Handling ambiguous grammar:-

Ambiguous grammar:-

A context free grammar 'G' is said to be ambiguous grammar if and only if it creates (or) generates more than one parse tree.

As we have seen various parsing methods in which if at all grammar is ambiguous then it creates conflics and we can not parse the input string which such ambiguous grammar. If we implement ambiguous grammar we can add any new productions for special constructs.

YACC:-

YACC is a compiler from source to destination Here YACC stands for "yet Another Compiler Compiler" it is similar to Lex any compiler (or) programmer can convert the YACC program to 'c' compiler of a program

Now 'C' compiler can convert the c program to Machine Level language, i.e., a.out.

9/09/14

UNIT - 4

Semantic Analysis

Imp *** Imp

→ Intermediate forms of source program
  * Abstract syntax tree
  * polish notation
  * Three - address code.

→ Attribute grammar

→ Syntax directed transulation

→ Conversion of popular programming languages, Language construct into intermidiate code form

→ Type checker

Intermediate forms of source program :-

Intermediate source program some times called as intermediate code form here we implement

1) Abstract syntax tree
2) polish Notation
3) Three - address code

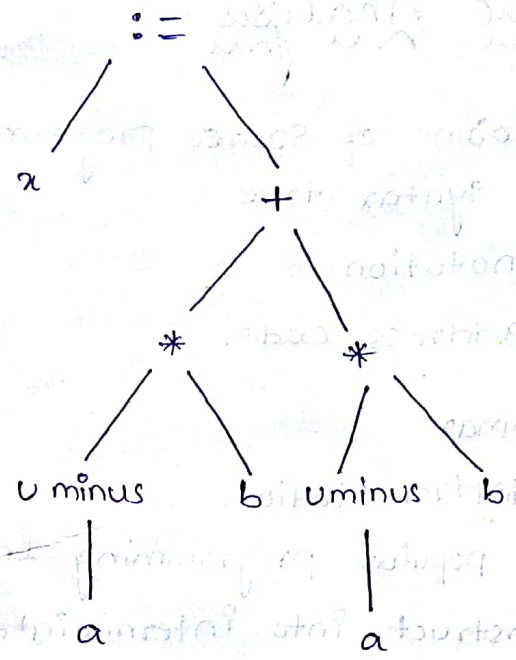
Abstract syntax tree :-

In abstract syntax tree we implement syntax tree and DAG (Direct Acyclic Graph)

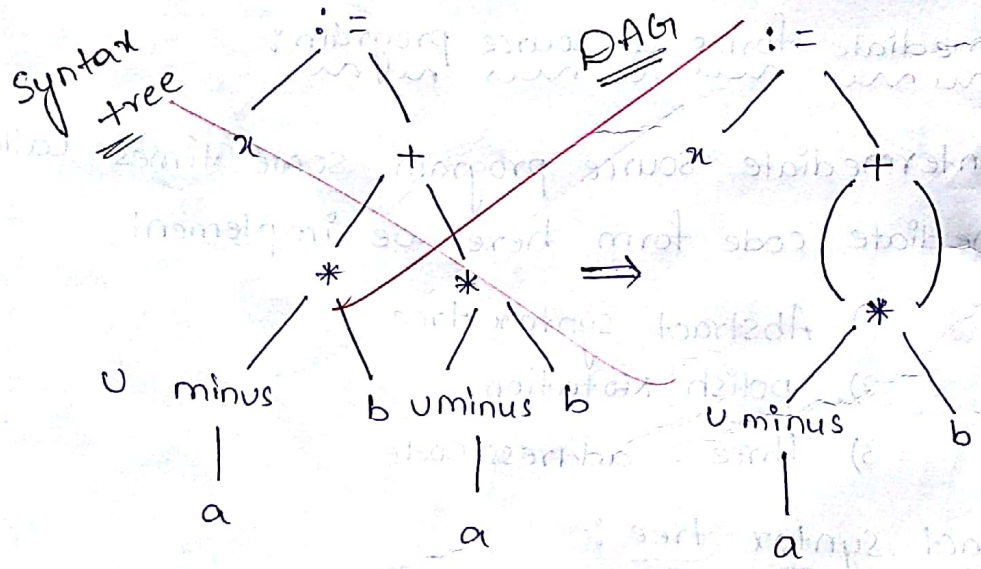
Eg:- $x := -a * b + -a * b$

(a*b) + (a*b)

Syntax tree :-



In above syntax tree (or) parse tree can be converted into DAG representation as shown in bellow.



polish notation :-

polish notation is nothing but postfix notation

Example for the polish notation is follows.

Eg:- $(a+(b*c)) \uparrow d-e/(f+g)$

$(a + \underbrace{(b*c)}_{T_1}) \uparrow d - e / (f+g)$

$(\underbrace{a + T_1}_{T_2}) \uparrow d - e / (f+g)$

$\underbrace{T_2 \uparrow d}_{T_3} - e / (f+g)$

$T_3 - e / \underbrace{(f+g)}_{T_4}$

$T_3 - \underbrace{e / T_4}_{T_5}$

$\underbrace{T_3 - T_5}_{T_6}$

T_6

$T_1 = b*c = bc*$

$T_2 = a + T_1 = aT_1 +$

$T_3 = T_2 \uparrow d = T_2 d \uparrow$

$T_4 = f+g = fg+$

$T_5 = e / T_4 = eT_4 /$

$T_6 = T_3 - T_5 = T_3 T_5 -$

T_6

$T_3 T_5 -$

$T_3 e T_4 / -$

$T_3 e fg+ / -$

$T_2 \uparrow d efg+ / -$

$aT_1 + \uparrow defg+ / -$

$abc*+ \uparrow defg+ / -$ postfix Notation.

Three-address code :-

Three-address code of a intermediate form of Source program can represent

1) quard triples
2) Triples
3) indirect triples

1) **quard triples :-** In quard triples we implement four items they are

→ operator
→ operand 1 (or) argument 1
→ operand 2 (or) argument 2
→ result

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

$t_1 := u\ minus\ a$
$t_2 := t_1 * b$
$t_3 := u\ minus\ a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$x := t_5$

quard triple form

S.No	operator	operand1	operand 2	result
(0)	u minus	a		t_1
(1.)	*	t_1	b	t_2
(2.)	u minus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

2) Triple :- In triple of 3-address code we implement three items they are.

→ operator
→ operand 1
→ operand 2

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

t_1 : u minus a

t_2 : $t_1 * b$

t_3 : u minus a

t_4 : $t_3 * b$

t_5 : $t_2 + t_4$

x := t_5

S.NO	operator	operand 1	operand 2
(0)	u minus	a.	
(1)	*	(0)	b
(2)	u minus	a.	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	(4)	

3) In- direct triple :-

STATES	
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

S.No	operator	operand 1	operand 2
(0)	u minus	a	
(1)	*	(11)	b
(2)	u minus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	(15)	

1) Write the quard triple, triple, in-direct triple for the statement i) a := b * -c + b * -c

ii) a * -(b+c)

Sol i) Quard triple :

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

t_1 : u minus c

t_2 : b * t_1

t_3 : u minus c

t_4 : b * t_3

t_5 : t_2 + t_4

x := t_5

S.No	operator	operand 1	operand 2	Result
(0)	u minus	c		t_1
(1)	*	b	t_1	t_2
(2)	u minus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triple :-

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

$t_1 := u\ minus\ c$

$t_2 := b * t_1$

$t_3 := u\ minus\ c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$x := t_5$

S.No	operator	operand 1	operand 2
(0)	u minus	c	
(1)	*	b	(0)
(2)	u minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	(4)	

In-direct triple :-

STATES	
(0)	(A)
(1)	(B)
(2)	(C)
(3)	(D)
(4)	(E)
(5)	(F)

S.No	operator	operand 1	operand 2
(0)	u minus	c	
(1)	*	b	(A)
(2)	u minus	c	
(3)	*	b	(C)
(4)	+	(B)	(D)
(5)	:=	(E)	

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

Sol

quard triple :-

$x := a * -(b+c)$

t_1 : a

t_2 : c

t_3 : $(b + t_2)$

t_4 : u minus t_3

t_5 : $t_1 * t_4$

$x := t_5$

S.No	operator	operand 1	operand 2	Result
(0)		a		t_1
(1)		c		t_2
(2)	+	b	t_2	t_3
(3)	u minus	t_3		t_4
(4)	*	t_1	t_4	t_5
(5)	:=	t_5		x

Triple :-

$x := a * -(b+c)$

$t_1 : a$
$t_2 : c$
$t_3 : b+c$
$t_4 : u\ minus\ t_2$
$t_5 : t_1 * t_4$
$x := t_5$

S.No	Operator	Operand 1	Operand 2
(0)		a	
(1)		c	
(2)	+	b	(1)
(3)	u minus	(2)	
(4)	*	(0)	(3)
(5)	:=	(4)	

In-direct triple :-

STATES	
(0)	(11)
(1)	(22)
(2)	(33)
(3)	(44)
(4)	(55)
(5)	(66)

S.No	operator	operand1	operand 2
(0)		a	
(1)	+	b	(22)
(2)	u minus	(33)	
(3)	*	(11)	(44)
(4)	:=	(55)	

Construction of syntax tree :-

Here we are using three functions they are

1) Mknode (op, Left child, Right child);
2) Mk leaf (Id, entry);
3) Mk leaf (numb, val);

Algorithm :-

1) convert given expression into postfix notation
2) Follow the Mknode(), Mk leaf() functions
3) Construct syntax tree by using step 1 & step 2

1) Construct Syntax tree for the following postfix notation

a * b - (c + d)

Sol

Given expression is a * b - (c + d)

P_1 = mk leaf (Id, entry a);

P_2 = mk leaf (Id, entry b);

P_3 = mk node ('*', P_1, P_2);

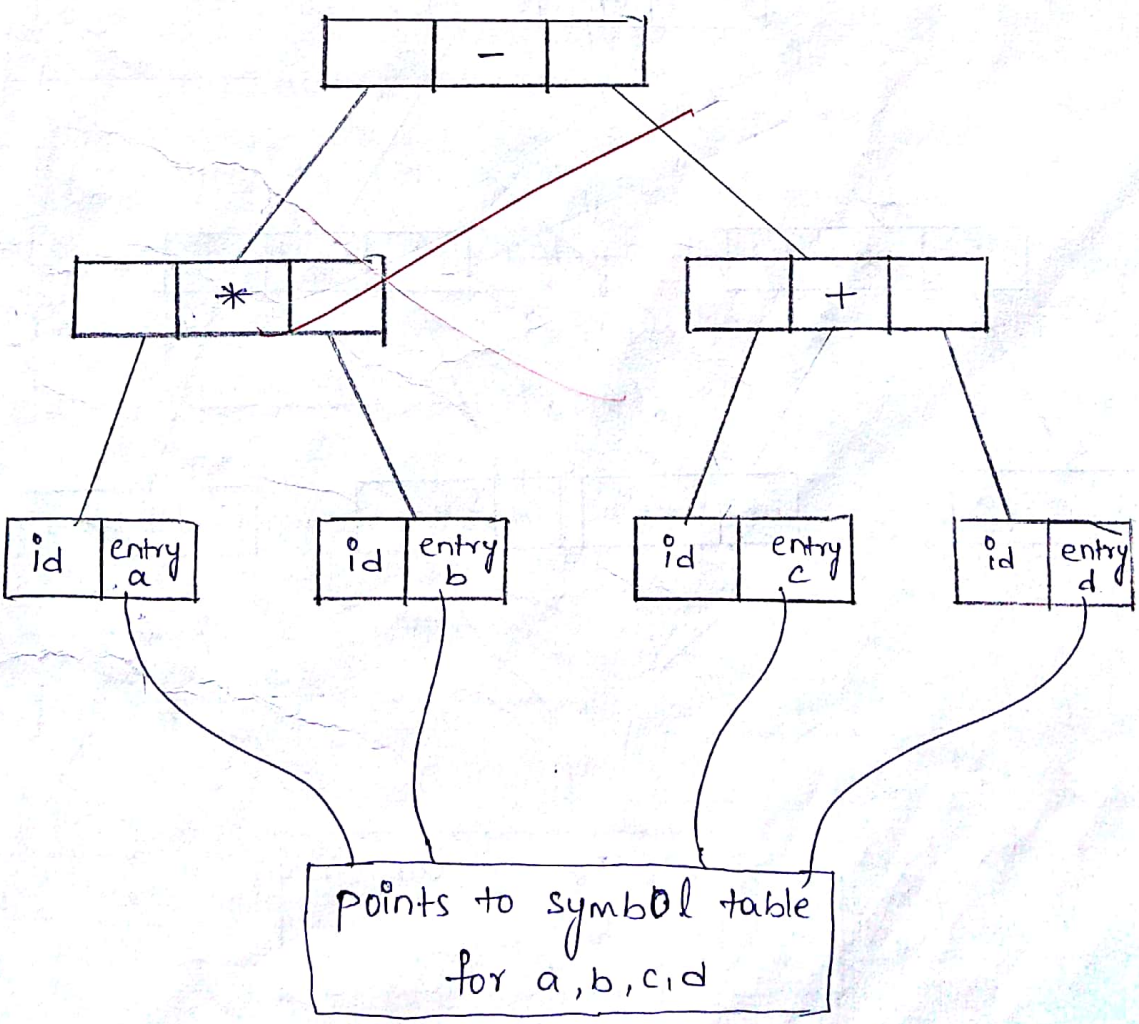
P_4 = mk leaf (Id, entry c);

P_5 = mk leaf (Id, entry d);

P_6 = mk node ('+', P_4, P_5);

P_7 = mk node ('-', P_3, P_6);

Syntax tree :-



2) a*b + (a*b)

P_1 = mk leaf (Id, entry a);

P_2 = mk leaf (Id, entry b);

P_3 = mk ~~leaf~~ node ('*', P_1, P_2);

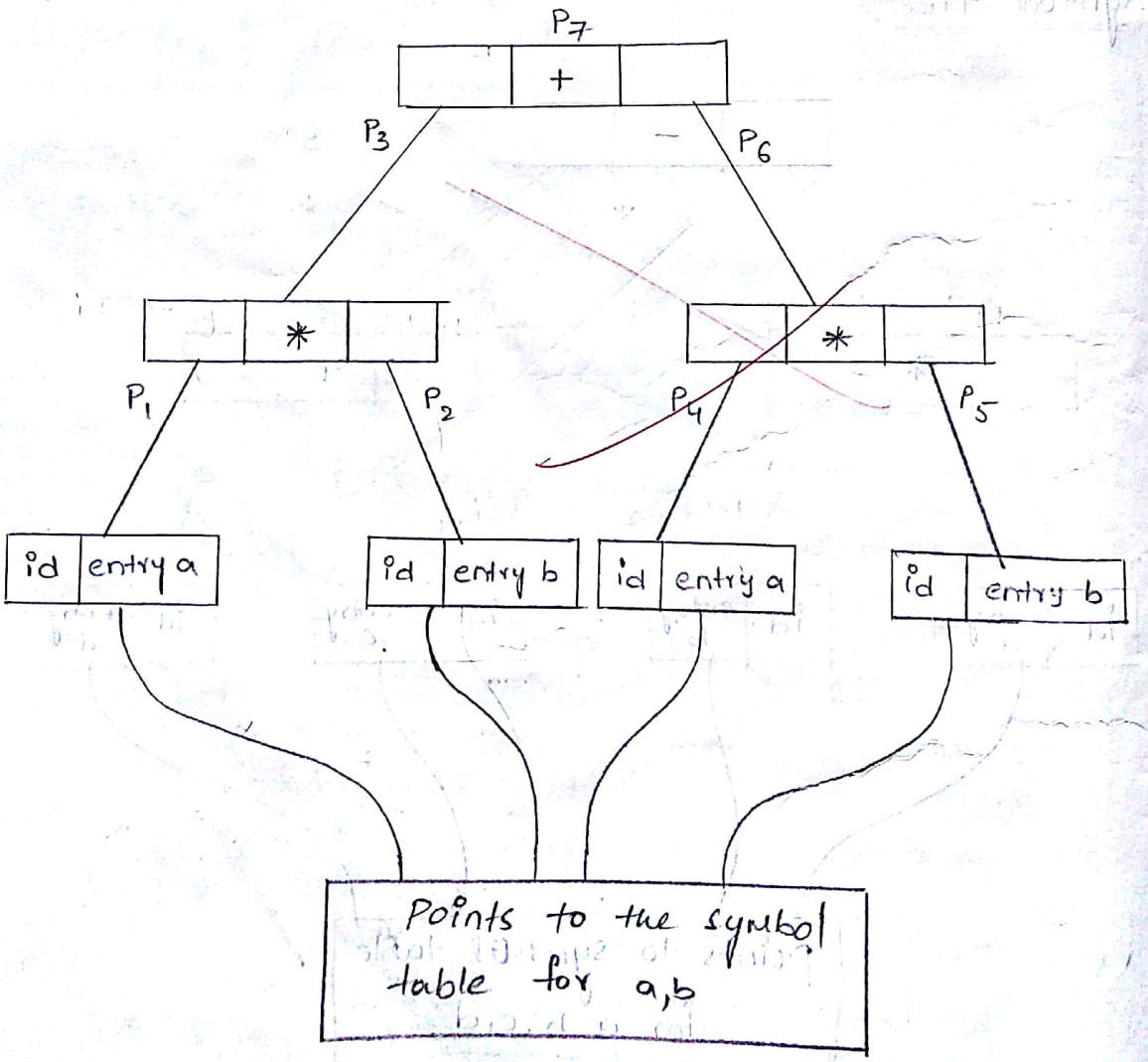
P_4 = mk leaf (Id, entry a);

P_5 = mk leaf (Id, entry b);

P_6 = mknode ('*', P_4, P_5);

P_7 = mknode ('+', P_3, P_6);

Syntax tree :-



1H M ~~***~~ 2003 Sep

3) Construct syntax tree for following expressions

a) a+b*c+d+a+e

b) a+b+c+d+e+f : ab+cd+ef+++

c) aab↑c↑d↑e : aabc↑de↑↑

Sol:-

a) Given expression a+b*c+d+a+e

Convert the expression into postfix notation

ab+cd+ae++*

P_1 = mk leaf (Id, entry a);

P_2 = mk leaf (Id, entry b);

P_3 = mk node ('+', P_1, P_2);

P_4 = mk leaf (Id, entry c);

P_5 = mk leaf (Id, entry d);

P_6 = mk node ('+', P_4, P_5);

P_7 = mk leaf (Id, entry a);

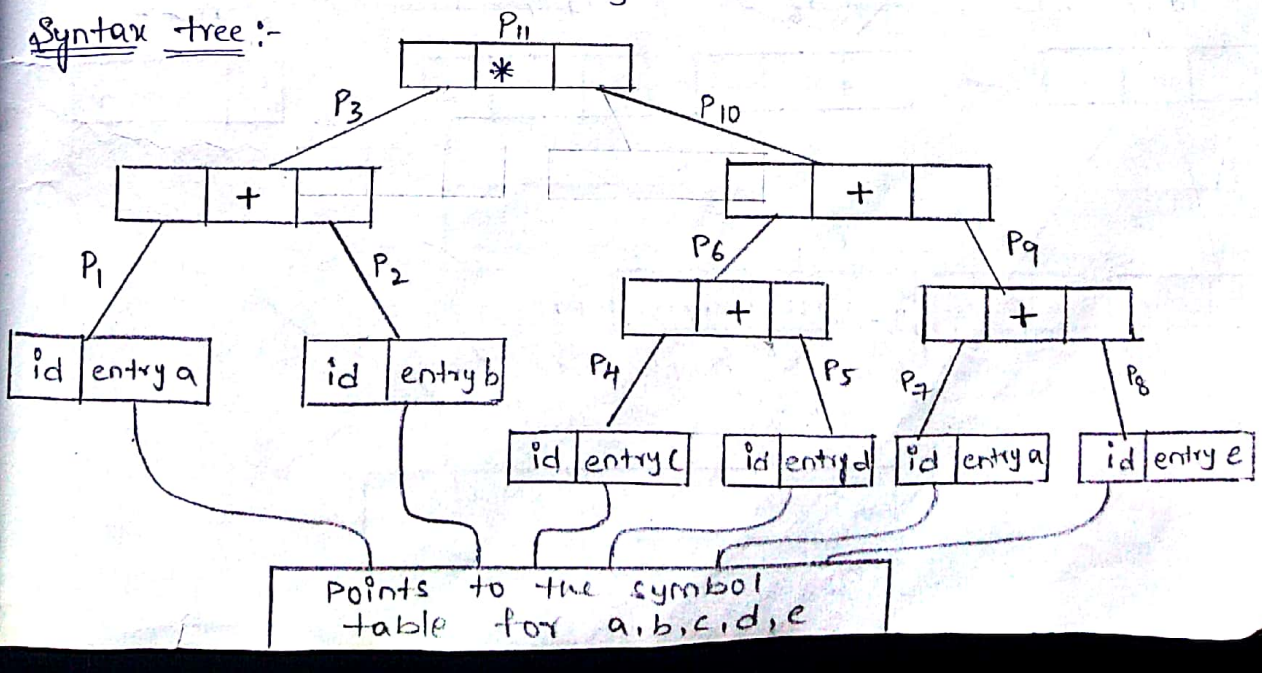
P_8 = mk leaf (Id, entry e);

P_9 = mk node ('+', P_7, P_8);

P_{10} = mk node ('+', P_6, P_9);

P_{11} = Mk node ('*', P_3, P_{10});

Syntax tree:-



b) Sol

Given expression is a+b+c+d+e+f

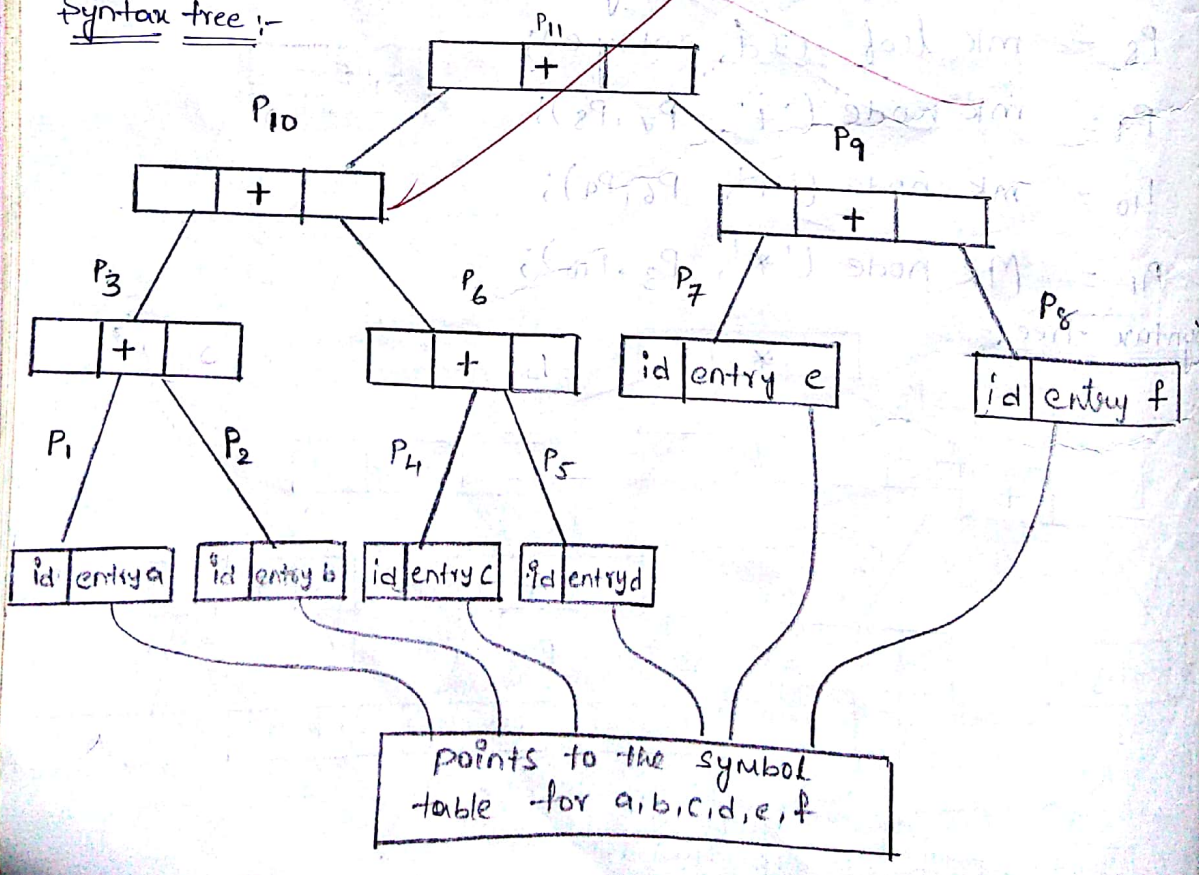
Convert the given expression into the form of postfix notation

ab+cd+ef+++

P_1 = mk leaf (id, entry a)
P_2 = mk leaf (id, entry b)
P_3 = mk node ('+', P_1, P_2)
P_4 = mk leaf (id, entry c)
P_5 = mk leaf (id, entry d)
P_6 = mknode ('+' P_4, P_5)
P_7 = mk leaf (id, entry e)
P_8 = mk leaf (id, entry f)
P_9 = mk node ('+', P_7, P_8)
P_{10} = mk node ('+', P_3, P_6)
P_{11} = mk node ('+', P_{10}, P_9)

c) Sol

Syntax tree :-



c) Given expression is $aab^{\wedge} c \uparrow d \uparrow e$

Sol

Convert the Given expression into the form of postfix notation. $aabc\uparrow de\uparrow\uparrow$

P_1 = mk leaf (id, entry a)

P_2 = mk leaf (id, entry a)

P_3 = mk leaf (id, entry b)

P_4 = mk leaf (id, entry c)

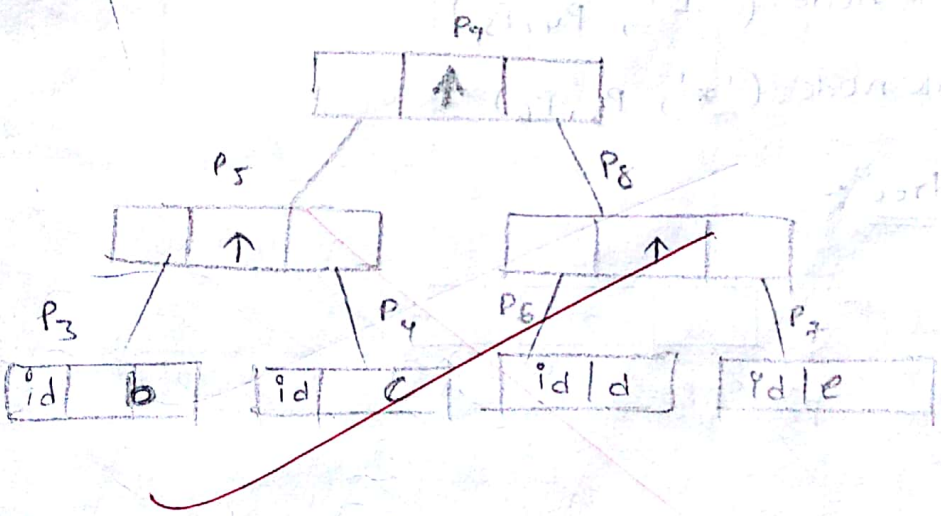
P_5 = mk node ($\uparrow$, P_3, P_4)

P_6 = mk leaf (id, entry d)

P_7 = mk leaf (id, entry e)

P_8 = mk node ($\uparrow$, P_6, P_7)

P_9 = mk node ($\uparrow$, P_5, P_8)



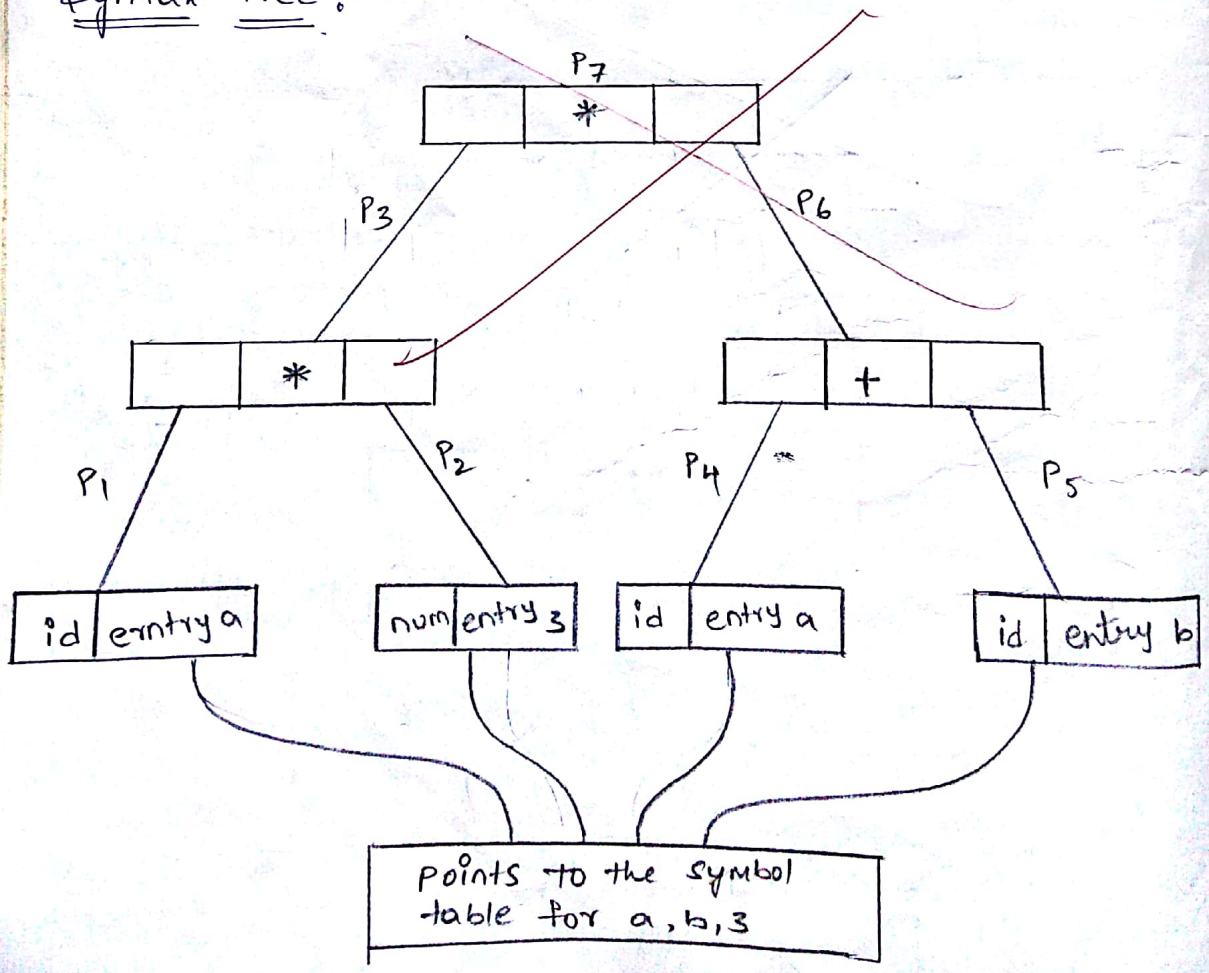
IHM (2009)
Convert the following arithmetic expression into Syntax tree and 3-address code.

i) b * 3 * (a+b)
ii) b * - (a+b)

Sol
i) Given that b * 3 * (a+b)

P_1 = mk leaf (Id, entry b)
P_2 = mk leaf (num, entry 3)
P_3 = mk node ('*', P_1, P_2)
P_4 = mk leaf (Id, entry a)
P_5 = mk leaf (Id, entry b)
P_6 = mk node ('+', P_4, P_5)
P_7 = mk node ('*', P_3, P_6)

Syntax tree :-



3-address code :-

quadruple :-

t_1 : b

t_2 : t_1 * 3

t_3 : a

t_4 : t_3 + b

t_5 : t_2 * t_4

x := t_5

S.NO	operator	operand 1	operand 2	result
(0)		b		t_1
(1)	*	t_1	3	t_2
(2)		a		t_3
(3)	+	t_3	b	t_4
(4)	*	t_2	t_4	t_5
(5)	:=	t_5		x

Triple

S.NO	operator	operand 1	operand 2
(0)		b	
(1)	*	(0)	3
(2)		a	
(3)	+	(2)	b
(4)	*	(1)	(3)
(5)	:=	(4)	

Indirect triple :-

(0)	(A)
(1)	(B)
(2)	(C)
(3)	(D)
(4)	(E)
(5)	(F)

(0)		b	
(1)	*	(A)	3
(2)		a	
(3)	+	(C)	b
(4)	*	(B)	(D)
(5)	:=	(E)	

Direct Acyclic Graph (DAG) for expression :-

A DAG for an expression defines the common sub expressions in the expression

Eg :- $a+a*(b-c)+(b-c)*d$ construct the DAG.

Sol

$P_1 = Mk\ leaf\ (Id, entry\ a)$

$P_2 = Mk\ leaf\ (Id, entry\ a)$

$P_3 = Mk\ leaf\ (Id, entry\ b)$

$P_4 = Mk\ leaf\ (Id, entry\ c)$

$P_5 = mk\ node\ ('-', P_3, P_4)$

$P_6 = Mk\ node\ ('*', P_2, P_5)$

$P_7 = mk\ node\ ('+', P_1, P_6)$

$P_8 = mk\ leaf\ (Id, entry\ b)$

$P_9 = mk\ leaf\ (Id, entry\ c)$

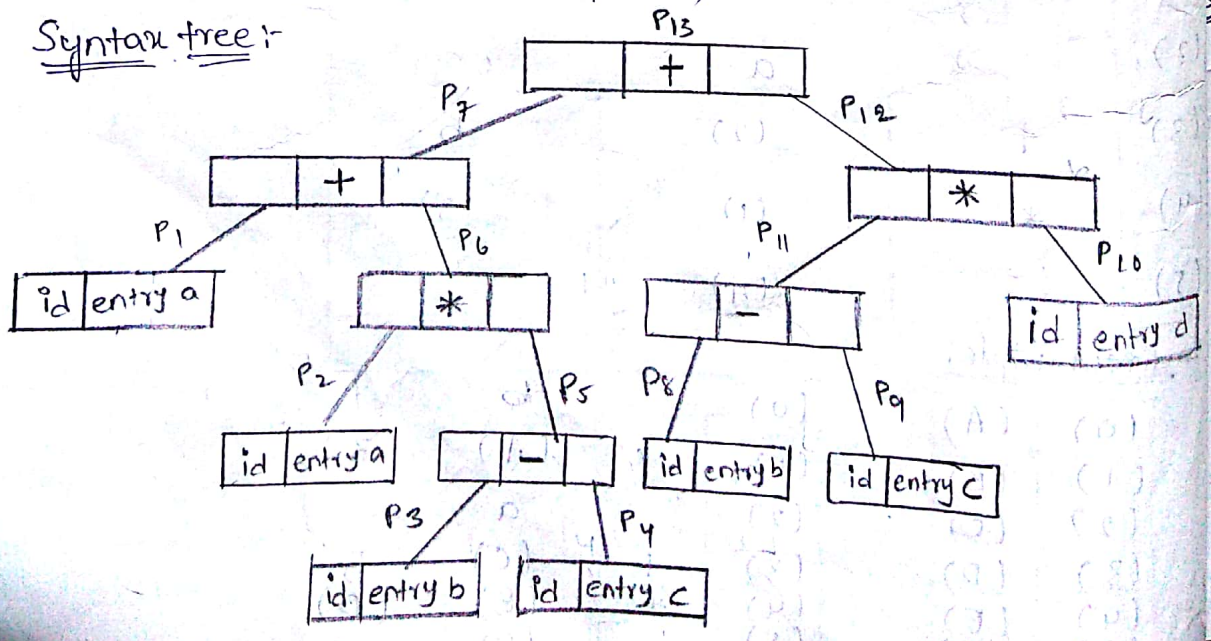
$P_{10} = mk\ leaf\ (Id, entry\ d)$

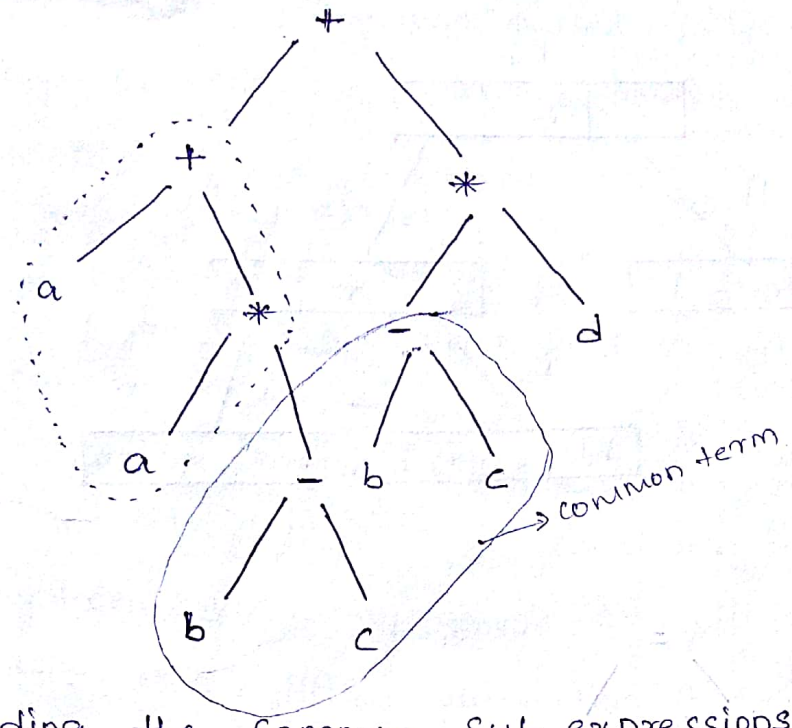
$P_{11} = mk\ node\ ('-', P_8, P_9)$

$P_{12} = mk\ node\ ('*', P_{11}, P_{10})$

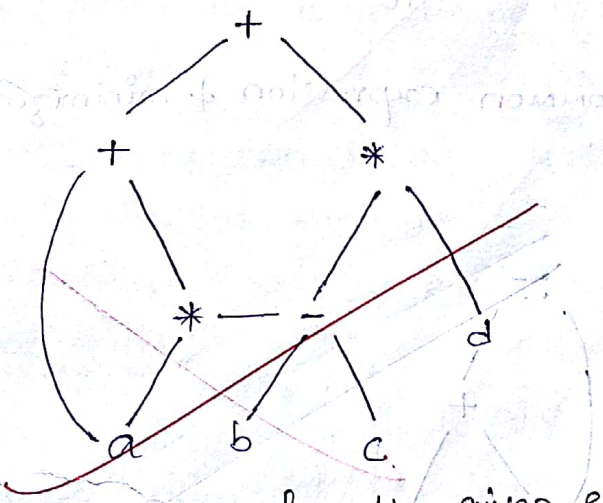
$P_{13} = mk\ node\ ('+', P_7, P_{12})$

Syntax tree :-





finding the common sub expressions and minimize the DAG tree.



7M
2) Draw the DAG tree for the given expression $i = i + 10$

Sol Given that $i = i + 10$

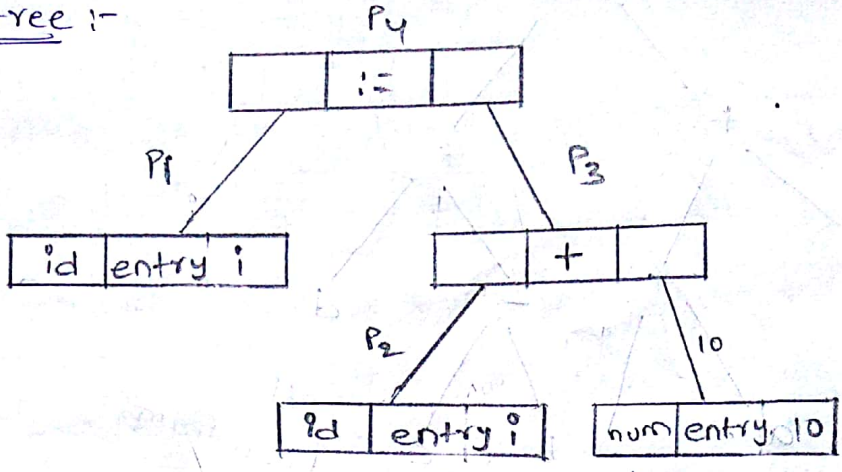
$P_1 =$ mkleaf (Id, entry i)

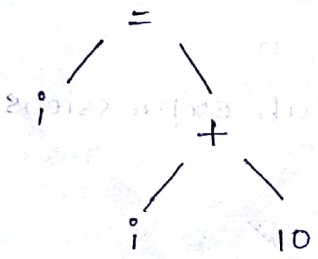
$P_2 =$ mkleaf (Id, entry i)

$P_3 =$ mknode ('+', P_2, 10)

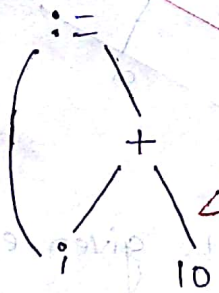
$P_4 =$ mknode ('=', P_1, P_3)

Syntax tree :-



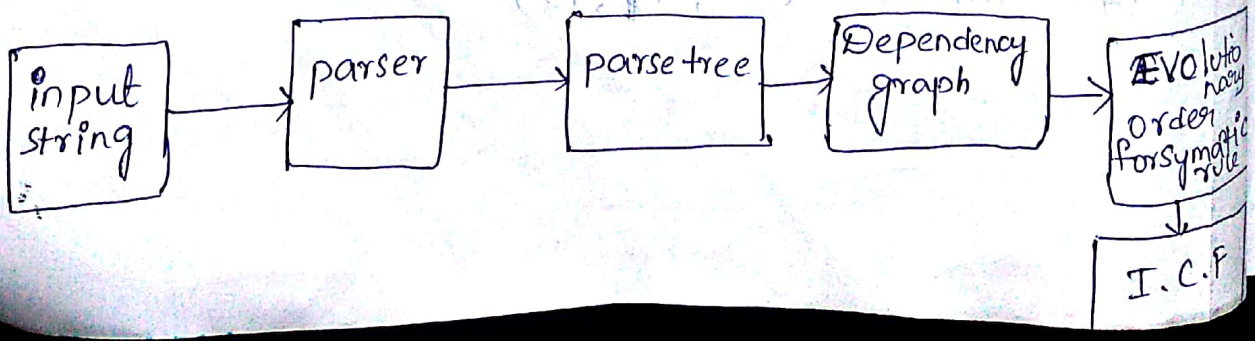


Finding the common expression & minimizing the DAG tree.



Syntax Directed Translation :-

SDT is a process ~~f~~ to ~~Construct~~ convert the i/p string into intermediate code form.



→ At first i/p string can parsed after parsing the i/p string that can submitting to parser. then the parser will generates.

→ By using parse tree the dependency graph is obtained.

→ In dependency graph each and every node symantic rule can be evoluated.

→ Any compiler (or) programmer can convert the input string to intermediate code generator by using parser, parsing tree, dependency graph, evalutionary order for symantic rule, DAG, construction of syntax tree.

Syntax directed definition :- (SDD)

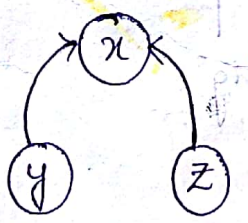
Attribute :- Attribute is a name of identifier, variable, constant or string.

The attributes can be classified into two types.

1) Synthesized attribute.
2) Inherited attribute.

Synthesized attribute :-

If we obtain the attribute value of one node by using attribute value of its children than it is called Synthesized attribute.



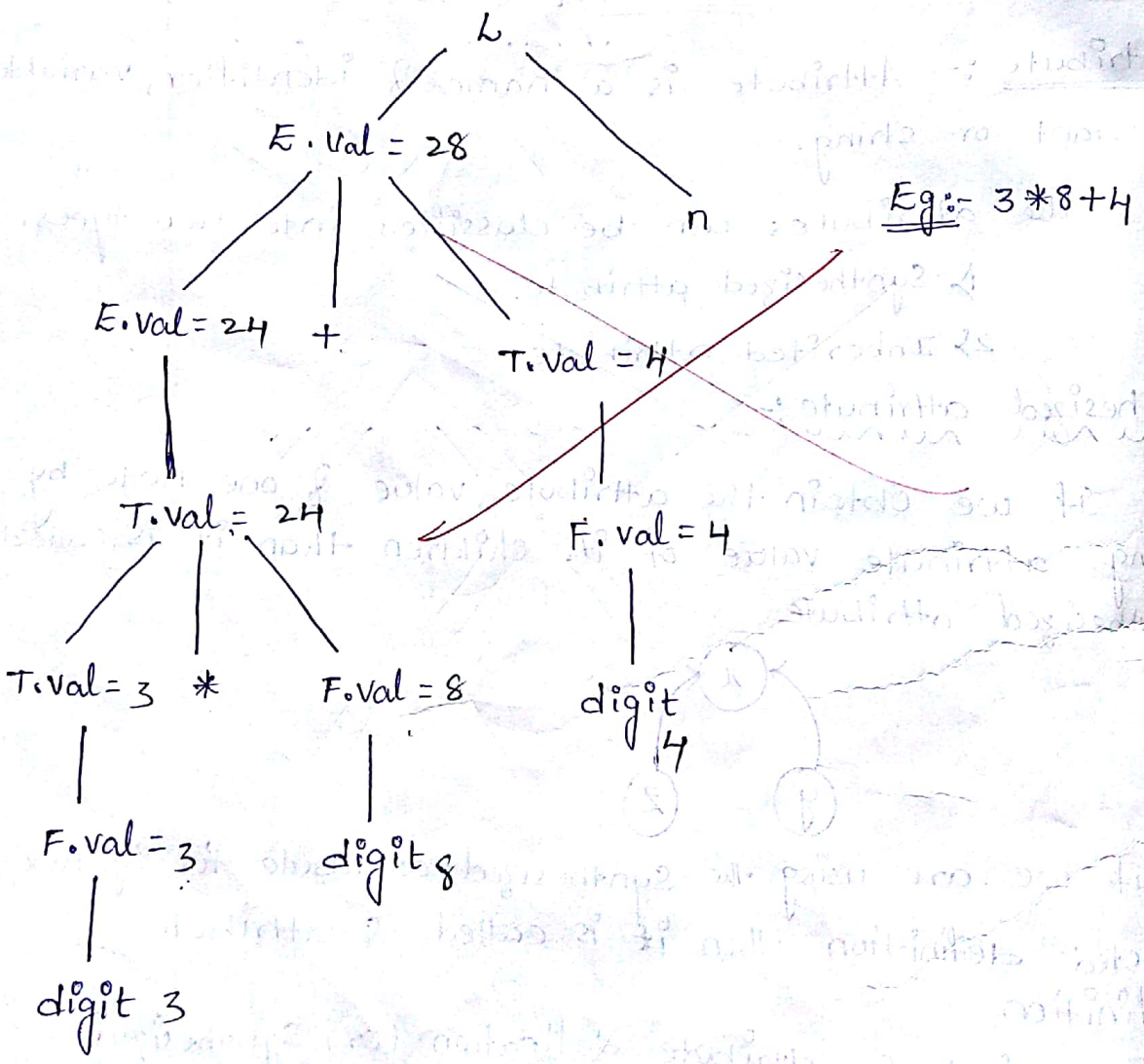
→ If we are using the Synthesized attribute for syntax directed definition then it is called S-attribute definition.

* Eg:- Find S-attribute defination (or) Synthesized attribute of an production

$L \to E_n$
$E \to E_1 + T$
$E \to T$
$T \to T_1 * F$
$T \to F$
$F \to (E)$
$F \to digit$

Sol Given productions

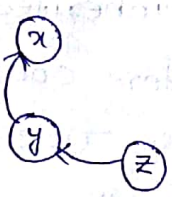
production	Semantic value
$L \rightarrow E_n$	print (E.value)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := (E.val)
$F \rightarrow$ digit	F.val := digit.Lexvalue



Eg:- 3*8+4

Inherited attribute :-

The attribute value of one node can be obtained attribute value of it's parent & sibling

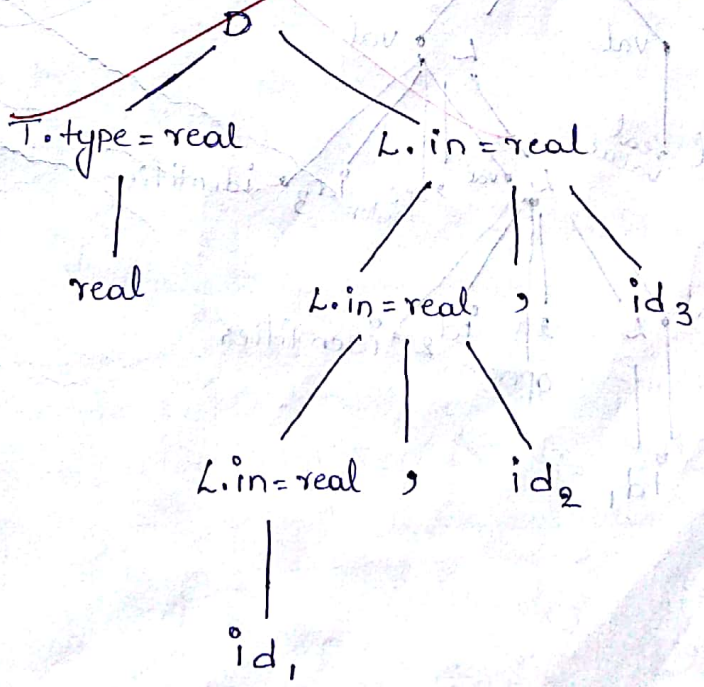


Eg:- Find the inherited attribute for the following production.

$D \rightarrow TL$
$T \rightarrow int$
$T \rightarrow real$
$L \rightarrow L_1, id$
$L \rightarrow id$

Sol

Production	Semantic value.
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow int$	$T.type := integer$
$T \rightarrow real$	$T.type := real$
$L \rightarrow L_1, id$	$\{L_1.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$



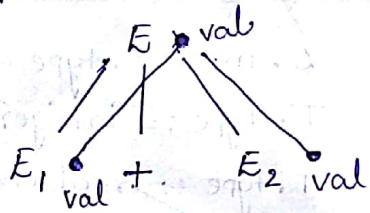
Dependency graph:-

A graph 'G' is said to be dependency graph it contains synthesized and inherited attributes at the nodes in a parse tree. Before constructing the dependency graph for a parse tree we put each semantic rule of the given production.

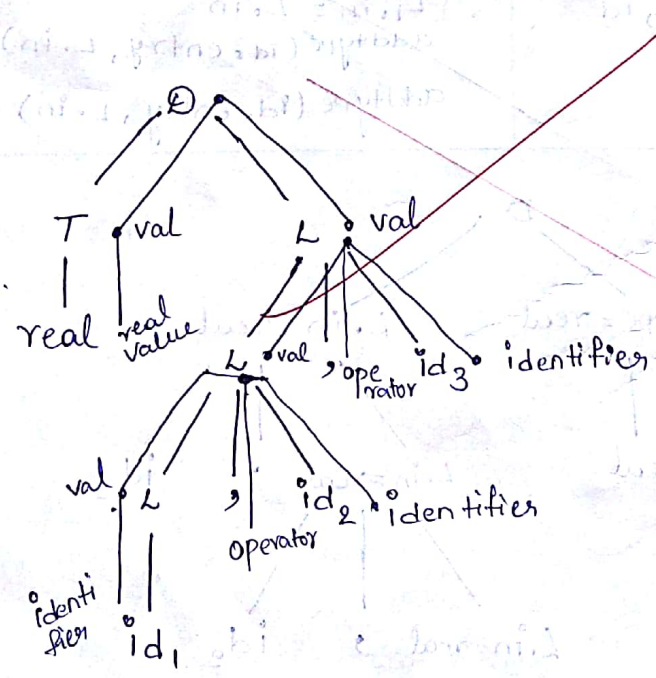
Eg:-1

production	Semantic rule
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$

dependency graph



Eg:-2



write extra

Conversion of Popular programming languages construct into intermidiate code form :-

***
7M

1) write a transilator schema to covert arithematic expression into 3-address code ?

Sol Let consider any CFG i.e.,

$E \to E_1 + T$
$E \to T$
$T \to T_1 * F$
$T \to F$
$F \to (E_1)$
$F \to \text{digit}$

} Arithematic expression Grammar (AEG)

Then impliment syntax directed transulation schema for the above grammar is

production	Symantic rule
$E \to E_1 + T$	$E.Val = E_1.val + T.val$
$E \to T$	$E.val = T.Val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E_1)$	$F.val = (E_1.val)$
$F \to digit$	$F.val = digit.Lexical value$

Thus, the three-address code make use of tempararies to hold intermediate result of an expression. the syntax directed transulation for converting arithemetic expression into 3-address code is given bellow.

production	Semantic rule for arithmatic expressions into 3-address code
$E \to E_1 + T$	{ t_1 = generate temporary( ); generate code ('+', $E_1.val$, $T.val$) $E.value = t_1$ }

$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	{ t_2 = generate temporary(); generate code('*', $T_1.val$, F.val) $T.value = t_2$ }
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E_1)$	$F.val = (E_1.val)$
$F \rightarrow$ digit	F.val = digit.lexme value