Environment-Passing Interpreters

Interpretation vs. compilation can be illustrated by a picture:

Interpreter execution:

character stream (program text)  \rightarrow \text{answer}

Lexical analysis  \rightarrow \text{Scanner}  \rightarrow \text{Syntax analysis}  \rightarrow \text{Parser}  \rightarrow \text{Semantic analysis}  \rightarrow \text{Interpreter}

Compiler execution:

character stream  \rightarrow \text{token stream}  \rightarrow \text{parse tree}  \rightarrow \text{compiled program}

Lexical analysis  \rightarrow \text{Scanner}  \rightarrow \text{Syntax analysis}  \rightarrow \text{Parser}  \rightarrow \text{Semantic analysis}  \rightarrow \text{Code Generator}  \rightarrow \text{Interpreter (HW)}  \rightarrow \text{answer}
Environment-Passing Interpreters (continued)

Programming language grammars have a special syntactic category (a nonterminal) called an *expression*. In Java, for example, an expression typically involves values (like variables, integers, and the results of method calls) and operators (like addition and multiplication). Examples expressions might be `'2+3'` or `'foo(11) && toggle'`. In all of the languages we discuss in this class, expressions will play a principal role. Such languages are called “expression-based languages”.

An *expressed value* is the value of an expression; for example, the expressed value of the arithmetic expression `'2+3'` is 5. A *denoted value* is the value bound to a symbol. Denoted values are internal to the interpreter, whereas expressed values are values that can be seen “from the outside”.

For a symbol, say `x`, you normally think that the value of the expression `x` is the same as the denoted value of `x`. But what about a language such as Java? In Java, the denoted value of a non-primitive variable is a *reference* to an object, whereas the expressed value of the variable is the object itself. This may seem like a subtle distinction, but you will see its importance later.

In summary, for a symbol, its expressed value is what gets displayed when you print it (its `toString` representation), and its denoted value is what gets stored internally in the symbol’s binding. In our early languages, the denoted values and expressed values will be the same. In our later languages, we will see why we need to separate denoted values from expressed values to implement language features such as mutation.
Environment-Passing Interpreters (continued)

You should also distinguish between the defined language (or the source language) and the defining language (or the host language).

The defined language is the language to be interpreted and the defining language is the language in which the interpreter is written.

In the rest of this course, the defined languages will be a collection of artificial languages used to illustrate the various stages of language design, and the defining language will be Java. Don’t be disappointed by the term ‘artificial’ here: the languages we define will have significant computational power, and they will serve to illustrate a number of core ideas that are present in all programming languages.
Language V0

Expressed value: number (an IntVal)
Denoted value: number

# Language V0
skip WHITESPACE '\s+'
LIT '\d+'
ADDOP '\+'
SUBOP '\-'
ADD1OP 'add1'
SUB1OP 'sub1'
LPAREN '\('
RPAREN '\)'
COMMA ','
VAR '[A-Za-z]\w*'
%
<program> ::= <exp>
# the following grammar rules define what it means to be an expression
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
<exp>:PrimAppExp ::= <prim> LPAREN <rands> RPAREN
<rands> **= <exp> +COMMA
<prim>:AddPrim ::= ADDOP
<prim>:SubPrim ::= SUBOP
<prim>:Add1Prim ::= ADD1OP
<prim>:Sub1Prim ::= SUB1OP
%
Language V0 (continued)

Example “programs” in this language:

3
x
+(3, x)
add1(+(3, x))
+(4, -(5, 2))
Language V0 (continued)

Here is a mapping from the concrete (BNF) syntax of the language to its Java representation as an abstract syntax. The Java class files are created automatically by the \texttt{plcc} translator. Each item in a \texttt{Box} is the signature of the corresponding class constructor.

\begin{verbatim}
<program> ::= <exp>
Program(Exp exp)

<exp>:LitExp ::= <LIT>
LitExp(Token lit)

<exp>:VarExp ::= <VAR>
VarExp(Token var)

<exp>:PrimappExp ::= <prim> \texttt{LPAREN} <rands> \texttt{RPAREN}
PrimappExp(Prim prim, Rands rands)

<rands> ::= \texttt{<exp>} \texttt{+COMMA}
Rands(List<Exp> expList)

<prim>:AddPrim ::= \texttt{ADDOp}
AddPrim()

<prim>:SubPrim ::= \texttt{SUBOP}
SubPrim()

<prim>:Add1Prim ::= \texttt{ADD1OP}
Add1Prim()

<prim>:Sub1Prim ::= \texttt{SUB1OP}
Sub1Prim()
\end{verbatim}
The term *abstract syntax* might seem odd because it refers to a collection of very concrete Java classes. Instead, the term *abstract* here means that these classes keep only the information on the right-hand-side (RHS) of the grammar rules that can change, principally by ignoring the RHS tokens. For example, the `PrimappExp` class does not have fields for the `LPAREN` and `RPAREN` tokens.

Because the `<exp>` and `<prim>` nonterminals appear on the LHS of two or more grammar rules, their corresponding grammar rules are disambiguated by annotating their LHS nonterminals with appropriate class names. The resulting classes are defined as extending an abstract class given by the nonterminal name, suitably capitalized to conform to Java naming rules.

For example, in the case of the `<exp>` nonterminal, which appears on the LHS of three grammar rules, the `LitExp`, `VarExp`, and `PrimappExp` classes extend the `Exp` abstract class. Similarly, in the case of the `<prim>` nonterminal, the `AddPrim`, `SubPrim`, `Add1Prim` and `Sub1Prim` classes extend the `Prim` abstract class.

The `Program` class has one instance variable named `exp` of type `Exp`. Since `Exp` is an abstract class, an object of type `Exp` must be an instance of a class that extends `Exp`: namely, an instance of `LitExp`, `VarExp`, or `PrimappExp`. Note that `Exp` does not have a constructor, so you can’t instantiate an object of type `Exp` directly.

The directory

/home/student/Classes/443/Code/V0

contains the grammar for this language.
Language V0 (continued)

The grammar file in Language V0 has three parts, separated by lines with a single %: the lexical specification, the grammar rules, and the code section.

If only the lexical specification is given, the plcc tool will only produce Java code for a scanner (class Scan), but nothing else. If only the lexical specification and grammar rules are given, the plcc tool will produce a scanner and a parser for the grammar, but nothing else.

The code section is the heart of the language semantics. In this section, the Java classes defined by the grammar rules are given life in terms of defining behavior – for example, by defining toString methods. We will presently see how a toString method can be used to print the value an expression.

It is in the code section, then, that semantics takes root.
Language V0 (continued)

Given a grammar file, the plcc tool produces a scanner (a Scan object) and parser (a Parser object) for the grammar. Its companion plccmk script automates both the parser generation and Java compilation.

Assuming that the grammar file has been created in a directory named V0, running the plccmk tool will create a subdirectory named Java and will compile all of the source files created by the plcc tool.

You can see that there are Java source files named Program.java, LitExp.java, and so forth, that correspond to the abstract syntax shown in slide 3.6. You will also see Java source files named Token.java, Scan.java, Parse.java, Parser.java, and Rep.java.

You can run the Parser program as follows:

```
java Parser 'add1( + (2,3))'
```

This will parse the command-line argument “program” `add1( + (2,3))` and will return a String representation of the parsed program – which is, at this point in time, the address of a Program object. If the parse fails, the parser will throw an exception.
You can also run the `Rep` program that will repeatedly prompt you for input (with `--->`), parse the input, and print the result – again, a `String` representation of the parse.

```java
java Rep
---> add1( + (2,3))
...
```

**What exactly does parsing do, and when you parse something, how can you use the result?** We will examine these questions next.

As we discussed in Chapter 1, parsing is the process by which a sequence of tokens (a *sentence*) can be determined to belong to the language defined by the grammar. We showed examples of leftmost derivations and how the derivation process can detect whether or not the sentence is syntactically correct.
In the code generated by the `plcc` tool, we get more than a yes or no answer from our parser: \textit{the plcc parser returns an object that is an instance of the class determined by the grammar start symbol}. This object is the root of what we call a \textit{parse tree} that captures all of the elements of the parsed sentence. (If the parse fails because the sentence is not syntactically correct, the parsing method throws an exception, and no object is returned.)

Every nonterminal in the grammar corresponds to a class whose name is the same as the nonterminal except with its first letter capitalized. For example, the `<program>` nonterminal corresponds to a class named `Program`, and the `<exp>` nonterminal corresponds to the (abstract) class named `Exp`.

An abstract class is created when the nonterminal appears more than once on the left-hand sides of the grammar rules. In this case, the LHS entries in the grammar rules must be annotated with appropriate class names that uniquely determine its LHS. So, for the `<exp>` nonterminal, there is a grammar rule whose LHS is `<exp>`:LitExp, meaning that `LitExp` is a class that extends the `Exp` abstract class and that uniquely identifies this grammar rule. Similar remarks apply to the `<exp>`:VarExp and `<exp>`:PrimappExp rules.

The `plcc` parser for the V0 language returns an instance of the `Program` class, since `<program>` is the start symbol for the grammar.
Language V0 (continued)

We have seen that every grammar rule has a unique class name associated with the rule. When the plcc-generated parser uses a rule to carry out one step in a leftmost derivation of a sentence, an instance of the corresponding class is created.

The RHS of a grammar rule determines what instance variables belong to the class defined by its LHS. Only those entries on the RHS that have angle brackets `<...>` appear as instance variables; any other RHS entries must be token names that are used in the parse but that do not appear as instance variables when generating the objects in the parse tree.

For example, consider the following grammar rule in our VO grammar:

\[
<\text{exp}> : \text{PrimappExp} := <\text{prim}>\ LPAREN\ <\text{rands}>\ RPAREN
\]

This rule says that PrimappExp is a class that extends the Exp class and that the instance variables in this class are

- Prim prim;
- Rands rands;

The names of the instance variables are the same as the nonterminal names, and the instance variable types are the same as the corresponding nonterminal types: just uppercase the first char of the nonterminal name to get its type.
Language V0 (continued)

For another example, consider the following grammar rule in our V0 grammar:

\[<\text{exp}>::=\text{LitExp} \rightarrow <\text{LIT}>\]

The \text{LIT} token name is defined in the lexical specification to be a sequence of one or more decimal digits. Because \text{LIT} is surrounded by angle brackets \(<\ldots>\) on the RHS of this grammar rule, the \(<\text{LIT}>\) item appears as an instance variable in the class \text{LitExp}. The instance variable name is derived from \text{LIT} by converting it to lowercase. Its type (since it’s a token) is \text{Token}. So this rule says that \text{LitExp} is a class that extends the \text{Exp} class, and the instance variable in this class is

\text{Token lit;}

Now you can see exactly what slide 3.6 is saying: each grammar rule defines a class given by its LHS with a well-defined set of instance variables corresponding to the entries in its RHS.

We will encounter a few situations when two RHS entries are defined by the same name in angle brackets. In these cases, we disambiguate the entries by providing different instance variable names.
For a final example, consider the following grammar rule in our V0 grammar:

\[
\langle \text{rands} \rangle \ ::= \langle \text{exp} \rangle +\text{COMMA}
\]

This rule says that the \( \langle \text{rands} \rangle \) nonterminal can derive zero or more \( \langle \text{exp} \rangle \) entries, separated by commas. The following sentences would match the \( \langle \text{rands} \rangle \) nonterminal:

\[
\begin{align*}
a, b, c & \quad \langle \text{exp} \rangle \ \text{COMMA} \ \langle \text{exp} \rangle \ \text{COMMA} \ \langle \text{exp} \rangle \\
1, +(2, 3) & \quad \langle \text{exp} \rangle \ \text{COMMA} \ \langle \text{exp} \rangle \\
ad1(x) & \quad \langle \text{exp} \rangle \\
& \quad \langle \text{empty string} \rangle
\end{align*}
\]

The class defined by this rule is named \text{Rands}. Its RHS shows only one nonterminal \( \langle \text{exp} \rangle \), but since the rule can match zero or more of them, the rule actually can have multiple instances of \( \langle \text{exp} \rangle \). So the \text{plcc} tool defines the \text{Rands} class as having an instance variable that is a \text{List} of \text{Exp} objects. Since the nonterminal name is \text{exp}, the instance variable is appropriately named \text{expList}. Thus the \text{Rand} class has one instance variable:

\[
\text{List<Exp>} \ \text{expList};
\]

This completes our description of how the BNF rules (concrete syntax) in our grammar for language \text{V0} get translated by the \text{plcc} tool into Java classes (abstract syntax), and how the instance variables in these classes are determined.
Language V0 (continued)

When the expression `add1( +(2, 3))` is parsed, an object of type `Program` is returned. The `Program` object has one instance variable: `exp` of type `Exp`. The value of the `exp` instance is an object of type `PrimappExp` (which extends the `Exp` class) that has two instance variables: `prim` of type `Prim` and `rands` of type `Rands`. The value of the `prim` instance is an object of type `Add1Prim` (which extends the `Prim` class) that has no instance variables. And so forth ...

On the following slide we show the entire parse tree of this expression.
Language V0 (continued)

Parse tree for `add1(+ (2, 3))`:

```
Program
  exp: Exp
    exp: PrimappExp
      prim: Prim
      rands: Rands
        prim: Add1Prim
        rands: Rands
          expList: List<Exp>
            expList[0]: PrimappExp
              prim: Prim
              rands: Rands
                prim: AddPrim
                rands: Rands
                  expList: List<Exp>
                    expList[0]: LitExp
                      lit: Token="2"
                      expList[1]: LitExp
                        lit: Token="3"
```
Language V0 (continued)

The Rep program prints the resulting value of the Program object as a string using its default toString method, since the Program class does not redefine the toString method. This default toString method prints the name of the class along with a representation of its memory address in the Java runtime environment, which is not particularly useful. So our next step is to define the toString method for the Program class and all of the other classes defined by this grammar so that the printed toString value of a Program object will be essentially the same string as the input.

This means that we should see the following when interacting with the Rep program from grammar V0:

\[
\begin{align*}
\text{--&gt; add1( +(2,3) )} \\
\text{add1(+(2,3))} \\
\text{--&gt; x} \\
\text{x} \\
\text{--&gt; + ( p , -(q,r) )} \\
\text{+(p,-(q,r))} \\
\text{--&gt; ...}
\end{align*}
\]
Language V0 (continued)

We will use the code section of the grammar (the part that follows the lexical specification and grammar rules) to add our toString methods to our plcc-defined classes (see slide 3.6). When adding methods to a class such as Program, use the following template:

```plaintext
<class name>
{%
<method definitions>
%
}
```

Since these methods become part of the code for the given class, they can access any of the instance variables in the class. So for the Program object, the methods can refer to the exp instance variable of type Exp. Since the RHS of the <program> grammar rule is just <exp>, the toString method of the Program class is simple:

```java
Program
{%
    public String toString() {
        return exp.toString();
    }
%
}
```
There are three `Exp` classes: `LitExp`, `VarExp`, and `PrimappExp`. The first two are particularly easy, since they both have right-hand sides that are just token strings. Thus their `toString` methods simply return the `String` value of the corresponding `Token` instance variables (see slide 3.6):

```java
LitExp

%%{
    public String toString() {
        return lit.toString();
    }
}

VarExp

%%{
    public String toString() {
        return var.toString();
    }
}
```
Examine the rule for a PrimappExp:

```
<exp>:PrimappExp ::= <prim> LPAREN <rands> RPAREN
```

A PrimappExp object has just two instance variables:

```java
Prim prim;
Rands rands;
```

But why aren’t there instance variables corresponding to `LPAREN` and `RPAREN`? We don’t need these instance variables because if we got to this point in the parse, the parser will have consumed these tokens. The only thing we need to do, then, is to re-insert them back into the `toString` result, in the same order as they appear on the RHS of the grammar rule:

```java
PrimappExp
%
%
{ public String toString() { return prim + "(" + rands + ")"; }
%
}
```
Each of the <prim> rules has an RHS that corresponds to a Token that is eaten by the parser. Just as we re-inserted the LPAREN and RPAREN tokens when we defined the toString method in the PrimExp class, each of the <prim> classes will simply return the corresponding string token:

```java
AddPrim

 public String toString() {
   return "+";
 }

SubPrim

 public String toString() {
   return "-";
 }

Add1Prim

 public String toString() {
   return "add1";
 }
```

The Sub1Prim code is similar and has been omitted.
Language V0 (continued)

We have covered all of the `plcc`-generated classes except for `Rands`. This needs a bit more attention since a `Rands` object has a `List` instance variable whose contents must be processed one-by-one. First examine the `<rands>` grammar rule:

```
<rands>  **= <exp> +COMMA
         Rands(List<Exp> expList)
```

To build a `toString` method for this class, we can construct a string from each of the `expList` entries, putting commas between them. Here is the code:

```java
Rands
{%
   public String toString() {
      String s = ""; // the string to return
      String sep = ""; // no separator for the first expression
      // get all of the expressions in the operand list
      for (Exp e : expList) {
         s += sep + e;
         sep = ","; // commas separate the rest of the expressions
      }
      return s;
   }
%
}
```

We can now re-build the Java code for this grammar using the `plccmk` command. Assuming that everything compiles correctly, we should get the desired behavior from the `Rep` program: each syntactically correct input expression will be parsed and re-displayed as a String in the same form as the input, with whitespace removed.
Language V0 (continued)

Note that we can also construct a Program object by hand, essentially building the same structure that the Rep program builds except that we will not depend on the parser to do so.

Let’s create a Test.java program in the V0/Java directory. Our Test.java program will have a main method that will create an instance of a Program object that represents a parse tree for the program \( \text{add1}( + (2, xxx)) \). Our main method will incrementally build the parse tree given on slide 3.16, but starting from the bottom and working up.

First we show how to create a LitExp object with the literal value 2 and a VarExp object with symbol value xxx. Remember that all instance variables corresponding to token names (ones that appear in angle brackets on the RHS) are typed as Tokens. Also, every LitExp object extends the Exp class, so we can treat these as Exp objects; similarly, every VarExp object extends the Exp class, so we can treat these as Exp objects as well.

```java
public static void main(String [] args) {
    Exp e1 = new LitExp("2");
    Exp e2 = new VarExp("xxx");
}
```

[Warning: This uses specially defined constructors for the LitExp and VarExp classes that take String parameters instead of Token parameters.]

Next we build a List<Exp> object with these two Exp objects in the list, and use this to build a Rands object.

```java
public static void main(String [] args) {
    Exp e1 = new LitExp("2");
    Exp e2 = new VarExp("xxx");
    List<Exp> expList1 = Arrays.asList(e1, e2);
    Rands r1 = new Rands(expList1);
}
```
We can now create the PrimappExp toward the bottom right of the parse tree:

```java
public static void main(String [] args) {
    Exp e1 = new LitExp("2"); // note that we use Exp, not LitExp here
    Exp e2 = new VarExp("xxx");
    List<Exp> expList1 = Arrays.asList(e1, e2);
    Rands r1 = new Rands(expList1);
    Exp e3 = new PrimappExp(new AddPrim(), r1);
}
```

The e3 variable must now be packaged into a List (all by itself, sadly) which is used to construct the second Rands object – we’re collapsing two steps into one here:

```java
public static void main(String [] args) {
    Exp e1 = new LitExp("2"); // note that we use Exp, not LitExp here
    Exp e2 = new VarExp("xxx");
    List<Exp> expList1 = Arrays.asList(e1, e2);
    Rands r1 = new Rands(expList1);
    Exp e3 = new PrimappExp(new AddPrim(), r1);
    Rands r2 = new Rands(Arrays.asList(e3));
}
```
Language V0 (continued)

We’re almost done, getting near the top of the parse tree. We create a second `PrimappExp` object from an `Add1Prim` object and the `Rands` object we just created, and finally create a top-level `Program` object from it:

```java
public static void main(String [] args) {
    Exp e1 = new LitExp("2"); // note that we use Exp, not LitExp here
    Exp e2 = new VarExp("xxx");
    List<Exp> expList1 = Arrays.asList(e1, e2);
    Rands r1 = new Rands(expList1);
    Exp e3 = new PrimappExp(new Add1Prim(), r1);
    Rands r2 = new Rands(Arrays.asList(e3));
    Exp e4 = new PrimappExp(new Add1Prim(), r2);
    Program p = new Program(e4);
    System.out.println(p);
}
```

When we run this, we get an output of `add1(+(2,3))`.

Now, would you rather build these parse trees by hand, or would you rather have `Rep` do them for you??
Just for grins, here’s the same thing but without intermediate variables (except p):

```java
public static void main(String[] args) {
    Program p = new Program(
        new PrimAppExp(
            new Add1Prim(),
            new Rands(
                Arrays.asList(
                    (Exp)new PrimappExp(
                        new AddPrim(),
                        new Rands(
                            Arrays.asList(
                                (Exp)new LitExp("2"),
                                (Exp)new VarExp("xxx")))))));
    System.out.println(p);
}
```

This program is in the file named `Test1.java`. 
Now that you see how a parse tree can *print* itself, let’s show how a parse tree can *evaluate itself.*

The term *evaluate* can have many meanings (one of which is to produce a *String* representation of itself), but for our purposes, to evaluate an arithmetic expression such as \[ \text{add1}( + (2, 3)) \] is to produce the value 6. In other words, the value of an expression is its numeric value.

If an expression involves an identifier (symbol), we will need to determine the value bound to that identifier in order to evaluate the expression. For example, suppose the identifier "x" is bound to the integer value 10: then the expression \[ \text{sub1}(x) \] would evaluate to 9. Every expression is *evaluated in an environment.*

The \texttt{Exp} class is the appropriate place to declare evaluation behavior, which we will implement using a method called \texttt{eval}. Here is how we declare the abstract \texttt{eval} method in the \texttt{Exp} class. Every instance of an \texttt{Exp} object must define this method:

\[
\begin{align*}
\text{Exp} & \quad \{ \\
& \quad \quad \text{public abstract Val eval(Env env);} \\
& \quad \} 
\end{align*}
\]

Language \texttt{V1} is the same as language \texttt{V0} except for adding \texttt{eval} methods to the classes that extend the \texttt{Exp} class. We will continue to consider the only \texttt{Val} object to be an \texttt{IntVal} that holds an integer value. The file \texttt{val} in the \texttt{Code/V1} directory has the appropriate definitions.
Three classes extend the Exp class: they are LitExp, VarExp, and PrimappExp. We’ll start with LitExp. Here is the code part of the grammar file that defines the eval behavior of a LitExp object. The eval behavior coexists with the toString behavior that we defined in language V0:

```java
LitExp%%{
    public Val eval(Env env) {
        return new IntVal(lit.toString());
    }

    public String toString() {
        return lit.toString();
    }
}
```

Obviously an environment doesn’t have anything to do with the value of a numeric literal – a 10 evaluates to 10 no matter what environment you have – so the eval routine for a LitExp simply returns the appropriate IntVal object.
Next we consider `VarExp`. Here is the code part of the grammar file that defines the `eval` behavior of a `VarExp` object.

```java
VarExp
{
    public Val eval(Env env) {
        return env.applyEnv(var);
    }

    public String toString() {
        return var.toString();
    }
}
```

A `VarExp` object has a `var` instance variable of type `Token`. Given an environment, the value bound to `var` is precisely the value returned by `applyEnv`, which in turn is the value of the expression.

*The value of an expression consisting of a symbol is the value bound to that symbol in the environment in which the expression is evaluated, as determined by the application of `applyEnv`.***
Finally we consider PrimappExp. A PrimappExp object has two instance variables: a Prim object named prim and a Rands object named rands. To evaluate such an expression, we will need to apply the given primitive operation (the prim object) to the values of the expressions in the rands object.

An object of type Rands has a List<Exp> instance variable named expList. In order to perform the operation determined by the prim object, we will need to evaluate each of the expressions in expList. A utility method named evalRands in the Rands class does the work for us. Of course, this method needs to know what environment is being used to evaluate the expressions, so an Env object is a parameter to this method. (The toString method in the Rands class is omitted from this listing.)

```java
Rands
{
    public List<Val> evalRands(Env env) {
        List<Val> args = new ArrayList<Val>();
        for (Exp e : expList)
            args.add(e.eval(env));
        return args;
    }
}
```
The `evalRands` method returns a list of `Vals`. In order to access these values easily and to apply normal arithmetic operations to them, we convert them into an array of `Val` objects. The utility method named `toArray` in the `Val` class accomplishes this.

The expressions appearing in an application of a primitive are called its operands; the values of these expressions are called its arguments.

We now have the pieces necessary to define the `eval` method in the `PrimappExp` class:

```java
public String toString() {
    return prim + "(" + rands + ")";
}

public Val eval(Env env) {
    // evaluate the terms in the expression list
    // and apply the prim to the array of integer results
    List<Val> args = rands.evalRands(env);
    Val [] va = Val.toArray(args);
    return prim.apply(va);
}
```

What’s left is to define the behavior of the `apply` method in the various `Prim` classes. Observe that by the time the `Prim` object gets its arguments, the environment no longer plays a role, since the arguments are already evaluated.
Since we are using the `apply` method with a `Prim` object, we need to add a declaration for this method to the `Prim` class. Here is how we do this:

```java
Prim
{%
    // apply the primitive to the passed values
    public abstract Val apply(Val [] va);
%
}
```

A `Prim` object (there are four instances of this class) has no instance variables. However, we can endow these objects with behavior, so that a `AddPrim` object will know how to add things, a `SubPrim` object will know how to subtract things, and so forth.

Two of the `Prim` objects need two arguments (for `+` and `−`), and two of them need one argument (for `add1` and `sub1`). Since `va` is an array of `Val` arguments, we can grab the appropriate items from this array – one or two of them, depending on the operation – to evaluate the result. Here is the code for the `AddPrim` class:

```java
AddPrim
{%
    public String toString() {
        return "+";
    }

    public Val apply(Val [] va) {
        if (va.length != 2)
            throw new RuntimeException("two arguments expected");
        int i0 = ((IntVal)va[0]).val;
        int i1 = ((IntVal)va[1]).val;
        return new IntVal(i0 + i1);
    }
%
}
The definition of apply for the SubPrim is entirely similar.

For the Add1Prim class, the apply method expects only one value, which is passed as element zero of the va array.

```java
Add1Prim
  
  public String toString() {
    return "add1";
  }

  public Val apply(Val [] va) {
    if (va.length != 1)
      throw new RuntimeException("one argument expected");
    int i0 = ((IntVal)va[0]).val;
    return new IntVal(i0 + 1);
  }
```

Again, the definition of apply for the Sub1Prim class is entirely similar.
Language V1 (continued)

The final step is to have the `toString` method of a `Program` object return the string representation of the value of its expression. An empty environment would only allow for integer expressions (no variables) since every variable would be unbound. Instead, we create an initial environment that has some variable bindings:

```
i => 1
v => 5
x => 10
l => 50
c => 100
d => 500
m => 1000
```

This environment can be obtained by a call to `Env.initEnv()`. These bindings will at least give us some variables to play with.

Here is the new `toString` method for the `Program` object:

```
Program

%%{
    public static Env initEnv = Env.initEnv();

    public String toString() {
        return exp.eval(initEnv).toString();
    }

%%}
```

To test this, run the `Rep` program in the `Java` directory and enter expressions at the prompt:

```
(cd Java ; java Rep)
```
Language V2

Language V2 is the same as language V1 with the addition of the syntax and semantics of an if expression. Here is the relevant grammar rule and abstract syntax representation:

\[
<\text{exp}> : \text{IfExp} ::= \text{IF} <\text{exp}>\text{testExp} \text{THEN} <\text{exp}>\text{trueExp} \text{ELSE} <\text{exp}>\text{falseExp}
\]

\[
\text{IfExp(Exp testExp, Exp trueExp, Exp falseExp)}
\]

Notice that we need to change our lexical specification to allow for token names IF, THEN, and ELSE.

The RHS of this grammar rule has three occurrences of the <exp> nonterminal. Since the <...> items on the RHS of a grammar rule define the instance variables of the class, we need a way to distinguish among those items having the same name. As you can intuit from the grammar rule, these instance variables will be called testExp, trueExp, and falseExp respectively. Each of these objects will refer to an instance of the Exp class. Thus the IfExp class has three instance variables:

\[
\begin{align*}
\text{Exp testExp;} \\
\text{Exp trueExp;} \\
\text{Exp falseExp;}
\end{align*}
\]

To evaluate an if expression with a given environment, we first evaluate the testExp expression. If this evaluates to true, we evaluate the trueExp expression and return its result as the value of the entire expression. If this evaluates to false, we evaluate the falseExp expression and return its result. Each of these expressions will be evaluated in the given environment.

Since all instances of Val are really IntVals (for the time being), we will regard the IntVal object corresponding to 0 to be false and all others to be true.
We define an `isTrue` method for an `IntVal` object as follows. This code is part of the `IntVal` class that is defined in the `val` file – only the definition of `isTrue` is given here:

```java
public boolean isTrue() {
    return val != 0;
}
```

Since we are applying the `isTrue` method to a `Val` object, we must include the abstract prototype for `isTrue` in the definition of the `Val` base class, which will look like this in the `Val` file.

```java
Val
%%%{
...
    public abstract boolean isTrue();
...
%%%}
```
Language V2 (continued)

Here is the eval code for the IfExp class in the grammar file:

```java
IfExp
%%{
    public Val eval(Env env) {
        Val v = testExp.eval(env);
        if (v.isTrue())
            return trueExp.eval(env);
        else
            return falseExp.eval(env);
    }

    public String toString() {
        return "if " + testExp + " then " + trueExp + " else " + falseExp;
    }
}%%
```

Notice how we use the isTrue boolean method applied to an instance of Val.

In the code for IfExp, observe that only one of the trueExp or falseExp expressions gets evaluated, not both. This is a semantic feature – not a syntax feature – of the definition of eval for this object. The term special form is sometimes used for semantic structures that look like expressions but that, when evaluated, don’t evaluate all of their constituent parts. An if expression is an example of a special form.

Some examples of if expressions are on the next slide.
Language V2 (continued)

if 1 then 3 else 4  
   => 3

if 0 then 3 else 4  
   => 4

if
   if 1 then 0 else 11 then
      42
else
   15
   => 15

+(3, if -(x,x) then 5 else 8)  
=> 11

You must understand that an if expression is an expression and therefore it evaluates to a value. It is entirely unlike if statements in imperative languages such as Java and C++, where the purpose of an if statement is to do one thing or another, not to return a value.
Language V3

Language V3 is the same as language V2 with the addition of a `let` expression. Here are the relevant grammar rules and abstract syntax representations:

\[
\text{<exp>:LetExp ::= LET <letDecls> IN <exp>}
\]

\[
\begin{align*}
\text{LetExp} & \quad \text{LetExp(LetDecls letDecls, Exp exp)} \\
\text{<letDecls> **= <VAR> EQUALS <exp>}
\end{align*}
\]

\[
\begin{align*}
\text{LetDecls} & \quad \text{LetDecls(List<Token> varList, List<Exp> expList)}
\end{align*}
\]

Notice that we will need to change our lexical specification to allow for token names `LET`, `IN`, and `EQUALS`. Here are the relevant lexical specifications:

- `LET` `'let'`
- `IN` `'in'`
- `EQUALS` `'= '`

Here is an example program in language V3 that evaluates to 7:

```
let
  three = 2
  four = 5
in
  +(three, four)
```
Language V3 (continued)

\[
\text{LetExp} ::= \text{LET } \langle \text{letDecls} \rangle \ \text{IN} \ \langle \text{exp} \rangle \\
\text{LetExp(} \langle \text{letDecls} \rangle, \langle \text{exp} \rangle \text{)} \\
\langle \text{letDecls} \rangle ::= \langle \text{VAR} \rangle \ \text{EQUALS} \ \langle \text{exp} \rangle \\
\text{LetDecls(} \langle \text{VAR} \rangle, \langle \text{exp} \rangle \text{)} \\
\text{LetDecls(List(Token) varList, List(Exp) expList)}
\]

To evaluate a LetExp, we perform the following steps:

1. create a set of local bindings by binding each of the \langle \text{VAR} \rangle symbols to the values of their corresponding \langle \text{exp} \rangle expressions in the \langle \text{letDecls} \rangle part, where the \langle \text{exp} \rangle expressions to the right of the EQUALS are all evaluated in the current environment;

2. extend the current environment with these local bindings to create a new environment; and

3. use this new environment to evaluate the \langle \text{exp} \rangle expression in the LetExp, and return this value as the value of the letExp expression.

The \langle \text{exp} \rangle part of a let expression is called the body of the expression.

Some examples if let expressions are on the next slide, where \( \Rightarrow \) means “evaluates to”.
Language V3 (continued)

```plaintext
let x = 3  y = 8
in  +(x, y)
   => 11

let z = 3  y = 8
in  +(x, y)
   => 18 (the enclosing (initial) environment binds x to 10)

let x = 3
in
  let
    x = add1(x)
    y = add1(x)
  in
    +(x, y)
   => 8
```

In the first example, a new environment is created binding \( x \) to 3 and \( y \) to 8, so that the \( +(x, y) \) expression evaluates to 11. Notice that this binding shadows the initial environment’s binding of \( x \) to 10.

In the second example, a new environment is created binding \( z \) to 3 and \( y \) to 8. The initial environment has \( x \) bound to 10, so that the \( (x, y) \) expression evaluates to 18.

In the third example, two new environments are created. The first binds \( x \) to 3 (shadowing the initial binding). The inner environment binds \( x \) to the value of \( \text{add1}(x) \) and \( y \) to the value of \( \text{add1}(x) \). In both of these, \( \text{add1}(x) \) is evaluated using the enclosing environment which has \( x \) bound to 3. Thus \( \text{add1}(x) \) evaluates to 4, in both cases. Thus, in the inner environment, \( x \) is bound to 4 and \( y \) is bound to 4, so that the \( +(x, y) \) expression evaluates to 8.
The code for `eval` in the `LetExp` class is straightforward:

```java
public Val eval(Env env) {
    Env nenv = letDecls.addBindings(env);
    return exp.eval(nenv);
}
```

```java
public String toString() {
    return "... LetExp ...";
}
```

The `addBindings` method applied to the `letDecls` object returns an `Env` object. This new environment extends the `env` environment by adding the bindings given in the `let` declarations. This extended environment is then used to evaluate the body of the `let` expression.
Observe that a `LetDecls` object has two instance variables: `varList` is a list of tokens representing the `<VAR>` part of the grammar rule, and `expList` is a list of expressions representing the `<exp>` part of the grammar rule. (The reason that these are `List`s is because the grammar rule has a `***=` instead of a `::=`.)

The idea is to evaluate each of the expressions in `expList` and bind them to their corresponding token strings in `varList`. These bindings are then used to extend the enclosing environment given by the `env` parameter, and this environment is returned to the `eval` method in the `LetExp` class.

By coincidence, the `Rands` object already has an `evalRands` method that evaluates each of the expressions in its `expList` instance variable, so we simply re-use the `Rands` class and its `evalRands` method here.

```java
LetDecls

%%{
    public Env addBindings(Env env) {
        Rands rands = new Rands(expList);
        List<Val> valList = rands.evalRands(env);
        Bindings bindings = new Bindings(varList, valList);
        return env.extendEnv(bindings);
    }
}
```

Since we can now introduce any bindings to variables that we choose using a `let` expression, our initial environments will henceforth be empty, without bindings for `i`, `v`, and so forth.
The languages we have discussed do not allow mutation of variables, although you might be tempted to think that this V3 program is doing something akin to mutation:

```plaintext
let
  x = 3
in
  let
    x = add1(x)
  in
    +(x, x)
```

This program evaluates to 8 (which is not surprising), but in the scope of the outer `let`, the variable `x` is still bound to 3. To see this, consider the following variant of this program:

```plaintext
let
  x = 3
in
  +(let x = add1(x) in x, x)
```

The last occurrence of `x` in this expression evaluates to 3. This is because the variable `x` in the inner `let` has scope only through the inner `let` expression body. Outside of the inner `let` expression body, the binding of `x` to 3 is unchanged. Thus the entire expression evaluates to 7.
Here are a couple of other observations you should pay attention to.

- In a `<letDecls>` derivation, each `<VAR>` symbol is called the left-hand side (LHS) of the binding and the corresponding `<exp>` is called its right-hand side (RHS). (Don’t confuse this with the LHS and RHS of the grammar rule.) All of the RHS expressions in a `LetDecls` are evaluated in the enclosing environment. The LHS `<VAR>` variables are bound to their corresponding RHS expression values after all of the RHS expressions have been evaluated. Thus the following expression

```plaintext
let
  p = 4
in
  let
    p = 42
    x = p
  in
    x
```

evaluates to 4

- An expression such as `add1(x)` does not modify the binding of `x`. In other words, `add1(x)` is treated like `x+1` in Java instead of `++x`. 

```plaintext
add1(x)
```
So far our languages do not allow for anything like repetition. In an expression-based language (ours fall into this category), repetition is typically accomplished by recursion, and recursion depends on the ability to apply procedures recursively. So we need to build the capability to define procedures.

In language V4, we will add procedure definitions and procedure application. The term *procedure* is synonymous with *function*.

Think of a procedure as a “black box” that, when given zero or more input values, returns a single result value. The number of inputs that a procedure accepts is called its *arity*.

To *define* a procedure means to describe how it behaves. To *apply* a procedure means to give the procedure the proper number of inputs and to receive its result.

Using mathematical notation, we can *define* a function $f$ by

$$f(x) = x + 3$$

and we can *apply* the function $f$ by

$$f(5)$$

The result of this particular application is 8.
In language V4, procedures will be treated as values just like integers. In particular, we will create a ProcVal class that extends the Val class. This means that a ProcVal object can occur anywhere a Val object is expected.

Here is an example of a V4 program that includes a procedure definition and application.

```v4
let
    f = proc(x) +(x,3)
in
    .f(5)
```

A procedure definition starts with the PROC token, and a procedure application starts with a DOT. It is possible that you can define and apply a procedure in one expression, such as

```v4
  .proc(x) +(x,3) (5)
```

Both of these expressions return the same value, namely the integer 8. Notice, too, that

```v4
  proc(x) +(x,3)
```

also returns a value, but the value is a procedure, not an integer. (One’s intent when defining a procedure is eventually to apply it, although this is not a requirement.)
Here are some examples of V4 programs using procedures:

```plaintext
let
  f = proc(x,y) +(x,y)
in
  .f(3,8)
=> 11

let
  f = proc(z,y) +(10,y)
in
  .f(3,8)
=> 18

let x = 10
in
  let
    x = 7
    f = proc(y) +(x,y)
in
    .f(8)
=> 18
```

In the third example, the \texttt{x} in the \texttt{proc} definition refers to the enclosing \texttt{x} (which is bound to 10), not to the inner \texttt{x} that is bound to 7. (Remember the rules for evaluating the \texttt{letDecl}s!)
Language V4 (continued)

Now consider the following examples, all of which evaluate to 5:

```
let
    app = proc(f,x) .f(x)
    g = proc(y) add1(add1(y))
in
    .app(g,3)

let
    app = proc(f,x) .f(x)
in
    .app(proc(y) add1(add1(y)), 3)

    .proc(f,x) .f(x) (proc(y) add1(add1(y)), 3)
```

In the first example, observe that we can pass a procedure (in this case `g`) as a parameter to another procedure. This `app` procedure takes two parameters and returns the result of applying the first parameter to the second. Of course, the first parameter had better be a procedure for this to work. (If it isn’t, an attempt to evaluate the expression will throw an exception.)

In the second example, we have eliminated the identifier `g` and instead simply replaced `g` in the application `.app(g,3)` with the nameless procedure `proc(y) add1(add1(y))` that used to be called `g`.

In the third example, we have even eliminated the identifier `app`.
Language V4 (continued)

Finally consider the following example, which evaluates to 120:

```plaintext
let
    fact = proc(f,x)
    if x
        then *(x,.f(f,-(x,1)))
    else 1
in
    .fact(fact, 5)
```

This example, quite a bit more subtle than the previous ones, shows how you can achieve recursion – factorial, in this case – using our simple language (which does not yet support direct recursion!).
We are now prepared to add syntax and semantics to support procedures. First we add grammar rules for procedure definition and application and display their corresponding abstract syntax classes:

\[ <\text{exp}>: \text{ProcExp} ::= <\text{proc}> \]
\[ \text{ProcExp}(\text{Proc proc}) \]

\[ <\text{proc}> ::= \text{PROC} \ LPAREN \ <\text{formals}> \ RPAREN \ <\text{exp}> \]
\[ \text{Proc}(\text{Formals formals}, \text{Exp exp}) \]

\[ <\text{formals}> ::= \star \star = <\text{VAR}> +\text{COMMA} \]
\[ \text{Formals}(\text{List<Token> varList}) \]

\[ <\text{exp}>: \text{AppExp} ::= \text{DOT} <\text{exp}> \ LPAREN <\text{rands}> \ RPAREN \]
\[ \text{AppExp}(\text{Exp exp}, \text{Rands rands}) \]

Before we can go any further, we need to tackle the notion of a ProcVal, which is what we should get when we evaluate a ProcExp expression.

A ProcVal object must capture the formal parameters as an instance of the Formals class, and it must remember its procedure body as an instance of Exp. But what environment should we use to evaluate the procedure body when the procedure is applied? In order to conform to our notion of static scope rules, we want to evaluate the procedure body using the environment in which the procedure is defined. So any variables in the procedure body which are not among the formal parameters will be bound to their values in the environment in which the procedure is defined.

The term closure is used to describe a ProcVal object. A closure captures all the ingredients necessary to evaluate the procedure once the values that are to be bound to the formal parameters (the arguments) are known: the formal parameters, the procedure body expression, and the environment in which the procedure is defined.
To apply a procedure, bind its formal parameters to its arguments (the values of its actual parameters), extend the captured environment with these bindings, and evaluate the body of the procedure in this extended environment. The result of this evaluation is the value of the application.

The `apply` method in the `ProcVal` class does this work. The principal pieces of this class appear here:

```java
public class ProcVal extends Val {

    Formals formals;
    Exp body;
    Env env;

    public ProcVal(Formals formals, Exp body, Env env) {
        this.formals = formals;
        this.body = body;
        this.env = env;
    }

    public Val apply(List<Val> args, Env e) {
        Bindings bindings = new Bindings(formals.varList, args);
        Env nenv = env.extendEnv(bindings);
        return body.eval(nenv);
    }
}
```
Language V4 (continued)

Let’s now examine the detailed semantics of a ProcExp.

```plaintext
<exp>:ProcExp ::= <proc>
ProcExp(Proc proc)

<proc> ::= PROC LPAREN <formals> RPAREN <exp>
Proc(Formals formals, Exp exp)

<formals> ::= <VAR> +COMMA
Formals(List<Token> varList)
```

As noted in the previous slide, a ProcVal closure is constructed with instance variables consisting of a Formals object (a list of formal parameters), the procedure body in an Exp object, and the environment in which the procedure is defined in an Env object.

The makeClosure method in the Proc class creates a ProcVal object given an environment.

```java
public Val makeClosure(Env env) {
    return new ProcVal(formals, exp, env);
}
```

The semantics of the eval method in the ProcExp class is now trivial:

```java
public Val eval(Env env) {
    return proc.makeClosure(env);
}
```
The remaining implementation issue is how to apply a procedure – in other words, how to evaluate an instance of the AppExp class. Since we already have the `apply` method in the `ProcVal` class, the `eval` method in the `AppExp` class only needs to evaluate the operand expressions and pass them along to the `ProcVal` object for evaluation. (The operand expressions are called the *actual parameters* and their corresponding values are called the *arguments*.)

Here’s the code:

```java
AppExp

public Val eval(Env env) {
    Val v = exp.eval(env);  // should be a ProcVal
    List<Val> args = rands.evalRands(env);
    v.apply(args, env);
}
```

Notice that the operand expressions (the `rands`) are evaluated in the environment in which the expression is applied. Also, the current environment `env` is passed as the second parameter to the `apply` method in the `ProcVal` class, even though the `apply` method does not actually use this value.

What happens if `v` in the above code doesn’t evaluate to a `ProcVal`? All we need to do is to define the default behavior of `apply` in the `Val` class to throw an exception.
Once we have procedures, we can entirely eliminate the `let` construct! Here’s an example:

```plaintext
let
   p = 3
   q = 5
in
   +(p, q)
```

This can be re-written as an application of an anonymous (un-named) procedure as follows:

```plaintext
.proc(p, q) +(p, q) (3, 5)
```

In general, a `let` construct in the form

```plaintext
let
   v1 = e1
   v2 = e2
   ...
in
   e
```

can be re-written in the form

```plaintext
.proc(v1, v2, ...) e (e1, e2, ...)
```

So why not ditch the `let` construct? The reason is simple: it’s easier to think about a program with a `let` in it than one without. This is an example of syntactic sugar: a syntactic and semantic construct that has another way of expressing it in the language but that programmers find easier to understand and use.
Though this language does not support direct recursion, its support of procedures as first-class objects is as powerful as recursion. Here is another example that recursively computes factorials using an “accumulator” and tail recursion:

```
let
  fact = proc(x)
  let
    fact = proc(g, x, acc)
    if zero?(x)
      then acc
    else .g(g, sub1(x), *(x, acc))
  in
    .fact(fact, x, 1)
  in
    .fact(5)
```
Language V4 (continued)

Finally, language V4 will include the ability to evaluate a sequence of expressions, returning the value of the last expression. This does not have any particular usefulness now because our language is not *side-effecting*, but it will turn out to be useful in later languages that do support side-effects.

\[
\text{SeqExp} ::= \{ \text{exp} \} \text{SeqExps} \\
\text{SeqExp} \text{Exp exp, SeqExps seqExps}
\]

The semantics of evaluating a SeqExp are given here:

\[
\text{SeqExp} \text{%%}{
\text{public Val eval(env) \{ \\
Val v = exp.eval(env); \\
for (Exp e : seqExps.expList) \\
 v = e.eval(env); \\
return v;
\}}
\]

Observe that we evaluate every expression in the list but only return the last value.

\[
\{1;3;5\} \\
=> 5
\]

\[
\{42\} \\
=> 42
\]
The sequence construct can be used to enclose a single expression that might otherwise look too unwieldy. Here’s an example:

\[ \{ \text{proc}(t, u) + (t, u) \} (3, 4) \]

The braces in this expression are not required, but they help to visualize the scope of the \text{proc}. 
We normally prefer to use direct recursion instead of using the contrived (but workable) tricks on slide 3.55. For example, we would like to write

```plaintext
let
    fact = proc(x) if zero?(x) then 1 else *(x, .fact(sub1(x)))
in
    .fact(5)
```

But this will not work! Why??

Remember that in a `let`, the RHS expressions (the expressions to the right of the ‘=’ tokens) are all evaluated in the environment that encloses the `let`; only after all the RHS expressions have been evaluated do we bind each of the LHS symbols to their RHS values.

In the definition of the `proc` above, the `proc` body refers to the identifier `fact`, but this identifier is not bound to a value in the enclosing environment. Thus an attempt to apply the `proc` will fail because of an unbound identifier.
In order to solve this problem, we will create a new let-like environment that supports direct recursion. Called letrec, it will allow us to define procedures that support direct recursion. Unlike let, letrec will only support bindings of identifiers to procedures.

This is what we want:

```plaintext
letrec
  fact = proc(x) if zero?(x) then 1 else *(x, .fact(sub1(x)))
in
  .fact(5)
=> 120
```
Here are the grammar rules and associated abstract syntax classes:

```
<exp>:LetrecExp ::= LETREC <letrecDecls> IN <exp>
LetrecExp(LetrecDecls letrecDecls, Exp exp)

<letrecDecls> ::= <VAR> EQUALS <proc>
LetrecDecls(List<Token> varList, List<Proc> procList)
```

The environment in which each proc of a letrec is evaluated should include bindings of each of the variables in the letrec to their corresponding closures. This is unlike a normal let, in which the values to which the variables are bound are evaluated in the enclosing environment.

The big question is, how can a closure (the value of a Proc object) capture an environment that hasn’t been created yet?

Here are the steps:

1. Extend the enclosing environment with null bindings. This will simply serve as a place-holder. Call this new environment nenv.

2. Create a Val list (actually a ProcVal list) consisting of the closures of each proc in procList (procList is a field in the letrecDecls object), where these closures capture the environment nenv. Use this list to create a Bindings object that binds each of the values in this list to its corresponding identifier in varList (varList is a field in the letrecDecls object).

3. Replace the null place-holder in the nenv object with the bindings created in the previous step.

Once the new bindings are part of the nenv environment, all of the closures that captured this environment will be able to access the other procedure identifiers defined in the letrec (including themselves, recursively), and we evaluate the body of the letrec in this modified environment.
**Language V5 (continued)**

Similar to the `LetDecls` class, the `LetrecDecls` class will take care of adding the appropriate bindings to the previous environment by following the steps described in the previous slide.

```java
public Env addBindings(Env env) {
    // Step 1
    Env nenv = env.extendEnv(null); // place-holder
    // Step 2
    List<Val> valList = new ArrayList<Val>();
    for (Proc p: procList)
        valList.add(p.makeClosure(nenv));
    Bindings bindings = new Bindings(varList, valList);
    // Step 3
    nenv.replaceBindings(bindings);
    return nenv;
}
```
We can now evaluate a `LetrecExp` object in exactly the same way as a `LetExp` object:

```java
LetrecExp
    %%%{
        public Val eval(Env env) {
            Env nenv = letrecDecls.addBindings(env);
            return exp.eval(nenv);
        }
    }%
```

The principal idea, then, is to create the RHS closures of a `letrec` in an environment that (ultimately) includes all of the bindings in the `letrec`. 
This picture illustrates the three steps for creating the environment `nenv` used to evaluate the body of the following `letrec` example:

```
letrec
  f = proc(x) ... 
  g = proc(y) ... 
in ... 
```

1. Create a new environment `nenv` by extending the old environment with null bindings;
2. Create a `Bindings` object that binds the LHS identifiers (f and g in this example) to closures of the RHS procedures, where these closures capture the extended environment `nenv` created in Step 1;
3. Replace the null bindings of `nenv` with the bindings created in Step 2.
So far, our defined language has no capability to define top-level procedures that persist from one expression evaluation to another. We would like to extend the language to allow for such definitions.

We already have the tools to do this, since the Env class has methods add and addFirst that makes it possible to add bindings to the current local environment without having to extend it. All we need to do is to add bindings to the *initial environment*, the environment that all expressions in the defined language extend from.

The initial environment of our languages is a **static** Env env variable in the Program class, obtained from the initEnv static method in the Env class. Notice that this initial environment starts out having an empty list of bindings. Our strategy for making top-level definitions is to take advantage of the addFirst method in the Env class. This method takes a Binding object and adds it at the beginning of the Bindings object in the current environment. When applied to the initial environment, we can add bindings to the top-level environment that will be known in any subsequent expression evaluation.
Since a “program” can now have two forms – a top-level “define” or an expression evaluation, we need to have two grammar rules for the <program> nonterminal. Here are their grammar rules and corresponding abstract syntax classes:

\[
\begin{align*}
\text{<program>}:\text{Define} & \quad ::= \quad \text{DEFINE} \quad \text{VAR} \quad \text{EQUALS} \quad \text{exp} \\
& \quad \Rightarrow \quad \text{Define}(\text{Token var, Exp exp}) \\
\text{<program>}:\text{Eval} & \quad ::= \quad \text{exp} \\
& \quad \Rightarrow \quad \text{Eval}(\text{Exp exp})
\end{align*}
\]
Language V6 (continued)

Here is an example of expressions that use the define feature in our defined language:

```plaintext
define i = 1
define ii = add1(i)
define iii = add1(ii)
define v = 5
define iv = subl(v)
define x = 10
define f = proc(x) if zero?(x) then 1 else *(x, f(.g(x)))
    .f(v) % ERROR: g is unbound
define g = proc(x) subl(x)
    .f(v) % OK: g is now bound
    .f(iii)
```

The expression `.f(iii)` evaluates to 6 (3 factorial), and the expression `.f(v)` (second invocation) evaluates to 120 (5 factorial). As long as you stay in the Rep loop, the defined variable bindings will be remembered.

Notice that, in the definition for `f`, the body of the procedure refers to a procedure named `g`, but `g` hasn’t been defined yet. The attempt, in the next line, to evaluate `.f(v)` will fail. After defining `g` on the next line, evaluating `.f(v)` works. This is because by the time you attempt to apply `f` the second time, the `g` procedure has been defined, and the body of `f` will recognize its definition. This only works for top-level defines and cannot be used in expressions.
Notice that for top-level procedure definitions, `define` works the same way as `letrec` in terms of being able to support direct recursion. This is because every top-level procedure definition captures (in a closure) the initial environment, which gets modified every time another top-level definition is encountered. When a new binding is added to the top-level environment, all of the top-level closures will then know this binding, as well as any others that may crop up later! Thus the following works:

```scheme
define even? = proc(x)
    if zero?(x) then 1 else .odd?(sub1(x))

.even?(11) => Error: unbound procedure odd?

define odd? = proc(x)
    if zero?(x) then 0 else .even?(sub1(x))

.even?(11) => 0
 .odd?(11) => 1
```

Observe that a top-level define can *shadow* a previous definition, since new bindings are added at the head of the list of top-level bindings using the `addFirst` method. A previous definition still appears in the list of bindings, but because of the way `applyEnv` works, the one that appears closer to the head of the list will always be returned when looking up the identifier.