CS558 Programming Languages

Winter 2013

Lecture 3

Names and Binding

One essential part of being a “high-level” language is having convenient **names** for things:

variables	classes
constants	modules
types	record fields
functions	operators
etc.	

• Allowed syntactic form of names varies for different languages, but intended to be human-readable.

We distinguish **binding** and **use** occurrences of a name.

• A **binding** makes an association between a name and the thing it names.

• A **use** refers to the thing named.

Binding and Use Examples

For example, in this OCaml code:

```
let rec f (x:int) =
   if (x > 0) then
     f(x + 1)
   else 0
```

The first line binds both `f` (as a recursive function) and `x` (as a formal parameter); the second line uses `x`; the third line uses both `f` and `x`.

It is common for some names to be **pre-defined** for all programs in a language, e.g., the type name `int` in the above example. Often the binding of these names is done in a standard library that is implicitly included in all programs. It may or may not be legal for programs to redefine these names.

Don't confuse pre-defined names with built-in **keywords** in the language, like `let` and `if` in the above example. Keywords are a fixed part of the language syntax and can never be redefined.

Are operators like + and > fixed or definable ?

Scoping Rules

A key characteristics of any binding is its **scope**: in what textual region of the program is the binding visible?

• I.e., where in the program can the name be used?

• Alternatively: given a use of the name, how do we find the relevant binding for it?

In most languages, the scope of a binding is based on certain rules for reading the program text. This is called **lexical** scope. (Or **static** scope, because the connection between binding and use can be determined statically, without actually running the program.)

The exact rules for lexical scoping depend on the kind of name and the language.

Delimiting Scopes

Depending on the language, many different kinds of syntax constructs might be used to define scope boundaries:

- Functions (e.g. in C/C++/Java/...):

```
int f(int x) { return x+1; }
```

- Blocks (e.g. in C/C++/Java/...):

```
while (1) { int x = 10; y = x+1; }
```

- Expressions (e.g. in OCaml):

```
(let x = 10 in x+1) + 10
```

- Classes (e.g. in Java):

```
class Foo { int y; int f(int x) { return x + y; } }
```

- Files (e.g. in C)

Typical Scoping Rules Illustrated in C

```
static int x = 101;
bar (double y) {
  if (y > x)
    bar(y + 1.0);
}
main () {
  double v = bar (3.14);
  double z = 99.0;
  { double w;  /* inner block */
    w = z + x;
  }
}
```

- `x` is in scope throughout this C file (but not in other files).
- `bar` is in scope from its point of definition to the end of the C file (including its own body); similarly for `main`.
- `y` is in scope throughout the body of `bar`; similarly for `v`.
- `z` is in scope from its point of definition to the end of `main`.
- `w` is in scope inside the inner block of `main`.

Name Binding Conflicts

What happens when the same name is bound in more than one place?

- If the bindings are to different kinds of things (e.g., types vs. variables), the language's concrete syntax often gives a way to distinguish the uses, so no problem arises (except perhaps for readability):

```
typedef int foo;  /* foo is a synonym for int */
foo foo = 3;
foo w = foo + 1;
```

- Here we say that types and variables live in different **name spaces**.

But what if there are duplicate bindings within a single name space?

- If the bindings occur in a single scoping construct, this is usually treated as an error.
- Sometimes additional rules (such as typing information) is used to determine which binding is meant. Names like this are said to be **overloaded**.

Nested Scopes

Scoping constructs can often be **nested** (e.g. a block within a function). What if the same name is bound in a scope and also in some nested inner scope?

- In most languages, a re-binding in a nested inner scope **hides** (or **shadows**) the outer binding; this creates a **hole** in the scope of the outer binding. E.g. in C:

```
int a = 0;
int f(int b) {
  return a+b;   /* use of global a */
}
void main() {
  int a = 1;
  print (f(a)); /* use of local a; prints 1 */
}
```

- Under this sort of scoping, we can find the binding relevant to a particular use by looking **up** the program text and **out** through any nesting constructs.

Python Scope Rules

• These nested scope rules are quite common, but there's nothing magic about them; some languages use different rules.

• E.g., Python normally uses no variable declarations. The local variables of a function are defined to be just the set of variables that are written to somewhere in the (textual) body of the function.

```
a = 100
def f(x):
  return x+a   # reads global a

f(10)  # yields 110
def g(x):
  a = 101      # writes local a
  return x+a   # reads local a

g(10)  # yields 111
a      # yields 100
def h(x):
  a = a + 1    # reads, writes local a
  return a

h(10)  # yields an error (local a has no value when read)
```


Python Scope Rules (continued)

• In order to write the value of a variable that lives at an outer scope, code must explicitly declare it as `global` or (since Python3) `nonlocal`.

```
a = 100
def g(x):
  global a
  a = 101      # writes global a
  return x+a   # reads global a

g(10)  # yields 111
a      # yields 101
def f():
  a = 200      # writes f's local a
  def g():
    nonlocal a
    a = a+1    # writes f's local a
    return 1
  b = g()
  return a+b

f()            # yields 202
```


Named Scopes: Modules, Classes, ...

Often, the construct that delimits a scope can itself have a name, allowing the programmer to manage explicitly the visibility of the names inside it.

• OCaml modules example

```
module Env = struct
  type env = (string * int) list
  let empty : env = []
  let rec lookup (e:env) (k:string) : int = ...
end
let e0 : Env.env = Env.empty in Env.lookup e0 "abc"
```

• Java classes example

```
class Foo {
  static int x;
  static void f(int x);
}
int z = Foo.f(Foo.x)
```


Handling Mutually Recursive Definitions

• In most languages, scopes flow strictly downward from definitions, so that if the definition of `x` depends on `y`, then `y` must be defined first (textually).

• But this doesn't allow for **mutually recursive** definitions (because no single ordering works), which is commonly wanted for functions and types (less often for values).

• So some languages widen the scope of a binding to include the entire syntactic construct in which it is placed. E.g. in Java:

```
class Foo {
  static void f(double x) { g(x+1.0); }
  static void g(double y) { f(y-1.0) + Bar.h(42.0); }
}
class Bar {
  static void h(double z) { Foo.f(z); }
}
```

Recursive Definitions (continued)

• Another alternative is distinguish **declarations** from **definitions**. E.g. in C:

```
void g (double y);  /* declares g but doesn't define it */
void f(double x) { g(x+1.0); }
void g(double y) { f(y-1.0); }   /* definition is here */
```

• Historically, this approach was taken so that compilers could process programs one function in a single forward pass (no longer a common requirement).

• A third alternative is to use explicit syntax to link mutually resursive definitions. E.g. in OCaml:

```
let rec f(x:float) = g(x +. 1.0)
and     g(y:float) = f(y -. 1.0)
```

• Note that all these approaches to recursion break the "up and out" rule for finding bindings.

Dynamic Scope

There's an alternative approach to scoping which depends on program execution order rather than just the static program text. Under **dynamic scoping**, bindings are (conceptually) found by looking backward through the program **execution** to find the most recent binding that is still active.

An earlier example (still in C syntax):

```
int a = 0;
int f(int b) {
  return a+b;  }
void main() {
  int a = 1;
  print (f(a)); }
```

• Here the use of `a` in `f` refers to the local declaration of `a` within `main`.

• Global `a` isn't used at all; result printed is `2`.

• Early versions of LISP used dynamic scope, but it was generally agreed to be a mistake.

• Some scripting languages still use it (mainly to simplify implementation).

Free Names

In any given program fragment (expression,statement, etc.) we can ask: which names are used but not bound ? These are called the **free** names of the fragment.

The notion of **free** depends both on the name and the fragment. For example, given the C fragment

```
int f (int x) {
  return x + y;
}
```

we say that `y` is free, but `x` is not. (What other names are free?)

However, in the sub-fragment

```
return x+y;
```

we say that **both** `x` and `y` are free.

The **meaning** of a fragment clearly must depend on the values of its free names. To handle this, we usually give semantics to fragments relative to an **environment**.

Environments

An **environment** is just a mapping from names to their meanings. Exactly what gets associated to a name depends on the kind of name:

For example:

• a **variable** name usually gets bound to the location containing the value of the variable.

• **function** names usually get bound to descriptions of the function's parameters and body.

• **type** names get bound to a description of the type, including the layout of its values.

• **module** names get bound to a list of the contents of the module.

• **constant** names get bound to a value (either at compile time or at run time). "Variables" in purely functional languages work act like constants computed at run time.

Representing Environments

To implement operational semantics as an interpreter, we need to choose a concrete representation for environments that supports the operations.

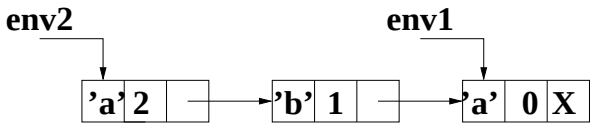
In particular, we want to make it easy to **extend** an environment with new bindings — which may hide existing bindings — while still keeping the old environment around. (This is useful so that we can enter and exit nested scopes easily.)

A simple approach is to use a singly-linked list, which is always searched from, and extended at, its head.

A more efficient approach might be to use a balanced tree or hash table. But we'll stick to the simpler list-based approach in our interpreters.

List-based environments: Example

```
int a = 0;    /* env1 */
{
   int b = 1;
   int a = 2; /* env2 */
   a = a + b;
}
a = a + 1;
```



Values and Locations

In most imperative programming languages, **variable** names are bound to **locations**, i.e. memory addresses, which in turn contain **values**. So declaring a variable typically involves two separate operations:

• allocating a new location (e.g., on the stack or in the heap) and perhaps initializing its contents;

• creating a new binding from the variable name to the location.

At an abstract level, we can temporarily ignore the question of where new locations are created, and simply say that the program has a mutable **store**, which maps locations to values.

But most languages support several different kinds of storage locations; to understand these languages, we'll need to model the store in more detail.

Storage Lifetimes

Typically, a computation requires more locations over the course of its execution than the target machine can efficiently provide — but at any given point in the computation, only some of these locations are needed.

Thus nearly all language implementations support the **re-use** of locations that are no longer needed.

The **lifetime** of an allocated piece of memory (loosely, an "object") extends from the time when it is allocated into one or more locations to the time when the location(s) are freed for (potential) re-use.

For the program to work correctly, the lifetime of every object should last as long as the object is being used. Otherwise, a memory bug will occur.

Most higher-level languages provide several different classes of storage locations, classified by their lifetimes.

Storage Classes Based on Data Lifetime

Static Data : Permanent Lifetimes

- Global variables and constants.
- Allows fixed address to be compiled into code.
- No runtime management costs.
- Original FORTRAN (no recursion) used even for local variables.

Stack Data : Nested Lifetimes

- Allocation/deallocation and access are cheap (via stack pointer).
- Good **locality** for VM systems, caches.
- Most languages (including C, Algol/Pascal family, Java, etc.) use stack to store local variables (and internal control data for function calls).

Heap Data : Arbitrary Lifetimes

- Supports explicit allocation; needs deallocation or garbage collection.
- Lisp, OCaml, many interpreted languages use heap to store local variables, because these have non-nested lifetimes.

Scope, Lifetime, and Memory Bugs

Lifetime and scope are closely connected. To avoid memory bugs, it suffices to make sure that in-scope identifiers never point (directly or indirectly) to deallocated storage areas.

For stack data, the language implementation normally enforces this requirement automatically.

- A function's local variables are typically bound to locations in a stack frame whose lifetime lasts from the time the function is called to the time it returns — exactly when its variables go out of scope for the last time.

For heap data, the requirement is trickier to enforce, unless the language uses a garbage collector.

- Most collectors work by recursively **tracing** all objects that are accessible from identifiers currently in scope (or that might come back into scope later during execution).
- Only unreachable objects are deallocated for possible future re-use.

Problems with Explicit Control of Lifetimes

Many older languages support pointers and explicit deallocation of storage, which is typically somewhat more efficient than garbage collection.

But explicit deallocation makes it easy for the programmer to accidentally kill off an object even though it is still accessible, e.g.:

```
char *foo() {
  char *p = malloc(100);
  free(p);
  return p;}
```

Here the allocated storage remains accessible (via the value of variable `p`) even after that storage has been freed (and possibly reallocated for something else).

This is usually a **bug** (a **dangling pointer**). The converse problem, failing to deallocate an object that is no longer needed, can cause a **space leak**, leading to unnecessary failure of a program by running out of memory.

Large Values

Real machines are very efficient at handling small, fixed-size chunks of data, especially those that fit in a single machine **word** (e.g. 16-64 bits), which usually includes:

- Numbers, characters, booleans, enumeration values, etc.
- Memory addresses (locations).

But often we want to manipulate larger pieces of data, such as records and arrays, which may occupy many words.

There are two basic approaches to representing larger values:

- The **unboxed** representation uses as many words as necessary to hold the contents of the value.
- The **boxed** representation of a large value **implicitly** allocates storage for the contents on the heap, and then represents the value by a pointer to that storage.

Boxed vs. Unboxed

For example, consider an array of 100 integers. In an **unboxed** representation, the array would be represented directly by 100 words holding the contents of the array. In a **boxed** representation, the array would be indirectly represented by an implicit 1-word pointer to 100 consecutive locations holding the array contents.

The language's choice of representation makes a big difference to the semantics of operations on the data, e.g.:

- What does assignment mean?
- How does parameter passing work?
- What do equality comparisons mean?

Unboxed Representation Semantics

Earlier languages often used unboxed representations for records and arrays. For example, in Pascal and related languages,

```
TYPE Employee =
RECORD
  name : ARRAY (1..80) OF CHAR;
  age : INTEGER;
END;
```

specifies an unboxed representation, in which value of type `Employee` will occupy 84 bytes (assuming 1 byte characters, 4 byte integers).

The semantics of assignment is to copy the entire representation. Hence

```
VAR e1,e2 : Employee;
e1.age := 91;
e2 := e1;
e1.age := 19;
WRITE(e1.age, e2.age);
```

prints 19 followed by 91.

Unboxed Representation Problems

Assignment using the unboxed representation has appealing semantics, but two significant problems:

- Assignment of a large value is expensive, since lots of words may need to be copied.
- Since compilers need to generate code to move values, and (often) allocate space to hold values temporarily, they need to know the **size** of the value.

These problems make the unboxed representation unsuitable for value of **arbitrary size**. For example, unboxed representation can work fine for pairs of integers, but not for pairs of arbitrary values that might themselves be pairs.

Boxed Representation Semantics

OCaml **implicitly** allocates tuples and constructed values in the heap, and represents **values** of these types by **references** (pointers) into the heap. Python does the same thing with objects.

As a natural result, both languages use so-called **reference** semantics for assignment and argument passing. Python example:

```
class emp:
  def __init__(self, name, age):
    self.name = name
    self.age = age
e1 = emp("Fred",91)
e2 = e1
e1.age = 18
print(e2.age)
```

prints 18

Boxed Representation (2)

If you want to copy the entire contents of record or object, you must do it yourself, element by element (though Python has a standard library method called `copy` to do the job).

Notice that the difference between copy and reference semantics only matters for **mutable** data; for immutable data (the default in OCaml), you can't tell the difference (except perhaps for efficiency).

Neither language allows user programs to manipulate the internal pointers directly. And neither supports explicit **deallocation** of records (or objects) either; both provide automatic **garbage collection** of unreachable heap values.

Explicit Pointers

Many languages that use unboxed semantics also have separate **pointer types** to enable programmers to construct recursive data structures, e.g. in C++

```
struct Intlist {
  int head;
  Intlist *tail;
};
Intlist *mylist = new Intlist;
mylist->head = 42;
mylist->tail = NULL;
delete mylist;
```

In C/C++, pointers can hold **arbitrary** memory locations, which opens up many more possibilities for memory bugs beyond those already allowed by explicit deallocation.

In most other languages with explicit pointers, the pointers can only ever hold addresses returned from the allocation operator, so they don't add any additional memory bug potential.

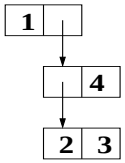
Pairs

We can start studying "large" values in our interpreters by adding in just one new kind of value, the **pair**. You can think of a pair as a record with two fields, each containing a value — which might be an integer or another pair.

We write pairs using "infix dot" notation. For example:

```
(1 . ((2 . 3) . 4))
```

corresponds to the structure:



We can build larger records of a fixed size just by nesting pairs.

Lists

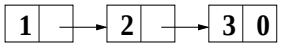
We can also build all kinds of interesting arbitrary-sized **recursive** structures using pairs.

For example, to represent **lists** we can use a pair for each link in the list. The left field contains an element; the right field points to the next link, or is 0 to indicate end-of-list.

Example:

`[1,2,3]`

`(1.(2.(3.0)))`



Note that for programs to detect when they've hit the end of a list, they'll need a way to distinguish integers from pairs.

Formal Operational Semantics

So far, we've presented operational semantics using interpreters. These have the advantage of being **precise** and **executable**. But they are not ideally **compact** or **abstract**.

Another way to present operational semantics is using **state transition judgments**, for appropriately defined machine states.

For example, consider a simple language of imperative expressions, in which variables must be defined before use, using a `local` construct.

```
exp := var | int
    |  '(' '+' exp exp ')'
    |  '(' 'local' var exp exp ')'
    |  '(' ':=' var exp ')'
    |  '(' 'if' exp exp exp ')'
    |  '(' 'while' exp exp ')'
    |   etc.
```

Informally, the meaning of `(local` x e_1 e_2`)` is: evaluate e_1 to a value v_1, create a new store location l bound to x and initialized to v_1, and evaluate e_2 in the resulting environment and store.

State Machine

To evaluate this language, we choose a machine state consisting of:

• the current **environment** E, which maps each in-scope variable to a location l.

• the current **store** S, which maps each location l to an integer value v.

• the current **expression** e, to be evaluated.

We give the state transitions in the form of **judgments**:

$$\langle e, E, S\rangle \Downarrow \langle v, S'\rangle$$

Intuitively, this says that evaluating expression e in environment E and store S yields the value v and the (possibly) changed store S'.

Operational Semantics by Inference

To describe the machine's operation, we give **rules of inference** that state when a judgment can be derived from judgments about sub-expressions.

The form of a rule is

$$\frac{\textit{premises}}{\textit{conclusion}}\ \text{(Name of rule)}$$

We can view evaluation of the program as the process of building an inference tree.

This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content of judgments is different.

Environments and Stores, Formally

• We write $E(x)$ means the result of looking up x in environment E. (This notation is because an environment is like a **function** taking a name as argument and returning a meaning as result.)

• We write $E + \{x \mapsto v\}$ for the environment obtained from existing environment E by **extending** it with a new binding from x to v. If E already has a binding for x, this new binding replaces it.

The **domain** of an environment, *dom*(E), is the set of names bound in E.

Analogously with environments, we'll write

• $S(l)$ to mean the value at location l of store S

• $S + \{l \mapsto v\}$ to mean the store obtained from store S by extending (or updating) it so that location l maps to value v.

• *dom*(S) for the set of locations bound in store S.

Also, we'll write

• $S - \{l\}$ to mean the store obtained from store S by removing the binding for location l.

Evaluation Rules (1)

$$\frac{l = E(x) \quad v = S(l)}{\langle x, E, S\rangle \Downarrow \langle v, S\rangle} \text{ (Var)}$$

$$\frac{}{\langle i, E, S\rangle \Downarrow \langle i, S\rangle} \text{ (Int)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle (\texttt{+}\ \ e_1\ \ e_2), E, S\rangle \Downarrow \langle v_1 + v_2, S''\rangle} \text{ (Add)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad l \notin \mathit{dom}(S') \qquad \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\}\rangle \Downarrow \langle v_2, S''\rangle}{\langle (\texttt{local}\ \ x\ \ e_1\ \ e_2), E, S\rangle \Downarrow \langle v_2, S'' - \{l\}\rangle} \text{ (Local)}$$

$$\frac{\langle e, E, S\rangle \Downarrow \langle v, S'\rangle \quad l = E(x)}{\langle (\texttt{:=}\ \ x\ \ e), E, S\rangle \Downarrow \langle v, S' + \{l \mapsto v\}\rangle} \text{ (Assgn)}$$

Evaluation Rules (2)

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad v_1 \neq 0 \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle (\texttt{if}\ \ e_1\ \ e_2\ \ e_3), E, S\rangle \Downarrow \langle v_2, S''\rangle} \text{ (If-nzero)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle \quad \langle e_3, E, S'\rangle \Downarrow \langle v_3, S''\rangle}{\langle (\texttt{if}\ \ e_1\ \ e_2\ \ e_3), E, S\rangle \Downarrow \langle v_3, S''\rangle} \text{ (If-zero)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle \qquad \langle (\texttt{while}\ \ e_1\ \ e_2), E, S''\rangle \Downarrow \langle v_3, S'''\rangle}{\langle (\texttt{while}\ \ e_1\ \ e_2), E, S\rangle \Downarrow \langle v_3, S'''\rangle} \text{ (While-nzero)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle}{\langle (\texttt{while}\ \ e_1\ \ e_2), E, S\rangle \Downarrow \langle 0, S'\rangle} \text{ (While-zero)}$$

Notes on the Rules

- The structure of the rules guarantees that at most one rule is applicable at any point.
- The store relationships constrain the order of evaluation.
- If no rules are applicable, the evaluation **gets stuck**; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

```
Value eval(Exp e, Env env) { .... }
```

returns a value `v` and has side effects on a global `store` such that

$$\langle \texttt{e}, \texttt{env}, \texttt{store}_{\mathit{before}}\rangle \Downarrow \langle \texttt{v}, \texttt{store}_{\mathit{after}}\rangle$$

The implementation of `eval` dispatches on the syntactic form of `e`, chooses the appropriate rule, and makes recursive calls on `eval` corresponding to the premises of that rule.

Question: how deep can the derivation tree get?