Lecture 10: Static Semantics Overview

Administrivia

• If you’re interested, there’s a workshop on writing just-in-time compilers using LLVM on Oct. 2nd in the Woz, from 8-10pm.

• First in-class test 10 October.
Overview

- **Lexical analysis**
  - Produces tokens
  - Detects & eliminates illegal tokens

- **Parsing**
  - Produces trees
  - Detects & eliminates ill-formed parse trees

- **Static semantic analysis**
  - Produces *decorated tree* with additional information attached
  - Detects & eliminates remaining static errors
Static vs. Dynamic

• We use the term static to describe properties that the compiler can determine without considering any particular execution.
  
  - E.g., in
    
    ```python
def f(x) : x + 1
```
    
    Both uses of x refer to same variable

• Dynamic properties are those that depend on particular executions in general.
  
  - E.g., will x = x/y cause an arithmetic exception?

• Actually, distinction is not that simple. E.g., after
  
  ```python
x = 3
y = x + 2
```

  compiler could deduce that x and y are integers.

• But languages often designed to require that we treat variables only according to explicitly declared types, because deductions are difficult or impossible in general.
Typical Tasks of the Semantic Analyzer

- Find the declaration that defines each identifier instance
- Determine the static types of expressions
- Perform re-organizations of the AST that were inconvenient in parser, or required semantic information
- Detect errors and fix to allow further processing
Typical Semantic Errors: Java, C++

- Multiple declarations: a variable should be declared (in the same region) at most once.
- Undeclared variable: a variable should not be used without being declared.
- Type mismatch: e.g., 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.
- Definite-assignment check (Java): conservative check that simple variables assigned to before use.
Output from Static Semantic Analysis

Input is AST; output is an annotated tree: identifiers decorated with declarations, other expressions with type information.

x = 3
def f (x):
    return x+y
y = 2

---

Id | Type  | Nesting
---|-------|--------
#1: x, Any, 0
#2: f, Any->Any, 0
#3: x, Any, 1
#4: y, Any, 0
Output from Static Semantic Analysis (II)

- Analysis has added objects we’ll call *symbol entries* to hold information about instances of identifiers.

- In this example, #1: x, Any, 0 denotes an entry for something named ‘x’ occurring at the outer lexical level (level 0) and having static type Any.

- For other expressions, we annotate with static type information.
Output from Static Semantic Analysis: Classes

• In Python (dynamically typed), can write

```python
class A(object):
    def f(self): return self.x
```

```python
a1 = A(); a2 = A()  # Create two As
a1.x = 3; print a1.x # OK
print a2.x            # Error; there is no x
```

so can’t say much about attributes (fields) of A.

• In Java, C, C++ (statically typed), analogous program is illegal, even without second print (the class definition itself is illegal).

• So in statically typed languages, symbol entries for classes would contain dictionaries mapping attribute names to types.
Scope Rules: Binding Names to Symbol Entries

- **Scope of a declaration**: section of text or program execution in which declaration applies

- **Declarative region**: section of text or program execution that bounds scopes of declarations (we’ll say “region” for short).

- If scope of a declaration defined entirely according to its position in source text of a program, we say language is *statically scoped*.

- If scope of a declaration depends on what statements get executed during a particular run of the program, we say language has *dynamically scoped*. 
Scope Rules: Name $\Rightarrow$ Declaration is Many-to-One

- In most languages, can declare the same name multiple times, if its declarations
  - occur in different declarative regions, or
  - involve different kinds of names.
  - Examples from Java?, C++?
Scope Rules: Nesting

• Most statically scoped languages (including C, C++, Java) use:

  *Algol scope rule:* Where multiple declarations might apply, choose the one defined in the *innermost* (most deeply nested) declarative region.

• Often expressed as “inner declarations hide outer ones.”

• Variations on this: Java disallows attempts to hide local variables and parameters.
Scope Rules: Declarative Regions

• Languages differ in their definitions of declarative regions.

• In Java, variable declaration’s effect stops at the closing ‘}’, that is, each function body is a declarative region.

• What others?

• In Python, function header and body make up a declarative region, as does a lambda expression. But nothing smaller. Just one x in this program:

    def f(x):
        x = 3
        L = [x for x in xrange(0,10)]
Scope Rules: Use Before Definition

- Languages have taken various decisions on where scopes start.
- In Java, C++, scope of a member (field or method) includes the entire class (textual uses may precede declaration).
- But scope of a local variable starts at its declaration.
- As for non-member and class declarations in C++: must write

```cpp
extern int f(int); // Forward declarations
class C;
int x = f(3)        // Would be illegal w/o forward decls.
void g(C* x) {
    ...
}

int f(int x) { ... } // Full definitions
class C { ... }
```
Scope Rules: Overloading

• In Java or C++ (not Python or C), can use the same name for more than one method, as long as the number or types of parameters are unique.

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

• The declaration applies to the signature—name + argument types—not just name.

• But return type not part of signature, so this won’t work:

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

• In Ada, it will, because the return type is part of signature.
Dynamic Scoping

- Original Lisp, APL, Snobol use *dynamic scoping*, rather than static:
  
  Use of a variable refers to most recently executed, and still active, declaration of that variable.

- Makes static determination of declaration generally impossible.

- Example:

  ```
  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); }
  ```

- With static scoping, illegal.

- With dynamic scoping, prints "10" and "hello"
Explicit vs. Implicit Declaration

- Java, C++ require explicit declarations of things.
- C is lenient: if you write \texttt{foo(3)} with no declaration of \texttt{foo} in scope, C will supply one.
- Python implicitly declares variables you assign to in a function to be local variables.
- Fortran implicitly declares any variables you use, and gives them a type depending on their first letter.
- But in all these cases, there is a declaration as far as the compiler is concerned.
So How Do We Annotate with Declarations?

- Idea is to recursively navigate the AST,
  - in effect executing the program in simplified fashion,
  - extracting information that isn’t data dependent.
- You saw it in CS61A (sort of).
Environment Diagrams and Symbol Entries

• In Scheme, executing

```
(set! x 7)
(define (f x) (let ((y (+ x 39))) (+ x y)))
(f 3)
```

would eventually give this environment at (+ x y):

```
global environment
          x: 7
          f: ...
          x: 3
          y: 42
current environment
```

• Now abstract away values in favor of static type info:

```
#1. x: Any
#2. f: Any→Any
#3. x: Any
#4. y: Any
```

• and voila! A data structure for mapping names to current declarations: a block-structured symbol table.