Symbol table

**Why are Symbol Tables used in a compiler?**

* A compiler records the identifiers used in the source program and collects the information about various attributes of those identifiers
* Symbol Table stores the information about an identifier along with their attributes
* Attributes provide information about the identifier type, its scope and in the case of procedure names, such as the number and type of its arguments, the method for passing each argument (by reference) and the type returned if any
* To store or retrieve the information about an identifier quickly at compile time as well as at runtime environment
* To achieve compile time efficiency
* To verify that used identifier have been declared

* The Symbol Table is a data structure containing a record for each variable name or identifier with fields for the attributes of the name
* The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

**Symbol Table used by various Phases**

Symbol Table is used by various phased as:

**Lexical Analyzer phase:**Stores the information of identifiers such as its lexeme, its type, its position in storage etc, into symbol table if it is already not present

**Syntax analyzer phase:** Symbol Table is used to verify the information about the tokens being generated while checking syntax of the source code

**Semantic Analyzer phase:** Symbol Table is used for type compatibility or type checking issues

**Code generation Phase:** Symbol Table is used to know the runtime space allocated and the type of runtime space allocated.

**Symbol Table Format:**

A Symbol Table format that associates lexical names with their attributes is shown:

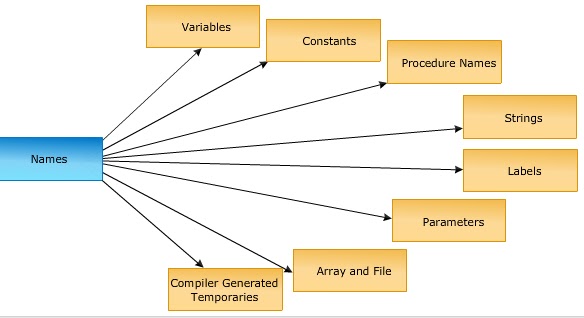
|  |  |
| --- | --- |
| Names | Attributes |
| Variables, Constants | Type, Size, line number where declared, scope, lifetime, binding, address |
| Procedure/Functions | Number of parameters, return types |
| Array | Number of dimensions, array bounds |

Table: Names and their Attributes

**Symbol Table Operations**

**The Symbol Table stores the following items:**

* Variable names
* Constants
* Procedure / Function names
* Strings
* Compiler generated temporaries
* Labels in source language
* Parameters
* Array and files

[](https://3.bp.blogspot.com/-yIXj_RNRKQ4/WZmpzlS_VgI/AAAAAAAAC1E/iSqv6ea87TQSN3eAc3D9lnGpkjQzd6jcgCLcBGAs/s1600/Symbol%2Btable%2Bnames.bmp)

**Compiler uses following types of information from symbol table:**

* Data type
* Name
* Declaring procedures
* Storage
* If structure or record, then pointer to structure table
* For parameters, whether parameter passing is by value or reference?
* Number and type of arguments passed to the function
* Base address

**Symbol Table Operations:**

Major operations required of a symbol table are:

* Insertion
* Search
* Deletions which are ‘Purely Logical’ depending scope and visibility and ‘not physical’

**Symbol Table and its relations with various phases of Compiler**

* Symbol Table is accessed at every stage of the compilation process

$\triangleright $Scanning includes insertion of new identifiers  
$\triangleright $Parsing access to the symbol table to ensure that an operand exists (declaration before use)

$\triangleright $Semantic analysis includes,  
$\triangleright $Determination of types of identifiers from declarations

$\triangleright $Type Checking to ensure that operands are used in type valid contexts

$\triangleright $Checking scope, visibility notations

Intermediate generation, memory allocation and for relative address (i.e., relative to a base address that is known only at run time) calculation

Optimization, all memory accesses through symbol table

Target code, translation of relative addresses to absolute address in terms of word length, word boundary etc.

**Organization for Block Structured Languages**

**Organization for Block Structured Languages:**

* The block structured language is a kind of language in which sections of source code is within some matching pair of delimiters such as “{“ and “}” or begin and end
* Such a section gets executed as one unit or one procedure or a function or it may be controlled by some conditional statements (if, while, do-while)
* Normally, block structured languages support structured programming approach

                Example: C, C++, JAVA, PASCAL, FORTRAM, LISP and SNOBOL

* Non-block structured languages are LISP, FORTRAN and SNOBOL

**Implementation of Symbol Table:**

* Each entry in the symbol table can be implemented as a record with several fields
* The following data structures are used for organization of block structured languages:

                Linear List

                Self-Organizing List

                Hashing

                Tree Structure

**Symbol table organization using Linear List:**

* A linear list of records is the easiest way to implement the symbol table
* In this method, an array is used to store names and associated information
* The new names are added to the symbol table in the order they arrive
* The pointer “available” is maintained at the end of all stored records
* To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name we get an error “use of undeclared name”
* While inserting a new name we should ensure that it is not already present. If it is already present then another error occurs, i.e., “Multiple Defined Name”.

**Advantages:**

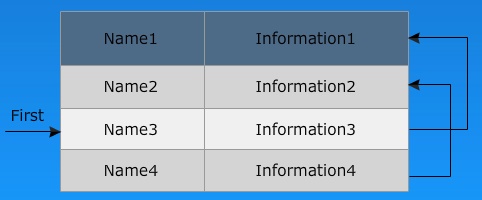
* It takes minimum amount of space
* Additions are simple
* Limitations
* Higher access time

*Linear List as follows:*

|  |  |
| --- | --- |
| **Name 1** | **Information 1** |
| Name 2 | Information 2 |
| . | . |
| . | . |
| . | . |
| . | . |
| Name n | Information n |

**Symbol Table organization using self-Organizing List:**

* In this method, symbol table is implemented using *linked list.*A link field is added to each record
* We search the records in the order pointed by the link of link field
* A pointer “First” is maintained to point to first record of the symbol table



* The reference to these names can be Name 3, Name 4, Name 2
* When the name is reference or created, it is moved to the front of the list
* The most frequently referred names will tend to be at the front of the list. Hence, access time to most frequently referred names wil be the least

**Advantages**

* It takes minimum amount of space
* Additions are simple

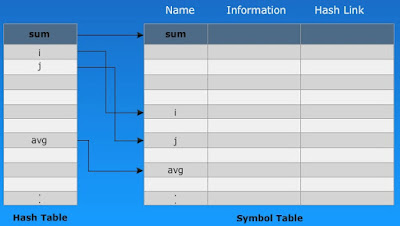
**Limitation**

* Higher access time

# Symbol Table Organizing Using Hashing

## Symbol Table Organizing Using Hashing

* Hashing is an important technique used to search the records of symbol table. This method is superior to list organization
* In hashing scheme, two tables are maintained – hash table and symbol table
* The hash table consists of K entries from 0, 1, 2, … to K-1. These entries are basically pointers to symbol table pointing to the names of symbol table.
* To determine whether the ‘Name’ is in symbol table, we use a hash function ‘h’ such that h (name) will result any integer between 0 to K-1. We can search any name by
* Position = h (name)
* Using the position we can obtain the exact locations of name in symbol table.
* The hash table and symbol table are shown below

[](https://3.bp.blogspot.com/-5rHenqVECPQ/Wa34ZBk4HFI/AAAAAAAAC4s/vyFqzxmsDwEWef5Fwv_ygwYbofq7Ef04gCLcBGAs/s1600/2.jpg)

* The hash function should not result in uniform distribution of names in symbol able
* The hash function should be such that there are minimum number of collisions
* Collision is a situation where hash function results in same location for storing the names
* Various collision resolution techniques are,

                Open addressing

                Chaining

                Rehashing

**Advantage**

* Quick search is possible
* Limitations
* Implementing hashing is complicated
* Extra space is required
* Obtaining scope of variable is very difficult.

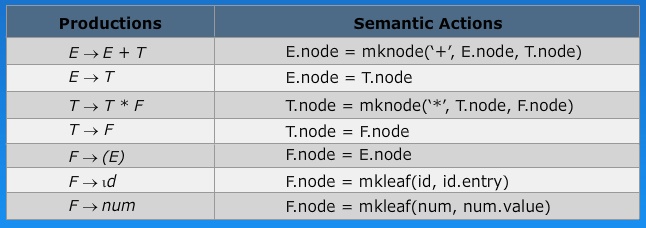
**Tree Structure Representation of Scope Information:**

* When the scope information is presented in hierarchical manner then it forms a tree structure representation which is an efficient approach for symbol table organization
* We add two links left and right in each record in the search trees
* Whenever a name is to be added first, the name is searched in the tree
* If it does not exist then a record for new name is created and added at the proper position
* It is alphabetical accessibility
* The typical data structure is shown on the screen

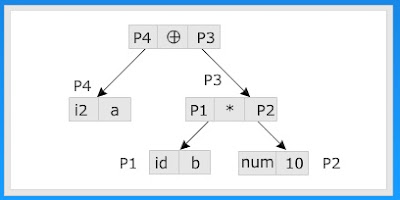
|  |  |  |  |
| --- | --- | --- | --- |
| **Left Child** | **Name of Symbol** | **Information** | **Right Child** |

* Construction of Binary Tree for Expressions:
* The nodes of binary tree for expressions can be constructed by using the following functions. Each function returns a pointer to a newly created node
* mlnode(L, op, R) creates an operator node with label op and two fields contain pointers to Left and Right with labels L and R
* mkleaf(id, entry) creates a leaf node with label id and field contains a pointer to the symbol table entry for ‘id’
* mkleaf(num, value) creates a leaf node with label num and field contains a value of that node

**Example**: Construction of a binary tree for the expression a + b \* 10. Consider the grammar

[](https://1.bp.blogspot.com/-Gzmx6xxU7mU/Wa34dyHHhwI/AAAAAAAAC4w/h7CM3c5xofQ0evRRaHAZuClAl4lgICrrgCLcBGAs/s1600/3.jpg)

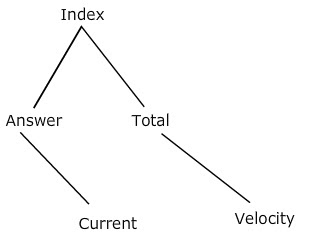
* Steps in the construction of the syntax tree for a + b \* 10
* Create a leaf node with label b that is p1 = mkleaf(id, b)
* Create a leaf node with label 10 that is p2 = mkleaf(num, 10);
* Create parent node label \* and two children nodes as p1, p2 that is p3 = mknode(‘\*’, p1, p2);
* Create a leaf node with label a that is p4 = mkleaf(id, a);
* Create a parents node with label + and two children nodes as p4, p3 that is p5 = mkknode(‘+’, p4, p3);
* Where p1, p2, p3, p4, p5 are the pointer to the newly created nodes.
* Syntax tree is

[](https://3.bp.blogspot.com/-6PuBeuTHpjY/Wa34mPvOL5I/AAAAAAAAC40/tlrOWffXy7wXAbJiSJ8Z7NKOH0p4jDVgACLcBGAs/s1600/4.jpg)

**Example:**

int index, total, velocity, current, answers;

The tree structure representation of above identifiers is:

[](https://2.bp.blogspot.com/-0pSk1kFWD-k/Wa34rMNaahI/AAAAAAAAC44/1IzVbaY_RDgUUxEikrFTOt5TgHeJMJ9vACLcBGAs/s1600/5.jpg)

**Block Structure and Non-Block Structure Storage Allocation**

**Block Structure and Non-Block Structure Storage Allocation:**

* Compiler must carry out the storage allocation and provide access to variables and data
* Allocation can be done in two ways:

**Compiler Time Allocation or Static Storage Allocation**

* A static storage allocation decision is taken by the compiler by looking only at the text of the program, but not by looking at what the program does when it executes
* Allocation is done at compile time

**Runtime Allocation or Dynamic Storage Allocation**

* A storage allocation decision is dynamic if it can be decided only while program is running
* Allocation is done at runtime
* Storage allocation can be done for two types of data variable:

                Local data

                Non local data

* The local data can be handled using activation record where as non-local data can be handled using scope information
* The block structured storage allocation can be done using static scope
* The non-block structured allocation can be done using dynamic scope

**Scope storage allocation includes,**

* Definition of procedure
* Declaration of a name or variable
* Scope of declaration

**Dynamic storage allocation includes,**

* Activation of procedure
* Binding of a name
* Lifetime of a binding

**Activation Record**

* Information needed for each execution of a procedure is stored in a record called activation record
* Procedure calls and returns are usually managed by a runtime stack called the control stack
* Each live activation has an activation record or a frame
* The root of activation tree is a the bottom of the stack
* The current execution path specifies the content of the stack with the last activation as record in the top of the stack
* Activation record contains 7 fields which are:

|  |
| --- |
| Returned |
| Actual parameters |
| Control link |
| Access link |
| Machine status |
| Local data |
| Temporary data |

* Size of each field of activation record can be estimated at compile time.

**Temporary Values:**

* Temporary values are those arising from the evaluation of expressions, in case where those temporaries cannot be held in registers.

**Local Data:**

* The local data is a data that is local to the execution of a procedure which is stored in activation record.

**Machine Status:**

* It holds the information regarding the status of the machine just before the call to a procedure. It contains the machine registers and program counter.

**Access Link:**

* It is an optional link
* It locates data needed by the called procedure which is found somewhere else i.e., in another activation record
* It also known as static link field

**Control Link:**

* it is an optional link
* It points to the activation record of the calling procedure
* It is also known as dynamic link field

**Actual Parameters:**

* The parameters of a calling procedure are known as actual parameters
* Actual parameters are passed to the called procedure
* These values are not placed in the activation record but rather in registers

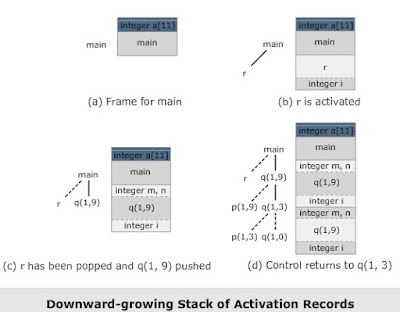
**Return Values:**

* Return values return the results from called procedure to calling procedure
* It is used to store the result of a function call

**Note:**

* Not all fields mentioned are used by all compilers or languages. Registers can take the place of one or more fields.
* Sizes of almost all fields can be determined at compile time. An execution arises when the procedure has a local array whose size is determined by the value of an actual parameter that is available only when the procedure is called at runtime

**Example:**

[](https://4.bp.blogspot.com/-PAhWZajyPvw/Wa4oCCQA5II/AAAAAAAAC5I/Nij7XUzlyzc_3nxGAqJr-YVjLyTJpsQ2ACLcBGAs/s1600/6.jpg)