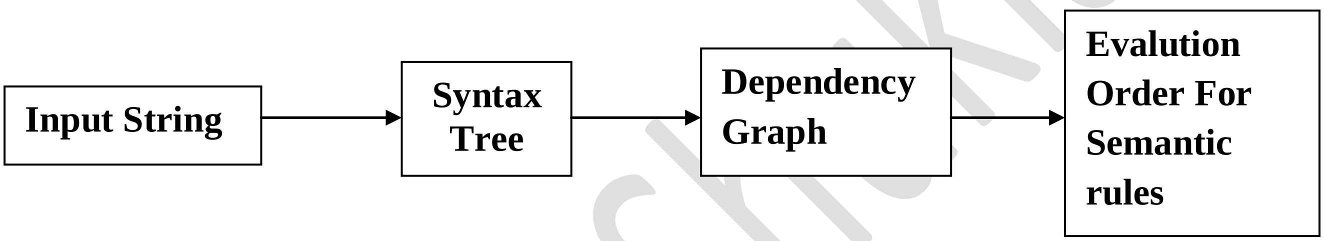
Syntax Directed Translation

Syntax Directed Definition:-

It is the generalization of a context free grammar in which each grammar symbol has an associated set of attributes and rules are associated with productions. If X is a symbol and a is one of its attributes, then X **.** a denotes the value of a at a particular parse-tree node labeled X. An attributes can represent a string , a number, a type, memory location etc. The value of an attribute at a parse tree node is defined by the semantic rules associated with the production used by that node.



SDT=Grammar + Semantic Rule

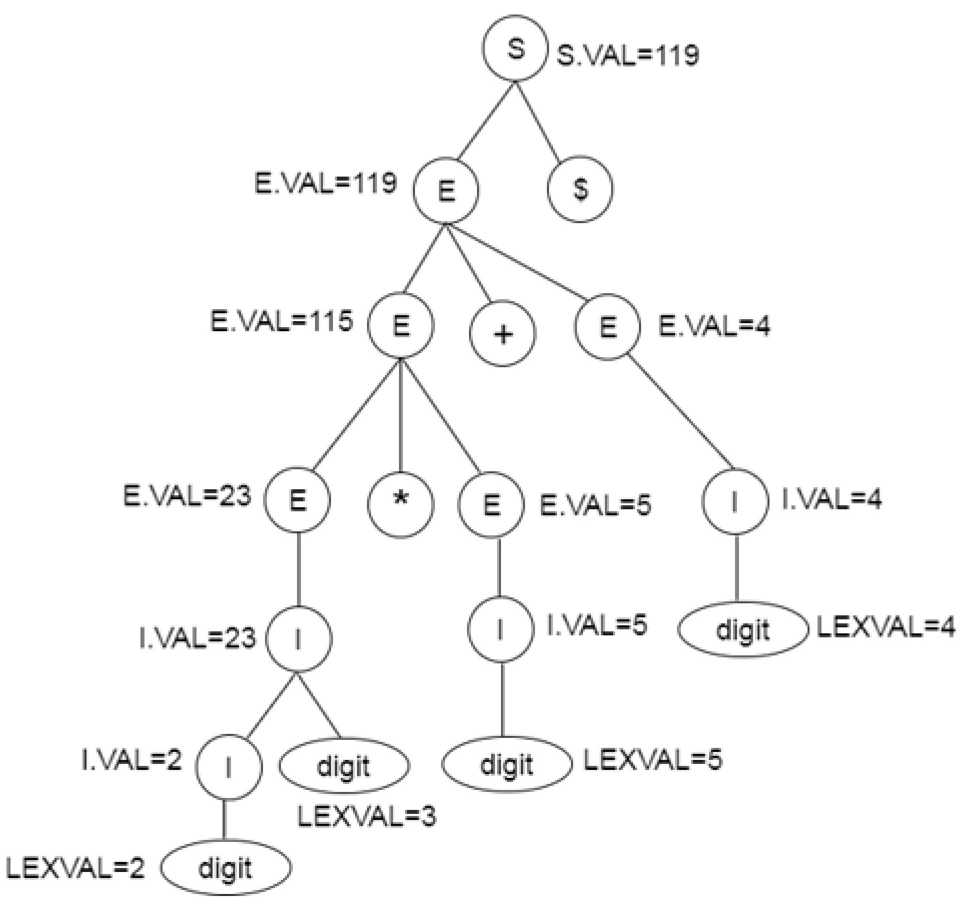
SDT= Grammar + Semantic Rule

Application:-

- Generates intermediate code
- Put information into the symbol
- Perform type checking
- Issue error messages
- Perform almost any activity

Ex.

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}



Definition:- In a SDD, each grammar production X→ α is associated with it a set of semantic rules of the form a:f(b_1,b_2,b_3……..b_k) , where a is an attribute obtained from the function f.

Attribute is of two types:

I. Synthesized Attribute
II. Inherited Attribute

Synthesized Attribute:

The attribute 'a' is called synthesized attribute of X and b_1,b_2,b_3……..b_k are attributes belonging to the production symbols. The value of synthesized attribute at node is computed from the value of attributes at children of that node in the parse tree.

Inherited Attribute:

The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production(i.e. α) and b_1,b_2,b_3……..b_k are belonging to either X or α . The inherited attributes can be computed from the values of the attributes at the siblings and parent of that node.

1. Synthesized Attribute

How to compute Synthesized Attribute?

Consider the CFG as

$S \rightarrow EN$
$E \rightarrow E+T$
$E \rightarrow E-T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow T/F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow digit$
$N \rightarrow ;$

The syntax-directed definition can be written for the above grammar by using semantic actions for each production.

Production Rules	Semantic Actions
$S \rightarrow EN$	Print (E.val)
$E \rightarrow E_1+T$	$E.val := E_1.val + T.val$
$E \rightarrow E_1-T$	$E.val := E_1.val - T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1*F$	$T.val := T_1.val \times F.val$
$T \rightarrow T_1/F$	$T.val := T_1.val / F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$
$N \rightarrow ;$	Can be ignored by Lexical Analyzer as ; is terminating symbol

For the non-terminals E, T and F the values can be obtained using the attribute "val". Here "val" is a attribute and semantic rule is computing the value of val. In the rule $S \rightarrow EN$, symbol S is the start symbol. This rule is to print the final output of the statement.

- In SDD, terminals have synthesized attributes only.
- The SDD that uses only synthesized attributes is called S-attributed definition.

In a parse tree, at each node the semantic rule is evaluated for annotating (computing) the S-attributed definition. This processing is in bottom-up fashion i.e. from leaves to root.

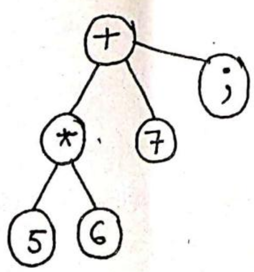
Steps to compute S-attributed definition

(i) Write the SDD using the appropriate semantic actions for corresponding production rule of the given grammar.

(ii) The annotated parse tree is generated and attribute values are computed. The computation is done in bottom-up manner.

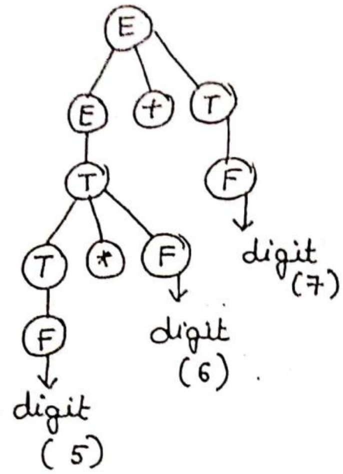
(iii) The value obtained at the root node is supposed to be the final output.

Example: Construct parse tree, syntax tree and annotated parse tree for the input string is 5 * 6 + 7;

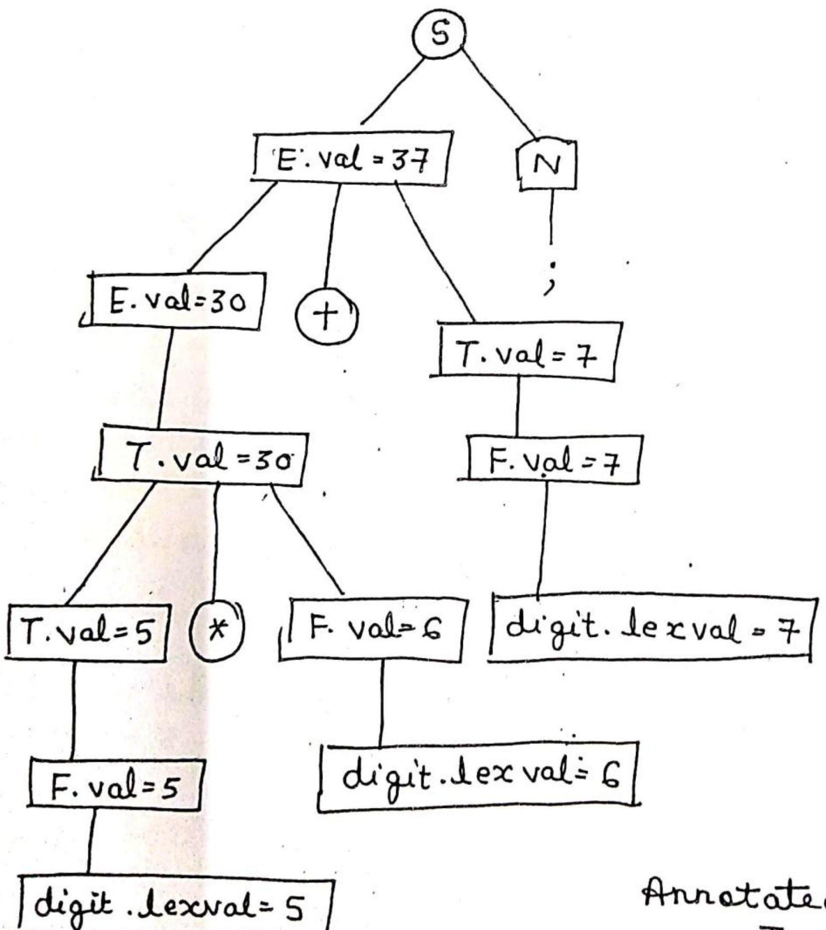


syntax tree

$E \rightarrow E + T$
$E \rightarrow T + T$
$E \rightarrow T * F + T$
$E \rightarrow F * F + T$
$E \rightarrow F * F + F$
$E \rightarrow digit * digit + digit$



Parse Tree



Annotated Parse Tree

Thus, S-attributed definition can be computed by a bottom-up fashion or using postorder traversal.

Inherited Attribute

The value of inherited attribute at a node in a parse tree is defined using the attribute values at the parent or siblings.

Example - Annotate the parse tree for the computation of inherited attributes for the given string:

int a, b, c ;

The grammar is given as -

S → TL
T → int
T → float
T → char
T → double
L → L_1, id
L → id

For the string int a, b, c we have to distribute the data type int to all the identifiers a, b and c; such that a becomes integer, b becomes integer and c becomes integer.

Steps :

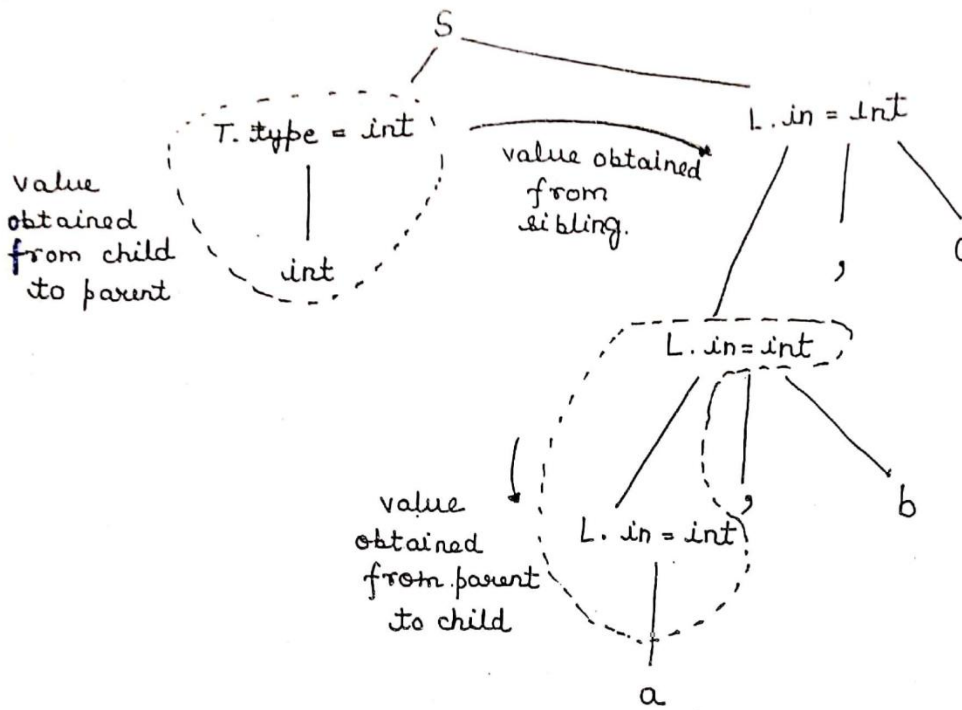
(i) Construct the syntax-directed definition (SDD) using semantic action.

(ii) Annotate the parse tree with inherited attributes by processing in top-down fashion.

The syntax-directed definition for the above grammar is -

Production Rule	Semantic Actions
S → TL	L.in := T.type
T → int	T.type := integer
T → float	T.type := float
T → char	T.type := char
T → double	T.type := double
L → L_1, id	L_1.in := L.in Enter_type (id.entry, L.in)
L → id	Enter_type (id.entry, L.in)

Annotated Parse Tree



The computation of type is done in top-down manner or in preorder traversal. Using function Enter_type the type of identifiers a, b and c is inserted in the symbol table at corresponding id. entry (The id. entry is the address of corresponding identifier in the symbol table).

Dependency Graph : The directed graph that represents the interdependencies between synthesized and inherited attributes at the nodes in the parse tree is known as dependency graph.

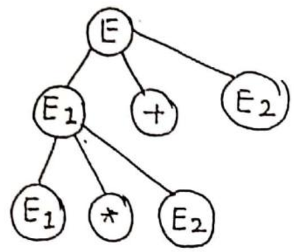
For the rule, $X \rightarrow YZ$, the semantic action is given by $X.x := f(Y.y, Z.z)$ then synthesized attribute is $X.x$ and $X.x$ depends upon attributes $Y.y$ and $Z.z$.

Eg – Design the dependency graph for the following grammar.

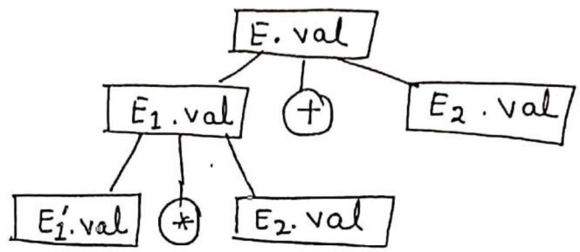
$$E \rightarrow E_1 + E_2$$

$$E \rightarrow E_1 * E_2$$

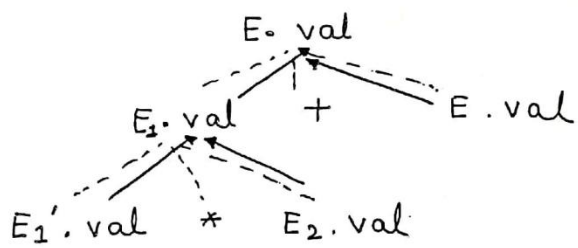
Production Rule	Semantic Rule
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E_1 \rightarrow E_1 * E_2$	$E_1.val := E_1'.val * E_2.val$



Parse tree



Annotated Parse tree



Dependency graph

The synthesized attributes can be represented by .val. Hence the synthesized attributes are given by $E.val$, $E_1.val$ and $E_2.val$. The dependencies among the nodes is given by solid arrows. The arrows from E_1 and E_2 show that value of E depends upon E_1 and E_2.

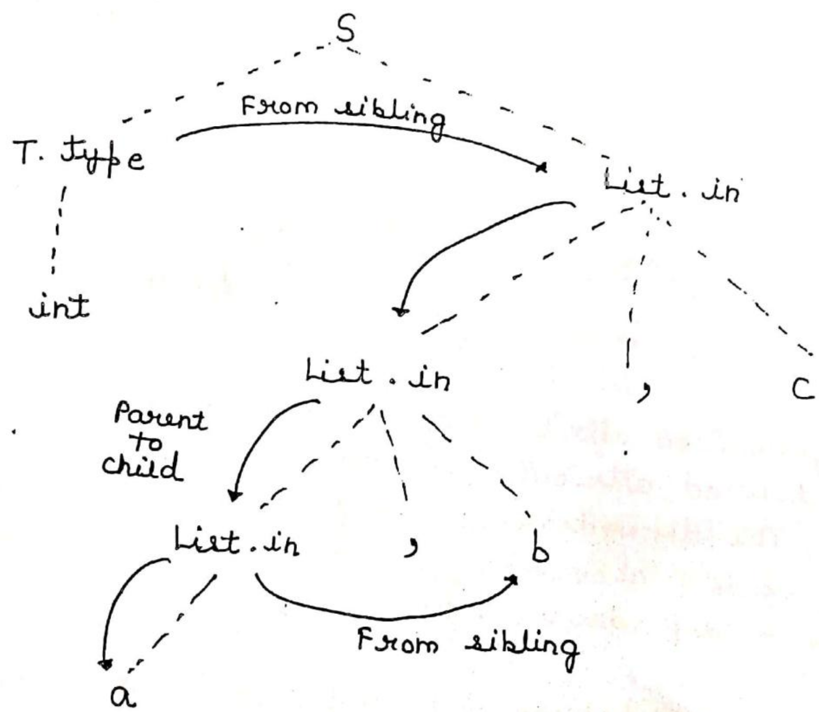
Ques) Design the dependency graph for the following grammar.

$S \rightarrow T\ List$
$T \rightarrow int$
$T \rightarrow float$
$T \rightarrow char$
$T \rightarrow double$
$List \rightarrow List_1, id$
$List \rightarrow id$

The dotted line is for representing the parse tree.

int a, b, c

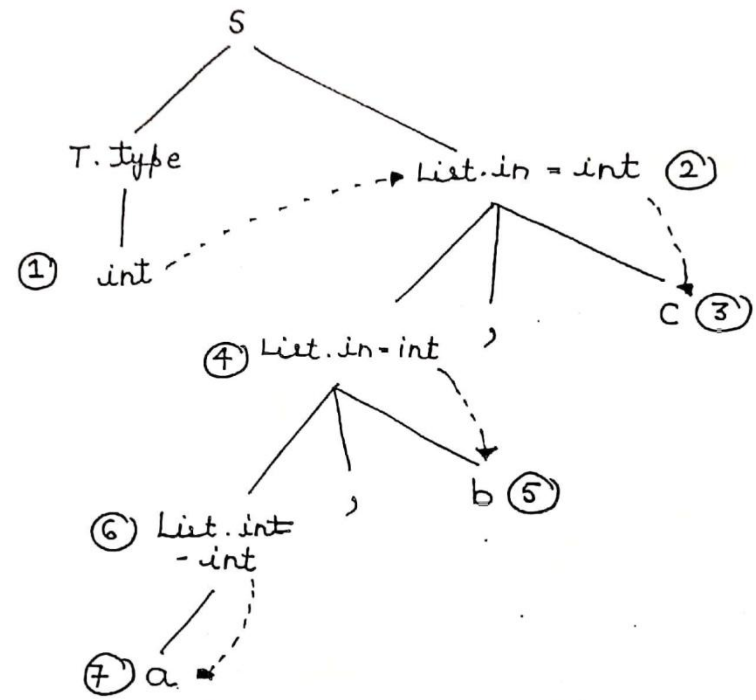
Production Rule	Semantic Actions
$S \rightarrow T\ List$	List.in := T.type
$T \rightarrow int$	T.type := integer
$T \rightarrow float$	T.type := float
$T \rightarrow char$	T.type := char
$T \rightarrow double$	T.type := double
$List \rightarrow List_1 , id$	$List_1$.in = List.in Enter_type (id.entry, List.in)
$List \rightarrow id$	Enter_type (id.entry, List.in)



Dependency Graph

NOTE: Syntax directed translation scheme embeds program fragments called semantic actions within production body.

Evaluation Order : The topological sort of the dependency graph decides the evaluation order in a parse tree. In deciding evaluation order the semantic rules in the SDD are used. Thus the translation is specified by SDD.



The evaluation order can be decided as follows:

1. The type int is obtained from lexical analyzer by analyzing the input token.
2. The List.in is assigned the type int from the sibling T.type.
3. The entry in the symbol table for identifier c gets associa- -ted with the type int. Hence variable c becomes of integer type
4. The List.in is assigned the type int from the parent List.in.
5. The entry in the symbol table for identifier b get associated with the type int. Hence variable b becomes of integer type.
6. The List.in is assigned the type int from the parent List.in.
7. The entry in the symbol table for identifier a gets associated with the type int. Hence variable a becomes of integer type.

Thus, by evaluation the semantic rules in this order stores the type int in the symbol table entry for each identifier a, b and c.

Implementation of SDT:-

To implementation SDT we have to use a stack which consist of pair of arrays STATE & VALUE. Each state entry show a pointer to the parsing table and each value represent the value associated with corresponding state symbol.

State array consist of all the variables or identifiers appeared in CFG and value array consist of the values of each variable.

Ex.

A→BC

STATE	VALUE
C	C.VALUE
B	B.VALUE

Stack before Reduction

STATE	VALUE
A	A.VALUE

Stack after reduction

Ex
Consider the desk calculator grammar

S→E$
E→E+E
E→E*E
E→ (E)
E→I
I→I digit
I→digit where digit=0,1,2,3,……9

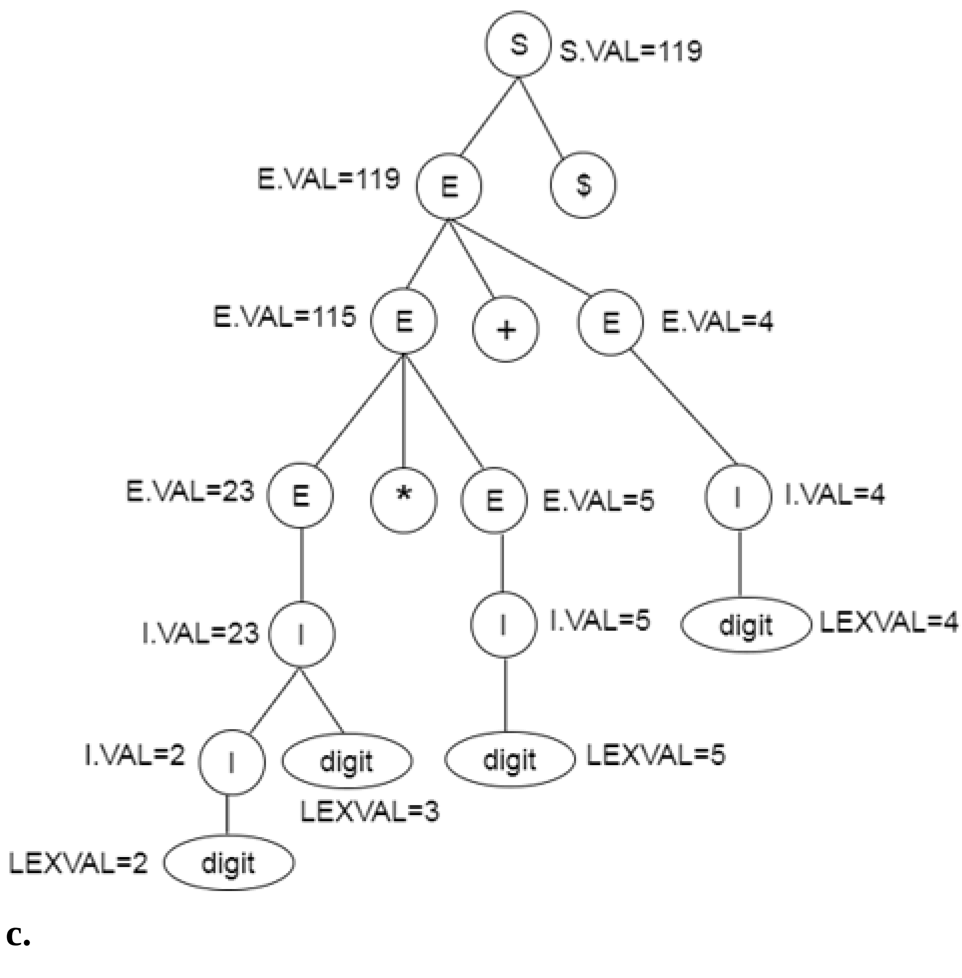
- **a.** Write down the SDT scheme for the above desk calculator grammar
- **b.** Construct parse tree with translation for string **23*5+4$**
- **c.** Find the sequence of moves made by parser for string **23*5+4$**

Solution:-

a.

S→E$	**{S.value=E.value}**
E→E+E	**{E.value=E.value+E.value}**
E→E*E	{ **E.value=E.value*E.value** }
E→ (E)	{ **E.value=E.value**}
E→I	{ **E.value=I.value**}
I→I digit	{ **I.value=I.value*10+digit.lvalue** }
I→digit	{ **I.value=digit.lvalue** }

b.



c.

Input String	STATE	VALUE
23*5+4$	-	-
3*5+4$	2	-
3*5+4$	I	2
*5+4$	I3	2
*5+4$	I	23
*5+4$	E	23
5+4$	E*	23
+4$	E*5	23
+4$	E*I	23-5
+4$	E*E	23-5
+4$	E	115
4$	E+	115-
$	E+4	115-
$	E+I	115-4
$	E+E	115-4
$	E	119
	E$	119
	S	119

Intermediate Code:- After performing semantic analysis on the parse tree the compiler converts parse tree into the intermediate code.

Parse tree with translation → **Intermediate Code Generation** → intermediate code

Advantages of Intermediate Code:-

1. It is machine independent
2. It makes the task of code optimization easy
3. Perform efficient code generation

Types of Intermediate Code:- There are mainly 3 types of intermediate code representation.

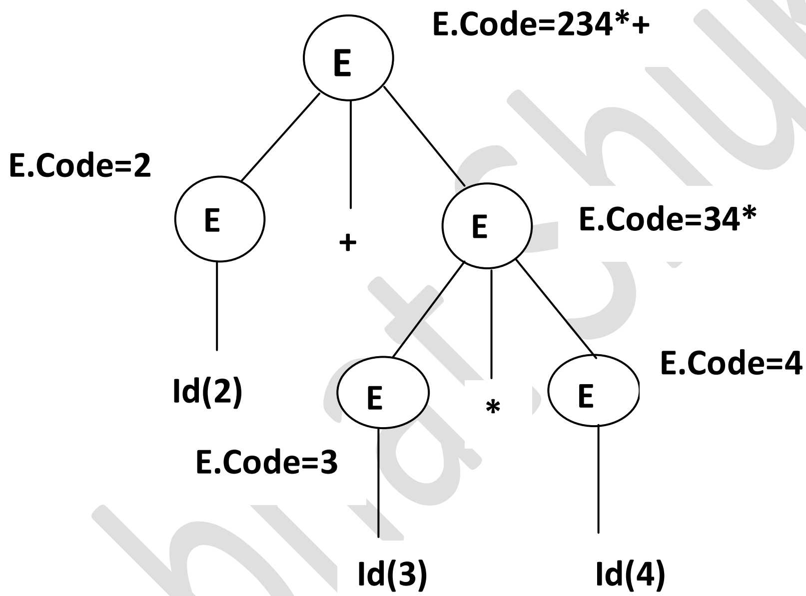
1. Syntax Tree
2. Postfix Notation
3. Three address code(3AC)

Postfix Notation:-

Question:-Write the SDT rule to generate intermediate code (Postfix form)

Solution: - Let us assume the grammar

E→E+E { E . code =E . code || E . code || '+' }
E→E*E { E . code =E . code || E . code || '*' }
E→E+E { E . code =id .value }



Syntax Tree:-

Consider the following grammar

E→E+T

E→E-T

E→E*T

E→T

T→id

T→num

mknode(op , left , right):-_This function creates a node with the field operator having operator as a label, and the two pointers(left and right)

op	Left	Right

mkleaf(id, entry):- This function creates an identifier node with label id and a pointer to symbol table is given by entry

id	Entry

mkleaf(num, val):- This function creates node for number with label num and val is for value of that number.

num	Val

Question : - Construct the syntax tree for the expression

x * y - 5+z

Solution:-

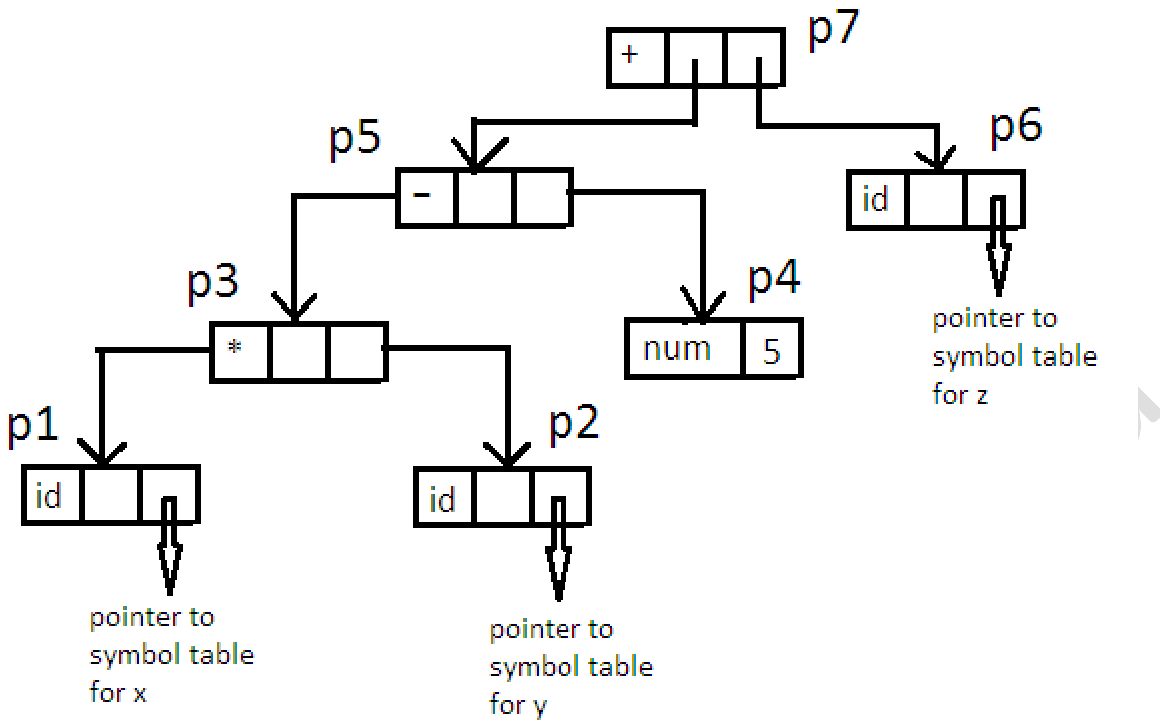
Step 1:- Convert the expression from infix to postfix

xy*5-z+

Step 2:- Make use of the function mknode(), mkleaf()

Step 3:- The sequence of function call is given

Symbol	Operation
X	**p1=mkleaf(id , ptr to entry x)**
Y	**p2=mkleaf(id , ptr to entry y)**
*	**p3=mknode(* , p1 , p2)**
5	**p4= mkleaf(num,5)**
-	**p5= mknode(- , p3 , p4)**
Z	**p6=mkleaf(id, ptr to entry z)**
+	**p7=mknode(+ , p5 , p6)**



The SDT for the above grammar is:-

E→E+T **{E.nptr=mknode('+', E . nptr , T.nptr)}**

E→E-T **{E.nptr=mknode('-', E . nptr , T.nptr)}**

E→E*T **{E.nptr=mknode('*', E . nptr , T.nptr)}**

E→T **{E.nptr=T.nptr}**

T→id **{E.nptr=mkleaf(id, id.ptr_entry)}**

T→num **{T.nptr=mkleaf(num, num.val)}**

Three Address Code:-

Three-address code is a sequence of statements of the general form

x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like x+ y*z might be translated into a sequence

t1: = y * z
t2: = x + t1

where t1 and t2 are compiler-generated temporary names.

Types of Three-Address Statements:-

The common three-address statements are:

1. Assignment statements of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation *relop to y*. If not, the three-address statement following if *x relop y* goto L is executed next as in the usual sequence.
6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,
   param x1
   param x2
   . . .
   param xn
   call p,n
   generated as part of a call of the procedure p(x1, x2, …. ,xn).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.
8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Implementation of three address code:-

Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. There are 3 representations used for 3-address code such as quadruples, triples and indirect triples.

Quadruple Representation:-

The quadruple is a structure with at the most four fields such as op, arg1, arg2, result. The op field is used to represent the operator, the arg1, arg2 represent the two operands used and result field is used to store the result of an expression.

Ex.

x= - a * b + -a * b

The three address code for the above expression is:

t1= - a
t2=t1*b
t3= - a
t4=t3*b
t5=t2+t4
x=t5

Quadruple Representation for the above expression is

	op	Arg1	Arg2	Result
(0)	-	a		t1
(1)	*	t1	b	t2
(2)	-	a		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		x

Triples:-

In the triple representation the use of temporary variables is avoided by referring the pointers in the symbol table

Ex.

x= - a * b + -a * b

The three address code for the above expression is:

t1= - a
t2=t1*b
t3= - a
t4=t3*b
t5=t2+t4
x=t5

Triples Representation for the above expression is

	op	Arg1	Arg2
(0)	-	a	
(1)	*	(0)	b
(2)	-	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=		(4)

Indirect Triples:-

In the indirect triple representation the listing of triples is been done and listing pointers are used instead of using statements.

	op	Arg1	Arg2
(0)	-	a	
(1)	*	(11)	b
(2)	-	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	=		(15)

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

Question: - Write the quadruple , triples, and indirect triples for the following expression-

(x + y) * (y + z) + (x + y + z)

Solution:-

Three –Address code

t1=x + y
t2=y + z
t3=t1*t2
t4=t1+z
t5=t3+t4

Quadruple

Location	Operator	Operand1	Operand2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t2	z	t4
(5)	+	t3	t4	t5

Triple

Location	Operator	Operand1	Operand2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Indirect Triple

Location	Operator	Operand1	Operand2
(1)	+	x	y
(2)	+	y	z
(3)	*	(11)	(12)
(4)	+	(11)	Z
(5)	+	(13)	(14)

Location	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Assignment Statements:-

The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record.

Example – Obtain the translation scheme for obtaining the three-address code for the grammar –

S → id : = E
E → E₁ + E₂
E → E₁ * E₂
E → – E₁
E → (E₁)
E → id

Here, we will use the translation scheme which in turn will generate the three-address code.

Production Rule	Semantic Actions
S → id := E	{ id_entry := look_up(id.name); if id_entry ≠ nil then append (id-entry ':=' E.place) else error; /* id not declared */ }
E → E₁ + E₂	{ E.place := newtemp(); append (E.place ':=' E₁.place '+' E₂.place) }
E → E₁ * E₂	{ E.place := newtemp(); append(E.place ':=' 'E₁.place '*' E₂.place)
E → – E₁	{ E.place := newtemp(); append (E.place ':=' 'uminus' E₁.place) }
E → (E₁)	{ E.place := E₁.place }
E → id	{ id_entry := look_up (id.name); if id_entry ≠ nil then append (id_entry ':=' E.place) else error; /* id not declared */ }

The look-up returns the entry for id.name in the symbol table if it exists there.

The function append is for appending the 3-address code to the output file. Otherwise an error will be reported.

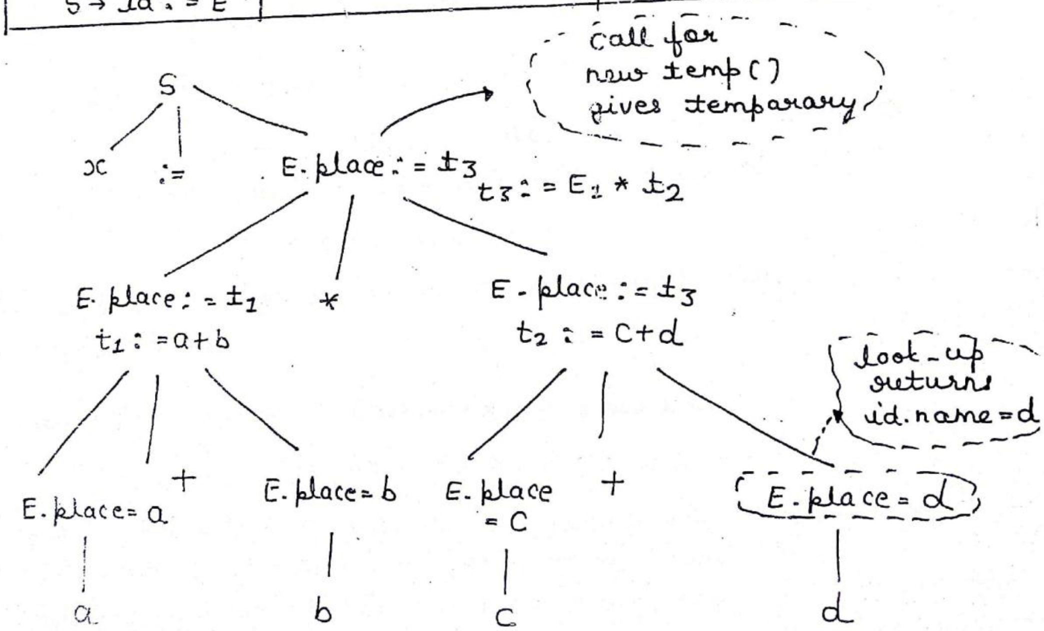
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E.

Sem

Consider the assignment statement $x := (a+b) * (c+d)$

We will assume all these identifiers are of the same type.

Production Rule	Semantic action for attribute evaluation	Output
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a+b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	$t_2 := c+d$
$E \rightarrow E_1 * E_2$	E.place := t_3	$t_3 := (a+b)*(c+d)$
$S \rightarrow id := E$		$x := t_3$



Annotated Parse Tree

Boolean Expression:-

There are two types of Boolean expressions used-

1. For computing the logical values
2. In conditional expressions using if-then-else or do-while

Ex. Consider the Boolean expression generated by following grammar

E→E or E
E→E and E
E→ NOT E
E→ (E)
E→ id relop id
E→TRUE
E→FALSE

Here the relop is denoted by ≤ , ≥ , ≠ , < , >. The **OR** and **AND** are left associative. The highest precedence is to NOT then AND and then OR.

Numerical Representation- The translation scheme for Boolean expression having numerical representation is as given below.

Production Rule	Semantic Rule

E→E or E	{
	E .place=newtemp()
	append(E .place=E .place or E .place)
	}

E→E and E	{
	E .place=newtemp()
	append(E .place=E .place and E .place)
	}

E→ NOT E	{
	E .place=newtemp()
	append(E .place=NOT E .place)
	}

E→ (E)	{
	E.place=E.place
	}

E→ id relop id	{ E.place=newtemp()

	append('if' id1.place relop id2.place 'goto' next_state+3) append(E.place='0') append('goto' next_state+2) append(E.place='1') }

E→TRUE	{
	E.place=newtemp();
	append(E.place='1')
	}

E→FALSE	{
	E.place=newtemp();
	append(E.place='0')
	}

The function append generates the three address code and newtemp() is for generation of temporary variables. For the semantic action for the rule **E→id1 relop id2** contains next_state which gives the index of next three address statements in the output sequence.

Ex. **generates the three-address code using above translation scheme**

p>q AND r<s OR u>v

Solution:-

100: if p>q goto 103
101: t1=0
102: goto 104
103: t1=1
104: if r>s goto 107
105: t2=0
106: goto 108
107: t2=1
108: if u>v goto 111
109: t3=0
110: goto 112
111: t3=1
112: t4=t1 AND t2
113: t5=t4 OR t3

In this three address code , goto is used to jump on some specific statement. This method of evaluation is called **“short-circuit”.** The AND has higher precedence over OR hence at location 112 the AND is performed and then in the immediate next statement OR is performed.

Ex. Generate three address code for **A OR B AND NOT C**

Solution:-

t1=NOT C
t2=B AND t2
t3=A OR t2

Ex. Generate three address code for

```
if(a<b)
        then 1
else
        then 0
```

Solution:-

```
100:   if a<b goto 103
101:   t=0
102:   goto 104
103:   t=1
104:   stop
```

Flow-of-Control Statements:-

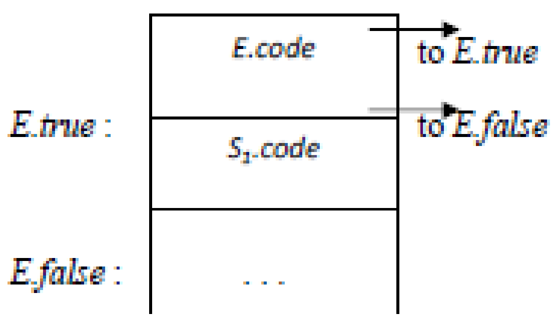
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

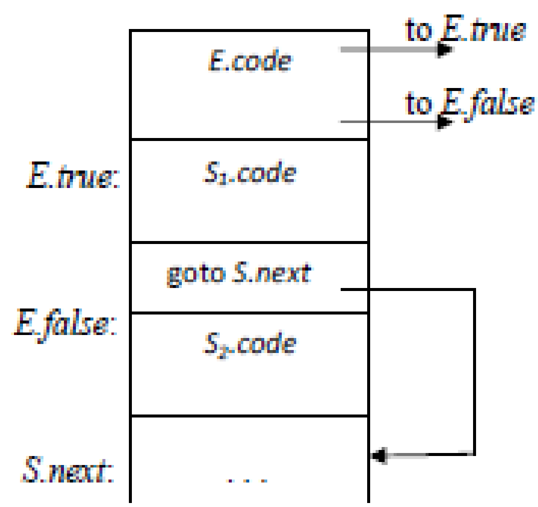
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

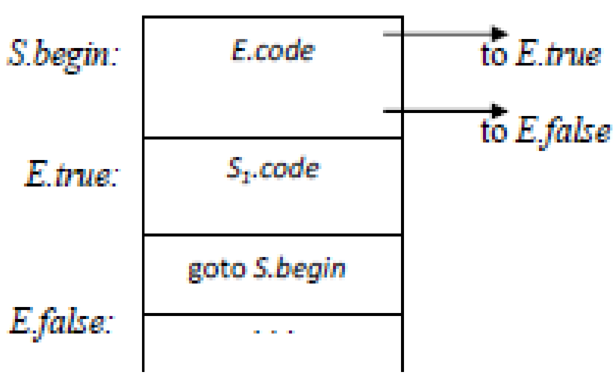
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code \parallel$ $gen(\text{'}\mathbf{goto}\text{'}\ S.next) \parallel$ $gen(E.false\ ':') \parallel S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \parallel E.code \parallel$ $gen(E.true\ ':') \parallel S_1.code \parallel$ $gen(\text{'}\mathbf{goto}\text{'}\ S.begin)$

Ex. Generate three address code for

```
while(i<10)
{
    x=0;
    i=i+1;
}
```

Solution:-

```
100:  L1: if i<10 goto L2
101:  goto Lnext
102:  L2: x=0
103:  i=i+1
104:  goto L1
105:  Lnext
```

Case Statements:-

Consider the following switch statement:
switch *E*
begin
case V_1 : S_1
case V_2 : S_2
. . .
case V_{n-1} : S_{n-1}
default : S_n
end

The translation scheme for case statement:

```
        code to evaluate E into t
        goto test
L1 :    code for S1
        goto next
L2 :    code for S2
        goto next
        . . .
Ln-1 :  code for Sn-1
        goto next
Ln :    code for Sn
        goto next
test :  if t = V1 goto L1
        if t = V2 goto L2
        . . .
        if t = Vn-1 goto Ln-1
        goto Ln
next :
```

To translate into above form:

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression *E* is parsed, the code to evaluate *E* into **t** is generated. After processing *E* , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label L*i* is created and entered into the symbol table.
- A pointer to this symbol-table entry and the value *Vi* of case constant are placed on a stack (used only to store cases).

Procedure Calls:-

Procedure or function is an important programming construct which is used to obtain the modularity in the user program.

Let us consider a grammar for a simple procedure call statement

S →**call id** (L)

L→ L, *E*

L→ *E*

Here the non-terminal S denotes the statement and non terminal. L denotes the list of parameters and E denotes the expression it could be id as well.

The translation scheme is as follows-

Production Rule	Semantic Action
S→call id (L)	{ for each item p in queue do append('param ' p); append('call' id.place) }
L→L,E	{ insert E. place in the queue }
L→E	{ initialize queue and insert E.place in the queue }

BACKPATCHING:-

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for Boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels ***backpatching***.

To manipulate lists of labels, we use three functions :

1. ***makelist*(*i*)** creates a new list containing only *i*, an index into the array of quadruples; *makelist* returns a pointer to the list it has made.

2. ***merge*(*p1*,*p2*)** concatenates the lists pointed to by *p1* and *p2,* and returns a pointer to the concatenated list.

3. ***backpatch(p,i)*** inserts i as the target label for each of the statements on the list pointed to by *p.*

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for Boolean expressions during bottom-up parsing. The grammar we use is the following:

E →*E1* **or** *M E2*
| *E1* **and** *M E2*
| **not** *E1*
| (*E1*)
| **id1 relop id2**
| **true**
| **false**
M → ε

Synthesized attributes *truelist* and *falselist* of nonterminal *E* are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by *E.truelist* and *E.falselist*

Consider production *E*→ *E1* **and** *M E2.* If *E1* is false, then *E* is also false, so the statements on *E1.falselist* become part of *E.falselist*. If *E1* is true, then we must next test *E2*, so the target for the statements *E1.truelist* must be the beginning of the code generated for *E2*. This target is obtained using marker nonterminal *M.*

Attribute *M.quad* records the number of the first statement of *E2.code*. With the production M → ε we associate the semantic action
{ *M.quad : = nextquad* }

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the *E1.truelist* when we have seen the remainder of the production $E \rightarrow E1$ **and** $M\ E2$. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\ E_2$	{ *backpatch* (E_1*.falselist*, *M.quad*); *E.truelist* : = *merge*(E_1*.truelist*, E_2*.truelist*); *E.falselist* : = E_2*.falselist* }
(2) $E \rightarrow E_1$ **and** $M\ E_2$	{ *backpatch* (E_1*.truelist*, *M.quad*); *E.truelist* : = E_2*.truelist*; *E.falselist* : = *merge*(E_1*.falselist*, E_2*.falselist*) }
(3) $E \rightarrow$ **not** E_1	{ *E.truelist* : = E_1*.falselist*; *E.falselist* : = E_1*.truelist*; }
(4) $E \rightarrow (E_1)$	{ *E.truelist* : = E_1*.truelist*; *E.falselist* : = E_1*.falselist*; }
(5) $E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.truelist* : = *makelist* (*nextquad*); *E.falselist* : = *makelist*(*nextquad* + *1*); *emit*('if' **id**$_1$*.place* **relop**.*op* **id**$_2$.place '**goto**_') *emit*('**goto**_') }
(6) $E \rightarrow$ **true**	{ *E.truelist* : = *makelist*(*nextquad*); *emit*('**goto**_') }
(7) $E \rightarrow$ **false**	{ *E.falselist* : = *makelist*(*nextquad*); *emit*('**goto**_') }
(8) $M \rightarrow \varepsilon$	{ *M.quad* : = *nextquad* }

Flow-of-Control Statements:-

A translation scheme is developed for statements generated by the following grammar:

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L\ ;\ S$
(7) | S

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes *E.truelist* and *E.falselist*. *L* and *S* also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L..nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

The semantic rules for the revised grammar are as follows:

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ *backpatch* (*E.truelist*, M_1*.quad*);
backpatch (*E.falselist*, M_2*.quad*);
S.nextlist : = *merge* (S_1*.nextlist*, *merge* (*N.nextlist*, S_2*.nextlist*)) }

We backpatch the jumps when E is true to the quadruple $M_1.quad$, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S_2. The list *S.nextlist* **includes all jumps out of** S_1 and S_2, as well as the jump generated by N.

(2)	$N \rightarrow \varepsilon$	{ *N.nextlist* : = *makelist*(*nextquad*); *emit*('**goto** _') }
(3)	$M \rightarrow \varepsilon$	{ *M.quad* : = *nextquad* }
(4)	$S \rightarrow$ **if** E **then** $M\ S_1$	{ *backpatch*(*E.truelist*, *M.quad*); *S.nextlist* : = *merge*(*E.falselist*, S_1*.nextlist*) }
(5)	$S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$	{ *backpatch*(S_1*.nextlist*, M_1*.quad*); *backpatch*(*E.truelist*, M_2*.quad*); *S.nextlist* : = *E.falselist* *emit*('**goto**' M_1*.quad*) }
(6)	$S \rightarrow$ **begin** L **end**	{ *S.nextlist* : = *L.nextlist* }
(7)	$S \rightarrow A$	{ *S.nextlist* : = **nil** }

The assignment *S.nextlist* : = **nil** initializes *S.nextlist* to an empty list.

(8)	$L \rightarrow L1\ ;\ M\ S$	{ *backpatch*(L_1*.nextlist*, *M.quad*); *L.nextlist* : = *S.nextlist* }

The statement following L_1 in order of execution is the beginning of S. Thus the *L1.nextlist* list is backpatched to the beginning of the code for S, which is given by *M.quad*.

(9)	$L \rightarrow S$	{ *L.nextlist* : = *S.nextlist* }

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the ith element of array A begins in location

$$base + (i - low) \times w$$

where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of A*[low]*.

The expression can be partially evaluated at compile time if it is rewritten as

$$i \text{ x } w + (\ base - low \text{ x } w)$$

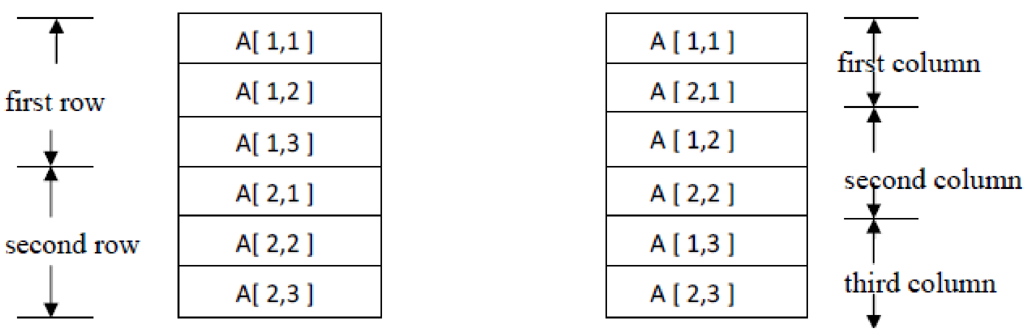
The subexpression $c = base - low \text{ x } w$ can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.

Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR **(b) COLUMN-MAJOR**

In the case of row-major form, the relative address of A[i_1 , i_2] can be calculated by the formula

$$base + ((i_1 - low_1) \text{ x } n_2 + i_2 - low_2) \text{ x } w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2, then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((\ i_1 \text{ x } n_2\ ) + i_2\ ) \text{ x } w + (\ base - ((\ low_1 \text{ x } n_2\ ) + low_2\ ) \text{ x } w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of A[$i_1,i_2,...,i_k$]

$((\ldots ((i_1n_2 + i_2)\ n_3 + i_3) \ldots)\ n_k + i_k) \text{ x } w + base - ((\ldots ((low_1n_2 + low_2)n_3 + low_3) \ldots)$
$n_k + low_k) \text{ x } w$

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist\ ]$
(6) $L \rightarrow$ **id**
(7) $Elist \rightarrow Elist , E$
(8) $Elist \rightarrow$ **id** $[E$

We generate a normal assignment if *L* is a simple name, and an indexed assignment into the location denoted by *L* otherwise :

(1) $S \rightarrow L := E$ { **if** *L.offset* = **null then** /* *L is a simple* **id** */
emit (*L.place* ': =' *E.place*) ;
else
emit (*L.place* '[' *L.offset* ']' ': =' *E.place*) }

(2) $E \rightarrow E_1 + E_2$ { *E.place* : = *newtemp*;
emit (*E.place* ': =' E_1*.place* '+' E_2*.place*) }

(3) $E \rightarrow (E_1)$ { *E.place* : = E_1*.place* }

When an array reference L is reduced to E , we want the r-value of L. Therefore we use indexing to obtain the contents of the location $L.place$ [$L.offset$] :

(4) $E \rightarrow L$ { **if** $L.offset$ = **null then** /* L *is a simple* **id** */
$E.place$: = $L.place$
else begin
$E.place$: = *newtemp;*
emit ($E.place$ ': =' $L.place$ '[' $L.offset$ ']')
end }

(5) $L \rightarrow Elist$] { $L.place$: = *newtemp;*
$L.offset$: = *newtemp;*
emit ($L.place$ ': =' *c*($Elist.array$));
emit ($L.offset$ ': =' $Elist.place$ '*' *width* ($Elist.array$)) }

(6) $L \rightarrow$ **id** { $L.place$:= **id**.*place;*
$L.offset$:= **null** }

(7) $Elist \rightarrow Elist_1$, E { t := *newtemp;*
m : = $Elist_1.ndim$ + 1;
emit (t ': =' $Elist_1.place$ '*' *limit* ($Elist_1.array,m$));
emit (t ': =' t '+' $E.place$);
$Elist.array$: = $Elist_1.array$;

$Elist.place := t;$
$Elist.ndim := m$ }

(8) $Elist \rightarrow$ **id** [E { $Elist.array :=$ **id**.$place;$

$Elist.place := E.place;$
$Elist.ndim := 1$ }

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production $E \rightarrow E + E$ is :

$E \rightarrow E + E$ { $E.type :=$
if $E_1.type = integer$ **and**
$E_2.type = integer$ **then** *integer*
else *real* }

The entire semantic rule for $E \rightarrow E + E$ and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Semantic action for $E \rightarrow E_1 + E_2$

E.place := *newtemp;*
if E_1.*type* = *integer* **and** E_2.*type* = *integer* **then begin**
    emit(E.place ': =' E_1*.place* 'int +' E_2*.place);*
    E.type : = *integer*
end
else if E_1.*type* = *real* **and** E_2.*type* = *real* **then begin**
    emit(E.place ': =' E_1*.place* 'real +' E_2*.place);*
    E.type : = *real*
end
else if E_1.*type* = *integer* **and** E_2.*type* = *real* **then begin**
    u : = *newtemp;*
    emit(u ': =' 'inttoreal' E_1*.place);*
    emit(E.place ': =' *u* ' real +' E_2*.place);*
    E.type : = *real*
end
else if E_1.*type* = *real* **and** E_2.*type* =*integer* **then begin**
    u : = *newtemp;*
    emit(u ': =' 'inttoreal' E_2*.place*);
    emit(E.place ': =' E_1*.place* ' real +' *u);*
    E.type : = *real*
end
else
    E.type : = *type_error;*

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

t_1 : = i int* j
t_3 : = inttoreal t_1
t_2 : = y real+ t_3
x : = t_2