

Copyright (C) 2021 Timothy Fossum

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. A copy of the license is included in the file "COPYING", entitled "GNU Free Documentation License".

Environments

In virtually all programming languages, programmers create symbols and associate values with them. We discussed bindings earlier. What we 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 symbol in a finite set of symbols – that is, it represents a set of bindings. We can represent an environment as a set of pairs

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

that encode the binding of symbol s_1 to value v_1 , s_2 to value v_2 , etc. One 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 `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

```
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 environment `env`, `env.applyEnv("x")` throws an exception.

In addition to getting the current value bound to a symbol, our environment provides a way to create an empty environment (one without bindings) and a way to extend an existing environment (essentially to enter a new scope by adding new bindings).

But wait: what type does `applyEnv` return? In other words, what is a “value”? For the time being, we assume that a “value” is an instance of the class `Val`. We will refine the notion of “value” later. If you’re not sure about this, just pretend that instances of the `Val` class represent integers.

Environments (continued)

Our environments are implemented as instances of a Java abstract class. A 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 bindings

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

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

Environments (continued)

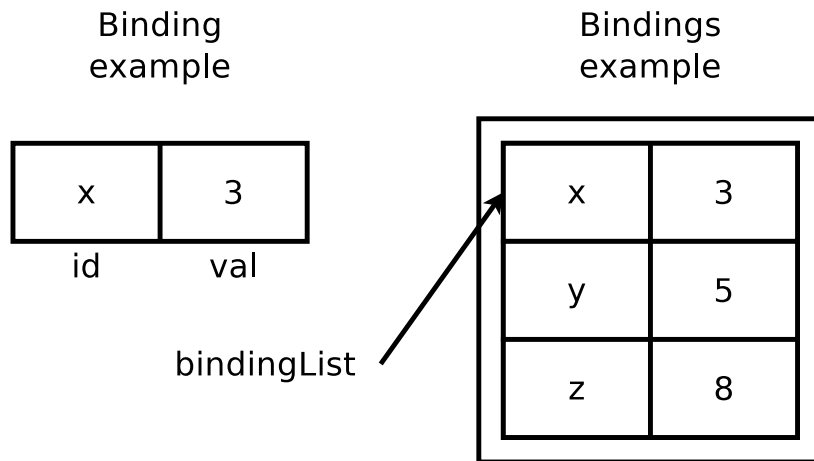
A *local environment* is a list of zero or more bindings. In the context of structured languages, you can think of a local environment as capturing the bindings defined in a particular block. We represent a local environment as a class `Bindings`. A `Bindings` object has a single field named `bindingList`, 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;  
    }  
  
    ... methods lookup() and add() are given later  
  
}
```

Environments (continued)

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

The following diagram gives an example of a Binding object with symbol "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 want to be able to look up the Binding associated with a particular symbol and to add Binding objects to the local environment. It is also convenient to consider adding a Binding obtained from another local environment to a local environment. The following methods are part of the Bindings (local environment) class.

```
// look up the Binding associated with a given symbol
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 object has two fields: a `Bindings` object named `bindings` that contains the local bindings and an `Env` object named `env` that refers to an enclosing environment (static scope rules) environment.

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

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

- if `lookup` returns a non-null `Binding` object, `applyEnv` returns the value in the `Binding`.
- if `lookup` returns `null`, `applyEnv` returns the result of calling `applyEnv` recursively 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 bindings  
    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 object has no fields. The `lookup` method in the `EnvNull` class returns `null` (because no binding can be found in the empty environment), and the `apply` method in this class throws a `PLCCEException`.

```
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 PLCCEException("no binding for "+sym);
    }

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

}
```

Environments (continued)

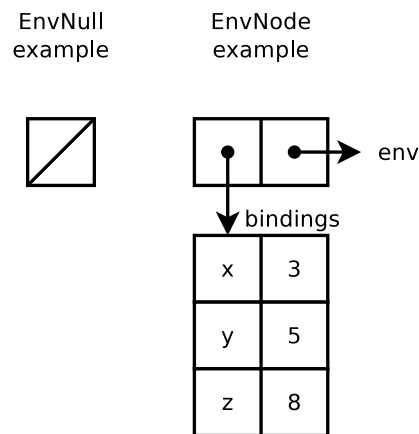
Both the `EnvNode` and `EnvNull` classes define an `add` method – an abstract method in the `Env` class – that takes a `Binding` object as argument. In the `EnvNode` class, this method adds the binding to its local environment. In the `EnvNull` class, this method throws an exception (because there is no environment to add to).

Attempting to add a binding to an `Env` object should be considered an error if the binding's symbol already appears in the local bindings, although this is not checked in the `Env` classes.

Environments (continued)

The following diagrams give examples of an `EnvNull` object and an `EnvNode` object.

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



In the right-hand `EnvNode` example, a call to `lookup("x")` returns a `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 node is an 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 of bindings and uses them to return a new `EnvNode` that becomes the 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` and a `List` of values by binding each symbol to its corresponding value. The resulting `Bindings` object can then be used to extend an environment constructor for a `Bindings` object that does this pairing.

```
public Bindings (List<?> idList, List<Val> valList)
    // idList.size() and valList.size() must be the same
    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 declaration is to allow for a `List` of either `Strings` or `Tokens`.

This constructor relies on the two `List` parameters having the same size. It is the responsibility of the caller to ensure that this is the case.

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 `EnvNode` and `EnvNode`), `Binding`, and `Bindings` as summarized on the next slide.

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
```

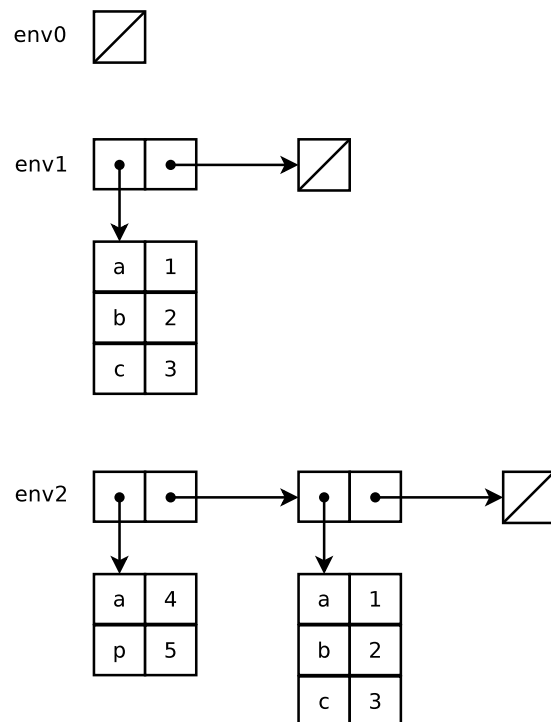
Environments (continued)

Here is a test program in the Env class. This program illustrates how to use the various 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());
    System.out.println("env1:\n" + env1.toString());
    System.out.println("env2:\n" + env2.toString());
    System.out.print("a(env2) => "); System.out.println(env2.lookup("a"));
    System.out.print("a(env1) => "); System.out.println(env1.lookup("a"));
    System.out.print("p(env2) => "); System.out.println(env2.lookup("p"));
    System.out.print("p(env1) => "); System.out.println(env1.lookup("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 (called `grammar`) in the same way that we include Java code into code generated by PLCC. However, the environment-related classes `Env`, `EnvNull`, `Binding`, and `Bindings` do not appear in our BNF. 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, *entire* code for each stand-alone Java source file – including `import` statements and class constructor – must appear in the language specification section of 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 files. To manage these complex languages, we separate the contents section of the grammar file into separate files grouped by purpose. In some cases, different languages may share some of the same source files. The contents section then simply identifies the names of these files using an `include` statement that treats the contents of these files as if they were part of the entire grammar.

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

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