III. INTERMEDIATE CODE GENERATION

Syntax directed translation scheme - Three Address Code – Representation of three address code - Intermediate code generation for: assignment statements - Boolean statements - switch case statement –Procedure call - Symbol Table Generation.

Syntax Directed Translation:

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known. Semantic analysis involves adding information to the symbol table and performing type checking. It needs both representation and implementation mechanism.

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.

We associate Attributes to the grammar symbols representing the language constructs. Values for attributes are computed by Semantic Rules associated with grammar productions

Evaluation of Semantic Rules may:

- Generate Code;
- Insert information into the Symbol Table;
- Perform Semantic Check;
- Issue error messages;

There are two notations for attaching semantic rules:

1. Syntax Directed Definitions.High-level specification hiding many implementation details (also called Attribute Grammars).

2.Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

There are two kinds of attributes:

1. Synthesized Attributes: They are computed from the values of the attributes of the children nodes.
2. Inherited Attributes: They are computed from the values of the attributes of both the siblings and the parent nodes

Example: Let us consider the Grammar for arithmetic expressions (DESK CALCULATOR)
The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

Production	Semantic Rules
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Definition: An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.
Evaluation Order: Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or Post Order, traversal of the parse-tree.
The annotated parse-tree for the input 3*5+4n is:

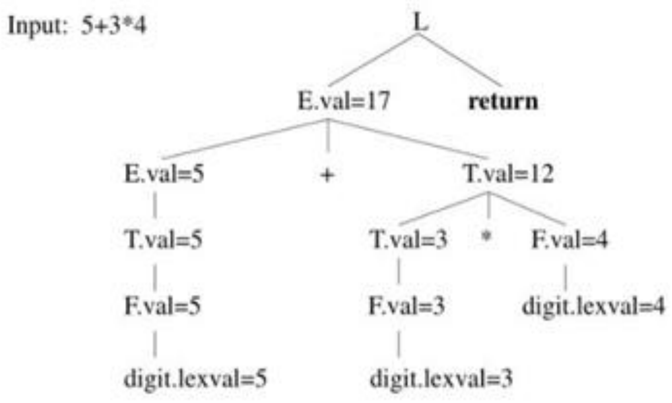
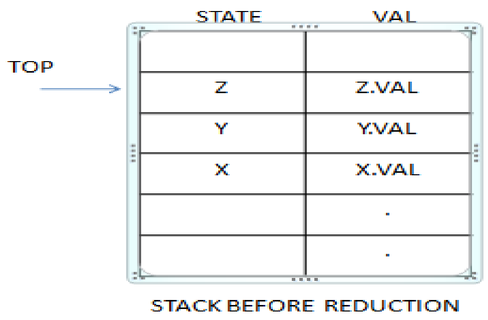


Fig 3.1 Annotated Parse Tree

Implementation of SDT for Desk Calculator:

Use an extra fields in the parser stack entries corresponding to the grammar symbols. These fields hold the values of the corresponding translations.



The grammar and corresponding Semantic action is shown below:

Production	**Semantic Action**
$S \longrightarrow E\ \$$	{ print E.VAL }
$E \longrightarrow E_1 + E_2$	{E.VAL := E_1.VAL + E_2.VAL }
$E \longrightarrow E_1 * E_2$	{E.VAL := E_1.VAL * E_2.VAL }
$E \longrightarrow (E_1)$	{E.VAL := E_1.VAL }
$E \longrightarrow I$	{E.VAL := I.VAL }
$I \longrightarrow I_1$ digit	{I.VAL := 10 * I_1.VAL + LEXVAL }
$I \longrightarrow$ digit	{ I.VAL:= LEXVAL}

The Annotated parse tree for the expression 23*4+5 is depicted in Fig 3.2.

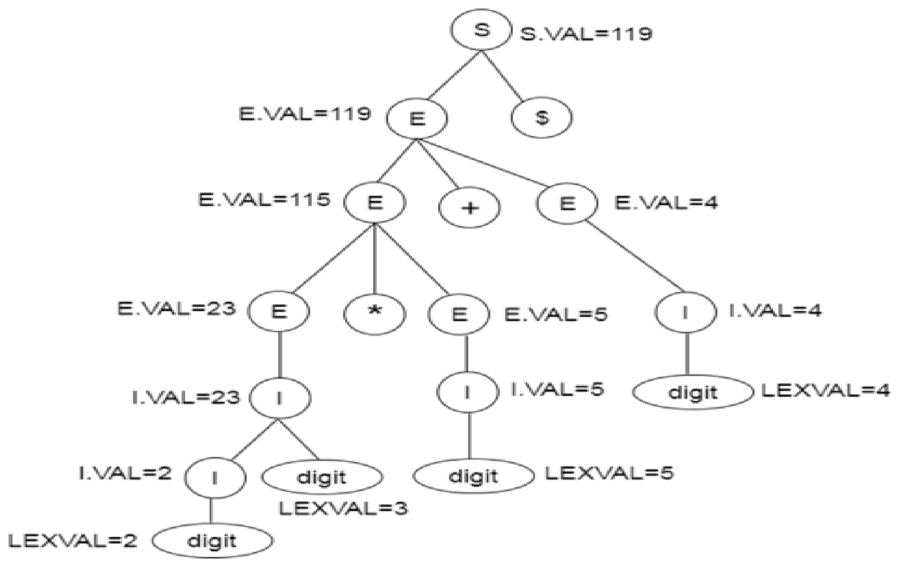


Fig 3.1 Annotated Parse Tree for 23*4+5$

The method of implementing the desk calculator is given using the program fragment represents the stack implementation to evaluate a given expression.

Production	**Program Fragment**
$S \rightarrow E\ \$$	print VAL[TOP]
$E \rightarrow E + E$	VAL[TOP] := VAL[TOP] + VAL[TOP-2]
$E \rightarrow E * E$	VAL[TOP] := VAL[TOP] * VAL[TOP-2]
$E \rightarrow (E)$	VAL[TOP] := VAL[TOP-1]
$E \rightarrow I$	none
$I \rightarrow I$ digit	VAL[TOP] := 10 * VAL[TOP] + LEXVAL
$I \rightarrow$ digit	VAL[TOP] := LEXVAL}

The sequences of moves to implement the evaluation of an expression are shown in Fig 3.3.

Moves	Input	STATE	VAL	Production used
(1)	23*5+4$	_	_	
(2)	3*5+4$	2	_	
(3)	3*5+4$	I	2	$I \rightarrow$ digit
(4)	*5+4$	I3	2_	
(5)	*5+4$	I	(23)	$I \rightarrow I$ digit
(6)	*5+4$	E	(23)	$E \rightarrow I$
(7)	5+4$	E*	(23) _	
(8)	+4$	E*5	(23) _ _	
(9)	+4$	E*I	(23) _ 5	$I \rightarrow$ digit
(10)	+4$	E*E	(23) _ 5	$E \rightarrow I$
(11)	+4$	E	(115)	$E \rightarrow E * E$
(12)	4$	E+	(115) _	
(13)	$	E+4	(115) _4	
(14)	$	E+I	(115) _4	$I \rightarrow$ digit
(15)	$	E+E	(115) _4	$E \rightarrow I$
(16)	$	E	(119)	$E \rightarrow E + E$
(17)	_	E$	(119) _	
(18)	_	S	PRINT 119	$S \rightarrow E\ \$$

Fig 3.3 Sequence of moves

Intermediate Code Representation:

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

1. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
2. The second part of compiler, synthesis, is changed according to the target machine.
3. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate codes can be represented in a variety of ways and they have their own benefits.

1. High Level IR - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
2. Low Level IR - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

- During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text.
- The complexity of this code lies between the source language code and the object code.
- The intermediate code can be represented in the form of:
  - Postfix notation,
  - Syntax tree,
  - Three-address code.

Postfix Notation:

The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b. The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the postfix notation is:

e1e2 +

No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.

The postfix representation of the expressions:

1. (a+b)*c → **ab+c***
2. a*(b+c) → **abc+***
3. (a+b)*(c+d) → **ab+cd+***
4. (a – b) * (c + d) + (a – b) → **ab - cd + *ab - +**

Let us introduce a 3-ary (ternary) operator ,the conditional expression. Consider **if e then x else y** ,whose value is

$$\begin{cases} x, if\, e \neq 0 \\ y, if\, e = 0 \end{cases}$$

Using ? as a ternary postfix operator, we can represent this expression as: **exy?**

Evaluation of Postfix Expression:

Consider the postfix expression ab+c*.Suppose a,b and c have values 1,3 and 5 respectively. To evaluate 13+5*,perform the following:

1. Stack 1
2. Stack 3
3. Add the two topmost elements; pop them off stack and the stack the result 4.
4. Stack 5
5. Multiply the two topmost elements, pop them off stack and the stack the result 20.

The value on top of the stack at the end is the value of the entire expression.

Syntax Directed Translation for Postfix:

E.Code is a string-valued translation. The value of the translation **E.code** for the first production is the concatenation of **E$^{(1)}$.code** , **E$^{(2)}$.code** and the symbol **op**

Production	Semantic Rule	Program Fragment
E → E1 op E2	E.code = E1.code ||E2.code||op	Print op
E → (E1)	E.code = E1.code	
E → id	E.code = id	Print id

Fig 3.3 SDT for postfix

Processing the input a+b*c ,a syntax-directed translator based on an LR parser, has the following sequence of moves and generates the postfix abc*+.

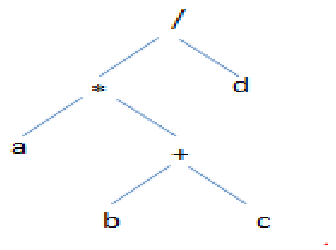
1. Shift a
2. Reduce by E→ id and print a
3. Shift +
4. Shift b
5. Reduce by E→ id and print b
6. Shift *
7. Shift c
8. Reduce by E→ id and print c
9. Reduce by E→ E op E and print *
10. Reduce by E→ E op E and print +

Syntax tree:

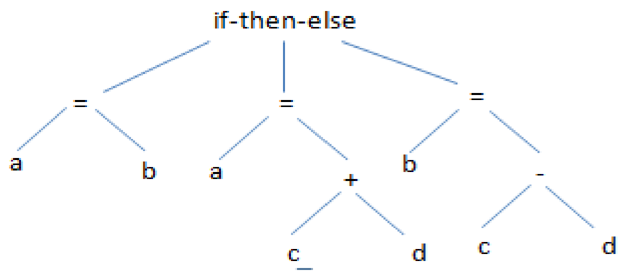
Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example:

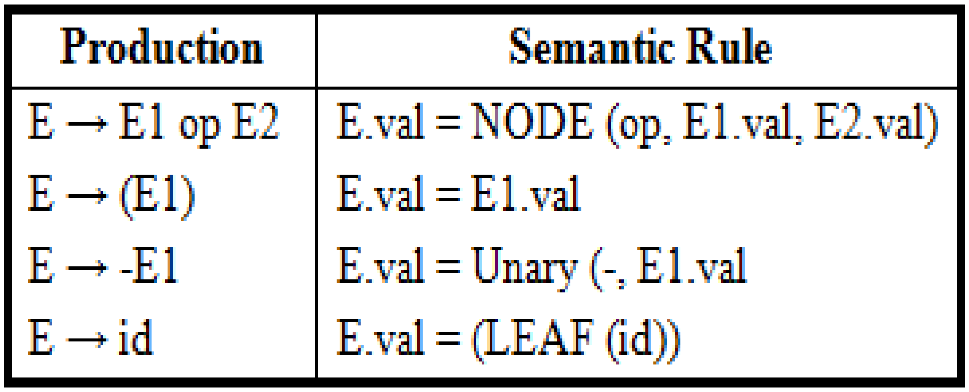
The syntax tree for the expression a*(b+c)/d is as shown in the figure.



The syntax tree for the statement if a=b then a=c+d else b=c-d



Syntax Directed Translation for syntax tree:



Production	Semantic Rule
E → E1 op E2	E.val = NODE (op, E1.val, E2.val)
E → (E1)	E.val = E1.val
E → -E1	E.val = Unary (-, E1.val
E → id	E.val = (LEAF (id))

Fig 3.4 SDT for syntax tree

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.

Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement.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:

Three-address Statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of three-address statement in the array holding intermediate code.

Here are the common three address statements used :

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 fixed-point number to a floating-pointnumber.
3. Copy statement of the form x=y where the value of y is assigned tox.
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 is the usual sequence.

6. Param x and call p, n for procedure calls and return y. where y representing a returned value is optional Their typical use it as the sequence of three.address statements.

```
    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. The first of these sets x to the value in the location i memory units beyond location y. The stat[i]=y sets the contents of the location I units beyond x to the value of y. In both these instructions, x, y. and i refer to data objects.

8. Address and pointer assignments of the form x=&y, x=*y and*x=y

Implementation of Three Address Statements:

A three-address statement is an abstract form of intermediate code.In a compiler, these statements can be implemented as records with fields for the operator and the operands.Three such representations are quadruples, triples, and indirect triples.

Quadruples:

A quadruple is a record structure with four fields, which we call op,. arg1, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x =y op z is represented by placing y in arg1, z in arg2, and x in result. Statements with unary operators like x = -y or x= y do not use arg2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result.

Example : Consider expression a = b * – c + b * – c.

The three address code is:

t1 = uminus c

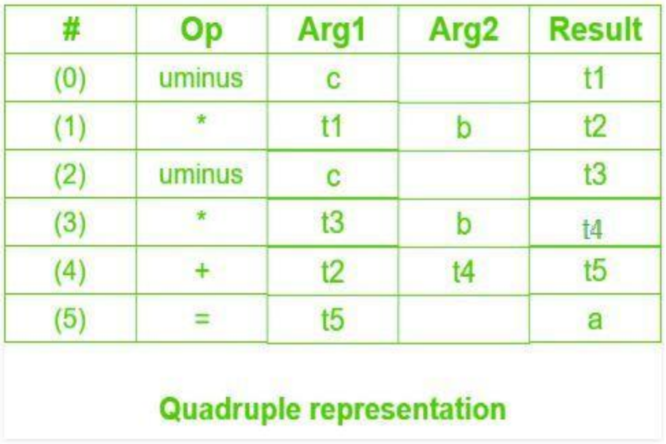
t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5

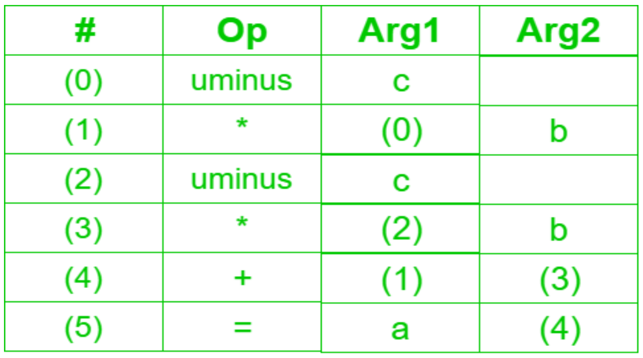


#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples:

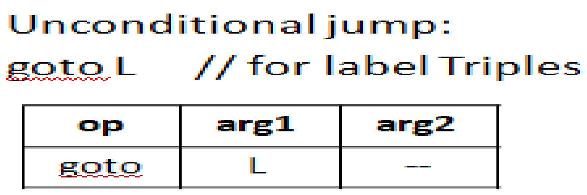
To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. Doing so ,the three address statements can be represented by records with only three fields :op,arg1,arg2.



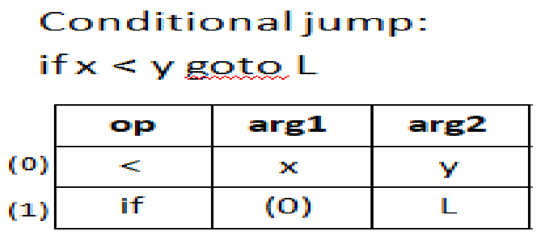
#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

The contents of fields arg1,arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created. Triples for conditional and unconditional statements are shown in Fig 3.5.


Unconditional jump:
goto L   // for label Triples

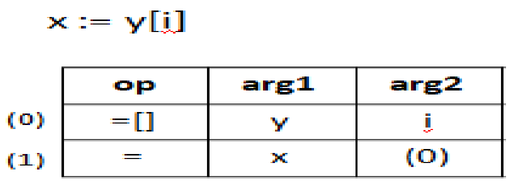
op	arg1	arg2
goto	L	--


Conditional jump:
if x < y goto L

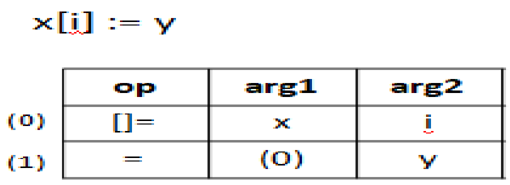
	op	arg1	arg2
(0)	<	x	y
(1)	if	(0)	L

Fig 3.5 Triples

Triples for indexed assignment statements are shown in Fig 3.6.


x := y[i]

	op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

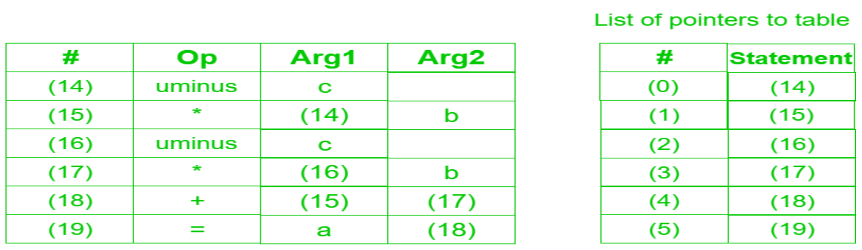

x[i] := y

	op	arg1	arg2
(0)	[]=	x	i
(1)	=	(0)	y

Fig 3.6 Triples for indexed assignment statements

Indirect Triples:

Another implementation of three address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples. A triple and their indirect triple representation ins shown below.


#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Syntax Directed Translation for Assignment Statements :

In the syntax directed translation, assignment statement is mainly deals with expressions.

The expression can be of type real, integer, array…

Consider the grammar:

S $\rightarrow$ id := E

E $\rightarrow$ $E_1 + E_2$

E $\rightarrow$ $E_1 * E_2$

E $\rightarrow$ (E_1)

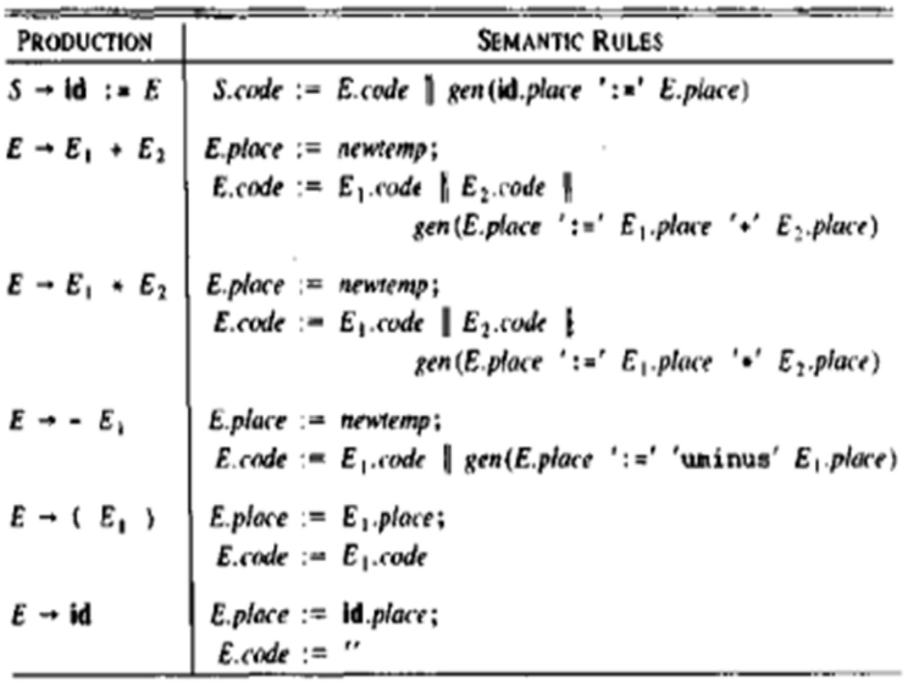
E $\rightarrow$ id

The m=non-terminal E has two attributes:

- E.place, a name that will hold the value of E, and

- E.code, the sequence of three-address statements evaluating E.

A function gen(…) to produce sequence of three address statements .The statements themselves are kept in some data structure, e.g. list – SDD operations described using pseudo code. The syntax directed translation for assignment statement is given as below;



PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** := E	*S.code* := *E.code* ∥ *gen*(**id**.*place* ':=' *E.place*)
$E \rightarrow E_1 + E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ∥ E_2.*code* ∥ *gen*(*E.place* ':=' E_1.*place* '+' E_2.*place*)
$E \rightarrow E_1 * E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ∥ E_2.*code* ∥ *gen*(*E.place* ':=' E_1.*place* '*' E_2.*place*)
$E \rightarrow - E_1$	*E.place* := *newtemp*; *E.code* := E_1.*code* ∥ *gen*(*E.place* ':=' 'uminus' E_1.*place*)
$E \rightarrow (E_1)$	*E.place* := E_1.*place*; *E.code* := E_1.*code*
$E \rightarrow$ **id**	*E.place* := **id**.*place*; *E.code* := ''

The sequence of moves in generating the three address code for an assignment statement is given by the following example.

Example: Generate the three address code for the assignment statement **A= − B*(C+D)**

Input	Stack	PLACE	Generated Code
A= – B*(C+D)			
= – B*(C+D)	id	A	
– B*(C+D)	id=	A –	
B*(C+D)	id = –	A– –	
*(C+D)	id = – id	A– – B	
*(C+D)	id = – E	A– – B	T_1 = - B
*(C+D)	id = E	A– T_1	
(C+D)	id= E *	A– T_1 –	
C+D)	id= E * (	A– T_1 – –	
+D)	id= E * (id	A– T_1 – – C	
+D)	id= E * (E	A– –T_1 – – C	
D)	id= E * (E +	A– T_1 – – C –	
)	id= E * (E + id	A– T_1 – – C – D	
)	id= E * (E + E	A– T_1 – – C – D	T_2 = C + D
)	id= E * (E	A– T_1 – – T_2	
	id= E * (E)	A– T_1 – – T_2 –	
	id= E * E	A– T_1 – T_2	T_3 = T_1 * T_2
	id= E	A– T_3	A = T_3
	S	S	

Assignment Statement with mixed types:

The constants and variables would be of different types, hence the compiler must either reject certain mixed-mode operations or generate appropriate coercion (mode conversion) instructions.

Consider two modes:

REAL

INTEGER, with integer converted to real when necessary.

An additional field in translation for E is E.MODE whose value is either REAL or INTEGER.

The semantic rule for E.MODE associated with the production **E $\rightarrow$ E₁ + E₂** is:

E $\rightarrow$ E_1 + E_2 {if E_1.MODE = INTEGER and E_2.MODE = INTEGER then E.MODE = INTEGER else E.MODE = REAL }

When necessary the three address code

A=inttoreal B is generated,

whose effect is to convert integer B to a real of equal value called A.

Example: Consider the assignment statement

X=Y+I*J

Assume X and Y to be REAL and I and J have mode INTEGER.

Three-addresss code:

T_1=I int * J

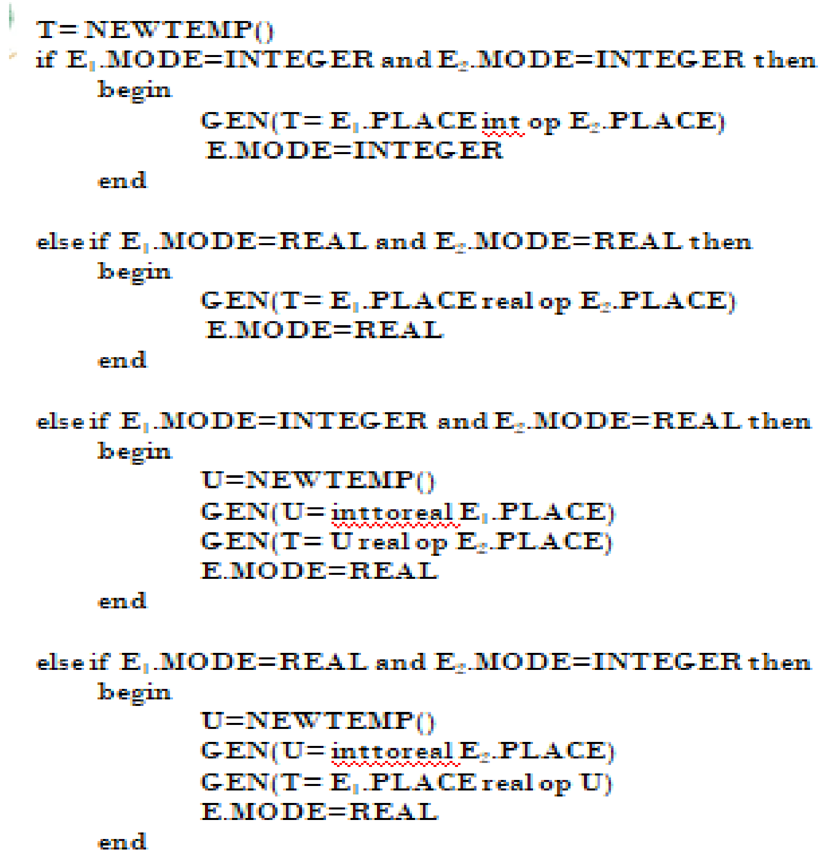
T_2= inttoreal T_1

T_3=Y real +T_2

X=T_3

The semantic rule uses two translation fields E.PLACE and E.MODE for the non-terminal E

The semantic rule for E → E_1 op E_2 is given as below:



```
T= NEWTEMP()
if $E_1$.MODE=INTEGER and $E_2$.MODE=INTEGER then
    begin
            GEN(T= $E_1$.PLACE int op $E_2$.PLACE)
            E.MODE=INTEGER
    end

else if $E_1$.MODE=REAL and $E_2$.MODE=REAL then
    begin
            GEN(T= $E_1$.PLACE real op $E_2$.PLACE)
            E.MODE=REAL
    end

else if $E_1$.MODE=INTEGER and $E_2$.MODE=REAL then
    begin
            U=NEWTEMP()
            GEN(U= inttoreal $E_1$.PLACE)
            GEN(T= U real op $E_2$.PLACE)
            E.MODE=REAL
    end

else if $E_1$.MODE=REAL and $E_2$.MODE=INTEGER then
    begin
            U=NEWTEMP()
            GEN(U= inttoreal $E_2$.PLACE)
            GEN(T= $E_1$.PLACE real op U)
            E.MODE=REAL
    end
```

SDT for Boolean Expression:

Boolean expressions have two primary purposes.

1. They are used for computing the **logical values**.
2. They are also used as **conditional expression** that alter the flow of control, such as if-then-else or while-do.

Boolean expression are composed of the boolean operators(**and** ,**or**, and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 relop E_2 , where E_1 and E_2 are arithmetic expressions.

Consider the grammar

E $\rightarrow$ E **or** E

E $\rightarrow$ E **and** E

E $\rightarrow$ **not** E

E $\rightarrow$ (E)

E $\rightarrow$ id **relop** id

E $\rightarrow$ TRUE E $\rightarrow$ id

E $\rightarrow$ FALSE

The relop is any of $<, \leq, =, \neq, >$, or $\geq$.. We assume that **or** and **and** are left associative. **or** has lowest precedence ,then **and** ,then **not**.

Methods of Translating Boolean Expressions:

There are two methods of representing the value of a Boolean.

1.To encode true or false numerically

- 1 is used to denote true.
- 0 is used to denote false.

2.To evaluate a boolean expression analogously to an arithmetic expression.

- Flow of control –representing the value of a boolean expression by a position reached in a program.
- Used in flow of control statements such as if-then-else or while-do statements.

Numerical Representation of Boolean Expressions:

First consider the implementation of boolean expressions using 1 to denote TRUE and 0 to denote FALSE. Expressions will be evaluated from left to right.

Example 1:

A or B and C

Three address sequence:

T_1 = B and C

T_2 = A or T_1

Example 2:

A relational expression A< B is equivalent to the conditional statement.

if A < B then 1 else 0

Three address sequence:

(1) if A < B goto (4)

(2) T=0

(3) goto (5)

(4) T=1

(5) ---

Semantic Rules for Boolean Expression:

The Semantic rules for Boolean expression are given as below:

$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen (E.place = E_1.place **or** E_2.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); gen (E.place = E_1.place **and** E_2.place)}
$E \rightarrow$ NOT E_1	{E.place = newtemp(); gen (E.place = **not** E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{ E.place = newtemp(); gen (if id_1.place **relop.op** id_2.place goto nextquad + 3); gen(E.place =0) gen(goto nextquad + 2) gen(E.place=1) }
$E \rightarrow$ TRUE	{E.place := newtemp(); gen(E.place=1)}
$E \rightarrow$ FALSE	{E.place := newtemp(); gen(E.place =0)}

Example: Code for a < b or c < d and e <f

100: if a < b goto 103

101: t1 = 0

102: goto 104

```
103: t1 = 1
104: if c < d goto 107
105: t2 =0
106: goto 108
107: t2 = 1
108: if e < f goto 111
109: t3 =0
110: goto 112
111: t3 = 1
112: t4 = t2 and t3
113: t5 = t1 or t4
```

Control Flow Representation:

Short Circuit Evaluation of Boolean Expressions:

Flow of control statements with the jump method.

Consider the following : (short circuit code)

S → if E then S | if E then S1 else S2

S → while E do S1

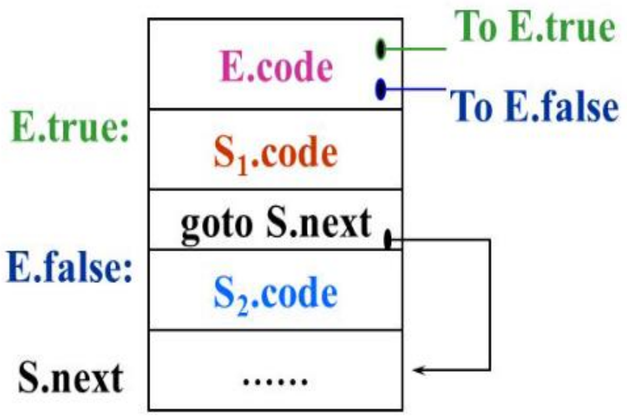


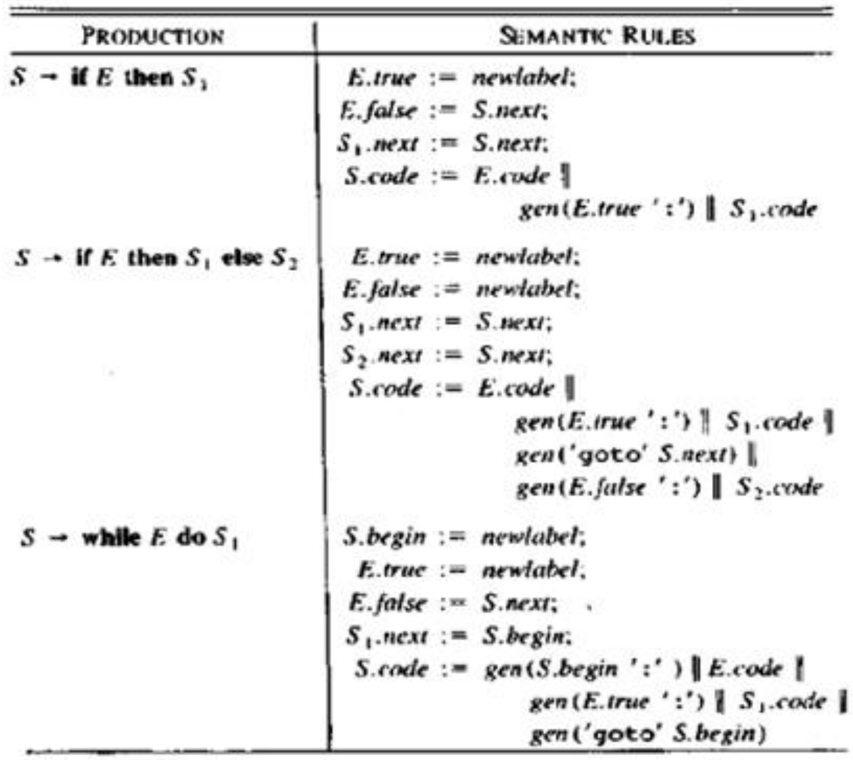
Fig 3.7 Attributes used for if E then S_1 else S_2

E is associated with two attributes:

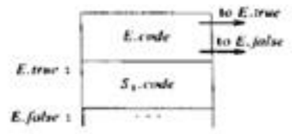
1. E.true: label which controls if E is true
2. E.false :label which controls if E is false

S.next - is a label that is attached to the first three address instruction to be executed after the code for S (S_1 or S_2)

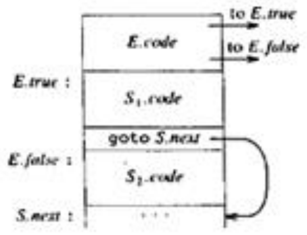
The Semantic rule for flow of control statements is given by:



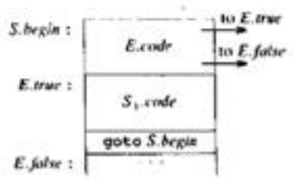
PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true$:= $newlabel$; $E.false$:= $S.next$; $S_1.next$:= $S.next$; $S.code$:= $E.code$ ‖ $gen(E.true$ ' : ') ‖ $S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true$:= $newlabel$; $E.false$:= $newlabel$; $S_1.next$:= $S.next$;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



(a) if-then



(b) if-then-else



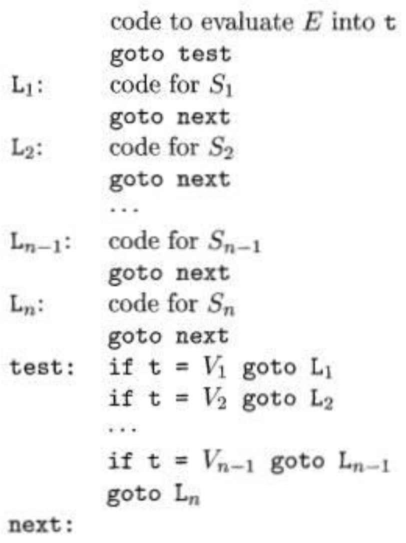
(c) while-do

SDT for Switch Case statement:

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows: Syntax for switch case statement

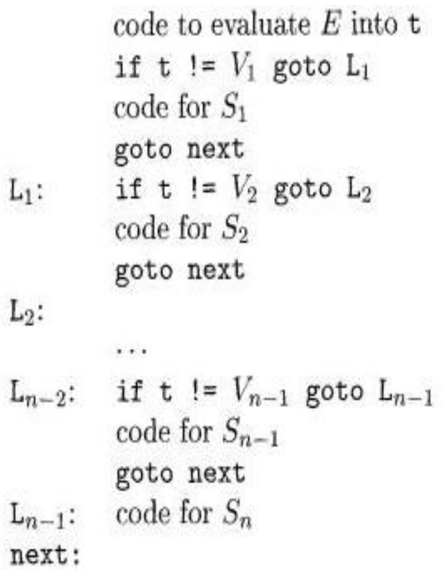
```
switch E
begin
case V1: S1
case V2: S2
…
case Vn-1: Sn-1
default: Sn
end
```

When switch keyword is seen then a new temporary t and two new labels test and next are generated. When the case keyword occurs then for each case keyword, a new label L_i is created and entered into the symbol table. The value of V_i of each case constant and a pointer to this symbol-table entry are placed on a stack.



```
           code to evaluate $E$ into t
           goto test
$L_1$:     code for $S_1$
           goto next
$L_2$:     code for $S_2$
           goto next
           ...
$L_{n-1}$: code for $S_{n-1}$
           goto next
$L_n$:     code for $S_n$
           goto next
test:      if t = $V_1$ goto $L_1$
           if t = $V_2$ goto $L_2$
           ...
           if t = $V_{n-1}$ goto $L_{n-1}$
           goto $L_n$
next:
```

Translation of a switch-statement



```
           code to evaluate $E$ into t
           if t != $V_1$ goto $L_1$
           code for $S_1$
           goto next
$L_1$:     if t != $V_2$ goto $L_2$
           code for $S_2$
           goto next
$L_2$:
           ...
$L_{n-2}$: if t != $V_{n-1}$ goto $L_{n-1}$
           code for $S_{n-1}$
           goto next
$L_{n-1}$: code for $S_n$
next:
```

Another translation of a switch statement

SDT for Procedure Call:

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record
- Evaluate the argument of the called procedure.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

Consider the grammar for a simple procedure call statement,

(1) S→ call id(Elist)

(2) Elist → Elist,E

(3) Elist → E

Syntax-Directed Translation for Procedure call is given as:

Production Rule	Semantic Action
S → call id(Elist)	{for each item p on QUEUE do GEN (param p) GEN (call id.PLACE) }
Elist → Elist, E	{append E.PLACE to the end of QUEUE }
Elist → E	{initialize QUEUE to contain only E.PLACE}

QUEUE is used to store the list of parameters in the procedure call.

Symbol Table Generation:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces,etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. It is built in lexical and syntax analysis phases.

The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code. It is used by compiler to achieve compile time efficiency. It is used by various phases of compiler as follows:-

Lexical Analysis: Creates new table entries in the table, example like entries about token.

Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and updates it accordingly.

Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

Code Optimization: Uses information present in symbol table for machine dependent optimization.

Target Code generation: Generates code by using address information of identifier present in the table.

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<Symbol Name, Type, Attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

Then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table includes:

Operations	Functions
Allocate	To allocate a new empty symbol table
Free	To remove all entries and free the storage of symbol table
Lookup	To search for a name and return pointer to its entry
Insert	To insert name in a symbol table and return a pointer to its entry
Set_Attribute	To associate an attribute with a given entry
Get_Attribute	To get an attribute associated with a given entry

Implementation of Symbol table – Following are commonly used data structure for implementing symbol table:-

List:

- ❖ In this method, an array is used to store names and associated information.
- ❖ A pointer “available” is maintained at end of all stored records and new names are added in the order as they arrive
- ❖ To search for a name we start from beginning of list till available pointer and if not found we get an error “use of undeclared name”
- ❖ While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. “Multiple defined name”
- ❖ Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- ❖ Advantage is that it takes minimum amount of space.

Linked List:

- ❖ This implementation is using linked list. A link field is added to each record.
- ❖ Searching of names is done in order pointed by link of link field.
- ❖ A pointer “First” is maintained to point to first record of symbol table.
- ❖ Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Hash Table:

- ❖ In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
- ❖ A hash table is an array with index range: 0 to table size – 1.These entries are pointer pointing to names of symbol table.
- ❖ To search for a name we use hash function that will result in any integer between 0 to table size –1.
- ❖ Insertion and lookup can be made very fast –O(1).
- ❖ Advantage is that search is possible and disadvantage is that hashing is complicated to implement.

Binary Search Tree:

- ❖ Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- ❖ All names are created as child of root node that always follows the property of

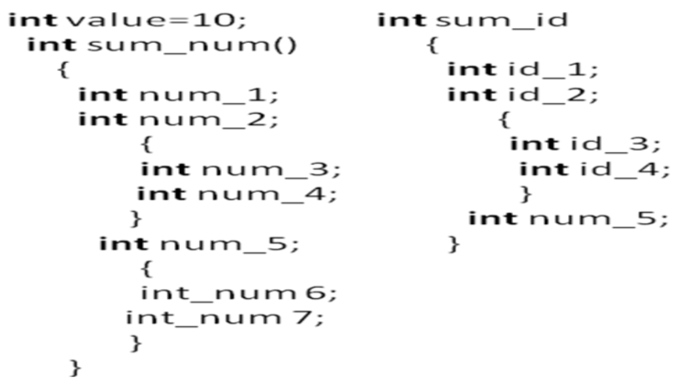
binary search tree.

- ❖ Insertion and lookup are O(log2 n) on average.

Scope Management:

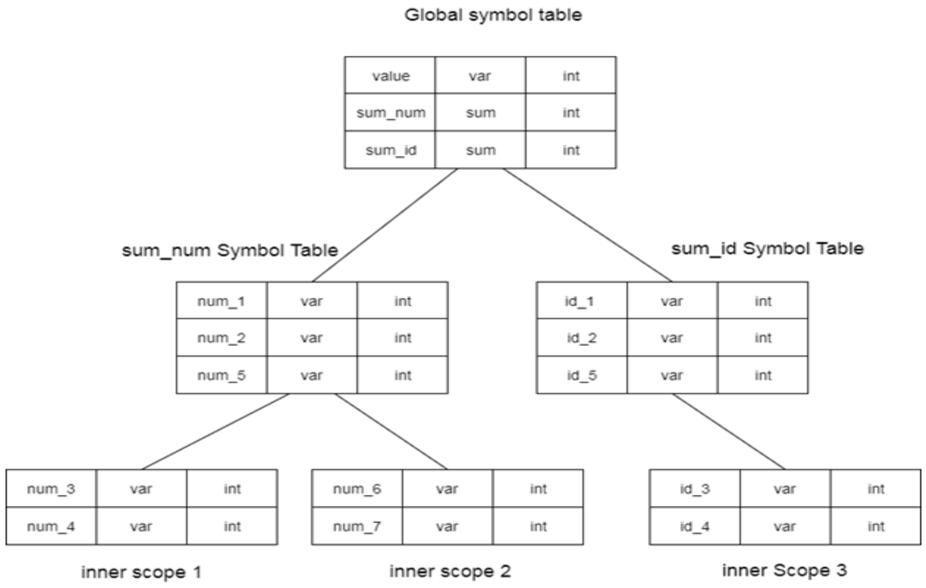
❖ A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

❖ To determine the scope of a name, symbol tables are arranged in hierarchical structure consider the example as shown below:



```
int value=10;                 int sum_id
 int sum_num()                    {
   {                                int id_1;
     int num_1;                     int id_2;
     int num_2;                        {
          {                              int id_3;
          int num_3;                      int id_4;
          int num_4;                      }
         }                            int num_5;
       int num_5;                   }
          {
          int_num 6;
        int_num 7;
         }
    }
```

The above program can be represented in a hierarchical structure of symbol tables:



Global symbol table

value	var	int
sum_num	sum	int
sum_id	sum	int

sum_num Symbol Table

num_1	var	int
num_2	var	int
num_5	var	int

sum_id Symbol Table

id_1	var	int
id_2	var	int
id_5	var	int

inner scope 1

num_3	var	int
num_4	var	int

inner scope 2

num_6	var	int
num_7	var	int

inner Scope 3

id_3	var	int
id_4	var	int

The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.