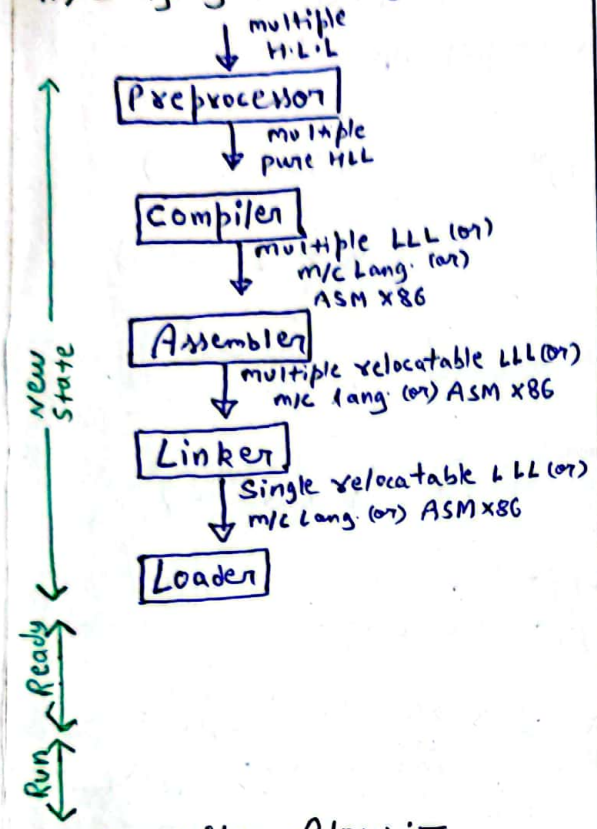
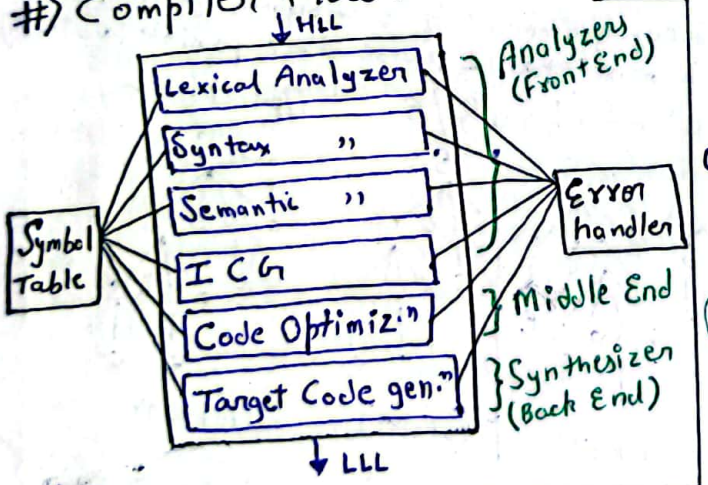
CD

#) Language Processing flow :-



#) Compiler flow :-



#) Example of Compiler flow :-

~~float x, a, b;~~ → declaration time Semantic analyzer will store in Symbol table

S.no	variable name	var type
1	x	float
2	a	float
3	b	float

x = a + b * 60;

↓

(Tokeniz.n) [L A] (or) Scanner

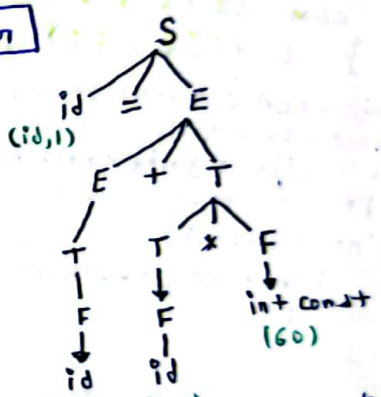
Lexeme	Type of token
x	(id, 1) // identifier
=	(operator, Assignment)
a	(id, 2)
+	(operator, Binary Addition)
b	(id, 3)
*	(operator, Mul)
60	(Int const, 60)

(id,1) = (id,2) + (id,3) * 60 ← Token String

↓ Grammar + Token String

(Physical checking) [Sy. A] (or) [Parser]

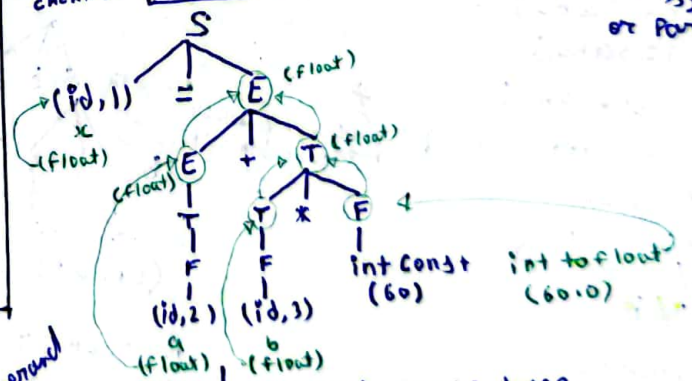
G: S → id=E
E → E+T/T
T → T*F/F
F → id/int const.



Abstract syntax tree or Parse tree

↓ Parse Tree

(Logical checking) [Se. A]



↓ modified parse tree

[ICG]

t_1 = (float) 60
t_2 = b * t_1
t_3 = a + t_2
x = t_3

↓

[Code optiz.n]

t_1 = 60.0
t_2 = b * t_1
x = a + t_2

Target Code Generation

			i.e.
MOV	R1	60.0	R1 = 60.0
MOV	R2	b	R2 = b
MUL	R1	R2	R1 = R1 * R2
MOV	x	a	x = a
ADD	x	R1	x = x + R1

↓ Assembly code

- a & b are memory operand
- R1 & R2 are processor register

Note :) In Java like languages, work is done till ICG & byte code is generated, i.e. front end only. You can create backend on any m/c

Note :) Interpreter: Without compilation directly running. Taking line by line & run. Takes more time than Compilation + Running. Give more errors

*) Find Tokens/Lexemes :-

```
int main ( )        => 4
{ /*Comment*/       => 1
int a , b ;         => 5
if ( a > b )        => 6
      pf ( "a" ) ;  => 5
else  pf ( "b" ) ;  => 6

int a = 'b' ;       => 5
char c = "abc" ;    => 5
int c = 'd' ;       => 6
i/*Comment*/nt c = 'e' ;   => 6
for ( i=1 ; i<=n ; i++ )   => 13
int *** a = 10.5 ;  => 8
char && && d = 20.7 ;  => 7
      fi(x == 5) }  => 7
```

Note:) • LA eliminates comments, white space characters.
• LA helps in giving error messages by providing row no. & col. no.
• int f(int n)
  { return (n*f(n-1)); }

Recursive program => without termination => Stack overflow Error => Runtime Error

•
a = b /*c+d-e ; (Never Ending comment)
char c = "abc ; (" " String)
int 123abc ;
"abc" "def" "gh" ;

Lexical Error Examples → Length of variable → Start with digit

• a /*
~~~~*
~~~~*/
~~~~*/ b

#) Grammar :-

*) G(V, T, P, S)
V = Variables = {S, A, B}
T = Terminals = {a, b}
S = Start = {S}
P = Productions = S → AB
A → a/ε
B → b

*) LHS single variable, RHS anything => CFG (T-2)

LHS single variable, RHS some conditions => RG (T-3)

*) Type-0 → Unrestricted Grammar
Type-1 → CSG → |α| ≤ |β|
Type-2 → CFG → |α| = 1 non-terminal
Type-3 → RG → |α| = |β| = 1 non-terminal

*) α → β
α ⇓ (V+T)⁺ , β ⇓ (V+T)* → T-0 (UG)
Atleast one V required

{① should be T-0, ② |α| ≤ |β| → T-1 (CSG)

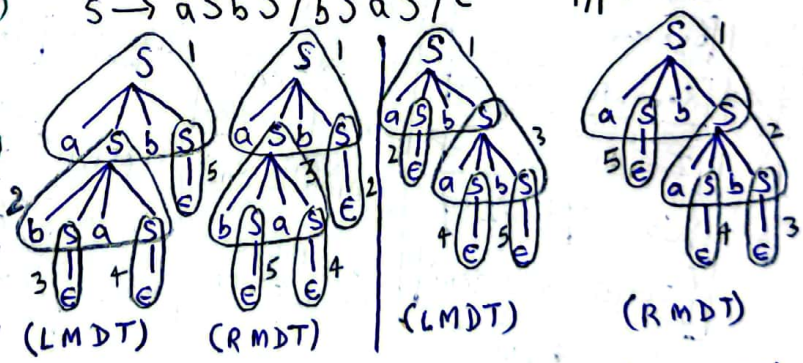
{① Should be T-1, ② α → β → T-2 (CFG)
α ⇓ Single Var. , β ⇓ Anything

{① Should be T-2, ② α → β → T-3 (RG)
α ⇓ Single Var. , β ⇓ VT*/T* → LLG
T*V/T* → RLG

*) Examples -
Ex S → aS/a ⇓ RLG ⇓ RLG => T-3
Ex S → a ⇓ RLG✓ LLG✓ => T-3
Ex A → bcd, e → ABc, α ≠ V => None type

*) Ambiguous grammar : G is ambiguous iff ∃w ∈ G to which more than 1 parse tree possible (w = string)

S → aSbS/bSaS/ε i/p: abab



No. of LMDT = No. of RMDT = No. of Parse tree = 2 => Ambiguous
If more than 1 parse tree => Ambiguous grammar

Note:) Since we scan string left to right, so practically only LMDT is possible, but theoritically RMDT also possible.

Note:) Itself is LMDT & itself is RMDT if only 1 choice
Ex:) S → aS/Sa/a

Note:) In grammar if everyone on same level => Ambiguous
Ex:) E → E+E/E*E/id i/p: id+id*id

Note:) checking Unambiguity is undecidable problem.

Note:) (T-3) (T-2) (T-1) (T-0) Chomsky hierarchy true when null productions (ε) not there.

Note: Examples of Semantic errors:
→ Multiple declarations
→ Type mismatch
→ Not declaring a variable

CamScanner
~~~~

*) Ambiguous Grammar → Unambiguous Grammar

Ex:)

Symbol	Priority	Associativity
/ , ⊕	1 (L)	L-R
* , &	4 (H)	R-L
− , %	3	L-R
+ , ⊙	2	R-L

$A \to A/B \,/\, A \oplus B \,/\, B$

$B \to C+B \,/\, C \odot B \,/\, C$

$C \to C\%D \,/\, C-D \,/\, D$

$D \to E*D \,/\, E\&D \,/\, E$

$E \to id \,/\, (A)$

*) Indirect Recursion: -

Ex $S \to Sa \mid Sb \mid Aa \mid c \mid d$ | $S \Rightarrow A_1$, $A \Rightarrow A_2$

$A \to Sc \mid b \mid d$

$A_1 \to A_1a \mid A_1b \mid A_2a \mid c \mid d$ $\Rightarrow$ (Removing direct recursion) $A_1 \to A_2aA_1' \mid cA_1' \mid dA_1'$, $A_1' \to aA_1' \mid bA_1' \mid \epsilon$

$A_2 \to A_1c \mid b \mid d$ (L>R) $\Rightarrow$ (Removing indirect rec. by substitution) $A_2 \to A_2aA_1'c \mid cA_1'c \mid dA_1'c \mid b \mid d$

$\Rightarrow A_2 \to cA_1'cA_2' \mid dA_1'cA_2' \mid bA_2' \mid dA_2'$

$A_2' \to aA_1'cA_2' \mid \epsilon$

Now, $A_1 \Rightarrow S$ $A_2 \Rightarrow A$ (final answer)

$S \to Aas' \mid cs' \mid ds'$

$s' \to as' \mid bs' \mid \epsilon$

$A \to cs'cA' \mid ds'cA' \mid bA' \mid dA'$

$A' \to as'cA' \mid \epsilon$

Ex $S \to Ac \mid d$ | $S \Rightarrow A_1$, $A \Rightarrow A_2$

$A \to Sa \mid b$

$A_1 \to A_2c \mid d$ ✓

$A_2 \to A_1a \mid b$ (L>R) $\Rightarrow A_2 \to A_2ca \mid da \mid b$

$\Rightarrow A_2 \to daA_2' \mid bA_2'$

$A_2' \to caA_2' \mid \epsilon$

Note: Inherently ambiguous grammar:
Grammar for which we can't remove ambiguity
Ex:) $L = \{a^m b^n c^n \cup a^m b^m c^n \mid m,n \geq 1\}$

*) Recursion: $S \to (V+T)^* S (V+T)^* / T^*$

- Left Recursion
  $S \to S(V+T)^* / T^*$
  Elimination required cuz. top-down parser (recursive decent parser) may go to infinite loop (Stack overflow)
- Right Recursion
  $S \to (V+T)^* S / T^*$
  No need to eliminate

- Left recursion elimination by EBNF:

Ex $E \to E+T \mid T \Rightarrow T \mid T+T \mid T+T+T \cdots \Rightarrow T(+T)^* \Rightarrow T\{+T\}$

$T \to T*F \mid F \Rightarrow F \mid F*F \mid F*F*F \cdots \Rightarrow F(*F)^* \Rightarrow F\{*F\}$

$F \to id$

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

Left recursion eliminated but grammar not CFG (T-2) bec. of brackets { }.

- Left recursion elimination by CFG form:

Ex $E \to E+T \,/\, T \Rightarrow T \mid T+T \mid T+T+T \mid \cdots \Rightarrow E \to TE'$, $E' \to +TE' \mid \epsilon$

$T \to T*F \,/\, F \Rightarrow F \mid F*F \mid F*F*F \mid \cdots \Rightarrow T \to FT'$, $T' \to *FT' \mid \epsilon$

$F \to id$

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

using II solve I

Ex $S \to Sa \,/\, b \,/\, c \,/\, d$ (I, II, III, IV) $\Rightarrow b \mid ba \mid bba \mid \cdots \Rightarrow S \to bS'$ $S' \to aS' \mid \epsilon$

$S \to cS'$

$S \to dS'$

using II, III, IV solve I

$S \to bS' \mid cS' \mid dS'$	
$S' \to aS' \mid \epsilon$	

Ex $S \to Sa \mid Sb \mid Sc \mid d \mid e \mid f$

$S \to dS' \mid eS' \mid fS'$

$S' \to aS' \mid bS' \mid cS' \mid \epsilon$

or $S \to dS' \,/\, eS' \,/\, fS' \,/\, d \,/\, e \,/\, f$

$S' \to aS' \,/\, bS' \,/\, cS' \,/\, a \,/\, b \,/\, c$

Avoid doing like this

Ex $A \to ABd \,/\, Aa \,/\, a$ (A', A')

$B \to Be \,/\, b$ (B')

$A \to aA'$

$A' \to BdA' \mid aA' \mid \epsilon$

$B \to bB'$

$B' \to eB' \mid \epsilon$

CamScanner

*) Left factoring: Compiler confused which production to take, bec. more than 1 choice

Ex $S \to \underline{a}b \mid \underline{a}c \mid \underline{a}d$ i/p: ad

$\Rightarrow S \to aS'$
$S' \to b/c/d$ } Left factored grammar (Everywhere 1 choice)

Ex $S \to \underline{iEt}S / \underline{iEt}SeS / a / b$
$E \to b$

$\Rightarrow S \to iEtSS' / a / b$
$S' \to eS / \epsilon$
$E \to b$

Ex $S \to \underline{a} / \underline{ab} / \underline{abc} / \underline{abcd} / b / c / d$

$\Rightarrow S \to aS' / b / c / d$
$S' \to bS'' / \epsilon$
$S'' \to cS''' / \epsilon$
$S''' \to d / \epsilon$

Note:) Grammar which don't have Left Rec., Left factoring, ambiguity, is the perfect parser.

#) Parser:

*) Parsing: Deriving string from grammar.

*) Parser

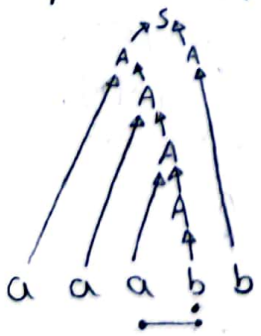
Top Down	Bottom up
• Start symbol to string.	• String to start symbol
• LMD used	• R-RMD used
• Difficulty: Selecting correct production	• Difficulty: Selecting correct handle (Substring to compress)
• Sol^n: Recursive Descent parser, LL(1) parser (Best top down parser)	• Sol^n: LR(0), SLR(1), LALR(1), CLR(1) (best BUP)

• CLR(1) more powerful than LL(1)

Note:) working of BUP

$S \to AA$
$A \to aA / b$ → compression

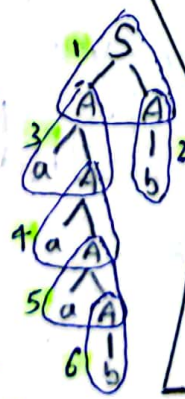
i/p: aaabb$



1	⑥	$S \to AA$
2	⑤	$A \to b$
3	④	$A \to aA$
4	③	$A \to aA$
5	②	$A \to aA$
6	①	$A \to b$

Stack

Execution order used by BUP



R-RMD

*) Top down parser

1. with Backtracking ⇒ Recursive descent parser
2. without Backtracking ⇒ LL(1) parser (or) Non-Recursive descent parser (or) Predictive parser

1.) Recursive descent parser:

$S \to aS / b$
i/p: aaab$
↑ LAS

```
S(){
  Select any prod.n of S;
  for(i=1; i≤n; i++)
  {if(xi == some variable) xi();
   else if (xi == LAS) inc. i/p pointer
   else error (Backtrack)}
}
```

S()
$S \to b$ backtrack | $S \to aS$
i=1 inc.
i=2 S()
$S \to aS$
i=1 inc.
i=2 S()
$S \to b$
i=1 inc.

Note:) Here, $S \to x_1, x_2, x_3 \cdots x_n$

Note:) In Recursive descent parser:-
- Every variable have same recursive code (fx.^{n})
- Backtracking wastes lot of time (wc $T(2^n)$)
- If G has left recursion, it may goto infinite loop.
- ——— left factoring, it may produce parsing error.
- we're using stack DS.

2.) LL(1) parser: Left to Right Scanning, LMD, 1 symbol at a time (in scanning).
⇒ No backtracking cuz. we've LL(1) parse table.

Note:) First(S): Ex
$S \to aBDh$
$B \to bC / \epsilon$
$C \to cC / \epsilon$
$D \to EF$
$E \to e / \epsilon$
$F \to f / \epsilon$

	Fi()	Fo()
S	a	$
B	b, ε	f, e, h
C	c, ε	f, e, h
D	e, f, ε	h
E	e, ε	h, f
F	f, ε	h

Follow(S): • Terminals immediately to right of S.
• can never contain ε
• Follow(Start) = $

• $A \to BCXEF$ $Fo(X) = Fi(EF) = Fi(E) = Fi(gh) = g$
$E \to gh$
• $A \to BCX$ $Fo(X) = Fo(A)$
• $A \to BCXD$ $Fo(X) = Fi(D) = \epsilon \Rightarrow Fo(A)$
$D \to \epsilon$
• $A \to BCD$, $E \to GHI$ — nowhere E ⇒ $Fo(E) = \phi$

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

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

CamScanner

LL(1) parsing table construction:

for each production $A \to \alpha$:-

① Add $A \to \alpha$ at $m[A,b]$; $b \in F_i(\alpha)$

② Add $A \to \alpha$ at $m[A,c]$; $c \in F_o(\alpha)$ if $F_i(\alpha) = \epsilon$

Ex Check given grammar LL(1)?

$E \to TE'$ ①

$E' \to \epsilon \mid +TE'$ ② ③

$T \to FT'$ ④

$T' \to \epsilon \mid *FT'$ ⑤ ⑥

$F \to id \mid (E)$ ⑦ ⑧

m	+	*	(	)	id	$
E			①		①	
E'	③			②		②
T			④		④	
T'	⑤	⑥		⑤		⑤
F			⑧		⑦	

No entry contain multiple prod^{n}s ⇒ LL(1) grammar ✓

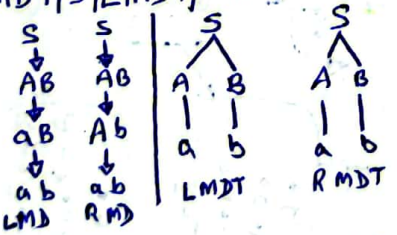
Note:) • E contain only 1 production, so in E row no multiple productions can come in one cell.

• E' contain more than 1 prod^{n}s, so in E' row multiple prod^{n}s may come in one cell.

Note:) • Gaps in table shows parsing error. like m[E, *] means parser saying I don't know.

Note:) • LMD & RMD may differ even if |RMDT| = |LMDT|

ex
$S \to AB$
$A \to a$
$B \to b$
i/p: ab



Ex Check given grammar LL(1)?

$S \to AaAb \mid BbBa$ ① ②

$A \to \epsilon$ ③

$B \to \epsilon$ ④

	$F_i()$	$F_o()$
A	ϵ	a, b
B	ϵ	a, b
S	a, b	$

$F_i(S) = F_i(AaAb) \cup F_i(BbBa) = F_i(ab) \cup F_i(ba) = \{a, b\}$

m	a	b	$
S	①	②	
A	③	③	
B	④	④	

LL(1) ✓

To check given grammar LL(1) without table:

① If grammar don't have null productions

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

$F_i(\alpha_1) \cap F_i(\alpha_2) = \phi$ &&

$F_i(\alpha_2) \cap F_i(\alpha_3) = \phi$ &&

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

i.e. All pairwise disjoint ⇒ LL(1) ✓

② If grammar contain null productions

In $A \to \alpha_1 / \alpha_2 / \epsilon$

$F_i(\alpha_1)$ $F_i(\alpha_2)$ $F_o(\epsilon)$

All pairwise disjoint ⇒ LL(1) ✓

Note:) Ex production के लिए अलग-अलग check करना है ① या ②

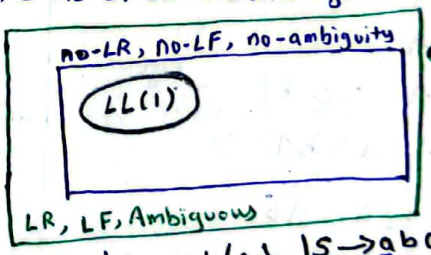
Ex $S \to E / a$ $\{F_i(E) = b\} \cap \{F_i(a) = a\} = \phi$

$E \to b$ 1 prod^n ⇒ no problem = ϕ

LL(1) ✓

Note:)
1) Ambiguous G can never be LL(1)
2) G with left recursion can never be LL(1)
3) " " left factoring " " " "
4) Every RG need not be LL(1)
5) Every RL is LL(1), LR(1) & unambiguous
6) Every DCFL need not be LL(1), but it is LR(1) & unambiguous
7) Some CFL's are always ambiguous (inherently ambiguous)

Note:)



All Grammars

Note:) $S \to a$ LL(0)

$S \to ab / cd$ LL(0) ✗ LL(1)

$S \to abc / abd$ LL(0) ✗ LL(1) ✗ LL(2) ✗ LL(3)

LL(1) LL(2) . . . LL(K)

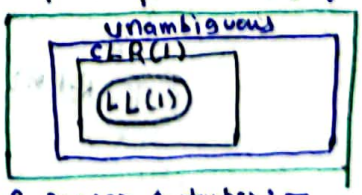
*) Bottom up parser: All BUP are without backtracking cuz. we've table

1. Operator precedence parser
2. LR parser (or Shift-Reduce SR parser):

→ Scanning i/p L to R, using R-RMD

→ LR parser possible only if G is unambiguous

→



All Grammars

→ LR parser 4 types:-

• LR(0) Basic • SLR(1) Simple • LALR(1) LookAhead • CLR(1) or LR(1) canonical

Power & Cost →

Note:) For all LR parsers, parsing algo same, parsing table may differ.

•) LR(0) parsing algorithm:

Ex: $S \to AA$ ①
$A \to aA/b$ ② ③

i/p: abab$

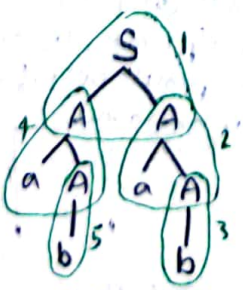
LR(0) parse table →

	Action a	Action b	Action $	Goto S	Goto 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		

viable prefixes

i/p String	Stack	Reductions
abab$	0 ⇒ 0a3	
bab$	0a3 ⇒ 0a3b4	
ab$	0a3b4 ⇒ r_3 (Replace with A)	A → b
	0a3A6 [Goto(3, A)] = 6	
ab$	0a3A6 ⇒ r_2	A → aA
	0A2	
ab$	0A2a3	
b$	0A2a3b4	
$	0A2a3b4 ⇒ r_3	A → b
	0A2a3A6	
$	0A2a3A6 ⇒ r_2	A → aA
	0A2A5	
$	0A2A5 ⇒ r_1	S → AA
	0S1	
$	0S1	acc

5	S → AA	1
4	A → aA	2
3	A → b	3
2	A → aA	4
1	A → b	5



R-RMD

Note:) • BUP table size ≅ 2 × TDP table size.
• viable prefixes: All prefixes present in stack.
• Action contain terminals, goto contains variables.

•) LR(0) parsing table construction:

Note:)
•) G: S → AA
A → aA/b

Augmented G': (augmented prod^n)
S' → S
S → AA
A → aA/b

•) S → •xyz
x•yz
xy•z
xyz•
} LR(0) items; xyz• — Reduced/completed/final LR(0) item

•) closure(S → A•A)
S → A•A
A → •aA
•b
• Terminal, so stop

closure(S' → •S)
S' → •S
S → •AA
A → •aA
•b

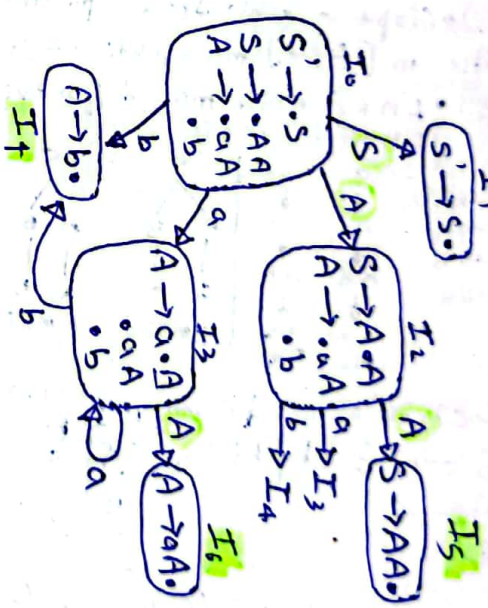
•) Goto(S → •AA, A)
S → A•A
A → •aA
•b

Goto(A → •aA, a)
A → a•A
•b

Goto(A → •b, b)
A → b•

Ex-1 construct LR(0) PT
G: S → AA ①
A → aA/b ② ③

G': S' → S
S → AA
A → aA/b



In LR(0) PT no shift/Reduce clash, so LR(0) ✓

	action a	action b	action $	Goto S	Goto 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		

Blindly reduce written everywhere.

⇒ SLR(1) ✓
⇒ LALR(1) ✓
⇒ CLR(1) ✓

Note:) Inadequative State (or) Conflicting State :-

- At least 2 productions
- At least 1 Reduce (other than augmented prod.n)

Note:)

- LR(0), SLR(1), LALR(1) → may ✓ may ✗

LL(1) ✓ pass ⇒ • CLR(1) ✓

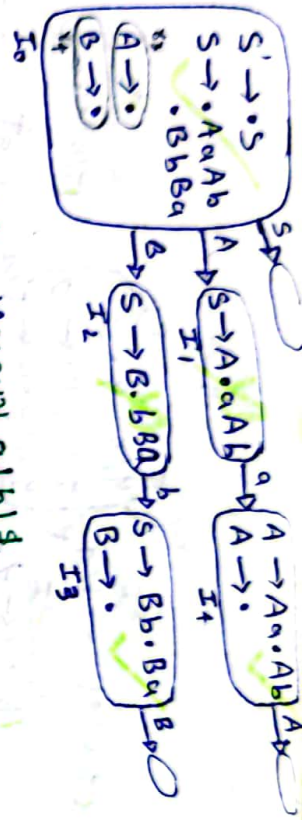
LL(1) ✗ fail ⇒ LR(0), SLR(1), LALR(1), CLR(1) → may ✓ may ✗

CLR(1) LL(1)

EX-3 ~~E → E+T/T~~ ~~T → TF/F~~ ~~F → F*/a~~ $S \to AaAb / BbBa$

$A \to \epsilon$ SLR(1)?

$B \to \epsilon$



LR(0)	a	b	$
0	r2	r2	r2
3		r3	r3
4			r4

SLR(1)	a	b	$
0	r2		
3		r3	
4			r4

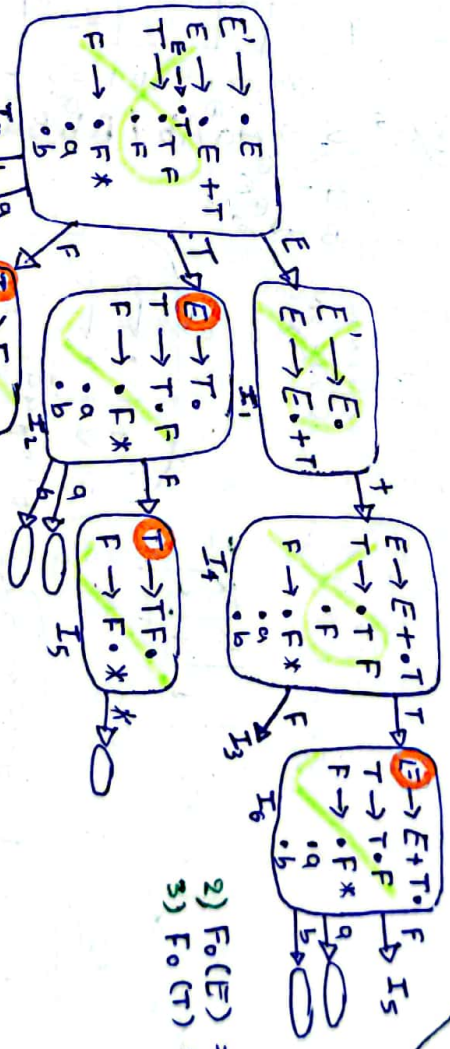
F0(A) = a, b

F0(B) = a, b

EX-2 $E \to E+T / T$ SLR(1)?

$T \to TF / F$

$F \to F* / a / b$



LR(0)	*	+	a	b	$	E	T	F
0			S1/r4	S1/r4				
1		r1			r1			
2	r2	r2	S1/r4	S1/r4	r2			
3	S4/r4	r4			r4			
4			S1/r4	S1/r4				
5	S4/r4	r4			r4			
6		r3			r3			

⇒ 2-SR ⇒ 1-SR ⇒ 1-SR ⇒ 2-SR

⇒ LR(1) ✗ G-SR 4-Inadequative States

SLR(1)	*	+	a	b	$	E	T	F
0			S1	S1				
1		r1			r1			
2	r2	r2	S1	S1	r2			
3	S4	r4			r4			
4			S1	S1				
5	S4	r4			r4			
6		r3			r3			

⇒ SLR(1) ✓ LALR(1) ✓ CLR(1) ✓

2) $F_0(E) = \$, +$

3) $F_0(T) = F_0(E) = \$, +$

$= F_0(F) = a, b$

Note:)

- If no one going out by reading terminal ⇒ No shift there ⇒ No S-R clash
- If 1 reduce only ⇒ no R-R clash

In this way I_3, I_4 can be eliminated

• CLR(1) & LALR(1) PT constn :-

Note:) $S \to AA$

$A \to aA / b$

$S' \to •S, \$$

LR(0) item LAS

LR(1) item

LR(0) items → LR(0), SLR(1)

LR(1) items → CLR(1), LALR(1)

minimized CLR(1) is another name of LALR(1)

closure $(S' \to •S, \$)$

① $S' \to •S, \$$

② $S \to •A, \$$ // F: (, $)

③ $A \to •aA, a/b$ // F: (A, b)

$•b, a/b$

Goto $(S \to •AA, c/d, A)$

LR(1) item T/P

$S \to A•A, c/d$

$A \to •aA, c/d$

$•b, c/d$

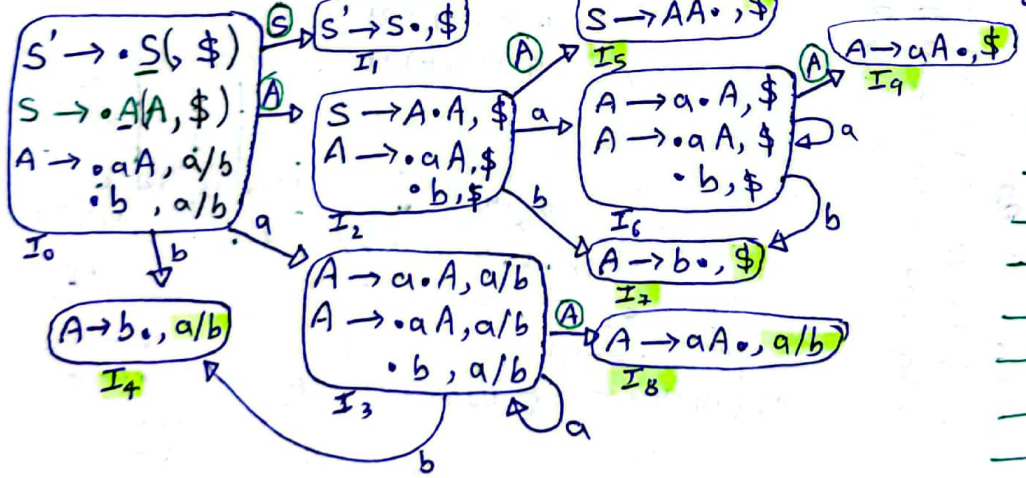
• ⇒ LALR(1) PT Construction :-

In Ex-4; merge (4,7), (8,9), (3,6) cuz. they only differ by LAS

LALR(1)	a	b	$	S	A
0	S3 6	S4 7		1	2
1			acc		
2	S6	S7			5
3 6	S36	S47			8 9
4 7	γ3	γ3	γ3		
5	γ1	γ1	γ1		
6	S6	S7			0
7			γ3		
8 9	γ2	γ2	γ2		
0					

Ex-5 S → ⊖AA
A → ⊕aA / ⊕b

CLR(1)?



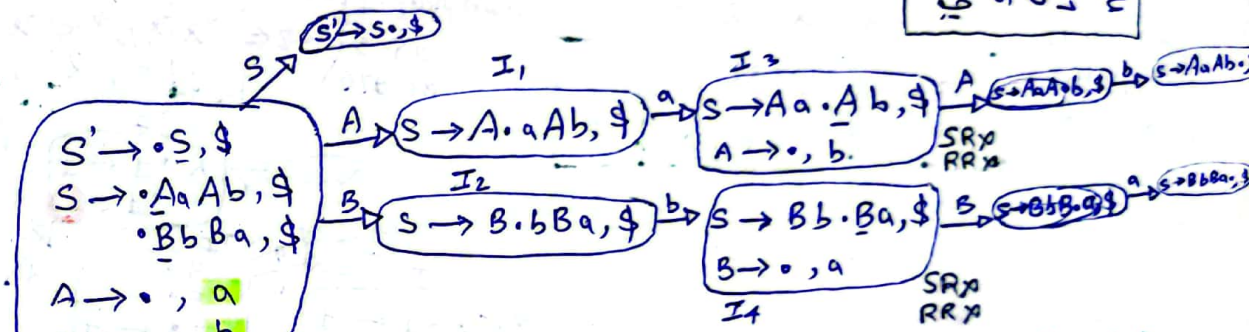
CLR(1)	action a	action b	action $	goto S	goto A
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	γ3	γ3			
5			γ1		
6	S6	S7			9
7			γ3		
8	γ2	γ2			
9			γ2		

CLR(1) PT

no conflict
⇒ CLR(1) ✓

Note:) Minimiz.n Algo on CLR(1) gives LALR(1) i.e. if 2 states differ by only LAS merge them

Note:) if n_1 = LR(0) states, n_2 = SLR(1), n_3 = CLR(1), n_4 = LALR(1)
$n_1 = n_2 = n_4 \le n_3$



SR✗
RR✗

SR✗
RR✗

	action a	action b	action $
0	γ3	γ4	

⇒ CLR(1) ✓

No two state differ by LA
⇒ Already minimized
⇒ LALR(1) ✓

Ex-5 S → ⊖AaAb / ⊖BbBa
A → ⊕ε
B → ⊕ε

No conflict ⇒ LALR(1) ✓

LALR(1)?

No one going out by reading terminal ⇒ Shift ✗ ⇒ SR✗

RR✓ ⇒ LR(0) ✗
SLR(1) ✗

Note:)

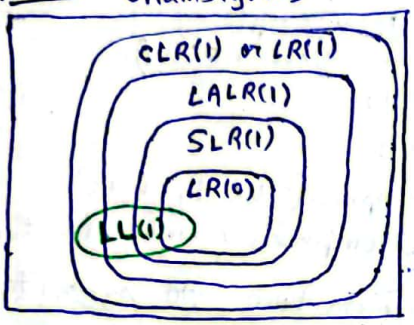
① If CLR(1) parser don't have RR-conflict, then LALR(1) parser may contain RR-conflict

② If CLR(1) don't have SR-conflict, then LALR(1) also don't have SR-conflict.

③ If G is CLR(1) but not LALR(1)
⇒ LALR(1) got RR-conflict

④ If G is not LALR(1) due to SR-conflict
⇒ It's not CLR(1), cuz CLR(1) also have SR-conflict

Note:) Unambiguous :



Note:) • Every DCFL is unambiguous
• Every DCFL is LR(1), but it need not be LL(1).

•) Operator precedence parser: Needs G to be operator grammar (i.e. ① No null prⁿ's ② No 2 adjacent variable on RHS of prⁿ like $S \to AB$).
• This parser sometimes possible even if G is ambiguous
• convert G to operator Grammar:

$S \to SAS / a$ → $S \to SbAbS / SbS / a$
$A \to bAb / b$ → $A \to bAb / b$

#) Syntax Directed Translation (SDT):

*) Basics: • SDT = G + Semantic action
• Synthesized att. : Att. value calculated using children att. value.
• Inherited att : Att. value calculated using sibling or parent att. value
• SDT Definition :—

① S-attributed : → Use only Synthesized att. or no att.
→ uses bottom up evaluation.
→ Semantic action should be placed on RHS of rightmost production

② L-attributed : → Use both Synthesized & inherited but use of Inherited right sibling att. not allowed.
→ uses top down evaluation (Dependency graph & Topo. Sort)
→ semantic actions can be placed anywhere on RHS.

L-att.
S-att.

• SDT example:
$S \to ABC \{S.val = A.val\}$ → Syn
$A \to CD \{D.val = C.val\}$ → Inh-Left
] L-att.

*) Applications of SDT :—
Constructing SDT for :—

① Evaluation of Arithmetic Exp. :

$E \to E+T \{E.val = E.val + T.val\}$
$\quad | T \quad \{E.val = T.val\}$
$T \to T*F \{T.val = T.val * F.val\}$
$\quad | F \quad \{T.val = F.val\}$
$F \to id \{F.val = id\}$ val ⇒ Syn att. ⇒ L-att ✓ S-att ✓

② Infix to postfix conversion :
i/p: a+b*c
o/p: abc*+

$E \to E+T \{pf(+)\}$ no att. ⇒ L ✓ S ✓
$\quad | T$
$T \to T*F \{pf(*)\}$
$\quad | F$
$F \to id \{pf(id)\}$

③ Infix to prefix :
i/p: a+b*c
o/p: +a*bc

$E \to \{pf(+)\} E+T$ L ✓
$\quad | T$ S ✗
$T \to \{pf(*)\} T*F$
$\quad | F$
$F \to id \{pf(id)\}$

④ Binary to Decimal :
i/p : 111 101.111
o/p : 7 5.875

$S \to L.L \{S.DV = L_1.DV + L_2.DV \times 2^{-L_2.nb}\}$
$\quad | L \{S.DV = L.DV\}$
$L \to L\,B \{L.DV = 2 \times L.DV + B.DV,\ L.nb = L_1.nb + B.nb\}$
$\quad | B \{L.DV = B.DV,\ L.nb = B.nb\}$
$B \to 1 \{B.DV = 1,\ B.nb = 1\}$
$\quad | 0 \{B.DV = 0,\ B.nb = 1\}$

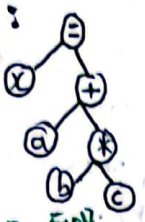
DV → Decimal value
nb → no. of bits
] Syn
⇒ S ✓ L ✓

Note:
• compressed form of parse tree = Syntax tree
• In Syntax tree, Leafs = operands
Internals = operators

CamScanner

⑤ Constr.ⁿ of Syntax tree for Arithmetic exp. :

i/p: $x = a+b*c$ o/p:



$S \to F = E$ { S.P = mknode(F.P, =, E.P); return(S.P) }

$E \to E+T$ { E.p = mknode(E₁.p, +, T.p) }

| T { E.p = T.p }

$T \to T*F$ { T.p = mknode(T₁.P, *, F.P) }

| F { T.p = F.p }

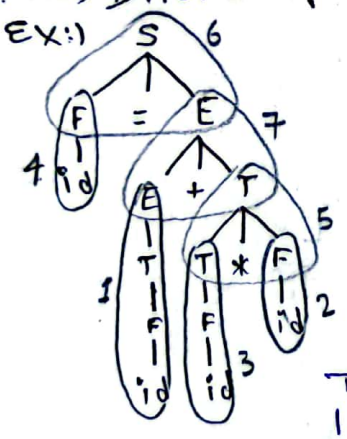
$F \to id$ { F.p = mknode(N, id, N) }

P = Pointer → Syn ⇒ L ✓ S ✓

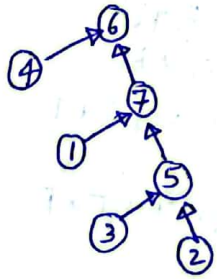
mknode = makenode

Note:) DAG & Topological sort :-

Ex:)



Dependency graph (DAG)



Topological sort order

1, 4, 2, 3, 5, 7, 6

4, 1, 3, 2, 5, 7, 6

2, 3, 5, 1, 7, 4, 6

Note:) L-attributed to S-attributed conversion :-

Ex:)

$E \to \{pf(+)\} E+T$	$E \to mE+T$
| T	| T
$T \to \{pf(*)\} T*E$	$m \to \epsilon \{pf(+)\}$
| F	$T \to m_1 T*E$
$F \to id \{pf(id)\}$	| F
L-att.	$m_1 \to \epsilon \{pf(*)\}$
	$F \to id$
	S-att.

⑥ Counting no. of reductions needed to execute arithmetic exp. :

i/p: $a+b*c$ o/p: 8

$E \to E_1+T$ { E.nr = E₁.nr + T.nr + 1; pf(E.nr) }

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

$T \to T_1*F$ { T.nr = T₁.nr + F.nr + 1 }

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

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

nr = no. of reductions = Syn. ⇒ S ✓ L ✓

⑦ Generation of intermediate code for Arithmetic expression :

i/p: $x = a+b*c$ o/p: $t_1 = b*c$, $t_2 = a+t_1$, $x = t_2$

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

$E \to E_1+T$ { E.val = NewTemp(); gen(E.val = E₁.val + T.val) }

| T { E.val = T.val }

$T \to T_1*F$ { T.val = NewTemp(); gen(T.val = T₁.val * F.val) }

| F { T.val = F.val }

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

gen() will generate o/p code (like $t_1 = b*c$)

NewTemp() " new temporary variables ($t_1, t_2, t_3 \ldots$)

⑧ Storing type information into Symbol table :

i/p: float x, y, z;

o/p:

S.no.	Variable Name (V.N)	Variable Type (V.T)
1	x	float
2	y	float
3	z	float

$D \to T\ V;$ { V.i = T.S }

$T \to$ int { T.S = int }

| float { T.S = float }

| char { T.S = char }

$V \to V_1, id$ { AddType(id, V.i); V₁.i = V.i }

| id { AddType(id, V.i) }

i ⇒ V का type

S ⇒ data type का नाम

i, S ⇒ inherited, Syn. ⇒ L ✓ S ✗

#) Runtime Environment :

*) • Fx.ⁿ called ⇒ Activation record pushed in Stack

• Fx.ⁿ Running ⇒ " " modifying

• Fx.ⁿ completed ⇒ " " popped.

*) Activation Record of f() contains :-

x, a, b, c ← 1. Local vars.

t₁, t₂, t₃ ← 2. Temp. "

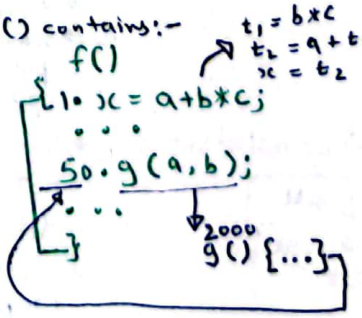
keyboard pressed 'A' ← 3. M/c & status

2000 ← 4. Ctrl link

5. Access link

50 ← 6. Return Address

a, b ← 7. Actual parameters



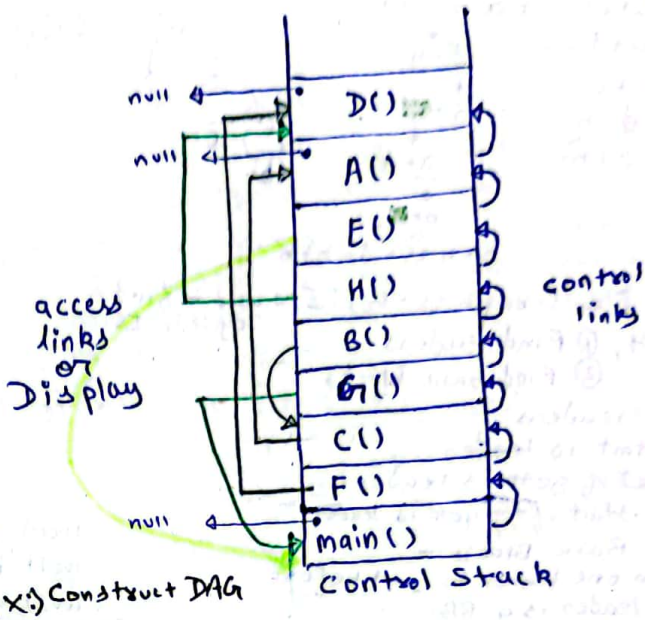
• Access link

Ex:)

```
main()
{ ...
  E()
  { ...
    A();
    ...
  }
  ...
  G()
  { ...
    B();
    ...
  }
  ...
  F();
}
```

```
A()
{ ...
  C()
  { ...
    B()
    { ...
      H();
      ...
    }
    G();
  }
  D();
}
```

```
D()
{ ...
  F()
  { ...
    C();
    ...
  }
  ...
  H()
  { ...
    E();
    ...
  }
  ...
}
```



*) Storage Allocation:

- Static, Stack, Heap
- Static :-
  - → Static variable allocate memory one time, i.e. at compile time.
  - → Static variable binding is done at compile time & doesn't change at runtime.
  - → It don't support recursion & dynamic data structures.
- Stack :-
  - → Fx.n called ⇒ push in stack
    Fx.n completed ⇒ pop in stack
  - → Supports recursion
  - → Don't support Dynamic DS.
  - → Always ~~any~~ local variable belongs to new activation record only (Drawback).
- Heap :-
  - → Memory allocation/deallocation any time based on user requirement.
  - → Supports both recursion & dynamic DS.

) Intermediate Code Generation:

*) Intermediate code is independent of m/c i.e. its portable.

*) Representations of IC :

1. Linear
   - → Postfix notation
   - → 3-Address code
     - Static Single Assignment (SSA)
     - non-" " " (~~SSA~~)
2. Tree
   - → Syntax tree
   - → DAG

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

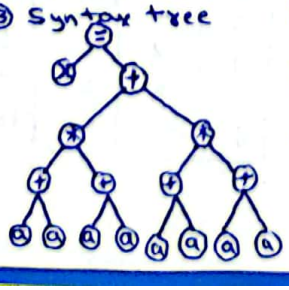
① 3-Address code

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

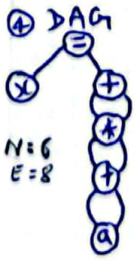
② Postfix

$x\,aa+aa+*\,aa+aa+*\,+=$

③ Syntax tree

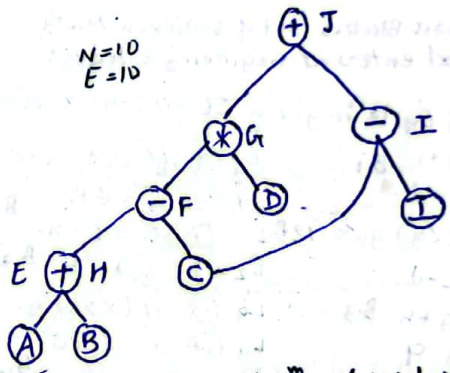


④ DAG



Ex:) Construct DAG

$E = A+B$
$F = E-C$
$G = F*D$
$H = A+B$
$I = I-C$
$J = I+G$



Ex:) Convert ~~SSA~~ to SSA. Min^m no. of total vars.?

$x = v-t$	$x_1 = v-t$
$y = x*v$	$y_1 = x_1 * v$
$x = y+w$	$x_2 = y_1 + w$
$y = t-z$	$y_2 = t-z$
$y = x*y$	$y_3 = x_2 * y_2$

new temp vars ⇒ $x_1, x_2, y_1, y_2, y_3 = 5$
old vars ⇒ $v, t, v, w, z = 5$] 10 total

Ex:) ~~SSA~~ → SSA

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

New ⇒ p_1, p_2, q_1, q_2 ⇒ 4
old ⇒ a, b, c, u, v ⇒ 5
9

Ex:) Consider Basic Block. Min^m nodes & Edges in DAG?

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

$t_4 = t_3 + t_2$
$t_3 = t_1 * t_2$, $t_2 = a*b$
$t_1 = a*a$, $t_2 = a*b$

Go Bottom to up & Substitute whenever necessary

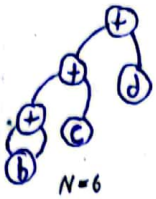
$(a*a)*(a*b) + (a*b)$

Ex:) Consider Basic Block. Min^m nodes & edges in DAG.

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

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

$b + b + c + d - \cancel{b} + \cancel{b}$



N=6
E=6

*) Control Flow Graph (CFG): It's used to find cycle in IC.

To draw it, ① Find leaders
② Find basic blocks

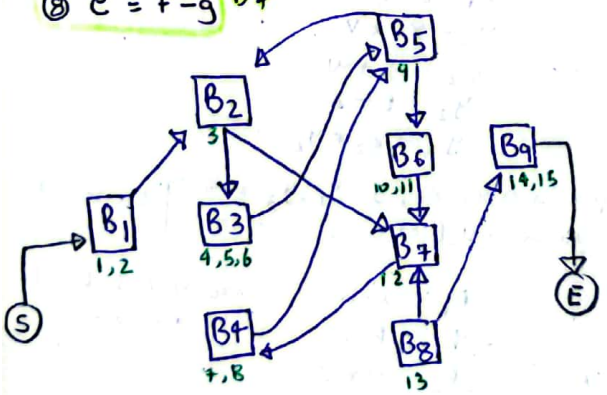
- To Find leaders:-
  → 1st stmt is leader
  → Target of goto is leader
  → Next stmt after goto is leader.
- To find Basic Block:-
  → From one leader to just before next leader is a BB.

Note:) Basic Block is set of 3-Address stmts where ctrl enters at beginning & leaves at end.

Ex:) Find Cycle in given IC (or) 3-Addr. code.

L1 ① $x = a+b$ B_1
② $y = c+d$
L4 ③ if $(x<y)$ goto 12 B_2
L6 ④ $a = c-d$
⑤ $e = g+h$ B_3
⑥ goto 9
L5 ⑦ $c = g+h$
⑧ $e = f-g$ B_4

L3 ⑨ if$(a<e)$ goto 3 B_5
L7 ⑩ $c = d+e$
⑪ $e = c-x$ B_6
L2 ⑫ goto 7 B_7
L8 ⑬ if$(x>y)$ goto 12 B_8
L9 ⑭ $a = c+d$
⑮ $e = g+h$ B_9



N=11
E=13

On doing DFS, Cycle found $B_1 B_2 B_3 B_5 B_2$

Ex:) Give 3-address code for:-

1) if((a<b) && (c>d)) {e=1;}
else { f=0;}

1000) if(a<b) goto 1002
1001) goto 1003
1002) if(c>d) goto 1005
1003) f=0
1004) goto 1006
1005) e=1
1006) — (Halt)

Back patching

2) for (k=25; k≤n; k++)
x = y+z*a+b;

1000) k = 25
1001) if(k≤n) goto 1003
1002) goto 1009
1003) $t_1 = z*a$
1004) $t_2 = y+t_1$
1005) $t_3 = t_2+b$
1006) $x = t_3$
1007) k = k+1
1008) goto 1001
1009) — (Halt)

3) k = 25;
switch(k)
{case 7: $x_1 = a_1+b_1*c_1$;
break;
case 27: $x_2 = a_2+b_2*c_2$;
break;
default: $x_3 = a_3+b_3*c_3$;
}

1000) k = 25
1001) if(k==7) goto 2000
1002) if(k==27) goto 3000
1003) $t_1 = b_3*c_3$
1004) $t_2 = a_3+t_1$
1005) $x_3 = t_2$
1006) — (H)

2000) $t_1 = b_1*c_1$
2001) $t_2 = a_1+t_1$
2002) $x_1 = t_2$
2003) goto 1006

3000) $t_1 = b_2*c_2$
3001) $t_2 = a_2+t_1$
3002) $x_2 = t_2$
3003) goto 1006

4) Given: int a[30][40]; Row major
w = 2B
Give 3-Addr. code for:-
x = a[i][j]

$t_1 = i*40$
$t_2 = t_1+j$
$t_3 = t_2*2$
$t_4 = a+t_3$
$x = *t_4$

Loc(A[i][j])
$= B + w[(i-0)(39-0+1) + (j-0)]$
$= B + 2[i*40 + j]$
a[i][j] = *( )

*) Types of 3-Address codes allowed:-

① $x = y+z$
② $x = y$
③ $x = *y$
④ $x = \&y$
⑤ $x = f(y)$;
⑥ $x = a[i]$
⑦ $a[i] = x$
⑧ if$(x<y)$ goto z
⑨ goto z

*) Representations in memory of 3-Address codes:-
- Quadruple
- Triple
- Indirect triple

Ex:) $a = (b*-c) + (b*-c)$

Quadruple

S.no.	operator	opnd1	opnd2	Res
1	-	c		t_1
2	*	b	t_1	t_2
3	-	c		t_3
4	*	b	t_3	t_4
5	+	t_2	t_4	t_5
6	=	t_5		a

Triple: S.no. acts as Res

S.no.	operator	opnd1	opnd2
1	-	c	
2	*	b	(1)
3	-	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	a	(5)

Indirect triple: Relocate
Ability to

S.no.	memory
1	→ 5000
2	
3	
4	→ 6000
5	
6	

Note:) Control flow graph is also an intermediate representation of source program.

#> Code Optimization:

can be done at High level, intermediate level, low level.

*) Techniques:-

1) Loop optimization
   i) Loop invarient: Statement which is not varying in loop, comes outside of loop
   ii) Loop unrolling: Decreasing comparisons (or) Test cases
   iii) Loop jamming & combining loops
   iv) Induction variable elimin^n: Eliminating variable which is always inc./dec. by constant factor.

2) Strength red.^{n}: $x^2 \rightarrow x*x$, $2*x \rightarrow x<<1$, $x/4 \rightarrow x>>2$ | $*, /$ ↑ cost, $<<, +, -, >>$

3) Algebraic simplif.^{n}: $1*x \rightarrow x$, b&&True $\rightarrow$ b

4) Constant folding: $10+2*3 \rightarrow 16$
   $(10>20)$? goto L_1 : goto L_2; $\rightarrow$ goto L_2

5) copy propagation: Do not copy & propagate
   $t_2 = t_1$, $t_3 = t_2 * t_1$ $\rightarrow$ $t_3 = t_1 * t_1$

6) Const. propagation: Do not copy constant & propagate.
   $a = 10$, $c = b+d+a$ $\rightarrow$ $c = b+d+10$

7) Dead code elimination: Eliminate code not participating in program.

8) Common subexpression elimination: Draw DAG

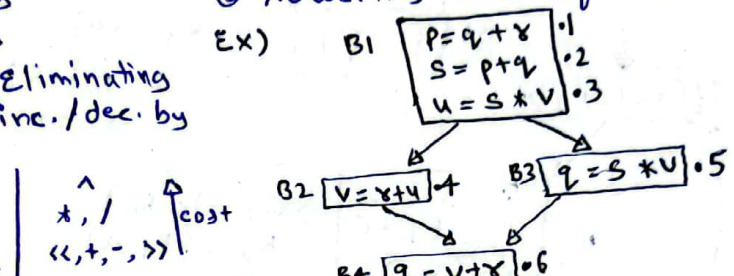
*) Code optimization on m/c dependent code:

1) Peephole optimization:
   i) Redundant Load/Store instr.^{n}s
      Load R_0, a
      ~~Store a, R_0~~
   ii) Use of m/c instructions
      mov R_0, 1
      mov R_1, i $\rightarrow$ inc i
      add R_1, R_0
      mov i, R_1
   iii) Strength reduction
   iv) Removal of unreachable code
      if(0) pf(hi) $\rightarrow$ pf(bye)
      else pf(bye)
   v) Flow control optimization
      L_1: goto L_2
      L_2: goto L_3 $\rightarrow$ L_1: goto L_3
      L_3: ...
   vi) Algebraic simplification
   vii) constant folding
2) Register allocation optimization: use less registers without changing meaning.

*) Live Variable Analysis: Used for Register allocation optimization & dead code elimination.

•) At statement-i, variable a is live if:-
   ① Some statement at j should read 'a'
   ② There should be path from i to j
   ③ No writing to 'a' before j

Ex)



at statement 4th, live = r, u
— 5 — = s, u, v, r
— 2 — = p, q, v, r

•) Local Data (within BB) properties for LVA:-
   i) $Gen_R = \{v \mid$ variable v is read in BB R & v is not preceded by write in BB R$\}$
   ii) $Kill_R = \{v \mid$ variable v contain write in BB R$\}$

Ex:)

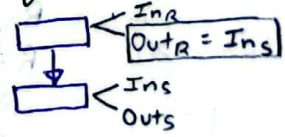
B_R
x = a+b
y = x+z
c = a+d
if (x<e)

$Gen_R = \{a, b, z, d, e\}$
$Kill_R = \{x, y, c\}$

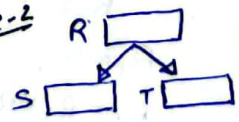
   iii) In_R = Variables live at entrance of BB R
   iv) Out_R = " " " exit " " "

Here, To find Out_R :-

case-1 (approaching bottom-up)

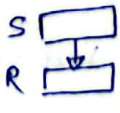


case-2



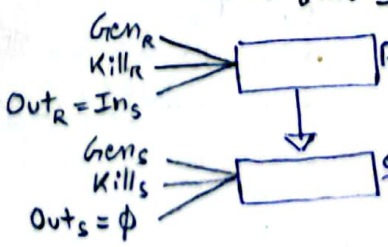
$Out_R = In_S \cup In_T$

case-3



$Out_S = In_R$
$Out_R = \phi$

Here, To find In_R :-

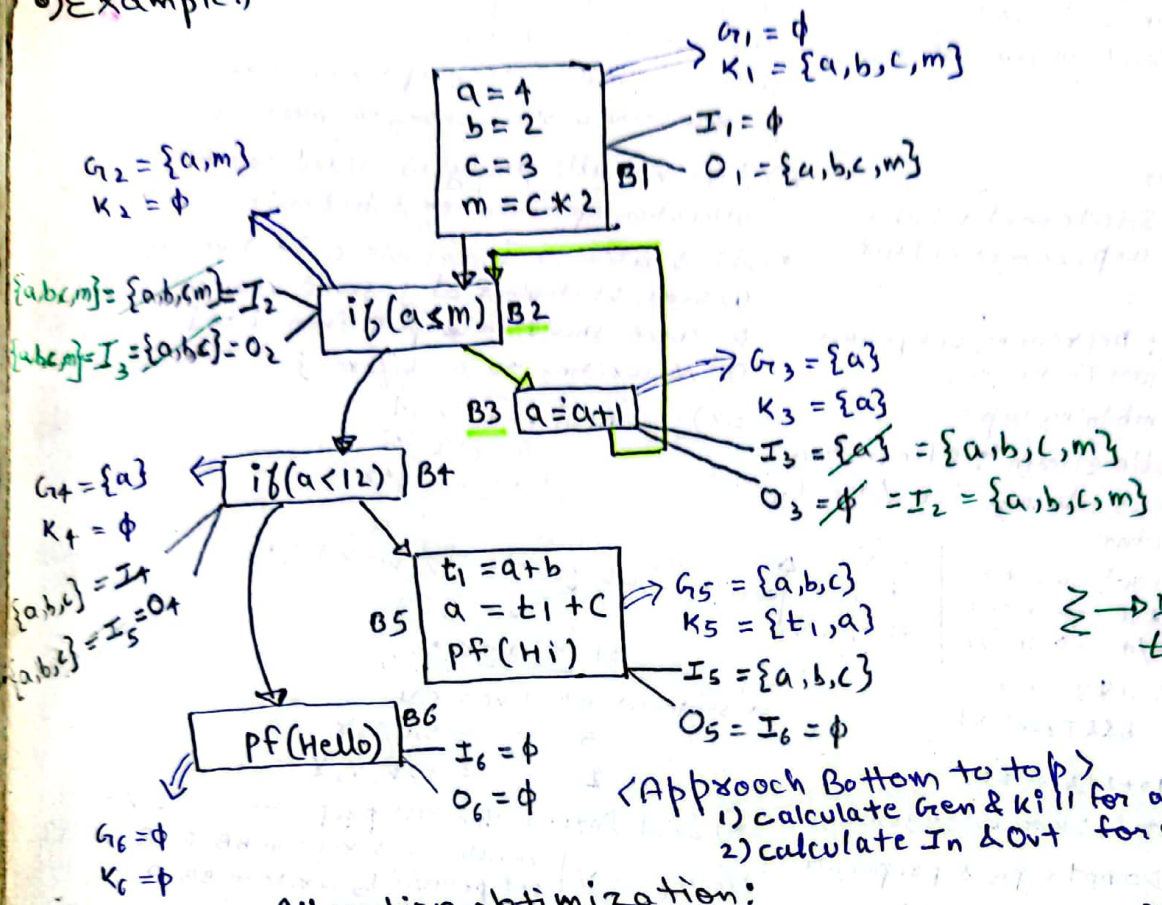


$In_R = (Out_R - Kill_R) \cup Gen_R$

$In_S = (Out_S - Kills) \cup Gens$

CamScanner

•) Example:)



Σ → Doing one more time bec. of Loop

<Approach Bottom to top>
1) calculate Gen & Kill for all blocks
2) calculate In & Out for all blocks

•) Register Allocation optimization:

Ex:) The program below uses 6 temporary variables (a,b,c,d,e,f). Assuming all operations ~~operands~~ take their operands from registers. Find min.m registers needed (Spilling not allowed).

a = 1 ✓
b = 10 ✓
c = 20 ✓
d = a+b ✓
e = c+d ✓
f = c+e ✓
b = c+e ✓
e = b+f ✓
d = 5+e ✓
return d+f ✓

$R_1 = \cancel{a}\ \cancel{b}\ \cancel{e}\ \cancel{b}\ \cancel{f}\ d$
$R_2 = \cancel{b}\ f$
$R_3 = \cancel{c}\ e$

$R_1 = R_1 + R_2$
$R_1 = R_3 + R_1$
$R_2 = R_3 + R_1$
$R_1 = R_3 + R_1$
$R_3 = R_1 + R_2$
$R_1 = 5$, $R_1 = R_1 + R3$
$R_1 = R_1 + R_2$, return R_1

<Approach: Variable not live afterwards, then replace>

Note:) Spilling: Storing data from register to temporary variable i.e. $t_1 = R_1$

Note:) Single pass compiler is faster & smaller than multi pass compiler

Note: • In int a = 0x9f(i); // Lexical error cuz 'i' not allowed in Hex.
• In isDigit(x); // 5 tokens are there
• In x == '\t'; // 4 tokens are there

Note: consider basic block

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

$e = d + c$ → // d को expand किया b/c उपर वाले level में नहीं है

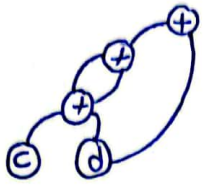
$c+d$ → $b+d$ → $a-b$ → $b+c$

$b+d$ → $a-b$ → $b+c$

$a-b$ → $b+c$

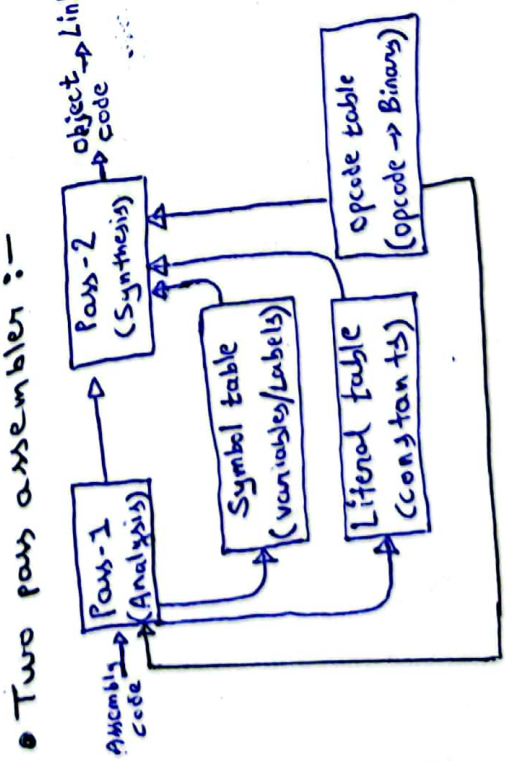
$b+c$ → // d को expand नहीं किया b/c उपर वाले level में है / (c+d)

$= \cancel{b}+c-\cancel{b}+d+d+\cancel{b}+c-\cancel{b}+d$
$= c+d+c+d+d$



Note: Among all LR parsers, only the no. of reduce entries may differ, # of shift entries are same.

Note:
- Symbol table implement.^{n} s: Array, LL, Tree, Hash table.
- Lex A → finite automata
  Syn A → PDA / Parser / Production tree
  Sem. A → LBA / Type checking
  ICG → Syntax trees
  C opt. → DAG / Data flow analysis
  TCG → Register allocation / Graph coloring
- If ∈ is in the language, then you can't remove ∈ production from start symbol (like $L=a^*$)
- Inherently ambiguous:–
  → L is non-Deterministic AND L is ambiguous ⇒ L is inherently ambiguous
  → L is Deterministic ⇒ L is not-inherently ambiguous
  → L is non-deterministic ⇒ L may/may not be inh. amb.
  → L is ambiguous AND no unambiguous grammar possible for L ⇒ L is inh. amb.
- $I_0, I_1, I_2 \ldots$ } LR(0) items
- LR → TC ⇒ O(n) [EC]
  LL(1) → TC ⇒ O(n^3) [BC] (Right Recursive grammar)
  O(n^4) [WC] (Left recursive ")
- In many OS, Loader is permanently resident in MM.
- For any CFG, there is a parser called CYK algorithm which takes O(n^3) time to parse a string of length 'n'.
- why we do code opt.^{n} on IC?
  To enhance portability of compiler

- In BUP of SDT, inherited attributes can be evaluated only if definition is L-attributed.
- SDT for conditional & iterative stmt:
  S → if E then S_1 { E.true = new label
  E.false = S.next
  S_1.next = S.next
  S.code = append(E.code,
- If you look at SLR(1) table & LALR(1) table
  Goto part, Shift entries ⇒ Same in both
  Reduce entries, Error entries ⇒ may differ (Due to conflicts)
- Definition of Handle (as per LR parsing)
  The production P that will be used for reduction in the next step along with a position in the sentential form where RHS of the production may be found.

- Two pass assembler :-