Q.1 Attribute translation grammar to generate AST.

The grammar used is as follows:

Exp-> E {print_tree(E.nodept)}

E 1-> E2 + T {E1.nodept=generate_new('+',E2.nodept,T.nodept)}

-> E2 – T {E1.nodept=generate_new('-',E2.nodept,T.nodept)}

->T {E1.nodept=T.nodept}

T1 ->T2 * F {T1.nodept=generate_new('*',T2.nodept,F.nodept)}

->T2/F {T1.nodept=generate_new('/',T2.nodept,F.nodept)}

->T2%F {T1.nodept=generate_new('%',T2.nodept,F.nodept)}

->F {T1.nodept=F.nodept}

F -> (E) {F.nodept=E.nodept}

->NUMBER {F.nodept=NUMBER.nodept}

Here

Generate_new(char,node * left ,node* right) : Function that creates new node with content as char , left and right child as second and third argument respectively.

Print_tree(node * root) : Prints inorder of the AST generated.

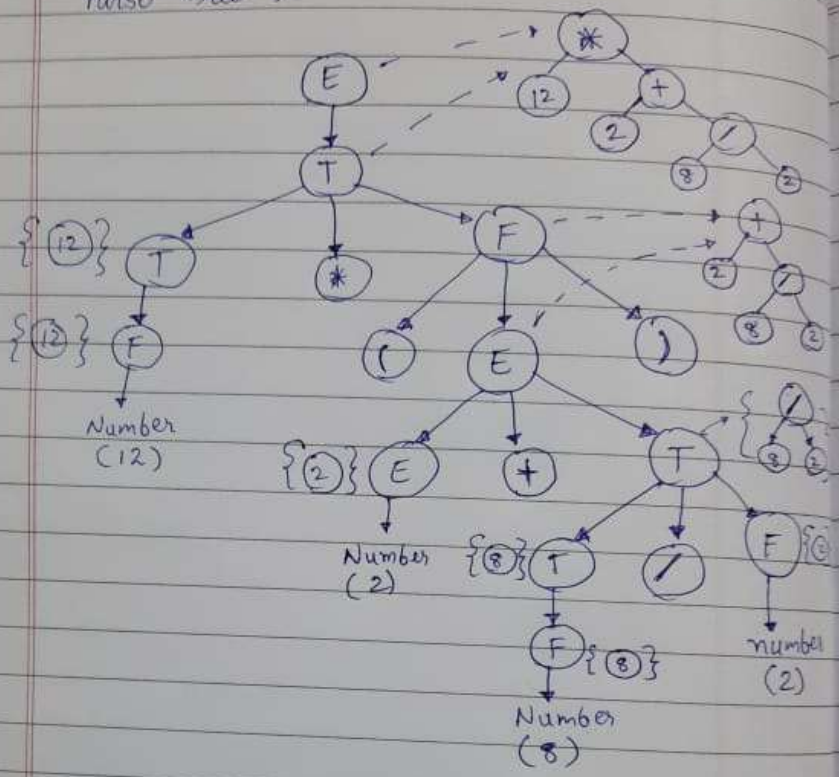
Example:

12*(2+8/2)

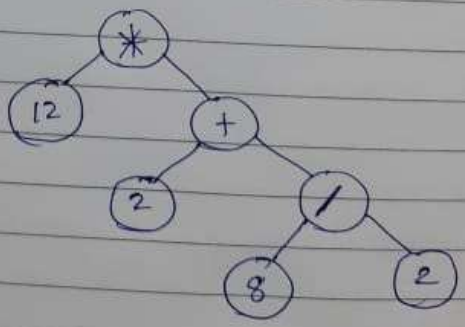
```
aneri@DESKTOP-64GOF59:/mnt/d/semester7/cd_lab/cd_lab9/ast$ ./a.out
12*(2+8/2)

Result
12
*
2
+
8
/
2
```

Parse Tree :-



Final AST :



Q.2. Attribute translation grammar to generate 3 Address Code.

The grammar is as follows:

Exp -> E {print (new_temp()=E.value)}

E1 -> E2 + T {t=new_temp() ; print(t=E2.value+T.value); E1.value=t;}

-> E2 - T {t=new_temp() ; print(t=E2.value-T.value); E1.value=t;}

->T {E1.value=T.value}

T1 -> T2 * F {t=new_temp() ; print(t=T2.value * F.value); T1.value=t;}

-> T2 / F {t=new_temp() ; print(t=T2.value / F.value); T1.value=t;}

-> T2 % F {t=new_temp() ; print(t=T2.value % F.value); T1.value=t;}

->F {T1.value=F.value}

F -> (E) {F.value=E.value}

->NUMBER {F.value=lexeme(NUMBER)}

->ID {F.value=lexeme(ID)}

Here

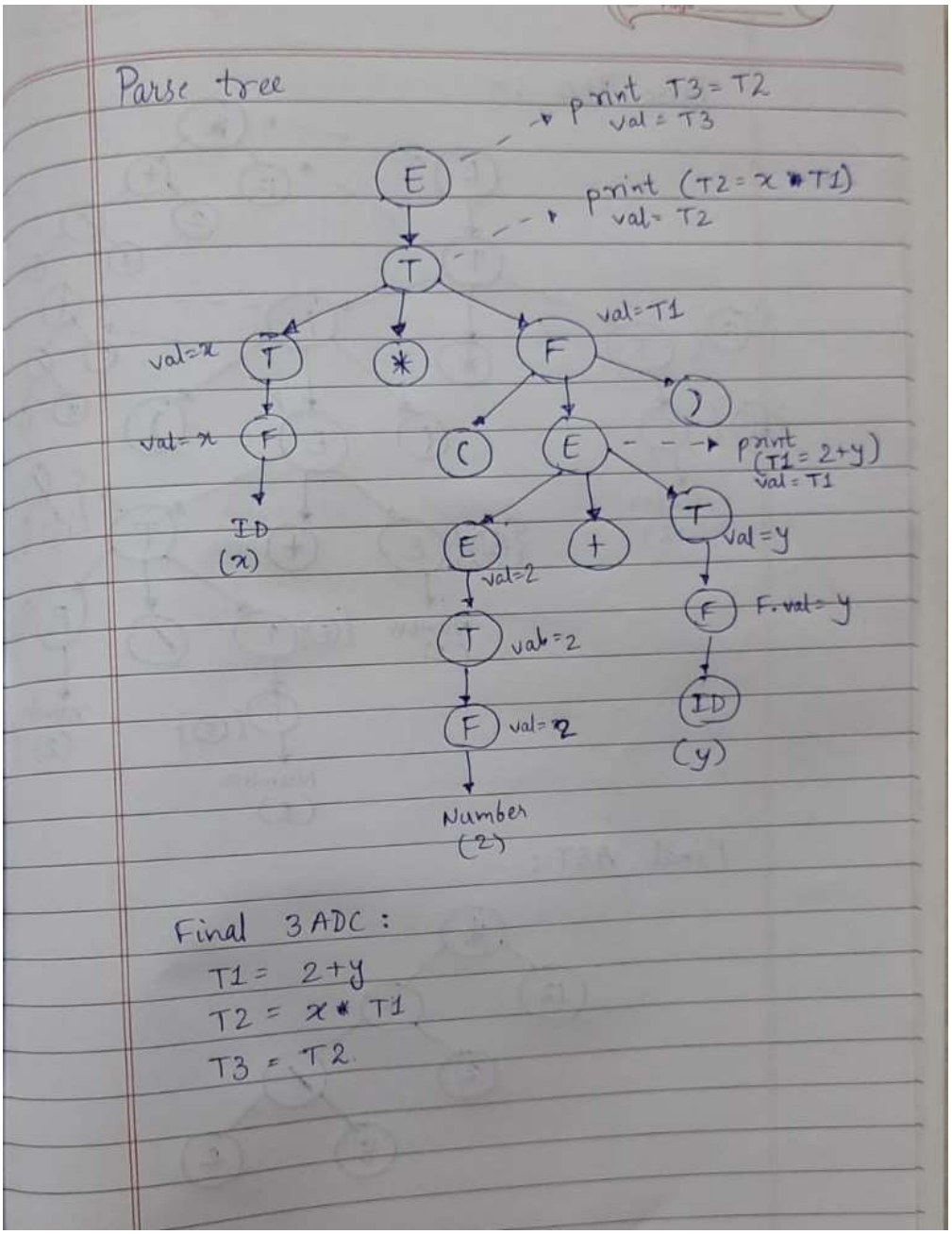
3 Address code are printed as they are generated.

New_temp(): Gives a string that represents new temporary variable name.

Example:

X*(2+y)

```
aneri@DESKTOP-64GOF59:/mnt/d/semester7/cd_lab/cd_lab9/3_add_code$ ./a.out
x * (2 + y)
A = 2 + y
B = x * A
C = B
```



Q.3 Attribute translation grammar to generate DAG.

The grammar is as follows:

Exp-> E {postorder(E.nodept)}

E 1-> E2 + T {E1.nodept=get_node('+',E2.nodept,T.nodept)}

-> E2 – T {E1.nodept=get_node('-',E2.nodept,T.nodept)}

->T {E1.nodept=T.nodept}

T1 ->T2 * F {T1.nodept=get_node('*',T2.nodept,F.nodept)}

->T2/F {T1.nodept=get_node('/',T2.nodept,F.nodept)}

->T2%F {T1.nodept=get_node('%',T2.nodept,F.nodept)}

->F {T1.nodept=F.nodept}

F -> (E) {F.nodept=E.nodept}

->NUMBER {F.nodept=NUMBER.nodept}

Here

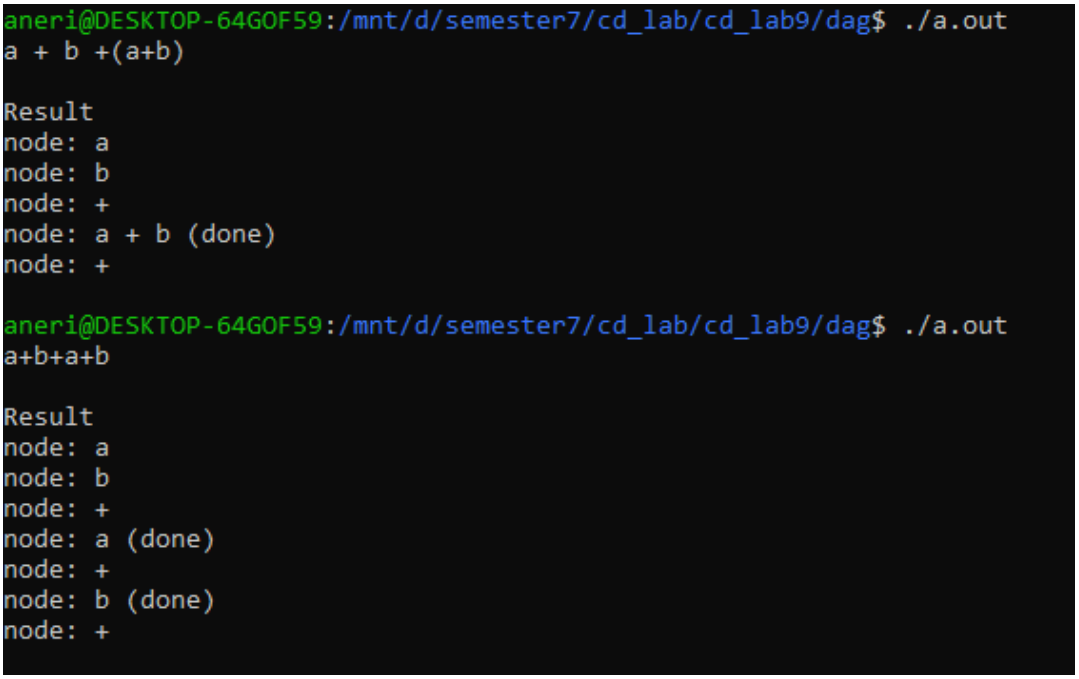
Get_node(): If expression is already calculated, returns existing pointer to the node else creates a new node with content as first argument and left and right child as second and third argument.

Postorder(): Prints the nodes in postorder. If the node is already visited and is repeated in post order, (done) is printed beside it.

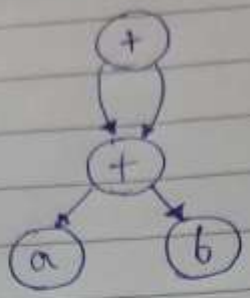
Eg:

(i)a + b +(a+b)

(ii) a+b+a+b



(i)



(ii)