Compiler Design

→ Ambigious Grammar

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

$w = id + id * id$

$V = \{E\}$

$T = \{+, *, id\}$

LMD (Left most derivation)

$E \Rightarrow E+E$
$\Rightarrow id + E$
$\Rightarrow id + E*E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

RMD (Right most derivation)

$E \to E+E$
$\Rightarrow E + E*E$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

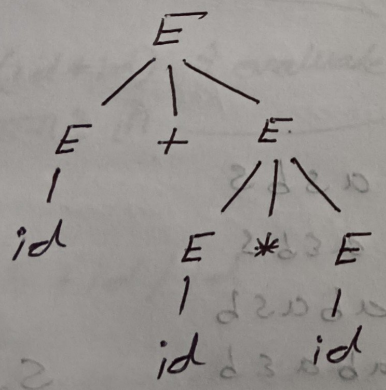
Leftmost मतलब जो variable सबसे left में होगा उसको substitute करेंगे।

LMD_2

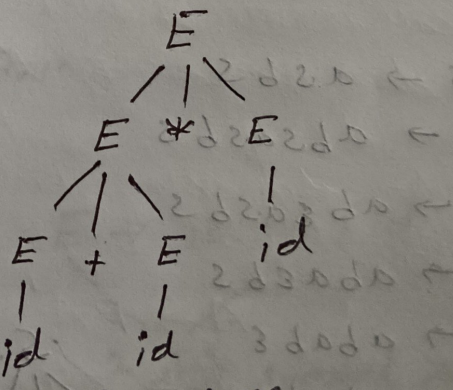
$E \to E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

RMD_2

$E \to E*E$
$\Rightarrow E* id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$



Parse tree



Parse tree

दोनों parse tree का structure अलग है।

किसी भी grammar के लिए 2 different parse tree मिले same structure के लिए या 2 different LMD या 2 different RMD then grammar ambiguous होगा।

Q $S \to as \mid sa \mid a$

Check whether grammar is ambiguous or not

कोई भी string ले के देख लेते हैं

$w = aa$

LMD$_1$ $S \to aS$ {Left most variable S ले}

$\to aa$

LMD$_2$ $S \to Sa$ {Left most variable S ले}

$\to aa$

दो LMD ∴ Ambiguous Grammar.

यहाँ एक ही symbol है तो वही Right most, left most होगा तो RMD for above production भी same होगा।

RMD$_1$

$S \to aS$

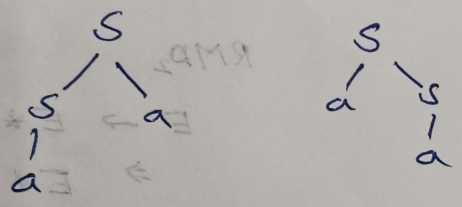
$\to aa$

RMD$_2$

$S \to Sa$

$\to aa$

Parse tree



* Ambiguous Grammar find करने का कोई proper तरीका नहीं है जैसे ही hit & trial से find करते हैं।

Q $S \to asbs \mid bsas \mid \varepsilon$

$w = abab$

LMD$_1$

$S \to asbS$

$\to abSasbS$

$\to ab\varepsilon asbS$

$\to aba\varepsilon bS$

$\to abab\varepsilon$

$\to abab$

LMD$_2$

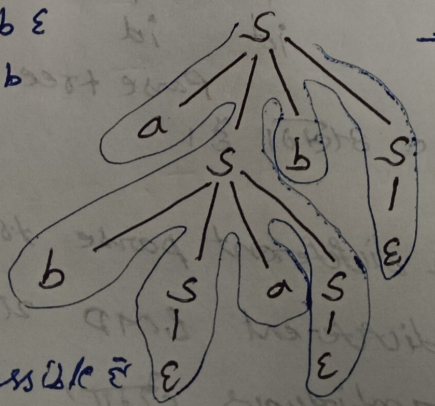
$S \to asbS$

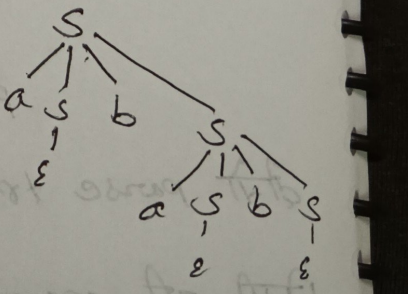
$\to a\varepsilon bS$

$\to abasb$

$\to aba\varepsilon b$

$\to abab$





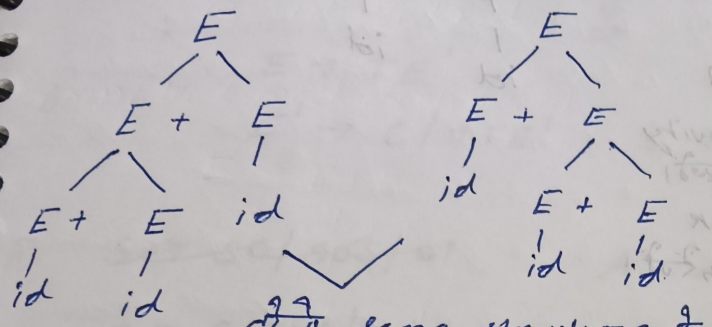
Ambiguous grammar एक से ज्यादा LMD or RMD possible है

दोनों same structure के Parse tree है

→ Ambiguous to unambiguous Grammar

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

$w = id + id + id$



दोनों same structure के Parse tree है।

Ambiguous grammar

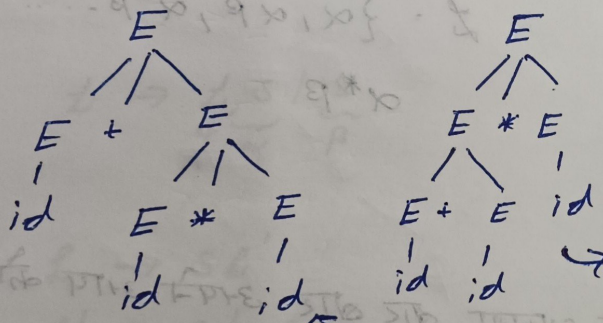
LMD₁

$E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E+E$
$\rightarrow id+id+E$
$\rightarrow id+id+id$

LMD₂

$E \rightarrow E+E$
$\rightarrow E+E+E$
$\rightarrow id+E+E$
$\rightarrow id+id+E$
$\rightarrow id+id+id$

$w = id + id * id$



→ यहाँ evaluation पहले addition का होता है

$w = id + (id * id)$ से evaluate करना है तो → से parse tree नहीं होगा।

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

यें दोनों symbol same है तो Left Recursion

$E \rightarrow id+E \mid id$

यें दोनों symbol same है तो Right recursion.

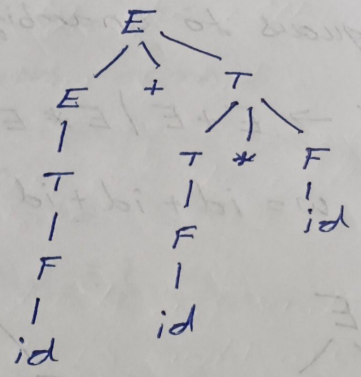
* variable जो arrow के left में है वही variable अगर leftmost में है तब Left recursion अगर Right most में है तब Right recursion.

unambiguous grammar

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$



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

यहाँ operator +, * इनकी precedence और associativity के हिसाब से grammar बनाएंगे।

+, * की associativity L → R

∴ Left recursion use करेंगे।

→ Recursion

Left

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

इससे language generate होती है

$L = \{\beta, \beta\alpha, \beta\alpha^2 \cdots$

$\beta\alpha^* \Rightarrow$ regular expression

Right

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

$L = \{\alpha, \alpha\beta, \alpha^2\beta \cdots$

$\alpha^*\beta$

→ Left recursion loop में चला जाएगा बार बार अपने आप को ही call कर रहा है इसलिए।

Top-down parser को left recursion बिल्कुल मंजुर नहीं है।

Regular expression लिख लो अब आराम से grammar लिख लोगे।

$A \to \beta A'$

$A' \to \varepsilon \mid \alpha A'$ $\Big\} \Rightarrow \beta\alpha^*$

Right recursion से कोई problem नहीं होती।

Q. Remove left recursion

i) $E \to E+T / T$

compare करे तो

$A \to A\alpha / \beta$

$E \to E \, (+T) \, / \, (T) \Rightarrow \beta$, where $+T \Rightarrow \alpha$

$\therefore$

$E \to TE'$

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

ii) ~~S → S0 / S0S / 01~~

$A \to A\alpha / \beta$

$S \to S0S0S / 01$ (S = A, 0S0S = α, 01 = β)

$\beta\alpha^*$

$A \to \beta A'$

$A' \to \varepsilon / \alpha A'$

~~A → 01 A~~

$S \to 01S'$

$S' \to \varepsilon / 0S0SS'$

iii) $L \to L,S / S$ (L = A, ,S = α, S = β)

* $A \to A\alpha_1 / A\alpha_2 / \ldots\ldots / \beta_1 / \beta_2 / \beta_3 \ldots\ldots$

इसको grammar को same उसी तरीके से लिखेंगे।

$A \to \beta_1 A' / \beta_2 A' / \ldots$

$A' \to \varepsilon / \alpha_1 A' / \alpha_2 A' \ldots$

$L \to SL'$

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

* अगर production में operator जहाँ use हुआ है वहाँ left recursion है तो उस operator की associativity left होगी

अगर Right है तो Right associativity होगी मतलब (Right → left)

Q consider the grammar defined by the following production rules, with two operation * and +

$S \to T * P$

$T \to +/T * U \Rightarrow *$ left

$P \to Q + P / Q$ ↳ + → right

$Q \to id$

$U \to id$

which one of the following is true

a) + is left associative, while * is right associative

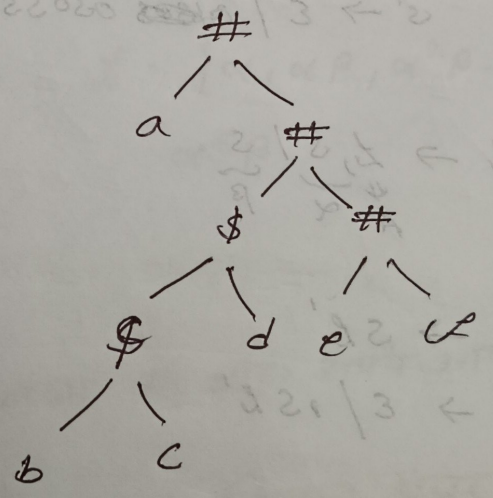
b) + is right associative, while * is left associative

c) Both + & * are right associative

d) Both + & * are left associative

Q consider the following parse tree for the expression

a # b $ c $ d # e # f



comment on associative & precedence.

bottom से up जाएंगे

पहले b $ c evaluate हुआ

फिर (b $ c) $ d

~~पहलवान~~ $ की associativity L → R

और $ पहले evaluate हुआ : precedence $ की ज्यादा

a # ((b $ c) $ d) # e # f

~~(b $ c) $ d~~ अब e # f evaluate होगा।

फिर ~~एवरेज~~ a # (((b $ c) $ d) # (e # f))

∵ # की associativity Right है।

Q Consider two binary operator ↑ and ↓ with the precedence of operator ↓ being lower than that of operator ↑. operator ↑ is right associative while ↓ is left associative. which one of the following represent the parse tree for expression

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

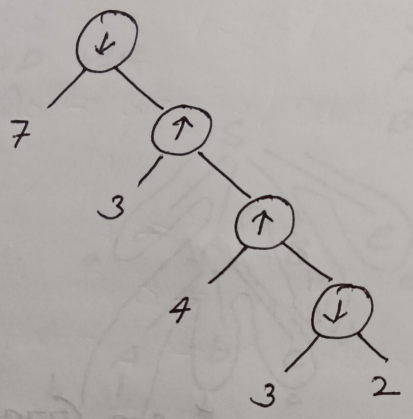
$\left(\left(7 \downarrow \left(3 \uparrow (4 \uparrow 3)\right)\right) \downarrow 2\right)$

↑ > ↓

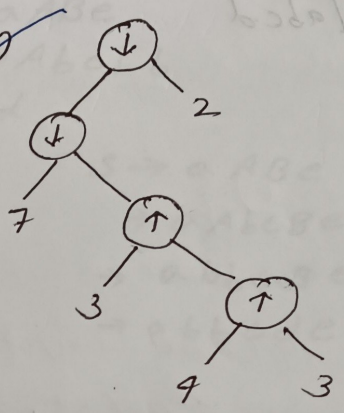
↓ R→L

↓ L→R

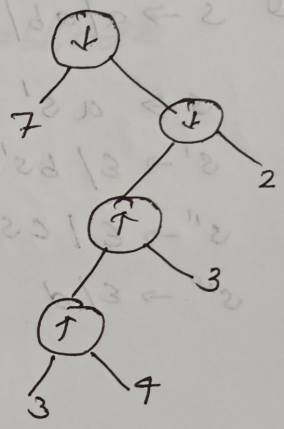
a)



b)



c)



→ Non-Deterministic Grammar को Deterministic Grammar में convert करने को बोलते हैं Left factoring

$$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_4$$

Grammar में अगर prefix common आ रहा है तब Non-Deterministic Grammar same variable के production में

$$\left.\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 / \beta_2 / \beta_3 / \beta_4 \end{aligned}\right\} \text{Deterministic Grammar}$$

$$\left.\begin{aligned} A &\to \alpha\beta_1 \\ B &\to \alpha\beta_2 \end{aligned}\right\}$$ इसमें Non Deterministic नहीं है।

दो different production है।

जैसे ही left factoring eliminate करते है।

Q $S \to iEts / iEtses / a$

$E \to b$

$S \to iEtSS' / a$

$S' \to \varepsilon / es$

$E \to b$

Q $S \to assbs / asasb / abb / b$

~~$S \to ass' / abb / b$~~

$S \to as' / b$

$s' \to bb / ss''$

$s'' \to sbs / asb$

Q $S \to bss aas / bss asb / bsb / a$

$S \to bss' / a$

$s' \to b / sas''$

$s'' \to as / sb$

Q $s \to a / ab / abc / abcd$

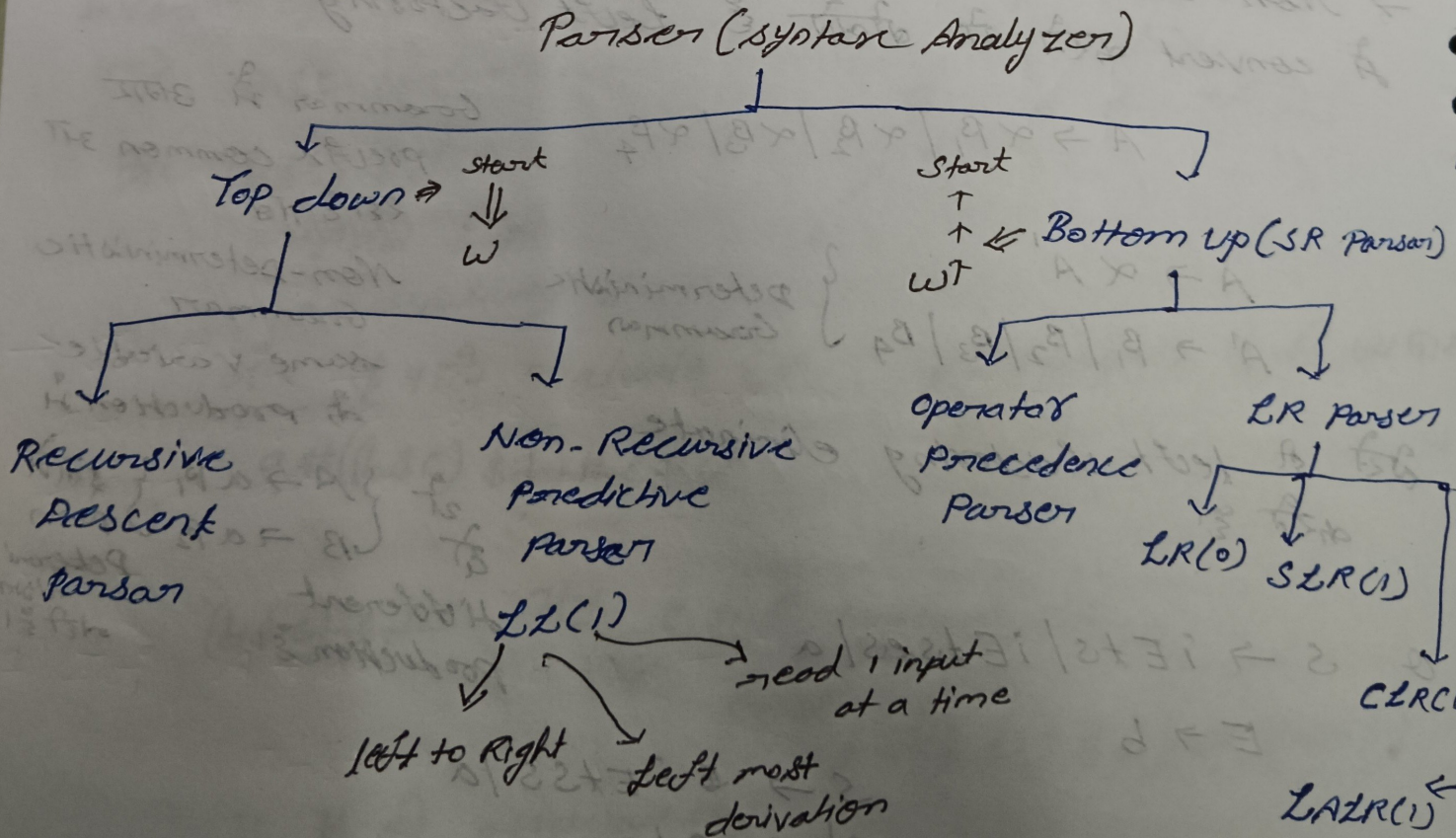
$s \to as'$

$s' \to \varepsilon / bs''$

$s'' \to \varepsilon / cs'''$

$s''' \to \varepsilon / d$

→ Process involved in generating string from Grammar is known as Parser.



→ operator precedence parser ही ऐसा ambiguous grammar को accept करती है।

→ Top down approach use कर रहे हैं तो न तो Left Recursion हो और न ही left factory हो grammar में।

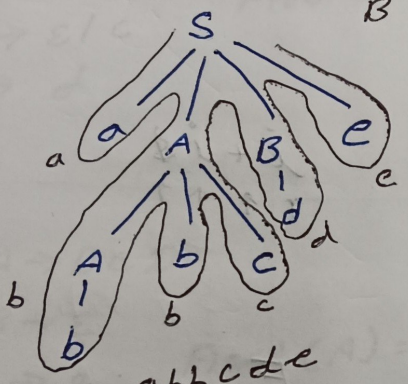
→ Parser में जो कुछ भी पढ़ेंगे तो context free grammar होगी।

→ Top Down parser

LMD use करेंगे।

$S \to aABe$
$A \to Abc \mid b$
$B \to d$

$w = abbcde$



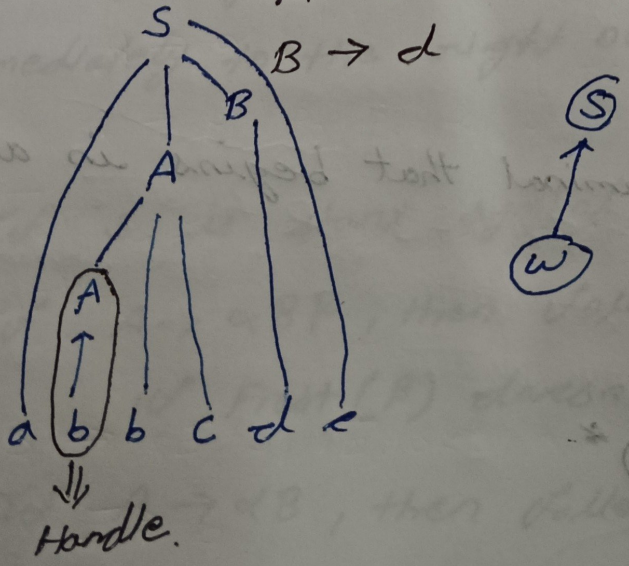
$S \to aABe$
$\to aAbcBe$
$\to abbcBe$
$\to abbcde$

→ Bottom up parser

Right most derivation reverse order में use करना है।

$S \to aABe$
$A \to Abc \mid b$
$B \to d$

$w = abbcde$



$S \to aABe$
$S \to aAde$
$S \to aAbcde$
$\to abbcde$

पहले RMD लिख लो फिर bottom से up tree बना लेंगे।

→ Recursive Descent Parser

$E \rightarrow iE'$

$E' \rightarrow +iE' / \varepsilon$

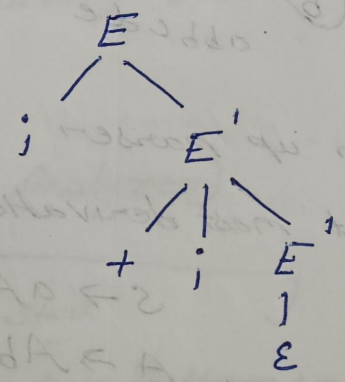
simple है E, E' का function बना लेते हैं।

```
Main() {
    E();
    if (input == $)
        Parsing Successful
}

E() {
    if (input == i)
        input ++;
    E'();
}

E'() {
    if (input == +) {
        input ++;
        if (input == i) {
            input ++;
            E'();
        }
        else return;
    }
```

i + i $
↑ ↑ ↑ ↑



→ First ()

First(A) give sets of terminal that begins in all strings derived from A

Rule:-

1) If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$

$First(A) = \{a\}$

2) If $A \rightarrow \varepsilon$ then $First(A) = \{\varepsilon\}$

3) If $A \to BC$ then

i) First(A) = First(B) if First(B) doesn't contain ε

ii) First(A) = First(B) ∪ First(C) if First(B) contain ε

Q $A \to ab/ba/cd$

First(A) = {a, b, c}

Imp. point

$A \to BCD$

First(A) में ε नहीं रखेंगे

और B, C, D का combination null देगा।

Q $A \to BCD$

$B \to \varepsilon/b$

$C \to \varepsilon/c$

$D \to d$

First(A) = First(B) ∪ First(C) ∪ First(D)

= {ε, b} ∪ {ε, c} ∪ {d}

= {b, c, d, ~~ε~~}

→ but ε को null को नहीं रखेंगे।

Q $A \to BCD$

$B \to \varepsilon$

$C \to a$

$D \to \varepsilon$

First(A) = First(B) ∪ First(C)

= {ε} ∪ {a}

= {a}

→ Follow()

Follow(A) give set of all terminals that follow immediately to the right of A

Rule:

1) If S is start symbol then Follow(S) = {$}

2) If $A \to \alpha B \beta$, then follow(B) = First(β) if First(β) doesn't contain ε

3) If $A \to \alpha B$, then follow(B) = Follow(A)

Q $A \rightarrow aBC$

$B \rightarrow b$

$C \rightarrow c$

~~Follow(A) = Follow(C)~~

~~Follow(C) = {c~~

Follow को हमेशा bottom से निकाला करो

Follow(A) = {$} → start symbol है

Follow(B) = ~~Follow(C)~~ First(C)
= {c}

Follow(C) = Follow(A) = {$}

> * Start symbol का Follow {$} आता है
> &
> Follow में ε नहीं आता।

Q $S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \varepsilon$

$D \rightarrow EF$

$E \rightarrow g / \varepsilon$

$F \rightarrow f / \varepsilon$

Variable	First()	Follow()
S	{a}	{$}
B	{c}	{g, f, h}
C	{b, ε}	{g, f, h}
D	{g, f, ε}	{h}
E	{g, ε}	{f, h}
F	{f, ε}	{h}

Follow(B) = First(D)

Follow(C) = Follow(B)

$S \rightarrow aB(Dh)$

Follow(B) = First(D) = {g, f, ε} D के जगह पे ε रख देंगे और देखेंगे immediately right में क्या आएगा

$S \rightarrow aBDh$

aBgh ⇒ g

aBfh ⇒ f

aBεh ⇒ h

Follow में ε नहीं होता।

Follow(C) = Follow(B)

Follow(D) = D वाले के Follow वाले ε।

$C \to bE$

$B \to cC$

$D \to EF$

Follow(E) = First(F)
$\{f, \varepsilon\}$
$= \{f, k\}$

$D \to EF$

Ef $E\varepsilon \Rightarrow D \to E$

Follow(D)

Follow(F) = Follow(D)

(2) $E \to TE'$

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

$T \to FT'$

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

$F \to id / (E)$

E start symbol है इसलिए $

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

First(E) = First(T) = First(F)
{id, (}

Follow(E) = {)}
↓
$F \to id / (E)$

Follow(T) = {), +, $}
↓
$E \to TE'$

$E' \to +TE'$

↓ First E'

$E \to T\varepsilon$ $E \to T+$ $E' \to +T$ $E' \to +T+$

Follow(E) = ε

Follow(F):
↓
$T \to FT'$ $T' \to *FT'$
$T \to F\varepsilon$, $T \to F*$
Follow(T')
/ ↘
$T \to FT'$ $T' \to *FT'$
Follow(T) Follow(T')

Follow(E')
↙ ↘
$E \to TE'$ $E' \to +TE'$
Follow(E)

Q $S \to Bb \mid Cd$

$B \to aB \mid \varepsilon$

$C \to cC \mid \varepsilon$

Variable	First()	Follow
S	{a, b, c, d}	{$}
B	{a, ε}	{b}
C	{c, ε}	{d}

Follow(B) = b

Follow(C)

↳ $S \to Cd$

$S \to aAbd$, $A \to BC$

Follow(C) कैसे Follow(A) ही रहा है?

$S \to aBCbd$ simply substitute कर दो

Follow(C) = b

Follow(A) = b

$S \to aAbd$

* कोई भी string है उसके last में $ लगा देते हैं

id + id $

और start variable के end में भी $ लगा देते हैं

मतलब start variable जो return करेगा उसके बाद ($) S$ जब $, $ match हो गया तो string accepted.

First(S) = First(B) ∪ b ∪ First(C) ∪ d

= {a, b} ∪ {c, d}

= {a, b, c, d} ε कभी generate नहीं होगा.

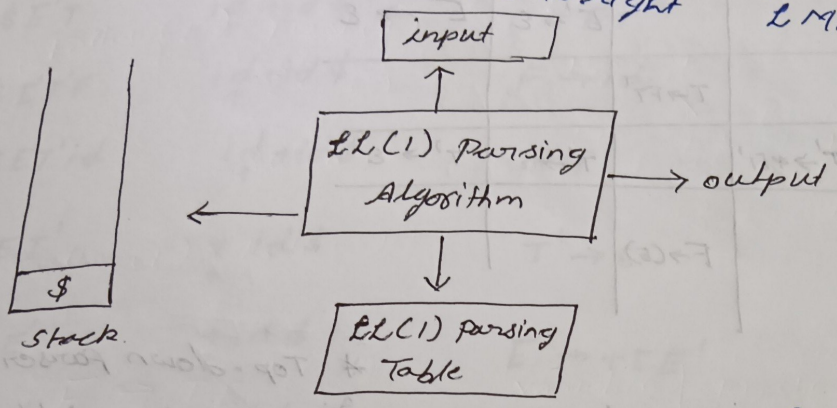
Q $S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

Variable	First()	Follow
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{b, a}

→ Non-Recursive Descent Parsar
or
Predictive Parsar or LL(1) → Left to right, LMD, at a time one symbol read करेंगे।



How to construct LL(1) parsing Table → ये तभी बनेगा जब given grammar ambiguous न हो, न ही Left recursion हो, न ही left factory हो।

Rule

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

2) Add $A \to \alpha$ under $M[A, a]$ where $a \in Follow(A)$ if $First(\alpha)$ contain ε

Table ← Row - variable
→ Column - Terminal

	First	Follow
$E \to TE'$	$\{id, ($	$\{\$,)\}$
$E' \to \varepsilon \mid +TE'$	$\{\varepsilon, +\}$	$\{\$,)\}$
$T \to FT'$	$\{id, ($	$\{+, \$,)\}$
$T' \to \varepsilon \mid *FT'$	$\{\varepsilon, *\}$	$\{+, \$,)\}$
$F \to id \mid (E)$	$\{id, ($	$\{*, +, \$\}$

* Parsing Table के 1 cell में एक से ज्यादा entry आ गई तो conflict इसका मतलब given grammar LL(1) नहीं है या तो ambiguous है या तो उसमें Left recursion है या Left factory है।

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

$F \to id / (E)$

$F \to id \mid F \to (E)$

* Top-down parser में जहाँ 1 variable के corresponding 2 production हों किसको choose करें। यह problem इस Table से resolve हो जाएगी।

$E \to \underbrace{T}_{\alpha} E'$

A

$\alpha \to First(T) = \{ id, (\}$

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

पहले consider कर रहे हैं

$E' \to \underbrace{+TE'}_{\alpha}$

A

$First(\alpha) = +$

$\underbrace{E'}_{A} \to \underbrace{\varepsilon}_{\alpha}$

$Follow(E')$

$\{ \$,) \}$

$T \to \underbrace{FT'}_{\alpha}$

A

$First(\alpha) = First(F)$

$= \{ id, (\}$

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

$T' \to \underbrace{*FT'}_{\alpha}$

$First(\alpha) = \{ * \}$

$T' \to \varepsilon$

$Follow(T')$

$\{ +, \$,) \}$

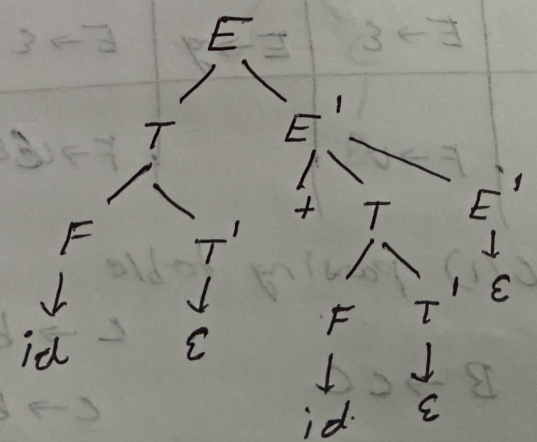
* End

Stack	Input	Production
$E	id+id$	$E \to TE'$
$E'T	id+id$	$T \to FT'$
$E'T'F	id+id$	$F \to id$
$E'T'id	id+id$	pop
$E'T'	+id$	$T' \to \varepsilon$
$E'	+id$	$E' \to +TE'$
$E'T+	+id$	pop
$E'T	id$	$T \to FT'$
$E'T'F	id$	$F \to id$
$E'T'id	id$	pop
$E'T'	$	$T' \to \varepsilon$
$E'	$	$E' \to \varepsilon$
$	$	Accepted

Top of stack को replace E के जगह TE' से replace कर दिया।

Top of stack & input symbol match किया तो pop करेंगे। और pointer को increment करेंगे।

Production जिस order में लिखा है उसी order में parse tree बना लेंगे।



LMD

Q Construct parsing Table & parse tree for acbh$

	First	Follow
$S \to aBDh$	{a}	{$}
$B \to cC$	{c}	{g, f, h}
$C \to bC \mid \varepsilon$	{b, ε}	{g, f, h}
$D \to EF$	{g, f, ε}	{h}
$E \to g \mid \varepsilon$	{g, ε}	{f, h}
$F \to f \mid \varepsilon$	{f, ε}	{f h}

Follow(B) = First(D)

$S \to aBDh$

$S \to acbh$ (f) $S \to aBfh$

$S \to aBgh$ (ε) $S \to aBh$

Follow(E) = First(F)

$D \to EF$

$D \to Ef$ {f}

$D \to E$ (ε) Follow(D) {h}

Now, Parsing Table

M	a	b	c	f	g	h	$
S	$S \to aBDh$						
B			$B \to cC$				
C		$C \to bC$		$C \to \varepsilon$	$C \to \varepsilon$	$C \to \varepsilon$	
D		.		$D \to EF$	$D \to EF$	$D \to EF$	
E				$E \to \varepsilon$	$E \to g$	$E \to \varepsilon$	
F				$F \to f$		$F \to \varepsilon$	

LL(1) parsing table

$S \to aBDh$

α

{a}

$E \to g \mid E \to \varepsilon$

$B \to cC$

$D \to EF$

$D \to gF$ (ε) $D \to F$

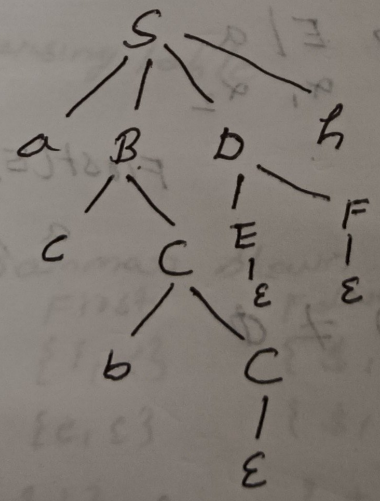
{g} $D \to f$ $D \to \varepsilon$ Follow(D) = {h}

$C \to bC \mid \varepsilon$

$C \to bC \mid C \to \varepsilon$

{b} Follow C {g, f, h}

Stack	input	Production
$S	acbh$	S → aBDh
$hDBa	acbh$	pop
$hDB	cbh$	B → cC
$hDCc	cbh$	pop
$hDC	bh$	C → bC
$hDCb	bh$	pop
$hDC	h$	C → ε
$hD	h$	D → EF
$hFE	h$	E → ε
$hF	h$	F → ε
$h	h$	pop
$	$	accepted



LMD

→ How to check given Grammar is LL(1) or not

1) If G doesn't contain ε

$A \to \alpha_1 / \alpha_2 / \alpha_3$

$First(\alpha_1) \cap First(\alpha_2) = \phi$

$First(\alpha_2) \cap First(\alpha_3) = \phi$

$First(\alpha_3) \cap First(\alpha_1) = \phi$

2) If G contains ε

$A \to \alpha_1 / \alpha_2 / \varepsilon$

$First(\alpha_1) \cap First(\alpha_2) = \phi$

$First(\alpha_1) \cap Follow(A) = \phi$

$First(\alpha_2) \cap Follow(A) = \phi$

Q $S \to E / a$

$E \to a$

$S \to E / a$
$\alpha_1 \quad \alpha_2$

$First(E) \cap First(a)$
$a \cap a = a$

∴ LL(1) नहीं है

$First(\alpha_1) \cap First(\alpha_2) \neq \phi$

∴ LL(1) नहीं है।

Q $S \to aABb$ ⇒ इसमें कोई problem नहीं :

$A \to a / \varepsilon$

$B \to d / \varepsilon$

$A \to a / \varepsilon$
$\alpha_1 \quad \alpha_2$

~~$First(A) \cap First(\alpha) = \phi$~~

$Follow(A) \cap First(a)$
$\{d, b\} \cap \{a\} = \phi$

LL(1) ✓

~~∴ LL(1) नहीं है~~

$S \to aABb$
$S \to aAdb \quad aAb$

$Follow(B) \cap First(\phi)$
$b \cap \phi = \phi$

Q $S \to aSA \mid \varepsilon$
$A \to c \mid \varepsilon$

$First(aSA) \cap Follow(S)$
$\{a\} \cap \{\$, c\} = \phi$

$First(c) \cap Follow(A)$
$c \cap \{\$, c\} = c \neq \phi$

LL(1) ✗

Q $S \to \underset{\alpha_1}{aSbS} \mid \underset{\alpha_2}{bSaS} \mid \underset{\alpha_3}{\varepsilon}$

$First(\alpha_1) \cap First(\alpha_2)$	$First(\alpha_1) \cap Follow(S)$	$First(\alpha_2) \cap Follow(S)$
$a \cap b = \phi$	$a \cap \{b, a, \$\}$	$b \cap \{b, a, \$\}$
	a	b

∴ LL(1) ✗

For checking grammar is LL(1) or not we have 2 method — Direct method
Parsing table

Q Consider the grammar shown below

	First	Follow
$S \to iE+SS' \mid \alpha$	$\{i, \alpha\}$	$\{\$, e\}$
$S' \to eS \mid \varepsilon$	$\{e, \varepsilon\}$	$\{\$, e\}$
$E \to b$	$\{b\}$	$\{+\}$

In the predictive parse table M of this grammar the entries $M[S', e]$ and $M[S', \$]$ respectively are

a) $\{S' \to eS\}$ and $\{S' \to \varepsilon\}$
b) $\{S' \to eS\}$ and $\{\ \}$
c) $\{S' \to \varepsilon\}$ and $\{S' \to \varepsilon\}$
d) $\{S' \to eS, S' \to \varepsilon\}$ and $\{S' \to \varepsilon\}$

$S' \to eS$
$\{e\}$

$S' \to \varepsilon$
$\Downarrow$ Follow(S')
$\{\$, e\}$

Q Consider the grammer

$S \to aAbB \mid bAaB \mid \varepsilon$

$A \to S$

$B \to S$

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

M	a	b	$
S	E_1	E_2	$S \to \varepsilon$
A	$A \to S$	$A \to S$	
B	$B \to S$	$B \to S$	E_3

$S \to aAbB$ {a}

$S \to bAaB$ {b}

$S \to \varepsilon$ {$, a, b}

E_1 : $S \to aAbB$, $S \to \varepsilon$

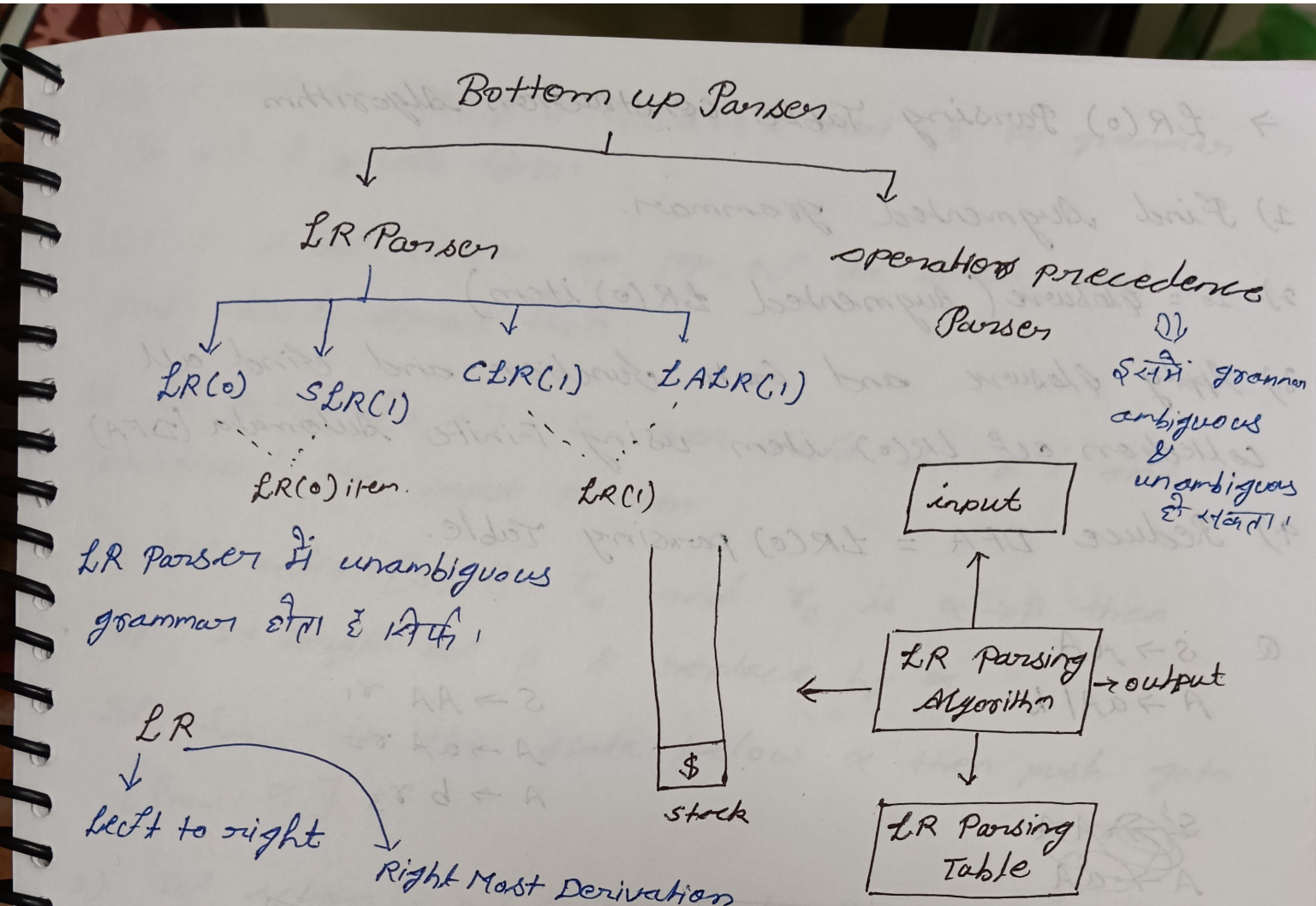
E_2 : $S \to bAaB$, $S \to \varepsilon$

E_3 : $B \to S$

a → ~~B → a~~ {a}

b → {b}

ε → $B \to \varepsilon$, Follow(B)



→ Closure (I)

1) Add LR(0) item I to Closure (I)

2) If $A \to \alpha . B\beta$ is LR(0) item I and $B \to \gamma$ is in G then Add $B \to .\gamma$ to Closure (I) also Repeat above two steps for every newly added LR(0) item.

→ Goto (I, X)

1) Add LR(0) item I by moving dot after X

2) Apply Closure to the result obtained in step I

→ LR(0) Parsing Table construction Algorithm

1) Find Augmented grammar.

2) I_0 = Closure (Augmented LR(0) item)

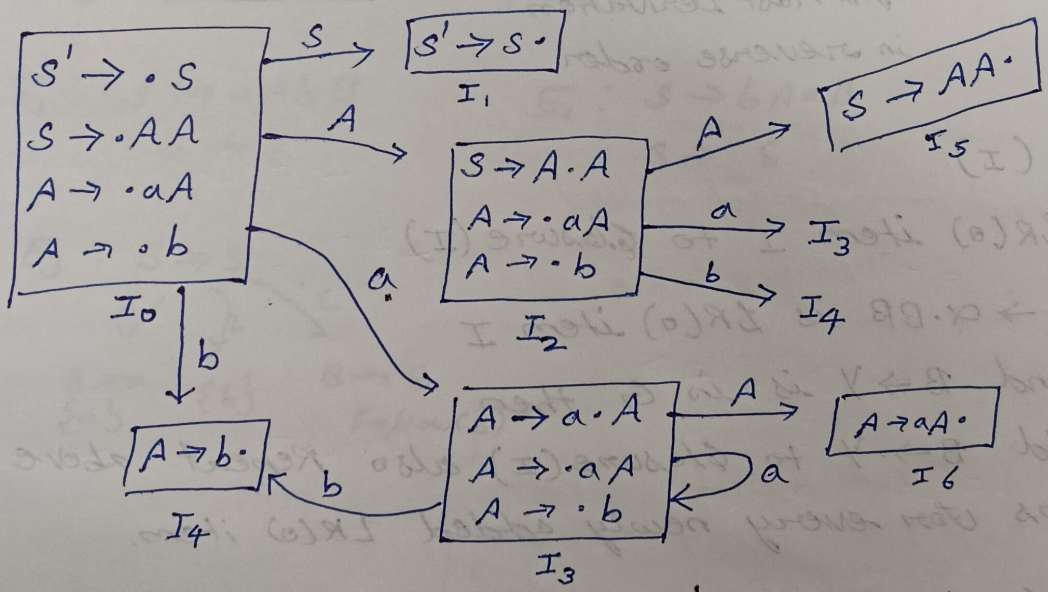
3) Apply Closure and Goto function and find all collection of LR(0) item using Finite Automata (DFA)

4) Reduce DFA = LR(0) parsing Table.

Q $S \to AA$
$A \to aA \mid b$

$S \to AA \ r_1$
$A \to aA \ r_2$
$A \to b \ r_3$

$S' \to \cdot AA$
$A \to \cdot aA$



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

कि~~स~~A ~~A~~ slot में Multiple entry नहीं है Hence LR(0)

Check करना है string S = aabb$ part है इस grammar कि नहीं है उसके लिए।

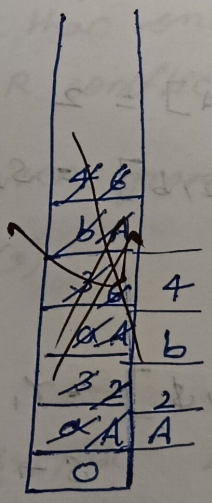
If S is the states on top of the stack and a is look ahead symbol then

1) If action [S, a] = S_i then shift a & i and increment the input pointer

2) If action [S, a] = r_j and r_j is $\alpha \to \beta$ then pop 2× length of β & replace by α

If S_{m-1} is the state below α then push goto $[S_{m-1}, \alpha]$

3) If action [S, a] = accepted then successful completion



a a b b $
↑ ↑ ↑ ↑

action [4, b] = r_3

$A \to b$

2 pop

goto [3, A] = 6

action [6, a] = r_2

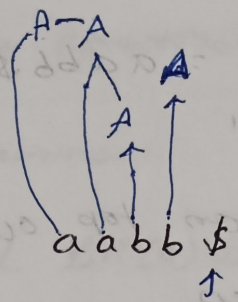
$A \to aA$ 4 pop

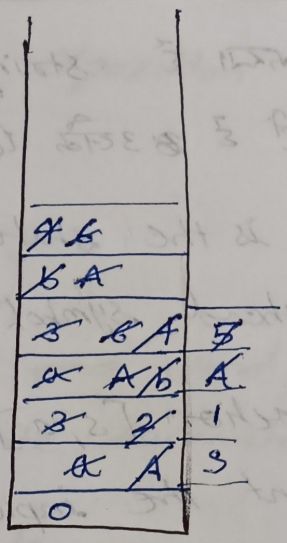
action [6, b] = r_3

$A \to b$ ⇒ 2 pop

action [2, b] = S4

$w = a\ a\ b\ b\ \$$





action [0, a] = s_3

action [3, a] = s_3

action [3, b] = s_4

action [4, b] = r_3

$A \to b$ ⇒ 2 pop

goto [3, A] = 6

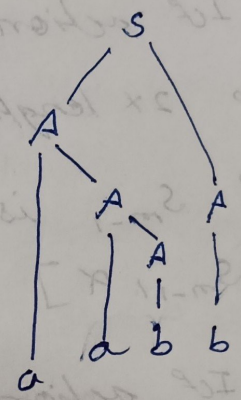
action [6, b] = r_2

$A \to aA$ 4 pop

goto [3, A] = 6

action [6, b] = r_2

$A \to aA$ ⇒ 4 pop



goto [0, A] = 2

action [2, b] = s_4

action [4, $] = r_3

$A \to b$

2 pop

goto [2, A] = 5

goto [0, S] = 1

action [1, $] = acc.

action [5, $] = r_1

$S \to AA$ 4 pop

∴ string accepted.

→ SLR(1) Parser

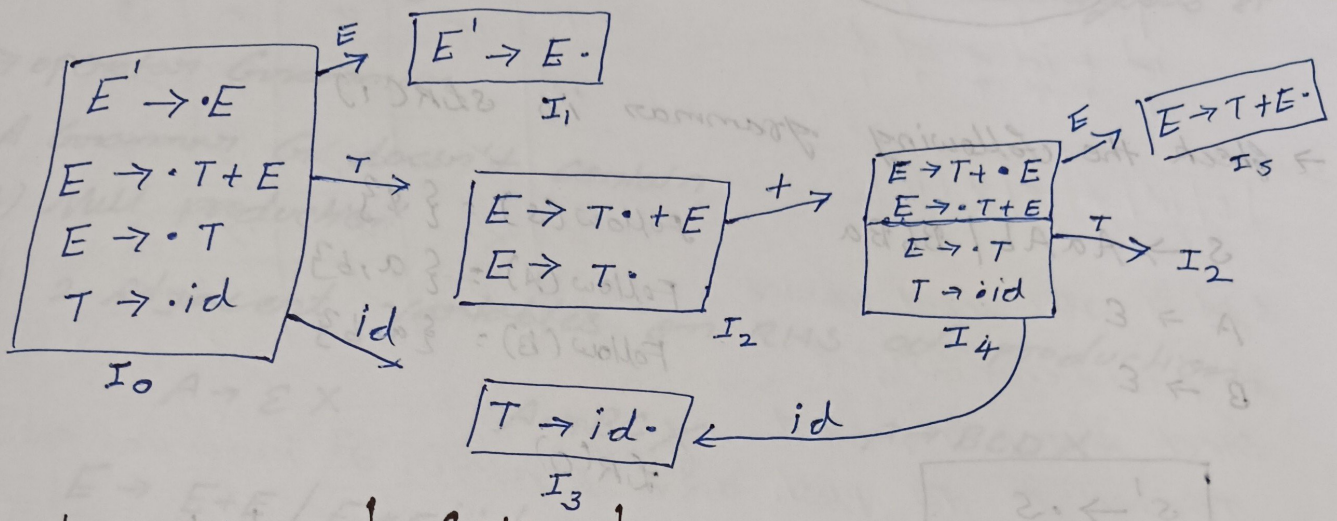
→ $E \to T+E \mid T$

$T \to id$

$E \to T+E \quad r_1$

$E \to T \quad r_2$

$T \to id \quad r_3$



	Action			Goto	
	id	+	$	E	T
0	S_3			1	2
1			acc		
2	~~r_2~~	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

→ More than one entry
S-R conflict

So, given grammar is not LR(0)

जो grammar reduced है वहा है suppose $A \to \alpha$ then Follow of A निकाल लेंगे और जो terminal आएगा उसी में entry करेंगे।

r_1: $E \to T+E$.
Follow(E) = {+, $}

r_2 : $E \to T$.
Follow(E) = {+, $}

r_3: $T \to id$
Follow(T) = {+, $}

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

SLR(1)
LR(0)

→ Check the following grammar is SLR(1)

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

Follow(S) = {$}

Follow(A) = {a, b}

Follow(B) = {a, b}

$S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to .\varepsilon$
$B \to .\varepsilon$

I_0

LR(0)

	a	b	$
0	r_3/r_4	r_3/r_4	r_3/r_4

r_3 $A \to \varepsilon$ Follow A {a, b}

r_4 $B \to \varepsilon$ {a, b}

SLR(1)

	a	b	$
0	r_3/r_4	r_3/r_4	

∴ SLR(1) नहीं है।

पूरा table बनाने कि जरूरत नहीं है।

Q $S \to \overset{r_1}{Aa} \mid \overset{r_2}{bAC} \mid \overset{r_3}{dC} \mid \overset{r_4}{bda}$

$A \to \overset{r_5}{d}$

Follow(S) = {$}

Follow(A) = {a, c}

$S' \to .S$
$S \to .Aa$
$S \to .bAC$
$S \to .dC$
$S \to .bda$
$A \to .d$

I_0 —d→ I_4 $S \to d.C$, $A \to d.$

SLR(1)

	a	b	$	c	d
4	r_5			r_5/S_7	

conflict

∴ SLR(1) नहीं है।

→ operator Precedence Parser

↓

इसमें grammar, Ambiguous or unambiguous हो सकता है।

→ operator Grammar

A Grammar G doesn't contain

1.) Null production

2.) 2-Adjacent variables on RHS of production

$A \to \varepsilon X$ $A \to BCX$ $A \to BCDX$

$E \to E+E / E*E / id$ ✓

$E \to E+E / E*E / \varepsilon$ ✗ not a operator grammar

अगर Grammar OG में नहीं है तब कैसे convert करेंगे।

$S \to SAS / a$
$A \to bSb / b$ } ↓ Not in OG

जहाँ problem है उसी production वहाँ डाल दो.

$S \to SbSbS / SbS / a$
$A \to bSb / b$] अब OG में है.

Q. $P \to SR / S$
$R \to bSR / bS$
$S \to wbS / w$
$w \to \mathcal{L} * w / \mathcal{L}$
$\mathcal{L} \to id$

$P \to SR.$
$R \to bSR$ } ← दो problem है

$P \to \varepsilon SbSR / SbS / S$

$P \to SR$ also SR की जगह P रख देते है

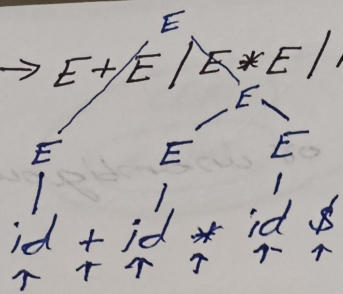
$P \to SbP / SbS / S$

useless production ⇐ $R \to bP / bS$

$S \to wbS / w$
$w \to \mathcal{L} * w / \mathcal{L}$
$\mathcal{L} \to id$

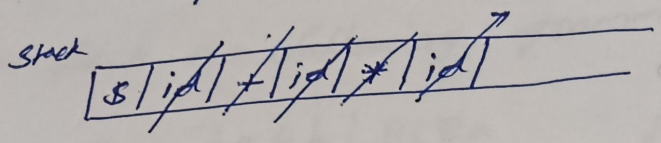
} OG

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



operator relation table

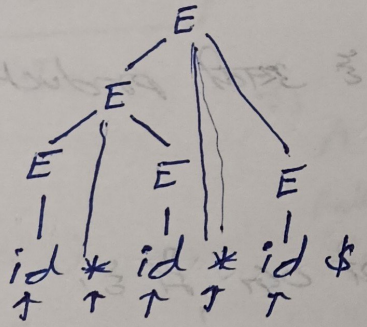
	id	*	+	$
id		$\cdot>$	$\cdot>$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

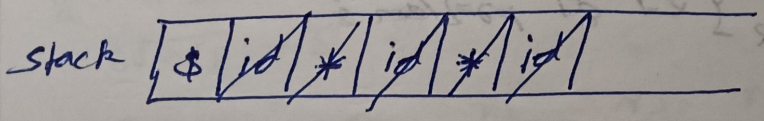


$, id से आगर input की precedence ज्यादा है
stack के top के present
Terminal से तब push करेंगे & pointer को increment कर देंगे।
नहीं तो pop

$, $ तथा तब accepted.

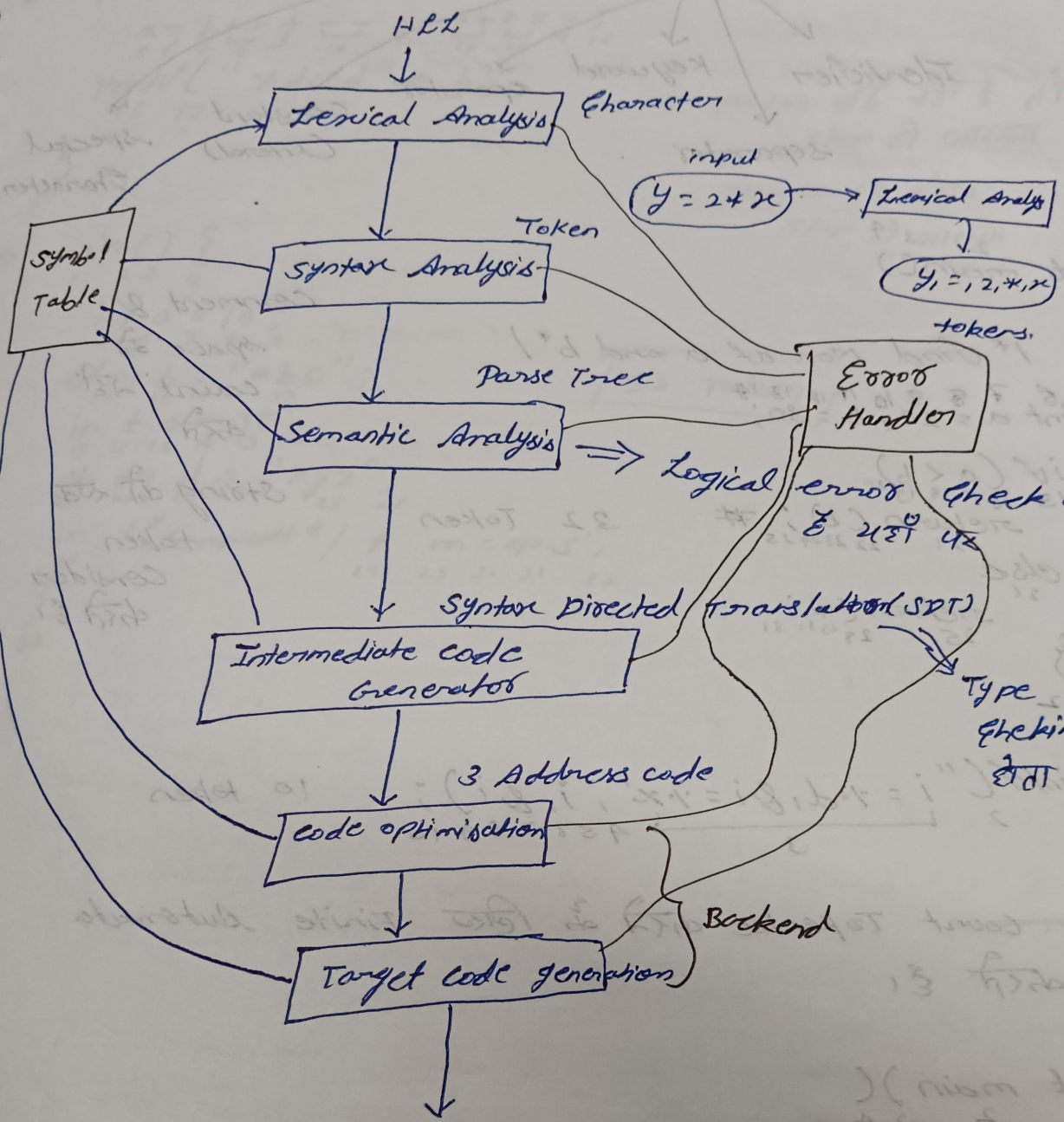
terminal, alphabet की precedence ज्यादा होती है operator से।





$ top : $
$ input : $
accepted

→ Phases of compiler



→ Lexical Analysis [Lexer, Tokenizer, Scanner]

1) Tokenization
2) Give Error messages → Exceeding length
→ unmatched string
→ Illegal character
3) Eliminate comment, white space (Tab, Blank space, New line)

Tokens

- Identifier
- Separator
- Keyword
- Operator
- Constant (literal)
- Special Character

Q.
```
int main()
{
  /* find max of a and b */
  int a = 20, b = 30;
  if (a < b)
    return (b);
  else
    return (a);
}
```

32 Token

Comment & space को count नहीं करते

String को एक token consider करते हैं।

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

10 token

~~Token count~~ Tokenise करने के लिए Finite Automata use करते हैं।

Q.
```
int main)(
}
x = y+z;
int x, y, z;
printf("sum %d %d", x);
{
```

26 token

Q.
```
main(){
a = b+++--_ _-++ + == ;
printf("%d %d, a,b);
}
```

27

~~quotes start हुआ but end नहीं हुआ~~

Q. main() {
1 2 3 4

a = b++ + -- -- ++ + == ;
5 6 7 8 9 10 11 12 13 14 15

printf(" %d %d", a, b);
16 17

}

// quotes बंद नहीं है तो
stop हो जाएगा
answer 17 होगा
stop हो जाएगा

Q. main() {
1 2 3 4

int a = 10 ;
5 6 7 8 9

Char b = "abc" ;
10 11 12 13 14

in t c = 30 ;
15 16 17 18 19 20

Ch ar d = "xyz" ;
21 22 23 24 25 26

in /*comment*/ t m = 40.5 ;
27 28 29 30 31 32

}
33

33 Token

→ CLR(1) & LALR(1)

LR(1) = LR(0) item + look ahead symbols

जिस Variable के बाद से जो production है उसके बाद वालों का First निकाल लेंगे।

$S \to aAbC$
$A \to b$

$S' \to \cdot S$
$S \to \cdot AbC$ ⇒
$A \to \cdot b$

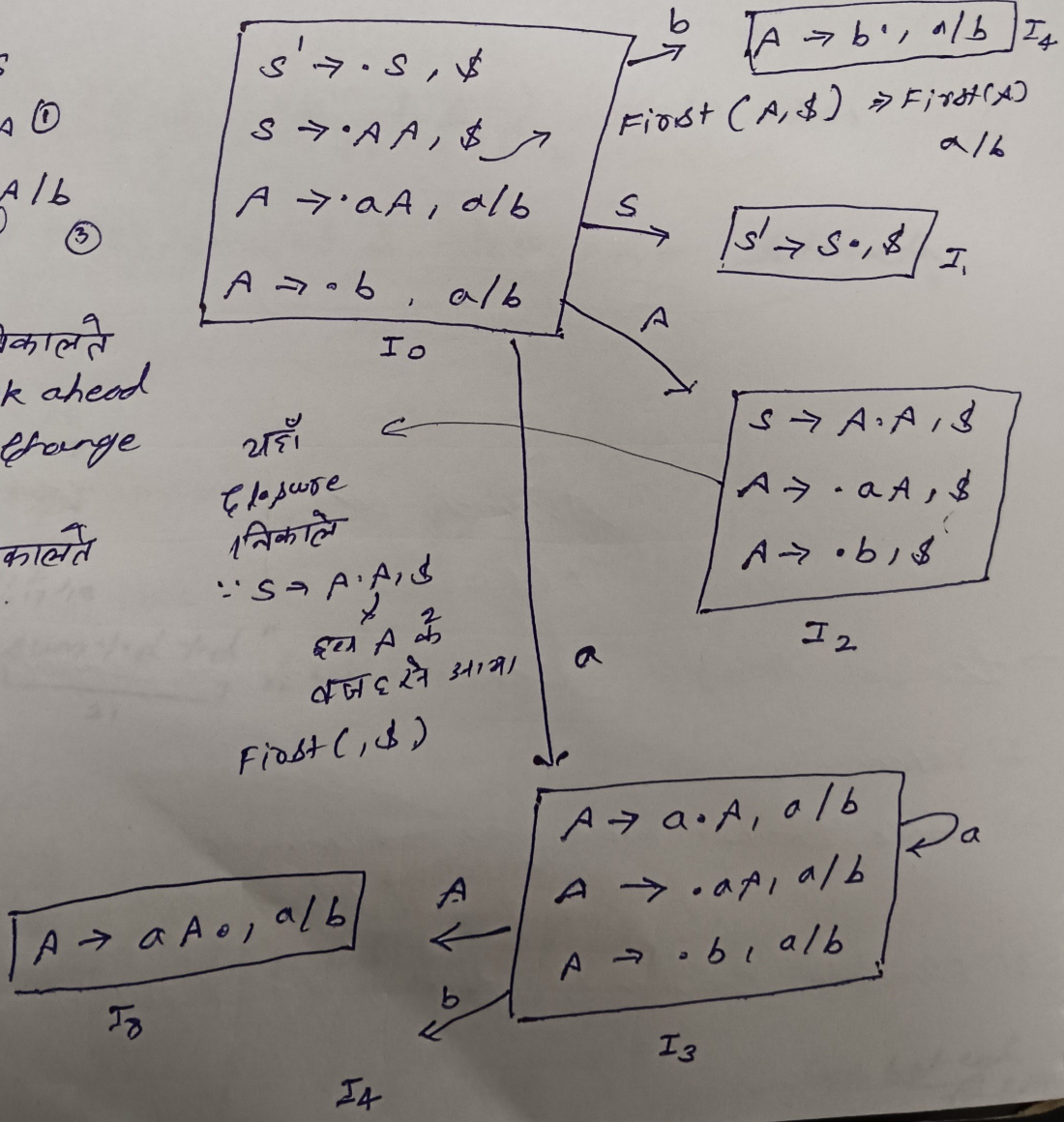
$S' \to \cdot S, \$$
$S \to \cdot AbC$ → से वाली production S के बाद से आई है $S \to \cdot S, \$$
$S \to \cdot AbC, \$$
$A \to b, b$

First(, $) , को First में नहीं लेंगे.
$

Ex:- $S \to AA$
$A \to aA/b$

$S' \to S$
$S \to AA$ ①
$A \to aA/b$ ② ③

Closure निकालते समय look ahead symbol change हो सकता Goto निकालते time नहीं.



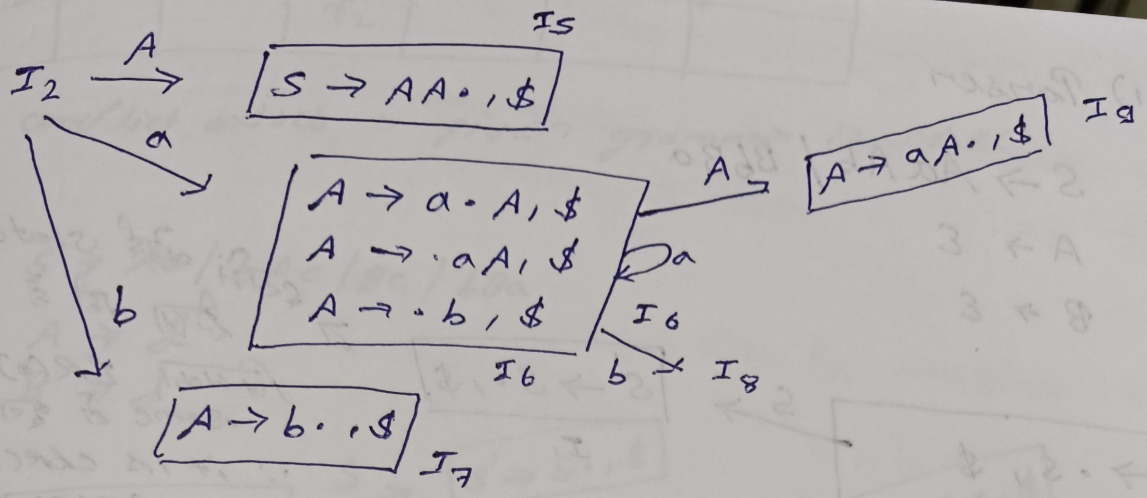


table में कितने स्टेट्स तो reduce grammar वाली look ahead symbol में कितने स्टेट्स।

	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	γ_3	γ_3			
5			γ_1		
6	S_6	S_9			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

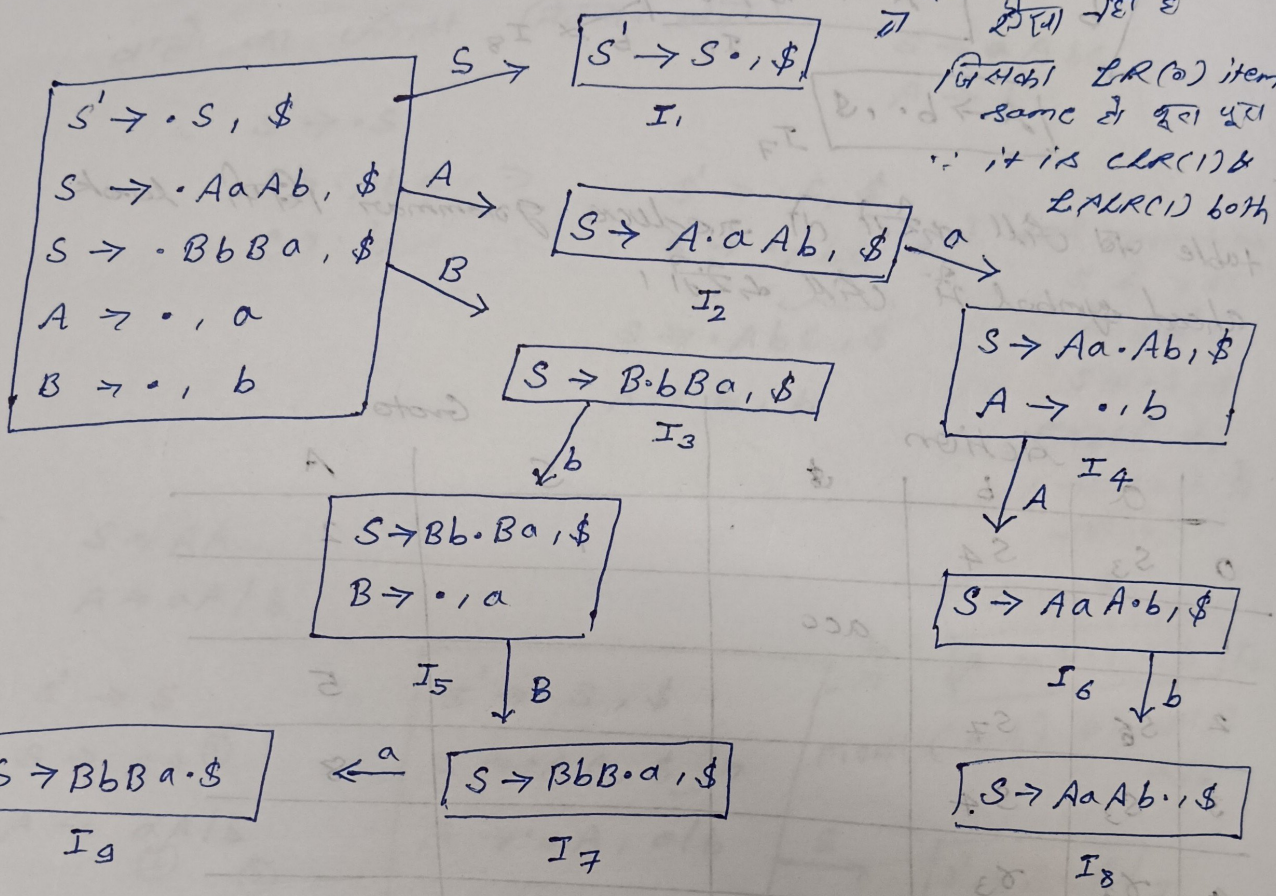
CLR(1) Parsing table

→ CLR(1) Parser

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$



इसमें कोई state शिफ्ट होगा नहीं है जिसका LR(0) item same है पूरा पूरा ∴ it is CLR(1) & LALR(1) both

CLR(1) Parsing table

	Action			Goto		
	a	b	$	S	A	B
0	r_3	r_4		1	2	3
1			acc			
2	S_4					
3		S_5				
4		r_3			6	
5	r_4	~~S_8~~			~~6~~	7
6		S_8				
7	S_9					
8			r_2			

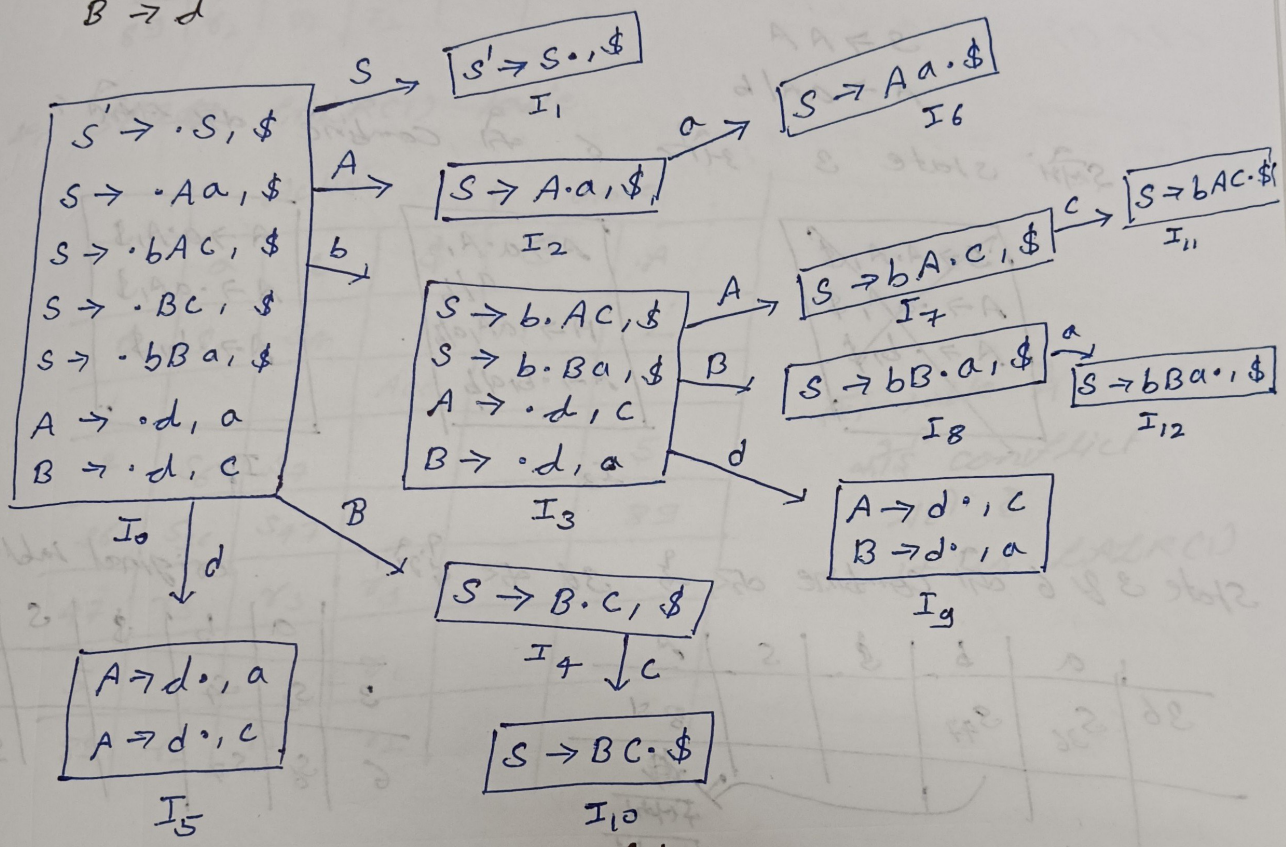
9			r_2		

No conflict arises ∴ given grammar is CLR(1)

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

$A \to d$

$B \to d$



	Action					Goto		
	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					acc			
2	S_6							
3				S_9			7	8
4			S_{10}					
5	r_5		r_6					
6					r_1			
7			S_{11}					
8	S_{12}							
9	r_6		r_5					
10					r_3			
11					r_2			
12					r_4			

conflict नहीं है

∴ given grammar is CLR(1)

⇒ फिर भी LALR(1) table बना सकते हैं state 5 व 9 को merge करके

	a	b	c	d	$
59	r_5/r_6		r_5/r_6		

So, conflict है ∴ LALR(1) ✗

→ LALR(1) Parser → minimise version of CLR(1)

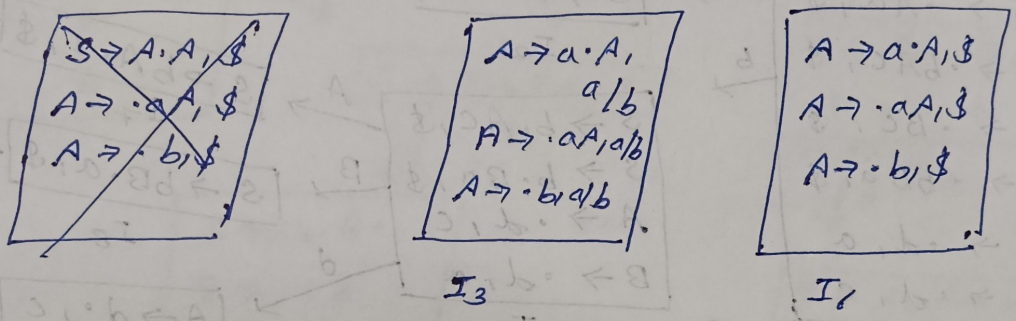
वो state जिनका सिर्फ Look ahead symbol अलग हो बाकी सब same हो उन state को बस combine करेंगे।

Previous question की उत्तर है

$S \to AA$

$A \to aA/b$

इसमें state 3 और 6 को combine कर सकते हैं



State 3 & 6 को combine कर के 36 कर देंगे।

	a	b	$	S	A
36	S_{36}	S_{47}			8

New table

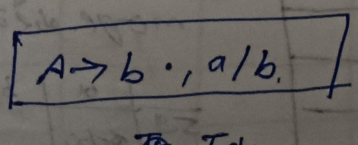
original table

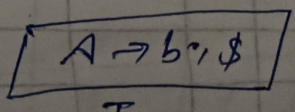
	a	b	$	S	A
3	S_3	S_4			8
6	S_6	S_7			9

इसका मतलब State 4 & State 7 को 47 combine कर सकते हैं

47 State में कोई conflict नहीं आया तब LALR(1) है नहीं तो नहीं है

मतलब 4 & 7 का LR(0) item same होगा





	a	b	$	S	A
4	r_3	r_3			
7			r_3		

89 को 4A merge करना पड़ेगा।

	a	b	$	S	A
47	r_3	r_3	r_3		
89	r_2	r_2	r_2		

Ex, ~~SLR(1)~~ LALR(1) table

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

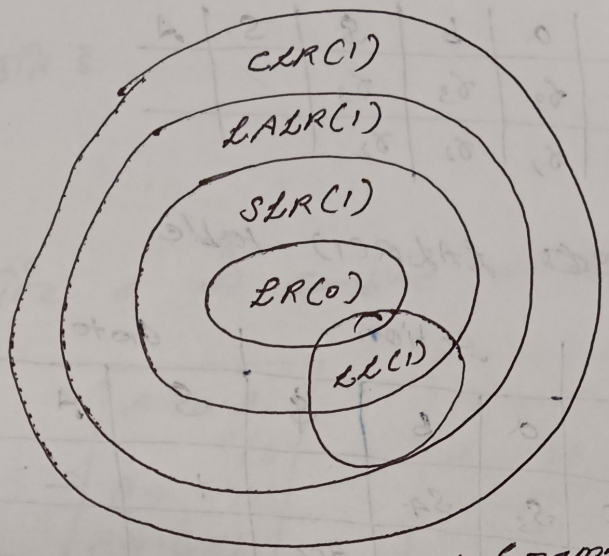
~~इस~~ table में कोई conflict नहीं है
∴ it is LALR(1)

* No. of state in CLR(1) :- n_1
No. of state in LALR(1) :- n_2
then
$n_1 \geq n_2$

→ CLR(1) को ही LR(1) कहते हैं।

* LR(0) : n_1 state
  SLR(1) = n_2 state
  LALR(1) = n_3 state
  CLR(1) = n_4 state

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



unambiguous Grammar

→ Syntax Directed Definition (SDD)

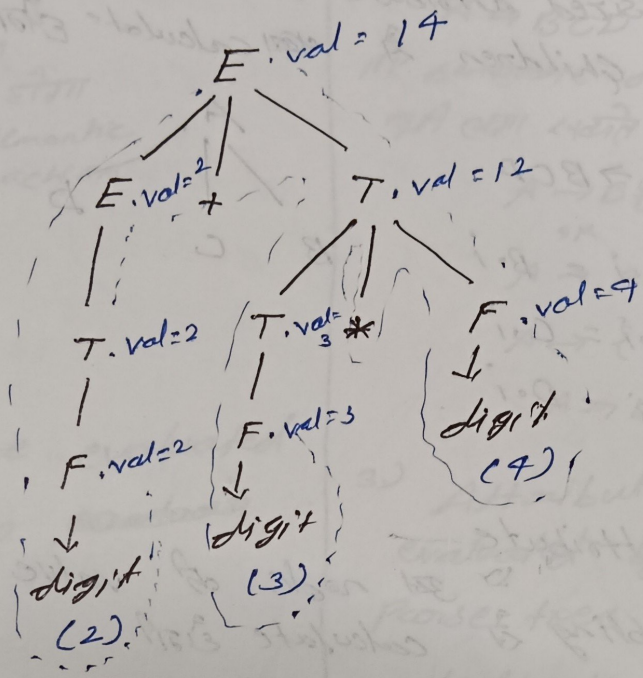
SDD = Grammar + Semantic Rule

Production	Semantic Rule
$E \to E+T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to digit$	F.val = digit.val

1.) Attributes are associated with grammar symbols and semantic rules are associated with productions.

2. Attributes may be number, strings, reference, datatype etc.

2 + 3 * 4



Semantic Rule

→ E → E # T E. val = E. val * T. val

E → T E. val = T. val

T → T & F ⇒ & की Precedence ज्यादा है # because start variable से काफी दूर है। T. val = T. val - F. val

T → F

F → digit T. val = F. val

F. val = digit. val

8 # 12 & 4 # 16 & 12 # 4 & 2

8 × (12 - 4) × (16 - 12) × (4 - 2)

= 8 × 8 × 4 × 2

= 256 × 2

= 512

- ve की Precedence ज्यादा है

* left associative

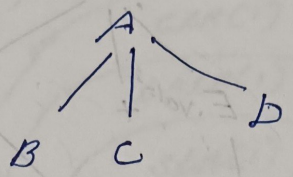
E → E # T

→ Types of SDD

synthesized attribute में उस node की value उसके children से calculate होती

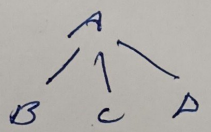
$A \to BCD$

$A.i = B.i$
$A.i = C.i$
$A.i = D.i$



inherited attribute में उस node की value parent या sibling से calculate होती.

$A \to BCD$

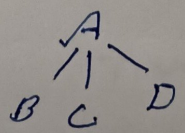


$C.i = A.i$
$C.i = B.i$
$C.i = D.i$

S-Attributed

1. A SDD that uses only synthesized attribute is called S-Attributed SDD

Eg: $A \to BCD$

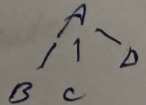


$A.i = B.i$
$A.i = C.i$
$A.i = D.i$

L-Attributed

1) A SDD that uses both synthesised and inherited attributes is called as L-Attributed SDD but each inherited attribute is restricted to inherit from parent or Left sibling only

Eg: $A \to BCD$



$C.i = A.i$
$C.i = B.i$

2.) Semantic action are placed at right end of the production

$A \to BCD \{ \}$

↑ यहाँ होगा Semantic action

3) Attributes are evaluated with Bottom up approach (BUP)

2.) Semantic action are placed anywhere on RHS of the production

$A \to BCD$

में Semantic action कहीं भी रख सकते हैं

$A \to B\{\}CD$

or

$A \to BC\{\}D$

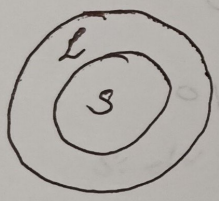
or

$A \to \{\}BCD$

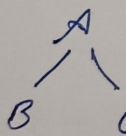
or

$A \to BCD\{\}$

3.) Attributes are evaluated by traversing Parse tree depth first, left to right

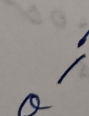


Q1 $A \to BC \quad \{B.i = A.i, C.i = B.i, A.i = C.i\}$



L-Attributed parents & left sibling से calculate हो रहा

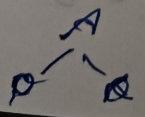
Q2 $A \to QR \quad \{R.i = A.i, Q.i = R.i, A.i = Q.i\}$



~~L-Attribute~~ $Q.i = R.i$ ⇒ right sibling से calculate हो रहा

∴ neither L-attributed nor S-attributed.

Q3 $A \to PQ \quad \{A.i = P.i, A.i = Q.i\}$



S-Attribute & L-attribute both

→ Binary to Decimal

$S \to L \quad \{ S.dv = L.dv \}$

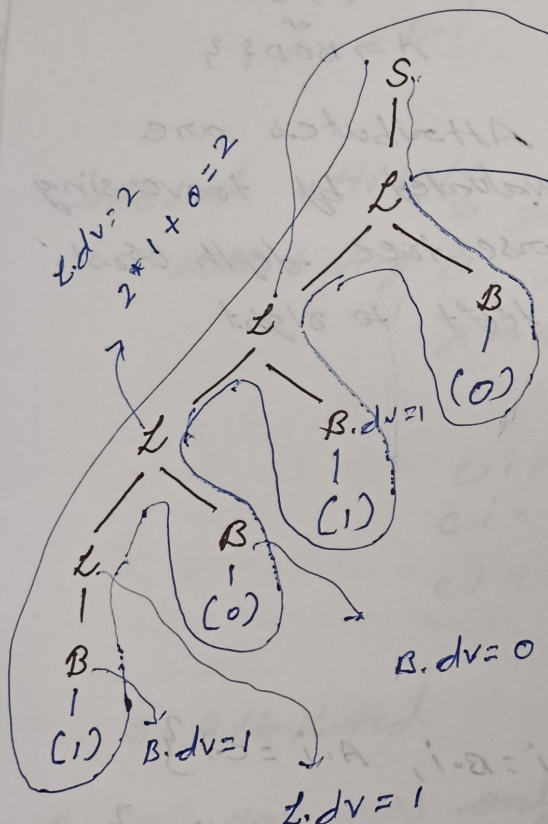
$L \to LB \quad \{ L.dv = 2 * L.dv + B.dv \}$

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

$B \to 0 \quad \{ B.dv = 0 \}$

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

$1010 = 10$



$L.dv = 2 * 2 + 1$

$= 5$

$L.dv = 2 * 5 + 0$

$= 10$

$S.dv = 10$

→ Binary to Decimal with Fraction

$S \to L_1 . L_2 \quad \{ S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \}$

$L \to L.B \quad \{ L.dv = 2 * L.dv + B.dv$

$L.nb = L.nb + B.nb \}$

$nb = no. of bits$

$L \to B \quad \{ L.dv = B.dv$

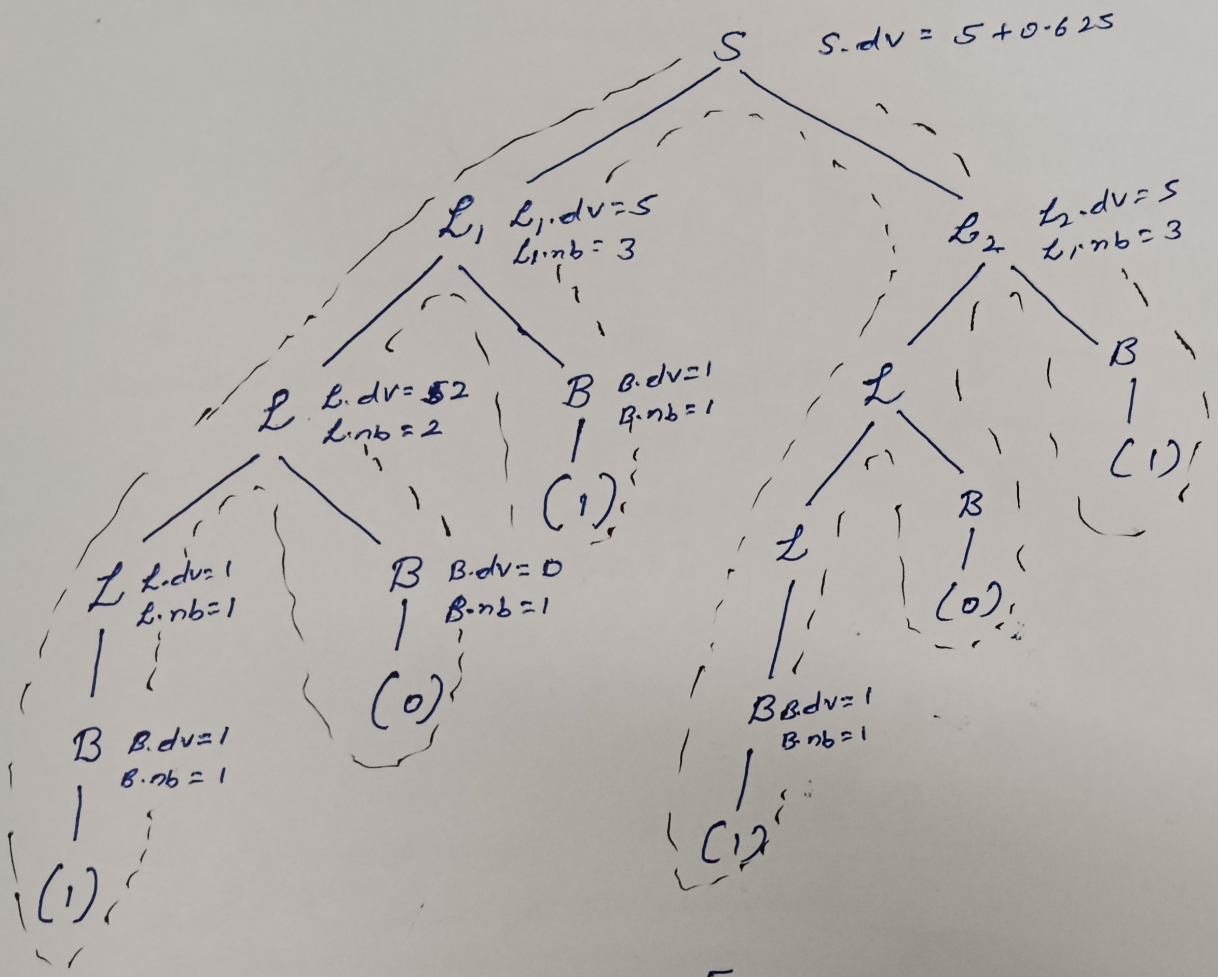
$L.nb = B.nb \}$

$B \to 0 \quad \{ B.dv = 0$

$B.nb = 1 \}$

$B \rightarrow 1 \quad \{ B.dv = 1 \;\; B.nb = 1 \}$

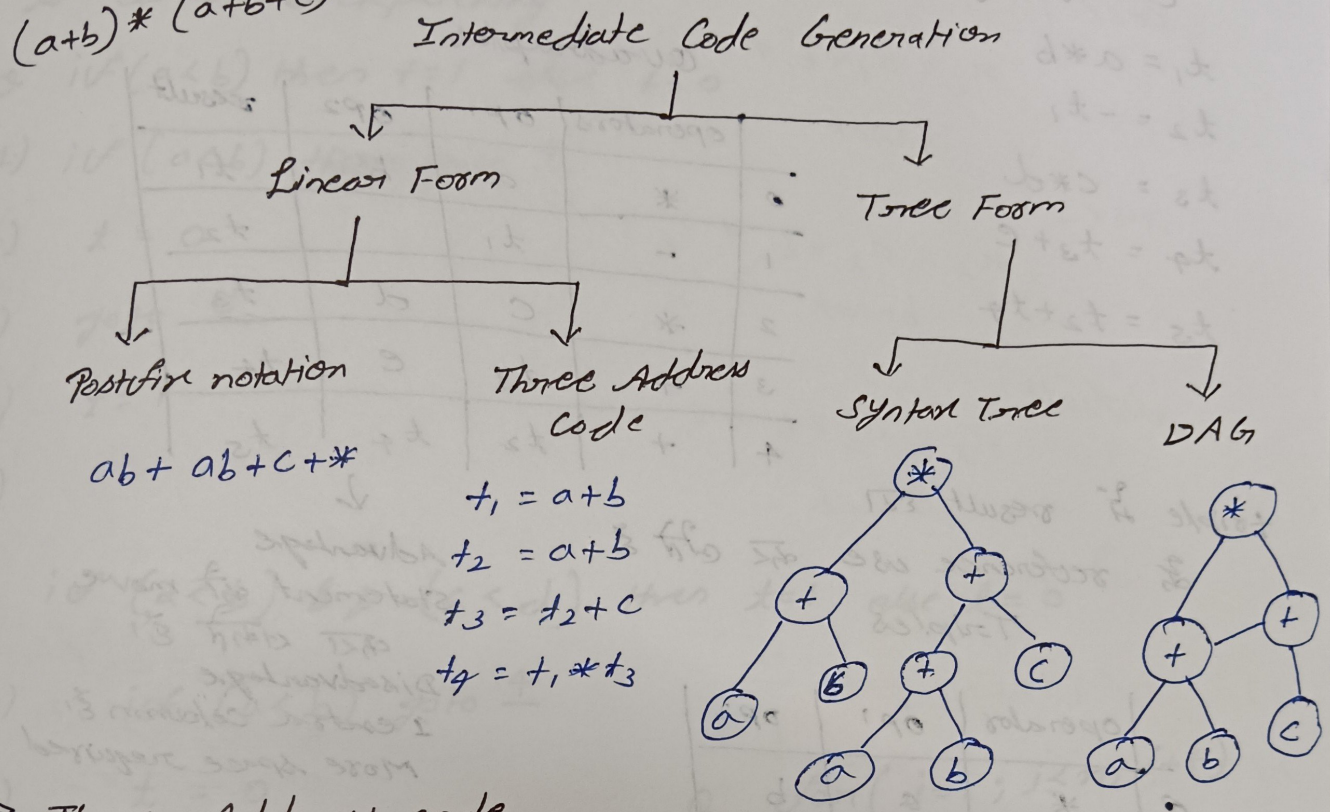
$101.101 = 5.625$



$$S.dv = 5 + \frac{5}{2^3}$$

Intermediate Code Generation

$(a+b) * (a+b+c)$



→ Three Address code

ईस line में 3 से ज्यादा variable नहीं होने चाहिए।

Type of Three Address Code

1) $x = y \text{ op } z$
2) $x = \text{op } z$
3) $x = y$
4) if (x rel op y)
   goto L
5) goto L
6) $A[i] = x$
   $y = A[i]$
7) $x = *P$
   $y = \&x$

$x = a + b * c$

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

} 3-address code

→ Representation of 3 address code

$-(a*b)+(c*d+e)$

$t_1 = a*b$

$t_2 = -t_1$

$t_3 = c*d$

$t_4 = t_3+e$

$t_5 = t_2+t_4$

Quadruples

	operators	op1	op2	result
0	*	a	b	t_1
1	-	t_1		t_2
2	*	c	d	t_3
3	+	t_3	e	t_4
4	+	t_2	t_4	t_5

↓

Advantage

Statement को move करा सकते हैं।

Disadvantage

1 extra column है।

More space required

triple में result हटा के reference use कर लेते हैं

Triples

	operator	op1	op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

⇒ statement को move नहीं करा सकते disadvantage

Advantage:-

It require less space

Move कराने से मतलब है

Quadruple में पहले 2 → 0 → 1 तो answer पर कोई फर्क नहीं आएगा।

Triples में यही नहीं कर सकते because यहाँ dependent है (reference)

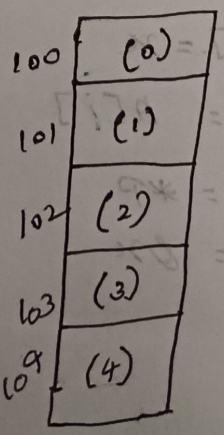
Indirect Triples ⇒ triples के reference को indirectly store कर लेते हैं

instruction

Advantage ⇐ है कि यहाँ move करा सकते हैं statement को।

Disadvantage:

2 memory reference



→ Backpatching

Leaving the lables as empty and filling them later is called backpatching

Q if (a<b) then t=1 else t=0

1) if (a<b) goto 4
2) t = 0
3) goto 5
4) t = 1
5)

Q if (a<b) && (c<d) then t=1 else t=0

1) if (a<b) goto 4
2) t = 0
3) goto 7
4) if (c<d) goto 6
5) goto 2
6) t = 1
7)

OR

1) i = 1
2) if (i>n) goto 8
3) t_1 = b*c
4) t_2 = a+t_1
5) x = t_2
6) i = i+1
7) goto 2
8)

Q for(i=1 ; i≤n; i++)
{
x = a+b*c;
}

1) i = 1
2) if (i≤n) goto 4
3) goto 9
4) t_1 = b*c
5) t_2 = a+t_1
6) x = t_2
7) i = i+1
8) goto 2
9)

Q) switch (i)
{
case 1:
$x_1 = a_1 + b_1 * c_1$;
break;

case 2:
$x_2 = a_2 + b_2 * c_2$;
break;

default:
$x_3 = a_3 + b_3 * c_3$;
break

}

1) if (i == 1) goto 7
2) if (i == 2) goto 11
3) $t_1 = b_3 * c_3$
4) $t_2 = f_1 + a_3$
5) $x_3 = t_2$
6)

case 1
7) $t_1 = b_1 * c_1$
8) $t_2 = a_1 + t_1$
9) $x_1 = t_2$
10) goto 6

case 2
11) $t_1 = b_3 * c_3$
12) $t_2 = a_3 + t_1$
13) $x_3 = t_2$
14) goto 6

Q
```
int A[10], B[10]
int x=0, i;
for (i=0; i<10; i++) {
    x = x + A[i] * B[i];
}
```

2.) $x = 0$

2.) $i = 0$

3) if $(i >= 10)$ goto 15

4) t_1 = base add of A

5) $t_2 = i * 2$

6) $t_3 = t_1[t_2]$

7) t_4 = base add of B

8) $t_5 = i * 2$

9) $t_6 = t_4[t_5]$

10) $t_7 = t_3 * t_6$

11) $t_8 = x + t_7$

12) $x = t_8$

13) $i = i + 1$

14) goto 3

15)

→ $X = A[i][j]$

$t_1 = i * N_c$

$t_2 = t_1 + j$

$t_3 = t_2 * 2$

t_4 = base Add of A

$t_5 = t_4[t_3]$

$X = t_5$

→ Directed Acyclic Graph (DAG)

$$x = \Big(\big((a+a)_{t_1} + (a+a)_{t_2}\big) + \big((a+a)_{t_4} + (a+a)_{t_5}\big)\Big)$$

3 address code

$t_1 = a + a$

$t_2 = a + a$

$t_3 = t_1 + t_2$

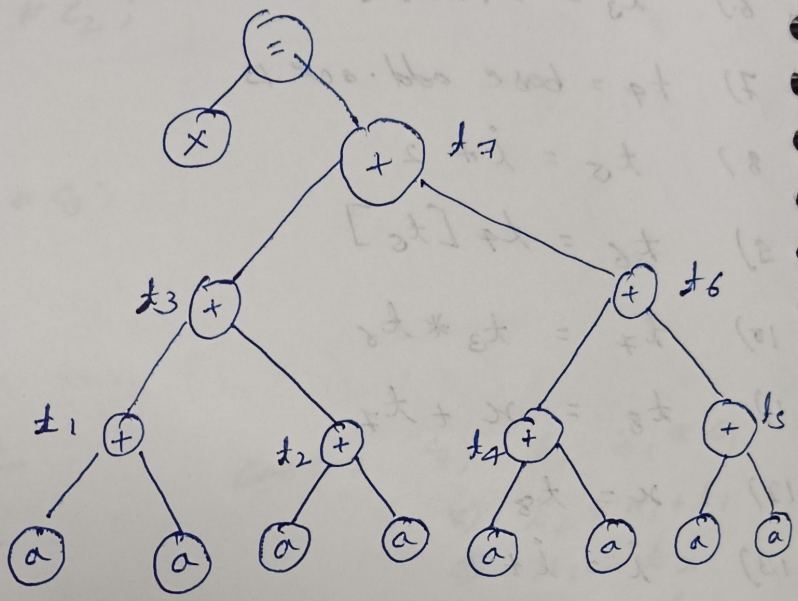
$t_4 = a + a$

$t_5 = a + a$

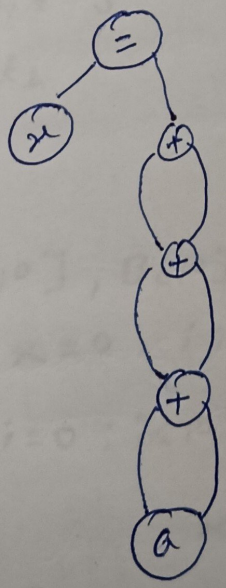
$t_6 = t_4 + t_5$

$t_7 = t_3 + t_6$

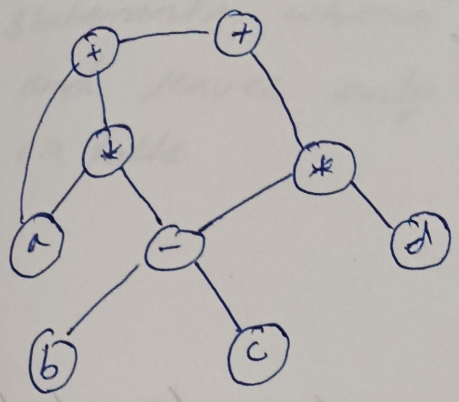
$x = t_7$



इसका DAG

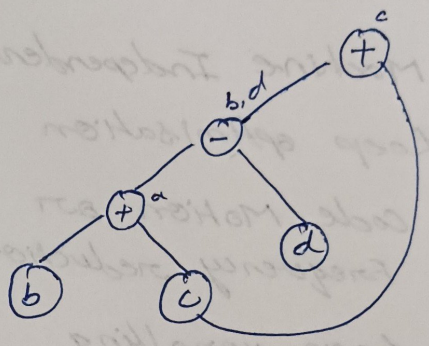


Q $a + a*(b-c) + (b-c)*d$

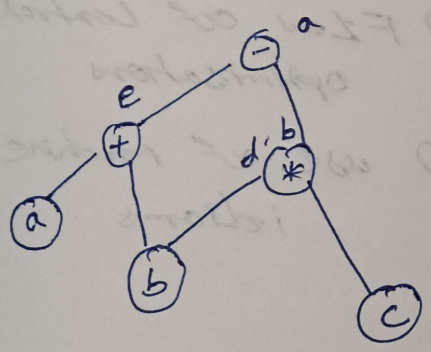


Q
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

top down evaluate करते जाओ



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



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

min, number of nodes, edges

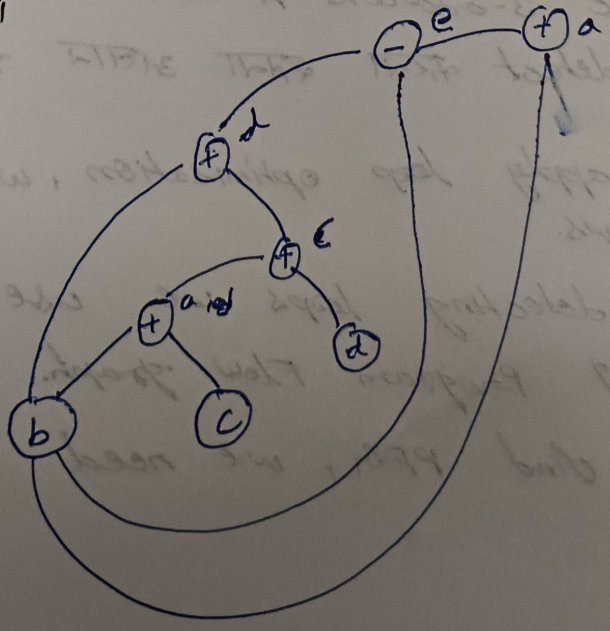
node = 8
edge = 10

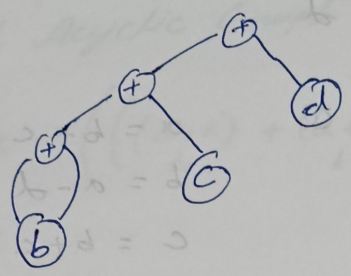
इसको और simplify कर देंगे.

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

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

इसके corresponding DAG design करेंगे.





6 node
6 edge

→ Code optimisation

Machine Independent

1) Loop optimisation
   a) Code Motion or Frequency reduction
   b) Loop unrolling
   c) Loop Jamming
2) Folding
3) Redundancy elimination
4) Strength Reduction
5) Algebraic simplification

Machine dependent

1) Register Allocation
2) use of Addressing Modes
3) peephole optimisation
   a) Redundant LOAD/STORE
   b) Flow of control optimisations
   c) use of machine idioms

→ Loop optimisation
code 3-address में convert हो चुका होगा तो अब में loop detect करना इतना आसान भी नहीं होगा।

1) To apply loop optimisation, we must First detect loops.
2) For detecting loops we use control Flow analysis using Program Flow graph.
3) To find PFG, we need to find Basic Block

6) A basic block is a segment of a machine program from control entry to the beginning and ends only at the end without any jumps in halts