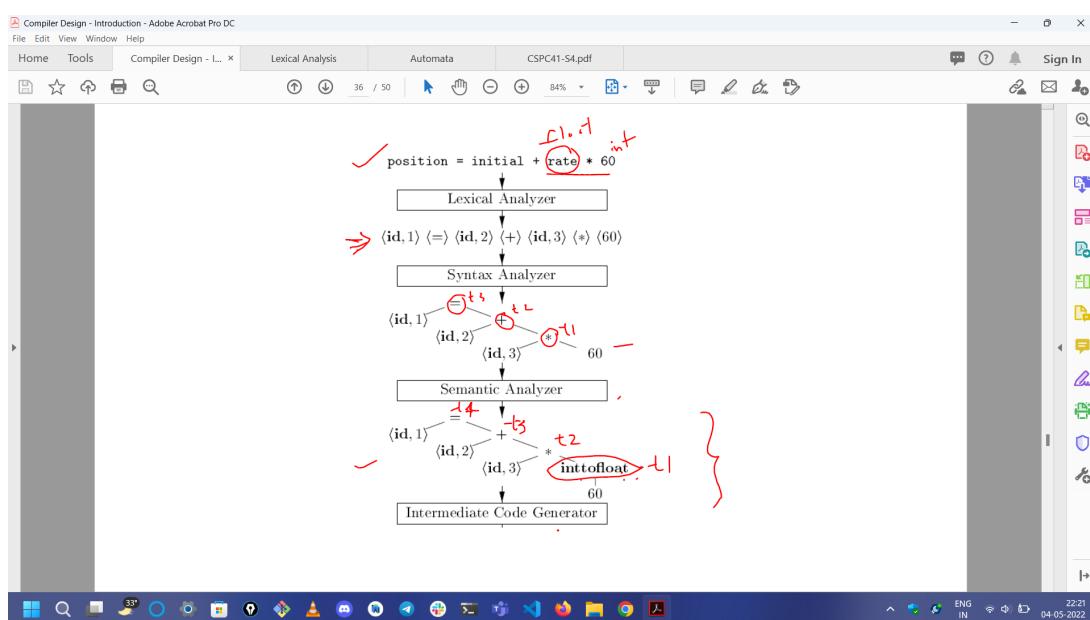
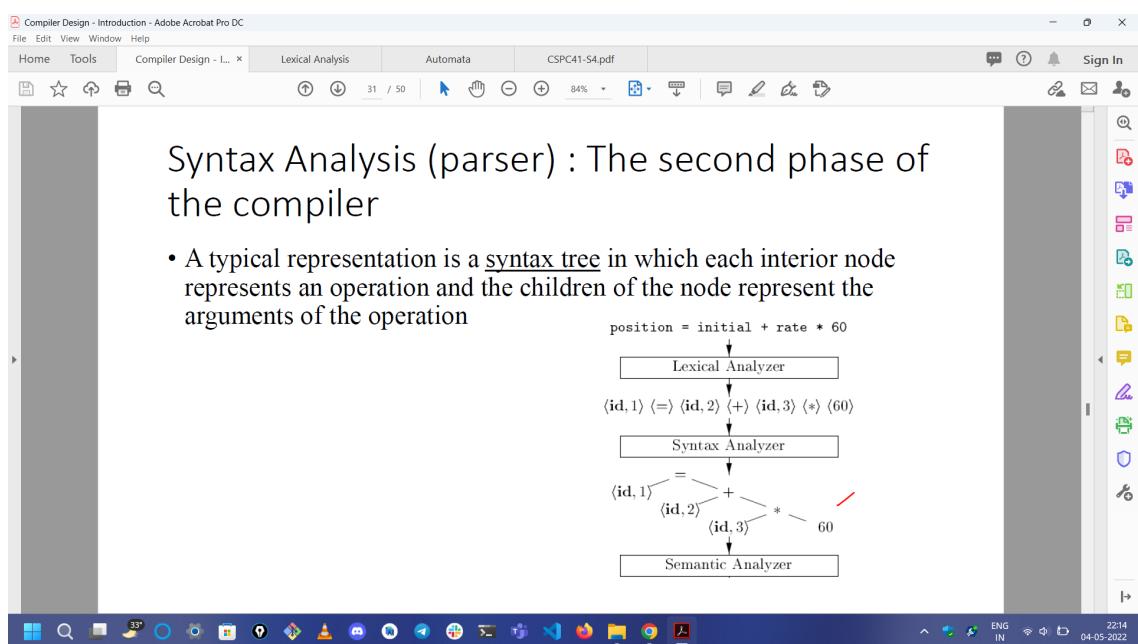
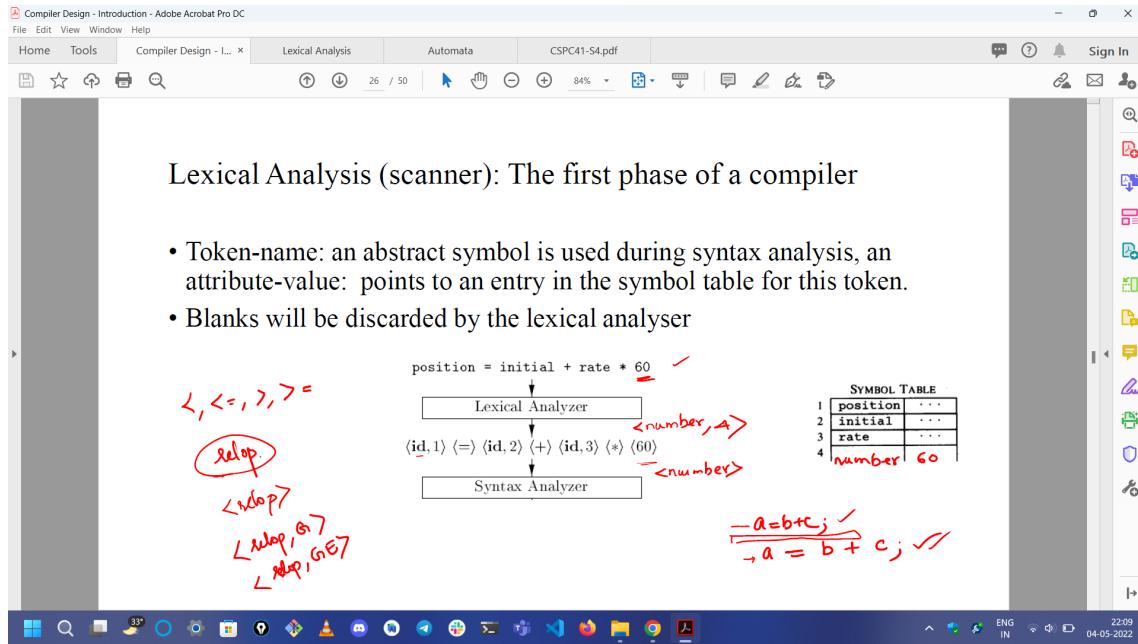
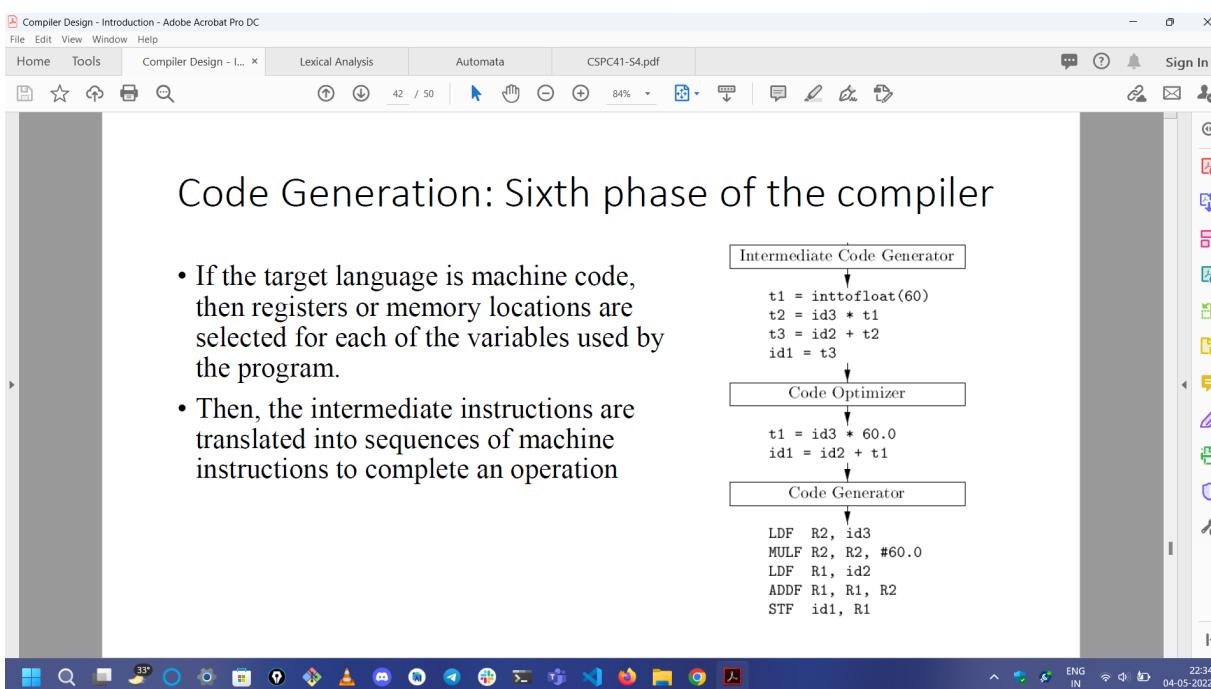
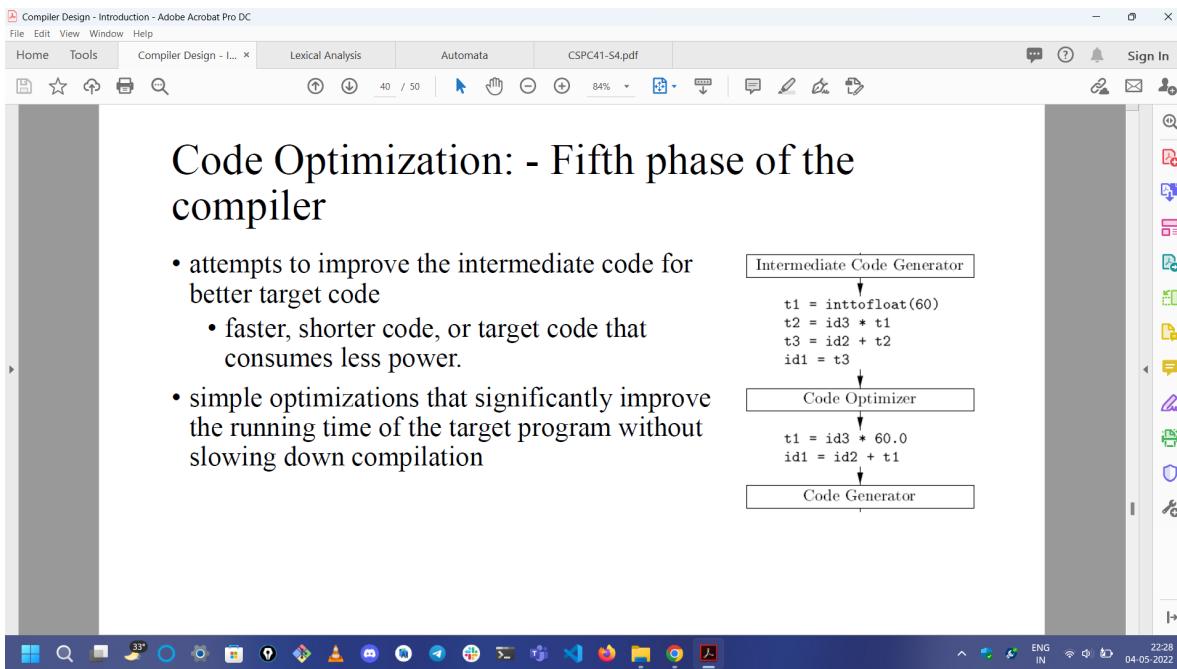
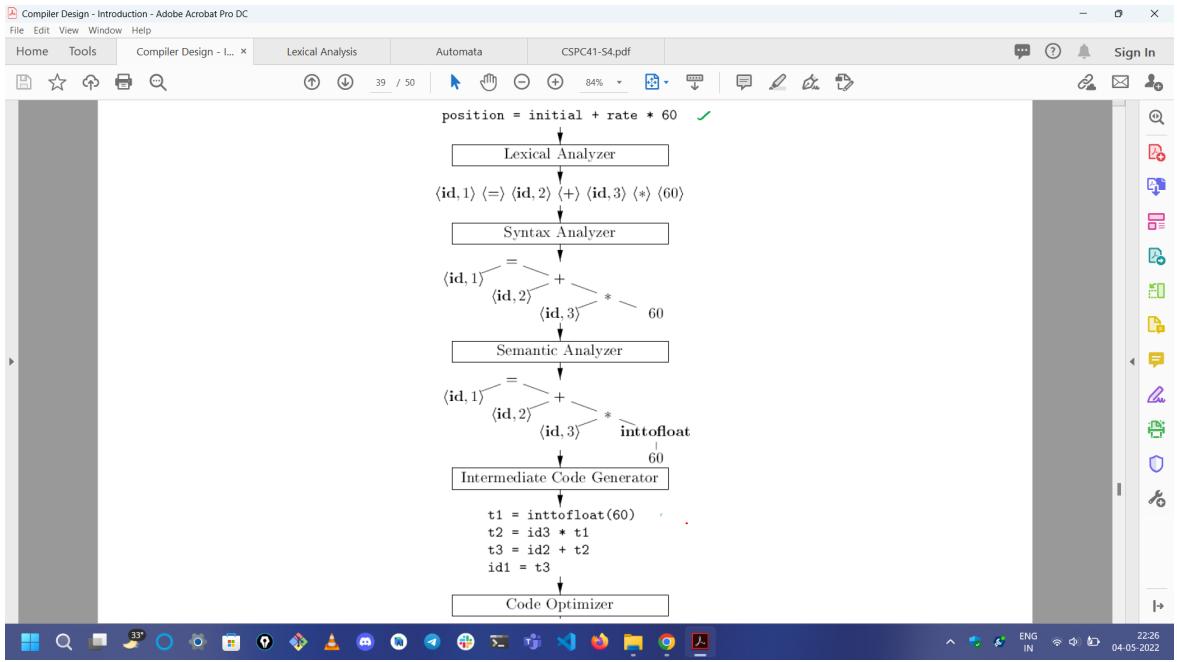


Compiler

- Phases of the compiler.
- Lexical Analysis
- Constructing DFA from RE
- Parsers - LL, SLR, CALR, LALR
- Unit 3 - all topics (except PPT on Semantic Phase of the Compiler).
- Code Generator Introduction
- Basic Blocks, Flow Graphs, Transformations on Basic blocks
- Code Generation Algorithm with examples
- DAG
- Code generation from DAG
- Loops in Flow graphs
- Peephole Optimizations.
- Global Data Flow Analysis - Data flow equations, Reaching Definitions.

Phases of the compiler.





Lexical Analysis

The screenshot shows a Microsoft Windows desktop with an Adobe Acrobat Pro DC window open. The window title is 'Lexical Analysis - Adobe Acrobat Pro DC'. The menu bar includes File, Edit, View, Window, Help, Home, Tools, Lexical Analysis, Automata, and a file named 'CSPC41-S4.pdf'. The toolbar contains various icons for file operations like Open, Save, Print, and Search. A search bar labeled 'Search tools' is at the top right. On the right side of the window is a sidebar with a list of tools: Create PDF, Combine Files, Edit PDF, Export PDF, Organize Pages, Send for Comments (NEW), Comment, Fill & Sign, Scan & OCR, Protect, and More Tools. Below the sidebar is a note: 'Store and share files in the Document Cloud' with a 'Learn More' link. The status bar at the bottom shows 'ENG IN' and the date '04-05-2022'.

Definitions

- Lexeme is a particular instant of a token.
- Token: a group of characters having a collective meaning.
 - token: identifier, lexeme: area, rate etc.
- Pattern: the rule describing how a token can be formed.
 - identifier: $([a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9])^*$

Handwritten notes:

$a^n b^n \{ \} _ \}$

$\underbrace{a \sim}_{aB} \underbrace{B \sim}_{B2}$

$([wyz] | [a-z] | [A-Z])^*$

Count vowels and const

```
%{
int vow_count=0;
int const_count =0;
%}

%%

[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
%%

int yywrap(){}
int main()
{
    printf("Enter the string of vowels and consonents:");
    yylex();
    printf("Number of vowels are: %d\n", vow_count);
    printf("Number of consonants are: %d\n", const_count);
    return 0;
}
```

```
%{
#include<stdio.h>
#include<string.h>
int i;
%}

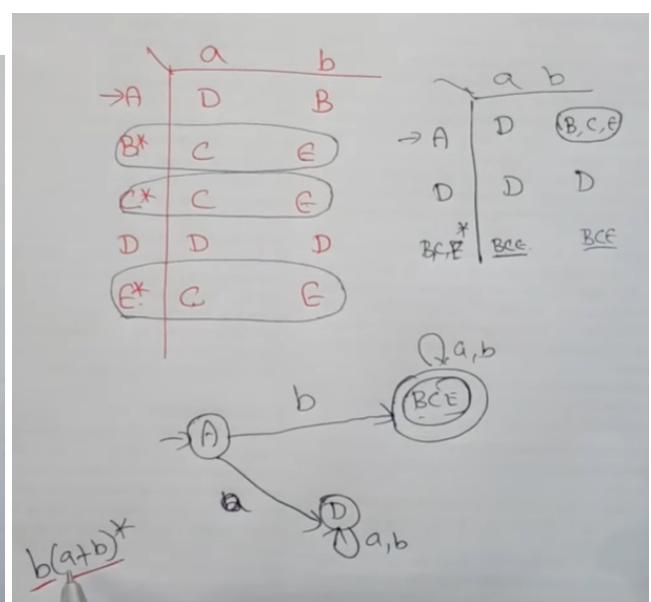
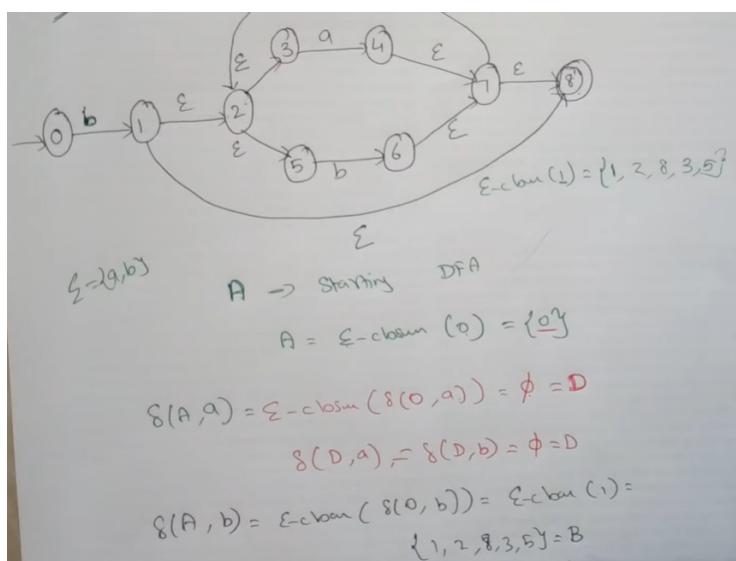
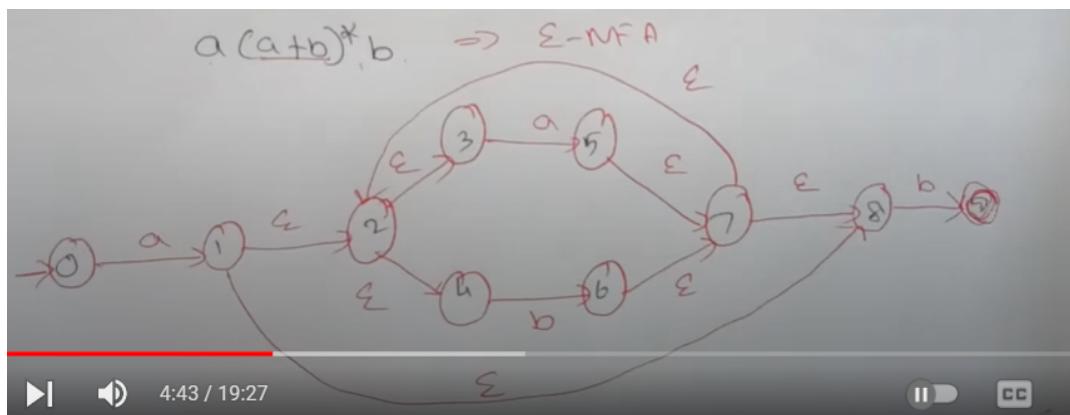
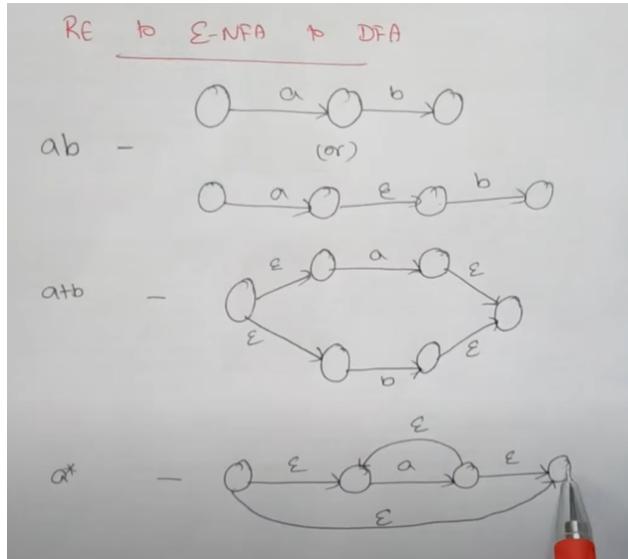
%%

[a-zA-Z]* {
    for(i=0;i<yylen;i++){
        if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c')){
            yytext[i]='A';
            yytext[i+1]='B';
            yytext[i+2]='C';
        }
    }
    printf("%s",yytext);
}

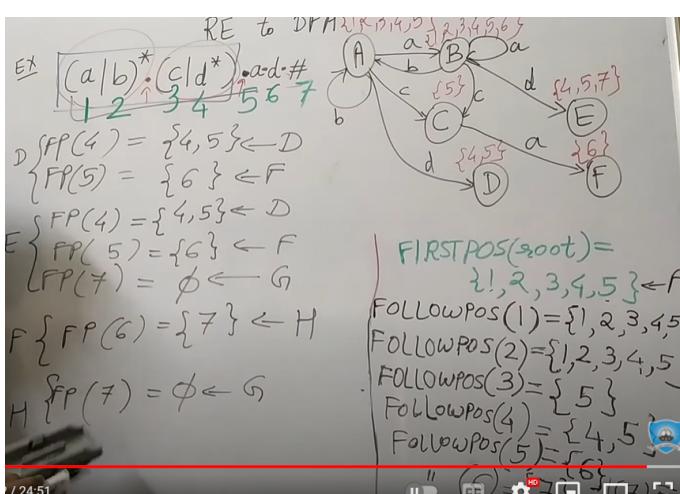
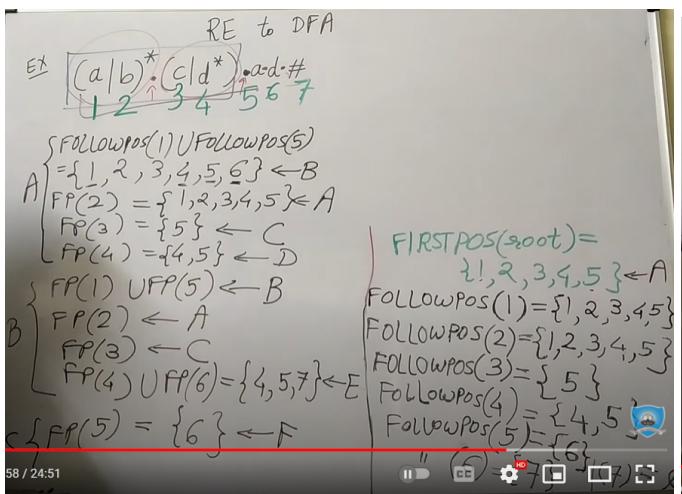
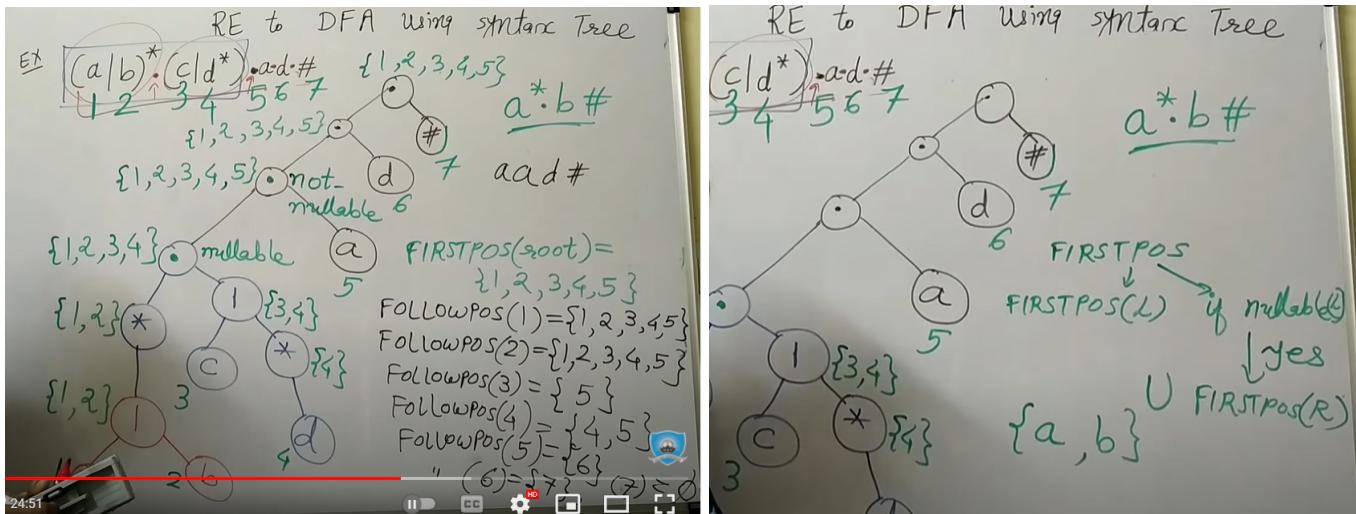
main(){
    yylex();
}

int yywrap(){
return 1;
}
```

Constructing DFA from RE



Using syntax tree



Comments RE

```
"//".*\n"/*[^*/]*\"
```

Parsers - LL, SLR, CALR, LALR

First() and Follow() help in making the parsing table for the top-down or bottom-up parsers.
So here we go:

First() of a Grammar

Always start finding Firsts from the terminals and then go up above for variables dependent on them.

"Finding First() & Follow()"

First(A) Contains all terminals present in first place of every string derived by A.

(1) $S \rightarrow abc \mid def \mid ghi$

(2) First (terminal) = terminal

(3) First (ϵ) = ϵ

$S \rightarrow ABC \mid ghi \mid jkl$

$A \rightarrow a \mid b \mid c$

$B \rightarrow b$

$D \rightarrow d$

$S \rightarrow ABC$

$A \rightarrow a \mid b \mid \epsilon$

$B \rightarrow c \mid d \mid \epsilon$

$C \rightarrow e \mid f \mid \epsilon$

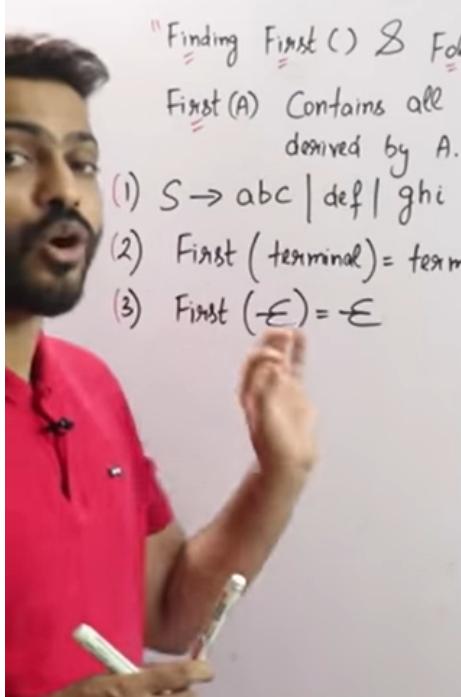
$E \rightarrow TE'$

$E' \rightarrow *TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow \epsilon \mid +FT'$

$F \rightarrow id \mid (E)$



"Finding First() & Follow()"

2.00 Contains all terminals present in first place of every string derived by A.

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

terminal) = terminal

$(\epsilon) = \epsilon$

$S \downarrow$

$F(ABC) \downarrow$

$A \downarrow$

$S \rightarrow ABC$

$A \rightarrow a \mid b \mid \epsilon$

$B \rightarrow b$

$D \rightarrow d$

$S \rightarrow ABC$

$A \rightarrow a \mid b \mid \epsilon$

$B \rightarrow c \mid d \mid \epsilon$

$C \rightarrow e \mid f \mid \epsilon$

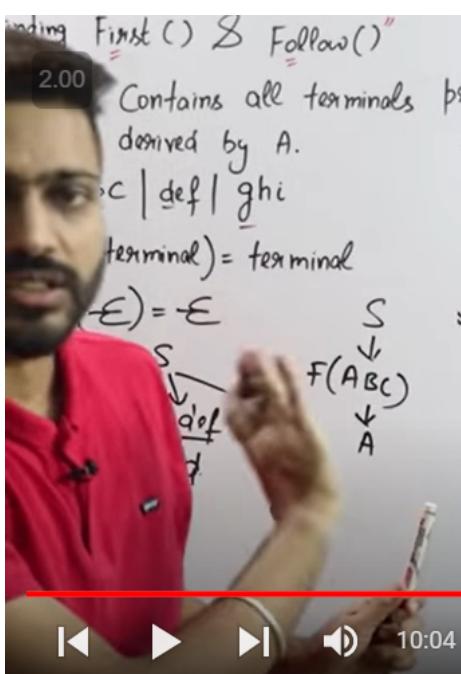
$E \rightarrow TE'$

$E' \rightarrow *TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow \epsilon \mid +FT'$

$F \rightarrow id \mid (E)$



◀ ▶ ▶| ⏴ 10:04 / 10:36⠇ ⚙ ⌂ ⌃ ⌄

NOTE : Main thing not to forget is: Whenever something first includes an epsilon, then you can add to it by the next term also. See the middle-down example.

Follow() of a Grammar

3 Fundamental Basic Points:

- 1) Follow is never epsilon.
- 2) Follow of start symbol (topmost left wala variable) is \$ (dollar symbol). There can be more as well but \$ to hota hi hai.
- 3) Follow hamesha right side main dekhke nikalte hai.

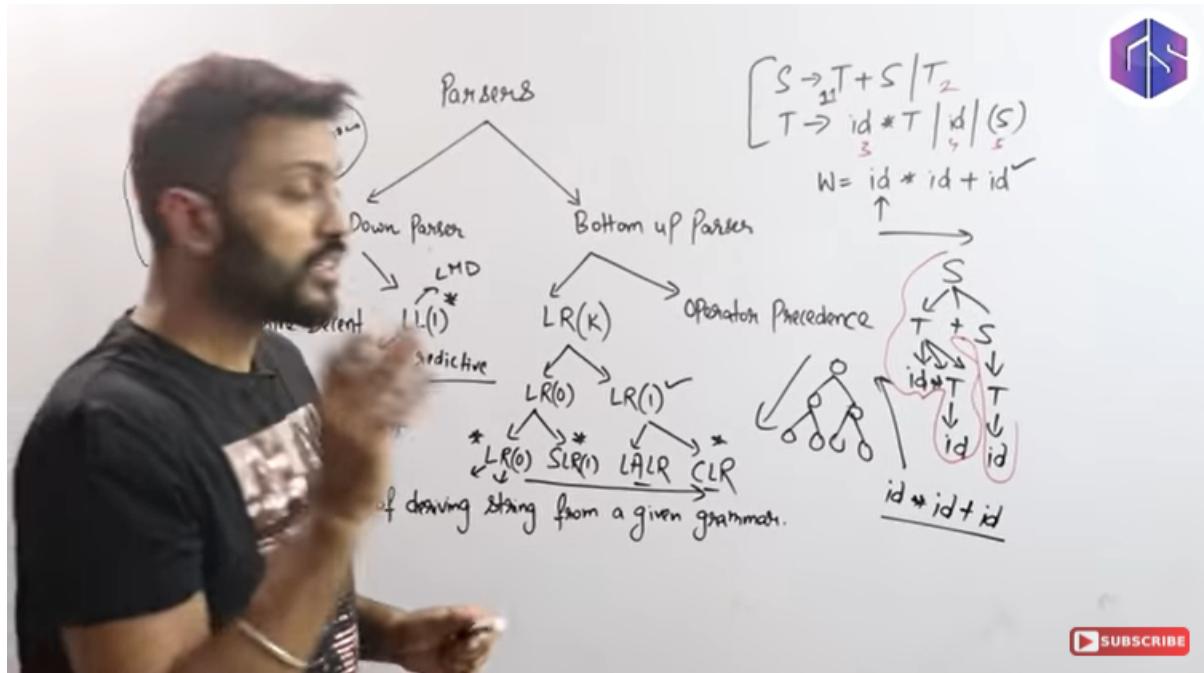
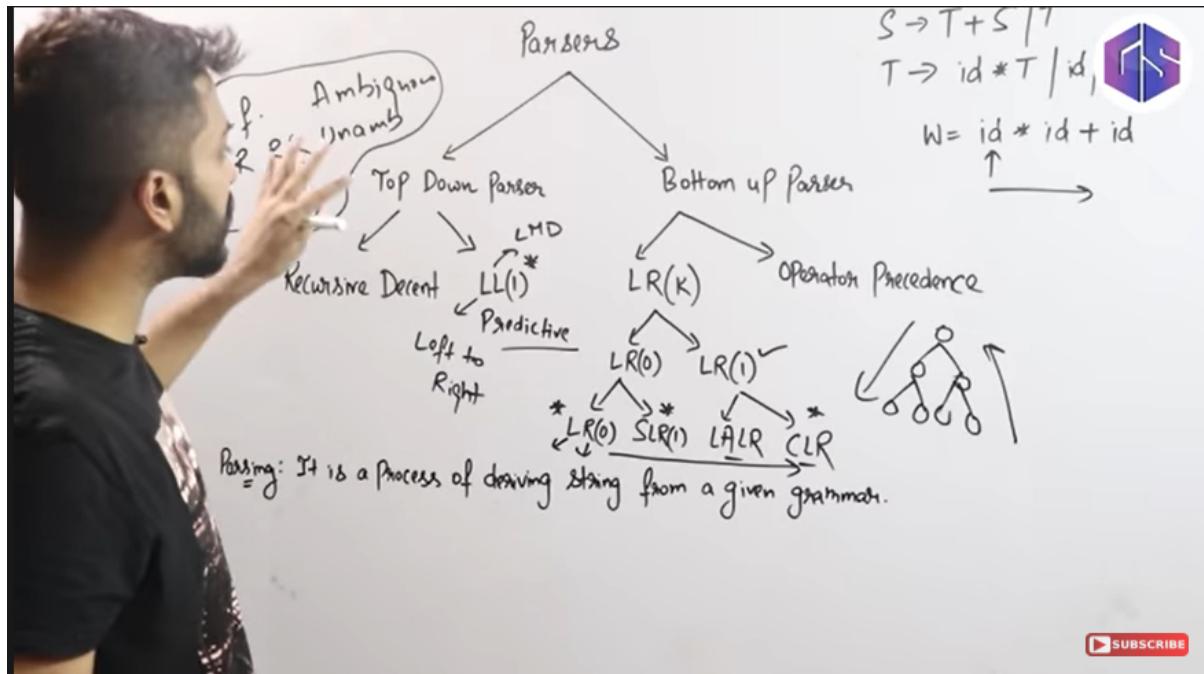
General Rules for finding Follow :

Follow(S) nikalna hai to arrow (kisi bhi) ke right side main S ki occurrences ke just **right** main dekho:

- i) agar terminal hai then that's the follow.
- ii) agar variable hai (let's say A) then First(A) is the follow iff this First doesn't come equal to epsilon.
- iii) If the First comes as epsilon, then find the first of right of this variable.
- iv) If Everything till the rightmost end comes out to be epsilon, then the Follow is the follow of the left side.

Parsing

Syntax Analysis ke liye ye phase hai. Parsing Tree se ek-ek karke check karte hai if string belongs to the grammar or not. Grammar used here is CFG(Context Free Grammar).



LL(1) Parser

¹ LL(1) Grammar Parsing Table "Follow First"

$$S \rightarrow^1 (L) \mid^2 a \{ (, a \} \{ \$, ? \} \}$$

$$L \rightarrow SL' \mid^3 \{ (, a \} \{) \} \}$$

$$L' \rightarrow \epsilon \mid^4 , S L' \{ \epsilon, , \} \{) \} \}$$

	()	a	,	\$
S	1		2		
L		3	3		
L'	L'	4		5	

first \$

Follow \$

^{2.00} LL(1) Grammar Parsing Table "Follow First"

$$S \rightarrow^1 (L) \mid^2 a \{ (, a \} \{ \$, ? \} \}$$

$$L \rightarrow SL' \mid^3 \{ (, a \} \{) \} \}$$

$$L' \rightarrow \epsilon \mid^4 , S L' \{ \epsilon, , \} \{) \} \}$$

	()	a	,	\$
S	1		2		
L		3	3		
L'	L'	4		5	

first \$

Follow \$

Not LL(1)

Compiler Design (Complete Playlist)
Syllabus Discussion for Competitive & College/University
COMPILER DESIGN
36 videos Exams

◀ ▶ 🔍 ⏴ ⏵ ⏷ ⏸ ⏹

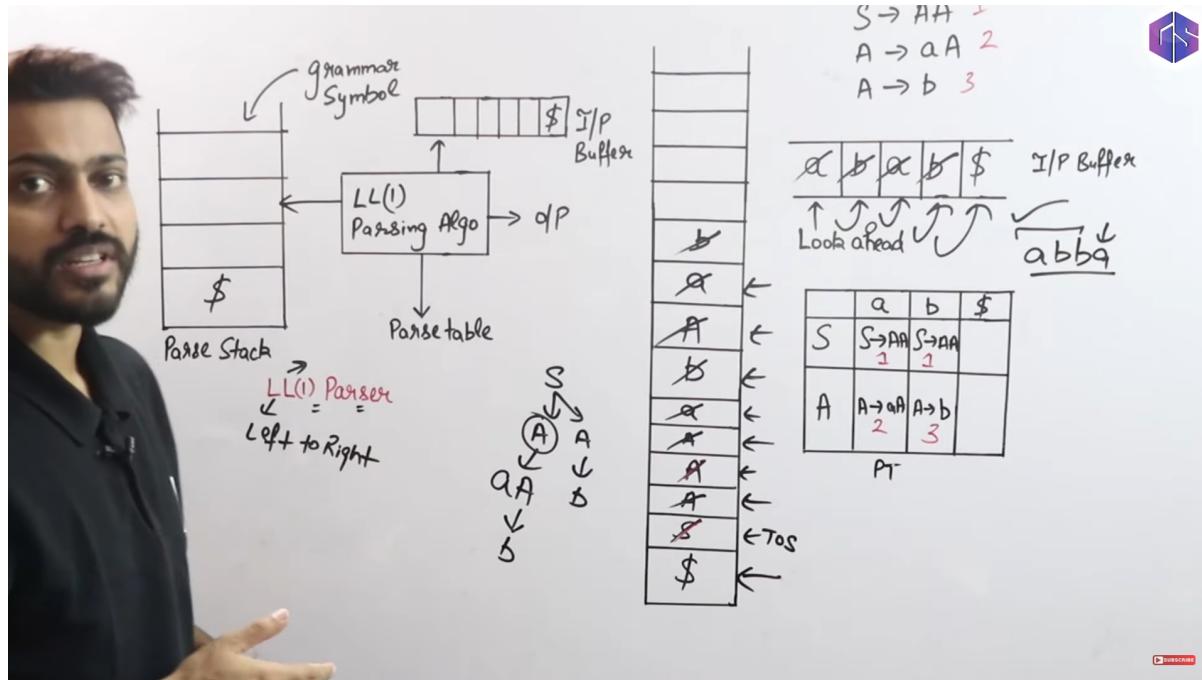
In above screenshots, Grammar on the left is accepted by LL(1) Parser and Grammar on the right is rejected by LL(1) Parser:

Steps to check if a grammar is LL(1) or not:

- 1) Har variable ka First and Follow nikal lo
- 2) Ab ye Table fill karni hai variables vs. terminals ki(\$) hamesha include karna hai as a column).
- 3) Har Production rule (For eg: X->Y) ke corresponding, X ki row main First(Y) column pe production rule number likho **if and only if First(Y) isn't epsilon.**
- 4) Agar First(Y) epsilon aa gaya, to X ki row main Follow(X) column pe likho.

FINAL DECISION: Final Table main kisi bhi cell main multiple entries nahi hai then Grammar is accepted.

LL(1) -> check whether a string is accepted by a grammar.



LR(0):

Columns has 2 sections:

Terminals - Action Region

Variables - Goto Region

CI -> canonical items

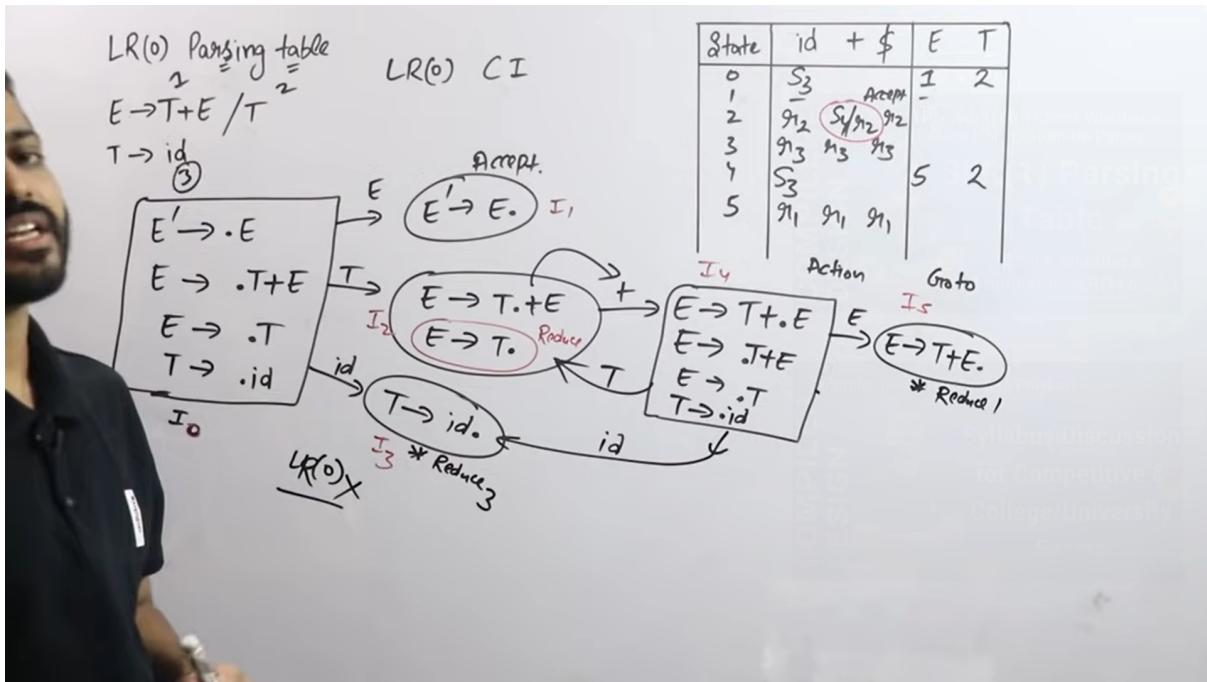
I0, i1 are canonical items

jismain bhi production rule(given grammar) complete ho jaye, usmain I me reduce likhna hota hai pure terminal wale(Action) row portion mai

Terminals(Action Region) ke corresponding S1, S2 type se likhna hota hai varna (Goto) keval 1,2 Likho

** Agar khud ki state reduce h to sare row me r likho

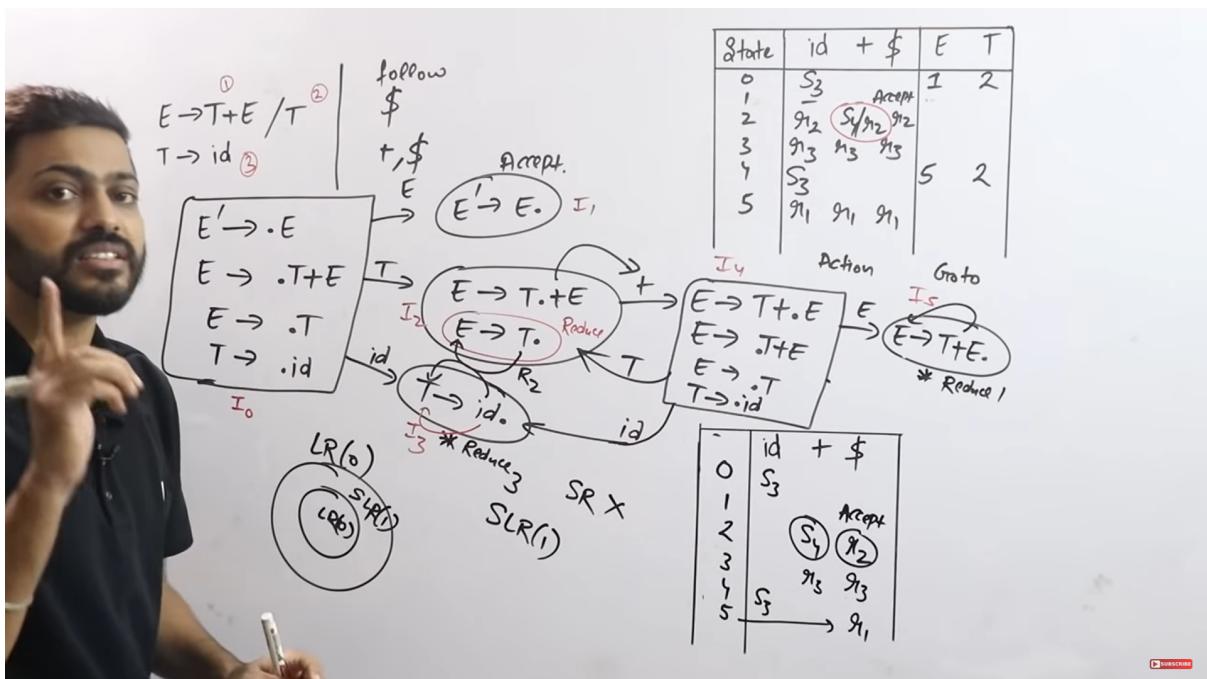
Grammar is LR(0) accepted - 1 cell shouldn't contain shift-reduce or reduce-reduce conflict.



SLR(1) Parser

Same as LR(0) but reduce is written only in Follow of left side of corresponding production rule (reduced CI).

Grammar is SLR(1) accepted - Checking rule is same as LR(0) - 1 cell shouldn't contain shift-reduce or reduce-reduce conflict.



CLR :

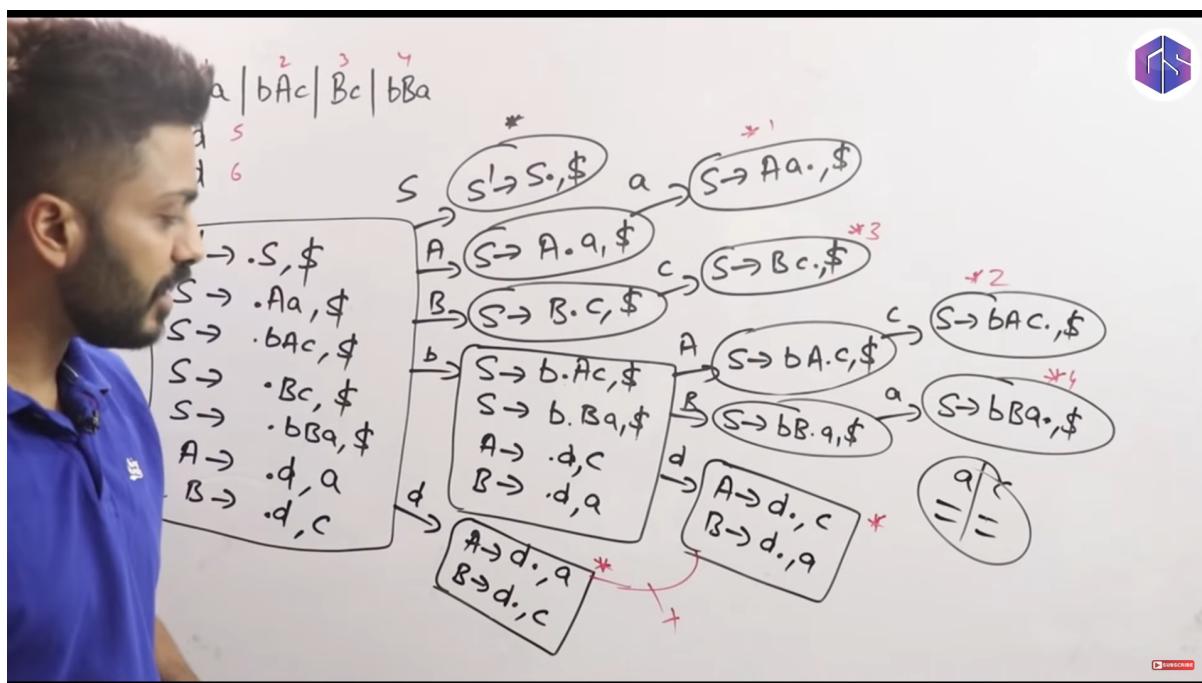
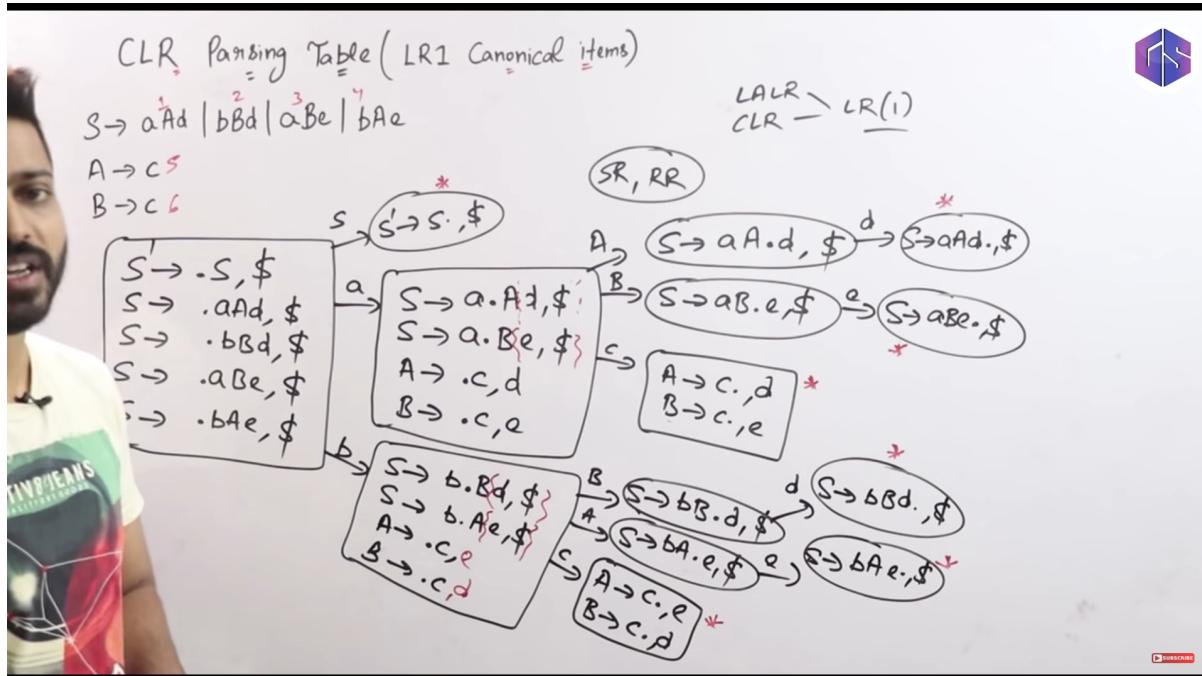
LR1 canonical item (look ahead)

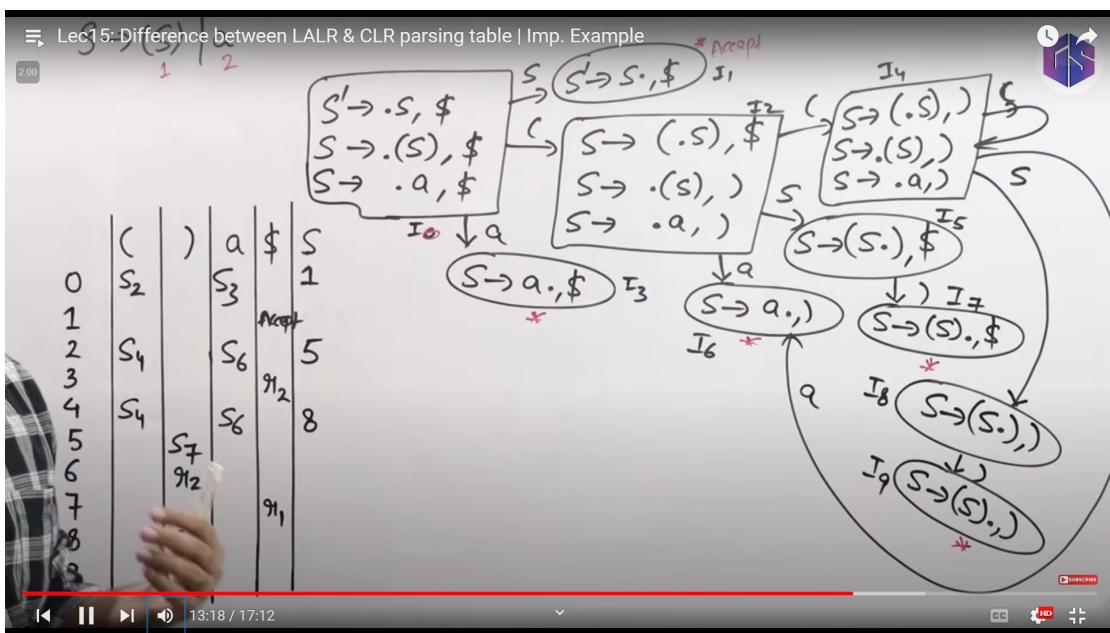
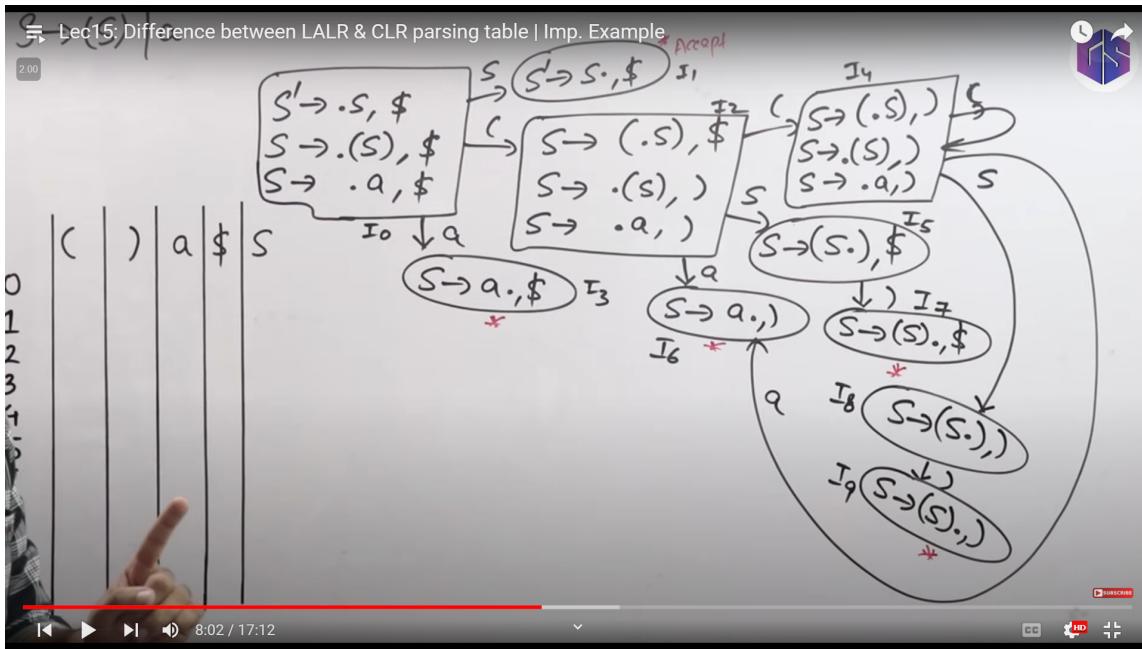
Write reduced state in the look-ahead column

After ' , ' add look ahead

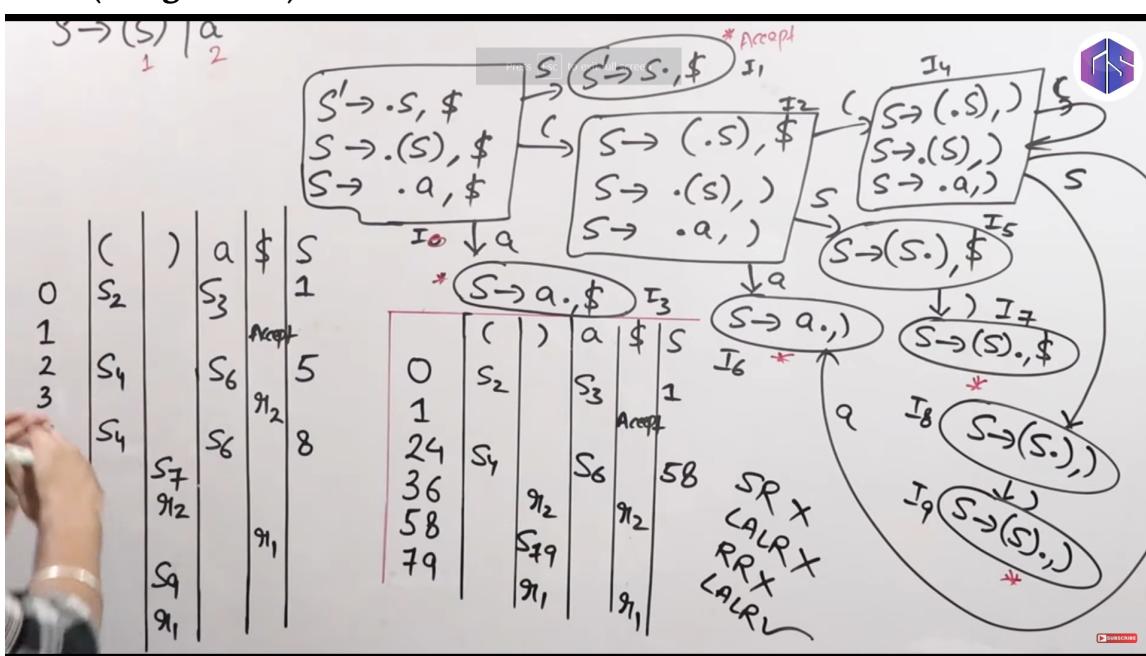
Start with S' but write \$ as look ahead,

** now for the same state write first(remaining) as look ahead.





LALR:(merged CLR)

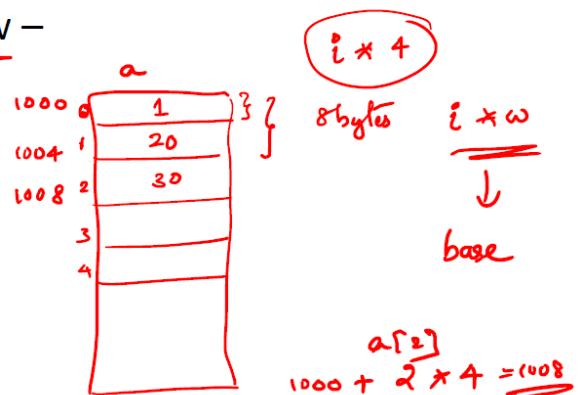


Unit 3 - all topics (except PPT on Semantic Phase of the Compiler).

-> SDT of Array

Addressing array elements

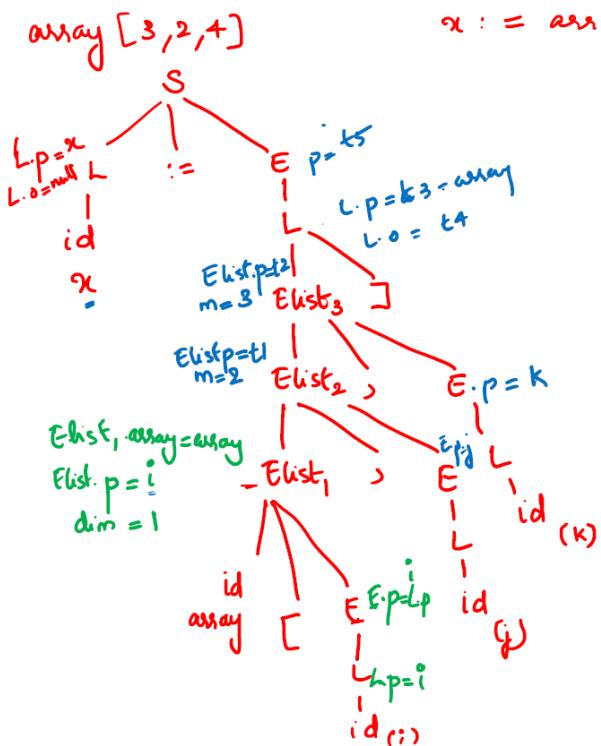
- Array can be easily accessed if the elements are consecutive locations
- width = w, ith element = base + (i - low) * w ✓
- Base – relative address of A[low]
- i * w + (base - low * w) ✓



Addressing Array Elements: Multi-Dimensional Arrays

```
A : array [1..2,1..3] of integer; (Row-major)
... := A[i,j]
    ↓
t1 := i * 3
t1 := t1 + j
t2 := c // c = baseA - (1 * 3 + 1) * 4
t3 := t1 * 4
t4 := t2[t3]
... := t4
```

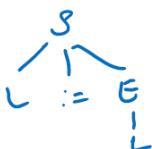
$= base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$
 $= ((i_1 * n_2) + i_2) * w + c$
where $c = base_A - ((low_1 * n_2) + low_2) * w$
with $low_1 = 1; low_2 = 1; n_2 = 3; w = 4$



```

t1 := i * 2
t1 := t1 + j
t2 := t1 * 4
t2 := t2 + k
t3 := array
t4 := t2 * 4 ← w
t5 := t3 [t4]
x := t5

```



$$x[i,j] = a+b \quad \text{width} \geq w \quad 5 \times 10 \quad n_1=5 \quad n_2=10$$

$t_1 := i * 10$

$$t_2 := t_1 + j$$

$$t_3 := t_2 * \omega$$

$$t_f := x[t_3]$$

$$t_5 := a + b$$

$$x[t_3] := t_5$$

-> Control flow

Code

Code- $a < b$ or $c < d$ and $e < f$

if $a < b$ goto Ltrue

goto L1

L1: if $c < d$ goto L2

goto Lfalse

L2: if e < f goto Ltrue

goto Lfalse

L1: if $a < b$ goto L2

goto Lnext

L2: if $c < d$ goto L3

goto L4

L3: t1 := y+ z

x := t1

goto L1

L4: t2 := y - z

x := t2

go

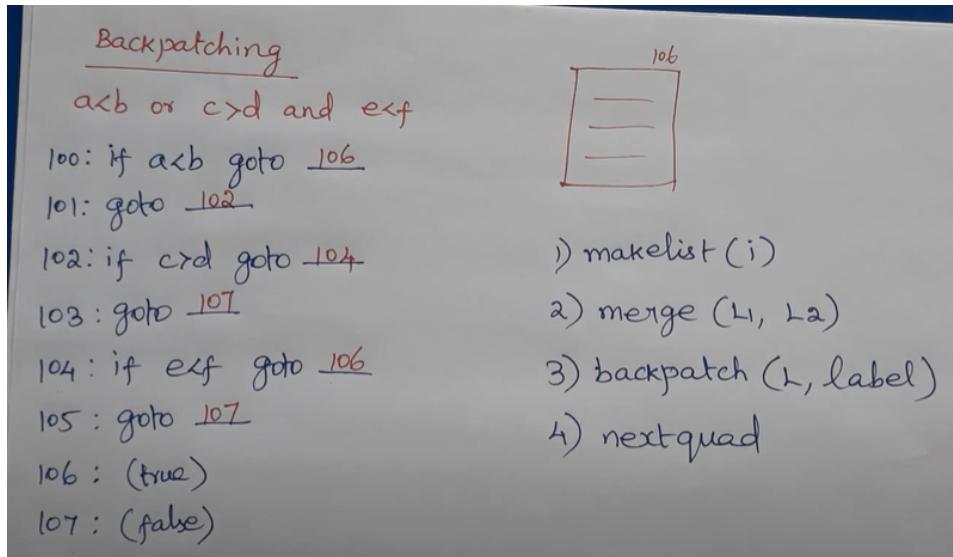
while a < b do

if $c < d$ then

WITCH

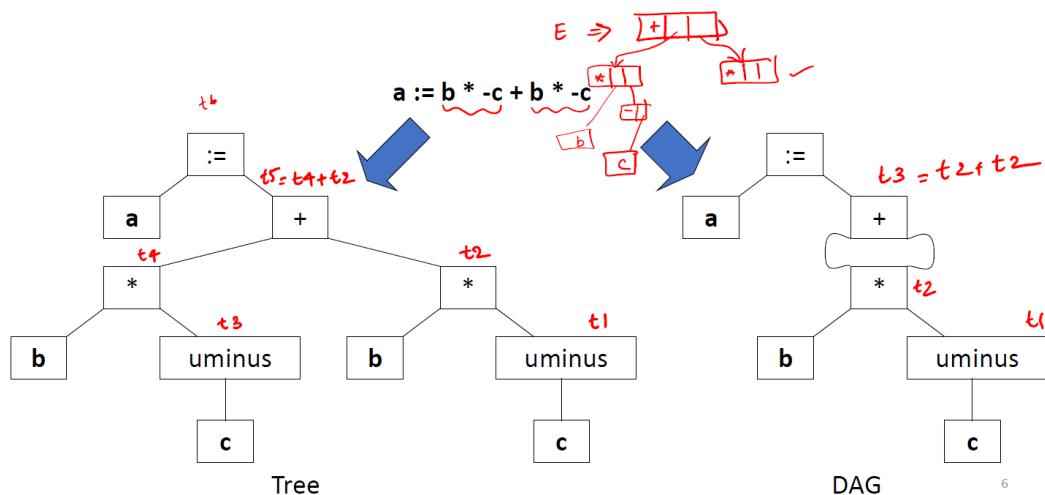
1

->Backpatching

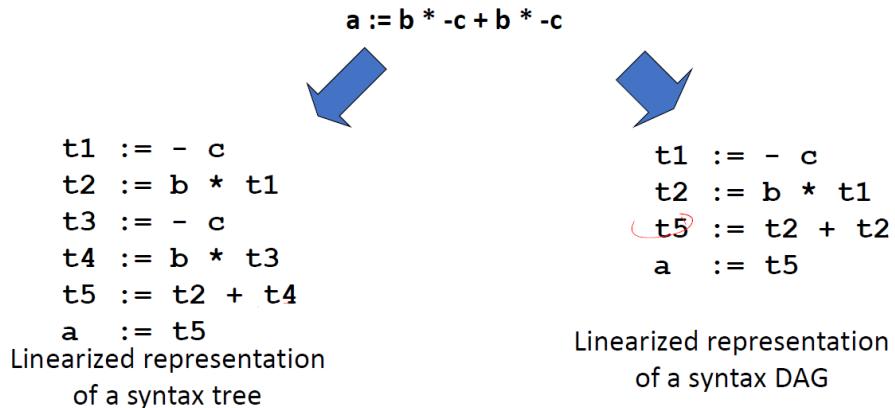


-> Intermediate Code Generation

Abstract Syntax Trees versus DAGs



Three-Address Code



do i = i+1; while (a[i] < v);

```
L:    t1 = i + 1
      i = t1
      t2 = i * 8
      t3 = a[t2]
      if t3 < v goto L
```

Symbolic labels

```
100:   t1 = i + 1
101:   i = t1
102:   t2 = i * 8
103:   t3 = a[t2]
104:   if t3 < v goto 100
```

Position numbers

Quadruples

Has four fields: op, arg1, arg2, and result

Example: $a := b * -c + b^* -c$

- $t1 := -c$
- $t2 := b * t1$
- $t3 := -c$
- $t4 := b * t3$
- $t5 := t2 + t4$
- $a := t5$

	Op	Arg1	Arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Triples

Field corresponding to temporary results are not used and references to instructions are available

	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Indirect triples

In addition to triples we use a list of pointers to triples

	Statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

	Op	Arg1	Arg2
(10)	uminus	c	
(11)	*	b	(0)
(12)	uminus	c	
(13)	*	b	(2)
(14)	+	(1)	(3)
(15)	:=	a	(4)

Code Generator Introduction

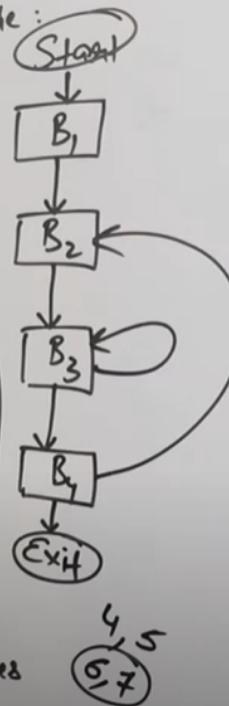
Basic Blocks, Flow Graphs,

Transformations on Basic blocks

Consider the intermediate code:

- 1. $i = 1$ B_1
- 2. $J = 1$ B_2
- 3. $t_1 = 5 * i$ B_3
- 4. $t_2 = t_1 + J$
- 5. $t_3 = 4 * t_2$
- 6. $t_4 = t_3$
- 7. $a[t_4] = -1$
- 8. $J = J + 1$
- 9. if $J \leq 5$ goto (3)
- 10. $i = i + 1$ B_4
- 11. if $i < 5$ goto (2)

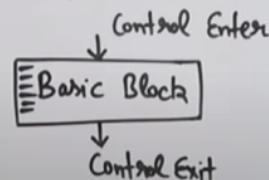
Find the no. of nodes and edges in Control flow graph?



How to find leaders in basic block?

- Step 1) first statement is always a leader
- Step 2) Address of Conditional, unconditional goto are leaders
- Step 3) Next line of Conditional and unconditional goto are leaders.

Basic block properties:



* First line of basic block is leader

What is Basic block and How to partition a Code into Basic Block

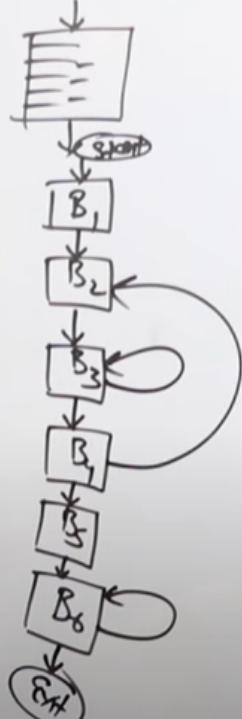
→ Sequence of intermediate code with single entry and single exit.

→ No jumps in the middle.

Algorithm to make basic blocks.

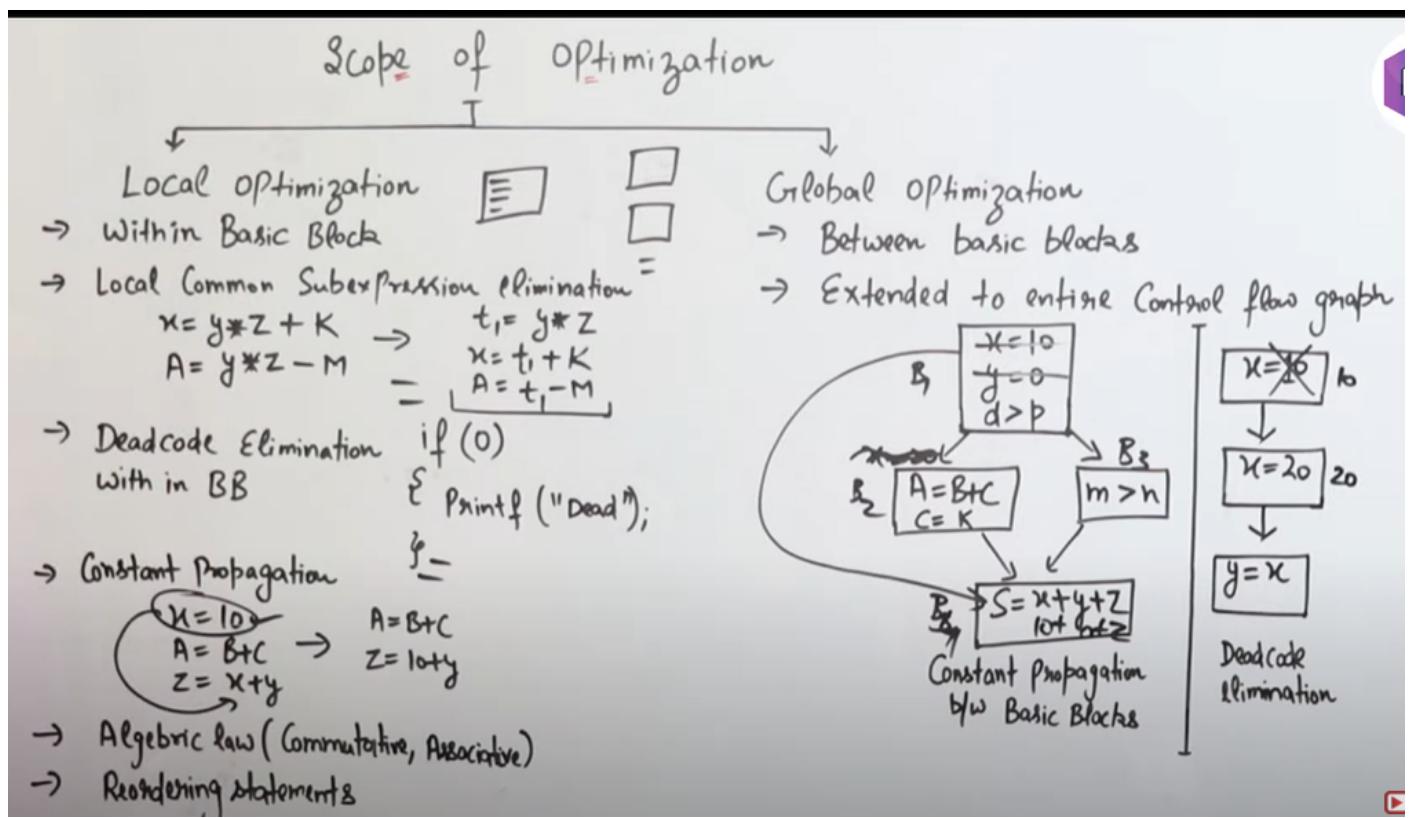
- 1) Identify leaders first
- 2) First line of code is leader
- 3) Address of Conditional, unconditional goto are leaders.
- 4) Immediate next line of goto are leaders.
- 5) Make basic block from leader to line before next leader.

- 1) $i = 1$ B_1
- 2) $J = 1$ B_2
- 3) $t_1 = 10 * i$ B_3
- 4) $t_2 = t_1 + J$
- 5) $t_3 = 8 * t_2$
- 6) $t_4 = t_3 - 88$
- 7) $a[t_4] = 0.0$
- 8) $J = J + 1$
- 9) if $J \leq 10$ goto (3)
- 10) $i = i + 1$
- 11) if $i \leq 10$ goto (2) B_4
- 12) $i = i + 1$ B_5
- 13) $t_5 = i - 1$
- 14) $t_6 = 88 * t_5$
- 15) $a[t_6] = 1.0$
- 16) $i = i + 1$
- 17) if $i \leq 10$ goto (13)



8:20 / 8:30





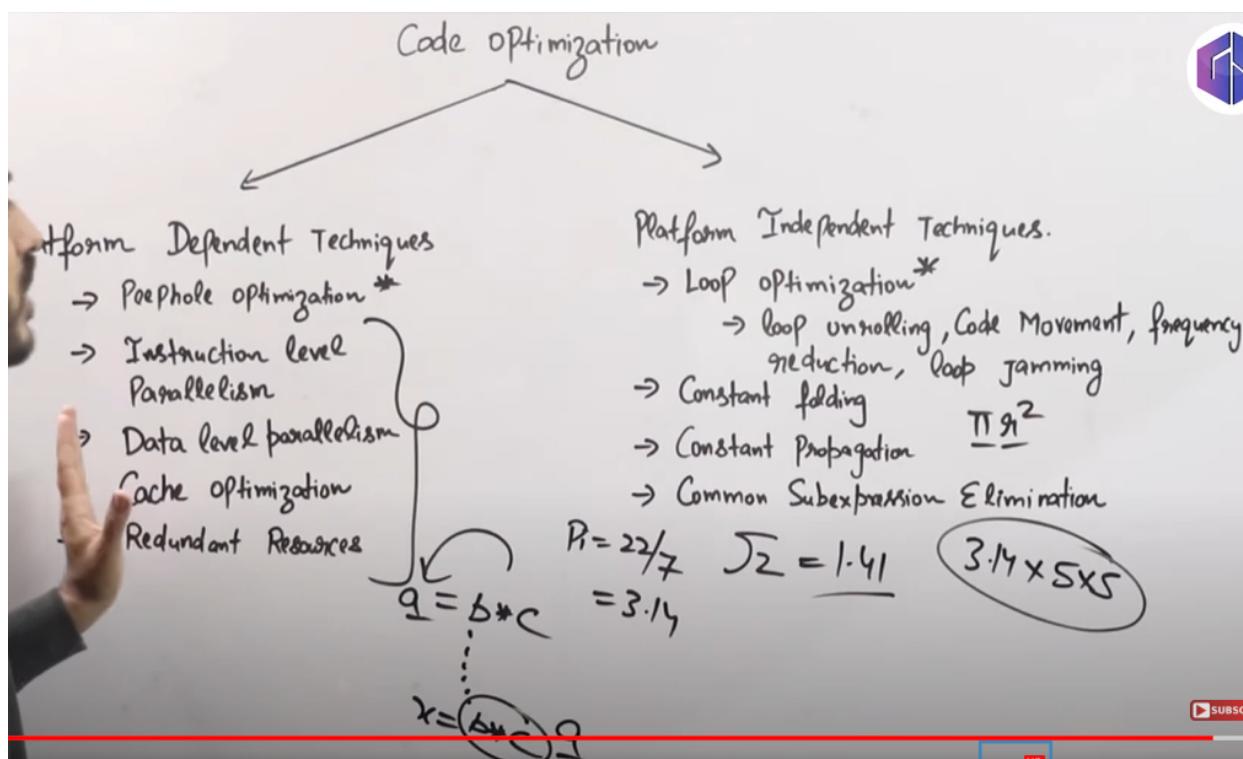
Code Generation Algorithm with examples

DAG

Code generation from DAG

Loops in Flow graphs

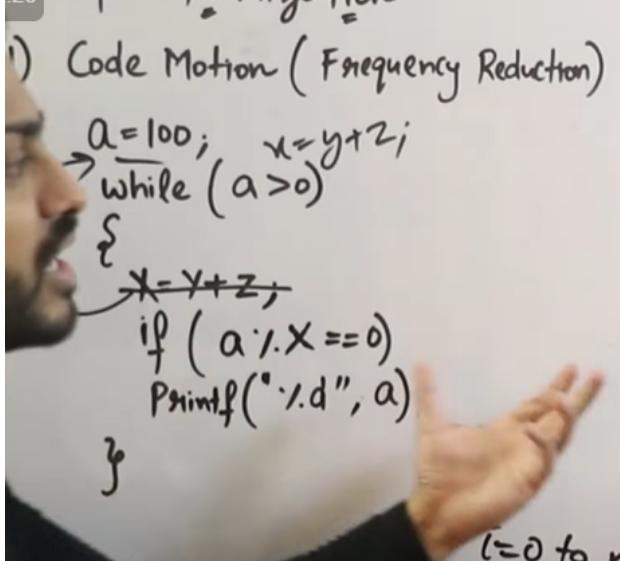
Code Optimization



Peephole Optimizations.

Peephole optimization		3) Simplify Algebraic Expressions	5) Deadcode Elimination
2.25	<p>1) Redundant Load and Store</p> $a = b + c \quad \text{Add } a, b, c$ $d = a + e \quad \text{Add } d, a, e$ <pre> Mov b, R0 Add C, R0 Mov R0, a Mov a, R0 Add e, R0 Mov R0, d </pre>	$\cancel{a = a + a}$ $\cancel{a := a * 1}$ $\cancel{a := a / 1}$ $\cancel{a := a - 0}$ <p>4) Replace slower Instructions with faster</p> $\times \text{Add } \#1, R \Rightarrow \text{INC } R \checkmark$ $\times \text{Sub } \#1, R \Rightarrow \text{DEC } R \checkmark$ $\rightarrow \begin{array}{l} \text{aload } x \\ \text{aload } x \\ \text{Mul} \end{array} \quad \left(\begin{array}{l} \text{aload } x \\ \text{dup} \\ \text{Mul} \end{array} \right)$	<pre> int dead(void) { int a = 10; int b = 20; int C; C = a * 10; return C; b = 30; b = b * 10; return 0; } </pre>

Loop optimization

2.25	Loop optimization	(1) Code Motion (Frequency Reduction)	(2) Loop fusion	(3) Loop unrolling
	 <pre> a = 100; x = y + z; while (a > 0) { x = y + z; if (a % x == 0) printf(".1.d", a); } </pre> <p style="text-align: center;">$i=0 \text{ to } n \quad n$</p> <p style="text-align: center;">$\left\{ \begin{array}{l} \text{for } j=0 \text{ to } n \quad n^2 \\ \text{for } k=0 \text{ to } n \quad n^3 \end{array} \right.$</p>	<pre> int i, a[100], b[100] for (i=0; i<100; i++) a[i] = i; </pre>	<pre> int i, a[100], b[100] for (i=0; i<100; i++) b[i] = 2; </pre>	<pre> for (i=0; i<5; i++) printf("Varun"); </pre> <p style="text-align: center;"> $Pf(v)$ $Pf(v)$ $Pf(v)$ $Pf(v)$ $Pf(v)$ </p>

Global Data Flow Analysis - Data flow equations, Reaching Definitions.

2.25

node	USE	DEF	1st go		out-Def		$IN[n] = USE[n] \cup (OUT[n] - DEF[n])$	$OUT[n] = \bigcup_{S \in SUCCESS[n]} IN[S]$
			IN	OUT	IN	OUT		
1	q, r_2, v	b, s, u	q, q, v	r_1, s, v	r_1, r_2, v	$r_1, 4, s, v$		
2	r_1, u	v	r_1, u	v, r_2	r_1, u	r_1, v		
3	s, u	q	s, u	v, r_2	v, r_2, s, u	r_1, v		
4	v, r_2	q	v, r_2	q, r_1, v	r_1, v	r_1, r_2, v		

$IN[B]$ = Set of variables live at beginning of B

$OUT[B]$ = " " " " Just after B

$USE/GEN[B]$ = Variables that are used in B (Variables in R.H.S before any assignment but not in L.H.S of prior statement in B)

$DEF/KILL[B]$ = Variables that are assigned a value in B (Variable in L.H.S)

