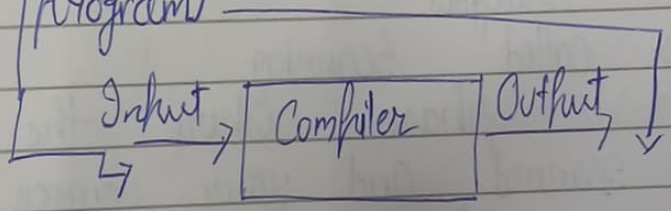
Ch - 1

Overview of COMPILER & its Structure

1) What is Compiler ? What is Front end and back end of compiler? [W-22, @-1(a)]

Ans Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program)



⇒ Front end Compiler
- Deals with analyzing source code and it Includes
  - → Lexical Analysis
  - → Syntax " "
  - → Semantic

⇒ Back end Compiler
Deals with code generation & Optimization
- Includes
  - → Intermediate Code Generation
  - → Code Optimization
  - → Target Code Generation

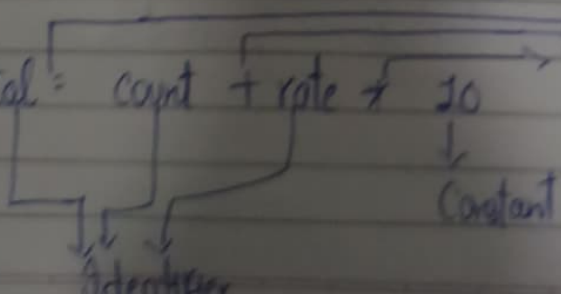
Q-2 Explain input, output and action performed by each phases of compiler with example [W-22 Q-1(c)]

Ans
1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation
5) Code Optimization
6) Code Generation

1) Lexical Analysis
→ Also called Scanning
→ It is phase where the complete source code is scanned and your source program is broken up into group of strings called token

Tokens
1) identifier x, y
2) assignment =
3) plus + } Operator
4) minus -
5) Constants
6) Seperator (,), {, }
7) Keyword int, else, while

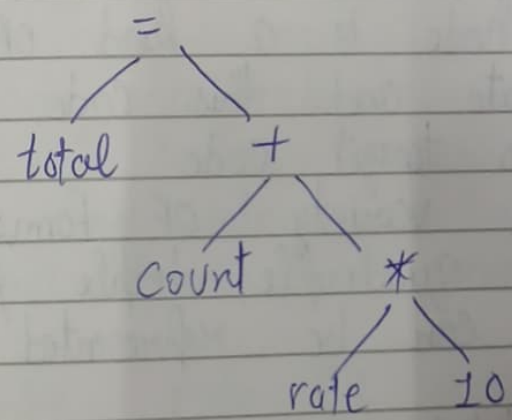
Example



id1 = id2 + id3 * 10

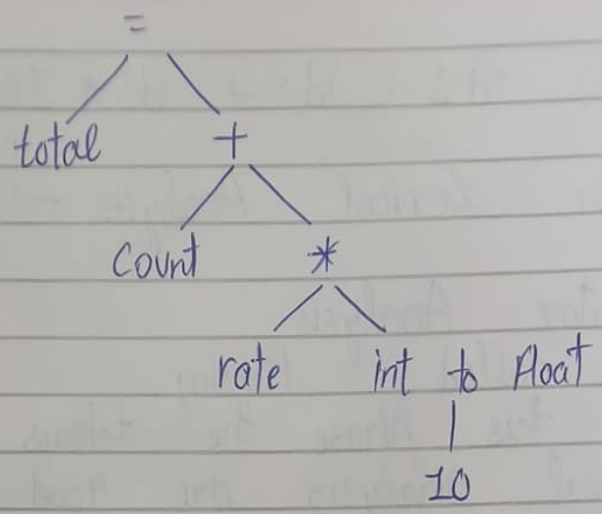
After Lexical Analysis

2) Syntax Analysis
→ Also called Parsing
→ In this phase the tokens generated by the lexical analyser are good grouped together to form a hierarchical structure



total = count + rate * 10

3) Semantic Analysis
→ This phase determines the meaning of the source string
⇒ It performs following operations
1) Matching of parenthesis in expression
2) Matching of IF.. else statement
3) Performing arithmatic operation that are type compatible
4) Checking the scope of operation
⇒ Example
total : count + rate * 10



4) Intermediate Code Generation

→ The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code

→ This code is in variety of forms such as three address code, quadruple, triple, postfix

→ Intermediate code can be represented in three address code

total = count + rate * 10

t_1 = int_to_float (10)
t_2 = rate x t_1
t_3 = count + t_2
total = t_3

5) Code Optimization

→ The code optimization phase attempts to improve the intermediate code

→ This is necessary to have a faster executing code or less consumption of memory

→ Thus overall running time of target program can be improved

Example

total = count + rate * 10

Intermediate code

t_1 = int_to_float(10)
t_2 = rate * t_1
t_3 = count + t_2
t_3 = total × [total = t_3]

↓

Code optimization

t_1 = rate * 10
total = count + t_1

6) Code Generation
→ In this phase the target code gets generated
→ The intermediate code instructions are translated into sequence of machine instructions

```
MOV   rate, R1
MUL   #10 , R1
MOV   count, R2
ADD   R2, R1
MOV   R1, total
```

Q-3 Explain role of linker, loader and preprocessor in the process of compilation [W-23, Q-1(a)]

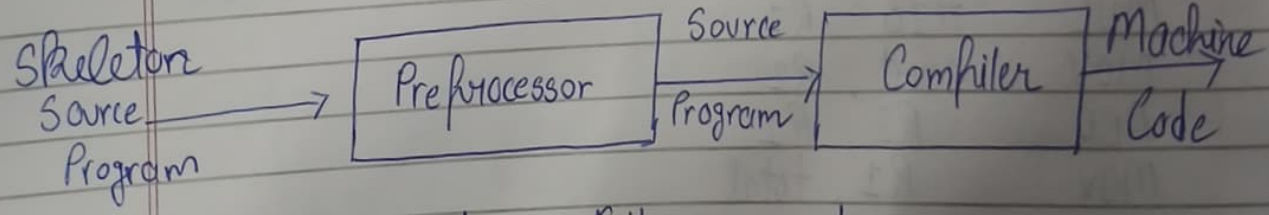
Ans 1) Linker
→ The linker combines multiple object files and library files into a single executable file
→ It resolves function calls between different modules

2) Loader

→ The process of loading consists of
1) Taking relocatable machine code
2) Altering the relocatable address
3) Placing the altered instructions and data memory at proper location

3) Preprocessor

→ Some tasks performed by preprocessor
1) Macro processing : Allows users to define macros
2) File inclusion : A preprocessor may include the header file into the program
3) Rational processor : It provides built in macro for construct like while or if statement
4) Language extensions : Add capabilities to the language using built-in-macros



Q-4 Differentiate following terms
1) interpreter and compiler
2) parse tree and syntax tree [W-24, Q-1(a)]

Ans 1) Interpreter & Compiler

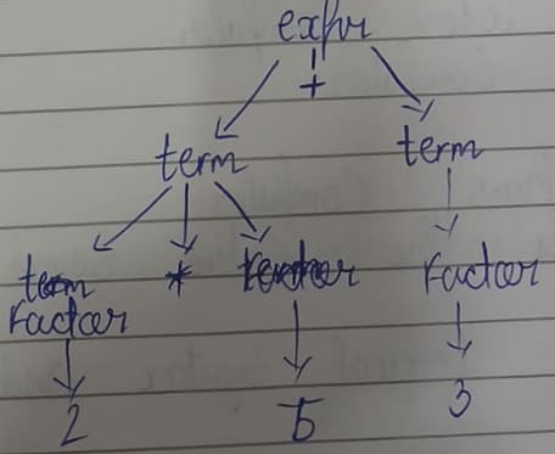
Feature	Interpreter	Compiler
1) Execution Method	Translates entire program into machine code at once	Translates and executes the program line by line

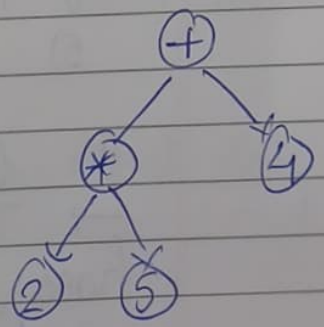
2)	Output	Generates an executable file	Does not
3)	Speed	Faster	Slower
4)	Error Detection	Shows all errors after compilation	Stops immediately when an error is found
5)	Examples	C, C++, JAVA	Python, JS, BASIC

2) Parse tree & Syntax tree

	Feature	Parse Tree	Syntax Tree
1)	Represents	Complete derivation using grammar	Simplified logical structure
2)	Contains	All grammar symbols	Only essential nodes
3)	Size	Larger	Smaller
4)	Use	Syntax Analysis	Semantic Analysis, Code Generation

5) Example
2*5+4





Page No.
Date

Q-5 State the applications of language processors [S-24, Q-1(a)]

Ans
1) Program Translation
2) Program Execution
3) Code Optimization
4) Error Detection
5) Debugging Code
6) Software Development Tools

Q-6 Enlist types of compiler. Explain any two in brief [S-24, Q-2(a)]

Ans Types of Compiler
1) One Pass Compiler
2) Two Pass Compiler
3) Incremental Compiler
4) Native Code Compiler
5) Cross Compiler.

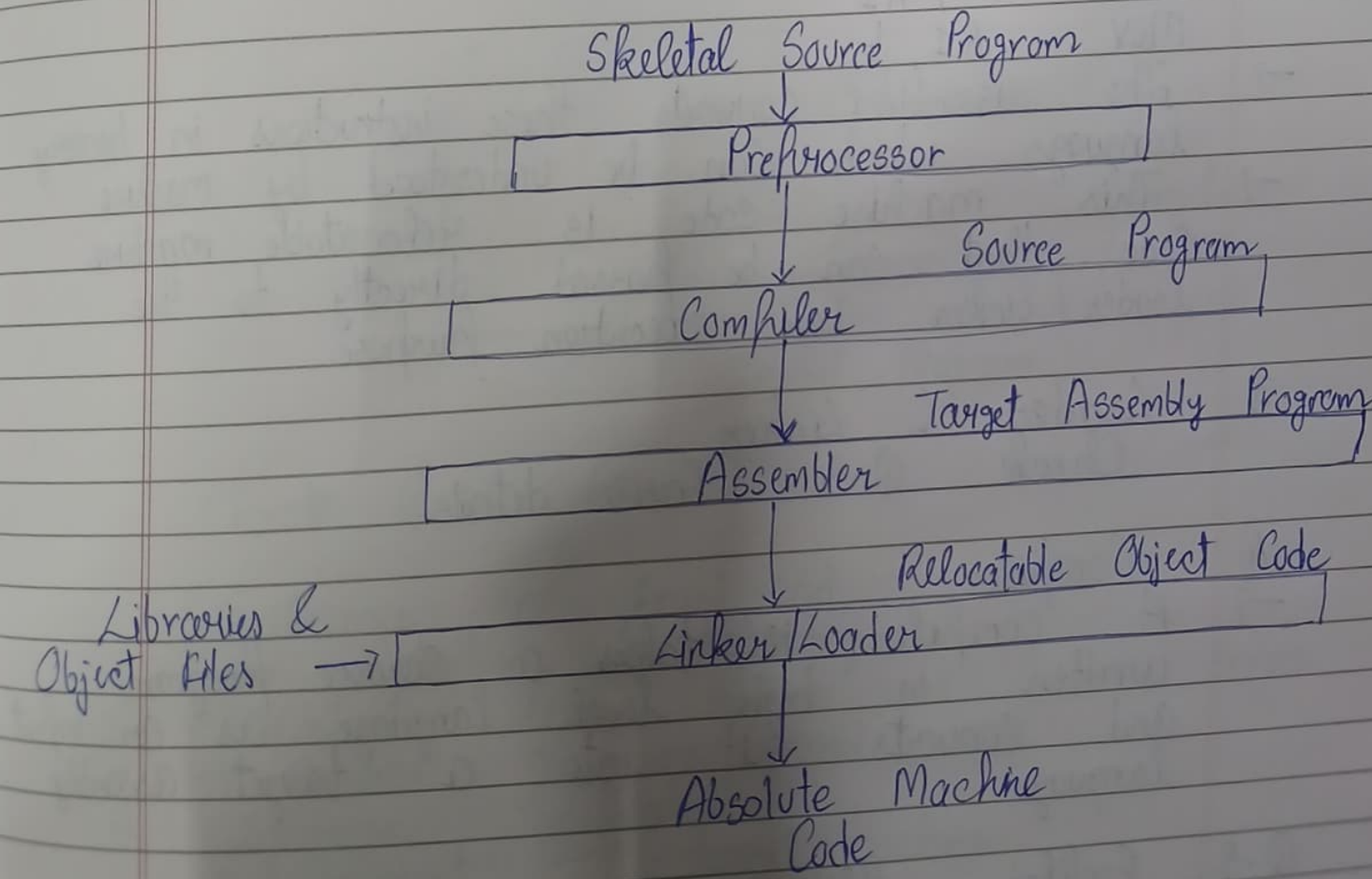
(1) One Pass Compiler
→ Translates the entire source code in one pass
→ Performs lexical, syntax and code generation together
→ Fast
→ Used for simple & small programs

2) Cross Compiler
→ Used to compile a source code for different platforms

→ It runs on one machine (host) but produces executable code for another machine (target) with different hardware or Operating System

Q-7 Explain cousins of compiler in detail [S-24, Q-1(c)]

Ans
1) Preprocessor
2) Compiler
3) Assembler
4) Linker / Loader



1) Preprocessor
Check Q-3

2) Assemblers

→ Some compilers produce the assembly code as output which is given to assemblers as an input

→ Assembler is a kind of translator which takes the assembly program as input and produces machine code as output

```
MOV   a, R1
MUL   #5, R1
ADD   #7, R1
MOV   R1, b
```

→ The assembler converts these instructions in binary language which can be understood by machine

→ This machine code is relocatable machine code that can be passed directly to the loader / linker for execution purpose

3) Loader & Linker

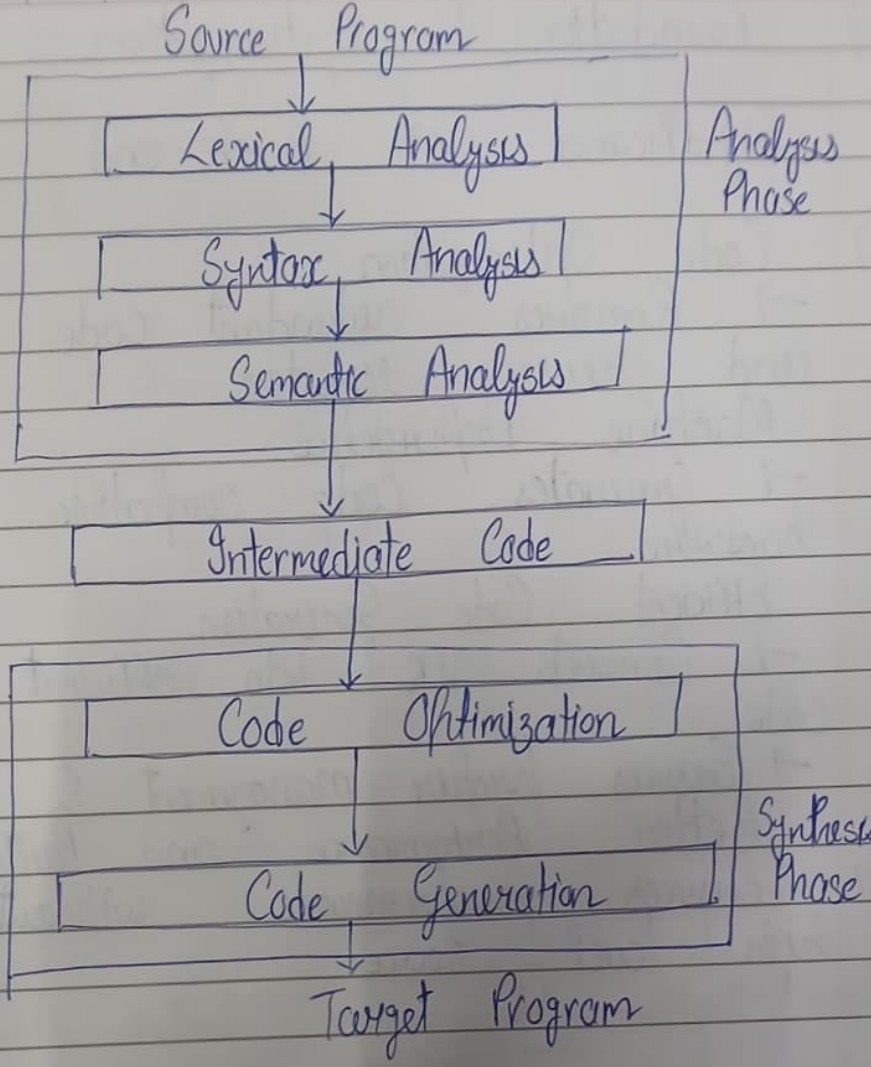
Check Q-3 for details

4) Compiler

→ A compiler is takes a source program written in high level language as an input and converts it into a target assembly language

Q-8 Explain phases of compiler. Give significance of both [5-25, Q-360]

Ans



Check Q-2 for more

Significance of Front-end

1) Error Detection & Reporting
   → Detects lexical, syntax and semantic errors
   → Ensures grammar and logic
2) Language Independence
   → Independent of target machine
3) Intermediate Code Generation
   → Produces IR (Intermediate Representation) which acts as bridge between Front-end and back-end

4) Foundation & Optimization

Significance of Back-end

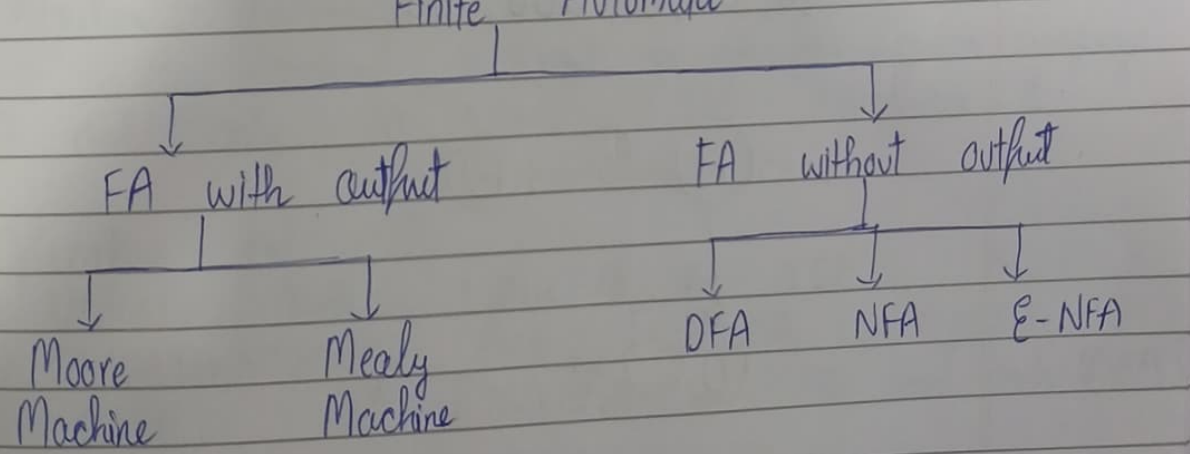
1) Code Optimization
→ Removes redundant code, optimizes loops, and reuses registers
2) Machine Dependence
→ Generates code compatible with target hardware
3) Efficient Code Generation
→ Converts IR into efficient, executable machine code
→ Ensures proper management & register allocation
4) Better Performance and Portability
→ Ensures performance without changing the high level source

Ch-2 Lexical Analysis

Finite State Machine

→ Symbol : a, b, c, 0, 1, 2, 3
→ Alphabet : Collection of symbols Eg: {a, b}
→ String : Sequence of symbols Eg a, b, 0, 1,
→ Language : Set of String
Eg : $\Sigma = \{0, 1\}$

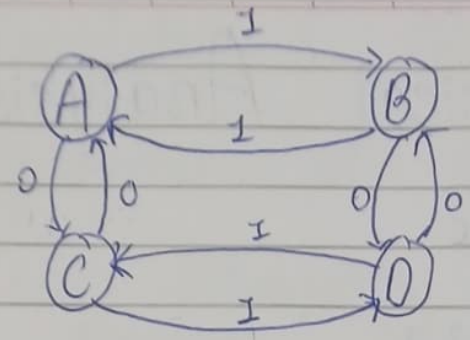
L_1 = Set of all strings of length 3
$= \{000, 001, 010, 011, 100, 101, 110, 111\}$



DFA

→ Simple
→ Limited Memory

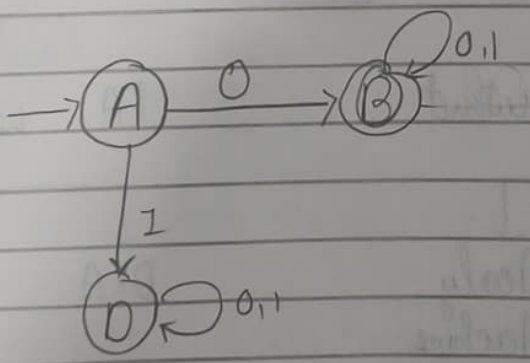
$(Q, \Sigma, q_0, F, \delta)$
Q = Set of all states
Σ = inputs
q_0 = Start State
F = Set of Final states
δ = transition Function from $Q \times \Sigma \to Q$



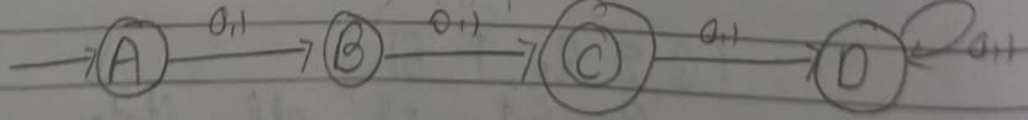
$Q = \{A, B, C, D\}$
$\Sigma = \{0, 1\}$
$q_0 = A$
$F = \{D\}$

	0	1
A	C	B
B	A	D
C	A	D
D	B	C

1) L_1 = Set of strings start with '0'
= {0, 00, 01, 000, 010



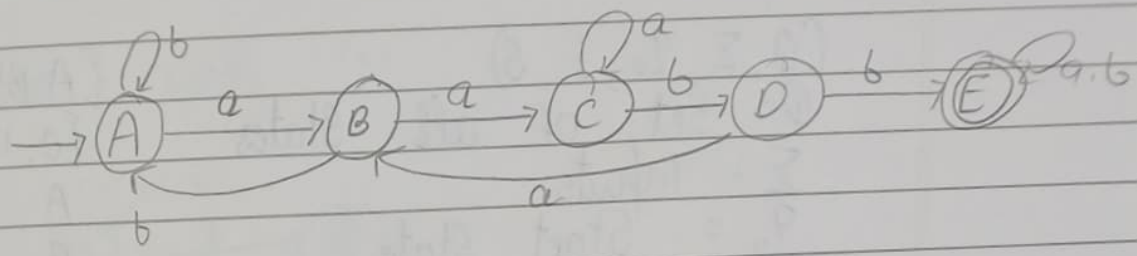
2) L_2 = {0,1} of length 2
= {00, 01, 11, 10}

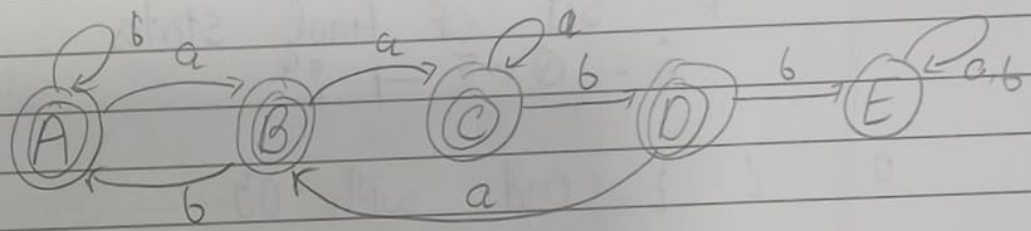


3) $\{a,b\}$ does not contain aabb

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

aaabbaa

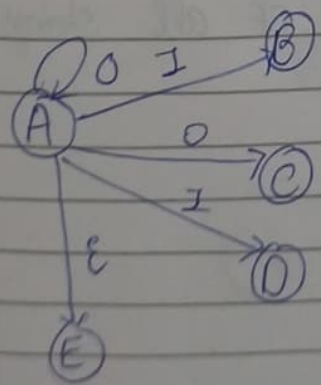




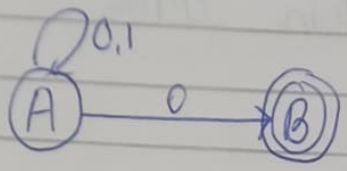
o Regular Language [RL]
→ A Language is said to be a Regular Language if and only if some FSM recognizes it

⇒ Operations on RL
1) UNION : $A \cup B = \{x | x \in A \text{ or } x \in B\}$
2) CONCATENATION : $AoB = \{xy | x \in A \text{ and } y \in B\}$
3) STAR : $A^* = \{x_1 x_2 x_3 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

o NFA - NON Deterministic FA



- L = { Set of all strings that end with 0 }



(Q, Σ, q_0, F, δ)

Q = Set of all states — {A, B}

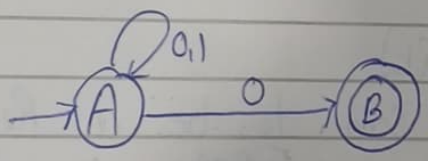
Σ = inputs — {0, 1}

q_0 = Start state — A

F = Set of Final states — B

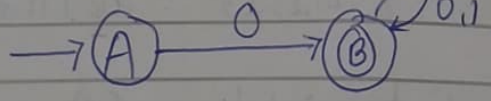
δ = Q × Σ → 2^Q

- L = { end with 0 }



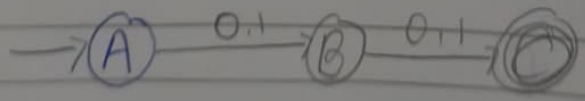
E

- L = { start with 0 }

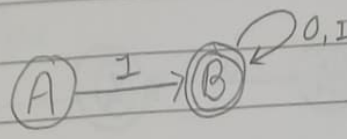


- NFA that accepts sets of all strings over {0,1} of length 2

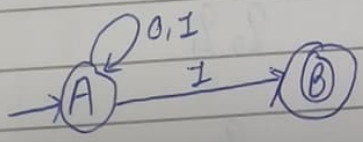
= {00, 01, 10, 11}



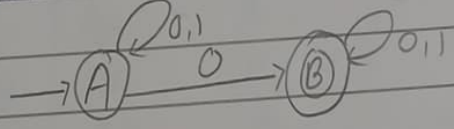
Eg: starts with '1'



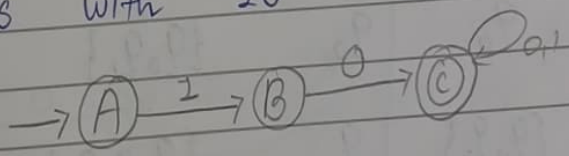
Eg ends with '1'



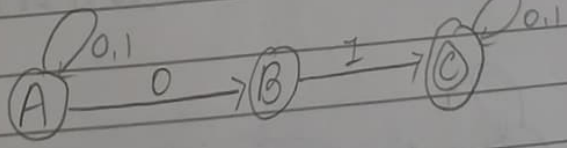
Eg: contain '0'



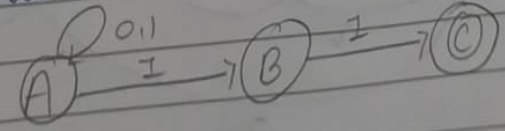
Eg: starts with '10'



Eg: Contain '01'

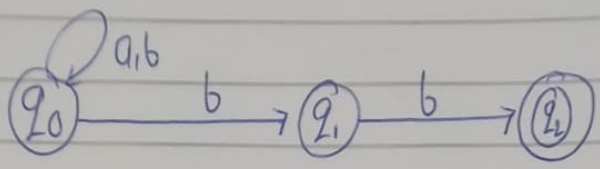


Eg: ends with 11



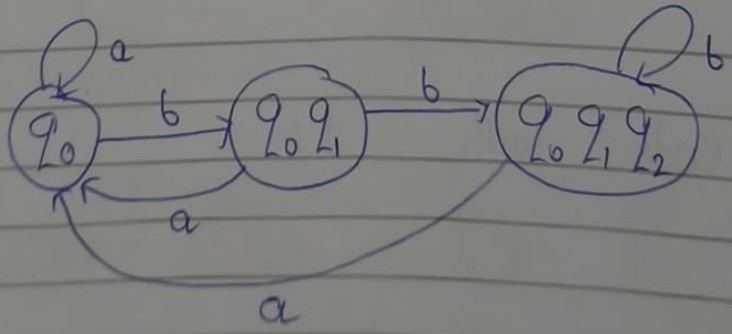
Page No.
Date

Conversion of NFA to DFA

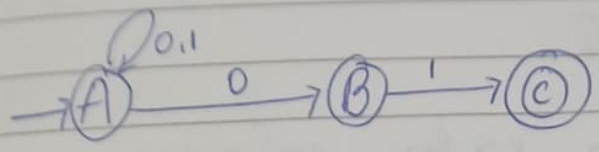


	a	b
$\rightarrow q_0$	q_0	$q_0 q_1$
q_1	-	q_2
(q_2)	-	-

	a	b
q_0	q_0	$\{q_0 q_1\}$
$\{q_0 q_1\}$	q_0	$\{q_0 q_1 q_2\}$
$\{q_0 q_1 q_2\}$	q_0	$\{q_0 q_1 q_2\}$

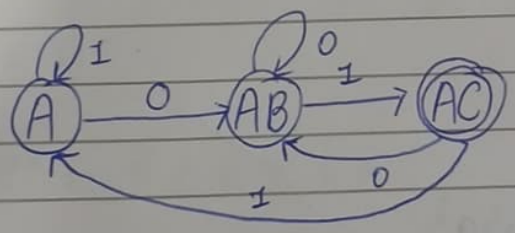


Eg:

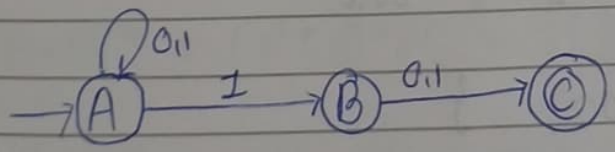


	0	1
→A	A,B	A
B	∅	C
(C)	∅	∅

	0	1
→A	AB	A
AB	AB	AC
(AC)	AB	A

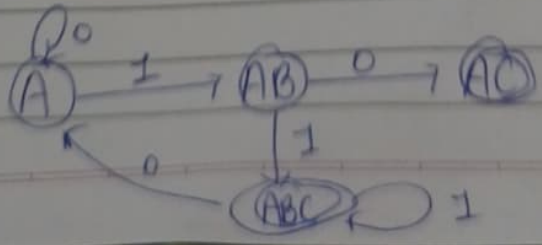


Eg: L = { second last symbol is always '1' }



	0	1
→A	A	AB
B	C	C
(C)	-	-

	0	1
→A	A	AB
AB	AC	ABC
(ABC)	AC	ABC
(AC)	A	AB
(ABC)	AC	ABC



Page No.
Date

Regular Expressions

1) $a, b, c \ldots \wedge, \emptyset$

2) Union of two regular expressions
$R_1, R_2 \quad (R_1 + R_2)$

3) Concatenation
$R_1, R_2 \quad (R_1 . R_2)$

4) $R \rightarrow R^*$

Eg: $\{0, 1, 2\}$
$R = 0 + 1 + 2$

Eg $\{\wedge, ab\}$
$R = \wedge ab$

Eg $\{abb, a, b, bba\}$

$R = abb + a + b + bba$

Eg $\{\wedge, 0, 00, 000, \ldots\}$

$R = 0^*$

Eg $\{\wedge, 1, 11, 111, \ldots\}$

$R = 1^*$

Identities of Regular Expression

1) $\emptyset + R = R$

2) $\emptyset R + R\emptyset = \emptyset$

3) $\epsilon R = R\epsilon = R$

4) $\epsilon^* = \epsilon$ & $\emptyset^* = \epsilon$

5) $R + R = R$

6) $R^* R^* = R^*$

7) $RR^* = R^*R$

8) $(R^*)^* = R^*$

9) $\epsilon + RR^* = \epsilon + R^*R = R^*$

10) $(P\emptyset)^*P = P(\emptyset P)^*$

11) $(P+\emptyset)^* = (P^*\emptyset^*)^* = (P^* + \emptyset^*)^*$

12) $(P+\emptyset)R = PR + \emptyset R$ and $R(P+\emptyset) = RP + R\emptyset$

Q/ Explain construction of DFA from RE without making NFA with suitable example [S-24, Q-2(c)]

↳ RE to DFA

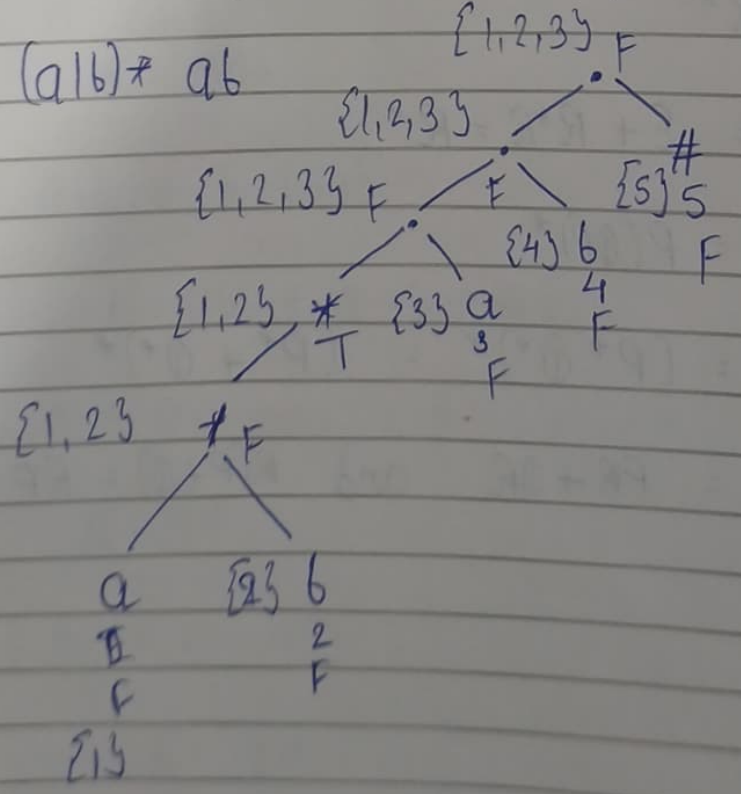
(a|b)* ab

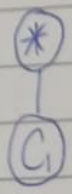
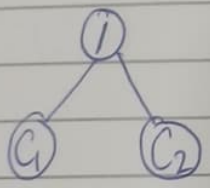
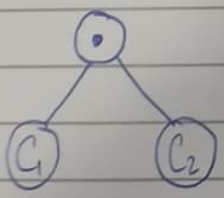
1) Create Syntax Tree
2) Number the leaf nodes
3) Find Nullable of each node
4) Find first pos of each node
5) Find last pos of each node
6) Find follow pos of each node

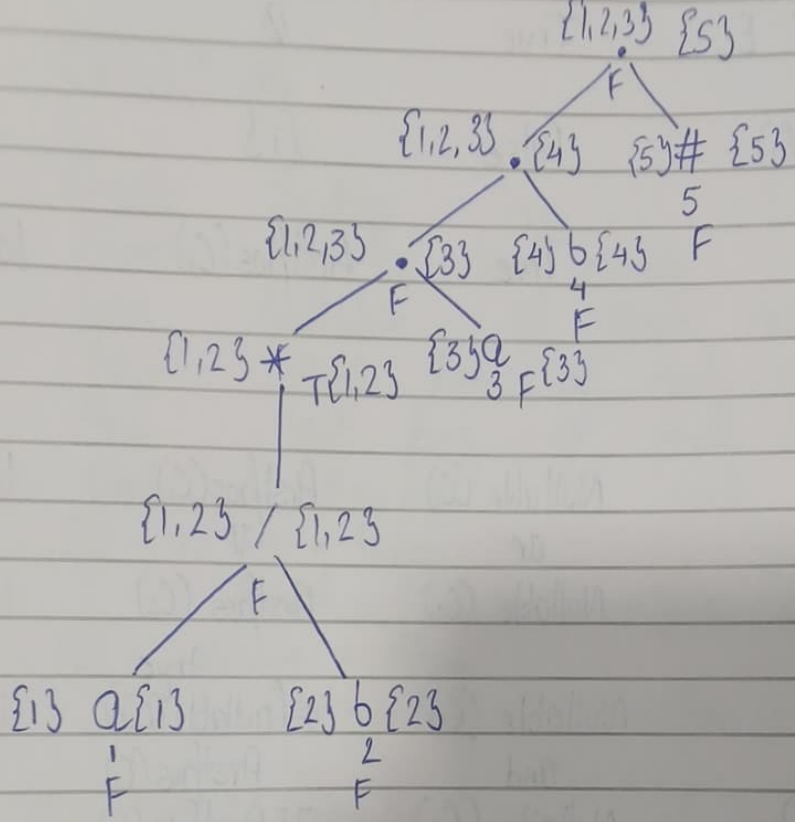
(i) $n \rightarrow \cdot$ then
firstpos $(c_2) \rightarrow$ lastpos (c_1)

(ii) $n \rightarrow *$ then
firstpos (n) $\rightarrow$ lastpos (n)

Eg. (a|b)* ab



	Node	Nullable	Firstpos(n)	Lastpos(n)
1)	N → Leaf E	True	∅	∅
2)	N → leaf (i)	False	{i}	{i}
3)		True	Firstpos(C1)	lastpos(C1)
4)		Nullable (C1) or Nullable (C2)	Firstpos(C1) ∪ Firstpos (C2)	lastpos (C1) ∪ lastpos (C2)
5)		Nullable (C1) and Nullable (C2)	if (nullable(C1)) Firstpos (C1) ∪ Firstpos (C2) else Firstpos (C1)	if (nullable (C2)) lastpos (C1) ∪ lastpos(C2) else lastpos(C2)



Node	Follow Pos
1	{3, 1, 2}
2	{3, 1, 2}
3	{4}
4	{5}
5	∅

FirstPos(Root Node) = {1, 2, 3} → (A)

1) (A, a) = {1, 3}
= FOP(1) ∪ FOP(3)
= {1, 2, 3, 4} → (B)

2) (A, b) = {2}
= FOP(2) = {1, 2, 3} → (A)

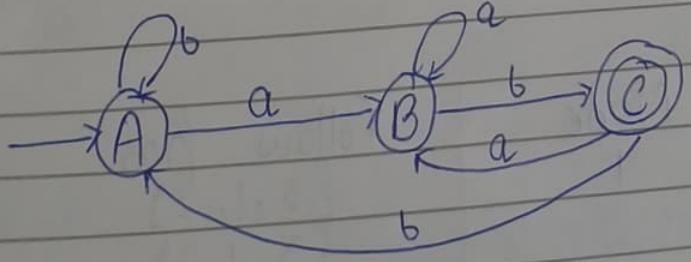
2) $(B, a) = \{1, 3\}$
$= \{1, 2, 3, 4\} \rightarrow (B)$

$(B, b) = \{2, 4\}$
$= FoP(2) \cup FoP(4)$
$= \{1, 2, 3, 5\} \rightarrow (C)$

3) $(C, a) = \{1, 3\}$
$= \{1, 2, 3, 4\} \rightarrow (B)$

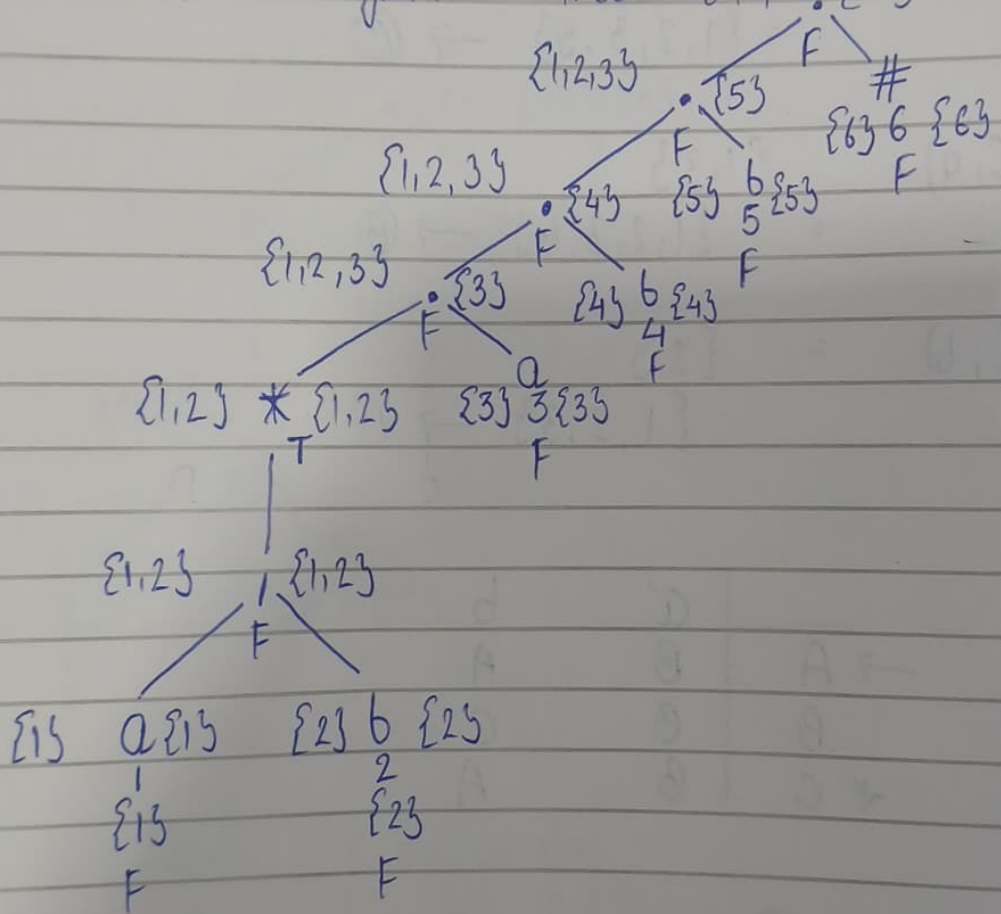
$(C, b) = \{2\}$
$= \{1, 2, 3\} \rightarrow (A)$

	a	b
→ A	B	A
B	B	C
* C	B	A



(a|b)* abb#

Construct Syntax tree



Node	Follow Pos
1	{3, 1, 2}
2	{3, 1, 2}
3	{4}
4	{5}
5	{6}
6	∅

First Pos (Root Node) = {1,2,3} → A

1) (A, a) = {1,2,3} {1,3}
= FoP(1) ∪ FoP(3)
= {1,2,3,4} → (B)

(A, b) = {2}
= {1,23} → (A)

2) (B, a) = {1,2,3,4} {1,3}
= FoP(2) ∪ FoP(4) {1,2,34} → (B)
= {1,2,3,5} → (C)

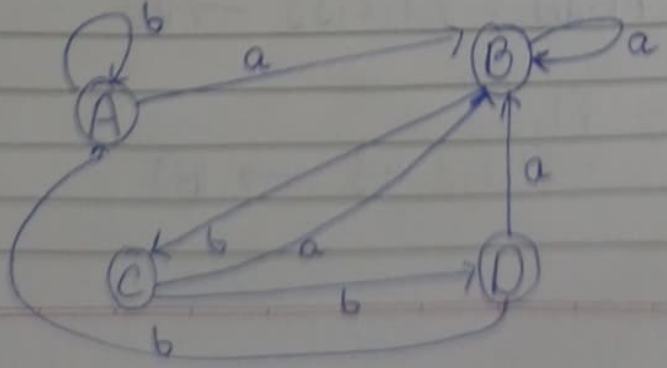
(B, b) = {2,4}
= {1,2,3,5} → (C)

3) (C, a) = {1,3}
= {1,2,3,4} → (B)

(C, b) = {2,5}
= {1,2,3,6} → (C) (D)

4) (D, a) = {1,3}
= {1,2,3} → (A) {1,2,3,4} → (B)

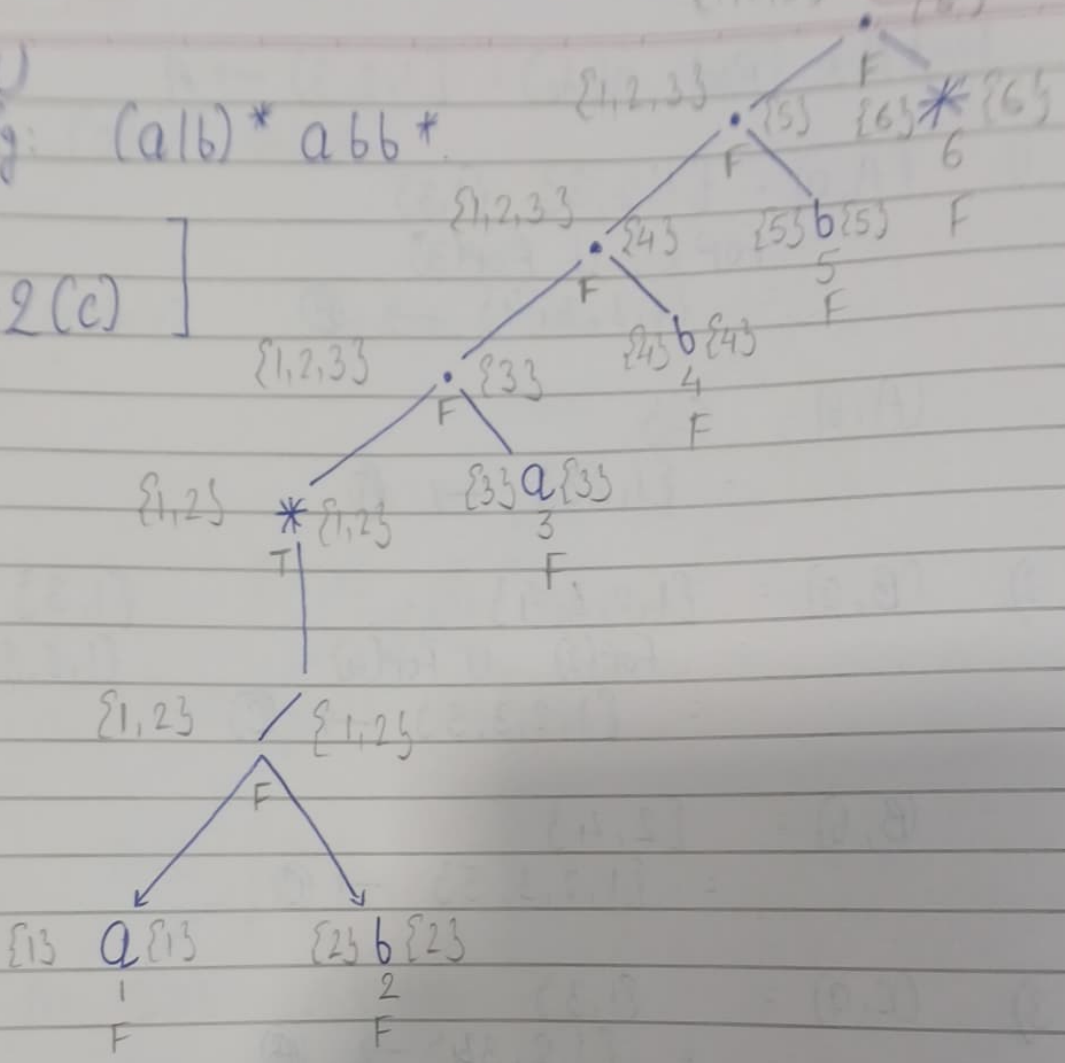
(D, b) = {2}
= {1,2,3} → (A)



GTU

Eg: $(a|b)^* abb^*$

[Q-22 OR Q-2(c)]



Node	Follow Pos
1	{3,1,2}
2	{3,1,2}
3	{4}
4	{5}
5	{6}
6	∅

FirstPos(Root Node) = {1,2,3} → (A)

1) (A, a) = {1,3}
= {1,2,3,4} → (B)

(A, b) = {2}
= {1, 2, 3} → (A)

2) (B, a) = {2, 3}
= {1, 2, 3, 4} → (B)

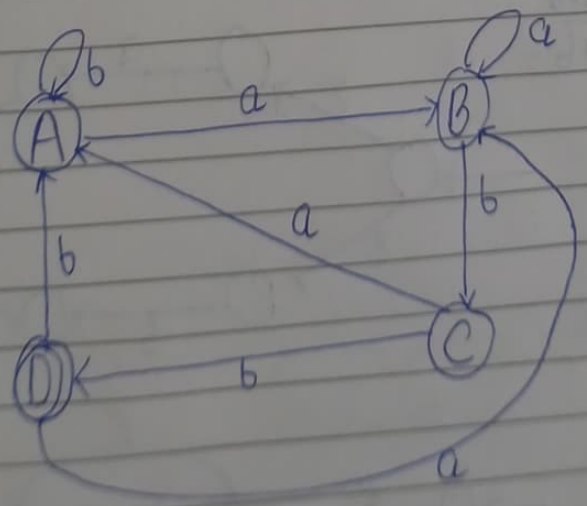
(B, b) = {2, 4}
= {1, 2, 3, 5} → (C)

3) (C, a) = {1, 2}
= {1, 2, 3} → (A)

(C, b) = {2, 5}
= {1, 2, 3, 6} → (D)

4) (D, a) = {1, 3}
= {1, 2, 3, 4} → (B)

(D, b) = {2}
= {1, 2, 3} → (A)

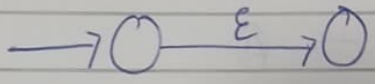


Regular Expression to NFA conversion using Thompson's rule

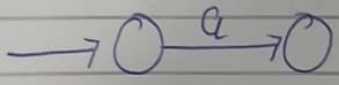
(a|b)*

Thompson's Rules

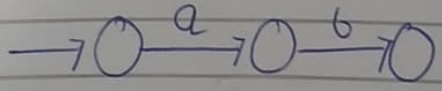
1) For ε,



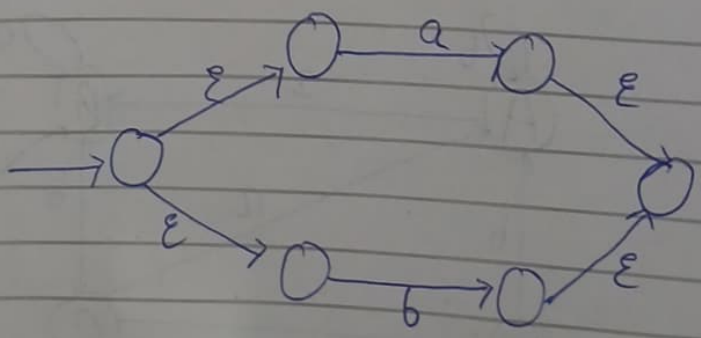
2) For a,



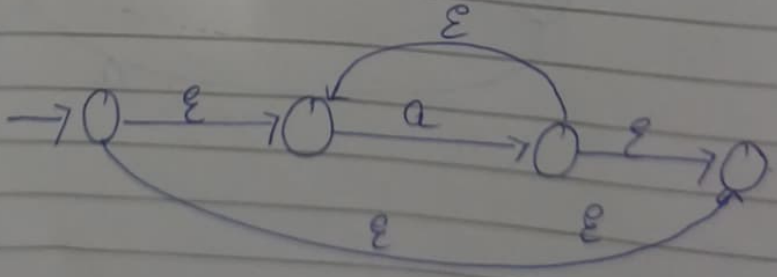
3) For a.b,



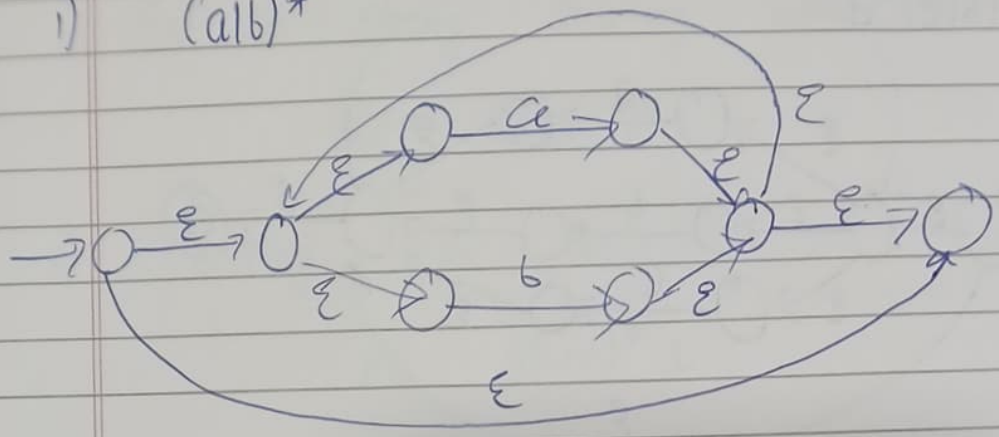
4) For $\frac{a|b}{a+b}$



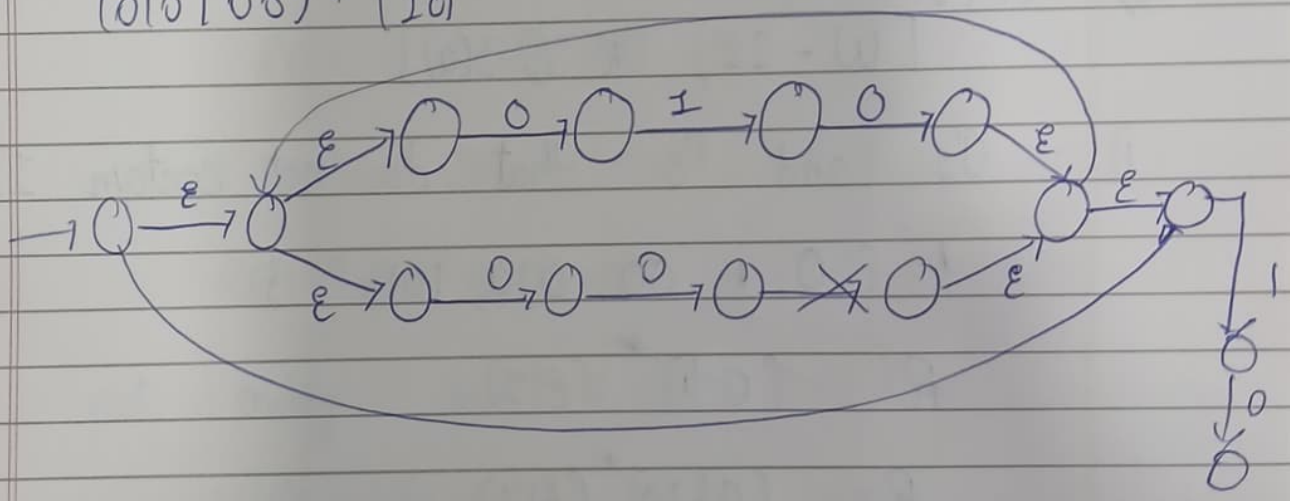
5) For a*,



1) (a|b)*

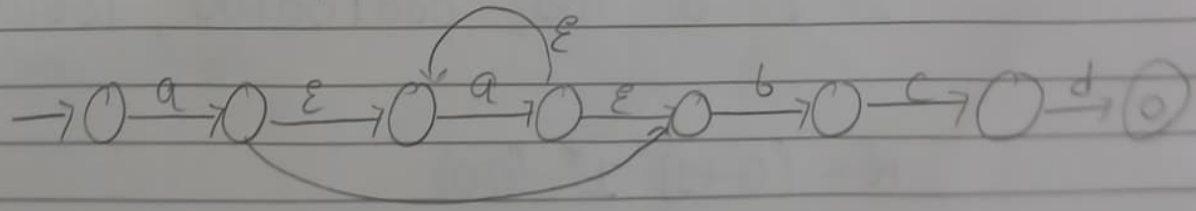


2) (0|0|00)* (10)

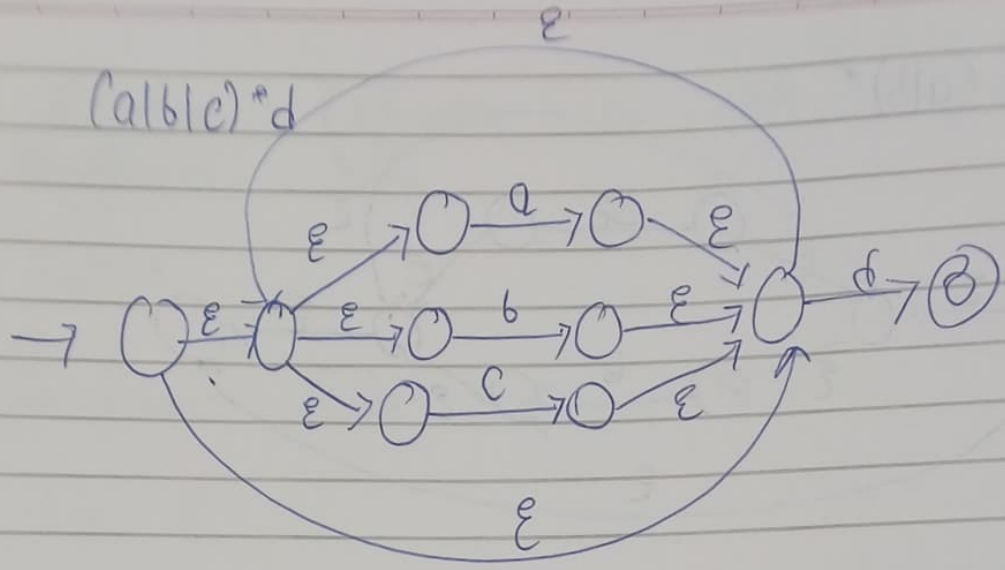


3) a^+bcd

$a.a^*bcd$



4) (a|b|c)*d



Q// GTU

[W-22, OR Q-3(a)]

1) 0's and 1's that do not contain 11

L = { 0, 1, 00, 010, 101, ... }

~~R = (0|1)* (ε|1)~~

R = (0|10)* (ε|1)

2) All strings of 0's and 1's that every 1 is followed by 00

L = { 0, 100, 000100100, 100100 }

R = (0+1)* 1* . (00)

R = (0|100)*

1) Set of string starting with 0

RE = $0(0+1)^*$

2) ending in 1

RE = $(0+1)^* 1$

3) containing exactly two 0's

RE = $1^*01^*01^*$

4) atleast two 0's

RE = $1^*01^*0(1+0)^*$

5) not ending in 01

RE = $(1+0)^*(00+11+10)$

6) starting with 11

$11(1+0)^*$

7) even length

RE = $(00)^*$

$L = \{\varepsilon, 0$

atleast one 0 and atleast one 1

$RE = (0+1)^* 0 (0+1)^* 1 (0+1)^*$

atleast two occurences of 1's between any two occurances of 0's

$(011 1^* 0)^*$

$RE = (1 + (011 1^* 0))^*$

GTU

[W-23, Q-2(a)]

1) containing exactly three 1's

✓ $RE = 0^* 1 0^* 1 0^* 1 0^*$

2) begins or ends with 00

× $RE = 00 + (0+1)^* + 00$

× $RE = 00 + (0+1)^* + 00$

✓ $RE = 00(0+1)^* + (0+1)^* 00$

3) contains atleast three 1's

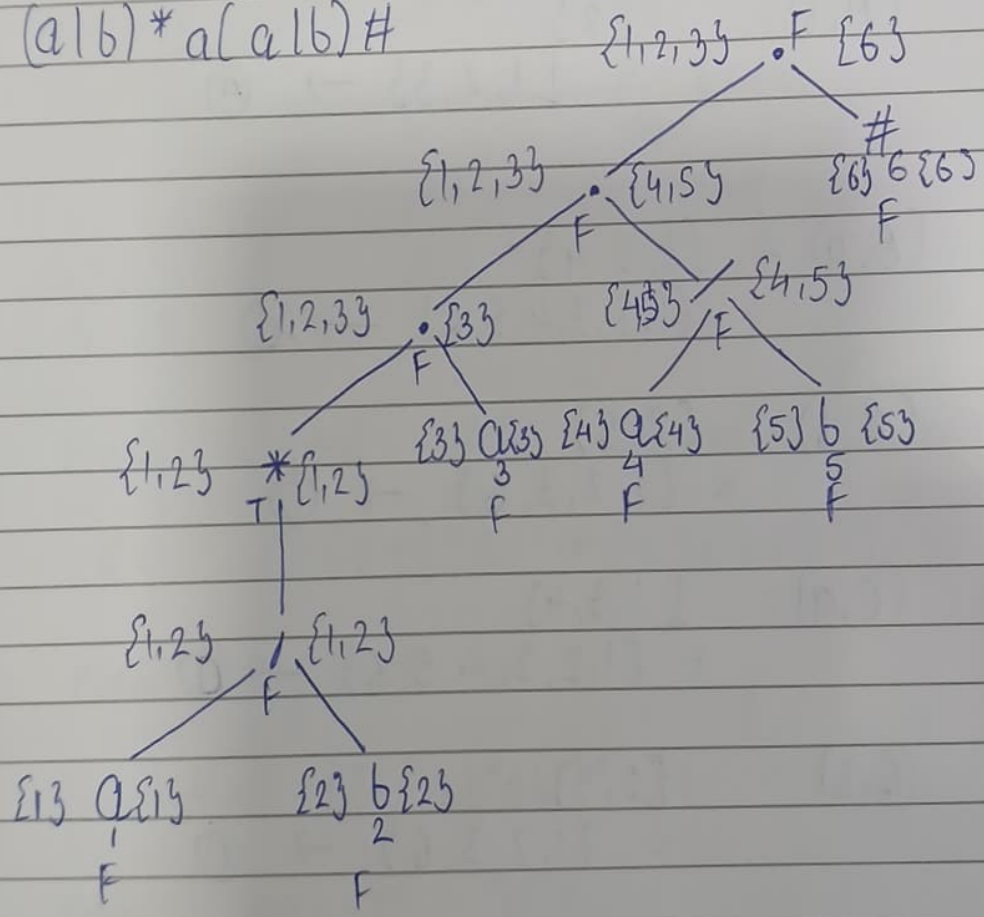
× $RE = (0+1)$

× $RE = 0^* 1 0^* 1 0^* 1 (0+1)^*$

RE = $(0+1)^*1(0+1)^*\ 1\ (0+1)^*1(0+1)^*$

Q. W-23. Q-2(c)

$(a|b)^*a(a|b)\#$



Node	Follow has
1	{3, 1, 2}
2	{3 1, 2 3
3	{4, 5 }
4	{6 }
5	{6 }
6	∅

Root node = {1, 2, 3} → A

1) (A, a) = {2, 3}
= {1, 2, 3, 4, 5} → (B)

(A, b) = {2}
= {1, 2, 3} → (A)

2) (B, a) = {1, 3, 4}
= {1, 2, 3, 4, 5, 6} → (C)

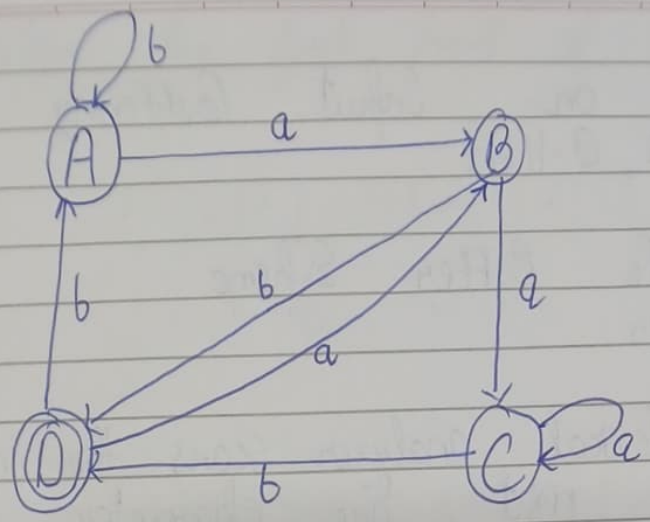
(B, b) = {2, 5}
= {1, 2, 3, 6} → (D)

3) (C, a) = {1, 3, 4}
= {1, 2, 3, 4, 5, 6} → (E)

(C, b) = {2, 5}
= {1, 2, 3, 6} → (D)

4) (D, a) = {1, 3}
= {1, 2, 3, 4, 5} → (B)

(D, b) = {2}
= {1, 2, 3} → (A)



Q// GTU
S-25, Q-2(b)

1) starts & end with different symbols

R: $0(0+1)^*1 + 1(0+1)^*0$

2) even no. of "a"

R: $b^* + (ab^*ab^*)^*$

3) odd no. of "b"

R: $a^*(ba^*ba^*)^*ba^*$

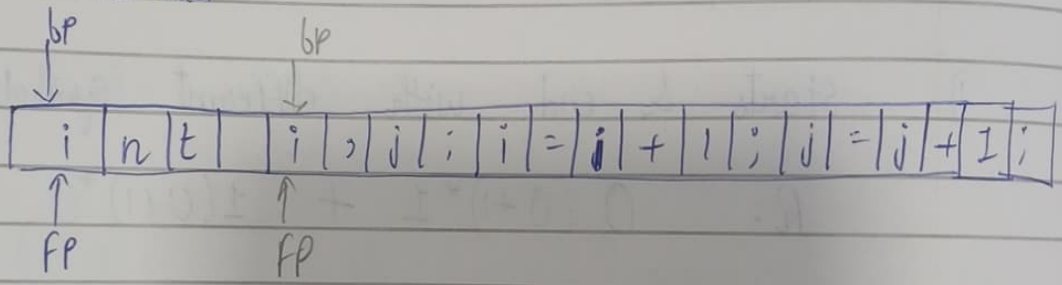
4) Atleast two times a

R: $(a+b)^*a(a+b)^*a(a+b)^*$

Q. Note on Input Buffering techniques [W-22, Q-1(b)]

Ans 1) Single Buffer Scheme
2) Two " " " "

→ The lexical analyzer scans the input string from left to right one character at a time
→ It uses two pointers begin_ptr (bp) and forward_ptr (fp) to keep track of portion of input scanned

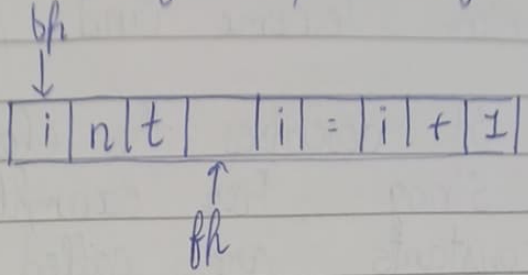


The input character is thus read from secondary storage. But reading in this way is costly

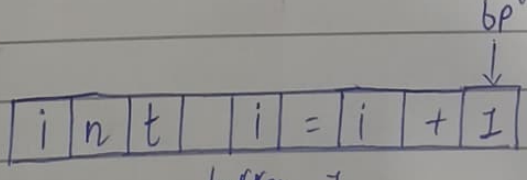
→ So comes Input Buffering
→ A block of data is first read into a buffer, and then scanned by lexical analyzer
Two methods

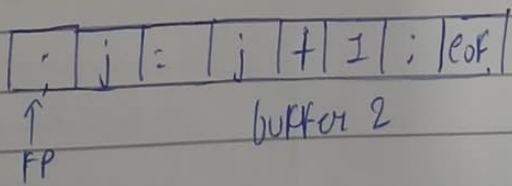
1) One buffer Scheme
→ In this scheme, only one buffer is used to store the input string
→ But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of lexeme the buffer has to be refilled, that makes

overwriting the first part of lexeme



2) Two buffer Scheme
→ To overcome problem of (1) this came
→ the first and second buffer are scanned alternatively
→ When end of current buffer is reached the other buffer is filled
→ Only problem is that if length of lexeme is longer than length of the buffer then scanning input cannot be completely scanned





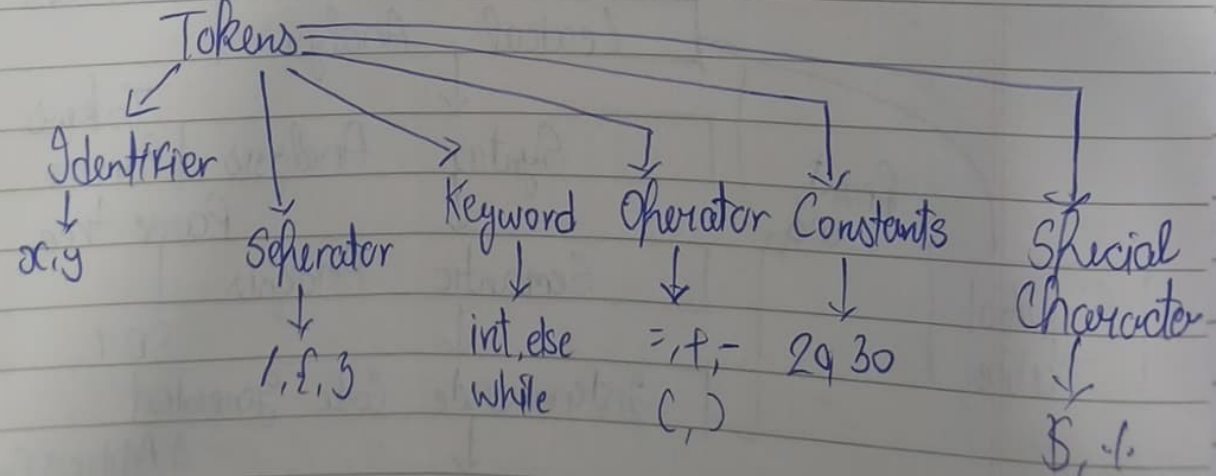
The eof character introduced at end is called Sentinel which is used to identify the end of buffer

Lexical Analysis

1) Tokenization

2) Give Error messages

3) Eliminate Comments, White Space



Q/ Define token, lexeme and pattern

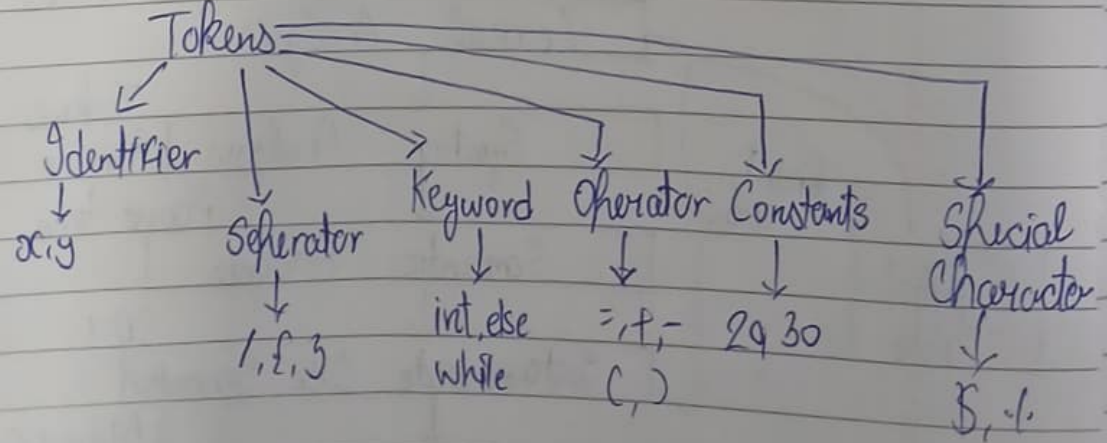
Ans Tokens: It describes the class or category of input string. For example, identifiers, keywords, constants are called tokens

Lexical Analysis

1) Tokenization

2) Give Error messages

3) Eliminate Comments, White Space



Pattern: set of rules that describe the token

Lexemes: Sequence of characters in the source program that are matched with the pattern of the token. For Example, int i, num, ans, choice

Eg:

```
int x = 5;
```

Lexeme	tokens
int	Keyword
x	identifier
=	operator
5	Constant
;	operator

Eg:

```
void swap (int a, int b)
{
    int k;
    k = a;
    a = b;
    b = k;
}
```

Lexeme	token
a	id
a	id
=	op
b	id
b	id
; k }	op id op

Lexeme	Tokens
void	Keyword
swap	identifier
(	operator
int	Key
a	id
,	op
int	Key

Lexeme	tokens
b	id
)	op
{	op
int	Key
k	id
k	id
=	op

Q GTU

W-22 . Q-2(c)

```
const h = 10;
if (a<h)
{
  a++;
  if (a<P) if (a==5)
  }          continue;
```

Lexeme	tokens
const	keyword
h	identifier
=	operator
10	constant
;	operator
if	keyword
(	operator
a	identifier
h	identifier
)	operator
{	operator
a	keyword
++	operator
+	operator
;	operator
if	keyword
(	op
a	identifier
=	op
=	op

```
    5
    }
    Continue
```

Q GTU
W-23, Q-1(b)

```
void change (int c, int d)
  {
     int m;
     m = c;
     c = d;
     d = m;
  }
```

Ans

Lexeme	token	Lexeme	token
void	Keyword	c	id
change	identifier	=	op
(	op	d	id
int	key	;	op
c	id	d	id
)	op	=	op
int	key	m	id
d	id	;	op
)	op	}	op
{	op		
int	key		
m	id		
;	op		
m	id		
=	op		
c	id		
;	op		

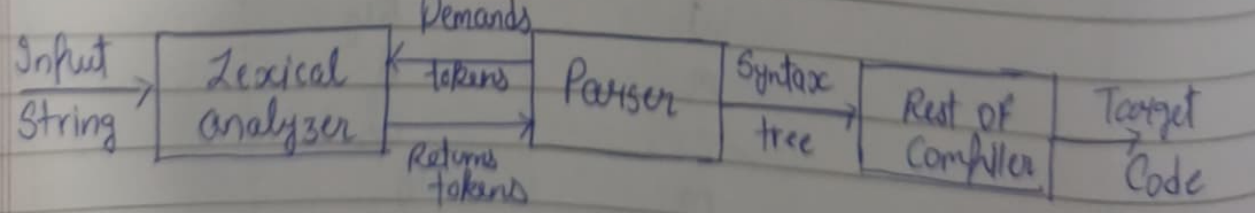
Q What is Lexical analysis? Which are the tasks performed by lexical analyzer

Ans Checke Ch-1

Tasks performed by lexical analyzer

1) It produces steam of tokens
2) It eliminates blank and comments
3) It generates symbol table which stores the information about identifiers, constants encountered in the input
4) It keeps track of line numbers
5) It reports the error encountered while generating the tokens

Lexical analyzer is the first phase of compiler
→ The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens
→ Each token is a single logical cohesive unit such as identifier, keywords, operators and punctuation marks
○ The role of lexical analyzer in the process of compilation as shown below



Page No.
Date

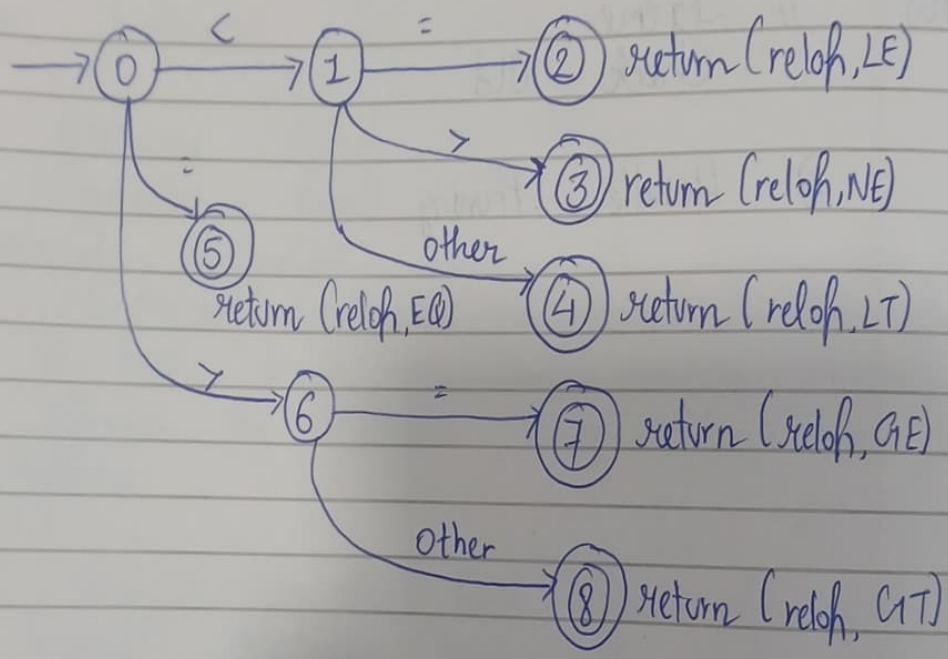
Q Define
1) Lexeme
2) Handle Pruning
3) Viable Prefixes

Ans 1) Lexeme
Check Old

2) Handle Pruning

Q Construct transition diagram for relational operators. Explain roles of assembler, loader, linker and preprocessor. [W-24, Q-2(c)]

Ans

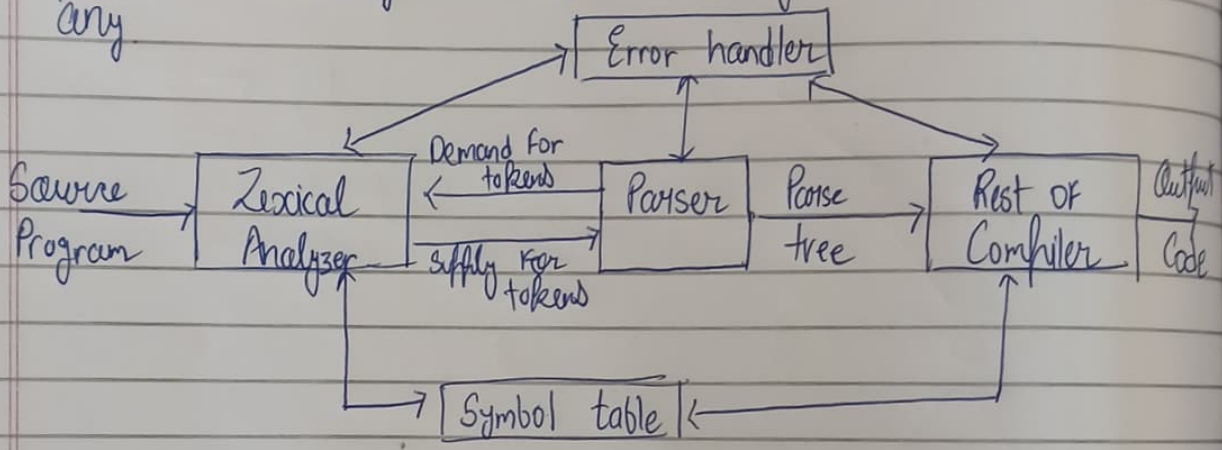


Q

Ch-3 Syntax Analysis

- Role of Parser

The Parser collects sufficient number of tokens and builds a parse tree. Thus by building it, parser smartly finds the syntactical errors if any.



→ Lexical Analyzer scans the input program and collects the tokens from it.

→ On other hand Parser builds a parser tree using these tokens

- Context Free Grammar

$G = (V, T, P, S)$

V = Set of non-terminals
T = " " Terminals
P = Set of Production Rules
S = Start Symbol

Non-terminal → Non-terminals
or Non-terminal → terminals

Derivation

1) Left Most Derivation
2) Right Most Derivation

Parse / Derivation tree

Root Node ⇒ Start Symbol
Internal / Parent Node ⇒ Non terminals
leaf nodes → terminals

Eg: $E \to E+E \mid E*E \mid (E) \mid id$
obtain Left most & right most derivation for string
id + id * id

Left Most Derivation

$E \to E+E$ ($\because E \to E+E$)
$\to id+E$ ($\because E \to id$)
$\to id+E*E$ ($\because E \to E*E$)
$\to id+id*E$ ($\because E \to id$)
$\to id+id*id$ ($\because E \to id$)

Right Most Derivation

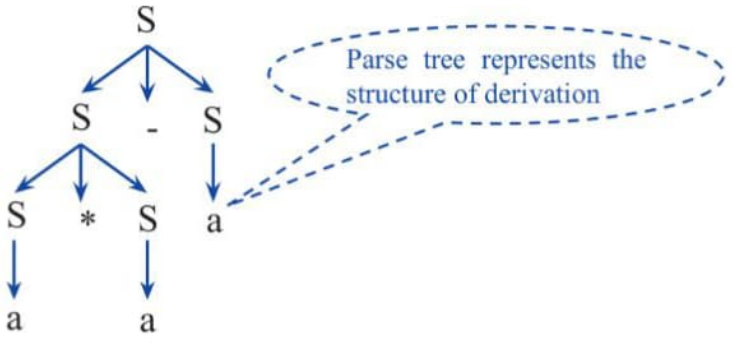
$E \to E+E$
$\to E+E*E$ ($\because E \to E*E$)
$\to E+E*id$
$\to E+id*id$
$\to id+id*id$

Leftmost derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Grammar: S→S+S | S-S | S*S | S/S | a Output string: a*a-a

S
→**S**-**S**
→**S***S-S
→**a***S-S
→a***a**-S
→a*a-**a**

Leftmost Derivation



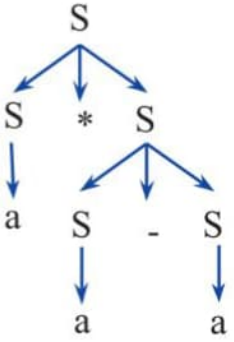
Parse tree

Rightmost derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- It is all called canonical derivation.
- Grammar: S→S+S | S-S | S*S | S/S | a Output string: a*a-a

S
→**S***S
→S***S-S**
→S***S-a**
→S***a**-a
→**a***a-a

Rightmost Derivation



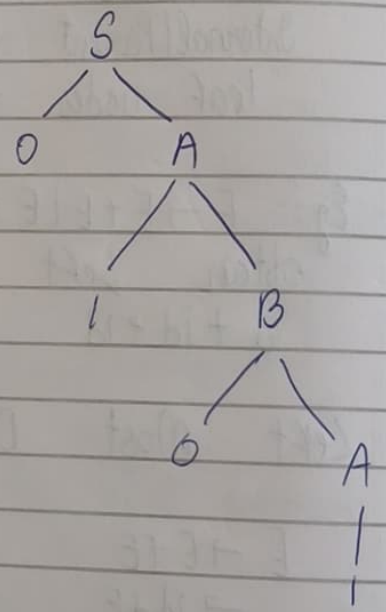
Parse Tree

Eg: $S \to 0A|0B|0|1$
$A \to 0S|1B|1$
$B \to 0A|1S$

Construct LMD & Parse tree

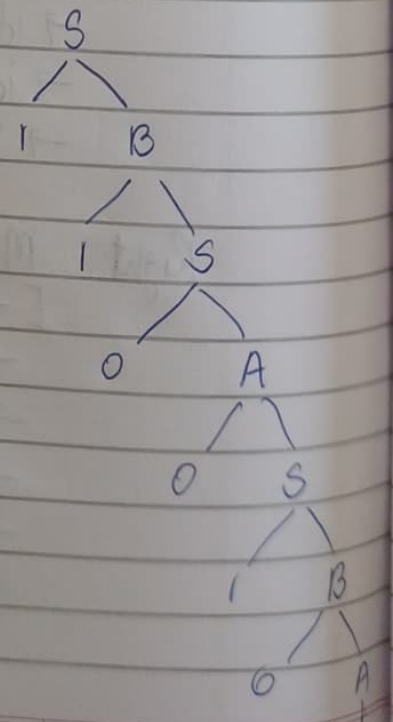
(i) 0101

$S \to 0\underline{A}$
$S \to 01\underline{B}$
$S \to 010\underline{A}$
$S \to 0101$



(ii) 1100101

$S \to 1\underline{B}$
$S \to 11\underline{S}$
$S \to 110\underline{A}$
$S \to 1100\underline{S}$
$S \to 11001\underline{B}$
$S \to 110010\underline{A}$
$S \to 1100101$



Ambiguous Grammar

A grammar is said to be ambiguous if it generates more than one parse tree for corresponding input string

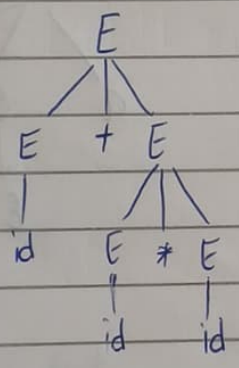
Eg: $E \to E+E \mid E*E \mid (E) \mid id$

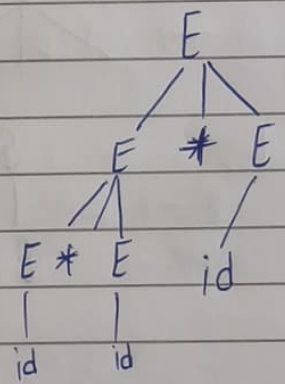
LMD

$E \to E+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

RMD

$E \to E+E$
$\to E+id*E$
$\to E+id*id$
$\to id+id*id$





Eg: Show that following grammar is ambiguous

$S \to aSbS$
$S \to bSaS$
$S \to \varepsilon$

String:

$S \to aSbS$
$S \to a\underline{b}S\underline{a}SbS$
$S \to ab\varepsilon a\underline{S}bS$
$S \to ab\varepsilon a\varepsilon bS$
$S \to ababE$
$S \to abab$

$S \to aSbS$
$S \to a\varepsilon bS$
$\to abaSbS$
$\to aba\varepsilon bS$
$\to ababE$
$\to abab$

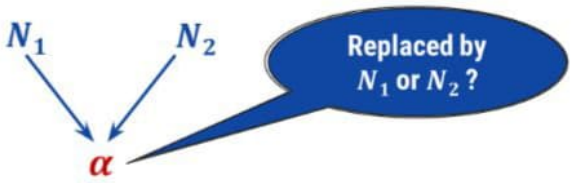
Ambiguity

- In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.
- Grammar:

$$N1 \rightarrow \alpha$$

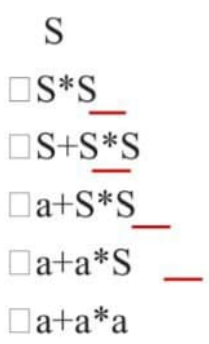
$$N2 \rightarrow \alpha$$

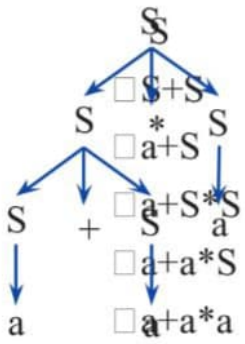
- α can be derived from either N1 or N2

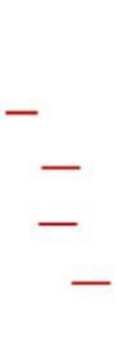


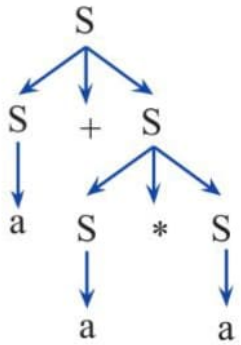
Ambiguous grammar

- Ambiguous grammar is one that produces more than one leftmost or more then one rightmost derivation for the same sentence.
- Grammar: S□S+S | S*S | (S) | a Output string: a+a*a

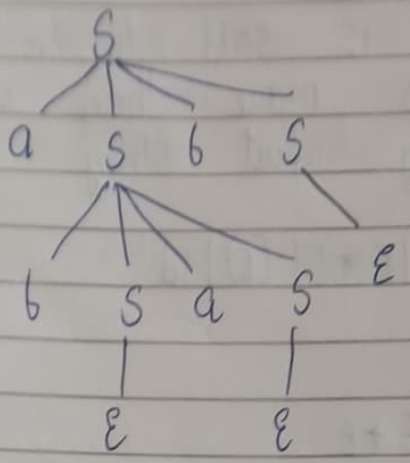


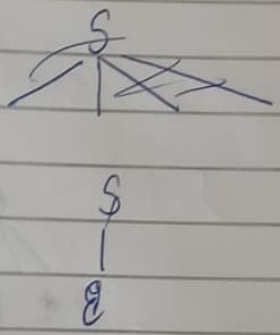


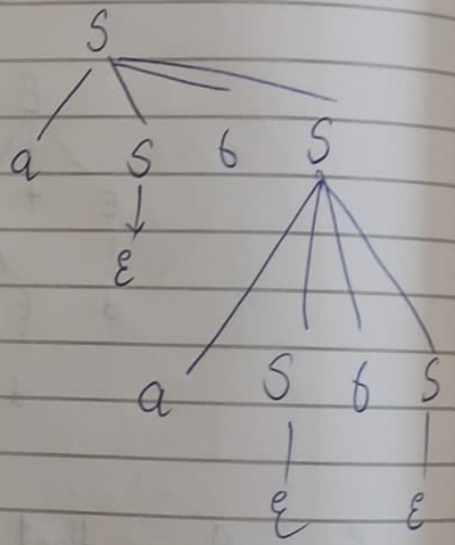




- Here, Two leftmost derivation for string a+a*a is possible hence, above grammar is ambiguous.

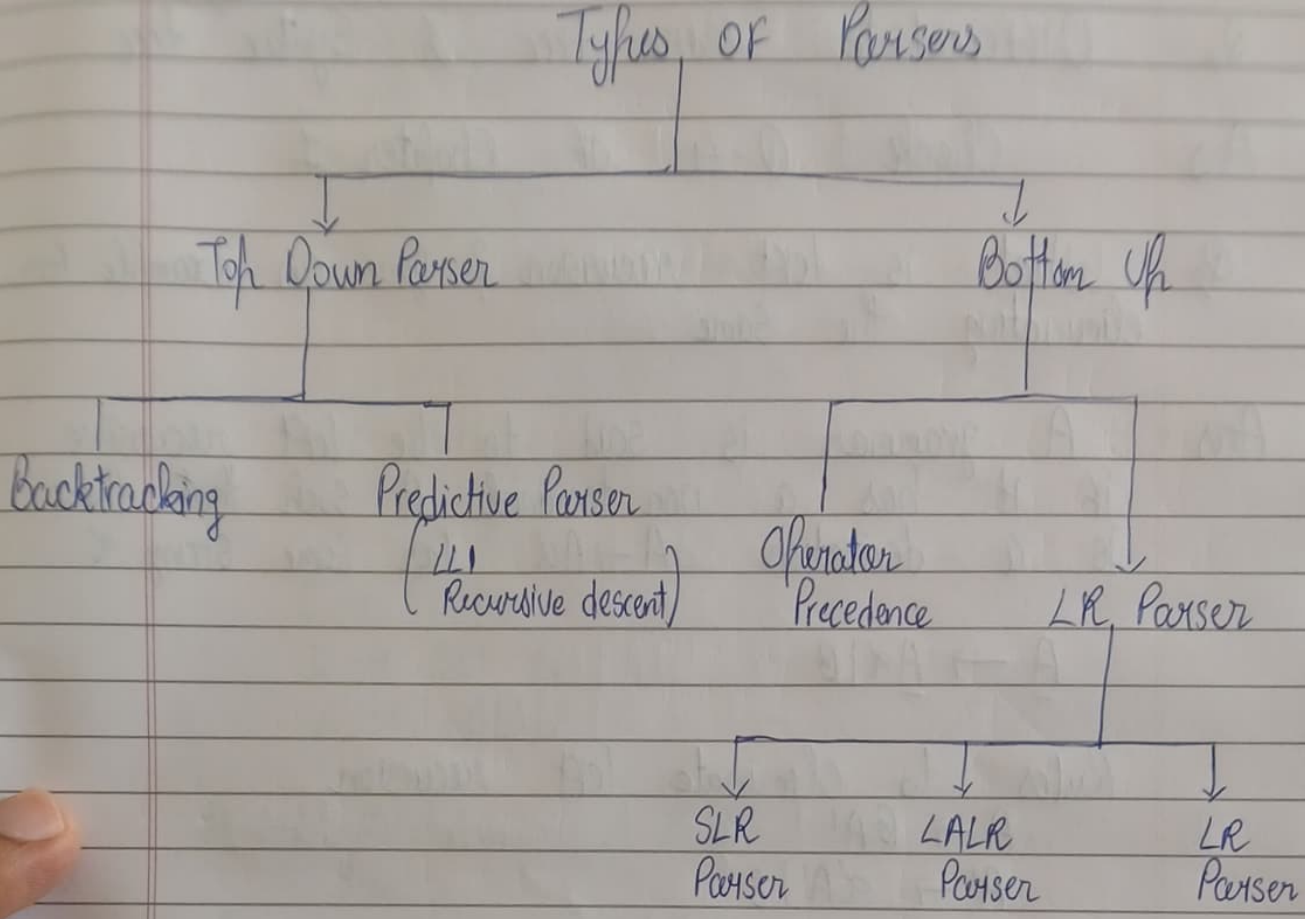






- Parsing
  It is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message, indicating that string is not a valid

Types of Parsers



Q. Differentiate Top-Down & Bottom Up Parsing

Top Down	Bottom-Up
1) Parse tree can be built from root to leaves	1) Parse tree is built from 2) leaves to root
2) Simple to Implement	Complex
3) Less efficient	3) More Efficient
4) Applicable to small class of languages	4) to broad class of languages
5) Eg: 1) Recursive descent 2) Predictive Parser	5) Eg: 1) Shift reduce 2) Operator Precedence 3) LR Parser

Q. Differentiate Parse tree & Syntax tree

Ans Check Q-4 of Chapter-1

Q. What is left recursion? Give an example for eliminating the same

Ans A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow Ad$ for some string α

$A \rightarrow A\alpha | \beta$

Rules to eliminate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \varepsilon$

Eg:

$E \rightarrow E+T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

$E \rightarrow E+T | T$

$\alpha \quad \beta$

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow T*F | F$

$\alpha \quad \beta$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

Eg:

$S \to \underset{A}{S}\,0\,\underset{\alpha}{S}\,1\,S \mid \underset{B}{0} \mid 1$

$S \to 0 \mid S'$
$S' \to 0S1SS' \mid \varepsilon$

Eg:

$S \to (L) \mid x$
$L \to \underset{A}{L},\underset{\alpha}{S} \mid \underset{B}{S}$

$S \to (L) \mid x$
$L \to SL'$
$L' \to ,SL' \mid \varepsilon$

$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \beta_1 \mid \beta_2 \ldots$

then

$A \to \beta_1 A' \mid \beta_2 A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

Eg: $expr \to \underset{A}{expr} + \underbrace{expr}_{\alpha_1} \mid \underset{A}{expr} * \underbrace{expr}_{\alpha_2} \mid \underset{B}{id}$

$expr \to id\ expr'$
$expr' \to +expr\ expr' \mid *expr\ expr' \mid \varepsilon$

Eg: $S \to \underset{A}{S}\underset{\alpha_1}{x} \mid \underset{A}{S}\underset{\alpha_2}{Sb} \mid \underset{\beta_1}{x}\underset{}{S} \mid \underset{\beta_2}{a}$

$S \to xSS' \mid aS'$
$S' \to xS' \mid SbS' \mid \varepsilon$

Eg: $S \to Aa|b$
$A \to Ac|Sd|f$

Indirect Left recursion

$S \to Aa|b$
$A \to Ac|Aad|bd|f$
$A\alpha_1 \; A\alpha_2 \; \beta_1 \; \beta_2$

$S \to Aa|b$
~~$A \to bdA'|$~~
~~$A \to Ac|Aa$~~
$A \to bdA'|fA'$
$A' \to cA'|adA'|\varepsilon$

Q. Left Factoring (Recursive Factoring)

$A \to \alpha\beta_1|\alpha\beta_2|\alpha\beta_3|..\; \gamma_1|\gamma_2..$

then we replace it with

$A \to \alpha A'|\gamma_1|\gamma_2$
$A' \to \beta_1|\beta_2|\beta_3..$

Eg: I

$S \to \underset{\alpha_1}{iEtS}|\underset{\alpha}{iEtSeS}|a$
$E \to b$ (β_1, β_2, γ_1)

~~$S \to iEtS'|a$~~
$S' \to S|SeS$

$S \to iEtSS'|a$
$S' \to \varepsilon|eS$

Eg:2

$A \to aAB \mid aA \mid a$

$B \to bB \mid b$

$\alpha \quad \beta_1 \quad \alpha$

$A \to aAB \mid aA \mid a$

$\alpha \beta \quad \alpha \quad \gamma_1$

~~$A \to aAA' \mid a$~~

~~$A' \to B \mid \varepsilon$~~

$A \to aAB \mid aA \mid a$

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

$A \to aA'$

$A' \to AB \mid A \mid \varepsilon$

$B \to bB'$

$B' \to B \mid \varepsilon$

Eg:3 $X \to X+X \mid X*X \mid D$ Y_1

$\alpha \quad \beta_1 \quad \alpha \quad \beta_2$

$D \to 1 \mid 2 \mid 3$

$X \to XX' \mid D$

$X' \to +X \mid *X$

$D \to 1 \mid 2 \mid 3$

Eg:4 $E \to T+E \mid T$

$T \to int \mid int * T \mid (E)$

$E \to TE'$

$E' \to +E \mid \varepsilon$

$T \to intT' \mid (E)$

$T' \to \varepsilon \mid *T$

- First & Follow
- Rules to compute First of Non-Terminal

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

2) If $A \to \varepsilon$ then First(A) = {ε}

3) if $A \to BC$ then
First(A) = First(B) if First(B) doesn't contain ε
if First(B) contains ε then
First(A) = First(B) ∪ First(C)

- Rules for finding follow

1) If 'S' then Follow(S) = {$}

2) If $A \to \alpha B\beta$ then Follow(B) = First(β)
if First(β) does not contain epsilon (ε)

3) if $A \to \alpha B$ then Follow(B) = Follow(A)

4) if $A \to \alpha B\beta$ where $\beta \to \varepsilon$
then Follow(B) = Follow(A)

Eg:

$E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow (E) | id$

First (E) = { (, id }
First (E') = { +, ε }
First (T) = { (, id }
First (T') = { *, ε }
First (F) = { (, id }

Follow (E) = { $,) }
Follow (E') = { $,) }
Follow (T) = { +, ε } { +, $,) } { +, (, id } { +, $,) }
Follow (T') = { +, ε } { +, $,) } { +, $,) }
Follow (F) = { *, ε } { *, +, { *, +, $,) }

Eg: $S \rightarrow ABCDE$
$A \rightarrow a | \varepsilon$
$B \rightarrow b | \varepsilon$
$C \rightarrow c$
$D \rightarrow d | \varepsilon$
$E \rightarrow e | \varepsilon$

$S \rightarrow \varepsilon BCDE$
$S \rightarrow \varepsilon BCDE$
$S \rightarrow CDE$

$S \rightarrow AB$

First
S = { a, b, c }
A = { a, ε }
B = { b, ε }
C = { c }
D = { d, ε }
E = { e, ε }

Follow
S = { $ }
A = { b, c }
B = { c }
C = { d, e, $ }
D = { e, $ }
E = { $ }

Eg:

$S \to Bb \mid Cd$
$B \to aB \mid \varepsilon$
$C \to cC \mid \varepsilon$

First
$S = \{a, b, c, d\}$
$B \to \{a, \varepsilon\}$
$C \to \{c, \varepsilon\}$

Follow
$S = \{\$\}$
$B = \{b\}$
$C = \{d\}$

Eg:

$S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \varepsilon$
$C \to h \mid \varepsilon$

$S \to ACB$
$S \to CbB$
$S \to Ba$
$A \to BC^{\varepsilon\varepsilon}$

First
$S = \{d, a, g, h, b\}$
$A = \{d, g, h, \varepsilon\}$
$B = \{g, \varepsilon\}$
$C = \{h, \varepsilon\}$

Follow
$S = \{\$\}$
$A = \{h, g, \varepsilon, \$\}$
$B = \{\$, a, h, g\}$
$C = \{\$, a, h,$

$C = \{g, \$, b, h\}$

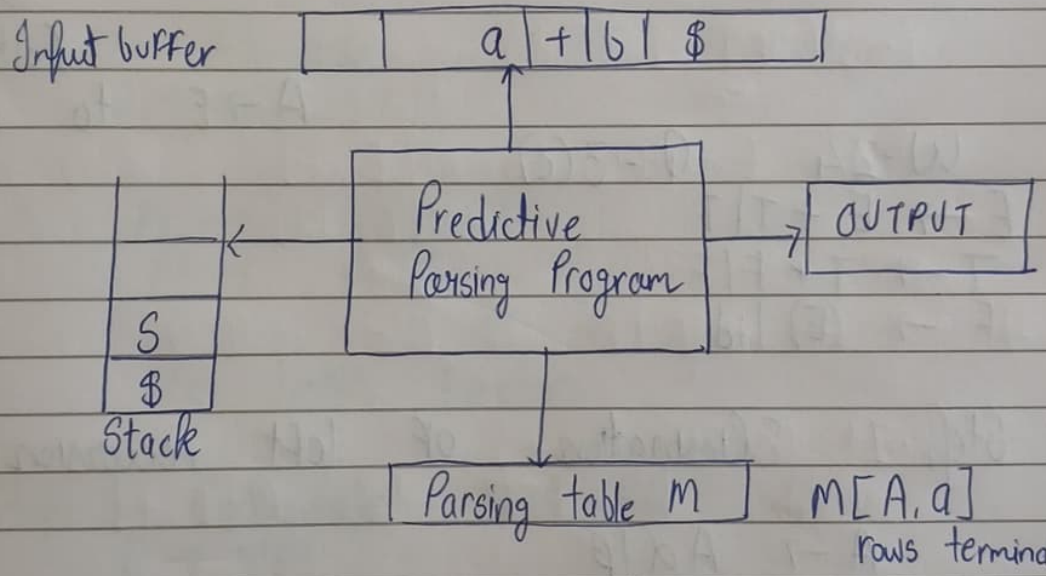
Eg: $S \to aABb$
$A \to c \mid \varepsilon$
$B \to d \mid \varepsilon$

First
$S = \{a\}$
$A \to \{c, \varepsilon\}$
$B = \{d, \varepsilon\}$

Follow
Follow (S) = $
Follow (A) = {d, b}
Follow (B) = {b}

LL(1) Parser or Predictive Parser or Non-recursive descent parser

1) First L indicates input is scanned from left to right
2) The second L means it uses leftmost derivation for input string
3) 1 means it uses only input symbol to predict the parsing process



Steps to construct LL(1) parser
1) Remove left recursion / Perform left factoring (if any)
2) Compute First & Follow of non terminals
3) Construct Predictive parsing table
4) Parse the input string using parsing table
OR
4) Check whether I/P string is accepted by parser or not

Construction of Parse table

$A \to \alpha$

1) $A \to \alpha$

First (α)

$\hookrightarrow a$

Add $A \to \alpha$ to $M[A, a]$

2) if First (α) contains ε or $A \to \varepsilon$

↓

Follow (A) = b

$A \to \varepsilon$ to $M[A, b]$

Q. W-24 Q-3(c)

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

Step: 1 Elimination of left recursion

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$\underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+T} \mid \underset{\beta}{T}$

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$\underset{A}{T} \to \underset{A}{T} \underset{\alpha}{*F} \mid \underset{\beta}{F}$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$ No left recursion

After eliminating left recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Step:2 Eliminate of left factoring

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \gamma_1 | \gamma_2$

$A \rightarrow \alpha A' | \gamma_1 | \gamma_2$

$A' \rightarrow \beta_1 | \beta_2$

No left Factors

Step:3 Calculate First & Follow

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

First	Follow
$E = \{(, id\}$	$E = \{\$,)\}$
$E' = \{+, \varepsilon\}$	$E' = \{\$,)\}$
$T = \{(, id\}$	$T = \{+, \varepsilon, \$,)\}$
$T' = \{*, \varepsilon\}$	$T' = \{+, \$,)\}$
$F = \{(, id\}$	$F = \{*, +, \$,)\}$

Step: 4
Constrain of Parsing table

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

1) $\underset{A}{E} \to \underset{\alpha}{TE'}$

First(TE') = { (, id }

Add $E \to TE'$ to M[E, (]
M[E, id]

2) ~~$\underset{A}{E'} \to \underset{\alpha}{+TE'} | \varepsilon$~~ 2) $E' \to +TE' | \varepsilon$

First (+TE') = {+}
Add $E' \to +TE'$ to M[E', +]

First (ε
$E' \to \varepsilon$
Follow (E') = { $,) }

Add $E' \to \varepsilon$ to M[E', $]
M[E',)]

3) $T \to \underset{\alpha}{ET'}$

First (FT') = { (, id }

Add $T \to FT'$ to $M[T, (]$
$M[E, id]$

4) $T' \to *FT' | \varepsilon$

$T' \to *FT'$
First $(*FT') = \{*\}$

Add $T' \to *FT'$ to $M[T', *]$

$T' \to \varepsilon$
Follow $(T') = \{+, \$,)\}$ to

Add to $M[T', +]$, $M[T', \$]$, $M[T',)]$

5) $F \to (E) | id$

First $((E)) = \{(\}$

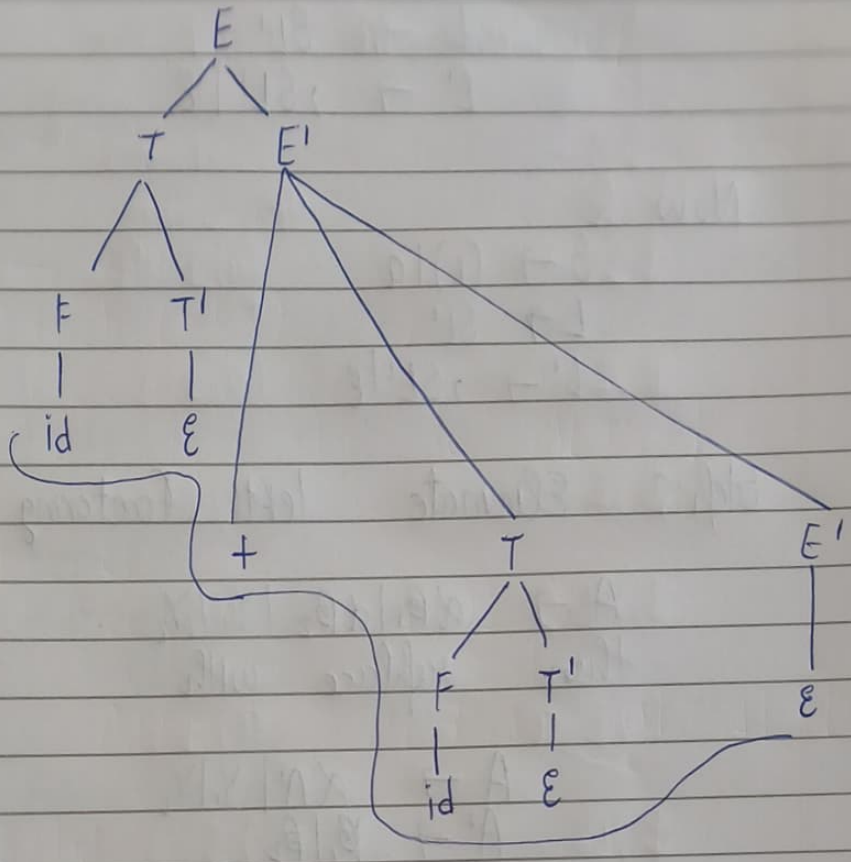
Add $F \to (E)$ to $M[F, (]$
$F \to id$
First $(id) = \{id\}$

Add $F \to id$ to $M[F, id]$

Step: 5

Let string be id + id $

Stack	Input String	Action
$E	id + id $	E → TE'
$E'T	id + id $	T → FT'
$E'T'F	id + id $	F → id
$E'T'id	id + id $	pop
	Same then perform pop	
$E'T'	+ id $	T' → ε
$E'	+ id $	E' → +TE'
$E'T+	+ id $	pop
$E'T	id $	T → FT'
$E'T'F	id $	F → id
$E'T'	$	pop
$E'	$	T' → ε
$E'	$	E' → ε
$	$	Accepted



id ε + id ε ε
⇒ id + id

Eg: 2

$S \to (L) \mid a$
$L \to L,S \mid S$

Step: I

Elimination of left recursion

$A \to \alpha A \alpha \mid \beta$

then

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

$S \to (L) \mid a$
$L \to L,S \mid S$
$A \quad A \; \alpha \quad \beta$

$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \varepsilon$

Now

$S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \varepsilon$

Step:2 Eliminate left factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma_1 \mid \gamma_2$
then replace with

$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2$
$A' \rightarrow \beta_1 \mid \beta_2$

No left factoring

Step:3 Find First & Follow

NT	First	Follow
S	{(, a}	{$, ,,)}
L	{(, a}	{)}
L'	{,, ε}	{)}

$S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \varepsilon$

Step:4 Construction of Parse table

	(	)	,	a	$
S	S → (L)			S → a	
L	L → SL'			L → SL'	
L'		L' → ε	L' → ,SL'		

1) S → (L) | a

S → (L)
First (S) = {(}
Add S → (L) to M[S, (]

Follow (a) = {a}

2) L → SL'
First (SL') = {(, a}

3) L' → ,SL' | ε

First (,SL') = {,}

L' → ε
Follow (L') = {)}

Let String be (a) $

Stack	i/p String	Action
$ S	(a) $ ↑	S → (L)
$) L (	(a) $	Pop
$) L	a) $	L → SL'
$) L' S	a) $	S → (L)
$) L' a	a) $	Pop
$) L'	) $	L' → ε
$)	) $	Pop
$	$	accepted

Example : 3
input string : "aabb"

S → aAB | bA | ε
A → aAB | ε
B → bB | ε

Step : 1 & Step : 2

No need bro

	First	Follow
S	{a, b, ε}	{$}
A	{a, ε}	{b, $}
B	{b, ε}	{$}

	a	b	$
S	S→aAB	S→bA	S→ε
A	A→aAb	A→ε	A→ε
B		B→bB	B→ε

1) S → aAB | bA | ε

S → aAB
First (aAB) = {a}

S → bA
First (bA) = {b}

S → ε
Follow (S) = {$}

2) A → aAb | ε

A → aAb
First (aAb) = {a}

A → ε
Follow (A) = {b, $}

3) B → bB | ε
B → bB
First (bB) = {b}

B → ε
Follow (B) = {$}

Stack	input string	action
$S	aabb$	S → aAB
$$BAa	aabb$	pop
$$BA	abb$	A → aAb
$$bAa	abb$	pop
$$bA	bb$	A → ε
$$b	bb$	pop
$$	b$	B → bB
$$b	b$	pop
$$	$	B → ε
$	$	accepted

Example: 4

S → iEtS | iEtSeS | a
E → b

Step: 1: No Left Recursion

Step: 2

$S \rightarrow iEtSS' \mid iEtSeS \mid a$

$\alpha \quad \beta_1 \quad \alpha \quad \beta_2 \quad \gamma_1$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

$E \rightarrow b$

	First	Follow
S	{i, a}	{e, $}
S'	{ε, e}	{e, $}
E	{b}	{t}

Step: 4

	i	a	e	b	$	t
S	$S \rightarrow iEtSS'$	$S \rightarrow a$				
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$		$S' \rightarrow \varepsilon$	
E						

It is Not LL(1) Grammar

1) $S \rightarrow iEtSS' \mid a$

a) $S \rightarrow iEtSS'$

First $(iEtSS') = i$

b) $S \rightarrow a$

2) $S' \rightarrow \varepsilon \mid eS$

follow $(S') = \{e, \$\}$

First $(eS) = \{e\}$

Q. S-25 GTU Q-2 (c)

S → aABb
A → c|ε
B → d|ε

Step: 1 Eliminate Left recursion

Not there

Step: 2 Eliminate Left Factoring

Not there

Step: 3 Find First & Follow

	First	Follow
S	{a}	{$}
A	{c, ε}	{d, b}
B	{d, ε}	{e}

Step: 4 Parsing table

	a	b	c	d	$
S	S → aABb				
A		A → ε	A → c	A → ε	
B				B → d	B → ε

Step: I

S → aABb

1) First(aABb) = a

2) A → c|ε

a) First(c) = c

b) Follow(A) = {d, $}

3) B → d|ε

a) First(d) = d

b) Follow(B) = {e}

Q/ Find out First & Follow set for all Non Terminals

W-23, OR Q-3(b) GTU

S → IAB | ε
A → IAC | OC
B → OS
C → I

NT	First	Follow
S	{I, ε}	{$}
A	{I, O}	{I, O}
B	{O}	{$}
C	{I}	{I, O}

Page No.
Date

Recursive Descent Parser

1) Non-Terminal
   ↳ Call

2) if input is terminal then compare terminal with corresponding input symbol
   if they are same input ++

3) If NT produces more than one production then all production rules should be written in corresponding function

4) No need to declare any main function or variable

Eg: E → iE'
E' → +iE'|ϵ

1) Define procedure for E

```
E()
{
    if (input == 'i')
        input ++;
    EPRIME();
}

EPRIME()
{
    if (input == '+')
    {
        input ++;
```

```
        if (input == 'i')
        {
            input++;
            EPRIME();
        }
    }
    else
        return;
}
```

2) $E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

```
E()
{
    T();
    EPRIME();
}

EPRIME()
{
    if (input == '+')
    {
        input++;
        T();
        EPRIME();
    }
    else
        return;
}
```

```
T()
{
    F();
    TPRIME();
}

TPRIME()
{
    IF (input == '*')
    {
input++;    ~~input++;~~ F();
        TPRIME();
    }
    else
        return;
```

```
F()
{
    if (input == '(')
    {
        input++;
        E();
        if (input == ')')
            input++;
    }
    else if (input == 'id')
        input++;
    }
```

Q// Implement the following grammar using Descent Parser

S → Aa | bAc | bBa
A → d
B → d

```
Procedure S()
    {                          S → bAC | bBa
        IF (lookahead = 'b')
        {
            B();
            if (lookahead = 'c')
                match('c')
            else if (lookahead = 'a')
                match('a')
            else if (lookahead = '$')
            {
                declare SUCCESS,
            else
```

```
{          // S -> Aa
    A();
    if (lookahead == "a")
        match('a');
    }
}
```

```
Procedure (A)          A -> d
{
    if (lookahead = 'd')
    match ('d')
    else
        error();
}
```

```
Procedure (B)
{
    if (lookahead = 'd')
    match ('d')
    else
        error();
}
```

Q. Explain Recursive descent parsing technique with suitable example

2)	In ambiguous grammar there are less number of non-terminals.	In unambiguous grammar there are more number of non-terminals.
3)	The length of generated parse tree is less.	The length of generated parse tree is large.
4)	For example - $E \rightarrow E + E \mid E * E \mid id$.	For example - $E \rightarrow E+T$, $E \rightarrow T$, $T \rightarrow T*F$, $T \rightarrow F$, $F \rightarrow id$

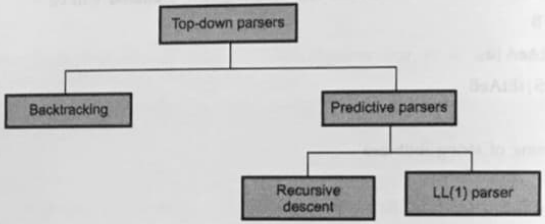


Fig. 3.3.6 Types top - down parsers

There are two types by which the top-down parsing can be performed.

1. Backtracking
2. Predictive parsing

A backtracking parser will try different production rules to find the match for the input string by backtracking each time. The backtracking is powerful than predictive parsing. But this technique is slower and it requires exponential time in general. Hence backtracking is not preferred for practical compilers.

As the name suggests the predictive parser tries to predict the next construction using one or more lookahead symbols from input string. There are two types of predictive parsers :

1. Recursive descent
2. LL(1) parser.

Let us discuss these types along with some examples.

3.3.2 Recursive Descent Parser

A parser that uses collection of recursive procedures for parsing the given input string is called **Recursive Descent (RD) Parser**. In this type of parser the CFG is used to

build the recursive routines. The R.H.S. of the production rule is directly converted to a program. For each non-terminal a separate procedure is written and body of the procedure (code) is R.H.S. of the corresponding non-terminal.

Basic steps for construction of RD parser

The R.H.S. of the rule is directly converted into program code symbol by symbol.

1. If the input symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Let us take one example to understand the construction of RD parser. Consider the grammar having start symbol E.

$E \rightarrow num\ T$

$T \rightarrow *\ num\ T \mid \varepsilon$

```
procedure E
{
    if lookahead = num then
    {
        match(num) ;
        T;                    /* call to procedure T */
    }
    else
        error;
    if lookahead = $
    {
        declare success;   /* Return on success */
    }
    else
        error;
}/*end of procedure E*/
procedure T
{
    if lookahead = '*'
    {
        match('*');
        if lookahead ='num'
        {
            match(num);
```

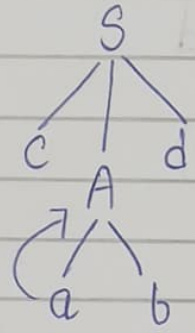
Backtracking

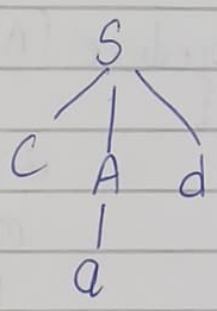
Eg 1:

$S \rightarrow cAd$

$A \rightarrow ab|a$

"cad"





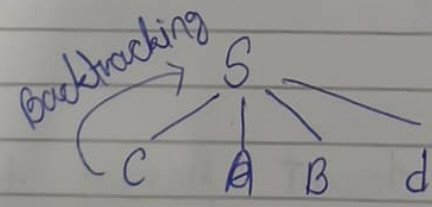
Eg: 2

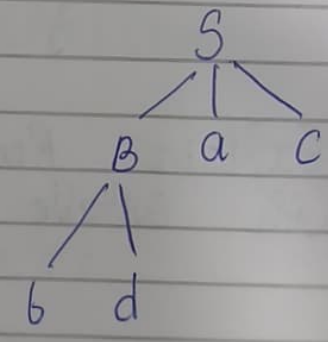
$S \rightarrow cABd \mid Bac$

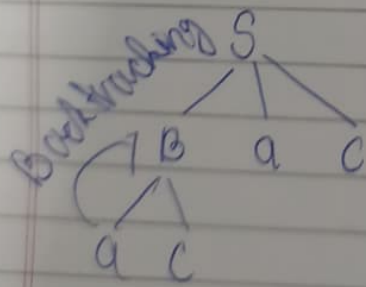
$A \rightarrow Bcd \mid aab$

$B \rightarrow ac|dA|bd$

"bdac"







Start Symbol (ROOT) Node

Bottom-Up Parsing

Bottom (leaves) → Input String (Leaves) Node

Eg

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

input string "abbcde"

abbcde (A → b) Handle

aAbcde (Abc → A) Handle

aAde (d → B) Handle

aABe (S → aABe)

S

Process of reducing input string to the starting symbol of the grammar is called Bottom-Up Parsing

S
S → aABe
→ aAbe
→ aAbcde
→ abbcde

Eg

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

id * id

id * id	(F → id)
F * id	(T → F)
T * id	(F → id)
T * F	(T → T*F)
T	(E → T)
E	

Handle & Handle Pruning [GTU]

1) Handle
→ A "handle" of a string is a substring of the string that matches the right side of a production and whose reduction of the non-terminal of production is one step along the reverse of RightMost Derivation

2) Handle Pruning
The process of discovering a handle and reducing it to appropriate left hand side non-terminal is called handle pruning

S → aABe
A → Abc | b
B → d

abbcde

Handles during Parsing of abbcde

Right Selected From	Handle	Reducing Production
abbcde	b	A → b
aAbcde	Abc	A → Abc
aAde	d	B → d
aABe	aABe	S → aABe
S		

- Shift Reduce Parser (SRP)
- → Stack, input buffer

Actions of SRP

1) Shift
2) Reduce
3) Accept
4) Error

Eg:

E → E+T | T
T → T*F | F
F → (E) | id

Stack

E → E+T | T
T → T*F | F
F → (E) | id

id * id

Stack	Input Buffer	Action
$	id * id $	Shift
$ id	* id $	Reduce by F→id
$ F	* id $	Reduce T→F
$ T	* id $	Shift
$ E * id		
$ T *	id $	Shift
$ T * id	$	reduce F→id
$ T * F	$	reduce T→T*F
$ T	$	Reduce T→E
$ E	$	Accepted

1) Shift reduce conflict
2) Shift reduce - reduce conflict

Eg 2 S → (L) | a
L → L, S | S

(a , (a,a))

Stack	Input Buffer	Action
$	(a,(a,a)) $	Shift
$ (	a,(a,a)) $	Shift
$ (a	,(a,a)) $	Reduce S→a
$ (S	,(a,a)) $	Reduce L→S
$ (L	,(a,a)) $	Shift
$ (L,	(a,a)) $	Shift

$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce S→a
$(L,(S	,a))$	Reduce L→S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce S→a
$(L,(L,S	))$	L→L,S
$(L,(L	))$	Shift
$(L,(L)	)$	S→(L)
$(L,S	)$	L→L,S
$(L	)$	Shift
$(L)	$	reduce S→(L)
$S	$	

0 Operator Precedence Parser

(OPG) Operator Grammar: To perform To define mathematical operations with some restrictions

1) 2 variables can't be adjacent on right side
2) No Epsilon (ε) on right side of production

Eg: $E \to E+E \mid E*E \mid id$

Eg: $E \to EAE \mid id$ } Non OPG
$A \to + \mid *$

$E \to E+E \mid E*E \mid id$ } It's OPG
$A \to + \mid *$

Parse the input string id+id*id

Step 1: Check operator gramcer or not

Step 2: Operator Precedence Relation table

Step 3: Parse the input string

Step 4: Generate Parse tree

E → E+E | E*E | id id+id*id

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	Accept

id, a, b → high
$ → lower
+ > +
+ < *
* > *

Operator	Precedence	Associative
↑	1	right
*, /	2	left
+, -	3	left

Stack input String Output Ch

Stack	Relation	Input String	Operation
$	<	id + id * id $	Shift id
$id	>	+ id * id $	reduce E → id
$E	<	+ id * id $	Shift +
$E+	<	id * id $	Shift id
$E+id	>	* id $	reduce E → id
$E+E	<	* id $	Shift *
$E+E*	<	id $	Shift id
$E+E*id	>	$	reduce E → id
$E+E*E	>	$	reduce E → E*E
$E+E	>	$	reduce E → E+E
$E		$	Accept

Eg: 2

a+b*c*d

E → E + T | T

T → T * F | F

F → a | b | c | d

	+	*	a	b	c	d	$
+	>	<	<	<	<	<	>
*	>	>	<	<	<	<	>
a	>	>	-	-	-	-	>
b	>	>	-	-	-	-	>
c	>	>	-	-	-	-	>
d	>	>	-	-	-	-	>
$	<	<	<	<	<	<	Accept

Stack	Relation	Input String	Operation
$	<	a+b*c*d$	Shift a
$a	>	+b*c*d$	reduce F→a
$F	<	+b*c*d$	Shift +
$F+	<	b*c*d$	Shift b
$F+b	>	*c*d$	reduce F→b
$F+F	<	*c*d$	Shift *
$F+F*	<	c*d$	Shift c
$F+F*c	>	*d$	reduce F→c
$F+F*F	>	*d$	reduce T→F
$F+F*T	>	*d$	reduce T→T*F
$F+T	<	*d$	Shift *
$F+T*	<	d$	Shift d
$F+T*d	>	$	reduce F→d
$F+T*F	>	$	reduce T→T*F
$F+T	>	$	Not accepted

Explain operator grammar. Generate precedence table

E → EAE|id

A → +|*

E → E+E|E*E|id

A → +|*

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	Accept

id + id * id

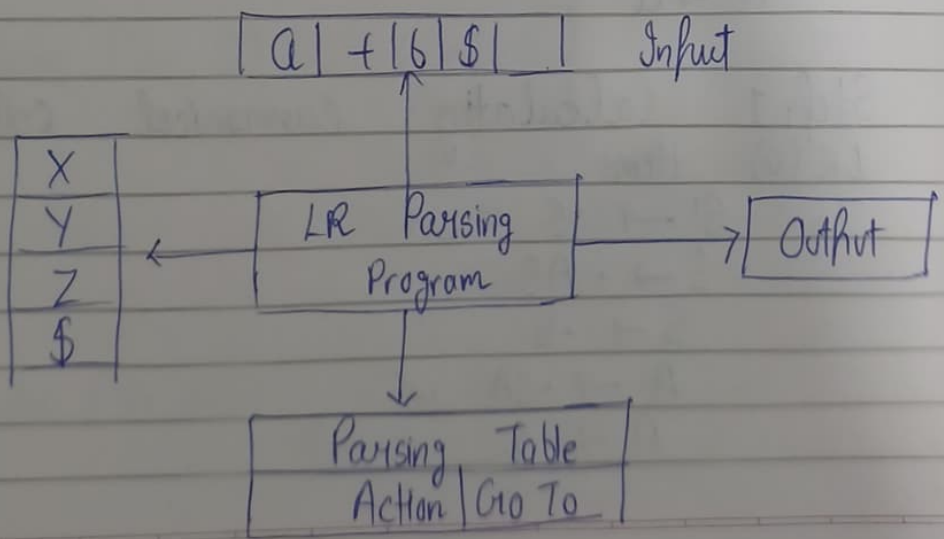
Stack	Relation	input string	Operation
$	id ~~+~~ id . <	id + id * id $	Shift id
$ id	>	+ id * id $	reduce E → id
$ E	<	+ id * id $	Shift +
$ E +	<	id * id $	shift id
$ E + id	>	* id $	reduce id E → id
$ E + E	<	* id $	Shift *
$ E + E *	<	id $	Shift id
$ E + E * id	>	$	store E → id
$ E + E * E	>	$	reduce E → E * E
$ E + E	>	$	reduce E → E + E
$ E		$	Accept.

Q/ Explain LR(1) Parser with suitable example

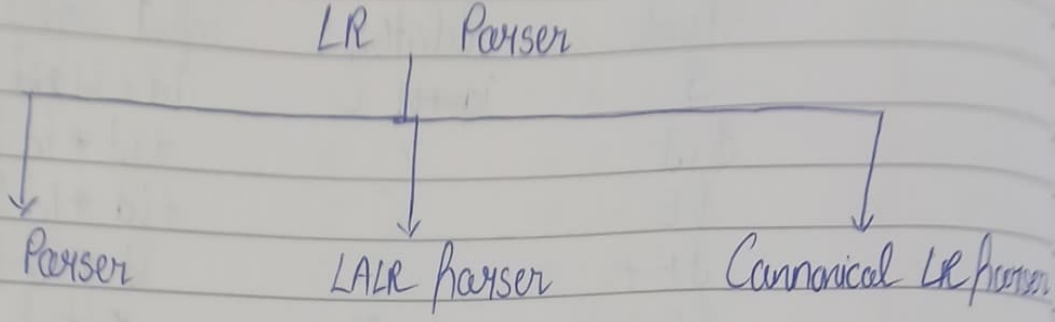
LR(1) means SLR for eg.

Ans Bottom - Up Parser

- L stands for left to right scanning
- R stands for rightmost derivation in reverse
- k is number of input symbols. when k is omitted k is assumed to be 1



Types of LR Parser



SLR Parser

S → ASlb
A → SA|a
Construct SLR parse table for grammar.
Show actions for string "abab".

Augmented Grammar

S' → S
S → AS
S → b
A → SA
A → a

Step I: Calculating cannonical collection of LR(0) items
I_0: S' → ·S ✓
S → ·AS
S → ·b
A → ·SA ✓
A → ·a

goto (I_0, S) :- $S' \to S.$
$A \to S.A$

I_1 :- $A \to .S$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_0, A) :- $S \to A.S$

I_2 $S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_0, b) :- I_3 $S \to b.$

goto (I_0, a) :- I_4 $A \to a.$

goto (I_1, A) :- $A \to SA.$
$S \to A.S$

I_5 $S \to b.$
$S \to .AS$
$A \to .SA$
$A \to .a$

goto (I_1, S) : $A \to S.$

I_6 $A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_1, a) :-
$A \to a.$ I_4

goto (I_1, b):

I_3

goto (I_2, A):

I_7

$S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_2, A):

I_2

$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_2, a):

I_4

goto (I_2, b): I_3

goto (I_5, S):

I_7

$S \to AS.$
$A \to S.A$
$A \to .a$
$A \to .SA$
$S \to .AS$
$S \to .b$

goto (I5, A):-

$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to a$
$A \to .SA$

I_2 state che

goto (I5, a) :- $A \to a.$ } I_4

goto (I5, b) :- $B \to b.$ } I_3

goto (I6, A) :- $A \to SA.$
$S \to A.S$ } I_5

goto (I6, S) :- $A \to S.A$
$A \to .a$
$A \to S.A$
$S \to .AS$
$S \to .b$ } I_6

goto (I6, a) :- I_4

goto (I6, b) :- I_3

goto (I7, A) :- $A \to SA.$
$B \to A.S$
$S \to .b$
$A \to .a$
$A \to .SA$
$S \to .AS$ } I_5

goto (I_1, S):

$A \to S.A$ I_6

goto (I_1, a):

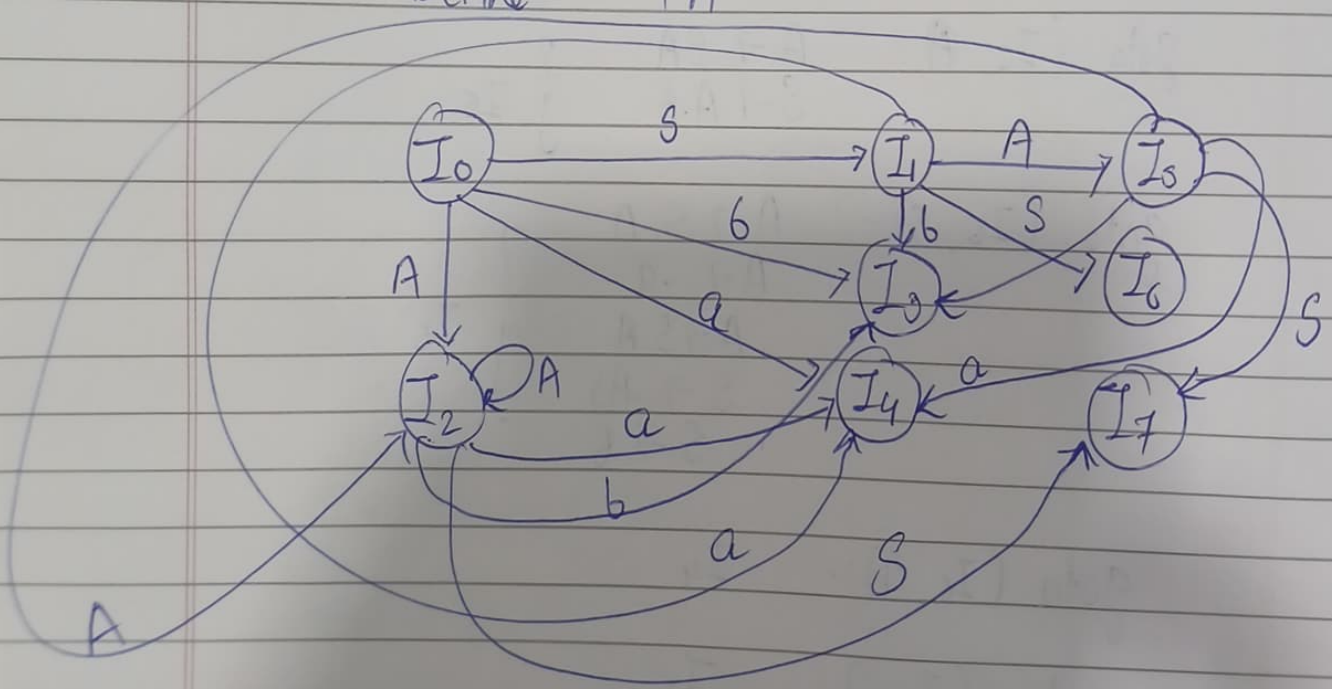
$A \to a.$ I_4

goto (I_1, b):

$B \to b.$ I_3

Step 2:

Define FA



Step 3

Construction of SLR Parse statement

$S \to AS \mid b$

$A \to SA \mid a$

First(S) = { b, a }

First (A) = { a, b }

Follow(S) = { $, a, b }

Follow(A) = { a, b }

Parse Table

	Action			goto	
	a	b	$	S	A
0	S4	S3		1	2
1	S4	S3	Accepted	6	5
2	S4	S3		7	2
3	R_2	R_2	R_2		
4	R_4	R_4			
5	S4/R_3	S3/R_3		7	2
6	S4	S3		6	5
7	S4/R_1	S3/R_1	R_1	6	5

$I_1 : S' \to S.$

$I_3 : S \to b.$ Final item

I_0 : 0 $S' \to .S$ 0

1 $S \to .AS$ 1

2 $S \to .b$ 2

3 $A \to .SA$ 3

4 $A \to .a$ 4

I_3 : $S \to b.$

r_2

Follow (S) = {$, a, b}

I_4 : $A \to a$

r_4

Follow (A) = {a, b}

I_5 : $A \to SA.$

r_3

Follow (A) = {a, b}

I_7 : ~~$A \to SA.$~~ $A \to AS.$

~~r_5~~ r_1

$, a, b

Stack	Input buffer	Action
$0	abab$	shift 4
$0a4	bab$	reduce A → a
$0A2	bab$	shift 3
$0A2b3	ab$	reduce S → ~~Ab~~ b
$0A257	ab$	reduce S → AS
$0S1	ab$	shift 4
$0S1a4	b$	reduce A → a
$0S1A5	b$	reduce A → SA

$0A2	b$	Shift 3
$0A2b3	$	reduce S→b
$0A2S7	$	red. S→AS
$ 0S1	$	Accepted

S → L=R|R
L → *R|id
R → L

Augmented grammar

S' → S	0
S → L=R	1
S → R	2
L → *R	3
L → id	4
R → L	5

Canonical LR(0) items

0 I_0: S' → .S
1 S → .L=R
2 S → .R
3 L → .*R
4 L → .id
5 R → .L

goto (I_0, S) :- S' → S.
I_1 : ~~S → L.=R~~

goto (I_0, L) :- S → L.=R
I_2 R → L.

goto (I_0, R) :

I_3 : $S \to R.$

goto $(I_0, *)$: $L \to *.R$

I_4 $R \to .L$

$L \to .*R$

~~goto $(I_0, =)$~~ : $L \to .id$

goto (I_0, id) : $L \to id.$

I_5

goto $(I_2, =)$: $S \to L=.R$

$R \to .L$

I_6 $L \to .*R$

$L \to .id$

goto (I_4, R) = $L \to *R.$

I_7

goto (I_4, L) = $R \to L.$

I_8

goto $(I_4, *)$: $L \to *.R$

$R \to .L$

I_4 $L \to .*R$

$L \to id$

goto (I_4, id) : $L \to id.$

I_5

goto (I_6, R) :

I_9 $S \to L : R.$

goto (I_6, L) :

I_8 $R \to L.$

goto $(I_6, *)$:

I_4 $L \to *.R$

$R \to .L$

$L \to .*R$

$L \to .id$

goto (I_6, id) :

I_5 $L \to id.$

SLR Parse tree

	Action					goto	
	*	id	$	=	S	L	R
0	S4	S5			I	2	3
1			Accept				
2			r5	S6/r5			
3			r_2				
4	S4	S5				8	7
5			r_4	r_4			
6	S4	S5				8	9
7			r_3	r_3			
8			r_5	r_5			
9			r_1				

I_1 : $S' \to S.$ Final term

I_2 : $R \to L.$ Final item

$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Follow(S) = {$}
Follow(L) = {=, $}
Follow(R) = {$, =}

$I_2 : R \to L.$

$\xrightarrow{I_5}$

Follow(R) = {$, =}

$I_3 : S \to R.$

$\xrightarrow{I_2}$

Follow(S) = {$}

$I_5 : L \to id.$

$\xrightarrow{I_4}$

Follow(L) = {$, =}

$I_7 : L \to *R.$

$\xrightarrow{I_3}$

$I_1 : R \to L.$

Folow = {$, =}

$\xrightarrow{I_5}$

$I_9 : S \to L = R.$

$\xrightarrow{I_1}$

Follow(S) = {$}

Stack	ip buffer	Action
$0	*id=id$	Shift 4
$0*4	id=id$	Shift 5
$0*4id5	=id$	reduce L→id
$0*4L8	=id$	Reduce R→L
$0*4R7	=id$	Reduce L→*R
$0L2	=id$	Shift 6 ~~Reduce R→L~~
$0L2=6	id$	Shift 5
$0L2=6id5	$	R4 L→id
$0L2=6L8	$	R→L
$0L2=6R9	$	S→L=R
$0S1	$	Accept

Q. W-22 OR Q-3(c)

S → (L) | a
L → L,S | S

Augmented grammar
0 S' → S
1 S → (L)
2 S → a
3 L → L,S
4 L → S

Follow(S) = { $,), , }

Follow(L) = {), , }

Cannonical LR(0) items
I0: S' → .S
S → .(L)
S → .a
L → .L,S
L → .S

goto (I_0, S) : $S' \to S.$ Fin
I_1 $L \to S.$ Fi

goto (I_0, L) : $L \to L., S$
I_2

goto (I_0, a) :
I_3 $S \to a.$ Fi

goto $(I_0, ()$
$S \to (.L)$
I_4 $L \to .L, S$
$L \to .S$
$S \to .(L)$

goto $(I_2, ,)$
$L \to L, .S$
I_5 $S \to .a$
$S \to .(L)$

goto (I_4, S) : $L \to S.$
I_6 Fi

goto (I_4, L) : $S \to (L.)$
I_7 $L \to L., S$

goto $(I_4, ()$: $S \to (.L)$
I_4

goto (I_7, L): S → L.

goto (I_5, S): L → L, S. I_9

goto (I_5, a): S → a. I_3

goto (I_5, (): S → (.L) I_8

Parse Table

	Action					goto	
	(	)	,	$	a	S	L
I_0	S4				S3	I_0	I_2
I_1		R_4	R_4	Accept			
I_2		S5					-
I_3		R_2	R_2	R_2			
I_4		S8				I_6	I_7
I_5	S8				S3	I_9	
I_6		R_4	R_4				
I_7							
I_8							
I_9		R_5	R_3				

I_1: S' → S.
L → S. R_4
Follow (L): {), ,}

I_3: S → a. R_2 {$,), ,}

I_6: $L \to S.$
Follow(L): { , ,) $ }
r_4

I_9: $L \to L, S$
r_3

Follow(L) = { , ,) }

GTU S-25 Q-3(c)

$S \to AaAb \mid BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

Follow(S) = {$}
Follow(A) = {a, b}
Follow(B) = {a, b}

I_0:

0 $S' \to S$
1 $S \to AaAb$
2 $S \to BbBa$
3 $A \to \varepsilon$
4 $B \to \varepsilon$

I_0:

$S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to \varepsilon .$
$B \to \varepsilon .$

goto (I_0, S) : $S' \to S.$ Fi
I_1

goto (I_0, A) : $S \to A.aAb$
I_2

goto (I_0, B) : $S \to B.bBa$
I_3

goto (I_2, a) : $S \to Aa.Ab$
I_4 $A \to .\cancel{AaAb}$ Fi

goto (I_3, b) : $S \to Bb.Ba$
I_5 $B \to .$ Fi

goto (I_4, A) : $S \to AaA.b$
I_6

goto (I_5, B) : $S \to BbB.a$
I_7

goto (I_6, b) : $S \to AaAb.$ Fi
I_8

goto (I_7, a) : $S \to BbBa.$ Fi
I_9

I_1: $S' \to S.$

I_8: $S \to AaAb.$
Follow(S) = {

State	Action			goto		
	a	b	$	S	A	B
0	r3/r4	r3/r4		1	2	3
1						
2	S4					
3		S5				
4	r3	r3			6	
5	r4	r4 r4				7
6		S8/r4				
7	S9					
8						
9						

I_4 : $A \to .$ $\{a, b\}$ r3, r4

I_5 : $B \to .$

ok got it

Go

Page No.
Date

Q Construct CLR Parser for following grammar

In SLR LR(0)
In CLR LR(1) = LR(0) + lookahead

S → CC
C → cC|d

Eg: S → CC
C → cC|d

Ans S → CC
C → cC
C → d

Step: 1 Augmented grammar

0 S' → S
1 S → CC
2 C → cC
3 C → d

Step: 2 Calculating LR(1) items

I_0 : 0 S' → .S , $
1 S → .CC , $
2 C → .cC, c|d
3 C → .d, c|d

goto (I_0, S): S' → S. , $
I_1

goto (I_0, C): S → C.C, $
I_2 C → .cC, $
C → .d, $

Page No.
Date

0 Lookahead Calculation

$S' \to S, \$$

$S' \to .S, \varepsilon \$$

$S \to$ First $(\varepsilon, \$)$

$(\$)$

$S \to .CC, \$$

First $(CC, \$)$

First $(C) = \{c, d\}$

~~goto (~~

$S \to C.C, \$$

First $(\$) = \$$

goto (I_0, C): I_3

$C \to C.C, c/d$

$C \to .CC, c/d$

$C \to .d, c/d$

goto (I_0, d): I_4

$C \to d., c/d$ Fi

goto (I_2, C): I_5

$S \to CC., \$$ Fi

goto (I_2, c): I_6

$C \to C.C, \$$

$C \to .CC, \$$

$C \to .d, \$$

goto (I_2, d):

I_7 $C \to d\cdot, \$$

goto (I_2, C):

$C \to cC\cdot, c/d$ H

I_8 ~~$C \to \cdot cC$~~

goto (I_3, c):

$C \to c\cdot C, c/d$

$C \to \cdot cC, c/d$

I_3 $C \to \cdot d, c/d$

goto (I_3, d)

$C \to d\cdot, c/d$

I_4

goto (I_6, C):

$C \to cC\cdot, \$$ Hi

I_9

goto (I_6, c):

$C \to c\cdot C, \$$

$C \to \cdot cC, \$$

I_6 $C \to \cdot d, \$$

goto (I_6, d):

$C \to d\cdot, \$$ I_7 Hi

Sujit

Page No.
Date

Construction of CLR Parse table

State	Action			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Accepted		
2	S6	S7			5
3	S3	S4			8
4	R_3	R_3			
5			R_1		
6	S6	S7			9
7			R_2		
8	R_2	R_2			
9			R_2		

I_1: $S' \to S.,\ \$$ FA

~~$I_{[illegible]}$: $CC. \to$~~

I_4: $C \to d.,\ c|d$ FA

I_5: $S \to CC.,\ \$$ FA

$I_{[illegible]}$: $C \to d.,\ \$$ FA

I_1: $C \to cC.,\ c|d$ FA

I_9: $C \to CC.,\ \$$ FA

Stack	Input String	Action
$0	dd$	Shift 4
$0d4	d$	reduce C→d
$0C2	d$	Shift 7
$0C2d7	$	reduce C→d
$0C2C5	$	S→CC
$0S1	$	Accepted

Q. Construct LALR Parser

S → L=R | R
L → *R | id
R → L

Id ≠ id

LALR is compressed version of CLR parser eg

L → *.R, =/$ & L → *.R, $
S → id., =/$ & S → id., $

same

different lookahead symbol so they can be merged

that's the concept of LALR

Augmented Grammar

S' → S
S → L=R
S → R
L → *R
L → id
R → L

LR(1) items

S' → .S, $
S → .L=R, $
S → .R, $
L → .*R, $/= /$
L → .id, =/$
R → .L, $

First (=, $)

goto (I_0, S) :

I_1 :- $S' \to S.,\ \$$ Fi I_1

goto (I_0, L)

I_2 :- $S \to L.=R,\ \$$
$R \to L.,\ \$$ Fi I_2

goto (I_0, R)

I_3 :- $S \to R.,\ \$$ Fi I_3

goto $(I_0, *)$:-

I_4 : $L \to *.R,\ =/\$$
$R \to .L,\ =/\$$
$L \to .*R,\ =/\$$
$L \to .id,\ =/\$$

goto (I_0, id) :-

I_5 $L \to id.,\ =/\$$ Fi I_5

goto $(I_2, =)$:

I_6 :- $S \to L=.R,\ \$$
~~$R \to .*R,$~~
$R \to .L,\ \$$
$L \to .*R,\ \$$
goto(I_4, $L \to .id,\ \$$

goto (I_4, R) :-

I_9 $L \to *R.,\ =/\$$ Fi I_7

goto (I_4, L):

I_2: $R \to L.$, =|$ Fi I_8

goto $(I_4, *)$:

I_4: $L \to *.R$, =|$
$R \to .L$, =|$
$L \to .*R$, =|$
$L \to .id$, =|$

goto (I_4, id):

I_5 $L \to id.$, =|$ Fi I_5

goto (I_6, R):

I_9 $S \to L=R.$, $

goto (I_6, L):

I_{10} $R \to L.$, $

goto $(I_6, *)$

$L \to *.R$, $
$R \to .L$, $
I_{11} $L \to .*R$, $
$L \to .id$, $

goto (I_6, id)

I_{12} $L \to id.$, $ Fi

goto (I_{11}, R):

I_{13} $L \to *R.$, $ Fi

goto (I_{11}, L):

I_{10}: $R \to L.$ Fi

goto (I_4, L):

I_2: $R \to L.$, =|$ Fi I_2

goto $(I_4, *)$:

I_4: $L \to *.R$, =|$
$R \to .L$, =|$
$L \to .*R$, =|$
$L \to .id$, =|$

goto (I_4, id):

I_5 $L \to id.$, =|$ Fi I_5

goto (I_6, R):

I_9 $S \to L=R.$, $

goto (I_6, L):

I_{10} $R \to L.$, $

goto $(I_6, *)$

$L \to *.R$, $
$R \to .L$, $
I_{11} $L \to .*R$, $
$L \to .id$, $

goto (I_6, id)

I_{12} $L \to id.$, $ Fi

goto (I_4, R):

I_3 $L \to *R.$, $ Fi

goto (I_{11}, L):

I_{10}: $R \to L.$ Fi

goto (I_0, c):

I_3 $C \to c.C, cld$
$C \to .CC, cld$
$C \to .d, cld$ I_{36}

goto (I_0, d)
I_4 $C \to d., cld$ Fi

goto (I_2, C)
I_5 $S \to CC., \$$ Fi

goto (I_2, c)
I_6 $C \to c.C, \$$
$C \to .d, \$$
$C \to .CC, \$$ I_3

goto (I_2, d)
I_7 $C \to d., \$$ Fi

goto (I_6, C)
I_8 $C \to CC., \$$

goto (I_3, C)
I_8 $C \to CC., cld$ Fi

goto (I_3, c)
$C \to c.C, cld$
$C \to .CC, cld$
$C \to .d, cld$
goto C

goto (I_4, *):

$L \rightarrow * . R, \$$ } I_{11}

goto (I_{11}, id):

$L \rightarrow$ id . , \$ I_{12}

Combine I_4 & I_{11} → I_{411}
Combine I_5 & I_{12} → I_{512}
Combine I_7 & I_{13} → I_{713}
Combine I_8 & I_{10} → I_{810}

Construction of LALR Parse tree.

	Action				goto		
	id	=	*	$	S	L	R
0	S512		S411		1	2	3
1				Accepted			
2		S6		R5			
				R2			
3							
411	S512		S411			810	713
512		R4		R4			
6	S512		S411			810	9
713		R3		R3			
810		R5		R5			
9				R1			

Stack	I/P String	Action
$0	id=id$	S 512
$0 id 512	=id$	reduce L→id
$0 L 2	=id$	S 6
$0 L 2 = 6	id$	S 512
$0 L 2 = 6 id 512	$	reduce L→id
$0 L 2 = 6 L 8 10	$	reduce R→L
$0 L 2 = 6 R 9	$	S→L=R
$0 1	$	Accept

Q// Design LALR Parsing table [GTU, W-24 OR Q-3(c)]

Ans
S → CC
C → cCId

Augmented Grammar

I_0: S' → .S, $
S → .CC, $
C → .cC, $ cId
C → .d, $ cId

goto (I_0, S)

I_1: S' → S., $ Fi

goto (I_0, C)

I_2: S → C.C, $
C → .cC, $
C → .d, $

goto (I_0, C):

I_3 $C \to c.C, cId$ $C \to .CC, cId$ $C \to .d, cId$ I_{36}

goto (I_0, d)

I_4 $C \to d., cId$ Fi I_{74}

goto (I_2, C)

I_5 $S \to CC., \$$ Fi

goto (I_2, c)

I_6 $C \to c.C, \$$ $C \to .d, \$$ $C \to .CC, \$$ $I_3 I_6$

goto (I_2, d)

I_7 $C \to d., \$$ Fi $I_7 I_4$

goto ~~(I_6, C)~~

I_8 ~~$C \to CC., \$$~~

goto (I_3, C)

I_8 $C \to CC., cId$ Fi I_{89}

goto (I_3, c)

$C \to c.C, cId$ $C \to .CC, cId$ $C \to .d, cId$ } I_3

goto (I_3, d)

I_4 $C \to d., cId$ Fi

goto (I_6, C):

I_9 $C \to cC.$, \$ I_{89}

goto (I_6, c)

I_6: $C \to c.C$, \$
$C \to .cC$, \$
$C \to .d$, \$

goto (I_6, d):

I_7 $C \to d.$, \$ Fi

Construction of CLR

	Action			goto	
	c	d	$	S	C
0	S2	S47		1	2
1					
2	S36	S47			5
3	S36	S47			89
4					
5					
6	S36	S47			89
7					
8					
9					

Combine I_3 & I_6 into I_{36}

" " I_7 & I_4 into I_{74}

Combine I_1 & I_9 into I_{89}

Stack	I/P String	Action
$0	id=id$	S 5 12
$ 0 id 5 12	=id$	reduce L→id
$ 0 L 2	=id$	S 6
$ 0 L 2 = 6	id$	S 5 12
$ 0 L 2 = 6 id 5 12	$	reduce L→id
$ 0 L 2 = 6 L 8 10	$	reduce R→L
$ 0 L 2 = 6 R 9	$	S→L=R
$ 0 1	$	Accept

Q Design LALR Parsing table [GTU, W-24 OR @-3(c)]

Ans
S → CC
C → cCld

Augmented Grammar

I_0: S' → .S, $
S → .CC, $
C → .cC, $ cld
C → .d, $ cld

goto (I_0, S)
I_1: S' → S., $ Fi

goto (I_0, C)
I_2: S → C.C, $
C → .cC, $
C → .d, $

goto (I_0, C):

I_2 $C \to C.C, c/d$ $C \to .CC, c/d$ $C \to .d, c/d$ I_{36}

goto (I_0, d)

I_4 $C \to d., c/d$ Fi I_{74}

goto (I_2, C)

I_5 $S \to CC., \$$ Fi

goto (I_2, c)

I_6 $C \to c.C, \$$ $C \to .d, \$$ $C \to .CC, \$$ $I_3 I_6$

goto (I_2, d)

I_7 $C \to d., \$$ Fi $I_7 I_4$

goto (I_6, C)

I_8 $C \to CC., \$$

goto (I_3, C)

I_8 $C \to CC., c/d$ Fi I_{89}

goto (I_3, c)

$C \to c.C, c/d$ $C \to .CC, c/d$ $C \to .d, c/d$ $\}$ I_3

goto (I_3, d)

I_4 $C \to d., c/d$ Fi

goto (I_6, C):

I_9 $C \to cC.$, \$ I_{89}

goto (I_6, c)

$C \to c.C$, \$

I_6: $C \to .cC$, \$

$C \to .d$, \$

goto (I_6, d):

I_7 $C \to d.$, \$ A

Construction of CLR

	Action			goto	
	c	d	$	S	C
0	S2	S47		1	2
1					
2	S36	S47			5
3	S36	S47			89
4					
5					
6	S36	S47			89
7					
8					
9					

Combine I_3 & I_6 into I_{36}

" " I_7 & I_4 into I_{74}

Combine I_1 & I_9 into I_{89}

Shift reduce parser

- The shift reduce parser performs following basic operations:

1. **Shift**: Moving of the symbols from input buffer onto the stack, this action is called shift.
2. **Reduce**: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.
3. **Accept**: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.
4. **Error**: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

Example: Shift reduce parser

Grammar:
E□E+T | T
T□T*F | F
F□id
String:
id+id*id

Stack	Input Buffer	Action

Operator precedence parsing

- **Operator Grammar**: A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator grammar.
- Example: E□ EAE | (E) | id

  A□ + | * | -
- Above grammar is not operator grammar because right side ***EAE*** has consecutive non terminals.
- In operator precedence parsing we define following disjoint relations:

Relation	Meaning
a<·b	a "yields precedence to" b
a=b	a "has the same precedence as" b
a·>b	a "takes precedence over" b

Precedence & associativity of operators

Operator	Precedence	Associative
↑	1	right
*, /	2	left
+, -	3	left

Steps of operator precedence parsing

1. Find Leading and trailing of non terminal
2. Establish relation
3. Creation of table
4. Parse the string

Leading & Trailing

Leading:- Leading of a non terminal is the first terminal or operator in production of that non terminal.

Trailing:- Trailing of a non terminal is the last terminal or operator in production of that non terminal.

Example: E□E+T | T

T□T*F | F

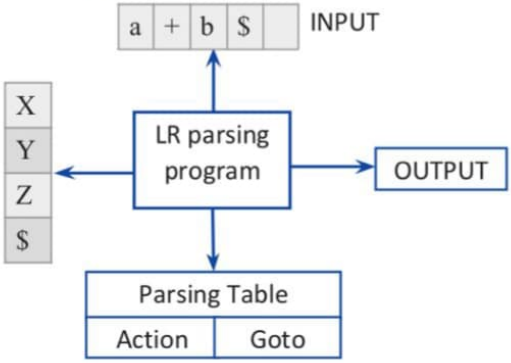
F□id

Non terminal	Leading	Trailing
E		
T		
F		

Introduction to LR Parser

LR parser

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR(k) parsing:
  1. The "L" is for left to right scanning of input symbol,
  2. The "R" for constructing right most derivation in reverse,
  3. The "k" for the number of input symbols of look ahead that are used in making parsing decision.



C → • d

Parsing table

State	Action			goto		
	c	d	$	S	T	C
0	S3	S4			1	2
1			ACCEPT			
2	S3	S4				5
3	S3	S4				6
4	r4	r4	r4			
5			r2			
6	r3	r3	r3			

Parsing of string cdcd will be

Stack	Input	Action
$ 0	cdcd $	shift 3
$ 0c3	dcd $	shift 4
$ 0c3d4	cd $	reduce by C → d
$ 0c3C6	cd $	reduce by C → cC
$ 0C2	cd $	Shift 3
$ 0C2c3	d $	Shift 4
$ 0C2c3d4	$	reduce by C → d

is LL(1) but not SLR(1). **GTU : Winter-14, Summer-17,19 Marks 7**

Solution : For LL(1) grammar refer example 3.3.16. To show that this grammar is not SLR(1) we will construct a canonical collection of set of item -

I_0 : S → • AaAb
S → • BbBa
A → •
B → •

I_1 : goto (I_0, A)
S → A • aAb

I_2 : goto (I_0, B)
S → B • bBa

I_3 : goto (I_1, a)
S → Aa • Ab
A → •

I_4 : goto (I_2, b)
S → Bb • Ba
B → •

I_5 : goto (I_3, A)
S → AaA • b

I_6 : goto (I_4, B)
S → BbB • a

I_7 : goto (I_5, b)
S → AaAb •

I_8 : goto (I_6, a)

S → BbBa •

The parsing table can be constructed as follows -

State	Action				goto	
	a	b	$	S	A	B
0	r3/r4	r3/r4			1	2
1	S3					
2		S4				
3	r3	r3			5	
4	r4	r4				6
5		S7				
6	S8					
7						
8						

As conflicting entries appear in above table. This grammar is not SLR(1).

Example 3.7.12 *Construct SLR parsing table for the following grammer.*
1) E → E – T | T 2) T → F ↑ T | F 3) F → (E) | id **GTU : Summer-15, Marks 7**

Solution : We will construct a canonical set of items for given production rules.

I_3 : goto (I_0, F)
T → F • ↑ T
T → F •

I_4 : goto (I_1, ()
F → (• E)
E → • E – T
E → • T
T → • F ↑ T
T → • F
F → • (E)
F → • id

I_5 : goto (I_0, id)
F → id •

I_6 : goto (I_1, –)

Ch-4 Error Recovery

Q. Explain Error Recovery Strategies
[4m & 7m sometimes]

Ans
1) Panic mode
2) Phase Level Recovery
3) Error Production
4) Global Generation

1) Panic Mode
→ On discovering an error, the parser discards input symbols one at a time untill one of a designated a set of synchronizing tokens is found.
→ The synchronizing tokens are usually delimiters, such as semicolon or },
→ These tokens indicate an end of the statement
→ If there is less number of errors in the same statement then this strategy is best choice.

Eg:

int a, 5abcd, sum, $2;
↑ ↑
Panic mode stop

2) Phase level Recovery

→ In this method, on discovering an error parser performs local correction on remaining input

→ The local correction can be
1) Replacing comma by semicolon
2) Deletion of semicolons
3) Inserting missing semicolon

→ This type of local correction is decided by compiler designer

→ This method is used in many error-repairing compilers

3) Error Production

→ If we have good knowledge of common errors that might be encountered, then we can augment the grammar for corresponding language with error production that generates the erroneous constructs

→ Then we use the grammar augmented by these error production to construct a parser

→ If error production is used then, during parser we can generate appropriate error message and parsing can be continued

Eg: abcd

S → A
A → aA | bA | a | b
B → cd

X

E → SB
S → A
A → aA | bA | a | b
B → cd

✓

4) Global Correction

→ We often want such a compiler that makes very few changes in processing an incorrect input string

→ Given an incorrect input string x and Grammar G, the algorithm will find a parse tree for a related string y, such that number of insertions, deletions and changes of token require to transform x into y is as small as possible

→ Such methods increase time and space requirements at parsing time

→ Global correction is thus simply a theoretical concept

Page No.
Date

Ch-5 Run Time Environments

Ch-5 Intermediate Code Generation

0 Construction of Syntax trees

1) mknode (op, left, right)

2) mkleaf (id, entry to Symbol Table)

3) mkleaf (num, value)

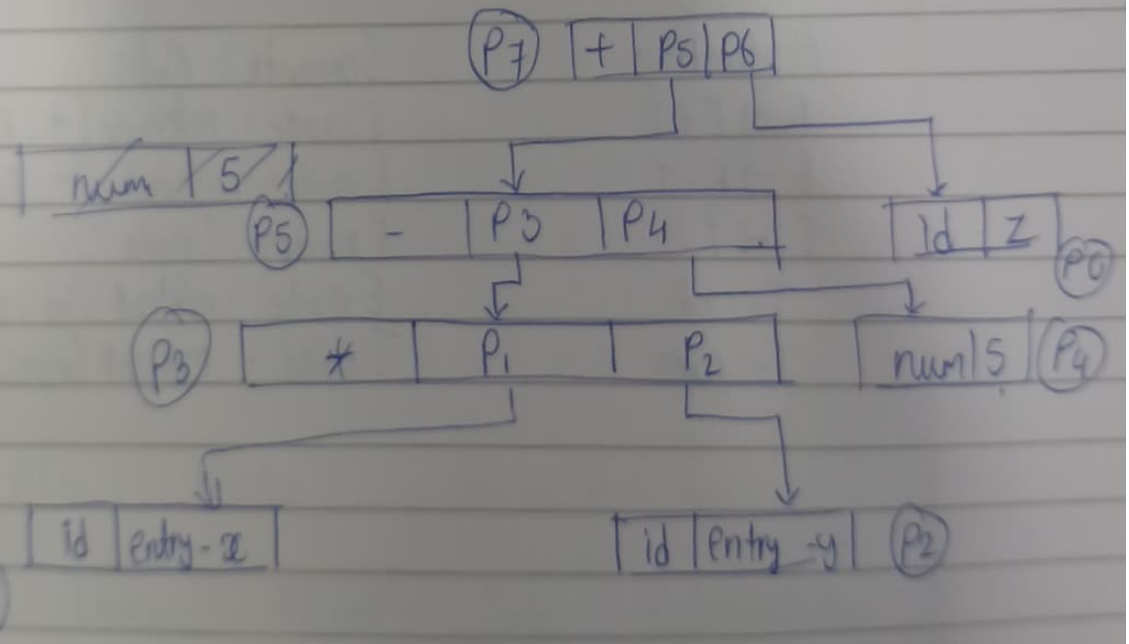
Eg:1 $x * y - 5 + z$

Steps to Construct Syntax tree for $x * y - 5 + z$

Symbol	Operation
x	P_1 = mkleaf (id, entry-x)
y	P_2 = mkleaf (id, entry-y)
*	P_3 = mknode (*, P_1, P_2)
5	P_4 = mkleaf (num, 5)
-	P_5 = mknode (-, P_3, P_4)
z	P_6 = mkleaf (id, entry-z)
+	P_7 = mknode (+, P_5, P_6)

SDD for x+y-5+z

Productions	
$E \to E_1 + T$	Enode = mknode(+, E_1.node, T.node)
$E \to E_1 - T$	Enode = mknode(-, E_1.node, T.node)
$E \to E_1 * T$	Enode = mknode(*, E_1.node, T.node)
$E \to T$	Enode = T.node
$T \to id$	T.node = mkleaf(id, id.entry)
$T \to num$	T.node = mkleaf(num, num.value)



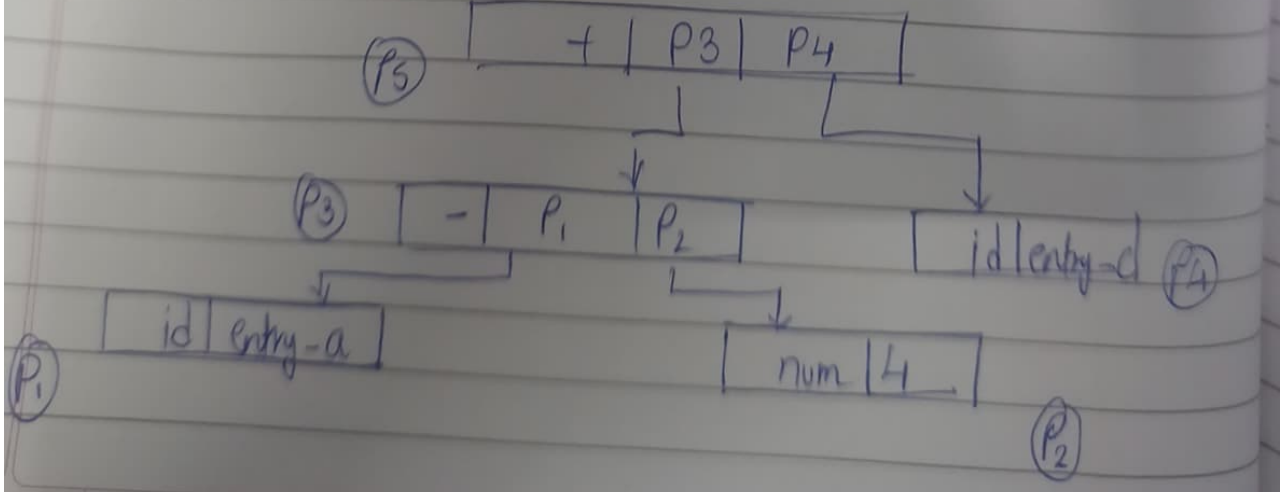
Eg: a - 4 + c

Steps

P_1 = mkleaf (id, entry-a)
P_2 = mkleaf (num, 4)
P_3 = mknode (-, P_1, P_2)
P_4 = mkleaf (id, entry-c)
P_5 = mknode (+, P3, P4)

SDD

Productions	Semantic Rules
$E \to E+T$	E.node = mknode (+, E.node, T.node)
$E \to E-T$	E.node = mknode (-, E.node, T.node)
$E \to T$	E.node = T.node
$T \to id$	E.node = mkleaf (id, entry to ST)
$T \to num$	T.node = mkleaf (num, num.value)



Page No
Date

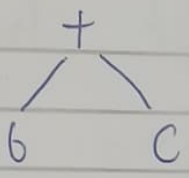
o Forms of Intermediate Code:-

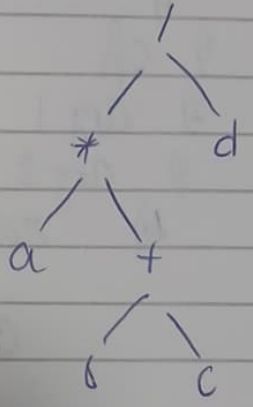
1) Using Syntax trees or AST

2) Postfix notation

3) Three - Address code representation

Eg: a * (b+c)/d





(1) Syntax Trees

Eg: (a+b)*c Infix

Pfix Postfix ab+c*

Eg: (3) Three Address Code Representation

(1) Each Instruction should contain atmost 3 addresses

(2) Atmost 1 operator on RHS

Represented by:-

1) Quadruple
2) Triple
3) Indirect Triple

Eg: $x + y * z$

$t_1 = y * z$
$t_2 = x + t_1$

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

Quadruple contains four fields
1) op
2) arg 1
3) arg 2
4) result

$a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$

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

	op	arg 1	arg 2	Result
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	* -	b c	t_3	t_4 t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5

2) Triple

3 fields

	op	arg 1	arg 2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

3) Indirect triple

$a = b * -c + b * -c$

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

Q/ Explain Quadruple, Triple and indirect Triple with example [GTU]

Ans Quadruple

→ It is a structure with at the most four fields such as op, arg1, arg2, and result

→ The op field is used to represent the internal code for operator

→ The arg1 and arg2 represent the two operands

→ And result field is used to store the result of an expression

$x : -a * b + -a * b$

$t_1 : -a$

$t_2 : t_1 * b$

$t_3 : -a$

$t_4 : t_3 * b$

$t_5 : t_2 + t_4$

$x : t_5$

	op	arg1	arg 2	Result
(0)	-	a		t_1
(1)	*	t_1	b	t_2
(2)	-	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

2) Triple

→ To avoid entering temporary names into symbol table, we might refer a temporary value by the position of statement that computes it.

→ It has three fields: op, arg 1 & arg 2

	op	Arg 1	Arg 2
(0)	-	a	
(1)	*	b	(0)
(2)	-	a	(1)
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	x	(4)

3) Indirect Triple

→ In the Indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement

→ This implementation is called indirect triples

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

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

$t_1 = (a+b)$

x
$t_1 = a+b$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$t_4 = a+b$
$t_5 = t_4 * c$
$t_6 = t_3 * t_5$

$t_1 = a+b$
$t_2 = c+d$
$t_3 = b*c$
$t_4 = a+t_3$
$t_5 = t_1 * t_2$
$t_6 = t_5 * t_4$
$t_7 = -t_6$

Quadruple

	op	arg 1	arg 2	Result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	+	a	b	t_4
(4)	*	t_4	c	t_5

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

$t_1 = a * b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2 + t_3$
$t_5 = a+b$
$t_6 = t_5 + t_3$
$t_7 = t_4 - t_6$

Quadruple

	Op	Arg 1	Arg 2	Result
(0)	*	a	b	t_1
(1)	uminus	t_1		t_2
(2)	+	c	d	t_3
(3)	+	t_2	t_3	t_4
(4)	+	a	b	t_5
(5)	+	t_5	t_3	t_6
(6)	-	t_4	t_6	t_7

Triple

	op	arg 1	arg 2
(0)	*	a	b
(1)	uminus	(0)	
(2)	+	c	d
(3)	+	(1)	(2)
(4)	+	a	b
(5)	+	(4)	(2)
(6)	-	(3)	(5)

Indirect Triple

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)
106	(6)

Page No
Date

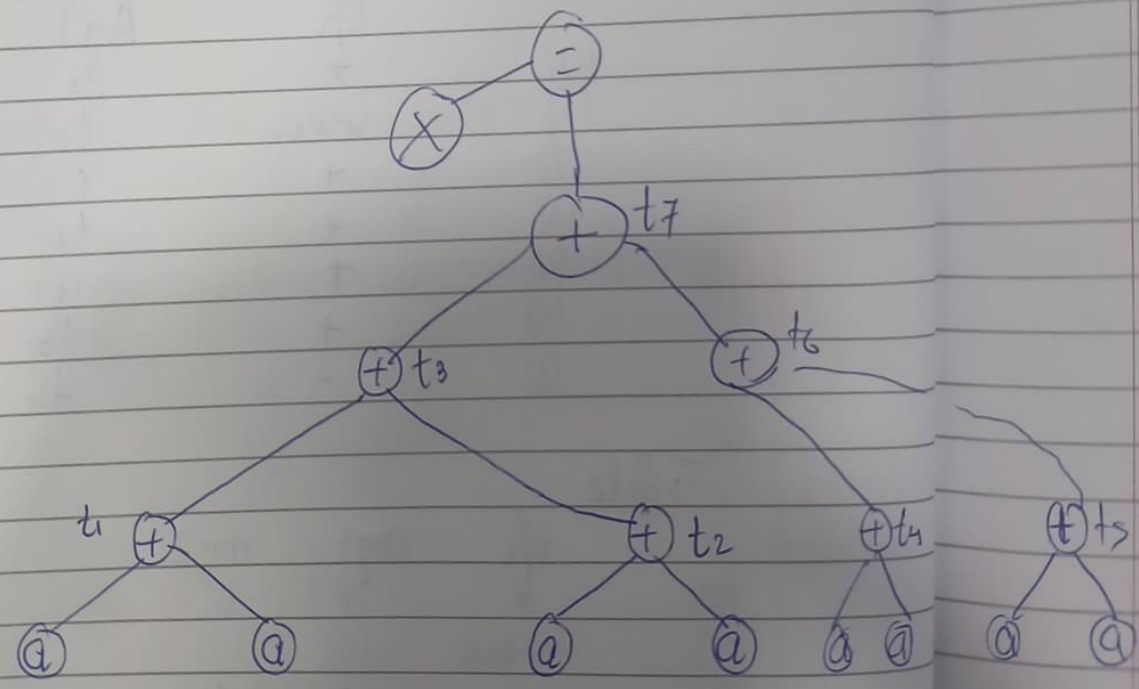
Directed Acyclic graph

t_3 t_6

$X = (((a+a) + (a+a)) + ((a+a) + (a+a)))$

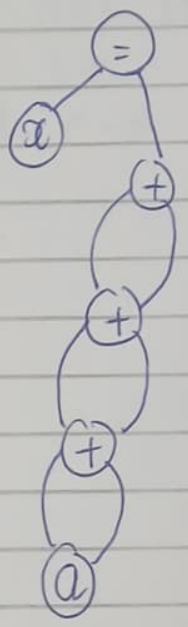
t_1 t_2 t_2 t_5

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

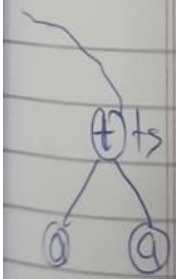


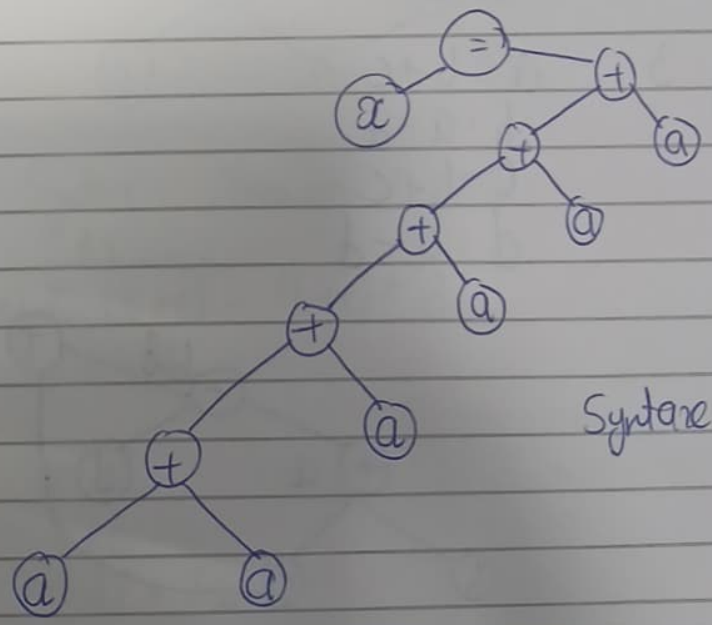
a + a Syntax tree

DAG



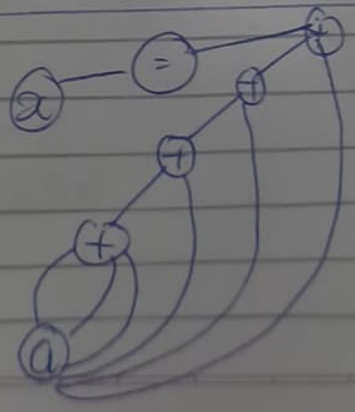
X = a+a+a+a+a+a





Syntax

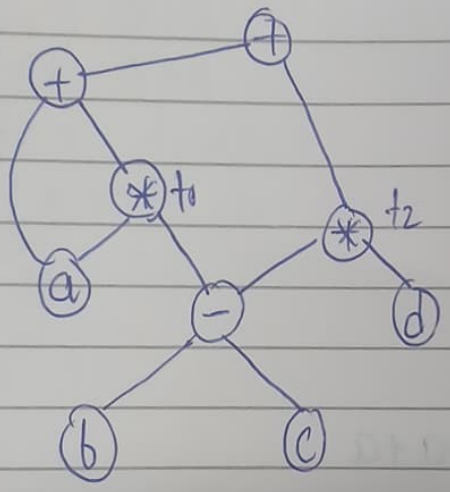
DAG



Eg: 2

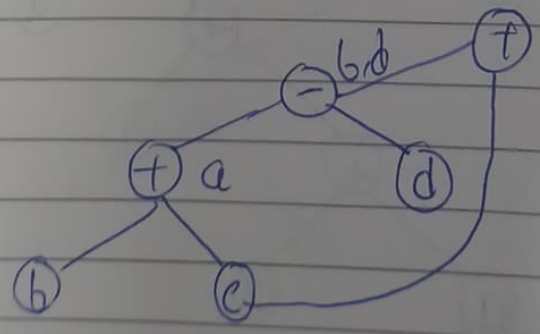
$$\underset{t_1}{a} + \underset{t_2}{a} * \underset{t_0}{(b-c)} + \underset{t_2}{(b-c)} * d$$

$$a + t_1 + t_2$$



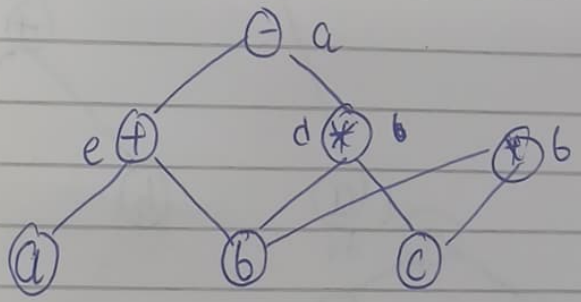
Eg: 3

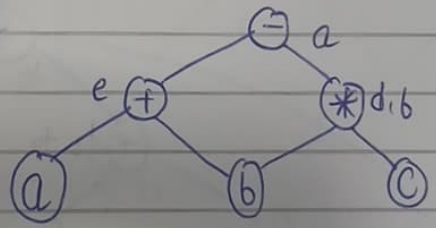
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$



Eg:4

```
a = b * c
b = a + b
b = b * c
a = e - d
```



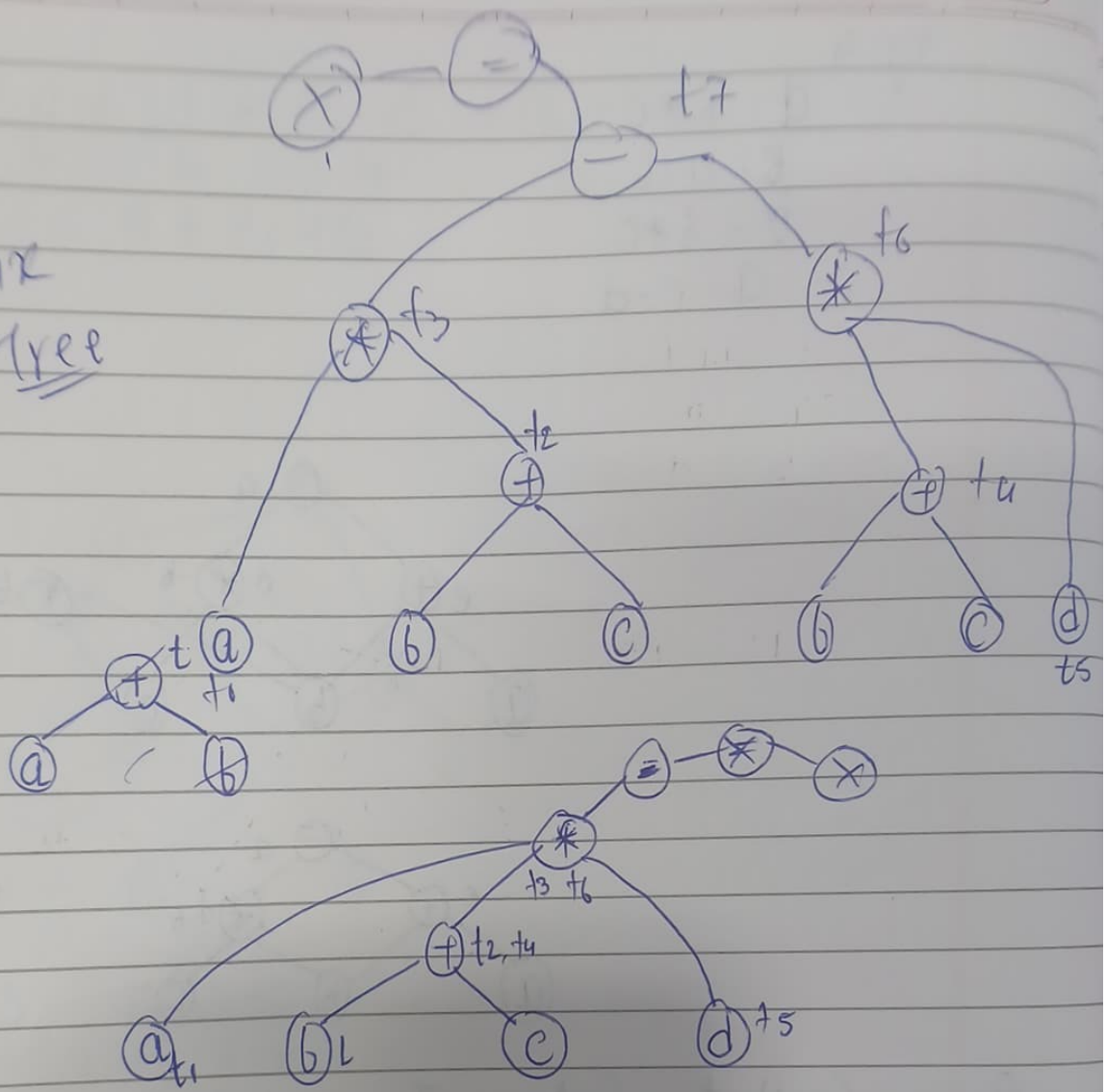


Q1 Construct Syntax Tree & DAG

$$X = \overbrace{\underbrace{a}_{t_1} * \underbrace{(b+c)}_{t_2}}^{t_3} - (b+c) * d$$

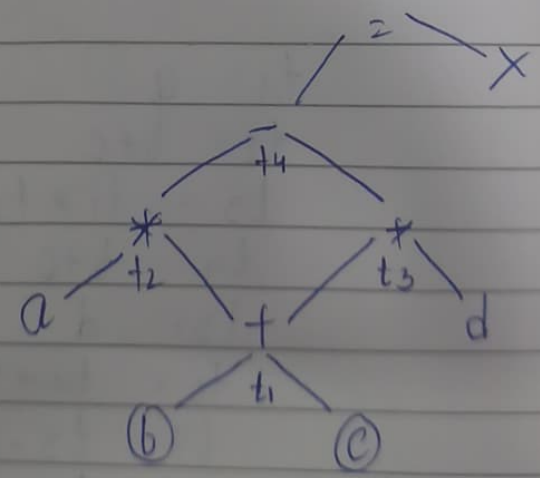
```
t1 = a
t2 = b + c
t3 = t1 * t2
t4 = b + c
t5 = d
t6 = t4 * t5
t7 = t3 - t6
X = t7
```

Syntax Tree



DAG

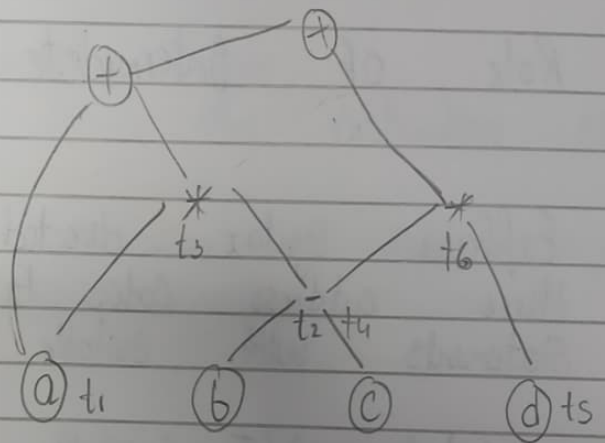
$t_1 = b + c$
$t_2 = a * t_1$
$t_3 = t_1 * d$
$t_4 = t_2 - t_3$
$x = t_4$



Q. $a + a * (b - c) + (b - c) * d$

(t_3 and t_6)

$t_1 = a$
$t_2 = b - c$
$t_3 = t_1 * t_2$
$t_4 = b - c$
$t_5 = d$
$t_6 = t_4 * t_5$
$t_7 = t_3 + t_6$
$t_8 = a + t_7$



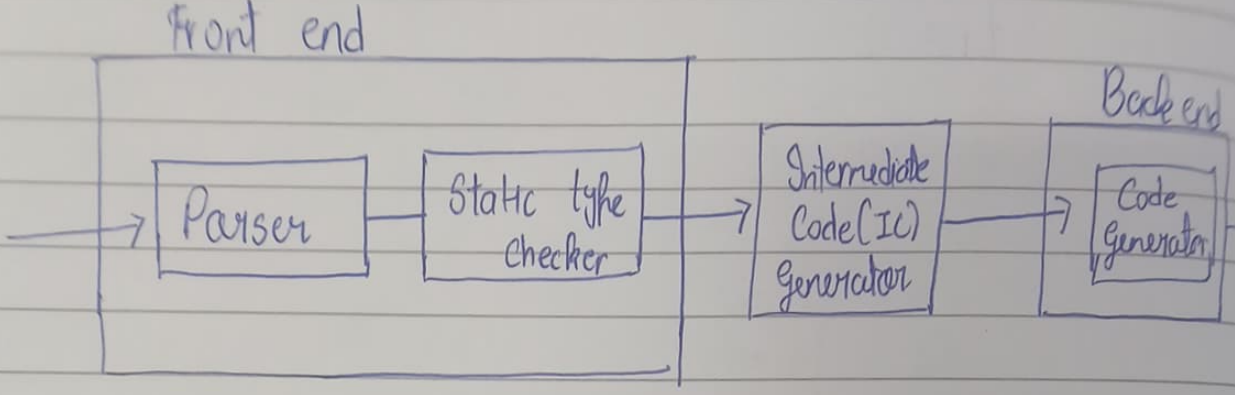
Q// Write three address code can be asked

$a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

Page No.
Date

Intermediate Code Generation



Role of Intermediate Code Generation

Q/ Explain Syntax directed definition to produce three address codes for flow of control statements with suitable example

Ans Watch YT Video Sudhakar Atchala

Ch-6 Run-Time Environment

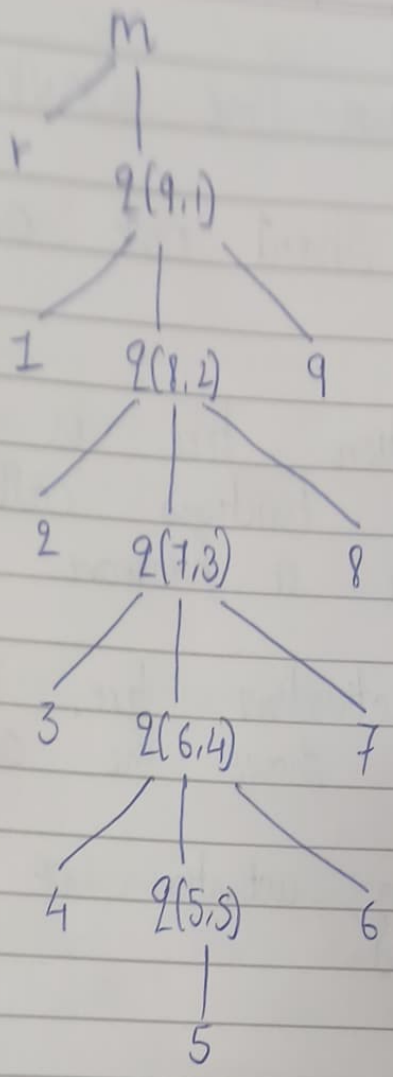
Q. Write a short note on Activation Tree

Ans An activation tree is like a tree showing which functions call which other functions while a program runs

In an activation tree,
1) Each node shows the activation of procedures
2) For showing activation of main program root is used
3) If

Properties of Activation tree are
1) Each node represents an activation of procedure
2) The root shows activation of main function
3) The node for procedure 'x' is the parent node for procedure 'y' if and only if the control flows from procedure x to procedure y

Eg. Activation tree for QuickSort



Write a short note on Activation Record

ns The Activation Record is a block of memory used for managing information needed by a single execution of a procedure

Return Value
Actual Parameters
Control Link (Dynamic
Access Link (Static)
Saved Machine Status
Local Variables
Temporaries

Fields of Activation Record

1) Temporary values:-
The temporary values are needed during the evaluation of expressions. Such variables are stored in temporary field of activation record

2) Local Variables
→ The local data is a data that is local to the execution of procedure is stored in this field of activation record.

3) Saved Machine Status:-
→ This field holds the information regarding the status of machine just before the procedure is called. This field contains the machine registers and program counter

4) Control link
→ This field is optional. It points to the activation record of calling procedure. also called Dynamic Link

5) Access Link
→ Optional field
→ It refers to non local data in other activation record
→ also called static link field

6) Actual Parameters
→ Holds info about actual parameters
→ These actual parameters are passed to the called procedure

7) Return Values:- It is used to store result of function call

→ the size of each field of activation record is determined at the time when a procedure is called

Q// Compare Static V/S Dynamic Memory Allocation

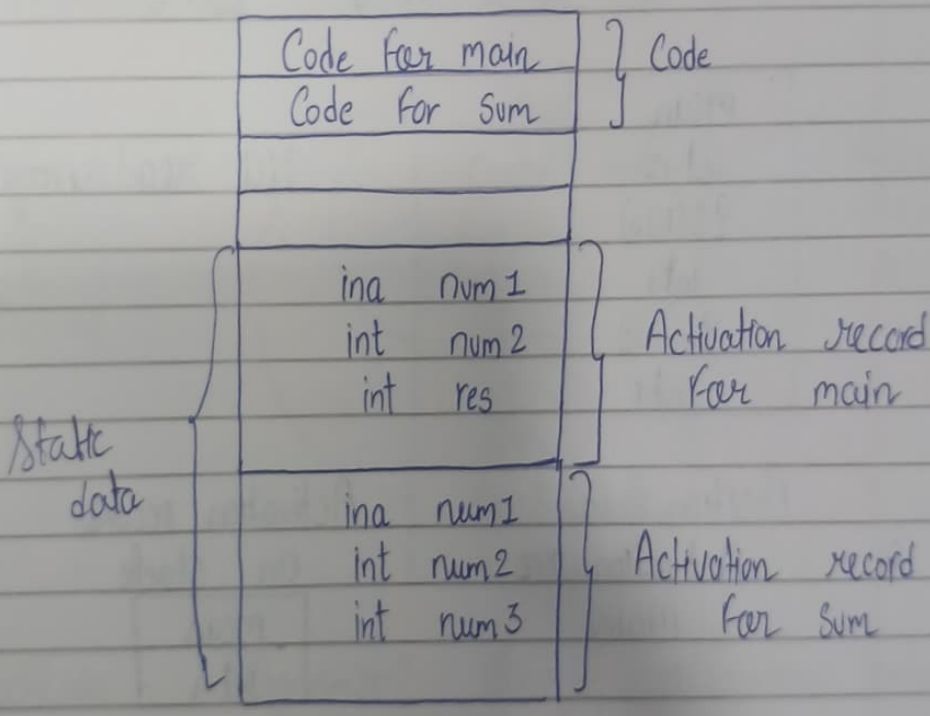
Ans

Static Memory Allocation	Dynamic Memory
1) Memory is set aside before the program starts running	1) set aside while the program is running as needed
2) Implemented using stack or data segment	2) heap
3) Fixed memory size	3) Flexible
4) Speed is Faster (allocation happens at compile time)	4) Speed is Slower (allocation happens during runtime)
5) Managed automatically by compiler	5) Manually managed by programmer
6) Less prone to errors	6) More
7) Less efficient in memory usage	7) More efficient

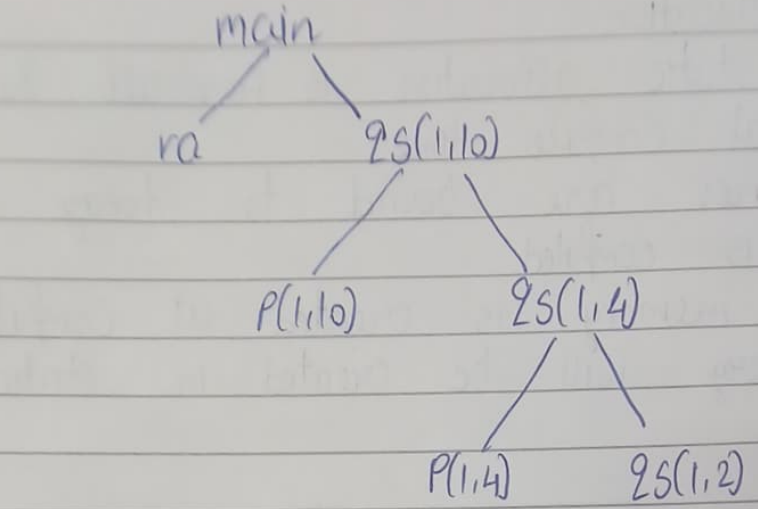
Q// Explain Storage allocation strategies

Ans
1) Static Allocation
2) Stack Allocation
3) Heap Allocation

1) Static Allocation
→ The static allocation is for all the data objects at compile time.
→ Names are bound to storage as the program is compiled
→ If memory is created at compile time then the memory will be created in static area only once



2) Stack Allocation
→ Each time a procedure is called, space for its local variables is pushed onto a stack, and
→ when procedure terminates, the space is popped off the stack
→ Locals are bound to fresh storage in each activation
→ Locals are deleted when the activation ends



main
int n
qs(1,10)
int i
qs(1,4)
int i

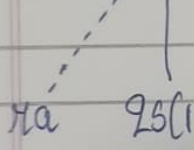
ra: read array()

	Position in Activation tree	Activation records on stack	Remarks
1)	main	main int n	Frame for main
2)	main / ra	main int n ra int i (ra flushed)	ra is activated
3)	main / \ . ra qs(1,10)	main int n qs(1,10) int i	Frame ra has been flushed & qs(1,10) is pushed

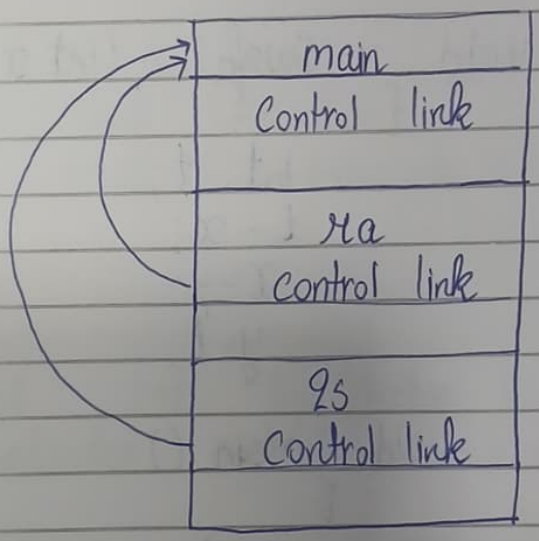
3) Heap Allocation

Stack Allocation is not possible when on
1) the values of local names must be retained when an activation ends
2) A called activation outlives the caller

Position in Activation tree



Activation records in the heap



Remarks

Retained activation record for ra

Q// Explain "Dynamic" Storage allocation techniques

Ans Write about Stack & Heap allocation

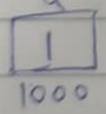
Q// Explain various parameter passing methods

Ans
1) call by value
2) call by reference
3) copy restore
4) call by name

1) Call by value
→ Simplest method
→ the actual parameters are evaluated and their r-values are passed to called procedure
→ the operations on Formal parameters do not change the values of actual parameters

Eg:
```
void swap (int x, int y) // Formal
{
    int t;
    t = x;
    x = y;
    y = t;
}
void main ()
{
    int a=1, b=2;
    swap (a,b); // Actual Parameters
    printf ("a=%d, b=%d", a,b);
}
```

// r-values
1, 2
// l-value
a=1



2) Call by ~~Value~~ Reference
→ This method is also called as call by address or call by location
→ The L-value, the address of actual parameter is passed to the called routines activation record
→ the values of actual parameters can be changed
→ Actual parameters should have an L-value

Eg:

```
void swap (int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
void main ()
{
    int a=1, b=2;
    swap (&a, &b);
    Print ("a=%d, b=%d", a, b);
}
```

3) Copy Restore
→ Hybrid of call by value & call by reference
→ Caller evaluates actuals & passes R-value to formals
→ During execution of called procedure, the actual parameters value is not affected
→ When control returns, R-values of formals are copied back into L-values of actuals

Eg:

```
int y;
copy restore (int x)
{
    x=2;
    y=0;
}
main() {
    y=10;
    copyrestore (y);
    print(y);
}
```

4) Call by Name
→ Actual parameters are literally substituted for the formals by macro expression or inline expansion

Eg:

```
int i=100;
void callbyname(int x)
  {
    print(x);
    print(x);
  }
main()
  {
    callbyname(i=i+1);
  }
```

Q// Write a short note on Symbol table Management.

Ans Symbol Table
→ Symbol tables are data structures that are used by compiler to hold information about source program constructs
→ It is used to store information about the occurence of various entities such as objects, classes, variable names, functions etc
It is used by both analysis phase and synthesis phase

o 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 source code are semantically correct.

Q. Difference between stack allocation and heap allocation of space

Ans

Parameter	Stack	Heap
1) Basic	Memory is allocated in a contiguous block	Memory is allocated in any random order
2) Allocation and De-allocation	Automatic by compiler instructions	Manual by the programmar
3) Cost	Less	More
4) Implementation	Easy	Hard
5) Main Issue	Shortage of memory	Memory fragmentation
6) Flexibility	Fixed size	Resizing is possible

Q. What do you mean by dangling references

Ans A dangling reference is a situation in CO where a pointer or reference points to a memory location that has been deallocated or released.
→ This can happen when an object is deleted or goes out of scope, but the reference to it still exists and is later accessed
Eg: int *p = malloc(sizeof(int));
*p = 10;
free(p)
printf("%d", *p) // invalid (dangling reference)

Ch-7 Code Generation & Optimization

Q-1 Explain various issues in design of code generator [Asked 4 times] [7m]

Ans Issues in Code Generation are
1) Input to code generator
2) Target Program
3) Memory Management
4) Instruction Selection
5) Register Allocation
6) Choice of evaluation
7) Approaches to ~~good~~ code generation

1) Input to code generator
→ Input to code generator consists of the intermediate representation of the source program.
→ Types of intermediate language are
1) Postfix Notation
2) Three Address Code
3) Syntax trees or DAGs
→ The detection of semantic error should be done before submitting the input to the code generator
→ The code generation phase requires complete error free intermediate code as an input

2) Target Program
The output maybe in form of
1) Absolute machine language : It can be placed in a memory location and immediately execute

2 Relocatable machine language: the subroutine can be compiled seperately. A set of relocatable object modules can be linked together and loaded for execution

3 Assembly language: Producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form

3) Memory Management
→ Mapping names in the source program to addresses of data objects in run time memory
→ is done cooperatively by the front-end and the code generator
→ We assume that a name in a three-address statement refers to a symbol table entry for the name
→ From the symbol table information, a relative address can be determined for the name in a data area

4) Instruction Selection
Example: the sequence of statements

```
a = b+c
d = a+e
```

would be translated into

```
MOV   b,R0
ADD   c,R0
MOV   R0,a
MOV   a,R0
ADD   e,R0
MOV   R0,d
```

```
MOV   b,R0
ADD   c,R0
ADD   e,R0
MOV   R0,d
```

→ So, we can eliminate redundant statements

5) Register Allocation
→ The use of registers is often subdivided into two sub problems
→ During register allocation, we select the set of variables that will reside in registers at a point in the program
→ During a subsequent register assignment phase, we pick the specific register that a variable will reside in
→ Finding an optimal assignment of registers to variables is difficult, even with single register value.
→ Mathematically the problem is NP-complete

6) Choice of Evaluation
→ The order in which computations are performed can affect the efficiency of the target code
→ Some computation orders require fewer registers to hold intermediate results than others
→ Picking a best order is another difficult, NP-complete problem.

7) Approaches to code generation
→ The most important criterion for a code generator is that it produces correct code
→ The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.

Page No.
Date

Ch-7

Q-2 Describe various code optimization techniques with example

Ans (1) Compile time Evaluation
(2) Common sub expressions elimination
(3) Code Movement or Code Motion
(4) Dead Code Elimination
(5) Reduction in Strength

(1) Compile time Evaluation
→ Compile time evaluation means shifting of computations from run time to compilation time
Two methods
1) Folding
→ In this technique, the computation of constant is done at compile time instead of execution time
For example:

$$length = \frac{22}{7} * d$$

Here folding is implied by performing the computation of 22/7 at compile time instead of execution time

2) Constant Propagation
→ Here value of variable is replaced and computation of an expression is done at compilation time
Example:

```
pi = 3.14;
r = 5;
Area = pi * r * r
```

pi → 3.14
r → 5

2) Common Sub Expression Elimination
→ Common sub expression is an expression appearing repeatedly in the program which is computed previously
→ If operands of this sub expression do not get changed at all then result of each sub expression is used instead of recomputing it each time.

For Example

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_4 = 4 * i$
$t_5 = n$
$t_6 = b[t_4] + t_5$

After Optimization
$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_5 = n$
$t_6 = b[t_1] + t_5$

3) Code Movement or Code Motion
There are two basic goals of code movement:-
i) To reduce the size of the code ie to obtain the space complexity

3) To reduce the frequency of execution of code ie to obtain the time complexity

Eg:

```
for (i=0; i<10; i++)
{
    x = y*5;
    k = (y*5) + 50;
}
```

After

```
temp = y*z
for (i=0; i<10; i++)
{
    x = z;
    k = z+50;
}
```

(4) Dead Code Elimination

→ The variable is said to be dead at a point in a program if the value contained into it is never been used

→ The code containing such a variable supposed to be dead code

Eg.

```
i=0
if (i==1)
{
    a = x+5
}
```

} Dead Code

→ If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

5) Strength Reduction

→ The strength of certain operators is higher than others

→ For instance strength of * is higher than +

→ In strength reduction technique the higher strength operators can be replaced by lower strength operators

For example

```
For (i=0; i<=50; i++)
{
    count = i*7;
}
```

count values as 7, 14, 21

temp = 7

```
For (i=0; i<=50; i++
{
    count = temp;
    temp = temp + 7;
}
```

```
begin
  prod := 0
       i := 1;
  do
     prod := prod + a[t1] * b[t
       i := i + 1;
  while i<=20
end
```

Q-3 Explain Basic Block with example

Ans A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

Input: A sequence of three-address statements
Output: A list of basic blocks with each three address statement in exactly one block
Method
1) We first determine the set of leaders. For that we use the following rules:
a) The first statement is leader
b) Any statement that is the target of a conditional or unconditional goto is a leader
c) Any statement that immediately follows a goto or conditional goto statement is a leader

2) For each leader, its basic block consists of the leader and all statements up to but not including the next leader or end of program

For example

(1) prod = 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a[t1]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)

Explain Peephole optimization technique

Peephole optimization is a technique used to make the generated target code (like machine code or assembly code) slightly better (faster or smaller).
→ It's considered a local optimization because it only looks at a very small number of instructions at a time

Common Peephole Optimization Techniques
1) Redundant Store & Loads Elimination
2) Flow of Control Optimization / Unreachable
3) Algebraic Simplification
4) Reduction in Strength
5) Machine Idioms

1) Redundant Store & Loads Elimination
→ It is used to remove instructions that are unnecessary

Eg:
Original:
```
MOV R0, x
MOV x, R0
```

Optimization: If the value of x or R0 hasn't changed between these two instructions, the second MOV x, R0 is redundant and can be deleted

2) How of Control Optimization
Explanation: Simplifying sequences of jump

Eg: 1 (Jump over Jump)

Original
goto L1
..
L1: goto L2

Can be changed to goto L2;
If L1 is not targeted by any other jump;
the L1: goto L2 might also become unreachable

Eg:2 (Conditional Jump)

Original:
if a<b goto L1
..
L1: goto L2

Optimization:
Can be changed to if a<b goto L2;

3) Algebric Simplification
→ Peephole optimization is an effective technique for algebric simplification
→ the statements such as x:x+0 or x:x*1 can be eliminated by Peephole optimization

4) Reduction in Strength

→ Certain machine instructions are cheaper than others
→ In order to improve performance of the intermediate code we can replace these instructions by equivalent cheaper instruction
→ For example, x^2 is cheaper than $x * x$
→ Similarly, addition & subtraction are cheaper than multiplication and division. So we can add effectively equivalent add & sub for mul & div.

5) Using Machine Idioms

→ The target instructions have equivalent machine instructions for performing some operations
→ Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency
→ Some machines have auto-increment or auto-decrement addressing modes (Eg: INC i)
→ These modes can be used in code for statement like i=i+1

Q. What is global optimization? Name the 2 types of analysis performed for global optimization. **3M S22**

Global Optimization refers to code improvement techniques that look at larger parts of the program, like whole functions or loops, instead of just small sequences of instructions (basic blocks). The aim is to make the program better by understanding how different sections interact.

The two main types of analysis used for global optimization are:

1. **Control Flow Analysis:** This involves figuring out the structure of the program – how execution can jump between different sections (basic blocks). It helps identify loops and different paths the program might take.
2. **Data Flow Analysis:** This analysis tracks how data (the values stored in variables) moves through the program along the paths identified by control flow analysis. It helps determine things like where a variable gets its value, where that value is used, and whether a calculation might be redundant across different program sections.

Ch-8 Instruction-level Parallelism

Q-1 Explain Code Scheduling Constraints

Ans Code scheduling is a form of program optimization that applies to the machine code that is produce by code generator

Three kinds of constraints
1) Control dependence constraints: All the operations executed in original program must be executed in the optimized one
2) Data Dependence constraints: The operations in the optimized program must produce the same result as the corresponding ones in the original program
3) Resource Constraints: The schedule must not oversubscribe the resources of the machine

→ These scheduling constraints gurantee that the optimized program produces the same result as original program
→ However, because code scheduling changes the order in which the operation execute, the state of the memory at any one point may not match any of the memory states in sequential execution

Q-2 Explain Basic Block Scheduling in brief

Ans 1) Data-Dependence Graphs:
→ First step. We create a graph where each instruction (operation) is a node.
→ An edge from instruction A to instruction B means B depends on A's result.

→ the edge is labeled with the delay needed between starting A and starting B
→ this graph shows us the fundamental order we must respect

2 List Scheduling of Basic Blocks
→ this is the main algorithm used for scheduling
→ It keeps a list of instructions that are "ready" to run (all their dependencies are met)
→ Cycle by cycle, it picks one or more instructions from the ready list and schedules them making sure not to use more resources (like multipliers or memory units) than the processor has
→ Once it scheduled, an instruction is removed, potentially making other instructions ready

3) Prioritized topological Orders:

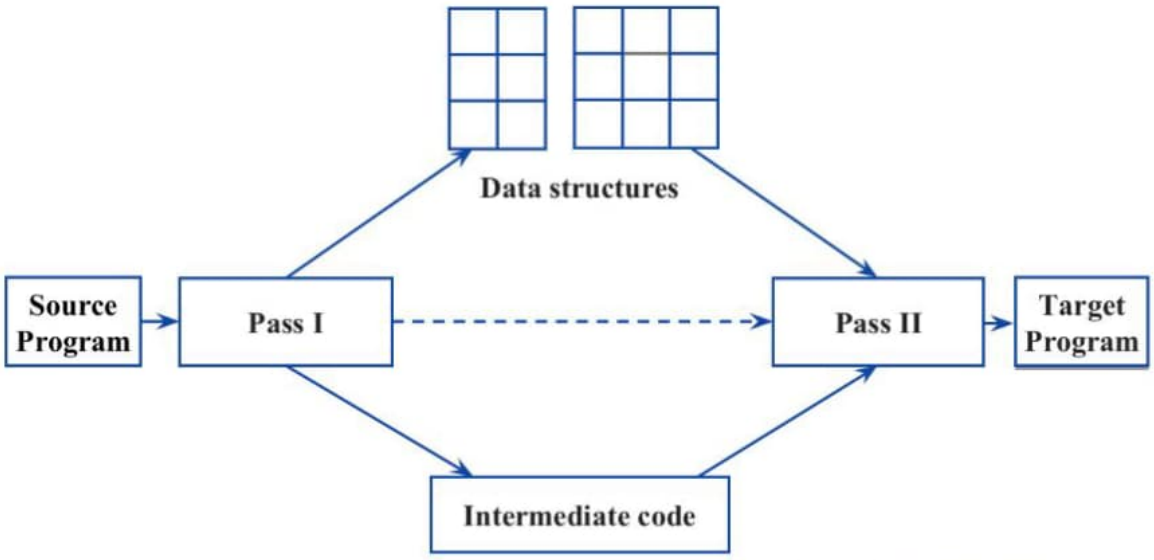
3. **Prioritized Topological Orders:**
    - This isn't a separate scheduling algorithm, but rather the **strategy** used *within* List Scheduling to decide **which** instruction(s) to pick from the ready list in each cycle. Since there might be multiple ready instructions, we need a way to prioritize.
    - Common priorities (heuristics) include:
        - **Longest Path:** Pick the instruction that is part of the **longest remaining chain** of dependent instructions first. Finishing tasks on the longest chain helps the entire block finish sooner.
        - **Resource Usage:** Pick the instruction that needs a **busy or rare resource** first. This frees up that limited part sooner so other waiting instructions can use it.
        - **Source Order:** If multiple instructions seem equally important (it's a tie), just pick the one that appeared **earlier in the original code**.
    - So, List Scheduling follows a topological order (respecting dependencies), but it uses these priorities to choose *which* valid topological order to create.

In short: You build the **Dependence Graph** first. Then you use the **List Scheduling algorithm**, and *within* that algorithm, you use a **Prioritized Topological Order** strategy to

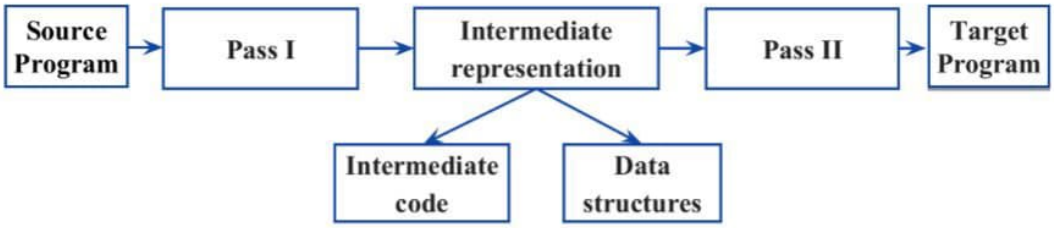
Pass structure of assembler

- A complete scan of the program is called pass.
- Types of assembler are:
  1. Two pass assembler (Two pass translation)
  2. Single pass assembler (Single pass translation)

Two pass assembler (Two pass translation)

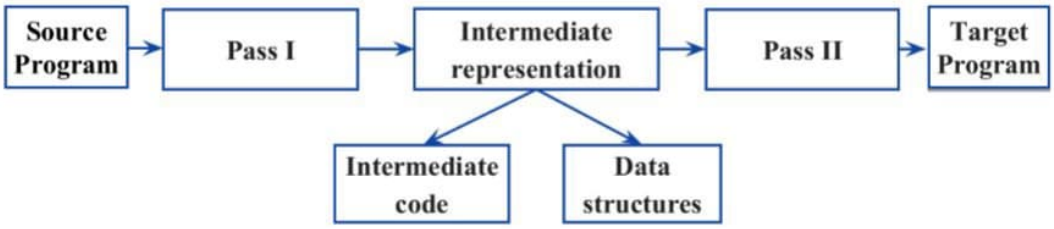


Two pass assembler (Two pass translation)



- The first pass performs analysis of the source program.
- The first pass performs Location Counter processing and records the addresses of symbols in the symbol table.
- It constructs intermediate representation of the source program.
- Intermediate representation consists of following two components:
  1. Intermediate code
  2. Data structures

Two pass assembler (Two pass translation)



- The second pass synthesizes the target program by using address information recorded in the symbol table.
- Two pass translation handles a forward reference to a symbol naturally because the address of each symbol would be known before program synthesis begins.

Use of a Symbol that precedes its definition in a program.

One pass assembler (One pass translation)

- A one pass assembler requires one scan of the source program to generate machine code.
- Location counter processing, symbol table construction and target code generation proceed in single pass.
- The issue of forward references can be solved using a process called back patching.