UNIT-4

Symbol Table

Symbol table is an important data structure used in a compiler.
Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

1. <symbol name, type, attribute>

For example, suppose a variable store the information about the following variable declaration:

1. **static int** salary

then, it stores an entry in the following format:

1. <salary, **int**, **static**>

The clause attribute contains the entries related to the name.

Implementation

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

- Linear (sorted or unsorted) list
- Hash table
- Binary search tree

Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

1. **int** x;

Should be processed by the compiler as:

1. insert (x, **int**)

lookup()

In the symbol table, lookup() operation is used to search a name. It is used to determine:

- The existence of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.
- Initialization of the symbol.
- Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

```
1. lookup (symbol)
```

This format is varies according to the programming language.

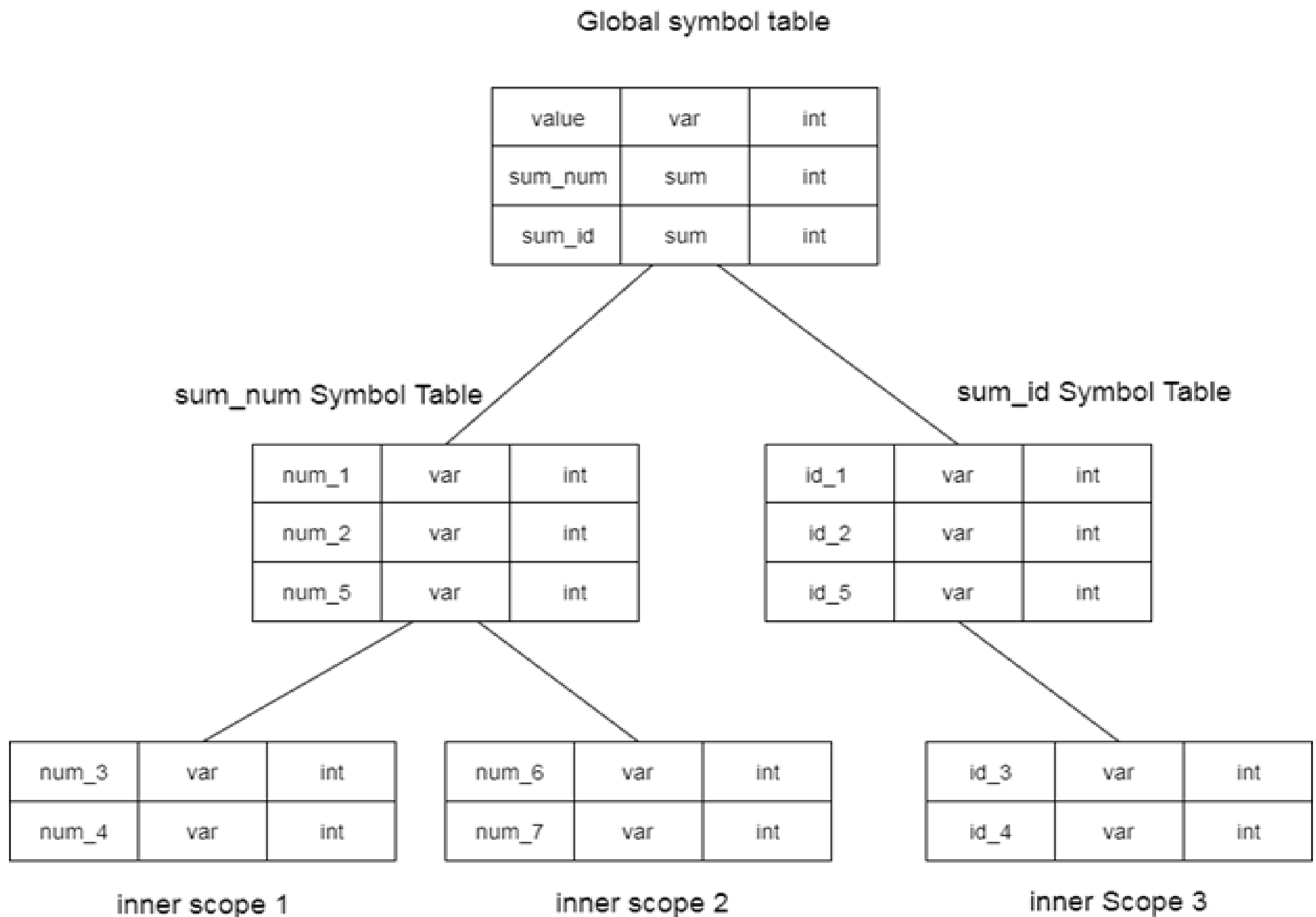
Data structure for symbol table

- A compiler contains two type of symbol table: global symbol table and scope symbol table.
- Global symbol table can be accessed by all the procedures and scope symbol table.

The scope of a name and symbol table is arranged in the hierarchy structure as shown below:

```
1. int value=10;
2.
3. void sum_num()
4.     {
5.       int num_1;
6.       int num_2;
7.
8.           {
9.            int num_3;
10.           int num_4;
11.          }
12.
13.      int num_5;
14.
15.          {
16.           int_num 6;
17.           int_num 7;
18.          }
19.    }
20.
21. Void sum_id
22.    {
23.      int id_1;
24.      int id_2;
25.
26.          {
27.           int id_3;
28.           int id_4;
29.          }
30.
31.      int num_5;
32.    }
```

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



The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.
Data structure hierarchy of symbol table is stored in the semantic analyzer. If you want to search the name in the symbol table then you can search it using the following algorithm:

- First a symbol is searched in the current symbol table.
- If the name is found then search is completed else the name will be searched in the symbol table of parent until,
- The name is found or global symbol is searched.

Representing Scope Information

In the source program, every name possesses a region of validity, called the scope of that name.

The rules in a block-structured language are as follows:

1. If a name declared within block B then it will be valid only within B.
2. If B1 block is nested within B2 then the name that is valid for block B2 is also valid for B1 unless the name's identifier is re-declared in B1.

- These scope rules need a more complicated organization of symbol table than a list of associations between names and attributes.
- Tables are organized into stack and each table contains the list of names and their associated attributes.
- Whenever a new block is entered then a new table is entered onto the stack. The new table holds the name that is declared as local to this block.
- When the declaration is compiled then the table is searched for a name.
- If the name is not found in the table then the new name is inserted.
- When the name's reference is translated then each table is searched, starting from each table on the stack.

For example:

```
1.  int x;
2.  void f(int m) {
3.      float x, y;
4.  {
5.          int i, j;
6.          int u, v;
7.  }
8.  }
9.  int g (int n)
10. {
11.     bool t;
12. }
```

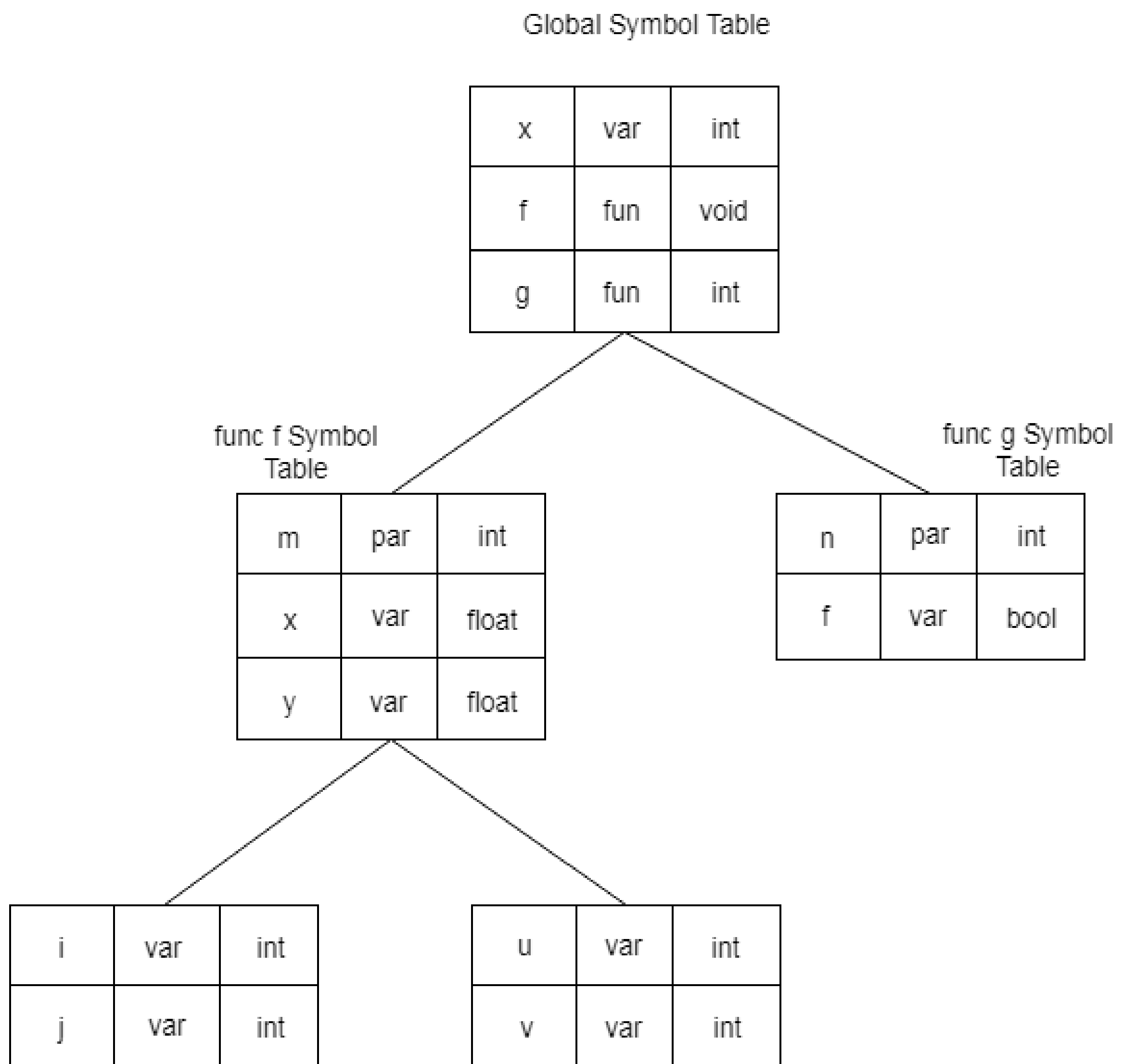
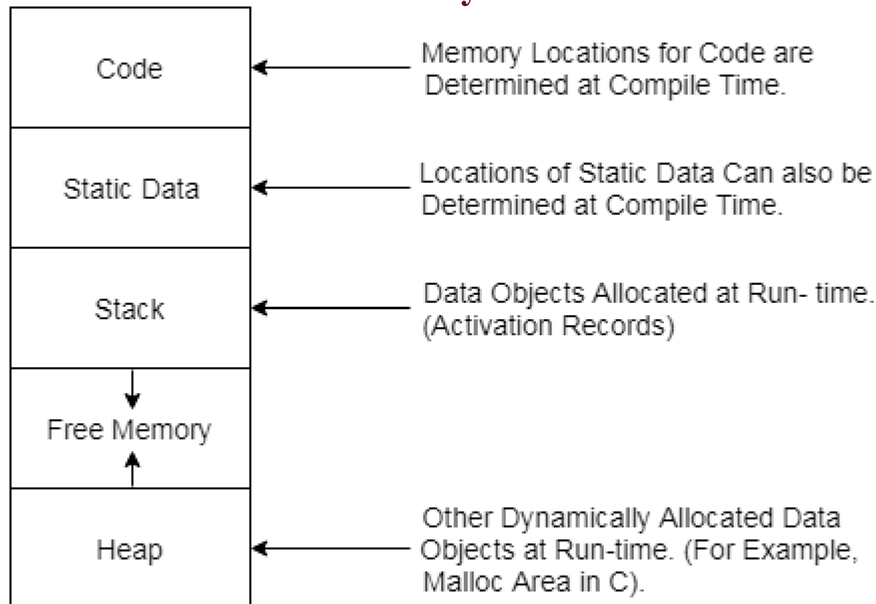


Fig: Symbol table organization that complies with static scope information rules

Storage Organization

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Subdivision of Run-time Memory:



- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:
  1. Generated executable code
  2. Static data objects
  3. Dynamic data-object- heap
  4. Automatic data objects- stack

Activation Record

- Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
- When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- Activation record is used to manage the information needed by a single execution of a procedure.
- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:

Return value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

Return Value: It is used by calling procedure to return a value to calling procedure.
Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.
Control Link: It points to activation record of the caller.
Access Link: It is used to refer to non-local data held in other activation records.
Saved Machine Status: It holds the information about status of machine before the procedure is called.
Local Data: It holds the data that is local to the execution of the procedure.
Temporaries: It stores the value that arises in the evaluation of an expression.

Storage Allocation

The different ways to allocate memory are:

1. Static storage allocation
2. Stack storage allocation
3. Heap storage allocation

Static storage allocation

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

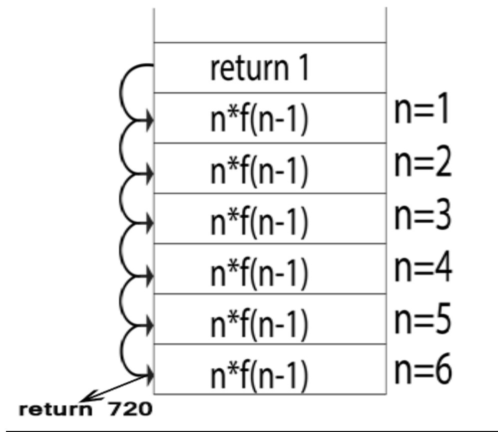
Heap Storage Allocation

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

Example:

```
1. fact (int n)
2. {
3.    if (n<=1)
4.        return 1;
5.    else
6.        return (n * fact(n-1));
7. }
8. fact (6)
```

The dynamic allocation is as follows:



Lexical Error

During the lexical analysis phase this type of error can be detected.
Lexical error is a sequence of characters that does not match the pattern of any token. Lexical phase error is found during the execution of the program.

Lexical phase error can be:

- Spelling error.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.
- To remove the character that should be present.
- To replace a character with an incorrect character.
- Transposition of two characters.

Example:

```
1. Void main()
2. {
3.     int x=10, y=20;
4.     char * a;
5.     a= &x;
6.     x= 1xab;
7. }
```

In this code, 1xab is neither a number nor an identifier. So, this code will show the lexical error.

Syntax Error

During the syntax analysis phase, this type of error appears. Syntax error is found during the execution of the program.

Some syntax error can be:

- Error in structure
- Missing operators
- Unbalanced parenthesis

When an invalid calculation enters into a calculator then a syntax error can also occurs. This can be caused by entering several decimal points in one number or by opening brackets without closing them.

For example 1: Using "=" when "==" is needed.

```
1. 16  if (number=200)
2.   17      count << "number is equal to 20";
3.   18  else
4.   19    count << "number is not equal to 200"
```

The following warning message will be displayed by many compilers:

Syntax Warning: assignment operator used in if expression line 16 of program firstprog.cpp

In this code, if expression used the equal sign which is actually an assignment operator not the relational operator which tests for equality.

Due to the assignment operator, number is set to 200 and the expression number=200 are always true because the expression's value is actually 200. For this example, the correct code would be:

```
1. 16   if (number==200)
```

Example 2: Missing semicolon:

```
1. int a = 5        // semicolon is missing
```

Compiler message:

```
1. ab.java:20: ';' expected
2. int a = 5
```

Example 3: Errors in expressions:

```
1. x = (3 + 5;  // missing closing parenthesis ')'
2. y = 3 + * 5;   // missing argument between '+' and '*'
```

Semantic Error

During the semantic analysis phase, this type of error appears. These types of error are detected at compile time.

Most of the compile time errors are scope and declaration error. **For example:** undeclared or multiple declared identifiers. Type mismatched is another compile time error.

The semantic error can arises using the wrong variable or using wrong operator or doing operation in wrong order.

Some semantic error can be:

- Incompatible types of operands
- Undeclared variable
- Not matching of actual argument with formal argument

Example 1: Use of a non-initialized variable:

```
1. int i;
2. void f (int m)
3. {
4.     m=t;
5. }
```

In this code, t is undeclared that's why it shows the semantic error.

Example 2: Type incompatibility:

```
1. int a = "hello";     // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
1. String s = "...";
2. int a = 5 - s;     // the - operator does not support arguments of type String
```