and Computer Sciences
Computer Science Division

CS 164
Fall 2006

P. N. Hilfinger

Static Analysis: Scope and Types

1 Terminology

Programs, in general, are simply collections of definitions of terms, which we often call *declarations*[1]. Each declaration may *use* other declarations, referring to them by the names they *introduce.* In programming languages, the *scope of a declaration* that introduces some name is the portion of the program in which the meaning (or a possible meaning) of that name is the one given by the declaration. Many authors (from less distinguished institutions) refer loosely to the scope of a *name* as opposed to the scope of a *declaration.* The term "scope of a name," however, is clearly an inadequate notion, since the same name may be used in multiple declarations.

2 Environments and Static Scoping

In CS61A, you saw an abstract model of both scope rules and rules about the *extent* or *lifetime* of variables. In this model, there is, at any given time, an *environment* consisting of a linked sequence of *frames*. Each frame contains *bindings* of names to slots that can contain values. The value contained in a slot may be a function; such a value consists of a pair of values: the *body* (or code) of the function and the *environment* that gives the meaning of names used by the function when it executes.

Figure 1 illustrates the scope of declarations in C (or Java or C++). The sections of text controlled by the various declarations of variables, parameters, and functions are indicated by the brackets on the right. Brackets on the left indicate *declarative regions*—portions of the text of a program that bound the scopes of the declarations within. Declarations in C obey the rule that their scope runs from the declaration to the end of the innermost declarative region that contains them. The declarative regions in C are the boundaries of the source file

[1]C and C++ distinguish *declarations,* which introduce (or re-introduce) names and some information about them from *definitions,* which provide complete information about names. A declaration of a function tells us its name, parameter types, and return type. A definition of a function tells us all this and also gives its body. In these notes, I will use the term *declaration* to refer to both of these functions.

The environment diagram in Figure 1 shows a snapshot of the program during its execution. To find the current meaning (binding) of any identifier, one traverses the environment structure from the current environment, following the pointers (*links*) to enclosing environments, until one finds the desired identifier. Each frame corresponds to an instance of some declarative region in the program. When a function is called, a frame is created for that call that contains the variables declared in that function with a static link that is set from the environment part of the function (the leftmost "bubbles" in the figure). Inner blocks are treated like inner functions, and have static links that point to instances of their enclosing blocks[2]. Since the language in the example is C, all named functions' environments are the *global environment,* encompassing all declarations in a given source file.

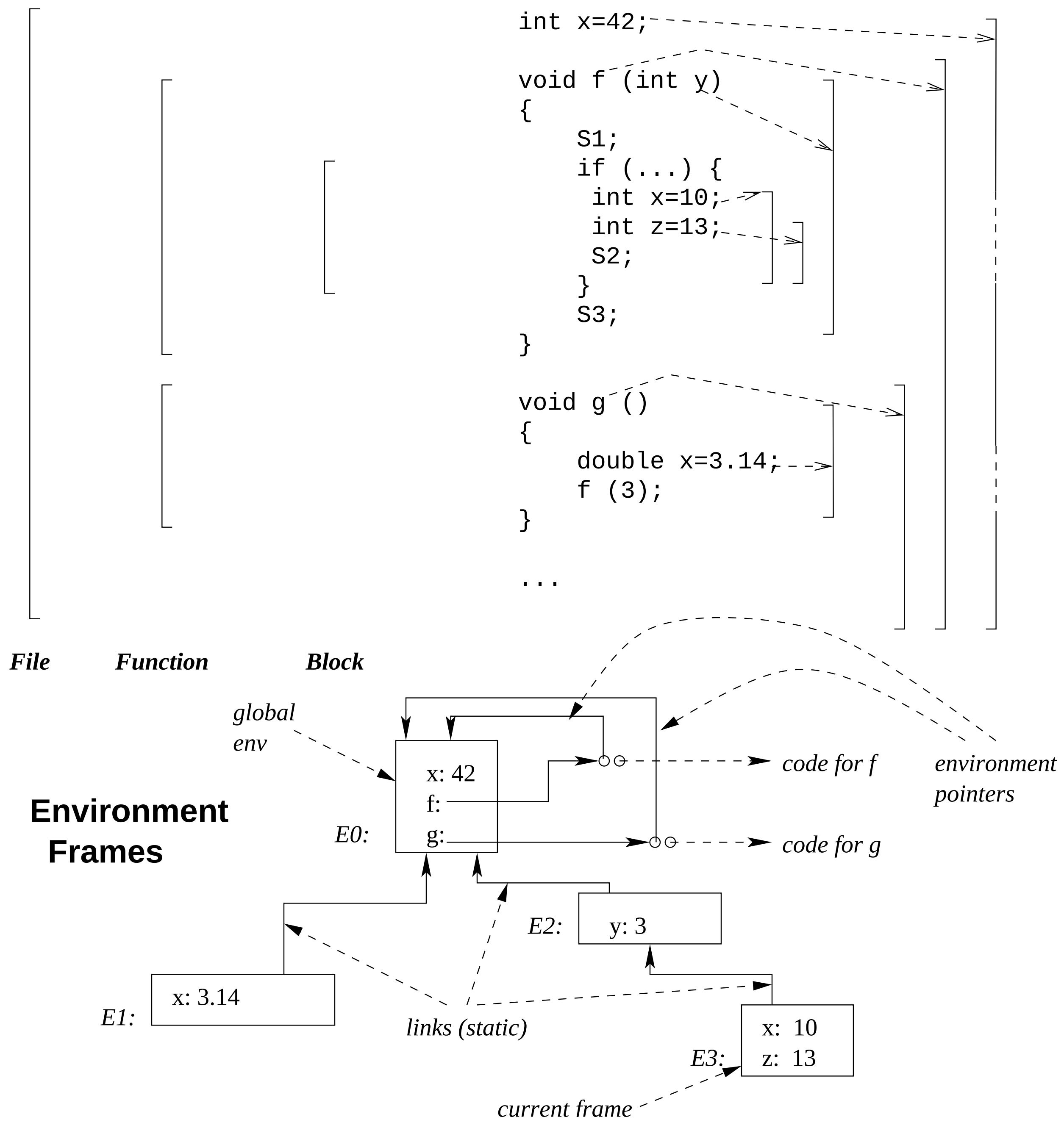
As it was presented in CS61A, these environments are dynamic entities, constructed during execution time. However, it is a property of most languages—including C, C++, Java, Pascal, Ada, Scheme, the Algol family, COBOL, PL/1, Fortran, and many others—that at any given point in a program, the chain of environment frames starting at the current environment is always the same at that point in the program, except for the actual values in the slots of environment. That is, the number of frames in the chain and the names in each of these frames is always the same, and depends only on where in the program we are. We say that these languages use use *static scoping* (also known as *lexical scoping*), meaning that their scope rules determine static regions of text that are independent of any particular execution of the program. The links between frames in this case are called *static links.*

To see why this property holds (the constancy or staticness of the environment chain), consider how the links get set in the first place. When a function value is created in a statically scoped language (i.e., as opposed to being called), its value is constructed from a *code pointer* and an *environment pointer*. The environment pointer, furthermore, is always the current frame, which is a frame containing declarations from the function whose text contains the definition of the function. The environment pointer, in other words, depends only on where in the text of the program the function is defined, and points to the same kind of frame (same names) each time. When a function is called, a new current frame is created to contain declarations of the function's parameters and local variables, and its link (in these languages) is copied from the environment pointer of the function being called. Thus, every time a frame for a given function is created, its link always points to a frame with the same structure.

3 Dynamic Scoping

An alternative rule (found in Logo, APL, SNOBOL, and early versions of Lisp, APL, among other languages) defines the scope of a declaration as proceeding from the *time* at which the declaration is "executed" (or *elaborated,* to use Algol 68 terminology) to the time it terminates. Under dynamic scoping, the link of an environment frame for a function is equal to the current frame at the time of the *call.* To see the distinction, consider the following

[2]WARNING: This is a *conceptual* description. The actual execution of a program involves different data structures, as we will see in later lectures



```
int x=42;

void f (int y)
{
    S1;
    if (...) {
     int x=10;
     int z=13;
     S2;
    }
    S3;
}

void g ()
{
    double x=3.14;
    f (3);
}

...
```

Figure 1: Scope of declarations in C. The brackets on the right indicate the scope of the declarations in the program. The dashed portions of the rightmost bracket (for the first declaration of `x`) indicate the portion of text in which its declaration is *hidden* by another declaration. The brackets on the left are declarative regions, which bound the scopes of items defined inside them. The environment diagram below shows the situation during execution of the inner block of `f`, when it has been called from `g`.

```
void f(int x)     /* (2) */
{
   g ();
}

void g ()
{
   print (x);
}

void doit ()
{
   int x = 12;
   f(42);
   g();
}
```

In normal C (or C++ or Java), this program would print '3' twice. Were these languages to use dynamic scoping instead, it would print '42', and then '12'. Figure 2 shows the environment structure during the two calls to g under static scoping and under dynamic scoping. There isn't one declaration of `x` in the body of `g` (hence the term "dynamic").

4 Compiler Symbol Tables

For languages with lexical scoping, the environment model suggests properties of a data structure (or *symbol table,* as it is generally known) whereby a compiler can keep track of definitions in the program it is processing. It can use the environment structure, but rather than store *values* in the slots of the frames, it can store *declarations* (well, actually, pointers to things that represent these declarations). This data structure allows the compiler to map any identifier in the program it is processing to the declaration for that identifier (and thus to any information contained in that declaration). Figure 3 shows how this would work for the example in Figure 1.

We can make minor changes to the environment model to accommodate a range of language features:

- Java and C++ both allow overloading of functions, based on argument types. We model this by having the names include *argument signatures.* For example, the function `f` in the figures could appear in its environment frame as

  ```
  void f(int):
  ```

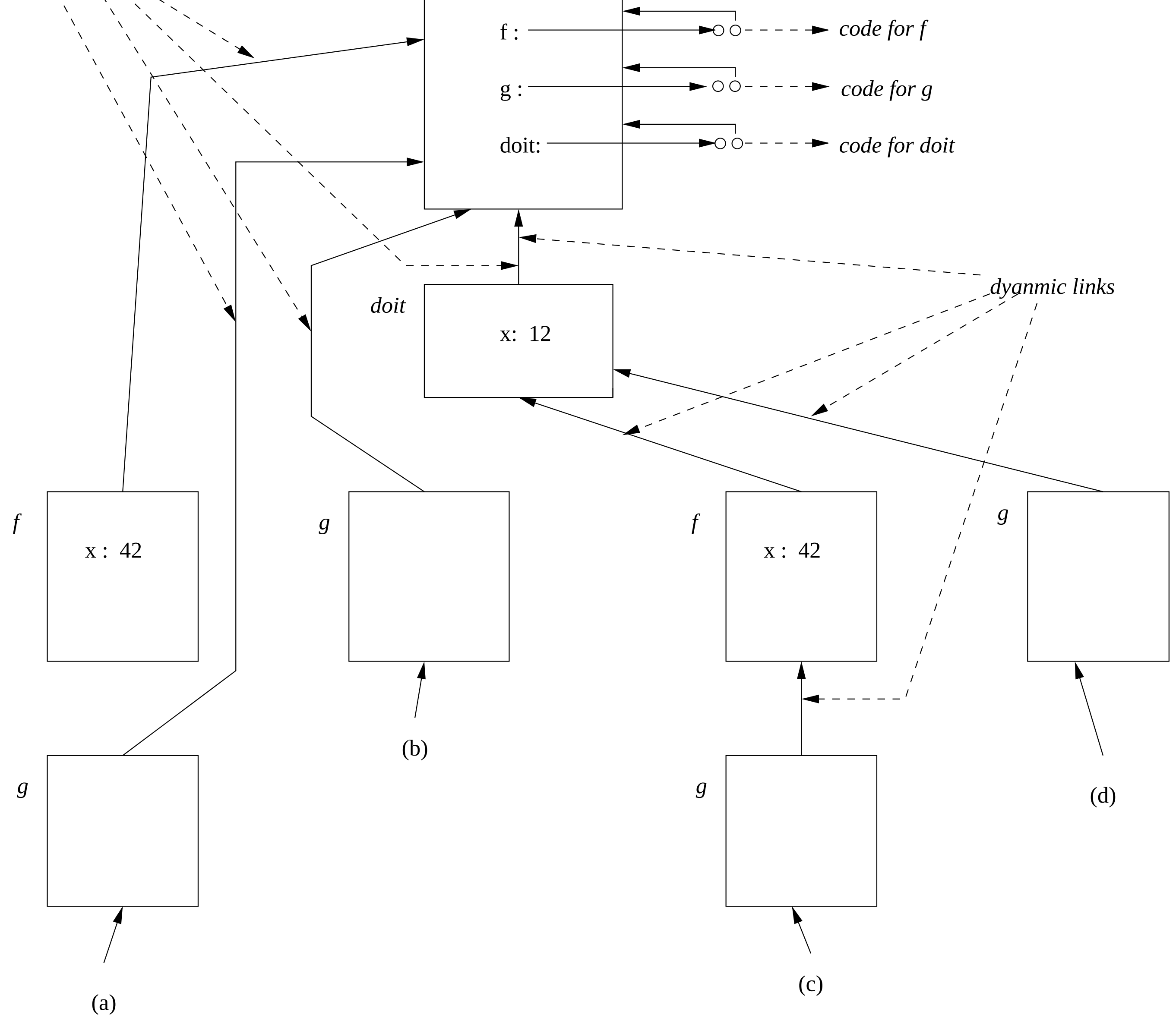
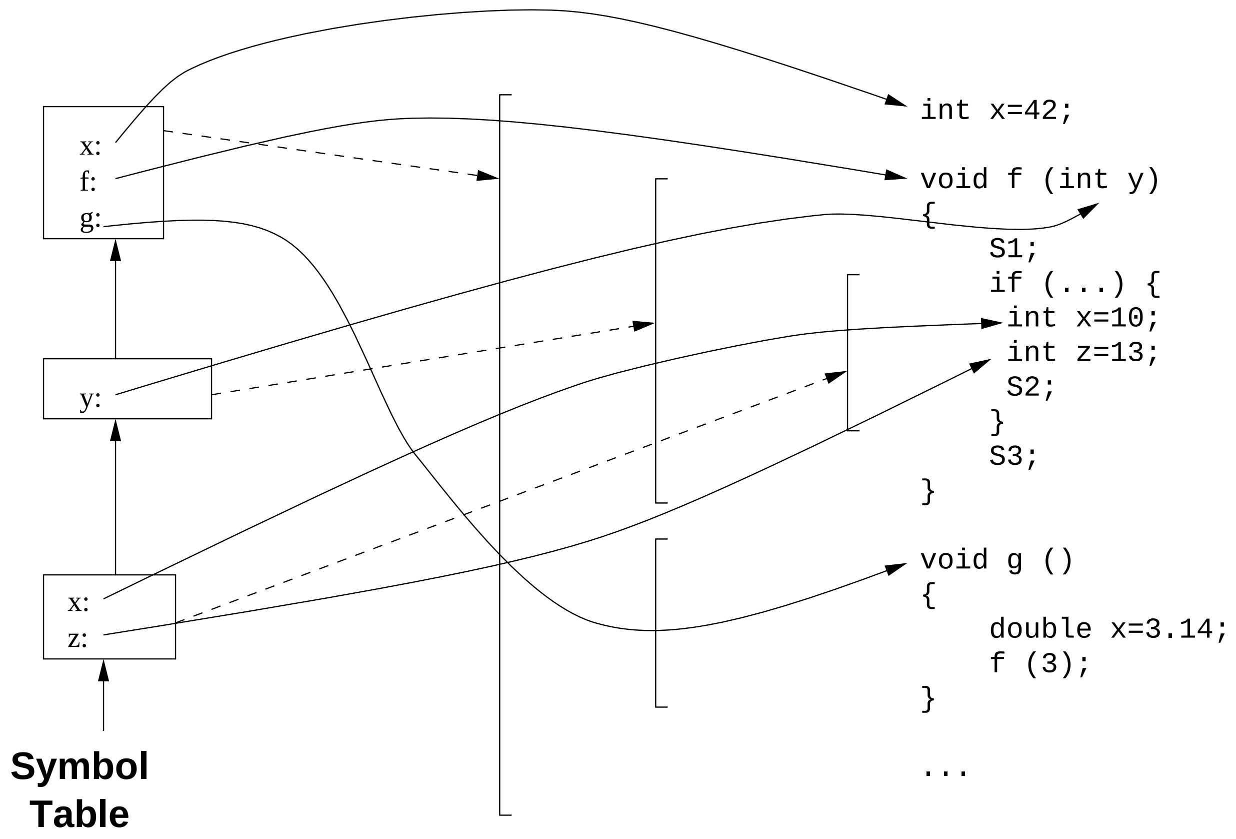


Figure 2: Situation during call to `g` in four situations: (a) called from `f` using static scoping, (b) called from `doit` using static scoping, (c) called from `f` using dynamic scoping, and (d) called from `doit` using dynamic scoping. Situations (a) and (b) show what happens in languages like C, C++, Scheme, and Java (both print 3). Situations (c) and (d) apply to older Lisps, APL, SNOBOL, and others (case (c) prints 42 and (d) prints 12). The links between environments are static for (a) and (b) and dynamic for (c) and (d). The static and dynamic links from `doit` happen to be identical in this case.



```
int x=42;

void f (int y)
{
    S1;
    if (...) {
     int x=10;
     int z=13;
     S2;
    }
    S3;
}

void g ()
{
    double x=3.14;
    f (3);
}

...
```

Figure 3: Adapting environment diagrams as symbol tables. This shows the compiler's symbol table (in the abstract) when processing the inner block of `f`. Compare this with Figure 1. The values in the previous diagram become declarations in the symbol table. Each frame corresponds to a declarative region.

x.y, we look up y in starting from the environment that represents x's type.

- When languages have structure types with inheritance, as do C++ and Java, it can be represented by having the static link of a frame representing one type point to the parent or base type of that type. Where multiple inheritance is legal (again as in C++ or Java), we can generalize the static link to be a set of pointers rather than a single one.

5 Lifetime

Finally, we leave with one important point. The scope of a declaration refers only to those sections of the program text where that declaration determines the meaning of its name. This is a distinct concept from that of how long the named entity actually exists. The latter is what we'll call the *lifetime* of the entity. For example, going back to Figure 1, the declaration of `x` inside the text of `g` is out of scope (invisible) during execution of `f`, but the variable (slot) created by that declaration does not go away. Nor does variable created by the first (global) declaration of `x` go away in the "scope holes" where it is hidden. The situation in the C++ declaration

```
void f ()
{
   Foo* x = new Foo;
   g ();
   other stuff involving x;
}
```

is even more complicated. The declaration of `x` introduces a variable called `x` and then stores into it a pointer to an *anonymous* object (created by **new**) that is not named by any declaration. The lifetime of a variable `x` ends upon return from the call to `f` that creates it. The lifetime of the anonymous thing it points to, however, continues until it is explicitly deleted. Both variable `x` and the object it points to, of course, continue to exist during the call to `g`, even though the declaration of `x` is not visible in `g`.

6 Static and Dynamic Typing

In programming languages, a *type* is a characteristic of an expression, parameter, variable (or other entity that is, denotes, or holds a value) that characterizes what values the entity may have (or denote or ...) and what operations may be applied to it. When, as in Scheme, the type of a quantity is not determined, in general, until a program is executed, we say the language is *dynamically typed*. When—as in C, C++, FORTRAN, COBOL, Algol, Pascal, PL/1, or Java—the type of an entity in a program is determinable solely from the text of the program, we say that the language is *statically typed*.

of two entities to match. For example, consider the following code in C or C++:

```
struct { int x, y; } A, B;
struct { int x, y; } C;
struct T { int x, y; } D;
struct T E;
int* F;
int* G;

main()
{
    A = B;  /* OK */
    A = C;  /* Error: type mismatch */
    A = D;  /* Error: type mismatch */
    D = E;  /* OK */
    F = G;  /* OK */
}
```

The constructs '`struct {...}`' and '`...*`' are *type constructors:* given a set of *type parameters,* represented here by '`...`', they construct a new type.

As the comments show, the rule in C and C++ (also in Pascal, Ada, and many other languages) is that each distinct occurrence of `struct` creates a brand new type, differing from (not equivalent too) all other types, even those with identical declarations. `A` and `B` have the same type, since it is "generated" by the same instance of `struct`. `D` and `E` have the same type, since the definition of `D` introduces the name `T` to stand for the newly-constructed type, and the declaration of `E` then refers to that type by name. We call this kind of rule a *name equivalence* rule.

On the other hand, the types of `F` and `G` are identical, despite the fact that their types come from two distinct instances of a generator. The two instances of `T*` define types with identical *structures:* identical arguments to the type constructor. The rule in C and C++ is that structurally identical pointer types are all the same. We call this kind of rule a *structural equivalence* rule.

As you can see, Java, C, and C++ mix the two types of rule freely: array types, pointer types, reference types (C++), and function types obey structural equivalence rules, while class (struct) and union types obey name equivalence rules. The languages Pascal and Ada, on the other hand, adhere more consistently to the name equivalence model (array types, for example, are different if produced from two separate constructors). The language Algol 68 adheres consistently to structural equivalence (all the struct types in the example above are identical in Algol 68).

usually prove to be unduly burdensome to programmers. Most languages therefore provide some set of implicit *coercions* that automatically convert one type of value to another. In Java, C, and C++, for example, we have the standard numeric promotions, which convert, e.g., `short` and `char` values to `int` or `long`. We also have the standard pointer conversions, which translate any pointer to an object to a `void*`, and convert `B*` to `A*` if `A` is a base class of `B`.

6.3 Overloading

The purpose of name resolution (scope rule checking), is to associate declarations with instances of names. Java and C++ introduces a new wrinkle—the possibility of having several declarations referred to by the same name. For example,

```
int f(int x) { ... }  /* (1) */
int f(A y) { ... }    /* (2) */
```

In the presence of these declarations, scope rule checking will tell us that `f` in

```
f(3)
```

can mean *either* declaration (1) or (2). Until we analyze the type of the expression, however, we can't tell which of the two it is.

C++ requires that the decision between declarations (1) and (2) must depend entirely on the types of the arguments to the function call. The following is illegal:

```
int f2(int x) { ... } /* (3) */
A   f2(int x) { ... } /* (4) */
int x;
...
x = f2(3)
```

On the other hand, the language Ada allows these declarations (well, in its own syntax), and is able to determine that since `f2` must return an `int` for the assignment to be legal, one declaration (3) can apply. That is, Ada uses the result types of the overloaded declarations as well as their argument types.

Both C++ and Ada provide for *default parameters:*

```
int g(int x, int y = 0) { ... } /* (5) */
```

This doesn't really introduce anything new, however; we can treat it as

```
int g(int x, int y) { ... } /* (5a) */
int g(int x) { return g(x, 0); } /* (5b) */
```

C++ does not resolve function calls based on return types, but it does allow user-defined conversions that may be chosen according to the required type of an expression. For example,

```
        ...
        operator int() const { ... }
        ...
    }
```

defines an operator (which is, after all, just a function with an attitude) that will convert an object of type `A` into an `int`, if one is needed in a given context. That makes it legal to write, e.g.,

```
    int h(int x) { ... }
    A z;
    ...
    h(z);
```

Any system of coercions as complex as that of C++ tends to give rise to unwanted ambiguities (several different ways of coercing the arguments of to a function that match several different possible overloadings of the function). To counter this, C++ has a complex set of rules that place a preference order on implicit coercions. For example, given the definitions of `f` and `A` above, it is acceptable (unambiguous) to write

```
    A z;
    ... f(z)...
```

despite the fact that declaration (2) matches the call and declaration (1) matches it after an application of the conversion defined from `A` to `int`. This is because the rules indicate a preference for calls that do not need user-defined conversion.

Of all the features in this section, Java uses only overloading on the argument type.

7 Unification (aka Pattern-Matching)

In C++, one can define *template functions,* such as

```
    /* sort A[0..n-1] so that le(A[i], A[i+1]) for 0 <= i < n-1 */
    template <class T>
    void sort(T* A, int n, bool le(const T&, const T&))
    {
        ...
    }
```

which will work for any type `T`:

```
    bool double_le(const double& x, const double& y)
    {
        return x <= y;
    double V[100];

    ... sort(V, 100, double_le) ...
```

For each distinct `T` that is used in a call to `sort`, the compiler creates a new overloading of `sort`, by filling in the '`T`' slot of the template with the appropriate type. This feature of C++ therefore raises the questions of how one matches a call against a rule like this and how one determines what type to plug into `T`.

Given a call, such as `sort(V, 100, double_le)`, we can determine the types of all the arguments, and represent them as a list of trees, such as the following (to use Lisp notation).

```
(list
 (ArrayType (DoubleType))
 (IntType)
 (FunctionType
   ((RefToConstant (DoubleType))
    (RefToConstant (DoubleType)))
   (BoolType)))
```

Likewise, we can do the same for the formal parameter list of `sort`:

```
(list
 (ArrayType T)
 (IntType)
 (FunctionType
   ((RefToConstant T)
    (RefToConstant T))
   (BoolType)))
```

Each S-expression $(AY_1 \cdots Y_n)$ denotes a tree node with operator (label) A and n children Y_i. I've put in a dummy operator, `list`, in order to make the algorithm below easier to write.

The task of seeing whether these two types match is one of pattern matching, or to use the fancy phrase, *unification:* that is, finding a substitution for all the type variables (here, the single template parameter `T`) that makes the list of argument types and the formal parameter list match.

The algorithm finds out whether a pattern matches an argument list, and finds a *binding* of each type variable to a type. Initially, each type variable is unbound. The construct `binding(`x`)` denotes the thing that type variable x is currently bound to. The construct `oper(`t`)` gives the operator of tree node t, and `children(`t`)` gives its children. The following algorithm is a version of the one in §6.7 of the book.

```
/* Return true iff the pattern P matches the pattern A, and binds */
/* type variables in A and P so as to make the two match. */
match(P, A) {
    }
    while (A is a bound type variable) { /* X */
        A = binding(A);
    }
    if (P is an unbound type variable) {
        binding(P) = A;
        return true;
    }
    if (A is an unbound type variable) {  /* X */
        binding(A) = P;
        return true;
    }
    if (oper(P) != oper(A)
        || length(children(P)) != length(children(A)))
        return false;
    for each child cp in children(P)
             and corresponding child ca in children(A) {
        if (! match (cp, ca))
            return false;
    }
    return true;
}
```

For use with C++, `A` never contains type variables, so we can eliminate the two statements marked `/* X */`.

7.1 Type inference

Languages such as ML make interesting use of unification to get static typing without having to mention the types of things explicitly. For example, consider the following function definition (the syntax is fictional):

```
    sum(L) = if null(L) then 0 else head(L) + sum(tail(L)) fi;
```

Here, `if...then...else...fi` is a conditional expression, as for '`if`' in Scheme or '`?`' and '`:`' in C. The rules for this fictional language say that

- `null` is a family of functions (what is called a *polymorphic function*) whose argument types are `list of` T for all types T, and whose return type is `bool`.
- `head` is a family of functions whose argument types are `list of` T and whose return type is T for all types T.
- `tail` is a family of functions whose argument and return types are `list of` T for all types T.

of the ‘if’ as a whole.

- A (one-argument) function has a type $T_0 \rightarrow T_1$ for some types T_0 and T_1. Its arguments must have type T_0 and its return value is type T_1.

To begin with, we don’t know the types of `L` or `sum`. So, we initially say that their types are the type variables T_0 and T_1, respectively. Now we apply the rules using the matching procedure described above.

- `sum` is a function, so its type (T_1) must be $T_2 \rightarrow T_3$ for some types T_2 and T_3; that is `match(`T_1`, `$T_2 \rightarrow T_3$`)` must be true.
- `x` is an argument to `sum` so `match(`T_0`, `T_2`)`.
- The argument type of `null` is `list of` T_4 for some type T_4, so `match(`T_0`, list of` T_4`)`.
- If the argument type to `head` is `list of` T_4, the the return type is T_4.
- Since the operand types of ‘+’ must be `int`, `match(`T_4`,int)`.
- The return type of `sum` is the type returned by the ‘if’. That, in turn, must be the same as the type of `0` (its ‘then’ clause). So, `match(`T_3`,int)`.

Work this all through (executing all the ‘`match`’ operations) and you will see that we end up with the type of `x` (T_0) being `list of int` and that of `sum` being `list of int` $\rightarrow$ `int`.

7.2 Overload resolution in Ada

As indicated in the last lecture, C++ disallows

```
int f2(int x) { ... } /* (3) */
A   f2(int x) { ... } /* (4) */
int x;
...
x = f2(3);
```

because the the call on `f2` cannot be resolved on the basis of the argument type alone. Ada, on the other hand, does allow this situation (well, with a different syntax, of course).

To make the problem uniform, first of all, we can treat all operators as functions. Thus, we rewrite

```
x = f2(3);
```

as

```
operator=(x, f2(3));
```

- Look at all definitions of the function in question for one whose formal parameters match the resulting list of argument types (as for the `match` procedure above).

When the return type of a function matters, however, things get complicated.

The naive approach is to try all combinations of definitions of all functions mentioned in an expression (for the example above, all possible definitions of `operator=` with all possible definitions of `f2`). If the average number of overloadings for any function definition is k (the geometric mean, to be precise), and the number of function calls in an expression is N, then this naive procedure requires $\Omega(N^k)$ time, which is definitely bad.

A much better procedure is the following. We operate on an abstract syntax tree representing an expression.

- Perform a post-order traversal of the tree determining a *list of possible types* for the expression:
  - If the expression is a literal or variable, return its (single) possible type.
  - If the expressions is a function call,
    - recursively determine the possible types for each operand;
    - look at all overloadings of the function and find which of them match one of the possible types in each argument position.
    - return the list of return types for each of the overloadings that match.
- If this procedure results in more than one possible type for the expression as a whole, the expression is ambiguous and there is an error.
- Otherwise, perform a pre-order traversal of the tree, passing down the (single) type, R that the expression must have:
  - If the expression is a literal or variable, nothing needs to be done on this pass.
  - For a function call, check that only one of the possible overloaded definitions for the called function (determined on the post-order pass) returns the given type, R. If not, there is an error.
  - Otherwise, the single definition selected in the preceding step determines the types of the arguments. Recursively resolve the argument expressions, passing down these argument types as R.