Symbol Table

What is it

- Essential data structure used by compilers to remember information about identifiers in the source program
- Usually lexical analyzer and parser fill up the entries in the table, later phases like code generator and optimizer make use of table information
- Types of symbols store in the symbol table include variables, procedures, functions, defined constants, labels, structures etc.
- Symbol tables may vary widely from implementation to implementation, even for the same language

What is in Symbol Table

- Name
  - Name of the identifier
  - May be stored directly or as a pointer to another character string in an associated string table-names can be arbitrarily long
- Type
  - Type of the identifier: variable, label, procedure name etc.
  - For variables, its type: basic types, derived types etc.
- Location
  - Offset within the program where the identifier is defined
- Scope
  - Region of the program where the current definition is valid
- Other attributes: array limits, fields of records, parameters, return values etc.

Usage of Symbol Table Information

- Semantic Analysis – check correct semantic usage of language constructs, e.g. types of identifiers
- Code Generation – Types of variables provide their sizes during code generation
- Error Detection – Undefined variables. Recurrence of error messages can be avoided by marking the variable type as undefined in the symbol table
- Optimization – Two or more temporaries can be merged if their types are same

Symbol Table operations

- Lookup: Most frequent, whenever an identifier is seen it is needed to check its type, or create a new entry
- Insert: Adding new names to the table, happens mostly in lexical and syntax analysis phases
- Modify: When a name is defined, all information may not be available, may be updated later
- Delete: Not very frequent. Needed sometimes, such as when a procedure body ends

Design Issues

- Format of entries: Various formats from linear array to tree structured table
- Access methodology: Linear search, Binary Search, Tree search, Hashing, etc.
- Location of storage: Primary memory, partial storage in secondary memory
- Scope issues: In block-structured language, a variable defined in upper blocks must be visible to inner blocks, not the other way

Simple Symbol Table

- Works well for languages with a single scope
- E.g. assuming that variables are stored at any place is accessible even before its declaration
- Commonly used techniques are
  - Linear table
  - Ordered list
  - Tree
  - Hash table

Linear Table

- Simple array of records with each record corresponding to an identifier in the program

e.g.

int x, y;

float z

…

Procedure abc

….

L1:….

…..

Name	Type	Location
x	Int	Offset of x
y	Int	Offset of y
z	Float	Offset of z
abc	Procedure	Offset of abc
L1	Label	Offset of L1

String Table

char array

Linear Table

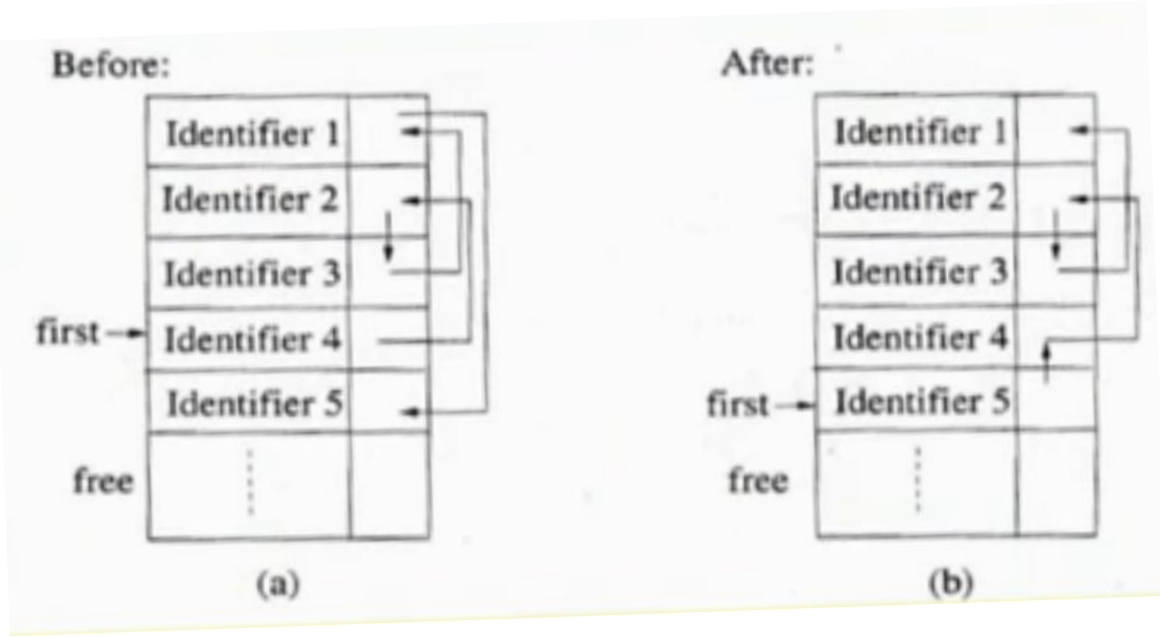
- If there is no restriction in the length of the string for the name of an identifier, string table my be used, with name field holding pointer
- Lookup, insert, modify take O(n) time
- Insertion can be made O(1) by remembering the pointer to the next free index
- Scanning most recent entries first may probably speed up the access due to program locality – a variable defined just inside a block is expected to be referred to more often than some earlier variables

Ordered list

- Variation of linear tables in which list organization is used
- List is sorted in some fashion, then binary search can be used with O(log n) time
- Insertion needs more time [O(n)]
- A variant – self-organizing list: neighbourhood of entries changes dynamically

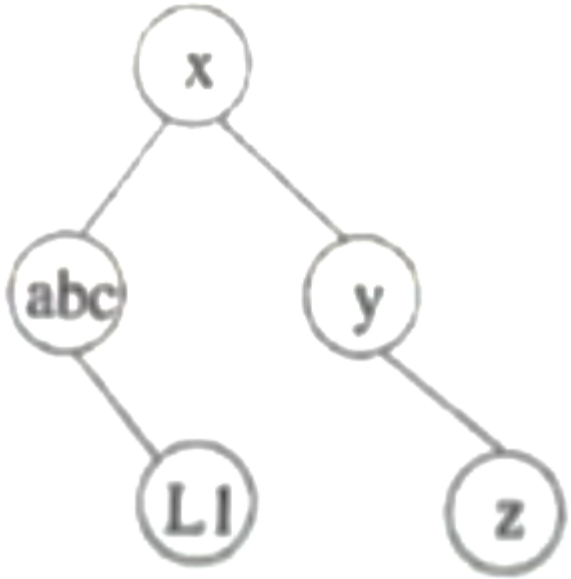
Self-Organizing List

- In Fig(a), Identifier4 is the most recently used symbol Identifier2, Identifier3 and so on
- In fig(b), Identifier5 is accessed next, accordingly the order changes
- Due to program locality, it is expected that during compilation, entries near the beginning of the ordered list will be accessed more frequently
- This improves lookup time



Tree

- Each entry represented by a node of the tree
- Based on string comparison of names, entries lesser than a reference node are kept in its left subtree, otherwise in the right subtree
- Average lookup time O(log n)
- Proper height balancing techniques need to be utilized



Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash Function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, has table is chosen to be of size between n and 2n for n keys (or for a chain at collision locations)
- There could be several locations empty

Desirable Properties

- Should depend on the name of the symbol. Equal Emphasis be given to each part
- Should be quickly computable
- Should be uniform in mapping names to different parts of the table. Similar names (such as, data1 and data2) should not cluster to the same address
- Computed value must be within the range of table index

Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is visible
- Block structured languages permit different types of scopes for the identifiers – scope rules for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

Scoping Rules

- Two categories depending on the time at which the scope gets defined
- Static or Lexical Scoping
  - Scope defined by syntactic nesting
  - Can be used efficiently by the compiler to generate correct references
- Dynamic or Runtime Scoping
  - Scoping depends on execution sequence of the program
  - Lot of extra code needed to dynamically decide the definition to be used

Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

```
Procedure P1
…
   Procedure P2
   …
   end procedure
Procedure P3
x=
…
```

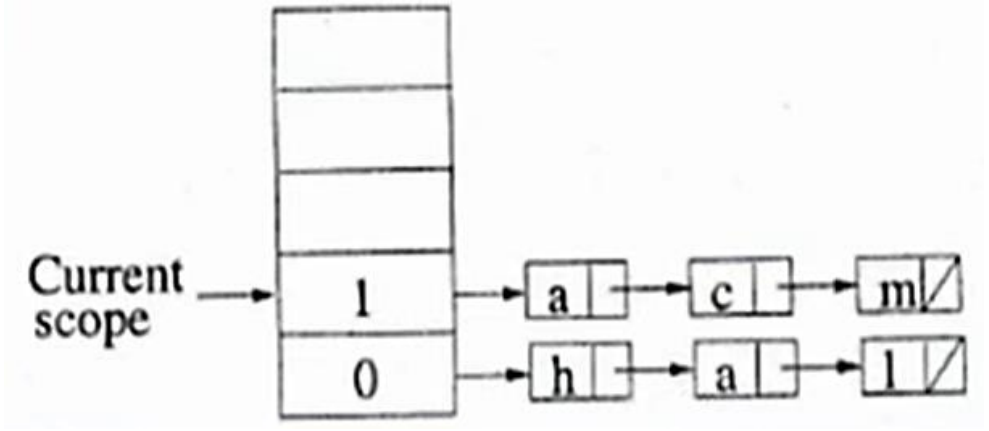
Current scope of x is P3, it has another open scope P1

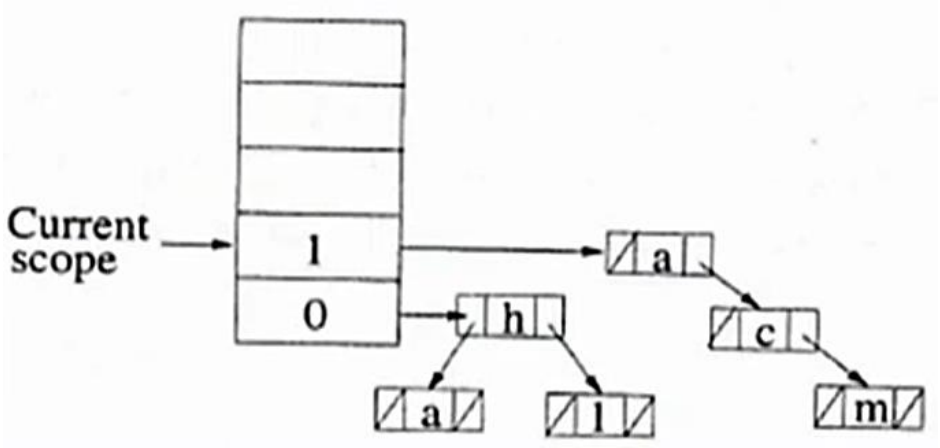
Visibility Rules

- Used to resolve conflicts arising out of same variables being defined more than once
- If a name is defined in more than one scope, the innermost declaration closest to the reference is sued to interpret
- When a scope is existed all declared variables in that scope are deleted and the scope is thus closed
- Two methods to implement symbols tables with nested scope
  - One table for each scope
  - A single global table

One Table per scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Drawbacks:
  - For a single-pass compiler, table can be popped out and destroyed when a scope is closed, not for a multi-pass compiler
  - Search may be expensive if variable is defined much above int the hierarchy
  - Table size allotted to each block is another issue
- Lists, Trees, Hash Tables can be used

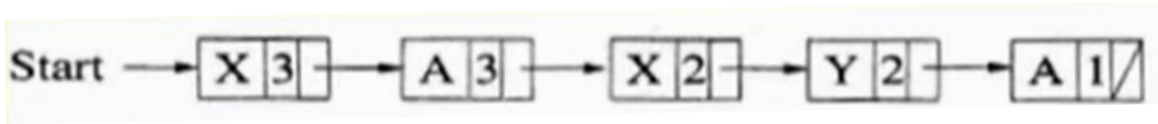




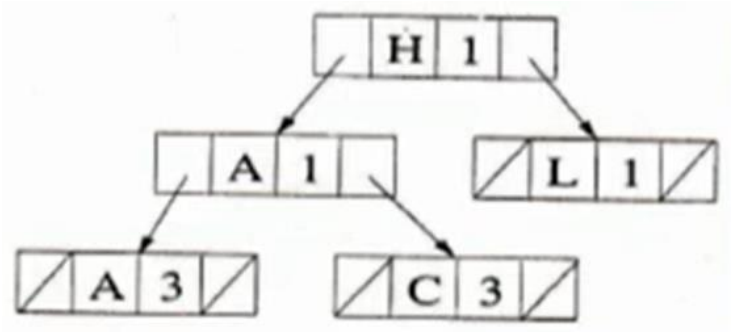
One Table for all scope

- All identifiers are stored in a single table
- Each entry in the symbol table has an extra filed identifying the scope
- To search for an identifier, start with the highest scope number, then try out the entries having next lesser scope number, and so on
- When a scope gets closed, all identifiers with that scope number are removed from the table
- Suitable particularly for single-pass compilers
- List, Tree and Has Table can be used

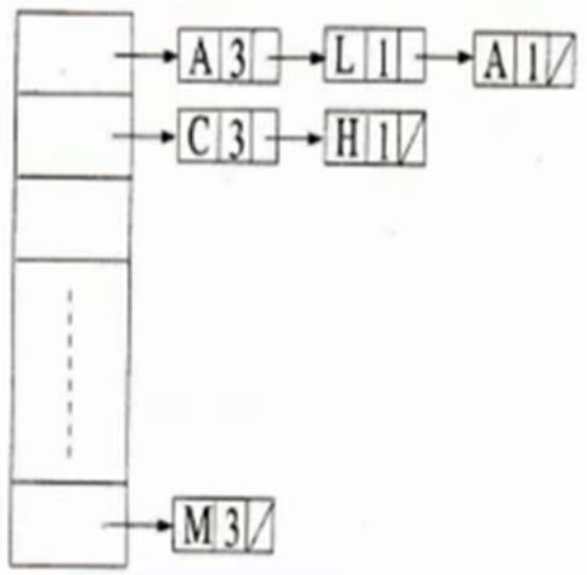
One Table for all scopes



Single LIST




```
{
    int H,A,L;
    …
    {
        …
        {
            int A,C;
            …
        }
    }
}
```



Hash Table

```
{
    int A;
    …
    {
        int X,Y;
        …
        {
            int X, A;
            …
        }
    }
}
```

Summary

- Symbol table helps in compilation process
- Lexical and Syntax analysis fills up symbol table, other phases just use it
- Symbol table organization is purely dependent on scope rules of language
- Linear Table, Ordered list, tree, hash table, etc are some commonly used Data structure