# Lexical Analysis and Symbol Tables

## I. Symbol Tables

- A Symbol Table is an important *data structure* created and maintained by the compiler in order to keep track of information about source-program constructs, such as variable and function names-denoted by identifiers, objects, classes etc.
- The information stored is collected by *analysis phase*, and used by all phases, more importantly, by *synthesis phase* to generate target codes.
- Typically, a symbol table needs to support multiple declarations of the same id within a program.
- Possible information stored in symbol tables:
    - ✓ Name of the identifier, that is, the lexeme such as i, x_1, func1, etc.;
    - ✓ Type of the identifier, that is, variable, procedure/function, label, etc.;
    - ✓ Value/return-value type;
    - ✓ Last assigned value (for variables);
    - ✓ Number and types of parameters (for functions);
    - ✓ Position in the storage, for example, line number, where it is first found;
    - ✓ Scope of the current definition;
    - ✓ Other, like array limits, structure/record fields, function parameters, etc.

- Important issues to be considered for formation of symbol table entries

    *i)* *Scope*

    Global, Local; Nested Blocks: Symbol tables can be allocated and removed in a stack-like fashion, separate table for separate scopes may also be generated; Static or lexical scoping versus Dynamic or runtime scoping*; Formal function parameters; Multiple references to same variables / functions; ….

    ii) *Operations on symbol tables*: Lookup, insert, modify, delete

    iii) *Format and storage of entries*: arrays of structures, linked lists, trees, etc.

    iv) *Access methodology*: linear search, binary search, tree search, hashing, etc.

    v) *Location of storage*: Primary memory, Primary + secondary

- Implementation of symbol tables depends heavily upon designers of a particular compiler.

## II. Formation and storage of symbol table entry by the lexical analyzer

A token in the token stream generated by the lexical analyzer has the following format:

*<Token Name, Attribute value>*

Token name: It is an *abstract symbol representing a kind of lexical unit*, e.g., a particular keyword, or a sequence of input characters denoting an identifier. Say, *id* for all identifiers.

Attribute value: Reference (pointer) to symbol table entry

*<id, 1>*, *<id, 2>*, …

For certain tokens, especially operators, punctuation, and keywords, the attribute values are omitted as they are directly placed in the token stream.

*In examples above, a token has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the pointer to the entry.*

Simplified form of a static symbol-table entry for a C compiler:

| Sl. No. | Name | Id type | Value type | Scope | More |
|---|---|---|---|---|---|

## III. A brief example

```
float x1 = 3.125;

double f2_xy(int x, float y)
 {
   double z;
   z = 0.01+x/5.5+0.25*y;
   return z;
 }

int main(void)
{
   int n1;   float n2,  x;
   double    z, n3;
   n1=25; n3=1.2;
   z = f2_xy(n1, n3);
   {
      float n1=10.5;
      n3 = n1*n2 + x1*z;
   }
   return 0;
}
```

| Sl. No. | Name | Id type | Value type | Scope | More |
|---|---|---|---|---|---|
| 1 | x1 | var | float | global | [T2 1] |
| 2 | f2_xy | func | double | global | [T3 1] |
| 3 | x | var | int | f2_xy | [T2 2] |
| 4 | y | var | float | f2_xy | [T2 3] |
| 5 | z | var | double | f2_xy | [T2 4] |
| 6 | main | func | int | global | [T3 2] |
| 7 | n1 | var | int | main | [T2 5] |
| 8 | n2 | var | float | main | [T2 6] |
| 9 | x | var | float | main | [T2 7] |
| 10 | z | var | double | main | [T2 8] |
| 11 | n3 | var | double | main | [T2 9] |
| 12 | n1 | var | float | main | [T2 10] |

- ‘More’ for a variable may mean pointer to another table that keeps info like last value assigned.
- ‘More’ for a function may mean pointer to another table that keeps info like number of parameters and their types.

**Implementation of Symbol Table:**

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted list) list
- Binary Search Tree
- Hash Table

Among all, *symbol tables are mostly implemented as hash tables*, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

**Operations:**

A symbol table, either linear or hash, should provide some operations on the table like:

**insert()**

- This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table.
- This operation is used to add information in the symbol table about unique names occurring in the source code.
- The format or structure in which the names are stored depends upon the compiler in hand.
- An attribute for a symbol in the source code is the information associated with that symbol.
- This information contains the value, state, scope, and type about the symbol.
- The *insert()* function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

*int a;* should be processed by the compiler as: *insert(a, int);*

**lookup()**

*lookup()* operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of *lookup()* function varies according to the programming language. The basic format should match the following: *lookup(symbol)*

This method *returns 0 (zero)* if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

*In Static or Lexical scoping, the compiler first searches in the current block, then in the global variables, then in successively smaller scopes.

In Dynamic or Runtime scoping, the compiler first searches in the current block and then successively all the calling functions.