

CS558 Programming Languages

Winter 2008

Lecture 5

NAMES AND BINDING

One essential part of being a “high-level” language is having convenient **names** for things: operators, variables, constants, types, procedures, classes, 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 ML code:

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

The first line binds both f (as a 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.

SCOPING RULES

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

- 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.

SAMPLE SCOPING RULES

C provides some typical examples:

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

- `int` and `double` are predefined throughout (all) programs.
- `x` is in scope throughout this C file.
- `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 inside the body of `bar`.
- `w` is in scope inside the inner block of `main`.

NAME 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:

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

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

But what if there are duplicate bindings within a single namespace?

- Some languages disallow this.
- More often, language allows **holes** in the scope of a binding; these are regions of the program where the binding is **hidden** by another binding of the same name.
- Sometimes additional rules (such as typing information) is used to determine which binding is meant. Names like this are said to be **overloaded**.

HIDING IN BLOCK-STRUCTURED LANGUAGES

C Example:

```
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.
- These lexical scope rules are quite common, but there's nothing magic about them; other languages may differ.

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.

Same 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:

- **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 its possible values.
- **constant** names get bound to a value (either at compile time or at run time). “Variables” in purely functional languages work act like constants.

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:

- creating 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.

REPRESENTING ENVIRONMENTS

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

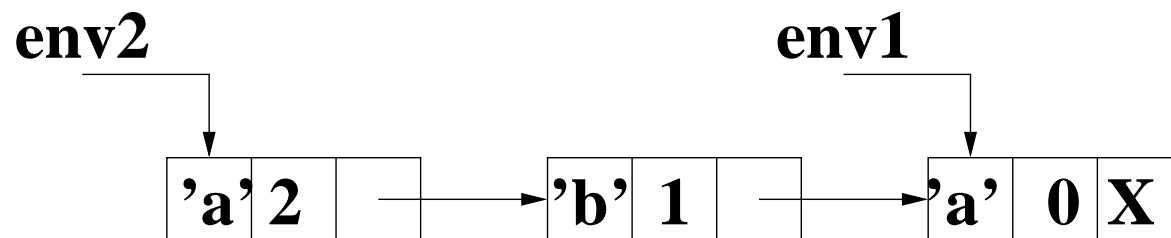
In particular, we want to make it easy to **extend** an environment with new bindings — which may hide existing bindings — while still keeping the old environment around. (This is useful so that we can enter and exit scopes easily.)

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.

LIST-BASED ENVIRONMENTS: EXAMPLE

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



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** uses a pointer to the (first of the) locations holding the contents.

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 works fine for pairs of integers, but not for pairs of arbitrary values that might themselves be pairs.

BOXED REPRESENTATION SEMANTICS

ML implementations **implicitly** allocate tuples and datatype values on the heap, and represent record **values** by **references** (pointers) into the heap. Java does the same thing with objects (although we must say `new` explicitly at points of allocation).

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

```
class emp {
    String name;
    int age;
}
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.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 Java objects do have a standard library method called `clone` to do the job).

Notice that the difference between copy and reference semantics only matters for **mutable** data; for immutable data, 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.

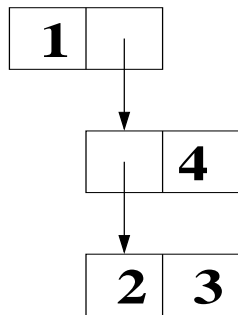
PAIRS

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

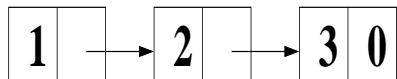
We can also build all kinds of interesting arbitrary-sized **recursive** structures using pairs.

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.

EXPLICIT POINTERS

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

```
typedef struct intcell *intlist;
struct intcell {
    int head;
    intlist tail;
}
intlist mylist =
    (intlist) malloc(sizeof(struct intcell));
while (list != NULL)
    if (list->head != i) then
        list = list->tail;
```

In most such languages, pointers are restricted to addresses returned by allocation operations, but C/C++ allows the address of **anything** to be taken and later dereferenced, and supports **pointer arithmetic**. While this feature can support very efficient code, it also destroys language safety.

LIFETIME

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) get re-used.

For the program to work, the lifetime of an object should last as long as the object is (potentially) being used.

LIFETIME VS. SCOPE

More precisely:

- An object whose location is bound to a variable should live as long as the variable is still in scope.
 - This is normally enforced by the language implementation.
 - E.g., 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 corresponding to the variable's scope.
- An object whose location is itself a value (implicit or explicit) should live as long as the value is accessible. (Normally, values are accessible from variables or fields in other values.)
 - This is trickier to enforce, unless the language uses a garbage collector.

PROBLEMS WITH EXPLICIT CONTROL OF LIFETIMES

If the language supports pointers and explicit deallocation is allowed, it is 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. Using a **garbage collector** avoids both problems.

PROBLEMS WITH UNRESTRICTED POINTERS

Languages like C/C++ that permit the address of **anything** to be used as a pointer can cause even worse problems:

```
int *foo() {  
    int x = 1;  
    return &x;}  
}
```

Here `foo` returns a pointer to its own local variable `x`, but the storage for `x` will almost certainly be overwritten by the next function call.

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 static activation records.

Stack Data : Nested Lifetimes

- Allocation/deallocation is cheap (just adjust stack pointer).
- Most architectures support cheap *sp*-based addressing.
- Good **locality** for VM systems, caches.
- C, Algol/Pascal family, Java use stack for activation records.

Heap Data : Arbitrary Lifetimes

- Needs explicit allocation and (dangerous) explicit deallocation or G.C.
- Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.