Unit-IV

Symbol table, Errors and Code optimization

- **The contents of a symbol table**

A symbol table is a table with two fields, a name field and an information field.

Symbol table is used to

- Determine whether a given name is in the table
- Add a new name to the table
- Access the information associated with a given name
- Add new information for a given name
- Delete a name or group of names from the table

a) Names and symbol table records

The simplest way to implement a symbol table is a linear array of records, one record per name.

A record normally consists of a known number of consecutive words of memory.

The identifier could be stored in the record as shown in following figure 2(a):

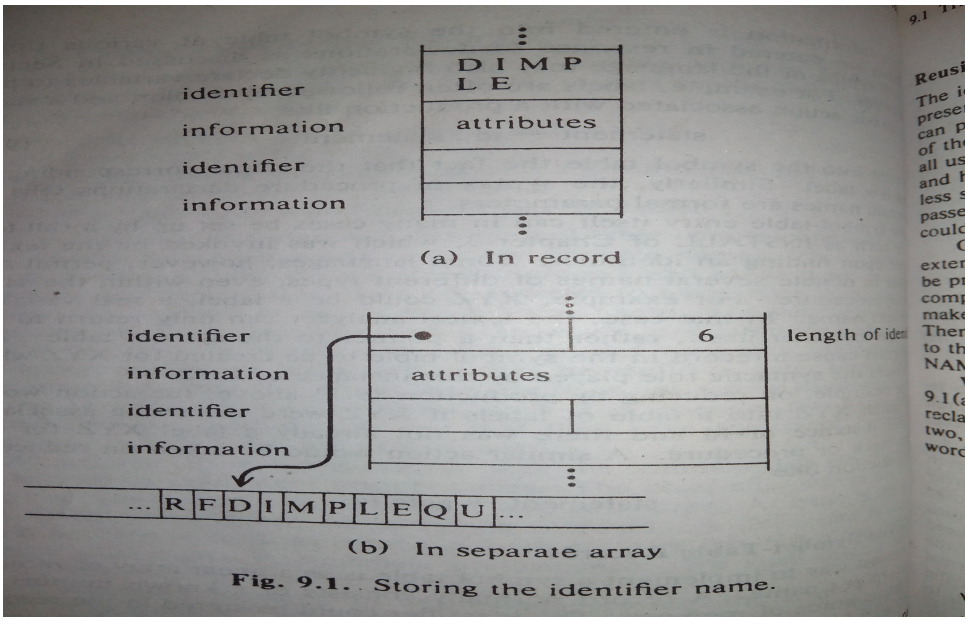


Fig. 9.1. Storing the identifier name.

This method is appropriate if there is fixed upper limit on the length of an identifier.

Ex. FORTRAN permits identifiers of up to eight characters

Figure 2(a) shows an appropriate format for this case, with an identifier filled out with blanks to make eight characters.

In the case of ALGOL, there is no limit on the length of an identifier and in PL/I permits limit of 31 characters, use the indirect scheme as shown in figure 2(b).

In the record for DIMPLE we find a pointer to another array of characters and a count (6) giving the length of the identifier. The pointer gives the position of the first character of the identifier. This indirect scheme permits the size of the name field in the symbol table itself to remain a constant.

The identifier used by the programmer to denote a particular name must be preserved in the symbol table until no further references to that identifier can possibly denote the same name. This is essential that all uses of the identifier can be associated with the same symbol table entry and hence the same name. However, a compiler can be designed to run in less space if the space used to store identifiers can be resumed in subsequent passes.

- **Data structures for symbol table**

During compilation process names get added in symbol table in which manner they encountered in program as well as the information about that name is also added.

If new information about an existing name is discovered then that information is also added. Thus in designing a symbol table mechanism, we would like a scheme that allows us to add new entries and find a scheme that allows us to add new entries and find existing entries in table efficiently.

There are three data structures used to implement symbol table:

1. Linear list

2. Binary tree

3. Hash table

1. Linear list or lists

It is a simplest and easiest to implement data structure.

We use a single array to store names and their associated information.

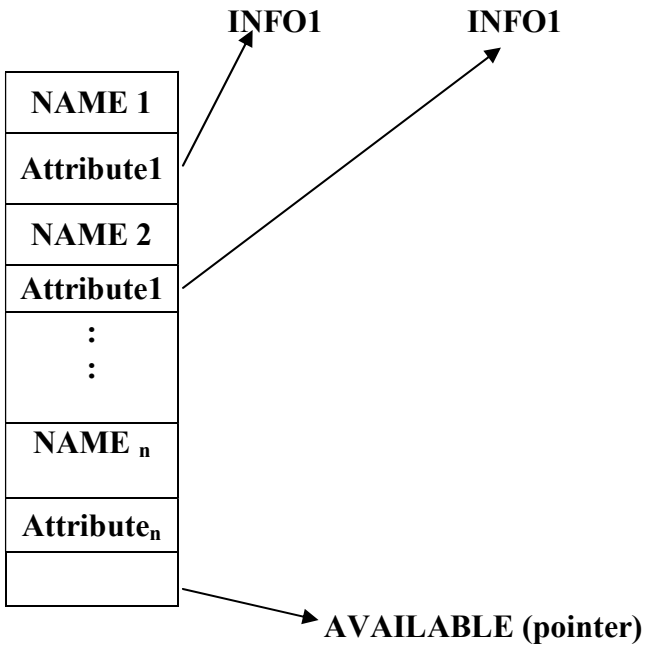
New names are added to the list in the order in which they are encountered.

To insert a new name, we must scan down the list to make sure that it is not already there. If not then add it otherwise an error message i.e. Multiply declared name.

When the name is located, the associated information can be found in words following next.

To retrieve information about a name, we search from the beginning of the array upto the position marked by AVAILABLE pointer, which indicates the beginning of the empty portion of array.

If we reach AVAILABLE pointer without finding NAME, we have a fault-the use of an undefined name.



To find data about the NAME, we shall on the average search N/2 names. So the cost of an enquiry is proportional to N.

One advantage of list organization is that the minimum possible space is taken in simple compiler.

ii) Trees

It is a more efficient approach to symbol table organization. Here we add two link fields LEFT and RIGHT to each record.

Following algorithm is used to look for NAME in a binary search tree where p is initially a pointer to the root.

1. while p≠null do

2. if NAME=NAME(p) then /* NAME found take action on success*/

3. else if NAME<NAME(p) then

p:=LEFT(p) /* visit left child*/

4. else /* NAME (p)<NAME*/

P:= RIGHT(p) /*visit right child*/

iii) Hash table

Hashing table technique is suitable for searching and hence it is implemented in compiler.

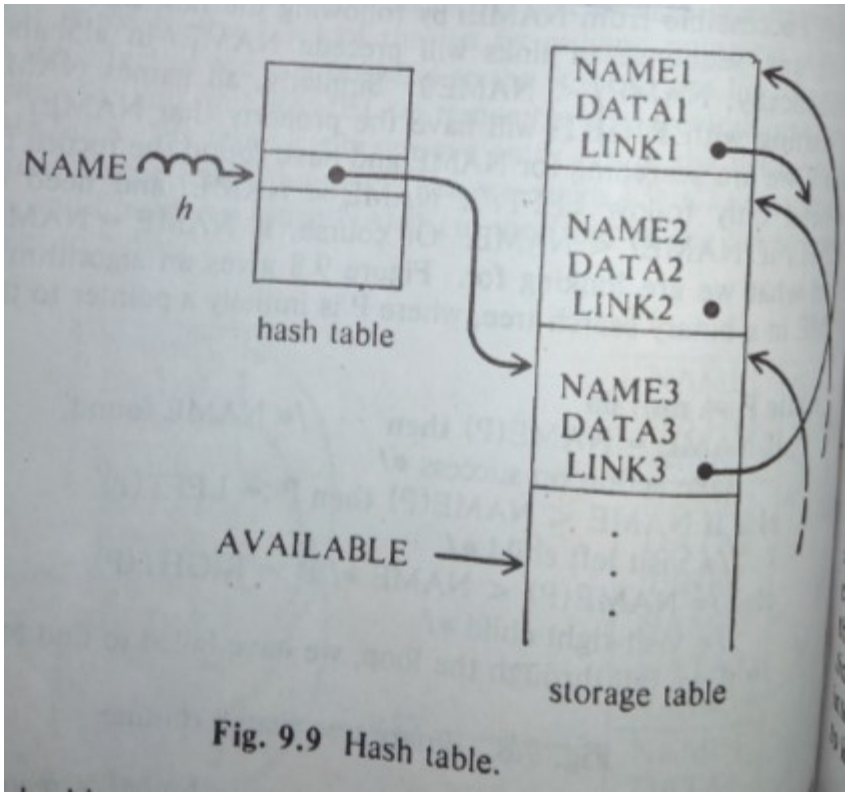


Fig. 9.9 Hash table.

Here, basic hashing schema is shown in above figure(3). Two tables: hash table and a storage table are used.

The hash table consists of k words numbered 0,1,2,…,k-1.These words are pointers into the storage enable to the heads of k separate linked lists I(some lists may be empty). Each record in the symbol table appears on one of these lists.

To determine whether NAME is in the symbol table, we apply NAME as a hash function h such that h(NAME) is an integer between 0 to k-1.

Hash function

The mapping between an item and the slot where that item belongs in the hash table is called the hash function.

Hash function will take an item in the collection and return an integer in the range of slot names between 0 to m-1.

Example:

Assume that we have the set of integer items 54, 28, 93, 17, 77 and 31.

Consider m=11 (number of slots of hash table)

0	1	2	3	4	5	6	7	8	9	10

Suppose we want to insert the above elements in the slots by using formula :

h(item)%m

54%11=4

28%11=6

93%11=5

17%11=7

77%11=0

31%11=9

After inserting above elements the hash table looks as shown below:

0	1	2	3	4	5	6	7	8	9	10
77				54	93	28	17			31

- **Errors and its types**

Introduction:

The compiler should be able not only to detect errors but also to recover from them. That is, even in the presence of errors, the compiler should scan the entire program and compile it to detect as many errors as possible.

Errors

There are a variety of ways in which a compiler can react to mistakes or errors in the source program .If the compiler follows unacceptable modes of behavior then it results in a system crash, emit invalid output and quit on the first detected error.

A compiler should attempt to recover from each error and continue analyzing its input.

i) A simple compiler may stop all activities other than lexical and syntactic analysis after the detection of the first error.

ii)A complex compiler may attempt to repair the error that is transform the errorneous input into a similar but legal input on which normal processing can be resumed.

iii)A more sophisticated compiler may attempt to correct the errorneous input by making a guess as to what the user intended,

Almost all compilers recover from common errors and continue syntax checking.

a) Reporting errors

Good error diagnostics helps to reduce debugging and maintenance effort. The good error diagnostics should have a number of properties:

1. The messages should pinpoint the errors in terms of the original source program rather than in terms of internal representation that is not understandable to the user.

The error messages should be tasteful and understandable by the user.

Ex. “missing right paranthesis in line 5” rather than cryptic error code such as “0H17”.

The messages should be specific and should localize the problem.

Ex. PI is not declared in procedure AREA rather than “missing declaration”.

The messages should not be redundant.

If a variable PI is undeclared that should be said once, not every time PI appears in the program.

b) Sources of error

1. The design specifications for the program may be inconsistent or faulty.

2. The algorithms used to meet the design may be inadequate or incorrect ("algorithmic errors");

3. The programmer may introduce errors in implementing the algorithms which may introduce logical errors or coding errors.

4. Keypunching or transcription errors can occur when the program is typed onto cards or into a file.

5. The program may exceed a compiler or machine limit not implied by the definition of the program language.

Ex:- An array may be declared with too many dimensions to fit the symbol table or an array may be too long to be allocated space at run time

6. A compiler can insert errors as it translates the source program into an object program (compiler error).

- **Types of errors**

1. Syntactic error

Following are some common examples of syntactic errors:

i. Missing right paranthesis:

MIN(A,2*(3+B) ------deletion error

ii. Extraneous comma:

int p=1,000; -------insertion error

iii. Colon in place of semicolon:

I=9;

J=7; **------**replacement error

iv. Misspelled keyword

F:PORCEDURE OPTIONS(MAIN) ------transposition error

v. Extra blank

/* comment */ ------insertion error

Also it is difficult to determine exactly how many errors there are or where they have occurred without knowing the intent of the programmer,

2. Semantic errors

Semantic errors can be detected both at compile time and at run time.

The most common semantic errors that can be detected at compile time are:

- Errors of declaration and scope
- Undeclared or multiply declared identifiers
- Type incompatibilities between operators and operands and between formal and actual parameters.

Example: the amount of type checking that can be done depends on the language. Some languages have only one data type, so data type incompatibility is there, such languages are known as strongly typed. Some languages such as PL/I provide automatic type conversions between operators and operands and different data types. It eliminates the possibility of general semantic error detection.

3. Dynamic errors

In some languages, certain kinds of errors can only be detected at run time.

Ex. In languages like C++, divide by zero exception is generated which is an example of dynamic error.

Another common kind of range checking for certain values that is array subscripts and case statement selectors.

A subscript out of range could cause an arbitrary memory location to be overwritten.

An arbitrary value in a case statement could cause a jump to an unknown memory location.

- **Code optimization**

Code optimization refers to techniques a compiler can employ in an attempt to produce a better object program than the most obvious for a given source program.

The quality of an object program is generally measured by its size or its running time.

The optimized code executes faster, efficient memory usage and yields better performance

- **The Principal sources of optimization**

Code optimization techniques are generally applied after syntax analysis usually both before and during code generation.

These techniques consist

Detecting patterns in the program

Replacing these patterns by equivalent but more efficient constructs.

These patterns may be local or global and the replacement strategy may be machine dependent or machine independent.

i) Inner loops

This is the most obvious target for optimization is handling inner loops.

Ex. Removal of Loop invariant computations, elimination of induction variables etc.

ii) Language implementation details inaccessible to the user

Some programming languages allow a programmer to write source programs that get compiled into efficient object programs. In these languages the optimizations can be done by the programmer at the source level.

Other languages may not permit certain kinds of optimizations to be specified at the source level.

Example: In FORTRAN and similar languages, the array references are made by indexing rather than by pointer or address calculation. This prevents the programmer from address calculations in arrays.

iii) Further optimization techniques

Additional important sources of optimization are:

1. Dead code elimination: removes code that does not affect the program. Dead code is a code that is never executed or that does nothing useful.

```
a=5,b=10
for i=1 to 100
{
    x=b+i
    z=a+5
}
```

Before optimization

```
a=5,b=10
z=a+5;
for i=1 to 100
{
    x=b+i
}
```

After optimization

2. Common subexpression elimination

Searches for instances of identical expressions and replace them with a single variable.

```
a=b*c+g
d=b*c+d
```

Before optimization

```
temp=b*c
a=temp+g
d=temp+d
```

After optimization

3. Induction variable elimination

An induction variable is a variable that gets increased or decreased by a fixed amount on each iteration of a loop.

```
for(i=0;i<10;++i)
{
    j=17*i;
}
```

Here i and j are induction variables.

4. Reduction in strength: replaces an expensive expression by a cheaper one.

Example: X=2*2

X=2+2

Here * takes more time for execution than +.

5. Constant folding is the process of simplifying constant expressions at compile time

Ex. Replacing expressions consisting of constants (e.g. "3+5") with their final value("8") at compile time.

6. Loop interchange

for i=1 to 10	for i=1 to 20
for j=1 to 20	for j=1 to 10
A[i,j]=i+j;	A[i,j]=i+j;
Before optimization	**After optimization**