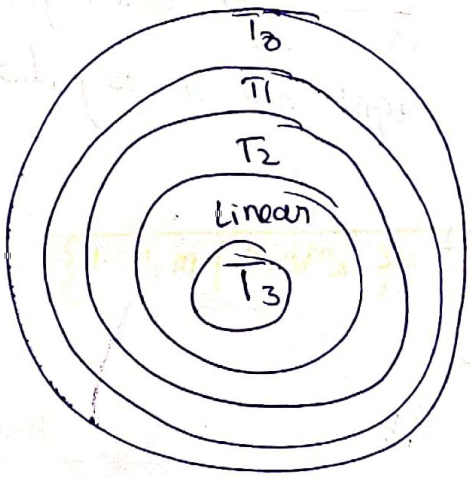
Regular grammer = left, right variable

eg $A \rightarrow T^*BT^*/T^*$ $A, B \in V$

$A, B \in V$

// It is not left linear or right linear it is linear grammer.





Linear Grammer

$A \rightarrow T^*BT^*/\alpha$

$A, B \in V$

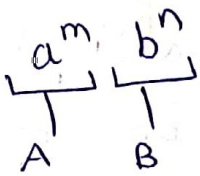
→ When asked to compare CFG and linear (don't consider Null production don't consider it every time.

Context free — left hand single or Right

Q Give CFG for $L = \{ a^m b^n \mid m, n \geq 1 \}$

ms:
mysol
$B \to AB/\epsilon$
$A \to aB/a$
$B \to bB/$
// lack of Relation divide it

thus..



ms:

$a^m / m \geq 1 \Rightarrow A \to aA/a$ (recursion) unlimited number of A^+ ie a...

$b^m / n \geq 1 \Rightarrow B \to bB/b$

Since order is imp $S \to AB$ (// first A will come then B will come)

To check if the grammer is CGF, match on its grammer or make its PDA

Q Give CFG for $L = \{ a^m b^n c^n \mid m, n > 1 \}$

$S \to ABC$

$A \to aA/a$

$B \to bBc/bc$ — for every B, c should also be there

CFG for $L = \{(a+b)^* abb (a+b)^*\}$

ms ✗ $S \to ABA$

$A \to aA \mid bB$

$B \to abb$

sol $S \to AabbA$

$A \to \varepsilon \mid aA \mid bBA$

CFG for $L = \{a^i b^{i+j} c^j \mid i, j \geq 1\}$

ms $S \to AbC$

$A \to aAb \mid$

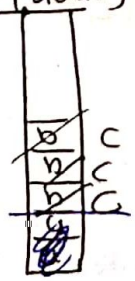
$C \to bcC \mid$

sol $S \to AB$ — $a^i b^{i+j} c^j$

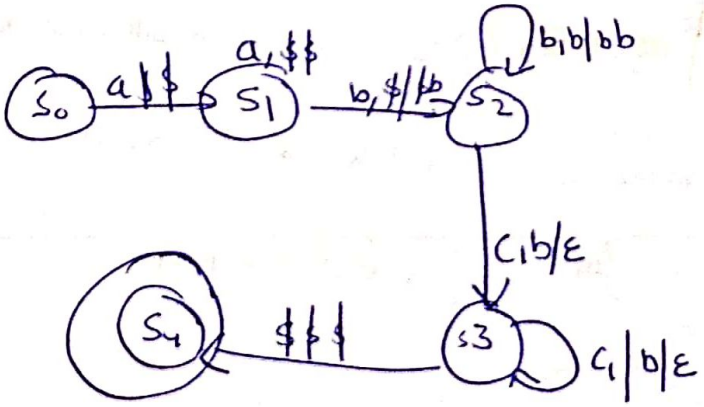
$A \to aAb \mid ab$ // $a^i b^i$

$B \to bBC \mid bc$ // $b^i c^j$

PDA (stack) — change state only to remember



$abbbccc\$$, $L = \{a^m b^n c^n, m, n \geq 1\}$



ex 

// accept anything no state required

$ — final state encounter final / pop everything else final.

Stack = where push and pop operation is allowed.

ai bi bi ci

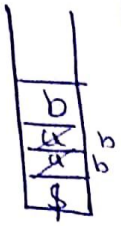
Q $L = \{ a^i b^{i+j} c^j \mid i,j \geq i \}$ //CFL

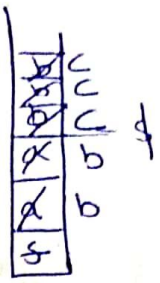
$S \to AB$
$A \to aAb \mid ab$
$B \to bBc \mid bc$

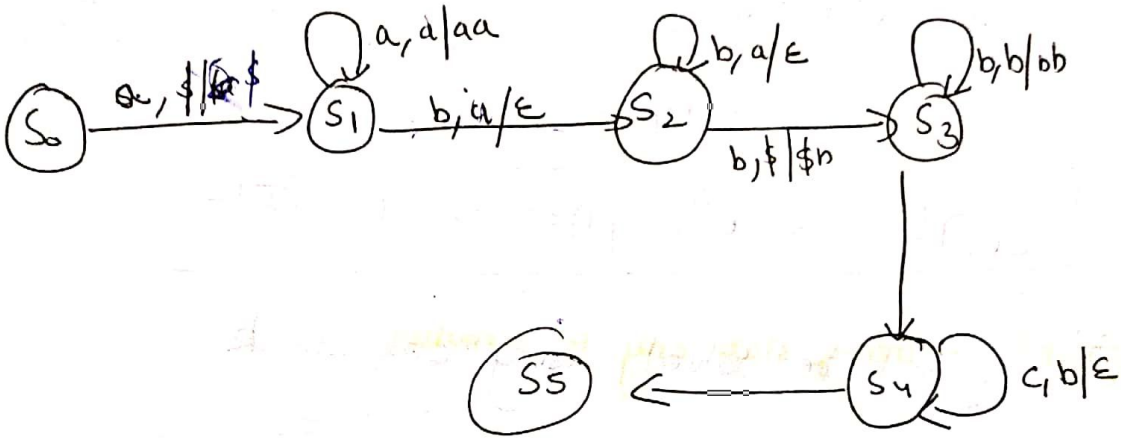
push bop

a a bb bbb ccc pop

Push L of bb stack empty push







S → 0

Q $\{ a^m b^n \mid m, n \geq 1 \}$

aaabb $

① final state as soon as $ is encountered (Regular can be solved without stack since no relation or comparison is required

Q $L = \{(a+b)^* a\, bb\, (a+b)^*\}$ // Stack more but not required

// CFL and regular

404 Stack

Q $L_1 : a^m b^n c^n \mid m,n \geq 1\}$ CFL (also DCFL)

(skip, Push pop)

L_2 $a^m b^m$ c^n — CFL (also DCFL)

Push pop, skip

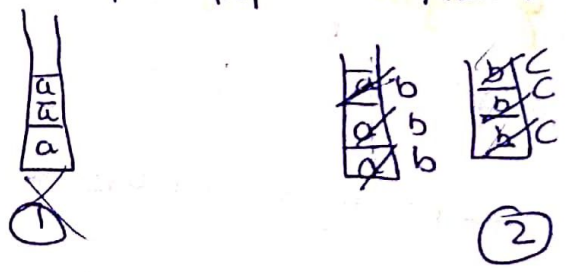
$L_1 \cap L_2 = a^n b^n c^n \mid n \geq \Rightarrow$ not CFL

not CFL ∩ CFL = not CFL

meaning ⌊ cannot be solved using one stack

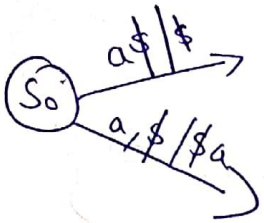
eg aaabbbccc

Push pop ⌊ How to compare



→ with 2 stack can do anything solve any problem

$L = L_1 \cup L_2 \Rightarrow \{ a^m b^n c^n \cup a^n b^m c^n \}$ not DCFL



on above union single machine cannot handle both sine 'a' machine will not ~~no know~~ which path to choose therefore it ~~is~~ there is a dilemma therfore U may not be DCFL /(in this cas it is DCFL)

3 Points
- CFL U CFL → may may not be DCFL
- DCFL ∩ DCFL → may not b DCFL
- CFL U CFL → may not be CFL

If dilemma is not there and machine able to distingush then it ~~may~~ will be a DCFL

e.g. $(a+b)^*$ ⇒ regular

$a^n b^n$ = Subset of Regular L may not be Regular.

e.g. ab

→ Every finit Language is regular

→ Finite union of Regular → Regular
Infinit union → Irregular

$a^1b^1 \cup a^2b^2 \cup a^3b^3$ ——— ∞

regular | irregular

① $i<j$ | $j<k$ ③
② $i>j$ | $i>k$ ④

Q If any write CFG

$$L = \{ a^i b^j c^k \mid i \neq j \text{ (or) } j \neq k,\ i, j, k \geq 1 \}$$

MS

$S \rightarrow ABC$

$A \rightarrow aAC$

$B \rightarrow$

→ 2 comparison but with 1 condition at a time

→ Prove either one

Sol

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$C \rightarrow cC \mid c$

$D \rightarrow aDb \mid ab$

$E \rightarrow bEc \mid bc$

Dry run

$$S \rightarrow \underset{i>j}{ADC} \mid \underset{i<j}{DBC} \mid \underset{k>j}{AEC} \mid ABE$$

(2)

$(a^n b^n c^n)$ = Any two are not equal (above eg)

Note

→ complement of CFL may not be CFL

(Start of compiler)

Q1 Give CFG for

L = { set of all arithmatic expression over alphabet ID (identifier) }

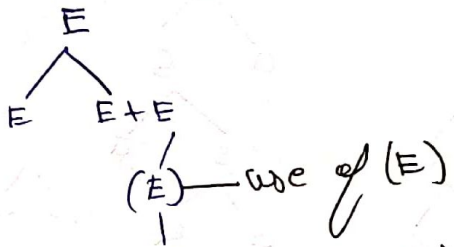
eg
id + id
id + id − id
id + id + id
(id + id) * id

(id, +, −, *)
{ id (+ −) * id }*

E → id | E + E | E * E | E / E | E − E | (E)

Any string not satisfying this will be Syntax error in C

eg Id + (id − id)



use of (E)

Q set of all Regular expressions over alphabet a, b)

L = { }

eg
a + b
a . b
a . b* + a
a . (b + a)

ns
E → Ea | Eb | (E) | E + E |

Sol
R → ε | a | b | R + R | R . R | R* | (R)*

Q. $L = \{$ set of boolean expressions over alphabet 0 or 1 $\}$

Ans: $B \rightarrow 0 \mid 1 \mid B+B \mid B \cdot B \mid \overline{B} \mid (B)$

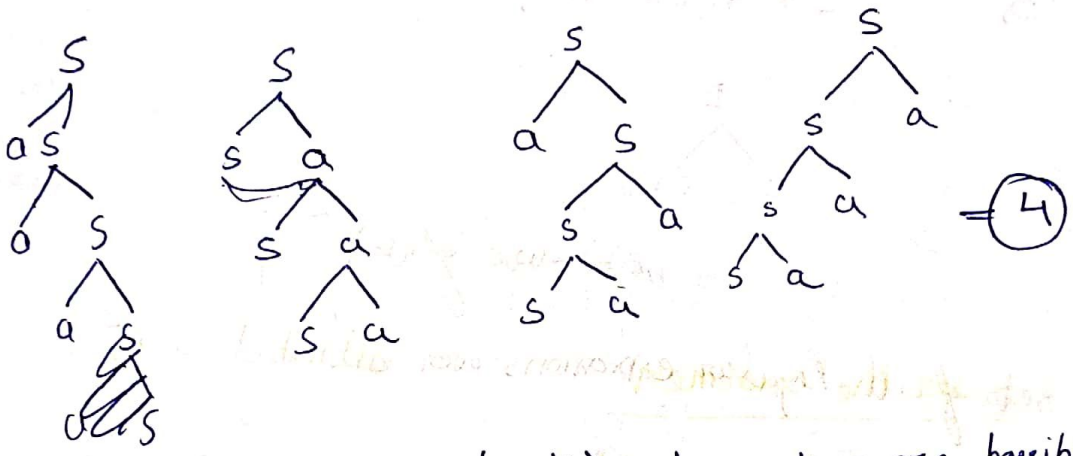
0+1
0

→ more than no + or ε

Q. Consider

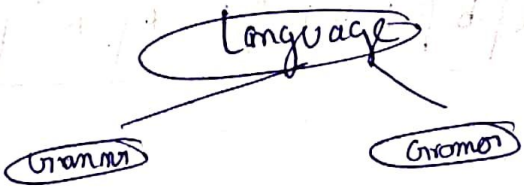
$S \rightarrow aS \mid Sa \mid a$

i/p: aaa $ → How many parse tree possible to generate this string.



∴ Ambiguous Grammer ⇒ multiple parse tree are possible
for atleast one string more parse tree

→ Checking a grammer if Ambiguous grammer undecidable
Problem since all the possible infinit possibilities it is to be checked.



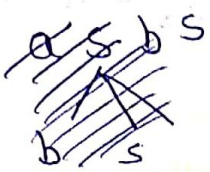
For language to be unambiguous

↳ min one grammer ambiguous.

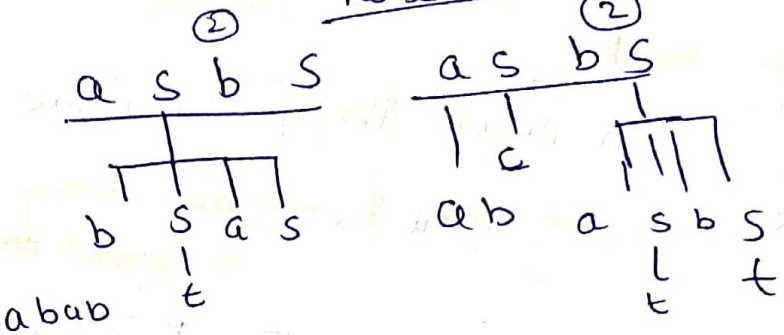
e consider the following grammer

$S \rightarrow aSbS \mid bSaS \mid t$

ip : abab $



Parse tree = 2



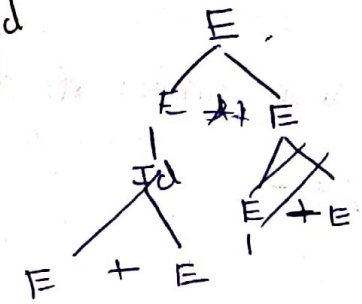
→ ambiguous grammer

Note There is no algo to verify if given Gr is ambiguous or not so it is an undecidable problem.

e Consider the Grammer

$E \rightarrow E+E \mid E*E \mid Id$ (No Priority)

ip : id + id * id



Total Parse = 2

// ambiguous

// Programming Language unambig

// else in gramer

// If any grammer exist unambiguous then — t

Conversion from Ambigious grammer to unambigious

to convert like c Lang compiler.

Ambigious

$E \rightarrow E+E / E*E / t$ (unambigious)

ip: ~~t2 "(1+2*3+4)~~ i:p: id+i+id+id

ip = id + id + id

(2) (1)

↓ (In Parse tree giving lower duels more high priority) ~~above~~ or.

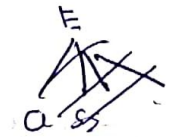
unam

$E \rightarrow E+T / (T) / E-T$

$T \rightarrow T*E / F / T/F$

$T = id / (E)$ ~~(E)~~ not ambigious

— Introducing Associativity



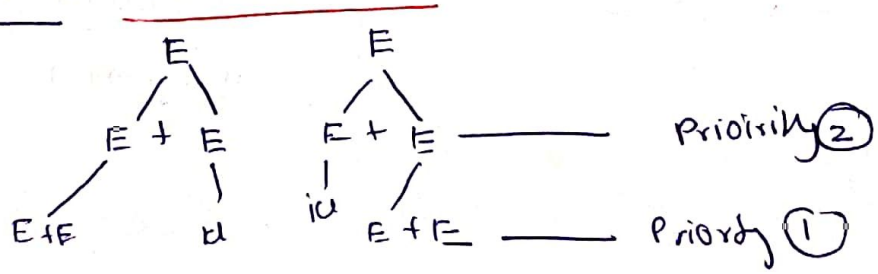
→ Grammer will decide Priority in comparison.

→ PL unambigious

→ Single Gramer Production cannot decide priority

$E \rightarrow E+E / E+$

on C id+id+E



$E = E + +$

Left to Right Associativity

$E = T + E$

Right to Left Associativity

Q Give unambiguous context free grammar for the following rules:

Ans	①	+ ⊙	Highest — L-R
	②	* $	R-L
	③	/ ⊕	R-L
	④	−	L-R
	⑤	↑ #	Lowest — R-L

Ans

Lowest: $A \rightarrow E$

$E \rightarrow E + T \mid E \odot T \mid T$

$T \rightarrow T * F \mid \$$

$F \rightarrow S \oplus F \mid S * F$

$S \rightarrow S - Q$

Highest: $Q \rightarrow Q \uparrow S \mid S \# Q \mid S$

Sol

$A \rightarrow B \uparrow A \mid B \# A \mid B$

$B \rightarrow B - C \mid C$

$C \rightarrow D / C \mid D \oplus C \mid D$

$D \rightarrow E * D \mid E \$ D \mid E$

$E \rightarrow E + F \mid E \odot F \mid F$

$F \rightarrow id \mid (A)$

Q Type

① Look for start symbol (Lowest Priority)

② Sequence Highest → Lowest

③ R → L or L → R

Q. Give unambigious CFG L = { set of all boolean expressions }

NOT
AND (L-R)
OR

Ans

Highest ~~B~~

$S \rightarrow \overline{S} \mid E$

$E \rightarrow E \cdot R \mid R$

$R \rightarrow R + T \mid T$

$T \rightarrow$ ~~id~~ $0 \mid 1 \mid (B)$

Lowest

Ans

$B \rightarrow B \text{ or } C \mid C$

$C \rightarrow C \text{ and } D \mid D$

$D \rightarrow$ ~~B~~ not $D \mid E$

$E \rightarrow 0 \mid 1 \mid (B)$ — Bracket

Q. Give unambigious CFG L = { set of all regular expressions } over the alphabet A, B

Ans

* Highest
. (L-R)
\+ Lowest

$S \rightarrow S + A \mid A$

$A \rightarrow A . B \mid B$

$B \rightarrow B^{*}$ ~~B~~ $\mid C$

$C \rightarrow id \mid b \mid \epsilon \mid (R)$ — Bracket

Note

$S \rightarrow aS \mid Sa \mid S$

→ converting any ambigious → unambigious undecidable because there are no operators and there are millions of productions. But for small grammer.

$S \rightarrow aS \mid$ ~~S~~ ~~A~~ $\mid$: Eliminate one

$S \rightarrow aS \mid A$

→ Those | Inherent unambiguous grammar | from which ambiguity cannot be removed

eg $S \to S_1 \mid S_2$

$S_1 \to a^m b^n b^n$ | string → | aaabbbccc | Both satisfy

$S_2 \to a^m b^m c^n$ | will make both unambiguous

→ To convert ambiguous to equivalent unambiguous is undecidable Problem / Cannot be converted.

→ Every regular grammer need not be ambiguous.

ex $S \to aA \mid a$

$A \to aA \mid \epsilon$

Recursion

→ Production symbol calling itself

Left $A \to A\alpha$ (is left Recursion)

① $A \to A\alpha \mid \beta$ (Left Recursion)

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

$\beta \in T^a$

A()

{ A()

}

Right

① $A \to \alpha A \mid \beta$ (Right Recursion)

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

$\beta \in T^a$

② Bcs of Left Recursion on of the top down parser one (Recursive descent) Parser will go to infinite loop so we have to eliminate left Recursion (LL)

① Bcs of Right Recursion no problem so no need to eliminate. (LR)

Elemination of Left - Recursion (Rewise)

ex1 $S \rightarrow Sa \mid b\varepsilon$ [b, ba, baaaaa, ...] $\Rightarrow ba^* = b\{a\}$

backwards normal form

$S \rightarrow bS'$ — ε for this

$S' \rightarrow \varepsilon \mid aS'$

ex2 $E \rightarrow E+T \mid TF'$ [$E \rightarrow E+T+T \mid (E+T)+T \mid$

$T \rightarrow T*F/F$

$F \rightarrow (\varepsilon) \mid id$

ms $S \rightarrow E \rightarrow T+E'$

Soln

E [$E \rightarrow TE'$, $E' \rightarrow \varepsilon \mid +TE'$] — $E \rightarrow TF' \mid T$, $F' \rightarrow +T \mid +TE'$

T [$T \rightarrow FT'$, $T' \rightarrow \varepsilon \mid FT'$

F [$F \rightarrow (\varepsilon) \mid id$

→ Sometimes in options removal of left recursion will give different options.

retry

Eliminate null production

Put ϵ in E'

ex

$E \to TE' / T$

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

$\Rightarrow$

$E \to TE' / T$

$E \to +T / +TE$

---

ex

---

ex $S \to (L) / a$

$L \to L,S / S$

$\Rightarrow$

$S \to (L) / a$

$L \to SL'$

$L' \to \epsilon / SL'$

---

ex $S \to Sa / Sb / Sc / d / e / f$

$S \to ds' / es' / fs'$

$S' \to \epsilon / as' / bs' / cs'$

---

ex $S \to Sa / Sb / Ac / d / e$

$A \to Sa / b / c$

$S \to Sa / Sb / d / e / Sac / bc / cc$

$S \to ds' / es' / bcs' / ccs'$

$S' \to \epsilon / as' / bs' / acs'$

Elimination of left recursion not only means left recursion directly but indirectly also

3 left recursion

2 direct

1 - indirect

Left Factoring

$S \rightarrow ab \mid ac \mid ad \mid e \mid f$

i/p → ad

→ From above grammer to generate string 'ad' three productions are fighting because in input 'a' — $E \begin{matrix} ab \\ ac \\ ad \end{matrix}$ gives first match.

this problem is known as left factoring.

⇓ Elemination of left factoring

ex1

$S \rightarrow e \mid f \mid aS'$

$S' \rightarrow b \mid c \mid d$ (~~called~~ left factored grammer)

ex2 $S \rightarrow iEts \mid iEtses \mid a \mid b$

ans $S \rightarrow i \mid iEtsS' \mid a \mid b$

$S \rightarrow \varepsilon \mid \varepsilon S$

ex3 $S \rightarrow a \mid ab \mid abc \mid abcd \mid e \mid f$

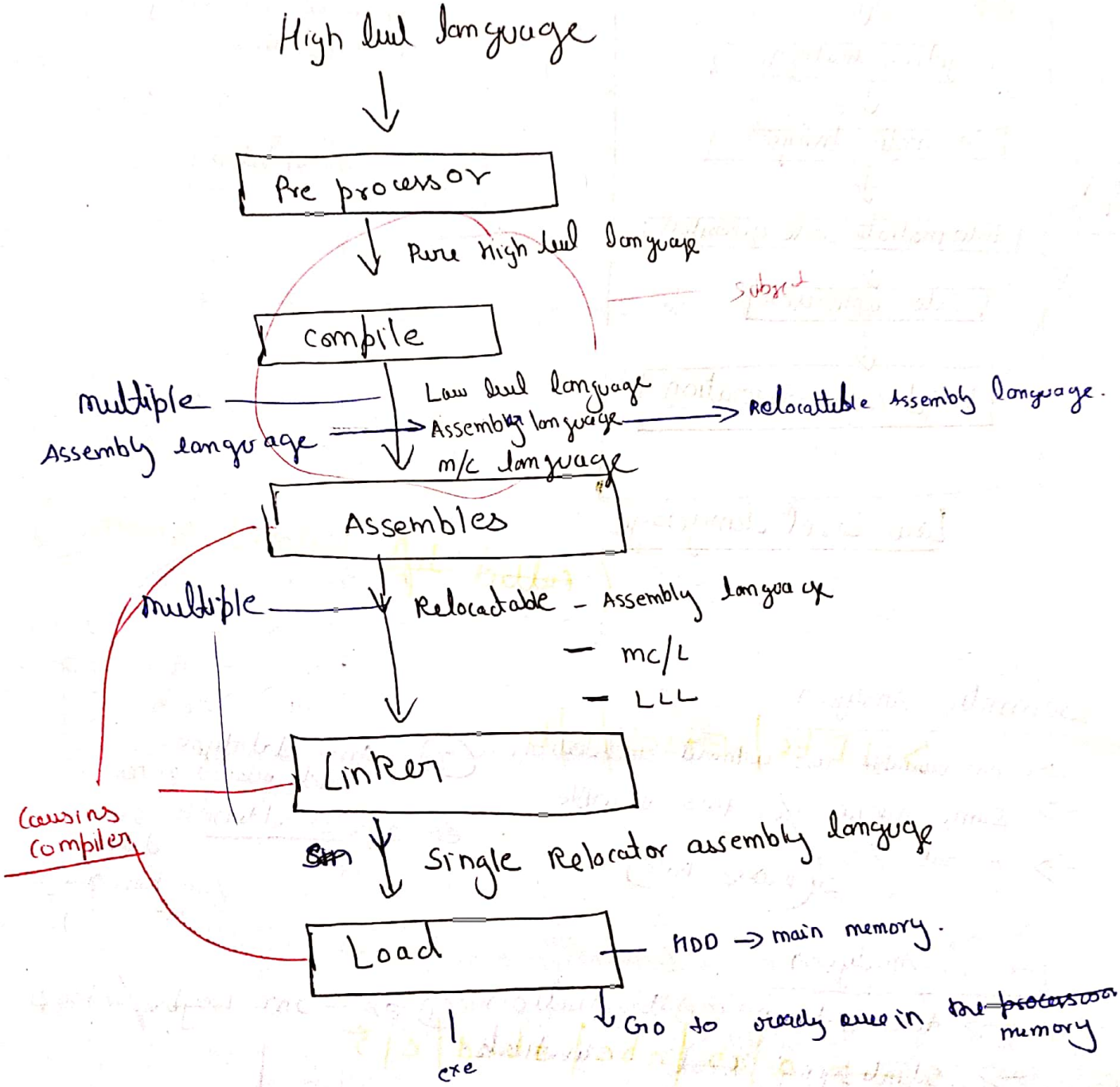
$S \rightarrow aS' \mid e \mid f$

$S' \Rightarrow \varepsilon \mid b S'' \mid$

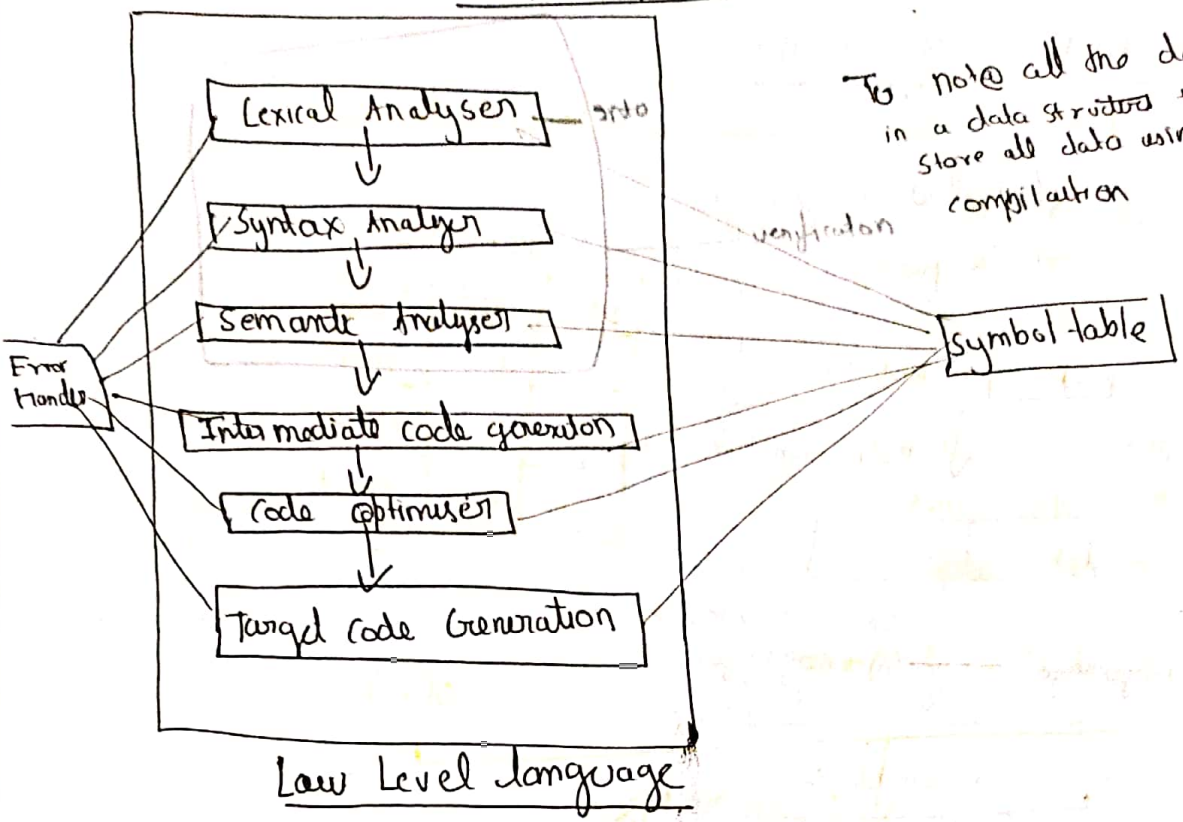
$S'' \rightarrow \varepsilon \mid cS''' \mid$

$S''' \rightarrow \varepsilon \mid d$

1) Introduction to compiler



Compiler HLL



Low Level language

To not@ all the data in a data structure to store all data using compilation

Semantic Analyser

→ no variable use without declaration
→ same name for two variable
→ o out of scope → (parse tree)

Lexical Analyser

→ tokenisation, introduction
→ identifying ~~data~~ individuals.

Syntax

→ syntax checking

(1) ex – in keyword for loop

(2) checking if valid loop syntax or not

(3) same datatype and increment or decrease

ex like in addition = same datatype for both integer +

CG

```
x = a+b * 60
float a, b, x
```

↓

Lexical Analyser

x - id - (id,1)
= - opn (Assignment)
a - idn - (id,2)
+ - opn - (B.Add)
b - idn - (id,3)
* - opn - mul
60 - int const

Symbol table (either all store or semantic)

S.NO	Variable Name	Variable Type
1	x	float
2	a	float
3	b	float

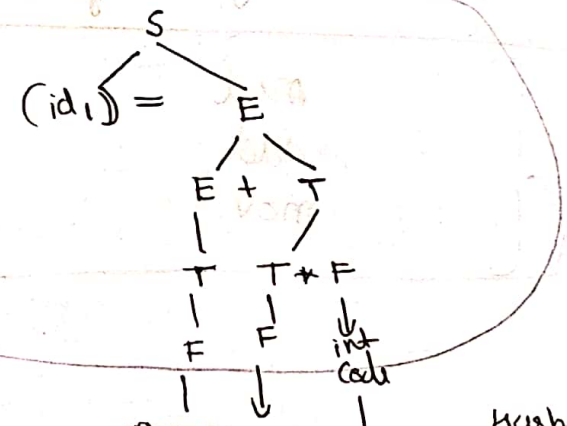
Identifier & literal info in are store

(id,1) = (id,2) + (id,3) * 60

O(n)

↓ Syntax Analyser

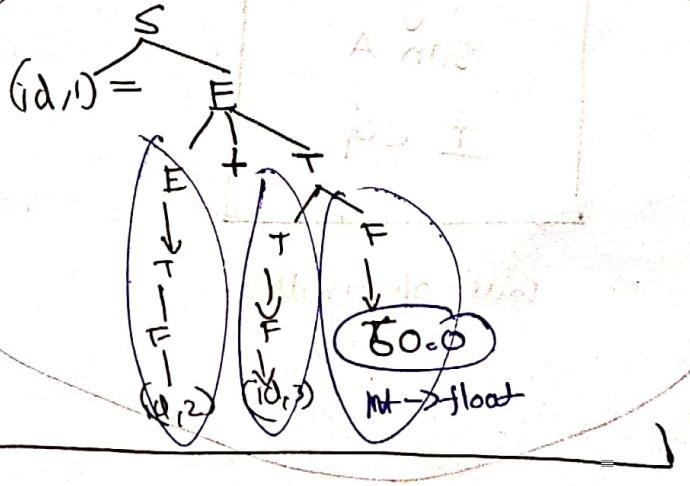
S ⇒ id = E
E → E + T / T
T → T * F / F
F → id / intconst



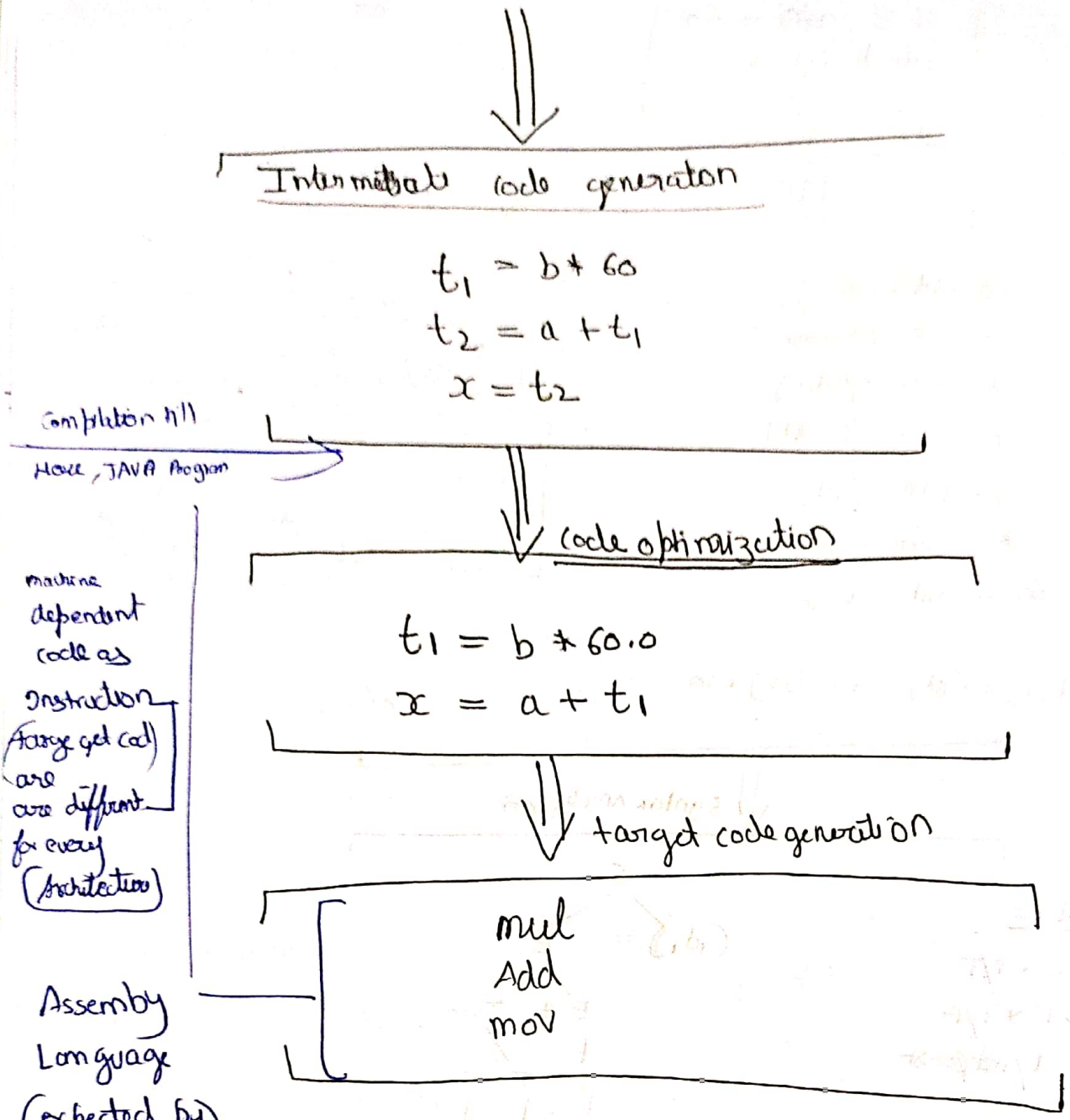
Hashing across id

(id,2) (id,3) (60)

↓ Semantic Analyser



Gate Scope



Front End

L.A
Sy. A
Sem A
I CG

optional — code optimization

→ If type of variable missing in Symbol table
(error: undeclared variable).

→ If variable or identifier is encountered it is stored in Symbol table and token number is given to it.

Symbol table is used by semantic analyser (2) else if in option else all can store (1)

(2) May all run in parallel as soon as encountered by error it goes to semantic SY

→ Symantic = type mismatch, diff type, variable undeclared.

→ Tokenisation: breaking into parts and assigning token number.

→ All 3 anlyses are done at a time it is difficult to seprate.

→ compiler runn all at ance and shows all error

→ Interpreter stop on error and does line by line.

→ If code size is big and main memeory i small then code is passed in passes one by one by a multi ~~root~~ to pass compiler

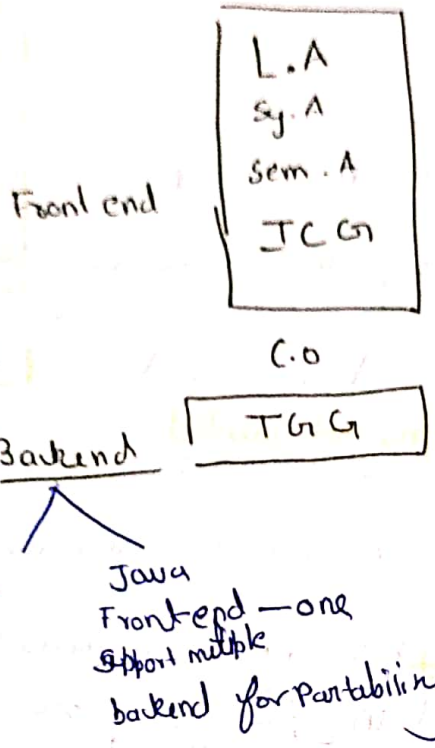
less space, more time.

there is a single pass compiler also

draw — more space
— less time

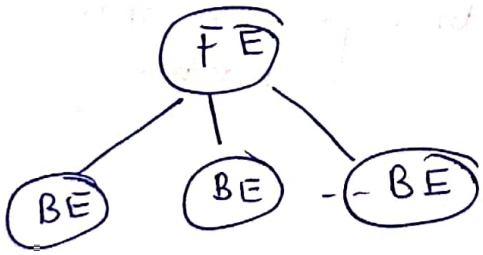
27th Oct

-> From one processor to another instruction set varies
-> In Java compiler does compilation up to intermediate code generation which make it platform independent
-> Java program running takes more time on C programming time while compilation takes less in C.



Time

Less Front end — Ja

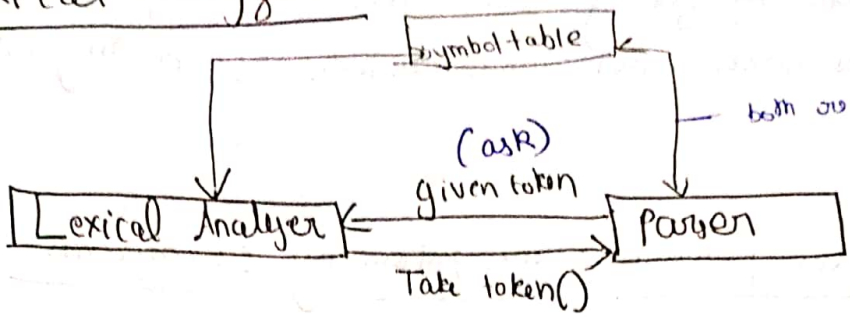


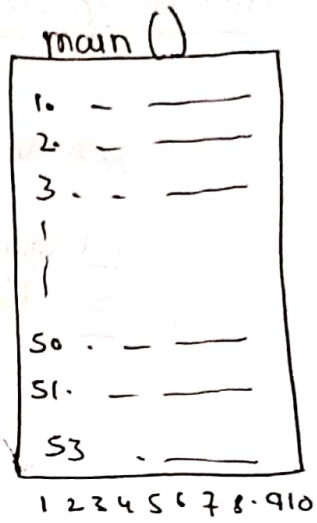
// Portability

40th

~~Multiplication~~ multiple declaration
syntax analysis
-> NO declaration
-> type mismatch

Lexical Analyser





Row = 53
Coln = 10

→ Lexical Analyser skip the comments

// in single line it skip till \n

/* till end of comment */

→ int a

2 tokens are there, Space not counted

→ Impⁿ + Tokenisation
+ Give row number and colⁿ number
+ Skip comment

Note → Lexical Analyser is first phase of compiler which is also known as scanner.

→ Function of Lexical Analyser
↳ It divides the given program into meaningful term (tokens)
→ Different types of tokens are
1) identifier type tokens.
2) Keyword
3) Operators
4) Constants
5) Special symbols

→ It will skip the comment lines, new line character, tabs blank spaces.

→ It will help to giving error messages by providing row number and column number. (for pointing out errors)

Q. Find no. of tokens present in the following C Program

```
//finding maximum element
int main()
{
    int a=10, b=20;
    if (a<b)
        return(b);
    else
        return(a);
}
```

All correct

// = 1
== 1

→ Longest matching is token

either (a, b, c) or (abc)

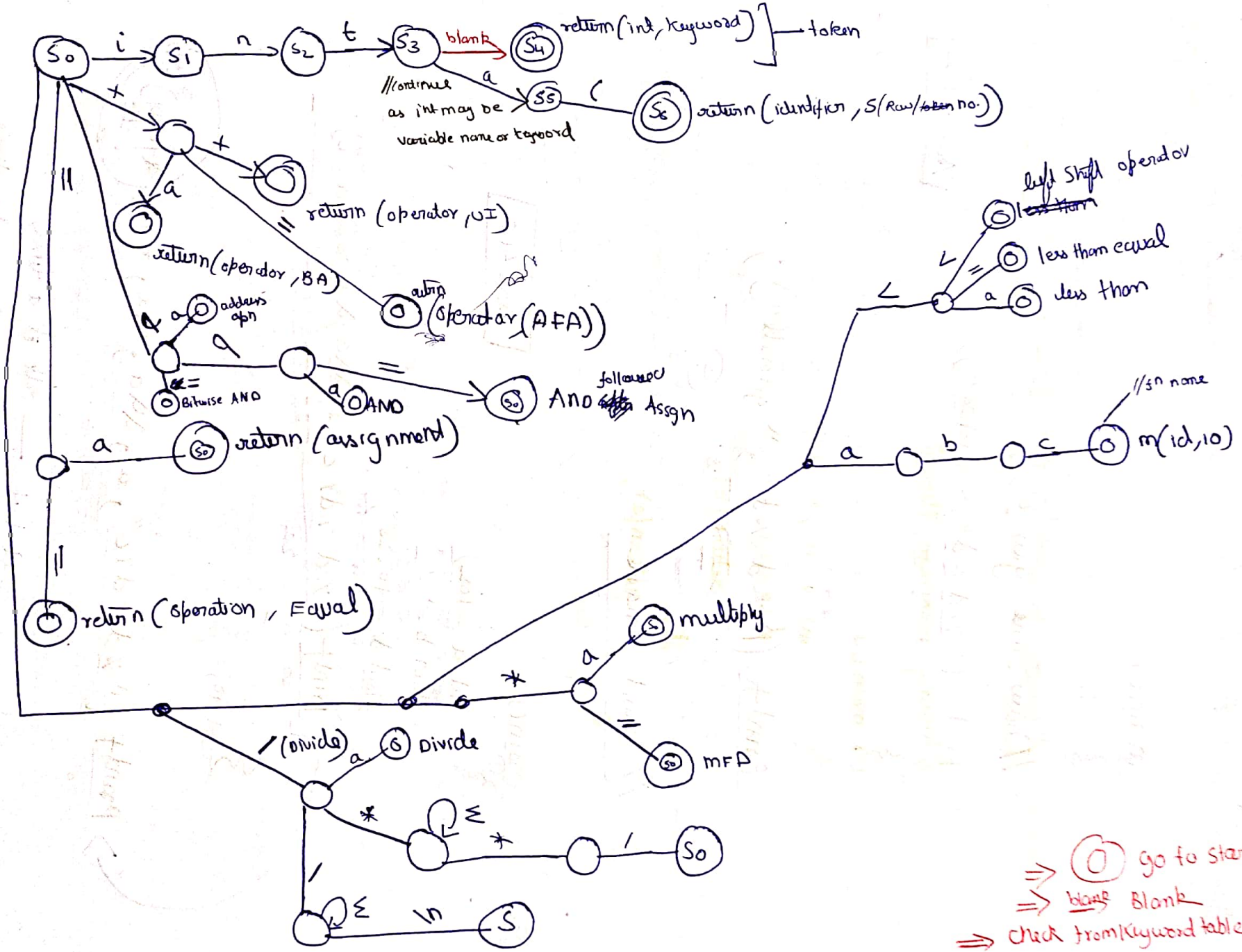
token not take if abc present

token — Longest matches

→ += is single token (ie short hand)

<int>
<

→ Every ( ) is individual token.



Q. Find no. of tokens for following

```
main()
{
// how are you ?
printf ("%d %d", "gotohell");
}
```

Ans: 17

// count single

Q.

```
main()
{
int a = 102;
char b = 'a';
float d = 10.25;
printf("%d %d %c", a, a++, a++, *a, a * b);
}
```

Ans 40

a++ — all at a time

**b — implemented one at a time

a++ — and * operator combined implemented

Q Prev question

ex

```
int a = 10
int *b = &a
int ** c = &b
```

a: 10 (1000)

b: 1000 (2000)

c: 2000 (3000)

when we find value of

* (* c) → 200 → 1000 → 10

Q

```
main()
{
    b = c+d;
    d = e-f;
    16
    c = a+++++-----===*/*+++---&&&/**/===&&==+-+-+-;
    pf(" %d %d ", a, b);
}
```

//comment not counted

&& = 1

- a &= b → a = a & b
- a &&= b → a = a && b

As 40

*/ ~~~~~ / = =

not counted

not linke or grouped together after skipping comment.

→ comments are not counted.

Ans 3 9

e)

```
main()
{
    int a = 10;
    int t a= 10;
    float b = 20;
    flo at b = 20;
    in /* comment */ t a = 10;
    flo /* comment */ at b = 20;
}
```

e) find no. of tokens in the following:-

```
main()
{
    char a = "String";
    float b = '4050';
    b = a + + + + + b /*/* /* * * * abc;
    Printf ("%d %d", a, b);
}
```

(b = a + + + + + b: 15 16 17 18 19 20 21; /*/* /* * * * abc; : 22 23 24 25 26 27; Printf: 28; Printf (" %d %d, a, b); : 29)

Lexical error

f)

```
main()
{ char a = "String";
  float b = '4050';
  Printf("%d %d", a, b);
}
```

(start, stop)

Parser (Syntax Analyser)

→ The process of deriving the string from given grammer is known as derivation (Parsing).

→ Depending upon how derivation is performed we have 2 types of parsers

1) Top Down Parser 2) Bottom up Parser

→ How top Down Parser will work

ex

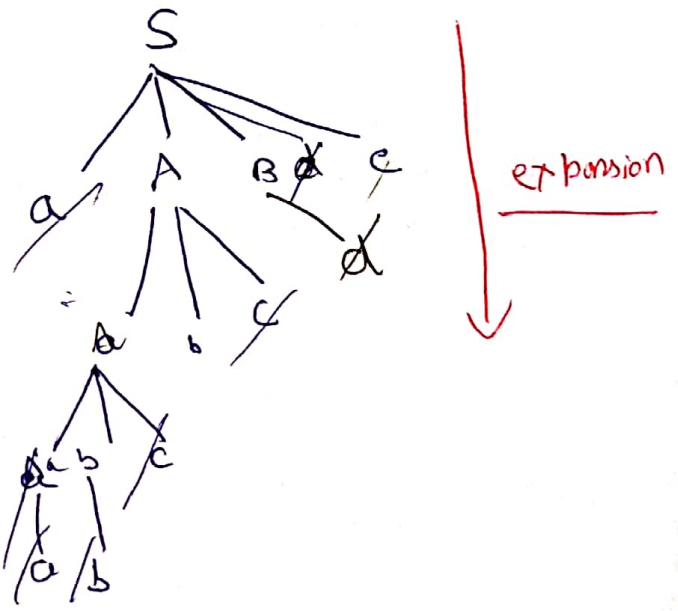
$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

ip : $a\,b_1\,b_2\,c\,d\,e$

Look ~~datermbot~~ Lookahead symbol



left most expansion

Note → Top down parser will start from ~~start~~ Start and processed to string.

→ It follows left most derivation

→ The difficulty of top down parser is selecting correct production if variable more than one possibility.

Working of Bottom Parser

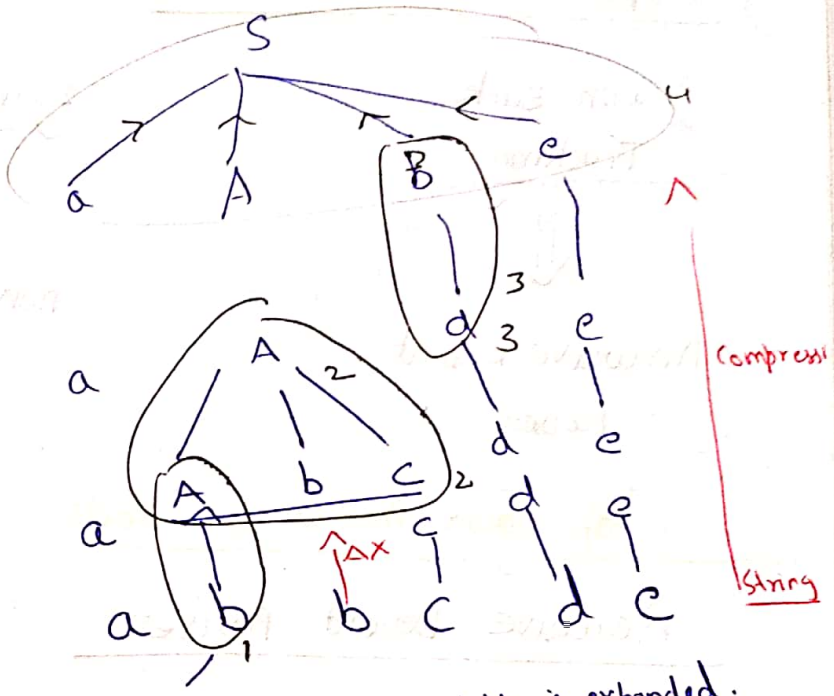
ex

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

i/p : abbcde $



4 S → aABe
3 B → d
2 A → Abc
1 A → b

∴ Handle : the term which is expanded.

// Bottom up parser dont have derivation it has compression.

i) Bottom up parser we start from string and proceed to start symbol.

ii) The difficulty in bottom up parser is identifying correct handle which will give start symbol

substring.

iii) It follows revers of right most derivation.

Top Down Parser

1) with Back Tracking ⇓ Recursive Descent Parser

2) without Backtracking ⇓ non recursive descent Parser or LL(1)

Recursive Descent Parser

```
S()
{
    Select only production of S (S → x1 x2 x3 ---- xk)
    for(i=1; i≤k; i++)
    {
        if (xi is variable)
            xi();
        else
        if (xi == lookahead symbol)
            increase i/p pointer
        else
            error [Try other possibilities]
    }
}
```

Difficulty: changing to any particular — then it will become LL(1)

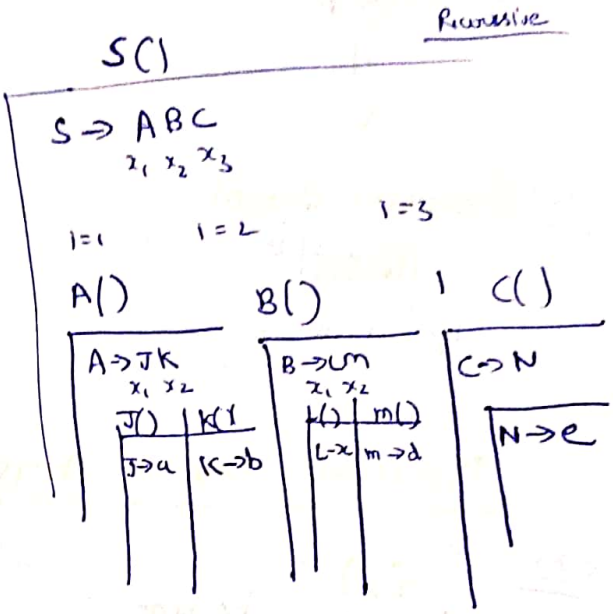
Ex1

$S \rightarrow ABC \mid DEF \mid GHI$

$A \rightarrow JK$
$B \rightarrow LM$
$C \rightarrow N$
$J \rightarrow a$
$K \rightarrow b$
$L \rightarrow c$
$m \rightarrow d$
$N \rightarrow e$

$D \rightarrow f$
$E \rightarrow g$
$F \rightarrow h$
$G \rightarrow i$
$H \rightarrow k$
$I \rightarrow j$

Recursive



i/p : abcde$

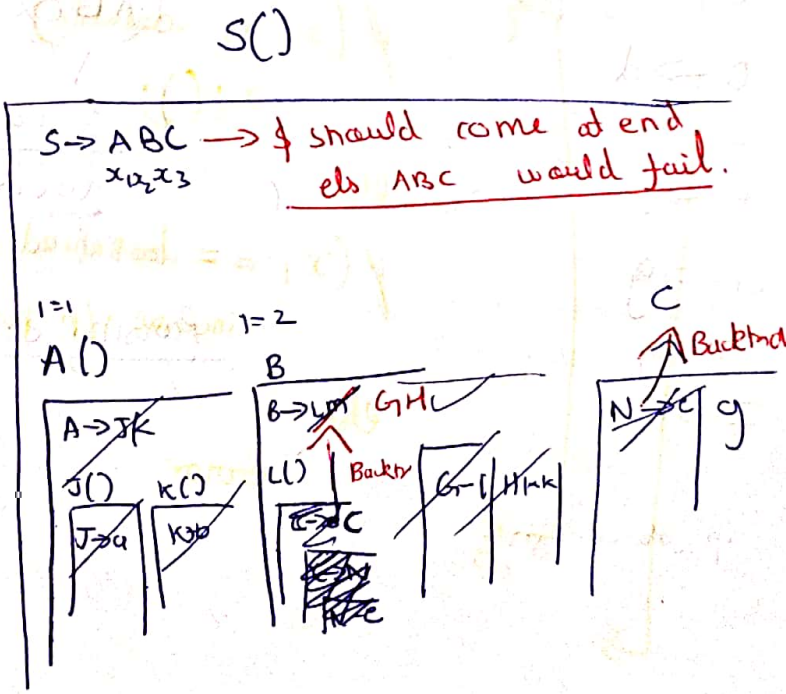
Data structure = Stack

∴ Stack and DFA → PDA

Ex2 $S \rightarrow ABC \mid DEF \mid GHI$

$A \rightarrow JK$
$B \rightarrow LM/GH$
$C \rightarrow N/E$
$J \rightarrow a$
$K \rightarrow b$
$L \rightarrow c$
$m \rightarrow d$
$N \rightarrow e$

$D \rightarrow f$
$E \rightarrow g$
$F \rightarrow h$
$G \rightarrow i$
$H \rightarrow k$
$I \rightarrow j$



i/p : abikg$

ex3

$S \rightarrow Sa|b$

i/p baaa $

S()

S

// Left Recursion will go into ∞ loop.

Stack overflow

ex4

$S \rightarrow aS|b$

ip: ~~daaab~~$

S()

~~a~~S

a S() — b()

b, 0

Backtrack

ex5

$S \rightarrow ABC|DEF|GHI$

$A \rightarrow a|ab|abc|abcd$

$B \rightarrow ef|ij$

$C \rightarrow gh|kl$

$D \rightarrow d$

$E \rightarrow e$

$F \rightarrow f$

$G \rightarrow g$

$H \rightarrow h$

$I \rightarrow i$

S

ABC ✗ DEF

If one gets cancelled all its work done is also cancelled

ip: abcdefghi$

Parsing error

1) In Recursive descent parser for every variable we are writing recursive function.

2) Recursion is using stack data structure

3) If grammer containg left recursion, recursive descent parser may go into infinite loop (stack overflow)

4) If the grammer contain left factoring it may give parsing error

5) Lot of time is wasted in backtracking. (worst case 2^N)

6

Non Recursive descent Parser (LL_1 parser)

-> In LL_1 parser there is no backtracking because we are using LL_1 table.

-> To construct LL_1 parsing table we need two function

1) First()
2) Follow()

1) First()

First of S() = Set of all terminals present in first place of every string derived by S.

ex1 $S \rightarrow abc \mid def \mid ghi$

First(S) = Fi(abc), Fi(def), Fi(ghi)

= Fi(a), Fi(d), Fi(g)

= a, d, g

ex2 $Fi(t) = t$ — terminal

$Fi(\varepsilon) = \varepsilon$

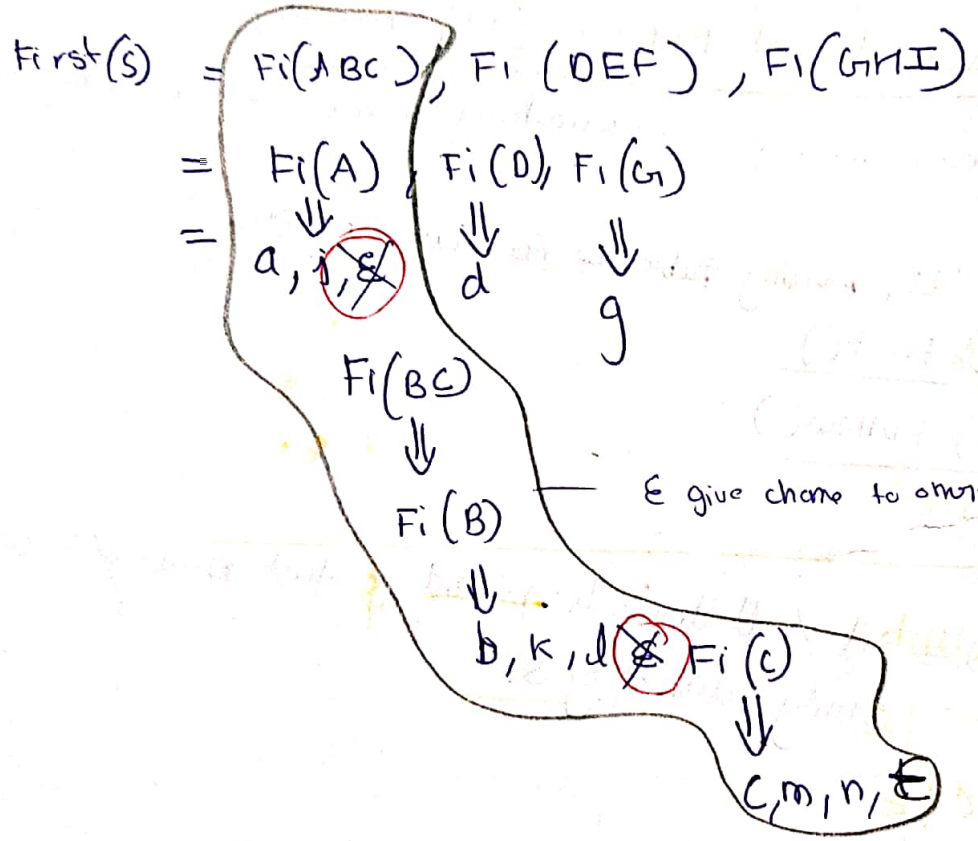
ex3

$S \rightarrow ABC \mid DEF \mid GHI$

$A \rightarrow a \mid j \mid \varepsilon$ $D \rightarrow d$ $G \rightarrow g$

$B \rightarrow b \mid k \mid l \mid \varepsilon$ $E \rightarrow e$ $H \rightarrow h$

$C \rightarrow c \mid m \mid n \mid \varepsilon$ $F \rightarrow f$ $I \rightarrow i$



ex4 $E \to TE'$

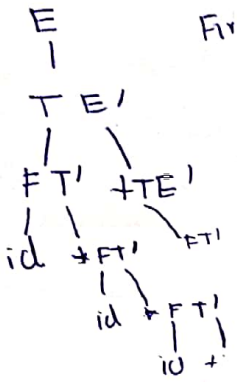
Rules $E' \to \varepsilon \mid +TE'$

$T \to FT'$

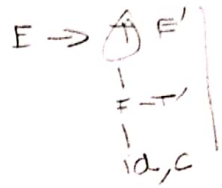
$T' \to \varepsilon / *FT'$

$F \to id/(E)$

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



First(E) = First (T, E)
= First (T)
F T'
(id, ε)
*F T'



$E' \to \varepsilon +$
$T \to F T'$
$T' \to \varepsilon, *$
$F \to id, ($

ex5 $S \to (L)/a$

$L \to SL'$

$L' \to E/,SC'$

	(, a
S	(, a
L	(, a
L'	ε, ,

(a
L → SL'
(a
L' → ε, ,

ex Find Follow

$S \to aBDh$ —— a —— $

$B \to cC' \mid \varepsilon$ —— c, ε —— Fo(B)-

$C' \to bC \mid \varepsilon$ —— b, ε

$D \to EF$ —— f, ε, g

$E \to f \mid \varepsilon$ —— f, ε

$F \to g \mid \varepsilon$ —— g, ε

e) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$ — ε

$B \rightarrow \varepsilon$ — ε

MS $S \rightarrow a, b$ (Left recursion)

e) $S \rightarrow asbs \mid bsas \mid \varepsilon$

$S \rightarrow a, b, \varepsilon$

Follow ()

$S \rightarrow ABC$

$A \rightarrow DEF$

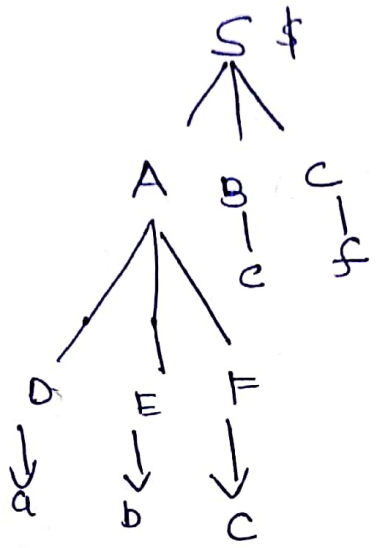
$D \rightarrow a$

$E \rightarrow b$

$F \rightarrow c$

$B \rightarrow e$

$C \rightarrow f$



Follow(S) = Set of all terminals that may follow it immediately right of S

e) $Fo(D) = Fi(EF)$

$\Downarrow$

$Fi(E)$

$\Downarrow$

b

$Fo(F) = Fo(A)$

$= Fi(BC)$

$\downarrow$

$Fi(B)$

$\downarrow$

ε

$Fo(C) = Fo(S)$

$= Fi(\$)$

$= \$$

Follow()

Ex2

S → ABd

A → DEF

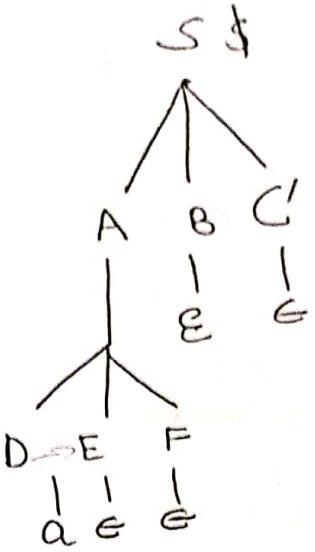
D → a

E → ε

F → ε

B → c

C → ε



1) Fo(D) = Fi(EF)

⇓

Fi(E)

⇓

~~ε~~ Fi(F)

⇓

~~ε~~ Fo(A)

⇓

Fi(Bc)

⇓

Fi(B)

⇓

c

2) Fo(B) = Fi(C)

⇓

~~ε~~ Fo(S)

⇓

Fi($)

⇓

$

E F

| |

BC

ε

ex③

$S \rightarrow ABC'$

$A \rightarrow DEF$

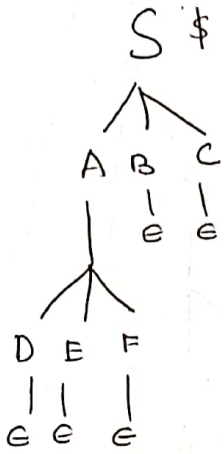
$D \rightarrow \varepsilon$

$E \rightarrow \varepsilon$

$F \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$



Fo(D) = Fi(EF)
⇓
Fi(E)
⇓
~~ε~~ Fi(F)
⇓
~~ε~~ Fo(A)
⇓
Fi(BC)
⇓
Fi(B)
⇓
~~ε~~ Fi(C)
⇓
~~ε~~ Fo(S)
⇓
Fi($)
⇓
$

Rules to find follow:

never give ε

① If S is start symbol
Fo(S) = $

② $S \rightarrow ABC'$, $B \rightarrow d$

Fo(A) = Fi(BC')
= Fi(B)
= d

③ $S \rightarrow ABC'$ (or) $S \rightarrow ABC'D$
$D \rightarrow \varepsilon$

Fo(C) = Fo(S)

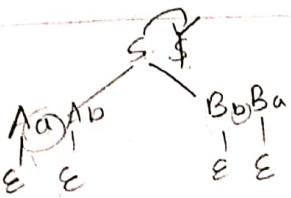
③ Find that symbol a occurance mai Right side in production

ex④ $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

	Fi( )	Fo( )
S	a, b	~~ε~~ $
A	ε	a, b
B	ε	b, a



ex5
Retry

$S \to aSbS \mid bSaS \mid \varepsilon$

	Fi()	Fo()
S	a, b, ε	$, a, b

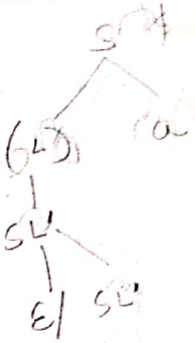
ex6
Retry

$S \to (L) \mid a$

$L \to SL'$

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

	Fi()	Fo
S	(a	$,), ,
L	(a	)
L'	ε	)
	ε, ,	



Return place can be put nu

ex7
Done

$E \to TE'$

$E' \to \varepsilon / +TE'$

$T \to FT'$

$T' \to \varepsilon / *FT'$

$F \to id \mid (E)$

	Fi	Fo()
E	id, (	$)
E'	ε, +	$)
T	id, (	+ $)
T'	ε *	+ $)
F	id, (	* + $)

Fi(E) → TE'
T → FT'
F → (E)
E' = ε, +

Follow
$F(E') =$ Right E' not exist then check E' to parents which child is E' → E and then any right occurs of E

workbook

Q24 $A \to BA'$

$A' \to +BA' \mid \epsilon$

$B \to CB' \mid \epsilon$

$B' \to *CB' \mid \epsilon$

$C \to (A) \mid id$

Ans $First(A) = First(BA')$

$First(B)$

$First(C)$

$(, id$

$First(A) \to First(BA')$

$First(B)$

$(, id$

$First(A) \to (B\ A')$

$First(B)$

$First(CB' \mid \epsilon)$

$+, id, \epsilon$

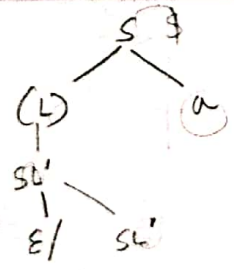
ϵ

$First(A) \to First(B\ A')$

$C, \epsilon, *, \epsilon$

$+(id, *)$

R.C



	First	Follow
S	(a	$, a
L	(a	)
L'	ε	)

28/10/19

LL(1) – Parsing Table construction Algo

For each production $A \to \alpha$ repeat following steps

(1) Add $A \to \alpha$ under $m[A, \alpha]$ (A – variable, α – terminals)

$\forall a \in Fi(\alpha)$ First(α)

(2) If First(α) containing $-\epsilon$ then add $A \to \alpha$ under $m[A, b]$ $\forall b \in$ follow(A)

---

ex construct LL(1) Parsing table for the following grammer

(1) $S \to (L) / a$

$L \to SL'$

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

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

m	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→ε		L'→,SL'	

3 rows, 5 columns (LL(1) – PT)

ϵ – is not included bcs its not a character on input string but $ is an character on input string bcus it is present in table

Note:

If → Given grammer is LL(1) bcs in LL(1) Parsing table no entry contain multiple productions.

→ Error Entry = Blank Spaces

Q check if the following grammer is LL(1) or not

Ans

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow \epsilon / *FT'$

$F \rightarrow id / (E)$

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

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

Retry

8. Check the LL1 grammer

$S \xrightarrow{(1)} a\,B\,D\,h$

$B \xrightarrow{(2)} C\,C'$

$C' \xrightarrow{(3)} bC \;/\; \in \;\; (4)$

$D \xrightarrow{(5)} E\,F$

$E \xrightarrow{(6)(7)} f \;/\; \in$

$F \xrightarrow{(8)(9)} g \;/\; \in$

follow

Verify

On no variable is common the LL1 not cont
Ques contain large production then check
for common variable to check for LL1

	a	b	c	f	g	h	$
S	(1)						
B			(2)				
C'		(3)		(4)	(4)	(4)	
D				(5)	(5)	(5)	
E				(6)	(7)	(7)	
F					(8)	(9)	

follow

$C' \rightarrow f\ g\ h$

$D \rightarrow h$

E

(1) To check for LL1
only check for variable
with common not sure confir
ie C E F in this example

Q

$S \to \overset{①}{A}a\overset{②}{A}b \mid \overset{③}{B}b\overset{④}{B}a$

$A \to \varepsilon$

$B \to \varepsilon$

	First	Follow
S	ε, a, b	$
A	ε, b	a, b
B	ε	b, a

Ans

	a	b	$
S	①	②	
A	③	③	
B	④	④	

③ ∧ ③

Ex-4

$S \to a\overset{①}{S}bs \mid b\overset{②}{S}as \mid \overset{③}{\varepsilon}$

	a	b	$
S	(1) (3)	(2) (3)	

S(ε)

// (not LL1)

Checking: If grammer don't have null production

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$

$Fi(\alpha_1)$ $Fi(\alpha_2)$ $Fi(\alpha_3)$

a, b c, d e, f

$Fi(\alpha_1) \cap Fi(\alpha_2) = \phi$
and
$Fi(\alpha_2) \cap Fi(\alpha_3) = \phi$
and
$Fi(\alpha_1) \cap Fi(\alpha_3) = \phi$

Pair wise disjoint

If Grammer contains null Production

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$

$Fi(\alpha_1)$ $Fi(\alpha_2)$ $Fi(\alpha_3)$

a, b c, d e, f

$F_1(\alpha_1) \cap Fi(\alpha_2) = \phi$
and
$Fi(\alpha_1) \cap Fo(A) = \phi$
and
$F_1(\alpha_2) \cap Fo(A) = \phi$

Pair wise disjoint

LL(1)

Q check for LL(1) grammer.

	a
S	① ②

$S \to E / a$

$E \to a$

$Fi(E) \cap Fi(a)$

$\Downarrow$

$a \cap a = a$

not LL_1

Q Retry $S \to aABb$

$A \to a / \varepsilon$

$B \to d / \varepsilon$

	a	b
S		
A		
B		

$Fi(B) \cap F(\varepsilon)$

$\Downarrow \quad \Downarrow$

$d \quad b = \phi$

$Fi(A) \cap F(\varepsilon)$

$\Downarrow \quad \Downarrow$

$a \quad a \not= d$

$Fi(B\,b)$

$Fi(B)$

$d, b \quad \not\phi$

LL_1 ✓

Q Retry $S \to aSA / \varepsilon$

$A \to c / \varepsilon$

ms

$S \Rightarrow F(aSA)$

$\downarrow$

a

$F(\varepsilon)$

$S\ A$

$B\ \ C$

$Sa \quad \varepsilon$

Ans $Fi(aSA) \cap Fo(S)$

$\downarrow \qquad \downarrow$

$a \quad \$ \qquad Fi(A)$

$c \quad \not\varepsilon - Fo(S)$

$= \phi$

$Fi(c) \cap Fo(A)$

$\downarrow$

$c \cap Fo(S) = c$

$\downarrow$

$\$, c$

→ If the grammer contain ① left factoring it cannot be LL1

$S \to aS|a$

$a \cap a = a$ not (LL1) // Regular grammer but not LL(1).

Ex $S \to Sa|b$

Fi(Sa) → Fi(S) → Fi(Sa), Fi(b) → Fi(S) ∩ b / b

Fi(b) → b / not LL(1)

② → If grammer contain left recursion, there it cannot be LL(1).

Ex $S \to aSbS|bSaS|\epsilon$

a, b, Fo(S) → $a, b — intersection not (LL1)

③ Ambiguity

→ not(LL1) grammer are unambigious since ∩ not ∅ — converse not true — one side true.

Gate

Ex consider gram — LG — not LL bcz

a) Left Rec
b) R Recursion ③
c) R Left Factoring

a) Ambiguity — no ② ∴ All given A/1 ①

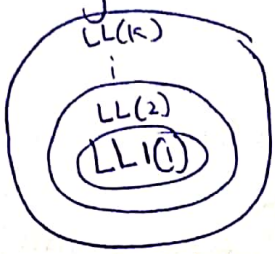
Gate pattern

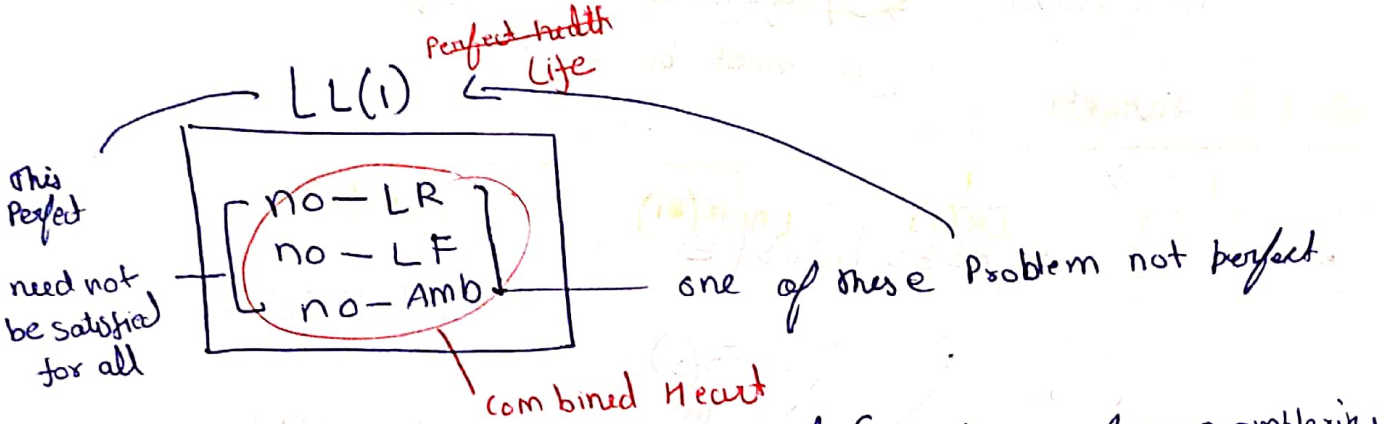
Every Regular Grammer may not be LL(1)
but every ~~LL(1)~~ Regular language is LL(1)

- S → ab | ac | ad

LL(1) ✗ (since a)

LL(2) ⊆ (Reading 2 at a time





i) Bottom up parser is more powerful (But at cost of more complexity

Bottom UP Parser (No Backtracking bcz every Parser Have table)

- L-R Parser
  - Need → Unambigious
- operator Precedence Parser
  - operator Grammer

Note → every LR is unambigious

→ every unambigious need not be LR

(LR unambigious)

LR Parser (shift reduce Parser)

- LR(0)
- LR(1)
- LALR(1)
- CLR(1) (more power more costly)

Note → For LR Parser Parsing algo same but parsing table different

LR Parsing Algorithm

Let X - be a state on Top of Stack

a - lookahead symbol

① If action[X, a] = Si then

Shift – a and increment i/p pointer

② If action [X, a] = rj and jth production is

$\alpha \rightarrow \beta$ then pop 2*|β| symbols and replaced by α

If S_{m-1} is one state below α, then push goto[S_{m-1}, A]

③ If action [X, a] = acc (accepted) then sucessful completion of parsing

④ If action [X, a] = blank then parsing error.

Ex ① S → AA ②
A → aA ③ | b ③

ip : abab$

	action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

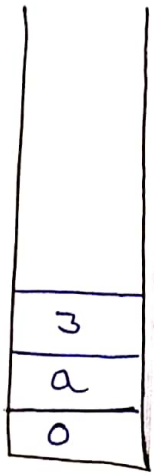
LR (0) – PT

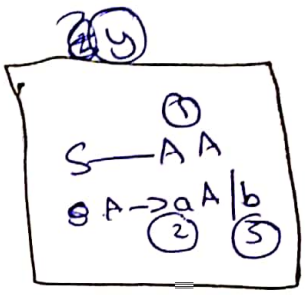
State no. come from constructing DFA

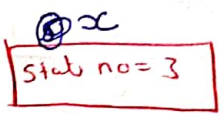
ex 4 Table same as previous

i/p :— aabb$

—a a b b $
↑ ↑
(Reduce not done on reduction)







i/p	stack	Reduction
~~a~~abb$	0 0 a3 "from table	
~~ab~~b$	0 a 3 0 a 3 a 3	
~~bb$~~	0 a 3 a 3 0 a 3 a 3 b 4 — from table reduce from 2c 0 a 3 a A 4	

404

aab$ 0a3a3b4 A

i/p : aab$

i/p	stack	Reduction
aab$	0 0a3	
ab$	0a3 0a3a3	
b$	0a3a3	
$	0a3a3b4 0a3a3A 0a3a6	A → b
$	0a3a6 0a3A	A → aA
$	0a3A 0a3A6	A → aA
$	0a3A6	A → aA
$	0A3 0 Blank	

0a3A
0a3A6
0A2

404

→ conflict will always come from Action Part

LR(0) Parsing table construction

→ To construct LR(0) parsing table we need two fn 1) closure()
2) Goto()

Defination

closure(I) = 1) Add I

2) If A → B.CDE is then expand C → FGH is in G then add C → FGH to closure(I)

3) Repeat 2nd step for every newly added LR(0) item

~~G~~ G: S → AA
A → aA/b
— Given

G' → S'
S' → AA
A → aA/b

G' = S' → S — Augmented Production
S → AA
A → aA/b
Augmented grammar

ex1 closure(S' → .S) = ① S' → .S

same copy, expand

② S → .AA

ex

③ S → .aA
.b

ex2 closure(S → .A) = ① S → A . A
② A → . aA
. b

ex3 closure(A → . a) = ① A → . a

B Goto()

Defn GoTO(I, x) = ① Add I by move dot after x
② ~~Error~~ Find closure (1st step)

ex1

Goto(S' → . S, S)

⇓

1. S' → S .

---

ex2 Goto(S → . AA, A) Skip it

① S → A . A expand
② A → . aA
. b

---

ex3 Goto(A → . aA, a)

⇓

① A → a . A
A → . aA
. b

LR(0) - Parsing Table construction Algo

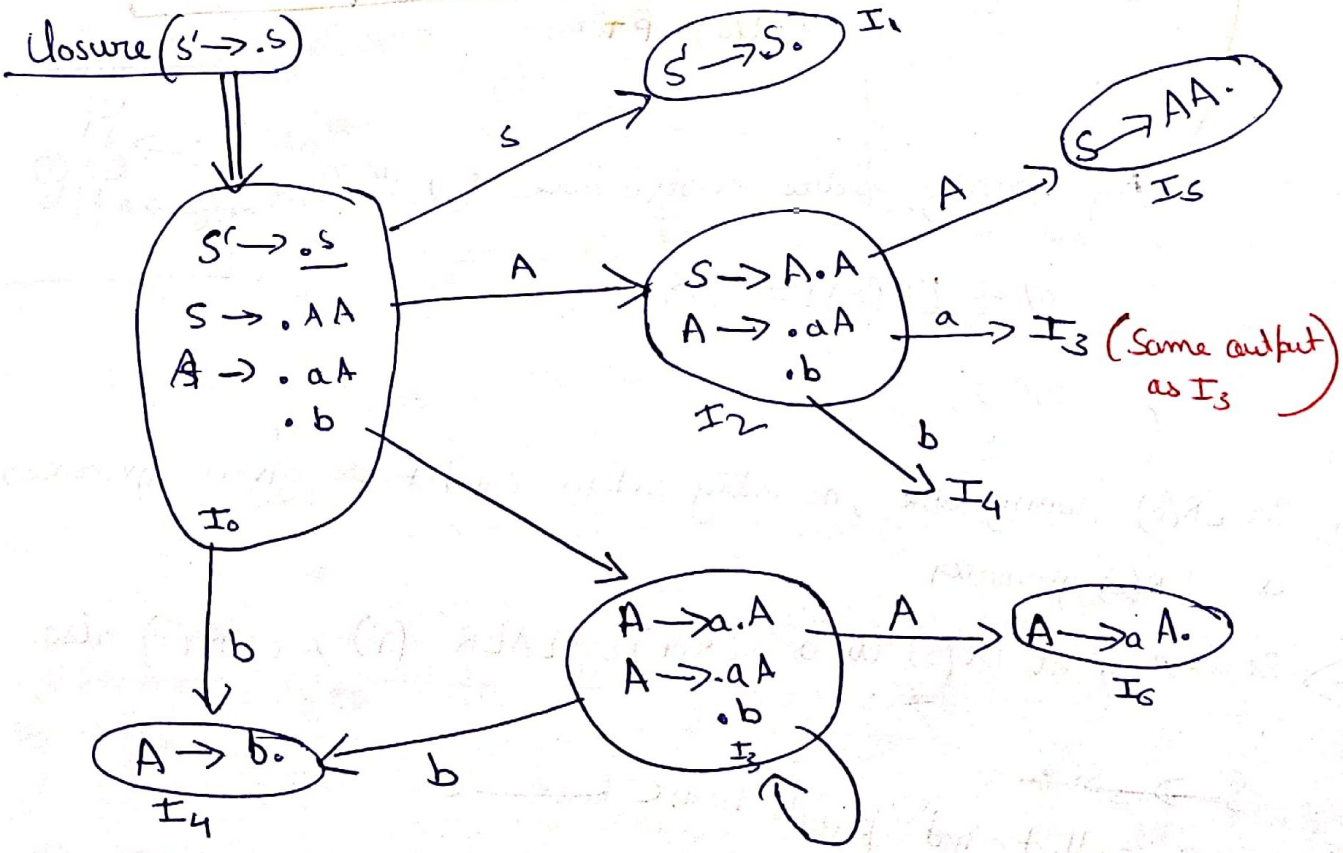
① I_0 = closure (Augmented LR(0)-item)

② using I_0 construct DFA

③ convert DFA into LR(0) - PT

ex(1)

$S \to AA$
$A \to aA|b$] Augmented grammar → $S' \to S$, $S \to AA$, $A \to aA|b$



	action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

LR(0) - PT

By writing reduce everywhere no problem will come.

$S \to AA$ ① ② ③
$A \to aA \mid b$

If $L(LR(0)) = SLR(1)$

1) In LR(0) Parsing table, no entry contain conflict so given grammer is LR(0) grammer.

→ Because of its LR(0) it is SLR(1), LALR (1), CLR(1) also.

~~$S \to SS$~~

SS conflict not possible beacs ~~base~~

State Conflict $\left(\begin{matrix}SR\\RR\end{matrix}\right)$ = min 2 productions, one of them reduce.

→ Every shift entry will come because of not completed

→ Every not ~~conflict~~ completed cant give shift.

Q1 If n_1 no. of reduce states in LR(0)

if n_2 ——————— SLR(1)

then relationship b/w n_1 and n_2?

Ans $n_1 \geq n_2$

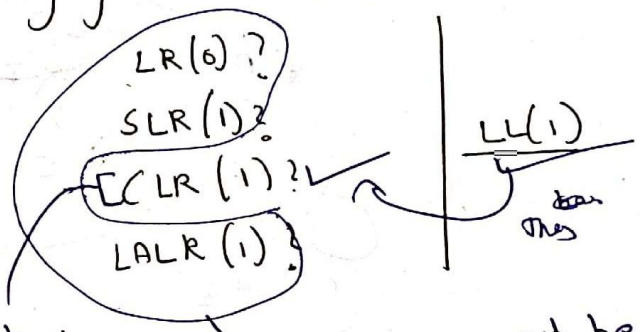
Q2 Check the following gammer is SLR(1) grammer or not.

$S \rightarrow dA/aB$

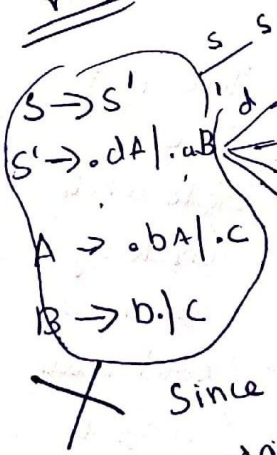
$A \rightarrow bA/c$

$B \rightarrow bB/c$

not in I_6

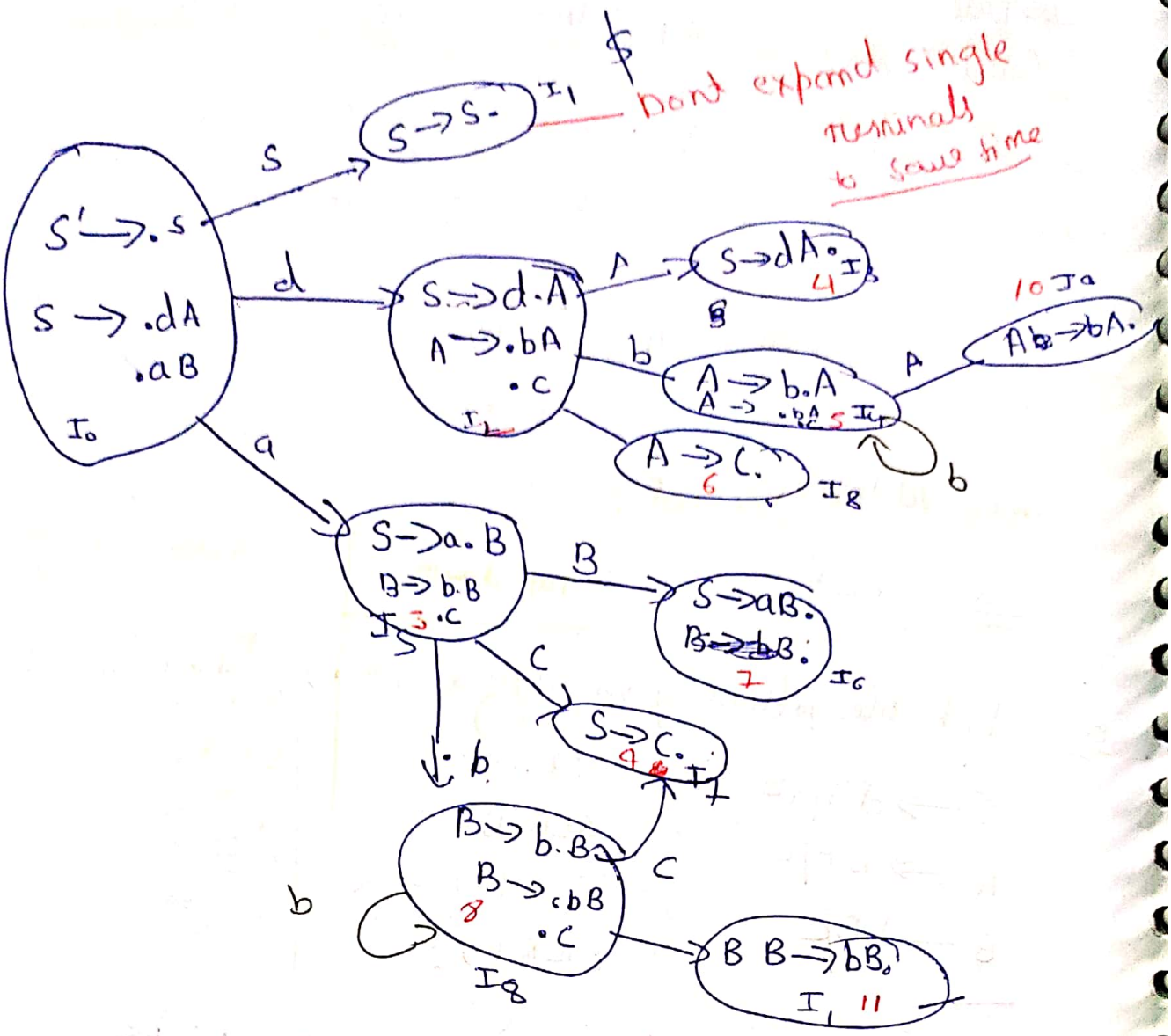


ms



1) check for LL(1) first to check for CLR(1)

by checking for intersection of ϕ in the production.



To check only

	Action					Goto		
	a	b	c	d	$	S	A	B
0	S_3			S_2		1		
1					acc			
2		S_5	S_6				4	
3		S_8	S_9					7
4	r_1	r_1	r_1	r_1	r_1			
5		S_5	S_6				10	
6	r_4	r_4	r_4	r_4	r_4			
7	r_2	r_2	r_2	r_2	r_2			
8		S_8	S_9					11
9	r_6	r_6	r_6	r_6	r_6			
10	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5			

Need not Draw Goto while checking as Goto dont have conflicts

⇒ Next time after drawing DFA dont draw ~~problem~~ table

Canal production having single production and starting Production.

⇒ min. two production, one is reduce
- no cancel
- yes draw chart

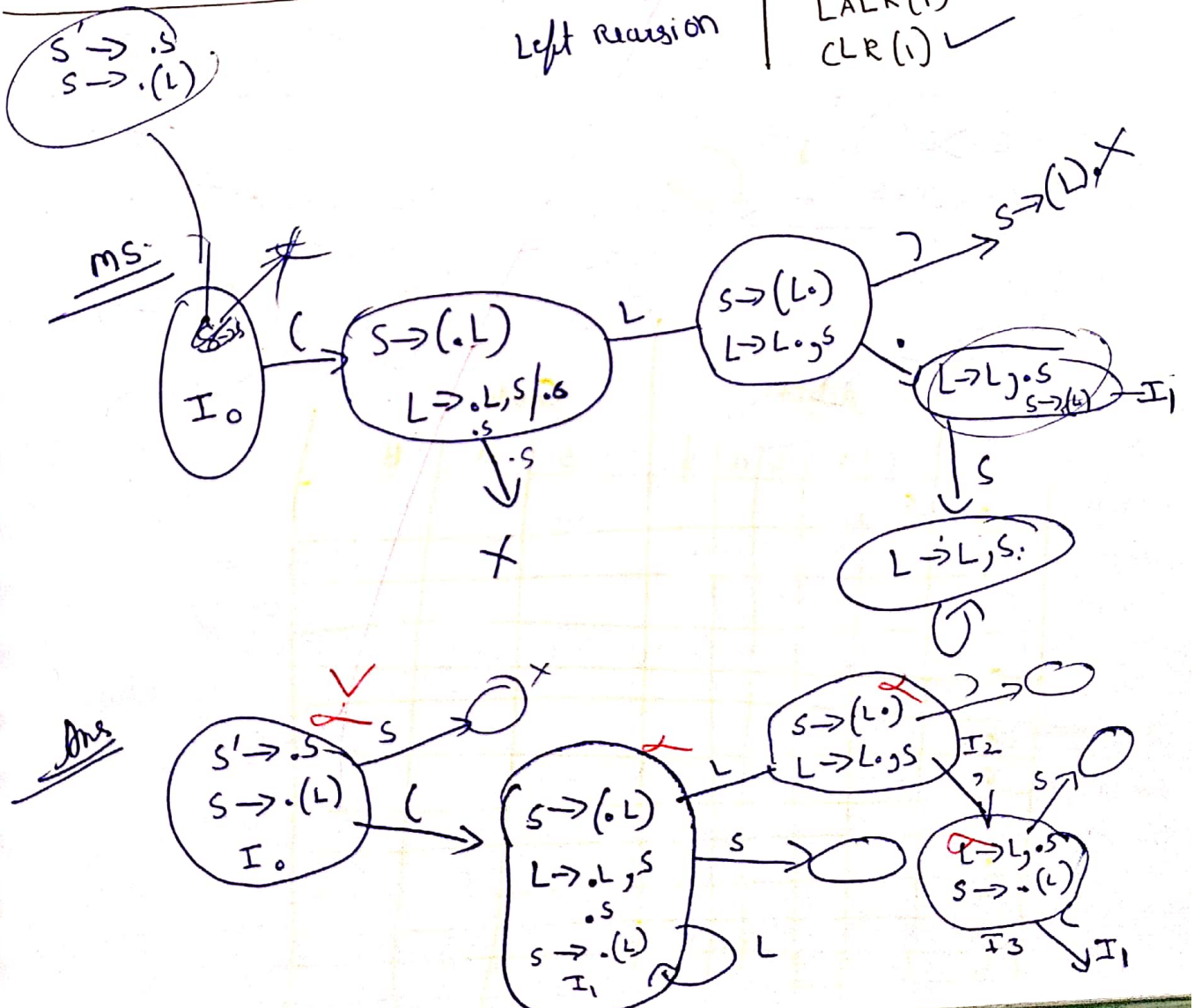
min 2 production
1 - reduce

⇒ check if SLR(1) or not ?

$S \to (L)$

$L \to L,S / S$

Top down	Bottom up (more powerful)
LL(1) ✗	LR(0) ✓
Left Recursion	SLR(1) ✓
	LALR(1) ✓
	CLR(1) ✓



17/11/18

Check weather following grammes SLR (1) or not ?

Ans: $E \to T+E/T$

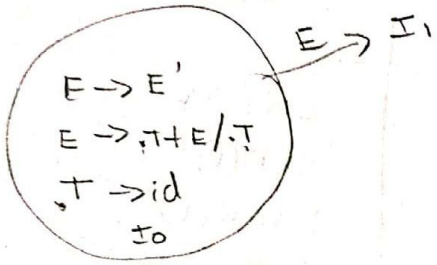
$T \to id$

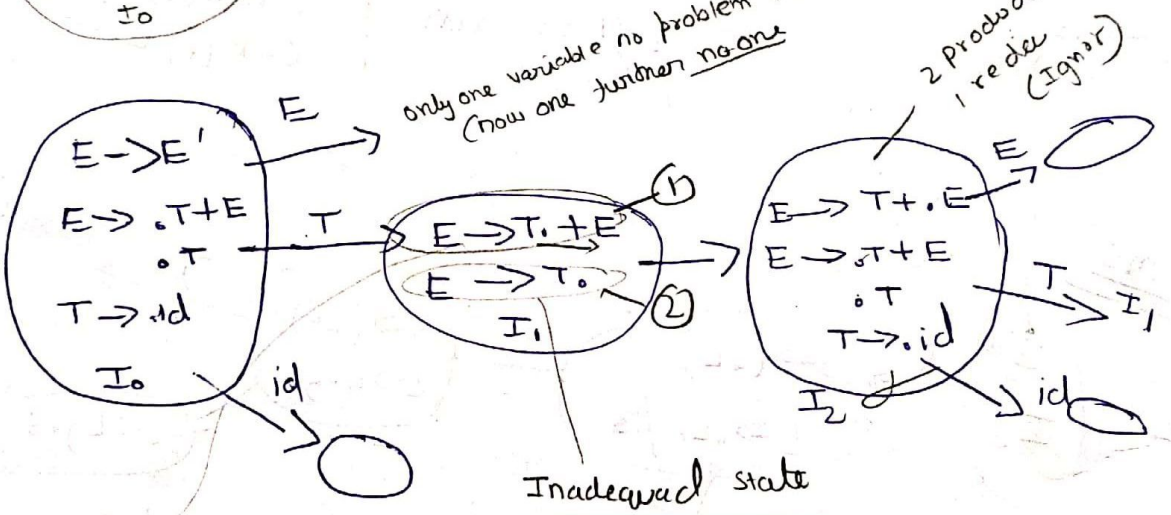
① Augmented grammar

$E \to E'$

$E \to .T+E/.T$

$T \to id$





Inadequad state

unnecessary state

		T	
1	r_2	S_2/r_2	r_2

LR(0)

If all S ppl together borrowing keep ignoring, if you find one alone like

$E \to T.$ Handle // ist studio stop check LR(0)

thanks your problem kill hum

thus at this stage ① is expanded and ② is ending

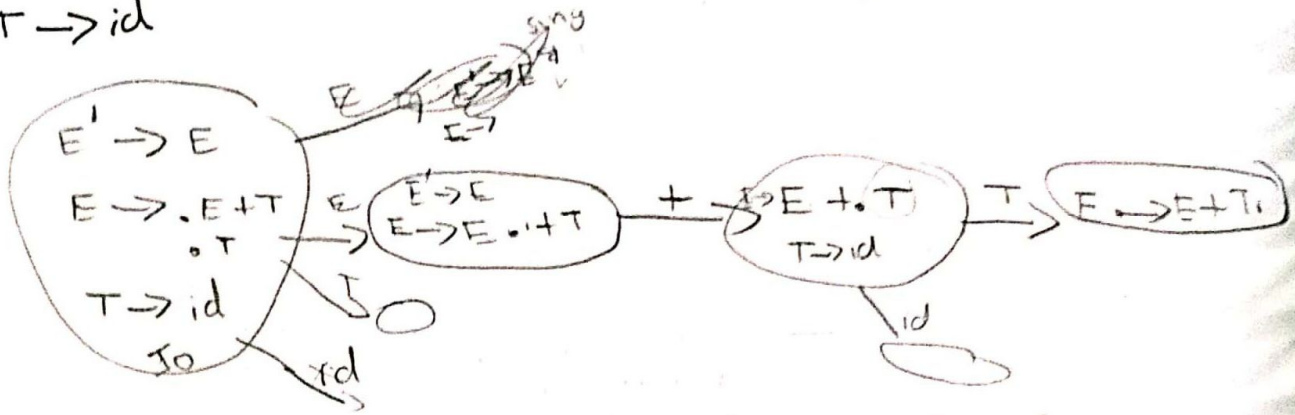
Q check the following grammer is SLR1 or not

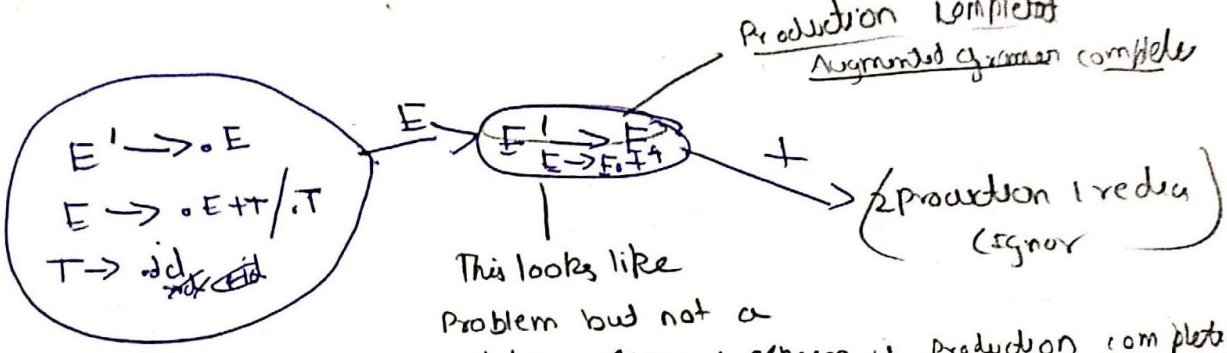
Sol

$E \rightarrow E+T \mid T$

$T \rightarrow id$

LR(0) is not SLR(1)





This looks like Problem but not a problem since 1 grammer is production completed Augment grammer.

404 theory

Q

$E \to E+T / T$

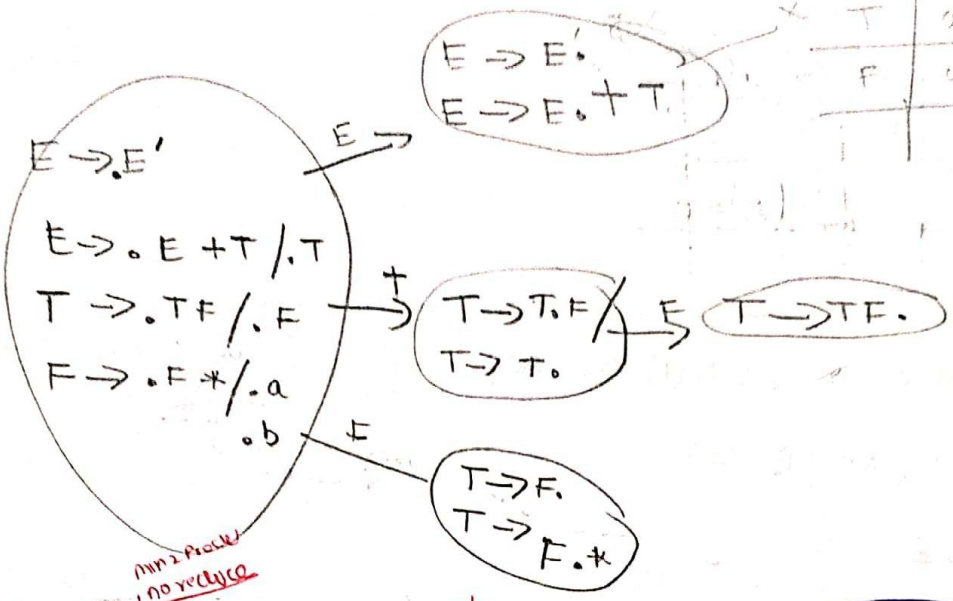
$T \to TF / F$

$F \to F* / a / b$

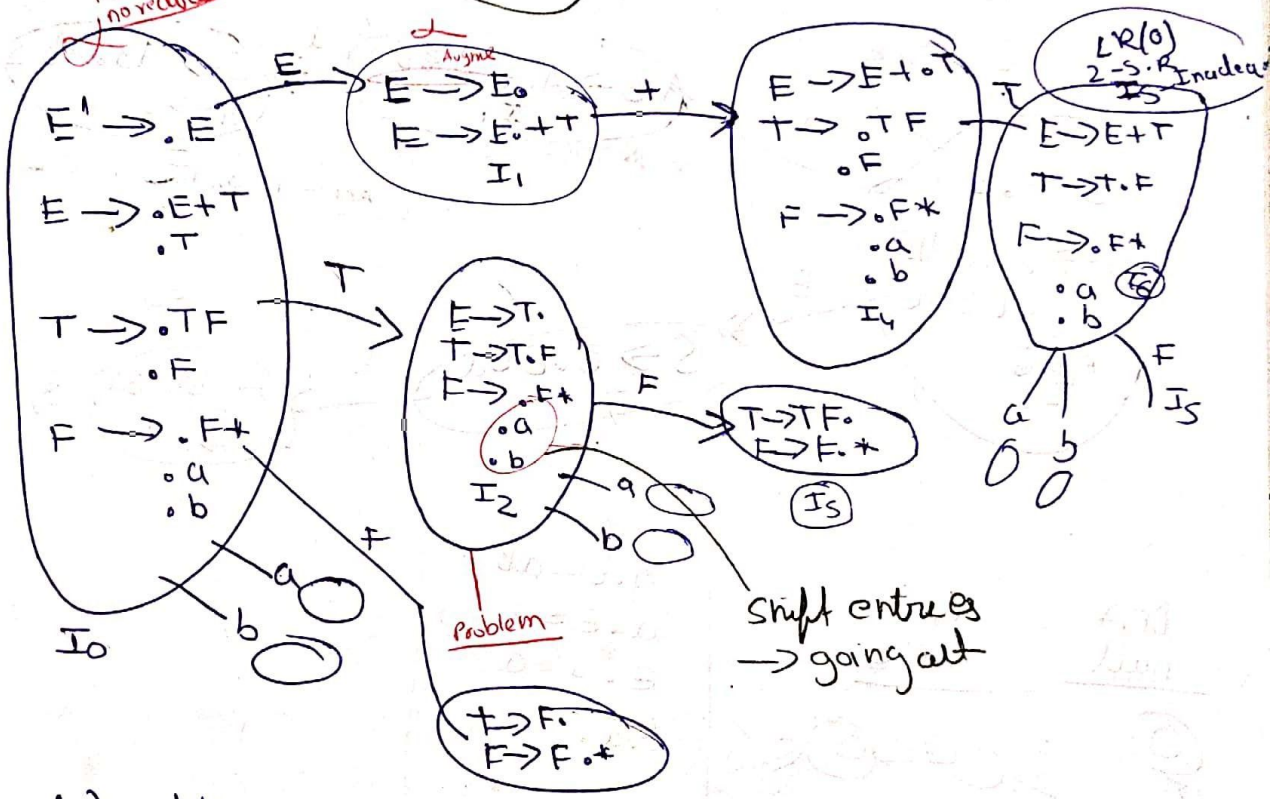
LR(0) or not ?

	a b	
E		
T	a b	not LR(0)
F	a b	

m.s



Sol



LR(0) table

	+	*	a	b	$	
2	r_2	r_2	S/r_2	S/r_2	r_2	2 SR
3	r_4	S/r_4	r_4	r_4	r_4	1 SR
5	r_3	S/r_3	r_3	r_3	r_3	1 SR
6	r_1	r_1	S/r_1	S/r_1	r_1	2 SR

4 - Inadequate System

6 - SR conflict

LR or LL problem mere in gate

SLR

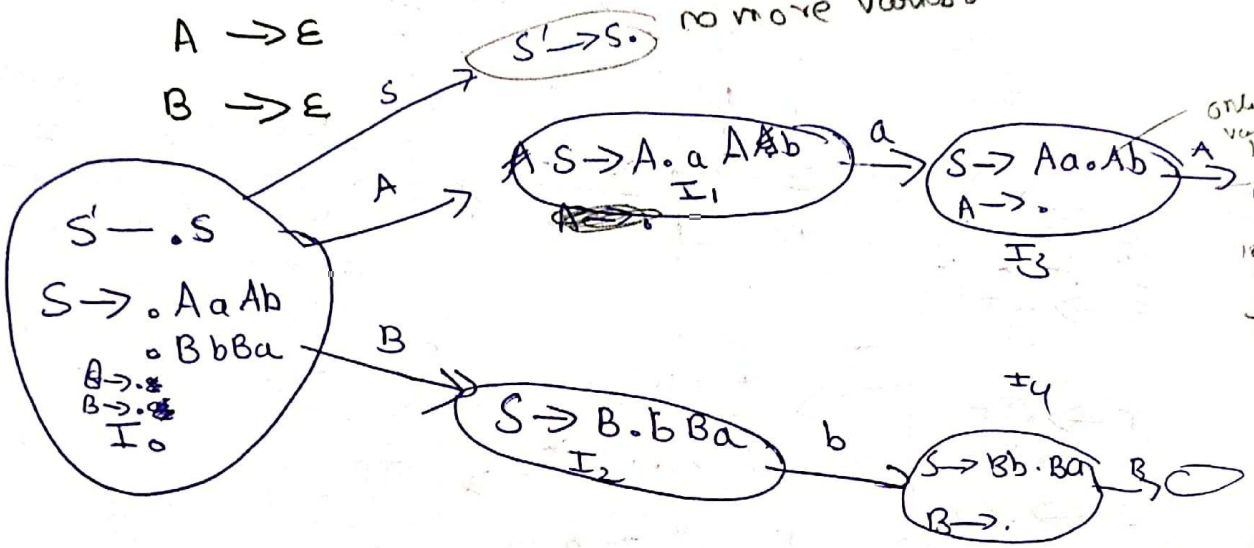
	+	+	a	b	$
2	r_2		S		r_2
3	r_4	S	r_4	r_4	r_4
5	r_3	S	r_3	r_3	r_3
6	r_1		S	S	r_1

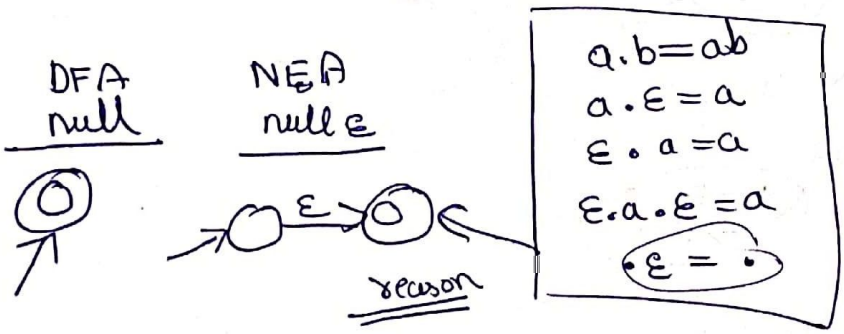
Q Check grammar is SLR(1) or not? (the ε. is catch in one question)

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$





In first state only A, and B are reducing by null production But in DFA mere (u real null ~~position~~ production

Q But mere are two reduction at I_0 only

A ε $A \to \varepsilon$ r_3

A.b $A \to \varepsilon$ r_4

] Two RR R R conflict

∴ LR(0) fails

LR(0) table

Action

a	b	$
r3/r4	r3/r4	r3/r4

SRR conflict

1 - Inadequate state

LR(0) table

SLR(1) table

Fo(A) = a, b
Fo(B) = a, b

2RR conflict

SLR(1) fail

Here

LL(1) ✓

LR(0) ✗
SLR(1) ✗
LALR(1)? — check this
LLR(1) ✓

Top Down | Bottom up

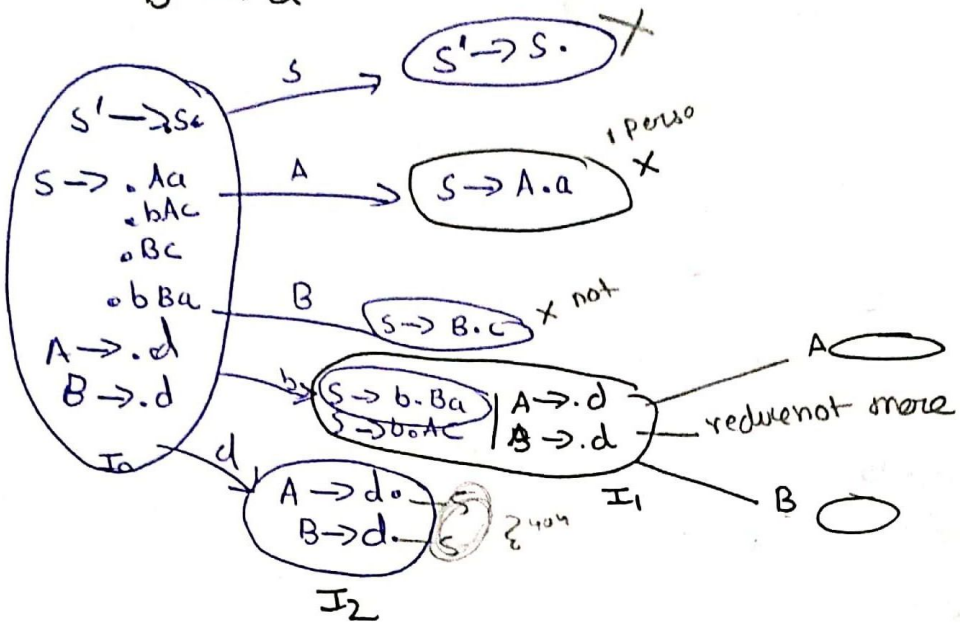
→ If LL(1) then definitely SLR(1)

(

Q $S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$



LR(0) fails

S-RR

	a	b	c	d	$
2	rs/r6	rs/r6	rs/r6	rs/r6	rs/r6

SLR(1) fails

2RR

	a	b	c	d	$
2	rs/r6		rs/r6		

$F_2(A) = a, c$

$F_6(B) = a, c$

2RR

LL(1) | LR(0) — fail S-RR

SLR(1) — fail 2-RR

CRR(1)

LALR(1)

G: $S \rightarrow AA$

$A \rightarrow aA/b$

G' = $S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA/b$

$S \rightarrow AA$ production

$S \rightarrow .AA$ LR(0) item

$S \rightarrow .AA$, c/d

LR(0) item — Look ahead symbol

LR(0) item

LR(1) item

CLR(1) LALR(1)

Defn closure (I)

⇓

① Add — I

② If $A \rightarrow B.CD$, a/b ∈ I and $C \rightarrow EF$ ∈ in G then add

add $C \rightarrow .EF$, First (D, a/b)

③ Repeat 2nd step for every newly added LR(1) item

LR(0) - Item

LR(0) SLR(1)

closure ($S \rightarrow .AA$, c/d)

⇓

① $S \rightarrow .AA$, c/d

② $A \rightarrow .aA$, a/b

$.b$, a/b

closure ($A \rightarrow .aA$, c/d)

⇓

① $A \rightarrow .aA$, c/d

closure ($S' \rightarrow .S$, a/b)

⇓

① $S' \rightarrow .S$, (a/b)

② $S \rightarrow .AA$, a/b

③ $A \rightarrow .aA$, a/b

$.b$, a/b

goto()

Ex Goto(S → .AA/, cld, A)

① S → A.A, $/c/d

② A → .aA, c/d
.b, c/d

Goto(I, x) ⇒

① add - I by moving dot after x

② Find closure (1st step)

Q Construct CRL(1) Parsing table for the following grammar

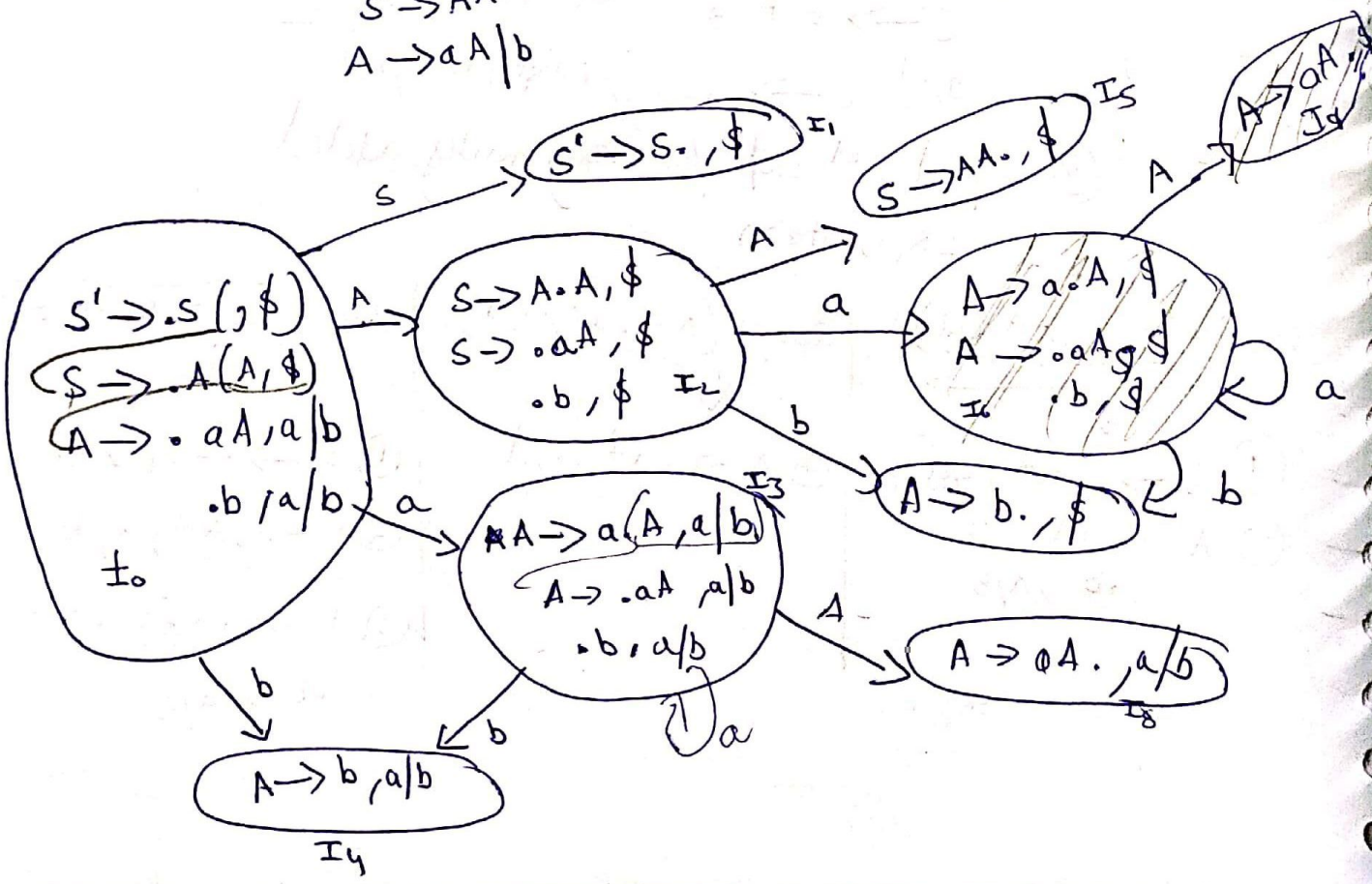
G: S → AA
A → aA/b

G': S' → S
S → AA
A → aA/b

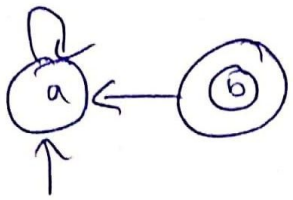
LR(0) = n1
SLR(1) = n2
CLR(1) = n3

Goto

n1 = n2 < n3



Note in minimization of



this will not reduce to 1 state as final and initial state cannot be merged.

// CLR 1 Parsing table

	Action				goto
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

CLR(1) – PT

It is CLR(1) because no conflict

-> Disadvantage with CLR(1) parser is, it will accept more grammers.
-> Drawback with CLR(1) parser is more cost as more states
-> to decrease the cost of CLR(1) we apply minimization Algorithm
-> If two states differ by lookahead make it a single state
-> Minimization of CLR(1) is LALR(1)

LALR(1) (gd by minimization CLLR CLR(1)(b)

	a	b	$	S	goto A
0	S_{36}	S_{47}		1	2
1					
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
89	r_2	r_2	r_2		
5			r_1		

LALR(1) – Parsing table

only shift combine

3,6 ⇒ 3,6
4,7 ⇒ 4,7
8,9 ⇒ 8,9

combined

⇓

LALR(1)

$S' \rightarrow .S, \$$

$S \rightarrow .A(A,\$)$

reduce the one having same Production but different Look ahead symbol

reduce combine

reduce digit cannot be combined as r_{36}

will point to 36 reduction thus do not combine. them

LR(0) – n_1

SLR(1) – n_2

CLR(1) – n_3

LALR(1) – n_4

$n_1 = n_2 = = n_4 < n_3$

CLR taking care every time

Power of LR(1) > LALR(1)

Q check the following grammer is LALR(1) or not ?

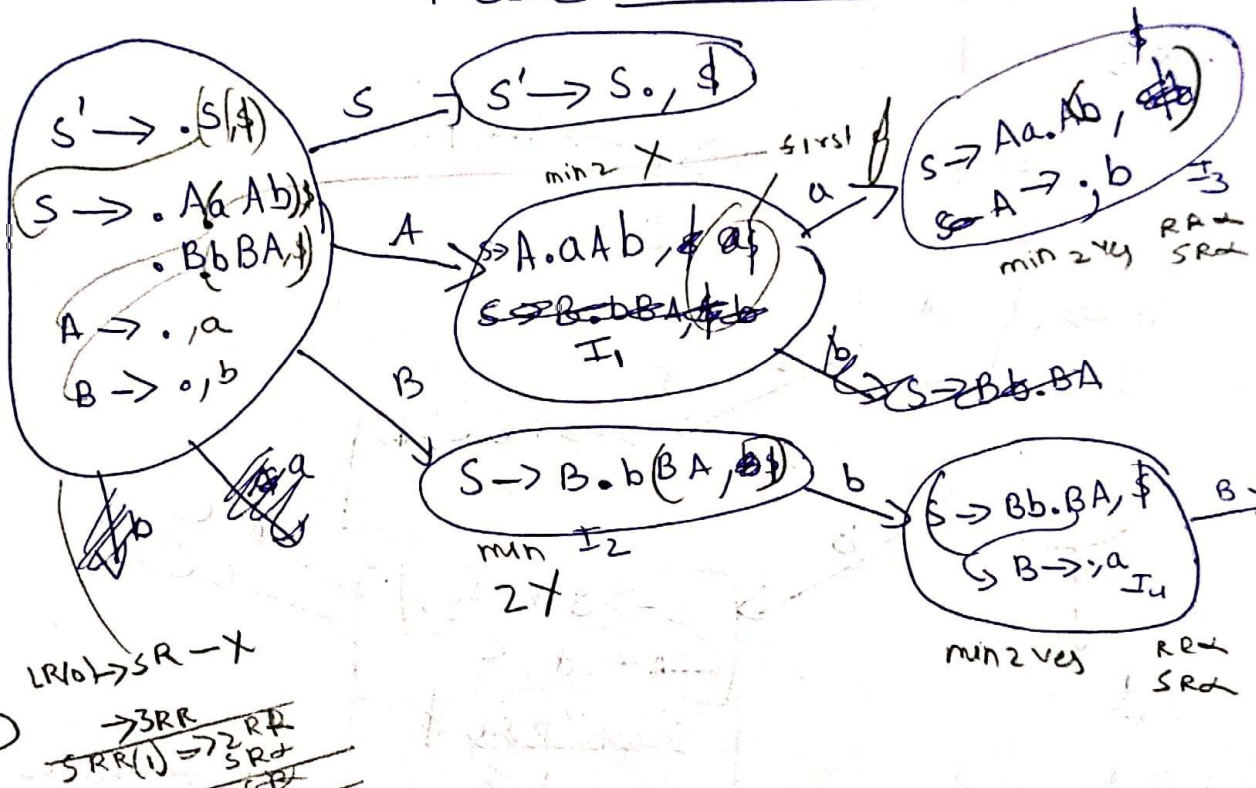
Ans $S \rightarrow Aa\,Ab \mid Bb\,BA$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

LALR(1)

LL(1) | LR(0) X
SLR(1) X
CLR(1) check first



If given all option check for CLR(1) and while checking compare

→ In the above CLR(1) no two states differ by look ahead therefore they are LALR(1) also.

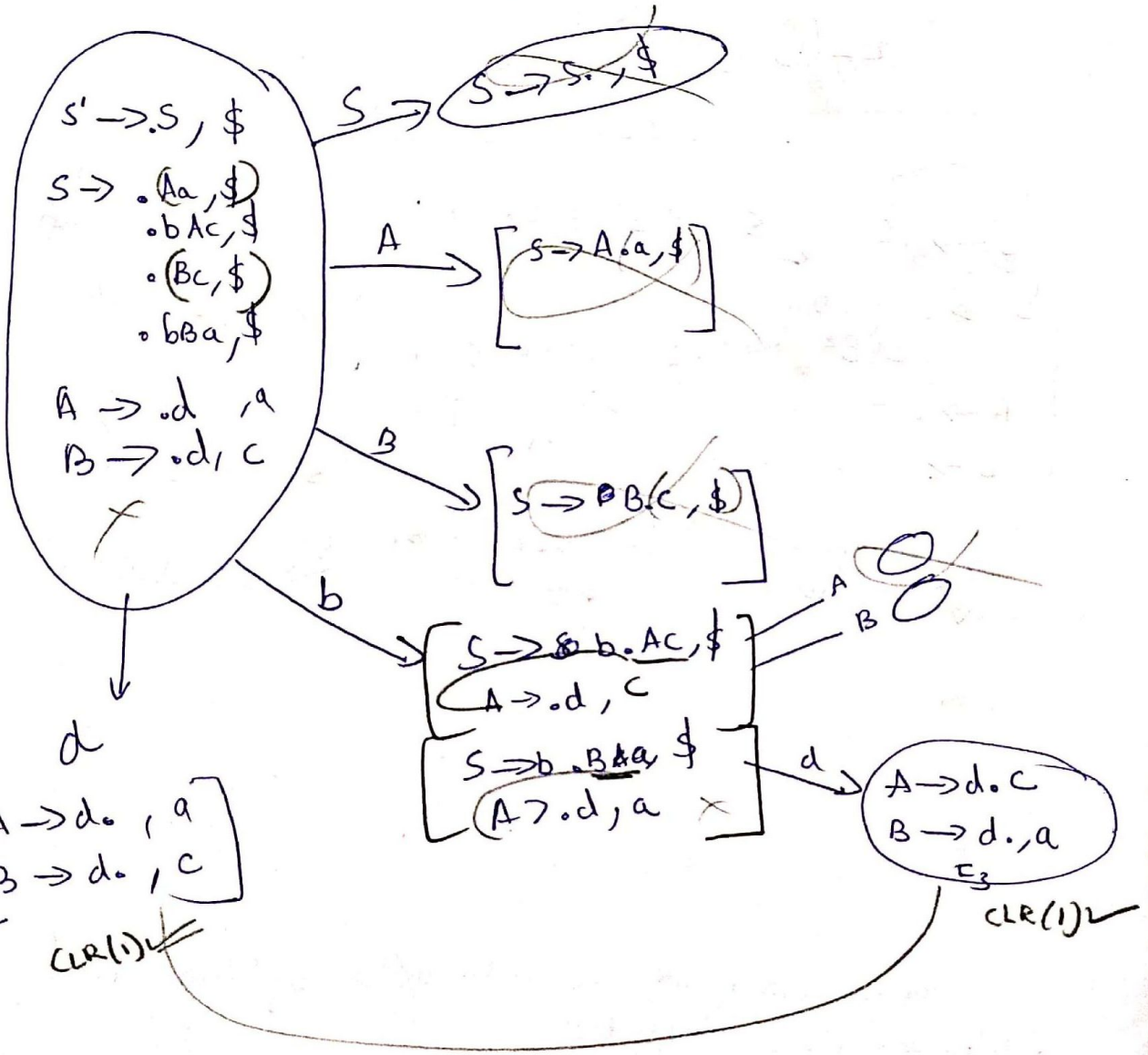
Q check me following gramer.

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

LL(1)	LR(0) × SLR(1) × CLR(1) LALR(1) } check



different by look ahead symbol

Same in SLR(1), LR(0)

n/got CLR(1)

CLR(1)

	a	b	c	d	$
2	r_5		r_6		
3	r_6		r_5		

LALR(1)

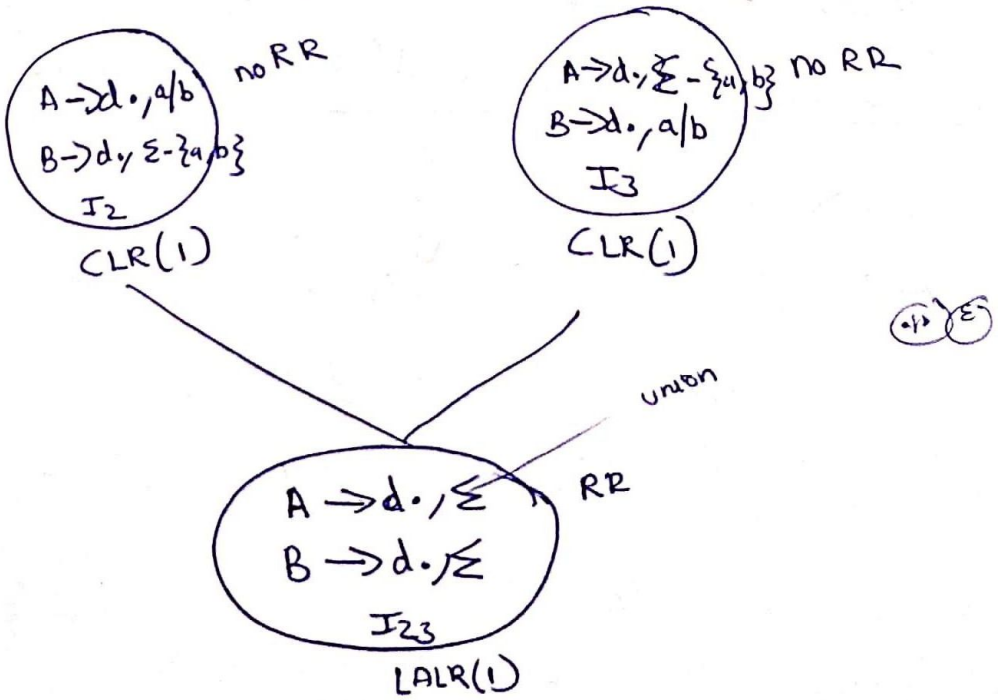
	a	b	c	d	$
23	r_5/r_6		r_6/r_5		
3					

2-RR conflict

Note *

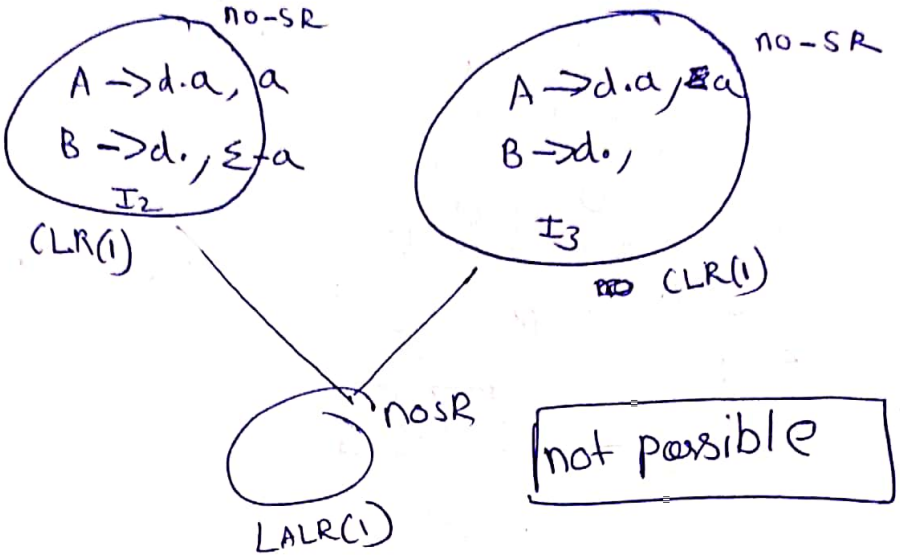
1) If CLR(1) don't have RR conflict then LALR(1) parser may contain RR conflict

Counter example



2) Note²

If CLR(1) dont have (SR) then LALR may contain SR



⇒ "If CLR(1) dont have (SR) then LALR dont contain SR"

⇒ mcq question from these above 2 theory notes.

G ⇒ LALR(1)
SR ↓
CLR
SR

G
↓
CCR(1) ✓
LALR(1) ✗

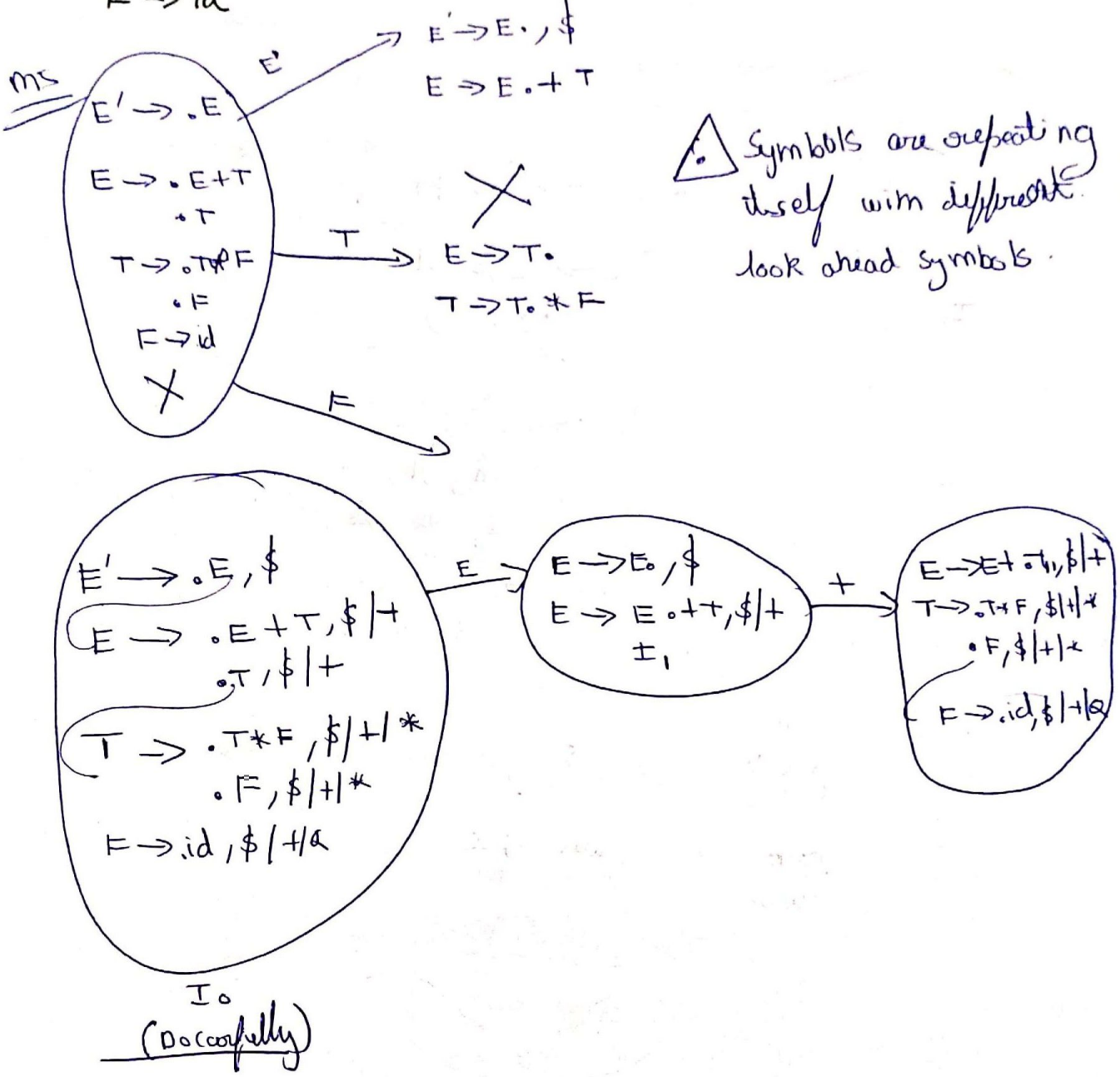
LALR(1) or not [Tricky]

Q Grate

$E \to E+T/T$

$T \to T*F/F$

$F \to id$



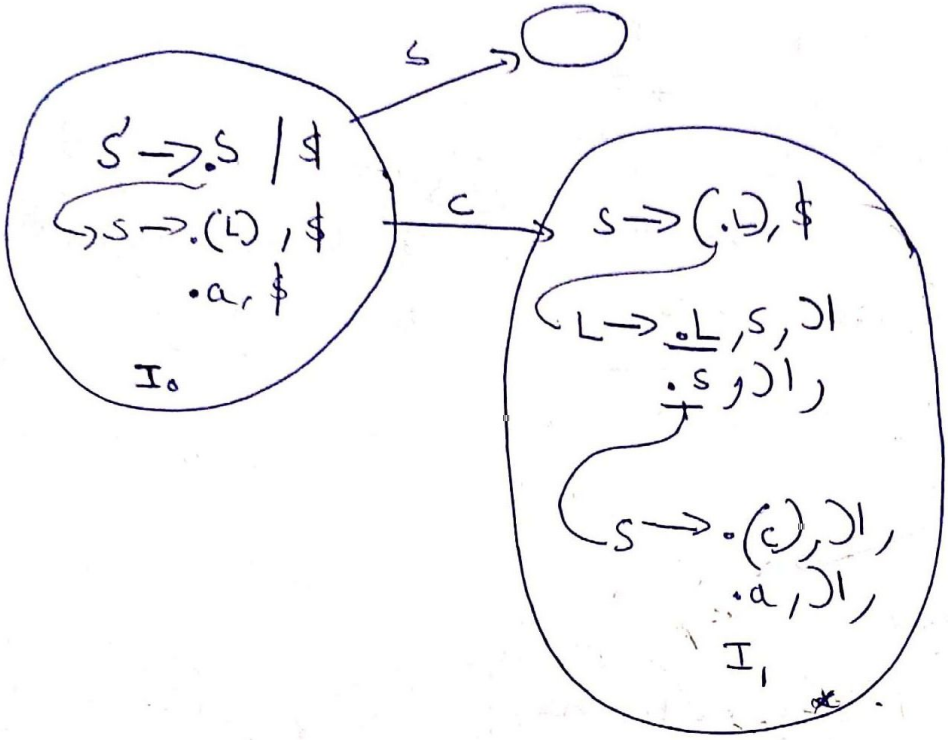
⚠ Symbols are repeating itself with different look ahead symbols.

Q. LALR(1) or not?

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$

[check till stage 4]



Gate $S \rightarrow (S) | a$

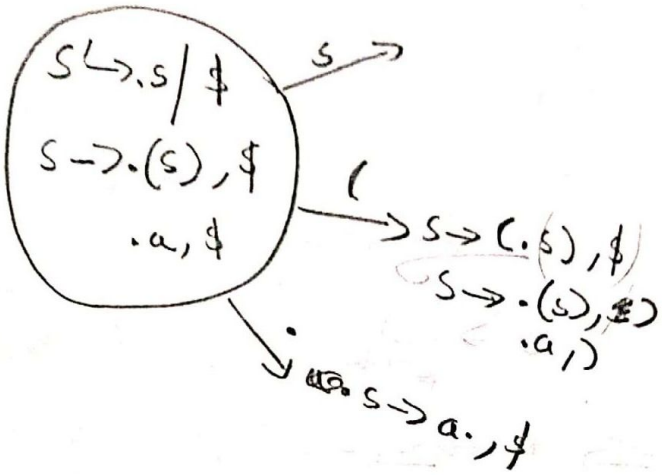
LR(0) — n_1

SLR(1) — n_2

LALR(1) — n_3

CLR(1) — n_4

$n = n_2 = n_3 \leq n_4$ ← general answer



//when counting states check all states for different look ahead symbol as may count different

8

YACC tool (Yet another compiler compiler)

- Creat LALR parsing tool
- It will resolve conflict by S/R | R/R removin SR conflict

S/R removing R

Microsoft interview

SR conflict

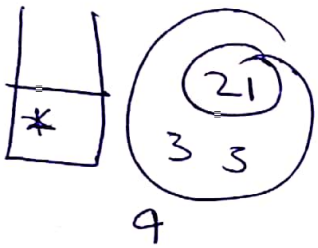
by giving favor to shift over reduce ie push operation means Right → Left ~~to~~ Associativity thus last one will be done first.

Given grammer

E → E + E | E * E | id

3 → ⌴ stack

3 * 2 + 1



→ * → |*| 3

→ 2 → |*| 3,2

→ + |*| 3,2

→ YACC tool will not consider priority it will always push

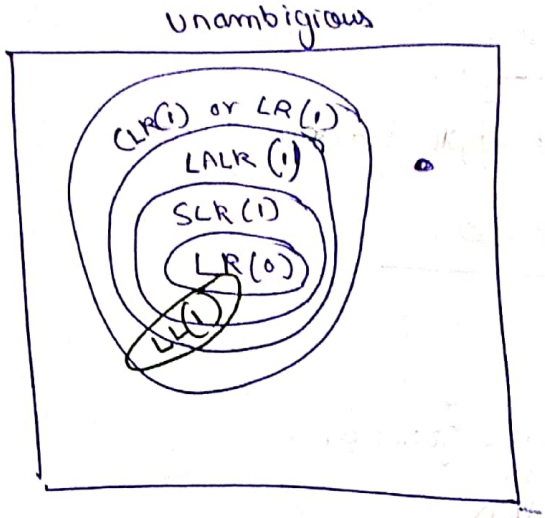
→ |+ / *| 3 21

Pop 3 * (2 + 1)

right

left

Ambiguity



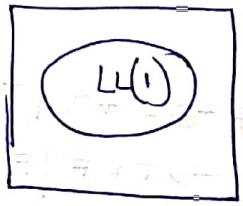
ambigious

$LL(1) \subseteq LR(1)$

$LL(2) \subseteq LR(2)$

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

no-LR
no-LF no AG



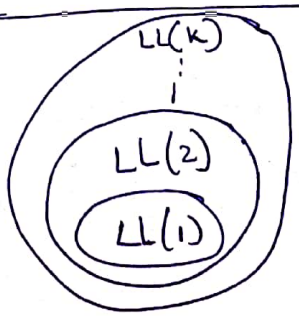
Θ $S \rightarrow a,b \mid a,c \mid ad$

LL(1) ✗
LL(2) ✗

$S \rightarrow abc \mid abd \mid abe \mid$

LL(2) X
LL(3) ✓

$\Rightarrow$



LR(K) does not mean how many symbols we reading, it means by seeing which K symbols are doing reduction.

Operator Precedence Parser (no question past 15 years)

⇓ will be applicable only for

operator grammer

A grammer G is said to operator grammer

1) G does not have null production
2) G donot have 2 adjacent variable and right ~~asst~~ hand side of the production.

ex

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

operator grammer ✓

Ambigious grammer ✓

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

OG ✓

$E \to E+E \mid E+E \mid \in$

Amb ✓

Operat X

null production

$S \to AB$

$A \to a$

$B \to b$

unary ✓

operator X

Two variables side by side.

18/11/18

Syntax Directed Translation (SDT)

combination of

① SDT = Sg Grammer + Semantic Actions

Ex $E \to E_1 + E_2$ } if (E_1.type ≠ E_2.type) Pf (type mismatch) or E.type = E.type

---

Ex2 $S \to abc$ } Pf ((hi)} } there is no attributes

Types of Attributes

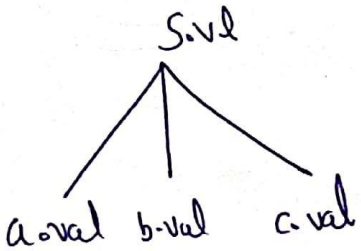
Synthesized Attributes

fn of children

$S \to ABC$ { S.val = f(A.val, B.val, C.val) }

S.val = sval = A.val + b.val * -c.val

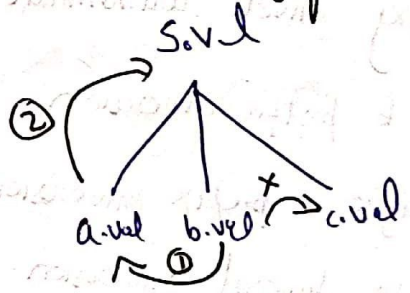
(* may not be included in challenge)



Inherited Attributes

$S \to ABC$ { B.val = f(S.val, A.val, b.val) }

fn of sibling parent



B.val = A.val + c.val - S.val

// inherit from senior brother to chon parent

Types of SDT

S-attributed Defination (single)

1) It uses Synthesised Attributes
2) Bottom up evaluation.
3) Semantic action should be in RHS of production.

L-Attributed Defination (combined)

L - because inputting from left first

1) It uses both inherited & Synthesised
2) Depth first order & Left to Right evaluation.
3) Semantic actions place anywhere on RHS of the production.

Both (parent + Sibling) left first

Grate

Applications of SDT

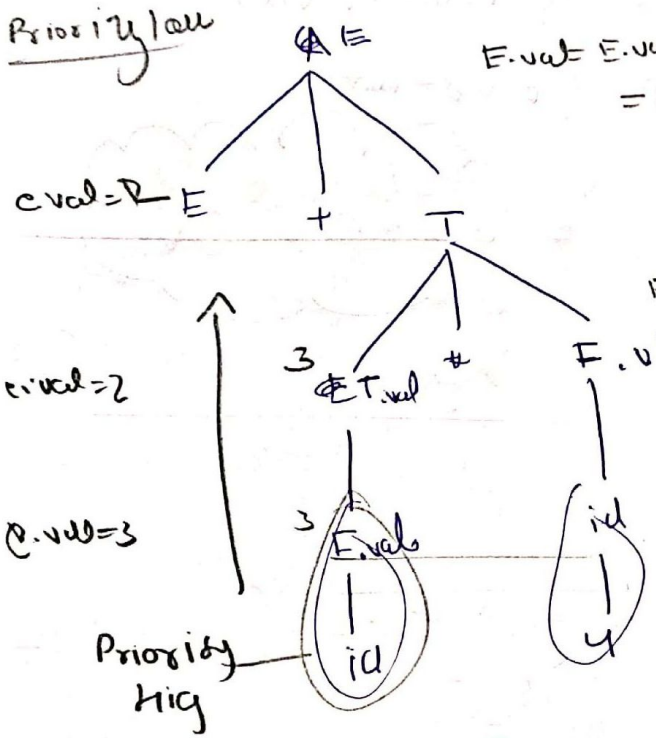
1) Executing Given arithmatic expressions.
2) Infix to postfix conversion
3) ~~Prefi~~ Infix to prefix conversion
4) Binary to decimal conversion.
5) Creating Syntax tree (it helps it)
6) Generating intermediate code.
7) Storing type information to Symbol table
8) Type checking.

9) finding number of reductions
and many more ----

Construct SDT to execute the given arithmetic expression

input : 2+3*4
output : 14

val synthesised attribute
↓
Synthesised Attribute
↓
~~Inherited~~ L-attributed Def



$E \rightarrow E_1 + T$ { E.val = E.val + T.val
pf (c.val)
T.val = F.val)

$T \rightarrow T_1 * F$ { T.val = T1.val * F.val)
(T_1.val = F.val)

To differentiate from T

$F \rightarrow (E) \mid id$ { F.val = id }

In exam more symbol would be changed

Q Consider the following SDT

$E_1 \rightarrow E \# T$ { E.val = E_1.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 }

Right

ip = 2 # 3 * 5 # 6 * 4

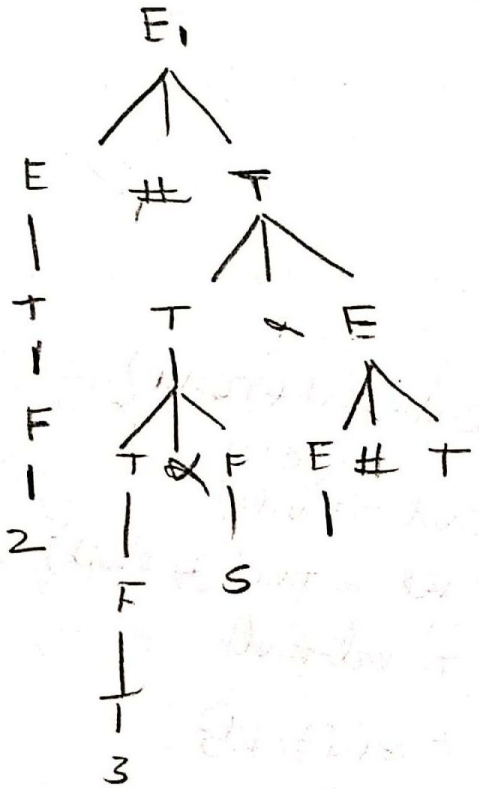
8 10

80

160

(Left → Right Associative)

Redraw tree



Q Consider following SDT

$E \rightarrow E_1 \# T$ { E.val = E1.val * T.val }

| T { E.vl = T.vl }

$T \rightarrow T_1 \& F$ { Tval = T.val - E.val }

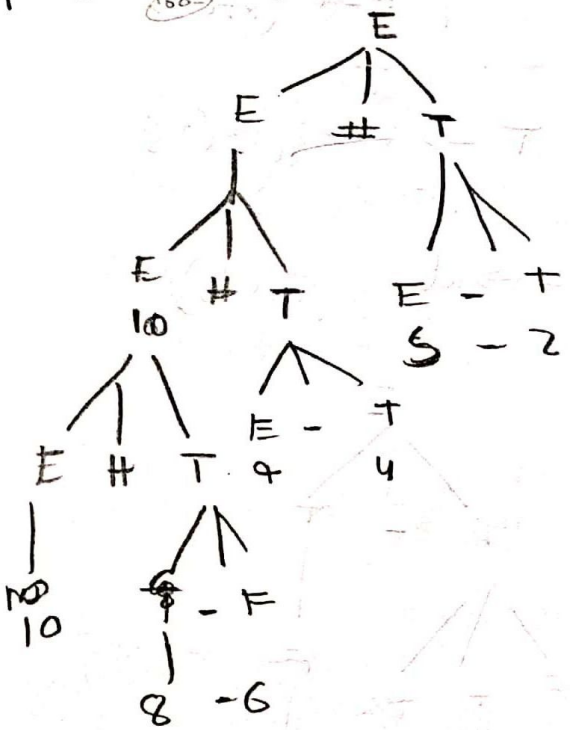
| F { T.val = F.val }

F = num

ip = 10 # 8 & 6 # 9 & 4 # 5 & 3

op = 300 (100) 2 30 5 15 3

Solve by substituting all symbols



⚠ Grammar is giving both priority and Associativity not the symbol Priority

Q. construct SDT to given infix expression to postfix

i/p : Infix = a + b * c − d

op : Postfix a b c * + d −

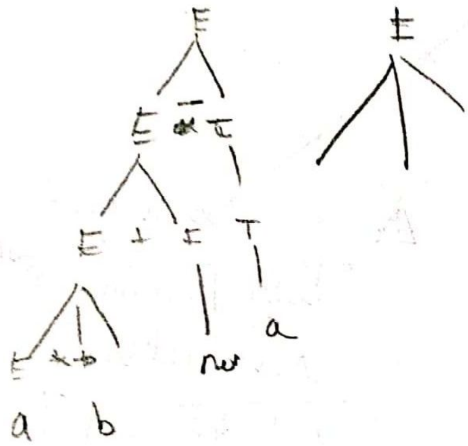
m.s

E = E₁ − T) E.val = E.val − T.val)

E = E₁ − T / E₁ + T

E₁ = E * T

F = id



Sol

E → E + T (pf +) (Same Priority)

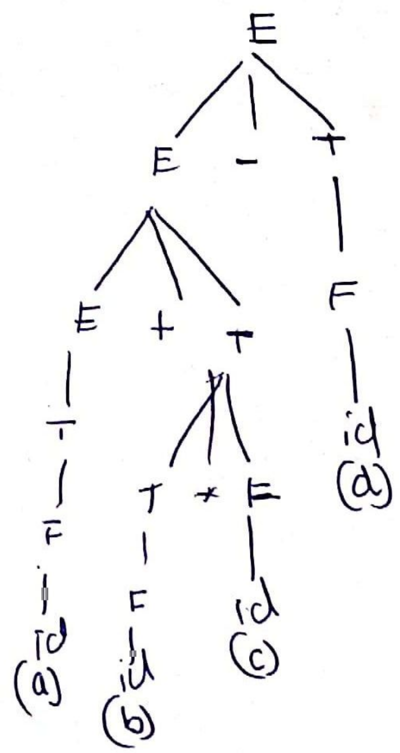
| E − T (pf (−))

| T

T → T * F {(pf (*}

| F

F → id {(pf (id))}



S attributed derivation

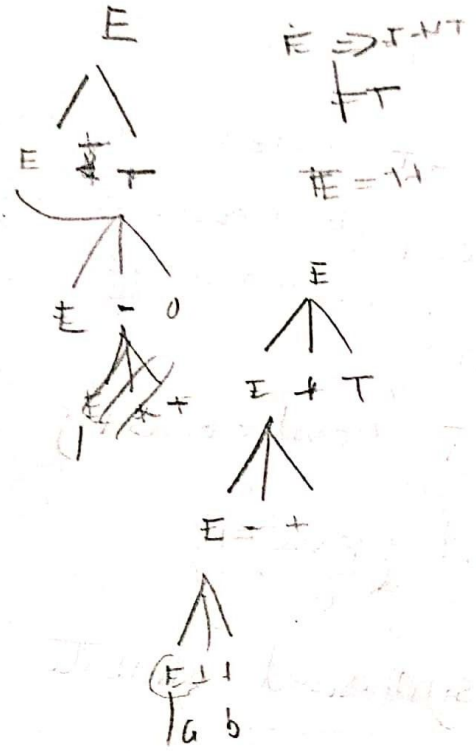
⇓

L attributed

Q construct SDT from infix to pos prefix conversion.

(infix) i/p = a + b * c - d

(prefix) op = - + a * b c d



print before operator

E -> { Pf(+) } E + T
| { Pf(-) } E - T
| T

T -> { Pf(+) } T * F
| F

F -> id { Pf(id) }

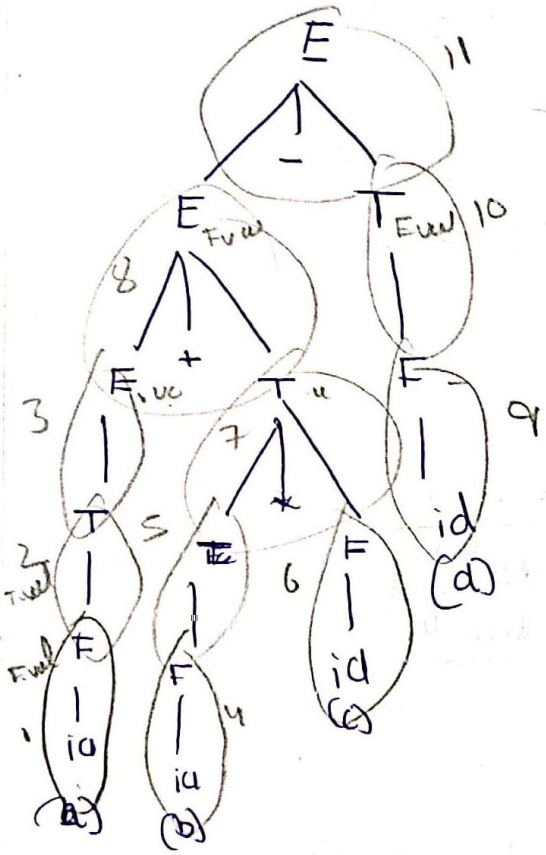
L attributed
but
not attributed

Construct SDT to count no. of reductions required to execute given Arithmetic expression.

Ans:

ip: a+b*c-d

op: 11



L chil 1 + chil 2 + 1

E → E+T {E.vl = E.vl + T.vl + 1}

→ E-T {E.vl = E.vl + T.vl + 1}

| T {E.vl = T.vl + 1}

T → T*F

| F {T.val = F.val + 1}

F → id {F.val = 1}

val is a synthesised Attribute

⇓

S attribute derivation

⇓

L Attribute derivation

Gate

Consider the following SDT

$S \rightarrow AS \{Pf(1)\}$

$S \rightarrow AB \{Pf(2)\}$

$A \rightarrow a \{Pf(3)\}$

$B \rightarrow bc \{Pf(4)\}$

$B \rightarrow dB \{Pf(5)\}$

$C \rightarrow c \{Pf(6)\}$

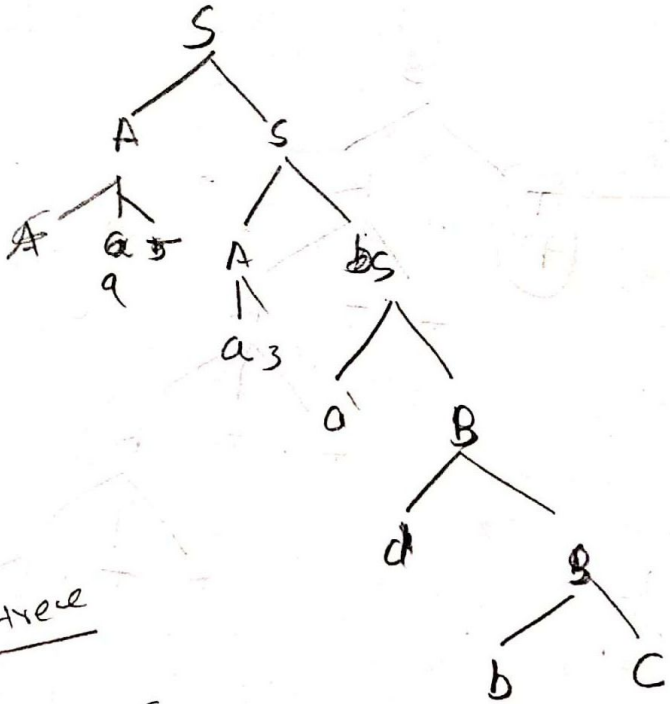
Ip- aaadbc

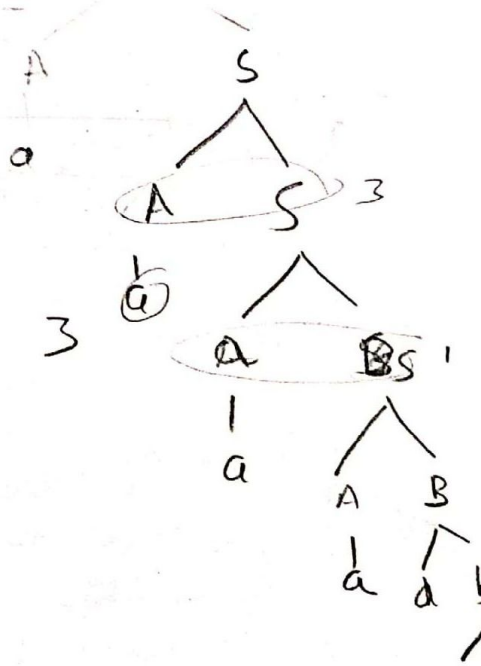
Op → 3 3 3 6 4 5 2 1 1

3 3 3 5

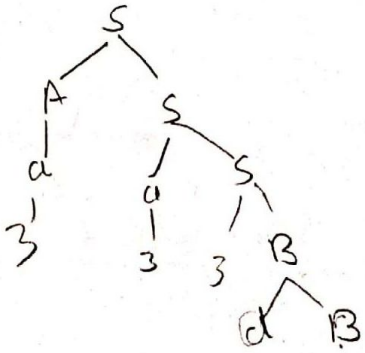
→ 33

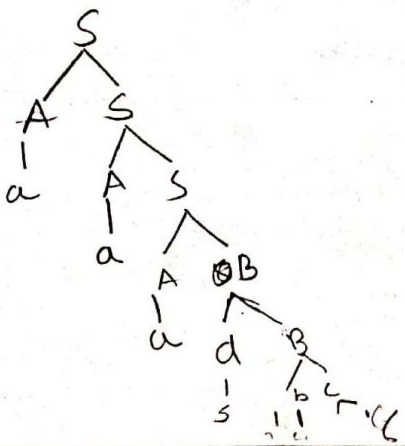
→ 33354





Draw tree





Q. Consider the following SDT

$S \rightarrow T \{ Pf(*) \} R$

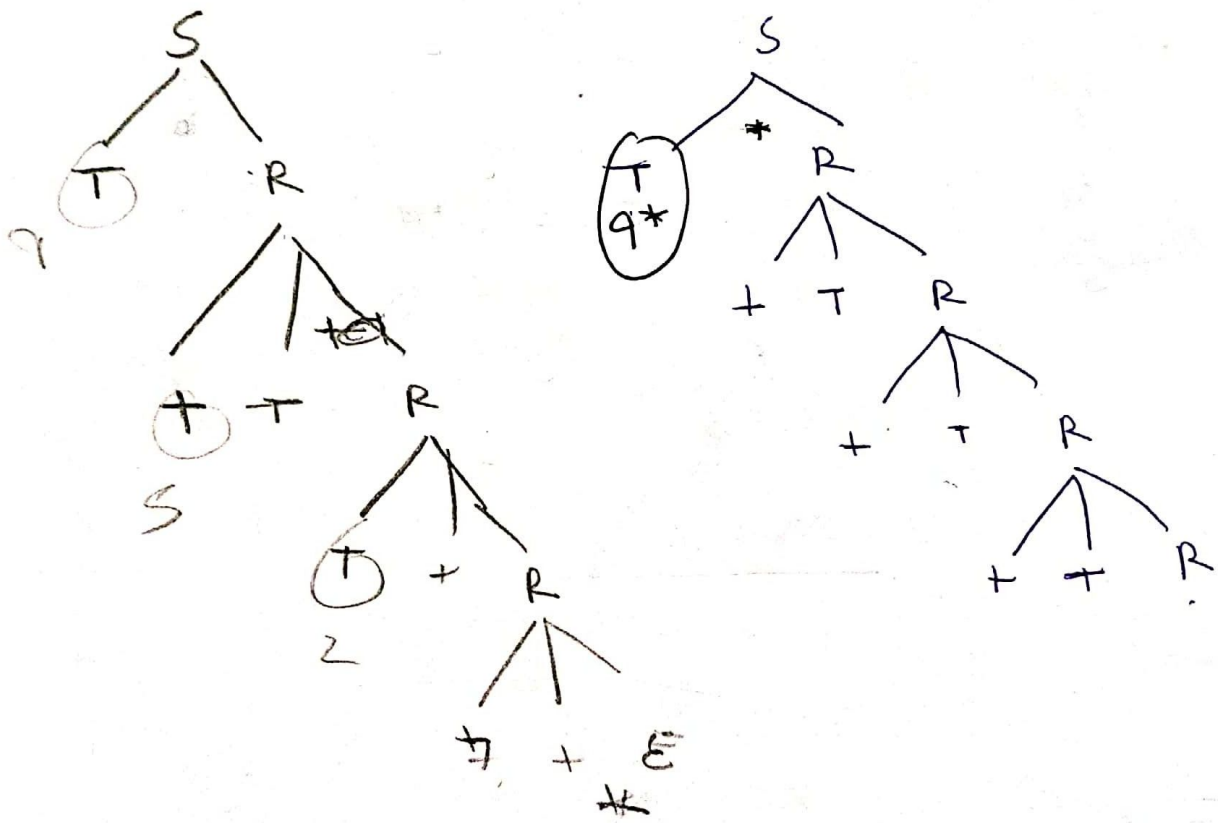
$R \rightarrow + T \{ Pf(*) \} R \{ Pf(+) \}$

$| \in \{ Pf(*) \}$

$T \rightarrow num \{ Pf(*) \}$

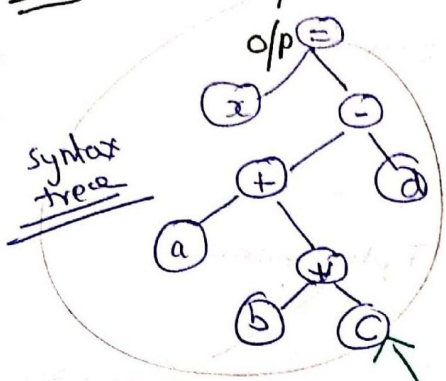
ip = 2+5+2+7

Ans O/P : + * + * + * +* *++ +



Q construct SDT to create syntax tree for given Arithmatic expression

Ans: i/p : $x = a + b * c - d$



Syntax tree

Step ①: take grammer for input

Step ②:

~~E → E + E~~ S ⇒ ~~S~~ F = E

$E \rightarrow E + T$

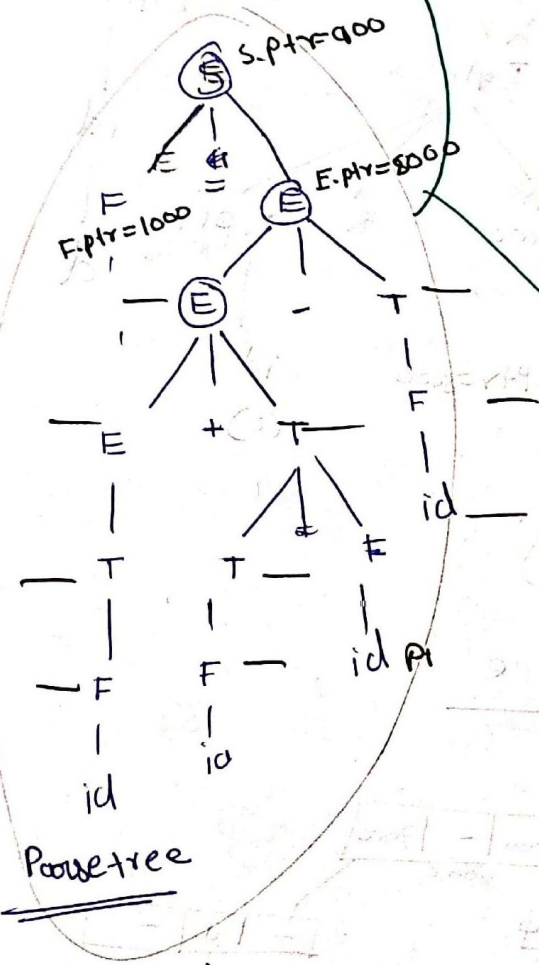
$E - T$

T

$T \rightarrow T * F$

$F \rightarrow id$ (F)

Sol

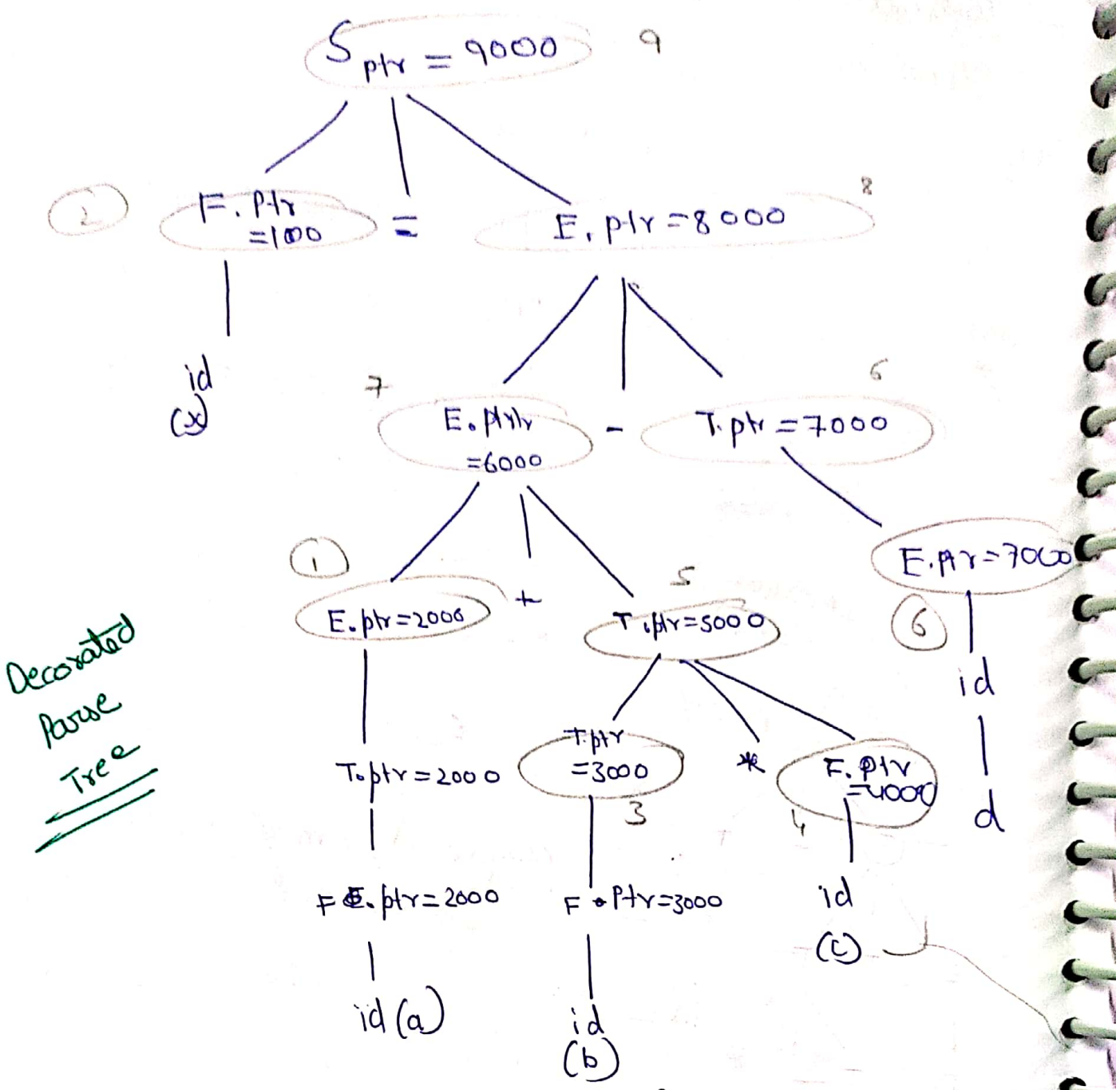


Parse tree

compressed form of parse tree is Syntax tree.

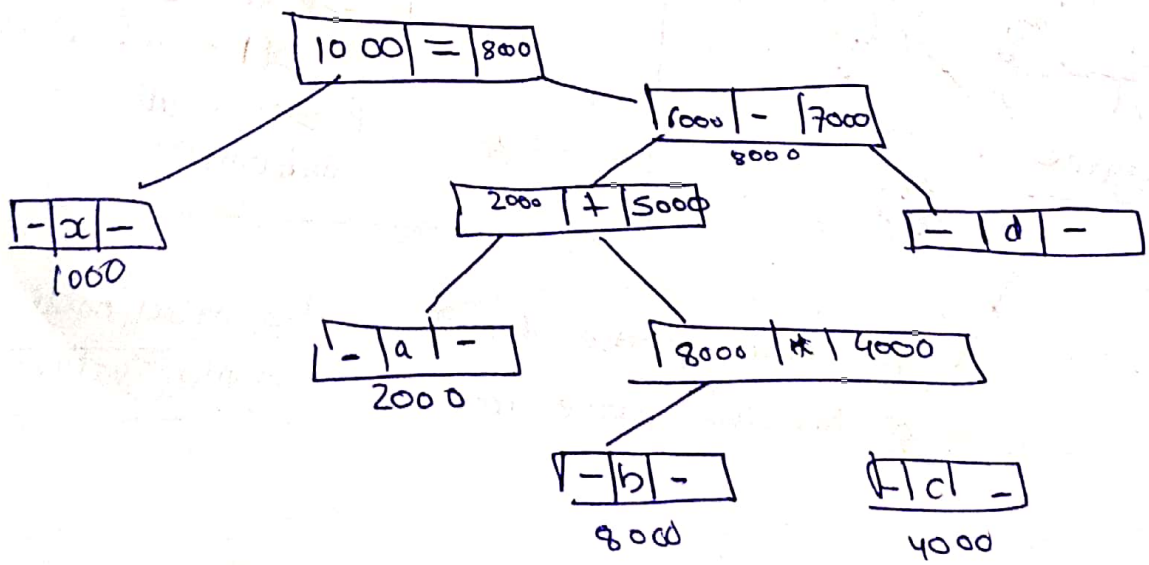
```
using malloc
p=malloc
p->data = ε
p->lc=null
p->rc = null
return(p)
```

Decorated parse tree or Annoted parse tree — has every node contains value



Decorated Parse Tree

Parse tree



Sol

$S \rightarrow F = E$ { S.ptr = mknode(F.ptr, =, E.ptr); return(S.ptr) }

$E \rightarrow E_1 + T$ { E.ptr = mknode(E_1.ptr, +, T_1.ptr) }

| $E_1 - T$ { E.ptr = mknode(E_1.ptr, −, T_1.ptr) }

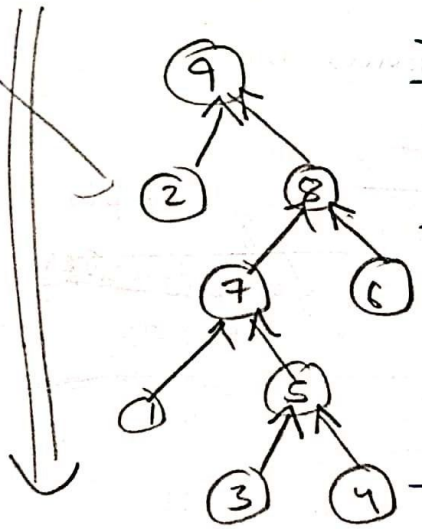
| T { E.ptr = T.ptr }

$T \rightarrow T_1 * F$ { T.ptr = mknode(T_1.ptr, *, F.ptr) }

| F { T.ptr = F.ptr }

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

→ created using 1 malloc() only



→ ~~on decorated~~
→ On topological sort of Decorated Parse tree can be used to find who will execute first.

⇒ only applicable for ~~dy~~ directed acyclic graph.

→ 3 and 4 are independent but 5 depends upon 3 and 4

both correct Provided no dependency

Dependency graph

↳ who depends upon

2 1 3 4 5 8 7 6 8 9

or

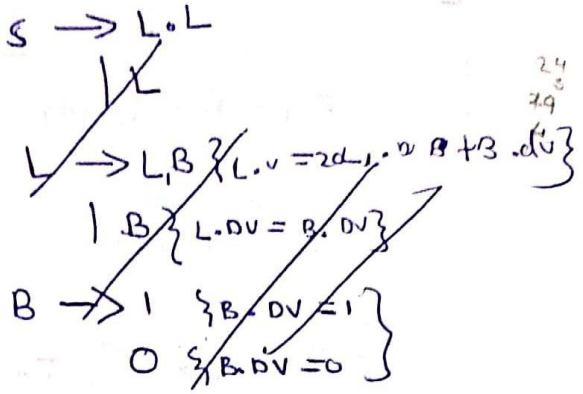
⇒ 1 2 3 4 5 6 7 8 9 (Topological sort)

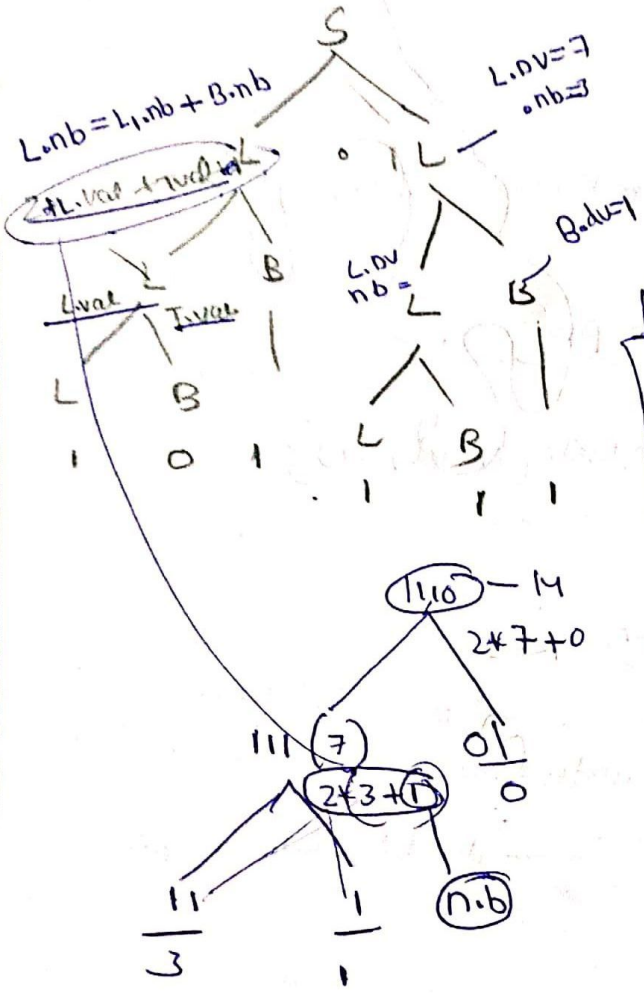
Ex: Consider SDT to convert the given Binary number into decimal

i/p: 101.111

o/p = 5.875

Grammar





To find the no. after decimal calculate the value as it as normal ie

.111 = 7

then then divide by

7

2 (7) — no of contributors for explosives

$S \rightarrow L_1 . L_2$ { $S.DV = L_1.DV + L_2.DV$; $Pf(S.DV)$ }

Decimal value

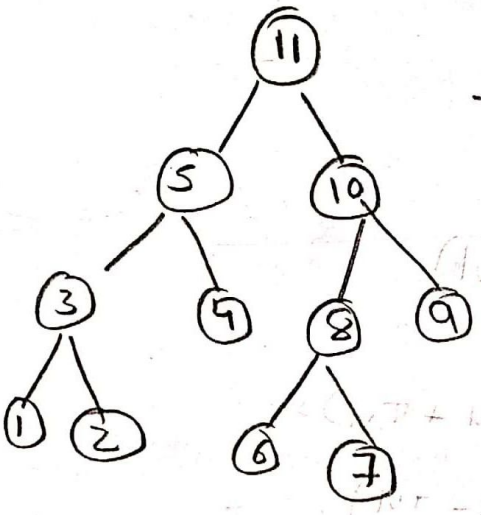
$L \rightarrow L_1 B$ { $L_1.DV = 2 * L.DV + B.V$; $L.nb = L_1.nb + B.nb$ }

no. of bits

$L \rightarrow B$ { $L.DV = B.DV$; $L.nb = B.nb$ }

$B \Rightarrow 1$ { $B.DV = 1$; $B.nb = 1$ }

0 { $B.DV = 0$; $B.nb = 1$ }



Topological sort

1 2 3 4 5 6 7 8 9 10 11

12 6 7 4 9 3 8 5 10 11

Q Construct SDT to generate intermediate code for Gate given Arithmetic expression

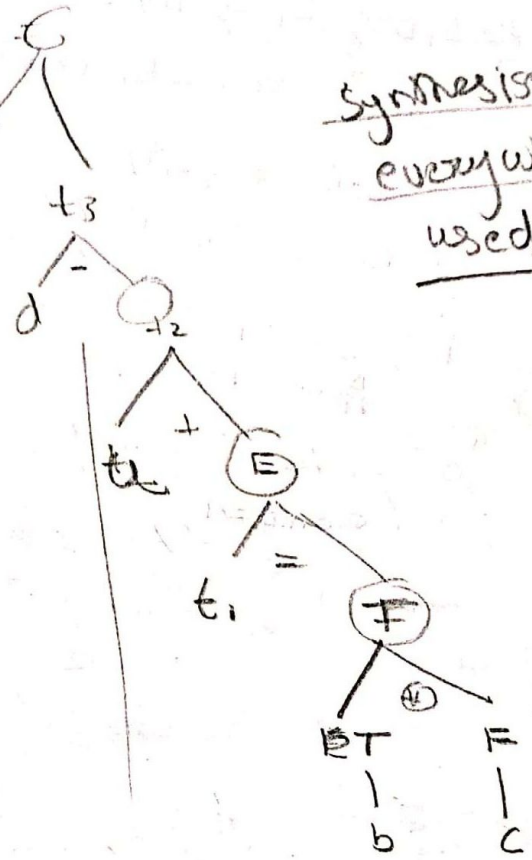
Ans i/p x = a + b * c - d

o/p = $t_1 = b * c$

$t_2 = a + t_1$

$t_3 = t_2 - d$

$x = t_3$



Synthesised grammar everywhere used

Grammar

ms

$S \rightarrow F = E$ { gen (F.vl = E.vl) }

$E \rightarrow E_1 + T$ { E.vl = newtemp() gen (E.vl = E_1.vl + T.vl) }

| E - T { E.vl = newtemp() gen (E.vl = E_1.vl − T.vl) }

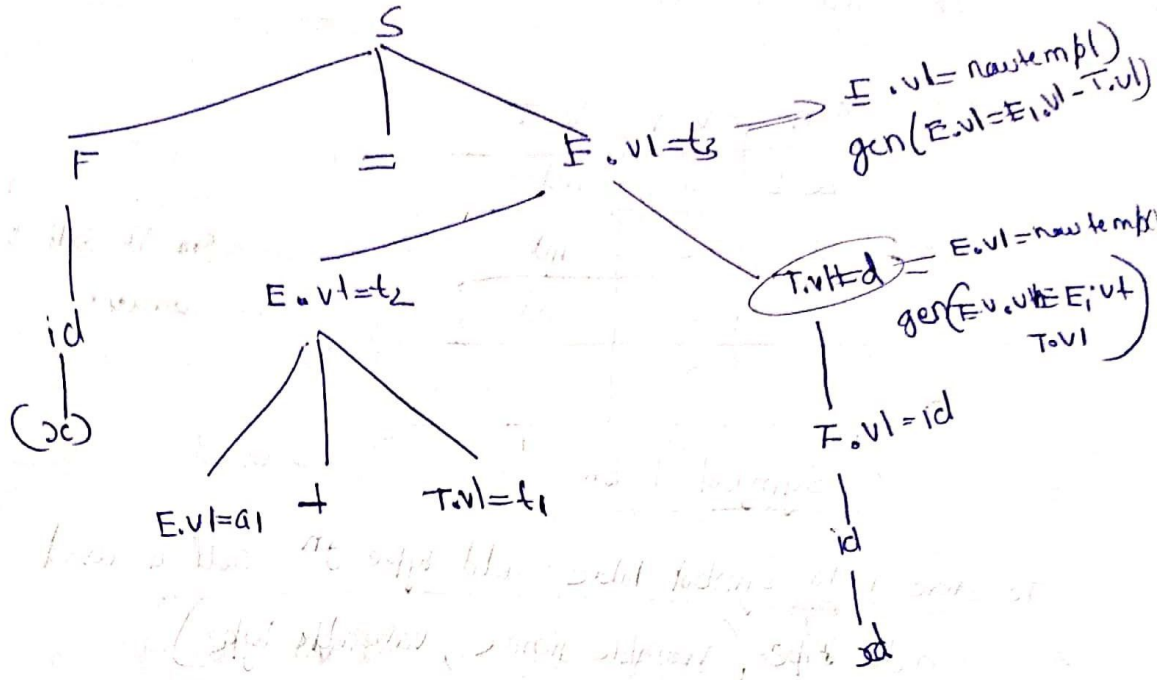
| T { E.vl = T.vl }

$T \rightarrow T * F$ { T.vl = newtemp() gen (T.vl = T_1.vl * F.vl) }

| F { T.vl = F.vl }

F — id { F.vl = id }

→ Not using printf inside generator method given to print the temporary variable.



Q consider the following SDT

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

$E \rightarrow E_1 + E_2$ { E.vl = newtemp() gen (E.vl = E_1.vl + E_2.vl) }

| id { E.vl = newtemp() gen (E.vl = id) }

Create new temporary variable on each function call in sequential order

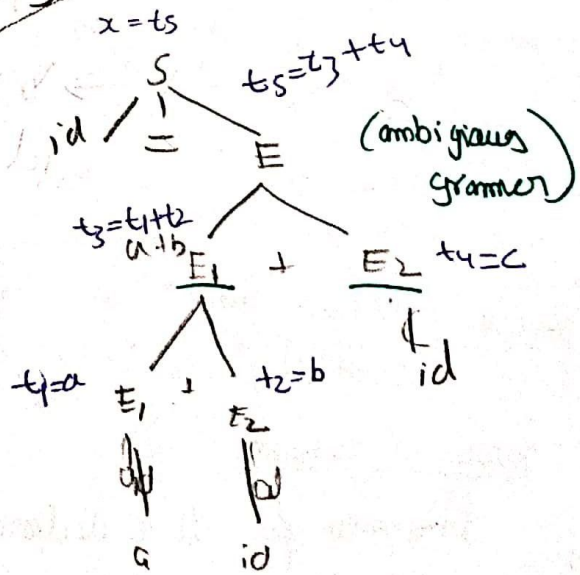
ip: x = a+b+c

op: ÷

(So assume Left → Right Associativity)

$t_1 = a$
$t_2 = b$
$t_3 = t_1 + t_2$
$t_4 = c$
$t_5 = t_3 + t_4$
$x = t_5$

$t_1 = a+b$
$t_2 = t_1 + b$
$x =$



Q. Consider SDT to type information into symbol table

Ans: [Introducing Inherited Attributes]

i/p : int a, b, c;

o/p :

S.n	V.n	V.T
1	a	int
2	b	int
3	c	int
4		

Symbol table

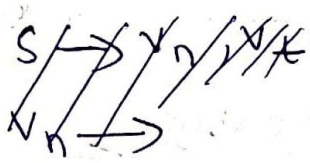
From top it will search where ever gap present it will store here the variable.

Initially empty

To store into symbol table add type fn call is used

[add type (variable name , variable type)]

Q:

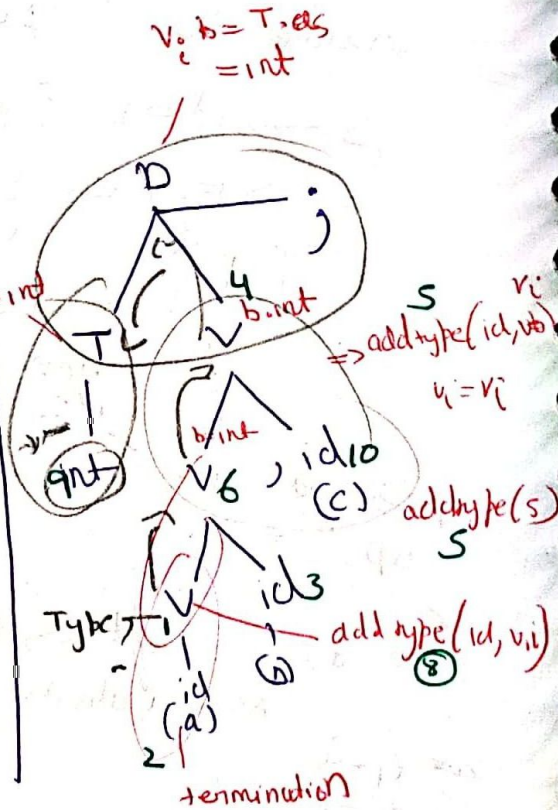


$\Rightarrow$ D → T , V ;

T → int
char
float

V → V , id
id

// Doesn't matter wherever you start it depending it will solve depending first



// Evaluation based on topological sort

Interview question

Grammar for all c declaration or gmail id

→ if asked check rules then write

Grammar

Code

$D = T_a\ v_i \{ v.i = T.s \}$ — order to check taking from Right Siblin

$T.s = v.i$ — cannot keep in between

$T \Rightarrow$ int $\{ T.s = int \}$

char $\{ T.s = char \}$

float $\{ T.s = float \}$

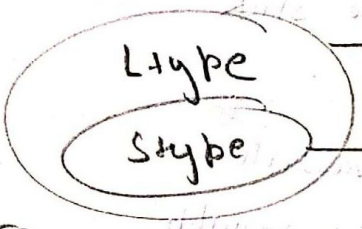
Sequence matter you cannot keep this in between

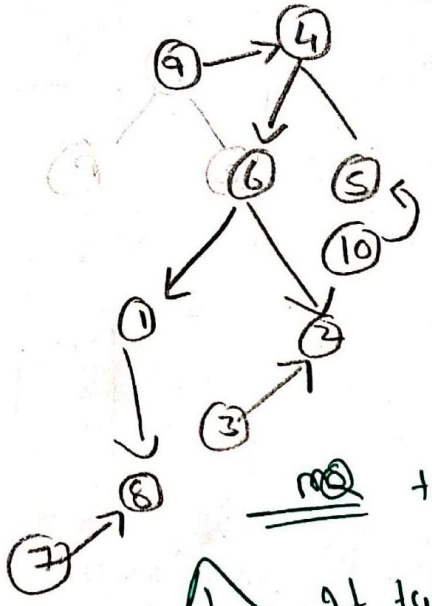
$V \longrightarrow V_1, id \}$ add space $(id, v.i)$ $v_1.i = v_0.i$

id $\}$ add type $(id, v.i) \}$;

example

// inherited SDT (all other where synthesized)





Topological sort

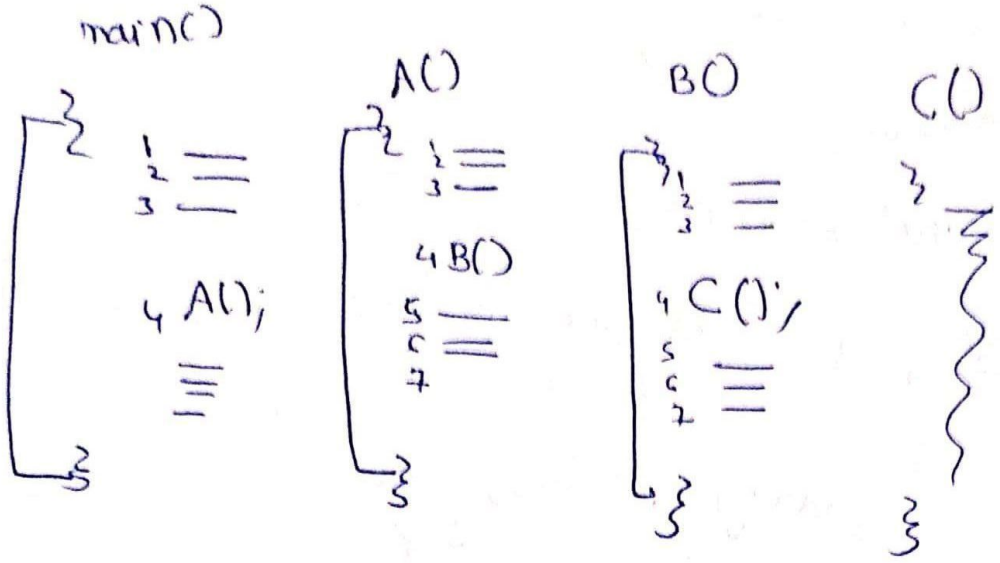
→ 9, 4, 6, 1, 7, 8, 3, 2, 10, 5

→ who will execute first check dependency task

MQ to check if L- or V

⚠ If taking from Right Sibling then not SDT

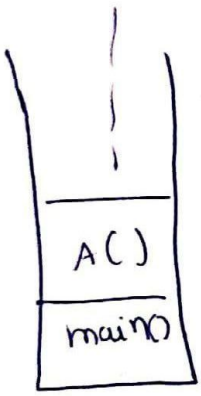
Runtime Environment



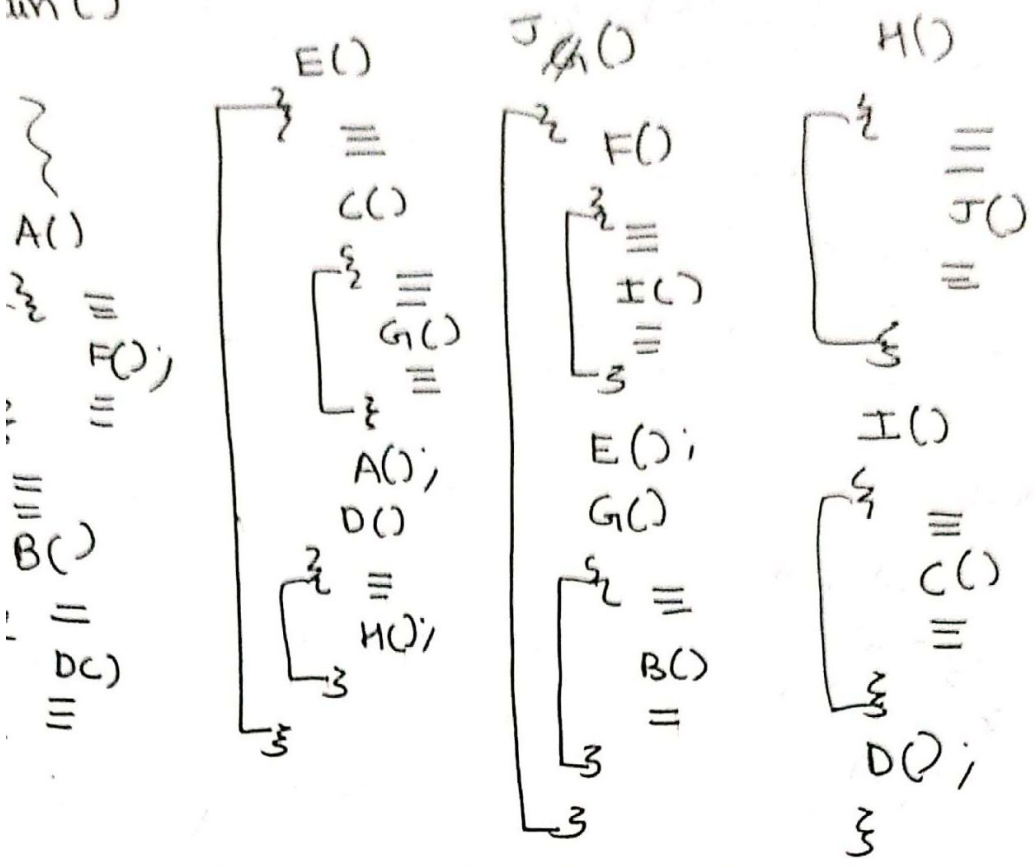
1) → when a fn is called A() its activation record is created. and it will be pushed inside the stack

2. A() fn Activation record contains

A → local variable
→ temporary variable
→ control variable
→ Access link
→ Return Status
→ Actual Parameter.
→ m/c status



Runtime Stack()
or
Control Stack();



→ If outside main the Acess code null

~~A → F → I → C~~

→ Access code of C() is in E()
link

→ ~~A~~ A code access link in main()

→ F Acess code in J()

→ H access code in main

Acess link
└ from where we accessed this code where

controlink = where you are going
└ fn calling who

→ main() acess code is null

→ G() access code is J()

G() control link B()

Storage Allocation Techniques

1. Static Storage Allocation
2. Stack storage Allocation
3. Heap Storage Allocation.

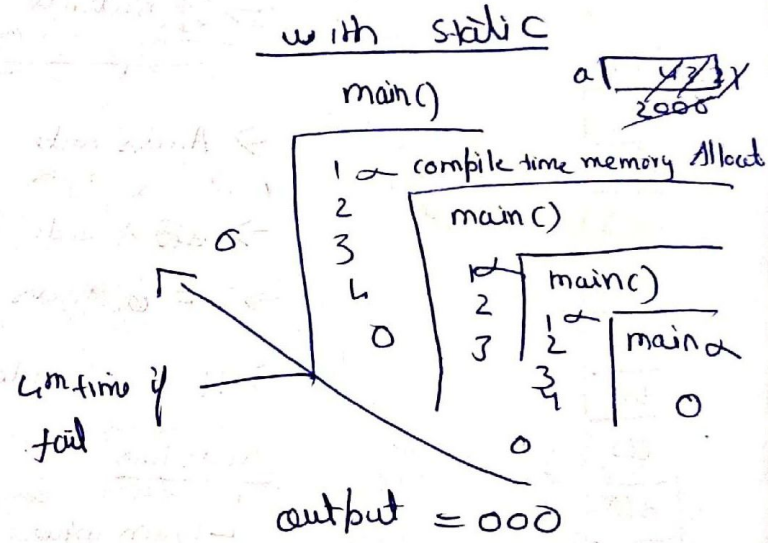
Code:

```
Main()
{
    Static int a=4;
    if(a--)
    {
      main();
      Pf(a)
    }
}
```

without static

→ Stack overflow

with static

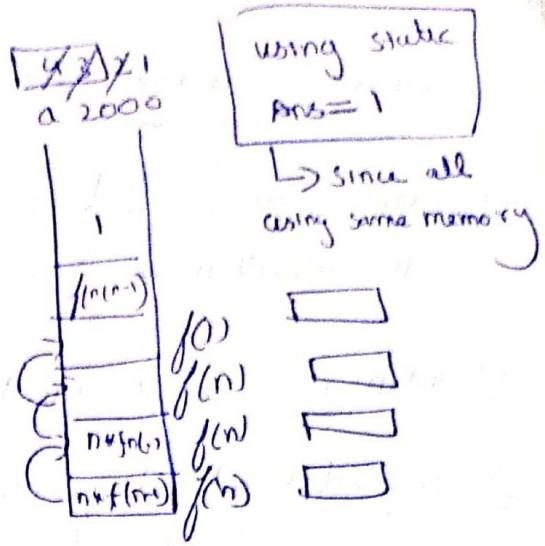


→ memor requirement estimation in static done by compiler compile time only

→ without static (asking os for memory every time)

Ex2

```
factorial /*
fact(int n)
{
    if(n==1)
    return(1);
    else
    return(n * fact(n-1))
}
```



→ on every call only parameter change.
→ memory created every time. (since not static)
→ Activation record keeps track.

→ Static memory allocation not viable in recursion as all the values will be same thus no use of f() calls

Static Storage Allocation

→ For static variable memory will create only once because memory will be created compile time.

→ If binding is done compile time then its binding cannot change runtime.

→ The drawback with the static storage Allocation ~~is~~ is recursion is not supported, thus stack come in.....

→ one more drawback with the static storage Allocation is dynamic data structure are not supported.

Stack Storage Allocation

1) when a fn is called its activation record will create and its activation record is pushed inside stack

2) When a fn over its activation record popped out of the stack

3) The greatest advantage with the stack storage ~~solution~~ allocation is recursion is supported

4) Drawback with the stack storage solution is Dynamic Data Structure is not supported

* one more drawback with stack storage is once a fn is completed it will be popped out of the stack that means we cannot get it back if required afterwards.

← only top will be printed if P(n)

— cannot acess local variable

-> One more drawback with stack storage allocations is always local variable are belong to new activation record only

// major Advantage recursion

Disadvantage (above)

Heap Storage Allocation

→ Allocation deallocation will be done anytime depending upon requirement

→ on c language — malloc() // Allocation
free() // Delete

C++ — new()
delete()

JAVA — new()
— (Garbage collector will take care)

→ Recursion is supported,
→ Dynamic Data structure is supported

Intermediate code Generation (every year 1 question)

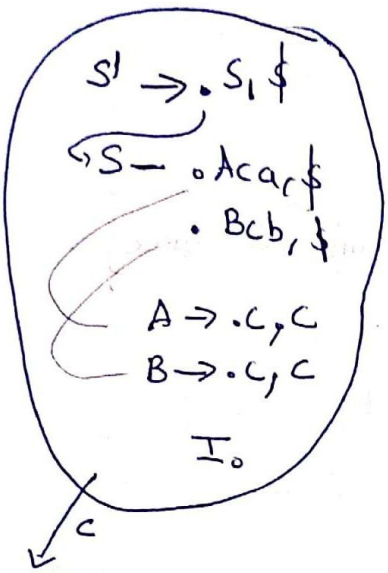
WORKBOOK – 2

1. d 3. a 5. d
2. d 4. a 6. a

7: G: $S \rightarrow Aca \mid Bcb$

$A \rightarrow c$

$B \rightarrow c$



$S \rightarrow Aca \mid Bcb$

$A \rightarrow c$

$B \rightarrow c$

LL(1) ✗

LL(2) ✗

→ expand min 2 production and 1 reduction

CLR(1) Fail then ↑

---

8. c 9. b 10. d 11. b 12. c 13. b 14. c

15. c 16. c 17. b

Remove unreachable or non existing productions.

Remove Reachable but not generating.

$S \rightarrow CA$

$A \rightarrow a$

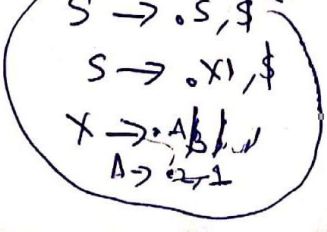
$C \hookrightarrow$

18. c 19. a 20. a 21 b 22 c 23. b 24 b 25 d

26. b 27 c. 28. c 29. a 30. a 31. c 32 d

33. c 34. a 35. b 36. c

37. a 38. c 39. c 40. c 41. a



42. C
43. C
45. d
~~46. d~~
46. b
47. b
48. c
49. a
50. a
51. d
52. c
53. a
54. b
55. e
56. a
57. c
58. a

44 — Domination of CFG

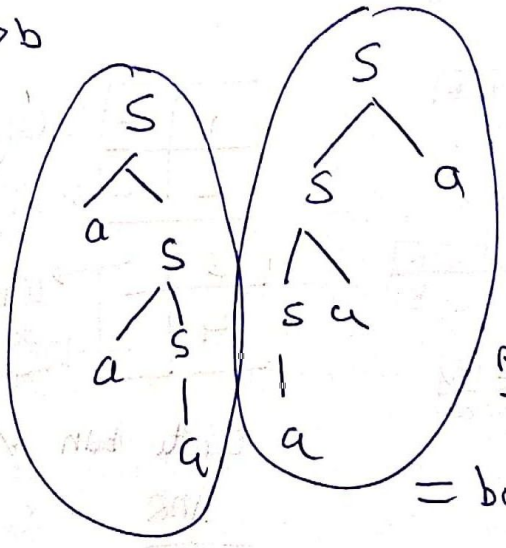
$x = (a, b)$

$x = a, b$

(Shift Reduce — Bottom up parser)

59. b	65. b
60. b	66. c
61. a	67. c
62. b	68. c
63. b	69. a
64. d	

T1) $S \to AB$
$A \to a$
$B \to b$



Left most derivation tree

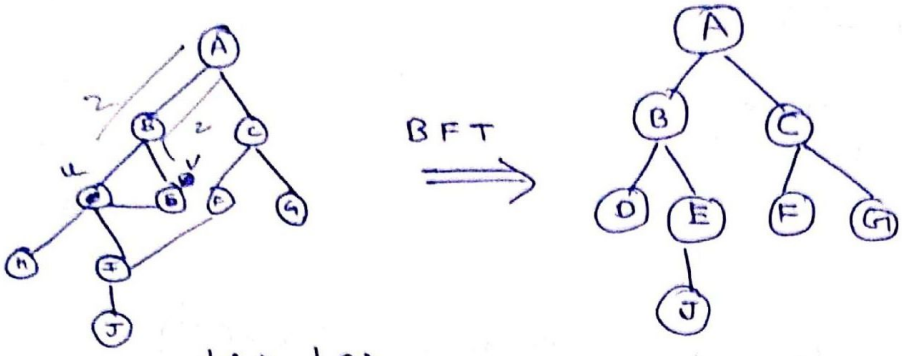
Both are left most derivation tree since there is only one variable / Hence both are same

Right most derivation tree

= both are same Hence 1

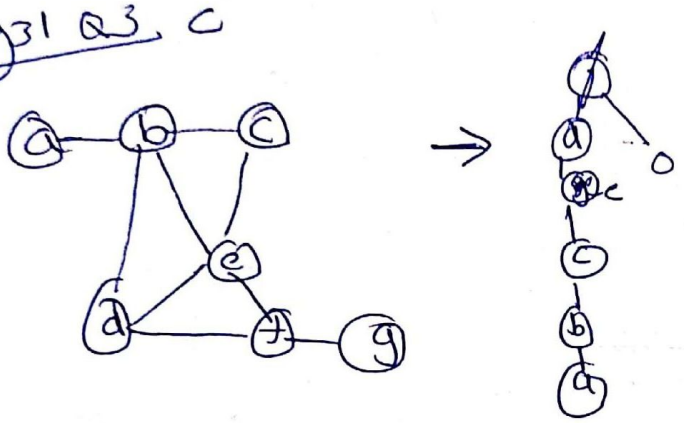
Chapter – 4 (Algo)

Pg 34 QT 1



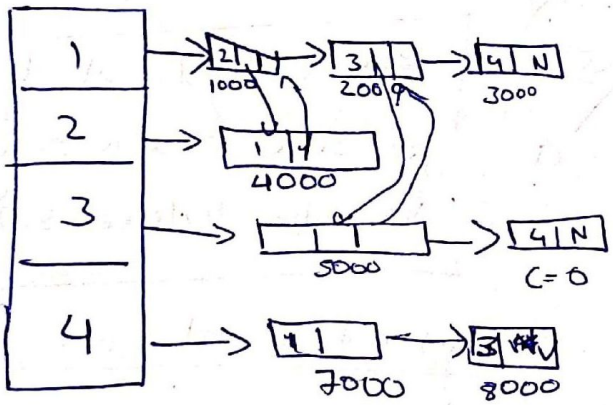
d(u) d(v) =

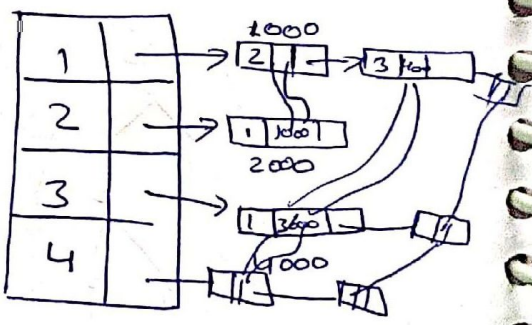
Pg 31 Q3, C



decbag

Pg 30 QT 9





Create both of them and link.

b) $O(n+m)$

Intermediate code Generation

ex

$x = a + b * c$

$\Downarrow$

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

→ The advantage of intermediate code generation is that you will get machine independent code (Portability)

Representation of intermediate code

ex $x = (a + b * c) - (a - b * c)$

- Tree form
  - DAC
    - Common subexpression elimination
  - Syntax tree
- Linear Form
  - Postfix notation: $x\,a\,b\,c\,*\,+\,a\,b\,c\,*\,-\,-\,=$
  - 3 address code

$t_1 = b * c$

$t_2 = a + t_1$

$t_3 = b * c$

$t_4 = a - t_3$

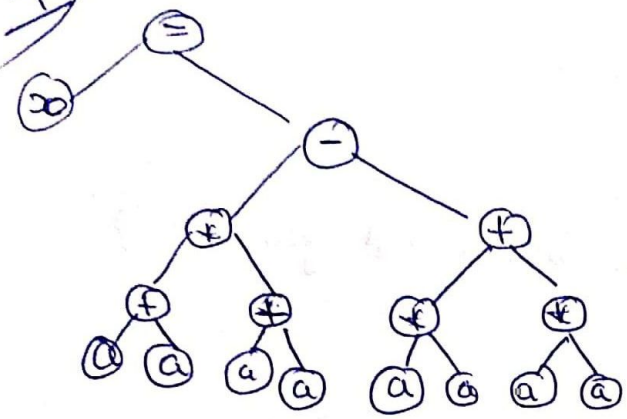
$t_5 = t_2 - t_4$

$x = t_5$

Ex 2 ~~x=(a+b*c)-(a-b+c)~~

$x=((a+a)*(a+a))+((a+a)+(a+a))$

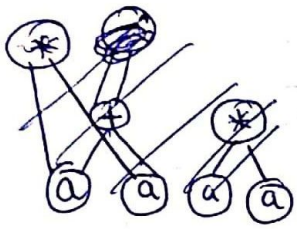
I)



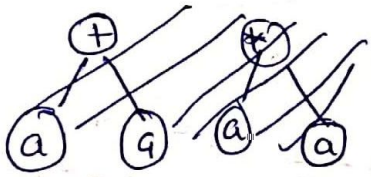
II

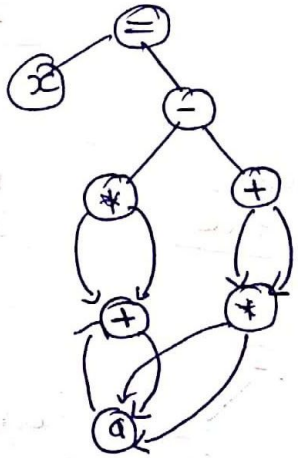
$t_1 = a+a$
$t_2 = a+a$
$t_3 = a+a$
$t_4 = a+a$
$t_3 = t_1 * t_1$
$t_4 = t_3 + t_2$
$t_5 = t_3 - t_4$
$t_6 = t_5$
$x = t_5$

$t_1 = a+a$
$t_2 = a+a$
$t_3 = t_1 * t_2$
$t_4 = a+a$
$t_5 = a*a$
$t_6 = t_4 + t_5$
$t_7 = t_3 - t_6$
$x = t_7$



III)

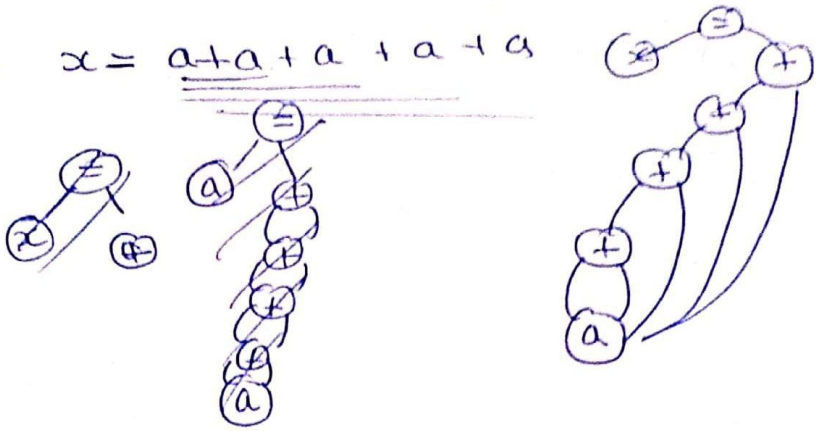




IV) Postfix

$x\ a\ a+a\ a+*\ a\ a+a\ a+*+-=$

Ex3 $x = a+a+a+a+a$



3 - Address code

Types of 3 address code

1. $x = y \text{ op } z$
2. $x = \text{op } y$
3. $x = y$
4. if $a \leq b$ goto c
5. goto c
6. $x = a[i]$
7. ~~$x = a[i][j]$~~ not 3 address
8. ~~$x = f(a,b)$~~

Static Single Assignment = $t_1 = a+b$

~~x~~ $t_1 = a-b$ ✗

not possible

t_1 can only support one value

Ex if $a<b$ and $c>d$ then $e = f$ else $g = h$

```
1000> if a<b goto 1003
1001> g = h
1002> goto 1006
1003> if c>d goto 1005
1004> goto 1001
1005> e = f
1006> ____
```

Ex

```
for(k=10; k<=100; k++)
{
    x = a + b * c;
}
```

Ans

```
1000  k = 10
1001  if k<=100 goto 1002
20000 goto 1007
1002  t1 = b * c
1003  t2 = a + t1
4)    x = t2
5)    k = k + 1
6)    goto 1001
7)    ____
```

Back Patching

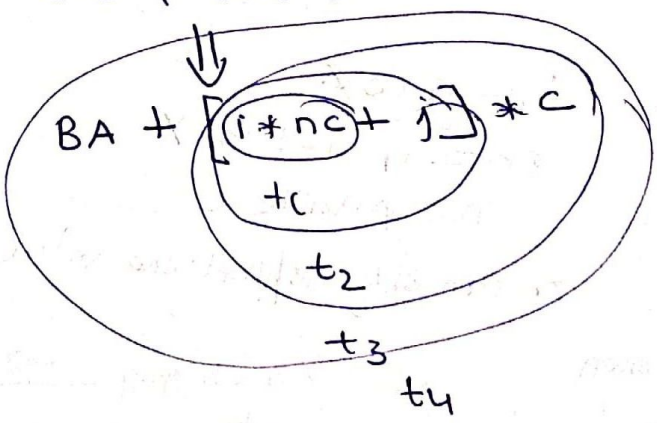
→ Filling back blank after going forward

Q Give 3 address code for the following statement.

$x = a[i][j]$

Ans:

$x = LOC(a[i][j])$



$x = *t_4$

Control flow Graph (CFG)

① $a = b + c$ L1
② $t_1 = b + d$

Basic Block 1

③ if $a < b$ goto 12 L4 — B_2

④ $c = d - e$ L6
⑤ $e = f + g$
⑥ if $d - e$ goto 9 — B_3

⑦ ~~goto~~ $c = x + y$ L7
⑧ $d = y + c$ — B_4

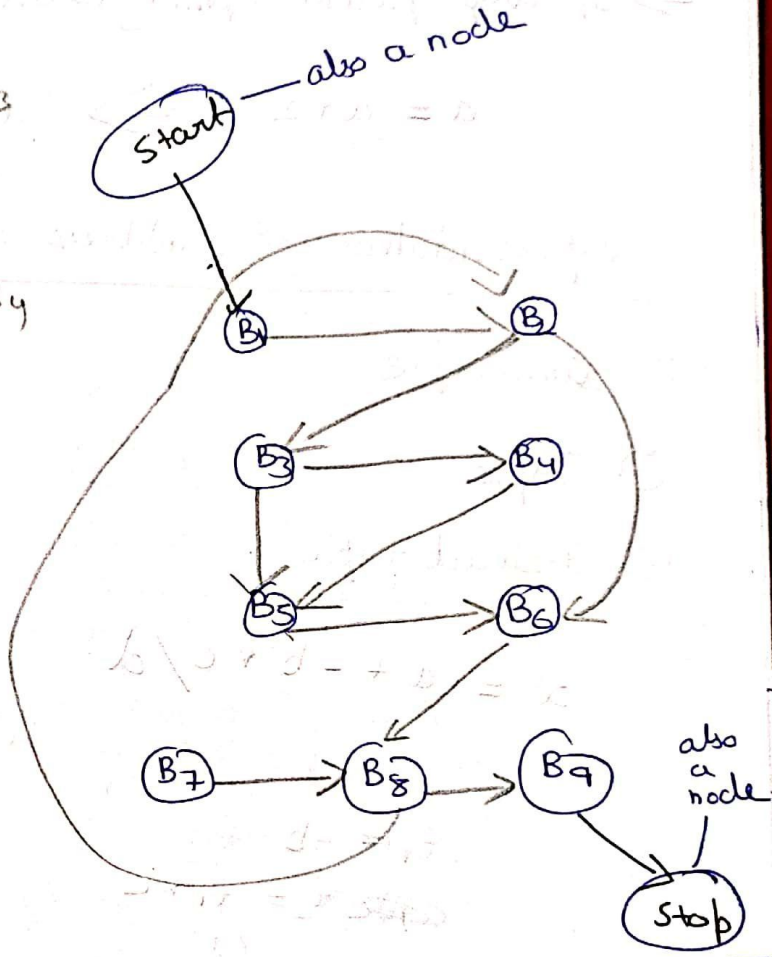
⑨ $y = x + c$ L9
⑩ $z = a + b$
⑪ $a = b + c$ — B_5

⑫ goto 15 L2 — B_6

⑬ $a = c - d$ L8
⑭ $d = a + c$ — B_7

⑮ if $d - e$ goto 3 L5 — B_8

⑯ $e = g + h$ L4
⑰ $f = e - g$ — B_9



Gate
Q How many nodes in CFG
nodes + start + stop

How to find a Leader

① 1st statement leader
② Target of goto is leader
③ Next statement after goto is leader

⇒ Total leader = 9

Basic Block = one leader to statement before next

No. of leader = No. of Basic Block

⇒ If loop present optimization should be done

$a = a + 2 \quad \Rightarrow \quad a = a + 2$

Representation of 3 address code

① Quadraple

② Triple

③ Indirect triple

$x = a + -b * c / d$

⇓ 3 address code.

$t_1 = -b$

$t_2 = t_1 * c$

$t_3 = t_2 / d$

$t_4 = a + t_3$

$x = t_4$

Quadraple

(Advantage = can move results anywhere)

S.no	operator	operand 1	operand 2	result
1	−	b		t_1
2	*	t_1	c	t_2
3	/	t_2	d	t_3
4	+	a	t_3	t_4
5	=	t_4		x

Triple

S.no	operator	operand 1	operand 2
(1)	−	b	
2	*	(1)	c
3	/	(2)	d
4	+	a	(3)
5	=	x	(4)

⇒ Result binded to memory allocation.

Indirect triple

(after triple, create Indirect table and move results)

S.no	
(1)	
(2)	⟹ 5000
(3)	
(4)	⟹ 1000
(5)	