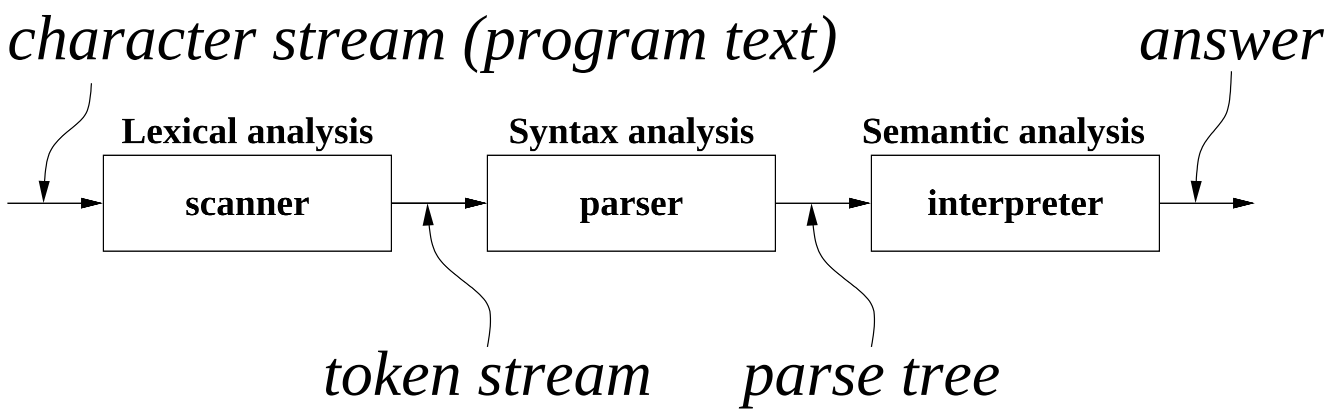
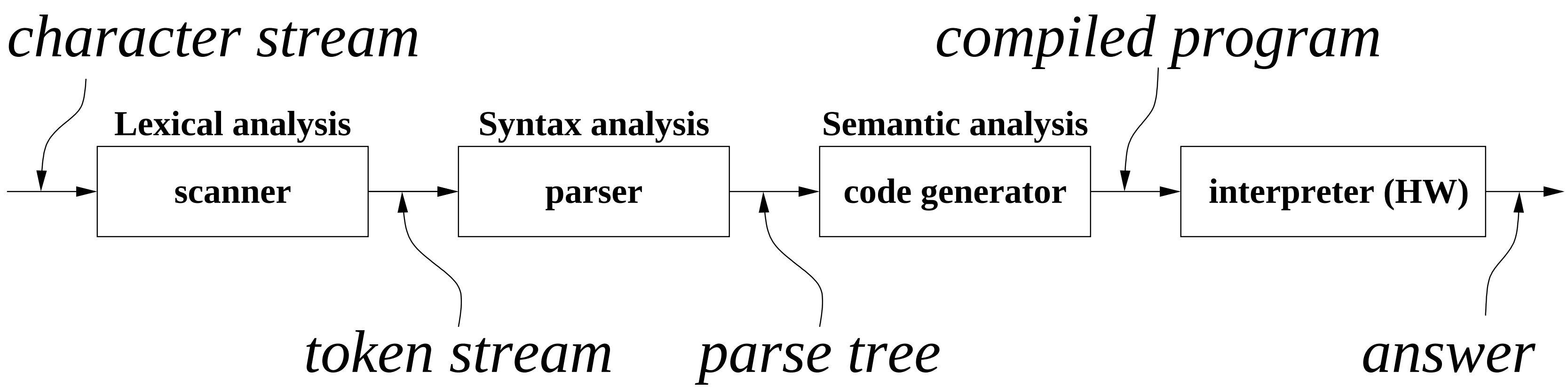
Environment-Passing Interpreters

Interpretation vs. compilation can be illustrated by a picture:

Interpreter execution:



Compiler execution:



Environment-Passing Interpreters (continued)

Most programming languages have grammar rules defining 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 (using its `toString` representation, for example), and its denoted value is the value bound to the symbol in its environment. 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 a *source language* and its *implementation language*. A source language is a language to be interpreted, and its implementation language is the language in which the interpreter is written. (The term *defined language* is often used to refer to a source language. Similarly, the term *defining language* is often used to refer to its implementation language.)

In the rest of this course, our source languages will be a collection of artificial languages used to illustrate the various stages of language design, and our implementation 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 "Language V0" – think of this as "Language 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)

Observe that this PLCC language specification has all three of the parts described in Slide Set 1a: the lexical specification section (token specifications), the syntax specification section (given as BNF rules), and the semantic specification section (just an `#include` line).

You can consider a grammar rule like

```
<program>          ::= <exp>
```

to mean that “a `program` consists of an `exp`” (where `exp` stands for an “expression”). Similarly, you can consider a grammar rule like

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

to mean that “a `LitExp` is (an instance of) an expression (an `exp`) that consists of a `LIT`.” Similar remarks apply to all the grammar rules.

The ‘`#include code`’ line at the end of this file means that the contents of the file named `code` should be inserted into the `grammar` file input stream at that point to be processed by PLCC.

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 these class notes – we write arithmetic expressions in *prefix form*, where the arithmetic operator (such as '+' or '−') precedes its operands. (A prefix form expression like '`+(3,x)`' would normally be written mathematically as '`3+x`', using *infix form*. It turns out that prefix form expressions are easier to parse and evaluate than infix form expressions, which is why we use prefix form in our languages. See the `INFIX` language in Slide Set 6 for a further discussion of infix form.)

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 Language V0 to its Java representation as an abstract syntax. The Java class files are created automatically by PLCC. 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 explicit 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 certain 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, we must disambiguate these grammar rules 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 create the `grammar` specification file and run `plccmk`, you can examine the Java code in the `Java` subdirectory. Here you can see, for example, that the `LitExp` class extends the `Exp` class and that the `AddPrim` class extends the `Prim` class.

Language V0 (continued)

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.

The `grammar` file in Language V0 has three parts, separated by lines with a single '`%`': the **lexical specification section**, the **syntax specification section**, and the **code (semantics) section**.

Recall that if your `grammar` file has only the lexical specification section, the `plccmk` tool produces Java code for a scanner (`Scan`), but nothing else. If your `grammar` file has only the lexical specification and syntax specification sections, the `plccmk` tool produces Java code for a scanner (`Scan`) and a parser (`Parse`) for the grammar, but nothing else.

Language V0 (continued)

The code section defines the language semantics. In this section, the Java classes defined by the grammar rules are given life by defining their behavior – specifically, by defining the `$run()` method in the start symbol class (`Program`, in the case of language `V0`).

In the absence of a redefined `$run()` method – for example, if the code section is omitted – the default `$run()` behavior in the `_Start` class is to print the name of the start symbol in a format illustrated in Slide 1.29.

In later versions of this language (`V1` and beyond), we see how such a `$run()` method can be used to print the arithmetic value of an expression. But in Language `V0` we will be content with simply printing a copy of the expression itself.

In the code section of a PLCC language specification, the behavior of the `$run()` *method in the start symbol class defines the language semantics.*

Language V0 (continued)

Assuming that we have created the `grammar` file in a directory named `V0`, running the `plccmk` tool creates a `Java` subdirectory with source files named `Program.java`, `LitExp.java`, and so forth, that correspond to the abstract syntax classes shown in Slide 3.7. In the `Java` directory, you can also see Java source files named `Token.java`, `Scan.java`, `Parse.java`, and `Rep.java`.

The `Rep` program repeatedly prompts you for input (with '`-->`'), parses the input, and prints the result: a `String` representation of the expression with extra white-space removed. 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:

```
$ java -cp Java Rep
--> add1( + (2
  , 3))
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 success or failure response from our parser: *the* `plcc` *parser returns a Java object that is an instance of the class determined by the BNF 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 Language V0 grammar, the start symbol is `<program>`, so the root of the parse tree is an instance of the `Program` class.

Language V0 (continued)

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 Language V0:

```
<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 Language V0:

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

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

Continuing in this way, each of the BNF grammar rules of Language V0 (Slide 3.7) defines a class given by its LHS with a well-defined set of instance variables corresponding to the <*angle bracket*> entries in its RHS.

As we have already observed, when we encounter situations where two RHS entries have the same (Token or nonterminal) name in angle brackets, we disambiguate the entries by providing different instance variable names.

Language V0 (continued)

Recall that repeating grammar rules have fields that are Java lists. For example, our Language 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 a name like “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. (Similarly, some language designs use the term “rator” as a shortened form of the term “operator”.)

Language V0 (continued)

When we parse a program such as

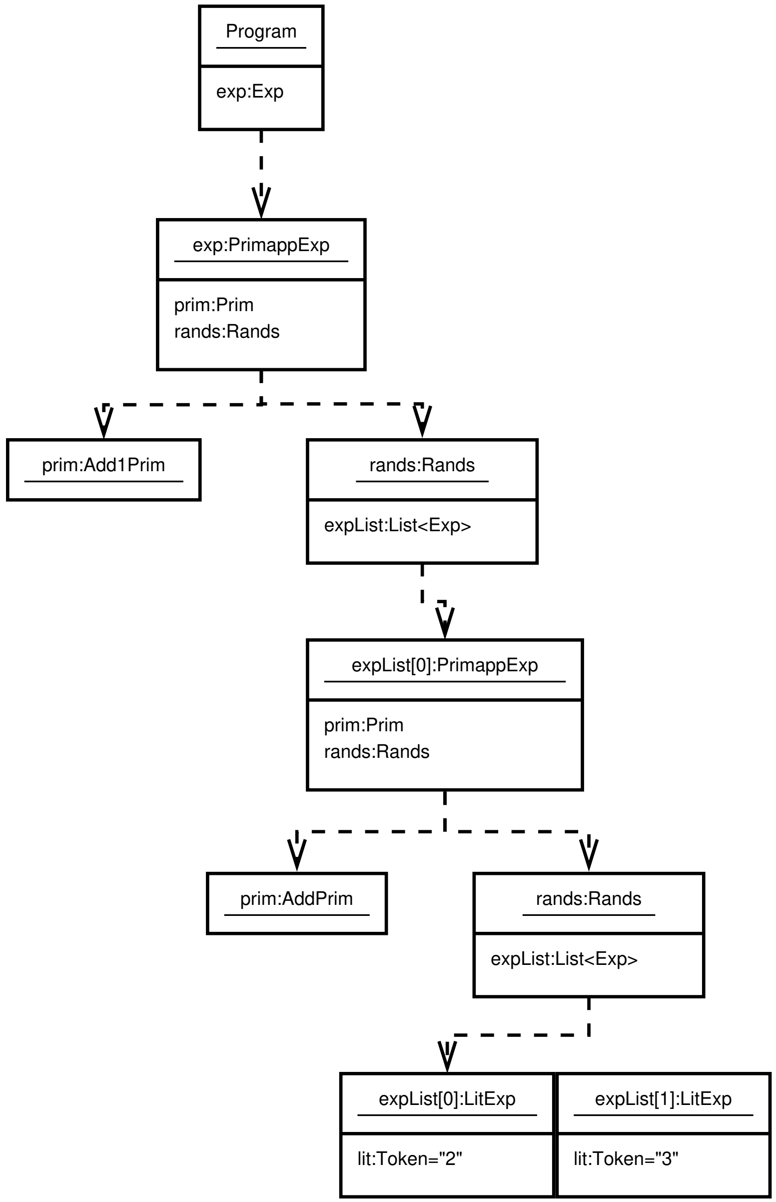
```
add1( +(2,3))
```

the parser returns an object of type `Program`. 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 defaults to running the `$run()` method in the `Program` class, which prints the `Program` object as a `String`. (Actually, it prints the `_Start` object as a `String`, but since the `Program` class extends the `_Start` class, these behaviors are the same.) The default `$run` behavior – which would be used if `$run` is not redefined in the `Program` class – displays a string that looks like this:

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

We want to show how to override the default `$run()` method in the `Program` class so that it prints the same text as the program input with extra whitespace removed!

This means that we should see the following when interacting with the `Rep` program from Language 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 redefine the default behavior of the `$run()` method in the `Program` class. *In all of our languages, the observable* ***semantics*** *of a program is the output produced by the* `$run()` *method applied to the root of the parse tree of the program.*

Recall that to add Java code (fields and method definitions) to a class such as `Program`, use the following template:

```
Program
%%%
...Java code...
%%%
```

Since PLCC inserts this code verbatim into the body of the `Program` class, any methods defined in this code can access all of the instance variables in the class. So for the `Program` object, these methods can refer to the `exp` instance variable of type `Exp`. Since the RHS of the `<program>` grammar rule is just `<exp>`, the `$run()` method of the `Program` class is simple:

```
Program
%%%
    public void $run() {
        System.out.println(exp.toString());
    }
%%%
```

Language V0 (continued)

To finish our implementation, we show how to implement the `toString()` method for an `Exp` object so that it returns a `String` representation of itself.

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 3.7):

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

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

Language V0 (continued)

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.

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 an `expList` instance variable which is a `List` of expressions. First examine the `<rands>` grammar rule:

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

To build a `toString` method for this class, we call the `toString()` method on each of the `expList` entries and construct a `String` that puts 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: `Rep` parses each syntactically correct input expression and displays the resulting `Program` object as a `String`, which appears the same 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))` means 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 of `Val`.)

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.

The interpreter evaluates every expression in some environment. This environment determines how to obtain the values bound to the identifiers 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.

Language V1 has three new files included in its `grammar` language definition file: `envRN`, `val`, and `prim`.

- The `envRN` file contains Java class definitions to implement environments, as discussed in Slide Set 2.
- The `val` file defines the `Val` class and its `IntVal` subclass – a straight-forward wrapper for Java `int`s.
- The `prim` file defines the semantics of the seven `<prim>` BNF rules specified in Language V1.

Language V1 (continued)

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 can coexist with the `toString` behavior that we defined in Language V0, but we do not show this here.)

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

Remember that a `LitExp` has a `Token` field named `lit`. When we apply the `toString()` method to this field, we get the string of decimal digits that came from the part of the program text we are parsing. The `IntVal` constructor converts this into a real Java `int` that becomes part of the `IntVal` instance. 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.

Language V1 (continued)

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`.

Language V1 (continued)

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>(expList.size());
        for (Exp exp : expList)
            args.add(exp.eval(env));
        return args;
    }
%%%
```

Language V1 (continued)

We *specify* that the expressions in an `evalRands` method call be evaluated in first-to-last (or left-to-right, depending on how you are looking at it) order. This is not necessarily the case for all programming languages. In particular, see Slide 3a.38 *et seq*. for a more complete discussion of order of evaluation.

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**.

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`), we evaluate the operands (a `Rands` object) in the given environment and pass the resulting argument array to the `apply` method of the primitive (a `Prim`) object, which returns the appropriate value.

We are left to define the behavior of the `apply` methods in the various `Prim` classes. Observe that by the time a `Prim` object gets its array of arguments, the environment no longer plays a role, since we have already evaluated all of the operand expressions: only values remain.

Language V1 (continued)

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 seven instances of this class) has no instance variables. However, we can endow these objects with behavior, so that an `AddPrim` object knows how to add things, a `SubPrim` object knows how to subtract things, and so forth.

Language V1 (continued)

Four of the `Prim` objects need two arguments ('+', '−', '∗', and '/'), and three of them need one argument (`add1`, `sub1`, and `zerop`). 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 PLCCException("two arguments expected");
        int i0 = va[0].intVal().val;
        int i1 = va[1].intVal().val;
        return new IntVal(i0 + i1);
    }
%%%
```

The `intVal()` method calls shown in this code convert `Val` objects (such as `Va[0]`) into `IntVal` objects – essentially like "downcasting". These objects, in turn, have Java `int` fields named `val`. So both `i0` and `i1` are legitimate Java `int`s that can be added together to return the resulting `IntVal` object. The `Val` class defines the `intVal()` method behavior: an attempt to apply the `intVal()` method to a `Val` object that is *not* an `IntVal` throws an exception.

Language V1 (continued)

The definitions of `apply` for the classes `SubPrim`, `MulPrim`, and `DivPrim` (the latter two are added in V1) have obvious implementations, except that in `DivPrim`, the `apply` method throws an exception if it detects an attempt to divide by zero: this code is not shown here.

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 PLCCException("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 definition of `apply` for the `ZeropPrim` class returns an `IntVal` of 1 (true) for a zero argument and an `IntVal` of 0 (false) for a nonzero argument.

Language V1 (continued)

In our final implementation step, we will define the `$run()` method of a `Program` object that displays the string representation of the *value* of its expression. See the next slide for how we implement this.

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()`. In the `Program` class, we set the static `initEnv` variable to this value. 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 `$run()` method for the `Program` object, along with the definition of the initial environment `initEnv` in the `Program` class:

```
Program
%%%
    public static Env initEnv = Env.initEnv();

    public void $run() {
        System.out.println(exp.eval(initEnv).toString());
    }
%%%
```

To test this, run the `Rep` program defined in the `Java` subdirectory and enter expressions at the prompts:

```
java -cp Java Rep
```

Language V2

Language V2 is the same as Language V1, with the addition of the syntax and semantics of an if expression. The relevant grammar rule and abstract syntax representation are shown here:

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

Notice that we need to add token names IF, THEN, and ELSE to our lexical specification, along with their obvious regular expressions.

The RHS of the IfExp grammar rule has three occurrences of the <exp> nonterminal. Since the <exp> items on the RHS of this 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. 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’.]

Language V2

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.

Language V2 (continued)

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 currently 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:

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

The `isTrue()` boolean method applies to any instance of `Val`. It is a Java helper method used only to implement the semantics of the `if...then...else` expression; it is *not* part of the source language. On the other hand, `zero?` is a *primitive* in the source language (starting with Language V1), not a method in Java. The `zero?` primitive applies only to integer values in the source language. We define the *semantics* of the `zero?` primitive using the `apply` Java method in the `ZeropPrim` class. You may find this somewhat confusing.

Language V2 (continued)

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

Observe that the `eval` method in the `IfExp` class evaluates *only one* of the `trueExp` or `falseExp` expressions, never both. This is a semantic feature – not a syntax feature – of the definition of `eval` for an `if` expression. The term *special form* refers to semantic structures that behave 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 /(5,0) else 8)
  % => 11 (note that the /(5,0) expression is not evaluated!
```

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 our source languages must have both a `then` part and an `else` part, even though only one of these expressions ends up being evaluated.

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:

0. create a set of local bindings (a Bindings object) 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;

1. extend the enclosing environment with these local bindings to create a new environment; and

2. 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.

Examples of let expressions are on the next slides, where => means "evaluates to". Remember: *a* let *expression is an expression, and as such, it evaluates to something*!

Language V3 (continued)

In an instance of the `LetDecls` class, we require that no `varList` variable occurs twice. Slide 3.48 shows how we can achieve this.

Now that we can define our own environments, we will remove our initial environment with bindings for Roman numerals.

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.

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

In the second `let` expression example, a new environment is created binding `x` to 10. The body of this `let` expression is itself a `let` expression with bindings of `z` to 3 and `y` to 8. The environment of the inner `let` extends the environment of the outer `let`, so that the `x` in the expression `+(x,y)` is bound to 10. The entire expression therefore evaluates to 18.

```
let x = 10
in
  let z = 3 y = 8
  in +(x,y) % => 18
```

Language V3 (continued)

In the third example, two new environments are created. The outer `let` binds `x` to 3. The inner `let` binds `x` to the value of `add1(x)` and `y` to the value of `add1(x)`. Both of these `add1(x)` RHS expressions 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
```

Observe also that the `add1` primitive is *not* side-effecting. This means that the expression `add1(x)` does *not* modify the value bound to `x`: `add1(x)` in our languages behaves like `x+1` and *not* like `++x` as you would find in languages such as C++ and Java.

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 3.49, the `addBindings` method returns an `Env` object that extends the `env` environment parameter by adding the bindings given in the `let` declarations. We use this extended environment 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)
```

A `LetDecls` object has two instance variables: `varList` is a list of `Token` objects representing the `<VAR>` part of the BNF grammar rule, and `expList` is a list of expressions representing the `<exp>` part of the BNF grammar rule. (The reason that these are `Lists` is because the `letDecls` BNF grammar rule is repeating.) Our plan for defining the `addBindings` method in the `LetDecls` class involves evaluating each of the expressions in `expList` in the enclosing environment and binding these values to their corresponding token strings in `varList`. We then use these bindings to extend the enclosing environment given by the `env` parameter, and we return this new environment to the `eval` method in the `LetExp` class.

The `LetDecls` constructor throws an exception if it finds duplicate identifiers in its `varList`. This means that a `let` expression cannot have two instances of the same LHS idenfier. The code to check for duplicates is inserted into the `LetDecls` class constructor using the (*context-sensitive*) `:init` hook, so the check for duplicates occurs during parsing, not during expression evaluation.

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);
    }
%%%
```

Language V3 (continued)

The languages we have discussed do not allow mutation of variables, although you might be tempted to think that this Language 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` 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` remains 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 is another observation 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 become 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.

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 Language V4 program that includes a procedure definition and application.

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

In Language V4, 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 Language 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`.

Language V4 (continued)

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

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

One final observation: `add1` is a *primitive*, not a *procedure*. Primitives are not expressions, and they do not evaluate to anything, so you can't pass a primitive as an actual parameter to a procedure. In particular, the following will not work:

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

Be aware that the syntax for *applying* a primitive looks somewhat like applying a procedure, except that applying a primitive does *not* use a DOT.

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.

Language V4 (continued)

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 1, 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 3.59, 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 3.59, and we described the semantics of a procedure application on Slide 3.60. 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 3.60: it evaluates the `exp` expression – which should evaluate to a `ProcVal` – and then it evaluates the operand expressions to get a list of `Val`s.

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 `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 PLCCException("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 3.60).

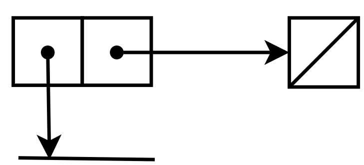
```
public Val apply(List<Val> args, Env e) {
    // bind the formals to the arguments (step 2a)
    Bindings bindings = new Bindings(formals.varList, args);
    // extend the captured environment with these bindings (step 2b)
    Env nenv = env.extendEnv(bindings);
    // and evaluate the body in this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

Language V4 also checks for duplicate identifiers while parsing the `Formals` in a procedure definition. This code is inserted into the `Formals` class constructor using the `:init` hook.

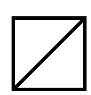
Drawing Envinronments

On Slide 2.20, 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:



Drawing Envinronments (continued)

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

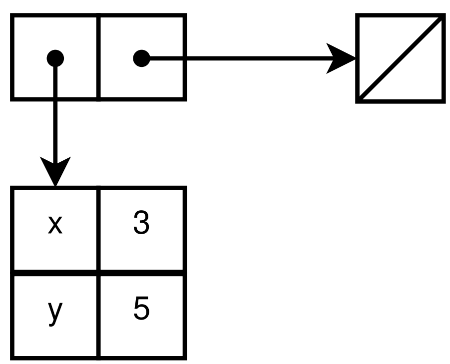
- evaluating a `let` expression
- evaluating a procedure application

A `let` expression, creates a list of local bindings: 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:

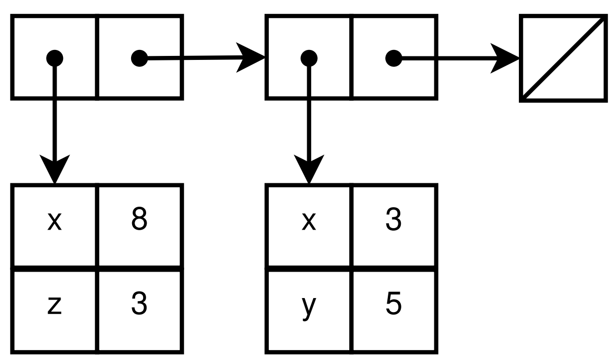


Drawing Envinronments (continued)

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)

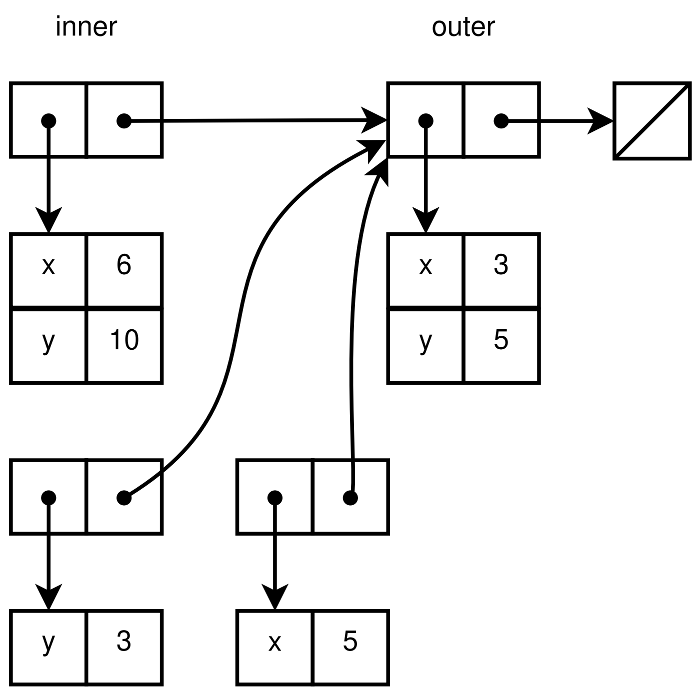
Since a `let` expression is an expression, it must evaluate to a value, so a `let` expression can occur as the RHS of a binding in another `let` expression. Consider this example:

```
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
```

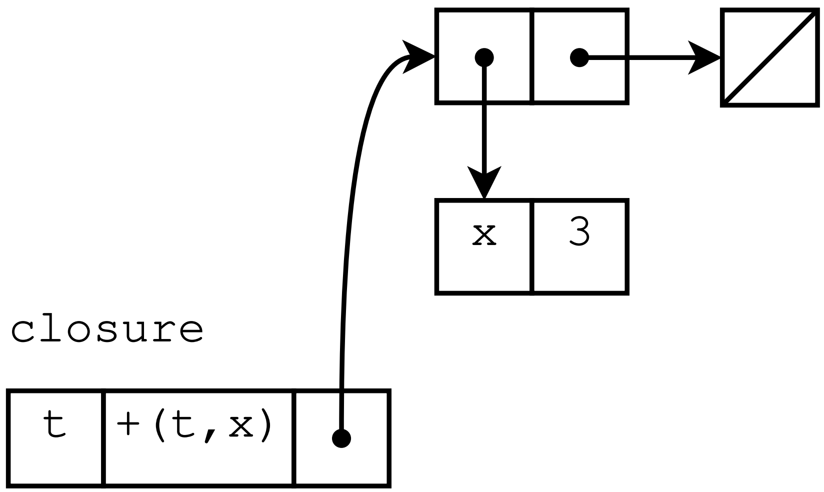
Here, the *captured environnment* is the environment in which the procedure is defined. For example, consider the following expression:

```
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`. (Recall that we also use the term *closure* to refer to a a `ProcVal` object.) We normally display a `ProcVal` object as a rectangle with its three compartments, in this order: formals, body, and captured environment. We show the `formals` as a comma-separated list of identifiers, the `body` as an (un-evaluated) expression, and the captured environment (`env`) as an arrow pointing to the appropriate node in the environment in which the procedure definition occurs. 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)
```



We described the rules for *applying* a procedure on Slide 3.60. In the `apply` method in the `ProcVal` class, the key steps are: (2a) bind the procedure's formal parameters to the values of the actual parameter expressions (a `Bindings` object); (2b) extend the captured environment with these bindings to create a new environment; (3) evaluate the procedure body using the new environment. The value obtained in step (3) is the value of the procedure application.

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

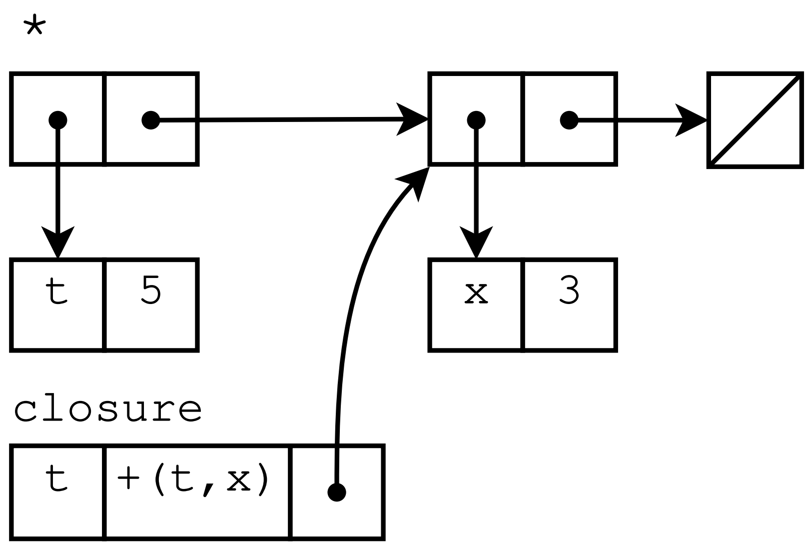
```
let
  x = 3
in
  .proc(t) +(t,x) (5)
```

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 '$\star$'.

Drawing Envinronments (continued)


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



Drawing Envinronments (continued)

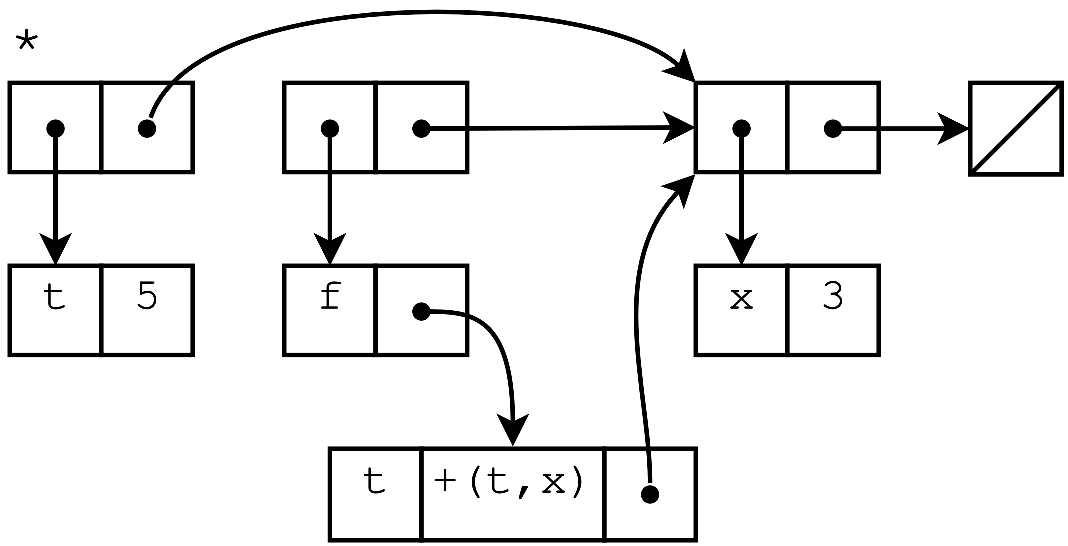
Finally, consider this example:

```
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 we evaluate the body of `f` is marked with an asterisk '$\star$'.

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)
```

Language V4 (continued)

In general, a `let` expression

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

can be re-written as an equivalent procedure application expression

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

This conversion is *algorithmic*: it can be carried out unambiguously. 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. A `let` expression is an example of *syntactic sugar*: a syntactic and semantic construct that has another, equivalent way of expressing it in the language but that programmers find easier to read, understand, and use.

Language V4 (continued)

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

Observe that the identifier `f` that appears in the `proc(f, x, acc)` definition is a *formal parameter* name that binds all occurrences of `f` 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;
    }
%%%
```

Language V4 (continued)

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

```
{1;3;5}
  % => 5

{42}
  % => 42
```

We can use the sequence construct 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 may help to visualize the extent of the `proc` definition. Don't get into the habit of doing this on a regular basis, however: throwing in extra braces can result in an expression that is unnecessarily *noisy* and that is actually more difficult to read than one without.

Language V5

We normally prefer to use direct recursion instead of using the contrived (but workable) tricks on Slides 3.56 and 3.80. 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` which is free in the procedure definition – there is no `fact` in its list of formal parameters – 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.

Language V5 (continued)

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.

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 is the grammar rule and associated abstract syntax class:

```
<exp>:LetrecExp ::= LETREC <letDecls> IN <exp>
                   LetrecExp(LetDecls letDecls, Exp exp)
```

The RHS expressions in a `letrec` are evaluated in the order in which they appear, using an envionment where *all* of the previous (LHS, RHS) bindings in the `letrec` are acesssible. In addition, if the RHS is is a procedure, it captures the entire environment created by *all* of the (LHS, RHS) bindings in the `letrec`. This means that procedures defined in a `letrec` can refer to each other. In particular, they can call themselves recursively.

This is unlike a normal `let`, in which the RHS expressions are all evaluated in the *enclosing* environment, and the (LHS, RHS) bindings created by the `let` are only accessible in the body of the `let`. Moreover, in a `let`, the RHS expressions can be evaluated in any order, which is sometimes called *parallel evaluation*.

Notice that the syntax of a `letrec` is the same as the syntax of a `let`. The only difference between the semantics of `let` and `letrec` is in the way in which we build the environment in which the `letDecls` bindings are created.

We proceed to describe how `letrec` evaluation is handled.

To implement the recursive behavior of a `letrec` as described above, we create a new method `addLetrecBindings` in the `LetDecls` class. This new method is passed the environment in which the `letrec` expression appears, and it returns a new environment as described in the following steps.

0. Extend the environment actual parameter with an empty `Bindings` object of sufficient size to hold all of the variable bindings. Assign this new environment to the `env` parameter.

1. The two fields in the `LetDecls` class are `List<Token> varList` and `List<Exp> expList`. Create iterators for these two lists and iterate through them together, in order. For each step in the iteration, get the next identifier `var` from the `varList`, and save its `String` representation in a variable `str`. Also, get the next expression `exp` from the `expList`, evaluate it in the environment `env` obtained in Step 0, and save its value in a variable `val`. Then create a `new Binding(str, val)` and add it to the `env` environment obtained in Step 0. This binding now becomes part of the local bindings in `env`, together with the other local bindings previously added during this iteration.

2. Once all of the new bindings have been added to `env`, return `env` as the value of this method.

Language V5 (continued)

The implementation of `addLetrecBindings` in the `LetDecls` class is shown here:

```
LetDecls
%%%
    public Env addLetrecBindings(Env env) {
        // Step 0
        env = env.extendEnv(new Bindings(varList.size()));
        // Step 1
        Iterator<Token> varIter = varList.iterator();
        Iterator<Exp> expIter = expList.iterator();
        while (varIter.hasNext()) {
            String str = varIter.next().toString();
            Val val = expIter.next().eval(env);
            env.add(new Binding(str, val));
        }
        return env; // Step 2
    }
%%%
```

Notice that we have previously defined an `addBindings` method in the `LetDecls` class (see Language V3) used to implement the `eval` semantics of a `let` expression. The `addLetrecBindings` method simply becomes another part of the `LetDecls` class.

Language V5 (continued)

```
<exp>:LetrecExp ::= LETREC <letDecls> IN <exp>
                 LetrecExp(LetDecls letDecls, Exp exp)
```

Recall that the `LetDecls` constructor checks for duplicate LHS identifiers during parsing. Since the `LetrecExp` grammar rule uses `LetDecls`, a `letrec` expression also makes this check.

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

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

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

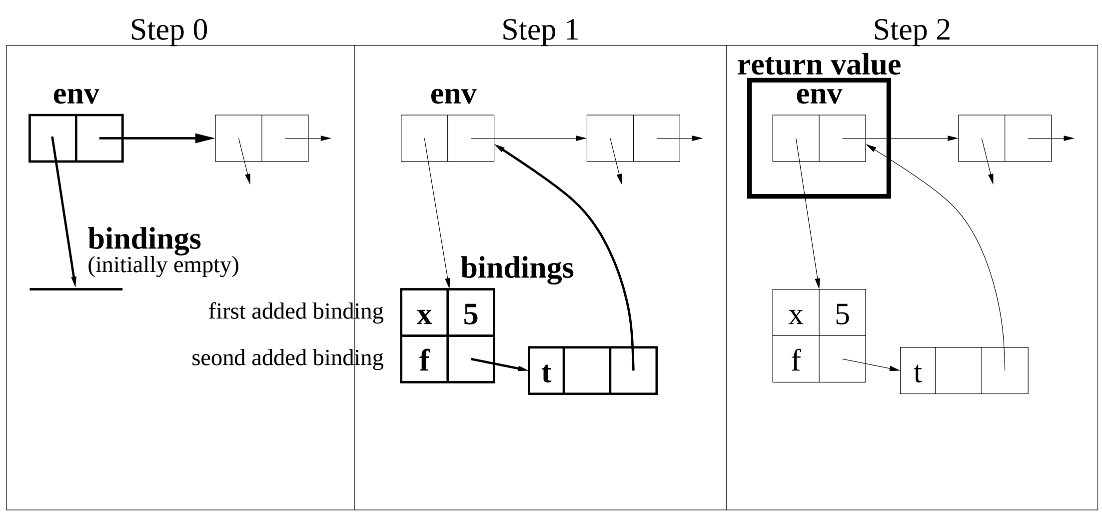
Language V5 (continued)

This picture illustrates the three steps carried out in `addLetrecBindings`, for the following `letrec` example:

```
letrec
    x = 5
    f = proc(t) *(t,x)
in
    .f(42)
```

0. Create an extended environment by extending the enclosing environment with an empty list of bindings;
1. In the order in which the LHS identifiers appear, create a `Binding` of the LHS identifier (`x` and then `f` in this example) to the value of its corresponding RHS expression (5 and then `proc(t) ...` in this example) – where each RHS expression is evaluated in the extended environment – and add this binding to the extended environment;
2. Once all of the (LHS, RHS) bindings have been added to the extended environment, return the extended environment as the value of `addLetrecBindings`.

Observe that the environment captured by procedure `f` knows about the binding of `x` to `5`, so the `letrec` expression evaluates to 210. If this had been a `let` instead of a `letrec`, the `x` in the body of `f` would be unbound.



Language V5 (continued)

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 define the `odd?()/even?()` mutually recursive procedures in Language V4 without `letrec`.]

Notice that we have used `?` in the variable names for the `even?` and `odd?` procedures to suggest that these procedures should be considered as *predicates* that return true (1) or false (0). This is a lexical feature we have added to Languages V5 and beyond.

Language V6

So far, our source 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 our languages to allow for such definitions. All we need to do is to add bindings to the *initial environment*, the environment that all expressions in the source language extend.

The initial environment for our languages is the `static Env env` variable in the `Program` class, obtained by calling the `initEnv()` static method in the `Env` class. Notice that this initial (top-level) environment starts out having an empty list of bindings.

Our strategy for making top-level definitions is to take advantage of the `add` method in the `Env` class. To add a new top-level definition, we create a `Binding` object and add it to the top-level `Env` object. Once we add bindings in this way, these bindings will be known in any subsequent expression evaluation that uses the initial environment.

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)
```

Language V6 (continued)

Here is an example of expressions that use the `define` feature in our source 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 `define`d 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.

Language V6 (continued)

Notice that for top-level procedure definitions, `define` works similar to `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 we add a new binding to the top-level environment, the binding gets added to the *local bindings* instead of extending the top-level environment. In this way, all of the top-level closures can access this binding, 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 *redefine* a previous definition. We do this by looking up the LHS identifier in the top-level environment. If a binding to this identifier already exists in the top-level environment, we replace the binding's value with the value of the new RHS.

Language V6 (continued)

Since a `define` evaluates its RHS in the *current* environment, we can capture (and save) the value of a variable using a `let`, even though a subsequent definition may redefine the variable. Consider this example:

```
define x = 2
define f = proc() x % x is the top-level x
.f()                % evaluates to 2
define x = 3        % redefine top-level x
.f()                % now evaluates to 3
```

Compare this to the following:

```
define x = 2
define f =
  let
    x = x           % the RHS is the current value (2),
                    % and the LHS is a local copy
  in
    proc() x        % the proc captures the local copy
.f()                % evaluates to 2
define x = 3        % redefine top-level x
.f()                % local copy still evaluates to 2
```