Symbol Tables and Static Checks

Contents

Introduction

The parser ensures that the input program is syntactically correct, but there are other kinds of correctness that it cannot (or usually does not) enforce. For example:

- 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.

The next phase of the compiler after the parser, sometimes called the **static semantic analyzer** is in charge of checking for these kinds of errors. The checks can be done in two phases, each of which involves traversing the abstract-syntax tree created by the parser:

1. For each scope in the program: Process the declarations, adding new entries to the symbol table and reporting any variables that are multiply declared; process the statements, finding uses of undeclared variables, and updating the "ID" nodes of the abstract-syntax tree to point to the appropriate symbol-table entry.
2. Process all of the statements in the program again, using the symbol-table information to determine the type of each expression, and finding type errors.

Below, we will consider how to build symbol tables and how to use them to find multiply-declared and undeclared variables. We will then consider type checking.

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. Let's consider some different kinds of scoping rules before continuing our discussion of symbol tables.

Scoping

In most languages, the same name can be declared multiple times if the declarations occur in different scopes, and/or involve different kinds of names. For example, in Java you can use the same name for a class, a field of the class, a method of the class, and a local variable of the method (this is not recommended, but it is legal):

```
class Test {
  int Test;

  void Test(  ) {
      double Test;  // could also be declared int
  }
}
```

In both Java and C++ (but not in Pascal or C), you can use the same name for more than one method as long as the number and/or types of parameters are unique.

In Java you *cannot* declare a variable x in a method if there is also a parameter named x, or another variable named x declared in an enclosing block or *for* loop. However, such declarations *are* allowed in C++. For example, the following is a legal C++ function, but not a legal Java method:

```
void f( int k ) {   // k is a parameter
   int k = 0;       // also a local variable

   while (...) {
      int k = 1;   // and another local variable, inside the loop
      ...
   }
}
```

In general, the scope rules of a language determine which declaration of a named object corresponds to each use. C++ and Java use what is called **static scoping**; that means that the correspondence between uses and declarations is made at compile time. C++ uses the "most closely nested" rule to match nested declarations to their uses: a use of variable x matches the declaration in the most closely enclosing scope such that the declaration precedes the use. In C++, there is one, outermost scope that includes all function names and the names of the global variables (the variables that are declared outside the functions). Each function has two or more scopes: one for the parameters, one for the function body, and possibly additional scopes for each *for* loop and each nested block (delimited by curly braces) in the function.

In the example given above, the outermost scope includes just the name "f", and function f itself has three (nested) scopes:

1. The outer scope for f just includes parameter k.
2. The next scope is for the body of f, and includes the variable k that is initialized to 0.
3. The innermost scope is for the body of the while loop, and includes the variable k that is initialized to 1.

So a use of variable k inside the while loop matches the declaration in the loop (has the value 1), while a use of k outside the loop (either before or after the loop) matches the declaration at the beginning of the function (has the value 0).

TEST YOURSELF #1

Question 1: Consider the names declared in the following code. For each, determine whether it is legal according to the rules used in Java.

```
class animal {
  // methods
  void attack(int animal) {
     for (int animal=0; animal<10; animal++) {
         int attack;
     }
  }

  int attack(int x) {
     for (int attack=0; attack<10; attack++) {
        int animal;
     }
  }

  void animal() { }

  // fields
  double attack;
  int attack;
  int animal;
}
```

Question 2: Consider the following C++ code. For each use of a name, determine which declaration it corresponds to (or whether it is a use of an undeclared name).

```
int k=10, x=20;

void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
        int x;
        if (x == k) {
           int k, y;
           k = y = x;
        }
        if (x == k) {
           int x = y;
        }
    }
}
```

Not all languages use static scoping. Lisp, APL, and Snobol use what is called **dynamic** scoping. A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the **most-recently-called still active** function. For example, consider the following code:

```
void main() {
  f1();
  f2();
}

void f1() {
  int x = 10;
  g();
}
```

```
void f2() {
  String x = "hello";
  f3();
  g();
}

void f3() {
  double x = 30.5;
}

void g() {
  print(x);
}
```

Under dynamic scoping this program outputs "10 hello". The first call to g comes from f1, whose copy of x has value 10. The next call to g comes from f2. Although f3 is called by f2 before it calls g, the call to f3 is not active when g is called; therefore, the use of x in g matches the declaration in f2, and "hello" is printed.

TEST YOURSELF #2

Assuming that dynamic scoping is used, what is output by the following program?

```
void main() {
  int x = 0;
  f1();
  g();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  int x = 20;
  f1();
  g();
}

void g() {
  print(x);
}
```

It is generally agreed that dynamic scoping is a bad idea; it can make a program very difficult to understand, because a single use of a variable can correspond to many different declarations (with different types)! The languages that use dynamic scoping are all old languages; recently designed languages all use static scoping.

Another issue that is handled differently by different languages is whether names can be used before they are defined. For example, in Java, a method or field name *can* be used before the definition appears, but this is *not* true for a variable:

```
class Test {
  void f() {
    val = 0;   // field val has not yet been declared -- OK
    g();       // method g has not yet been declared  -- OK
    x = 1;     // variable x has not yet been declared -- ERROR!
    int x;
  }
```

```
  void g() {}
  int val;
}
```

In what follows, we will assume that we are dealing with a language that:

- uses static scoping
- requires that *all* names be declared before they are used
- does not allow multiple declarations of a name in the same scope (even for different kinds of names)
- *does* allow the same name to be declared in multiple nested scopes (but only once per scope)
- uses the same scope for a method's parameters and for the local variables declared at the beginning of the method

Symbol Table Implementations

In addition to the assumptions listed at the end of the previous section, we will assume that:

- The symbol table will be used to answer two questions:
    1. Given a declaration of a name, is there already a declaration of the same name in the current scope (i.e., is it multiply declared)?
    2. Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?
    3. The symbol table is only needed to answer those two questions (i.e., once all declarations have been processed to build the symbol table, and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry, the symbol table itself is no longer needed).

Given these assumptions, the symbol-table operations we will need are:

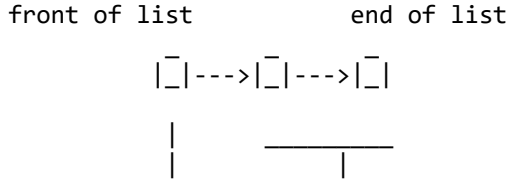
1. Look up a name in the current scope only (to check if it is multiply declared).
2. Look up a name in the current and enclosing scopes (to check for a use of an undeclared name, and to link a use with the corresponding symbol-table entry).
3. Insert a new name into the symbol table with its attributes.
4. Do what must be done when a new scope is entered.
5. Do what must be done when a scope is exited.

We will look at two ways to design a symbol table: a list of tables, and a table of lists. For each approach, we will consider what must be done when entering and exiting a scope, when processing a declaration, and when processing a use. To keep things simple, we will assume that each symbol-table entry includes only:

- the symbol name
- its type
- the nesting level of its declaration

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:



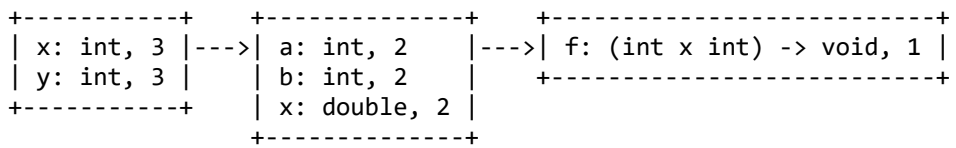
```
declarations   |
made in S      declarations made in scopes that enclose S; each hashtable
               in the list corresponds to one scope (i.e., contains all
               of the declarations for that scope)
```

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:



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.

TEST YOURSELF #3

For all three questions below, assume that the symbol table is implemented using a list of hashtables.

Question 1: Recall that Java does not allow the same name to be used for a local variable of a method, and for another local variable declared inside a nested scope in the method body. Even with this restriction, it is not a

good idea to put *all* of a method's local variables (whether they are declared at the beginning of the method, or in some nested scope within the method body) in the *same* table. Why not?

Question 2: C++ does not use exactly the scoping rules that we have been assuming. In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name (and any uses of the name refer to the local variable).

Consider the following code. Draw the symbol table as it would be after processing the declarations in the body of f under:

- the scoping rules we have been assuming
- C++ scoping rules

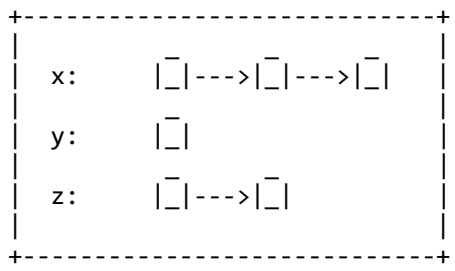
```
void g(int x, int a) { }

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

Question 3: Which of the four operations (scope entry, process a declaration, process a use, scope exit) described above would change (and how would it change) if Java rules for name reuse were used instead of C++ rules (i.e., if the same name can be used within one scope as long as the uses are for different kinds of names, and if the same name *cannot* be used for more than one variable declaration in nested scopes)?

Method 2: Hashtable of Lists

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



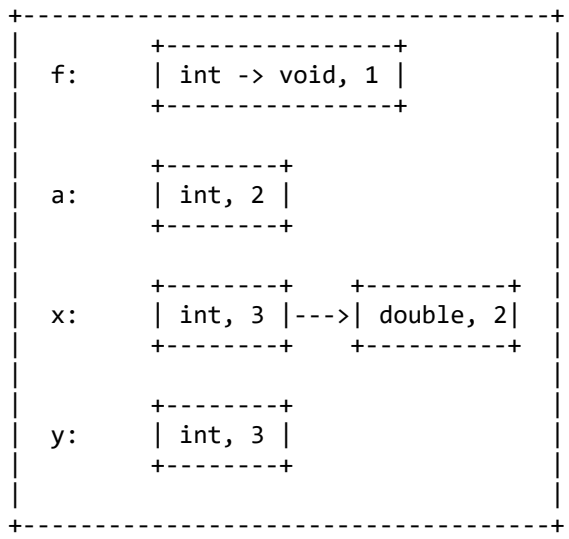
There is just one big hashtable, containing an entry for each variable for which there is some declaration in scope S or in a scope that encloses S. Associated with each variable is a list of symbol-table entries. The first list item corresponds to the most closely enclosing declaration; the other list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

TEST YOURSELF #4

Assume that the symbol table is implemented using a hashtable of lists. Draw pictures to show how the symbol table changes as each declaration in the following code is processed.

```
void g(int x, int a) {
   double d;
   while (...) {
      int d, w;
      double x, b;
      if (...) {
         int a,b,c;
      }
   }
```

```
        while (...) {
           int x,y,z;
        }
     }
```

Type Checking

As mentioned in the Introduction, the job of the type-checking phase is to:

- Determine the type of each expression in the program (each node in the AST that corresponds to an expression).
- Find type errors.

The **type rules** of a language define how to determine expression types, and what is considered to be an error. The type rules specify, for every operator (including assignment), what types the operands can have, and what is the type of the result. For example, both C++ and Java allow the addition of an int and a double, and the result is of type double. However, while C++ also allows a value of type double to be assigned to a variable of type int, Java considers that an error.

TEST YOURSELF #5

List as many of the operators that can be used in a Java program as you can think of (don't forget to think about the logical and relational operators as well as the arithmetic ones). For each operator, say what types the operands may have, and what is the type of the result.

In addition to finding type errors caused by operators being applied to operands of the wrong type, the type checker must also find type errors having to do with expressions that, because of their **context** must be boolean, and type errors having to do with method calls. Examples of the first kind of error include:

- the condition of an *if* statement
- the condition of a *while* loop
- the termination condition part of a *for* loop

and examples of the second kind of error include:

- calling something that is not a method
- calling a method with the wrong number of arguments
- calling a method with arguments of the wrong types