Comparison of different TAC implementations is given in table 6.4.

Table 6.4: Comparison of TAC

Parameter	Quadruples	Triples	Indirect triples
Indirection	No indirection is present. All variables (temporary as well as programmer defined) have immediate access through symbol table.	Indirection is present. However, since we allocate memory for every variable whether temporary or programmer defined, the indirection does not help much.	Indirection present.
Suitability to optimization	The quadruples lend well to optimization. When the statements need to be rearranged for optimization, we have to move the quads. For moving a quad, there is no extra dependence on other quads.	The triples are not suited to optimization. When the statements need to be rearranged for optimization, we have to move the triples. If we move a triple all the references (In the form of parenthesized numbers) in arg1 and arg2 arrays also have to updated accordingly. This is a time consuming operation making it less compile time efficient.	The indirect triple lends well to optimization. When statements need to be rearranged for optimization, we merely have to reorder the statement list. The references do not change.
Space	The space required for storage is not optimum because of the additional result field.	Requires less space than the quadruple	More space is needed when compared to triples. But, it can save some space when compared to quadruples because the statement list can point to the same triple for a temporary value, in case it is used more than once.

6.4 Translation into Intermediate Forms

Syntax Directed Translation (SDT) is the name given to the process of translating the input source to intermediate code based on the syntax of the language. This process is initialized during syntax analysis phase itself. For example consider the CFG for assignment statements given below. The parse tree generated for the input string "i=p+q+r;" is given in Fig 6.5.

SL → SL S
| S
S → id = E ;
E → E +T
| E-T
| T
T → T *F
| T/F
| F
F → (E)
| id

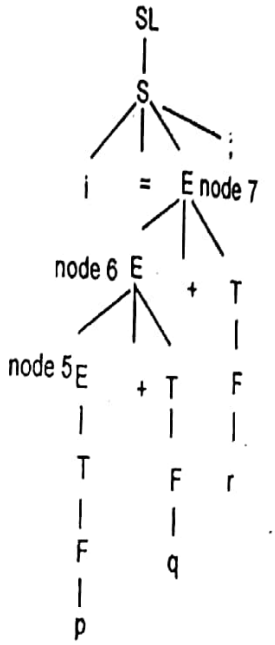


Fig 6.5 : PT for i = p+q+r;

The intermediate code cannot be generated directly from the Parse Tree (PT) shown in Fig. 6.5. The nodes have to be populated with more information. This process is called decorating the parse tree. Attribute grammars are a method to **decorate or annotate the parse tree**. For example, consider an identifier, the place and type of id, known as its **attributes**, can be added to the Parse Tree (PT). For example, for the identifier 'i' node 1, its place **value** is 'i'. For an 'E' node, the place value is the place (variable name) where the evaluation of the expression is placed. For example for 'E' at node '5' the value of expression evaluated is stored in the same node '5' (ie) 'p'. Another attribute is **code** generated. It is the intermediate code generated from the bottom till that point including that particular reduction. TAC is used for code generation. So for 'E' node 5 the reduction is E → T, which is a null code. Now consider 'E' node 6. Here a temporary variable 't0' is to be created to hold the value of expression 'p+q' . Hence its place is 't0'. The code attribute of node 6 (E+T) is "t0=p+q" due to reduction (E → E+T). Similarly the place value and code for node 7 (E+T) is 't1' and "t0=p+q; t1=t0+r;" Hence the decorated parse tree for input 'i=p+q+r' is given in fig 6.6.

In the reduction of T to E using E→T, there is no need to create a new temporary variable as place for 'E' node, as there is no additional computation required from 'T' node. The same place as 'T' node can store place for 'E' node .It is expressed as E.place = T.place . Consider the production E→ E1 +T | E1 - T, here a new temporary node has to be created to hold result value for E node. It is represented as E.place = NewTemp(). It returns the unused temporary variable like t0, t1, t2 etc. To distinguish between expression nodes on RHS & LHS, the expression node on RHS is represented as E1. The symbol '||' is used for concatenation. Rules are added to show the relationship between the LHS and RHS of the grammar rules. Such relationships are called the semantic rules. The CFG in which the productions are shown along with the semantic rules is called as the Syntax Directed Definition (SDD). The syntax directed translation for the assignment statement described previously is given below.

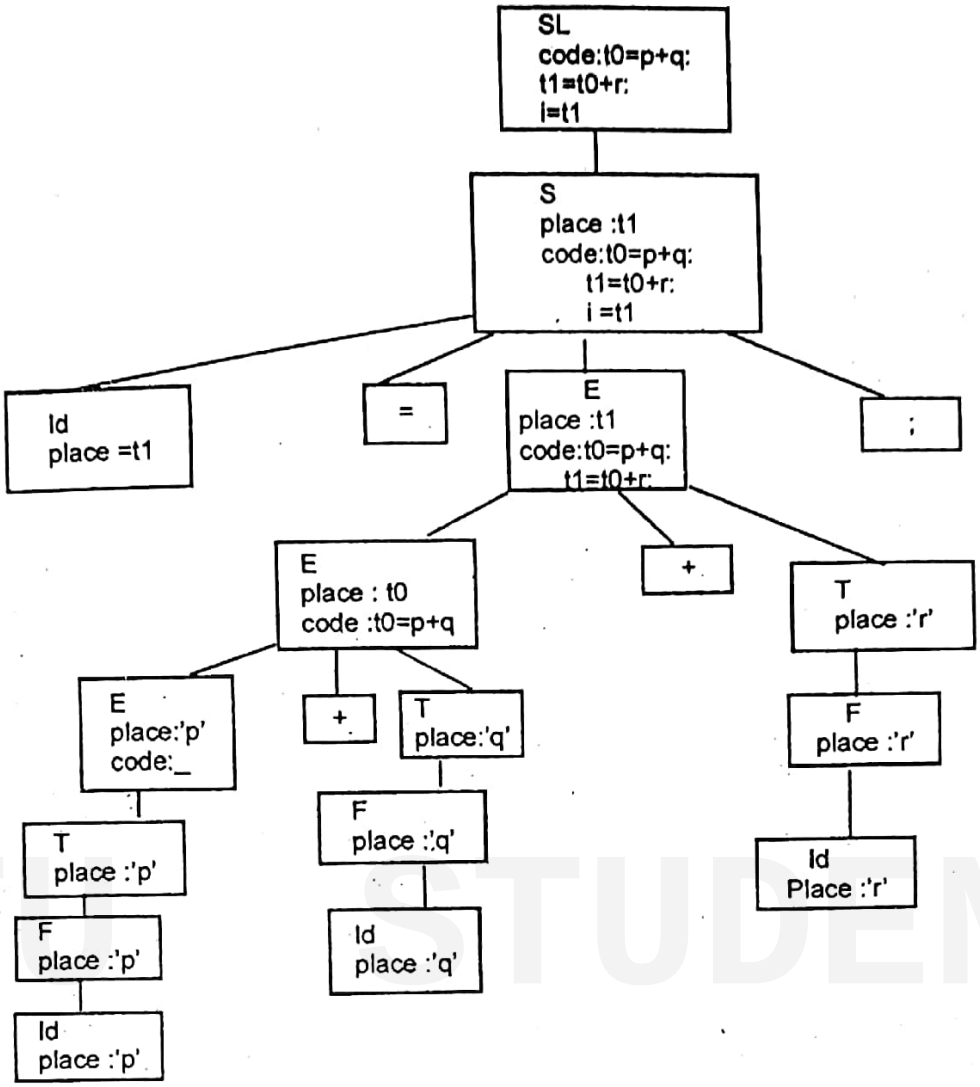


Fig 6.6 Decorated PT for i = p+q+r

SNO	Production	Semantic Rule
1.	SL → SL S | S	SL.code = SL1.code || S.code SL.code = S.code
2.	S → id := E;	S.place = id.place S.code = E.code || S.place := E.place
3.	E → E+T	E.place = NewTemp() E.code = E1.code || T.code || E.place = E1.place+T.place
	| E - T	E.place = NewTemp() E.code = E1.code || T.code || E.place = E1.place - T.place
	| T	E.place = T.place E.code = T.code

4.	T → T*F	T.place = NewTemp() T.code = T1.code || F.code || T.place = T1.place*F.place
	| T/F	T.place = NewTemp() T.code = T1.code || F.code || T.place = T1.place / F.place
	| F	T.place=F.place T.code=F.code
5.	F → (E)	F.place=E.place F.code=E.code
	| id	F.place=id.place F.code=""
	| constant	F.place = NewTemp() F.code = F.place= constant.value

If the addition of the parent node (LHS) depends on attributes of its children (RHS) such attributes are **synthesized attributes**. For example 'F' node having synthesized attribute as its value is synthesized from id's place node. Similarly as the code attribute of T is synthesized from code attributes of T and F (in production T → T*F | T/F) it is a synthesized attribute. Syntax directed translation that uses only synthesized attributes is called an **S-attributed definition**. The statement list CFG generates an S-attributed definition.

S-attributed definition: It is a syntax-directed definition that uses synthesized attributes only.

A parse tree can be represented using a directed graph. A post-order traversal of the parse tree can properly evaluate grammars with S-attributed definitions in a bottom up fashion.

If the attributes of the child depends on the attribute of parent node, then such an attribute is called **Inherited attribute**.

L-attributed definition is an attributed definition in which;

- Each attribute in each semantic rule for the production A → X1……Xn is either a synthesized attribute or an inherited attribute Xj which depends only on the inherited attribute of A and/or the attributes of X1; ……..;Xj-1.
- Independent of the evaluation order.
- Every S-attributed definition is an L-attributed definition.

For example, consider the syntax directed translation for variable declarations as given below:

SNO	Production	Semantic Rules
1.	decl_list → decl_list decl	
2.	| decl	
3.	decl → dtype id_list	id_list.type = dtype.category
4.	dtype → INT	dtype.category = INT
5.	| CHAR	dtype.category = CHAR
6.	| FLOAT	dtype.category = FLOAT
7.	id_list → id_list, ID	id_list1.type = id_list.type add_to_symboltable(ID.place,id_list.type)
8.	ID	add_to_symbol table(ID.place,id.list.type)

Consider the semantic rule for production 3, the attribute type for id_list depends on its child (dtype's) category. Similarly id_type of id_list1 in rule 7 is derived from its parent. Such attributes are called inherited attributes. Consider an input string 'int a, b;'. Its parse tree is shown in figure 6.7a The decorated parse tree for the same is shown in fig 6.7 b.The attributes are category for dtype node, type for id_list, and place for id

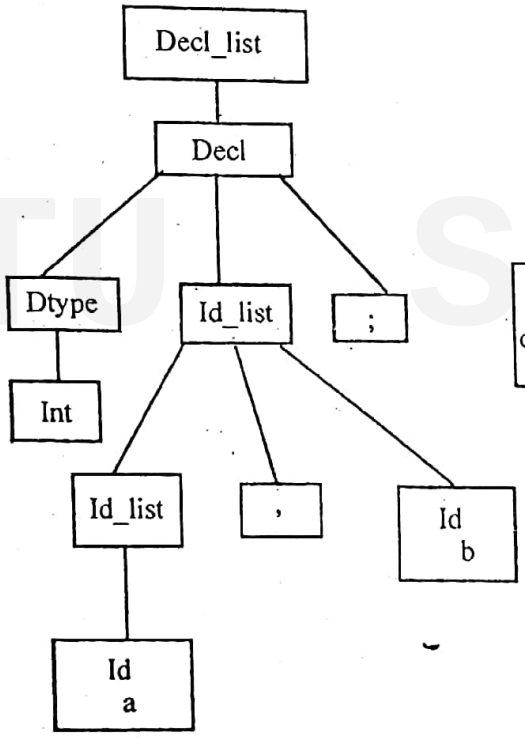


Fig 6.7a: PT for input "int a,b;"

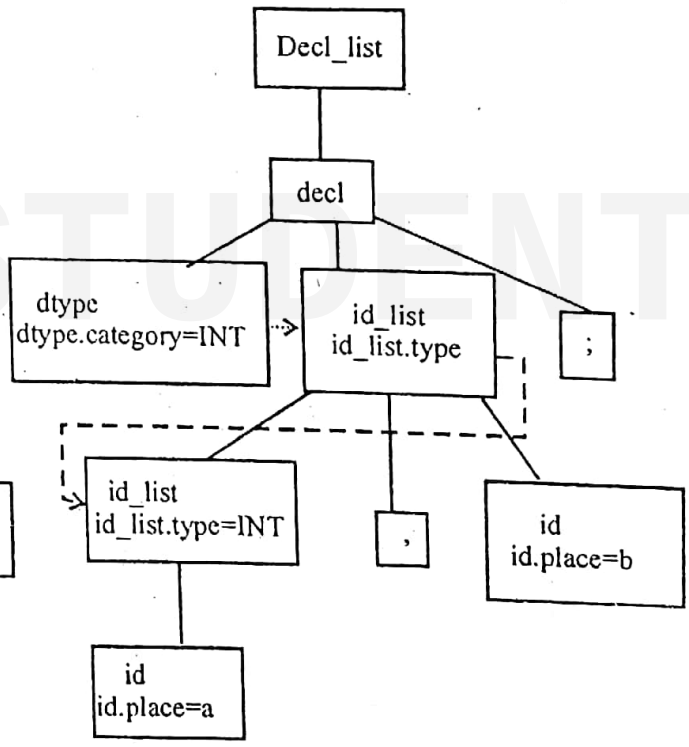
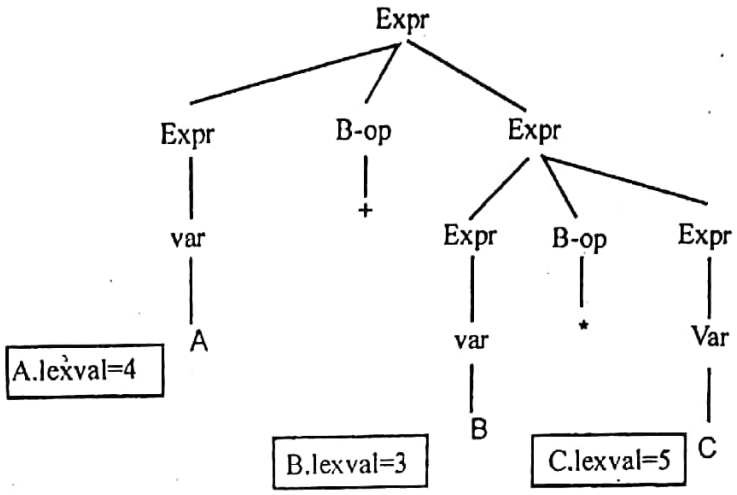


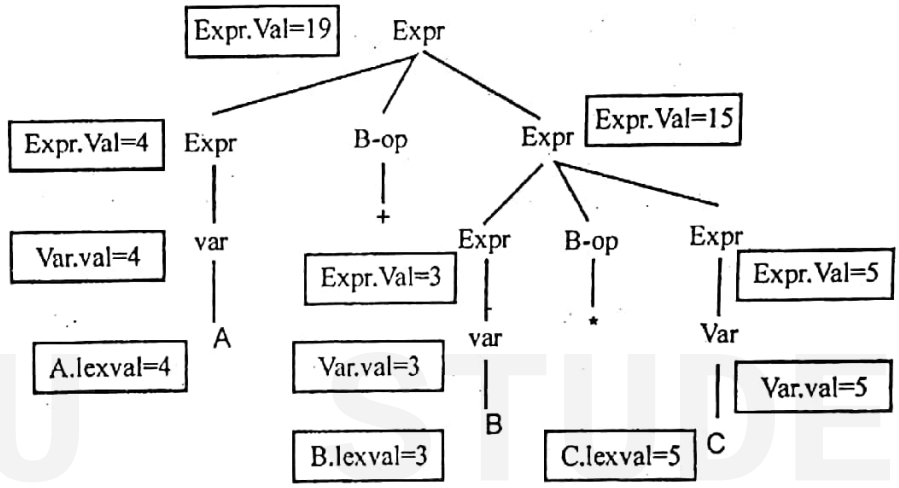
Fig 6.7b : Decorated PT with dependency

As shown in Fig 6.7b dtype category attribute depends on id_lists type, which depends on id's type. Such a graph is called as **dependency graph**. The topological sort of this graph gives the order in which the attributes associated with the nodes in the parse tree are to be evaluated.

Example 6.6: Draw the annotated parse tree for the following parse tree:



The annotated parse tree for the same is given below.



Example 6.7: Write down the syntax directed translation for the following CFG.

$L \rightarrow E$

$E \rightarrow E1 + T$

$E \rightarrow T$

$T \rightarrow T1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow digit$

The SDD is given below:

Production	Semantic rules
$L \rightarrow E$	print(E.val)
$E \rightarrow E1 + T$	E.val = E1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T1 * F$	T.val = T1.val * F.val

$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	Fval = digit.lexval

Example 6.8: Write the SDD for the following grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow I$

$I \rightarrow I$ Digit

$I \rightarrow$ Digit

Also show the actions of the LR parser.

The SDD is given below:

$S \rightarrow E\$$	{ Print E.VAL }
$E \rightarrow E(1) + E(2)$	{ E.VAL := E(1) .VAL + E(2).VAL }
$E \rightarrow E(1) * E(2)$	{ E.VAL := E(1) .VAL * E(2).VAL }
$E \rightarrow E(1)$	{ E.VAL := E(1) .VAL }
$E \rightarrow I$	{ E.VAL := I .VAL }
$I \rightarrow I(1)$ Digit	{ I.VAL := I(1) .VAL * 10 + LEXVAL }
$I \rightarrow$ Digit	{ I.VAL := LEXVAL }

The actions of the LR parser are shown below for the input string 123 * 5.

S.NO	INPUT	STATE	VALUE	PRODUCTION USED
1	123*5$	-	-	-
2	23*5$	I	1	I → digit
3	3*5$	I	12	I → I*10+digit
4	*5$	I	123	I → I*10+digit
5	5$	E	(123)	E → I
6	$	E*	(123)	
7	$	E*I	(123) 5	I → digit
8	$	E*E	(123) (5)	E → I
9	$	E	615	E → E*E
10	$	S	PRINT 615	S → E$

Example 6.9: Are the attributes in the CFG given below synthesized or inherited? Give reasons.

var → IntConstant

{ $0.val = $1.lexval; }

Expr → Var

{ $0.val = $1.val; }

Expr → Expr B-op Expr
{ $0.val + $2. val ($1.val, $3.val); }
B-op → +
{ $0.val = PLUS; }
B-op → *
{ $0.val = TIMES; }

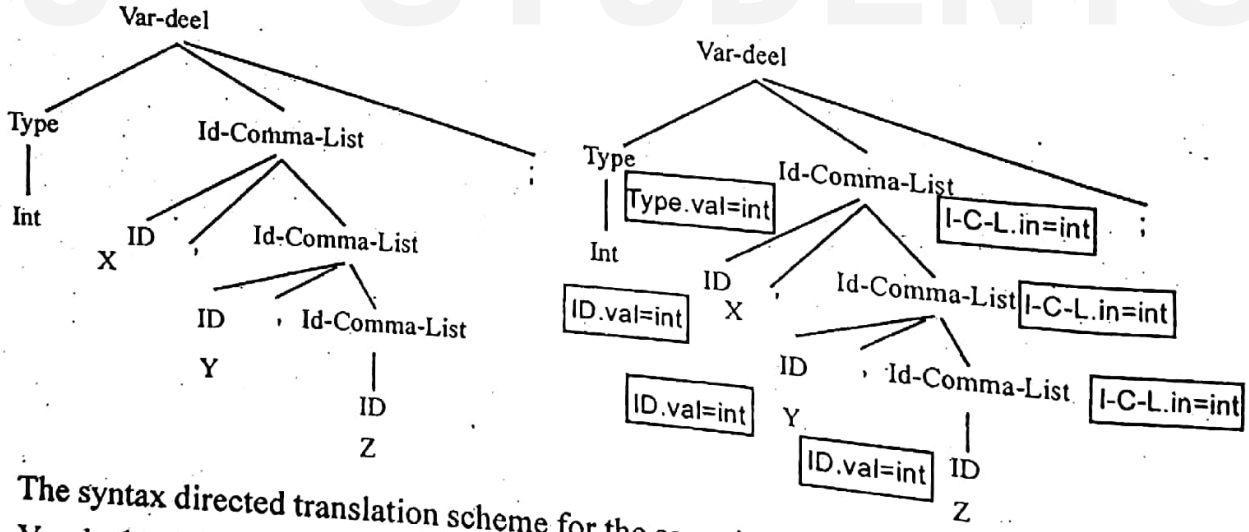
Consider the flow of the attributes in the 'Expr' syntax-directed definition. The LHS attribute is computed using the RHS attributes. This procedure is purely bottom up. Attribute values of all children (RHS) in the parse tree is computed and then used to compute the attribute value of the parent on the LHS. Thus the attributes are **Synthesized attributes** as it is computed purely bottom-up. Such a grammar plus semantic actions is called an **S-attributed definition**

Example 6.10: Are the attributes in the following grammar synthesized or inherited? Give reasons.

Var-decl → Type Id-comma-list ;
Type → **int | bool**
Id-comma-list → ID
Id-comma-list → ID , Id-comma-list

Consider an example input string int x, y, z ;

The parse tree and annotated parse tree for the input string are given below:



The syntax directed translation scheme for the same is given below:

Var-decl → Type Id-comma-list ;
{ $2.in =$1.val; }
Type → int | bool
{ $0.val = int; } & { $0.val = bool; }

Id-comma-list → ID

{ $1.val =$0.in; }

Id-comma-list → ID , Id-comma-list

{ $1.val =$0.in; $3.in = $0.in; }

Here, ID takes its attribute value from its parent node. Id-comma-list takes its attribute value from its left sibling type. Computing attributes purely bottom-up is not sufficient in this case. Hence **inherited attributes** are used. Inherited attributes are attributes that are computed at a node based on attributes from siblings or the parent. Typically, synthesized attributes and inherited attributes are combined. In such a case, it is possible to convert the grammar into a form that only uses synthesized attributes.

A syntax-directed definition is L-attributed if for a CFG rule of the form $A \rightarrow X_1..X_{j-1}X_j..X_n$ two conditions hold:

i. Each inherited attribute of X_j depends on $X_1..X_{j-1}$

ii. Each inherited attribute of X_j depends on A

These two conditions ensure left to right and depth first parse tree construction. Every S-attributed definition is L-attributed.

Example 6.11: Are the following SDDs, inherited or synthesized?

a. E→E +E {E.VAL=E.VAL+E.VAL}

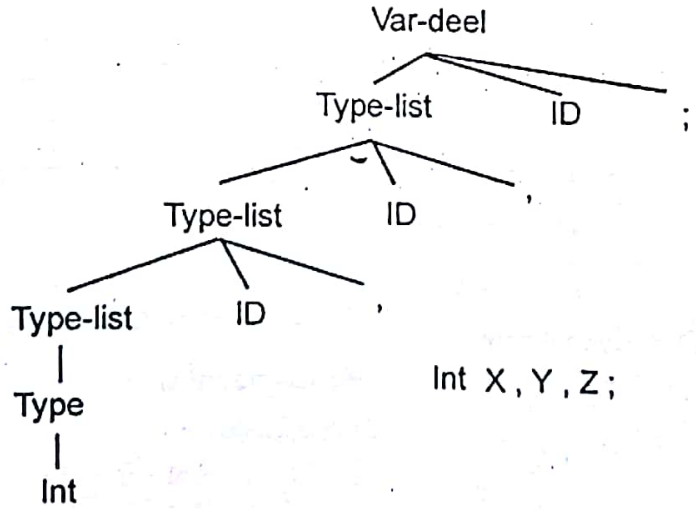
b. A→XYZ {Y.VAL=2*A.VAL}

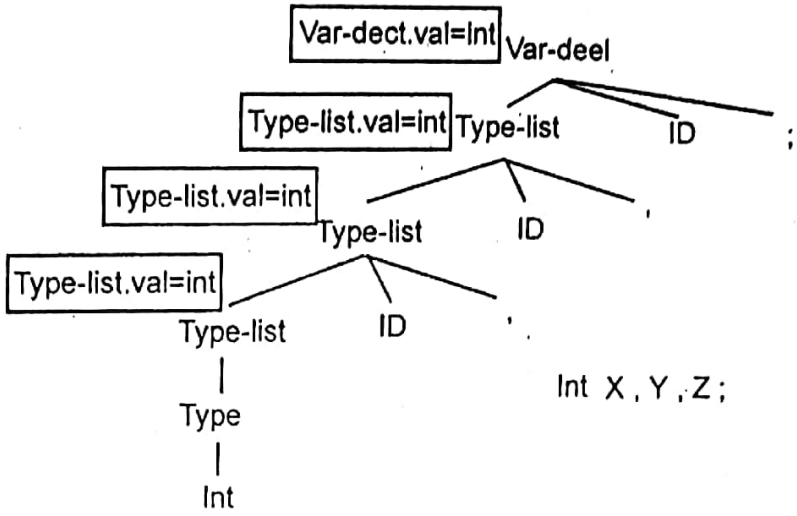
Ans:

a. E→E +E {E.VAL=E.VAL+E.VAL} is synthesized

b. A→XYZ {Y.VAL=2*A.VAL} is inherited.

Example 6.12: Remove the inherited attributes in the declaration example (example 6.10)





The syntax directed translation is given below:

Var-decl → Type-list ID ;
{ $0.val = $1.val; }

Type-list → Type-list ID
{ $0.val = $1.val; }

Type-list → Type
{ $0.val = $1. val; }

Type → int | bool
{ $0.val = int; } & { $0.val = bool; }

The translation scheme is a CFG, where each rule is associated with a semantic attribute.

Example 6.13: Convert the following SDD into SDD with synthesized translations:

E → TR
R → +T { print ('+'); } R
R → -T { print ('-'); } R
R → ε
T → id { print(id.lookup); }

For this purpose, non marker terminals N and M are used as follows:

E → TR
R → +TMR
R → -TNR
R → ε
T → id { Print(id.lookup); }
M → ε { Print('+'); }
N → ε { Print('-'); }

The translation of the parse tree into intermediate form is based on the syntax of the input language and is often referred to as **Syntax Directed Translation.**

The front end of most of the compilers translates the input source into one of the following **intermediate forms** as explained below:

1. Postfix Notation
2. Abstract Syntax Tree
3. Three Address Code

The creation of DAG is identical to the AST except for the extra check to determine whether a node with identical properties already exists. In the event of the node already created, we merely chain the existing node avoiding a duplicate node. The DAG is optimal on space as compared to the AST.

There are 3 common **implementations** for the TAC statements in a compiler. They are:

1. Quadruples
2. Triples
3. Indirect Triples

The intermediate code cannot be generated directly from the Parse Tree (PT). The nodes have to be populated with more information. This process is called decorating the parse tree. **Attribute grammars** are a method to decorate or annotate the parse tree

There are two types of attributes. They are:

1. If the addition of the parent node (LHS) depends on attributes of its children (RHS), such attributes are **synthesized attributes.**
2. If the attributes of the child depends on the attribute of parent node, then such an attribute is called **Inherited attribute.**

6.5 Methods of Translation into intermediate forms

There are four methods of translating the input source into intermediate code. They are

1. **Parse tree method**: In this method the parse tree generated during syntax analysis phase is used to create the dependency graph. The value of each attribute in dependency graph is evaluated. This evaluation yields the intermediate code.
2. **Bottom up translation**: In this method the semantic rules are evaluated and used to generate the intermediate code when the production is reduced. It can be applied easily to S-attributed grammars.
3. **Top-down translation**: Here, the semantic rule is evaluated and the intermediate code is generated when the production is expanded. The overload of creating parse tree dependency graphs can be overcome in this method.
4. **Recursive evaluator model**: In this method, the parse tree is constructed. Then

it is traversed and the semantic rules are evaluated in a particular order. This method can be used for a wide class of grammars, but it is expensive as it has to build the PT and then traverse it.

These methods are explained in the following sections

6.5.1 Parse tree method:

In this method, the translation of source code to intermediate code occurs in 3 steps. They are:

1. **Conversion of input source program into parse tree:** This process is done in the parsing phase itself. For example, consider the syntax directed translation for assignment statement in section 6.4. The parse tree for the input "i= j+k" is given in Fig 6.8.

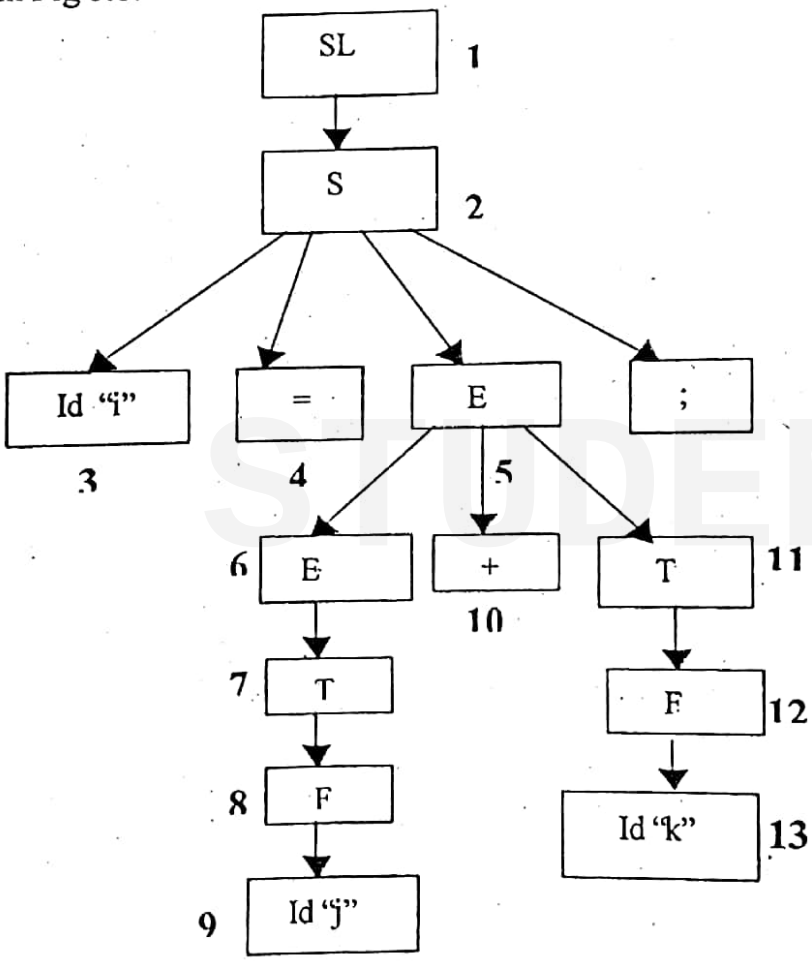


Fig 6.8 PT for input "i = j + k"

The rules for creation of parse tree are given below. Consider a production P→ABC

1. Create a node P
2. Make the nodes A, B, C as children to P from left to right. This process is carried out by performing the following steps (a to c):

a.. The PT is traversed using **depth first traversal method** as given below :

```
Procedure dfvisit (node n)
{
    process_the_ node ( );
    for each of the child m of n from left to right
    {
        dfvisit (m);
    }
}
```

Using this method, the parse tree given in fig 6.8 is evaluated as shown by the numbering of the nodes in Fig 6.8

b. **Creation of depending graph**: The aim of drawing the dependency graph is to identify the dependency of the attributes of the node of the parse tree on one another. For example

The statement node (S) has attribute

S.code = E.code || S.place = E. place

Expression (E) node has attribute E.code and E.place.

Consider the statement portion of syntax directed translation

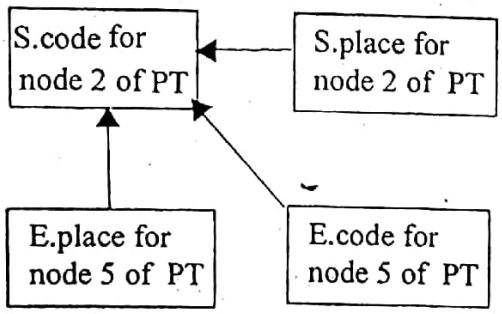
S → id := E ; S.place = id.place

S.code = E.code || S.place = E.place

From this we can see that

- S. code in node 2 depends on E.code (at node5)
- S.code in node 2 depends on S.place (at node 2)
- S.code in node 2 depends on E.place in node 5

Hence dependency graph for node 2 is given below



The creation of the dependency graph is a 2 step process as given below :

i. Creation of dependency graph node for each of the attributes corresponding to each of the parse tree node

ii. Creation of edges from one dependency graph node 'a' to another graph node 'b', if 'b' depends on 'a'.

Then the topological sort of the directed acyclic graph is performed. In this ordering, if an edge occurs from mi to mj , then the topological sort will have node mi ahead

of mj. For example for the dependency graph shown in Fig 6.8 the topological sort is in the order shown by the numbering of the node (ie) 1, 2, 3, 4.

> The **dependency graph** identifies the dependency of the attributes of the node of the parse tree on one another.
>
> The **topological sort** of the dependency graph tells us which node comes ahead of another node.

c. **Evaluation of attribute using the dependency graph:** The order of evaluation of attribute is obtained from a topological sort of the dependency graph. The evaluation of attributes yields the intermediate code. The value for each of the attribute represented by a node in the dependency graph is computed using the related semantic rule. If the dependency graph (Fig 6.9) is drawn and topological sort is done, the result of traveling it is "i = j+k"

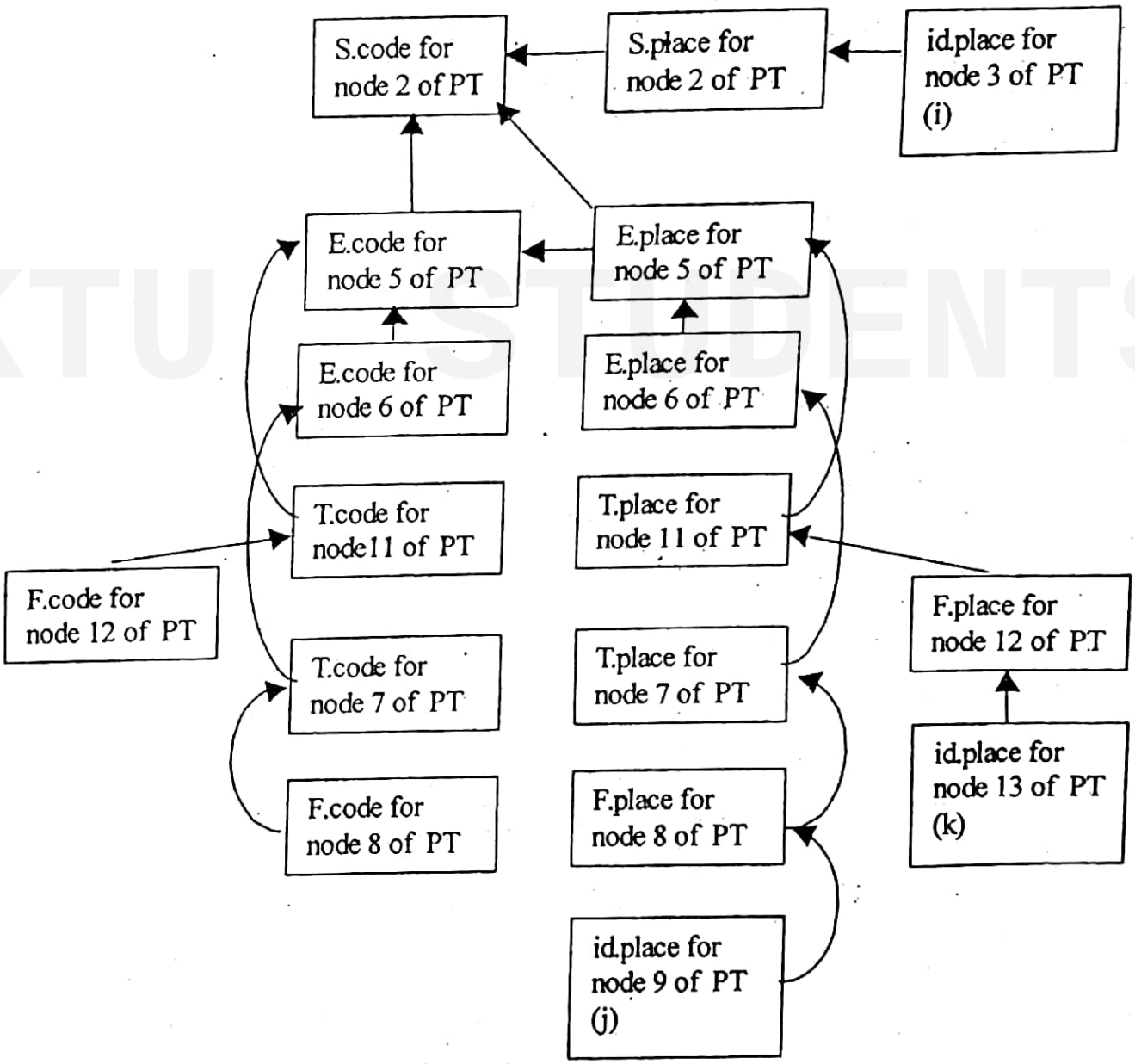


Fig 6.9: Dependency graph

The algorithm for DFS evaluates the attributes as given below:

```
procedure dfvisit (node *n )
    {
       for each of the child m of n from left to right
    {
       evaluate inherited attribute of m
       dfvisit (m);
    }
/* when all children of node N are processed*/
evaluate synthesized attribute of node n
}
```

Consider the production A→x1, x2, ...xn Depth First evaluation method is as follows:

1. The inherited attribute of node xj depends on attribute of x1, x2, ...xj-1 but not on its siblings like xj+1....xn (as xj+1....xn should be evaluated only after xj).
2. The uninherited attributes of any particular node xj can depend on A, as A would have been evaluated before xj.

For example consider SDD given in table for declarations in section 6.3. Here,

Id _list.type = dtype.category is 'L' attribute.

But dtype is on RHS of id_list. Then the SDD is not L-attribute grammar as it violates rule2.

A translation scheme is the notation that helps us to represent the code fragments associated with each reduction. A translation scheme is thus a CFG in which the attributes are associated with the grammar symbols and semantic action enclosed between braces is given to right of the production.

For example consider the translation scheme given below:

S → { B. attr 1 = 100} B {C. attr 1 = 200} C

B → b {B.attr 2=f (b.val ,B.attr 1) }

C → c (C.attr 2 = g(c.val, C.attr1)

The parse tree for the translation scheme is given below (Fig 6.10), along with the DF traversal of the tree shown by dotted lines

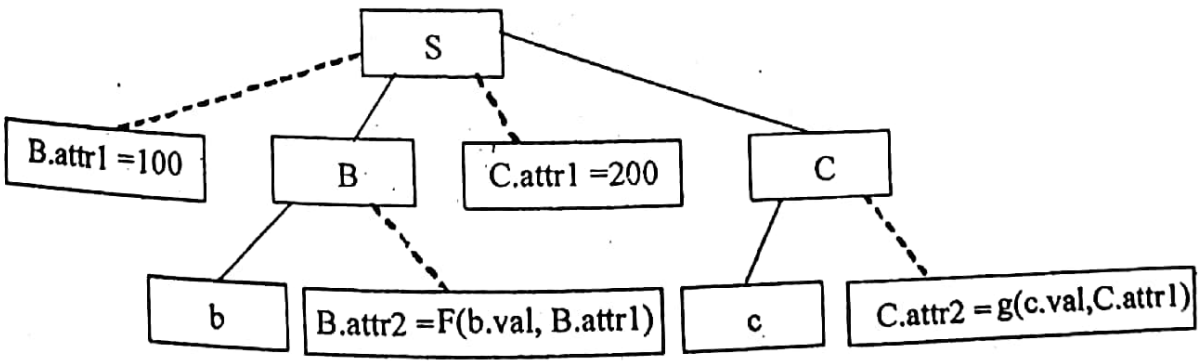


Fig 6.10 Semantic action in parse tree

The semantic actions are connected by dotted lines as shown Fig 6.10.

The rules for L attribute definition are given below:

1. An uninherited attribute for a symbol on right side of the production is an action before that action.
2. An action must not refer to a synthesized attribute of a symbol to the right of the action.
3. A synthesized attribute for the non-terminal at left is computed only after all attributes it references have been computed. The action of computing such attributes can usually be placed at the end of the right side production.

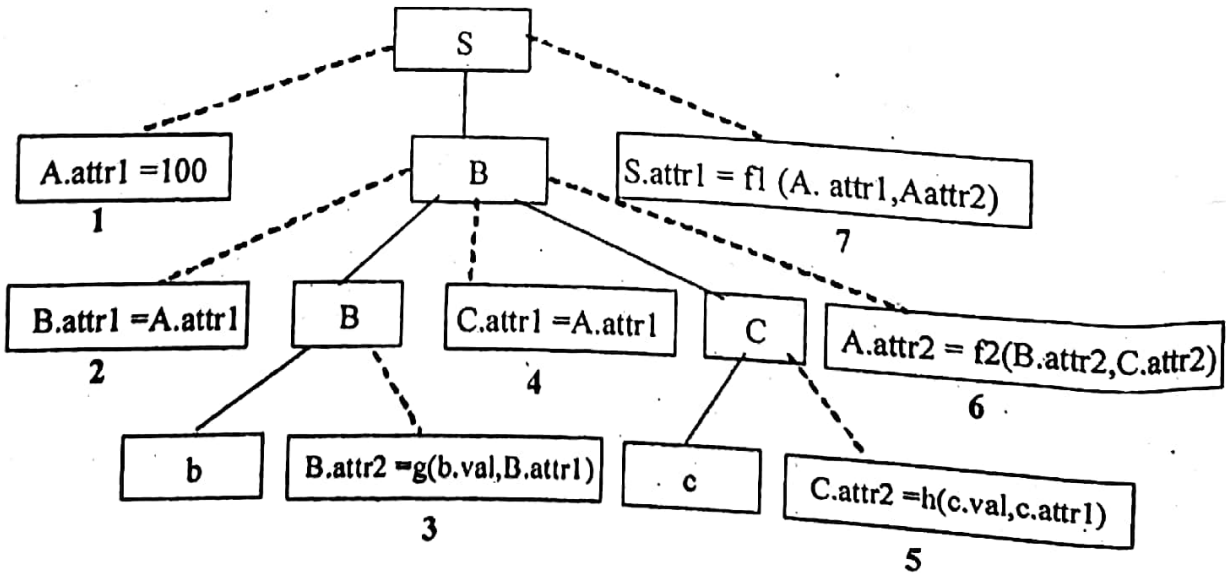
For example, consider the following SDD

Production	Semantic Rules
S→A	A.attr 1=100 S.attr 1=f1(A.attr1,A.attr2)
A→B C	B.attr1=A.attr1 C.attr1=A.attr1 A.attr2=f2(B.attr2,C.attr2)
B → b	B.attr2=g(b.val,B.attr1)
C → c	C.attr2=h(c.val,C.attr1)

The translation scheme given below:

1. S → { (A.attr 1=100) A (S.attr = f1 (A.attr1, A.attr2) }
2. A → { (B.attr1=A.attr1) B (C.attr1=A.attr1) C (A.attr2=f(B.attr2,C.attr2) }
3. B → b { B.attr2 = g (b.val, B.attr1)}
4. C → c { C.attr2 = h (c.val, C.attr1) }

In production 1, attr1 of A is uninherited ; while S.attr1 is synthesized. In production 2, attr1 is uninherited for B and C while attr2 of A is synthesized. The attr2 of B and C are also synthesized. The DF traversal of the parse tree with order of the evaluation of attributes is given below

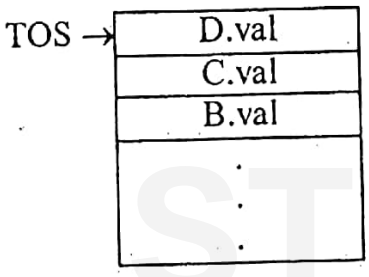


A translation scheme is the notation to represent the code fragments associated with each reduction. A translation scheme is a CFG in which the attributes are associated with the grammar symbols and semantic action enclosed between braces is given to right of the production

6.5.2 Bottom up translation

This method is used during bottom up parsing in LR parsers. LR parsers use stack while scanning input. In order to facilitate the evaluation of semantic rules and translation of input source, an additional stack containing the values of attributes of symbols involved in reduction is used. Such a stack is called VAL stack. The 'shift' action pushes an attribute of input grammar symbol on VAL stack. The reduce action pops out as many elements as present in RHS of the production and pushes an attribute of LHS of the production on the VAL stack. If a symbol has no attribute, then the value pushed onto stack is undefined.

For example, consider the production A → BCD, the contents of VAL stack are given below.



Bottom-up translation of S-attributed definitions is explained in this paragraph. For assignment statements, the translation scheme is shown below. The **emit** statement in the translation scheme emits the intermediate code as and when reduction happens.

1. SL → SL
   | S

2. S → id := E; {emit (id.place := E.place)}

3. E → E+T { E.place= NewTemp()
   emit(E.place = E1.place + T.place)}
   | E-T { E.place= NewTemp()
   emit(E.place = E1.place - T.place)}
   | T { E.place = T.place}

4. T → T*F { T.place= NewTemp()
   emit(T.place = T1.place * F.place)}
   | T/F { T.place= NewTemp()

			emit(T.place = T1.place / F.place)}
		| F	{ T.place= F.place}
5.	F	→ (E)	{ F.place=E.place }
		| ID	{ F.place=ID.place }
		| CONST	{ F.place= NewTemp()
			emit(F.place=CONST.val) }

Bottom-up translation method is used during bottom up parsing in LR parsers. LR parsers use stack while scanning input. In order to facilitate the evaluation of semantic rules and translation of input source, an additional stack containing the values of attributes of symbols involved in reduction is used. Such a stack is called **VAL stack.**

STATE stack	VAL stack	input	Operation
$0	$	i:=j+k;$	Shift
$0 1	$ID.place=i	:=j+k;$	Shift
$0 1 4	$ID.place,undefined	j+k;$	Shift
$0 1 4 8	$ID.place,undefined,ID.place=j	+k;$	Reduce F → id
$0 1 4 10	$ID.place,undefined,ID.place=j,F.place=j	+k;$	Reduce T → F
$0 1 4 11	$ID.place,undefined,ID.place=j,F.place=j; T.place = j;	+k;$	Reduce E → T
$0 1 4 9	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j	+ik;$	Shift
$0 1 4 9 13	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j,undefined	k;$	Shift
$0 1 4 9 13 8	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j,undefined,ID.place=k	;$	Reduce F → ID
$0 1 4 9 13 10	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j, undefined, ID.place=k, F.place=k	;$	T→ F
$0 1 4 9 13 19	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j, undefined, ID.place=k, F.place=k, T.place=k	;$	Reduce E → E+T
$0 1 4 9	$ID.place,undefined,ID.place=j,F.place=j, T.place = j; E.place=j, undefined, ID.place=k, F.place=k, T.place=k, E.place=j+k **emit (t0 = j + k)**	;$	Shift 15
$0 1 9 15	$ID.place,undefined,ID.place=j,F.place=j, E.place=j,undefined,ID.place=k,F.place=k, T.place=k, E.place=j+k, E.place=j+k ;	;$	Reduce S → id := E;
$0	$ID.place,undefined,ID.place=j,F.place=j, E.place=j,undefined,ID.place=k,F.place=k, T.place=k, E.place=j+k, E.place=j+k, S.place=i = j+k, undefined **emit (i = t0)**	$	
$0	completed	$	accept

The actions of the LR parser in the STATE stack along with contents of VAL stack are given in the table . SLR parse table for expression in chapter 5 (example 5.9) is to be considered for state changes .

Emitted intermediate code is given below:

```
t0=j+k;
i=t0;
```

For L-attributed definition, the semantic actions between the non-terminals shift to modify so that they occur at the end. For this, extra ε productions are added and the grammar is restructured. Consider the translation scheme to convert infix to postfix format as shown below:

```
E  →T                          E1
E1→+T              {print '+'} E1
   | -T            {print '-'} E1
   | ε
T  → CONST         {print CONST.val}
```

This scheme is restructured to include semantic actions in the end as follows

```
E  →T E1
E1→+T M1 E1
   | -T M1 E1
   | ε
T  → CONST {print(CONST.val)}
M1→ ε {print '+'}
M2→ ε  {print '-'}
```

Now the code is generated using bottom up method as already illustrated.

6.5.3 Top down translation method

It is used along with top down parser. Hence, left recursion should be removed. After eliminating left recursion, the SDD for assignment statements is given below.

S.No.	Production	Semantic action (translation scheme)
1	SL→SL SR	
2	S →S SR	
3	|∈	
4	S →ID=E	{emit (ID.place=E.place)}
5	ER →T	{ ER.i= T.place} ER {E.place=ER.S;}
6	ER →+ T ER1	{ place=NewTemp() emit(place, := , ER.i , + , T.place) ER.i=place } {ER.s=ER1.s}

7	ER → - T ER1	{ place=NewTemp(); emit(place, :=, ER.i, _ , T.place); ER1.i=place } { ER.s=ER1.s}
8	| ε	{ER.s=ER.i}
9	T → F TR	{TR.i=F.place;} { T.place=TR.s;}
10	TR → / F TR1	{ place=NewTemp() emit(place, :=, TR.i, *, F.place); TR1.i=place; } { TR.s=TR1.s }
11	TR → / F TR1	{ place=NewTemp(); emit(place, :=, TR.i, / , F.place); TR1.i=place } { TR.s=TR1.s}
12	TR → ε	{ TR.s =TR1.i}
13	F → ID	{ F.place=ID.placei}
14	F → CONST	{ place=NewTemp(); emit(place , := , CONST, value); F.place=place }
15	F→ (E)	{ place=NewTemp(); emit(place, := , E.place); F.place=place }

Top down/ recursive descent parsers are implemented using procedures as given below. In these procedures, inherited attributes (.i)have been evaluated previously; hence they are passed by value, whereas synthesized attributes (.s) should be passed by reference.

```
int ER (ER_i, &ER_S)
{
   if(match('+'))
   {
   sign = '+';                    /*for sign*/
   }
   else if (match('-'))
```

```
    {
    sign = '-';
    }
    else
    {
    ER_s= ER1_i;
    return(SUCCESS);              /*for ER1*/
    }
    if(T(T.place)==SUCCESS)
    {
        place=NewTemp();
        emit(place.()ER_i(), sign(),T.place());
        ER_i=place;               /*for    T   */
    }
    if(ER(ER1_i,ER_s)==SUCCESS)
    {
        ER_s=ER1_s;               /*for E*/
        return(SUCCESS);
    }
}
return(FAILURE);
}
```

Similarly the code for 'S' is given below:

```
int S()
{
    if(curr_token==ID)
    {
        iID_place=strdup(yytext);
        match(ID);
        if(match(':='))
        {
            if(E(E_PLACE)== SUCCESS)
            {
                if(match(';'))
                {
                    emit(ID_place.(),E_place(),',');
                    return(SUCCESS);
                }
                }
        }
      }
  return(FAILURE);
  }
```

To generate the intermediate code for "i=j+k" using the top down translation scheme, SL() procedure is called. This calls the S() procedure given above. In this procedure ID is matched. Then it searches for ':=' token. If it matches, E's E.place has to be got.

It is got by calling E() procedure. E() procedure has to call ER() procedure. This ER() procedure checks for '+' sign. Then it matches T() procedure to return success. This returns ER+T value. For this again ER.place value is obtained which returns success. Once this is obtained E() procedure returns success. Then a match is obtained in S() procedure. If a 'i' is encountered then it emits the three address code for ID's place and E's place values (ie i=t0)

Hence the TAC emitted is

t0 = k+j

i = t0

Top down/ recursive descent parsers are implemented using procedures. Here, the inherited attributes (.i)have been evaluated previously. Hence they are passed by value. The synthesized attributes (.s) should be evaluated later. Hence they are passed by reference.

6.5.4 Recursive evaluator method

The parse tree method offers only low compile time efficiency, but is capable of handling all types of grammars that are not circular. Top-down and bottom-up translation interleaves translation with parsing and are compile time efficient. They can handle a restricted class of grammars. In parse tree method a parse tree is created, a dependency graph is drawn and then the semantic rules of the parse tree nodes are evaluated in topological sorted order.

In recursive evaluator method, a parse tree is explicitly created during parsing. Then it is traversed and the semantic rules are evaluated using mutually recursive functions. The nodes can be visited left to right or right to left. The main considerations involved in arriving at the correct order for the traversal of the parse tree are listed below:

1. All inherited attributes at a node are computed before the first visited.
2. The synthesized attributes are computed before we leave the node for the last time.

For example consider the following translation scheme for declaration

rule 3: declaration → id_list '=' dtype

{ id_list.type = dtype.category; }

The recursive procedure of this production initially creates an id_list and a dtype_list.Then it obtains category synthesized term dtype-node (passing by reference as its value is evaluated later). Then this category is passed onto id_list node's type recursively. The main characteristic of this method is to evaluate the semantic rules of the root node of the tree. This internally evaluates the semantic rules of all its children. This procedure is done recursively. This method is similar to top down translation functions.

```
void declaration_node :: eval_semantic_rules( )
{
    id_list_node  * id_list_ptr;
    d_type_node   * dtype_node_ptr;
    if ( rule_no == 3) {
        /*decl →  id_list ' . ' dtype ' ; '  */
        id_list_ptr = (id-list_node *) children[0];
        dtype_node_ptr = (dtype_nod *) children[1];

        /* getting the 'category' synthesized term dtype_node */
        dtype_node_ptr →  eval_semantic_rules( dtype_category);

        /* passing the 'category' to the children of id_list */
        id_list_ptr →  eval_semantic_rules(dtype_category);
    }
}
```

6.5.5 Comparison of Translation Methods

Parameter	Parse tree Method	Bottom up	Top down	Recursive evaluation
Principle on which the method is based	Creates a parse tree, makes a dependency graph, evaluates the attributes based on topological sort of dependency graph and gives out the translated code	Does not create parse tree. Emits out the translated code during bottom up parsing	Does not create the parse tree. Emits out translated code during top down parsing	Creates parse tree, traverses it to evaluate attributes and emits the translated code. These two steps are delinked totally.
Applicability	Can be applied to any grammar provided there are no cycles in dependency graph	LL(1) and most of the LR(1) grammar	LL(1) grammar	Any grammar
Efficiency	Creation of parse tree and dependency graph reduces the efficiency	Efficient since we do not create a parse tree	Efficient since we do not create a parse tree	Creation of parse tree reduces the efficiency

6.5.6. Translation to other intermediate forms.

a. Translation to postfix intermediate code

SL → SL S	
| S	
S → ID = E ;	{ S.code = ID.place || E.code || '=' emit (S.code)}
E → E + T	{ E.code = E1.code || T.code || '+' }
| E - T	{ E.code = E1.code || T.code || '-' }
| T	{ E.code = T.code }
T → T * F	{ T.code = T1.code || F.code || * }
| T / F	{ T.code = T1.code || F.code || / }
| F	{ T.code = F.code }
F →(E)	{ F.code = E.code }
| ID	{ F.code = ID.place }
| CONST	{ F.code = CONST.value }

For example, consider the input "i = j + k", the postfix code is generated as follows:

SL → S	
S → ID = E	{ S.code = ID.place || E.code || '=' emit (S.code); }
E → E + T	{ E.code = E1.code || T.code || '+' }
T → F	{ T.code = F.code }
F → id	{ F.code = ID.place }

Based on this jk+ is evaluated, then ijk+= is evaluated.

b. Translation to abstract syntax trees

Here, we construct a node for each operator and operand. The children of the operator node are root nodes representing the sub expression constituting the operands of that operator. Each node in a syntax tree is implemented as a record with several fields. In the node of an operator one field identifies the operator and the remaining fields contain pointer to the nodes for the operands.

SL → SL S	{add_child(SL1.nptr,S.nptr) SL.nptr=SL1.nptr}
|S	{SL.nptr=new SLnode(S.nptr)}
S → id := E	{S.nptr =n ew assignnode (:=,ID.nptr,E.nptr)}
E → E+T	{E.nptr=newbin_op_node('+', E1.nptr,T.nptr)}
| E-T	{E.nptr=newbin_op_node('-', E1.nptr,T.nptr)}
|T	{E.nptr=T.nptr}
T → T * F	{T.nptr=newbin-op_node('*', T1.nptr,F.nptr)}
| T / F	{T.nptr=newbin-op_node('/', T1.nptr,F.nptr)}

| F	{T.nptr=F.nptr}
F → (E)	{F.nptr=E.nptr}
| id	{E.nptr=id.nptr}
| CONST	{F.nptr=CONST.nptr}

For example consider the input statement "i=j+k"; the syntax tree is generated as shown below:

SL→ S {newnode is created for S}

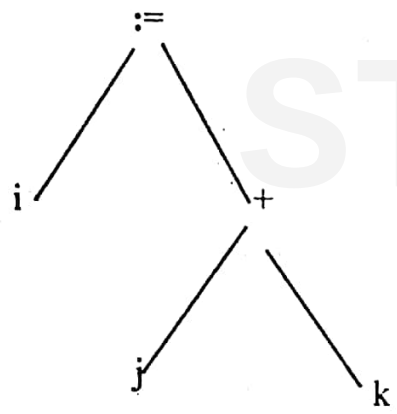
S→ id:=E {new assignment node := with left and right child 'id' and 'E' respec tively are created}

:= level 0

$\left.\begin{matrix} i \\ + \end{matrix}\right\}$ level 1

$\left.\begin{matrix} j \\ k \end{matrix}\right\}$ level 2

is created at output. The equivalent AST is



Example 6.14 : Give the top-down translation scheme for the following grammar:

E → E+T
{ $0.val = $1.val +$3.val; }

E → E-T
{ $0.val = $1.val +$3.val; }

T → IntConstant
{ $0.val = $1.lexval; }

B → T
{ $0.val = $1.val; }

T → (E)
{ $0.val = $1.val; }

The trace for the input string "id val=3 * id val=2 is given below:

Stack	Input	Action	Attributes
0	(id) * id $	Shift 5	
0 5	id) * id $	Shift 8	a.Push id.val=3;
0 5 8	) * id $	Reduce 3 F → id, Pop 8, goto [5,F]=1	{ $0.val = $1.val } a.Push id.val=3;
0 5 1	) * id $	Reduce 1 T→ F Pop 1, goto [5,T]=6	{ $0.val = $1.val } a.Push id.val=3;
0 5 6	) * id $	Shift 7	{ $0.val = $1.val }
0 5 6 7	* id $	Reduce 4 F → (T) Pop 7 6 5, goto [0,F]=1	3 pops; a.Push 3

Stack	Input	Action	Attributes
0 1	* id $	Reduce 1 T → F, Pop 1' goto [0'T]=2	{ $0.val = $1.val } a.pop; a.Push 3
0 2	* id $	Shift 3	a.Push mul
0 2 3	id $	Shift 8	a.push id.val=2
0 2 3 8	$	Reduce 3 F → id, Pop 8, goto [3,F]=4	a.pop a.Push 2
0 2 3 4	$	Reduce 2 T→ T * F	{ $0.val = $1.val * $2.val; }
0 2	$	Pop 4 3 2, goto [0,T]=2	3 pops;
		Accept	a.Push 3*2=6

Example 6.17:

Consider the following SDD with inherited attributes, Show the actions of the LR parser.

E→ TR

{ $2.in = $1.val; $0.val = $2.val;}

R→ +TR

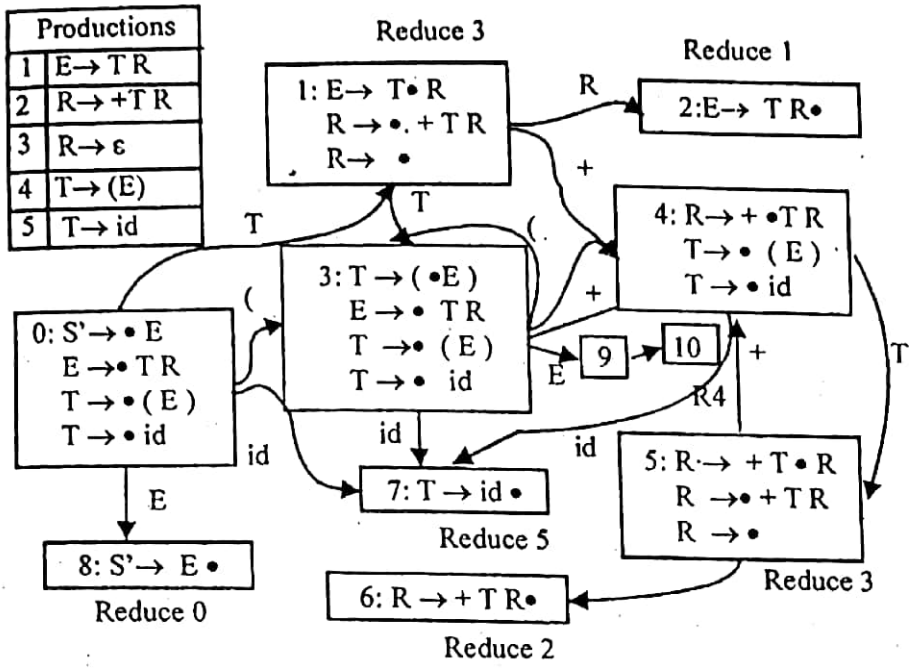
{$3.in = $0.in + $2.val: $0.val = $3.val; }

R→ ε { $0.val =$1.val; }

T→ (E) { #0.val = $1.val; }

T→ id {$0.val = id.lookup; }

The LR parser states are given below:



The trace for the input string "id val=3 * id val=2 is given below:

Stack	Input	Action	Attributes
0 1	Id + id $	Shift 7	{ $0.val =id.val
0 7	+ id $	Reduce 5 T → id	pop; attr.Push (3)
		Pop 7, goto [0'T]=2	$ 2.in = $1.val
0 1	+ id $	Shift 4	R.in := (1).attr }
0 1 4	+ id $	Shift 7	{ $0.val = id.val
0 1 4 7	$	Reduce 5 T → id,	pop; attr.Push (2); }
		Pop 7, goto [4,T]=5	{ $3.in = $0.in+$1.val
0 1 4 5	$	Reduce 3 R→ ε	(5).attr = (1).attr+2
		Goto [5,R]=6	$0.val = $0.in
			$0.val = (5).attr*5 }

Stack	Input	Action	Attributes
0 1 4 5 6	$	Reduce 2 R → T R	{ $0.val = $3.val
		Pop 4 5 6, goto [1,R]=2	pop; attr.Push (5); }
0 1 2		Reduce by E → T R	{ $0.val = $3.val
	$	Pop 1 2, goto [0,E]=8	pop; attr.Push (5); }
0 8	$	Accept	{ $0.val = 5
			attr.top =5; }

Conclusion:

In chapter 6, the need for intermediate code has been explained. The different forms of intermediate codes - postfix notation, AST and TAC have been explained along with their implementation and examples. The methods of implementation of TAC - quadruple, triples and indirect triples have also been explained. This chapter also ex-

similar action. It uses the operation mktable(top(tblptr)) to create a new symbol table. Here, top(tblptr) gives the enclosing scope of the new table. A pointer of the enclosing symbol table is then pushed along with the offset initialized to 0.

2. Including structure / record type for non-terminal T:

When the keyword **struct** is encountered, the marker non-terminal L creates a new symbol table for the fields of the structure. A pointer to this symbol table is then pushed onto stack with the offset 0. The action 'end' returns the width of the synthesized attribute T.width. Using pointer to the struct symbol table, the types and the widths of the individual fields can be retrieved from the symbol table.

Hence the translation scheme for the nested procedures with **struct** data types can be rewritten as follows:

CFG	Semantic actions
$P \rightarrow MD$	{ addwidth((top(tblptr),top(offset)); pop(tblptr); pop (offset) }
$M \rightarrow \varepsilon$	{ t = mktable(nil); push(t,tblptr); push(0,offset); }
$D \rightarrow D_1 ; D_2$	{ enter(top(tblptr), id.name,T.type, top(offset)); top(offset) = top(offset) + T.width }
$D \rightarrow$ T id;	{ t = top(tblptr); addwidth(t,top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.name, t) }
$D \rightarrow$ proc id; N D_1; S	
$N \rightarrow \varepsilon$	{ t = mktable(top(tblptr)); push(t,tblptr); push(0,offset); }
$T \rightarrow$ integer	{ T.type = integer; T.width = 4 }
$T \rightarrow$ real	{ T.type = real; T.width = 8 }
T → array [num] of T1	{ T.type = array (num.val, T1.type); T.width = num.val x T1.width }
$T \rightarrow$ & T1	{ T.type = pointer(T1.type); T.width = 4 }
T → record L D end	{T.type= record(top(tblptr)); T.width = top(offset); pop(tblptr);pop(offset); }