Symbol Tables and Static Checks

Lecture 14

Outline

- How to build symbol tables
- How to use them to find
  - multiply-declared and
  - undeclared variables.
- How to perform type checking

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
    - e.g.: main$ ();
- Parsing
  - Detects inputs with ill-formed parse trees
    - e.g.: missing semicolons
- Semantic analysis
  - Last "front end" phase
  - Catches all remaining errors

Introduction

- typical semantic errors:
  - multiple declarations: a variable should be declared (in the same scope) at most once
  - undeclared variable: a variable should not be used before being declared.
  - type mismatch: type of the left-hand side of an assignment should match the type of the right-hand side.
  - wrong arguments: methods should be called with the right number and types of arguments.

A sample semantic analyzer

- works in two phases
  - i.e., it traverses the AST created by the parser:

1. For each scope in the program:
   - **process the declarations** =
     - add new entries to the symbol table and
     - report any variables that are multiply declared
   - **process the statements** =
     - find uses of undeclared variables, and
     - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.
2. Process all of the statements in the program again,
   - use the symbol-table information to determine the type of each expression, and to find type errors.

Symbol Table = set of entries

- purpose:
  - keep track of names declared in the program
  - names of
    - variables, classes, fields, methods,
- symbol table entry:
  - associates a name with a set of attributes, e.g.:
    - kind of name (variable, class, field, method, etc)
    - type (int, float, etc)
    - nesting level
    - memory location (i.e., where will it be found at runtime).

Scoping

- symbol table design influenced by what kind of scoping is used by the compiled language
- In most languages, the same name can be declared multiple times
  - if its declarations occur in different scopes, and/or
  - involve different kinds of names.

Scoping: example

- Java: can use same name for
  - a class,
  - field of the class,
  - a method of the class, and
  - a local variable of the method
- ***legal Java program:***

```
class Test {
   int Test;
   Test( ) { double Test; }
}
```


Scoping: overloading

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

```
int add(int a, int b);
float add(float a, float b);
```


Scoping: general rules

- The scope rules of a language:
  - determine which declaration of a named object corresponds to each use of the object.
  - i.e., scoping rules map uses of objects to their declarations.
- C++ and Java use ***static scoping***.
  - mapping from uses to declarations is made at compile time.
  - C++ uses the "most closely nested" rule
    - a use of variable x matches the declaration in the most closely enclosing scope such that the declaration precedes the use.
    - a deeply nested variable x hides x declared in an outer scope.
  - in Java:
    - inner scopes cannot define variables defined in outer scopes

Scope levels

- Each function has two or more scopes:
  - one for the parameters,
  - one for the function body,
  - and possibly additional scopes in the function
    - for each *for* loop and
    - each nested block (delimited by curly braces)

Example (assume C++ rules)

```
void f( int k ) {          // k is a parameter
   int k = 0;              // also a local variable (not legal in Java)
   while (k) {
        int k = 1;         // another local var, in a loop (not ok in Java)
   }
}
```

- the outmost 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.

TEST YOURSELF #1

- This is a C++ program. Match each use to its declaration, or say why it is a use of an undeclared variable.

```
class Foo {
   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; }
      }
   }}
```


Dynamic scoping

- Not all languages use static scoping.
- Lisp, APL, and Snobol use **dynamic** scoping.
- 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.

Example

- 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); }
```


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); }
```


Static vs dynamic scoping

- generally, dynamic scoping is a bad idea
  - can make a program difficult to understand
  - a single use of a variable can correspond to
    - many different declarations
    - with different types!

Used before declared?

- can a name be used before they are defined?
  - Java: a method or field name *can* be used before the definition appears,
    - *not* true for a variable!

Example

```
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;
      // var x has not yet been declared -- ERROR!
      int x;
   }
   void g() {}
   int val;
}
```


Simplification

- From now on, assume that our language:
  - 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
- Rules in PA4/5 may differ slightly!

Symbol Table Implementations

- In addition to the above simplification, 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?

Note

- 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,
  - then the symbol table itself is no longer needed
    - because no more lookups based on name will be performed

What operation do we need?

- Given the above assumptions, we will need:

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.

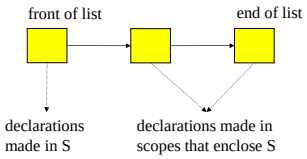
Two possible symbol table implementations

1. a list of tables
2. 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.
- Simplification:
  - assume each symbol-table entry includes only:
    - the symbol name
    - its type
    - the nesting level of its declaration

Method 1: List of Hashtables

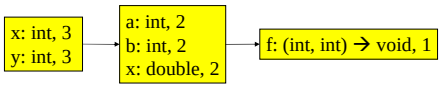
- The idea:
  - symbol table = a list of hashtables,
  - one hashtable for each currently visible scope.
- When processing a scope S:



Example:

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

- After processing declarations inside the while loop:



List of hashtables: the operations

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.

… continued

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

- method names belong into the hashtable for the outermost scope
  - not into the same table as the method's variables
- For example, in the example 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.

The running times for each operation:

1. **Scope entry**:
   - time to initialize a new, empty hashtable;
   - 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 #1

- **Question 1:** 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
    - 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

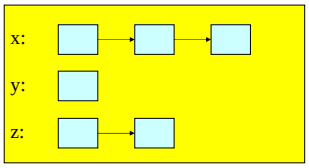
```
void g(int x, int a) { }
void f(int x, int y, int z) { int a, b, x; ... }
```


… continued

- **Question 2:**
  - Which of the four operations described above
    - scope entry,
    - process a declaration,
    - process a use,
    - scope exit
  - would change (and how) if the following rules for name reuse were used instead of C++ rules:
    - the same name can be used within one scope as long as the uses are for different kinds of names, and
    - the same name *cannot* be used for more than one variable declaration in a nested scope

Method 2: Hashtable of Lists

- the idea:
  - when processing a scope S, the structure of the symbol table is:



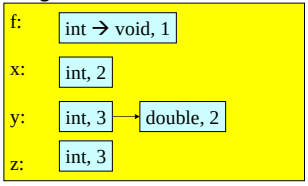
Definition

- 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.

Example

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

- After processing the declarations inside the while loop:



Nesting level information is crutial

- 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.

Hashtable of lists: the operations

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.

… continue

1. To process a use of x:
   - look up x in the symbol table.
   - If it is not there, then issue an "undeclared variable" error.
2. 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.

Running times

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 #2

- 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

- 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.

Example

- both C++ and Java allow the addition of an int and a double, and the result is of type double.
- However,
  - C++ also allows a value of type double to be assigned to a variable of type int,
  - Java considers that an error.

TEST YOURSELF #3

- 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.

Other type errors

- the type checker must also
  1. find type errors having to do with the **context** of expressions,
     - e.g., the context of some operators must be boolean,
  2. type errors having to do with method calls.
- Examples of the context errors:
  - the condition of an *if* statement
  - the condition of a *while* loop
  - the termination condition part of a *for* loop
- Examples of method errors:
  - 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