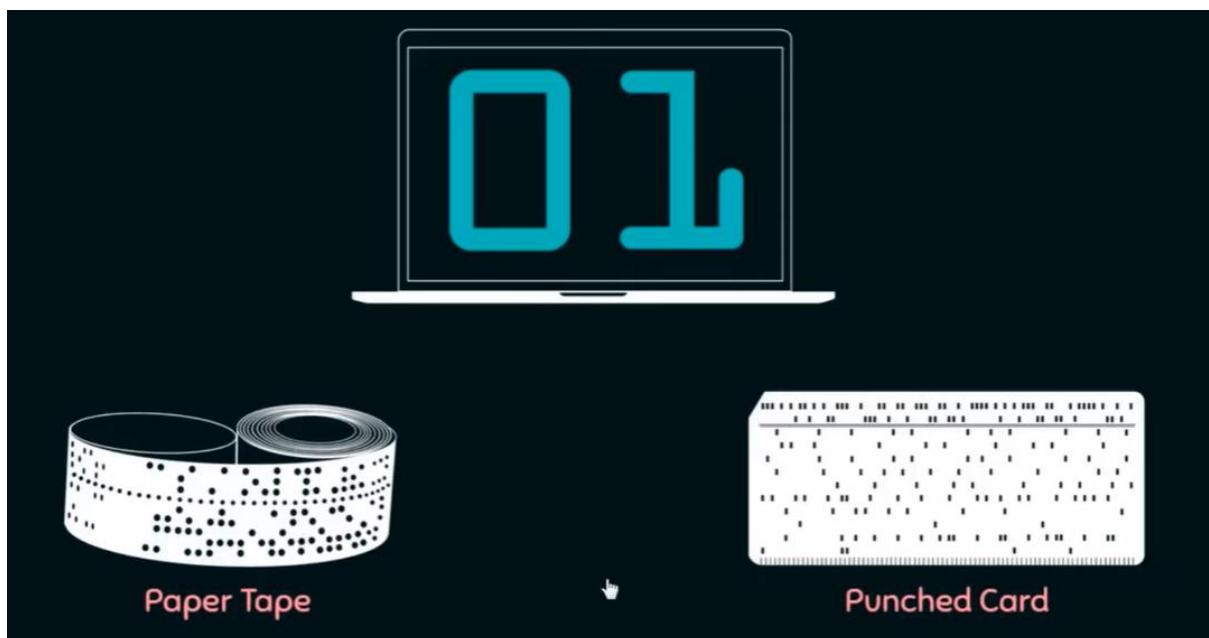


System Programming and Compiler Construction

<https://github.com/Dare-marvel/>

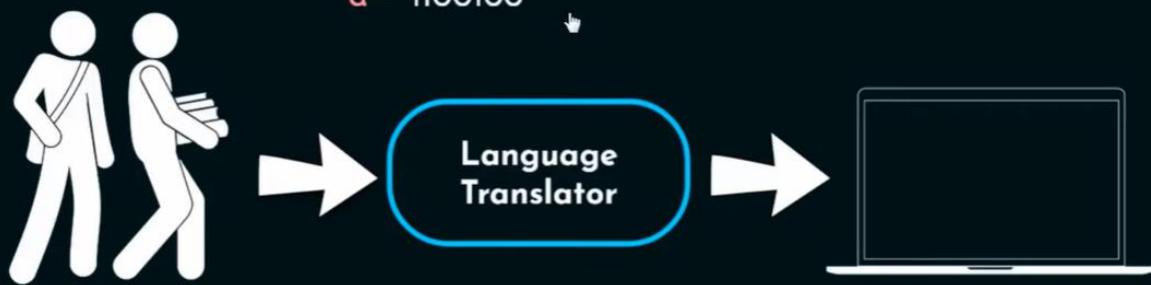
Introduction to Compiler Design



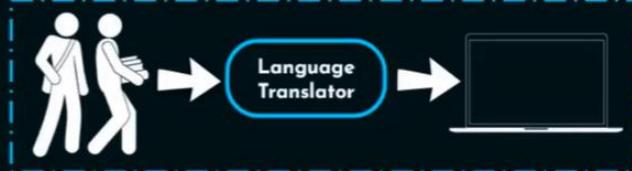
Punched Card:

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

ASCII



Language Translator:



i. Assembler:

```
MOV R1, 02H  
MOV R2, 03H  
ADD R1, R2  
STORE X, R1
```

Assembly
Language

Assembler

```
0110100101010  
0010101010010  
0100111100101  
01010101010101
```

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

```
#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

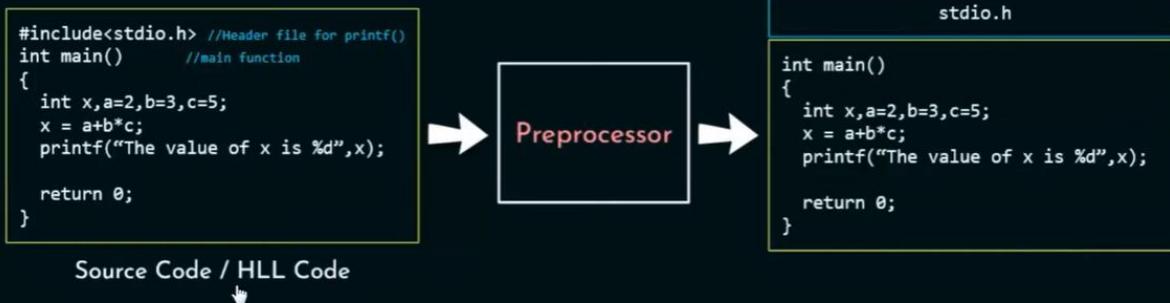
```
0010101010010
10110100101010
11010101010010
10011110010101
01010101010101
11010101010010
10011110010101
```

Machine Code

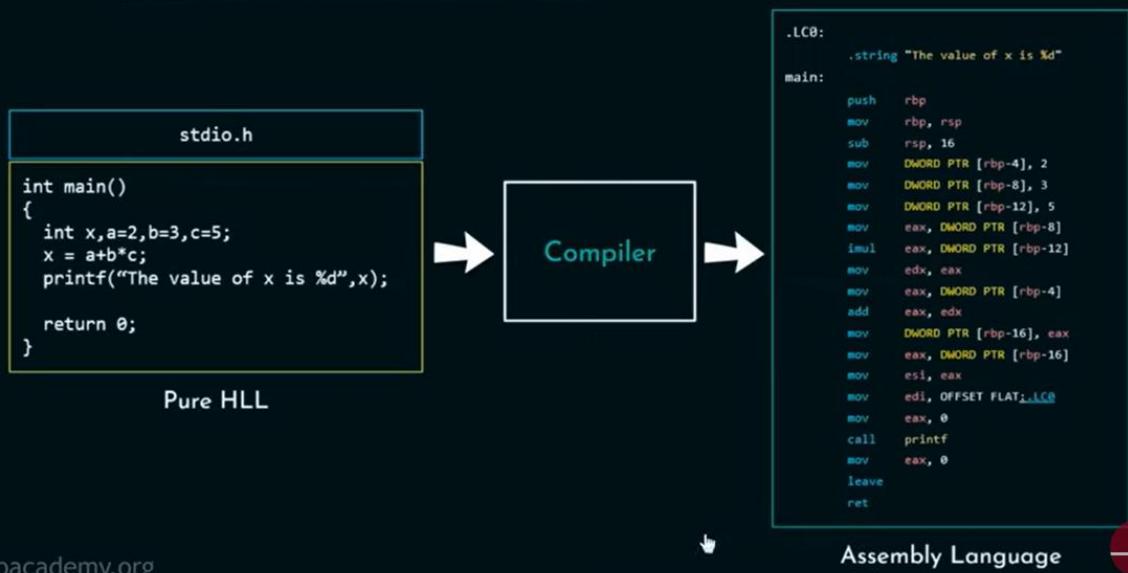
Language Translator – Internal Architecture



Language Translator – Internal Architecture



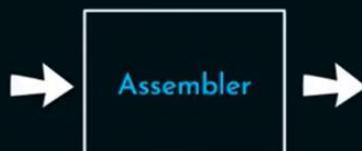
Language Translator – Internal Architecture



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



i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
⋮

Relocatable
Machine Code



Language Translator – Internal Architecture

i+0:001010101001
i+1:0101101001100
i+2:10101101010101
i+3:0100101001101
i+4:11100101010101
i+5:01010101010111
⋮

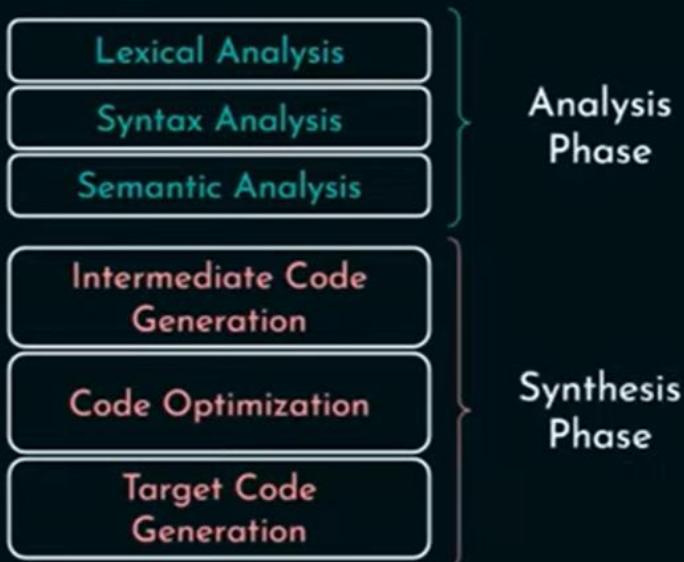
Relocatable
Machine Code



0x000004B8: 001010101001
0x000004B9: 0101101001100
0x000004BA: 10101101010101
0x000004BB: 0100101001101
0x000004BC: 11100101010101
0x000004BD: 010101010111
⋮

Absolute
Machine Code

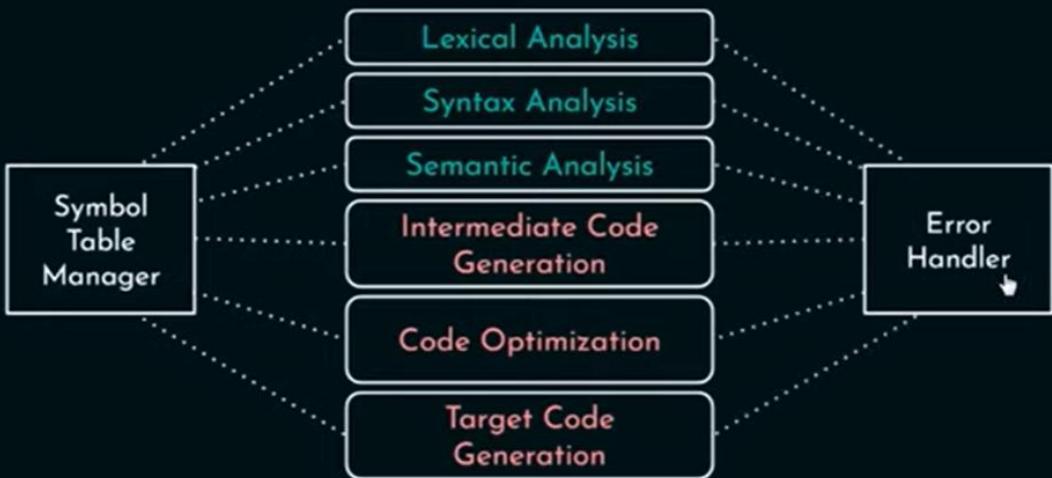
Compiler - Internal Architecture



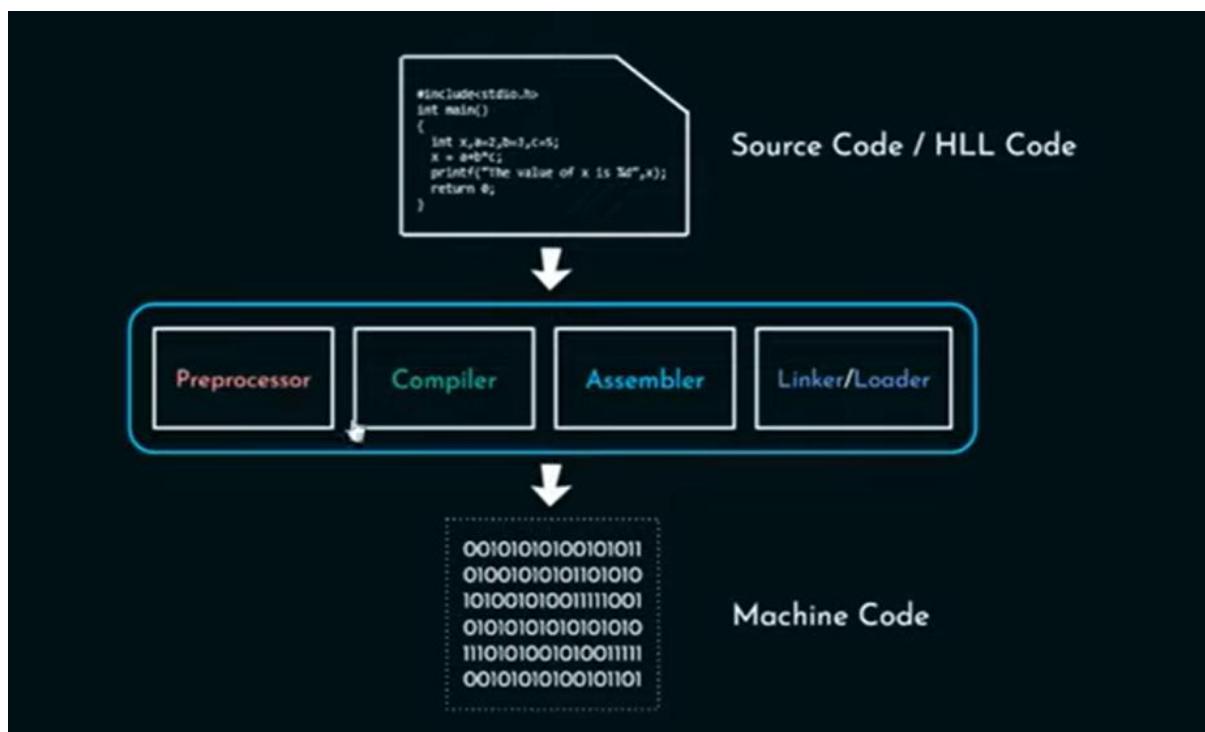
Compiler - Internal Architecture

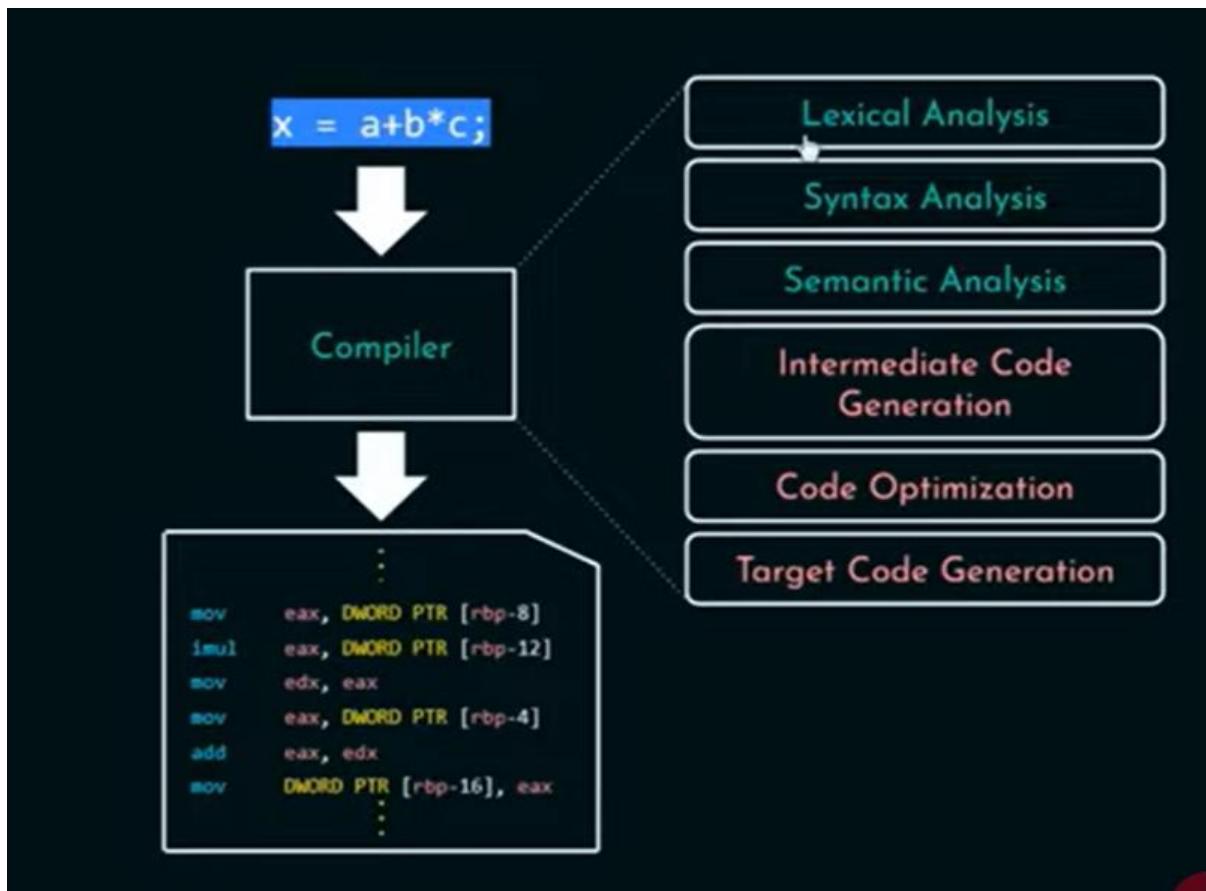


Compiler – Internal Architecture

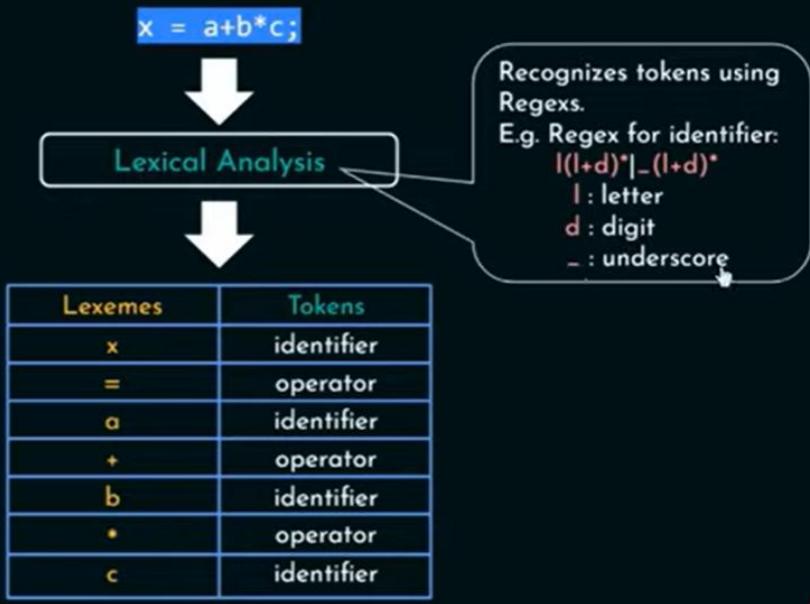


Different Phases of Compiler





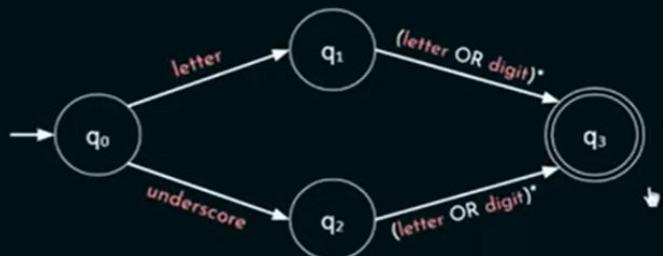
Lexical Analyzer:



Lexical Analyzer:

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

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



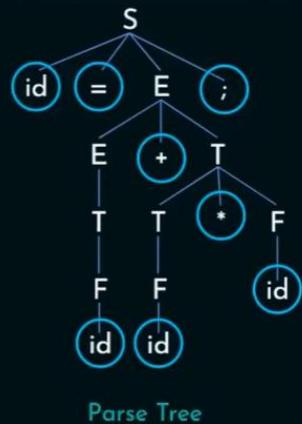
Syntax Analyzer:

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

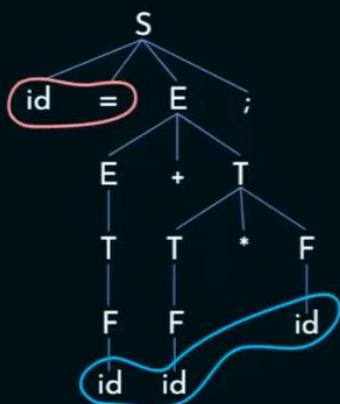
Syntax Analysis

$S \rightarrow id = E ;$
 $E \rightarrow E + T | T$
 $T \rightarrow T * F | F$
 $F \rightarrow id$

id = id + id * id ;



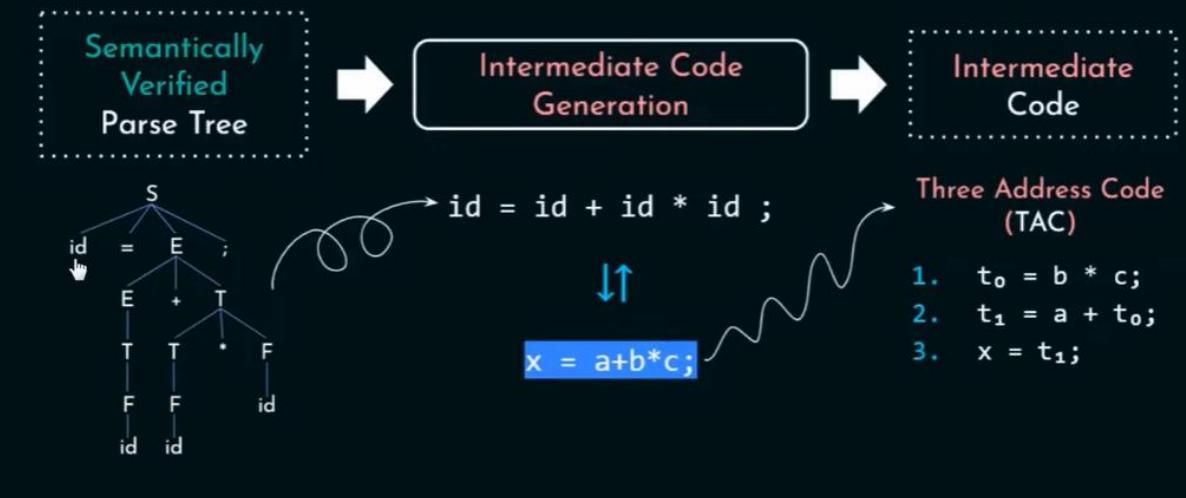
Semantic Analyzer:



Semantic Analysis

Semantically Verified Parse Tree

Intermediate Code Generator:



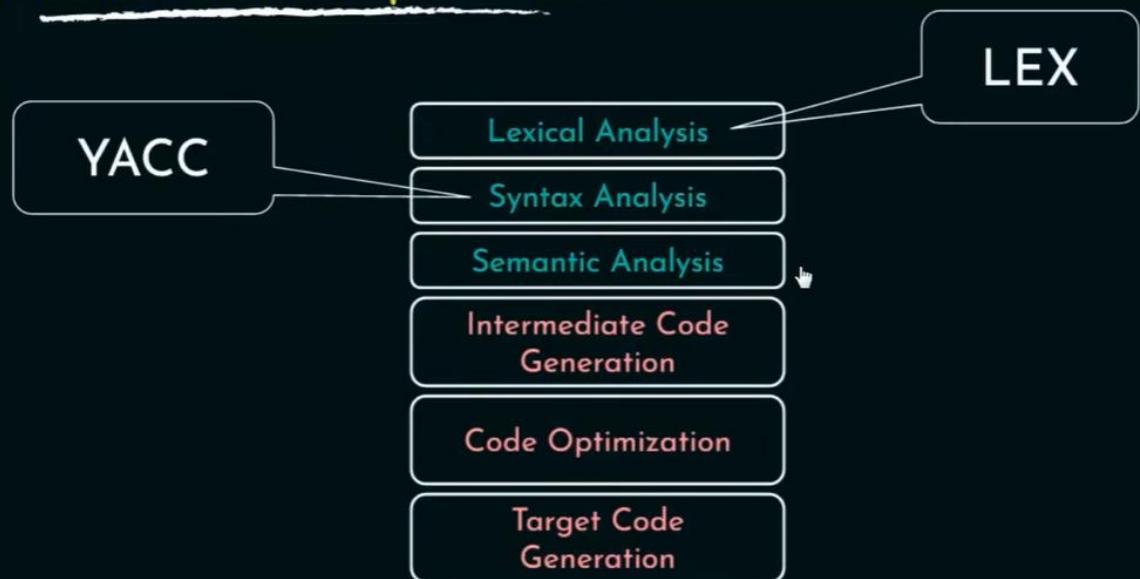
Code Optimizer:



Target Code Generator:

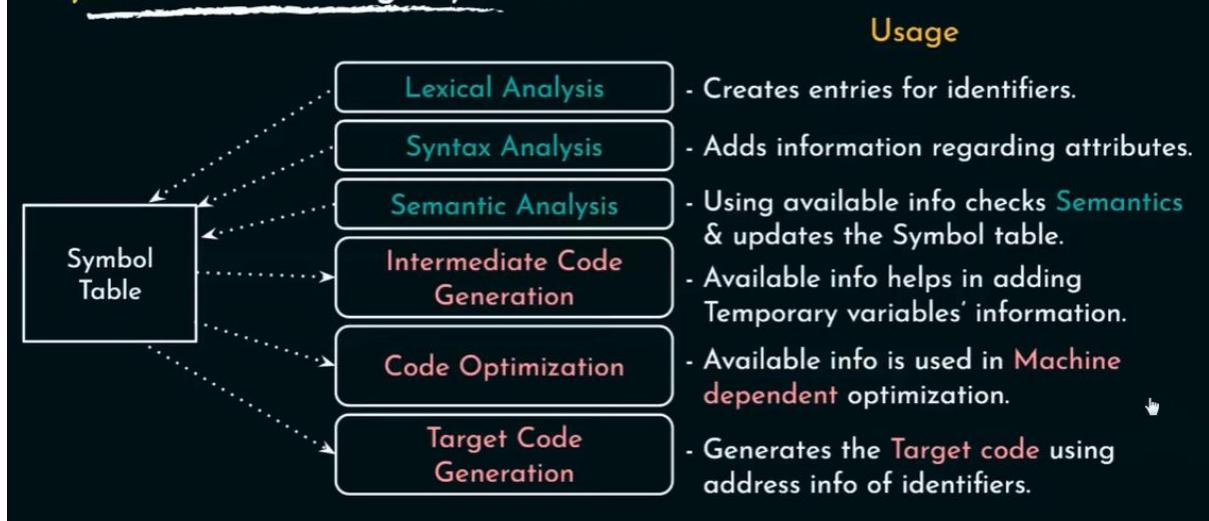


Tools for Practical Implementation:



Symbol Table

Symbol Table – Usage by Phases



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?

- (A) Three Address Code
(B) Abstract Syntax Tree (AST)
(C) Symbol Table
(D) Control Flow Graph (CFG)

GATE 2021

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

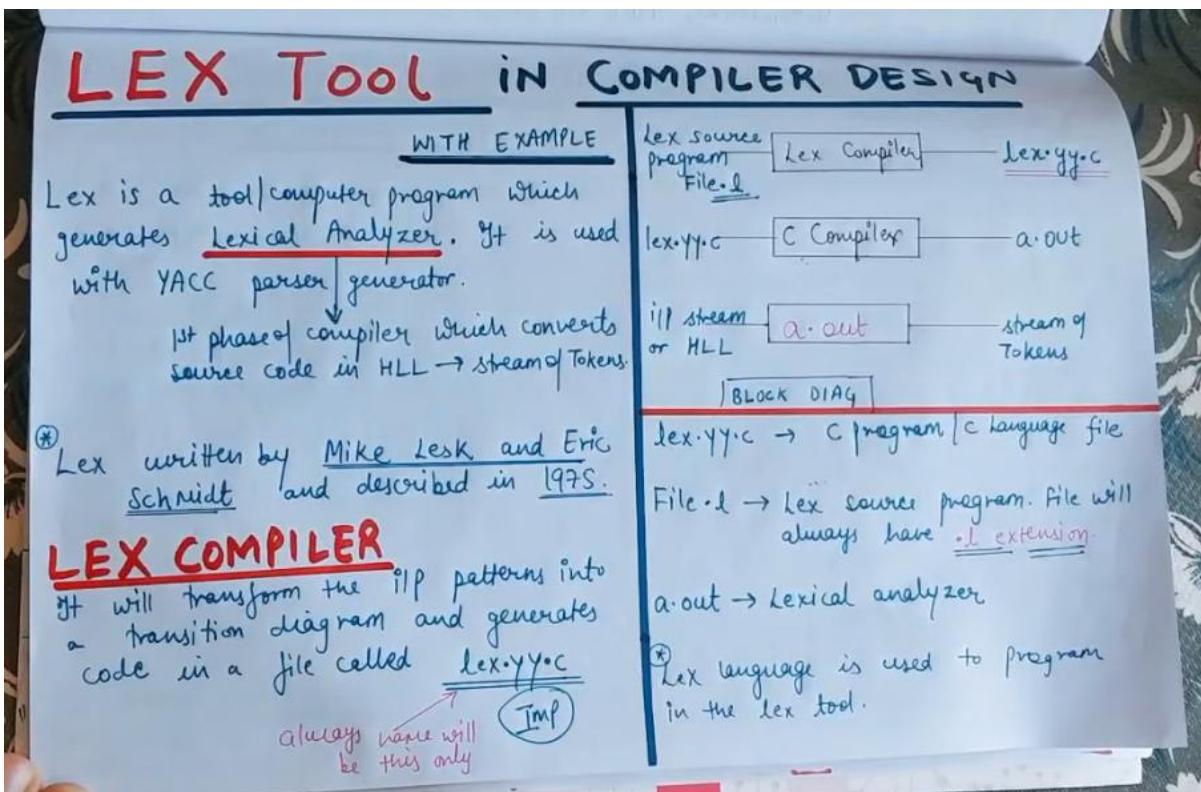
ISRO 2016

- (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 a. Arrays b. Linked Lists ii. Unordered Lists	$O(n)$ $O(n)$ $O(1)$	$O(\log n)$ $O(n)$ $O(n)$	i. For Ordered Lists, every insertion is preceded by lookup operation. ii. Access time is directly proportional to table size.
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

MAY
2020

THURSDAY

Wk 20 • 135-231

M	T	W	T	F	S
4	5	6	7	8	9
11	12	13	14	15	16
18	19	20	21	22	23
25	26	27	28	29	30

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

i.e. lex.yy.c program and

produces an output a.out

Lexical Analyzer

3)

a.out transforms an I/O stream into a sequence / stream of Tokens.

M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

LEX

MAY | 15
2020

FRIDAY

or

LEX file format

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

{ declarations } || declaration of
% % variables,
including file libraries

{ Translation Rules } rules
% % form
contains
regular exp

{ Auxiliary functions }

g) % { #include <stdio.h>
int c = 0;

% } ;

% . % .

pattern { action } pattern { action }

% . % .

main()

{ }

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

14 2020 THURSDAY WK 20 • 135-231	M T W T F S S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
---	---

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.y.c a C program as output.
- 2) C compiler runs this C code ie lex.y.c program and produces an output a.out
- 3) a.out transforms an IP stream into a sequence / stream of Tokens.

Lexical Analyzer

STRUCTURE OF LEX MAY 15 FRIDAY Wk 20 • 136-231
--

or LEX file format

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

```

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

y) { of #include <stdio.h>
 int c=0;
 %.
 pattern fraction } pattern fraction
 %.
 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.

8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

2020 | 00

Wk 19 • 129-237

FRIDAY

yylex() - reads the i/o stream
and generate tokens
acc to the
regular expressions
written in rules section

yytext - pointer to the i/o
string

eg (1) % {

SATURDAY
Wk 19 • 130-236

11 12 13 14 15
18 19 20 21 22
25 26 27 28 29

#include <stdio.h>

% }

% %

"hi" { printf("By"); }

action (will be
in C lang
syntax)

* { printf(" wrong"); }

% %.

```
main()
```

```
{ printf("enter i/p");
```

```
    yyflex(); // Take i/p & generate the  
    tokens acc to patterns in  
    section
```

10 Sunday ↗ int yywrap() // to identify
 { return 1; }

13 19 1/ 19 19 20 21
22 23 24 25 26 27 28
29 30

Wk 20 • 132-234

2020 |

MONDAY

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");
```

```
}
```

* { printf ("WRONG not a no."); }
 % { int yywrap() { return 1; }
 int main() { pf ("enter the IP");
 yylex(); return 0; } }

Now Run it

PLS Like
and

③

Check operator entered is
12 | MAY relational operator or not
2020

TUESDAY

Wk 20 • 133-233

Same declaration as prev ques
of.

">" | "<" | "<=" | ">=" | "=" | "!="

* { printf("wrong"); }

% {

same fns as prev ques

MAY 2020
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

MAY 13
2020
WEDNESDAY

{ printf ("Relational operator
= %s", yytext); }

Introduction to Lexical Analyzer

Lexical Analyzer:

x = a+b*c;



Lexical Analysis

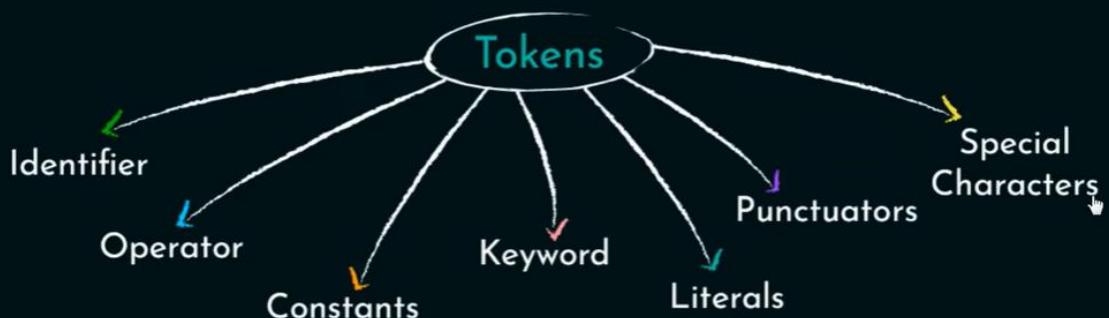


Recognizes tokens using RegExs.
E.g. Regex for identifier:
 $|(+d)^*|_(|+d)^*$
| : letter
d : digit
_ : underscore

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**.

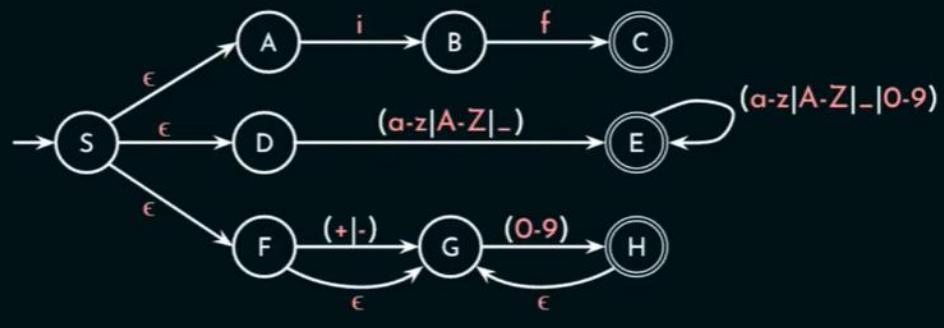


Lexical Analyzer:

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

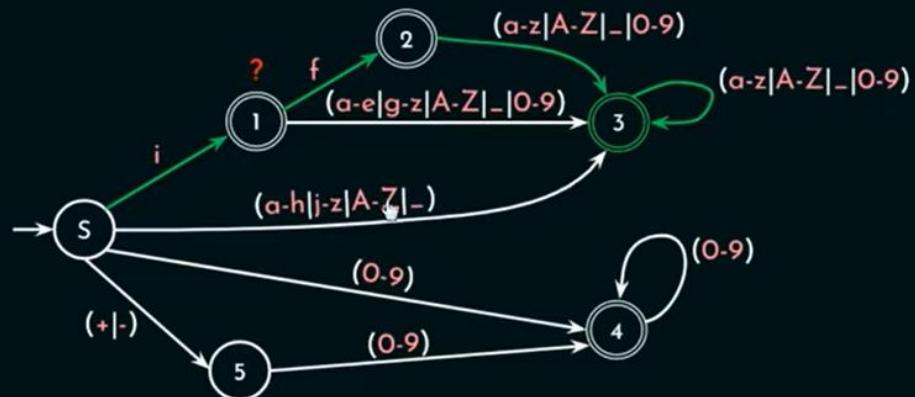


C-Tokens:



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*/
```

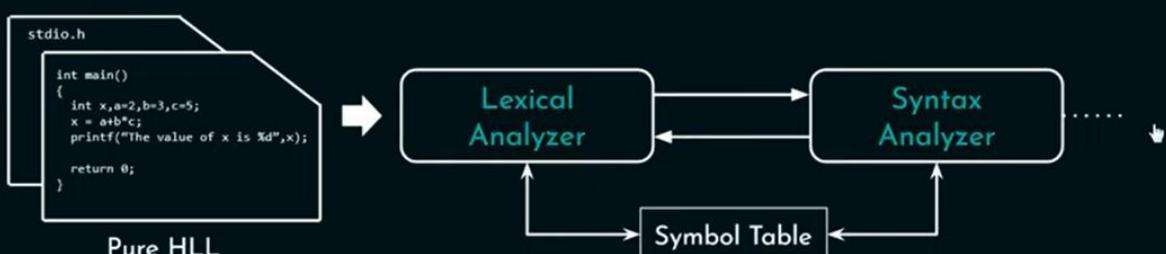
```
int NE/*it's a comment*/SO;
```

```
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-Tokenization:



Tokens:

1. **Keyword:** int , return
2. **Identifier:** main,x,a,b,c,printf
3. **Punctuator:** (,),{,,;,,}
4. **Operator:** =,+,*
5. **Constant:** 2,3,5,0
6. **Literal:** “The value of x is %d”

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

    return 0;
}
```

Count: 39

Lexical Analyzer (Solved Problems) - Set 1

Q1: The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense?

- (A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-Deterministic pushdown automata
(D) Turing Machine

GATE 2011

Q2: The output of a lexical analyzer is

- (A) A parse tree
(B) Intermediate code
(C) Machine code
(D) A stream of tokens

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}.

$$\begin{array}{ll} T_1: a? (b|c)^* a & \\ T_2: b? (a|c)^* b & \\ T_3: c? (b|a)^* c & \end{array}$$

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) T₁T₂T₃
- (B) T₁T₁T₃
- (C) T₂T₁T₃
- (D) T₃T₃

GATE 2018

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

$$\begin{array}{ll} T_1: a? (b|c)^* a & \longrightarrow (b|c)^* a + a(b|c)^* a \\ T_2: b? (a|c)^* b & \longrightarrow (a|c)^* b + b(a|c)^* b \\ T_3: c? (b|a)^* c & \longrightarrow (b|a)^* c + c(b|a)^* c \end{array}$$

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

- (A) T₁T₂T₃
- (B) T₁T₁T₃
- (C) T₂T₁T₃
- (D) T₃T₃

(A) $T_1 T_2 T_3$

$bbaacabc$
 $\underbrace{}_{T_1} \underbrace{}_{T_2} \underbrace{}_{T_3}$



(B) $T_1 T_1 T_3$

$bbaacabc$
 $\underbrace{}_{T_1} \underbrace{}_{T_1} \underbrace{}_{T_3}$



(C) $T_2 T_1 T_3$

$bbaacabc$
 $\underbrace{}_{T_2} \underbrace{}_{T_1} \underbrace{}_{T_3}$



(D) $T_3 T_3$

$bbaacabc$
 $\underbrace{}_{T_3} \underbrace{}_{T_3}$



$bbaacabc$
 $\underbrace{}_{T_1} \underbrace{}_{T_2} \underbrace{}_{T_3}$

$bbaacabc$
 $\underbrace{}_{T_1} \underbrace{}_{T_1} \underbrace{}_{T_3}$

$bbaacabc$
 $\underbrace{}_{T_2} \underbrace{}_{T_1} \underbrace{}_{T_3}$

$bbaacabc$
 $\underbrace{}_{T_3} \underbrace{}_{T_3}$

- (A) $T_1 T_2 T_3$ (B) $T_1 T_1 T_3$ (C) $T_2 T_1 T_3$ (D) $T_3 T_3$



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

□ Function computed from the syntax tree

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

□ Rules to compute nullable, firstpos, lastpos

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

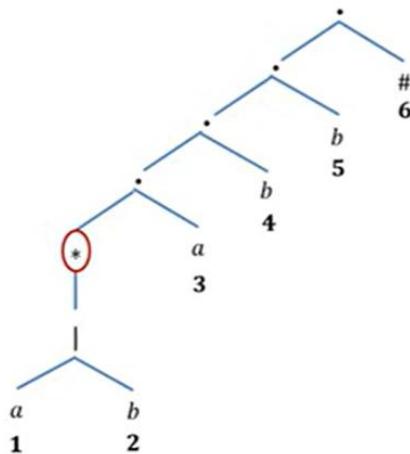


□ Rules to compute followpos

1. If n is **concatenation** node with left child c_1 and right child c_2 and i is a position in $\text{lastpos}(c_1)$, then all position in $\text{firstpos}(c_2)$ are in $\text{followpos}(i)$
2. If n is ***** node and i is position in $\text{lastpos}(n)$, then all position in $\text{firstpos}(n)$ are in $\text{followpos}(i)$

□ Conversion from regular expression to DFA without constructing NFA

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



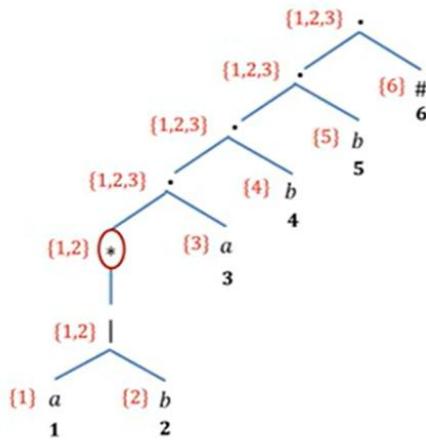
Step 1: Construct Syntax Tree

Step 2: Nullable node

Here, ***** is only nullable node

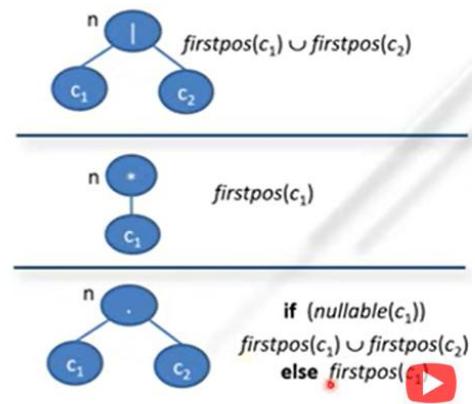
□ Conversion from regular expression to DFA without constructing NFA

Step 3: Calculate firstpos



Firstpos —

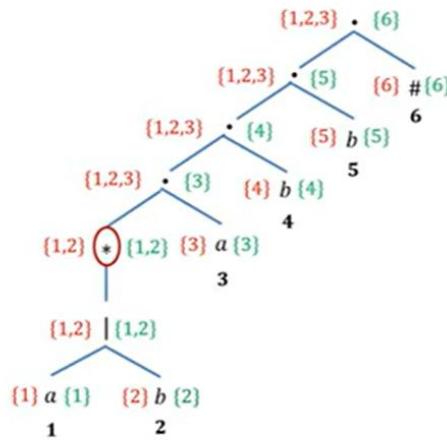
A leaf with position $i = \{l\}$



SUBSCRIBE

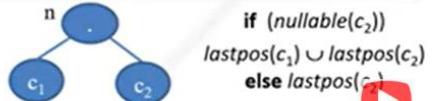
□ Conversion from regular expression to DFA without constructing NFA

Step 3: Calculate lastpos



Lastpos —

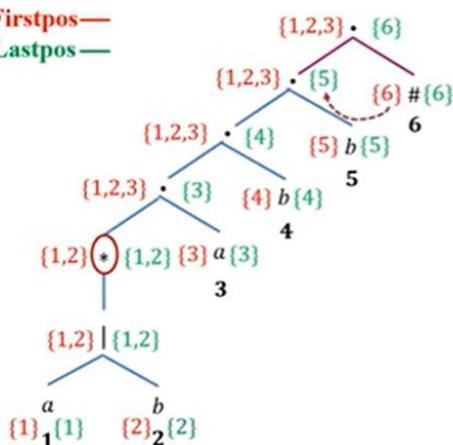
A leaf with position $i = \{l\}$



□ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos

Firstpos—
Lastpos —



Position	followpos
5	6



{1,2,3} c₁ {5} {6} c₂ {6}

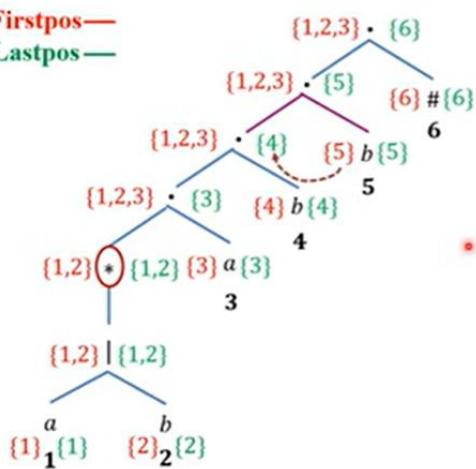
$i = \text{lastpos}(c_1) = \{5\}$
 $\text{firstpos}(c_2) = \{6\}$
 $\text{followpos}(5) = \{6\}$



□ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos

Firstpos—
Lastpos—



Position	followpos
5	6
4	5

$$\{1,2,3\} c_1 \{4\} \{5\} c_2 \{5\}$$

$$i = \text{lastpos}(c_1) = \{4\}$$

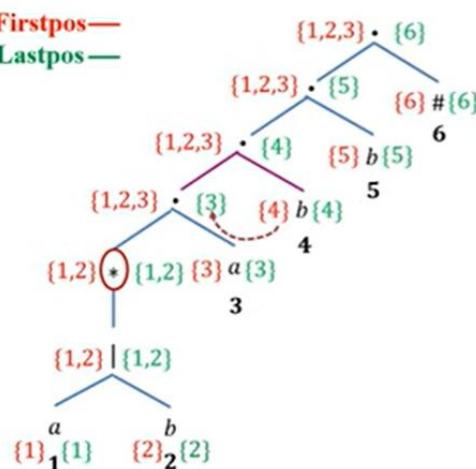
$$\text{firstpos}(c_2) = \{5\}$$

$$\text{followpos}(4) = \{5\}$$

□ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos

Firstpos—
Lastpos—



Position	followpos
5	6
4	5

$$\{1,2,3\} c_1 \{3\} \{4\} c_2 \{4\}$$

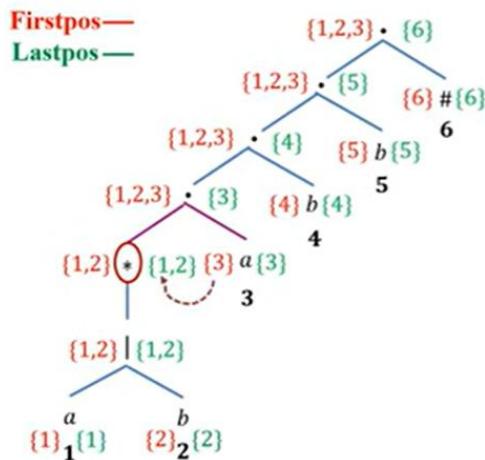
$$i = \text{lastpos}(c_1) = \{3\}$$

$$\text{firstpos}(c_2) = \{4\}$$

$$\text{followpos}(3) = \{4\}$$

□ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos



Position	followpos
5	6
4	5
3	4

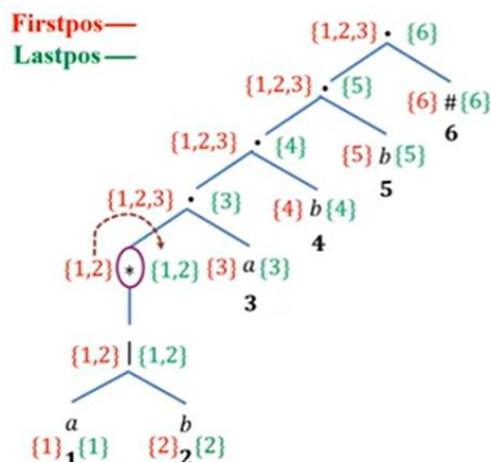
{1,2} c₁ {1,2} {3} c₂ {3}

$$\begin{aligned} i &= \text{lastpos}(c_1) = \{1,2\} \\ \text{firstpos}(c_2) &= \{3\} \\ \text{followpos}(1) &= \{3\} \\ \text{followpos}(2) &= \{3\} \end{aligned}$$



□ Conversion from regular expression to DFA without constructing NFA

Step 4: Calculate followpos



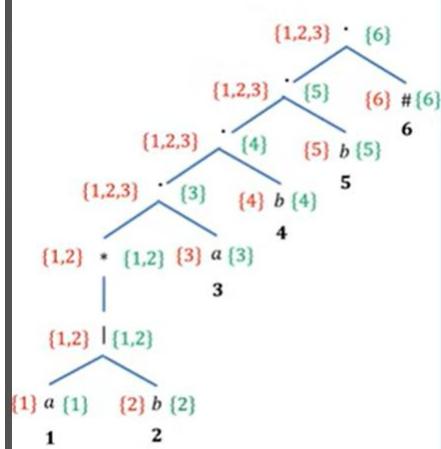
Position	followpos
6	--
5	6
4	5
3	4
2	1,2,3
1	1,2,3

{1,2} * {1,2}
n

$$\begin{aligned} i &= \text{lastpos}(n) = \{1,2\} \\ \text{firstpos}(n) &= \{1,2\} \\ \text{followpos}(1) &= \{1,2\} \\ \text{followpos}(2) &= \{1,2\} \end{aligned}$$



□ Constructing DFA



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

State A

$\delta(A, a) = \text{followpos}(1) \cup \text{followpos}(3)$
 $= \{1,2,3\} \cup \{4\} = \{1,2,3,4\}$ ----- B

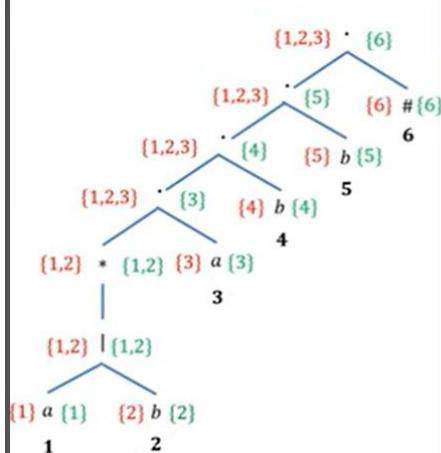
$\delta(A, b) = \text{followpos}(2)$
 $= \{1,2,3\}$ ----- A

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

$\delta(B, a) = \text{followpos}(1) \cup \text{followpos}(3)$
 $= \{1,2,3\} \cup \{4\} = \{1,2,3,4\}$ ----- B

$\delta(B, b) = \text{followpos}(2) \cup \text{followpos}(4)$
 $= \{1,2,3\} \cup \{5\} = \{1,2,3,5\}$ ----- C

State C

$\delta(C, a) = \text{followpos}(1) \cup \text{followpos}(3)$
 $= \{1,2,3\} \cup \{4\} = \{1,2,3,4\}$ ----- B

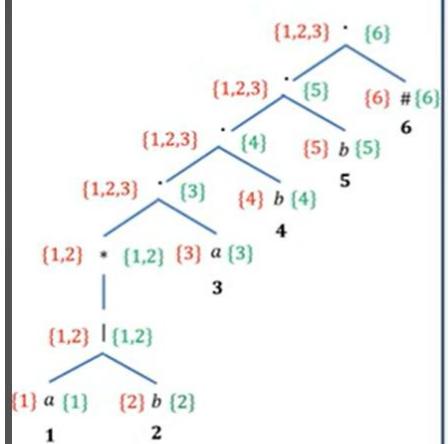
$\delta(C, b) = \text{followpos}(2) \cup \text{followpos}(5)$
 $= \{1,2,3\} \cup \{6\} = \{1,2,3,6\}$ ----- D

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}		



□ Constructing DFA



State D

$$\delta(D,a) = \text{followpos}(1) \cup \text{followpos}(3) \\ = (1,2,3) \cup (4) = \{1,2,3,4\} \longrightarrow B$$

$$\delta(D,b) = \text{followpos}(2) \\ = (1,2,3) \longrightarrow A$$

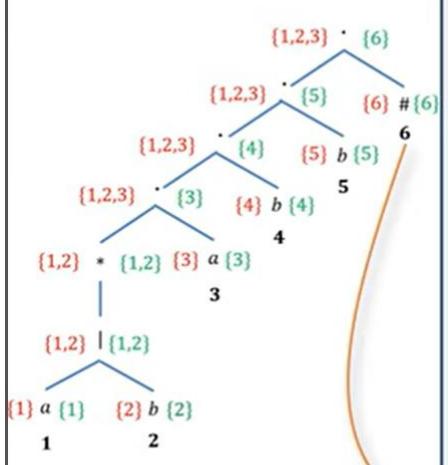
Position	followpos
6	--
5	{6}
4	{5}
3	{4}
2	{1,2,3}
1	{1,2,3}

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

Transition Table for DFA



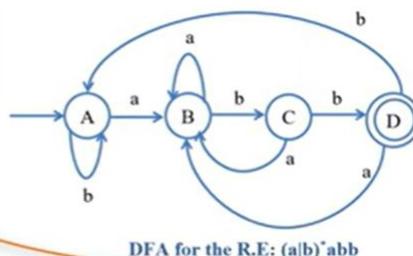
□ Constructing DFA



State D

$$\delta(D,a) = \text{followpos}(1) \cup \text{followpos}(3) \\ = (1,2,3) \cup (4) = \{1,2,3,4\} \longrightarrow B$$

$$\delta(D,b) = \text{followpos}(2) \\ = (1,2,3) \longrightarrow A$$



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}	B	C
C={1,2,3,5}	B	D
D={1,2,3,6}	B	A

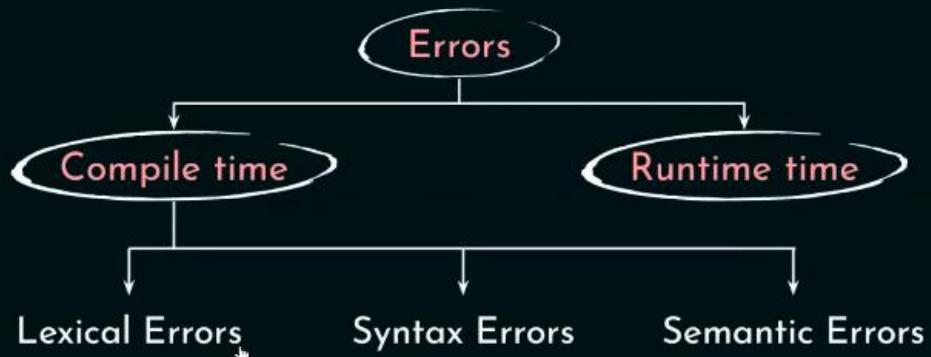
Transition Table for DFA



Errors and Error Recovery in Lexical Analysis

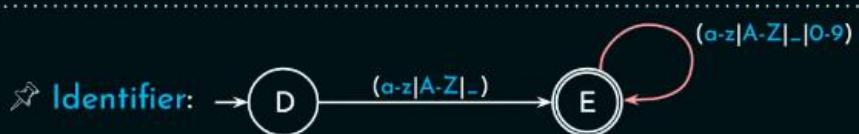
<https://www.geeksforgeeks.org/error-recovery-strategies-in-compiler-design/>

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.

```
unpin 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

Apowersoft Video Converter

Input Buffering

lexical errors

- Reading the source program for lexical analysis is a difficult task.
- One need to look one or many characters beyond lexemes to recognize the end of it.
- E.g.: In C, single character operators like - , = or < can be beginning of two character operators like ->, ==, or <=.

Apowersoft Video Converter

Input Buffering

- Input character is read from the secondary storage at first. Which is costly in terms of timing and efficiency. That is where buffering is used.

lexemeBegin
↓
E = M *
forward

Single Buffer Input Scheme

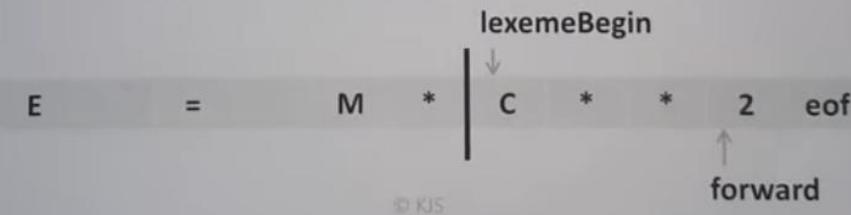
© KJS

Single Buffer

- **Problem** with single bufferinput scheme is, it can not work with long lexemes which crosses the **buffer boundary**.
3.2. length
- For that **buffer** has to be **refilled** and first part of **lexeme** has to be **over written**.

Buffer Pairs

- Specialized input techniques have been developed to reduce character processing time during lexical analysis.
- Such important technique involving two buffers that are alternately reloaded is buffer pairs.



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

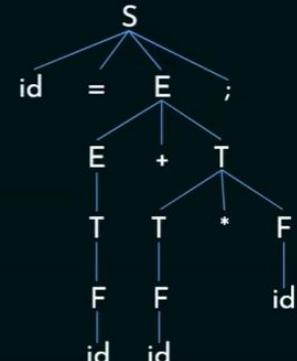
Introduction to Formal Grammars

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

Syntax Analyzer:

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

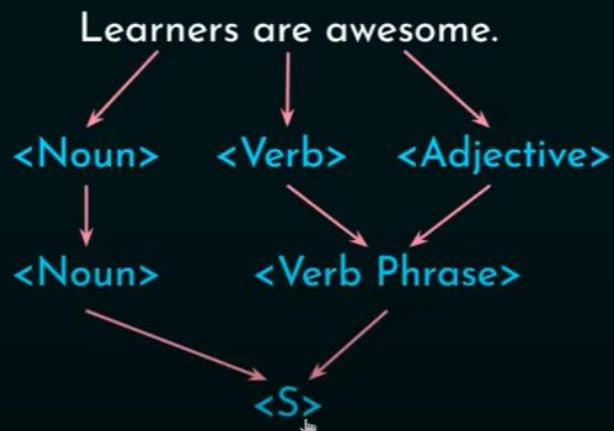
id = id + id * id ;



Syntax Analysis

$S \rightarrow id = E ;$
 $E \rightarrow E + T | T$
 $T \rightarrow T * F | F$
 $F \rightarrow id$

Grammar:



Grammar:



Variables
or
Non-terminals



Constants
or
Terminals

Grammar:

(N , T , P , S)

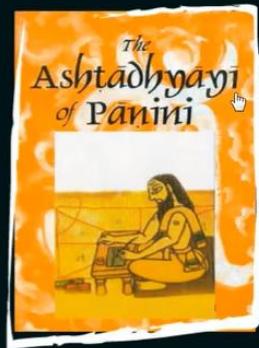
$\langle S \rangle \rightarrow \langle \text{Noun} \rangle \langle \text{Verb Phrase} \rangle$
 $\langle \text{Verb Phrase} \rangle \rightarrow \langle \text{Verb} \rangle \langle \text{Adjective} \rangle$
 $\langle \text{Noun} \rangle \rightarrow \text{Learners} \mid \text{Neso}$
 $\langle \text{Verb} \rangle \rightarrow \text{are} \mid \text{is}$
 $\langle \text{Adjective} \rangle \rightarrow \text{awesome} \mid \text{good}$



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. $N \cap T = \emptyset$
- 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. $N \cap T = \emptyset$
- 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,

$\alpha \rightarrow \beta$

Grammar:

1. Type-0 / Unrestricted Grammar:

$$\alpha \rightarrow \beta$$

$$V^* = (N \cup T)^*$$



$$V^* N V^* \rightarrow V^*$$

$$\alpha \in V^* N V^*$$

$$\beta \in V^*$$

G:

$$P: abABcde \rightarrow abXcde$$

$$AB \Rightarrow X$$

Grammar:

2. Type-1:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

$$|\alpha A \beta| \leq |\alpha \gamma \beta|$$

$$A \in N$$

$$\alpha, \beta, \gamma \in V^*$$

$$\text{Type-0: } V^* N V^* \rightarrow V^*$$

G:

$$P: abABcde \rightarrow abXcde$$

$$V^* N V^* \rightarrow V^*$$

↓ ↓ ↓ ↓
 $\alpha A \beta \rightarrow \overbrace{\alpha \gamma \beta}$

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 → precedence of operators is
eg. $E \rightarrow E + E \mid E * E \mid id$ violated / not respected

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

we have to derive $id + id * id$

LMD: $E \rightarrow E + E$
→ $id + E$
→ $id + E * E$
→ $id + id * E$
→ $id + id * id$

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

RMD.

$$\begin{array}{l} | \quad E \rightarrow E + E \\ | \quad E \rightarrow E + \underline{E * E} \\ \rightarrow E + E * id \\ \rightarrow E + id * id \\ E \rightarrow id + id * id \end{array}$$

RMD:

$$\begin{array}{l} E \rightarrow E * E \\ E \rightarrow E * id \\ E \rightarrow \underline{E + E * id} \\ E \rightarrow \underline{E + id * id} \\ E \rightarrow id + id * id \end{array}$$

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.

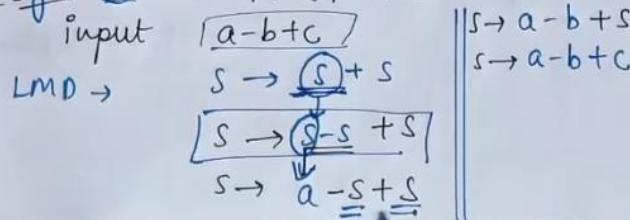
2 options for this

Left most Derivation

Right most Derivation

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

$$\text{eg } S \rightarrow S+S \mid S-S \mid a/b/c$$



Right most Derivation

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

$$\text{eg) } S \rightarrow S+S \mid S-S \mid a/b/c$$

same eg as prev.

RMD will be

$$\begin{aligned} S &\rightarrow S-S \\ S &\rightarrow S-S+S \\ S &\rightarrow S-S+c \\ S &\rightarrow S-b+c \\ S &\rightarrow a-b+c \end{aligned}$$

PARSE TREE

It is a typical depiction of how the start symbol of a grammar 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

$$\text{eg } S \rightarrow XYZ$$

$$\begin{aligned} X &\rightarrow a \\ Y &\rightarrow b \\ Z &\rightarrow c/d \end{aligned}$$

$$L = \{ \underline{abc}, abd \}$$

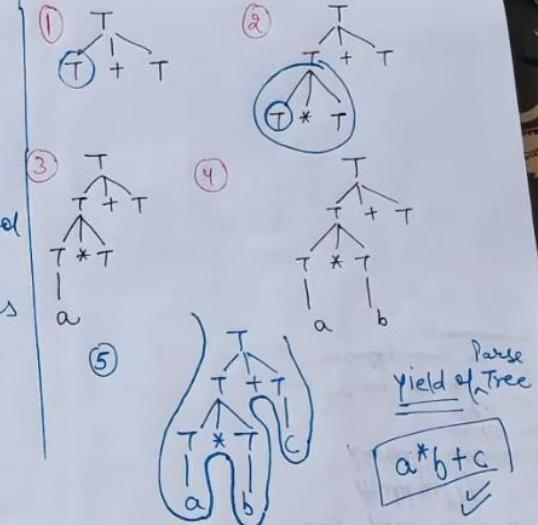


RIGHT MOST DERIVATION

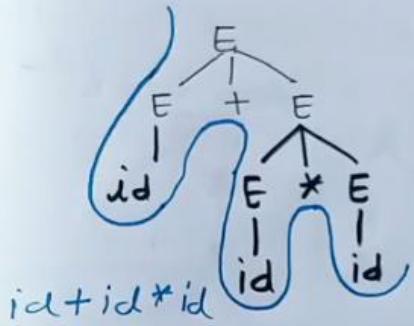
$$\begin{array}{l} \text{eg } T \rightarrow T+T \mid T*T \\ T \rightarrow a/b/c \end{array}$$

input $a*b+c$

$$(5 \times 2) + 3 \\ 10 + 3 = 13$$



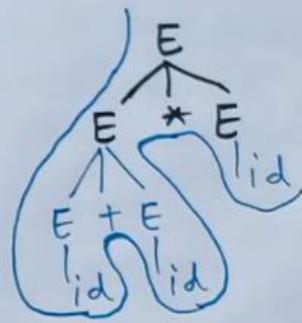
Parse Tree 1 (is valid)



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".

Parse Tree -2



$id + id * id$

5 id+id*id

So, since we have more than 1 parse tree possible
∴ Grammar 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 diff

e.g. if we have O/P $1 + 5 * 3$
our answer should be = 16

but parse tree 1 generates 16

and parse tree 2 generates 18

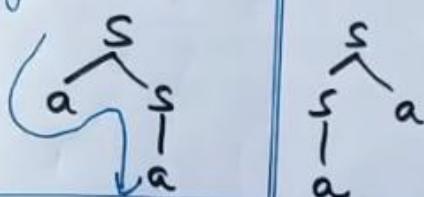
so parsers don't allow ambiguous grammar
except of one

operator precedence

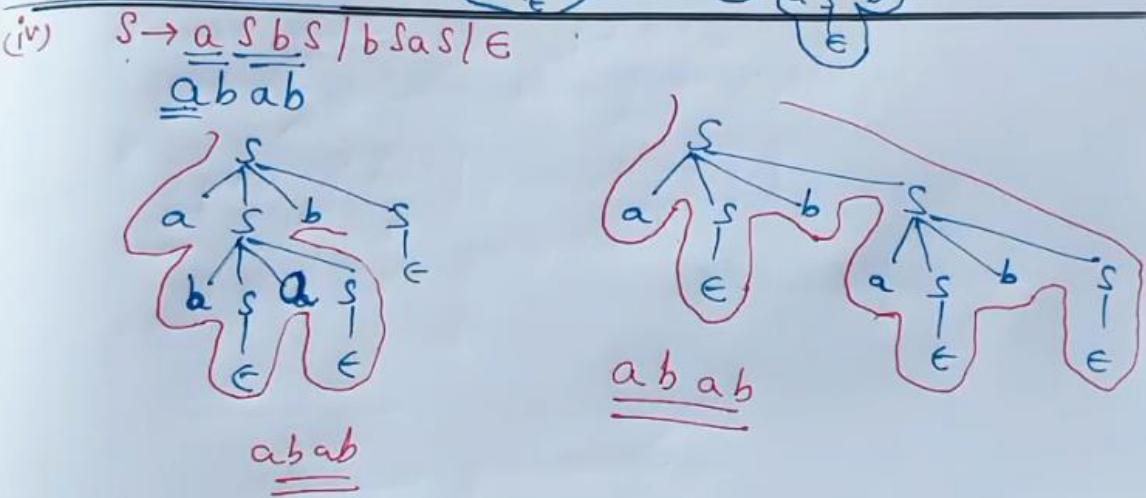
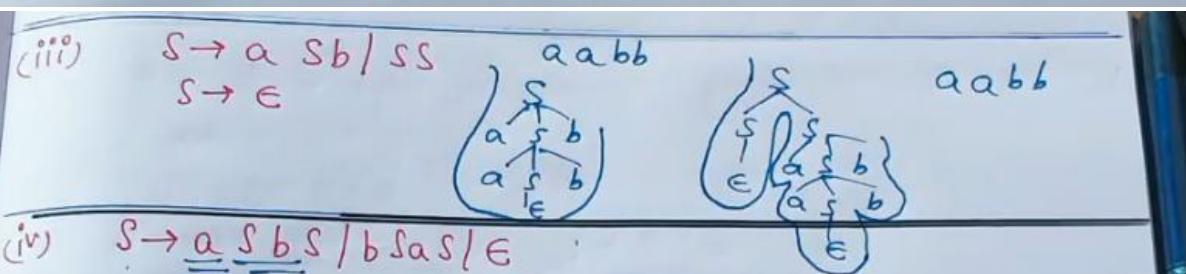
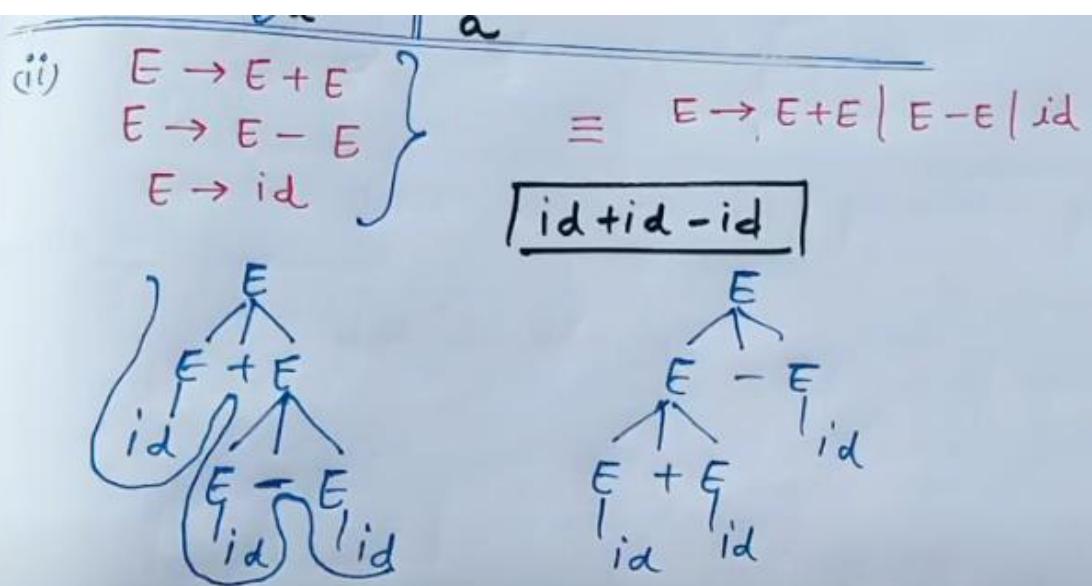
Ques Check grammar is ambiguous or not?

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

string - aa



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



Context Free Grammar:

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

id + id × id

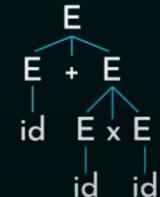
Left Most Derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E \times E \\ &\Rightarrow id + id \times E \\ &\Rightarrow id + id \times id \end{aligned}$$

Right Most Derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E \times E \\ &\Rightarrow E + E \times id \\ &\Rightarrow E + id \times id \\ &\Rightarrow id + id \times id \end{aligned}$$

Parse Tree Derivation:



Context Free Grammar:

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

id + id × id

Left Most Derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E \times E \\ &\Rightarrow id + id \times E \\ &\Rightarrow id + id \times id \end{aligned}$$

Right Most Derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E \times E \\ &\Rightarrow E + E \times id \\ &\Rightarrow E + id \times id \\ &\Rightarrow id + id \times id \end{aligned}$$

Parse Tree Derivation:



$$\begin{aligned} E &\Rightarrow E \times E \\ &\Rightarrow E + E \times E \\ &\Rightarrow id + E \times E \\ &\Rightarrow id + id \times E \\ &\Rightarrow id + id \times id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E \times E \\ &\Rightarrow E \times id \\ &\Rightarrow E + E \times id \\ &\Rightarrow E + id \times id \\ &\Rightarrow id + id \times id \end{aligned}$$



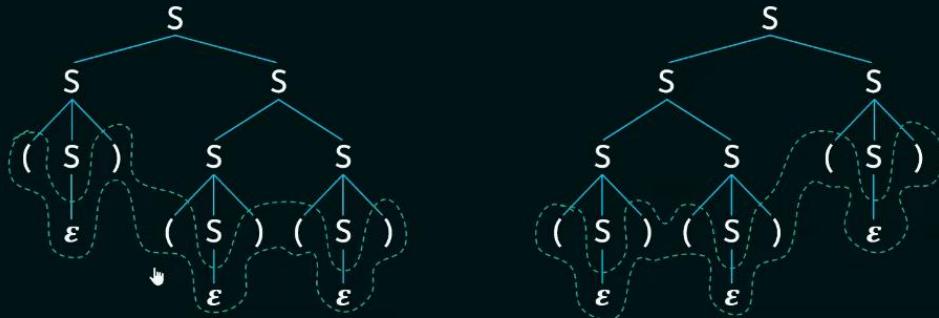
Ambiguity in CFGs - Solved Problems (Set 1)

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

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

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,

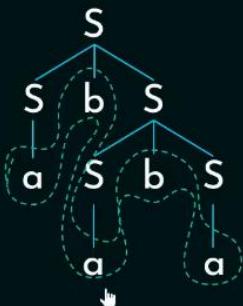
$$G1: S \rightarrow SbS \mid a$$

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

Which of the following option is correct?

Sol. G1: $S \rightarrow SbS \mid a$

ababa



Q2: Consider the following two Grammars,

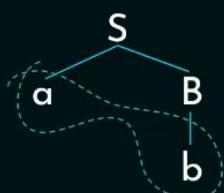
$$G1: S \rightarrow SbS \mid a$$

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

Which of the following option is correct?

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

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



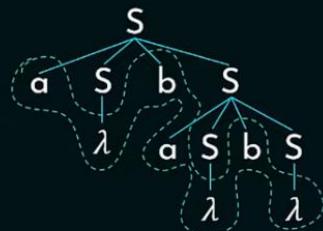
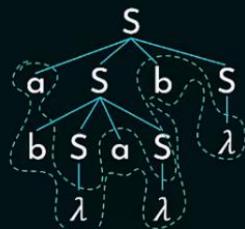
Q: Which of the following grammars is (are) ambiguous?

- ✓ (A) $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
 (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. (B) $S \rightarrow aSbS + bSaS + \lambda$

abab



nesoacademy.org

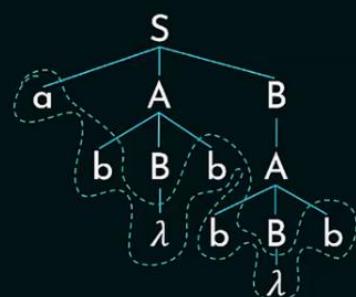
Q: Which of the following grammars is (are) ambiguous?

- ✓(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. (C) $S \rightarrow aAB, A \rightarrow bBb, B \rightarrow A \mid \lambda$

abbbb



nesoacademy.org

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.

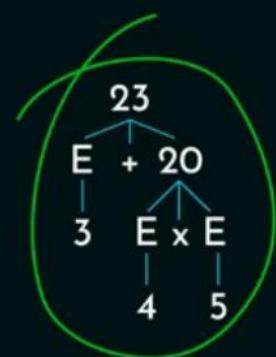
- 1. (A) and (C) only
- 2. (B) only
- 3. (B) and (C) only
- 4. (A) and (B) only

Problems of Ambiguity in CFGs

Context Free Grammar:

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

3 + 4 × 5

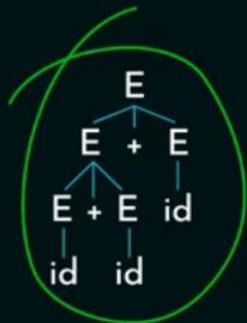


Context Free Grammar:

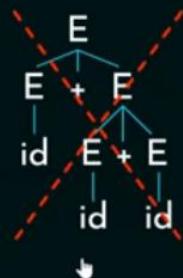
1. $E \rightarrow E + E$
2. $E \rightarrow E \times E$
3. $E \rightarrow id$

$id + id + id$

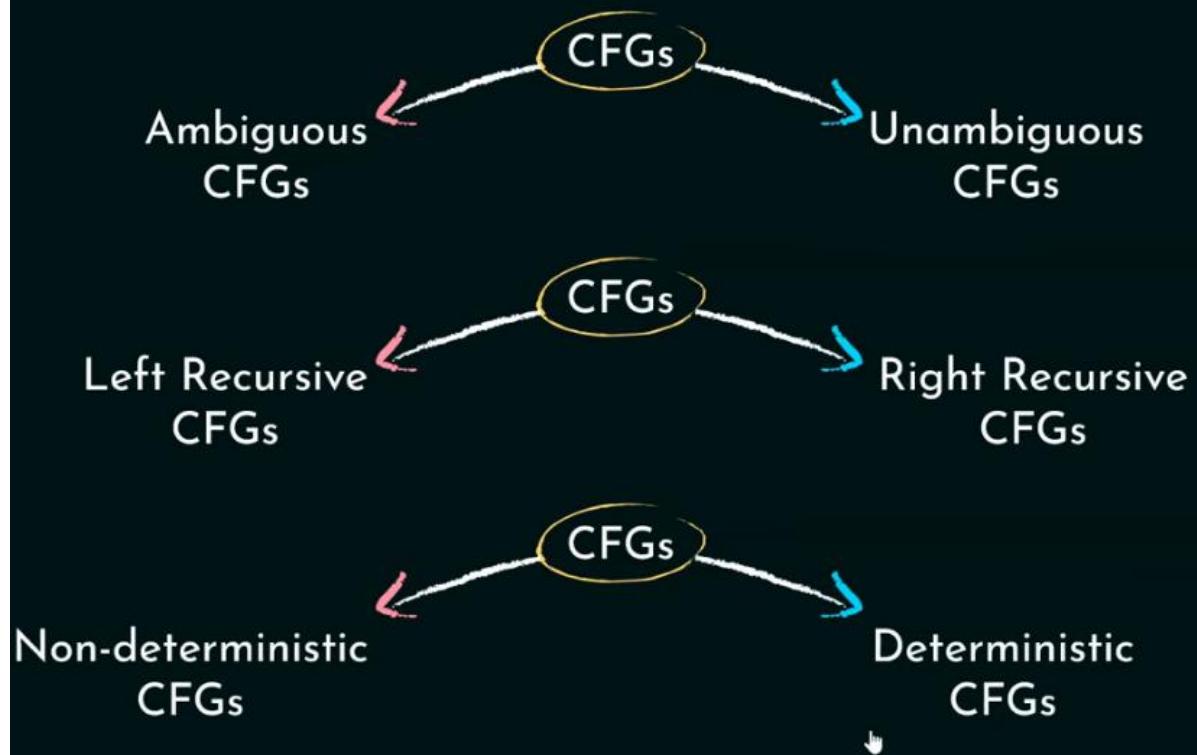
$(id + id) + id$

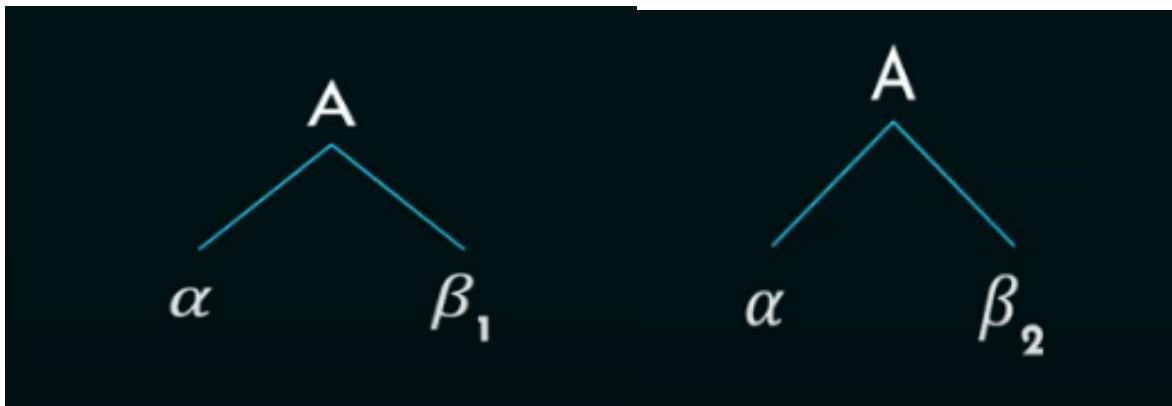


Associativity
Property
Violation



Non-Deterministic CFGs

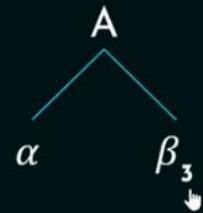
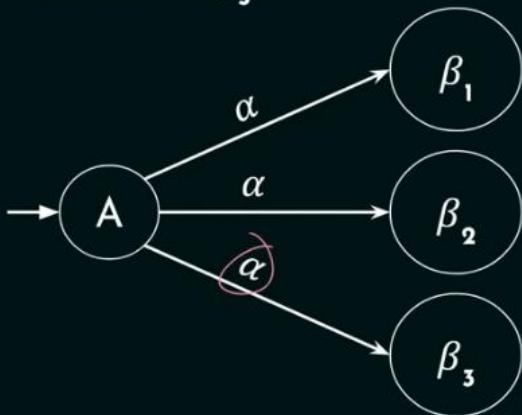




Non-deterministic CFGs:

1. $A \rightarrow \alpha\beta_1$
2. $A \rightarrow \alpha\beta_2$
3. $A \rightarrow \alpha\beta_3$

$\boxed{\alpha\beta_3}$



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:

1. $A \rightarrow \alpha\beta_1$
2. $A \rightarrow \alpha\beta_2$
3. $A \rightarrow \alpha\beta_3$

$$\boxed{A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3}$$

Left Factoring



Non-deterministic CFGs – Solved Problems (Set 1)

Eliminate non-determinism:

Ex1: $A \rightarrow \alpha AB \mid \alpha Bc \mid \alpha Ac$
 $\downarrow \quad \downarrow \quad \downarrow$
 $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \alpha \beta_3$

$$\boxed{A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3}$$

$$\boxed{A \rightarrow \alpha A' \\ A' \rightarrow AB \mid Bc \mid Ac}$$

Ex1: $A \rightarrow \alpha AB \mid \alpha Bc \mid \alpha Ac$

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow AB \mid Ac \mid Bc \\ A &\rightarrow \alpha \beta_1 \mid \alpha \beta_2 \end{aligned}$$

$$\boxed{A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3}$$

$$\begin{aligned} A' &\rightarrow AA'' \mid Bc \\ A'' &\rightarrow B \mid c \end{aligned}$$

$$\begin{aligned}
 A &\rightarrow aA' \\
 A' &\rightarrow AA'' \mid Bc \\
 A'' &\rightarrow B \mid c
 \end{aligned}$$

Ex2: $S \rightarrow iEtS \mid iEtSeS \mid a$
 $E \rightarrow b$

$$\begin{aligned}
 S &\rightarrow iEtSe\epsilon \mid iEtSeS \mid a \\
 A &\rightarrow \alpha \beta_1 \mid \alpha \beta_2
 \end{aligned}$$

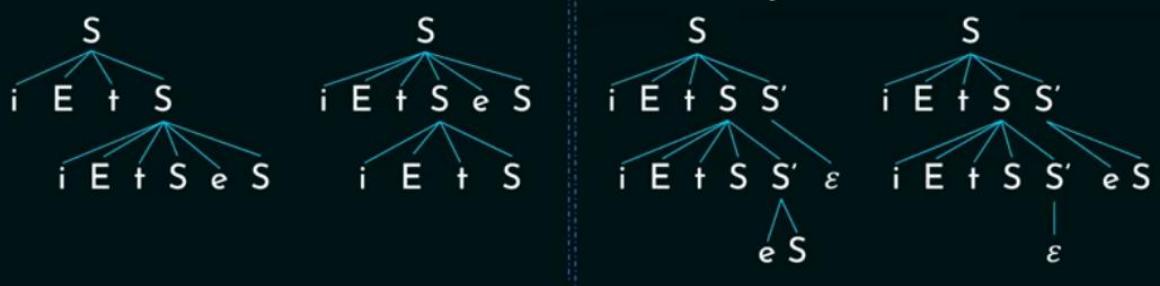
$$\begin{aligned}
 A &\rightarrow \alpha A' \\
 A' &\rightarrow \beta_1 \mid \beta_2 \mid \beta_3
 \end{aligned}$$

$$\begin{aligned}
 S &\rightarrow iEtSS' \mid a \\
 S' &\rightarrow \epsilon \mid eS \\
 E &\rightarrow b
 \end{aligned}$$

Determinism vs Ambiguity:

$$\begin{aligned}
 S &\rightarrow iEtS \mid iEtSeS \mid a \\
 E &\rightarrow b
 \end{aligned}$$

iEt*iEtSeS*

$$\begin{aligned}
 S &\rightarrow iEtSS' \mid a \\
 S' &\rightarrow \epsilon \mid eS \\
 E &\rightarrow b
 \end{aligned}$$


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

Reason for Ambiguity:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$



iEtSeS

```
if(E)
{
    if(E)
        { S }
    else
        { S }
}
```

Non-deterministic CFGs – Solved Problems (Set 2)

Ex1: $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

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

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

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SSbS \mid SaSb \mid bb$$

Ex1: $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SSbS \mid SaSb \mid bb$$

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

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

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SS'' \mid bb$$

$$S'' \rightarrow SbS \mid aSb$$

Ex2: $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$
 $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \alpha \beta_3$

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

$S \rightarrow bSS' \mid a$
 $S' \rightarrow SaaS \mid SaSb \mid b$

Ex2: $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$
 $S \rightarrow bSS' \mid a$
 $S' \rightarrow SaaS \mid SaSb \mid b$
 $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$

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

$S \rightarrow bSS' \mid a$
 $S' \rightarrow SaS'' \mid b$
 $S'' \rightarrow aS \mid Sb$

Ex3: $S \rightarrow a \mid ab \mid abc \mid abcd$
 $S \rightarrow aS'$
 $S' \rightarrow bS'' \mid \epsilon$
 $S'' \rightarrow c\epsilon \mid cd \mid \epsilon$
 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

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

$S \rightarrow aS'$
 $S' \rightarrow bS'' \mid \epsilon$
 $S'' \rightarrow cS''' \mid \epsilon$
 $S''' \rightarrow d \mid \epsilon$

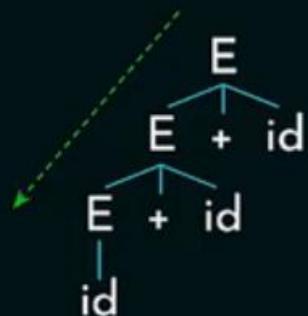
Associativity Violation and Solution in CFGs

Context Free Grammar:

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

$id + id + id$

Left Recursive



$(id + id) + id$

Precedence Violation and Solution in CFGs

Context Free Grammar:

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

$id + id \times id$



Precedence
Property
Violation



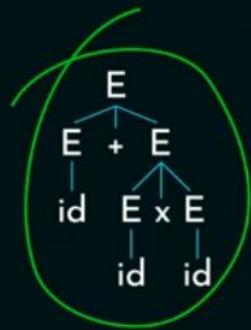
Context Free Grammar:

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

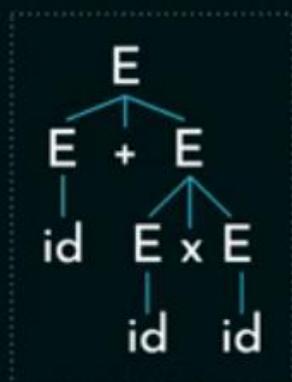
id + id x id



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. $F \rightarrow id$ ↘

Context Free Grammar:

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

id + id × id

Defining Levels



Associativity and Precedence in CFGs

Context Free Grammar:

1. $E \rightarrow E + T \mid T$
2. $T \rightarrow T \times F \mid F$
3. $F \rightarrow G^{\wedge} \mid G$
4. $G \rightarrow id$

Determine Associativity and Precedence of the operators:



1. $E \rightarrow E + T \mid T$
2. $T \rightarrow T \times F \mid F$
3. $F \rightarrow G^{\wedge} \mid G$
4. $G \rightarrow id$

► Precedence: $\wedge \quad x \quad +$
High Low

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 \wedge is right associative.

As \wedge is occurring on the lowest level hence it has the highest precedence, then x and the +.

Context Free Grammar:

Consider the following ambiguous CFG on boolean expressions:

$bExp \rightarrow bExp \text{ AND } bExp \mid bExp \text{ OR } bExp \mid \text{NOT } bExp \mid \text{True} \mid \text{False}$

The precedence of the boolean operators are NOT, AND, OR (high to low). AND, OR have left to right associativity.

1. $bExp \rightarrow bExp \text{ OR } bExp1 \mid bExp1$
2. $bExp1 \rightarrow bExp1 \text{ AND } bExp2 \mid bExp2$
3. $bExp2 \rightarrow \text{NOT } bExp2$
4. $bExp2 \rightarrow \text{True}$
5. $bExp2 \rightarrow \text{False}$

↓

1. $bExp \rightarrow bExp \text{ OR } bExp1 \mid bExp1$
2. $bExp1 \rightarrow bExp1 \text{ AND } bExp2 \mid bExp2$
3. $bExp2 \rightarrow \text{NOT } bExp2 \mid \text{True} \mid \text{False}$

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

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

$$\begin{aligned}S &\rightarrow T \times P \\T &\rightarrow U \mid T \times U \\P &\rightarrow Q + P \mid Q \\Q &\rightarrow \text{id} \\U &\rightarrow \text{id}\end{aligned}$$

GATE
2014

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

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

$$\begin{array}{ll}
 S \rightarrow T \times P & \\
 T \rightarrow U \mid T \times U & \text{Left recursive} \\
 P \rightarrow Q + P \mid Q & \\
 Q \rightarrow \text{id} & \\
 U \rightarrow \text{id} &
 \end{array}
 \quad
 \begin{array}{ll}
 T \rightarrow \textcolor{red}{T} \times U & \\
 P \rightarrow Q + \textcolor{blue}{P} & \text{Right recursive} \\
 \end{array}$$

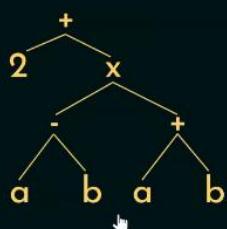
► Precedence: $\begin{array}{c} + \qquad \qquad \times \\ \parallel \qquad \qquad \parallel \\ \text{High} \qquad \qquad \text{Low} \end{array}$

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) $2 + a - b \times a + b$
- (iii) $(2 + ((a - b) \times (a + b)))$
- (iv) $2 + (a - b) \times (a + b)$

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

$2 + ((a - b) \times (a + b))$

a b a b

Consider

- (i) $2 + a - b$
- (ii) $2 + a - b \times a + b$
- (iii) $(2 + ((a - b) \times (a + b)))$
- (iv) $2 + (a - b) \times (a + b)$

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:

$$\begin{array}{l} E \rightarrow E \times F \mid F + E \mid F \\ F \rightarrow F - F \mid id \end{array}$$

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

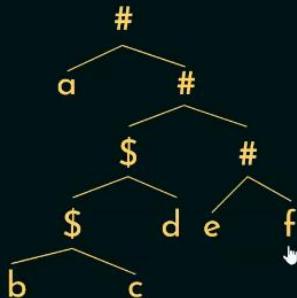
(D) '+' has higher precedence than 'x'

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)

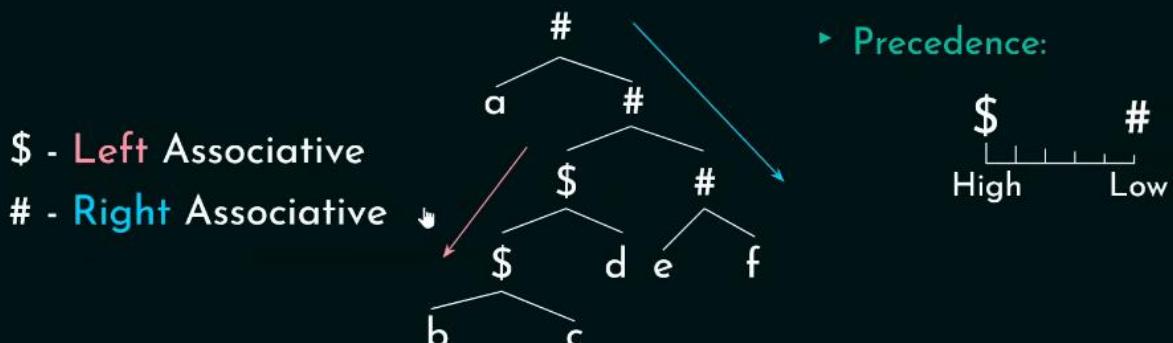
Q1: Consider the following parse tree for the expression $a\#b\$c\$d\#e\#f$, involving two binary operators $\$$ and $\#$.

GATE 2018



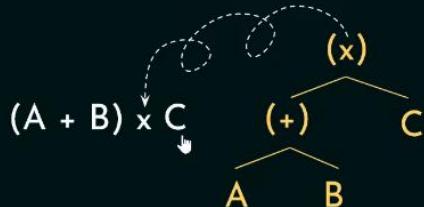
- (A) ' $\$$ ' has higher precedence and is left associative; ' $\#$ ' is right associative
- (B) ' $\#$ ' has higher precedence and is left associative; ' $\$$ ' is right associative
- (C) ' $\$$ ' has higher precedence and is left associative; ' $\#$ ' is left associative
- (D) ' $\#$ ' has higher precedence and is right associative; ' $\#$ ' is left associative

Involving two binary operators $\$$ and $\#$.



- (A) ' $\$$ ' has higher precedence and is left associative; ' $\#$ ' is right associative

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



UGC-NET
JUN 2010

- (A) $(A + B) \times C$
- (B) $A + x BC$
- (C) $A + B \times C$
- (D) $A \times C + B$

(A) $(A + B) \times C$

Q3: Given the grammar:

$$\begin{array}{ll} S \rightarrow T \times S \mid T & S \rightarrow T \times \overset{\circ}{S} \\ T \rightarrow U + T \mid U & T \rightarrow U + \overset{\circ}{T} \\ U \rightarrow \text{id} & \end{array}$$

Right recursive

► Precedence:
 $\begin{array}{c} + \\ \text{High} \quad \quad \quad \times \\ \text{Low} \end{array}$

ISRO CS
2020

Which of the following statements is wrong?

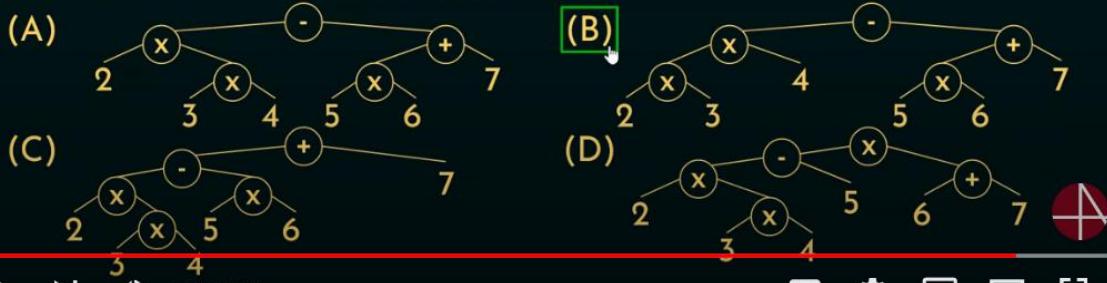
- (A) Grammar is not ambiguous
- (B) Priority of '+' over 'x' is ensured
- (C) Right to left evaluation of 'x' and '+' happens
- (D) None of these**

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

$$\begin{aligned} E &\rightarrow T - E \quad \text{Right recursive} \\ T &\rightarrow T \times F \quad \text{Left recursive} \\ T &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

TIFR PHD CS 2015

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



Recursion in Context Free Grammars (CFGs)

Left Recursion

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

$$A \mid \beta$$

$$\begin{array}{c} A \\ | \\ A \quad \alpha \\ | \\ \beta \end{array}$$

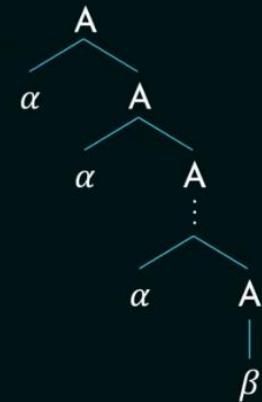
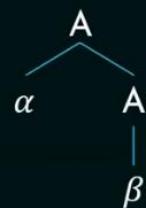
$$A \Rightarrow \beta\alpha^*$$

$$\begin{array}{c} A \\ | \\ A \quad \alpha \\ | \\ A \quad \alpha \\ | \\ \vdots \\ | \\ A \quad \alpha \\ | \\ \beta \end{array}$$

Right Recursion

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

A
|
 β



$$A \Rightarrow \alpha^* \beta$$

esoacademy.org



Left Recursion

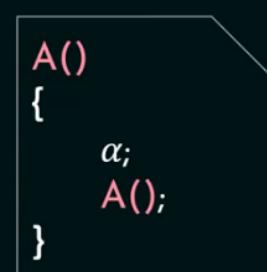
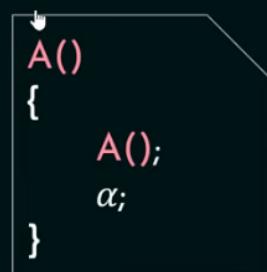
$$A \rightarrow A\alpha \mid \beta$$

$$A \Rightarrow \beta\alpha^*$$

Right Recursion

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

$$A \Rightarrow \alpha^* \beta$$

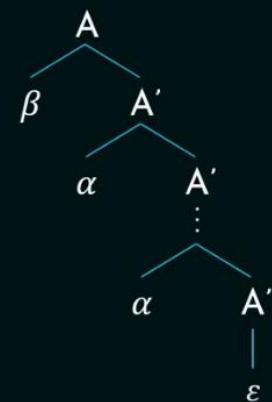
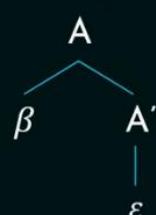


Conversion

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

$$A \Rightarrow \beta\alpha^*$$

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{array}$$



Elimination of left recursion

Eliminate Left Recursion

Ex1: $P \rightarrow P + Q \mid Q$

$\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$

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

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

$$\begin{aligned} P &\rightarrow QP' \\ P' &\rightarrow +QP' \mid \varepsilon \end{aligned}$$



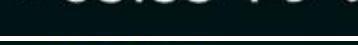
Ex2: $S \rightarrow S0S1S \mid 01$

$\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$

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

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S &\rightarrow 01S' \\ S' &\rightarrow 0S1SS' \mid \varepsilon \end{aligned}$$



Ex3: $A \rightarrow (B) \mid b$

$B \rightarrow B \times A \mid A$

$\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$ $\xrightarrow{\quad}$

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

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

$$\begin{aligned} A &\rightarrow (B) \mid b \\ B &\rightarrow AB' \\ B' &\rightarrow xAB' \mid \varepsilon \end{aligned}$$

Eliminate Left Recursion

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

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

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \mid \epsilon$$

Recursion

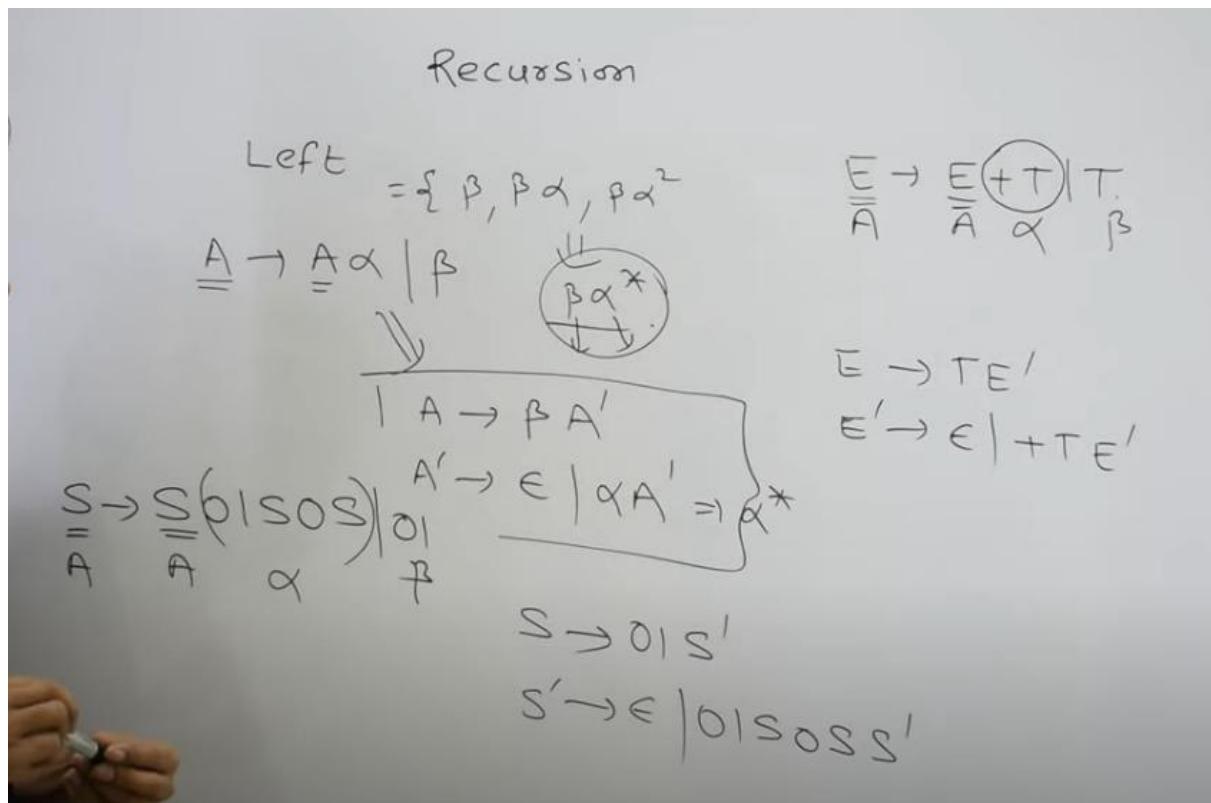
$$\begin{array}{ll} \text{Left} & = \{\beta, \beta\alpha, \beta\alpha^2\} \\ & \Downarrow \\ \overline{A} \rightarrow \overline{A}\alpha \mid \beta & \overline{A} \rightarrow \beta \overline{\alpha^*} \\ \overline{\alpha} & \overline{\alpha} \end{array} \quad \begin{array}{l} \text{Right} = \{\beta, \alpha\beta, \alpha^2\beta, \dots\} \\ \Downarrow \\ A \rightarrow \alpha \overline{A} \mid \beta = \alpha^* \beta \\ A \rightarrow \alpha A \\ \rightarrow \alpha \cdot \beta \\ A \rightarrow \alpha A \\ \rightarrow \alpha \alpha \beta \end{array}$$

Recursion

$$\begin{array}{ll} \text{Left} & = \{\beta, \beta\alpha, \beta\alpha^2\} \\ & \Downarrow \\ \overline{A} \rightarrow \overline{A}\alpha \mid \beta & \overline{A} \rightarrow \beta \overline{\alpha^*} \\ \overline{\alpha} & \overline{\alpha} \end{array} \quad \begin{array}{l} \overline{E} \rightarrow \overline{E} (+T) \mid T \\ \overline{A} \quad \alpha \quad \beta \\ \Downarrow \\ \left| \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \epsilon \mid \alpha A' \Rightarrow \alpha^* \end{array} \right. \end{array}$$

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow \epsilon \mid +TE' \end{array}$$

Recursion



$$\underline{\underline{A}} \rightarrow \underline{\underline{A}}(\underline{\underline{\beta}} | \underline{\underline{S}}) | \underline{\underline{\beta}}$$

$$\underline{\underline{L}} \rightarrow \underline{\underline{S}} \underline{\underline{L'}}$$

$$\underline{\underline{L'}} \rightarrow \underline{\underline{\epsilon}} | \underline{\underline{\beta}} \underline{\underline{S}} \underline{\underline{L'}}$$

$$\begin{aligned} A \rightarrow \underline{\underline{A}}\alpha_1 | \underline{\underline{A}}\alpha_2, \quad & \underline{\underline{A'}} \rightarrow \underline{\underline{\epsilon}} | \underline{\underline{\alpha}} \underline{\underline{A'}} = \underline{\underline{\alpha}}^* \\ A \rightarrow \beta_1 | \beta_2 \underline{\underline{L}} \end{aligned}$$

$$\begin{aligned} \underline{\underline{L}} \rightarrow \underline{\underline{S}} \underline{\underline{L}}, \quad & \underline{\underline{L'}} \rightarrow \underline{\underline{\epsilon}} | \underline{\underline{\beta}} \underline{\underline{S}} \underline{\underline{L'}} \end{aligned}$$

$$\begin{aligned} A \rightarrow \beta_1 A' | \beta_2 A' | \dots, \quad & \\ \underline{\underline{A'}} \rightarrow \underline{\underline{\epsilon}} | \underline{\underline{\alpha}}_1 \underline{\underline{A'}} | \underline{\underline{\alpha}}_2 \underline{\underline{A'}} = \dots \end{aligned}$$

Precedence and Associativity of Operators Examples

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

GATE-2014

$$S \rightarrow T * P$$

which one of the following is TRUE?

$$T \rightarrow T \mid T * U$$

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

$$P \rightarrow Q + P \mid Q$$

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

$$Q \rightarrow id$$

c) Both + & * are right associative.

$$U \rightarrow id$$

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

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

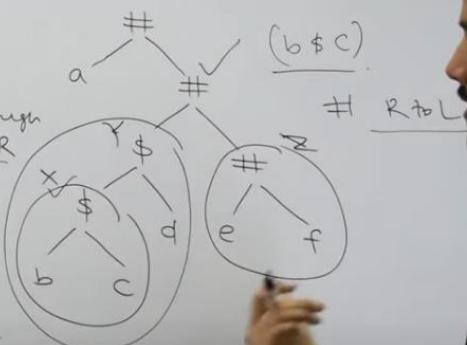
which of the following is correct
for the given parse tree.

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

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

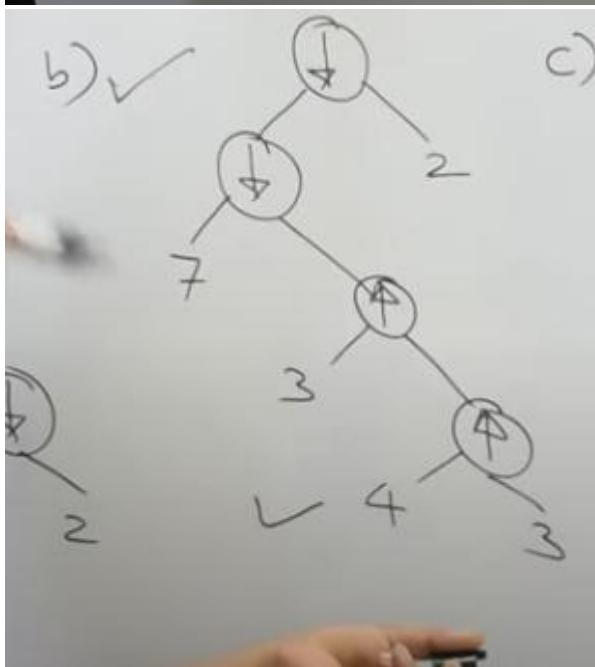
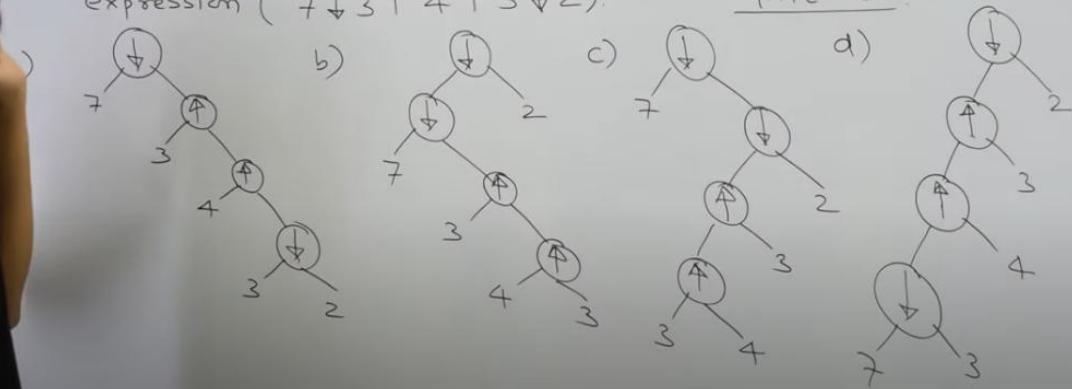
X c) \$ has higher precedence and is left
associative, # is left associative.

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



Consider two binary operators ' \uparrow ' and ' \downarrow ' with the precedence of operator \downarrow being lower than that of operator \uparrow . Operator \uparrow is right associative while operator \downarrow is left associative. Which one of the following represents the parse tree for expression $(7 \downarrow 3 \uparrow 4 \uparrow 3 \downarrow 2)$.

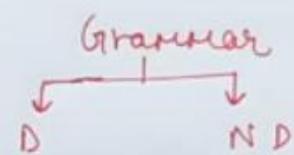
GATE-2011.



Elimination of left factoring

LEFT FACTORING

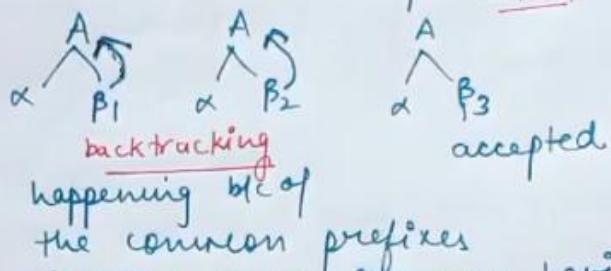
Sometimes, it is not clear which production to choose / expand a non-terminal b/c multiple productions begin with the same (terminal/non terminal) lookahead. This type of Grammar \rightarrow Non Deterministic grammar, or Grammar containing left factoring



$$A \rightarrow \underline{\alpha} \beta_1 \mid \underline{\alpha} \beta_2 \mid \underline{\alpha} \beta_3$$

common prefixes
common prefixes

suppose we have to accept $\underline{\alpha} \beta_3$



One or more productions on the RHS are having something common in the prefixes
This is also called common prefix problem or non deterministic grammar.

Q Y backtracking happened?

b/c we are making our decision only after seeing α .
i.e.

which production to choose without seeing full R.H.S.

if iIP is $\alpha \beta_3$
 we should make the decision $\alpha \beta_3$ if available
 then only we will go into that production.

so, for $A \rightarrow \underline{\alpha} \beta_1 / \underline{\alpha} \beta_2 / \underline{\alpha} \beta_3$ } Non Deterministic Grammar.

$\alpha \begin{cases} A \\ A' \\ \beta_3 \end{cases}$ $\begin{cases} A \rightarrow \underline{\alpha} A' \\ A' \rightarrow \beta_1 / \beta_2 / \beta_3 \end{cases}$ } we have postponed the decision making

Deterministic grammar

The procedure we used to convert Non deterministic grammar \rightarrow Deterministic Grammar is called
Left Factoring.

Problem with Non Deterministic Grammar?
 was most of the top down parser will have to
backtrack.

so, we don't need Non-deterministic grammar.

$$(i) S \rightarrow \underline{iE + Se} S / \underline{iE + s} / a$$

$$E \rightarrow b$$

$$\text{soln} \rightarrow \quad \begin{cases} S \rightarrow iEts s' / a \\ s' \rightarrow es / \epsilon \\ E \rightarrow b \end{cases}$$

$$(ii) \quad S \rightarrow \underline{a} SSbS$$

$$| \underline{a} \quad Sasb$$

$$| \underline{a} \quad bb$$

$$| b$$

$A \rightarrow aA$ } not
 $B \rightarrow aB$ } common
 prefix
 problem

$$\boxed{S \rightarrow as' / b}$$

$$\boxed{S' \rightarrow \underline{ssbs} \quad | \underline{sasb}} \quad \left. \begin{array}{l} \\ \end{array} \right\} \text{again common prefixes}$$

$$\downarrow$$

$$\left. \begin{array}{l} S' \rightarrow ss'' / bb \\ S'' \rightarrow sbs / aSb \end{array} \right\}$$

Left factoring.

$$\checkmark ND \Rightarrow \text{Det}$$

$$A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | \alpha \beta_4$$

$$A \rightarrow \alpha \underline{\beta}'$$

$$\beta' \rightarrow \beta_1 | \beta_2 | \beta_3 | \beta_4$$

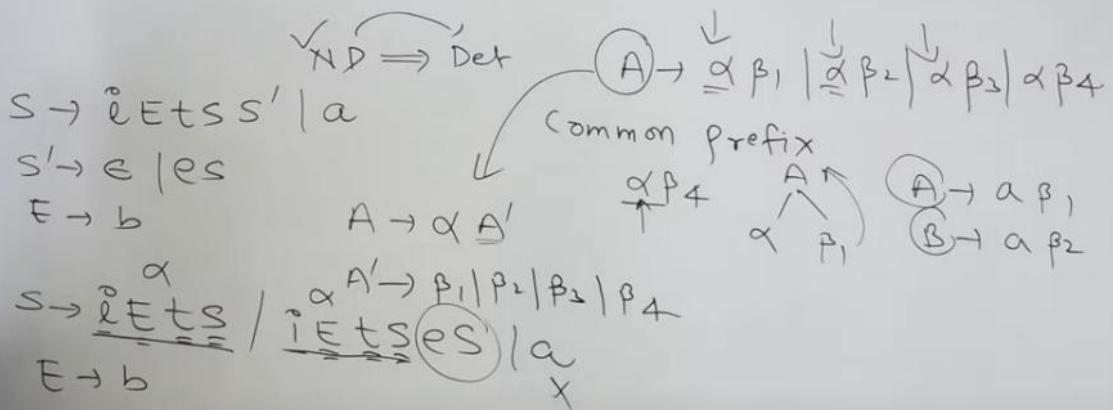
common prefix

$$\alpha \beta_4 \quad \alpha \beta_1 \quad \alpha \beta_2 \quad \alpha \beta_3$$

$$A \rightarrow a \beta_1$$

$$B \rightarrow a \beta_2$$

Left factoring.



$S \rightarrow \alpha \underline{(ssbs)} | \alpha \underline{ss} \underline{asb} | \alpha \underline{bb} | b$

$S \rightarrow \alpha s' | b$

$S' \rightarrow \underline{s} \underline{(sbs)} | \underline{s} \underline{asb} | \underline{bb}$

$\beta_4 \quad S' \rightarrow ss'' | bb$
 $S'' \rightarrow sbs | asb$

Left factoring

$$S \rightarrow b \underline{S} (\underline{\underline{S}} a a S) | b \underline{S} \underline{\underline{S}} a S b | b \underline{S} \underline{\underline{S}} b | a$$

$$\checkmark S \rightarrow b S S' | a$$

$$S' \rightarrow \underline{\underline{S}} a a S | \underline{\underline{S}} a S b | b$$

$$\checkmark S' \rightarrow S a S'' | b$$

$$\checkmark S'' \rightarrow a S | S b$$

$$S \rightarrow \underline{\underline{a}} | \underline{\underline{a}} b | \underline{\underline{a}} b c | \underline{\underline{a}} b c d$$

$$\textcircled{1} S \rightarrow a S'$$

$$\times S' \rightarrow \underline{\epsilon} | b | \underline{\underline{b}} c | \underline{\underline{b}} c d$$

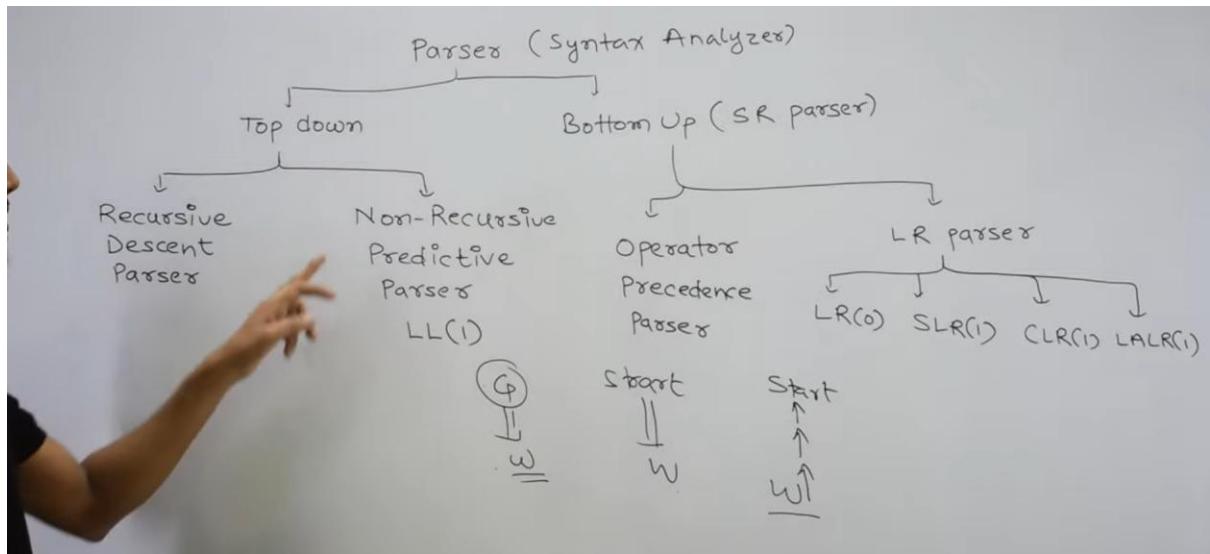
$$\textcircled{2} S' \rightarrow \underline{\epsilon} | b S''$$

$$\times S'' \rightarrow \underline{\epsilon} | c | \underline{\underline{c}} | \underline{\underline{c}} d$$

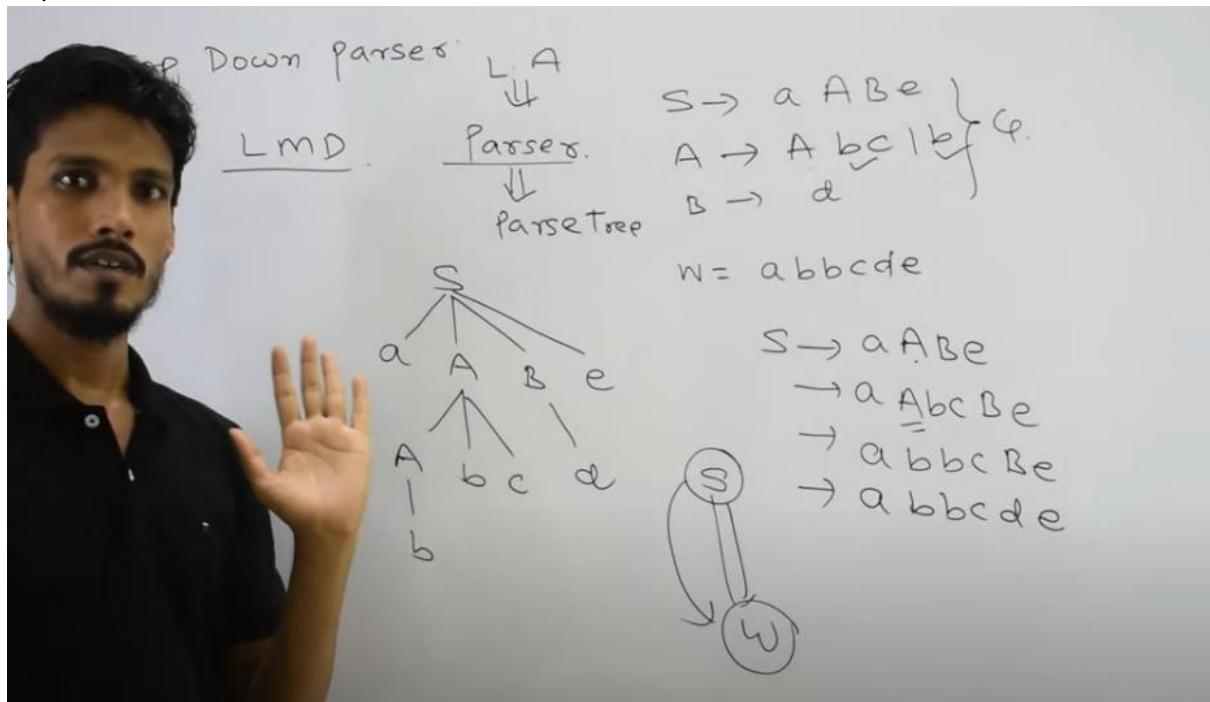
$$\textcircled{3} S'' \rightarrow \underline{\epsilon} | c S'''$$

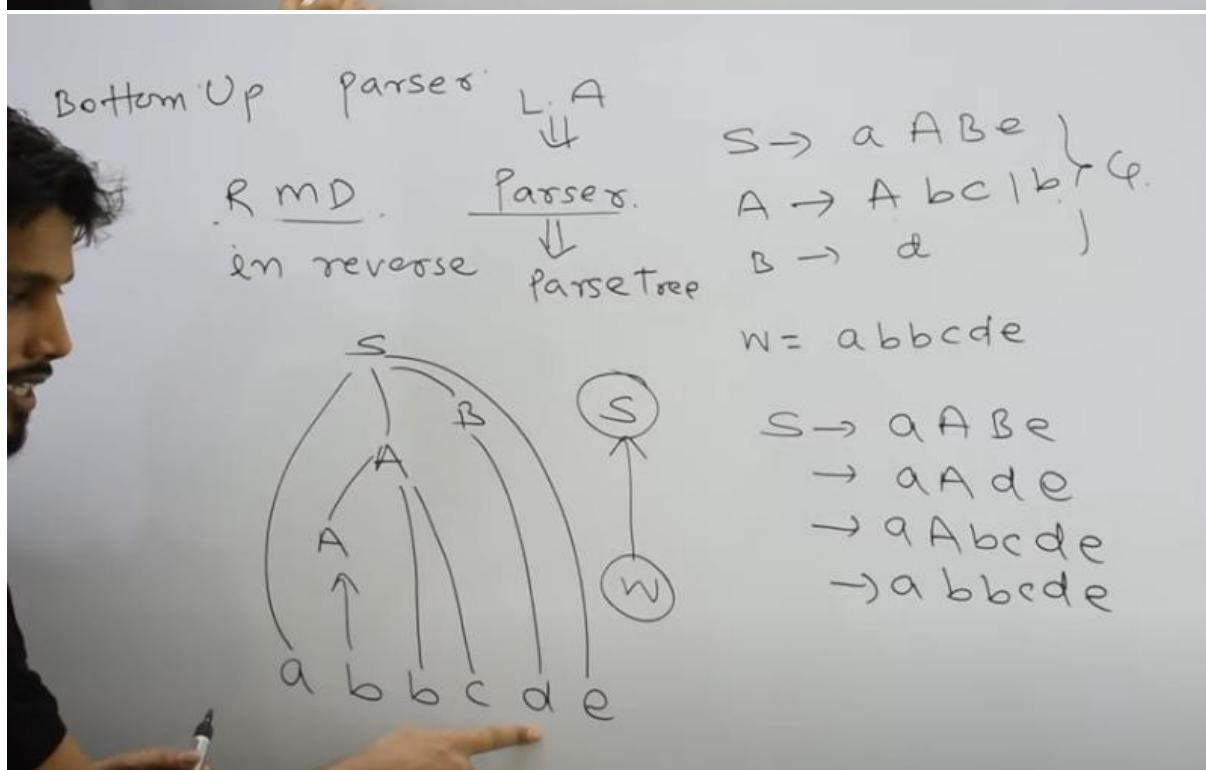
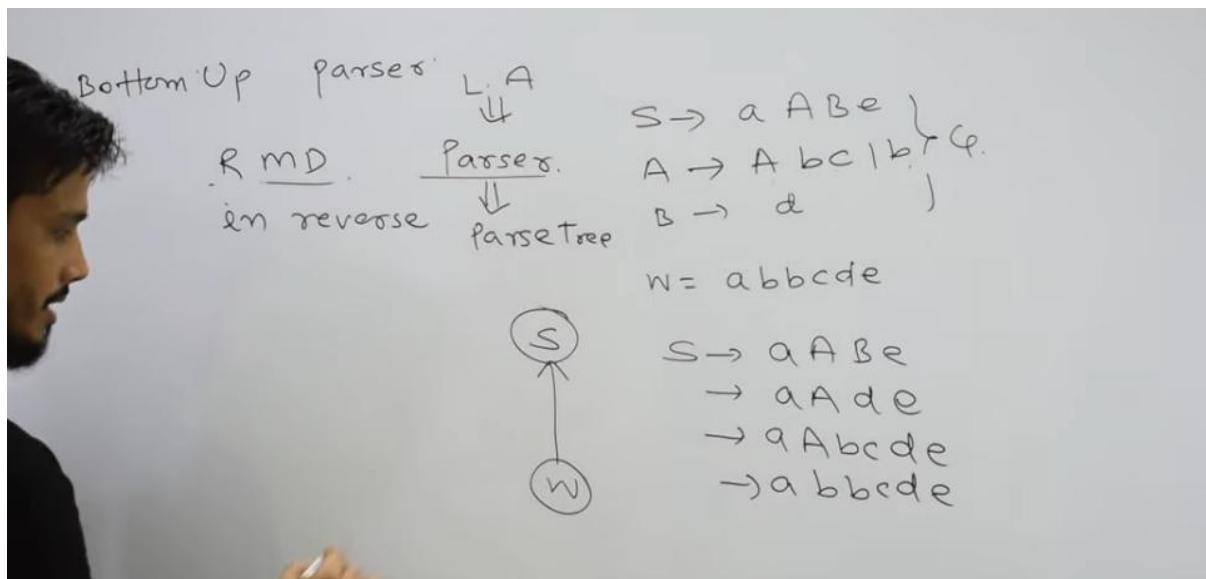
$$\textcircled{4} S''' \rightarrow \underline{\epsilon} | \underline{\underline{d}}$$

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.

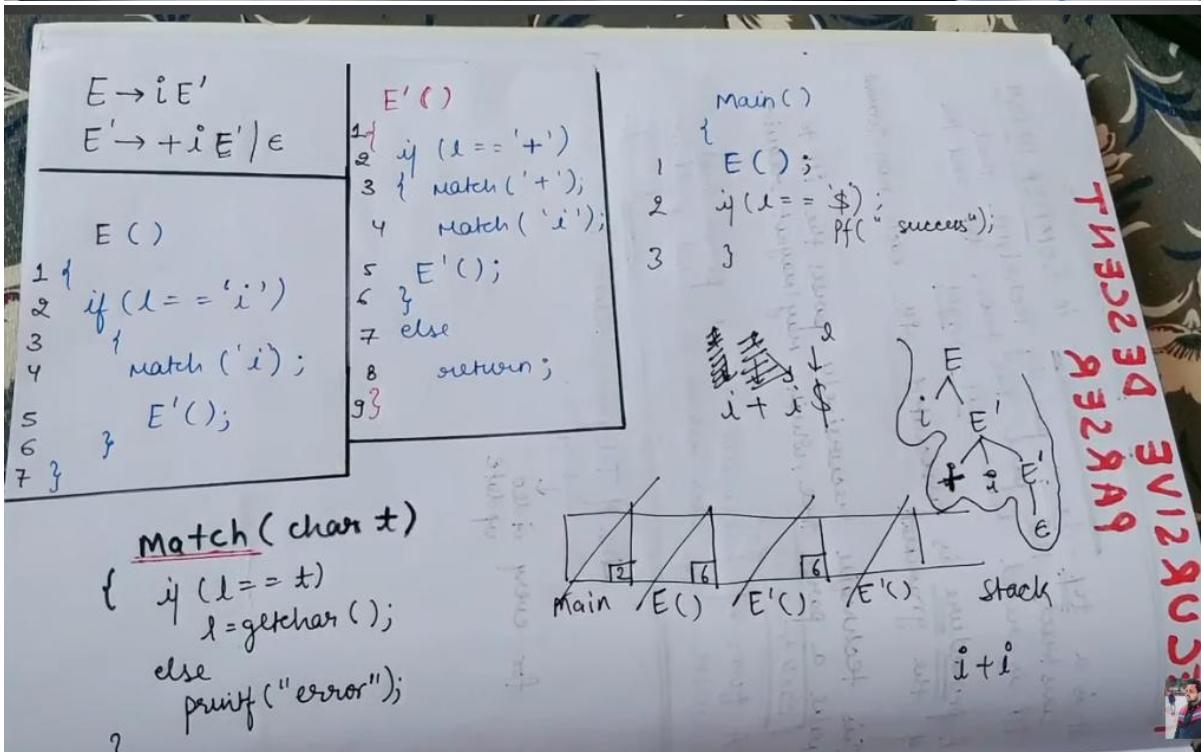
RECURSIVE DESCENT PARSER

It is a top-down parsing technique that constructs the parse tree from top and the input is read from left → right.

A procedure is associated with each non-terminal of the grammar.

This technique recursively parses the input to make a parse tree, which may/maynot require backtracking.

A form of recursive descent parser that does not require any backtracking is called predictive parsing.



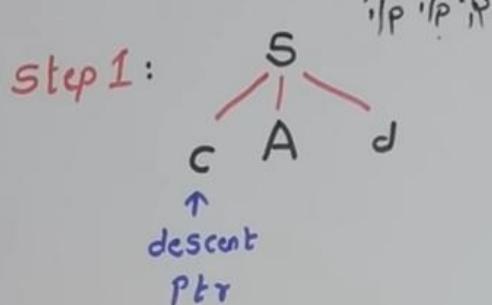
Example for backtracking:

Consider the grammar G:

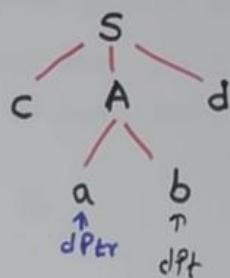
$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

and input string $w = cad$

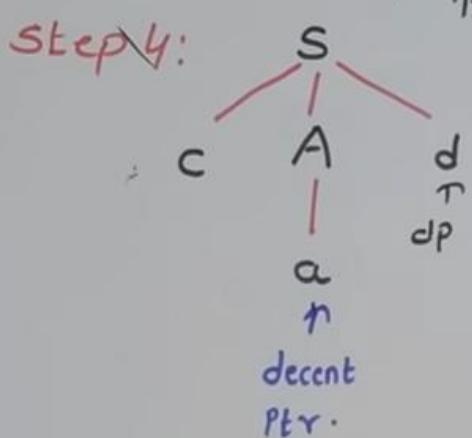


Step 2:

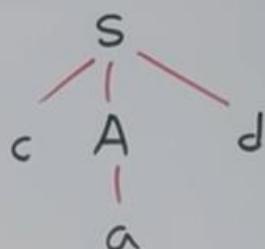


Step 3: Backtracking

Input string $w = ca d$



Yes the given string can be
Parse and the parse
tree is shown below



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: First(A) gives set of terminals that begins in all strings derived from A.

Follow: Follow(A) gives set of all terminals that follow immediately to the right of A.

1. If $A \rightarrow a\alpha$, $a \in (V \cup T)^*$
 $\text{First}(A) = \{a\}$

2. If $A \rightarrow \epsilon$ then $\text{First}(A) = \{\epsilon\}$

3. If $A \rightarrow BC$ then
 $\rightarrow \text{First}(A) = \text{First}(B)$
 If $\text{First}(B)$ doesn't contain ϵ .
 $\rightarrow \text{First}(A) = \text{First}(B) \cup \text{First}(C)$
 If $\text{First}(B)$ contains ϵ .

1. If S is start symbol then $\text{follow}(S) = \{\$\}$.
 2. If $A \rightarrow \alpha B \beta$, then $\text{follow}(B) = \text{First}(\beta)$
 If $\text{First}(\beta)$ doesn't contain ϵ .
 3. If $A \rightarrow \alpha B$, then $\text{follow}(B) = \text{follow}(A)$.



immediately to the right of A.

$\{\$\}$ = $\overset{1}{A} \xrightarrow{} a \underset{2}{B} C, \{a\}$
 $\{\$\}$ $B \rightarrow b \quad \{b\}$
 $\{\$\}$ $C \rightarrow c \quad \{c\}$

First and Follow Examples

Computation of FIRST

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

$$\text{FIRST } (a) = \{a \mid a \rightarrow^* a\beta \text{ for some string } \beta\}$$

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

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

If $X \rightarrow \epsilon$, then $\text{FIRST}(X) = \{\epsilon\}$

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

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

(a) If Y is terminal, then

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

(b) If Y_1 is Non-terminal and

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

(c) If $\text{FIRST}(Y_1)$ contains ϵ , then.

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

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

$\text{FIRST}(Y_2, Y_3) = \text{FIRST}(Y_2)$, if $\text{FIRST}(Y_2)$ does not contain ϵ .

If $\text{FIRST}(Y_2)$ contain ϵ , then

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

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

If Y_k 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.

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

Rules to find FOLLOW

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

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

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

Or

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

$$\text{FOLLOW} (B) = \text{FIRST} (\beta) - \{\epsilon\} \cup \text{FOLLOW} (A)$$

\because 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
$S \rightarrow a B D h$	{a}	{ $\$$ }
$B \rightarrow C C$	{C}	{g, f, h}
$C \rightarrow b C \mid \epsilon$	{b, ϵ }	{g, f, h}
$D \rightarrow E F$	{g, f, ϵ }	{h}
$E \rightarrow g \mid \epsilon$	{g, ϵ }	{f, h}
$F \rightarrow f \mid \epsilon$	{f, ϵ }	{h}
$E(F) S \rightarrow a \underline{B}(D h)$	a $\underline{B} f h$	
$E f$	$\underline{B} \circled{g} h$	
$E -$	$a \underline{B} \cancel{f} h$	
	$a \underline{B} \circled{h}$	

	First	Follow
$E \rightarrow TE'$	{id, (,)}	{\$,)}
$E' \rightarrow \epsilon \mid TE'$	{\epsilon, +,)}	{\$,)}
$T \rightarrow FT'$	{id, c,)}	{+, \$,)}
$T' \rightarrow \epsilon \mid \times FT'$	{\epsilon, \times)}	{+, \$,)}
$F \rightarrow \underline{id} \mid (E)$	{id, (,)}	{\times, +, \$,)}
$T \rightarrow F(T')$		

	First	Follow
$S \rightarrow Bb \mid C\alpha$	{a, b, c, \alpha}	{\\$}
$B \rightarrow aB \mid \epsilon$	{a, \epsilon}	{b}
$C \rightarrow cC \mid \epsilon$	{c, \epsilon}	{\alpha}

	First	Follow
$S \rightarrow \underline{A}a\underline{Ab} \mid BbBa$	{a, b}	{\\$}
$A \rightarrow \epsilon$	{\epsilon}	{a, b}
$B \rightarrow \epsilon$	{\epsilon}	{b, a}

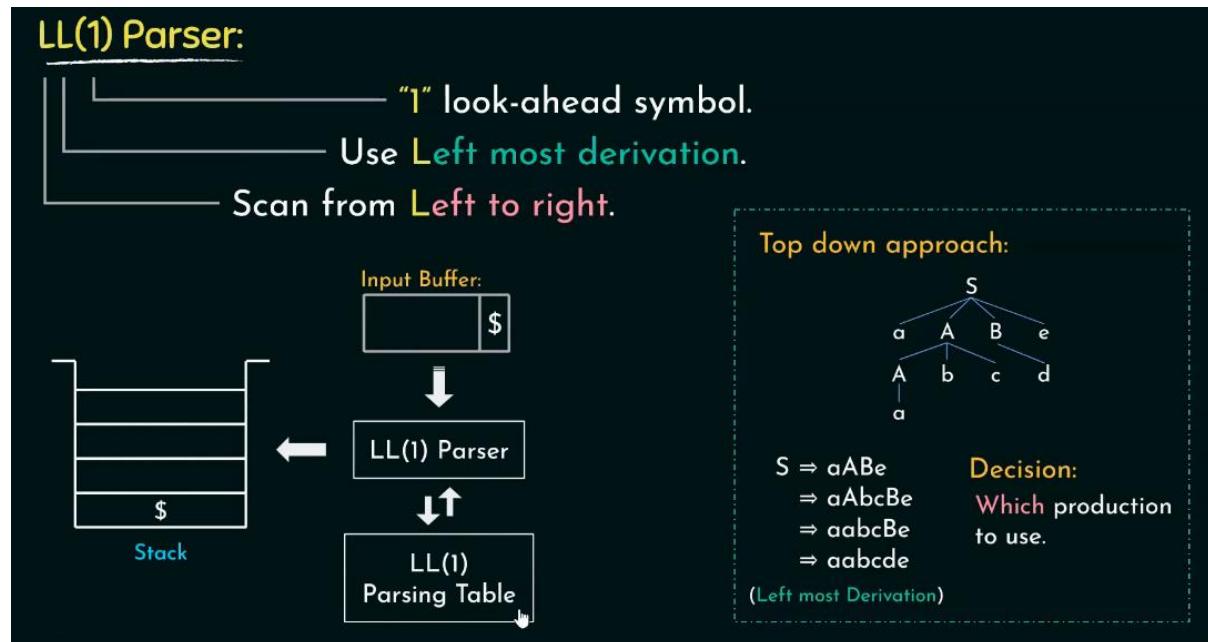
$S \rightarrow \underline{a}b \mid \underline{b}a$

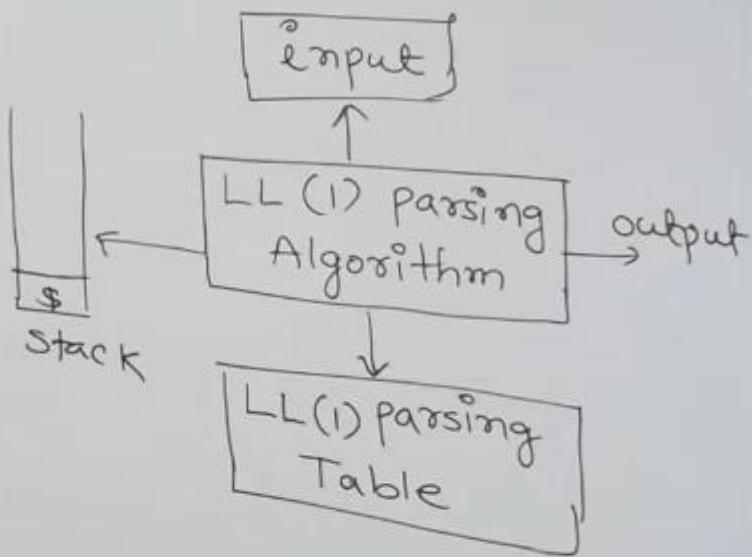
LL(1) parser

<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 $A \rightarrow \alpha$ under $M[A, a]$
where $a \in \text{First}(\alpha)$
2. Add $A \rightarrow \alpha$ under $M[A, a]$
if $\text{First}(\alpha)$ contains ϵ .



	First	Follow	Predictive Par
$E \rightarrow TE'$	{ id, (}	{ \$,) }	
$E' \rightarrow \epsilon +TE'$	{ ε, + }	{ \$,) }	
$T \rightarrow FT'$	{ id, (}	{ +, \$,) }	
$T' \rightarrow \epsilon *FT'$	{ ε, * }	{ +, \$,) }	
$F \rightarrow id (E)$	{ id, (}	{ *, +, \$,) }	

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

1. Add $A \rightarrow \alpha$ under $M[A, a]$

where $a \in \text{First}(\alpha)$

2. Add $A \rightarrow \alpha$ under $M[A, a]$

where $a \in \text{Follow}(A)$
if $\text{First}(\alpha)$ contains



Non Recursive Descent parser.

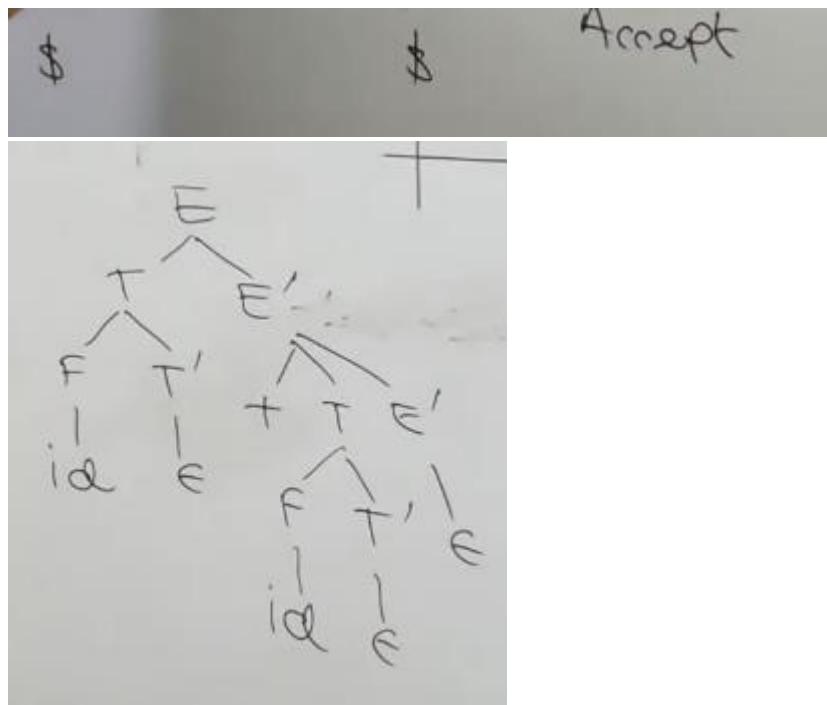
Predictive parser or LL(1)

Stack	Input
\$ E	id + id \$
\$ E' T	id + id \$
\$ E' T' F	id + id \$
\$ E' T' id	id + id \$
\$ E' T'	+ id \$
\$ E'	+ id \$
\$ E' +	+ id \$
\$ E' T	id \$
\$ E' T' F	id \$
\$ E' T' id	id \$
\$ E' T'	\$
\$ E'	\$
\$	\$

Production:

$E \rightarrow TE'$
$T \rightarrow FT'$
$F \rightarrow id$
pop
$T' \rightarrow E$
$E' \rightarrow +TE'$
pop
$T \rightarrow FT'$
$F \rightarrow id$
pop
$T' \rightarrow E$
$E' \rightarrow E$

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



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

How to check given grammar is LL(1) or not.

1. If G doesn't contain ϵ .

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$$

$$\text{First}(\alpha_1) \cap \text{First}(\alpha_2) = \emptyset$$

$$\text{First}(\alpha_2) \cap \text{First}(\alpha_3) = \emptyset$$

$$\text{First}(\alpha_3) \cap \text{First}(\alpha_1) = \emptyset$$

2. If G contains ϵ .

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \epsilon$$

$$\text{First}(\alpha_1) \cap \text{First}(\alpha_2) = \emptyset$$

$$\text{First}(\alpha_1) \cap \text{Follow}(A) = \emptyset$$

$$\text{First}(\alpha_2) \cap \text{Follow}(A) = \emptyset$$

$S \rightarrow E | a$ LL(1) \times
 $E \rightarrow a$ $a \cap a = a$

$S \rightarrow a A(B b)$
 $A \xrightarrow{a} a | \epsilon$ $\xrightarrow{\phi} a \cap \{d, b\}$
 $B \xrightarrow{d} d | \epsilon$ $a \cap b =$
 $\{a, \epsilon\} = \phi.$
 $(B b)$ LL(1)
 $d b$
 $\epsilon \cdot b = b$

$$1. \quad \begin{array}{l} \checkmark s \xrightarrow{\quad} aSA | \epsilon \\ A \rightarrow c | \epsilon \end{array}$$

Follow(s)
 $= \{c, \$\}$

$$2. \quad \begin{array}{l} F(A) \\ a \cap \{c, \$\} = \{c, \$\} \\ a \cap \{c, \$\} = \emptyset \end{array}$$

$$\begin{array}{l} c \cap \{c, \$\} \\ = \{c\} \neq \emptyset \xrightarrow{s \rightarrow as} a sc \\ \text{LL(1) X.} \end{array}$$

How to check 0..

$$1. \quad \begin{array}{l} s \xrightarrow{\alpha_1} aSbS | \quad \xrightarrow{\alpha_2} bSaS | \epsilon \\ \text{Follow(s)} = \{b, a, \$\} \end{array}$$

$\checkmark a \cap b = \emptyset$

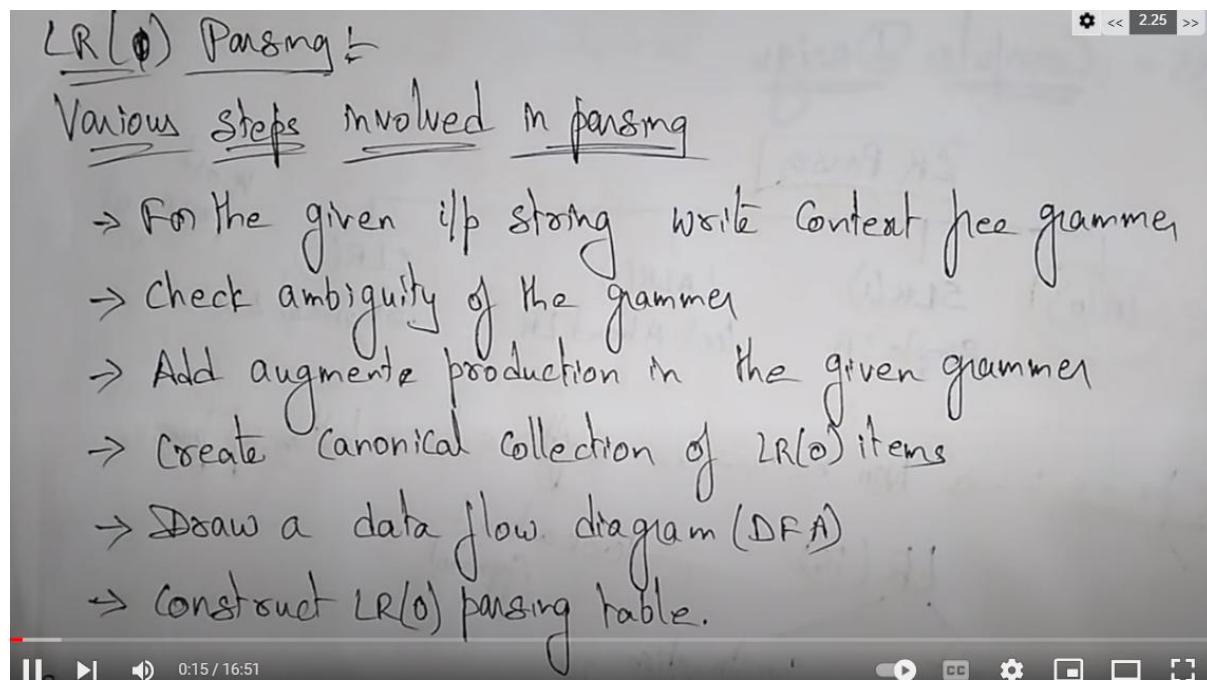
$$2. \quad a \cap \{b, a, \$\} = a \neq \emptyset.$$

$$b \cap \{b, a, \$\} = b \neq \emptyset$$

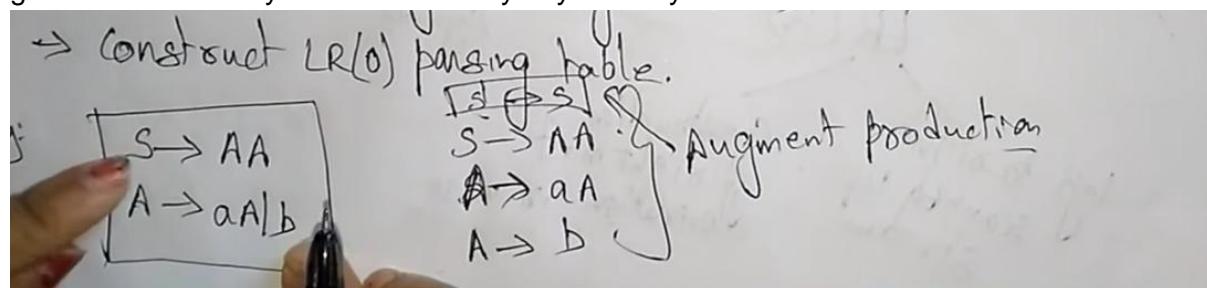
$\nwarrow \textcircled{4} \quad \text{LL(1) X}$

LR(0) Parser

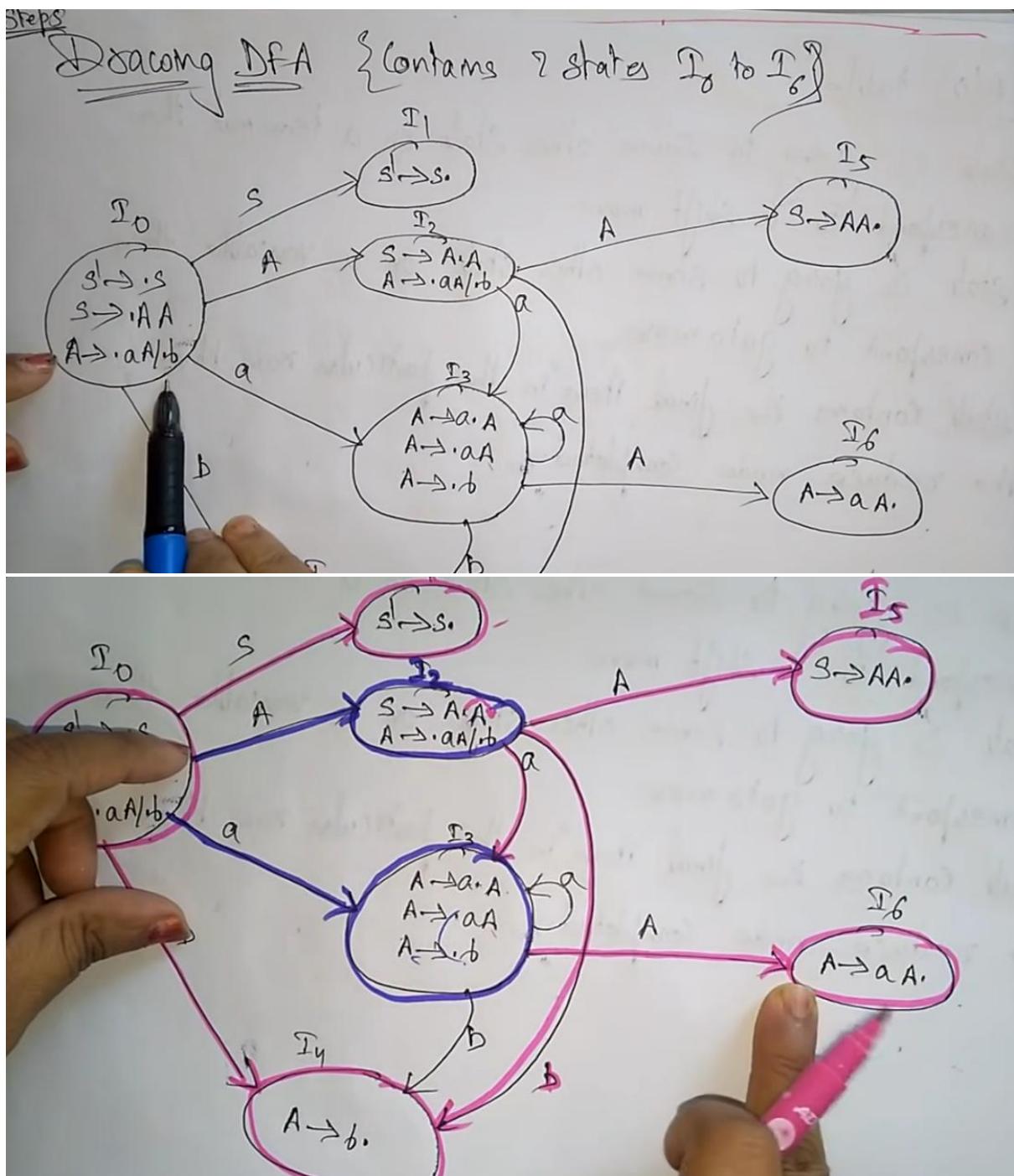
<https://www.geeksforgeeks.org/problem-on-lr0-parser/>

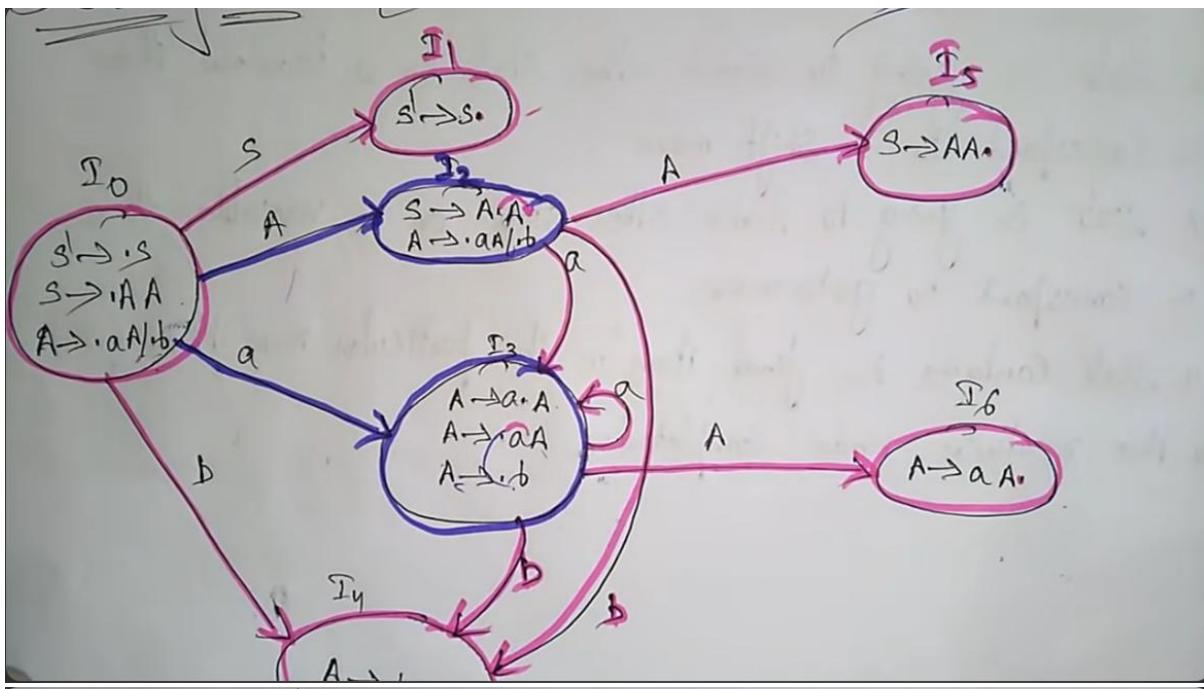


Augmented grammar helps us to identify when to stop the parser. Generally in augmented grammar the start symbol is derived by any other symbol



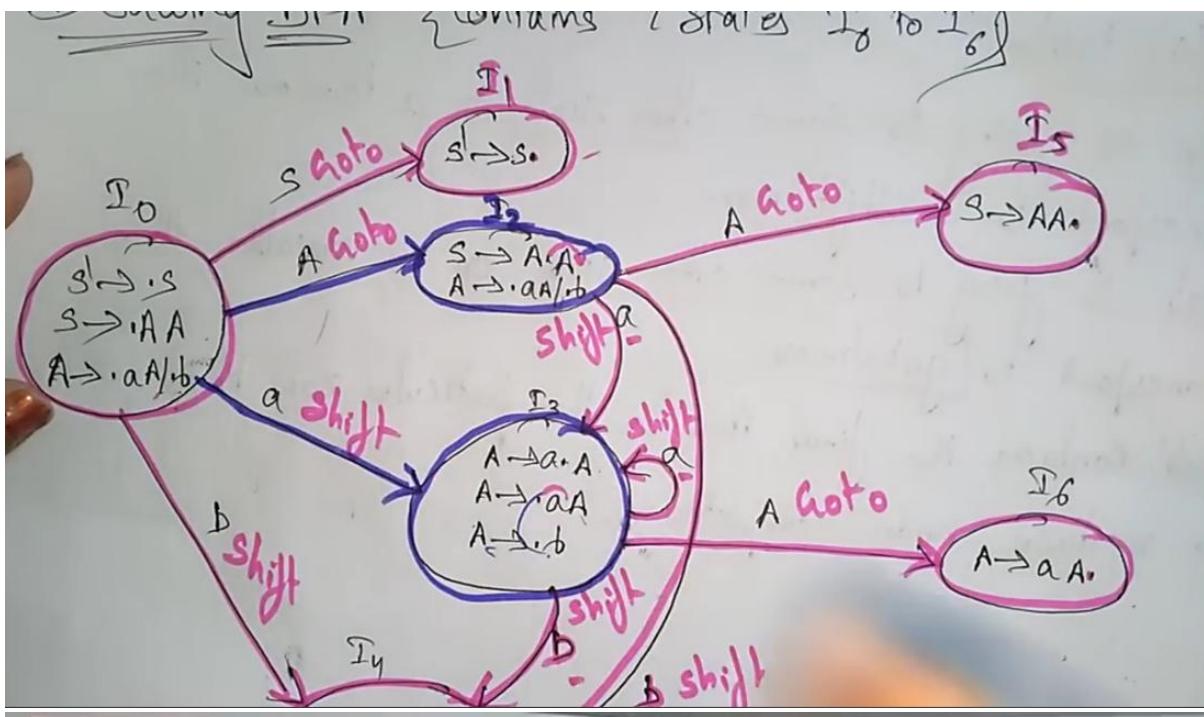
- Step 1: Create canonical collection of LR(0) items
- * An LR(0) item is a production A with dot at some position the right hand side of production.
 - * LR(0) items is useful to indicate that how much of the has been scanned up to a given point in the process of
 - * In LR(0), we place the reduce node in entire row.





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 the reduce node completed.



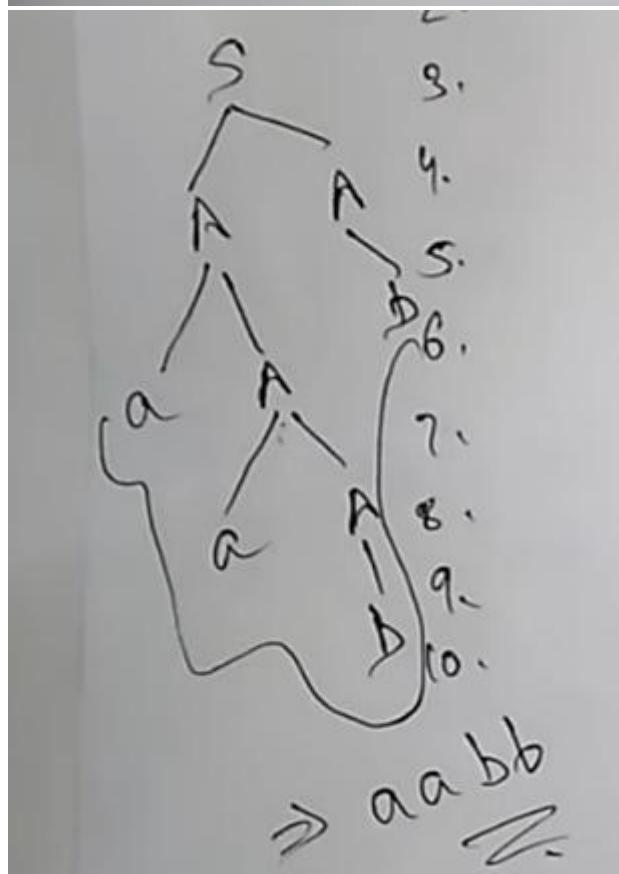
States	Action			GOTO
	a	b	\$	A
I0	S ₃	S ₄		2 S
I1			accept	1
I2	S ₃	S ₄		5
I3	S ₃	S ₄		6
I4	S ₃	S ₃	S ₃	
I5	S ₁	S ₁	S ₁	
I6	S ₂	S ₂	S ₂	

I₄, I₅, I₆ all contains final states

- $S \rightarrow AA$ — ①
- $A \rightarrow aA$ — ②
- $A \rightarrow b$ — ③

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

<u>Parsing steps</u>	<u>Parsing stack</u>	<u>Up</u>	<u>Action</u>
1.	\$	aabb\$	Shift a3
2.	\$ 0 a ₂ 3	<u>a</u> bb\$	shift a3
3.	\$ 0 a ₃ a ₃	<u>b</u> b\$	shift b4
4.	\$ 0 a ₃ a ₃ b ₄	b \$	reduce δ_3 ($A \rightarrow b$)
5.	\$ 0 a ₃ a ₃ A ₆	b \$	reduce δ_2 ($A \rightarrow aA$)
6.	\$ 0 [a ₃ A ₆]	b \$	reduce δ_2 ($A \rightarrow aA$)
7.	\$ 0 A ₂	b \$	shift b4
8.	\$ 0 A ₂ b ₄	\$	reduce δ_3 ($A \rightarrow b$)
9.	\$ 0 A ₂ A ₅	\$	reduce δ_1 ($S \rightarrow AA$)
10.	\$ 0 S 1	\$	Accept



LR(0) with epsilon productions

Bottom-up Parsing technique

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

To Design LR(0) follow the following steps.

- 1) Augmented grammar
- 2) Calculation of First and Follow set.
- 3) Transition diagram / State diagram
- 4) LR(0) Parsing table.

1) Augmented grammar

- We use dot to Scan or process symbol after dot.
- T (Terminal)
- N (Non Terminal)

$F \rightarrow \cdot xyz \rightarrow$ Ready to scan x.
 $F \rightarrow x \cdot yz \rightarrow$ x is scanned and Ready to scan y.
 $F \rightarrow xy \cdot z \rightarrow$ All symbol scanned.

Solution:

Given the grammar

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Design SLR Parser.

1) Augmented grammar

$$S' \rightarrow S \quad -0$$

$$S \rightarrow \cdot AaAb \quad -1$$

$$S \rightarrow \cdot BbBa \quad -2$$

$$A \rightarrow \cdot \quad -3$$

$$B \rightarrow \cdot \quad -4$$

2) Calculation of First and Follow set

First set:

$$\text{First}(B) = \{\epsilon\}$$

$$\text{First}(A) = \{\epsilon\}$$

$$\text{First}(S) = \text{First}(AaAb) \cup \text{First}(BbBa)$$

$$= \text{First}(A) - \epsilon \cup \text{First}(aAb) \cup \text{First}(B)$$

$$= \epsilon \cup \text{First}(bBa)$$

$$= \{\epsilon\} - \epsilon \cup \{a\} \cup \{\epsilon\} - \epsilon \cup \{b\}$$

$$= \{a, b\}$$

Follow set:

$$\text{Follow}(S) = \{\$\}$$

$$\text{Follow}(A) = \text{First}(aAb) \cup \text{First}(b) = \{a, b\}$$

$$\text{Follow}(B) = \text{First}(bBa) \cup \text{First}(a) = \{a, b\}$$

Augmented grammar

$$S' \rightarrow \cdot S \quad -0$$

$$S \rightarrow \cdot AaAb \quad -1$$

$$S \rightarrow \cdot BbBa \quad -2$$

$$A \rightarrow \cdot \quad -3$$

$$B \rightarrow \cdot \quad -4$$

Transition diagram

```

graph LR
    I0["I0  
S' → · S"] -- "a, b" --> I1["I1  
S' → S ·"]
    I1 -- "a, b" --> I2["I2  
S · → S"]
    I2 -- "a, b" --> I3["I3  
S · → S ·"]
    I3 -- "a, b" --> I4["I4  
S · → · AaAb"]
    I4 -- "a" --> I5["I5  
S · → · Aa · Ab"]
    I5 -- "a" --> I6["I6  
S · → · Aa · · b"]
    I6 -- "b" --> I7["I7  
S · → · Aa · b ·"]
    I7 -- "b" --> I8["I8  
S · → · Aa · Ab ·"]
    I8 -- "a" --> I9["I9  
S · → · AaAb ·"]
    I9 -- "a" --> I0
    I0 -- "$" --> I1
    I1 -- "$" --> I2
    I2 -- "$" --> I3
    I3 -- "$" --> I4
    I4 -- "a" --> I5
    I5 -- "a" --> I6
    I6 -- "b" --> I7
    I7 -- "b" --> I8
    I8 -- "a" --> I9
    I9 -- "a" --> I0
    I0 -- "A" --> I1
    I1 -- "A" --> I2
    I2 -- "A" --> I3
    I3 -- "A" --> I4
    I4 -- "B" --> I5
    I5 -- "B" --> I6
    I6 -- "B" --> I7
    I7 -- "B" --> I8
    I8 -- "B" --> I9
    I9 -- "B" --> I0
  
```

LR(0) Parsing table

State	Action	a	b	\$	s	A	B	GOTO
I0	R3/R4	R3/R4			1	2	3	
I1			Accept					
I2	S4							
I3	S5							
I4	R3	R3						
I5	R4	R4						
I6	·	S8						
I7	S9	S8						
I8			R1					
I9			R2					

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

S/S - LR(0)
S/R - LR(1)
R/S - LR(0)
R/R - 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 grammar
- 2) Calculation of First and Follow set.
- 3) Transition diagram / State diagram
- 4) LR(0) Parsing table.

1) Augmented grammar

- We use dot to scan or process symbol after dot.
 - T (Terminal)
 - N (Non-Terminal)
- $F \rightarrow \cdot xyz$ → Ready to scan x.
 $F \rightarrow x \cdot yz$ → x is scanned and ready to scan y.
 $F \rightarrow xy \cdot z$ → All symbol scanned.

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

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

Solution: 1) Augmented grammar

$$\begin{aligned} E' &\rightarrow \cdot E & \rightarrow 0 \\ E &\rightarrow \cdot E+E & \rightarrow 1 \\ E &\rightarrow \cdot E * E & \rightarrow 2 \\ E &\rightarrow \cdot (E) & \rightarrow 3 \\ E &\rightarrow \cdot id & \rightarrow 4 \end{aligned}$$

2) Calculation First and Follow set

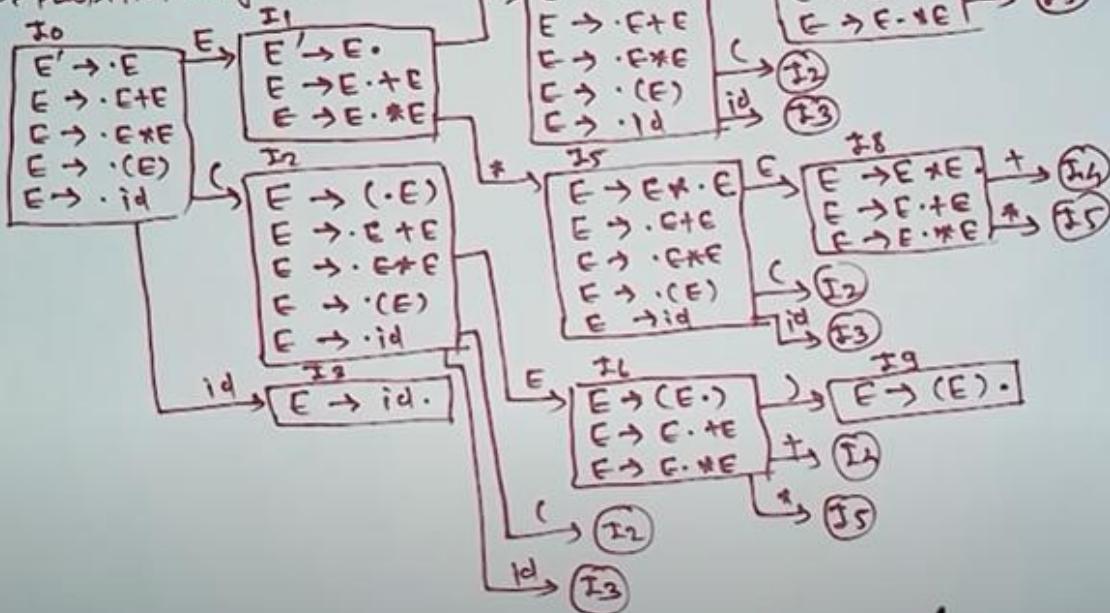
$$\begin{aligned} \text{First}(E) &= \text{First}(E+E) \cup \text{First}(E * E) \cup \{(E)\} \cup \{\text{id}\} \\ &= \text{First}(E+E) \cup \text{First}(E * E) \cup \{C\} \cup \{\text{id}\} \\ &= \text{First}(E) \cup \text{First}(E) \cup \{(id\}) \\ &= \{(id\}\} \end{aligned}$$

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

Augmented grammar

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot E+E \\ E &\rightarrow \cdot E * E \\ E &\rightarrow \cdot (E) \\ E &\rightarrow \cdot id \end{aligned}$$

3) Transition diagram



4) LR(0) Parsing table

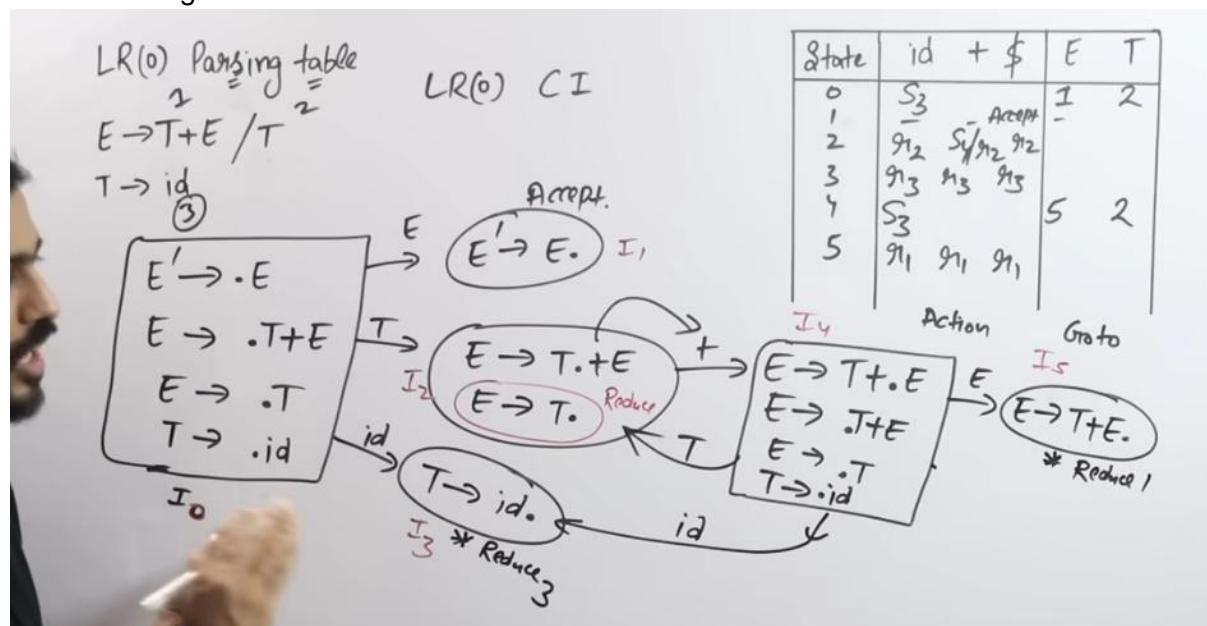
State	Action							Goto
	+	*	()	id	\$	E	
I ₀			S ₂		S ₃		I	
I ₁	S ₄	S ₅				Accept		
I ₂			S ₂		S ₃		6	
I ₃	R ₄	R ₄		R ₄		R ₄		
I ₄			S ₂		S ₃		7	
I ₅			S ₂		S ₃		8	
I ₆	S ₄	S ₅		S ₉				
I ₇	S ₄ /R ₁	S ₅ /R ₁		R ₁		R ₁		
I ₈	S ₄ /R ₂	S ₅ /R ₂		R ₂		R ₂		
I ₉	R ₃	R ₃		R ₃		R ₃		

$S/S \rightarrow LR(0)$
 S/R
 R/S } \rightarrow Not
 R/R } LR(0)

Result: Grammar contains multiple entries. Thus given grammar is not LR(0).

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.



LR(0) parsing Table construction

Algorithm:

1. Find Augmented grammar.
2. $I_0 = \text{Closure}(\text{Augmented LR}(0) \text{ item})$
3. Apply closure and goto function and find all collection of $\text{LR}(0)$ item using Finite Automata (DFA)
4. Reduce DFA = $\text{LR}(0)$ parsing Table.

SLR(1) parser

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

SLR(1) parsing

↓
Simple LR

⇒ Smallest class of grammar
⇒ few no: of states
⇒ simple & fast to construct

→ In SLR we place the reduce move only in the follow of left hand side not to entire row.

- LR(0) → LR(0) items
- SLR(1) → LR(1) items
- CLR(1) → LR(1) items
- LALR(1) → LR(1) items

SLR(1) Parser

Step1: Augment the Grammar and number it

Step2: First & Follow

Step3: Construct DFA

Step4: Parsing Table

Step5: Stack Implementation

Step6: Parse Tree

$S \rightarrow cAd$
 $A \rightarrow able$

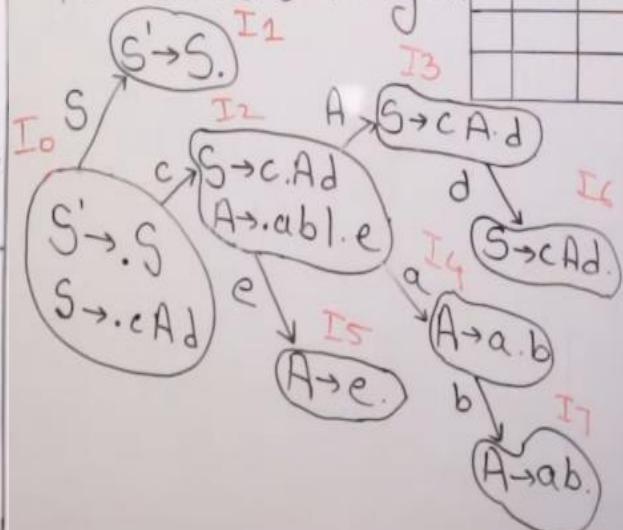
String: ced\$

Step1: Augment the grammar and number it | First & follow

$S' \rightarrow S$	c	\$
① $S \rightarrow cAd$	c	\$
$A \rightarrow a\text{ble}$	a, e	d

② ③

Step3: construct DFA diagram



I0 contains the closure of the starting symbol always.

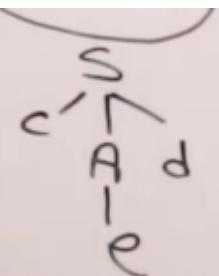
I5: $\boxed{A \rightarrow e}$
 $[d, 5] : r_3$
I6: $\boxed{S \rightarrow cAd}$
 $[\$, 16] : r_1$
I7: $\boxed{A \rightarrow ab}$
 $[d, 7] : r_2$

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

States	a	b	c	d	e	\$	A	Goto
I ₀			S ₂					
I ₁							A	1
I ₂	S ₄				S ₅		3	
I ₃			S ₆					
I ₄		S ₇						
I ₅				r ₃				
I ₆						r ₁		
I ₇				r ₂				

Step 5: Stack Implementation

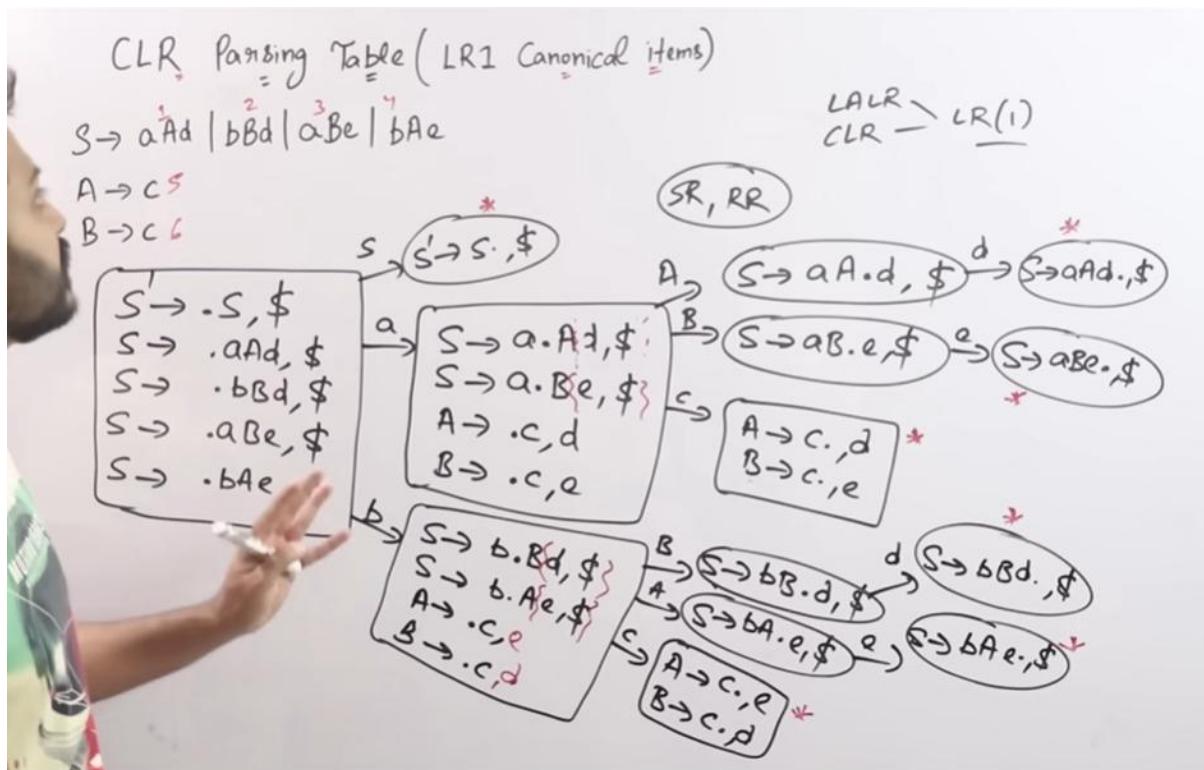
State	Input	Action
O	c e d \$	Shift c → S ₂
O C ₂	e d \$	Shift e → S ₅
O C ₂ e S ₅	d \$	Reduce A → e
O C ₂ A 3	d \$	Shift d → S ₆
O C ₂ A 3 d S ₆	\$	Reduce S ₆ & A d
O S ₁	\$	Accept



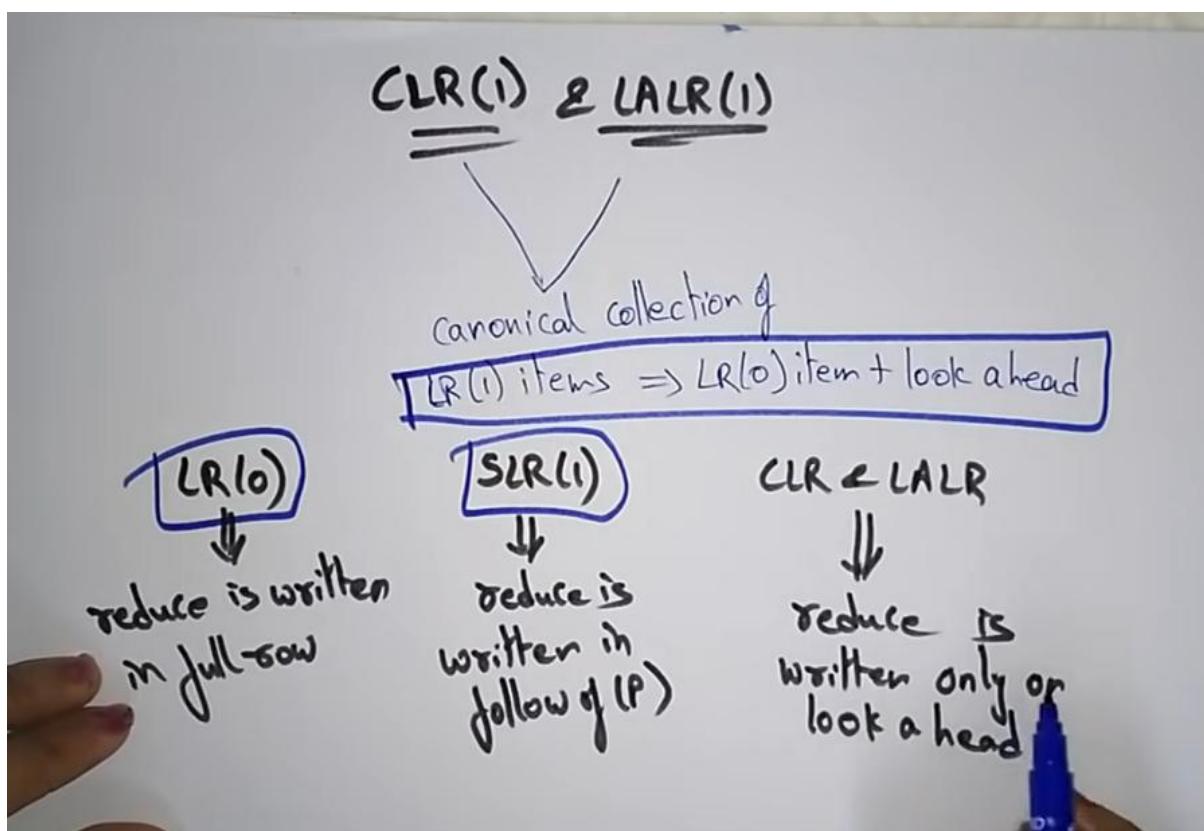
CLR (1) parser

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

Example 1



Example 2



$E \rightarrow BB$

$B \rightarrow CB/d$

Augment grammar & LR(1) items

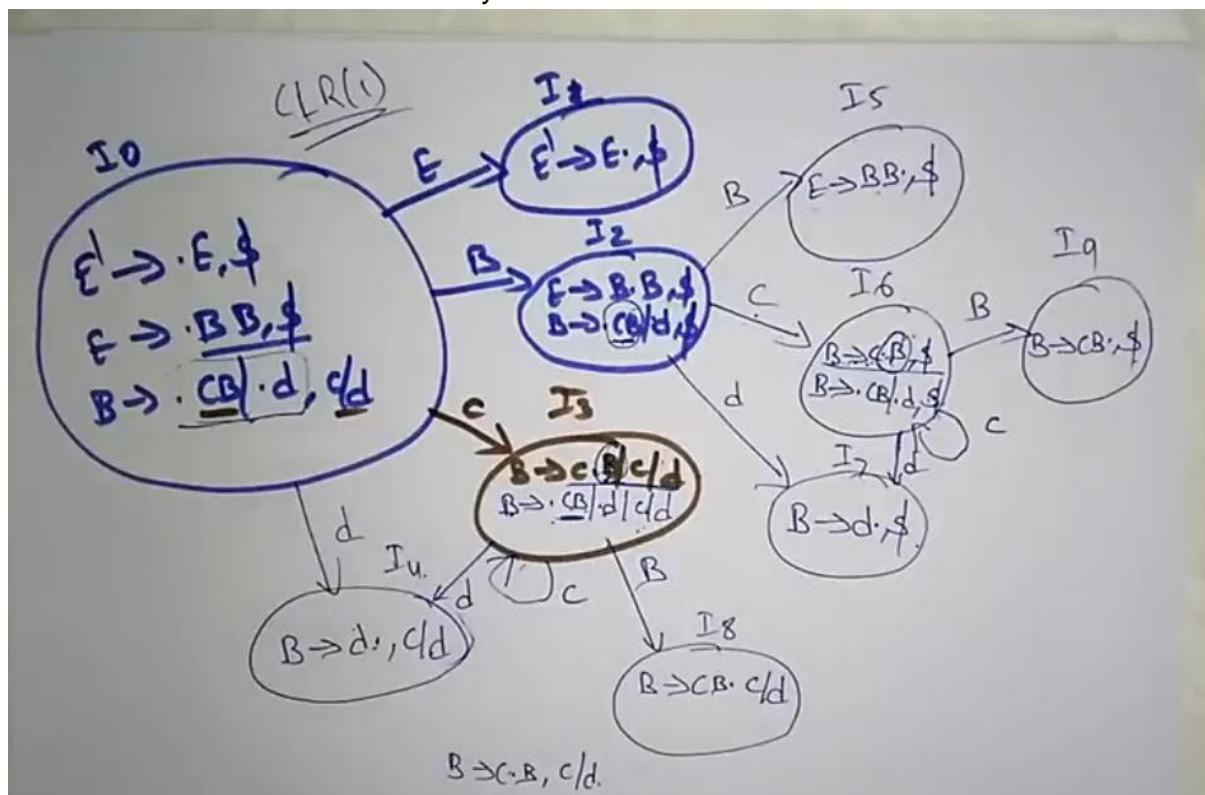
$E' \rightarrow \cdot E, \$$ — lookahead item.

$E \rightarrow \cdot B(B), \$$ — lookahead item

$B \rightarrow \cdot CB \cdot d, C/d$ — lookahead item

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.



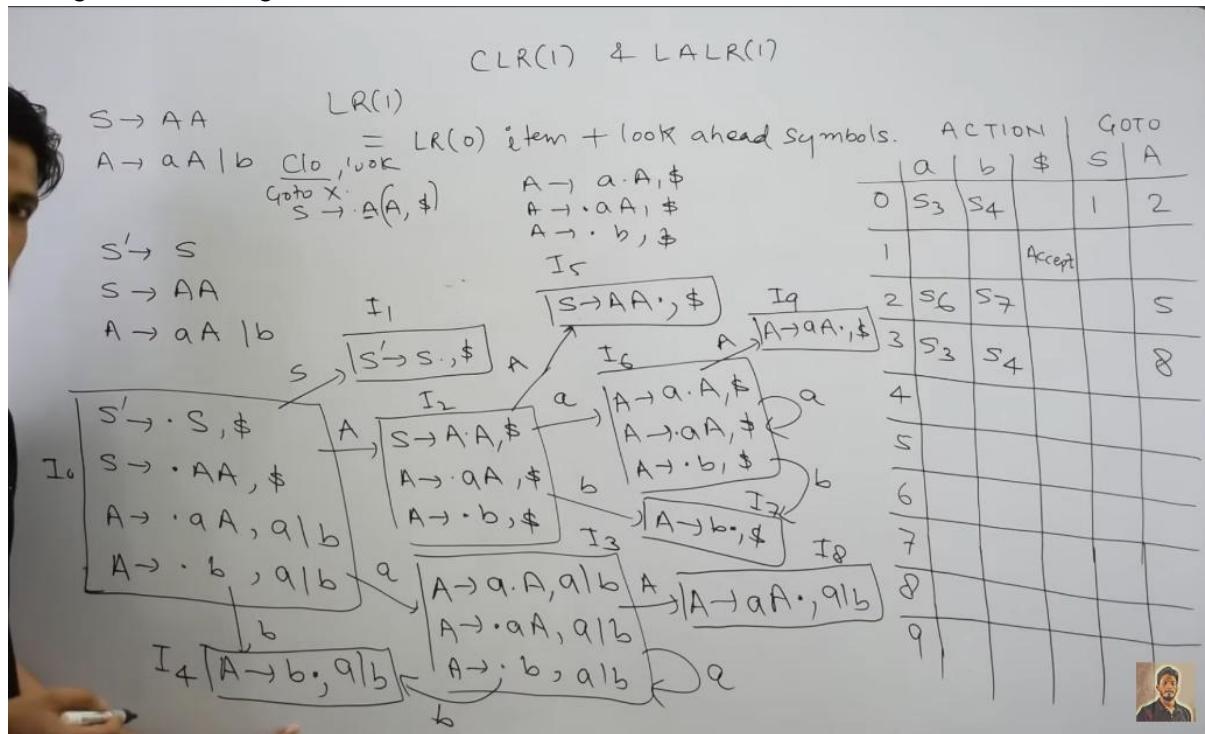
State	Action			Goto	
	c	d	\$	E	B
I ₀	s ₃	s ₄		1	2
I ₁			Accept		
I ₂	s ₆	s ₇			5
I ₃	s ₃	s ₄			8
I ₄	r ₃	r ₃			
I ₅			r ₁		
I ₆	s ₆	s ₇			9
I ₇	r ₂	r ₂			
I ₈			r ₂		

Example 2

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

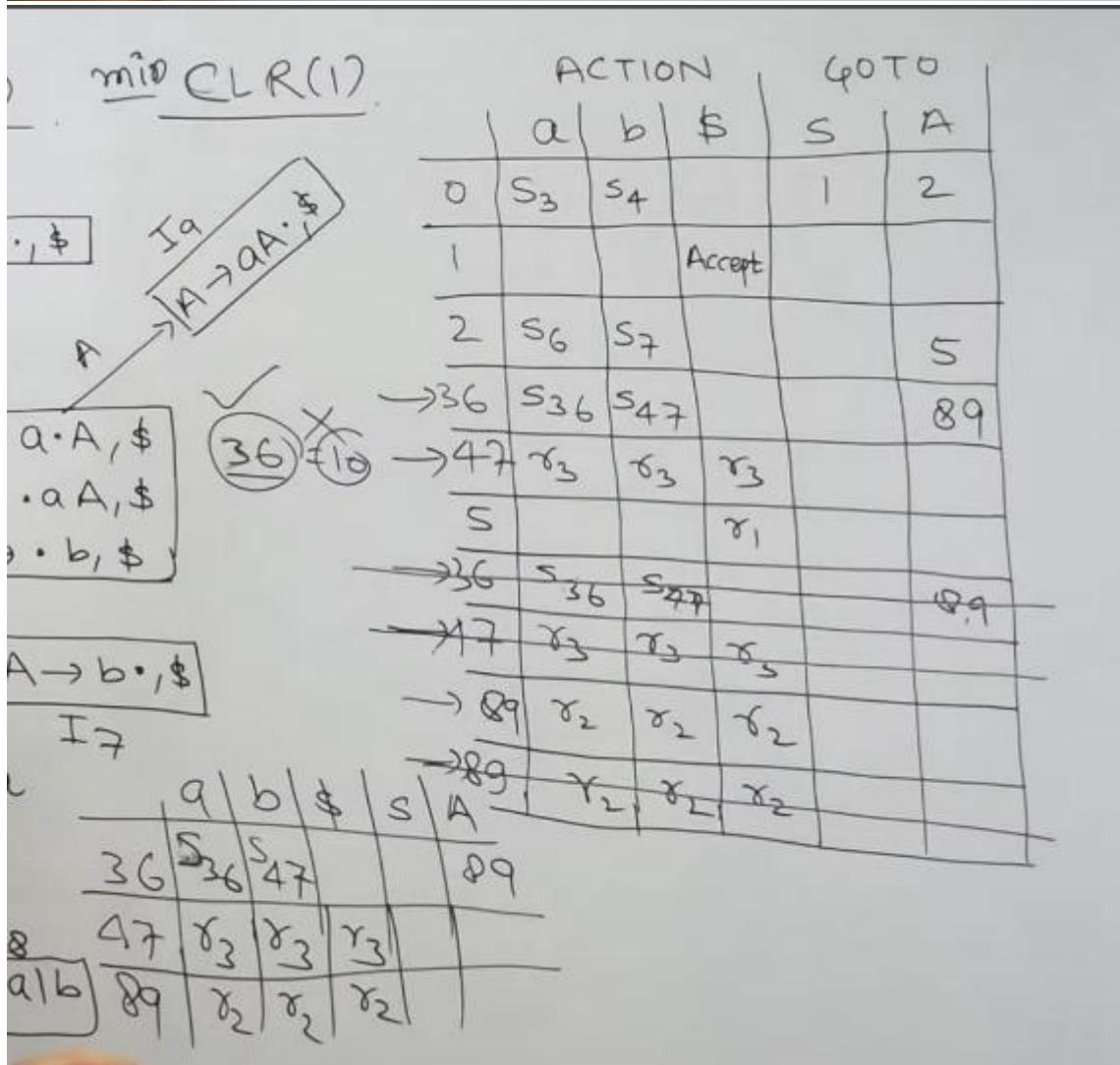
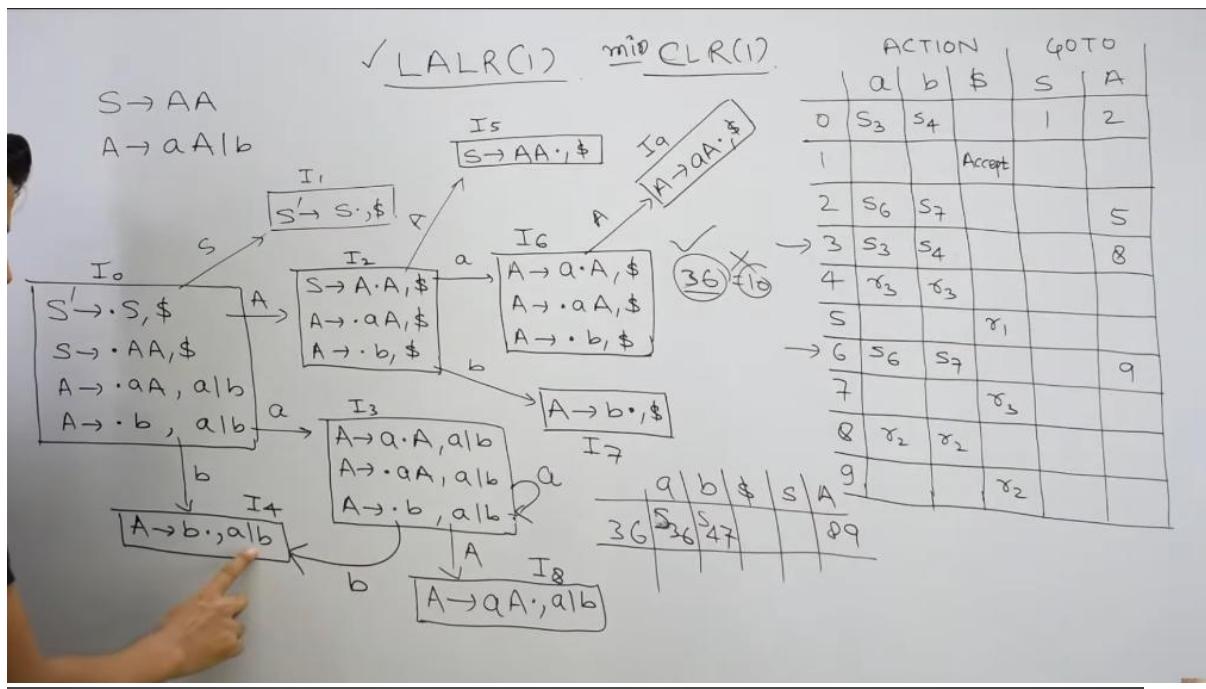
bols.	ACTION			GOTO	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			Accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5				6 ₁	
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		



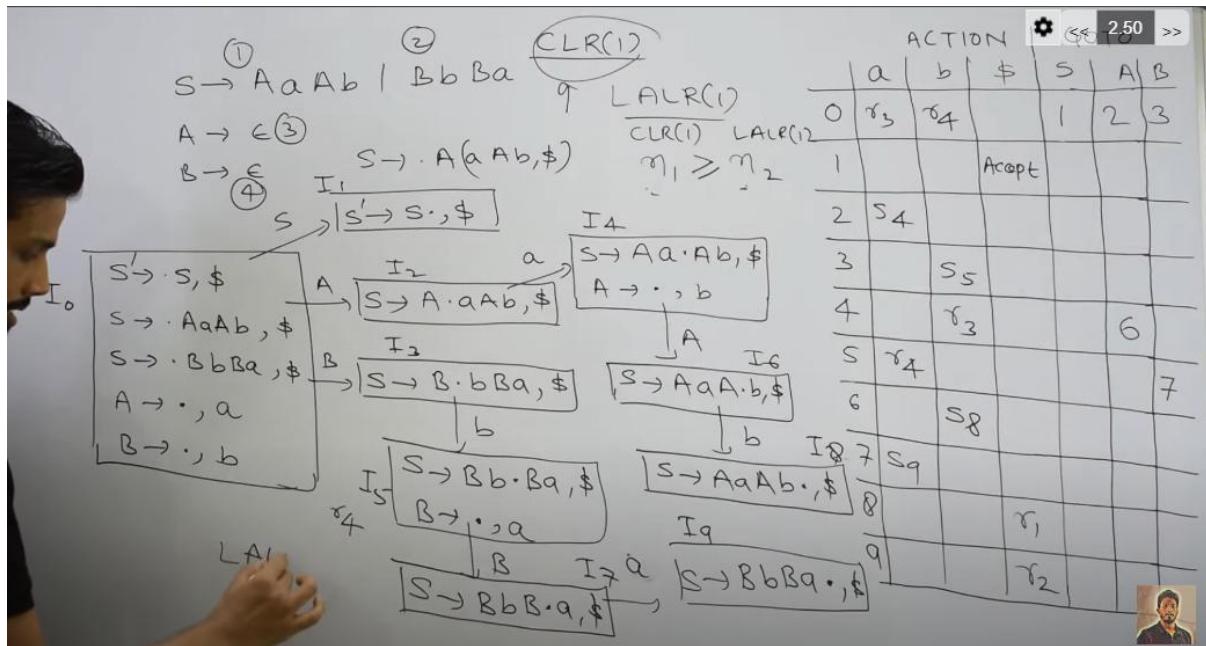
LALR(1) Parser

It's the minimised version of CLR parser.

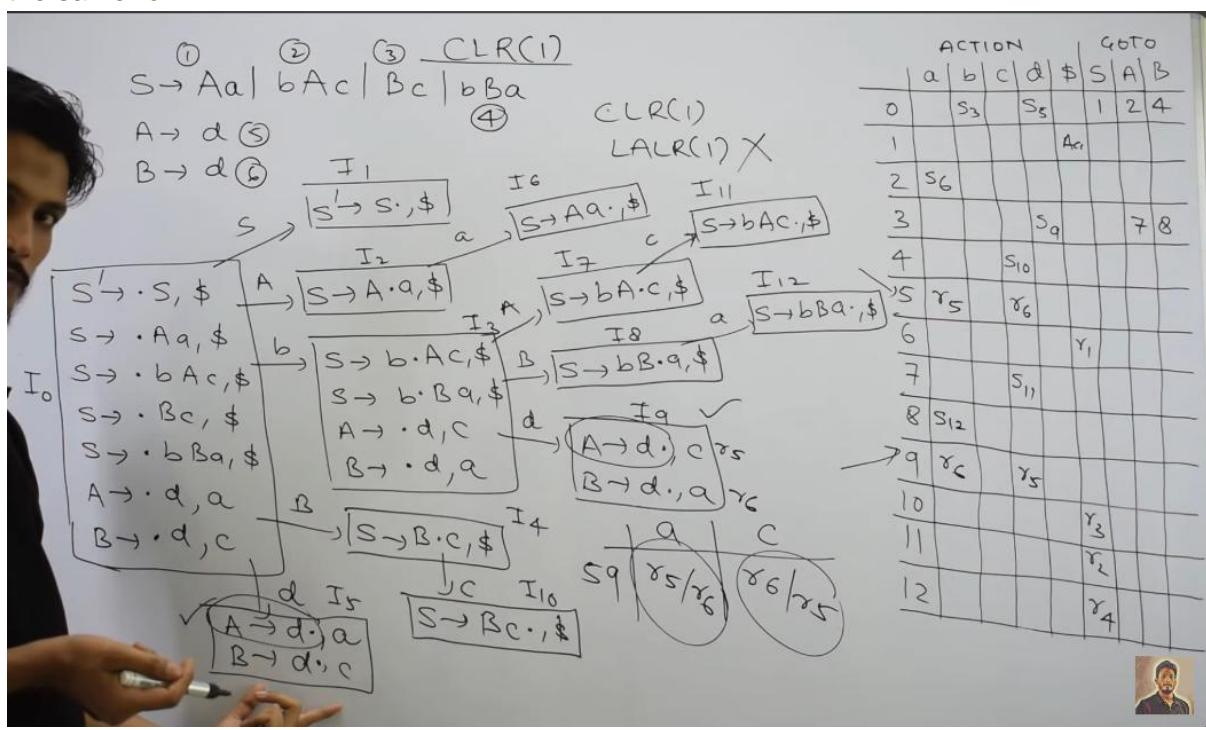
If any two states have the same LR(0) items and different lookahead symbols, then we can combine them.



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 precedence grammar.

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

- (i) no. R.H.S of any production has a ϵ .
- (ii) no two non terminals are adjacent on RHS.

eg) $E = E + E | E * E | id$ //operator grammar

eg) $S \rightarrow S A S | a$ //NOT operator grammar
 $A \rightarrow b S b | b$

we will convert it into operator grammar

- (i) no two non terminals are adjacent on RHS.
- (ii) no production has a ϵ .

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

eg) $S \rightarrow S A S | a$ //NOT operator grammar
 $A \rightarrow b S b | b$

we will convert it into operator grammar

$S \rightarrow S b S b S | a | s b s$ //operator grammar

$A \rightarrow b S b | b$ } not required now

There are 3 operator precedence relations:
y not required now

- 1) $a > b \rightarrow$ means a has higher precedence than terminal b
or a "takes precedence over" b.
- 2) $a < b \rightarrow$ means a "yields precedence to" b
or a has lower precedence than terminal b
- 3) $a \doteq b \rightarrow$ means a has same precedence as b.

We will construct operator relation table for a given ambiguous grammar

Leading and Trailing in Operator Precedence Parsing

& Trailing in operator precedence parser | Operator precedence parser

Computation of LEADING & TRAILING Function	
Find the LEADING & TRAILING of following grammar	
$E \rightarrow E + T \mid T$	$E \rightarrow E + T \quad (\text{Lead}(E) = [+]) \quad \text{---} ①$
$T \rightarrow T * F \mid F$	$E \rightarrow T \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ②$
$F \rightarrow (E) \mid \text{id}$	$E \rightarrow E + T \quad (\text{Lead}(E) = [+]) \quad \text{---} ①$
<u>Computation of LEADING</u>	
<u>Rule-1</u> $A \rightarrow Y \quad a \quad \delta$ or single non-terminal	$E \rightarrow T \quad (\text{Not Applicable})$
$\text{Lead}(A) = \{a\}$	$E \rightarrow T * F \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ④$
<u>Rule-2</u> $A \rightarrow B \quad \alpha$ anything	$E \rightarrow T * F \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ④$
$\text{Lead}(A) \leftarrow \text{Lead}(B)$	$E \rightarrow T * F \quad (\text{Not Applicable})$
<u>Computation of TRAILING</u>	
<u>Rule-1</u> $A \rightarrow Y \quad a \quad \delta$ for single terminal non-terminal	$E \rightarrow T * F \quad (\text{Trail}(E) = [+]) \quad \text{---} ④$
$\text{Trail}(A) = \{a\}$	$E \rightarrow T * F \quad (\text{Trail}(E) \leftarrow \text{Trail}(T)) \quad \text{---} ⑤$
<u>Rule-2</u> $A \rightarrow \alpha \quad B$ anything	$E \rightarrow T * F \quad (\text{Not Applicable})$
$\text{Trail}(A) \leftarrow \text{Trail}(B)$	$E \rightarrow T * F \quad (\text{Not Applicable})$
$E \rightarrow E + T$	$E \rightarrow E + T \quad (\text{Lead}(E) = [+]) \quad \text{---} ①$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow E + T \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ②$
$E \rightarrow E + T$	$E \rightarrow E + T \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ②$
$A \rightarrow B \quad \alpha$	$E \rightarrow E + T \quad (\text{Not Applicable})$
$E \rightarrow T$	$E \rightarrow T \quad (\text{Not Applicable})$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow T \quad (\text{Not Applicable})$
$E \rightarrow T * F$	$E \rightarrow T \quad (\text{Not Applicable})$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow T * F \quad (\text{Lead}(E) = [*]) \quad \text{---} ④$
$E \rightarrow T * F$	$E \rightarrow T * F \quad (\text{Lead}(E) \leftarrow \text{Lead}(T)) \quad \text{---} ④$
$A \rightarrow B \quad \alpha$	$E \rightarrow T * F \quad (\text{Not Applicable})$
$E \rightarrow F$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow F \quad (\text{Not Applicable})$
$E \rightarrow F$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow B \quad \alpha$	$E \rightarrow F \quad (\text{Not Applicable})$
$F \rightarrow (E)$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow F \quad (\text{Not Applicable})$
$F \rightarrow (E)$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow B \quad \alpha$	$E \rightarrow F \quad (\text{Not Applicable})$
$F \rightarrow \epsilon \mid id$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow F \quad (\text{Not Applicable})$
$F \rightarrow \epsilon \mid id$	$E \rightarrow F \quad (\text{Not Applicable})$
$A \rightarrow B \quad \alpha$	$E \rightarrow F \quad (\text{Not Applicable})$
$F \rightarrow id$	$E \rightarrow id \quad (\text{Lead}(F) = \{id\}) \quad \text{---} ⑦$
$A \rightarrow Y \quad a \quad \delta$	$E \rightarrow id \quad (\text{Lead}(F) = \{id\}) \quad \text{---} ⑦$
$F \rightarrow id$	$E \rightarrow id \quad (\text{Lead}(F) = \{id\}) \quad \text{---} ⑦$
$A \rightarrow B \quad \alpha$	$E \rightarrow id \quad (\text{Lead}(F) = \{id\}) \quad \text{---} ⑦$
$F \rightarrow id$	$E \rightarrow id \quad (\text{Not Applicable})$
$A \rightarrow B \quad \alpha$	$E \rightarrow id \quad (\text{Not Applicable})$

LEADING(A)

(i) If $A \rightarrow \alpha\beta$, α is single variable or ϵ
 $\text{LEADING}(A) = \{\alpha\}$

(ii) If $A \rightarrow B\alpha\beta$
 $\text{LEADING}(A) = \text{LEADING}(B)$

Ex:

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

$T \rightarrow T, S \mid S$

$\text{LEADING}(S) = \{a, \uparrow, (\}$

$\text{LEADING}(T) = \{, \text{LEADING}(S)\}$

$= \{, , a, \uparrow, (\}$

TRAILING(S)

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

$$\text{TRAILING}(A) = \{\alpha\}$$

(ii) If $A \rightarrow \alpha a \underline{\beta}$

$$\text{TRAILING}(A) = \text{TRAILING}(\beta)$$

Ex

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

$$T \rightarrow T, S \mid S$$

$$\text{TRAILING}(S) = \{a, \uparrow,)\}$$

$$\begin{aligned}\text{TRAILING}(T) &= \{, \text{TRAILING}(S)\} \\ &= \{, a, \uparrow,)\}\end{aligned}$$

Operator Precedence Parser

OPERATOR PRECEDENCE PARSER

→ Bottom up parser that interprets an operator grammar.

→ This parser is only used for operator grammar.

→ Ambiguous grammar are not allowed in any parser except operator precedence parser.

We will make operator relation table or we can say operator precedence relation table for a given grammar. for parsing

4)

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

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

$$(a+b)_c \stackrel{unary}{\sim} ab$$

id > +

} we have defined
here the
precedence rules

Two id's 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 precedence compared to any other operator.

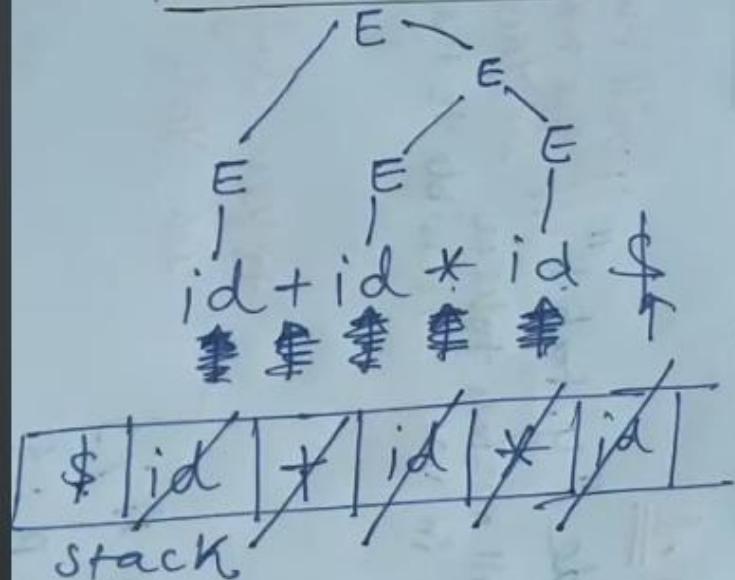
using this operator precedence table, parser will parse the PIP.

Disadvantage of operation Relation Table \rightarrow no. of entries
if we have 4 operators, we will have 16 entries
parse the PIP.

Disadvantage of operation Relation Table \rightarrow no. of entries
if we have 10 " " " " " " " " " " " " " " 100 "
for N operators $\rightarrow O(n^2)$ size of table will be v. big

top of stack

is \leftarrow , we will push else pop ie reduce
se $\boxed{id + id * id}$



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 columns g

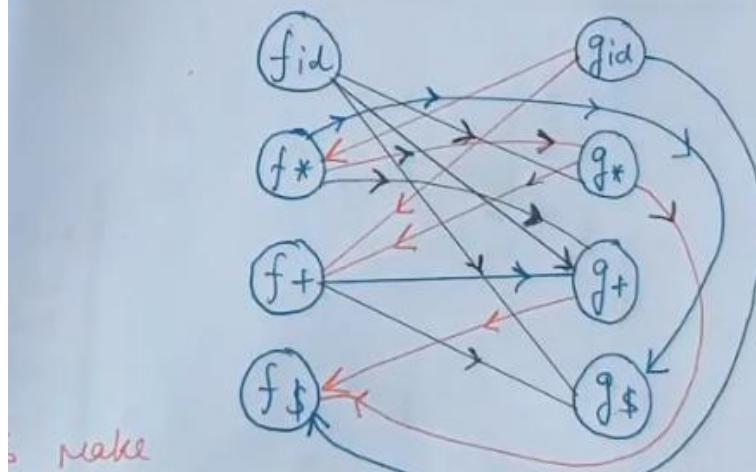
	g_{id}	+	*	\$
f_{id}	-	>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	-

Now, we will construct a graph

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

Now, we will construct a graph

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



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

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

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.

f id → g * → f + → g + → f \$

g id → f * → g * → f + → g + → f \$ (Advantage)

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

||size is less

$O(2n)$

blank entries → errors

Disadvan of using Fn Table

even though we have blank entries in relation table
we get non blank entries in fn Table

so error detecting capability of Fn Table < error
detecting capability of relation table

Operator Precedence Parser (COPP)

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.

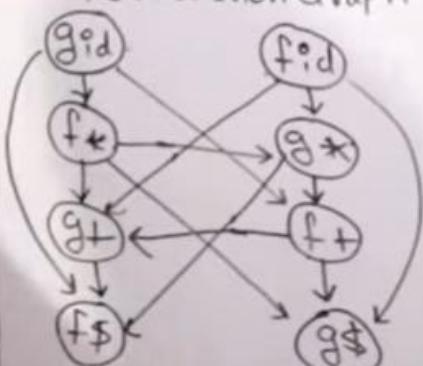
if $LHS < RHS$
then Shift
else
reduce

1. $T \rightarrow T+T \mid T*T \mid id$
 String: $id + id * id$

Step 1: OP Relation Table

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

Step 3: Function Graph



Step 2: OP table

Stack	Input	Comment
\$	id + id * id \$	Shift
\$ id	+ id * id \$	$T \rightarrow id$
\$ T	+ id * id \$	Shift
\$ T +	id * id \$	Shift
\$ T + id	* id \$	Reduce $T \rightarrow id$
\$ T + T	* id \$	Shift
\$ T + T *	id \$	Shift
\$ T + T * id	\$	$T \rightarrow id$
\$ T + T * T	\$	$T \rightarrow T * T$
\$ T + T	\$	$T \rightarrow T + T$
\$ T	\$	$T \rightarrow T + T$

Step 4: Table showing longest path

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

Operator Precedence Parse.

A, U

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

$S \rightarrow SAS \mid a$

$A \rightarrow bSb \mid b$

$S \rightarrow S \frac{bSb}{A} S \mid S \frac{bS}{A} a$

$A \rightarrow bSb \mid b$

→ Operator Grammar.

A grammar G doesn't contain

1. Null Production

2. 2- Adjacent Variables

on R.H.S of Production.

$A \rightarrow \epsilon X$ $A \rightarrow \underline{BCD} X$
 $A \rightarrow BC X$

$\underline{E \rightarrow E+E \mid E * E \mid (\epsilon)} X$



Operator Precedence Parser.

A, U

→ Operator Grammar.

$E \rightarrow E+E E * E id$	$P \rightarrow SR S$	$P \rightarrow S bSR S bS S$	A grammar G doesn't contain
	$R \rightarrow bSR bS$	$R \rightarrow b\underline{SR} bS$	1. Null Production
$S \rightarrow WbS W$		$P \rightarrow SbP Sbs S$	2. 2- Adjacent Variables
$W \rightarrow L * w L$		$R \rightarrow bP bS$	on R.H.S of Production.
$L \rightarrow id$		$S \rightarrow WbS W$	
		$W \rightarrow L * w L$	$A \rightarrow \epsilon X$
		$L \rightarrow id$	$A \rightarrow BC \bar{D} X$
			$E \rightarrow \underline{E+E} \underline{E * E} \emptyset X$

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.

Operator Precedence Parser.

A, U

$E \rightarrow E+E | E * E | id$

$\begin{matrix} id & id & id & id & \$ \\ id & - & > & > & > \\ * & < & > & > & > \\ + & < & < & > & > \\ \$ & < & < & < & - \end{matrix}$ precedence.

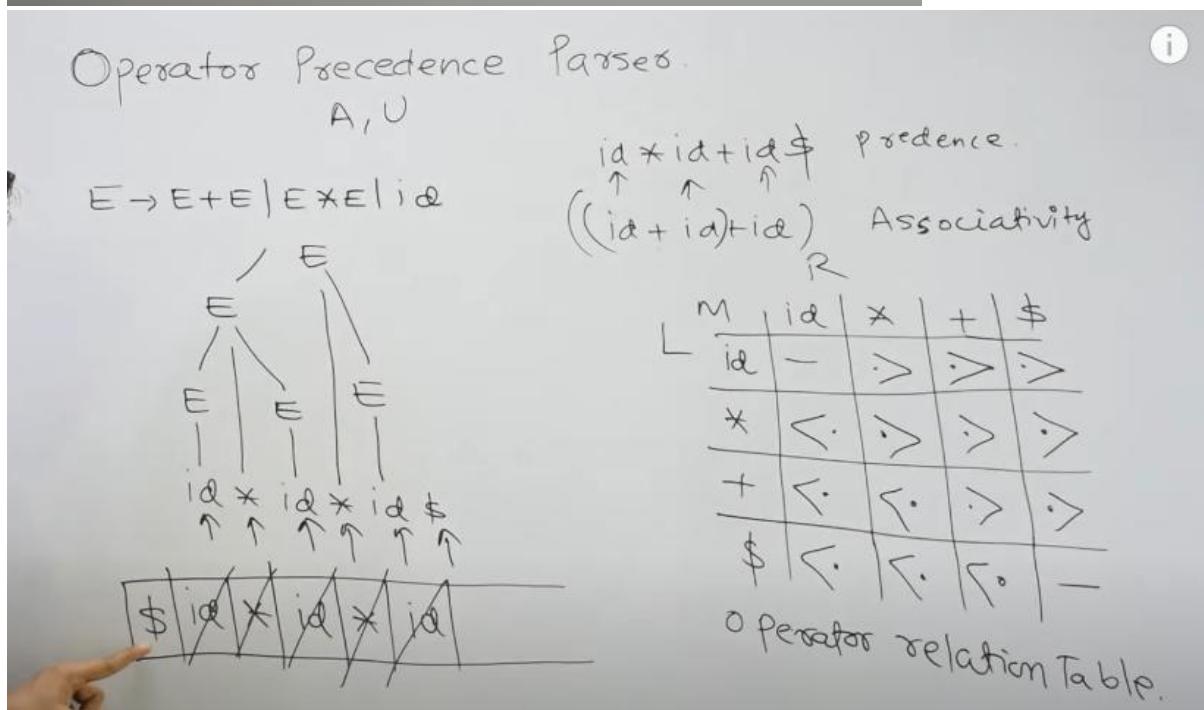
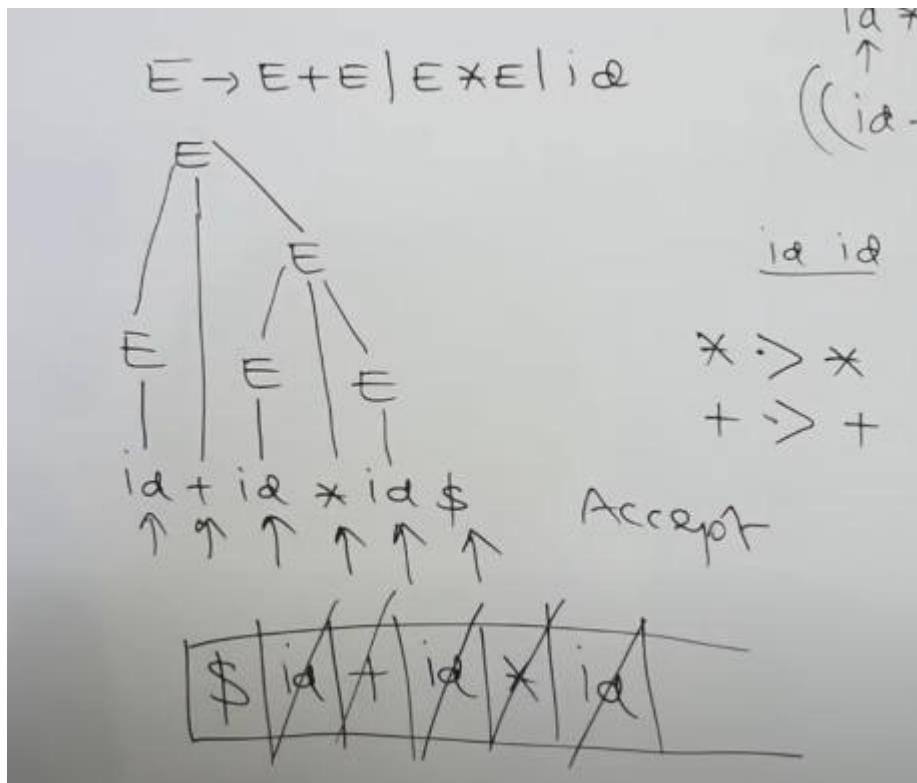
$\begin{matrix} id & id \\ * & > * \\ + & > + \end{matrix}$

M	id	*	+	\$
id	-	>	>	>
*	<	>	>	>
+	<	<	>	>
\$	<	<	<	-

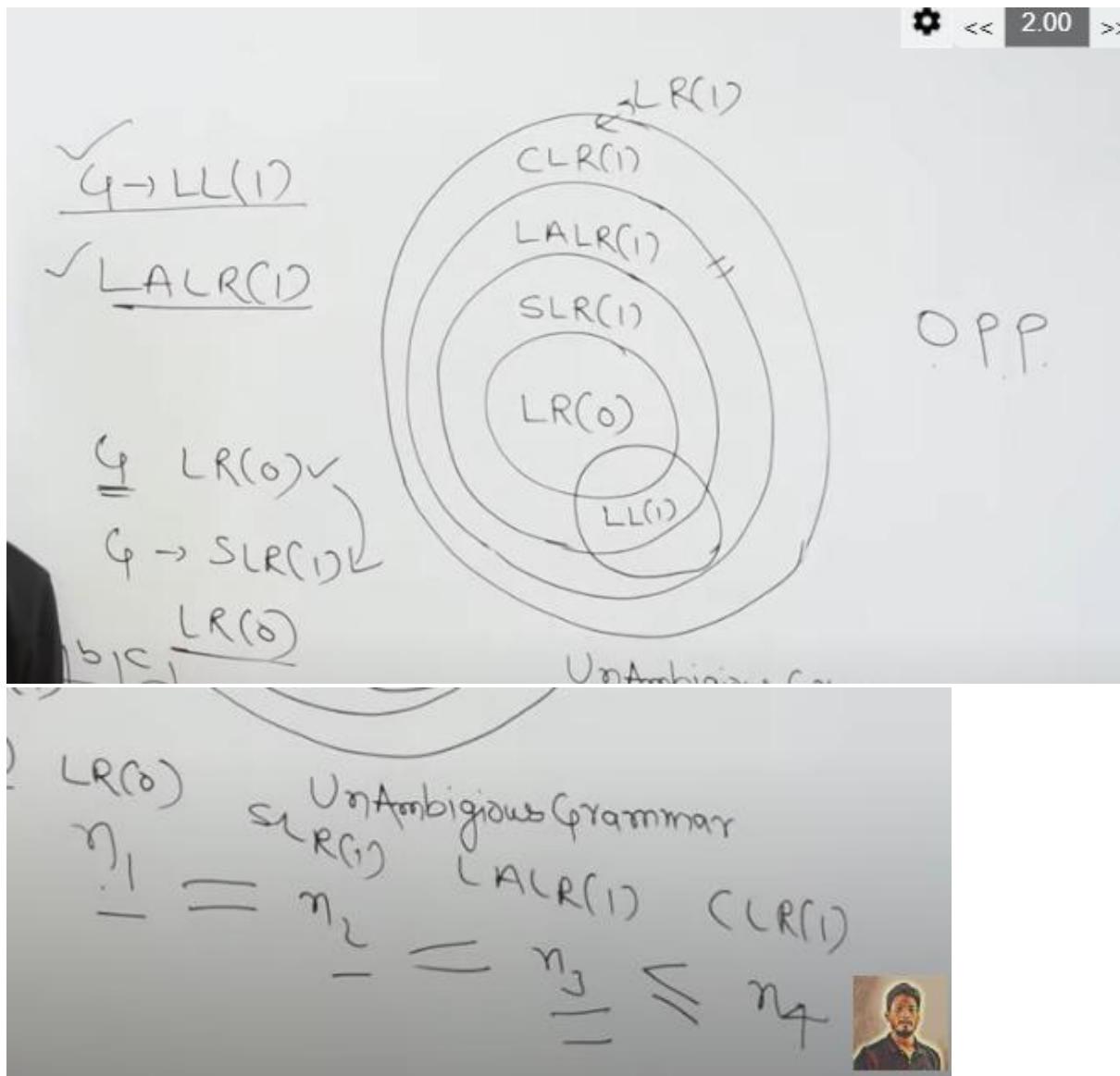
Operator relation Table.

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

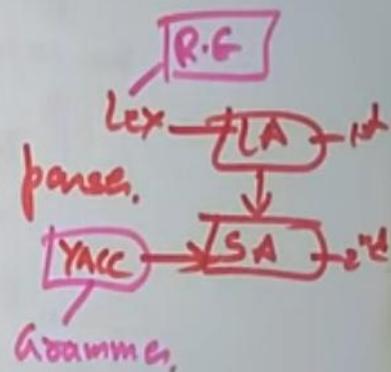
YACC \Rightarrow Yet Another Compiler Compiler

lex — Lexical Analyzer generator

YACC — Parser generator.



It is a tool which generates LALR parser.



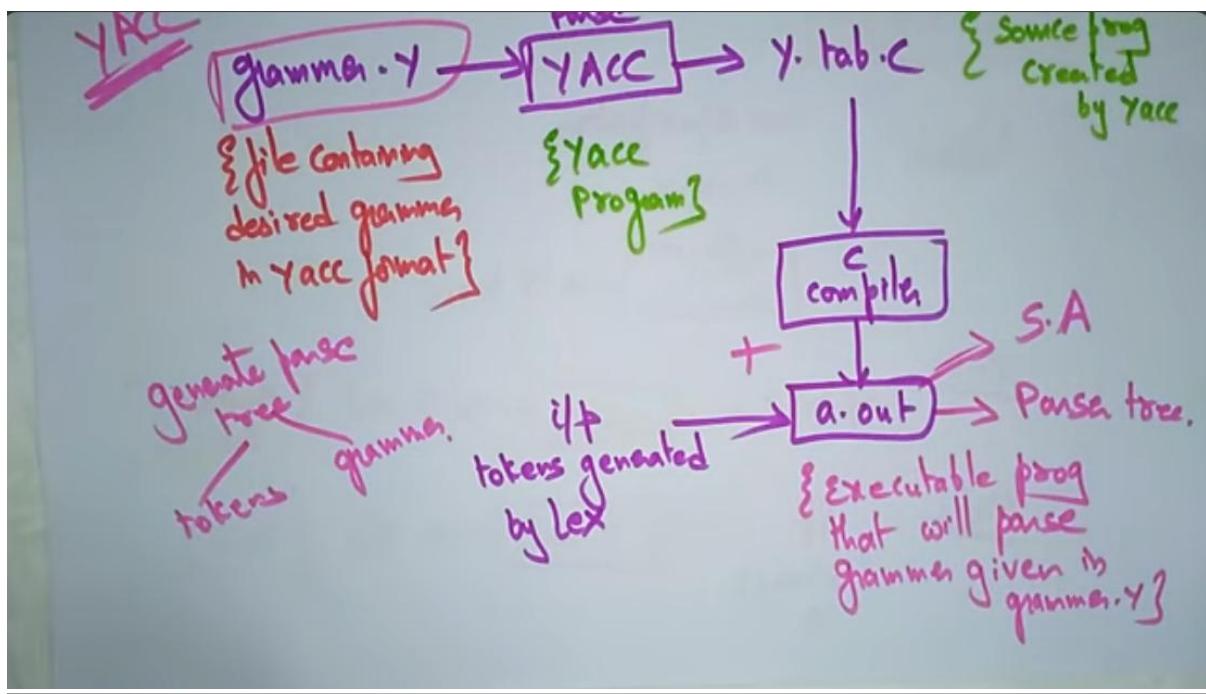
YACC Working

Step 1: Yacc Specification
Parser.y

\downarrow
Yacc compiler \rightarrow y.tab.c

Step 2: y.tab.c \rightarrow C compiler \rightarrow a.out { Syntax analyzer }

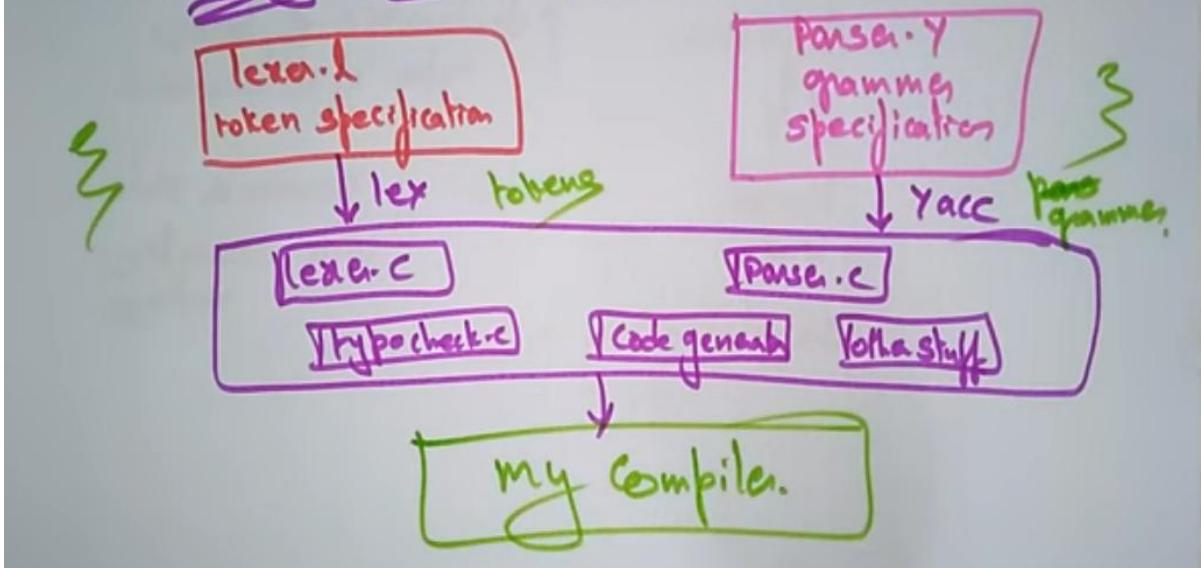
Step 3: { tokens } \rightarrow a.out \rightarrow o/p { Parse }



Syntax

definitions	{ declaration of tokens types of values used }
% %	
Rules	{ list of grammar rules with semantic routines }
% %	
Supplementary Code	

Big picture of lex/Yacc



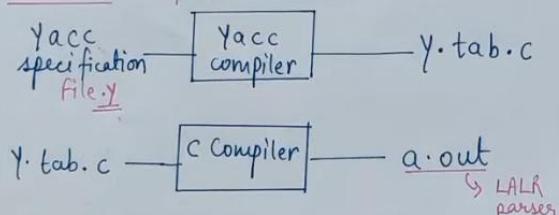
YACC IN COMPILER DESIGN

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

It is a tool for generating Look Ahead Left-to-Right (LALR) parser.

It 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

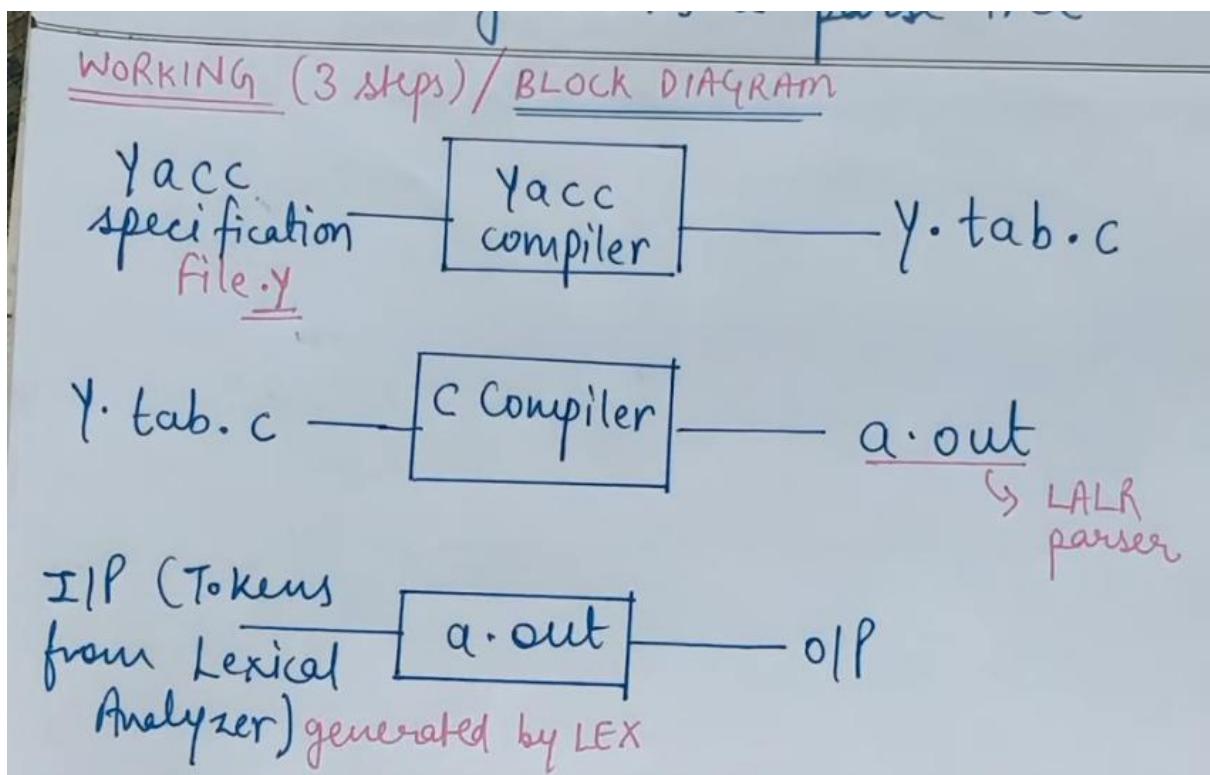


WORKING

- 1) I/P to the Yacc compiler will be a file with .y extension. 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 parser (i.e. 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

- 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

definitions | declarations
% tokens
%

Rules

head : body1 { action } | body2 { action2 }

%

Auxiliary Routines / Supplementary Code

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

To design program in Yacc lang,

1) yyval - values associated with the tokens are returned by lex in the variable yyval.
Suppose we have input a no. & we want to extract it. But our input is in character stream format.

2) yyval = atoi(yytext)
Convert string to int
i.e. converts input to numeric and stores in a variable

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

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

5) **yparse()** - responsible for parsing to occur. It reads tokens and executes the actions.
↓
if it gives 0 means string accepted

↓ which we will give in our yacc program

Ques: check aaabb is present or not?
S → a S b / ε

eg L = { ab, aabb, ... }

we have to make syntax analyzer and lexical analyzer both.

so, we will use LEX tool and YACC tool.

LEX filename.l
%{ #include "y.tab.h" // will have all the tokens used in YACC file/prog declaration in yacc accessible to the lex program %}
%{ %}

[a] { return A; } token defined in YACC file
[b] { return B; }
[ln] { return 0; }
%{ %}

/* main fn will not be included here */

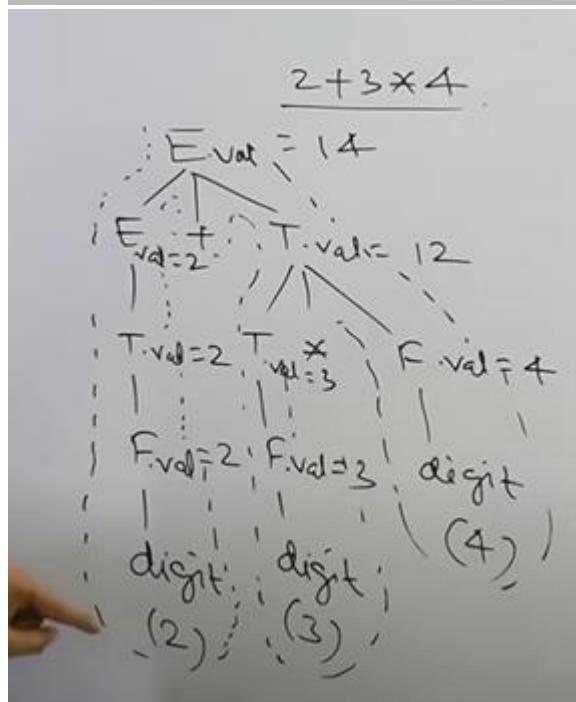
YACC part program.y
%{ #include <stdio.h>
#include <stdlib.h>
%}
%token A B // tokens defined in YACC
%{ %}
start : S 'n' { return 0; } // verification step
S: A S B
| ;
%{ %}
main()
{ printf("enter string ");
if (yparse() == 0)
printf("valid string ");
}
yyerror() // if string not accepted this is called
{ 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

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow E + T$	$E.\text{val} = E.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T * F$	$T.\text{val} = T.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lval}$

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

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



2. $10 \# 8 \# 6 \# 9 \# 4 \# 5 \# 2 = ?$

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow E \# T$	$E.\text{val} = E.\text{val} * T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T \# F$	$T.\text{val} = T.\text{val} - F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$ (GATE-2004)
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{ival}$

$\begin{array}{c} - \\ \times + \\ 8 \# 12 \# 4 \# 16 \# 12 \# 4 \# 2 \end{array}$

$8 \times 12 \times 4 \times 16 \times 12 \times 4 \times 2$ 1) If the expression $8 \# 12 \# 4 \# 16 \# 12 \# 4 \# 2$ is evaluated to 512 then which of the following is correct replacement for blank.

$8 \times (2+4) \times (6+12) \times (4+2)$

$8 \times 6 \times 2 \times 8 \times 6$ X1) $T.\text{val} = T.\text{val} * F.\text{val}$
 $= 512 \times$ X2) $T.\text{val} = T.\text{val} + F.\text{val}$

$8 \times (12-4) \times (16-12) \times (4-2)$ X3) $T.\text{val} = T.\text{val} - F.\text{val}$

$8 \times 8 \times 4 \times 2$ 4) NOT
 $= 2^3 \times 2^3 \times 2^2 \times 2^1 = 2^9 = 512$



2. $10 \# 8 \# 6 \# 9 \# 4 \# 5 \# 2 = ?$

$10 \times (8 - 6) \times (9 - 4) \times (5 - 2)$

$= 10 \times 2 \times 5 \times 3$

- = 300 ✓
 $\times +$

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

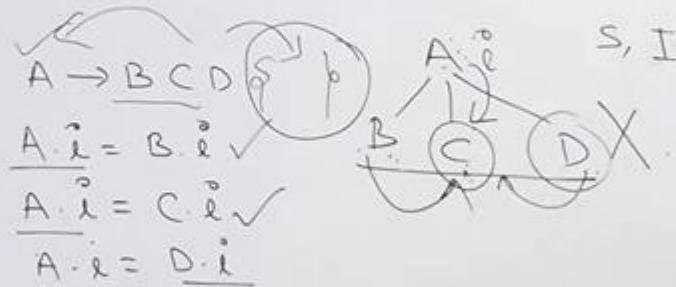
- **Synthesized Attribute:** It is an attribute whose value at a node in the parse tree is determined by the attribute values at its **child nodes**.
- **Inherited Attribute:** It is an attribute whose value at a node in the parse tree is determined by the attribute values at its **parent and/or sibling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

Types of SDD

S-Attributed

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

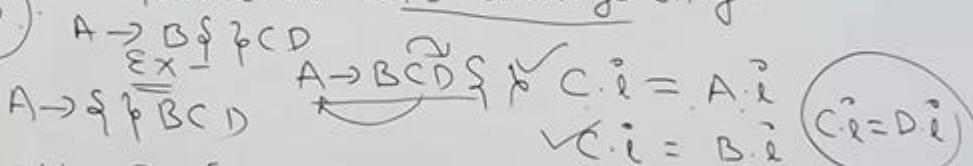
Ex -



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.



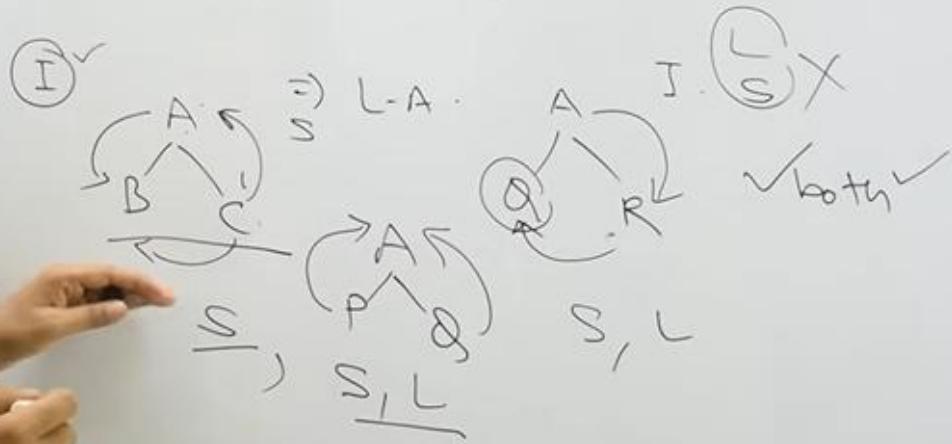
- right 2. Semantic action are placed anywhere in R.H.S of the production.

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

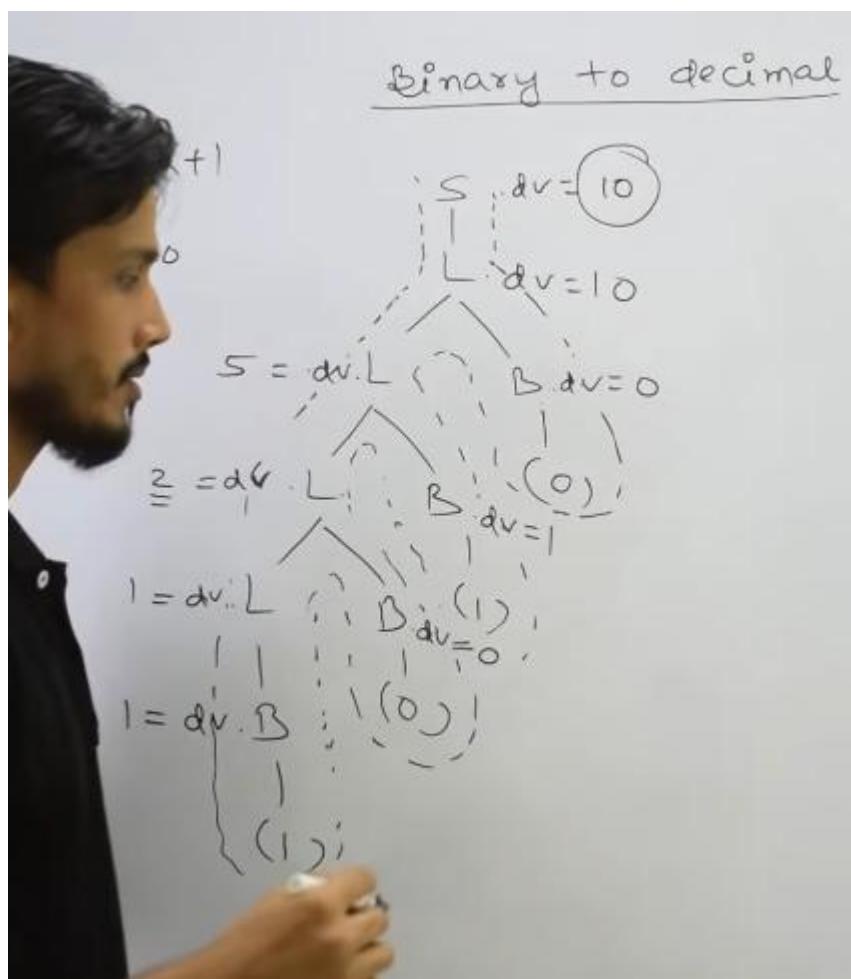
✓ 1) $A \rightarrow BC$ { $B.i = A.i$, $C.i = B.i$, $A.i = C.i$ }

X. 2) $A \rightarrow QR$ { $R.i = A.i$, $Q.i = R.i$, $A.i = Q.i$ }

✓ 3) $A \rightarrow PQ$ { $A.i = P.i$, $A.i = Q.i$ }



SDD to Convert Binary to Decimal



$$S \rightarrow L \quad \{ \quad S \cdot dv = L \cdot dv \}$$

$$L \rightarrow L \cdot B \quad \{ \quad L \cdot dv = 2 \times L \cdot dv + B \cdot dv \}$$

$$L \rightarrow B \quad \{ \quad L \cdot dv = B \cdot dv \}$$

$$B \rightarrow 0 \quad \{ \quad B \cdot dv = 0 \}$$

$$B \rightarrow 1 \quad \{ \quad B \cdot dv = 1 \}$$

$$\begin{aligned} & \text{MSD} \quad \text{LSB} \\ & (1010) = 10 \\ & 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 \\ & = 4 + 1 = 5 \end{aligned}$$

SDD to Binary to Decimal with Fraction

Binary to Decimal with fraction

(i)

$$S \rightarrow L_1 \cdot L_2 \left\{ \begin{array}{l} S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.n_b}} \end{array} \right.$$

$$L \rightarrow L \cdot B \left\{ \begin{array}{l} L.dv = 2 \cdot L.dv + B.dv \\ L.n_b = L.n_b + B.n_b \end{array} \right.$$

$$L \rightarrow B \left\{ \begin{array}{l} L.dv = B.dv \\ L.n_b = B.n_b \end{array} \right. \quad \begin{matrix} 101 \cdot 101 \\ \downarrow \quad \downarrow \end{matrix}$$

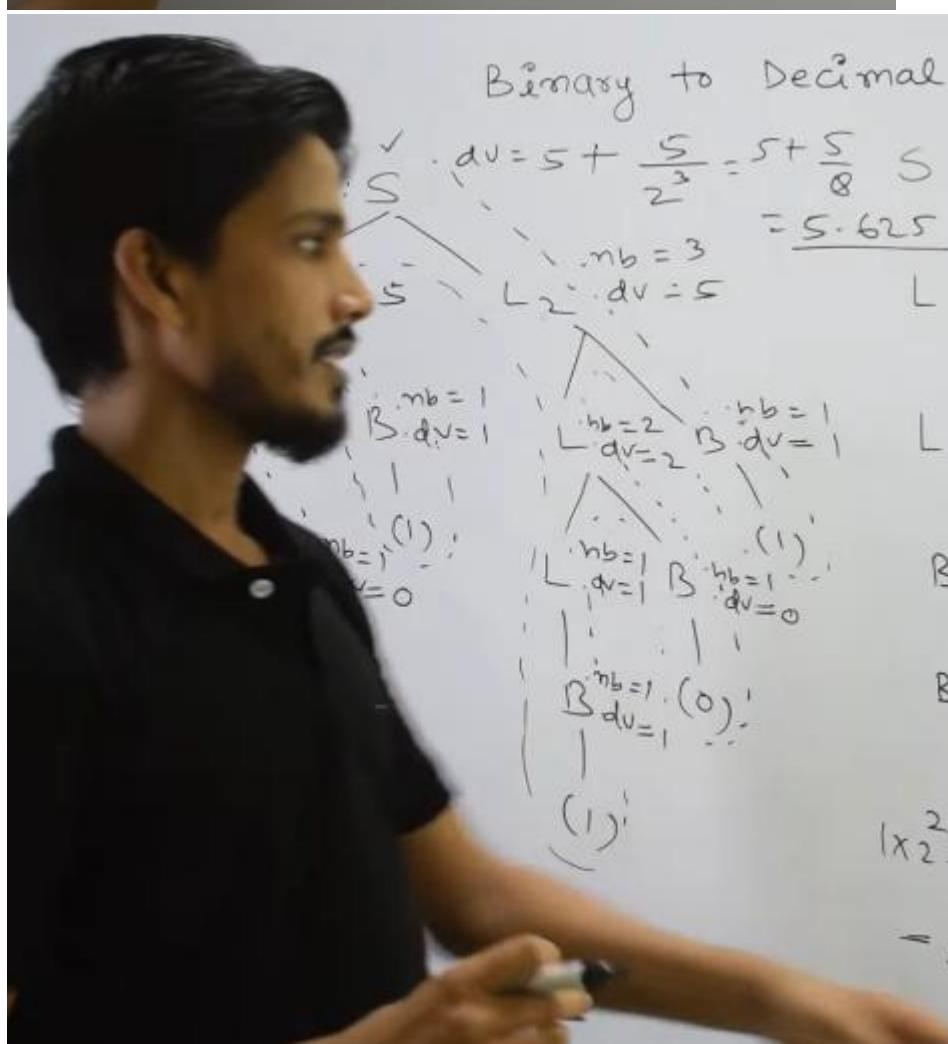
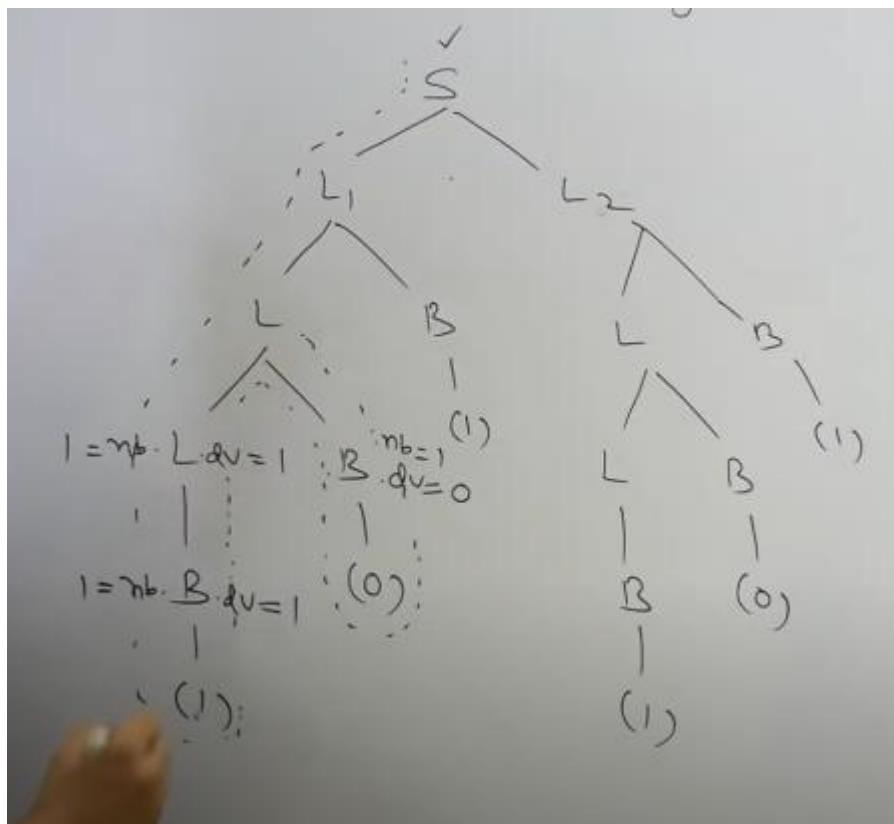
$$B \rightarrow 0 \left\{ \begin{array}{l} B.dv = 0 \\ B.n_b = 1 \end{array} \right. \quad S + \frac{S}{2^3}$$

$$B \rightarrow 1 \left\{ \begin{array}{l} B.dv = 1 \\ B.n_b = 1 \end{array} \right. \quad \begin{matrix} = S + \frac{S}{2^3} \\ = S + \frac{S}{8} \\ = S + \frac{S}{64} \\ = S.625 \end{matrix}$$

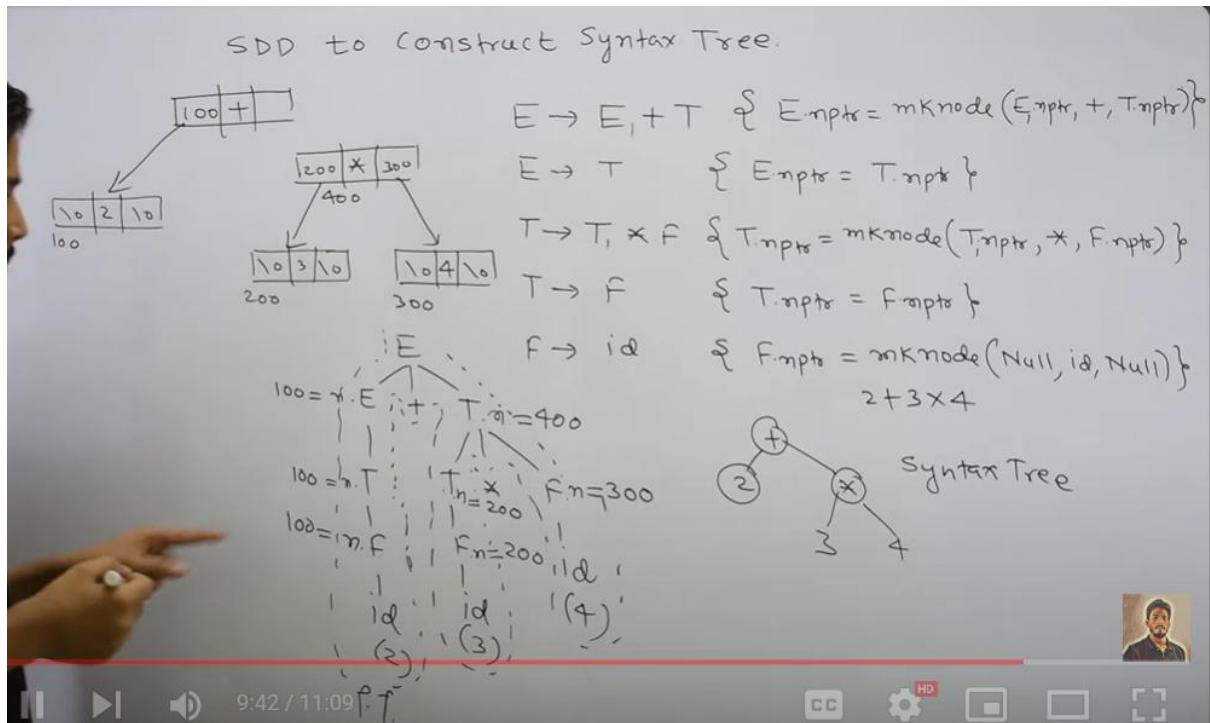
$$\begin{matrix} (101) \cdot (101) \\ 1x_2^2 + 1x_2^0 \\ = S.625 \end{matrix} \quad \begin{matrix} 0 \times \frac{1}{2} \rightarrow \\ 1 \times \frac{1}{2} + 1 \times \frac{1}{2^2} \\ = \frac{S}{8} = 0.625 \end{matrix}$$



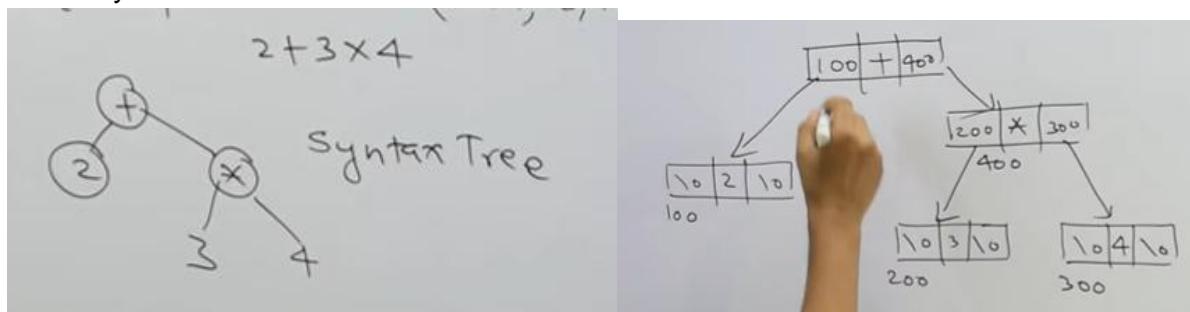
nb = number of bits



SDD to construct syntax tree



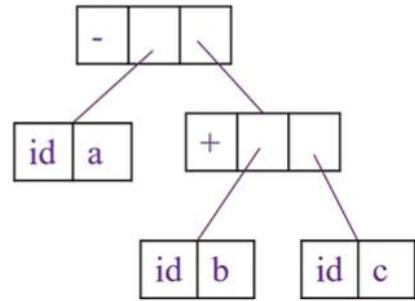
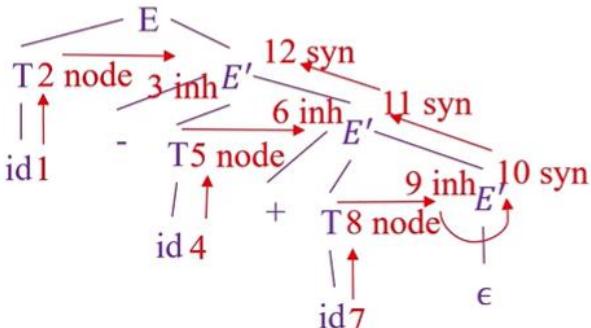
Here E1 and T1 denotes the right hand side production and 100,200 and 300 are just randomly assumed addresses for those locations.



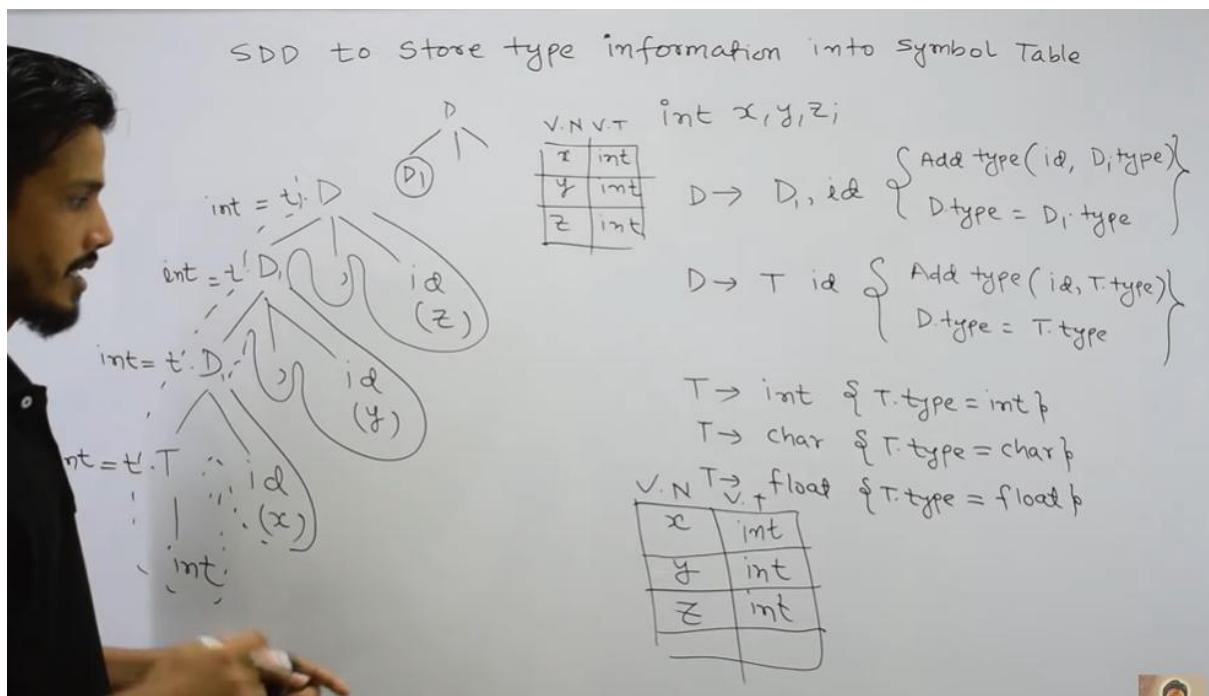
• Ex 8: Syntax tree construction

• Production Semantic Rules

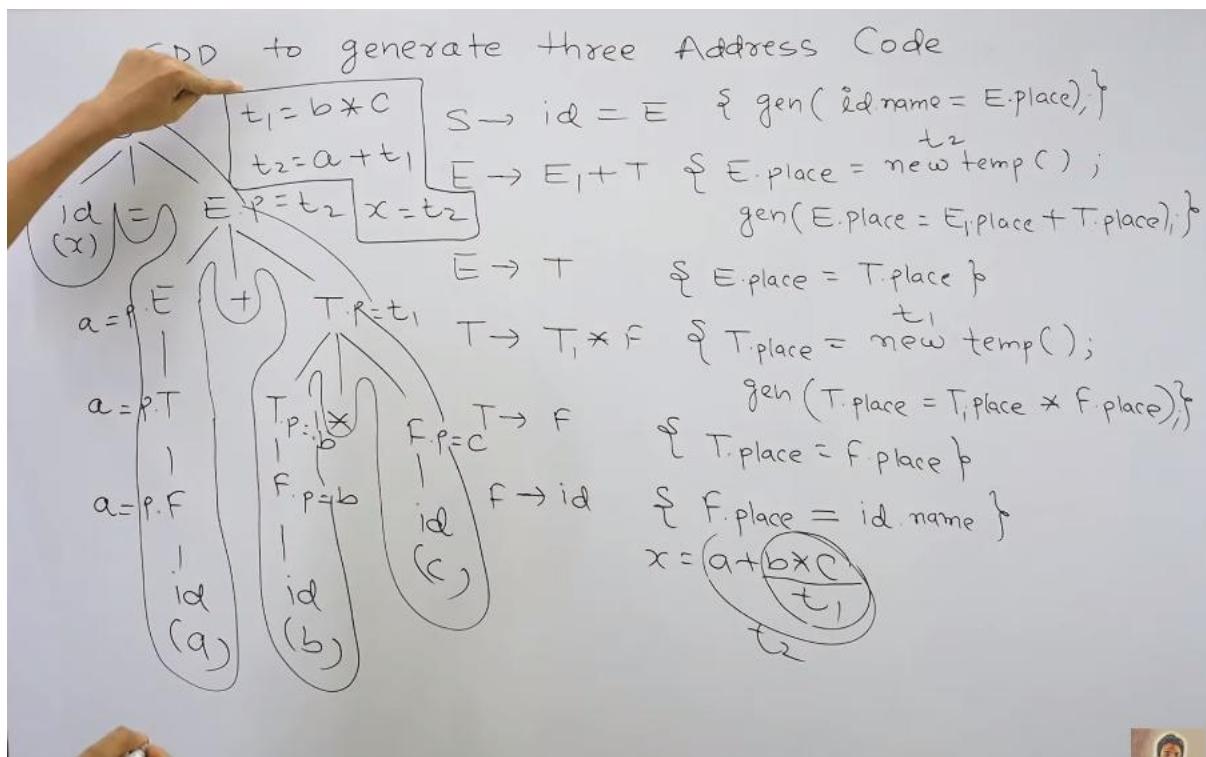
- $E \rightarrow TE'$ {E.node=E'.syn , E'.inh=T.node }
- $E' \rightarrow +TE'_1$ { $E'_1.inh=new\ Node('+', E'.inh, T.node), E'.syn= E'_1.syn$ }
- $E' \rightarrow -TE'_1$ { $E'_1.inh=new\ Node('-', E'.inh, T.node), E'.syn= E'_1.syn$ }
- $E' \rightarrow \epsilon$ { $E'.syn= E'_1.inh$ }
- $T \rightarrow (E)$ {T.node=E.node}
- $T \rightarrow id$ {T.node=new Leaf(id , id.val)}
- Inherited attribute inh ,Synthesized attribute syn.L-Attributed SDT
- Syntax tree for a-b+c.

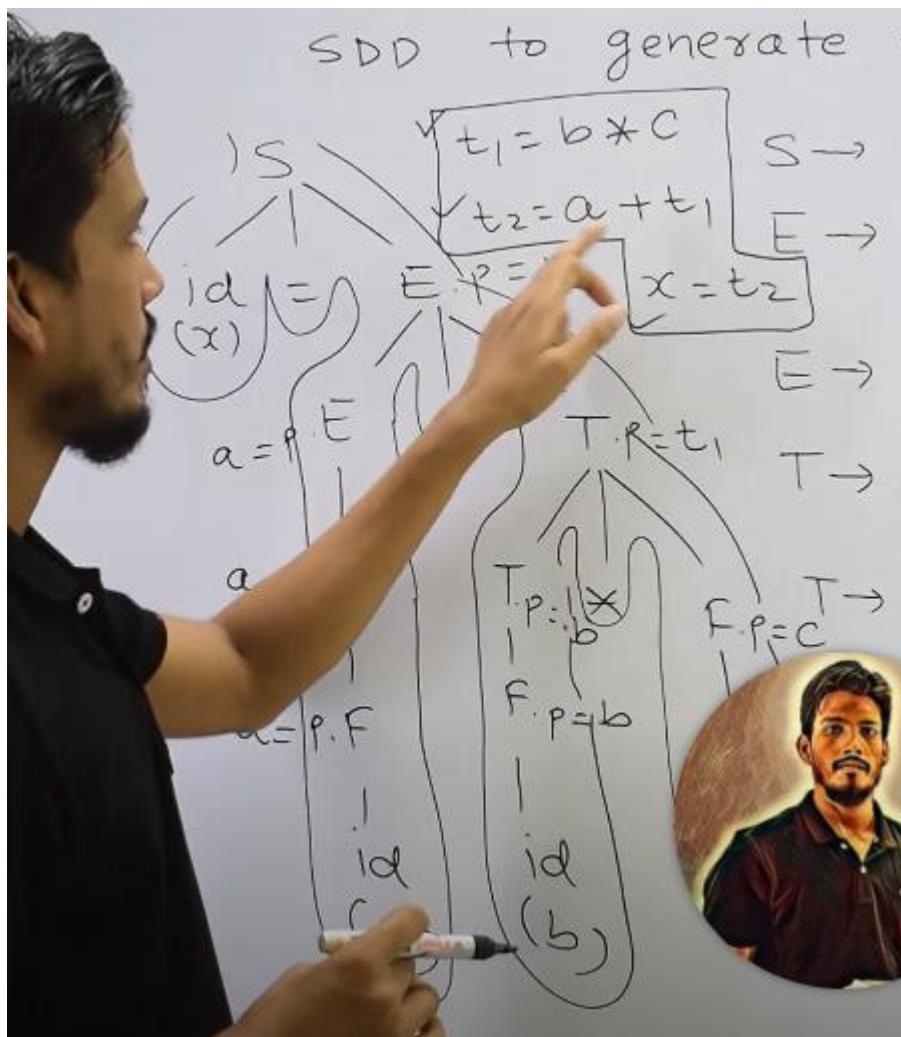


SDD to store type information in symbol table

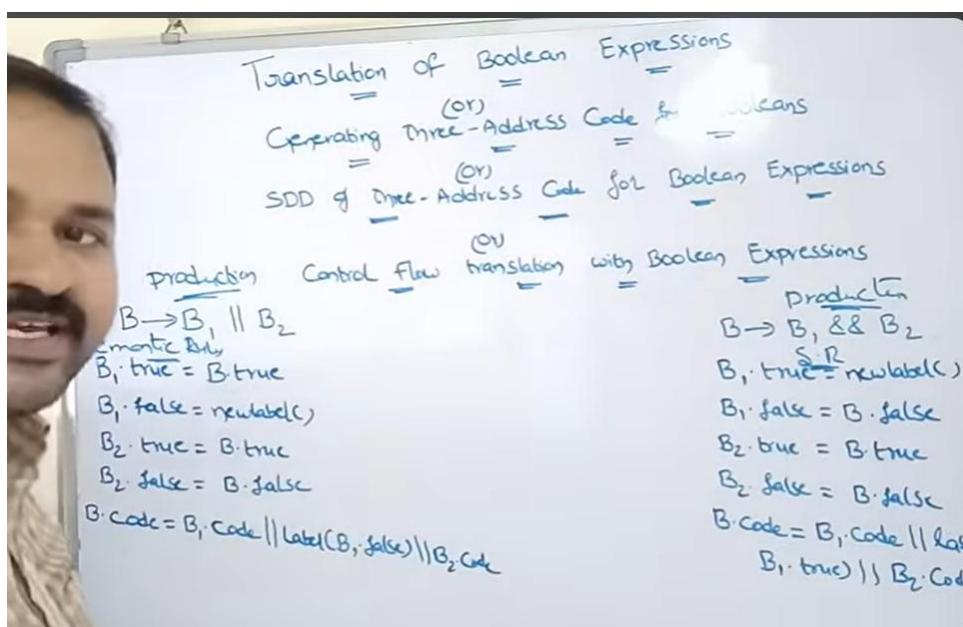


SDD to generate Three Address Code





SDD to generate 3 address code for booleans



production

$$B \rightarrow !B,$$

$$B \rightarrow \text{true}$$

Semantic Rule

$$B, \cdot \text{true} = B, \cdot \text{false}$$

$$B, \cdot \text{false} = B, \cdot \text{true}$$

$$B, \cdot \text{Code} = B, \cdot \text{Code}$$

$$B, \cdot \text{Code} = \text{generate('goto' } B, \cdot \text{true})$$

Syntax Directed Translation | SDT

Syntax Directed Translation (SDT)



Grammar + Semantic rule = SDT

↓
informal notations

- ★ In SDT, Every non-terminal can get one more alt
- ★ In semantic rule, alt $\xrightarrow{\text{VAL}}$ strong
 $\xrightarrow{\text{no:}}$
 $\xrightarrow{\text{mem loc}}$

eg SDT Production

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow \text{Num}$$

{Action} Semantic rule

$$E\text{-Val} := E\text{-Val} + T\text{-Val}$$

$$E\text{-Val} := T\text{-Val}$$

$$T\text{-Val} := T\text{-Val} * F\text{-Val}$$

$$T\text{-Val} := F\text{-Val}$$

$$F\text{-Val} := \text{num. lexical}$$

↓
att return by
LA.

Syntax Directed Translation scheme $SDT = \text{Grammar} + \text{Semantic rule}$

- * The Syntax Directed Translation scheme is a context free grammar.
- * 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 1

Production

$$S \rightarrow E\$$$

$$E \rightarrow E+E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow I$$

$$I \rightarrow I \text{ digit}$$

$$I \rightarrow \text{digit}$$

augment grammar.

Actions

Semantic Rules

$$\{ \text{Point } E \cdot \text{Val} \}$$

$$\{ E \cdot \text{Val} := E \cdot \text{Val} + E \cdot \text{Val} \}$$

$$\{ E \cdot \text{Val} := E \cdot \text{Val} * E \cdot \text{Val} \}$$

$$\{ E \cdot \text{Val} := E \cdot \text{Val} \}$$

$$\{ E \cdot \text{Val} := I \cdot \text{Val} \}$$

$$\{ I \cdot \text{Val} := 10 * I \cdot \text{Val} + \text{LexVal} \}$$

$$\{ I \cdot \text{Val} := \text{LexVal} \}$$

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 2

grammar.

Actions { Semantic rules }

~~E → E + T~~ {E.value = E.val + T.val}

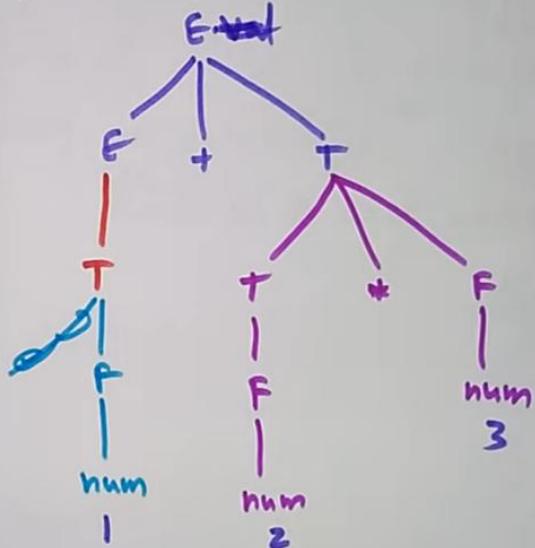
E → T {E.val := T.val}

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: $1 + 2 * 3$

$E \rightarrow E + T$
 $T \rightarrow T * F$

$$E \rightarrow E + T \rightarrow E + T \cdot F \rightarrow E + T \cdot F \cdot F \rightarrow 1 + 2 * 3 = 7$$

$$E.\text{val} = +$$

$$= 7$$

$$T.\text{val} = 2 * 3 = 6$$

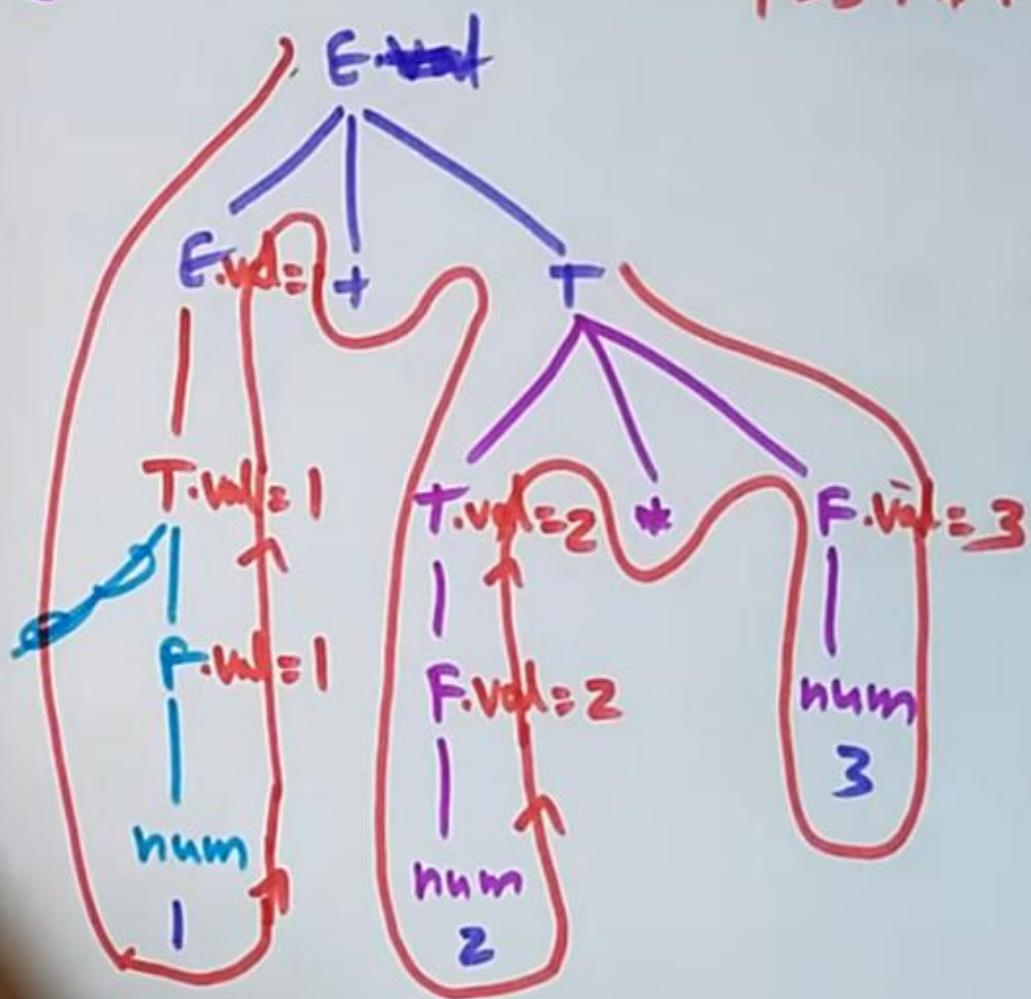
$$T.\text{val} = 1$$

$$T.\text{val} = 2$$

$$F.\text{val} = 3$$

Eg: $1 + 2 * 3$

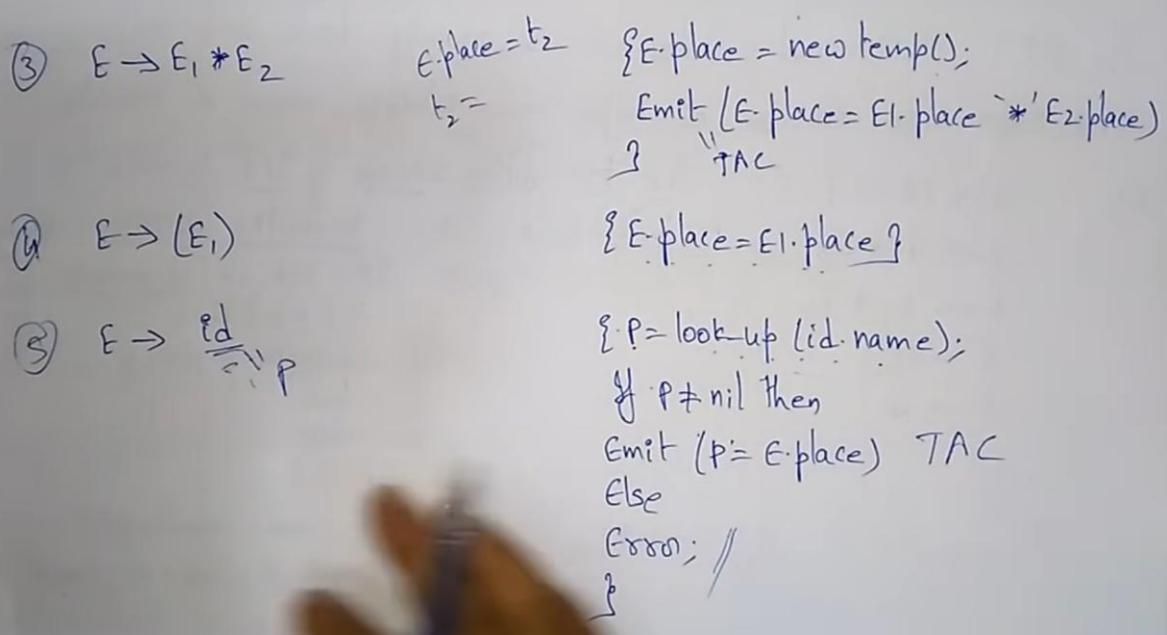
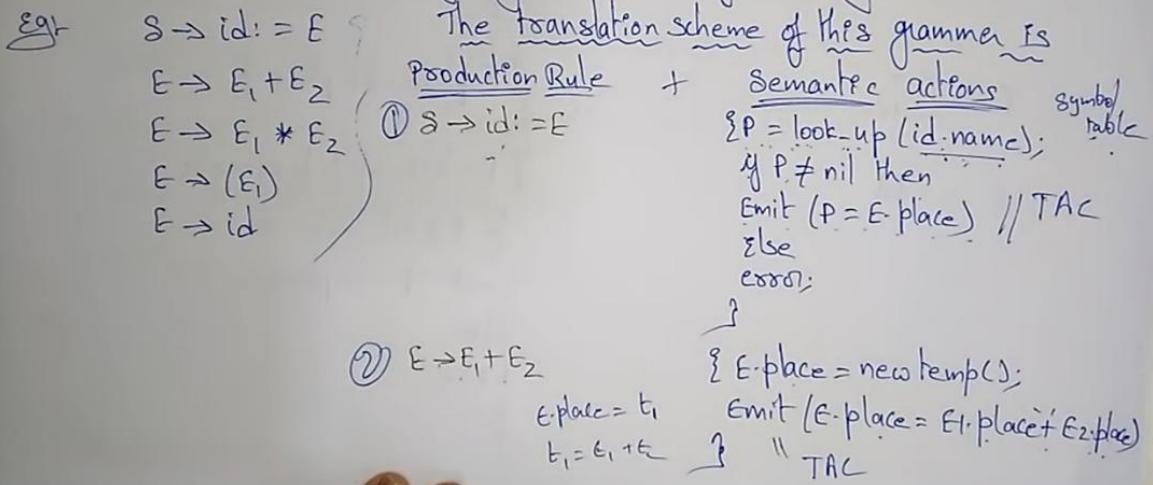
$T \rightarrow T * F$



Translation of Assignment Statements

Translation of Assignment statements

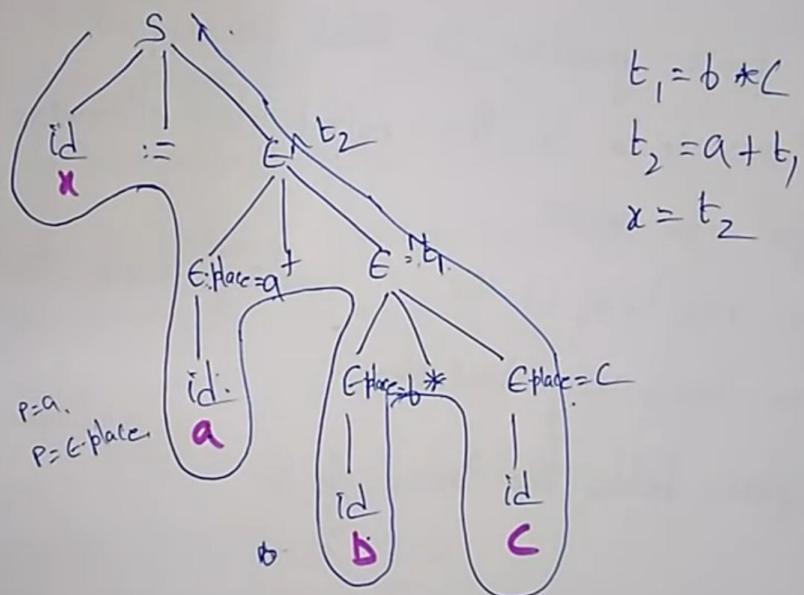
In SDT, assignment statement is mainly deals with expressions.
 The expression can be of type real, integer, array & records.



- ⇒ The p returns the entry for cd.name in the symbol table
- ⇒ the Emit function is used for appending the three address code to the output file. otherwise it will repeat an error.
- ⇒ the newtemp() is a function used to generate new temporary variables
- ⇒ E.place holds the value of E.

Example:

$$x = a + b * c$$



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,

$$\begin{array}{ll}
 E \rightarrow E \text{ OR } E & E \rightarrow \text{id relop id} \\
 E \rightarrow E \text{ AND } E & E \rightarrow \text{True} \\
 E \rightarrow \text{NOT } E & E \rightarrow \text{False} \\
 E \rightarrow (E) &
 \end{array}$$

The relop is denoted by $<$, $>$,
if-then-else & while-do

⇒ Consider a grammar,

$$\begin{array}{ll}
 1. E \rightarrow E \text{ OR } E & 5. E \rightarrow \text{id relop id} \\
 2. E \rightarrow E \text{ AND } E & 6. E \rightarrow \text{True} \\
 3. E \rightarrow \text{NOT } E & 7. E \rightarrow \text{False} \\
 4. E \rightarrow (E) &
 \end{array}$$

The relop is denoted by $<$, $>$.

relational operators

⇒ The AND & OR are left associated

⇒ NOT has higher precedence than AND & lastly OR.

Production Rule

① $E \rightarrow E_1 \text{ OR } E_2$

{ $E \cdot \text{place} = \text{new temp}();$
emit $(E \cdot \text{place}) := E_1 \cdot \text{place} \text{ 'or' } E_2 \cdot \text{place}$

② $E \rightarrow E_1 \text{ AND } E_2$

{ $E \cdot \text{place} = \text{new temp}();$
emit $(E \cdot \text{place}) := E_1 \cdot \text{place} \text{ 'AND' } E_2 \cdot \text{place}$

Production Rule

③ $E \rightarrow \text{NOT } E_1$

④ $E \rightarrow (E_1)$

⑤ $E \rightarrow \text{id}_1 \text{,op,id}_2$

Production Rule

⑥ $E \rightarrow \text{True} = 1$

⑦ $E \rightarrow \text{False} = 0$

Semantic Actions

$\{ E.place = \text{newtemp}();$
 $\text{emit}(E.place := \text{'NOT'} E_1.place)$

$\}$
 $\{ E.place = E_1.place \}$

$\{ E.place = \text{newtemp}();$
 $\text{emit}('if' id1.place \text{,op,id}_2.place$
 $\text{'goto' nextstate+3});$
 $\text{emit}(E.place := '0')$
 $\text{emit}('goto' nextstate+2);$
 $\text{emit}(E.place := '1')$

Semantic Action

$\{ E.place := \text{newtemp}();$
 $\text{emit}(E.place := '1')$

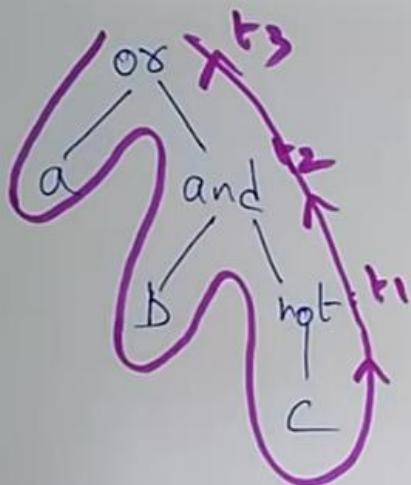
$\}$

$\{ E.place := \text{newtemp}();$
 $\text{emit}(E.place := '0')$

$\}$

⇒ Emit generates TAC & newtemp() generates temporary variable
 ⇒ $E \rightarrow \text{id}_1 \text{,op,id}_2$ contains next state & it gives the index
 of next three address stm in the output sequence.

Eg: $a \text{ or } b \text{ and not } c$



$$\begin{aligned} t_1 &= \text{not } c \\ t_2 &= b \text{ and } t_1 \\ t_3 &= a \text{ or } t_2 \end{aligned}$$

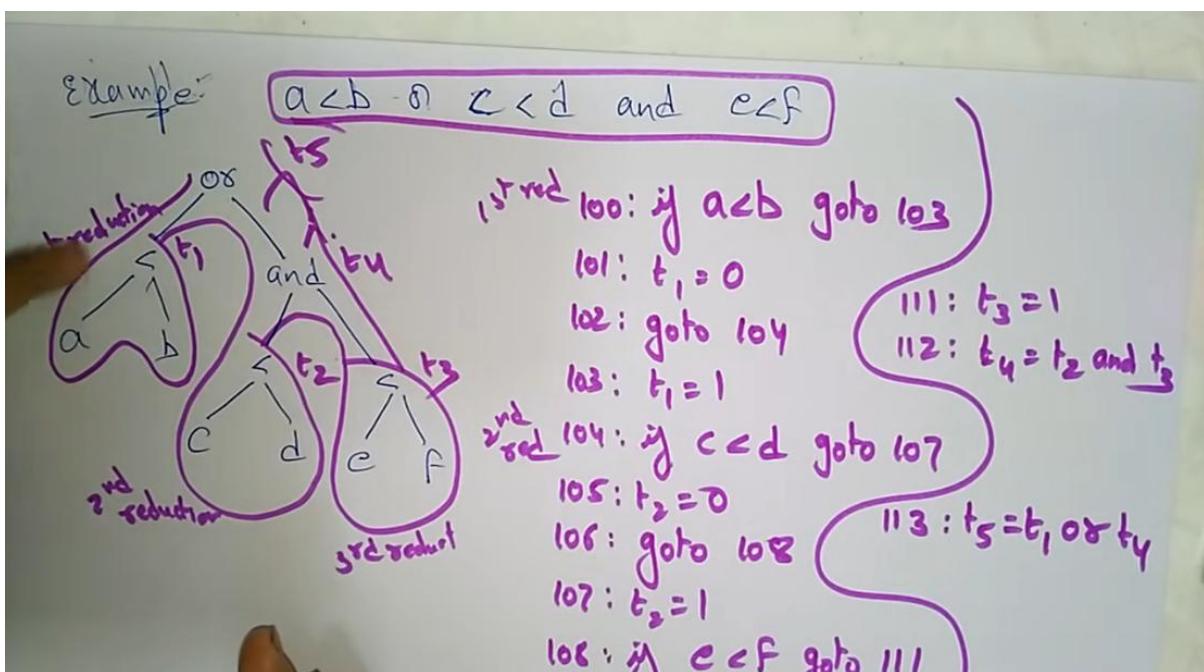
Eg: If $a < b$ Then !. Else 0

100: if $a < b$ goto 103

101: $t_1 = 0$

102: goto 104

103: $t_1 = 1$



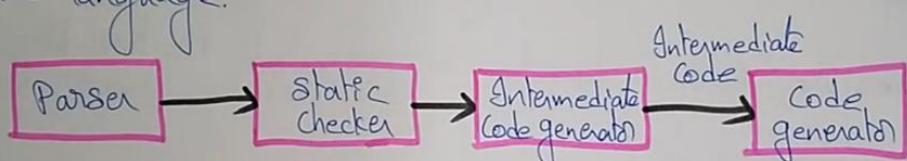
109 : $t_3 = 0$

110 : goto 112

Intermediate Code Generation

Intermediate code

It is used to translate the source code into the machine code. It lies between high level lang and machine language.



- ★ If the compiler directly translates source code into machine code without generating intermediate code then a full native compiler is required for each new machine.
- ★ Intermediate code generator receives input from its predecessor.

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 be represented as source code

low level intermediate code

↓

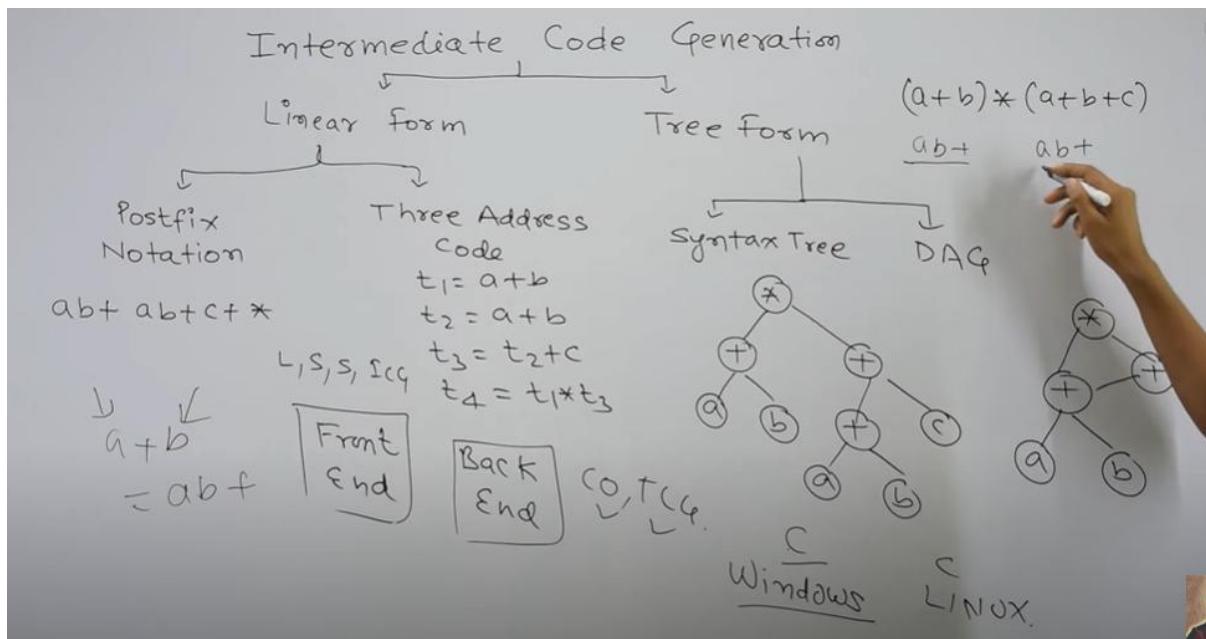
* It is close to the target machine. It is used for machine dependent optimizations.

The different forms of Intermediate code

① Abstract syntax tree

② Polish Notation

③ Three address code



Types of Three Address Code.

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

3-address code for if-then-else

if $a < b$ then $x = y + z$ else $p = z + y$

Ans

- ① if $a < b$ goto (3)
 - ② goto (6)
 - ③ $t_1 = y + z$
 - ④ $x = t_1$
 - ⑤ goto --
 - ⑥ $t_2 = z + y$
 - ⑦ $p = t_2$
 - ⑧ goto --
- true
false

3-address code for FOR LOOP

for ($i = 1$; $(i \leq 20)$; $i++$) $i = (i + 1)$

- ① $i = 1$
- ② $x = y + z$
- ③ if $(i \leq 20)$ goto (7) (true)
- ④ goto -- (false)
- ⑤ $t_1 = (i + 1)$
- ⑥ $i = t_1$
- ⑦ goto (2)
- ⑧ $t_2 = y + z$
- ⑨ $x = t_2$
- ⑩ goto (4)

3-address code for while loop

- ① if $(a < b)$ goto (2) (true)
- ② goto -- (false)
- ③ $t_1 = y + z$
- ④ $x = t_1$
- ⑤ goto (1)

3-address code for Boolean Expression

if $A < b$ AND $C > d$
then $P = z + y$

- ① if $A < b$ goto (2)
- ② goto -- (false)
- ③ if $C > d$ goto (2)
- ④ goto -- (false)
- ⑤ $t_1 = z + y$



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:
```

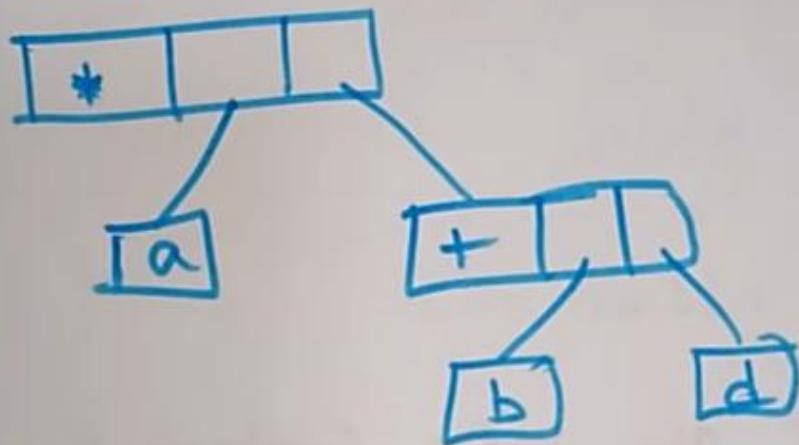
Abstract Syntax tree (AST) | Intermediate Code

⇒ ① A ST are more compact than a parse tree
& can be easily used by compiler.

Eg: $a * b + c$



AST is represented as



Polish

- ① Abstract syntax tree infix $\Rightarrow a + b$
② Polish Notation prefix $\Rightarrow + a b$
③ Three address code postfix $\Rightarrow a b +$

\Rightarrow Postfix notation is a linear representation of
syntax tree.

$$x+y \Rightarrow xy+$$

e.g. Productions

$$E \rightarrow E_1 \text{ op } E_2$$

$$E \rightarrow (E_1)$$

$$E \rightarrow id$$

Semantic rule

$$E \cdot \text{Code} = E_1 \cdot \text{Code} \parallel E_2 \cdot \text{Code} \parallel \text{op}$$

$$E \cdot \text{Code} = E_1 \cdot \text{Code}$$

$$E \cdot \text{Code} = id$$

$$\begin{aligned}
 \text{Eg:- } Z &= (A - B) * (C - D) + [E + (F/G)] \\
 Z &= AB - + CD - + [E + (F/G)] \\
 Z &= AB - + CD - + E FG / + \\
 Z &= AB - CD - * EF G / + + \\
 &\qquad\qquad\qquad \text{Postfix Notation}
 \end{aligned}$$

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

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

- Three address code IC ← AST Polish ← TAC
- * It is an intermediate code. It is used by the optimizing compilers.
 - * In three address code, the given expression is broken down into several separate instructions. These instructions can easily translate into Assembly language.
 - * Each three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

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

Eg: $x + y * z$

$$t_1 = y * z$$

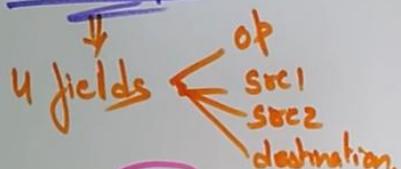
$$t_2 = x + t_1$$

t_1, t_2 are compiler generated temporary names

Egt $\frac{a + a * (b - c) + d * (b - c)}{TAC}$

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= d * t_1 \\ T_5 &= t_3 + t_4 \end{aligned}$$

① Quadruples



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

TAC:
 $t_1 = -b$
 $t_2 = c + d$
 $t_3 = t_1 * t_2$
 $a := t_3$

Quadruples

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

② Triples

\downarrow
 3 fields \leftarrow operator
 src_1
 src_2

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

TAC:

$$\begin{aligned} t_1 &= -b \\ t_2 &= c + d \\ t_3 &= t_1 + t_2 \\ a &:= \underline{t_3} \end{aligned}$$

Quadruples

	<u>operator</u>	<u>Source 1</u>	<u>Source 2</u>	<u>result</u>	<u>triples</u>
(0)	uminus	b	-	t ₁	(0) - b
(1)	+	c	d	t ₂	(1) + c d
(2)	*	t ₁	t ₂	t ₃	(2) * (0) (1)
(4)	:=	t ₃	-	a	(4) := (2) -

$t_1 := -b$

In triples take the references from the quadruples.

Representation of Three Address Code

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

$$t_1 = a * b$$

$$t_2 = -t_1$$

$$t_3 = c * d$$

$$t_4 = t_3 + e$$

$$t_5 = t_2 + t_4$$

Quadruples				
operator	op1	op2	result	
0 *	a	b	t1	
1 -	t1		t2	
2 *	c	d	t3	
3 +	t3	e	t4	
4 +	t2	t4	t5	

Triples

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

2, 0, 1 ✓MOVE
extra
space

X MOVE
less space



Indirect Triples

instruction

xd	100	(0)	(2)
b	101	(1)	(0)
	102	(2)	(1)
	103	(3)	
	104	(4)	



Construct Quadruples, triples, Indirect triples for
the statement $(a+b) * (c+d) - (a+b+c)$.

Quadruple

	OP	arg1	arg2	Result
(0)	+	a	b	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	+	t1	c	t4
(4)	-	t3	t4	t5

	OP	arg1	arg2
(3)	+	(0)	c
(4)	-	(2)	(3)

$$t1 = a + b$$

$$t2 = c + d$$

$$t3 = t1 * t2$$

$$t4 = t1 + c$$

$$t5 = t3 - t4$$

Triple

	OP	arg1	arg2
(0)	+	a	b
(1)	+	c	
(2)	*	(0)	c

Backpatching Example 1 (If & Else Case)

Backpatching

Leaving the tables as empty and filling them later is called backpatching.

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

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

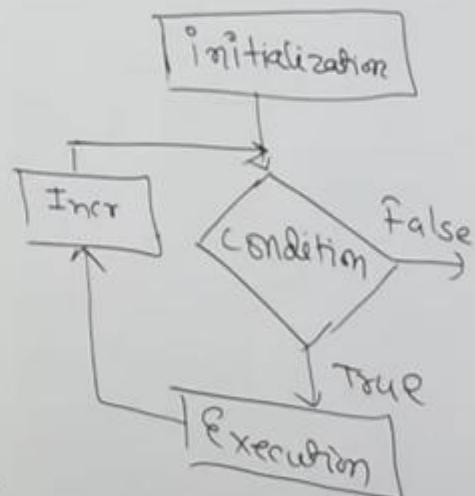
if ($a < b$) \wedge ($c < d$) then $t = 1$ else $t = 0$

- 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)

```
for (  $i = 1$ ;  $i \leq n$ ;  $i++$ )
    {
         $x = a + b * c;$ 
    }
```

- 1) $i = 1$
- 2) $\text{if } (i \leq n) \text{ goto } 4$
- 3) $\text{goto } 9$
- 4) $t_1 = b * c$
- 5) $t_2 = a + t_1$
- 6) $x = t_2$
- 7) $i = i + 1$
- 8) $\text{goto } 2$
- 9)



1) $i = 1$

2) $\text{if } (i > n) \text{ goto } \underline{8}$

3) $t_1 = b * c$

4) $t_2 = a + t_1$

5) $x = t_2$

6) $i = i + 1$

7) $\text{goto } \underline{2}$

8)

- 1) $\text{if } (i == 1) \text{ goto } 7$
- 2) $\text{if } (i == 2) \text{ goto } 11$
- 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)

$d \leftarrow i$
Case 1:

$$x_1 = a_1 + b_1 * c_1;$$

break;

Case 2:

$$x_2 = a_2 + b_2 * c_2;$$

break;

default:

$$x_3 = a_3 + b_3 * c_3;$$

break;

}

Case 2

- 11) $t_1 = b_2 * c_2$
- 12) $t_2 = a_2 + t_1$
- 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];
}
(x < 10) or (x ≥ 10)
```

- 1) $x = 0$
- 2) $i = 0$
- 3) $\text{if } (i \geq 10) \text{ goto } 15$
- 4) $t_1 = \text{base add of } A$
- 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

 7) $t_4 = \text{base add of } B$
 $A[3] = BA + (i-1)_b * C$ 8) $t_5 = i * 2$
 $= 100 + (2-1)_b * 2$ 9) $t_6 = t_4[t_5]$
 $= 106$ 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) —

$$x = A[i][j]$$

$$A: 10 \times 15$$

$$\begin{matrix} c=1 \\ i=2 \\ f=4 \end{matrix}$$

$$t_1 = i * 15$$

$$A[4][4] =$$

$$\begin{bmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{bmatrix}$$

$$t_2 = t_1 + j$$

$$23 = 2 \times 4 + 3 = 11$$

$$32 = 3 \times 4 + 2 = 14$$

$$LOC(23) = BA + (2 \times 4 + 3) \times 2$$

RMO

$$\begin{array}{ccccccccccccc} 00 & 01 & 02 & 03 & 10 & 11 & 12 & 13 & 20 & 21 & 22 & | & 23 & 30 & 31 & | & 32 & 33 \\ \nearrow & & & & & & & & & & & & & & & & & & \\ BA & & & & & & & & & & & & & & & & & & \end{array}$$

$$BA + (i \times N_c + j) \times w$$

$$x = A[i][j] \quad A:$$

$$t_1 = i * 15$$

$$t_2 = t_1 + j^o$$

$$t_3 = t_2 * 2^w$$

$$t_4 = \text{base Add of } A \quad \underline{\text{fMO}}$$

$$\left(\begin{array}{l} t_5 = t_4[t_3] \\ x = t_5 \end{array} \right) \quad \text{oo}$$

$$\rightarrow x = t_4[t_3]$$

Backpatching GATE Solved Example (3 D Array)

For a C program accessing $X[i][j][k]$, the following intermediate code generated by a compiler. Assume that size of an integer is 32 bits and the size of character is 8 bits. (GATE-2014)

4 byte $\text{int } X[32][32]\{8\}$ RMO $t_0 = i \times 1024$ $X[32][4]$
 RMO $i \quad j \quad k$ $t_1 = j \times 32$ $0 \quad 00 \quad 01 \quad 02 \quad 03$
 $t_2 = K \times 4$ $1 \quad 10 \quad 11 \quad 12 \quad 13$
 $t_3 = t_1 + t_0$ $2 \quad 20 \quad 21 \quad 22 \quad 23$
 $t_4 = t_3 + t_2$ $(2) = 2 \times 4 + 1$
 $t_5 = X[t_4] = (i \times N_c + j) \omega$

$$X[i][j][k] = X[i \times 32 \times 8 + j \times 8 + k] \times 4$$

$$= X[i \times 32 \times 8 \times 4 + j \times 8 \times 4 + k \times 4]$$

$$= X[i \times 1024 + j \times 32 + k \times 4]$$

$$t_5 = X[t_4] \quad \underline{\underline{t_3}} \quad \underline{\underline{t_2}} \quad \underline{\underline{t_1}} \quad \underline{\underline{t_0}}$$

a) X is declared as $\text{int } X[32][32]\{8\}$
 b) " " " $\text{int } X[4][1024]\{32\}$
 c) " " " $\text{char } X[4][32]\{8\}$
 d) " " " $\text{char } X[32]\{16\}\{2\}$



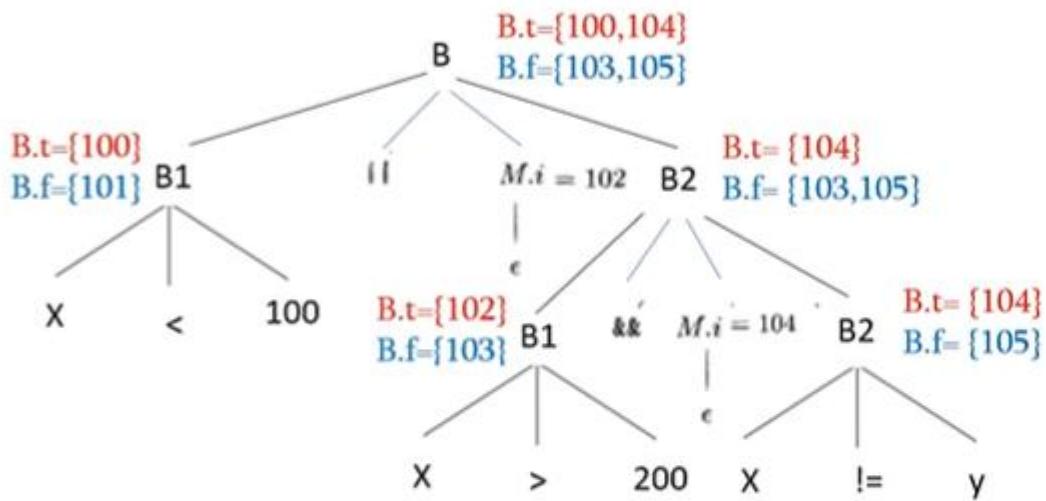
Backpatching for Booleans

Backpatching **Backpatching**

- **Backpatching** can be used to **generate code** for **Boolean expressions** and **flow of control statements** in one pass.
- The main **difficulty** with **code generation in one pass** is that **we may not know the target of a branch** when we generate flow of control statements.
- **Backpatching** is the technique to **get rid of this problem**.
 - **Generate** branch instructions with **empty targets**
 - When the **target is known** **fill the label of branch instructions** (Backpatching)
- We **utilize three functions** to modify the list of jumps:
 - **Makelist (i):** *Create a new list* including only i, and **returns a pointer** to the newly generated list.
 - **Merge(p1,p2):** *Concatenates the lists* pointed to by p1, and p2 and **returns a pointer** to the concatenated list.
 - **Backpatch (p, i):** *Inserts i as the target label* for each of the instructions on the **record pointed to by p**.

Productions	Semantic Actions
$B \rightarrow B1 \parallel M B2$	{backpatch(B1.falselist, M.instr); B. truelist = merge(B1. truelist, B2. truelist); B. falselist = B2. falselist; }
$B \rightarrow B1 \&& M B2$	{backpatch(B1 . truelist, M. instr); B. truelist = B2 . truelist;, B. falselist = merge(Bl. falselist, B2 . falselist); }
$B \rightarrow ! B1$	{ B. truelist = B1 . falselist; B. falselist = B1 . truelist; }
$B \rightarrow (B1)$	{ B. truelist = B1. truelist; B. falselist = B1 .falselist; }
$B \rightarrow E1 \text{ rel } E2$	{ B. truelist = makelist(nextinstr) ; B. falselist = makelist(nextinstr + 1); emit('if E1 .addr rel.op E2.addr 'goto----'); emit('goto -----'); }
$B \rightarrow \text{true}$	{ B . truelist = makelist(nextinstr) ; emit('goto -'); }
$B \rightarrow \text{false}$	{ B .falselist = makelist(nextinstr) ; emit('goto -'); }
$M \rightarrow \epsilon$	M. instr = nextinstr;

Example: $x < 100 \mid\mid x > 200 \&\& x \neq y$



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

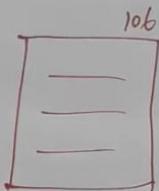
103: goto 107

104: if e & f goto 106

105: goto 107

106: (true)

107: (false)



) makelist(i)

2) merge (L₁, L₂)

3) backpatch (L, label)

4) nextquad

$E \cdot \text{truelist} = \text{merge}(E_1 \cdot \text{truelist},$
 $E_2 \cdot \text{truelist});$
 $E \cdot \text{falselist} = E_2 \cdot \text{falselist} \}$

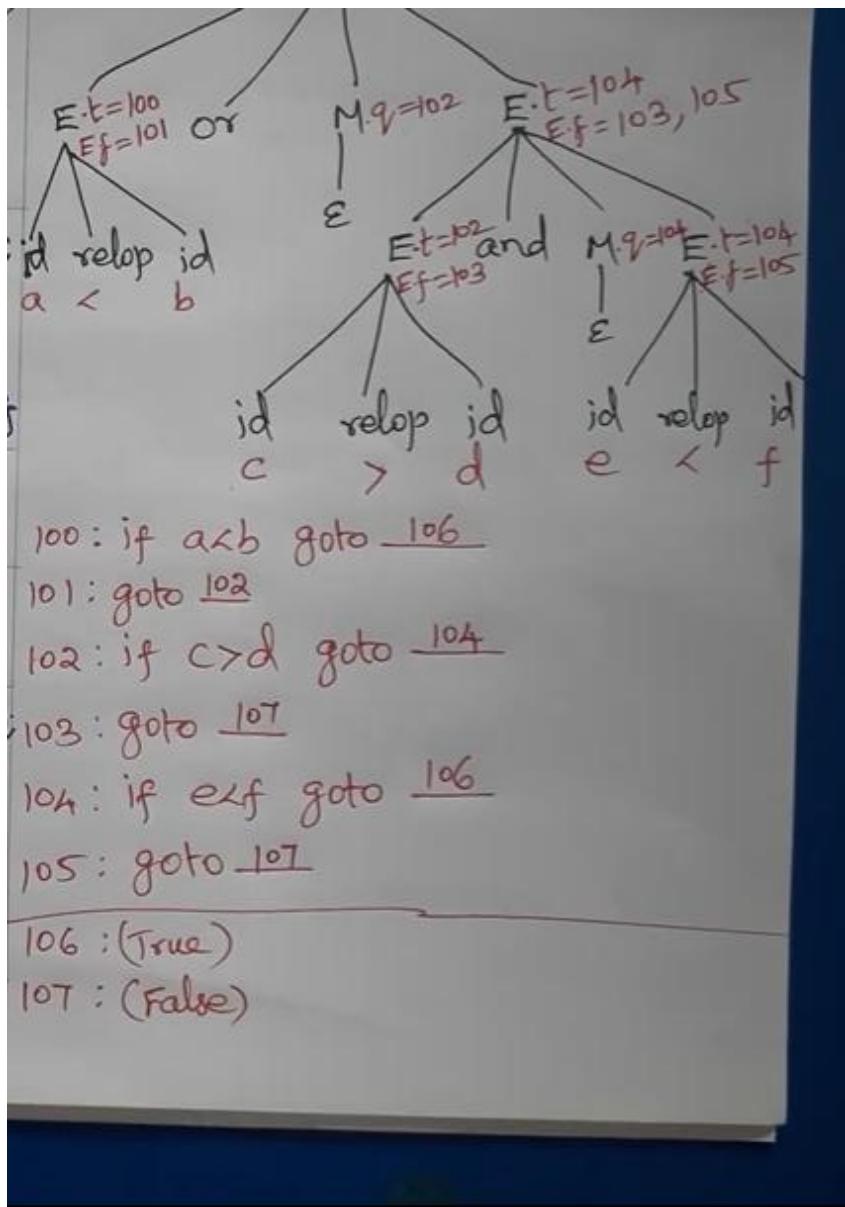
$E \rightarrow E_1 \text{ and } M E_2 \quad \{ \text{backpatch}(E_1 \cdot \text{truelist}, M \cdot \text{quad});$
 $E \cdot \text{truelist} = E_1 \cdot \text{truelist};$
 $E \cdot \text{falselist} = \text{merge}(E_1 \cdot \text{falselist},$
 $E_2 \cdot \text{falselist}) \}$

$E \rightarrow \text{not } E_1 \quad \{ E \cdot \text{truelist} = E_1 \cdot \text{falselist};$
 $E \cdot \text{falselist} = E_1 \cdot \text{truelist} \}$

$E \rightarrow (E_1) \quad \{ E \cdot \text{truelist} = E_1 \cdot \text{truelist};$
 $E \cdot \text{falselist} = E_1 \cdot \text{falselist} \}$

$E \rightarrow \text{id, relop id}_2 \quad \{ E \cdot \text{truelist} = \text{makelist}(\text{nextquad});$
 $E \cdot \text{falselist} = \text{makelist}(\text{nextquad}+1);$
 $\text{gen('if' id, relop id}_2 \text{ goto -'})$
 $\text{gen('goto -') } \}$

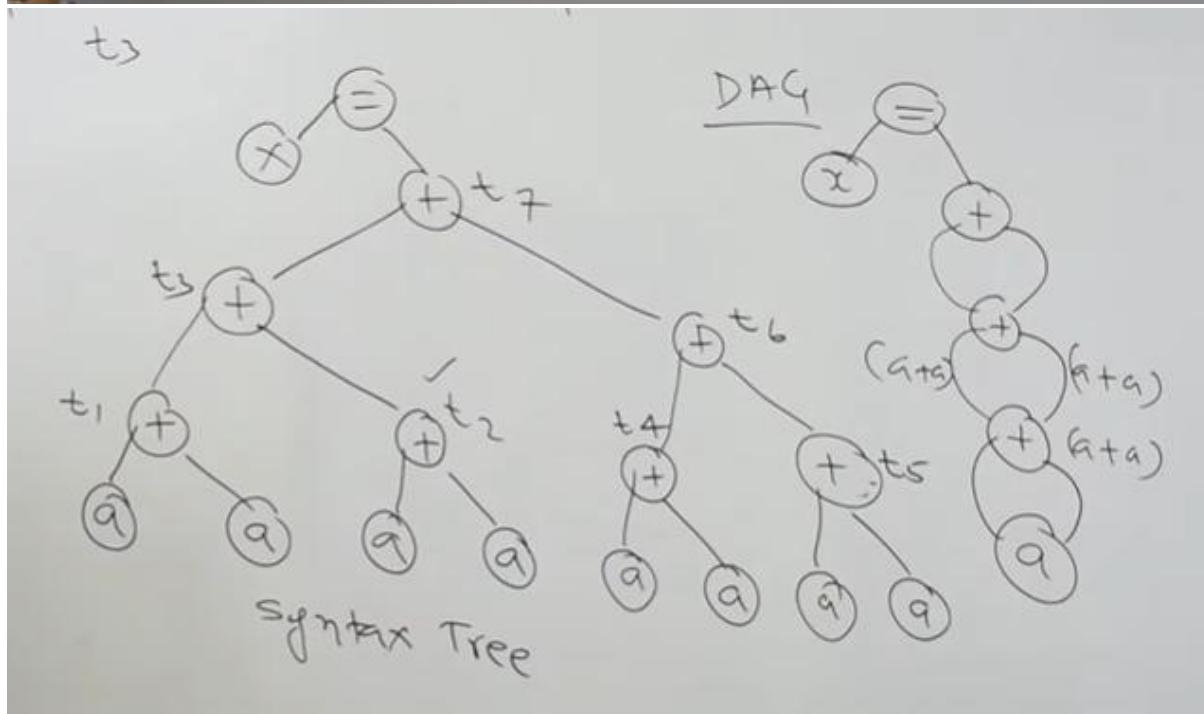
$M \rightarrow E \quad M \cdot \text{quad} = \text{nextquad}$

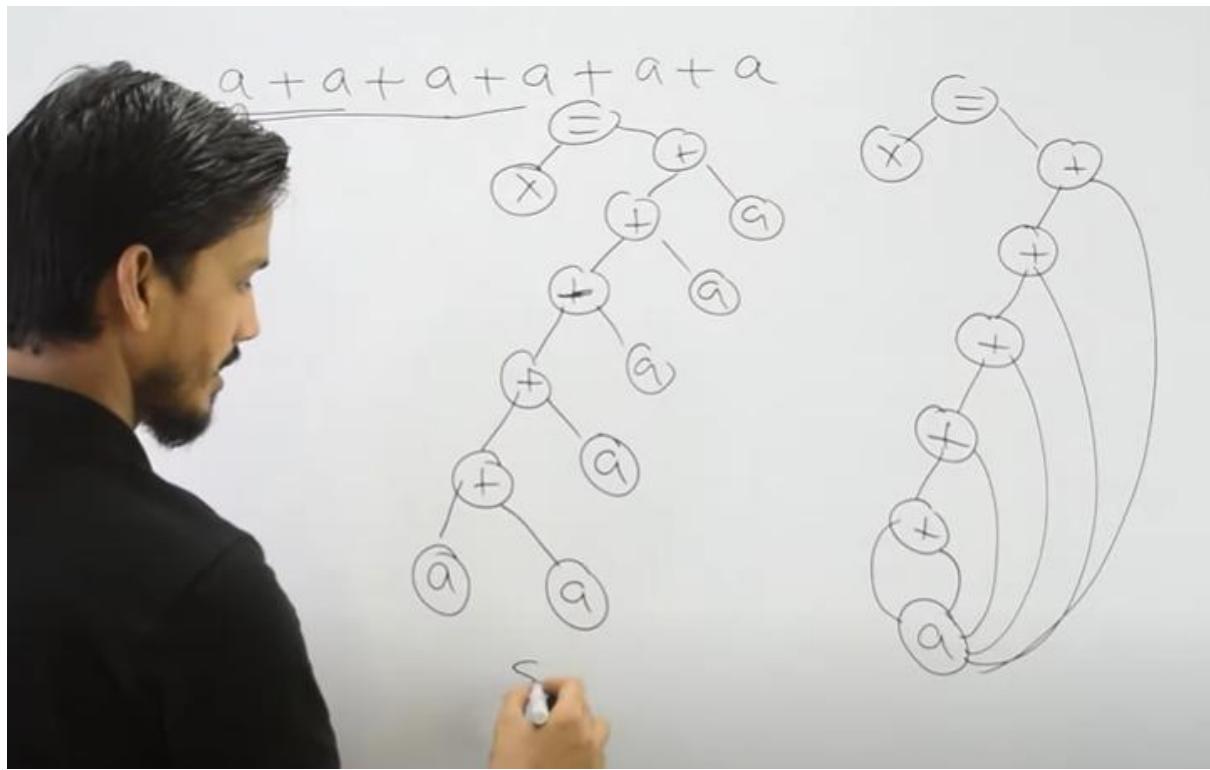


Directed Acyclic Graph | DAG Examples 1 | Intermediate Code Generation

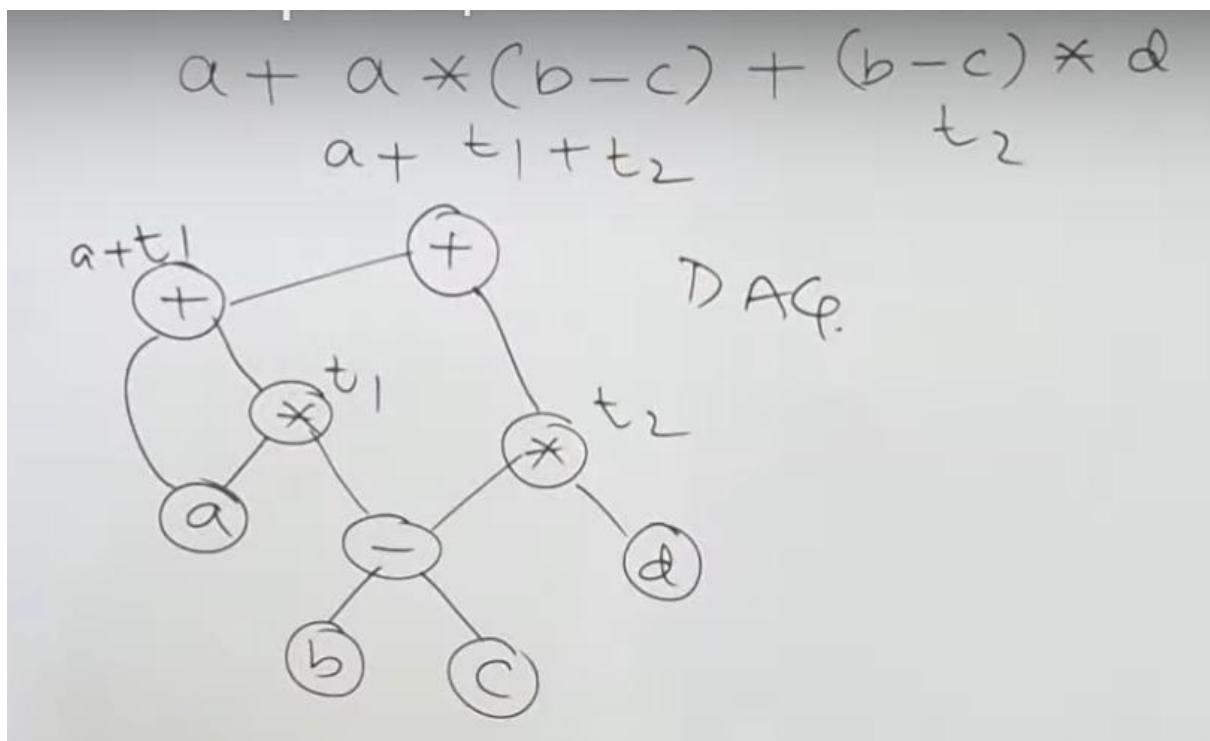
$$x = (((a+a) + (a+a)) + ((a+a) + (a+a)))$$

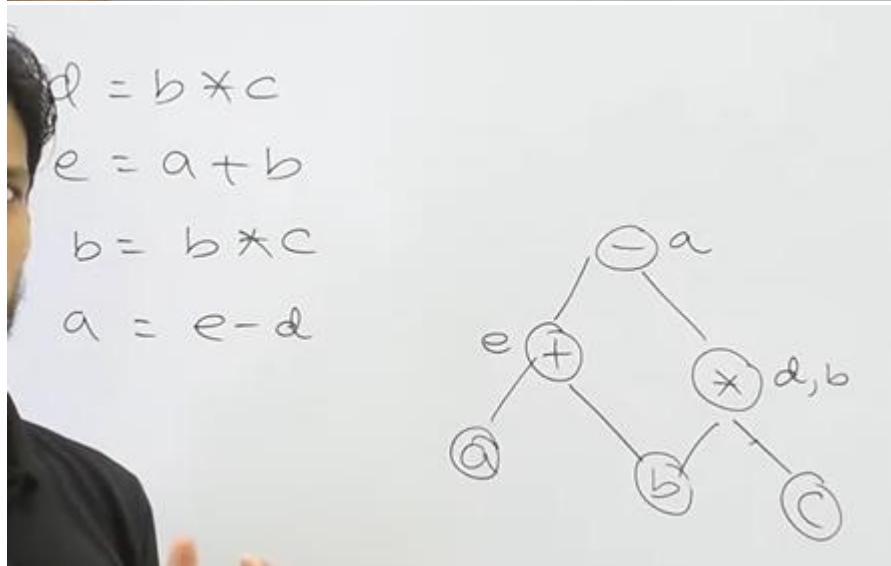
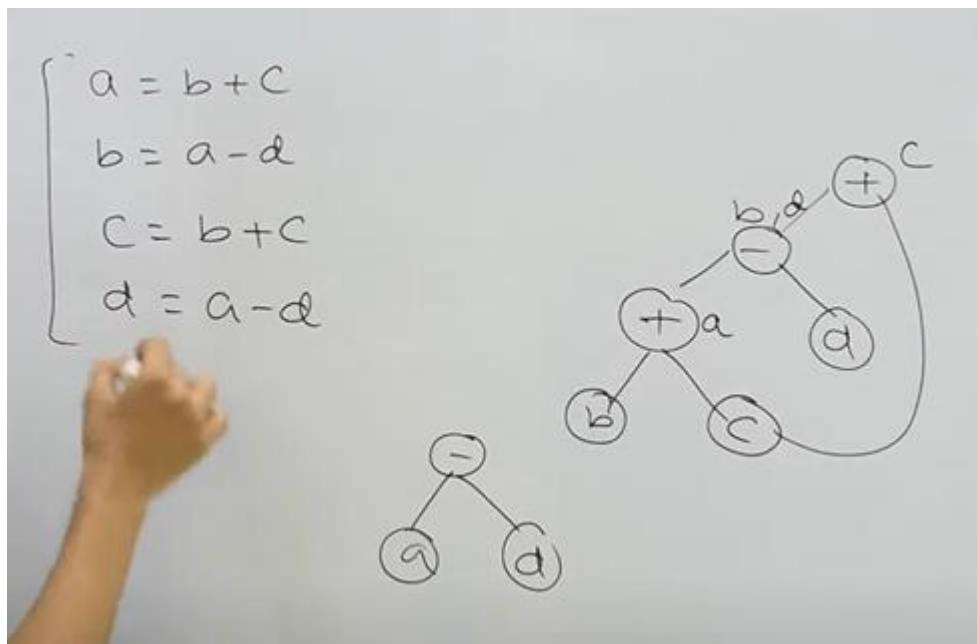
$t_1 = a+a$
 $t_2 = a+a$
 $t_3 = t_1 + t_2$
 $t_4 = a+a$
 $t_5 = a+a$
 $t_6 = t_4 + t_5$
 $t_7 = t_3 + t_6$
 $x = t_7$



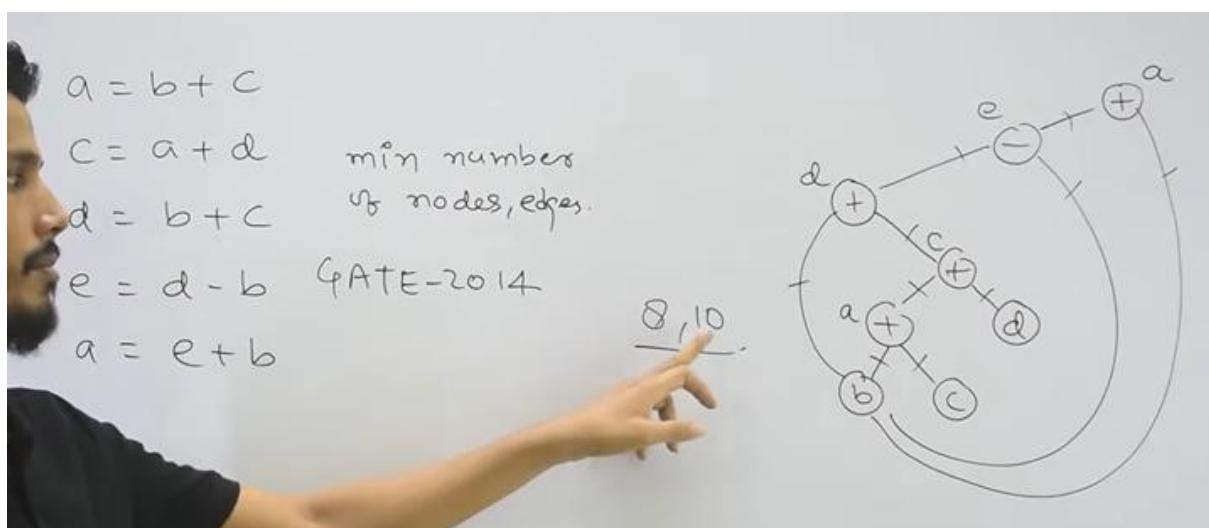


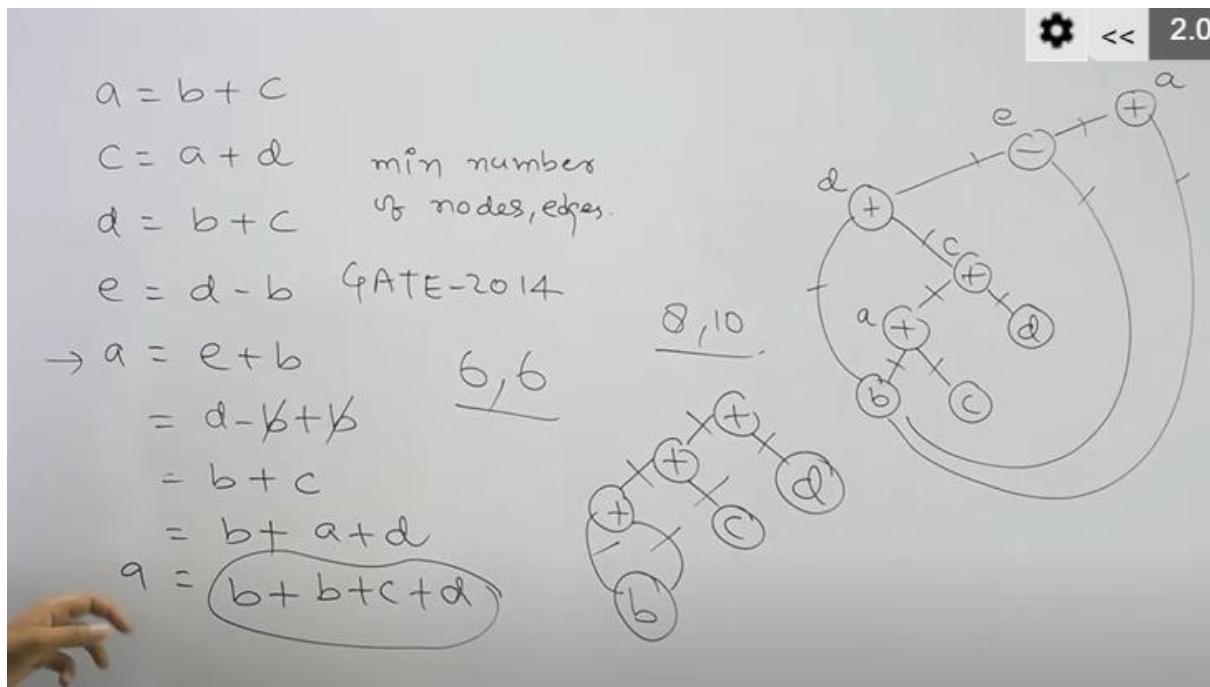
Directed Acyclic Graph | DAG Examples 2





Directed Acyclic Graph | DAG Examples 3





Code generation using DAG

Generation of Code from DAGs

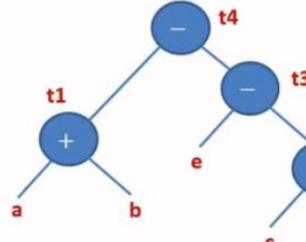
- Methods generating code from DAGs are:
 1. Rearranging Order
 2. Heuristic ordering
 3. Labeling algorithm

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



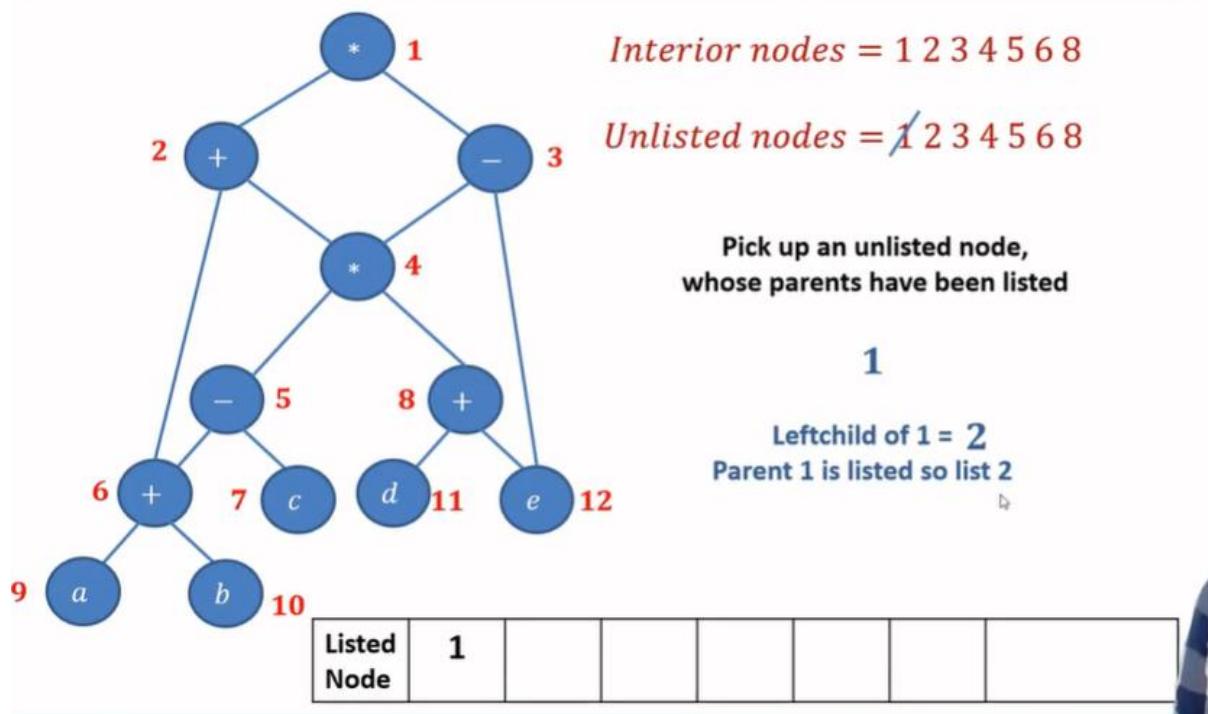
Example: Rearranging Order

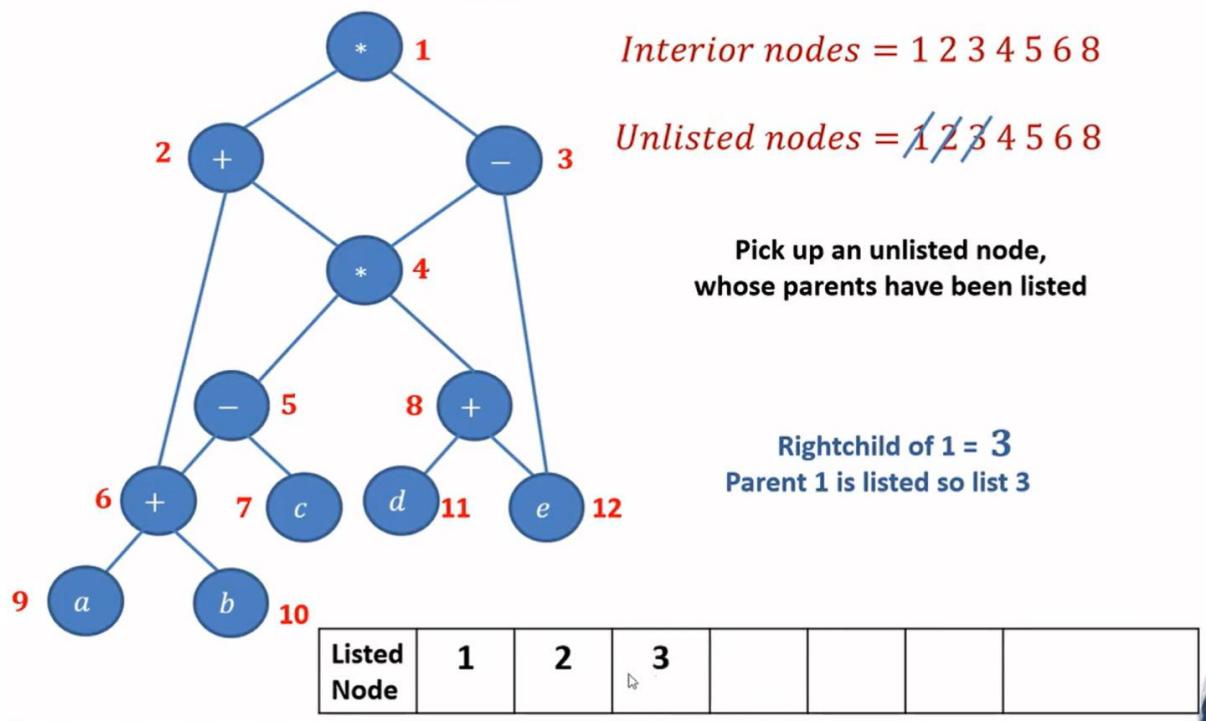
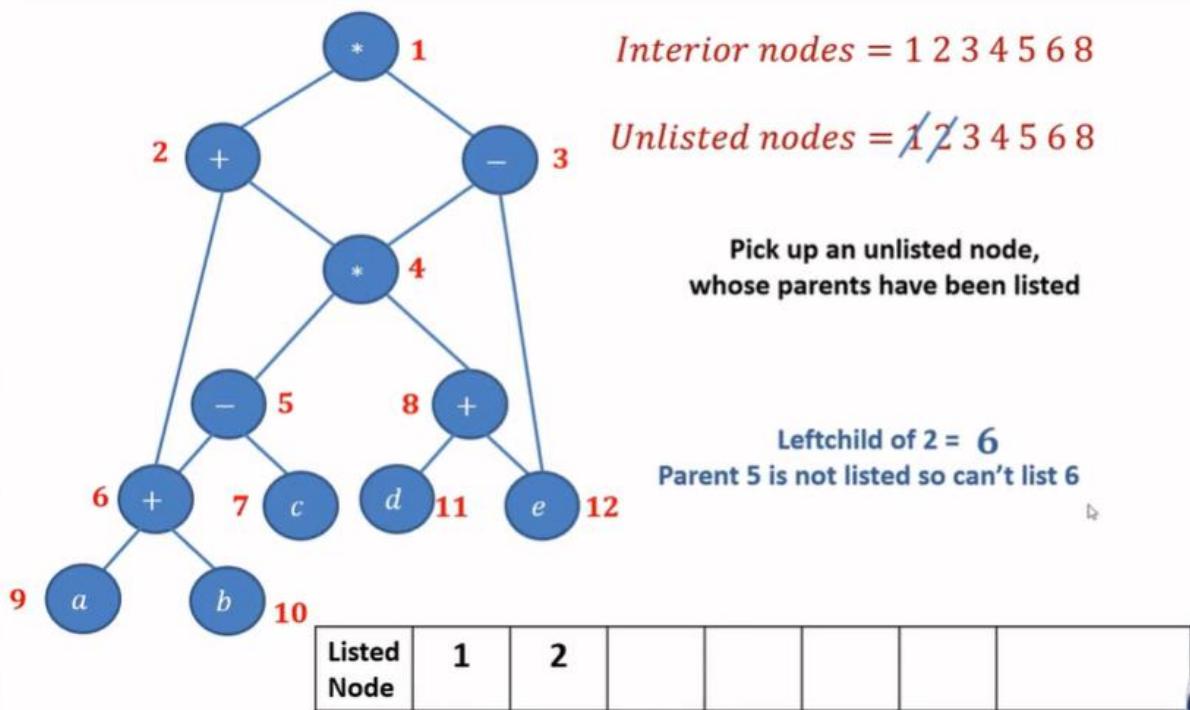
t1:=a+b	Re-arrange	t2:=c+d
t2:=c+d		t3:=e-t2
t3:=e-t2		t1:=a+b
t4:=t1-t3		t4:=t1-t3
Three Address Code		Three Address Code
MOV a, R0		MOV c, R0
ADD b, R0		ADD d, R0
MOV c, R1		MOV e, R1
ADD d, R1		SUB R0, R1
MOV R0, t1		MOV a, R0
MOV e, R0		ADD b, R0
SUB R1, R0		SUB R1, R0
MOV t1, R1		MOV R0, t4
SUB R0, R1		
MOV R1, t4		
Assembly Code		Assembly Code

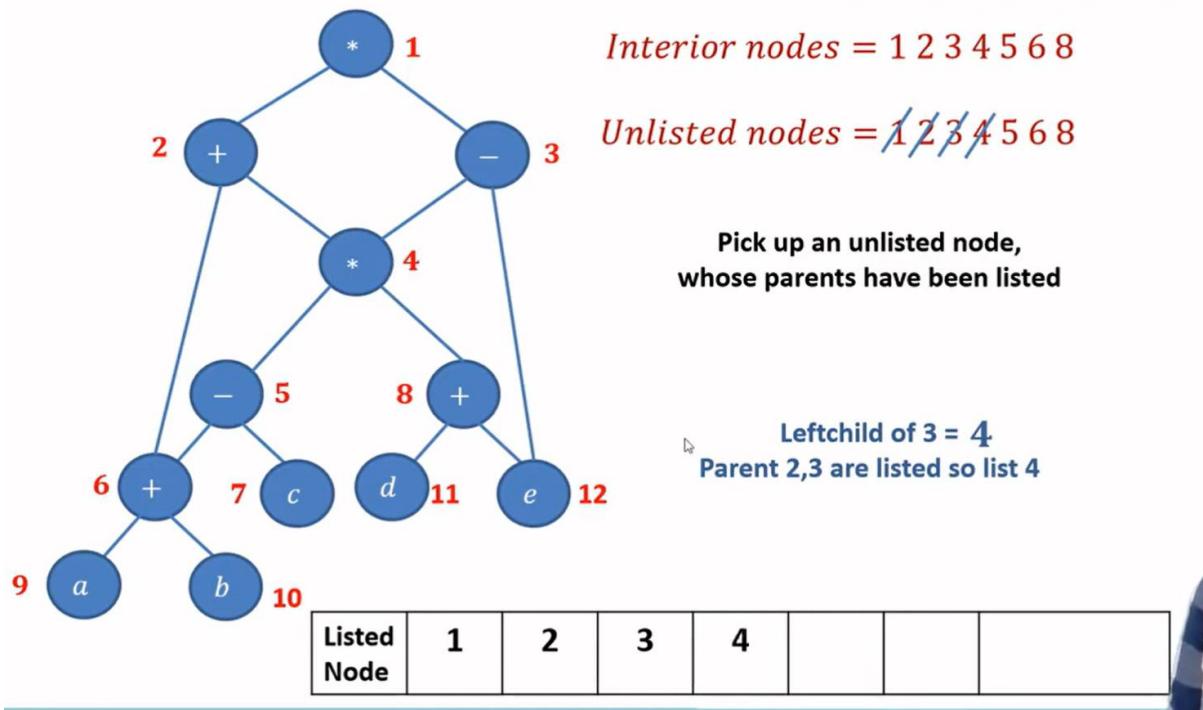
Algorithm: Heuristic Ordering

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

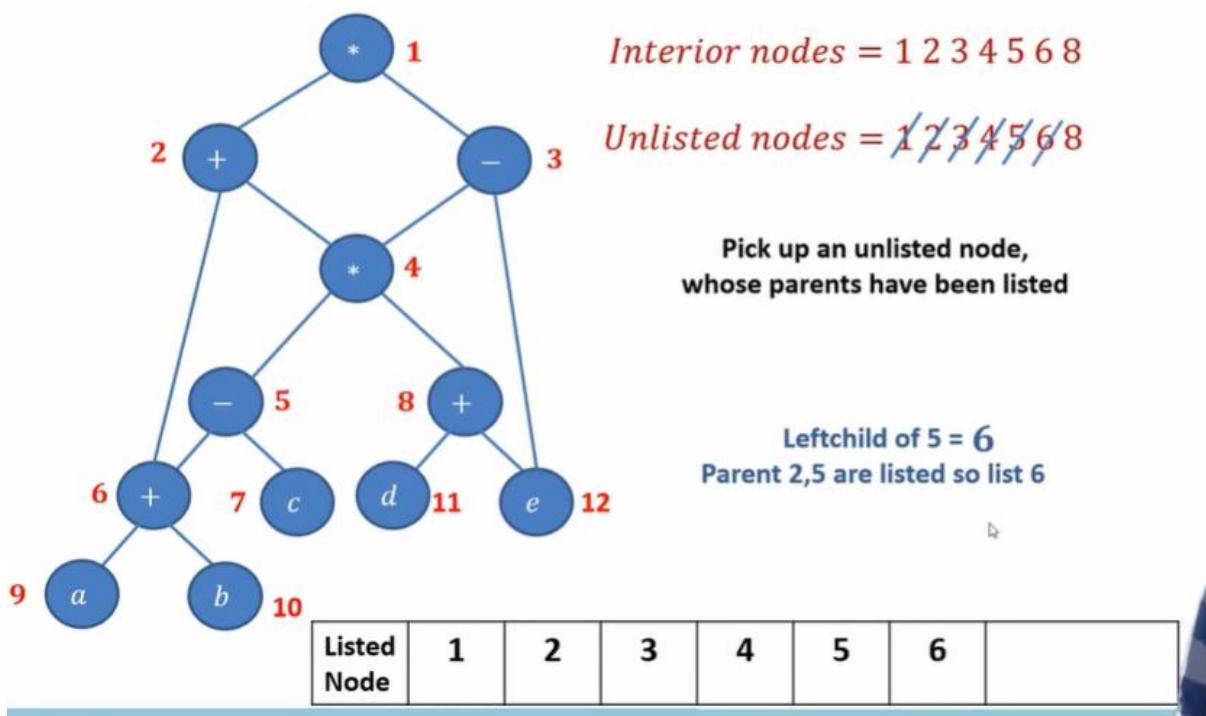
```
while(unlisted interior nodes remain)
{
    pick up an unlisted node n, whose parents have been listed
    list n;
    while(the leftmost child m of n has no unlisted parent AND is not leaf)
        {
            List m;
            n=m;
        }
}
```

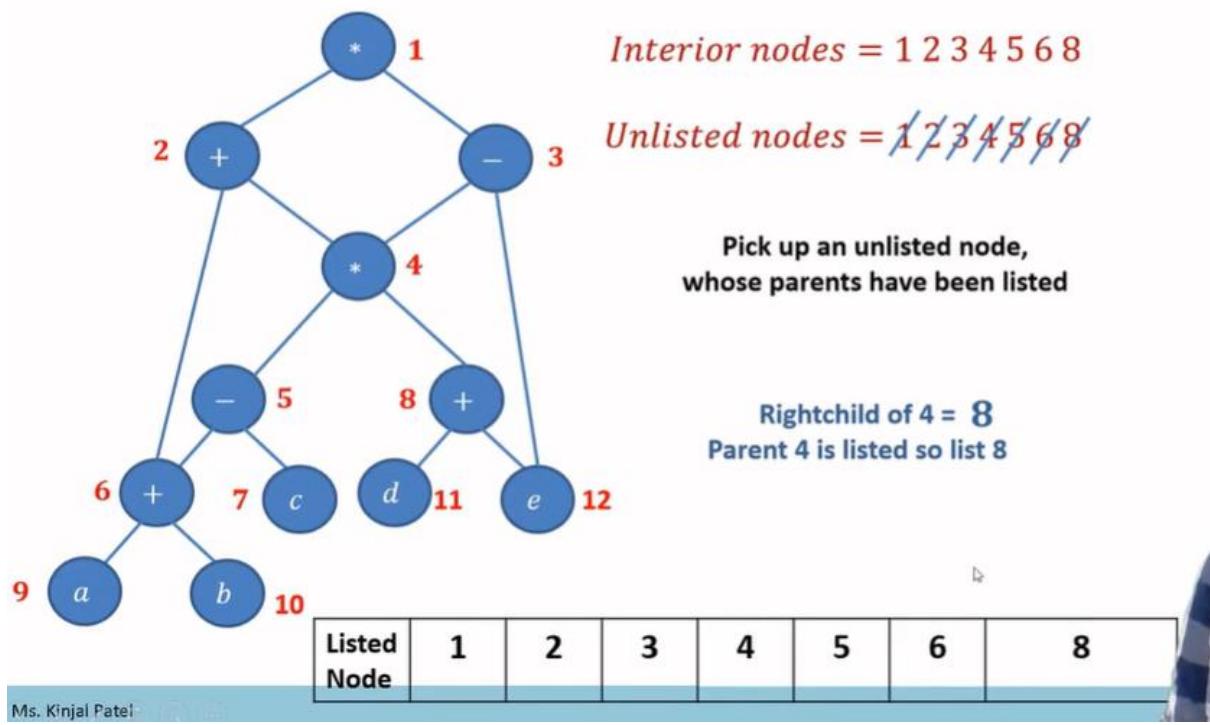






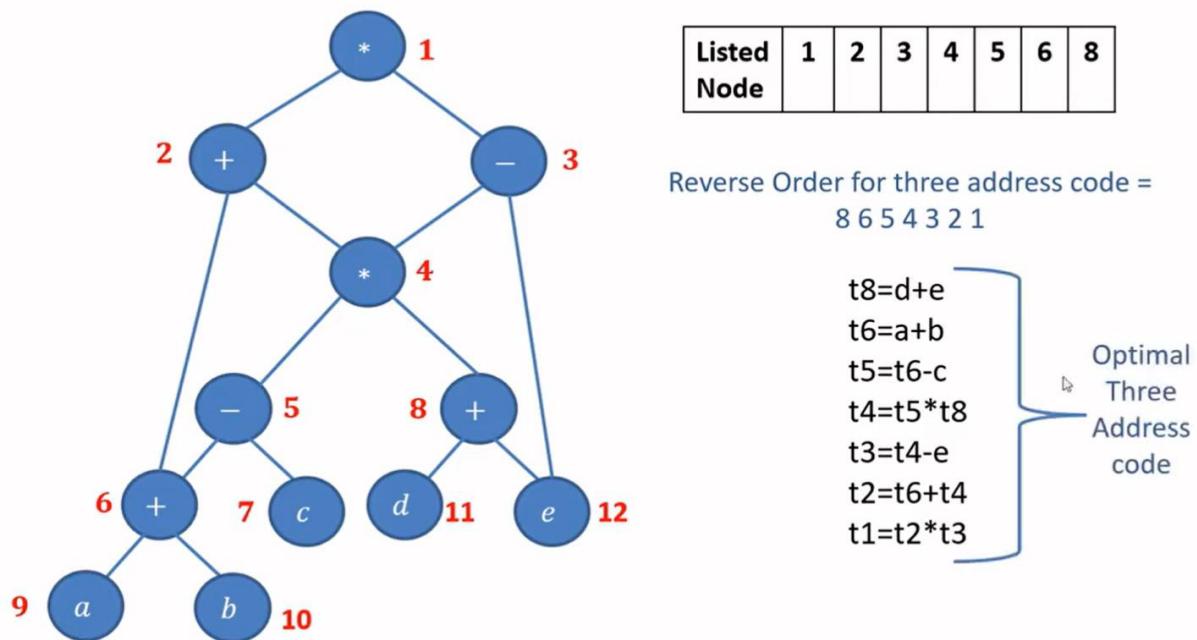
Leftchild of 4 = 5
Parent 4 is listed so list 5





Ms. Kinjal Patel

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,

$$\begin{aligned} \text{Label}(n) &= \\ \max(L1, L2) \text{ if } L1 \neq L2 \\ \text{Label}(n) &= L1 + 1 \text{ if } L1 = L2 \end{aligned}$$

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

↳

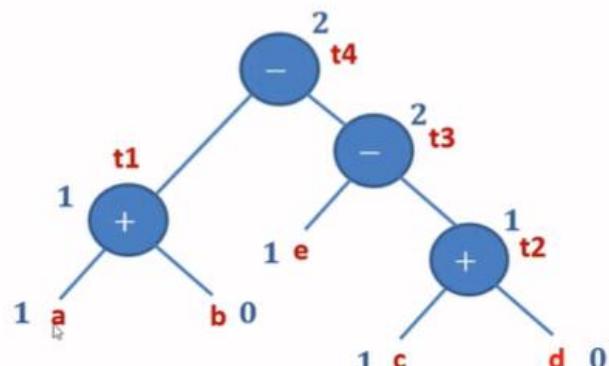
Example: Labeling Algorithm

Example:

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

Three Address Code

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



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

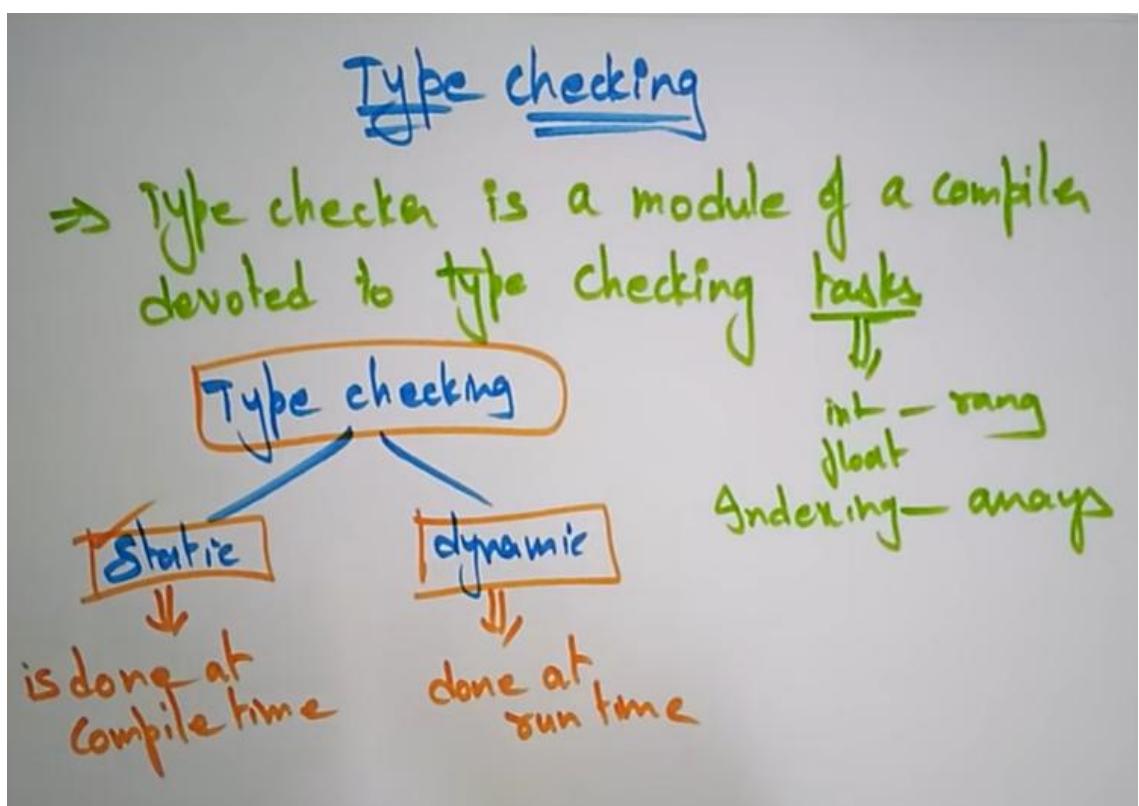
$l \circ \text{---} x$ if $l=r$ then $x = l+1$

Or the minimum of l and r

Same way continue till root $\max(l, r)$

(Example (E + R))

Type checking

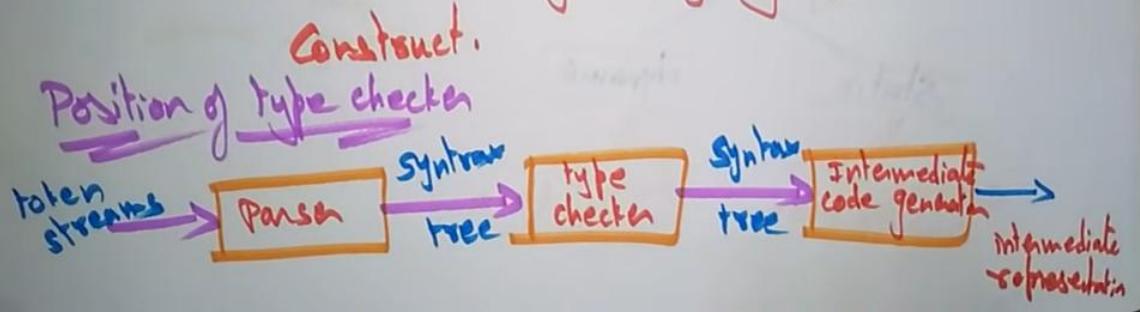


The design of type checker depends on

- Syntax, structure of lang constructs
- The type expression of lang
- The rules for assigning types to construct.

The design of type checker depends on

- syntactic structure of lang constructs
- The type expression of lang
- The rules for assigning types to



Type Expressions & System

Type Expression & Type systems

Will denotes the type
of lang construction.

T.E
↳ Basic type {int, real, boolean, char}
↳ Type name {Type constructors}
↳ Arrays, product, points, function ...

T → T.E

array (I, T) → T.E

Eg:- array [1...10] of int = array [1...10, int];

Basic type

Type constructor

Type systems

Collection of rules for arranging type expression.

Components of type system

Basic type

Eg:- int, char, float

Type constructor

Eg:- array, struct, string

Type Equivalence

name Equi
e.g.: char a, b;
a = 'A';
b = a;

Structural
equiv.
struct a, b;

Type Conversion | Implicit & Explicit

Type Conversion (or) type casting

A type cast is basically a conversion from one type to another.

=> There are two types of conversions

Type conversions

Implicit type conversion

Explicit type conversion

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.

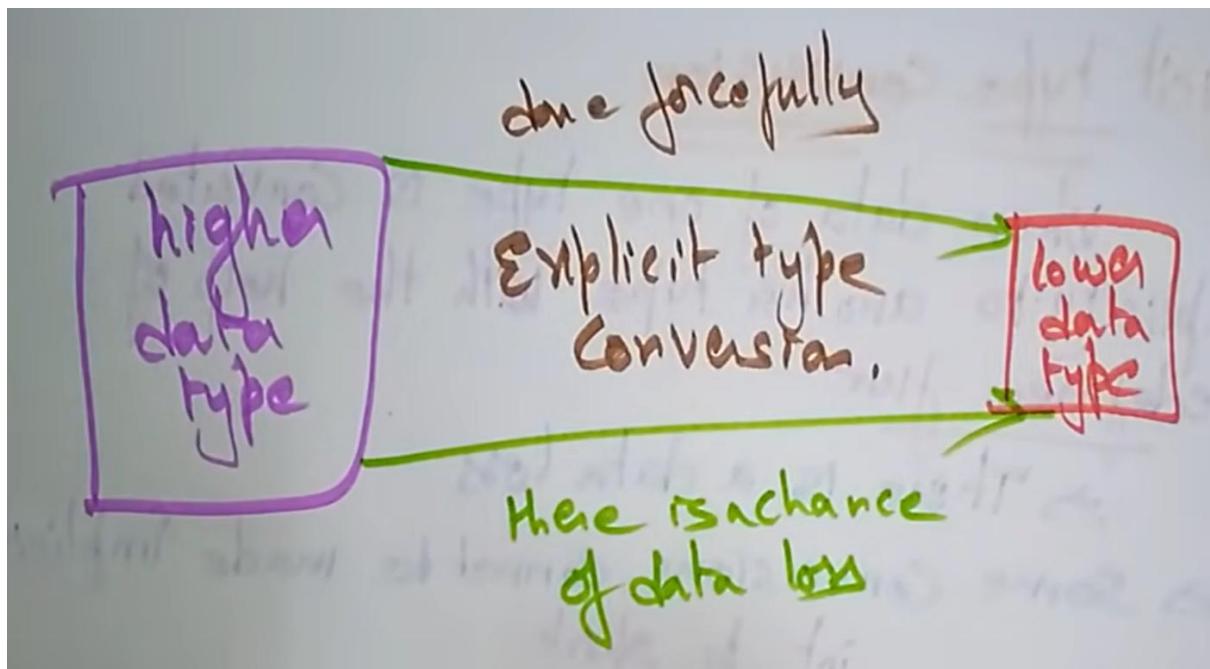
`bool → char → short int → int`
`float ← long ←`

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

Symbol Table

- ⇒ Symbol tables are data structures that are used by compilers to hold information about source-program constructs.
- ⇒ It is used to store information about the occurrence of various entities such as, objects, classes, variable names, functions etc,
- It is used by both analysis phase and synthesis phases.

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

Implementation of symbol Table

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

- ⇒ A symbol table can be implemented in one of the following techniques:
- ★ Linear (sorted or unsorted) list
 - ★ Hash table
 - ★ Binary search tree

⇒ Symbol tables are mostly implemented as hash table.

- ⇒ The operations provided by symbol table are
- Insert()
 - lookup()

- ⇒ Insert() :- It is more frequently used in analysis phase → front end
When tokens are identified and names are stored in table.
- ⇒ The insert() function takes the symbol and its value L.A
in the form of argument.

Eg:- int a; should be processed by compiler as: symbol table
int a; insert(a, int) Tokens

- ⇒ lookup() :- It is used to search a name & it determine:
- ★ The existence of symbol in the table
 - ★ Declaration of the symbol before it is used.
 - ★ check whether the name is used in the scope.
 - ★ Initialization of the symbol.
 - ★ checking whether the name is declared multiple items.

The basic format of `lookup()` function is
`lookup(symbol)`

This format varies according to the programming language.

Symbol table organization:

Var x, y: integer;

Procedure P:

Var x, a: boolean;

Procedure q:

Var x, y, z: real;

begin

end

...

...

...

RAM,

TOP	→	z	Real
		y	Real
		a	Real

symbol table for q

q	Real
x	Real-boolean
a	Real-boolean

symbol table for p

symbol table	in main
q	Real
x	Real-boolean
a	Real-boolean

symbol table
in main

P	Proc
y	int
x	int

P Proc

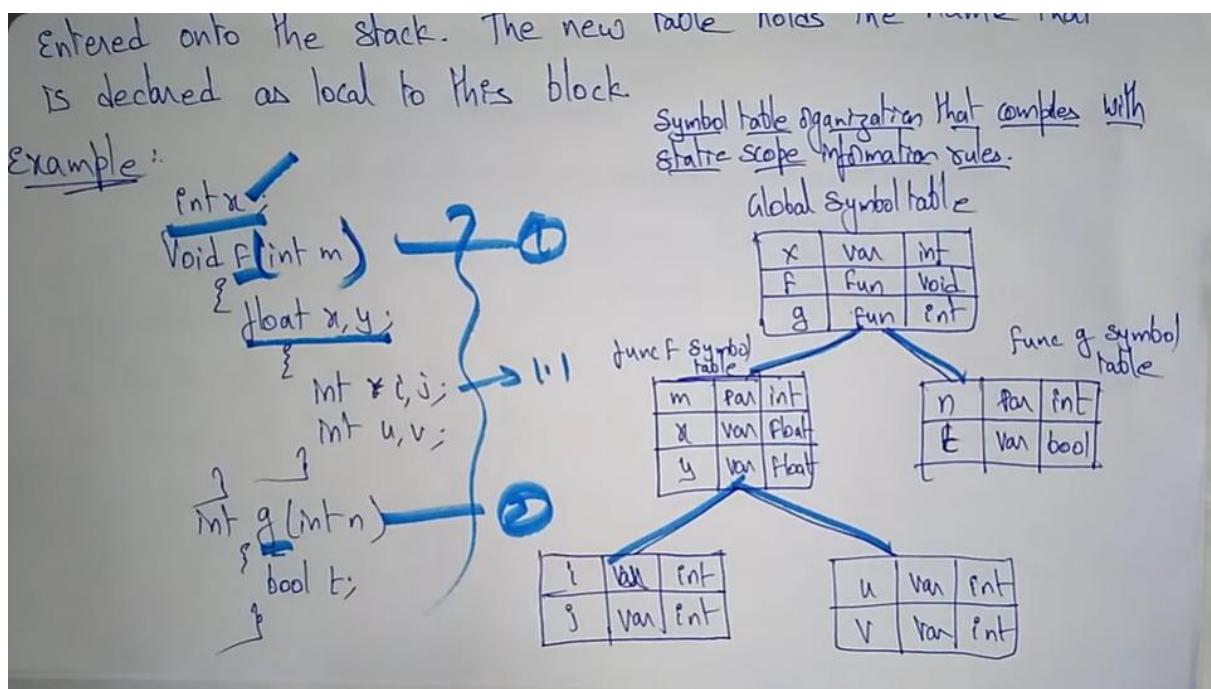
Representing scope Information

Representing scope information

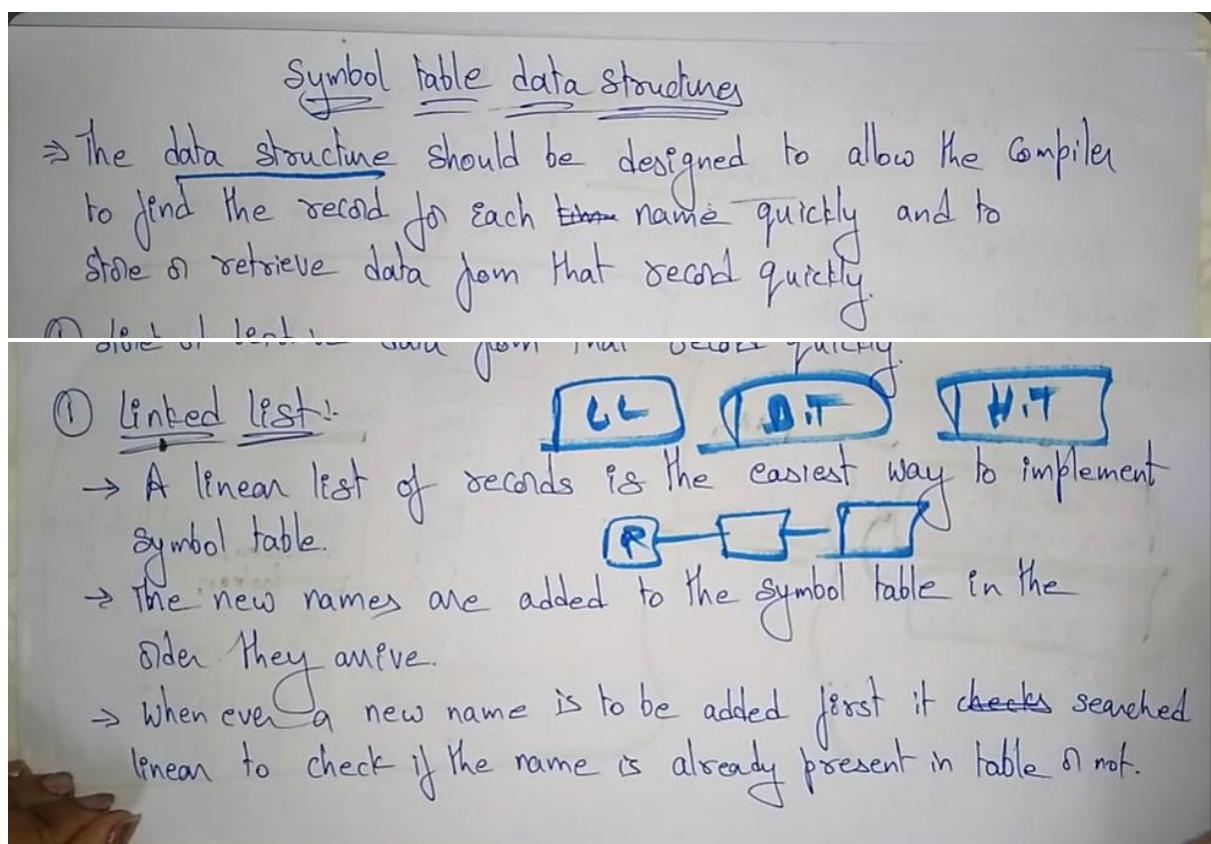
In source program, every name possesses a region of validity, called the scope of that name.

⇒ The rules in a block-structured language are as follows:

- ① If a name declared within block B, then it will be valid only within B.
- ② If B₁ is nested within B₂ then the names that is valid for B₂ is also valid for B₁ unless the name's identifier is redeclared in B.
- ③ The scope rules need a more complicated organization of symbol table than a list of associations between names and attributes.



Symbol table Structure



Time Complexity - $O(n)$

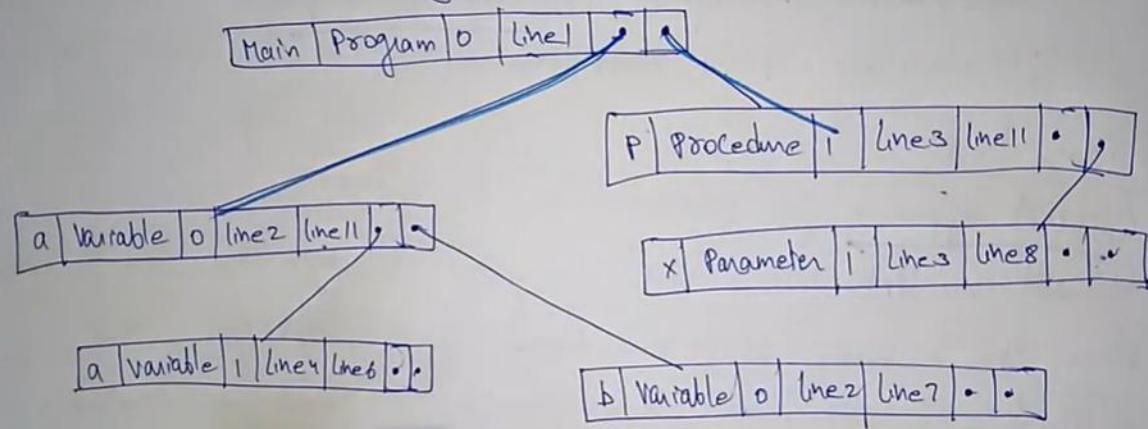
Advantage - less space, additions are simple

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.
- Whenever a name is to be added, first search in the tree,
if it does not exist then a record for new name is created
& added at proper position.
- This has alphabetical accessibility

Binary tree.



③ Hash table

⇒ In hashing scheme two tables are maintained.

- a hash table and a symbol table

⇒ A hash table is an array with index range 0 to table size - 1.

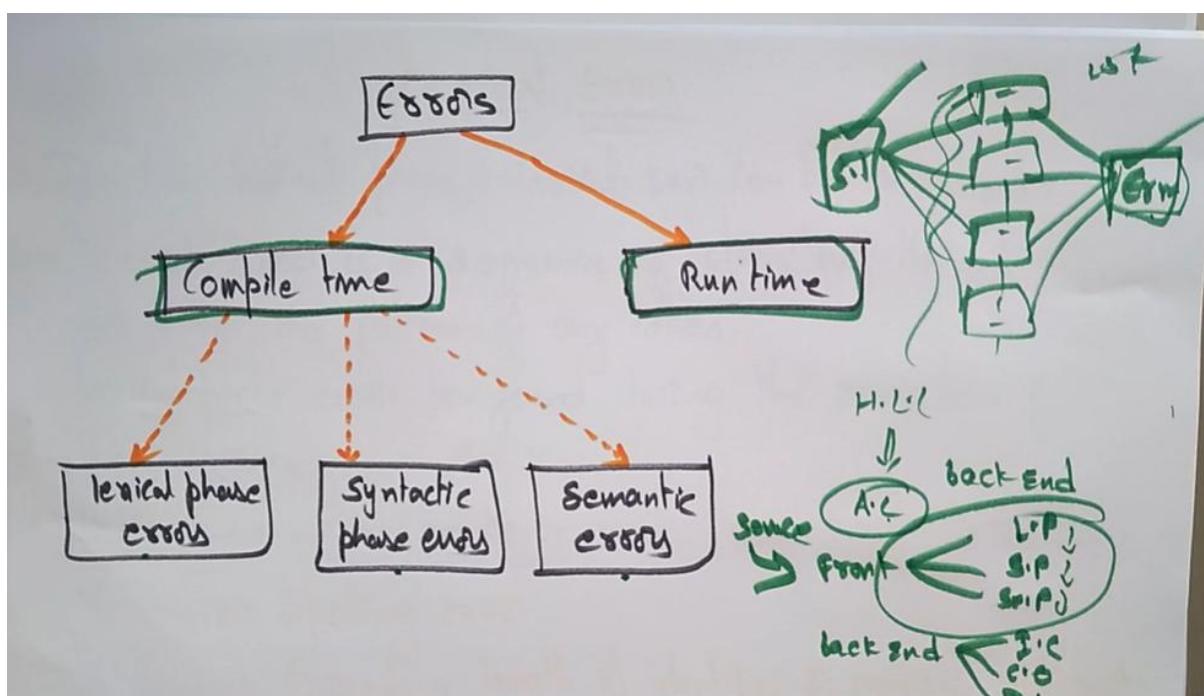
These entries are pointers pointing to names of symbol table.

⇒ To search for a new name we use hash function that will result in any integer between 0 to table size - 1.

⇒ Insertion & lookup can be made very fast - $O(1)$.



Errors | Lexical, Syntax & Semantic



Lexical phase error can be :-

→ Spelling error ✓

→ Exceeding length of identifiers or numeric constants.

- 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=y*xabs;
```

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

3

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 operators
- Unbalanced parenthesis

Eg:- Using "=" When "==" is needed

if (number=200)

else Count << "no: is equal to 200;"

Count << "no: is not equal to 200;"

In this example, Syntax Warning will come,

In this code, if expression used the equal sign which is actually an assignment operator not the relational operator which test for equality.

Eg 2 : float x = 1.2 // semicolon is missing

Eg 3 : 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.

Ex1- Use of non-initialized Variable :

```
int i;  
void f(int m)  
{  
    m = f; // i is undeclared, it shows  
    } semantic error.
```

Ex2- Type incompatibility :-

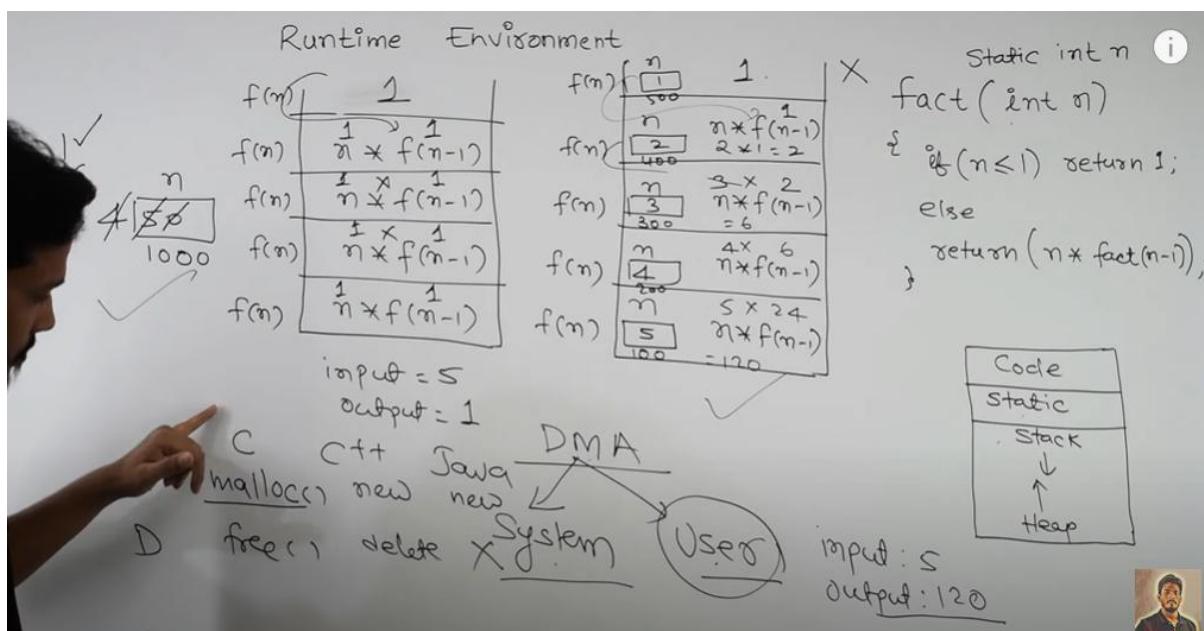
`int a = "hello";` // The types string & int are
not compatible

Ex3- Errors in Expressions:-

`String s = " - - -";` → Semantic error.

`int a = 8 - 8;` // The '-' does not support arguments

Run Time Environment



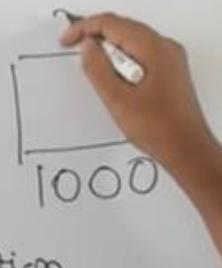
Runtime Environment

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.

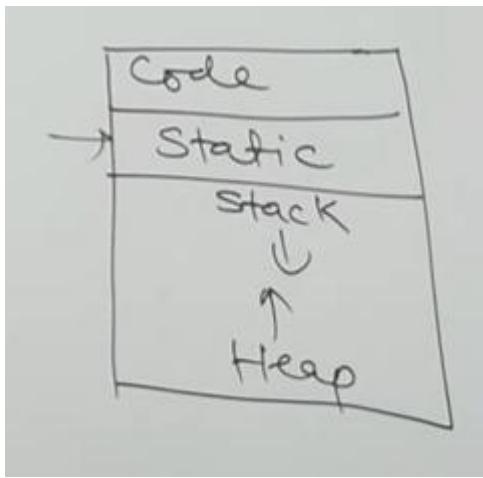
Stack Allocation

- 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 requirement.
- ② Recursion supported
- ③ Dynamic data structure supported.

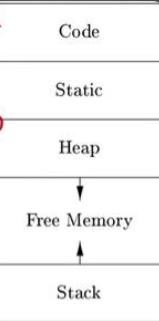


Run-Time Environments

- 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
 - The layout and allocation of storage locations for the objects named in the source program,
 - The mechanisms used by the target program to access variables,
 - The linkages between procedures,
 - The mechanisms for passing parameters,
 - 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.



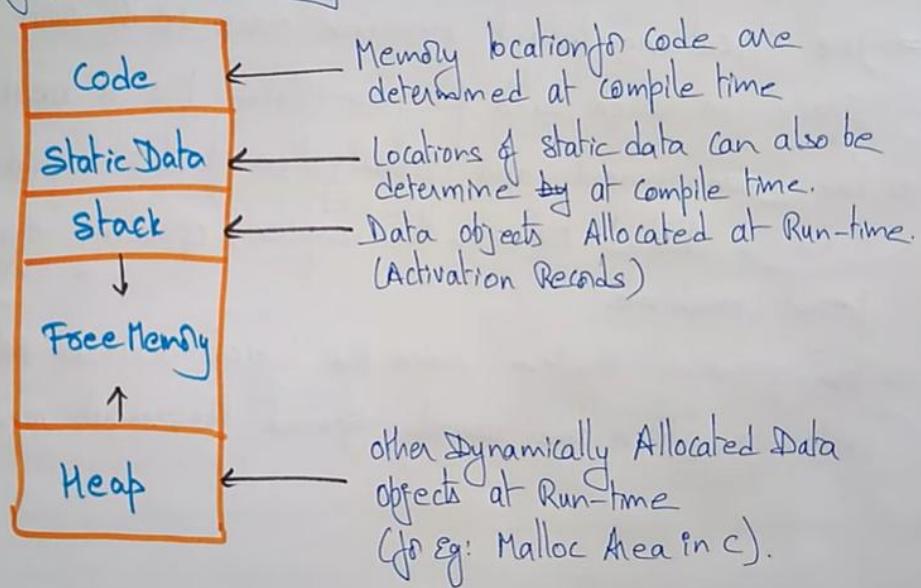
Storage organisation | Runtime memory

[Storage Allocation Strategies in Compiler Design - GeeksforGeeks](#)

Storage organisation

- ⇒ 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 address into physical addresses, which are usually spread throughout memory.

Sub division of Runtime Memory

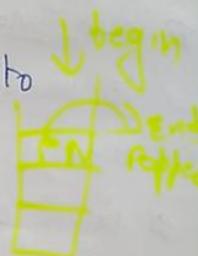


- ⇒ Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the **four bytes** a machine word can form.
- ⇒ object of multi byte is stored in consecutive bytes and gives the first byte address
- ⇒ Runtime storage can be subdivide to hold the different 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.

* 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.



Return Value
Actual Parameters
Control link
Access link
Saved Machine Status
Local Data
Temporaries

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 fⁿ 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 callers 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 → B(x, y)

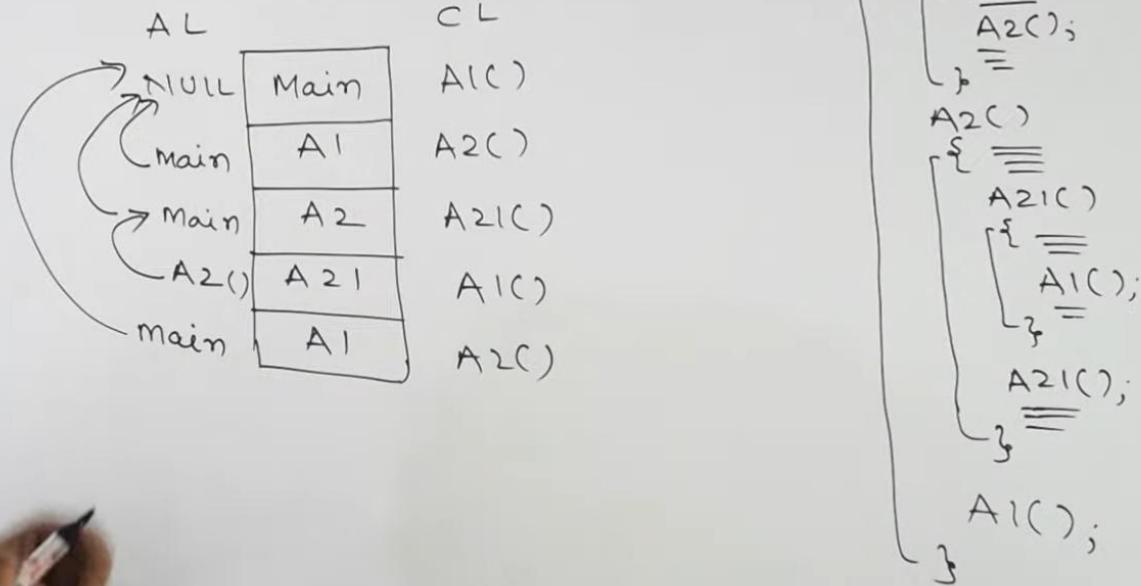
PC = ① —————

A()
cs

Code
S & G
Stack

Activation Record

Consider the following calling chain: Main → A1 → A2 → A21 → A1



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

Storage Allocation

The different ways to allocate Memory are 1-

1. Static storage Allocation

2. Stack storage Allocation

3. Heap storage Allocation

Static Storage Allocation :

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.

- static allocation supports the dynamic data structure that means, memory is created only at compile time and deallocated 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 :-

- The 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.

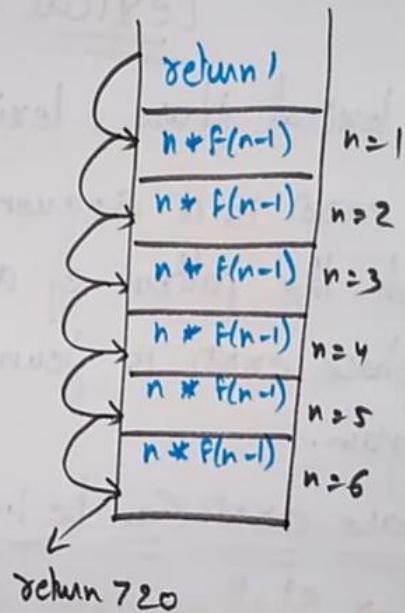
→ Heap storage Allocation :-

- 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 :-

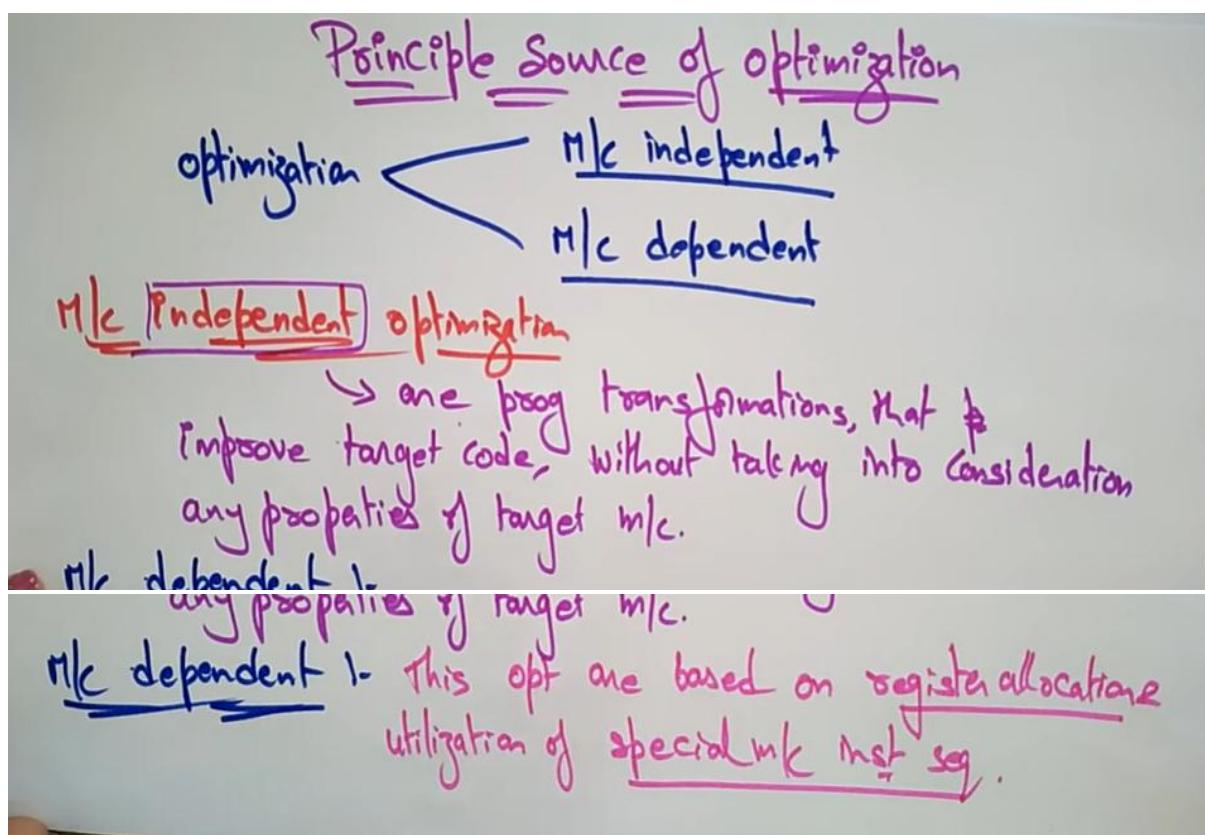
```
fact (int n)
{
    if (n <= 1)
        return 1;
    else
        return (n * fact (n-1));
}
```

fact (6)



return 720

Source of Optimisation | Principal



Code Optimisation technique

Machine Independent

- 1) Loop Optimization
 - a) Code Motion or Frequency reduction
 - b) Loop Unrolling
 - c) Loop Jamming
 - 2) folding
 - 3) Redundancy elimination
 - 4) Strength Reduction
 - 5) Algebraic Simplification
- ICG → TAC → DFA → IL → OPT

Machine Dependent

1. Register Allocation
2. Use of Addressing Modes
3. peephole Optimization
 - a) Redundant LOAD/STORE
 - b) Flow of Control Optimizations
 - c) Use of Machine idioms

for ($i \leq 1000$) $\approx 1K$

Code optimization techniques

Compile time evaluation

Variable propagation

dead code elimination

induction
variable &
strength
reduction

Code motion

① Compile time evaluation:

(a) $Z = \{5 \times (45.0 / 5.0) \times Y\}$
at compile time =

(b) $x = 5.7$
 $y = x / 3.6$
evaluate $\left\{ \frac{5.7}{3.6} \right\}$

② Variable propagation:

before optimization

$$c = a * b$$

$$x = a$$

$$\text{till } d = a * b + y$$

After optimization

$$c = a * b$$

$$x = a =$$

$$\text{till } d = \underline{\underline{a * b + y}}$$

③ Dead Code elimination:

Before elimination

$$c = a * b$$

$$\text{till } \boxed{x = b} \Rightarrow \text{dead state}$$

After elimination

$$c = a * b$$

$$\text{till } d = a * b + y$$

(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 % b == 0)  
        printf ("%d", a);  
}
```

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

(5) Induction variable & strength reduction :-



Eg: Before reduction

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

↓
by replace high strength operatn
by low strength operatn

After reduction

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

* Constant Folding :

Replacing an expression that can be computed at compile time by its value.

$$\text{ex} \quad C = 2 \times 3.14 \times \pi \Rightarrow C = 6.28 \times \pi$$

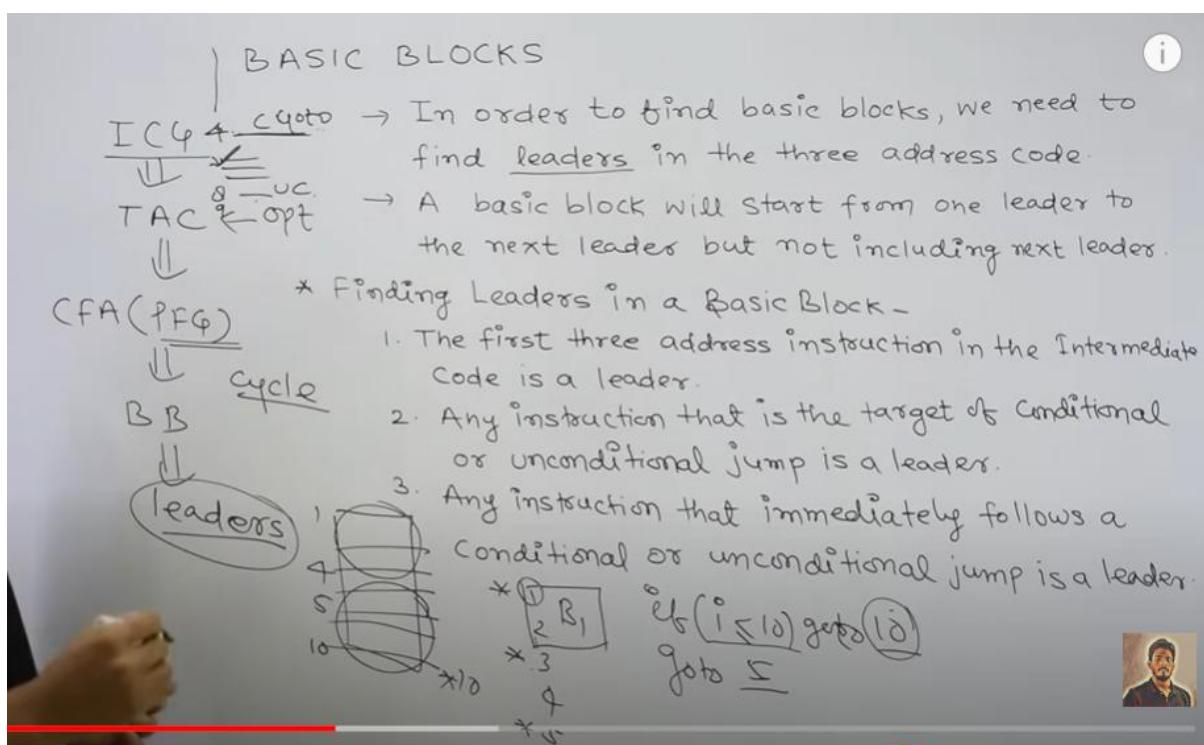
$$\text{ex} \quad 2 + 3 + B + C = 5 + B + C$$

* Redundancy Elimination :

$$\begin{array}{c} a = b + c \\ \hline e = d + b + c \end{array} \Rightarrow a = b + c \quad e = d + a$$



Basic Blocks and Flow Graphs



BASIC B

fact(n)

→

{

int $f = 1$

for ($i = 2; i \leq n; i++$)

→

{

$f = f * i;$ * Find i

return $f;$

1.

* 1) $f = 1$

2) $i = 2$

2

* 3) $\text{if } (i > n) \text{ goto } \underline{\underline{8}}$

3

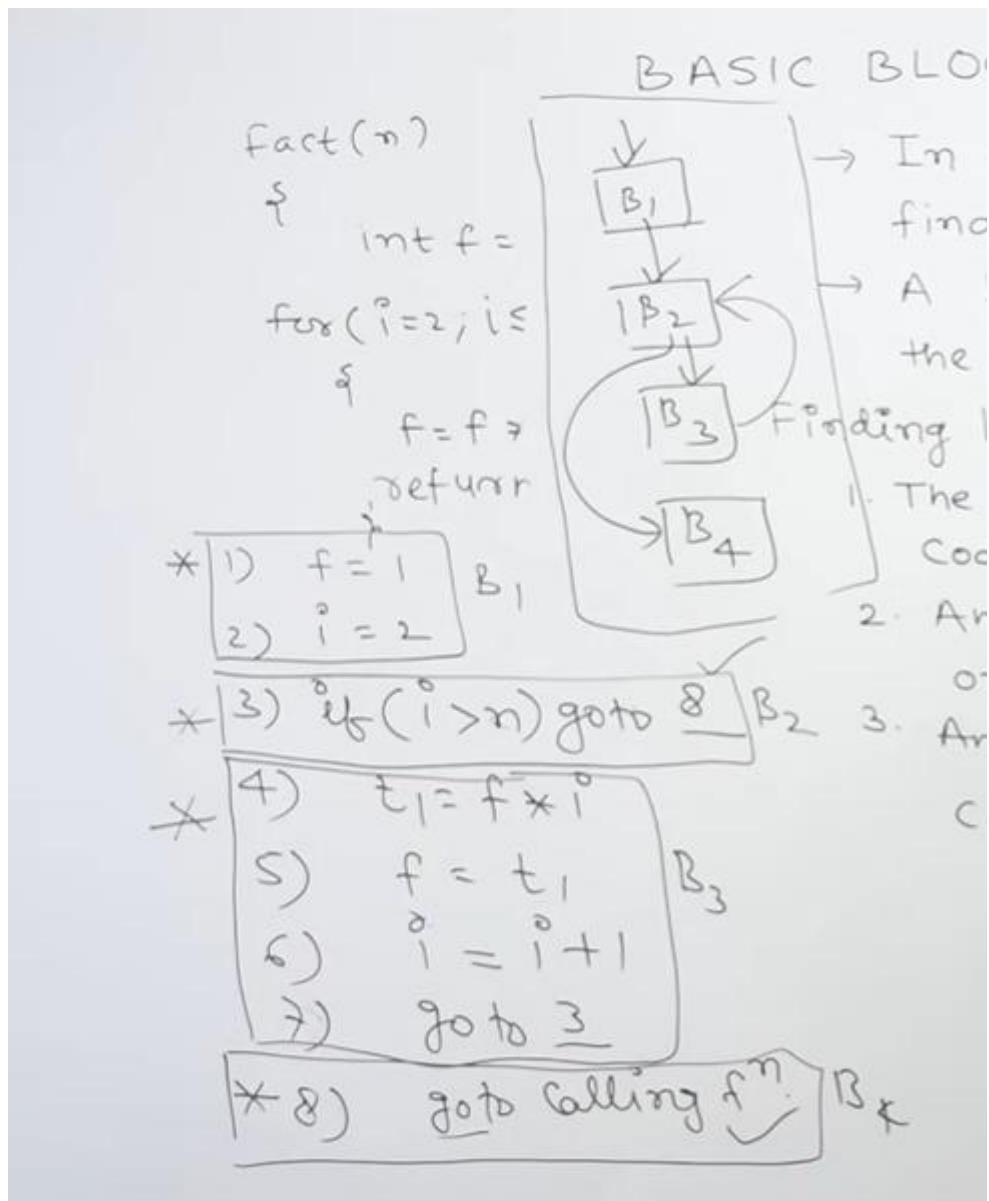
* 4) $t_1 = f * i$

5) $f = t_1$

6) $i = i + 1$

7) $\text{goto } \underline{\underline{3}}$

* 8) $\text{goto calling } \{ \}$

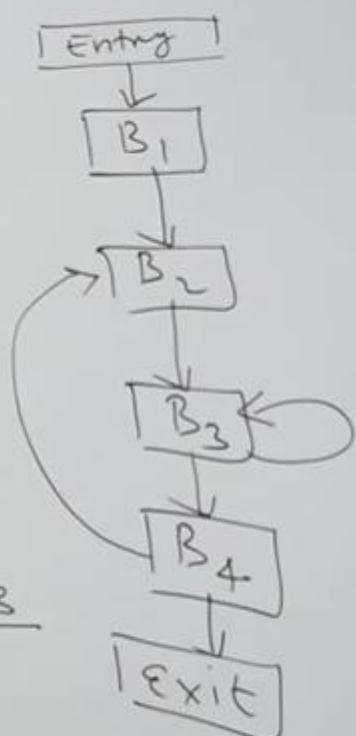


The number of nodes
and edges in the control flow
graph constructed for the code
respectively, are

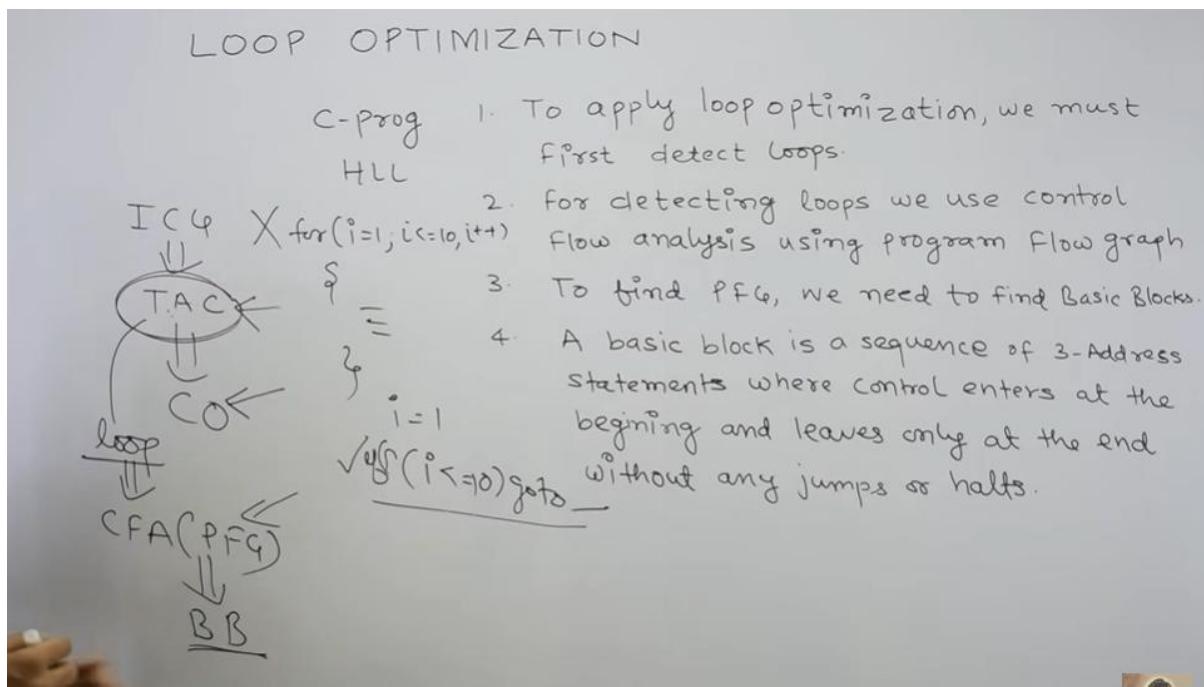
- 1) 5 and 7 ~~✓~~ 6 and 7
 3) 5 and 5 4) 7 and 8

Intermediate code given below.

* 1.	$i = 1$	$] B_1$
* 2.	$j = 1$	$] B_2$
*	$t_1 = 5 * i$	
B ₃	$t_2 = t_1 + j$	
*	$t_3 = 4 * t_2$	
-	$t_4 = t_3$	
*	$a[t_4] = -1$	
-	$j = j + 1$	
*	$\text{if } (j \leq 5) \text{ goto } 3$	
B ₄	$i = i + 1$	
*	$\text{if } (i < 5) \text{ goto } 2$	



Loop Optimisation | Code motion & Strength reduction



loop optimization

It is most valuable machine-independent optimization, because program's inner loop takes bulk to time of a programme,
→ If we decrease the number of instructions in an inner loop then the running time of a program may be improved
Even if we increase the amount of code outside that loop.

For loop optimization the following three techniques are important:

1. Code motion
2. Induction-Variable Elimination
3. Strength reduction.

Loop 011

a) Frequency Reduction (Code Motion)

b) Loop
Reducing the
are made

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

Ex $i = 0$ $i = 0$
 $t = a/b$

while($i < 1000$) \Rightarrow while($i < 1000$)

{ $A = \left(\frac{a}{b}\right) + i$ }

} $i++$ $A = t + i$
 $i++$

1. Code Motion -

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.

Ex: `while (i <= limit - 2)` // stmt does not change limit

After Code motion the result is as follows:

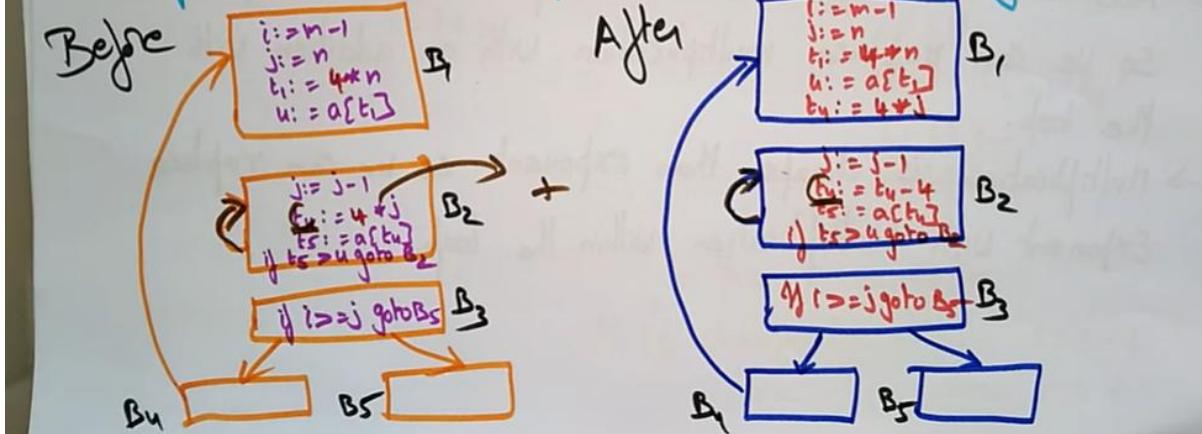
$$a = \lim_{x \rightarrow 2} f(x)$$

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

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

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 Unrolling

Reducing the no. of times comparisons/Combinations made in the loop.

c) Loop

Combining the bodies of two loops.

c) Loop Jamming

Combining the bodies of two loops.

Ex for (

for (i=0; i<10; i++) {

 printf("Hi") ; , , ,

 } for (i=0; i<10; i=(i+2)) { , , ,

 printf("Ho") ; , , ,

 printf('Hi') ; , , ,

 } for (i=0; i<5; i++) { , , ,

 printf("Hi") ; , , ,

 printf("Hi") ; , , ,

Ex for (

for (i=0; i<5; i++) {

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

Ex for (

for (i=0; i<5; i++) {

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

 } for (i=0; i<5; i++) { , , ,

<1000)

t+1

3. Reduction in strength :-

- Strength reduction is used to replace the expensive operation by the cheaper one on the target machine.
- Addition of a constant is cheaper than a multiplication.
So we can replace multiplication with an addition with the loop.
- Multiplication is cheaper than exponent. So we can replace Exponent with multiplication within the loop.

ex > mul > add

Example :-

before

```
while (i<10)
{
    j = 3 * i + 1;
    a[i] = a[i]-2;
    i = i+2;
}
```

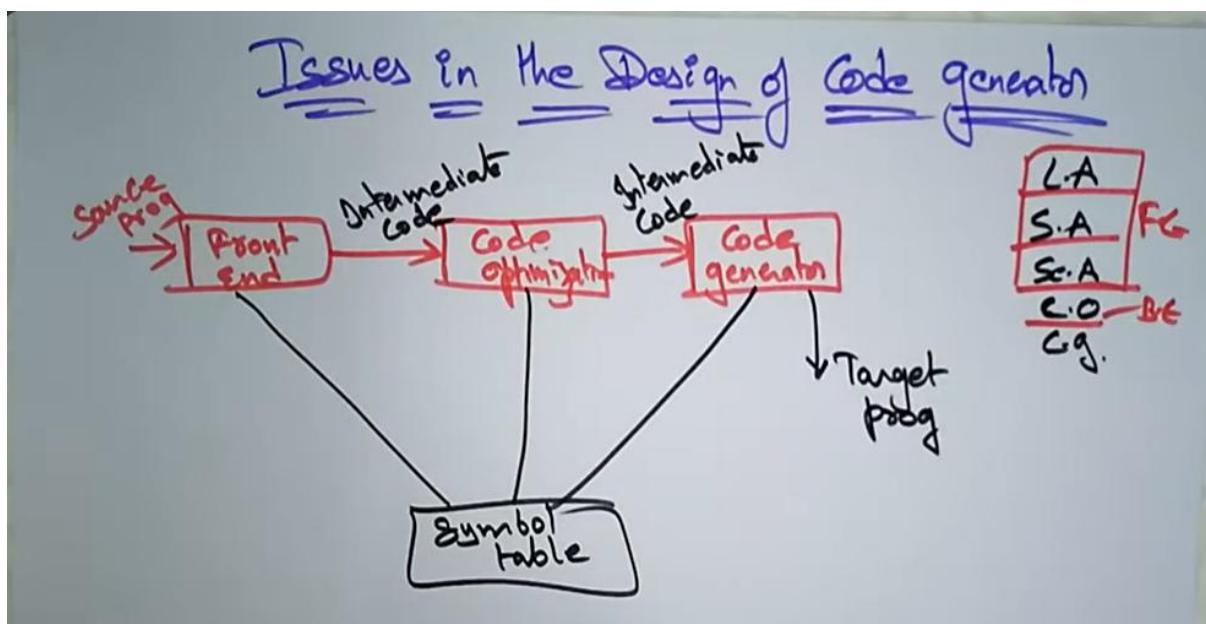
After strength reduction
the code will be :

```
s=3 * i + 1;
while (i<10)
{
    j=s;
    a[i] = a[i]-2;
    i = i+2;
    s=s+6;
}
```

In the above code,
It is cheaper to compute $s=s+6$
than $j=3*i+1$

Code generator | Issues in design

[Design Issues - javatpoint](#)



6 Design Issues

- ① Input to code generator
- ② Target program
- ③ Memory Management
- ④ Instruction Selection
- ⑤ Register allocation issues
- ⑥ Evaluation order

① Input to code generator → {Intermediate code}

Linear representation
like postfix & TAC (or) DAG
⇒ Up 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 linker
- Assembly lang.



④ Instruction Selection ↗

code generator takes I.C \rightarrow If & convert into target mt inst set.

\Rightarrow It is the responsible for code generator to choose appropr. inst.

\Rightarrow The quality of the generated code is determined by its speed & size.

Eg: $x = y + z$

LD R0, Y
ADD R0, R0, Z
ST X, R0

⑤ Register Allocation ↗

What value to hold in what reg?

Inst involving reg operands { fast }

Mem operands { longer & } slow }

\rightarrow Register Allocation
during which we select the set of var that will reside in reg at a pt in prog.

\rightarrow Register Assignment.
during which, we pick specific reg that a var will reside in

④ 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.

$$\text{Eg 1- } (a+b) - \underline{(c+d)} * c$$

$$\begin{aligned} \text{TAC} \Rightarrow & \quad t_1 = a + b \\ & \quad t_2 = c + d \\ & \quad t_3 = c * t_2 \\ & \quad t_4 = \underline{t_1 - t_3} \end{aligned} \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\}$$

reorder

$$\begin{aligned} t_2 &= c + d \\ t_3 &= c * t_2 \\ t_1 &= a + b \\ t_4 &= t_1 - t_3 \end{aligned}$$

MOV R₀ R_{0, a}.
 ADD R₀ R_{0, b}.
 MOV R_{0, t₁}.
 ADD d, R₁.
 MOV e, R₀.
 MUL R₁, R₀.
 MOV t₁, R₁.
 Sub R₀, R₁.
 MUL R₁, t₄

MOV C, R₀ -
 ADD d, R₀ -
 MOV e, R₁ -
 MUL R₀, R₁ -
 MOV a, R₀ -
 ADD b, R₀ -
 Sub R₂, R₀ -
 MUL R₀, t₄ -

Global Dataflow analysis

(Global) Dataflow Analysis

It collects the info about entire program & distributed this info to each block in the flow graph.

⇒ A typical data flow eqn:

$$\text{out}[s] = \text{gen}[s] \cup \{\text{in}[s] - \text{kill}[s]\}$$



$\text{out}[s]$ ⇒ Definitions that reach B's exit

$\text{gen}[s]$ ⇒ definitions within B that reach the end of B.

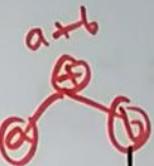
$\text{in}[s]$ ⇒ that reaches B's entry



$\text{kill}[s]$ ⇒ that never reach the end of B

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

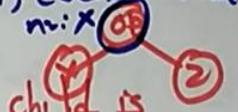
- Each node contains a label. For leaves, the label is an identifier
- Each node contains a list of attached identifiers to hold the computed values.

Case(i) $x := y \oplus z$

Case(ii) $x := op y$

Case(iii) $x := y$

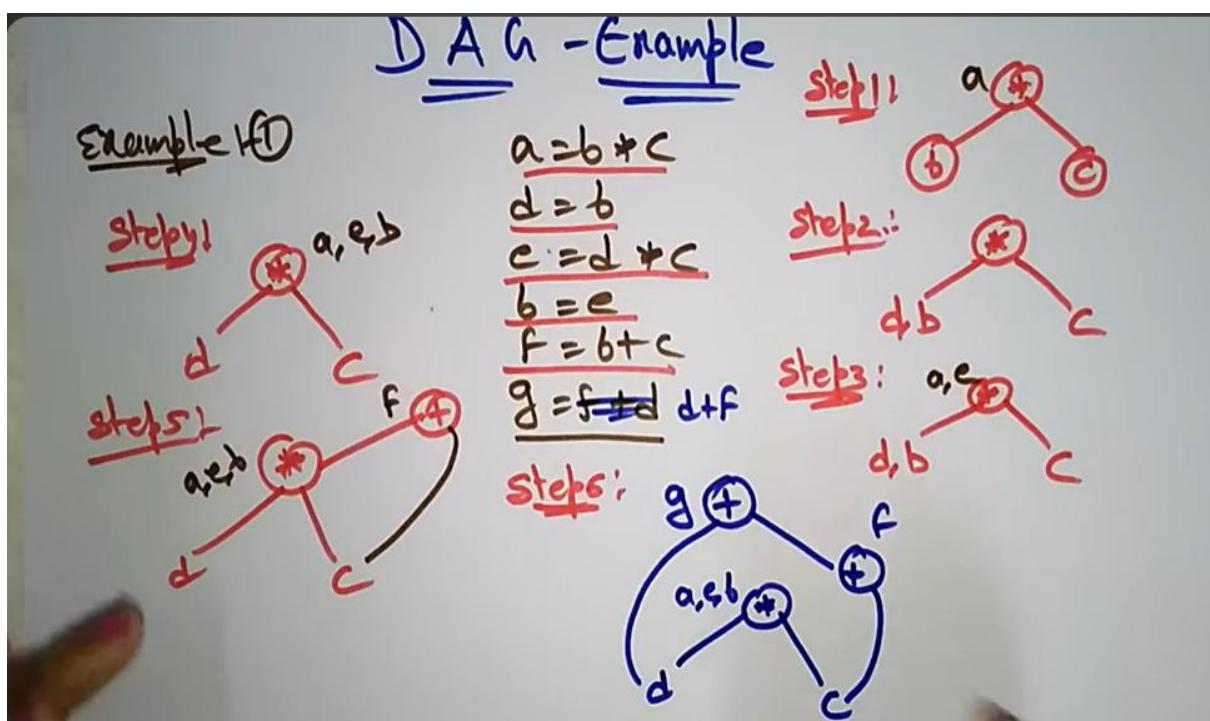
Method:

Step 1: If y operand is undefined then Create node(y).
 If z operand is undefined then for case(i) Create node(z).


Step 2: For case(i), Create node(op) whose right child is node(z) and left child is node(y). $x = op \ z$
 For case(ii), check whether there is node(op) with one child node(y). $x = op \ y$
 For case(iii), node n will be node(y). $x = y$

Output: For node(x) delete x from the list of identifiers.
 Append x to attached identifiers list for the node n found in
 Step 2. Finally set node(x) to n .

Directed Acyclic Graph (DAG) Examples



Example 2 -

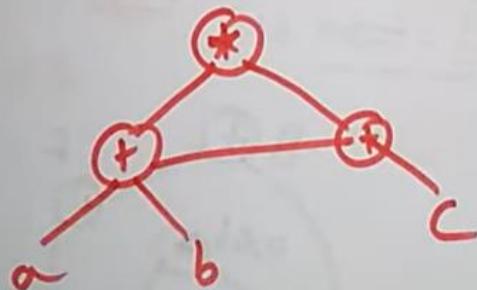
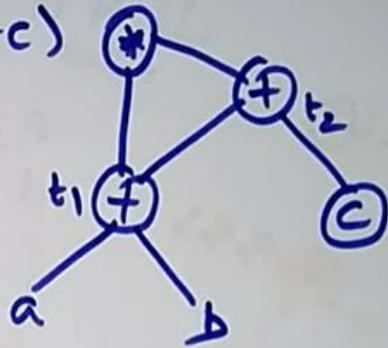
TAC \Rightarrow

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

$$t_1 = a+b$$

$$t_2 = t_1 + c$$

$$t_3 = t_1 * t_2$$



Peephole Optimisation

Peephole optimization

- This technique works locally on source code to transform it into an optimized code.
- The peephole optimization is a short seq of tangent inst. that can be replaced by shorter or faster seq inst.
- It examine at most a few inst. transforming inst. into other less expensive ones such as turning multiplication of x by 2 into an addition of x with itself.

Characteristics of peephole 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 b)

ADD R₀, c

STORE a, R₀

$$R_0 = b + c$$



$$a = b + c$$

1) LOAD R₀, b

2) ADD R₀, c

~~STORE a, R₀~~

~~LOAD R₀, a~~

3) ADD R₀, e

LOAD R₀, l 4) STORE d, R₀

ADD R₀, #1

STORE l, R₀

INC l

c) Use of M/C idioms

i = i + 1

1) LOAD R₀, l

2) ADD R₀, e

3) STORE d, R₀

4) STORE l, R₀

INC l

b) Flow of Control Optimization

i) Avoid Jumps on Jumps

L₁: jump L₂ L₄

1) Eliminate Dead Code.

+C
R₀

====

int i=0;

L₂: jump L₃

if (i==1)

L₃: jump L₄

printf ("Dead code");

L₄: x=a+b*c

}



② Unreachable Code :-

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

→ Programmers may have accidentally written a piece of code that can never be reached.

Eg: void add_ten (int n)

{ return n+10;

printf ("Value of n is %d", n);

}

In this statement if stmt will never be executed as program control returns back before it executes, hence P/D 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 reaches L₂.

MOV R₁, R₂
Goto L₂

MOV R₁, R₂
Goto L₂

L₂: INC R₁

① Algebraic Simplifications :-

There are occasions where algebraic expression can be made simple.

Ex: $a = a + 0$ // Can be replaced by a itself

$a = a + 1$ // Can simplify by replaced by $\text{INC } a$.

$$a = a + 1$$

Liveness Analysis

LIVENESS ANALYSIS

(Register Allocation)

Purpose:- Assigning multiple variables to single register without changing the program behaviour.

	Y

X is live variable at statement S_i iff .

1. There is a statement S_j using X. (Reading)
2. There is a path from S_i to S_j . (Reachability)
3. There is no new definition to X before S_j .

$$1. X = a + b$$

$$C = 4$$

$$2. Y = d + C$$

$$C = a + b$$

$$3. X = X + b$$

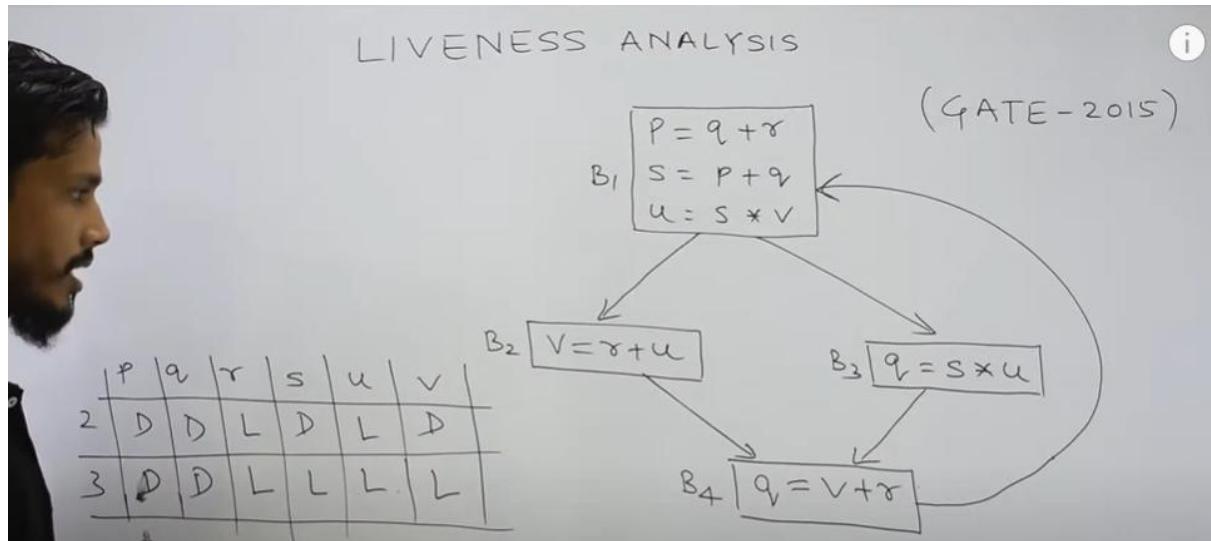
$$4. a = b + d$$

$$5. X = a + b + c$$



	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.



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?

$$R_1 \leftarrow 1$$

(GATE-2010)

$$R_2 \leftarrow 10$$

$$R_3 \leftarrow 20$$

$$\rightarrow R_1 d = R_1 + R_2$$

$$R_1 e = R_3 + R_1$$

$$R_2 f = R_3 + R_1$$

$$R_1 b = R_3 + R_1$$

$$R_1 e = R_1 + R_2$$

$$R_1 d = R_3 + R_1$$

return(d+f)

A) 2 B) 3

C) 4 D) 6

Code Generation

A Simple Code Generator Algorithm

- It generates target code for a sequence of instructions.
- It uses a function `getReg()` to assign registers to variables.
- It uses 2 data structures : 1. Register Descriptor
Register descriptor :- used to keep track of which variable is stored in a register. Initially all registers are empty.
2. Address Descriptor
Address descriptor :- used to keep track of location where variable is stored. Location may be register, memory address, stack, ...

A Simple Code Generator Algorithm

The following actions are performed by Code generator for an instruction $x = y \text{ op } z$. Assumes that L is the location where the output of $y \text{ op } z$ is to be saved.

1. Call function `getReg()` to get the location of L.
2. Determine the present location of 'y' by consulting Address descriptor of y. If y is not present in location 'L' then generate the instruction `mov y, L` to copy value of y to L.
3. If present location of z is determined using Step 2 & the instruction is generated as $\text{op } z, L$.
4. Now L contains the value of $y \text{ op } z$ i.e. Assigned to x. So, if L is a register then update its descriptor that it contains value of x. Update Address descriptor of x to indicate that it is stored in 'L'.
5. If y, z have no future use, then update the descriptors to remove y & z.

Simple Code Generation			
generator algorithm in compiler design	Ex: $d = (a-b) + (a-c) + (a-c)$	Once Address code :-	
R_0 t3	$t_1 = a - b$	$t_1 = a - b$	
R_1 t2	$t_2 = a - c$	$t_2 = a - c$	
	$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$	
	$d = t_3 + t_2$	$d = t_3 + t_2$	
Statement	Code Generation	Register Descriptor	Address Descriptor
$t_1 = a - b$	mov a, R0 Sub b, R0	R0 Contains t1	t1 in R0
$t_2 = a - c$	mov a, R1 Sub c, R1	R0 Contains t1 R1 Contains t2	t1 in R0 t2 in R1
$t_3 = t_1 + t_2$	Add R1, R0	R0 Contains t3 R1 Contains t2	t3 in R0 t2 in R1
$d = t_3 + t_2$	Add R1, R0	R0 Contains d	d in R0.

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 language

Register and Address Descriptors

1. Register Descriptor : - Keeps track of what is currently in each register

- Initially all registers are empty.

2. Address Descriptor : - keeps track of the location where the current value of the name can be found

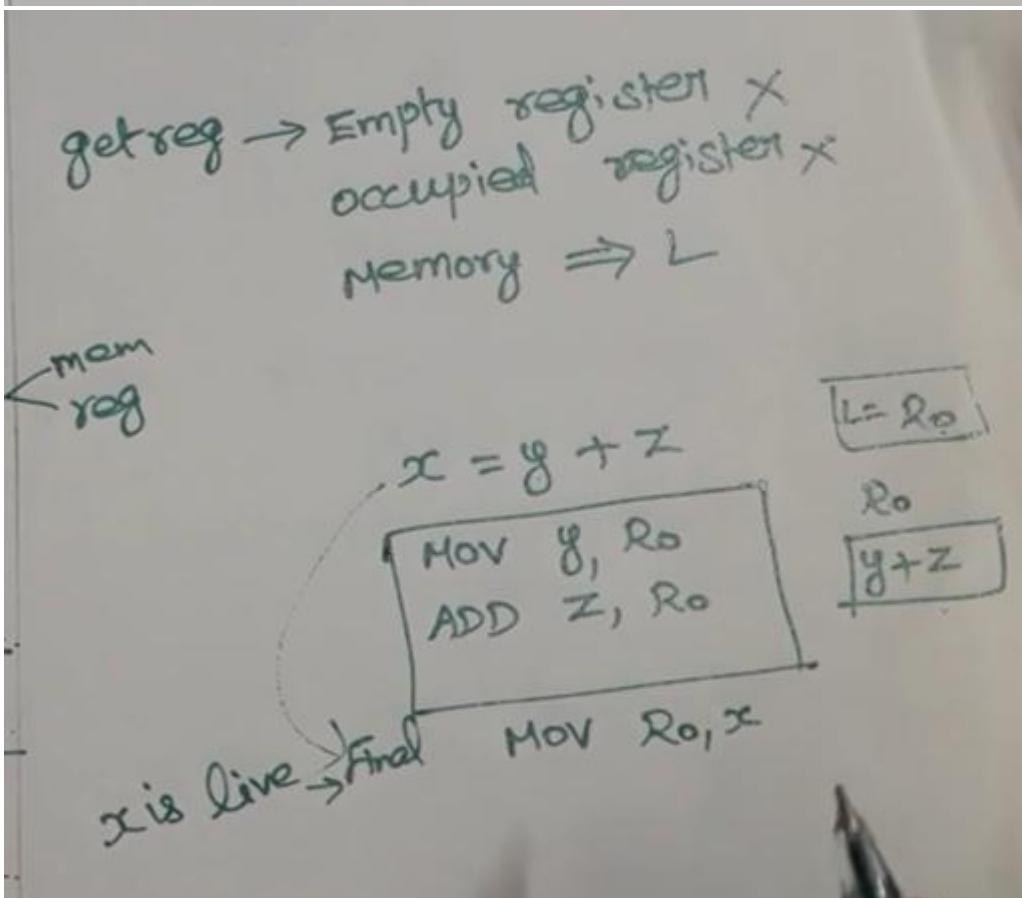
- Location may be register, a stack location or memory address

A code Generation Algorithm

for each three address statement
of the form $x = y \text{ op } z$,

1. Invoke a function getreg to determine the location L , where result of $y \text{ op } z$ 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 $\text{MOV } y', L$.
3. Generate the instruction $\text{op } z', L$.
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

$$t_1 = a-b$$

$$t_2 = a-c$$

$$t_3 = t_1 + t_2$$

$$d = t_3 + t_2$$

Statements	Code Generated	Register Descriptor Registers are empty R ₀ contains t ₁	Address Descriptor t ₁ in R ₀
t ₁ = a - b	MOV a, R ₀ SUB b, R ₀	R ₀ contains t ₁ R ₁ contains t ₂	t ₁ in R ₀ t ₂ in R ₁
t ₂ = a - c	MOV a, R ₁ SUB c, R ₁	R ₀ contains t ₃ R ₁ contains t ₂	t ₂ in R ₁ t ₃ in R ₀
t ₃ = t ₁ + t ₂	ADD R ₁ , R ₀	R ₀ contains d	d in R ₀
d = t ₃ + t ₂	ADD R ₁ , R ₀ MOV R ₀ , d		d in R ₀ and Memory

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

3

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.

4

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.

5

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.

6

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.

9

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

10

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 $<$, \leq , $>$, \geq , $==$.

- 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

Opcode(8)	x	Address (15)
-----------	---	--------------

- 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	$x = 0$	$TA = \text{address}$
Indexed	$x = 1$	$TA = \text{address} + (x)$

- 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 $<$, \leq , $>$, \geq , $==$.

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

D - x e

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 COMP R
- 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

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

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

- Program-Counter Relative Addressing Mode

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

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 ($0 \leq \text{disp} \leq 4095$)

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

$n=1, i=1, b=0, p=0, TA=(X)+\text{disp}$ (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

$n=1, i=0, x=0, TA = (\text{disp})$

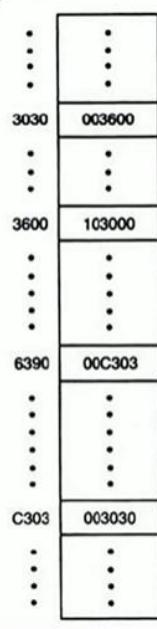
SIC/XE – Target Address Calculation

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

⋮	⋮
3030	003600
⋮	⋮
3600	103000
⋮	⋮
6390	00C303
⋮	⋮
C303	003030
⋮	⋮

SIC/XE – Target Address Calculation



(B) = 006000
(PC) = 003000
(X) = 000090

Machine instruction								Value loaded into register A
Hex	Binary						Target address	
	op	n	i	x	b	p	e	disp/address
032600	000000	1	1	0	0	1	0	0110 0000 0000
03C300	000000	1	1	1	1	0	0	0011 0000 0000
022030	000000	1	0	0	0	1	0	0000 0011 0000
010030	000000	0	1	0	0	0	0	0000 0011 0000
003600	000000	0	0	0	0	1	1	0110 0000 0000
0310C303	000000	1	1	0	0	0	1	0000 1100 0011 0000 0011

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 translates programs written in assembly language into machine language.

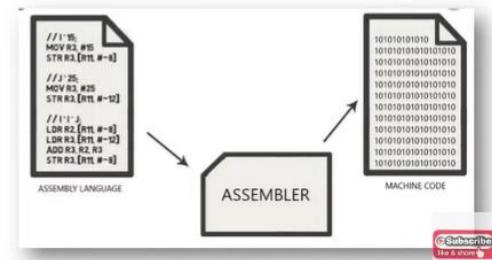


- **Assembly Language:** Assembly language is a kind of low level programming language, which uses symbolic codes or mnemonics as instruction.
- Some examples of mnemonics include ADD, SUB, LDA, and STA that stand for addition, subtraction, load accumulator, and store accumulator, respectively.



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

- 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.
- 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

- Data Declaration:** Data can be declared in a variety of notations including the decimal notation.

Example: NUM1 03 OR

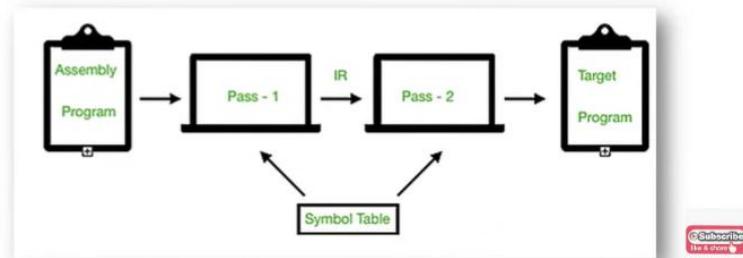
NUM1 0011H

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	Comparison
07	BC	Branch on condition
08	DIV	Division
09	READ	Reading memory
10	PRINT	Writing to memory

Assembly Process

- Convert .asm file into .obj file.

- Pass 1:** Complete scan .asm file. Find all labels, instructions & calculating corresponding address.
- Pass 2:** Convert all the instructions into machine language format.
- Symbol Table:** Stores 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

[Subscribe](#)

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.

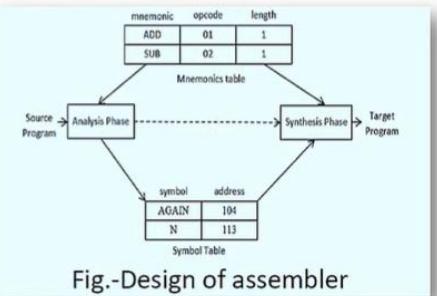


Fig.-Design of assembler



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

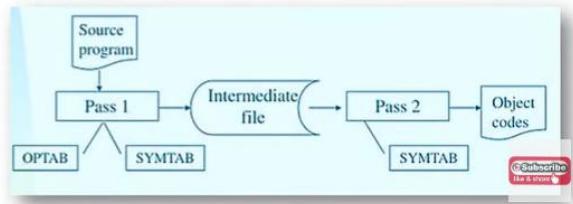
Assembly Program:

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 their 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: $\text{STL} \rightarrow 14$, $\text{JSUB} \rightarrow 48$ $LDA \#3$
2. Convert symbolic operands to their equivalent machine address.
 - Ex: $\text{Cloop} \rightarrow 100$ $STA \text{TABLE}$
3. Build machine instruction in the proper format (format 1, 2, 3 or 4).
4. Convert the data constant to internal machine representation.
 - Ex: $\text{EOF} \rightarrow 4546$ H
 T
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. *SUM START 4m*
- **END:** Indicate the end of the source and specify the first executable instruction in the program. *END locv*
- **BYTE:** Generate character or hexadecimal constant occupying as many bytes as needed to represent the constant. *✓ 3 bytes*
- **WORD:** Generate one-word integer constant. *RESB 10*
- **RESB:** Reserves the indicated number of bytes for a data area. *RESW 10*
- **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 = 1000*
- 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. *RESW 10
3 by 10*

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.
LDA → 00
- 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/
SYMTAB
- During Pass 1, labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses (from LOCCTR).
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

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✓ 400A 400B			TIX	COUNT
6	400C 400D 400E			JLT	LOOP
7	400F 4010 4011			STA	TOTAL
8	4012			RSUB	
9	4015		TABLE	RESW✓	2000(X3 = 6000 bytes)
10	5785		COUNT	RESW	1 0X1770
11	5788		ZERO	WORD	0
12	578B C,D		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

Line no	LOCATION COUNTER	LABEL	OPCODE	Operand	Object Code
1	81 ^{15 add} OP X 0 4000 ⁰⁰⁰⁰⁰⁰⁰¹⁰¹⁰¹	SUM	START	4000(H)	-
2	000000010101	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		✓ ZERO WORD	0	000000
12	578B TOTAL		→ RESW	1	-
13	578E → END		→ 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 2 col → 1 byte
- End Record → only one
- H^SUM → 7x6 → 4.2 col
004000^00178E → 21 bytes
- T^004000^15^045788^005788^18C015^2C5785^384006^0C578B^4C0000 → 15 bytes
- T^005788^3^000000 → 6 col → 3 bytes
- E^004000. → 1 byte

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)

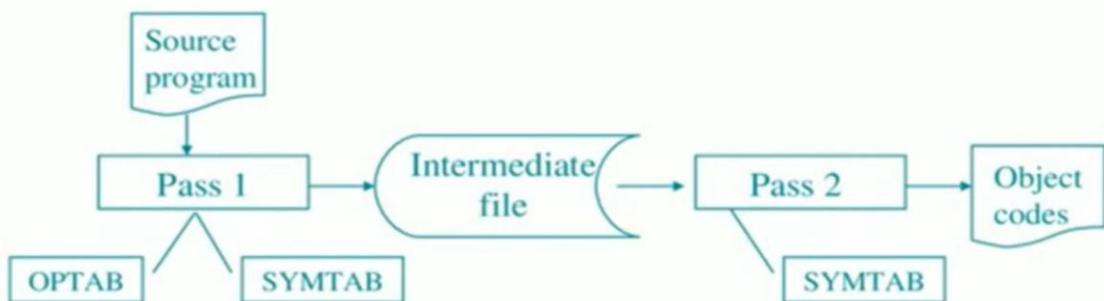
Text record:

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



Pass 1

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

```

```

while OPCODE != '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}

```

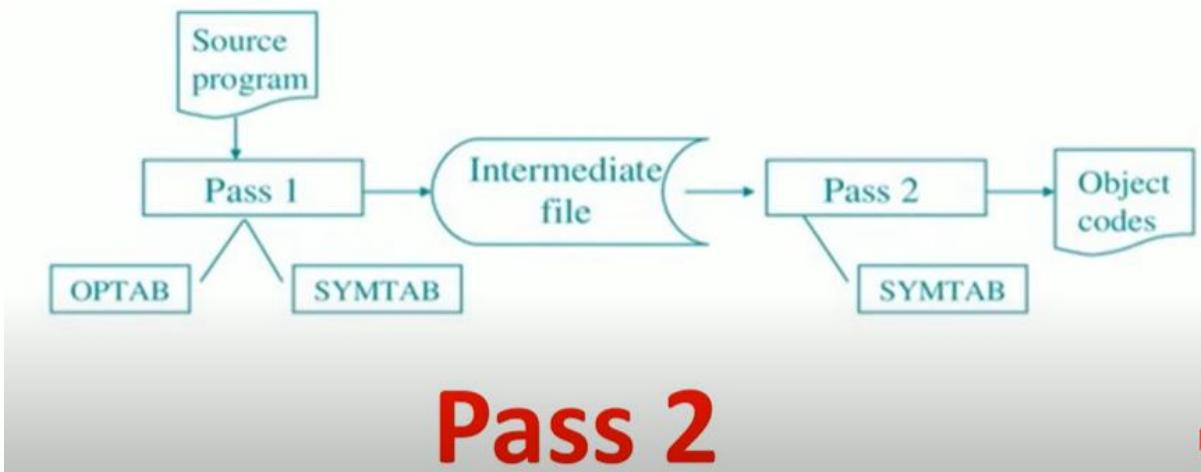
PASS – 1**Assembler**

Subscribe to Mahesh Huddar

Visit: vtupulse.com**SYMTAB**

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

Pass - 2 Assembler of Two-pass assembler



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 (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)
      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 (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)
        end
    end
end
  
```

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 (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)
    end
end
  
```

HⁿSUMⁿ 00400Vⁿ 00174ⁿ
Tⁿ 00h00Vⁿ 69ⁿ
Tⁿ — —
Eⁿ 00L00Vⁿ

How to generate Object Code and Object program for the SIC/XE 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

SIC/XE Machine – Generate Object Program – 1

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=

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		



Type 1 – 1 Byte

Type 2 – 2 Byte

Type 3 – 3 Byte

Type 4 – 4 Byte

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	Mnemonic	Number
10110100	0001	0000	A	0
			X	1
			L	2
			B	3
			S	4
			T	5
			F	6
			PC	8
			SW	9

CLEAR=B4 = 10110100

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

Subscribe

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
001110	1	1	0	0	1	0	FF7

JLT=38= 00111000

Disp = 0009 – 0012 = -9 = FF7

PC Relative: -2048 to 2047

[Subscribe](#)

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

[Subscribe](#)

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

[Subscribe](#)

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).

H^SUM__^000000^178E

T^000000^18^B410^010000^6910178

B^1BA00F^2F2009^3B2FF7^0F4000

^4F0000

E^000000

Length = 178E – 0000 = 178E

 Subscribe

Introduction to Macro in System Programming

MACRO

* Example

A 1,DATA	Add content of DATA to reg1
A 2,DATA	Add content of DATA to reg2
A 3,DATA	Add content of DATA to reg3
:	
A 1,DATA	Add content of DATA to reg1
A 2,DATA	Add content of DATA to reg2
A 3,DATA	Add content of DATA to reg3
:	

DATA DC F'5'

* Structure: MACRO → Start of Definition
() → MACRO Name
==== } Sequence to be abbreviated
MEND → End of Definition

* Macro definition, Macro Call &
Macro Expansion

MACRO

JAZZ

A 1,DATA

A 2,DATA

A 3,DATA

MEND

:

:

:

JAZZ

:

:

:

JAZZ

:

:

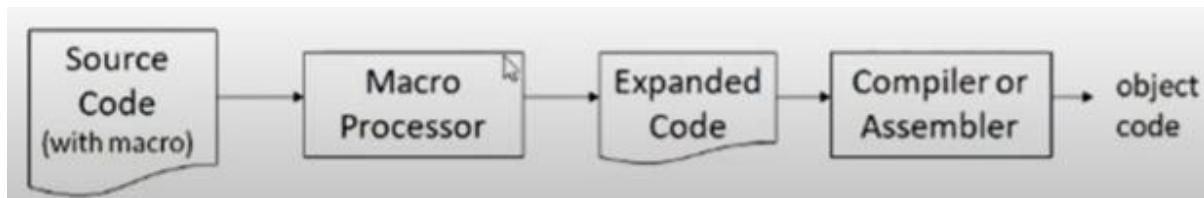
DATA DEF'S'

Expanded Source

{ A 1,DATA
 A 2,DATA
 A 3,DATA

{ A 1,DATA
 A 2,DATA
 A 3,DATA

Function can return a value while macro can't.



Macro vs Function

MACRO	Void f9()
JAZZ	{
A 1, DATA	g
A 2, DATA	f9();
A 3, DATA	f9();
MEND	

JAZZ	{ A 1, DATA A 2, DATA A 3, DATA

JAZZ	{ A 1, DATA A 2, DATA A 3, DATA
DATADCF'5'	

Macro

1. Macro increases the size of the Program
2. Macro don't alter the flow of Execution
3. Program using Macro shall get Executed faster
4. Macro cannot return value
5. Macro is useful when small code is repeated multiple times.

Function

function does not increase the size of a program

function alters the flow of execution

Program using function won't execute faster as compared to Macro

Function can return a value

Function is useful when large code is repeated multiple times.

Macro Instruction Arguments

PlanetOjas ▶

Macro Instruction Arguments

Source	Expanded Source code
MACRO	
JAZZ AARG	
A 1,AARG	{ A 1,DATA1
A 2,AARG	{ A 2,DATA1
A 3,AARG	{ A 3,DATA1
MEND	
JAZZ DATA1	{ A 1,DATA2
:	{ A 2,DATA2
DATA1 DCF'5'	{ A 3,DATA2
DATA2 DCF'10'	
JAZZ DATA2	
DATA1 DCF'5'	
DATA2 DCF'10'	

=> MACRO

```

# Example 2
      ALAB JAZZ AARG1,AARG2,AARG3
      ALAB A 1,AARG1
      A 2,AARG2
      A 3,AARG3
      MEND

      Loop1 A 1,DATA1
      A 2,DATA2
      A 3,DATA3
      MEND

      Loop2 A 1,DATA3
      A 2,DATA2
      A 3,DATA1

      DATA1 DC F'5'
      DATA2 DC F'10'
      DATA3 DC F'15'

      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

Nested Macro Call

MACRO

ADD1 λ ARG

A 1, λ ARG

A 2, λ ARG

A 3, λ ARG

MEND

MACRO

ADDS λ ARG1, λ ARG2, λ ARG3

ADD1 λ ARG1

ADD1 λ ARG2

ADD1 λ ARG3

MEND

```
MACRO  
ADD1 AARG  
A 1,AARG  
A 2,AARG  
A 3,AARG
```

```
MEND
```

```
MACRO  
ADDS AARG1,AARG2,AARG3  
ADD1 AARG1  
ADD1 AARG2  
ADD1 AARG3  
MEND
```

```
ADDS D1 D2 D3
```

```
D1 DC F'1'  
D2 DC F'2'  
D3 DC F'3'
```

Expansion
Level 1

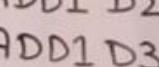
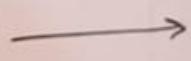
Expansion
Level 2

{ A 1,D1
A 2,D1
A 3,D1

{ A 1,D2
A 2,D2
A 3,D2

{ ADD1 D1
ADD1 D2
ADD1 D3

{ A 1,D3
A 2,D3
A 3,D3



Conditional Macro Expansion

Conditional Macro Expansion

:

Loop1 A 1, DATA1
A 2, DATA2
A 3, DATA3

:

Loop2 A 1, DATA3
A 2, DATA2

:

Loop3 A 1, DATA1

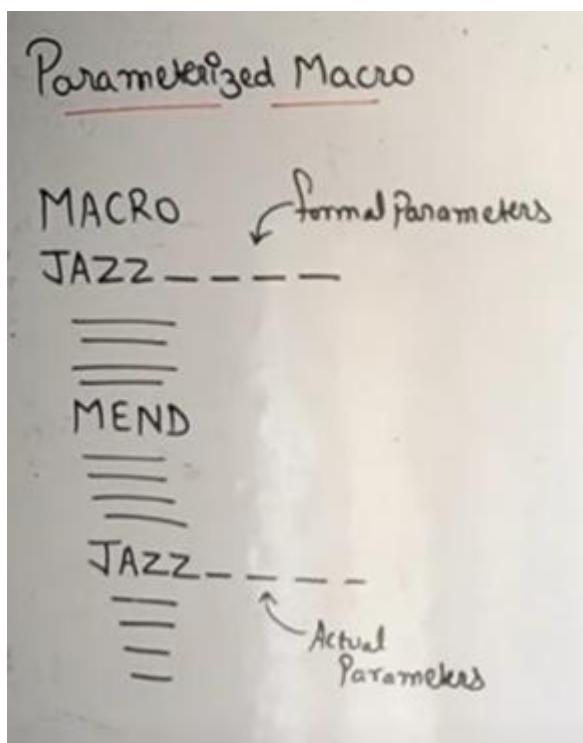
:

DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
:

MACRO
 AARG JAZZ ACOUNT,AARG1,AARG2,AARG3
 AARG A 1,AARG1
 AIF (ACOUNT EQ 1).FINI
 A 2,AARG2
 AIF (ACOUNT EQ 2).FINI
 A 3,AARG3
 .FINI MEND
 :
 Loop1 JAZZ 3,DATA1,DATA2,DATA3
 :
 Loop2 JAZZ 2,DATA3,DATA2
 :
 Loop3 JAZZ 1,DATA1
 :
 DATA1 DC F'S'
 DATA2 DC F'10'
 DATA3 DC F'15'
Expanded Source Code

Loop1 JAZZ 3,DATA1,DATA2,DATA3 : Loop2 JAZZ 2,DATA3,DATA2 : Loop3 JAZZ 1,DATA1	{ 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

Positional Parameter

MACRO

JAZZ <VAL1,>VAL2

A1,<VAL1

A2,>VAL2

MEND

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

====

JAZZ DATA1,DATA2 { A1,DATA1
A2,DATA2

JAZZ DATA9,DATA4 { A1,DATA9
A2,DATA4

END

Keyword Parameter

Keyword Parameter

MACRO

JAZZ XVAL1=, XVAL2=

A1, XVAL1

A2, XVAL2

MEND

====

JAZZ XVAL2= DATA1, XVAL1 = DATA2

=====

A1, DATA2

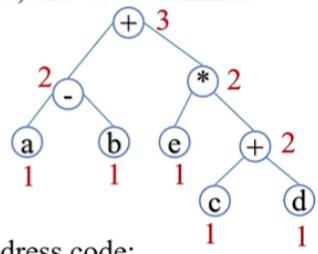
A2, DATA 1

END

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:
 - Label all leaves 1.
 - The label of an interior node with one child is the label of its child.
 - The label of an interior node with two children is
 - The larger of the labels of its children, if those labels are different.
 - 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:
- $t1 = a - b$
- $t2 = c + d$
- $t3 = e * t2$
- $t4 = t1 + t3$



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” $b \geq 1$ for the registers ,actual registers used are $R_b, R_{b+1}, \dots, R_{b+k-1}$. The result always appears in R_{b+k-1} .
- To generate machine code for an interior node with label k and two equal labels children:
 - Recursively generate code for the right child, using base $b + 1$. The result of the right child appears in register R_{b+k-1} .
 - Recursively generate code for the left child, using base b ; the result appears in R_{b+k-2} .
 - Generate the instruction $OP\ R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$, OP is the operation for the interior node.
- 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 $m < k$:
 - Recursively generate code for the big child, using base b ; appears in register R_{b+k-1} .
 - Recursively generate code for the little child, using base b ; appears in register R_{b+m-1} .
 - 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.
- 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

$b: 1$
 $k: 2$
 $R: 1+2-1$
 2

