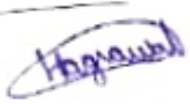
Declaration and statement of authorship

I, bearing Registration Number 106118036, agree and acknowledge that:

1. The assessment was answered by me as per the instructions applicable to each assessment, and that I have not resorted to any unfair means to deliberately improve my performance.

2. I have neither impersonated anyone, nor have I been impersonated by any person for the purpose of assessments.

Signature of the Student : 

Full Name : Harshit Agrawal

Roll No. : 106118036

Sub Code : CSPC 41

Mobile No. : 7016004637

1) Statement :- $x = b + a \wedge c / 4.5 - 3$

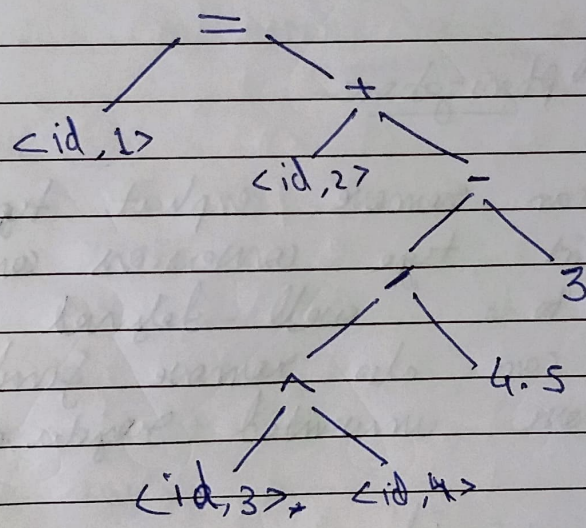
($\wedge$ → Power)

A) Stage 1 :- Lexical Analysis

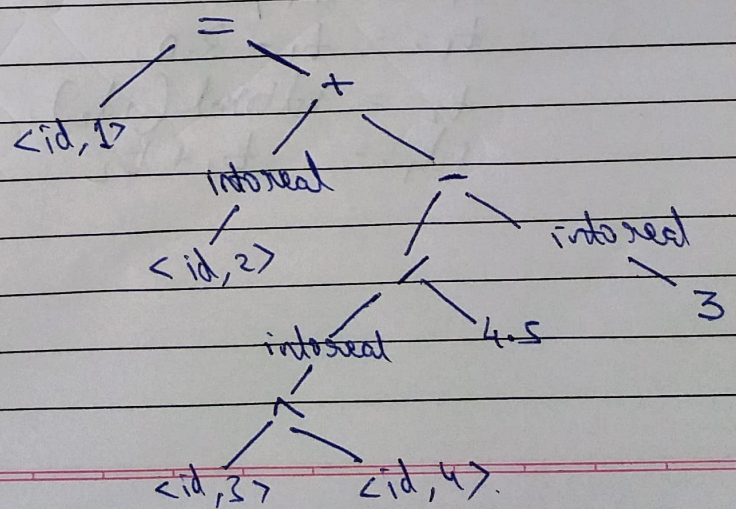
<id,1> <=> <id,2> <+> <id,3> <^> <id,4>
</> <4.5> <-> <3>

B) Syntax Analyzer :-

Generating Parse Tree



C) Semantic Analysis :-



D) ICG :- We write the 3 address code.

$t_1 := id_4$
$t_2 := id_3 \wedge t_1$;
$t_3 := \text{inttoreal}(t_2)$;
$t_4 := t_3 / 4.5$;
$t_5 := \text{inttoreal}(3)$;
$t_6 := t_4 - t_5$;
$t_7 := \text{inttoreal}(id_2)$;
$t_8 := t_7 + t_6$;
$id_1 := t_8$;

E) Code Optimization :-

- We can remove explicit type conversions as implicit type conversions can take place due to a well-defined 'x'.
- We can also remove final copy propogation.
- Remove unwanted registers or extra variables.

$t_1 := id_4$
$t_1 := id_3 \wedge t_1$
$t_1 := t_1 / 4.5$ [Implicit Conversion happens]
$t_1 := t_1 - 3.0$
$t_2 := \text{inttoreal}(id_2)$
$id_1 := t_1 + t_2$

(E) Code generation:-

```
LDF    R1, id4;
POW    id3, R1;
DIV    #4.5, R1;
SUB    #3.0, R1;
LDF    R2, id2;
ADD    R2, R1;
STF    id1, R1;
```

∴ id1 get the value from R1 after final result.

2.) Lex Program :- Count Vowels in string.

Program :-

```
%{
        int vCount = 0;
%}

%%
[aeiouAEIOU] { vCount++; }
%%

int yywrap() {}
int main()
{
    printf(" Input String: ");
    yylex();
    printf(" Num Vowels : %d\n", vCount);
    return 0;
}
```

3.) Grammar:-

$S \rightarrow BCD \mid aD$
$A \rightarrow CFB \mid aA$
$B \rightarrow b \mid \epsilon$
$C \rightarrow dB \mid \epsilon$
$D \rightarrow cA \mid \epsilon$
$E \rightarrow e \mid fE$

We can see that the grammar here is free from any Left Recursion & Left Factoring.

∴ We can write :-

$First(S) = \{a\} \cup first(B) \cup First(C) \cup First(D)$

$= \{a, b, c, d, \epsilon\}$

$First(A) = \{a\} \cup First(C) \cup First(E)$

$= \{a, d, e, f, \epsilon\}$

$First(B) = \{b, \epsilon\}$
$First(C) = \{d, \epsilon\}$
$First(D) = \{c, \epsilon\}$
$First(E) = \{e, f\}$

→

Now, for follow, we get

Follow (S) = {$}

follow (A) = follow (D)
= {$}

Follow (B) = first (C) ∪ follow (C) ∪ first (D) ∪ Follow (D)
= {d, c, $}

follow (C) = {c} ∪ follow (B) ∪ first (E)
= {c, f, e, $}

follow (D) = follow (S) = {$}

follow (E) = first (B) ∪ follow (S)
= {b, $}

4.) Grammar

$S' \rightarrow S$

$S \rightarrow L = E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

$L \rightarrow id$

$Elist \rightarrow Elist , E$

$Elist \rightarrow id [E$

So,

Goto	State	Closure
	0	{ [S' → .S, $], [S → .L = E, $], [L → .Elist], =], [L → .id, =], ...
goto(0, S)	1	{ [S' → S., $] }
goto(0, L)	2	{ [S → L. = E, $] }
goto(0, Elist)	3	{ [L → Elist.], =/$/+/], , /)], [Elist → Elist. , E,],] }
goto(0, id)	4	{ [L → id., =/$/+/], /)], [Elist → id. [E,], /,] }
goto(2, =)	5	{ [S → L = . E, $], [E → . E + E, $/+], ...
goto(3,])	6	{ [L → Elist]., =/$/+/], /)] }
goto(3, ,)	7	{ Elist → Elist , . E,], /,], ...
goto(4, [)	8	...
goto(5, E)	9	...

Goto	State	Closure
goto (5, c)	9	{ [S → L = E . , $] [E → E . + E , $/+]
goto (5, L)	10	{ [E → (. E) , $/+/)] ...
goto (5, Elist)	11	{ [E → L . , $/+/)] , [ )] }
~~goto (5, id)~~		
~~goto (7, E)~~	3	
goto (7, E)	4	
goto (7, ()	12	{ [Elist → Elist E .] / ,] ; ...
goto (7, L)	10	
.	11	
.	3	
.	4	
.	13	

Following the closure table, we get the LR table.

State	Action					Goto		
=	+	(	)	]	id, c, $	S', S	E	L	Elist
0									
1					S_4				
2	S_5								
3									
4	r_6		r_6			S_6	S_7		
5						r_6	r_6		
6									
.					S_{10}				
.									
11					S_{10}				
12									
13						S_{17}			
.						r_4			
17						r_5			

5.) TAC for the given code can be written as:-

low = -2 ---- leader (B_0)
high = 100

L1: if low >= high goto L7 ---- (B_1)
flag = 0 ------ B_2
if low > 1 goto L2
low = low + 1
goto L1

L2: i = 2 ---- B_4
t_1 = low / 2

L3: if i > t_1 goto L5 ---- B_5
t_2 = low % i ---- B_6
if t_2 != 0 goto L4
flag = 1 ------ B_7
goto L5

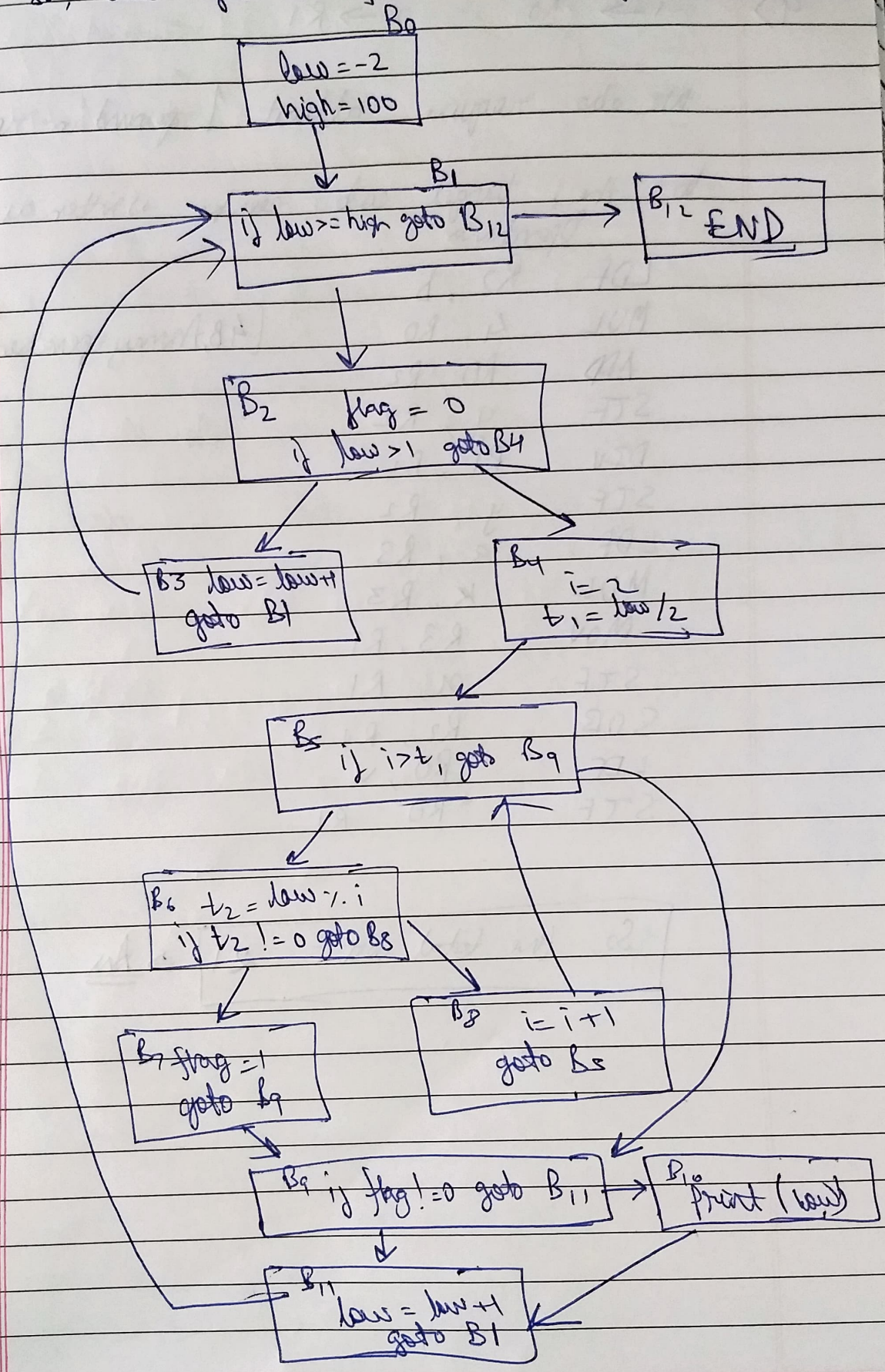
L4: i = i + 1 ---- B_8
goto L3

L5: if flag != 0 goto L6 ---- B_9
print (Low) ---- B_{10}

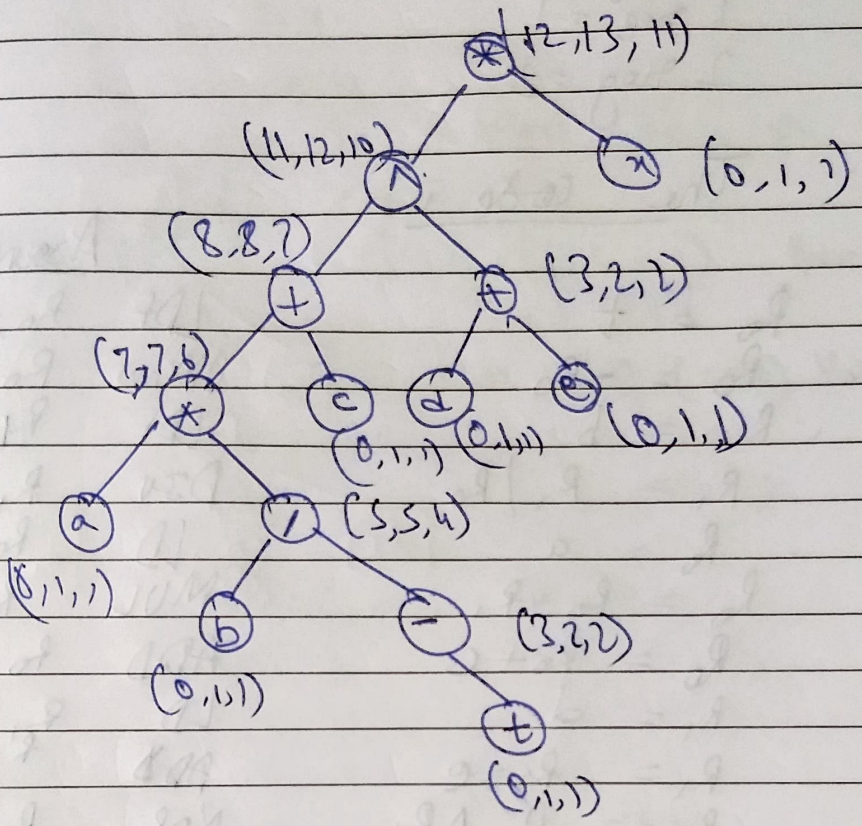
L6: low = low + 1 ---- B_{11}
goto L1

L7: END ---- B_{12}

So, we get the flow control graph as follows:



6.) $(a+b/-t)+c\wedge(d+e)*x$



→ for the '+' operator

2 reg = min (8,9,7)
= 7

1 reg = 8

0 reg = 8

For the '^' operator

2 reg = 10

1 reg = 12

0 reg = 11

for * operator :-

2 reg = 11

1 reg = 13

0 reg = 12

for all leaf nodes,

0 reg = 0

1 reg = 1

2 reg = 2

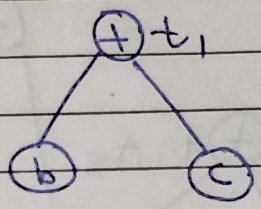
∴ The code is:-

	Assembly:-
$R_0 = t$	LDF R_0, t
$R_0 = -R_0$	UMinus R_0, R_0
$R_1 = b$	LDF R_1, b
$R_1 = R_1 / R_0$	DIV R_1, R_1, R_0
$R_0 = a$	LD R_0, a
$R_0 = R_0 * R_1$	MUL R_0, R_0, R_1
$R_0 = R_0 + c$	ADD R_0, R_0, c
$R_1 = d$	LD R_1, d
$R_1 = R_1 + e$	ADD R_1, R_1, e
$R_0 = R_0 \wedge R_1$	XOR R_0, R_0, R_1
$R_0 = R_0 * x$	MUL R_0, R_0, x

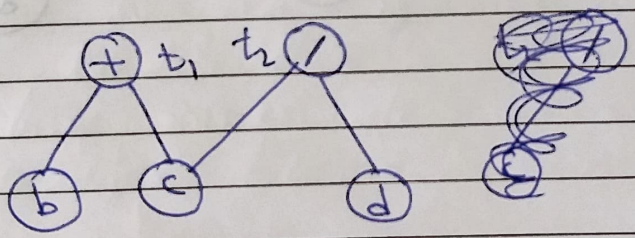
7> Basic Block:-

$t1 := b + c$
$t2 := c / d$
$t3 := b + c$
$b := c / d$
$d := b + c$

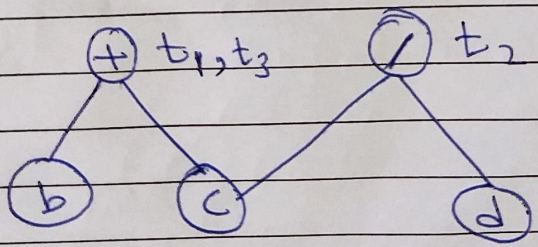
So, Step 1:- $t1 := b + c$



Step 2:- $t2 := c / d$

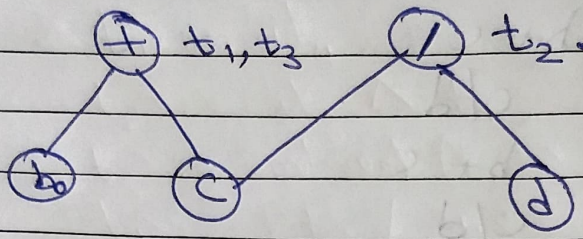


Step 3:- $t3 := b + c$



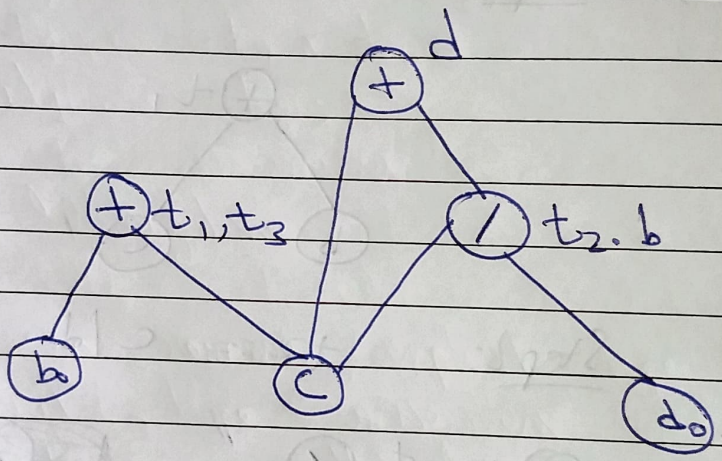
P.T.O.

Step 4:- $b := c / d$, c/d stored in t_2



[∵ Attaching 'b' to t_2]

Step 5:- $d := b + c$



8.) 3-AC in Quadruples Representation

z : a[m] ^ (b* - m + 1)/c * (d%e) - e.

Here ^ → Bitwise XOR.

So, TAC:-

$t_1 = m$;
$t_1 = 4 * t_1$;
$t_2 = a[t_1]$;
$t_3 =$ uminus (m) ;
$t_4 = b * t_3$;
$t_5 = t_4 + 1$;
$t_6 = t_5 / c$;
$t_7 = e$;
$t_8 = d \% t_7$;
$t_9 = t_6 * t_8$;
$t_{10} = t_9 - t_7$;
$t_{11} =$ t_2 bitexor t_{10} ;
$z = t_{11}$;

⇒

9.) $i \rightarrow R_0$ $x \rightarrow R_1$

We also require atleast 1 operand in register.

So, the target code can be written as:-

Operations			cost
LDF	R2, b		2
MUL	4, R0	[4B, Array operation]	2
ADD	R0, R2		1
STF	y, R2		2
DIV	R1, R2		1
STF	y, R2		2
LDF	a, R3		2
MUL	K, R3		2
MOV	R3, R1		1
STF	x, R1		2
SUB	R2, R1		1
LDF	R0, v		2
STF	*R0, R1		1

So, the total cost = 21 ⇒ Ans

10) d_1 : j=1

d_2 : a=2

d_3 = i=m

d_4 = k=2

while i<j and j<n or k<n do

d_5 : c=1

if (c<j) then

d_6 : x = x+b

else if (c>a) then

d_7 : x = 2+c ;

d_8 : a = a+1 ;

else

d_9 : x = c*a ;

d_{10} : a = a-1 ;

d_{11} : c = c+1 ;

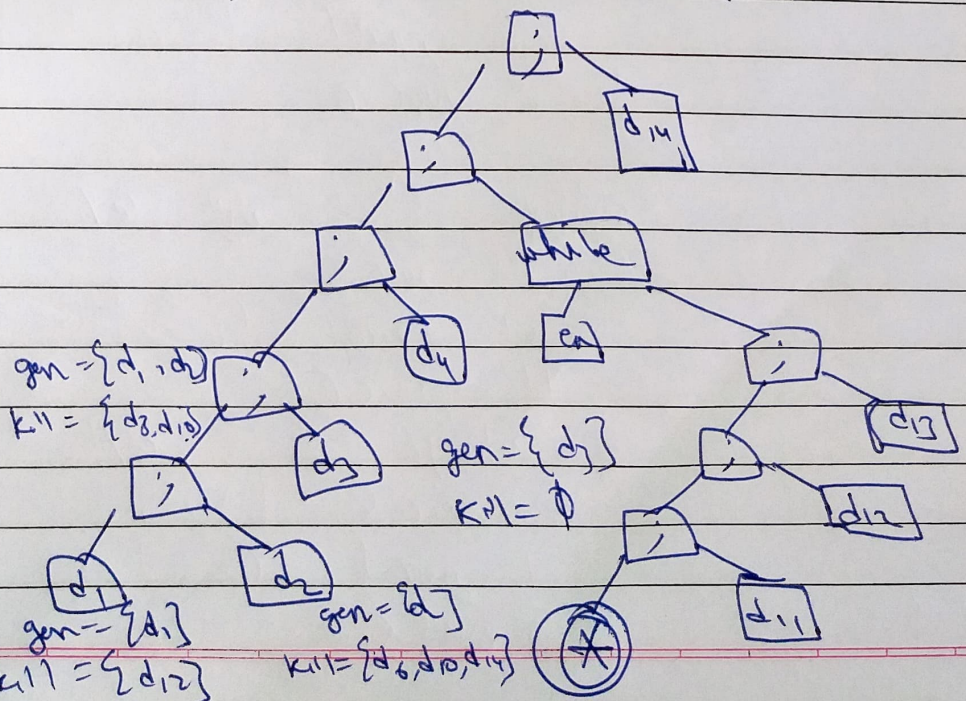
d_{12} : j = j+1 ;

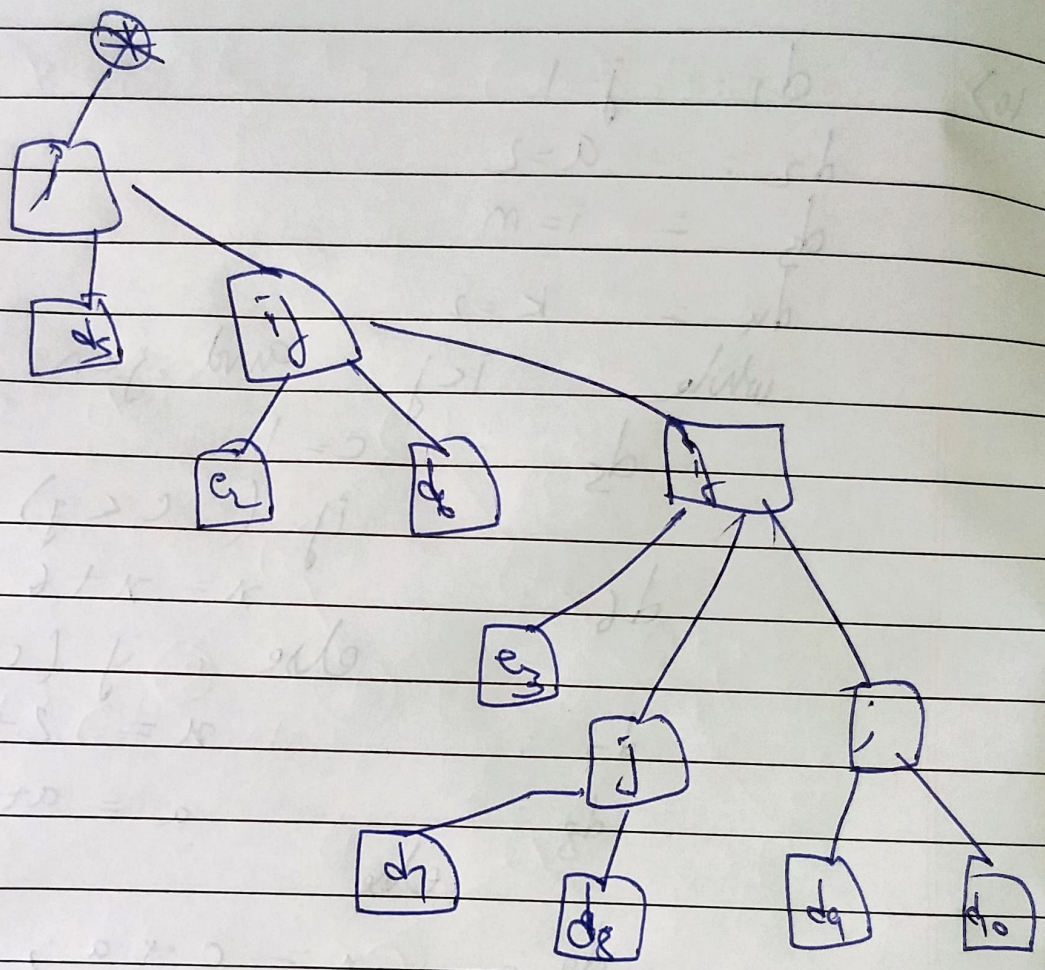
d_{13} : k = k*2 ;

end

d_{14} : a = x+n.

So, tree can be drawn as :-




k
i
e
d
e
j
h
d
d
d
d