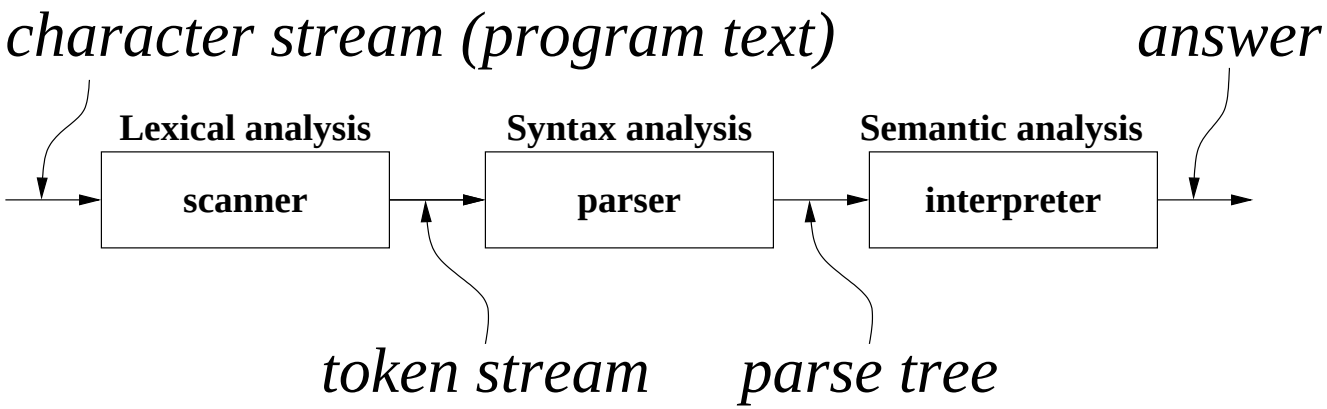
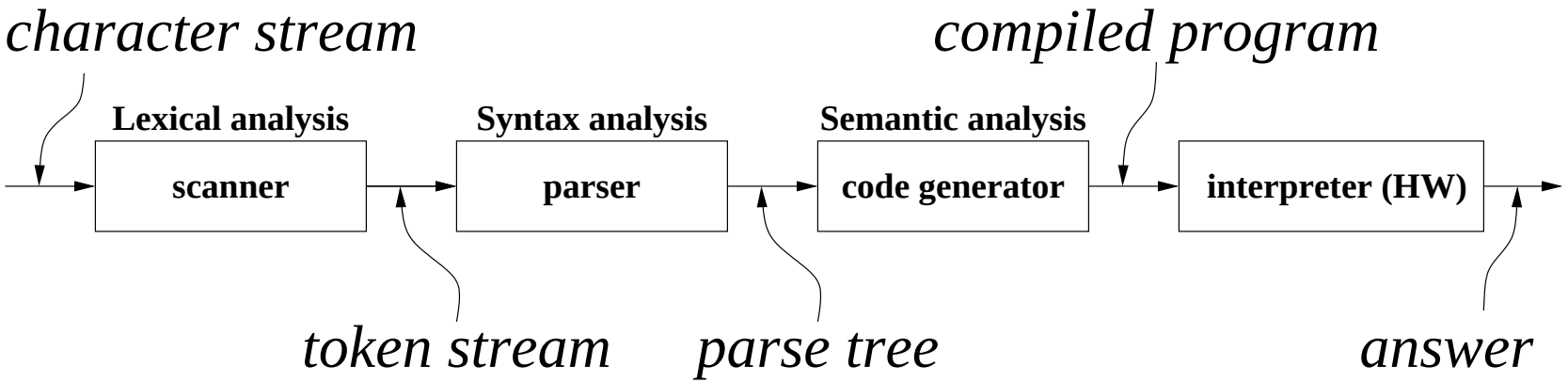
Environment-Passing Interpreters

Interpretation vs. compilation can be illustrated by a picture:

Interpreter execution:



Compiler execution:



Environment-Passing Interpreters (continued)

Most programming language grammars have a special syntactic category (a nonterminal) representing 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). Example Java expressions are '`2+3`' and '`foo(11) && toggle`'. In all of the languages we discuss in this class, every program consists of evaluating expressions. Such languages are called "expression-based languages".

An *expressed value* is the value of an expression as specified by the language semantics; for example, the expressed value of the Java 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 of expressions 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 have significant computational power, and they serve to illustrate a number of core ideas that are present in all programming languages.

We start with a language we call `V0` – think of this as 'Version zero'. Its `grammar` specification file appears on the next slide.

Language V0


```
# Language V0
skip WHITESPACE '\s+'
LIT '\d+'
ADDOP '\+'
SUBOP '\-'
ADD1OP 'add1'
SUB1OP 'sub1'
LPAREN '\('
RPAREN '\)'
COMMA ','
VAR '[A-Za-z]\w*'
%
<program>         ::= <exp>
# these three 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
%
include code
```

Language V0 (continued)

Example "programs" in this language:

```
3
x
+(3, x)
add1( +(3,x) )
+(4, -(5,2))
```

Observe that in language `V0` – and in most of the other languages you will see in this class – arithmetic expressions are written in prefix form, where the arithmetic operator (such as '+' or '−') precedes its operands. Prefix form is not entirely unusual: languages in the Lisp family (including Scheme) use prefix form. Contrast this to languages such as C, Java, and Python, where arithmetic operators appear principally in infix form.

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 `plcc` translator. Each item in a `Box` is the signature of the corresponding class constructor.

```
<program>          ::= <exp>
                   Program(Exp exp)
<exp>:LitExp       ::= <LIT>
                   LitExp(Token lit)
<exp>:VarExp       ::= <VAR>
                   VarExp(Token var)
<exp>:PrimappExp   ::= <prim> LPAREN <rands> RPAREN
                   PrimappExp(Prim prim, Rands rands)
<rands>            **= <exp> +COMMA
                   Rands(List<Exp> expList)
<prim>:AddPrim     ::= ADDOP
                   AddPrim()
<prim>:SubPrim     ::= SUBOP
                   SubPrim()
<prim>:Add1Prim    ::= ADD1OP
                   Add1Prim()
<prim>:Sub1Prim    ::= SUB1OP
                   Sub1Prim()
```

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 RHS tokens. For example, the `<exp>:PrimappExp` grammar rule has tokens `LPAREN` and `RPAREN` on its RHS, but the generated `PrimappExp` class does not have fields corresponding to these tokens: they are "abstracted away".

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. For these grammar rules, the LHS nonterminal corresponds to the name of an abstract (base) Java class whose name is obtained by capitalizing the first letter of the nonterminal name. The annotated classes `LitExp`, `VarExp`, and `PrimappExp` extend the abstract base class `Exp`. Similarly, the `AddPrim`, `SubPrim`, `Add1Prim` and `Sub1Prim` classes extend the abstract base class `Prim`.

Once you have run `plccmk` on the specification file, you can examine the Java code in the `Java` subdirectory to see, for example, that the `LitExp` class extends the `Exp` class and that the `AddPrim` class extends the `Prim` 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 `/usr/local/pub/plcc/Code/V0` contains the specification file, named `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 (semantics) section.

If only the lexical specification is given, the `plccmk` tool only produces Java code for a scanner (class `Scan`), but nothing else. If only the lexical specification and grammar rules are given, the `plccmk` tool produces 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 arithmetic value of an expression, but for now we are content with simply printing a copy of the expression itself.

The code section defines the language semantics.

Language V0 (continued)

As we have observed, the `plccmk` tool produces a scanner (`Scan`) and parser (`Parser`) for the grammar.

Assuming that the `grammar` file has been created in a directory named `V0`, the `plccmk` tool creates a `Java` subdirectory with source files named `Program.java`, `LitExp.java`, and so forth, that correspond to the abstract syntax shown in Slide 3.6. In the `Java` directory, you can also see Java source files named `Token.java`, `Scan.java`, `Parse.java`, `Parser.java`, and `Rep.java`.

The `Rep` program repeatedly prompts you for input (with '`-->`'), parses the input, and prints the result – again, a `String` representation of the parse. If you want to run this program from the directory that has the `grammar` file – `V0` in this case – you can run it as follows:

```
$ (cd Java ; java Rep)
--> add1( + (2,3))
...
```

As we discussed in Chapter 1, parsing is the process by which a sequence of tokens (a *program*) 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 program is syntactically correct.

Language V0 (continued)

We get more than a yes or no answer from our parser: *the* `plcc` *parser returns a Java object that is an instance of the class determined by the grammar start symbol.* This object is the root of the *parse tree* of the program that captures all of the elements of the parsed program.

In our `V0` grammar, the start symbol is `<program>`, so the root of the parse tree is an instance of the `Program` class.

We have seen that 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 are abstracted away when generating the objects in the parse tree.*

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

```
<exp>:PrimappExp ::= <prim> LPAREN <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;
```

Language V0 (continued)

Consider the following grammar rule in our `V0` grammar:

```
<exp>:LitExp ::= <LIT>
```

This rule creates a Java class `LitExp` having a single field named `lit` of type `Token`. The `LIT` token name is defined in the lexical specification to be a sequence of one or more decimal digits.

Continuing in this way, each of the BNF grammar rules of language `V0` (Slide 3.6) defines a class given by its LHS with a well-defined set of instance variables corresponding to the (angle bracket) entries in its RHS.

We will encounter a few situations when two RHS entries are defined by the same name in angle brackets. As we have already observerd, in these cases, we disambiguate the entries by providing different instance variable names.

Recall that repeating grammar rules have fields that are Java lists. For example, our `V0` grammar has the following repeating rule:

```
<rands> **= <exp> +COMMA
```

This rule says that the `<rands>` nonterminal can derive zero or more `<exp>` entries, separated by commas. The following sentences would match the `<rands>` nonterminal:

```
a,b,c      <-- <exp> COMMA <exp> COMMA <exp>
1, +(2,3)  <-- <exp> COMMA <exp>
add1(x)    <-- <exp>
           <-- empty string
```

The class defined by this rule is named `Rands`. Its RHS shows only one nonterminal `<exp>`, so its corresponding Java class `Rands` has a field `expList` of type `List<Exp>`.

You might wonder how we chose the name "rands". It's actually a shortened form of the term "operands". In mathematics and in programming, operands are the things being operated on. For example, given the expression `+(2,3)`, the operator is '`+`' and its operands are `2` and `3`. (Some language designs use the term "rator" as a shortened form of the term "operator".)

When the program

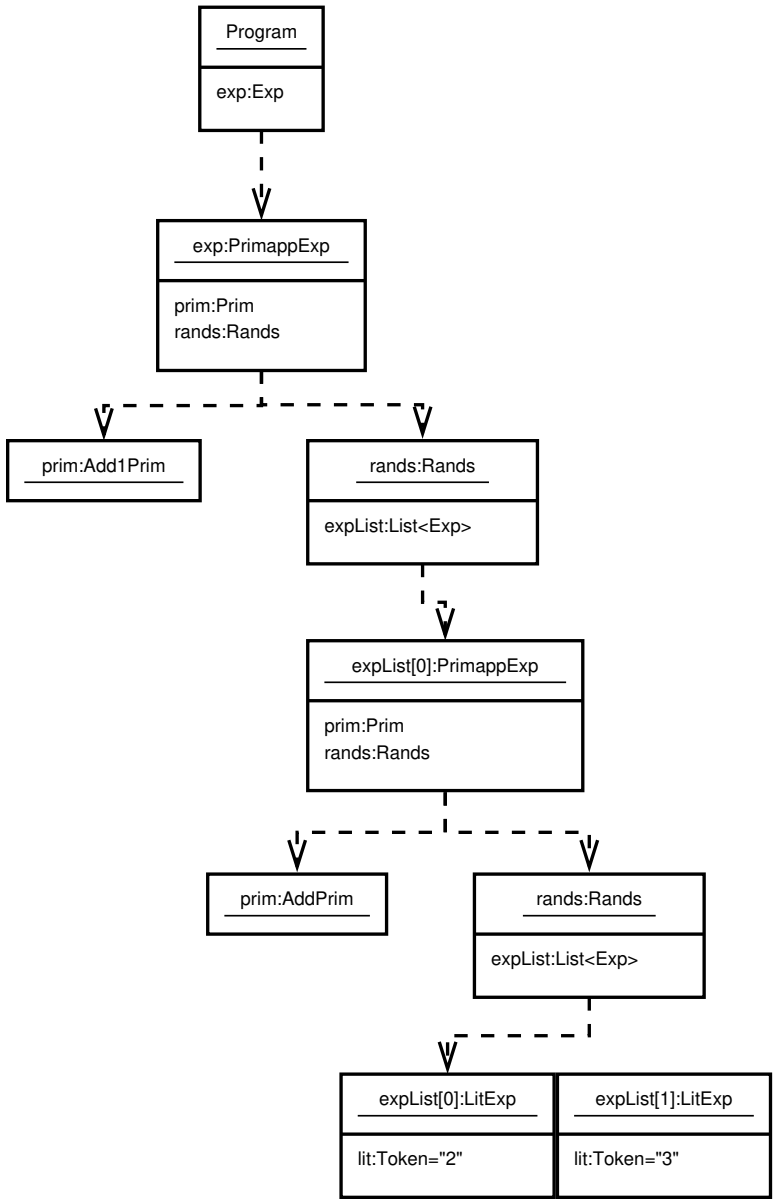
```
add1( +(2,3))
```

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))` in UML format:



The `Rep` program prints the `Program` object as a `String`. If the `Program` class does not redefine its `toString()` method, its default `toString()` method is used. As we have seen, this returns a string that looks like this:

```
Program@....
```

So our next step is to show how we can define the `toString()` method in the `Program` class so that the printed `toString()` value of a `Program` object is essentially the same string as the input with extra whitespace removed!

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

```
--> add1( +(2,3))
add1(+(2,3))
--> x
x
--> +  ( p  ,  - ( q,r)  )
+(p,-(q,r))
--> ...
```

Language V0 (continued)

We follow the method described in Slide Set 1 to modify the default behavior of the `toString()` method in the `Program` class. *In all of our languages, the observable* ***semantics*** *of an expression is the output produced by the* `toString()` *method applied to the root of the parse tree of the expression.*

Recall that to add methods to a class such as `Program`, use the following template:

```
Program
%%%
<method definitions>
%%%
```

Since these methods become part of the code for the `Program` 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:

```
Program
%%%
    public String toString() {
        return exp.toString();
    }
%%%
```

There are three `Exp` classes: `LitExp`, `VarExp`, and `PrimappExp`. Defining a `toString` method for the first two classes is particularly easy, since they both have right-hand sides that are just token strings: their `toString` methods simpy return the `String` value of the corresponding `Token` instance variables (see Slide 3.6):

```
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
                     PrimappExp(Prim prim, Rands rands)
```

A `PrimappExp` object has just two instance variables:

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

There aren't any instance variables corresponding to `LPAREN` and `RPAREN`, because the `PrimappExp` class abstracts away 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. The `toString()` methods for `prim` and `rands` are called implicitly.

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

Language V0 (continued)

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 PrimappExp class, each of the <prim> classes simply returns the corresponding string token:

```
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.

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 construct a string from each of the `expList` entries, putting commas between them. Here is the code:

```
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 exp : expList) {
        s += sep + exp; // exp.toString() is called implicitly
        sep = ",";   // commas separate the remaining 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 is parsed and re-displayed as a `String` in the same form as the input, with whitespace removed.

Language V1

Now that you see how a parse tree for language `V0` 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 `add1( + (2,3))` is to produce the integer value 6. In other words, the value of an arithmetic expression is its numeric value using usual rules for arithmetic.

(Remember that we are abstracting the notion of *value* to refer to an instance of the `Val` class. In this setting, a numeric value is an instance of the `IntVal` subclass.)

If an expression involves an identifier (symbol), we 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 `sub1(x)` would evaluate to 9.

Every expression is evaluated in an environment. This environment determines how to obtain the values of the the variables that occur in the expression

Language V1

The `Exp` class is the appropriate place to declare evaluation behavior, which we implement using a method called `eval`. Here is how we declare the abstract `eval` method in the (abstract) `Exp` class. Every class that extends `Exp` must therefore define this method:

```
Exp
%%%
    public abstract Val eval(Env env);
%%%
```

Language `V1` is the same as language `V0` except for adding `eval` methods to the classes that extend the `Exp` class. We continue to consider the only `Val` object to be an `IntVal` that holds an integer value. The file `val` in the `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`:

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

Obviously an environment doesn't have anything to do with the value of a numeric literal – a literal `10` evaluates to the integer value 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.

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

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 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 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.

```
Rands
%%%
    public List<Val> evalRands(Env env) {
        List<Val> args = new ArrayList<Val>();
        for (Exp exp : expList)
            args.add(exp.eval(env));
        return args;
    }
%%%
```

The `evalRands` method returns a *list* of `Val`s. 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****, also called* ***actual parameters****; the values of these expressions are called its* ***arguments****.*

As a careful reader of these notes, you will have observed that the class name `Rands` is derived from the word **operands**, and that the name `args` in the `evalRands` method is derived from the word **arguments**.

Language V1 (continued)

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

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

In summary, to evaluate a primitive application expression (a `PrimappExp`), evaluate the operands (a `Rands` object) in the given environment and pass the resulting arguments to the `apply` method of the primitive (a `Prim`) object, which returns the appropriate value.

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 (abstract) `Prim` class. Here is how we do this:

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

We use the parameter name '`va`' to suggest the idea of a **v**alue **a**rray.

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 knows how to add things, a `SubPrim` object knows 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:

```
AddPrim
%%%
    public Val apply(Val [] va) {
        if (va.length != 2)
            throw new RuntimeException(
                "two arguments expected"
            );
        int i0 = va[0].intVal().val;
        int i1 = va[1].intVal().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.

```
Add1Prim
%%%
    public Val apply(Val [] va) {
        if (va.length != 1)
            throw new RuntimeException(
                "one argument expected"
            );
        int i0 = va[0].intVal().val;
        return new IntVal(i0 + 1);
    }
%%%
```

Again, the definition of `apply` for the `Sub1Prim` class is entirely similar.

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 with no variables, since every variable would be unbound. To test language `V1`, we create an initial environment `initEnv` specific to this language that has the following variable bindings (think Roman numerals):

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

For language `V1`, this environment can be obtained by a call to `Env.initEnv()`. These bindings give us some variables to play with, though, as you will see, we will dispense with them later.

Language V1 (continued)

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:

```
<exp>:IfExp ::= IF <exp>testExp THEN <exp>trueExp ELSE <exp>falseExp
                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 have named these instance variables `testExp`, `trueExp`, and `falseExp`, respectively. Each of these objects refers to an instance of the `Exp` class. Thus the `IfExp` class has three instance variables:

```
Exp testExp;
Exp trueExp;
Exp falseExp;
```

[**Exercise** (not to hand in): See what would happen if you used '`<IF>`' in the RHS of this grammar rule instead of '`IF`'.]

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 is evaluated in the given environment.

Since all instances of `Val` are really `IntVals` (for the time being), we 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:

```
public boolean isTrue() {
    return val != 0; // nonzero is true, zero is false
}
```

Observe that the `eval()` method in the `IfExp` class applies the `isTrue()` method to a `Val` object, so we must include a declaration for the `isTrue()` method in the `Val` base class. **Since we treat any `Val` object as true if it's not an `IntVal` of zero, our default `isTrue()` method in the `Val` class defaults to returning `true`.**

```
Val
%%%
...
    public boolean isTrue() {
        return true;
    }
...
%%%
```

Language V2 (continued)


```
<exp>:IfExp ::= IF <exp>testExp THEN <exp>trueExp ELSE <exp>falseExp
            IfExp(Exp testExp, Exp trueExp, Exp falseExp)
```

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

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

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

Language V2 (continued)


```
<exp>:IfExp ::= IF <exp>testExp THEN <exp>trueExp ELSE <exp>falseExp
                IfExp(Exp testExp, Exp trueExp, Exp falseExp)
```

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* refers to 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. Also observe that an `if` expression in language `V2` must have both a `then` part and an `else` part

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:

```
<exp>:LetExp ::= LET <letDecls> IN <exp>
             LetExp(LetDecls letDecls, Exp exp)
<letDecls>   **= <VAR> EQUALS <exp>
             LetDecls(List<Token> varList, List<Exp> expList)
```

Notice that we 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)
```

The purpose of a `let` expression is to create an environment with new variable bindings and to evaluate an expression using these variable bindings.

Language V3 (continued)


```
<exp>:LetExp ::= LET <letDecls> IN <exp>
               LetExp(LetDecls letDecls, Exp exp)
<letDecls>   **= <VAR> EQUALS <exp>
               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 `<VAR>` symbols to the values of their corresponding `<exp>` expressions in the `<letDecls>` part, where the `<exp>` expressions to the right of the `EQUALS` are all evaluated in the enclosing environment;
2. extend the enclosing environment with these local bindings to create a new environment; and
3. use this new environment to evaluate the `<exp>` expression in the `LetExp`, and return this value as the value of the `letExp` expression.

The `<exp>` part of a `let` expression is called the *body* of the `let` expression.

Some examples if `let` expressions are on the next slide, where => means "evaluates to".

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.

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

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.

```
let z = 3 y = 8
in +(x,y)
  % => 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 `add1(x)` and `y` to the value of `add1(x)`. In both of these, the RHS expressions `add1(x)` in the inner `let` are evaluated *using the outer [enclosing] environment* which has `x` bound to 3. Thus `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.

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

Language V3 (continued)


```
<exp>:LetExp ::= LET <letDecls> IN <exp>
               LetExp(LetDecls letDecls, Exp exp)
<letDecls>   **= <VAR> EQUALS <exp>
               LetDecls(List<Token> varList, List<Exp> expList)
```

The code for `eval` in the `LetExp` class is straight-forward:

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

As we show on Slide 3.48, the `addBindings` method returns an `Env` object that extends the `env` environment parameter by adding the bindings given in the `let` declarations. This extended environment is then used to evaluate the body of the `let` expression.

Language V3 (continued)


```
<exp>:LetExp ::= LET <letDecls> IN <exp>
               LetExp(LetDecls letDecls, Exp exp)
<letDecls>   **= <VAR> EQUALS <exp>
               LetDecls(List<Token> varList, List<Exp> expList)
```

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 `Lists` is because the grammar rule has a '`**=`' instead of a '`::=`'.)

The plan for defining the `addBindings` method in the `LetDecls` class is to evaluate each of the expressions in `expList` in the enclosing environment 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 new environment is returned to the `eval` method in the `LetExp` class.

Language V3 (continued)

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.

```
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.

Language V3 (continued)

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:

```
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:

```
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`.

Language V3 (continued)


```
<exp>:LetExp  ::= LET <letDecls> IN <exp>
              LetExp(LetDecls letDecls, Exp exp)
<letDecls>    **= <VAR> EQUALS <exp>
              LetDecls(List<Token> varList, List<Exp> expList)
```

Here are a couple of other observations you should pay attention to.

- In the `<letDecls>` rule, 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 itself.) 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

  ```
  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`.

Language V4

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 add procedure definitions and procedure application. The term *procedure* is synonomous 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.

Language V4 (continued)

In language `V4`, procedures are treated as values just like integers. In particular, we 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.

```
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

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

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

```
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.)

Language V4 (continued)

Here are some examples of `V4` programs using procedures:

```
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 `x` in the `proc` definition refers to the enclosing `x` (which is bound to `10`), not to the inner `x` (which is bound to `7`). Remember the rules for evaluating the `letDecls`!

Language V4 (continued)

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

```
let
  app = proc(f,x) .f(x)
  add2 = proc(y) add1(add1(y))
in
  .app(add2,3)

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 `add2`) as a parameter to another procedure. This `app` procedure takes two parameters and returns the result of applying the first actual parameter to the second. Of course, the first parameter had better be bound to a procedure for this to work. (If it isn't, an attempt to apply it throws an exception.)

In the second example, we have eliminated the identifier `add2` and instead simply replaced `add2` in the application `.app(add2,3)` with the nameless procedure `proc(y) add1(add1(y))` that used to be called `add2`.

In the third example, we have even eliminated the identifier `app`.

Finally consider the following example, which evaluates to `120`:

```
let
  fact = proc(fact,x)
    if x
    then *(x,.fact(fact,sub1(x)))
    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!).

Language V4 (continued)

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:

```
<exp>:ProcExp ::= <proc>
              ProcExp(Proc proc)
<proc>        ::= PROC LPAREN <formals> RPAREN <exp>
              Proc(Formals formals, Exp exp)
<formals>     **= <VAR> +COMMA
              Formals(List<Token> varList)
<exp>:AppExp  ::= DOT <exp> LPAREN <rands> RPAREN
              AppExp(Exp exp, Rands rands)
```

Before we can go any further, we need to tackle the definition 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 – in other words, the variables that *occur free* in the procedure body – are bound to their values in the environment in which the procedure is defined.

In Programming Languages terminology, the term *closure* refers to an entity that captures all of the ingredients necessary to apply a procedure. In language `V4`, `ProcVal` objects are closures.

The fields of the `ProcVal` class appear here:

```
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;
    }

    ...
}
```

Recall that we can do two things with a procedure: *define* it and *apply* it. We will discuss procedure definition shortly, but first we give the semantics of procedure *application*.

Language V4 (continued)


```
<exp>:AppExp ::= DOT <exp> LPAREN <rands> RPAREN
                 AppExp(Exp exp, Rands rands)
```

Here are the steps to evaluate a procedure *application* – in other words, to evaluate an `AppExp` expression:

0. Evaluate `exp` in the current environment; this must evaluate to a `ProcVal` object (a closure) with fields `formals`, `body`, and `env`.
1. Evaluate `rands` (the *actual parameter* [a.k.a. *operand*] expressions) in the current environment to get a list of `Vals` (the *arguments*). [Note: we did exactly the same thing when evaluating the `rands` of a `PrimappExp`.]
2. a. Create bindings of the procedure's list of formal parameters (`formals`) to the list of values obtained in step 2, and

   b. use these bindings to extend the environment (`env`) captured by the procedure.
3. Evaluate the `body` of the procedure in the (extended) environment obtained in step 2.

Steps 2 and 3 are carried out by the `apply` method in the `ProcVal` class. The value obtained in step 3 is the value of the `AppExp` expression evaluation.

Language V4 (continued)

Let's now examine the detailed semantics of a `ProcExp`, which is used to *define* a procedure.

```
<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 Slide 3.58, a `ProcVal` closure is constructed with instance variables consisting of the list of formal parameters (a `Formals` object), the procedure body (an `Exp` object), and the environment in which the procedure is defined (an `Env` object).

Language V4 (continued)


```
<exp>:ProcExp ::= <proc>
                  ProcExp(Proc proc)
<proc>        ::= PROC LPAREN <formals> RPAREN <exp>
                  Proc(Formals formals, Exp exp)
<formals>     **= <VAR> +COMMA
                  Formals(List<Token> varList)
```

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

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

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

```
ProcExp
%%%
    public Val eval(Env env) {
        return proc.makeClosure(env);
    }
%%%
```

Language V4 (continued)


```
<exp>:AppExp ::= DOT <exp> LPAREN <rands> RPAREN
                 AppExp(Exp exp, Rands rands)
```

We provided the structure of the fields in the `ProcVal` class on Slide 3.58, and we described the semantics of a procedure application on Slide 3.59. We now proceed to give Java code to implement application semantics.

We start with the `eval` method in the `AppExp` class. As shown on the next slide, this method carries out steps 0 and 1 of procedure application semantics given on Slide 3.59: it evaluates the `exp` expression – which should evaluate to a `ProcVal` – and then it evaluates the operand expressions to get a list of `Vals`.

It then passes these arguments along to the `apply` method in the `ProcVal` class to carry out steps 2 and 3 of application semantics. This method returns the value of the original `AppExp` expression. (As we noted earlier, the operand expressions are called the *operands* or *actual parameters*, and their corresponding values are called the *arguments*.)

You can find the code on the following two slides.

Language V4 (continued)


```
<exp>:AppExp ::= DOT <exp> LPAREN <rands> RPAREN
                 AppExp(Exp exp, Rands rands)
```

```
AppExp
%%%
    public Val eval(Env env) {
        // evaluate exp in the current environment (step 0)
        Val v = exp.eval(env); // should be a ProcVal
        // evaluate rands in the current environment
        // to get the arguments (step 1)
        List<Val> args = rands.evalRands(env);
        // let v (step 0) determine what to do next (steps 2 and 3)
        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 `Val` class, even though you can see that the `apply` method in the `ProcVal` class does not acually use this value.

Language V4 (continued)

The only thing we have left is to implement the behavior of the `apply` method in the `Val` class. Since we want `apply` only to be meaningful for a `ProcVal` object, we define a default behavior in the (abstract) `Val` class to throw an exception for anything but a `ProcVal`:

```
public Val apply(List<Val> args, Env e) {
    throw new RuntimeException("Cannot apply " + this):
}
```

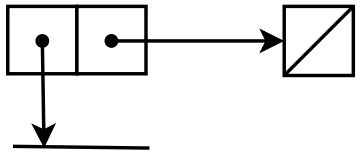
For a `ProcVal`, here's the implementation of `apply`. Notice that this implementation carries out steps 2a, 2b, and 3 in the semantics for evaluating a procedure application (Slide 3.59):

```
public Val apply(List<Val> args, Env e) {
    // bind the formals to the arguments (step 2a)
    Bindings bindings = new Bindings(formals.varList, args);
    // extend the captured environment with these bindings (step 2b)
    Env nenv = env.extendEnv(bindings);
    // and evaluate the body in this new environment (step 3)
    return body.eval(nenv);
}
```

Drawing Envinronments

On Slide 2.17, we showed how to display environments as a linked lists. Each node in the list is a pair (`EnvNode`) consisting of a reference to a `Bindings` object (which we have called *local bindings*) and a reference to the next node in the list. The end of the list is an empty environment, an `EnvNull` object, which appears as a box with a slash through it. We usually display the linked list nodes *from left to right*, with the head of the list at the left and the empty environment at the right. We display the local bindings as an array (it's actually an `ArrayList`) of bindings stacked vertically. Each binding is a pair consisting of an identifier string and a value.

The *initial environment* in language `V4` is a linked list consisting of an `EnvNode` with an empty local environment (no bindings) and a reference to an `EnvNull` object. Here is how we display the initial environment:



To simplify things in languages `V4` and `V5`, we omit displaying the node with the empty local environment, so we display the initial environment as follows:

There are exactly two ways in which programs in language `V4` create new environments using the `extendEnv` method:

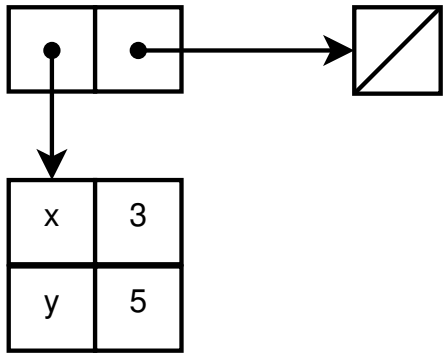
- evaluating a `let` expression
- evaluating a procedure application

For a `let` expression, a list of local bindings is created. Each binding uses the LHS string as its `id` field and the value of the RHS expression as its `val` field. Remember that the RHS expressions are evaluated *in the enclosing environment*, not in the environment being created. An example expression is given on the next page

Drawing Envinronments (continued)

```
let
  x=3
  y=5
in
  +(x,y)
```

This `let` expression creates an environment that extends the inital (empty) environment with bindings for `x` and `y`. This extended environment is the one in which the body expression `+(x,y)` is evaluated. The environment diagram showing the extended environment is shown here:

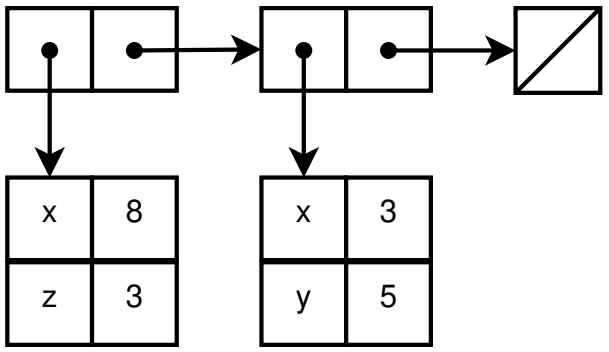


Drawing Envinronments (continued)

Now consider the following expression, with nested `lets`. The inner `let` extends the environment defined by the outer let (with one node, as shown on the previous page), so the envinroment of the inner `let` is a linked list with two nodes.

```
let
  x = 3
  y = 5
in
  let
    x = +(x,y) % the RHS evaluates to 8
    z = x      % the RHS evaluates to 3 (why?)
  in
    +(x,y)
```

In the following diagram, the leftmost node is the environment created by the inner `let`:



The inner `let` body expression evaluates to 13 (why?).

Drawing Envinronments (continued)

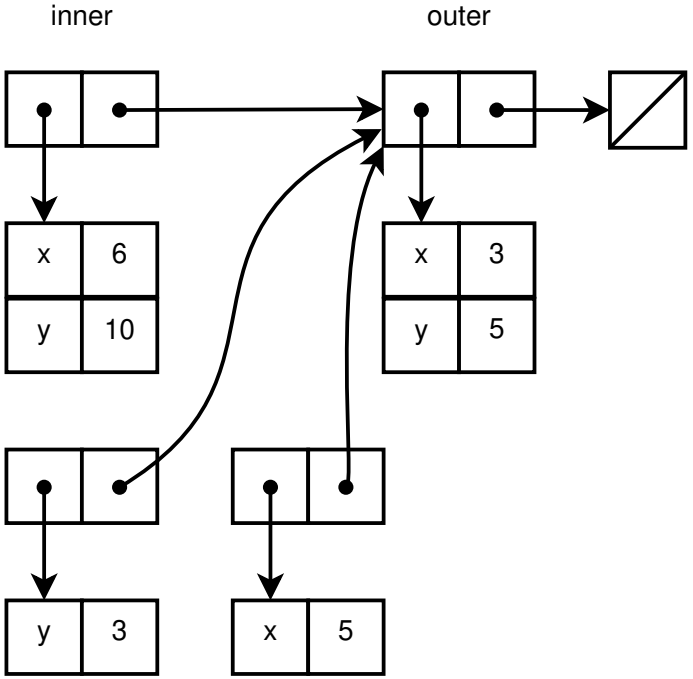
Since a `let` expression is an expression, it must evaluate to a value, so a `let` expression can occur as the RHS of a binding in another `let` expression. Consider this example:

```
let % outer
  x = 3
  y = 5
in
  let % inner
    x = let y=x in +(x,y) % LHS x is bound to 6
    y = let x=y in +(x,y) % LHS y is bound to 10
  in
    +(x,y) % evaluates to 16
```

The environment defined by the outer `let` has one node with bindings for `x` and `y` (to 3 and 5, respectively). The inner let extends the environment of the outer let, so the inner environment has two nodes. The RHS expressions for the bindings of the inner `let` are evaluated in the environment defined by the outer `let`. Since each of these RHS expressions are themselves `let` expressions, each of them extends the environment defined by the outer `let`. A total of four environments get created: the outer `let` (extending the initial null environment), the inner `let` (extending the outer `let`), and one for each of the RHS expressions in the inner `let` (extending the outer `let`). The next slide shows all of these environments.

Drawing Envinronments (continued)


```
let % outer
  x = 3
  y = 5
in
  let % inner
    x = let y=x in +(x,y) % LHS x is bound to 6
    y = let x=y in +(x,y) % LHS y is bound to 10
  in
    +(x,y) % evaluates to 16
```



Drawing Envinronments (continued)

In the definition of the `ProcVal` class, a `ProcVal` object has three fields:

```
public Formals formals; // list of formal parameters
public Exp body;        // procedure body
public Env env;         // captured environment
```

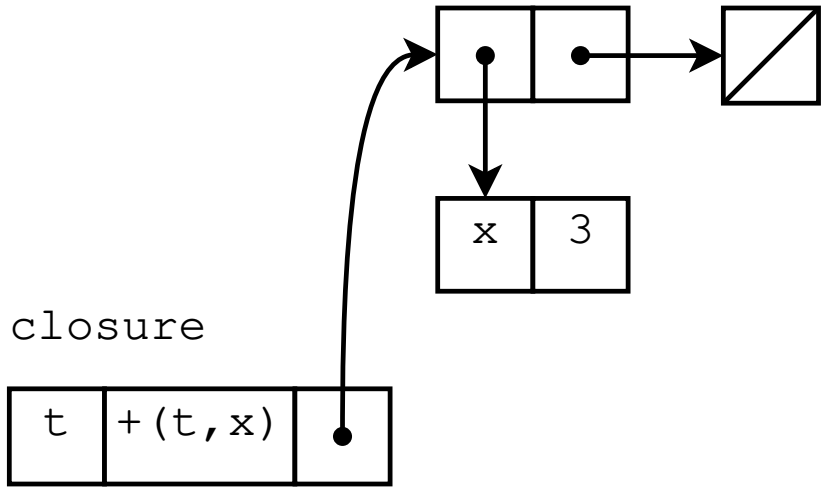
Here, the *captured environnment* is the environment in which the procedure is defined. For example, consider the following expression:

```
let
  x = 3
in
  proc(t) +(t,x)
```

This expression evaluates to a `ProcVal`: its `formals` field consists of a list having a single string, '`t`', its `body` is the expression '`+(t,x)`', and its captured environment is the one defined by the `let`, having a single binding of `x` to the value `3`. In many programming languages, a `ProcVal` object is called a *closure*. We normally display a `ProcVal` object as a rectangle with its three compartments, in this order: formals, body, and captured environment. The formals are shown as a comma-separated list of identifiers, the body is shown as an expression, and the captured environment is shown as an arrow pointing to the appropriate node in the environment linked list. The following slide shows the `ProcVal` that results from the evaluation of the above expression.

Drawing Envinronments (continued)


```
let
  x = 3
in
  proc(t) +(t,x)
```



Drawing Envinronments (continued)

The rules for *applying* a procedure are given on Slide 3.59. With respect to environments, the key ideas are: (2a) the procedure's formal parameters are bound to the values of the actual parameter expressions (a `Bindings` object), and then (2b) the captured environment is extended with these bindings to create a new environment; finally, (3) the value of the procedure application is the value of the procedure body using the new environment.

Consider the following expression, which is the same as the previous expression except that the procedure is applied to the actual parameter 5:

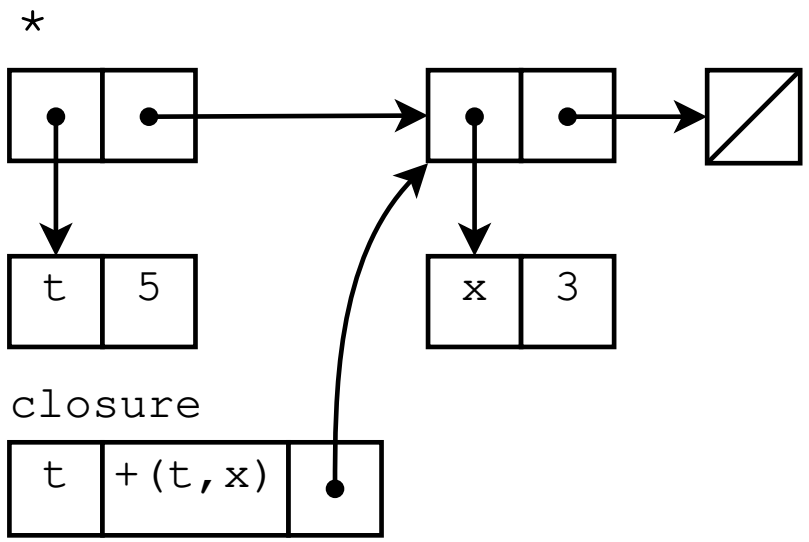
```
let
  x = 3
in
  .proc(t) +(t,x) (5)
```

From the above discussion, this procedure application creates a local binding of the formal parameter `t` to the value 5 (the actual parameter expression's value). This binding is used to extend the environment captured by the procedure, and this extended environment is used to evaluate the body of the procedure. The value of this application is 8.

The following page displays the environment created by this application, marked with an asterisk '`*`'.

Drawing Envinronments (continued)


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



Drawing Envinronments (continued)

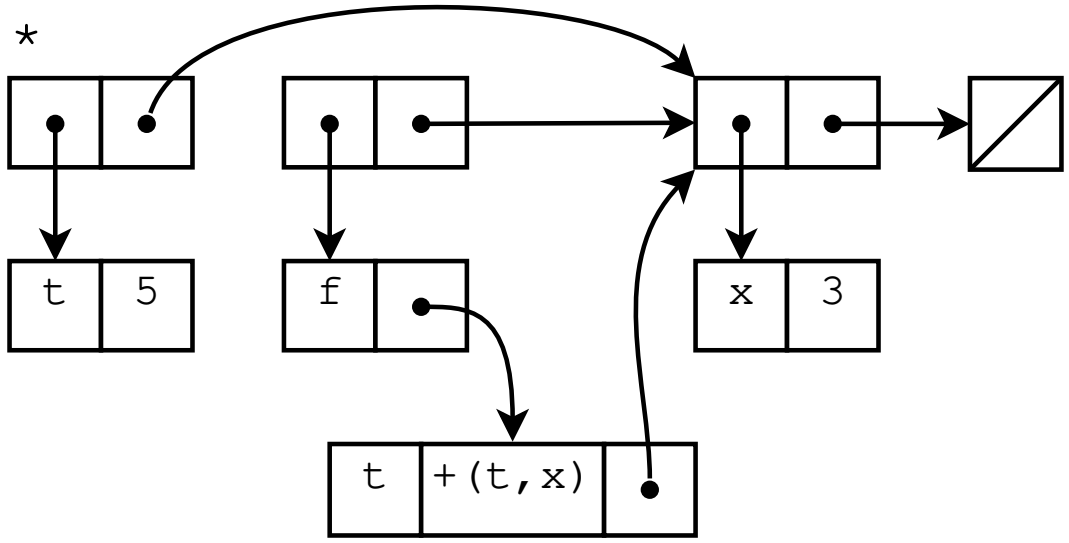
Finally, consider this example:

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

The value of this expression is also 8. Evaluating this expression creates three environments: one with a binding of `x` to 3, another with a binding of `f` to the closure, and a third created by applying `f` to the argument 5. The resulting environment diagram is shown on the following page. The environment in which the body of `f` is marked with an asterisk '`*`'.

Drawing Envinronments (continued)


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



Language V4 (continued)

Once we have procedures, we can entirely eliminate the `let` construct! Here's an example:

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

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

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

In general, a `let` construct in the form

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

can be re-written in the form

```
.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. The `let` construct aligns the LHS variables `v1`, `v2`, *etc.* physically close to their corresponding RHS expressions `e1`, `e2`, *etc.*, so it's cognitively easy for the reader to see how these LHS variables become bound to the values of their RHS expressions. In the equivalent procedure application, the formal parameters `v1`, `v2`, *etc.* are physically distant from their corresponding RHS expressions, making it difficult to visualize these bindings.

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 read, understand, and use.

Though language V4 does not support direct recursion, its support of procedures as first-class entities – that is, they are values that are treated the same way as other values, so they can be passed as parameters and returned as values – is as powerful as recursion. Here is another example that recursively computes factorials using an "accumulator" and tail recursion (we will return to this topic later):

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

Observe that the identifier `fact` that appears in the `proc(fact,...)` definition is a *formal parameter* name that binds all occurrences of `fact` that appear in the procedure body. You may find it instructive to display all of the environments that get created during the evaluation of this expression. (Replace '5' by '2' to make things easier.)

Language V4 (continued)

Finally, language `V4` includes the ability to evaluate a sequence of expressions, returning the value of the last expression. The component expressions in a sequence expression are always evaluated left-to-right. Sequence expressions do not have any particular usefulness now because our language is not *side-effecting*, but they will turn out to be useful in later languages that do support side-effects.

```
<exp>:SeqExp ::= LBRACE <exp> <seqExps> RBRACE
             SeqExp(Exp exp, SeqExps seqExps)
<seqExps>    **= SEMI <exp>
             SeqExps(List<Exp> expList)
```

The semantics of evaluating a `SeqExp` are given here:

```
SeqExp
%%%
    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:

```
.{proc(t,u) +(t,u)} (3,4)
```

The braces in this expression are not required, but they help to visualize the scope of the `proc` expression.

Language V5

We normally prefer to use direct recursion instead of using the contrived (but workable) tricks on slides 3.55 and 3.79. For example, we would like to write

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

But this does 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` fails because of an unbound identifier.

In order to solve this problem, we create a new `let`-like environment that supports direct recursion. Called `letrec`, it allows us to define procedures that support direct recursion. Unlike `let`, `letrec` only supports bindings of identifiers to procedures.

This is what we want:

```
letrec
  fact = proc(x) if zero?(x) then 1 else *(x,.fact(sub1(x)))
in
  .fact(5)
  % => 120
```

Language V5 (continued)

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?

The steps are shown on the next slide:

Here are the steps:

1. Extend the enclosing environment with `null` bindings. This simply serves 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 are 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)


```
<exp>:LetrecExp  ::= LETREC <letrecDecls> IN <exp>
                 LetrecExp(LetrecDecls letrecDecls, Exp exp)
<letrecDecls>    **= <VAR> EQUALS <proc>
                 LetrecDecls(List<Token> varList, List<Proc> procList)
```

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

```
LetrecDecls
%%%
    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;
    }
%%%
```

Language V5 (continued)


```
<exp>:LetrecExp  ::= LETREC <letrecDecls> IN <exp>
                 LetrecExp(LetrecDecls letrecDecls, Exp exp)
<letrecDecls>    **= <VAR> EQUALS <proc>
                 LetrecDecls(List<Token> varList, List<Proc> procList)
```

We can now evaluate a `LetrecExp` object in exactly the same way as a `LetExp` object:

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

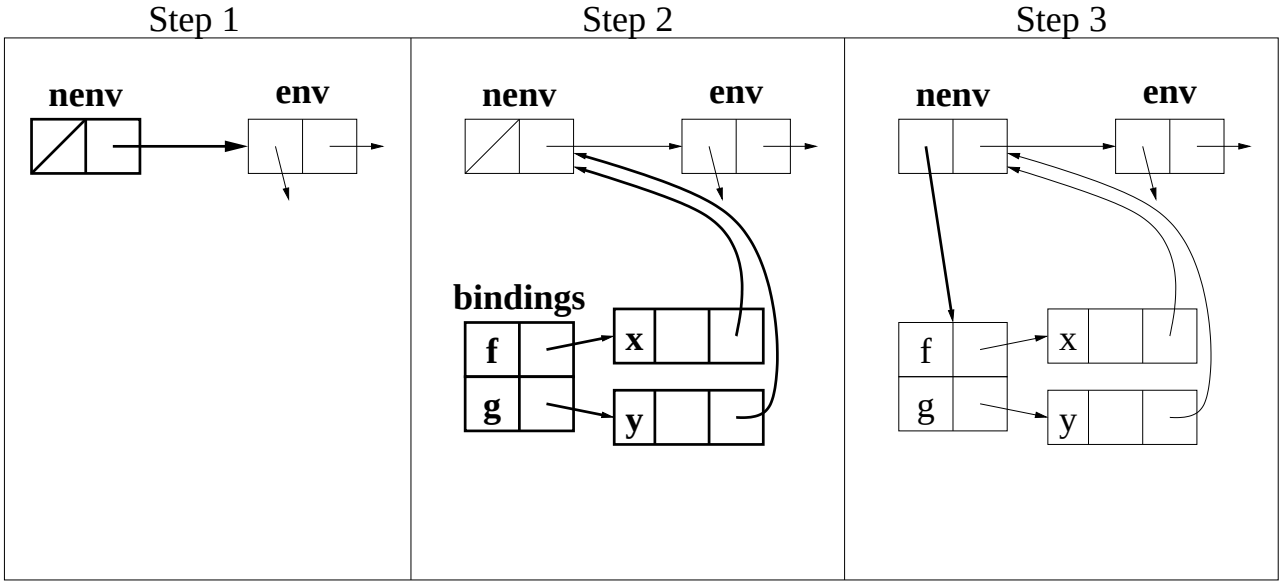
The principal idea, then, is to create the RHS closures of a `letrec` in an environment that (self-referentially) includes all of the bindings in the `letrec`.

Language V5 (continued)

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.



The `letrec` construct allows us to define *mutually recursive procedures* – two or more procedures that call each other in a recursive fashion. Here's a classic example:

```
letrec
  even? = proc(x) if zero?(x) then 1 else .odd?(sub1(x))
  odd? = proc(x) if zero?(x) then 0 else .even?(sub1(x))
in
  .even?(11) % => 0 (false)
```

[**Exercise** (not to hand in): See if you can do this in language V4 without `letrec`.]

Notice that we have used `?` in the variable names for the `even?` and `odd?` procedures. This is an additional feature we have added to languages `V5` and beyond.

Language V6

So far, our defined language has no capability to define top-level variables that persist from one expression evaluation to another. A *top-level* variable is one that has a binding in the initial ("top-level") environment. 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.

Language V6 (continued)

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:

```
<program>:Define ::= DEFINE <VAR> EQUALS <exp>
                     Define(Token var, Exp exp)
<program>:Eval   ::= <exp>
                     Eval(Exp exp)
```

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

```
define i = 1
define ii = add1(i)
define iii = add1(ii)
define v = 5
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) sub1(x)
.f(v)   % => 120 -- g is now bound
.f(iii) % => 6
```

As long as you stay in the `Rep` loop, the `defined` variable bindings are 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)` fails. After defining `g` on the following 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` now recognizes its definition. This *only* works for top-level defines and cannot be used in `let` 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 can access this binding (these closures all capture the top-level environment), as well as any others that may crop up later! Thus the following works:

```
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.