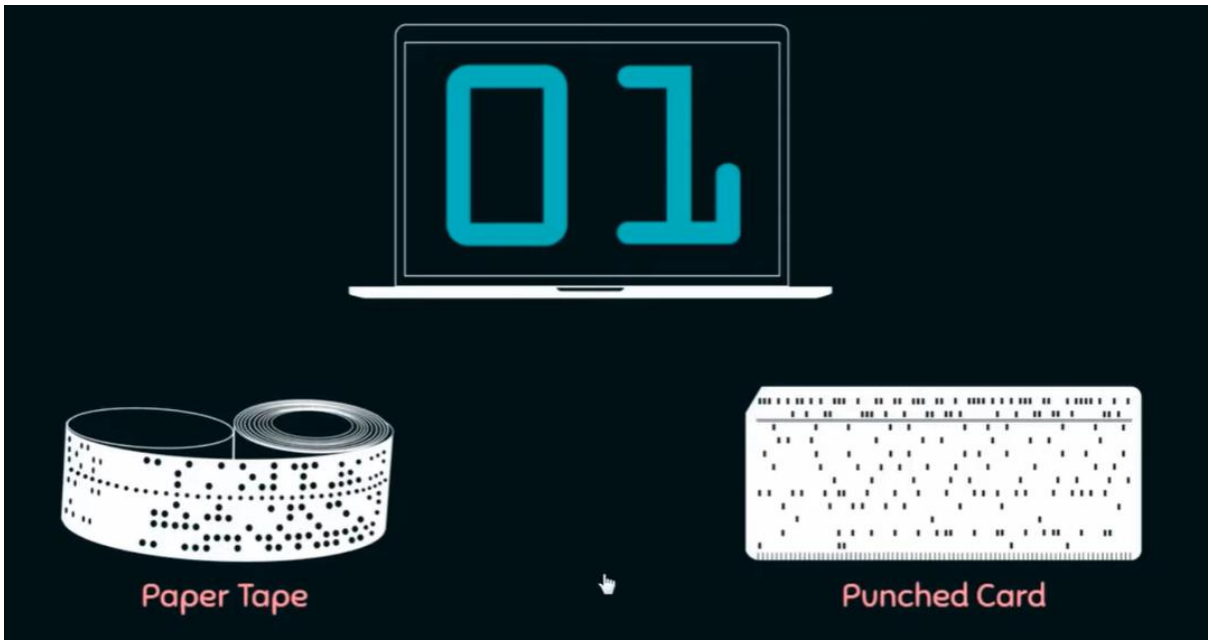
Introduction to Compiler Design



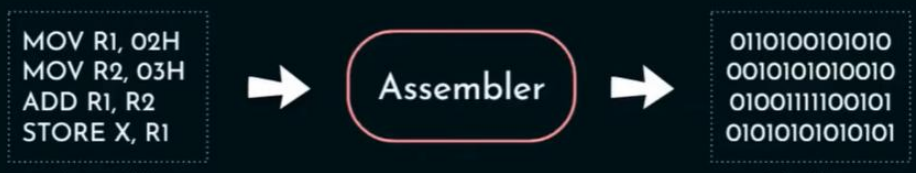
Punched Card:

ASCII

P –	1010000	C –	1000011
u –	1110101	a –	1100001
n –	1101110	r –	1110010
c –	1100011	d –	1110011
h –	1101000		
e –	1100101		
d –	1100100		

Language Translator:

i. Assembler:



Assembly Language

Machine Code

Language Translator:

i. Assembler

ii. Interpreter:

iii. Compiler:

-- Middle level Language

- Direct Memory Access through Pointers
- Bit manipulation using Bitwise Operator
- Writing Assembly Code within C Code

-- High Level Language

```c
#include<stdio.h> //Header file for printf()
int main()       //main function
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Source Code / HLL Code

Language Translator

```
00101010100 10
10110100101010
11010101010010
10011111001010 1
01010101010101
11010101010010
10011111001010 1
```

Machine Code

Language Translator – Internal Architecture

Language Translator – Internal Architecture

```c
#include<stdio.h> //Header file for printf()
int main()       //main function
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Source Code / HLL Code

Preprocessor

stdio.h

```c
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Language Translator – Internal Architecture

stdio.h

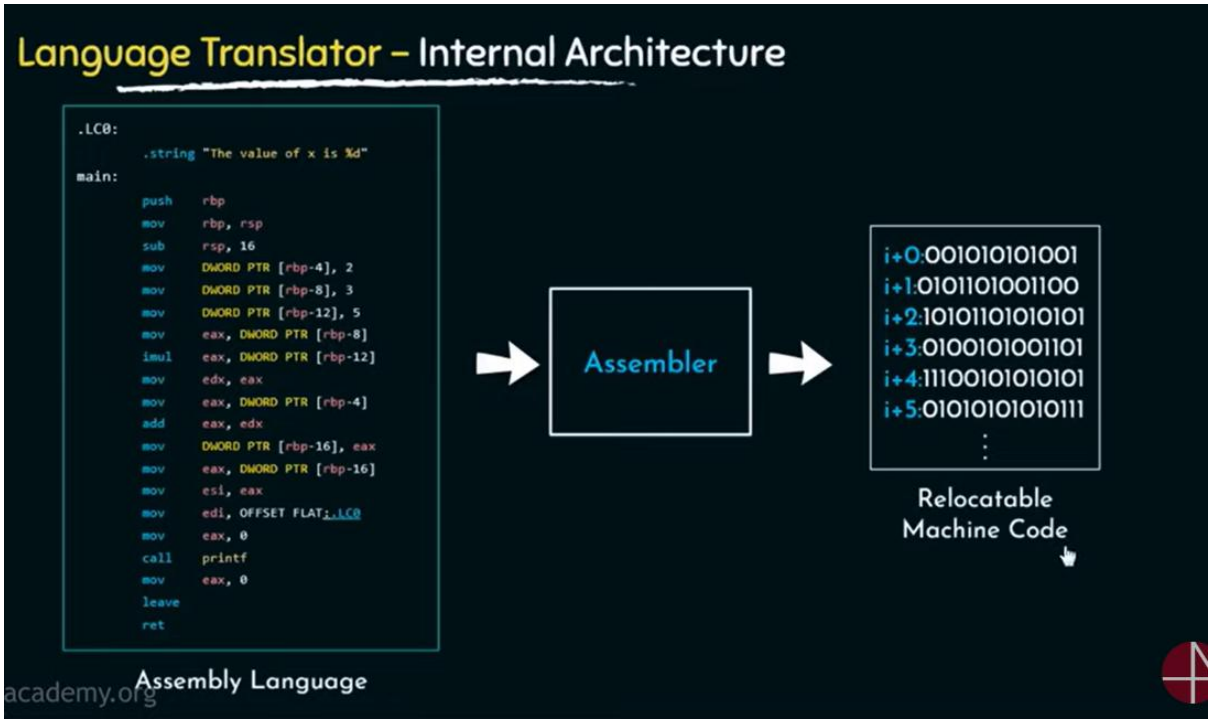
```c
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

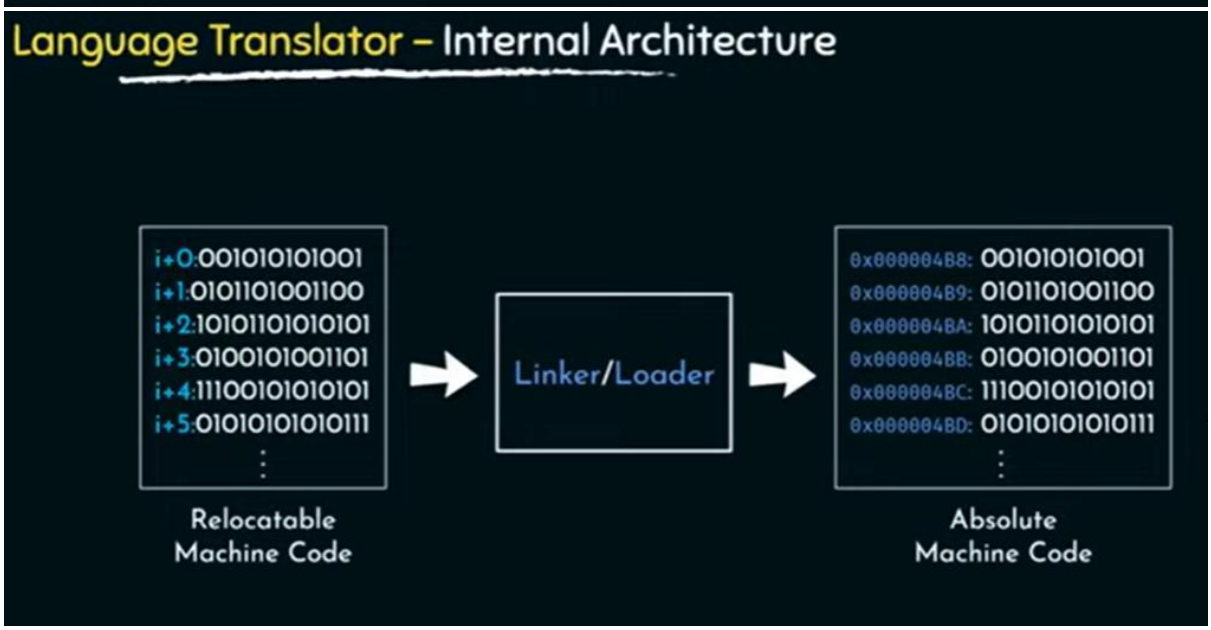
Pure HLL

Compiler

```asm
.LC0:
        .string "The value of x is %d"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 2
        mov     DWORD PTR [rbp-8], 3
        mov     DWORD PTR [rbp-12], 5
        mov     eax, DWORD PTR [rbp-8]
        imul    eax, DWORD PTR [rbp-12]
        mov     edx, eax
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-16]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

Assembly Language


Language Translator – Internal Architecture
.LC0:
.string "The value of x is %d"
main:
push rbp
mov rbp, rsp
sub rsp, 16
mov DWORD PTR [rbp-4], 2
mov DWORD PTR [rbp-8], 3
mov DWORD PTR [rbp-12], 5
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-12]
mov edx, eax
mov eax, DWORD PTR [rbp-4]
add eax, edx
mov DWORD PTR [rbp-16], eax
mov eax, DWORD PTR [rbp-16]
mov esi, eax
mov edi, OFFSET FLAT:.LC0
mov eax, 0
call printf
mov eax, 0
leave
ret
Assembly Language
Assembler
i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
Relocatable
Machine Code
academy.org


Language Translator – Internal Architecture
i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
Relocatable
Machine Code
Linker/Loader
0x000004B8: 001010101001
0x000004B9: 0101101001100
0x000004BA: 10101101010101
0x000004BB: 0100101001101
0x000004BC: 11100101010101
0x000004BD: 01010101010111
Absolute
Machine Code

Compiler – Internal Architecture

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

Analysis Phase

- Intermediate Code Generation
- Code Optimization
- Target Code Generation

Synthesis Phase

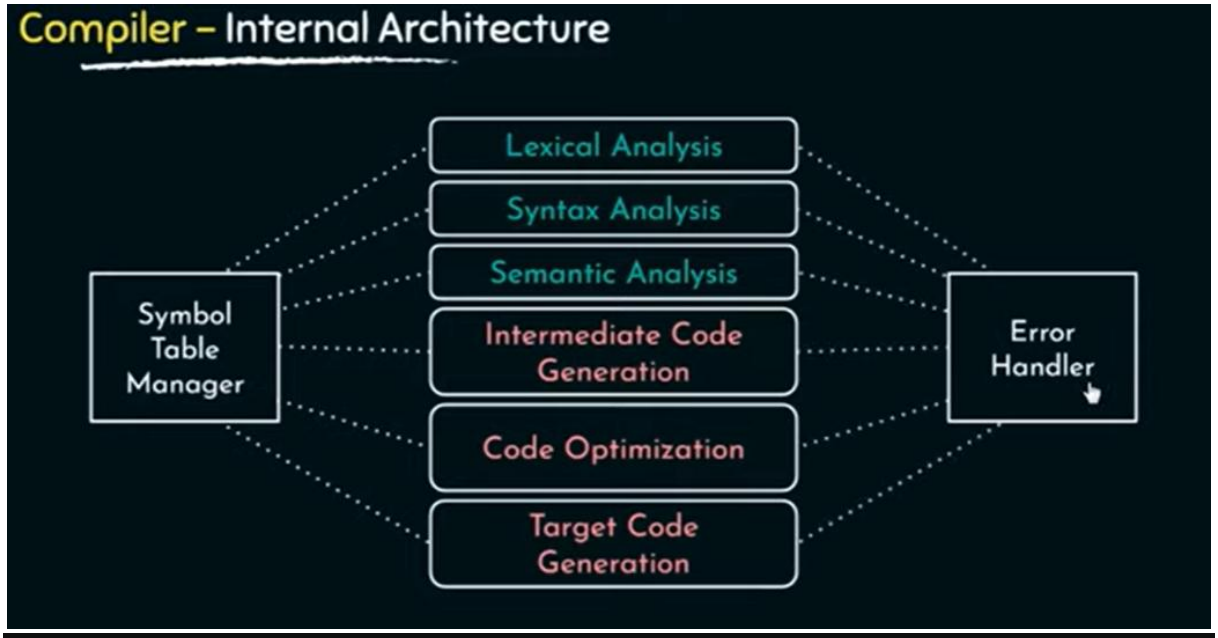
Compiler – Internal Architecture

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation

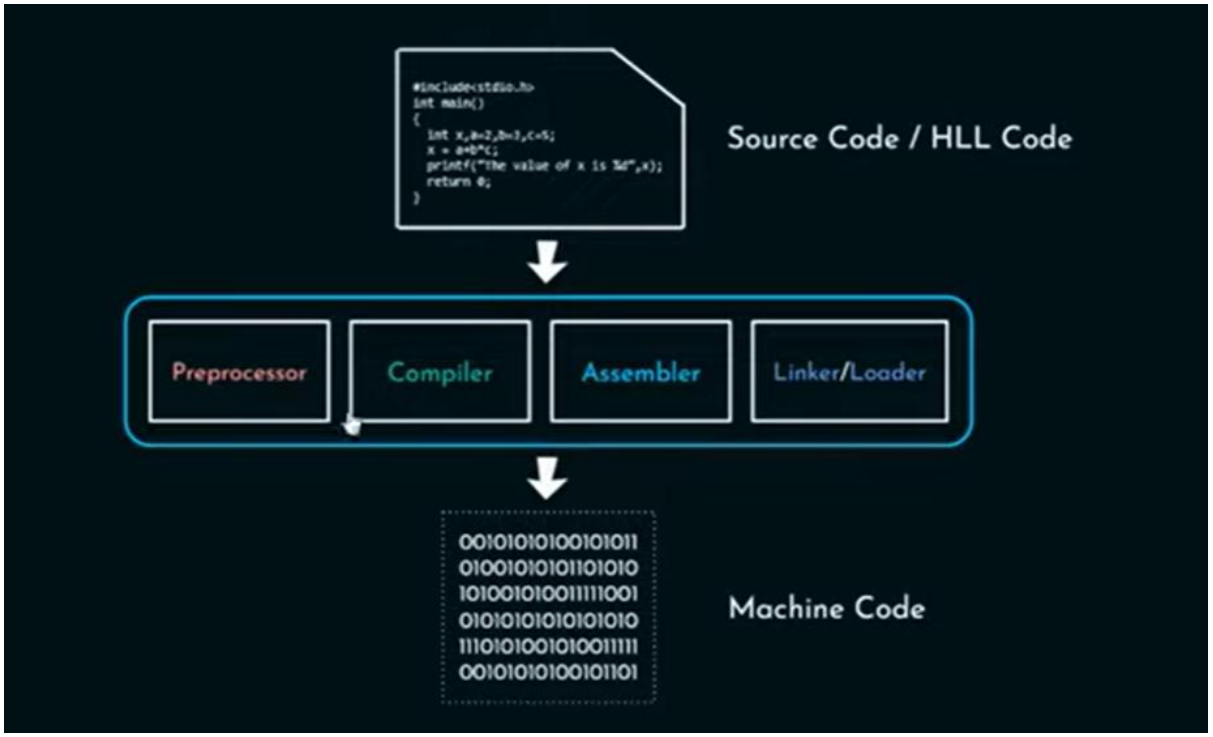
Front-End

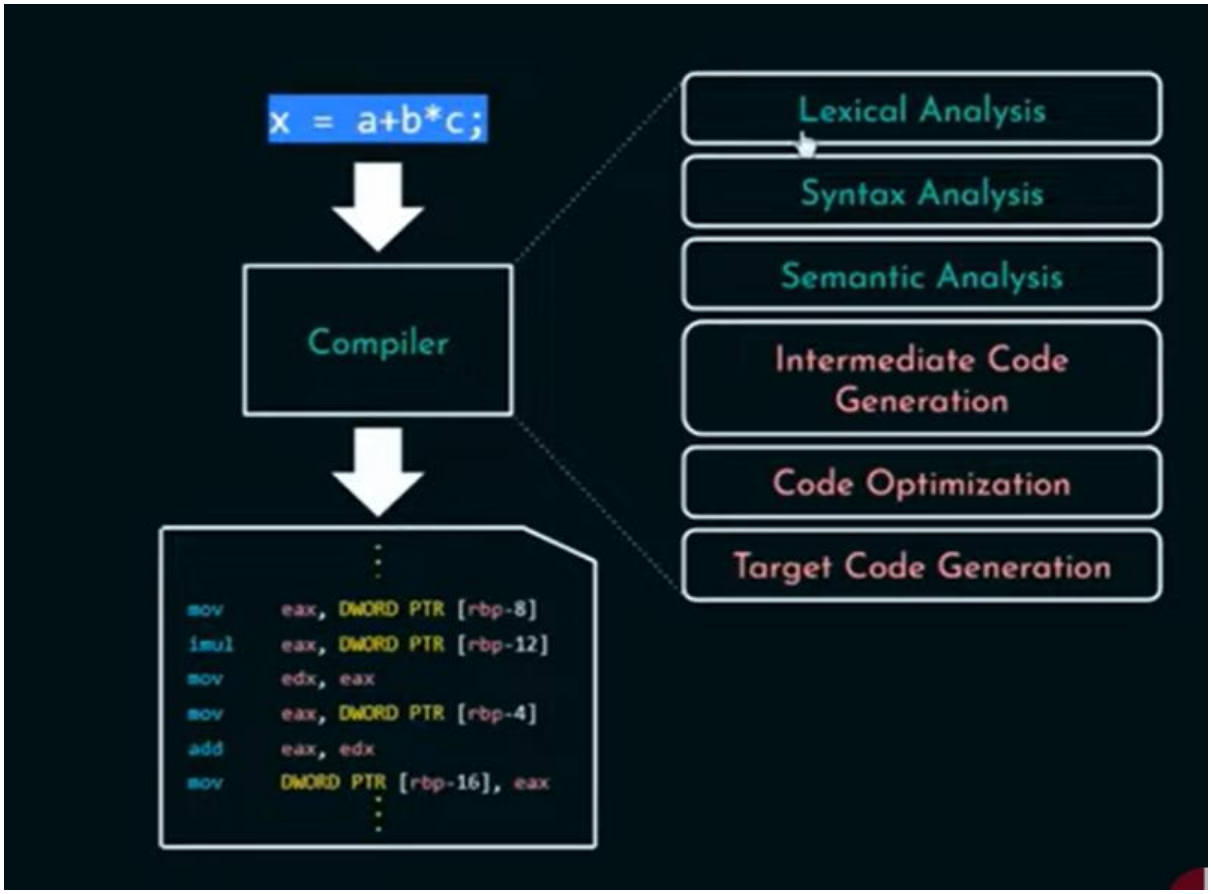
- Code Optimization
- Target Code Generation

Back-End

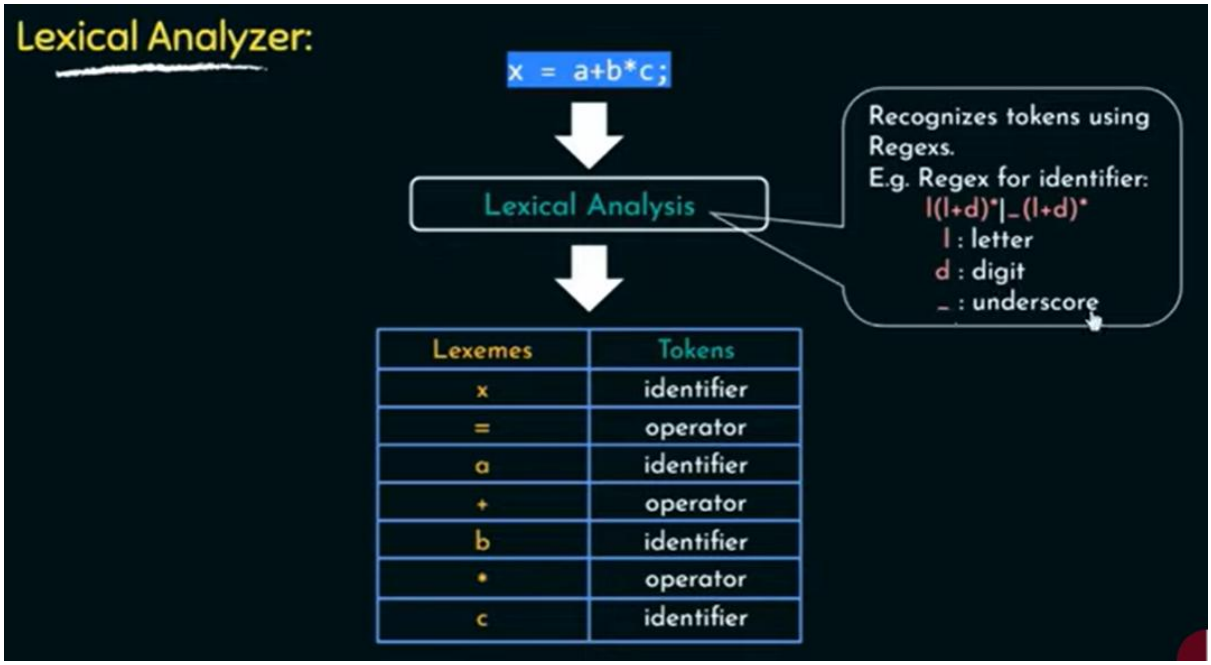


Different Phases of Compiler





Lexical Analyzer:

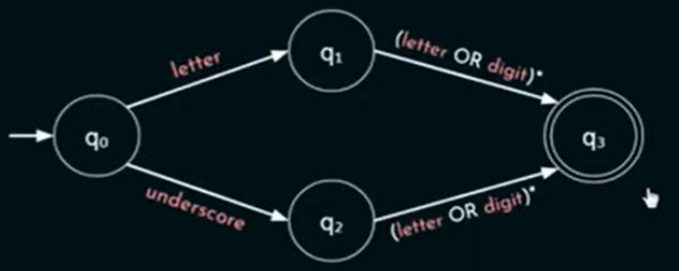


Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

Lexical Analyzer:

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

E.g. Regex for identifier:
l(l+d)*|_(l+d)*
l : letter
d : digit
_ : underscore



Syntax Analyzer:

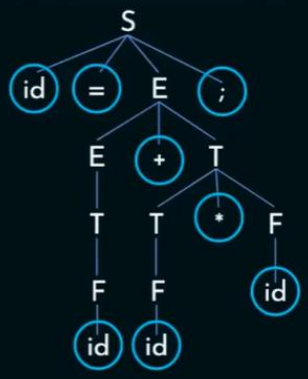
id = id + id * id ;

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

Syntax Analysis

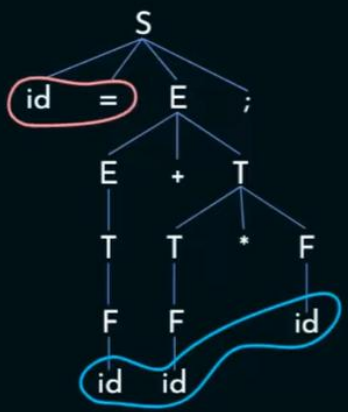
S → id = E ;
E → E + T|T
T → T * F|F
F → id

x = a+b*c;



Parse Tree

Semantic Analyzer:



Parse Tree

Semantic Analysis

Semantically Verified Parse Tree

Intermediate Code Generator:

Semantically Verified Parse Tree → Intermediate Code Generation → Intermediate Code


```
id = id + id * id ;
```

⇅

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

Three Address Code (TAC)

```
1.  t0 = b * c;
2.  t1 = a + t0;
3.  x = t1;
```

Code Optimizer:

Intermediate Code → Code Optimization → Optimized Code

Three Address Code (TAC)

```
1.  t0 = b * c;
2.  t1 = a + t0;
3.  x = t1;
```

Three Address Code (TAC)

```
1.  t0 = b * c;
2.  x = a + t0;
```

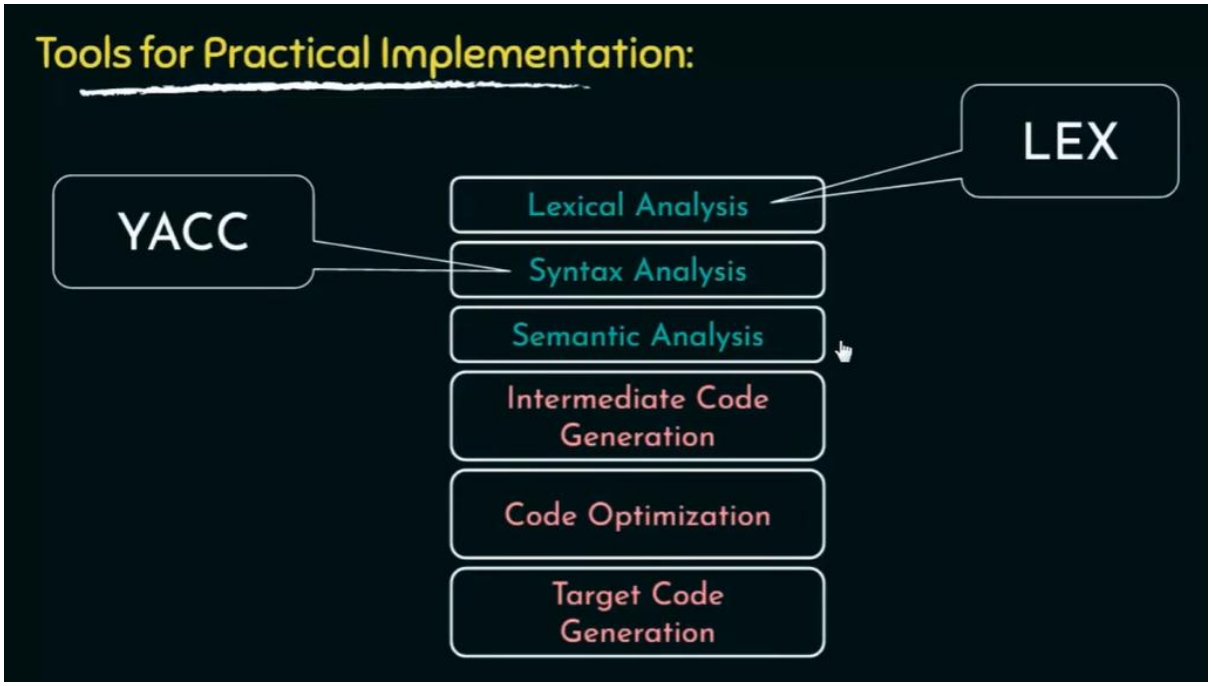
Target Code Generator:

Optimized Code → Target Code Generation →

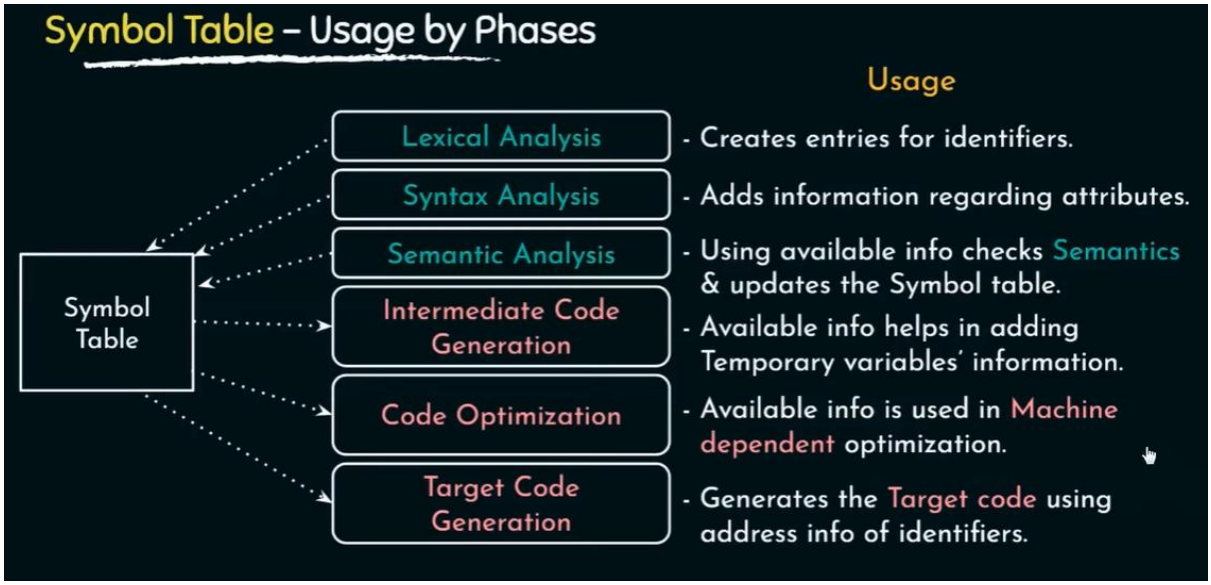
Three Address Code (TAC)

```
1.  t0 = b * c;
2.  x = a + t0;
```

```
:
mov     eax, DWORD PTR [rbp-8]
imul    eax, DWORD PTR [rbp-12]
mov     edx, eax
mov     eax, DWORD PTR [rbp-4]
add     eax, edx
mov     DWORD PTR [rbp-16], eax
:
```



Symbol Table



Symbol Table – Entries

```
int count;
char x[] = "NESO ACADEMY";
```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
count	int	2	0	--	--	--
x	char	12	1	--	--	--

Symbol Table – Operations

- **Non-Block Structured Language:**
  - -- Contains single instance of the variable declaration.
  - -- Operations:
    - i. Insert() ii. Lookup()
- **Block Structured Language:**
  - -- Variable declaration may happen multiple times.
  - -- Operations:
    - i. Insert() ii. Lookup() iii. Set() iv. Reset()

Symbol Table – Solved PYQs

Q1: In the context of compilers, which of the following is/are NOT an intermediate representation of the Source program?

GATE 2021

(A) Three Address Code

(B) Abstract Syntax Tree (AST)

(C) Symbol Table

(D) Control Flow Graph (CFG)

Q2: Access time of the Symbol table will be logarithmic if it is implemented by

ISRO 2016

(A) Linear List

(B) Search Tree

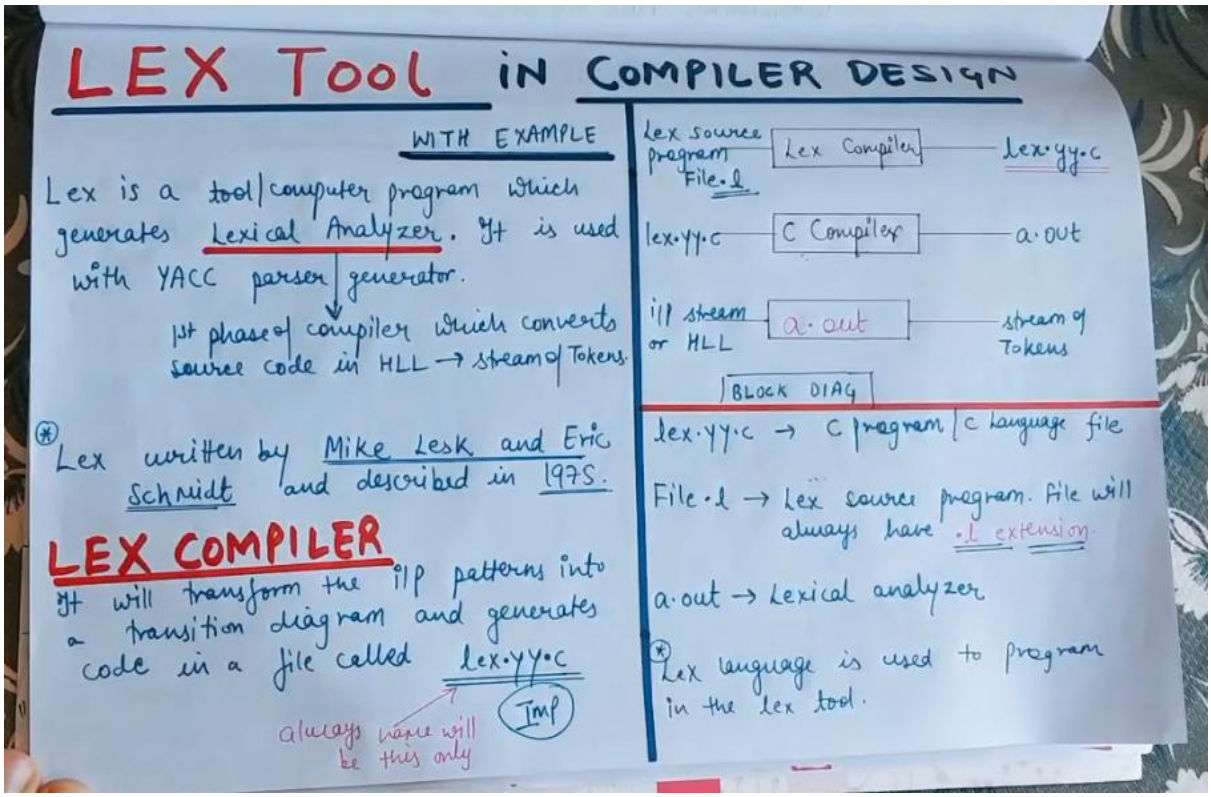
(C) Hash Table

(D) None of these

Symbol Table – Various Implementations:

Implementation	Insertion Time	Lookup Time	Disadvantages
A. Linear Lists			
i. Ordered Lists			i. For Ordered Lists, every insertion is preceded by lookup operation.
a. Arrays	O(n)	O(logn)	
b. Linked Lists	O(n)	O(n)	ii. Access time is directly proportional to table size.
ii. Unordered Lists	O(1)	O(n)	
B. Search Tree	$O(\log_m n)$	$O(\log_m n)$	Always needs to be balanced.
C. Hash Table	O(1)	O(1)	Too many collisions increases the Time complexity to O(n).

LEX tool in Compiler Design



14 | 2020

THURSDAY Wk 20 • 135-231

FUNCTION OF LEX

1) Source code is in Lex language with filename.l extension.
It is given to Lex Compiler which is the Lex tool, and produces lex.yy.c a C program as output.

2) C compiler runs this C code ie lex.yy.c program and produces an output a.out

Lexical Analyzer

3) a.out transforms an i/p stream into a sequence/stream of Tokens.

Be patient while analysing and impatient while executing.

MAY 2020 | 15 FRIDAY

LEX prog

or

LEX file format

A Lex program is separated into 3 sections by %% delimeters.

```
{declarations}   // declaration of variables, including files, libraries
%% %%
{Translation Rules}   -> contains rules in form of Regular exp
%% %%
{Auxiliary functions}
```

eg)

```
%{ #include <stdio.h>
   int c=0;
%}
%%
pattern {action}
%%
main()
{ }
```

pattern {action}

The happiness of your life depends on the wholesomeness of your thoughts.

14 | 2020
THURSDAY
Wk 20 • 135-231

FUNCTION OF LEX

1) Source code is in Lex language with filename.l extension.
It is given to Lex Compiler which is the Lex tool, and produces lex.yy.c — a C program as output.

2) C compiler runs this C code ie lex.yy.c program and produces an output a.out

Lexical Analyzer

3) a.out transforms an i/p stream into a sequence/stream of Tokens.

MAY 2020 | 15
FRIDAY

STRUCTURE OF LEX Prog

or

LEX file format

A Lex program is separated into 3 sections by %% delimiters.

```
{declarations}      // declaration of variables, including file/libraries
%% %%
{Translation Rules}   // contains rules in form of Regular exp
%% %%
{Auxiliary functions}
```

eg)
```
%%{ #include <stdio.h>
    int c=0;
%%}
%%.
pattern {action}   pattern {action}
%%.
main()
{ }
```

depends on the wholesomeness of your thoughts.

abc - match abc

[a-z] - match small a to z

[a-z]* - match all the strings in small letters with null or more than 1 character

[a-z]+ - 1 or more character

[A-Za-z]+

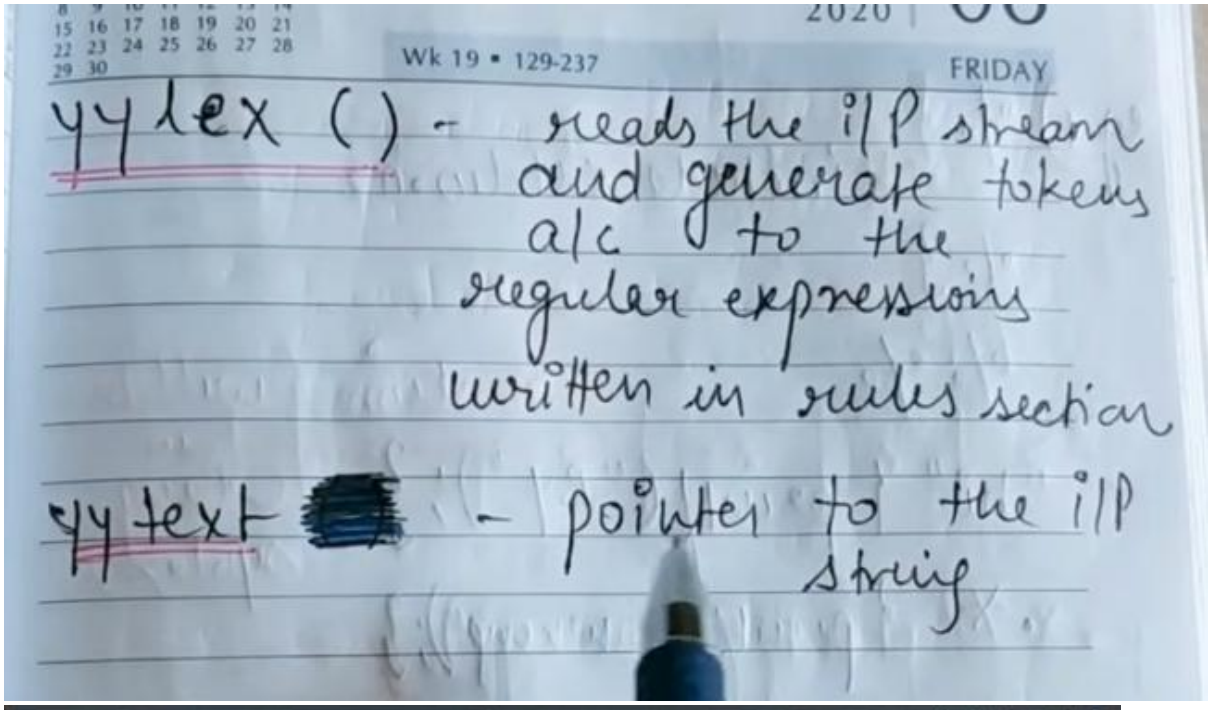
^a means a should come at start

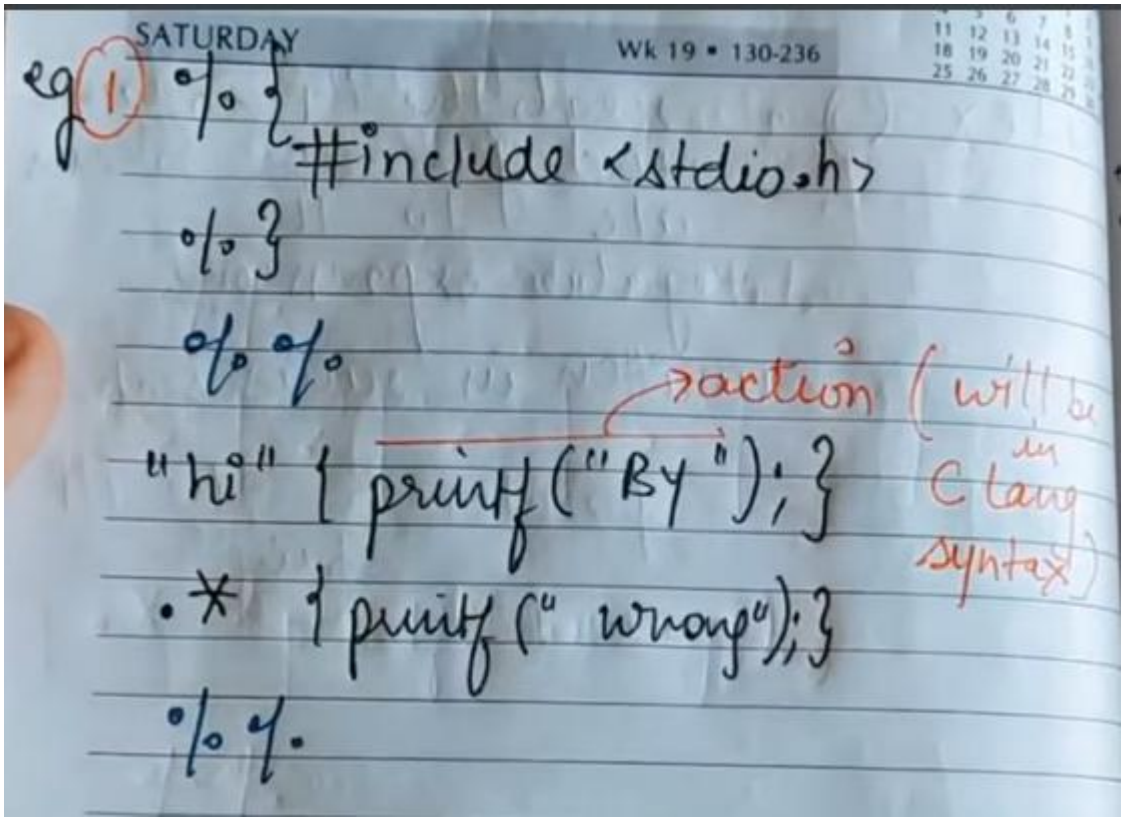
a$ means a at end

[0-9]+ 1 or more digits

yywrap() - called by lex tool when i/p is exhausted

return 1 if i/p is finished else 0!


2020
Wk 19 • 129-237
FRIDAY
yylex() - reads the i/p stream and generate tokens a/c to the regular expressions written in rules section
yytext - pointer to the i/p string.

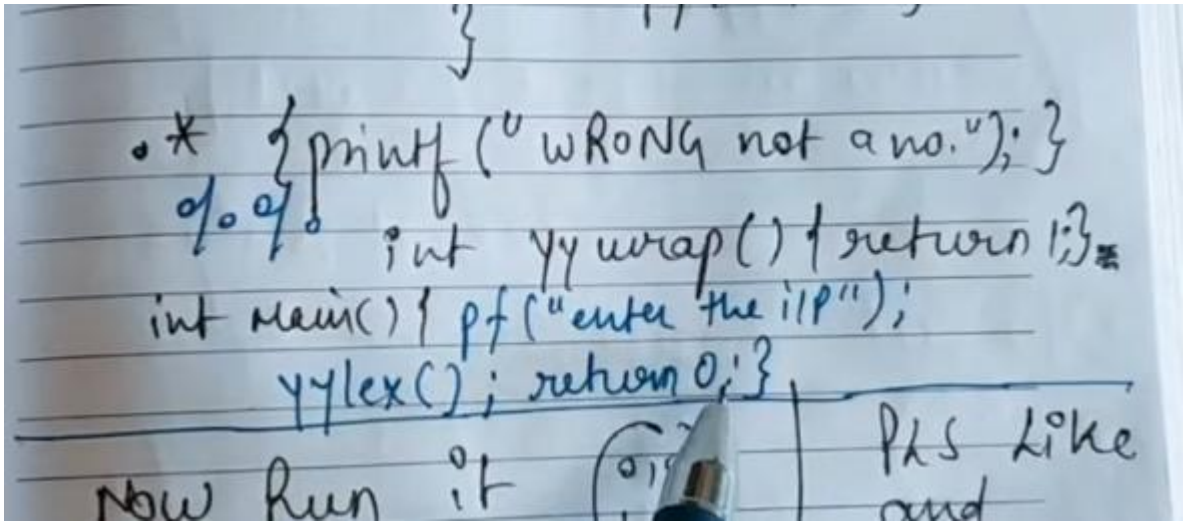

SATURDAY
Wk 19 • 130-236
eg (1)
%{
#include <stdio.h>
%}
%%
"hi" { printf("BY"); } → action (will be in C lang syntax)
.* { printf("wrong"); }
%%

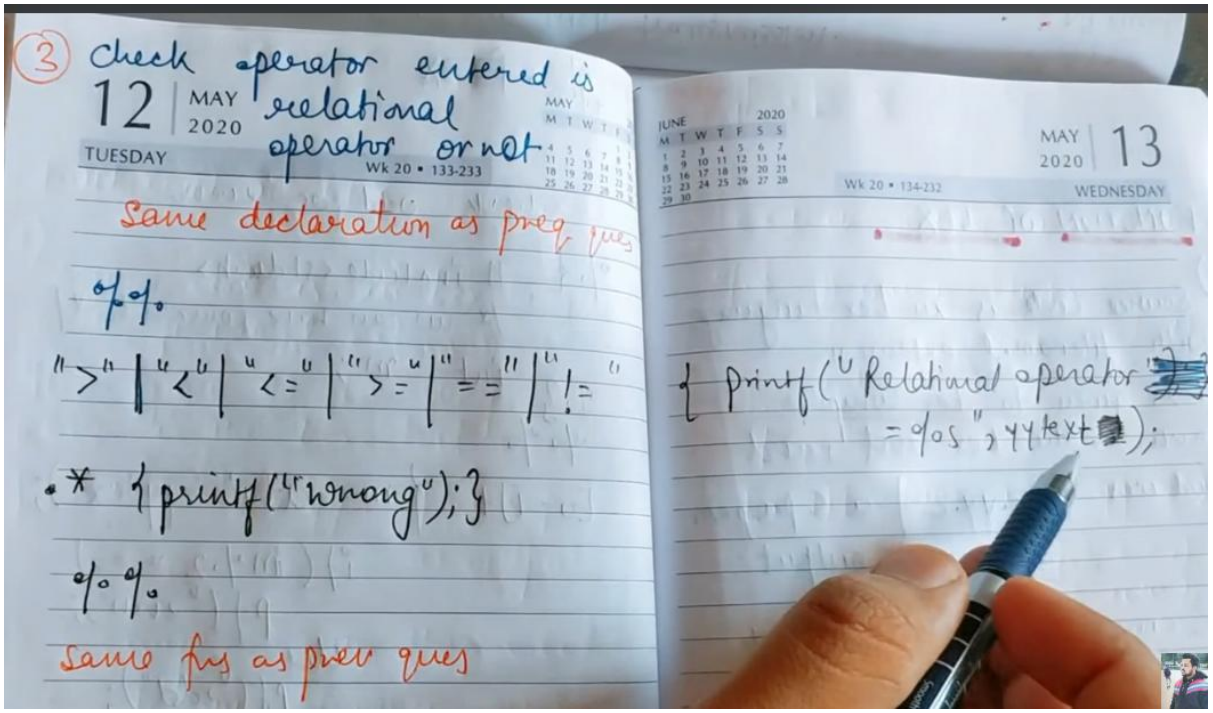
```
main()
{ printf("enter i/p");
  yylex(); // Take I/P & generate the tokens a/c to patterns in rule section
}

int yywrap() // to identify end of i/p
{ return 1; }
```

10 Sunday

~~7~~ (2) check odd or even

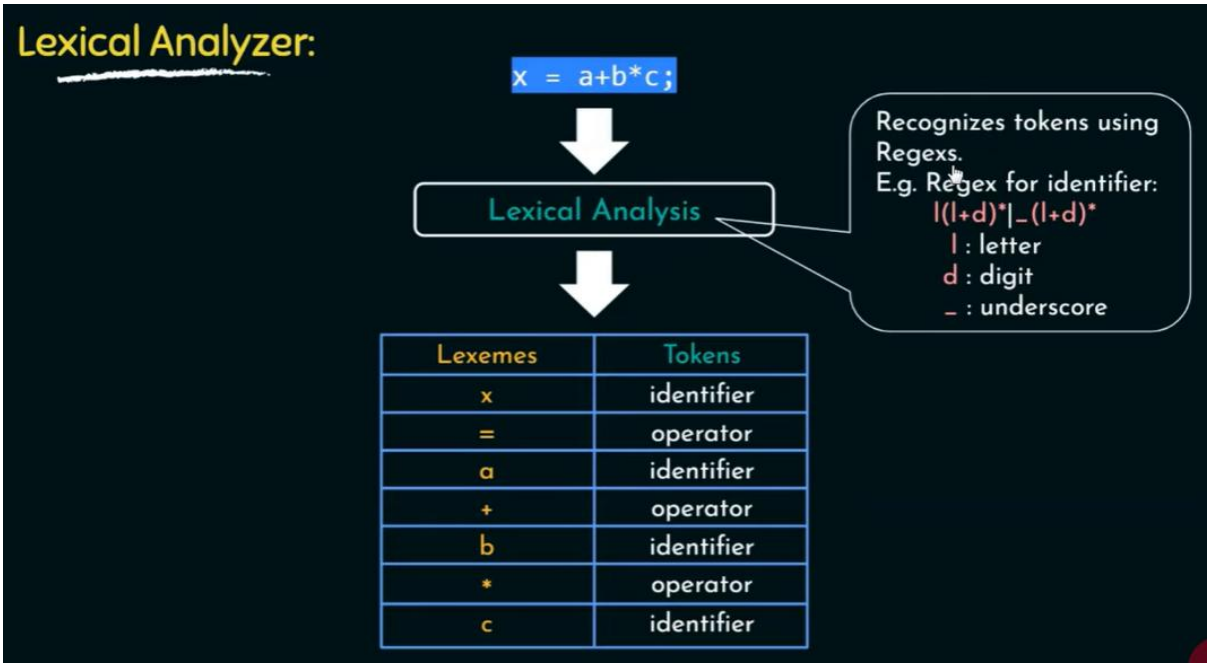
```
%{ #include<stdio.h>
   /* we can have comments */
   int m;
%}
%%
[0-9]+ { m= atoi(yytext);
         if(m%2==0)
           pf("even");
         else
           pf("odd");
       }
```





Introduction to Lexical Analyzer

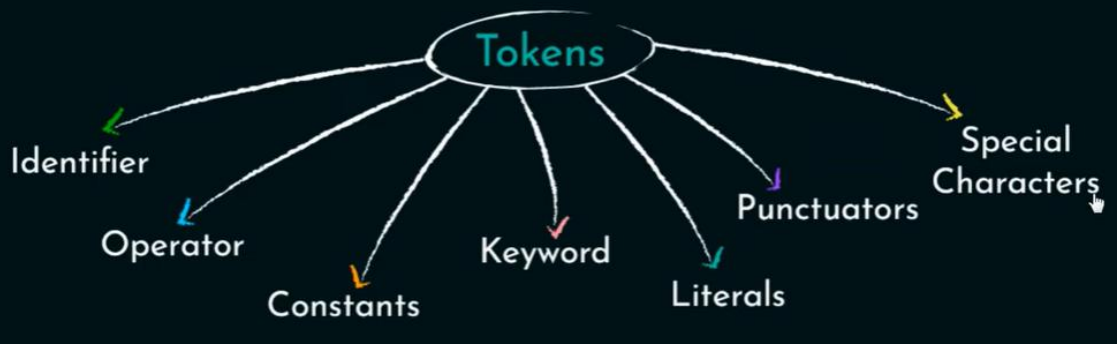
Lexical Analyzer:



Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

Lexical Analyzer:

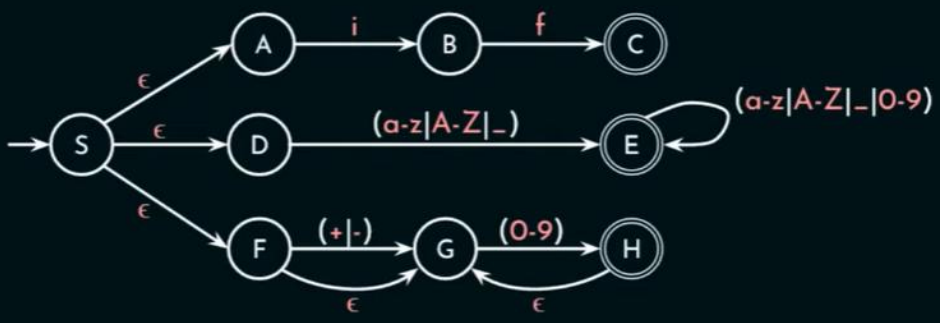
- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.



Lexical Analyzer:

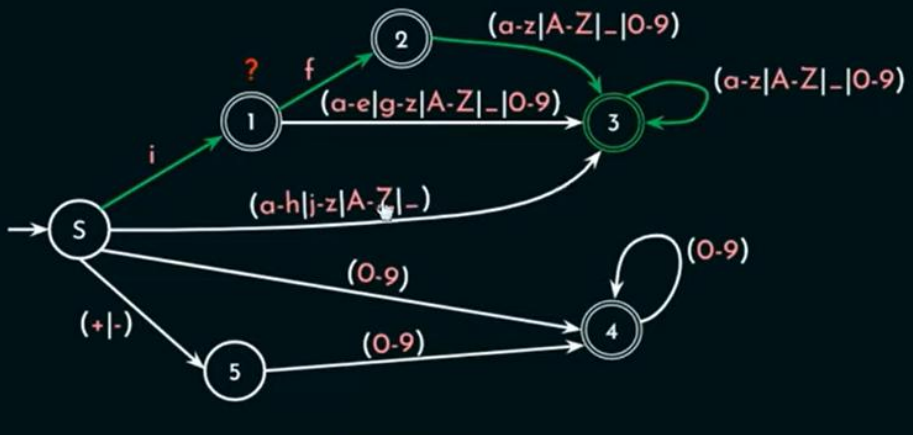
- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.

C-Tokens:



NFA

C-Tokens:



DFA

Lexical Analyzer:

- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.

Uses DFA for pattern matching!

Lexical Analyzer:

- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.
- Removes comments and whitespaces from the Pure HLL code.

```
// Single line comment

/* Multi
   line
   Comment*/
```

```
int NE/*it's a comment*/SO;
```

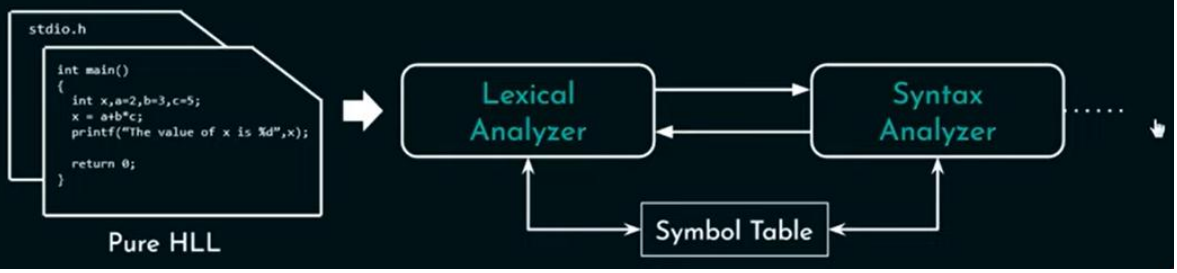
↓

```
int NE SO;
```

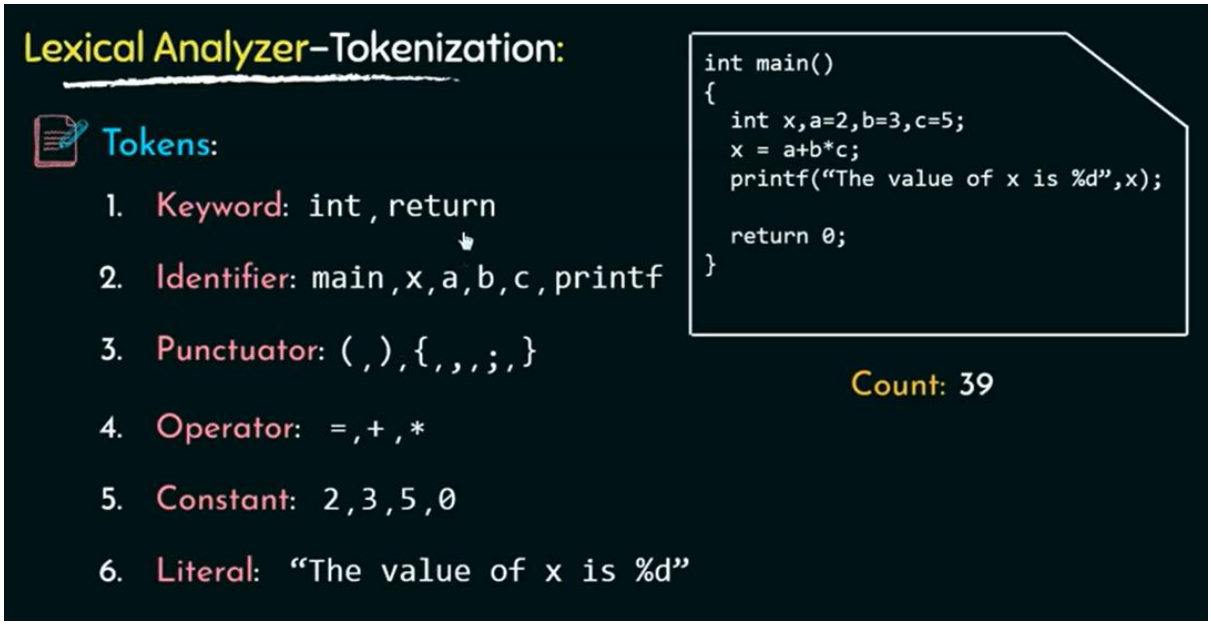
' '	space
'\t'	horizontal tab
'\n'	newline
'\v'	vertical tab
'\f'	form feed
'\r'	carriage return

Lexical Analyzer:

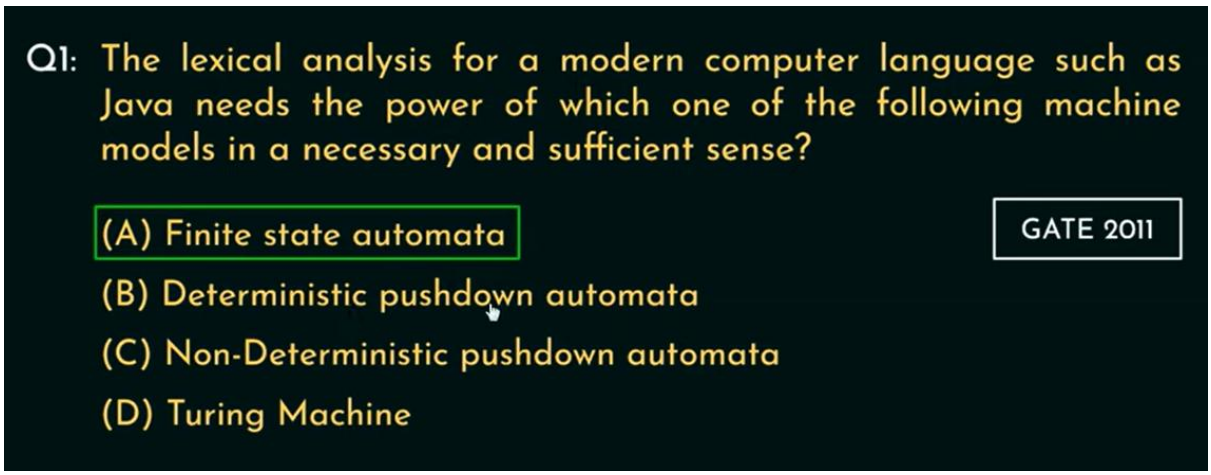
- Scans the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.
- Removes comments and whitespaces from the Pure HLL code.
- Helps in macro expansion in the Pure HLL code.

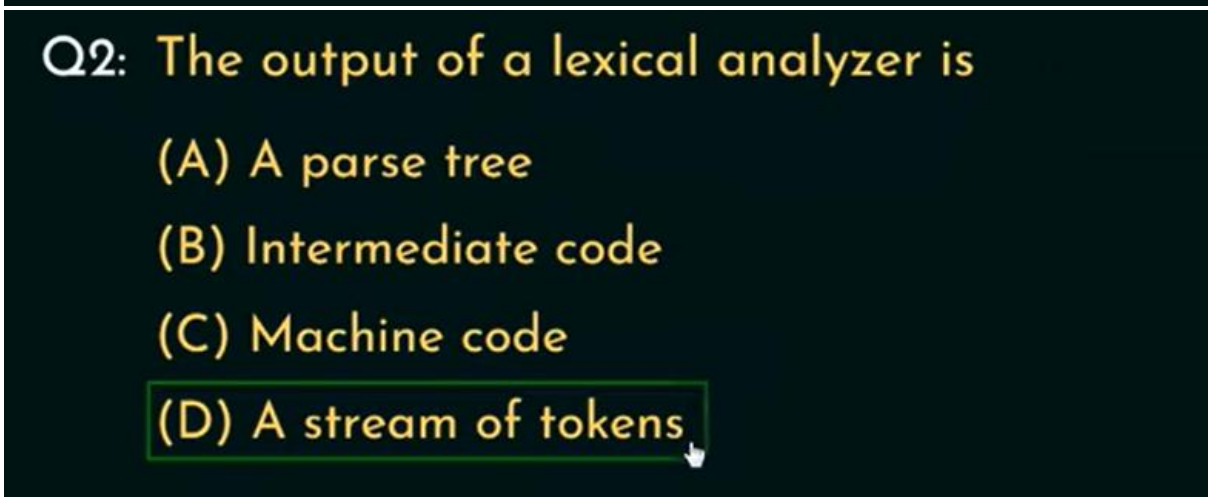


Lexical Analyzer – Tokenization



Lexical Analyzer (Solved Problems) - Set 1





Q3: The number of tokens in the following C statement is

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

GATE 2000

(A) 3

(B) 26

(C) 10

(D) 21

Lexical Analyzer (Solved Problems) - Set 2

Q1: In a compiler, keywords of a language are recognized during

(A) parsing of the program

(B) the code generation

(C) the lexical analysis of the program

(D) dataflow analysis

GATE 2011

NIELIT Scientist-B 2017

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

T_1: $a?\ (b|c)^*\ a$

T_2: $b?\ (a|c)^*\ b$

T_3: $c?\ (b|a)^*\ c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string *bbaacabc* is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) T_3T_3

GATE 2018

Q2: A lexical analyzer uses the following patterns to recognize three tokens T1, T2, and T3 over the alphabet {a,b,c}.

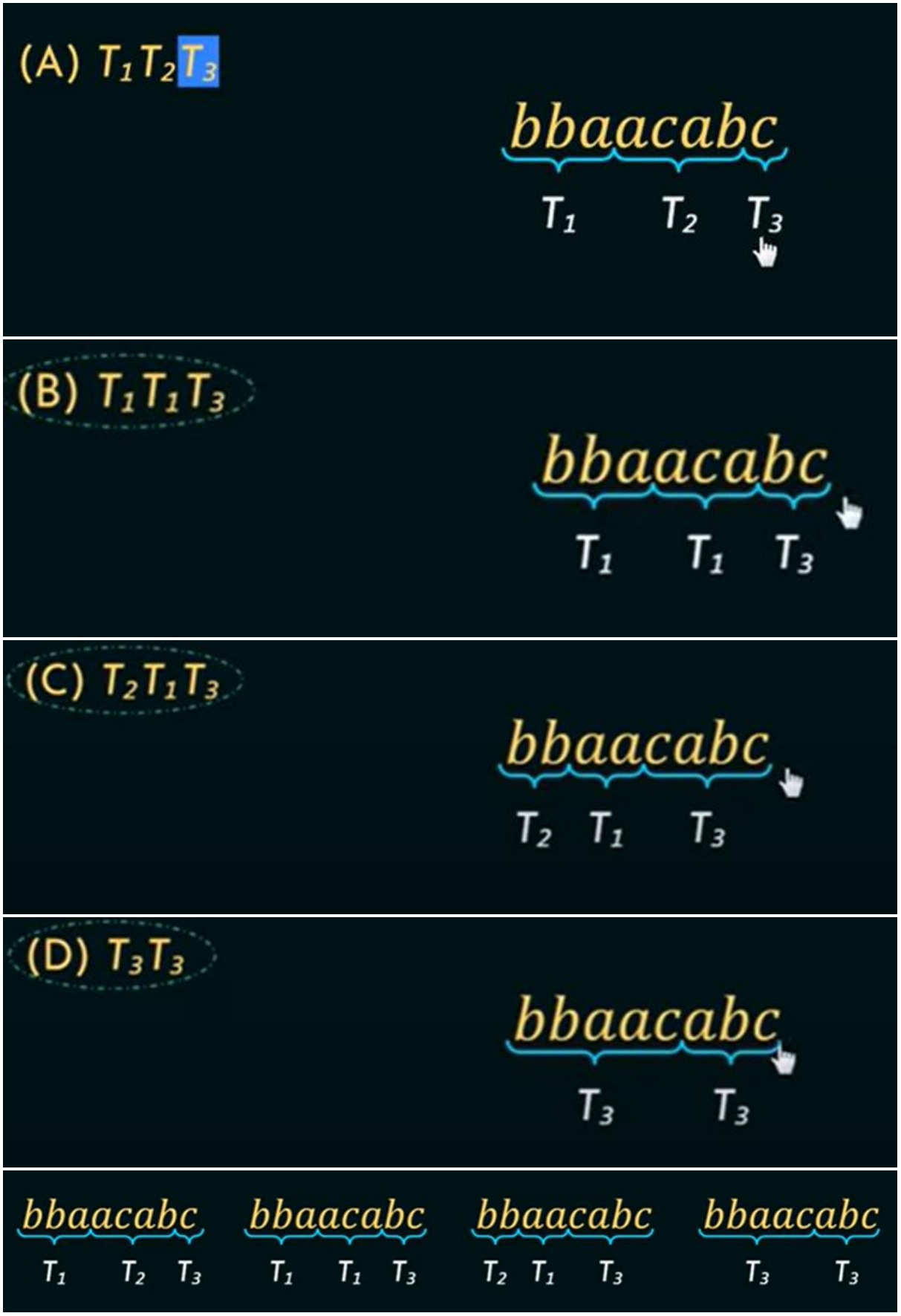
T_1: $a?\ (b|c)^*\ a \longrightarrow (b|c)^*a + a(b|c)^*a$

T_2: $b?\ (a|c)^*\ b \longrightarrow (a|c)^*b + b(a|c)^*b$

T_3: $c?\ (b|a)^*\ c \longrightarrow (b|a)^*c + c(b|a)^*c$

If the string *bbaacabc* is processed by the analyzer, which one of the following is the sequence of tokens it outputs?

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) T_3T_3


(A) $T_1T_2T_3$
bbaacabc
T_1 T_2 T_3
(B) $T_1T_1T_3$
bbaacabc
T_1 T_1 T_3
(C) $T_2T_1T_3$
bbaacabc
T_2 T_1 T_3
(D) T_3T_3
bbaacabc
T_3 T_3
bbaacabc T_1 T_2 T_3
bbaacabc T_1 T_1 T_3
bbaacabc T_2 T_1 T_3
bbaacabc T_3 T_3

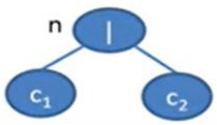
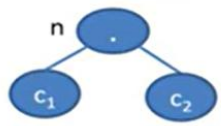
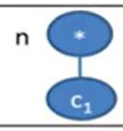
(A) $T_1T_2T_3$ (B) $T_1T_1T_3$ (C) $T_2T_1T_3$ (D) T_3T_3

Regular Expression to DFA Pattern matching, FIRSTPOS, LASTPOS and FOLLOWPOS

❑ Function computed from the syntax tree

- *nullable(n)*
  - The subtree at node n generates languages including the empty string.
- *firstpos(n)*
  - The set of positions that can match the first symbol of a string generated by the subtree at node n.
- *lastpos(n)*
  - The set of positions that can match the last symbol of a string generated be the subtree at node n.
- *followpos(i)*
  - The set of positions that can follow position i in the tree.

❑ Rules to compute nullable, firstpos, lastpos

Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
A leaf labeled by ε	**true**	∅	∅
A leaf with position ***i***	**false**	$\{i\}$	$\{i\}$
	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
	*nullable*(c_1) **and** *nullable*(c_2)	**if** (*nullable*(c_1)) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** (*nullable*(c_2)) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
	***true***	*firstpos*(c_1)	*lastpos*(c_1)

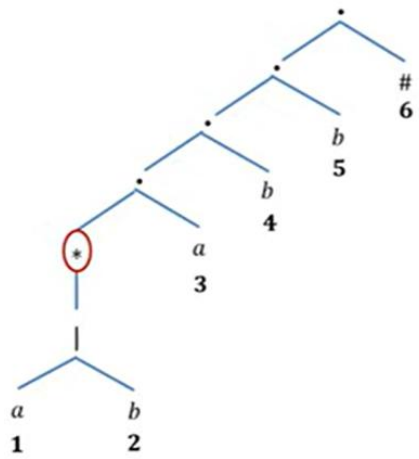
❑ Rules to compute followpos

1. If n is **concatenation** node with left child c1 and right child c2 and ***i*** is a position in lastpos(c1), then all position in firstpos(c2) are in followpos(i)

2. If n is * node and ***i*** is position in lastpos(n), then all position in firstpos(n) are in followpos(i)

❑ Conversion from regular expression to DFA without constructing NFA

R.E: (a|b)*abb #



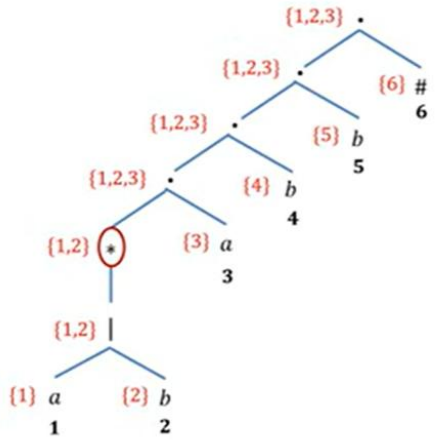
Step 1: Construct Syntax Tree

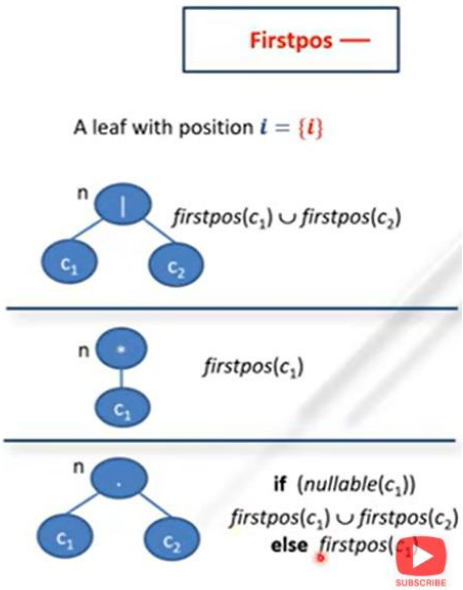
Step 2: Nullable node

Here, * is only nullable node

❑ Conversion from regular expression to DFA without constructing NFA

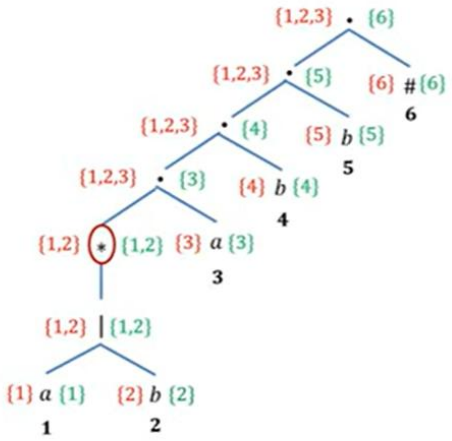
Step 3: Calculate firstpos





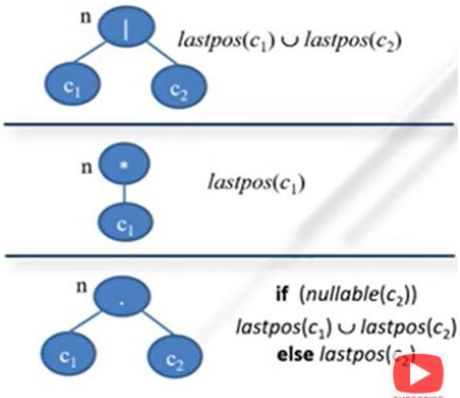
❑ Conversion from regular expression to DFA without constructing NFA

Step 3: Calculate lastpos



Lastpos —

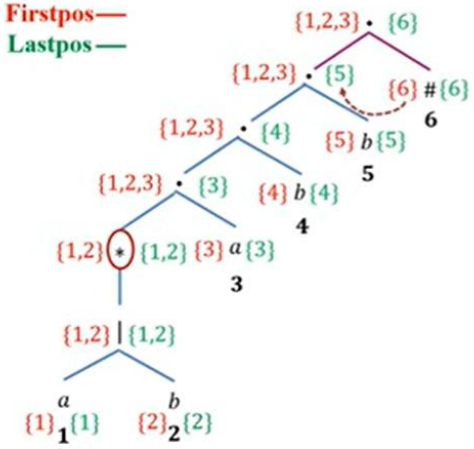
A leaf with position $i = \{i\}$



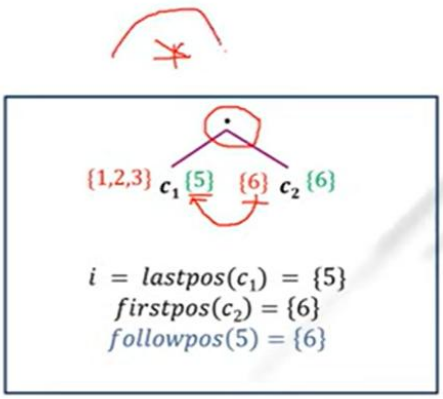
❑ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos

Firstpos—
Lastpos—

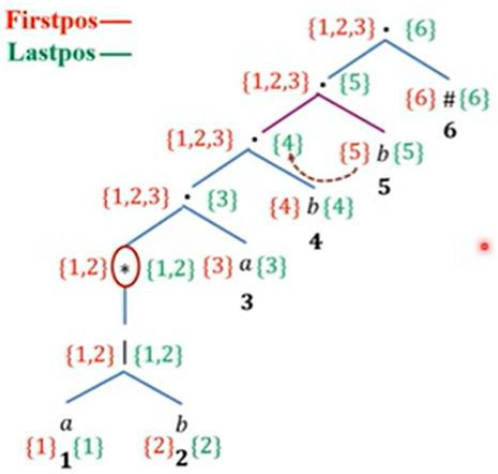


Position	followpos
5	6



❑ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos

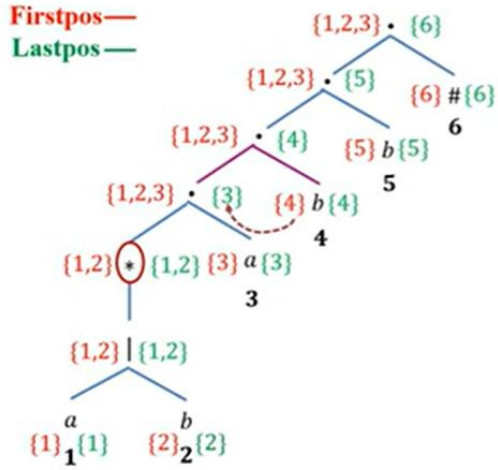


Position	followpos
5	6
4	5

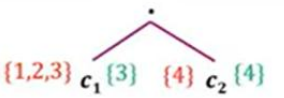
$i = lastpos(c_1) = \{4\}$
$firstpos(c_2) = \{5\}$
$followpos(4) = \{5\}$

❑ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos



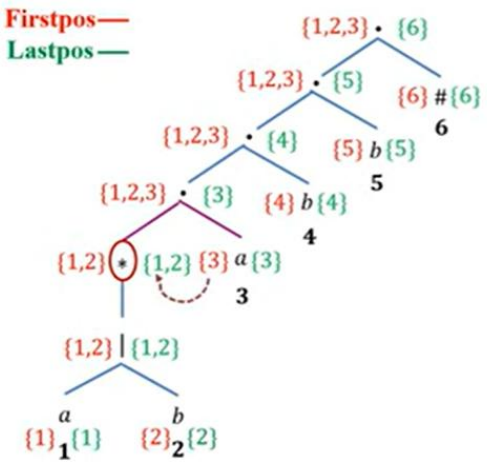
Position	followpos
5	6
4	5



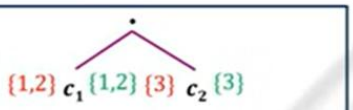
$i = lastpos(c_1) = \{3\}$
$firstpos(c_2) = \{4\}$
$followpos(3) = \{4\}$

❑ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos



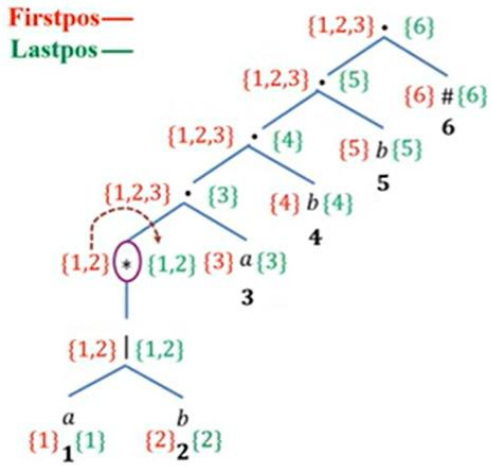
Position	followpos
5	6
4	5
3	4



$i = lastpos(c_1) = \{1,2\}$
$firstpos(c_2) = \{3\}$
$followpos(1) = \{3\}$
$followpos(2) = \{3\}$

❑ Conversion from regular expression to DFA without constructing NFA

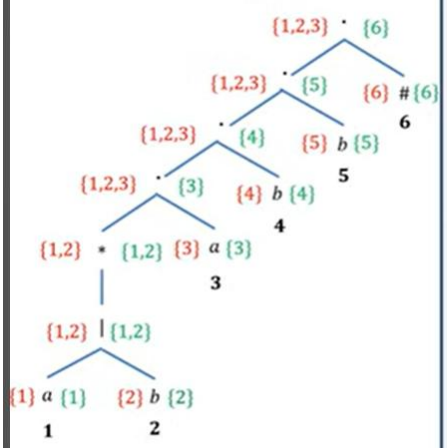
Step 4: Calculate followpos



Position	followpos
6	--
5	6
4	5
3	4
2	1,2,3
1	1,2,3

$i = lastpos(n) = \{1,2\}$
$firstpos(n) = \{1,2\}$
$followpos(1) = \{1,2\}$
$followpos(2) = \{1,2\}$

❑ Constructing DFA



Initial state = *firstpos* of root = {1,2,3} ----- A

State A

δ((A,a) = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

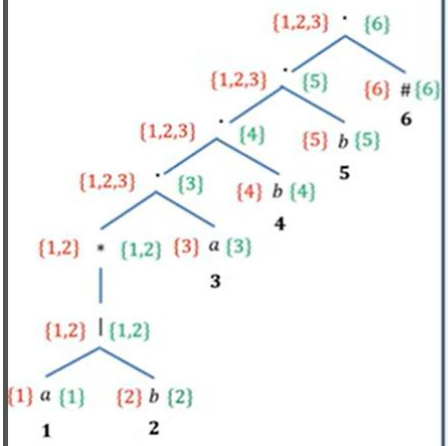
δ((A,b) = followpos(2)

=(1,2,3) ----- A

Position	followpos
6	--
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}		

❑ Constructing DFA



State B

δ((B,a) = followpos(1) U followpos(3)

=(1,2,3) U (4) = {1,2,3,4} ----- B

δ((B,b) = followpos(2) U followpos(4)

=(1,2,3) U (5) = {1,2,3,5} ----- C

State C

δ((C,a) = followpos(1) U followpos(3)

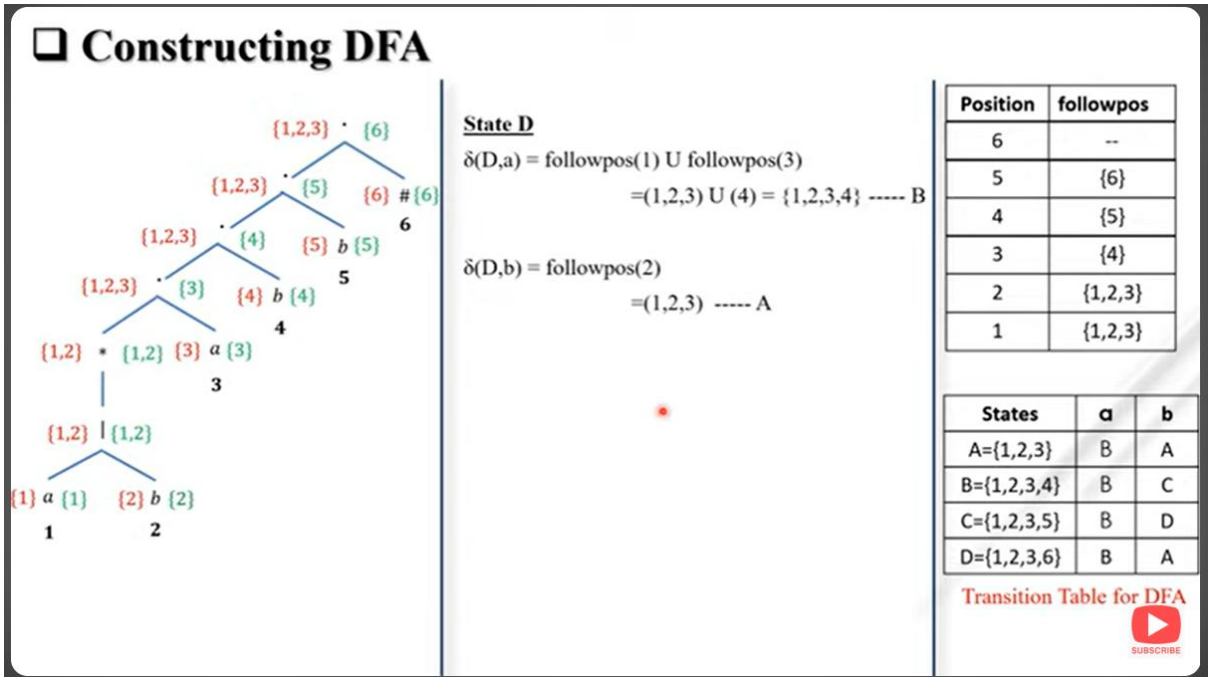
=(1,2,3) U (4) = {1,2,3,4} ----- B

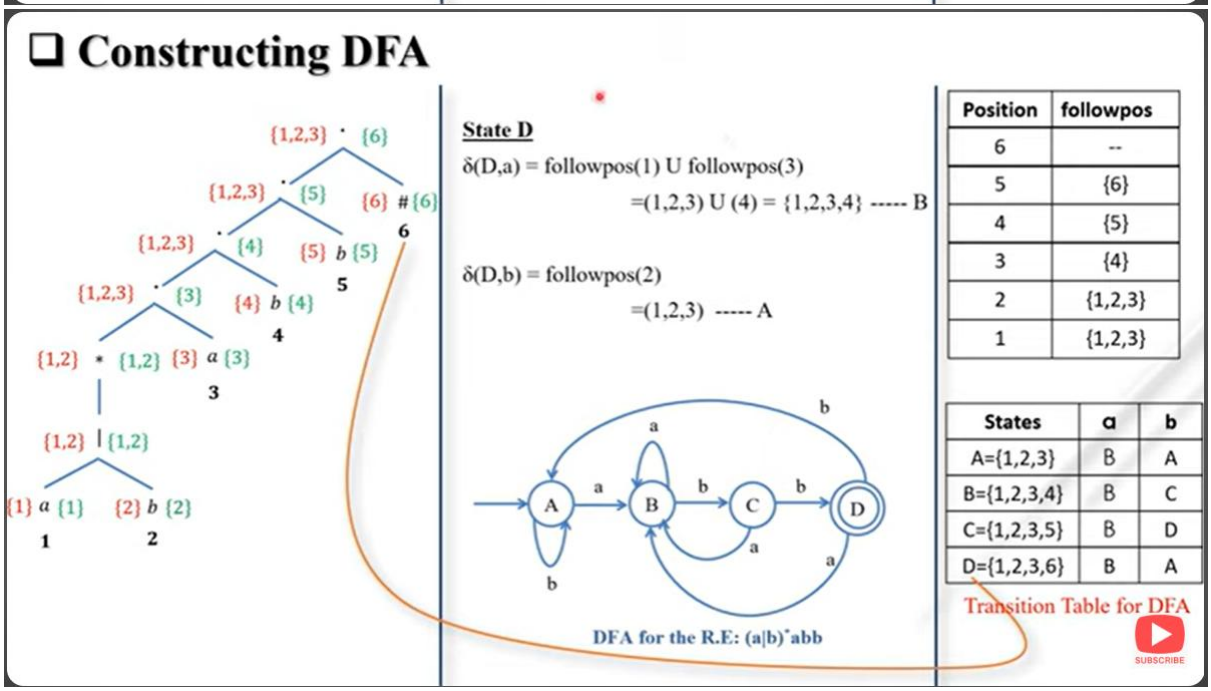
δ((C,b) = followpos(2) U followpos(5)

=(1,2,3) U (6) = {1,2,3,6} ----- D

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}	B	C
C={1,2,3,5}	B	D
D={1,2,3,6}		

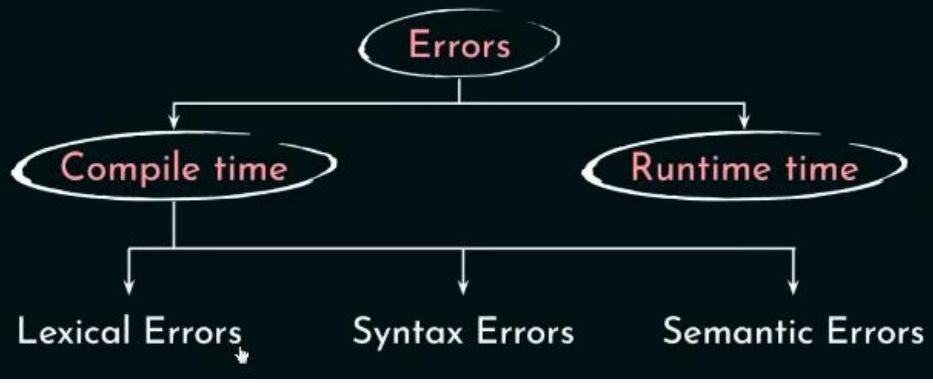




Errors and Error Recovery in Lexical Analysis

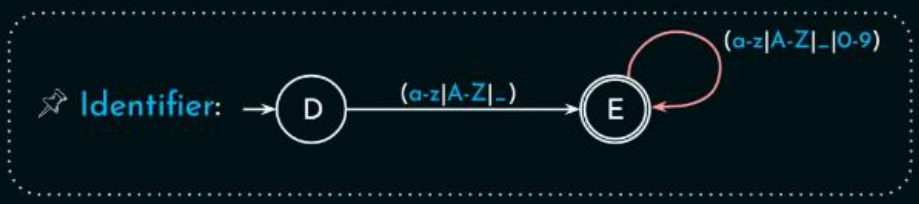
https://www.geeksforgeeks.org/error-recovery-strategies-in-compiler-design/

Classification of Errors:



Lexical Errors:

- Identifiers that are way too long.



: 31/247 : 2048 : 79

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.

```
int i = 4567891;
```

Size: 2 Bytes

-32,768 to 32,767

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.
- Numeric constants which are ill-formed.

```
int i = 4567$91;
```

Lexical Errors:

- Identifiers that are way too long.
- Exceeding length of numeric constants.
- Numeric constants which are ill-formed.
- Illegal characters that are absent from the source code.

```
char x[] = "NESO ACADEMY";$
```

Lexical Error–Recovery:

- Panic-mode recovery.
- Transpose of two adjacent characters.

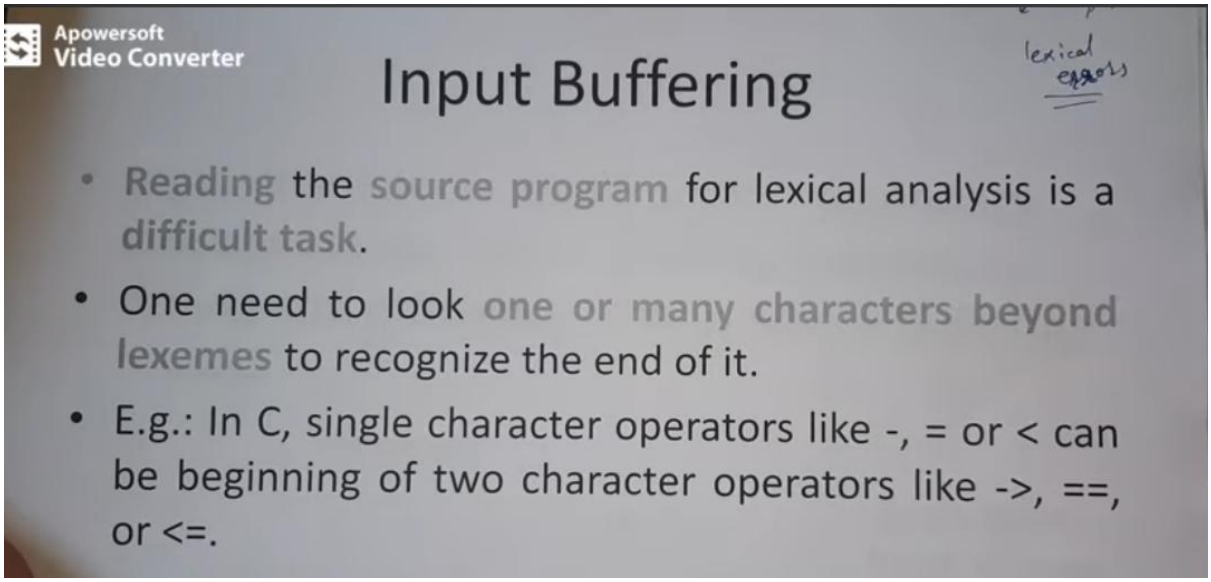
```
unoin test
{
   int x;
   float y;
} T1;
```

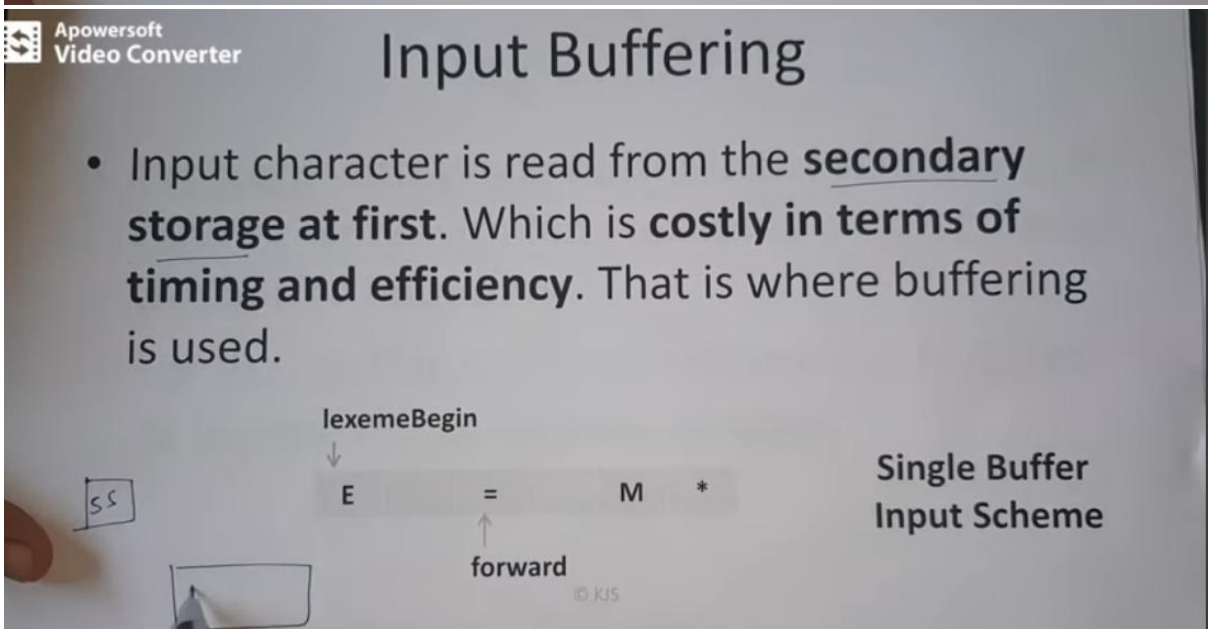
```
union test
{
   int x;
   float y;
} T1;
```

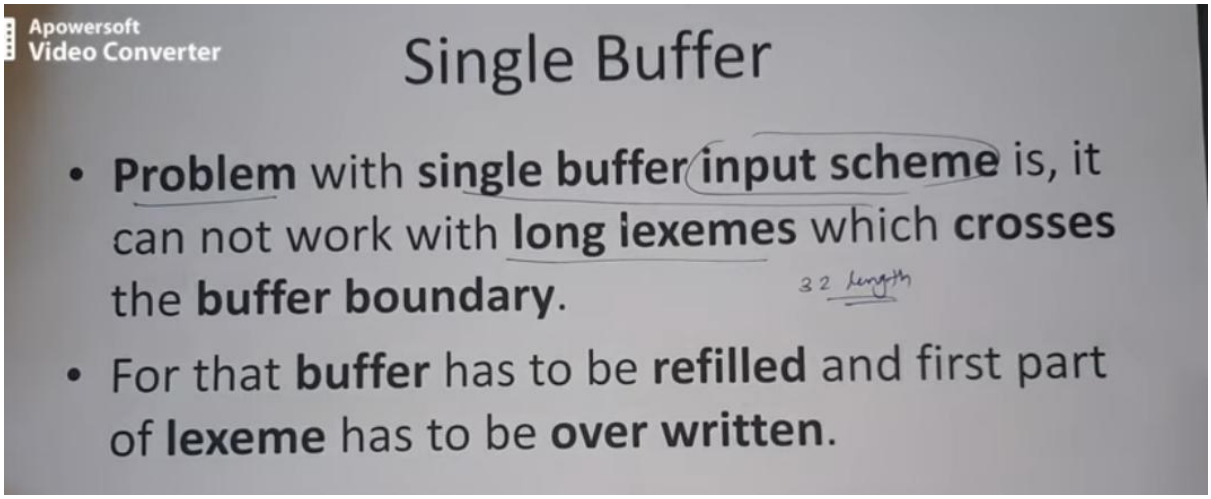
Lexical Error–Recovery:

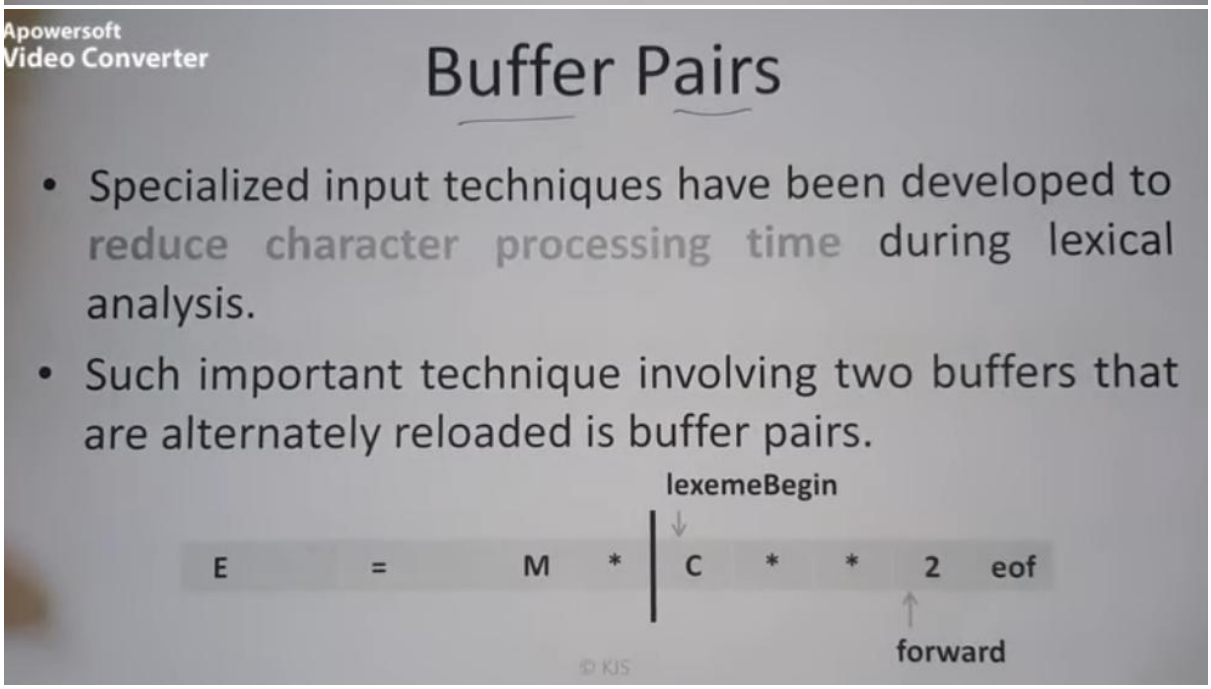
- Panic-mode recovery.
- Transpose of two adjacent characters.
- Insert a missing character.
- Delete an unknown or extra character.
- Replace a character with another.

Input Buffer in Lexical Analyzer





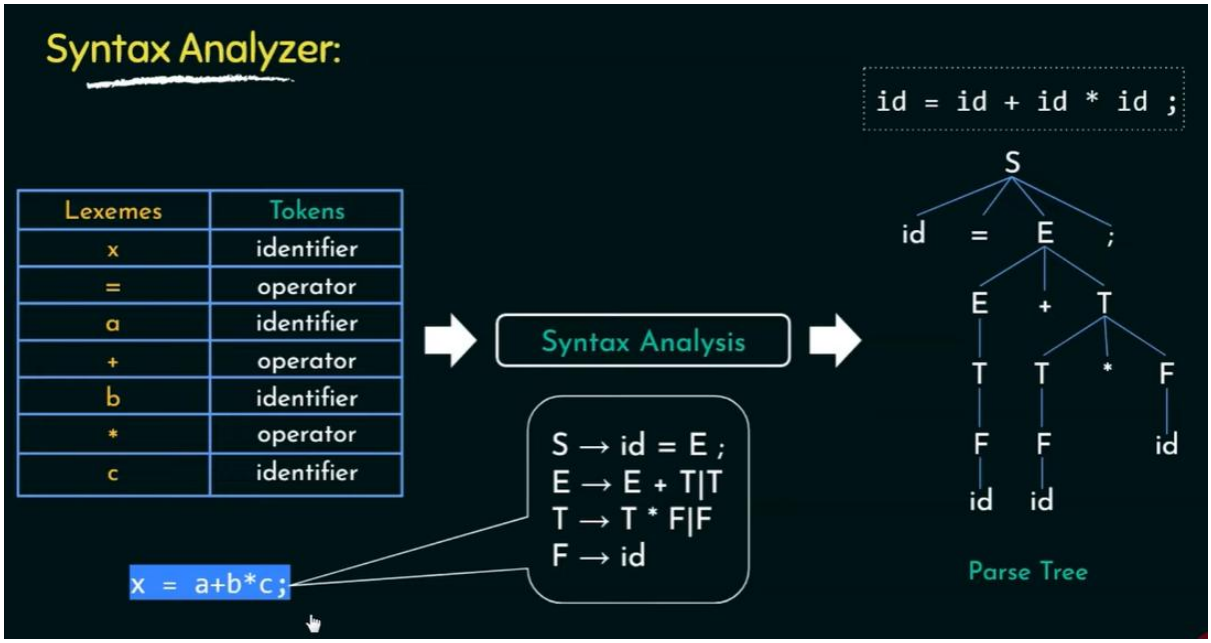


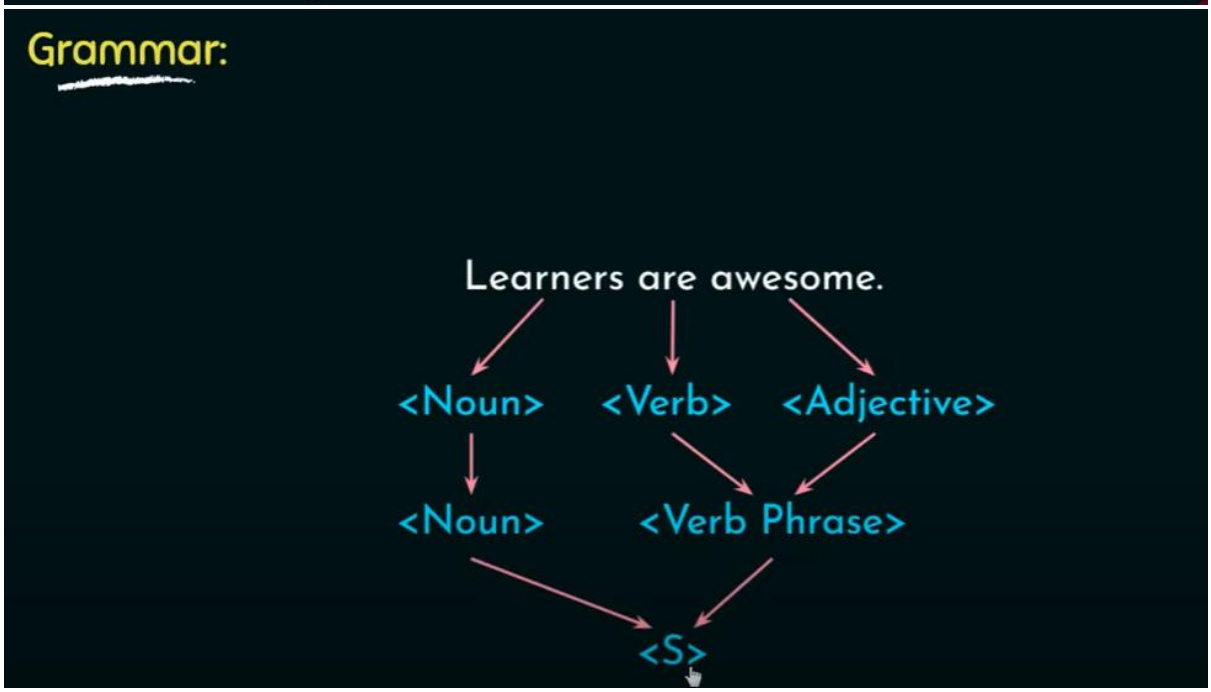


https://www.tutorialspoint.com/what-is-input-buffering-in-compiler-design

Introduction to Formal Grammars

https://www.geeksforgeeks.org/token-patterns-and-lexems/


Syntax Analyzer:
id = id + id * id ;
Lexemes
Tokens
x
identifier
=
operator
a
identifier
+
operator
b
identifier
*
operator
c
identifier
Syntax Analysis
S → id = E ;
E → E + T|T
T → T * F|F
F → id
x = a+b*c;
S
id = E ;
E + T
T T * F
F F id
id id
Parse Tree


Grammar:
Learners are awesome.
<Noun>
<Verb>
<Adjective>
<Noun>
<Verb Phrase>
<S>

Grammar:

Variables
or
Non-terminals

Constants
or
Terminals

Grammar: (N, T, P, S)

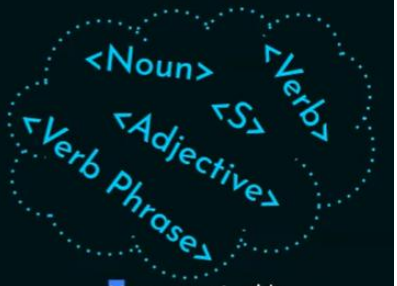
<S> ⟶ <Noun> <Verb Phrase>

<Verb Phrase> ⟶ <Verb> <Adjective>

<Noun> ⟶ Learners | Neso

<Verb> ⟶ are | is

<Adjective> ⟶ awesome | good

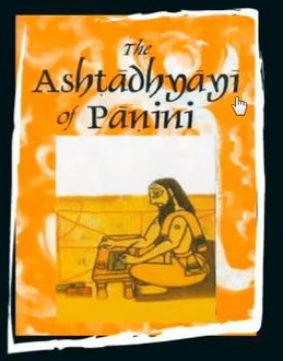


Non-terminals

Terminals

Grammar:

A phrase structure grammar (or simply grammar) is (N, T, P, S) where



i. N is a finite, non-empty set of Non-terminals.

ii. T is a finite, non-empty set of Terminals.

iii. $N \cap T = \Phi$

iv. S is a special non-terminal (i.e. $S \in N$), called Start symbol.

v. P is a finite set whose elements are of the form, $\alpha \rightarrow \beta$

NUT

Production Rules

Classifications of Formal Grammars (Part 1)

Grammar:

A phrase structure grammar (or simply grammar) is (N, T, P, S) where

i. N is a finite, non-empty set of Non-terminals.

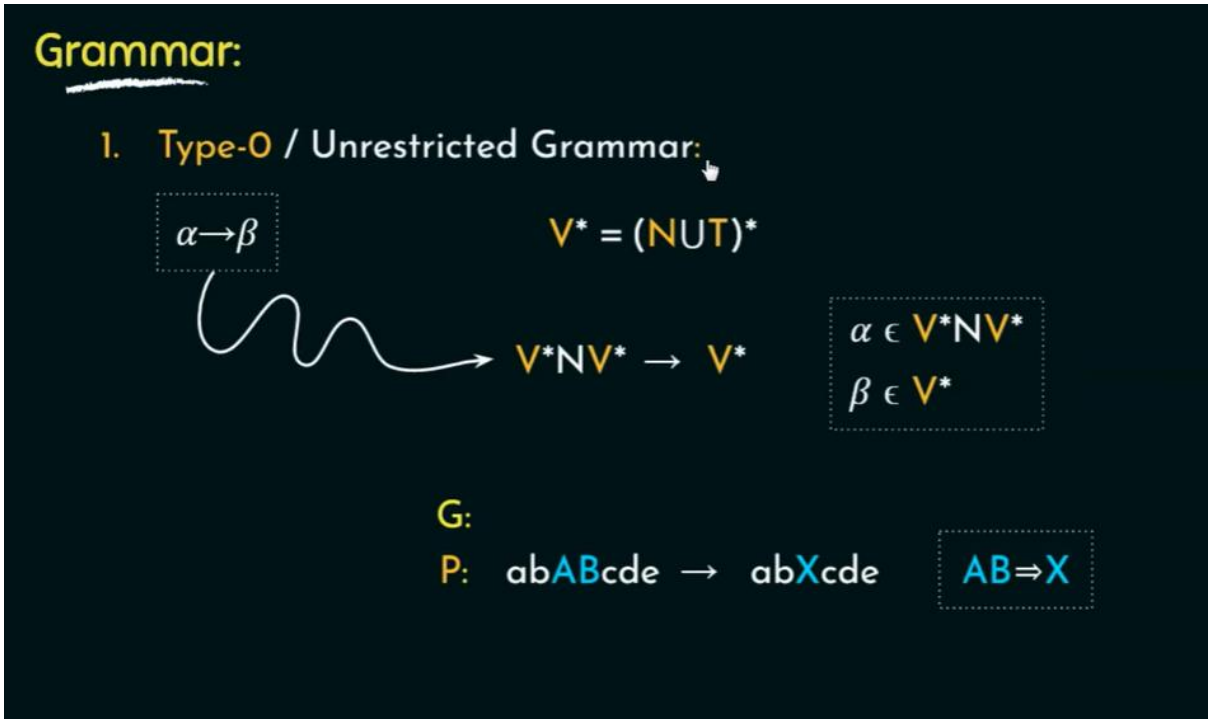
ii. T is a finite, non-empty set of Terminals.

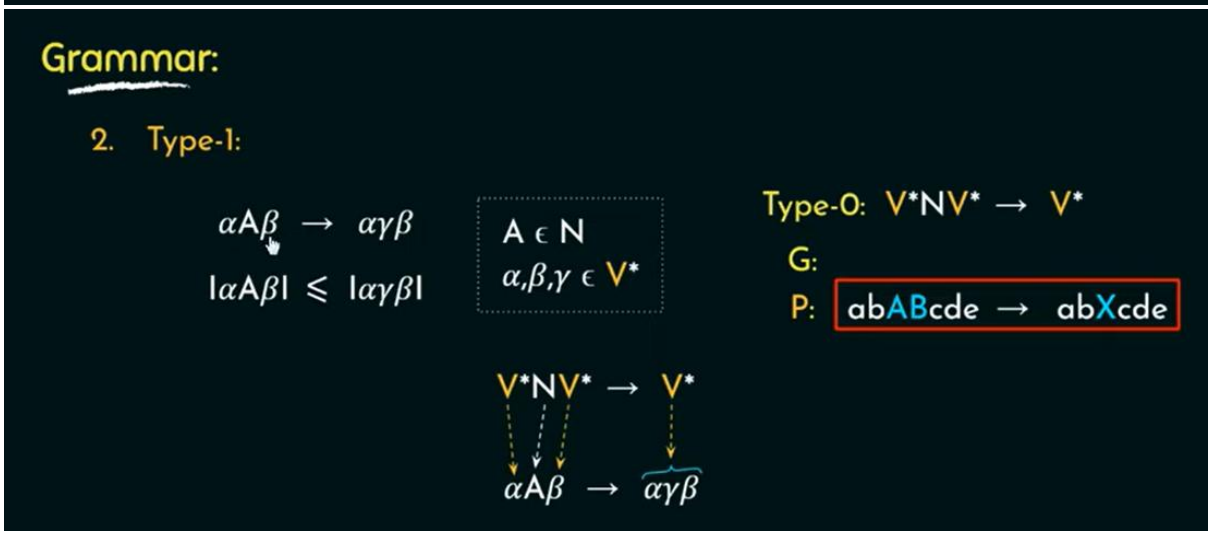
iii. $N \cap T = \Phi$

iv. S is a special non-terminal (i.e. $S \in N$), called Start symbol.

v. P is a finite set Production Rules of the form,

$\alpha \rightarrow \beta$





Classifications of Formal Grammars (Part 2)

Derivations of CFGs

Check from theory notes of Sunil Ghane sir

Ambiguity in CFGs

AMBIGUOUS GRAMMAR

A CFG is said to be ambiguous if there exists more than one derivation trees for the given i/p string. [or]
more than one leftmost derivation
[or] " " " rightmost "

There is [no] standard method to check ambiguity.
we have to do it by practice / hit & trial method.
problem with ambiguity → precedance of operators is violated / not respected
eg. $E \to E+E \mid E*E \mid id$

problem with ambiguity → precedance of operators is violated / not respected
eg. $E \to E+E \mid E*E \mid id$

we have to derive $id+id*id$

LMD: $E \to (E)+E$
$\to id+E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

LMD: // in exams don't write LMD use full form
$E \to E*E$
$\to E+E*E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

RMD.
$E \to E+E$
$E \to E+E*E$
$\to E+E*id$
$\to E+id*id$
$E \to id+id*id$

RMD:
$E \to E*E$
$E \to E*id$
$E \to E+E*id$
$E \to E+id*id$
$E \to id+id*id$

Left-most DERIVATION

Derivation is a sequence of production rules. It is used to get the i/p string through these production rules.

We have to decide
- → which Non-Terminal to replace
- → production rule by which the non-terminal will be replaced.

2 options for this

Left most Derivation | Right most Derivation

↓

I/P scanned & replaced with the production rule from left → right

eg $S \rightarrow S+S \mid S-S \mid a \mid b \mid c$

input: a-b+c

LMD →

$S \rightarrow S+S$

$S \rightarrow S-S+S$

$S \rightarrow a-S+S$

$S \rightarrow a-b+S$

$S \rightarrow a-b+c$

Right most Derivation

In Right most derivation i/p is scanned and replaced with the production rule from Right → Left

eg) $S \rightarrow S+S \mid S-S \mid a \mid b \mid c$ same eg as prev.

RMD will be

$S \rightarrow S-S$

$S \rightarrow S-S+S$

$S \rightarrow S-S+c$

$S \rightarrow S-b+c$

$S \rightarrow a-b+c$

PARSE TREE

It is a typical depiction of how the start symbol of a grammer derives a string in the language.

or

It is a graphical representation of symbol that can be terminals or non-terminals

Properties

1) Root is always the start symbol
2) All leaf nodes are terminals
3) All interior nodes → non-terminals

eg $S \rightarrow XYZ$, $X \rightarrow a$, $Y \rightarrow b$, $Z \rightarrow c \mid d$

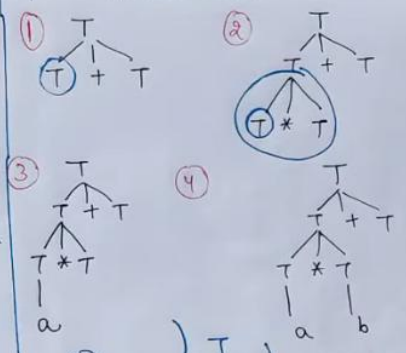
$L = \{abc, abd\}$

a b c

eg $T \rightarrow T+T \mid T*T$, $T \rightarrow a \mid b \mid c$

input: a*b+c

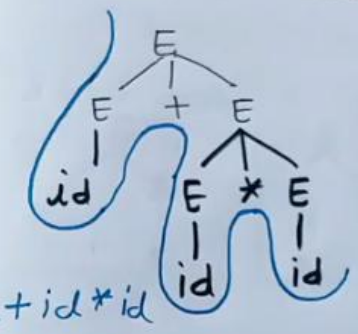
(5*2)+3 = 10+3 = 13



Parse Tree yield

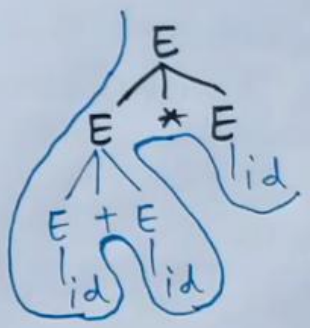
a*b+c

Parse Tree 1 (is valid)



id+id*id

Parse Tree -2



id+id*id

So, since we have more than 1 parse Tree possible
∴ Grammer is ambiguous.

Ambiguous in the sense that " if we have 2 parse trees possible then parser will get confused about which one to generate/ or which one is correct/right."

trees are showing

id + id * id

So, since we have more than 1 parse tree possible
∴ Grammer is ambiguous.

Ambiguous in the sense that "if we have 2 parse trees possible then parser will get confused about which one to generate/ or which one is correct/right."

In the above case, although parse trees are showing o/p is same
but evaluation is d/f

eg if we have i/p 1 + 5 * 3
our answer should be = 16

but parse tree 1 generates 16
and parse Tree 2 generates 18

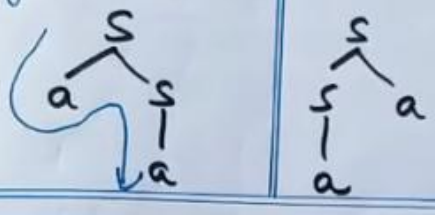
So parsers donot allow ambiguous grammar except of one

operator precedance parser

Ques Check grammar is ambiguous or not?

(i) $S \rightarrow aS \mid Sa \mid a$

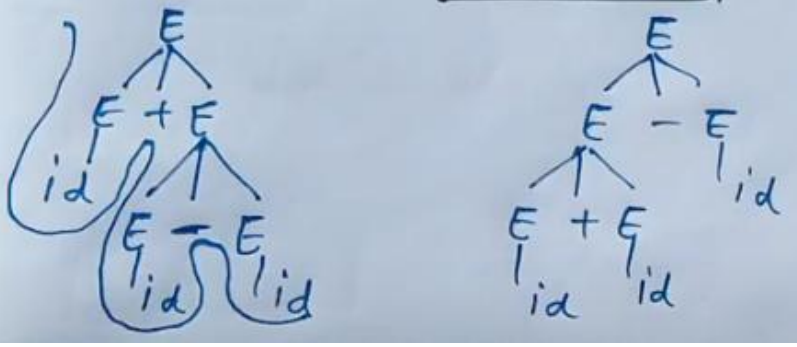
string - aa



To check ambiguity, there are no algos
ie it is undecidable.

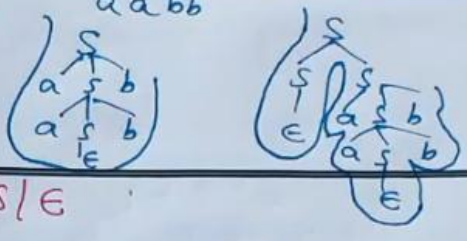
(ii) $E \to E+E$
$E \to E-E$
$E \to id$
$\equiv$ $E \to E+E \mid E-E \mid id$

id+id-id



(iii) $S \to aSb \mid SS$
$S \to \epsilon$

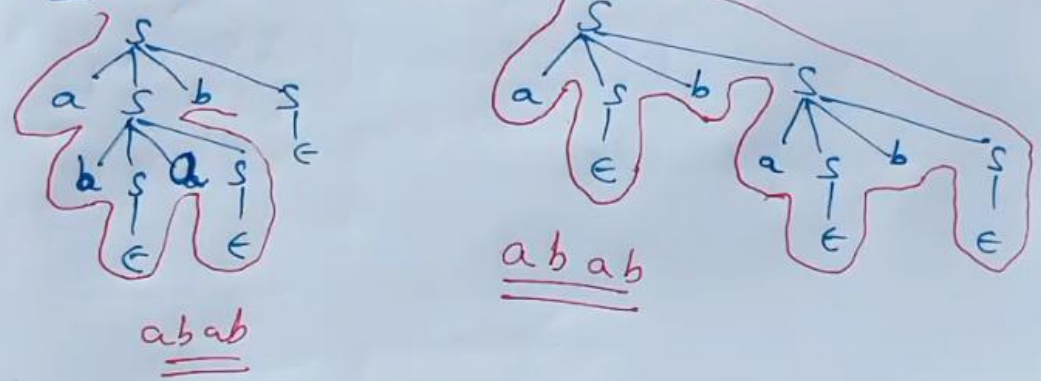
aabb



aabb

(iv) $S \to aSbS \mid bSaS \mid \epsilon$

abab



abab

abab

Context Free Grammar:

1. $E \rightarrow E + E$
2. $E \rightarrow E \times E$
3. $E \rightarrow id$

id + id x id

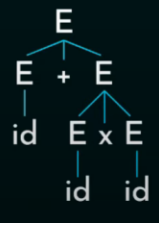
Left Most Derivation:

$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E \times E$
$\Rightarrow id + id \times E$
$\Rightarrow id + id \times id$

Right Most Derivation:

$E \Rightarrow E + E$
$\Rightarrow E + E \times E$
$\Rightarrow E + E \times id$
$\Rightarrow E + id \times id$
$\Rightarrow id + id \times id$

Parse Tree Derivation:



Context Free Grammar: E → E + E | E x E | id

id + id x id

Left Most Derivation:

$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E \times E$
$\Rightarrow id + id \times E$
$\Rightarrow id + id \times id$

$E \Rightarrow E \times E$
$\Rightarrow E + E \times E$
$\Rightarrow id + E \times E$
$\Rightarrow id + id \times E$
$\Rightarrow id + id \times id$

Right Most Derivation:

$E \Rightarrow E + E$
$\Rightarrow E + E \times E$
$\Rightarrow E + E \times id$
$\Rightarrow E + id \times id$
$\Rightarrow id + id \times id$

$E \Rightarrow E \times E$
$\Rightarrow E \times id$
$\Rightarrow E + E \times id$
$\Rightarrow E + id \times id$
$\Rightarrow id + id \times id$

Parse Tree Derivation:

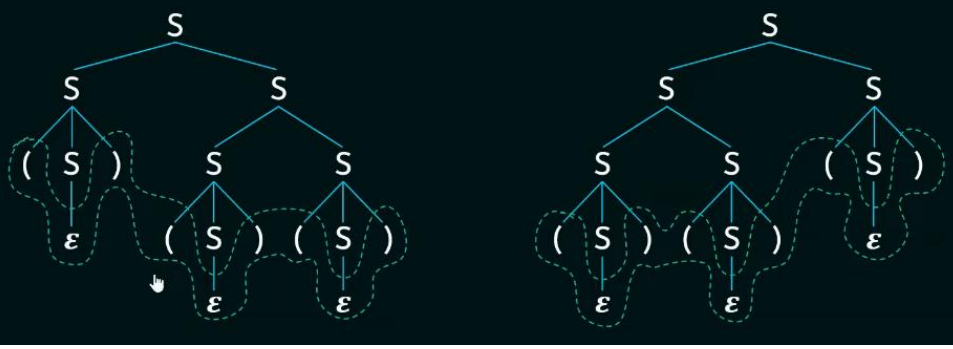
Ambiguity in CFGs - Solved Problems (Set 1)

Q1: Determine whether the following grammar is suitable for predictive parsing,

$$S \rightarrow (S) \mid SS \mid \varepsilon$$

UGC-NET DEC 2018

Sol. Parse Tree Derivation: ()()()



Since for the derivation the same string we obtain two different parse trees hence the grammar is ambiguous.

Q2: Consider the following two Grammars,

G1: $S \rightarrow SbS \mid a$
G2: $S \rightarrow aB \mid ab$, $A \rightarrow AB \mid a$, $B \rightarrow ABb \mid b$

UGC-NET JUNE 2018

Which of the following option is correct?

(A) Only G1 is ambiguous

(B) Only G2 is ambiguous

(C) Both G1 and G2 are ambiguous

(D) Both G1 and G2 are unambiguous

Q2: Consider the following two Grammars,

G1: S → SbS | a
G2: S → aB | ab, A → AB | a, B → ABb | b

Which of the following option is correct?

Sol. G1: S → SbS | a

ababa

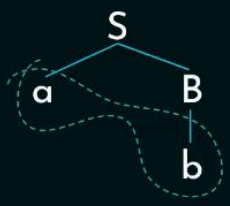
Q2: Consider the following two Grammars,

G1: S → SbS | a
G2: S → aB | ab, A → AB | a, B → ABb | b

Which of the following option is correct?

Sol. G2: S → aB | ab, A → AB | a, B → ABb | b

ab



Q2: Consider the following two Grammars,

G1: $S \rightarrow SbS \mid a$
G2: $S \rightarrow aB \mid ab$, $A \rightarrow AB \mid a$, $B \rightarrow ABb \mid b$

UGC-NET JUNE 2018

Which of the following option is correct?

(A) Only G1 is ambiguous

(B) Only G2 is ambiguous

(C) Both G1 and G2 are ambiguous

(D) Both G1 and G2 are unambiguous

Ambiguity in CFGs - Solved Problems (Set 2)

Q: Which of the following grammars is (are) ambiguous?

(A) $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
(B) $S \rightarrow aSbS \mid bSaS \mid \lambda$
(C) $S \rightarrow aAB$, $A \rightarrow bBb$, $B \rightarrow A \mid \lambda$

UGC-NET NOV 2020

Where λ denotes empty string.

1. (A) and (C) only
2. (B) only
3. (B) and (C) only
4. (A) and (B) only

Q: Which of the following grammars is (are) ambiguous?

✓(A) $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
(B) $S \rightarrow aSbS \mid bSaS \mid \lambda$
(C) $S \rightarrow aAB$, $A \rightarrow bBb$, $B \rightarrow A \mid \lambda$

Where λ denotes empty string.

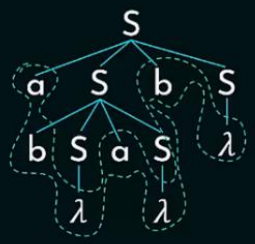
Sol. (A) $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$ λ

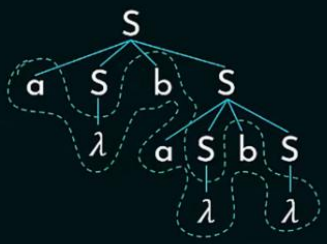
Q: Which of the following grammars is (are) ambiguous?

✓(A) S → SS | aSb | bSa | λ
(B) S → aSbS | bSaS | λ
(C) S → aAB, A → bBb, B → A | λ

Where λ denotes empty string.

Sol. (B) S → aSbS | bSaS | λ abab



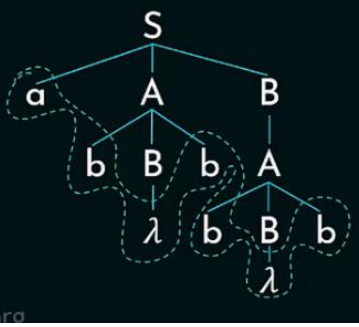


Q: Which of the following grammars is (are) ambiguous?

✓(A) S → SS | aSb | bSa | λ
✓(B) S → aSbS | bSaS | λ
(C) S → aAB, A → bBb, B → A | λ

Where λ denotes empty string.

Sol. (C) S → aAB, A → bBb, B → A | λ abbbb



Q: Which of the following grammars is (are) ambiguous?

✓(A) S → SS I aSb I bSa I λ
✓(B) S → aSbS I bSaS I λ
✓(C) S → aAB, A → bBb, B → A I λ

UGC-NET
NOV 2020

Where λ denotes empty string.

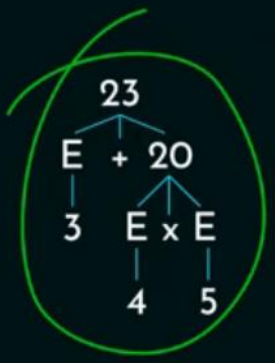
1. (A) and (C) only
2. (B) only
3. (B) and (C) only
4. (A) and (B) only

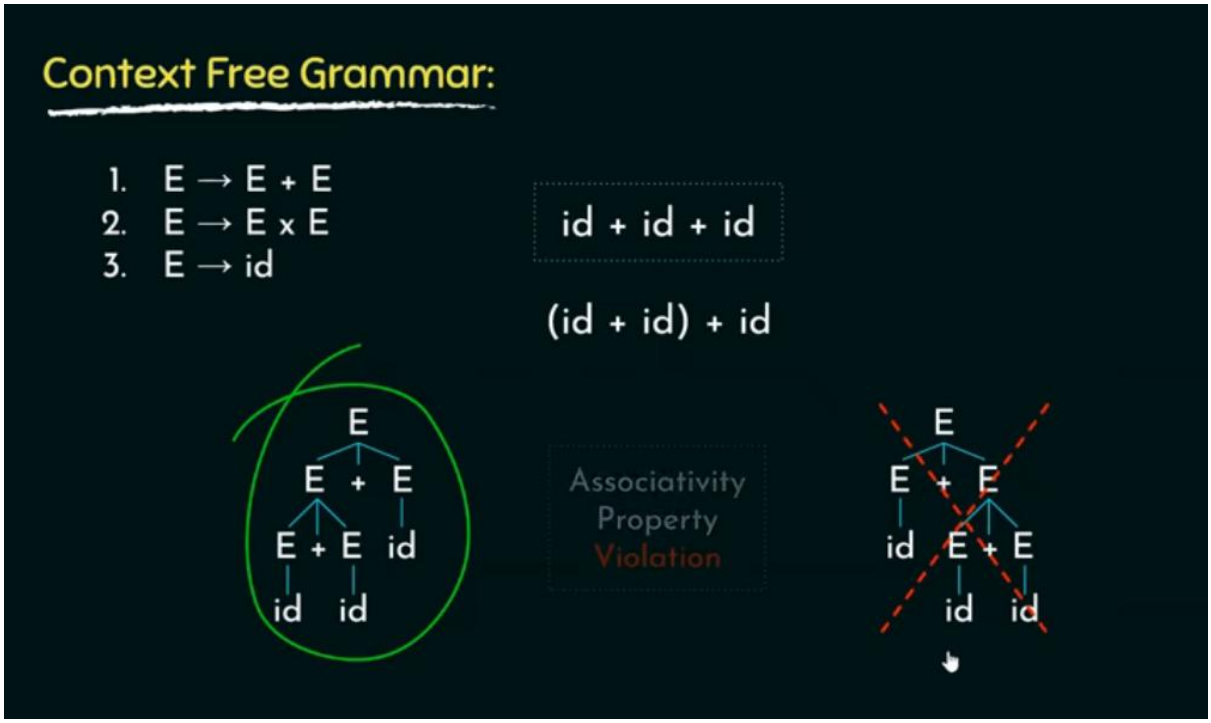
Problems of Ambiguity in CFGs

Context Free Grammar:

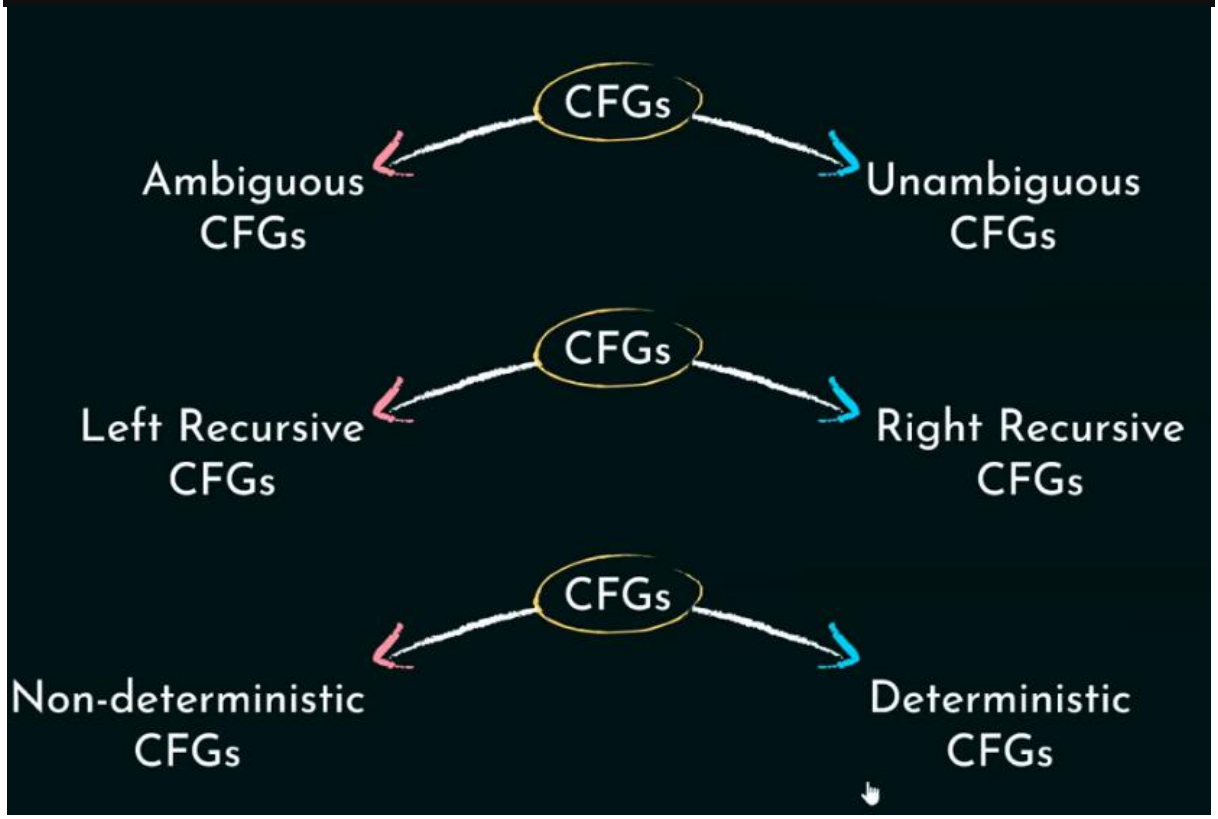
1. E → E + E
2. E → E x E
3. E → id

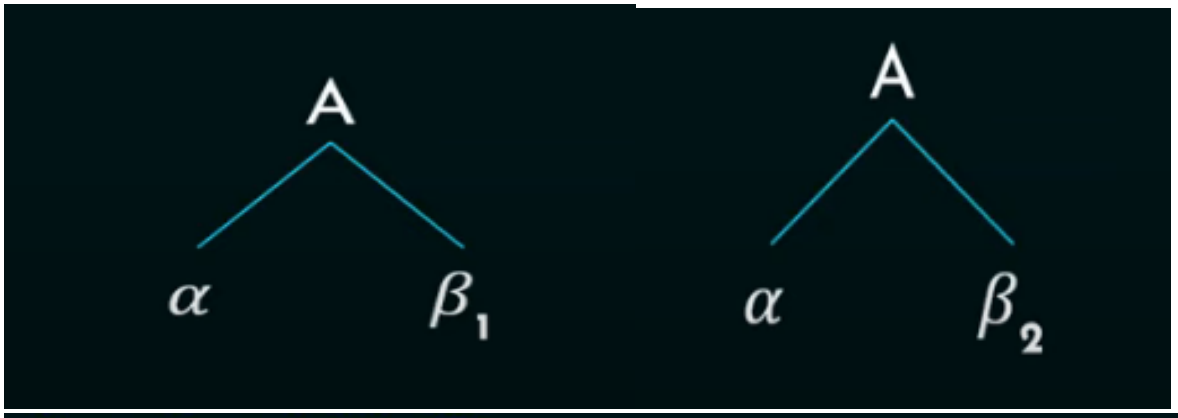
3 + 4 x 5

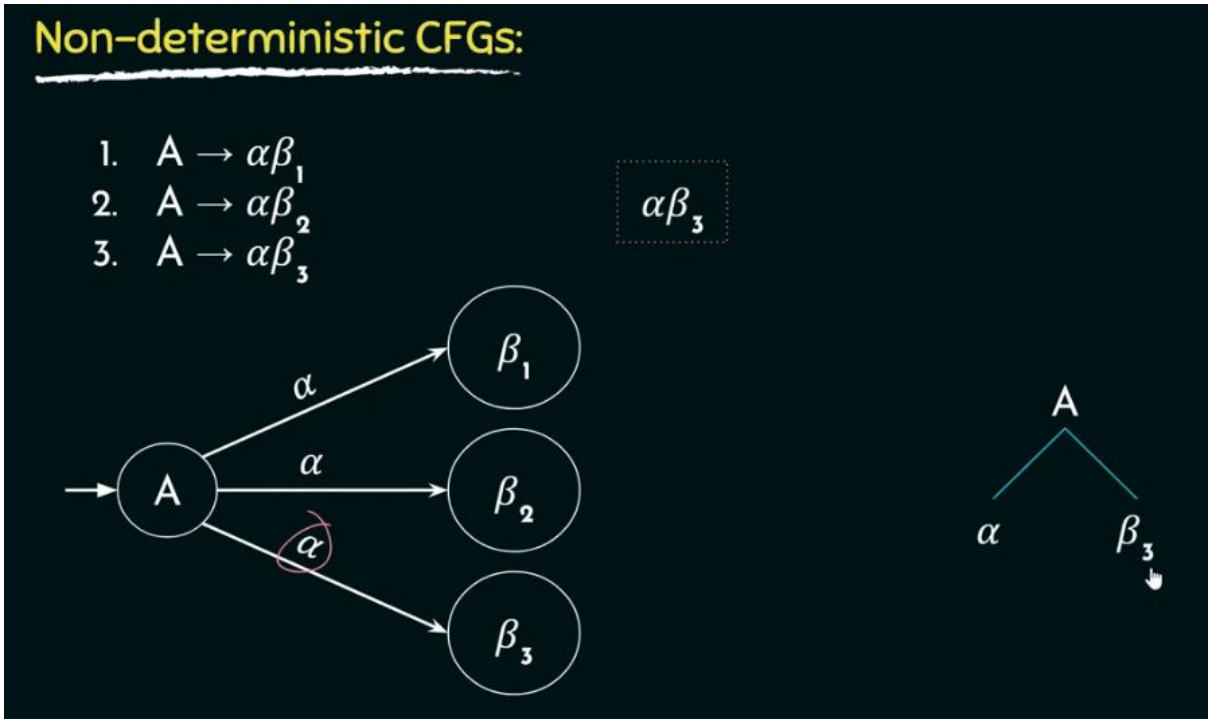




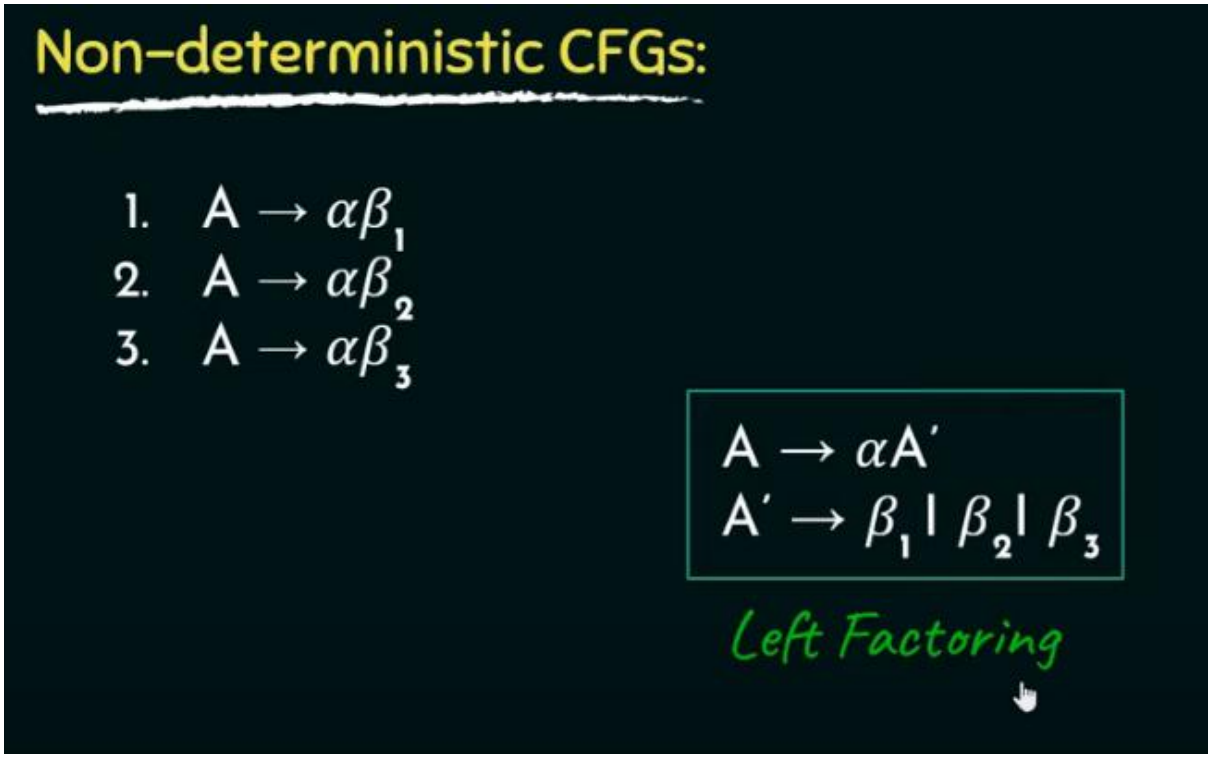
Non-Deterministic CFGs



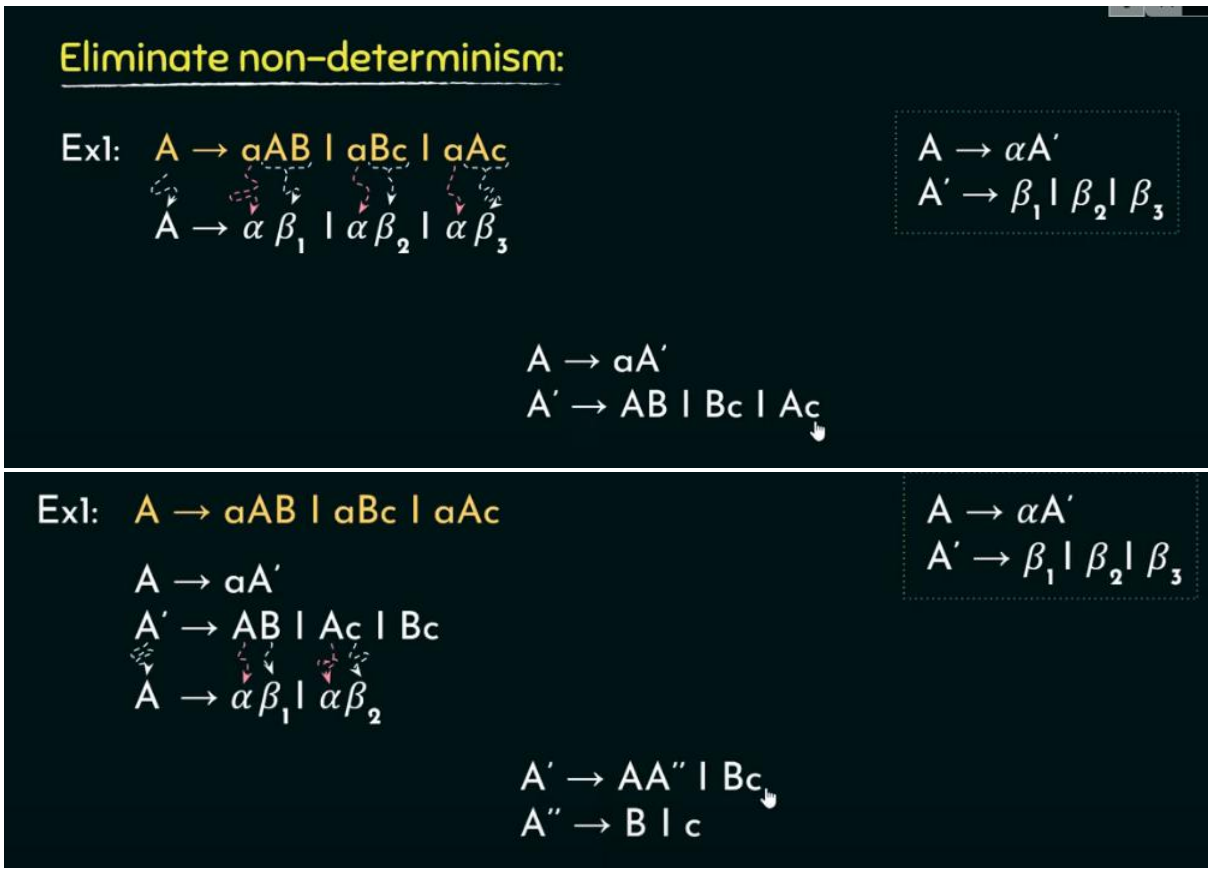


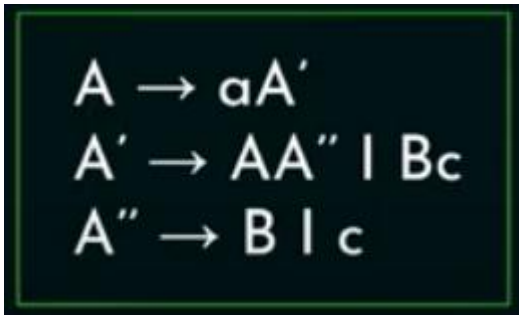


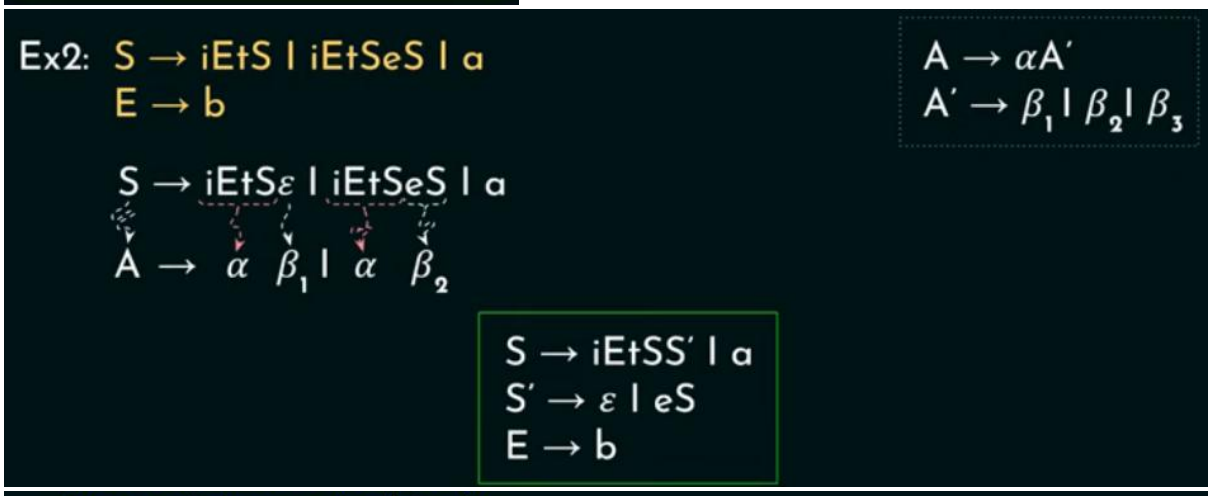
To derive the string the parser has to perform backtracking more than once. Hence to eliminate non-determinism we need to perform left-factoring.

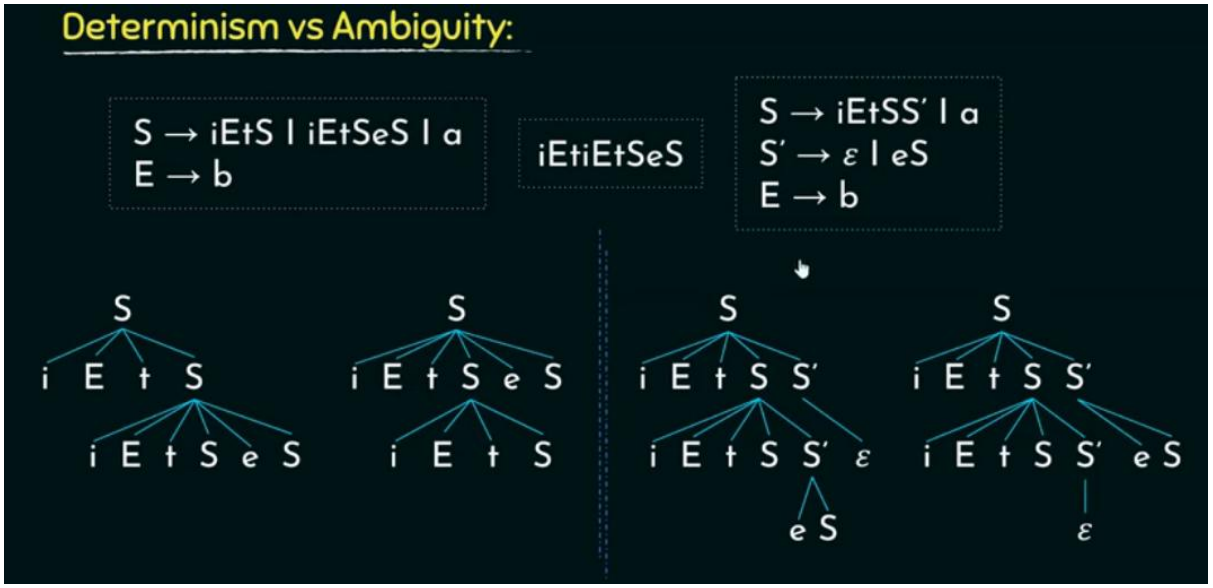


Non-deterministic CFGs – Solved Problems (Set 1)

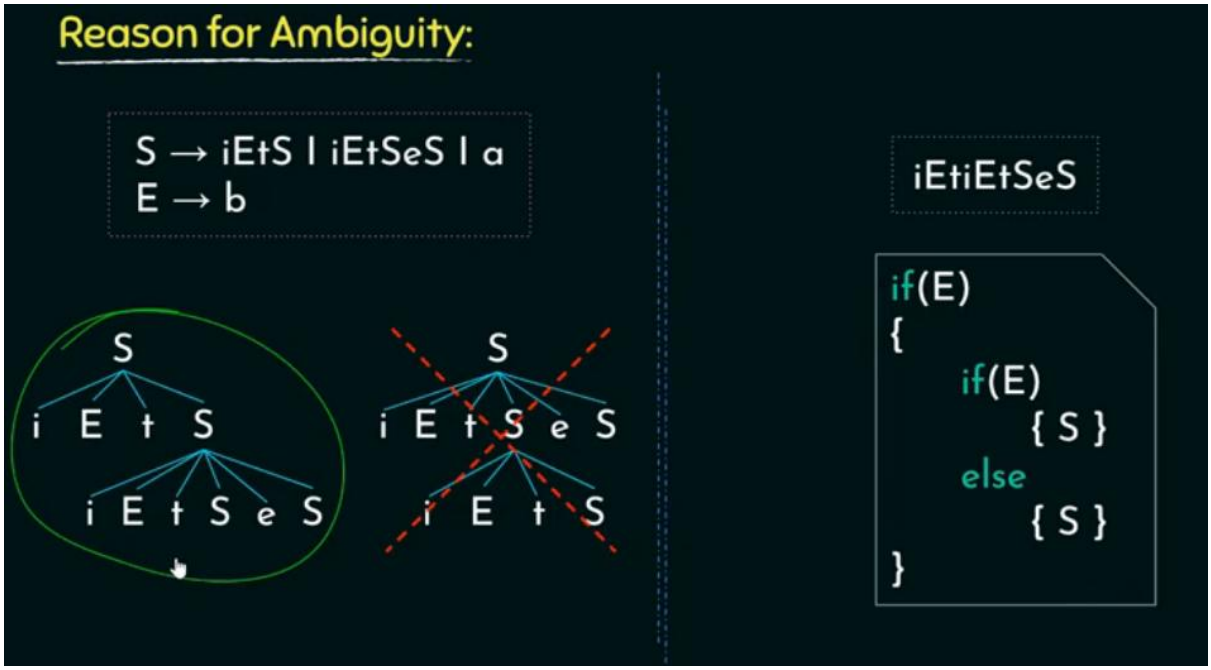




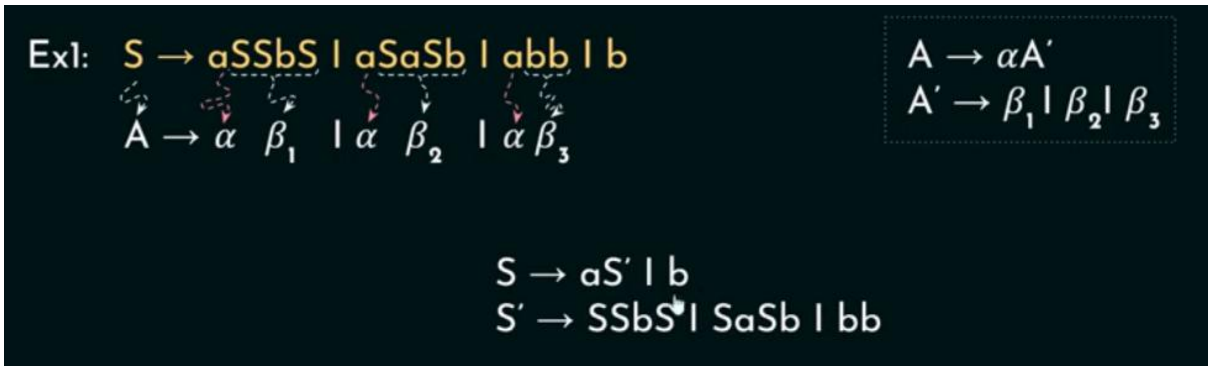


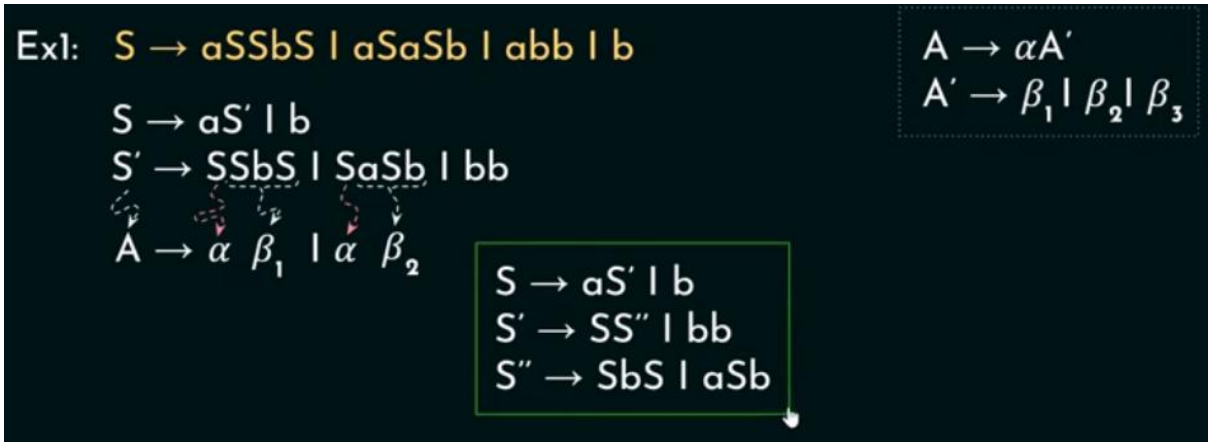


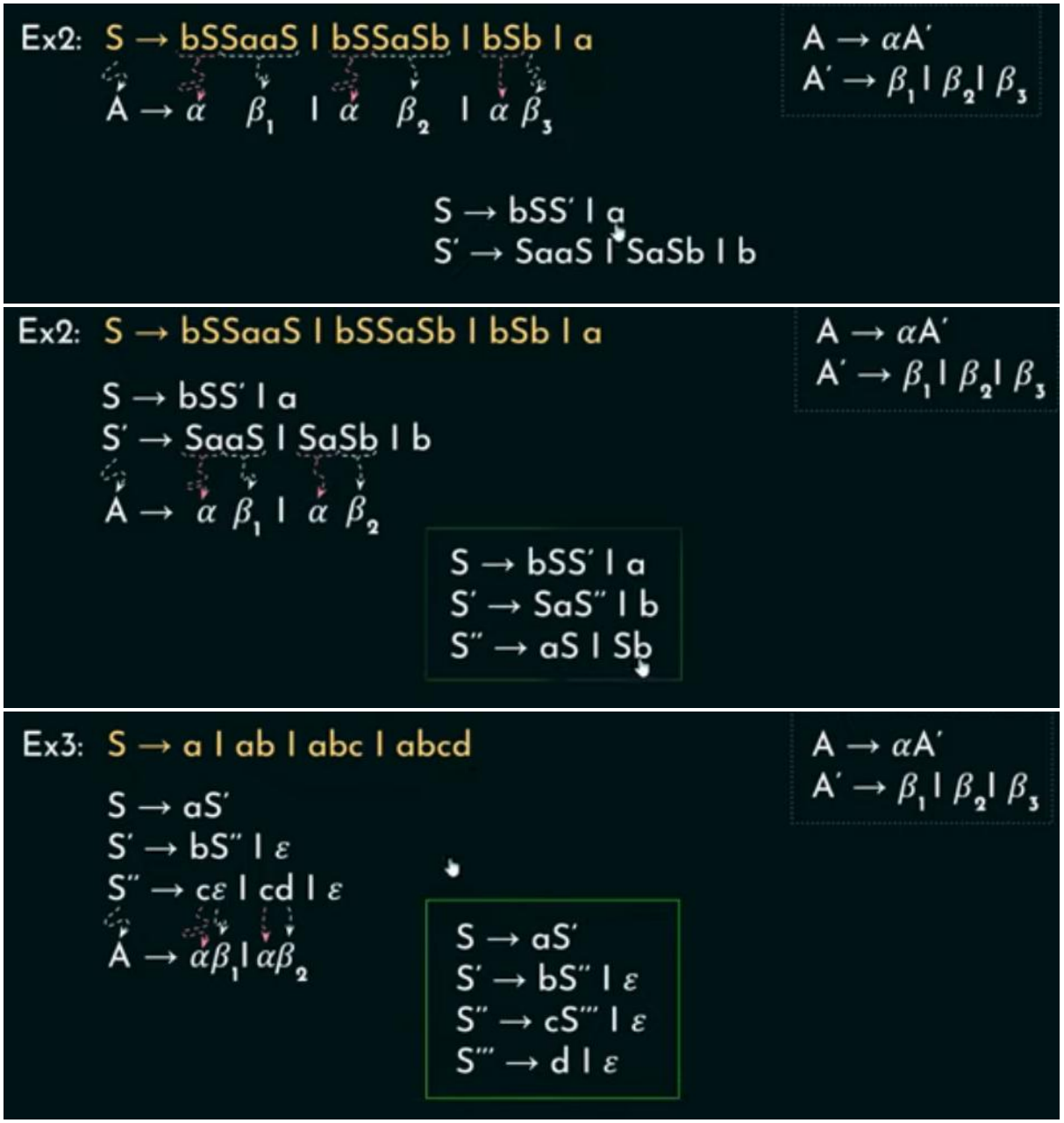
You can see from this example even if you remove non-determinism you can't ensure unambiguity.



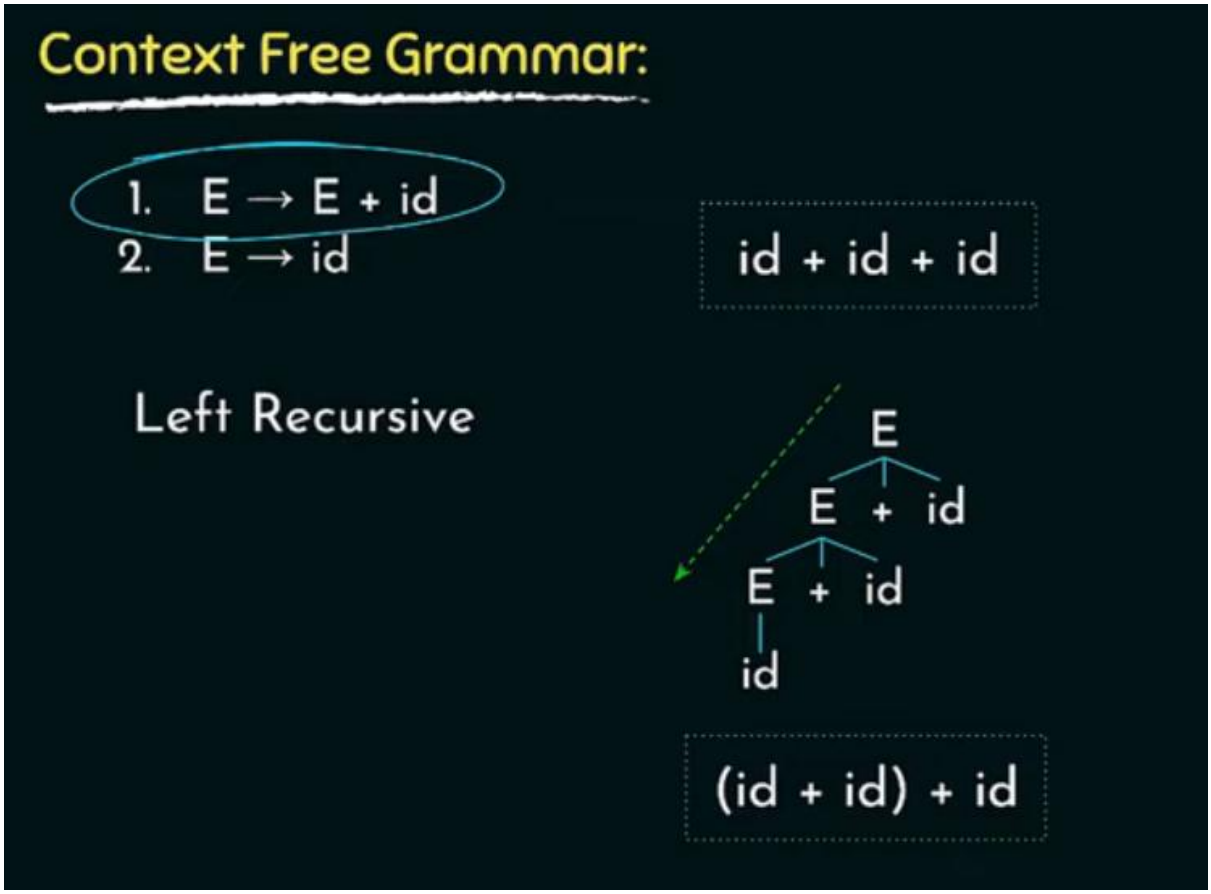
Non-deterministic CFGs – Solved Problems (Set 2)



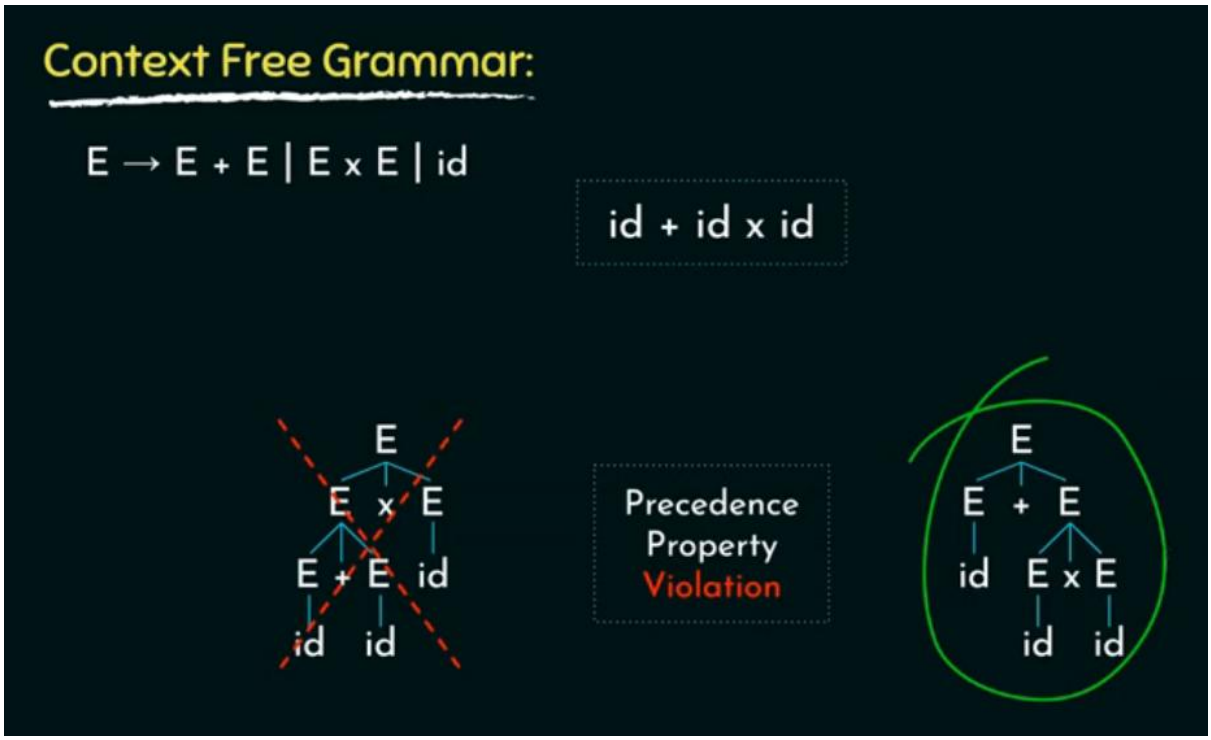




Associativity Violation and Solution in CFGs



Precedence Violation and Solution in CFGs



Context Free Grammar:

$E \rightarrow E + E \mid E \times E \mid id$

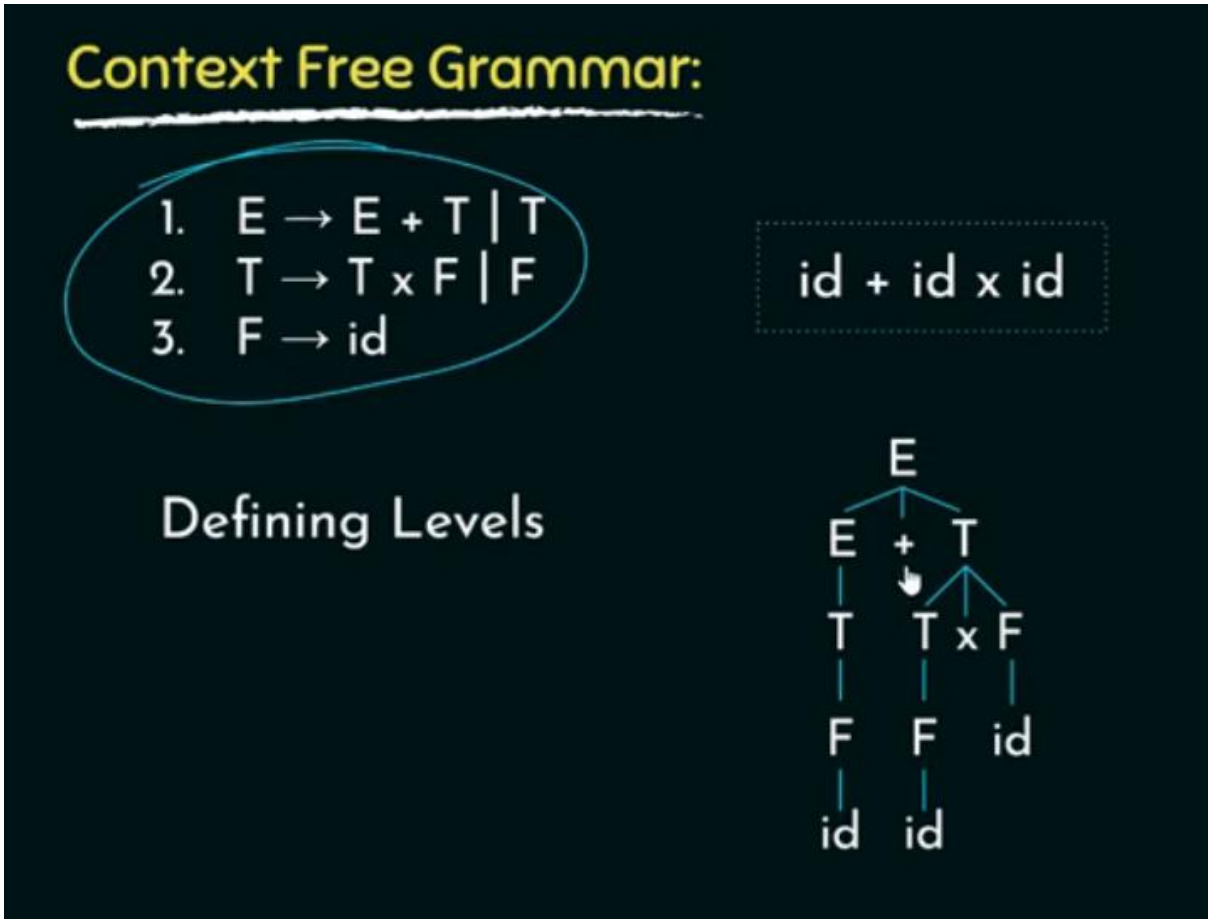
id + id x id

Precedence Property Violation

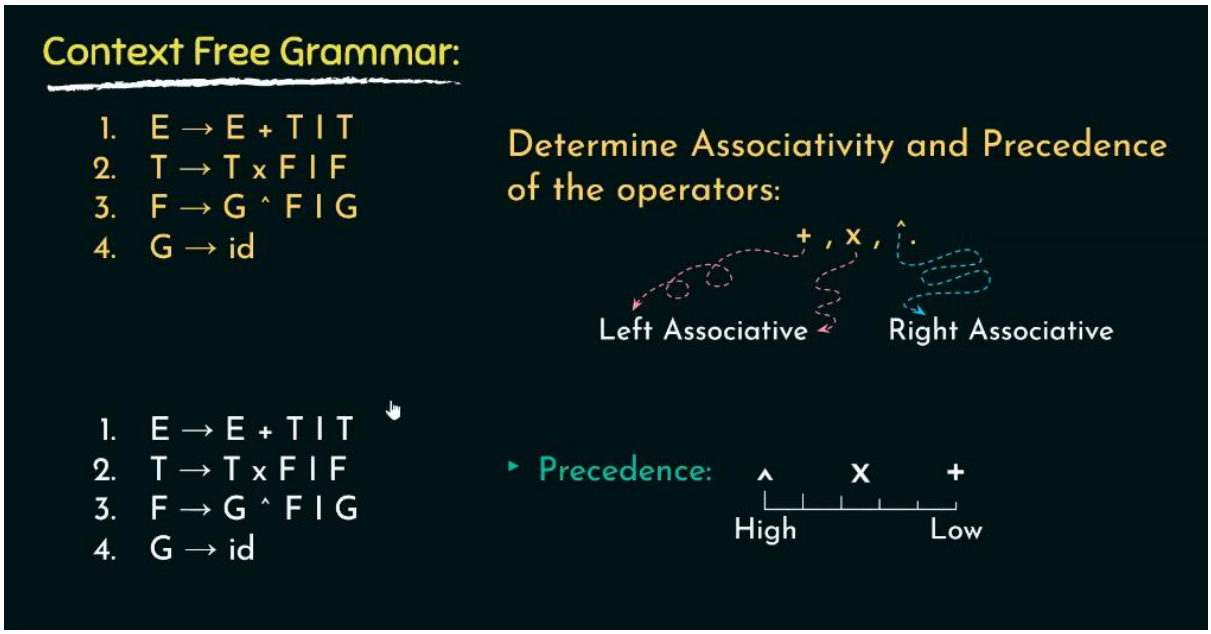
Context Free Grammar:

$E \rightarrow E + E \mid E \times E \mid id$

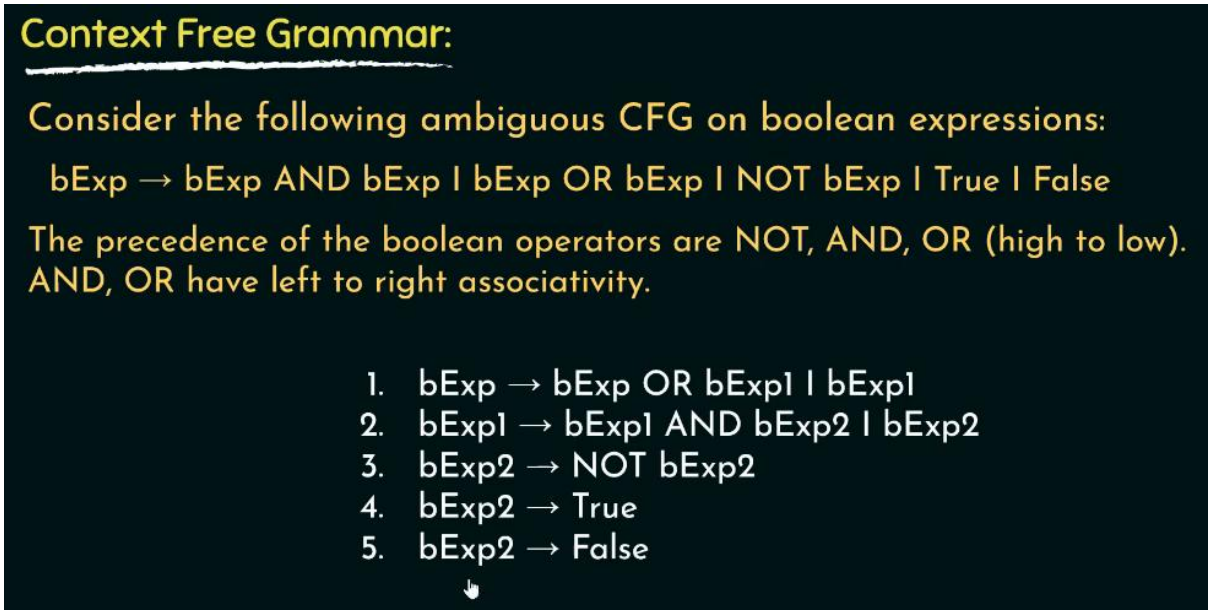
1. $E \rightarrow E + T \mid T$
2. $T \rightarrow T \times F$
3. $F \rightarrow id$

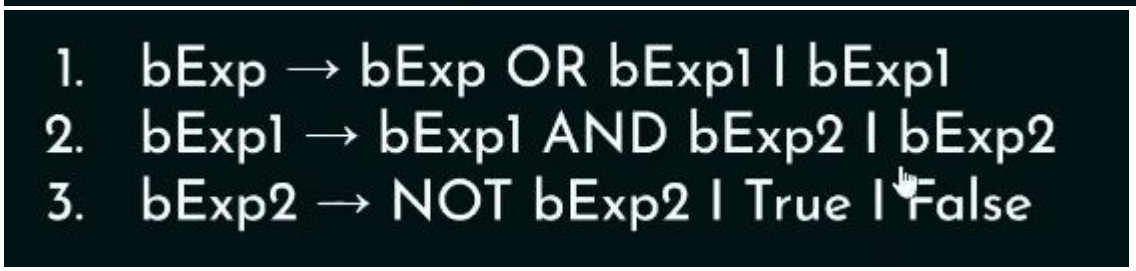


Associativity and Precedence in CFGs

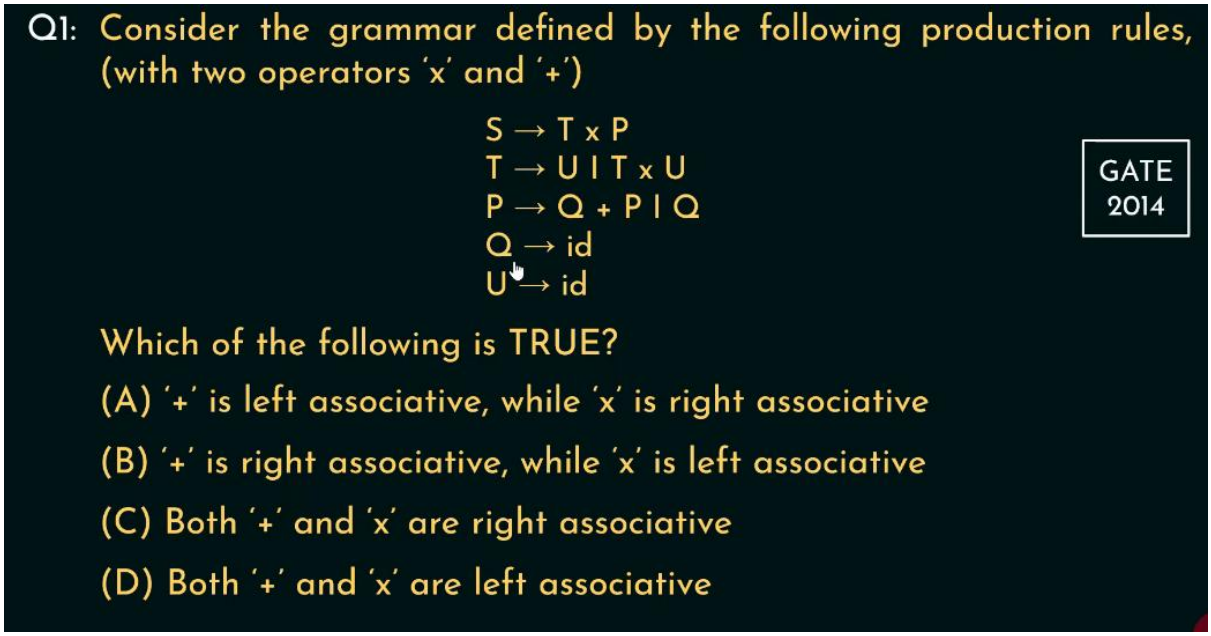


As you can see in the first and second production rules that E and T respectively are left recursive, hence + and x are left associative, whereas in the third production rule that F is right recursive hence ^ is right associative.
As ^ is occurring on the lowest level hence it has the highest precedence, then x and the +.





Associativity and Precedence in CFGs – Solved Problems (Set 1)



Q1: Consider the grammar defined by the following production rules, (with two operators 'x' and '+')

$S \rightarrow T \times P$
$T \rightarrow U \mid T \times U$
$P \rightarrow Q + P \mid Q$
$Q \rightarrow id$
$U \rightarrow id$

Left recursive: $T \rightarrow T \times U$

Right recursive: $P \rightarrow Q + P$

- Precedence: + x

High Low

Which of the following is TRUE?

(A) '+' is left associative, while 'x' is right associative

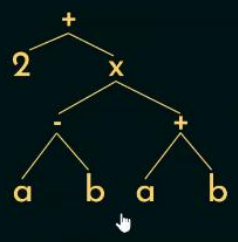
(B) '+' is right associative, while 'x' is left associative

(C) Both '+' and 'x' are right associative

(D) Both '+' and 'x' are left associative

Q2: Consider the parse tree

TIFR PHD CS 2012



Assume that 'x' has higher precedence than '+', '-' and operators associate right to left (i.e. a+b+c = (a+(b+c))).

Consider
(i) 2 + a - b
(ii) 2 + a - b x a + b
(iii) (2 + ((a - b) x (a + b)))
(iv) 2 + (a - b) x (a + b)

The parse tree corresponds to
(A) Expression (i) only
(B) Expression (ii) only
(C) Expression (iv) only
(D) Expressions (ii), (iii) and (iv)
(E) Expressions (iii) and (iv) only

2 + ((a - b) x (a + b))

a b a b

Consider
(i) 2 + a - b
(ii) 2 + a - b x a + b
(iii) (2 + ((a - b) x (a + b)))
(iv) 2 + (a - b) x (a + b)

The parse tree corresponds to
(A) Expression (i) only
(B) Expression (ii) only
(C) Expression (iv) only
(D) Expressions (ii), (iii) and (iv)
(E) Expressions (iii) and (iv) only

Q3: Given the following expression grammar:

$E \rightarrow E \times F \mid F + E \mid F$
$F \rightarrow F - F \mid id$

Which of the following is true?

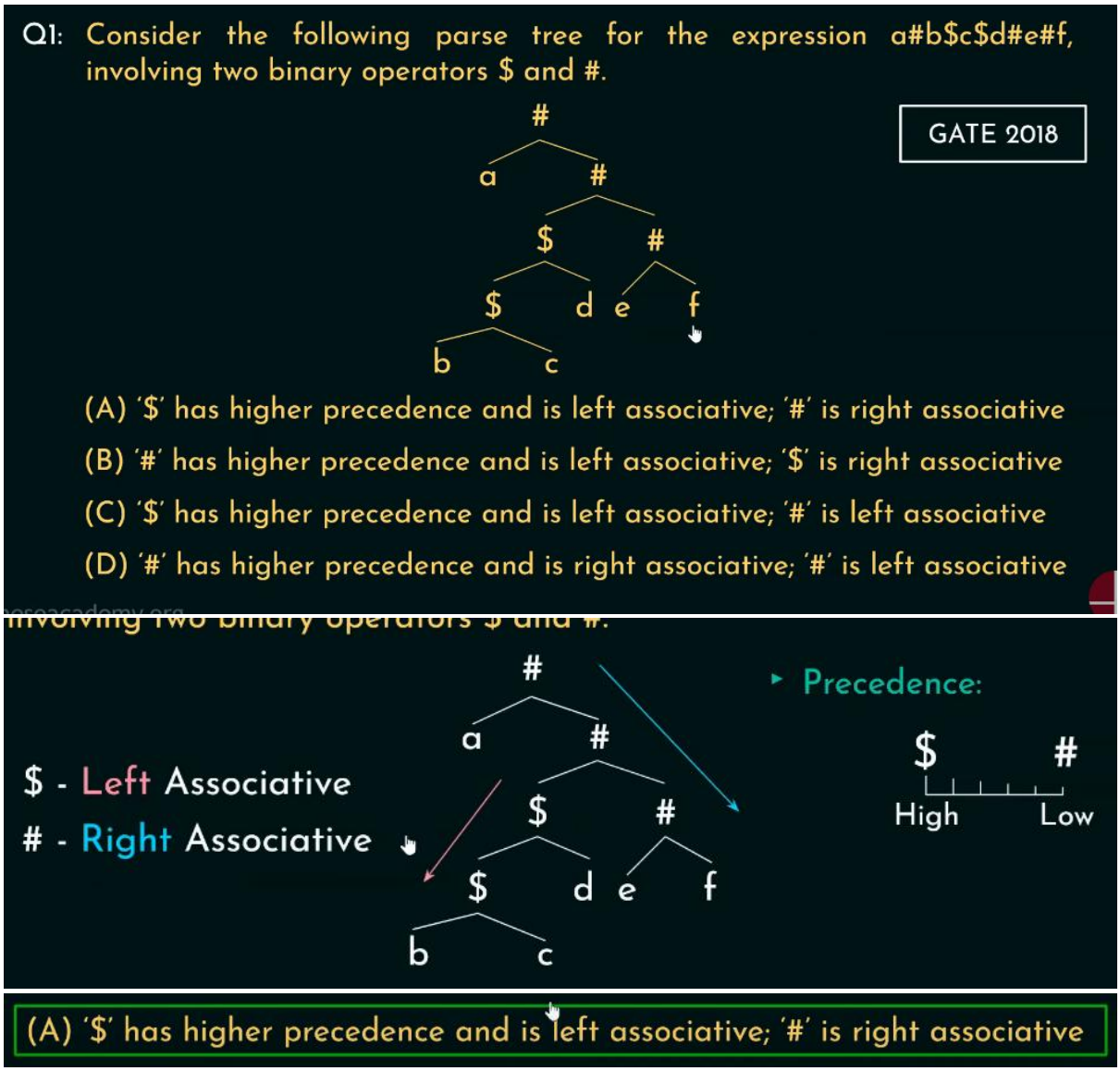
(A) 'x' has higher precedence than '+'
(B) '-' has higher precedence than 'x'
(C) '+' and '-' have same precedence
(D) '+' has higher precedence than 'x'

GATE 2000
UGC-NET Dec 2012
ISRO CS 2015

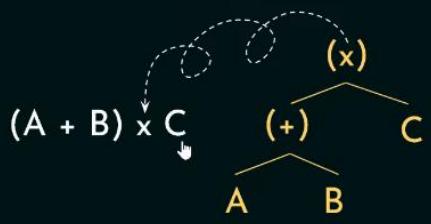
(A) 'x' has higher precedence than '+'
(B) '-' has higher precedence than 'x'
(C) '+' and '-' have same precedence
(D) '+' has higher precedence than 'x'

As x and + occur on the same row they have the same precedence and - has a higher precedence than x and +.

Associativity and Precedence in CFGs – Solved Problems (Set 2)



Q2: Which of the following expression is represented by the parse tree?



UGC-NET JUN 2010

(A) (A + B) x C

(B) A + x BC

(C) A + B x C

(D) A x C + B

(A) (A + B) x C

Q3: Given the grammar:

S → T x S I T
T → U + T I U
U → id

S → T x S
T → U + T

Right recursive

- Precedence:

+ x

High Low

ISRO CS 2020

Which of the following statements is wrong?

(A) Grammar is not ambiguous

(B) Priority of '+' over 'x' is ensured

(C) Right to left evaluation of 'x' and '+' happens

(D) None of these

Q4: Consider the following grammar (the start symbol is E) for generating expressions.

$E \rightarrow T - E \mid T + E \mid T$ — Right recursive

$T \rightarrow T \times F \mid F$ — Left recursive

$T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

TIFR PHD CS 2015

With respect to this grammar, which of the following tree is valid evaluation tree for the expression ((2 x 3) x 4) - ((5 x 6) + 7) ?

(A)

(B)

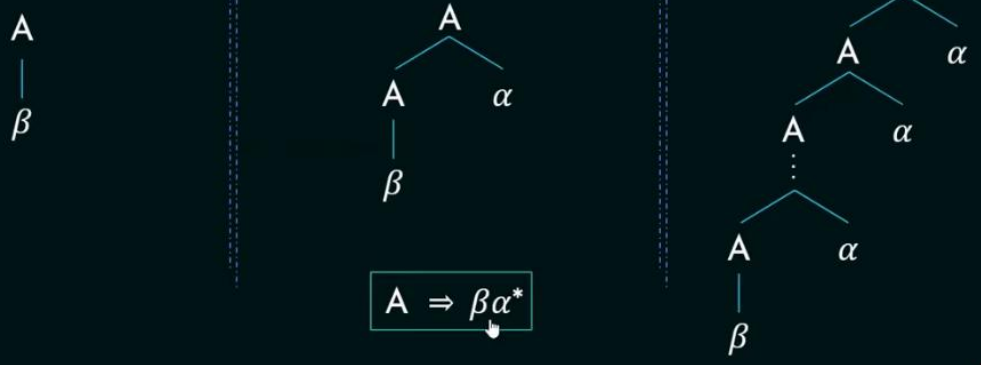
(C)

(D)

Recursion in Context Free Grammars (CFGs)

Left Recursion

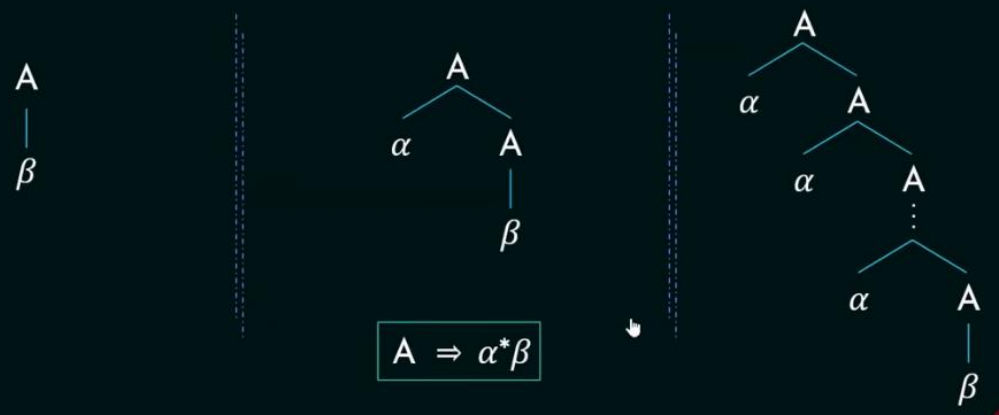
$A \rightarrow A\alpha \mid \beta$



$A \Rightarrow \beta\alpha^*$

Right Recursion

$A \rightarrow \alpha A \mid \beta$



$A \Rightarrow \alpha^* \beta$

Left Recursion

$A \rightarrow A\alpha \mid \beta$

$A \Rightarrow \beta\alpha^*$

```
A()
{
    A();
    α;
}
```

Right Recursion

$A \rightarrow \alpha A \mid \beta$

$A \Rightarrow \alpha^* \beta$

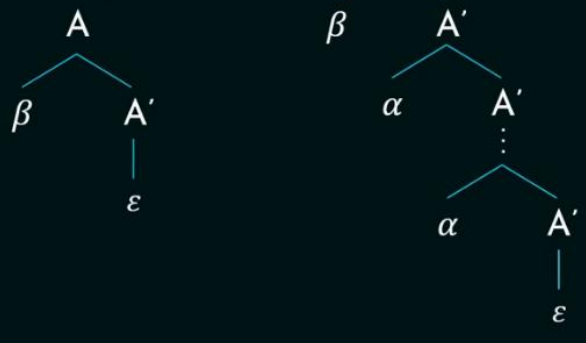
```
A()
{
    α;
    A();
}
```

Conversion

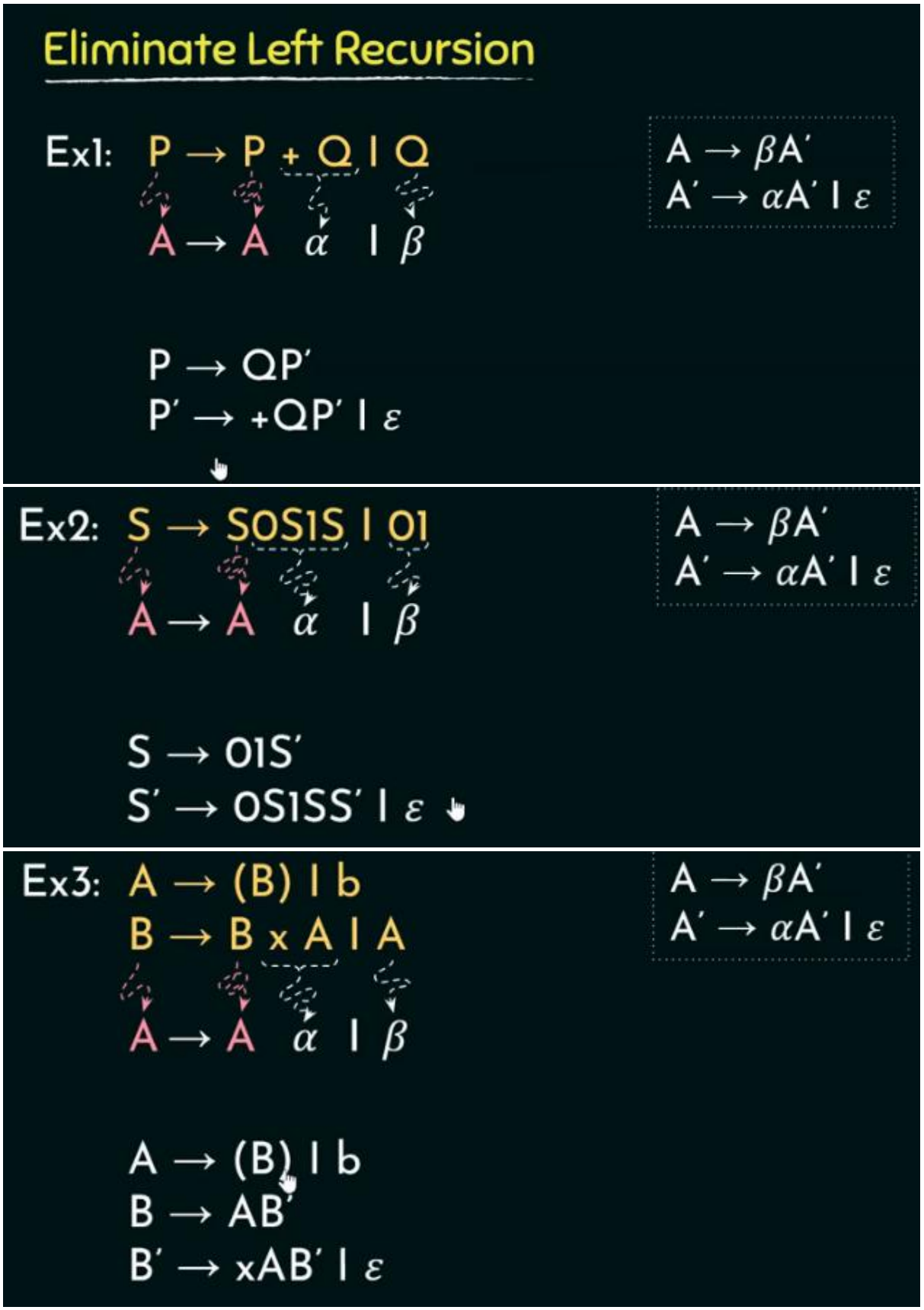
$A \rightarrow A\alpha \mid \beta$

$A \Rightarrow \beta\alpha^*$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$



Elimination of left recursion

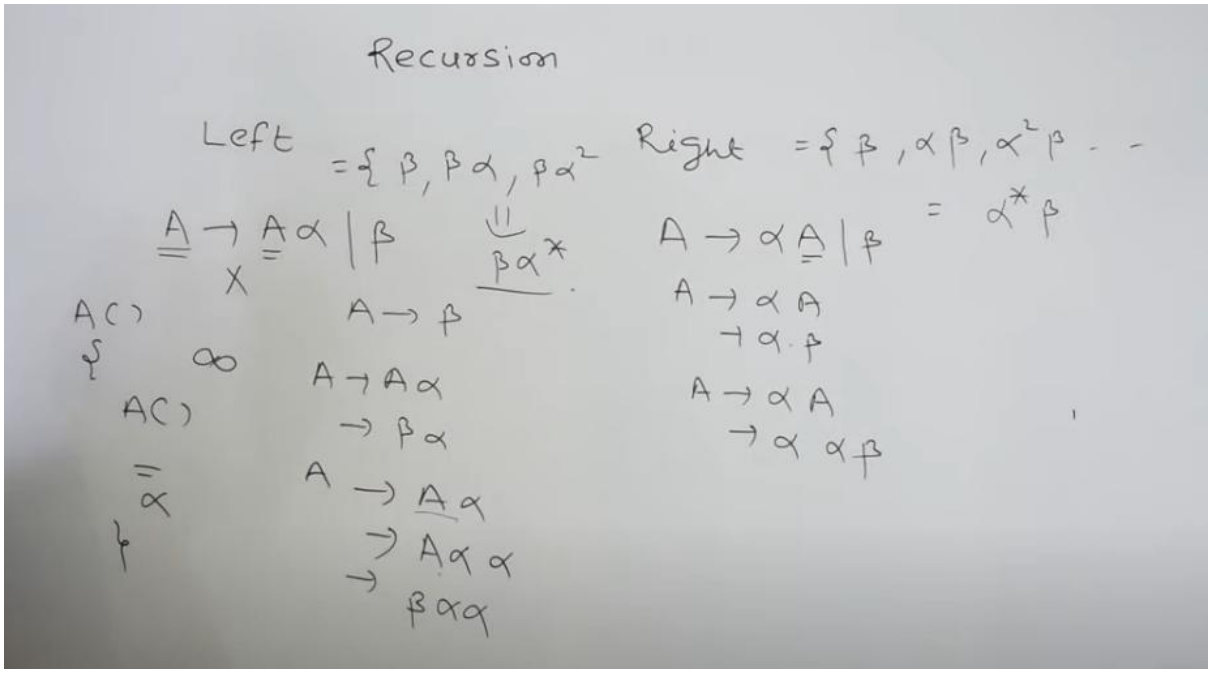


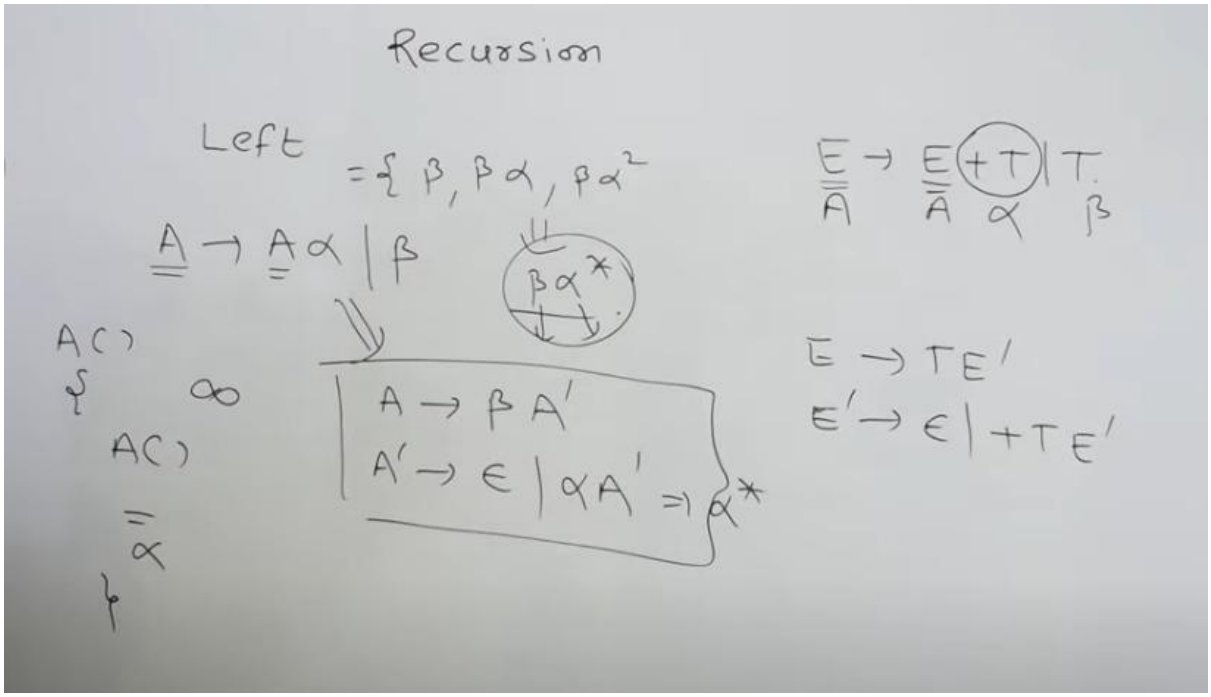
Eliminate Left Recursion

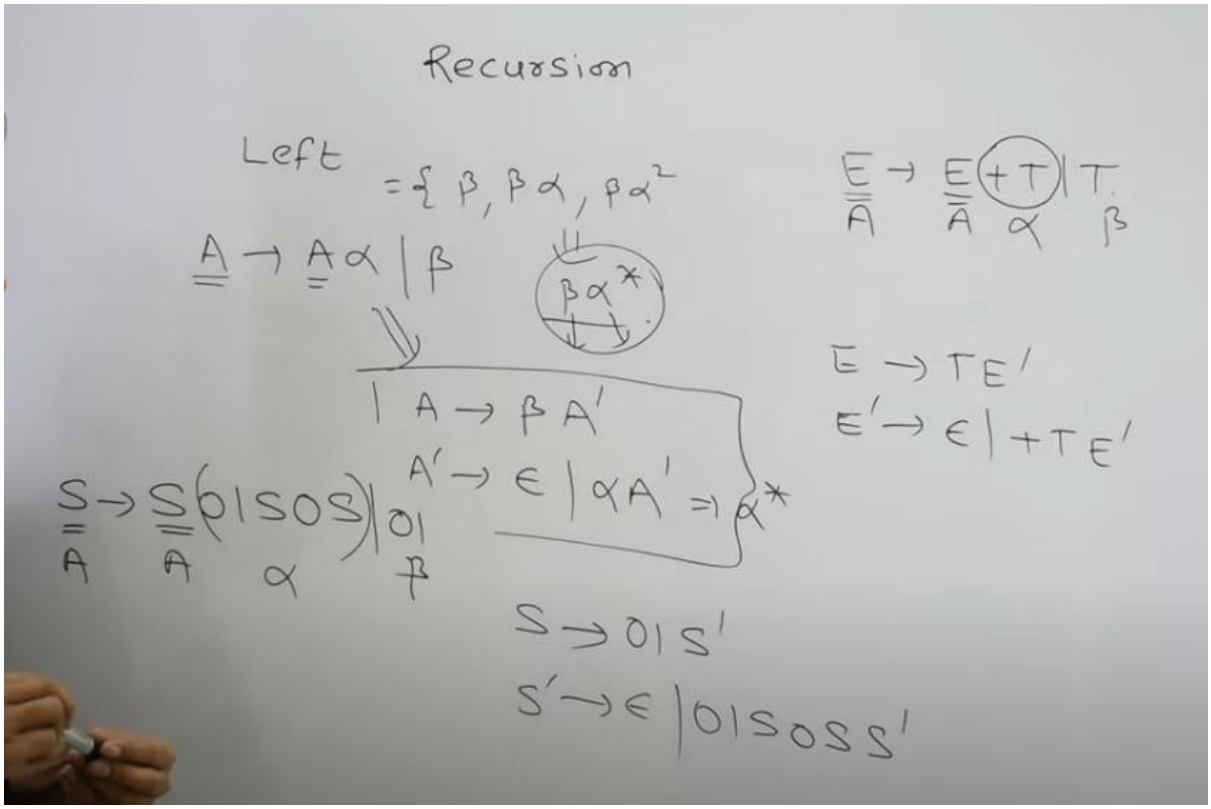
Ex4: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

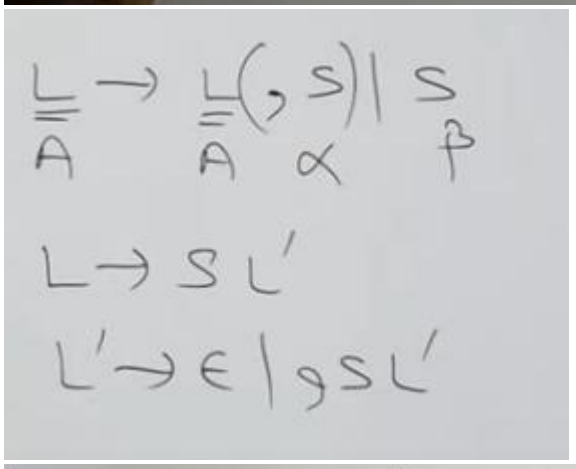
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

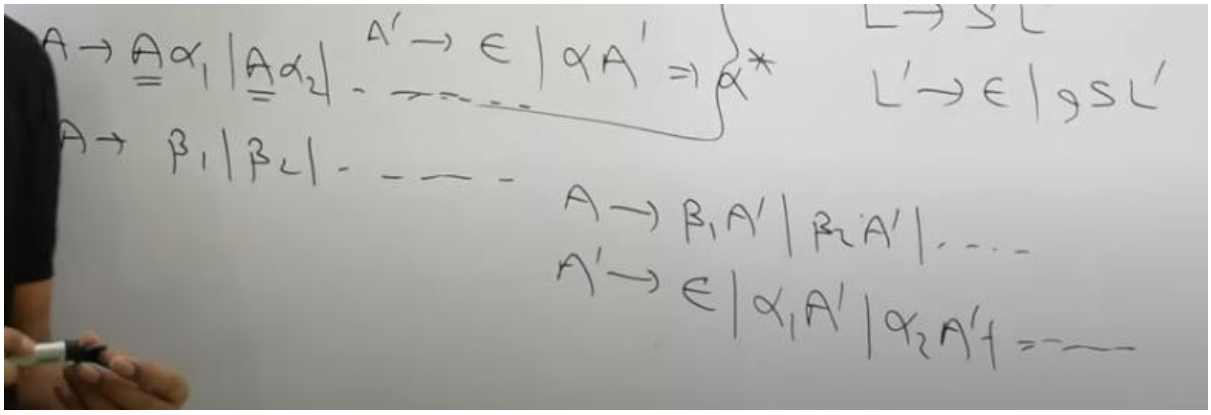
$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$











Precedence and Associativity of Operators Examples

Consider the grammar defined by the following production rules, with two operators * and +.

GATE-2014

$S \rightarrow T * P$

$T \rightarrow T \mid T * U$

$P \rightarrow Q + P \mid Q$

$Q \rightarrow id$

$U \rightarrow id$

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.

GATE-2014

R.G

which one of the following is TRUE?

X a) + is Left associative, while * is right associative

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

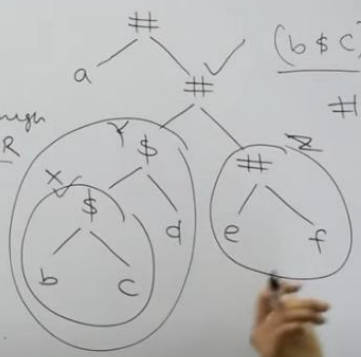
X c) Both + & * are right associative.

X d) Both + & * are left associative.

Consider the following parse tree for the expression

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

which of the following is correct for the given parse tree.

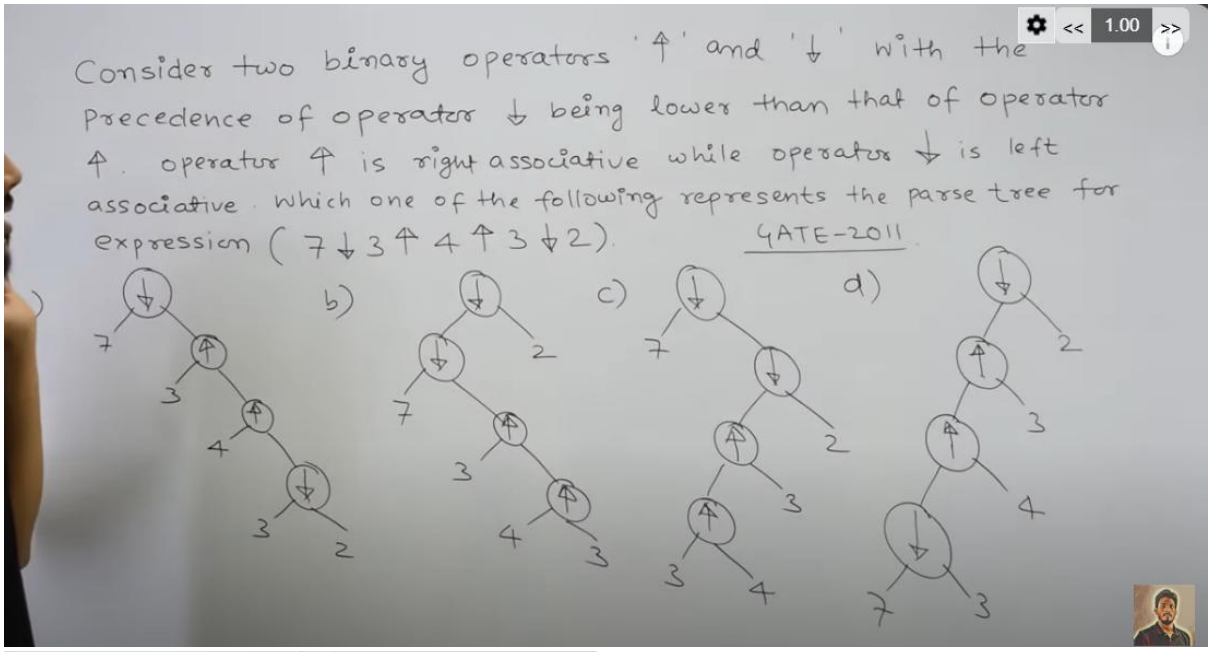


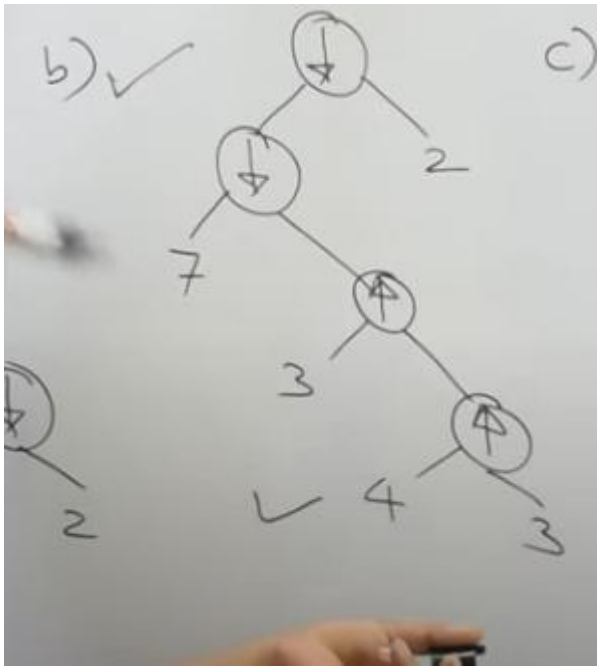
a) $ has higher precedence and is left associative, # is right associative.

X b) # has higher precedence and is left associative, $ is right associative.

X c) $ has higher precedence and is left associative, # is left associative.

X d) # has higher precedence and right associative, $ is left associative.

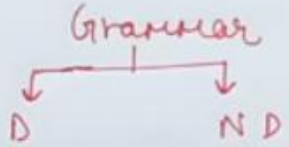




Elimination of left factoring

LEFT FACTORING

Sometimes, it is not clear which production to choose to expand a non-terminal blc multiple productions begin with the same (terminal/non terminal)/lookahead. This type of Grammar → Non Deterministic grammar, or Grammar containing left factoring



$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

common prefixes

suppose we have to accept $\alpha\beta_3$

backtracking happening blc of the common prefixes

accepted

One or more productions on the RHS are having something common in the prefixes

This is also called common prefix problem or non deterministic grammer.

α | γ backtracking happened?

blc we are making our decision only after seeing α.

ie

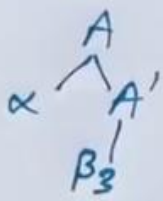
which production to choose without seeing full i/p.

if i/p is $\alpha\beta_3$

we should make the decision ki β_3 if available then only we will go into that production.

so, for

$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$ } Non Deterministic Grammar.



$A \to \alpha A'$
$A' \to \beta_1 / \beta_2 / \beta_3$ } we have postponed the decision making

Deterministic grammar

The procedure we used to convert Non deterministic grammar → Deterministic Grammar is called

Left Factoring.

Problem with Non Deterministic Grammar?

was most of the top down parser will have to **backtrack**.

So, we donot need Non-deterministic grammar.

(i) $S \to iEtSeS /$
$iEtS / a$
$E \to b$

soln →

$S \to iEtSS' / a$
$S' \to eS / \in$
$E \to b$ }

(ii) $S \to aSSbS \mid aSasb \mid abb \mid b$

$A \to aA$, $B \to aB$ } not a common prefix problem

$S \to aS' \mid b$

$S' \to SSbS \mid Sasb \mid bb$ } again common prefixes

$S' \to SS'' \mid bb$

$S'' \to SbS \mid aSb$

Left factoring

NP ⟹ Det

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

Common prefix

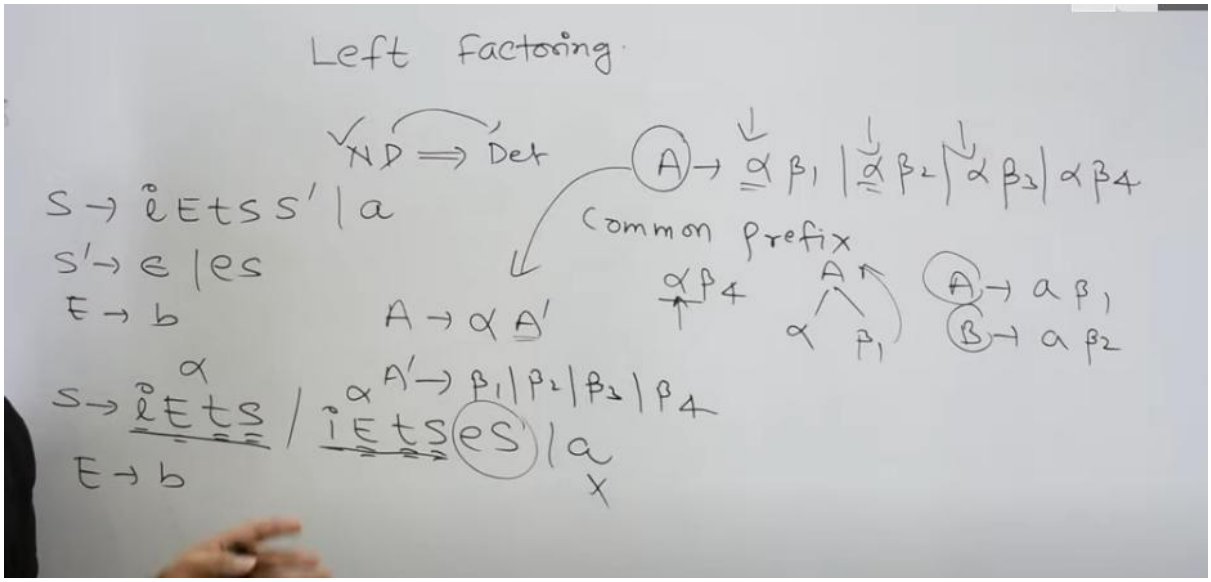
$\alpha\beta_4$

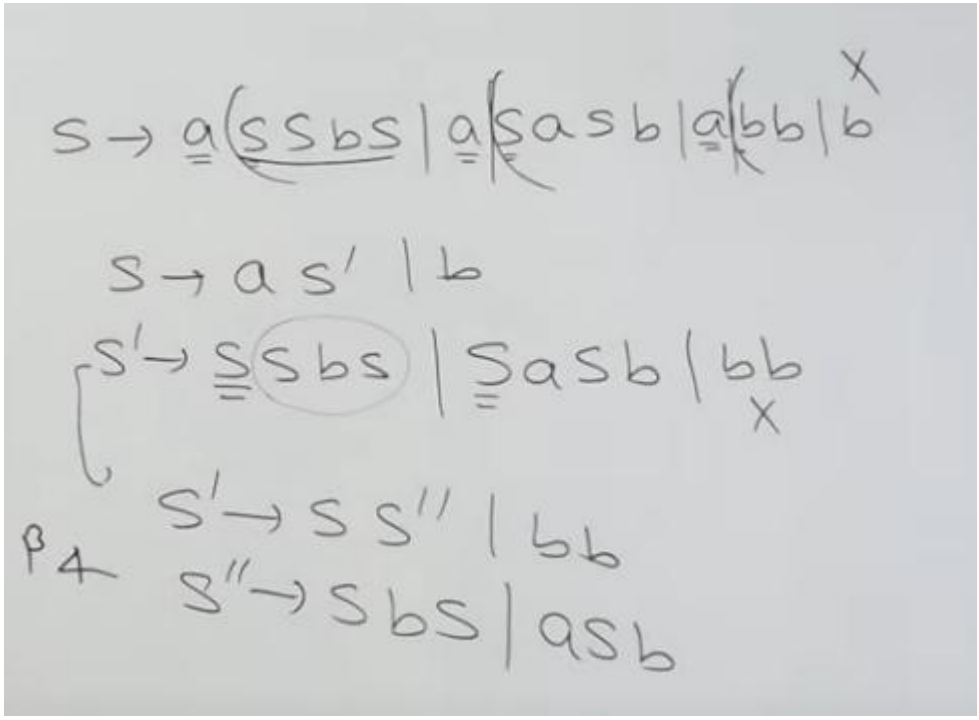
$A \to \alpha A'$

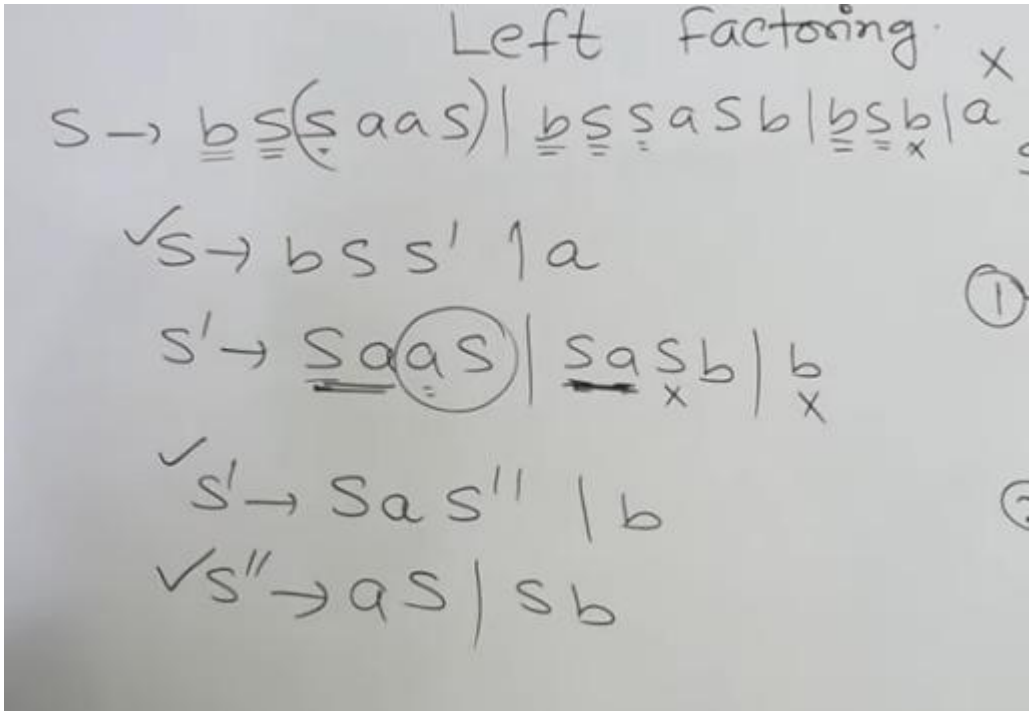
$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4$

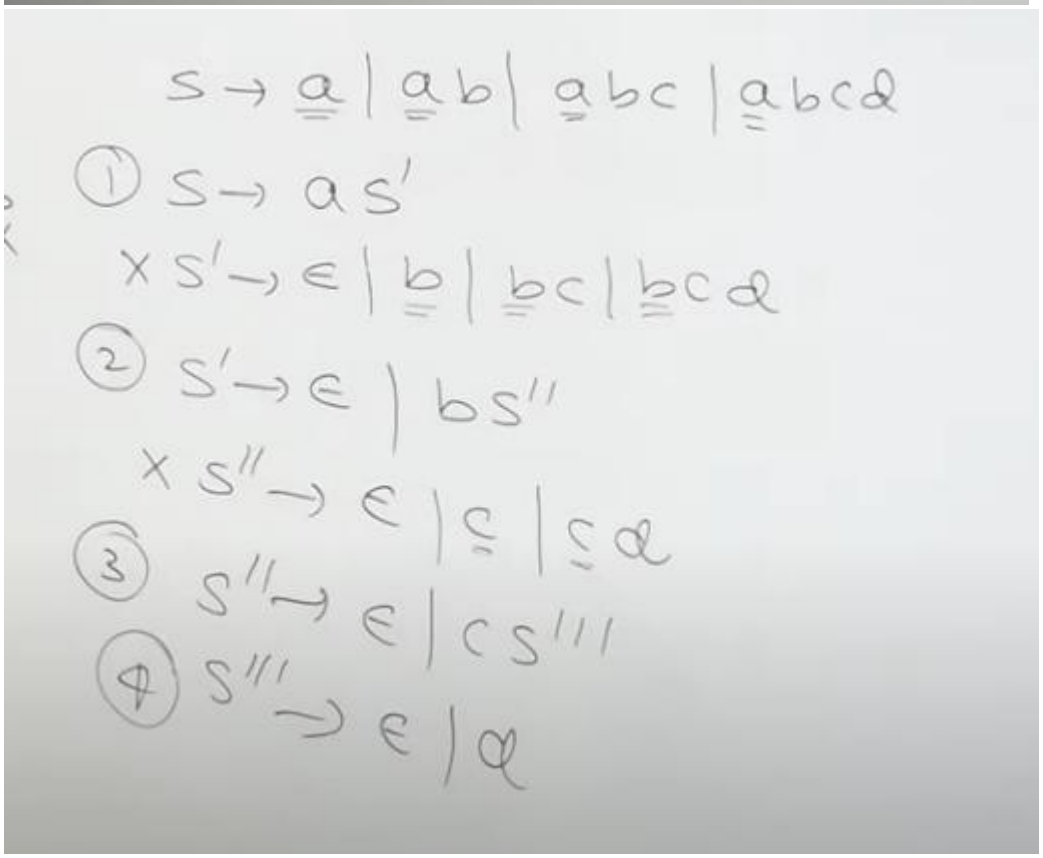
$A \to a\beta_1$

$B \to a\beta_2$

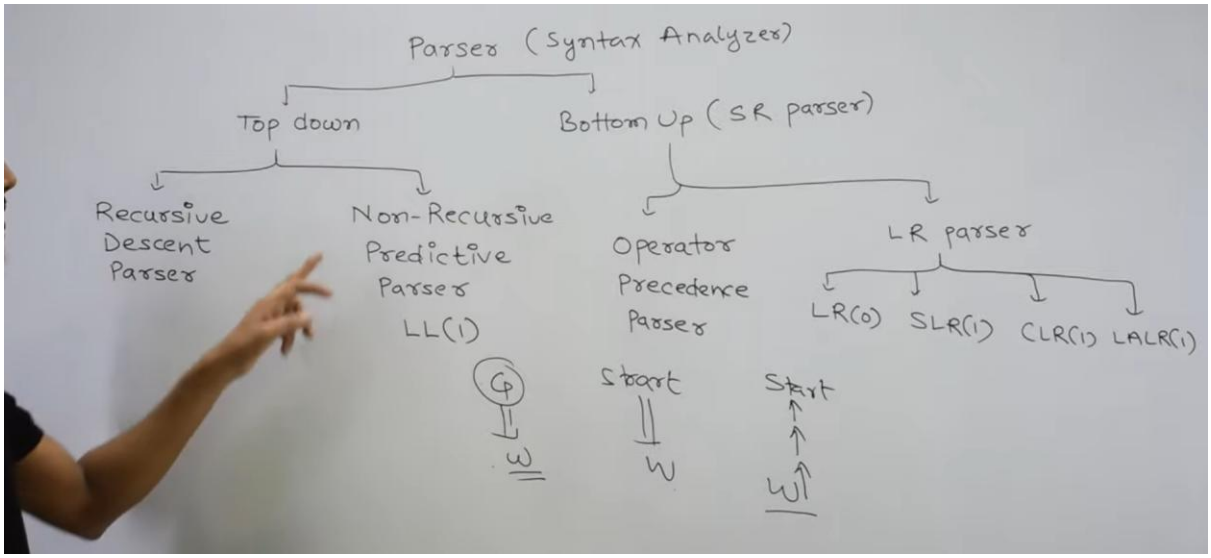

Left factoring
NP ⟹ Det
S → iEtSS' | a
S' → ∈ | eS
E → b
A → αβ1 | αβ2 | αβ3 | αβ4
Common prefix
A → αA'
A' → β1 | β2 | β3 | β4
S → iEtS | iEtSeS | a
E → b


S → aSSbS | aSaSb | abb | b
S → aS' | b
S' → SSbS | SaSb | bb
S' → SS'' | bb
S'' → SbS | aSb

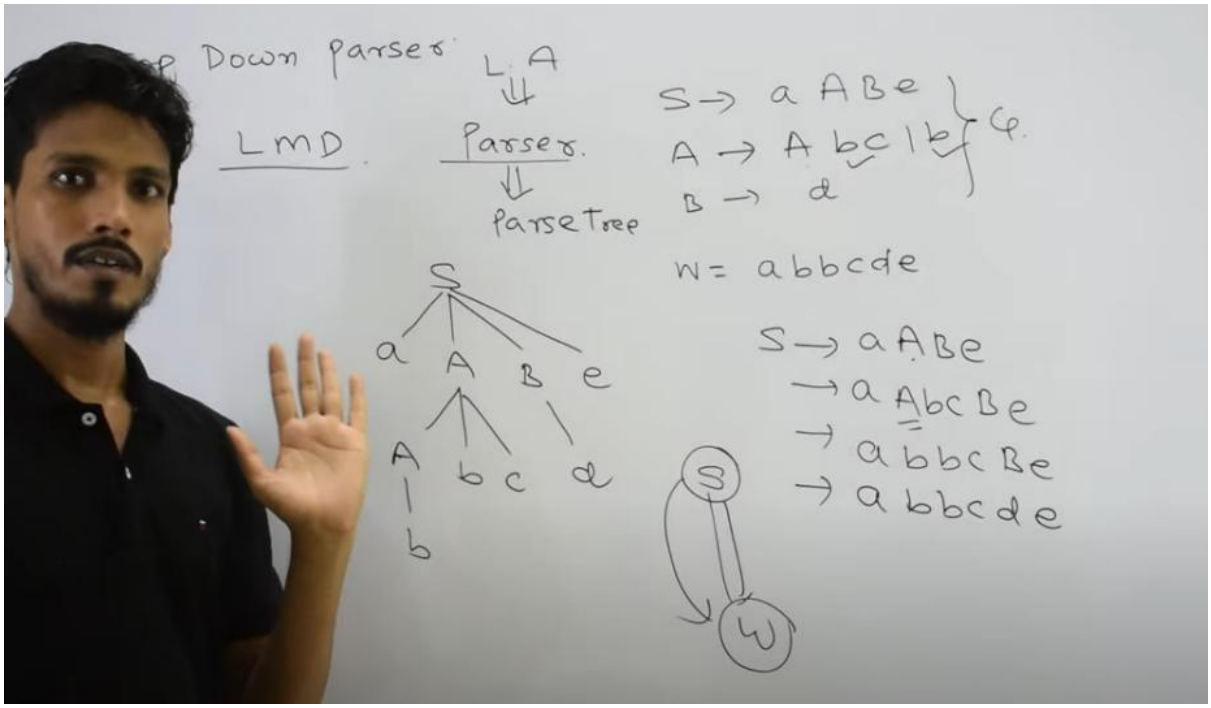


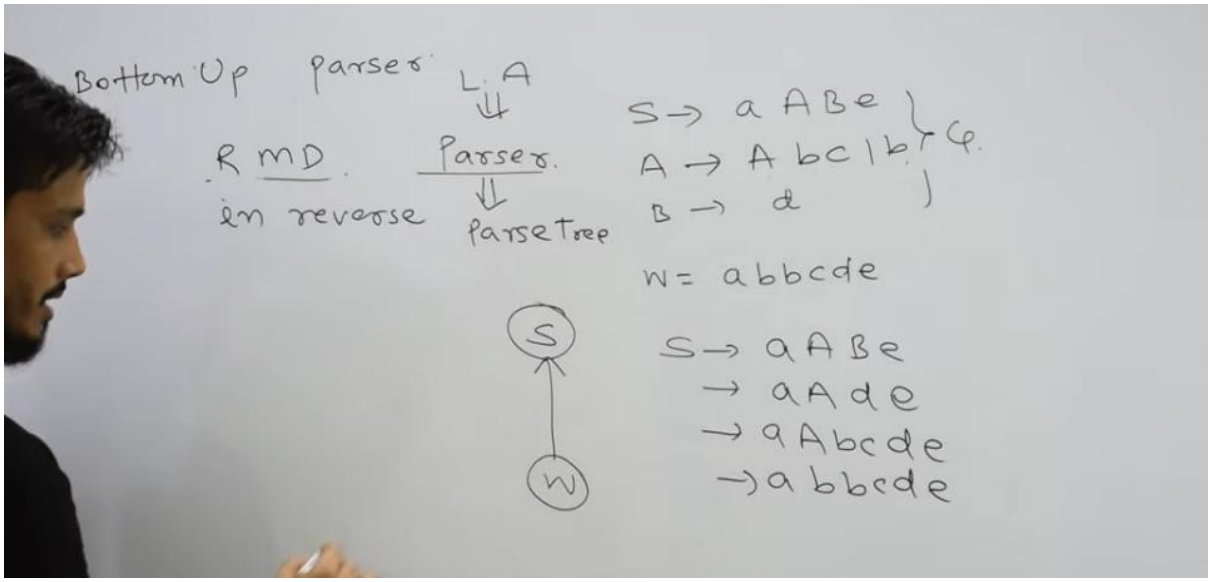


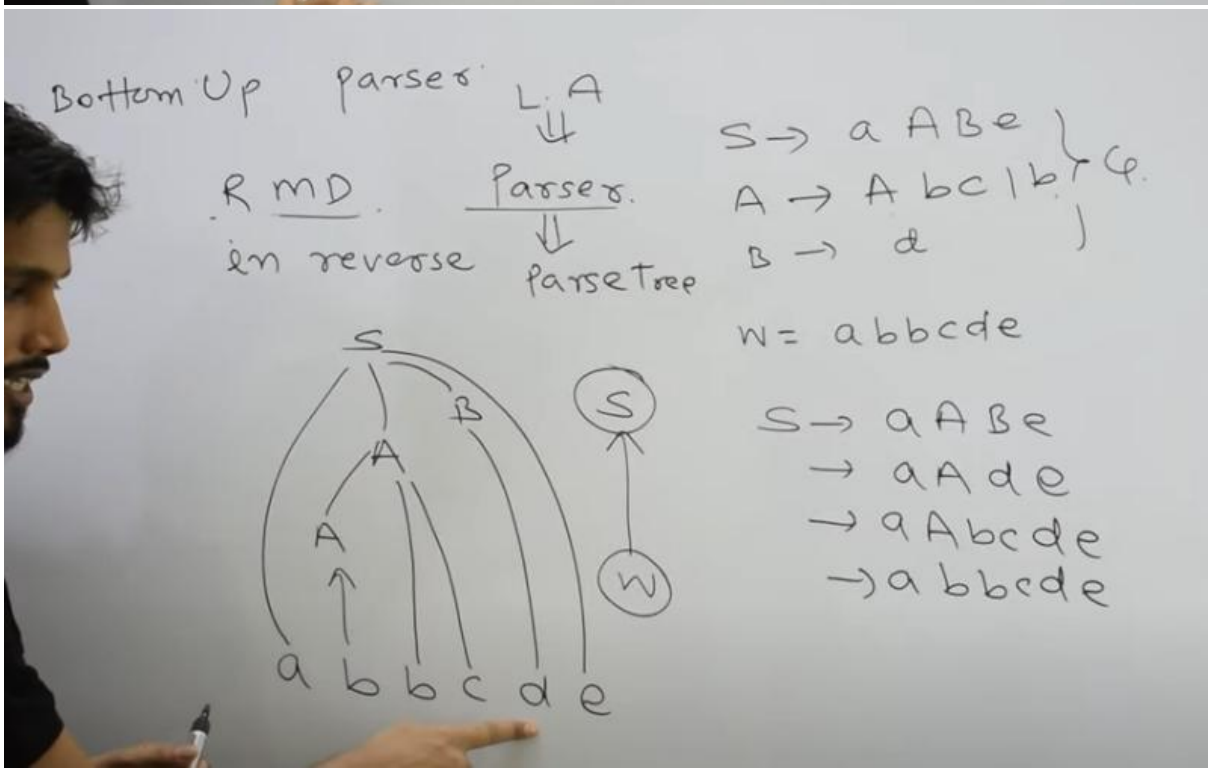
Introduction to Parser | Top Down and Bottom Up Parser



Top Down Parser





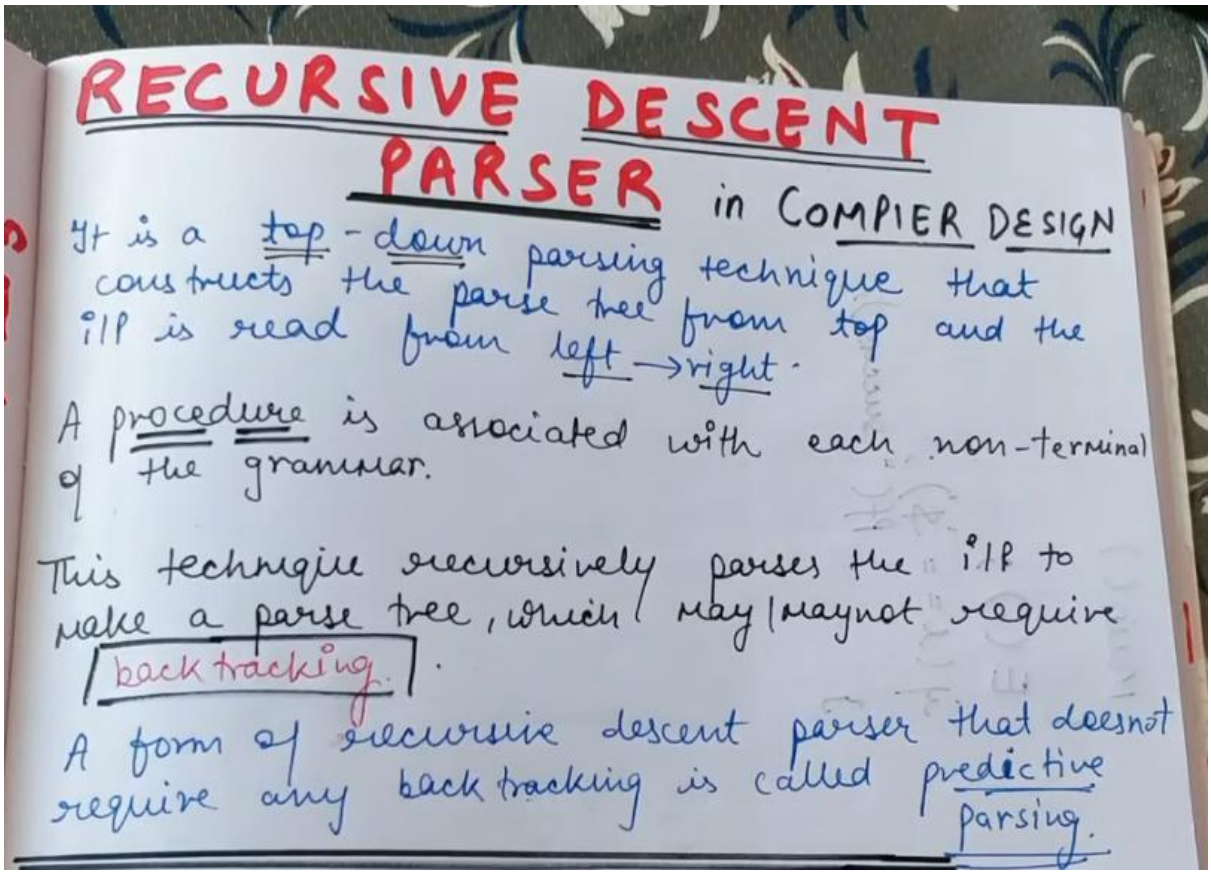


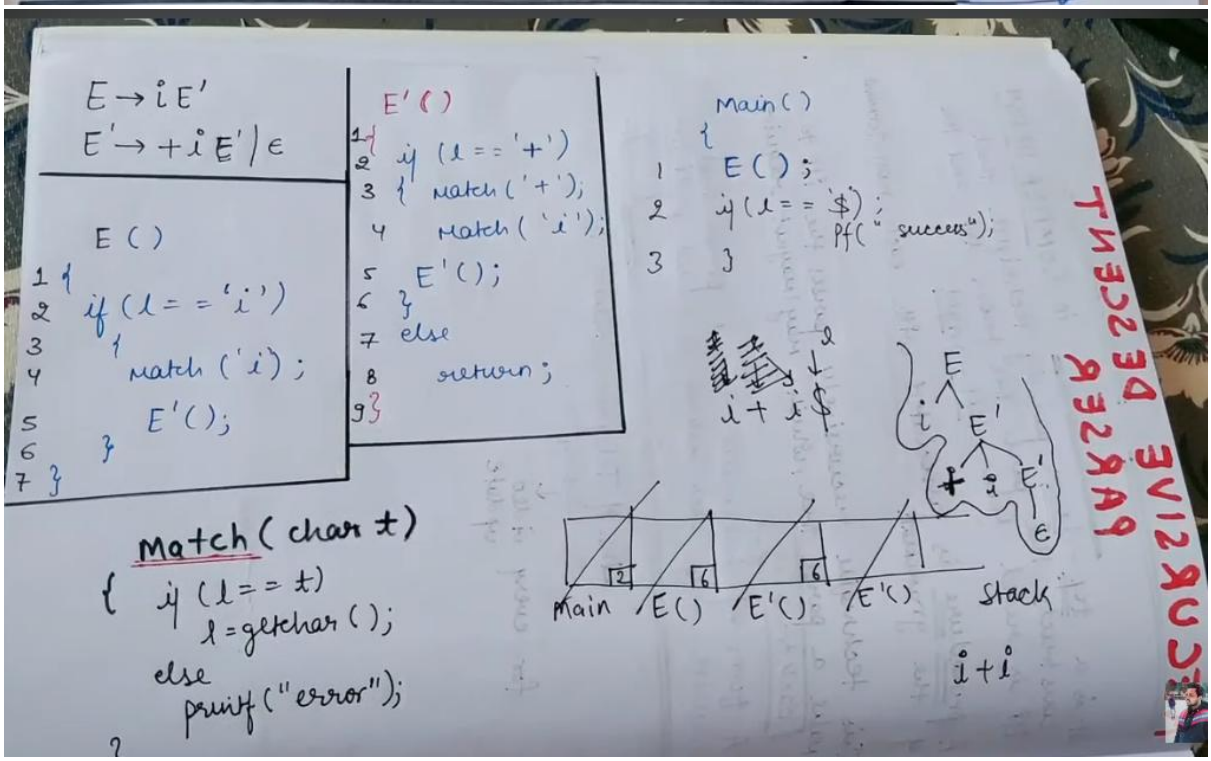
Recursive Descent Parser with solved example

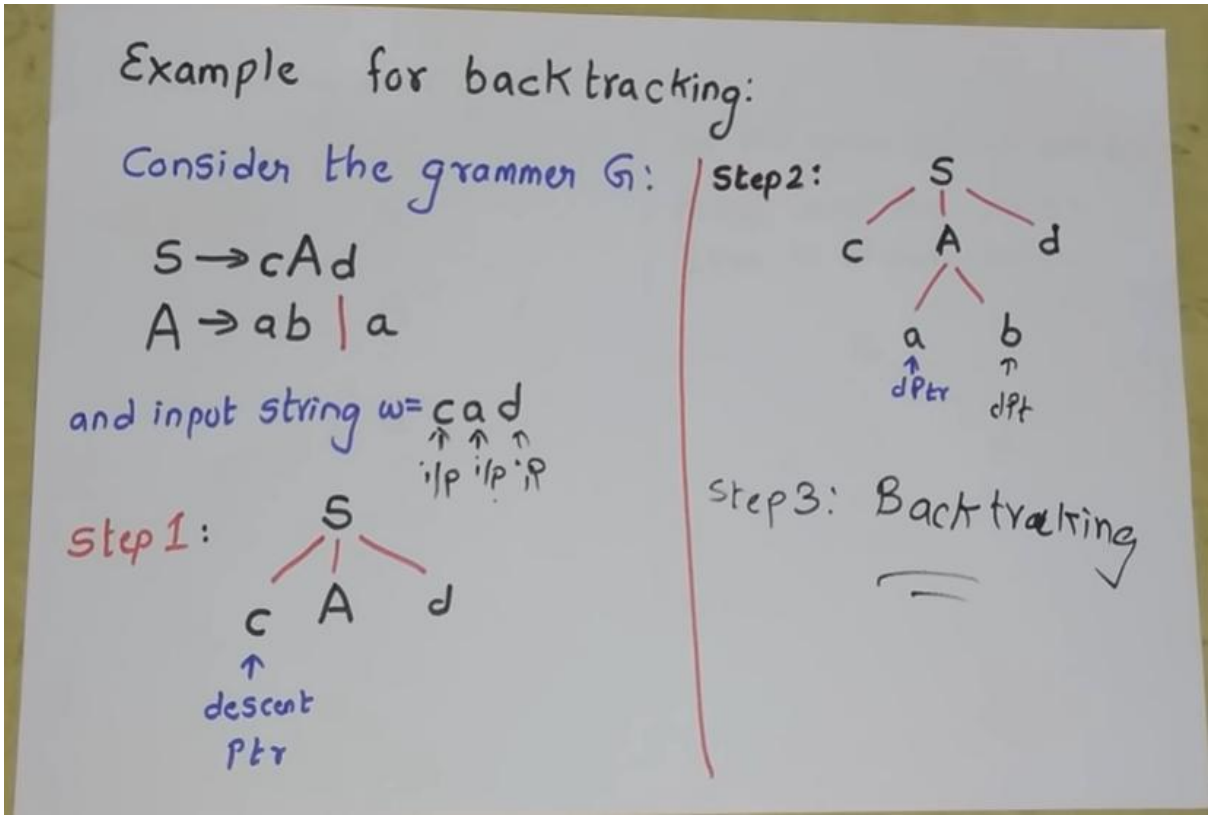
A Recursive Descent Parser has difficulty with left recursive grammars and non-deterministic grammars.

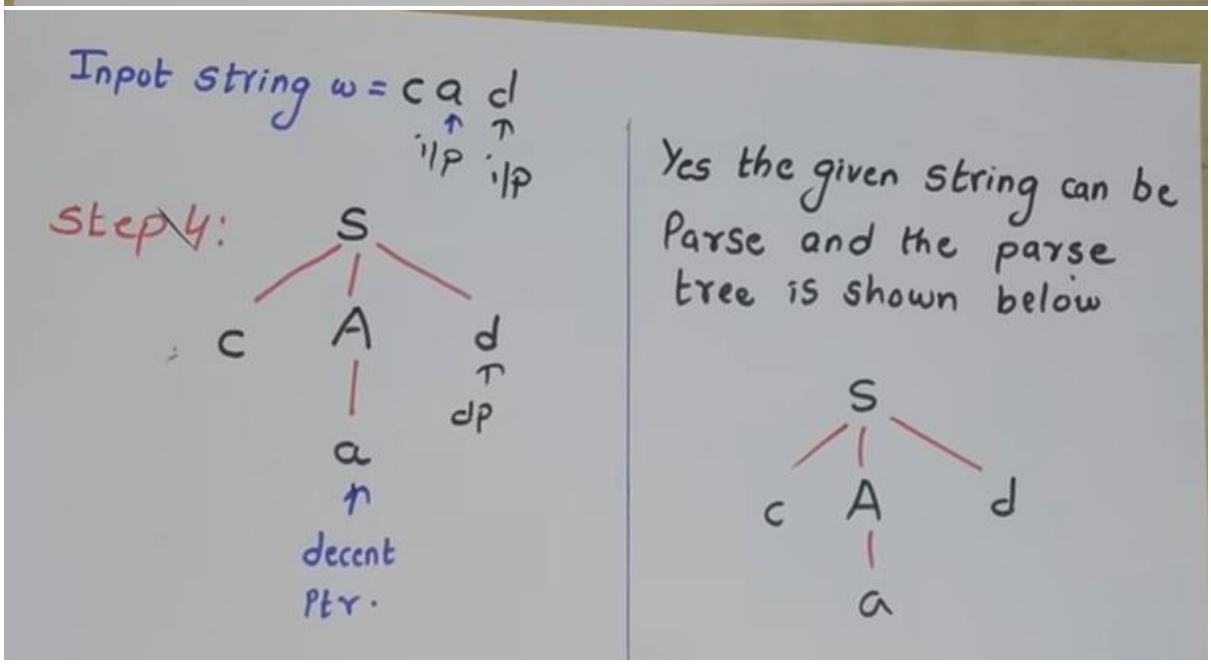
- For **left recursive grammars**, the parser would enter into an **infinite loop**, so the grammar needs to be rewritten to eliminate left recursion.
- For **non-deterministic grammars**, while **it's not impossible**, it can be **challenging** for a Recursive Descent Parser. It depends on whether an LL(k)

grammar (which can be parsed by a Recursive Descent Parser) exists for the language.



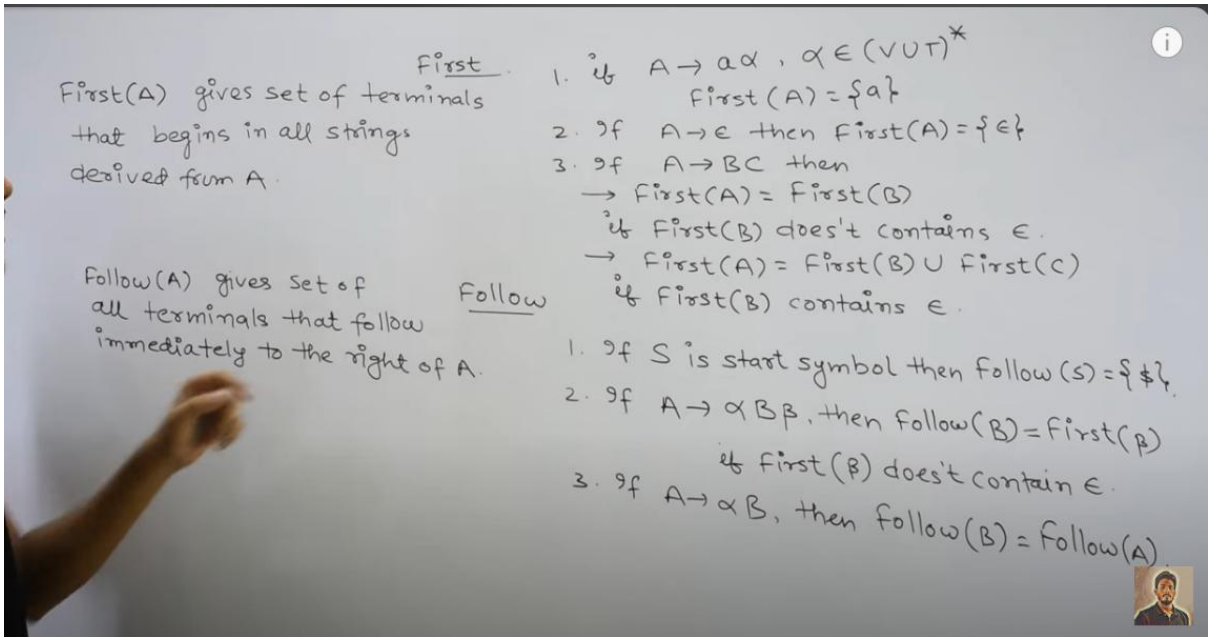


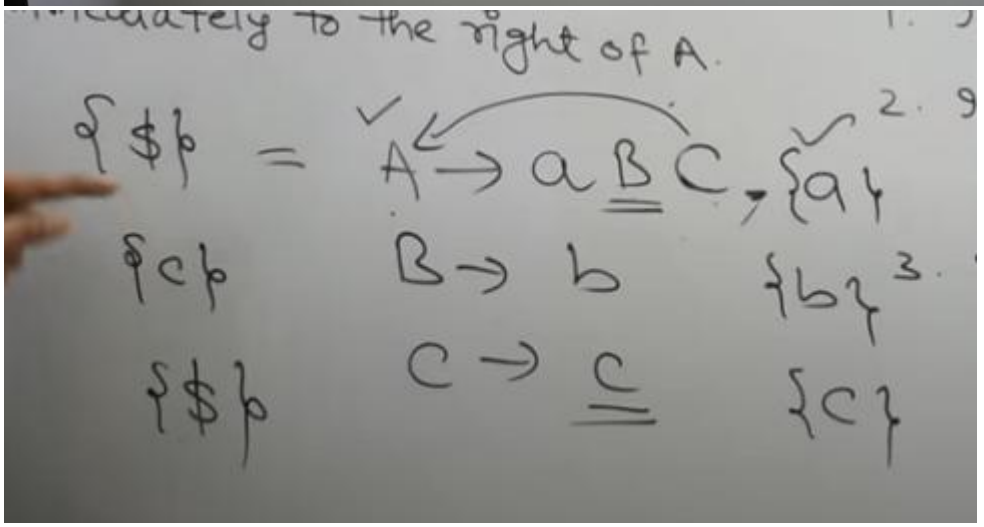




How to Find First and Follow

https://www.gatevidyalay.com/first-and-follow-compiler-design/
https://www.codingninjas.com/studio/library/first-and-follow-in-compiler-design





First and Follow Examples

Computation of FIRST

FIRST (α) is defined as the collection of terminal symbols which are the first letters of strings derived from α.

FIRST (α) = {α |α →∗ αβ for some string β }

If X is Grammar Symbol, then First (X) will be −

If X is a terminal symbol, then FIRST(X) = {X}

If X → ε, then FIRST(X) = {ε}

If X is non-terminal & $X \rightarrow a\ \alpha$, then FIRST (X) = {a}

If $X \rightarrow Y_1, Y_2, Y_3$, then FIRST (X) will be

(a) If Y is terminal, then

$$FIRST(X) = FIRST(Y_1, Y_2, Y_3) = \{Y_1\}$$

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then, $FIRST(X) = FIRST(Y_1, Y_2, Y_3) = FIRST(Y_1)$

(c) If FIRST (Y_1) contains ε, then.

$$FIRST(X) = FIRST(Y_1, Y_2, Y_3) = FIRST(Y_1) - \{\varepsilon\} \cup FIRST(Y_2, Y_3)$$

Similarly, $FIRST(Y_2, Y_3) = \{Y_2\}$, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

$FIRST(Y_2, Y_3) = FIRST(Y_2)$, if FIRST ($Y_2$) does not contain ε.

If FIRST (Y_2) contain ε, then

$FIRST(Y_2, Y_3) = FIRST(Y_2) - \{\varepsilon\} \cup FIRST(Y_3)$

Similarly, this method will be repeated for further Grammar symbols, i.e., for $Y_4, Y_5, Y_6 \ldots . Y_K$.

If Yk contains empty string then empty string would be there in the first.

Computation of FOLLOW

Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

$$FOLLOW(A) = \{a | S \Rightarrow^* \alpha A a \beta \text{ where } \alpha, \beta \text{ can be any strings}\}$$

Rules to find FOLLOW

If S is the start symbol, FOLLOW (S) ={$}

If production is of form $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$.

(a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

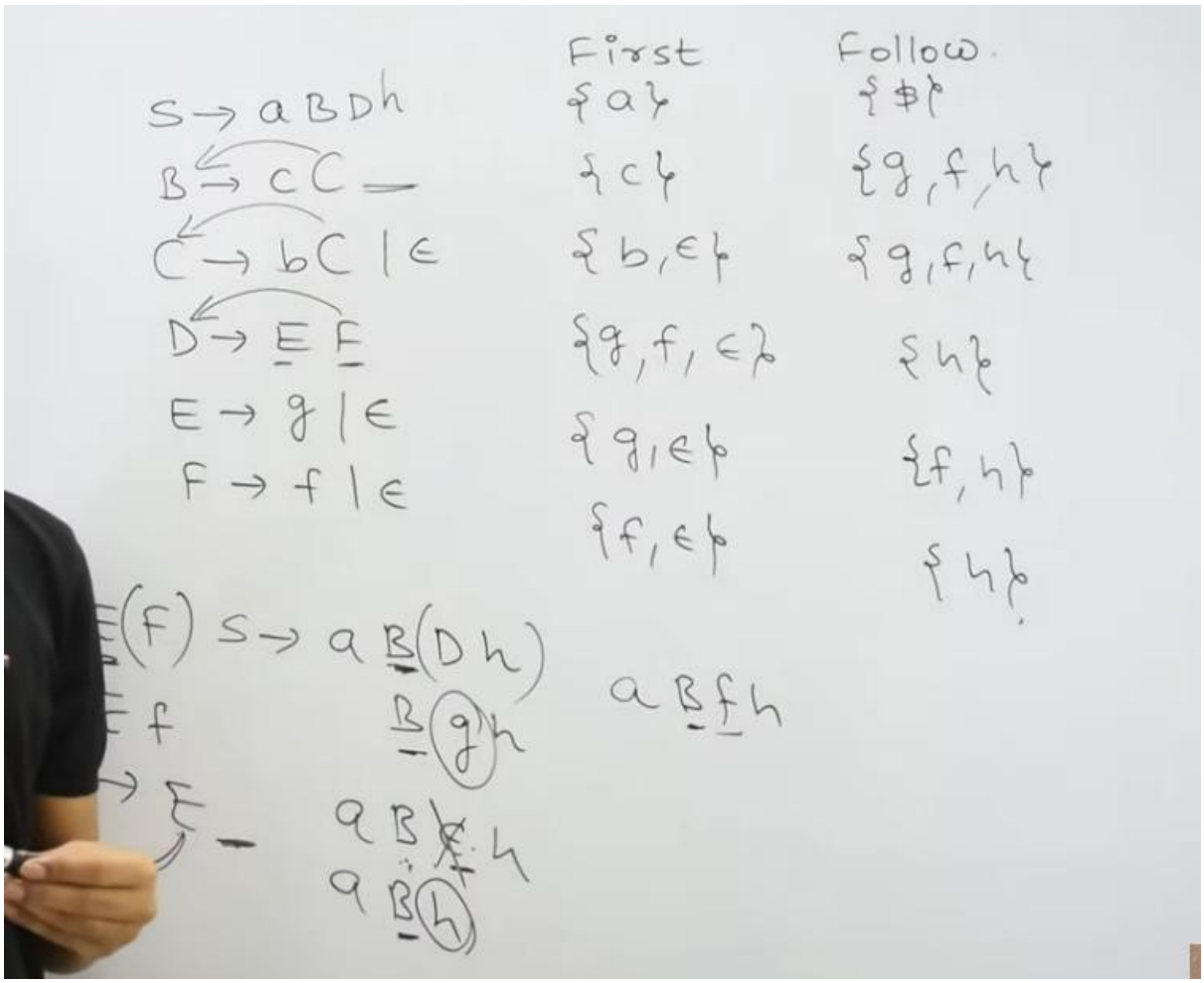
Or

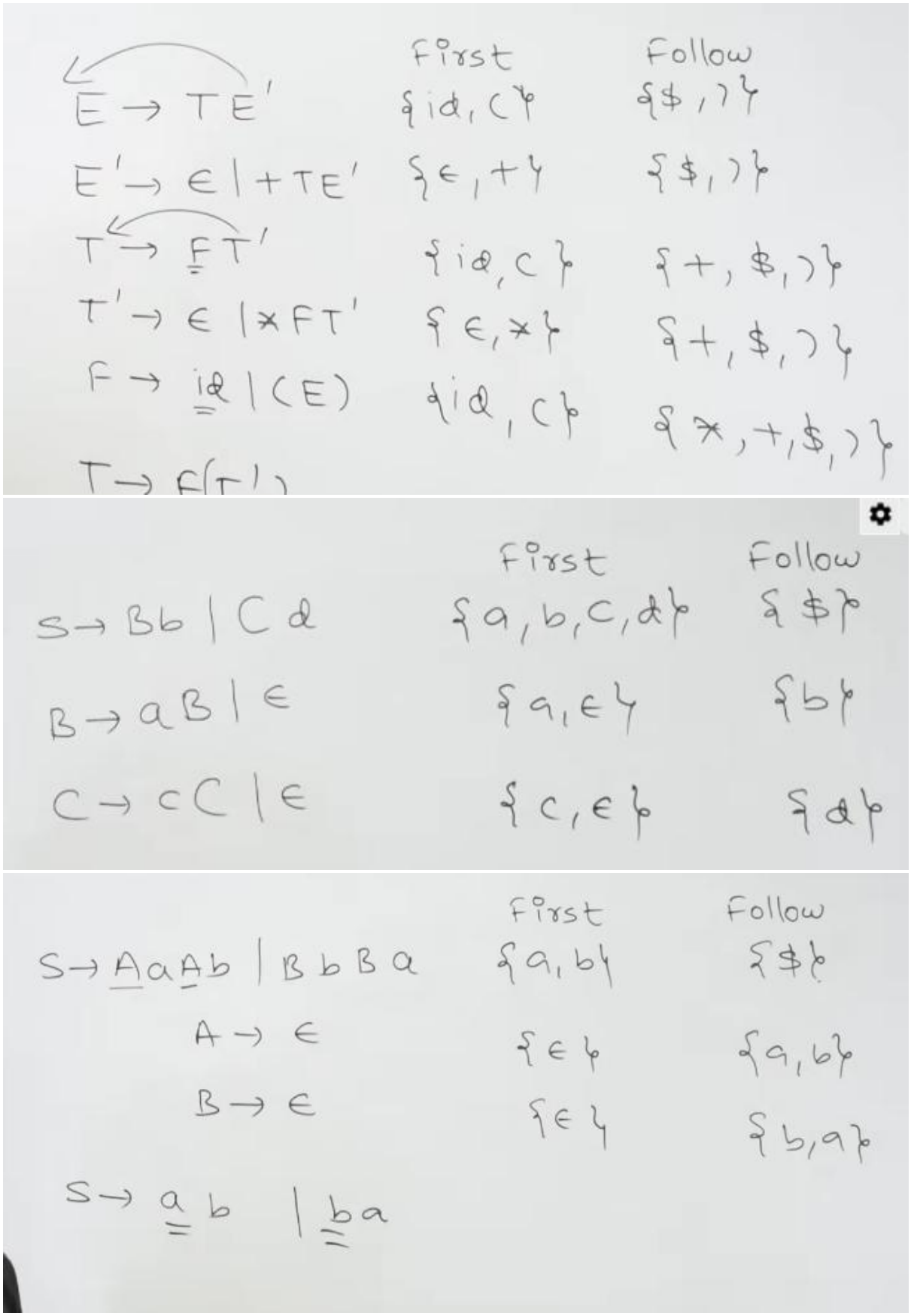
(b) If FIRST (β) contains ε (i. e. , $\beta \Rightarrow^* \varepsilon$), then

FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)

∵ when β derives ε, then terminal after A will follow B.

If production is of form $A \rightarrow \alpha B$, then Follow (B) ={FOLLOW (A)}.

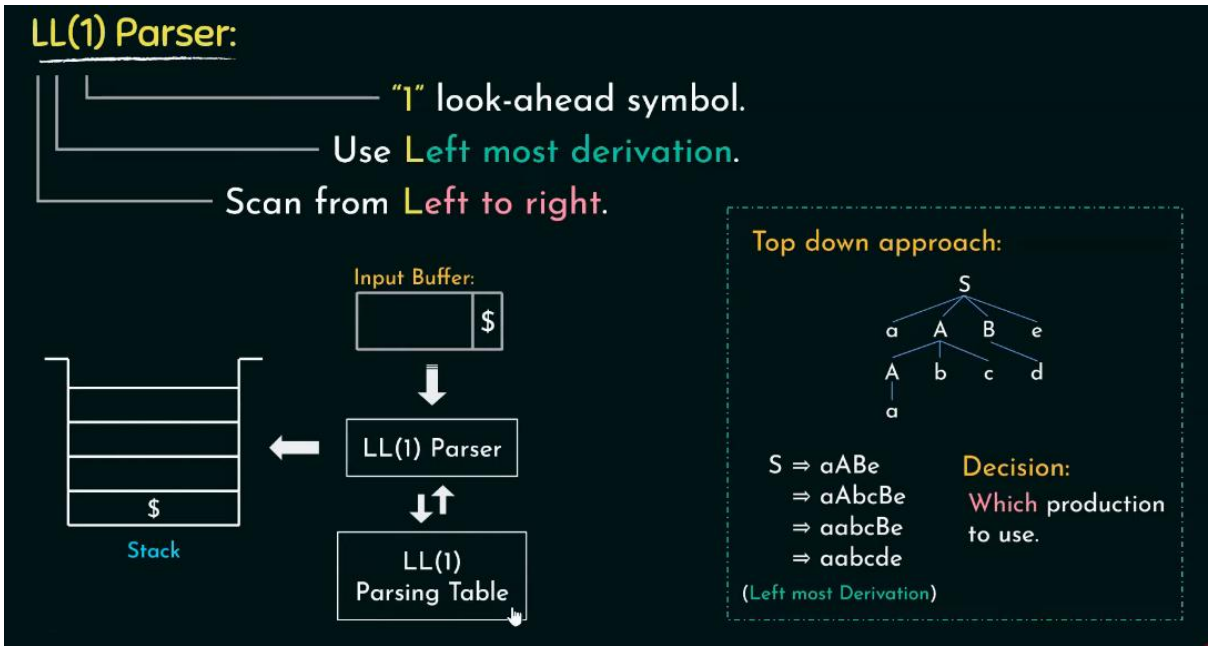


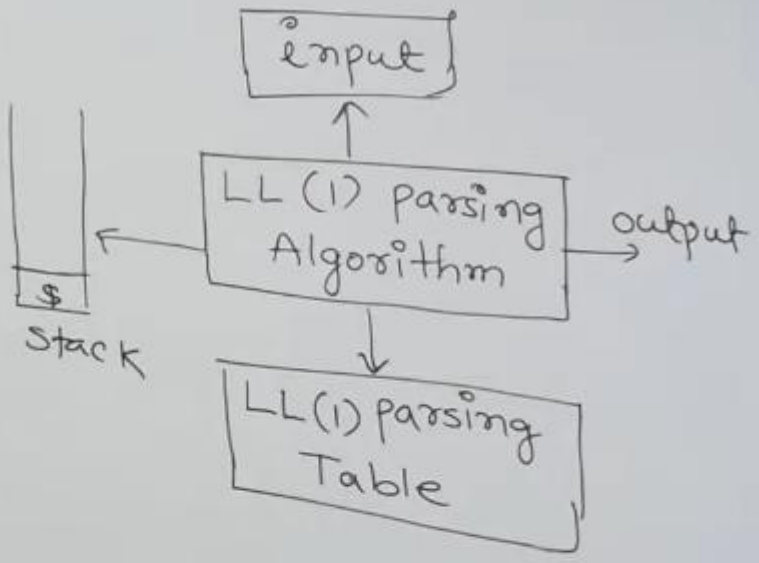

First
Follow
E → TE'
{id, (}
{$,)}
E' → ∈ | +TE'
{∈, +}
{$,)}
T → FT'
{id, (}
{+, $,)}
T' → ∈ | *FT'
{∈, *}
{+, $,)}
F → id | (E)
{id, (}
{*, +, $,)}
S → Bb | Cd
{a, b, c, d}
{$}
B → aB | ∈
{a, ∈}
{b}
C → cC | ∈
{c, ∈}
{d}
S → AaAb | BbBa
{a, b}
{$}
A → ∈
{∈}
{a, b}
B → ∈
{∈}
{b, a}
S → ab | ba

LL(1) parser

https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/

The given grammar must **not be ambiguous**
It should **not be left recursive** or **non-deterministic and no left-factoring**





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 $\in$.

Predective par

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

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

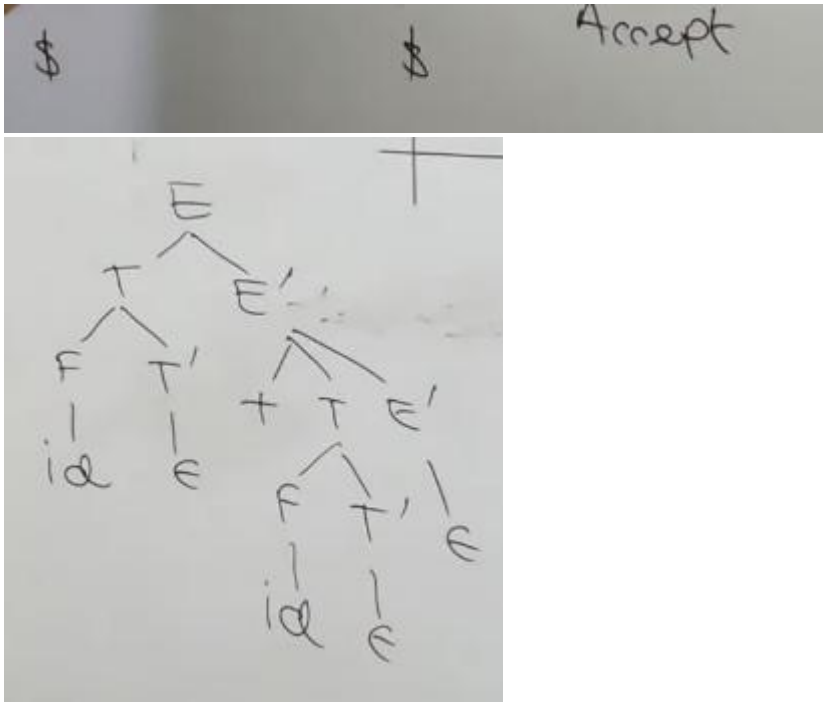
1. Add $A \to \alpha$ under $M[A, a]$ where $a \in first(\alpha)$
2. Add $A \to \alpha$ under $M[A, a]$ where $a \in follow(A)$ if $first(\alpha)$ contain ϵ

Non Recursive Descent parser.

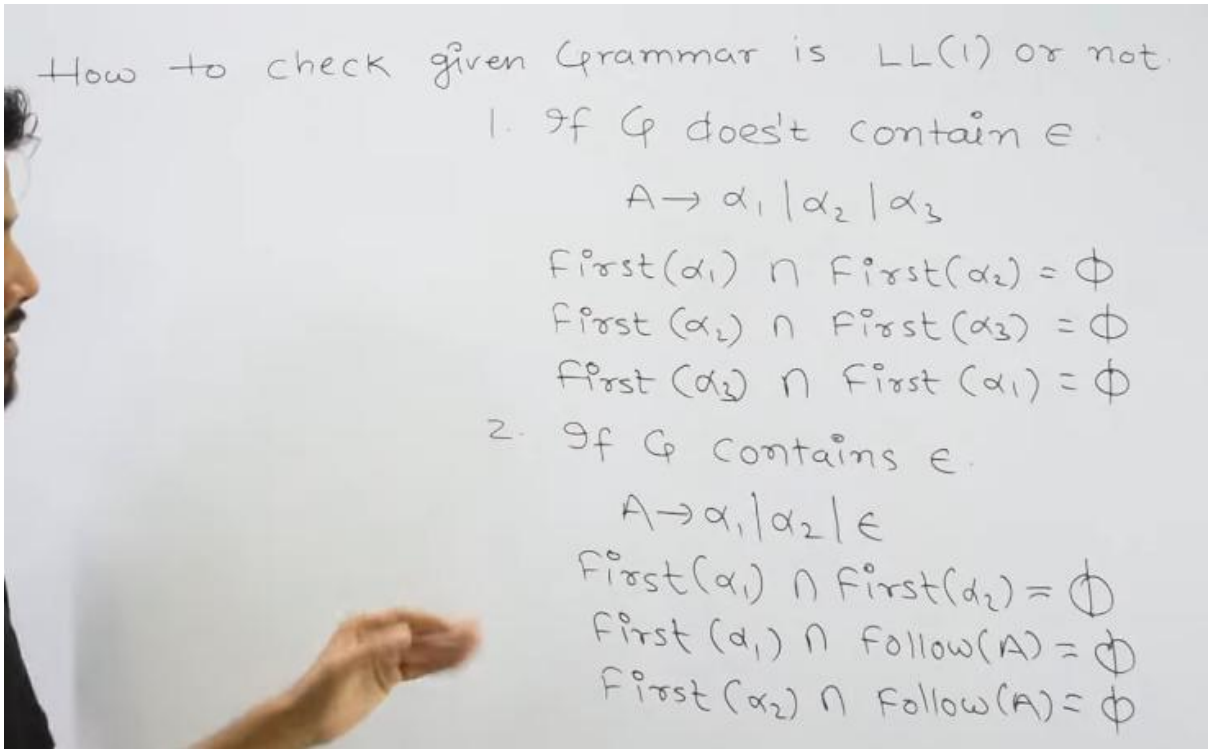
Predective parser or LL(1)

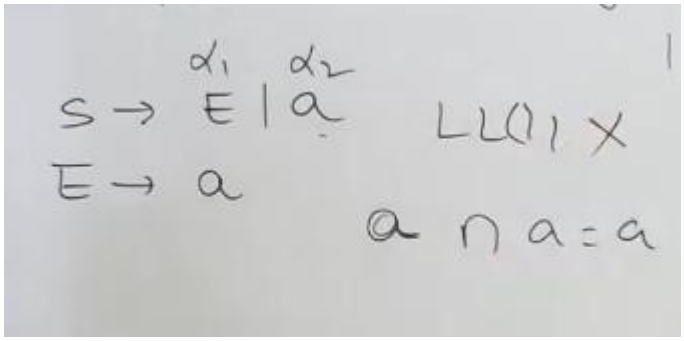
Stack	input	production.
$E	id+id$	$E \to TE'$
$E'T	id+id$	$T \to FT'$
$E'T'F	id+id$	$F \to id$
$E'T'id	id+id$	pop
$E'T'	+id$	$T' \to \epsilon$
$E'	+id$	$E' \to +TE'$
$E'T+	+id$	pop
$E'T	id$	$T \to FT'$
$E'T'F	id$	$F \to id$
$E'T'id	id$	pop
$E'T'	$	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	

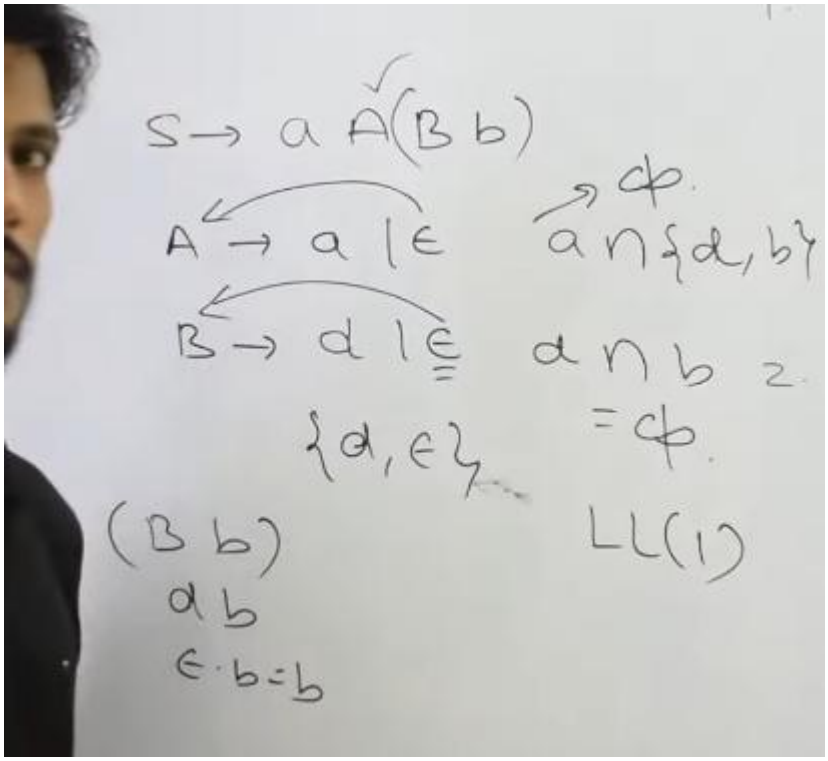
M	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

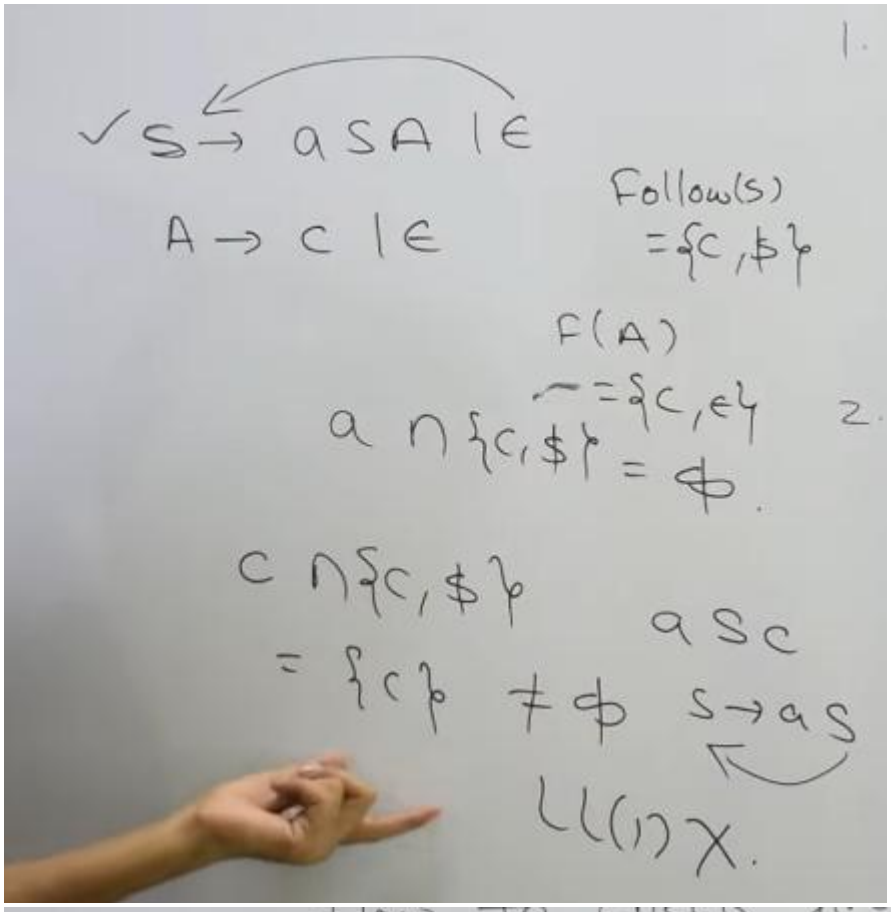


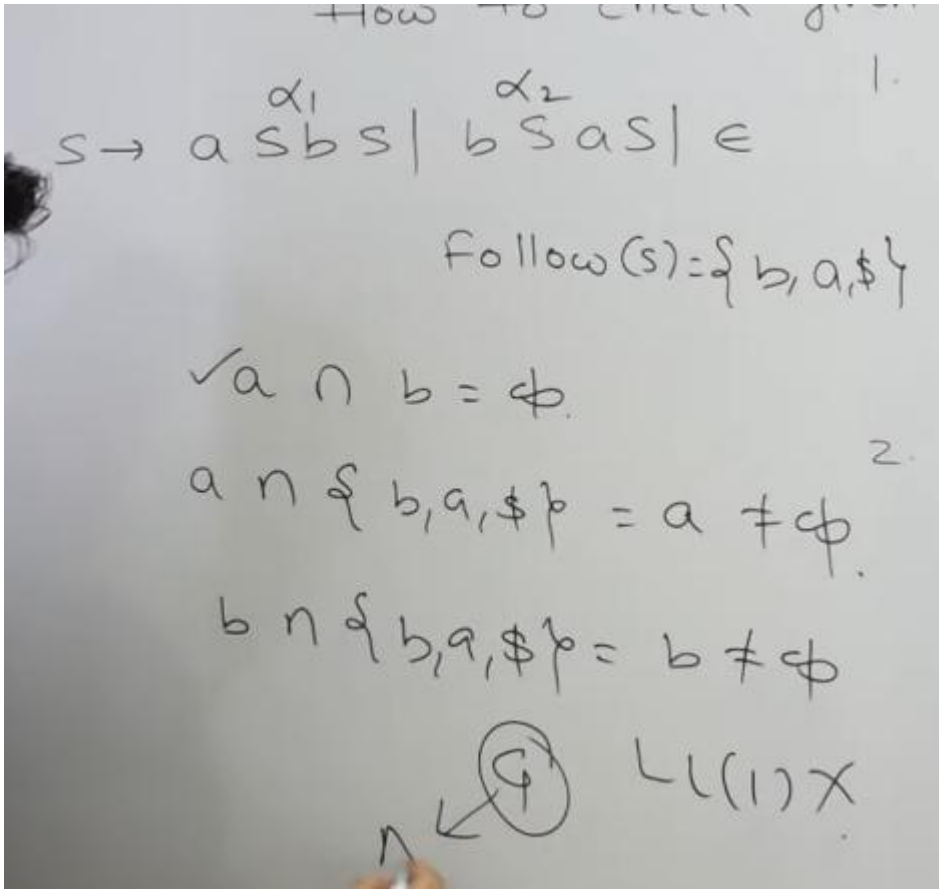
How to Check a Grammar is LL(1) or not




α_1 α_2
S → E | a
E → a
LL(1) X
a ∩ a = a

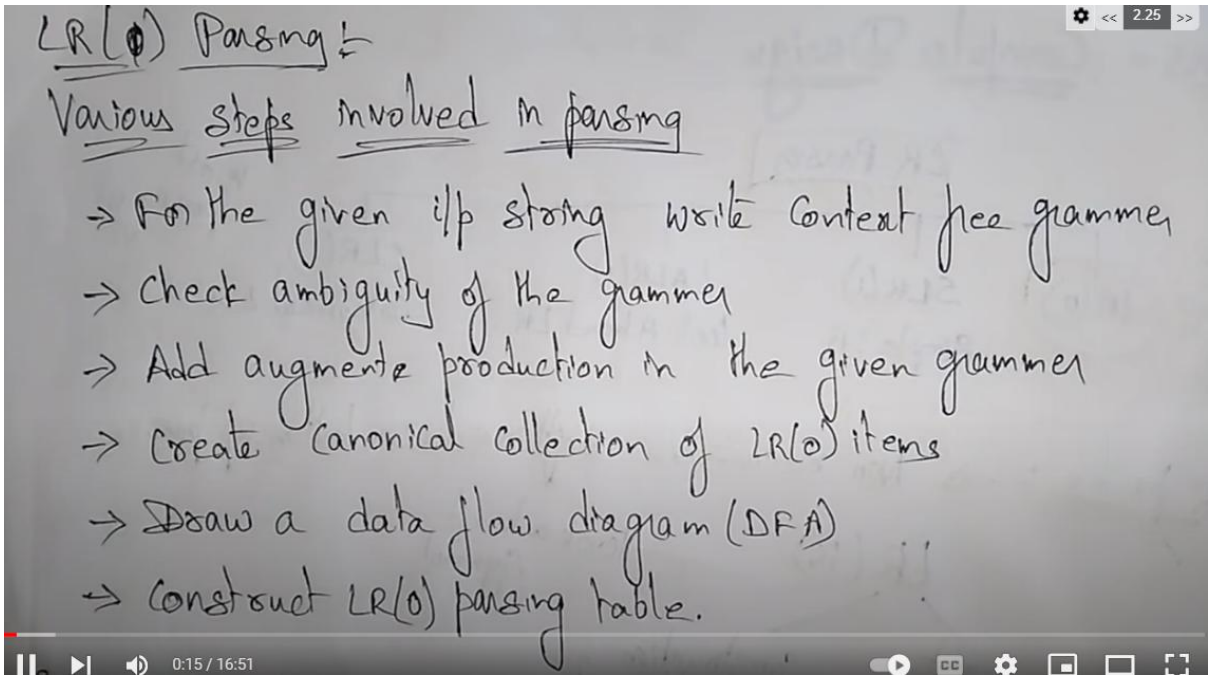

S → a A (B b)
A → a | ∈
a ∩ {a, b} = ϕ
B → d | ∈
a ∩ b = ϕ
{d, ∈}
LL(1)
(B b)
d b
∈ · b = b


1.
✓S → aSA | ∈
A → c | ∈
Follow(S)
= {c, $}
F(A)
= {c, ∈}
2.
a ∩ {c, $} = ϕ.
c ∩ {c, $}
= {c} ≠ ϕ
a Sc
S → aS
LL(1) ✗.

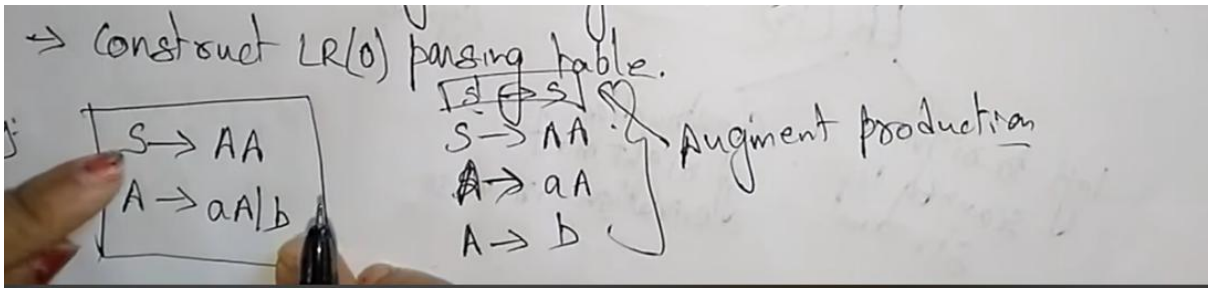

1.
S → a S b S | b S a S | ∈
α1
α2
Follow(S) = {b, a, $}
✓a ∩ b = ϕ.
2.
a ∩ {b, a, $} = a ≠ ϕ.
b ∩ {b, a, $} = b ≠ ϕ
LL(1) ✗.

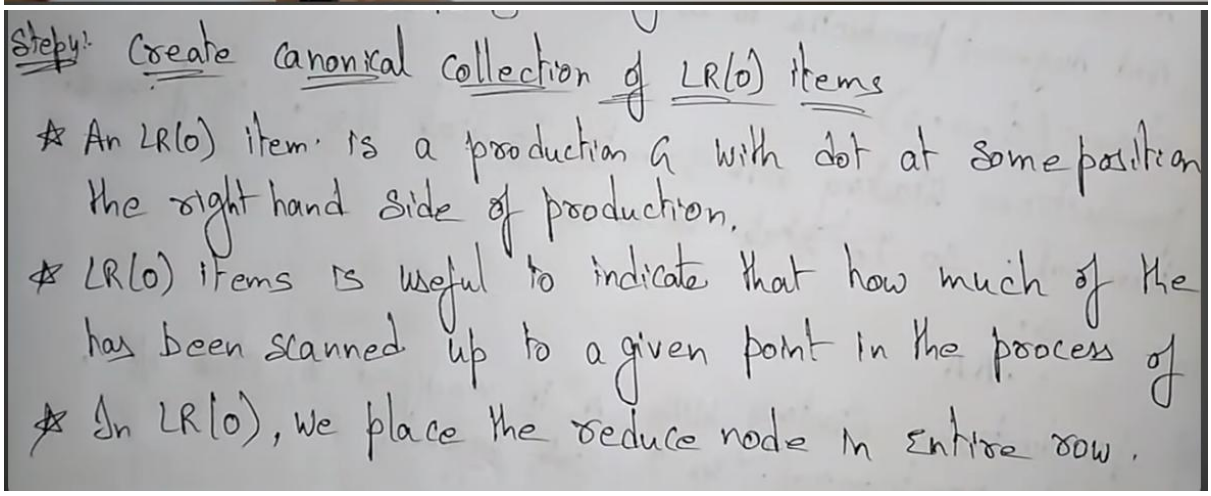
LR(0) Parser

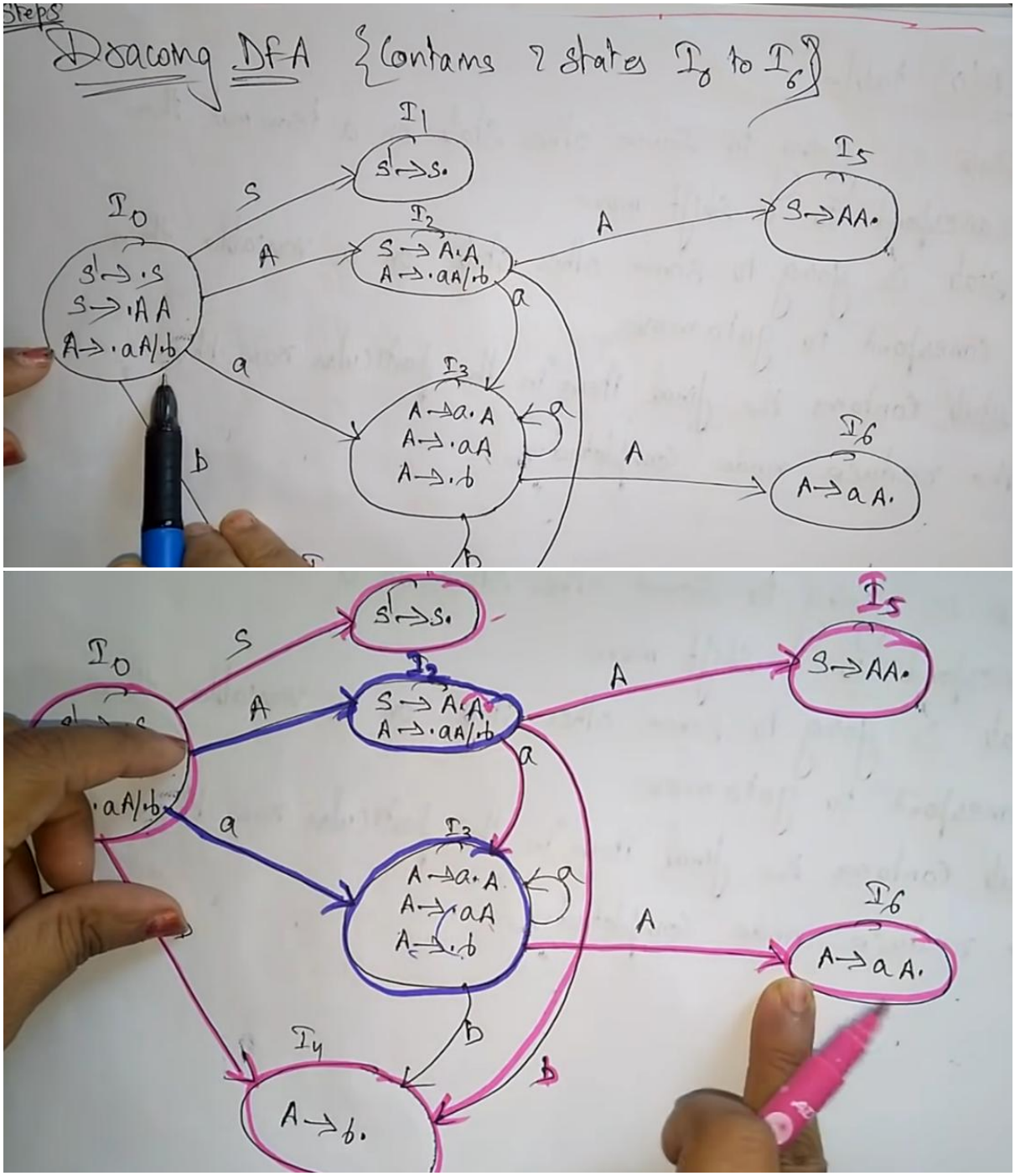
https://www.geeksforgeeks.org/problem-on-lr0-parser/

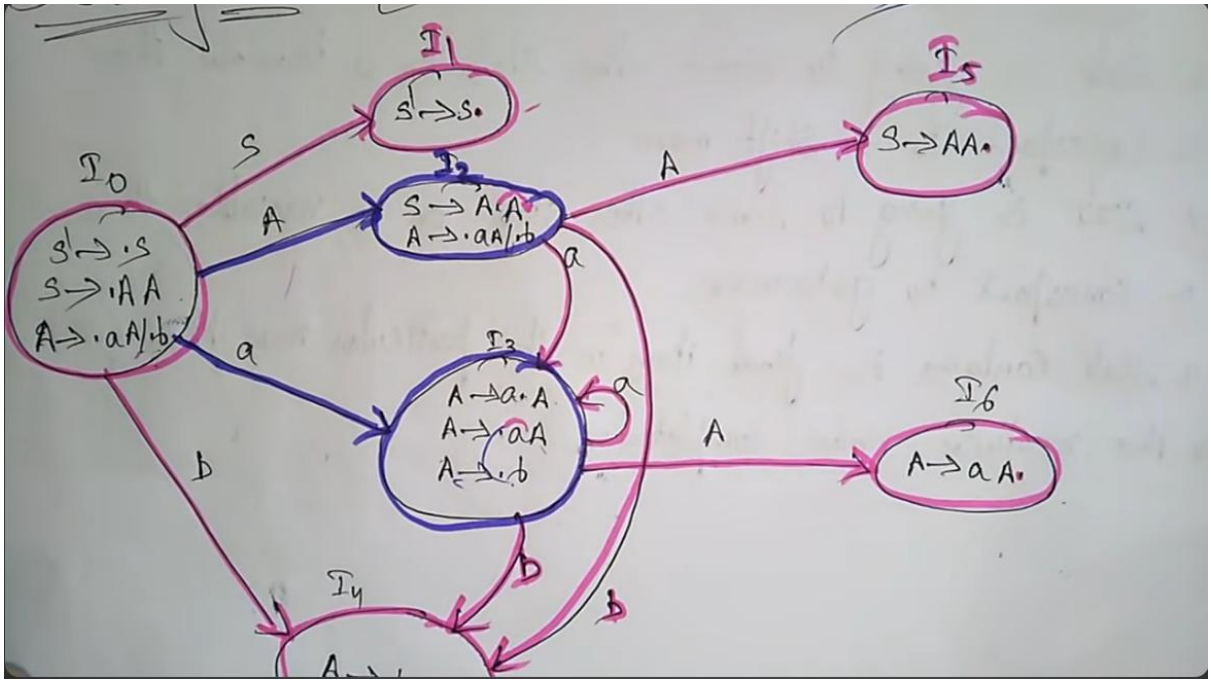


Augmented grammar helps us to identify when to stop the parser. Generally in augmented grammar the start symbol is derived by any other symbol



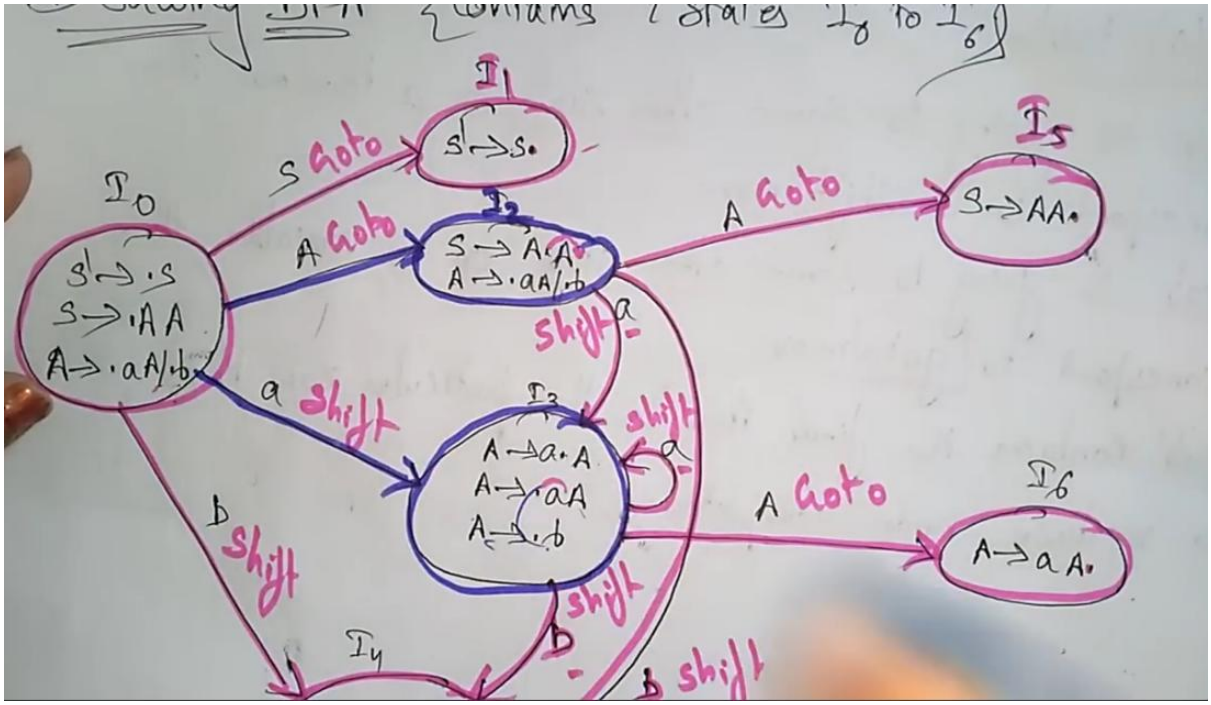



Steps
Drawing DFA {Contains 7 states I_0 to I_6}
I_0
$S' \to .S$
$S \to .AA$
$A \to .aA/.b$
I_1
$S' \to S.$
I_2
$S \to A.A$
$A \to .aA/.b$
I_3
$A \to a.A$
$A \to .aA$
$A \to .b$
I_4
$A \to b.$
I_5
$S \to AA.$
I_6
$A \to aA.$
S
A
a
b



Step 6 :- LR(0) table

* If a state is going to some other state on a terminal it is correspond to a shift move
* If a state is going to some other state on a variable it is correspond to goto move
* If a state contains the final item in the particular row then write the reduce node completed.



States	Action			GOTO	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	γ_3	γ_3	γ_3		
I_5	γ_1	γ_1	γ_1		
I_6	γ_2	γ_2	γ_2		

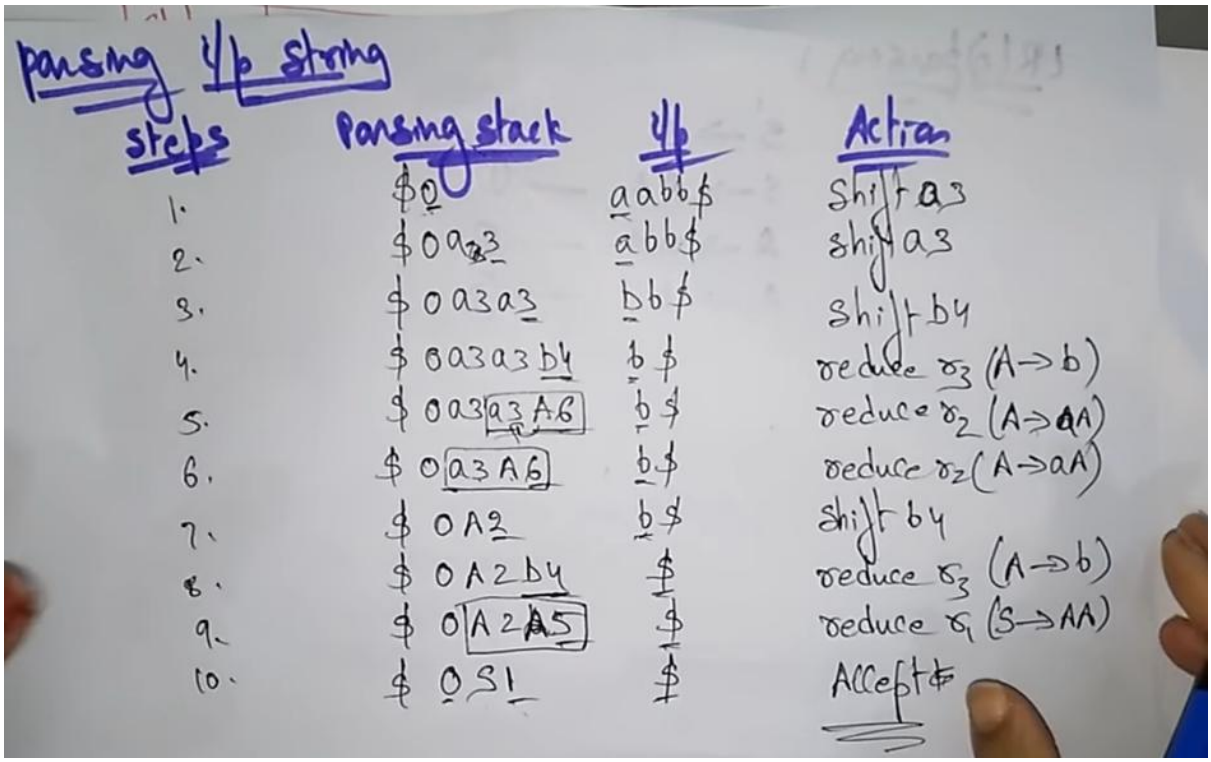
I_4, I_5, I_6 all contains final states

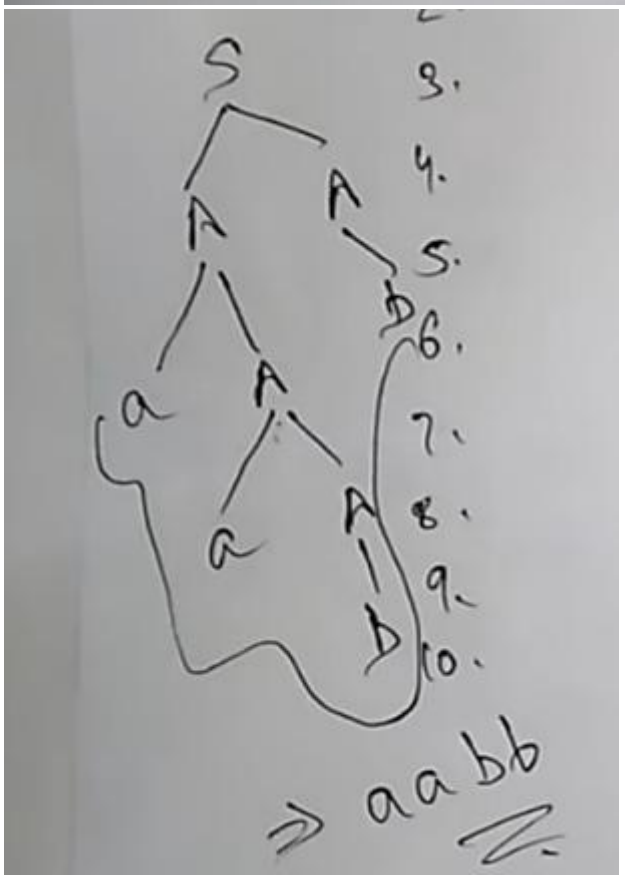
$S \to AA$ — (1)

$A \to aA$ — (2)

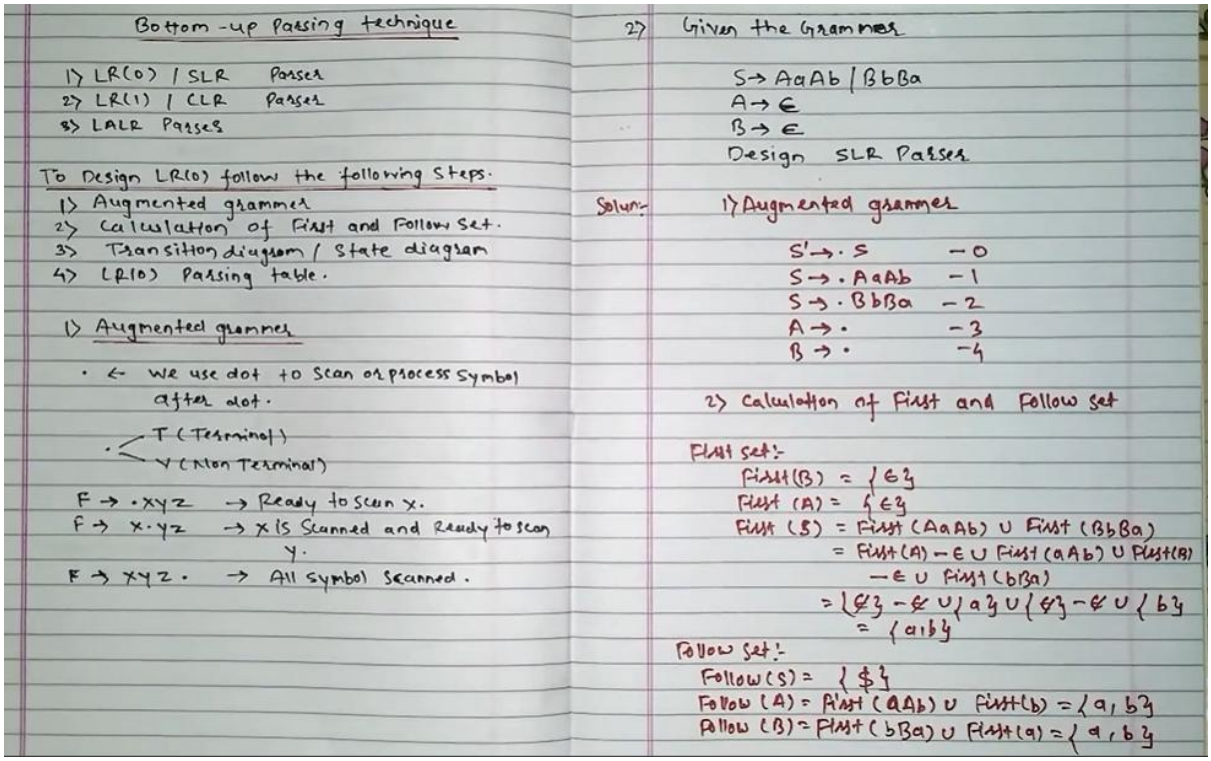
$A \to b$ — (3)

In I4, I5, I6,..... r3,r1,r2 numbers are given based on the production numbers shown in the above image.





LR(0) with epsilon productions



Bottom-up Parsing technique

1) LR(0) / SLR Parser
2) LR(1) / CLR Parser
3) LALR Parser

To Design LR(0) follow the following steps.
1) Augmented grammer
2) Calculation of First and Follow Set.
3) Transition diagram / State diagram
4) LR(0) Parsing table.

1) Augmented grammer

. ← We use dot to scan or process symbol after dot.

. → T (Terminal) / V (Non Terminal)

F → .XYZ → Ready to scan X.
F → X.YZ → X is Scanned and Ready to scan Y.
F → XYZ. → All symbol Scanned.

2) Given the Grammer

S → AaAb | BbBa
A → ∈
B → ∈
Design SLR Parser

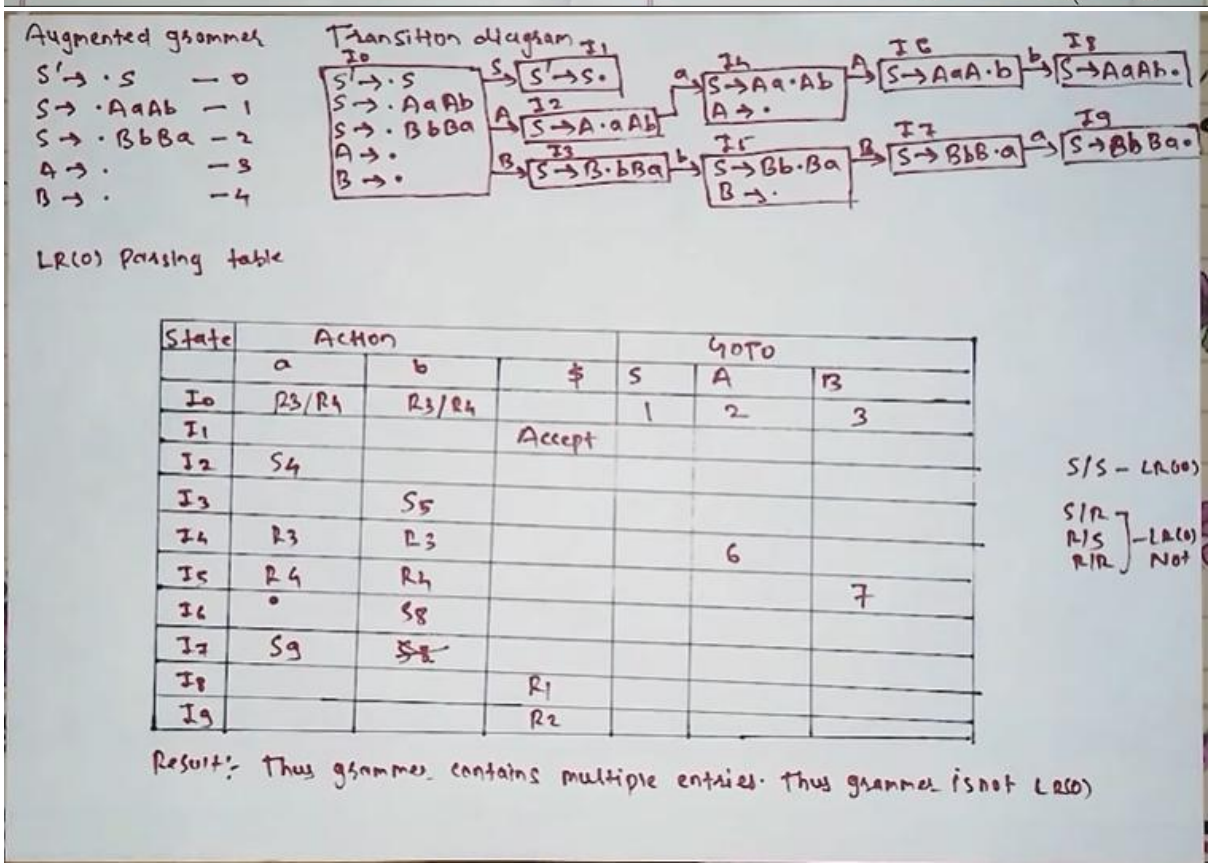
Soln:- 1) Augmented grammer

S' → .S — 0
S → .AaAb — 1
S → .BbBa — 2
A → . — 3
B → . — 4

2) Calculation of First and Follow set

First set:-
First(B) = {∈}
First(A) = {∈}
First(S) = First(AaAb) ∪ First(BbBa)
= First(A) − ∈ ∪ First(aAb) ∪ First(B) − ∈ ∪ First(bBa)
= {∈} − {∈} ∪ {a} ∪ {∈} − {∈} ∪ {b}
= {a, b}

Follow set:-
Follow(S) = {$}
Follow(A) = First(aAb) ∪ First(b) = {a, b}
Follow(B) = First(bBa) ∪ First(a) = {a, b}



Augmented grammer

S' → .S — 0
S → .AaAb — 1
S → .BbBa — 2
A → . — 3
B → . — 4

Transition diagram

I0: S' → .S, S → .AaAb, S → .BbBa, A → ., B → .
I1: S' → S.
I2: S → A.aAb, A → .
I3: S → B.bBa, B → .
I4: S → Aa.Ab
I5: S → Bb.Ba, B → .
I6: S → AaA.b
I7: S → BbB.a
I8: S → AaAb.
I9: S → BbBa.

LR(0) Parsing table

State	Action			Goto		
	a	b	$	S	A	B
I0	R3/R4	R3/R4		1	2	3
I1			Accept			
I2	S4					
I3		S5				
I4	R3	R3			6	
I5	R4	R4				7
I6		S8				
I7	S9	~~S8~~				
I8			R1			
I9			R2			

S/S – LR(0)
S/R, R/S, R/R – LR(0) Not

Result:- Thus grammer contains multiple entries. Thus grammer is not LR(0)

Bottom-up Parsing technique

1) LR(0) / SLR Parser
2) LR(1) / CLR Parser
3) LALR Parser

To Design LR(0) follow the following steps.
1) Augmented grammer
2) Calculation of First and Follow Set.
3) Transition diagram / State diagram
4) LR(0) Parsing table.

1) Augmented grammer

. ← we use dot to scan or process symbol after dot.

. → T (Terminal)
. → V (Non Terminal)

F → .XYZ → Ready to scan X.
F → X.YZ → X is Scanned and Ready to scan Y.
F → XYZ. → All Symbol Scanned.

1) Construct LR(0) Parsing table and check whether the given grammer is LR(0).

E → E+E / E*E / (E) / id

Soln: 1) Augmented grammer

E' → .E — 0
E → .E+E — 1
E → .E*E — 2
E → .(E) — 3
E → .id — 4

2) calculation First and Follow set

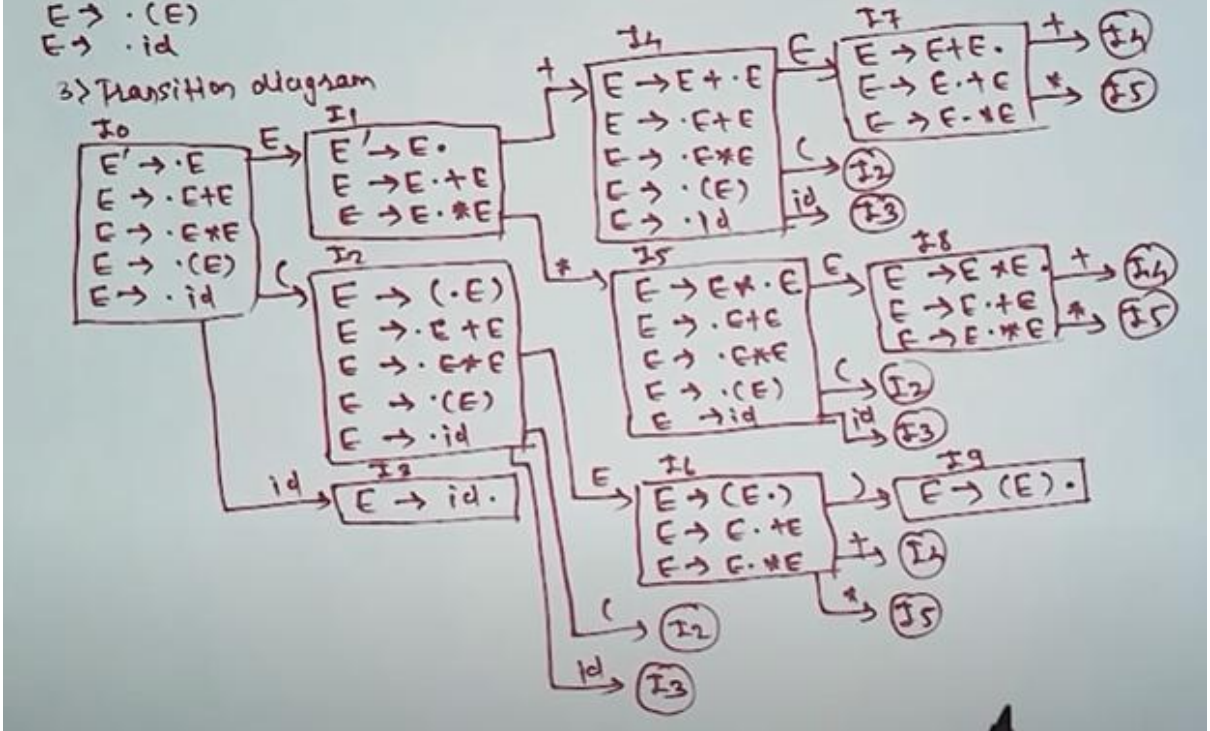
First(E) = First(E+E) ∪ First(E*E) ∪ ((E)) ∪ First(id)
= First(E+E) ∪ First(E*E) ∪ {(} ∪ {id}
= First(E) ∪ First(E) ∪ {(, id}
= {(, id}

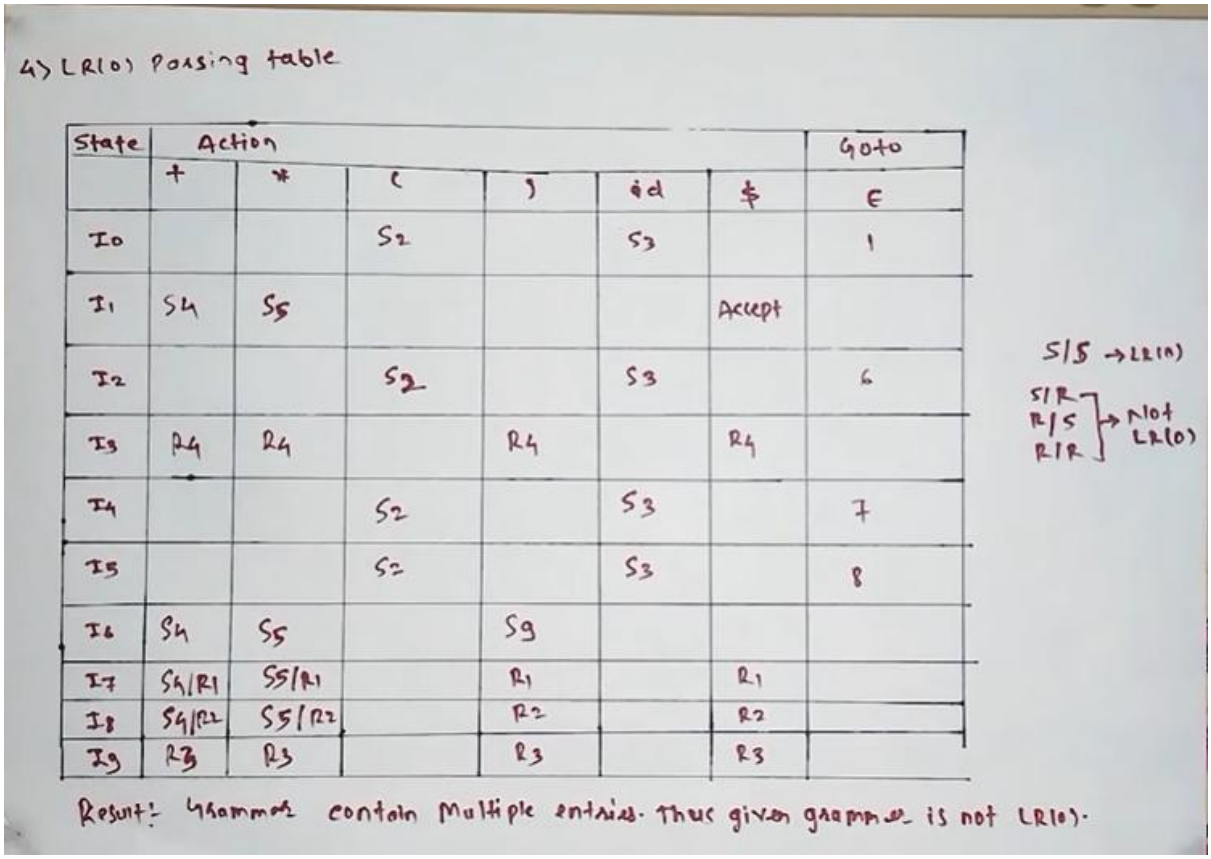
Follow(E) = {$, +, *,)}

Augmented grammer

E' → .E
E → .E+E
E → .E*E
E → .(E)
E → .id

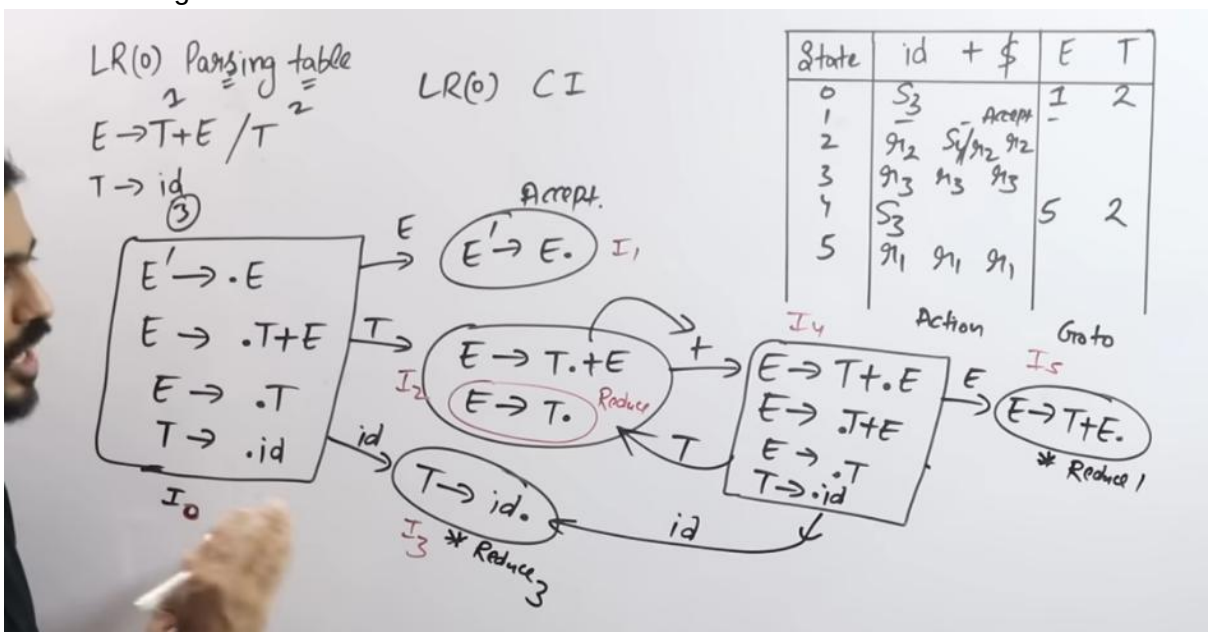
3) Transition diagram

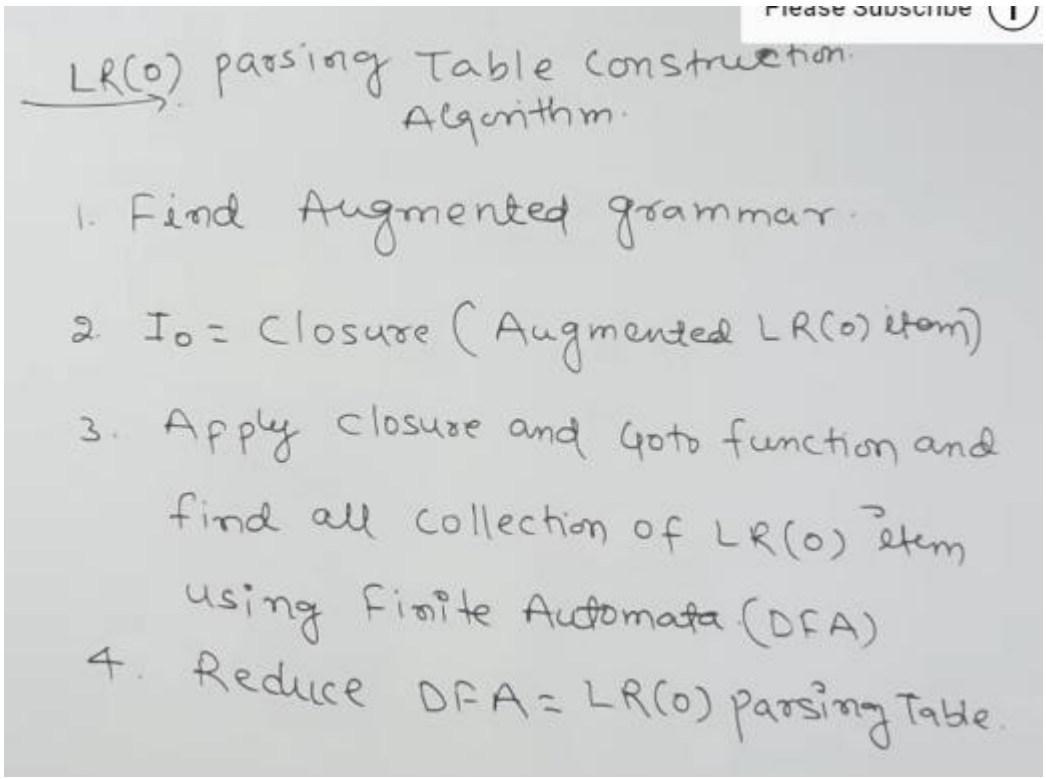




Check Whether a Grammar is LR(0) or not

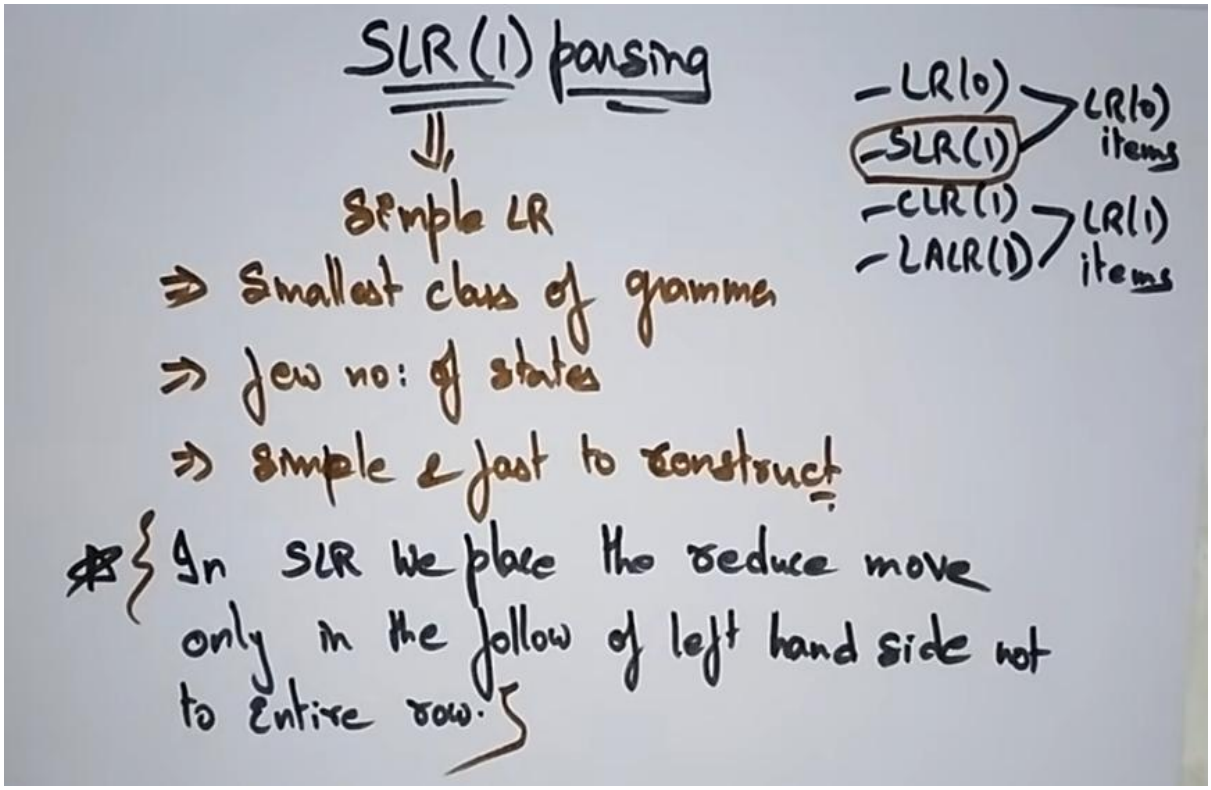
If the parsing table contains more than one entry in the same cell like shift and reduce or reduce and reduce etc. then it's not a LR(0) grammar.Below is given an example demonstrating it.

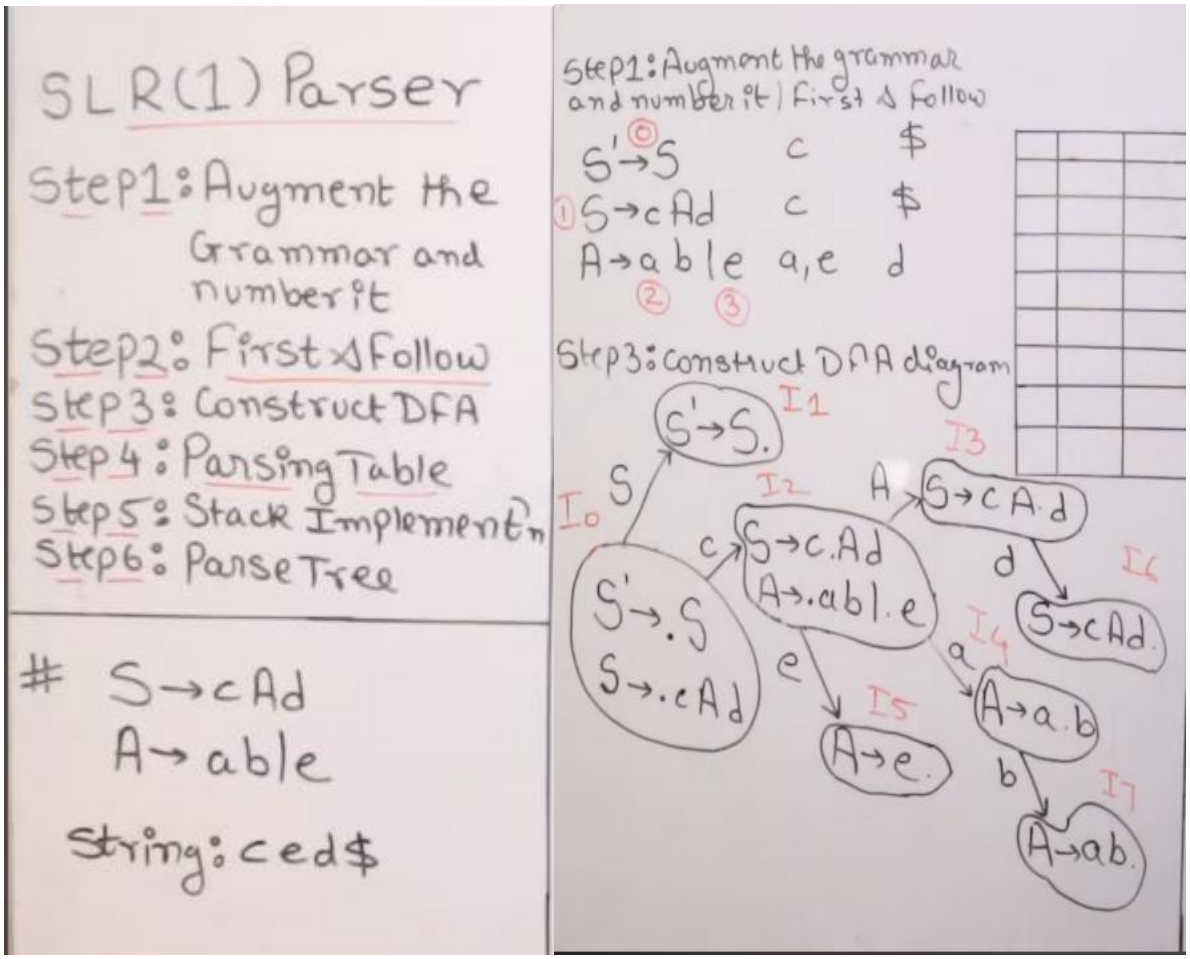




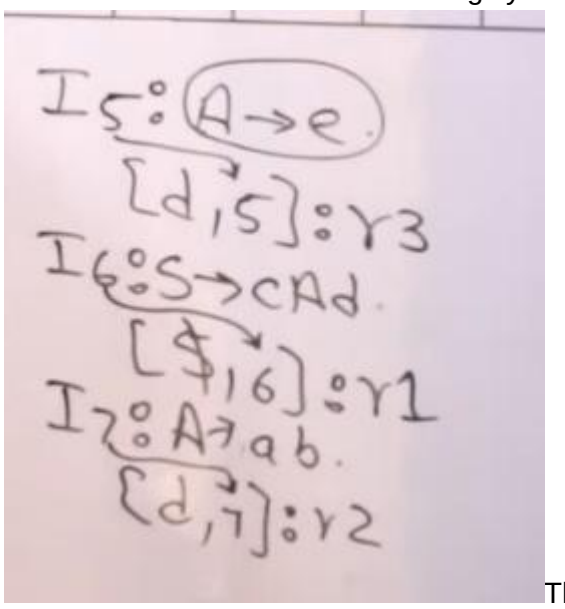
SLR(1) parser

https://www.geeksforgeeks.org/slr-parser-with-examples/



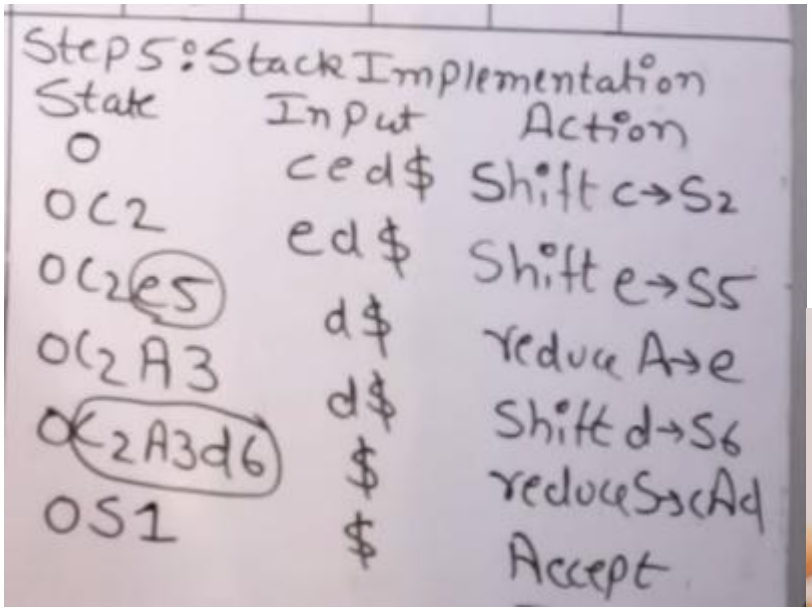


I0 contains the closure of the starting symbol always.



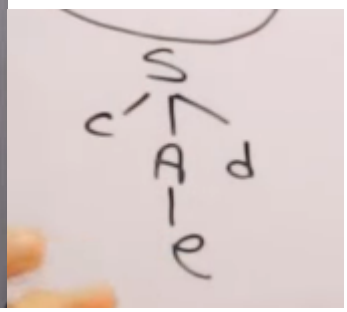
The terminal taken in the square brace is the follow and the number succeeding r is the number of the production.

States	Action						Goto	
	a	b	c	d	e	$	A	S
I0			S2					1
I1						A		
I2	S4				S5		3	
I3				S6				
I4		S7						
I5				r3				
I6						r1		
I7				r2				



Steps: Stack Implementation

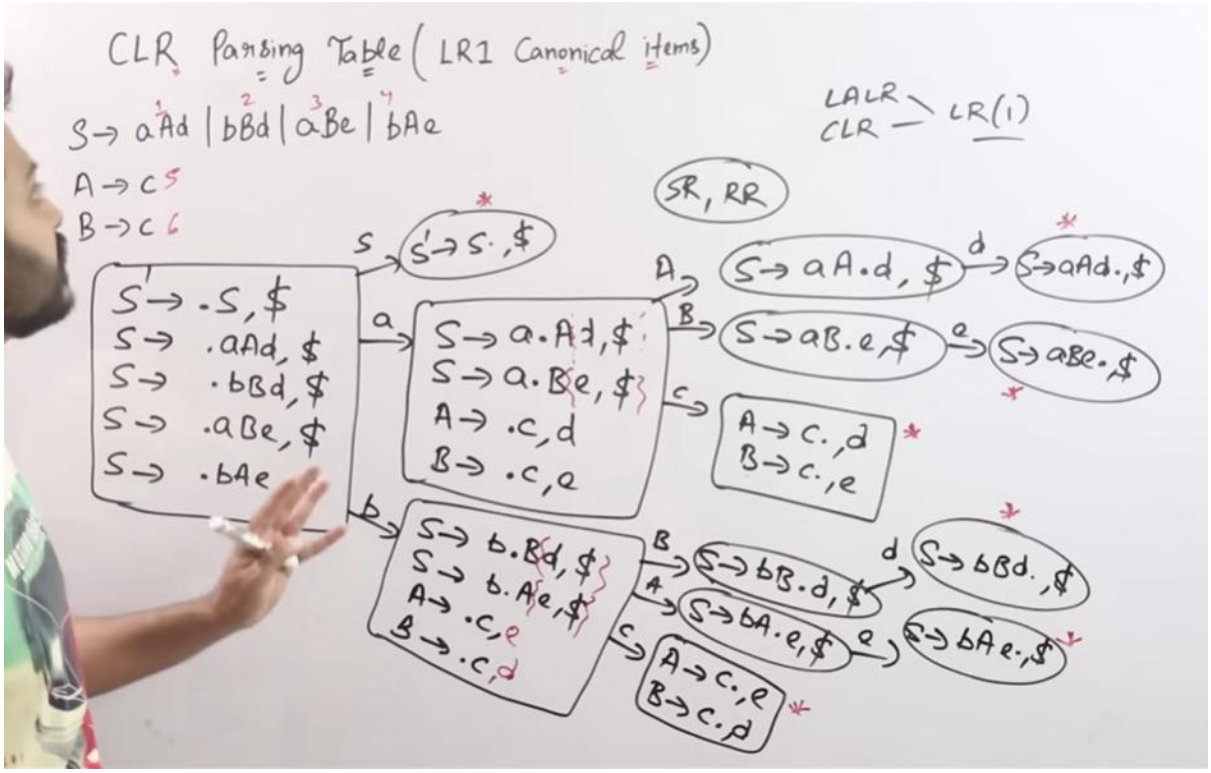
State	Input	Action
0	ced$	Shift c→S2
0c2	ed$	Shift e→S5
0c2e5	d$	reduce A→e
0c2A3	d$	Shift d→S6
0c2A3d6	$	reduce S→cAd
0S1	$	Accept



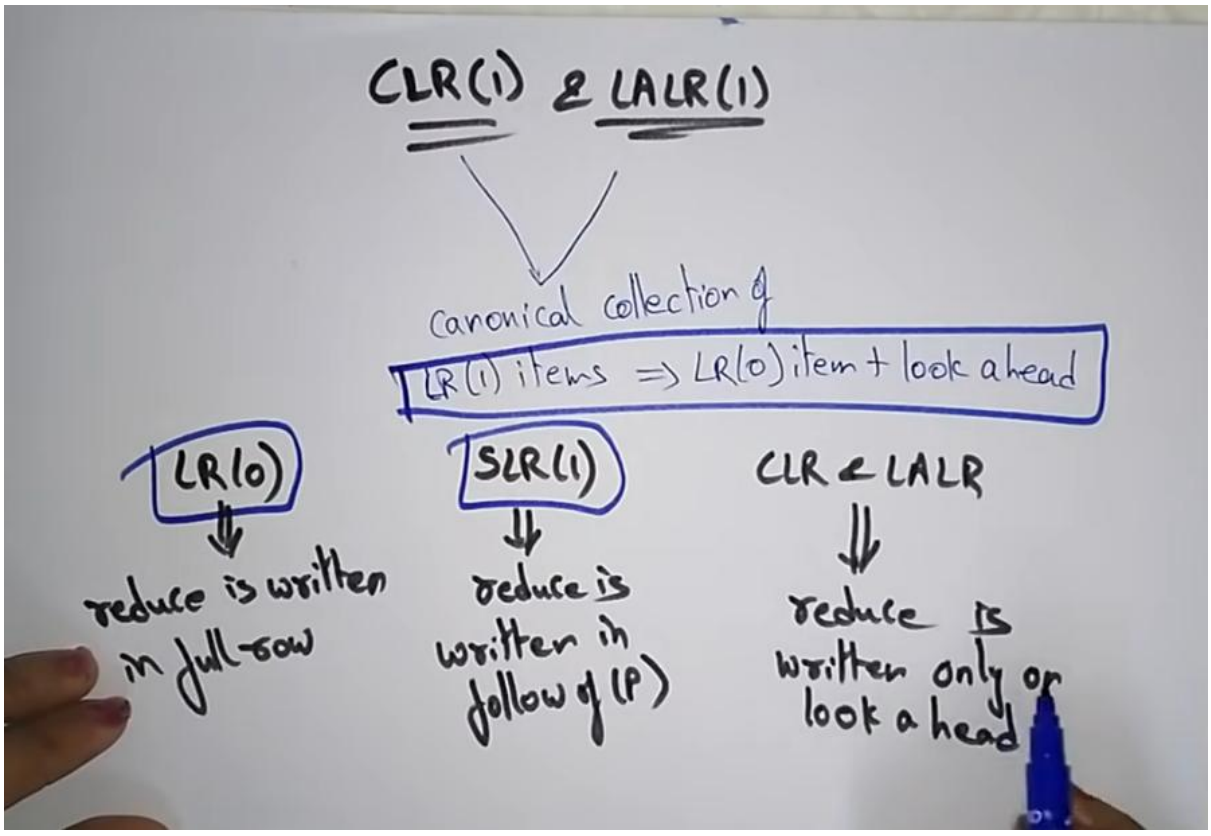
CLR (1) parser

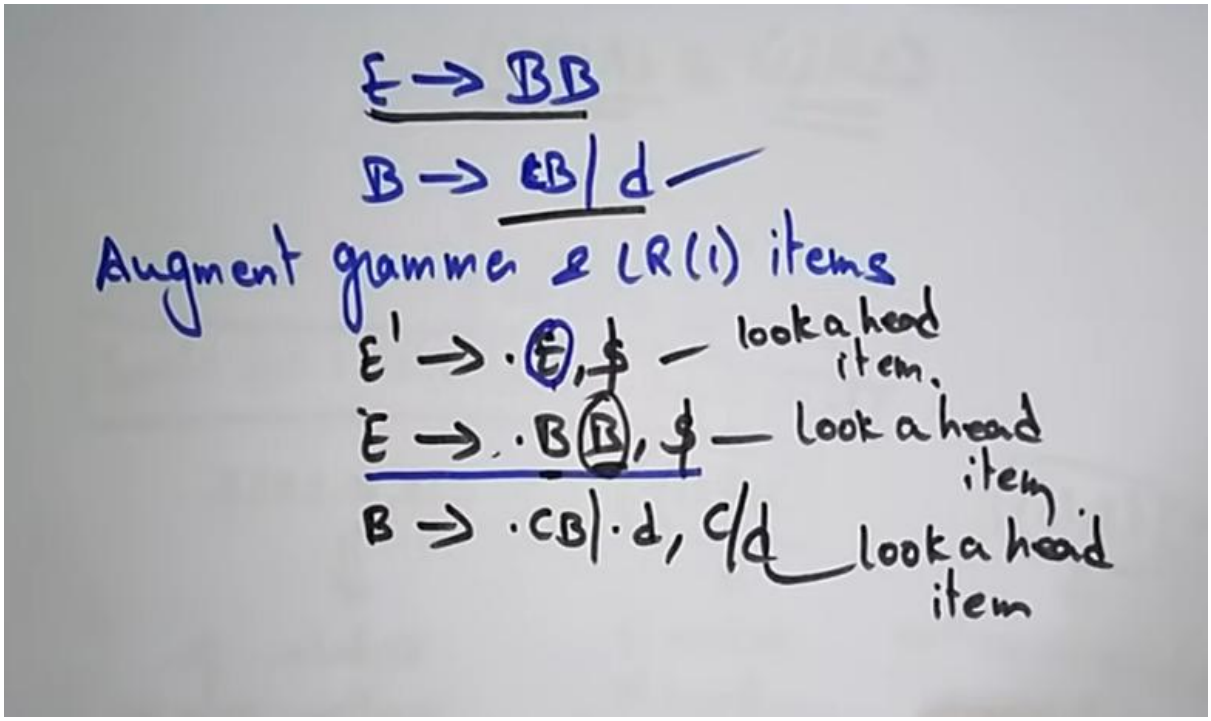
https://www.geeksforgeeks.org/clr-parser-with-examples/

Example 1



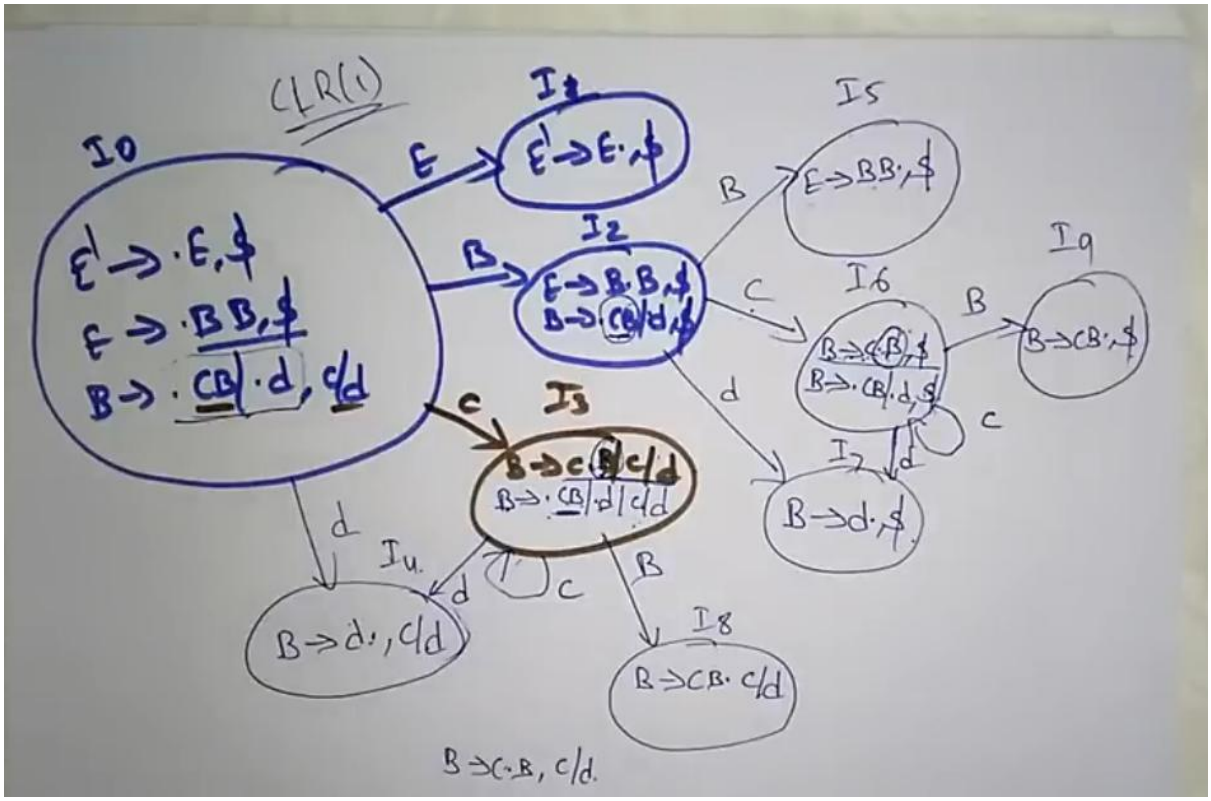
Example 2

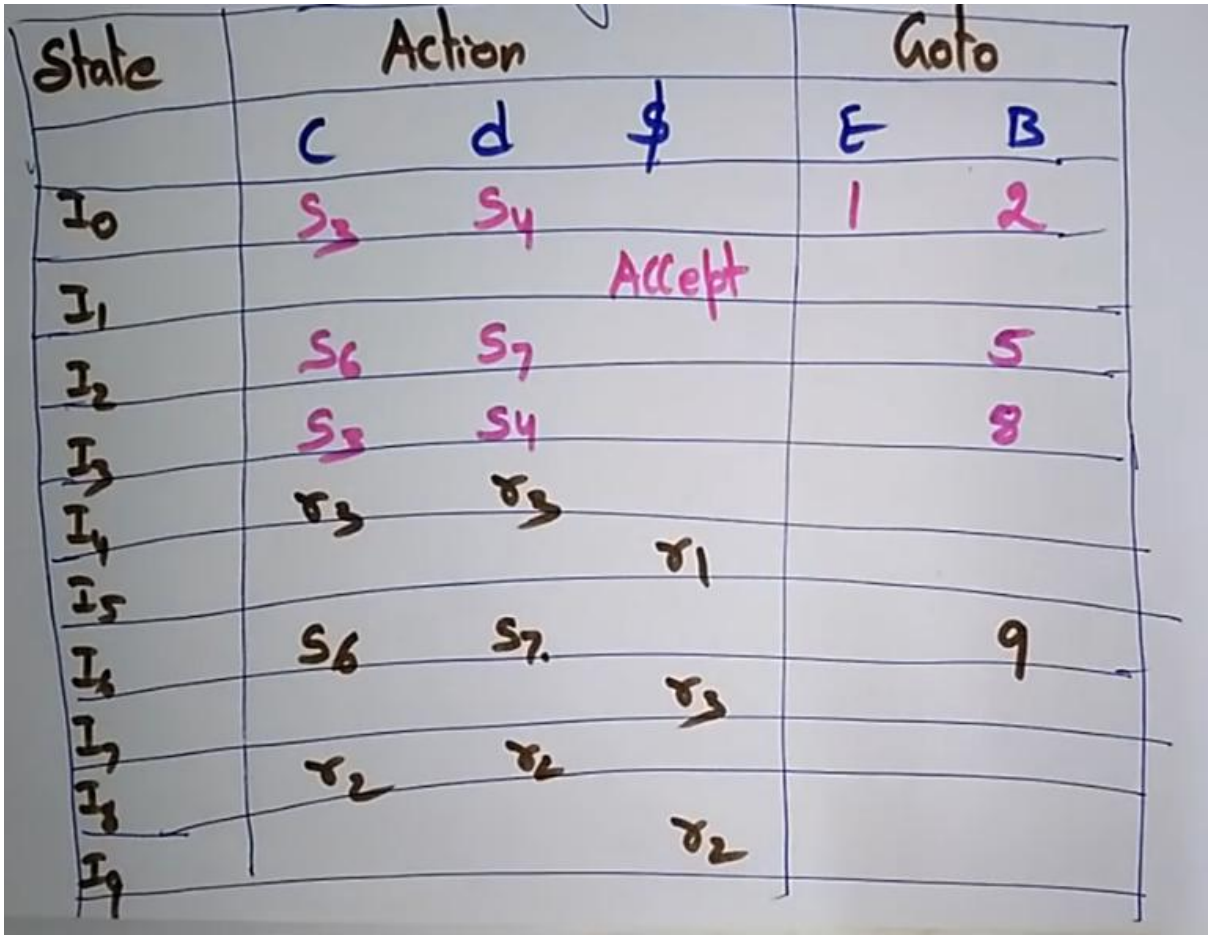




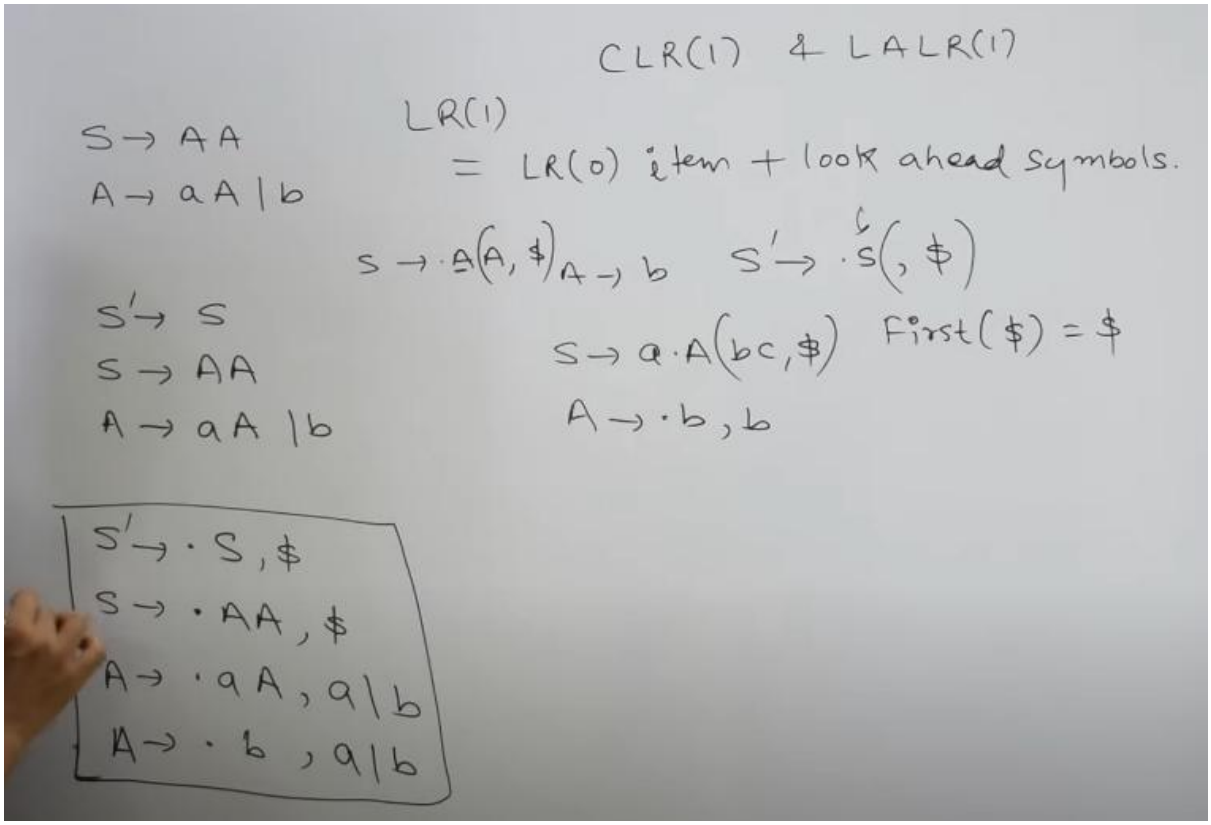
If after a non-terminal nothing is there then just write the lookahead value whatever it is there.

If there is another non-terminal then you have to take the first of that non-terminal.

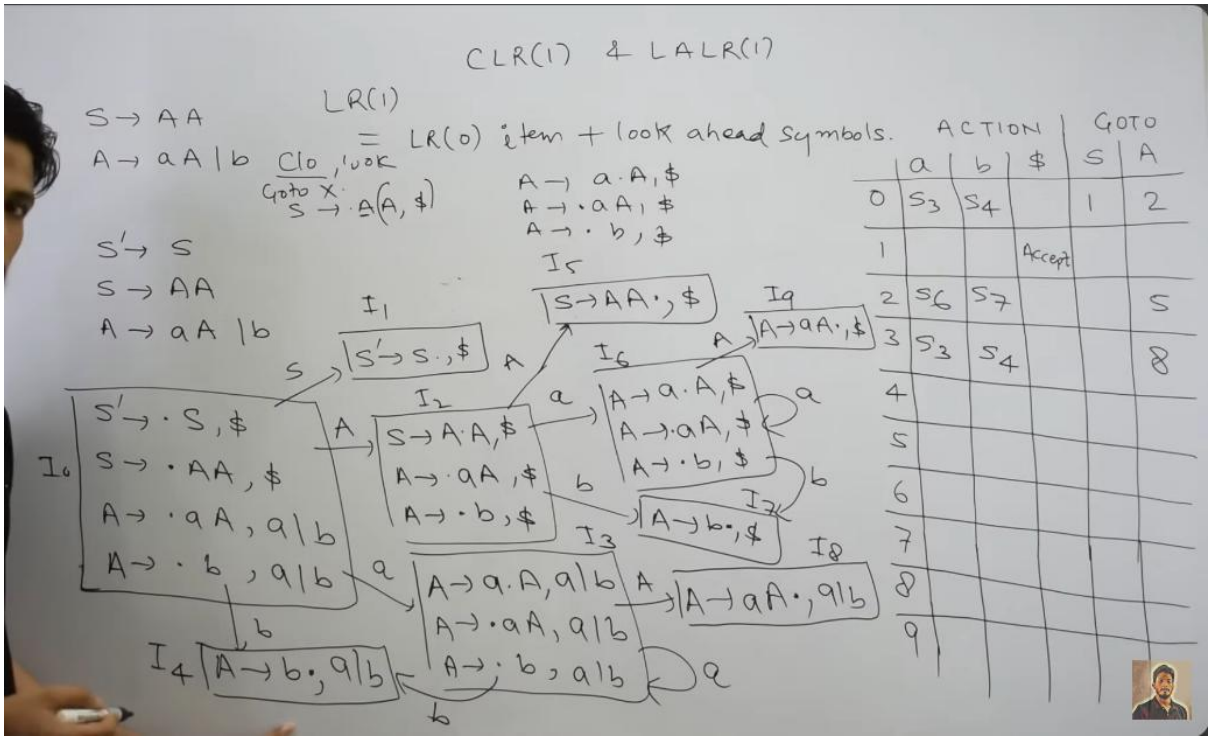



State
Action
Goto
c
d
$
E
B
s_3
s_4
1
2
Accept
s_6
s_7
5
s_3
s_4
8
r_3
r_3
r_1
s_6
s_7
9
r_3
r_2
r_2
r_2

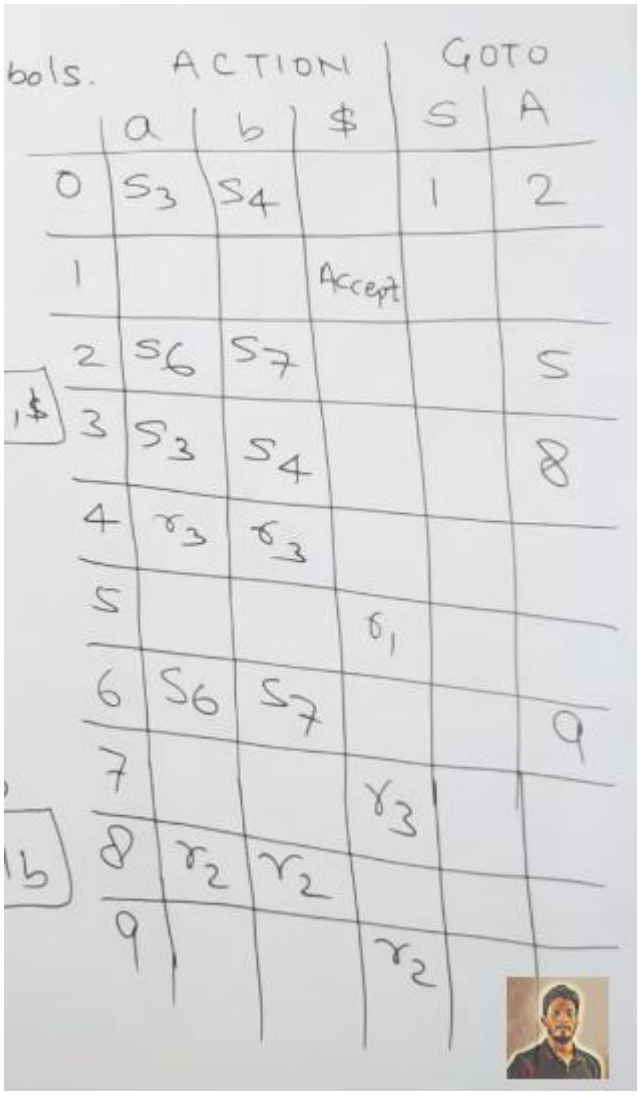
Example 2


CLR(1) & LALR(1)
LR(1)
= LR(0) item + look ahead symbols.
S → AA
A → aA | b
S → ·A(A, $) A → b
S' → ·S(, $)
S' → S
S → AA
A → aA | b
S → a·A(bc, $)
First($) = $
A → ·b, b
S' → ·S, $
S → ·AA, $
A → ·aA, a|b
A → ·b, a|b

When a new production is added due to the previous one, we take the first of the remaining symbols after excluding the first one after dot.
When applying goto function the previous look-ahead symbol remains the same. It might change while finding the closure.

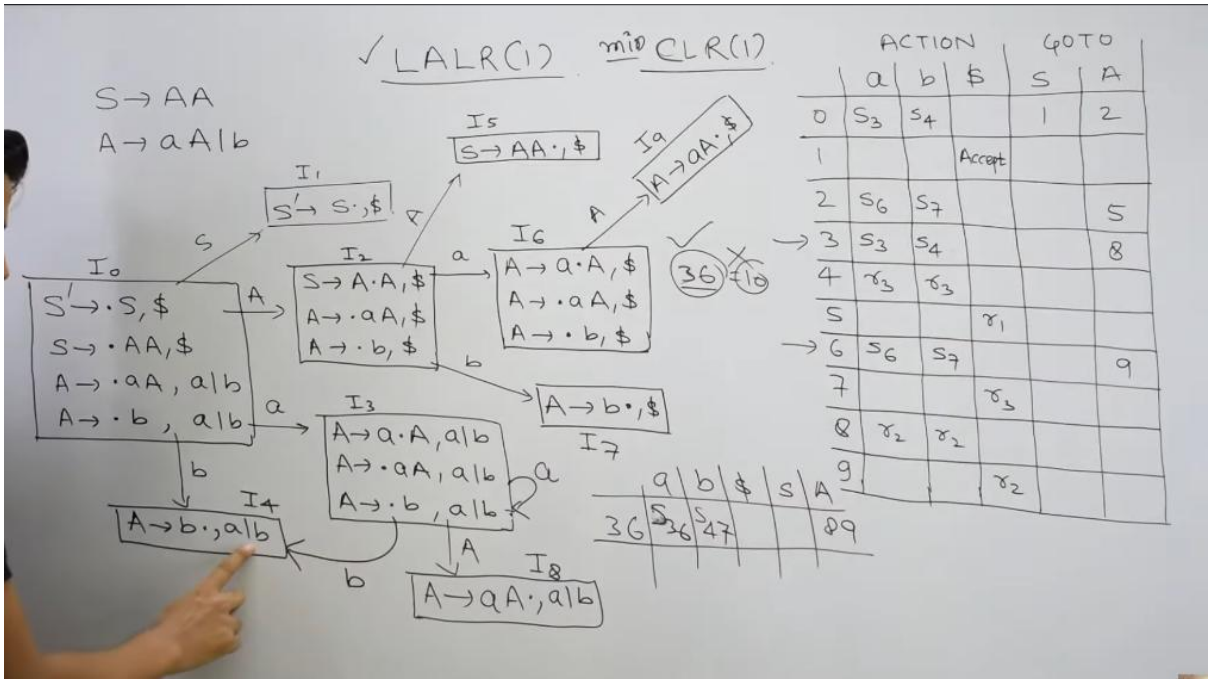


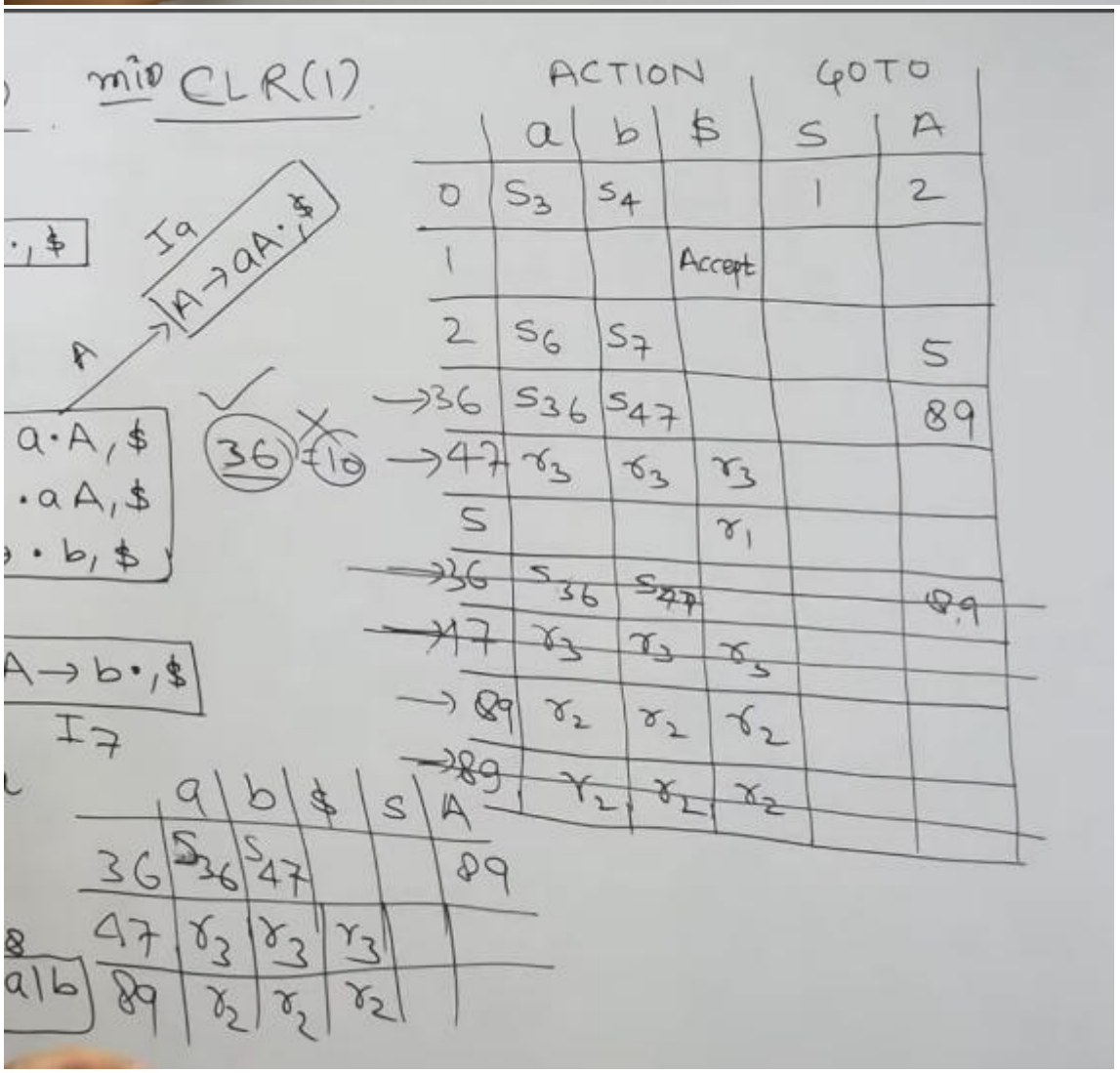
In row 4 i.e. for I4 as we can see that it has reached the end so we need to write rn in it…… so the question arises where should we write rn, so we look at the lookahead symbol and add rn in the following columns, also the value of n is the number of production in the given grammar.



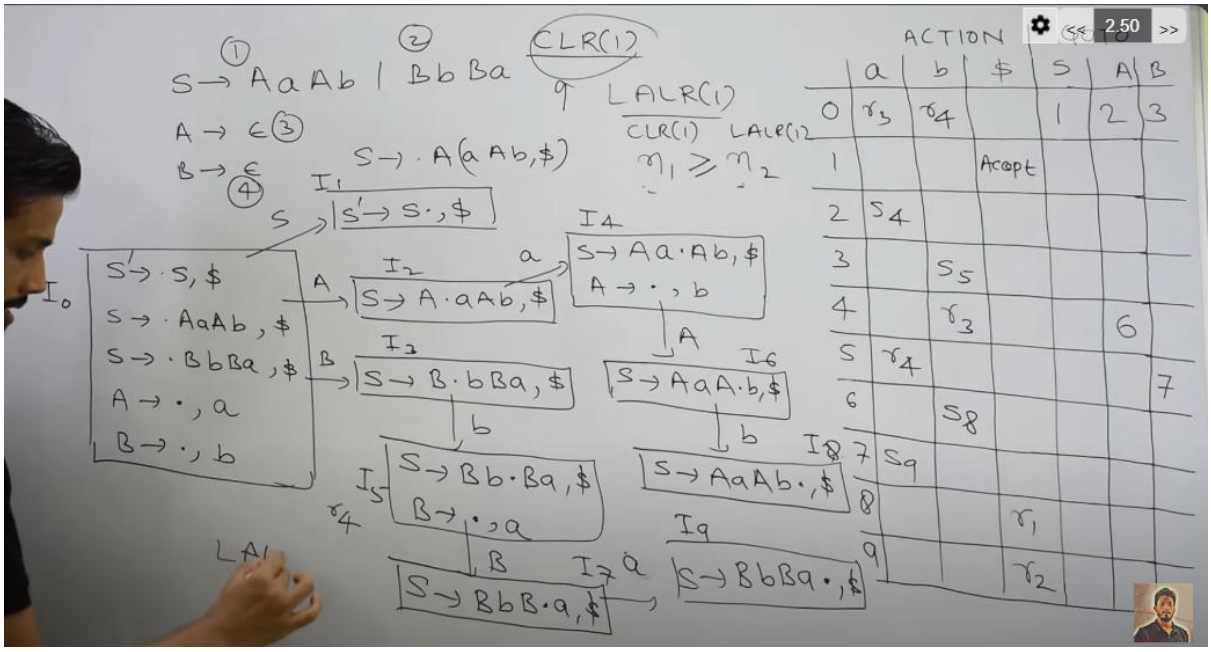
LALR(1) Parser

It’s the minimised version of CLR parser.
If any two states have the same LR(0) items and different lookahead symbols, then we can combine them.

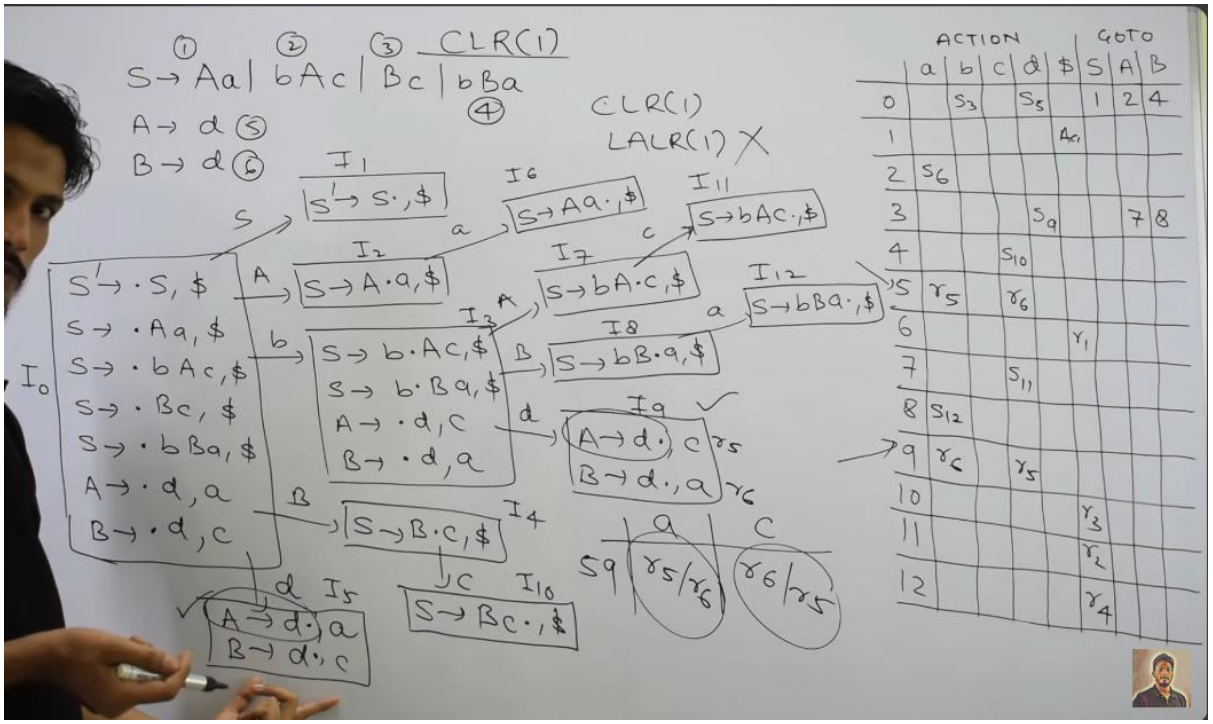




In the below example as all the states have different LR(0) items so we can conclude that the CLR(1) parser is already converted to LALR(1).



The below given example is a CLR parser but not a LALR parser as it gives different rn in the same row.



Operator Grammar

Operator precedence grammar also accepts ambiguous grammar.

OPERATOR GRAMMAR

A grammar that is used to define mathematical operators is called an operator grammar or operator precedance grammar.

A grammar is said to be operator precedance grammar if it has 2 properties:

(i) no. R.H.S of any production has a $\in$.

(ii) no two non terminals are adjacent on RHS.

eg) $E = E+E \mid E*E \mid id$ // operator grammar

eg) $S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$ // NOT operator grammar

we will convert it into operator grammar

(i) no. R.H.S of any production has a $\in$.

(ii) no two non terminals are adjacent on RHS.

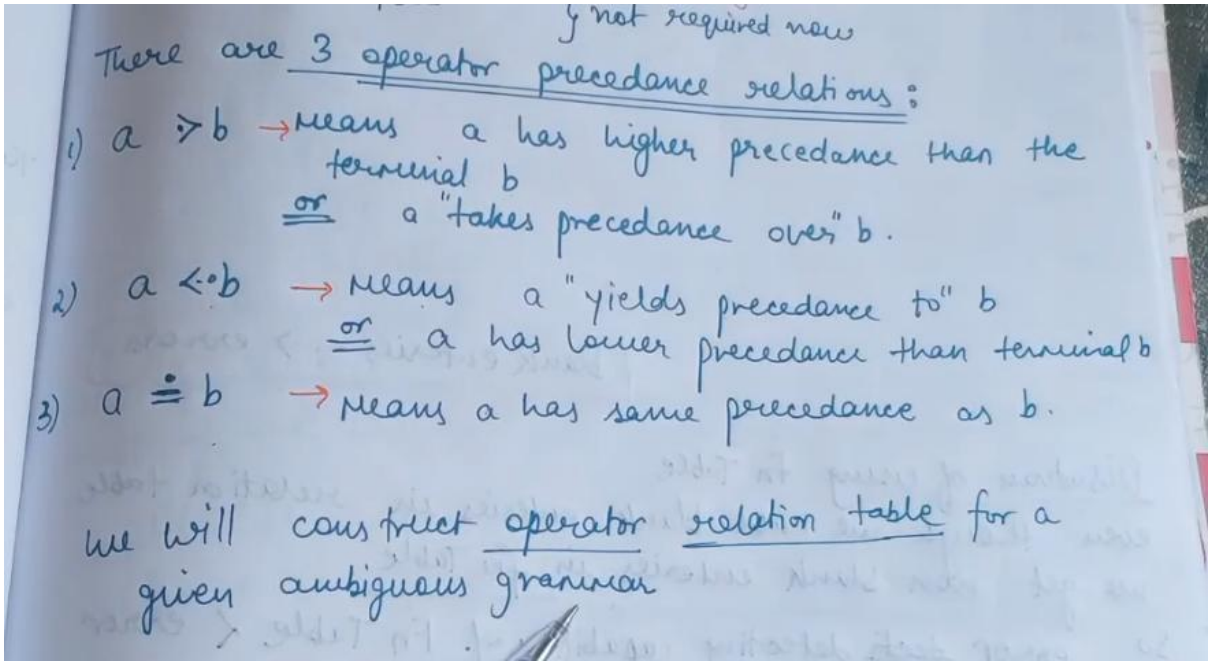
eg) $E \rightarrow E+E \mid E*E \mid id$ // operator grammar

eg) $S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$ // NOT operator grammar

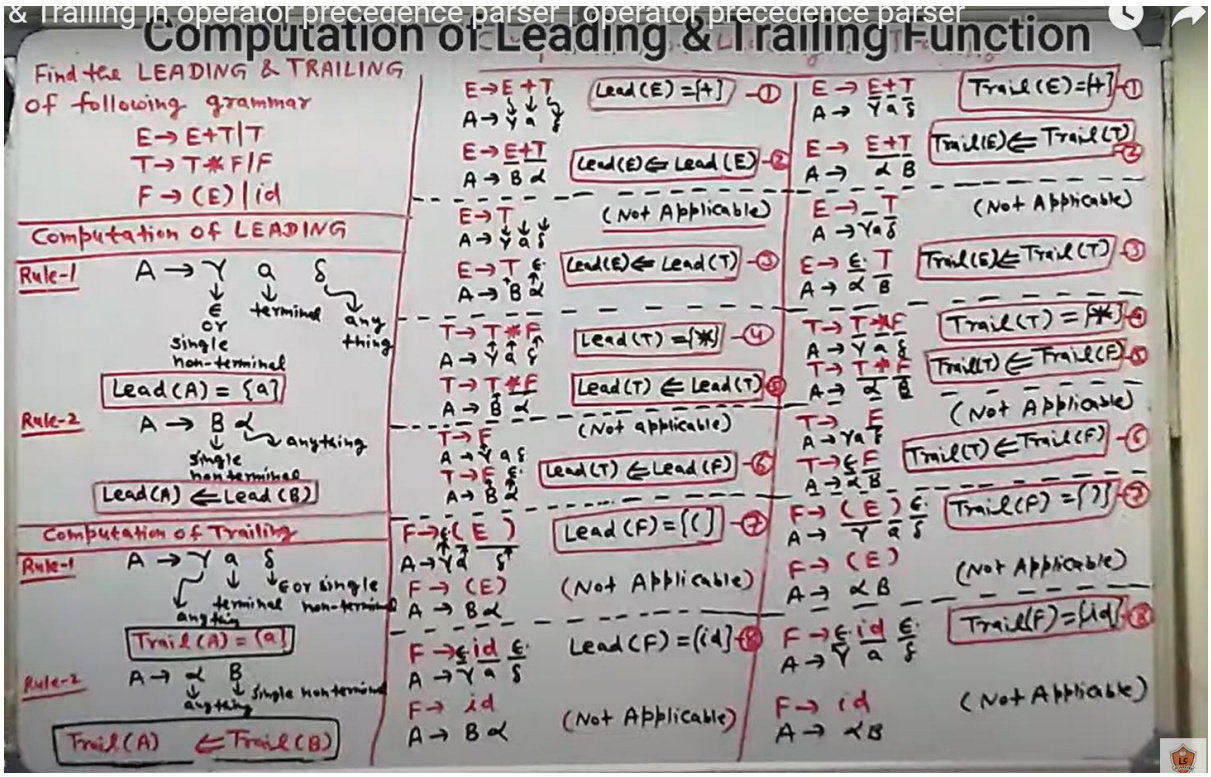
we will convert it into operator grammar

$S \rightarrow SbSbS \mid a \mid SbS$ // operator grammar

~~$A \rightarrow bSb \mid b$~~ } not required now



Leading and Trailing in Operator Precedence Parsing



LEADING(A)

(i) If $A \rightarrow \alpha a \beta$, α is single variable or ε

LEADING(A) = {a}

(ii) If $A \rightarrow Ba\beta$

LEADING(A) = LEADING(B)

EX:

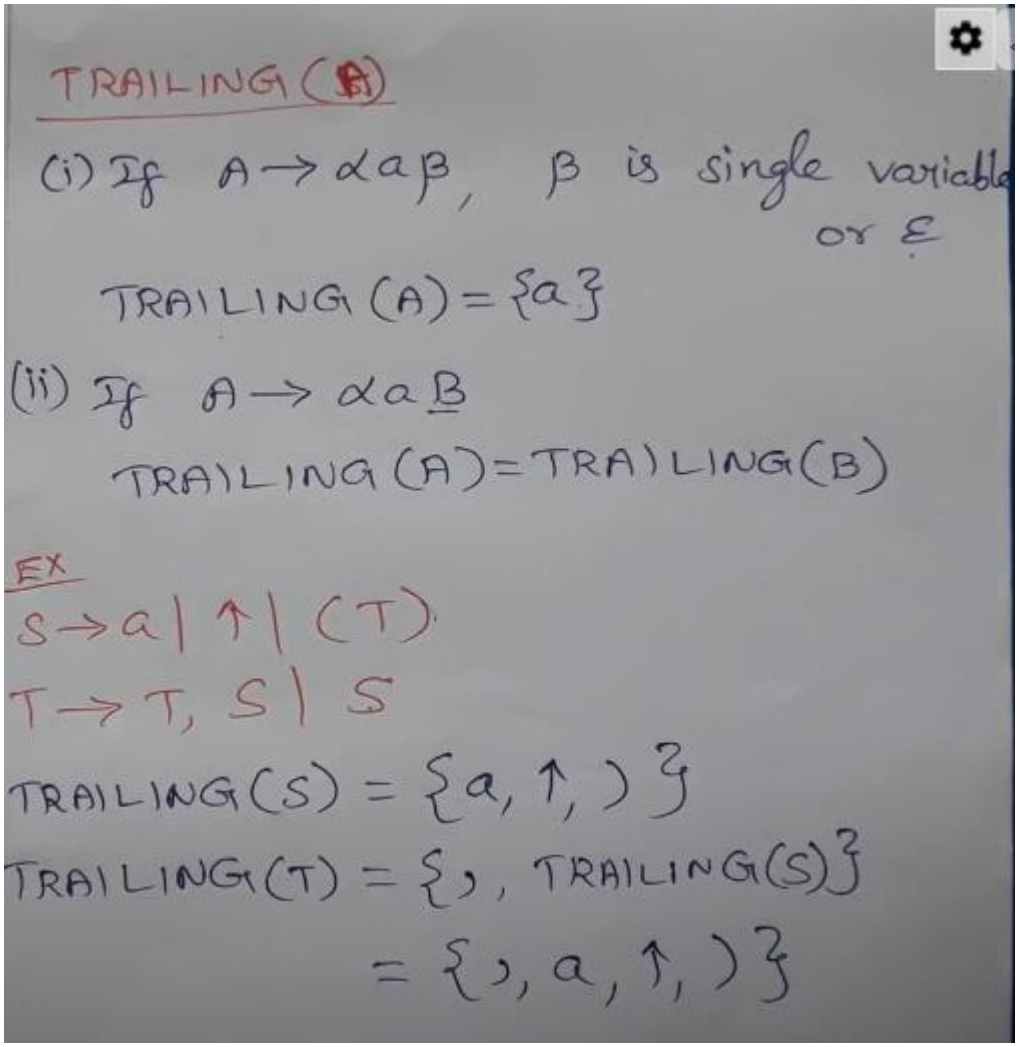
$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

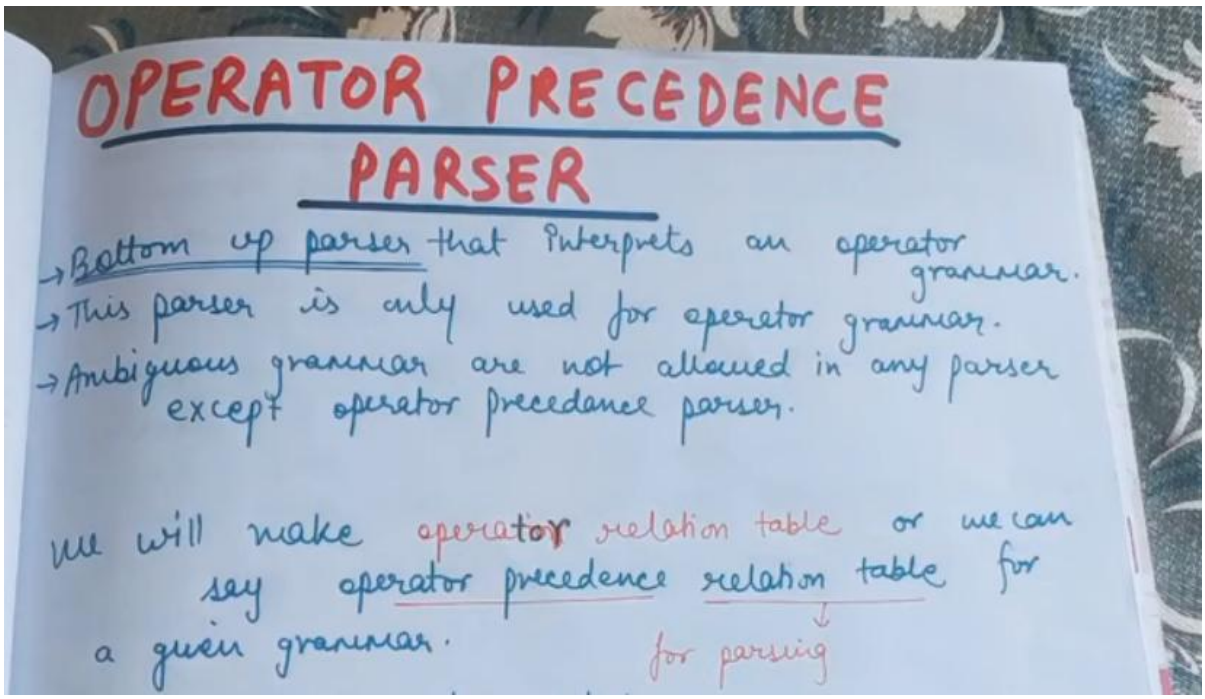
LEADING(S) = {a, ↑, (}

LEADING(T) = {, LEADING(S)}

= {, , a, ↑, (}



Operator Precedence Parser



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

a*b*c

(a+b)+c ab

id > +

	id	+	*	$
id	—	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	—

} we have defined here the precedence rules

Two ids will never be compared b/c they will never come side by side.

✓ Identifier will be given highest precedence compared to any other operator.

$ has least precedance compared to any other operator.

using this operator precedance table, parser will parse the i/p.

Disadvantage of operation Relation Table → no. of enteries

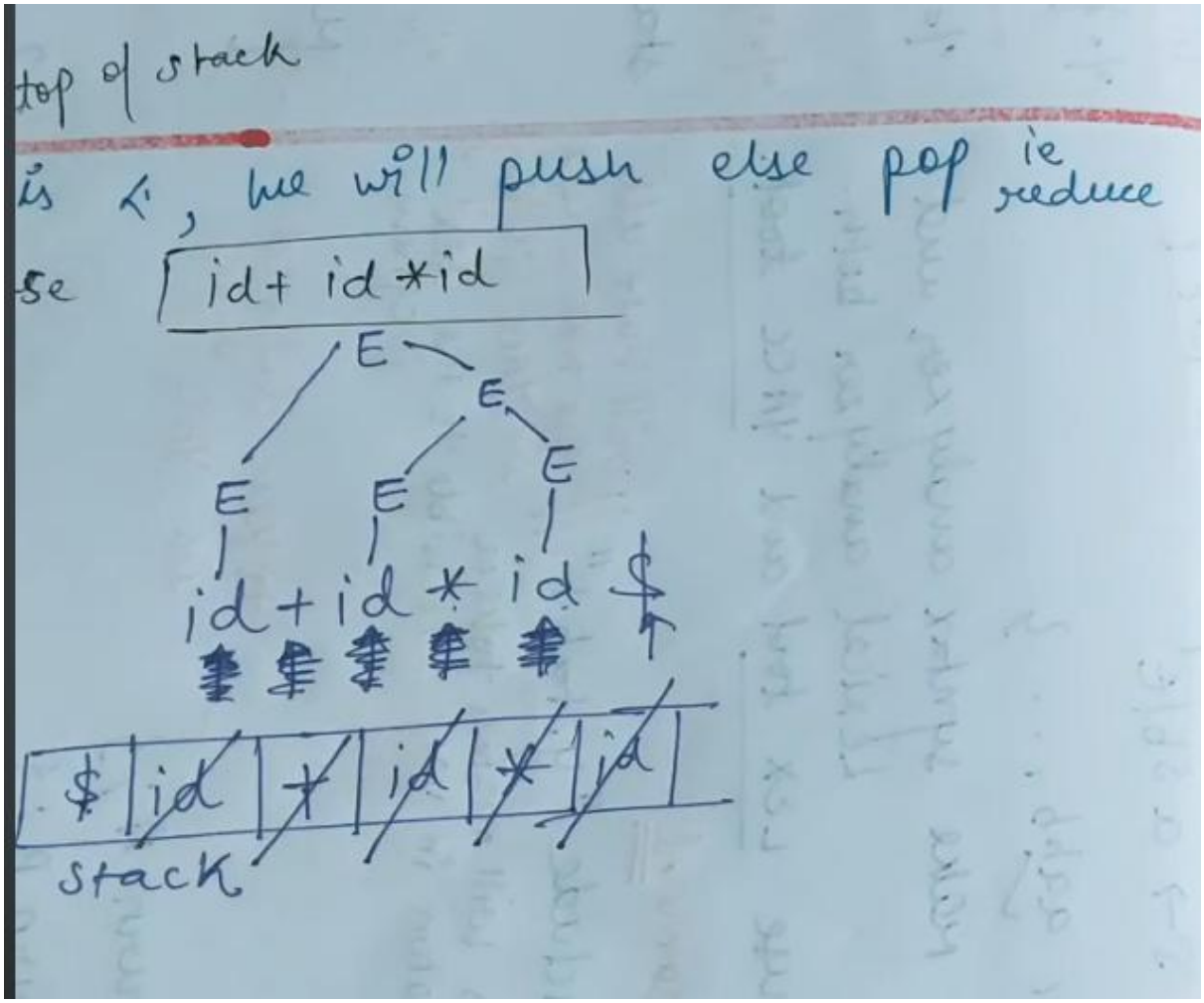
parse the i/p.

Disadvantage of operation Relation Table → no. of enteries

ie if we have 4 operators we will have 16 enteries

" 10 " " " " 100 " size of table will be v. big

for N operators → $O(n^2)$



Method 2 - To reduce the space complexity from n^2 …..

So, to decrease the size of the table, we use operator fn table.

for rows, we will use f and for column g

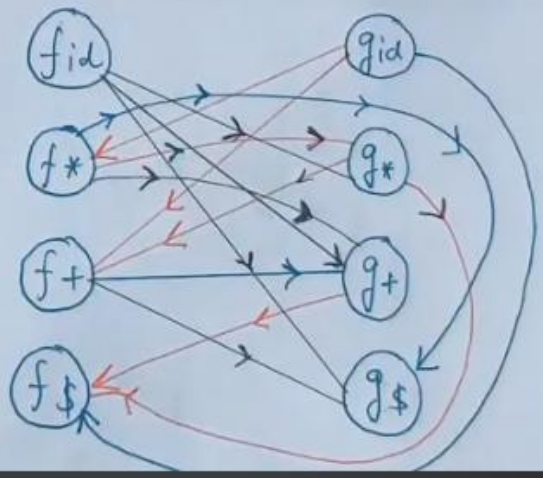
f \ g	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

Now, we will construct a graph

Now, for every operator, we will have fid, gid, f+, g+, f*, g* and g$, f$.

Now, we will construct a graph

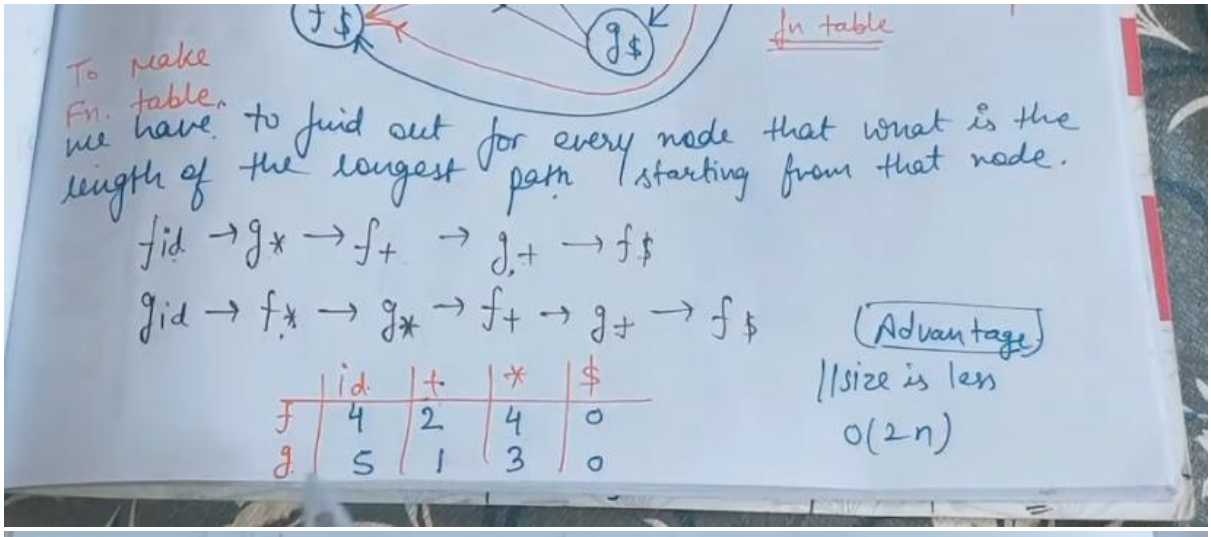
Now, for every operator, we will have fid, gid, f+, g+, f*, g* and g$, f$.



if we found any cycle in the graph, we will stop then and there b/c then we cannot construct a operator fn table

make

How to make arrows - For e.g. you can see fid is greater than g+ so you draw an arrow from fid towards g+



Fn table

To make Fn. table we have to find out for every node that what is the length of the longest path starting from that node.

$fid \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$

$gid \rightarrow f* \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$

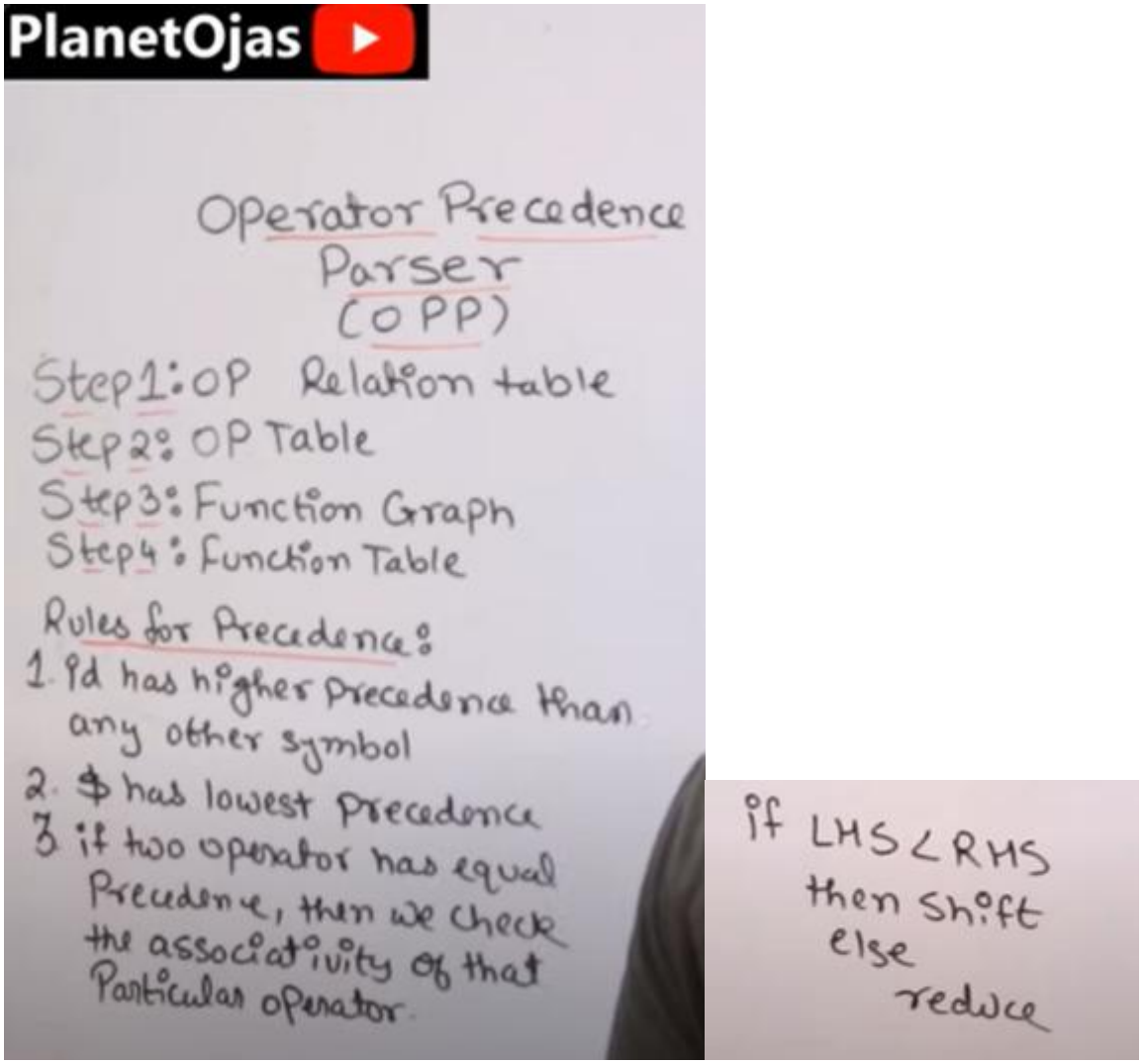
	id	+	*	$
f	4	2	4	0
g	5	1	3	0

(Advantage)
// size is less
$O(2n)$

blank enteries ⟶ errors

Disadvan of using Fn Table
even though we have blank enteries in relation table we get non blank enteries in fn Table

So error ~~dect~~ detecting capability of Fn Table < error detecting capability of relation table


PlanetOjas
Operator Precedence Parser (OPP)
Step 1: OP Relation table
Step 2: OP Table
Step 3: Function Graph
Step 4: Function Table
Rules for Precedence:
1. Id has higher precedence than any other symbol
2. $ has lowest precedence
3 if two operator has equal Precedence, then we check the associativity of that Particular operator
if LHS < RHS
then shift
else
reduce

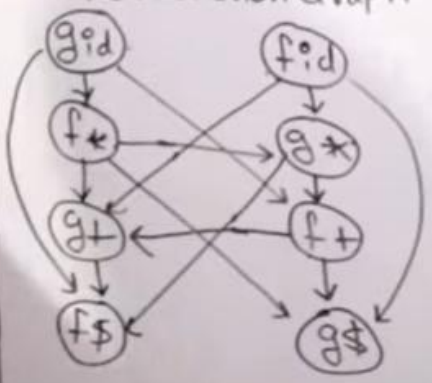
1. $T \to T+T \mid T*T \mid id$

String: id+id*id

Step1: OP Relation Table

f \ g	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	—	>
$	<	<	<	A

Step3: Function Graph

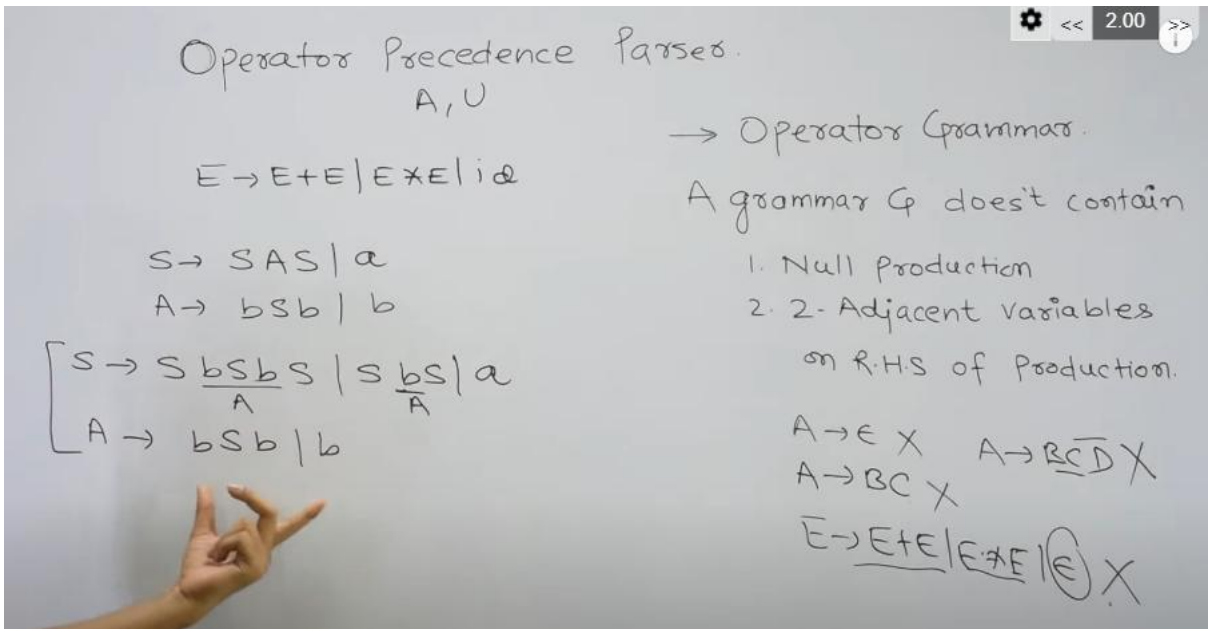


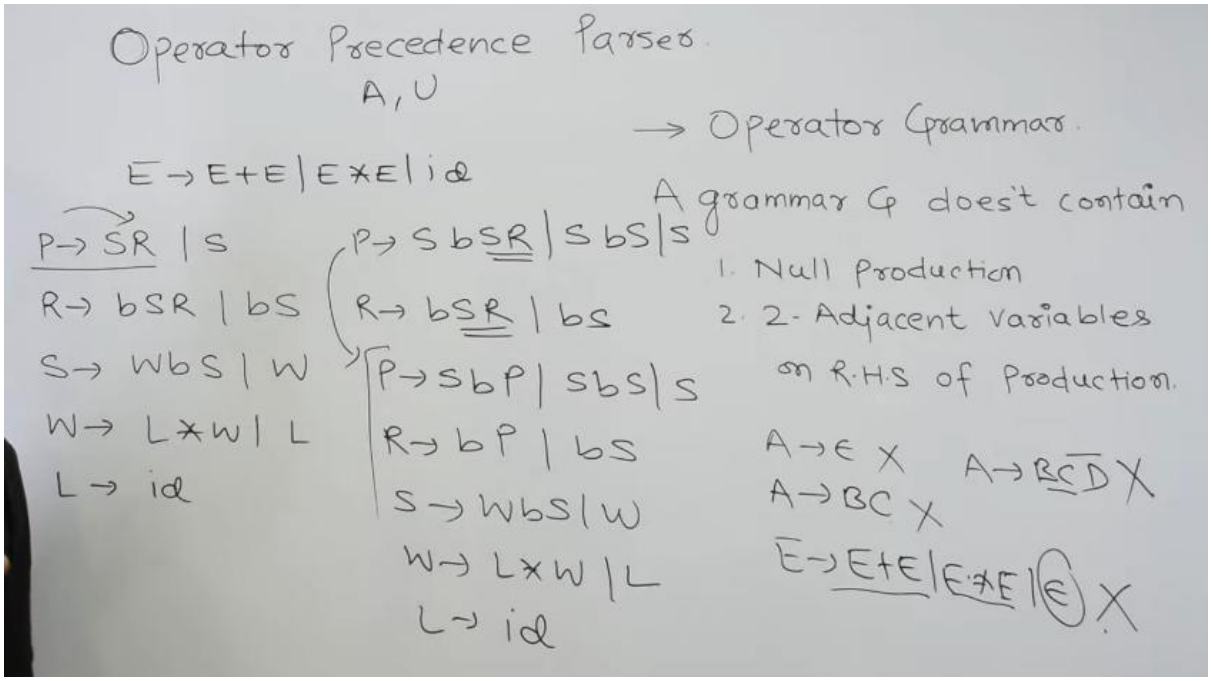
Step2: OP table

Stack	input	comment
$	id+id*id$	Shift
$id	+id*id$	T→id
$T	+id*id$	Shift
$T+	id*id$	shift
$T+id	*id$	reduce T→id
$T+T	*id$	shift
$T+T*	id$	Shift
$T+T*id	$	T→id
$T+T*T	$	T→T*T
$T+T	$	T→T+T
$T	$	A

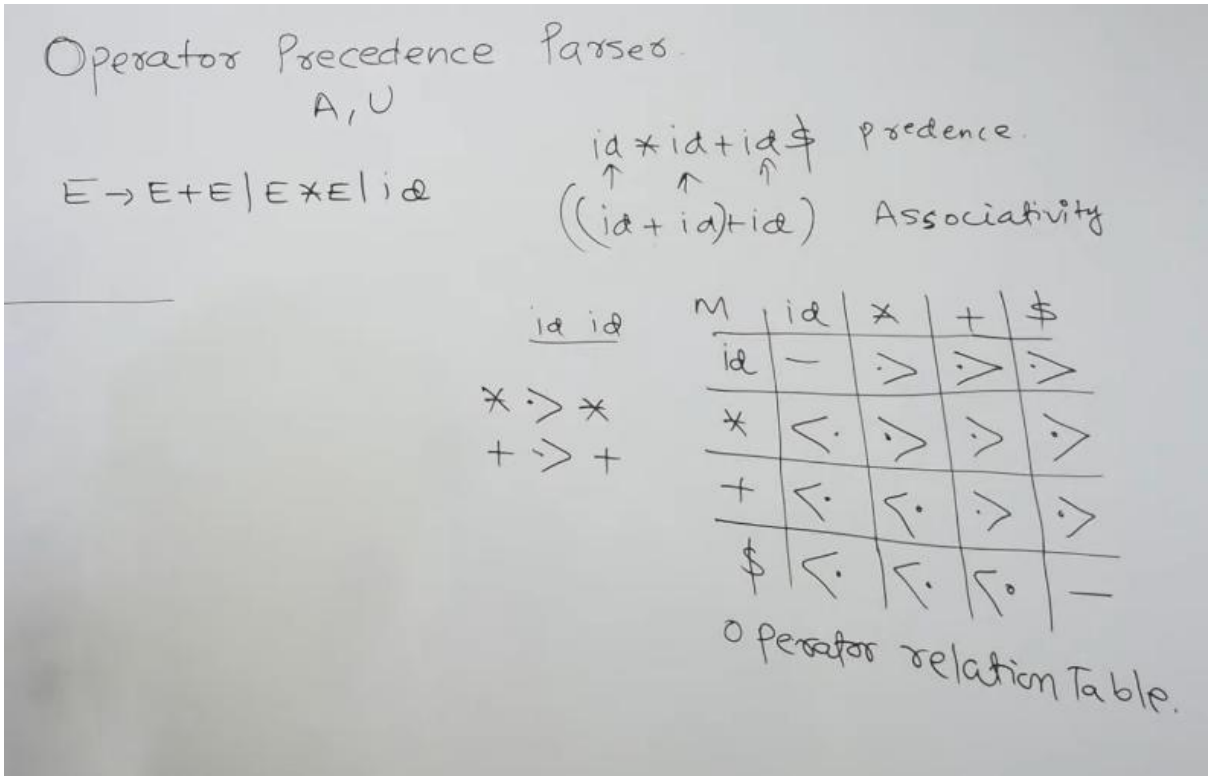
Step4: Table showing longest path

	+	*	id	$
f	2	4	4	0
g	1	3	5	0



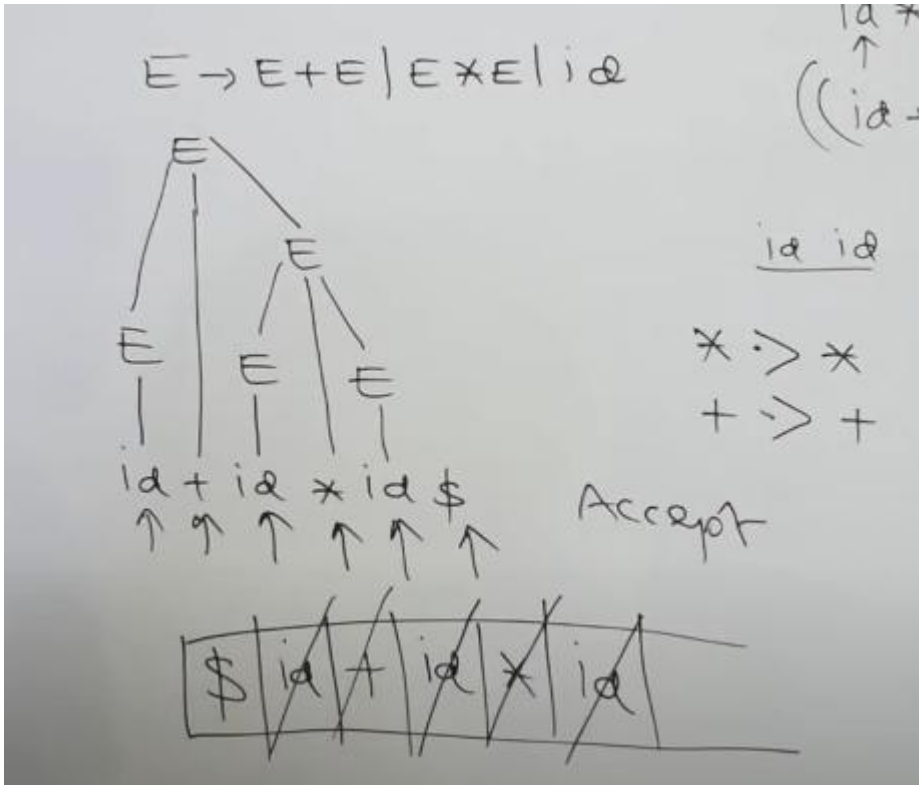


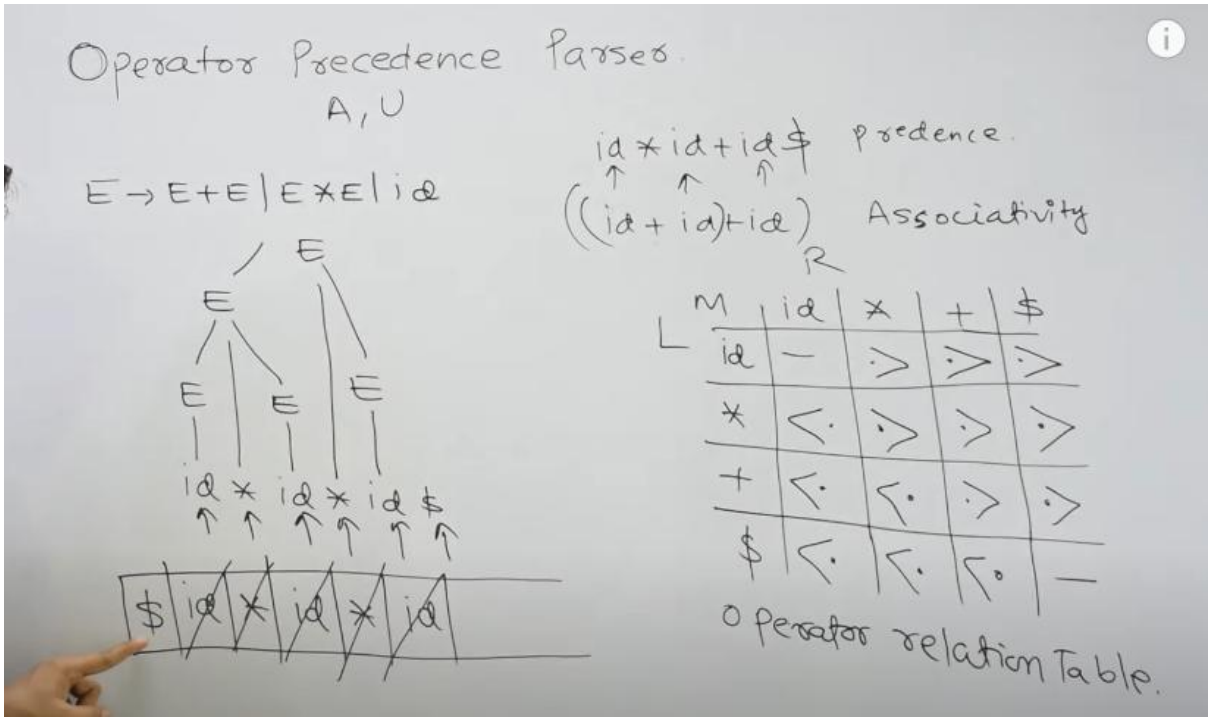
In the below table the symbols written in the rows are kept on the left and the symbols in the columns are kept on the right.



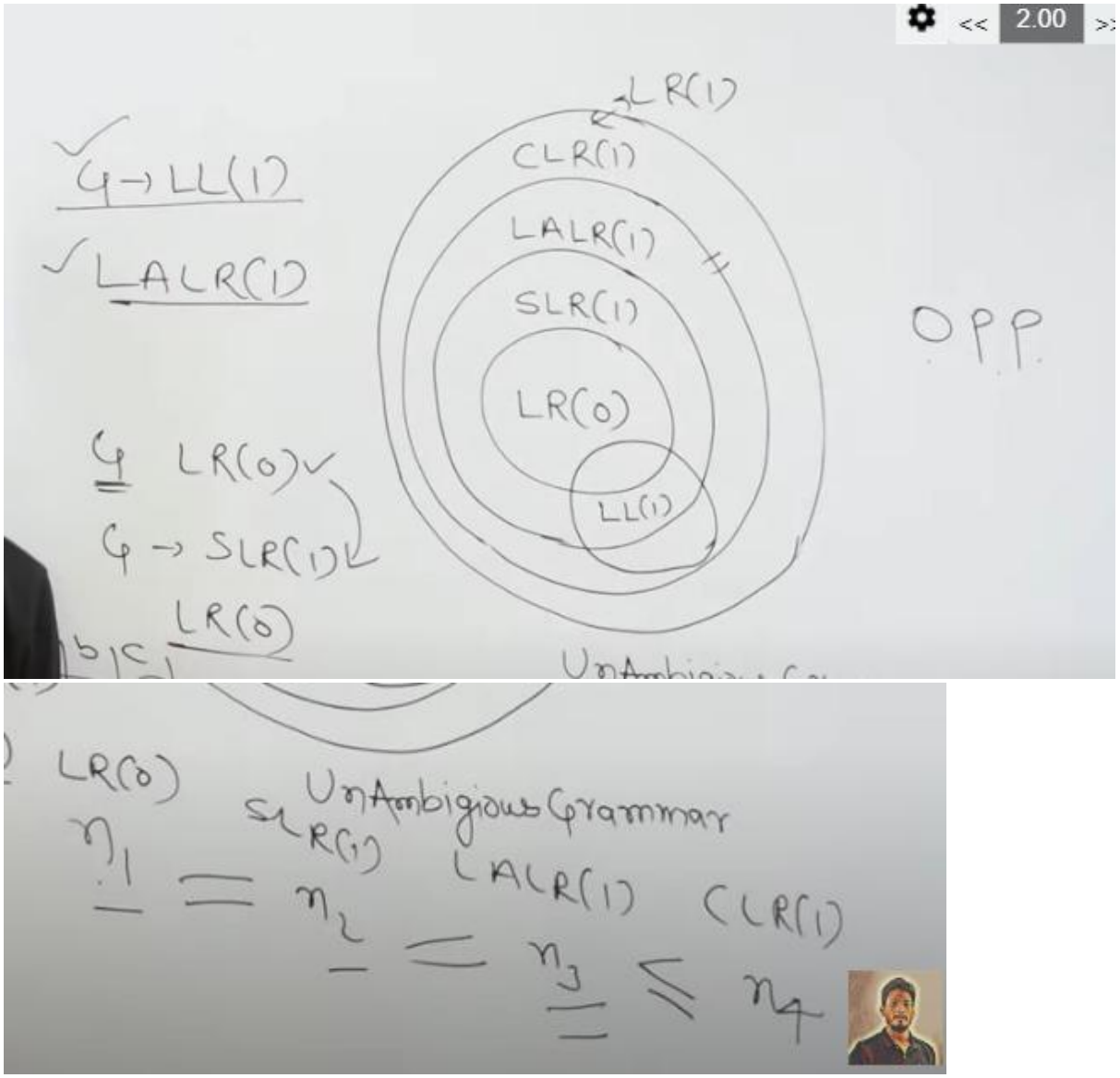
In the stack if an operator has an higher precedence than the operator already present in the stack then only it can be pushed onto the stack, Else if the operator having higher precedence is already present in the stack then it is first popped before an operator with lower precedence is pushed in the stack.

When we push a symbol onto the stack then only the pointer moves forward else if we pop the pointer remains where it was.





Relationship between LL(1), SLR(1), LALR(1), CLR(1) and LR(1) Parsers



Parser using YACC tool

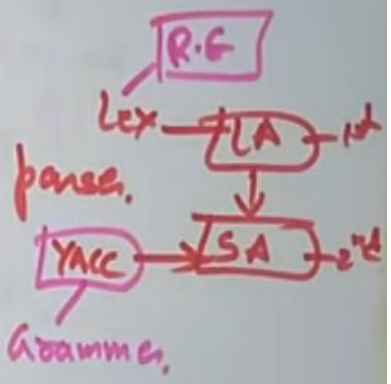
YACC ⇒ Yet Another Compila Compiler

lex — Lexical Analyzer generator

YACC — Parser generator.

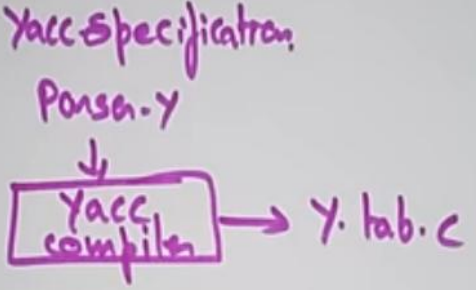
⇓

It is a tool which generate LALR parser.



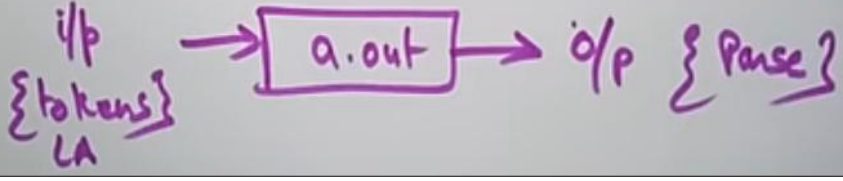
YACC Working

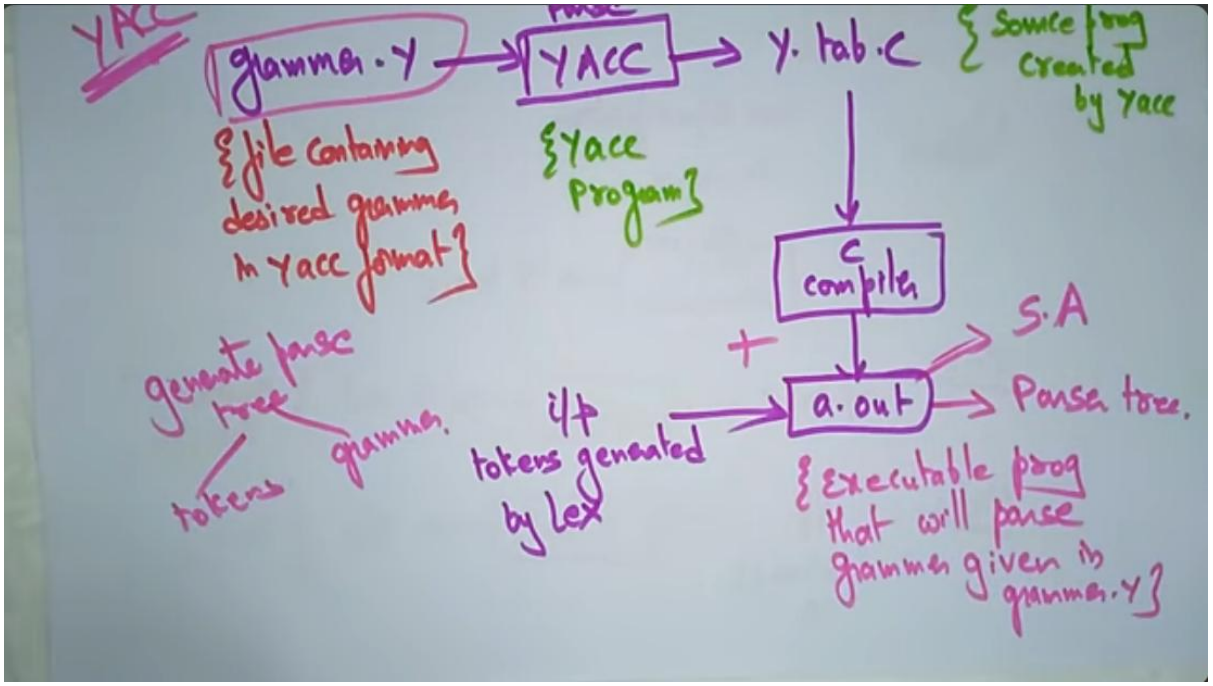
Step 1 :



Step 2:

Step 3:





Syntax:

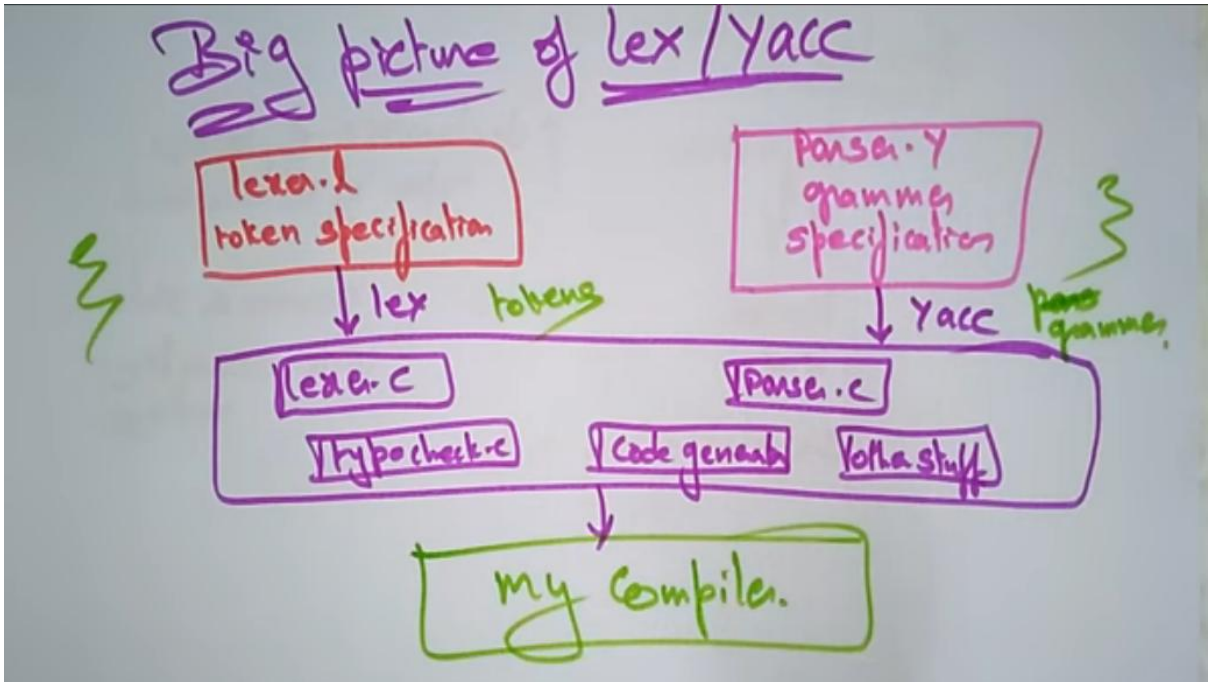
definitions { declaration of tokens, types of values used

%%

Rules { list of grammar rules with semantic routines

%%

Supplementary code

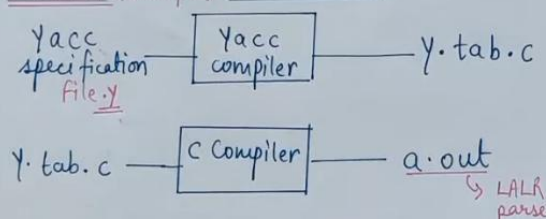


YACC IN COMPILER DESIGN

Yt stands for Yet Another Compiler-Compiler (developed by Stephen C. Johnson)

- Yt is a tool for generating Look Ahead Left-to-Right (LALR) parser.
- Yt takes i/p from the Lexical Analyzer & generates parse Tree.
- Syntax Analyzer / Parser is the 2nd phase of the compiler which takes i/p as tokens and generates a parse tree.

WORKING (3 steps) / BLOCK DIAGRAM



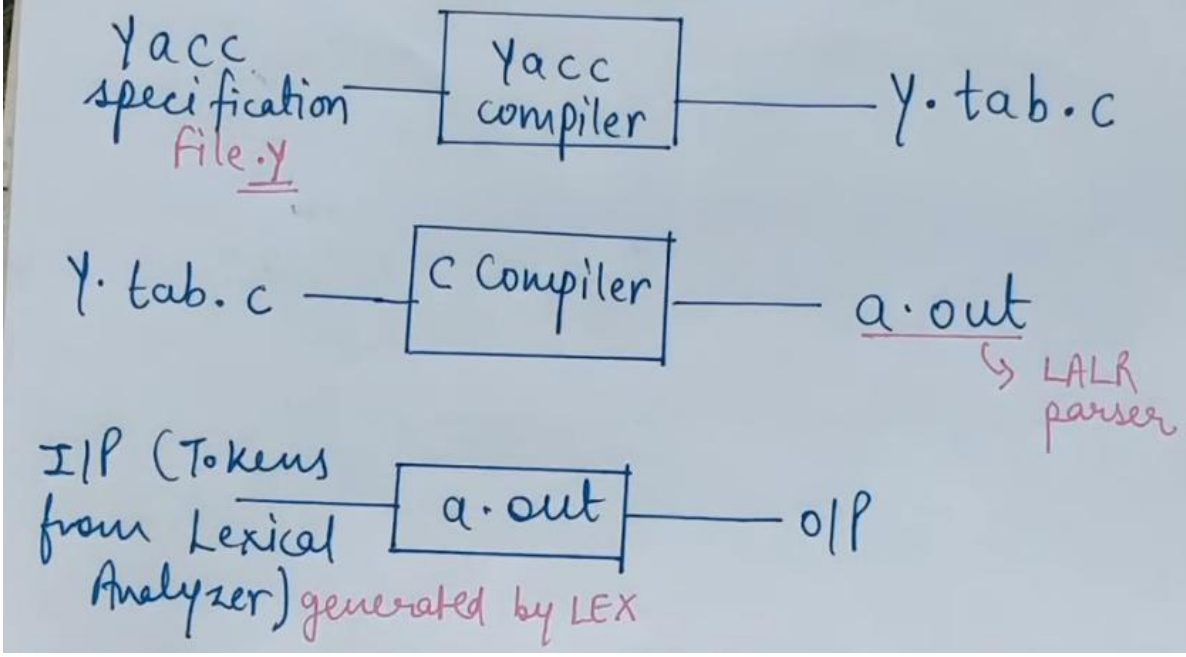
WORKING

1) I/p to the Yacc compiler will be a file with .y extension. Yt will contain desired grammar in Yacc format. YACC compiler will convert it into a C code in the form of y.tab.c file.

2) This y.tab.c file will be given as a input to the C Compiler and the output will be the LALR parser (ie. a.out)

3) Tokens generated by the lexical analyzer (using the lex tool) will be given as a input to a.out ie. our LALR parser and we will get the parse tree as output.

WORKING (3 steps) / BLOCK DIAGRAM



WORKING

1) I/P to the Yacc compiler will be a file with .y extension. It will contain desired grammar in Yacc format. YACC compiler will convert it into a C code in the form of y.tab.c file.

2) This y.tab.c file will be given as a input to the C compiler and the output will be the LALR(1) parser (ie. a.out)

3) Tokens generated by the lexical analyzer (using the lex tool) will be given as a input to a.out ie. our LALR parser and we will get the parse tree as output.

SYNTAX

Join Telegram channel for regular video updates or Join FB/Insta for all video updates
Link in description :)

definitions / declarations
%% tokens
%%
Rules head : body1 {action} | body2 {action2}
%%
Auxiliary Routines / Supplementary Code → C code

To design program in Yacc lang,

1) **YYval** - values associated with the tokens are returned by lex in the variable yyval.

suppose we have i/Ped a no. & we want to extract it - But our i/p is in character stream format.

2) **yyval = atoi(yytext)**

Convert string to int
ie converts i/p to numeric and stores in a variable

3) **yytext** - pointer to the i/p character stream / matched i/p string

(small y)

4) **yywrap()** → called by lex or yacc when i/p is exhausted / finished (return 1 when i/p finished).

5) **yyparse()** - responsible for parsing to occur. It reads tokens and executes the actions. (which we will give in our yacc program)

(if it gives 0 means string accepted)

Does check aaabbb is present or Not ?

$S \rightarrow a\,S\,b \mid \epsilon$

eg $L = \{ ab, aabb, \ldots \}$

we have to make syntax analyzer and Lexical analyzer both.

So, we will use LEX tool and YACC tool.

Step 1

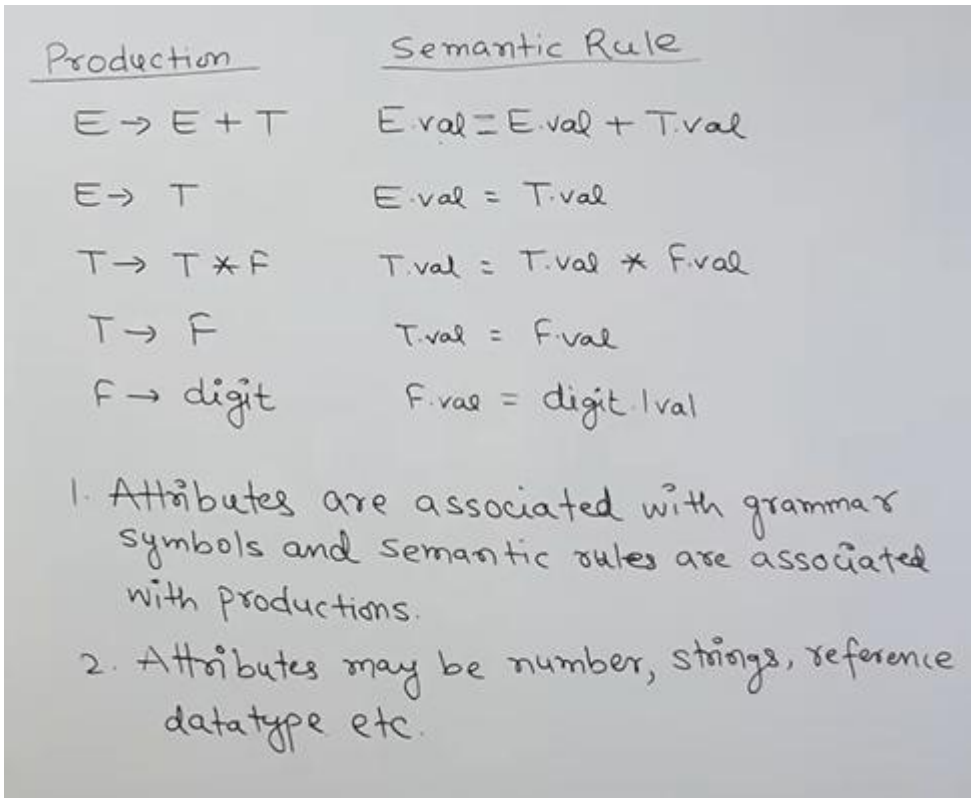
LEX filename.l

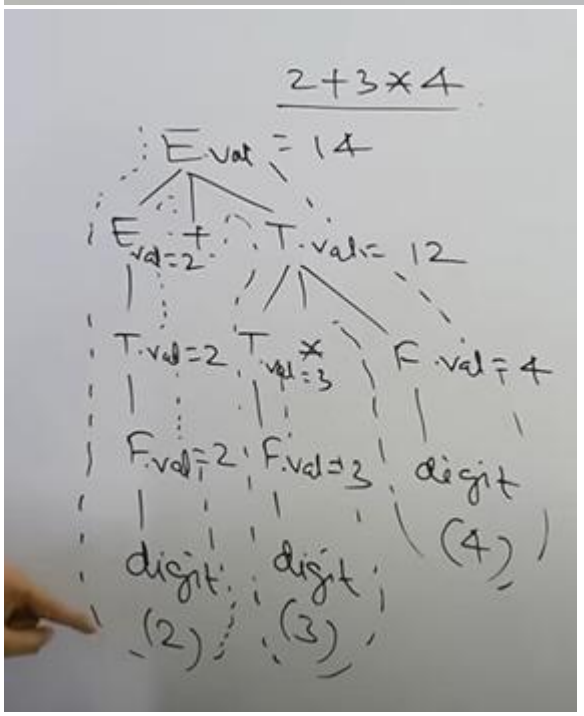
```
%{
#include "y.tab.h" // will have all the tokens used in YACC file/prog
This will match token declaration in yacc accessible to the lex program
%}
%%
[a]   {return A;}      token defined in YACC file
[b]   {return B;}
[\n]  {return 0;}
%%
/* main fn will not be included here */
```

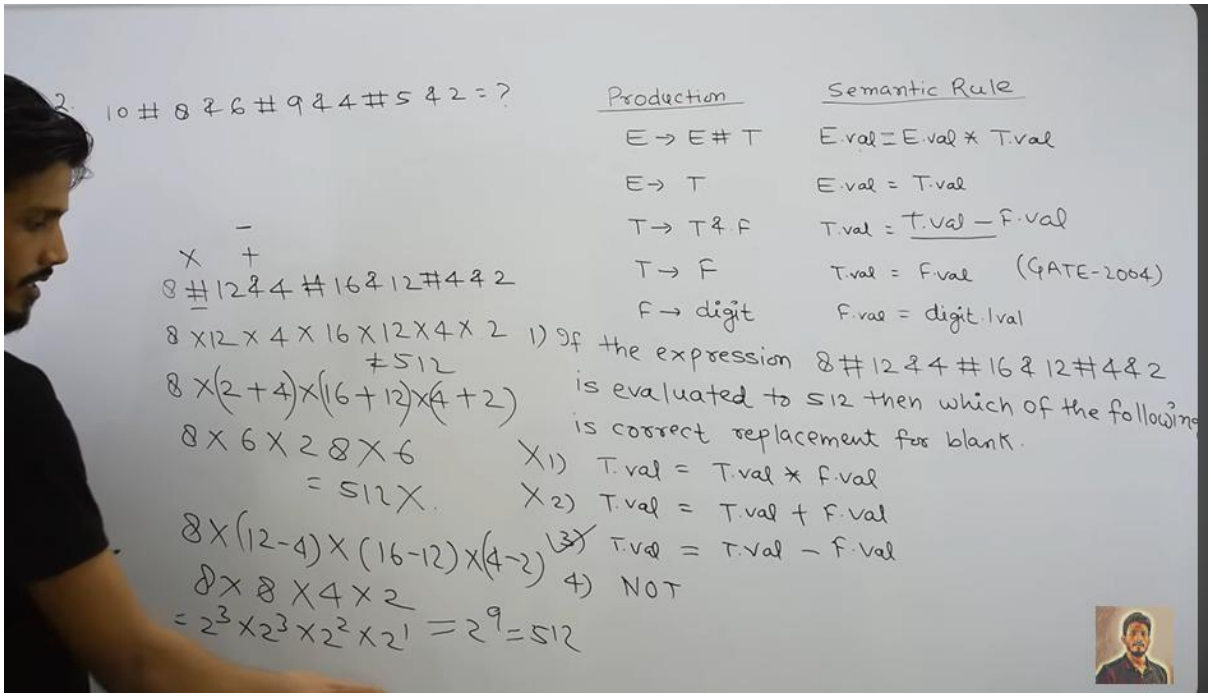
YACC part : prog.y file

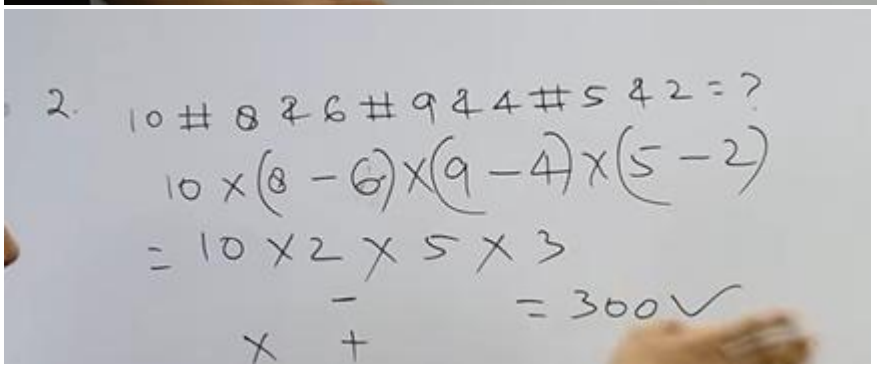
```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token A B          // tokens defined in YACC
%%
start : S '\n' {return 0;} // verification step
    S : A S B
      | ;
%%
main()
{ printf("enter string ");
    if (yyparse() == 0)
        printf("valid string");
}
yyerror() // if string not accepted this is called
{ printf("not accepted");
    exit(0);
}
int yywrap() // will be called when i/p is exhausted.
{ return 1;
}
```

Introduction to Syntax Directed Definition (SDD)| Syntax Directed Translation









S-Attributed and L-Attributed SDD | Types of SDD

- **Synthesized Attribute**: It is an attribute whose value at a node in the parse tree is determined by the attribute values at its **child nodes**.
- **Inherited Attribute**: It is an attribute whose value at a node in the parse tree is determined by the attribute values at its **parent and/or sibling nodes.**

Types of SDD

S- Attributed

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

Ex -

$A \rightarrow BCD$ { b }

$A.i = B.i$

$A.i = C.i$

$A.i = D.i$

2. Semantic actions are always placed at right end of the production. (Postfix SDD)

3. Attributes are evaluated with BUP.

L- Attributed

1. A SDD that uses both synthesized and inherited attributes is called as L-Attributed SDD but each inherited attribute is restricted to inherit from Parent or Left Siblings only.

Ex - $A \rightarrow B\ \{\ \}\ CD$

$A \rightarrow \{\ \}\ BCD$

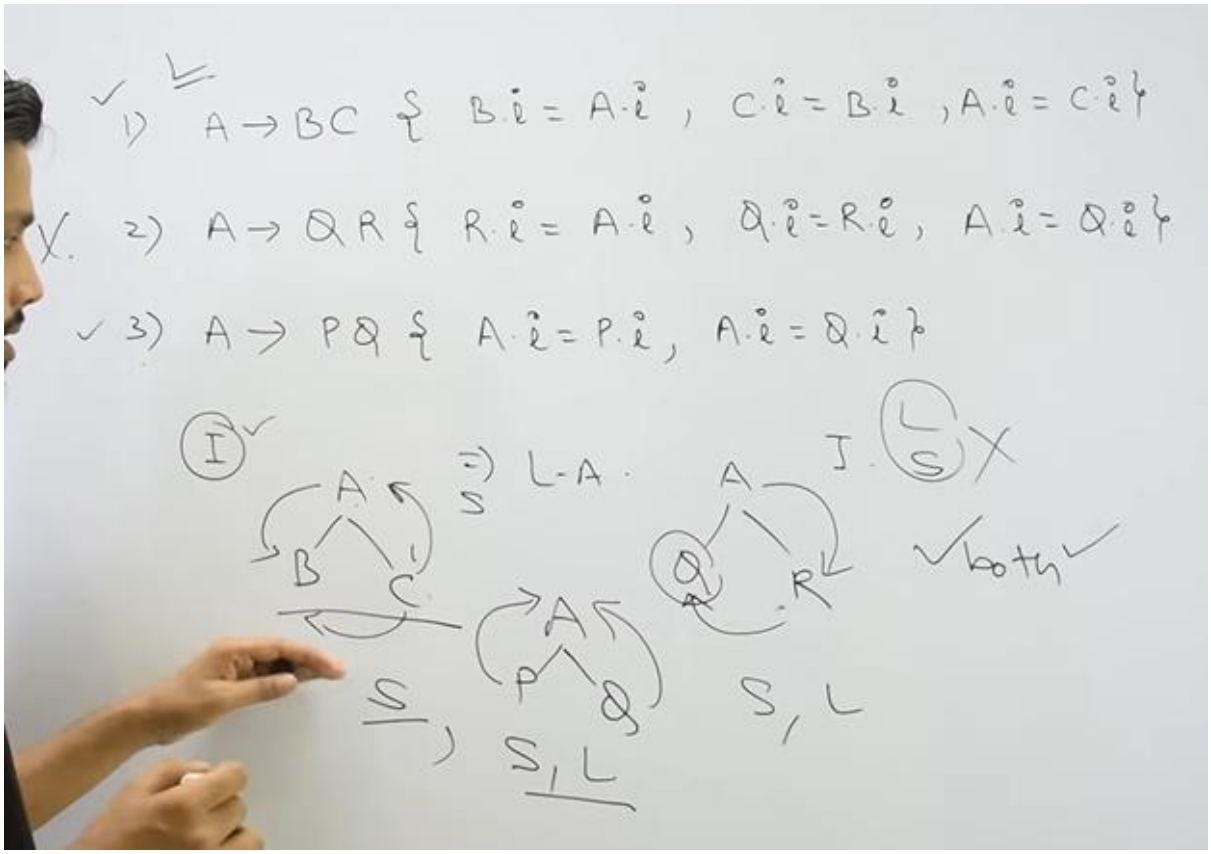
$A \rightarrow BCD$ { b } $C.i = A.i$

$C.i = B.i$

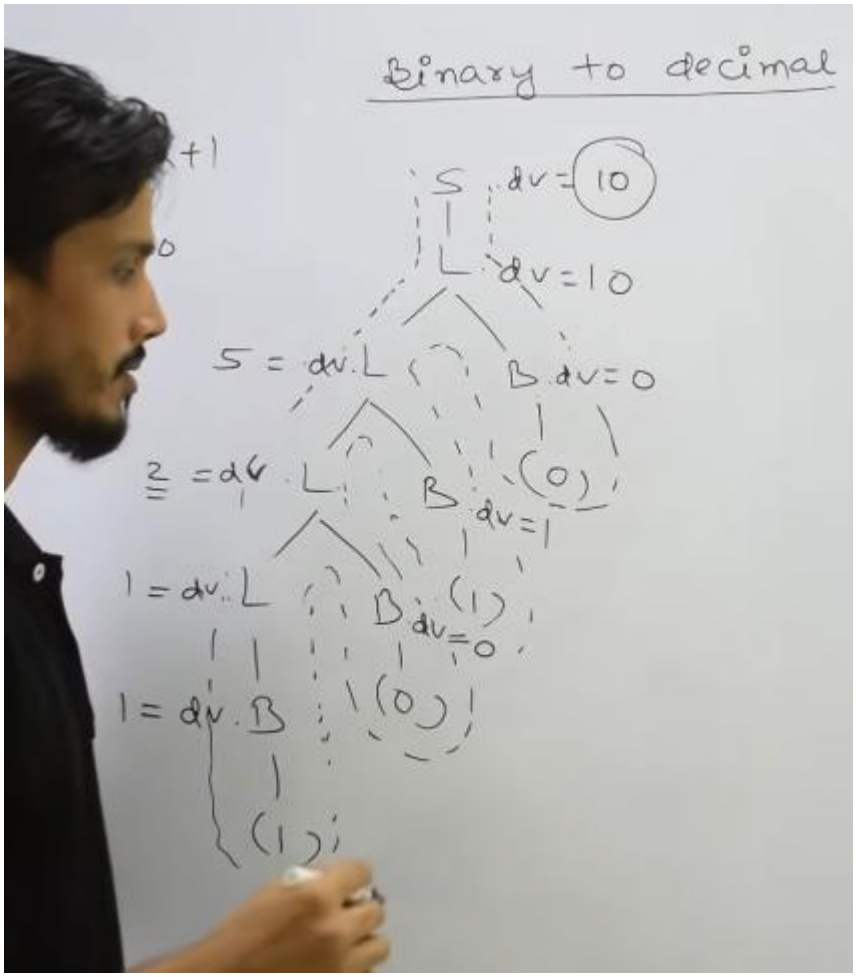
$C.i = D.i$

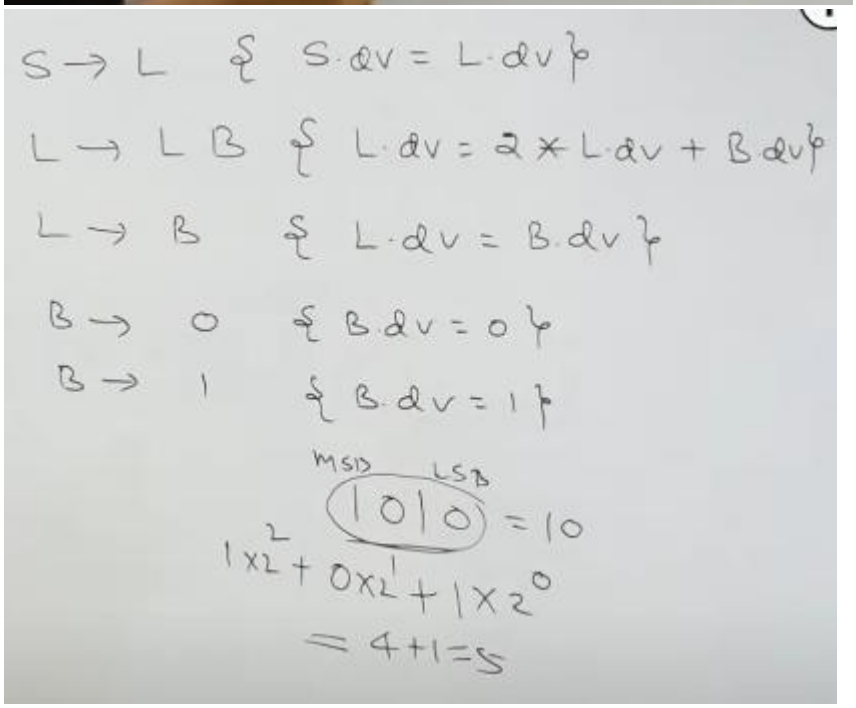
2. Semantic action are placed anywhere on R.H.S of the production.

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

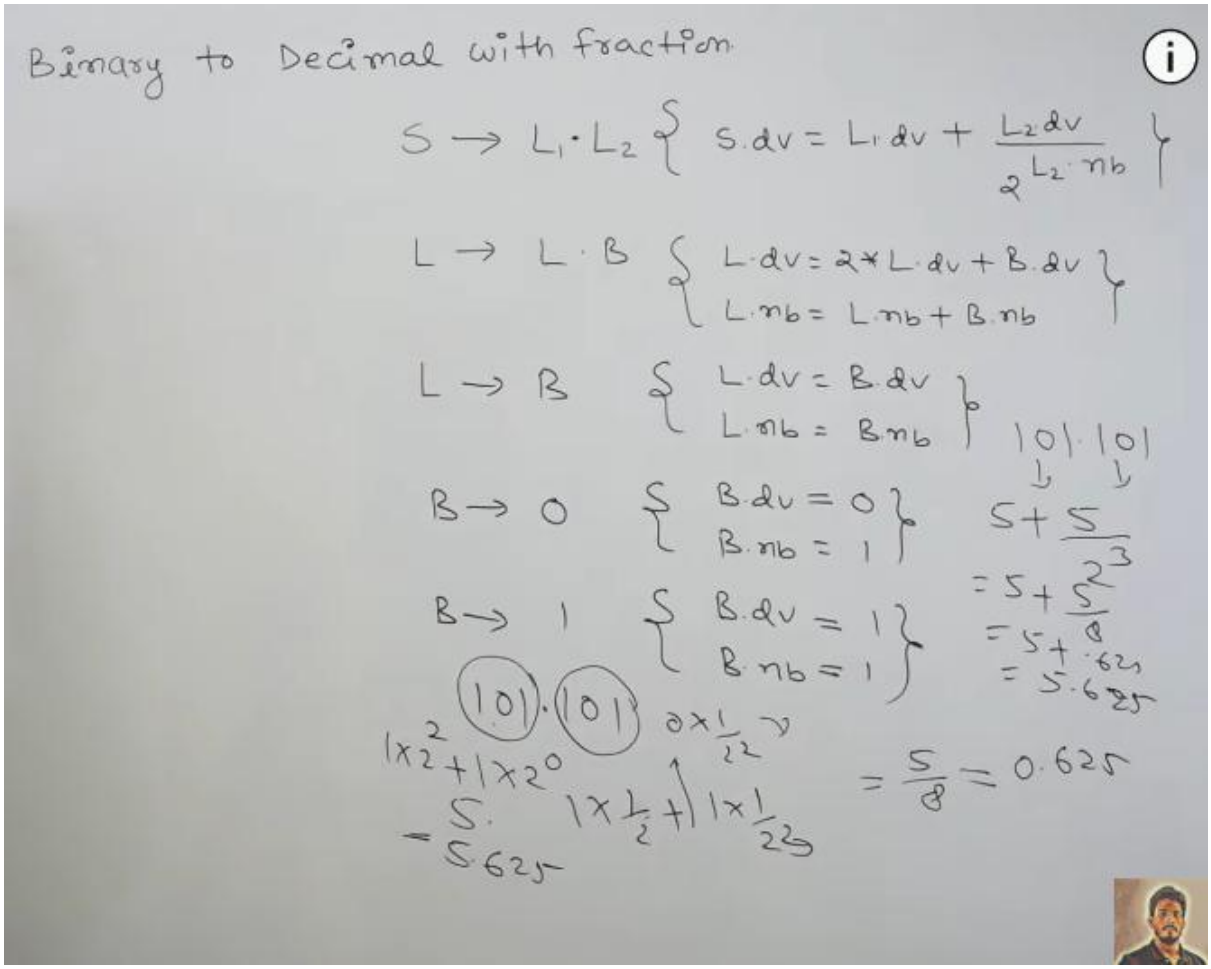

1) A → BC { B.i = A.i , C.i = B.i , A.i = C.i }
2) A → QR { R.i = A.i , Q.i = R.i , A.i = Q.i }
3) A → PQ { A.i = P.i , A.i = Q.i }
I
A
B
C
S
L-A
A
Q
R
J
L
S
both
A
P
Q
S
S, L
S, L

SDD to Convert Binary to Decimal

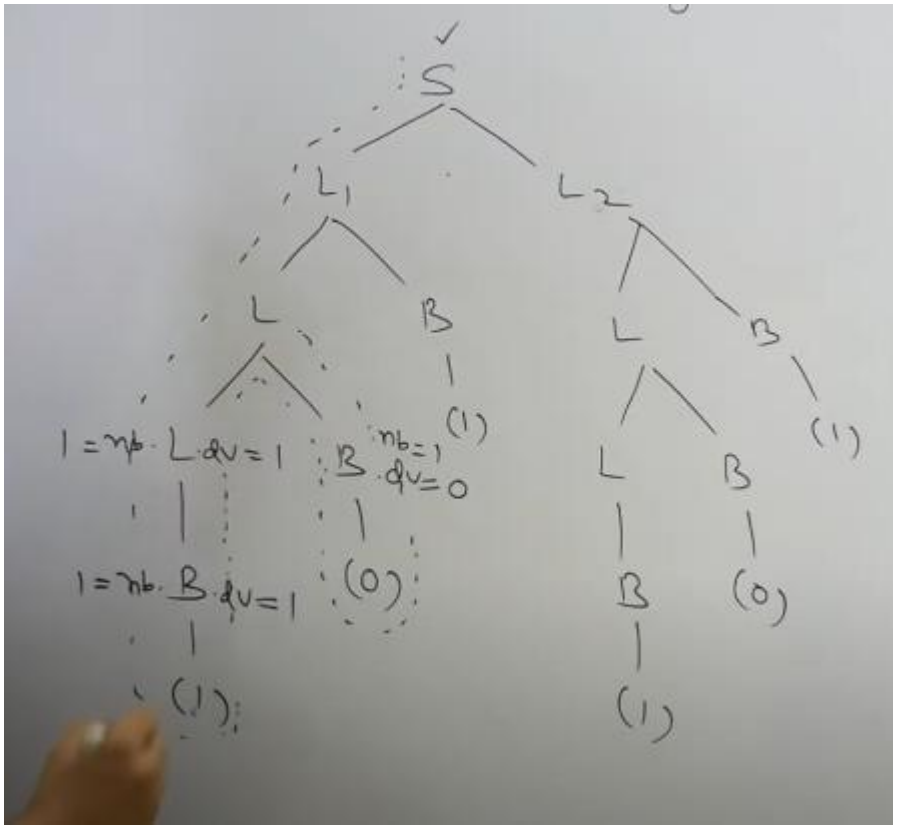


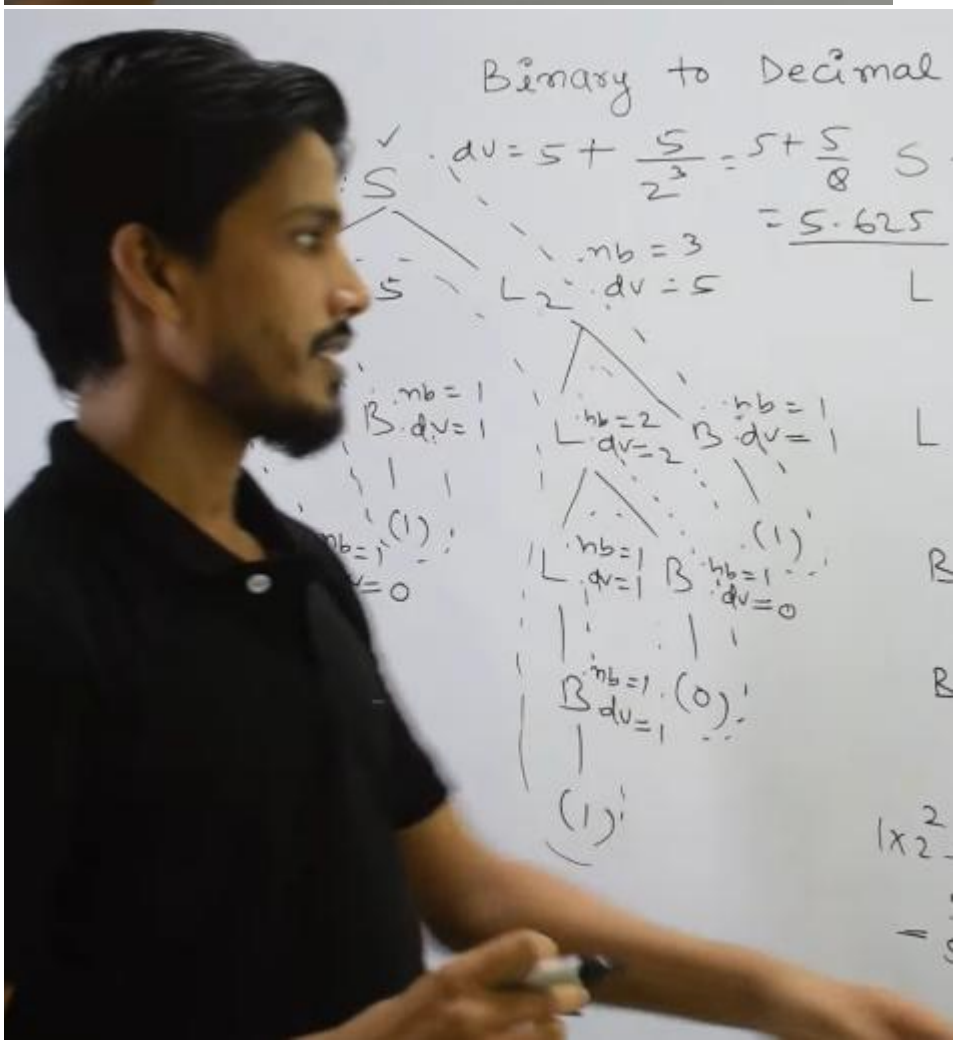


SDD to Binary to Decimal with Fraction

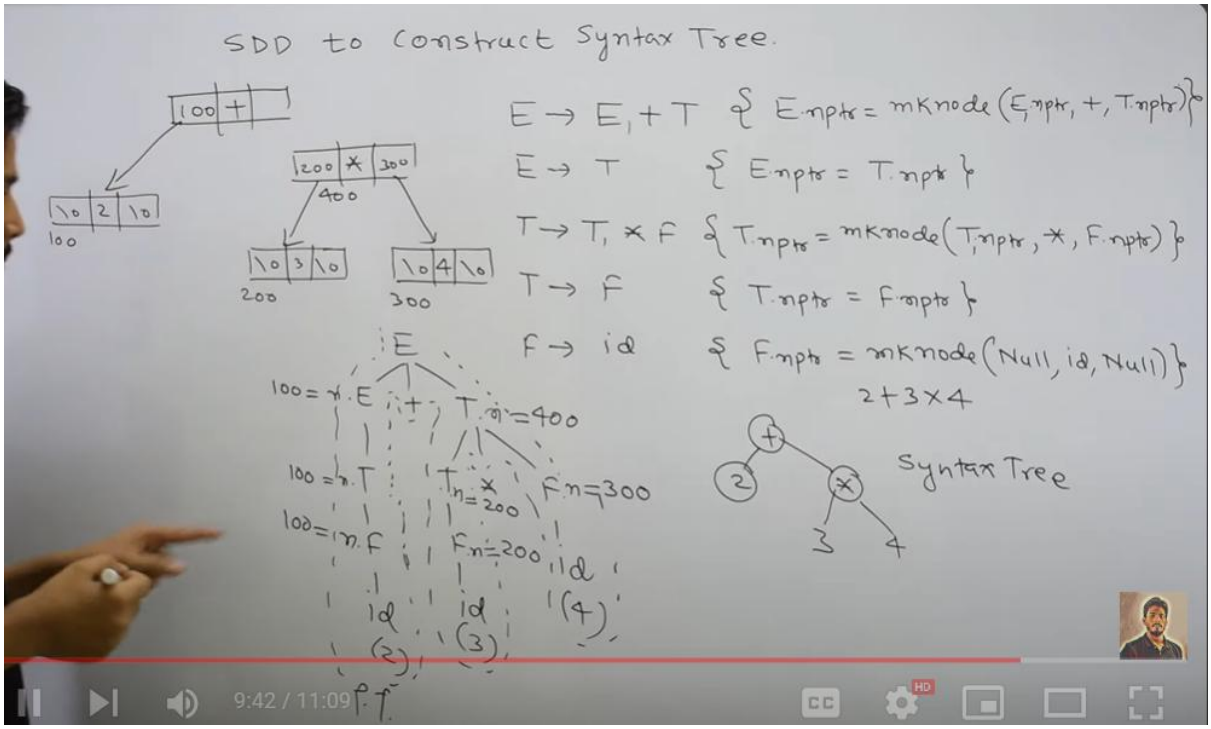


nb = number of bits

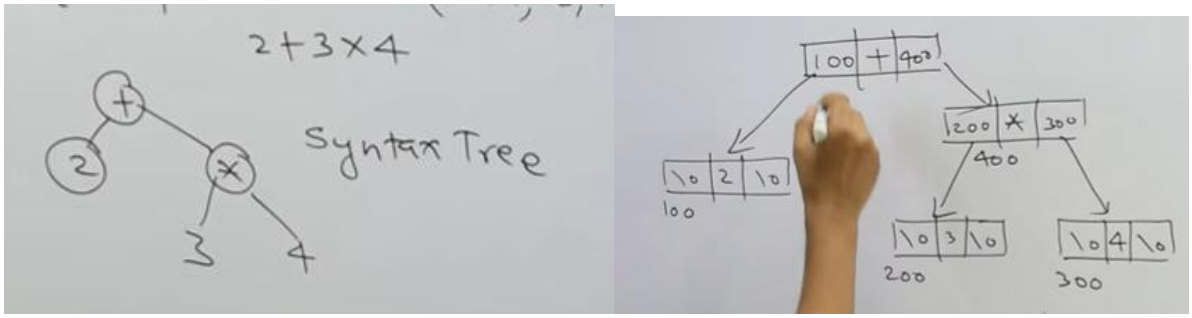

S
L1
L2
L
B
L
B
1 = nb
L dv = 1
B dv = 0
nb = 1
(1)
(0)
1 = nb
B dv = 1
(1)
L
B
(1)
L
B
(0)
(1)


Binary to Decimal
dv = 5 + 5/2³ = 5 + 5/8
= 5.625
S
L2
dv = 5
nb = 3
B dv = 1
nb = 1
L dv = 2
nb = 2
B dv = 1
nb = 1
L dv = 1
nb = 1
B dv = 0
nb = 1
B dv = 1
(1)
(0)
(1)
(1)
L
L
B
B
1x2²

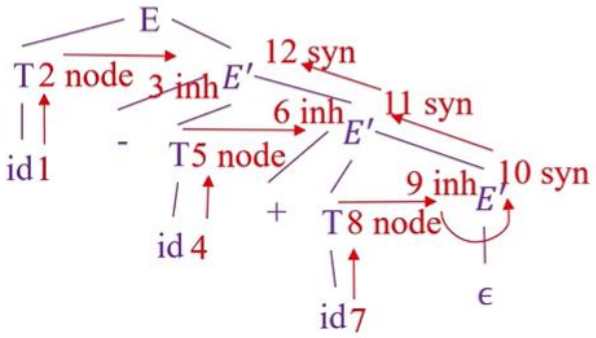
SDD to construct syntax tree

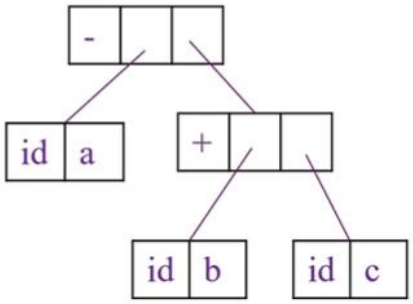


Here E1 and T1 denotes the right hand side production and 100,200 and 300 are just randomly assumed addresses for those locations.

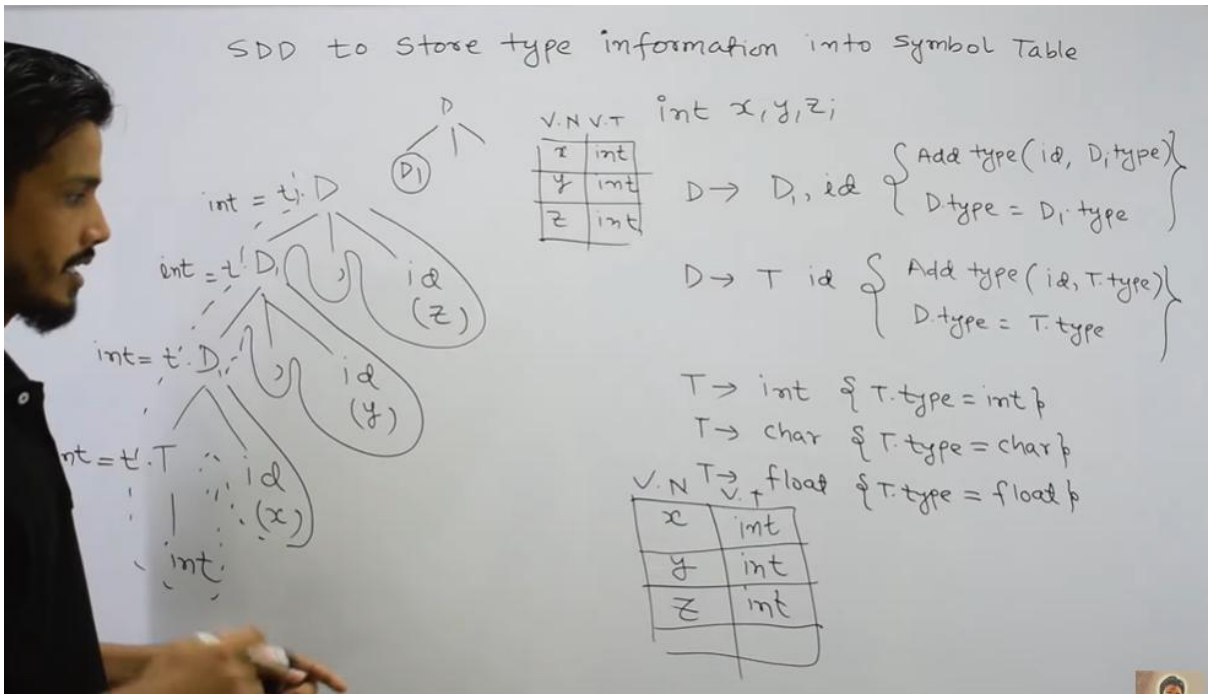


- Ex 8:Syntax tree construction
- **Production** **Semantic Rules**
- E →TE' {E.node=E'.syn ,E'.inh=T.node }
- E' →+TE'_1 {E'_1.inh=new Node('+', E'.inh,T.node), E'.syn= E'_1.syn}
- E' →-TE'_1 {E'_1.inh=new Node('-', E'.inh,T.node), E'.syn= E'_1.syn}
- E' → ϵ {E'.syn= E'_1.inh}
- T →(E) {T.node=E.node}
- T →id {T.node=new Leaf(id , id.val)}
- Inherited attribute inh ,Synthesized attribute syn.L-Attributed SDT
- Syntax tree for a-b+c.

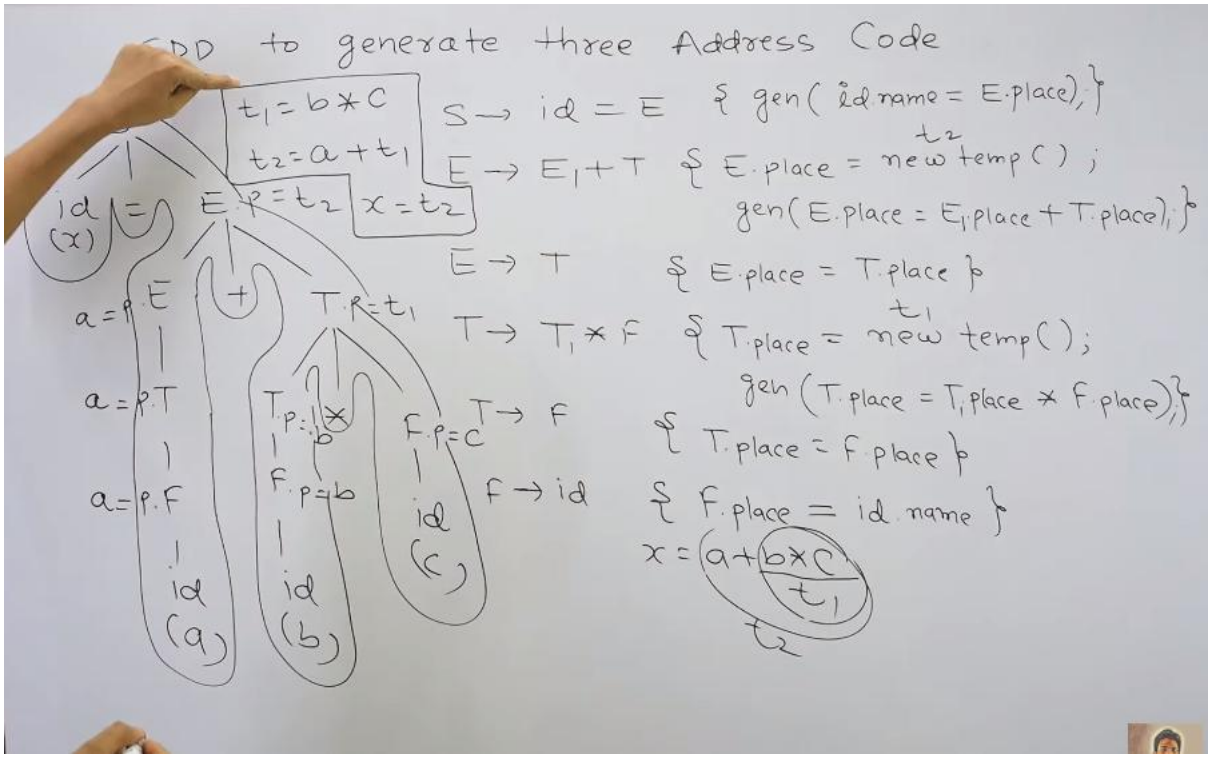


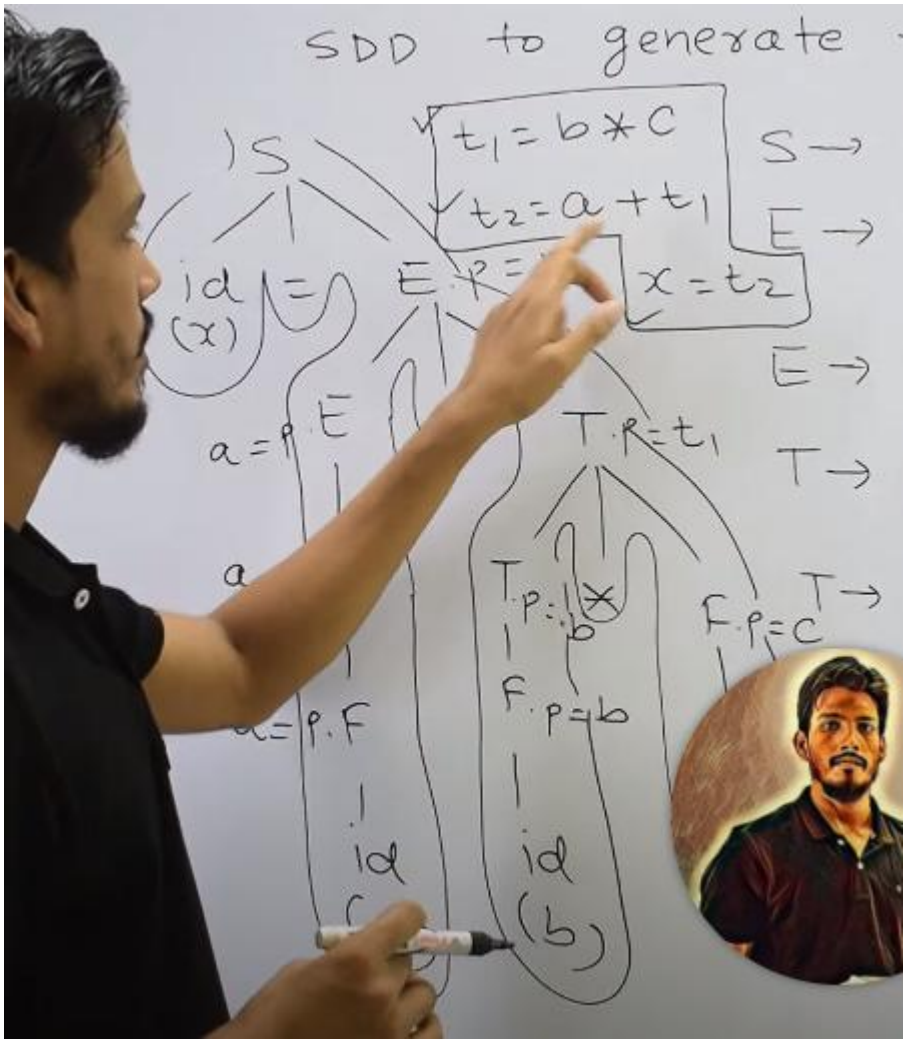


SDD to store type information in symbol table

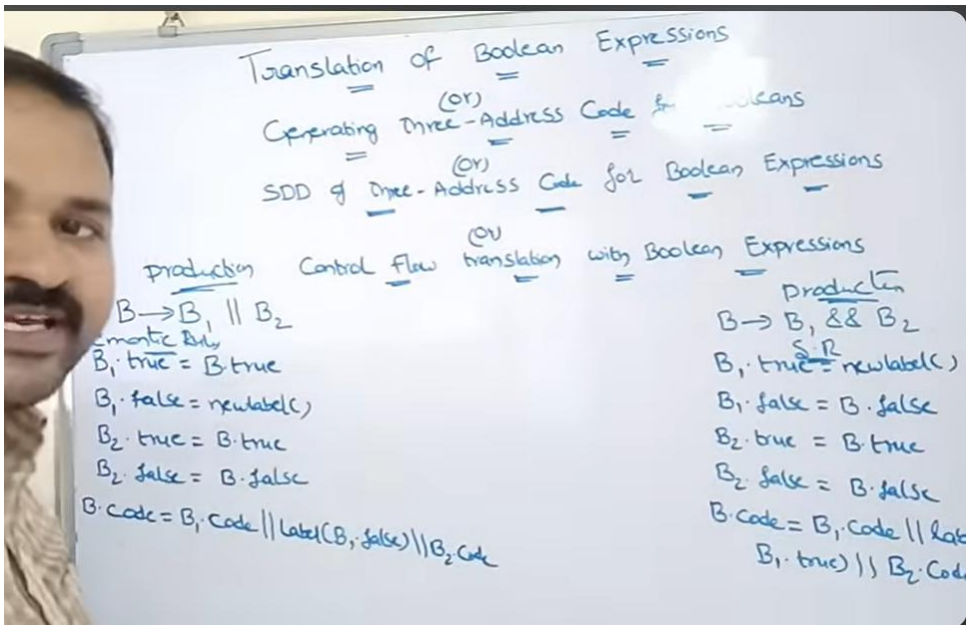


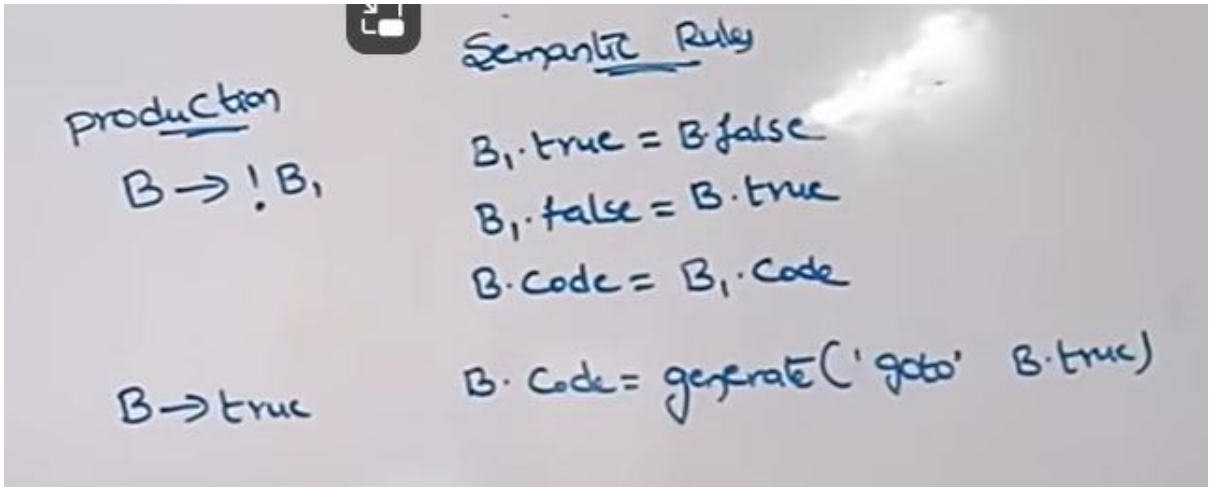
SDD to generate Three Address Code



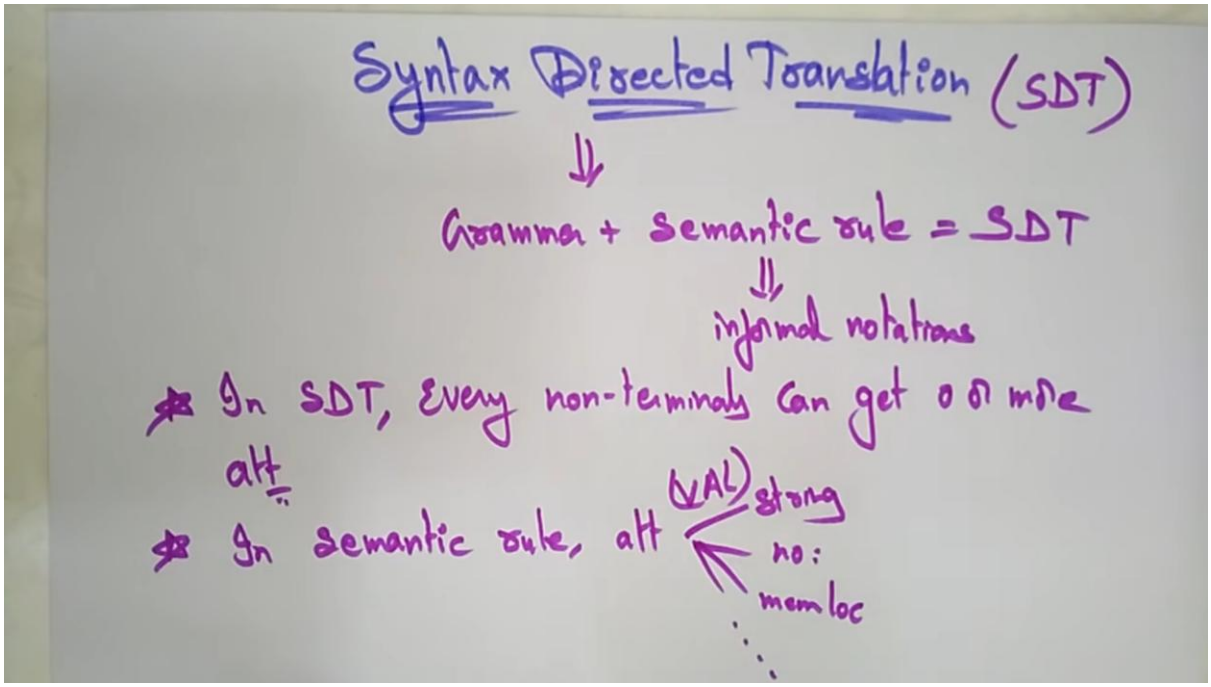


SDD to generate 3 address code for booleans





Syntax Directed Translation | SDT



Eg SDT

Production	{Actions} Semantic rule
$E \rightarrow E+T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T*F$	$T.val := T.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow Num$	$F.val := num.lexval$

⇓

att return by LA.

Syntax Directed Translation scheme

SDT = Grammar + Semantic rule

- The Syntax Directed Translation scheme is a context free grammer.
- It is used to evaluate the order of Semantic rules
- In translation scheme, the semantic rules are embedded within the right side of the production.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production

Example:

Production	Actions / Semantic Rules
$S \rightarrow E\$$	{ Print E.val }
$E \rightarrow E+E$	{ E.val := E.val + E.val }
$E \rightarrow E*E$	{ E.val := E.val * E.val }
$E \rightarrow (E)$	{ E.val := E.val }
$E \rightarrow I$	{ E.val := I.val }
$I \rightarrow I$ digit	{ I.val := 10 * I.val + Lexval }
$I \rightarrow$ digit	{ I.val := Lexval }

augment grammer.

Implementation of Syntax directed translation

* STD is implemented by constructing a parse tree and performing the actions in a left to right depth first order.
* STD is implementing by parse the input and a parse tree as a result

Example:

grammer Actions { Semantic rules

E → E+T { E.value = E.val + T.val }

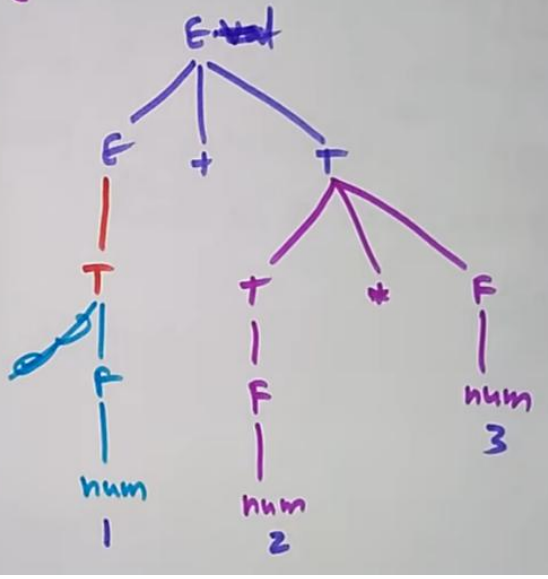
E → T { E.val := T.val }

T → T * F { T.val := T.val * F.val }

T → F { T.val := F.val }

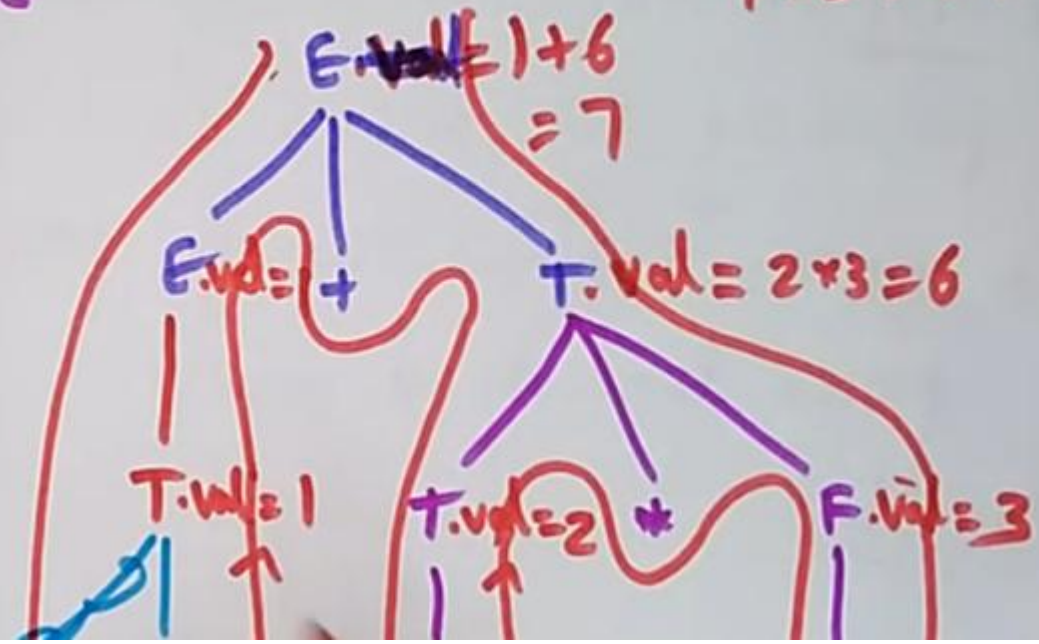
F → num { F.val := num.lval }

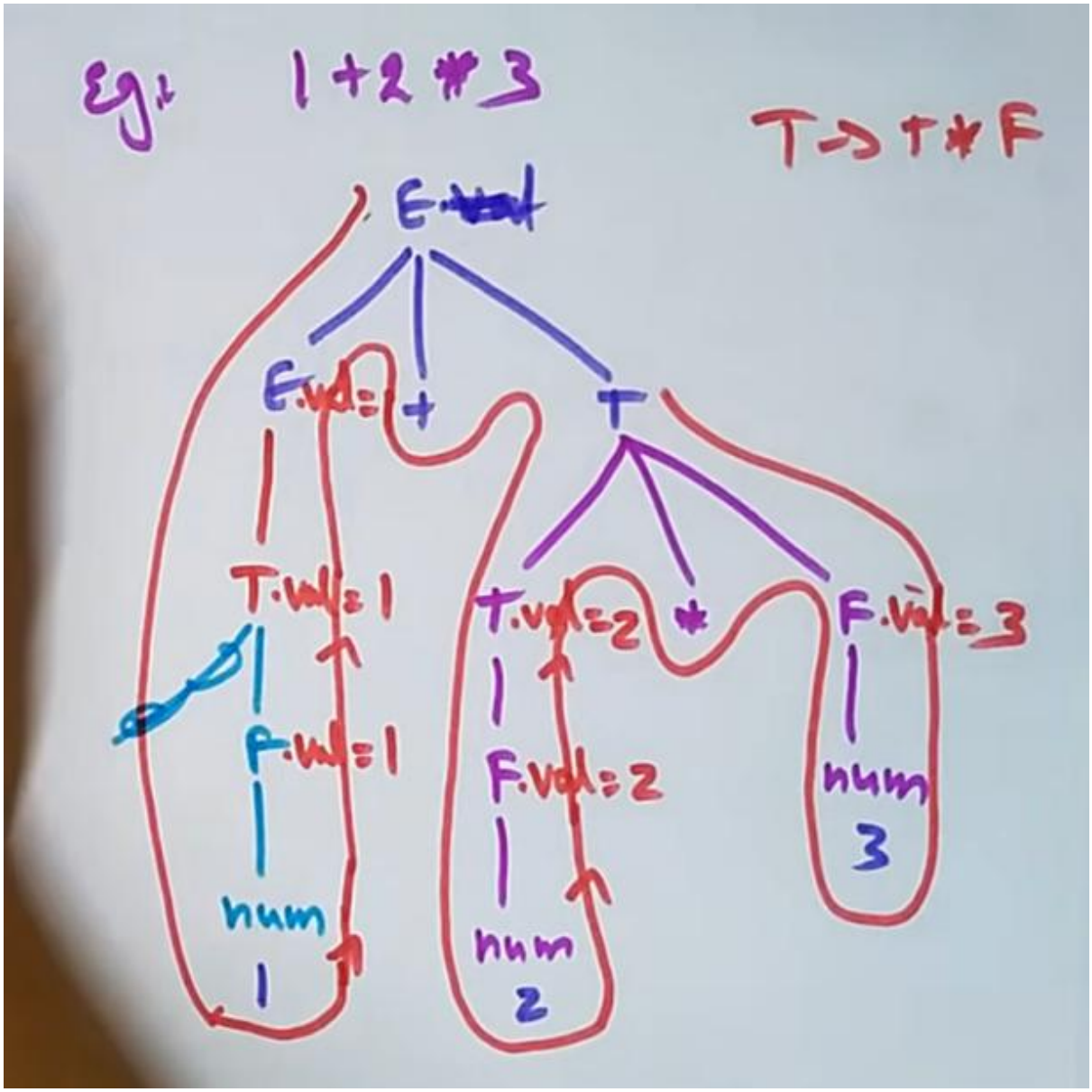
Eg: 1 + 2 * 3



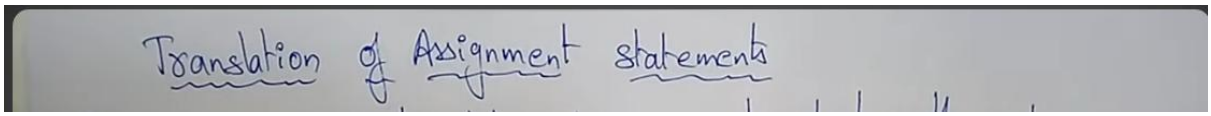
Eg: 1 + 2 * 3

T → T * F





Translation of Assignment Statements



In SDT, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array & records.

Eg: $S \to id := E$
$E \to E_1 + E_2$
$E \to E_1 * E_2$
$E \to (E_1)$
$E \to id$

The translation scheme of this grammar is

Production Rule	+	Semantic actions

① $S \to id := E$		{P = look_up (id.name); (symbol table)
		if P ≠ nil then
		Emit (P = E.place) // TAC
		Else
		error;
		}

② $E \to E_1 + E_2$	E.place = t_1 $t_1 = E_1 + E_2$	{ E.place = newtemp(); Emit (E.place = E1.place + E2.place) } // TAC
③ $E \to E_1 * E_2$	E.place = t_2 $t_2 =$	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) } // TAC
④ $E \to (E_1)$		{E.place = E1.place}

⑤ $E \to id$ // P		{P = look_up (id.name);
		if P ≠ nil then
		Emit (P = E.place) TAC
		Else
		Error; //
		}

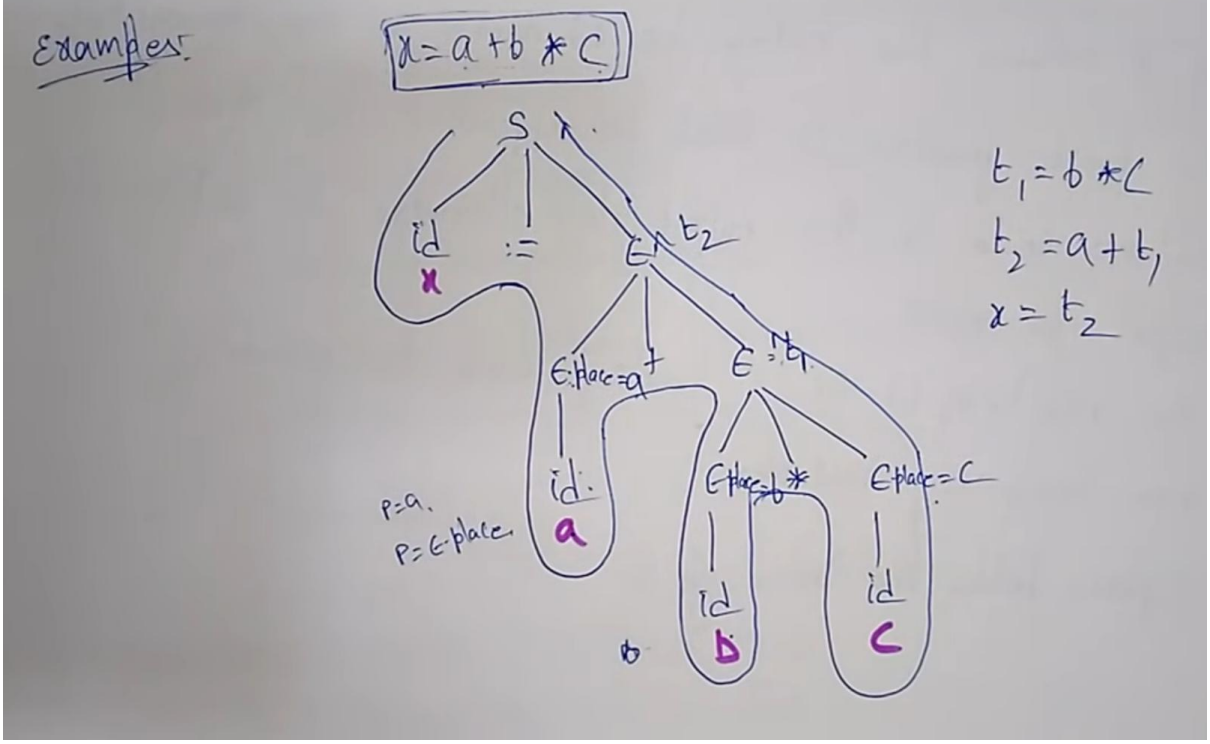
⇒ The p returns the entry for id. name in the symbol table

⇒ The Emit function is used for appending the three address code to the output file. otherwise it will report an error.

⇒ The newtemp() is a function used to generate new temporary variables

⇒ E.place holds the value of E.

Examples:



Boolean Expression

Boolean Expressions

⇒ Boolean Expressions have two primary purposes.

① used for computing logical values

② used as conditional expressions using if-then-else & while-do

⇒ Consider a grammar,

$E \rightarrow E$ OR E

$E \rightarrow E$ AND E

$E \rightarrow$ NOT E

$E \rightarrow (E)$

$E \rightarrow$ id relop id

$E \rightarrow$ True

$E \rightarrow$ False

The relop is denoted by <, >.

if-then-else & while-do

⇒ Consider a grammar,

relational operator

1 $E \rightarrow E$ OR E

2 $E \rightarrow E$ AND E

3 $E \rightarrow$ NOT E

4 $E \rightarrow (E)$

5 $E \rightarrow$ id relop id

6 $E \rightarrow$ True

7 $E \rightarrow$ False

The relop is denoted by <, >.

⇒ The AND & OR are left associated

⇒ NOT has higher precedence then AND & lastly OR.

Production Rule	Semantic Action
① $E \rightarrow E_1$ OR E_2	{ E.place = newtemp(); Emit (E.place ':=' E_1.place 'OR' E_2.pla }
② $E \rightarrow E_1$ And E_2	{ E.place = newtemp(); Emit (E.place ':=' E_1.place 'AND' E_2.place) }

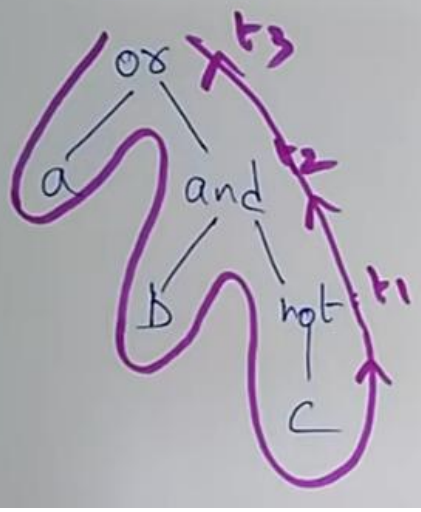
Production Rule	Semantic Actions
③ $E \rightarrow NOT\ E_1$	{ E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
④ $E \rightarrow (E_1)$	{ E.place = E1.place }
⑤ $E \rightarrow id_1\ relop\ id_2$	{ E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstate+3); Emit (E.place ':=' '0') Emit ('goto' nextstate+2) Emit (E.place ':=' '1')

Production Rule	Semantic Action
⑥ $E \rightarrow True := 1$	{ E.place := newtemp(); Emit (E.place ':=' '1') }
⑦ $E \rightarrow False := 0$	{ E.place := newtemp(); Emit (E.place ':=' '0') }

⇒ Emit generates TAC & newtemp() generates temporary variable

⇒ $E \rightarrow id_1\ relop\ id_2$ contains next_state & it gives the index of next three address stm in the output sequence.

Eg: a or b and not c



t_1 = not c

t_2 = b and t_1

t_3 = a or t_2

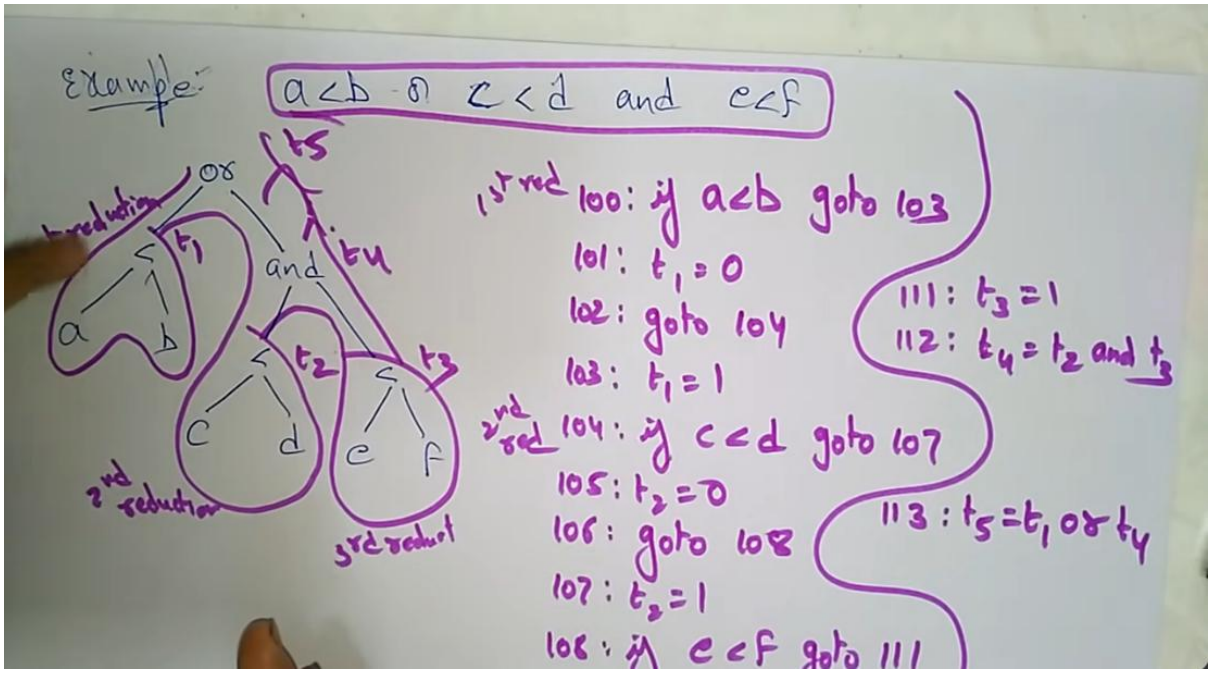
Eg:- if a<b then 1 else 0

100: if a<b goto 103

101: t_1 = 0

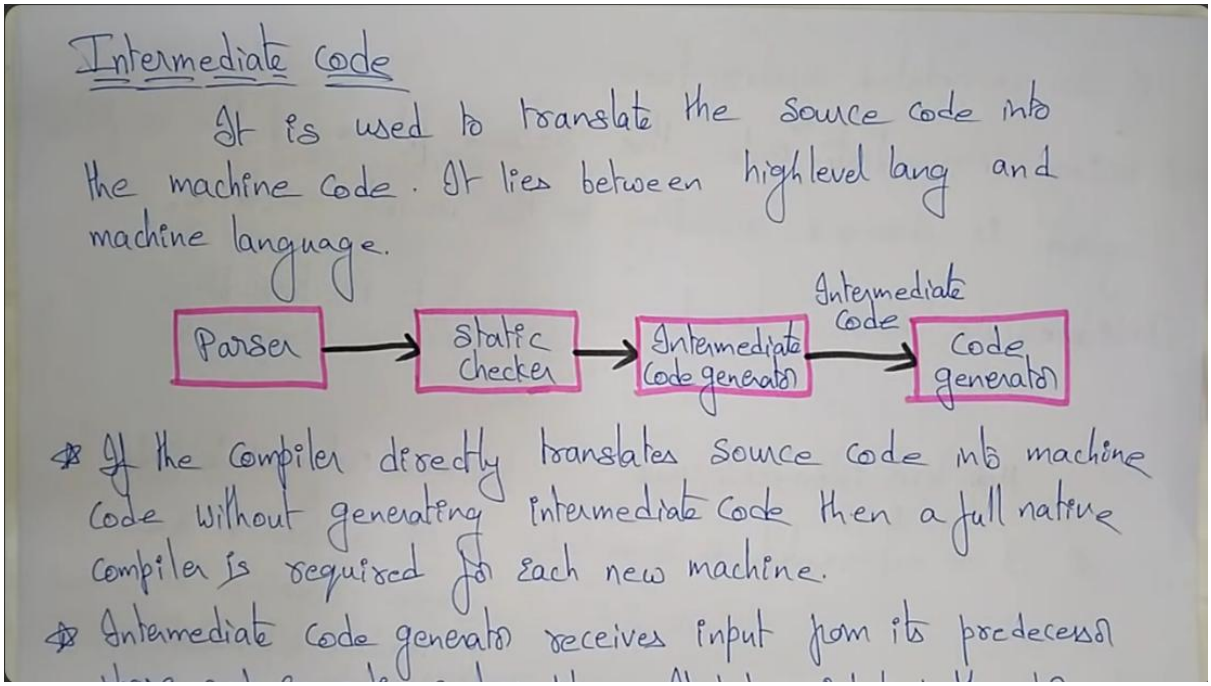
102: goto 104

103: t_1 = 1



109 : t3 = 0
110 : goto 112

Intermediate Code Generation



of an annotated syntax tree.

* using intermediate code, the second phase of the compiler ~~System~~ is changed according to the target machine.

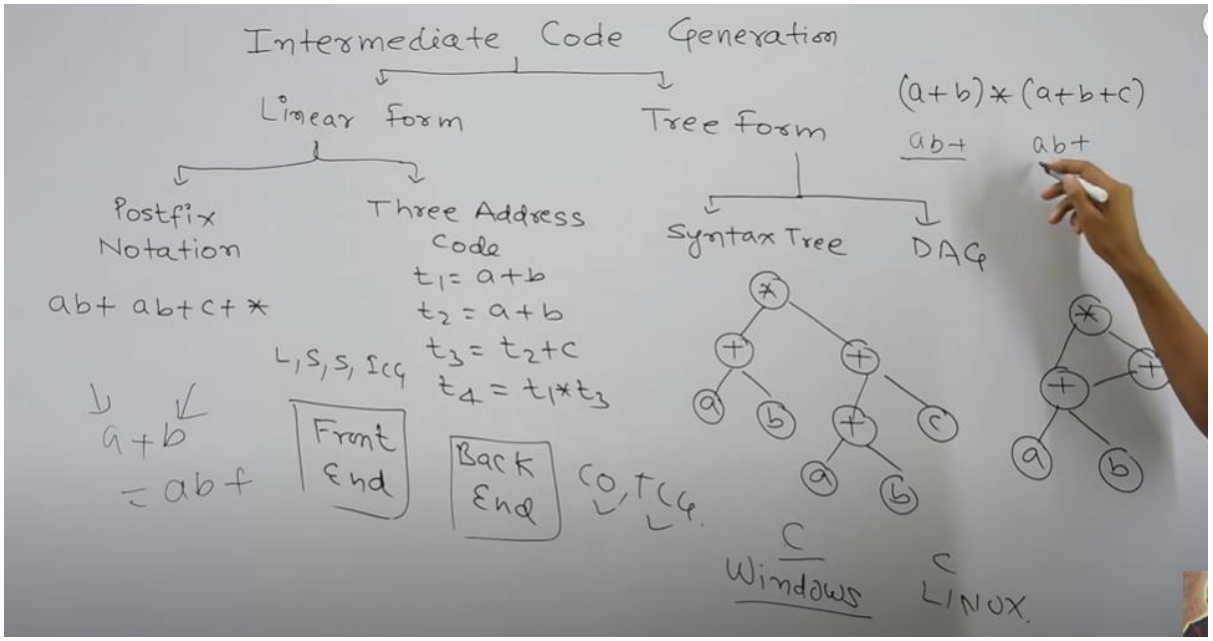
Intermediate code can be represented in two ways

High level Intermediate code
* It can represented as Source code

low level intermediate code
* It is close to the target machine. It is used for machine dependent optimizations

The different forms of Intermediate Code

1. Abstract syntax tree
2. Polish Notation
3. Three address code



Types of Three Address Code.

1) $x = y \text{ op } z$

2) $x = \text{op } z$

3) $x = y$

4) if $(x \text{ rel op } y)$ goto L

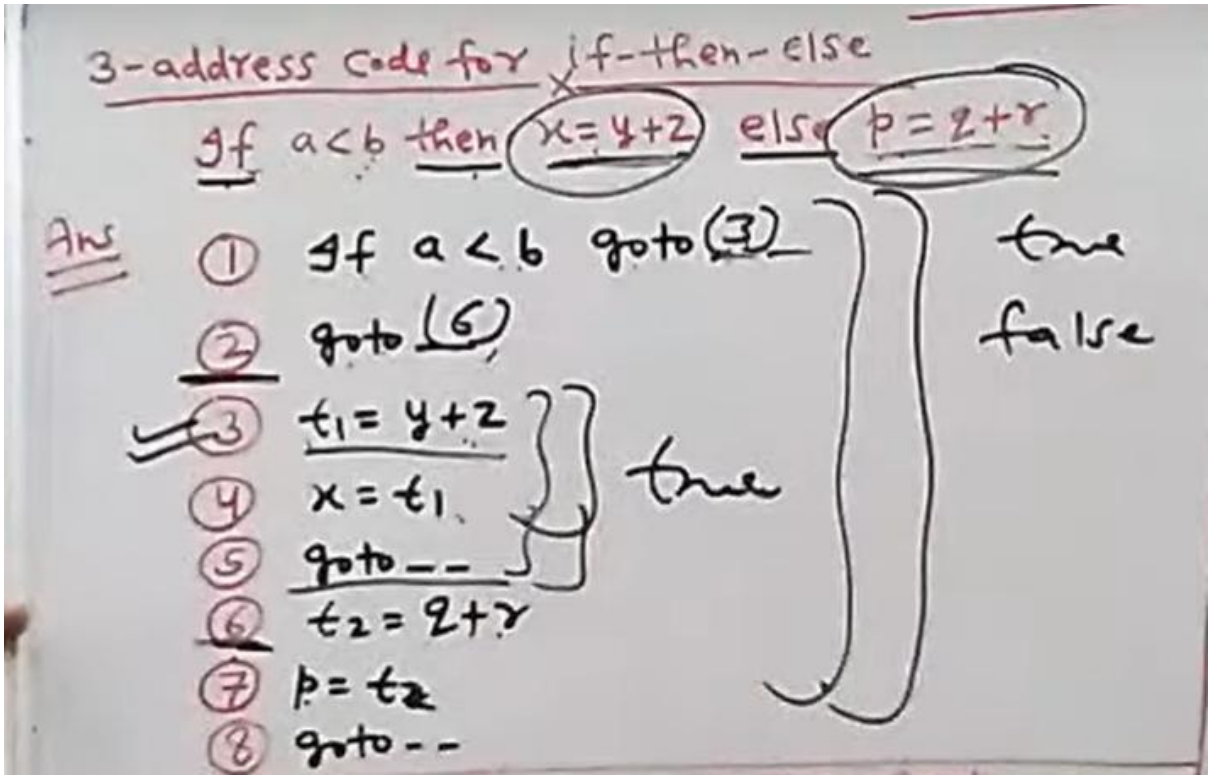
5) goto L

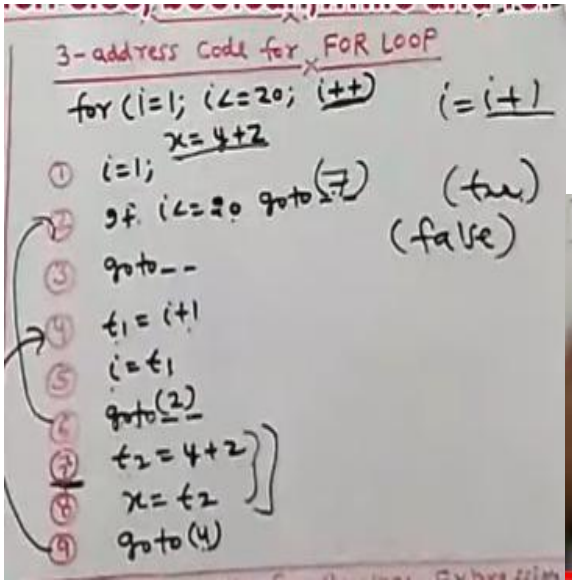
6) $A[i] = x$

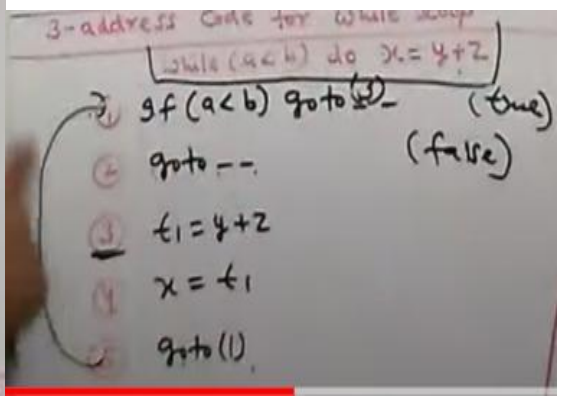
$y = A[i]$

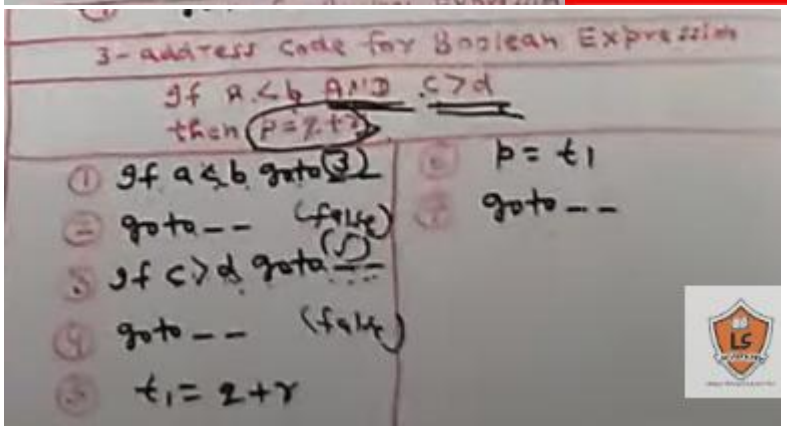
7) $x = *p$

$y = \&x$









Do- while loop

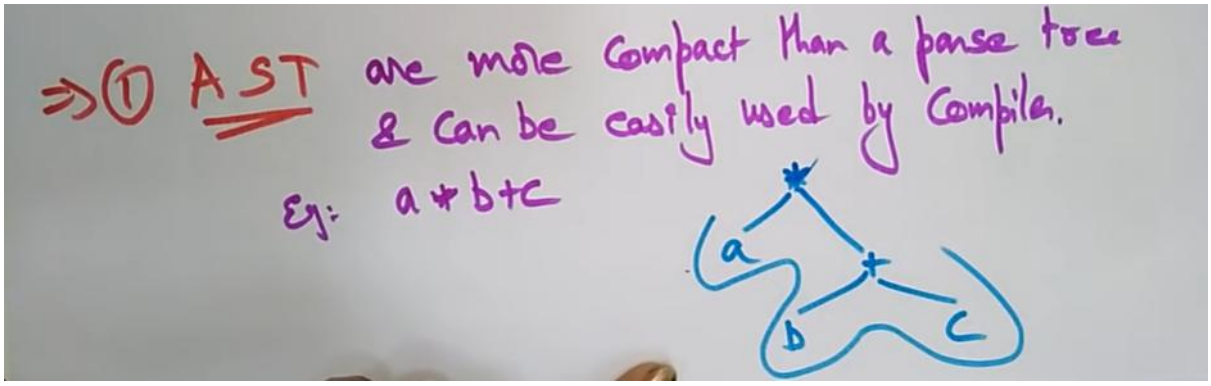
Actual code

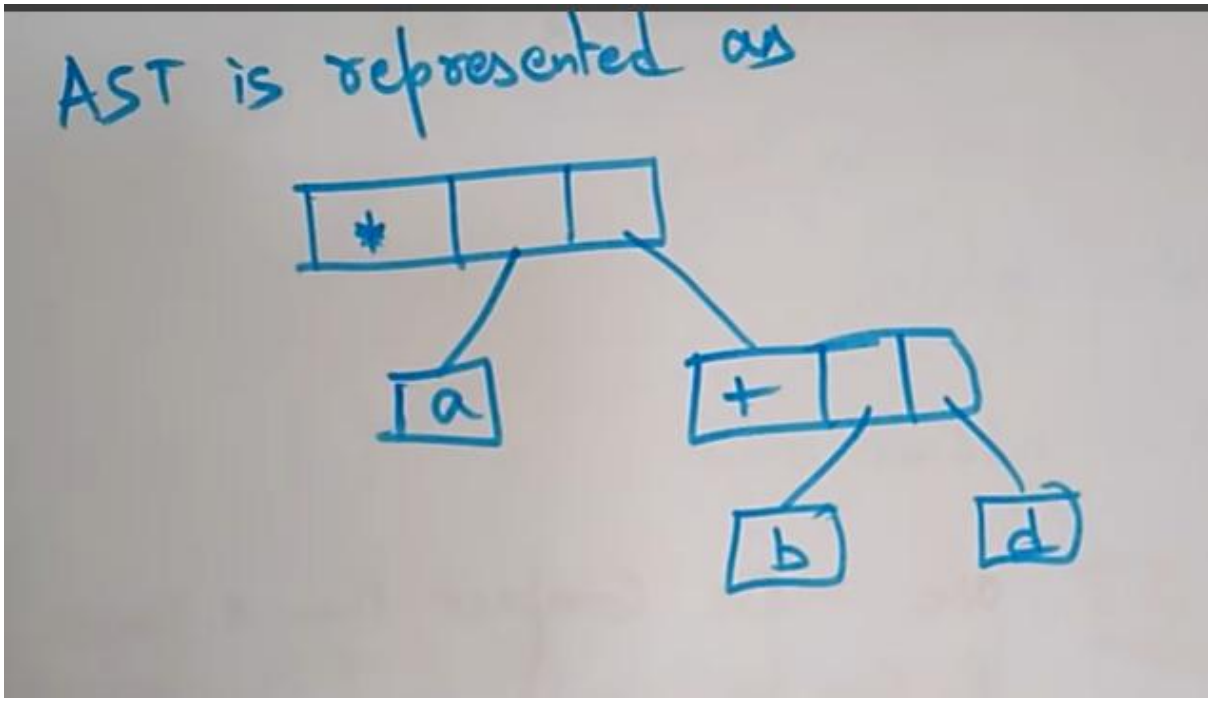
```
#include <stdio.h>

int main() {
    int i = 1;
    do {
        printf("%d\n", i);
        i++;
    } while (i <= 5);
    return 0;
}
```

3AC

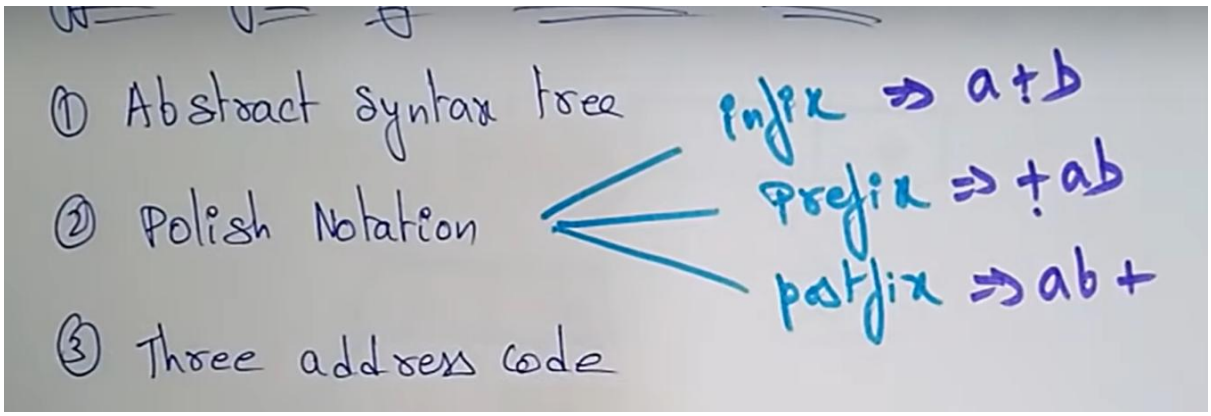
```
begin:
t1 = i <= 5
if t1 goto loop_body
goto end
loop_body:
t2 = i
param t2
call printf, 1
t3 = i + 1
i = t3
goto begin
end:
```

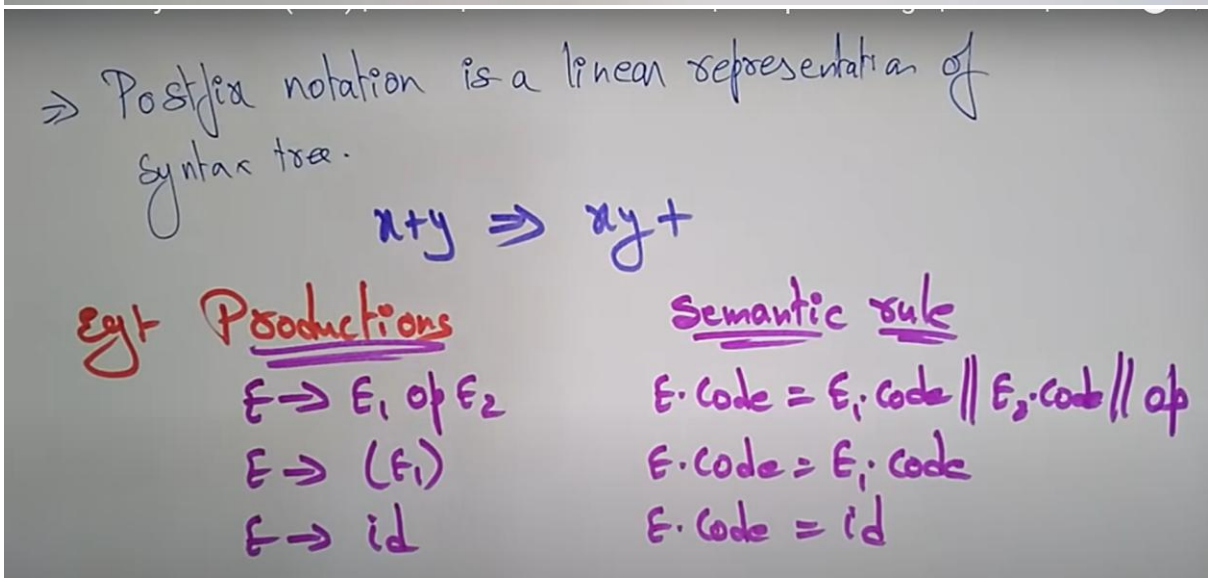
Abstract Syntax tree (AST) | Intermediate Code

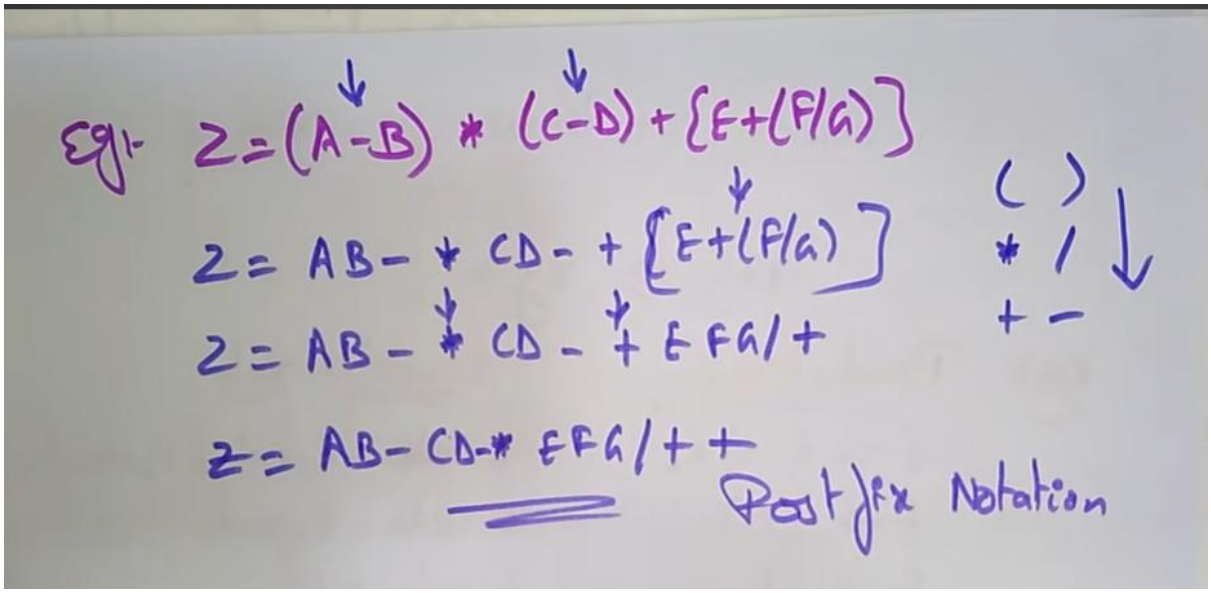



AST is represented as
*
a
+
b
d

Polish

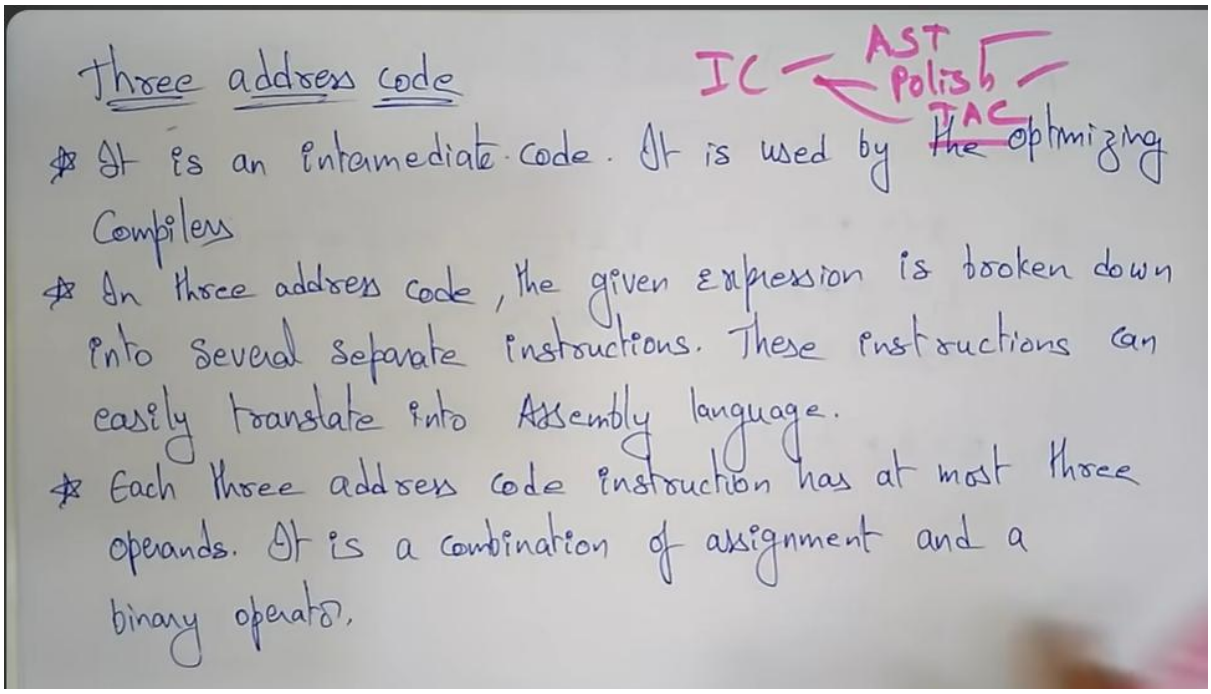

① Abstract syntax tree
② Polish Notation
③ Three address code
infix ⇒ a+b
prefix ⇒ +ab
postfix ⇒ ab+


⇒ Postfix notation is a linear representation of syntax tree.
x+y ⇒ xy+
Eg:- Productions
E → E1 op E2
E → (E1)
E → id
Semantic rule
E.code = E1.code || E2.code || op
E.code = E1.code
E.code = id



Three address Code | Intermediate Code | Quadruples,Triples,Indirect Triples | Representation of three address code

https://www.naukri.com/code360/library/three-address-code



In TAC, there is at most one operator on the right side of an instruction.

eg: $x + y * 2$

$t_1 = y * 2$

$t_2 = x + t_1$

t_1 & t_2 are compiler generated temporary names

Eg: $a + a * (b - c) + d * (b - c)$

TAC

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = d * t_1$

$t_5 = t_3 + t_4$

① Quadruples

4 fields ← op, src1, src2, destination.

Eg: $a := -b * c + d$

TAC:

$t_1 = -b$

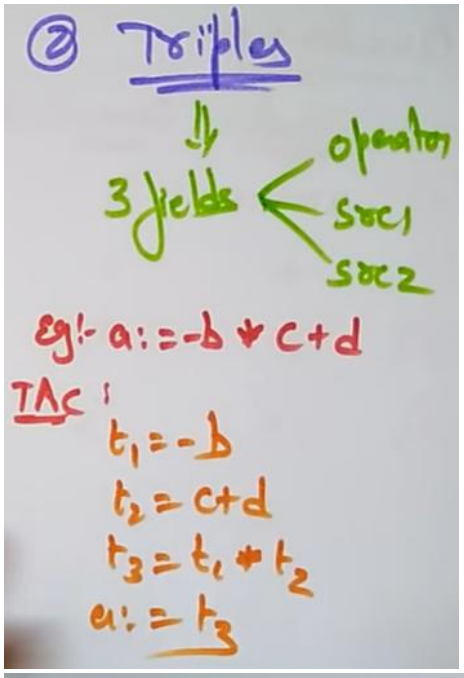
$t_2 = c + d$

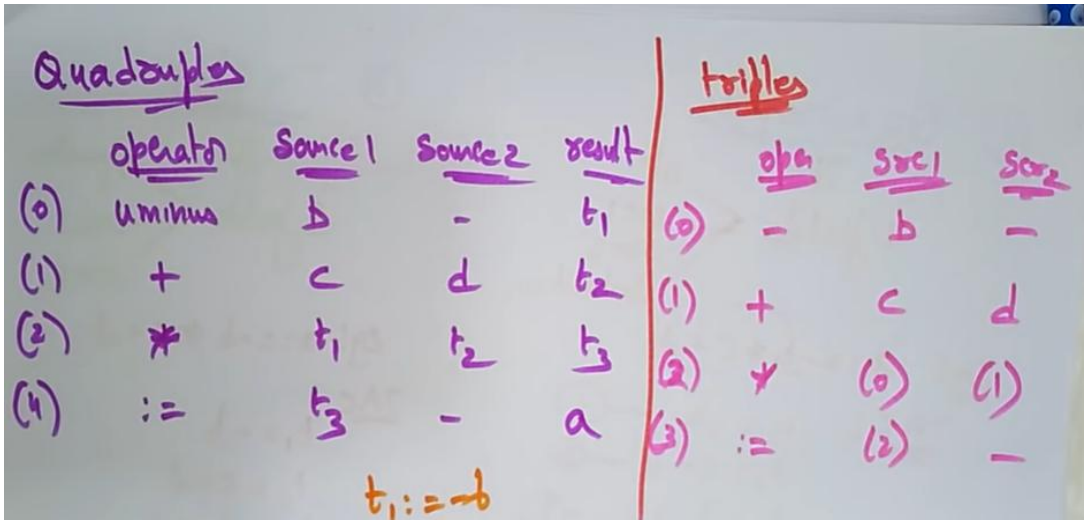
$t_3 = t_1 * t_2$

$a := t_3$

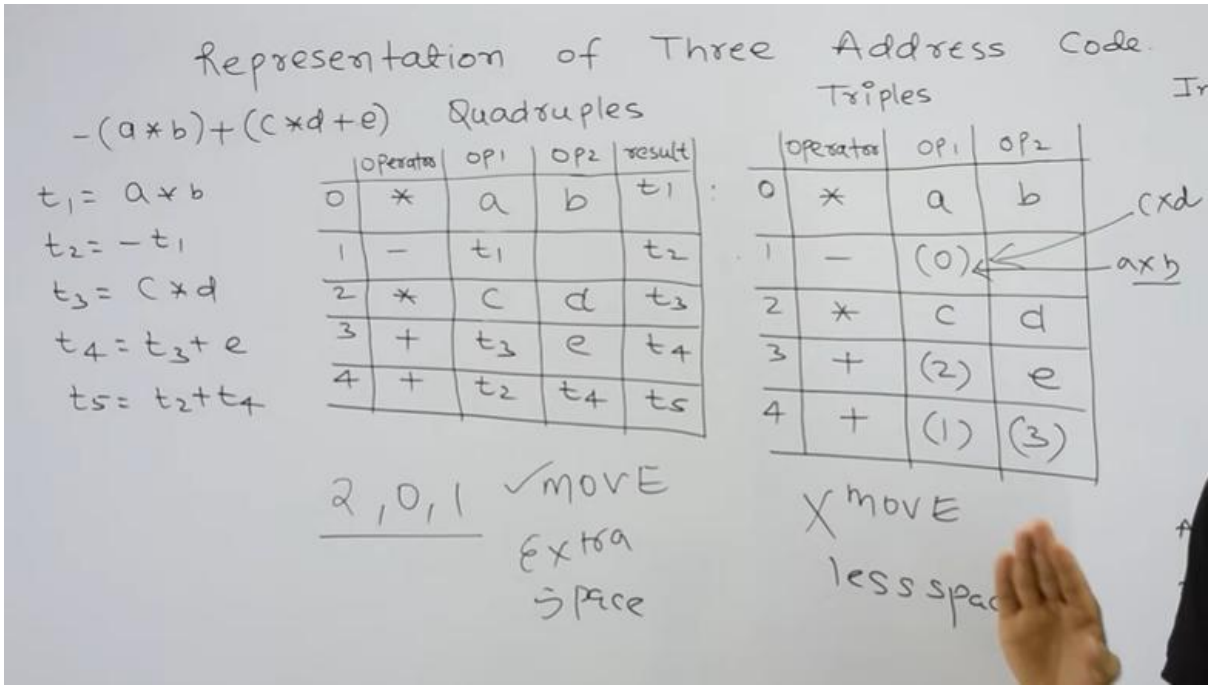
Quadruples

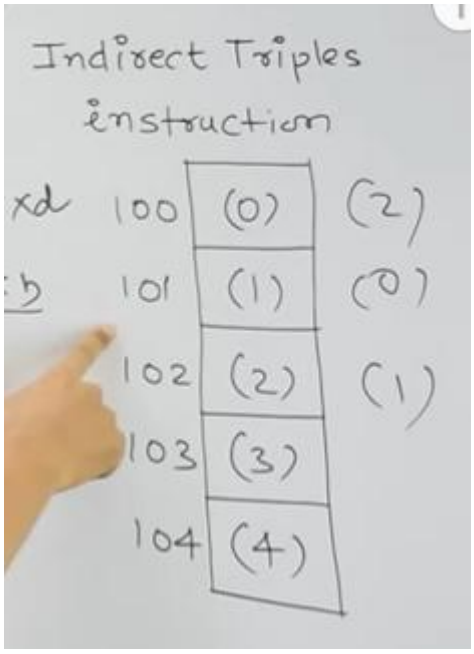
	operator	Source 1	Source 2	result
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(4)	:=	t_3	-	a

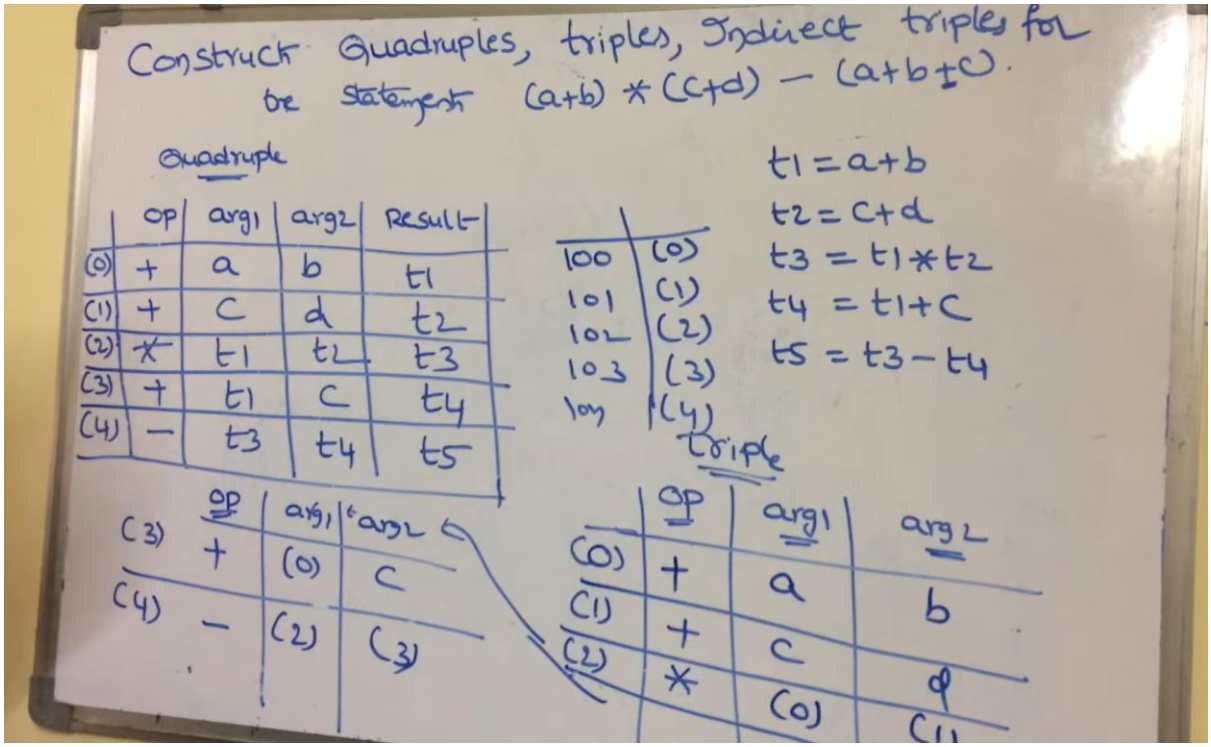




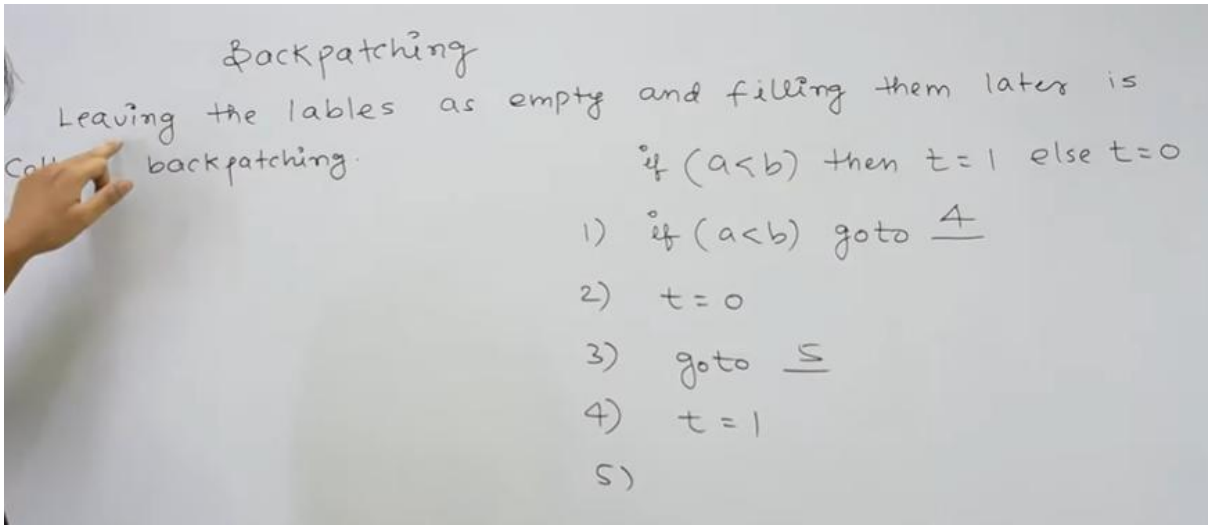
In triples take the references from the quadruples.


Representation of Three Address Code
-(a*b)+(c*d+e)
Quadruples
t1 = a * b
t2 = - t1
t3 = c * d
t4 = t3 + e
t5 = t2 + t4
Triples
2,0,1
✓move
extra
space
X move
less spac


Indirect Triples
instruction
100
101
102
103
104



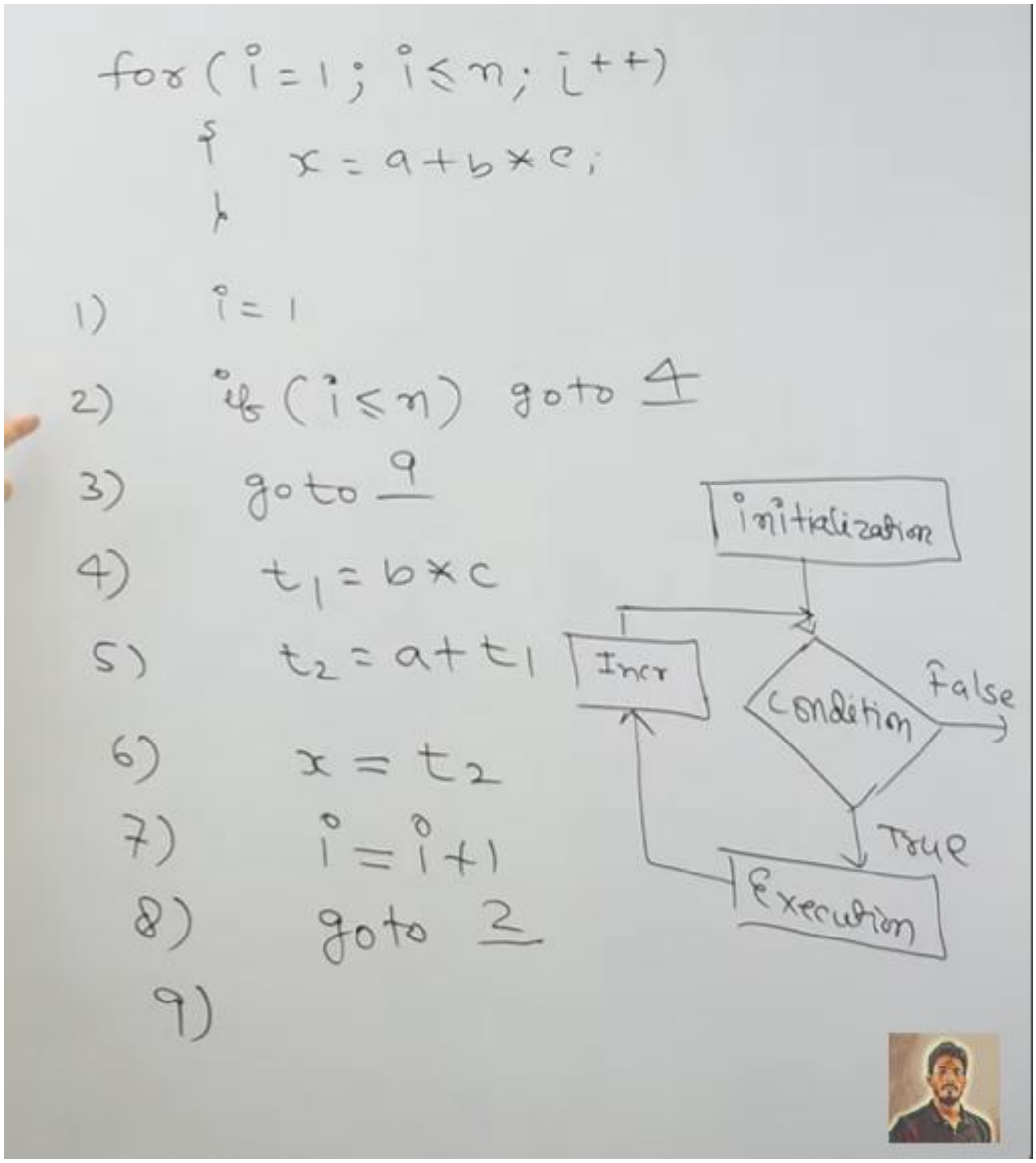
Backpatching Example 1 (If & Else Case)

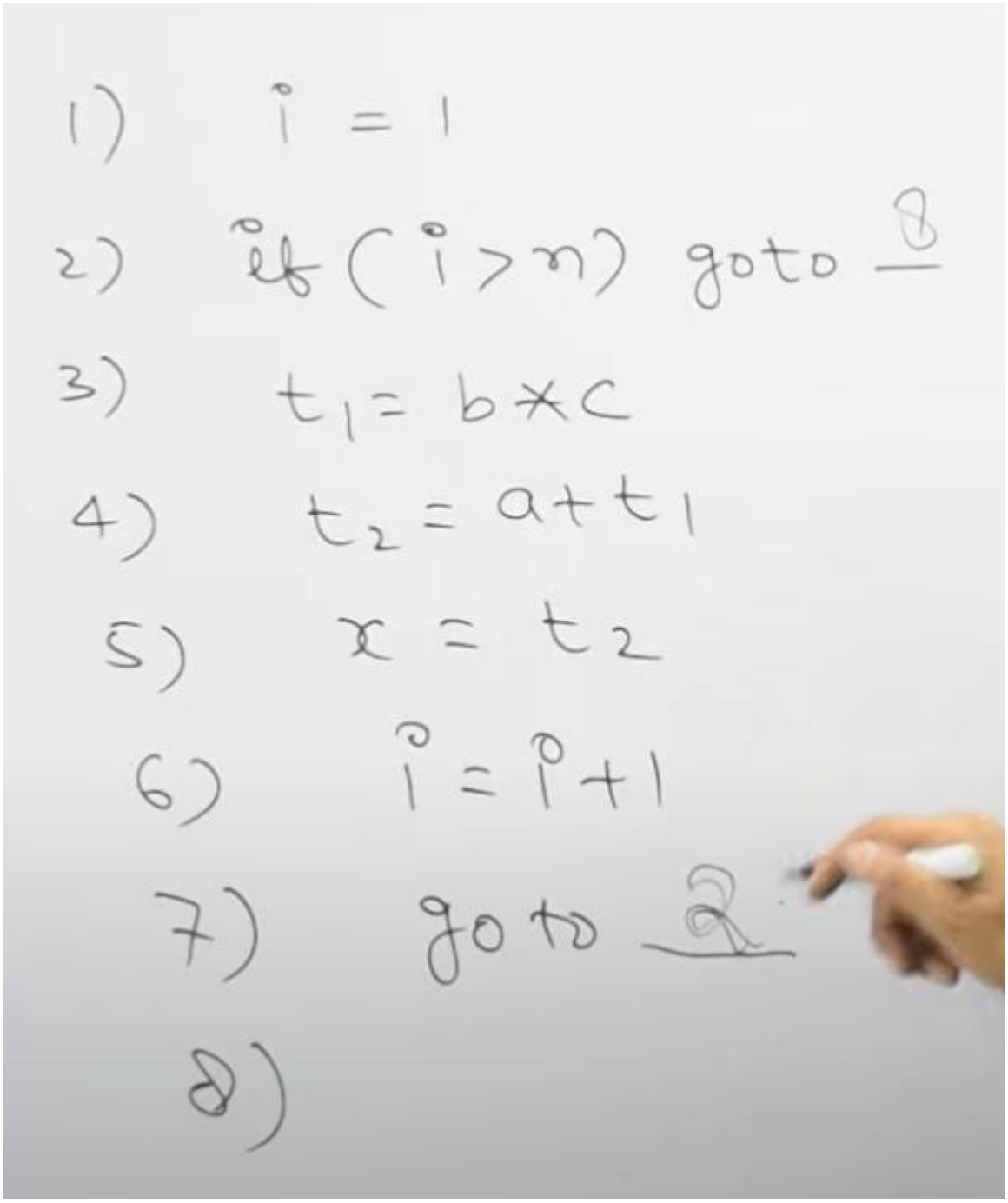


if (a<b) && (c<d) then t=1 else t=0

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)

Backpatching Example 2 (Loop & Switch Case)




1) i = 1
2) if (i > n) goto 8
3) t1 = b * c
4) t2 = a + t1
5) x = t2
6) i = i + 1
7) goto 2
8)

1) if (i==1) goto 7

2) if (i==2) goto 11

3) $t_1 = b_3 * c_3$

4) $t_2 = a_3 + t_1$

5) $x_3 = t_2$

6)

Case 1 | 7) $t_1 = b_1 * c_1$

8) $t_2 = a_1 + t_1$

9) $x = t_2$

10) goto 6

```
switch(i)
{
Case 1:
        x1 = a1 + b1 * c1;
        break;
Case 2:
        x2 = a2 + b2 * c2;
        break;
default:
        x3 = a3 + b3 * c3;
        break;
}
```

Case 2 | 11) $t_1 = b_2 * c_2$

12) $t_2 = a_2 + t_2$

13) $x = t_2$

14) goto 6

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

(i<10) or (i≥10)

1) $x = 0$

2) $i = 0$

3) if (i≥10) goto 15

4) t_1 = base add of A

5) $t_2 = i * 2$

6) $t_3 = t_1[t_2]$

A

100	102	104	106	108
4	5	9	3	2
0	1	2	3	4

$A[3] = BA + (i - lb) \times C$

$= 100 + (3-0) \times 2$

$= 106$

7) t_4 = base add of B

8) $t_5 = i * 2$

9) $t_6 = t_4[t_5]$

10) $t_7 = t_3 * t_6$

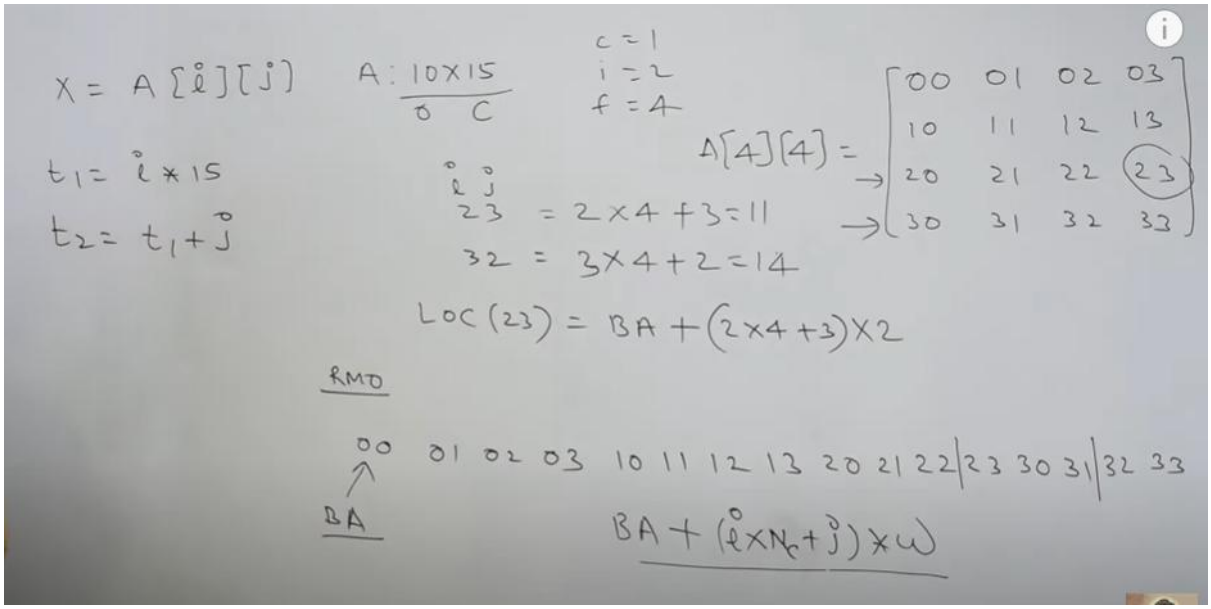
11) $t_8 = x + t_7$

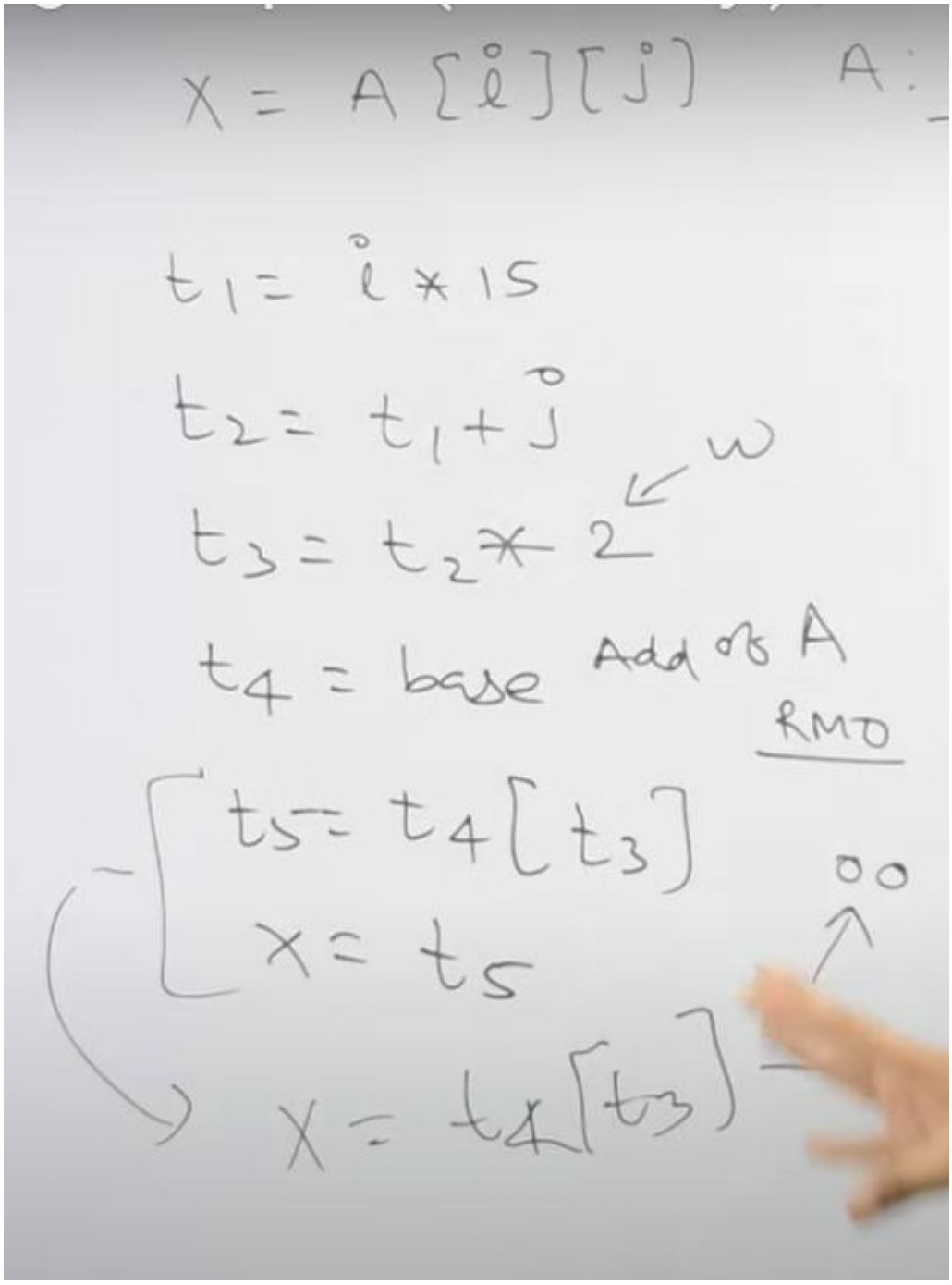
12) $x = t_8$

13) $i = i + 1$

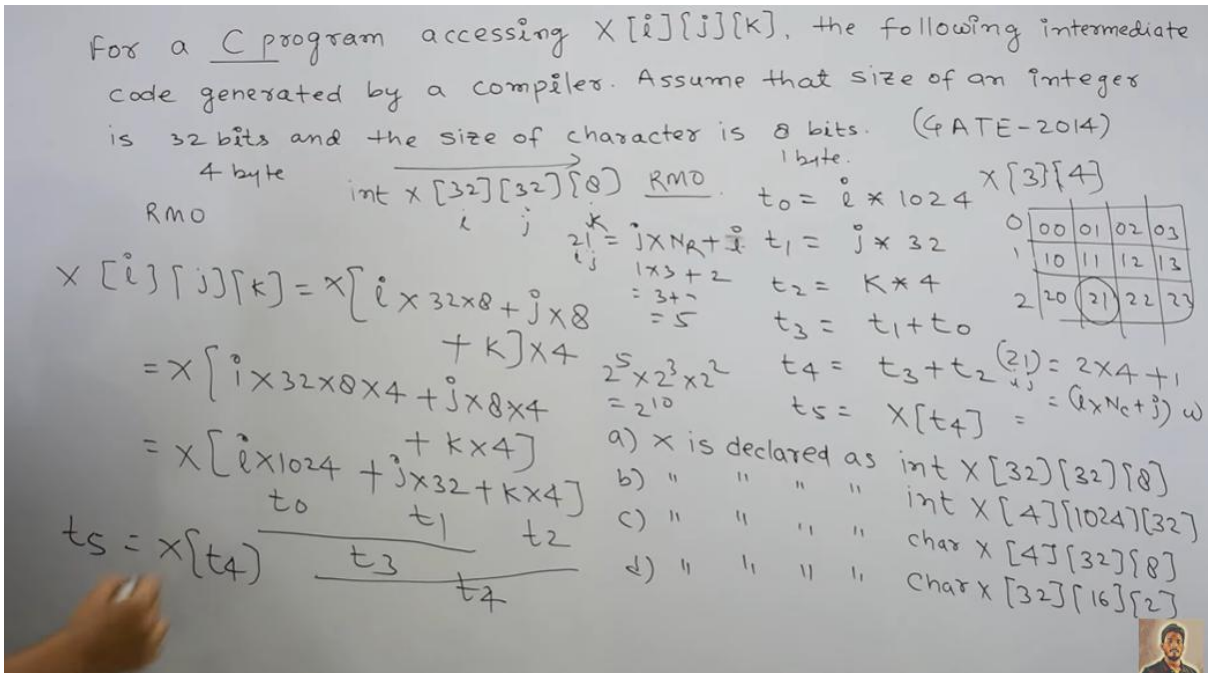
14) goto 3

15) —

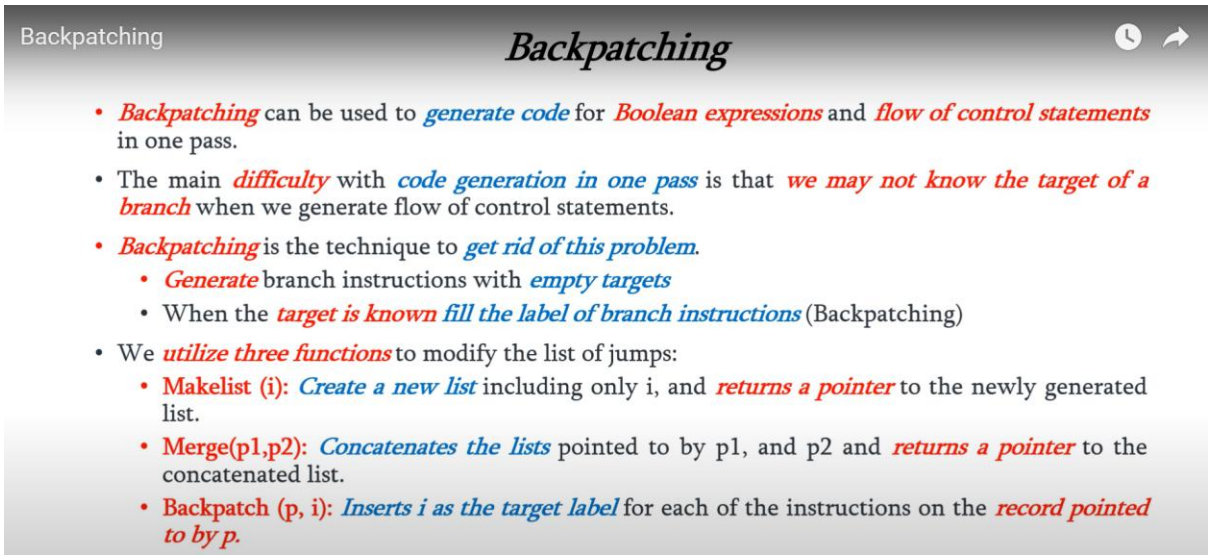

X = A[i][j]
A: 10X15
0 C
c = 1
i = 2
f = 4
t1 = i * 15
t2 = t1 + j
A[4][4] =
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32 33
i j
23 = 2X4 + 3 = 11
32 = 3X4 + 2 = 14
Loc(23) = BA + (2X4 + 3)X2
RMO
00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
BA
BA + (i X N + j) X w


X = A[i][j]
A:
t1 = i * 15
t2 = t1 + j
w
t3 = t2 * 2
t4 = base Add of A
RMO
t5 = t4[t3]
X = t5
00
X = t4[t3]

Backpatching GATE Solved Example (3 D Array)

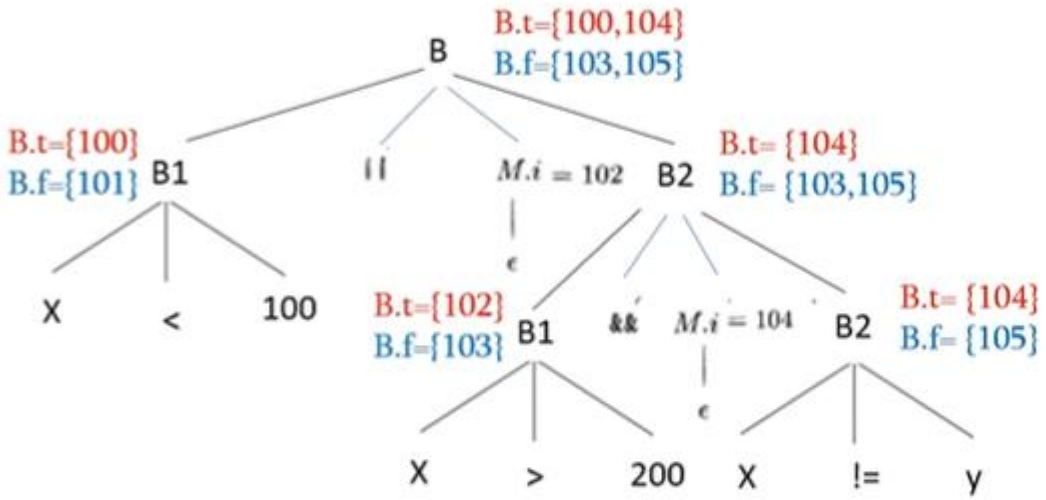


Backpatching for Booleans



Productions	Semantic Actions
B -> B1 || M B2	{backpatch(B1.falselist, M.instr); B. truelist = merge(B1. truelist, B2. truelist); B. falselist = B2. falselist; }
B -> B1 && M B2	{backpatch(B1 . truelist, M. instr); B. truelist = B2 . truelist;, B. falselist = merge(Bl. falselist, B2 . falselist); }
B -> ! B1	{ B. truelist = B1 . falselist; B. falselist = B1 . truelist; }
B->(B1)	{ B. truelist = B1. truelist; B. falselist = B1 .falselist;)
B -> E1 rel E2	{ B. truelist = makelist(nextinstr) ; B. falselist = makelist(nextinstr + 1); emit('if' E1 .addr rel.op E2.addr 'goto----'); emit('goto ------'); }
B -> true	{ B . truelist = makelist(nextinstr) ; emit('goto –'); }
B -> false	{ B .falselist = makelist(nextinstr) ; emit('goto –'); }
M ->€	M. instr = nextinstr;

Example: x<100 | | x> 200 && x!=y



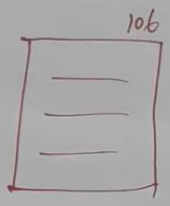
Three Address Code:

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

Backpatching

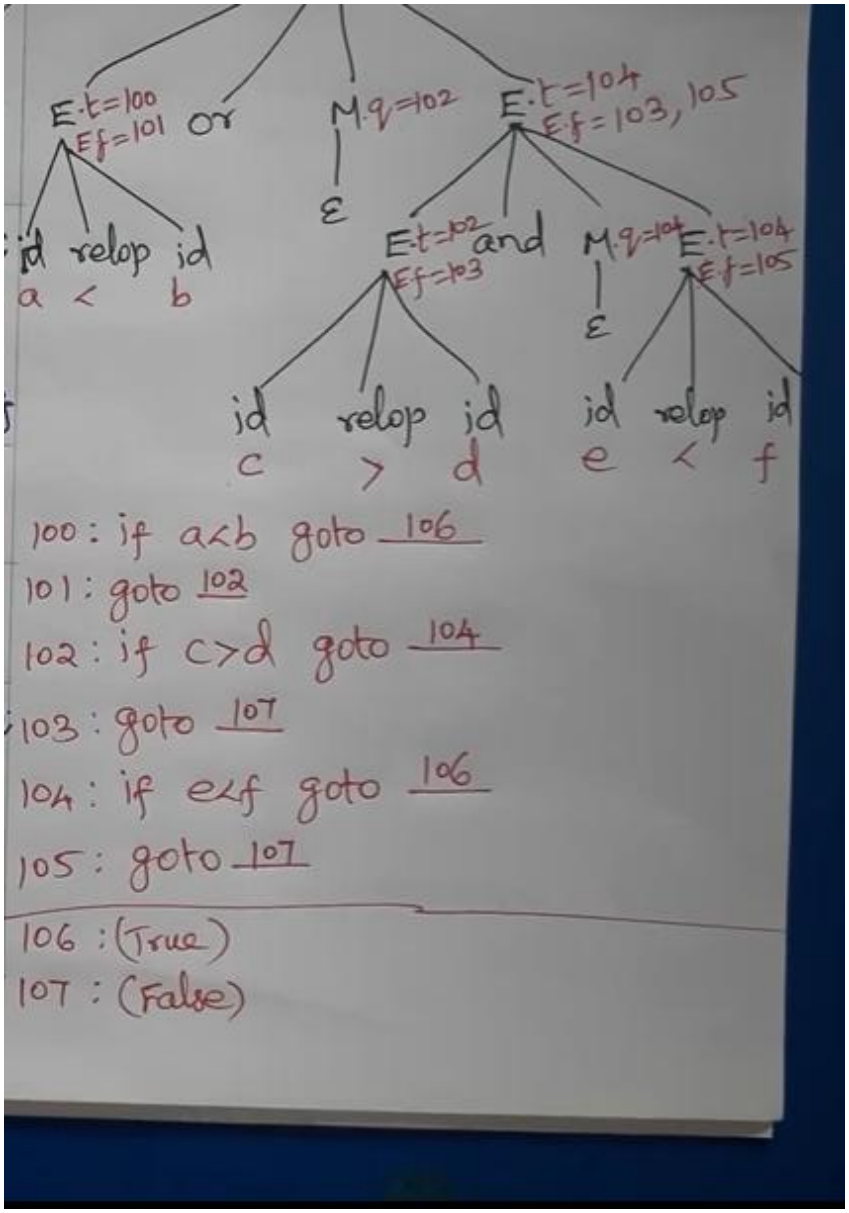
a<b or c>d and e<f

100: if a<b goto 106
101: goto 102
102: if c>d goto 104
103: goto 107
104: if e<f goto 106
105: goto 107
106: (true)
107: (false)

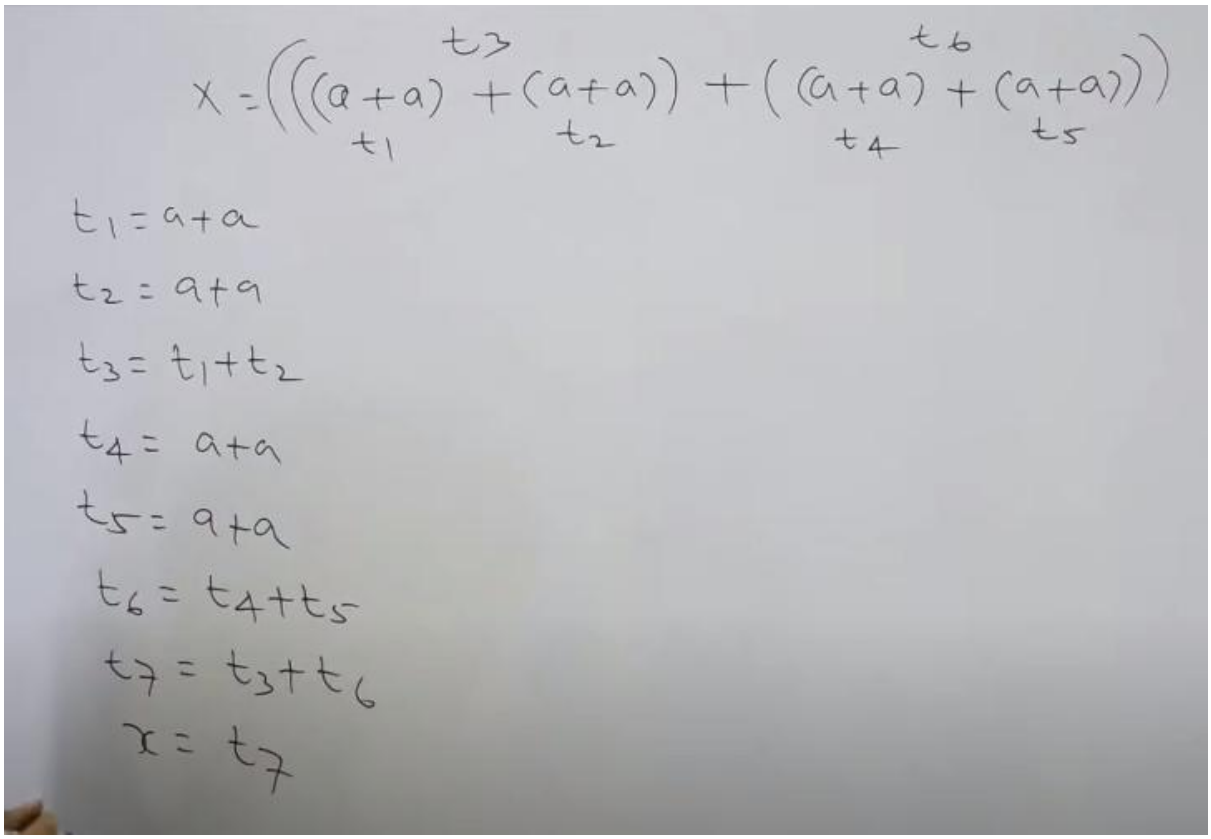


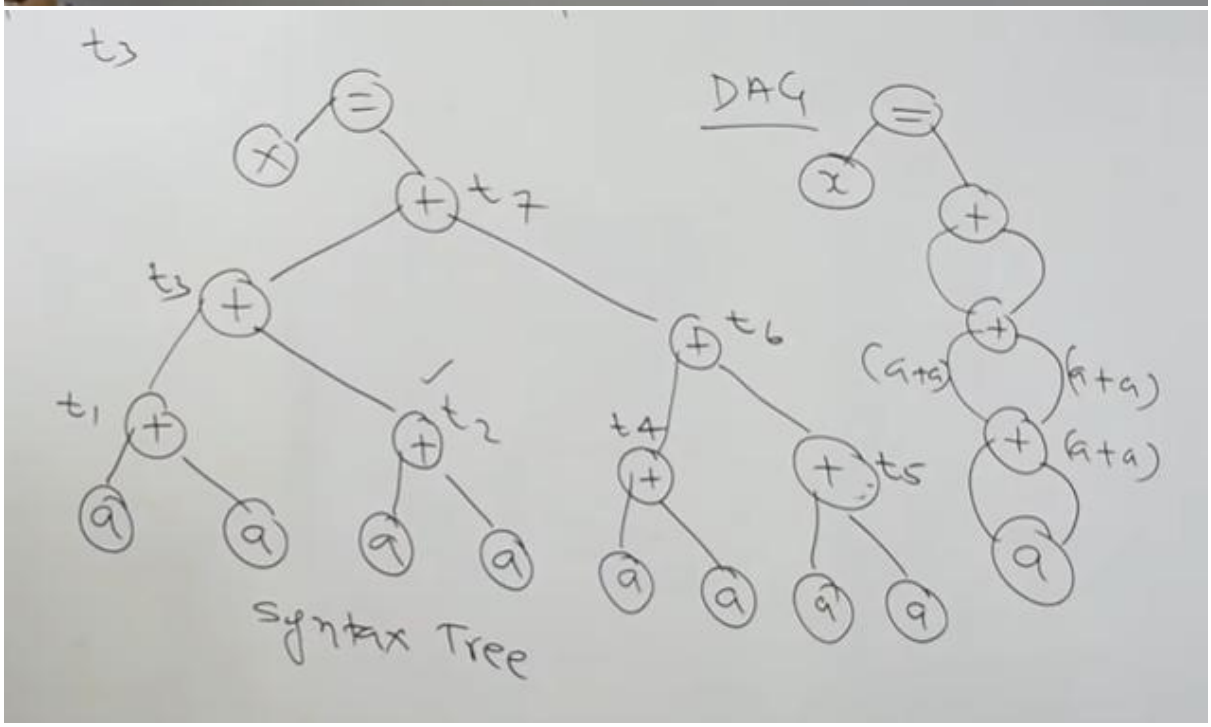
1) makelist(i)
2) merge(L1, L2)
3) backpatch(L, label)
4) nextquad

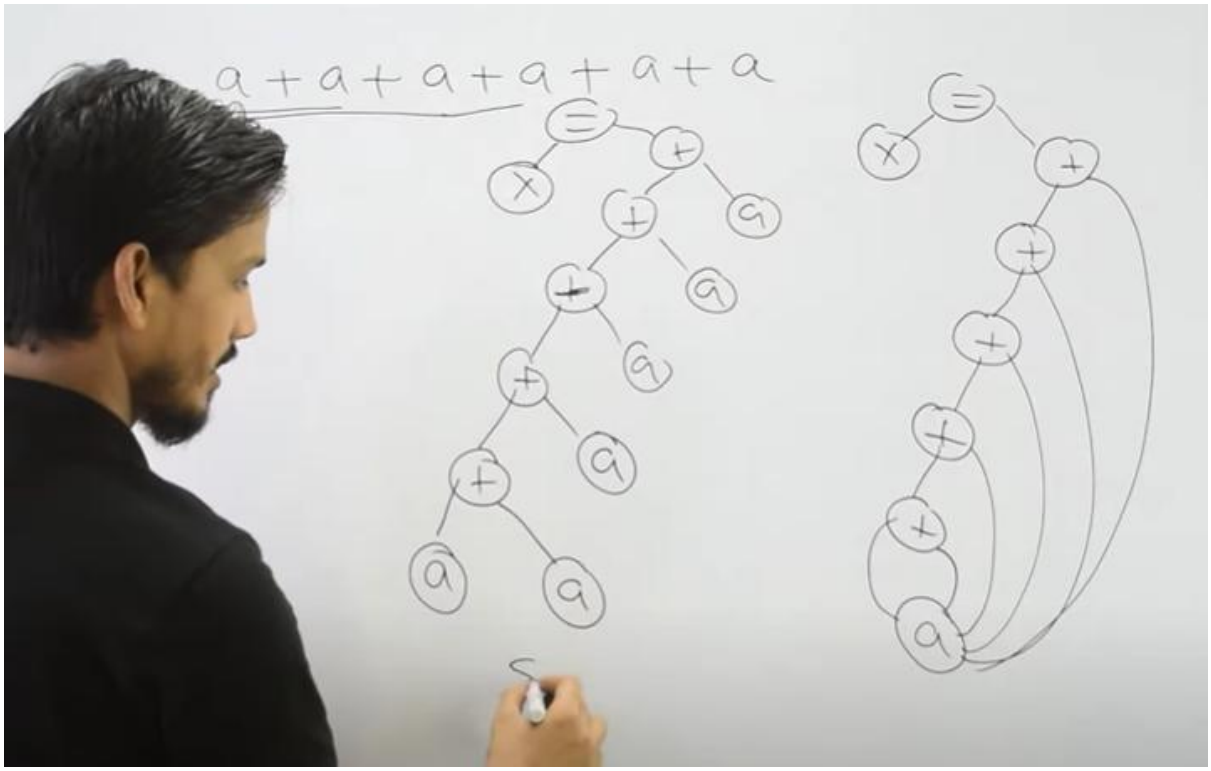
	E.truelist = merge(E1.truelist, E2.truelist); E.falselist = E2.falselist }
E → E1 and ME2	{backpatch(E1.truelist, M.quad); E.truelist = E2.truelist; E.falselist = merge(E1.falselist, E2.falselist)}
E → not E1	{E.truelist = E1.falselist; E.falselist = E1.truelist }
E → (E1)	{ E.truelist = E1.truelist; E.falselist = E1.falselist }
E → id1 relop id2	{E.truelist = makelist(nextquad); E.falselist = makelist(nextquad+1); gen('if' id1 relop id2 'goto __') gen('goto __') }
M → ε	M.quad = nextquad


E.t=100
Ef=101
or
M.q=102
E.t=104
Ef=103, 105
ε
id relop id
a < b
E.t=102
Ef=103
and
M.q=104
E.t=104
Ef=105
ε
id relop id
c > d
id relop id
e < f
100 : if a<b goto 106
101 : goto 102
102 : if c>d goto 104
103 : goto 107
104 : if e<f goto 106
105 : goto 107
106 : (True)
107 : (False)

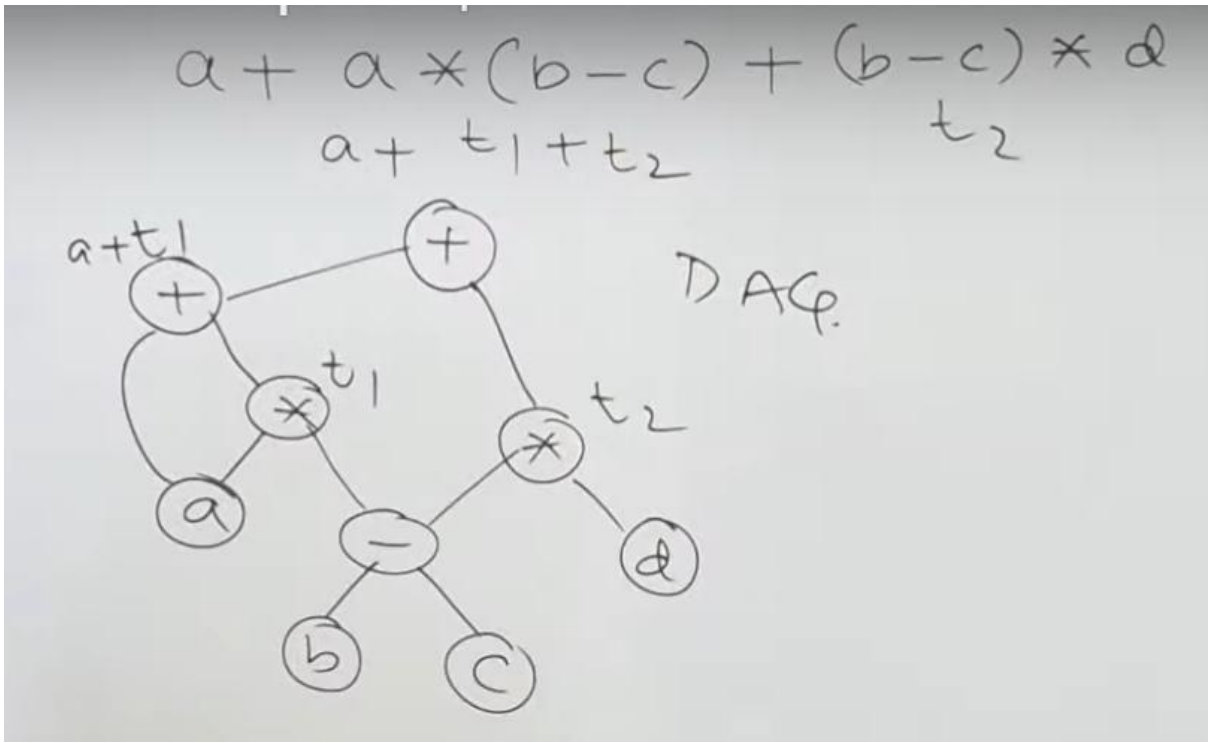
Directed Acyclic Graph | DAG Examples 1 | Intermediate Code Generation

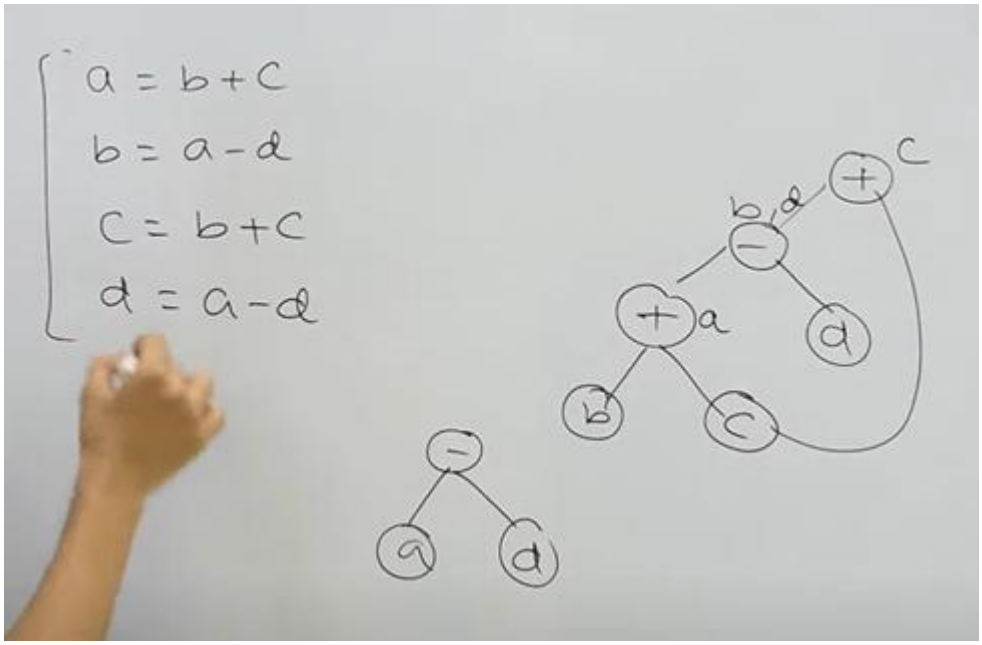


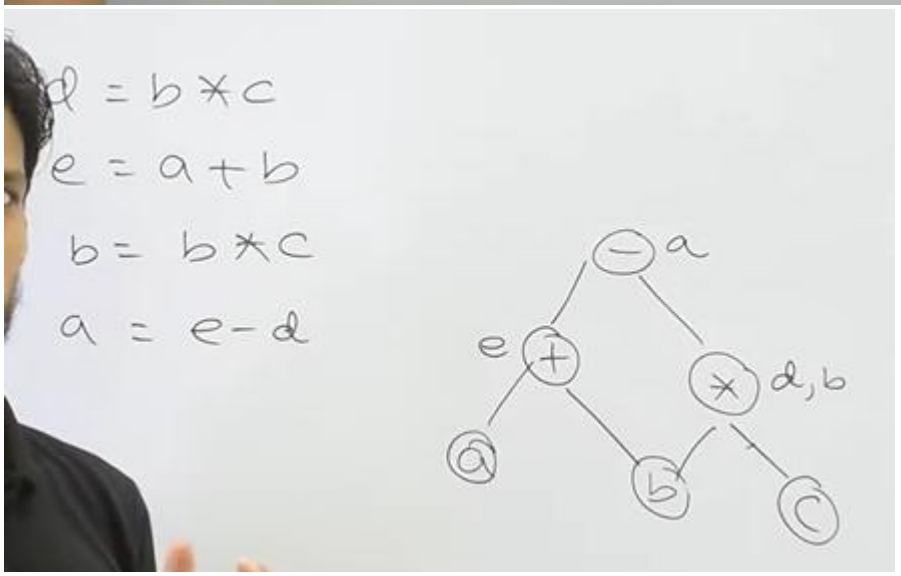




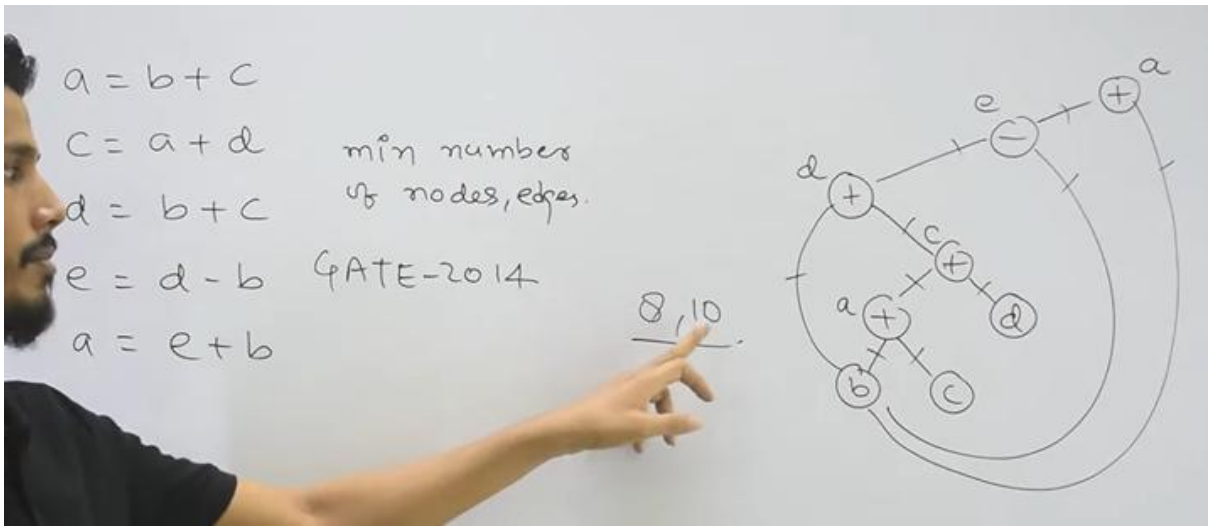
Directed Acyclic Graph | DAG Examples 2

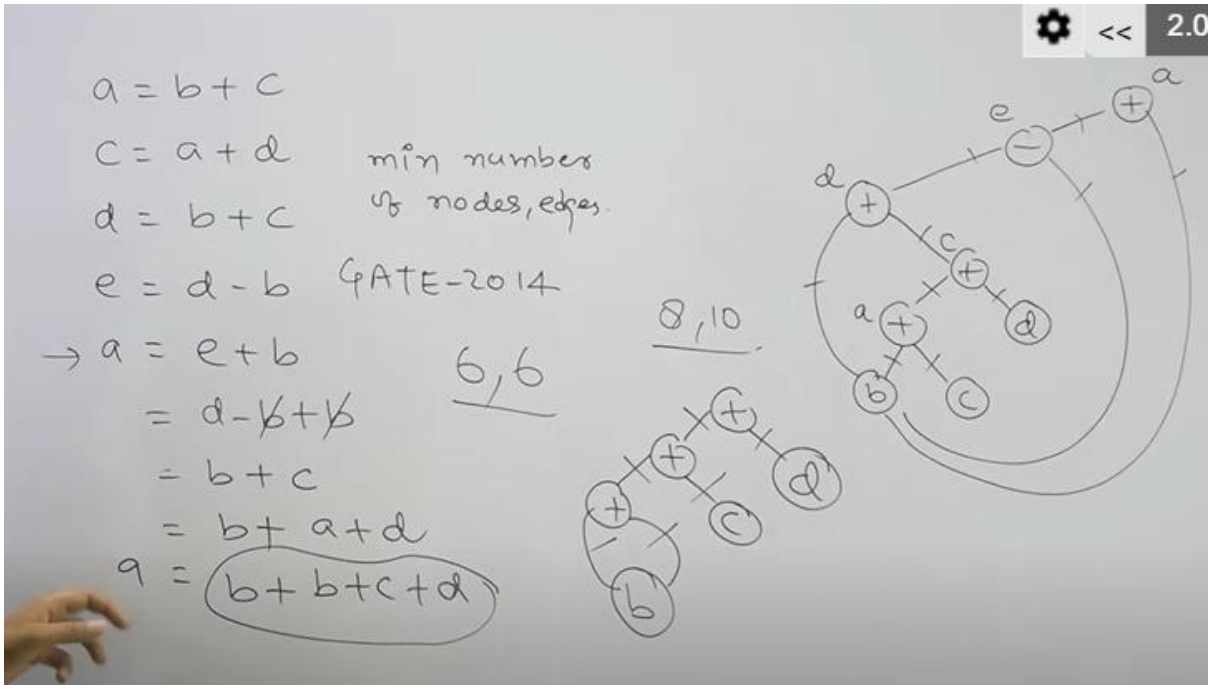






Directed Acyclic Graph | DAG Examples 3





Code generation using DAG

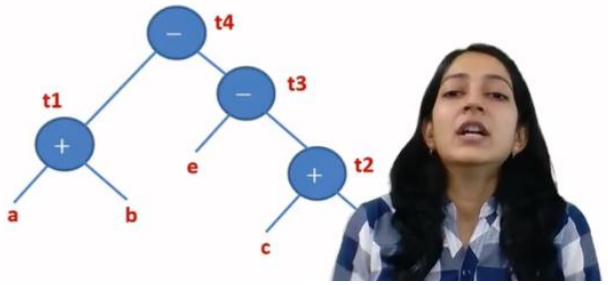
Generation of Code from DAGs

- Methods generating code from DAGs are:
  1. Rearranging Order
  2. Heuristic ordering
  3. Labeling algorithm

Rearranging Order

- The order of three address code affects the cost of the object code being generated.
- By changing the order in which computations are done we can obtain the object code with minimum cost.
- Example:

t1:=a+b
t2:=c+d
t3:=e-t2
t4:=t1-t3
Three Address Code



Example: Rearranging Order

t1:=a+b
t2:=c+d
t3:=e-t2
t4:=t1-t3
Three Address Code

Re-arrange →

t2:=c+d
t3:=e-t2
t1:=a+b
t4:=t1-t3
Three Address Code

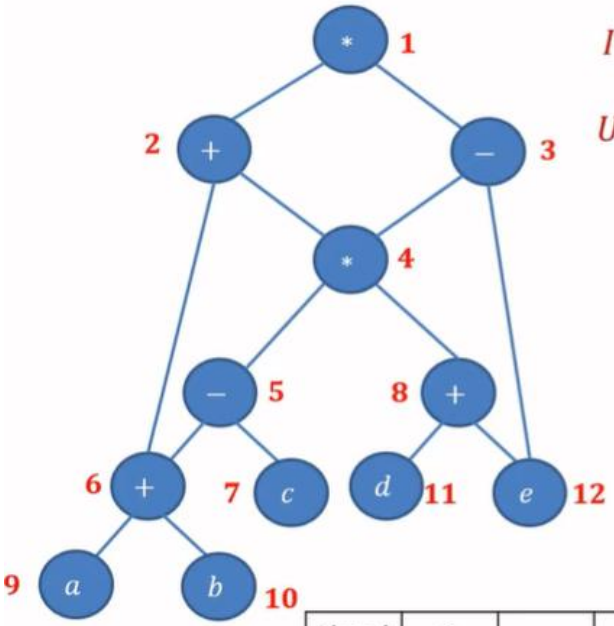
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
Assembly Code

MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
Assembly Code

Algorithm: Heuristic Ordering

Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.

```
while(unlisted interior nodes remain)
    {
       pick up an unlisted node n, whose parents have been listed
       list n;
       while(the leftmost child m of n has no unlisted parent AND is not leaf)
                    {
                    List m;
                    n=m;
                    }
    }
```



$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

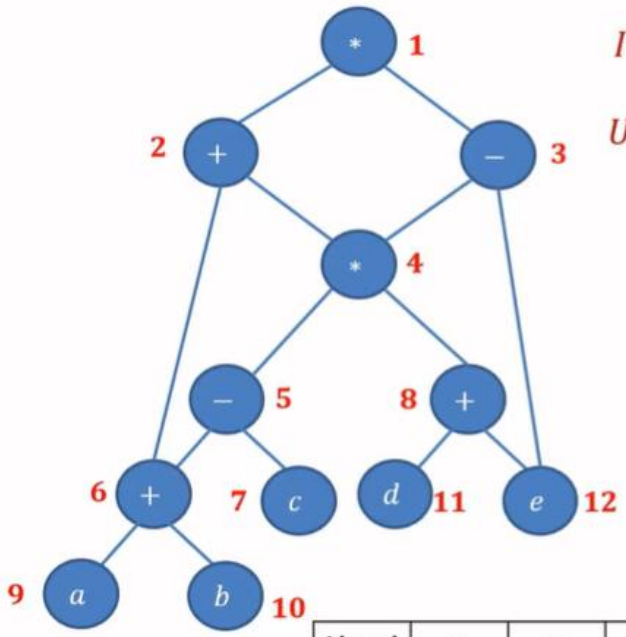
$Unlisted\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

Pick up an unlisted node, whose parents have been listed

1

Leftchild of 1 = 2
Parent 1 is listed so list 2

Listed Node	1						



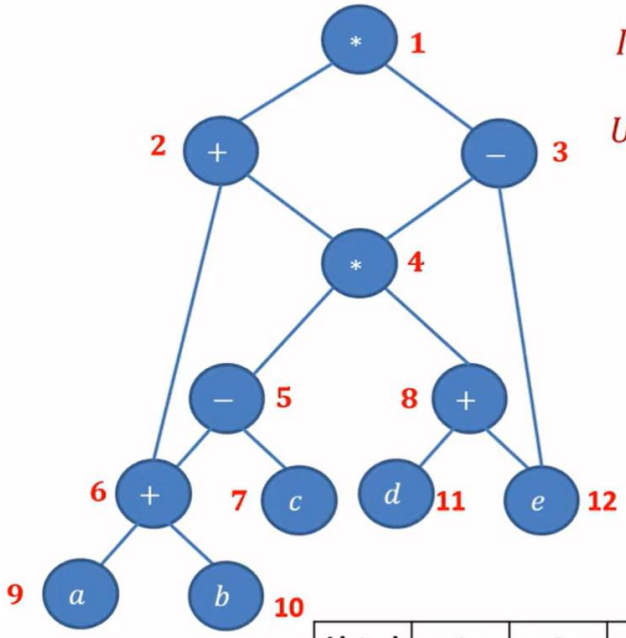
$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ 3\ 4\ 5\ 6\ 8$

Pick up an unlisted node, whose parents have been listed

Leftchild of 2 = 6
Parent 5 is not listed so can't list 6

Listed Node	1	2					



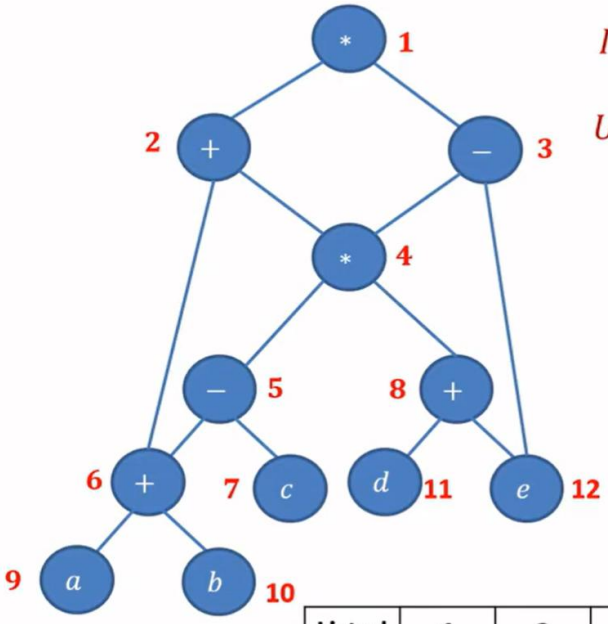
$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ 4\ 5\ 6\ 8$

Pick up an unlisted node, whose parents have been listed

Rightchild of 1 = 3
Parent 1 is listed so list 3

Listed Node	1	2	3				



$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

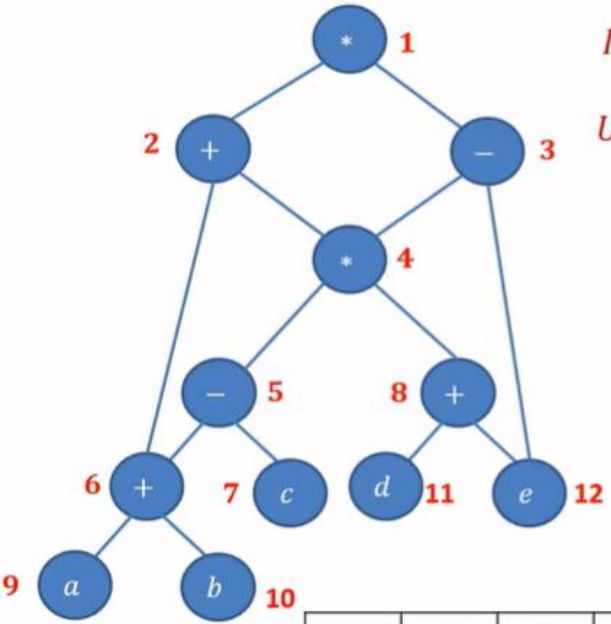
$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ 5\ 6\ 8$

Pick up an unlisted node, whose parents have been listed

Leftchild of 3 = 4
Parent 2,3 are listed so list 4

Listed Node	1	2	3	4			

Leftchild of 4 = 5
Parent 4 is listed so list 5



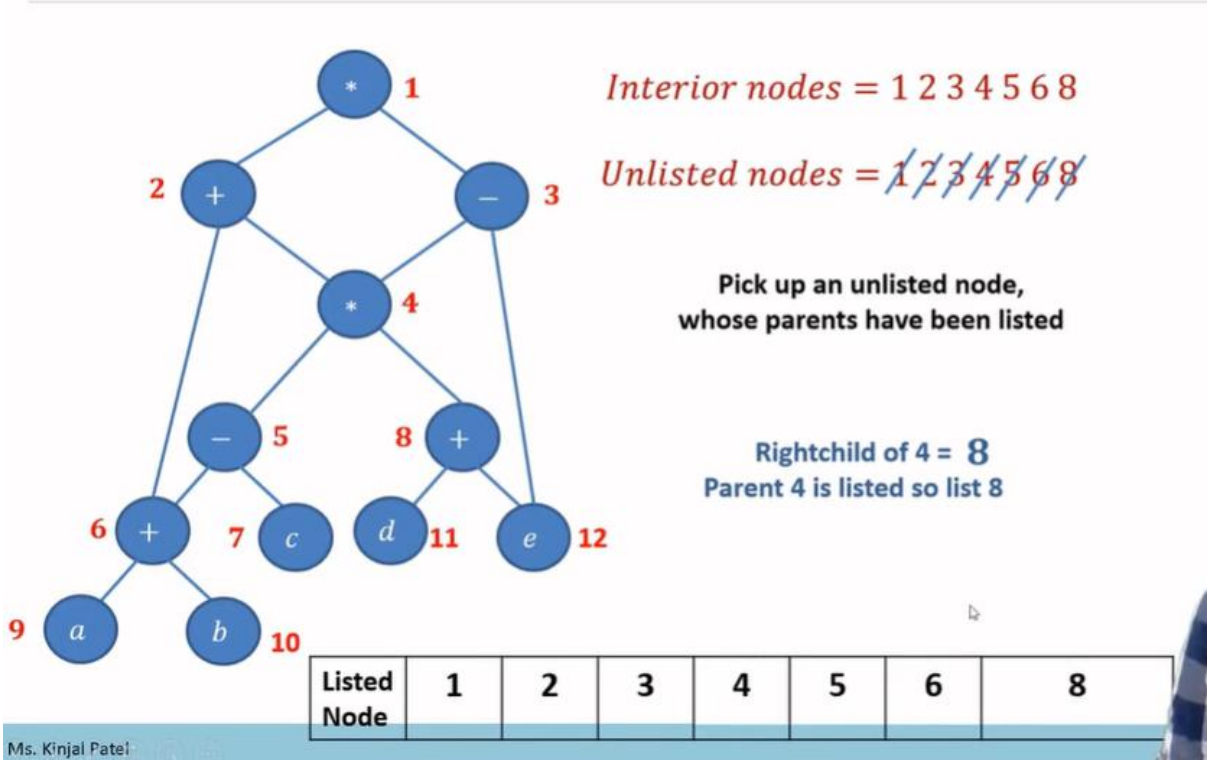
$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ \cancel{5}\ \cancel{6}\ 8$

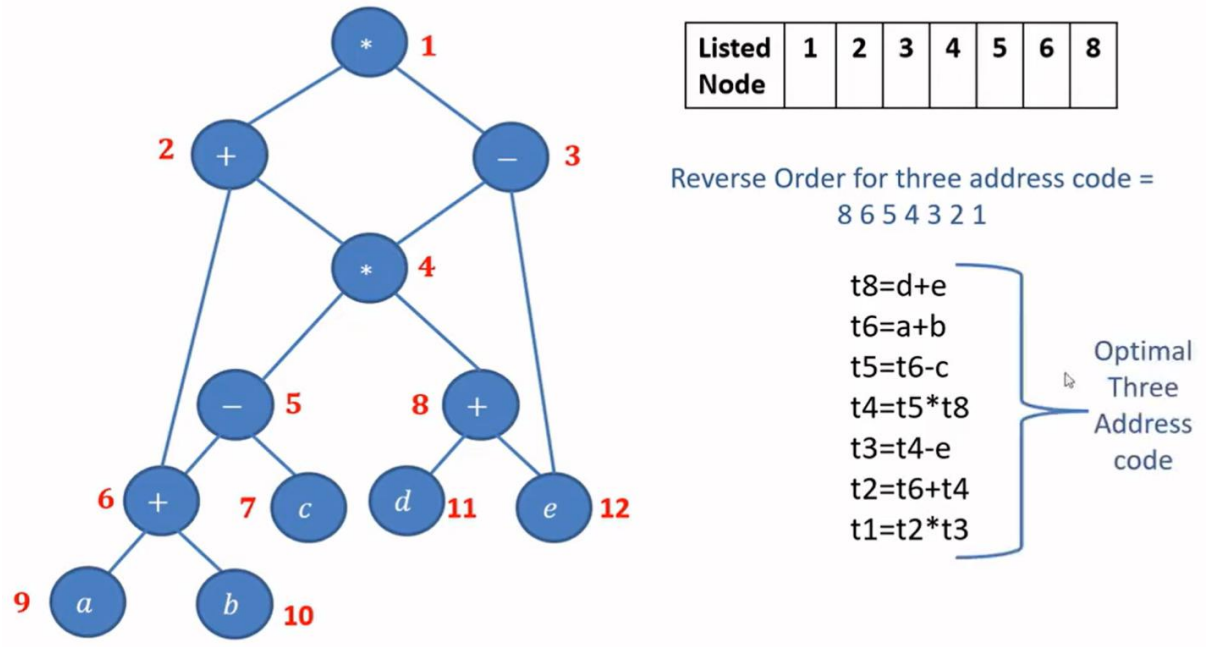
Pick up an unlisted node, whose parents have been listed

Leftchild of 5 = 6
Parent 2,5 are listed so list 6

Listed Node	1	2	3	4	5	6	



Example: Heuristic Ordering



If n is a leaf node ,

If n is a left child then it's value is '1'

Else if n is a right child it's value is '0'

Labeling Algorithm

- The labeling algorithm generates the optimal code for given expression in which minimum registers are required.
- Using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order.
- For computing the label at node n with the label L1 to left child and label L2 to right child as,

$$Label\ (n) =$$
$$\max(L1, L2)\ if\ L1\ not\ equal\ to\ L2$$
$$Label(n) = L1 + 1\ if\ L1 = L2$$

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

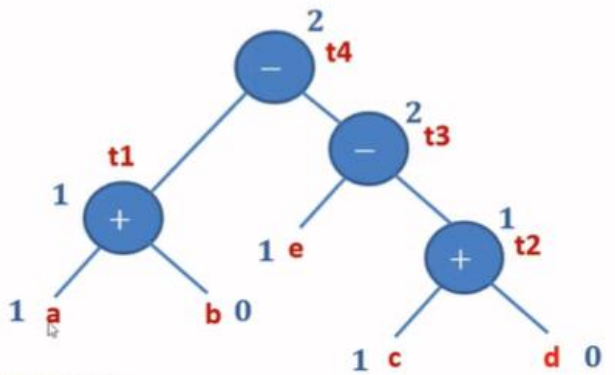
Example: Labeling Algorithm

Example:

$$Label(n) = \begin{cases} Max(l1, l2) & if\ l1 \neq l2 \\ L1 + 1 & if\ l1 = l2 \end{cases}$$

t1:=a+b
t2:=c+d
t3:=e-t2
t4:=t1-t3

Three Address Code



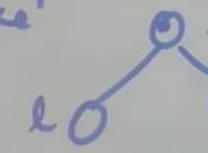
$postorder\ traversal = a\ b\ t1\ e\ c\ d\ t2\ t3\ t4$

Code generation Using DAG

(1) Numbering phase :- Find out that how many registers are needed to evaluate subtree of given node.

(2) Code generation :- Generate code based on the number assigned to each node.

Numbering phase :- Always starts with leaf node, assign 1 or 0 to the node Except leaf node number are assigned like

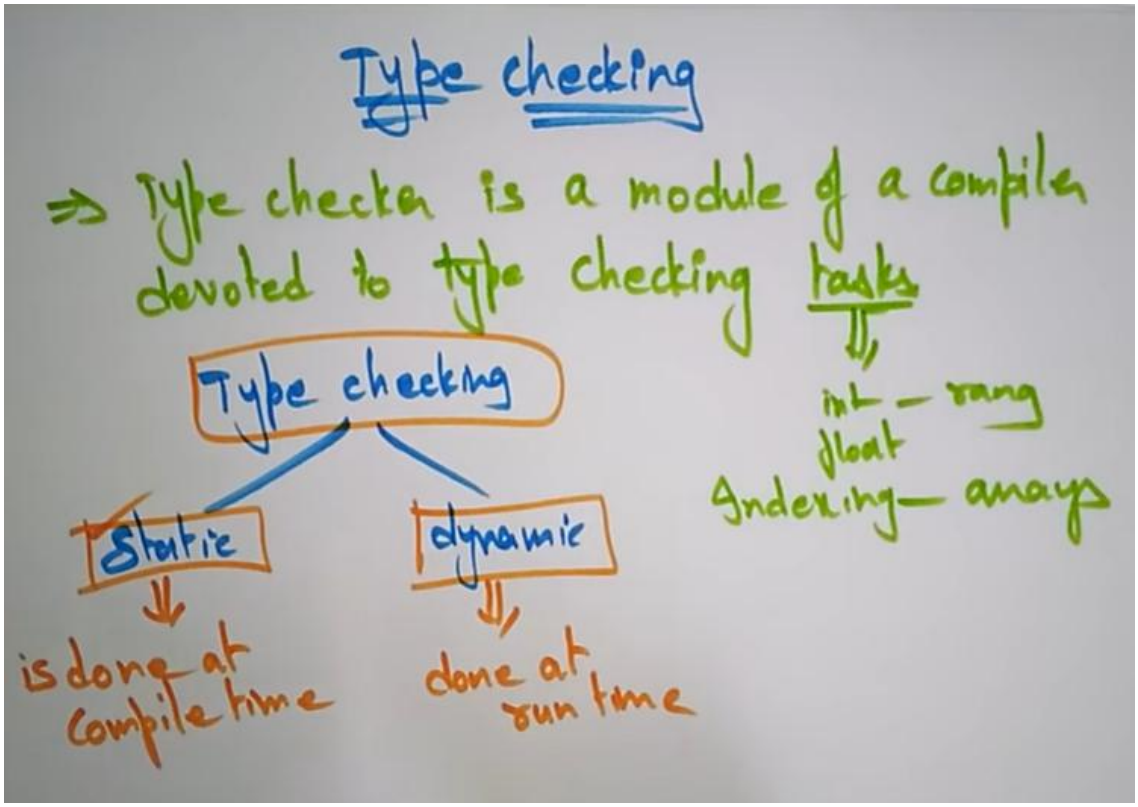


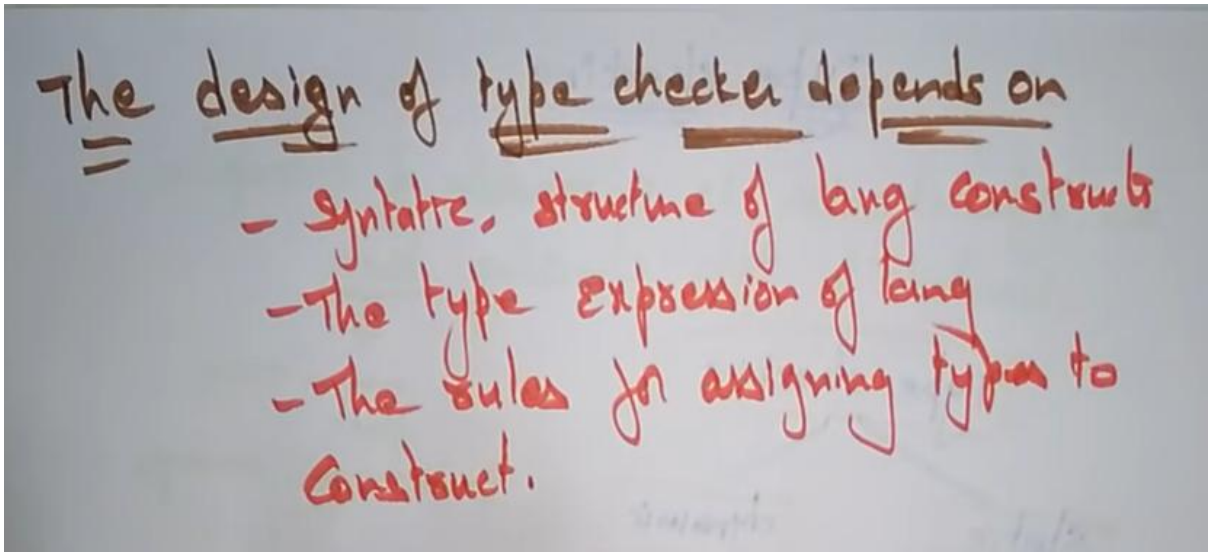
if $l = r$ then $x = l + 1$

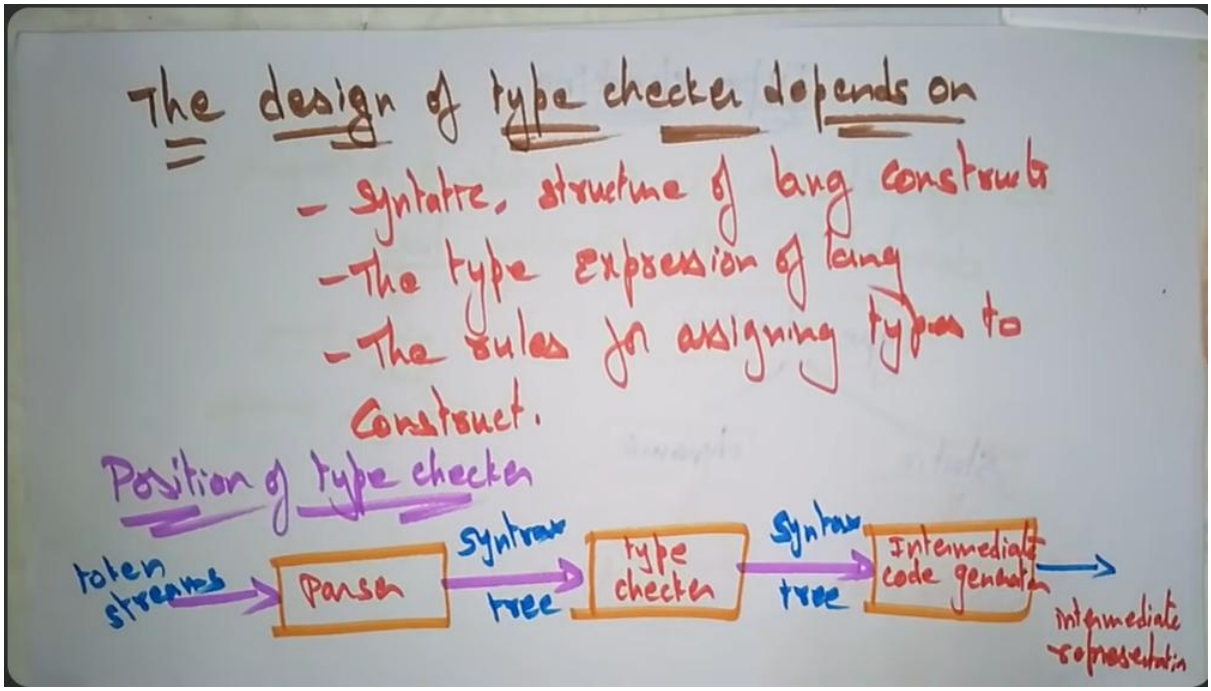
else maximum of l and r
max (l, r)

Same way continue till root

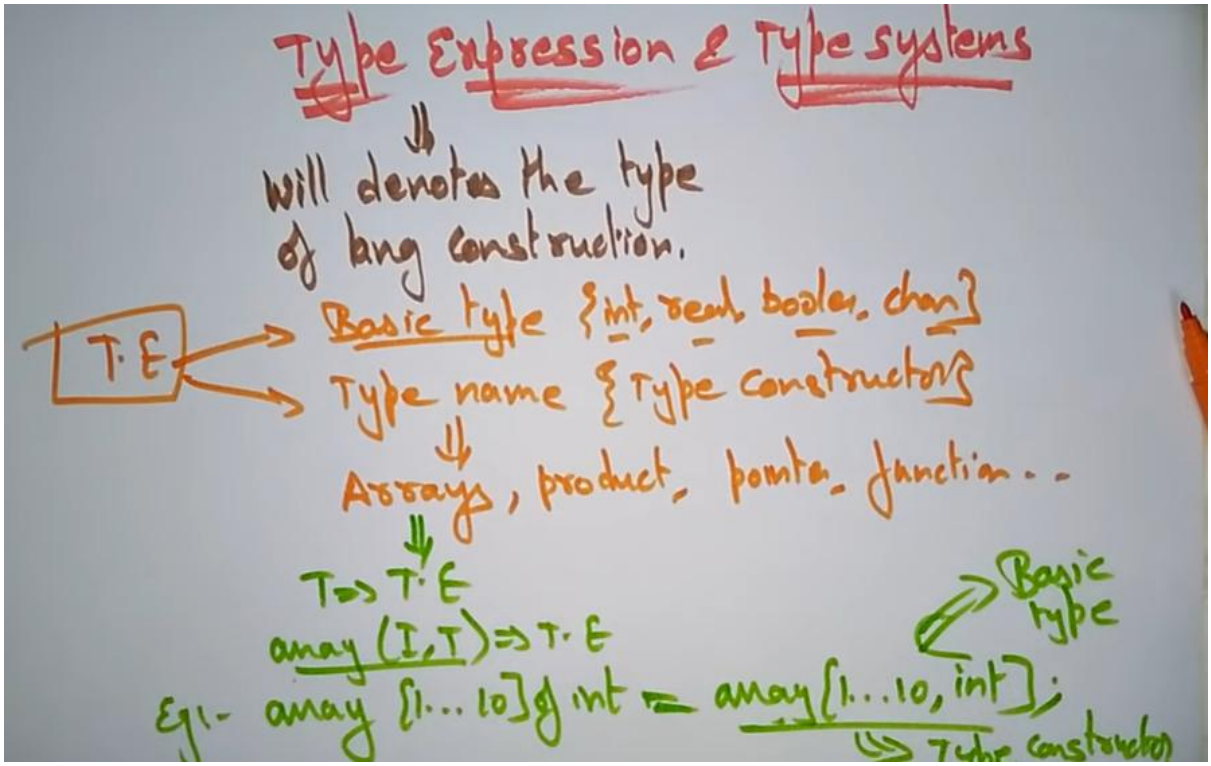
Type checking

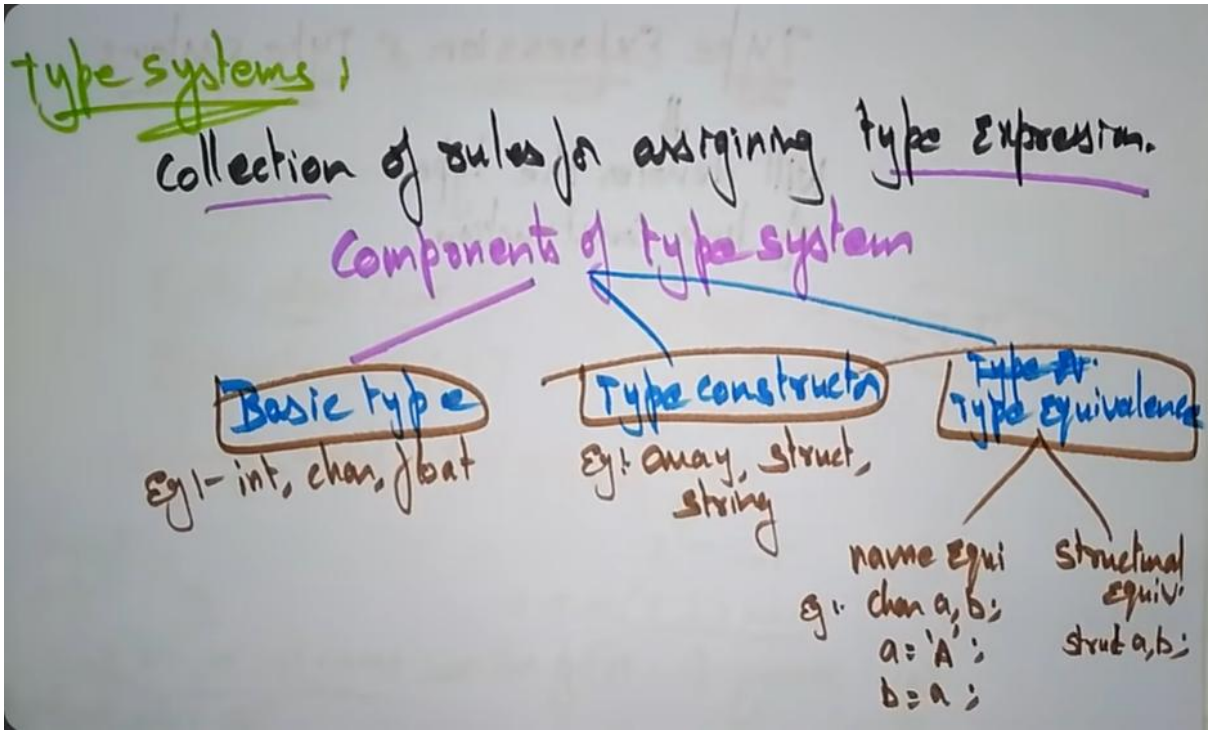




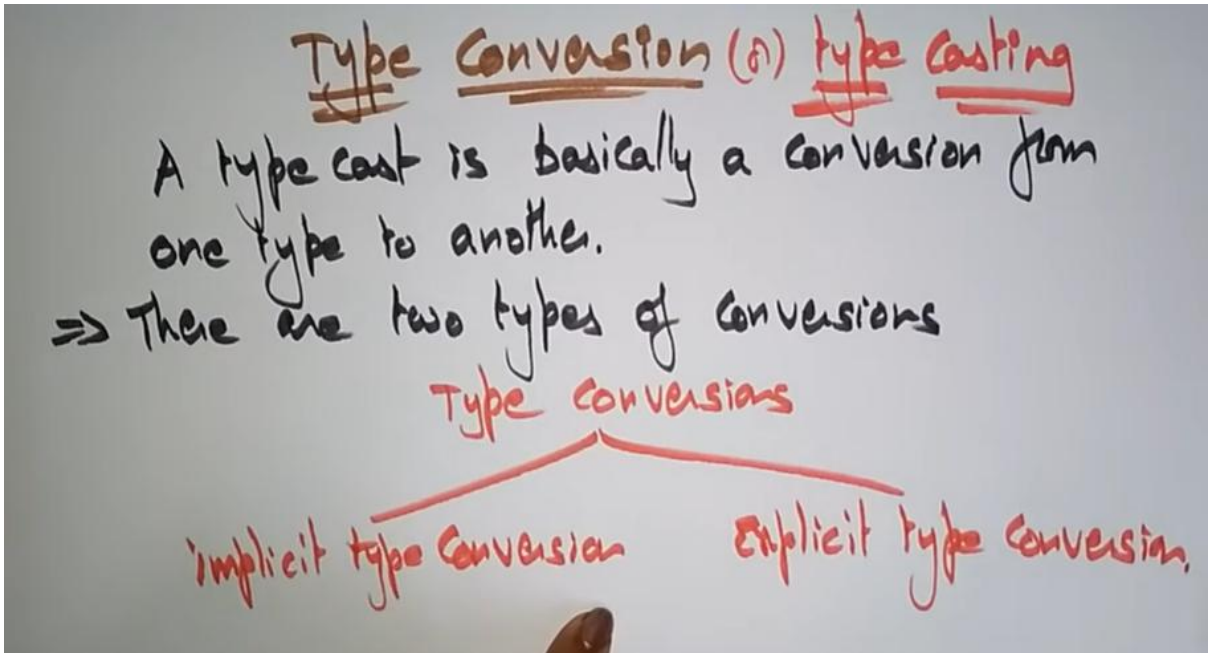


Type Expressions & System





Type Conversion | Implicit & Explicit



Implicit type conversion :-

If a compiler converts one data into another type of data automatically

⇒ There is no data loss

Eg:- short a = 20;
int b = a; // implicit conversion.

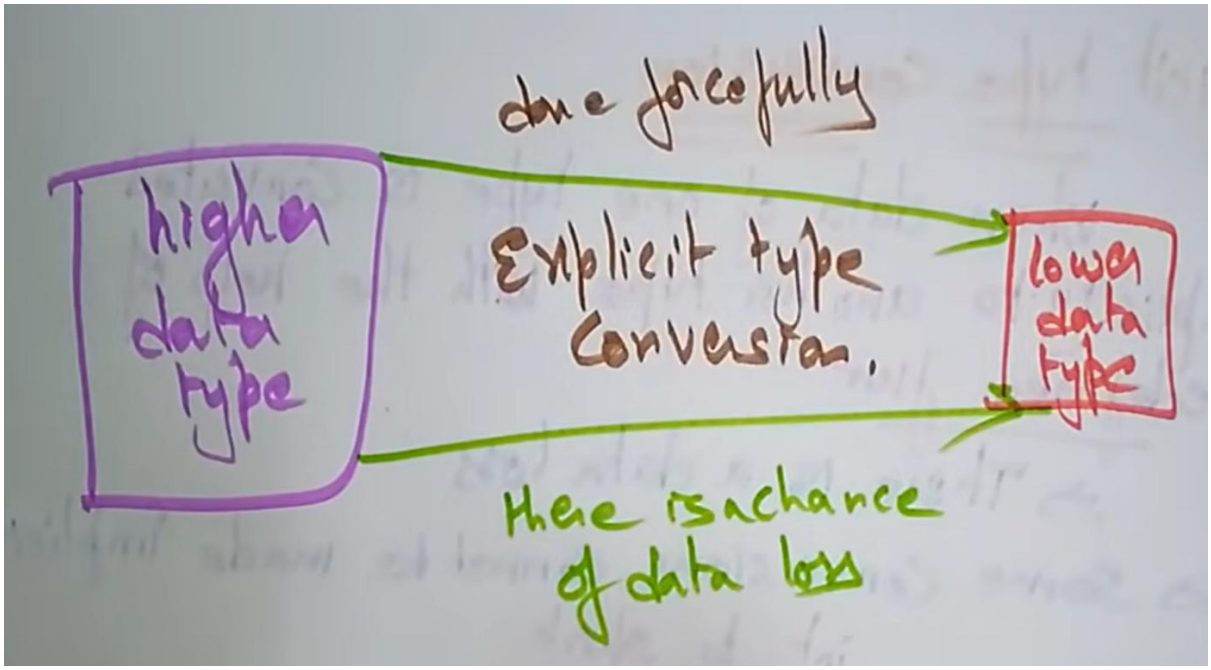
bool → char → short int → int
float ← long ←

Explicit type conversion

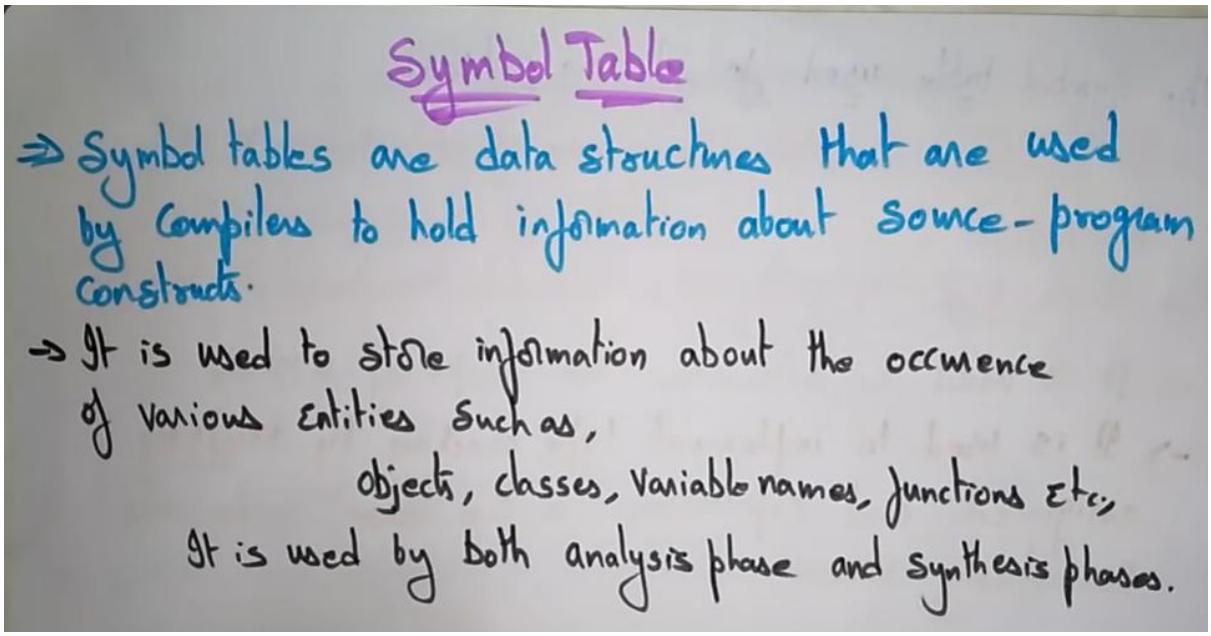
When data of one type is converted explicitly to another type with the help of predefined func

⇒ There is a data loss

⇒ Some conversions cannot be made implicit
int to short



Symbol table



The Symbol table used for following purposes:

⇒ It is used to store the name of all entities in a structured form at one place.

→ It is used to verify if a variable has been declared

→ It is used to determine the scope of a name

→ It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct

A Symbol table can either be linear or a hash table.

⇒ It maintains the entry for each name as,

<symbol name, type, attribute>

Eg: <static, int, salary> { variable declaration is: static int salary }

↓

symbol table stores an entry in this format.

⇒ Use of Symbol Table

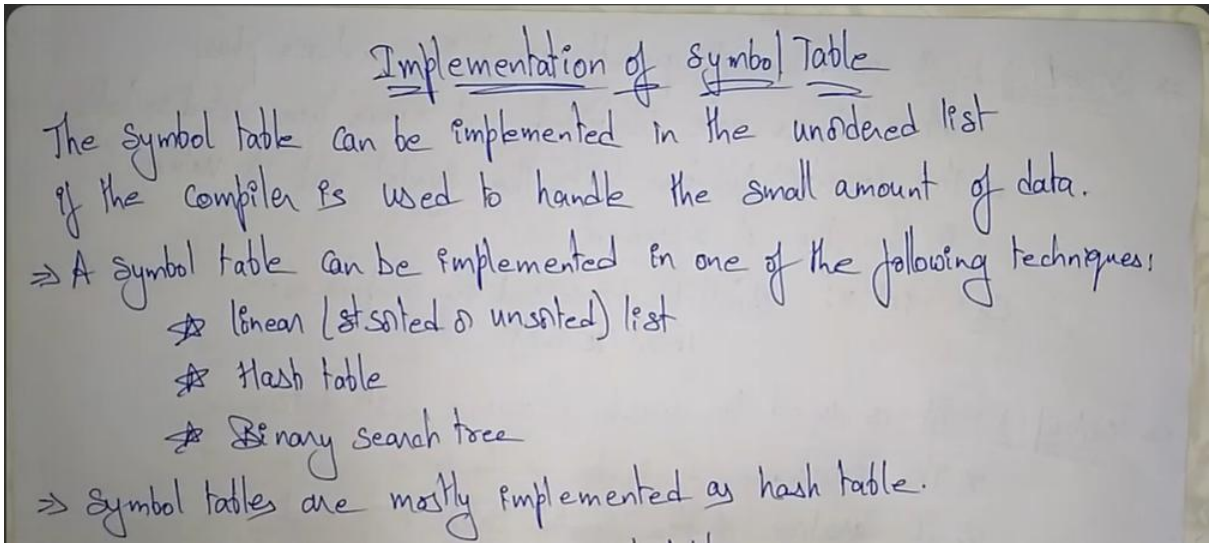
★ Symbol table information is used by the analysis and synthesis phases.

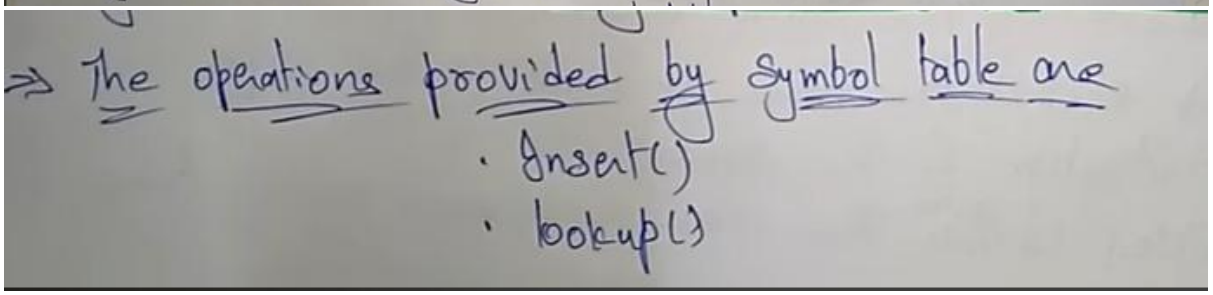
★ To verify that used identifiers have been defined (declared)

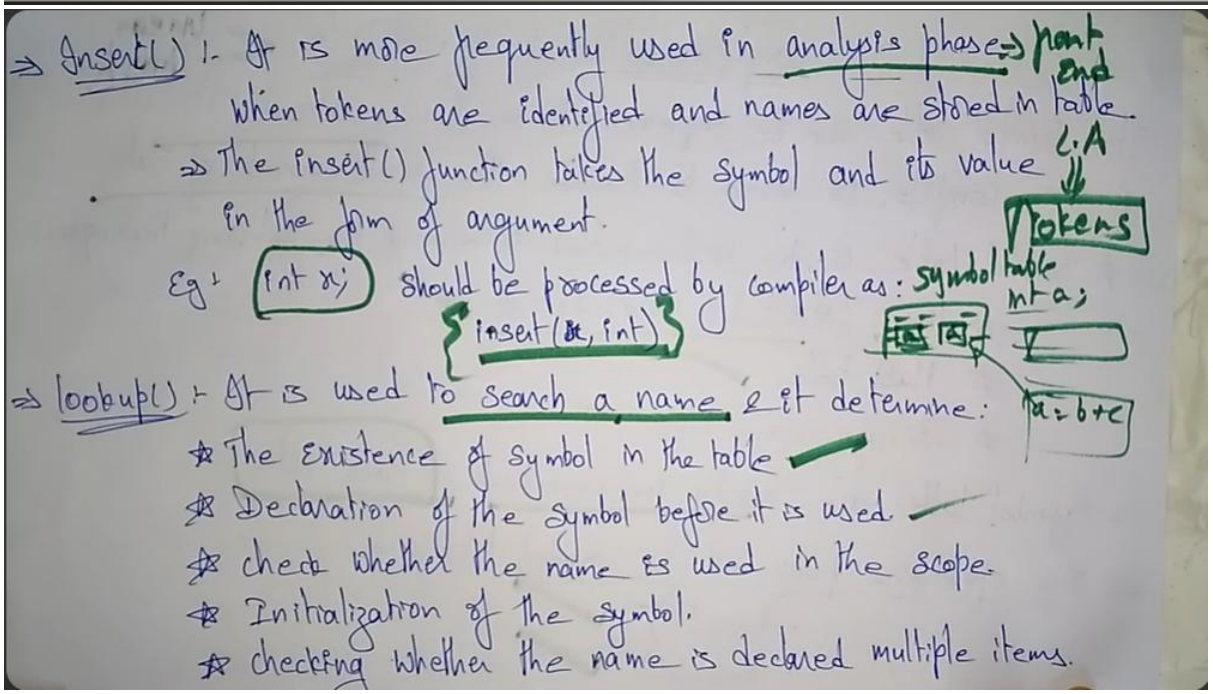
★ To verify that expressions and assignments are semantically correct - type checking

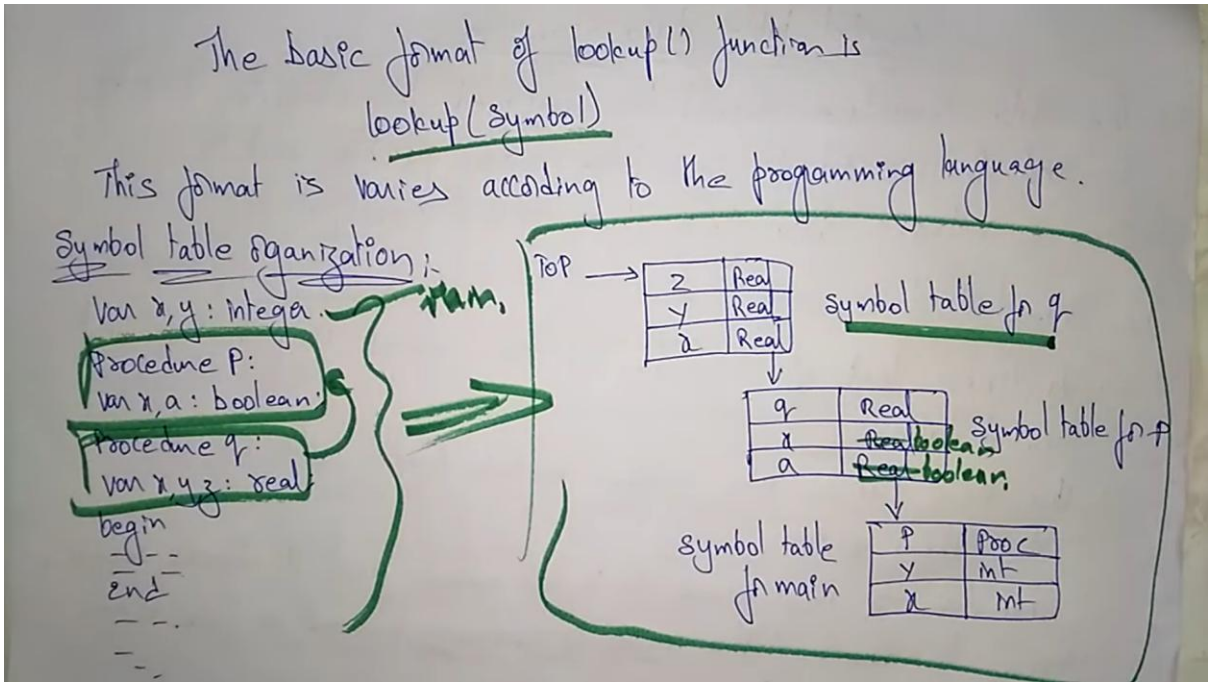
★ To generate intermediate or target code

Symbol table Implementation

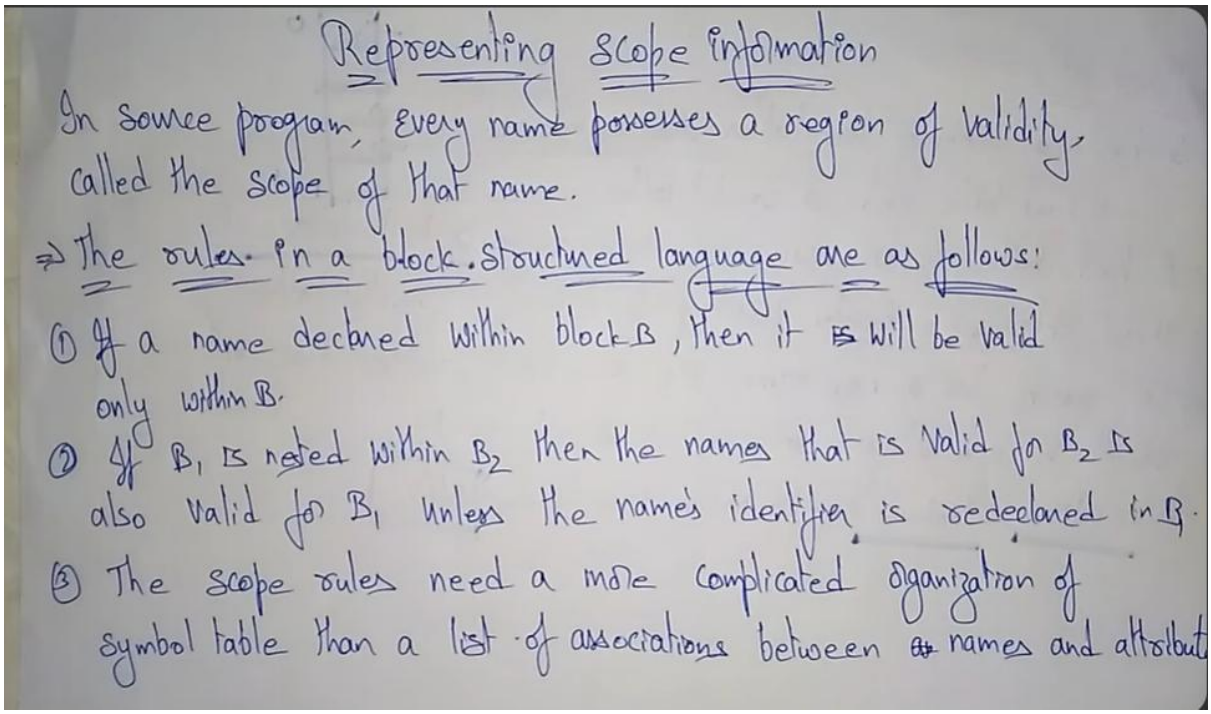


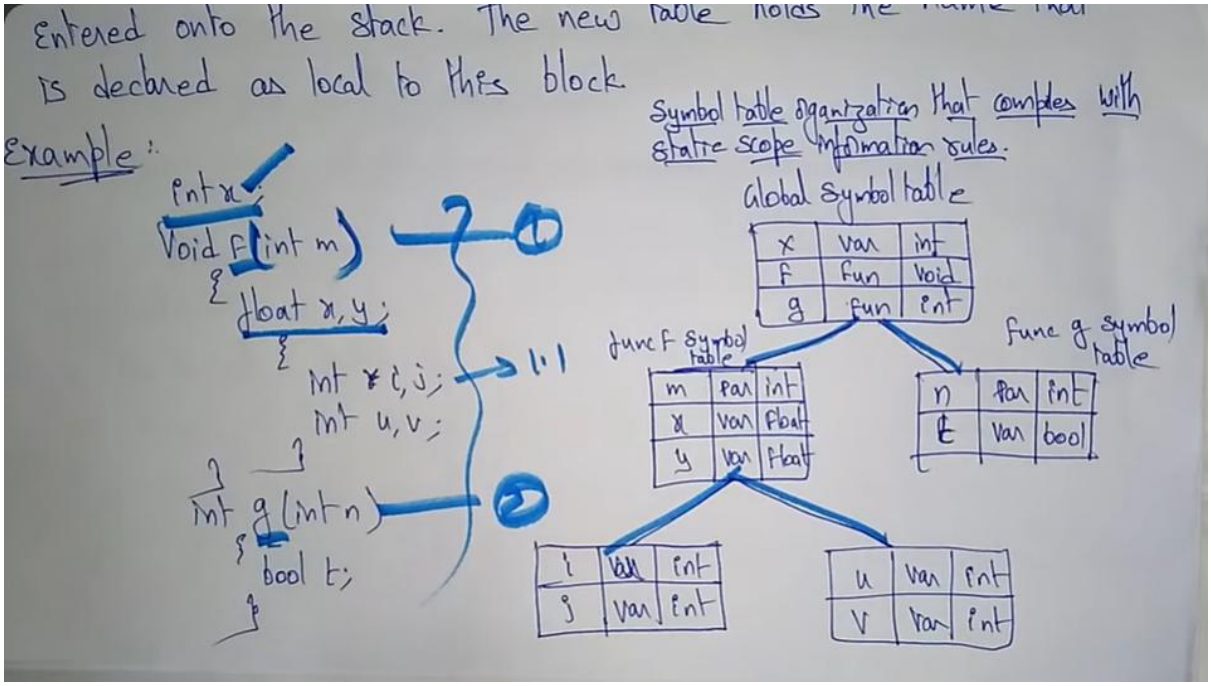




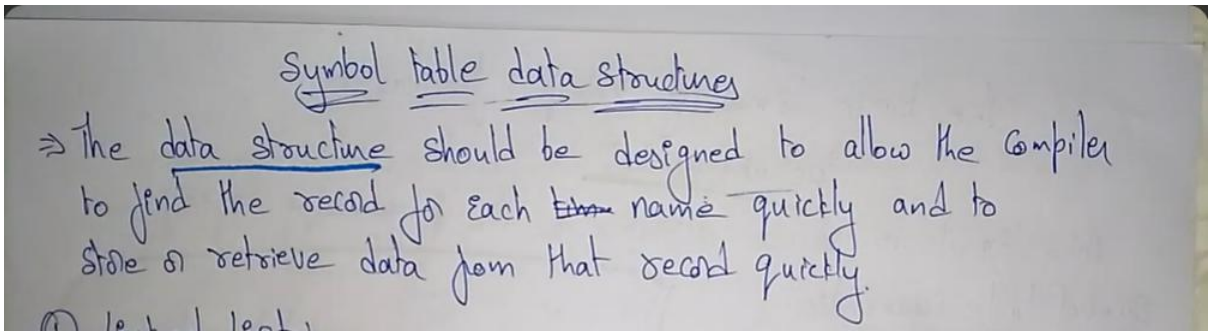


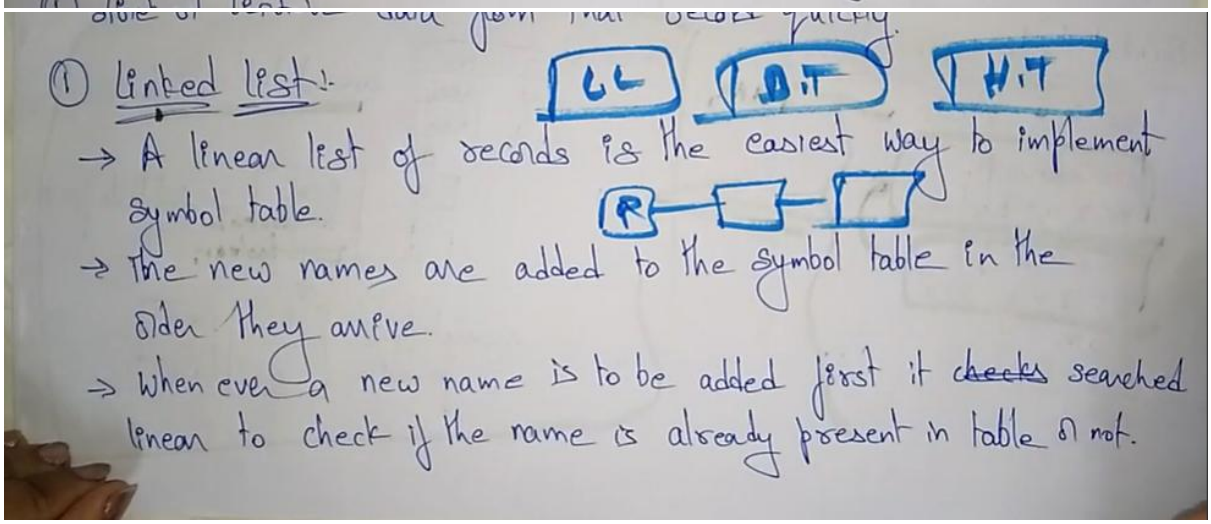
Representing scope Information





Symbol table Structure





Time complexity - O(n)

Advantage - less space, additions are simple

Disadvantage - higher access time

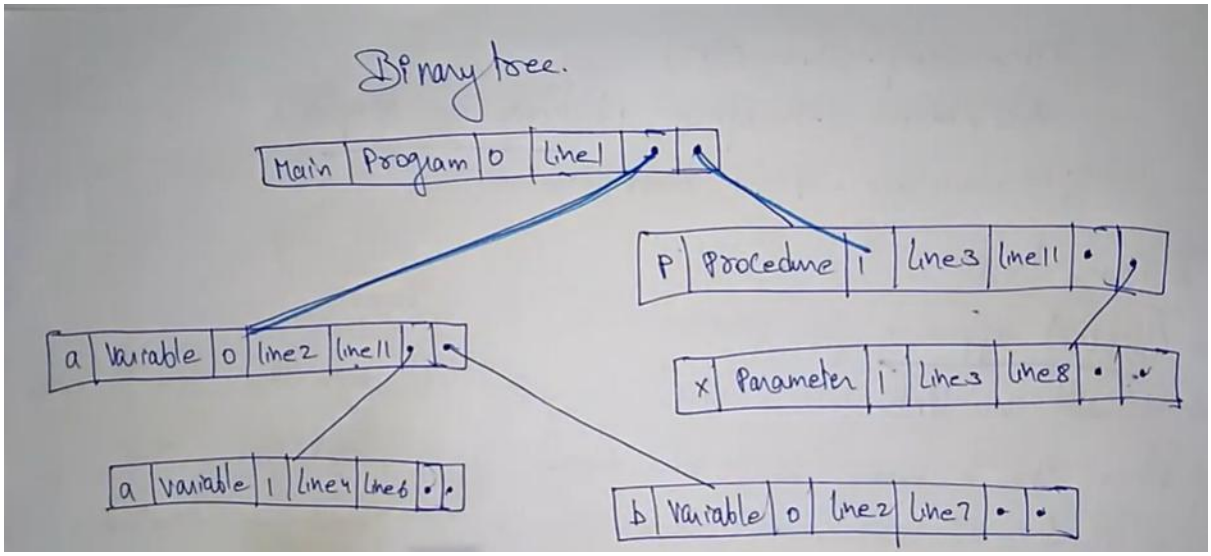
② Binary trees

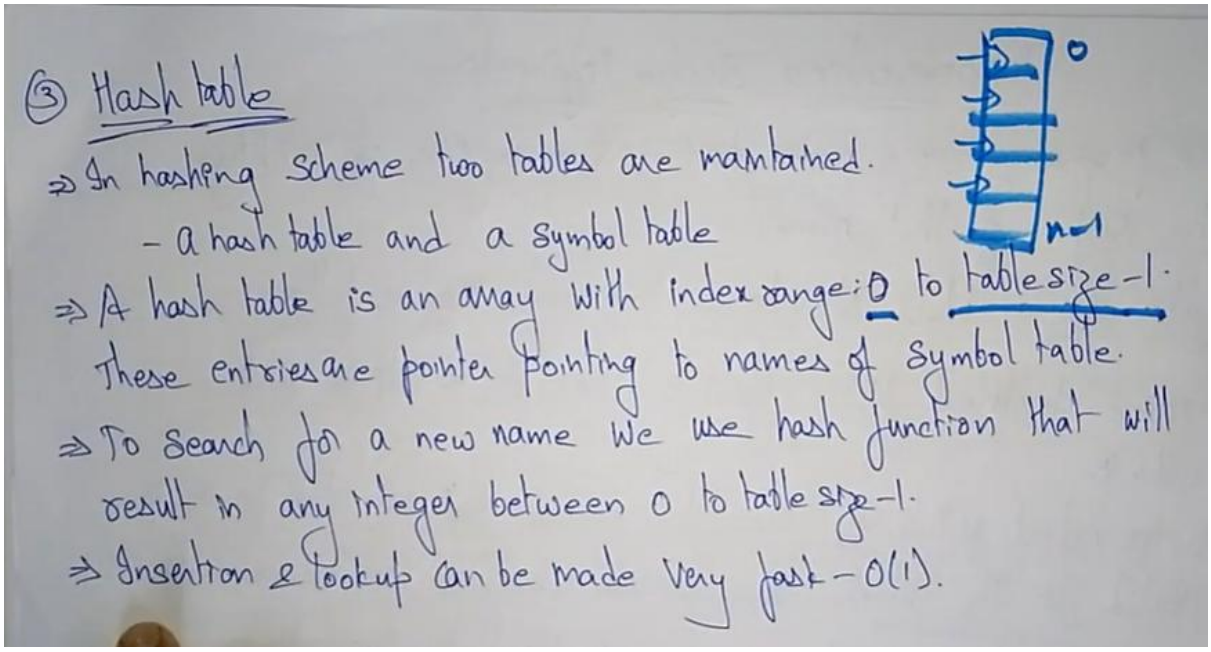
⇒ Efficient approach for symbol table organization.

→ We add two links left & right in each record in the search tree.

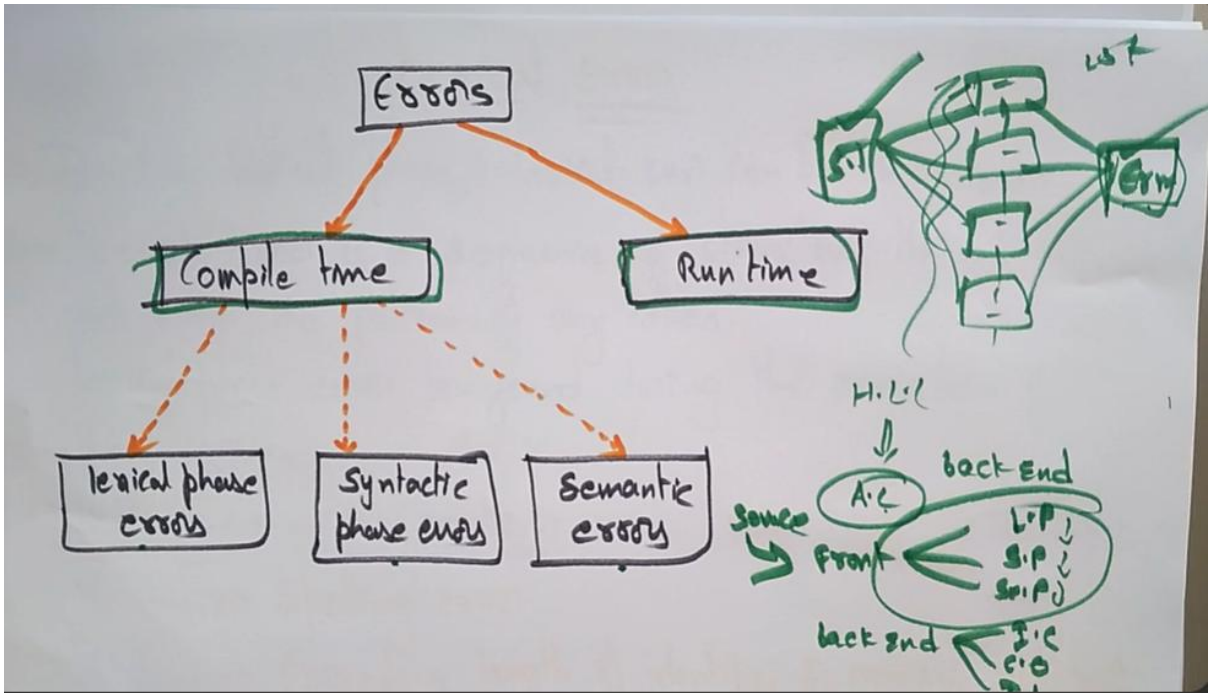
→ When ever a name is to be added, first search in the tree, if it does not exists then a record for new name is created & added at proper position.

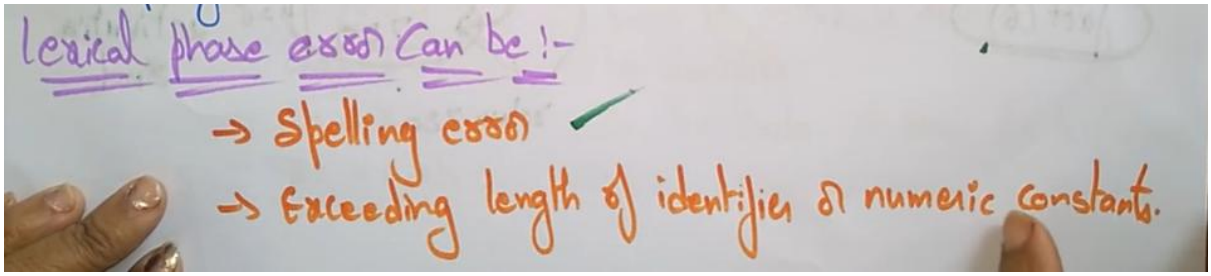
→ This has alphabetical accessibility





Errors | Lexical, Syntax & Semantic





→ Appearance of illegal characters
→ To remove the character that should be present
→ To replace a character with an incorrect character
→ Transposition of two characters.

Example :-

```
Void main()
{
int x=10, y=20;
char *a;
a= &x;
x= 4xab;
}
```

In this code,
4xab is neither a number nor an identifier.
So, this code will show the lexical error.

Syntax Error

2nd phase

→ Syntax error is appears during syntax analysis phase.
→ It is found during the execution of the program.
→ Some syntax error can be :-
- Error in structure
- Missing operator
- Unbalanced parenthesis

Eg :- using "=" when "==" is needed

```
if (number=200)
    cout << "no: is equal to 200";
else
    cout <<
```

In this example. Syntax warning will come,
In this code, if expression used the equal sign which is actualy an assignment operator not the relational operator which test for equality.

Eg2 :-

```
float x= 1.2  // semicolon is missing
```

Eg3 :- Errors in expressions.

```
x= (3+5 ;  // missing closing parenthesis ')'
```

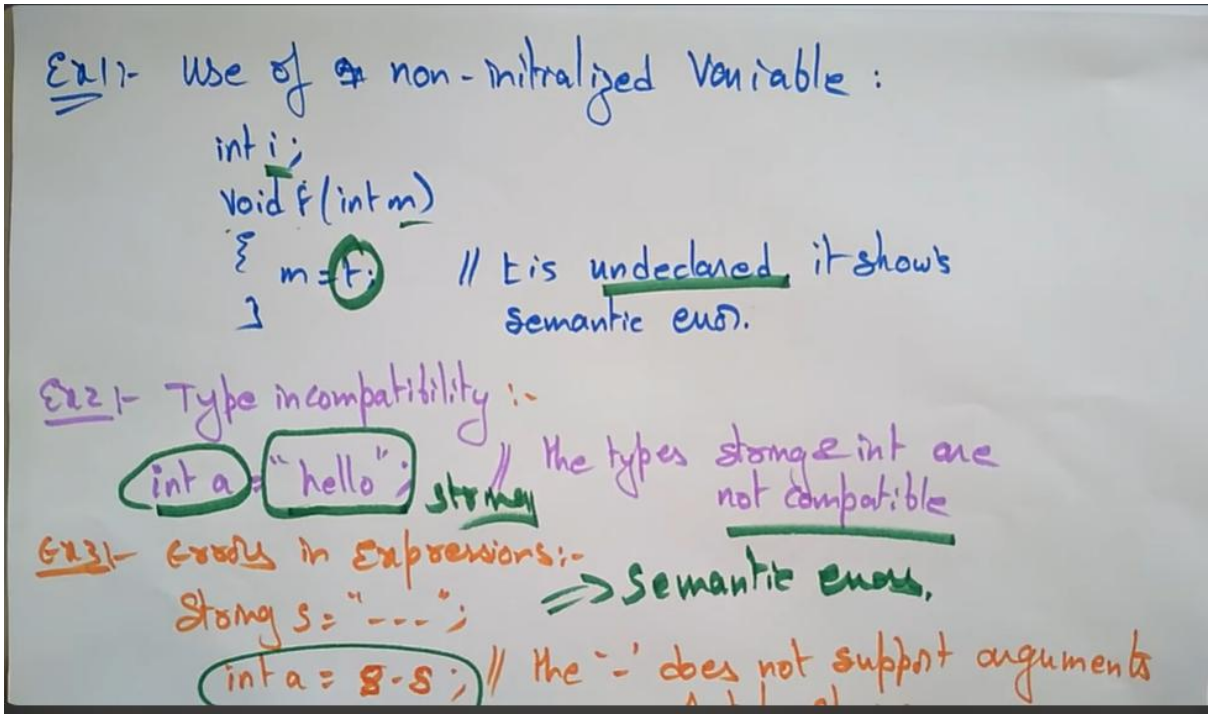
Semantic Error

3rd phase

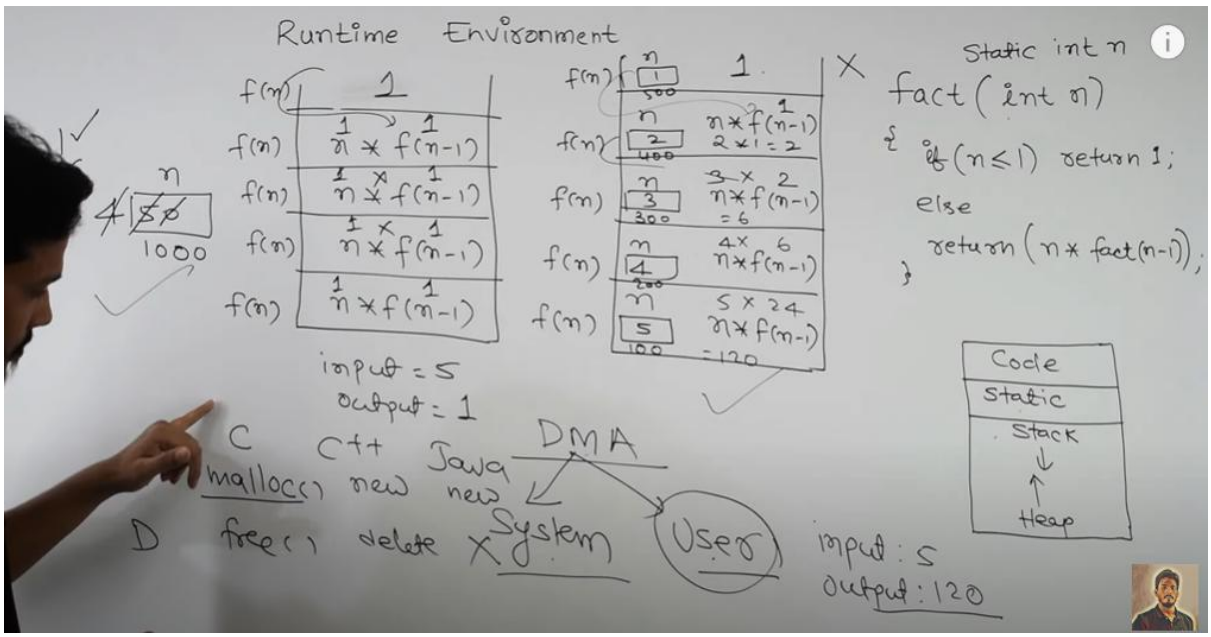
→ Semantic error occurs during semantic phase. It is detected at compile time.

→ Semantic error can be :-

- Incompatible type of operands
- undeclared variable
- Not matching of actual argument with formal argument.



Run Time Environment



Runtime Environment

Static Allocation

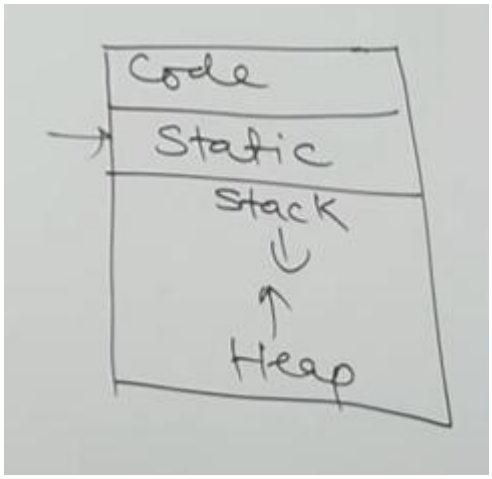
1. Memory allocation is done at compile time.
2. Binding do not change at runtime.
3. Recursion is not supported
4. Size of data objects must be known at compile time.
5. Dynamic data structure not supported.

Stack Allocation

1. Recursion supported
2. Local variable belongs to new activation record.
3. Dynamic data structure not supported.

Heap allocation.

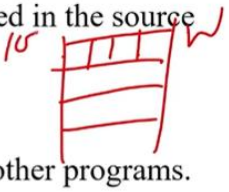
1. Allocation and Deallocation will be done at anytime based on user requirment.
2. Recursion Supported
3. Dynamic data structure supported.



Run-Time Environments

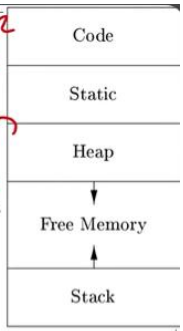
- The compiler creates and manages a run-time environment in which it assumes its target programs are being executed.
- This environment deals with a variety of issues such as

1. The layout and allocation of storage locations for the objects named in the source program,
2. The mechanisms used by the target program to access variables,
3. The linkages between procedures,
4. The mechanisms for passing parameters,
5. The interfaces to the operating system, input/output devices, and other programs.



- Storage Organization:
- The executing target program runs in its own **logical address space** in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.
- The run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory.
- A byte is eight bits and four bytes form a machine word . Multibyte objects are stored in consecutive bytes and given the address of the first byte.

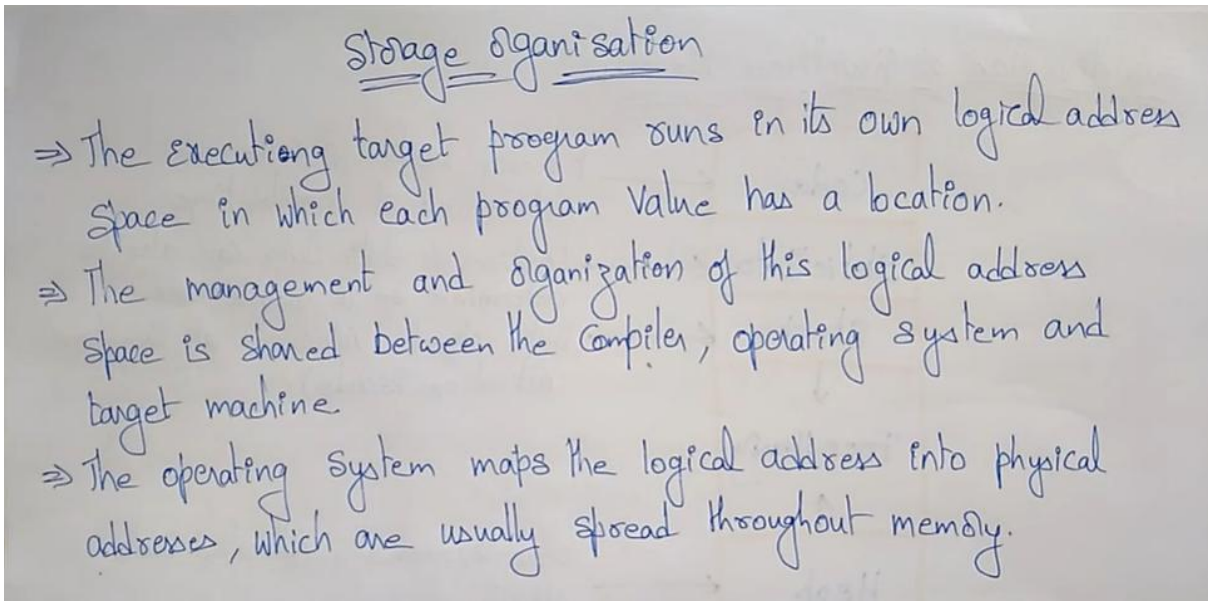
- The amount of storage needed for a name is determined from its type.
- A character array of length 10 needs only enough bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment , leaving 2 bytes unused.
- Space left unused due to alignment considerations is referred to as *padding*..
- When space is premium, a compiler may *pack* data so that no padding is left.
- *Subdivision of run-time memory into code and data areas→*

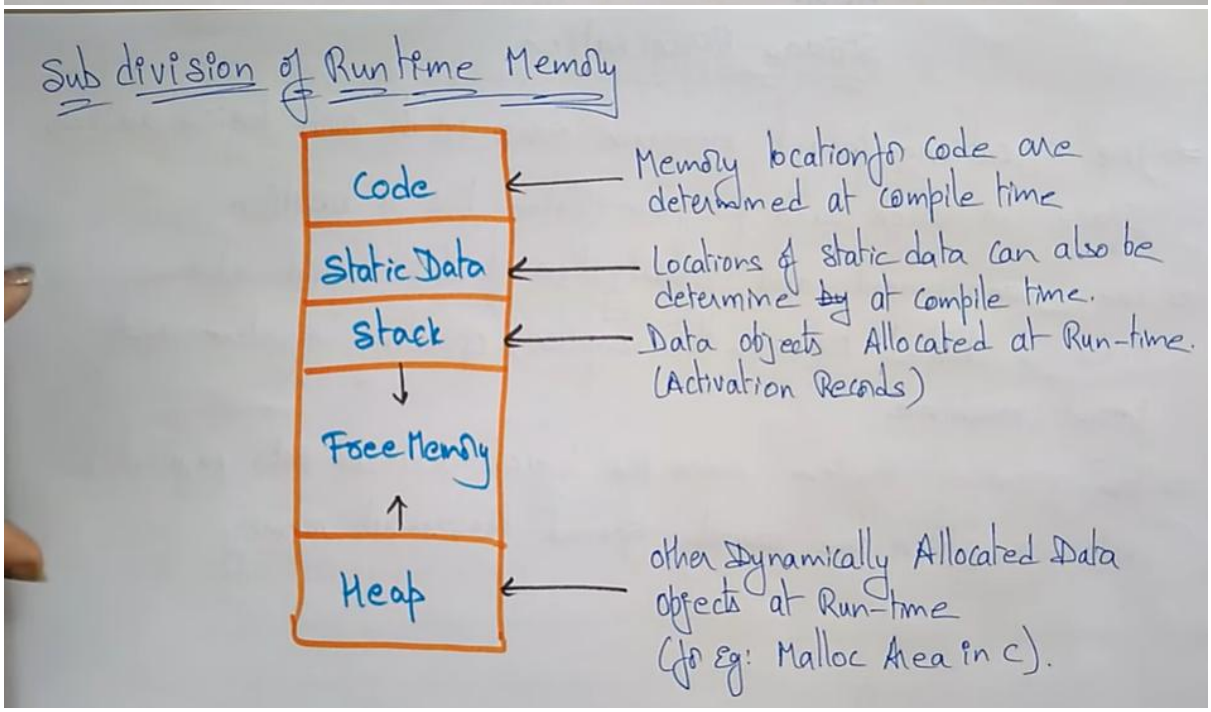


- **Code** :The size of the target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area *Code*.
- **Static**: The size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called *Static*.
- **Stack & Heap** :To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space.
- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed.
- The *stack* is used to store data structures called activation records that get generated during procedure calls.
- The stack grows towards lower addresses, the heap towards higher.

Storage organisation | Runtime memory

[Storage Allocation Strategies in Compiler Design - GeeksforGeeks]





- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form.
- object of multibyte is stored in consecutive bytes and gives the first byte address
- Runtime storage can be subdivide to hold the different components of an executing program:
  1. Generated executable code
  2. static data objects
  3. Dynamic data object - heap
  4. Automatic data objects - stack

Activation record

Activation Record

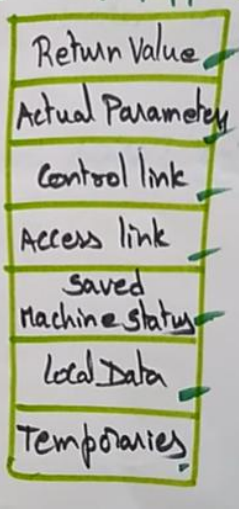
⇒ Control stack is a runtime stack which is used to keep track of the live procedure activations,
  i.e., It is used to find out the procedures whose execution have not been completed.

⇒ When the activation begins then the procedure name will push on to the stack and when returns (activation ends) then it will popped.

⇒ Activation record is used to manage the information needed by a single execution of a procedure.

⇒ An Activation record is pushed into the stack when a procedure is called, and is popped when the control returns to the caller function.

calling popped.



* The Contents of Activation records:

Return value:- It is used by called procedure to return a value to calling procedure.

Actual Parameters:- It is used by calling procedure to supply parameters to called procedure.

Control link:- It points to activation record of the caller

Access link:- It is used to refer to non local data held in other activation records.

Saved Machine status:- It holds the information about status of machine before the procedure is called.

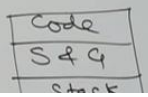
Local data:- It holds the data that is local to the execution of the procedure.

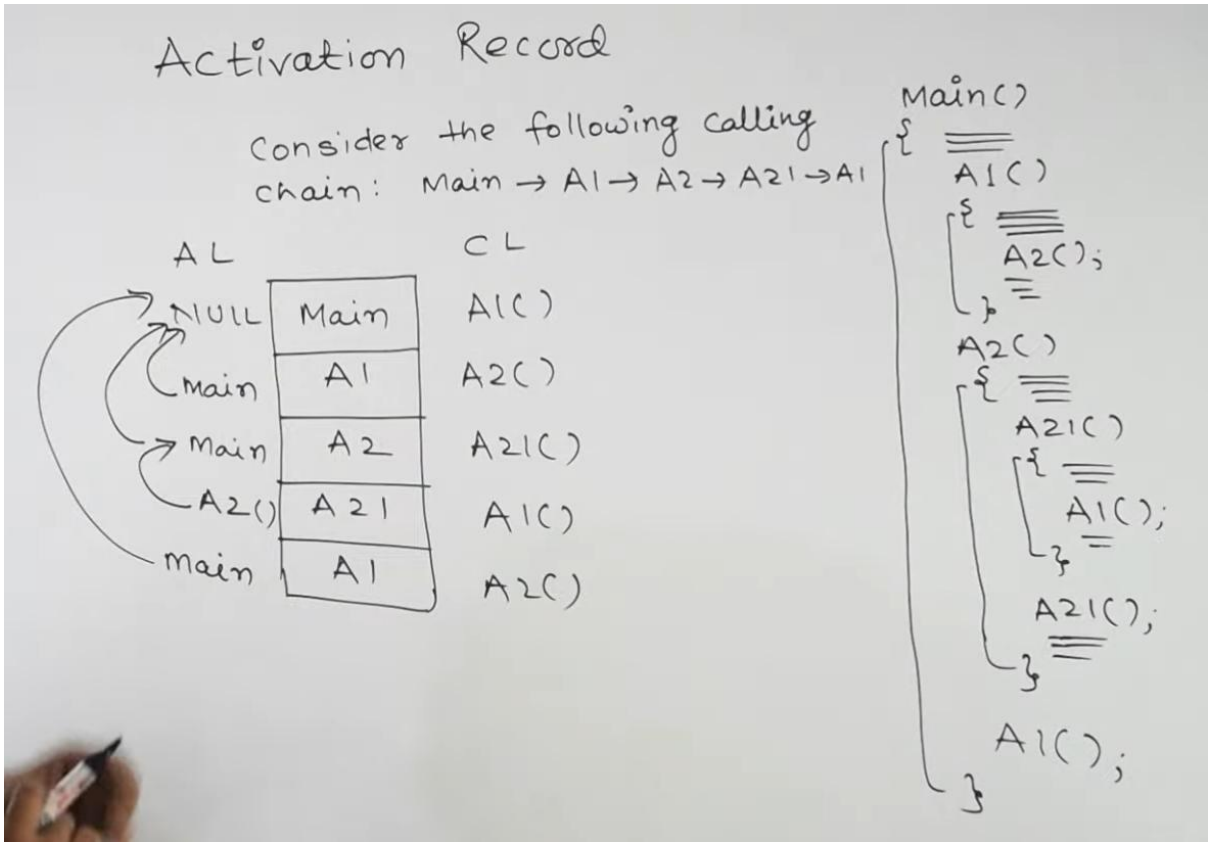
Temporaries:- It stores the value that arises in the evaluation of an expression

Activation Record

1. Local Variable : Holds the data that is local to the execution of the function.
2. Temporary Values : Stores the values that arises in the evaluation of an expression.
3. Machine Status : Holds the info. about the status of the machine just before the f^n call.
4. Access Link : It is used to refer to non-local data held in other activation record.
5. Control Link : Store the address of activation record of the caller function.
6. Actual Parameters: Store A.P. that are used to send input to the caller function.
7. Return Values : To store the result of function call.

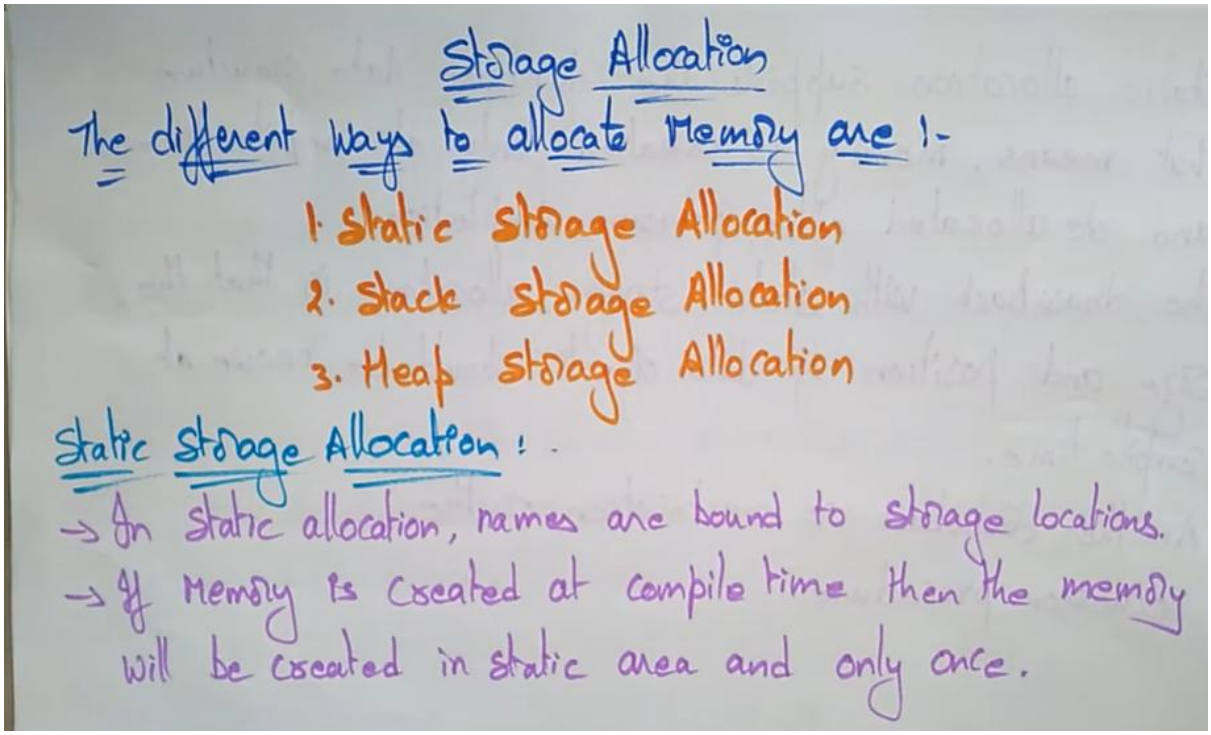
200 ... P(X,Y) PC= ① ____ A() CS





Access Link - It is the function where the particular function is defined
Control Link - It is the function which is called in that particular function

Storage allocation | Static, Stack & Heap



→ Static allocation supports the dynamic data structure that means, memory is created only at compile time and de allocated after program completion.

→ The drawback with static storage allocation is that the size and position of data objects should be known at compile time.

→ Another drawback is restriction of the recursion procedure.

→ Stack Storage Allocation :-

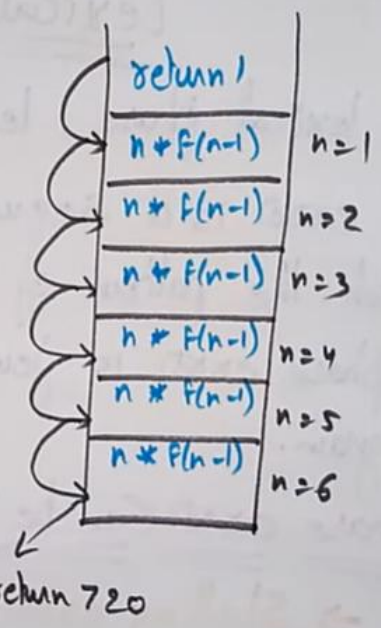
- Storage is organized as a stack.
- Activation records are pushed and popped
- Activation record contains the locals so that they are bound to fresh storage in each activation record.
- The value of locals is deleted when the activation ends
- It works on the basis of LIFO and this allocation supports the recursion process.

→ Heap storage Allocation :-

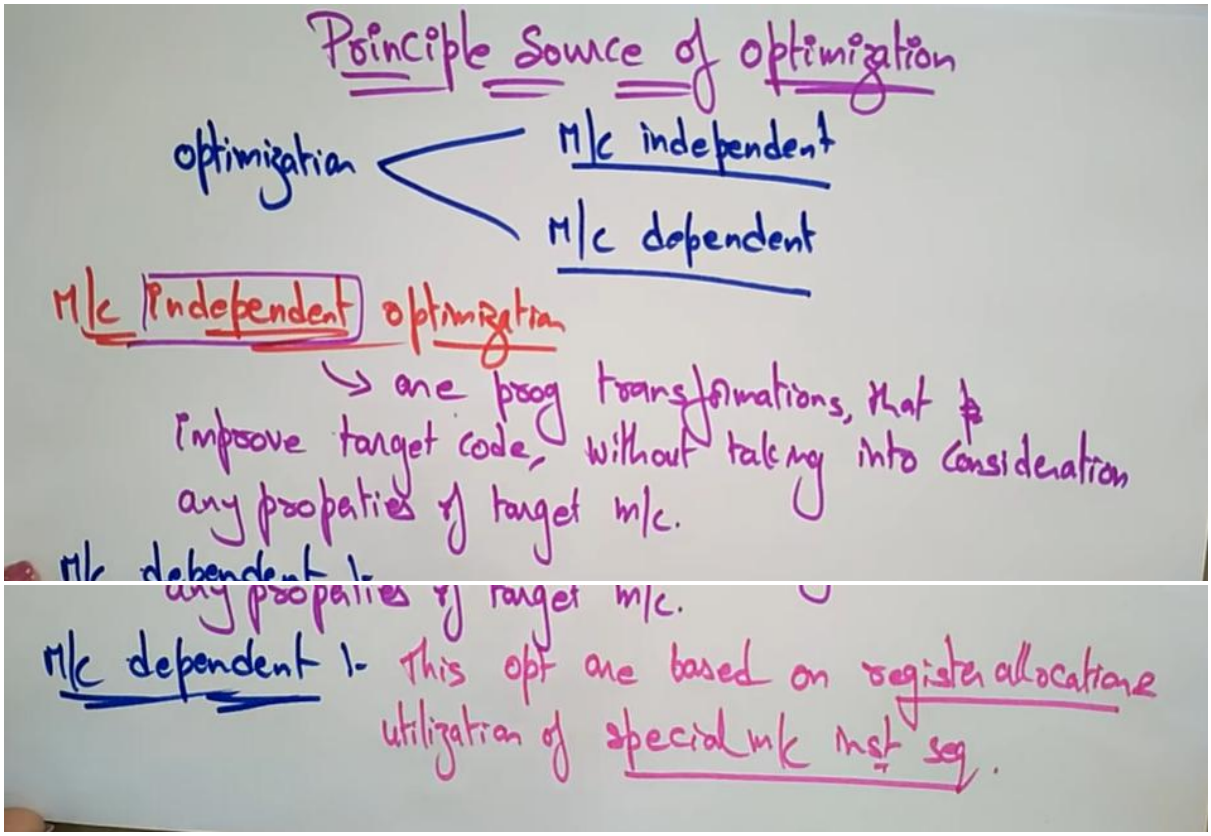
- It is the most flexible allocation scheme
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

Example :-

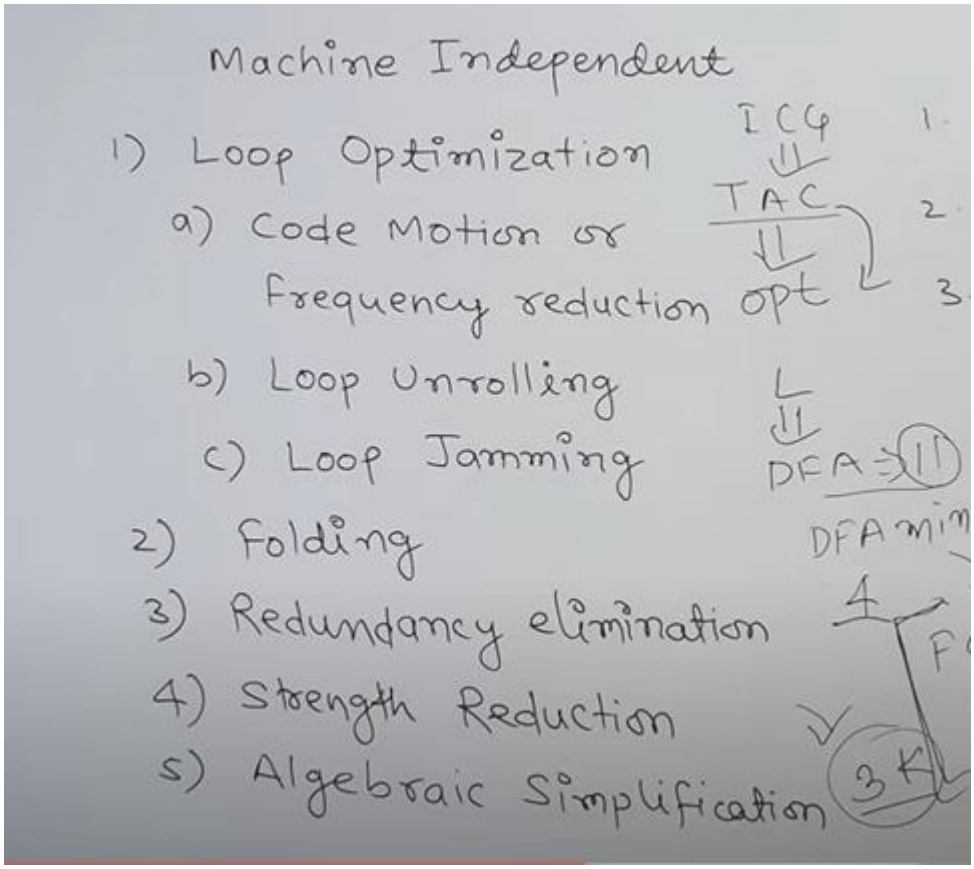
```
fact (int n)
{
if (n<=1)
 return 1;
else
 return(n*fact(n-1));
}
fact (6)
```

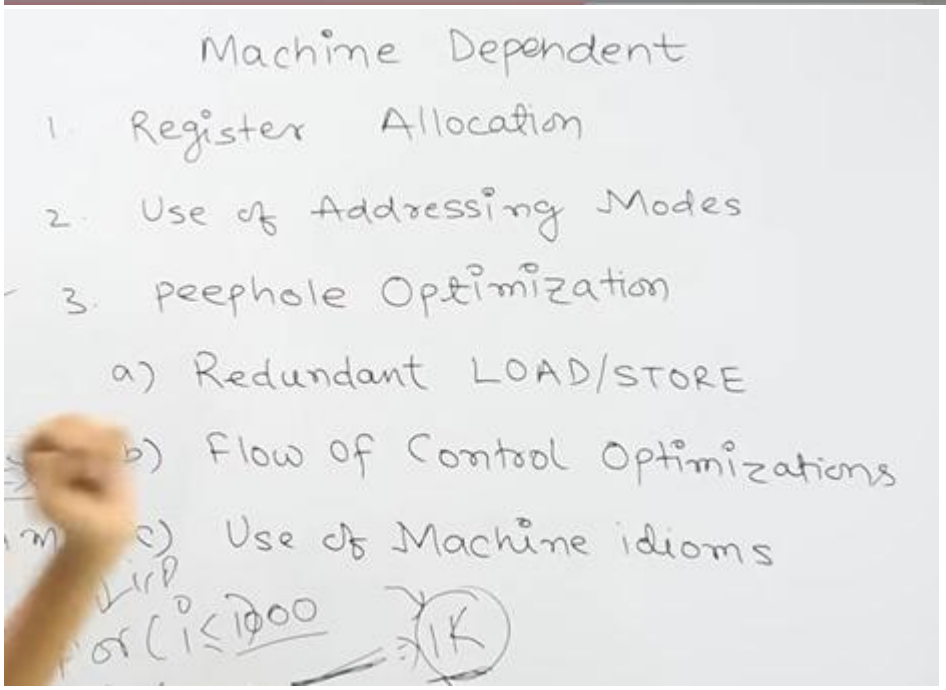


Source of Optimisation | Principal



Code Optimisation technique





Code optimization techniques

- Compile time Evaluation
- Variable propagation
- dead code elimination
- Code motion
- induction Variable & strength reduction

① compile time evaluation :-

(a) $z = \{5 \times (45.0/5.0) \times 8\}$

!!, at compile time

(b) $x = 5.7$

$y = x / 3.6$

evaluate $\left\{\frac{5.7}{3.6}\right\}$

② Variable propagation :-

before optimization

$c = a \times b$

till $x = a$

$d = x * b + y$

After optimization

$c = a \times b$

$x = a$

till $d = a * b + y$

③ Dead Code elimination :-

Before elimination

$c = a \times b$

till $x = b$ ⇒ dead state

$d = a \times b + y$

After elimination

$c = a \times b$

till $d = a \times b + y$

(4) Code Motion :-

- It reduces the evaluation frequency of expression.
- It brings loop invariant stmt out of the loop

```
a = 200;
while (a > 0)
{
    b = x + y;
    if (a % 1 == 0)
    Plf ("%d", a);
}
```

```
a = 200;
b = x + y;
while (a > 0)
{
    if (a % b == 0)
    Plf ("%d", a);
}
```

(5) Induction variable & strength reduction :-

⇓

Eg: Before reduction

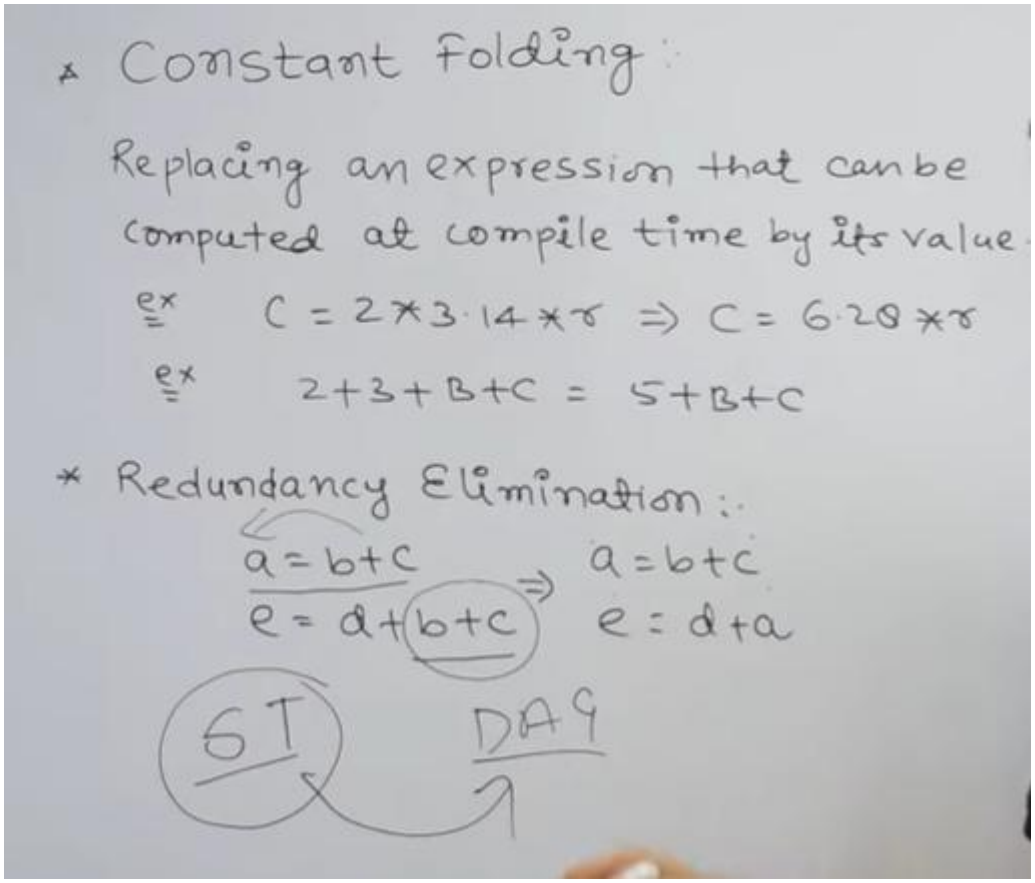
```
i = 1;
while (i < 10)
{
    y = i * 4;
}
```

⇓

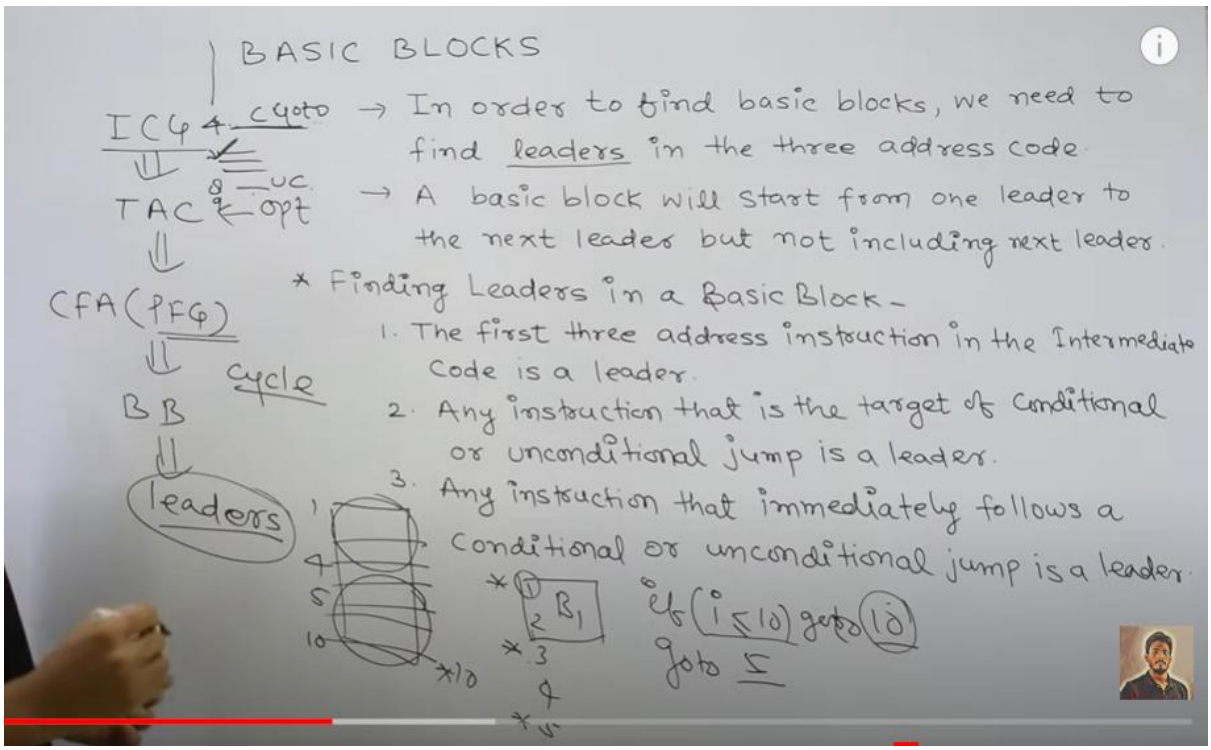
replace high strength operator by low strength operator

After reduction

```
i = 1;
t = 4;
{while (t < 10)
    y = t;
    t = t + 4;
}
```



Basic Blocks and Flow Graphs

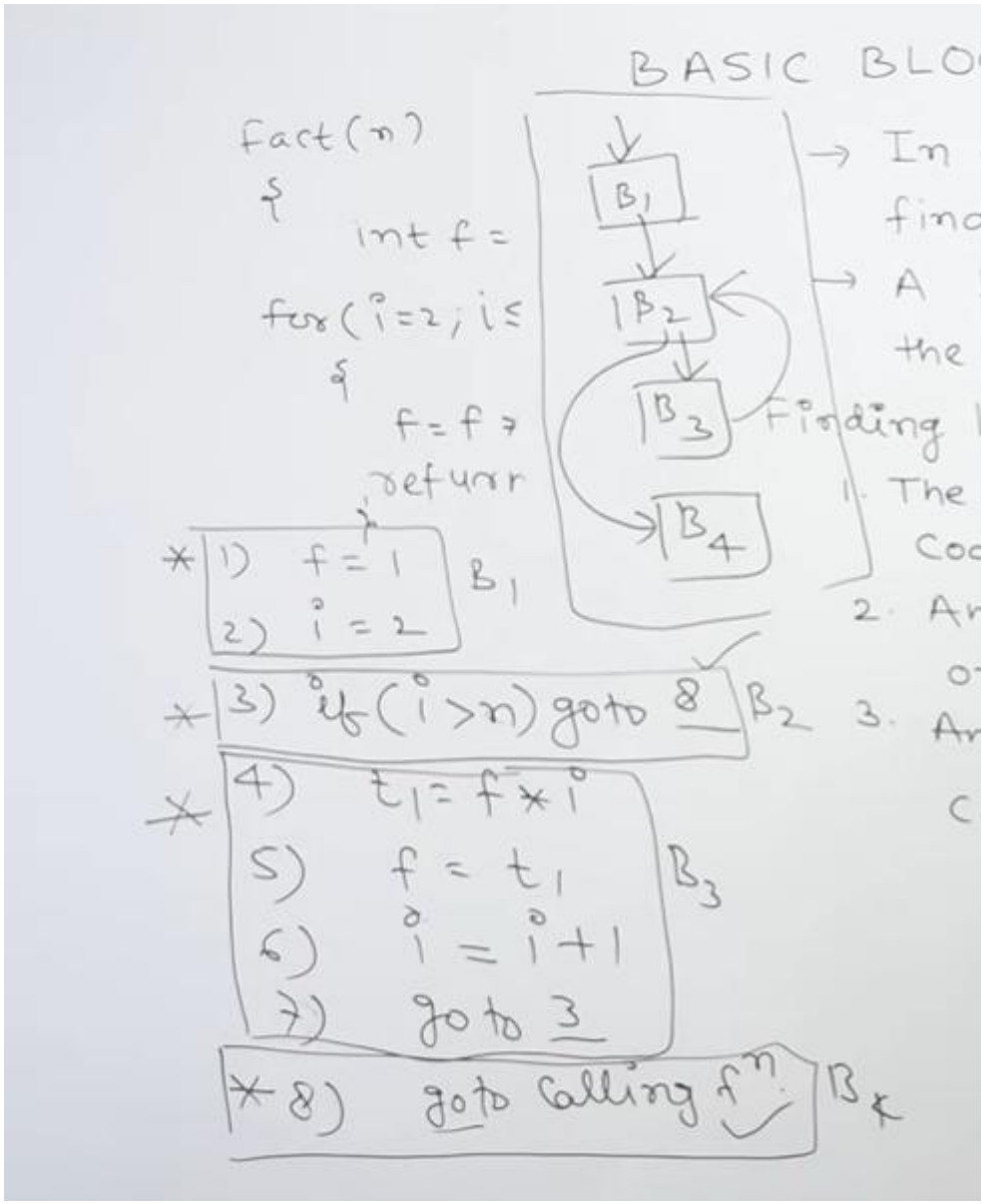


BASIC B

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

* 1) f = 1
2) i = 2
* 3) if(i > n) goto 8
* 4) $t_1 = f * i$
5) $f = t_1$
6) $i = i + 1$
7) goto 3
* 8) goto calling fn.

→
→
* Findi
1.
2
3

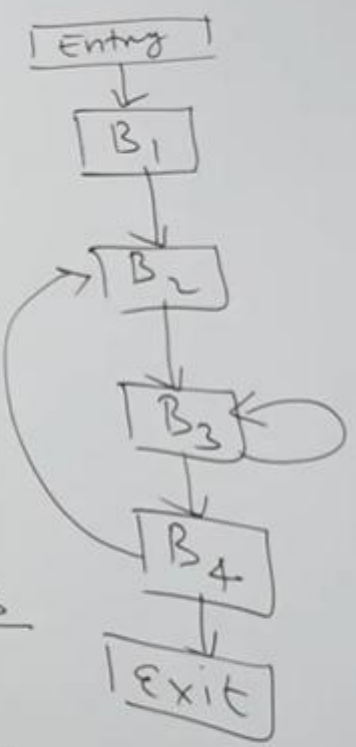

BASIC BLO
fact(n)
{
int f =
for(i=2; i<=
{
f = f *
return
}
1) f = 1
2) i = 2
B1
3) if(i > n) goto 8
B2
4) t1 = f * i
5) f = t1
6) i = i + 1
7) goto 3
B3
8) goto calling fn
Bx
B1
B2
B3
B4
Finding
In
find
A
the
The
Co
2. Ar
3. Ar

The number of nodes and edges in the control flow Graph constructed for the code respectively, are

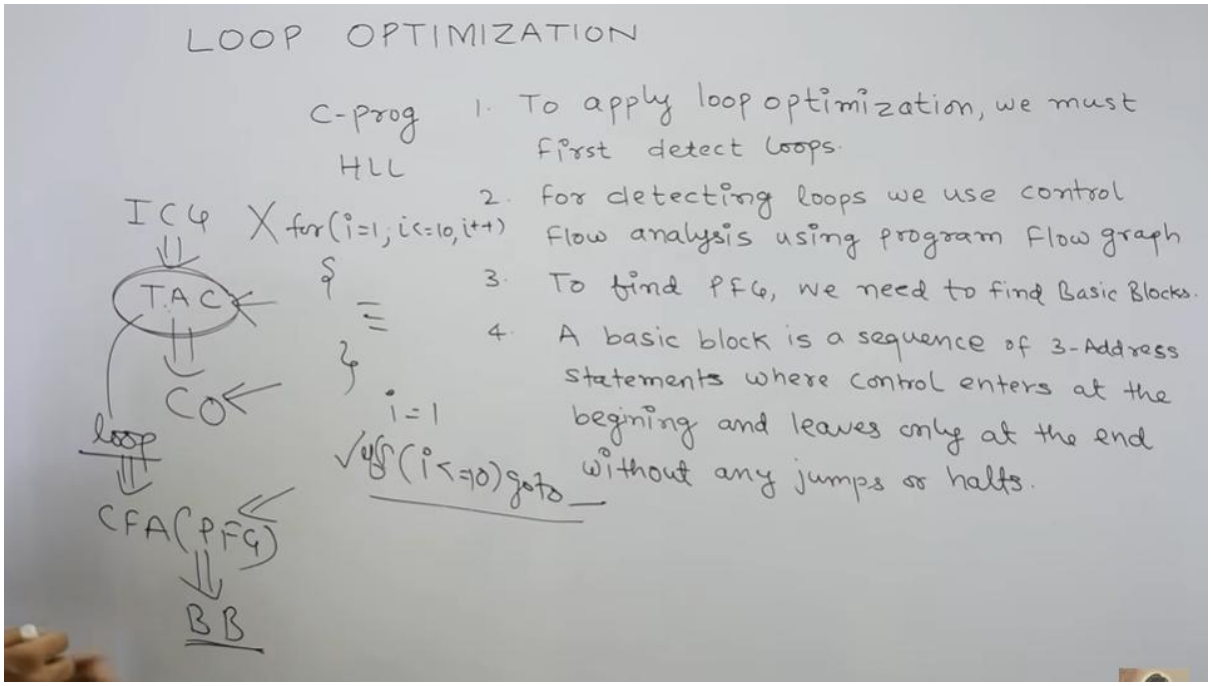
1) 5 and 7
2) 6 and 7
3) 5 and 5
4) 7 and 8

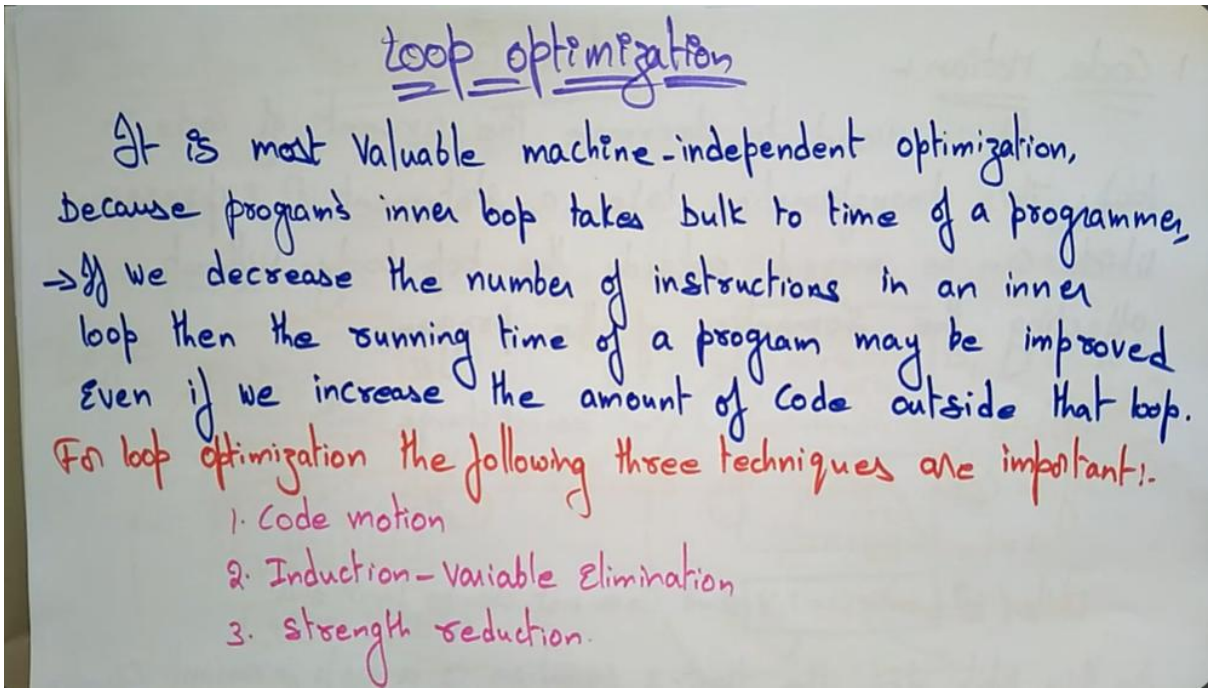
termediate code given below.

```
*  1.  i = 1                    ] B1
*  2.  j = 1                    ] B2
*  3.  t1 = 5 * i               ]
   4.  t2 = t1 + j              ]
   5.  t3 = 4 * t2              ]
B3 6.  t4 = t3                  ]
   7.  a[t4] = -1               ]
   8.  j = j + 1                ]
   9.  if (j<=5) goto 3         ]
*  10. i = i + 1                ]
B4 11. if (i<5) goto 2          ]
```



Loop Optimisation | Code motion & Strength reduction





a) Frequency Reduction (Code Motion)

b) Loo
Reducing the
are mac

A statement or expression which can be moved outside the loop body without affecting the semantic of the program.

ex

```
Ex   i=0                          i=0
     while(i<1000)          ⇒     t=a/b
     {                            while(i<1000)
       A=(a/b)+i                  {
       i++                          A=t+i
     }                              i++
                                  }
```

1. Code Motion :-

It is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

Eg:

```
While (i<=limit-2)   // stmt does not change limit
```

After code motion the result is as follows:

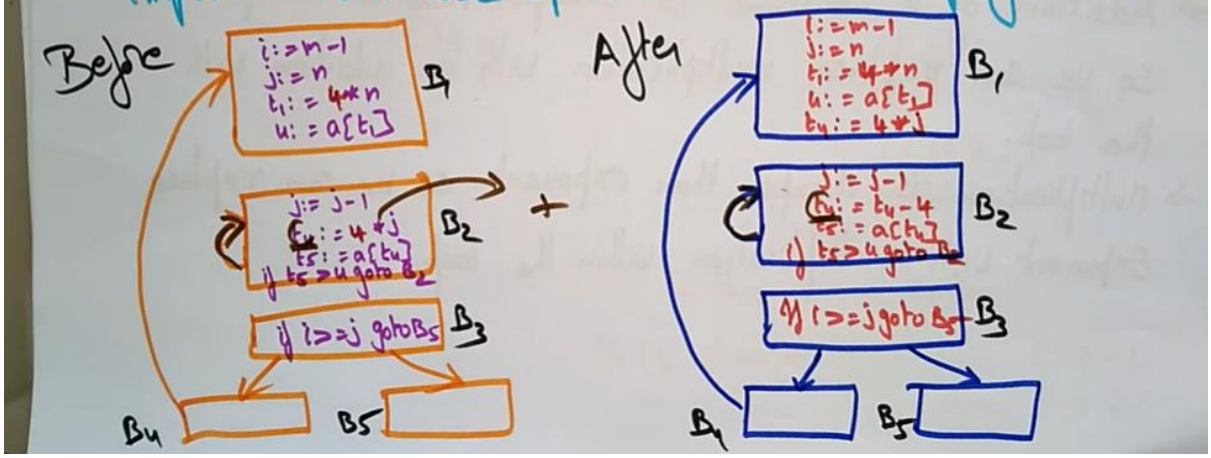
```
a=limit-2;
while(i<=a)          // stmt does not change limit of a
```

In this while stmt, the limit-2 equation is a loop invariant equatio

Induction variable elimination

2. Induction - Variable Elimination :-

- It is used to replace variable from inner loop.
- It can reduce the number of additions in a loop. It improves both code space and runtime performance.



b) Loop Unrooling

Reducing the no of times comparisons are made in the loop.

ex

```
for(i=0; i<10; i++) 10
{
    printf("Hi");
}
```

⇓opt

```
for(i=0; i<10; i=i+2) 2,2,2
{
    printf("Hi");
    printf("Hi");
}
```

```
for(i=0; i<5; i++)
{
    Pf("Hi");
    Pf("Hi");
}
```

c) Loop Jamming

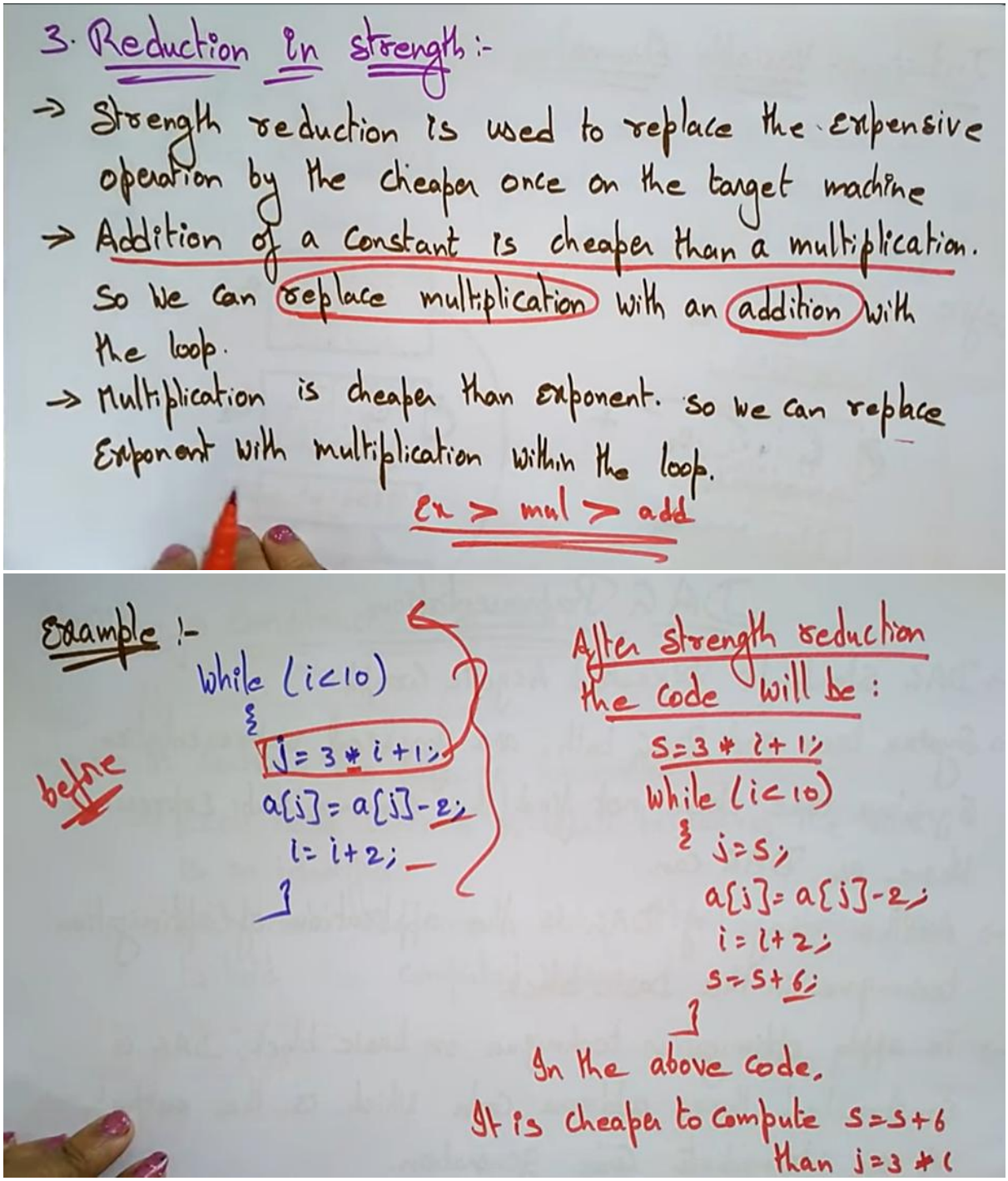
Combine the bodies of two loops.

Ex

```
for(i=0; i<5; i++)
{
    a = i+5
}
for(i=0; i<5; i++)
{
    b = i+10
}
```

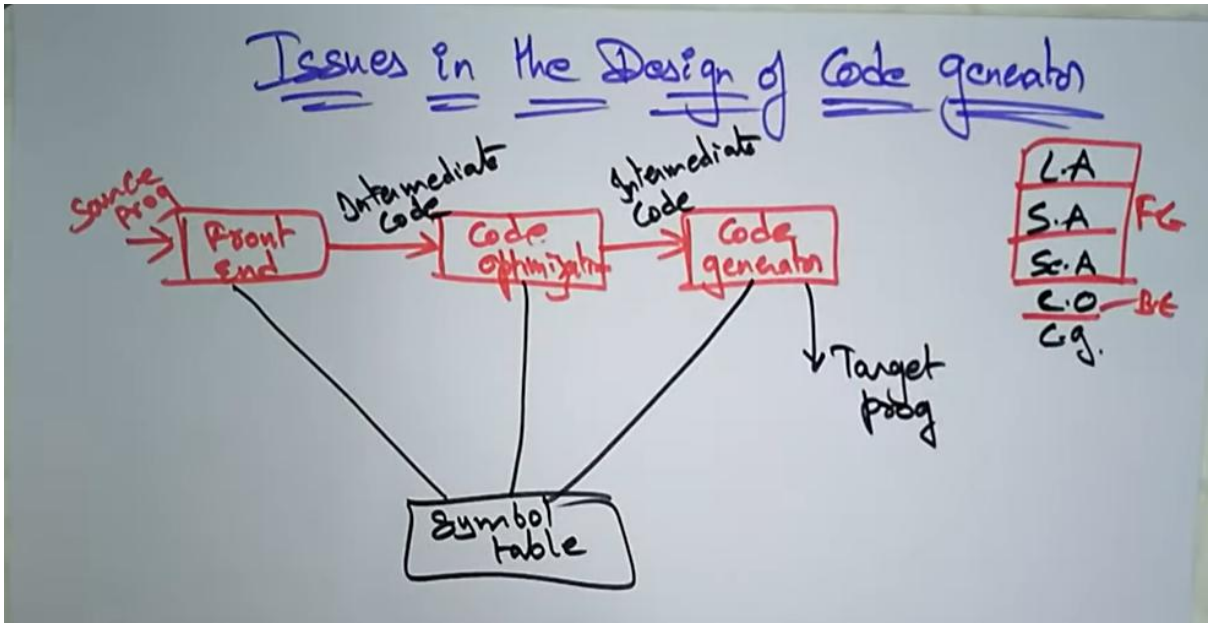
⇓opt

```
for(i=0; i<5; i++)
{
    a = i+5
    b = i+10
}
```

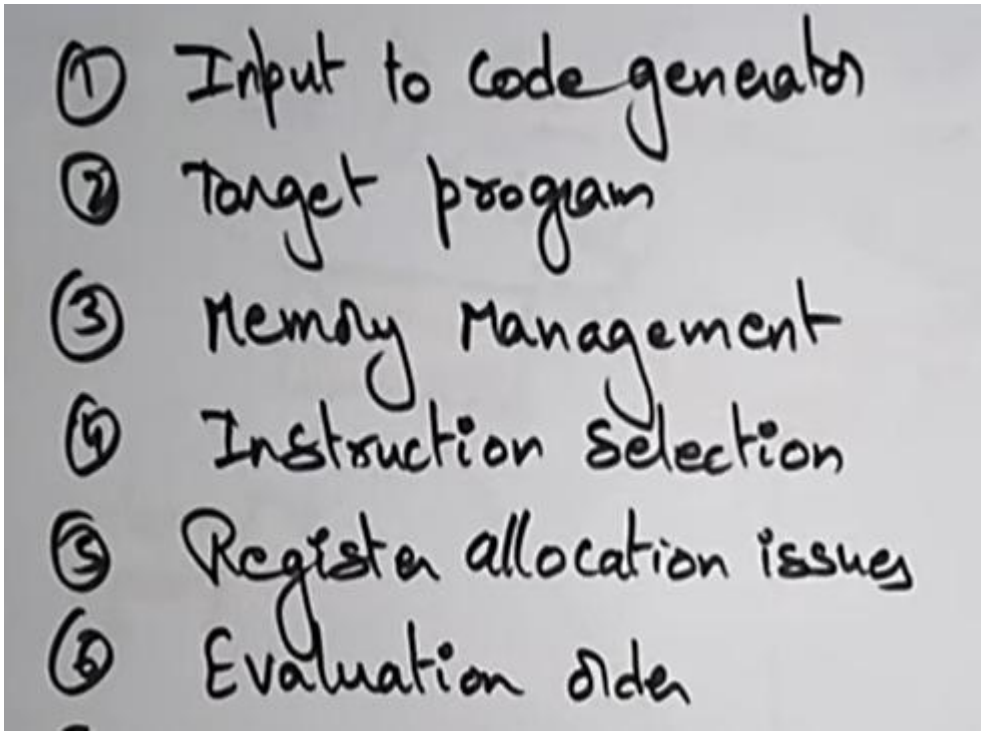


Code generator | Issues in design

Design Issues - javatpoint



6 Design Issues



① Input to code generator :- {Intermediate code}

Linear representation

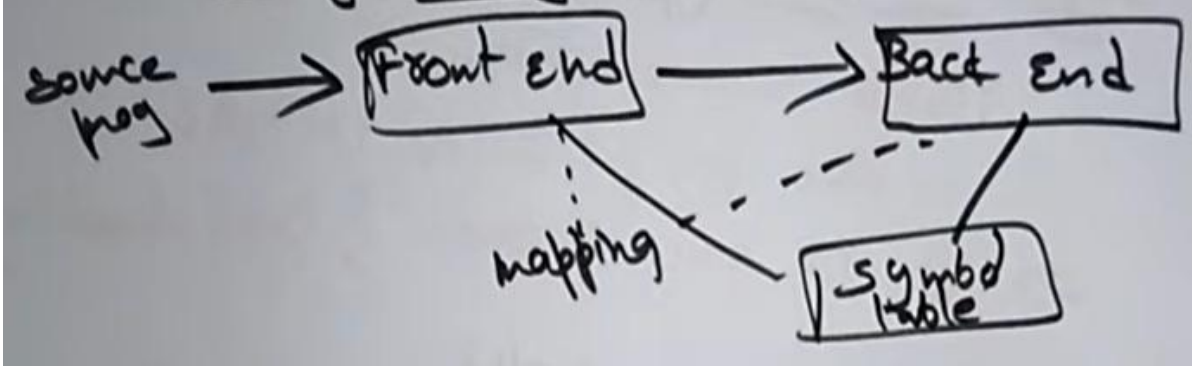
like postfix & TAC (or) DAG

⇒ i/p is free of errors {type checking}

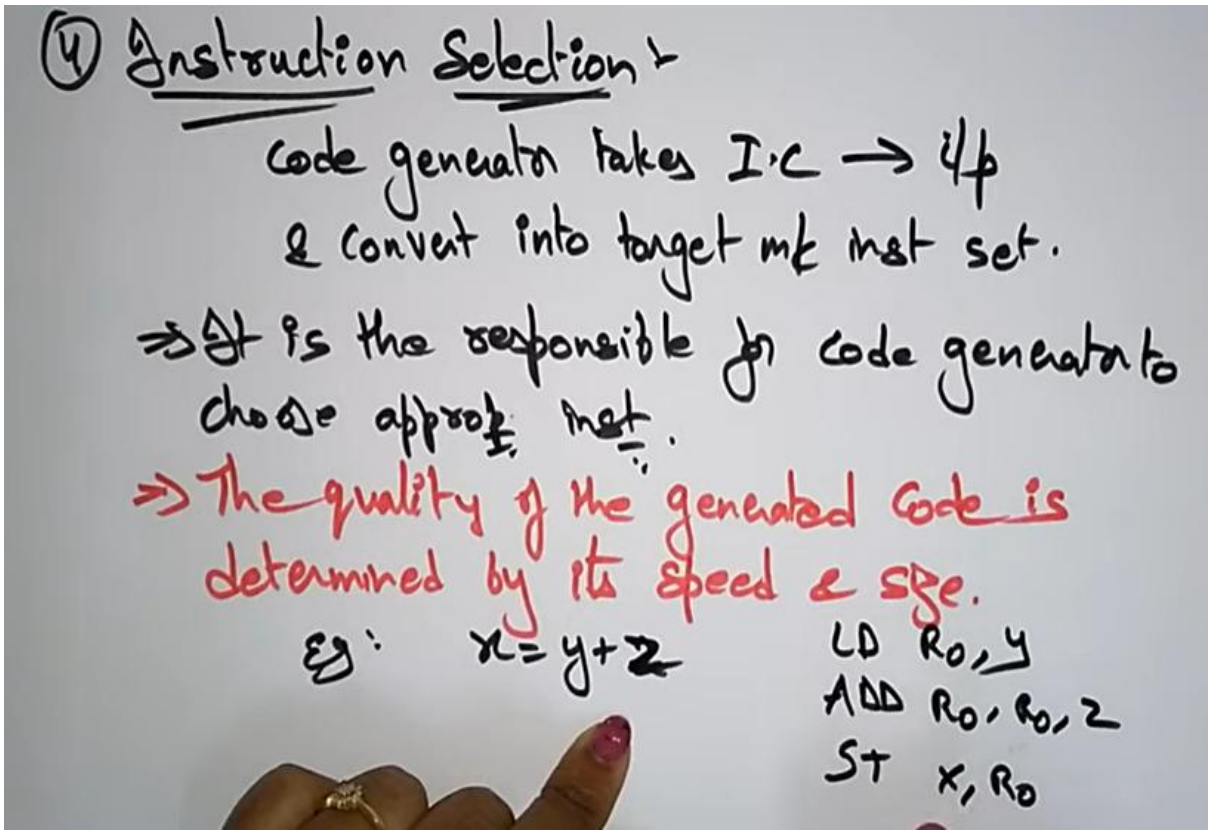
② Target program :- {o/p}

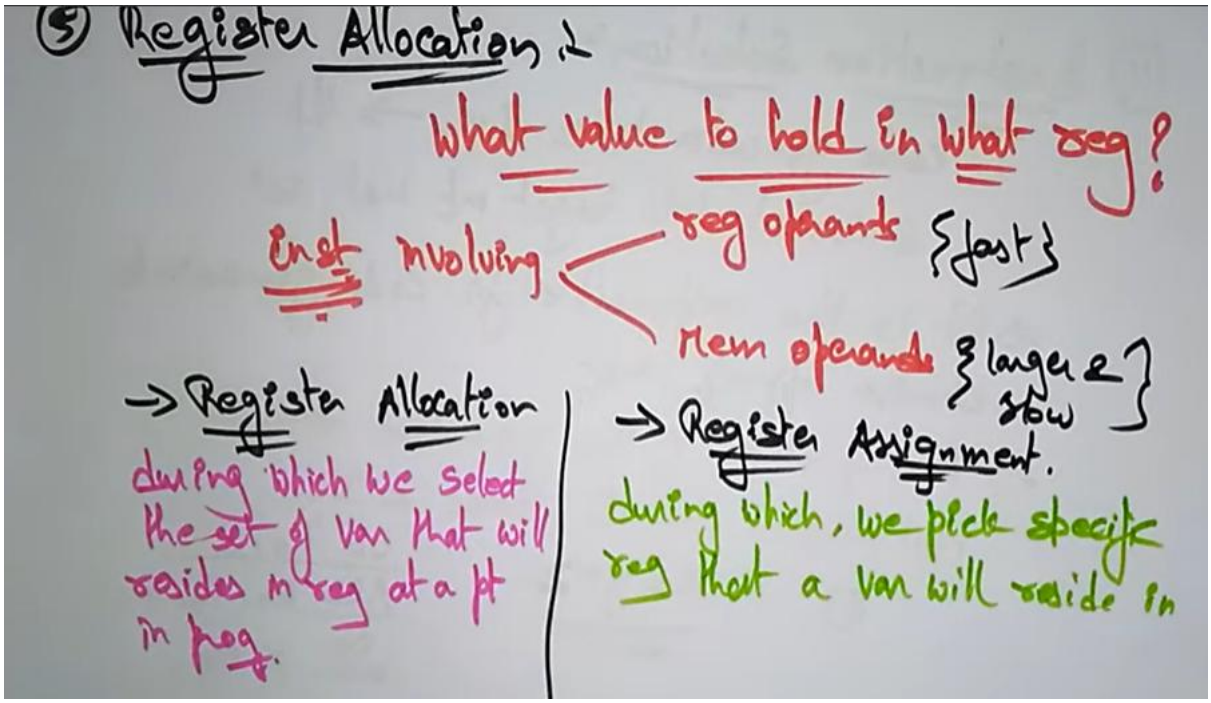
- Absolute m/c lang {Executable code}
- Relocatable m/c lang {of object files for linkers}
- Assembly lang.

③ Memory Management :-



Code generator | Issues in design | Part- 2/2





⑥ Evaluation Order :-

⇒ the order in which computations are performed can affect the efficiency of the target code.

⇒ When inst are independent their Evaluation order can be changed.

Eg 1- $(a+b)-(c+d)*e$

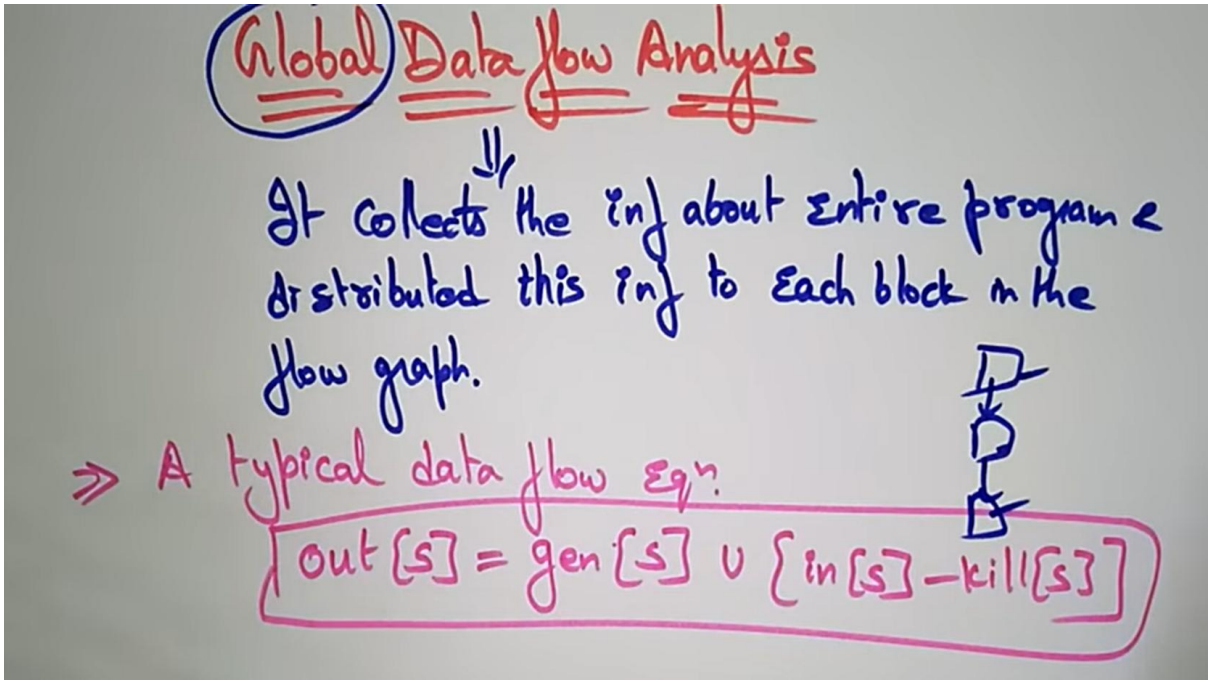
TAC ⇒
$t_1 = a+b$
$t_2 = c+d$
$t_3 = e * t_2$
$t_4 = t_1 - t_3$

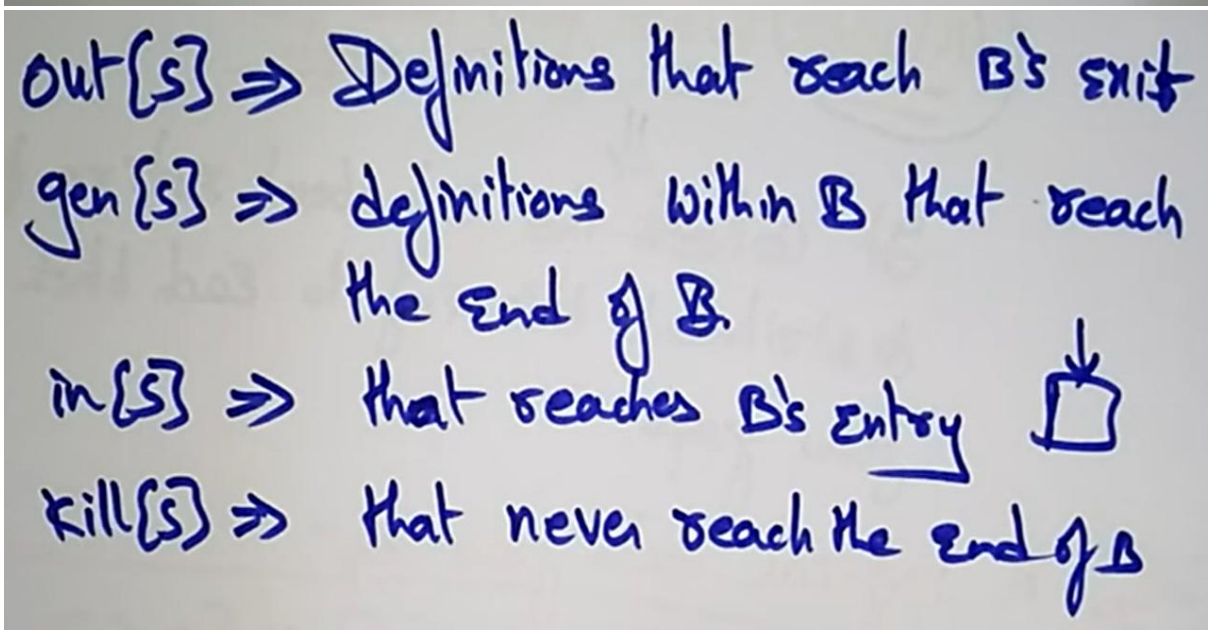
reorder
$t_2 = c+d$
$t_3 = e * t_2$
$t_1 := a+b$
$t_4 := t_1 - t_3$

```
MOV  a, R0      R0, a
ADD  b, R0      R0, b
MOV  R0, t1
ADD  d, R1
MOV  e, R0
MUL  R1, R0
MOV  t1, R1
SUB  R0, R1
MOV  R1, t4
```

```
MOV  c, R0   -
ADD  d, R0   -
MOV  e, R1   -
MUL  R0, R1  -
MOV  a, R0   -
ADD  b, R0   -
SUB  R1, R0  -
MOV  R0, t4  -
```

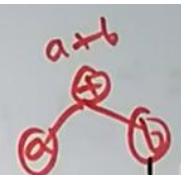
Global Dataflow analysis





Directed Acyclic Graph (DAG)

DAG Representation



→ DAG stands for Directed Acyclic Graph

→ Syntax tree and DAG both, are graphical representation, Syntax tree does not find the Common Sub Expressions where as DAG can.

→ Another usage of DAG is the application of optimization technique in the basic block.

→ To apply optimization technique on basic block, DAG is constructed three address code which is the output of an intermediate code generation.

Algorithm for construction of DAG :-

Input :- It contains a basic block

output :- It contains the following information

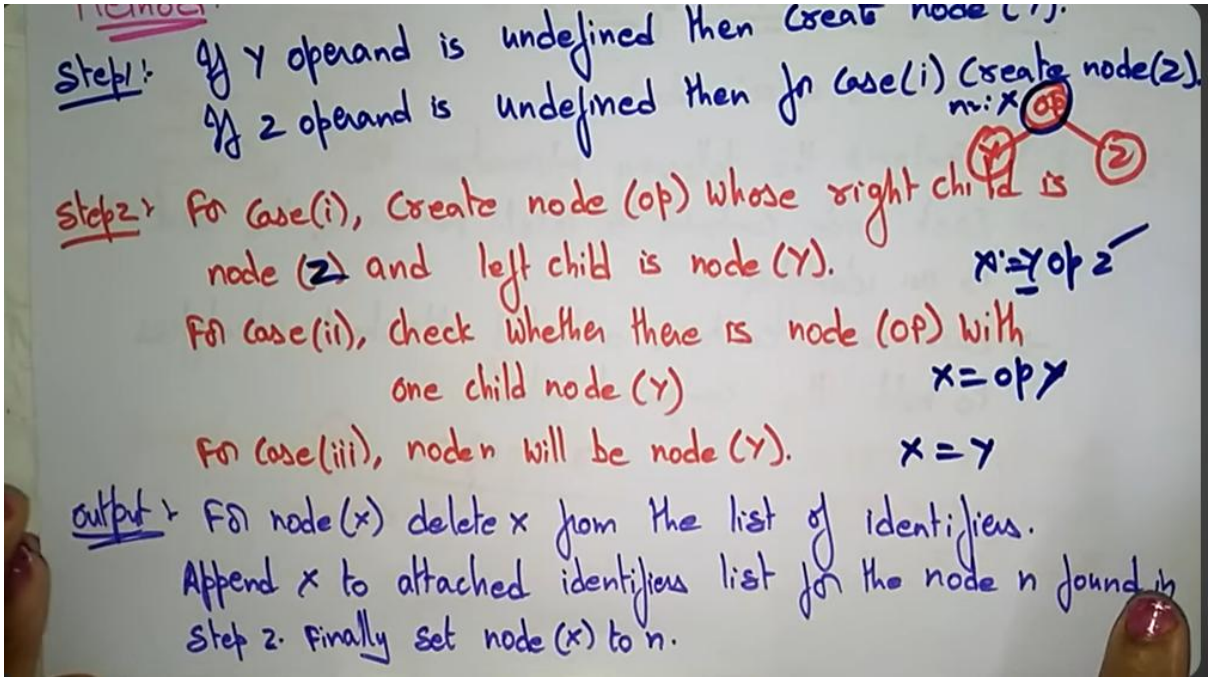
→ Each node contains a label. For leaves, the label is an identifier

→ Each node contains a list of attached identifiers to hold the computed values.

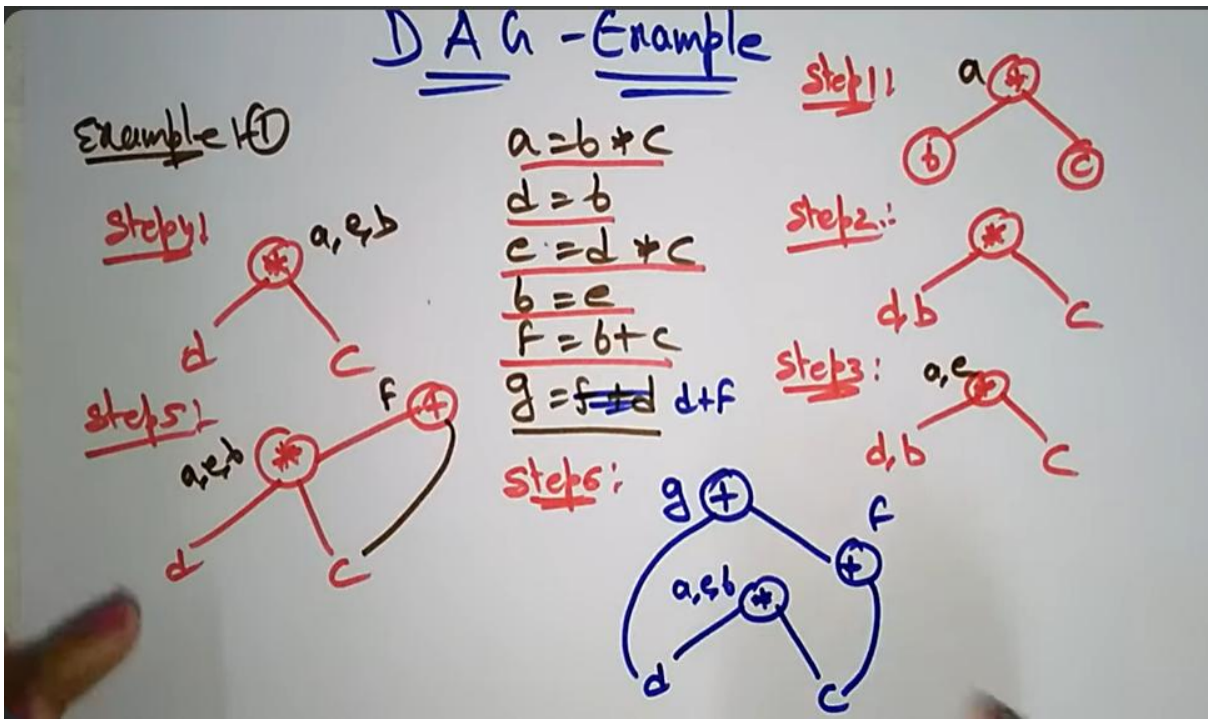
Case(i) x := Y op Z

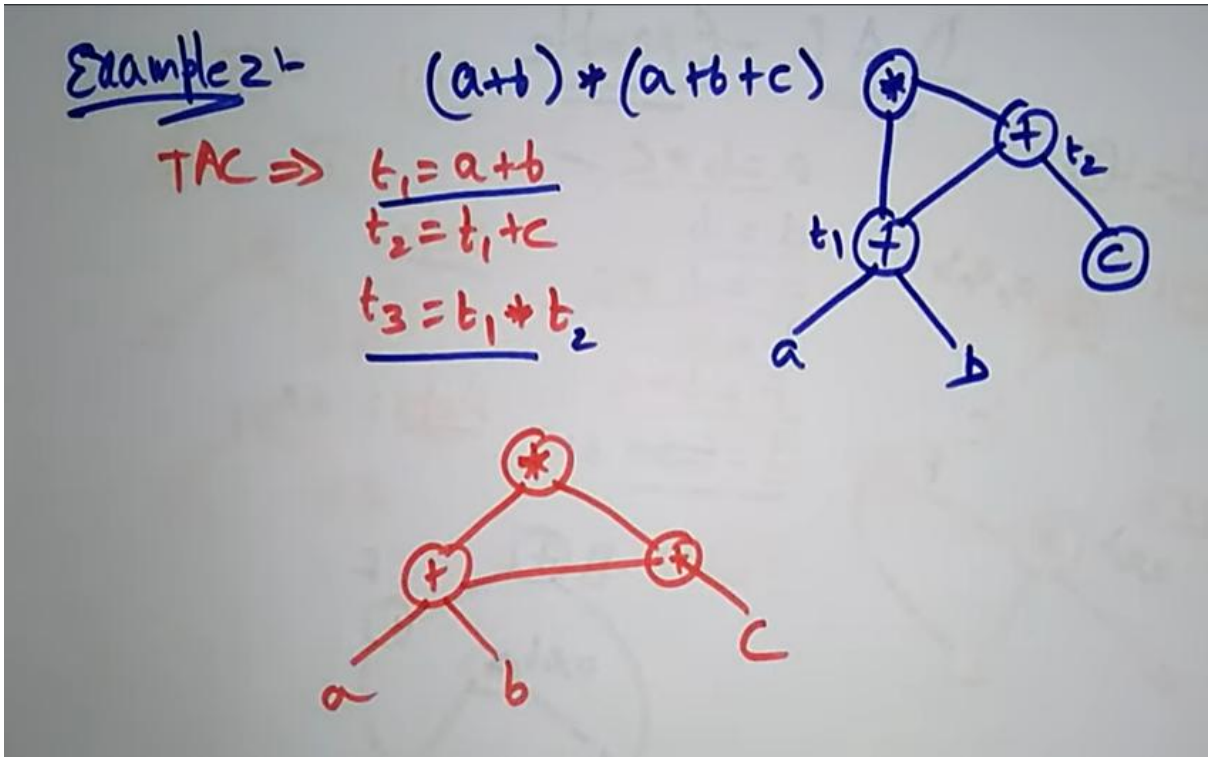
Case(ii) x := op Y

Case(iii) x := Y

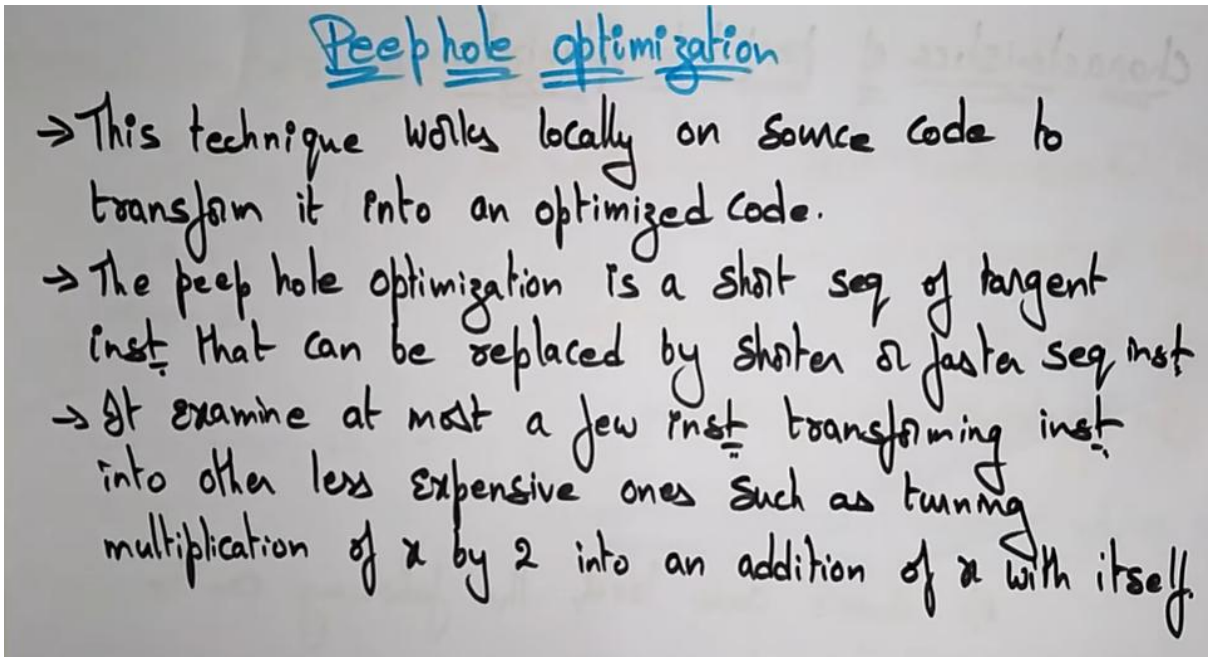


Directed Acyclic Graph (DAG) Examples





Peephole Optimisation



Characteristics of peep hole optimization :

1. Redundant inst elimination
2. Unreachable code
3. Flow of control optimization
4. Algebraic Simplifications

① ⇒ Redundant inst elimination :

At source code level, the following can be done by user.

①
```
int add_ten(int x)
{
  int y, z;
  y = 10;
  z = x + y;
  return z;
}
```

②
```
int add_ten(int x)
{
  int y;
  y = 10;
  y = x + y;
  return y;
}
```

③
```
int add_ten(int x)
{
  int y = 10;
  return x + y;
}
```

④
```
int add_ten(int x)
{
  return x + 10;
}
```

a) Redundant LOAD/STORE elimination

$a = b + c$

```
LOAD  R0, b
ADD   R0, c
STORE a , R0
```

b)

$R_0 = b + c$

1) LOAD R0, b
2) ADD R0, C
3) STORE a, R0
4) LOAD R0, a
5) ADD R0, e
6) STORE d, R0

a=b+c → R0

✓ $a = b + c$

$d = a + e$

$i = 4$

c) Use of M/C idioms

$i = i + 1$

```
LOAD R0, i
ADD R0, #1
STORE i, R0
INC i
```

b) Flow of Control Optimization

1) Avoid Jumps on Jumps

```
L1 : jump L2
     ≡≡≡
L2 : jump L3
     ≡≡≡
L3 : jump L4
     ≡≡≡
L4 : x = a+b*c
```

2) Eliminate Dead Code

```
int i = 0;
if (i == 1)
{
printf ("Dead code");
}
```

② Unreachable Code :-

It is a part of program code that is never accessed because of program constructs.

⇒ Programmers may have accidently written a piece of code that can never be reached.

eg:
```
void add_ten (int x)
{
    return x+10;
    printf ("Value of x is %d," x);
}
```

In this stmt p/f stmt will never executed as program control returns back before it execute, hence p/f can be removed

③ Flow of Control optimization

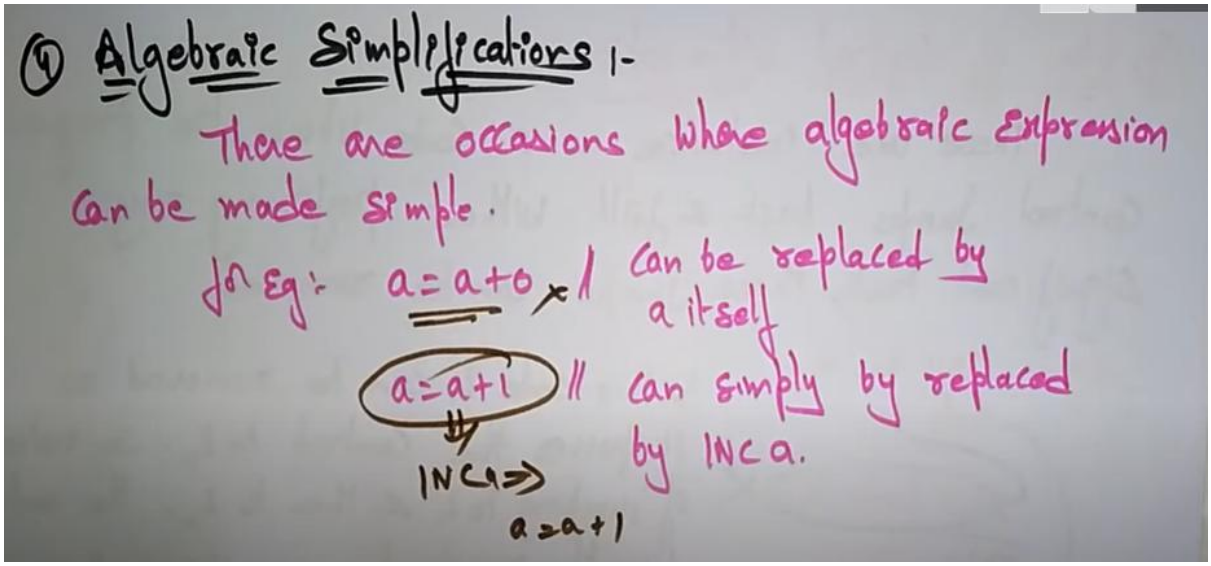
These are instances in a code where the program control jumps back & forth without performing any significant task, these jumps can be removed.

eg:
```
MOV R1, R2
Goto L1
- -
- -
L1: Goto L2
L2: INC R1
- -
- -
```

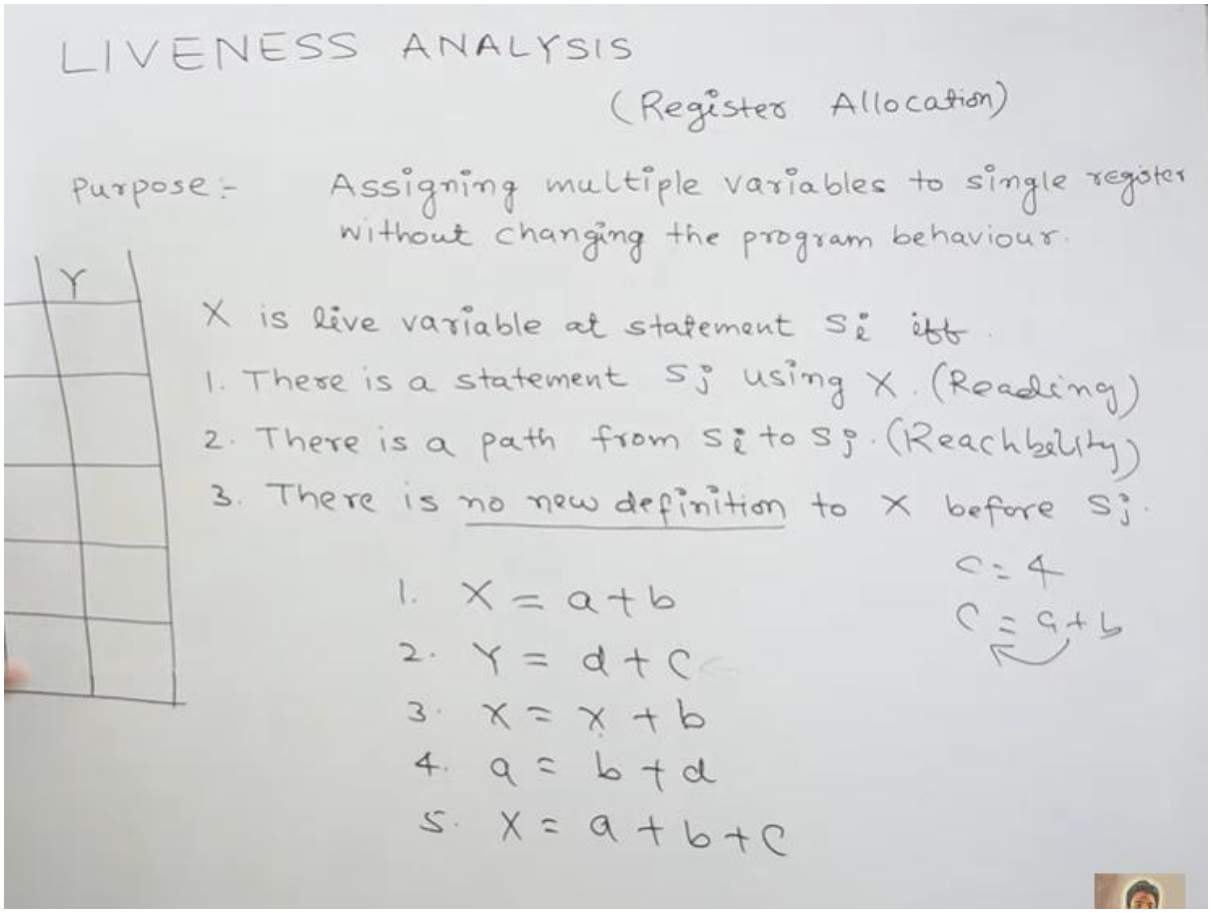
In this code L1 can be removed as it passes the control to L2. So, instead of jumping to L1 & then to L2. the control directly reach L2.

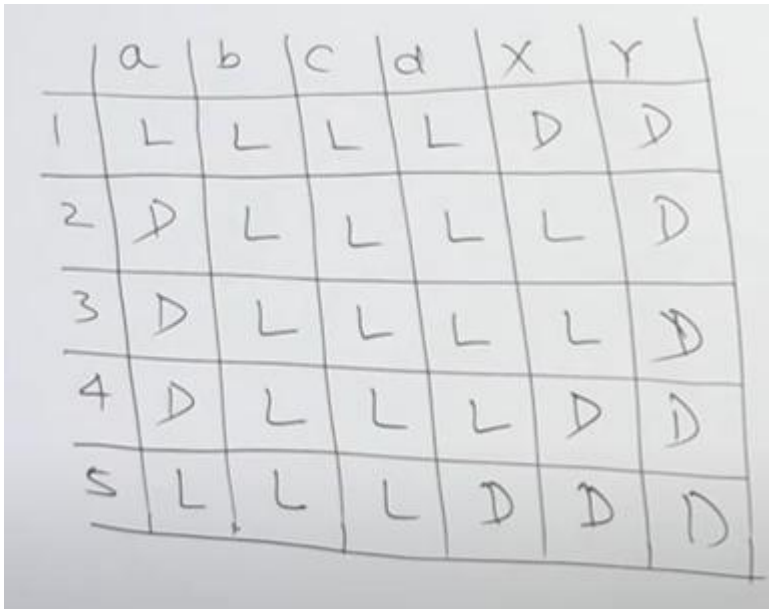
```
MOV R1, R2
Goto L2
- - - -
```

```
MOV R1, R2
Goto L2
- - - -
- - - -
L2: INC R1
```



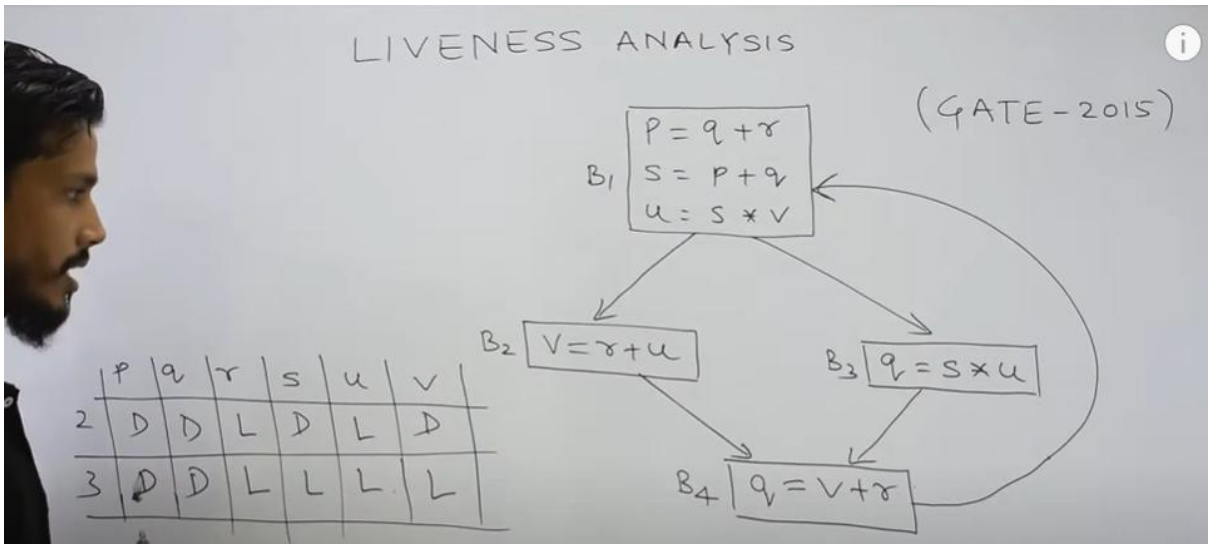
Liveness Analysis





	a	b	c	d	X	Y
1	L	L	L	L	D	D
2	D	L	L	L	L	D
3	D	L	L	L	L	D
4	D	L	L	L	D	D
5	L	L	L	D	D	D

Question : Find the common live variables in the code.



	p	q	r	s	u	v
2	D	D	L	D	L	D
3	D	D	L	L	L	L

Ans: (r,u)

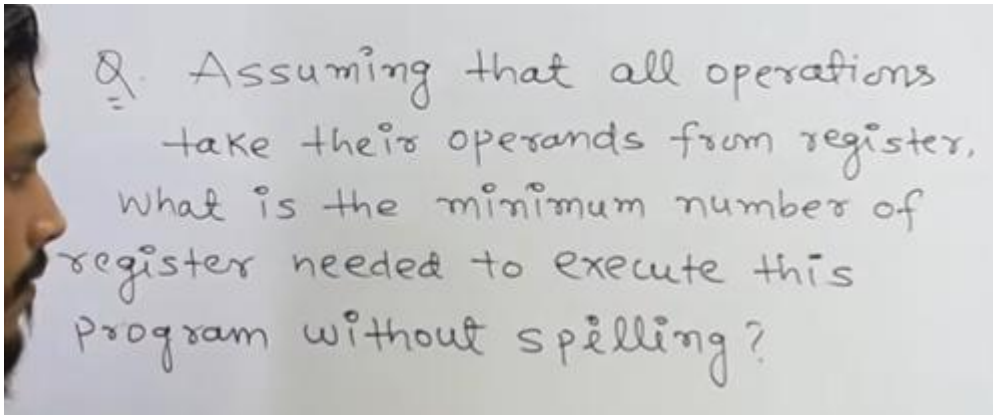
What is Spilling?

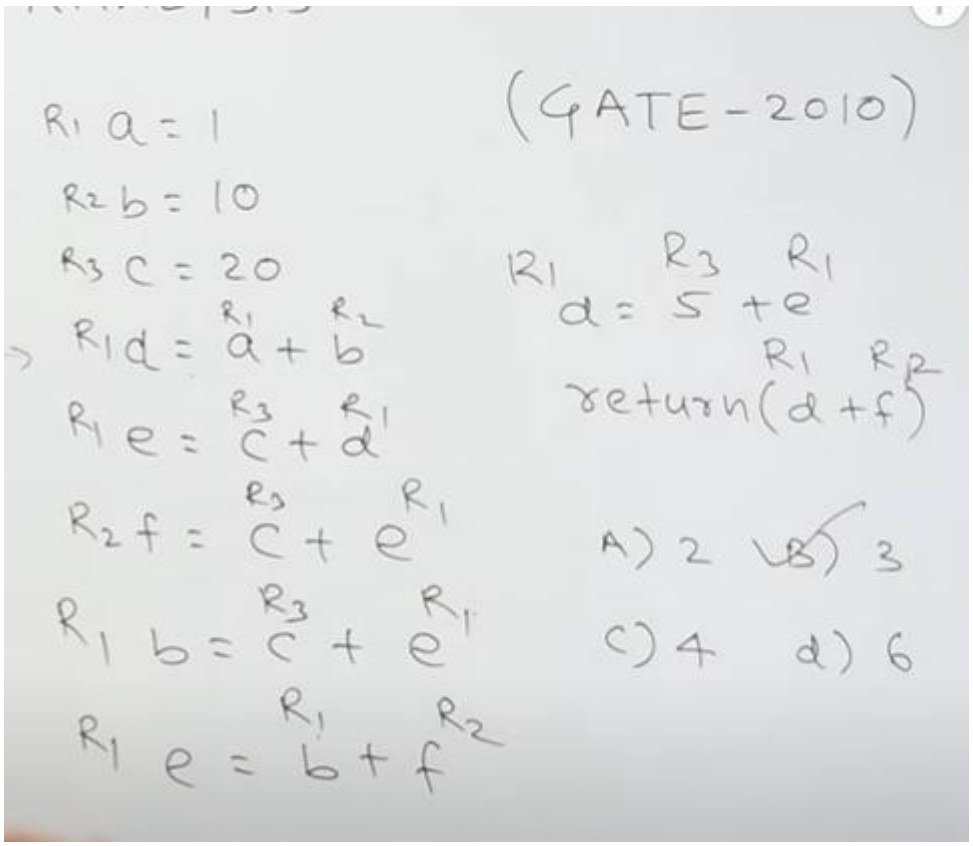
Formally speaking, spilling is a technique in which, a variable is moved out from a register space to the main memory(the RAM) to make space for other variables, which are to be used in the program currently under execution.

Why Spilling?

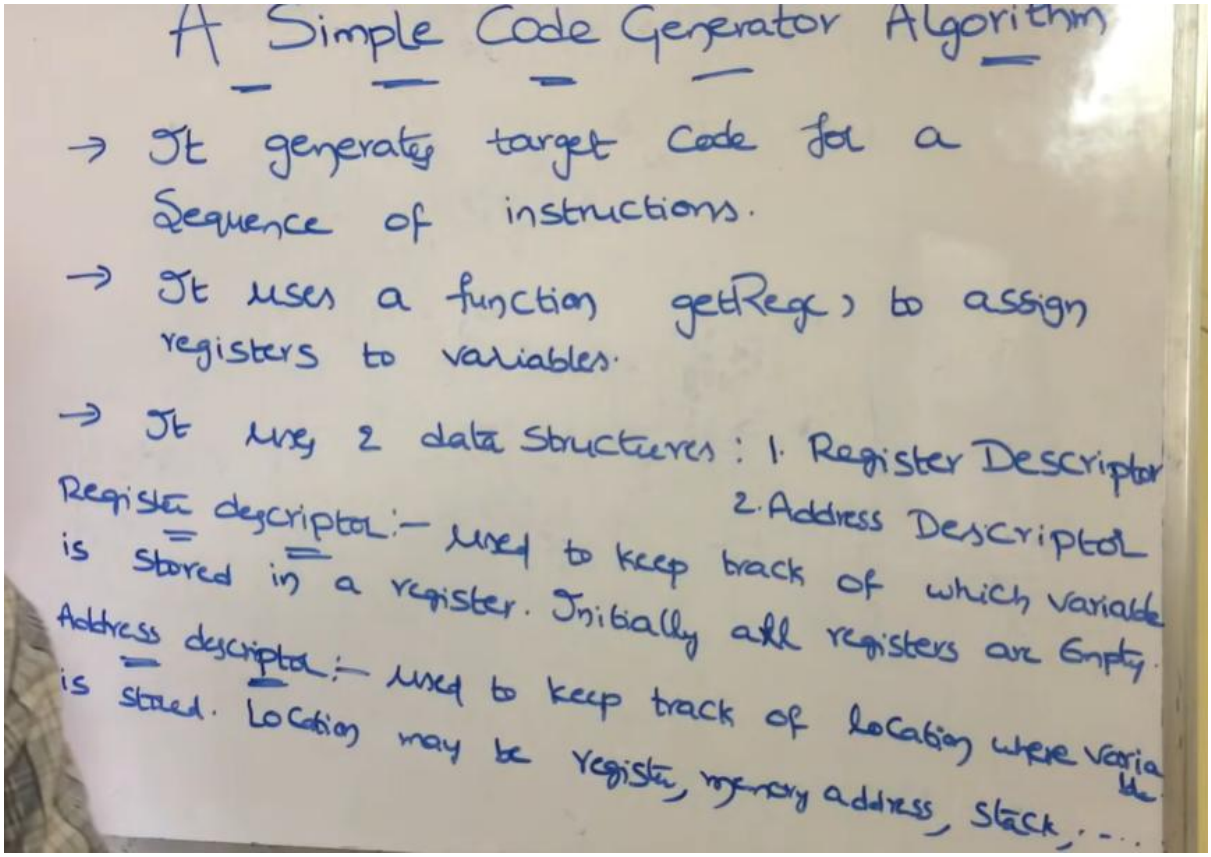
Spilling is done in order to satisfy the below conditions –

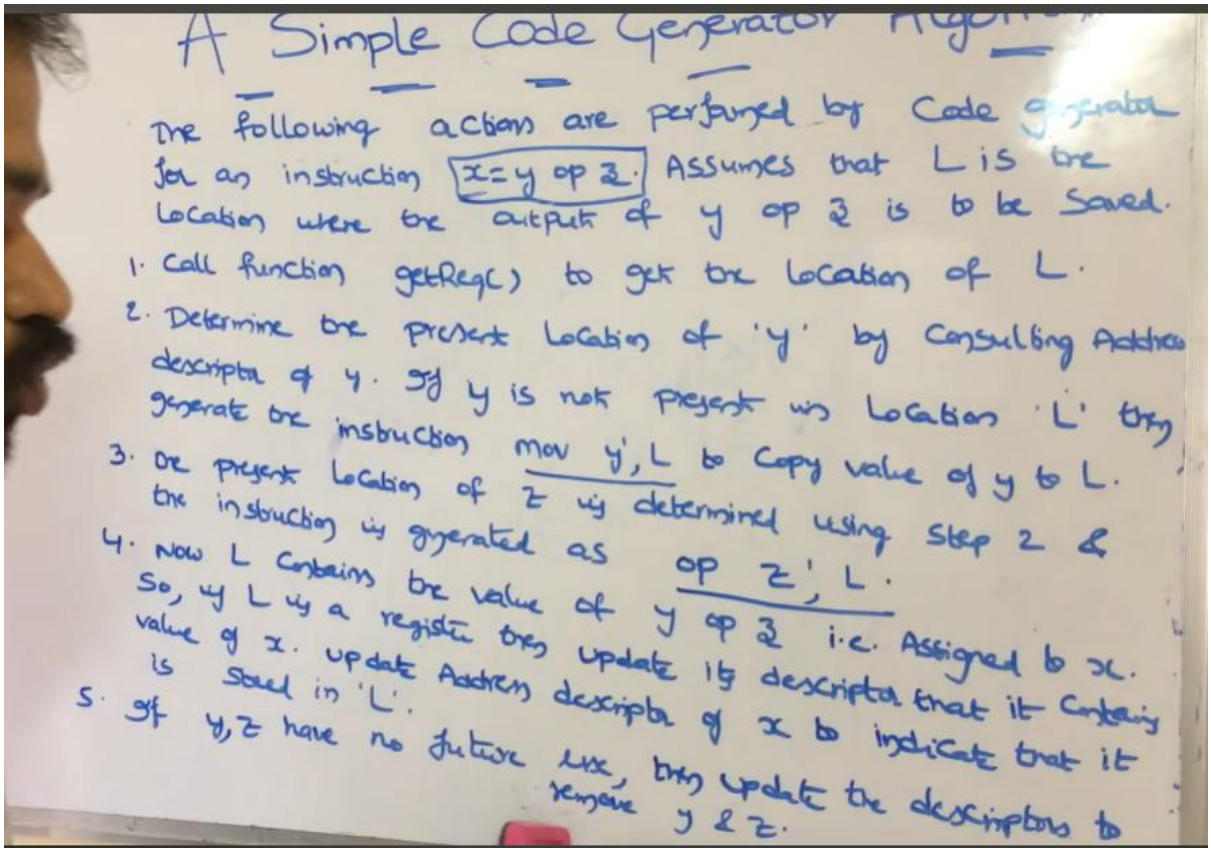
1. Avoid Register Allocation Failures –
   This happens, when there is a limited number of registers to hold the live variables, hence they are swapped between register and memory.
2. Parameter Passing in Function –
   Parameters are arguments that are passed to a function. When they cannot be passed into registers, they are stored in memory locations.
3. Dynamic Allocations –
   A programmer may need to add names, which are generated during the run-time. As registers are statically named, the compiler relies on the memory for dynamic name generation.

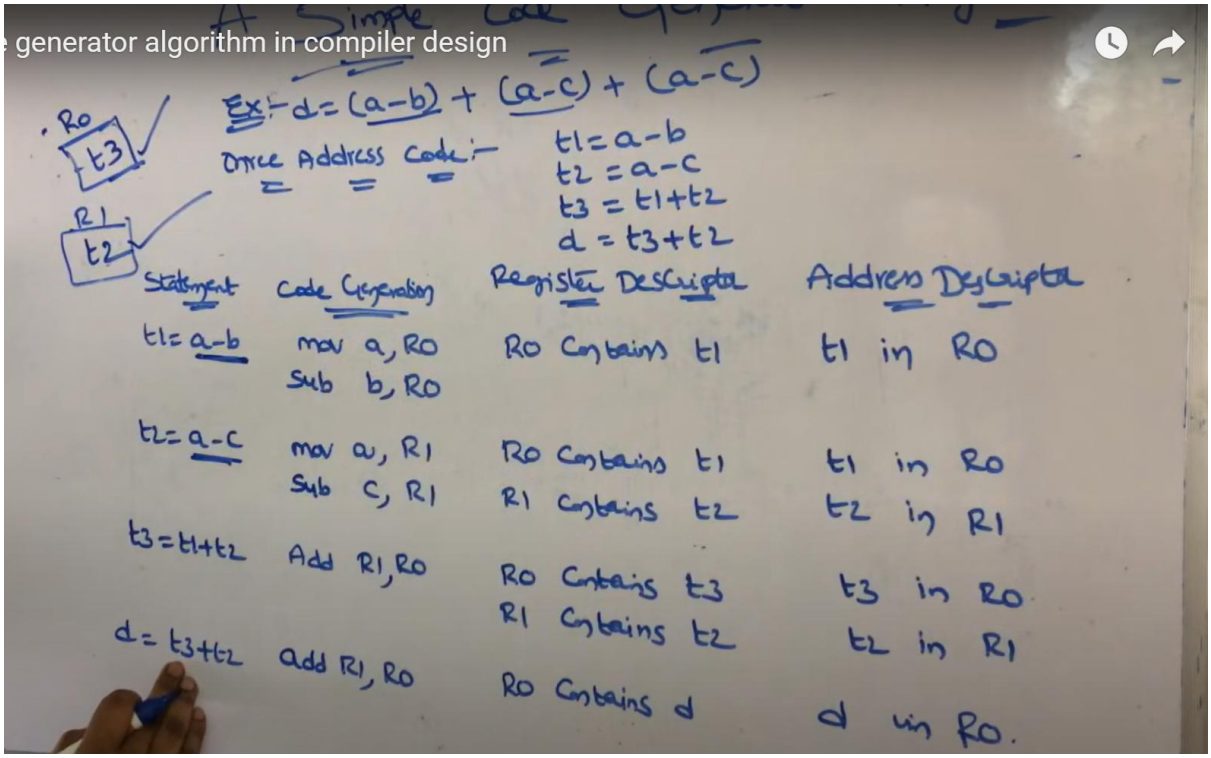

Q. Assuming that all operations take their operands from register, what is the minimum number of register needed to execute this program without spilling?


(GATE-2010)
R1 a = 1
R2 b = 10
R3 c = 20
R1 d = a + b
R1 e = c + d
R2 f = c + e
R1 b = c + e
R1 e = b + f
R1 R3 R1
d = 5 + e
R1 R2
return (d + f)
A) 2
B) 3
C) 4
d) 6

Code Generation







A simple code Generator

- Generates target code for a sequence of 3-address statements.
- For each operator in a statement, there is a corresponding target language operator.

target

Register and Address Descriptors

1. Register Descriptor :- keeps track of what is currently in each register (R0, R1)
   - Initially all registers are empty.

2. Address Descriptor :- keeps track of the location where the current value of the name can be found
   - Location may be register, a stack location or memory address

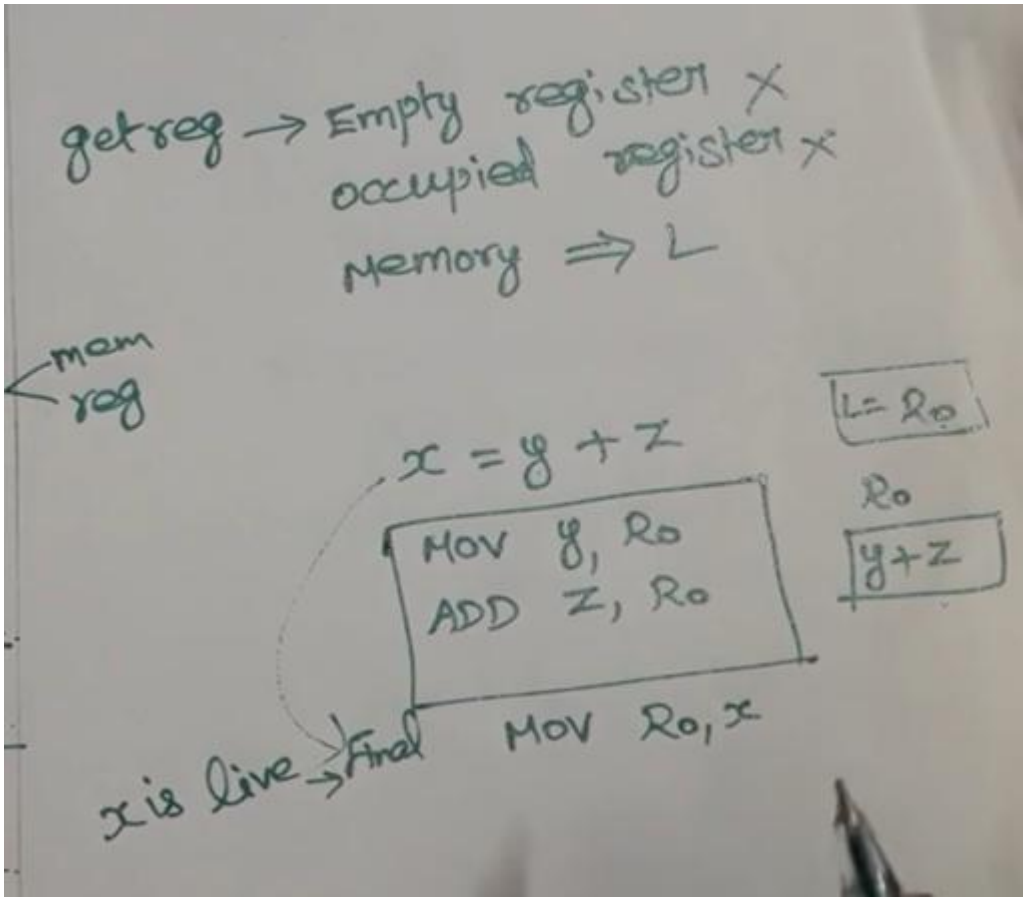
a

A code Generation Algorithm

For each Three address statement of the form $x = y$ op z,

1. Invoke a function getreg to determine the location L, where result of y op z should be stored.
2. consult address descriptor for y to determine y', the current location of y. If y is not already in L, generate MOV y', L.
3. Generate the instruction op z', L. update address descriptor of x to indicate that x is in L. If L is a register, update its descriptor to indicate that it contains the value of x.

4. If y and z have no next uses and not live on exit, update the descriptors to remove y & z.



Example:

$d = (a-b) + (a-c) + (a-c)$

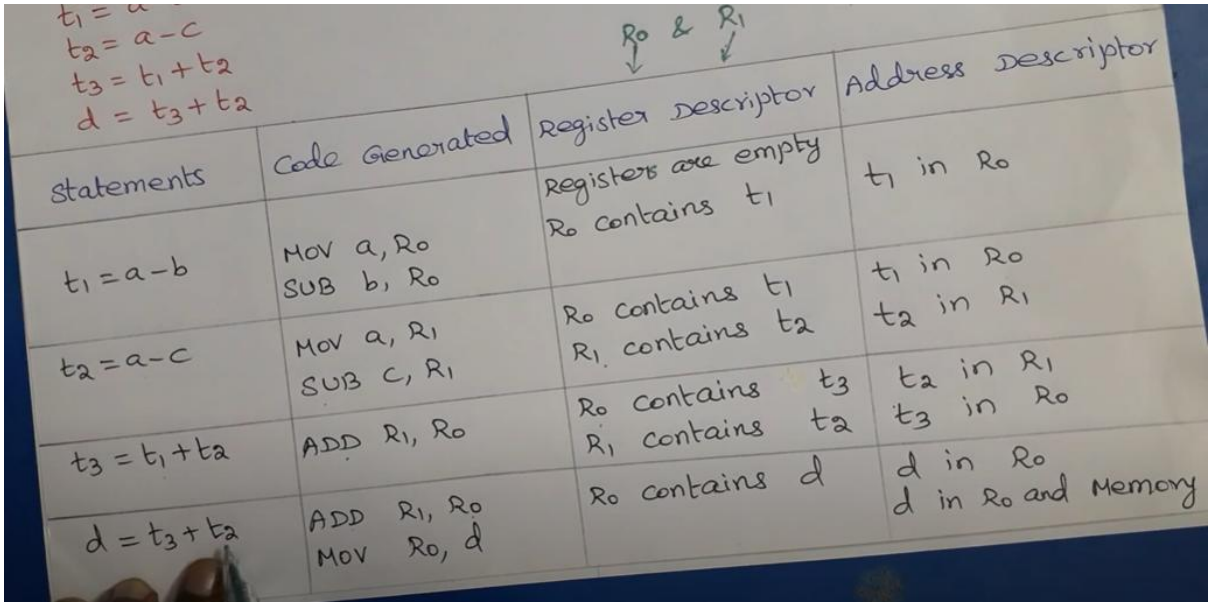
Three address code sequence

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$



Code Generator: Introduction

- **Code generator** converts the intermediate representation of source code into a form that can be readily executed by the machine.
- A code generator is expected to generate a correct code.
- Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

Code Generator: Issues

1. Input to the Code Generator
2. Target Programs
3. Memory Management
4. Instruction Selection
5. Register Allocation
6. Choice of Evaluation Order
7. Approaches to Code Generation

1. Input to the Code Generator

- The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation.
- Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's etc.
- Assume that they are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target Programs

- Target program is the output of the code generator.
- The output may be absolute machine language, relocatable machine language, assembly language.
- Absolute Machine Language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- Relocatable Machine Language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.
- Assembly Language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

3. Memory Management

- Mapping the names in the source program to addresses of data objects is done by the front end and the code generator.
- A name in the three address statement refers to the symbol table entry for name.
- Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction Selection

- Selecting best instructions will improve the efficiency of the program.
- It includes the instructions that should be complete and uniform.
- Instruction speeds and machine idioms also plays a major role when efficiency is considered.
- But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

5. Register Allocation

- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important.
- The use of registers are subdivided into two sub-problems:
  - During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
  - During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

6. Choice of Evaluation Order

- The code generator decides the order in which the instruction will be executed.
- The order of computations affects the efficiency of the target code.
- Among many computational orders, some will require only fewer registers to hold the intermediate results.
- However, picking the best order in general case is a difficult NP-complete problem.

7. Approaches to Code Generation

- Code generator must always generate the correct code.
- It is essential because of the number of special cases that a code generator might face.
- Some of the design goals of code generator are:
  - Correct
  - Easily maintainable
  - Testable
  - Maintainable

SIC – Simplified Instructional Computer Architecture

SIC Machine Architecture

- The SIC machine architecture depends on the following features:
  - Memory
  - Registers
  - Data Formats
  - Instruction Formats
  - Addressing Modes
  - Instruction Set
  - Input and Output

- Memory
  - Memory consists of 8-bit bytes
  - Any 3 consecutive bytes form a word (24 bits)
  - Total of 32768 (2^{15}) bytes in the computer memory

SIC Machine Architecture

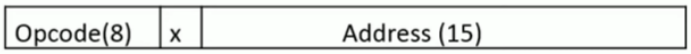
- Registers
  - Five 24-bits registers. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; contains the return address whenever control transferred to subroutine
PC	8	Program counter; contains the address of the next instruction to be executed.
SW	9	Status word, including Condition code such as <,≤,>,≥,==.

- Data Formats
  - Integers are stored as 24-bit binary number
  - 2's complement representation for negative values
  - Characters are stored using 8-bit ASCII codes
  - No floating-point hardware on the standard version of SIC

SIC Machine Architecture

- Instruction Formats
  - All machine instructions on the standard version of SIC have the 24-bit format as shown below



- Addressing Modes
  - There are two addressing modes available, which are as shown in the below table. Parentheses are used to indicate the contents of a register or a memory location.

Mode	Indication	Target address calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x)

- Instruction Set
  - Load and store registers - LDA, LDX, STA, STX, etc.
  - Integer arithmetic operations - ADD, SUB, MUL, DIV
    - All arithmetic operations involve register A and a word in memory, with the result being left in A
  - COMP – Comparison instruction
  - Conditional jump instructions - JLT, JEQ, JGT
  - Subroutine linkage - JSUB, RSUB
  - I/O (transferring 1 byte at a time to/from the rightmost 8 bits of register A)
    - Test Device instruction (TD)
    - Read Data (RD)
    - Write Data (WD)

- Memory
  - Memory consists of 8-bit bytes
  - Any 3 consecutive bytes form a word (24 bits)
  - Total of 1 Mb (2^{20}) bytes in the computer memory

SIC/XE Machine Architecture

- Registers
  - There are nine registers; each register is 24 bits in length except floating point register.
  - Their mnemonic, number and uses are shown in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; contains the return address whenever control transferred to subroutine
B	3	Base register; used for addressing
S	4	General working register-no special use
T	5	General working register-no special use
F	6	Floating point accumulator (48 bits)
PC	8	Program counter; contains the address of the next instruction to be executed.
SW	9	Status word, including Condition code such as <,≤,>,≥,==.

SIC/XE Machine Architecture

- Data Formats
  - Integers are stored as 24-bit binary number
  - 2's complement representation for negative values
  - Characters are stored using 8-bit ASCII codes
  - Support 48 bit floating-point numbers

0 - +ve
1 - -ve

1	11	36
s	exponent	fraction

  - There is a 48-bit floating-point data type, $F*2^{(e-1024)}$.

- Instruction Formats
- Format 1 (1 byte –

8
opcode

Example: RSUB

- Format 2 (2 byte) -

8	4	4
op	r1	r2

Example: ADDR S, T

- Format 3 (3 byte) -

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	Displacement

  - Example: LDA #3

- Format 4 (4 byte) -

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	Address

  - Example: +JSUB RDREC

- Addressing Modes and Flag Bits
  - Base relative (n=1, i=1, b=1, p=0)
  - Program-counter relative (n=1, i=1, b=0, p=1)
  - Direct (n=1, i=1, b=0, p=0)
  - Immediate (n=0, i=1, x=0)
  - Indirect (n=1, i=0, x=0)
  - Indexing (both n & i = 0 or 1, x=1)
  - Extended (e=1 for format 4, e=0 for format 3)

Instruction Set

- Load and store registers - LDA, LDX, STA, STX, LDB, STB etc.
- Integer arithmetic operations - ADD, SUB, MUL, DIV
- Floating-point arithmetic operations: ADDF, SUBF, MULF, DIVF
- COMP – Comparison instruction
- Conditional jump instructions - JLT, JEQ, JGT
- Subroutine linkage - JSUB, RSUB
- Register move instruction: RMO
- Register-to-register arithmetic operations: ADDR, SUBR, MULR, DIVR
- I/O (transferring 1 byte at a time to/from the rightmost 8 bits of register A)
  - Test Device instruction (TD)
  - Read Data (RD)
  - Write Data (WD)

- Base Relative Addressing Mode

	n	i	x	b	p	e	
opcode	1	1		1	0		disp

n=1, i=1, b=1, p=0, TA = (B) + disp (0≤disp ≤4095)

- Program-Counter Relative Addressing Mode

	n	i	x	b	p	e	
opcode	1	1		0	1		disp

n=1, i=1, b=0, p=1, TA= (PC) + disp (-2048≤disp ≤2047)

- Direct Addressing Mode

	n	i	x	b	p	e	
opcode	1	1		0	0		disp

n=1, i=1, b=0, p=0, TA = disp (0≤disp ≤4095)

	n	i	x	b	p	e	
opcode	1	1	1	0	0		disp

n=1, i=1, b=0, p=0, TA=(X)+disp (with index addressing mode)

- Immediate Addressing Mode

	n	i	x	b	p	e	
opcode	0	1	0				disp

n=0, i=1, x=0, Operand = disp

- Indirect Addressing Mode

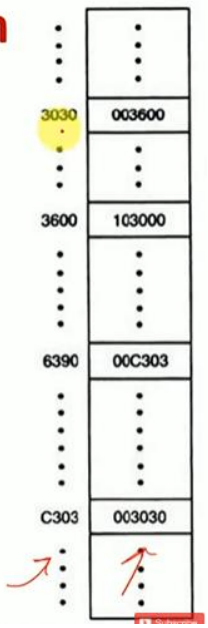
	n	i	x	b	p	e	
opcode	1	0	0				disp

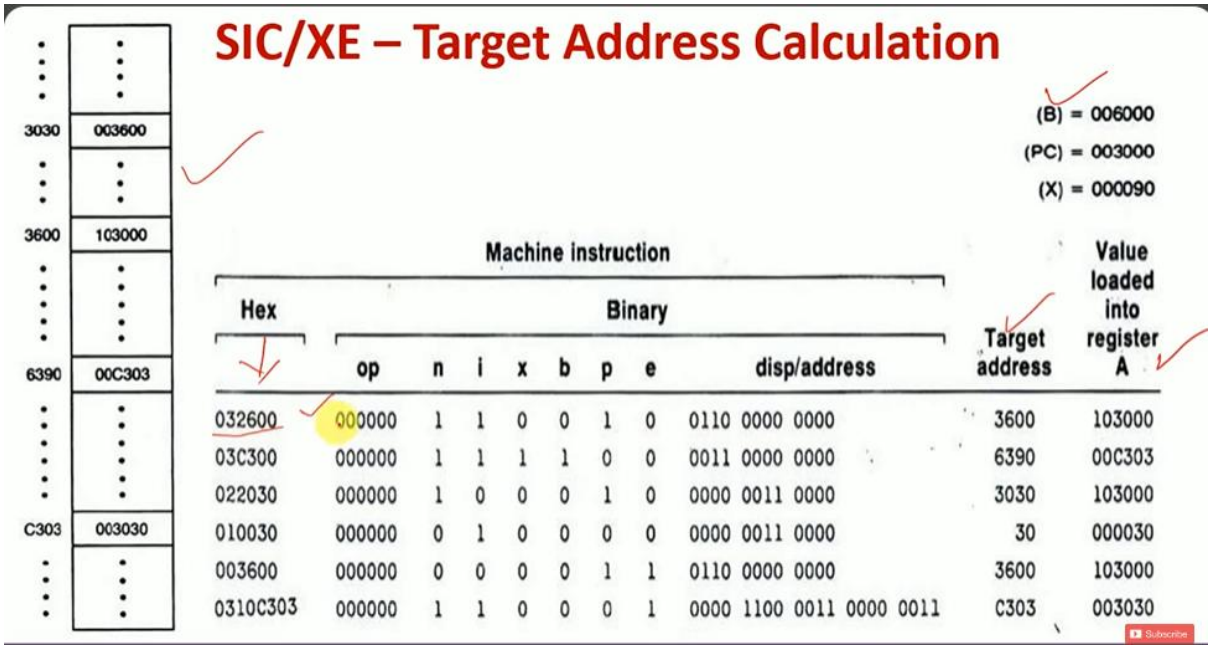
n=1, i=0, x=0, TA= (disp)

SIC/XE – Target Address Calculation

How to convert Hex code or Object code to Target address

- Calculate the Target address and the value of the following machine instructions.
- Given, (X)=000690, (B)=006030, (PC)=003060





For 1st address 32600, PC value is 3000 as given, p = 1, n = 1 and i = 1, hence its program counter relative addressing mode.
So add program counters value to the displacement - 110 -> 6, 0000 -> 0, 0000 -> 0 hence its value is 600.
So we check the value at the target address which is 103000 which is loaded into register A.

For 2nd one e = 0 therefore format 3, b = 1 hence base relative addressing mode, x = 1 hence it's indexed . Therefore target address = B + X + disp = 6000 + 90 + 300 = 6390.
So we check the value at the target address which is C303 which is loaded into register A.

For the 3rd one, n = 1, p = 1, hence it's program counter relative indirect addressing mode. Target address = PC + disp = 3000 + 30. As it's indirect addressing mode, value present at the target address calculated is actual target address , so we go to 3030 and get the value 3600 further we go to 3600 and get the value that needs to be stored in the register.
For the 4th one i = 1, so it's immediate addressing mode, so the value present in the displacement is the actual address.
Therefore Target address = disp = 30 , which is the value loaded in register A.

For the 5th one , it's extended format as well as program counter relative addressing mode.
Hence , Target address = PC + disp = 3000 + 600 = 3600.
So we check the value at the target address which is 103000 which is loaded into register A.

For the 6th one , it's neither immediate nor indirect addressing mode, it's extended format.
So whatever's present in the displacement is our target address.
Hence TA = C303

Introduction to Assembler | Functions | Directives | Data Structures

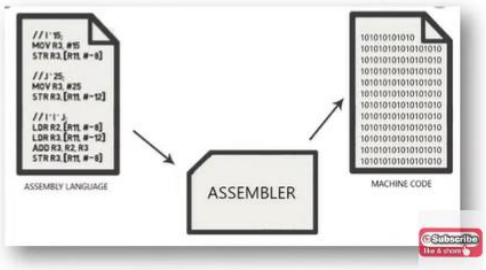
What is Assembler & Assembly Language ?

- **Assembler:** It is a system software which translate programs written in assembly language into machine language.

- **Assembly Language:** Assembly language is a kind of low level programming language, which uses symbolic codes or mnemonics as instruction.
- Some examples of mnemonics include ADD, SUB, LDA, and STA that stand for addition, subtraction, load accumulator, and store accumulator, respectively.

Applications of Assembly Language

- **Assembly language** is used for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues.
- Typical **uses** are device drivers (CD, HDD), low-level embedded systems (Keyboard, water tank indicator) and real-time systems (computer, notepad).



Elements of Assembly Language

1. **Mnemonic Operation Code:** The mnemonic operation codes for machine instructions (also called mnemonic opcodes) are easier to remember and use than numeric operation codes. **Example:** ADD, SUB, MOVE etc.

2. **Symbolic Operands:** A programmer can associate symbolic names with data or instructions and use these symbolic names as operands in assembly statements.
   Example: ADD R1,R2,R3

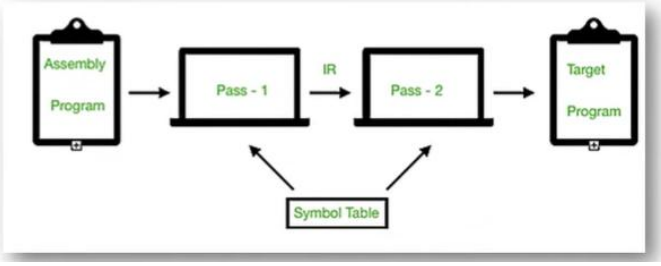
3. **Data Declaration:** Data can be declared in a variety of notations including the decimal notation.
   Example: NUM1 03 OR
   NUM1 0011H

Instruction opcode	Assembly mnemonic Instructions	Remarks
00	STOP	STOP Execution
01	ADD	Addition
02	SUB	Subtraction
03	MULT	Multiplication
04	MOVER	Move memory to Register
05	MOVEM	Move Register to Memory
06	COMP	Comparision
07	BC	Branch on condition
08	DIV	Division
09	READ	Reading m
10	PRINT	Writing to memory

Assembly Process

- Convert .asm file into .obj file.

1. **Pass 1:** Complete scan .asm file. Fine all lables, instructions & calculating corresponding address.
2. **Pass 2**: Convert all the instructions into machine language format.
3. **Symbol Table:** Stored all the information of assembly language data, variables, instructions, address etc,



Example of Assembly Language with Assembler Directives

- **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program.(JOHN is name for program)
- **MOVER:** It moves the content of literal(='3') into register operand R1.
- **MOVEM:** It moves the content of register into memory operand(X)
- **MOVER:** It again moves the content of literal(='2')

  into register operand R2 and its label is specified as L1.
- **LTORG:** It assigns address to literals(current LC value).
- **DS(Data Space):** It assigns a data space of 1 to Symbol X.
- **END:** It finishes the program execution.

```
[Label] [Opcode] [operand]

Example:  M  ADD  R1, ='3'
where, M - Label; ADD - symbolic opcode;
R1 - symbolic register operand; (='3') - Literal

Assembly Program:
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```

Design of working of Assembler

1. Analysis Phase (PASS 1 of Assembler):

- To build symbol table for synthesis phase to proceed.
- Determines address of each symbols called as memory allocation.
- Location counter used to hold address of next instruction.
- Isolated label, mnemonic opcode, operands, constants etc.
- Validate meaning & address of each statements.

2. Synthesis Phase: (PASS 2 of Assembler):

- Use data structures generated by analysis phase.
- To build machine instructions for every assembly statements as per mnemonic code & there address allocation.
- Synthesis machine instruction as per source code.

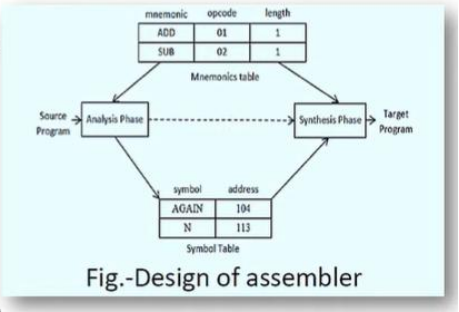


Fig.-Design of assembler

PASS 1 of Assembler (Analysis Phase)

Data Structures in Assembly Language

1. **Symbol Table (ST or SYMTAB):** Store value or address assign to the Label.

Label	Address
JOHN	200
L1	202
X	204

2. **Literal Table (LT or LITAB):** Store each literals or constants (= '3')with its location.

Index	Literal	Address
0	='3'	200
1	='2'	202

```
[Label] [Opcode] [operand]

Example:  M  ADD  R1, ='3'
where, M - Label; ADD - symbolic opcode;
R1 - symbolic register operand; (='3') - Literal

Assembly Program:
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```


PASS 1 of Assembler (Analysis Phase)

Data Structures in Assembly Language

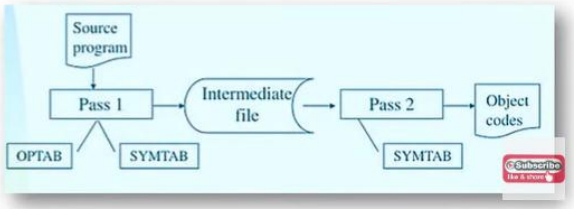
3. **Operation Code Table(OPTAB):** Store Mnemonic operation code with there opcodes & length.

Inst.	Opcode	Length(Bytes)
MOVER	3	2
MOVEM	X	1
MOVER	2	2


```
[Label] [Opcode] [operand]

Example:  M  ADD  R1, ='3'
where, M - Label; ADD - symbolic opcode;
R1 - symbolic register operand; (='3') - Literal

Assembly Program:
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```

PASS 2 of Assembler (Synthesis Phase)

- In the second pass the instructions are again read and are assembled using the symbol table.
- Basically, the assembler goes through the block of program and generates machine code for that instruction.
- Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created.
- Convert mnemonic operations code, symbolic table operands with there equivalent machine code.
- Convert data constants to internal machine representations.
- Convert complete program into .obj file.



Advantages & Disadvantages of Assembly Language Program

Advantages:

1. Hardware oriented.
2. Increasing readability by using data structure.
3. Useful in embedded system.
4. Less resources, managing size & code.
5. Access hardware drivers & system code easily as compare to high level lang.

Disadvantages:

1. Machine dependent.
2. Platform dependent, Porting to another platform is not easy.
3. More difficult to debug.
4. More complex in nature.

Definition of Assembler

- An assembler is a kind of translator that accepts the input in assembly language program and produces its machine language equivalent.
  - ✓ Ex: MASM, TASM

Basic Assembler Functions

1. Convert mnemonic operations code to their equivalent machine language.
   - Ex: STL→14, JSUB→48
2. Convert symbolic operands to their equivalent machine address.
   - Ex: Cloop →100
3. Build machine instruction in the proper format (format 1, 2, 3 or 4).
4. Convert the data constant to internal machine representation.
   - Ex: EOF→4546
5. Write the object program and the assembly listing.

Assembler Directives

Assembler directives are pseudo instructions,

- They provide definition to the assembler itself.
- They are not translated into machine operation code.
- In addition to the mnemonic machine instruction, we have used the following assembler directives.
- **START, END, BYTE, WORD, RESB, RESW**

Assembler Directives

- **START:** Specify name and starting address for the program.
- **END:** Indicate the end of the source and specify the first executable instruction in the program.
- **BYTE:** Generate character or hexadecimal constant occupying as many bytes as needed to represent the constant.
- **WORD:** Generate one-word integer constant.
- **RESB:** Reserves the indicated number of bytes for a data area.
- **RESW:** Reserves the indicated number of words for a data area.

Assembler Data Structures

- Our simpler assembler uses 2 major internal data structures.
  - **OPTAB (Operation Table)**
  - **SYMTAB (Symbol Table)**
  - **LOCCTR (Location Counter)**

Data Structures – LOCCTR Location Counter

- A Location Counter (LOCCTR) is used to be a variable and help in the assignment of addresses.
- LOCCTR initialized to be beginning address specified in the START statement.
- After each source statement is processed , the length of the assembled instruction or data area to be generated is added to LOCCTR.

Data Structures - OPTAB (Operation Table)

- It is used to look up mnemonic operation code and translate them to their machine language equivalent.
- In more complex assembler, this table also contains information about instruction format and length.
- During pass 1, OPTAB is used to look up and validate operation codes in the source program.
- During pass 2, it is used to translate the operation codes to machine language.

Data Structures - SYMTAB (Symbol Table)

- SYMTAB is used to store values assigned to labels.
- SYMTAB includes the name and value (address) for each label in the source program/
- During Pass 1, labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses (from LOCCTR).
- During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instruction.

Solved Example #1 Convert Assembly Language program to Object Program in SIC

SIC Machine – Generate Object Program – 1

- Generate the complete object program for the following assembly language program. Assume standard SIC machine and the following machine codes in hexa and also indicate the content of symbol at the end.
- LDA=00, LDX=04, STA=0C, ADD=18, TIX=2C, JLT=38, RSUB=4C

```
SUM     START   4000
FIRST   LDX     ZERO
        LDA     ZERO
LOOP    ADD     TABLE,X
        TIX     COUNT
        JLT     LOOP
        STA     TOTAL
        RSUB
TABLE   RESW    2000
COUNT   RESW    1
ZERO    WORD    0
TOTAL   RESW    1
        END     FIRST
```

SIC Machine – Generate Object Program – 1

LDA=00,
LDX=04,
STA=0C,
ADD=18,
TIX=2C,
JLT=38,
RSUB=4C

Line no	LOCATION COUNTER	LABEL	OPCODE	Operand	Object Code
1		SUM	START	4000(H)	
2	4000	FIRST	LDX	ZERO	
3	4003		LDA	ZERO	
4	4006	LOOP	ADD	TABLE, X	
5	4009		TIX	COUNT	
6	400C		JLT	LOOP	
7	400F		STA	TOTAL	
8	4012		RSUB		
9	4015	TABLE →	RESW	2000	
10	5785	COUNT →	RESW	1	
11	5788	ZERO →	WORD	0	
12	578B	TOTAL →	RESW	1	
13	578E		END	FIRST	

Upto instruction 8 all instructions are 3 bytes in size. The ninth instruction RES

W tells us to reserve 2000 words of memory. Each word is 3 bytes hence 2000 words are equivalent to 6000 bytes. Which would be equivalent to 1770 in hexadecimal. Hence we would add 1770 to 4015 in hexadecimal. Hence 4015 + 1770 = 5785.

RESW , START , END and RESB don't have any object code.
Object code has 24 bits, first 8 - opcode , 1 - x, 15 - address.

SIC Machine – Generate Object Program – 1

LDA=00,
LDX=04,
STA=0C,
ADD=18,
TIX=2C,
JLT=38,
RSUB=4C

Line no	LOCATION COUNTER	LABEL	OPCODE	Operand	Object Code
1		SUM	START	4000(H)	-
2	4000	FIRST	LDX	ZERO	045788
3	4003		LDA	ZERO	005788
4	4006	LOOP	ADD	TABLE, X	18C015
5	4009		TIX	COUNT	2C5785
6	400C		JLT	LOOP	384006
7	400F		STA	TOTAL	0C578B
8	4012		RSUB		4C0000
9	4015	TABLE	RESW	2000	-
10	5785	COUNT	RESW	1	-
11	5788	ZERO	WORD	0	000000
12	578B	TOTAL	RESW	1	-
13	578E		END	FIRST	-

If you have ,x in any instruction it would be indexed addressing mode, in all other instructions it would be in direct addressing mode.

For finding the object code , if the instruction doesn't have ,x then we , take the numbers associated with the opcode(given in the question) and for the next 4 digits we look at the address of each operand.

For the third instr. with ,x in it we can see that value of ADD = 18. Value of TABLE is 4015. It's written in 15 bits now. Which is equivalent to _100 0000 0001 0101. As the first bit tells us the addressing mode as it's indexed the first bit would be 1, hence resultant address = 1100 0000 0001 0101 which is equivalent to C015.

Length of program is 578E - 4000 which is equivalent to 178E.
2 columns = 1 byte

SIC Machine – Generate Object Program – 1

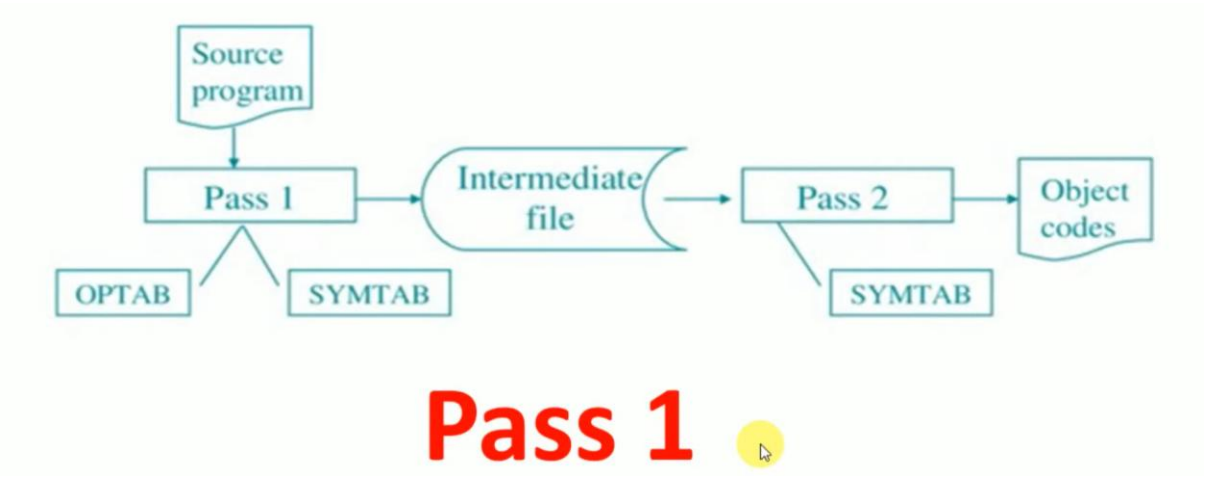
- **Header Record →only one**
- **Text Record →any number depends on program length**
- **End Record → only one**

- H^SUM___^004000^00178E
- T^004000^15^045788^005788^18C015^2C5785^384006^0C578B^4C0000
- T^005788^3^000000
- E^004000

In the text record we have written 7 object codes, each record is of 6 columns, hence total number of columns = 42. Hence 42 columns = 21 bytes. The hexadecimal equivalent of 21 is 15.

Header record:	
Col. 1	H
Col. 2–7	Program name
Col. 8–13	Starting address of object program (hexadecimal)
Col. 14–19	Length of object program in bytes (hexadecimal)
Text record:	
Col. 1	T
Col. 2–7	Starting address for object code in this record(hexadecimal)
Col. 8–9	Length of object code in this record in bytes (hexadecimal)
Col. 10–69	Object code, represented in hexadecimal (2 columns per byte of object code)
End record:	
Col. 1	E
Col. 2–7	Address of first executable instruction in object program (hexadecimal)

Pass -1 Assembler of Two-pass assembler



```
Assembler Pass 1:
begin
   read first input line
      if OPCODE ='START' then
            begin
            save #[OPERAND] as starting address
            initialize LOCCTR to starting address
            write line to intermediate file
            read next input line
            end {if START}
      else
            initialize LOCCTR to 0
```

PASS – 1

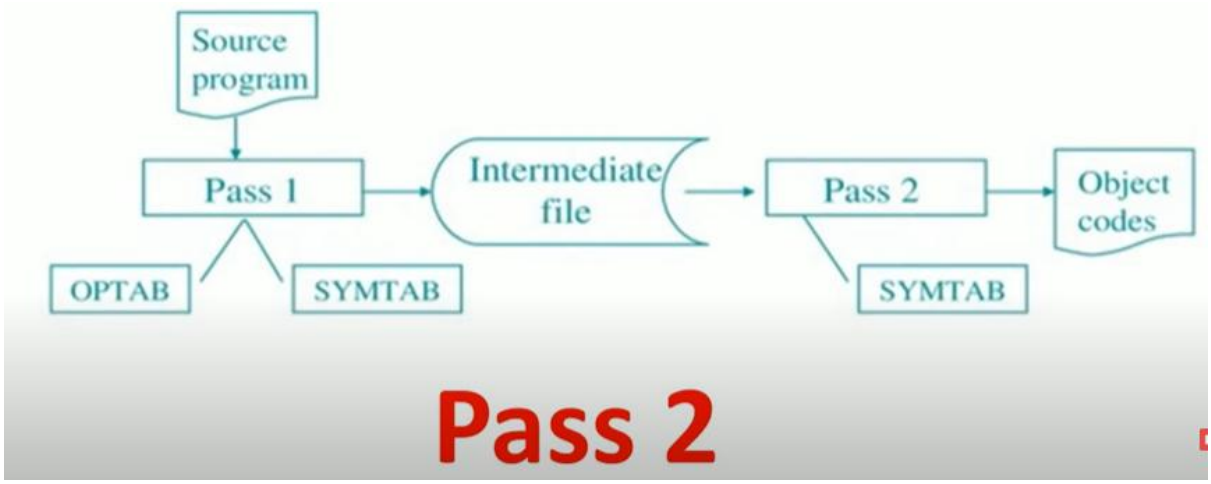
Assembler

```
while OCODE != 'END' do
      begin
         if this is not a comment line then
            begin
               if there is a symbol in the LABEL field then
                  begin
                     search SYMTAB for LABEL
                     if found then
                        set error flag (duplicate symbol)
                     else
                        insert (LABEL,LOCCTR) into SYMTAB
                  end {if symbol}
               search OPTAB for OPCODE
               if found then
                  add 3 {instruction length} to LOCCTR
               else if OPCODE='WORD' then
                     add 3 to LOCCTR
               else if OPCODE = 'RESW' then
                     add 3 * #[OPERAND] to LOCCTR
               else if OPCODE = 'RESB' then
                     add #[OPERAND] to LOCCTR
               else if OPCODE = 'BYTE' then
                  begin
                     find length of constant in bytes
                     add length to LOCCTR
                  end {if BYTE}
               else
                  set error flag (invalid operation code)
            end {if not a comment}
         write line to intermediate file
         read next input line
      end {while not END}
   write last line to intermediate file
   save (LOCCTR – starting address) as program length
end {Pass 1}
```


SYMTAB

SYMBOL	Address
FIRST	4000
LOOP	4006
TABLE	4015
COUNT	5785
ZERO	5788
TOTAL	578B

Pass - 2 Assembler of Two-pass assembler



PASS – 2 Assembler

```
begin
  read first input line (from intermediate file)
  if OPCODE = 'START' then
      begin
          write listing line
          read next input line
      end {if START}
  write Header record to object program
  initialize first Text record
  while OPCODE ≠ 'END' do
      begin
          if this is not a comment line then
              begin
                  search OPTAB for OPCODE
                  if found then
                      begin
                          if there is a symbol in OPERAND field then
                              begin
                                  search SYMTAB for OPERAND
                                  if found then
                                      store symbol value as operand address
                                  else
                                      begin
                                          store 0 as operand address
                                          set error flag (undefined symbol)
                                      end
                              end {if symbol}
                          else
                              store 0 as operand address
                          assemble the object code instruction
                      end {if opcode found}
                  else if OPCODE = 'BYTE' or 'WORD' then
                      convert constant to object code
                  if object code will not fit into the current Text record then
                      begin
                          write Text record to object program
                          initialize new Text record
                      end
                  add object code to Text record
              end {if not comment}
          write listing line
          read next input line
      end {while not END}
  write last Text record to object program
  write End record to object program
  write last listing line
end {Pass 2}
```

PASS – 2 Assembler

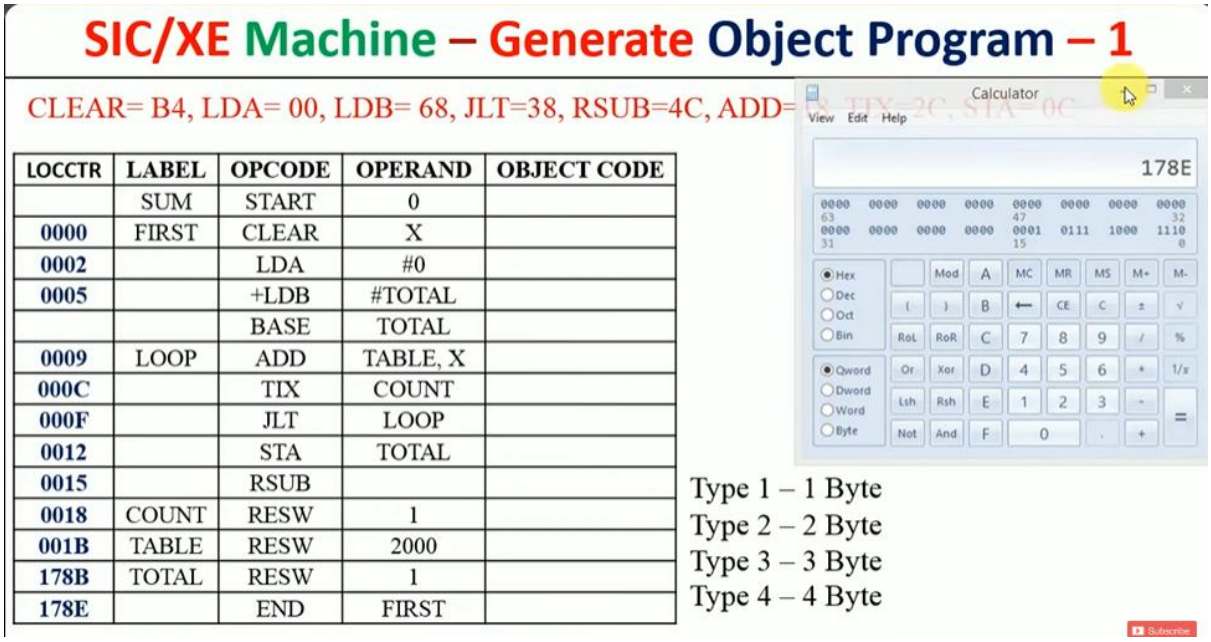
```
begin
  read first input line (from intermediate file)
  if OPCODE = 'START' then
      begin
          write listing line
          read next input line
      end {if START}
  write Header record to object program
  initialize first Text record
  while OPCODE ≠ 'END' do
      begin
          if this is not a comment line then
              begin
                  search OPTAB for OPCODE
                  if found then
                      begin
                          if there is a symbol in OPERAND field then
                              begin
                                  search SYMTAB for OPERAND
                                  if found then
                                      store symbol value as operand address
                                  else
                                      begin
                                          store 0 as operand address
                                          set error flag (undefined symbol)
                                      end
                              end {if symbol}
                          else
                              store 0 as operand address
                          assemble the object code instruction
                      end {if opcode found}
                  else if OPCODE = 'BYTE' or 'WORD' then
                      convert constant to object code
                  if object code will not fit into the current Text record then
                      begin
                          write Text record to object program
                          initialize new Text record
                      end
                  add object code to Text record
              end {if not comment}
          write listing line
          read next input line
      end {while not END}
  write last Text record to object program
  write End record to object program
  write last listing line
end {Pass 2}
```

How to generate Object Code and Object program for the SICXE Program

SIC/XE Machine – Generate Object Program – 1

- Generate the object program for the following SIC/XE program.
- CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LABEL	OPCODE	OPERAND
SUM	START	0
FIRST	CLEAR	X
	LDA	#0
	+LDB	#TOTAL
	BASE	TOTAL
LOOP	ADD	TABLE, X
	TIX	COUNT
	JLT	LOOP
	STA	TOTAL
	RSUB	
COUNT	RESW	1
TABLE	RESW	2000
TOTAL	RESW	1
	END	FIRST



LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	
0002		LDA	#0	
0005		+LDB	#TOTAL	
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

If an instr. has no register it is type 3, else if it has a register it's type 2.
Here first instr. is of type two hence we add 2 to 0000 then the 2nd instr. is of type 3 hence we add 3 to it. 3rd instr. is of type 4 hence we add 4 to it.
Now BASE is an assembly directive we should not write locctr for it.
ADD TABLE, X is of type 3 hence we add 3 to locctr.
From 0009 to 0015 instr. are of type 3, hence we add 3 to them.
From 0018 to 178B we add resultant bytes by multiplying it with appropriate factor.
1 word = 2 bytes

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	
0005		+LDB	#TOTAL	
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	R1	R2
10110100	0001	0000

CLEAR=B4 = 10110100

Mnemonic	Number
A	0
X	1
L	2
B	3
S	4
T	5
F	6
PC	8
SW	9

 # Symbol means Immediate addressing mode
If there is ,x in the instr. put 1 else 0
n → Indirect Addressing mode
i → Immediate Addressing mode
x → Index Addressing mode
b → Base relative addressing mode
p → Program counter relative addressing mode
e → 0 → Type 3
1 → Type 4

12 bit displacement → Type 3
20 bit address → Type 4

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
000000	0	1	0	0	0	0	0

LDA=00= 00000000

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
011010	0	1	0	0	0	1	0178B

LDB=68= 01101000

In this the most significant 6 bits of LDB are taken in the opcode. So the Object Code = 0110 1001 0001 0178B → 6910178B.

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
000110	1	1	1	0	1	0	00F

ADD=18= 00011000

Disp = 001B – 000C = F

PC Relative: -2048 to 2047

We find it's base relative or program counter relative via the displacement.

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
001011	1	1	0	0	1	0	009

TIX=2C= 00101100

Disp = 0018 – 000F = 9

PC Relative: -2048 to 2047

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
001110	1	1	0	0	1	0	FF7

JLT=38= 00111000

Disp = 0009 – 0012 = -9 = FF7

PC Relative: -2048 to 2047

The value of displacement is value of total - value of base register which contains total.

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
000011	1	1	0	1	0	0	000

STA= 0C= 00001100

Disp = 178B – 0015 = 1776

PC Relative: -2048 to 2047

Disp = 178B – 178B = 0

Displacement = TOTAL - BASE, but as here BASE also stores TOTAL , hence it would be address of TOTAL - address of TOTAL.

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015		RSUB		4F0000
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

Opcode	n	i	x	b	p	e	12 Disp / 20 Address
010011	1	1	0	0	0	0	000

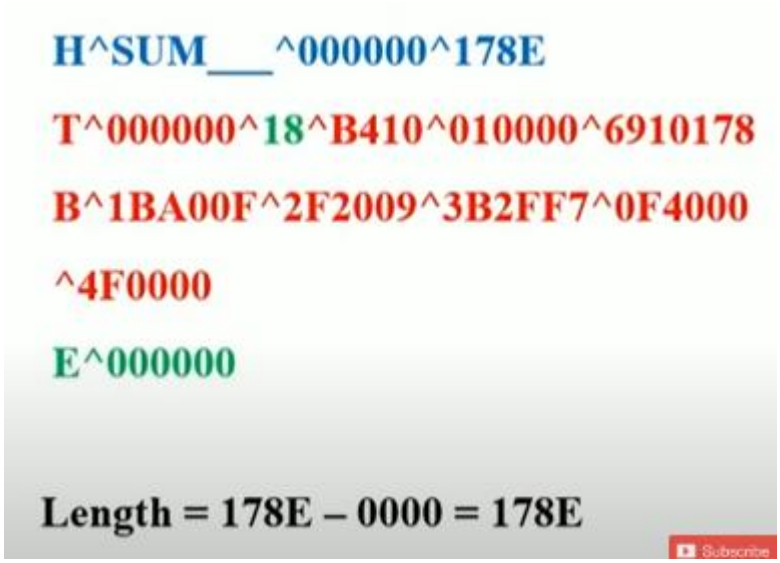
RSUB=4C= 01001100

As there are no operands or anything in RSUB so it's n and i are 1 indicating neither indirect nor immediate. Also x,b,p and e are 0 . Hence 12 bit displacement is 0.

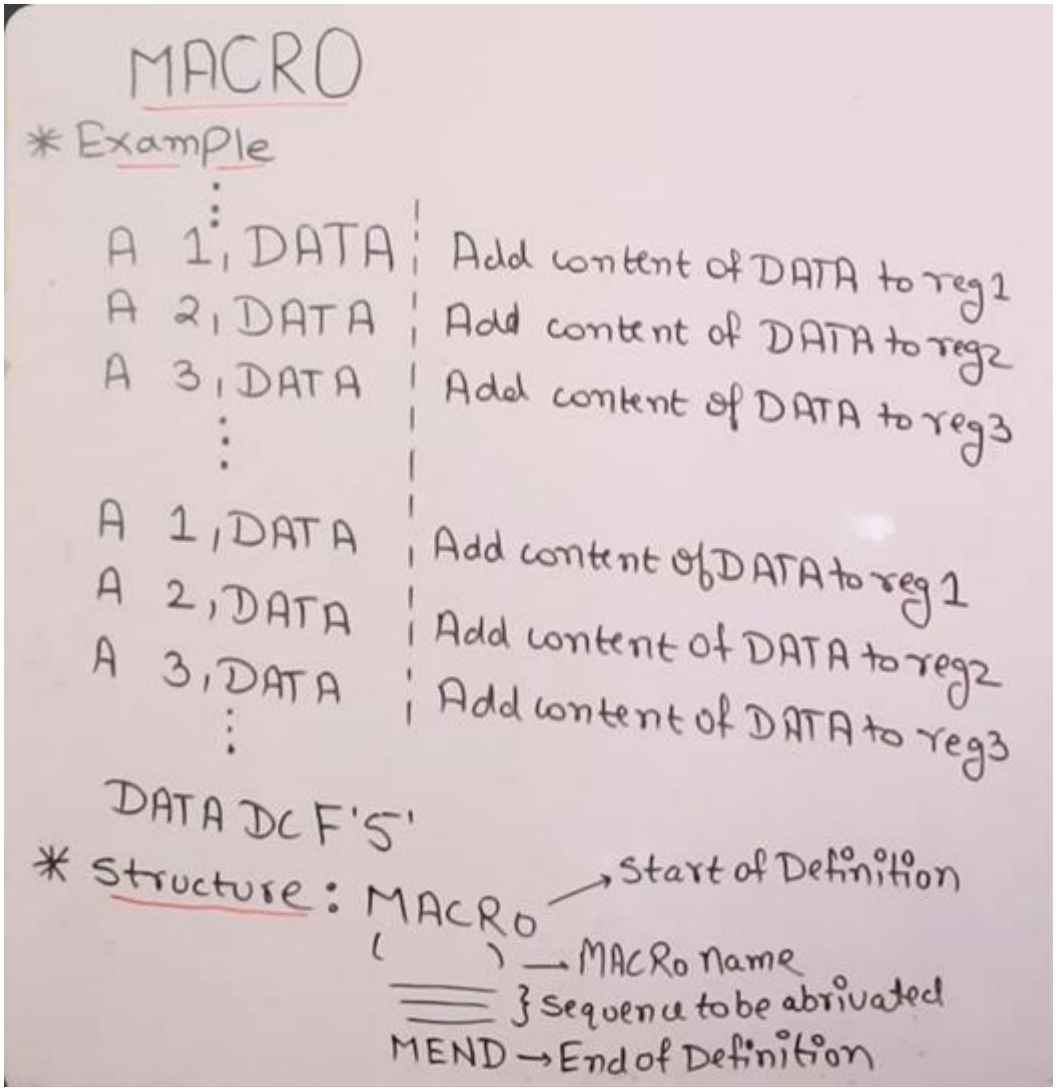
CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

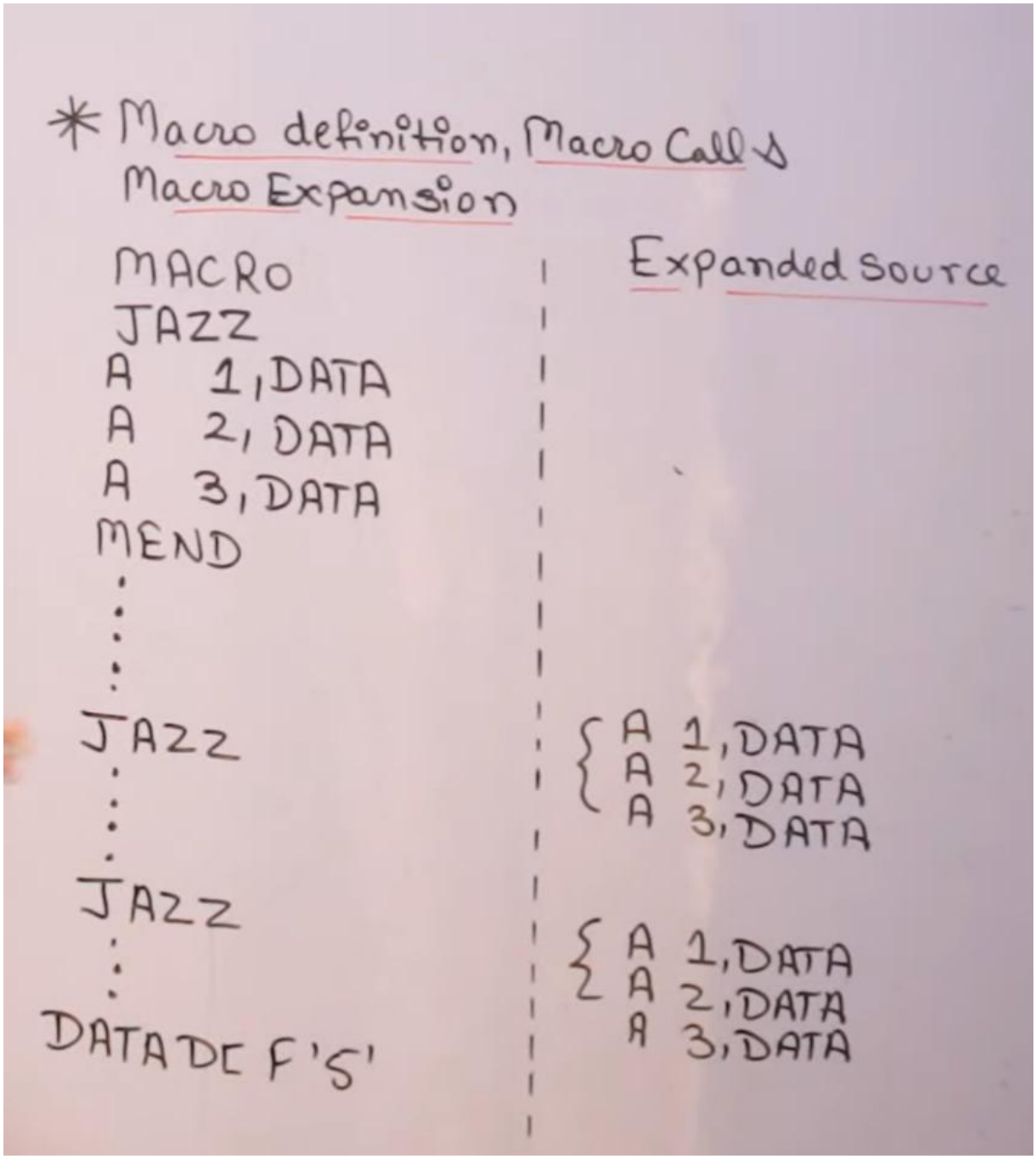
LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015		RSUB		4F0000
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	0000

For End we write the object code as 0000 (Starting Address).

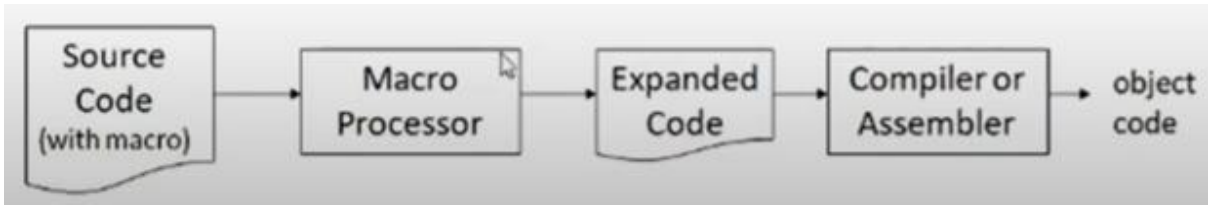


Introduction to Macro in System Programming

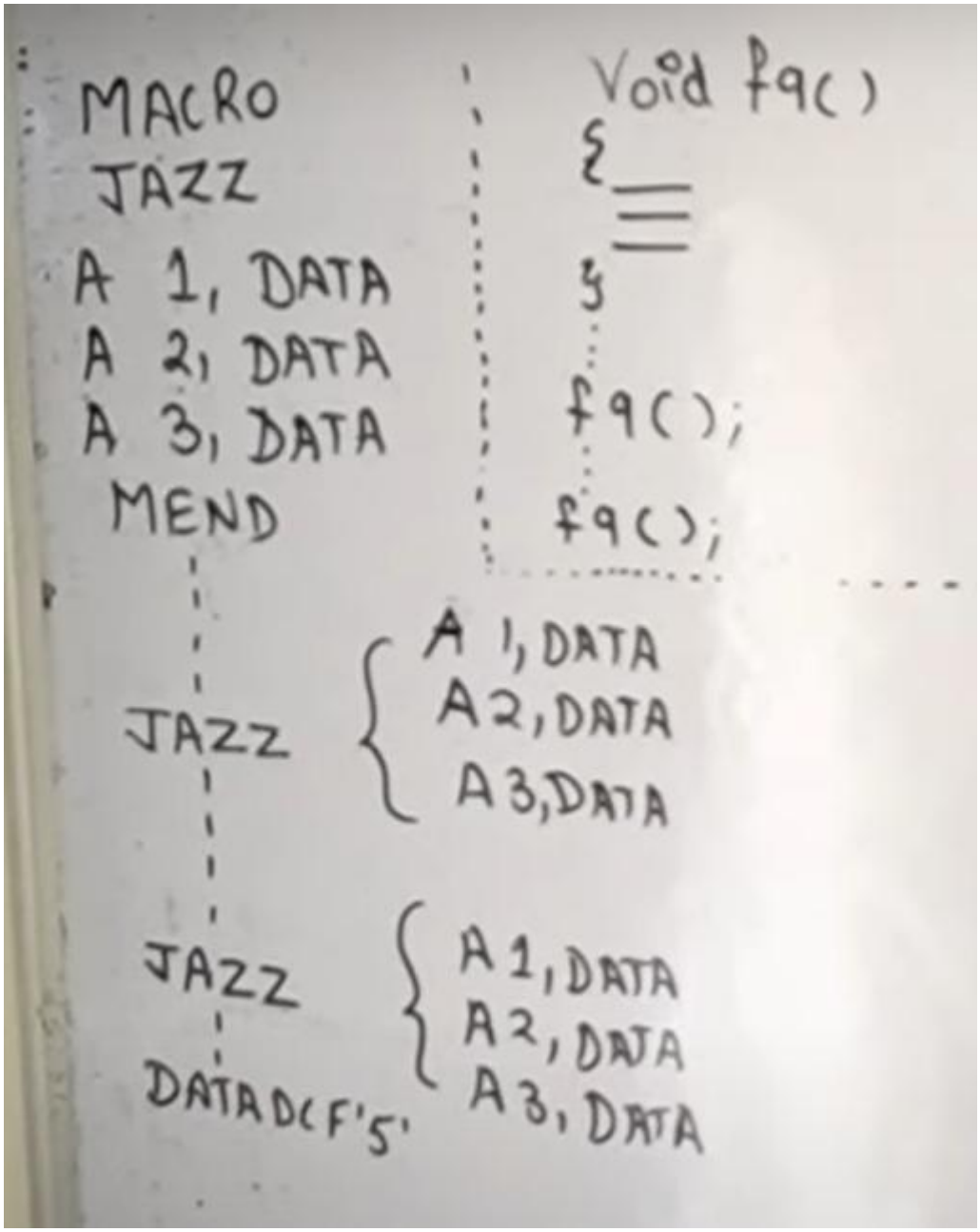




Function can return a value while macro can't.

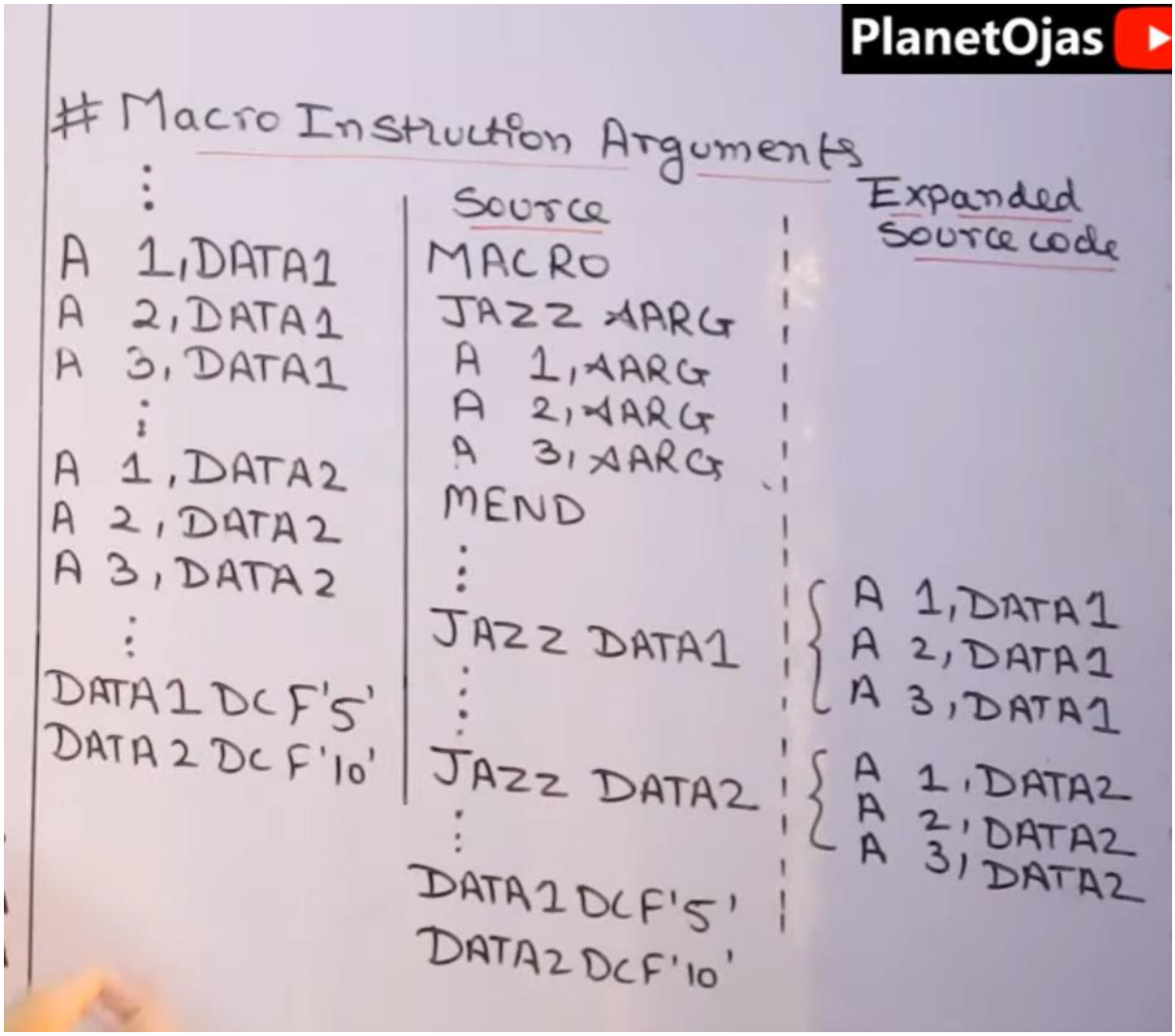


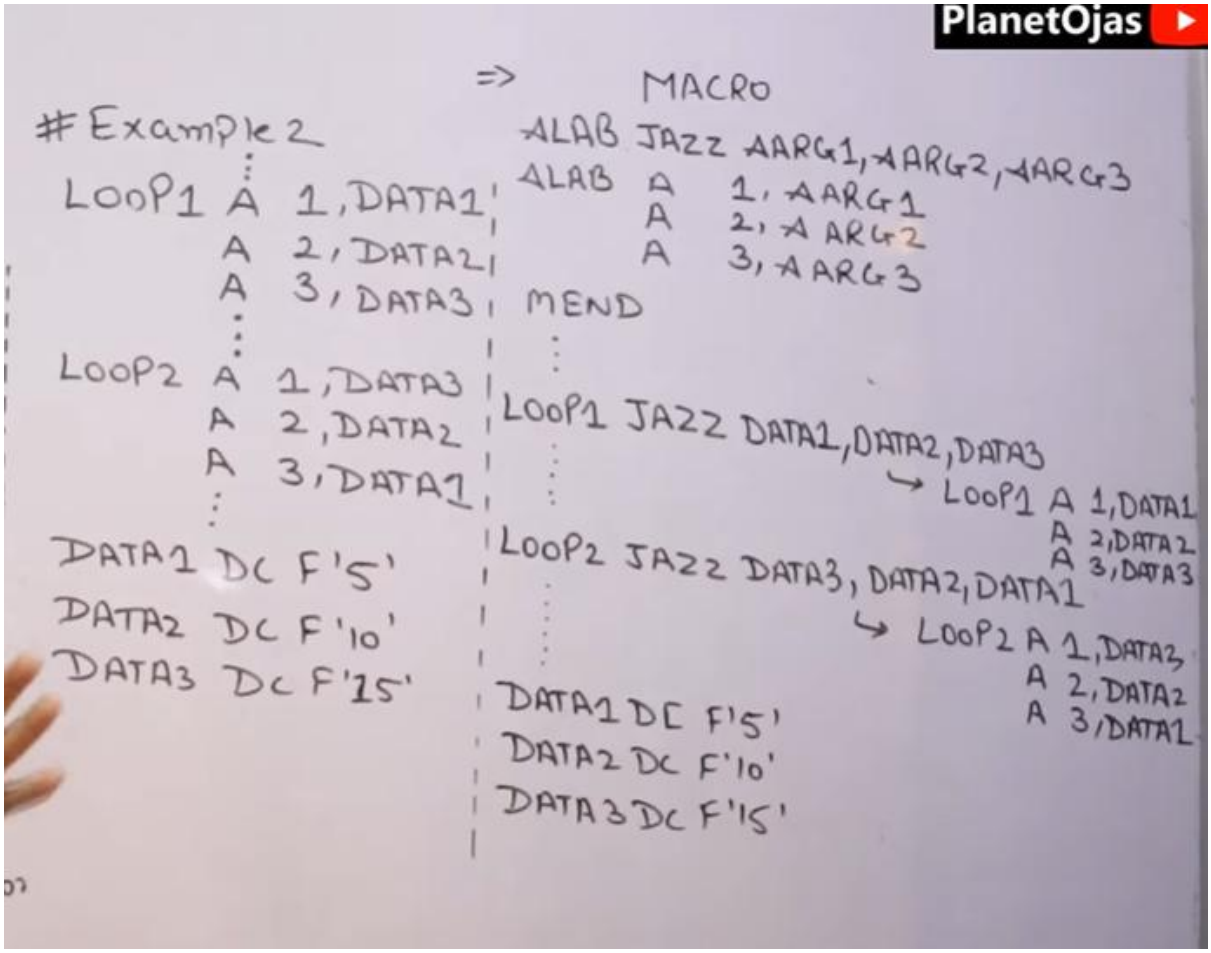
Macro vs Function



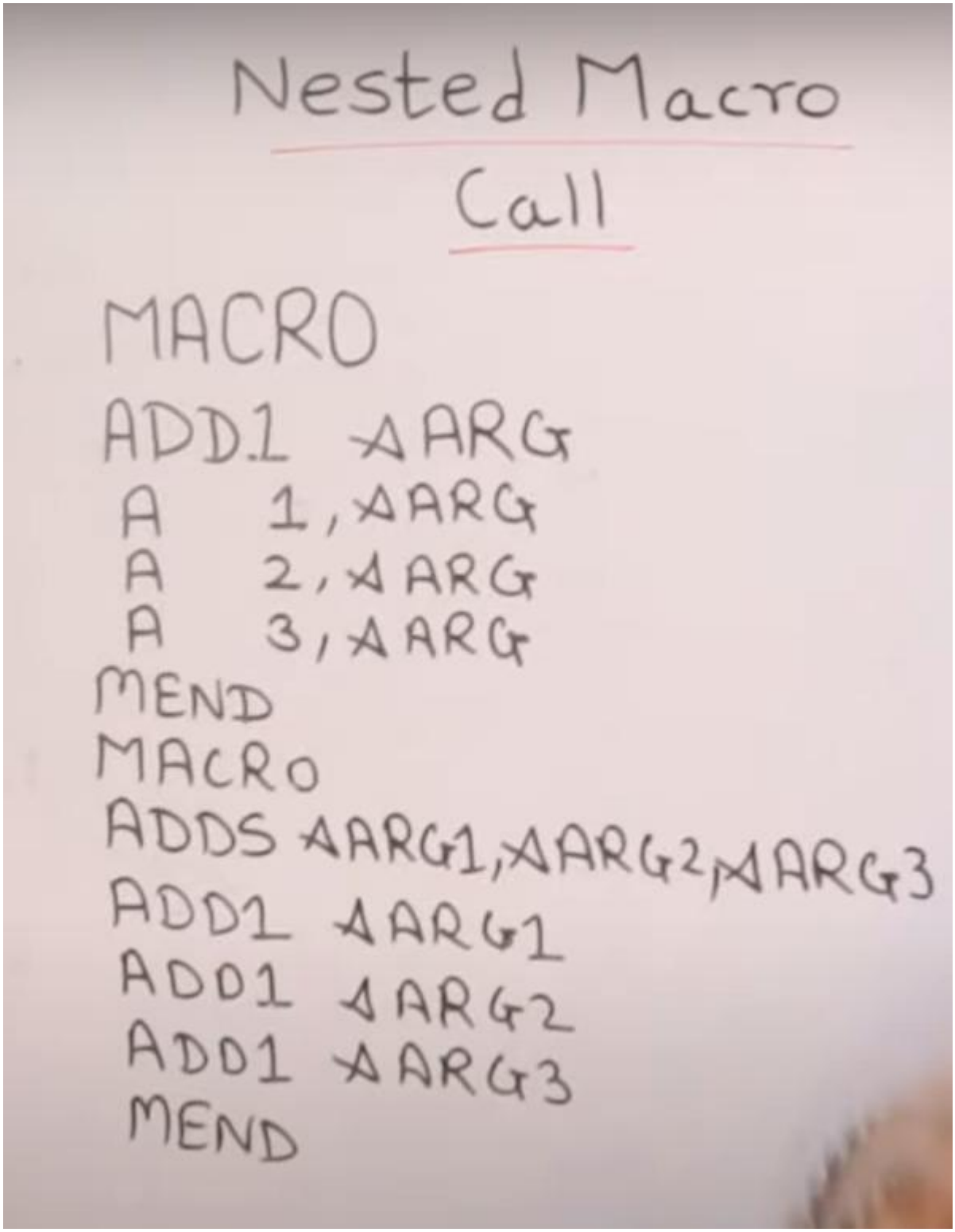
Macro	Function
1. Macro increases the size of the Program	Function does not increase the size of a Program
2. Macro don't alter the flow of Execution	Function Alters the flow of Execution
3. Program using Macro shall get Executed faster	Program using Function won't execute faster as compared to Macro
4. Macro cannot return value	Function can return a value
5. Macro is useful when small code is repeated multiple times	Function is useful when large code is repeated multiple times.

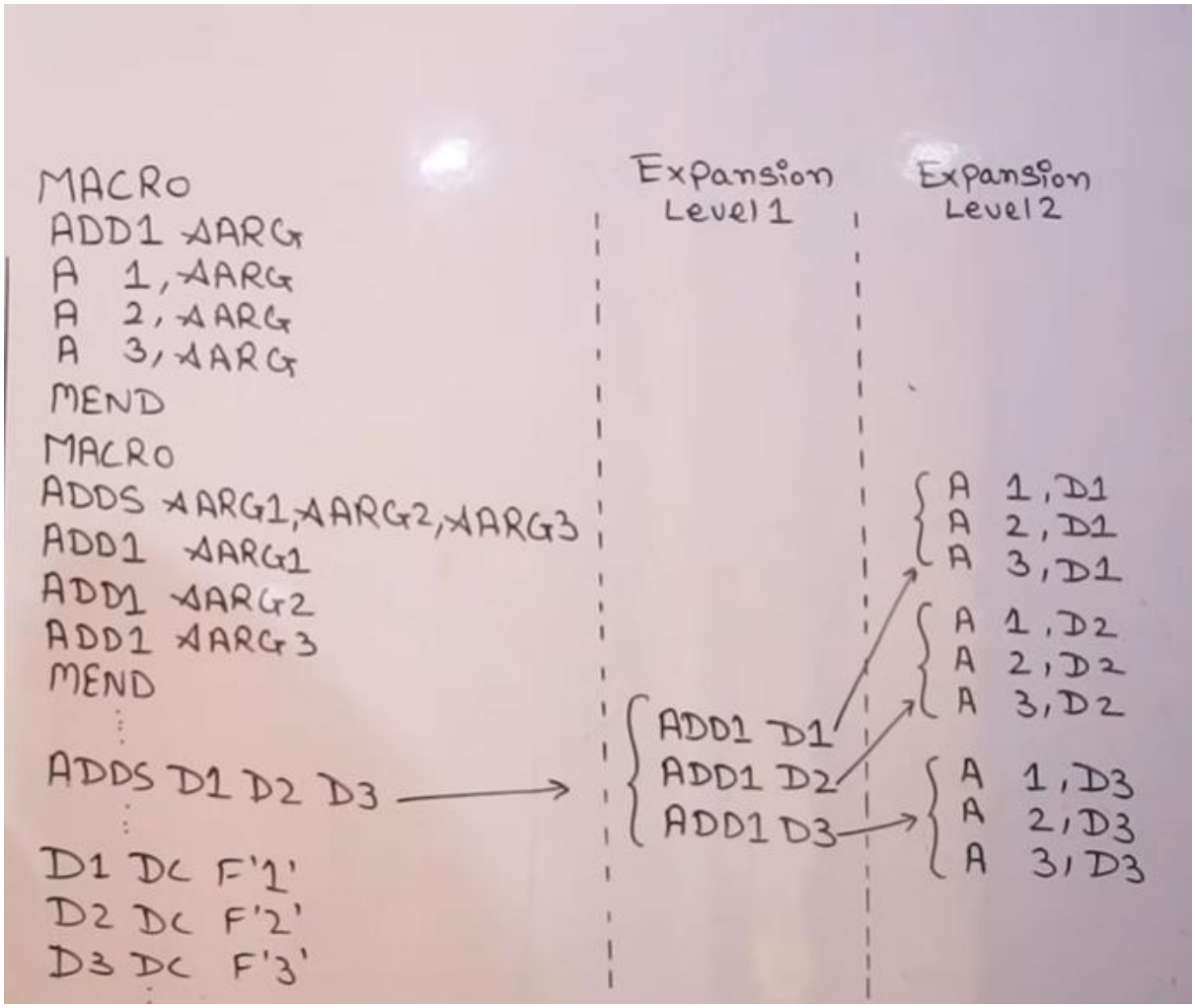
Macro Instruction Arguments



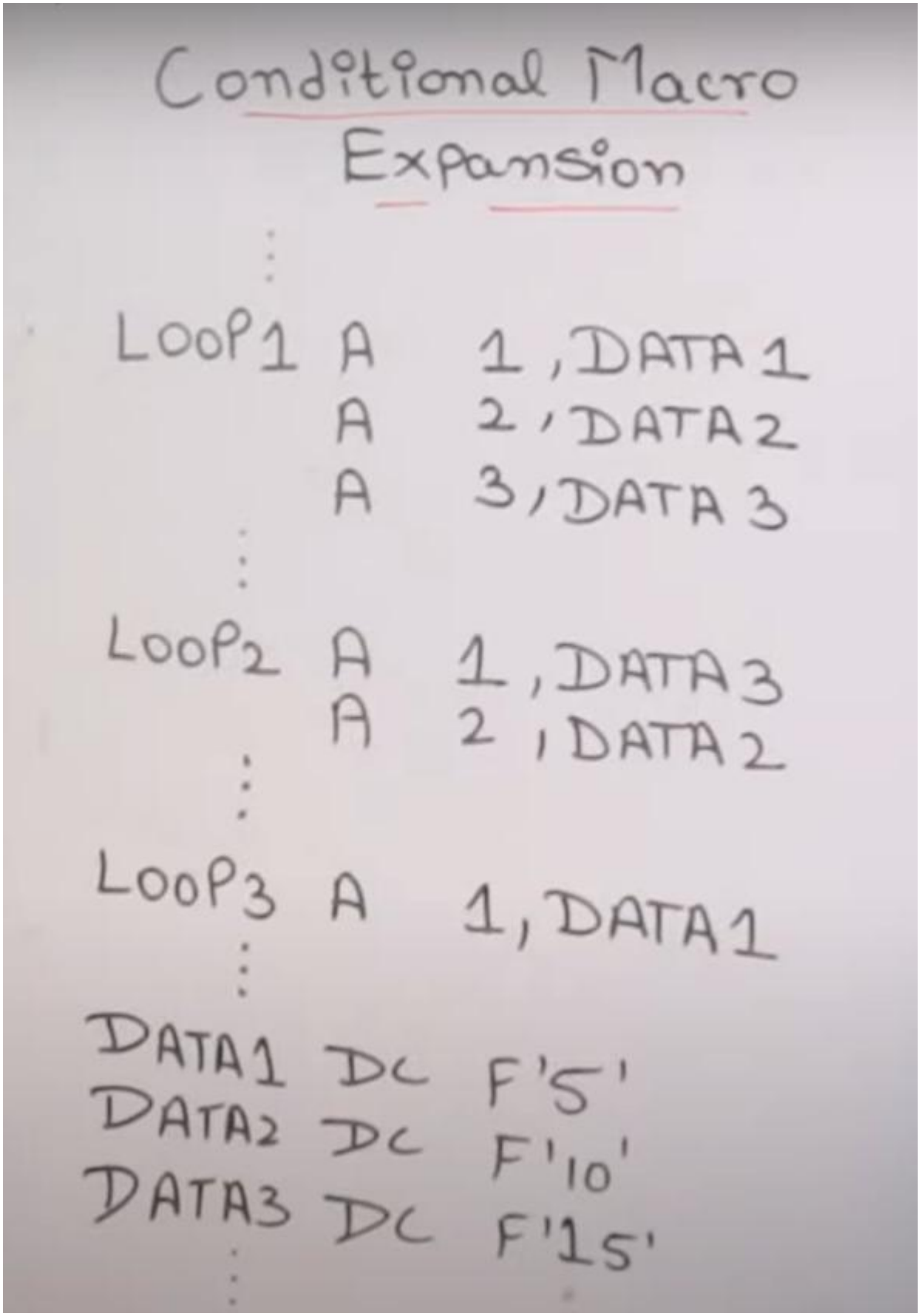

PlanetOjas
#Example 2
LOOP1 A 1,DATA1
A 2,DATA2
A 3,DATA3
LOOP2 A 1,DATA3
A 2,DATA2
A 3,DATA1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
=> MACRO
&LAB JAZZ &ARG1,&ARG2,&ARG3
&LAB A 1,&ARG1
A 2,&ARG2
A 3,&ARG3
MEND
LOOP1 JAZZ DATA1,DATA2,DATA3
↳ LOOP1 A 1,DATA1
A 2,DATA2
A 3,DATA3
LOOP2 JAZZ DATA3,DATA2,DATA1
↳ LOOP2 A 1,DATA3
A 2,DATA2
A 3,DATA1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'

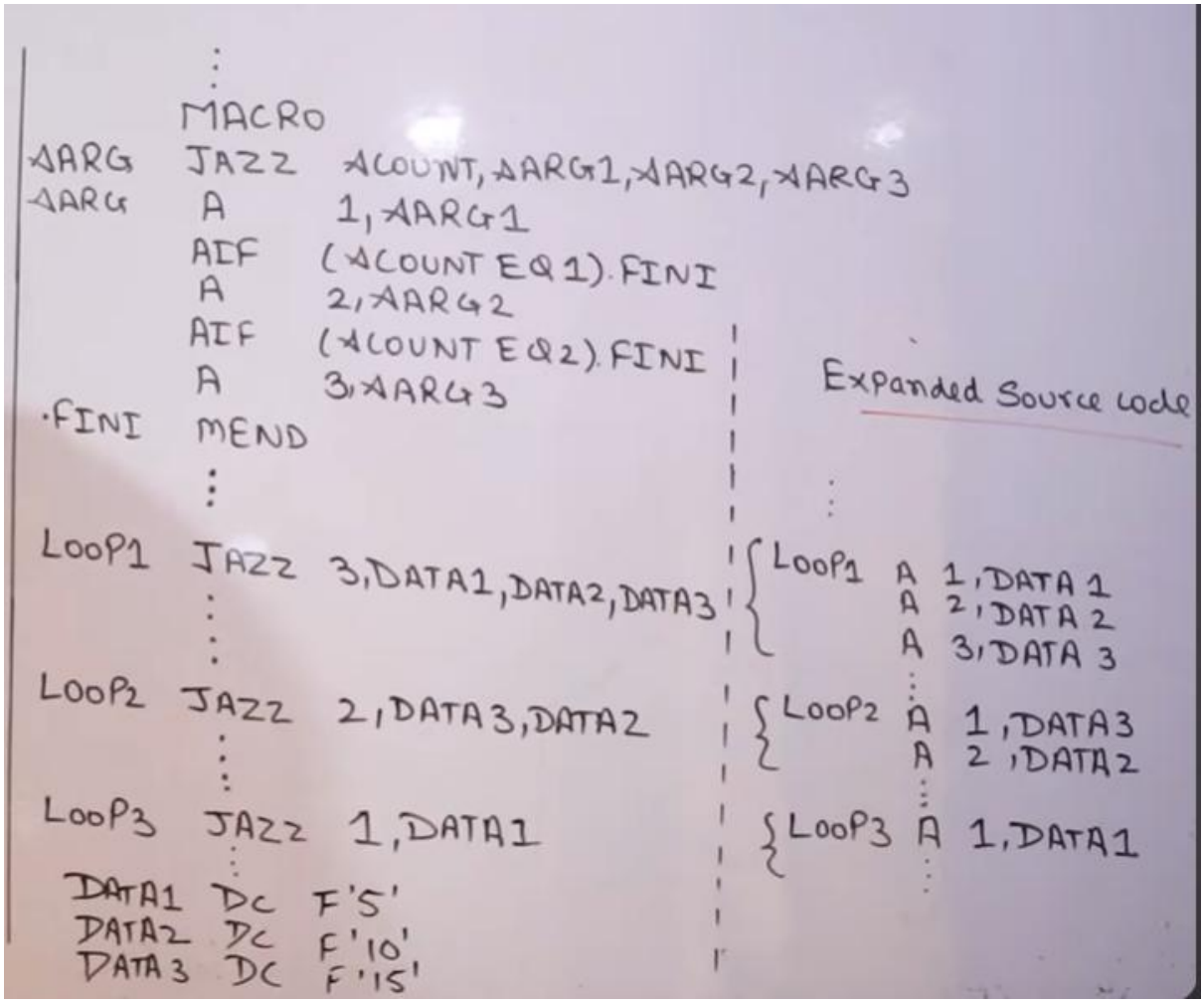
Nested Macro Call



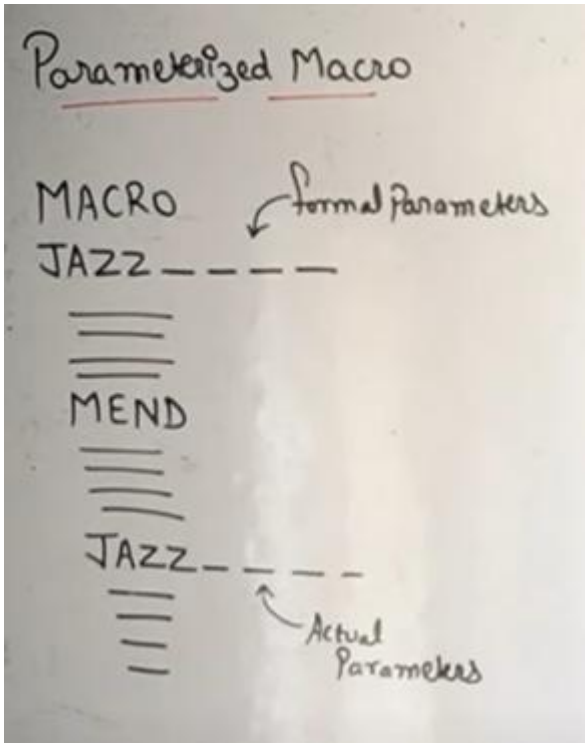

MACRO
ADD1 &ARG
A 1,&ARG
A 2,&ARG
A 3,&ARG
MEND
MACRO
ADDS &ARG1,&ARG2,&ARG3
ADD1 &ARG1
ADD1 &ARG2
ADD1 &ARG3
MEND
ADDS D1 D2 D3
D1 DC F'1'
D2 DC F'2'
D3 DC F'3'
Expansion Level 1
Expansion Level 2
ADD1 D1
ADD1 D2
ADD1 D3
A 1,D1
A 2,D1
A 3,D1
A 1,D2
A 2,D2
A 3,D2
A 1,D3
A 2,D3
A 3,D3

Conditional Macro Expansion

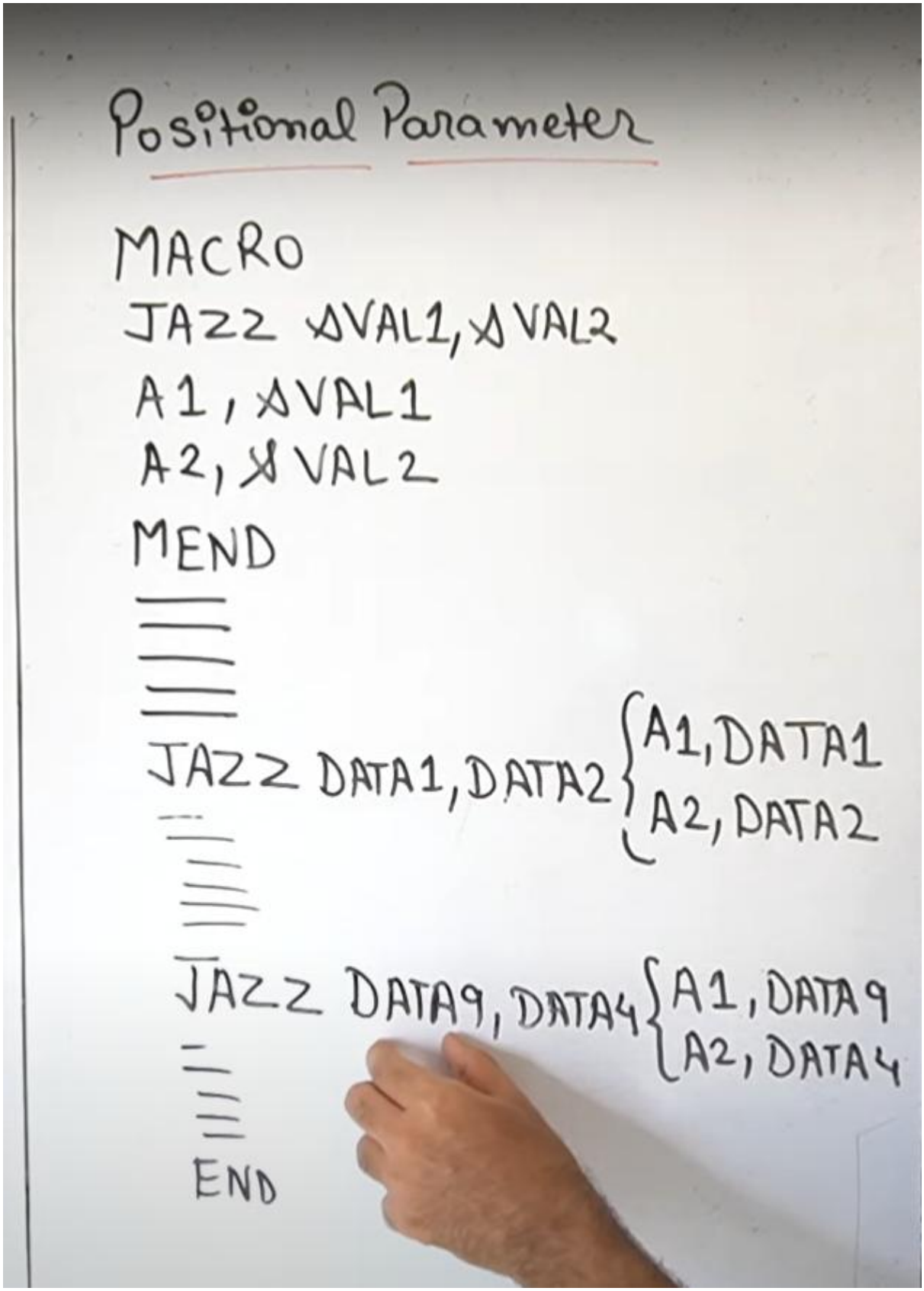



MACRO
&ARG JAZZ &COUNT,&ARG1,&ARG2,&ARG3
&ARG A 1,&ARG1
AIF (&COUNT EQ 1).FINI
A 2,&ARG2
AIF (&COUNT EQ 2).FINI
A 3,&ARG3
.FINI MEND
LOOP1 JAZZ 3,DATA1,DATA2,DATA3
LOOP2 JAZZ 2,DATA3,DATA2
LOOP3 JAZZ 1,DATA1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
Expanded Source code
LOOP1 A 1,DATA1
A 2,DATA2
A 3,DATA3
LOOP2 A 1,DATA3
A 2,DATA2
LOOP3 A 1,DATA1

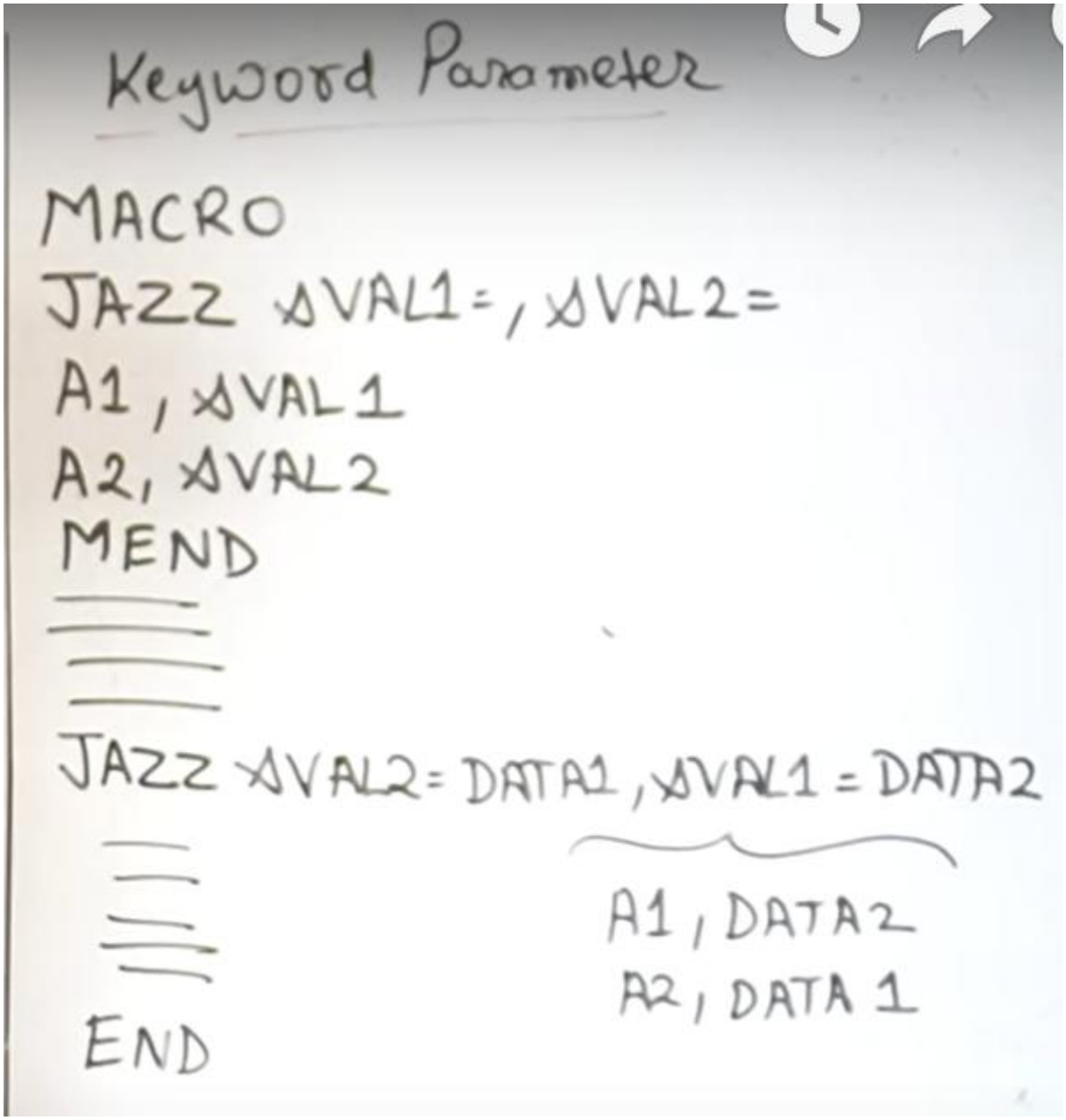
Parameterized Macro



Positional Parameter



Keyword Parameter



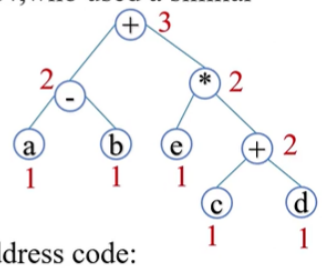
Register allocation through labeled expression tree |Optimal Code Generation |Ershov Numbers

Optimal Code Generation for Expression Tree

- We introduce a numbering scheme for the nodes of an expression tree (a syntax tree for an expression) that allows us to generate optimal code for an expression tree.
- **Ershov Numbers**
- We begin by assigning to each node of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries.
- These numbers are sometimes called Ershov numbers, after A. Ershov,who used a similar scheme for machines with a single arithmetic register.
- For our machine model, the rules are:

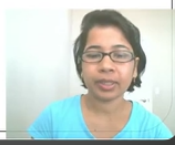
1. Label all leaves 1.
2. The label of an interior node with one child is the label of its child.
3. The label of an interior node with two children is

a) The larger of the labels of its children, if those labels are different.
b) One plus the label of its children if the labels are the same.



- Ex :Expression tree for expression (a- b) + e ∗ (c + d) or the three-address code:
- t1 = a - b
- t2 = c + d
- t3 = e * t2
- t4 = t1 + t3

- 3 registers are required.



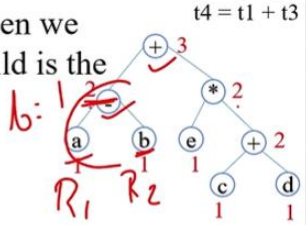
Generating Code From Labeled Expression Trees

- METHOD: The steps below are applied, starting at the root of the tree.
- If label k, then only k registers will be used. There is a "base" b ≥ 1 for the registers ,actual registers used are R_b, R_{b+1},….. R_{b+k-1}. The result always appears in R_{b+k-1}.

1. To generate machine code for an interior node with label k and two equal labels children:
   a) Recursively generate code for the right child, using base b + 1. The result of the right child appears in register R_{b+k-1}.
   b) Recursively generate code for the left child, using base b; the result appears in R_{b+k-2}.
   c) Generate the instruction OP R_{b+k-1}, R_{b+k-2}, R_{b+k-1}, OP is the operation for the interior node.
2. Suppose we have an interior node with label k and children with unequal labels. Then "big" child, has label k, and "little" child, has some label m < k :
   a) Recursively generate code for the big child, using base b; appears in register R_{b+k-1}.
   b) Recursively generate code for the little child, using base b; appears in register R_{b+m-1}.
   c) Generate the instruction OP R_{b+k-1} ,R_{b+m-1}, R_{b+k-1} or the instruction OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1}, depending on whether the big child is the right or left child, respectively.
3. For a leaf representing operand x, if the base is b generate the instruction LD R_b, x.

- Ex : Since the label of the root is 3, the result will appear in R3, and only R1, R2, and R3 will be used. The base for the root is b = 1.
- Since the root has children of equal labels, right child first, When we generate code for the right child of the root, we find the big child is the right child and the little child is the left child.
- We thus generate code for the right child first, with b = 2.
- LD R3, d
- LD R2, c
- ADD R3, R2, R3
- LD R2, e
- MUL R3, R2, R3
- For the left child of the root , base 1.
- LD R2, b
- LD R1, a
- SUB R2, R1, R2

t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3



- interior node
  - =
  - Right child , base=b+1,R_{b+k-1}
  - Left child , base=b,R_{b+k-2}
  - OP R_{b+k-1}, R_{b+k-2}, R_{b+k-1}
  - ≠
  - Big child, base=b, R_{b+k-1}
  - little child, base b, R_{b+m-1}
  - OP R_{b+k-1} ,R_{b+m-1}, R_{b+k-1} or OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1},
- leaf LD R_b, x.