**Symbol Table ( ST )**

A *symbol table* is a data structure containing a record for each identifier, with fields for attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

When an identifier in source program is detected by the *lexical analyzer*, the identifier is entered into *ST*. however, the attributes of an identifier cannot be determined during lexical analysis, the remaining phases enter information about identifier into *ST* and then use this information in various ways.

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

- We need an empty symbol table, in which no name is defined.
- We need to be able to bind a name to an object. In case the name is already
  defined in the symbol table, the new binding takes precedence over the old.
- We need to be able to look up a name in a symbol table to find the object the name is bound to. If the name is not defined in the symbol table, we need to be told that.
- We need to be able to enter a new scope.
- We need to be able to exit a scope, reestablishing the symbol table to what it was before the scope was entered.

**Symbol Table Contents :-**

A symbol table is most often conceptualized as series of rows, each row containing a list of attributes values that are associated with a particular variable. The kinds of attributes appearing in *ST* are dependent to some degree on the nature of language for which compiler is written. For example, a language may be typeless, and therefore the type attribute need not appear in *ST* .The following list of attribute are not necessary for all compilers, however, each should be considered for a particular compiler:-

1- **Variable Name:** A variable's name most always reside in the *ST*. major problem in *ST* organization can be the variability in <u>the length of identifier names.</u> For languages such as BASIC with its one - and two - character names and FORTRAN with names up to six characters in length, this problem is minimal and can usually be handled by storing the complete identifier in a fixed - size maximum length fields. While there are many ways of handling the storage of variable names, two popular approaches will be outlined, one which facilitates quick table access and another which supports the efficient storage of variable names. The provide quick access, yet sufficiently large, maximum variable name length. A length of sixteen or greater is very likely adequate, the complete identifier can then be stored in a fixed - length fields in
*ST*, in this approach, table access<u> is fast but the storage of</u> <u>short variable names is inefficient.</u>

A second approach is to place a string **Descriptor** in the name filed of the table. The descriptor contains (*position and length*) subfields. The pointer subfield indicates the position of the first character of the name in a general string area, and the length subfield describes the number of characters in the name. therefore, this approach results in slow table access, but the savings in storage can be considerable.

**2- Object Time Address:-** The relative location for values of variable at run time.

**3- Type:-** This fields is stored in *ST* when compiling language having either *implicit* or *explicit* data type. For *typeless* language such as "BASIC" this attribute is excluded. " FORTRAN" provides an example of what mean by *implicit* data typing. Variables which are not declared to be particular type are assigned default types implicitly (variables with names starting with I, J, K, L, M, or N are *integer*, all other variable are *real*).

**4- Dimension of array or Number of parameters for a procedure.**

**5- Source line number at which the variable is declared.**

**6- Source line number at which the variable is referenced.**

**7- Link filed for listing in alphabetical order.**

**Operation on *ST*:-**

The two operations that are most commonly performed on *ST* are: ***Insertion*** & ***Lookup (Retrieval)*** .For language in which explicit declaration of all variables is mandatory, an insertion is required when processing a declaration. If *ST* is *Ordered*, then insertion may also involve a lookup operation to find allocations at which the variable's attributes are to be placed. In such a situation an insertion is

at least as expensive as retrieval. If the ST is not ordered, the insertion is simplified but the retrieval is expensive.

Retrieval operations are performed for all references to variables which don't involve declaration statements.

The retrieved information is used for *semantic checking* and *code generation*. Retrieval operations for variables which have not been previously declared are detected at this stage and appropriate error messages can be emitted. Some recovery from such semantic errors can be achieved by posting a warning message and incorporation the nondeclared variable in *ST*.

When a programming language permits implicit declarations of variable reference must be treated as an initial reference, since there is no way of knowing a priori of the variable's attributes have been entered in *ST*. Hence any variable reference generates a lookup operation followed by an insertion if the variable's name is not found in *ST*.

For *block - structured languages,* two additional operations are required: **Set & Reset**.

The Set operation is invoked when the beginning of a block is recognized during compilation. The complementary operation, the Reset operation is applied when the end of block is encountered. Upon block entry, the set operation establishes a new sub table (within the *ST*) in which the attributes for the variables declared in the new block can be stored. Because an new sub table is established for each block, the duplicated variable- name problem can be resolve. Upon block exit the reset operation removes the sub table entries for the variables of the completed block.

## ST Organizations:-

The primary measure which is used to determine the complexity of a ST operation is the average length of search. This measure is the average number of comparisons required to *Retrieve* a

ST record in a particular table organization, the

name of variable for which an insertion or lookup operation is to be performed will be referred to as the *search argument*.

**1- Unordered ST:** The simplest method of organization *ST*,
is to add the attribute entries to the table in the order in which the variable are declared. In an insertion operation no comparisons are required.

**2- Ordered ST :** In this and following organization, we
described *ST* organization in which the table position of a variable's set of attributes is based on the variable's name. An *insertion* operation must be accompanied by *lookup* procedure which determines where in *ST* the variables attribute should be placed. The insertion of new of attributes may generate some additional overhead primarily because other sets of attributes may have to be moved in order to a chive the insertion.

**3- Tree - structured ST :**The time to performed an insertion
operation can be reduced by using a tree - structured type of storage organization.