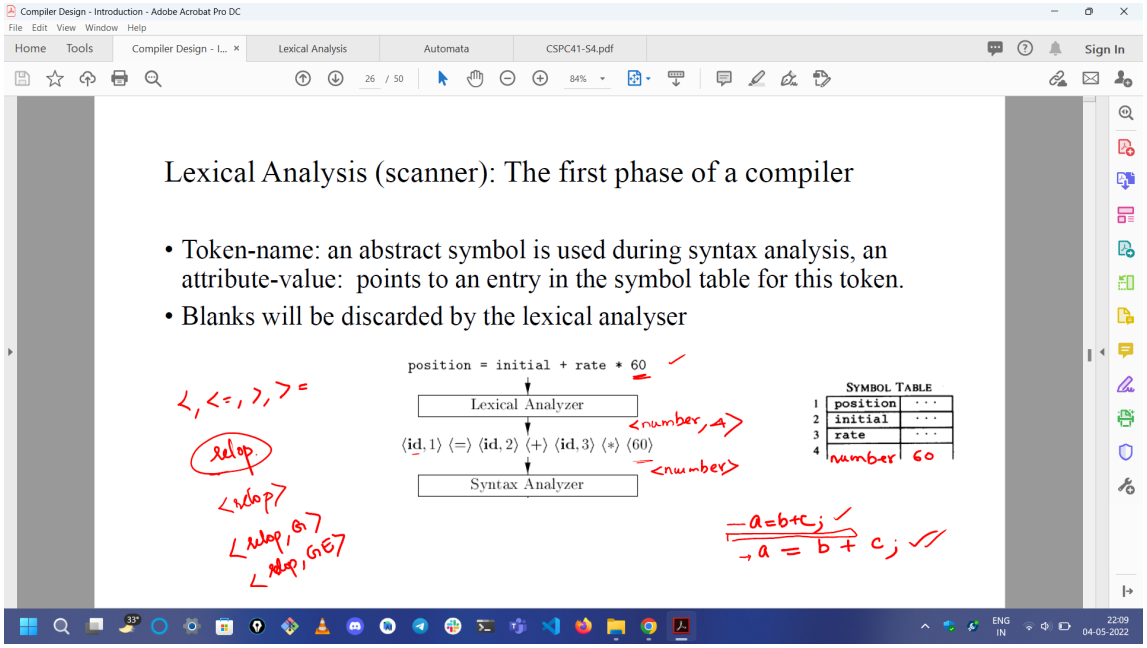
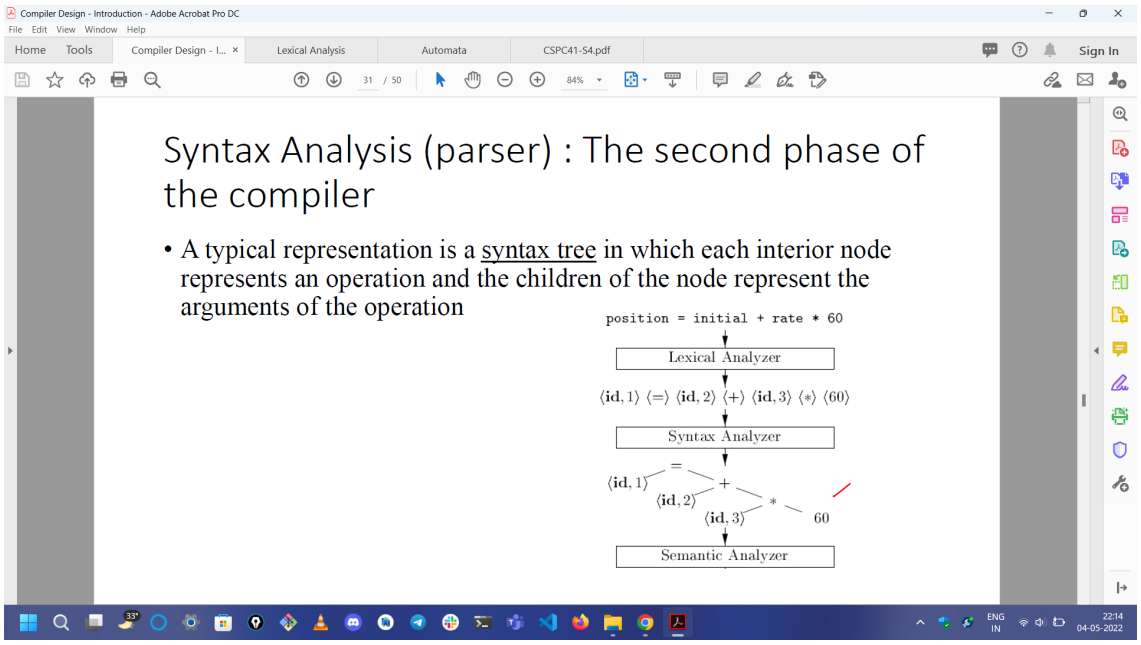
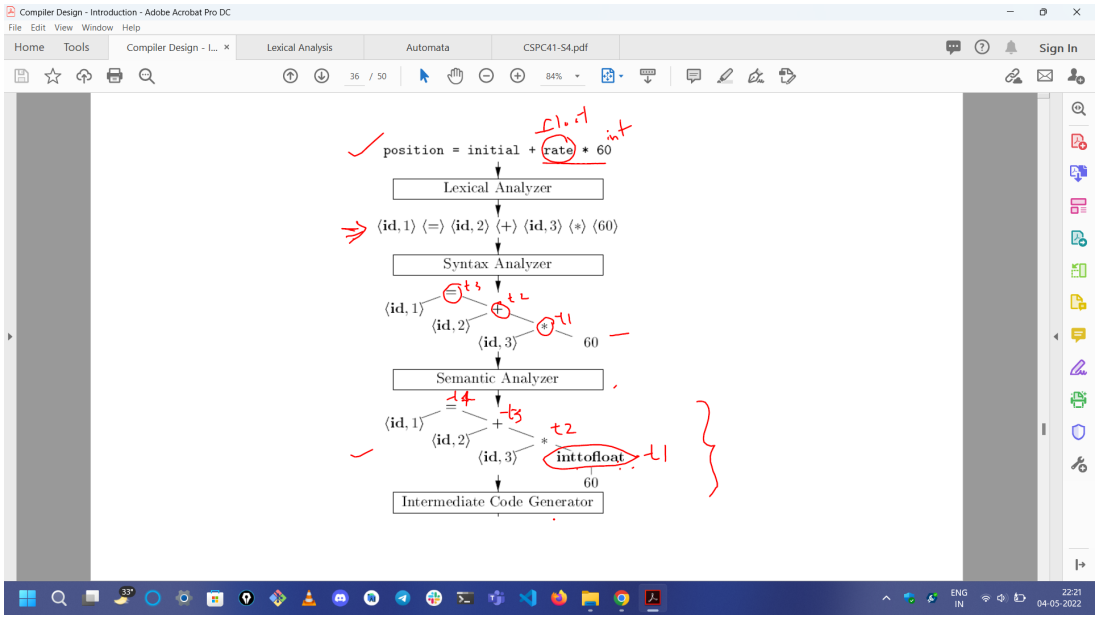
Compiler

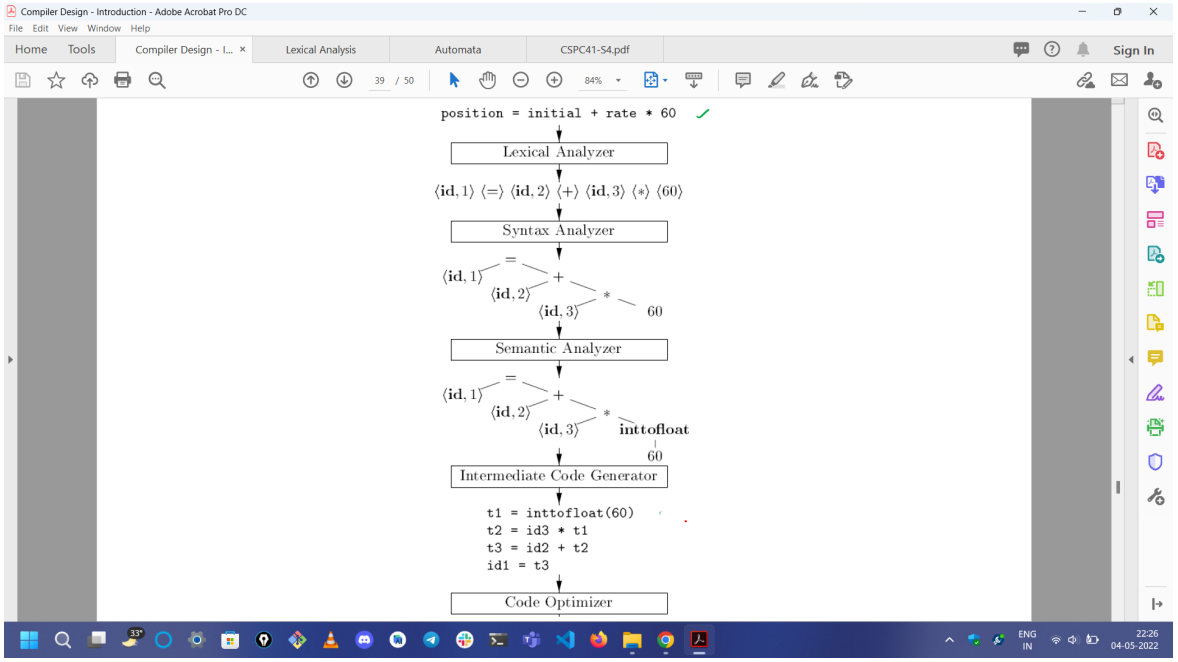
- [ ] Phases of the compiler.
- [ ] Lexical Analysis
- [ ] Constructing DFA from RE
- [ ] Parsers - LL, SLR, CALR, LALR
- [ ] Unit 3 - all topics (except PPT on Semantic Phase of the Compiler).
- [ ] Code Generator Introduction
- [ ] Basic Blocks, Flow Graphs, Transformations on Basic blocks
- [ ] Code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

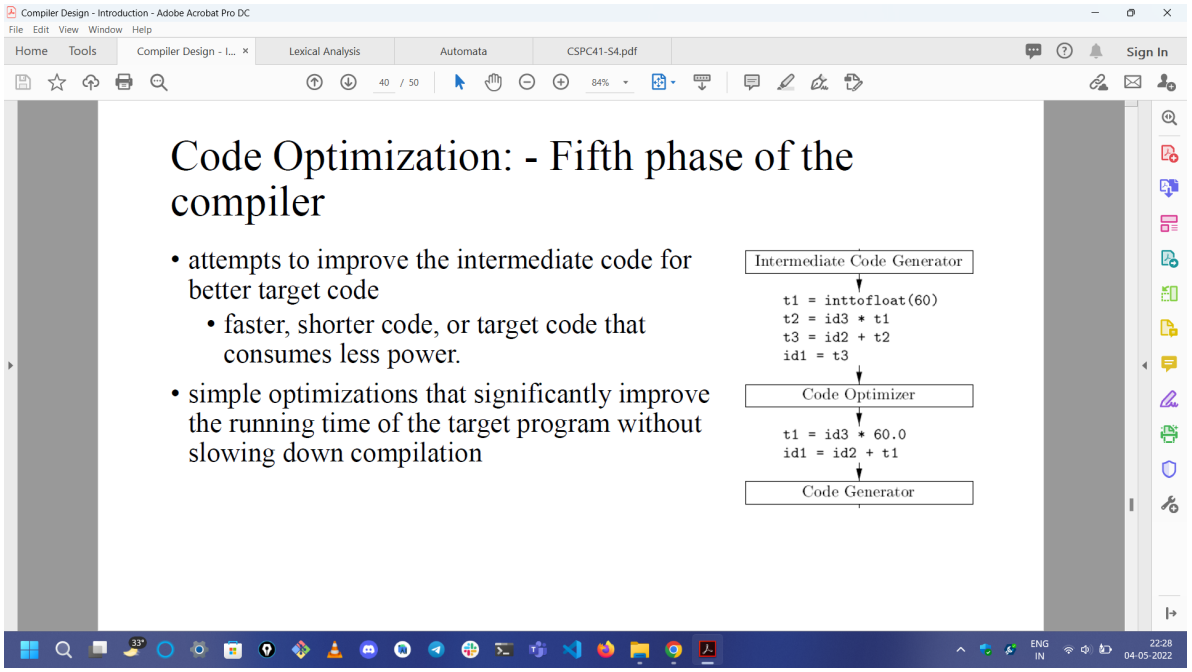
Phases of the compiler.

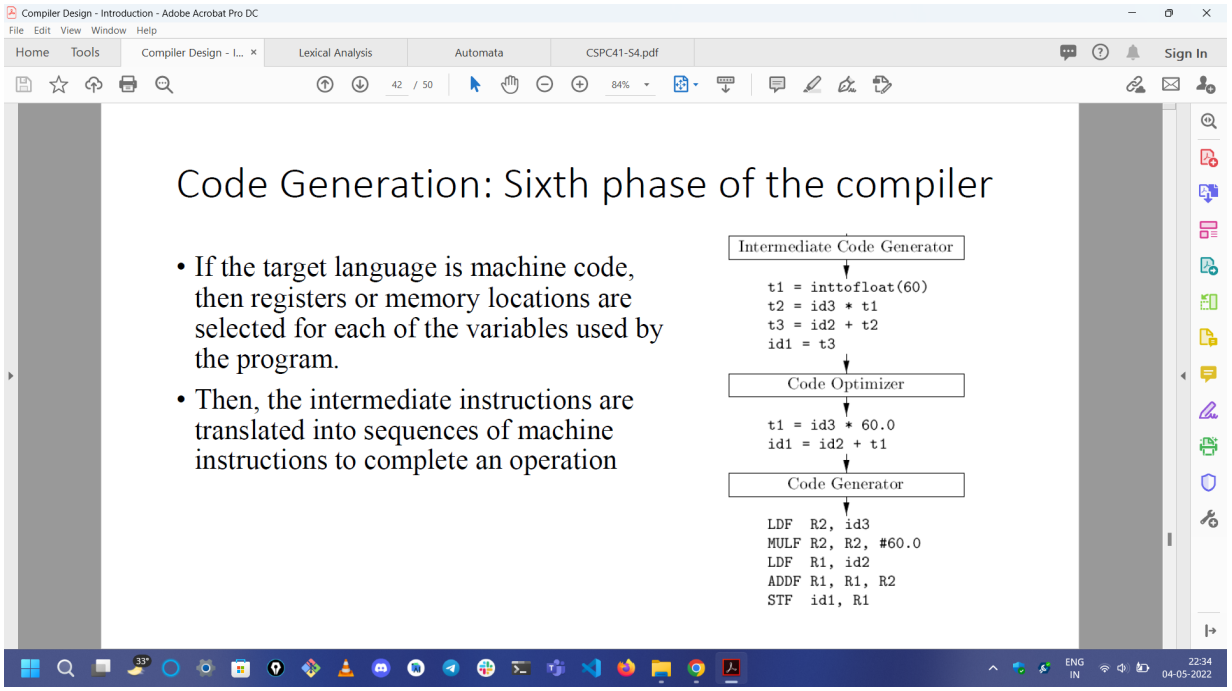




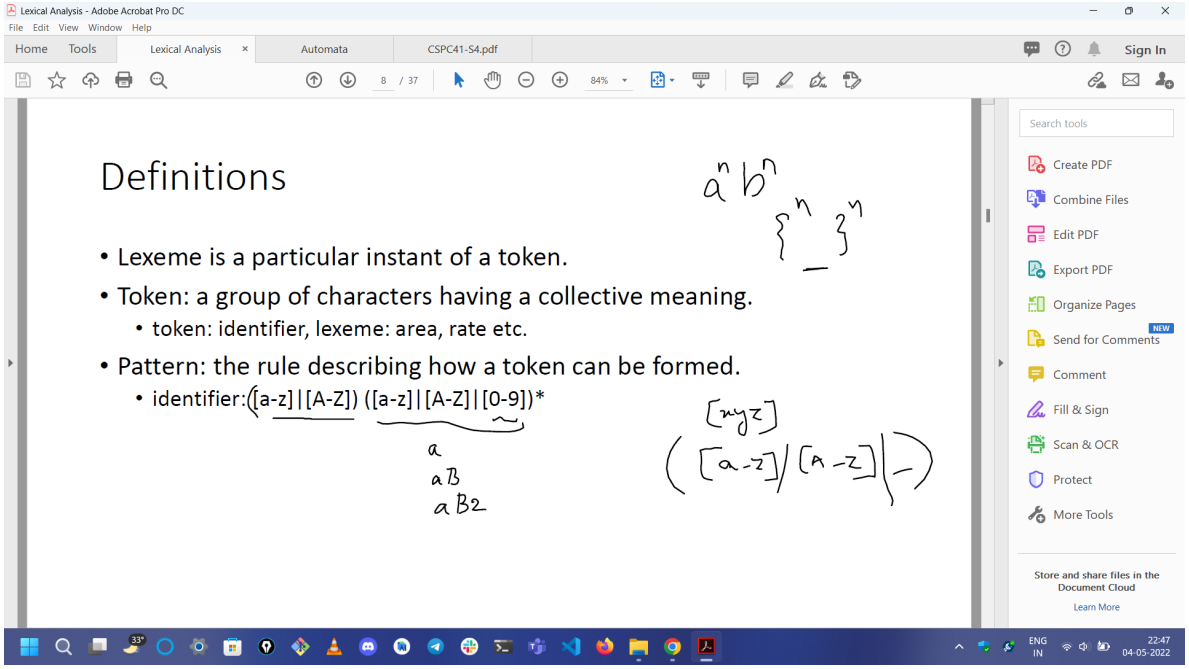



Compiler Design - Introduction - Adobe Acrobat Pro DC
position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer


Compiler Design - Introduction - Adobe Acrobat Pro DC
Code Optimization: - Fifth phase of the compiler
• attempts to improve the intermediate code for better target code
• faster, shorter code, or target code that consumes less power.
• simple optimizations that significantly improve the running time of the target program without slowing down compilation
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator


Compiler Design - Introduction - Adobe Acrobat Pro DC
Code Generation: Sixth phase of the compiler
• If the target language is machine code, then registers or memory locations are selected for each of the variables used by the program.
• Then, the intermediate instructions are translated into sequences of machine instructions to complete an operation
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Lexical Analysis

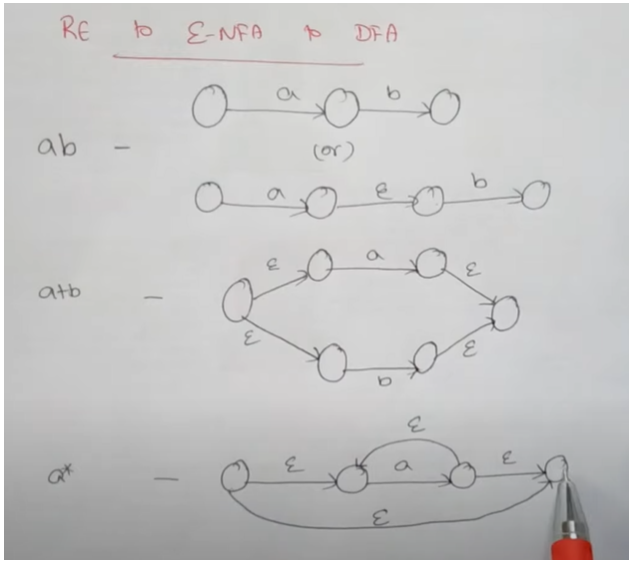


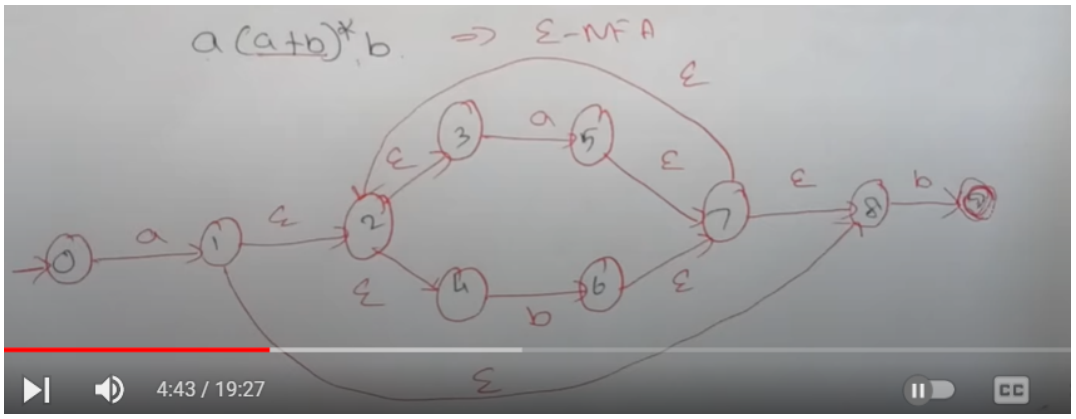
Count vowels and const

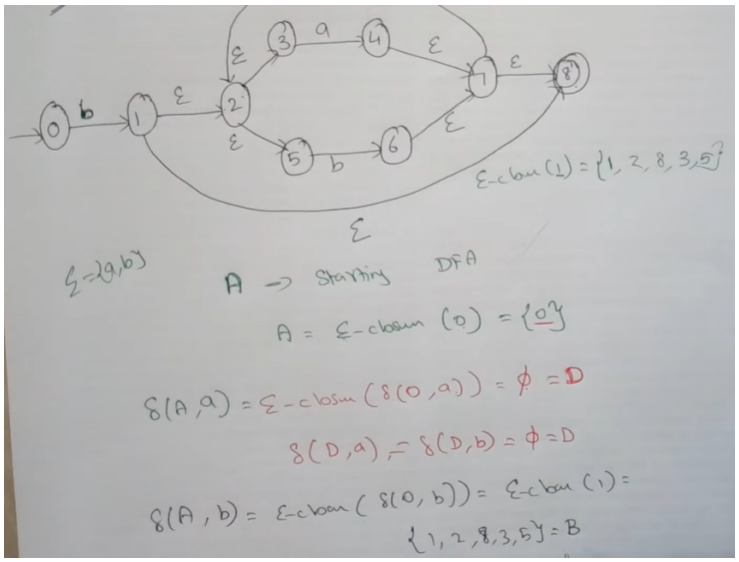
```
%{
    int vow_count=0;
    int const_count =0;
%}

%%
[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;
}
```

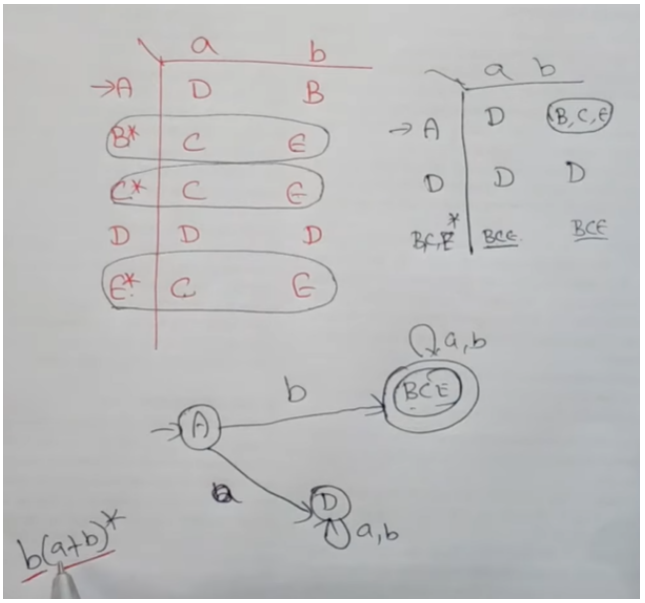
```
%{
    #include<stdio.h>
    #include<string.h>
    int i;
%}

%%

[a-z A-Z]*  {
        for(i=0;i<=yyleng;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

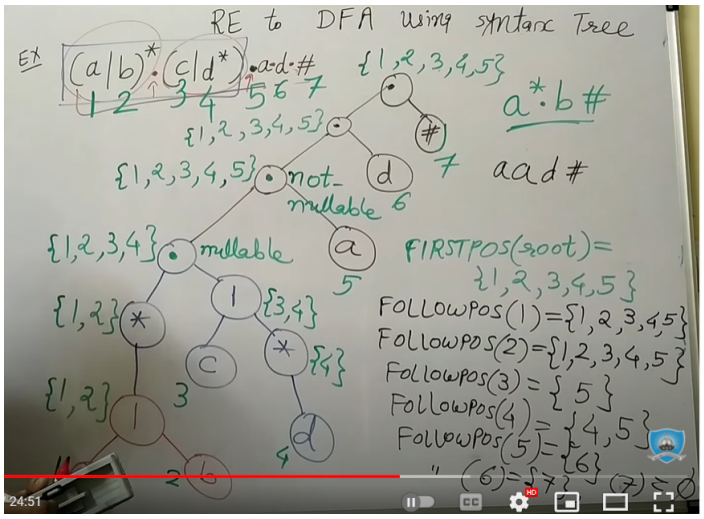


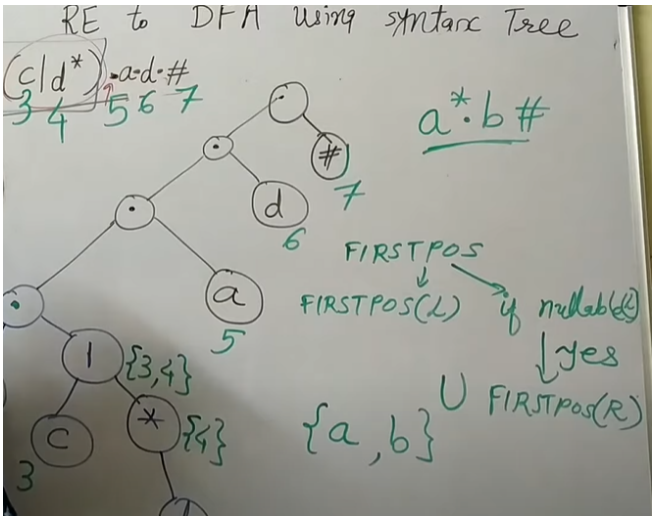


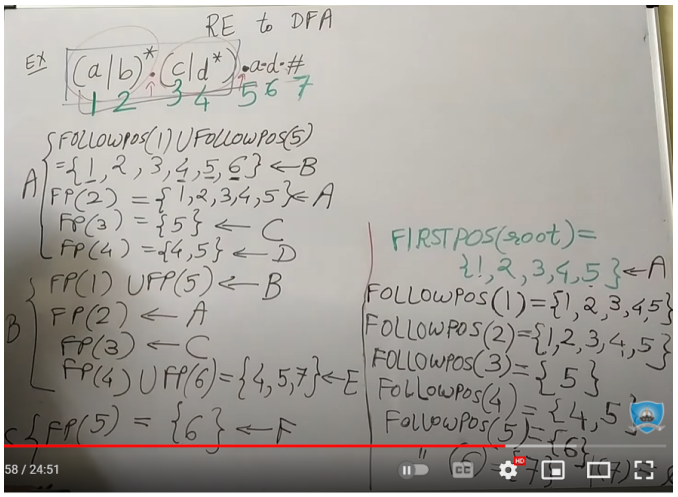


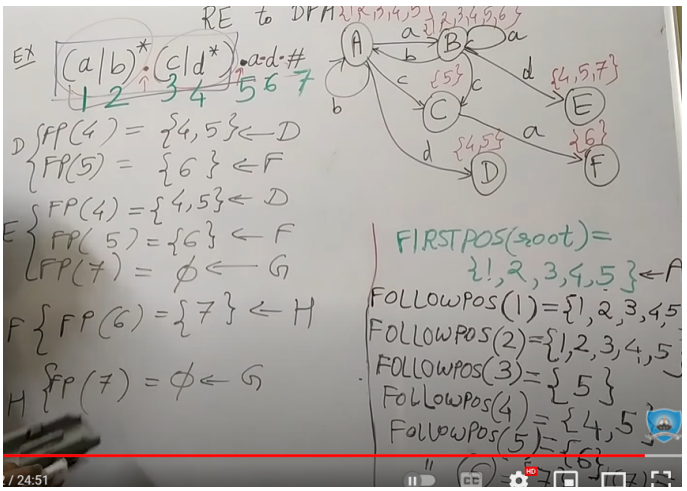


Using syntax tree









Comments RE

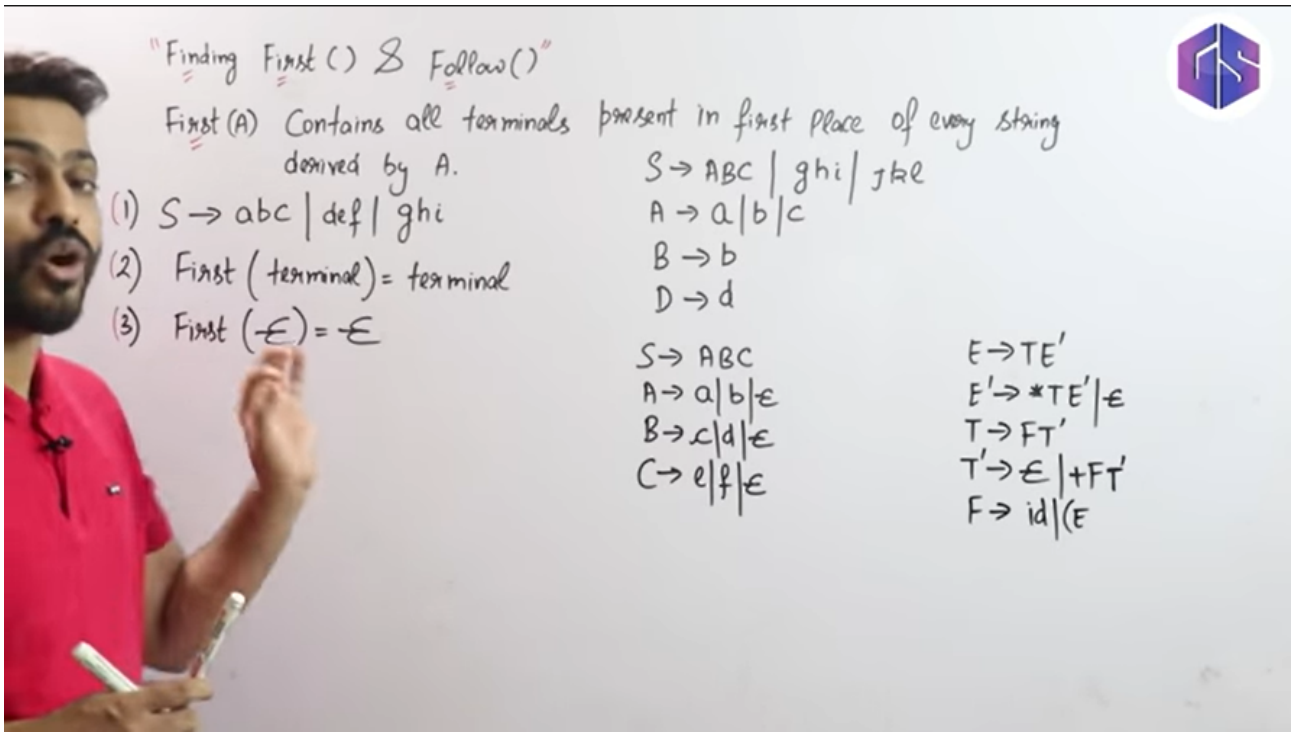
```
"//".*\n                { fprintf(yyout, ""); }
"/*"[^*/]*"*/"          { fprintf(yyout, ""); }
```

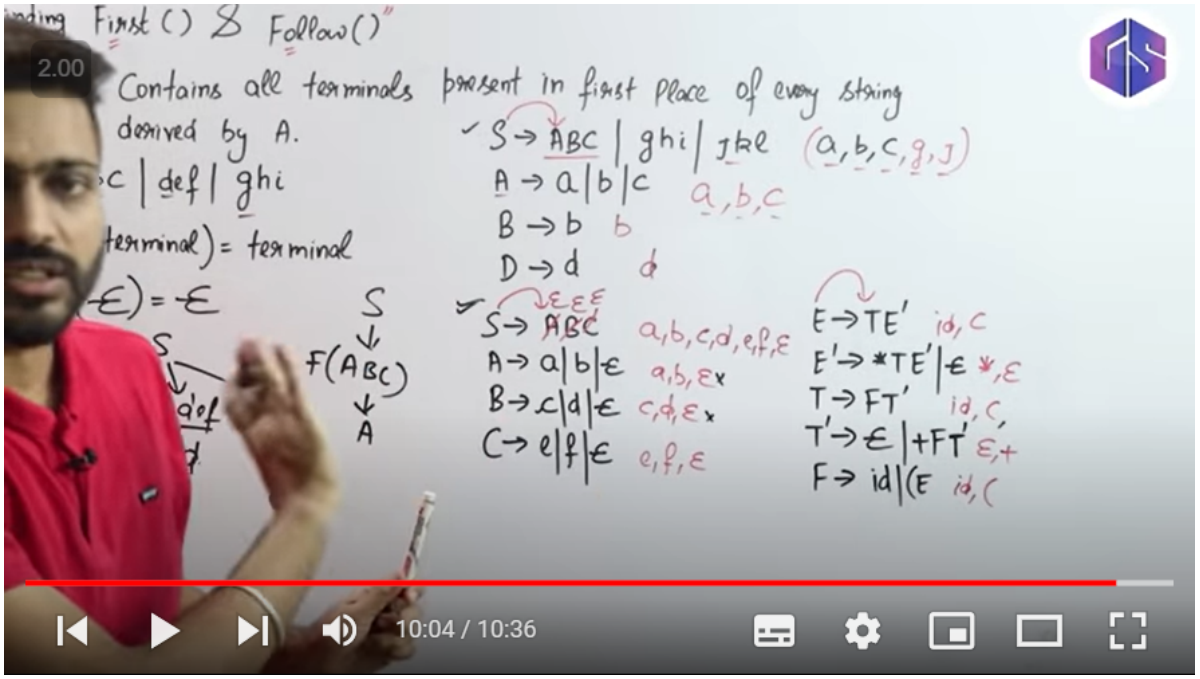
Parsers - LL, SLR, CALR, LALR

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

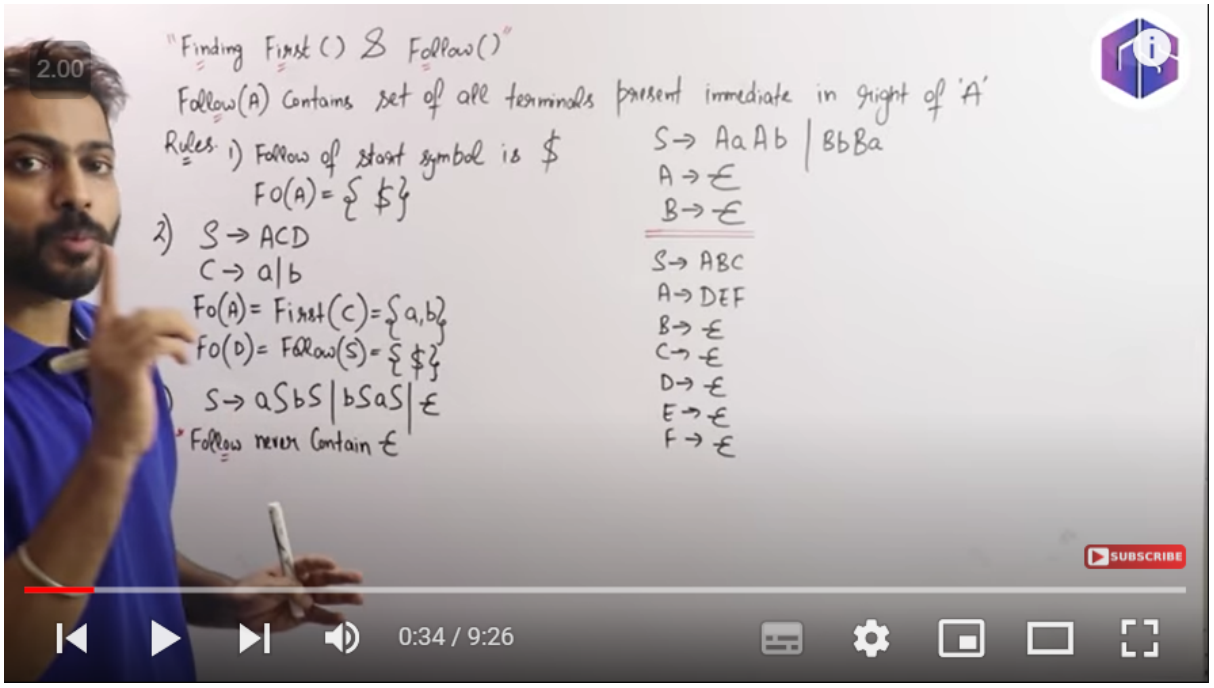
Always start finding Firsts from the terminals and then go up above for variables dependent on them.

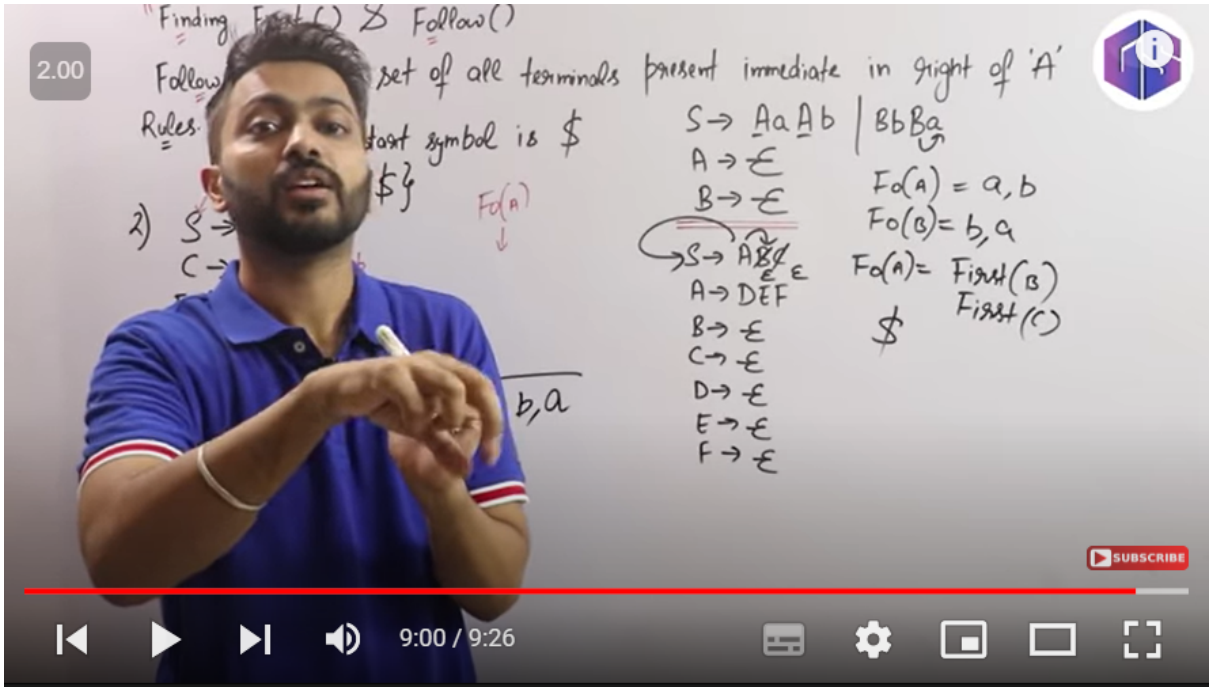




NOTE : Main thing not to forget is: Whenever something first includes an epsilon, then you can add to it by the next term also. See the middle-down example.

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.

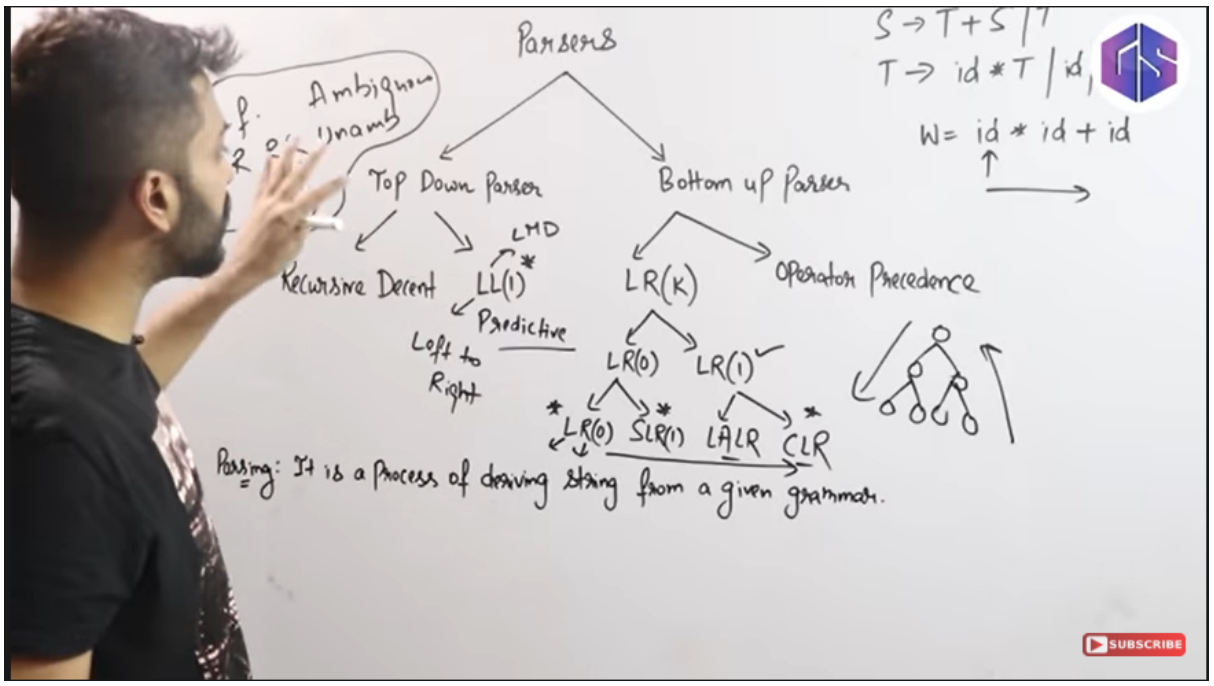
General Rules for finding Follow :

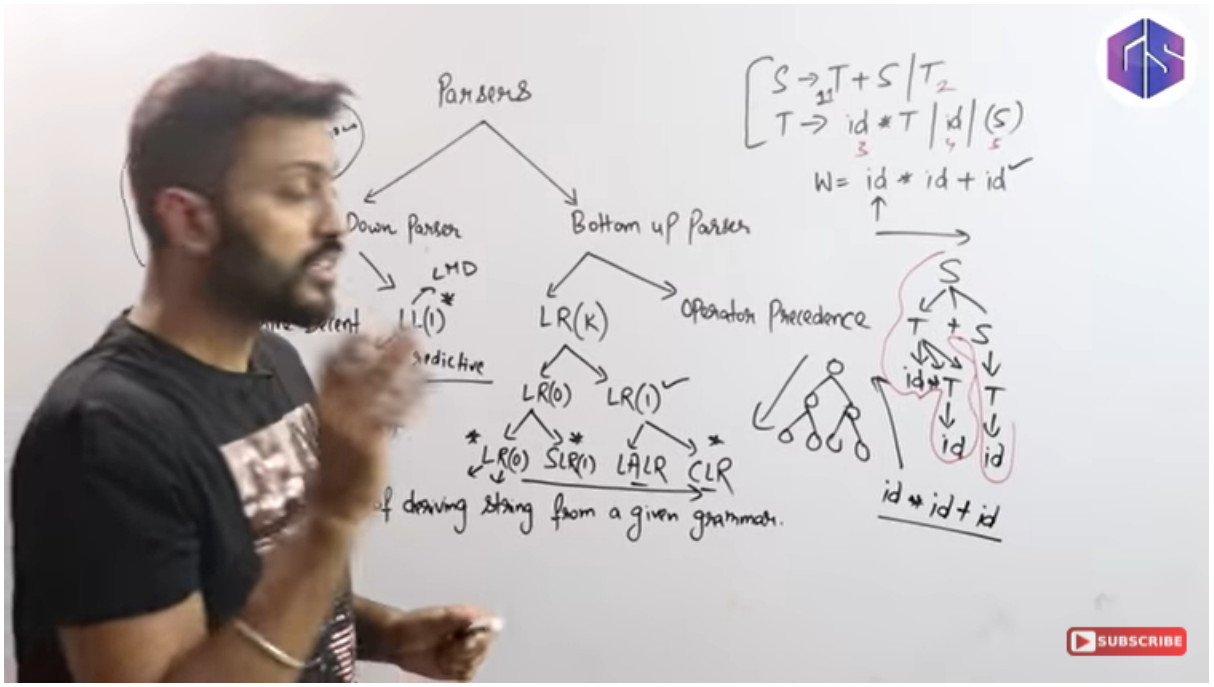
Follow(S) nikalna hai to arrow(kisi bhi) ke right side main S ki occurrences ke just **right** main dekho:

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.

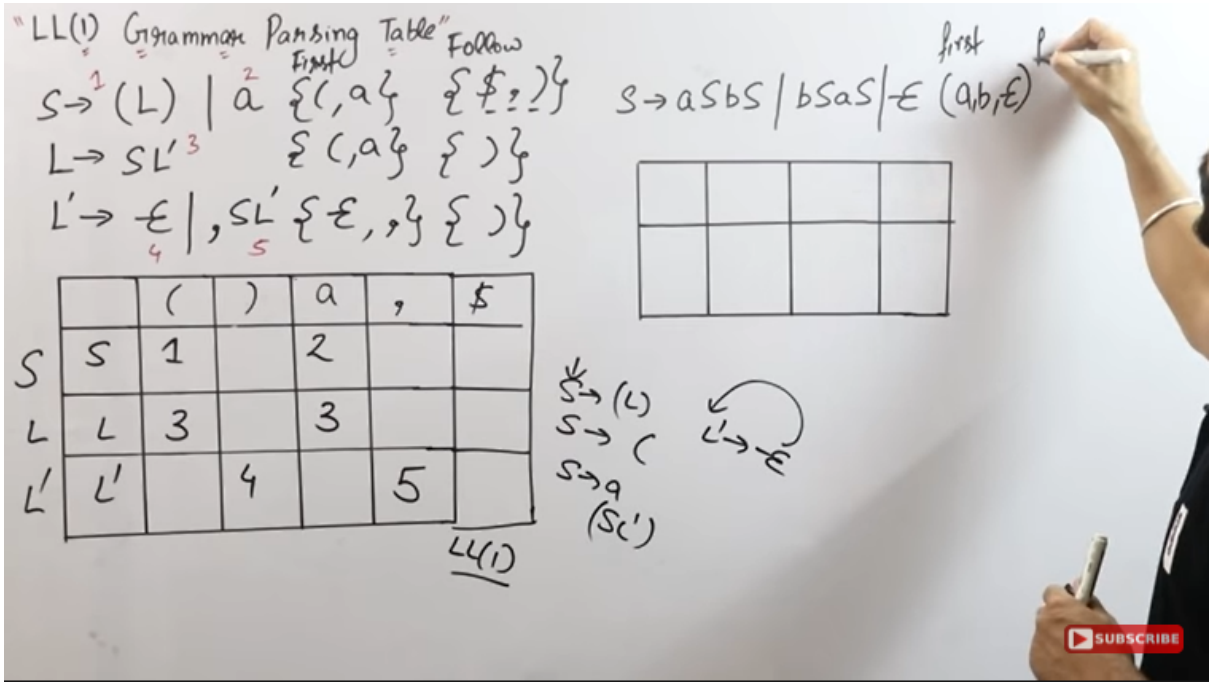
Parsing

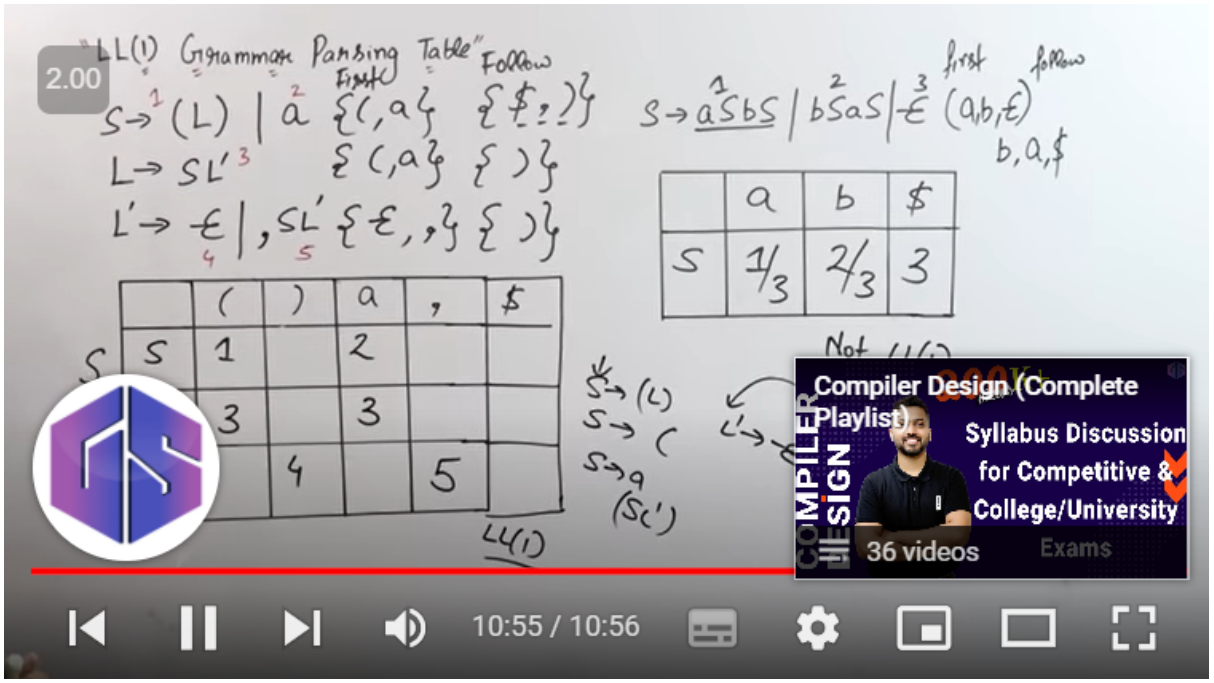
Syntax Analysis ke liye ye phase hai. Parsing Tree se ek-ek karke check karte hai if string belongs to the grammar or not. Grammar used here is CFG(Context Free Grammar).





LL(1) Parser





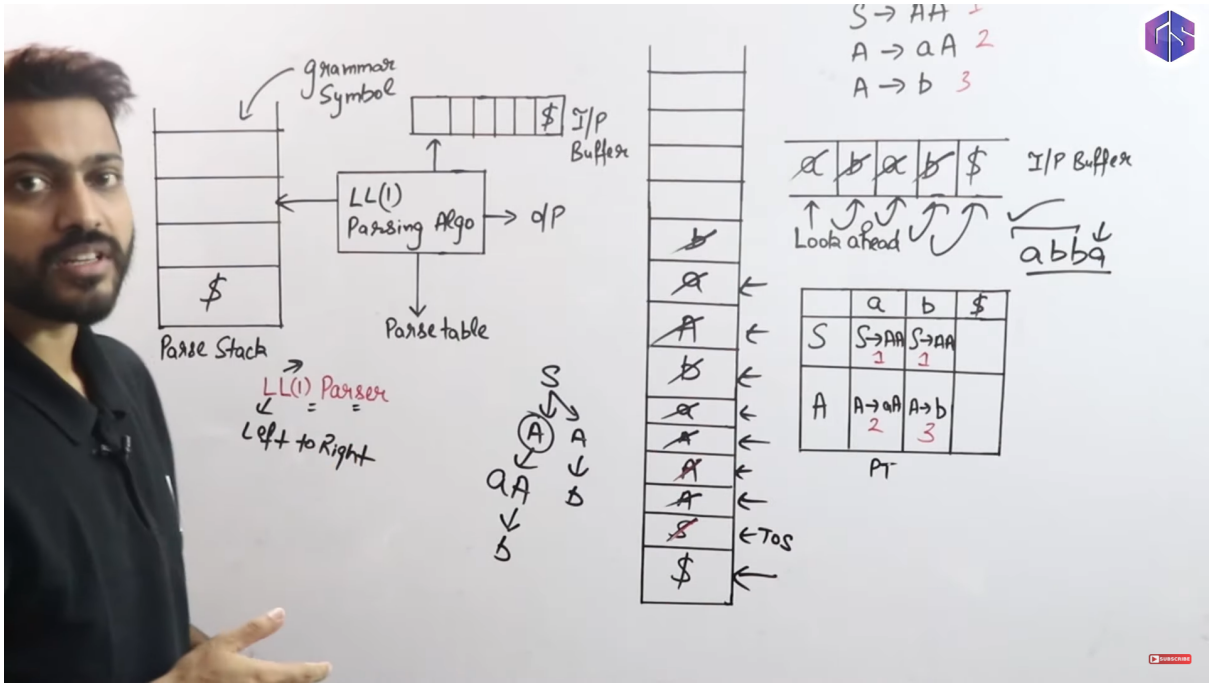
In above screenshots, Grammar on the left is accepted by LL(1) Parser and Grammar on the right is rejected by LL(1) Parser:

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

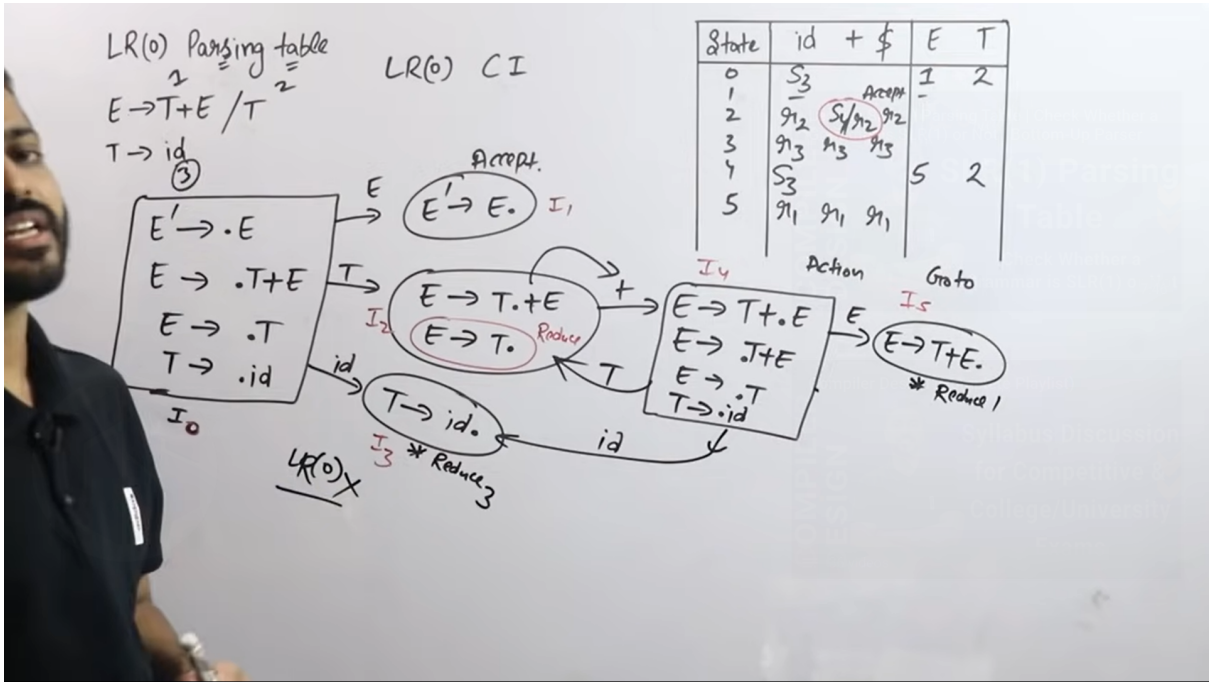
CI -> canonical items
I0, i1 are canonical items

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

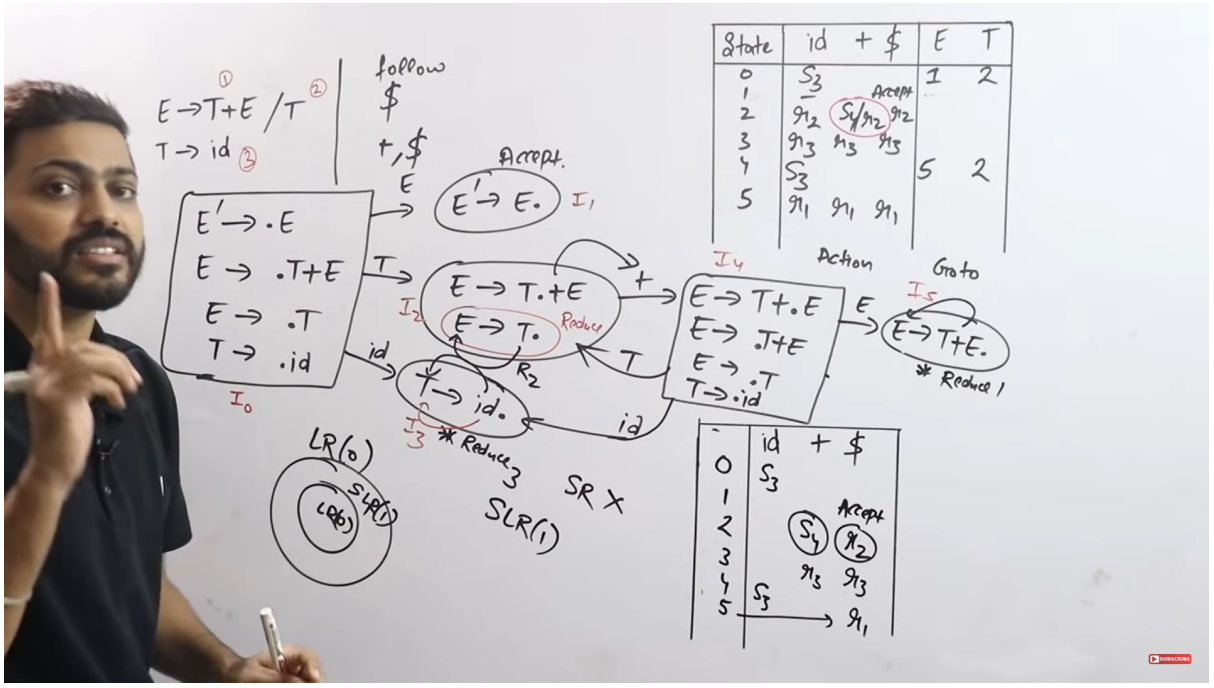
Grammar is LR(0) accepted - 1 cell shouldn't contain shift-reduce or reduce-reduce conflict.



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 :

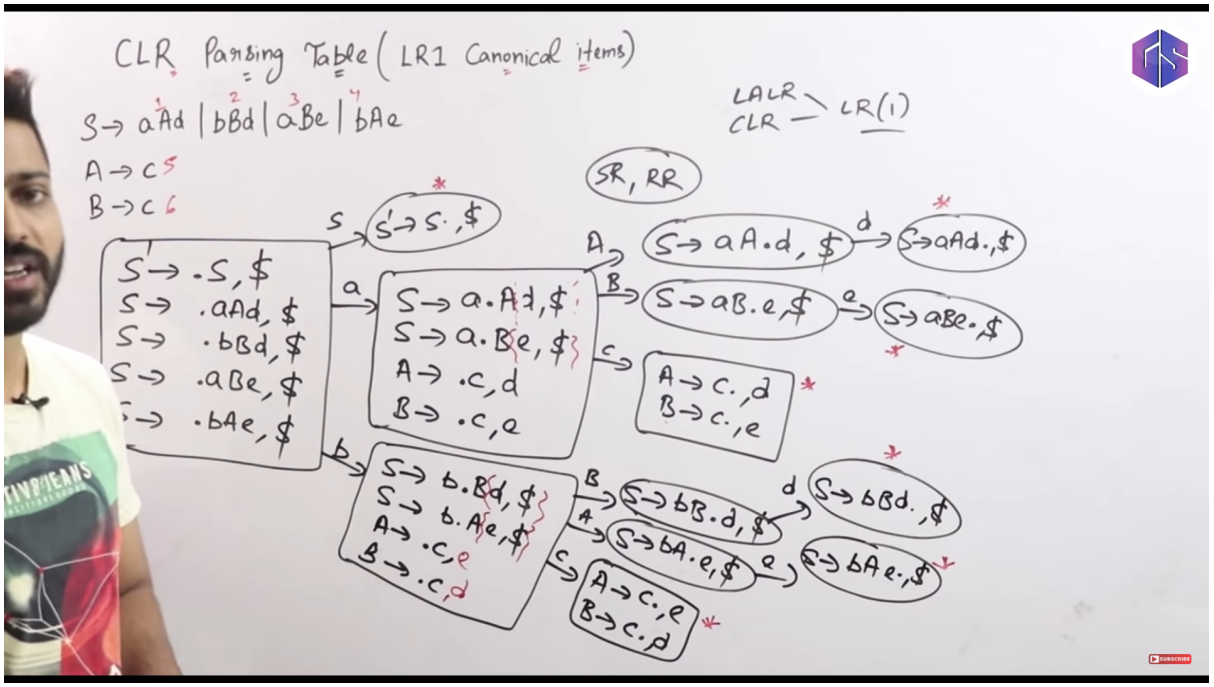
LR1 canonical item (look ahead)

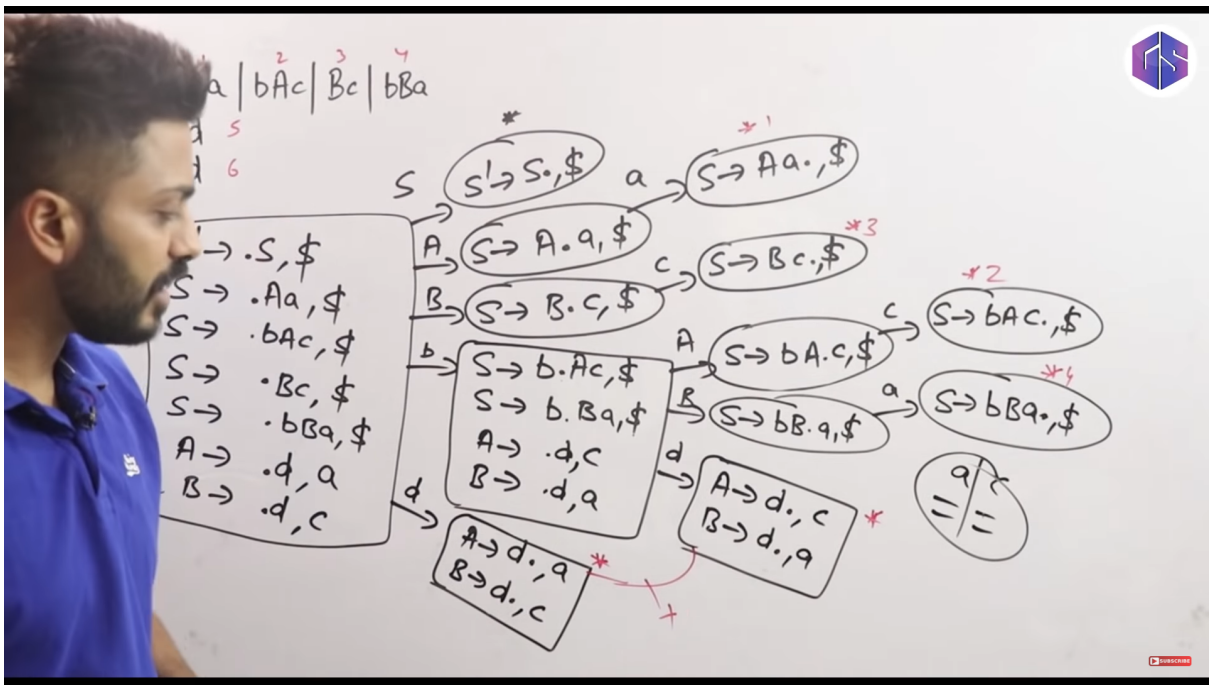
Write reduced state in the look-ahead column

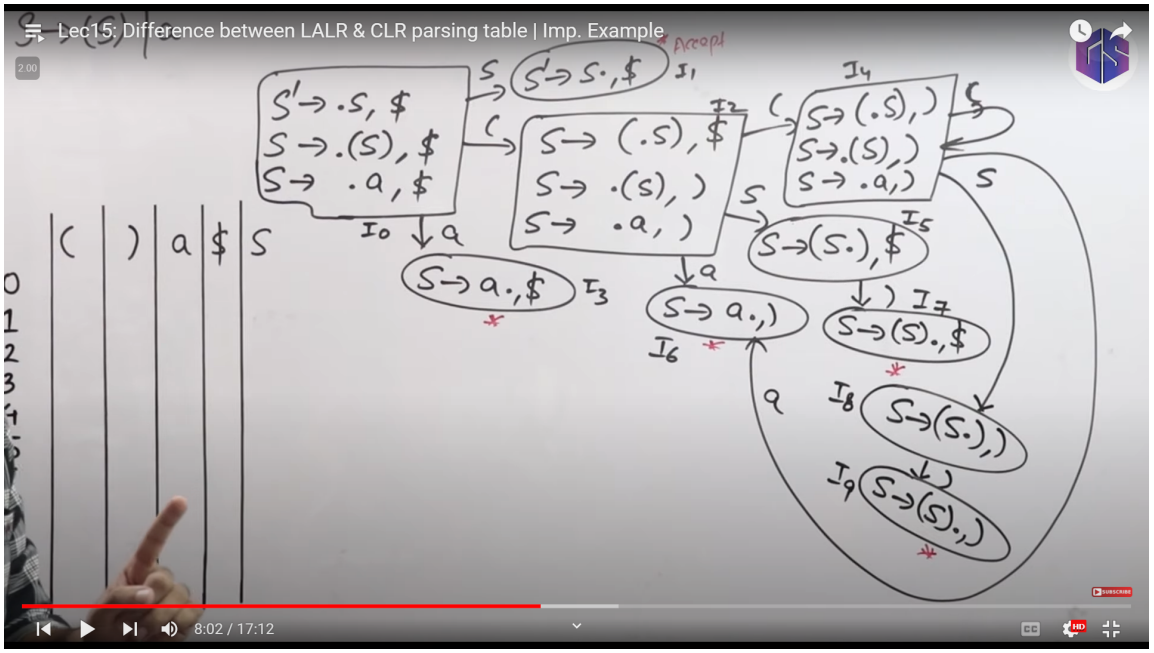
After ' , ' add look ahead

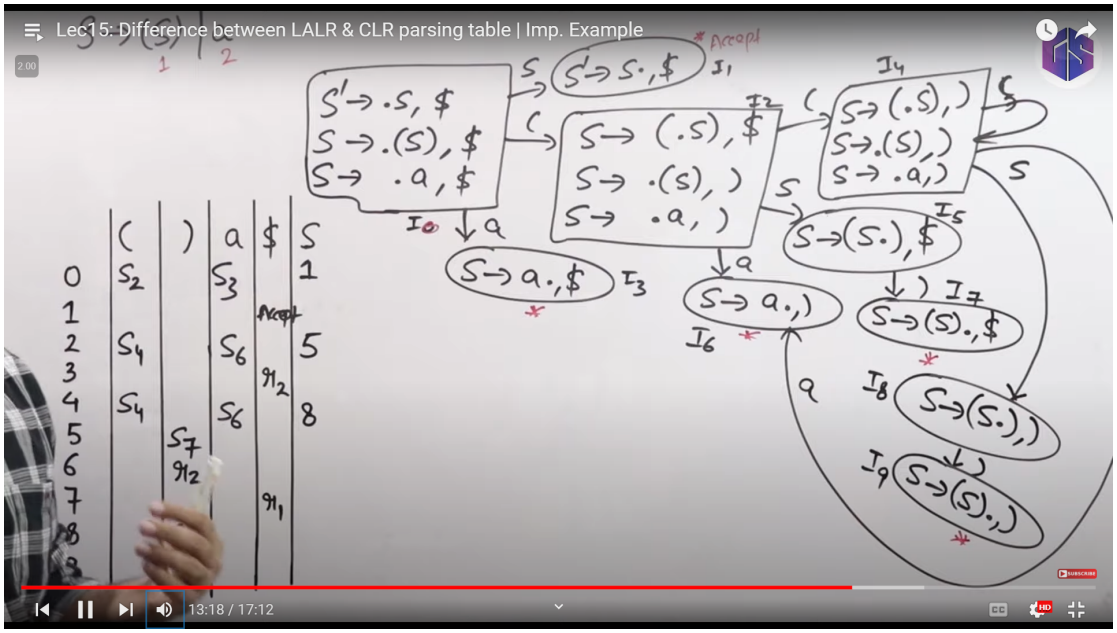
Start with S' but write $ as look ahead,

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

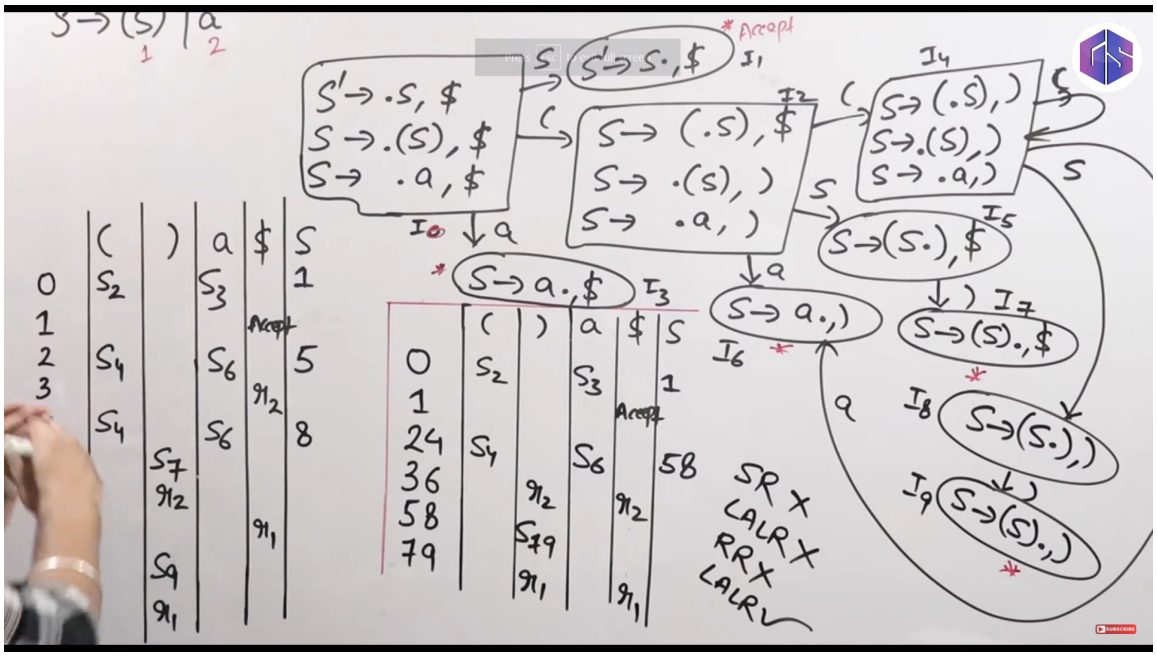








LALR:(merged CLR)

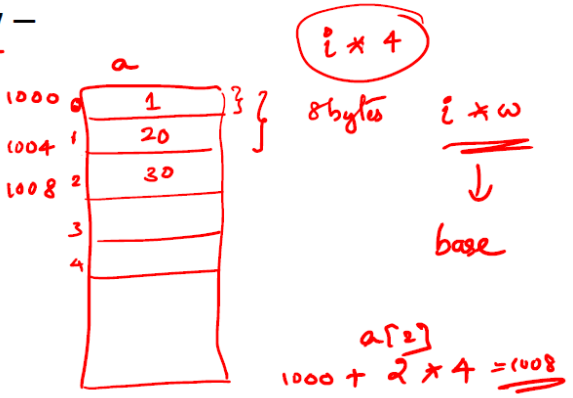


Unit 3 - all topics (except PPT on Semantic Phase of the Compiler).

-> 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]
```

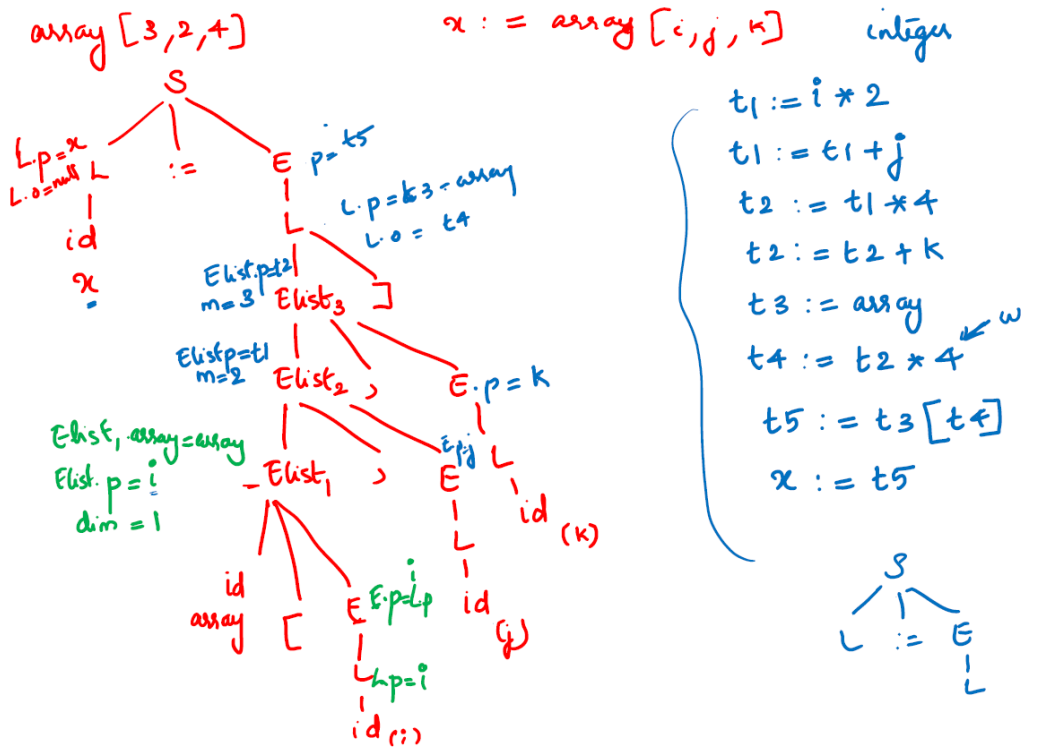
$= base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

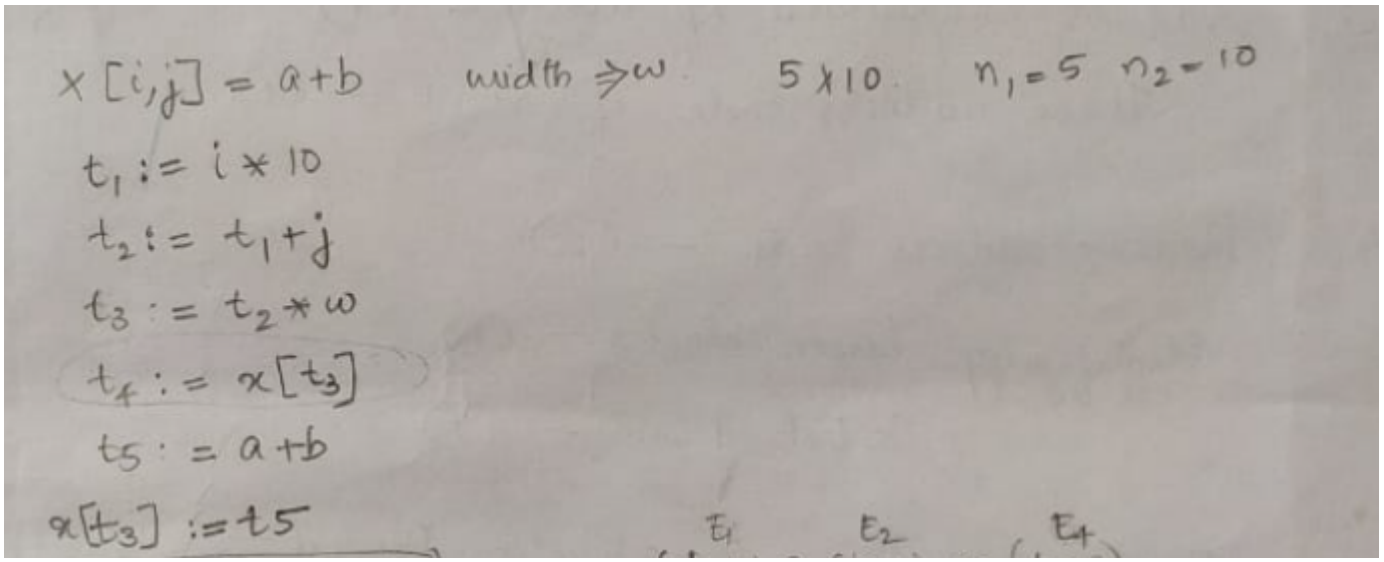
$= ((i_1 * n_2) + i_2) * w + c$

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 * 3
t1 := t1 + j
t2 := c   // c = base_A - (1 * 3 + 1) * 4
t3 := t1 * 4
t4 := t2[t3]
...  := t4
```





-> Control flow

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

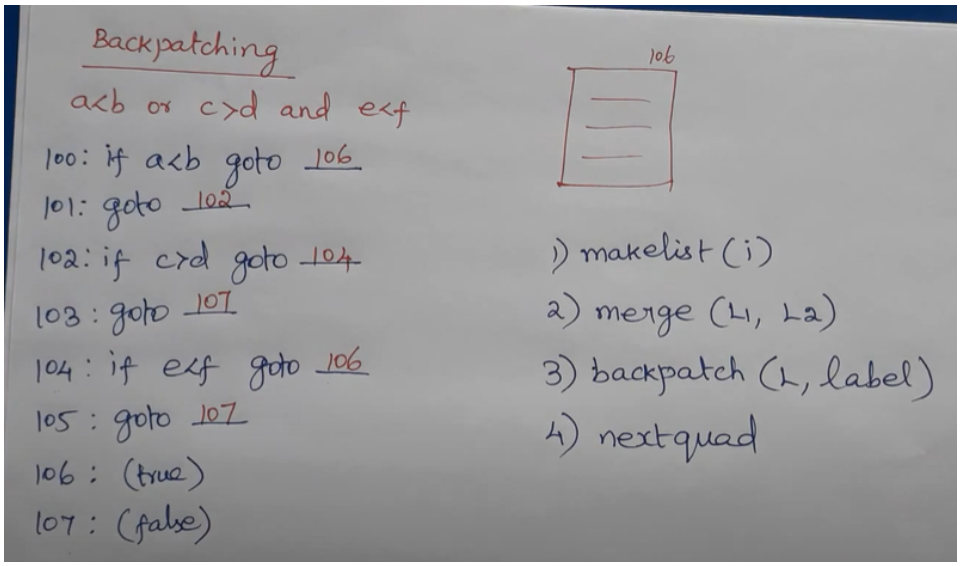
```
    if a < b goto Ltrue
       goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Code

```
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
      goto L1
Lnext:
```

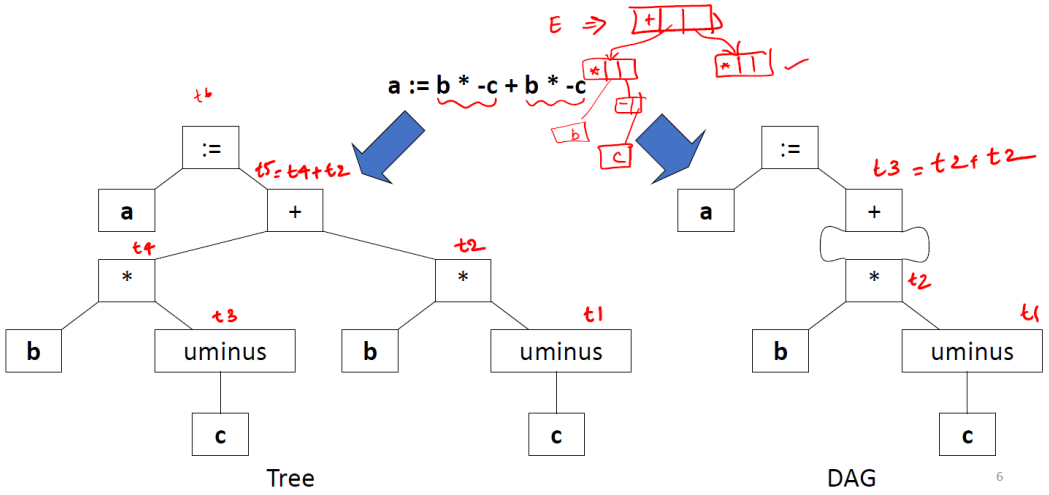
```
while a<b do
    if c<d then
        x:=y+z
    else
        x:=y-z
```

->Backpatching

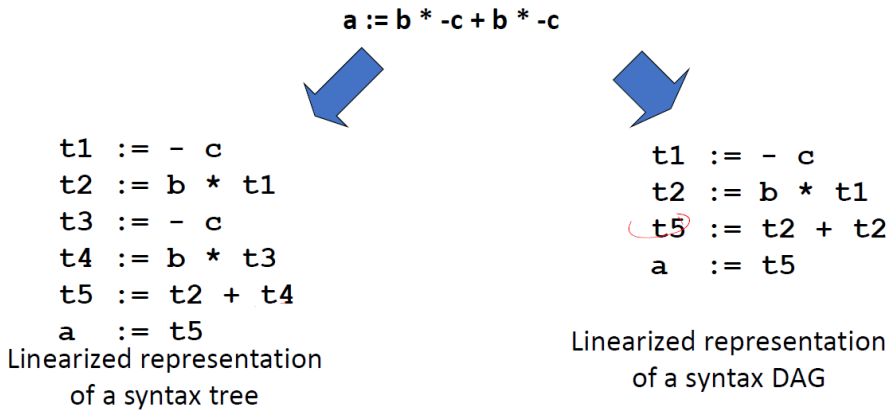


-> Intermediate Code Generation

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)

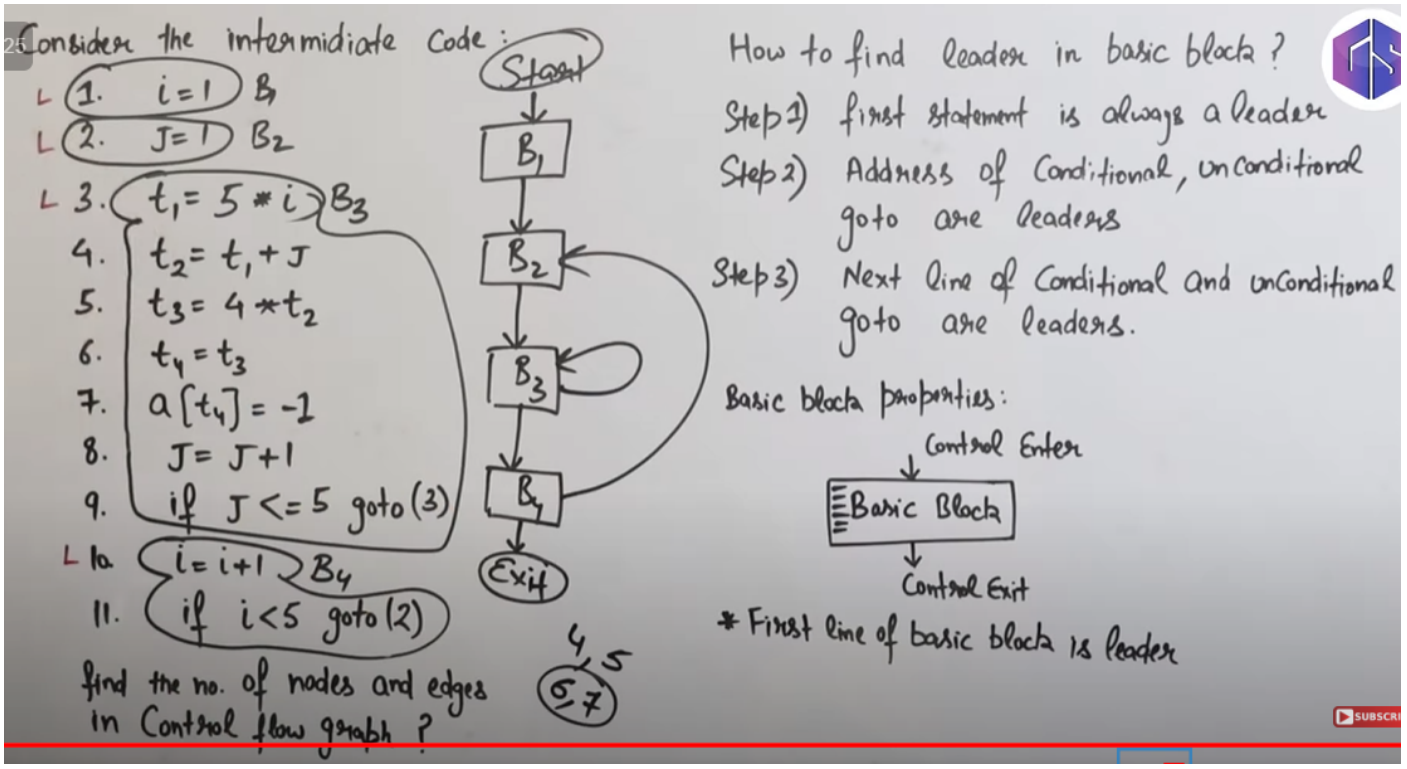
Indirect triples

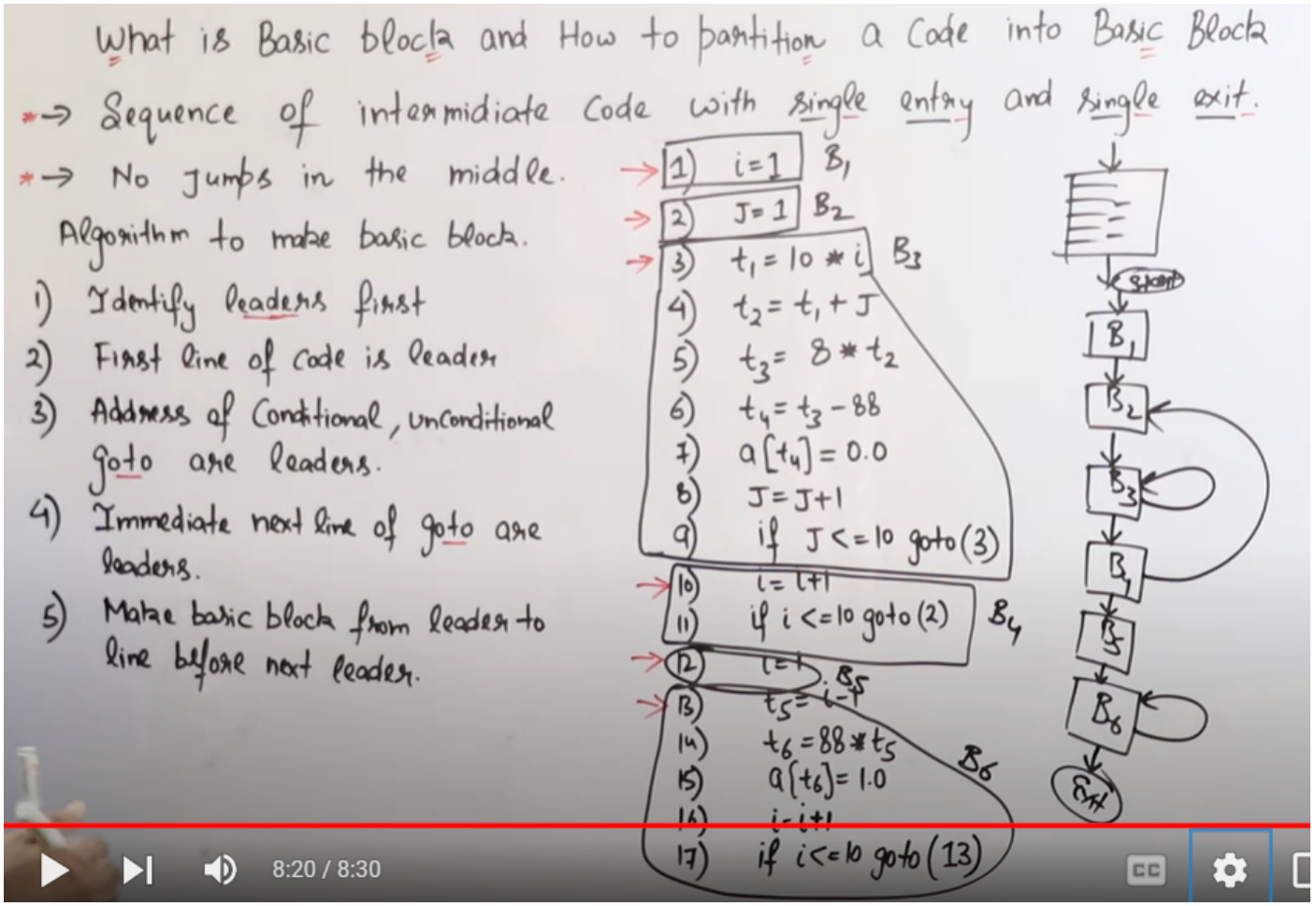
In addition to triples we use a list of pointers to triples

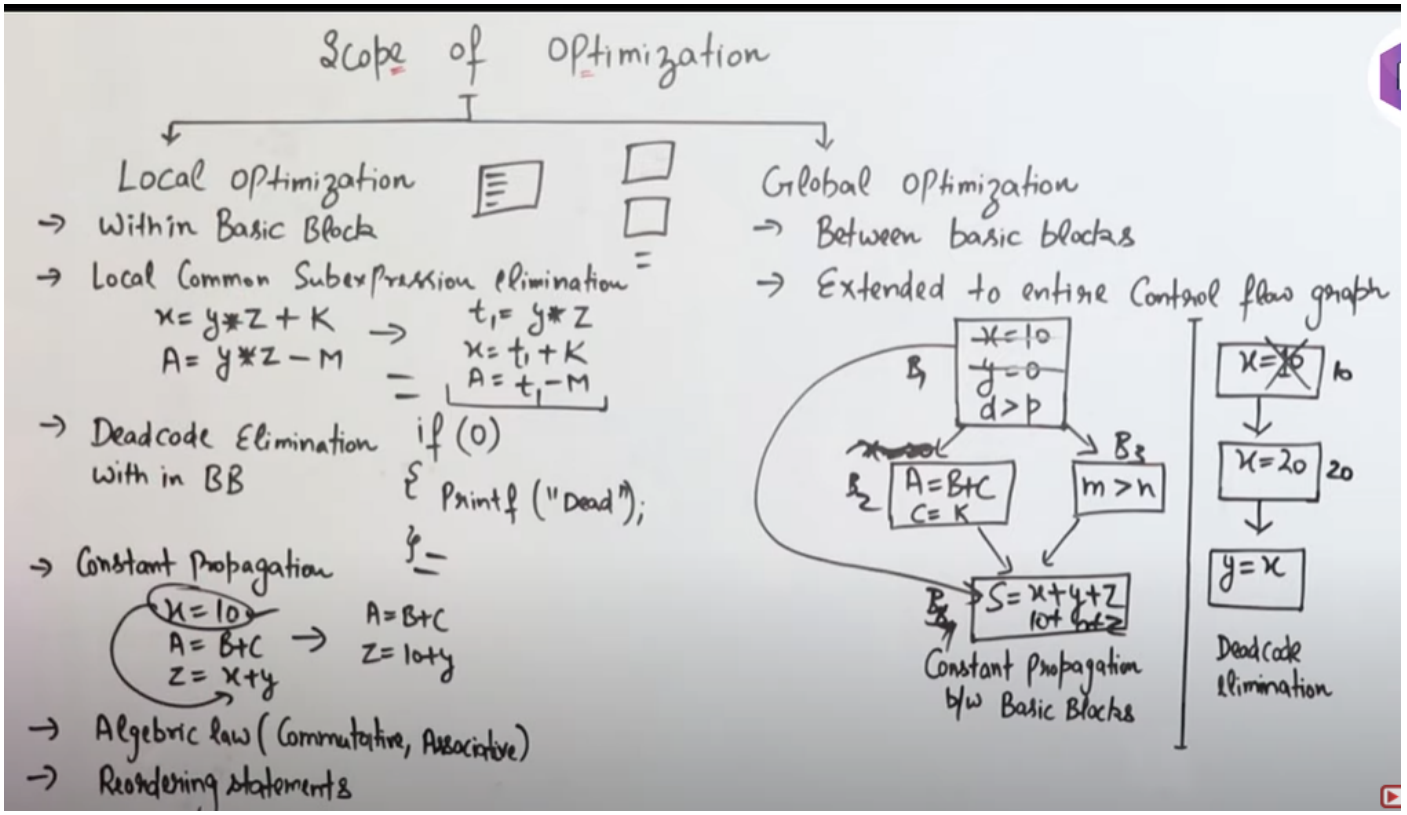
	Statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

	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







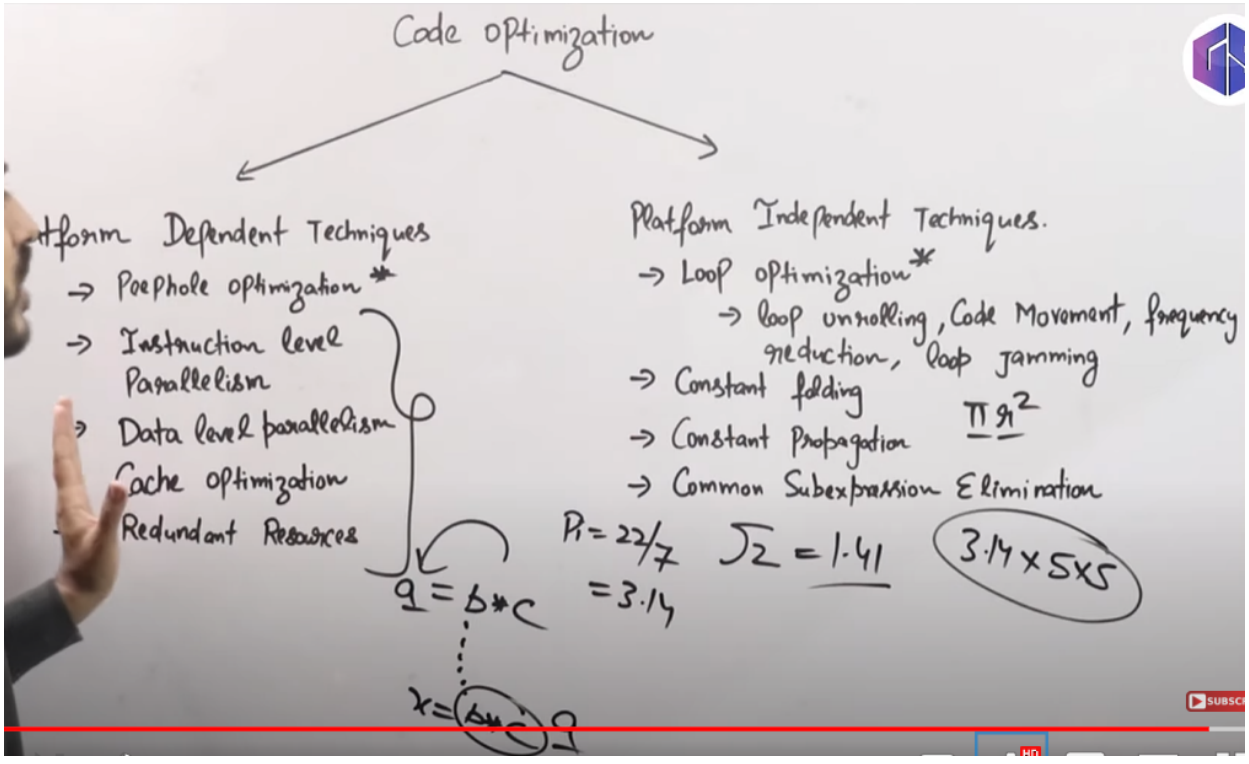
Code Generation Algorithm with examples

DAG

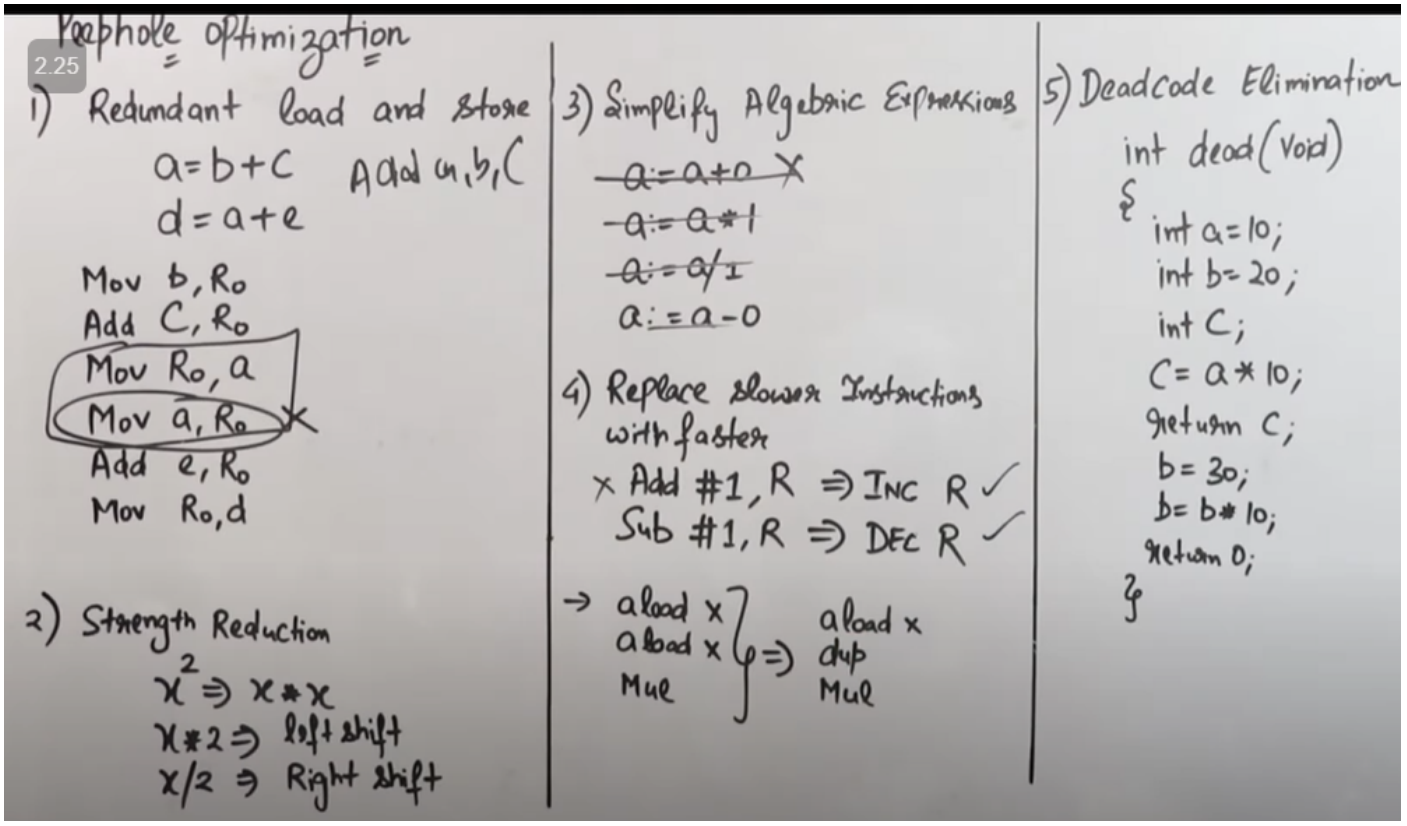
Code generation from DAG

Loops in Flow graphs

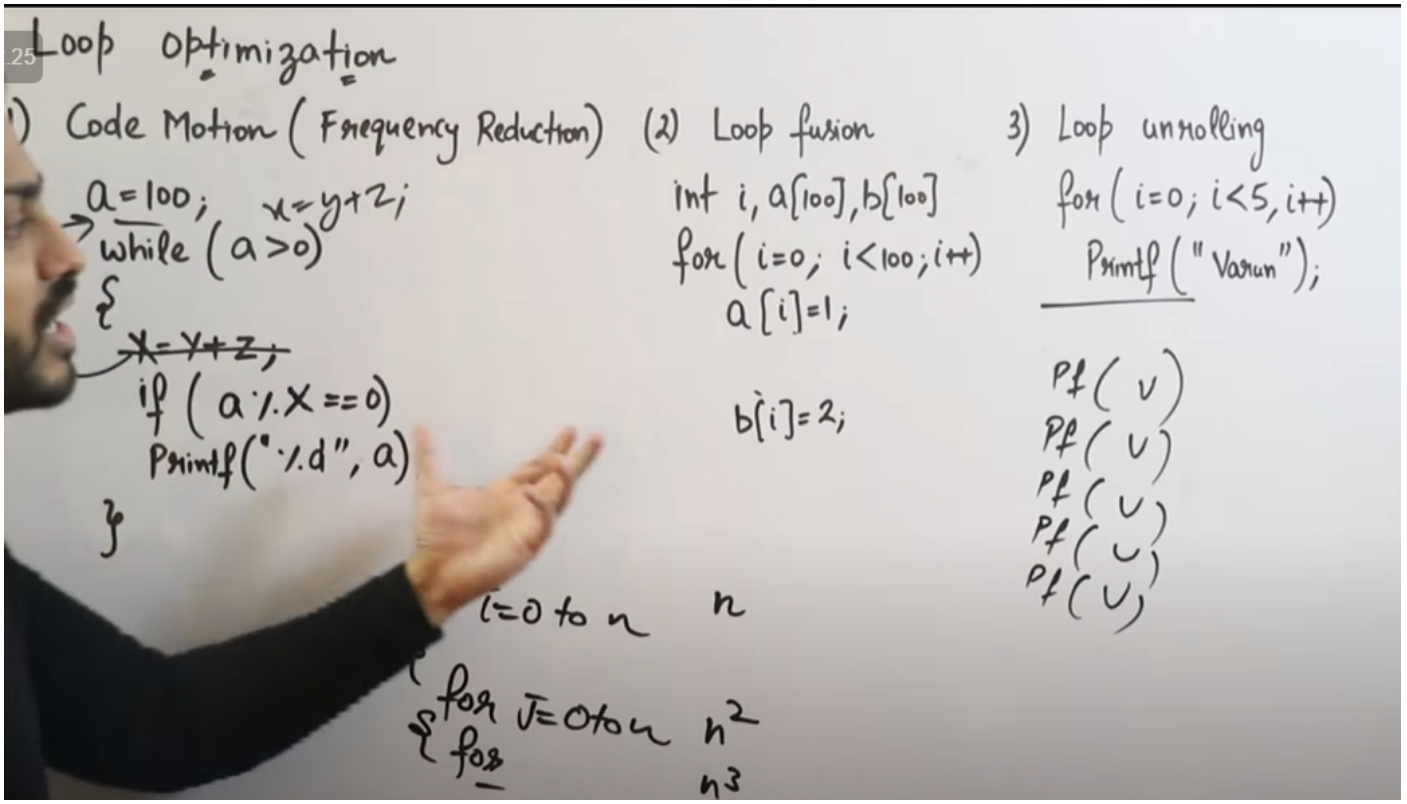
Code Optimization



Peephole Optimizations.



Loop optimization



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