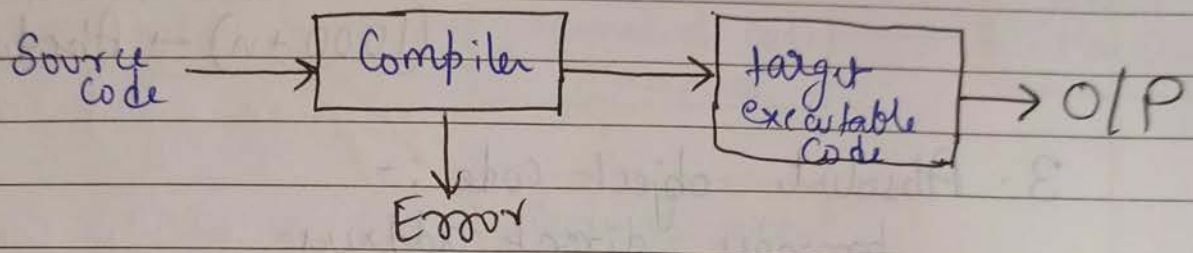
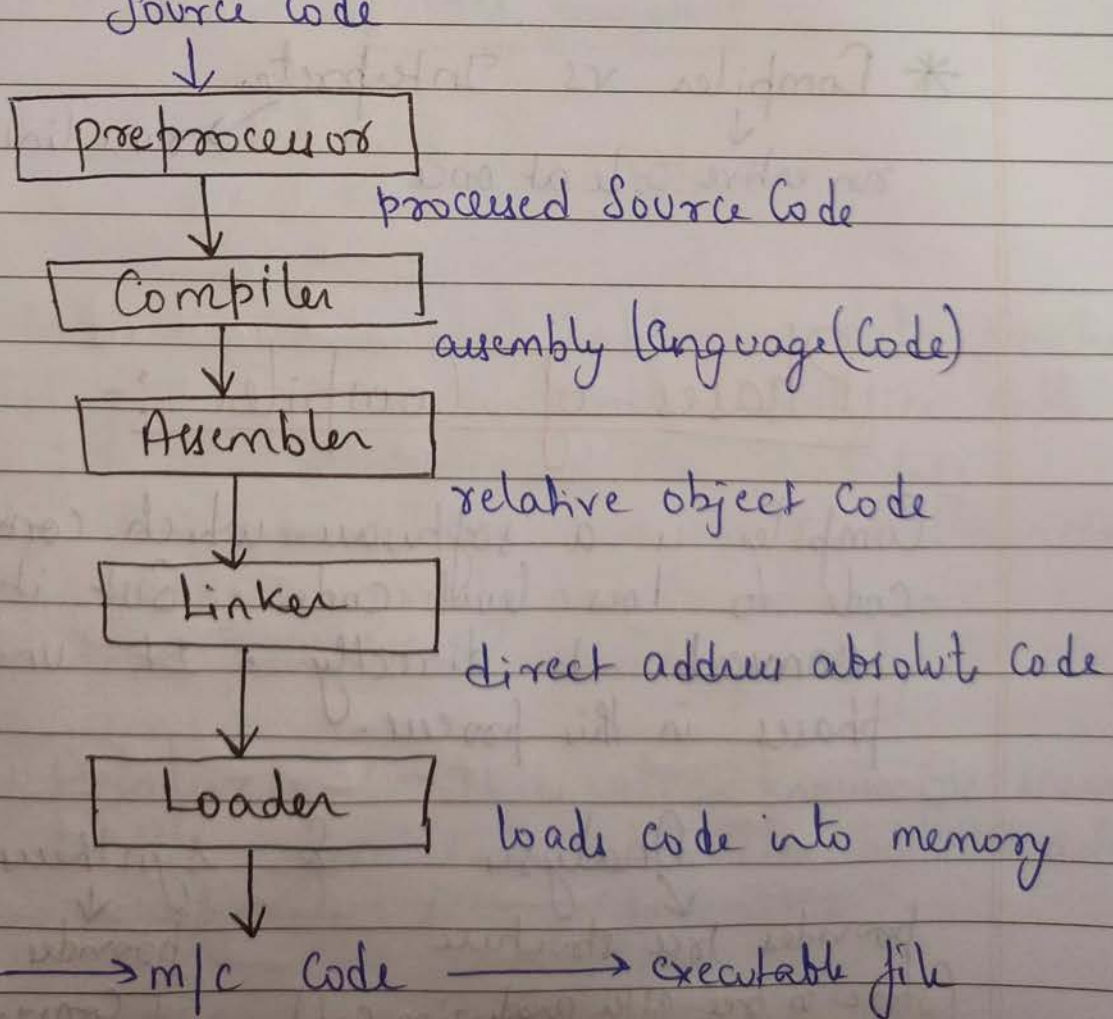
Compiler Design

Unit-1

Compiler :- Compiler is a execution environment that converts the source code to a target code (i.e high level language to machine level language).

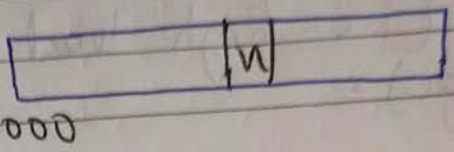


*



1. Assembly Code → in between high & low level language

2. Relocatable object code → provide relative address.



n → Relative address

(1000+n) → Absolute address

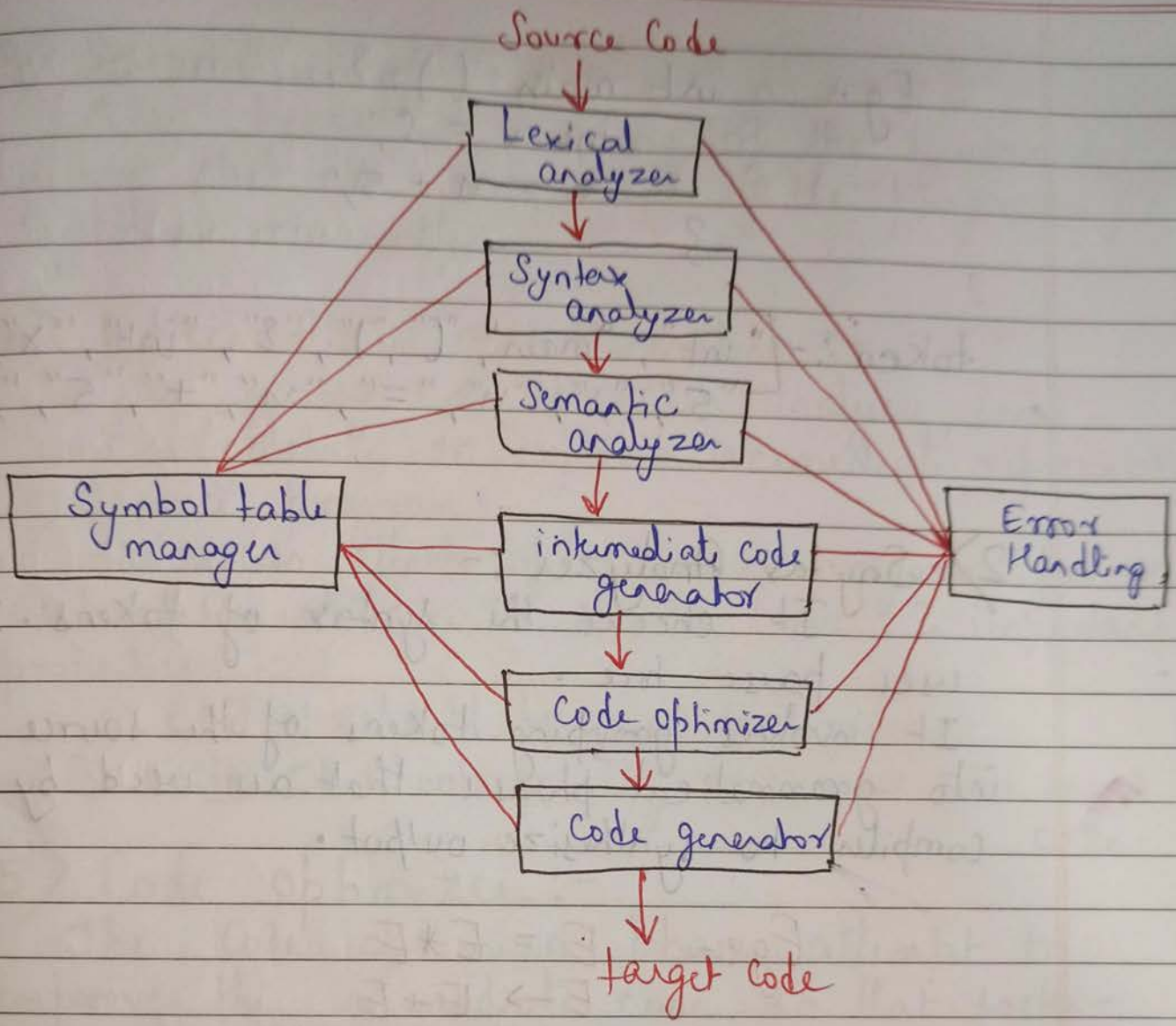
3. Absolute object code :-
provides direct address

* Compiler vs Interpreter
↓ run entire code at once
↘ runs line by line.

Phases of Compiler :-

Compiler is a software which convert high level code to low level code. But it does not convert it directly. It undergoes certain phases in this process.

Analysis & synthesis phase.
↓ provides tree structure (store in tree after analyzing code)
↓ provides m/c code (Convert tree structure to machine code)



Symbol table :- A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

* Lexical Analyzer :- It is also known as scanner. It reads the entire code character by character and convert it into tokens (lexeme).

Eg.
```
int main () {
    int x = 5;
    n = n + 5;
}
```

token :- ["int", "main", "(", ")", "{", "int", "x", "=", "5", ";", "n", "=", "n", "+", 5, ";", "}"]

2) Syntax Analyzer :-

It checks the syntax of tokens. It uses parse tree.

It involves grouping tokens of the source program into grammatical phrases that are used by compiler to synthesize output.

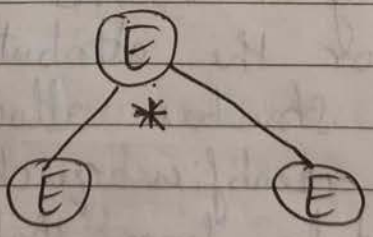
Eg:- $E = E*E$

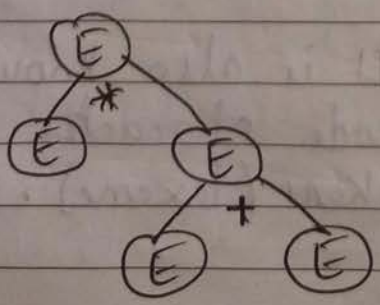
$E \rightarrow E+E$

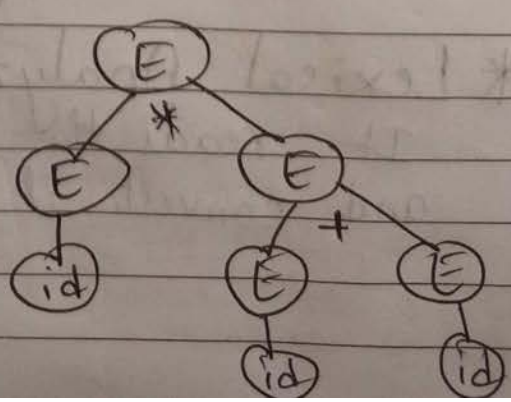
$E \rightarrow id$

Generate id * id + id.

1. 

2. 

3. 

4. 

3) Semantic analyzer :-
It checks the meaning of the syntax to ensure that the component of the program fit together manually.

4) Intermediate code Generation :-
After syntax and semantic analysis, some compiler generate an explicit intermediate representation of source program.
We can think of it as a program for an abstract machine. It should have 2 important properties :-
(i) It should be easy to produce
(ii) It should be easy to convert into target code.

5) Code optimizer :-
The code optimizer phase attempt to improve the intermediate code, so that faster-running machine code will result.
It removes the useless symbols without changing the meaning.

6) Code Generation :- The final phase of the compiler is the generation of target code which is machine dependent. It is unique for all the core system (i.e it is different for 32 bit & 64 bit and so on).

Errors in various Analysis phase :-

1) Lexical Analyzer :- Misspelling
Eg. main()

2) Syntax Analyzer :- Not in proper Syntax
Eg. int x = 5
printf("Hello)";

3) Semantics :- Syntax is valid or not (providing meaning or not)
Eg.
int x = 5.5;
char ch = 104.5;

In short Summary :-

machine independent code
↓
① → info about tokens
② → data types, scope
↓
③ → verify already stored information
↓
④ → Generate intermediate code based on info stored in Symbol table
↓
⑤ → optimize code based on info in Symbol table
↓
⑥ → machine dependent code

* front-end phase :- Machine independent
(analysis) Depend on programming language

Back-end phase :- Machine dependent
(synthesis) Doesn't Depend on programming language

* Passes in Compiler :-

1) One-pass :- Single pass me convert kar deta h.
→ faster but more space

2) Two-pass :- first convert to intermediate then in second pass convert to m/c code.
→ Slower but less space.

Cousin's of Compiler :-

1) Preprocessor :- It produce input to compiler. It may perform following function :-

(a) Macro-processing :- #define PI 3.14
(b) File inclusion :- #include <stdio.h>
(c) Rational preprocessor :- It augments the older language to the new language by the use of built in macros.
(d) Language extensions :-

2) Assemblers :- Some compilers produce assembly code that is passed to assembler which converts the assembly language to Machine level code.

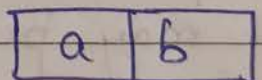
Assembly code is a mnemonic versions of machine code, in which names are used instead of binary codes for operations and names is also given to memory address.

Eg :- MOV a, B1
ADD #2, B1
MOV B1, b

3) Two Pass - Assembly :-

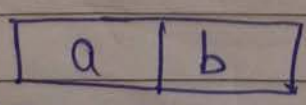
In first pass, all the identifiers that denote storage location are found and stored in symbol table. Basically it checks the code & assign the memory to variable according to requirements.

a 0
b 4



← Assembly Relative addrs.

In Second pass, the assembler scans the input again. This time it translate each operation code into the sequence of bit representing that operation in machine language. The output of this pass is usually relocatable machine code.



← Actual Absolute addrs

```
MOV a, R1          0001 01 00 '00000000*
ADD #2, R1  ->     0011 01 10  00000010
MOV R1, b          0010 01 00  00000100*
```

4) Loader & Link-Editors :-

Usually a program called loader performs the two functions of loading and link-editing. The process of loading consist of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at proper locations.

The link-editor allows us to make a single program from several files ~~from~~ of relocatable machine code.

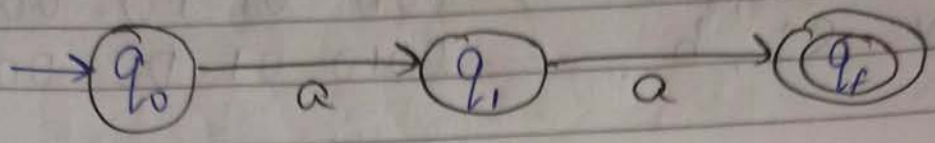
Automata Theory :-

DFA :- 5 tuples :-

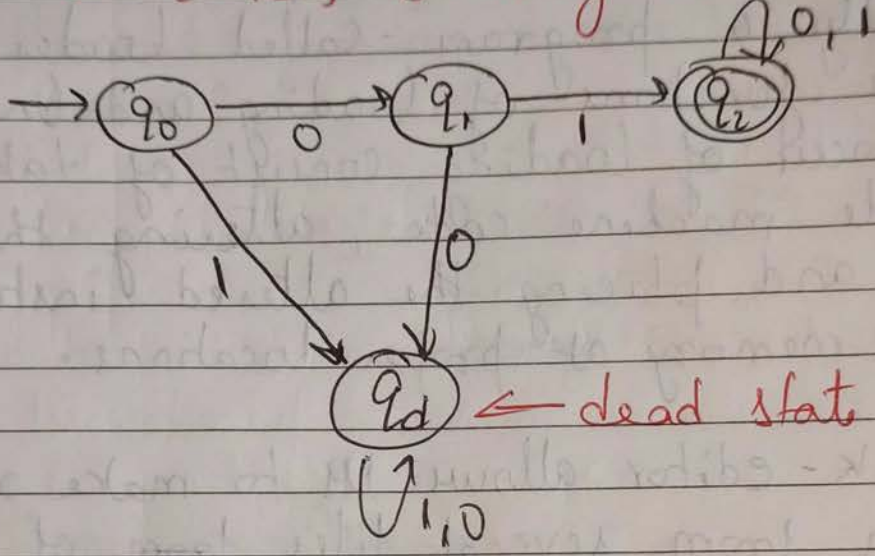
- Q → set of state
- Σ → set of input symbols
- δ → transition functions
- q_0 → Initial State
- F → Final State

$Q \times \Sigma \rightarrow Q$

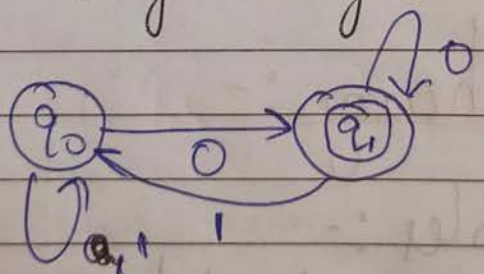
Q aa



Q $\Sigma = \{0, 1\}$ starting with 01.

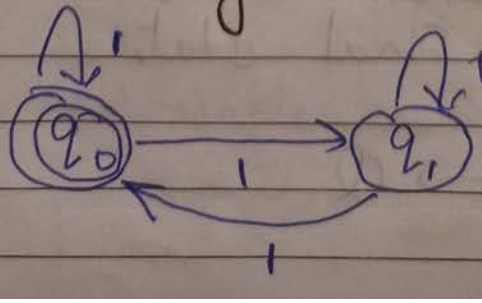


Q All string ending with 0.

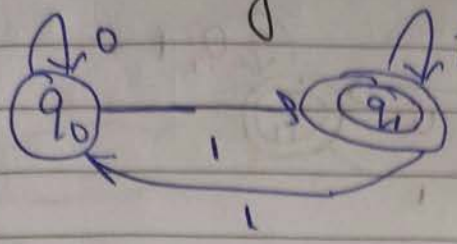


	0	1
q_0	q_1	q_0
q_1	q_1	q_0

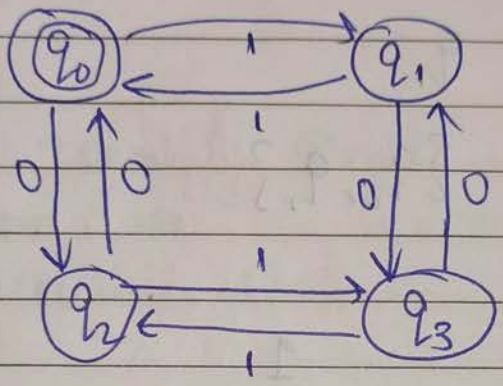
Q Even No. of 1's :-



Q. Odd no. of 1's :-



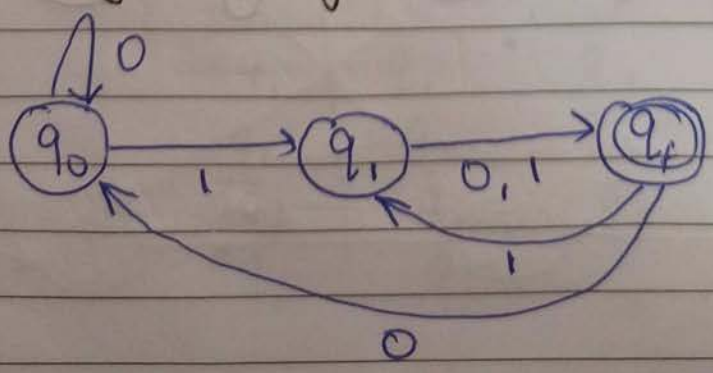
Q Even no. of 1's & even no. of 0's :-



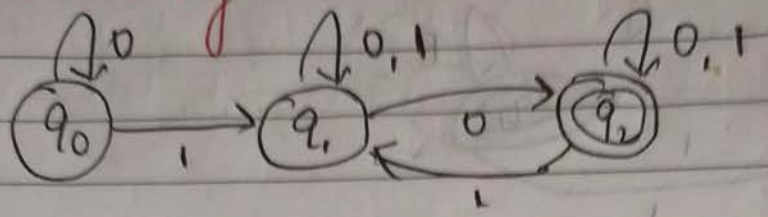
Practice Problems :-

1) Even no. of 1's or even No. of 1's.
2) odd no. of 1's and even no. of 0's.
3) odd no. of 1's or even no of 0.

Q. DFA for string with 2nd last bit as 1.

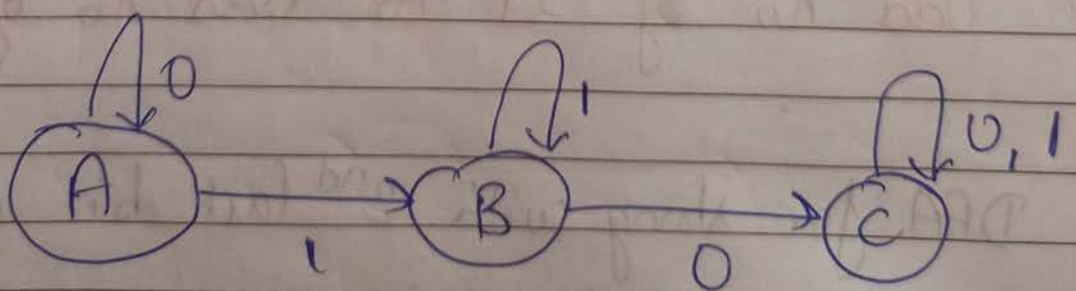


Converting NFA to DFA :-

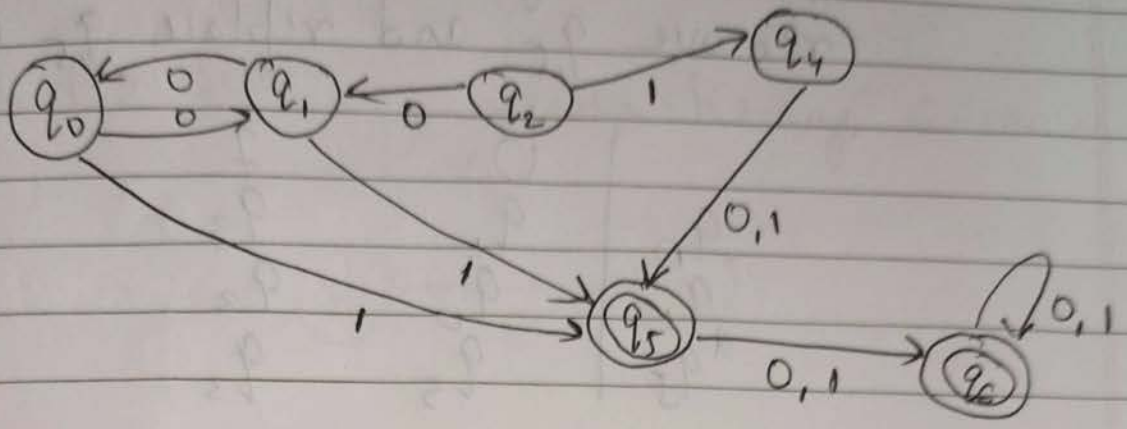


	0	1
q_0	q_0	q_1
q_1	$\{q_1, q_2\}$	q_1
q_2	q_2	$\{q_2, q_1\}$

		0	1
(A)	q_0	q_0	q_1
(B)	q_1	$\{q_1 q_2\}$	q_1
(C)	$q_1 q_2$	$\{q_1 q_2\}$	$\{q_1 q_2\}$



Minimization of DFA :-



Step 1 → Identify Dead state & Inaccessible state
Here, there is no dead state but there is 2 inaccessible state.

Step 2 :- Draw Transition table will all the states except dead & inaccessible one.

	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

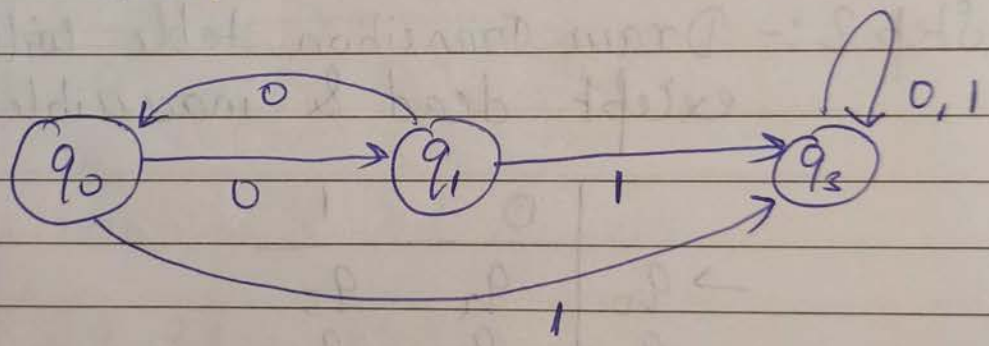
Step 3 :- Creat partition between final stat and all other state.

	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

Step 4 :- Since q_3 and q_4 both have same transition for 0 and 1. So, we can remove q_5 and replace q_5 with q_3.

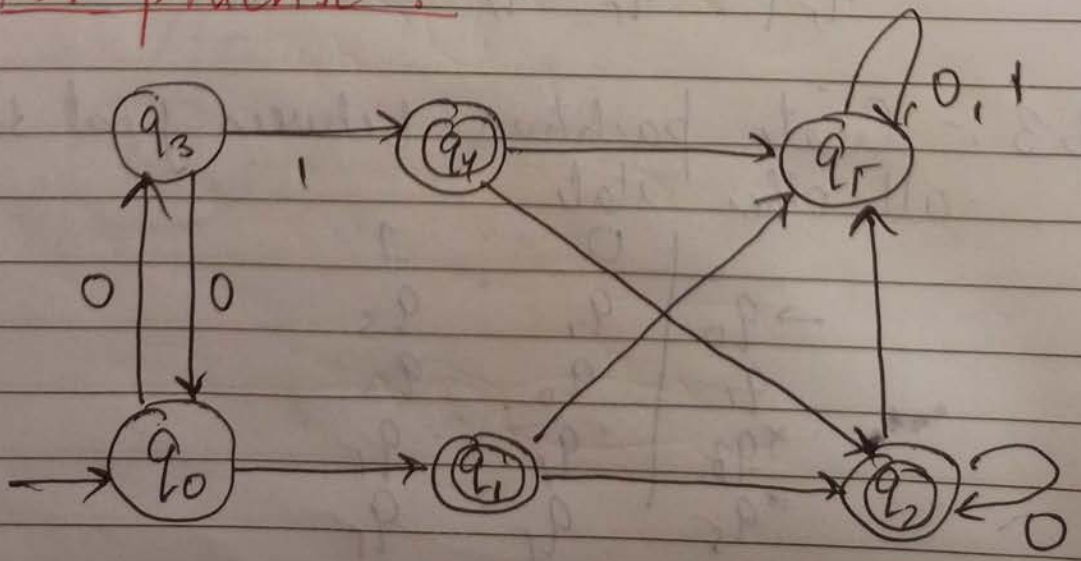
	0	1
q_0	q_1	q_3
q_1	q_0	q_2
*q_2	q_3	q_3

Step 5 :- Now, since there is no other pair of states with same transition, so, we stop and draw DFA for current transition table.



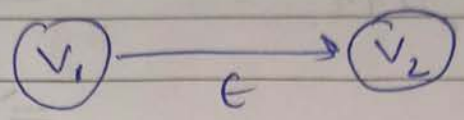
For practise :-

Q.



Converting ϵ-NFA to NFA :-

Step 1 :- Identify vertex having ϵ-move as V_1 and V_2.

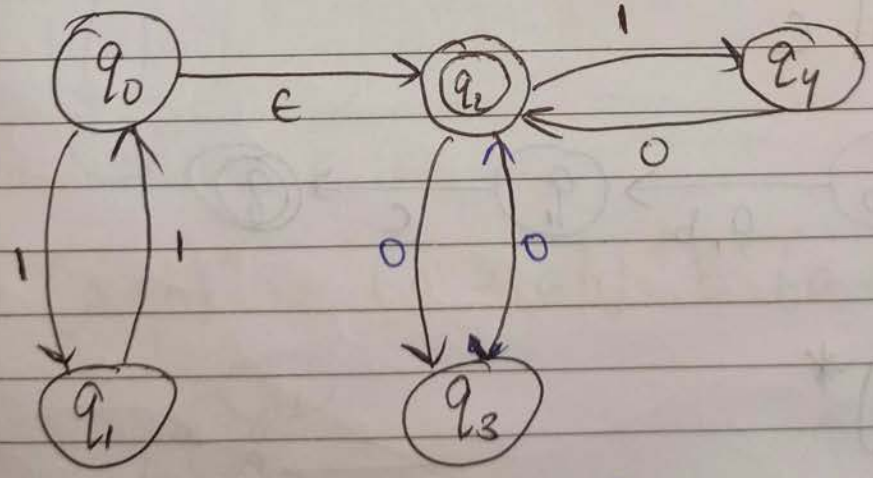


Step 2 :- Copy all the moves of V_2 ~~and~~ to V_1. Delete ϵ-edge.

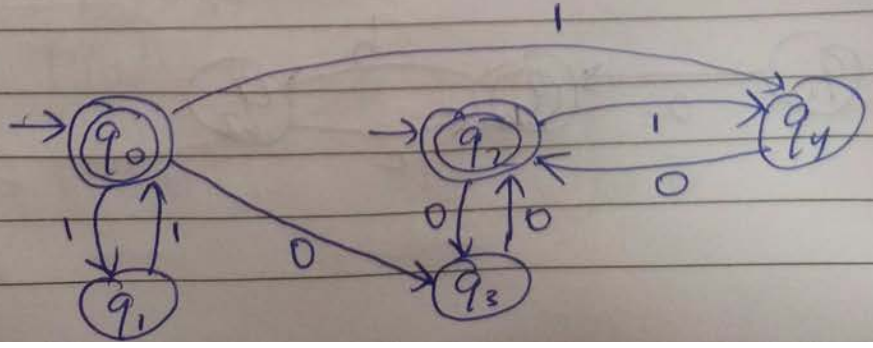
Step 3 :- If V_1 is initial stat make V_2 initial stat

Step 4 :- If V_2 is final stat make V_1 final stat.

Q.



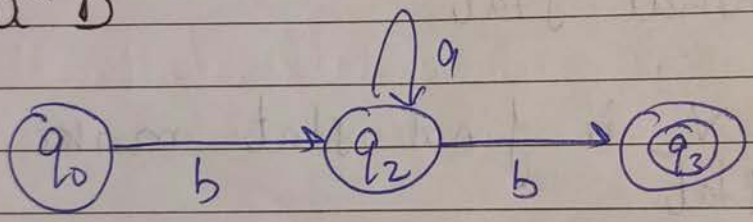
⇒



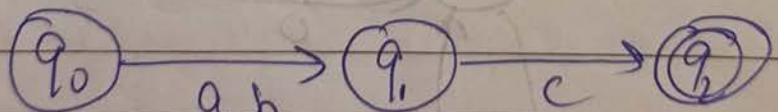
Regular Expression :-

1. $L = \{\epsilon, a, aa, \ldots\} \Rightarrow a^*$
2. $L = \{a, aa, aaa, \ldots\} \Rightarrow a^+$
3. $L = \{\} \Rightarrow \phi$
4. $L = \{\epsilon\} \Rightarrow \epsilon$

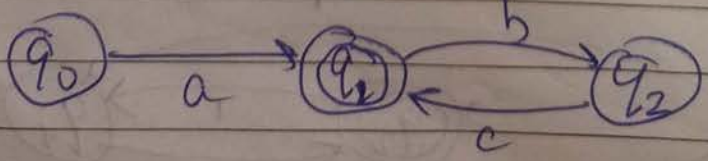
Q. ba^*b



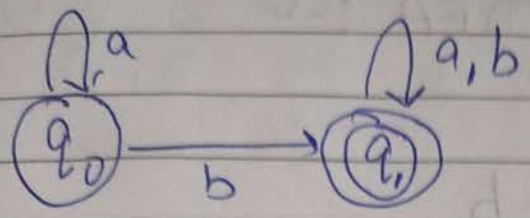
Q. $(a+b)c$



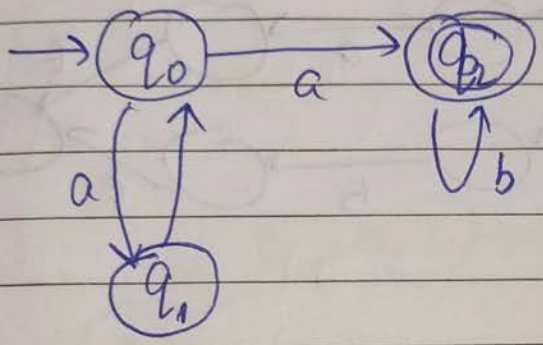
Q. $a(bc)^*$



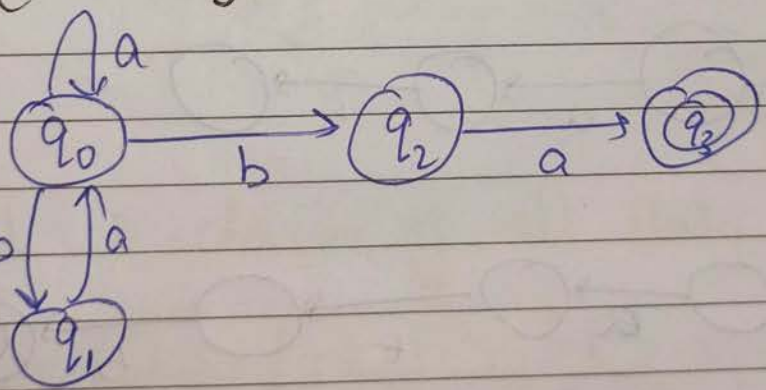
Q. $a^* b (a+b)^*$



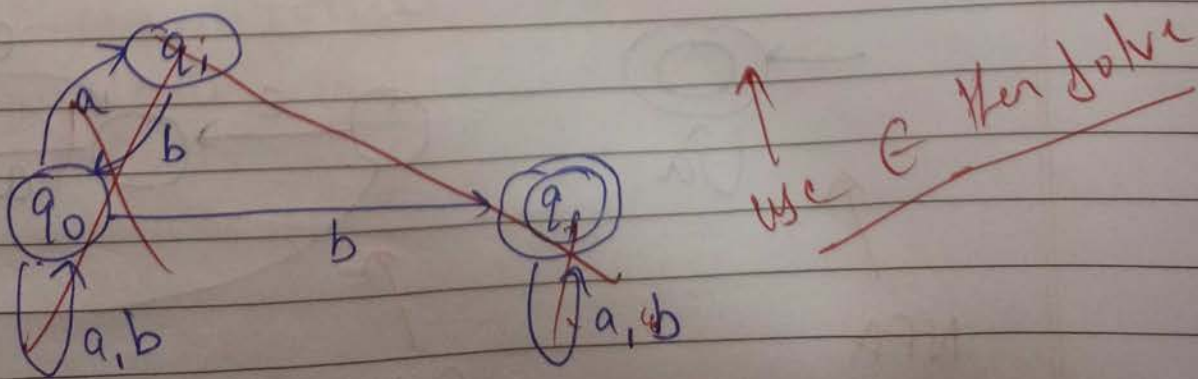
Q. $(ab)^* a b^*$



Q. $(a + ba)^* ba$

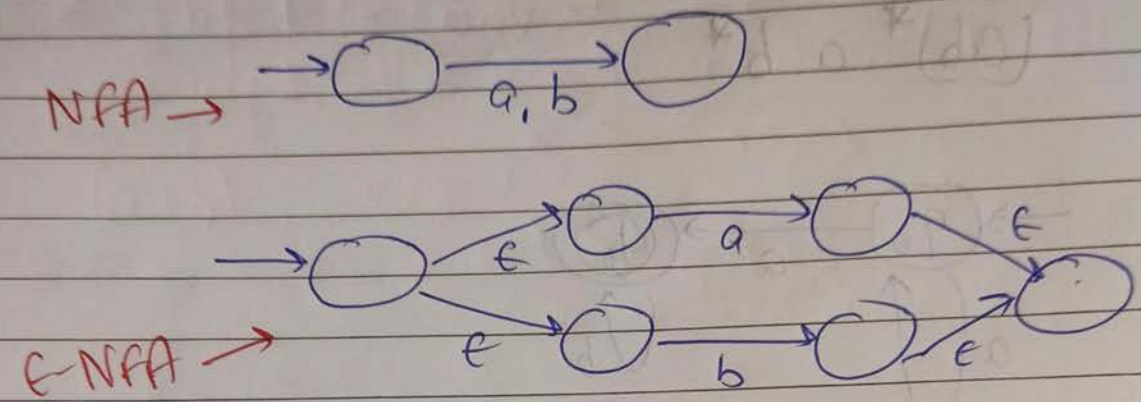


Q. $(a+b)^* + (a + ab)^* b^+ (a+b)^*$

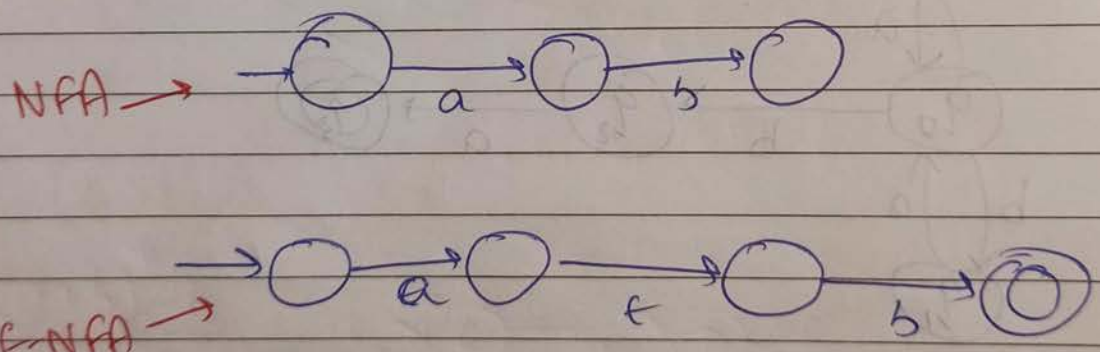


Thompson's Algorithm for Converting RE to DFA :-

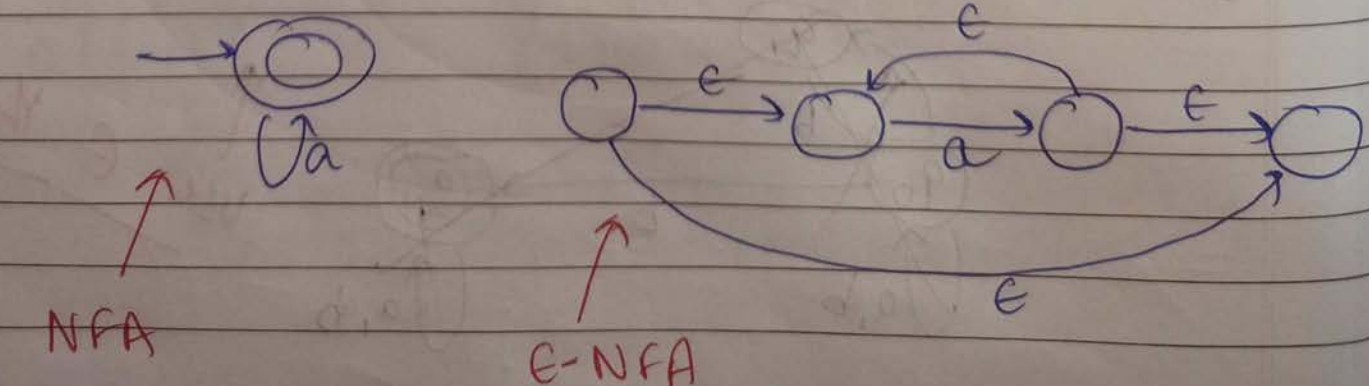
1. Union :- a + b

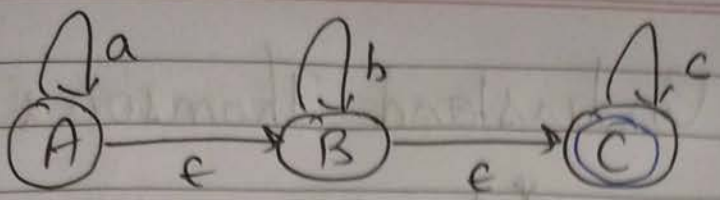


2. Concatenation :- ab



3. Closure :- a*



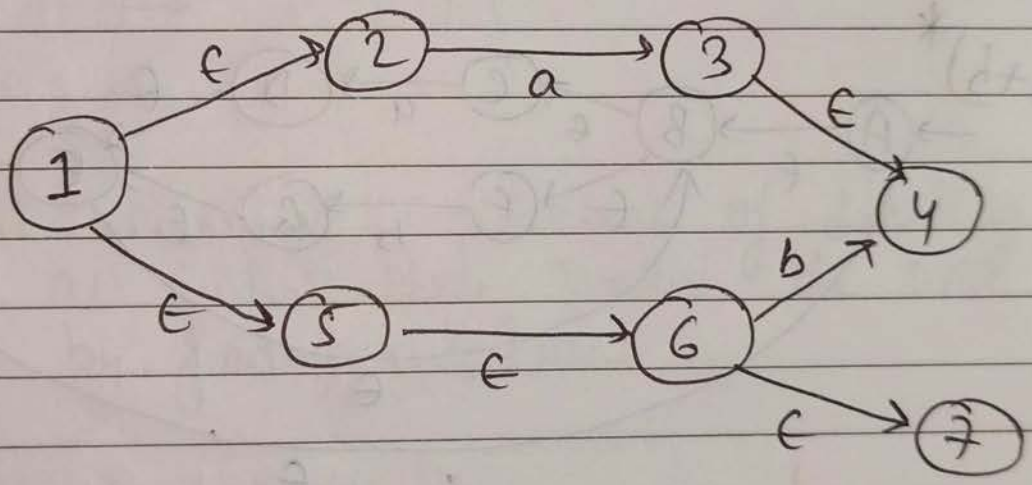


ϵ-closure (A) :- {A, B, C}

ϵ-closure (B) :- {B, C}

ϵ-closure (C) :- {C}

Q.



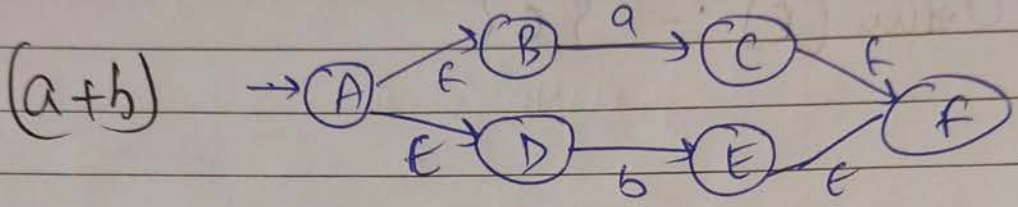
find ϵ-closure of all state

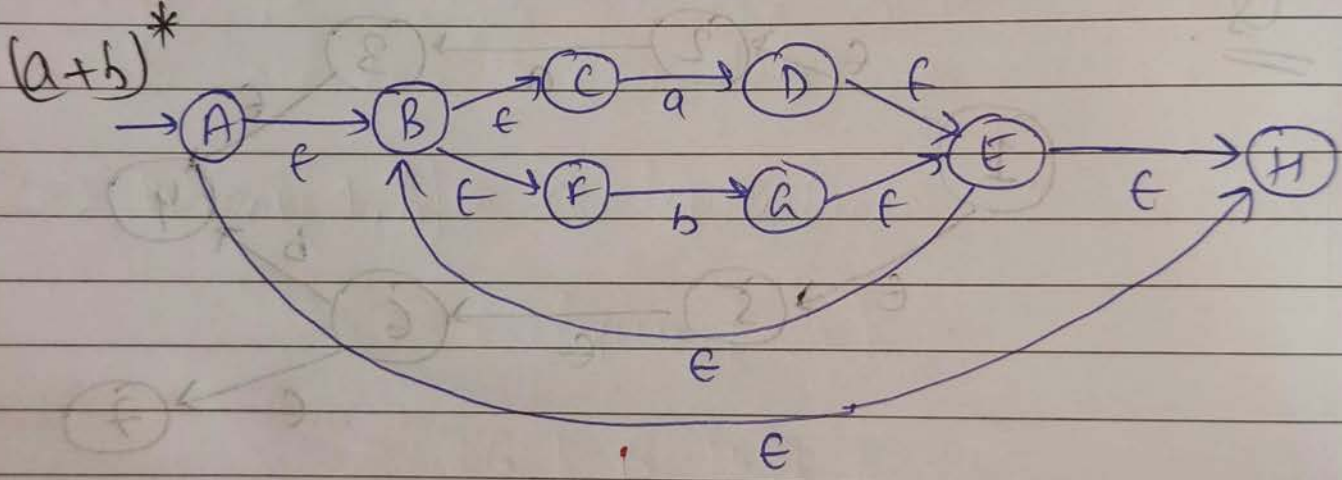
⇒ ϵ-closure (1) :- {1, 2, 5, 6, 7}
ϵ-closure (2) :- {2}
ϵ-closure (3) :- {3, 4}
ϵ-closure (4) :- {4}
ϵ-closure (5) :- {5, 6, 7}
ϵ-closure (6) :- {6, 7}
ϵ-closure (7) :- {7}

Problem to Understand Thomson's Method :-

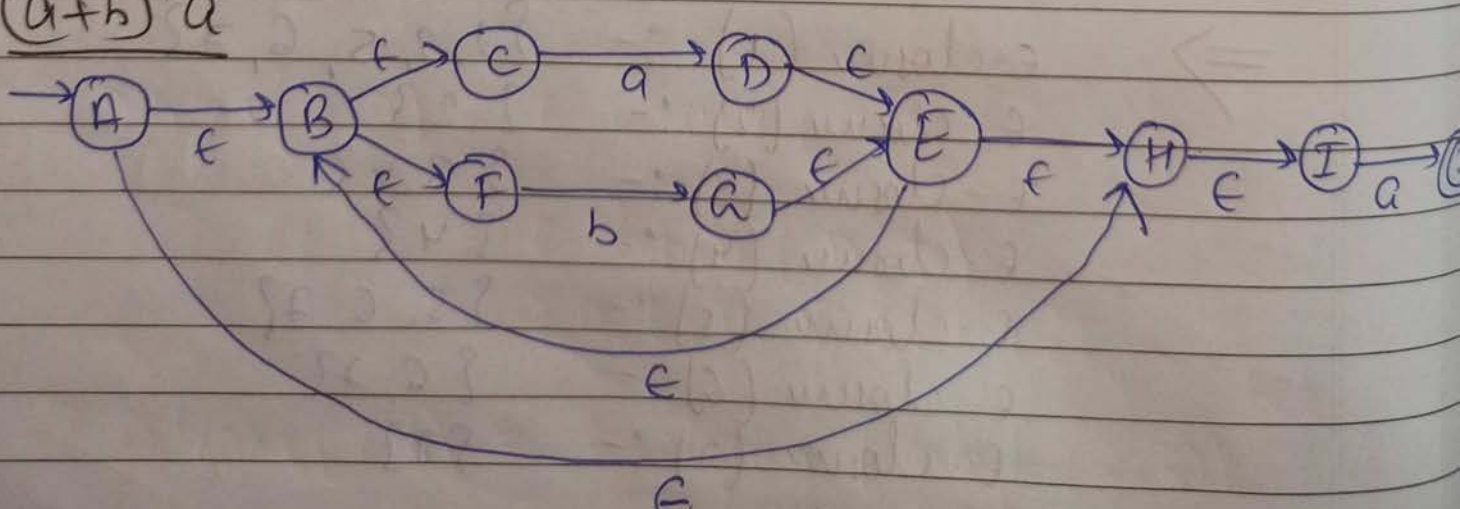
Q Convert $(a+b)^*a$ to DFA.

⇒ Step 1 :- Convert RE to ϵ-NFA





$(a+b)^*a$



Step 2 :- Find ε-closure of all states :-

A → {A, B, C, F, H, I}
B → {B, C, F}
C → {C}
D → {D, E, B, C, F, H, I}
E → {E, H, I, B, C, F}
F → {F}
G → {G, E, B, C, F, H, I}
H → {H, I}
I → {I}
J → {J}

Step 3 :- Taking ε-closure of starting state as initial state Draw the DFA table by finding DTrans.

ε(NFA) State	DFA State	a	b
→ {A, B, C, F, H, I}	1	2	3
{D, E, H, B, I, C, F, J}	2	2	3
{G, E, B, H, I, C, F}	3	2	3

final as it contain J

Formula :-

DTrans(State, input_Symbol) = ε-closure(move(State, input_Symbol))

⇒ Dtrans(1, a) = ε-closure(move(1, a))

= ε-closure(D, J)

= {D, E, H, B, I, C, F, J}

↑ since it is a new state we call it (2)

⇒ Dtrans(1, b) = ε-closure(move(1, b))

= ε-closure(G)

= {G, E, B, H, I, E, F}

↑ since it is a new state we call it (3).

⇒ Dtrans(2, a) = ε-closure(move(2, a))

= ε-closure(D, J)

= {D, E, H, B, I, C, E, J}

⇒ Dtrans(2, b) = ε-closure(move(2, b))

= ε-closure(G)

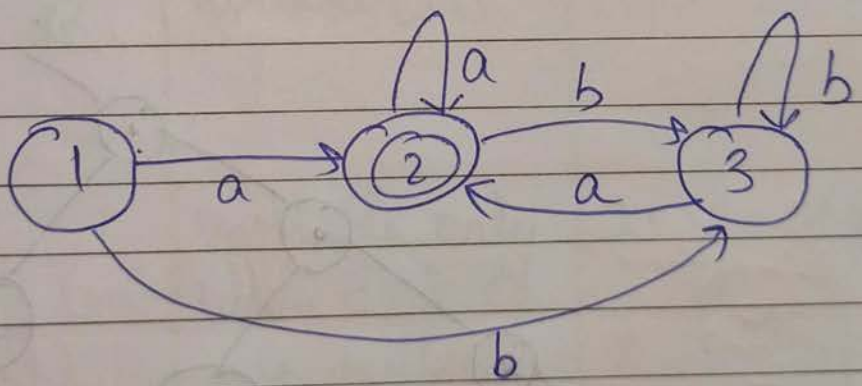
= {G, E, B, H, I, E, F}

$\Rightarrow$ Dtran (3, a) = ϵ-closure(move(3, a))
= ϵ-closure (D, I)
= {D, E, H, B, I, C, F, I}

$\Rightarrow$ Dtran (3, b) = ϵ-closure (move (3, b))
= ϵ-closure (G)
= {G, E, B, H, I, E, F}

↳ Since there is no new state available now, so we stop here.

Step 4 :- Draw DFA from the calculated table.



Question for practise :-

$\longrightarrow$ $(ab)^* + ab$

$\longrightarrow$ $(a + ab)^*$

Direct Method for Converting RE to DFA :-

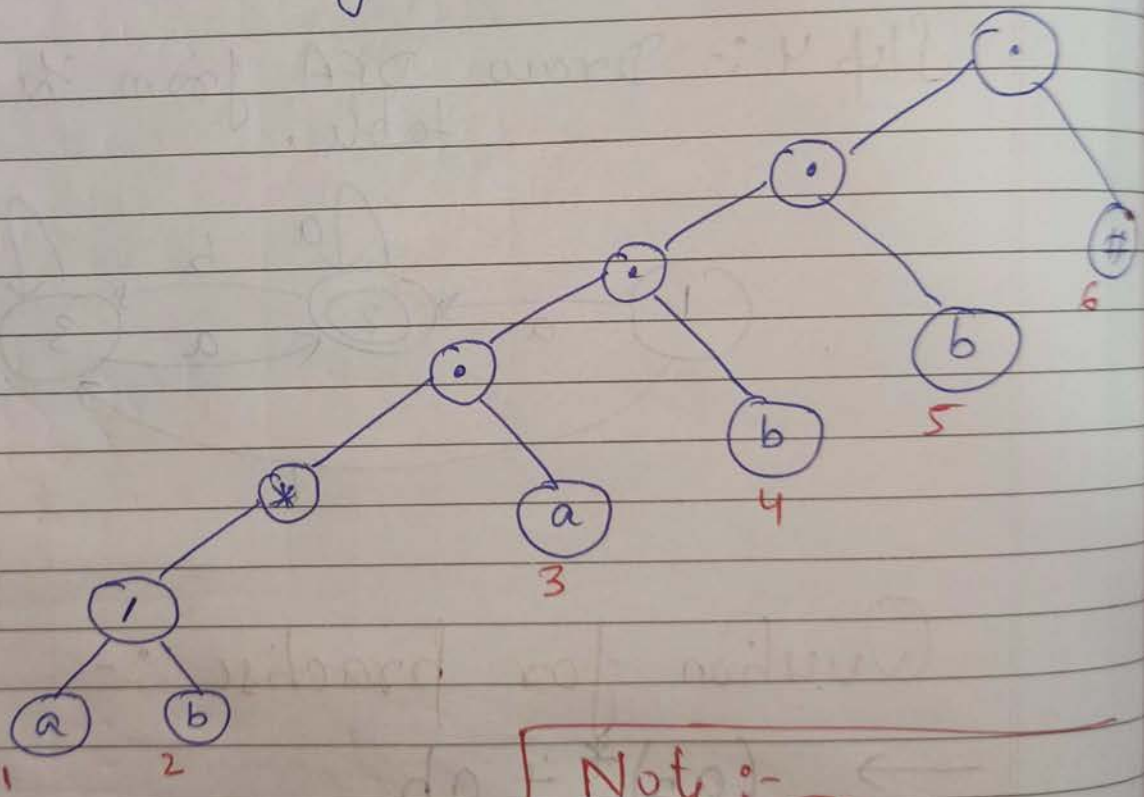
Q $(a/b)^*abb$

Step 1 :- Augment # at the end of regular expression.

$r' = (a/b)^* abb\#$

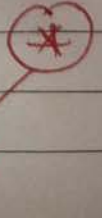
$\Rightarrow (a/b)^* \cdot a \cdot b \cdot b \cdot \#$

Step 2 :- Create Syntax tree for r'.

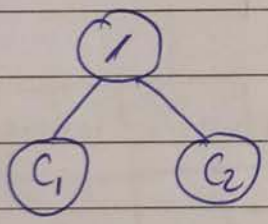
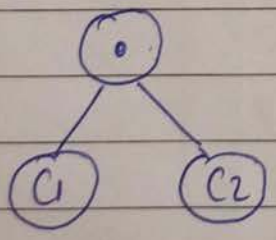
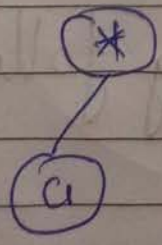


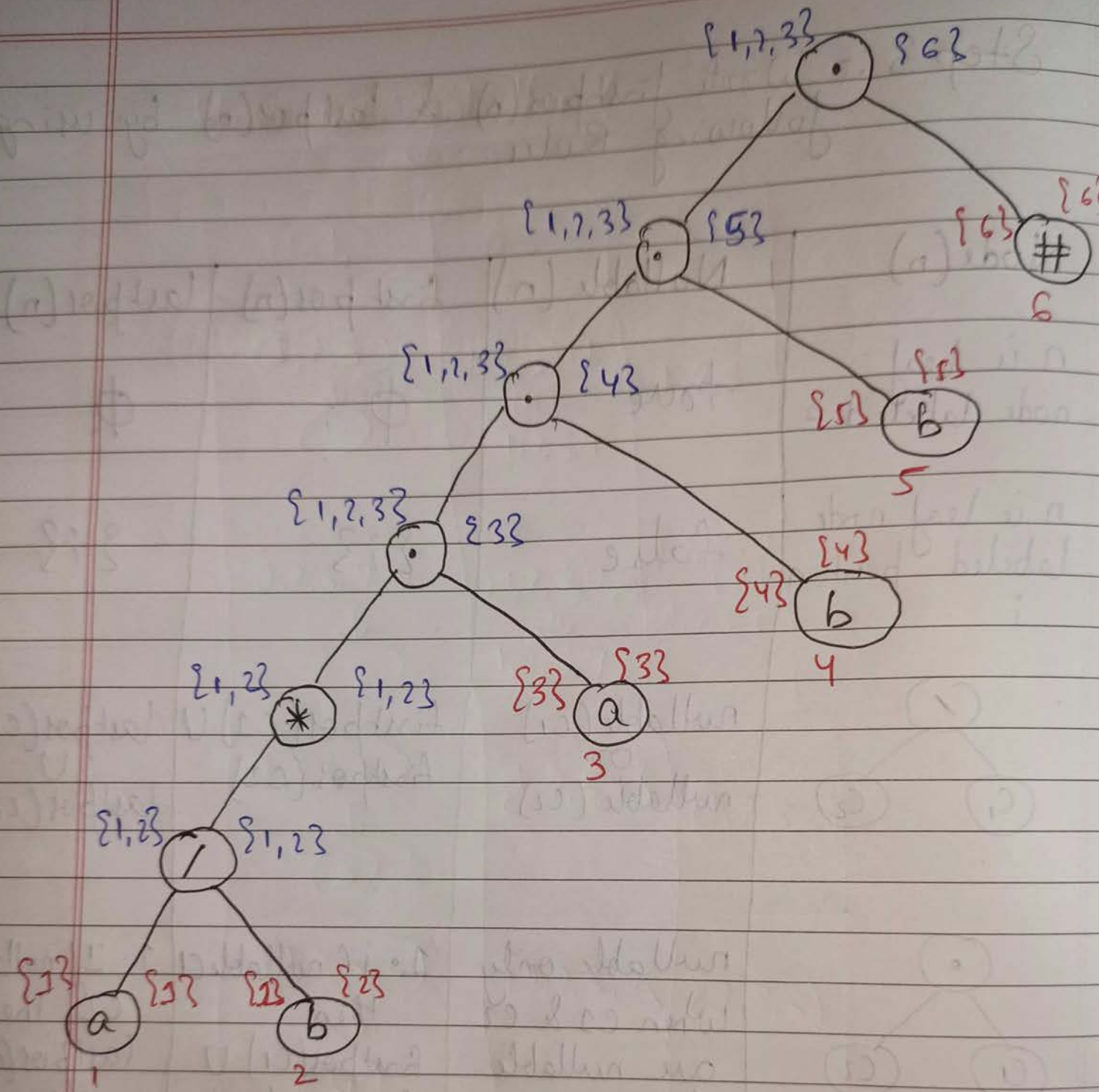
Note :-

(*) ka ek hi child hota h.



Step 3:- Write firstpos(n) & lastpos(n) by using following Rules.

S.No	Node(n)	Nullable (n)	first-pos(n)	lastpos(n)
1.	n is leaf node labeled ε	true	ϕ	ϕ
2.	n is leaf node labeled position i	false	{i}	{i}
3.		nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
4.		nullable only when c1 & c2 are nullable else not nullable	1. if nullable(c1) then firstpos(c1) ∪ firstpos(c2) 2. Else firstpos(c1)	1. If nullable c2 then lastpos(c1) ∪ lastpos(c2) 2. Else lastpos(c2)
5.		true	firstpos(c1)	lastpos(c1)



Step 4 :- Create a followpos(n) table by using Rules of computing followpos.

Rules for Computing followpos :-

1. If (n is a star * node)

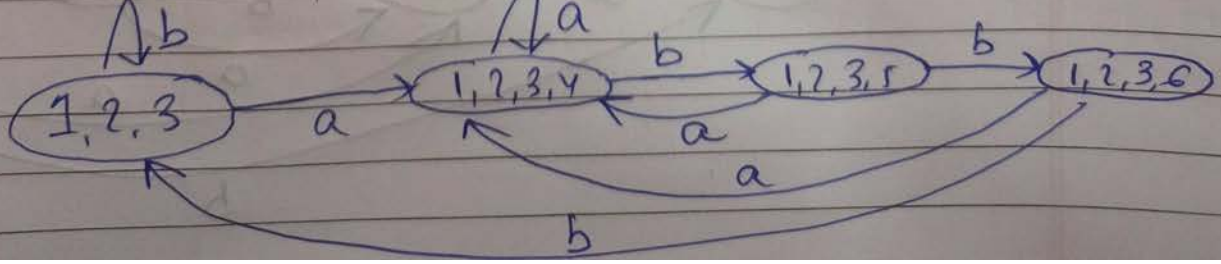
   for (each i in lastpos(n))
   followpos(i) = followpos(i) ∪ firstpos(n);

2. If (n is a cat (·) node)

   for (each i in lastpos(c1))
   followpos(i) = followpos(i) ∪ firstpos(c2)

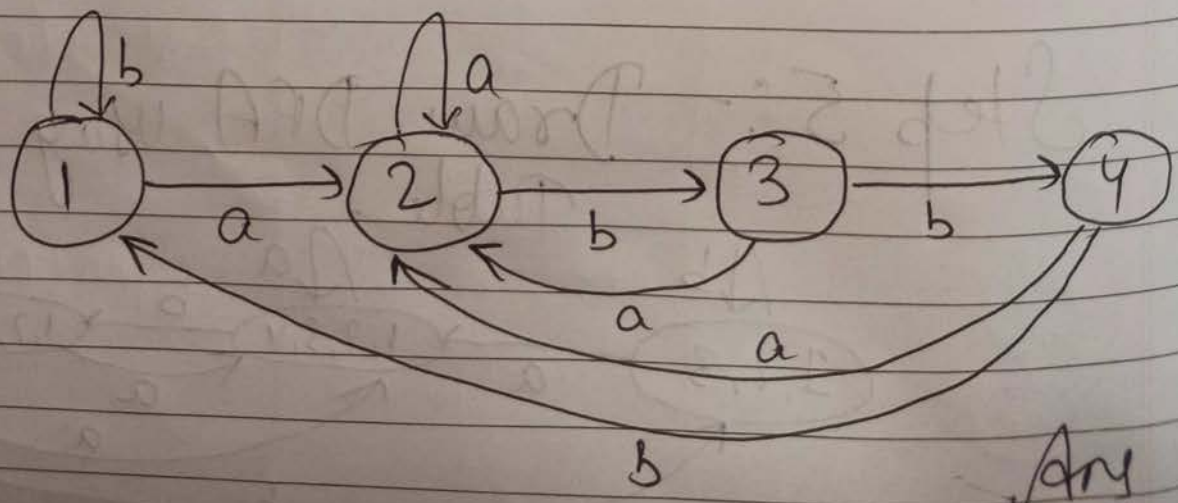
Symbol	Node	followpos
a	1	{1,2,3}
b	2	{1,2,3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	—

Step 5:- Draw DFA using followpos Table.



How we got this ?

State	a	b
1,2,3 → 1	{1,2,3} ∪ {4} = 1,2,3,4	1,2,3
1,2,3,4 → 2	{1,2,3} ∪ {4} → 1,2,3,4	{1,2,3} ∪ {5} → 1,2,3,5
1,2,3,5 → 3	{1,2,3,4}	{1,2,3,6}
1,2,3,6 → 4	{1,2,3,4}	{1,2,3}



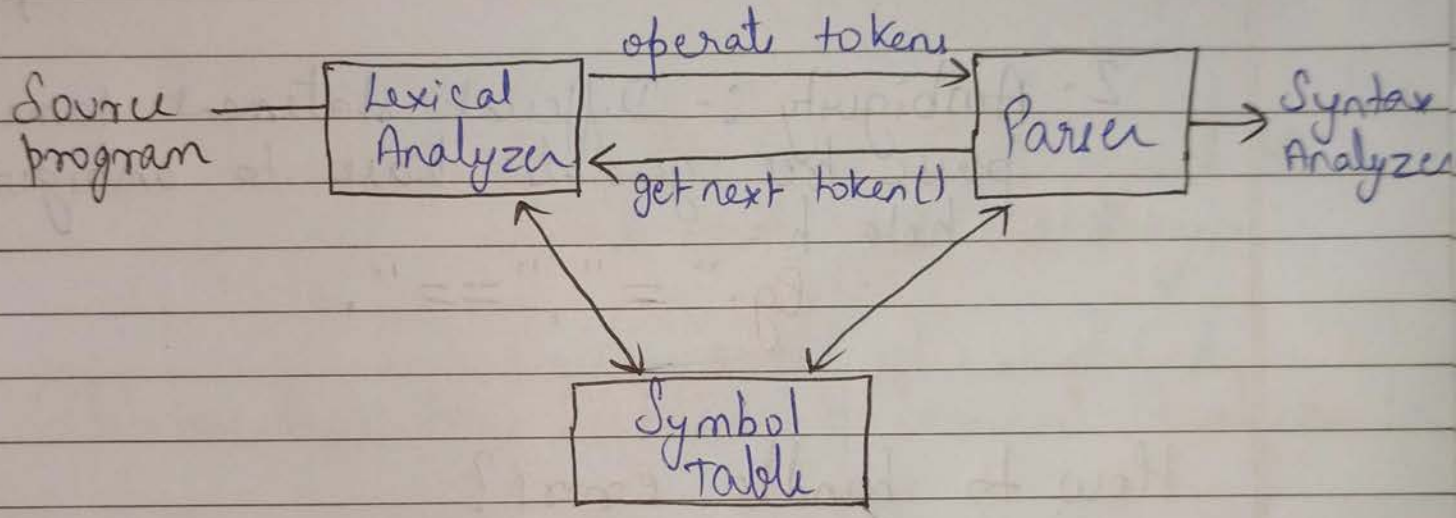
Any

Problem for practise :-

→ $(a+b)^* + (a \cdot c)^*$

→ $(a+b)^* \cdot (a \cdot b)^*$

Lexical Analyser



* Lexical Analyzer does :-

(i) Scanning :-

(ii) Tokenization :-

a. Lexeme (characters)

Eg. int a + b;

→ i, n, t, a, +, b, ;

b. Pattern (rules).

c. Tokens → meaningful words generate karega

int, a, +, b.

* Parser :- Lexical analyzer ko request karta h ki next line ke tokens generate kare.

Why we need lexical analyzer?
→ generate tokens & identify typing error.

Issues in Lexical Analysis?

1. Lookahead : Basically difficulty in deciding where to stop.

Eg. a + b + c = d + e
↑
in this it is not easy to identify where to stop.

2. Ambiguity :- Different statement same ka pause tree generate kare to ambiguity hota h.

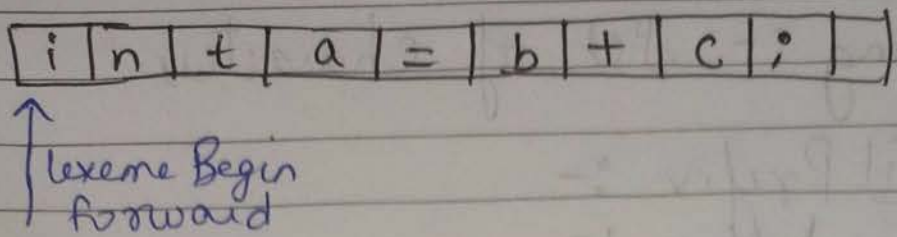
Eg. " = ", " == ".

How to handle errors?

→ ~~Insert~~ Delete one char → intn → delete
→ ~~Delete~~ Insert one char → pint → Insert
→ Replace one char → int → int
→ transpose two character → itn → int
Swap.

Input Buffering :-

Buffer → temporary space where we store data.

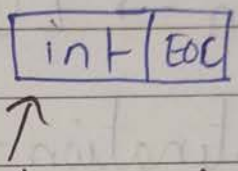


2

2 pointer work on this.

→ Lexeme Begin which identify 1st character of the word

→ forward which moves forward in search of whitespace.



Buffer space which match the word with our directory.

Approaches to Implement Lexical Analyzer :-

1) LEX TOOLS

2) Basic Programming Language

3) Assembly Language

Specification of Tokens :-

1. String
2. Language
3. Regular Expression

1. String :- Set of characters

(i) Prefix :-
(ii) Suffix :-
(iii) Substring :-

2. Language :- $L = \{0, 1\}$, $S = \{a, b, c\}$

(i) Union → $L \cup S = \{0, 1, a, b, c\}$

(ii) Concatenation → $LS = \{0a, 0b, 0c, 1a, 1b, 1c\}$

(iii) Klene closure → L^*

(iv) ~~Klene~~ Positive closure

3. Regular Expression :- used to represent regular language

Q. R.E for second element as 0.
⇒ $(0+1)0(0+1)^*$

Q. RE for first element as 1 & last element as 0
$1(0+1)^*0$

Q. RE for either start with 0 or end with 1.
⇒ $0(0+1)^* + (0+1)^*1$.

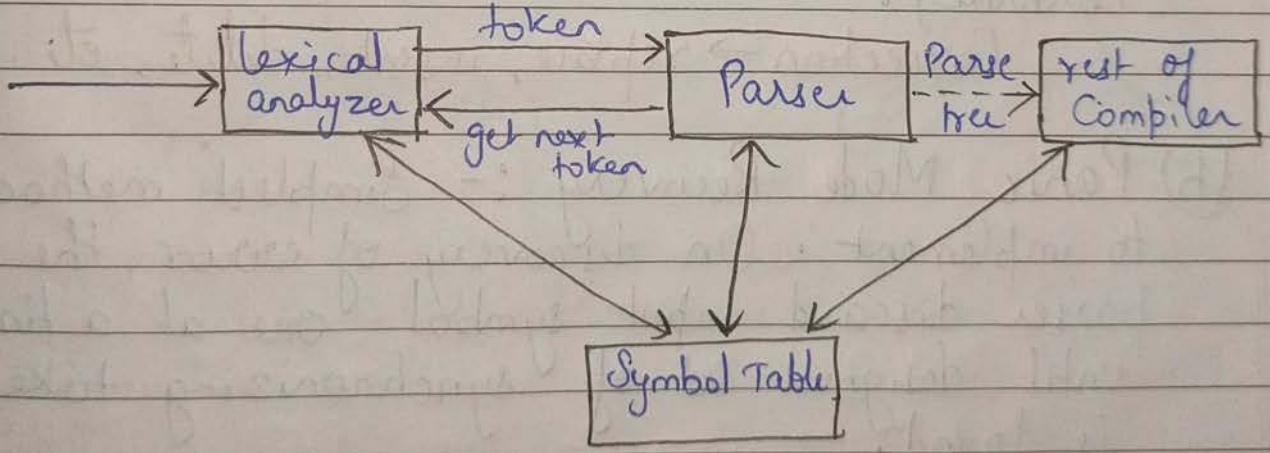
Compiler Design
Unit - 2

Syntax Analysis :- Checking if syntax is correct or not.

The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur form) notation.

* **Role of Parser :-**

The task of parser is to check syntax. The parser obtains a string of tokens from the lexical analyzer and verify that the string can be generated by grammar for the source language.



Syntax Error Handling :-

Types of Errors :-

(i) Lexical → misspelling
(ii) Syntactic → semicolon missing, extra Bracket, Bracket missing
(iii) Logical → infinite loop
(iv) Semantics → operator applied to incompatible operands.

Goals of error handler in a parser :-

→ It should report the presence of error clearly and accurately.
→ It should recover from each error quickly enough to be able to detect subsequent errors.
→ It should not significantly slow down processing of correct program.

* Error Recovery Strategies :-

(a) Phrase level recovery :- On discovery of an error a parser may perform local correction on the remaining input, (i.e it may replace a prefix to some string so that parser will continue).
Correction → replace, insert, delete, etc.

(b) Panic Mode Recovery :- Simplest method to implement. On discovery of error, the parser discard input symbol one at a time until designed set of synchronizing token is found.

(c) Error production → Replace common errors.

(d) Global Correction :- For a given string X with grammar G, we take similar string Y and compare them to find error.

Derivation :-

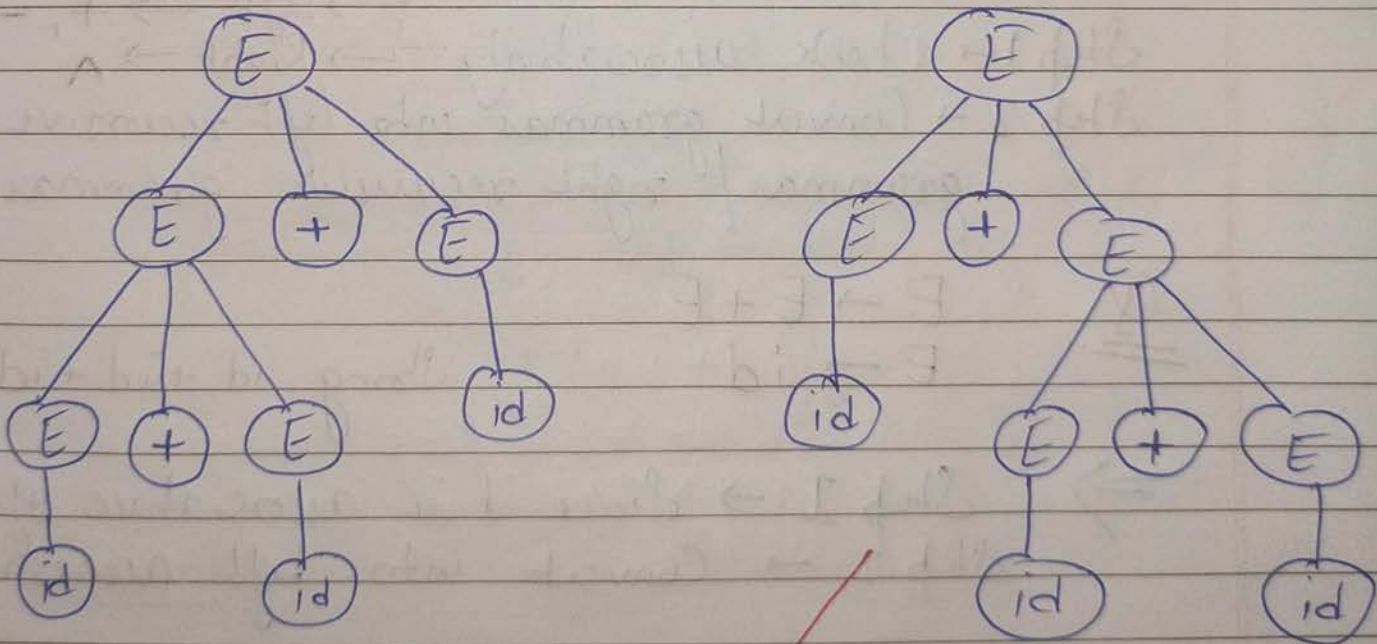
$E \rightarrow E + E$
$E \rightarrow id$

① Left-Most Derivation

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

② Right-Most Derivation

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



2-Different Parse tree
⇒ Ambigous

When such situation of ambigous happens and we have to choose which one to take. We follow following rules :-

(a) Associativity → (when symbols are same)
e.g id + id + id

(b) Precedence → (when symbols are different)
e.g id + id * id

So basically, to remove ambiguity we check for whether ~~or no~~ it is associative or Predence and then convert accordingly.

Step 1 → Check associativity → Left → +, −, *, /
→ Right → ^

Step 2 → Convert grammar into left recursive grammar/ right recursive grammar.

Q. $E \rightarrow E + E$
$E \rightarrow id$

String id + id + id

⇒ Step I → Since it is associative of '+'
Step 2 → Convert into left associative

$E \rightarrow E + E$
$E \rightarrow id$

⇒

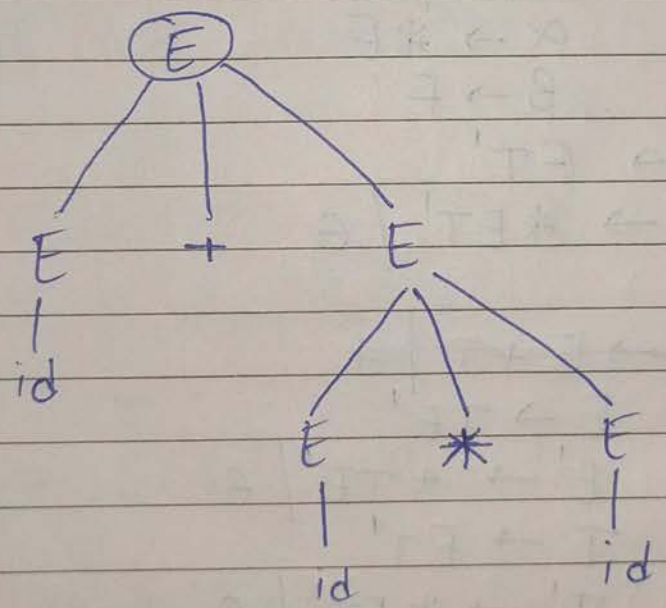
$E \rightarrow E + id$
$E \rightarrow id$

Q $E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$ String id + id * id

⇒ Since string contain different symbol so precedence will follow.

* Precedence of operators :-
1) Higher level will have operator of less priority.
2) Lower level will have operator of high priority.

⇒



Elimination of Left Recursion :-

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

$\Downarrow$

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

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

$\Rightarrow$ $E \to E+T \mid T$
$\alpha \to +T$
$\beta \to T$
$E \to TE'$
$E' \to +TE' \mid \epsilon$

$T \to T*E \mid F$
$\alpha \to *E$
$\beta \to F$
$T \to FT'$
$T' \to *ET' \mid \epsilon$

$\Rightarrow$ ~~$E \to E+T \mid +$~~
$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *ET' \mid \epsilon$
$F \to (E) \mid id$

Q $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \epsilon$

$\Rightarrow$ $S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \epsilon$

$A \to Ac$ | $A \to Aad$

$\alpha \to c$ | $\alpha \to ad$

$\beta \to bd / \epsilon$ | $\beta \to bd / \epsilon$

$A \to bdA' / \epsilon A'$ $\Rightarrow A \to bdA' / A'$

$A' \to adA' / cA' / \epsilon$

Left factoring :- (Common Prefix Problem) :-

$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \gamma$

$\Downarrow$

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 / \beta_3$

Q. $S \to iEtS / iEtSeS / \alpha$

$\Rightarrow$ $\alpha \to iEtS$

$\beta_1 \to \epsilon$

$\beta_2 \to eS$

$\gamma \to \alpha$

$\Rightarrow$ $S \to iEtSS' / \alpha$

$S' \to e / eS$

Q. $S \rightarrow aSSbSS / aSaSb / abb / b$

$\Rightarrow$ $\alpha \rightarrow a$
$\beta_1 \rightarrow SSbSS$
$\beta_2 \rightarrow SaSb$
$\beta_3 \rightarrow bb$
$\gamma \rightarrow b$

$S \rightarrow aS' / b$ — (1)
$S' \rightarrow SSbSS / SaSb / bb$
$\alpha \rightarrow S$
$\beta_1 \rightarrow SbSS$
$\beta_2 \rightarrow aSb$
$\gamma \rightarrow bb$
$S' \rightarrow SS'' / bb$ — (2)
$S'' \rightarrow SbSS / aSb$ — (3)

For Practice :-

Q. $S \rightarrow bSSaaS / bSSaSb / bSb / a$

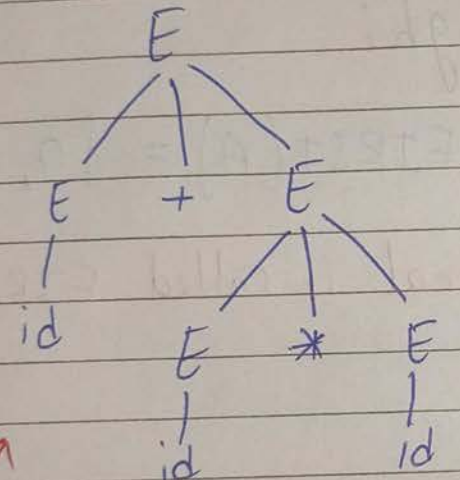
Parsing Methods

(1) Top Down Parsing	(2) Bottom-up Parsing
→ Start from start symbol	→ Just opposite

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

Generate id + id * id using both method

E
→ E + E
→ E + E * E
→ E + E * id
→ E + id * id
→ id + id * id



Top-Down

id + id * id
→ E + id * id
→ E + E * id
→ E * id
→ E * E
→ E

Step 1 → E + id * id
id

Step 2 → E + E * id
id id

Step 3 → E * id
E + E
id id

Step 4 → E * E
E + E id
id id

Step 5 → E
E * E
E + E id
id id

Bottom-up

Top-Down Parser

LL(1) Parser

- L → Left to right
- L → Left most derivation
- 1 → Lookahead symbol

* Computation of FIRST :-

Q. $A \rightarrow abc / def / ghi$

⇒ First of A ⇒ $FIRST(A) = \{a, d, g\}$

* Basically first terminal is called FIRST.

Q
$A \rightarrow BCD$
$B \rightarrow C$
$C \rightarrow d$
$D \rightarrow e / \epsilon$

⇒ $FIRST(A) = FIRST(B) = \{c\}$
$FIRST(B) = \{c\}$
$FIRST(C) = \{d\}$
$FIRST(D) = \{e, \epsilon\}$

Q
$S \rightarrow ABCD$
$A \rightarrow bDC$
$B \rightarrow CD$
$C \rightarrow d$
$D \rightarrow e / ABC / \epsilon$

⇒ FIRST(S) → {b}
FIRST(A) → {b}
FIRST(B) → {d}
FIRST(C) → {d}
FIRST(D) → {e, ϵ, b}

* Computation of FOLLOW :-

Q. A → AbC → Just after A what is there (terminal).

FOLLOW(A) = {b}

Q. A → ABc ⇒ FOLLOW(A) = {d}
B → d FOLLOW(B) = {c}

Q. S → ABCD ⇒ FOLLOW(A) = {c}
A → b FOLLOW(B) = {d}
C → d/e FOLLOW(D) = {$}
D → e FOLLOW(C) = {d e}
B → c FOLLOW(S) = {$}

* Note → Start symbol always have a $

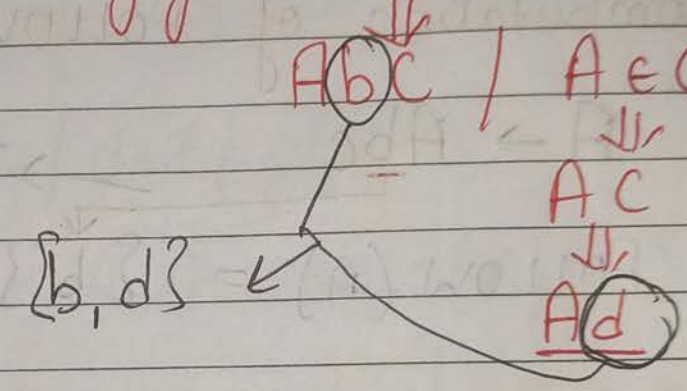
S → ABCD

D end me h ⇒ S ka follow copy hoga.

Q. $S \to ABC$
$A \to a$
$B \to b / \epsilon$
$C \to d$

$\Rightarrow$ { $ }
{ b, ϵ } = { b, d }
{ d }
{ $ }

Now, follow me ϵ nhi hoga
fo, ϵ ka value put karne ke bad
kya ayega uso check karenge.



Practise :-

Q. $S \to ABCD$
$A \to b / \epsilon$
$C \to d / \epsilon$
$D \to e$
$B \to c / \epsilon$

$\Rightarrow$ { $ }
{ c, d, e }
{ e }
{ $ }
{ d, e }

Q. $S \to Bb / Cd$
$B \to aB / \epsilon$
$C \to cC / \epsilon$

LL(1) Parsing Table :-

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

⇒ First of all we make sure there is no left recursion & left factoring in production rule

We already done this question in left recursion

⇒ $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *ET' \mid \epsilon$
$F \to (E) \mid id$

Now, Calculate first and follow.

		FIRST	FOLLOW
$E \to TE'$	⇒	{ (, id }	{ $,) }
$E' \to +TE' \mid \epsilon$	⇒	{ +, ϵ }	{ $,) }
$T \to FT'$	⇒	{ (, id }	{ $,), + }
$T' \to *ET' \mid \epsilon$	⇒	{ *, ϵ }	{ $,), + }
$F \to (E) \mid id$	⇒	{ (, id }	{ +, *,), $ }.

Now Creat Parsing Table

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

Q. $S \to AbAa \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

	FIRST	FOLLOW
$\Rightarrow$ $S \to AbAa \mid BaBb$	{b, a}	{$}
$A \to \epsilon$	{ε}	{b, a}
$B \to \epsilon$	{ε}	{a, b}

	a	b	$
S	$S \to AbAa$	$S \to BaBb$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Note :- If there is 2 production in any block of this Parse tree then LL(1) is not ~~at~~ obtained. (We can't Generate Tree).

So, Question can be asked to check whether it is a LL(1) or not.

Q. $S \to A \mid a$
$A \to a$

$\Rightarrow$ $S \to A/a$ | {a} | $
$A \to a$ | {a} | $

	a	$
S	$S \to A / S \to a$	
A	$A \to a$	

→ 2 Production in one block.
⇒ Not LL(1).

For Practise :-

Q. Check if following grammar is LL(1) or not.

1) $S \to aSbS / bSaS / \epsilon$

2) $S \to aABb$
$A \to c / \epsilon$
$B \to d / \epsilon$

3) $S \to aB / \epsilon$
$B \to bC / \epsilon$
$C \to cS / \epsilon$

4) $S \to aAa / \epsilon$
$A \to abS / \epsilon$.

* Making LL(1) Parsing tree from LL(1) Parsing table.

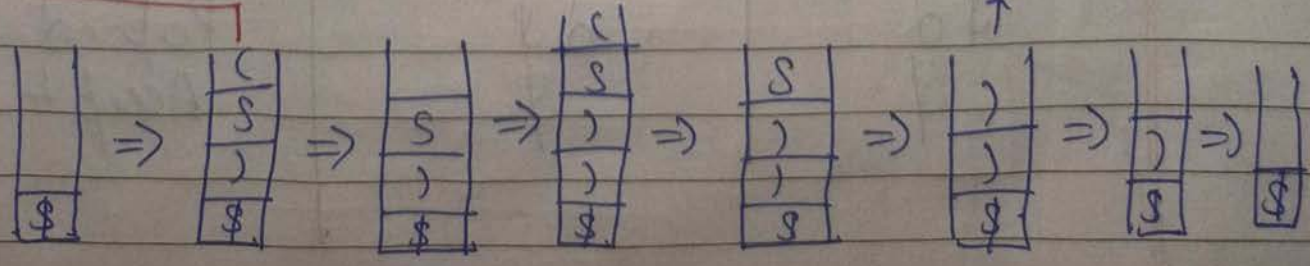
Q. $S \to (S) / \epsilon$ Generate (( ))

Sol. $\Rightarrow$ $S \to (S) / \epsilon$ $\Rightarrow$ $\{(, \epsilon\}$ $\{\$,)\}$

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

Same symbol pe Pop operation

(( ))



How to do in exam?

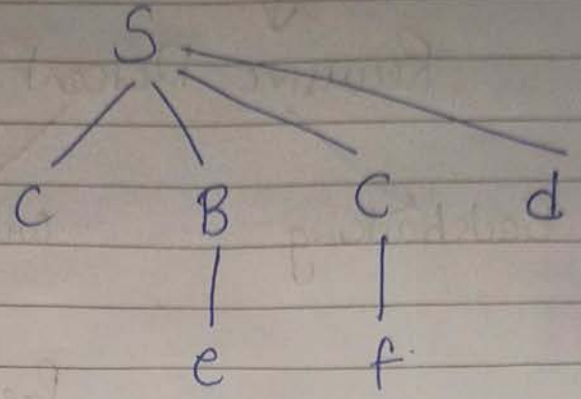
Q. $S \to cBCd$
$B \to e/\epsilon$
$C \to f/\epsilon$ Generate LL(1) tbu for cefd.

Sol → $S \to cBCd$ | $\{c\}$ | $\{\$\}$
$B \to e/\epsilon$ | $\{e, \epsilon\}$ | $\{f, d\}$
$C \to f/\epsilon$ | $\{f, \epsilon\}$ | $\{d\}$

	c	e	f	d	$
S	$S \to cBCd$				
B		$B \to e$	$B \to \epsilon$	$B \to \epsilon$	
C			$C \to f$	$C \to \epsilon$	

Stack	Input	Action
$	cefd $	
$S	cefd $	$S \to cBCd$
$dCBc	cefd $	Pop c
$dCB	efd $	$B \to e$
$dCe	efd $	Pop e
$dC	fd $	$C \to f$
$df	fd $	Pop f
$d	d $	Pop d
$	$	Accept

Now draw tree :-



Question for Practise :-

Q. $E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

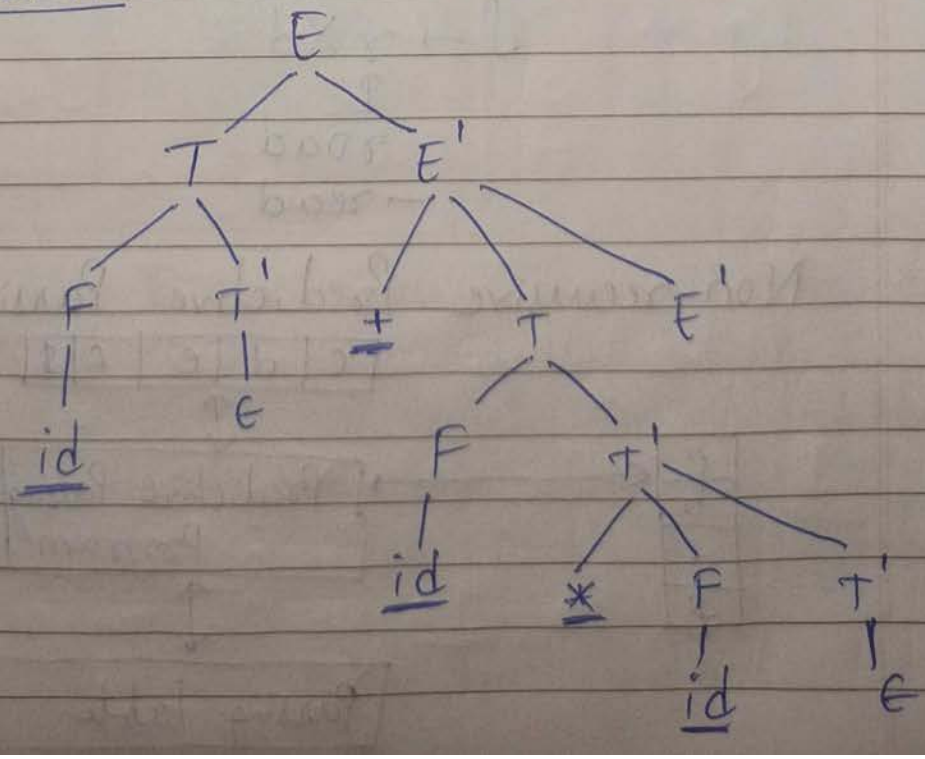
$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

Generate LL(1) tree for id + id * id

Ans to Match :-



Top-Down Parser

- Recursive Descent Parsing
  - Backtracking
  - Without Backtracking
    - Predictive Parser
      - LL(1) Parser
        - If generate error then don't backtrack.

Backtracking :-

$S \rightarrow arXd / arZd$

$X \rightarrow oa / ea$

$Z \rightarrow ai$

Generate string read

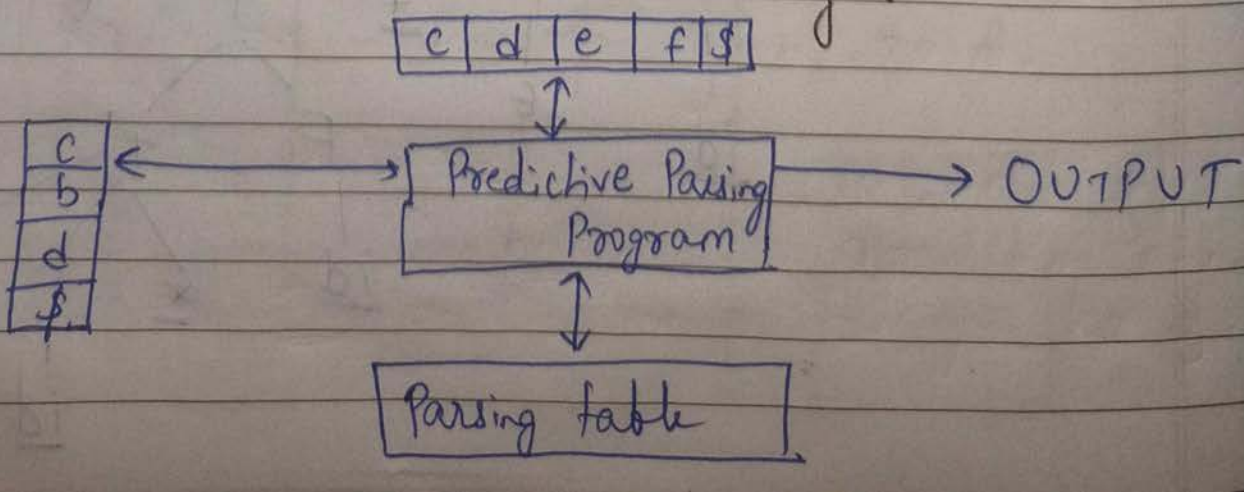
$S \rightarrow arXd$

↑

road

→ read

Non-recursive Predictive Parsing :-



Overview :-

TOP DOWN PARSER
↓ Remove left Recursion & Left factoring
Predictive Parser
↓ Remove Backtracking.
LL(1) Parser

Q. $S \to AbS/e/\epsilon$ ⇒ {e, ϵ, a, c} | {$}
$A \to a/cAd$ {a, c} | {b, d}
String aab$.

	a	b	c	d	e	$
S	S→AbS		S→AbS		A→e	A→ϵ
A	A→a		A→cAd			

Input string	Stack	Action
aab$	$S	S→AbS
aab$	$SbA	A→a
aab$	$Sba	Pop a
ab$	$Sb	Error

↳ Now, If Error occur. Then how to deal.

1) Panic Mode :-
a. If top of stack does not match then pop top element
b. If synch. then pop top element
c. If A→a is empty then skip input symbol.

2) Phrase Mode :-
insert/ replace or delete I/P Symbol.

Let understand how to see it :-

1.a. Ab error aaya tha usi ko continue karte h.

Stack	Input String → Yha aaya tha error. Now, solve	Action
$Sb	ab$	Remove top of Stack
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	Pop a
$Sb	b$	Pop b
$S	$	S → ∈
$	$	Accept String

1.b. Hmlog karte ye h ki S is v ~~follow~~ first ka symbol me ∈ nhi h. Uske me we put Synch for follow.

Eg. →

	a	b	c	d	e	$
S	S→AbS		S→AbS		S→∈	S→∈
A	A→a	Synch	A→c	Synch		

Ye To last ani solve kar rhe the usko A ka follow wale me synch. dalo.

1.c. For A → e, we just skip input Symbol. (i.e agar koi nhi ho 1a ya 1b me se to ye karo).

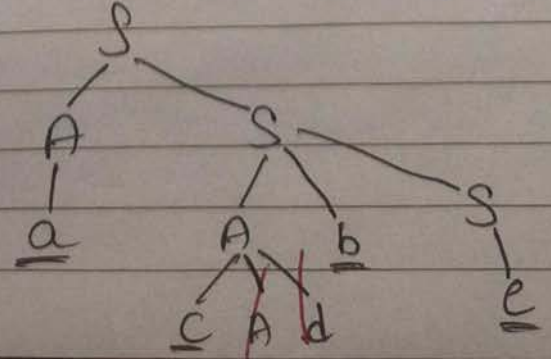
Question → $S \to AbS \mid e \mid \epsilon$
$A \to a \mid cAd$

Generate parse tree for input acbe.

$\Rightarrow$ $S \to AbS \mid e \mid \epsilon$ $\Rightarrow$ {a, c, e, ε} | {$}
$A \to a \mid cAd$ {a, c} | {b, d}.

	a	b	c	d	e	$
S	S→AbS		S→AbS		S→e	S→ε
A	A→a	Synch.	A→cAd	Synch.		

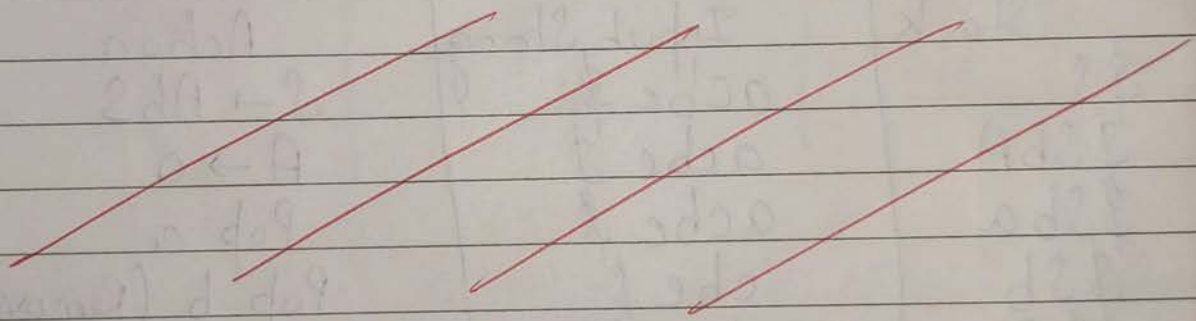
Stack	Input String	Action
$S	acbe$	S → AbS
$SbA	acbe$	A → a
$Sba	acbe$	Pop a
$Sb	cbe$	Pop b (Remove top of stack)
$S	cbe$	S → AbS
$SbA	cbe$	A → cAd
$SbdAc	cbe$	Pop c
$SbdA	be$	Pop A
$Sbd	be$	Pop d
$Sb	be$	Pop b
$S	e$	Pop $ S → e
$e	e$	Pop e
$	$	Accept



For Practise :-

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

Generate Paise tree for)id*+id.



Compiler Design

Unit-3

Bottom-up Parsing :-

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

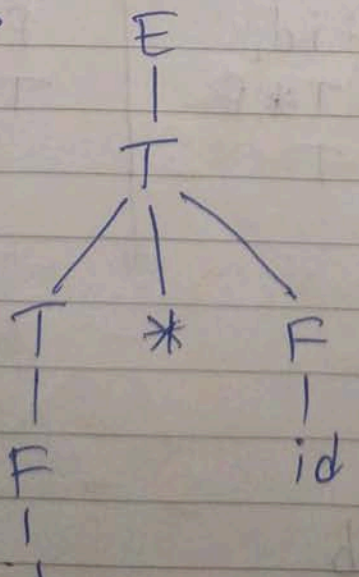
Generate Parse tree for id * id

⇒ Step 1 → F * id
|
id

Step 2 → T * id
|
F
|
id

Step 3 → T * F
| |
F id
|
id

Step 4 → T * F
| |
F id
|
id

Step 5 →


⇒ Reductions :- id * id, F * id, T * id, T * F, T, E

Handle Pruning :-

The handle is the substring that matches the body of a production whose reduction represents one step along with the reverse of the rightmost derivation.

→ Removing the children of left-hand side non-terminal from parse tree is called Handle pruning.

A rightmost derivation in reverse can be obtained by handle pruning.

For the same example we saw in last page, lets see how it works :-

Input String	Handle	Reductions
$id_1 * id_2$	id_1	$F \rightarrow id_1$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id_2$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E (Start)		

↑ output

Q. $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
Input String abbcde

Input String	Handle	Reduction
abbcde	b	A → b
aAbcde	Abc	A → Abc
aAde	d	B → d
aABe	aABe	S → aABe
S (Start)		

↑
accept

Shift Reduce Parsing :-

Q $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$
Input String → $id_1 * id_2$.

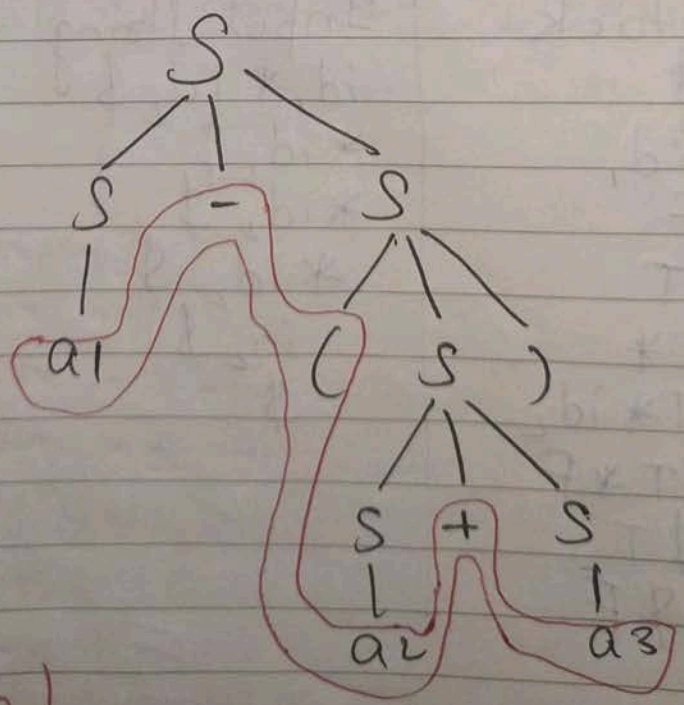
⇒

Stack	Input String	Action
$	$id_1 * id_2 \$$	Shift id_1
$\$ id_1$	$* id_2 \$$	Replace by $F \rightarrow id$
$\$ F$	$* id_2 \$$	Replace by $T \rightarrow F$
$\$ T$	$* id_2 \$$	Shift *
$\$ T*$	$id_2 \$$	Shift id_2
$\$ T* id_2$	$\$$	Replace by $F \rightarrow id$
$\$ T*F$		Replace $T \rightarrow T*F$
$\$ T$		Replace $E \rightarrow T$
$\$ E$		Accept String.

Q. $S \to S+S / S-S$
$S \to (S) / a$ String $\to a_1 - (a_2 + a_3)$

⇒

Stack	Input String	Action
$	$a_1-(a_2+a_3)\$$	Shift a_1
$a_1	$-(a_2+a_3)\$$	Replace $S \to a_1$
$S	$-(a_2+a_3)\$$	Shift −
$S-	$(a_2+a_3)\$$	Shift (
$S-(	$a_2+a_3)\$$	Shift a_2
$S-(a_2	$+a_3)\$$	Replace $S \to a_2$
$S-(S	$+a_3)\$$	Shift +
$S-(S+	$a_3)\$$	Shift a_3
$S-(S+a_3	$)\$$	Replace $S \to a_3$
$S-(S+S	$)\$$	Replace $S \to S+S$
$S-(S	$)\$$	Shift)
$S-(S)	$\$$	Replace $S \to (S)$
$S-S	$\$$	Replace $S \to S-S$
$S	$\$$	Accept string.



$a_1-(a_2+a_3)$

LR(0) Parsing

Left to right | Rightmost Symbol | 0 lookahead

Q Create LR(0) Parsing table for

$E \to T + E / T$

$T \to id$

⇒ Step 1 → Augment starting Symbol.

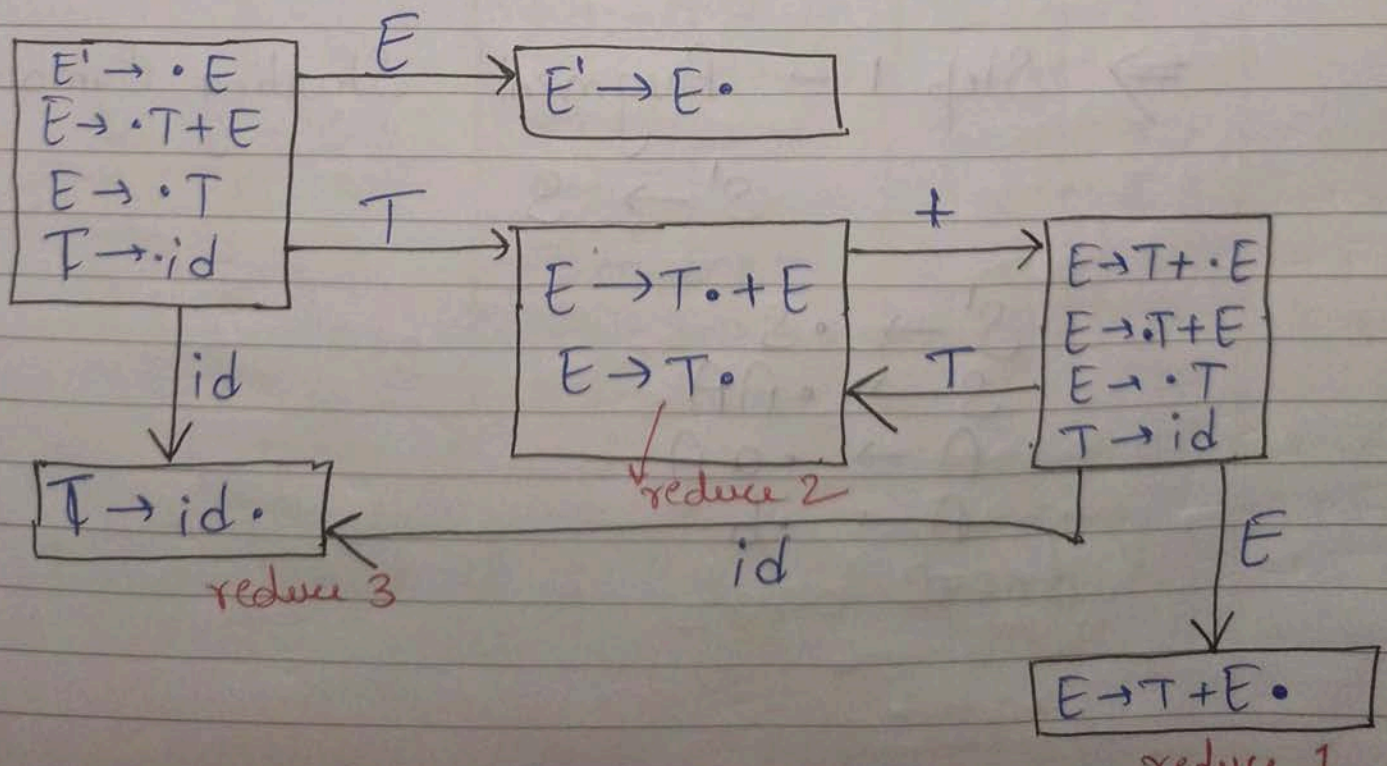
$E' \to E$

$E' \to \cdot E$

$E \to \cdot T + E$

$E \to \cdot T$

$T \to \cdot id$



States	Actions			Go to	
	+	id	$	E	T
I0		S3		1	2
I1			accept		
I2	S4/r2	r2	r2		
I3	r3	r3	r3		
I4		S3		5	2
I5	r1	r1	r1		

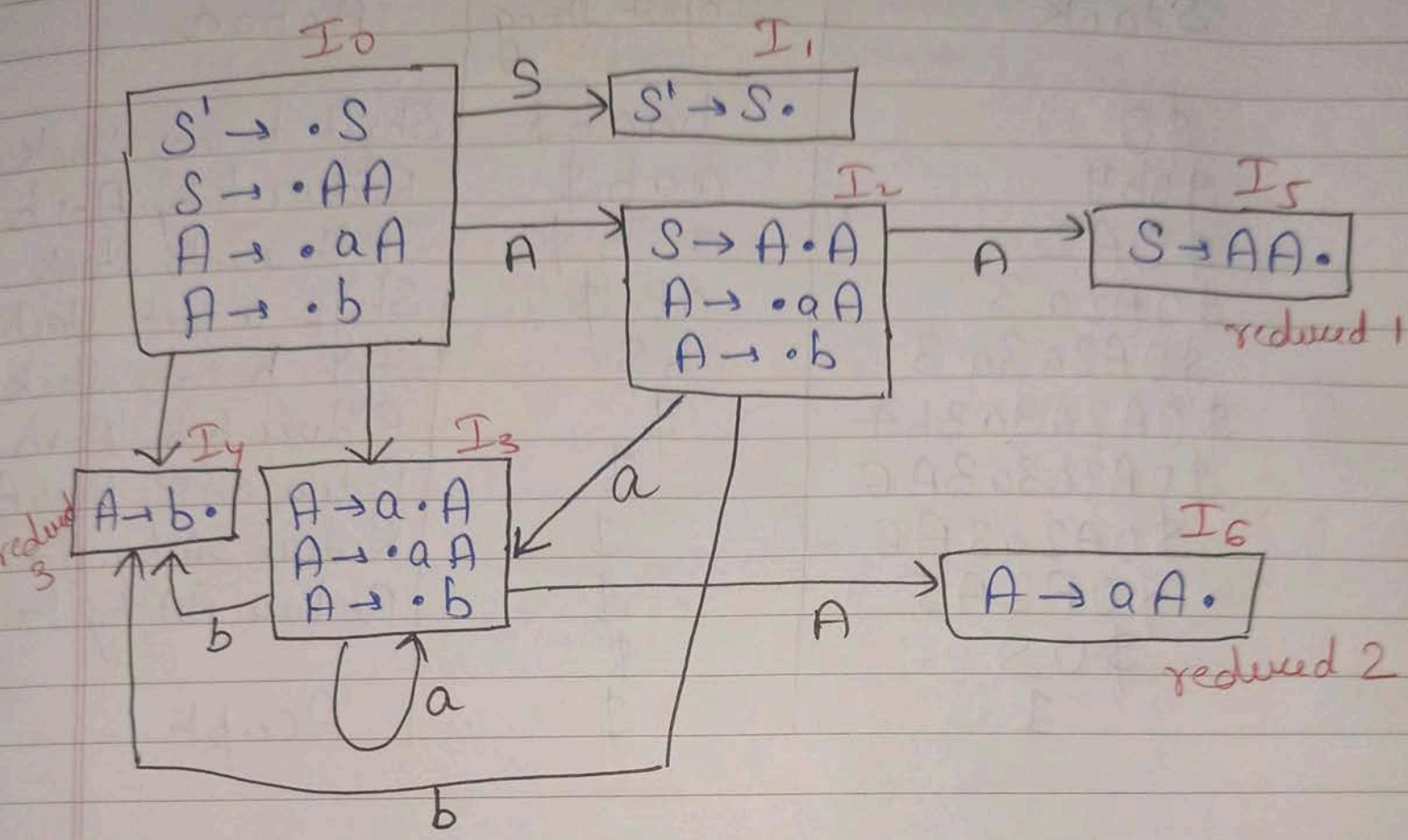
For terminals only, we use
1) Shift (S)
2) Reduce (r)

Q. $S \rightarrow AA$
$A \rightarrow aA / b$

⇒ Step 1 → Augment Starting Symbol

$S' \rightarrow S$

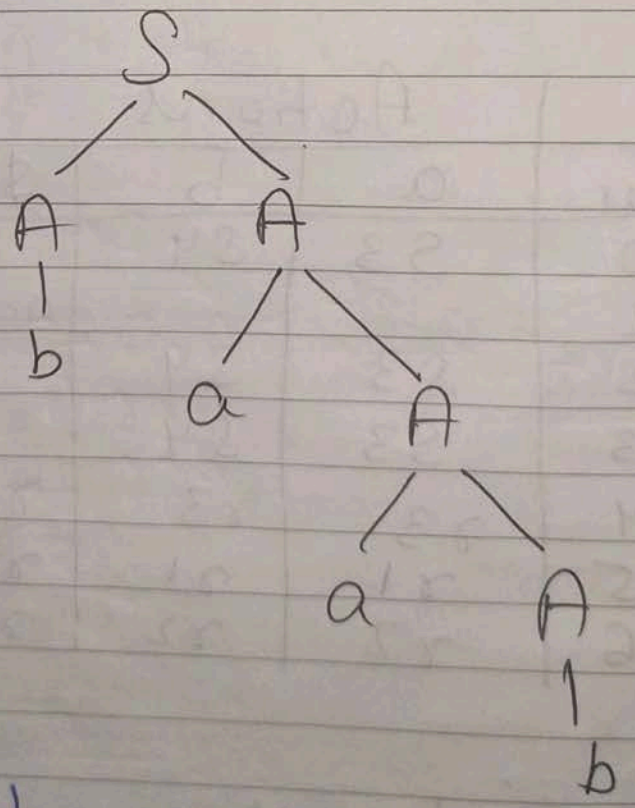
$S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$



	Actions				
States	a	b	$	S	A
I0	S3	S4		1	2
I1			accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

Now, check if it accept baab ?

Stack	Input String	Action
$0	baab$	Shift b into Stack
$0b4	aab$	Reduce b by A→b
$0A2	aab$	Shift a into Stack
$0A2a3	ab$	Shift a into Stack
$0A2a3a3	b$	Shift b into Stack
$0A2a3a3b4	$	Reduce b by A→b
$0A2a3a3A6	$	Reduce aA by A→aA
$0A2a3A6	$	Reduce aA by A→aA
$0A2A5	$	Reduce AA by S→AA
$0S	$	
1	$	Acept



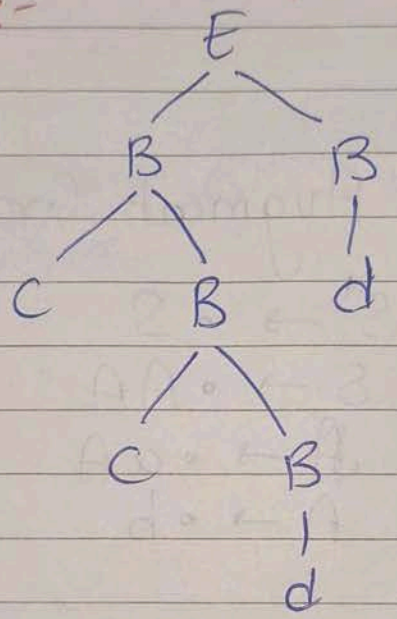
⇒ baab

Question to Practise :-

1) $E \to BB$
$B \to cB / d$
Generate string ccdd.

↳ It is similar to last quest.

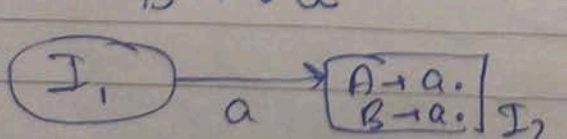
Ans to match :-



Note → While using LR(0), there are 2 types of Conflict that may arise.

1) RR Conflict

$A \to .a$
$B \to .a$

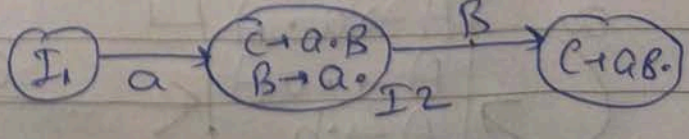


	a	A	B
I_1	S2		
I_2	r1/r2		

↑ 2 reduce (RR)

2) SR Conflict

$C \to .aB$
$B \to .a$



	a	B
I_1	S2	
I_2	S3/r2	S3

↳ Shift Reduce (SR)

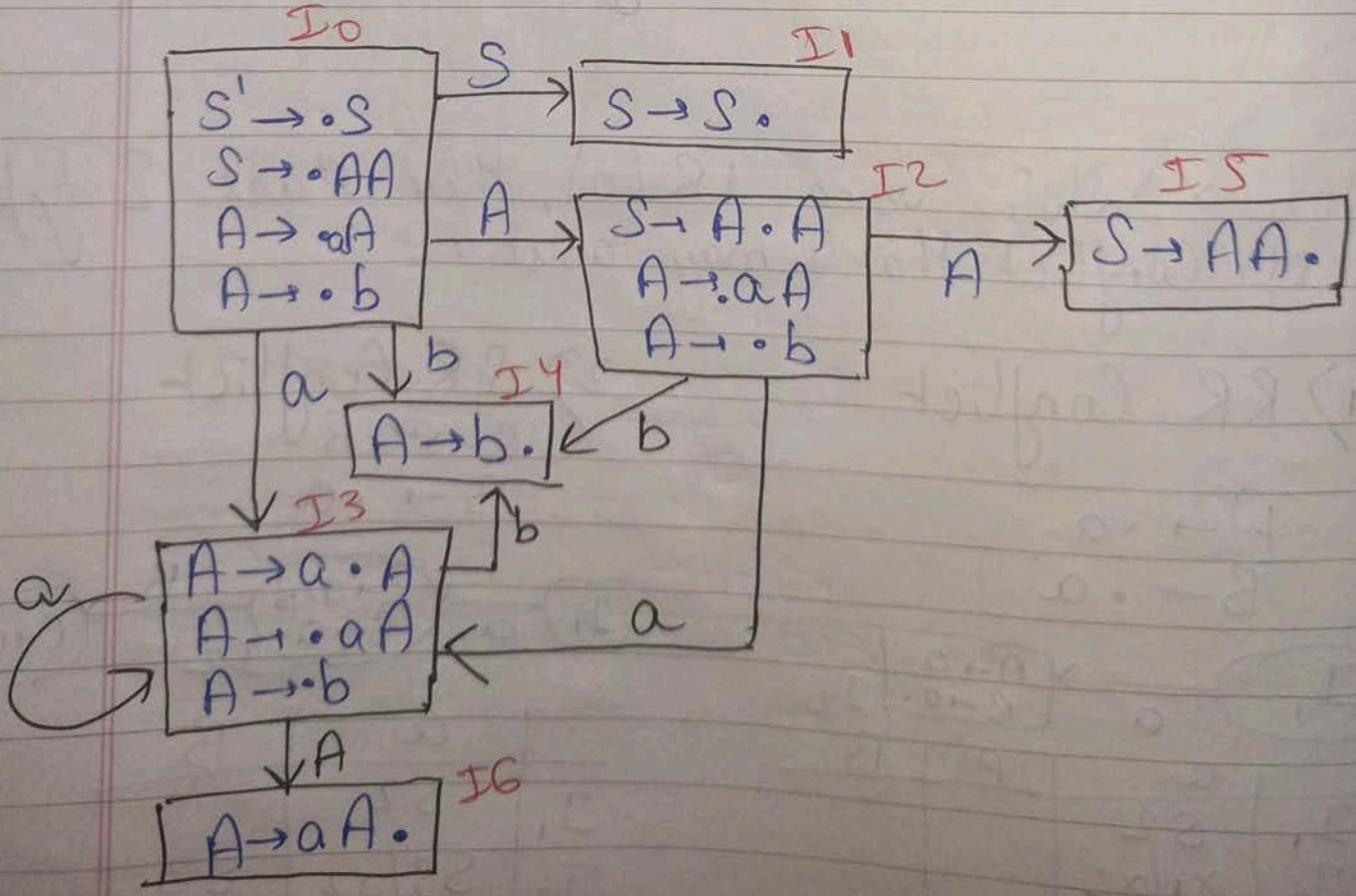
To overcome these conflicts, we use SLR parser (Simple LR Parser).

SLR(1) Parser

Q. $S \to AA$
$A \to aA / b$

⇒ Step 1 → Augment Grammar.

$S' \to \cdot S$
$S \to \cdot AA$
$A \to \cdot aA$
$A \to \cdot b$



Step 2 → Calculate follow of S and A.

FOLLOW(S) = {$}
FOLLOW(A) = {a, b, $}

	Action			Goto	
Stage	a	b	$	S	A
I0	S3	S4		1	2
I1			acept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5			r1		
I6	r2	r2	r2		

Q. E → T + E / T
T → id

⇒ Follow(E) = {$}
Follow(T) = {+, $}

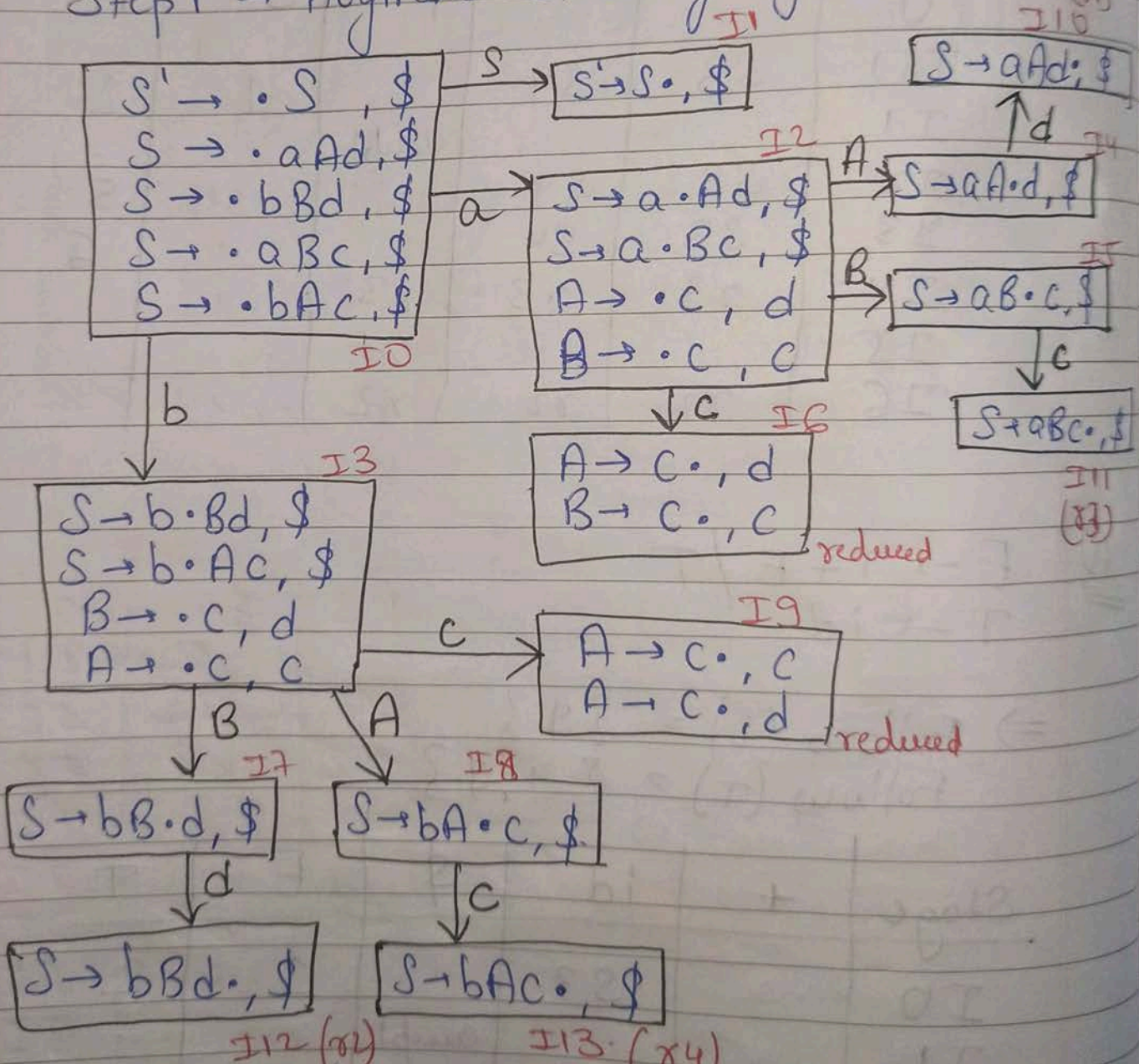
Stage	+	id	$	E	T
I0		S3		1	2
I1			acept		
I2	S4		r2		
I3	r3		r3		
I4		S3		5	2
I5			r1		

no Conflict this time using SLR.

Canonical Left Right Parser (CLR(1)) :-

Q. $S \to aAd / bBd / aBc / bAc$
$A \to c$
$B \to c$

⇒ Step 1 → Augment starting Symbol.



Stage	a	b	c	d	$	S	A	B
I0	S2	S3				1		
I1					accept			
I2			S6				4	5
I3			S9				8	7
I4				S10				
I5			S11					
I6			r6	r5				
I7				S12				
I8			S13					
I9			r5	r6				
I10				r1				
I11				r3				
I12				r2				
I13				r4				

Look-ahead left-Right (LALR)

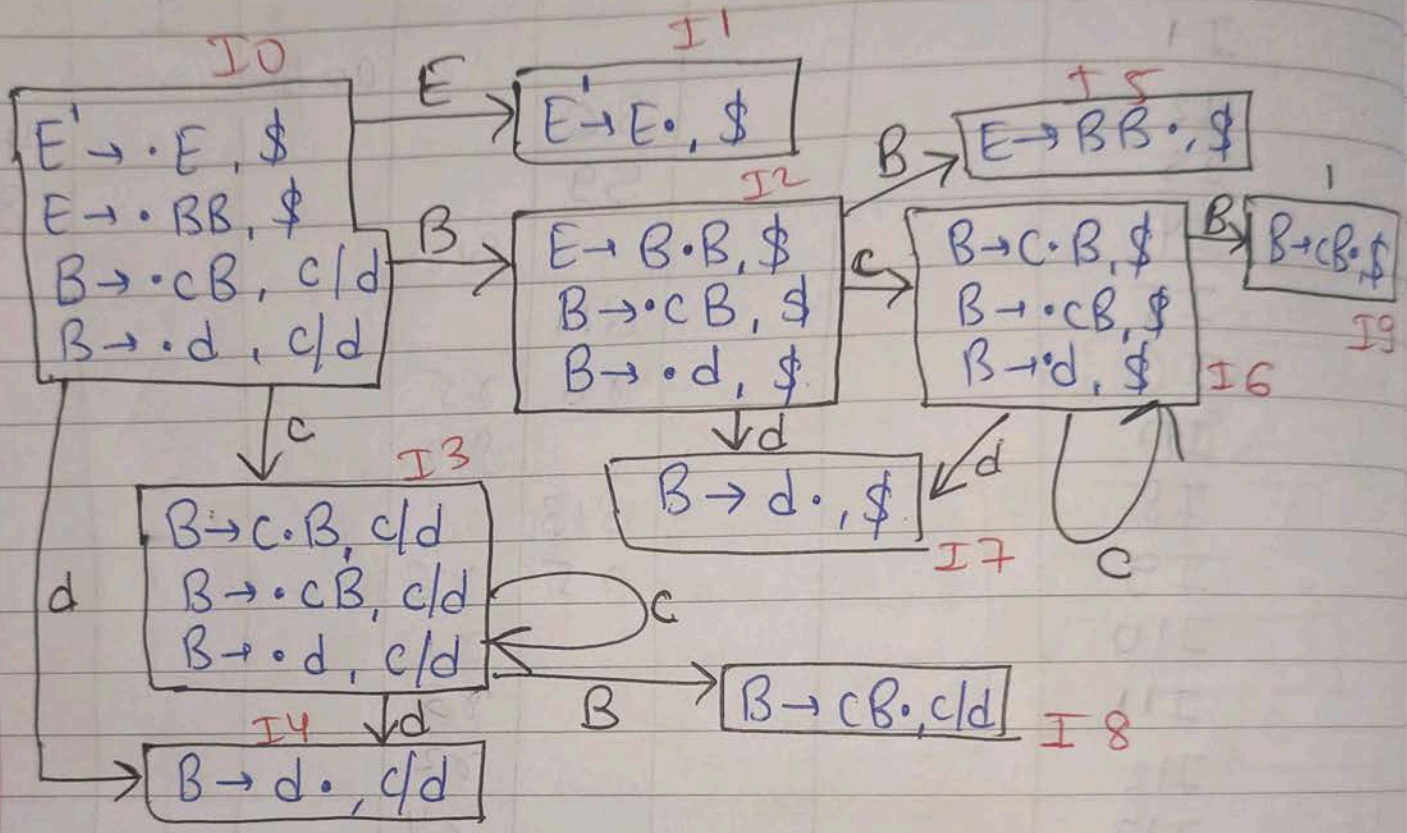
⇒ Similar to CLR But, We merge the production on same reduction. Like in this question, I6 and I9 is reducing same production. So, we can merge it.

	a	b	c	d	$	S	A	B
S1 → I6+I9			r6,r5	r5,r6				
I6 →			r6	r5				
I9 →			r5	r6				

2 production is one block

So, it is not LALR Parser. It is only CLR Parser

Q. $E \rightarrow BB$
$B \rightarrow cB \mid d$



	Action			Goto	
Stages	c	d	$	E	B
I0	S3	S4		1	2
I1					
I2	S6	S7			5
I3	S3	S4			8
I4	r3	r3			
I5			r1		
I6	S6	S7			9
I7			r3		
I8	r2	r2			
I9			r2		

For LALR :-

I3 & I6 → I36
I4 & I7 → I47
I5 & I9 → I89

	c	d	$	E	B
I0	S36	S47		1	2
I1			accept		
I2	S36	S47			5
I36	S36	S47			89
I47	r3	r3	r3		
I5			r1		
I89	r2	r2	r2		

Copy: I36 → I36, I47 → I47, I89 → I89

Now, I3&I6 having same values so we can ignore 1 of them. Same we can do with (I4&I7) & (I5&I9).

	c	d	$	E	B
I0	S36	S47		1	2
I1			accept		
I2	S36	S47			5
I36	S36	S47			89
I47	r3	r3	r3		
I5			r1		
I89	r2	r2	r2		

⇒ Both CLR(1) & LALR

Question to Practise :-

Q. Check if the following production is
(a) CLR or not
(b) LALR or not.

Ans to verify — (a) Yes,
(b) Yes

⇒ Isme dono table me difference v nhi ayega because, there is no 2 state which is proving same reduction.

Operator Precedence Parser :-

↳ Based on operator precedence grammar.
Rules :-

1. It should not have ϵ in RHS of production
$S \rightarrow \epsilon$ ✗
2. Two Non-terminal should not be adjacent
$E \rightarrow AC$ ✗
$E \rightarrow aC$ ✓
$E \rightarrow Ca$ ✓

Q. $E \rightarrow EAE \mid (E) \mid id$
$A \rightarrow + \mid - \mid * \mid /$

⇓

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

Precedence table Rules :-

1. id, a, b, c = higher precedence
   ↳ terminal

2. (+, +)
   (*, *)
   (/, /)
   (-, -)
   same operator → Left side has more precedence.

3. $ → lowest precedence

4. $$ → accept

→ Right

↓ left	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	>	>	<	=	<	X
)	<	<	X	>	X	>
id	>	>	X	>	X	>
$	<	<	<	X	<	accept

Q. E → EAE / id
A → + / *.

⇒ Step 1 → Check for operator precedence grammar
E → E+E / E*E / id
A → +, *.

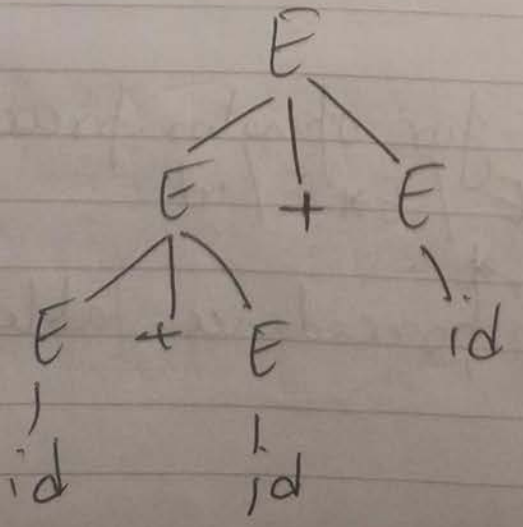
Step 2 → Draw precedence table

	id	+	*	$
id	≠	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	accept

Input String → id + id + id.
⇒ $id + id + id $.

Stack	Relation	String	Action
		$id + id + id $	Shift $
$.	<	id + id + id $	Shift id
$ id	>	+ id + id $	Reduce by E→id
$ E	<	+ id + id $	Shift +
$ E +	<	id + id $	Shift id
$ E + id	>	+ id $	Reduce E→id
$ E + E	>	+ id $	Reduce E→E+E
$ E	<	+ id $	Shift +
$ E +	<	id $	Shift id
$ E + id	>	$	Reduce E→id
$ E + E	>	$	Reduce E→E+E
$ E		$.	Accept

⇒



Q for Practise :-

$E \rightarrow EAE / id$

$A \rightarrow * / -$

Generate Parse tree for id - id * id.

Computation of Leading & Trailing :-

Leading :-

Rule 1 :- $A = \alpha a \gamma \rightarrow$ anything

(α : ϵ or single non-terminal; a : terminal)

$Leading(A) = a$

Rule 2 :- $A = B\alpha \rightarrow$ anything

(B : single Non-terminal)

$Leading(A) = Leading(B)$

Trailing :-

Rule 1 :- $A = \alpha a \gamma \rightarrow$ anything

(a : terminal; α : ϵ or single N-T)

$Trailing(A) = \{a\}$

Rule 2 :- $A = \alpha B \alpha \rightarrow$ anything

(B : single N.T)

$trailing(A) = trailing\{B\}$

Q. $E \to E+T$ ① | T ②
$T \to T*F$ ③ | F ④
$F \to (E)$ ⑤ | id ⑥

⇒ 1. $E \to E+T$

Leading (E) = {+}, Leading (E)
Trailing (E) = {+}, trailing (T).

How?

Leading :-

Rule 1 → $E \to \underbrace{E}_{\alpha} \underbrace{+}_{a} \underbrace{T}_{\gamma}$

Leading (E) = {+}

Rule 2 → $E \to \underbrace{E}_{B} \underbrace{+ T}_{\alpha}$

Leading (E) = Leading {E}

Trailing :-

Rule 1 → $E \to \underbrace{E}_{\alpha} \underbrace{+}_{a} \underbrace{T}_{\gamma}$

~~Leading~~ trailing (E) = {+}

Rule 2 → $E \to \underbrace{E +}_{\alpha} \underbrace{T}_{B}$

trailing (E) = trailing (T)

2. $E \to T$

Leading (E) = Leading (T)
Trailing (E) = Trailing (T).

Rule 1 not applicable as $E \to T$ does not in form $\alpha a \gamma$

↑ Terminal

Similarly, $T \to T * F$

$\Rightarrow$ Leading $(T) = \{*\}$, Leading (F)

$\Rightarrow$ Trailing $(T) = \{*\}$, Trailing (F)

$T \to F$

$\Rightarrow$ Leading (T) = Leading (F)

Trailing (T) = Trailing (F).

$F \to (E)$

$\Rightarrow$ Leading $(F) = \{(\}$

Trailing $(F) = \{)\}$.

$F \to id$

$\Rightarrow$ Leading $(F) = \{id\}$

Trailing $(F) = \{id\}$

Operator Precedence Relation Table Algo :-

1. Calculate Leading & Trailing for each N.T.
2. (a) $S \to ab$

$a = b$

(b) $S \to aAb$

(i) $a = b$

(ii) $a <$ Leading (A)

(iii) Trailing $(A) > b$

(c) If S = Starting N.T., $\$ \to$ end terminal

(i) $\$ <$ Leading (S)

(ii) Trailing $(S) > \$$

By using that let find Table of above question

N.T	Leading	Trailing
E	{+, *, id, (}	{+, *, id,)}
T	{*, id, (}	{*, id,)}
F	{id, (}	{id,)}

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	X
)	>	>	X	>	X	>
id	>	>	X	>	X	>
$	<	<	~~<~~	X	<	accept

$E \to E+T$

$\hookrightarrow E \to E+$ & $E \to +T$

form of $S \to Ab$ ⇓ form of $S \to aA$ ⇓

Trailing $(E) > +$; $+ < \text{Leading}(T)$

$\{+, *, id,)\} > +$; $+ < \{*, id, (\}$

(i) (ii)

$E \to T \Rightarrow$ ignored

3. $T \rightarrow T * F$

↳ $T \rightarrow T *$

⇓

Traling $(T) > *$

$\{*, id,)\} > *$ — (iii)

$T \rightarrow * F$

$* < Leading(F)$

$* < \{ id, (\}$ — (iv)

4. $T \rightarrow F \Rightarrow$ ignored

5. $F \rightarrow (E)$.

↳ $(=)$ — (v)

↳ $(< Leading(E)$

$\Rightarrow (< \{+, *, id, (\}$ — (vi)

↳ $Trailing(E) >)$

$\Rightarrow \{+, *, id,)\} >)$ — (vii)

6. $F \rightarrow id \Rightarrow$ ignored

$\$ < \{+, *, id, (\}$ — (viii)

$\{+, id, id,)\} > \$$ — (ix)

Now, just fill table using these 9 equation.

Unit - 4

Intermediate Code Generation

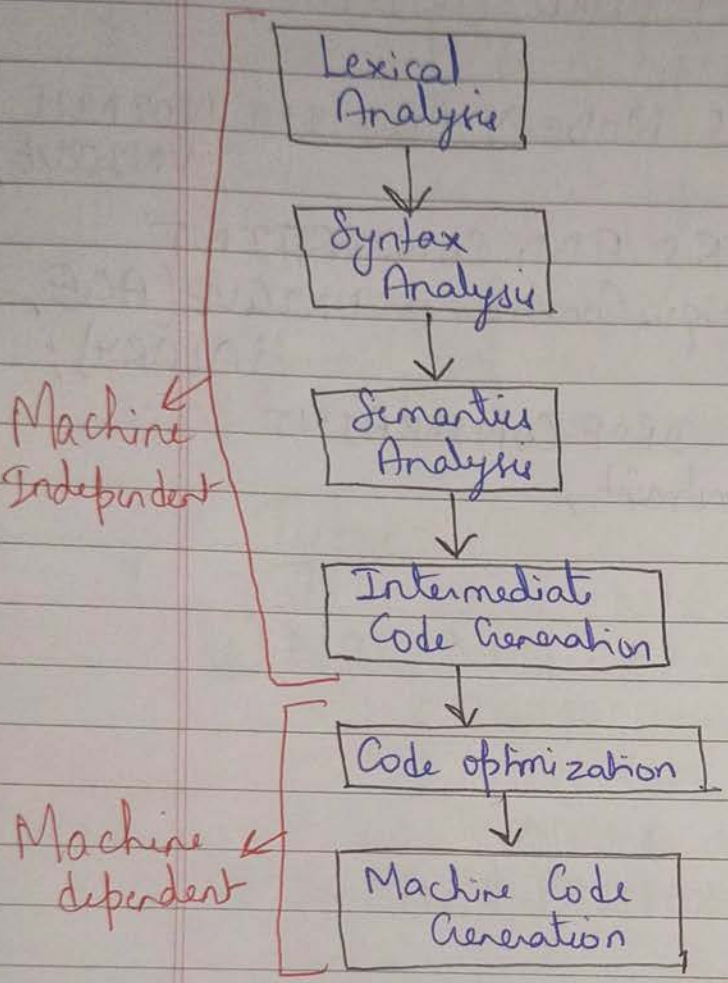


fig. Position of Intermediat Code Generation

Various forms :-

1. Linear form
   a. Prefix
   b. Postfix
   c. 3 address Code
2. Tree form
   a. Syntax Tree
   b. DAG

a. Prefix Notation :-

$a+b \Rightarrow +ab$

$(a+b)*c \Rightarrow *(+ab)c$

$(a+b)*(c-d) \Rightarrow *(+ab)(-cd)$

b. Postfix Notation :-

$a+b \Rightarrow ab+$

$(a+b)*c \Rightarrow (ab+)c*$

$(a+b)*(c-d) \Rightarrow (ab+)(cd-)*$

c. 3 address Code :-

1. x op z → (variable operator variable)
2. op z → (operator variable)

Ex → $u = a+b*c+d$

1. $T1 = b*c$
   $\Rightarrow a+T1+d$
2. $T2 = a+T1$
   $\Rightarrow T2+d$
3. $T3 = T2+d$
4. $u = T3$

Example 2 :- $a=(b*-c)+(b*-c)$

1. $T1 = -c$
2. $T2 = b*T1$
3. $T3 = -c$
4. $T4 = b*T3$
5. $T5 = T2+T4$
6. $a = T5$

Ans

Implementation of 3-address Code :-

A 3-address Code is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representation are :-

1) Quadruples
2) Triples
3) Indirect Triples

Quadruples :-

→ A quadruple is a record structure with four fields which are op, arg1, arg2 and result.

→ The op field contains an internal code for the operator. The 3-address statement x := y op z is represented by placing y in arg1, z in arg2. and x in result.

Triples :- → 3 fields op, arg1, arg2.

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

Indirect triples :-

Another implementation of 3-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.

So, lets see with the example. We take example of previous ques.

Quadruples :-

#	op	arg1	arg2	result
(1)	−	c		T1
(2)	*	b	T1	T2
(3)	−	c		T3
(4)	*	b	T3	T4
(5)	+	T2	T4	T5
(6)	—/=	T5		a

Triples :-

#	op	arg1	arg2
(1)	−	c	
(2)	*	b	(1)
(3)	−	c	
(4)	*	b	(1)
(5)	+	(2)	(4)
(6)	=	a	(5)

Indirect Triples :-

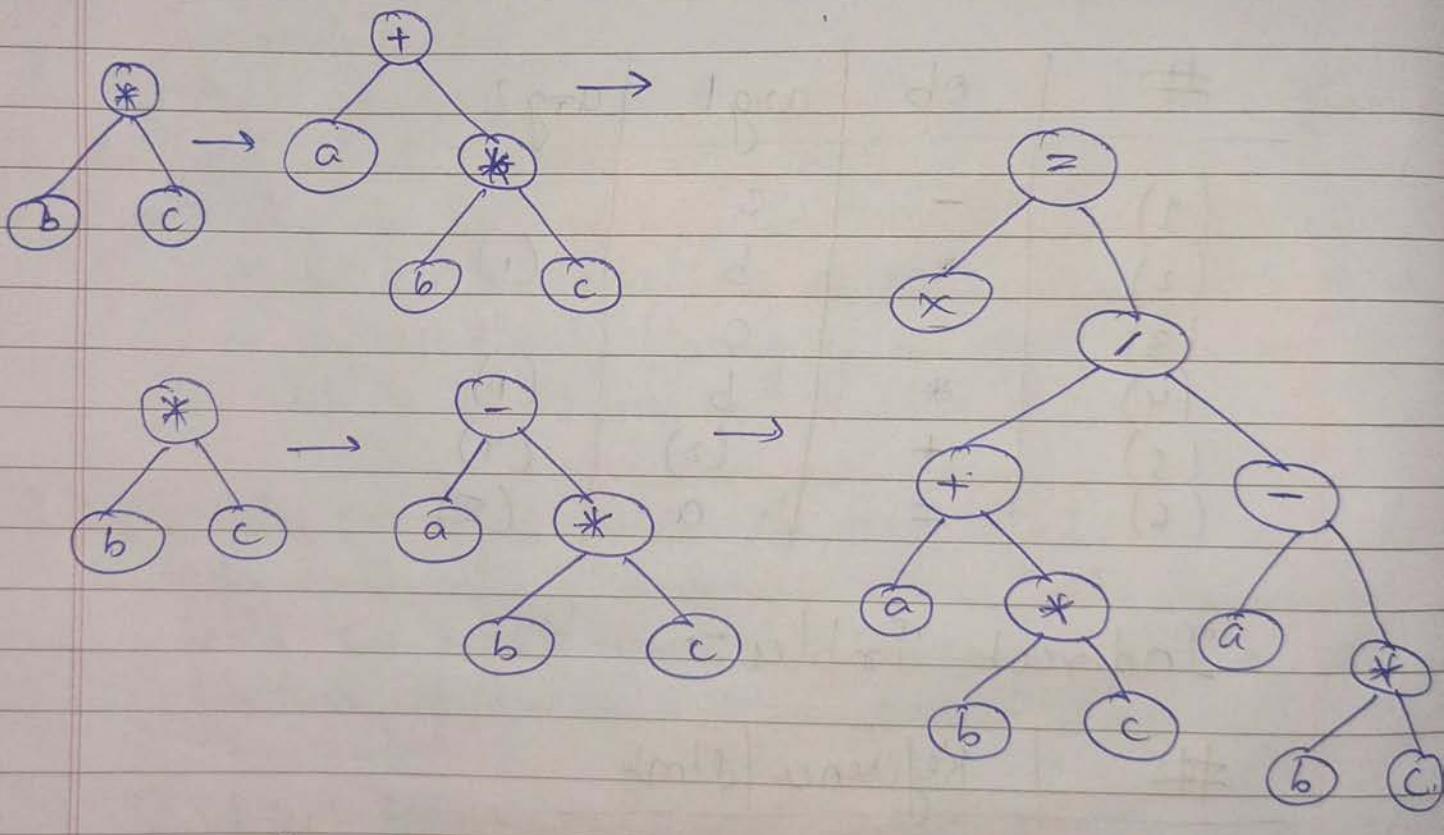
#	Reference/stmt
(1)	(13)
(2)	(14)
(3)	(15)
(4)	(16)
(5)	(17)
(6)	(18)

Question to practise :-

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

Syntax Tree :- A syntax tree depicts the natural hierarchical structure of a source program.

Q. $n = (a + b*c) / (a - b*c)$



Q. Basic Block & Control flow Graph :-

Partition Algorithm :-

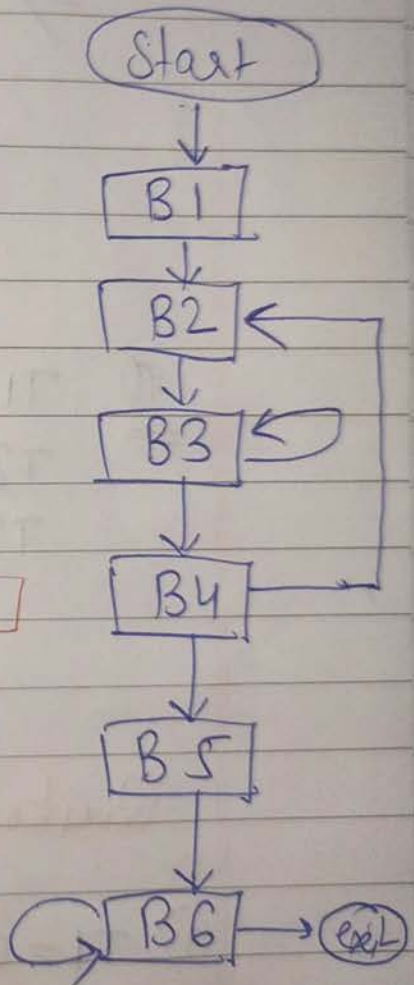
1. First statement of a 3 addres Code is a leader.
2. Target instruction of conditional/unconditional branch is leader
3. Next Stmt of conditional/unconditional branch is leader.

leader ← t1 = a * a
t2 = a * b
] B1

leader ← t3 = 2 + 1
t4 = t1 + t3
t5 = 2 * t2
t6 = t3 + t5
if i - - goto statement 3
] B2

leader ← t7 = a .
] B3

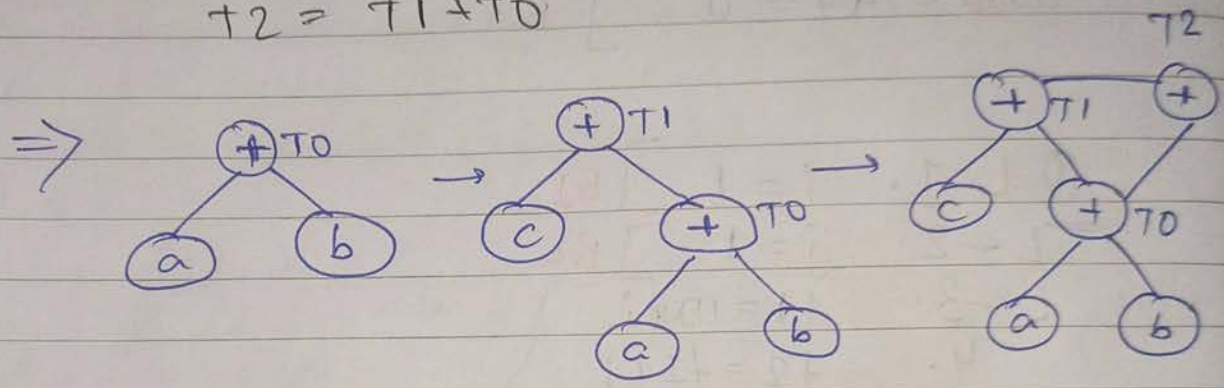
Q. L ← 1. i = 1] B1
L ← 2. j = 1] B2
L ← 3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 1.0
8. j = j + 1
9. If j <= 10 go to stmt 3] B3
L ← 10. i = i + 1] B4
11. if i <= 10 goto stmt 2
L ← 12. i = 1] B5
L ← 13. t5 = i - 1
14. t6 = t5 * 88
15. a[t6] = 1.0] B6
16. i = i + 1
17. if i <= 10 go to stmt 13.



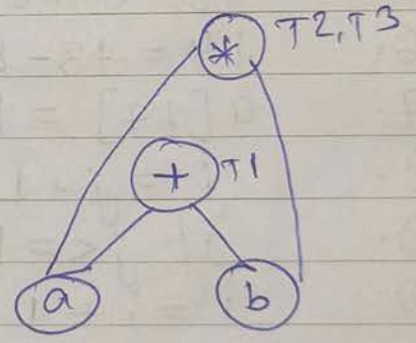
Directed Acyclic Graph :-

A DAG gives the same information as syntax tree but in a more compact way because common subexpressions are identified.

Q. T0 = a+b
T1 = c+T0
T2 = T1+T0



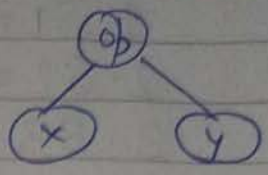
Q. T1 = a+b
T2 = a*b
T3 = a*b

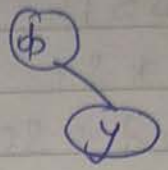


Question to practise :-

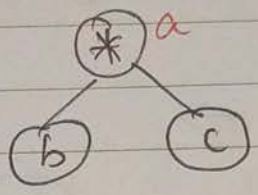
1) T1 = a-b
T2 = a+b
T3 = T1+T2
T4 = 4*T1
T5 = 4*T2
T6 = 4*T2

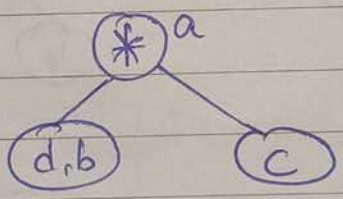
2) S1 = 4*i
S2 = a[S1]
S3 = 4*i
S4 = b[S3]
S5 = S2*S4
S6 = prod * S5

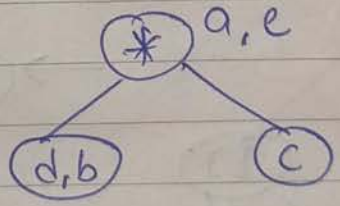
Rules :- 1) $a = x \text{ op } y \Rightarrow$ 

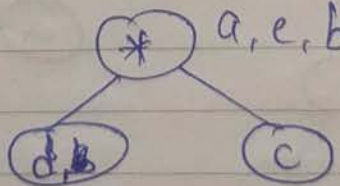
2) $b = \text{op } y \Rightarrow$ 

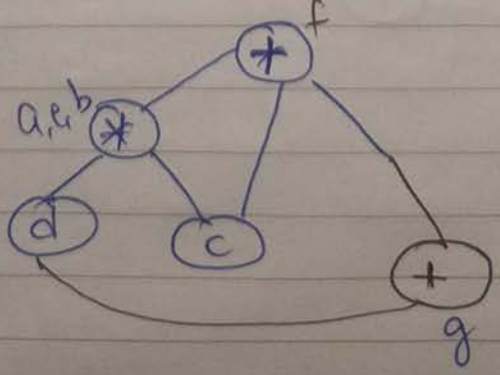
3) $u = y$
(i) x is a operand
(ii) y is a label $\Rightarrow$ ◯ y, x.

1. $a = b * c$ 

2. $d = b$ 

3. $e = d * c$ 

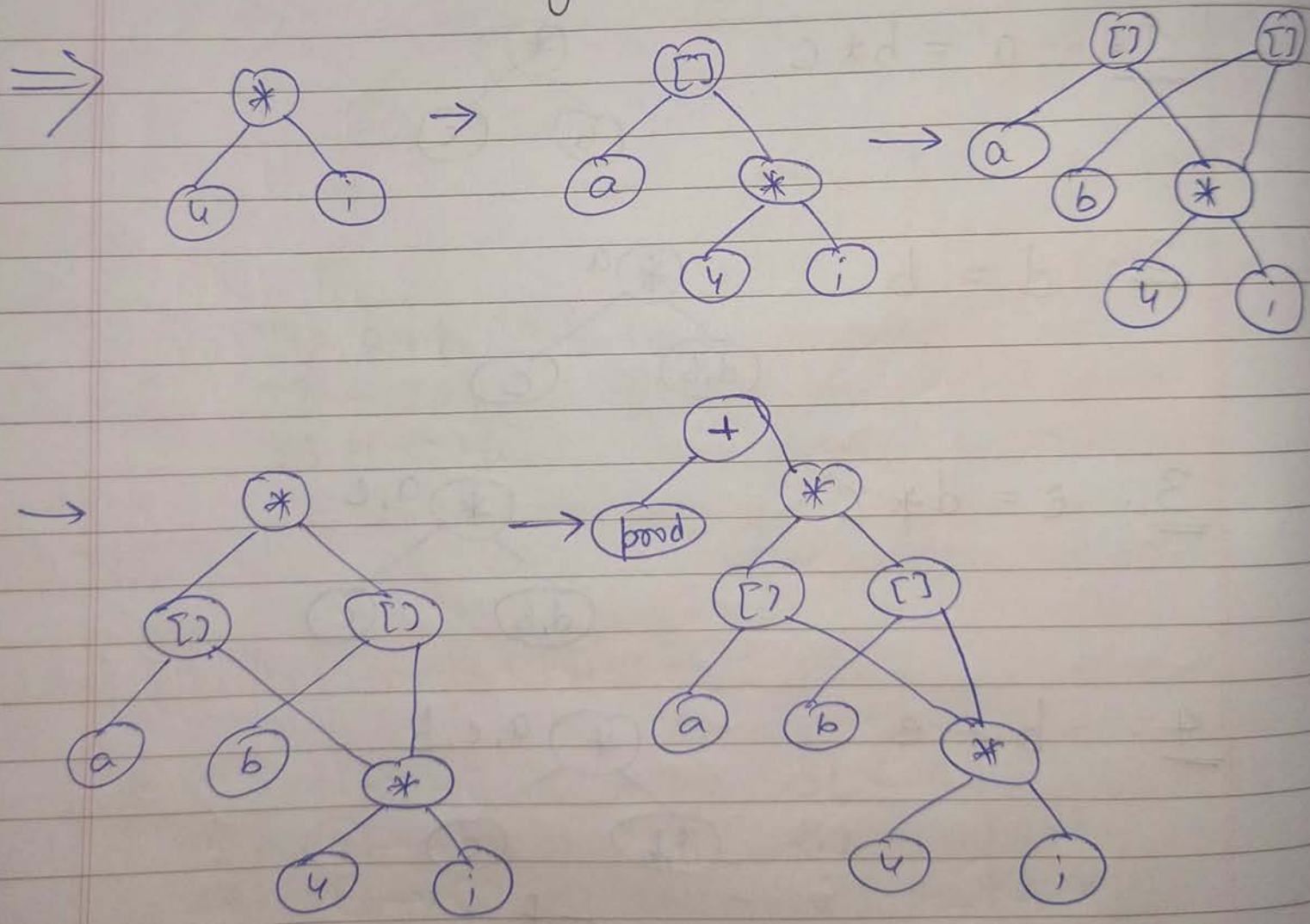
4. $b = e$ 

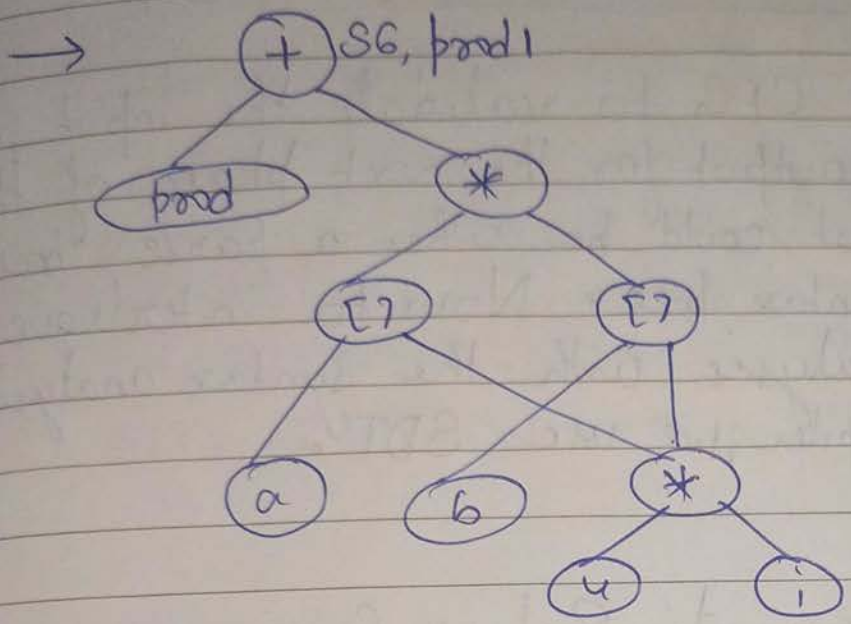
5. $f = b + c$ 

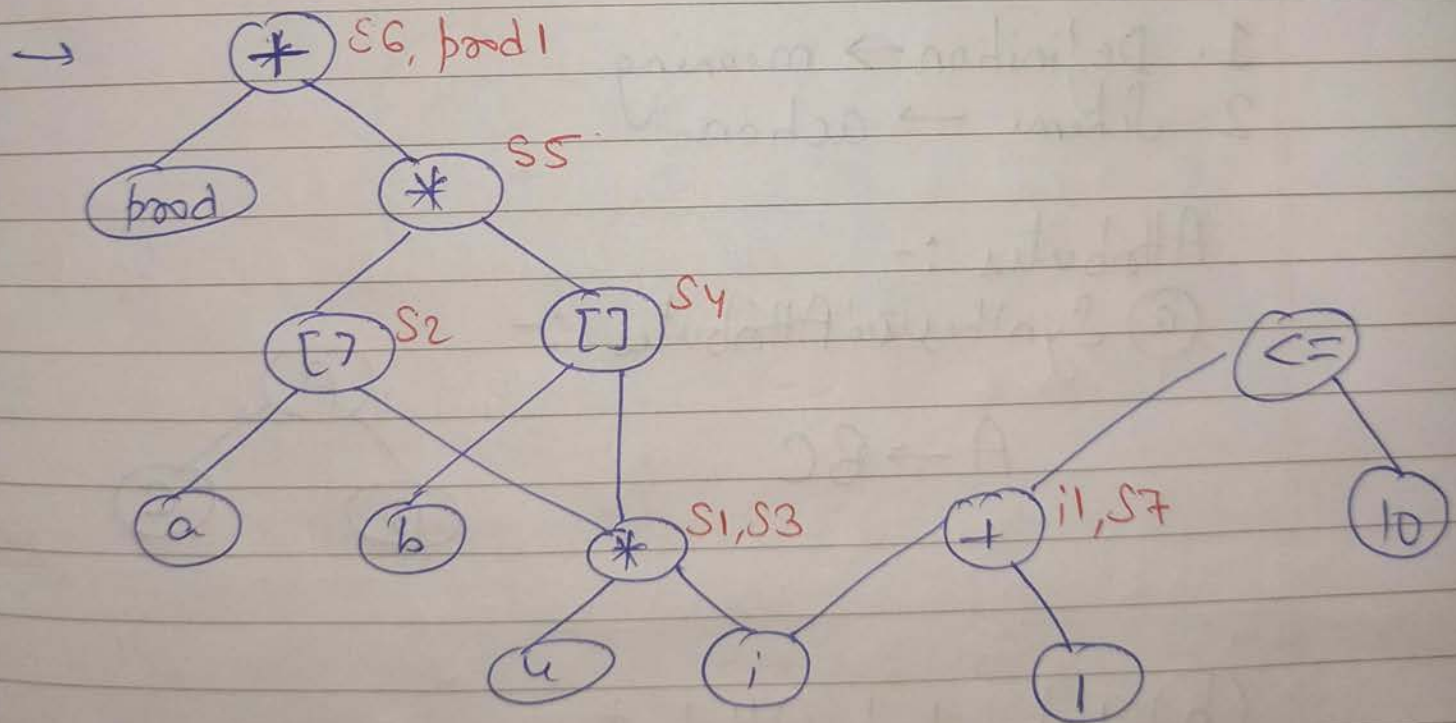
6. $g = d + f + d$

Q.

1. S1 = 4*i
2. S2 = a[S1]
3. S3 = 4*i
4. S4 = b[S3]
5. S5 = S2 * S4
6. S6 = prod + S5
7. prod = S6
8. S7 = i+1
9. i = S7
10. if i<=10 go to step 1




+
S6, prod1
prod
*
[]
[]
a
b
*
u
i


*
S6, prod1
prod
*
S5
[]
S2
[]
S4
a
b
*
S1, S3
u
i
<=
+
i1, S7
10
1

Syntax Directed Translation

Parser use a CFG to validate the input string and produce output for the next phase of the compiler. Output could be either a parse tree or an abstract syntax tree. Now to interleave semantic analysis with the syntax analysis phase of compiler, we use SDT.

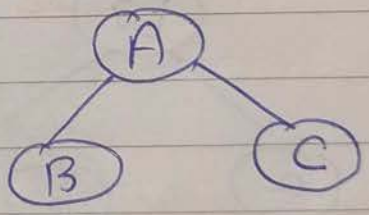
SDT = Semantic Rule + Grammar

1. Definition → meaning
2. Scheme → action

Attributes :-

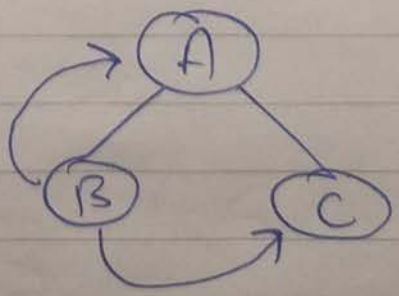
(a) Synthesized Attributes :-

$A \rightarrow BC$



(b) Inherited Attributes :-

$A \rightarrow BC$
$B \rightarrow A$
$B \rightarrow C$

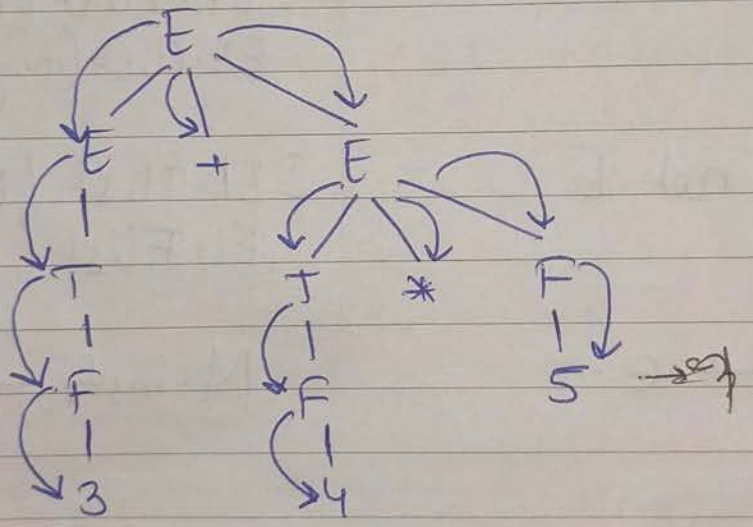


SDT Definition

Production Rule	Meaning	action
$E \rightarrow E+T$	E.value = E.value + T.value	print '+'
$E \rightarrow T$	E.value = T.value	{ }
$T \rightarrow T*f$	T.value = T.value + f.value	print '*'
$T \rightarrow F$	T.value = f.value	{ }
$F \rightarrow num$	F.value = num.lexvalue	print 'num'

SDT Scheme

Q. 3+4*5



⇒ print(3] → print 4 → print 5 → print * → print +

⇒ 345*+

Q. 3*4*5

Backpatching :-

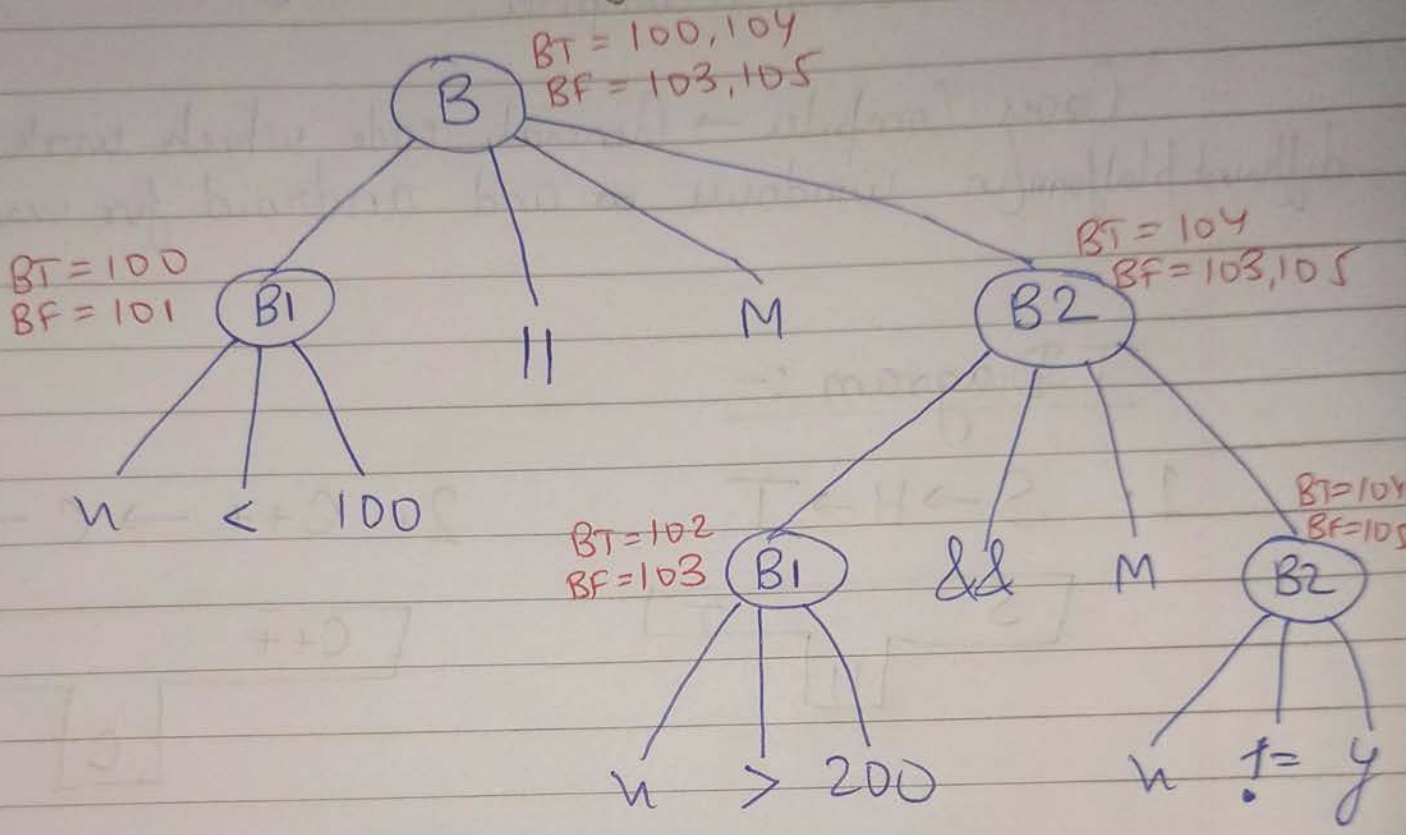
3 function :-
Backpatch ()
merge ()
makelist ()

Production Rule	Semantic Action
$E \rightarrow E_1$ or $M E_2$	{ backpatch(E1.falselist, M.instr) E1.Tlist(merge(E1.Tlist, E2.Tlist)) E2.Flist(E2.Flist) }
$E \rightarrow E_1$ and $M E_2$	{ backpatch(E1.tlist, M.instr) E1.Tlist(E1.Tlist) E2.flist(merge(E1.Flist, E2.Flist)) }
$E \rightarrow$ not E	{ E1.Tlist(E1.flist) E1.Flist(E1.Tlist) }
$E \rightarrow \epsilon$	M.instr → next instruction

Q. $a < 100 \,||\, a > 200 \;\&\&\; a != y$

⇒
```
100: if a<100 go to 106
101: else goto 102
102: if a>200 goto 104
103: else goto 107
104: if a!=y then goto 106
105: else goto 107
106: True statement
107: False statement
```

Backpatching



B → B1 || B2 :-

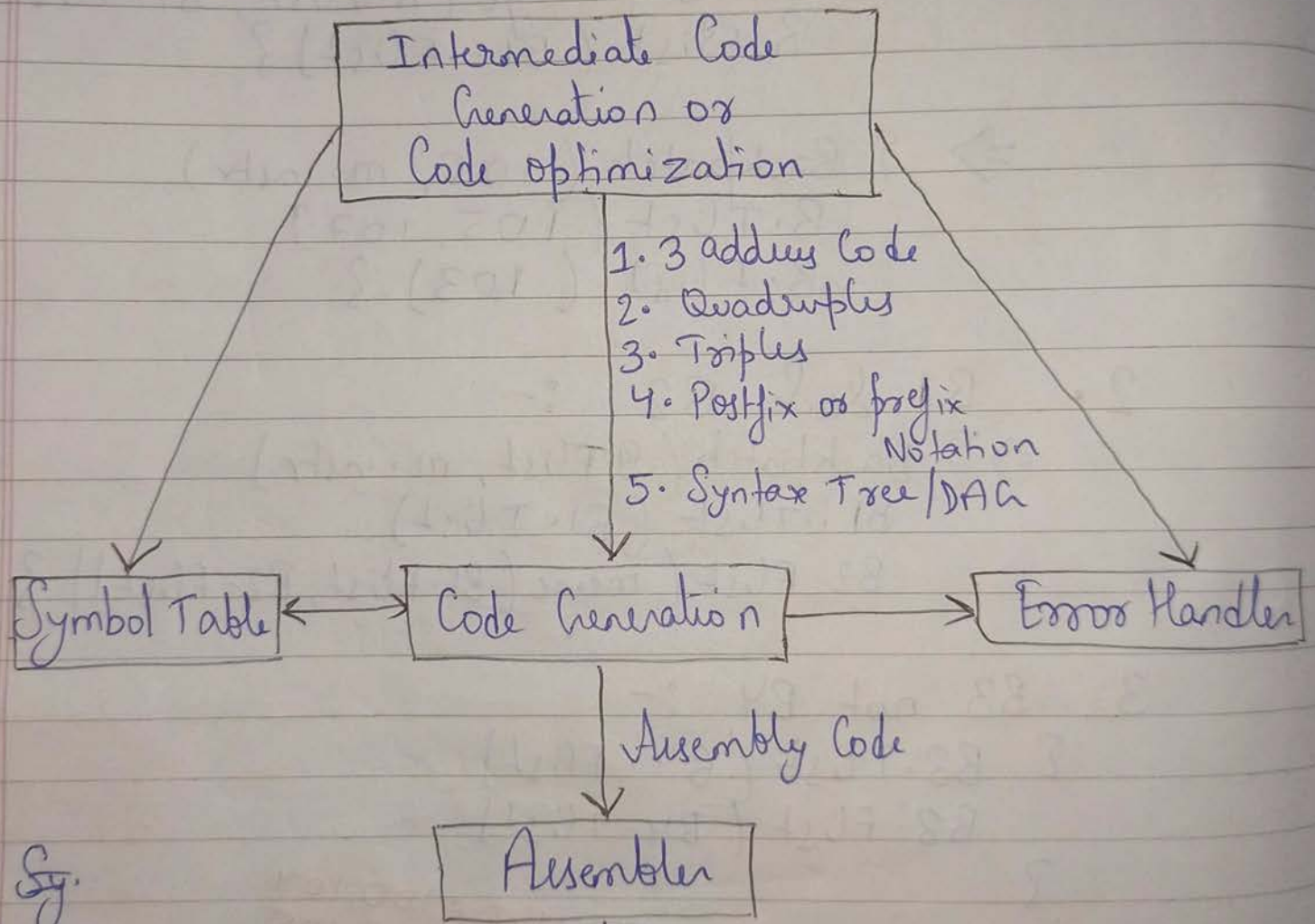
{ Backpatch(B.Flist, m.inst),
B.Tlist (merge (B1.Tlist, B2.Tlist))
B.Flist (B2.flist). }

B2 → B1 && B2 :-

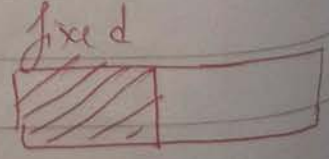
{
Backpatch (B.Tlist, m.inst).
B2.Flist(merge (B1.Flist, B2.Flist)),
B2.Tlist (B1.Tlist)) }

Code Generation

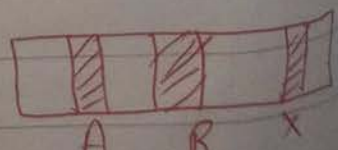
Properties :-
1. High performance
2. Correctness
3. Efficient use of resource of target machine
4. Quick code generation



Eg.



1. Absolute m/c Code
2. Relative m/c Code



Issues in Code Generation :-

1. Input target Problem
2. Code generation
3. Memory Management
4. Instruction Selection
5. Register allocation

Q. $x = y + z$

⇒
```
MOV y, R0
ADD z, R0
MOV R0, x
```

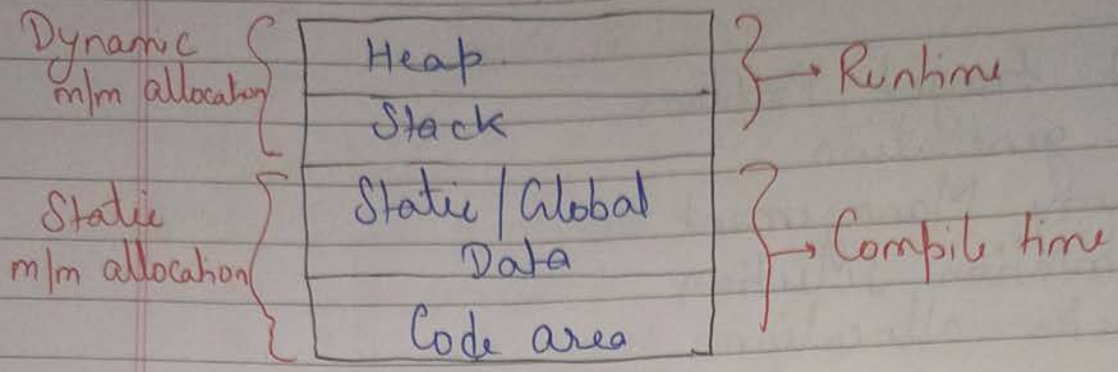
Q.
```
a = b + c
d = a + e
```

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

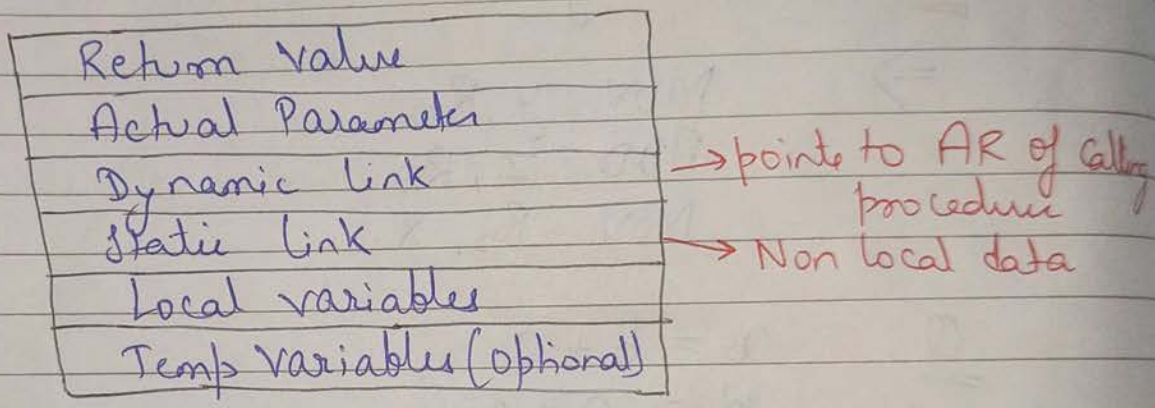
⇒
```
MOV b, R0
ADD c, R0
ADD e, R0
MOV R0, d
```

Q.
```
a = b * c
d = e * f
x = a + d
y = x + g
```

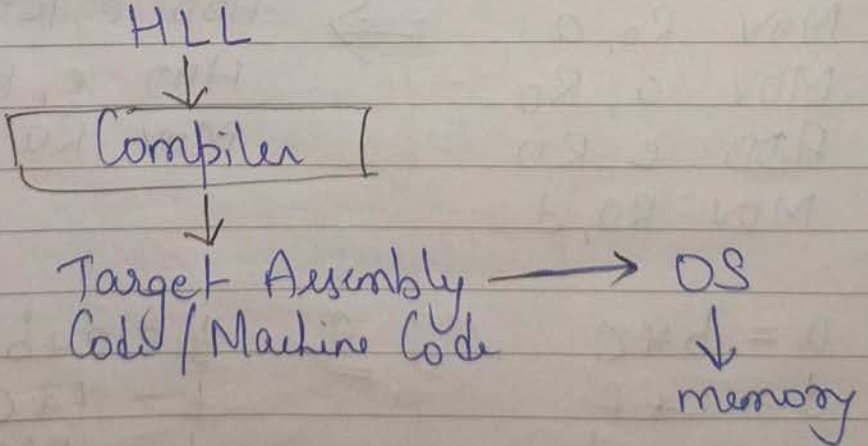
Q.
```
t = a + b
t = t * c
t = t / d
```



Stack allocation Scheme :-



Runtime Storage Management :-



Activation Record (AR) :-

```
main() {
    a1()
}
```

```
a1() {
    b1()?
b1() {
  - -
  3.
```

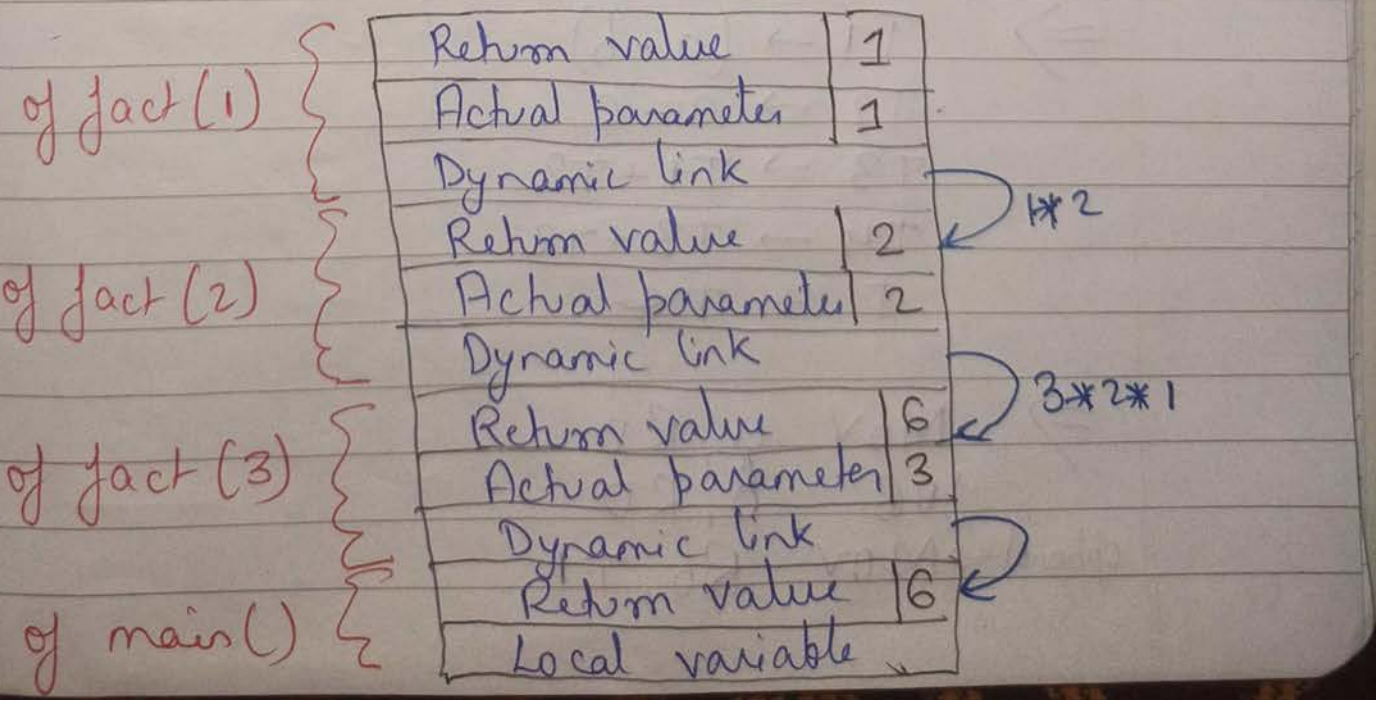
⇒ All segments have different activation allocation Record

AR → main
a1
b1

Q.
```
main() {
    int f;
    f = fact(3);
}
int fact (int n)
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
```

⇒
main()
fact(3)
fact(2)
fact(1)

⇒



Code Generation Algorithm

1. Invoke function getreg() to store output into register L.
2. If value of y is already in address descriptor then go ahead otherwise store address of y in y' and store it in address descriptor
3. Repeat step 2 for x
4. Update value of L, store output of x of y.
5. If there is no use of x and y in future store the value in register descriptor.

Register Descriptor → Store variables.

x	y
R_0	R_1

Address Descriptor → stores address

Q. ~~$T_1 \rightarrow a - b$~~
~~$T_2 \rightarrow a -$~~ $T = (a-b) + (a-c) + (a-c)$

⇒ $T1 \rightarrow (a-b)$
$T2 \rightarrow a-c$
$T3 \rightarrow T1+T2$
$T4 \rightarrow T3+T2$
$T = T4$

⇒ MOV a, R0
SUB b, R0
optional → MOV R0, T1

```
        MOV  a, R1
        SUB  c, R1
optional → MOV  R1, T2
        ADD  R1, R0
        ADD  R1, R0
optional → MOV  R0, T4
        MOV  R0, T
```

S.No	Statement	Code Generated	Register Descriptor	Address Descriptor
1.	T1 = a - b	MOV a, R0 SUB b, R0	R0 Contains T1	T1 in R0
2.	T2 = a - c	MOV a, R1 SUB c, R1	R0 Contains T1 R1 Contains T2	T1 in R0 T2 in R1
3.	T3 = T1 + T2	ADD R1, R0	R0 Contains T3 R1 Contains T2	T3 in R0 T2 in R1
4.	T4 = T3 + T2	ADD R1, R0	R0 Contains T4 R1 Contains T2	T4 in R0 T2 in R1
5.	T = T4	MOV R0, T		

Question to Practice :-

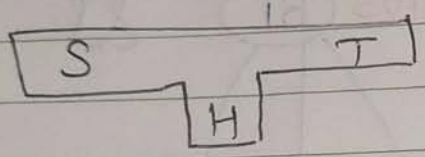
1) n = (a/b) + (c*d) - (c+a) + (f-c)
2) n = (a-b) + (c*d) - (e/f) + (g-a)
3) T1 = A - B
   T2 = C - D
   T3 = E - T1
   T4 = T1 + T2

Bootstrapping :- Process to generate compiler or cross compiler.

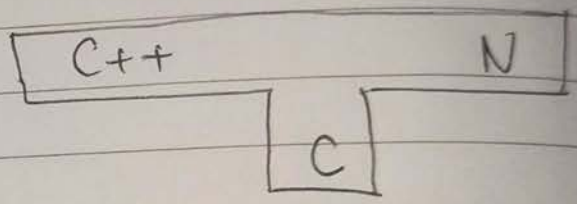
Cross Compiler → Generate code which work both in different platform (in windows and android for example).

T Diagram :-

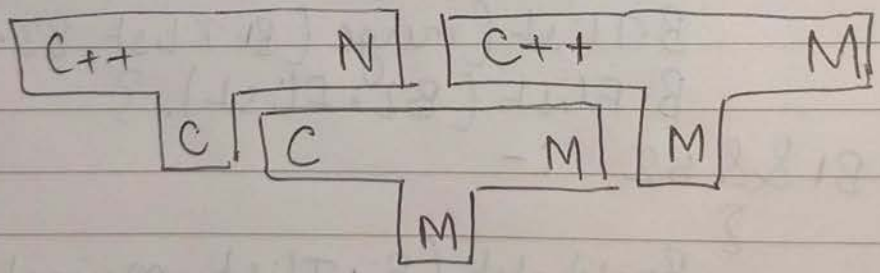
1. S → H → T



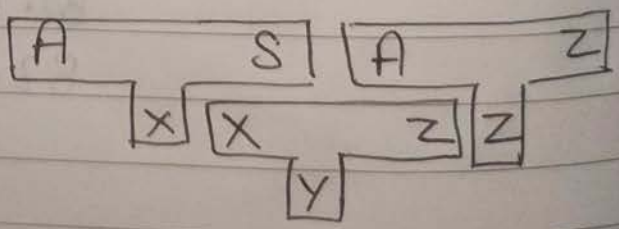
2. C++ → C → N



1. C → M → M
   C++ → C → N.



2. X → Y → Z
   A → X → S
   A → ? → Z



Q. 3. A → B → C D → ? → C. ?
   D → A → X

Compiler Design
Unit - 5

Code optimization :-

Elimination of unnecessary instruction in object code or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization". (Space & time).

Function Preserving Transformations :-

There are no. of ways in which a compiler can improve a program without changing the function it computes.

Function preserving Transformation examples :-

(i) Common Subexpression Problem :-

An occurrence of expression E is called common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

	old	new
a	2	5
b	3	1
c	2	3
d	4	1

value not changed

$$\begin{aligned} a &= b+c \\ b &= a-d \\ c &= b+c \\ d &= a-d \end{aligned} \quad \Longrightarrow \quad \begin{aligned} a &= b+c \\ b &= a-d \\ c &= b+c \\ d &= b \end{aligned}$$

so, we can replace

(ii) Constant folding :-

If the value of an expression is constant, use the constant instead of expression.

Eg → PI = 22/7 or PI = 3.14.

(iii) Copy Propagation :-

Eg →

```
x = a              x = a
y = x * b    ⇒     y = a * b
z = a * c          z = a * c
```

↳ Copy ka time bacha. But it introduced dead code. as x = a is useless now.

(iv) Dead Code Elimination :-

Eg →

```
x = a
y = a * b    ⇒     y = a * b
z = a * c          z = a + c
```

(v) Code Motion :- Loops are very important place for optimizations, especially the inner loops where program tend to spend bulk of their time. The running time of a program may be improved if we decrease the no. of instruction in an inner loop. So, an important modification that decrease the amount of code in a loop is Code Motion.

Ex→
```
while (i<10) {
    x = y+z
    i = i+1
}
```
⇒
```
x = y+z
while (i<10) {
    i = i+1
}
```

(vi) Induction variables elimination & Reduction in Strength :-

Another important optimization is to find induction variables in loops and optimize their computation.

A variable x is said to be an 'induction variable' if there is a positive or negative constant C such that each time x is assigned, its value increases by C. It can be computed with a single increment or decrement per loop iteration.

Eg →
```
i = 1
while (i<10) {
    t = i*4
    i = i+1
    print(t)
}
```
⇒
```
t = 4
while (t<=40) {
    t = t+4;
    print(t)
}
```

Code optimization in Basic Blocks :-

- Structure Preserving transformation
  - Dead Code Elimination
  - Common Subexpression Elimination
  - Renaming of temporary variables
  - Interchange of two independent adjacent statement.
- Algebric transformation
  - Constant folding
  - Copy propagation
  - Strength Reduction

→ Dead Code Elimination :-

```
x = 2
if (x > 2)
print ("Code is Correct");
else
print ("Code is incorrect");
```

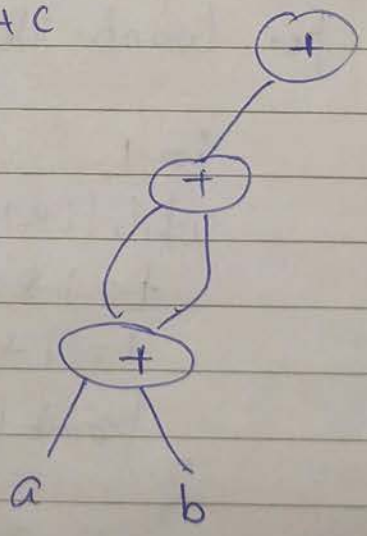
⇒

```
x = 2
print ("Code is incorrect");
```

→ Common ~~Subexpression~~ Subexpression Elimination :-

In this technique, the subexpression which are common and used frequently, are calculated only once and reused when needed. DAG is used to eliminate common subexpressions.

$x = (a+b) + (a+b) + c$



→ Renaming of temporary variables :-

$t = a+b$

$x = a+b$

→ Interchange of two independent adjacent Statement :-
If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

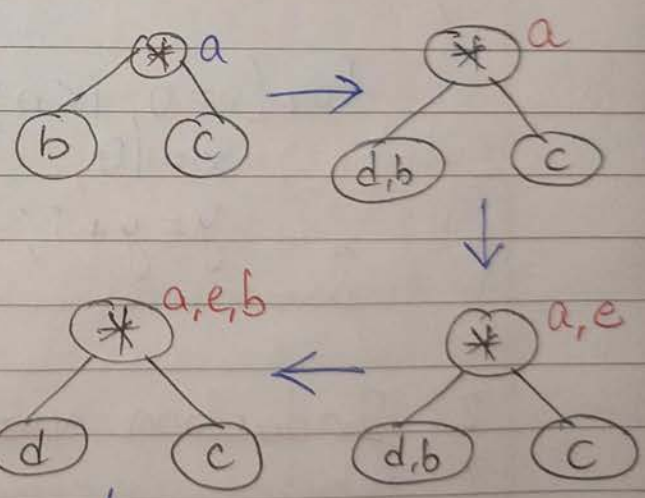
$t1 = a + b$
$t2 = c + b$

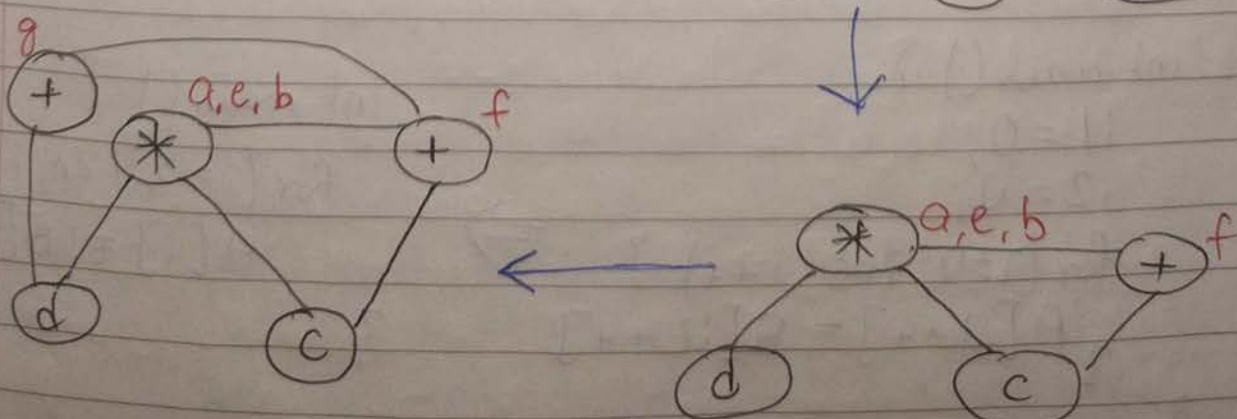
→ Algebraic Transformation :-
- → Constant folding :- $x = 2*3+y \Rightarrow x = 6+y$
- → Copy propagation :- $x = 3$, $y = x+c$ $\Rightarrow$ $x = 3$, $y = 3+c$
- → Strength Reduction :- $x = y*2$ $\Rightarrow$ $x = y+y$

Building Expressions of DAG :-

$a = b*c$
$d = b$
$e = d*c$
$b = e$
$f = b+c$
$g = d+f$

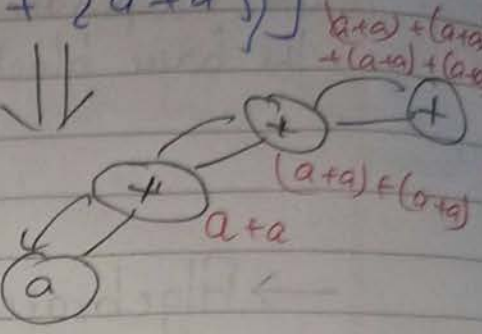
Draw optimized DAG :-



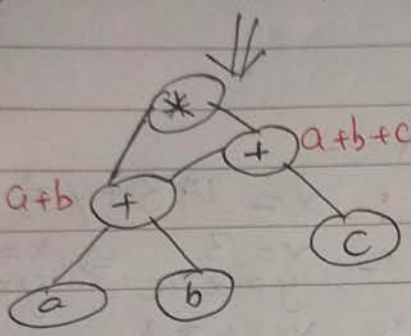


Question to practise :-

1) $[\{(a+a)+(a+a)\}+\{\{a+a\}+\{a+a\}\}]$



2) $(a+b)*(a+b+c)$.



Loop Optimization :-

1) Code Motion and Frequency reduction :-

```
for (i=0; i<n; i++){
    x=10;
    y=y+i;
}
```

⇒

```
x=10;
for (i=0; i<n; i++){
    y=y+i;
}
```

2) Induction ~~various~~ Variable Elimination :-

```
int main(){
    i1=0;
    i2=0;
    for (i=0; i<n; i++){
        A[i1++]=B[i2++]
    }
}
```

⇒

```
int main(){
    for (i=0; i<n; i++)
        A[i]=B[i]
}
```

3. Loop Merging :-

```
for(i=0; i<n; i++)
    A[i] = i+1;
for(j=0; j<n; j++)
    B[i] = j-1;
```

⇒

```
for(i=0; i<n; i++)
    A[i] = i+1;
    B[i] = i-1;
```

4. Loop Unrolling :-

```
main() {
    for(i=0; i<3; i++)
        Cout<<'cd';
}
```

⇒

```
main() {
    Cout<<'cd';
    Cout<<'cd';
    Cout<<'cd';
```

Peephole Optimization :-

1) Redundant load and store elimination
2) Constant folding
3) Strength Reduction
4) Null Sequence / Simplify Algebric Expression
5) Combine operations
6) Dead Code elimination

1) In this redundancy is eliminated.

```
y = x+5
i = y;
z = i;
w = z*3;
```

⇒

```
y = x+5;
w = y*3;
```

2) PI = 22/7 ⇒ PI = 3.14

3) y = x*2 ⇒ y = x+x

4) a = a+0 a = a/1
a = a*1 a = a-0

avoid using these. delete them if there

5) a = b+c ⇒ a = b+c+e
a = a+e

6) Dead code eliminate

Global Data flow Analysis

Steps :-

1) Assign a distinct number to each definition as d_1, d_2, d_3, etc
2) for each variable k, make a list of all definition in the entire diagram where it is used.
3) Calculate following :-
   a) GEN[B] = the set GEN[B] consist of all definition that are generated in blocks.
   b) KILL[B] → the set of all definitions that are outside of block & having same definition as block B.

Q4) Comput the following :-

a. IN[B] :- The set of all definitions reaching the point just before the first stmt of block B.

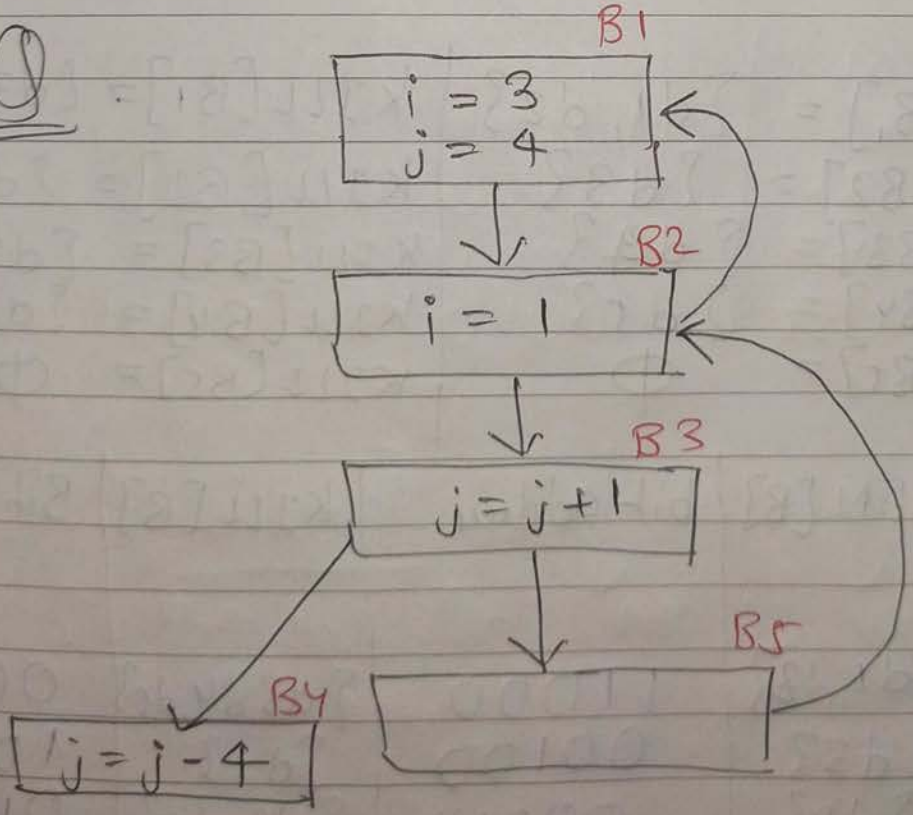
b. OUT[B] = ~~ending~~ the set of all definitions reaching the point just after the ending stmt of block B.

$$IN[B] = \bigcup OUT[B]$$

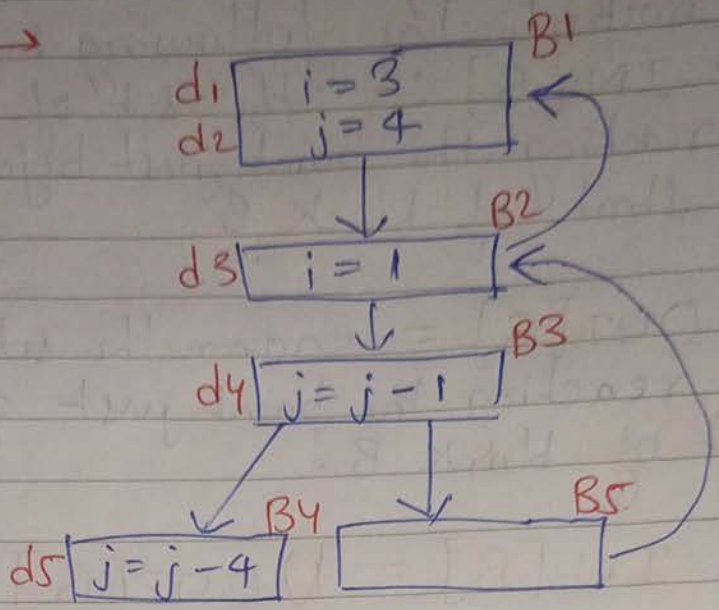
$$OUT[B] = IN[B] - KILL[B] \cup GEN[B]$$

$$= \{IN[B] \text{ AND } (\sim KILL[B])\} \text{ OR } GEN[B]$$

Q.



Sol → Step 1 →



Step 2 :- $i = \{d1, d3\}$
$j = \{d2, d4, d5\}$

Step 3 :-

GEN[B1] = {d1, d2}	KILL[B1] = {d3, d4, d5}
GEN[B2] = {d3}	KILL[B2] = {d1}
GEN[B3] = {d4}	KILL[B3] = {d2, d5}
GEN[B4] = {d5}	KILL[B4] = {d2, d4}
GEN[B5] = ϕ	KILL[B5] = ϕ

Block	GEN[B]	bit address	KILL[B]	Bit Analysis
B1	{d1, d2}	11000	{d3, d4, d5}	00111
B2	{d3}	00100	{d1}	10000
B3	{d4}	00010	{d2, d5}	01001
B4	{d5}	00001	{d2, d4}	01010
B5	ϕ	00000	ϕ	00000

Step 4 :-

Pass 0 :-

Block	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass 1 :-

→ IN[B1] = OUT[B2]
= 00100

→ IN[B2] = OUT[B1] ∪ OUT[B5]
= 11000 ∪ 00000 ⇒ 11000

→ IN[B3] = OUT[B2]
= 00100

→ IN[B4] = OUT[B3]
= 00010

→ IN[B5] = OUT[B3]
= 00010

→ OUT[Bi] = [IN[Bi] AND (~KILL[Bi])] OR GEN[Bi]
= [00100 AND (~00111)] OR 11000
= [00100 AND 11000] OR 11000
= [00000 OR 11000]
= 11000

OUT[B2] = [IN[B2] AND (~KILL[B2])] OR GEN[B2]
= (11000 AND 01111) or 00100
= 01000 or 00100
= 01100

OUT[B3] = [IN[B3] and (~KILL[B2])] OR GEN[B3]
= (00100 AND 10110) OR 00010
= 00100 OR 00010
= 00110

OUT[B4] = [IN[B4] and (~KILL[B4])] OR GEN[B4]
= (00010 AND 10101) or 00001
= 00000 or 00001 = 00001

OUT[B5] = [IN[B5] and (~KILL[B5])] OR GEN[B5]
= (00010 AND 11111) OR 00000
= 00010

Block	IN[B]	OUT[B]
B1	00100	11000
B2	11000	01100
B3	00100	00110
B4	00010	00001
B5	00010	00010

Similarly, we find Pass 2, Pass 3 — and so on, till we get 2 pass same.

	Pass 2		Pass 3		Pass 4	
Block	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B1	01100	11000	01110	11000	01110	11000
B2	11010	01110	01110	01110	01110	01110
B3	01100	00110	01110	00110	01110	00110
B4	00110	00101	00110	00101	00110	00101
B5	00110	00110	00110	00110	00110	00110

↓

These are value I got. You can verify by doing it on your own.

Introduction to Global Data Flow Analysis

Global Data Flow Analysis

- **To efficiently optimize the code** compiler **collects all the information about the program and distribute this information to each block of the flow graph.** This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

- Based on the local information a compiler can perform some optimizations. For example, consider the following code:
- x = a + b;
- x = 6 * 3
- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

For example, consider the following code:

```
a = 1;
b = 2;
c = 3;
if (....)
    x = a + 5;
else
    x = b + 4;
c = x + 1;
```

- In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

- Global data flow analysis is used to solve a specific problem "**User definition chaining**".
- Problem: Given that the identifier A is used at a point p in the program. At what point could the value of A have been defined.
- **Reaching definition:** Determination of each variable whenever they are declared in the program.

For this it follows following steps-

1. Assign a distinct number to each definition such as d1, d2, d3,…..
2. For each variable i, make a list of all definition in entire diagram where it is used.

For i, definitions d1 and d3 are used.

For j, definitions d2, d4 and d5 are used.

3. For each basic block calculate the following:

a. **GEN[B]-** The set GEN[B] consists of all definitions generated in block B**.**

GEN[B1] – d1, d2

b. **KILL[B]-** The set of all the definition outside block B that define the same variables having definitions in block B also.

KILL[B1]- d3, d4, d5

4. For all basic block compute the following:

a. IN[B]: The set of all definitions reaching the point just before the first statement of block B.

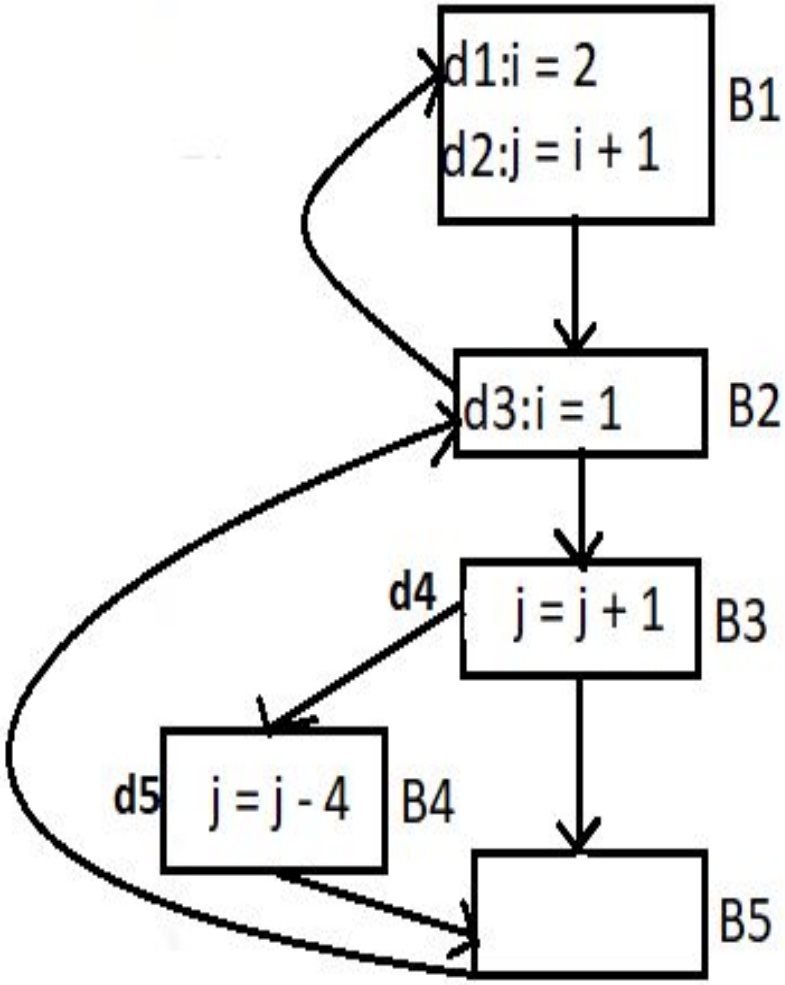
b. OUT[B]: The set of all definitions reaching the point just after the last statement of block B.

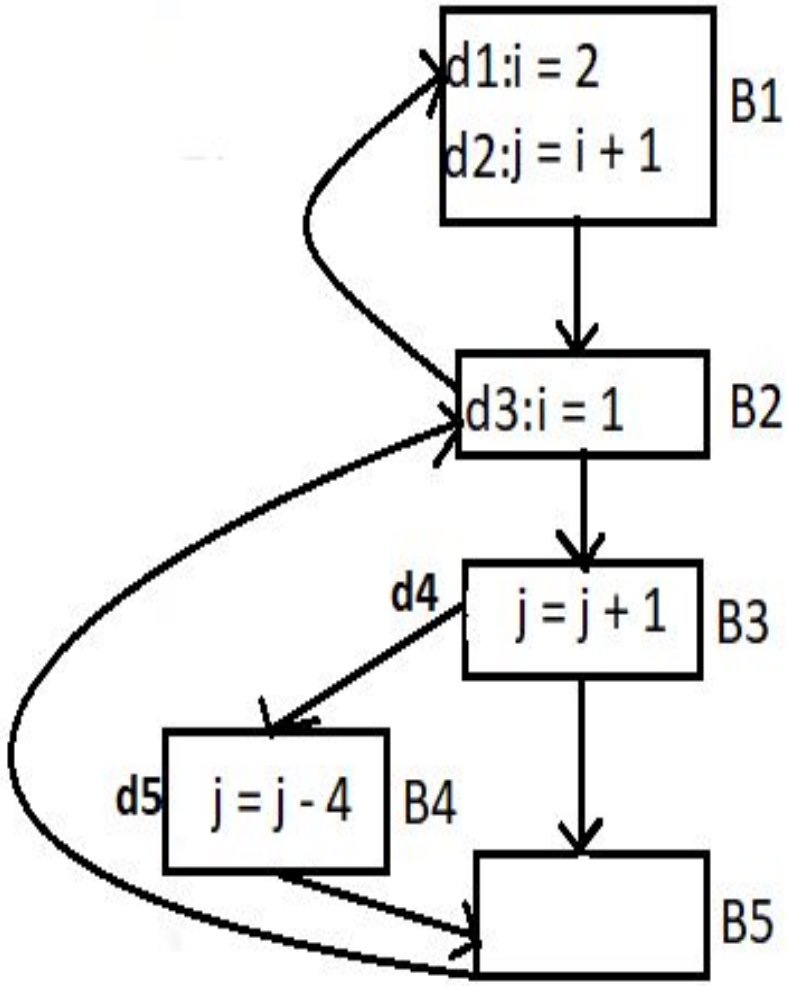
Data Flow Equation:

OUT[B] = IN[B] – KILL[B] U GEN[B]

OUT[B] = { IN[B] AND (~KILL[B]) U GEN[B]

IN[B] = U OUT[B] (Union of all out B from all previous blocks)





1. Find GEN and KILL for each block.
2. Find IN and OUT for reaching definitions.

Block B	GEN [B]	Bit vector d1 d2 d3 d4 d5	KILL [B]	Bit vector d1 d2 d3 d4 d5
B1	[d1, d2]	11000	[d3, d4, d5]	00111
B2	[d3]	00100	[d1]	10000
B3	[d4]	00010	[d2, d5]	01001
B4	[d5]	00001	[d2, d5]	01010
B5	Φ	00000	Φ	00000

Initially Pass 0,

- IN[B] = Φ, OUT[B] = GEN[B]

Block [B]	IN [B]	OUT [B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass 1:

Block [B]	IN [B]	OUT [B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

For Pass 1:
IN [B1] = OUT [B2] = 00100 (refer Pass 0)
OUT [B1] = {IN [B1] AND (~KILL [B1])} OR GEN [B1]
= (00100 AND 11000) OR 11000
= 11000

IN [B2] = OUT [B1] UNION OUT [B5]
= 11000 UNION 00000 = 11000
OUT [B2] = {IN[B2] AND (~KILL [B2])} OR GEN [B2]
= 11000 AND 01111 OR 00100
= 01100

Similarly calculate the values of Pass 2, Pass 3, Pass 4……., until all the values of two passes are same.

Parameter Passing

- All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments.

- In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition).

1. Call By Value

- In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable).
- The value is placed in the location belonging to the corresponding formal parameter of the called procedure.
- This method is used in C and Java, and is a common option in C++, as well as in most other languages.
- Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

2. Call- by-Reference

- In call- by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller.
- Changes to the formal parameter thus appear as changes to the actual parameter.
- If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own.
- Changes to the formal parameter change this location, but can have no effect on the data of the caller.

3 Call By Name

- A third mechanism - call-by-name - was used in the early programming language Algol 60.
- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).
- When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

Storage Allocation strategies

1 Static Versus Dynamic Storage Allocation:

- The layout and allocation of data to memory locations in the run-time environment are key issues in storage management.
- The two adjectives static and dynamic distinguish between compile time and run time, respectively.
- We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
- Conversely, a decision is dynamic if it can be decided only while the program is running.

- Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. Stack storage. Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

- To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly

2 Stack Allocation of Space:

- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.
- This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

2.1 Activation Tree

- Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time.
- We can represent the activations of procedures during the running of an entire program by a tree, called an *activation tree.*
- Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- Notice that one child must finish before the activation to its right can begin.

- The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

2. The sequence of returns corresponds to a postorder traversal of the activation tree.

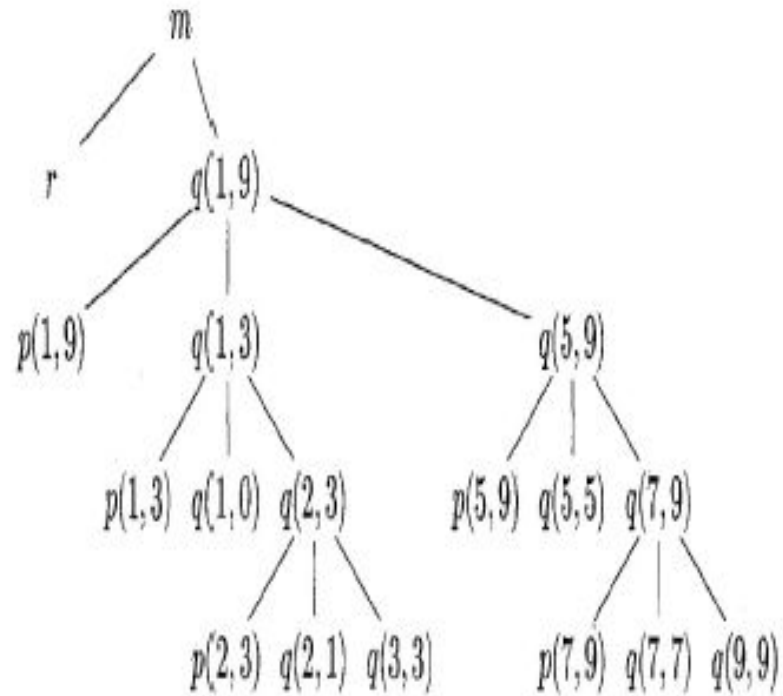
3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (live) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] s
       a[m..p-1] are less than v, a[p] = v, and a[p+1..
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Possible activations for the program

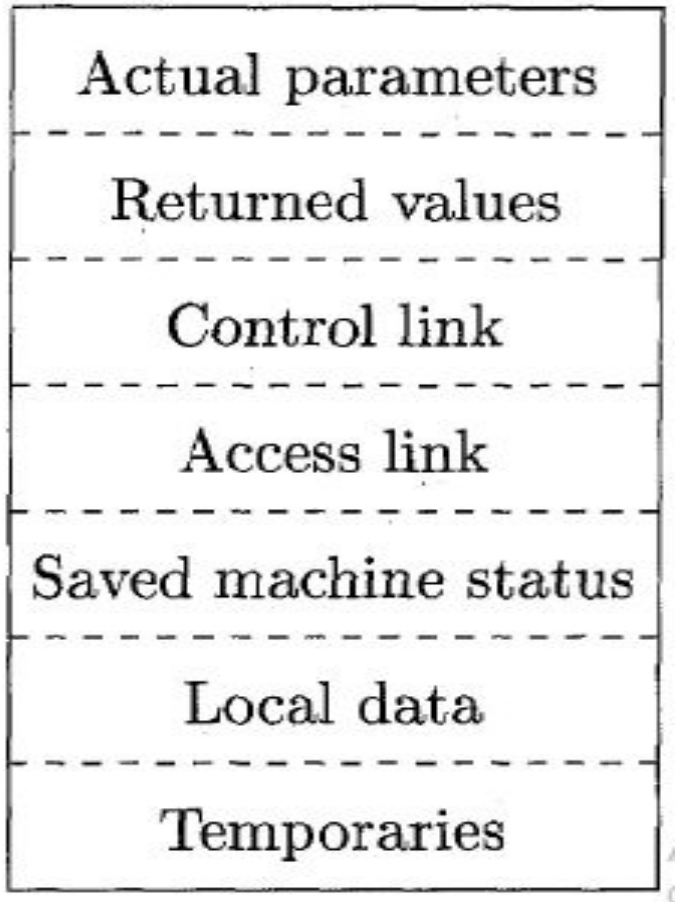


Activation tree representing calls during an execution of quicksort

2.2 Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack.

- The contents of activation records vary with the language being implemented.
- Here is a list of the kinds of data that might appear in an activation record:



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A control link, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

2.3 Calling Sequences

- Procedure calls are implemented by what are known as calling sequences, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

3 Heap Management

- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
- While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them.
- For example, both C++ and Java give the programmer ***new*** to create objects that may be passed - or pointers to them may be passed - from procedure to procedure, so they continue to exist long after the procedure that created them is gone.
- Such objects are stored on a heap.

3.1 The Memory Manager

- Memory manager is the subsystem that allocates and deallocates space within the heap; it serves as an interface between application programs and the operating system.
- The memory manager keeps track of all the free space in heap storage at all times.

It performs two basic functions:

1. Allocation.

- When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size.
- If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system.
- If space is exhausted, the memory manager passes that information back to the application program.

2. Deallocation.

- The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.
- Memory managers typically do not return memory to the operating system, even if the program's heap usage drops.

Memory management would be simpler if

(a) all allocation requests were for chunks of the same size, and

(b) storage were released predictably, say, first-allocated first-deallocated.

- There are some languages, such as Lisp, for which condition (a) holds; pure Lisp uses only one data element - a two pointer cell - from which all data structures are built.
- Condition (b) also holds in some situations, the most common being data that can be allocated on the run-time stack.

- However, in most languages, neither (a) nor (b) holds in general.
- Rather, data elements of different sizes are allocated, and there is no good way to predict the lifetimes of all allocated objects.
- Thus, the memory manager must be prepared to service, in any order, allocation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

1. Space Efficiency: A memory manager should minimize the total heap space needed by a program.

- Doing so allows larger programs to run in a fixed virtual address space.
- Space efficiency is achieved by minimizing "fragmentation.“

2. Program Efficiency: A memory manager should make good use of the memory subsystem to allow programs to run faster.

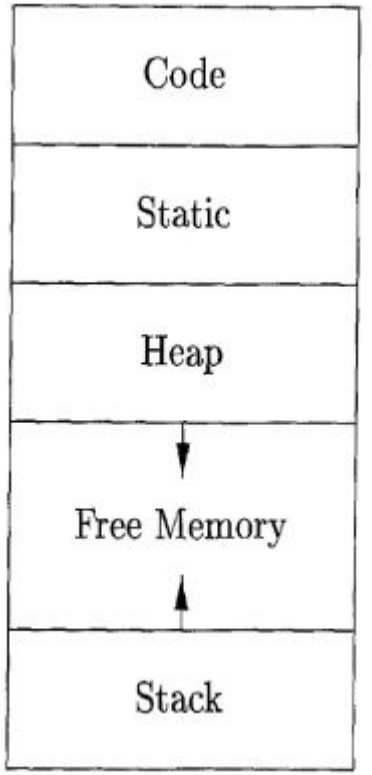
- The time taken to execute an instruction can vary widely depending on where objects are placed in memory.
- Fortunately, programs tend to exhibit "locality," a phenomenon which refers to the nonrandom clustered way in which typical programs access memory.
- By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster.

3. Low Overhead: Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.

- That is, we wish to minimize the *overhead* - the fraction of execution time spent performing allocation and deallocation.

Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.
- The run-time representation of an object program in the logical address space consists of data and program areas.
- A compiler for a language like C++ on an operating system like Linux might subdivide memory in this way.



Subdivision of run-time memory into code and data areas

- The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, usually in the low end of memory.

- Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called Static.

- One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

- To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space.

- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls.
- An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs.
- When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.
- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- For example, C has the functions **malloc** and **free** that can be used to obtain and give back arbitrary chunks of storage.
- The heap is used to manage this kind of long-lived data.

Runtime Environments

- A compiler must accurately implement the abstractions embodied in the source language definition.
- These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.

- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

- This environment deals with a variety of issues such as the

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