Environments

In virtually all programming languages, programmers create symb
and associate values with them. We discussed bindings earlier. W
show now is how to implement bindings. Our implementation allo
ment *static scope rules*, since this is the most common binding meth
ming languages today.

An *environment* is a data structure that associates a value with eac
finite set of symbols – that is, it represents a set of bindings. We co
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*
with this simple approach is that the same symbol may have differ
different parts of the program, and this approach doesn't make it cl
termine which binding is the *current* binding.

Instead, we specify an environment as a Java object having a
`applyEnv` that, when passed a symbol (a `String`) as a parame
current value bound to that symbol. So if `env` is an environment and

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

returns the value currently bound to the symbol `x`.

Environments (continued)

If there is no binding for the symbol `x` in the environme `env.applyEnv("x")` throws an exception.

In addition to getting the current value bound to a symbol, our envi mentation provides a way to create an empty environment (one wit and a way to extend an existing environment (essentially to enter a adding new bindings.

But wait: what type does `applyEnv` return? In other words, wh "value"? For the time being, we assume that a "value" is an inst aptly named `Val`. We will refine the notion of "value" later. If about this, just pretend that instances of the `Val` class represent inte

Environments (continued)

Our environments are implemented as instances of a Java abstrac simple version of which we show here:

```
public abstract class Env {

    public static final Env empty = new EnvNull();

    public abstract Env add(Binding b);

    public static Env initEnv() {
        // initial environment with no bindings
        return new EnvNode(new Bindings(), empty);
    }

    public abstract Binding lookup(String sym); // only local

    public abstract Val applyEnv(String sym);

    public Val applyEnv(Token tok) {
        return applyEnv(tok.toString());
    }

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

}
```

Environments (continued)

We represent a binding as an instance of the class `Binding`. A `Bi` has a `String` field named `id` that holds an identifier name (a sym 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 – su floats, and booleans. The only `Val` type we are concerned with at integer value represented by a class `IntVal` that extends the `Val` Think of an `IntVal` as a *wrapper* for the Java primitive type `int`. add new `Val` types as needed to extend functionality.

Environments (continued)

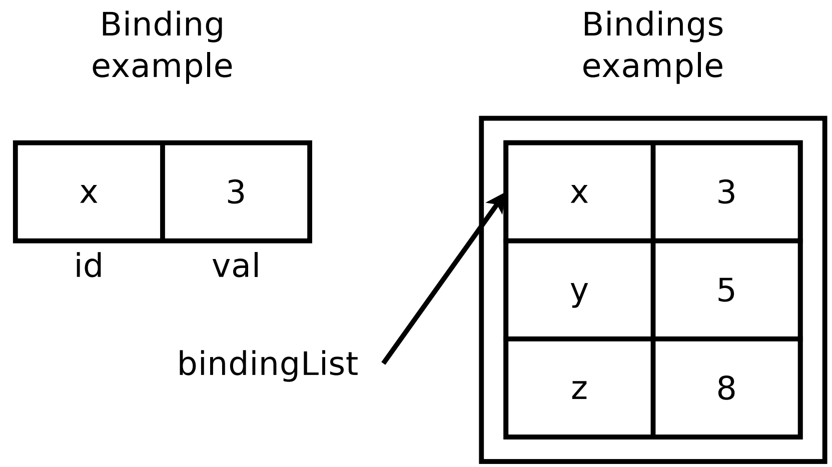
A *local environment* is a list of zero or more bindings. In the co structured languages, you can think of a local environment as capt bindings defined in a particular block. We represent a local environ class `Bindings`. A `Bindings` object has a single field named `b` which is a `List` of `Binding` 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;
    }

    ... methods lookup() and add() are given later

}
```

Environments (continued)

A local environment should not have two different bindings with the although our simple implementation does not enforce this. We will s we can achieve this restriction when we build our local environment

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



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

Environments (continued)

Given a local environment, we want to be able to look up the `Bind` with a particular symbol and to add `Binding` objects to the loca It is also convenient to consider adding a `Binding` obtaind from a value. The following methods are part of the `Bindings` (local envi

```
// look up the Binding associated with a given symbo
public Binding lookup(String sym) {
    for (Binding b: bindingList)
        if (sym.equals(b.id))
            return b;
    return null;
}

// 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)

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

In the `EnvNode` class, making a call to `lookup` for a symbol alwa
ing for it in the local bindings.

When we call `applyEnv` in the `EnvNode` class, we search for the
local bindings using `lookup`:

- if that returns a non-`null` `Binding` object, `applyEnv` return
  in the `Binding`.
- if that returns `null`, `applyEnv` returns the result of calling a
  cursively) on the enclosing environment.

The `EnvNode` class is given on the following slide.

Environments (continued)

```
public class EnvNode extends Env {

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

    public EnvNode(Bindings bindings, Env env) {
        this.bindings = bindings;
        this.env = env;
    }

    public Binding lookup(Symbol sym) {
        return bindings.lookup(sym);
    }

    public Val applyEnv(String sym) {
        Binding b = bindings.lookup(sym);
        if (b == null)
            return env.applyEnv(sym);
        return b.val;
    }

}
```

Environments (continued)

An empty environment is an instance of the class `EnvNull`. An ject has no fields. The `lookup` method in the `EnvNull` class ret `Binding` can be found in the empty environment), and the `apply` this class throws a `PLCCException`.

```
public class EnvNull extends Env {

    // create an empty environment
    public EnvNull() {
    }

    // find the Val corresponding to a given id
    public Val applyEnv(String sym) {
        throw new PLCCException("no binding for "+sy
    }

    public Binding lookup(String sym) {
        return null;
    }

}
```

Environments (continued)

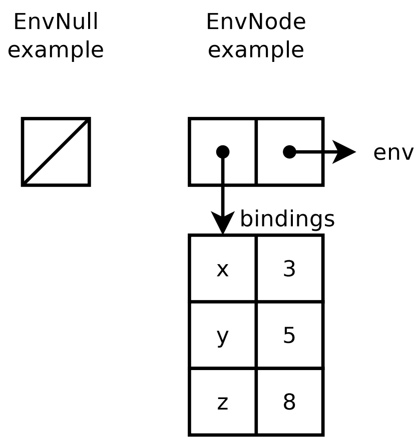
Both the `EnvNode` and `EnvNull` classes define an `add` method – abstract method in the `Env` class – that takes a `Binding` object In the `EnvNode` class, this method adds the binding to its local e the `EnvNull` class, this method throws an exception (because th environment to add to).

Attempting to `add` a binding to an `Env` object should be considere binding's symbol already appears in the local bindings, although t checked in the `Env` classes.

Environments (continued)

The following diagrams give examples of an `EnvNull` object anc
object.

The `EnvNull` object has no fields – we always draw such an enviror
below. The `EnvNode` object in this example has a `bindings` fie
slide 2.6 and an `env` field referring to some enclosing environment



In the right-hand `EnvNode` example, a call to `lookup("x'`
`Binding` object `("x", 3)` (think of this as a pair), a call to `app`
returns an `IntVal` object with value 3, and a call to `lookup("w")`

Environments (continued)

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

The `extendEnv` procedure, defined in the `Env` class, takes a set
ings and uses them to return a new `EnvNode` that becomes the
environment list, extending the current environment list. Here is th
`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 Li
and a List of values by binding each symbol to its correspondi
resulting Bindings object can then be used to extend an environ
constructor for a Bindings object that does this pairing.

```
public Bindings (List<?> idList, List<Val> valList)
    // idList.size() and valList.size() must be the
    bindingList = new ArrayList<Binding>();
    Iterator<?> idIterator = idList.iterator();
    Iterator<Val> valIterator = valList.iterator();
    while (idIterator.hasNext()) {
        String s = idIterator.next().toString();
        Val v = valIterator.next();
        this.add(new Binding(s, v));
    }
}
```

The purpose of the '?' wildcard in the List<?> parameter declara
for a List of either Strings or Tokens.

This constructor relies on the two List parameters having the sam
responsibility of the caller to ensure that this is the case.

Environments (continued)

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

- an environment is a (possibly empty) linked list of local environ
- 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 subcla
and EnvNode), Binding, and Bindings as summarized on the

Environments (continued)

Class/method summary:

```
abstract class Env
    public abstract Binding lookup(String sym)
    public Val applyEnv(String sym)
    public Env extendEnv(Bindings bindings)
    public Env add(Binding b)

class EnvNode extends Env
    public Binding lookup(String sym)

class EnvNull extends Env

class Bindings
    public Binding lookup(String sym)
    public void add(Binding b)

class Binding

abstract class Val

class IntVal extends Val
```

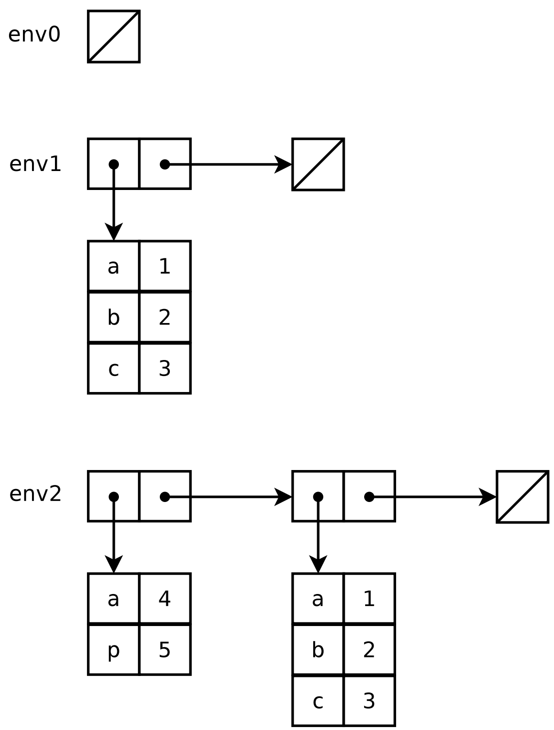
Environments (continued)

Here is a test program in the `Env` class. This program illustrates how to use th
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)
    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(en
    System.out.print("a(env1) => "); System.out.println(en
    System.out.print("p(env2) => "); System.out.println(en
    System.out.print("p(env1) => "); System.out.println(en
}
```

Environments (continued)

We show these environments in diagram form as follows:



Environments (continued)

We can include Java code for environments into our PLCC specificati
called `grammar`) in the same way that we include Java code into c
generated by PLCC. However, the environment-related classes `En`
`EnvNull`, `Binding`, and `Bindings` do not appear in our BN
PLCC doesn't generate stubs for them automatically.

As noted in Slide Set 1a, PLCC makes it possible to create stand-alo
files in exactly the same way as it adds methods to generated files,
entire code for each stand-alone Java source file – including `impor`
class constructor – must appear in the language specification sectio
BNF rules.

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

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

Environments (continued)

We will encounter language definitions that end up creating dozens
files. To manage these complex languages, we separate the content
tics section of the grammar file into separate files grouped by pur
cases, different languages may share some of the same source files.
section then simply identifies the names of these files using an `#in`
that treats the contents of these files as if they were part of the entire

For example, one of our early languages `V1` has the following gran
ture:

```
# lexical specification
...
%
# BNF grammar
...
%
#include code            # BNF grammar semantics
#include prim            # primitive operations (PLU
#include ../Env/envRN    # environment code (Env, B
#include val             # value semantics (IntVal,
```