Lecture 8 | Symbol Table

- A variable should not be declared more than once in the same scope.
- A variable should not be used before being declared.
- The type of the left-hand side of an assignment should match the type of the right-hand side.
- Methods should be called with the right number and types of arguments.

Symbol Tables

The purpose of the symbol table is to keep track of names declared in the program. This includes names of classes, fields, methods, and variables. Each symbol table entry associates a set of attributes with one name; for example:

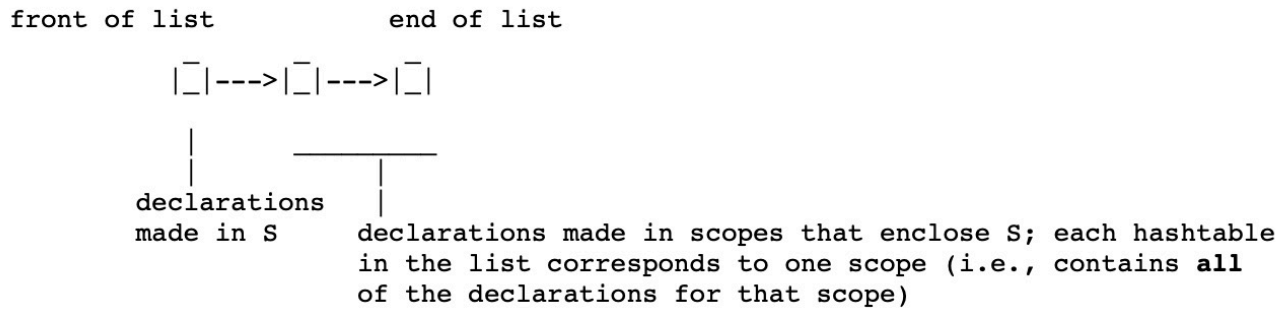
- which kind of name it is
- what is its type
- what is its nesting level
- where will it be found at runtime.

One factor that will influence the design of the symbol table is what scoping rules are defined for the language being compiled.

Symbol table implementations

Method 1: List of Hashtables.

The idea behind this approach is that the symbol table consists of a list of hashtables, one for each currently visible scope. When processing a scope S, the structure of the symbol table is:

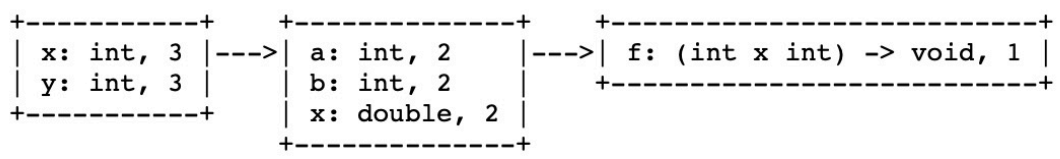


For example, given this code:

```
void f(int a, int b) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



no. of scopes = global scopes (=1) + no. of {}

The declaration of method g has not yet been processed, so it has no symbol-table entry yet. Note that because f is a method, its type includes the types of its parameters (int x int), and its return type (void).

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number and add a new empty hashtable to the front of the list.
2. To process a declaration of x: look up x in the first table in the list. If it is there, then issue a "multiply declared variable" error; otherwise, add x to the first table in the list.
3. To process a use of x: look up x starting in the first table in the list; if it is not there, then look up x in each successive table in the list. If it is not in *any* table then issue an "undeclared variable" error.
4. On scope exit, remove the first table from the list and decrement the current level number.

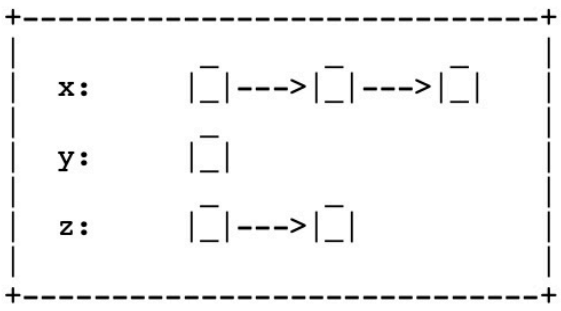
Remember that method names need to go into the hashtable for the outermost scope (not into the same table as the method's variables). For example, in the picture above, method name f is in the symbol table for the outermost scope; name f is *not* in the same scope as parameters a and b, and variable x. This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table.

Here are the times required for each operation:

1. **Scope entry**: time to initialize a new, empty hashtable; this is probably proportional to the size of the hashtable.
2. **Process a declaration:** using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined.
4. **Scope exit**: time to remove a table from the list, which should be O(1) if garbage collection is ignored.

Method #2: Hashtable of Lists

The idea behind this approach is that when processing a scope S, the structure of the symbol table is:



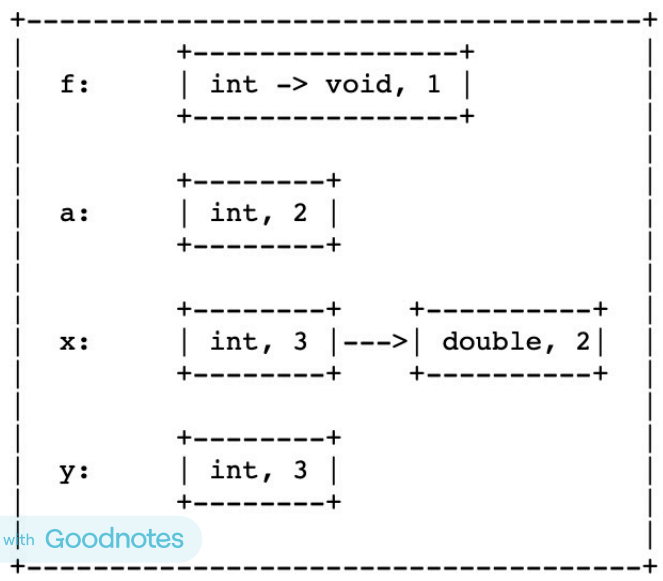
There is just one big hashtable, containing an entry for each variable for which there is some declaration in scope S or in a scope that encloses S. Associated with each variable is a list of symbol-table entries. The first list item corresponds to the most closely enclosing declaration; the other list items correspond to declarations in enclosing scopes.

For example, given this code:

```
void f(int a) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



Note that the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made in the current scope or in an enclosing scope.

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number.
2. To process a declaration of x: look up x in the symbol table. If x is there, fetch the level number from the first list item. If that level number = the current level then issue a "multiply declared variable" error; otherwise, add a new item to the front of the list with the appropriate type and the current level number.
3. To process a use of x: look up x in the symbol table. If it is not there, then issue an "undeclared variable" error.
4. On scope exit, scan all entries in the symbol table, looking at the first item on each list. If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry). Finally, decrement the current level number.

The required times for each operation are:

1. **Scope entry**: time to increment the level number, O(1).
2. **Process a declaration**: using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing, constant expected time (O(1)).
4. **Scope exit**: time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable buckets).