UNIT – 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.

3.1 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.
- Representation is Syntax Directed Translation
- 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:
    - **Synthesized Attributes**. They are computed from the values of the attributes of the children nodes.

- **Inherited Attributes**. They are computed from the values of the attributes of both the siblings and the parent nodes

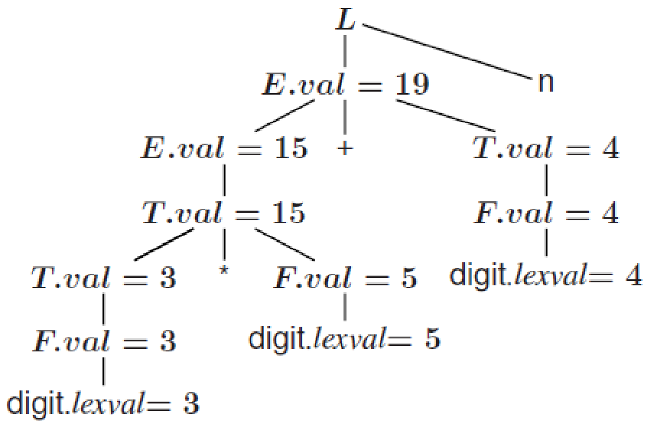
(Eg) Let us consider the Grammar for arithmetic expressions **(DESK CALCULATOR)**

- The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

PRODUCTION	SEMANTIC RULE
L → E	print (E.val)
E → E + T	E.val := E 1.val + T .val
E → T	E.val := T.val
T → T1 ∗ F	T .val := T1.val ∗ F.val
T → F	T .val := F.val
F → (E)	F.val := E.val
F → 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 PostOrder, traversal of the parse-tree.
- The annotated parse-tree for the input 3*5+4n is:



3.2 Intermediate Code Representation:

<u>Types</u>

1.Postfix

2.Syntax Tree

3.**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-point number.

3. Copy statement 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 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.

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 representedby records with only three fields :op,arg1,arg2. The quadruples and triple representation for the assignment a=b+-c+b+-c is shown below:

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.

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.

(Eg) Implement the Quadruple, Triple and Indirect Triple for the expression **E:=(a*b)+c**

Soln

Three address statements

T1=a*b

T2=T1+c

E:=T2

Quadruple

	OP	**Arg 1**	**Arg 2**	**Result**
(0)	*	a	b	T1
(1)	+	T1	c	T2
(2)	:=	T2		E

Triple

	OP	**Arg 1**	**Arg 2**
(0)	*	a	b
(1)	+	(0)	c
(2)	:=	E	(1)

Indirect Triple

(0)	(100)
(1)	(101)
(2)	(102)

	OP	**Arg 1**	**Arg 2**
(100)	*	a	b
(101)	+	(100)	c
(102)	:=	E	(101)

3.3 Syntax Directed Translation for Assignment Statements

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

S → id := E

{S.code := E.code || gen(id.place:= E.place) }

E → E1 + E2

{E.place:= newtmp E.code:= E1 .code || E2 .code || gen(E.place := E1 .place + E2 .place)}

E → E1 * E2

{E.place:= newtmp E.code := E1 .code || E2 .code || gen(E.place := E1 .place * E2 .place)}

E → -E1

{ E.place := newtmp E.code := E1 .code || gen(E.place := - E1 .place)}

E → (E1)

{E.place := E1 .place E.code := E1 .code}

E → id

{E.place := id.place E.code := ‘ ‘}

3.4 SDT for Boolean Expression

- Boolean Expressions are used for
  - Compute logical values
  - Change the flow of control
- Boolean operators are: and, or, not
- E → E or E | E and E | not E | (E) | id relop id | true | false
- **Methods of Translation**
  - Evaluate similar to arithmetic expressions, i.e., normally use 1 for true and 0 for false
  - Implement by flow of control, for example given expression E1 or E2 if E1 evaluates to true then E1 or E2 evaluates to true without evaluating E2
- **Numerical Representation of Boolean Expressions**
  - a or b and not c

    t1 = not c

    t2 = b and t1

    t3 = a or t2
  - relational expression a < b is equivalent to if a < b then 1 else 0

    1. if a < b goto 4.
    2. t = 0
    3. goto 5
    4. t = 1
    5.
- **Syntax Directed Translation**

E → E1 or E2	E.place := newtemp emit (E.place ':=' E1 .place 'or' E2 .place)
E → E1 and E2	E.place:= newtemp emit (E.place ':=' E1 .place 'and' E2 .place)

E → not E1	E.place := newtmp emit (E.place ':=' 'not' E1 .place)
E → (E1)	E.place = E1 .place
E → id1 relop id2	E.place := newtmp
	emit (if id1.place relop id2.place goto nextstat+3)
	emit (E.place = 0)
	emit (goto nextstat+2)
	emit (E.place = 1)
E → true	E.place := newtmp emit (E.place = '1')
E → false	E.place := newtmp emit (E.place = '0')

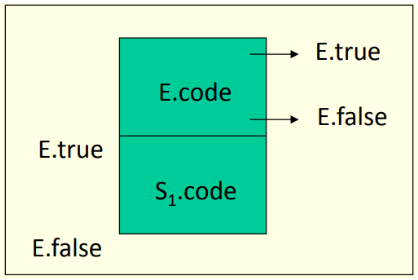
- **Example: Code for a < b or c < d and e < f**

```
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
```

- **Short Circuit Evaluation of Boolean Expressions**
  - Translate Boolean Expressions without:
    - Generating code for Boolean Operators
    - Evaluating the entire expression
  - Flow of control statements
    - S → if E then S1 | if E then S1 else S2 | while E do S1
- **Syntax Directed Translation**

S → if E then S1	E.true = newlabel
	E.false = S.next
	S1.next = S.next

S.code = E.code || gen(E.true':') || S1.code



S → if E then S1 else S2 E.true = newlabel

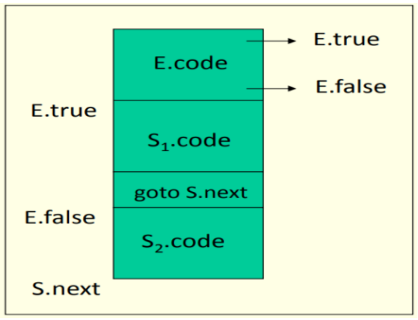
E.false = newlabel

S1.next = S.next

S2.next = S.next

S.code = E.code || gen (E.true':') || S1.code ||

gen (goto S.next) || gen (E.false':') || S2.code



S → while E do S1 S.begin = newlabel

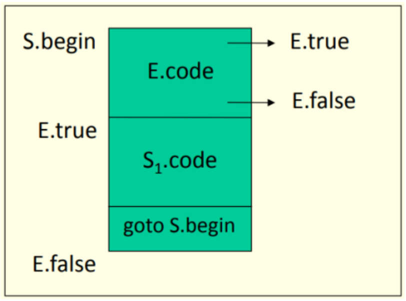
E.true = newlabel

E.false = S.next

S1.next = S.begin

S.code = gen(S.begin':') || E.code || gen(E.true':') || S1.code

|| gen(goto S.begin)



3.5 SDT for Switch Case

- **Syntax for switch case statement**

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

- **Syntax Directed Translation**

```
code to Evaluate E into t
goto Ltest
L1:     code for S1
        goto Lnext
L2:     code for S2
        goto Lnext
…
Ln-1:   code for Sn-1
        goto Lnext
Ln:     code for Sn
        goto Lnext
Ltest:  if t = V1 goto L1
        if t = V2 goto L2
        …
        if t = Vn-1 goto Ln-1
        goto Ln
Lnext:
```

3.6 SDT for Procedure Call

- For a function fun with n arguments a1,a2,a3….an i.e., fun(a1, a2, a3,…an)
- **Syntax Directed Translation**

```
        codeGen_expr (E):
        codeGen_expr_list (arguments);
```

```
E.place = newtemp (f.returnType);
E.code = code to evaluate the arguments
            Param a1
            Param a2
            …
            Param an
            Call fun, n …
            retrieve E.place;
```

Where param defines the arguments to function.

- **Example:**
- Consider the statement n=f(a[i]) where a is array of integers f is function from integers to integers
- The three address code for the procedure call will be as follows

```
t1 = i * 4
t2 = a [t1]
param t2
t3 = call f, 1
n = t3
```

3.7 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 :-**

1. **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.
2. **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
3. **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.

4. **Binary Search Tree:**

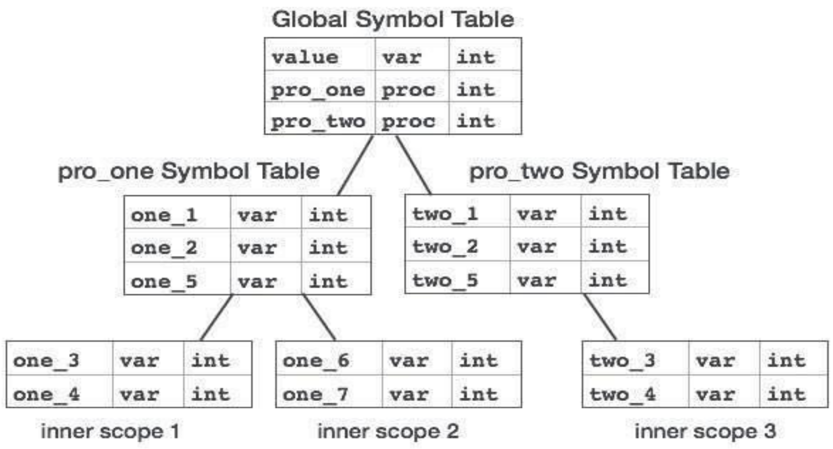
- Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follows the property of binary search tree.
- Insertion and lookup are O(log2 n) on average.

- **Scope Management**
  - A compiler maintains two types of symbol tables: a global symbol tablewhich 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 as shown in the example below:

```
. . .
int value=10;
void pro_one()
  {
  int one_1;
  int one_2;
    {            \
     int one_3;      |_  inner scope 1
     int one_4;      |
     }            /
   int one_5;
    {            \
     int one_6;      |_  inner scope 2
     int one_7;      |
     }            /
  }
 void pro_two()
  {
  int two_1;
  int two_2;
    {            \
     int two_3;      |_  inner scope 3
     int two_4;      |
     }            /
  int two_5;
```

```
    }
  . . .
```

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



- The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.
- This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:
  - First a symbol will be searched in the current scope, i.e. current symbol table.
  - if a name is found, then search is completed, else it will be searched in the parent symbol table until,
  - Either the name is found or global symbol table has been searched for the name.