Lecture 14: Static Semantics Overview[1]

Administrivia

- First in-class test 13 March.

[1] From material by R. Bodik and P. Hilfinger

Overview

- Lexical analysis
  - Produces tokens
  - Detects & eliminates illegal tokens
- Parsing
  - Produces trees
  - Detects & eliminates ill-formed parse trees
- Static semantic analysis $\Longleftarrow$ *we are here*
  - 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

    ```
    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

  ```
  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

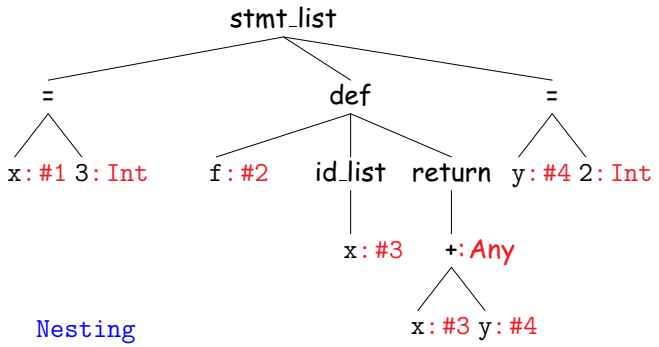
Typical Semantic Errors: Java, C++

- 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. Actually subset of type mismatch.
- 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
```



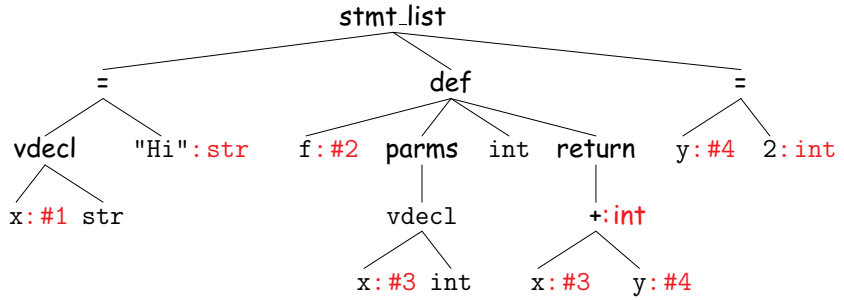

```
    Id  Type     Nesting
#1: x,  Any,       0
#2: f,  Any->Any,  0
#3: x,  Any,       1
#4: y,  Any,       0
```


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.
- These symbol entry decorations might be attached directly to the AST or stored separately in symbol tables and looked up: it's all a matter of representation.

Output from Static Semantic Analysis for Chocopy

Chocopy (like Java, C++) is *statically typed*, so we can have more specific information in symbols.




```
x: str = "Hi"
y: int = 2
def f(x: int) -> int:
   return x+y
```

```
    Id  Type     Nesting
#1: x,  str,       0
#2: f,  int->int,  0
#3: x,  int,       1
#4: y,  int,       0
```

Output from Static Semantic Analysis: Classes

- In Python (dynamically typed), can write

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

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). (Others use the term "scope" for what I'm calling a declarative region. I use a separate term, since I think it is a distinct concept.)
- 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⟹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* (or *shadow*) 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, modules, function headers and their bodies, lambda expressions, comprehensions (of lists, sets, and dictionaries) and generator expressions make up declarative regions, but nothing smaller. Just one `x` in this program:

```
def f(x):
   x = 3
   for x in range(6):
      print(x)
   print(x)
```

It prints 0–5 and then 5 again.