7 SYMBOL TABLE MANAGEMENT

7.1 THE SYMBOL TABLE

A symbol table is a data structure used by a compiler to keep track of scope/binding information about names. These names are used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements. The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.

For efficiency, our choice of the implementation data structure for the symbol table and the organization of its contents should stress on minimal cost when adding new entries or accessing the information of existing entries. Also, if the symbol table can grow dynamically as necessary, then it is more useful for a compiler.

7.2 IMPLEMENTATION

Each entry in a symbol table can be implemented as a record that consists of several fields. These fields are dependent on the information to be saved about the name. But since the information about a name depends on the usage of the name (i.e., on the program element identified by the name), the entries in the symbol table records will not be uniform. Hence, to keep the symbol table records uniform, some of the information about the name is kept outside of the symbol table record, and a pointer to this information is stored in the symbol table record, as shown in Figure 7.1. Here, the information about the lower and upper bounds of the dimension of the array named *a* is kept outside of the symbol table record, and the pointer to this information is stored within the symbol table record.

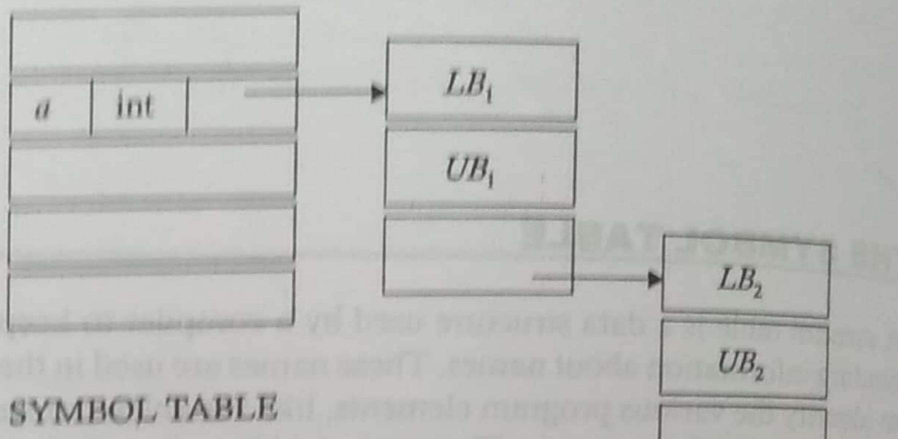

FIGURE 7.1 A pointer steers the symbol table to remotely stored information for the array a.

7.3 ENTERING INFORMATION INTO THE SYMBOL TABLE

Information is entered into the symbol table in various ways. In some cases, the symbol table record is created by the lexical analyzer as soon as the name is encountered in the input, and the attributes of the name are entered when the declarations are processed. But very often, the same name is used to denote different objects, perhaps even in the same block. For example, in C programming, the same name can be used as a variable name and as a member name of a structure, both in the same block. In such cases, the lexical analyzer only returns the name to the parser, rather than a pointer to the symbol table record. That is, a symbol table record is not created by the lexical analyzer, the string itself is returned to the parser, and the symbol table record is created when the name's syntactic role is discovered.

7.4 WHERE SHOULD NAMES BE HELD?

If there is a modest upper bound on the length of the name, then the name can be stored in the symbol table record itself. But if there is no such limit, or if the limit is rarely reached, then an indirect scheme of storing name is used. A separate array of characters, called a "string table," is used to store the name, and a pointer to the name is kept in the symbol table record, as shown in Figure 7.2.

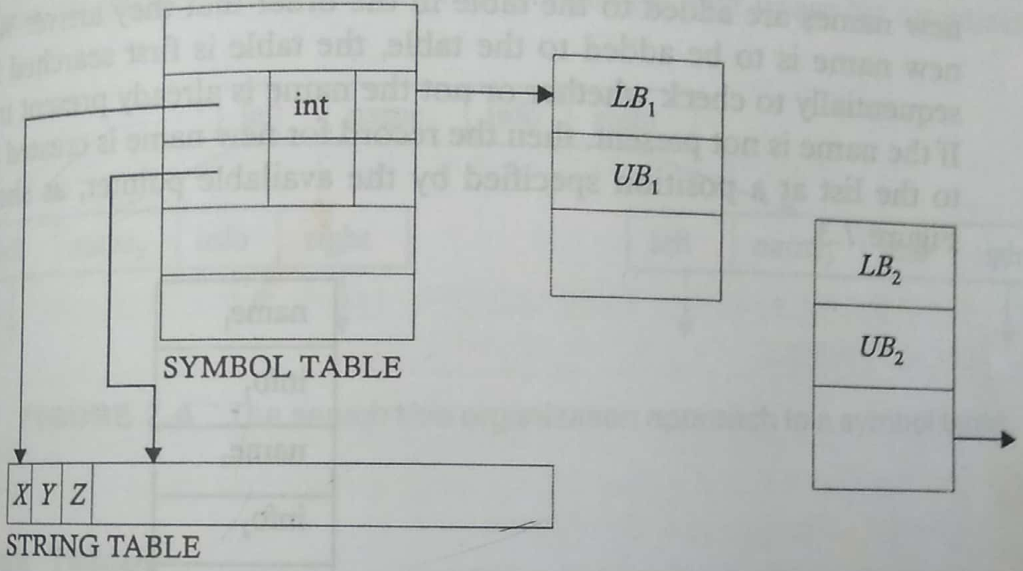

FIGURE 7.2 Symbol table names are held either in the symbol table record or in a separate string table.

7.5 INFORMATION ABOUT THE RUNTIME STORAGE LOCATION

The information about the runtime, name storage location is kept in the symbol table. If the compiler is going to be generating assembly code, then the assembler takes care of the storage locations of the various names. After generating the assembly code, the compiler scans the symbol table and generates the assembly language data definitions. These are appended to the assembly language code for each name. But if machine code is being generated, then the compiler must ascertain the position of each data object relative to a fixed origin.

7.6 VARIOUS APPROACHES TO SYMBOL TABLE ORGANIZATION

There are several methods of organizing the symbol table. These methods are discussed below.

7.6.1 The Linear List

A linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the name is already present in the table. If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer, as shown in the Figure 7.3.

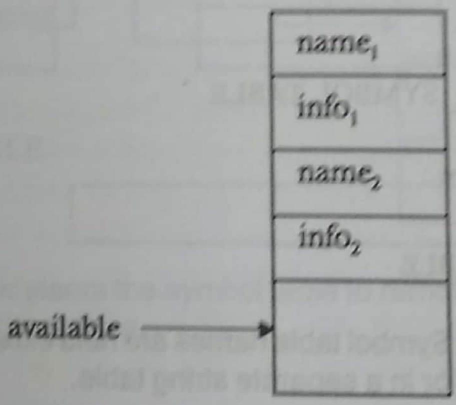

FIGURE 7.3 A new record is added to the linear list of records.

To retrieve the information about the name, the table is searched sequentially, starting from the first record in the table. The average number of comparisons, p, required for search are $p = (n + 1)/2$ for successful search and $p = n$ for an unsuccessful search, where n is the number of records in symbol table. The advantage of this organization is that it takes less space, and additions to the table are simple. This method's disadvantage is that it has a higher accessing time.

7.6.2 Search Trees

A search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a name is to be added, first the name is searched in the tree. If it does not exist, then a record for the new name is created and added at the proper position in the search tree. This organization has the property of alphabetical accessibility; that is, all the names accessible from $name_i$ by following a left link, precede $name_1$ in alphabetical order. Similarly, all the name accessible from $name_i$ by following right link follow $name_i$ in alphabetical order (see Figure 7.4). The expected time needed to enter n names and to make m queries is proportional to $(m + n) \log_2 n$; so for greater numbers of records (higher n) this method has advantages over linear list organization.

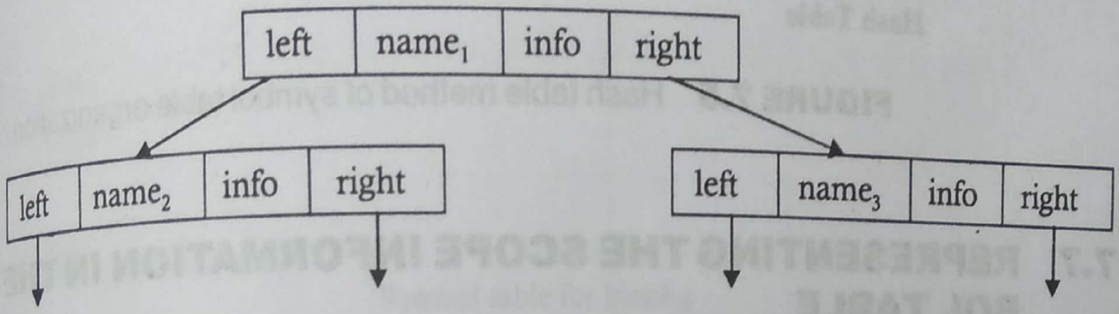

FIGURE 7.4 The search tree organization approach to a symbol table.

.3 Hash Tables

A hash table is a table of k pointers numbered from zero to k–1 that point to the symbol table and a record within the symbol table. To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function. The hash function maps the name into an integer between zero and k–1, and using this value as an index in the hash table, we search the list of the symbol table records that are built on that hash index. If the name is not present in that list, we create a record for name and insert it at the head of the list. When retrieving the information associated with the name the hash value of the name is first obtained, and then the list that was built on this hash value is searched for information about the name (Figure 7.5).

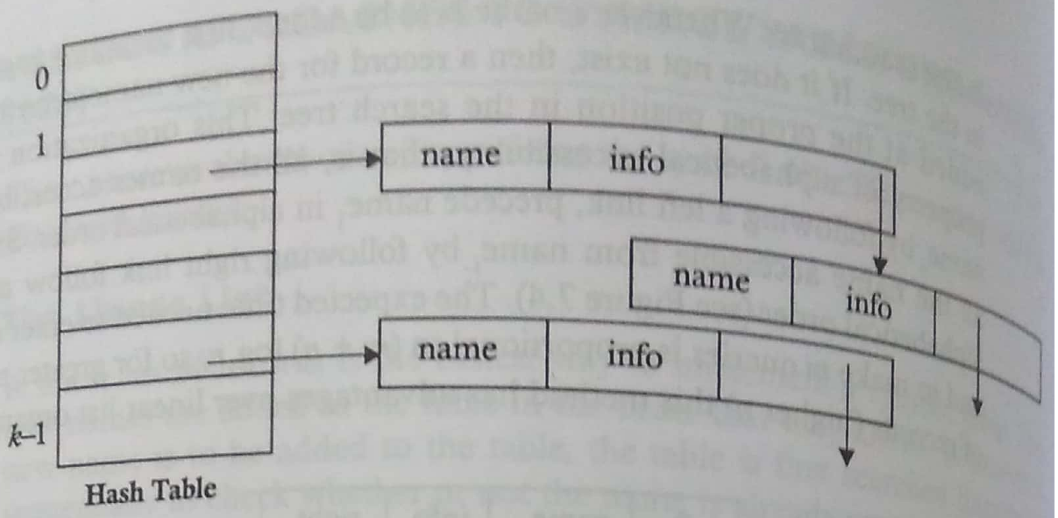

FIGURE 7.5 Hash table method of symbol table organization.

7.7 REPRESENTING THE SCOPE INFORMATION IN THE SYMBOL TABLE

Every name possesses a region of validity within the source program, called the "scope" of that name. The rules governing the scope of names in a block-structured language are as follows:

1.. A name declared within a block *B* is valid only within *B*.
2. If block *B*1 is nested within *B*2, then any name that is valid for *B*2 is also valid for *B*1, unless the identifier for that name is re-declared in *B*1.

These scope rules require a more complicated symbol table organization than simply a list of associations between names and attributes. One technique that can be used is to keep multiple symbol tables, one for each active block, such as the block that the compiler is currently in. Each table is list of names and their associated attributes, and the tables are organized into a stack. Whenever a new block is entered, a new empty table is pushed onto the stack for holding the names that are declared as local to this block. And when a declaration is compiled, the table on the stack is searched for a name. If the name is not found, then the new name is inserted. When a reference to a name is translated, each table is searched, starting from the top table on the stack, ensuring compliance with static scope rules. For example, consider following program structure. The symbol table organization will be as shown in Figure 7.6.

Program main
Var x,y : integer :

B2 –
{B1
}
}

```
Procedure P :
Var x,a : boolean;
Procedure q
Var x,y,z : real;
Begin
;
.
end
begin
:
end
begin
:
end
```

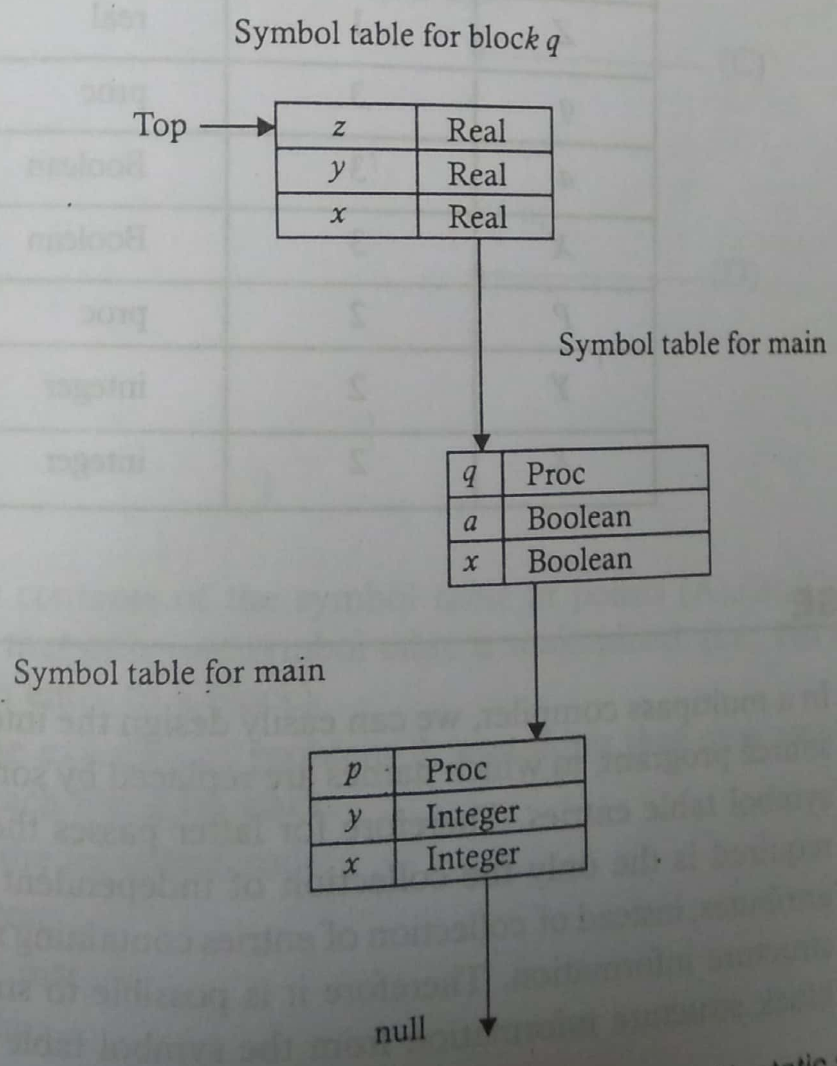

FIGURE 7.6 Symbol table organization that complies with static scope information rules.

Another technique can be used to represent scope information in the symbol table. We store the nesting depth of each procedure block in the symbol table and use the [procedure name, nesting depth] pair as the key to accessing the information from the table. A nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. This number is basically a count of how many procedures are there in the referencing environment of the procedure.

For example, refer to the program code structure above. The symbol table's contents are shown in Table 7.1.

TABLE 7.1 Symbol Table Contents Using a Nesting Depth Approach

X	1	real
Y	1	real
Z	1	real
q	3	proc
a	3	Boolean
X	3	Boolean
P	2	proc
Y	2	integer
X	2	integer

EXERCISE

1. In a multipass compiler, we can easily design the intermediate form of the source program, in which names are replaced by some form of pointers to symbol table entries. Therefore for latter passes the symbol table that is required is the only the collection of independent entries that contains attributes, instead of collection of entries containing name, attributes, block structure information. Therefore it is possible to strip off the names and block structure information from the symbol table entries. Comment

2. Consider the following C code fragment:

```
void fun()
    {
        int a,b,c;
              <-------------- (A)
        .
        .
        {
            int b,c;
                  <-------------- (B)
            .
            .
            {
                char a,d;
                      <-------------- (C)
                .
                {
                    int x;
                    <---------------(D)
                    .
                    .
                }
            }
        }
```

Show the contents of the symbol table at points (A),(B),(C), and (D), assuming that only one symbol table is maintained. (i.e., No new table is created on entry to a block).

3. Repeat the exercise of problem 2, assuming that new symbol table is built for each block on stack.

4. Consider the following code:

```
program test;
    var i,j : integer;
    procedure a;
        var i : integer;
        procedure b;
          var i : real;
                j : real;
                i := i + 10;
                a.i := i * 10;............................(A)
          end; { of procedure b}
        .
        .
      end; { procedure a}
    begin { of main}
    .
    a; { call to procedure a}
    end.
```

In the above program procedure **b** is defined inside procedure a, and therefore variable **i** which is local to procedure **a** is accessible to procedure b. But there is a local declaration of variable **i** in procedure **b** which therefore hides variable **i** local to procedure **a**, inside the body of procedure b, which again becomes available after exiting procedure **b.** If we modify the language scope rules in such a way that **i** in procedure **a** can be accessed inside the body of procedure **b** using the notation: **block-name.variable-name** as shown at point (A). How it will affect the symbol table organization. Discuss.