UNIT-4 (COMPILER DESIGN)

An important part of any compiler is the construction and maintenance of a dictionary containing names and their associated values, such type of dictionary is called a symbol table. In other words a symbol table is introduced as, "The part of an object table that gives the value of each symbol (usually as a section name and an offset) is called the symbol table." After syntax tree have been constructed, the compiler must check whether the input program is type correct (called type checking and part of the semantic analysis) or not, during type checking, a compiler checks whether the use of names (such as variables, functions, type names etc) is consistent with their definition in the program.

For example, if a variable x has been defined to be of type int, then x+1 is correct since it adds two integers while x[1] is wrong. When the type checker detects an inconsistency, it reports an error (such as ``Error: an integer was expected"). Another example of an inconsistency is calling a function with fewer or more parameters or passing parameters of the wrong type.

Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of a ***symbol table*.** A symbol table is a compile-time data structure. It's not used during run time by statically typed languages.

Formally, a symbol table maps names into declarations (called attributes), such as mapping the variable name x to its type int. More specifically, a symbol table stores:

For each type name, its type definition (eg. for the C type declaration typedef int* mytype, it maps the name mytype to a data structure that represents the type int*).

- For each variable name, its type. If the variable is an array, it also stores dimension information. It may also store storage class, offset in activation record etc.
- For each constant name, its type and value.
- For each function and procedure, its formal parameter list and its output type. Each formal parameter must have name, type, type of passing (by-reference or by-value), etc.

There are a numbers of phases associated with the construction of symbol tables. The principal phases of symbol table have included;

1. Building phase
2. Referencing phase

The building phase involves the insertion of symbols and their associated values into a table.
The referencing phase is the fetching or accessing of values from a table.

Note: In short "a symbol table is a file that compiler/Assemblers use to store symbols and their equivalent relative memory locations, values, data types for the following steps. It's an intermediate file, so the compiler where to store it. Some compiler deletes the file after processing."

Operation and Funtion on symbol table:

A symbol table is for storing the information associated with the various symbols used in the language. A symbol table can also be used in the lexical analysis stage to disambiguate tokens; this can make the parser job considerably easier.

We need to implement the following operations for a symbol table:

1. Insert (String key, Object binding)
2. Object_lookup (String key)
3. Delete or (begin_scope () and end_scope ())

1. ***Insert (s, t):*** Inserting a newly introduced variable or function name into the table with its required attribute. It return index of new entry for string's' and token't'.
2. ***Lookup (s):*** Lookup is used to looking up the attributes associated with a name. It return index of new entry for string's' or 0 if's' is not found.

3. ***Delete or (begin_scope () and end_scope ()):*** the delete operation removing items from the symbol table whose declarations are no longer visible. In a source program when we have a new block (i.e., when we encounter the token {), we begin a new scope (this is the begin_scope), similarly when we exit a block (i.e. when we encounter the token}) we remove the scope (this is the end_scope). When we remove a scope, we remove all declarations inside this scope. So basically, scopes behave like stacks.

This last operation/function on the symbol table is not actually necessary in a functional language as one lets the garbage collector handle the parts of the symbol table which are no longer needed.

Attributes of symbol table:

Symbol table is a data structure meant to collect information about names appearing in the source program or a symbol table is an abstract data type (ADT) for storing names encountered in the source program along with the relevant attributes for those names. Each entry in the symbol table is a pair of the form (Name, Information).

NAME	*INFORMATION*

1. Name:

Several alternatives are available for representing the name. symbol names may be:

a) Fixed length: the symbol table record can reserve a fixed amount of memory to store the name. for ex.: the original BASIC language (2 Bytes)
FORTRAN 66 (6 Bytes)
ANSI C (32 Bytes)

b) Variable length: A pointer to the start of the name is stored in the symbol table record. For ex.: Pascal.

The variable length alternative requires that the memory region for storing name is managed, either by:

i). Using the dynamic memory management functions provided by the OS (e.g. ; new and malloc)

ii). The compiler programmer.

2. Information:

The information field about a name may include the following attributes:

1. Type	**3.** Lifetime	**5.** Address
2. Size	**4.** Scope	**6.** Value

Type: The data structures needed to represent type information depend on the complexity of allowable types in the language.

1. Languages that allow only simple types can simply use an integer type code, e.g., int = 0, char =1, float=2, Boolean=3.
2. Language with compound or structured types like arrays or ranges in Pascal may need several fields, e.g., C-style arrays would require base type code and max. index fields.
3. Language that permit user defined types store a pointer to a type graph representing the type structure.

Lifetime: The lifetime of a variable is the period of time for which the variable has memory allocated. An integer code may be used to represent lifetime. There are two types of lifetime:

1. ***Static:*** Memory is allocated in a fixed location and size for the duration of the program. Global variable are usually static. In some languages like FORTRAN 66 all variables are static.
2. ***Semi- static:*** memory is allocated in a fixed size and location in the stack frame for the duration of a function invocation.

3. ***Semi- dynamic:*** during a particular function invocation, memory is allocated in a fixed size and location, but the size and location may change from invocation to invocation.
4. ***Dynamic:*** the size and location may be changed at nay time. Algol-68 allows the declaration of fully dynamic arrays.

Scope: Languages that support scope rules can store the scope value as an integer value. Scoped languages should choose a symbol table data structure that supports scope rules, i.e.: a variable defined in the current scope must be accessed before a variable in an enclosing scope. A stack organization provides this behavior. New scopes are pushed on the stack, and searching for a symbol begins at the top of the stack.

Value: Languages like Pascal and C++ allow declaration of constant variables. Initialization of variables with a value known at compiler time is also allowed in most languages. The symbol table can store a binary representation of the value that can be used to generate the initialization during code generation.

Address: The value stored for address depends on the lifetime of the symbol.

1. **Static variables:** The address of static variables is fixed and can be computed at compile time. The address is stored as an offset from some known value. For example; SPARC variables are stored in data segments, so we begin assigning address offsets from the start of the segment.
2. **Semi static variables:** Semi static variables are allocated storage in a stack frame. Their size and address is fixed relative to the start of the stack frame. An offset from some known point in the stack frame can be computed at compile time.
3. **Dynamic variables:** Dynamic variables are allocated memory on the heap. The heap is a region of memory from which chunks can be allocated as needed at run-time. The allocated chunks may be returned and reused in any order. A pointer to contain the address of the chunk is allocated, which may be in either static memory or in a stack frame.
4. **Semi dynamic variables:** Semi dynamic variable may be allocated either on the RTS or the heap. As for dynamic variables, allocate a pointer to access the memory.

Data Structure for Symbol table:

There are several data structure that is used in our symbol table, it including arrays, linear lists, search trees and Hash tables. We will consider the following factors when evaluating data structures;

1. Ease of programming
2. Speed of operations: a) search b) insert c) scope deletion
3. Size limits
4. Scope support
5. Popularity

Array: An array data structure for symbol table is sub divided into two categories;

1. **Unsorted array:**

   New symbols are added to the end of the table. Searching for a symbol starts at the end and is linear. The most recent scope is always at the end of the table. Following are some important point about this data structure:

   1) It is easy to program
   2) The brief about is an operation:
      a) It is slow in searching: ***O(n)***
      b) It is fast in insertion: ***O(1)***
      c) It is fast in scope deletion: ***O(1)***
   3) It has fixed size.

4) It has a good scope support.

2. **Sorted array:**

In this data structure new symbols are inserted in sorted order. Binary search is used. It has no scope support. Following are some important points about this data structure:

1) It is moderately easy to program.
2) The brief about its operations: as compared to unsorted array
   a) It is improved in searching: ***O(log n)***
   b) It is slow in insertion: ***O(n)***
   c) It is slow in scope deletion: ***O(n)***
3) It has fixed size.
4) It has poor scope support.

Linear list:

The simplest and easiest to implement data structure for symbol table is a linear list of records. A linear list data structure for symbol table is sub divided into two categories;

1. **Linear linked list:**

A linear linked list use single array or collection of several arrays for this purpose to store name and their associated information. In this data structure new symbols are inserted at the front of the list or to the end of array. End of array always marks by a point known as space. When we insert any name in this list then searching is done in whole array from 'space' to beginning of array. If word is not found in array then we create an entry at 'space' and increment 'space' by one or value of data type. At this time insert ( ), object look up ( ) operation are performed as major operation while begin_scope ( ) and end_scope( ) are used in simple table as minor operation field as 'token type' attribute etc. In implementation of symbol table first field always empty because when 'object-lookup' work then it will return '0' to indicate no string in symbol table.
Following are some important points about this data structure:

1) It is moderate programming difficulty.
2) The brief about its operations:
   a) It is slow in searching: **O(n)**
   b) It is fast in insertion: **O(1)**
   c) It is fast in scope deletion: **O(1)**
3) It has not size limit
4) It has good scope support.

Variable	Information(type	Space (byte)
a	Integer	2
b	Float	4
c	Character	1
d	Long	4

SPACE

Figure: symbol table for linear linked list

2. **Self Organizing List:**

To reduce the time of searching we can add an addition field 'linker' to each record field or each array index. When a name is inserted then it will insert at 'space' and manage all linkers to other existing name.

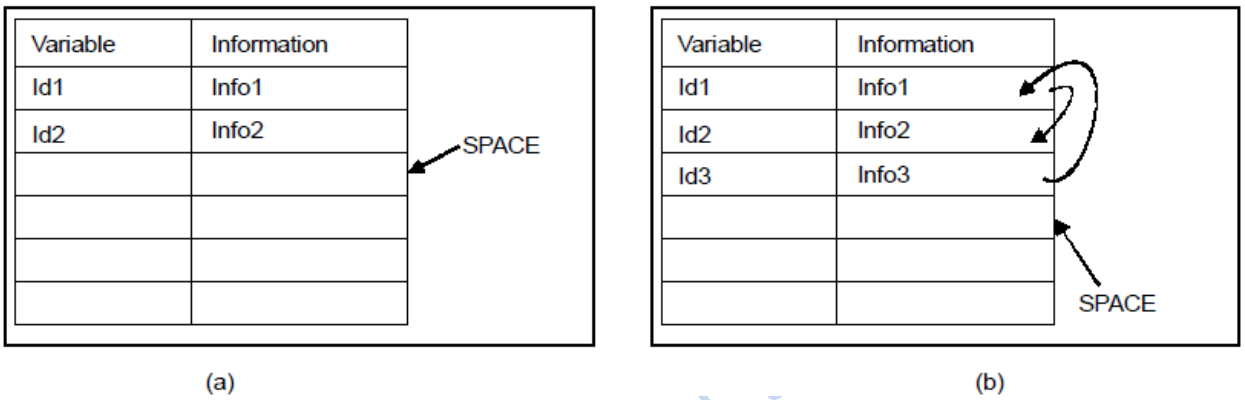

Figure: symbol table for self organizing list

Balanced search tree:

The structure of the tree represents the sorted order of the records. In this data structure maintaining balance is important because of the tendency of programmers to use systematic names with common prefixes/suffixes. It has no scope support; some problems as with sorted arrays. Following are some important points about this data structure:

1) It is moderate to high programming difficulty for balanced trees.
2) The brief about its operations:
   a) It is moderate in searching: **O(log n)**
   b) It is moderate in insertion: **O(log n)**
   c) It is fast in scope deletion: **O(1)**
3) It has no size limit.
4) It has poor scope support.

Hash table: A hash function data structure for symbol table is sub divided into two categories;

1. **Closed hash table:**

This data structure is an array of symbol table records inserted in a random order. A hash function applied to the name determines the location in the table where the symbol record should be stored. A collision resolution strategy must be employed to deal with name that hash to the same value. Following are some important points about this data structure:

   1) It moderate programming difficulty.
   2) The brief about its operations:
      a) It is fast in searching: **O(1)**
      b) It is fast in insertion: **O(1)**
      c) It is slow in scope deletion: **O(n)**
   3) It has fixed size.
   4) It has poor scope support.

2. **Open hash table:**

A **hash table**, or a **hash map**, is a data structure that associates keys with values 'Open hashing' is a key that is applied to hash table. This data structure is an array of pointers to linear linked lists of symbol table records. All symbols that have the same name hash to the name list. Insert new most records at the start of the list. It has good scope support; symbols in the most records recently entered scope are at the front of the list. Following are some important points about this data structure:

1) It has high programming difficulty
2) The brief about its operations:
  a) It is fast in searching: **O(1)**
  b) It is fast in insertion: **O(1)**
  c) It is slow in scope deletion: **O(1)**
3) It has no fixed size.
4) It has good scope support.

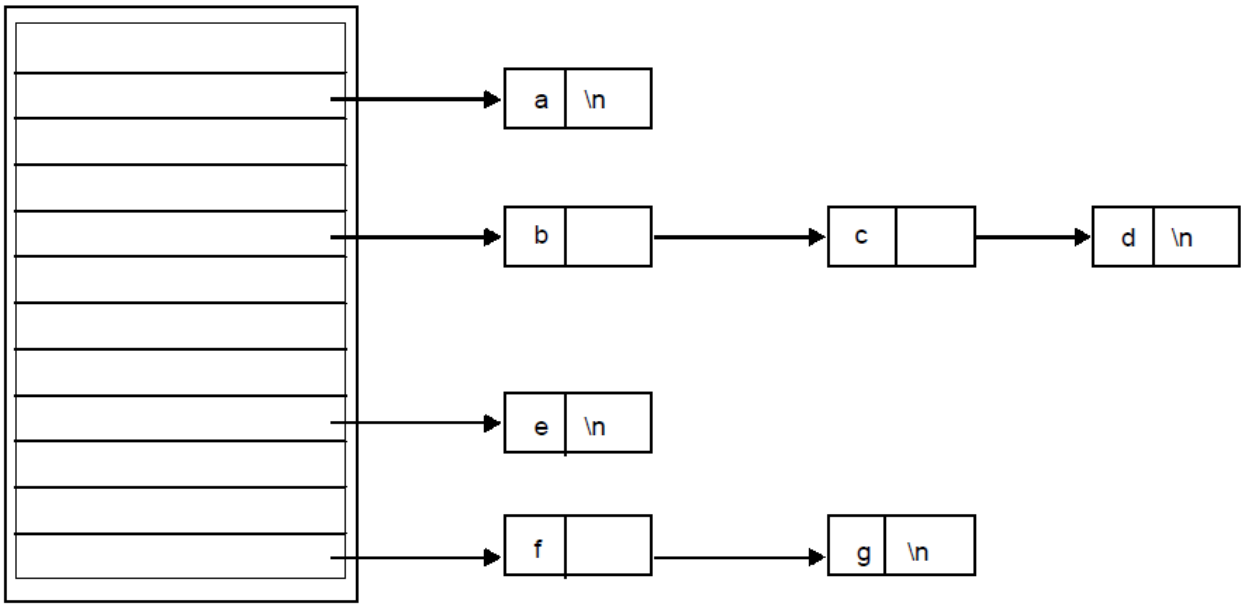

Figure: Structure of hash table in symbol table

Representing Scope Information:

The scope of a name is the portion of the program ove which it may be used or Each name possesses a region of validity within the source program called the scope of that name.
The rules governing the scope of names in a block-structured language are as follows:

- A name declared within block B is valid only within B.
- If block B1 is nested within B2, then any name that is valid for B2 is also valid for B1, unless identifier for that name is re-declared in B1.

These rules require a more complicated symbol table organization that simply a list of associations between names and attributes. One technique is to keep multiple symbol tables for each active block:

- Each table is list of names and their associated attributes, and the tables are organized on stack.
- Whenever a new block is entered, a new table is pushed on the stack.

- When a declaration is compiled, the table on the stack is searched for the name.
- If name is not found it is inserted.
- When a reference is translated, it is searched in all tables starting from top.

Another technique is to represent scope information in the symbol table.

- Store the nesting depth of each procedure block in the symbol table.
- Use the (procedure name, nesting depth) pair as the key to accessing the information from the table.
- The nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. It counts the number of procedure in the referencing environment of a procedure.

For example we consider the case of ALGOL;

ALGOL: the block structure in ALGOL makes all the names local to the block or procedure in which they are declared. The general structure of an ALGOL program is as:

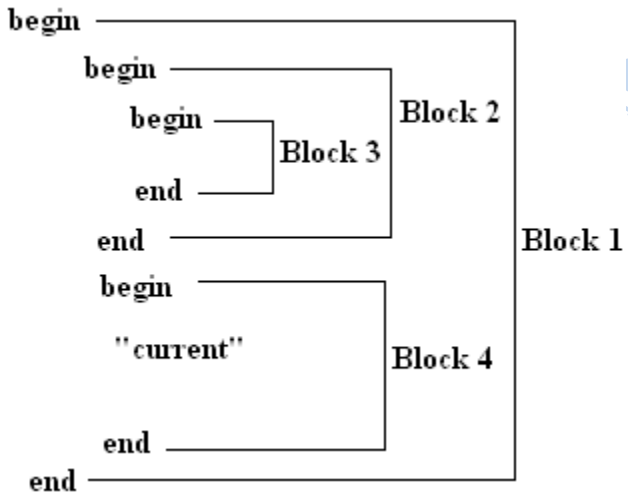

Figure: the structure of an ALGOL program

The most closely nested rule is used for binding identifiers to declarations.

Error Handling:

The programs are written by programmers(human beings) and therefore the programs can not be free from errors. Each phase of a compiler design encounters error. Detection and reporting of error in the source program is the main funciton of the compiler. A good compiler should be able not only to nature and source of common errors in programs and mechanisms by which each phase of the compiler can detect errors and recover from them. The techniques of error detection and recovery of errors from source progrma are performed by a component is known as error handler.

Soure of Error: There are various sources of error in any program. The most common source are:

- Errors due to user input.
- Errors due to devices.
- Errors due to physical limitations.
- Errors due to coding.

Goles of error handler: The main goles of error handler are as given below:

1) To detect the presence of errors and produce "meaningful" diagnostics.
2) To provide sufficient informaiton regarding the cause and nature of errors so that the user can manage to correct the program resomably.
3) To recover from the error quickly enough to be able to detect subsequent errors.
4) Error handling components should not slow down the processing of corrent programs.
5) To correct the error so as to generate the object code which can be executed subsequently.
6) Easy to program to and maintain.

Error reporting:

Error reporting is a common techique to print the offending line with a pointer to the position of a error. The parse might add a diagnostic message like "semicolon missing at this position" if it knows what likely error is.
An error reporting has two major works:

- Accurate indication of the position of the error.
- Provide helpful error message.

Classification of Errors: Most of the errors encountered during th process of program translation and execution can be classified as:

1. **Compile time errors:**
2. **Run time error:**

Compile time error: The error tha can be detected at compile time. The compile time errrors can be further divided in to the following three types:

- **Lexical errors: character** straams that don't match token patterns.
- **Syntactic Error:** tokens streams that don't match the grammar.

- **Semantic Errors:** syntactically legal statement that don't make sense.

Run time error: The errors that can be detected during the execution of a program. The run time error includes logical errors but the compiler can generatee code to check for the following things:

- **Divide by zero.**
- **Pointers going** beyond the program's memory.
- **Indices going** beyond the range of an array.
- **Infinite** loops or rcursive calls.

Errors that are encountered by virtually all of the phases of a compiler, shown below:

S.No.	Phases/Routine	Error
1.	Lexical anlysis	Misspelled next token e.g., misspelled identifier, keyword,or printer,
2.	Syntax analysis	Missing parenthesis e.g., ub MIN(A, 2*(3+B), errorneous arithmatic expressions an arithmetic expression with unbalanced parantheses etc.
3.	Semantic analysis	Type mismatch, errors of declaration and scope.
4.	Intermediate code generation	An operator whose operands have incompatible types.
5.	Code optimization	Not reachable statements due to wrongly written function calls.
6.	Code generation	A compiler vreated constant me too large to fit in a word of the target machine may be found.
7.	Symbol table	A multiply declared identifier with contradictory attributes.

Error Recovery Techniques:

Once an error is detected an reported by a phses, it must be able to recover form that error. There are various methods for error recovery. These are described as given below;

1. ***Panic-Mode Error Recovery*:** this technique Skipping the input symbols until a synchronizing token is found.
2. ***Phrase-Level Error Recovery:*** In this error recovery technique each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
3. ***Error-Productions:*** In this error recovery technique;
    1) If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
    2) When an error production is used by the parser, we can generate appropriate error diagnostics.
    3) Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
4. ***Global-Correction:*** In this error recovery technique;
    1) Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
    2) We have to globally analyze the input to find the error.
    3) This is an expensive method, and it is not in practice.