Semantic Analysis

- Scoping (Readings 7.1,7.4,7.6)
  - Static
  - Dynamic
- Parameter passing methods (7.5)
- Building symbol tables (7.6)
  - How to use them to find
    - multiply-declared and
    - undeclared variables
- Type checking (6.1-6.6)

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

Classes of Errors

- Lexical – detected by scanner
  - Examples:
    - Illegal character in input
    - Illegal comments
    - Unterminated string constants
- Syntactic – detected by the parser
  - Input is not a legal program, because it cannot be parsed by CFG
  - Example: a = * 5

Classes of Errors

- Static Semantic - can be detected in parser or in separate semantic analysis passes
  - Input can be parsed by CFG but some non context-free error
  - Examples
    - Multiply declared variables
    - Undeclared variables
    - Function call with wrong number of arguments
    - Type mismatches

Classes of Errors

- Semantic - may be detected at compile time, may also add checks in code for runtime
  - Examples
    - Division by 0
    - Array index out-of-bounds
    - Dereference of NULL pointer
  - Can include code to check for conditions and not allow them to occur
  - Advantage: error message instead of error behavior
  - Disadvantage: object code is longer and slower

Classes of Errors

- Semantic - may be detected at compile time, may also add checks in code for runtime
  - Examples
    - Division by 0
    - Array index out-of-bounds
    - Dereference of NULL pointer
  - Can include code to check for conditions and not allow them to occur
  - Advantage: error message instead of error behavior
  - Disadvantage: object code is longer and slower

Classes of Errors

- Logical – hardest to detect
  - Program is syntactically and semantically correct but does not do the “correct” thing
  - Compiler can sometimes detect problem things

```
if (x = 0) result = 1;
result = 2;
```

  - Assignment in the body of the if is “useless” – immediately overwritten by next assignment

Static 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

An 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 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;
   void 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

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

```
void f( int k ) {        // k is a parameter
   int k = 0;            // also a local variable
   while (k) {
        int k = 1;       // another local variable, in a loop
   }
}
```

- 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

Example

- Match uses and declarations

```
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); }
```

Example: Dynamic Scoping

- Assuming dynamic scoping is used, what is output by this 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); }
```

Scoping Comparison

- 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!
- 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;
}
```