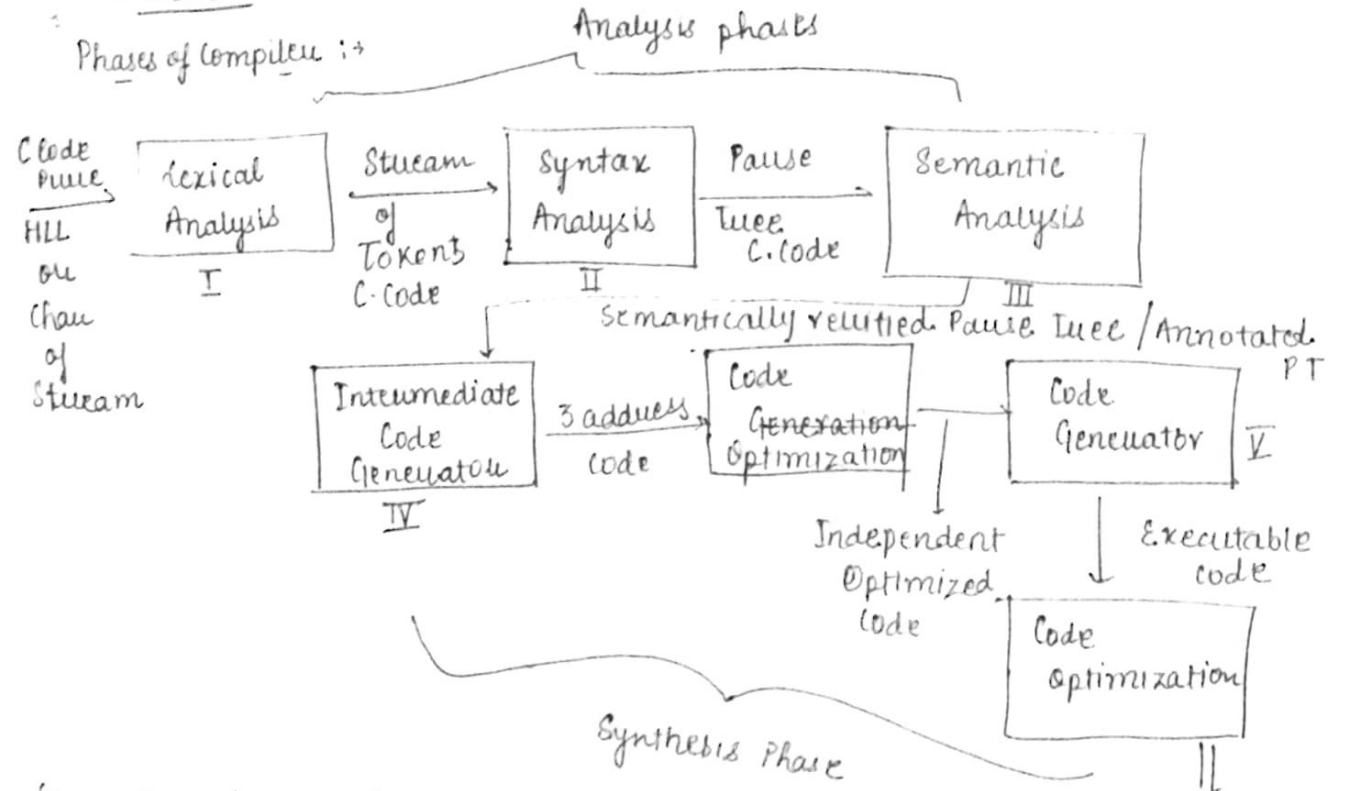
COMPILER DESIGN

Handwritten Notes!

Lecture - 01

Phases of Compiler :-



First 3 Steps - Analysis phase

Last 4 are - Synthesis phase

Total Phases 5 or 7

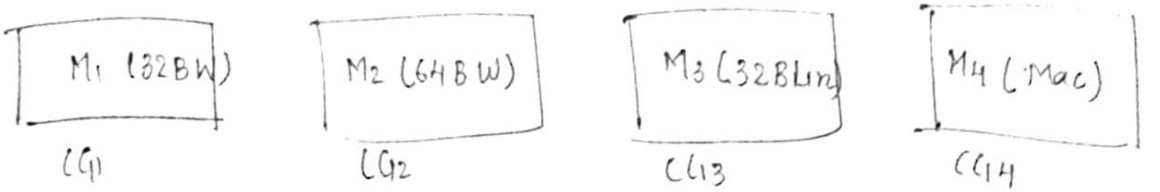
Code Optimization is not a crucial part of phases

Till ICG (IV) Compiler generates Independent Code & Till ICG we have Front end & Code Generator generate machine dependent Code & Comes under Back end

Why 7 Phases ?

If same compiler is to be applied to different machines of diff architecture Front end or Independent Code Generation part will remain same but the Code Generator will get changed according to the Architecture of Machine

In General for the below machines we need $4 \times 5 = 20$ phases



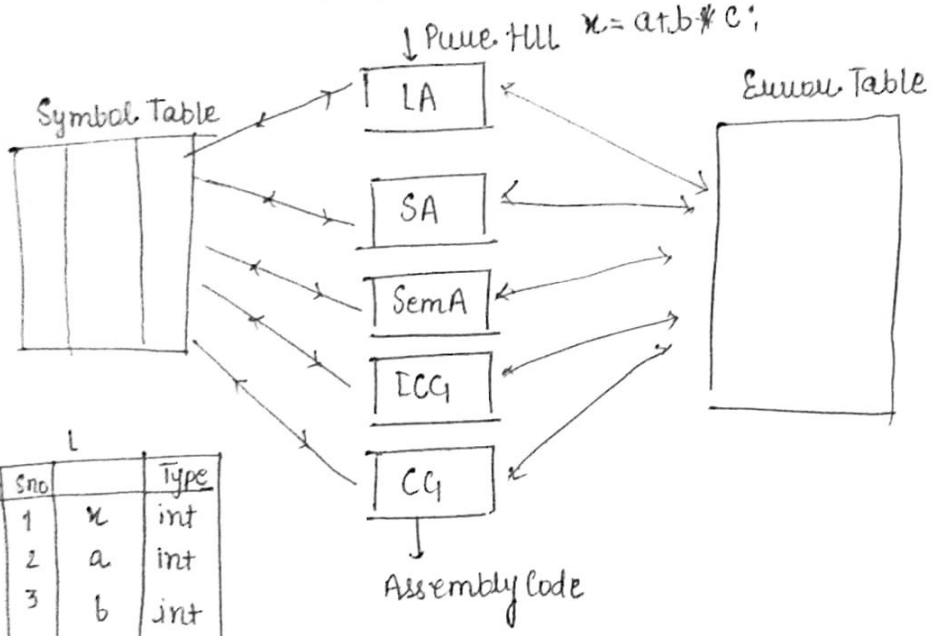
if we want to generate compiler we need only 4 Independent + 4 CGs → Total 8 phases

2) If I want different compilers on diff. Languages for same machine for Eg - C, C++, Java, python

We need - 4 × 4 + 1 = 17 phases
↓
4 languages × 4 phases each (Independent Code phases)

Symbol Table :→ Stores the Identifiers and Constants
- It is a data structure.

Error Handler Table :→ Store the Errors present in code.
- Also a data structure



I

Offset	Sno		Type
	1	x	int
	2	a	int
	3	b	int

→ data structure
List
Tree
Hash Table - O(1)
Mostly used

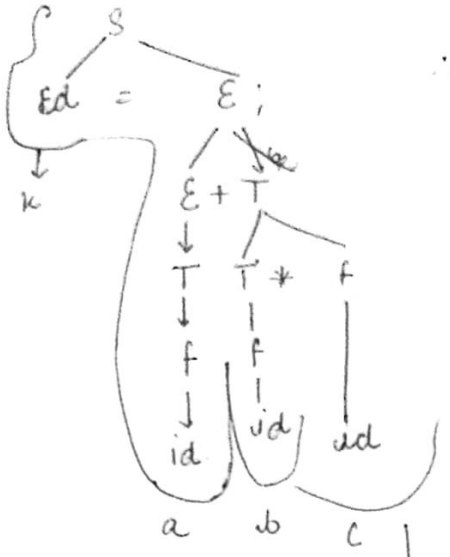
→ Operation in ST
→ Insert
→ Search
→ Delete
→ Modify

x = a+b*c;
↓
LA will scan it
id1, op, id2, op, id3, op, id4, Sp
Stream of Tokens
Syntax Analyser will check whether the stream of tokens can be generated by the CFG

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

Now Generating parse Tree.

Here,



∴ if in any case we can generate parse tree for the context tree grammar that matches the stream of tokens, this means the code is syntatically correct

This will move to Semantic Analyzer.

Semantic Analyser

↓

Semantically Verified Parse Tree → It will check
- Type checking
- Type conversion
- Variable declaration.

move to

ICG → will generate 3 address code

= Compute

Eg $t_1 = b * c$; } Three add. code
$t_2 = t_1 + a$;
$x = t_2$;

↓

C: Optimization → machine independent

↓

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

Lexical Analyser

C. Code → [L.A] → Stream of Tokens.

Lexical Analyser is a program that recognize 'lexemes' in the code and produce stream of tokens as a output and it also produce lexical error if any.

```
main()
{
int b;
}
```

(counter step)	lexeme	Token (internal representation of compiler)
①	main	id
②	(	Special Symbol / Operator token
③	)	" "
④	{	Special Symbol
⑤	int	Keyword
⑥	b	identifier
⑦	;	special symbol
⑧	}	Special Symbol

lexeme – Smallest unit of a given input

lexeme —forms→ Token

Tokens → Identifier – letter, digit (but not starting with digit)

↳ (l+_) (l+d+_)*

↳ Keyword – 32 type of Keyword in C

↳ Constant → 'A', 'Gate21'

↳ Special Symbol → (,), §, }

↳ Operator → +, -, *, /

functionality of LA → ~~Start from~~ 1. It recognise lexeme and produce stream of tokens.

2. It recognise white spaces, tab, comment and ignore or delete it.
3. It gives lexical error if any and also produce line no.
4. It scan start from beg of program and goes to end.
5. It matches the longest prefix.

↳ gate;

↳ g is an identifier but not a token

full gate is a token

Loop Holes in LA

x ? y : z ; 5 tokens ? : will be taken as single

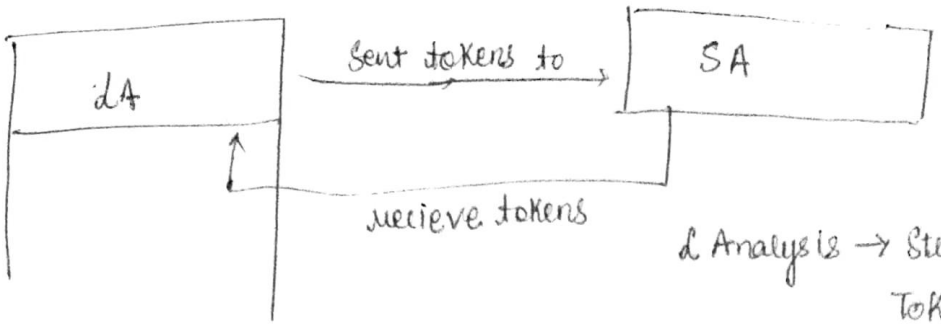
↳ LA will ask for Syntax analyzer to get no of tokens

x += y++;

↳ it is a single token 5 tokens

char *p = 'gate2021';

↳ This will be taken as single one.



L Analysis → Stream of Tokens

A [50];
↳ 4 tokens
[ ] will be a single token

Sizeof (x);
↳ 3 tokens

int x /* comment */; → 3 tokens
int x ; comment */ → 6 tokens

Build a Lexical Analyser
⇓
Pattern matching system
(l+_) (l+_+d)*
using RE ⇒ RL → FA

KISCO
- Keyword
- Special symbol
- operator
- Identifier
- constant

Ambiguous and Non Ambiguous grammar.
↳ Afterwards

Symbol table
↳ List
↳ Hashing
↳ Binary Tree.

y + z = x; → 6 tokens
no lexical error
Syntax error → can't generate parse tree.

int labc; → Lexical error
can't generate token

int float; → No Lexical Error - 3 tokens
Syntax error

int x = 10;
int y = 20; → No lexical error - 16 tokens
z = x + y; → Semantical Error z was not declared.

Grammar
{V, T, P, S}
↳ Variable / non terminal
↳ Terminal
↳ Production
↳ start symbol

Eg → E → E + E
E → E * E
E → E ↑ E
E → id
V = {E}
S → E
Terminal = {+, *, ↑, id}

Guammar

- Ambiguous: ∃ I P.T
- Nn ambiguous: ∃ 1

∃ more than one derivation / P.T for any word ∈ L(G)

∃ only one derivation / P.T for all word.

For guammar to be ambiguous guammar must have left and Right Recursion

For Eg ⇒ $S \to SS/a$

w = a
w = aa
w = aaa

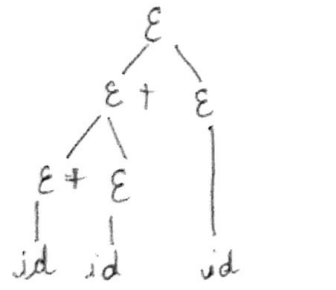
ambiguous

$S \to SaS \mid SbS \mid \in$ $S \to SSS/a$

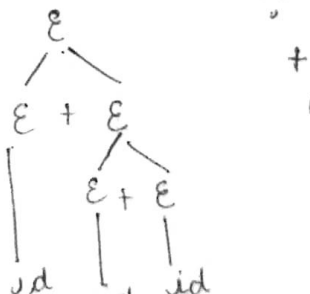
$E \to E+E/id$

Yes ambiguous

For w = id + id + id



For C ✓ it is correct



it is wrong

∵ in C we have + is left to Right Associative.

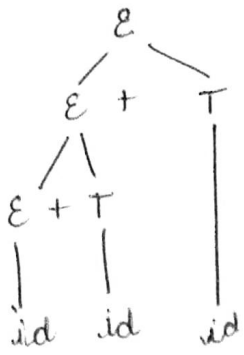
$E \to E+E/id$

⇓

unambiguous guammar

$E \to E+T/T$

$T \to id$



$E \to E+T \mid T$

$T \to id$

if the grammar is left recursive it is left associative

we can write it as

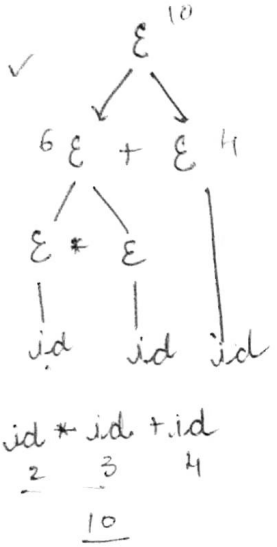
$E \to E+id \mid id$.

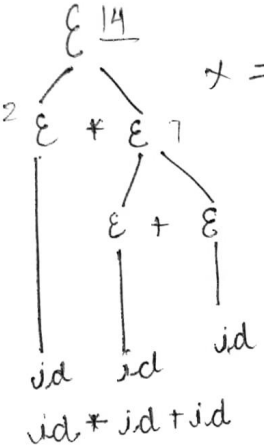
we can convert some ambiguous grammar to unambiguous

↓ Not all

$E \to E+E \mid E*E \mid id$ ⟶ unambiguous ⟶ $E \to E+T \mid T$

$T \to T*f \mid f$





$f \to id$ ✗

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

$E \to E+T \mid T$

$T \to T*f \mid f$

$f \to f \uparrow D \mid D$

$D \to id$

$\uparrow > * > +$

↑ is right associative

$*_L > *_R$

$\uparrow_L < \uparrow_R$

~~Access~~

There is no ~~grammar~~ algorithm that would find out whether given grammar is ambiguous.

~~Amig~~ Ambiguous → 2 parse tree → 2 LMD or 2 RMD

Left Most Derivation

if LMD = RMD the grammar is unambiguous. ← meaning

(if Both side derivation give same ans)

Lecture 3 / 19

$A \to A@B / B$

$B \to B\#C / C$

$C \to C\$D / D$

$D \to id$

fauthen the symbol mene precedence

$\boxed{\$ > \# > @}$

We can't convert all ambiguous grammar to unambiguous undecidable problem.

$\$_L > \$_R$

$\#_L > \#_R$

$@_L > @_R$

~~Lecture~~

Ques

c? $(b \text{ ou } a)^* c$

T_3 T_3

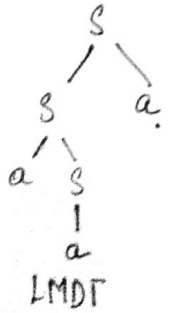
(C^*)

$\underbrace{bbaac}_{T_3}$ $\underbrace{abc}_{T_3}$

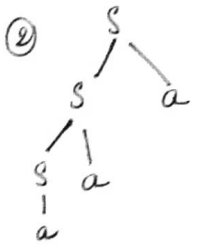
Lecture 4 / 19

Ambiguous and Unambiguous

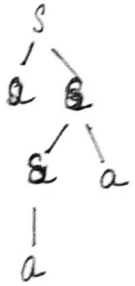
$S \to Sa / aS / a$

① 

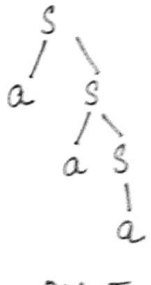
LMDT

② 

Left Most Derivation Tree.

③ 

RMDT

④ 

RMDT

→ Hence we have 2 LMDT and 2 RMDT and grammar is Ambiguous

→ The ambiguous grammar from which we cannot eliminate ambiguity is called inherently ambiguous grammar.

→ For Unambiguous Grammar LMDT = RMDT

Context free Grammar

Variable → (Variable + Terminal)*

- Recursive
  - $S \to Sa/b$
  - There exist at least one production having same variable in LHS & RHS
  - Recursion
    - Left Recursion
      - $S \to Sa/b$
      - $S \to SS/a$
      - Both left and Right
    - Right Recursion
      - $S \to aS/b$
- Non Recursive
  - $S \to AB$
  - $A \to a$
  - $B \to b$

$A \to A\alpha / \beta \Rightarrow$ Top Down Parser / RDP

Left Recursion → Infinite loop → Recursive Descent Parser

↳ it will generate

$B, B\alpha, B\alpha\alpha \ldots B\alpha^*$

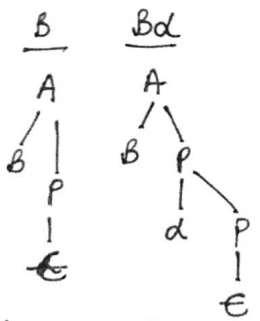
To make it pass through Top Down parser we need to convert it into Right Recursive Grammar.

$A \to A\alpha/B$

$\Downarrow$

$A \to BP$

$P \to \epsilon/\alpha P$



Eg-2

$A \to A\alpha / \beta_1 / \beta_2 / \beta_3 \ldots \beta_n$

$\Downarrow$

$A \to \beta_1 P / \beta_2 P / \ldots \beta_n P$

$P \to \epsilon / \alpha P$

Eg-3

$A \to A\alpha_1 / A\alpha_2 / A\alpha_3 \ldots - A\alpha_n / \beta$

$\Downarrow$

$A \to \beta P$

$P \to \epsilon / \alpha_1 P / \alpha_2 P / \alpha_3 P \ldots \alpha_n P$

Eg-4

$A \to A\alpha_1 / A\alpha_2 \ldots A\alpha_n / \beta_1 / \beta_2 \ldots \beta_n$

$\Downarrow$

$A \to \beta_1 P / \beta_2 P \ldots / \beta_n P$

$P \to \epsilon / \alpha_1 P / \alpha_2 P / \alpha_3 P \ldots / \alpha_n P$

Eg-5

$E \to E + T / T$

$T \to T * F / f$

$f \to id$

$\Downarrow$

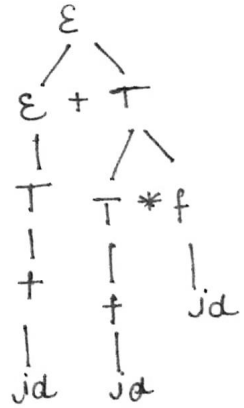
$E \to TP$

$P \to \epsilon / +TP$

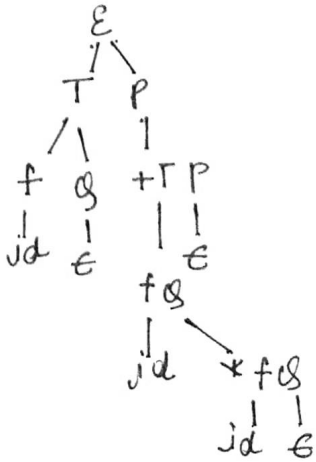
$T \to fQ$

$Q \to \epsilon / *fQ$

$F \to id$



id + id * id



id + id * id

Que2 $A \to Aab/c$.

Ans
$A \to cT$
$T \to \epsilon / abT$

Que3 $S \to (L)/a$
$L \to L,S/S$

Ans
$S \to (L)/a$
$L \to SP$
$P \to \epsilon /,SP$

Que4) $A \to A(A)/a$
$A \to aT$
$T \to \epsilon /(A)T$

Que5 $S \to SSS/a$
$S \to aS'$
$S' \to \epsilon / SSS'$

Que6 $S \to A$
$A \to Ad/Ae/aB/aC$
$B \to bBC/f$
$S \to A$
$A \to aBP/aCP$
$P \to \epsilon /dP/eP$
$B \to bBC/f$

Que7
$S \to AaB$
$A \to aA/ABa/ba$
$B \to AB/b$
indirect Recursion
$S \to AaB$
$A \to aAA'/baA'$
$A' \to \epsilon / BaA'$
$B \to AB/b$

Que8
$R \to RR/Ra/aR/b$
$R \to bT/aRT$
$T \to \epsilon /RT/aT$

Lecture 5

General Recursion – Neither left Recursion nor Right

Eg $S \to aSb/b$.

Recursion
- Direct Rec: L, R, Gen
- Indirect Rec: L, R, Gen

Que1
$S \to aBDh$
$B \to Bb/h$
$D \to Ef$
$E \to g/\epsilon$
$f \to f/\epsilon$
$\Rightarrow$
$S \to aBDh$
$B \to hT$
$T \to \epsilon /bT$
$D \to Ef$
$E \to g/\epsilon$
$f \to f/\epsilon$

Que2 $E \to E+E/E*E/id$
$E \to id\,T$
$T \to \epsilon / +ET/*ET$

Que3 $A \to AaAb/AbAa/\epsilon$ ★★
$A \to \epsilon T$
$T \to \epsilon /aAbT/bAaT$
$\epsilon T \Rightarrow T$

Que4 $S \to Aa/b$
$A \to Ab/Sc/\epsilon$
S ⇓

$S \to Aa / b$
~~$A \to \ldots T / T$~~
$A \to Ab / Aac / bc / \epsilon$
⇓
$S \to Aa / b$
$A \to T / bcT$
$T \to \epsilon / bT / acT$

Que 5 $A \to Aa / aA / b$
⇓
$A \to aAT / bT$
$T \to \epsilon / aT$

Que 6 $L = \{\epsilon, a, b, c \ldots z\}$
*** Select set which rep all 4 letter word
a) $L+L+L+L$
b) L^2+L^2
c) $L^4 - L^3$ ✓
d) L^4

L^0 represent all ϵ
L^1 " " one letter word
L^2 " " two " "
L^3 " " three " "
L^4 " " four " "

$L^4 - L^3$ will give words with 4 letter only

Que 7 two binary operator
*** $\uparrow > \downarrow$ (precedence)
$a \uparrow b = a^b$
$a \downarrow b = \log_b a$
$\uparrow$ = Right Associative
$\downarrow$ = Left "

Compute $65536 \downarrow 2 \uparrow 4 \uparrow 2 \downarrow 2$

1. $65536 \downarrow$ ~~2↑4↑2~~ 8 ... 1

8	65536
8	8192
8	1024
8	128

8	16
2	2

2/16, 2 4, 16/4 4

10 Ans

Que

Operators	Precedence	Association
*	3	Left
−	4	Left
+	2	Right
↑	1	Right

greater no greater precedence

$10 - 6 - 2 \uparrow 8 - 6 + 1 * 2 \uparrow 1 + 1$
$\Rightarrow 2 \uparrow 2 + 1 * 2 \uparrow 1 + 1$
$\Rightarrow 2 \uparrow 2 + 2 \uparrow 1 + 1$
$\Rightarrow 2 \uparrow 4 \uparrow 2$
$\Rightarrow$ **65536**

Que no of Tokens

I

```
Switch (inputvalue)
{
 case 1: b = c * d; break;
 Default: b = b++; break;
}
```

26

II

```
main()
{ int a, b;
  a = 10;
  b = 15;
  printf ("a = %d, b = %d", a++, b--);
}
```

29 ~~/37~~

Que
= Count no of Tokens

① printf ("the no of tokens are %d", &tcount);

8

② printf ("A %.B =", &i); 8

③ float abs_zero_Kelvin = -273;

6

④ X = X + (a + b) - 5;

12

⑤ abcd + (3 - 5 + x 9 1/3 - + ;

14

Que Output of lexical analyzer
~~a) Set of Reg expressions~~ d) Set of token

Que Not a functionality of C Compiler
a) Identifying Token
b) Identifying Syntax error
c) Linking
d) Loading

Que who check token arrangement
Syntax Analyzer
token generation is done by
Lexical Analyzer

Token
→ String of characters which logically belong together
→ A pair consisting of a token name and a optional attribute value.

Lexeme
→ The sequence of characters matched by a pattern to form corresponding token

Pattern
It is a description of the form that the lexeme of a token may take.

x = a + b + c;

Lexeme	Token
x	id
=	SpSy
a	id
+	op
b	id
+	op
c	id
;	SpSy

Parsers 6th sect

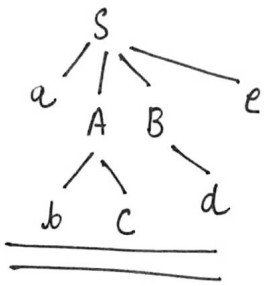
- Top Down Parsers
  - With Backtrack
    - Brute force Method
    - Recursive Descent Parsers (general)
  - Without Backtrack (Predictive Parsers) 1. LL 2. NLR
    - RDP (program)
    - LL(1) for LL we need grammar to be Left factored
- Bottom Up Parsers (Shift Reduce Parsers)
  - OPP Operator Precedence Parsers — Only Parsers can pass both Ambiguous & Unambiguous Grammars — Can't process all CFG But can parse operator grammar
  - unambiguous
    - LR Parsers
      - LR(0)
      - SLR(1)
      - CLR(1)

Top Down Parsers
→ Pass Unambiguous Grammar
→ No left Recursion

How TDP work?

$S \to aABe$
$A \to bc/b$
$B \to d$

$w = abcde$



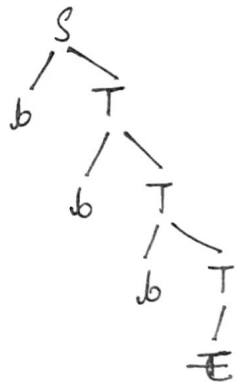
① Start from Start Symbol goes to string
② Perform Left most derivation

Q $S \to Sb/b$

$\Downarrow$

$S \to bT$
$T \to \epsilon / bT$

$w = bbb$



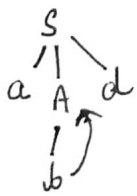
With Backtrack

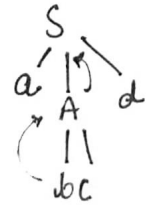
Brute force Method :→ (try all possibility)

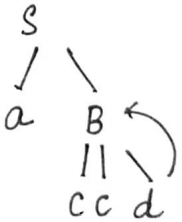
→ Costly
→ $O(2^n)$

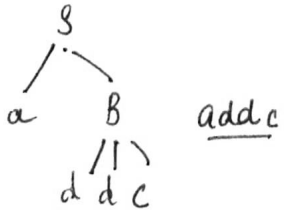
$S \to aAd/aB$
$A \to b/bc$
$B \to cd/dc$

$w \to addc$









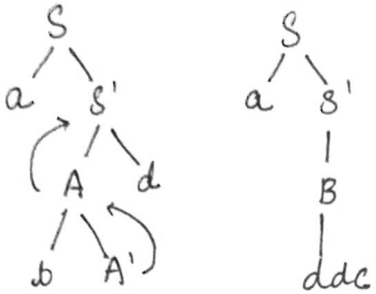
addc

Due to Heavy cost of Brute force we developed Recursive Descent Parser
Brute force method is programmed using recursion

General RDP (using Backtrack)

- Slightly smarter than Brute force
- Use Backtracking

Eg → $S \to aS'$
$S' \to Ad / B$
$A \to bA'$
$A' \to c / \varepsilon$
$B \to ccd / ddc$



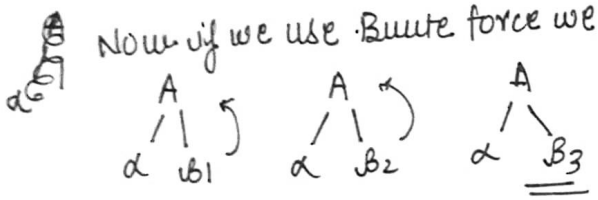
it is smarter so it will check the next input with the string and remove the unusal

Left factor Grammar

$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$

$w = \alpha\beta_3$

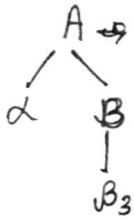
Now if we use Brute force we need to Backtrack again and again



we use left factored

$A \to \alpha B$
$B \to \beta_1 / \beta_2 / \beta_3$

Now Here it become simple for parser and it won't require Backtrack



Eg 2 $A \to aAb / aAd / f$

⇓ left factored

$A \to aAB / f$

$B \to b / d$

Now Eg 1 → $S \to Aa / b$
$A \to bCA / c$
⇓
$S \to bcAa / ca / b$
$A \to bcA / c$
⇓
$S \to bD / ca$
$A \to bcA / c$
$D \to cAa / \varepsilon$

because Here its nothing

Eg - 3 $S \to a / ab$
$S \to aA$
$A \to b / \varepsilon$

Eg → A → (A) A / (A) Ab / (A) Aba / e / f

⇓

A → (A) AT / e / f

~~T → A / Ab / Aba~~ T → ∈ / b / ba

⇓

~~A → (A) T / e / f~~ A → (A) AT / e / f

T → ∈ / bP

P → ∈ / a

~~T →~~

Eg → S → iEts / iEtses / a / b

S → iEtsP / a / b

P → ∈ / es

Eg → S → Sa / b / bC

S → ~~aCf~~ bT / bCTE

T → ∈ / aT

⇓

S → bB

B → T / CT ⇒ B → ∈ / aT / CT

⇓

S → bB

B → ∈ / aT / CT

Eg → Lecture 7/8

Without Backtrack

Note:→ If grammar is Ambiguous CFG then it can be Non-left factored

eg- S → A / a } S → a / a

A → a

common prefix

S → a

S → A → a

If grammar is left Recursive ⇒ Non left factored

common prefix = Non left factored

Recursive Descent parser :-

(Program) without Backtrack

E → id f

f → * id f / ∈

w = id * id * id

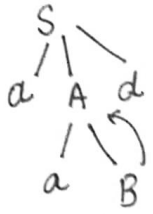
Backtrack depend on grammar

S → aAd / aB

A → a / aB

B → ccd / ddc

w = addc



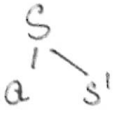
We are Backtracking due to non left factored grammar

S → aS'

S' → Ad / B

A → aP P → ∈ / B

B → ccd / ddc



Even though grammar is left factored it can be backtracking

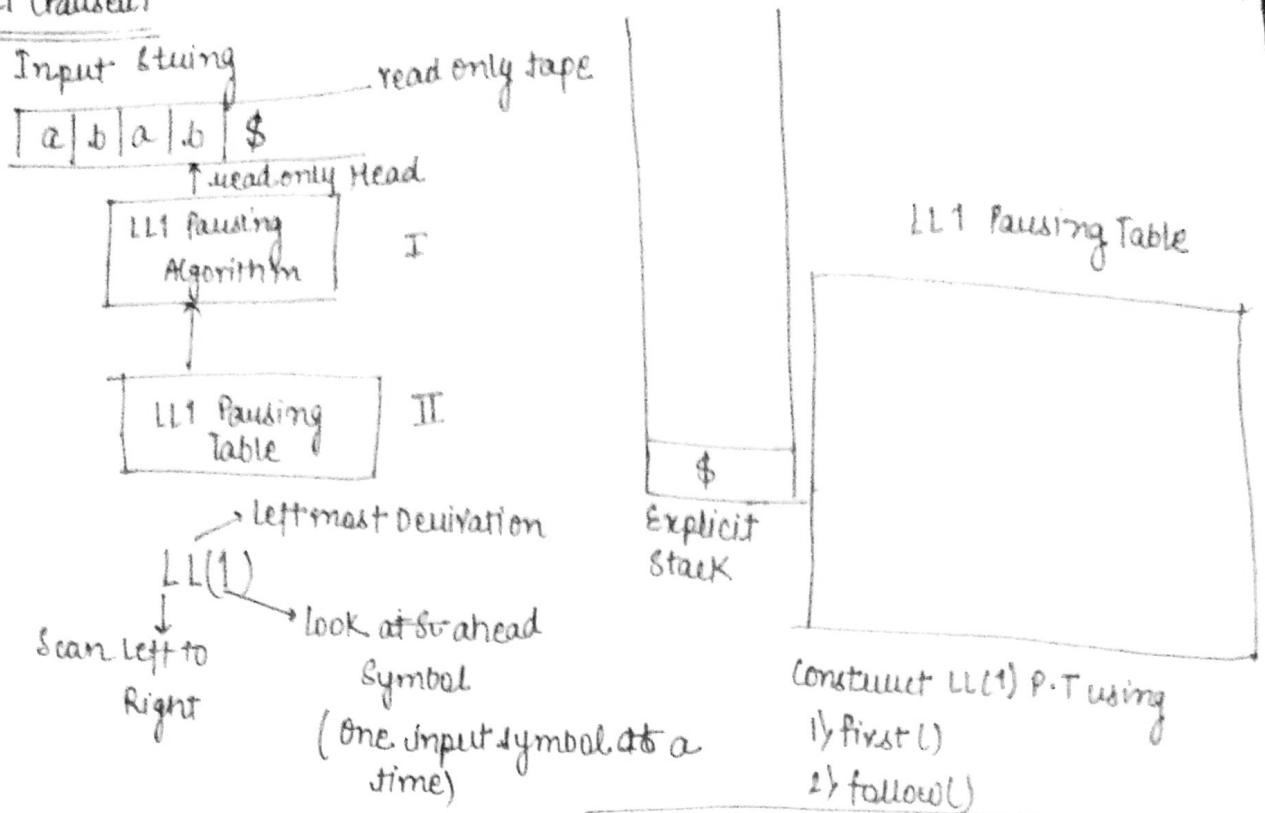
∴ We required LL1 parser

→ Unambiguous

→ Non left Recursive

→ Left factored

LL1 (Pausew)



first (x) → Set of terminals

1) A → a | b

first(A) = {a, b}

2) A → a | b | ϵ

first(A) = {a b ϵ}

3) S → a A | ϵ A → b

first(S) = {a, ϵ}

first(A) = {b}

4) S → Ae

A → bc | ϵ

first(S) = {a, b}

first(A) = {b, ϵ}

FIRST And FOLLOW function dec -8

FIRST(α) :→ first of α gives set of all terminals that begin in string derived from α

α → abc / bcd / ade first(α) = {a, b, d}

RULES

1. FIRST(α) = α if α is terminal
2. FIRST(ϵ) = ϵ
3. FIRST(α)

α → X1 X2 X3

FIRST(α) → FIRST(X1 X2 X2)

FIRST(X1) if X1 → ϵ

then FIRST(X2) if X2 → ϵ

then FIRST(X3) if

Que 1) $E \to TE'$

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

$T \to fT'$

$T' \to \epsilon / *fT'$

$f \to id / (E)$

FIRST(E) = {first(T) → first(f) → first(id) ∪ first(() = {id, (}

&

first(E') → first(ε) ∪ first(+TE') = {ε, +}

first(T) = {id, (}

first(T') = {ε, *}

first(f) = {id, (}

Que 2) $S \to aA / bB$

$A \to AB / c$

$B \to cB / a$

$A \to cT$

$T \to \epsilon / BT$

	first( )
S	a, b
A	c
B	c, a

Que 3) $S \to AB$

$A \to aA / \epsilon$

$B \to bB / \epsilon$

first	
a, b, ε	S
a, ε	A
b, ε	B

Que 4) $S \to AaB$

$A \to cA / \epsilon$

$B \to dB / e$

	first( )
S	c, a
A	c, ε
B	d, e

Que 5) $S \to AaB / Ba$

$A \to Ba / c$

$B \to aB / \epsilon$

	first( )
S	c, a, ε
A	a, ε, c
B	a, ε

Que 6) $S \to ABCDE$

$A \to a / \epsilon$

$B \to b / \epsilon$

$C \to c / \epsilon$

$D \to d / \epsilon$

$E \to \epsilon$

	first( )
S	a b c d ε
A	a ε
B	b ε
C	c ε
D	d ε
E	ε

Que 7) $S \to Aa Ab / BbBa$

$A \to \epsilon$

$B \to \epsilon$

	first( )
S	a b
A	ε
B	ε

Que 8) $S \to AaB / \epsilon$

$A \to BbA / a$

$B \to Cb / d$

$C \to e / f$

	first( )
S	ε e f a d
A	a d e f
B	d e f
C	e f

Que 9) $S \to AB$

$A \to aA / b / \epsilon$

$B \to c / d / \epsilon$

	first( )
S	a b c d ε
A	a b ε
B	c d ε

Que 10) $S \to (L) / a$

$L \to SL'$

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

	first( )
S	(a
L	(a
L'	ε ,

Que 11 $S \to aBDh$

$B \to cC$

$C \to bC/\epsilon$

$D \to Ef$

$E \to g/\epsilon$

$f \to f/\epsilon$

	first()
S	a
B	c
C	b ε
D	g f ε
E	g ε
f	f ε

Que 12 $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g/\epsilon$

$C \to h/\epsilon$

	first()
S	d, g, h, ε, b, a
A	d, g, h, ε
B	g, ε
C	h, ε

FOLLOW functions dect - 9

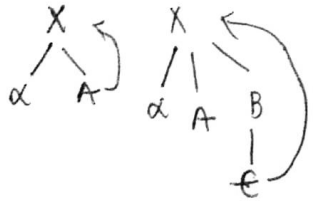
follow(A) → Set of all terminal that may follow immediately to right of A.

RULE

1. if A is start symbol of grammar then follow(A) is minimum $
2. if $X \to \alpha AB$ follow(A) = first(B)
3. $X \to \alpha A / \alpha AB$

   $B \to \epsilon$

   follow(A) = follow(X)



Que 1 $S \to a/b/\epsilon$

follow(S) ⇒ $

first(S) → {a, b, ε}

Que 2 $S \to aA/\epsilon$

$A \to b$

	follow()
S	$
A	$

Que 3 $S \to aAB/\epsilon$

$B \to b/\epsilon$

$A \to b/\epsilon$

	follow()
S	$
A	c ε $
B	$

Que 4 $S \to AB$

$A \to aA/\epsilon$

$B \to bB/\epsilon$

	follow()
S	$
A	b $
B	$

Que 5 $E \to TE'$

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

$T \to fT'$

$T' \to \epsilon / *fT'$

$f \to id/(E)$

	follow()	first()
E	ε) $	id (
E'	ε) $	ε +
T	+ ε) $	id (
T'	+ ε) $	ε *
f	$ *) +	id (

Que6 $S \to (L) / a$

$L \to SL'$

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

	first()	follow()
S	(a	$,)
L	(a	)
L'	ϵ ,	)

Check for Both

Que7 $S \to Aa\,Ab / Bb\,Ba$

$A \to \epsilon$

$B \to \epsilon$

	first()	follow()
S	a , b	$
A	ϵ	a b
B	ϵ	b a

Que8 $S \to Aa$

$A \to bcd / \epsilon$

	first()	follow()
S	b , a	$
A	b , ϵ	a

Que9 $S \to ABCDE$

$A \to a / \epsilon$

$B \to b / \epsilon$

$C \to c / \epsilon$

$D \to d / \epsilon$

$E \to \epsilon$

	first	follow
S	a b c d ϵ	$
A	a ϵ	b d $ c
B	b ϵ	c $ d
C	c ϵ	d $
D	d ϵ	$
E	ϵ	$

Que10 $S \to AaB / Ba$

$A \to Bc / b$

$B \to aB / \epsilon$

	first()	follow()
S	b, a, ϵ	$
A	a , b	a
B	a , ϵ	$, a , c

Que11 $S \to ACB / CbB / Ba$

$A \to da / BC$

$B \to g / \epsilon$

$C \to h / \epsilon$

	first()	follow()
S	d g h ϵ b a	$
A	d g h ϵ	$ h g
B	g ϵ	$ a h g
C	h ϵ	$ g b h

Que12

$S \to aBDh$

$B \to cC$

$C \to bc / \epsilon$

$D \to Ef$

$E \to g / \epsilon$

$f \to f / \epsilon$

	first()	follow()
S	a	$
B	c	g f h
C	b ϵ	g f h
D	g f ϵ	h
E	g ϵ	h f
f	f ϵ	h

Que13 $S \to Aa\,Ab / Bb\,Ba$

$A \to C / \epsilon$

$B \to d / \epsilon$

	first()	follow()
S	a C d b	$
A	C ϵ	a b
B	d ϵ	b a

Que14 $S \to AaB / \epsilon$

$A \to BbA / a$

$B \to Db / d$

$D \to e / f$

	first()	follow()
S	ϵ, a, d, e, f	$
A	a, d, e, f	a
B	d, e, f	$, b
D	e, f	b

How to Construct LL(1) Parsing Table Construction algorithm :→

Lect -10

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

1. Add $A \to \alpha$ under $M[A, b]$ where $b \in first(\alpha)$
2. if first (α) contain ϵ then add $M[A, C]$ where $C \in follow(A)$

Que Construct LL(1) Parsing Table for following grammar

$E \to E+T/T$
$T \to T*f/f$
$f \to id$

$\Rightarrow$

$E \to TE'$
$E' \to \epsilon / +TE'$
$T \to fT'$
$T' \to \epsilon / *fT'$
$f \to id / (E)$

Terminals

M	+	*	(	)	id	$
E	—	—	$E \to TE'$	—	$E \to TE'$	—
E'	$E' \to +TE'$	—	—	$E' \to \epsilon$	—	$E' \to \epsilon$
T	—	—	$T \to fT'$	—	$T \to fT'$	—
T'	$T' \to \epsilon$	$T' \to *fT'$	—	$T' \to \epsilon$	—	$T' \to \epsilon$
f	—	—	$f \to (E)$	—	$f \to id$	—

Variable E', Non Terminal T, T', f

① $E \to TE'$ $M[A, b]$
$A = E$
$b = first(TE')$
$= id$ (

② $E' \to \epsilon$ $M[A, C]$
C is follow(E')
$E' \to +TE'$
$M[A, b]$
$A \Rightarrow E'$
$b = \$$

③ $T \to fT'$
$M[A, b]$
$A \Rightarrow T$
$b = id$ (

④ $T' \to \epsilon$
$M[A, C]$
$A \Rightarrow T'$
$C = follow(T')$
$= + \$$

⑤ $T' \to *fT'$
$M[A, b]$

⑥ $f \to id$
$f \to (E)$

Ans Since each entries contain single production then the given grammar is LL1

Que 2 Construct LL1 Parsing Table for following grammar

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

$S \to (L) / a$

$L \to SL'$

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

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

① $S \to (L)$
$M[A, b]$
$A = S$
$b = first((L)) = ($

② $S \to a$
$M[A, b] = M[S, a]$

③ $L \to SL'$
$M[A, b]$
$M[L, first(SL')] \to [L, \{(, a\}]$

④ $L' \to \epsilon$
$M[A, C]$
$M[A, follow(L')] = [A, \{), \$ \}]$

⑤ $L' \to , SL'$
$M[A, b] = [L', ,]$

Since each entry contain single production so grammar is LL1.

If any entry contain multiple production then it is not LL1

Que3 Construct LL1 P.T

$S \to AaAb / BbBa$
$A \to \epsilon$.
$B \to \epsilon$

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	—
A	$A \to \epsilon$	$A \to \epsilon$	—
B	$B \to \epsilon$	$B \to \epsilon$	—

$S \to AaAb$
$M[A, b] = [S, \{a\}]$

This is also LL1.

Que4 $S \to aSa / \epsilon$

$S \to aSa$
$M[A, b] = [S, \{a\}]$

$S \to \epsilon$
$M[A, C] = [S, a]$

	a	$
S	$S \to aSa$ $S \to \epsilon$	$S \to \epsilon$

Since each entry doesn't contain single production ∴ Grammar isn't LL1

Que5 $S \to A / a$ ← Ambiguous also Not Left factored

$A \to a$ $S \to a$

$S \to A$

$M[A, b] = [S, \{a\}]$

	a	$
S	$S \to A/a$	-
A	$A \to a$	—

Not LL1 Grammar

Que6 $S \to aA$
$A \to b/\epsilon$

① $S \to aA$
② $A \to b$
③ $A \to \epsilon$

M	a	b	$
S	$S \to aA$		
A		$A \to b$	$A \to \epsilon$

Since each entry is singular therefore Grammar is LL1

Que7 $S \to AB$
$A \to a/\epsilon$
$B \to b/\epsilon$

① $S \to AB$
$[S, \{a, b, \$, \epsilon\}]$ ← Become follow of S

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to \epsilon$	$A \to \epsilon$
B	—	$B \to b$	$B \to \epsilon$

② $A \to a$ ③ $A \to \epsilon$ ④ $B \to b$ ⑤ $B \to \epsilon$

Lecture 11 /18

P.T is storing Grammar in form of Table.

Now we need to check input string is syntatically correct corresponding to grammar or not using stack and LL1 Parsing algorithm.

SHORT TRICK FOR CHECKING WHETHER GRAMAR IS LL1 OR NOT

LL(1) :
1. Unambiguous
2. non left Recursive / Right Recursive
3. Left factored

① $A \to \alpha_1/\alpha_2$ if $FIRST(\alpha_1) \cap FIRST(\alpha_2) = \phi$ ..
Then grammar is LL1

② $A \to \alpha_1/\epsilon$ if $FIRST(\alpha_1) \cap follow(A) = \phi$
Then grammar is LL1.

Eg → $S \to ab/cd$
$first(ab) \cap first(cd) = a \cap c = \phi$ LL1.

Eg2 → $S \to bSb/\epsilon$ $FIRST(bSb) \cap follow(S)$
$= b \cap b = b \neq \phi$ Not LL1.

Eg3 → $S \to (L)/a$
$L \to SL'$
$L' \to \epsilon/, SL'$

$first((L)) \cap first(a) = (\cap a = \phi$

$first(,SL') \cap foll(L') = , \cap) = \phi$

∴ In LL1

lecture-11/18 LL1 Parser TOP DOWN PARSER

If Grammar contains

① $A \to \alpha_1/\alpha_2/\alpha_3$ Here all α doesn't contain null($\in$)

then

$first(\alpha_1) \cap first(\alpha_2)$
$first(\alpha_2) \cap first(\alpha_3)$ } Pairwise disjoint
$first(\alpha_1) \cap first(\alpha_3)$

then grammar is LL1.

② $A \to \alpha_1/\alpha_2/\in$

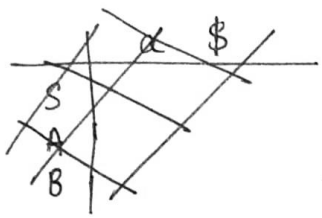
$first(\alpha_1) \cap first(\alpha_2) = \phi$
$first(\alpha_1) \cap follow(A) = \phi$ then grammar is LL1.
$first(\alpha_2) \cap follow(A) = \phi$

Eg → $S \to AB$
$A \to aA/a$]— Not LL1.

2) $E \to +TE/\in$ — yes

$T \to (f)$
$f \to id$

it is LL1.



Note:→ Every Ambiguous grammar is not LL1.

2. Every unambiguous grammar need not to be LL1.
3. Every LL1 Grammar is unambiguous.

LL(1) Algorithm :→ 1. Let X is top of stack and a is input symbol or look ahead symbol.

if (X == a == $) successful completion of parsing (accept)

if (X == a ≠ $) then pop Top of Stack and Increment Input Pointer.

if (X is a variable/nonterminal then see p.T M[X, a]

$X \to xyz$ then pop X and push xyz in reverse order.

if M[X, a] = blank then Error.

BOTTOM UP PARSER

Lect - 13/18

Parsers

- Top Down Parsers
- Bottom Up Parsers
  - Operator Precedence Parsers (OPP)
    - It can Pass both Ambiguous & Non Ambiguous Grammar
  - LR Parsers
    - LR(0)
    - CLR(1) or LR(1)
    - SLR(1)
    - LALR(1)

Bottom Up Parsers :→

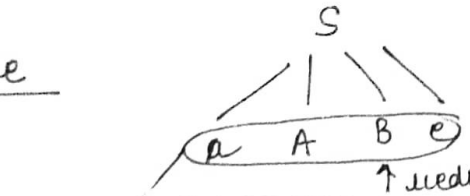
How BUP Work?

$S \to aABe$

$A \to bc$

$B \to d$

$w = abcde$



Reverse of Right Most Derivation.

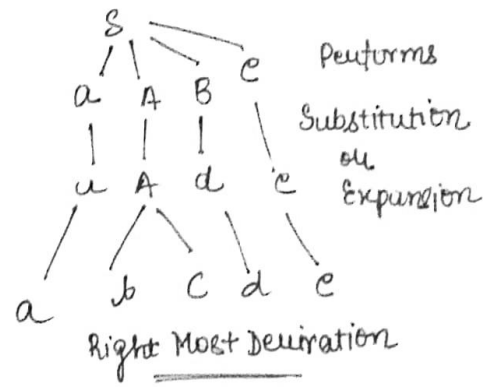
LR → Reverse of Right Most Derivation

↳ Scan Left to Right

SLR - Simple LR

LALR - Look Ahead LR

CLR - Cannonical LR



Handle - Part of i/p string that matches Right Hand Side of production

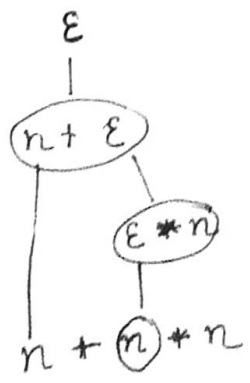
Handle Pruning - Replacing the variable with its LHS-Variable is known as Handle Pruning

Eg → Consider a grammar

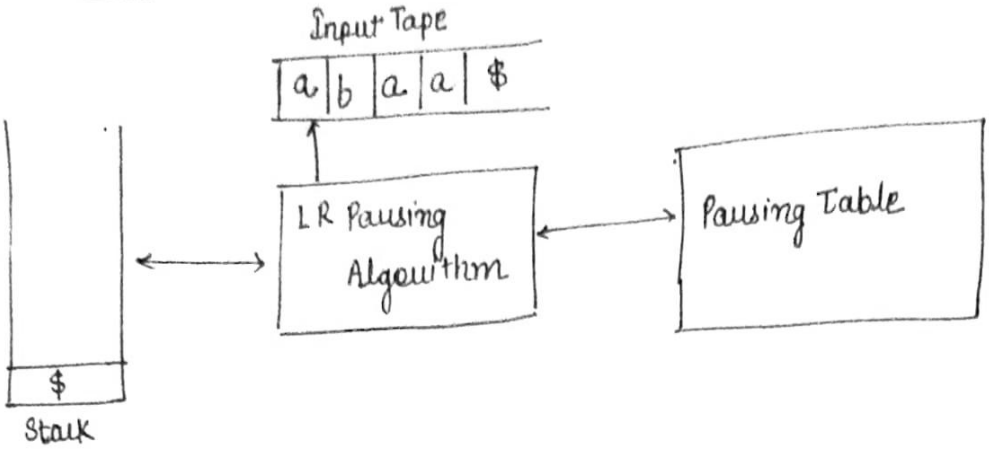
$E \to E+n / E*n / n$

$w = n+n*n$

no of Handles are - 3



LR Parsers



** ∀ Parsers Algorithm is same i.e Reverse of Right Most Derivation.
only parsing Table will be different for all algorithms.

LR(0) SLR(1)
CLR(1)
LALR(1)

CLR(1) is the Most powerful but Mostly used is LALR(1)
in terms of accepting more no of language.

Powerful Parsers LR(0) < SLR(1) < LALR(1) < CLR(1)

LR Parsers can parsers only Unambiguous Grammar.

Top Down & Bottom Up due to accepting more language.

LR(0) Parsers
- look ahead symbol
- Reverse of RMD
- Scan left to Right

Parsing Table of LR(0) Construction →

1. Construct Augmented Grammar
2. Construct canonical set of items using closure and goto function.
3. Construct DFA
4. Construct the P.T (~~Minimize~~ Minimize DFA to P.T)

P.T.O →

Construct LR(0) P.T for following Grammar

$A \to aA$
$A \to b$

Step 1: Construct Augmented Grammar

$A' \to A$
$A \to aA$
$A \to b$

Step 2: Make LR(0) item

$A' \to .A$

LR(0) item

Step 3: Using closure and Goto construct DFA

Closure $(A' \to .A) = A' \to .A$

↑ if after . we have variable then add prod. from grammar

$A \to .aA$
$\quad .b$

Eg-1 closure $(A \to .ab) = A \to .ab.$

Eg-2 closure $(A \to A.a) = A \to A.a$

Eg-3 closure $(A \to .A) = A \to .A$
$\quad A \to .aA$
$\quad A \to .b$

Eg-4 closure $(A \to A.A) = A \to A.aA$
$\quad A \to A.b$

Eg 5 - closure $(A \to AA.) = A \to AA.$

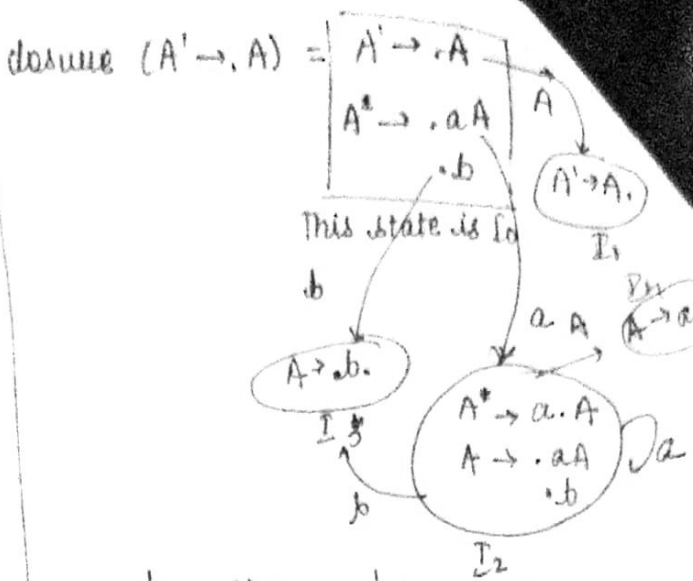
Goto :→ Goto $(A \to \alpha.x\beta \cdot x) =$ I/P

$A \to \alpha x.\beta$

Eg Goto $(A \to a.A \cdot A) = A \to aA.$ I/P

Goto $(A \to .aA \cdot a) = A \to a.A$
$= A \to a.aA$
$\to a.b$

closure $(A' \to .A) =$



	action			Goto part
	a	b	$	A
0	S2	S3		1
1			acc	
2	S2	S3		4
3	r2	r2	r2	
4	r1	r1	r1	

Parsing Table

Conflict → SR (Shift Reduce)
RR (Reduce Reduce)

LR(0) Item

Lect -14/18

- Reduce item: $A \to b.$
- Shift item: $A \to .aA$, $A \to .a$
- State item: $A \to a.A$

SR (Shift Reduce) → $A \to .a$, $A \to b.$

Both shift and reduce at same state

RR (Reduce Reduce) → $A \to a.$, $b.$

★ We don't have SS Conflict because we are making DFA.

★ Goto part doesn't create any conflict
↓
We don't have only state no entries

★ If any kind of conflict is there Grammar is not LR0

Que Construct LR(0) P.T for following grammar

$E \to T+E \mid T$ [I] $S \to .E$
$T \to a$ $E \to T+E$ (1) $\mid T$ (2)
$T \to a$ (3)

(II) Taking closure
closure($S \to .E$) =

	Action			Goto	
	+	a	$	E	T
0		S_3		1	2
1			accept		
2	S_4/r_2	r_1	r_2		
3	r_3	r_3	r_3		
4		S_3		5	2
5	r_1	r_1	r_1		

Since there is a conflict
∴ We get it not a LR(0)

LR(0)

1. Same DFA
2. Goto Entries are same in Table
3. Shift entries are same
4. Reduction Entry are different
(Blindly put in action part)

SLR(1) (Simple LR)

1. Same DFA
2. Goto entries are same
3. Shift entries are same
4. ←——→
5. Reduction entry are less as compared to LR(0)

	Action			Goto	
	+	a	$	E	T
0		S_3		1	2
1			acc		
2	S_4		r_2		
3	r_5		r_3		
4		S_3		5	2
5			r_1		

Since, No conflict ∴ Grammar is SLR(1)

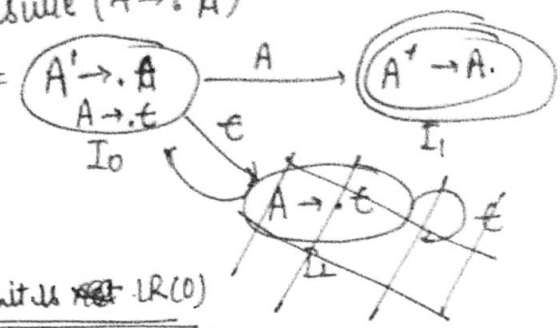
SLR(1) table for above problem
wherever there is a reduction just find follow of start symbol of that production like
$E \to T.$ follow(E) = $
$T \to a.$ follow(T) = + $
$E \to T+E.$ follow(E) = $

Que :→ $A \to \epsilon$ Construct LR(0) P.T

I ~~Accept~~ $A' \to A$, $A \to \epsilon$

II $A' \to .A$ LR(0) item

III Closure $(A' \to .A)$ =



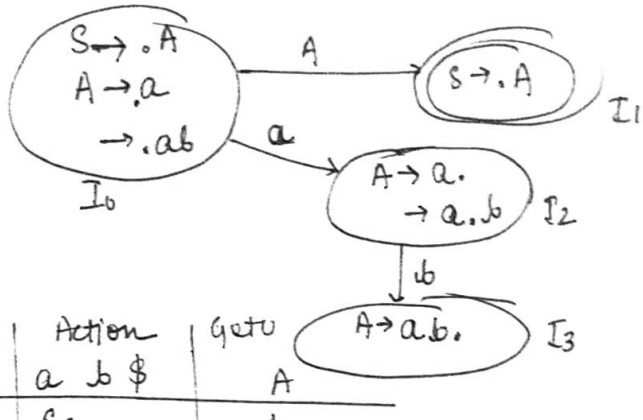
	Action		Goto
	ε	$	A
0	r0	r0	1
1		acc	
2			

Yes it is SLR(1)

if grammar is LR(0) then it is SLR(1)

Que $A \to a/ab$ Construct LR(0) & SLR(1)

I $S \to A$, $A \to a/ab$ II $S \to .A$ III Closure $(S \to .A)$ =



LR(0)	Action (1) a	b	$	Goto (2) A
0	S_2			1
1			acc	
2	r_2	S_3/r_4	r_4	
3	r_2	r_2	r_2	

∴ it is not LR(0)

SLR(1)	Action a	b	$	Goto A
0	S_2			1
1			acc	
2		S_3	r_4	
3			r_2	

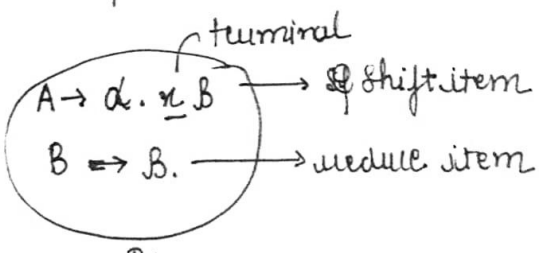
∴ it is SLR(1)

Que $S \to SS/a$

↓

Ambiguous

∴ No need to check for LR



→ In LR(0) Ii state we have SR conflict

→ In SLR(1) → first(x) ∩ follow(B) ≠ ϕ then SR conflict is there.

Eg → $A \to \alpha.$ → reduced, $B \to B.$ → reduced → In LR(0) we have RR conflict

↳ In SLR(1) follow(A) ∩ follow(B) ≠ ϕ we have RR conflict.

if a Grammar is ambiguous then

→ Then it is not LL(1)
" " " " LR(0)
" " " " SLR(1)
" " " " CLR(1)
" " " " LALR(1)

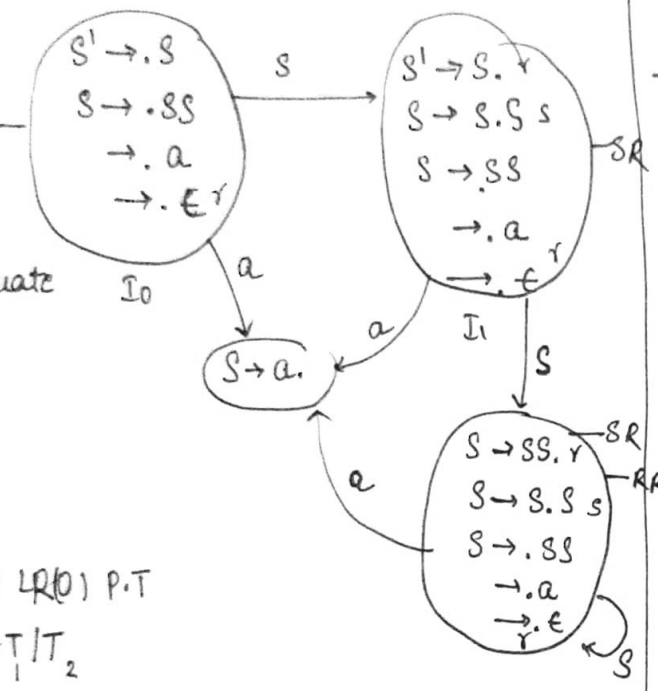
Que $S \to SS/a/\epsilon$ no of inadequate State and how many SR and RR conflict are there?

$S \to SS/a/\epsilon$

I $S' \to S$
$S \to SS/a/\epsilon$

II $S' \to .S$

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



3 Inadequate State
SR - 3
RR - 1

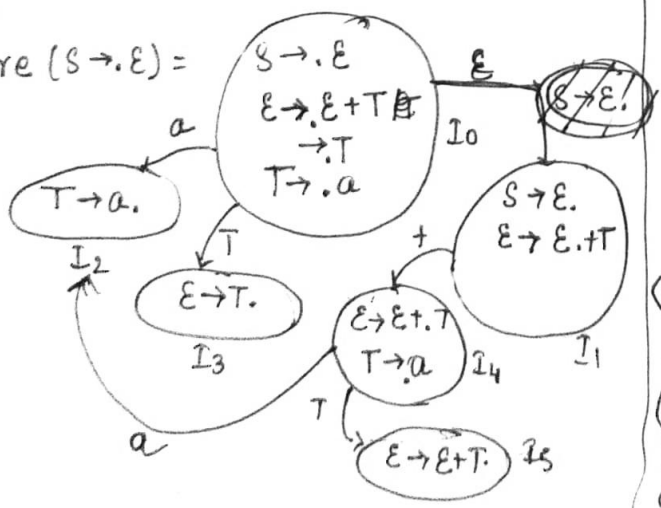
Que
Construct LR(0) P.T

$E \to E+T/T$ (1, 2)
$T \to a$ (3)

I $S \to E$

II $S \to .E$

III closure $(S \to .E)$ =



	Action			Goto	
	a	+	$	E	T
0	S_2			1	3
1		S_4	acc		
2	r_3	r_3	r_3		
3	r_2	r_2	r_2		
4	S_2				5
5	r_1	r_1	r_1		

it is LR(0) ∴ it is also SLR(1)

Que

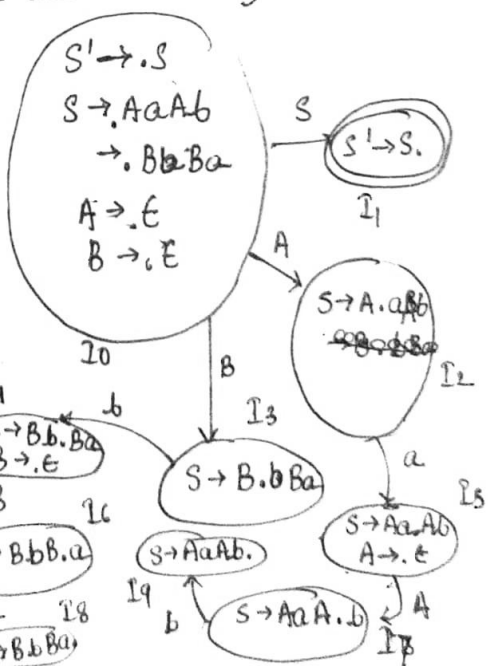
$S \to AaAb/BbBa$
$A \to \epsilon$
$B \to \epsilon$

	Action	Goto
	Shifted to next page →	

I $S' \to S$
$S \to AaAb/BbBa$
$A \to \epsilon$
$B \to \epsilon$

II $S' \to .S$

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



Due to RR Conflict it is not LR(0)

No of inadequate state = 1

Que Construct LR(0) & SLR(1)

$E \to E+T \mid T$

$T \to Tf \mid f$

$f \to f^* \mid a \mid b$

I $S \to E$

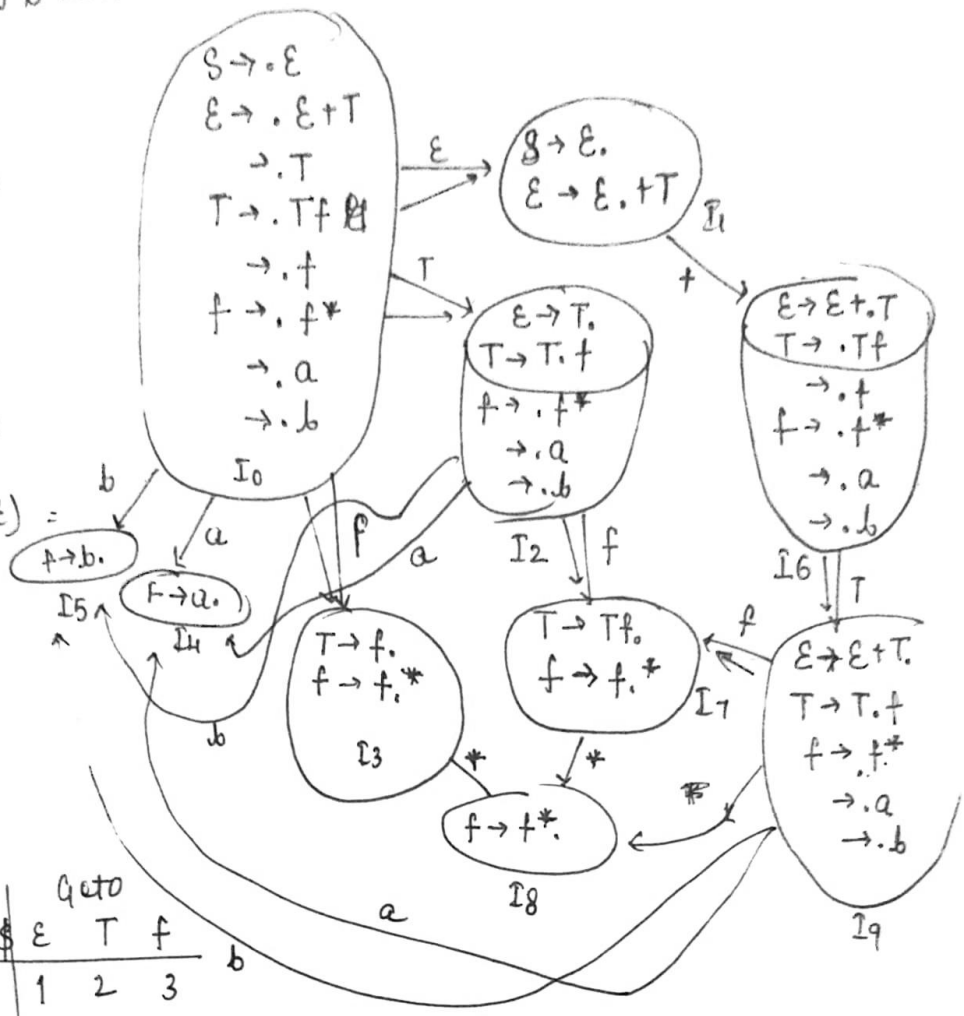
$E \to E+T_1 \mid T_2$

$T \to Tf_3 \mid f_4$

$f \to f^*_5 \mid a_6 \mid b_7$

II $S \to .E$

III Closure $(S \to .E)$ =



	Action					Goto		
	a	b	*	+	$	E	T	f
0	S_4	S_5				1	2	3
1				S_6	acc			
2	S_4/r_2	S_5/r_2	r_2	r_2	r_2			7
3	r_4	r_4	S_8/r_4	r_4	r_4			
4	r_6	r_6	r_6	r_6	r_6			
5	r_7	r_7	r_7	r_7	r_7			
6	S_4	S_5					9	
7	r_3	r_3	S_8/r_3	r_3	r_3			
8	r_5	r_5	r_5	r_5	r_5			
9	S_4/r_1	S_5/r_1	r_1	r_1	r_1			7

Not LR(0)

It is SLR(1)

CLR(1) / LR(1) and LALR(1)

we make LR1 item
⇓
LR(0) item + look ahead symbol

Eg → S → .AA , $
LR0 item, look ahead symbol

Construct CLR(1) / LR(1) for Grammar

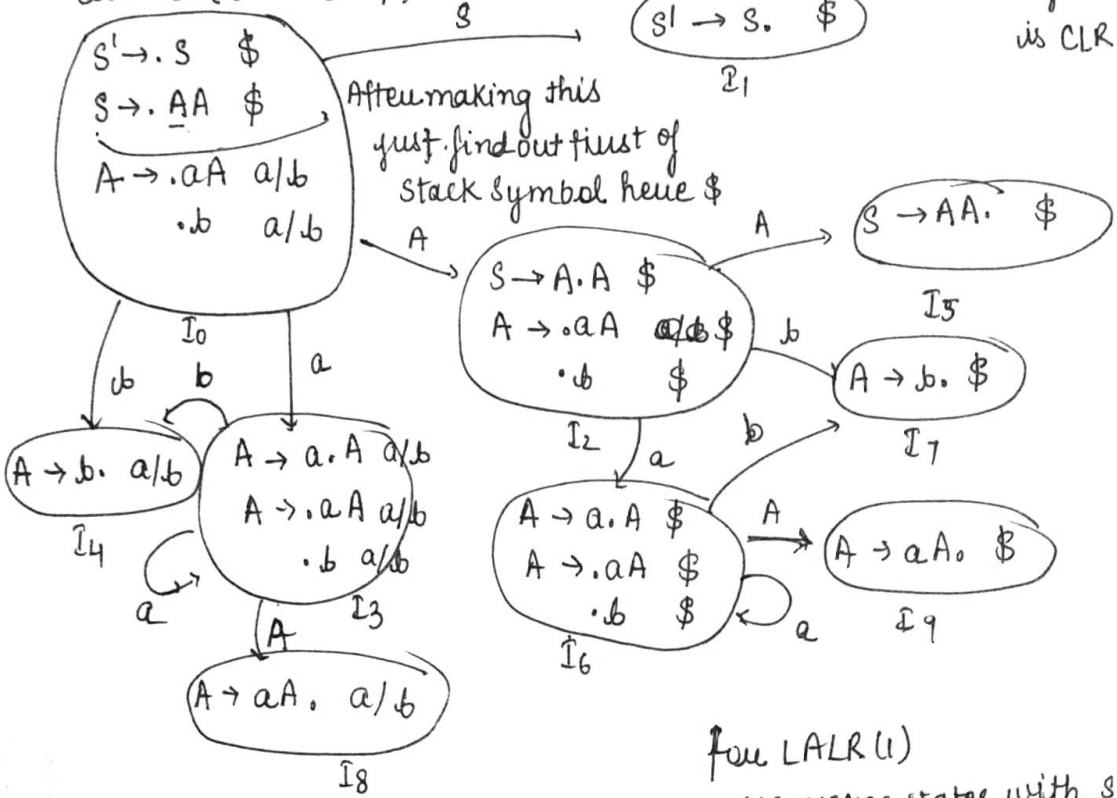
S → AA
A → aA/b

I S' → .S
S → AA
A → a A / b

II make LR(1) item and construct DFA using closure and Goto.

(S' → .S , $)
LR(0) item, LAS
LR(1) item

III closure (S' → .S $) =



	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		

Since there is no conflict this grammar is CLR(1)

For LALR(1)
we merge states with same production but diff look ahead symbol

A → aA a/b/$
A → .aA a/b/$
.b a/b/$

like I3 & I6 will merge

LALR(1) Table for problem

	Action			Goto	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
8					
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Since there is no conflict

∴ this is also LALR(1)

due find no of items in I0 in CLR(1) passing Table for

$E \to E+T/T$

$T \to T*f/f$

$f \to f*/a/b$

I → $S \to E$ $E \to E+T/T$ $T \to T*f/f$ $f \to f*/a/b$

II → ($S \to .E$ $)

III → closure ($S \to .E$ $) =

$S \to .E$	$	$E \to .E+T$	+/$	$T \to .T*f$	*/+/$
$E \to .E+T$	$/+/=	.T	+/$	.f	*/+/$
.T	$/+/=	$T \to .T*f$	+/$	$f \to .f*$	*/+/$
$T \to .T*f$	$/+/*	.f	+/$	.a	*/+/$
.f	$/+/*	$f \to .f*$	+/$	.b	*/+/$
$f \to .f*/a/b$	$/+/*	.a	+/$		
.a	$/+/*	.b	+/$		
.b	$/*/*				

I0

Que $S \to AaAb/BbBa$

$A \to \epsilon$

$B \to \epsilon$

No Conflict is there ∴ it is CLR(1) and LALR(1)

Construct LR(0)

$S \to AA$ ①

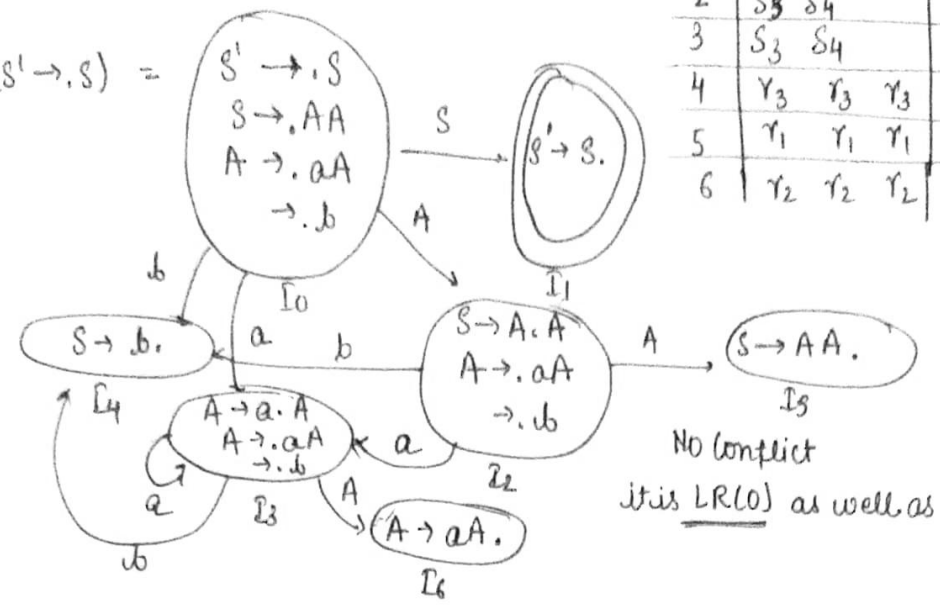
$A \to aA$ ② / b ③

I $S' \to S$

II $S' \to .S$

III Closure $(S' \to .S)$ =

	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		



No conflict

it is LR(0) as well as SLR(1)

	LR(0)	SLR(1)	CLR(1)	LALR(1)
no of state	n_0	n_2	n_3	n_4

relation ⇒ $n_1 = n_2 = n_4 \leq n_3$ **

Parsing table entries

- Shift Entry (Si) (terminals)
- State Entry (i) (Variable)
- Reduction Entry (Rs) ($A \to b.$)
- Acceptance Entry (acc)
- Blank / error

Reduction Entries $A \to B.$

LR0 → action part

SLR1 → follow(A)

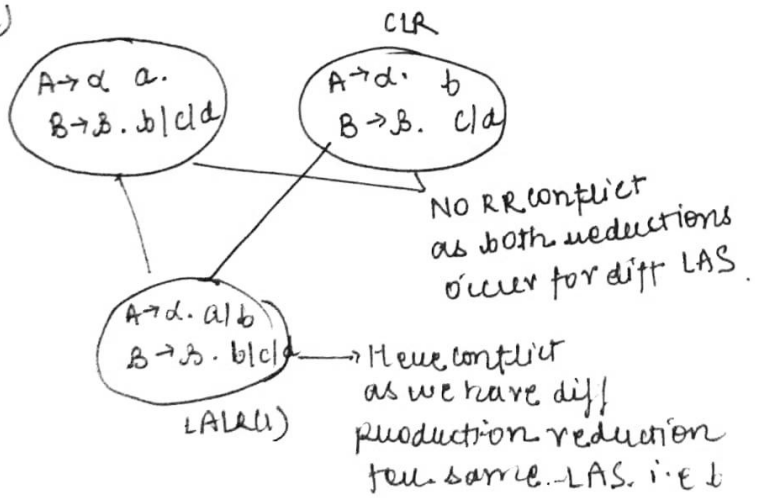
CLR1 → LOOK Ahead Symbol

LALR1 → LOOK ahead Symbol

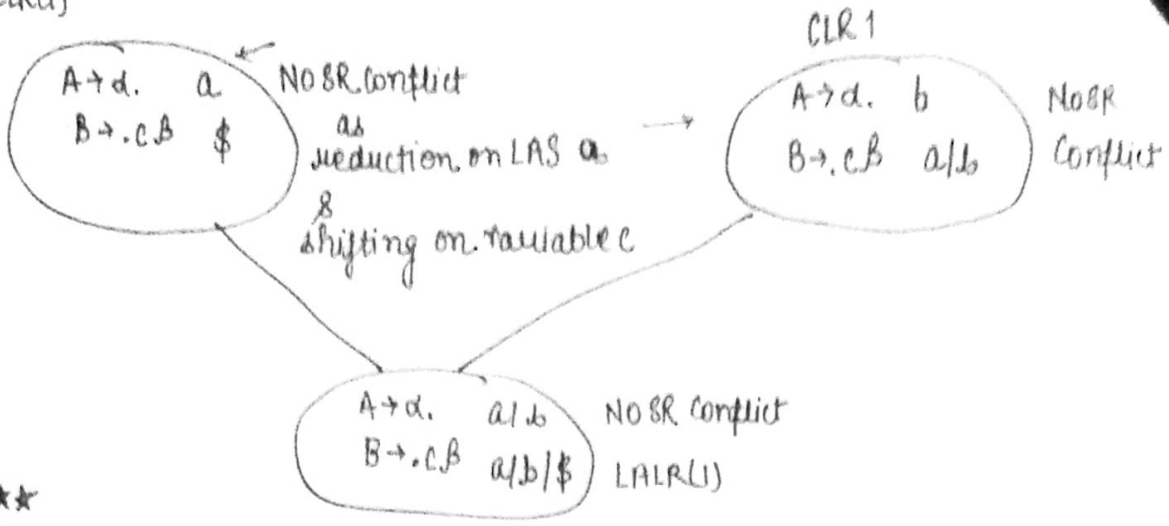
$A \to B.$ $\$$

1. No of shift entries is same in LR(0), SLR(1), LALR(1) < CLR(1)
2. " " Table " " " " " " " "
3. No of state " " " " " " "
5. Reduction entries in LR(0) ≥ SLR(1) ≥ LALR(1) ≥ CLR(1)

CLR(1)



CLR(1)

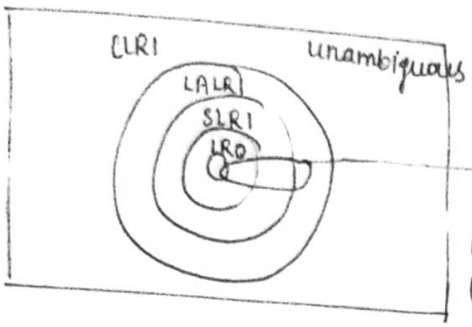


★★

If given grammar is CLR(1) but not LALR(1) then there must be RR conflict

★★

If LALR(1) don't have SR Conflict then CLR(1) also don't have SR Conflict.



Expressive power

CLR1 > LALR(1) > SLR(1) > LR(0)

① Every LL(1) is CLR(1)

② All grammars CLR(1), LALR(1), SLR(1), LL(1), LR(0) will be unambiguous

Algorithm for LR(0) :→

if S is Top of stack and a is look ahead symbol

1. if action [S, a] = Si then shift a and i and increment input pointer
2. if action [S, a] = rj and rj is α→β then pop 2x|β| from Top of stack and replace by α
3. if Sm-1 is the state below α then push goto [Sm-1, α]
4. if action [S, a] = acc then succesful completion of parsing
5. if action [S, a] = blank then error

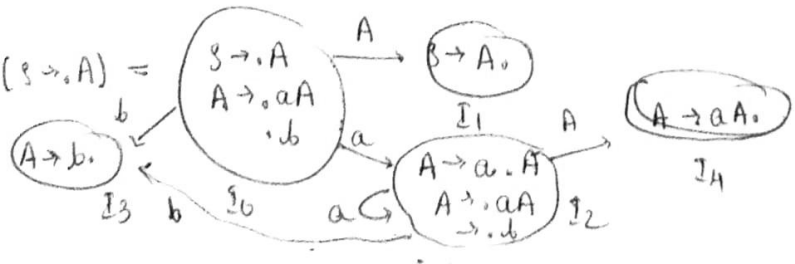
Construct LR(0) Parsing Table

A→aA|b

I A'→A

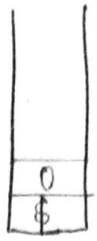
II S→.A

III closure (S→.A) =



	Action			Goto	
	a	b	$	S	A
0	S_2	S_3			1
1			acc		
2	S_2	S_3			4
3	r_2	r_2	r_2		
4	r_1	r_1	r_1		

Input → aab



input	Stack	
aab$	$0	0 on a gives S_2 (Shift a, 2)
aab$	$0a2	increment i/p pointer
aab$	$0a2a2	Shift (a, 2) = S_2, increment i/p
aab$	$0a2a2b3	Shift (b, 3) = S_3
		increment i/p
		3 on $ is reduction (r_2)
aab$	$0a2a2A	A→b pop (2 x length(b))
		Now if S_{n-1} is state below A we push
aab$	$0a2a2A4	$[S_{n-1}, A]$ i.e $[2, A] = 4$
		Now 4 on $ = r_1
aab$	$0a2A	A→aA length = 2
		∴ pop 2x2 items and replace with AA
	$0a2A	Now, push [2,A] = 4
aab$	$0a2A4	[4, $] = r_1 pop 2 x len(aA)
		= 4 items and replace with A
aab$	$0A	push [0, A] = 1
aab$	$0A1	Now [1, $] = acc
		∴ the string is accepted

YACC tool → LALR(1)

↳ SR Conflict

↳ it will ignore reduction in case of SR conflict

CLR(1)

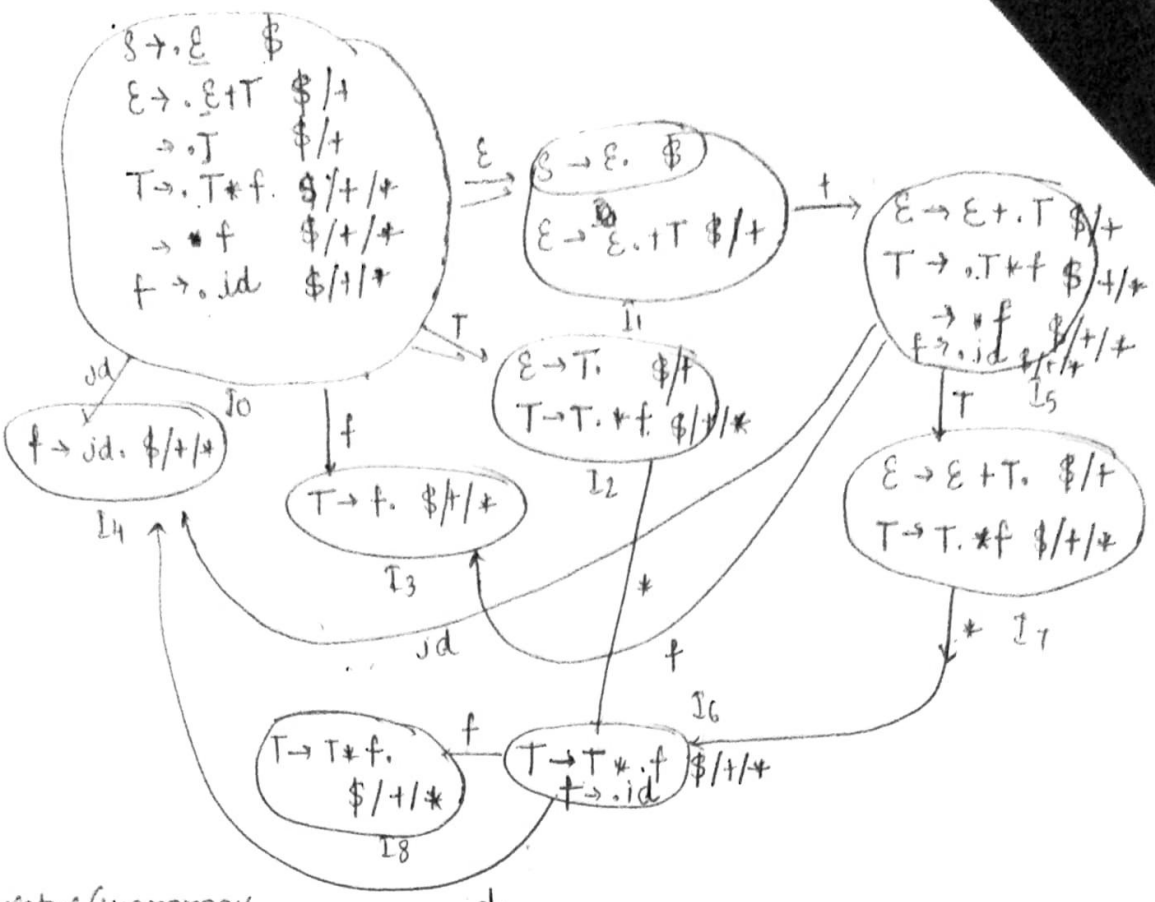
E → E+T / T

T → T*f / f

f → id

Closure(S → .E $)

⇒ P.T.O →



OPP → Operator Grammar
no null production
no 2 consequetive non-terminal / ~~grammar~~ variable in RHS of prod

Eg → A → BC
B → b
C → c
Not a operator Grammar

S → AaB
A → aA/b
B → bB/c
~~Not~~ operator Grammar

E → EAE/id
A → +/-/*
not a operator Grammar

E → E+E/ E-E/ E*E/id
operator Grammar

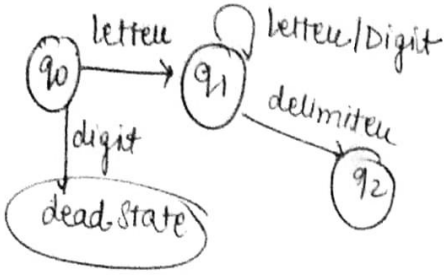
S → SAS/a
A → bSb/b
not a operator Grammar

⇓

S → SbSbS/ SbS/a

S → AaB
A → aA
B → bB/ϵ
yes it is operator grammar

PDA are used to design parsers or syntax analysers. And parser verifies syntax of text



lecture-19

SDT

Syntax directed translation

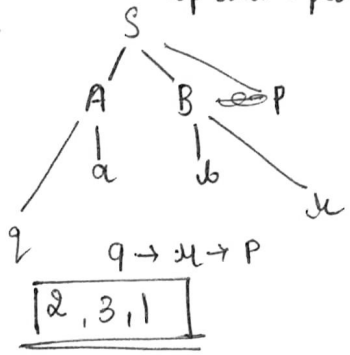
LA
↓
Syntax ← input CFG
↓ P.T
Semantic Analyser ← input SDT
↓
A.P.T

CFG + Semantic Action / Semantic Rule → SDT

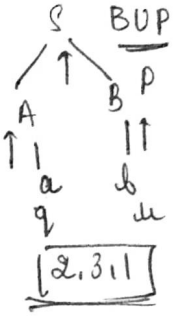
S → AB { printf(1) } → p
A → a { printf(2) } → q
B → b { printf(3) } → r
CFG — Semantic Action

w = ab

Top down parser



BUP



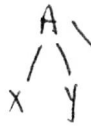
Application of SDT

1. Algebraic expression evaluation
2. Conversion infix to postfix (SDT construction)
3. Construct an SDT to convert infix to prefix exp.
4. " " " " " infix " infix exp.
5. Construct an SDT to convert parse tree to syntax tree.
6. " " " " " Binary number to Decimal number.
7. " " " " store data in symbol table.

Type of Attributes :-

① Synthesize attribute
② Inherited attribute.

A → XYZ

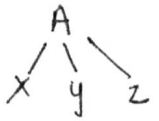


Derived from child.

A.S = f(X.S / Y.S / Z.S)

2 + 3 * 4;

A → XYZ



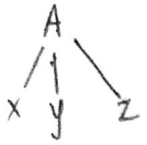
~~When~~ Derived either from parent or sibling

X.I = f(A.I / Y.I / Z.I)

Type of SDT Attribute

S attribute
↳ uses only synthesized attribute.

L attribute
↳ Uses both synthesized attribute and inherited attribute with restriction that inherited attribute can inherit value from left sibling



$A.S = f(X.S/Y.S/Z.S)$

$Y.S = f(A.S/X.S)$

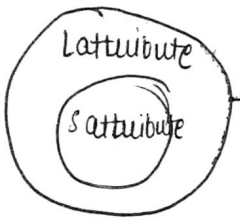
S

↳ action part at right most place
$S \rightarrow AB \{Pf(1)\}$

↳ It uses both Bottom up and top down approach

L

↳ Action part can be written anywhere in RHS
$S \rightarrow \{Pf(1)\} AB$
$S \rightarrow A \{Pf(1)\} B$
$S \rightarrow AB \{Pf(1)\}$
↳ Depth first search



→ Every S attribute is L attribute

Eg :→ $S_1 : A \rightarrow BC \{A.val = B.val + C.val\}$ → Synthesize att.

$S_2 : B \rightarrow DE \{B.val = D.val * E.val \text{ and } D.val = E.val\}$

↓ Synthesize att, ↓ Right inherited att.

∴ S_1 is both S & L
S_2 is not L
∴ option ③ $S_1 \rightarrow L$ & S_2 is not L

Application | ☆☆

1. Construct an SDT for algebraic expression evaluation

P.T.O →

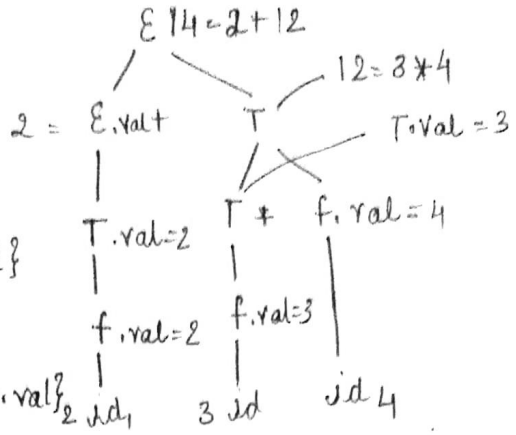
I/p → 2+3*4 O/p = 14

Grammar by Bottom up Parser

E → E+T/T

T → T*f/f

f → id



E → E+T { E.val = E.val + T.val }

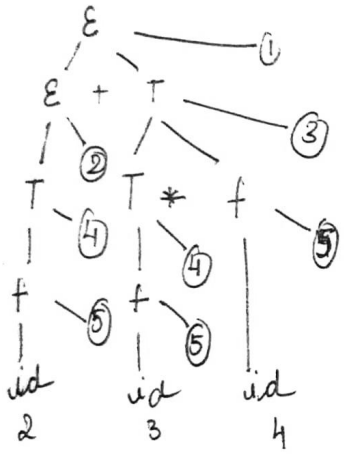
→ T { E.val = T.val }

T → T*f { T.val = T.val * f.val }

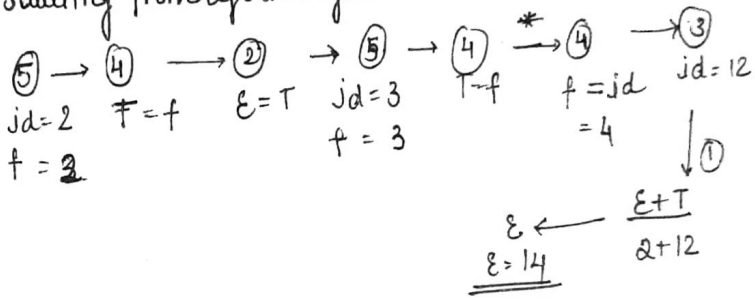
→ f { T.val = f.val }

f → id { f.val = id.val }

By Top Down Parser



Starting from left to Right



Lecture -20

SDT

↳ Cfg + Semantic Rule

S → AB { Pf(Ui) }

Cfg Action

SDT is also called SDD + Translation Scheme

↓ ⇓

Syntax Directed Definition Specification How do evaluate

what are attribute + Translation position

2+3*4

⇓

LA

↓

SA

↓ PT

SemA → Type checking

Type casting

Variable decl.

SDT

with attribute

$S \to AB$ {S.val = A.val + B.val}

S attribute — Depend on Children

L attribute — Depend on parent and left sibling

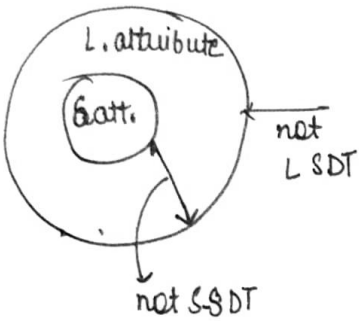
without attribute

$S \to AB$ {Pf(hi)}

Sattribute Lattribute

$A \to a$ {A.x = a.val} x here is S attribute

S – SDT → L – SDT



Eg) $S \to AB$ {B.x = A.y} → x → inherited

$A \to a$ {A.y = a.val} → S – SDT : y

$B \to b$ {B.z = b.val} → S – SDT : z

} Lattribute but not Sattribute SDT

Eg 3). $S \to AB$ {A.x = B.val} x :→ inherited

$A \to a$ {B.y = a.val} y :→ synthesized

$B \to b$ {B.z = b.val} z :→ synthesized

} Not Lattribute Not Sattribute SDT

Eg 4) $\varepsilon \to$ {Pf(1)} ε T → Lattribute SDT

$\varepsilon \to$ id {pf(id)} → Lattribute SDT

at leftmost

rightmost

5). $S \to AB$ {pf(1)} – ①

$A \to a$ {pf(2)} – ②

$B \to b$ {pf(3)} – ③

} S attribute SDT → LSDT

① if it is LSDT we will perform Depth first search with with left to write scanning

② Topological Order

③ L attribute evaluation

Sattribute

↳ we can apply ~~RRMD~~ BUP and perform RRMD or we can apply Depth LtoR scan

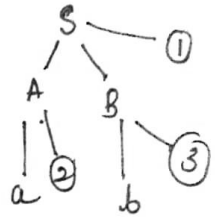
if nothing is given without attribute

Then apply Depth scan Left to Right

$S \to AB$ {Pf(1)}
$A \to a$ {Pf(2)}
$B \to b$ {Pf(3)}

] S-SDT as well L SDT

1) L attribute SDT - Depth Left to Right
2) LMD
3) Reverse of LMD
4) RMD
5) RRMD



1) $2 \to 3 \to 1$
2) ~~2 3 1~~
3)

So it will be $1 \to 3, \to 2$

5) RRMD
$2 \to 3 \to 1$

2)
S
1
AB
1
aB
1
ab

for 2) $1 \to 2 \to 3$

4)
S
1
AB
1
Ab
1
ab

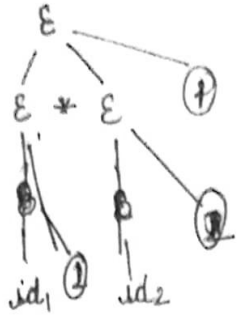
3) RLMD
$3 \to 2 \to 1$

Lecture 21

P.T.O →

Que $E \to E * E$ {pf (*)} — ① $E \to id$ {pf ("id")} — ② } SSL att SDT

$w = id_1 * id_2$



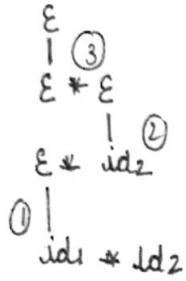
The output will be

2, 2, 1

Translation order

id_1

Using RRMD

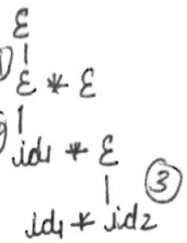


we will print

id1 id2 *

Using TDL

* id1 id2



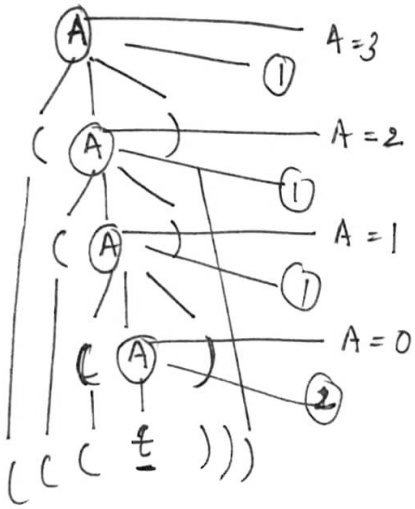
Using Que Construct an SDT to count Balanced ~~not Balanced~~ Parenthesis

I/p → ((( )))

o/p = 3

Grammar ⇒ $A \to (A)$ {A.count = A.count + 1} — ①

$A \to \epsilon$ {A.count = 0} — ②



∴ final ans will be

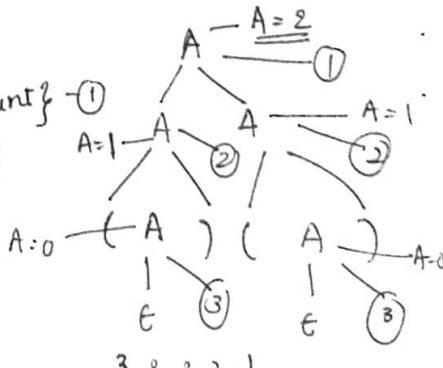
A=3

I/p = ( ) ( )

$A \to AA$ {A.count = A.count + A.count} — ①

$A \to (A)$ {A.count = A.count + 1} — ②

$A \to \epsilon$ {A.count = 0} — ③



final A = 2

3, 2, 3, 2, 1

Que Construct an SDT to convert from infix to ~~postfix~~ prefix

***

I/p = 2+3*4

postfix = 234*+

O/p = ~~43*2+~~ +2*34

Grammar - $E \to E+T$ {printf("+")} ①

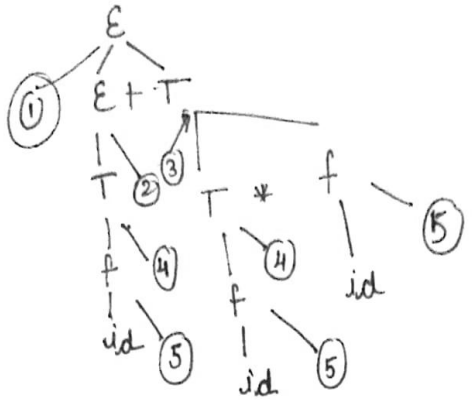
$\to T$ {} ②

$T \to T*f$ {printf("*")} ③

$\to f$ {} ④

$f \to id$ {printf(id)} ⑤

} L attribute SDT



∴ final ans

+id1 * id2 id3

1 → 5 → 4 → 2 → 3 → 5 → 4 → 5

**

For postfix

I/p = 2+3*4

O/p = 234*+

Grammar

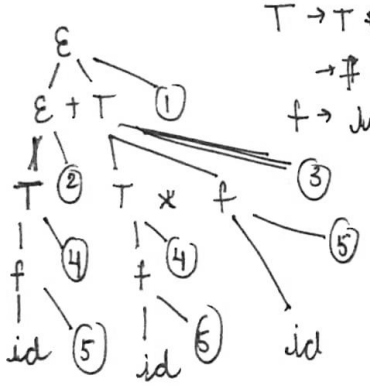
$E \to E+T$ {print("+")}

$\to T$ {}

$T \to T*f$ {printf("*")}

$\to f$ {}

$f \to id$ {printf("id")}



5 → 4 → 2 → 5 → 4 → 5 → 3 → 1

id1 id2 id3 * +

final ans

with attributes at statements

{Eval = Eval Tval +}

{Eval = Tval}

{Tval = Tval fval *}

{Tval = fval}

{fval = id.val}

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

$\to T$ {}

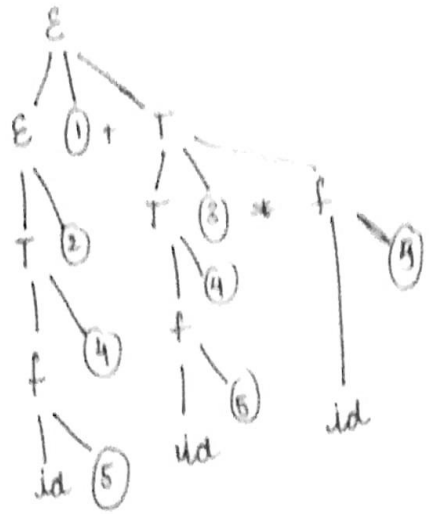
$T \to T$ {pf *} * f

$\to f$ {}

$f \to id$ {pf id}

I/p = 2+3*4

O/p = ?



O/p → 5 → 4 → 2 → 1 → 5 → 4 → 3 → 5

$id_1 + id_2 * id_3$

[2 + 3 * 4]

[Infix to Infix]

***

Lecture - 22

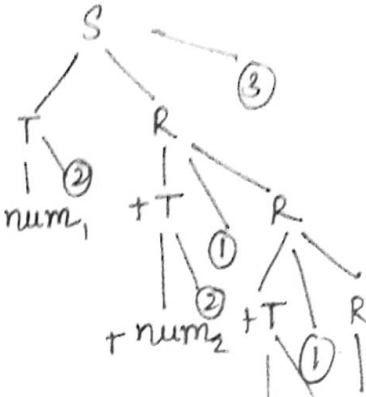
Que1 S → TR ③

R → +T { print('+'); } R/∈ - ①

T → num. { print(num.val); } - ②

I/p = 9+5+2

O/p = ?



2 → 2 → 1 → 2 → 1 → 3

9 5 + 2 + final output

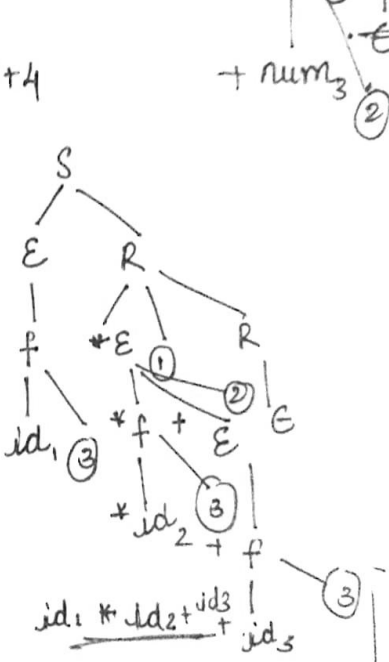
Que2 S → ER

R → * E { print("*"); } R/∈ - ①

E → f + E { print("+"); } / f - ②

f → (S) / id { print(id.value); } - ③

I/p = 2*3+4



3 → 3 → 3 → 2 → 1

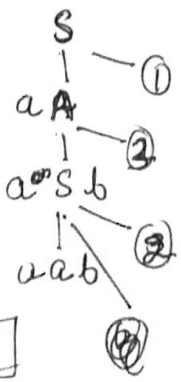
2 3 4 + * final output

Que3 S → aA { print 1 }

S → a { print 2 }

A → Sb { print 3 }

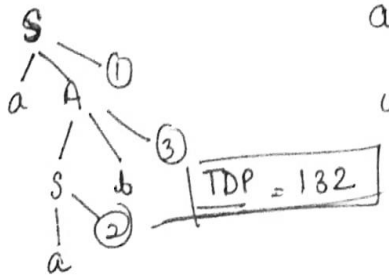
I/p - aab



2 → 3 → 1

final Output

231



TDP = 132

Q4 $A \to PS$ $A \to XY$

Rule 1 $P.i = A.i + 2$

$Q.i = P.i + A.i$ & $A.s = P.s + Q.s$ } L attribute

Rule 2 $X.i = A.i + Y$ & $Y.i = X.s + A.i$ } Not L attribute also Not S

Only Rule 1 is L attributed

Que 5) $E \to E_1 \# T$ { $E.val = E_1.val * T.val$ } -(1)

$\to T$ { $E.value = T.val$ } -(2)

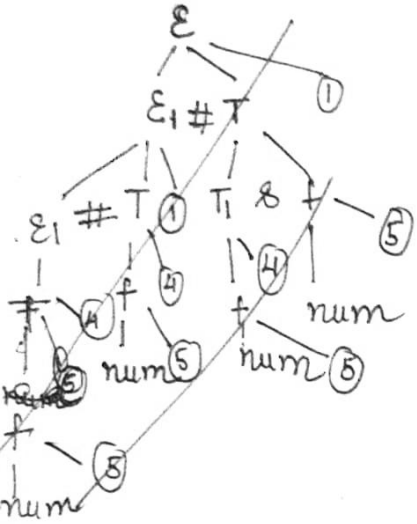
$T \to T_1 \& f$ { $T.val = T_1.value + f.value$ } -(3)

$\to f$ { $T.value = f.value$ } -(4)

$f \to num$ { $f.value = num.value$ } -(5)

I/p = 2#3&5#6&4

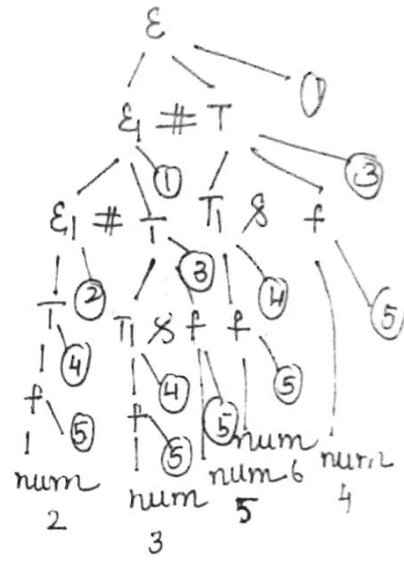
O/p = ?



5→4→5→1→5→4

5→4→2→5→4→5→3→1

→5→4→5→3→1



2 * 8 = 16

10 × 16 = 160

final ans

Que 6 Operator Grammar

$P \to QR$ ⟶ This violates OG as no 2 non-terminal should be adjacent

$P \to QSR$

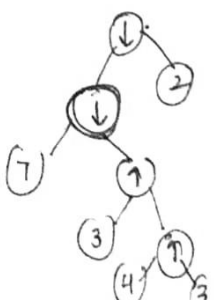
$P \to \epsilon$ ⟶ not allowed

$P \to QtRu$

Que 7) Consider 2 binary operator ↑ and ↓ with the precedence of operator ↓ being lower than that of the ↑ operator

↑ → right Associative 7↓3↑4↑3↓2

↓ → left Associative



Que In a Bottom up evaluation of a syntax directed definition inherited attribute can be evaluated

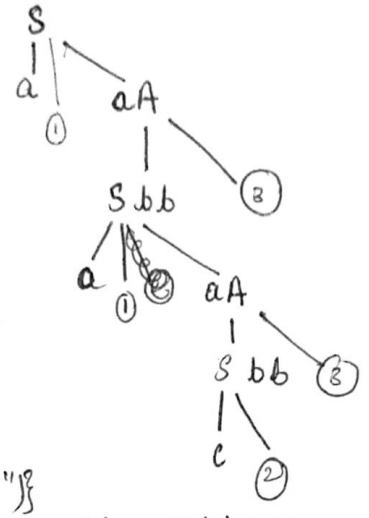
→ only if definition has synthesized attribute

Que
$S \to a$ {print 1} aA - (1)
$S' \to C$ {print 2} - (2)
$A \to Sbb$ {print 3} - (3)

} L attribute

I/p = aaaacbbbb
O/p = 23131

BUP



23131

2 → 3 → 1 → 3 → 1

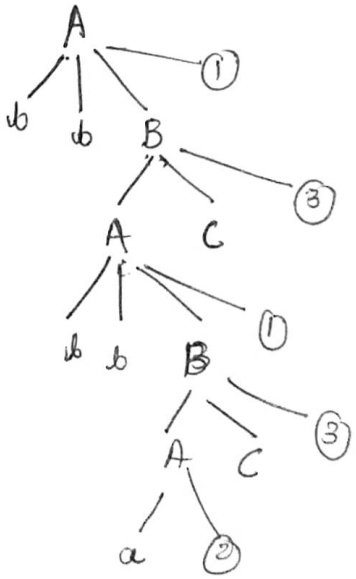
using L attribute eval
1 → 1 → 2 → 3 → 3

Que
$A \to bbb$ {printf("+")}
$A \to a$ {pf("x")}
$B \to Ac$ {pf("-")}

I/p = bbbbacc

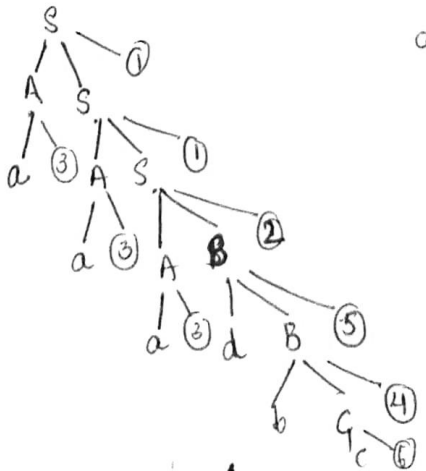
2 → 3 → 1 → 3 → 1

x - + x +



Que
$S \to AS$ {pf("1")}
$S \to AB$ " 2
$A \to a$ " 3
$B \to bc$ " 4
$B \to dB$ " 5
$C \to c$ " 6

w = aaadbc



3 → 3 → 3 → 4 → 5 → 2 → 1

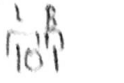
3345211

3345211

Construct an SDT to convert given Binary no into decimal.

Binary no : 101

Decimal no : 5



$2 \times L.1 \; B = 2 \times 2 + 1 = 5$

$2 \times 5 + 0$

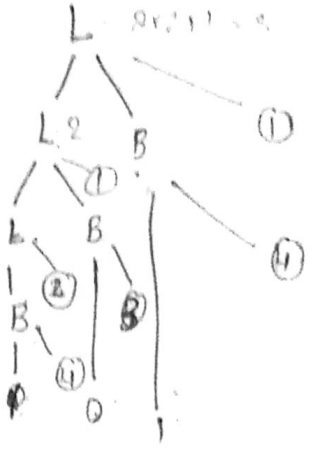
$2 \times L.decimal + B.decimal = 5 \qquad = 10$

$L \to LB \; \{ L.Dval = 2 * L.Dval + B.dval \}$

$\to B \; \{ L.Dval = B.Dval \}$

$B \to 0 \; \{ B.dval = 0 \}$

$\to 1 \quad \{ B.dval = 1 \}$



No of Bit Counting

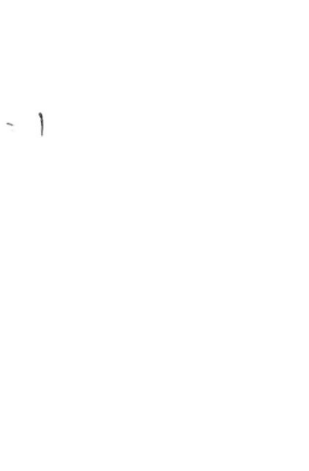
$4 \to 2 \to 3 \to 1 \to 4 \to 1$

$L \to LB \quad \{ L.x = L.x + B.x \}$

$\to B \quad \{ L.x = B.x \}$

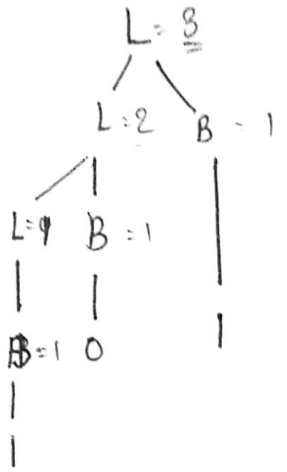
$B \to 0 \quad \{ B.x = 1 \}$

$\to 1 \quad \{ B.x = 1 \}$



I/p = 101

O/p $\Rightarrow$ L = 3



I/p = 101.101 $\to \frac{5}{8} = 0.625$

O/p $\Rightarrow$ 5 + 0.625

= 5.625

$\Rightarrow$ Grammar $\to$ $S \to L.L \; \{ S.dval = L_1.dval + L_2.dval / 2^{L_2}.nval \}$

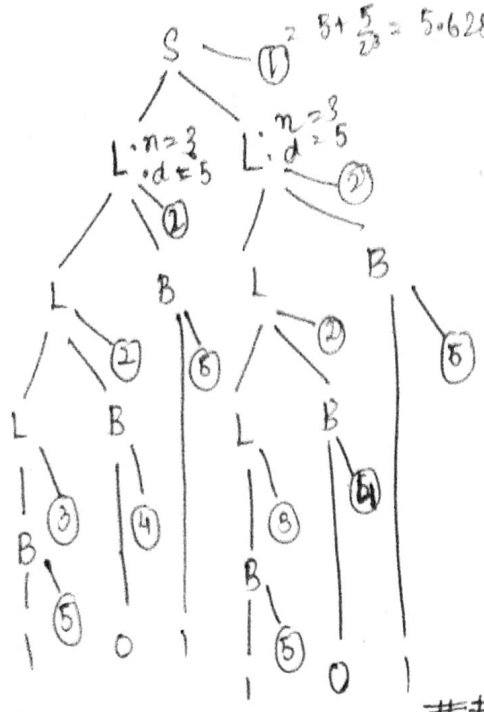
$L \to LB \; \{ L.nval = L.nval + B.nval, \; L.dval = 2 \times L.dval + B.dval \}$

$\to B \; \{ L.nval = B.nval, \; L.dval = B.nval \}$

$B \to 0 \; \{ B.nval = 0, \; B.dval = 0 \}$

$\to 1 \; \{ B.nval = 1, \; B.dval = 1 \}$

Tree $\to$



$5 + \frac{5}{2^3} = 5.625$

5→3→4→2→5→2→5→3→4→2→5→2→1

Total no of Terminal Count

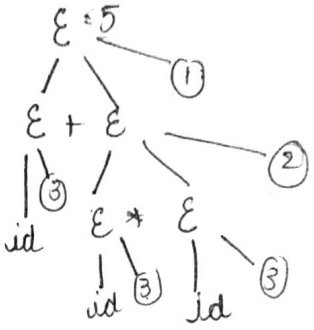
Que $\mathcal{E} \to \mathcal{E}_1 + \mathcal{E}$ $\{\mathcal{E}.x = \mathcal{E}_1.x + \mathcal{E}_2.x + 1\}$

$\mathcal{E} \to \mathcal{E} * \mathcal{E}$ $\{\mathcal{E}.x = \mathcal{E}_1.x + \mathcal{E}_2.x + 1\}$

$\mathcal{E} \to id$ $\{\mathcal{E}.x = 1\}$

I/p = 2+3*4

O/p = 5



Que $\mathcal{E} \to \mathcal{E}_1 + \mathcal{E}_2$ $\{\mathcal{E}.x = \mathcal{E}_1.x + \mathcal{E}_2.x + 1\}$

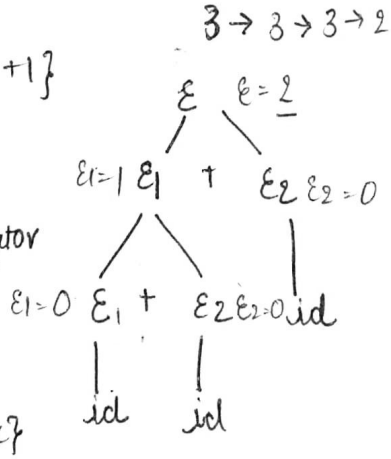
$\mathcal{E} \to id$ $\{\mathcal{E}.x = 0\}$

I/p = 2+3+4

Count no of operator

O/p = 2

3→3→3→2→1



Que $\mathcal{E} \to \mathcal{E}_1 + \mathcal{E}_2$ $\{\mathcal{E}.x = \mathcal{E}_1.x + \mathcal{E}_2.x\}$

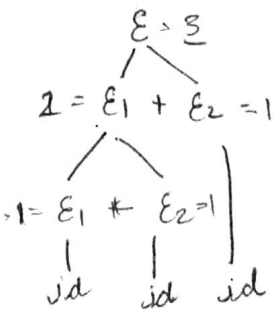
$\mathcal{E} \to \mathcal{E}_1 * \mathcal{E}_2$ $\{\mathcal{E}.x = \mathcal{E}_1.x + \mathcal{E}_2.x\}$

$\mathcal{E} \to id$ $\{\mathcal{E}.x = 1\}$

I/p = 2*3+4

O/p = 3

no of operands

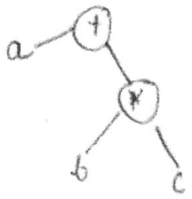


Syntax Tree

→ all nodes are terminal

I/p = a+b*c

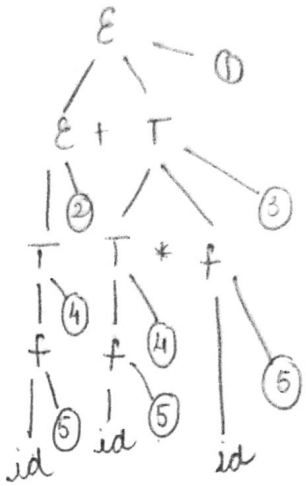
O/p =



Parse Tree (abstract syntax trees)

→ nodes can be both terminal or non terminal

E → E + T {E.ptr = mknode (E.ptr + T.ptr)}
→ T {E.ptr = T.ptr}
T → T * f {T.ptr = mk node (T.ptr * f.ptr)}
→ f {T.ptr = f.ptr}
f → id {f.ptr = mk node (null, id, Now)}

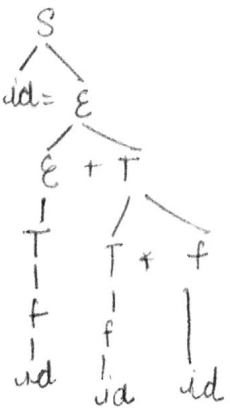


Intermediate Code Generator

Intermediate Code
- Linear form
  - Postfix
  - 3Add Code
  - Static Single Assignment (SSA)
- Nonlinear form (Trees)
  - Syntax Tree
  - Directed Acyclic graph (DAG)

Code
↓ LA
Stream of Token
↓ SA
Parse Tree
↓ Sem A
SVT (Decorated parse tree)
↓ ICG
Intermediate Code/Representation
→ Machine Independent Code

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



ICG → Postfix
→ 3 Add Code
→ SSA
→ Syntax tree
→ DAG

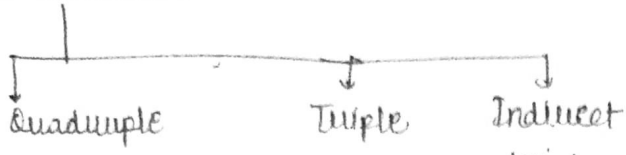
Postfix ICG

$x = a+b*c;$

$xabc*+=$

Optimized postfix $abc*+$

3add Code



3add.code: In a single line we have atmost 3 address

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

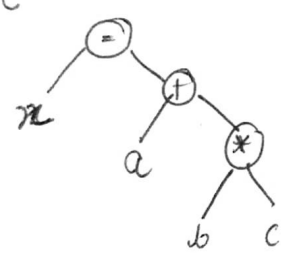
$t_1 = b*c;$
$x = a+t_1;$

⇒ further optimize

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

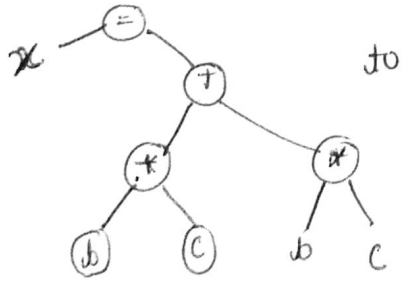
Syntax tree

$x = a+b*c$

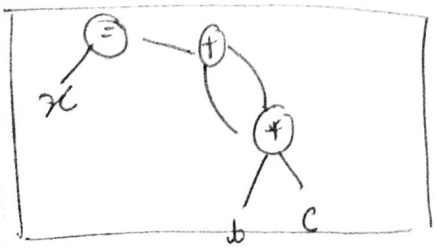


DAG : to eliminate common subexpression

$x = b*c + b*c$



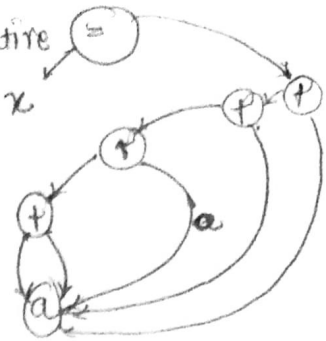
to reduce the repition of $b*c$



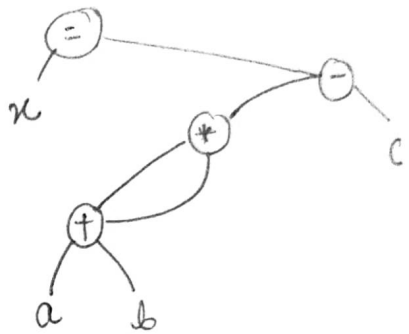
$x = a+a+a+a+a$

acc to C Rule

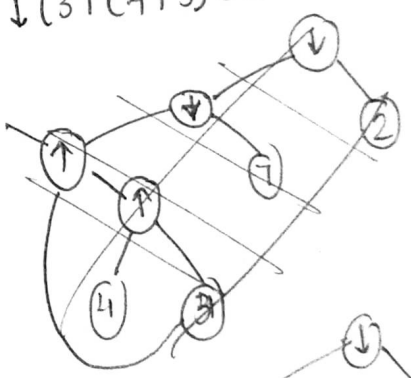
(+) is left associative

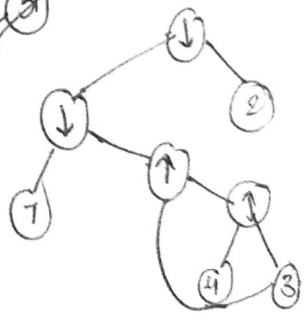


$x = (a+b) * (a+b) - c$



$7 \downarrow (3 \uparrow (4 \uparrow 3) \downarrow 2$





$x = a + b * c$ — 3 add code

```
t1 = b*c;
t2 = a + t1;
x  = t2
```

$S \to id = E$ { gen(id.val = E.val) }

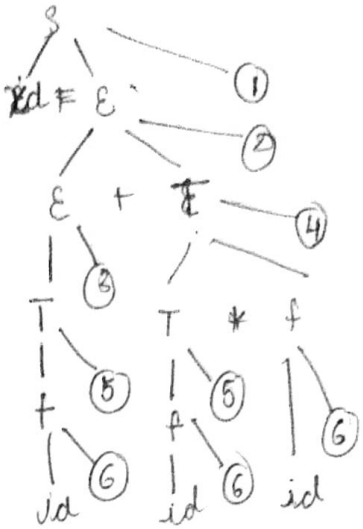
$E \to E + E$ { t2 = newtemp(); gen(t2 = E1.val + T1.val) E.val = t2 }

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

$T \to T * f$ { t1 = newtemp(); gen(t1 = T1.val * f.val) T.val = t1 }

$\to f$ { T.val = f.val }

$f \to id$ { f.val = id.val }



6 → 5 → 3 → 6 → 5 → 6 → 4 → 2 → 1

Que

$S \to id = E$ { gen(id.place = E.place) } -①

$E \to E_1 + E_2$ { t = newtemp(); gen(t = E1.place + E2.place); E.place = t } -②

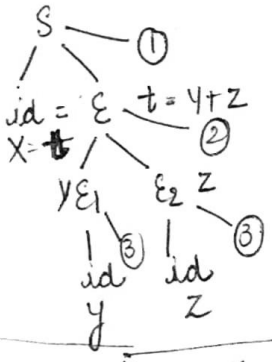
$E \to id$ { E.place = id.place } -③

$x = y + z$

3 → 3 → 2 → 1

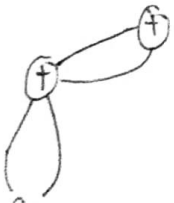
option B

```
t = y + z
x = t
```



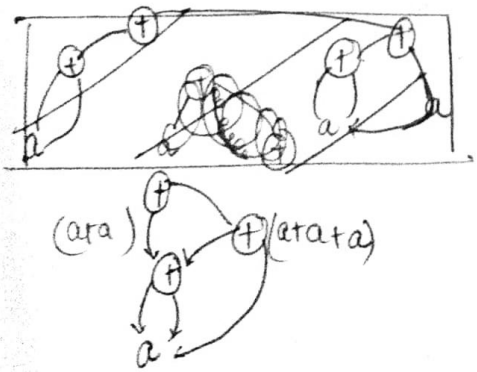
Que

Construct DAG for (a+a) + (a+a)



no of nodes = 3

Que Construct DAG for (a+a) + (a+a+a)



Que 3 address code - in a statement we have at most 3 address

1. $x = y$ op z
2. $x =$ op y
3. $x = y$
4. $x = *y$
5. $x = \&y$
6. $x = a[i]$ ($*(a+i)$)
7. $a[i] = x$
8. goto L (Unconditional Jump)
9. if $x < y$ goto L (conditional Jump)
10. $y = f(x_1, x_2, \ldots, x_n)$ X
11. $x = a[i, j]$ X

not 3 add code

Que Construct 3 add code for the following C statement

```
if (a>b)
printf("hi");
else
printf("bye");
```

```
100. if (a>b) goto 103
101 printf("bye");
102. goto 104
103. printf("hi");
104 Exit/End
```

Que if (a>b) then t=1 else s=0

```
i : if (a>b) goto i+3
i+1: s=0
i+2: goto i+4
i+3: t=1
i+4: Exit/End
```

Que if a<b and c<d then t=1 else s=0

```
i : if a<b goto i+2
i+1 : goto i+3
i+2 : if c<d goto i+5
i+3 : s=0
i+4 : goto i+6
i+5 : t=1
i+6 : Exit/End
```

Que

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

```
100. i=0
101. if (i<10) goto 103
102. goto 108
103. t1 = b*c;
104. t2 = a+t1;
105. x = t2;
106. i = i+1;
107. goto 101
108. Exit
```

Quadruple - we have 4 columns

x = a+b*c;

3 add code uses:→

1. Quadruple
2. tuple
3. Indirect tuple

```
t1 = b*c;
t2 = a+t1;
x = t1;
```

result	op1	op2	operator
t1	b	c	*
t2	a	t1	+
x	-	t2	=

Advantage:→ ① reuse the values.
② Time saving.

Disadvantage ① takes more space in form of result column

Tuiple - we have 5 columns

	operand1	operand2	operator
1000	b	c	*
2000	a	1000	+
3000	-	2000	=

add

Advantage - save space

Disadvantage - due computation of result
② more time consumption

Indirect Triple

We can save the result address in another address to improve reusability

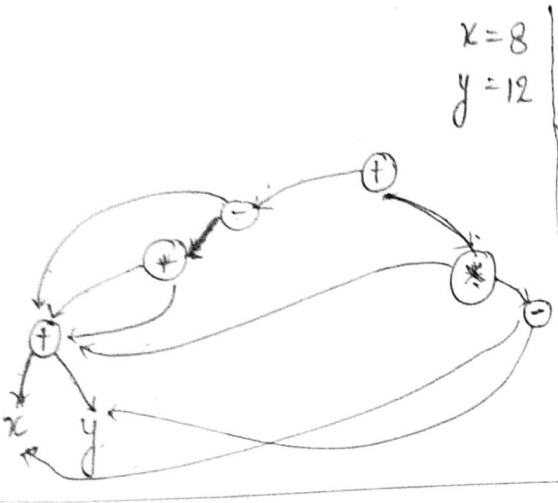
5000	1000
6000	2000
7000	3000

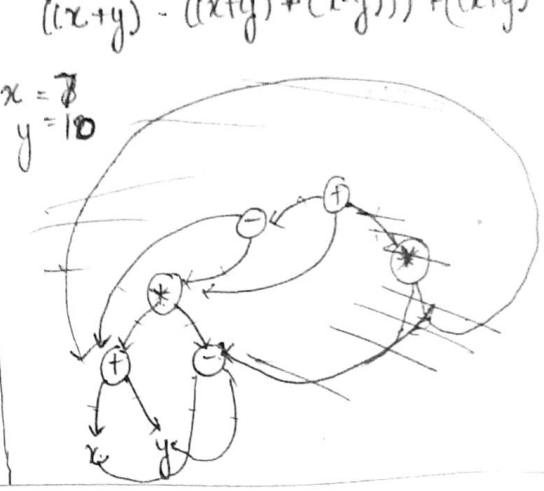
less space

Lecture 27

Construct a DAG for following statement and find x & y where x is no of nodes and y is no of edges.

$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$





Consider 3 add code :→ find out min no. temp variable in optimized 3add code

```
t1 = a+b;
t2 = c+d;
t3 = t1+t2;
t4 = t2+t3;
t5 = t4+t3;
```

```
a = a+b;
c = c+d;
t1 = a*c;
c = c+c;
t3 = t1+c;
```

min = 2 var (temp)

```
a = a+b;
c = c+d;
t1 = a*c;
c = c+c;
t2 = t1+c;
```

total var = 6

SSA - variable should be defined only once in LHS

Que Consider :→

```
x1 = u-t;
y1 = x1*v;
x2 = y1+w;
y2 = t-z;
y3 = x2*y2;
```

(min no var used to convert it into SSA)

total distinct var = 10

$x_1, x_2, y_1, y_2, y_3, u, t, v, w, z$

Lecture - 28

Code Optimization

Que SSA

$p = a - b$
$q = p * c$
$p = u * v$
$q = p + q$

$\Rightarrow$

$p_1 = a - b$
$q_1 = p_1 * c$
$p_2 = u * v$
$q_2 = p_2 + q_1$

Option B

Que $x = (a+b) * (a+b) + c$

$t = a + b$
$u = t * t$
$x = t + u$

$\Rightarrow$

$t = a + b$
$u = t * t$
$x = u + c$

$\Rightarrow$

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

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

Que $x = (y+z) * y + z$

$t = y + z$
$y = t * y$
$x = y + z$

4 variable $\Rightarrow$

$t = y + z$
$y = t * y$
$y = y + z$

Que $q + u/3 + 8 - t * 5 + u * v/w$

$t_1 = u/3$
$t_2 = v/w$
$t_3 = u * t_2$
$t_4 = t * 5$
$t_5 = q + t_1$
$t_6 = t_5 + 8$
$t_7 = t_3 + t_4$
$t_8 = t_6 - t_7$

Total 8 temp vari are required.

Que
```
x = 10;
a = a * x;
c = c - a;
```

	Operand 1	Op 2	Operator
1000	x	10	=
2000	a	1000	*
3000	c	2000	-

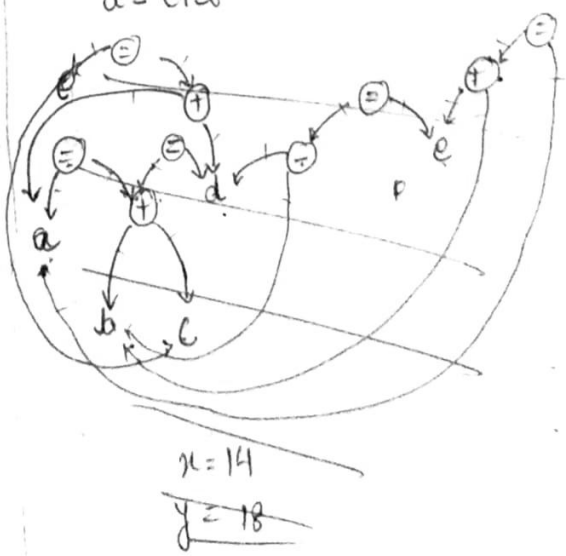
Indirect triple

5000	1000
6000	2000
7000	3000

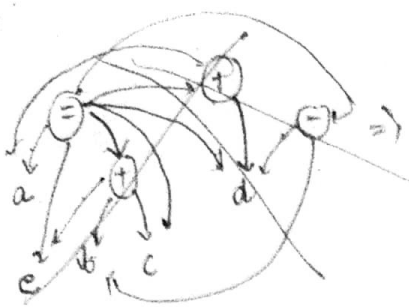
Que DAG

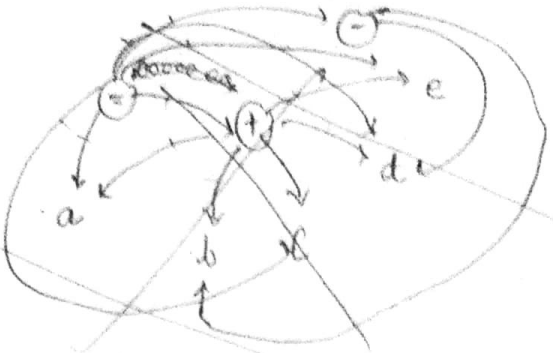
$a = b + c$
$c = a + d$
$d = b + c$
$e = d - b$
$a = e + b$

x = ? vertex
y = ? edge



$x = 14$
$y = 18$





wrong

$x = 9 \quad y = 13$

$x = 8$

$y = 6 + 2 + 5 = \underline{13}$

$a = b + c$

$c = a + d$

$d = b + c$

$e = d - b$

$a = e + b$

$\boxed{\begin{array}{l} a = d \\ e = c \end{array}}$

$a = e + b$

$a = d - b + b$

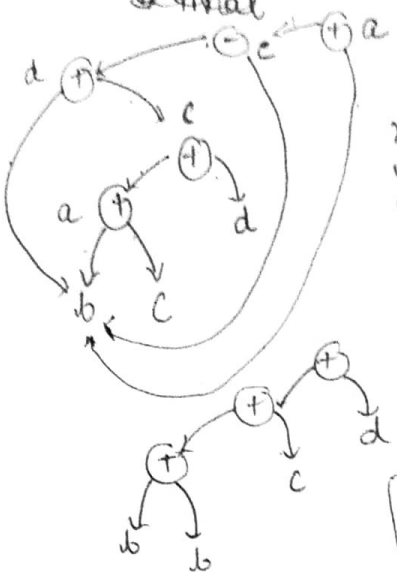
$a = d$

$a = b + c$

$a = b + a + d$

$a = \underline{b + b + c + d}$

Dt final

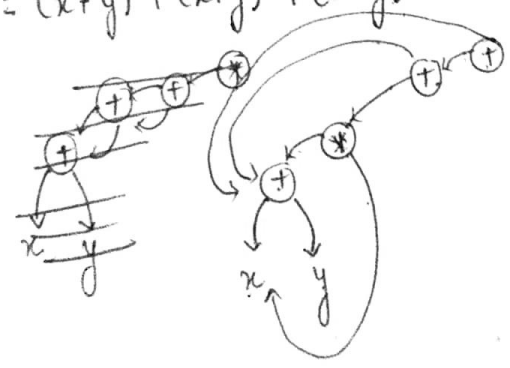


$x = 8$

$y = \underline{10}$

Construct DAG and find min no of node and edges

$a = (x+y) + (x+y) + (x+y) + x$



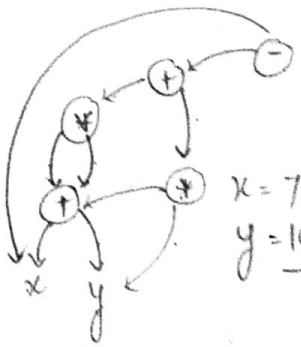
$x = 6$

$y = 8$

Construct DAG for following exp.

$((x+y) * (x+y)) +$

$(y + (x+y)) - x$



$x = 7$

$y = \underline{10}$

Code Optimization → HLL optimization
→ IC - Optimization
→ Assembly code - Optimization

↓ Machine Independent

→ Folding — Constant folding
→ Strength Reduction
→ Dead code elimination
→ Constant Propogation / Variable propogation (copy propogation)
→ Common Sub exp. - elimination
→ ~~Redu~~ Algebric Simplification
→ loop optimization → Code motion / loop invariant / frequed.
→ loop unrolling
→ loop Jamming
→ Induction variable elimination

↓ Machine Dependent

→ Register allocation
→ Addressing mode
→ peephole optimization
↳ reductant load/store
↳ Flow of Control optimization
↳ strength reduction
↳ use of machine idoms

Constant folding → Constant ~~को~~ fold ~~करके~~

① $x = 4 * 3.14;$ ⇒ $x = 8.56;$

② $a = 3 + 4 * b$ ⇒ Can't perform CF

③ $a = 3 * 4 + b$ ⇒ $a = 12 + b;$

④ for(int j=0 ; j<=n ; j++)
{ Arr[2*3] = 10; ⇒ Arr[6] = 10;
}

Que $x = -y * (a+b)$

$t_1 = a+b$
$t_2 = -y * t_1$
$x = t_2$

$t1 = a+b$
$x = -y * t1$

$a = a+b$
$y = -y * a$

result	op1	op2	operator
t1	a	b	+
t2	-y	t1	*
x	t2	—	=

Quadruple

1000	a	b	+
2000	-y	1000	*
3000	2000	-	=

Strength reduction - replace higher cost operator to lower cost operator

A = 4 * A; → A = A + A + A + A; → A = A << 2

Multiplication is higher cost operator

+ is lower cost

<< is again lower

↓

left shift

Copy propogation → Constant propogation

```
int x = 12;
int y = 7 - x/2;
return y * (24/x + 2);
```

Don't propogate constant variable

↓ here we'll put value of x

y = 7 - 12/2 = 7 - 6 = 1 ← Constant folding

return y * (24/12 + 2); ← y * (2+2) = 1 * 4 = 5

Eg → a = 5 | b = 5 + c
b = a + c | d = 5 + e
d = a + e

Variable propogation → a = x | b = x + c
b = a + c | d = x + e
d = a + e

Eg → a = x | b = x + c
b = a + c | a = d + e
a = d + e | g = a * f
g = a * f

Dead code Elimination:→ 1. A = 10
2. B = 20
3. C = 30
4. C = A + B * C → Dead code not being used anywhere
5. X = A + B
6. Y = X + A
7. Z = X + Y
8. Pf ("Z")

Common Sub Expression Elimination :→

a = b + c | a = b + c | a = b + c
d = 2 + 3 + b + c | d = 2 + 3 + a | d = 5 + a

Algebraic Simplification :- $a = a+0$
$a = a \times 1$

Loop Optimization

1) Code Motion / Loop Invariant / Code movement :-

```
while ( i<1000)
{
  A = sinx/cosx * j
}
```

sinx/cosx does not depend on loop and is computed every time

Should be moved either up ward or downward so that the program does not affect

```
t = sinx/cosx
while (j<1000)
{ A = t * j
}
```

Eg →
```
for (int i=0 ; i<n ; i++)
{ a = a + i ;
  x = y * z ;
}
```
inducting variable

```
x = y * z ;
for (i=0; j<n ; i++)
{ a = a + j ;
}
```

2) Loop Unrolling

```
while (j ≤ 100)
{ a[i] = j ;
  j = j+1 ;
}
```

```
while (j ≤ 50)
{ a[i] = j ;
  j = j+1 ;
  a[i] = j ;
} i = j+1 ;
```

Eg →
```
for (i=0 ; i≤99 ; i++)
{ printf ("Today class")
}
```

```
for (i=0 ; i ≤ 49 ; i++)
{ printf ("Today class")
  printf ("Today class")
}
```

3) Loop Jamming :-

```
for(i=1 ; i<=n ; i++)        for(j=1 ; j<=n ; j++)
{ x = x+j                    { y = y+j
}                            }
```

↘ ↙

```
for(j=1 ; j<=n ; j++)
{ x = x+j;
  y = y+j;
}
```

Lecture 30

Loop Optimization

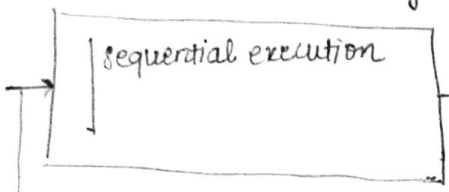
~~4. Induction Variable~~

How to identify loop in a program.

↓

Ans) Using Program flow graph
or
Control flow graph

Basic Block :- Sequence of 3 add. code with no jump.



$x = a + b * c$

$t1 = b * c$

$t2 = a + t1$

$x = t2$

↓↓ Set of instruction (3 add C) executed in a seq

Control never enter in middle and leave at middle

Basic Block start from one leader to next leader not including next leader.

How to find Basic Block :

⇓

Inorder to find basic block we need to find leader

⇓ what is leader? ?

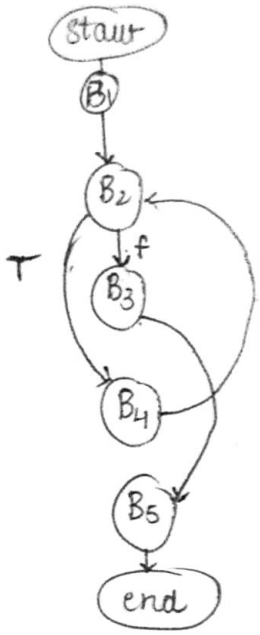
1. 1st statement is always a leader.
2. Target of conditional or unconditional jump is a leader
3. Statement after conditional or unconditional jump is a leader

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

1. f = 1 ⟶ leader
2. i = 2
3. if (i ≤ x) goto 5. → leader
4. goto 9. → leader
5. t_1 = f * i ⟶ leader
6. return f = t_1
7. i = i + 1
8. goto 3.
9. return f ⟶ leader

Basic Block -
Control -
node -
edge

Blocks B_1 → 1, 2
B_2 → 3
B_3 → 4
B_4 → 5, 6, 7, 8
B_5 → 9



Basic Block = 5
Control = 5
nodes = 7
edges = 7

Eg → 2.

```
a = 10
b = 20
while (a + b > c)
  { c = c + 1;
  }
  a = b * d;
```

1. a = 10 ⟶ leader
2. b = 20
3. if t1 = a + b
4. if (t1 < c) goto 7. → leader
5. c = c + 1 ⟶ leader
6. goto 4.
7. a = b * d → leader

1. a = 10 → leader
2. b = 20
3. t1 = a + b

B1

alternate

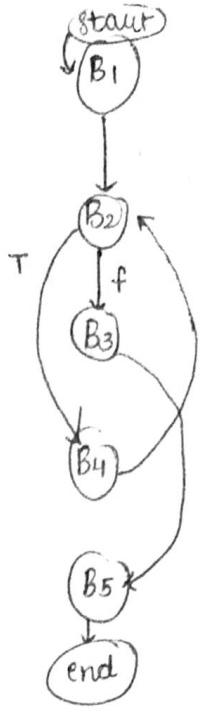
4. if (t1 > c) goto 7 → leader — B2
5. a = b * d → leader
6. goto 9

B3

7. c = c + 1 → leader
8. goto 4

B4

9. exit → leader — B5



Block - 5
Control - 5
node - 7
edge - 7

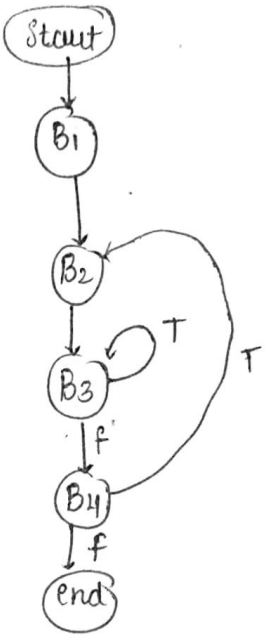
Eg → 3

1. i = 1 → leader — B1
2. j = 1 → leader — B2
3. t1 = j * j → leader
4. t2 = t1 + j
5. t3 = 4 * t2
6. t4 = t3
7. a[t4] = -1
8. j = j + 1
9. if (j < 5) goto 3

→ B3

10. i = i + 1 → leader
11. if (i < 5) goto 2

B4



nodes - 6
edges - 7

Lecture 31

Eg →

1. f = 1 → leader
2. i = 2

B1

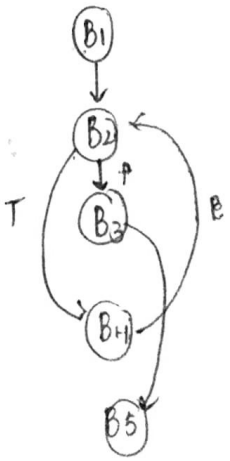
3. if (i <= x) goto 6 — leader — B2
4. return f — leader
5. goto 9

B3

6. f = f * i — leader
7. goto 3 → i = i + 1
8. exit goto 3

B4

9. exit — leader — B5



Que.

1. i = 2 — leader
2. j = 1

B_1

3. a = i * j — leader
4. b = i + j
5. if (b < a) goto 7

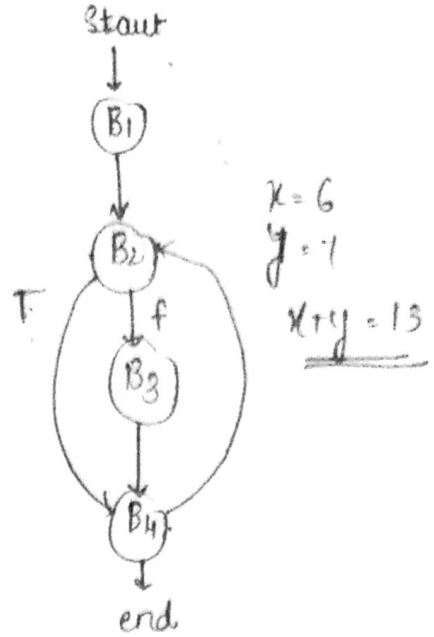
B_2

6. a = a + 1 — leader

B_3

7. j = j + 1 — leader
8. goto 3

B_4



x = 6
y = 7
x + y = 13

Que

```
J = 1
while (J ≤ n)
{
    R = a + b;
    R = R * c;
    R = R + d
    J++
}
```

B_1 1. J = 1 — leader

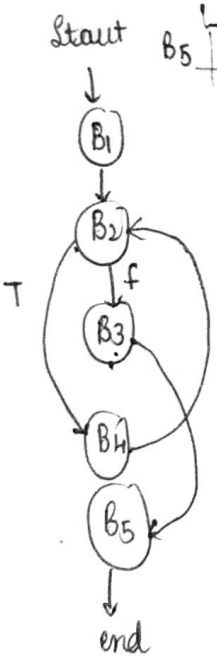
B_2 2. if (J ≤ n) goto 4 — leader

3. goto 9 — leader

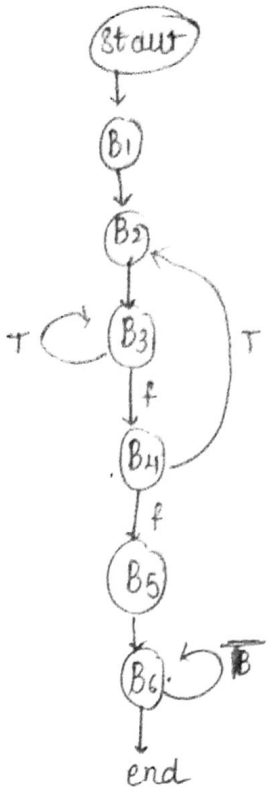
B_3

B_4
4. R = a + b — leader
5. R = R * c
6. R = R + d
7. J = J + 1;
8. goto 2

B_5 9. exit — leader



1. i = 1 —— leader — B_1
2. j = 1 —— leader — B_2
3. t_1 = 10 * i —— leader
4. t_2 = t_1 + j
5. t_3 = 7 * t_2 — B_3
6. t_4 = t_3 - 88
7. a[t_4] = 0
8. i = j + 1
9. if j < 10 goto 3.
10. j = j + 1 —— leader — B_4
11. if j < 10 goto 2
12. j = 1 —— leader — B_5
13. t_5 = j - 1 —— leader
14. t_6 = 88 * t_5 — B_6
15. a[t_6] = 1
16. j = j + 1
17. if j <= 10 goto 13.



node - 8
edge - 10

Que
X = A / b
Y = C + D
Y = Y - X
X = D + Y
Z = f + Y
Z = Z + 4

min var. req to convert to SSA

X_1 = A / B
Y_1 = C + D
Y_2 = Y_1 - X_1
X_2 = D + Y_2
Z_1 = f + Y_2
Z_2 = Z_1 + A

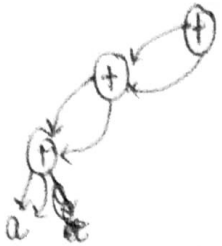
11

PYQ

Ans
1. if (x < y) goto 3
2. goto 12.
3. if (u < v) goto 5.
4. goto 12.
5. if (x = 1) goto 10
6. if (x <= v) goto 8.
7. goto 11.
8. x = x + 3
9. goto 6.
10. y = y + 1
11. goto 1.
12. exit

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

using C rule. + is left to Right associative



Boolean Expression Production

1. $B \rightarrow B_1 \,||\, B_2$ (Logical Expression OR)

$B_1.true = B.true$

$B_1.false = newlabel()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \,||\, label(B_1.false) \,||\, B_2.code.$

2. $B \rightarrow B_1 \,\&\&\, B_2$ (Logical And)

$B_1.true = newlabel()$

$B_1.false = B.false$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

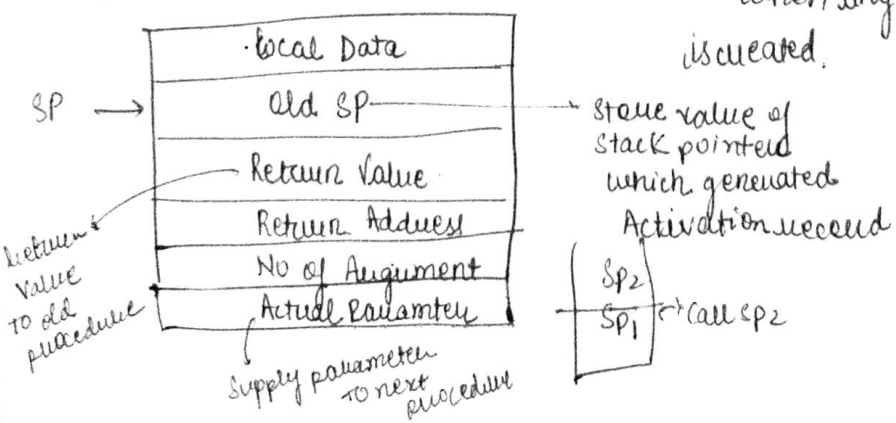
3. $B \rightarrow !\, B_1$ (NOT)

$B_1.false = B.true$

$B_1.true = B.false$

$B.code = B_1.code$

Activation Record – data structure which is invoked when any procedure is created.



5 Error Recovery Strategies

automatic type conversion is done by compiler

① Panic mode - pauses - discards i/p symbol one at a time
② Symbol table - pauses performs original correction like missing ;
③ Global Correction
④ Phrase level Recovery
⑤ Error production → specific errors are there with grammar & if error occurs a specific message is displayed

दूरी changes करता परिवर्तन i/p string me atleast amount of changes करता होगा जो करता है