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 will follow 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 will specify an environment as an object as having a method called \texttt{applyEnv} that, when passed a symbol (a \texttt{String}) as a parameter, returns the current value bound to that symbol. So if \texttt{env} is an environment and "x" is a symbol,

\texttt{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 will require a way to create an empty environment (one with no bindings), and a way to add bindings to an existing environment (essentially to enter a new block).

But wait: what type does applyEnv return? In other words, what exactly is a “value”? For the time being, we will 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.]

Our environments will be implemented as instances of a Java abstract class Env:

```java
// default method to return the value currently bound to sym.
// this method will be overloaded in subclasses of Env!
public Val applyEnv(String sym) {
    throw new RuntimeException("no binding for "+sym);
}

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

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

We will discuss the Bindings class next.
Environments (continued)

We will 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.

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

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 will represent a local environment using the class Bindings. A Bindings object has a single field named bindingList which is a List of Binding objects.
Informally, think of an environment as a linked list of local environments.

An empty environment is an instance of the class `EnvNull`. In the empty environment, `applyEnv` will always throw an exception, since no symbol is bound to any value in the empty environment.

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 is the next (enclosing, in the sense of static scope rules) environment in the list.

The `extendEnv` procedure creates a new environment by adding local bindings to the previous environment. Here is the definition of `extendEnv` for both the `EnvNull` and `EnvNode` classes:

```java
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.

```java
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 `Strings` or `Tokens`. 
**Environments** (continued)

The `applyEnv` procedure is now easy. If the environment is `EnvNull`, simply throw an exception. For any other environment (an instance of `EnvNode`) 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, search the next environment.

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

```java
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.

At this point we need to *test* our implementation.
Environments (continued)

For the remainder of these materials, we will 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:

```java
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
```
Here is a test program in the `Env` class:

```java
public static void main(String[] args) {
    Env env0 = new EnvNull();
    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(i2, v2);

    System.out.println("env2:
" + env2.toString(0));
    System.out.println("a(env2) => " + env2.applyEnv("a");
    System.out.println("b(env2) => " + env2.applyEnv("b");
    System.out.println("p(env2) => " + env2.applyEnv("p");
    System.out.println("a(env1) => " + env1.applyEnv("a");
    System.out.println("b(env1) => " + env1.applyEnv("b");
    System.out.println("p(env1) => " + env1.applyEnv("p");
}
```

In `env2`:
- `a` is bound to 4
- `b` is bound to 2
- `p` is bound to 5

In `env1`:
- `a` is bound to 1
- `b` is bound to 2
- `p` is unbound
Environments (continued)

We show environments in diagram form as follows:

env0

env1

env2