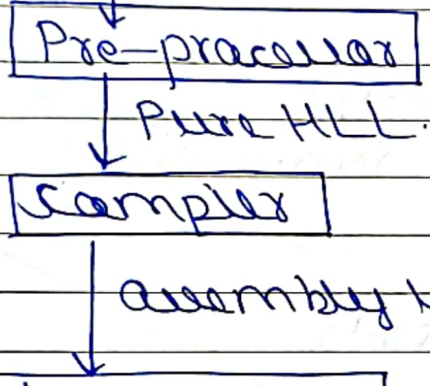
COMPILER DESIGN

* INTRO & VARIOUS PHASES OF COMPILER

Compiler : HLL ⟶ LLL

– we can't write in 0/1. (best Eng.)

* HLL



#<include> } HLL
#<define>

↓ pre-proc. removes by
– file inclusion
– macro
– expansion.

Pure HLL: no #

↙ pre-proc. directives.

– H/w → OS (platform)

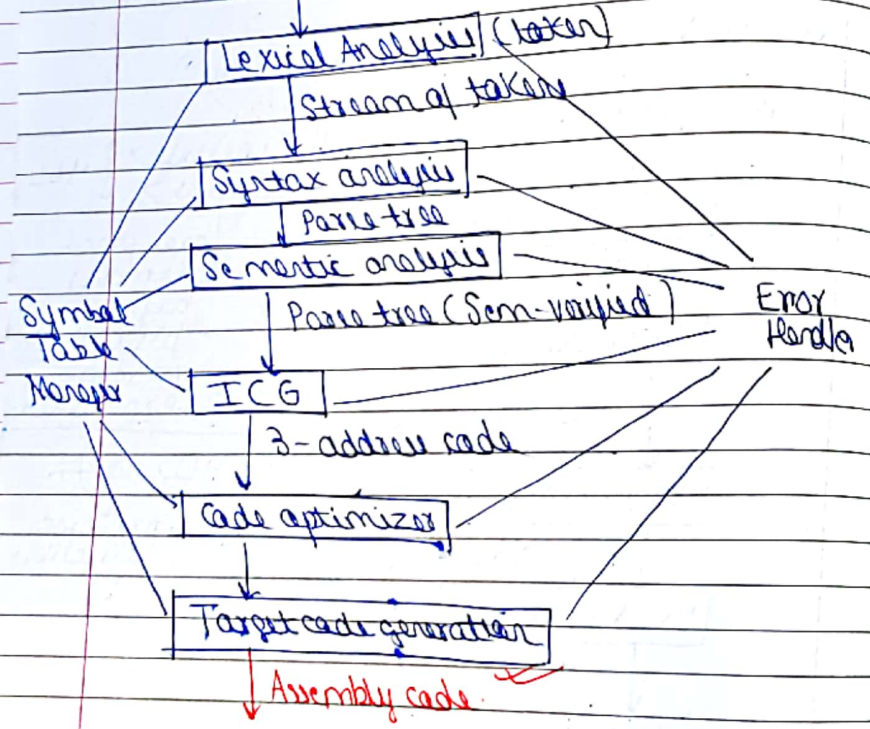
– Assembler: platform independent.

→ diff m/c → diff assembler

relocatable: can load in any part of PC & run it.

- Gcc/turbo: all these phase are included

* Phases of compiler



* ICG → Intermediate code generator

HLL

eg: x = a + b * c ; ← Source Prog.

Lexical analyser (LeX) → S/w tool
- convert into tokens
- remove whitespace
- remove comments
→ tokens

FA

Stream of lexims

- identifies using RE (patterns)
Id: l(l+d)*
- pattern known to LA b4

id = id + id * id (token String)

Rules

Syntax analyser (Yacc) [Yet another compiler compiler]

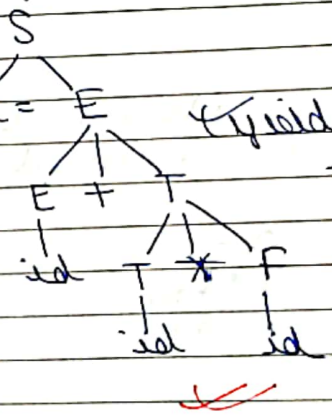
S → id = E ;
E → E + T / T
T → T * F / F
F → id

PDA

PT.

done by parser.

(checks format)



Yield of PT = i/p to SA (Sem
- Syntax errors can be detected.

id = id + id * id

↓

Syntax analyser (type checking)

↓

(Semantically verify PT)

↓

ICG

↓

$t_1 = b * c;$
$t_2 = a + t_1;$
$x = t_2;$

} 3 address (max).

↑ m/c independent
stage only last 2 phase to make a new compiler.

↓

CO → reduce no. of lines.

$t_1 = b * c;$
$x = a + t_1;$

↓

TCG : write code which assembler can understand.

↓

mul R_1, R_2 . $a \rightarrow R_0$
add R_0, R_2 $b \rightarrow R_1$
mov R_2, X. $c \rightarrow R_2$

- Theoritically

Lex
↓
Syntax
↓
~~Semantic~~
↓
ICG
↓
CO
↓
TCG

- Practically, most are done by Syntax Analyser.

∴ till ICG : front end
remaining : back end (CO & TCG)

- intro FE implemented together in

- LANCE : implem. of FE → code → BE. VI.

Symbol Table

- DS (Data Structure)
- info is stored & used whenever req.
- created & main by compiler
- info about occurrence of var. names, func. names, classes (entities are stored)
- collected by analysis phase, & used by synthesis phase

↓

LA, SA, SA

understood prog & store in ST.

Page No. 5
Date

Scanned with CamScanner

1. LA : create new entries for identifiers
2. SA : add info : type, scope, dimension, line of reference, use.
3. SA : use avail. info to check for semantics & is updated

type matching
- a = 10
- type checking.
- get into ST, & use entry of a.

4. ICG : add temp. var. to code

x = y + z;
t = y + z; x = t;

depends upon type of result.

- look into ST and get req. info.

5. CO :

- info from ST.

eg: x = y + z
↓
t1 = y + z
x = t1;
↓
replace t1 with y + z : CO.

6. TCG : using address info from identifiers & generates code.

A[i]
↓
*(A + i)

- Every phase uses ST. → VI

* ST Entries.

- attrib. that support compiler in diff phase

1. Name
2. Size
3. dimension (array)
4. Type (int/char...)
5. Line of declaration
6. Line of usage
7. Address

(4, 5, 6 updated by Syntax Analyser (where error has occur.))

EX: Name Type size dim line of dec L.U Add.
Ravi char 4 1 ... ...

- i → used at 10 places
- LOU : implemented as LL. (Linked list)
→→→

- All attributes → diff size.

* ST

(fixing size.)

- of size limitation (drawback)
  if big: waste space
  small: can't store many var

(use dynamic allocation)
→ increase size at compile time.

* Operations of ST : → depend on type of language.

- Block Structured / non-block structured lang.

```
{ int i;
    }
  }.

{ int i;
    }
```

- int i;
- for every var: 1 decl.
- Scope: end of prog

→ NBS

· contains 1 init. of var. & scope is throughout the prog.
→ insert entry
→ lookup entry

BS {

}
-when

· insert
· lookup
· set (when enter new block)
· reset

→ Var. may be declared & scope is within the block.

· Gate 2012 on ST

{ Insert, LookUp }

Access time will be logarithm in ST if it is implemented by a:

(a) Linear List : O(n)
(b) Search tree / BST : O($\log_k n$) K: children
(c) Hash Table : O(1)
(d) None of the above.

depends upon ht of tree.

* Various implementations of ST

(1) Linear List:

(a) Ordered list
- Array ✗ (static)
- LL ✓

Array: IT: O(n).
LT: O(log n). (BS)

IT = Insertion time
LT: Look up time
→ no duplicates.

LL: IT: O(n)
LT: O(n) (no BS in LL)

(b) Unordered list (not sorted).

IT: O(1)
LT: O(n). (Linear Search)

(2) Self organising List:

- Normal LL, if we find, move to head.
(Eg: accessed data → beginning of list).

IT: O(1) → unordered
LT: O(n)

(3) Search tree

IT: $O(\log_K n)$
LT: $O(\log_K n)$

$\propto$ ht of tree.

K → K-ary tree

(4) Hash-table

IT: O(1)
LT: O(1)

if bad implementation → collisions

Disadvantages

(1) Linear list.

- $\propto$ to size
- Insertion preceded with lookup. → VI

(2) when less freq. items are searched: Poor perfo.

→ O(n)

(3) we always have to keep it balanced.
$O(\log n) \rightarrow O(n)$: Skew

(4) when there are too many collisions.
↑ O(n).

Page No: 13

* Lexical Analyser → uses RE, sa: FA

- reads char by char
-
  ↓
  tokenization
- only at this phase, lexeme exist → a+3*4.
  → eliminate whitespace / comments.
  → convert lexeme → tokens.
  → line no. & all detected. (show error) → if not occur to patterns

Q. |int|max|(|x|,|y|)| 7

|int|x|,|y|;| 5

/* . . . */

|{| 1
|return|(|x|>|y|?|x|:|y|)|;| 11
|}| 1

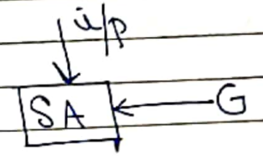
→ 25 tokens ✓

Q. |printf|(|"%d", & i"|,|&|x|)|;| (3)

→ 8 tokens ✓

Q. how many tokens? → Gate Qs.

* Syntax Analyser → Parser → generates a Parse tree.



i/p checked accor to grammar.

G = (V, T, P, S) ✓

eg: E → E+E | E*E | id. ← Production rules.

V = {E}
T = {+, *, id}.

String: id + id * id.

E → E+E (LMD) (1)
⇒ id+E
⇒ id + E*E
⇒ id + id * id.

E → E+E (RMD)
⇒ E+E*E
⇒ E+E*id
⇒ E+id*id
⇒ id+id*id

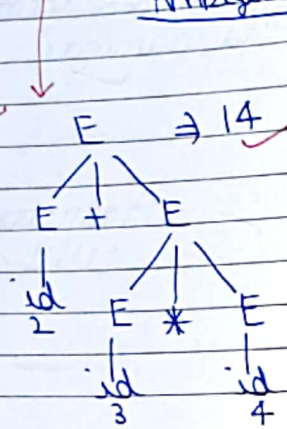
E → E*E (LMD-2).
E+E*E
id+E*E
id+id*E
id+id*id.

High priority

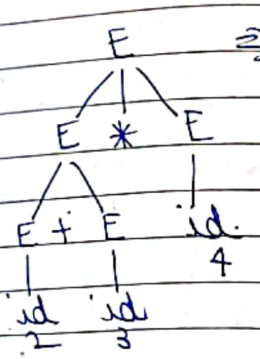
∴ More than 1 LMD / 1 RMD / 1 PT is possible.

↳ Ambiguous Grammar.

eg: E ⇒ 14



E ⇒ 20 ✗



∴ For a given String / Grammar, more than 1 LMD / RMD / PT : amb.

- Parser get confused which one is right

VI

↳ 14 / 20 - which is right?

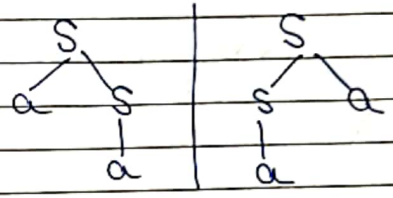
eg: 2+3*4 ⇒ 14.

(Almost every Parser doesn't allow amb. Grammar)

(i) how gram. is amb. } undecidable.
(ii) amb → unamb.

Q S → aS | Sa | a.

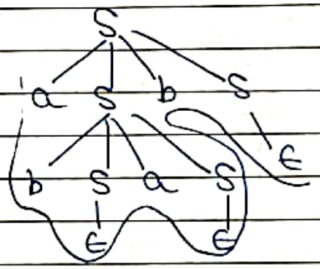
L = aa



∴ ambiguous

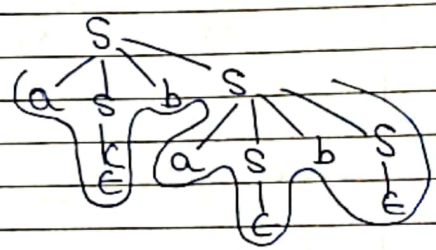
Q S → aSbS | bSaS | ∈

L = abab



yield = abab

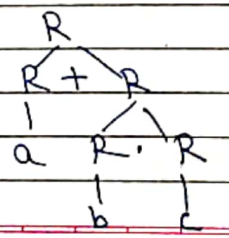
∴ amb.

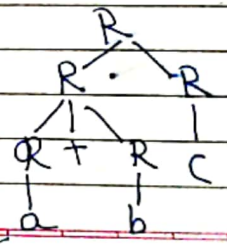


yield = abab

Q R → R+R | R·R | R* | a | b | c → all RLS.

a+b*c





Ambig Grammar

* Errors in LA

- Errors detected in LA → works with RE. (patterns)

1. Numer. literals are too long.

int A = 12345678;
> 65,535.

2. Long identifiers. X
3. Ill formed numeric literals: int a = $123;
4. I/p char that are not in source lang.
eg: Curren Symbols (Enlish)

Error Recovery → done by LA.

1. delete: unknown char are delete. (panic mode rec.)
eg: char → chr.

2. insert:
eg: cha → char.

3. transpose:
whiel → while.

4. replace
chrr → char

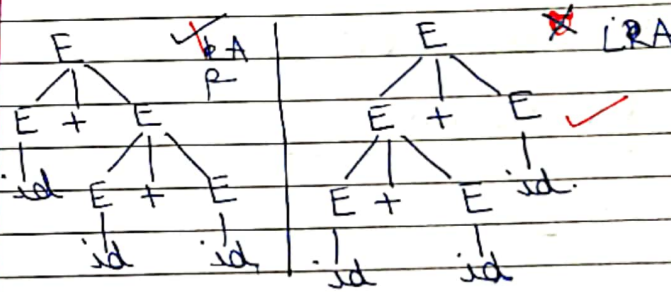
(if cant recover stop compil. tell user error in LNo.)
↳ char by char scanning.

* Amb. → Unamb. → undecidable.

(disambiguity rules)

eg: E → E+E | E*E | id

• id + id + id (Assoc.)

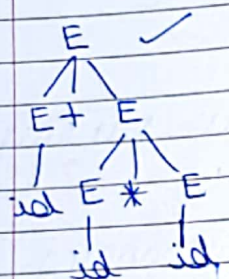


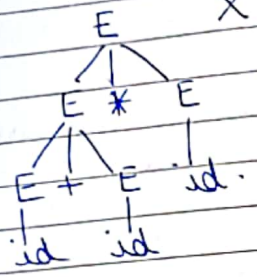
∴ 2 PTs

LM (+) evaluated first

E → E+T | T.
(left most derived)

· id + id * id





(+ wrd. first)

∴ 2 PTs.

+ → LA

* Reason:

→ 2 op/s on either side

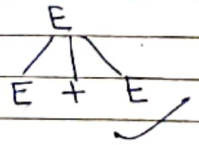
(id + id) + id → LA

id + (id + id) → RA

→ * > + (precedence)

↳ evaluated first

* Associativity

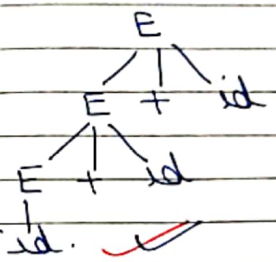


+ → Left Associative
* → Left Associative

(I) → Left recursive Gramm.

∴ E → E + id | id . (restrict growth in RHS)

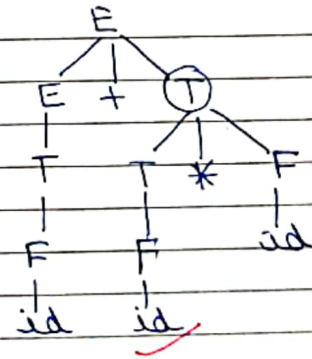
id + id + id



← only PT.

(II) → Highest prec. oper. in last level.

E → E + T | T : LR. (+ → LA)
T → T * F | F : LR (* → LA).
F → id



→ finish gen + in higher level.

(VII)

→ takes care of associat : Recursion
takes care of precedence : levels.

eg: $2 \uparrow 3 \uparrow 2 = (2^3)^2$

↳ R. Associative. VI

$F \rightarrow G \uparrow F \mid G$ (RR)
$G \rightarrow id$

$+, *, \uparrow$.

$\uparrow > * > +$ (precedence).

$E \rightarrow E+T \mid T$ } LR. ($+ \rightarrow LA$)
$T \rightarrow T*F \mid F$ } LR ($* \rightarrow LA$)
$F \rightarrow G \uparrow F \mid G$ } RR ($\uparrow \rightarrow RA$)
$G \rightarrow id$

Q bExp → bExp or bExp |
bExp and bExp |
not bExp |
True |
False

not > and > or : Precedence

or, and → LA
not → unary

bExp — remove ambiguity. VI

$E \rightarrow E \text{ or } F \mid F$
$F \rightarrow F \text{ and } G \mid G$
$G \rightarrow \text{not } G \mid \text{True} \mid \text{False}$

Q $R \rightarrow R+R \mid R.R \mid R* \mid a \mid b \mid c$

$* > . > +$

unamb Gramm.

$E \rightarrow E+T \mid T$ LR
$T \rightarrow T.F \mid F$ LR
$F \rightarrow F* \mid a \mid b \mid c$

GATE.

LR Grammar.

Q $A \rightarrow A \$ B \mid B$ LR ($\$ \rightarrow LA$) ①
$B \rightarrow B \# C \mid C$ LR ($\# \rightarrow LA$)
$C \rightarrow C @ D \mid D$ ($@ \rightarrow LA$)
$D \rightarrow d$

~~evaluated first~~
$-\$-\$$

$\$ > \$$
$\# > \#$
$@ > @$

② $@ > \# > \$$
precedence

unambiguous Grammar

Q $E \rightarrow E * F \mid F$ (LA) $+, *$ → same level.
$/F + E$ (RA)
$/F$
$F \rightarrow F - F$
$/id$

$\therefore * = +$ $\quad * > *$ (assoc)
$+ < +$

$- > + = *$ (prec)

- : assoc not defined (Left / right)
$\therefore$ Ambiguous.

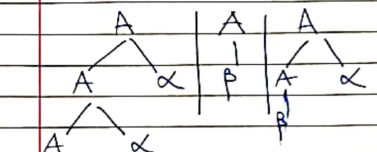
Lecture 2

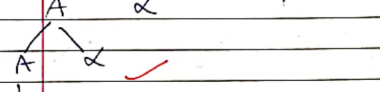
* Elimination of left Recursion & left factoring the Grammars

Recursion (V calling itself)

LR . RR.

$A \rightarrow A\alpha / \beta$ $\quad A \rightarrow \alpha A / \beta$

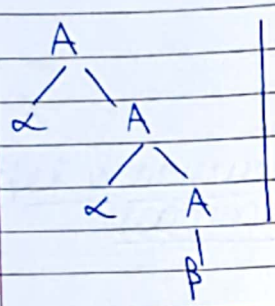




$\therefore \beta\alpha^* \rightarrow$ LRG

```
A()
{
  A()
  α
}
```
→ lead to infinite loop

```
A()
{
  α
  A()
}
```
→ cond checking.



LRG ≠ RRG

∴ $\alpha^*\beta$ → RRG

* Top Down Parsers can't work with LRG.
(eliminate it) → ∞ loop.

$L = \beta\alpha^*$.

· $A \rightarrow \beta\alpha^*$ X G. $A \rightarrow A\alpha|\beta$

$A \rightarrow \beta A'$
$A' \rightarrow \epsilon \mid \alpha A'$: RRG $\Leftrightarrow$ $A \rightarrow \alpha A|\beta$

Q $E \rightarrow E+T \mid T$ LRG

A A α | β

⇒ $E \rightarrow TE'$
$E' \rightarrow \epsilon \mid +TE'$ RRG equivalent to LRG.

Eliminate left recursion

· RRG → LRG

Q $S \rightarrow S0S1S \mid 01$ LR

A A α β

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

A
$S \rightarrow 01S'$
$S' \rightarrow \epsilon \mid 0S1SS'$

Eliminate Left Recursion

· $A \rightarrow A\alpha|\beta$

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

Q $S \rightarrow (L) \mid x$
$L \rightarrow L,S \mid S$
A A α β

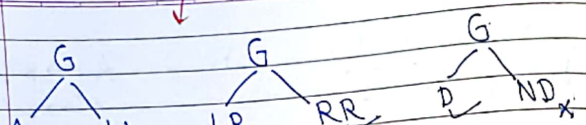
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \epsilon$

Q $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots \mid \beta_1 \mid \beta_2 \mid \beta_3 \cdots$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \cdots$
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \cdots \mid \epsilon$

VI

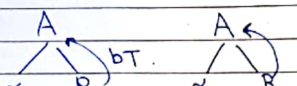
Grammar, I am... are comfortable with.



* 1) Ittermin. & ND Gramm.

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ (NDG). → bT.

$\alpha\beta_3$



backtracking (due to comm. prefixes).

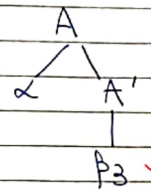
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 | \beta_2 | \beta_3$

Left factoring (postponed the dec: making)

$\alpha\beta_3$

DG

- NDG → DG
- left factoring.



NDG: most top Down Parsers fail & bT

if Exp. then S else S.

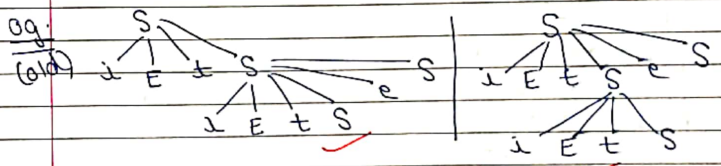
Q $S \rightarrow iE\,t\,S \mid iE\,t\,S\,e\,S \mid a$
$E \rightarrow b$

Sta → NDG & ambig.

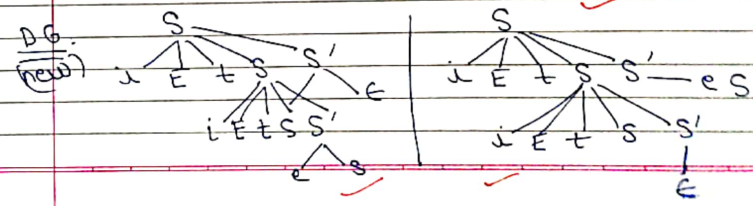
$S \rightarrow iE\,t\,S\,S' \mid a$: DG. (b → boolean exp.)
$S' \rightarrow \epsilon \mid eS$
$E \rightarrow b$.

iEtiEtSeS (interm. string).

Eg: (old)



DG: (new)



- [Applying left factoring doesn't remove ambiguity; it removes non-determinism] ✓ VI

→ Ambiguity due to assoc. of if & else

Q. $S \rightarrow aSSbS \mid aS\,aSb \mid abb \mid b$

→ $S \rightarrow aS' \mid b$ ← DG
• $S' \rightarrow SSbS \mid SaSb \mid bb$

→ $S' \rightarrow SS'' \mid bb$
→ $S'' \rightarrow SbS \mid aSb$

Q. $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \rightarrow bSS' \mid a$ VI
$S' \rightarrow SaaS \mid SaSb \mid b$

$S \rightarrow bSS' \mid a$
$S' \rightarrow SaS'' \mid b$
$S'' \rightarrow aS \mid Sb$

← deterministic Grammar [no comm prefix]

Left Factoring method

24 Syntax Analysis

* INTRO TO PARSERS AND LL(1)

PARSING

(Amb G. X)

- Parsers
  - Top Down (TDP)
    - TDP with full BT
    - TDP without full BT
      - Recursive descent
      - Non-recursive descent LL(1)
  - Bottom up (SR parser) (BUP)
    - op. precedence (AG ✓)
    - LR parser
      - LR(0)
      - CLR LR(1)
      - LALR

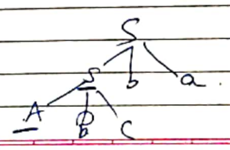
· NDGX.
· LRX

Token

ϕ Hrituja ← Gra

P↑

$S \rightarrow Sbal$

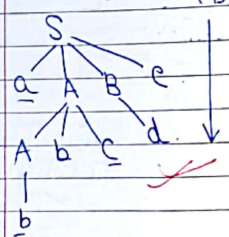


eg: Parser checks if the string is gen. by grammar by PT

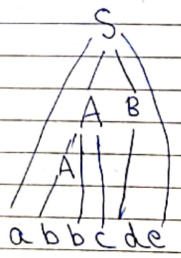
eg: S → aABe
A → Abc | b
B → d.

w: abbcde

TDP → follows LMD.
→ dec. of what to use.



w: abbcde.

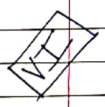


BUP → dec. of what (when to reduce)
→ reverse of RMD

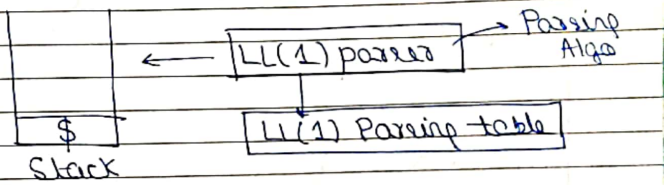
S → aABe
→ aAbcBe
→ abbCBe
→ abbcde → LMD manner

S → aABe
→ aAde
→ aAbcde
→ abbcde → RMD (reverse) (bottom up)

* LL(1) → no. of look aheads.
→ LMD.
→ how many symbols will see (to make dec)
→ scan. i/p from left to right
→ abcd

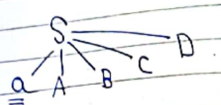


i/p buffer [$] → to guess when to stop



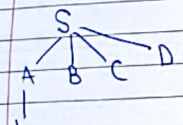
Functions → First() → if all string der. from V, the first symbol is called.
→ Follow()

eg: S → aABCD
A → b
B → c
C → d
D → e



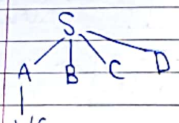
∴ first(S) = a.
first(A) = b
first(B) = c.

ii) aX.



∴ first(S) = b

A → b|ϵ.



first(S) = b|c.

eg: i/p → abcd$ → i/p buffer

S → ABCD$

ABcD$ follow(B) = d.

first. (follow doesn't contain ϵ).

VI

eg: S → ABCD
A → b|ϵ
B → c
C → d
D → e

follow(S) = {$}
follow(A) = c
follow(B) = first of CD ⇒ d.
follow(C) = e.
follow(D) = follow of S ⇒ {$}

eg: A → BC

* Examples on how to find first() and follow() → LL(1) Parsing.

S → ABCDE
A → a|ϵ
B → b|ϵ
C → c
D → d|ϵ
E → e|ϵ

follow(B): first of CDE.
(whereever a var is in RE, then follow is follow of LHS.)

	first	follow
A:	{a, ϵ}	{b, c}
B:	{b, ϵ}	{c}
C:	c	{d, e, $}
D:	{d, ϵ}	{e, $}
E:	{e, ϵ}	{$}
S:	{a, b, c}	{$}

Start	Stop

Q. S → Bb | Cd
B → aB | ∈
C → cC | ∈

first(S) = first(B), first(C)
a, b   c

	first	Follow
S :	{a,b,c,d}	{$}
B :	{a,∈}	{b}
C :	{c,∈}	{d}

Q.

	first	follow
E → TE′	{id, (}	{$,)}
E′ → +TE′ | ∈	{+, ∈}	{$,)}
T → FT′	{id, (}	{+, $,)}
T′ → *FT′ | ∈	{*, ∈}	{+, $,)}
F → id | (E)	{id, (}	{*, +, $,)}

Q.

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

Q.

	first	follow
S → aABb	{a}	{$}
A → c | ∈	{c,∈}	{d,b}
B → d | ∈	{d,∈}	{b}

Q.

	first()	follow()
S → aBDh	{a}	{$}
B → cC	{c}	{g,f,h}
C → bC | ∈	{b,∈}	{g,f,h}
D → EF	{g,f,∈}	{h}
E → g | ∈	{g,∈}	{f,h}
F → f | ∈	{f,∈}	{h}

* Construction of LL(1) P Table.

~~we~~ construct LL(1) P.T

→ T.

V ↓

	id	+	*	(	)	$
E	E→TE′			E→TE′		
E′		E′→+TE′			E′→∈	E′→∈
T	T→FT′			T→FT′		
T′		T′→∈	T′→*FT′		T′→∈	T′→∈
F	F→id			F→(E)		

• [LS th = row, col = first of RHS]
if ∈ → follow of LHS.

- to construct PT. LL(1) PT is made

L = (()), ...

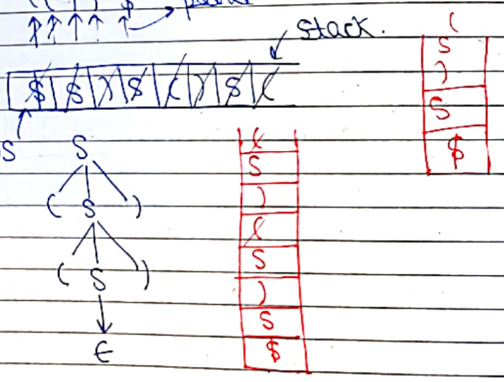
Q. $S \to (S) \mid \epsilon$ (,), $

first follow
(, ε), $

	(	)	$
S	S→(S)	S→ε	S→ε

- if mult entries in 1 cell → Gramm not feasible.

* (()) $ ← pointer.



{[Some of Gramm → cant construct LL(1) parsing table]}

{a},

Q. $S \to AaAb \mid BbBa$ {a, b} {$}.
$A \to \epsilon$ {ε} {a, b}
$B \to \epsilon$ {ε} {a, b}

LL(1).

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

VI · if LR X, ND X } cant be used for LL(1).

eg: $A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \alpha_n$.

	c d
A	

① (if LL(1), first of α_1, first of α_2 ... must be mutually exclusive)

② $A \to \alpha \mid \epsilon$
first of α ; follow of A . no multi entries in same cell.
if '=' X ∩ .
(mutually exclusive)

VI

{a, b} · {$, a, b)

(Amb)

Q. S → aSbS | bSaS | ∈

{a} {b} {a, b, $}

X

Q. S → aABb — S.raw, cal {a}
A → c | ∈ — {c}, {d, b}.
B → d | ∈ — {d}, {b}.

✓ LL(1)

Q. S → A | a — ambiguous — {a}, {a} [Same cal]
A → a

X LL(1)

Q. S → aB | ∈ — {a}, {$}
B → bC | ∈ — {b}, {$}.
C → cS | ∈ — {c}, {$}

✓ LL(1)

→ row

Q. S → AB
A → a | ∈ — {a}, {b, $}
B → b | ∈ — {b}, {$}.

✓ (LL1 Parser)

Q. S → aSA | ∈ — {a}, {c, $}.
A → c | ∈ — {c}, {c, $}

X

first follow

Q. S → A
A → Bb | Cd — {a, b}, {c, d}
B → aB | ∈ — {a}, {b}
C → cC | ∈ — {c}, {d}

✓

Q. S → aAa | ∈ — {a} {$, a}
A → abS | ∈ — {a} {a}

X (not LL(1))

Q. S → iEtSS' | a — {i} {a}
S' → eS | ∈ — {e} {e}. X
E → b

not LL(1)

✓

29/7
Tuesday

Lecture 3

TDP.

* Recursive descent Parser → writing func^n

$E \rightarrow iE'$
$E' \rightarrow +iE' \mid \epsilon$ } → 1 func^n for every var.

① E()
{
1 if (l == 'i')
{
2 match('i');
3 E'();
}
4 }

look ahead (global var.)

l = getchar();

at every point it takes next char.

② E'()
{
1. if (l == '+')
{
2 match('+');
3 match('i');
4 E'();
}
5 else
6. return;
7. }

Page No.: 41
Date:

③ match(char t)
{
if (l == t)
l = getchar();
else
printf("error").
}

→ checks & increment LA pointer.

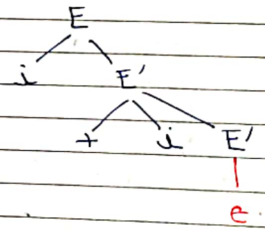
④ main()
{
1 E();
2 if (l == $)
printf("Parsing Success");
}

i/p: i + i $ → LA pointer.

recursion stack provided by OS.

(Parsing Success) → o/p

⑤ ∴ return (all popped).



Refer 24

* Operator Grammar & Operator Precedence Parser:

- can pass Amb. Grammar.
- BUP
- define math operators for compiler.

* Operator Grammar

$E \rightarrow E+E \mid E*E \mid id$

- no 2 var are adjacent
- No ϵ prod.

eg. $E \rightarrow EAE \mid id$ X
$A \rightarrow + \mid *$

→ in C compiler,
ab ≠ a*b

[No 2 variables are side by side].

eg: $S \rightarrow SAS \mid a$ X
$A \rightarrow bSb \mid b$

⇓ (Substitution)

$S \rightarrow SbSbS \mid SbS \mid a$ → Operator Gramm.
$A \rightarrow bSb \mid b$

(using op. relation table, it can pass ambig. grammar)

Q $E \rightarrow E+E \mid E*E \mid id$

Operator relation table (g across top, f down side); table talks abt precedence; operator prec. table

f \ g	id	+	*	$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	–

→ id > + (precedence)

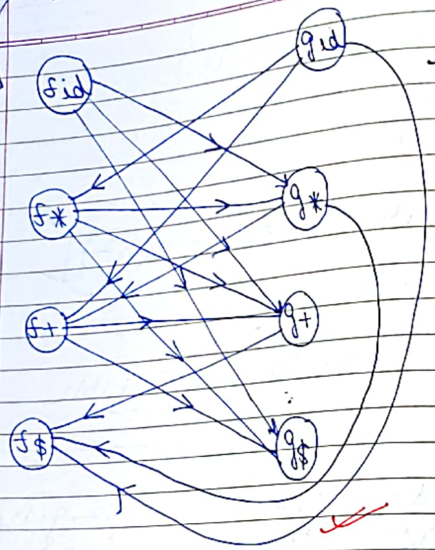
\+ → + is compared with +
\+

+ is LA (+>+)

- $ → least precedence compared to any other op.
- id → highest prec. comp. to other op.

Scanned with CamScanner

Op. precedence



- If any cycle: not possible to construct funct^n table.
- Find longest path starting from node.
  - fid → g* → f+ → g+ → f$
  - gid → f* → g* → f+ → g+ → f$

Op. function

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

↳ representing same info.

RT → $O(n^2)$
FT → $O(2n)$ [VI]

functionality

- f+ > g+ : 2 > 1   + → LA
- f* > g+ : 4 > 1   * > + (Prec.)
- *   * : f*   g* : 4 > 3   ∴ * → LA

- (FT is lesser size than RT).

Drawback [VI]

1. Even though, we have blank entries, we get non blank entries (id & id).
   Error detecting capability
   EDC(FT) < EDC(PT).

SR: Sub of sent

- Use FT always

Q
P → SR | S ①
R → bSR | bS ②
S → WbS | W
W → L * W | L
L → id.

P → Para.
R → Rec. Snt.
S → Sent.
W → Word
L → Letter.

· P → SbSR | SbS | S.

· P → SbP | SbS | S
S → WbS | W (RR)
W → L * W | L (RR)
L → id

} Operator Grammar.

op. relation table.

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

* < *
b < b → RR. ∴ RA

Page No. 49

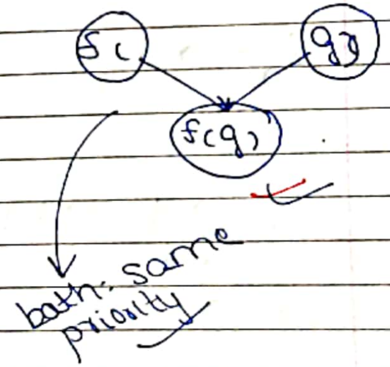
Q Convert op. rel table → op. funct table. (→ op. precedence table)

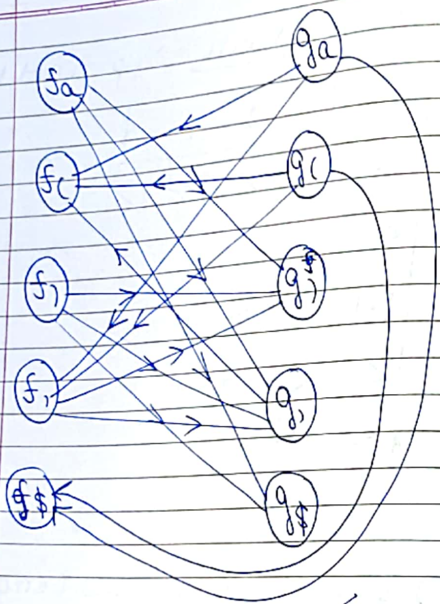
	a	(	)	,	$
a	–	–	>	>	>
(	<	<	(=)	<	
)	[·]		>	>	>
,	<	<	>	>	
$	<	<			

·) & a = not compared (error) (empty cells)

· (&) = eq. priority.

( )
f(g)





$f_a \to g_) \to f_) \to g_)$
$f_($ ~~→~~ .
$f_) \to g_)$

$\Rightarrow f_a \to g_) \to f_) \quad g_a \to$
$f_($
$f_) \to g_) \to f_)$
$f_; \to g_)$
$f_\$ \to$

	a	(	)	;	$
f	2	0	2	2	0
g	3	3	0	1	0

function table

* LR Parsing, LR(0) items and LR(0) Parsing Table

LR Parsers → BUP

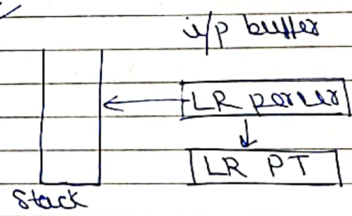
VI

- LR Parsers
  - LR(0)
  - SLR(1) (Simple LR)
  - LALR(1)
  - CLR(1)

LR(0), SLR(1): LR(0) items are used

LALR(1), CLR(1): LR(1) items

→ powerful



(only PT changes)

- To construct all parsers we need PT.
  - LR(0) / LR(1) items.
- LR(0) items used to construct LR(0) PT.

eg: $S \to AA$
$A \to aA \mid b$

closure()
goto()

3 productions.

$S' \to S$ ← Augmented Grammar.
$S \to AA$ ①
$A \to aA \mid b$ ② ③

VI

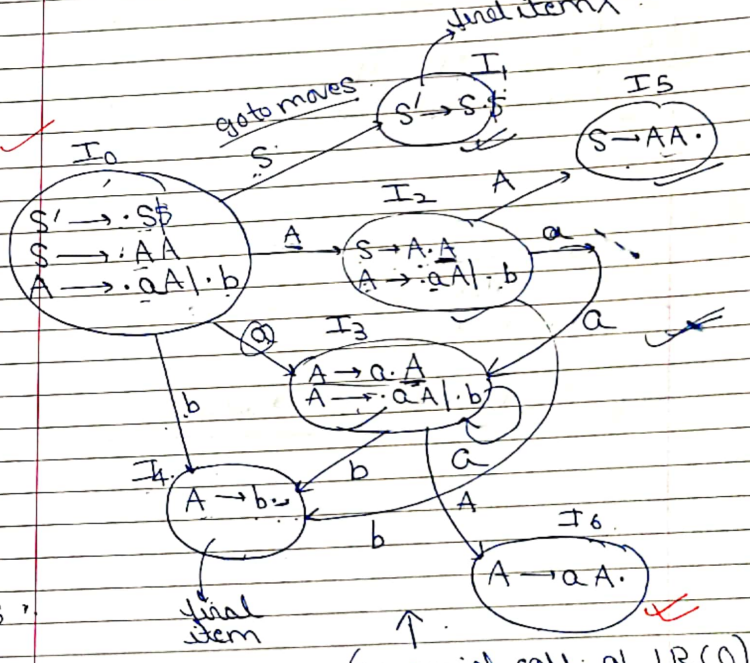
LR(0) items (any prod with ·)

$S \to \cdot AA$ (nothing in RHS).
$S \to A \cdot A$
$S \to AA \cdot$ (seen every in RHS) → final item.

- when to reduce?

$S' \to \cdot S$
closure
$S \to \cdot AA$
$A \to \cdot aA \mid \cdot b$

· → to the left of var.



- goto & closure (canonical coll of LR(0) items)

$A \to b \cdot$ (seen something so reduce).

LR(0) PT

States	action a	action b	action $	Goto A	Goto S
0	S_3	S_4		2	1
1			accept		
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		

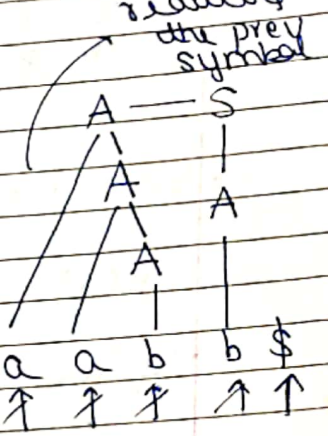
- r: reduce move.
S: shift move. (T)
goto move: (V)

T_1: S' (added by us) → accept.

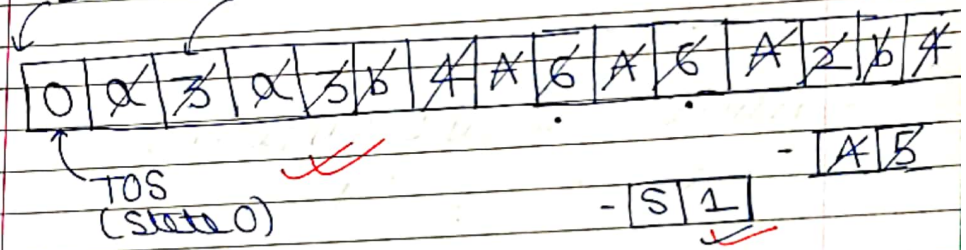
* LR(0) Parsing Table Example & SLR(1) Table:

① $S' \rightarrow S$
② $S \rightarrow AA$
③ $A \rightarrow aA \mid b$

w = a a b b $



w = a a b b $



r_2: A → b. L = |b| = 1.
n → 2n (pop).
push LHS(A) & fill with Sno.

r_3: A → aA.

reduce Prod 3.

Shift: shift.
reduce: reduce
pop. prev symb (double inc)

Symbol is!

- LR(0)
  - 0 Lookaheads.

A
a a b b $

- blank entries: errors.

LR(1)

- if we have pref S, don't put R moves blindly.

SLR(1)

reduce is done, if LHS is followed.

	a	b	$	A	S
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

I_4 = A → b.

follow of A : { a, b, $ }

I_5 → S → AA. {$}

I_6 → A → aA. {a, b, $}

- Shift & go To moves are same in LR(0) & LR(1)

DIH: placing of reduce moves

LR(0): place R in entire row
LR(1): place R in the follow of LHS

VI

eg: S → AA

diff in the LR. PT. (structure same).

S
AA $

(reduce AA to S, when the LA is follow of S: $).

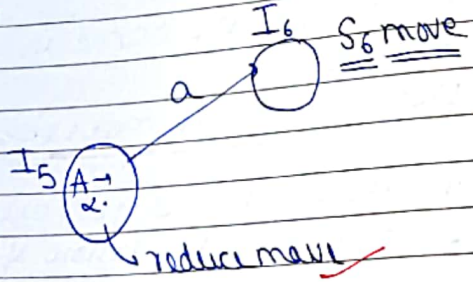
VI

SLR(1) → reduce when the next symbol is in the follow of LHS

- SLR(1) > LR(0)
  - detect max fast.
  - no. of blank moves more

- place function in BM, to continue parsing → error msg.

*

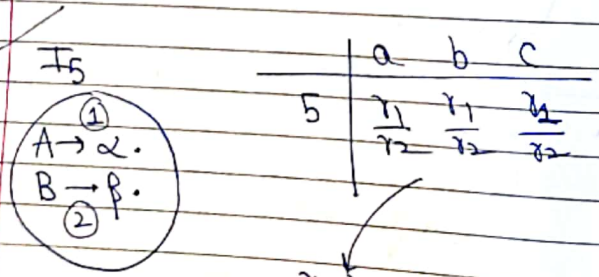


Q Is every gramm. LR(0)? (if rr/rs conflict).

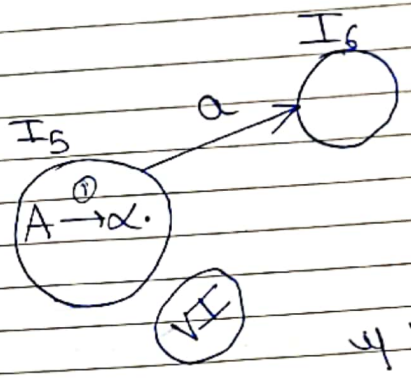
Action

	a	b
5	S_6/r_1	r_1

- Shift reduce conflict
rs conflict



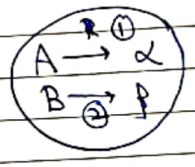
* SLR(1)



	a	b
5	S_6	

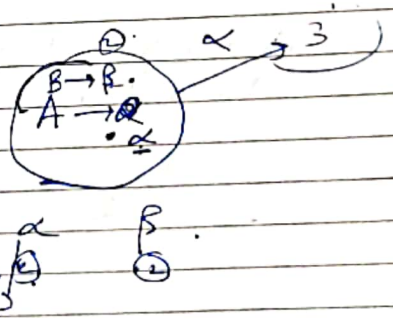
if follow(A) ≠ a
→ NP

I_5



- R_1 will be placed in follow(A)
- R_2 will be placed in follow(B).

follow(A) ∩ follow(B) ≠ ∅. = RR conflict



Tuesday

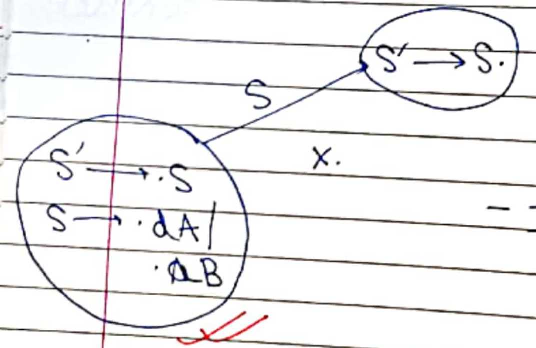
Lecture 4

Q $S \rightarrow dA \mid aB$
$A \rightarrow bA \mid c$
$B \rightarrow bB \mid c$

(i) LL(1)
(ii) LR(0)
(iii) SLR(1)

- SR conflict, RR conflict.

- construct canonical collect^n of LR(0) items.



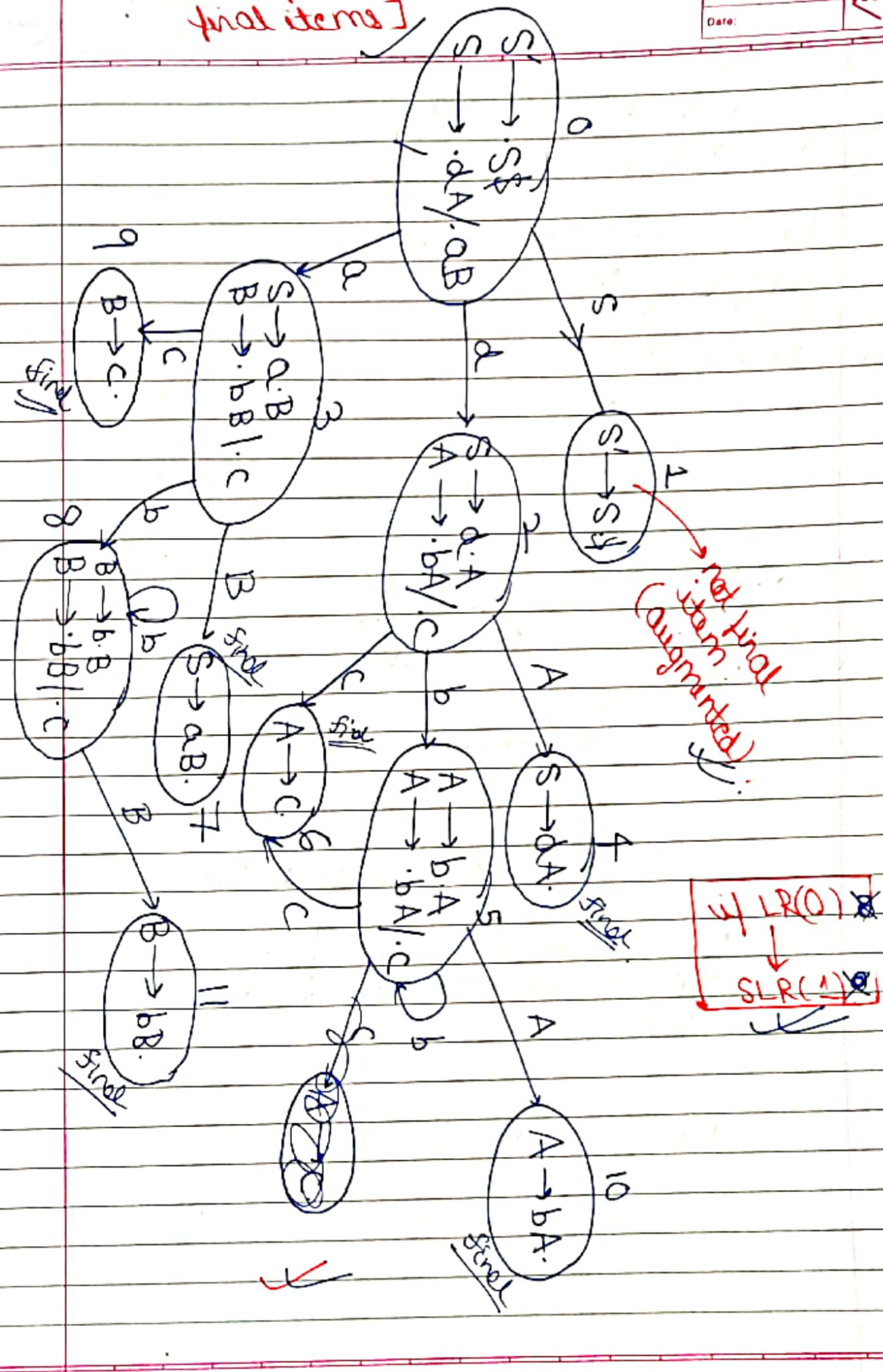
9 → RR, SR X.

4 → No RR, No SR.

6 → No RR.

- In the final state, no conflict.

∴ LR(0), LR(1).



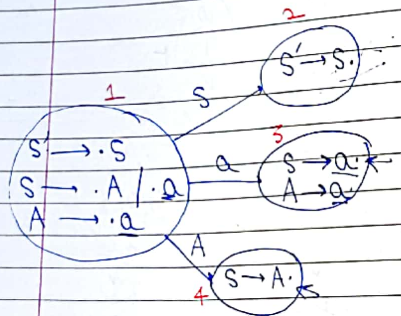
Q. $S \to A \mid a$ {a, a}
$A \to a$

Is Gramm. suitable for?

(i) LL(1) X
(ii) LR(0) X
(iii) SLR(1) X

	a	$
3		R2/R3

RR conflict
Not SLR(1)



∵ 2 reduce moves, LR(0) X

S → a in follow(S) = $
A → a in follow(A) = $
SLR X

can't be parsed by any parser.
∵ ambiguous.

(PC) → PC : Canonical

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

LL(1) X (LR)
LR(0) ✓
SLR(1) ✓

Gramm. is Left recursive ; ∴ LL(1) X
(TDP dont work with LR Gram).

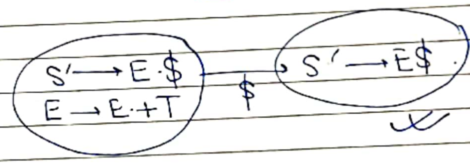
L → S. : FL

S → (L). NO RR / NO SR ✓
L → L, S. No SR / No RR ✓

if LR(0), then LR(1) ✓

Q. $E \to E+T \mid T$ → PC : DFA
$T \to i$ (PC)

S′ → E
E → E+T | T
T → i.



(Left Recursion)

No conflict.

E → LR

LL(1) X
LR(0) ✓
SLR(1) ✓

(PC)

Q. $E \rightarrow T + E \mid T$
$T \rightarrow i$

LL(1) X
(gives the same call (E, i))

$E \rightarrow T \cdot + E$
$E \rightarrow T \cdot$

\+ →

final item & shift move (SR)

LR(0) X

→ LR(0) PT shown.

r_2 → follow(E) = \$
follow(T) = {\$, +}.

SLR(1) ✓

* Examples of LR(0) & SLR(1)

Q. $E \rightarrow E + T \mid T$
$T \rightarrow TF \mid F$
$F \rightarrow F* \mid a \mid b$

→ Set of all REs.

LL(1) X

→ is it LL(1)? No → Left Recursion

→ is it LR(0)?

I_1: Augmented X

I_2: $E \rightarrow T \cdot$ → reduce

a, b → shift.

(SR conflict) → where SR conflict in LR(0)

∴ **LR(0) X**

2 → inadequate state.

SLR(1)?

Action

	a	b	+	*	\$
2	S_4	S_5	r_2		r_2
3	r_4	r_4	r_4	S_8	r_4
7	r_3	r_3	r_3	S_8	r_3
9	S_4	S_5	r_1		r_1

follow(E) = {+, \$}   follow(T) = {+, \$, a, b}

∴ 4 inadequate states have SR conflict in LR(0)

SLR(1) ✓

Q. $S \to AaAb \mid BbBa$ → (PC) $\{ab, ba\}$
$A \to \epsilon$
$B \to \epsilon$

LL(1) ? ✓
LR(0) ? X
SLR(1) ? X

LL(1) ✓ **VI**

$A \to .\epsilon$ → $A \to .$
$A \to \epsilon.$

LR(0) X → 2 reduce moves in same state.

	a	b	$
0	r_3/r_4	r_3/r_4	

→ rr conflict

SLR(1) X

Q. $S \to AS \mid b$ $\{a, b, b\}$
$A \to SA \mid a$

LL(1) ?
LR(0) ?
SLR(1) ?

→ No Parser can Pass it. (Ambig.)

LL(1) X

$S' \to S.$ → accepted item.

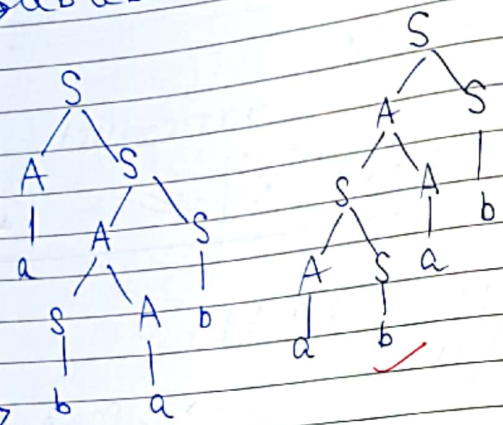
– not entire diag.

LR(0) X → SR conflict.

Follow(A) = {a, b} (SR complete)

SLR(1) X

⇒ abab.



VIII

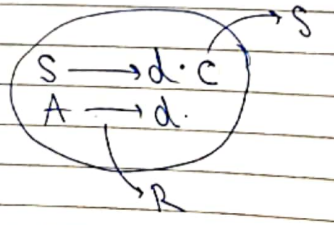
→ Construct diag. till the state with conflicts. See if SLR can resolve it or not.

{a} {b} {d}

Q. S → Aa | bAC | dC | bdAa
A → d

LL(1) X

S' → S
S → Aa | bAC | dC | bdAa
A → d.



S' → .S$
S → .Aa | .bAc | .dc |
.bdAa.
A → .d.

LR(0) X

Shift in {c}

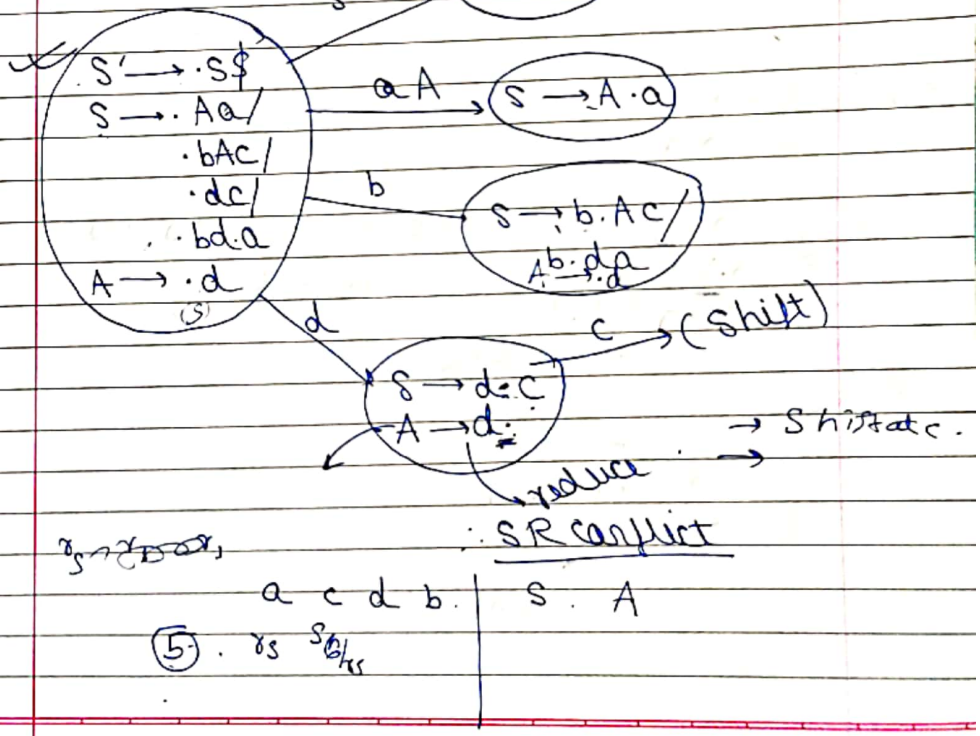
Follow(A) = {a, c}

	a	b	c	d	$
1	r		S r		

SLR(1) X

SLR1

(not F state)
accepted State.
X



12/4/19

CLR(1) AND LALR(1) PARSERS

- Constr^n of PT only varies.

→ Canonical coll of LR(1) items are used.

(LR(0) item + look Ahead).

eg: S ⟶ ·aA , a/b

S → aA·, c/d
(place reduce moves in c, d.)

---

eg: S ⟶ AA
A ⟶ aA | b

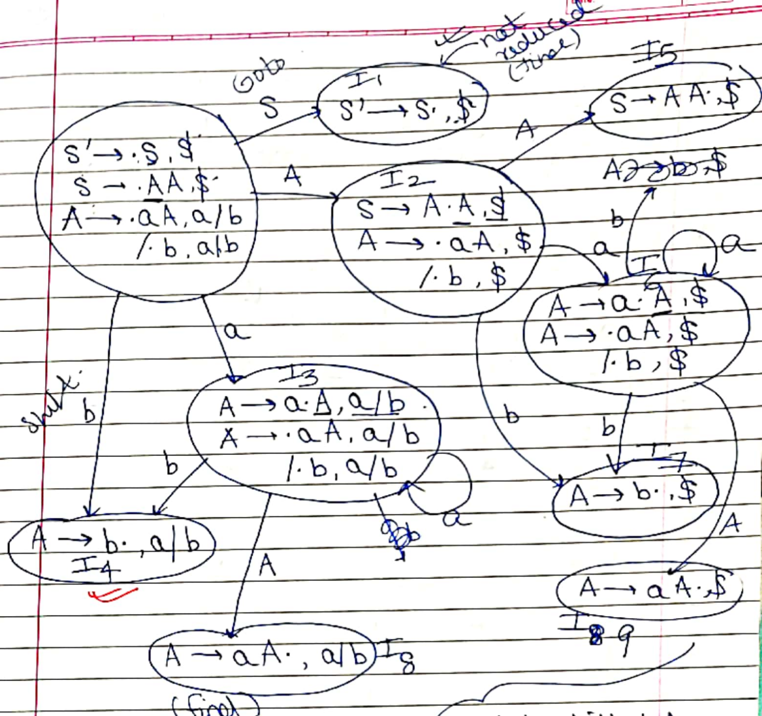
S′ ⟶ ·S , $ ← LA = $ for augmented prod^n
S ⟶ ·AA , $ ①
A ⟶ ·aA | ·b ② a,b ③ a,b

rem: A$ ∴ f(A,$)

LR(1) items

if A ⟶ α·Bβ , a/b
B ⟶ ·γ , c/d
(first of β)

- while applying ...



2 states diff. LA.
- in terms of LR(0) items, these states are same.

→ ∴ No. of states in LR(1) items more.

· In CLR(1), reduce moves are only present in look Aheads.
→ less no. of conflicts. (decrease no. of reduce moves).
- blank space ↑, error detecting capability ↑.

I_3, I_6 : same LR(0), diff. LA
I_4, I_7 : same LR(0), diff LA
I_8, I_9 : same LR(0) item, diff. LA

* CLR(1) Passing Table

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
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		

$I_3, I_6 \Rightarrow I_{36}$
$I_4, I_7 \Rightarrow I_{47}$
$I_8, I_9 \Rightarrow I_{89}$

by merging,

10 states → by LR(0) PT: 7 states

→ Because of LA states ↑.
(Size of CLR ↑)

No. of States in LR(0) = SLR(1)
∧
No. of States in CLR(1)

No of States (LALR(1)) = LR(0) = SLR(1)

	a	b	$	S	A
0	S_{36}	S_{47}			2
1	~~S_{36}~~				
2	S_{36}	S_{47}			5
3	S_{36}	S_{47}			89
4	r_3	r_3			
5			r_1		
6	S_{36}	S_{47}			89
7			r_3		
8	r_2	r_2			
9			r_2		

LALR(1) PT.

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

→ 7 states

* Conclusion & Example of CLR(1) and LALR(1)

LR(0) item

$A \to \alpha \cdot R$

$B \to \cdot Q$

$\in S$

LR(1) item

$Q \cdot$

$A \to \alpha \cdot \alpha\beta \; c/d$

$B \to \gamma \cdot , Q / \$$

SR conflict

$A \to \alpha \cdot , a$

$B \to \beta \cdot , a$

common LA

RR conflict.

∴ Gramm is not CLR(1)

↓

LALR(1)

If CLR(1), may not be LALR(1).

CLR(1)

$A \to \alpha \cdot , b$

$B \to \beta \cdot , a$

Merge

$A \to \alpha \cdot , a/b$

$B \to \beta \cdot , a/b$

LALR(1)

RR conflict.

If No SR conflict in CLR(1), no SR in LALR(1).

$S \to Aa \; Ab \mid Bb \; Ba$

$A \to \in$

$B \to \in$

LL(1) ✓

LR(0) X

S LR(1) X

$S' \to \cdot S$

$S \to \cdot AaAb$

$/ \cdot BbBa$

$A \to \cdot$

$B \to \cdot$

$\{a, b\}$

$\{a, b\}$

Scanned with CamScanner

Canonical coll. of LR(1) items

∴ CLR(1) ✓

→ Hai, CLR(1) = LALR(1)
(No merging)

LALR(1) ✗

(All Points with same no. of items)

Q. S → Aa /bAc /dc /bda /A → d. (next page)

LL(1), LR(0), SLR(1) X

CLR(1) ✓

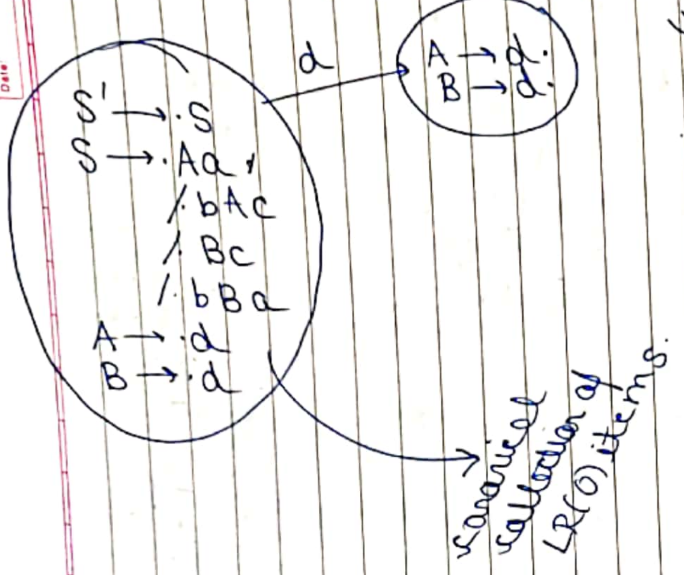
∴ No merging

LALR(1) ✓

Q. S → Aa /bAc /BC /bBa /A → d /B → d. (Pg 74)

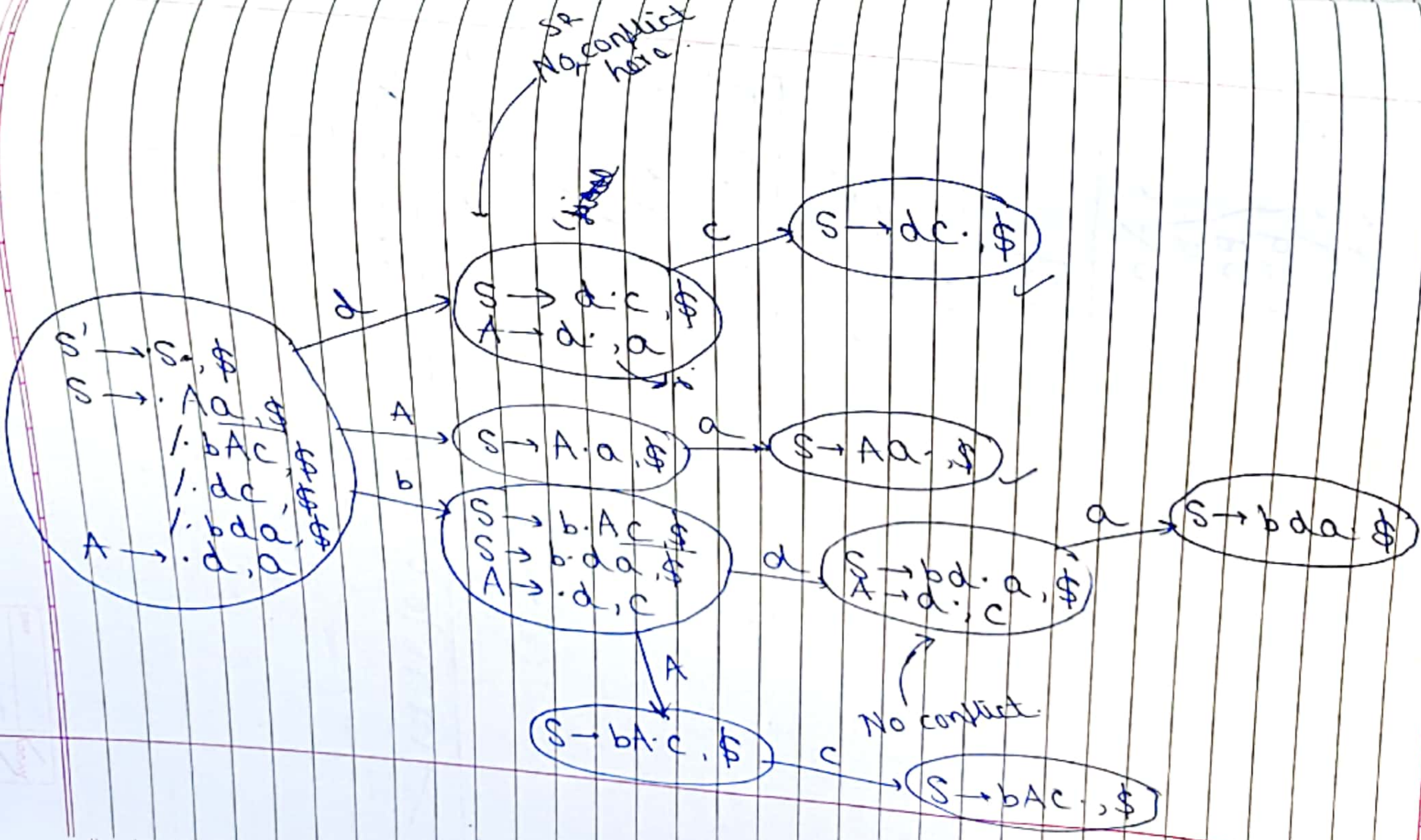
LL(1) X

∴ RR conflict LR(0) X



fol(A) = a, c
fol(B) = a, c

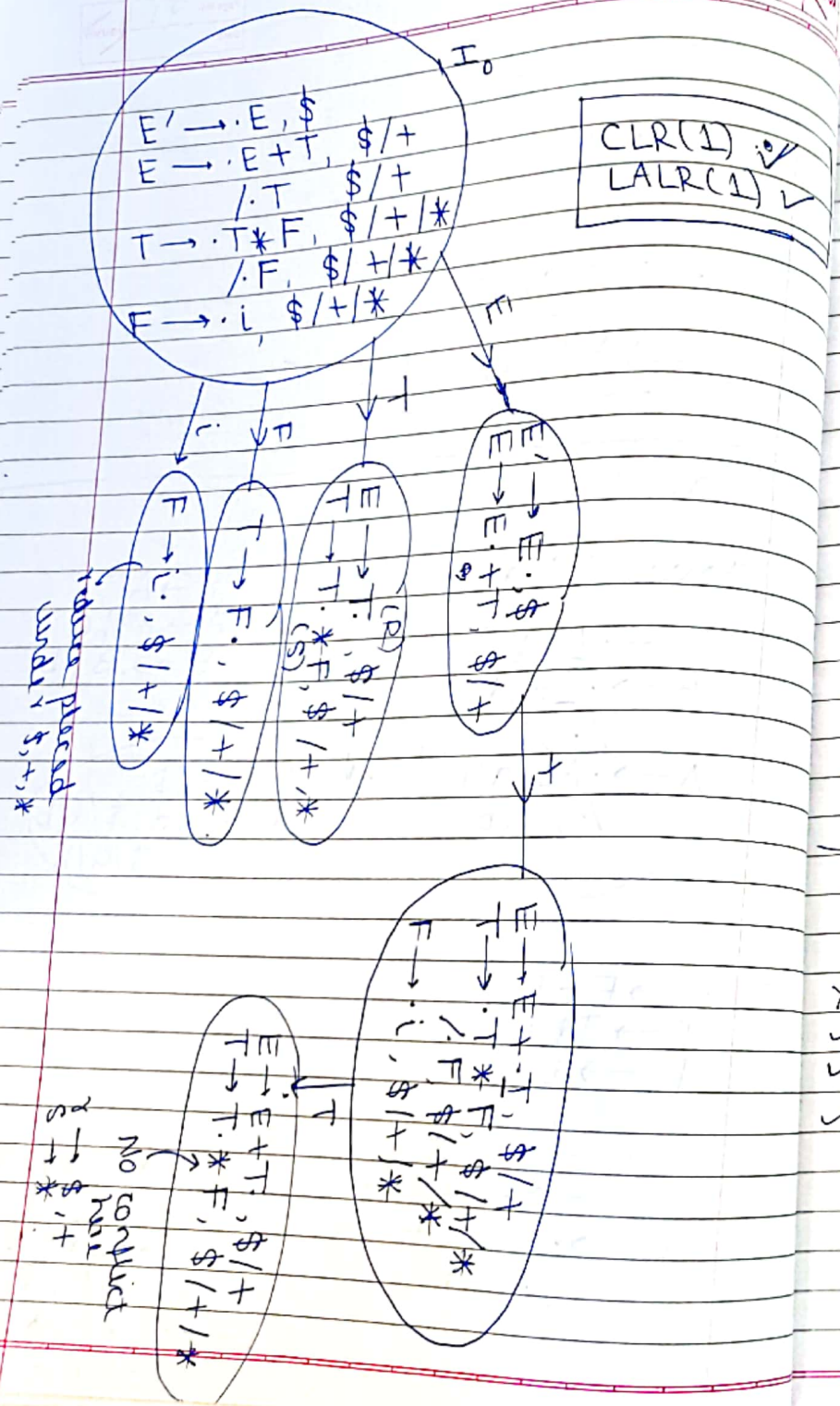
∴ SLR(1) X



No R/R Conflict

No conflict here.

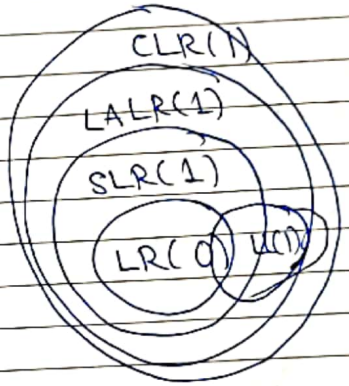
Scanned with CamScanner



eg: If G = amb.;

operator precd. parser can pow it.

LR(0) & LL(1) least powerful



SLR → more powerful than LR(0)

TS ↑ CLR(1).

if LR(0), it is SLR(1), LALR(1) & CLR(1).

(every LL(1) Gramm., is LALR(1))

Q. Consider SLR(1) & LALR(1) table for a CFG:

✗ (a) Goto of both tables may be different

✓ (b) Shift entries are identical

✓ (c) Reduce entries in table may be different

✓ (d) error entries in table may be diff.

No. of States & transitions exactly identical

→ change in Blank entries.

no. of states

n_1 = SLR(1), n_2 = LALR(1),

n_3 = CLR(1) → same LR(0), but diff LA.

∴ Relation:

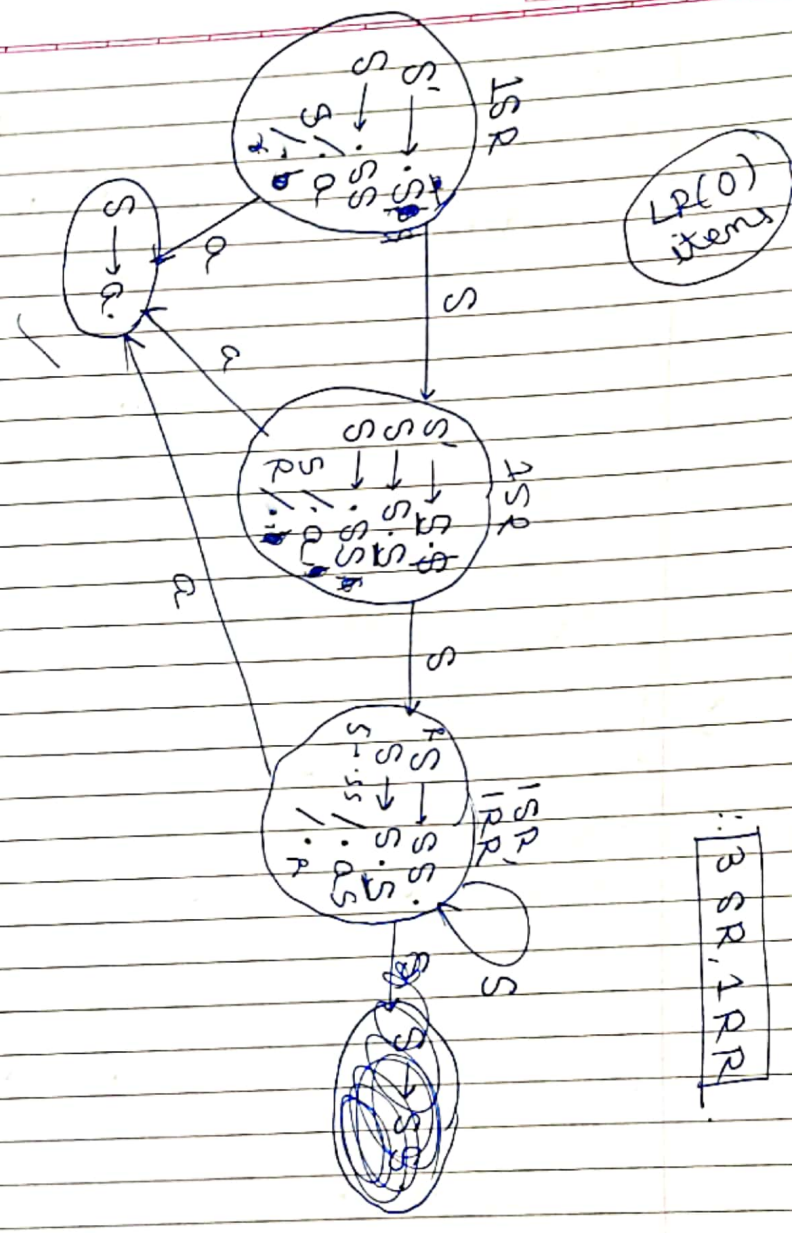
$$n_1 = n_2 \leq n_3$$

Q $S \rightarrow CC$
$C \rightarrow cC / d$ → clearly LL(1)
(c) (d)

(a) LL(1) ✓
(b) SLR(1) but not LL(1)
(c) LALR(1) but not SLR(1)
(d) CLR(1) but not LALR(1). ✓

Q Find no. of SR & RR conflicts in dfa with LR(0) items.

$S \rightarrow SS$
$/ a$
$/ \epsilon$



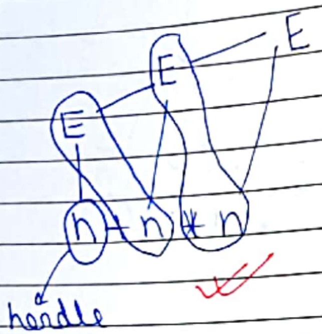
right seq.

Q (call)

n+n*n → String

E → E*n
⇒ E+n*n
⇒ n+n*n

→ 3 handles



(d).

n, E+n, E*n
handles

Q $S \to (S) / a$

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

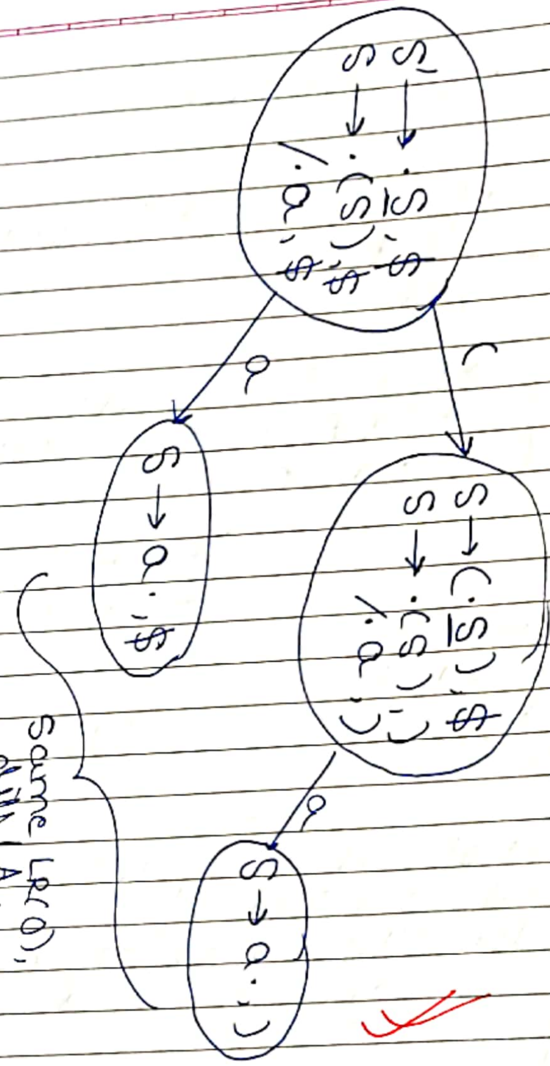
Sol^n

$n_1 = n_3$ directly (for any gramm.)

$n_1 = n_3 \leq n_2$

(i) 2 states with same LR(0) items but diff LA)

VI



∴ $n_2 > n_1 = n_3$

(b).

→ More conflicts, if we use AG.

$E \to E \cdot + E$
$E \to E + E \cdot$ } Conflict.

→ id + id + id.

$E \to E \cdot + E$
$E \to E * E \cdot$

→ E.
i/p = id * id + id
reduce / shift.
∴ reduce if, * = more prec.

eg:

$E \to E + E \cdot$
$E \to E \cdot * E$
(SR Conflict)

→ id + id * id

if reduce : + > *.
if shift : * > +.
∴ go for shift

∴ (we can go for Ambig. if we can resolve conflicts meaningfully.

YACC → Yet another comp. comp.
- generates parser (s/w)
- produces LALR(1) PT.

If, SR conflict → Shift
RR conflict → first reduce

r_1/r_2
- reduced first.

SYNTAX DIRECTED TRANSLATION

- Examples

Grammer + Semantic rules = SDT

- Parser can do now, if we give rules

eg: 2+3*4 = 14.

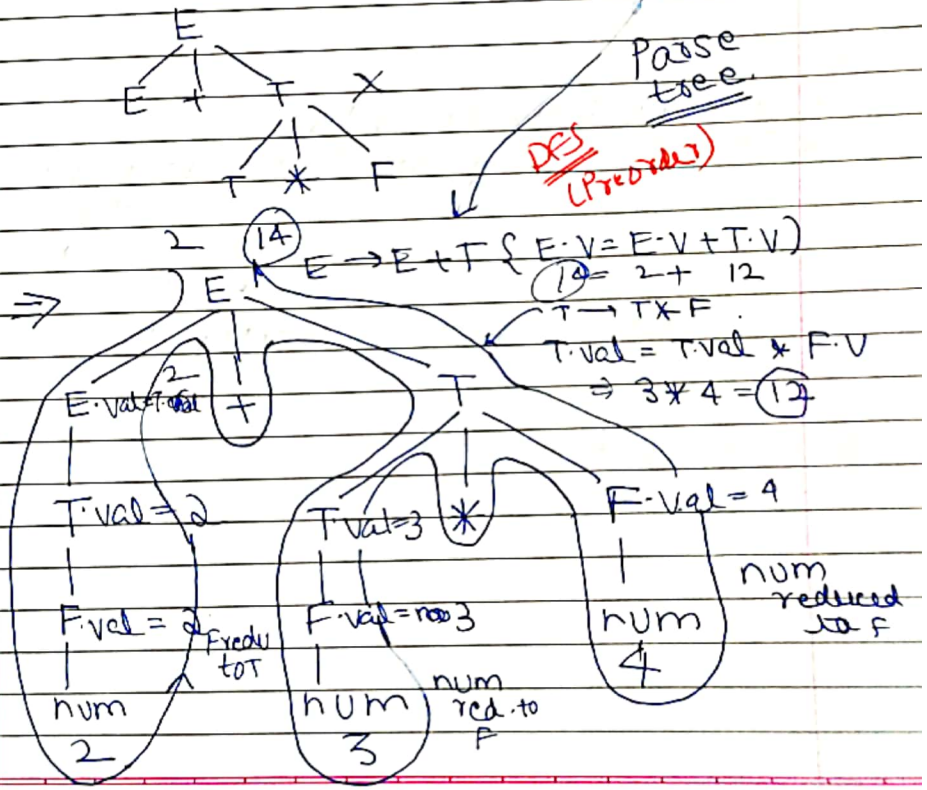
SDT

Semantic rules.

$E \to E+T$	$\{E.value = E.value + T.value\}$
$/T$	$\{E.value = T.value\}$
$T \to T*F$	$\{T.V = T.V * F.V\}$
$/F$	$\{T.V = F.V\}$
$F \to num$	$\{F.val = num.tval\}$

attribute.

⇒ 2+3*4



- After gn. PT, traverse top down left-right reduce.

- SDT

every var → attributes. ← semantic action

E → E + T		{ printf("+"); } ①
/ T		{ } ②
T → T * F		{ printf("*"); } ③
/ F		{ } ④
F → num		{ printf(num.val); } ⑤

→ convert infix to postfix exp.
→ SDT is consid att along with parse

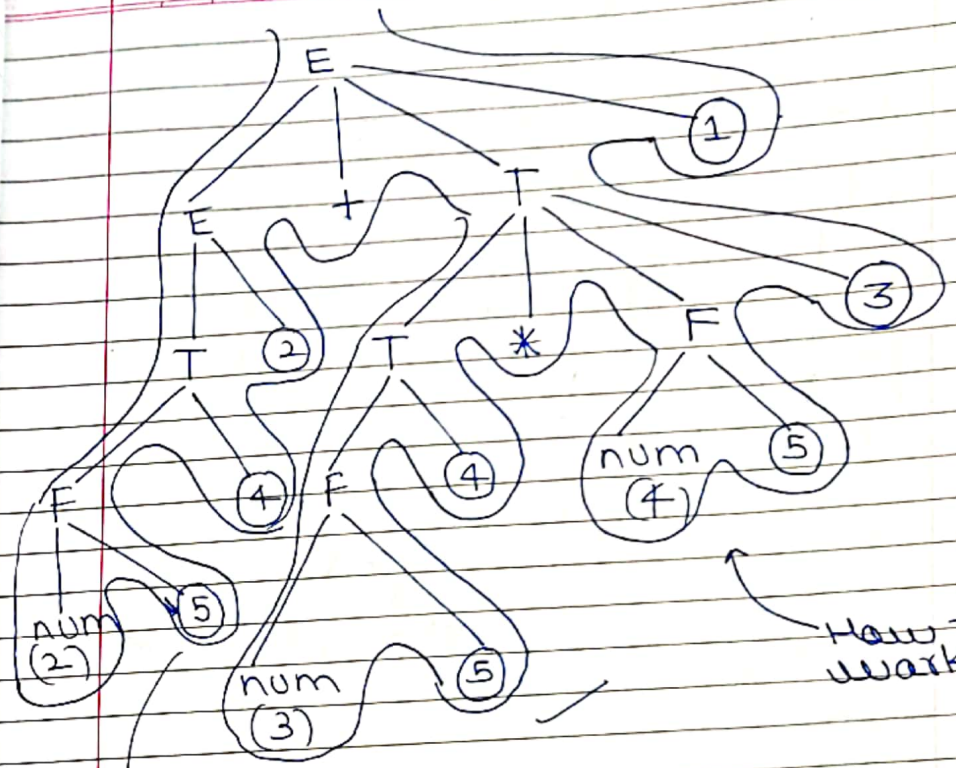
TDP ↓ BUP ↑

when reduction → action.

eg: 2+3*4

2+3*4

TDP construct this



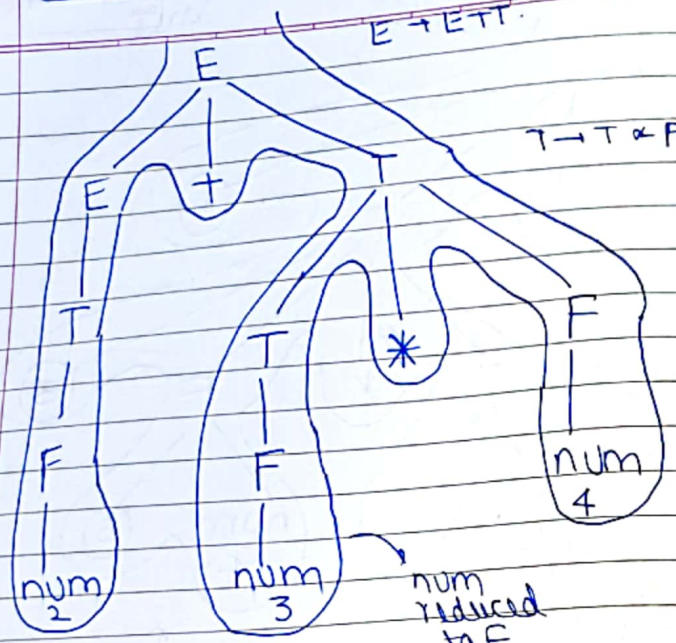
when com across Semantic actn → action

Postfix exp.

∴ 234*+

* How BUP work?

→ BUP construct tree



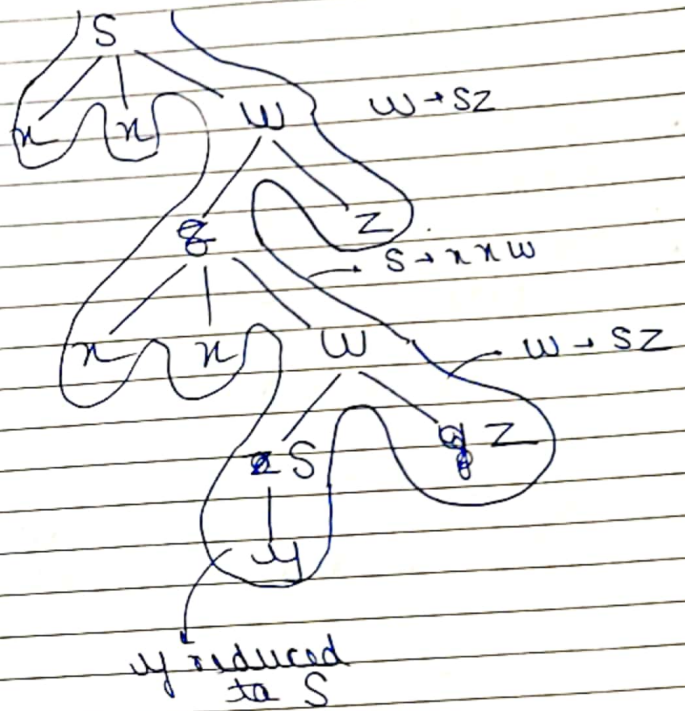
234 * +

↳ Postfix Exp.

Q
S → x x W { printf(1); }
/ y { printf(2); }
W → S z { printf(3); }

String: x x x x y z z.

BUP (LR)



2 3 1 3 1 → O/P printed.

Q
E → E * T { E.val = E.val * T.val }
/ T { E.val = T.val }
T → F − T { T.V = F.V − T.V }
/ F { T.v = F.v }
F → 2 { F.V = 2 }
/ 4 { F.V = 4 }

↑ here lexim directly use.

$w = 4 - 2 - 4 * 2$

- $* \rightarrow LA$
  $- \rightarrow RA$
- $* > -$ (prec.)

} from Gramm.

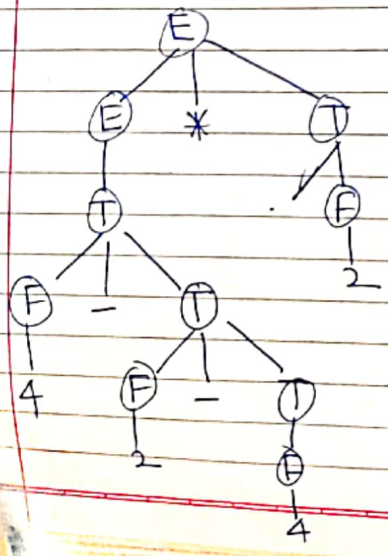
(no need to const PT).

$w = ((4 - (2 - 4)) * 2)$ (Short Cut)

$\Rightarrow (4 - (-2)) * 2$

$\Rightarrow$ 12 Ans.

* How BUP work?



every non-leaf = reduct

10

Q $E \rightarrow E \# T$ (all)

String: 2 # 3 & 5 # 6 & 4

2 * 3 + 5 * 6 + 4

(*) # $\rightarrow$ LA

(+) & $\rightarrow$ LA

& > (prec.) #

(+) > *

} 3 rules directly from Grammar

$\Rightarrow$ 2 * (3 + 5) * (6 + 4)

((2 * 8) * 10)

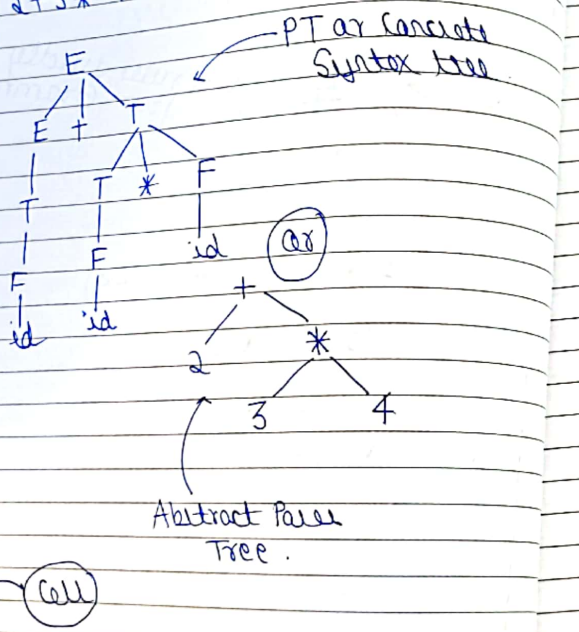
$\Rightarrow$ 160

(or construct BUP).

* EXAMPLES OF SDT

Call

Q: 2+3*4



Soln

Q — Call

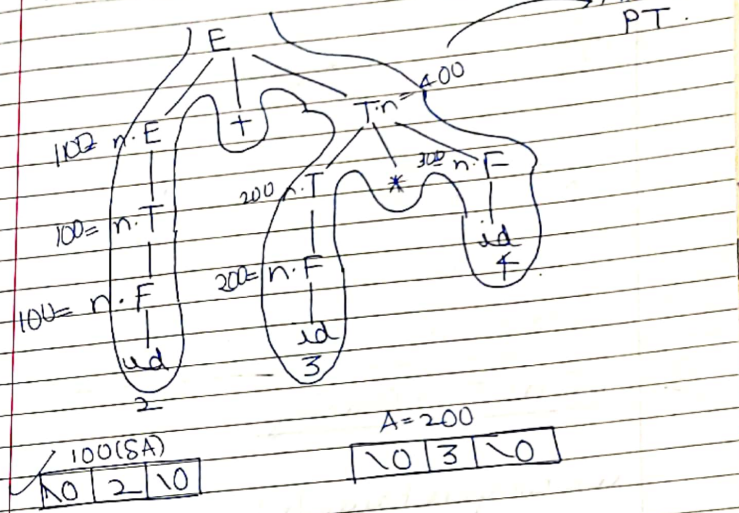
- Gramm spec mul/addn

F.npt = mknode(null, idname, null);

↳ funcn

mknode creates node

2+3*4



100(SA)

\0	2	\0

A=200

\0	3	\0

300

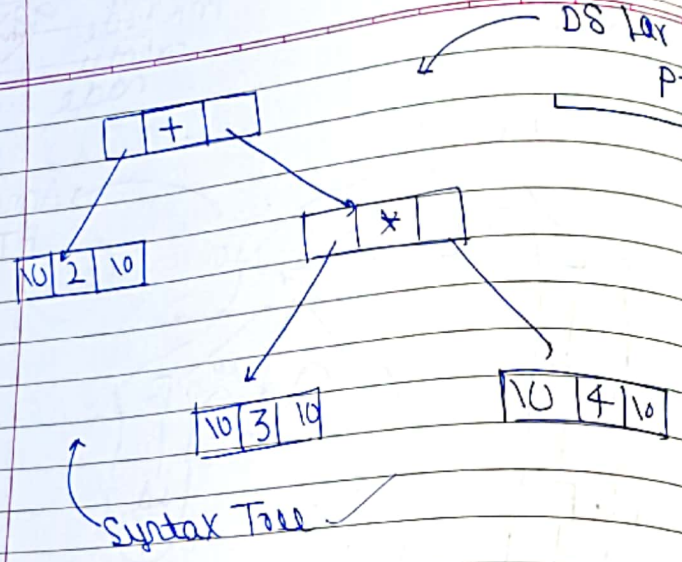
\0	4	\0

400

200	*	300

100	+	400

DS for Abstract PT.



Syntax Tree

∴ SDT to create Abstract Syntax tree.

SDT for Type Checking

- we compare boolean/int.

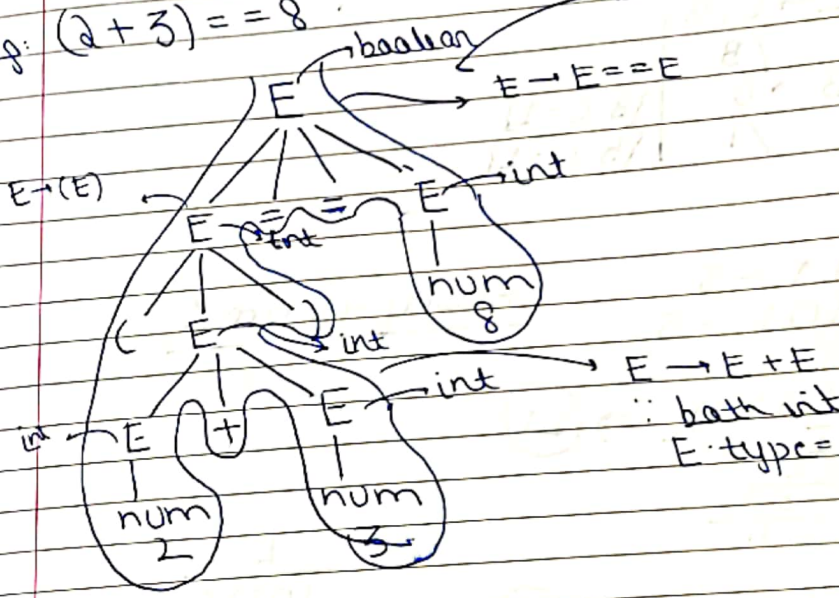
$E \to E_1 + E_2$ { if (E_1==L) call }

Soln

$E_1 + E_2$ → if { (E_1.type = E_2.type) && (E_1.type = int) then E.type = int else error.

is it meaningful according to semantics?

eg: (2+3)==8.



∴ Type of exp = boolean (overall)

Translation on Binary no.

	count 1s	count 0s
N → L	{N.C = L.C}	{N.C = L.C}
L → L.B	{L.C = L_1.C + B.C}	{L.C = L_1.C + B.C}
/ B	{L.C = B.C}	L.C = B.C
B → 0	{B.C = 0}	B.C = 1
/ 1	{B.C = 1}	B.C = 0

No. can be list.
list can be list of Bits / bits

Count All bits

$N \to L$	"
$L \to L, B$	"
$/B$	"
$B \to 0$	{B.C = 1}
$/1$	{B.C = 1}

Binary → Decimal

11 - 3
101 - 5

- $(1011)_2 \Rightarrow 11$

1 → 2 → 5 → 11

$(10110)_2 = 22$

1 → 2 → 5 → 11 → 22

L

10110

L_1 B

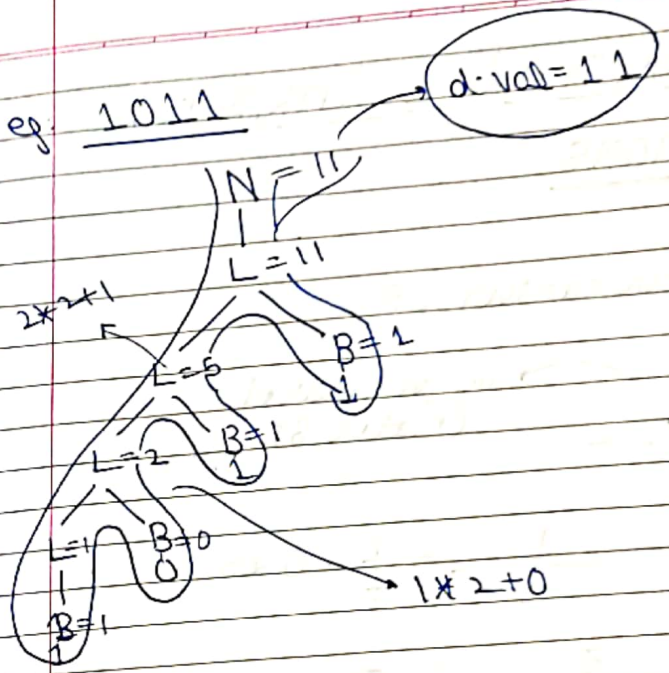
$L = L_1 * 2 + B$

B → D

Soln ⇒

$N \to L$	{N.dval = L.dval}
$L \to L, B$	{L.dval = L_1.dval * 2 + B.dval}
$/B$	{L.dval = B.dval}
$B \to 0$	{B.dval = 0}
$/1$	{B.dval = 1}

eg: 1011



→ What if radix point?

11.01

3.25

- We need to change Gramm. & Semantic Rules.

* S Attributed & L Attributed Definitions

→ No. can ever contain DP.

eg: 11.01 → in gn. method (not in SDT)

$2^1\ 2^0\ 2^{-1}\ 2^{-2}$

⇒ .01 $\quad \frac{1}{2^2} = \frac{1}{4} = 0.25$

⇒ .11 $= \frac{3}{2^2} = 0.75$

assume nRP

no of bits after.

Q

Gramm → SDT

Sol^n

$N \rightarrow L_1 . L_2$	{ N.dval = L_1.dval + $\frac{L_2.dval}{2^{L_2.count}}$ }
$L \rightarrow L_1 B$	{ L.C = L_1.C + B.C; L.dval = L_1.dval + B.dval }
/B	{ L.C = B.C }; L.dval = B.dval }
$B \rightarrow 0$	{ B.C = 1, B.dval = 0 }
/1	{ B.C = 1, B.dval = 1 }

$L \rightarrow L_1 B$
(bit shifted by a bit)

eg: 11.01
$3 + \frac{1}{2^2} = 3.25$

Q SDT to generate 3 address code

Call

x = a + b * c → 4 Address.

S = Statement
E = Exp
T = Term
F = Factor

place → placeholder.

* S ATTRIBUTED & L ATTRIBUTED DEFINITIONS

→ No. can ever contain DP.

eg: 11.01 → in gn. method (not in SDT)

$2^1\ 2^0\ 2^{-1}\ 2^{-2}$

⇒ .01 $\frac{1}{2^2} = \frac{1}{4} = 0.25$

⇒ .11 = $\frac{3}{2^2} = 0.75$

assume no RP

no of bits after.

Q

Gramm → SDT

Sol^n

$N \rightarrow L_1 . L_2$	{ N.dval = L_1.dval + $\frac{L_2.dval}{2^{L_2.count}}$ }
$L \rightarrow LB$	{ L.C = L_1.C + B.C ; L.dval = L_1.dval + B.dval }
/B	{ L.C = B.C ; L.dval = B.dval }
$B \rightarrow 0$	{ B.C = 1 , B.dval = 0 }
/1	{ B.C = 1 , B.dval = 1 }

$L \rightarrow L_1 B$
(bit followed by a bit)

eg: 11.01 → $3 + \frac{1}{2^2} = 3.25$

Q SDT to generate 3 address code

Call

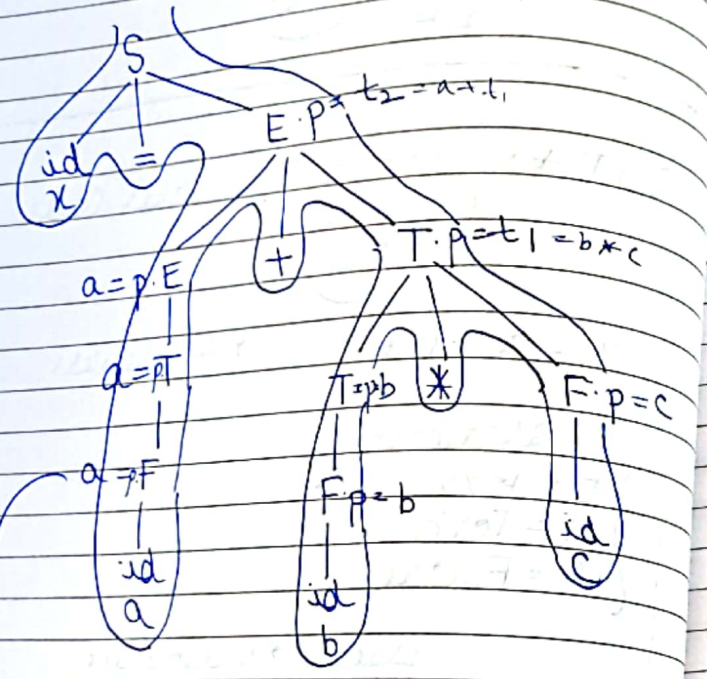
x = a + b * c → 4 Address

S = Statement
E = Exp
T = Term
F = Factor

place → placeholder.

$x = a + b * c$

Parse Tree (BUP)



Q (

Q

F will ren that it has ren a.

Sol^n

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

} every line max of 3 Add

3 Add. code.

* every var. → attribute

eg: $A \rightarrow BCD$

$A.S = f(B.S, C.S, D.S)$

A takes value from children.
∴ Synthesized attribute

ii. $C.i = A.i$ / $C.i = B.i$

child takes value from parent/sibling.
∴ ~~Inherited~~ Attribute Inherited

} 2 type

S-attributed SDT	L-attributed SDT
1) uses only synthesis attributes [less powerful]	1) use both inh. & syn attributes. Each inh. attribute is restricted to inherit either from parent or left siblings only. eg: $A \rightarrow XYZ$ { $Y.S = A.S$, ✓ $Y.S = X.S$, ✓ $Y.S = Z.S$ } X $A.S = X.S$ } ✓. [more powerful]

2) Semantic actions are placed at right end of prodⁿ.

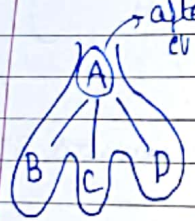
$A \rightarrow BCC\{\}$

postfix SDT

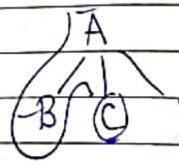
Semantic actions can be placed anywhere

$A \rightarrow \{\}BC$
$/ D\{\}E$

3) Attributes are evaluated during BUP.



4) Attributes are evaluated by traversing depth first, left to right



eg: Q → call

Soln

$A \rightarrow LM$
$A \rightarrow QR$

$L.i = f(A.i)$

I (Not S)

X (Not L)

$Q.i = f(R.i)$

$A \rightarrow BCD$

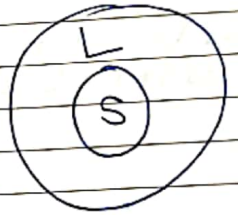
C can't take value from D → not L.

∴ (d) None

(c) $A \rightarrow BC \{B.S = A.S\}$

children take from parent (inheritance)

∴ L-attribute (b)



If SDT is SAT.
↳ only Synthesized Attr.
∴ L attributed.

* SDT TO ADD TYPE INFORMATION INTO SYMBOL TABLE

Q → call

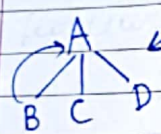
Q (

int x, y, z.

o/p

x	int
y	int
z	int

→ S attr. ~ parents take from children. (synthesis)



→ L attribute : A → BCD

C takes from A, B.
B takes from D X.

Q

D → TL (L).
T → int (S)
char
L → L, (L).

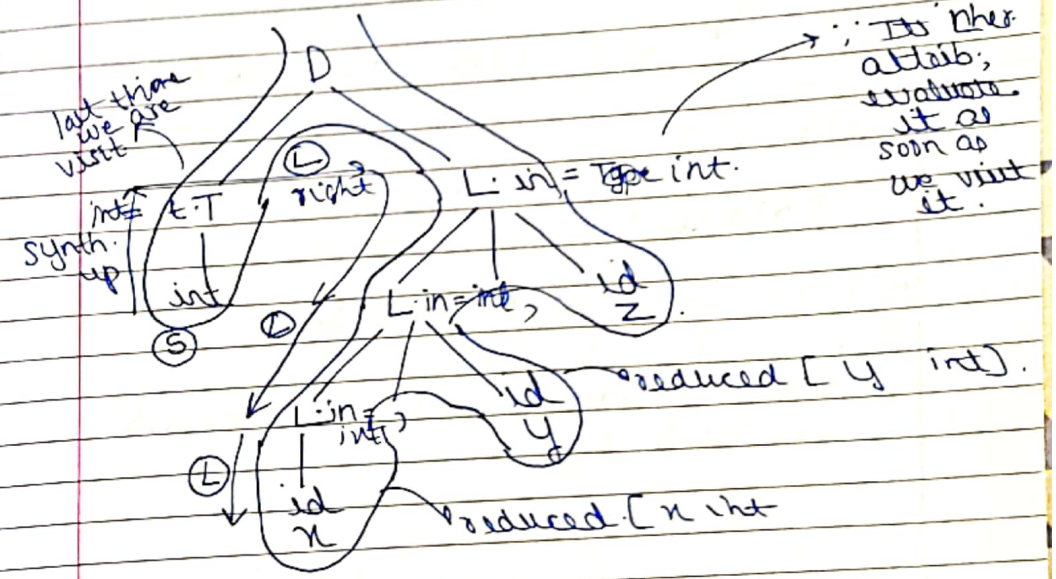
Sol^n ⇒ This SDT is L-attributed.

D → TL.

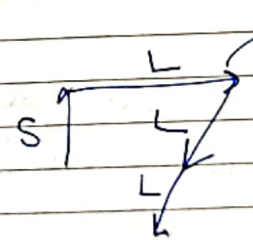
= Declaration is type followed by list of attributes.

string.

int x, y, z;



x	int
y	int
z	int



⇒ Y info flows only up → S attributed.

Q (

To convert L-att → S-att.

(change Gramm. + Semantic rules.)

Q Next SDT

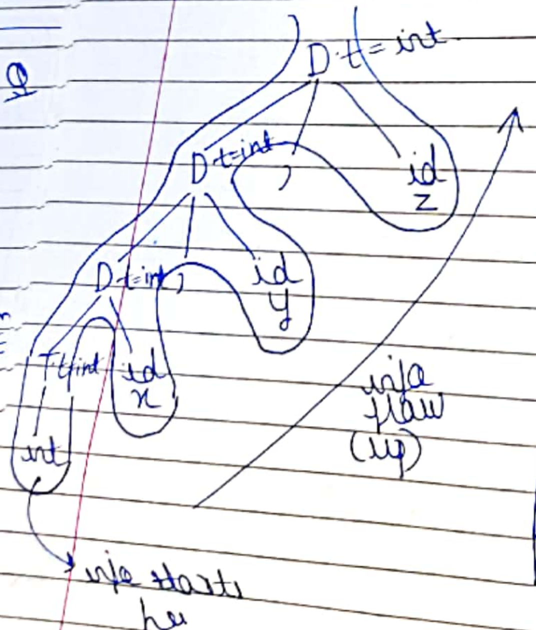
(S attributed) → (Cell).

- all are Synthesized attribute.
- Gramm changed.
- Info starts from lowest level & should move level up.

int x, y, z

Q

Sol^n



Symbol Table updated

x	int
y	int
z	int

∴ Only Synthesized attributes needed.

* INTERMEDIATE CODE GENERATION

- Having intermediate code is imp.
  - till same phase, last phase can take IC & gen MC.
  - now we want to design compiler for other m/c.

[ICG is same for all m/cs]

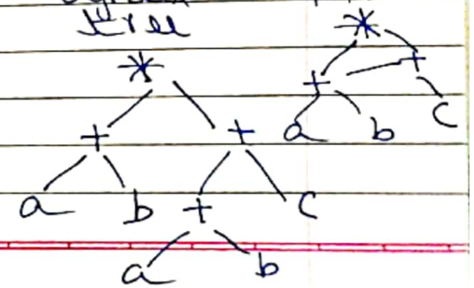
- only last 2 phases will be changed (BE).

eg: Byte Code → IC

(can be executed in any m/c.)

(a+b) x (a+b+c)

Intermediate Code
- Linear Form
  - Postfix: ab + ab + c + *
  - (*) 3 add code:
    - $t_1 = a + b$
    - $t_2 = a + b$
    - $t_3 = t_2 + c$
    - $t_4 = t_1 * t_3$
- Tree Form
  - Syntax tree
  - DAG



eg: (a+b) * (a+b+c)

Q ((DAG) avoids repitition

⇒ Most Popular : 3 Address Code

* Types of 3 Add. Code

1) x = y op z (binary)
2) x = op z (unary)
3) x = y
4) If x (relop) y, goto L (cond^n goto)
5) goto L (uncond^n goto)
6) A[i] = x (Array index)
   y = A[i]
7) x = *p (pointer)
   y = &p x

Q

Sol^n

Any HL → converted to

∴ Intermediate code can run on any m/c.

call

* Various representations of 3 Address Code

⇒ (a+b) * (c+d) + (a+b+c)

1)
2)
.
.
7)

3 add. code → how to be stored in comp. (following some format) [rep. format]

* Quadruples

opr	op 1	op 2	result

→ Statements can move around ∵ can be done in pipeline.
- too much space wasted.

* Triple

opr	op 1	op2
+	a	b

→ can store values in temp registers
→ we don't need temp variables
- no space wasted
- Statements can't be moved around.

* Indirect triples

Q - dont put init in prog, put init in Memory & use pointers.

- ∴ requires 2 memory access → drird
- statements can be moved around.
  1. address
  2. init.

→ all are popular & all are being used.

* BACK PATCHING & CONVERSATIONS TO 3 ADDRESS CODE

Q

if(a<b) then t=1 → not present in 3 Add. code
    else t=0.

Sol: Leaving the labels as empty & filling them later is called backpatching

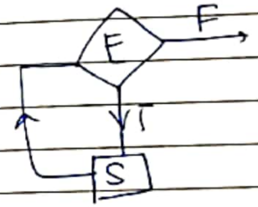
(i) if a<b goto i+3.
(i+1) t=0
(i+2) goto i+4
(i+3) t=1
(i+4)

t_1 t_2 t_3 → not 3 add. code

a<b and c<d or e<f

100) if (a<b) go to 103
101) $t_1=0$
102) goto 104
103) $t_1=1$
104) if (c<d) goto 107   and > or
105) $t_2=0$
106) goto 108
107) $t_2=1$
108) if (e<f) go to 111
109) $t_3=0$
110) goto 112
111) $t_3=1$ and > or
112) $t_4=t_1$ and t_2
113) $t_5 = t_4$ or t_3

* While E do S → gets converted to 3 address code



many ways to implement

L: If (E==0) goto L1
    S
    Goto L
L1:

(or)

(While loop

fill up the Blanks

L: If (E) goto L1
goto Lext
L1: S
goto L
Lext:

→ identifying loops in 3 Addr code is difficult ∵ code optim. req that

Q while(a<b) do : → chance of ∞ loop if user loop.
x=y+z

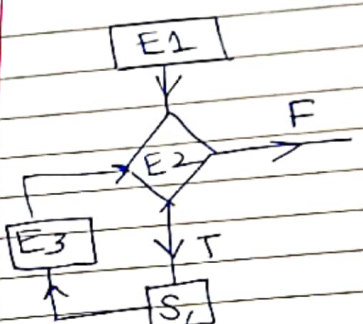
L: if a<b goto L1
goto Lext
L1: t=y+z
x=t
goto L
Lext:

SDT to gen. 3add code.

coming out of while loop.

HLL → intermediate code

for (E1 ; E2 ; E3)
initialise, cond^n, increment



(flow)

statement.

represented in inter code

for (i=0 ; i<10 ; i++)
a=b+c ;

i=0
L: if (i<10) goto L1
goto Lext
L1: t1=b+c
a=t1
t=i+1 → i++
i=t
goto L
Lext:

* No brackets/braces in IC. → difficult to understand
HLL → IC.
↳ we need to do code optimisation.

Scanned with CamScanner

* Switch → not there in 3 Add code

```
Switch(i+j)
{
  case(i) : a=b+c;
            break
  case(ii) : P=Q+R;
            break;
  default: x=y+z,
            break,
}
```

```
*   t=i+j
    goto test
L1  t1=b+c
    a=t1
    goto last        → ∴ break after every case.
L2  t2=Q+R
    P=t2
    goto last
L3  t3=y+z
    x=t3
    goto last
test: if(t==1) goto L1
      if(t==2) goto L2
      goto L3
last.
```

* 2 Dimensional Array to 3 Address code

→ indices

x = A[y,z]

* how 2D Array Stored in memory ?

A[4][4].   A[2 3].

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

: 2D Matrix

But, memory → 1D array.

→ row major order → Cases rM order default.

→ column major order.

⇒ [00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33]

[00 10 20 30 01 11 21 31 02 12 22 32 03 13 23 33].

cal → dont chang.

A[2 3] → user → cross 2 rows (2*4) + 3 elements = 11

A[2 3] → 2nd row, 3rd column

A[i, j] → cross i rows, then cross j elements → row

⇒ every element takes 4 bytes

A: 10×20

Q $x = A[y, z]$

{y*20 + z} → total elements crossed

{y*20+z}*4 → total bytes crossed

SA = 100
offset = 44
∴ Base offset = 144

A: 10×20

$x = A[y, z]$

$t_1 = y*20$	→ total ele cross
$t_2 = t_1 + z$	→ every ele = 4 bytes (assump)
$t_3 = t_2 * 4$	
t_4 = base add. of A	→ SA
$x = t_4[t_3]$	→ take the BA & add offset

- Help the programmers at user level.

GATE 2007 QUESTION

- OP Rj, Ri
- OP m, Rj
- MOV m, Rj
- MOV Ri, m   m = memory locn m

⇒ ADD R_1, R_2

$R_1 + R_2 \rightarrow R_2$
result in R_2

MOV → LOAD & STORE

R_1 $t_1 = a+b$ R_2. R_1
R_2 $t_2 = c+d$
R- $t_3 = e - t_2$
$t_4 = t_1 - t_3$

ADD d, R2

d loaded in R_2 & then added up with contents

$t_1 = a+b$

MOV a, R1 } t_1 $R_1 = a+b$
ADD b, R1

MOV c, R2 } t_2 $R_2 = c+d$
ADD d, R2   in R_2

SUB e, R2 } t3 $R_2 = e - r_2$
SUB R1, R2 } $R_2 = R_1 - R_2$
MOV R2, m

load R_2 into m.

∴ 3 Move oper^n's ✓

---

* GATE 2004 QUESTION

(Q.U)

* $E \longrightarrow E_1 - E_2$

R: E_2 Sub req. the first operand to be in register.

S

$E \longrightarrow E_1 - E_2$.   → put in reg.

When E_2 is evaluated first:

R: [E2] evalu. E_2
move → [ ] m

R: [E2]
eval E_1

R [E1]   [E2]

} less

· eval E_2
· move E_2 to M
· eval E_1
· Subt

SUB m, R.

When E_1 is eval. first

· eval E_1
· move E_1 to M.
· eval E_2
· mov. E_2 to M.
· load E_1
· Subtract

R: [E1] (evaluate).
move

R: [E1]   → [ ] m

move

R [E2]   m [E1]

evaluate.

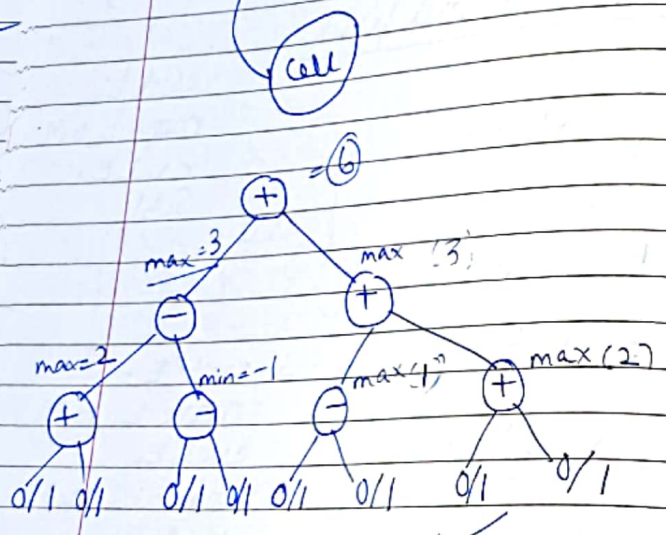
} more

R: [E1]   [E2]

∴ (b). ✓

GATE 2014

Q



0 0 0
0 1 -1
1 0 1
1 1 0

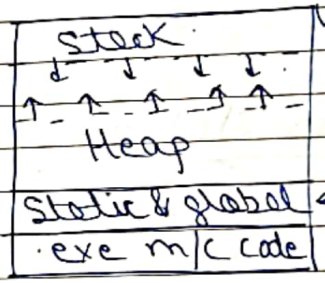
Max Value possible = 6.

Q

RUNTIME ENVIRONMENT

- when we run the prog (how prog is called & what support OS gives)

- execute prog → process
→ OS executes .exe file as process



OS provides mem for every proc.

← Static part

Static variables → perm storage
(array → static, by default)
need to specify size.

Heap → dynamic memallocn
↳ malloc().
↳ no strict boundary.

Stack: functn calling
↳ activn records pushed ↓.

S H

(both grow in app dirn) ↑ ↓

(to avoid wastage of memory)

if Heap → max, Stack → less

⇒ (if entire space used up → error)

→ Space reserved by OS for every process.

→ 3 Strategy techniques → S, H, S

* Storage Allocation Strategies

1) Static

(all)

- Allocn done at compile time
- Bindings dont chng at runtime (mem allocn fixed)
- 1 AR per procedure

So ∵ cant decide at runtime

Program

f_1
f_2
f_3
.
.
f_{10}

f_1(AR)
f_2
f_3

→ called only once

∴ if f_1 is called,

∴ Recursion not possible

→ dont know, no which funcn & times, it will be called

(Not Popular)

2) Stack

- when f is called, AR is pushed.
  - ↳ new space is created in Stack
  - ∴ Recursion ✓
- when funcn over, AR popped

Dis.

→ local Var. cant be retained once activn ends

eg:

```
f()
{
  int count = 0;
  count ++;
  printf(co
}
```

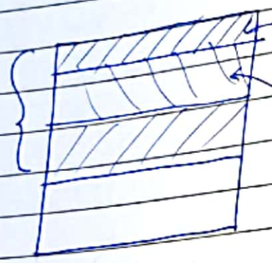
when we call f, fresh count will be created

f() | count = 01
f() | count = 01

local Variables.

3) Heaps:

Allocⁿ & deallocⁿ can be done in any order

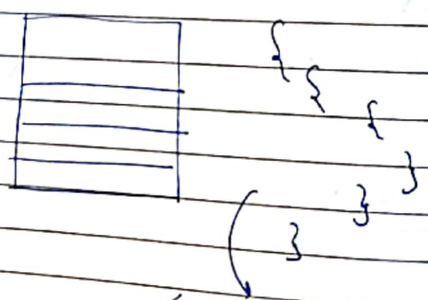


Because of hole, memory management is difficult.

eg: FN C: First fit used.

Summary:

AR → nested lifetime in case of stack allocⁿ.



→ We use Hybrid of all these Strategies.

INTRODUCTION TO CODE OPTIMIZATIO

- Phase after ICG
- reduce no. of lines in prog. (AIM)
- majority time of prog spent in loops.
- reduce no. of lines in loop → imp.
- reduce 1 line → 1000 lines

→ focus on loop.

Optimization
- → m/c independent: dont worry abt m/c.
- → m/c dependent: use features of m/c.
  - → depend on capacities of m/c.

* m/c independent

(call)

→ eg of each

* m/c dependent

(all)

* LOOP OPTIMISATIONS

- bigger picture

↓ detect loops

eg: for, while

HLL for( ) { } — visible.

↓

ICG — if, goto statements (not visible here)

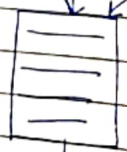
↓

(CO)

So,

↓ Use CFA using PFG to detect loops

↓ To find PFG, find basic block.



Seq. of 3 add code

∵ ICG = not visible

↓ opt

detect loops

↓

CFA (PFG)

↓

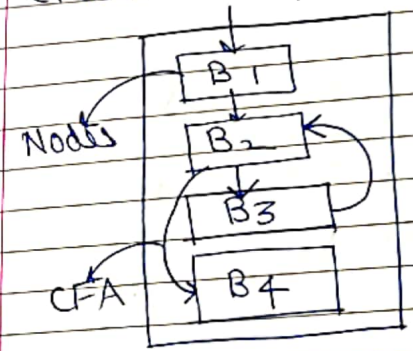
BB (seq. of 3 Add Code)

Chain react^n.

eg. (How BB helps in detecting loop)

(Prog: calculate of BB)

[PFG]



→ 4 BB.

if any cycle → loop

Q How to find Basic Blocks?

ALGORITHM TO FIND THE LEADERS

- find the BB.

L1 = Leader
L2 = Leader

L(1-9) = BB

* Identifying leaders in BB

1) I statement is always a leader.
2) Statement that is target of condn/uncondn statement is a leader.

if( ) goto 200 → leader : condn

Sa 300 : uncondn.

3) Statement that follows condn/uncondn goto = leader.

opt
↓
loops
↓
CFA(PFG)
↓
BB
↳ Leaders

flow

HLL

```
fact(x)
{
  int f=1;
  for(i=2; i<=x; i++)
      f=f*i;
  return f;
}
```

3 Addr. code (O/p of ICG)

Leader

1) f=1. | B1
2) i=2

B2 → conditional stat.

3) if(i>x) goto 9

immed. follows a condn goto

4) $t_1 = f * i$
5) $f = t_1$;
6) $t_2 = i + 1$;
7) $i = t_2$
8) goto (3)

B3

uncondn stat

9) goto calling function (ret) | B4

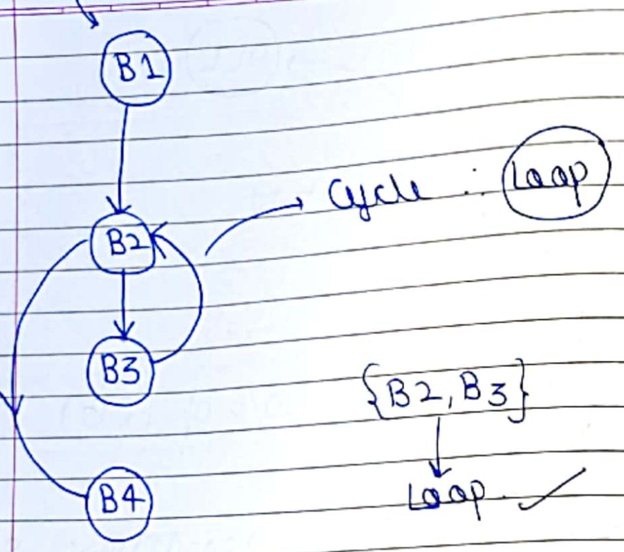
- Now CO comes.

4 leaders
(1), (3), (4), (9).

4 Basic Block.

n Leaders → n BB

CFA — Flow Graph



* TYPES OF LOOP OPTIMISATION

- decrease no. of lines in loop ①
- decrease no. of loops / no. of times a loop is executed ② ③

happens at 3 — add code

* frequency reductⁿ

moving code from high freq. region to low freq region is called code motion.

eg. i=0
```
while (i < 5000)
{
  A = sinx/cosx * i;
  i++;
}
```

⇓

```
t = sinx/cosx
while (i < 5000)
{
  A = t * i;
  i++;
}
```
→ reduced no. of division

* Loop unrolling

↓ (no. of times a loop executes).

- reduce no. of times, we do a comp.

```
while (i < 10)
{
  x[i] = 0
  i++
}
```
→ assigning one by one → 10 times

```
while (i < 10)
{
  x[i] = 0;
  i++;
  x[i] = 0;
  i++;
}
```
→ 5 times. → window = 2.

0 1 2 3 4 5 6 7 8 9

| 0 1 2 | 3 4 5 | 6 7 8 | 9 x x |

→ array index out of bound.

2, 5 ✓

↑ make every elem 0.

* Loop Jamming (↓ no. of loops)

→ combining 2 loop together
- decreasing no of loops.

(all)

* MACHINE INDEPENDENT OPTIMISATION → doesn't depend on hardware

(all)

⇒ Folding

eg: 2+3+C+B = 5+C+B.

replacing an exp that can be computed at compile time by its values

⇒ Redundancy elimn (DAG)

A = B+C
D = 2+B+3+C

D = 2+3+A
↓ folding
D = 5+A

If exp is already eval.

⇒ Strength redn

B = A * 2
↓
B = A << 2 (cheaper) → does in register.

⇒ Algebraic Simplifiⁿ

A = A+0
x = x * 1
} trivial.

eliminate all this.

* MACHINE DEPENDENT OPTIMIZATION

- depend on m/c.

(all)

1) register allocation
- depend no. of registers.

2) use of addressing mode

3)

x = y + z
- mov y, R0
- add z, R0
- mov R0, x

Same for $\left.\begin{matrix} a=b+c \\ d=a+e \end{matrix}\right\}$ → 6 lot.

```
mov b, R0
add c, R0
mov R0, a
mov a, R0
add e, R0
mov R0, d
```

↳ d = (b+c)+e.

(*) Flow of control optimiz^n

Avoid jumps or jumps

eliminate dead code.

```
# define x 0
if (x)
{
    x → never exec.
}
```

d) use of m/c idioms

i = i + 1	mov R0, i
	add R0, 1
	mov i, R0

or, inc i

if Processor has spcl instr. set.