After syntax tree have been constructed, the compiler must check whether the input program is type-correct (called *type checking* and part of the semantic analysis). During type checking, a compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program. Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of **a *symbol table***. Note that a symbol table is a compile-time data structure. It's not used during run time by statically typed languages. Formally, a symbol table maps names into declarations (called attributes), such as mapping the variable name `x` to its type `int`. More specifically, a symbol table stores:

- For each type name, its type definition.
- For each variable name, its type. If the variable is an array, it also stores dimension information. It may also store storage class, offset in activation record etc.
- For each constant name, its type and value.
- For each function and procedure, its formal parameter list and its output type. Each formal parameter must have name, type, type of passing (by-reference or by-value), etc.

9.1 OPERATION ON SYMBOL TABLE

We need to implement the following operations for a symbol table:

1. insert (String key, Object binding)
2. object_lookup (String key)
3. begin_scope () and end_scope ()

(1) insert (s,t)- return index of new entry for string 's' and token 't'

(2) lookup (s)- return index of new entry for string 's' or o if 's' is not found.

(3) begin_scope () and end_scope () : When we have a new block (ie, when we encounter the token {), we begin a new scope. When we exit a block (i.e. when we encounter the token }) we remove the scope (this is the end_scope). When we remove a scope, we remove all declarations inside this scope. So basically, scopes behave like stacks. One way to implement these functions is to use a stack. When we begin a new scope we push a special marker to the stack (e.g., 1). When we insert a new declaration in the hash table using insert, we also push the bucket number to the stack. When we end a scope, we pop the stack until and including the first –1 marker.

Example 9.1: Consider the following program:

```
1) {
2)   int a;
3)   {
4)     int a;
5)     a = 1;
6)   };
7)   a = 2;
8) };
```

we have the following sequence of commands for each line in the source program (we assume that the hash key for a is 12):

1) push(–1)
2) insert the binding from a to int into the beginning of the list table[12]
   push(12)
3) push(–1)
4) insert the binding from a to int into the beginning of the list table[12]

Contd...

push(12)
6) pop()
remove the head of table[12]
pop()
7) pop()
remove the head of table[12]
pop()

Recall that when we search for a declaration using lookup, we search the bucket list from the beginning to the end, so that if we have multiple declarations with the same name, the declaration in the innermost scope overrides the declaration in the outer scope.

(4) Handling Reserve Keywords: Symbol table also handle reserve keywords like 'PLUS', MINUS', 'MUL' etc. This can be done in following manner.

insert ("PLUS", PLUS);

insert ("MINUS", MINUS);

In this case first 'PLUS' and 'MINUS' indicate lexeme and other one indicate token.

9.2 SYMBOL TABLE IMPLEMENTATION

The data structure for a particular implementation of a symbol table is sketched in **figure 9.1**. In **figure 9.1**, a separate array 'arr_lexemes' holds the character string forming an identifier. The string is terminated by an end-of-string character, denoted by EOS, that may not appear in identifiers. Each entry in symbol-table array 'arr_symbol_table' is a record consisting of two fields, as *"lexeme_pointer"*, pointing to the beginning of a lexeme, and token. Additional fields can hold attribute values. In **figure 9.1**, the 0th entry is left empty, because lookup return 0 to indicate that there is no entry for a string. The 1st, 2nd, 3rd, 4th, 5th, 6th, and 7th entries are for the 'a', 'plus' 'b' 'and', 'c', 'minus', and 'd' where 2nd, 4th and 6th entries are for reserve keyword.

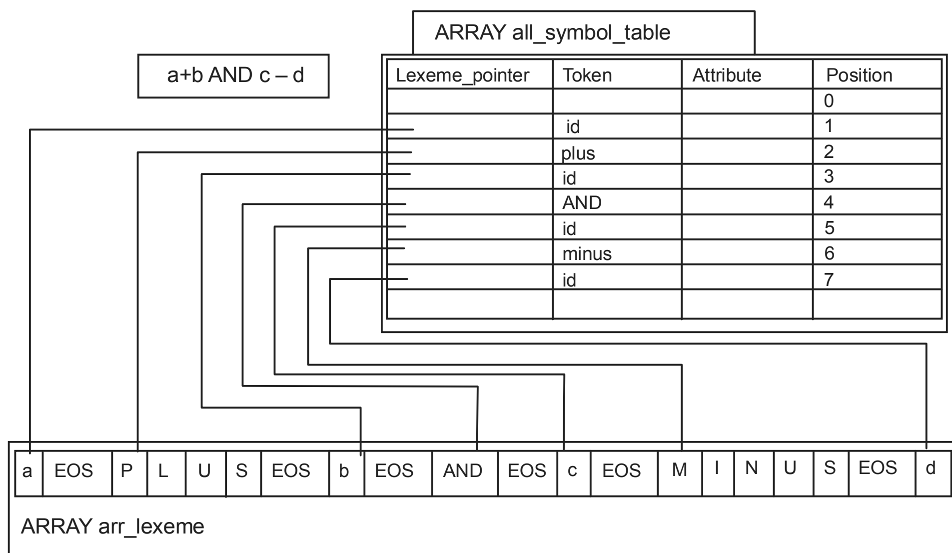

Figure 9.1: *Implemented symbol table*

When lexical analyzer reads a letter, it starts saving letters, digits in a buffer 'lex_bufffer'. The string collected in lex_bufffer is then looked in the symbol table, using the lookup operation. Since the symbol table initialized with entries for the keywords plus, minus, AND operator and some identifiers as shown in **figure 9.1** the lookup operation will find these entries if lex_buffer contains either div or mod. If there is no entry for the string in lex_buffer, i.e., lookup return 0, then lex_buffer contains a lexeme for a new identifier. An entry for the identifier is created using insert( ). After the insertion is made; 'n' is the index of the symbol-table entry for the string in lex_buffer. This index is communicated to the parser by setting tokenval to n, and the token in the token field of the entry is returned.

9.3 DATA STRUCTURE FOR SYMBOL TABLE

9.3.1 List

The simplest and easiest to implement data structure for symbol table is a linear list of records. We use single array or collection of several arrays for this purpose to store name and their associated information. Now names are added to end of array. End of array always marks by a point known as space. When we insert any name in this list then searching is done in whole array from 'space' to beginning of array. If word is not found in array then we create an entry at 'space' and increment 'space' by one or value of data type. At this time insert ( ), object look up ( ) operation are performed as major operation while begin_scope ( ) and end_scope( ) are used in simple table as minor operation field as 'token type' attribute etc. In implementation of symbol table first field always empty because when 'object-lookup' work then it will return '0' to indicate no string in symbol table.

Complexity: If any symbol table has 'n' names then for inserting any new name we must search list 'n' times in worst case. So cost of searching is O(n) and if we want to insert 'n' name then cost of this insert is O(n^2) in worst case.

Variable	Information(type	Space (byte)
a	Integer	2
b	Float	4
c	Character	1
d	Long	4

SPACE

Figure 9.2: *Symbol table as list*

9.3.2 Self Organizing List

To reduce the time of searching we can add an addition field 'linker' to each record field or each array index. When a name is inserted then it will insert at 'space' and manage all linkers to other existing name.

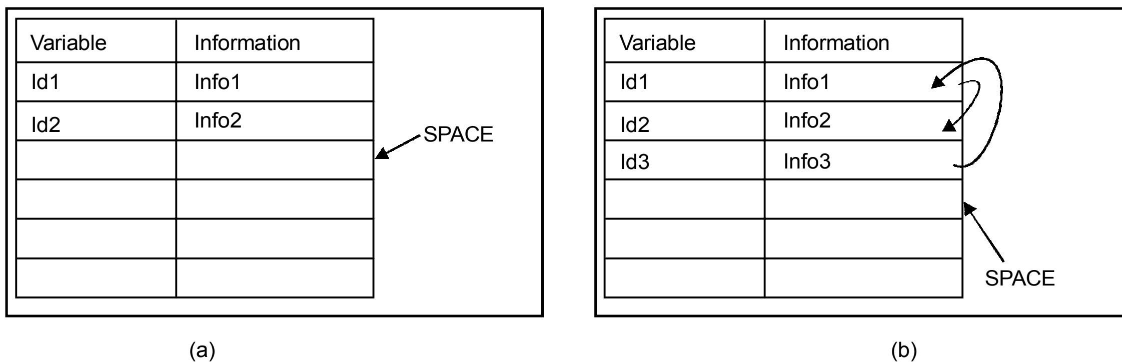

Figure 9.3: *Symbol table as self organizing list*

In above figure (a) represent the simple list and (b) represent self organzing list in which Id1 is related to Id2 and Id3 is related to Id1.

9.3.3 Hash Table

A **hash table**, or a **hash map**, is a data structure that associates keys with values 'Open hashing' is a key that is applied to hash table. In hashing –open, there is a property that no limit on number of entries that can be made in table. Hash table consist an array 'HESH' and several buckets attached to array HESH according to hash function. Main advantage of hash table is that we can insert or delete any number or name in O (n) time if data are search linearly and there are 'n' memory location where data is stored. Using hash function any name can be search in O(1) time. However, the rare worst-case lookup time can be as bad as O(n). A good hash function is essential for good hash table performance. A poor choice of a hash function is likely to lead to clustering, in which probability of keys mapping to the same hash bucket (i.e. a collision) occur. One organization of a hash table that resolves conflicts is chaining. The idea is that we have list elements of type:

```
class Symbol {
    public String key;
    public Object binding;
    public Symbol next;
    public Symbol ( String k, Object v, Symbol r ) { key=k; binding=v;
next=r; }
}
```

Structure of hash table look like as

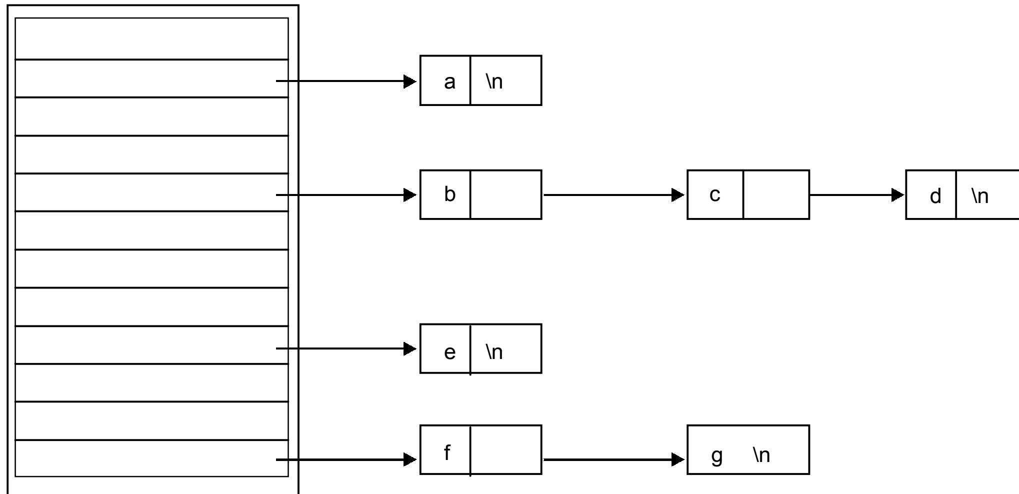

Figure 9.4: Symbol table as hash table (\n represent NULL)

9.3.4 Search Tree

Another approach to organize symbol table is that we add two link fields i.e. left and right child, we use these field as binary search tree. All names are created as child of root node that always follow the property of binary tree i.e. name <name ie and Namej <name. These two statements show that all smaller name than Namei must be left child of name otherwise right child of namej. For inserting any name it always follow binary search tree insert algorithm.

Example 9.2: Create list, search tree and hash table for given program for given program

```
int a,b,c;
int sum (int x, int y)
{
        a = x+y
        return (a)
}
main ()
{
        int u,
        u=sum (5,6);
}
```

(i) List

Variable	Information	Space(byte)
u	Integer	2 byte
a	Integer	2 byte
b	Integer	2 byte
c	Integer	2 byte
x	Integer	2 byte
y	Integer	2 byte
sum	Integer	2 byte

Space

Figure 9.5: *Symbol table as list for example 9.2*

(ii) Hash Table

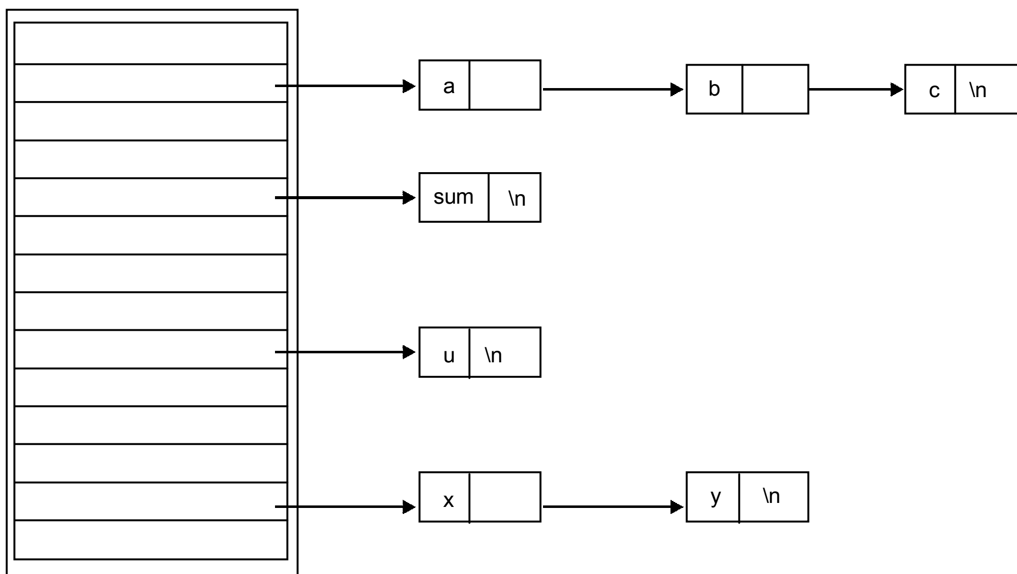

Figure 9.6: *Symbol table as hash table for example 9.2*

(iii) Search Tree

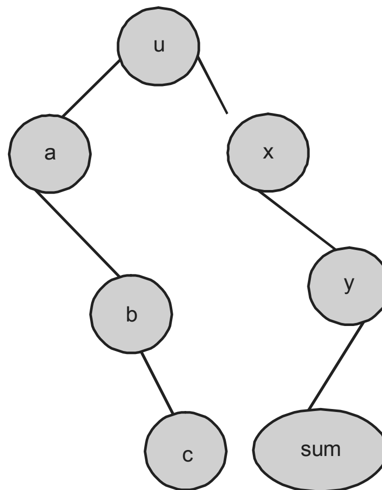

Figure 9.7: *Symbol table as search tree for example 9.2*

9.4 SYMBOL TABLE HANDLER

Any interface i.e. symbol table between source and object code must take recognisance of data-related concepts like *storage*, *addresses* and *data representation* as well as control-related ones like *location counter, sequential execution* and *branch instruction* which are fundamental to nearly all machines on which high-level language programs execute. Typically machines allow some operations which simulate arithmetic or logical operations on data bit patterns which simulate numbers or characters, these patterns being stored in an array like structure of memory whose elements are distinguished by addresses. In high-level languages these addresses are usually given mnemonic names. The context-free syntax of many high-level languages seems to draw a distinction between the "address" for a variable and the "value" associated with that variable and stored at its address. Hence we find statements like

$$X := X + 4$$

in which the '`X`' on the left of the '`:=`' operator actually represents an address (sometimes called the 'L-value' of '`X`') while the '`X`' on the right (sometimes called the 'R-value' of '`X`') actually represents the value of the quantity currently residing at the same address or we can say that each '`X`' i.e., all variable in the above assignment was syntactically a 'Designator'. Semantically these two designators are very different we shall refer to the one that represents an address as a 'Variable Designator' and to the one that represents a value as a 'Value Designator'.

To perform its task, the code generation interface will require the extraction of further information associated with user-defined identifiers and best kept in the symbol table. In the case of constants we need to record the associated values and in the case of variables we need to record the associated addresses and storage demands (the elements of array variables will occupy a contiguous block of memory).

Handling the different manners of entries that need to be stored in a symbol table can be done in various ways (described in section 9.4). One different method from 9.4 is object-oriented class based implementation one might define an abstract base class to represent a generic type of entry and then derive classes from this to represent entries for variables or constants. The traditional way, still required if one is hosting a compiler in a language that does not support inheritance as a concept, is to make use of union (in C++ terminology). Since the class-based implementation gives so much scope for exercises, we have chosen to illustrate the variant record approach which is very efficient and quite adequate for such a simple language. We extend the declaration of the '`TABLE_entries`' type to be

```
struct TABLE_entries {
  TABLE_alfa name;           // identifier
  TABLE_idclasses idclass;  // class
  union {
    struct {
      int value;
    } c;                    // constants
    struct {
      int size, offset;     // number of words, relative address
      bool scalar;          // distinguish arrays
    } v;                    // variables
  };
};
```

TRIBULATIONS

9.1 How would you check that no identifier is declared more than once?

9.2 How do real compilers deal with symbol tables?

9.3 How do real compilers keep track of type checking via symbol table? Why should name equivalence be easier to handle than structural equivalence?

9.4 Why do some languages simply prohibit the use of "anonymous types" and why don't more languages forbid them?

9.5 How do you suppose compilers keep track of storage allocation for struct or RECORD types, and for union or variant record types?

9.6 Find out how storage is managed for dynamically allocated variables in language like C++.

9.7 How does one cope with arrays of variable (dynamic) length in subprograms?

9.8 Identifiers that are undeclared by virtue of mistyped declarations tend to be trying for they result in many subsequent errors being reported. Perhaps in languages as simple as ours one could assume that all undeclared identifiers should be treated as variables and entered as such in the symbol table at the point of first reference. Is this a good idea? Can it easily be implemented? What happens if arrays are undeclared?

9.9 C language array declaration is different from a C++ one the bracketed number in C language specifies the highest permitted index value, rather than the array length. This has been done so that one can declare variables like

```
VAR Scalar, List[10], VeryShortList[0];
```