Environments

In virtually all programming languages, programmers create symbols (variables) and associate values with them. We discussed bindings earlier. What we want to show now is how to implement bindings. Our implementation allows us to implement *static scope rules*, since this is the most common binding method in programming languages today.

An *environment* is a data structure that associates a value with each element of a finite set of symbols – that is, it represents a set of bindings. We could think of an environment as a set of pairs

$$\{(s_1, v_1), \cdots, (s_n, v_n)\}$$

that encode the binding of symbol s_1 to value v_1, s_2 to value v_2, *etc.* The problem with this simple approach is that the same symbol may have different bindings in different parts of the program, and this approach doesn't make it clear how to determine which binding is the *current* binding.

Instead, we specify an environment as a Java object having a method called `applyEnv` that, when passed a symbol (a `String`) as a parameter, returns the current value bound to that symbol. So if `env` is an environment and `"x"` is a symbol,

```
env.applyEnv("x")
```

would return the value currently bound to the symbol `"x"`.

Environments (continued)

In addition to getting the current value bound to a symbol, our environment implementation provides a way to create an empty environment (one with no bindings), and a way to extend an existing environment (essentially to enter a new *block*) by adding new bindings.

But wait: what type does `applyEnv` return? In other words, what exactly is a "value"? For the time being, we assume that a "value" is an instance of a class aptly named `Val`. We will refine the notion of "value" later. If you're worried about this, just pretend that instances of the `Val` class represent integers. [Don't confuse this use of `Val` with the `Token.Val` class we described earlier – these are not the same.]

Environments (continued)

Our environments are implemented as instances of a Java abstract class `Env`:

```
public abstract class Env {
    // default method to return the value currently bound to sym.
    // this method is overloaded in subclasses of Env,
    // in particular, EnvNode
    public Val applyEnv(String sym) {
        throw new RuntimeException("no binding for "+sym);
    }

    // extend the current environment by adding bindings
    public Env extendEnv(Bindings bindings) {
        return new EnvNode(bindings, this);
    }

    // create an initial (empty) environment
    public static Env initEnv() {
        return new EnvNull();
    }
}
```

We discuss the implementation of bindings next.

We represent a binding as an instance of the class `Binding`. A `Binding` object has a `String` field named `id` that holds an identifier name (a symbol) and a `Val` field named `val` that holds the value bound to that variable.

```
public class Binding {

    public String id;
    publilc Val val;

    public Binding(String id, Val val) {
        this.id = id;
        this.val = val;
    }

}
```

Programming languages typically support many types of values – such as integers, floats, and booleans. The only `Val` type we are concerned with at this point is an integer value represented by a class `IntVal` that extends the `Val` abstract class. Think of an `IntVal` as a *wrapper* for the Java primitive type `int`. Later, we will add new `Val` types as needed to extend functionality.

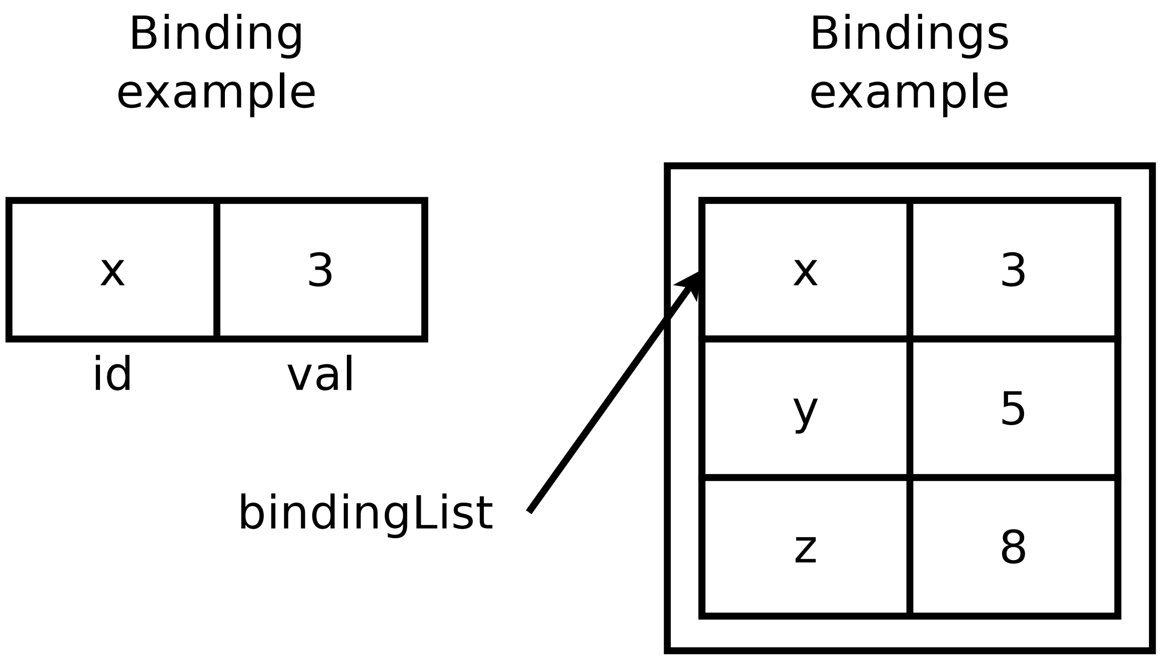
Environments (continued)

A *local environment* is a list of zero or more bindings. In the context of block-structured languages, you can think of a local environment as capturing all of the bindings defined in a particular block. We represent a local environment using the class `Bindings`. A `Bindings` object has a single field named `bindingList` which is a `List` of `Binding` objects.

```
public class Bindings {

    public List<Binding> bindingList;

    // create an empty list of bindings
    public Bindings() {
        bindingList = new ArrayList<Binding>();
    }

    public Bindings(List<Binding> bindingList) {
        this.bindingList = bindingList;
    }

}
```

Environments (continued)

The following diagram gives an example of a `Binding` object that binds the string `"x"` to the integer (`IntVal`) value `3`, and a `Bindings` object with a `bindingList` of size three.



For the sake of simplicity, we omit drawing the extra box around the `bindingList` in future `Bindings` diagrams.

Environments (continued)

Given a local environment, we may want to add a new binding to this local environment either as a `Binding` object or as a pair consisting of an identifier and its corresponding value. The following methods are part of the `Bindings` class:

```
// add a Binding object to this local environment
public void add(Binding b) {
    bindingList.add(b);
}

// add a binding (s, v) to this local environment
public void add(String s, Val v) {
    add(new Binding(s, v));
}
```

Environments (continued)

An empty environment is an instance of the class `EnvNull`.

```
public class EnvNull extends Env {

    public EnvNull () {
    }

}
```

In the empty environment, `applyEnv` – which defaults to the `Env` parent class definition – always throws an exception, since no symbol is bound to any value in the empty environment.

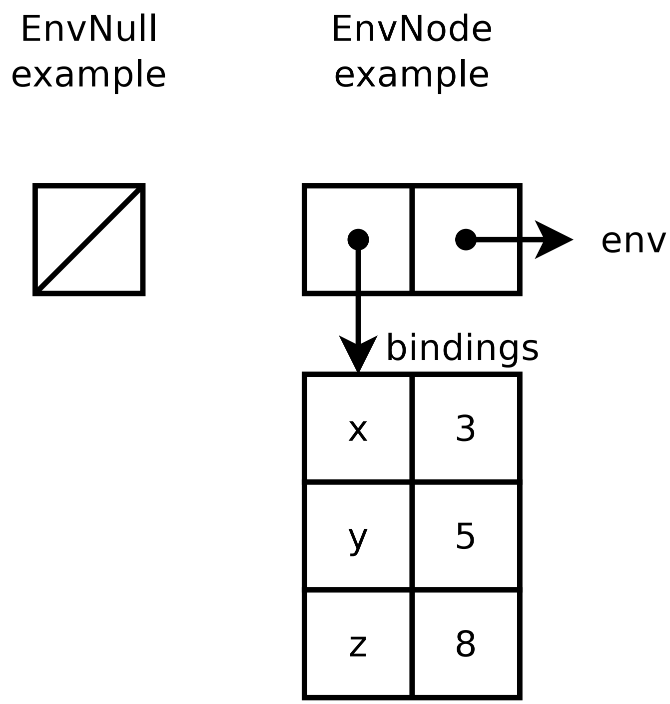
Environments (continued)

A nonempty environment is an instance of the class `EnvNode`. An `EnvNode` object has two fields: a `Bindings` object named `bindings` that holds the local bindings and an `Env` object named `env` that points to an enclosing (in the sense of static scope rules) environment.

```
public class EnvNode extends Env {

    public Bindings bindings; // list of local bindings
    public Env env;           // enclosing scope

    // create an environment
    public EnvNode(Bindings bindings, Env env) {
        this.bindings = bindings;
        this.env = env;
    }

}
```

Environments (continued)

The following diagram gives examples of an `EnvNull` object and an `EnvNode` object. The `EnvNull` object has no fields. The `EnvNode` object in this example has a `bindings` field as shown on slide 2.6 and an `env` field referring to some enclosing environment (not shown).



Environments (continued)

In summary, an environment is a linked list of nodes, where a node is either an instance of `EnvNode` (with its corresponding local bindings) or an instance of `EnvNull`, which terminates the list.

The `extendEnv` procedure, defined in the `Env` class, takes a set of local bindings and uses them to return a new `EnvNode` that becomes the head of a new environment list, extending the current environment list. Here is the definition of `extendEnv`:

```
public Env extendEnv(Bindings bindings) {
    return new EnvNode(bindings, this);
}
```

Environments (continued)

In some cases, we may want to create a `Bindings` object from a `List` of identifiers and a `List` of values by binding each identifier to its corresponding value. The `Bindings` object can then be used to extend an environment.

Here is a constructor for a `Bindings` object that does this pairing, in a slightly more general way that we will find useful later.

```
public Bindings (List<?> idList, List<Val> valList) {
    // the Lists must be the same size
    if (idList.size() != valList.size())
        throw new RuntimeException("Bindings: List size mismatch");
    bindingList = new ArrayList<Binding>();
    Iterator<?> is = idList.iterator();
    Iterator<Val> vs = valList.iterator();
    while (is.hasNext()) {
        bindingList.add(new Binding(is.next().toString(), vs.next());
}
```

The purpose of the `"List<?>"` parameter declaration is to allow for a `List` of either `String`s or `Token`s.

Environments (continued)

The `applyEnv` procedure in the `EnvNode` class is now easy. Simply march through the current `Bindings` to see if there is a binding with the given variable name. If so, return the corresponding value; if not, recursively search the next environment in the list.

Here is the code for `applyEnv` in the `EnvNode` class:

```
public Val applyEnv(String sym) {
    // look first in the local bindings
    for (Binding b : bindings.bindingList) {
        if (sym.equals(b.id))
            return b.val;
    }
    // not found in the local bindings,
    // so look in the next (enclosing) environment
    return env.applyEnv(sym);
}
```

The `applyEnv` method in the `EnvNull` class defaults to the `applyEnv` method in the `Env` class, which throws an exception.

Environments (continued)

For the remainder of these materials, we use the representation for environments that we have described here:

- an environment is a (possibly empty) linked list of local environments
- a local environment is a `List` of bindings
- a binding is an association of an identifier (symbol) to a value

Our implementation defines the following classes: `Env` (with subclasses `EnvNull` and `EnvNode`), `Binding`, and `Bindings` as summarized on the next slide.

Environments (continued)

Class summary:

```
abstract class Env
    public Val applyEnv(String sym)
    public Env extendEnv(Bindings bindings)

class EnvNull extends Env  // empty environment class, no fields

class EnvNode extends Env
    public Bindings bindings // local bindings
    public Env env     // next environment

class Bindings
    public List<Binding> bindingList

class Binding
    public String sym
    public Val val

abstract class Val

class IntVal extends Val
    public int num
```

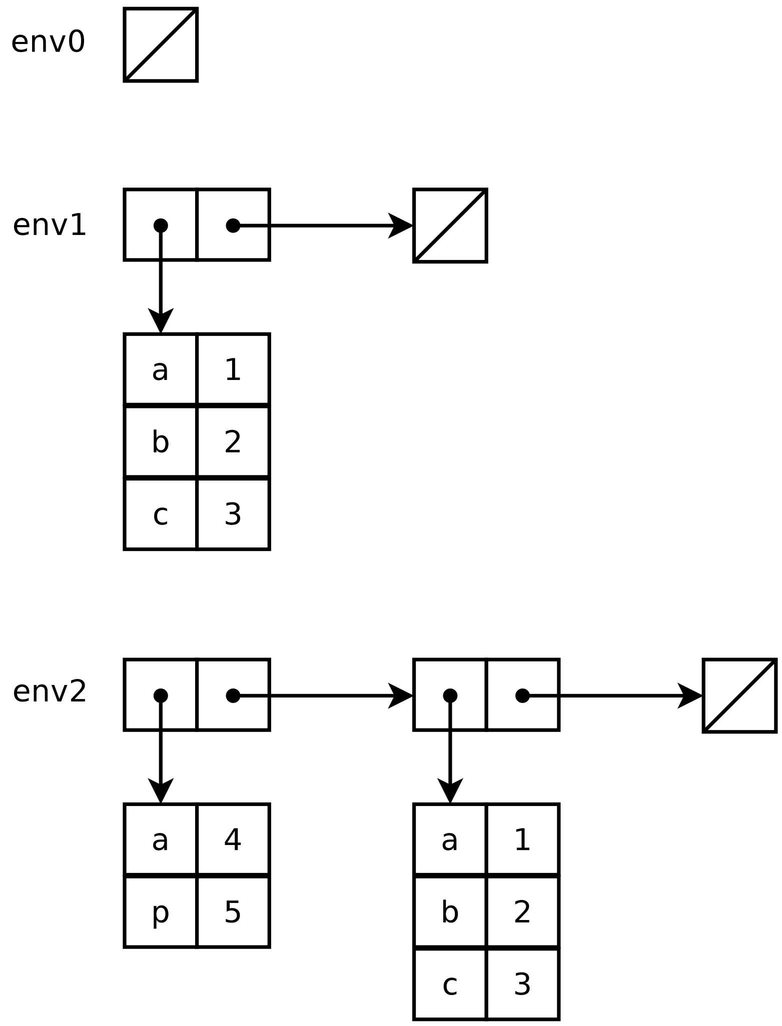
Environments (continued)

Here is a test program in the `Env` class. This program illustrates how to use the two constructor versions in the `Bindings` class.

```
public static void main(String [] args) {
    Env env0 = empty;
    Env env1 = env0.extendEnv(
        new Bindings(Arrays.asList(
            new Binding("a", new IntVal(1)),
            new Binding("b", new IntVal(2)),
            new Binding("c", new IntVal(3)))));
    List<String> i2 = Arrays.asList("a", "p");
    List<Val> v2 = Arrays.asList((Val)new IntVal(4), (Val)new IntVal(5));
    Env env2 = env1.extendEnv(new Bindings(i2, v2));
    System.out.println("env0:\n" + env0.toString(0));
    System.out.println("env1:\n" + env1.toString(0));
    System.out.println("env2:\n" + env2.toString(0));
    System.out.print("a(env2) => "); System.out.println(env2.applyEnv("a"));
    System.out.print("a(env1) => "); System.out.println(env1.applyEnv("a"));
    System.out.print("p(env2) => "); System.out.println(env2.applyEnv("p"));
    System.out.print("p(env1) => "); System.out.println(env1.applyEnv("p"));
}
```

Environments (continued)

We show these environments in diagram form as follows:



Environments (continued)

We can include Java code for environments into our PLCC specification file (usually called `grammar`) in the same way that we include Java code into our Java classes generated by PLCC. However, the environment-related classes `Env`, `EnvNode`, `EnvNull`, `Binding`, and `Bindings` do not appear in our BNF grammar, so PLCC doesn't generate stubs for them automatically.

As noted in Slide Set 1a, PLCC makes it possible to create stand-alone Java source files in exactly the same way as it adds methods to generated files, except that the *entire* code for each stand-alone Java source file – including `import` lines – must appear in the language specification section following the BNF rules.

For example, to create a Java source file named `Env.java`, use the following template:

```
Env
%%%
... code for the entire Env.java source file ...
%%%
```

Environments (continued)

We will encounter language definitions that end up creating dozens of Java source files. To manage these complex languages, we separate the contents of the semantics section of the grammar file into separate files grouped by purpose. In many cases, different languages may share some of the same source files. The semantics section then simply identifies the names of these files using an `include` feature that treats the contents of these files as if they were part of the entire grammar file.

For example, one of our early languages `V1` has the following grammar file structure:

```
# lexical specification
...
%
# BNF grammar
...
%
include code  # BNF grammar semantics
include prim  # primitive operations (PLUS, MINUS, etc.)
include env   # environment code (Env, Binding, etc.)
include val   # value semantics (IntVal, etc.)
```