Lecture 15: Static Semantics: Scope and Type[1]

[1] From material by G. Necula and P. Hilfinger

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

```
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 `foo(3)` with no declaration of `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?

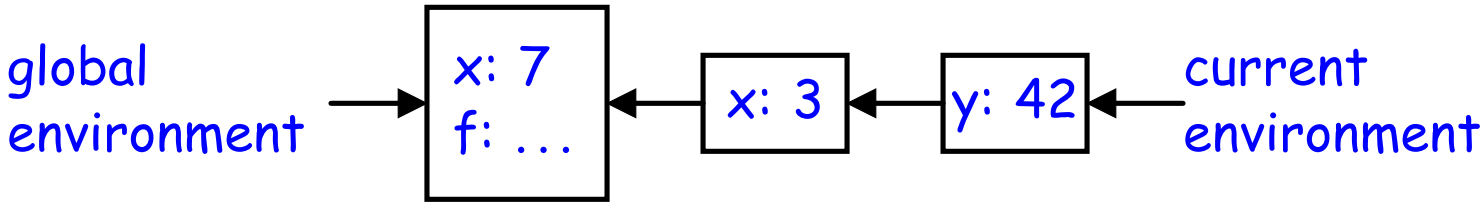
- Idea is to recursively navigate the AST,
  - in effect executing the program in simplified fashion,
  - extracting information that isn't data dependent.
- You saw it in CS61A (sort of).

Environment Diagrams and Symbol Entries

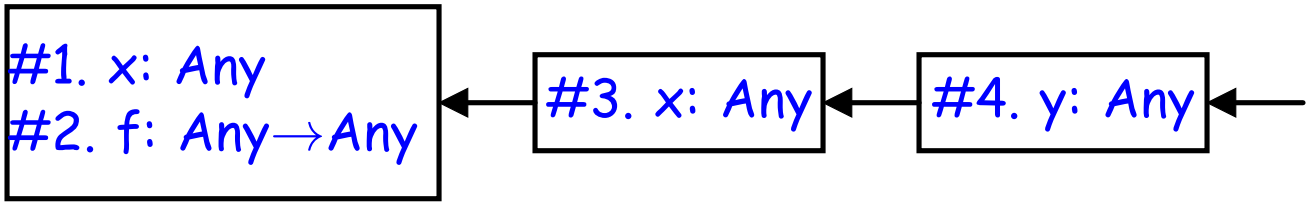
- In Scheme, executing

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

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



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



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

Type Checking Phase

- Determines the type of each expression in the program, (each node in the AST that corresponds to an expression)
- Finds type errors.
  - Examples?
- The *type rules* of a language define each expression's type and the types required of all expressions and subexpressions.

Types and Type Systems

- A type is a set of *values* together with a set of *operations* on those values.
- E.g., fields and methods of a Java class are meant to correspond to values and operations.
- A language's *type system* specifies which operations are valid for which types.
- Goal of type checking is to ensure that operations are used with the correct types, enforcing intended interpretation of values.
- Notion of "correctness" often depends on what programmer has in mind, rather than what the representation would allow.
- Most operations are legal only for values of some types
  - Doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation:

```
movl y, %eax;  addl x, %eax
```

Uses of Types

- Detect errors:
  - Memory errors, such as attempting to use an integer as a pointer.
  - Violations of abstraction boundaries, such as using a private field from outside a class.
- Help compilation:
  - When the Python compiler sees `x+y`, the *static* part of its type systems tells it almost nothing about types of `x` and `y`, so code must be general.
  - But during execution, the *dynamic part* of its type system, implemented by type information in the data structures, tells it what code to execute.
  - In C, C++, Java, code sequences for `x+y` are smaller and faster, because representations are known without runtime checks of type information.

Review: Dynamic vs. Static Types

- A *dynamic type* attaches to an object reference or other value. It's a run-time notion, applicable to any language.
- The *static type* of an expression or variable is a constraint on the possible dynamic types of its value, enforced at compile time.
- Language is *statically typed* if it enforces a "significant" set of static type constraints.
  - A matter of degree: assembly language might enforce constraint that "all registers contain 32-bit words," but since this allows just about any operation, not considered static typing.
  - C sort of has static typing, but rather easy to evade in practice.
  - Java's enforcement is pretty strict.
- In early type systems, $\text{dynamic\_type}(\mathcal{E}) = \text{static\_type}(\mathcal{E})$ for all expressions $\mathcal{E}$, so that in all executions, $\mathcal{E}$ evaluates to exactly type of value deduced by the compiler.
- Gets more complex in advanced type systems with subtyping.

Subtyping

- Define a relation $X \preceq Y$ on classes to say that:

  An object (value) of type X could be used when one of type Y is acceptable

  or equivalently

  X conforms to Y

- In Java this means that X extends Y.
- Properties:
  - $X \preceq X$
  - $X \preceq Y$ if X inherits from Y.
  - $X \preceq Z$ if $X \preceq Y$ and $Y \preceq Z$.

Example

```
class A { ... }
class B extends A { ... }
class Main {
   void f () {
      A x;           // x has static type A.
      x = new A();   // x's value has dynamic type A.
      ...
      x = new B();   // x's value has dynamic type B.
      ...
   }
}
```

Variables, with static type *A* can hold values with dynamic type $\preceq$ *A*, or in general...

Type Soundness

Soundness Theorem on Expressions.

$$\forall E.\ \text{dynamic\_type}(E) \preceq \text{static\_type}(E)$$

- Compiler uses static_type(E) (call this type C).
- All operations that are valid on C are also valid on values with types $\preceq C$ (e.g., attribute (field) accesses, method calls).
- Subclasses only add attributes.
- Methods may be overridden, but only with same (or compatible) signature.

Typing Options

- *Statically typed:* almost all type checking occurs at compilation time (C, Java). Static type system is typically rich.
- *Dynamically typed:* almost all type checking occurs at program execution (Scheme, Python, Javascript, Ruby). Static type system can be trivial.
- *Untyped:* no type checking. What we might think of as type errors show up either as weird results or as various runtime exceptions.