Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Symbol table

Study Guide

Mohammad Asif
Designation: Assistant Professor
CSE, PIT
Parul University

Parul® University
Vadodara, Gujarat
NAAC GRADE A++

INDEX

What is a Symbol Table?

A symbol table is a data structure that holds information about the identifiers used in a program. It maps the name of the identifier to its respective properties, such as its type, scope, and memory address. When writing code, you don't want to wade through a sea of variables and functions, trying to remember which is which. Instead, let the symbol table keep track of everything for you.

How does it work?

Imagine you're working on a compiler, and you come across a variable declaration, like int x; . The compiler needs to remember that x is an integer, and it'll need some space in memory to store its value. The symbol table steps in, acting like a librarian, keeping track of x, its type, and eventually, its memory location.

Later, when the compiler encounters a statement like x = 42; , it can refer to the symbol table, verify that x exists and is an integer, and then generate the appropriate code to store 42 at x's memory location.

Symbol Table Management

Managing a symbol table involves inserting, searching, and deleting entries. Let's dive into each of these operations and their importance.

Inserting Entries

When the compiler encounters a new identifier, it needs to be added to the symbol table. This process involves creating a new entry with the identifier's name, type, scope, and any other relevant information. The insertion process may also involve checking for duplicate identifiers within the same scope and throwing an error if one is found.

Searching Entries

When the compiler encounters an identifier in an expression or statement, it must look it up in the symbol table. Searching involves finding the relevant entry for the identifier based on its name and scope. For example, when processing a statement like y = x + 1; , the compiler searches for x in the symbol table to determine its type and memory location.

Deleting Entries

As the compiler processes the program, it may enter and exit various scopes. When exiting a scope, the entries associated with that scope should be removed from the symbol table. This cleanup process prevents memory leaks and ensures that identifiers from different scopes don't conflict.

In conclusion, symbol tables play a crucial role in compiler design, keeping track of identifiers and their attributes. Managing a symbol table involves inserting, searching, and deleting entries as the compiler processes the code. Selecting the appropriate data structure for implementing symbol tables is essential for achieving optimal performance.

Items stored in Symbol table

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Operations on Symbol Table

Following operations can be performed on symbol table-

- Insertion of an item in the symbol table.
- Deletion of any item from the symbol table.
- Searching of desired item from symbol table.

Applications of Symbol Table

- **Resolution of variable and function names:** Symbol tables are used to identify the data types and memory locations of variables and functions as well as to resolve their names.
- **Resolution of scope issues:** To resolve naming conflicts and ascertain the range of variables and functions, symbol tables are utilized.
- Information such as memory locations, are used to optimize code execution.
- **Code generation:** By giving details like memory locations and data kinds, symbol tables are utilized to create machine code from source code.
- **Error checking and code debugging:** By supplying details about the status of a program during execution, symbol tables are used to check for faults and debug code.
- **Code organization and documentation:** By supplying details about a program's structure, symbol tables can be used to organize code and make it simpler to understand.

Data Structures Used for Symbol Tables

The main work of a compiler is to convert a program written in a source language(high-level) to a low-level language(Object or Machine Language).

Data structure used for symbol tables include **hash tables, binary search trees, and linked lists**. The choice of data structure depends on the requirements of the application and the characteristics of the symbols being stored.

Objectives of Symbol Table Operations and Data Structures

A symbol table serves the following objectives:

- It verifies whether a variable has been declared.
- It stores all entities' names in a structured form in a single place.
- It determines the scope of a name.
- It implements type checking by verifying whether the assignments and expressions in the source code are semantically correct or not.
- It may be included in a process output to be used later during debugging sessions.
- It serves as a resource for creating a diagnostic report during or after program execution.

Hash tables are a fundamental data structure in compiler design, primarily used for implementing symbol tables. Symbol tables are crucial components that store information about identifiers (variables, functions, classes, etc.) encountered in the source code.

Here's how hash tables are used in this context:

- **Efficient Storage and Retrieval of Symbols:**
- When the compiler encounters an identifier during lexical analysis, it needs to store information about that identifier (like its type, scope, memory address, etc.) in the symbol table.
- Hash tables allow for very fast insertion and lookup of these identifiers. A hash function takes the identifier's name (a string) as input and calculates an index (a hash value) within an array. This index points to the location where the identifier's information is stored.
- This direct access, in contrast to linear or binary searches, significantly speeds up the compilation process, especially for large programs with many identifiers.
- **Handling Collisions:**
- Since different identifiers can potentially hash to the same index (a "collision"), hash tables employ collision resolution techniques.
- **Chaining**: is a common method where each array index (bucket) stores a linked list of all identifiers that hash to that index. When

searching for an identifier, the compiler first calculates its hash and then linearly searches the linked list at that index.

Scope Management:

- Hash tables can be extended to handle nested scopes in programming languages. When entering a new scope, new entries can be added to the symbol table, and when exiting a scope, the corresponding entries can be removed or marked as inactive.
- This ensures that the correct declaration of an identifier is retrieved based on its current scope.

Example:

Consider a variable x declared in a program. The compiler would:

- Calculate a hash value for the string "x".
- Use this hash value to determine an index in the hash table's array.
- Store a record containing information about x (e.g., its type int, its memory location) at that index, potentially within a linked list if collisions occur.
- Later, when x is referenced, the compiler quickly retrieves its information using the same hashing process.

Binary search, or more commonly, data structures that leverage its principles like Binary Search Trees (BSTs), are used in compiler design primarily for efficient searching and management of information within symbol tables.

Symbol Tables and Compiler Phases:

- **Lexical Analysis (Scanning):**

The lexical analyzer reads the source code and breaks it into tokens (keywords, identifiers, operators, etc.). Information about these tokens, especially identifiers, is stored in the symbol table.

- **Syntax Analysis (Parsing):**

The parser uses the symbol table to verify the grammatical structure of the program and ensure that identifiers are declared and used correctly according to the language rules.

- **Semantic Analysis:**

This phase checks for type compatibility, variable scope, and other semantic rules, often requiring lookups in the symbol table.

- **Code Generation:**

The symbol table is used to retrieve information about variables and functions needed to generate machine code.

How Binary Search (or BSTs) are Applied:

- **Efficient Lookups:**

When the compiler needs to find information about a specific identifier (e.g., its type, scope, memory address), it performs a lookup in the symbol table. If the symbol table is implemented using a sorted array and binary search, or more commonly, a Binary Search Tree, these lookups can be performed in logarithmic time complexity (O(log N)). This is significantly faster than a linear search (O(N)), especially for large programs with many identifiers.

- **Symbol Table Implementation:**
- **Sorted Array with Binary Search:** A simpler approach involves storing symbol table entries in a sorted array and using binary search to find specific entries. However, insertions and deletions in a sorted array can be expensive as they require shifting elements.

- **Binary Search Trees (BSTs):** BSTs are a more common and robust implementation for symbol tables. They maintain a sorted order while allowing efficient insertions, deletions, and searches, all with an average time complexity of O(log N). Each node in a BST represents a symbol table entry, and the tree structure ensures that searching for a specific symbol is efficient.

Example:

Consider a program with many variables. When the compiler encounters a variable name, it needs to quickly check if it has been declared and retrieve its associated information. A BST-based symbol table allows the compiler to navigate the tree based on the variable name (the "key") and locate the corresponding entry rapidly.

In essence, binary search and BSTs are crucial in compiler design for optimizing the performance of symbol table operations, which are fundamental to various phases of the compilation process.

Linked lists are a versatile data structure employed in various stages of compiler design due to their dynamic nature and efficient insertion/deletion capabilities.

Key Applications of Linked Lists in Compiler Design:

- **Symbol Table Management:**

- Symbol tables, crucial for storing information about identifiers (variables, functions, etc.), can be effectively implemented using linked lists. Each node in the linked list can represent a symbol, containing its name, type, scope, and other relevant attributes.

- Linked lists allow for dynamic resizing of the symbol table as new symbols are encountered during compilation, avoiding the fixed-size limitations of arrays.
- **Intermediate Code Representation:**
- Intermediate code, such as three-address code or quadruples, can be represented as a linked list of instructions. Each node can represent an instruction, containing the operation and its operands.
- This representation facilitates easy manipulation and optimization of the intermediate code, as instructions can be inserted, deleted, or reordered by simply adjusting pointers.
- **Abstract Syntax Tree (AST) Construction:**
- While trees are the primary structure for ASTs, linked lists can be used to manage the children of a node or to represent the sequence of statements within a block.
- Each node in the AST can have pointers to its children, which might be organized as a linked list to allow for dynamic addition or removal of child nodes.
- **Managing Free Memory Blocks:**
- In compilers that perform their own memory management, linked lists can be used to maintain a list of available (free) memory

blocks. This "free list" allows for efficient allocation and deallocation of memory during compilation.

- **Implementing Stacks and Queues:**
- Linked lists are a common underlying implementation for stacks and queues, which are frequently used in various compiler phases. For instance, a stack might be used for parsing (e.g., in a pushdown automaton), while a queue might be used for managing tasks or processing data in a specific order.

Representation of "scope"

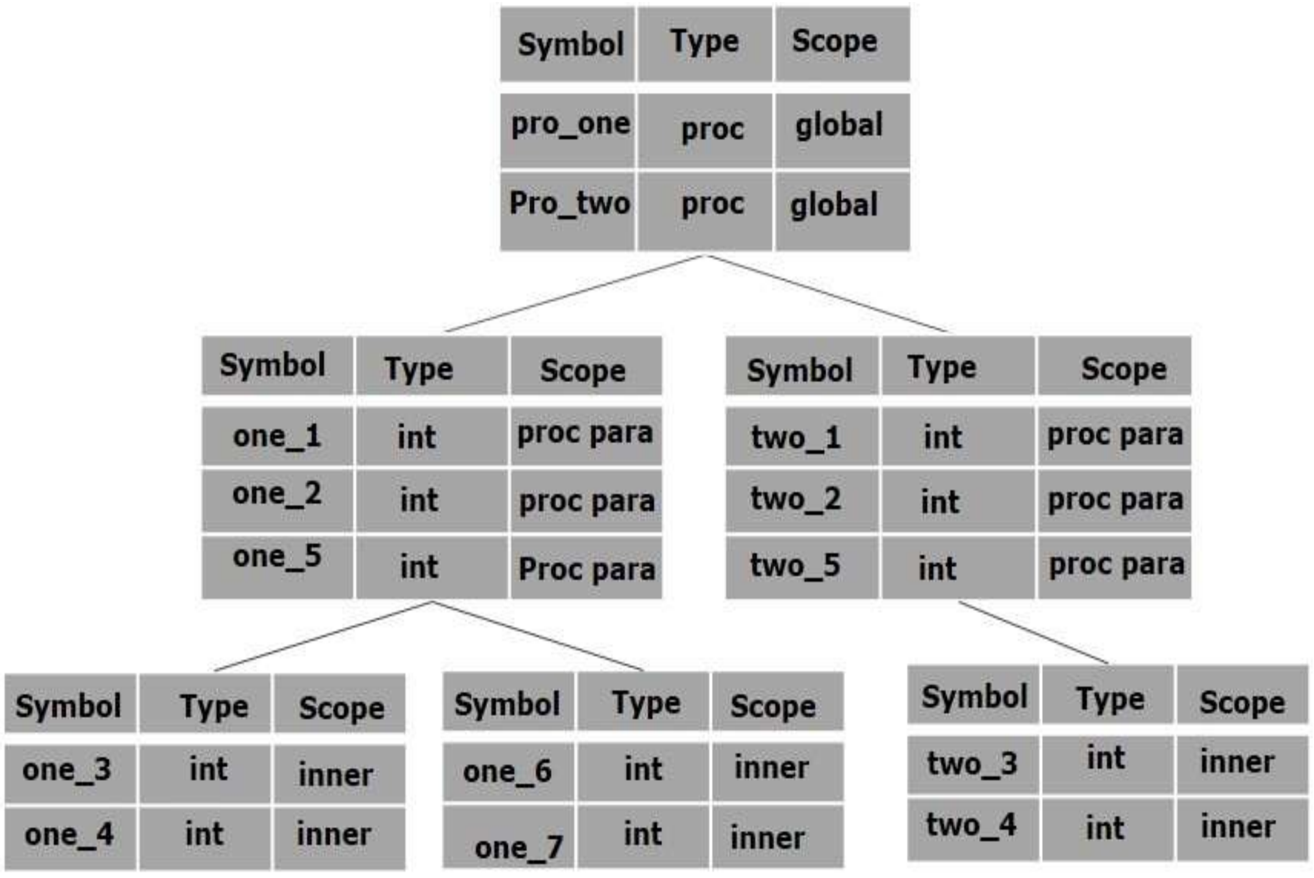
A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
  {
  int one_1;
  int one_2;

    {          \
    int one_3;     |_  inner scope 1
    int one_4;     |
    }          /

  int one_5;

    {          \
    int one_6;     |_  inner scope 2
    int one_7;     |
    }          /
  }
```

```
void pro_two()
  {
  int two_1;
  int two_2;

    {           \
    int two_3;      |_  inner scope 3
    int two_4;      |
    }           /

  int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



Symbol	Type	Scope
pro_one	proc	global
Pro_two	proc	global

Symbol	Type	Scope
one_1	int	proc para
one_2	int	proc para
one_5	int	Proc para

Symbol	Type	Scope
two_1	int	proc para
two_2	int	proc para
two_5	int	proc para

Symbol	Type	Scope
one_3	int	inner
one_4	int	inner

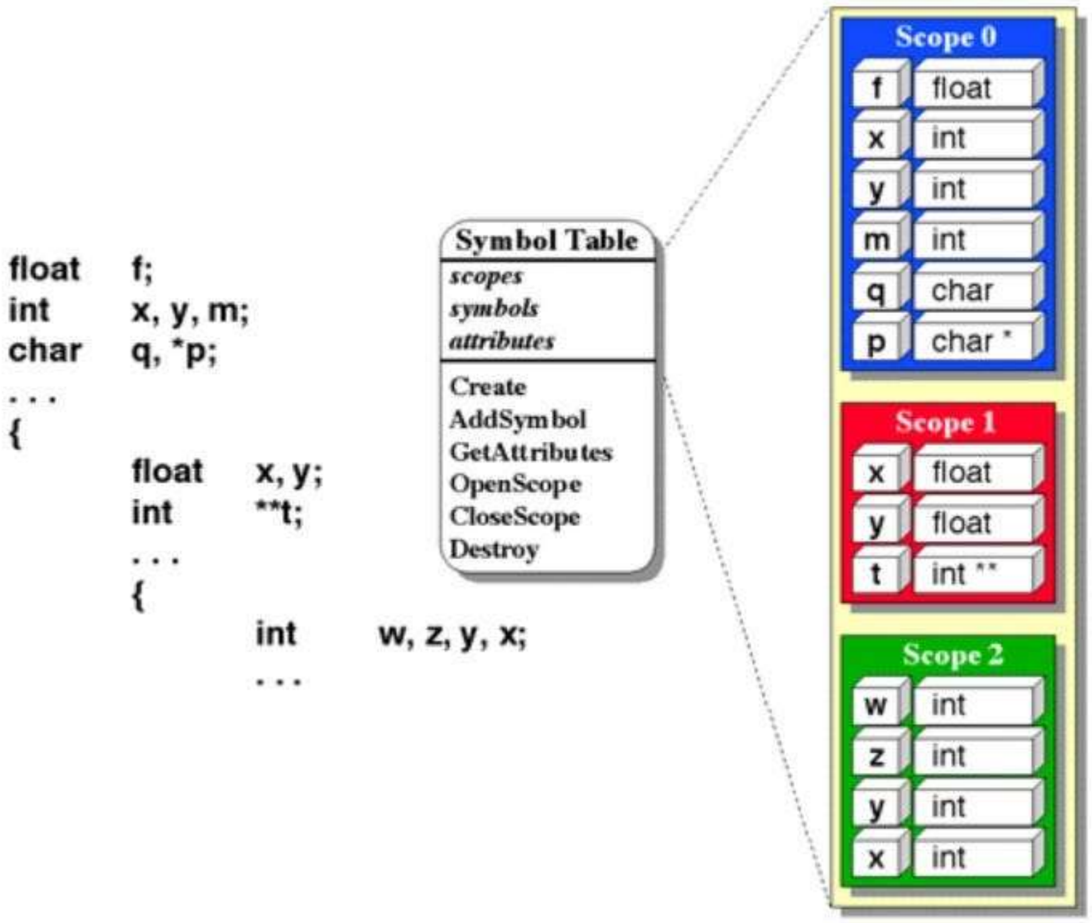
Symbol	Type	Scope
one_6	int	inner
one_7	int	inner

Symbol	Type	Scope
two_3	int	inner
two_4	int	inner

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- First a symbol will be searched in the current scope, i.e. current symbol table.
- If a name is found, then search is completed, else it will be searched in the parent symbol table until,
- Either the name is found or global symbol table has been searched for the name.



References:

1. Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson Education. (Also known as “The Dragon Book”)

2. Holub, A. I. (1990). Compiler Design in C. Prentice-Hall of India.

3. https://cratecode.com/info/symbol-table-management.

Parul® University
Vadodara, Gujarat
NAAC GRADE A++

https://paruluniversity.ac.in/