Programming Language Concepts
CSCI344

Course Objectives

- Lexical analysis
- Syntax
- Semantics
- Functional programming
- Variable lifetime and scoping
- Parameter passing
- Object-oriented programming
- Logic programming
- Continuations
- Exception handling and threading

Lexical Analysis, Syntax analysis, and Semantic analysis

The *syntax* (from a Greek word meaning "arrangement") of a programming language refers to the rules used to determine the structure of a program written in the language. *Syntax analysis* is the process of applying these rules to determine the structure of a program. A program is *syntactically correct* if it follows the syntax rules defining the language. Every programming language has syntax rules (these rules differ from one programming language to another) that are part of the programming language specification.

Before we can specify the syntax rules of a programming language, we must specify the *lexical* (from a Greek word meaning "word") structure of the language: the symbols used to construct a program in the language. These symbols are called *tokens*. *Lexical analysis* is the process of applying these rules by reading program input and isolating its tokens. Tokens comprise the "atomic structure" of a program.

Lexical analysis is also called *scanning*. You can think of scanning as what you do when you "scan" a line of printed text on a page for the words (tokens) in the text. Programming language tokens normally consist of things such as numbers ("23" or "54.7"), identifiers ("foo" or "x"), reserved words ("for", "while"), and punctuation symbols (".", "["). Every programming language has rules that define the tokens in the language (these rules differ from one programming language to another) that are part of the programming language specification.

Lexical Analysis, Syntax Analysis, and Semantic Analysis

A *lexical analyzer* is a program or procedure that carries out lexical analysis for a particular language. Such a program is also called a *scanner*, *tokenizer*, or *lexer*. The input to a scanner is a stream (sequence) of characters, and its output is a stream of tokens. The behavior of a scanner for a language is defined by the lexical specification of the language.

A *syntax analyzer* is a program or procedure that carries out syntax analysis for a particular language. Such a program is also called a *parser*. The input to a parser is a stream of tokens (produced by a seanner), and its output is a *parse tree* that is an abstract representation of the structure of the program. The behavior of a parser for a language is defined by the syntax specification of the language.

The string of input characters that makes up a token is called a *lexeme*. For example, when you read a word (token) from printed text on a page, the particular collection of characters that make up the word is its lexeme. In this paragraph, the first lexeme is "the" (ignoring case), consisting of the individual letters 't', 'h', and 'e'. This lexeme is an instance of an English part of speech called an "article". In this case, "article" is the token and "the" is the instance. The other instances of the "article" token (in English) are "a" and "an". **A token is an abstraction, and a lexeme is an instance of this abstraction**.

Lexical Analysis, Syntax Analysis, and Semantic Analysis (continued)

The *semantics* (from a Greek word meaning "meaning") of a programming language refers to the rules used to determine the meaning of a program written in the language. Here "meaning" refers to (a) whether the program makes sense, and (b) what the program does when it is run. In languages such as English, a sentence can make sense (such as "the dog ate the bone") but it may not have any meaning in terms of what it "does". Programs are expected to "do" something, and what they "do" is part of their semantics. *Semantic analysis* is the process of applying these rules to a program written in the language to determine its meaning.

A *semantic analyzer* is a program or procedure that carries out semantic analysis. The input to a semantic analyzer is a parse tree. The output is either a direct execution of the resulting program (as defined by what the program does when it is run) or some intermediate form (such as machine code) that can be run at some other time. Direct execution is called *interpretation*, whereas producing an intermediate form is called *compilation*. We will use the interpretation approach in these notes, though the techniques we describe apply as well to a compilation approach.

Lexical Analysis, Syntax Analysis, and Semantic Analysis (continued)

When a program produces some output, for example, the language semantics defines what specific behavior results from running the program. For example, the defined semantics of Java dictates that the following Java program sends, to the standard output stream, the decimal character 3 followed by a newline:

```
public class Div {
    public static void main(String [] args) {
        System.out.println(18/5);
    }
}
```

This course is about programming language syntax and semantics, with an emphasis on semantics. Syntax doesn't matter if you don't understand semantics.

Quick question: what output is produced by the following snippet of python3 code?

```
print(18/5)
```

Does this say anything about how the semantics of Java and Python differ when it comes to integer division?

Syntax and semantics

You can use a language *compiler* to tell you if a program you write is syntactically correct, but it's much more difficult to determine if your program always produces the behavior you *want* – that is, if a program is semantically "correct." There are two basic problems:

1. how to specify formally the behavior you want, and
2. how to translate that specification into a program that actually behaves according to the specification.

Of course, **a program is its own specification**: it behaves exactly the way its instructions say it should behave! But nobody knows exactly how to create a *behavioral specification* that precisely and unambiguously describes what you *want* – in part because what you want is often imprecise and ambiguous – and then translate this behavioral specification into a program whose semantics *provably behaves* according to the specification. Because of this, programming will always be problematic. (Creating behavioral specifications is a topic of interest in its own right and properly belongs in the discipline of software engineering.)

In this course, we are interested in defining precisely and unambiguously how a program *behaves*: its semantics. After all, if *you* don't know how a program behaves, it's hopeless to put that program into a production environment where other users expect it to behave in a certain way.

Syntax and semantics (continued)

This course is about programming languages, and particularly about *specifying* programming languages. A programming language *specification* is a document that defines:

1. the lexical structure of the language (its tokens);
2. the syntax of the language; and
3. the behavior of a program when it is run.

In particular, we give examples of language specifications that describe the lexical and syntax structure of a number of languages and how to implement their run-time behaviors (semantics). We show how variables are bound to values, how to define functions, and how parameters are passed when functions are called.

Because a program in a particular language must be syntactically correct before its semantic behavior can be determined, part of this course is about syntax. But in the final anaysis, semantics is paramount.

Tokens

Assume that we have a program written in some programming language. (Think of languages such as C, Java, Python, and so forth.) The first step in analyzing the structure of a program is to examine its lexical structure: the "atoms".

A program is, at the lowest level, a stream of characters. But some characters are typically ignored (for example, "whitespace", including spaces, tabs, and newlines), while some characters group together to form things that can be interpreted (for example) as integers, floats, and identifiers. Some specific character sequences are meaningful in the language, such as '`class`' and '`for`' in Java. Some individual characters are meaningful, such as parentheses, brackets, and the equals symbol, while some pairs or characters are meaningful such as '++' and '<='. We use the term *token* to refer to such atoms.

A *token* in a programming language is an abstraction that considers a string of one or more characters in the character stream as having a particular meaning in the language – a meaning that is more than the individual characters that make up the string. The term *lexical analysis* refers to the process of taking a stream of characters representing a program and converting it into a stream of *tokens* that are meaningful to the language.

Lexical analysis takes character stream input and produces token stream output. For example, if a language knows only about integers and the dot symbol '`.`', an input stream consisting of characters

```
23.587
```

might produce three tokens as output, with the following lexemes:

```
23
.
587
```

while a language that knows about floats and doubles might produce just a single token, with the following lexeme:

```
23.587
```

In what follows, we will often use the term "token" (an abstraction) when it might be more appropriate to use the term "lexeme" (an instance of the abstraction). The context should make our intent clear.

Tokens (continued)

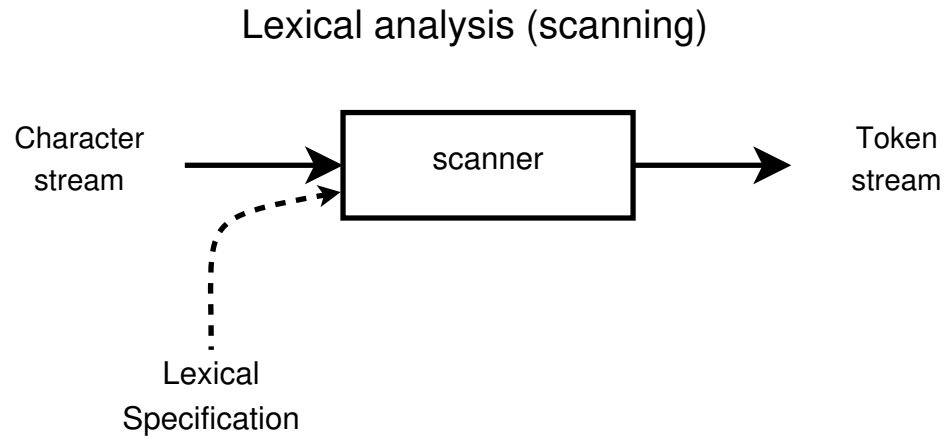
The purpose of lexical analysis is to take program input as a stream of characters and to produce output consisting of a stream of tokens that conform to the *lexical specification* of the language.

By a *stream*, we mean an object that allows us to examine the current item in the stream, to advance to the next item in the stream, and to determine if there are no more items in the stream. We accomplish something similar to this in Java with `hasNext()` and `next()` for Scanner objects in Java's `java.util.Scanner` library.

By a *stream of characters* we mean a stream of items consisting of individual characters in a character set such as ASCII or UTF-8. By *stream of tokens* we mean a stream of items that are tokens defined by a token specification. Our lexical analysis process provides a `hasNext()/next()` interface for accessing the stream of tokens, but it also provides a similar `cur()/adv()` interface described in more detail below.

As we noted earlier, lexical analysis is referred to as *scanning*, and a *scanner* is a program that carries out this process.

Tokens (continued)



In this course, a programming language specification is given in a text file normally named `grammar`. Each language has its own `grammar` file. A `grammar` file has three sections: the lexical specification, the syntax specification, and the semantic specification. These sections appear in the `grammar` file in this order: first lexical, then syntax, and finally semantic. A line consisting of a single '`%`' separate the sections.

We start by describing the structure of the lexical specification section.

Tokens (continued)

The lexical specification section of a `grammar` file uses *regular expressions* to specify language tokens. A regular expression is a formal description of a pattern that can match a sequence of characters in a character stream. For example, the regular expression '`d`' matches the letter `d`, the regular expression '`\d`' matches any decimal digit, and the regular expression '`\d+`' matches one or more decimal digits. **You should read the Java documentation for the `Pattern` class for information about how to write regular expressions.**

When specifying tokens, we must identify what input stream characters do *not* belong to tokens and should be skipped. Typically, we skip whitespace: spaces, tabs, and newlines. We identify these skipped characters in the `grammar` file using a *skip specification* line like this :

```
skip WHITESPACE '\s+'
```

The regular expression '`\s`' stands for "space" (the space character, a tab, or a newline), and the regular expression '`\s+`' stands for one or more spaces. We use the symbolic name "WHITESPACE" to identify this particular collection of characters to be skipped. (Any symbolic name would suffice, but it makes sense to use one that describes its purpose.) We also typically skip comments.

Characters to be skipped during lexical analysis are not tokens.

Tokens (continued)

To specify tokens, we use token specification lines having the format

```
token  <TOKEN NAME>  '<re>'
```

where <TOKEN NAME> identifies the token (it must be in ALL CAPS) and <re> is a regular expression that defines the structure of the token's lexeme.

We adopt one simplifying rule for *all* the languages we discuss in this class: *tokens cannot cross line boundaries*. Be warned, however, that not all programming languages conform to this rule.

For example, we may use the following lines in our `grammar` file to specify a *number*, the *reserved word* `proc`, and an *identifier*:

```
token NUM '\d+'
token PROC 'proc'
token ID '[A-Za-z]\w*'
```

We like to use symbolic names for our tokens that make it easy to remember what they represent.

You can find the documentation for regular expressions such as these in the Java `Pattern` class.

Whenever we are faced with two token specifications whose regular expressions match a string of consecutive input characters, we *always choose the one with longest possible match*. For example, if the next characters in the input stream are

```
procedure
```

the above specifications would produce an `ID` token with lexeme `procedure` instead of a `PROC` token with lexeme `proc`: both of these specifications match the beginning ('`proc`') of input, but the `ID` match is longer.

If two or more token specifications match the same input (longest match), we always *choose the first specification line in the* `grammar` *file that matches* to identify the token.

In summary, for a given input string, we always

1. choose the token specifications with the longest match, and
2. among those with the longest match, choose the first token specification that appears in the `grammar` file.

We use the phrase *first longest match* to describe these rules for token processing.

Writing a scanner is somewhat involved, so we have provided you with a tool set that produces a Java scanner automatically from a file that specifies the tokens using regular expressions. This tool set, named `PLCC`, consists of a program written in Python 3 along with some support files. `PLCC` stands for a "Programming Languages Compiler Compiler". You should be able to use this tool set with any system that supports Python 3 and Java. The `plcc.py` Python program and the `Std` subdirectory that contains its support files are on the RIT Ubuntu lab systems in this directory:

```
/usr/local/pub/plcc/src
```

This directory also contains a shell script called `plcc` that runs the `plcc.py` Python program, along with a script called `plccmk` that also compiles the Java programs created by PLCC. When you are working on one of our Ubuntu lab systems, you can simply run `plcc` or `plccmk` to process the various languages we will specify in this course.

See the `HOWTO.html` file in `/usr/local/pub/plcc/tvf` for *important information* about how to set up your CS account environment so that PLCC will be able to access the required program files and library routines on CS servers and lab workstations. This file also explains how you can use `github` to set up PLCC on a local Windows or Linux computer system. If you do this, you are on your own.

When running `plcc`, you need to give the name of the language specification file on the command line. For example:

```
plcc grammar
```

The `grammar` file is a text file that defines the tokens of the language using skip specifications and token specifications as we have illustrated earlier. The filename `grammar` is typically used, but you can name your file anything you wish, so that both of the following examples are acceptable:

```
plcc mygrammar
plcc foo
```

The language specification file can contain comments starting with a '#' character and continuing to the end of the line. These comments are ignored by `plcc`.

The `plccmk` script runs `plcc` on a language specification file whose name must be `grammar`. In addition to creating a `Java` subdirectory and depositing the `Scan.java` program in that directory (along with a few other necessary Java support files), `plccmk` compiles the Java programs in that directory. Our first `grammar` files will contain only token specifications. Later, we will use `grammar` files to define language syntax, and then semantics. For now, we concentrate only on token specifications.

Tokens (continued)

Here are some example specification files that you can try. Each of these examples can be put in a file named `grammar` for processing by `plccmk`. These examples define what input should be skipped and what input should be treated as tokens. Comments in a language specification file begin with the # character and go to the end of a line.

- ```
  # Every character in the file is a token, including whitespace
  token CHAR '.'
  ```
- ```
  # Every line in the file is a token
  token LINE '.*'
  ```
- ```
  # Tokens in the file are 'words' consisting of one or more
  # letters, digits or underscores -- skip everything else
  skip NONWORD '\W+'  # skip non-word characters
  token WORD '\w+'    # keep one or more word characters
  ```
- ```
  # Tokens in the file consist of one or more non-whitespace
  # characters, skipping all whitespace.
  # Gives the same output as Java's 'next()' Scanner method.
  skip WHITESPACE '\s+'  # skip whitespace characters
  token NEXT '\S+'       # keep one or more non-space characters
  ```

To test these, create a separate directory for each test (we use the convention that such directories have names that are written `IN_ALL_CAPS`), and create a `grammar` file in this directory with the given contents. Then, in this directory, run the following commands:

```
plccmk
java -cp Java Scan
```

The '`-cp Java`' command-line arguments sets the Java `CLASSPATH` environment to the `Java` directory; this, in turn, tells the Java interpreter where to find the `Scan` program.

The `Scan` program expects input from standard input (your keyboard) and produces output lines that list the tokens as they are scanned, in the form

```
lno: NAME 'string'
```

where `lno` is replaced by the input line number where the token is found, `NAME` is replaced by the token's symbolic name, and `string` is replaced by the token's corresponding lexeme from the input that matched the `NAME` token specification.

After running the `plccmk` command, a `Java` subdirectory is created, populated with the following Java source files

```
Token.java
Scan.java
```

as well as a few other Java support files. The `plccmk` command also compiles these source files. When you run

```
java -cp Java Scan
```

you can enter strings from your terminal and see what tokens are recognized by the scanner.

Examine the `Token.java` file to see how the token specifications in your `grammar` file are translated into Java code that associates the token names with their corresponding regular expression patterns, and similarly for skip patterns.

When specifying tokens in a `grammar` file, you can omit the `token` term (but not the `skip` term). This means that both

```
WORD '\S+'
```

and

```
token WORD '\S+'
```

are considered as equivalent. We follow this convention in all of our subsequent examples.

The `plccmk` script calls the `plcc.py` translator on the `grammar` specification file. The `plcc.py` translator takes the `Token.pattern` file in the `Std` directory and modifies it using the grammar specification, creating a Java source file `Token.java` in your `Java` subdirectory. It also copies the `Scan.java` file in the `Std` directory into your `Java` subdirectory. The `plccmk` script then compiles these two Java programs. If you have made any mistakes in your `grammar` file, these mistakes may show up during translation (with the `plcc.py` program) or during compilation of the Java source files.

The important pieces of the `Scan` class are the constructor and two methods: `cur()` and `adv()`. The `Scan` constructor must be passed a `BufferedReader`, which is the input stream of characters to be read by the scanning process. A `BufferedReader` can be constructed from a `File` object, from `System.in`, or from a filename given in a `String`. The `Scan` program reads characters from this `BufferedReader` object line-by-line, extracts tokens from these lines (skipping characters if necessary), and delivers the current token with the `cur()` method – `cur` stands for *cur*rent.

The `adv()` method advances the scanning process so that the token returned by the next call to `cur()` is the next token in the input. Notice that multiple calls to `cur()` without any intervening calls to `adv()` all return the same token.

A `Token` object has four public fields (also called *instance variables*): an `enum Match` field named `match` that is the token's symbolic name; a `String str` field that is the token's lexeme derived from the input stream (and is returned by the `toString()` method in this class); an `int lno` field that is the line number, starting at one, of the input stream where the token appears; and a `String line` field that is the source line of the input stream where the token appears.

For the purposes of compatibility, the `Scan` class defines methods `hasNext()` and `next()` that behave exactly like their counterparts in the Java `Scanner` class. The `boolean hasNext()` method returns `true` if and only if the input stream has additional tokens, in which case the `Token next()` method returns (and consumes) the next `Token` object from the input stream.

Tokens (continued)

For example, consider a `grammar` file (directory `IDNUM`) with the following lexical specification:

```
skip WHITESPACE '\s+'
NUM '\d+'          # one or more decimal digits
ID '[A-Za-z]\w*' # a letter followed by zero or more "word" chars
```

When you run `plccmk` on this specification, it creates the file `Token.java` in the Java subdirectory having a public inner `enum` class named `Match` whose elements consist of the following identifiers and associated patterns:

```
WHITESPACE ("\\s+", true) // the 'true' means it's a skip spec.
NUM ("\\d+")
ID ("[A-Za-z]\\w*")
```

Any Java file that needs to use the `enum` values `NUM` and `ID` can refer to them symbolically as `Token.Match.NUM` and `Token.Match.ID`.

Running the `Scan` program in the Java directory takes character stream input from standard input (typically your keyboard) and prints all of the resulting tokens to standard output (typically your screen), one token per line. Each printed line gives the line number where the token appears, the token name (`NUM` or `ID` in this example), and the lexeme (printed in single quotes). Any input that does not match one of the skip or token specifications prints the representation of an `$ERROR` token.

Tokens (continued)

The `printTokens()` method in the `Scan` class produces the output described on the previous slide. Here is the code for this method:

```
public void printTokens() {
    while (hasNext()) {
        Token t = next();
        String s;
        switch(t.match) {
        case $ERROR:
            s = t.toString();
            break;
        default:
            s = String.format("%s '%s'", t.match.toString(), t.str);
        }
        System.out.println(String.format("%4d: %s", lno, s));
    }
}
```

Running `plccmk` in the directory containing the `grammar` file generates Java source files in the `Java` directory and compiles them. You can then run the `Scan` program as follows:

```
java -cp Java Scan
```

Tokens (continued)

Two special "tokens" are defined in the `Token.java` class:

```
$ERROR
$EOF
```

Since the PLCC lexical specification requires that token symbol names begin with an uppercase letter, these "tokens" cannot be confused with language-specific token symbols.

The `$ERROR` "token" is produced when the scanner encounters an input character that does not match the beginning of any skip or token specification. The `toString()` value of this token is of the form `!ERROR(...)`, where the '`...`' part displays the offending character. In the context of syntax analysis (which we cover later), such a character cannot be part of a syntactically correct program, so syntax analysis will terminate with an error.

The `$EOF` "token" is produced when the scanner encounters end-of-file on the input stream. In the context of syntax analysis, encountering end-of-file signals that there are no further input tokens to process, so syntax analysis terminates (possibly prematurely).

Specifying Syntax

A *language*, in computer science theory, is a set of strings, where a *string* is a finite sequence of *symbols* chosen from a given *alphabet*. Computer science theory deals in part with how to specify languages using things such as Nondeterministic Finite Automata (NFAs), Context-Free Grammars (CFGs), and Turing Machines (TMs).

A *programming language* also defines a language in the theory sense, except that the strings in a programming language are called *programs*, and the symbols are called *tokens* (which we discussed in Slide Set 0). The *syntax* of a programming language is a set of rules used to specify the programs in the language. Most programming languages use a context-free grammar (or something close to it) to specify what sequences of tokens belong to the language.

A programming language also defines the run-time behavior of a program, called its *semantics*, which we discuss at length later.

Our first step is to describe a formal way in which we can define the syntax of a programming language. We start out with two simple examples of languages that describe familiar data structures. (These are not "programs" in the usual sense of the word, but they will at least get us started with how to specify syntax.)

Backus-Naur Form (BNF)

BNF is a meta-language used to specify a context-free grammar. Almost every modern programming language uses some sort of BNF notation to define its syntax. We work first with examples of languages that define two simple data structures: lists and trees. Remember that the "alphabet" of a programming language is a set of tokens, so any definition of the syntax of a language must first specify its tokens. In Slide Set 0, we showed how to specify tokens using PLCC.

Our first example is to define a language whose "programs" are lists of numbers. Some sample "programs" in this language are:

```
(3 4 5)
(    7 11   )
()
```

Backus-Naur Form (BNF) (continued)

Here is a BNF definition of this language, using three BNF formulas:

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

In these formulas, the token names `LPAREN` and `RPAREN` stand for left parenthesis '`(`' and right parenthesis '`)`', respectively. The token name `NUM` stands for any (unsigned) decimal number. [Think about what regular expressions would match these.] Conforming to PLCC specifications, we use all-UPPERCASE letters for our token names.

BNF: (continued)


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

Every BNF formula has the form

LHS `::=` *RHS*

The *LHS* (Left-Hand Side) of a BNF formula always has the form `<nonterm-symbol>` where `nonterm-symbol` is an identifier. PLCC requires that the first character of this identifier be a lowercase letter. A `<nonterm-symbol>` expression is called a *nonterminal*. In the example above, the nonterminals are `<lon>` and `<nums>`.

The *RHS* (Right-Hand Side) of a BNF formula is a (possibly empty) ordered list of token names and nonterminals.

Notes: The term *syntactic category* is sometimes used instead of the term *nonterminal*, and the term *terminal* is sometimes used instead of *token name*. Instead of using a token name such as `LPAREN`, some BNF formulas just use the corresponding actual character string such as '`(`'. In the examples on slide 1.2, you can see that we have also skipped whitespace.

BNF (continued)


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

BNF has some shortcuts **that we do not use** but that you may encounter in your reading. These shortcuts are usually called Extended BNF, or simply EBNF. For example, instead of writing two different formulas with `<nums>` on the LHS, one can use *alternation* notation "|":

```
<nums> ::= NUM <nums> | ϵ
```

Note: 'ϵ' means the empty string.

One could also use the *Kleene star* notation to define `<nums>`:

```
<nums> ::= { NUM }*
```

Note: We use a variant of the Kleene star notation later.

Parsing (syntatic derivation):


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

A BNF grammar defines the set of all "legal" token sequences that conform to the grammar rules. This set is the *language* of the grammar, where we use the term "language" in the sense of computational theory. In the context of programming languages, these legal token sequences are called "programs". We also use the term *syntax rules* to refer to the rules given by a BNF grammar, and we use the term *syntactically correct* to refer to token sequences that conform to the grammar rules. Finally, when there is little chance for confusion, we use the term *sentence* (in the sense of computational theory) to refer to a finite length sequence of lexemes.

If we had a set of grammar rules for the Java programming language, for example, then the set of all sentences that conform to this grammar would be the set of all syntactically correct Java programs.

In these notes, we casually use the term *token* to refer both to the symbolic name of the abstraction (like '`NUM`') used in a BNF rule and to its corresponding *lexeme* (like '`23`') used in a program. In most instances, you should not have difficulty understanding which meaning is intended.

Parsing (syntatic derivation):


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

A grammar can be used:

- to construct syntactically correct sentences, or
- to check to see if a particular target sentence belongs to the language (*i.e.*, is syntactically correct).

A programmer's job is to construct syntactically correct programs in a programming language, illustrating the first use of a grammar. This activity is called *programming*.

A compiler's job is, in part, to check a particular sentence for syntactic correctness, illustrating the second use of a grammar. This activity is called *syntax checking* or *parsing*.

Given a grammar, we want to generate an algorithm to carry out parsing. A program [written in Java, or C, or whatever] that carries out this algorithm is called, unsuprisingly, a *parser*. In this course, all of our parsers are Java programs.

Parsing (continued):

Given a BNF grammar, we describe here an algorithm (written in English, not Java) that parses a sentence in the language using what's called a *leftmost* derivation. The algorithm returns "success" if the parse is successful, "failure" otherwise.

1. Find the *start symbol* of the grammar. (For all of the grammars we define in these notes, the start symbol is the *first LHS nonterminal* in the set of grammar rules.) This nonterminal becomes the initial *sentential form* of the derivation. (A "sentential form" is a sentence-like thing that is a sequence of nonterminals, token names, and lexemes. A "sentence" only has lexemes. Once all of the nonterminal symbols have been removed and the token names have been replaced by lexemes using the steps given below, the result is a sentence.) Set the *unmatched* sentence to the target sentence.
2. Repeat Step 3 (see the next page) until the sentential form is a sentence – *i.e.*, consists only of lexemes (no nonterminals, no token names).

Parsing (continued):

3. (a) If the leftmost unmatched term in the sentential form is a **token name**, match it with the leftmost string in the unmatched sentence. [Here, *match* means that the token name in the sentential form describes exactly the leftmost string to be matched. For example, the token name `LPAREN` matches the string '`(`' and `NUM` matches the string '`42`'.] If there is no match, return failure. If there is a match, replace the leftmost token name in the sentential form with its matching string (its lexeme) from the unmatched sentence and remove the matched string from the unmatched sentence.

   (b) If the leftmost unmatched term in the sentential form is a **nonterminal**, choose a rule from the grammar with this nonterminal as its LHS and *replace* the nonterminal with the (possibly empty) RHS of the chosen rule. [Which rule to choose depends on finding a rule that is most likely to complete the derivation. For some grammars, there is only one choice: these grammars are said to be *predictive*. All of the grammars we use in this course are predictive.] If no rule can apply, return failure.

4. If the unmatched sentence is empty, return success: the target sentence has been successfully parsed. Otherwise, return failure.

It is possible, for some grammars, that Step 3 loops indefinitely. However, for all of the grammars that we encounter in this course, this step exits in a finite number of iterations.

Parsing (continued):


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

We perform a *leftmost derivation* of the target string `( 14 6 )`. Always start with the first nonterminal in the grammar (the *start symbol* – in this case it's `<lon>`) as the sentential form:

sentential form	**unmatched sentence**	**algorithm step taken**
<lon>	(14 6)	1
⇒ **LPAREN** <nums> RPAREN	(14 6)	3b
⇒ (**<nums>** RPAREN	14 6)	3a
⇒ (**NUM** <nums> RPAREN	14 6)	3b
⇒ (14 **<nums>** RPAREN	6)	3a
⇒ (14 **NUM** <nums> RPAREN	6)	3b
⇒ (14 6 **<nums>** RPAREN	)	3a
⇒ (14 6 _ **RPAREN**	)	3b (**<nums>** ⇒ empty)
⇒ (14 6)		3a
⇒ *success!*		4

In the above derivation, the leftmost unmatched token name or nonterminal is shown in **`boldface`**. The underlined parts are the matched lexemes for the unmatched token name as described in rule 3a, or the chosen substitutions for the LHS nonterminals as described in rule 3b.

A derivation ends successfully when there are no token names or nonterminals in the sentential form and no unmatched lexemes.

Parsing (continued):


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

Next we attempt a parse of '`( 14 ( 6 )`' using this grammar. As before, we start with the first nonterminal `<lon>` as the sentential form:

sentential form	**unmatched sentence**
<lon>	(14 (6)
⇒ **LPAREN** <nums> RPAREN	(14 (6)
⇒ (**<nums>** RPAREN	14 (6)
⇒ (**NUM** <nums> RPAREN	14 (6)
⇒ (14 **<nums>** RPAREN	(6) – *try first <nums> rule ...*
⇒ (14 **NUM** <nums> RPAREN	?? NUM doesn't match "("
⇒ (14 **<nums>** RPAREN	(6) – *try second <nums> rule ...*
⇒ (14 **RPAREN**	?? RPAREN doesn't match "("
⇒ *failure!*	

We can conclude that the string '`( 14 ( 6 )`' does not conform to the grammar specifications, so it is *not* a "list of numbers".

BNF (continued)

A *tree* is another data type that can be specified in BNF form:

```
<tree>  ::= NUM
<tree>  ::= LPAREN SYMBOL <tree> <tree> RPAREN
```

Here, `SYMBOL` is a token name that represents a string of alphanumeric characters starting with a letter. [Think of how to specify this using a regular expression.] The `NUM`, `LPAREN` and `RPAREN` token names are as before.

Here are some examples of trees that you can check for syntactic correctness using the parsing algorithm given above.

```
3
( bar 1 ( foo 1 2 ) )
( bar ( biz 3 4 ) ( foo 1 2 ) )
```

BNF (continued)


```
<tree>  ::= NUM
<tree>  ::= LPAREN SYMBOL <tree> <tree> RPAREN
```

In a given sentential form, the nonterminal `<tree>` can be replaced by the right-hand side of *either* rule having `<tree>` as its left-hand side. For example, parsing the string '`( bar 1 ( foo 1 2 ) )`' gives this:

<tree>
⇒ **LPAREN SYMBOL** <tree> <tree> RPAREN
⇒ (bar **<tree>** <tree> RPAREN *[*see below...]*
⇒ (bar **NUM** <tree> RPAREN
⇒ (bar 1 **<tree>** RPAREN
⇒ (bar 1 **LPAREN** SYMBOL <tree> <tree> RPAREN RPAREN
⇒ ...

Note: this line collapses two* ***match* *steps into one:*

LPAREN ⇒ '('
SYMBOL ⇒ bar

Parsers

Recall our grammar for a list of numbers (directory LON):

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

We have seen how to take a sentence in this language and use a leftmost derivation to parse it. What can we do to automate this algorithm?

A *parser* is a program that takes as input a sentence in a language and that carries out a parse of the sentence, producing either success or failure. Building a parser from the BNF specification of a language is conceptually simple: we only need to write a program to carry out the steps of the parsing algorithm. The difficulty is the algorithm's essential *nondeterminism*: given a nonterminal in a sentiential form, how do we replace this nonterminal with the right-hand side of the "correct" grammar rule having this nonterminal as its left-hand side? (For some grammars, there may even be more than one "correct" rule that leads to a successful parse – such grammars are said to be *ambiguous*.)

Parsers (continued)


```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

There are several tool sets that can be used to convert a BNF-like grammar specification into a parser. Many of these tool sets use C or C++ as the target language: input to such a tool set is a grammar specification, and the output is a set of target language programs that, when compiled, produces a parser. Learning how to use some of these tool sets can be a daunting task, and the generated parsers can be obscure. Furthermore, most of these tool sets are split into two separate parts: a lexical analysis (scanning) part and a syntax analysis (parsing) part.

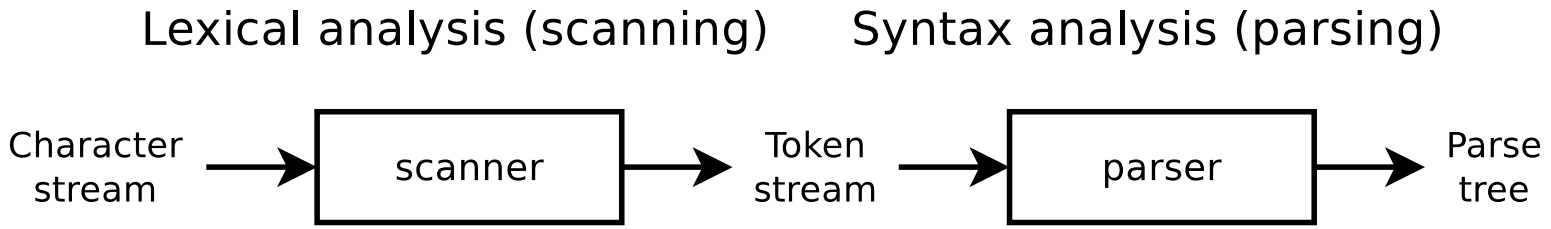
The PLCC tool set is designed to make it easy to build a scanner and parser for a large collection of programming languages. While it is not "industrial strength" like many of the standard tool sets, it is easy to learn. The target language for PLCC is Java, so all of the source code that PLCC generates consists of self-contained Java programs. To use PLCC, you only need to be comfortable reading and writing programs in Java.

To say that a parser returns only success or failure is cold comfort, since in most cases you want to "run" a program once it parses successfully. So a parser generally does one of two things: it "runs" the program *as it carries out the parse algorithm* (which is called *interpretation on the fly*), or it produces some form of output that can later be used to run the program *once the parse is complete*. PLCC uses the latter approach, since generating a parser is simpler if it is divorced from any attempts to carry out run-time behavior during the parse.

The input to a PLCC parser is a token stream – specifically, the tokens produced by an instance of the `Scan` class (see Slide Set 0). The output of a PLCC parser is a *parse tree* of a program: more specifically, it is a Java object that is the root of the parse tree.

Parsers (continued)

The following diagram shows how the output of a scanner (a program that implements lexical analysis) becomes input to a parser (a program that implements syntax analysis). The output of a parser is a *parse tree* – which we describe in more detail later.



For a particular language whose PLCC `grammar` file specifies the language tokens and BNF grammar, the `plccmk` program generates and compiles Java source files that implement a scanner and parser directly from the `grammar` file.

We start with the list-of-numbers (`LON`) example to show how this works.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

From each BNF grammar rule for a language, PLCC generates Java class. A rule such as

```
<lon> ::= LPAREN <nums> RPAREN
```

generates the Java class `Lon`. PLCC gets the name `Lon` from the LHS nonterminal `<lon>` of this grammar rule by converting the first letter of this nonterminal name to uppercase.

PLCC determines the *fields* of a Java class generated from a BNF grammar rule from the RHS terms of the rule – specifically, the terms that are enclosed in angle brackets `<...>`. The RHS of the `lon` grammar rule given above has one term in angle brackets: `<nums>`. The corresponding field name in the `Lon` class is `nums` (Java field names always begin with a lowercase letter). The *type* of this field is `Nums`, because `Nums` is the name of the Java class corresponding to the nonterminal `<nums>`.

Since `<nums>` appears as the LHS nonterminal on the second and third grammar rules for this language, PLCC generates a corresponding `Nums` class.

Parsers (continued)

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> ::= NUM <nums>
<nums> ::=
```

However, there are two grammar rules with `<nums>` as the LHS nonterminal. PLCC generates the class name `Nums` automatically (by converting the first character of the LHS nonterminal name to uppercase), but PLCC must generate a *unique* Java class name for each grammar rule. We accomplish this by annotating the LHS nonterminal on each of these lines with a Java class name that is different from `Nums` and that distinguishes one from the other. We modify these grammar rules with annotations as follows:

```
<nums>:NumsNode  ::= NUM <nums>
<nums>:NumsNull  ::=
```

(Any Java class names are OK for these annotations, but good naming conventions should prevail, and the names must be unique among the Java class names that PLCC generates.) A colon is used to separate the nonterminal from its annotated Java class name. The RHS entries of these grammar rules are unchanged.

With these modifications, PLCC generates two new classes, `NumsNode` and `NumsNull`. Both of these classes are declared to extend the `Nums` class, so an instance of a `NumsNode` class, for example, is also automatically an instance of its parent `Nums` class.

Here is a complete text file representation of Language LON, including its lexical and syntax specification sections, suitable for processing by plccmk. Notice that we have included the annotations for the two <nums> rules as described on the previous slide. Also notice that a line containing a single '%' separates the lexical specification from the syntax specification (BNF rules).

```
# Language specification for a list of numbers
# Lexical spec
skip WHITESPACE '\s+'
NUM '\d+'
LPAREN '\('
RPAREN '\)'
%
# Syntax spec (BNF rules)
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= NUM <nums>
<nums>:NumsNull   ::=
%
```

Parsers (continued)

```
# Language specification for a list of numbers
# Lexical spec
skip WHITESPACE '\s+'
NUM '\d+'
LPAREN '\('
RPAREN '\)'
%
# Syntax spec (BNF rules)
<lon>            ::= LPAREN <nums> RPAREN
<nums>:NumsNode  ::= NUM <nums>
<nums>:NumsNull  ::=
%
```

To summarize, the top lines of the file, up to the first percent (`%`) line, constitute the *lexical specification* of the language. For this language, these lines say that whitespace (including spaces, tabs, and newlines) should be skipped, that a `NUM` is a string of one or more decimal digits, and that `LPAREN` and `RPAREN` match the characters '`(`' and '`)`', respectively.

The remaining lines of the file constitute the *syntax specification* of the language, given in BNF form. PLCC takes this file as input and generates a collection of Java source files in a `Java` subdirectory that implement a scanner and parser for the language.

```
# Language specification for a list of numbers
# Lexical spec
skip WHITESPACE '\s+'
NUM '\d+'
LPAREN '\('
RPAREN '\)'
%
# Syntax spec (BNF rules)
<lon>            ::= LPAREN <nums> RPAREN
<nums>:NumsNode  ::= NUM <nums>
<nums>:NumsNull  ::=
%
```

Observe that the `LPAREN` and `RPAREN` token specifications use regular expressions having a backslash '`\`'. This is because parentheses have a special meaning in regular expressions, and the backslash voids this special meaning. So, for example, the regular expression `'\('` really matches the left parenthesis symbol. (You should take this opportunity to review how to write and interpret regular expressions in the Java `Pattern` class.)

To complete this example, assume that you have created a directory named `LON`, and that in this directory you have created a file named `grammar` that contains lines appearing above. In your `LON` directory, run the `plccmk` script as follows:

```
plccmk
```

Your script output should appear as follows:

```
Nonterminals (* indicates start symbol):
 *<lon>
  <nums>

Abstract classes:
  Nums

Source files created:
  NumsNode.java
  ...
```

```
<lon>              ::= LPAREN <nums> RPAREN
<nums>:NumsNode    ::= NUM <nums>
<nums>:NumsNull    ::=
```

(The above text shows just the syntax section of the `grammar` file we have been examining.) In your `LON` directory, change to the subdirectory named `Java`. The `plccmk` command has created this subdirectory and populated it with Java source code generated by `plcc.py`. In this subdirectory you will find (among other things) the following Java source files:

```
Lon.java
Nums.java
NumsNode.java
NumsNull.java
```

Each of these corresponds to one or more of the grammar rule lines. For example, the line beginning with `<lon>` results in the file `Lon.java` being created in the `Java` subdirectory. As you can see from looking at the Java code in the `Java` subdirectory for the `NumsNode` and `NumsNull` classes, both of these classes extend the `Nums` abstract class. This is because the `<nums>` nonterminal appears as the LHS of two grammar rules.

For every grammar rule, PLCC creates a Java class uniquely associated with the rule. For grammar rules that have the same nonterminal appearing on the LHS of multiple rules, PLCC creates an abstract class based on the nonterminal name, and the annotated class names become derived classes of this abstract class.

The first BNF nonterminal in a language's syntax specification section is the *start symbol* for the language. Given a program in the language, the parser produces an instance of the class of the start symbol, which is the root of the parse tree for the program.

Consider Language `LON`, whose `grammar` file syntax specification section is shown below. The start symbol is `<lon>`, and the corresponding class name is `Lon`. Given a program in Language `LON`, the output of the parser is an instance of the `Lon` class. We will see shortly how to interpret this instance as the root of a *parse tree*.

```
<lon>              ::= LPAREN <nums> RPAREN
<nums>:NumsNode    ::= NUM <nums>
<nums>:NumsNull    ::=
```

Parsers (continued)

From a particular language defined by a language specification file, PLCC generates a stand-alone parser with class name `Parse` that uses the Java classes created from processing the lexical and syntax specification sections. To run this parser in language directory `LON`, for example, run the `Parse` class file using '`-cp Java`' on the `java` command line so the Java interpreter will know where to find the `.class` files. This program displays a prompt '`--> `', after which it reads, from standard input, "programs" in Language `LON` to parse. (Standard input typically comes from your keyboard, sometimes called the *console*.) If the parse for a particular program succeeds, the `Parse` program displays the string `"OK"`. If the parse fails, it displays an error message indicating how the parse failed.

For example, in the `LON` directory (after having run `plccmk`), enter the following command:

```
java -cp Java Parse
```

With a standard input of '`( 14 6 )`', your input and output looks like this:

```
--> ( 14 6 )
OK
```

With a standard input of '`( 14 ( 6 )`', your input and output looks like this:

```
--> ( 14 ( 6 )
%%% Parse error: Nums cannot begin with LPAREN
```

You can also use the `parse` shell script, which is equivalent to running `java -cp Java Parse`.

```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= NUM <nums>
<nums>:NumsNull   ::=
```

When invoked with command-line arguments, the `Parse` program reads the files given as filename arguments and processes them as if they were entered from standard input.

Three command-line arguments have special meaning when running the `Parse` program:

- The '`-n`' command-line argument disables displaying the '`--> `' prompt when reading programs from standard input.
- The '`-t`' command-line argument toggles displaying a *parse trace* as programs are parsed. A parse trace is a text representation of the parse tree generated by the parser: an example of such a parse trace appears on Slide 1.30. It defaults to not displaying the parse trace.
- The '`-v`' command-line argument toggles displaying the names of the command-line files when processing them in left-to-right order. It defaults to not displaying the name.

The same command-line arguments pertain to the `Rep` program which we describe beginning with the next slide.

```
<lon>              ::= LPAREN <nums> RPAREN
<nums>:NumsNode    ::= NUM <nums>
<nums>:NumsNull    ::=
```

PLCC also generates an interactive parser/evaluator called `Rep` that resides in the `Java` subdirectory along with the `Parse` program. `Rep` executes a "Read-Eval-Print" loop that displays a prompt, *Reads* program input from standard input, *Evaluates* (parses) the program, and *Prints* the result obtained from "running" the program. Normally, this result is a representation of the "value" of the program's parse tree: an instance of the Java class defined by the start symbol of the language's BNF grammar.

How to determine the "value" of a parse tree is the essence of *semantic analysis*, which we describe in detail for each of the languages we consider in these notes. In some cases, this value may be just a re-cast version of the program text; in others, it may be the numeric value of a function application.

Parsers (continued)


```
<lon>              ::= LPAREN <nums> RPAREN
<nums>:NumsNode    ::= NUM <nums>
<nums>:NumsNull    ::=
```

By default, the value of a parse tree is simply the `toString()` value of the parse tree Java instance. For Language `LON`, this value is a Java `String` of the form `Lon@xxxx`, where the `xxxx` part is the hexadecimal address of where the object resides in memory. Observe that `Lon` is the Java class that defines the start symbol of the `LON` BNF rules.

The '`-t`' command-line argument turns on a parse trace when used with the `Parse` or `Rep` programs. The next slide shows the output using this feature.

Here's a sample interaction using this trace feature for Language `LON`, with the output edited for the sake of readability:

```
$ java -cp Java Rep -t
--> (14 6)
<lon>
| LPAREN "("
| <nums>:NumsNode
| | NUM "42"
| | <nums>:NumsNode
| | | NUM "6"
| | | <nums>:NumsNull
| RPAREN ")"
Lon@372f7a8d
```

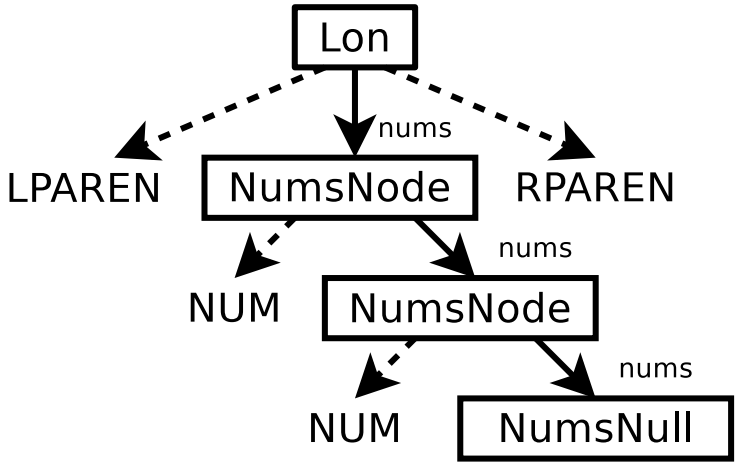
In this example, the root of the parse tree is a `Lon` object whose `nums` field is an instance of `NumsNode`. This instance in turn has a `nums` field that is also an instance of `NumsNode`. And finally, this instance has a `nums` field that is an instance of `NumsNull`. A `NumsNull` object has no fields. The trace also shows how each token is matched by the parser, along with the lexeme from the input program that matched the token. The line '`Lon@372f7a8d`' shows the result of the parse, which is a `Lon` object representing the root of the parse tree. The '`@372f7a8d`' part represents the location in memory where the `Lon` object resides.

Using this trace feature, we can construct a visual representation of the parse tree whose root is an object of type `Lon`. The diagrams below show the parse trace output on the left and the resulting parse tree on the right. Nodes in the tree are represented by boxes. The dashed lines point to tokens that are part of the conceptual parse tree but that do *not* correspond to fields in the node. (Normally this is the case when the BNF entries on its RHS are tokens that are fixed – such as LPAREN.)

Rep -t output

```
<lon>
| LPAREN "("
| <nums>:NumsNode
| | NUM "42"
| | <nums>:NumsNode
| | | NUM "6"
| | | <nums>:NumsNull
| RPAREN ")"
```

Parse tree



Parsers (continued)

To make the connection clear between a grammar rule and its PLCC-generated Java class, in these class notes we display the grammar rules in the following way:

```
<lon>              ::= LPAREN <nums> RPAREN
                   Lon(Nums nums)
<nums>:NumsNode    ::= NUM <nums>
                   NumsNode(Nums nums)
<nums>:NumsNull    ::=
                   NumsNull()
```

The item in the box following a grammar rule is the *Java signature of the Java class constructor* corresponding to the class PLCC generates from the grammar rule. So the box

```
Lon(Nums nums)
```

means that the constructor for the PLCC-generated class `Lon` has a single parameter `nums` of type `Nums`.

There is a one-to-one correspondence between the types and formal parameters in the constructor and the types and field names in the class. More specifically, when the constructor is invoked, the constructor body simply copies the values of its parameters into the corresponding field names of the instance being constructed.

As you can see from above, the `NumsNull` constructor takes no parameters, and the `NumsNull` class has no corresponding fields.

```
<lon>              ::= LPAREN <nums> RPAREN
<nums>:NumsNode    ::= NUM <nums>
<nums>:NumsNull    ::=
```

The parse tree for a program in this language accurately represents the length of the list of numbers given as input – count the number of instances of `NUM` in the parse trace given on Slide 1.30 above – but the actual `NUM` tokens are not preserved in the parse tree. The problem is that the parse tree instances of `NumsNode` objects do not have fields corresponding to their `NUM` tokens.

To remedy this situation, we want to define a field in the `NumsNode` class that captures the `NUM` token obtained by the parser, which in turn allows us to retrieve the token's lexeme. We already use angle brackets for all *nonterminals* on the RHS of grammar rules, and these nonterminals automatically become fields in the Java class. So to capture *tokens* on the RHS, we use angle brackets for these tokens as well. This means that the `NumsNode` line now looks like this:

```
<nums>:NumsNode   ::= <NUM> <nums>
```

The '`<NUM>`' entry creates a field named `num` (which is the token name '`NUM`' converted to lowercase) of type `Token` (since `NUM` is the name of a token in the `grammar` file).

Observe that it's unnecessary to capture tokens that always have the same lexeme, like `LPAREN`: every instance of the `LPAREN` token looks like every other instance, so there's no need to distinguish among them. This is not so with tokens that can take on multiple lexeme values, such as `NUM`. Indeed, for a list of numbers, knowing exactly *what* numbers are in the list can be essential – for example, if you want to find the sum of the numbers in the list. If these items do not appear as fields in the PLCC-generated classes, their values do not appear in the parse tree, and so their values are not retrievable after the parse.

The revised grammar now looks like this:

```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

Since we now have two fields in the `NumsNode` class, we need to modify the signature of the `NumsNode` constructor, as shown here (compare with slide 1.31):

```
<lon>             ::= LPAREN <nums> RPAREN
                      Lon(Nums nums)
<nums>:NumsNode   ::= <NUM> <nums>
                      NumsNode(Token num, Nums nums)
<nums>:NumsNull   ::=
                      NumsNull()
```

Parsers (continued)

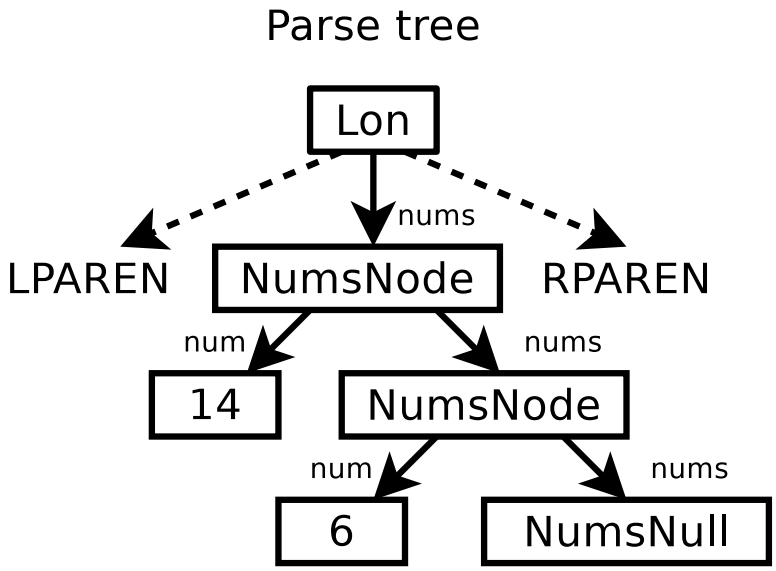
```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

We now have two fields in the `NumsNode` class: `num` (of type `Token`) and `nums` (of type `Nums`). When the parser generates a `NumsNode` object, it fills in its `num` field with the value of the `Token` object. The `toString()` method applied to this object retrieves the lexeme of the scanned token, which is a string of decimal digits. The rest of the `NumsNode` object is the same as before.

The following diagram shows the parse tree for the string

```
( 14 6 )
```

using this revised grammar. Notice that each `NumsNode` object now has a `num` field (a `Token`) containing decimal digits.



Let's return to our tree example (directory TREE). Here is a modified grammar for a tree as originally given on slide 1.12 that takes into account the PLCC requirements for unique class names and the introduction of fields for the NUM and SYMBOL tokens. Notice the Java signatures for the class constructors shown in boxes.

```
<tree>:Leaf      ::= <NUM>
                 Leaf(Token num)
<tree>:Interior  ::= LPAREN <SYMBOL> <tree> <tree> RPAREN
                 Interior(Token symbol, Tree tree, Tree tree)
```

But there is a problem with the Interior constructor signature. Java does not allow multiple field names or constructor formal parameters with the same name: specifically, in this case, there cannot be two field names with the name tree. Here is what PLCC has to say about this when given the above grammar (the line has been folded for clarity):

```
duplicate field name tree in rule RHS
LPAREN <SYMBOL> <tree> <tree> RPAREN
```

The solution is to use different identifiers for these field names and to their corresponding constructor formal parameters. PLCC allows duplicate RHS fields (in angle brackets) to be annotated – much like we have seen for duplicate LHS nonterminal names – with alternate names that avoid this conflict.

In the case we are considering, we can resolve this issue in the RHS of the `<tree>:Interior` grammar rule by using the identifier `left` for the first `<tree>` field and the identifier `right` for the second `<tree>` field, as shown here:

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
                    Interior(Token symbol, Tree left, Tree right)
```

In summary, this grammar rule creates:

- a class `Interior`
- that extends the abstract class `Tree` and
- that has a field `symbol` of type `Token`,
- a field `left` of type `Tree`, and
- a field `right` of type `Tree`.

Parsers (continued)

As a result, the following grammar is acceptable to PLCC:

```
<tree>:Leaf      ::= <NUM>
                 Leaf(Token num)
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
                 Interior(Token symbol, Tree left, Tree right)
```

When processed by PLCC, we get the following interaction using the `Rep` parser loop:

```
$ java -cp Java Rep
--> 3
Leaf@15db9742
--> (foo 5 8)
Interior@6d06d69c
--> (foo (bar 13 23) 8)
Interior@7852e922
--> (goo blah 99) % blah is the culprit here
%%% Parse error: Tree cannot begin with SYMBOL
-->
```

Examine the Java code for the `Interior` class. You will see the following fields:

```
public Token symbol;
public Tree left;
public Tree right;
```

Let's return to the list-of-numbers example.

```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

The `Nums` abstract class is extended by both `NumsNode` and `NumsNull` classes.

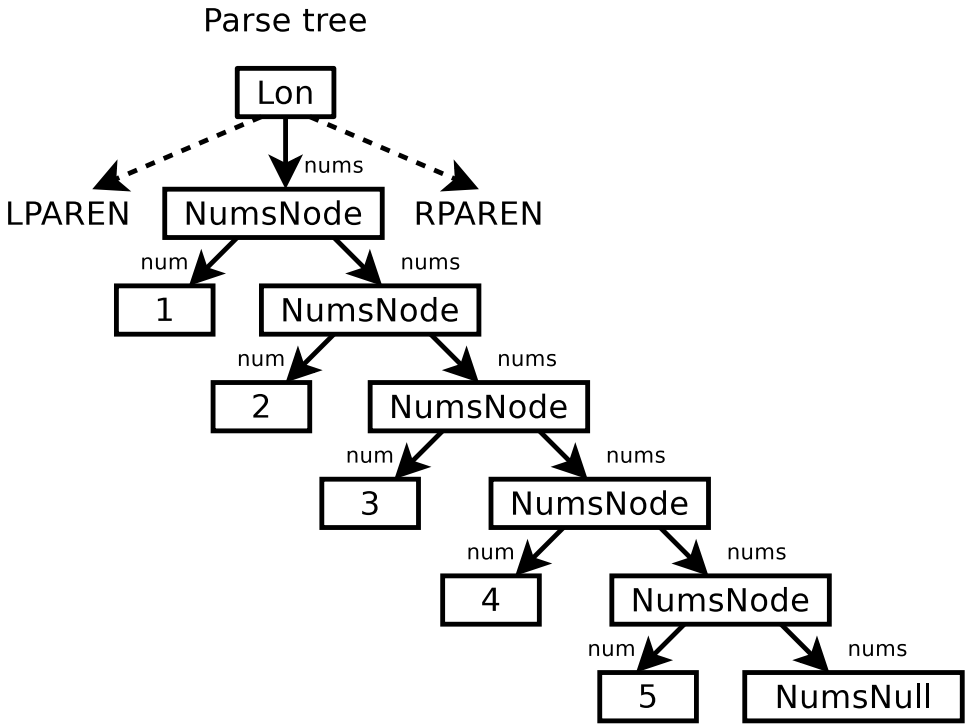
```
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

Clearly, when parsing the `<nums>` grammar rules, you get zero or more `NumsNode` instances but just one final `NumsNull` instance. Since the the RHS of the first `<nums>` rule has `<nums>` as its last entry (which means that this rule is *right recursive*), this rule results in a recursive parsing loop, ending only when there are no more `NUM` tokens in the program.

The parse tree for the following list-of-numbers

```
( 1 2 3 4 5 )
```

is shown here:



Because of this right-recursive looping, the nodes in the parse tree drift to the right as the number of elements in the list grows. Clearly a list of numbers with 100 entries would have 100 `NumsNode` nodes in the tree, weighted significantly to the right. How can we structure our tree nodes to avoid this?

This sort of looping occurs frequently in programming language specifications, and PLCC has a way to encode this. Instead of having two `<nums>` rules, with the first being right-recursive and the second having an empty RHS, we can re-write these rules using a special '`**=`' notation:

```
<nums>   **= <NUM>
         Nums(List<Token> numList)
```

The parser accumulates all of the `NUM` tokens into a single `numList` field. The `numList` field name is obtained from the `NUM` token name by converting all of its characters to lowercase and appending the string `List`.

Note: The use of '`**`' in the notation we have just introduced should suggest the *Kleene star* repetition notation used in EBNF as well as in regular expressions.

The modified list-of-numbers grammar is in the directory `LON2` as follows:

```
<lon>   ::= LPAREN <nums> RPAREN
<nums>  **= <NUM>
```

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

Here is an (edited) example of a parse trace for the list of numbers `(3 5 8 13)`:

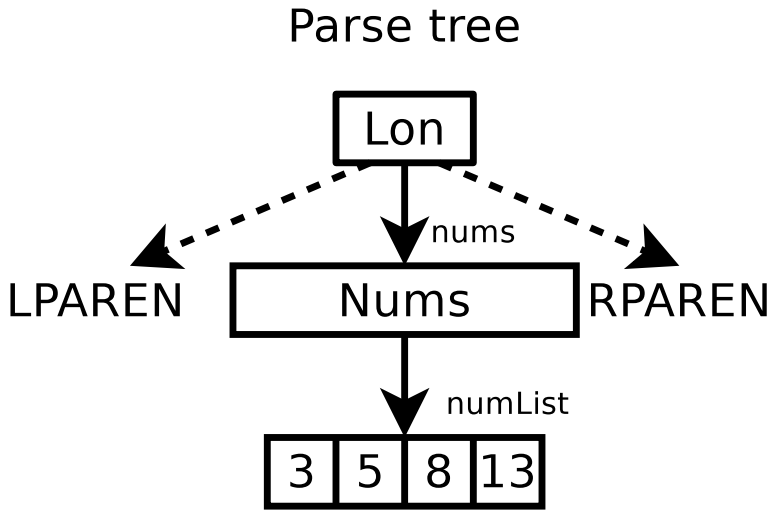
```
--> (3 5 8 13)
<lon>
| LPAREN "("
| <nums>
| | NUM "3"
| | NUM "5"
| | NUM "8"
| | NUM "13"
| RPAREN ")"
```

Compare this with the parse trace using the previous grammar that does not use the `**=` construct. The previous parse trace drifts to the right as additional `NUM` entries are encountered. Using the `**=` construct, the parse trace becomes flat.

PLCC grammar rules that use this construct are called *repeating grammar rules*. Repeating rules are useful in specifying most of our languages.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

The parse tree for the list of numbers `(3 5 8 13)` is shown here, using a box with compartments to represent a Java `List` structure:



This parse tree has fewer nodes, and the individual `NUM` token values are all packaged together into a single structure.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

How exactly can we have the `Rep` program access and display the values of the `NUM` fields from the parse tree? The `Rep` program first parses the program, yielding an instance of the start symbol class – an instance of `Lon`, in this case. It then runs (evaluates) this instance, which as we have seen defaults to displaying something like '`Lon@...`'.

This default behavior resides in the Java class `_Start`. This class defines a `void` method named `$run()` that simply displays the `toString()` value of its instance (see the Java code for `_Start.java` in the Java subdirectory of Language `LON`). Since the Java class `Lon` extends the `_Start` class, evauating the `$run()` method of a `Lon` object defaults to evaluating the `$run()` method of its superclass, which gives us the behavior we have already seen. `Rep` simply calls the `$run()` method on the `Lon` instance obtined by parsing the program.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

So we only need to redefine the `$run()` method in the `Lon` class to get the new behavior we want: to display the values of the `NUM` fields from the parse tree.

Fortunately, PLCC allows us to add methods to PLCC-generated source files by including the added methods in the `grammar` file. Every time `plccmk` is run, these added methods are incorporated into the Java source files automatically.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

For example, if we want to define the `$run()` method in the `Lon.java` source file, we put the following lines into the `grammar` file *in the semantics section following the syntax section*. The semantics section is separated from the syntax section by a line containing a single '`%`'.

```
Lon
%%%
    public void $run() {
        ...
    }
%%%
```

The `Lon` line tells PLCC that we are adding a method to the `Lon.java` file, and the two lines containing `%%%` bracket the Java code to be added. This technique can be used to add Java code to *any* PLCC-generated Java file arising from the BNF grammar lines. PLCC inserts this added code at the end of the class definition in the Java code generated automatically by PLCC for the class. In the case of the `Lon` class, the above Java code appears at the end of the automatically-generated `Lon.java` class code.

```
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
```

Recall that we want to define the `$run()` method in the `Lon` class so it displays the values of the tokens in the `numList` field. This field, an object of type `List<Token>`, is accessible in the `Lon` class as follows:

```
nums.numList
```

So we can display the `NUM` entries by iterating over this list, displaing the tokens (as Java `Strings`) as we do so. Here is the completed version of the PLCC specification that adds the `$run()` method to the `Lon.java` file to achieve our desired result. Since the parse tree does not include the tokens for parentheses, this code adds back the parentheses in the output to make the output look "pretty".

```
Lon
%%%
    public void $run() {
        System.out.print("( ");
        for (Token tok: nums.numList)
            System.out.print(tok.toString() + " ");
        System.out.println(")");
    }
%%%
```

Parsers (continued)

Here is the complete `grammar` file in code directory `LON2` with these changes:

```
# Lexical specification
skip WHITESPACE '\s+'
NUM '\d+'
LPAREN '\('
RPAREN '\)'
%
# Grammar
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM>
%

Lon
%%%
    public void $run() {
        System.out.print("( ");
        for (Token tok: nums.numList)
            System.out.print(tok.toString() + " ");
        System.out.println(")");
    }
%%%
```

Continuing with our list-of-numbers example, suppose we want our parser to require that the numbers in our list are separated by commas, like this:

```
(5, 8, 13, 21)
```

How can we devise a grammar that accommodates these separators?

PLCC provides a way to specify a token that serves as a *separator between items in a repeating grammar rule*. The separator must be a bare token name (no angle brackets) at the end of the rule, preceded by a '+' character. So if we want to separate the items in our lists by a comma (with token name COMMA), here is what the specification looks like:

```
skip WHITESPACE '\s+'
NUM '\d+'
LPAREN '\('
RPAREN '\)'
COMMA ','
%
<lon>  ::= LPAREN <nums> RPAREN
<nums> **= <NUM> +COMMA
%
```

While this grammar uses a COMMA separator between NUM items, the parser generates *exactly the same parse trees* for this language as for the previous specification.

Slide set 1a provides a summary of PLCC features. You should familiarize yourself with this section in preparation for material in subsequent parts of this course.

Static Properties of Variables

A *variable* in a program is a symbol that has an associated value at run-time. One of the principal issues in determining the behavior of a program is determining *how* to find the value of a variable at run-time. At any instance in time, the value associated with a variable is called a *binding* of the variable to the value.

An *expression* is a language construct that has a value at run-time. A variable, by itself, is therefore an expression, but other language constructs can also have values: for example `x+y` is an expression if `x` and `y` are numeric-valued variables.

A programming language that is designed solely for the purpose of evaluating expressions is called an *expression-based language*. Many of the languages we construct in these notes are expression-based, especially early on. `Scheme`, `ML`, and `Haskell` are examples of expression-based languages used in practice. Expression-based languages do their looping principally using recursion.

A programming language whose language constructs are designed to "do something" (such as assigning the value of an expression to a variable or displaying the value of an expression to standard output) is called an *imperative language*. In an imperative language, a language construct that "does somthing" is called a *statement*; Imperative languages are therefore often called *statement-based*. `C`, `Java`, and `Python` are examples of imperative languages used in practice. Imperative languages do their looping principally using a form of "goto".

Expression-based languages get their power from defining and applying *functions*, so another term describing such languages is *functional*.

Static Properties of Variables (continued)

Determing the value of an expression at run-time is at the heart of executing a program, particularly so in expression-based languages. Since most expressions involve variables, evaluating an expression requires determining the values of its constituent variables – in other words, finding the values bound to these variables.

At run-time, how can you find the value bound to a variable? There are two basic approaches:

- if the location of the value bound to a variable can be determined by *where* that variable appears *in the text of a program*, we call it *static binding*.
- if the location of the value bound to a varible can only be determined by *when* the variable is accessed *during program execution*, we call it *dynamic binding*.

Almost all programming languages commonly in use today use static bindings, principally because it is easier to reason (or prove things) about programs that use static bindings. You will have the opportunity to explore dynamic binding in your homework.

Static Properties of Variables (continued)

We consider only static bindings for now, so the program text tells us how to determine variable bindings.

For a given variable, the *scope* of the variable is the region of code in which that variable's binding can be determined. Consider the following Java program:

```
public class Foo {
   public static int y;
   public int z;
   public static void main(String [] args) {
     // args is local to main
     Foo f = new Foo(); // f is local to main
     int x = 1; // x is local in main
     Foo.y = 2; // y is static throughout in Foo
     f.z = 3; // z is known only within instances of Foo
   }
}
```

In the above code, the scope of `y` is *global*, from its declaration as a `public static` variable to the end of the class. The scope of `z` is *instance-global*, known only within (and throughout) instances of the class Foo. The scopes of `f` and `x` are *local*, from their declarations to the end of the `main` method body. The scope of `args` is also local, from the beginning of the method body to the end.

Static Properties of Variables (continued)

It is possible for one symbol to have multiple bindings depending on where it occurs in the program. Consider:

```
public class Bar {
  public static int x;
  public static void main(String [] args) {
    x = 3;
    System.out.println(x);
    { // beginning of block
      int x = 4;
      System.out.println(x);
    } // end of block
    System.out.println(x);
  }
}
```

When this program is run, the output appears as follows:

```
3
4
3
```

This is because the "`int x = 4;`" line defines a new variable `x` bound to the value 4 whose scope is from its point of declaration to the end of the *block* in which it is defined, as shown in the program comments. In this case, we say that the definition of `x` in the block *shadows* the global `int x`, and that the block definition *punches a hole in the scope* of the global definition.

Static Properties of Variables (continued)

Here is the `Foo` class given on the second previous slide:

```
public class Foo {
   public static int y;
   public int z;
   public static void main(String [] args) {
     Foo f = new Foo(); // f is local to main
     int x = 1; // x is local in main
     y = 2; // y is static throughout in Foo
     f.z = 3; // z is known only within instances of Foo
   }
}
```

Consider just the `main` procedure in this class:

```
   public static void main(String [] args) {
     Foo f = new Foo(); // f is local to main
     int x = 1; // x is local in main
     y = 2; // y is static throughout in Foo
     f.z = 3; // z is known only within instances of Foo
   }
```

In this method, the identifiers `f` and `x` are explicitly defined. In these cases, we say that these identifiers *occur bound* in the `main` procedure.

However, the variable `y` is not defined anywhere in the procedure `main`. In this case, we say that the identifier `y` *occurs free* in the `main` method, but it *occurs bound* in the enclosing class `Foo`.

The Lambda Calculus – OPTIONAL SECTION

The following grammar defines a formal language called "the lambda calculus". This language plays an important role in the foundations of computer science (similar to Turing Machines). The `PROC` token is the string `proc`, and the `SYMBOL`, `LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, and `DOT` tokens are straight-forward – see the examples below.

```
<exp> ::= <SYMBOL>
<exp> ::= PROC LPAREN <SYMBOL> RPAREN LBRACE <exp> RBRACE
<exp> ::= DOT <exp> LPAREN <exp> RPAREN
```

Consider the sentential form (remember what that means?) in this language obtained from the second grammar rule, where `s` replaces `<SYMBOL>`:

```
proc(s) { <exp> }
```

The occurrence of the symbol `s` in this expression is called a *variable declaration* that *binds* all occurrences of `s` that appear in `<exp>` unless some intervening declaration of the same symbol `s` occurs in `<exp>`. We say that the expression `<exp>` is the *scope* of the variable declaration for `s`.

Occurs Free, Occurs Bound (informal definitions):

A symbol `x` *occurs free* in an expression `E` if `x` appears somewhere in `E` in a way that is not bound by any declaration of `x` in `E`. A symbol `x` *occurs bound* in `E` if `x` appears in `E` in such a way that is bound by a declaration of `x` in `E`. It is possible for the same symbol to occur both bound and free in different parts of an expression. (Note that the declaration itself is not considered free or bound.)

```
proc(x) {x}                    ; x occurs bound
proc(x) {y}                    ; y occurs free
.proc(x) {x} (x)               ; first x is bound,
                               ; second is free
.proc(x) {x} (y)               ; y occurs free
proc(y) {.proc(x) {x} (y)}     ; y occurs bound
proc(x) {.proc(y) {x} (y)}     ; y occurs free
.t (u)                         ; t and u occur free
```

The Lambda Calculus (continued)


```
<exp> ::= <SYMBOL>
<exp> ::= PROC LPAREN <SYMBOL> RPAREN LBRACE <exp> RBRACE
<exp> ::= DOT <exp> LPAREN <exp> RPAREN
```

Formal definitions of *occurs free* and *occurs bound*:

For a Lambda Calculus expression `E`, a symbol `x` *occurs free* in `E` if

- *Rule 1*:
  `E` is a `<SYMBOL>` and `E` is the same as `x`.
  ```
  x                    ; x is free
  ```
- *Rule 2*:
  `E` is of the form `proc(y){E'}` where `y` is different from `x` and `x` occurs free in `E'`
  ```
  proc (y) {x}         ; x is free
  ```
- *Rule 3*:
  `E` is of the form `.E1(E2)` and `x` occurs free in `E1` or `E2`
  ```
  .proc(y) {x} (y)     ; x is free
  .proc(y) {y} (x)     ; x is free
  ```

The Lambda Calculus (continued)


```
<exp> ::= <SYMBOL>
<exp> ::= PROC LPAREN <SYMBOL> RPAREN LBRACE <exp> RBRACE
<exp> ::= DOT <exp> LPAREN <exp> RPAREN
```

For a Lambda Calculus expression `E`, a symbol `x` *occurs bound* in `E` if

- *Rule 1*:
  `E` is of the form `proc(y){E'}` where `x` occurs bound in `E'` or `x` and `y` are the same symbol and `y` occurs free in `E'`

```
proc(y){proc(x){x}}        ; x is bound
proc(y){y}                 ; y is bound
```

- *Rule 2*:
  `E` is of the form `.E1(E2)` and `x` occurs bound in `E1` or `E2`

```
.proc(y){proc(x){x}}(y)        ; x is bound
.proc(y){x}(proc(y){y})        ; y is bound
```

Lexical and Grammar specification for the Lambda Calculus:

```
# Lexical specification
skip WHITESPACE '\s+'
LPAREN '\('
RPAREN '\)'
LBRACE '\{'
RBRACE '\}'
DOT '\.'
PROC 'proc'
SYM '\w+'
%
# Grammar
<exp>:Var  ::= <SYM>
<exp>:Proc ::= PROC LPAREN <SYM> RPAREN LBRACE <exp> RBRACE
<exp>:App  ::= DOT <exp>rator LPAREN <exp>rand RPAREN
%
```

In the Lambda Calculus, if a symbol is bound by a declaration, we can easily determine the precise declaration that binds the variable. The Lambda Calculus is of interest theoretically, but it has no practical value as a programming language.

Static Properties of Variables (continued)

Most imperative programming languages are *block structured* and use *lexical binding*, another term for static scope rules. A *block* is a region of code introduced by one or more variable declarations and continuing to the end of the code where these declarations are active. In C, C++, and Java, blocks are delimited by matching pairs of braces '`{ . . . }`'.

In some languages, blocks may be *nested*, in which case variable bindings at outer blocks may be *shadowed* by bindings in inner blocks. Consider, for example the following C++ code fragment:

```
{ int x = 3;
   { int x = 5;
     cout << x << endl;
   }
   cout << x << endl;
}
```

This code displays 5 and then 3.

In block structured languages, a variable in an expression is bound to the variable with the same name in the *innermost* block that defines the variable. (Note that Java does not allow the same variable to be defined both in an outer block and in an inner block.)

Static Properties of Variables (continued)

Let's return to our C++ example. The following picture shows the blocks of the C++ program fragment given on the previous slide:

```
{ int x = 3;
    { int x = 5;
        cout << x << endl;
    }
    cout << x << endl;
}
```

To determine the binding of a variable in an expression, cross the boxes textually outwards (up) until a variable declaration with the same variable name is found.

When defining procedures in block structured languages, the formal parameter declarations are considered to be at the same lexical level as local variable declarations in the outermost block of the procedure. In the following C++ example, the formal parameter `x` is at the same lexical level as the local variable `y`:

```
int foo(int x) {
    int y;
    ...
}
```

PLCC

PLCC is the name of a tool set that takes a language specification file and generates a set of Java source files for an interpreter for the language.

The specification file is in three sections:

1. lexical specification
2. syntax
3. semantics

A line containing a single '`%`' is used to separate the lexical specification section from the syntax section and to separate the syntax section from the semantics secton.

A specification file can consist of just the lexical specification, in which case the PLCC tool set creates only a token scanner (`Scan`) for the language.

A specification file can consist of just the lexical specification and syntax, in which case the PLCC tool set creates only the scanner and parser (`Parse`) for the language.

The format of a language specification file is shown here:

```
# lexical specification
...
%
# syntax
...
%
# semantics
...
```

PLCC – Tokens

Comments may appear in a language specification file beginning with '#' and continuing to the end of the line, except inside a token specification regular expression or in Java code in the semantics section.

In the lexical specification section of a language specification file, a line is either a `skip` specification or a `token` specification. These specifications have the following form:

`{skip|token}` *NAME* `'`*re*`'`

where *NAME* is a string of uppercase letters, decimal digits, and underscores, starting with an uppercase letter, and *re* is a *regular expression* as defined by the Java `Pattern` class.

Here are some regular expression examples:

re	matches
`d`	the letter `d`
`\d`	a single decimal digit
`d+`	one or more `d`s
`\d+`	one or more decimal digits
`.`	any character
`\.`	the dot character (`.`)
`.*`	zero or more characters
`%.*`	the character `%` followed zero or more characters
`[a-z]`	any lowercase letter in the range `a` to `z`
`\w`	any letter (lowercase or uppercase), digit, or underscore
`[A-Za-z0-9_]`	the same as `\w`

PLCC regular expressions do not match anything that crosses a line boundary, so the regular expression `'.*'` matches everything up to the end of the current line, including the end-of-line marker.

PLCC – Tokens

A skip specification is used to identify things in a program that are otherwise not meaningful to the syntax structure of the program. We generally skip whitespace (spaces, tabs, and newlines) and language-defined comments. The format of a comment is language-dependent, but in our languages, a comment starts with a '`%`' character and continues to the end of the line. Here is an example of skip specifications for whitespace and comments:

```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
```

A token specification is used to identify things in a program that are meaningful to the syntax structure of the program. Examples are a language reserved word (such as `if`), a special character sequence (such as a left bracket '`[`'), a numeric literal (such as `346`), or a variable symbol (such as `xyz`). Here are specifications that match these tokens.

```
token IF 'if'
token LBRACK '\['
token LIT '\d+'
token VAR '[A-Za-z]\w*'
```

For token specifications, the initial '`token`' can be omitted. Notice that regular expression meta-characters such as '`[`' must be quoted with a backslash '`\`' if they are to be treated as ordinary characters.

PLCC – Tokens

The PLCC tool set uses the skip and token specifications to create source files `Token.java` and `Scan.java`.

The `Token.java` file defines the `Token.Match enum` class – one `enum` for each skip and token specification name. It also defines the structure of a `Token` object consisting of a `match` field of type `Token.Match`, a `str` field of type `String`, a `lno` field of type `int`, and a `line` field of type `String`. The `str` field consists of the characters in the input stream that match the token specification – the token's *lexeme*, the `lno` field contains the line number on the input stream where the token appears, and the `line` field contains the input stream line where the token appears. The `str` field always contains at least one character.

The `Scan.java` file defines an object that is constructed from an input stream (a `BufferedReader`). The `cur()` method delivers the current `Token` object to its client (skipping over strings that match skip specifications), and the `adv()` method advances to the next token. Multiple calls to `cur` without any intervening calls to `adv` returns the same token repeatedly.

Upon encountering the end of the input stream, `cur` returns a special `Token` object whose `match` field is `$EOF` and whose `toString()` representation is `!EOF`. The `cur` method is *lazy*, meaning that the `Scan` class does not read any characters from the input stream until necessary to satisfy an explicit `cur` method call.

PLCC – Tokens

If there are characters remaining in the input stream, the `cur()` method checks the token and skip regular expressions one-by-one, in the order in which they appear in the lexical specification. Each regular expression is matched against the current unmatched input, up to (and including) the end of the current line. If no characters match the regular expression, the next specification is tried.

If the regular expression is a skip specification that matches at least one input character and if no prior token specification has found a matching token candidate, the matched part is skipped and processing continues on the remainder of the input, *starting over from the first lexical specification*. If a prior token specification has found a matching token candidate, the skip specification is ignored.

If a token specification match of length at least one occurs, and if this match is longer than the match length of the previous token candidate (if any), this token specification is chosen as the current token candidate. If not, the previous token candidate is retained.

PLCC – Tokens

If – after iterating through all of the lexical specifications – a token candidate has been identified (first longest match), the token candidate is used to create an instance of the `Token` class. The `cur()` method advances the input stream beyond the characters matched, saves the `Token` object for possible future calls to `cur()`, and returns the `Token` object. If no token candidate has been identified, the `cur()` method returns a special `$ERROR` Token object. The `str` field of this object is a string of the form '`!ERROR(...)`', where the '`...`' part is (a representation of) the current input stream character where the match failed.

If the `cur()` method finds that there is a non-`null` saved `Token` object from a prior call to `cur()`, it simply returns that `Token` object without any further processing. Otherwise, processing occurs as described above.

The `adv()` method replaces the saved `Token` object with `null`, so that a subsequent call to `cur()` will be forced to get the next token from the input stream. (If it first detects that the saved token object is already null, it calls `cur()` to consume the next input token.)

As described earlier, the `cur` method returns a special `$EOF` token when it detects the end of the `BufferedReader` input.

PLCC – Syntax

The second section of a PLCC language specification file is the syntax section consisting of grammar rules in the style of Backus-Naur Form (BNF), as described in Slide Set 1. Recall that a BNF grammar rule has the following form:

```
LHS ::= RHS
```

where LHS (the *Left Hand Side*) is a nonterminal and the RHS (the *Right Hand Side*) is a sequence of nonterminals and terminals. The individual parts of a PLCC grammar rule, including the '`::=`' part, are separated by whitespace.

Nonterminals in PLCC are identifiers enclosed between angle brackets '<' and '>'. The identifier must begin with a lowercase letter and can consist of zero or more additional letters, digits, or underscores. Identifiers that match Java reserved words should be avoided.

Terminals in PLCC must begin with an uppercase letter and can consist of zero or more additional uppercase letters, digits, or underscores. A terminal must appear as the name of a token in the lexical specification section.

PLCC associates every grammar rule with a unique Java class with a class name derived from the LHS of the grammar rule by converting its first character to uppercase. Class names that match standard Java class names should be avoided.

PLCC – Syntax (continued)

If the LHS is a simple nonterminal, the Java class name associated with the BNF rule is the nonterminal name with its first letter converted to uppercase. In this example

```
<proc> ::= PROC LPAREN <formals> RPAREN <exp>
```

the Java class name is `Proc`.

If the LHS is a nonterminal annotated by adding a colon and a Java class name, then the class name associated with the BNF rule is the annotated Java class name. In this case, an abstract base class is also created whose Java class name is the nonterminal name with its first letter converted to uppercase (as described above), with the annotated Java class name as a subclass. In this example

```
<exp>:AppExp ::= DOT <exp> LPAREN <rands> RPAREN
```

the Java class name associated with this BNF rule is `AppExp`, and this class extends the base class `Exp`.

PLCC – Syntax (continued)

PLCC requires that there are no duplicates of Java class names associated with the given grammar rules. In particular, if two grammar rules have the same LHS nonterminal name, then their left-hand sides must have annotations giving distinct Java class names. For example, the grammar rules on Slide 1.3

```
<nums> ::= <NUM> <nums>
<nums> ::=
```

would not be acceptable to PLCC. The following annotations fix this:

```
<nums>:NumsNode ::= <NUM> <nums>
<nums>:NumsNull ::=
```

When LHS rules are annotated in this way, the nonterminal class name (`Nums`, in this example) becomes an abstract Java class, and the annotated class names are used to generate classes that extend the abstract class. In this example, both the `NumsNode` and `NumsNull` classes are declared to extend the abstract `Nums` class.

PLCC – Syntax (continued)

The RHS entries in a grammar rule are used to declare public fields in the (non-abstract) class associated with the grammar rule. Only those RHS entries enclosed in angle brackets '`<...>`' correspond to fields in the class.

A field can correspond to an RHS token (*e.g.* `<NUM>`) or an RHS nonterminal (*e.g.* `<nums>`).

If the field corresponds to an RHS *token*, its field *name* defaults to the token name with all of its letters in lowercase. For example, the field name corresponding to the RHS token `<NUM>` is `num`, and its field *type* is `Token`. If the field corresponds to an RHS *nonterminal*, its field *name* defaults to the nonterminal name (without change). For example, the field name coressponding to the RHS nonterminal `<nums>` would be `nums`, and its field *type* is the underlying type of the non-terminal – in this case, `Nums` – obtained by converting the first character of the nonterminal name to uppercase.

PLCC – Syntax (continued)

A BNF grammar rule may have the same nonterminal name appearing more than once on its RHS, as in

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree> <tree> RPAREN
```

(see Slide 1.2). PLCC requies that there are no duplicates of class field names associated with the RHS of a given grammar rule, so the above grammar rule would not be acceptable to PLCC. To solve the problem of duplicate field names, we annotate the duplicate field entries by appending an alternate field name (any Java identifier will do, but the convention is to have it start with a lowercase letter) which becomes the name of the corresponding field. The following annotations fix the above example:

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
```

The same requirement that there be no duplicate field entries applies to fields associated with tokens instead of nonterminals. So for a BNF rule such as

```
<hhmmss> ::= <TWOD> COLON <TWOD> COLON <TWOD> NL
```

we would annotate the three <TWOD> token fields with different names:

```
<hhmmss> ::= <TWOD>hh COLON <TWOD>mm COLON <TWOD>ss NL
```

PLCC – Syntax (continued)

Repeating grammar rules – ones that have '`**=`' instead of '`::=`' – result in RHS fields similar to non-repeating grammar rules, except that their RHS fields are `Lists` of the appropriate type, and their field names have the string `List` appended.

The following grammar rule (see Slide 1.36)

```
<nums> **= <NUM>
```

coressponds to the Java class `Nums`, having a single field `numList` of type `List<Token>`.

Similarly, the following grammar rule (we will encounter this later)

```
<letDecls> **= LET <VAR> EQUALS <exp>
```

corresponds to the Java class `LetDecls` having one field `varList` of type `List<Token>` and another field `expList` of type `List<Exp>`.

PLCC – Parsing

PLCC generates a unique Java class for every grammar rule. Each such class has a static `parse` method that is called with a `Scan` object parameter and that returns an instance of the class with the class fields (defined by the grammar rule RHS as described above) populated with appropriate values. For an RHS field corresponding to a token, the field value – a `Token` – comes directly from a lexeme in the input file being parsed. For an RHS field corresponding to a nonterminal, the field value comes from calling the `parse` method on the nonterminal class name.

Similar remarks apply to repetition rules, where the `parse` method uses a loop to populate the `List` fields in the class. The members of the `List` fields appear in the same order that their corresponding syntax entities appear in the input file.

An abstract Java class generated by a nonterminal that appears on the LHS of more than one grammar rules also defines a static `parse` method. This method looks at the current token (delivered by the `Scan` object) and determines which of the RHS grammar rules corresponds to that token. It then returns the value obtained by calling the `parse` method on the derived class corresponding to the selected grammar rule. The result is an instance of the derived class, which is also an instance of the given abstract class.

PLCC – Parsing (continued)

Here's a simple example grammar:

```
<tree>:Leaf     ::= <NUMBER>
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
```

The abstract `Tree` class looks similar to this:

```
public abstract class Tree {

    public static Tree parse(Scan scn$) {
        Token t$ = scn$.cur();
        Token.Match match$ = t$.match;
        switch(match$) {
        case NUM:
            return Leaf.parse(scn$);
        case LPAREN:
            return Interior.parse(scn$);
        default:
            throw new PLCCException(
                "Parse error",
                "Tree cannot begin with " + t$.errString()
            );
        }
    }
}
```

PLCC – Parsing (continued)

PLCC generates the `Interior` class as follows:

```
// <tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
public class Interior extends Tree {

    public Token symbol;
    public Tree left;
    public Tree right;

    public Interior (Token symbol, Tree left, Tree right) {
        this.symbol = symbol;
        this.left = left;
        this.right = right;
    }

    public static Interior parse(Scan scn$) {
        scn$.match(Token.Match.LPAREN);
        Token symbol = scn$.match(Token.Match.SYMBOL);
        Tree left = Tree.parse(scn$);
        Tree right = Tree.parse(scn$);
        scn$.match(Token.Match.RPAREN);
        return new Interior(symbol, left, right);
    }
}
```

PLCC – Parsing (continued)

The `parse` method for a repeating grammar rule uses a loop. For the grammar rule `<nums> **= <NUM>`, PLCC generates the following Java class `Nums`:

```
public class Nums {

    public List<Token> numList;

    public Nums (List<Token> numList) {
        this.numList = numList;
    }

    public static Nums parse(Scan scn$) {
        List<Token> numList = new ArrayList<Token>();
        while (true) {
            Token t$ = scn$.cur();
            Token.Match match$ = t$.match;
            switch(match$) {
            case NUM:
                numList.add(scn$.match(Token.Match.NUM, trace$));
                continue;
            default:
                return new Nums(numList);
            }
        }
    }
}
```

Similar code is generated by repeating rules with a token separator.

PLCC – Parsing (continued)

The code samples shown above omit a `Trace` parameter named `trace$` that plays a role in calling the `parse` methods. If this parameter is not `null` – when the '`-t`' command-line option is given to the `Parse` or `Rep` programs – parsing the program will display a visual "parse trace" to standard error (by default). In Language `LON`, an example parse trace for the program '`(1 3 5)`' looks like this:

```
1: <lon>
1: | LPAREN "("
1: | <nums>:NumsNode
1: | | NUM "1"
1: | | <nums>:NumsNode
1: | | | NUM "3"
1: | | | <nums>:NumsNode
1: | | | | NUM "5"
1: | | | | <nums>:NumsNull
1: | RPAREN ")"
```

Each line in the parse trace displays either a nonterminal or a token. If it displays a nonterminal, this shows that the parser calls the `parse` method for that nonterminal (possibly in the given subclass). If it displays a token, then the scanner also displays the token's lexeme. The decimal number at the beginning of each line is the line number in the source file where the parse found the nonterminal or token; the number of vertical bars indicates the recursive depth of the parse.

PLCC – Semantics

For a given "program" in the language defined by its specification, the `parse` method in the *start symbol class* – the class determined by the first nonterminal in the language BNF grammar rules – returns an instance of this class. This instance is the root of the parse tree for the program. For example, given the BNF rules

```
<tree>:Leaf     ::= <NUMBER>
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree>right RPAREN
```

the `parse` method defined in the `Tree` class returns an instance of a `Tree` object, which is the root of the parse tree. (In what follows, we use the term "parse tree" to refer to the root of the parse tree).

The runtime semantics of any PLCC program is the behavior obtained by calling the `void $run()` method on the parse tree. For any PLCC language, the start symbol class extends a special `_Start` class generated automatically by `PLCC`, so the parse tree is also an instance of `_Start`. Because the `_Start` class defines a `$run()` method whose behavior is to display the `toString` representation of this object to standard output, and because the parse tree is an instance of the `_Start` class, the `$run()` behavior defined on the parse tree defaults to displaying this `toString` representation. Here is the code for `$run()` in the `_Start` class:

```
public void $run() {
    System.out.println(this.toString());
}
```

PLCC – Semantics (continued)

To get a behavior different from this default behavior, the `$run()` method can be redefined in the start symbol class or any of its subclasses. For example, the `$run()` method may be redefined in the `Tree` class so that it displays a more human-readable `toString` representation of the object than the somewhat uninformative default representation. Here is code for the `$run()` method in the `Tree` class that redefines this behavior:

```
public void $run() {
   System.out.println("Tree: " + this.toString());
}
```

Since the `Tree` class appears more than once on the LHS of the grammar rules, each derived class `Leaf` and `Interior` can define its respective `toString` method.

For example, the `Leaf` class has a field `num` of type `Token`. The `toString` method in this class can be easily written as follows:

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

PLCC – Semantics (continued)

For an instance of class `Interior`, an appropriate `toString` method can be:

```
public String toString() {
    return "("+symbol.toString()+" "+left+" "+right+")";
}
```

This relies on the proper (recursive) `toString` behavior of the `left` and `right` fields, both of which are defined as instances of the `Tree` class.

Recall that PLCC generates a Java class for each of the BNF grammar rules given in the syntax section of the language specification. We can define *semantic actions* for these classes by adding entries to the semantics section of the language specification file having the form

```
ClassName
%%%
...
%%%
```

where `ClassName` stands for the name of a PLCC-generated class name, such as `Tree` in Language TREE. PLCC inserts the lines of code bracketed by the '`%%%`' lines verbatim into the `ClassName.java` file. Most often, these lines define one or more Java methods that can be applied to instances of the class.

PLCC – Semantics (continued)

The entire semantics section of the TREE language appears here:

```
Tree
%%%
    public void $run() {
        System.out.println("Tree: " + this.toString());
    }
%%%

Leaf # extends the Tree class
%%%
    public String toString() {
        // 'num' is a Token field in the Leaf class
        return num.toString();
    }
%%%

Interior # extends the Tree class
%%%
    public String toString() {
        // 'left' and 'right' are Tree fields in the Interior class
        return "("+symbol.toString()+" "+left+" "+right+")";
    }
%%%
```

PLCC – Semantics (continued)

As we observed above, PLCC automatically generates a Java source file for each of the classes – both abstract and non-abstract – that are derived from the BNF grammar rules in the syntax section. In the semantics section of the language specification file, if PLCC encounters an entry of the form

```
ClassName
%%%
...
%%%
```

where `ClassName` stands for a class that is *not* one of the automatically generated classes, then PLCC generates a new file `ClassName.java` containing the code bracketed by the '`%%%`' lines. This makes it possible for PLCC to generate Java source files that can be used to augment the semantics of the language. For example, we will use this to implement *environments*, which we describe later.

In this situation, there is no automatically generated source file, so the Java code bracketed by the '`%%%`' lines must be a complete Java source file, not just a method.

PLCC – Semantics (continued)

An `include` feature allows a PLCC language specification file to include the contents of other files, making them part of a single specification. In this way, separately created files can be combined together to form a single language specification. The names of include files must be given in the semantics section of the specification file, and generally appear at the end of the specification. Here is an example from a `grammar` file:

```
...
include code
include env
include prim
include val
```

In this example, the entire contents of the `code` file will be considered as if appended to the `grammar` file, then the `env` file, and so forth. The names of the `include` files will normally be representative of their purposes, but these names do not otherwise play a role in the generated Java files.

PLCC – Java predefined/reserved name conflicts

The Java class names generated by the LHS of rules in the grammar section of a PLCC file must not conflict with standard Java class names. This means that with a grammar rule such as

```
<string> ::= ...
```

PLCC creates a Java class named `String`, which results in a Java compile-time error since '`String`' is a reserved class name.

Similarly, the names of fields in the RHS of rules in the grammar section must not conflict with Java reserved words or predefined identifiers. This means with a grammar rule such as

```
<foo> ::= <IF> <blah>null
```

PLCC creates a field named `if` in the `Foo` class, which results in a Java compile-time error since `if` is a reserved word in Java. PLCC itself does not attempt to identify these errors, so these errors will only show up during compilation.

PLCC – plcc/plccmk command line arguments

The `plcc` script runs the `plcc.py` program with input from the filename arguments given on the command line. The `plccmk` script does the same, where the filename defaults to `grammar`, and also compiles all of the Java files in the `Java` directory. If `plccmk` is given the '`-c`' command line argument, all of the Java files in the `Java` directory are removed before running `plcc.py`.

PLCC – Flags

PLCC has several internal name/value bindings that control its behavior. For example, the value of `LL1` defaults to `True`; if it is changed to `False`, then PLCC will not check the grammar for being LL1. Here are some of the default bindings:

```
name      value  meaning
----      -----  -------
Token     True   (boolean) create a Token.java file
debug     0      (integer) larger values give more verbose output
destdir   'Java' (string)  Java source file destination directory
pattern   True   (boolean) create a scanner that uses re. patterns
LL1       True   (boolean) check for LL(1)
parser    True   (boolean) create a parser
semantics True   (boolean) create semantics routines
nowrite   False  (boolean) when True, produce *no* file output
PP        ''     (string)  cmd to filter (preprocess) generated files
```

Names bound to integer or string values can be changed using command line arguments:

```
... --debug=3 --destdir=JJJ ...
```

or at the top of the specification file on lines beginning with '`!`':

```
!debug=3
!destdir=JJJ
```

PLCC – Flags (continued)

Names bound to a boolean value can be set to True on the command line as in the following example:

```
... --nowrite ...
```

or at the beginning of the language specification file:

```
!nowrite
```

Names bound to a boolean value can be set to False on the command line as in the following example:

```
... --LL1= ...
```

or at the top of the language specification file:

```
!LL1=
```

Bindings given in the language specification file always override bindings given on the command line.

PLCC – Comments in the lexical specification section

In the lexical specification section, PLCC comments appearing in a token or skip specification line are defined as matching the Java regular expression `"\s+#.*"`. Such comments are removed before attemtping to process the remaining characters in the token or skip specification line. This means that a skip or token regular expression containing a substring like `" #"` will mistakenly be considered as the start of a PLCC comment. In this case, use `"[ ]#"` instead.

PLCC – Hooks

PLCC creates Java source files from the BNF grammar rules in a language specification file. For a grammar rule whose LHS is a nonterminal such as `<Formals>` PLCC creates a Java file named `Formals.java`. The initial contents of this file appears as follows:

```
//Formals:top//
//Formals:import//
import java.util.*;

// <formals> **= <VAR> +COMMA // the original BNF rule
public class Formals {

    public static final String $className = "Formals";
    public static final String $ruleString = "<formals> **= <VAR> +COMMA";

    public List<Token> varList; // the RHS field

    public Formals(List<Token> varList) { // the Formals constructor
//Formals:init//
        this.varList = varList;
    }

    public static Formals parse(Scan scn$, Trace trace$) {
        ... code to parse the RHS of the <formals> grammar rule
    }

//Formals//
}
```

PLCC – Hooks (continued)

The following lines in this file are called "hooks":

```
//Formals:top//
//Formals:import//
//Formals:init//
//Formals//
```

In the semantics section of the language specification file, you can arrange to replace these hooks with arbitrary text as needed.

Use the `:init` hook to add additional lines of Java code at the beginning of the class constructor (which is called when parsing a program). Consider an instance of the `Formals` class shown on the previous slide. If you want to check the `varList` field for duplicate identifiers, include the following lines in your `grammar` file (or in your `code` file, if there is an `include code` line in your `grammar` file):

```
Formals:init
%%%
    Env.checkDuplicates(varList, " in proc formals");
%%%
```

This code will then replace the `//Formals:init//` line appearing at the beginning of the `Formals` class constructor. (Since the "hook" appears as a comment in the original Java source code, it will not affect the compiled code if left unchanged.)

PLCC – Hooks (continued)

You can use the `:import` hook to add Java `import` lines to source files for classes that need to import Java packages other than the default `java.util.*`. You can see this used, for example, in Language ABC:

```
Program:import
%%%
import abcdatalog.engine.bottomup.sequential.*;
%%%
```

Similar comments apply to the `:top` hook.

PLCC – Hooks (continued)

You use the final hook in an automatically-generated Java source file to add method definitions (and sometimes field declarations) that will be appended to the class definition. We have already seen how this is used, for example, in Language LON on Slide 1.48, which we repeat here:

```
Lon
%%%
    public void $run() {
        System.out.print("( ");
        for (Token tok: nums.numList)
            System.out.print(tok.toString() + " ");
        System.out.println(")");
    }
%%%
```

Environments

In virtually all programming languages, programmers create symbols (variables) and associate values with them. We discussed bindings earlier. What we want to show now is how to implement bindings. Our implementation allows us to implement *static scope rules*, since this is the most common binding method in programming languages today.

An *environment* is a data structure that associates a value with each element of a finite set of symbols – that is, it represents a set of bindings. We could think of an environment as a set of pairs

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

that encode the binding of symbol s_1 to value v_1, s_2 to value v_2, *etc.* The problem with this simple approach is that the same symbol may have different bindings in different parts of the program, and this approach doesn't make it clear how to determine which binding is the *current* binding.

Instead, we specify an environment as a Java object having a method called `applyEnv` that, when passed a symbol (a `String`) as a parameter, returns the current value bound to that symbol. So if `env` is an environment and `x` is a symbol,

```
env.applyEnv("x")
```

returns the value currently bound to the symbol `x`.

Environments (continued)

If there is no binding for the symbol `x` in the environment `env`, then `env.applyEnv("x")` throws an exception.

In addition to getting the current value bound to a symbol, our environment implementation provides a way to create an empty environment (one with no bindings), and a way to extend an existing environment (essentially to enter a new *block*) by adding new bindings.

But wait: what type does `applyEnv` return? In other words, what exactly is a "value"? For the time being, we assume that a "value" is an instance of a class aptly named `Val`. We will refine the notion of "value" later. If you're worried about this, just pretend that instances of the `Val` class represent integers.

Environments (continued)

Our environments are implemented as instances of a Java abstract class `Env`, a simple version of which we show here:

```
public abstract class Env {

    public static final Env empty = new EnvNull();

    public abstract Env add(Binding b);

    public static Env initEnv() {
        // initial environment with no bindings
        return new EnvNode(new Bindings(), empty);
    }

    public abstract Binding lookup(String sym); // only local bindings

    public abstract Val applyEnv(String sym);

    public Val applyEnv(Token tok) {
        return applyEnv(tok.toString());
    }

    public Env extendEnv(Bindings bindings) {
        return new EnvNode(bindings, this);
    }

}
```

Environments (continued)

We represent a binding as an instance of the class `Binding`. A `Binding` object has a `String` field named `id` that holds an identifier name (a symbol) and a `Val` field named `val` that holds the value bound to that variable.

```
public class Binding {

    public String id;
    publilc Val val;

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

}
```

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

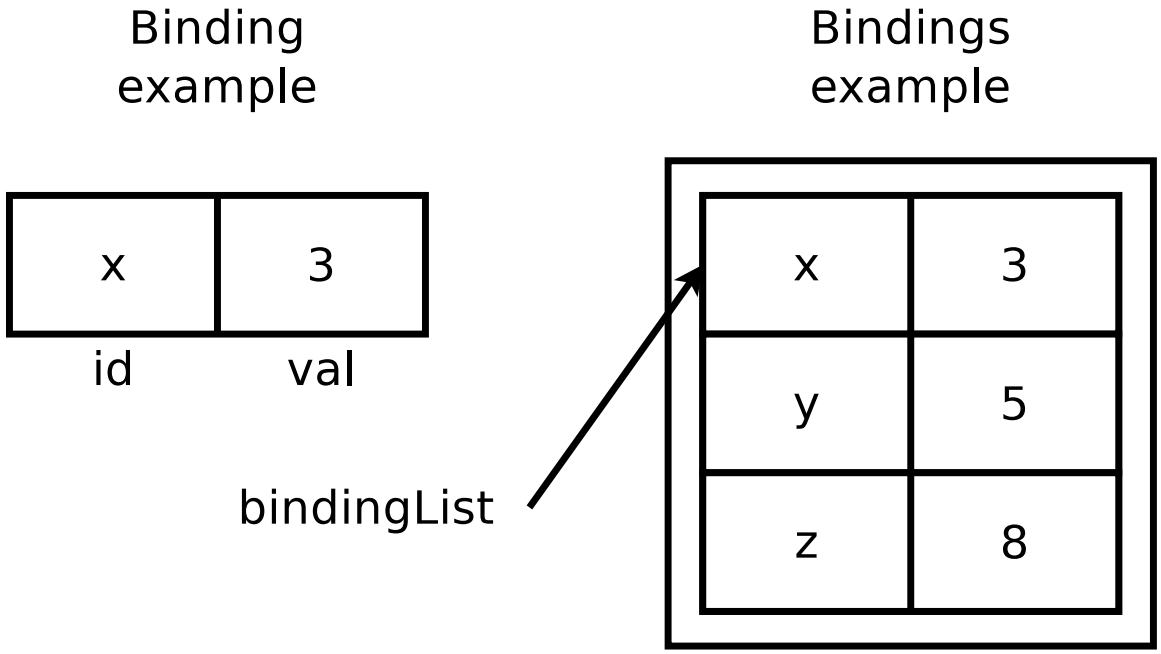
Environments (continued)

A *local environment* is a list of zero or more bindings. In the context of block-structured languages, you can think of a local environment as capturing all of the bindings defined in a particular block. We represent a local environment using the class `Bindings`. A `Bindings` object has a single field named `bindingList` which is a `List` of `Binding` objects.

```
public class Bindings {

    public List<Binding> bindingList;

    // create an empty list of bindings
    public Bindings() {
        bindingList = new ArrayList<Binding>();
    }

    public Bindings(List<Binding> bindingList) {
        this.bindingList = bindingList;
    }

    ... methods lookup() and add() are given later ...

}
```

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

The following diagram gives an example of a `Binding` object that binds the symbol `"x"` to the integer (`IntVal`) value `3`, and a `Bindings` object with a `bindingList` of size three.



For the sake of simplicity, we omit drawing the extra box around the `bindingList` in future `Bindings` diagrams.

Environments (continued)

Given a local environment, we want to be able to look up the `Binding` associated with a particular symbol and to add `Binding` objects to the local environment. It is also convenient to consider adding a `Binding` obtaind from a symbol and a value. The following methods are part of the `Bindings` (local environment) class:

```
// look up the Binding associated with a given symbol
public Binding lookup(String sym) {
    for (Binding b: bindingList)
        if (sym.equals(b.id))
            return b;
    return null;
}

// add a Binding object to this local environment
public void add(Binding b) {
    bindingList.add(b);
}

// add a binding (s, v) to this local environment
public void add(String s, Val v) {
    add(new Binding(s, v));
}
```

Environments (continued)

A nonempty environment is an instance of the class `EnvNode`. An `EnvNode` object has two fields: a `Bindings` object named `bindings` that holds the local bindings and an `Env` object named `env` that refers to an enclosing (in the sense of static scope rules) environment.

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

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

- if that returns a non-`null` `Binding` object, `applyEnv` returns the `Val` field in the `Binding`.
- if that returns `null`, `applyEnv` returns the result of calling `applyEnv` (recursively) on the enclosing environment.

The `EnvNode` class is given on the following slide.

Environments (continued)


```
public class EnvNode extends Env {

    public Bindings bindings; // list of local bindings
    public Env env;           // enclosing scope

    public EnvNode(Bindings bindings, Env env) {
        this.bindings = bindings;
        this.env = env;
    }

    public Binding lookup(Symbol sym) {
        return bindings.lookup(sym);
    }

    public Val applyEnv(String sym) {
        Binding b = bindings.lookup(sym);
        if (b == null)
            return env.applyEnv(sym);
        return b.val;
    }

}
```

Environments (continued)

An empty environment is an instance of the class `EnvNull`. An `EnvNull` object has no fields. The `lookup` method in the `EnvNull` class returns `null` (no `Binding` can be found in the empty environment), and the `applyEnv` method in this class throws a `PLCCException`.

```
public class EnvNull extends Env {

    // create an empty environment
    public EnvNull() {
    }

    // find the Val corresponding to a given id
    public Val applyEnv(String sym) {
        throw new PLCCException("no binding for "+sym);
    }

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

}
```

Environments (continued)

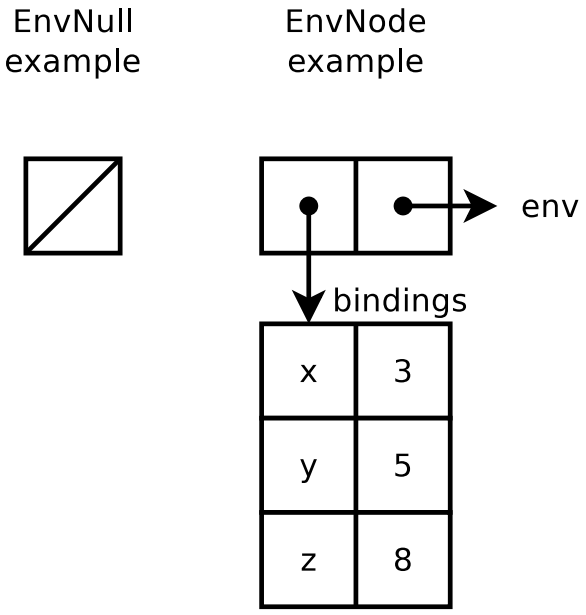
Both the `EnvNode` and `EnvNull` classes define an `add` method – declared as an abstract method in the `Env` class – that takes a `Binding` object as a parameter. In the `EnvNode` class, this method adds the binding to its local environment. In the `EnvNull` class, this method throws an exception (because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

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

Environments (continued)

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

The `EnvNull` object has no fields – we always draw such an environment as shown below. The `EnvNode` object in this example has a `bindings` field as shown on slide 2.6 and an `env` field referring to some enclosing environment (not shown).



In the right-hand `EnvNode` example, a call to `lookup("x")` returns the `Binding` object `("x", 3)` (think of this as a pair), a call to `applyEnv("x")` returns an `IntVal` object with value 3, and a call to `lookup("w")` returns `null`.

Environments (continued)

In summary, an environment is a linked list of nodes, where a node is either an instance of `EnvNode` (with its corresponding local bindings) or an instance of `EnvNull`, which terminates the list.

The `extendEnv` procedure, defined in the `Env` class, takes a set of local bindings and uses them to return a new `EnvNode` that becomes the head of a new environment list, extending the current environment list. Here is the definition of `extendEnv`:

```
public Env extendEnv(Bindings bindings) {
    return new EnvNode(bindings, this);
}
```

Environments (continued)

In some cases, we may want to create a `Bindings` object from a `List` of symbols and a `List` of values by binding each symbol to its corresponding value. The resulting `Bindings` object can then be used to extend an environment. Here is a constructor for a `Bindings` object that does this pairing.

```
public Bindings (List<?> idList, List<Val> valList) {
    // idList.size() and valList.size() must be the same
    bindingList = new ArrayList<Binding>();
    Iterator<?> idIterator = idList.iterator();
    Iterator<Val> valIterator = valList.iterator();
    while (idIterator.hasNext()) {
        String s = idIterator.next().toString();
        Val v = valIterator.next();
        this.add(new Binding(s, v));
    }
}
```

The purpose of the '`?`' wildcard in the `List<?>` parameter declaration is to allow for a `List` of either `Strings` or `Tokens`.

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

Environments (continued)

For the remainder of these materials, we use the representation for environments that we have described here:

- an environment is a (possibly empty) linked list of local environments
- a local environment is a `List` of bindings
- a binding is an association of an identifier (symbol) to a value

Our implementation defines the following classes: `Env` (with subclasses `EnvNull` and `EnvNode`), `Binding`, and `Bindings` as summarized on the next slide.

Environments (continued)

Class/method summary:

```
abstract class Env
    public abstract Binding lookup(String sym)
    public Val applyEnv(String sym)
    public Env extendEnv(Bindings bindings)
    public Env add(Binding b)

class EnvNode extends Env
    public Binding lookup(String sym)

class EnvNull extends Env

class Bindings
    public Binding lookup(String sym)
    public void add(Binding b)

class Binding

abstract class Val

class IntVal extends Val
```

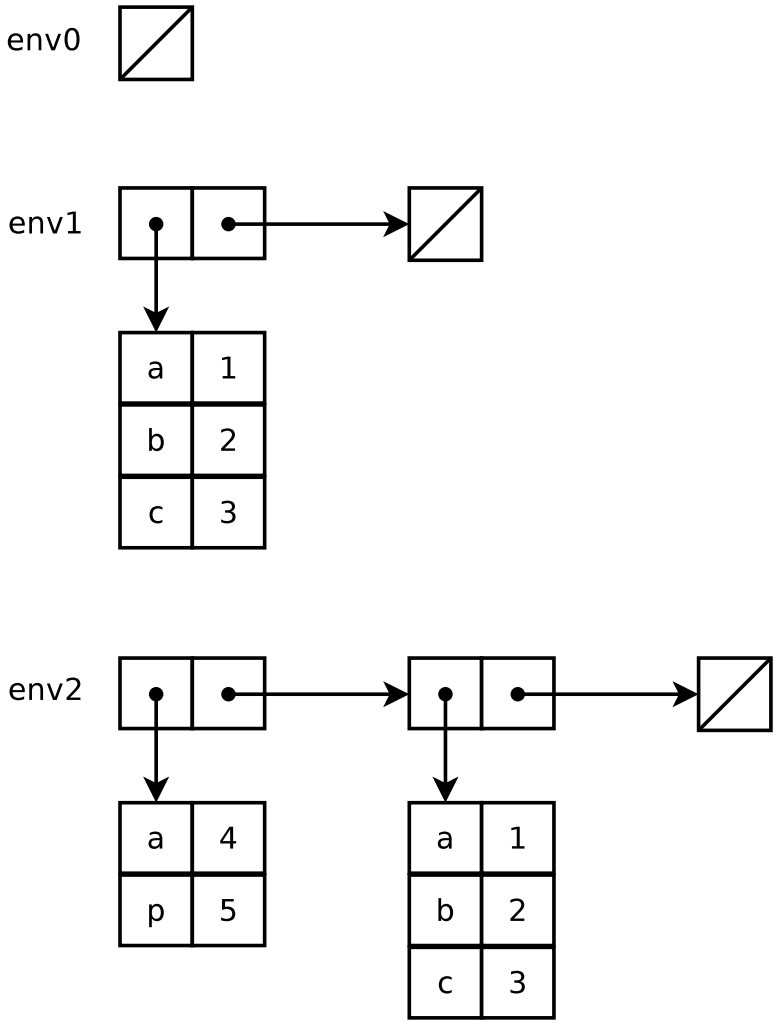
Environments (continued)

Here is a test program in the `Env` class. This program illustrates how to use the two constructor versions in the `Bindings` class.

```
public static void main(String [] args) {
    Env env0 = empty;
    Env env1 = env0.extendEnv(
        new Bindings(Arrays.asList(
            new Binding("a", new IntVal(1)),
            new Binding("b", new IntVal(2)),
            new Binding("c", new IntVal(3)))));
    List<String> i2 = Arrays.asList("a", "p");
    List<Val> v2 = Arrays.asList((Val)new IntVal(4), (Val)new IntVal(5));
    Env env2 = env1.extendEnv(new Bindings(i2, v2));
    System.out.println("env0:\n" + env0.toString(0));
    System.out.println("env1:\n" + env1.toString(0));
    System.out.println("env2:\n" + env2.toString(0));
    System.out.print("a(env2) => "); System.out.println(env2.applyEnv("a"));
    System.out.print("a(env1) => "); System.out.println(env1.applyEnv("a"));
    System.out.print("p(env2) => "); System.out.println(env2.applyEnv("p"));
    System.out.print("p(env1) => "); System.out.println(env1.applyEnv("p"));
}
```

Environments (continued)

We show these environments in diagram form as follows:



Environments (continued)

We can include Java code for environments into our PLCC specification file (usually called `grammar`) in the same way that we include Java code into our Java classes generated by PLCC. However, the environment-related classes `Env`, `EnvNode`, `EnvNull`, `Binding`, and `Bindings` do not appear in our BNF grammar, so PLCC doesn't generate stubs for them automatically.

As noted in Slide Set 1a, PLCC makes it possible to create stand-alone Java source files in exactly the same way as it adds methods to generated files, except that the *entire* code for each stand-alone Java source file – including `import` lines and the class constructor – must appear in the language specification section following the BNF rules.

For example, to create a Java source file named `Env.java`, use the following template:

```
Env
%%%
... code for the entire Env.java source file ...
%%%
```

Environments (continued)

We will encounter language definitions that end up creating dozens of Java source files. To manage these complex languages, we separate the contents of the semantics section of the grammar file into separate files grouped by purpose. In many cases, different languages may share some of the same source files. The semantics section then simply identifies the names of these files using an `include` feature that treats the contents of these files as if they were part of the entire grammar file.

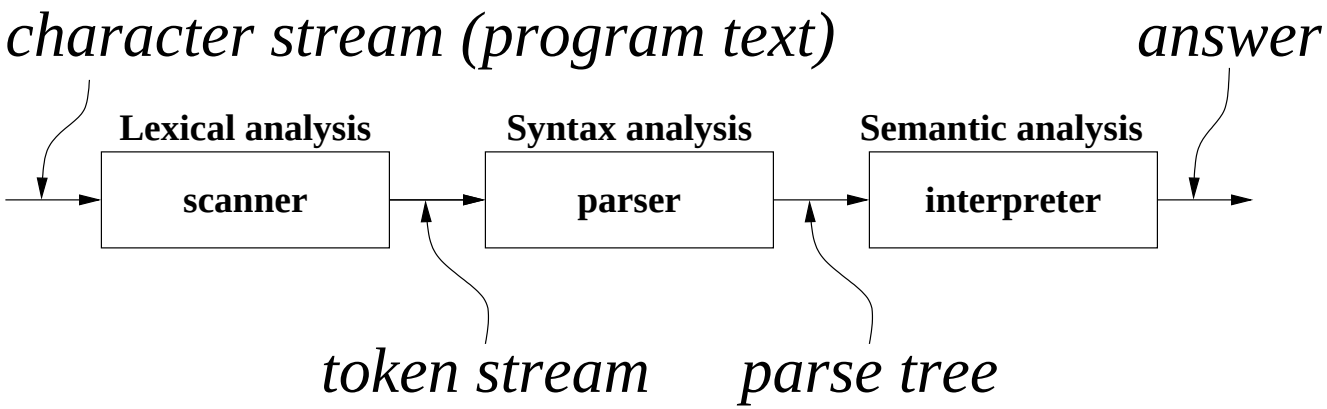
For example, one of our early languages `V1` has the following grammar file structure:

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

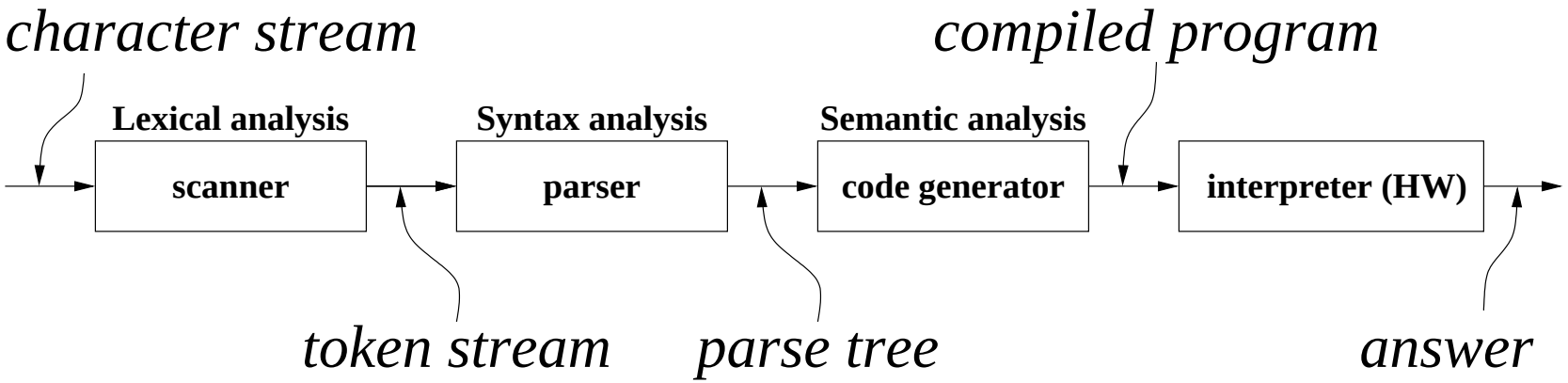
Environment-Passing Interpreters

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 this class – we write arithmetic expressions 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 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.

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 section**, the **syntax section**, and the **semantics (code) section**.

(Recall that if your `grammar` file has only the lexical specification, the `plccmk` tool produces Java code for a scanner (`Scan`), but nothing else. If your `grammar` file has only the lexical specification and syntax rules, the `plccmk` tool produces Java code for a scanner (`Scan`) and a parser (`Parse`) 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 by defining their behavior – specifically, by defining the `$run()` method in the start symbol class. We will presently see how such a `$run()` 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 of Language V0 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 – again, a `String` representation of the parse tree. 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))
...
```

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.

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 `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 3.7) defines a class given by its LHS with a well-defined set of instance variables corresponding to the (angle bracket) entries in its RHS.

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 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 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 we parse a program such as

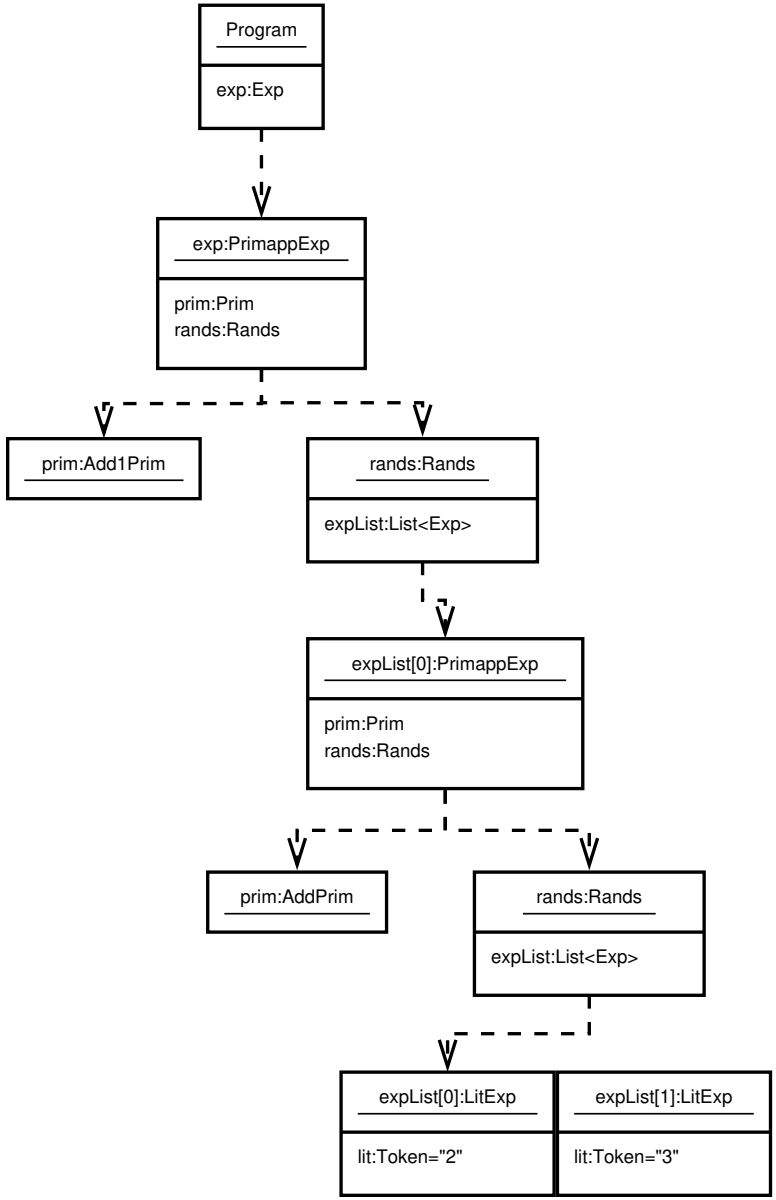
```
add1( +(2,3))
```

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.) As we have seen, this displays a string that looks like this:

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

So our next step is to show how we can redefine the `$run()` method in the `Program` class so that overrides the default behavior by printing 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 modify 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 methods to a class such as `Program`, use the following template:

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

Since these method definitions 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 `$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 four `<prim>` BNF rules specified in Language V1.

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());
    }
%%%
```

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.

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;
    }
%%%
```

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 `Vals`. In order to access these values easily and to apply normal arithmetic operations to them, we convert them into an *array* of `Val` objects. The utility method named `toArray` in the `Val` class accomplishes this.

The expressions appearing in an application of a primitive are called its ***operands****, 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`), 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.

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 now 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.

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 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.

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 and attempt to divide by zero.

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.

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()`. These bindings give us some variables to play with, though, as you will see, we will dispense with them later.

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 in the `Java` subdirectory and enter expressions at the prompt:

```
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`'.]

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 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 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 /(5,0) 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:

1. 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;
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.

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*!

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
```

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. What this means is 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`.

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!).

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 3.59, 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 `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 `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).

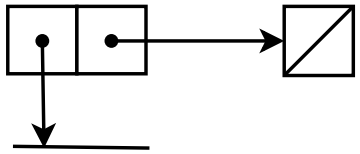
```
public Val apply(List<Val> args, Env e) {
    // bind the formals to the arguments (step 2a)
    Bindings bindings = new Bindings(formals.varList, args);
    // extend the captured environment with these bindings (step 2b)
    Env nenv = env.extendEnv(bindings);
    // and evaluate the body in this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

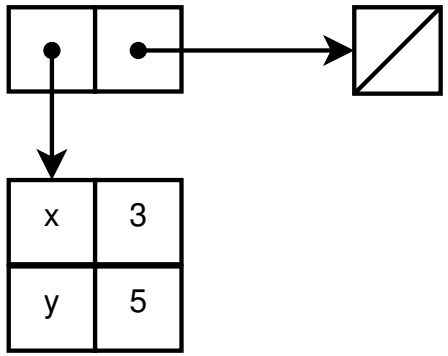
- evaluating a `let` expression
- evaluating a procedure application

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:

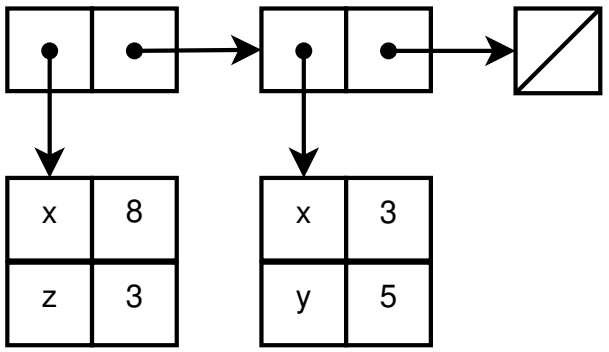


Drawing Envinronments (continued)

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)

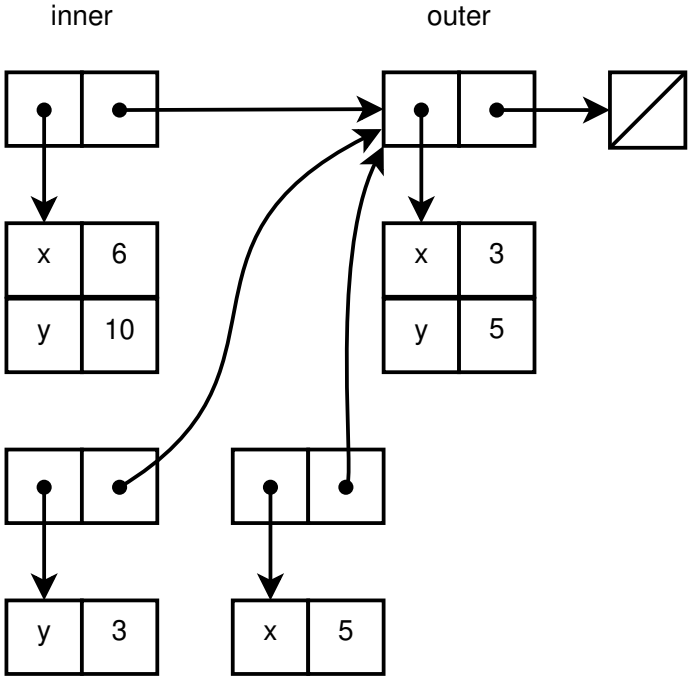
Since a `let` expression is an expression, it must evaluate to a value, so a `let` expression can occur as the RHS of a binding in another `let` expression. Consider this example:

```
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
```

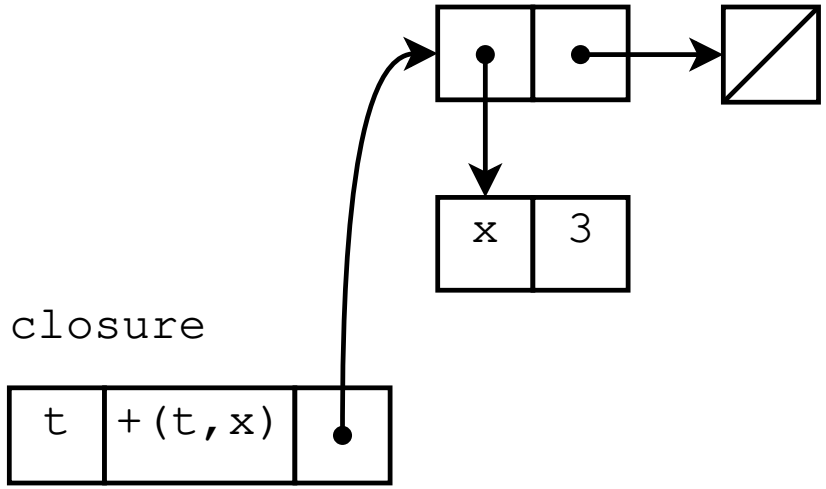
Here, the *captured environonment* is the environment in which the procedure is defined. For example, consider the following expression:

```
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)
```



Drawing Envinronments (continued)

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:

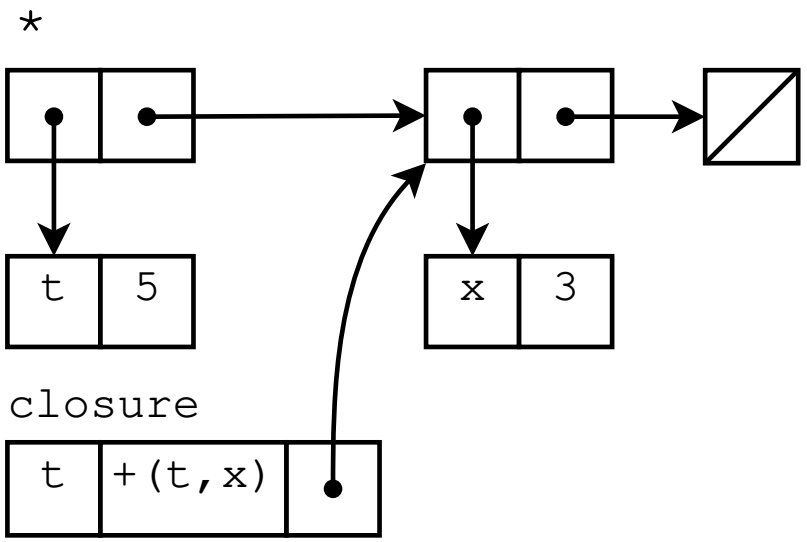
```
let
  x = 3
in
  .proc(t) +(t,x) (5)
```

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 '`*`'.

Drawing Envinronments (continued)


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



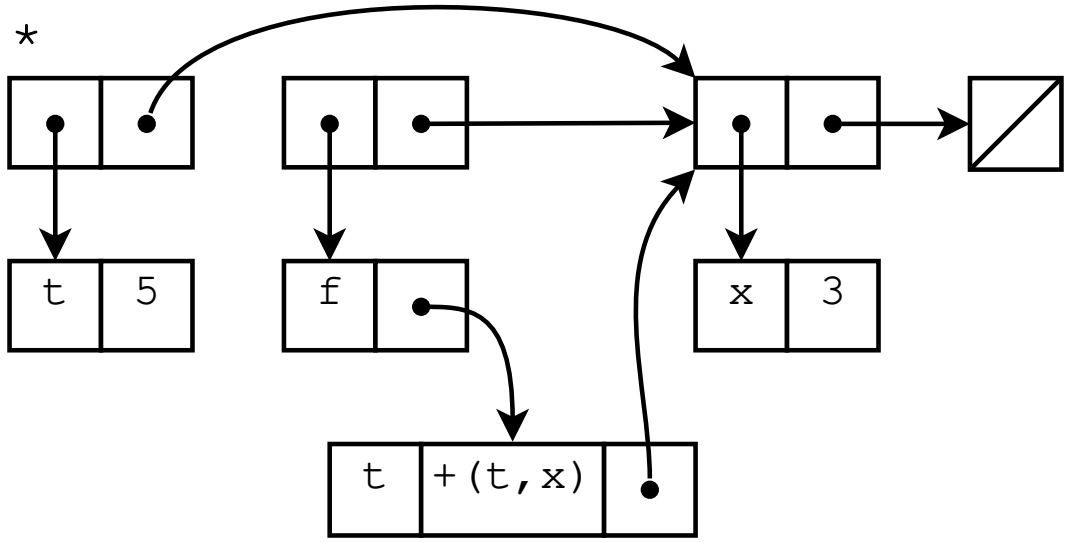
Finally, consider this example:

```
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)
```

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, ...)
```

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, 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 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;
    }
%%%
```

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` expression. 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`, 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.

This is what we want:

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

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. Call this new environment `nenv`.

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 `nenv` obtained in Step 0, and save its value in a variable `val`. Then create a `new Binding(str, val)` and add it to the `nenv` environment obtained in Step 0. This binding now becomes part of the local bindings in `nenv`, together with the other local bindings previously added during this iteration.

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

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

```
LetDecls
%%%
    public Env addLetrecBindings(Env env) {
        // Step 0
        Env nenv = 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(nenv);
            nenv.add(new Binding(str, val));
        }
        return nenv; // 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`.

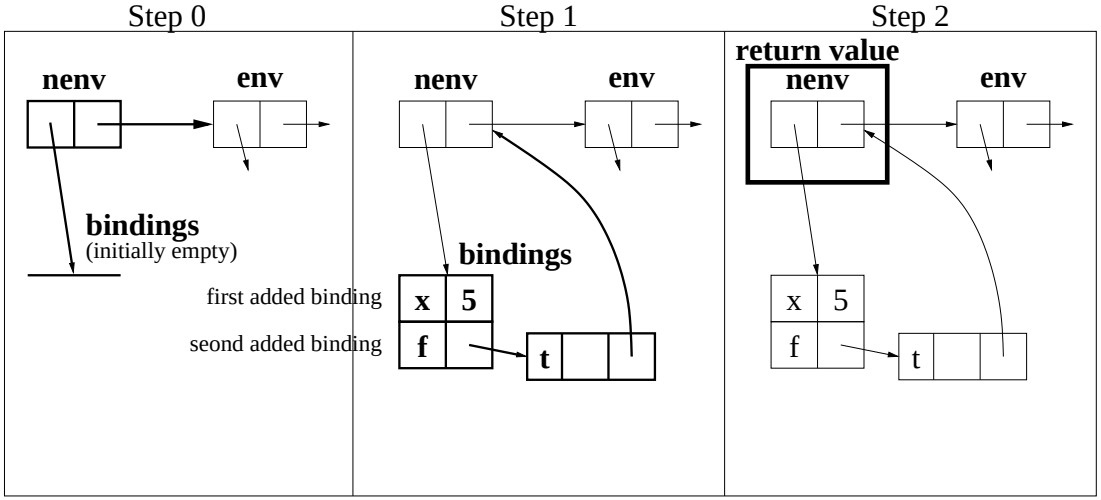
Language V5 (continued)

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 a new environment `nenv` by extending the old environment with an empty list of bindings;
1. Create a `Binding` of each LHS identifier (`x` and `f` in this example) to the value of its corresponding RHS expression (5 and `proc(t)...` in this example) – where each expression is evaluated in the environment `nenv` – and add this binding to `nenv`.
2. Once all of the (LHS, RHS) bindings have been added to `nenv`, return `nenv` 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.



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 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 (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)
```

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 `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.

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.

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
```

Language SET

In this version of our source language, we allow for the assignment of values to variables. Languages that allow for the mutation of variables are called *side-effecting*; such languages are inherently more difficult to reason about, which accounts for why functional programming has received so much attention and also for why it is so difficult to produce high-quality software in most side-effecting programming languages.

So far, Languages V1 through V6 have treated denoted values (the things that variables are bound to) as being the same as expressed values (the values that expressions can have). For example, a variable `x` in one of these languages always evaluates to the same thing no matter where it appears in its scope.

When we add variable mutation (also called "assignment"), such as with

```
set x = add1(x)
```

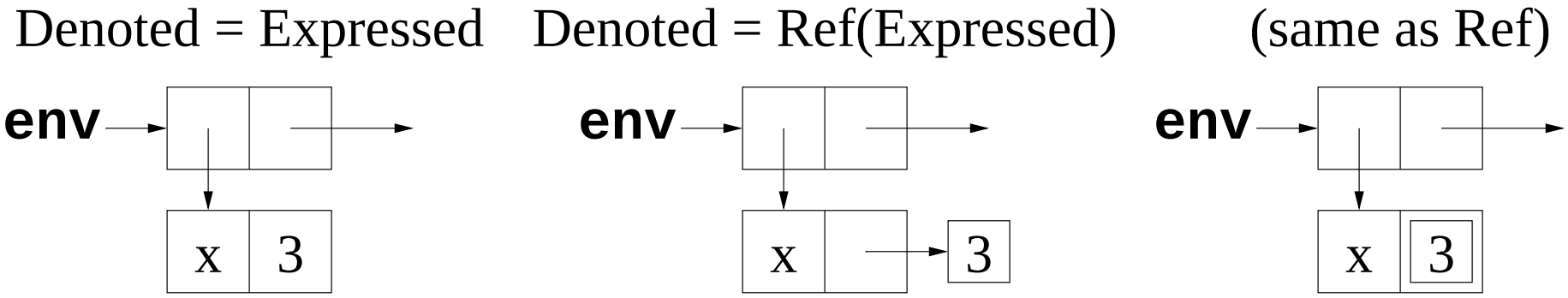
the meaning of `x` on the LHS is different from its meaning on the RHS. The expression `x` in the RHS of this "assignment" represents an expressed value, whereas `x` on the LHS represents a denoted value that can be modified. In order to implement variable assignment, we need to find a way to disconnect denoted values from expressed values.

Language SET (continued)

We introduce the notion of a *reference*, something that *refers to* a mutable location in memory. Instead of binding a variable directly to an expressed value, we bind the variable to a reference which itself contains an expressed value.

Expressed value = Val = IntVal+ProcVal
Denoted value = Ref(Expressed)

To mutate a variable bound to a reference, we change the *contents* of the reference; the variable is still bound to the same reference.



The two right-hand diagrams depict the same environment. The rightmost one uses a more compact representation.

Language SET (continued)

References will also be used to implement various parameter-passing mechanisms as described later in these notes.

We choose the following concrete and abstract syntax for variable mutation:

```
<exp>:SetExp ::= SET <VAR> EQUALS <exp>
                 SetExp(Token var, Exp exp)
```

We can now write the following program in our newly extended source language:

```
let
  x = 42
in
  { set x = add1(x) ; x }
```

This evaluates to 43.

Language SET (continued)

The ability to modify the value bound to a variable allows us to "capture" an environment in a procedure and use the procedure to modify its captured environment. For example, consider:

```
define g = let
             count = 0
           in
             proc() set count = add1(count)
.g() % => 1
.g() % => 2
.g() % => 3
```

The value of `count` is captured in the local `let` bindings that defines the `proc()`. Each time we evaluate `.g()`, the procedure increments the value of `count` and returns this newly incremented value. The variable `count` persists from one invocation to the other because the `proc` captures the environment in which it is defined, namely the one with the variable `count`.

In this example, the `count` variable is unbound in the top-level environment, so an attempt to evaluate it throws an exception:

```
count % unbound variable
```

For our purposes, we want a reference to be a Java object whose contents can be mutated. When we bind a variable to a reference (its denoted value), this binding does not change, but the contents of the reference itself – the thing it refers to – can change.

The `Ref` abstract class embodies our notion of a reference – the thing that a variable can be bound to. For now, the only subclass of the `Ref` class is the `ValRef` class.

```
ValRef(Val val)
```

The contents of a `ValRef` object is a `Val`, and we say that such an object is a *reference to a value*. (Recall that a `Val` object is either an `IntVal` or a `ProcVal` – the only two `Val` types that we currently have.)

A `Ref` object has two methods:

```
public abstract Val deRef();
public abstract Val setRef(Val v);
```

In the `ValRef` class, The `deRef` (dereference) method simply returns the `Val` object stored in the object's `val` field, and the `setRef` (set reference) method modifies the `val` field by changing it to the `Val` parameter `v` (and returning the new `Val` object as well).

Language SET (continued)


```
Ref
%%%
public abstract class Ref {

    public abstract Val deRef();
    public abstract Val setRef(Val v);

}
%%%
```

Language SET (continued)


```
ValRef
%%%
public class ValRef extends Ref {

    public Val val;

    public ValRef(Val val) {
        this.val = val;
    }

    public Val deRef() {
        return val;
    }

    public Val setRef(Val v) {
        return val = v;
    }
}
%%%
```

Language SET (continued)

Our denoted values (the things that variables are bound to) are now references instead of values, so we need to change our `Binding` objects to bind an identifier to a reference. (Notice that we use the terms "variable", "identifier", and "symbol" interchangeably.)

```
Binding(String id, Ref ref)
```

In the `Env` class, we want `applyEnv` to continue to return a `Val` object, whereas the bindings now associate identifiers with references, so we split up the responsibilities as follows:

```
// returns the reference bound to sym
public abstract Ref applyEnvRef(String sym);

public Val applyEnv(String sym) {
    return applyEnvRef(sym).deRef();
}
```

The `applyEnvRef` method behaves exactly like the previous `applyEnv` method (but returns a `Ref` instead) and throws an exception if there is no reference bound to the given symbol. The `applyEnv` method simply gets the `Ref` object using `applyEnvRef` and dereferences it to return the corresponding value.

Language SET (continued)

In our semantics code, we need to modify all of the instances of `Binding` or `Bindings` objects so that they use references instead of values. To create a "binding" of a variable to a value, first wrap the value into a new reference and then bind the variable to the newly created reference. Here's an example of how to create a binding of the variable named `x` to (a reference to) an integer `10`:

```
String var = "x";
Val val = new IntVal(10);
Binding b = new Binding(var, new ValRef(val));
```

The `valsToRefs` static method in the `Ref` class takes a list of `Vals` and returns a corresponding list of `Refs`. This is used, for example, in the code for `AppExp` objects (which need to bind formal parameter symbols to references to their actual parameter values) and for `LetExp` objects (which need to bind their LHS variable symbols to reference to their RHS expression values).

```
public static List<Ref> valsToRefs(List<Val> valList) {
    List<Ref> refList = new ArrayList<Ref>(valList.size());
    for (Val v : valList)
        refList.add(new ValRef(v));
    return refList;
}
```

Language SET (continued)


```
<exp>:SetExp ::= SET <VAR> EQUALS <exp>
                 SetExp(Token var, Exp exp)
```

So far, we have dealt only with the implementation details of environments. How do we implement the semantics of `set` expressions? Coding this is now simple:

```
SetExp
%%%
    public Val eval(Env env) {
        Val val = exp.eval(env); // the RHS expression value
        Ref ref = env.applyEnvRef(var); // the LHS reference
        return ref.setRef(val);  // sets the ref and returns val
    }
%%%
```

Notice that a `set` expression evaluates to the value of the RHS of the assignment. This means that multiple `set` operations can appear in one expression.

Language SET (continued)

For example, the following expression evaluates to 12:

```
let
  t = 3
  u = 42
  v = 0
in
  { set v = set u = set t = add1(t) ; +(t,+(u,v)) }
```

The first expression in the body of this `let` gets evaluated like this:

```
    set v = { set u = { set t = add1(t) } }
```

What happens if you try to mutate the value of an identifier that is one of the formal parameters to a procedure? For example, what value is returned by the following program?

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

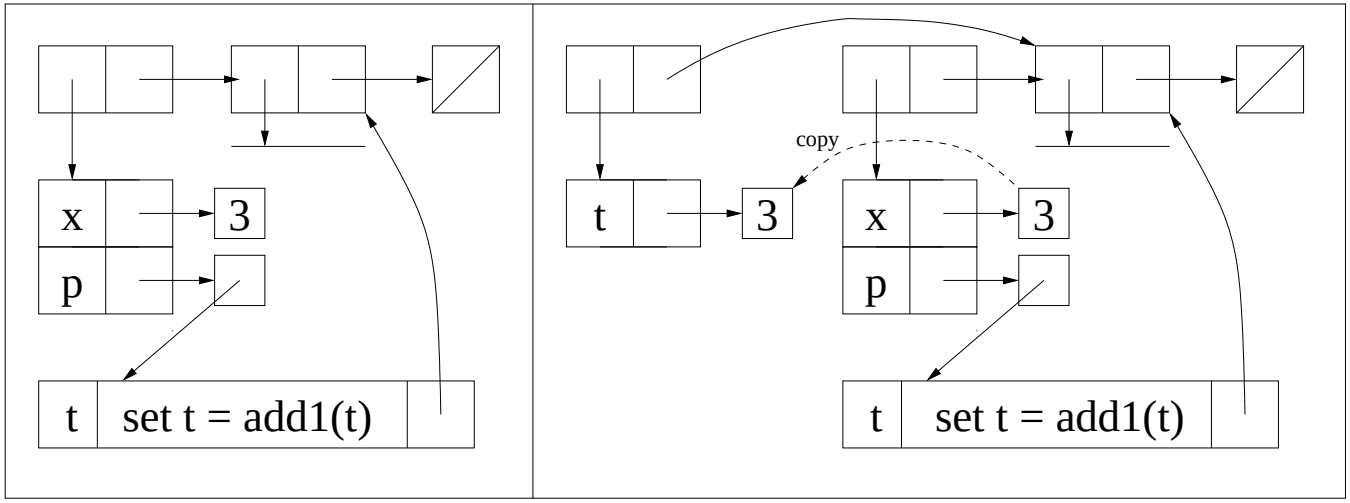
In our procedure application semantics (see the `AppExp` code), the formal parameters are bound to (references to) the *values* of the actual parameters. Since the value of the actual parameter `x` in the expression `.p(x)` is 3, this means that the variable `t` in the body of the procedure is bound to (a reference to) the value 3, and evaluating the body of the procedure modifies this binding to the value 4, but it's the variable `t`, not the variable `x`, that gets modified. Thus the value of this entire expression is 3.

Language SET (continued)

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

The following illustration shows

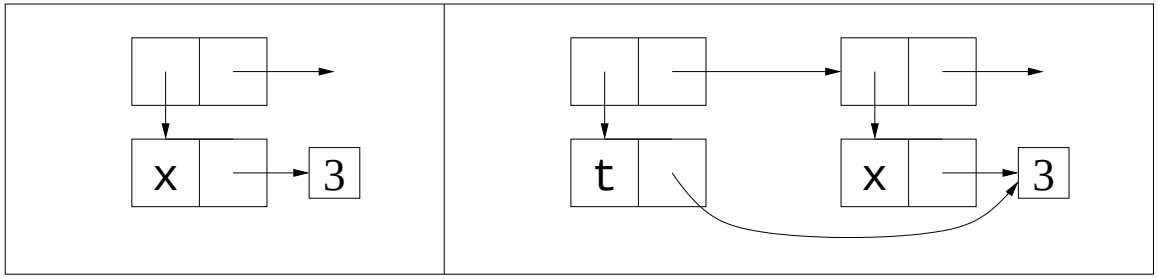
- the environment immediately before the procedure application `.p(x)` – in particular, the binding of `x` to a reference to the value 3,
- and the environment during the procedure application `.p(x)`, binding the formal parameter `t` to a *new* reference containing a copy of the value of `x` (as shown by the dashed line).



Language REF

A parameter passing semantics that evaluates actual parameters and that binds the formal parameters to these actual parameter values is called *call-by-value*. This is what is used in languages `V1` to `V6`. In the language `SET`, where bindings are to references instead of values, the actual parameter values are turned into *new* references, and these references are bound to the formal parameters.

Considering the illustration in the previous slide, Suppose we *want* a behavior that binds the formal parameter `t` to the *same* reference that is bound to `x` instead of a new reference containing a copy. The following diagram shows how the bindings in the previous diagram change when `t` is bound to the same reference as verb'x':



Such a parameter passing semantics is called *call-by-reference*. We explore call-by-reference next, along with variants on this theme.

Language REF (continued)

To repeat:

- The parameter passing semantics that we have been using up to now is called *call-by-value*. In call-by-value semantics – also referred to as simply *value semantics*, when an actual parameter expression in a procedure application is a variable, the procedure's corresponding formal parameter denotes a new reference to the expressed value of the actual parameter.
- In *call-by-reference* semantics – also referred to as simply *reference semantics*, when an actual parameter expression in a procedure application is a variable, the procedure's corresponding formal parameter denotes the *same reference* as the actual parameter.

The differences between value and reference semantics only apply when the actual parameter expression is a variable. When the actual parameter expression is not a variable, the corresponding formal parameter always denotes a new reference to the expressed value of the actual parameter.

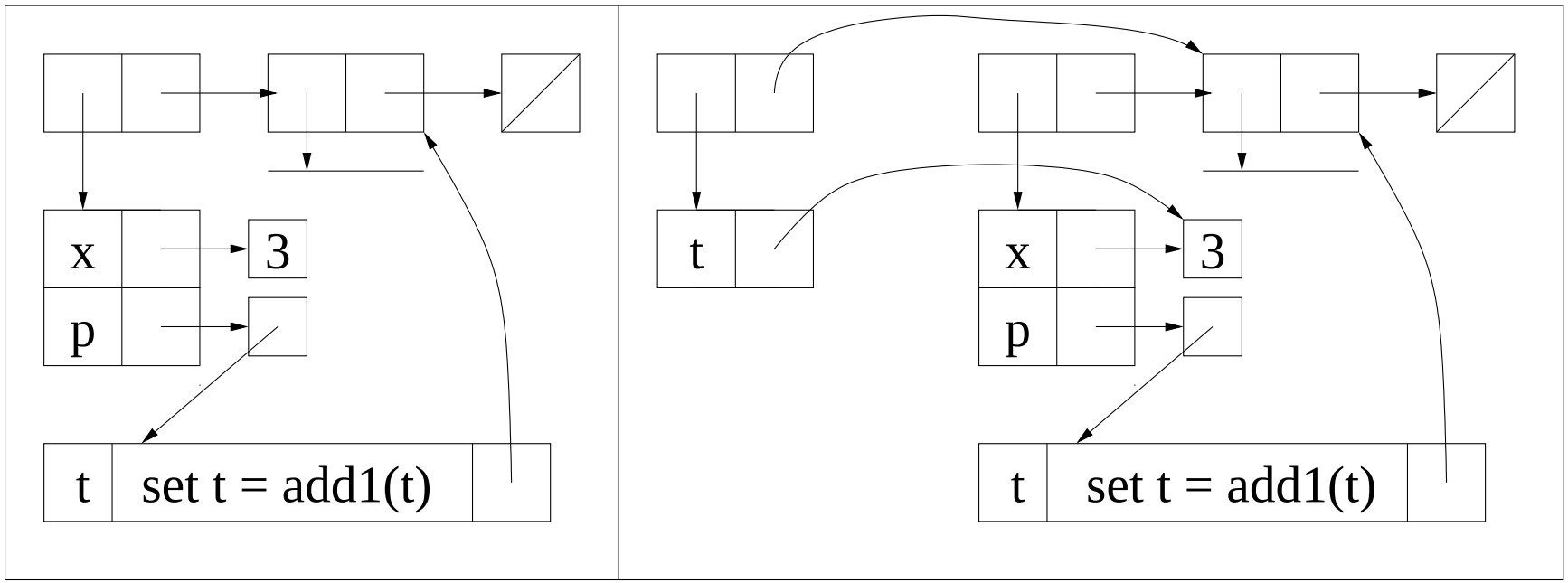
Observe that in `let` and `letrec` expressions, we always use value semantics for the variable bindings. This means that each LHS variable in a `let/letrec` expression always denotes a new reference to the expressed value of its corresponding RHS expression.

Language REF (continued)

Using call-by-reference semantics, the program

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

returns the value 4, since `t` denotes the same reference as `x`. The bindings created by the `let` and then during evaluation of the application `.p(x)` (just prior to evaluating the procedure body) are illustrated in the following figure:



When actual parameter expressions are not themselves variables, we use value semantics. To illustrate this, consider the value returned by the following program:

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(+(x,0)) ; x }
```

Clearly the expressed value of the actual parameter `+(x,0)` is the same as that of `x`, but the expression `+(x,0)` is not a variable, so value semantics apply to this actual parameter. This means that when we apply the procedure `p`, the formal parameter `t` denotes a *new* reference to the value of this expression: the variables `t` and `x` have the same expressed values, but they have different denoted values, so modifying `t` does not affect the value of `x`. This expression evaluates to 3.

Language REF (continued)

The term *L-value* refers to an expression that can be interpreted as a reference. (It's called an L-value because it is the sort of expression that can appear to the *left* of the = in a `set`.) While a variable `x` can always be considered as an L-value, the expression `+(x,0)` can only be interpreted as a value, never a reference. In Language REF, only variable expressions are L-values.

In summary, **if an actual parameter is a L-value in Language REF, (and therefore can be interpreted as a reference), then the corresponding formal parameter is bound to the same reference. If an actual parameter is something other than an L-value, then the corresponding formal parameter is bound to a *new* temporary reference containing the value of the actual parameter.**

Language REF (continued)

Our `REF` language has exactly the same grammar rules as our `SET` language. The *only* differences are in the bindings of formal parameters during procedure application. As the discussion on the previous slides show, we need to handle actual parameters that are variables differently from actual paramemeters that are expressions. The idea here is to define an `evalRef` method for instances of the `Exp` classes that takes care of how to translate themselves into a reference: for anything but a `VarExp`, `evalRef` evaluates the expression and returns a new reference to the value. For a `VarExp`, `evalRef` returns the same reference that the actual parameter denotes.

So in the `Exp` class, the `evalRef` method has the following *default* behavior:

```
public Ref evalRef(Env env) {
    return new ValRef(eval(env));
}
```

For the `VarExp` subclass – *and only for this class*, `evalRef` is implemented as:

```
public Ref evalRef(Env env) {
    return env.applyEnvRef(var);
}
```

The `evalRef` method in the `VarExp` class overrides the `evalRef` method in the `Exp` class. In all other classes that extend the `Exp` class, the default definition in the parent `Exp` class is used.

Language REF (continued)

The other change is in the `Rands` code. In the `SET` language, the `evalRands` method was used in the implementation of `eval` for both a `LetExp` object and an `AppExp` object, since both created new bindings to values. In the `REF` langauge, an `AppExp` object needs new bindings to values except for actual parameters which are variables – a situation that is described in the previous slide. Therefore, to implement the correct `eval` semantics for an `AppExp` object, we need to collect `evalRef` references instead of `eval` values to bind them to the formal parameters. The method `evalRandsRef` in the `Rands` class does this work for us. The `eval` method in the `AppExp` class uses the `evalRandsRef` method to create the bindings of the formal parameters to their appropriate references. The definition for `evalRandsRef` follows:

```
public List<Ref> evalRandsRef(Env env) {
    List<Ref> refList = new ArrayList<Ref>(expList.size());
    for (Exp exp : expList)
        refList.add(exp.evalRef(env));
    return refList;
}
```

Language REF (continued)

Remember that we always use value semantics for `let` bindings. This means that a Language REF program such as

```
let
  x = 3
in
  let
    y = x
  in
    { set y = add1(y) ; x }
```

evaluates to 3.

Our observation (see Slide 3.79) that any `let` can be re-written as an equivalent procedure application no longer applies with languages that implement call-by-reference semantics. Specifically, if we attempt to re-write the inner `let` in the above Language REF program as a procedure application using the algorithm given on Slide 3.79, we get

```
let
  x = 3
in
  .proc(y) {set y = add1(y) ; x } (x)
```

which evaluates to 4.

Language NAME

We now turn to a different parameter passing mechanism, *call-by-name*. In call-by-name procedure application, we bind each procedure's formal parameter to its corresponding *un-evaluated actual parameter expression*. Each time we reference a formal parameter in the procedure body, we evaluate its corresponding actual parameter expression *in the environment where the procedure was called* – the *calling environment*, and this value becomes the expressed value of the formal parameter.

Call-by-name has behaviors that differ from call-by-reference: (1) if we never reference the formal parameter in the procedure body, we never evaluate the actual parameter expression; and (2) every time we evaluate the formal parameter in the procedure body, we re-evaluate the actual parameter expression.

In the presence of side-effects, call-by-name has interesting properties that make it very powerful but often difficult to reason about. The language ALGOL 60 had call-by-value and call-by-name as its parameter passing mechanisms. ALGOL 60 had its greatest influence on languages such as Pascal, C/C++, and Java. Although call-by-name has been all but abandoned by modern imperative (side-effecting) programming languages – mostly because of its inefficiency, it still plays a role in functional programming: Scheme supports a variant, *call-by-need*, by means of `promise/force`; Haskell also supports call-by-need. We proceed to implement both call-by-name and call-by-need.

Two actual parameter expressions that can appear in procedure applications have *constant behavior*: They are `LitExp` and `ProcExp`. Evaluating these expressions do not produce any side-effects, and their expressed values can never change. (Don't confuse *evaluting* a procedure expression with *applying* such an expression.) Because of this, in call-by-name we can evaluate these actual parameter expressions directly during procedure application and can bind their corresponding formal parameters to new references to their values directly. In other words, we use value semantics for these actual parameter expressions.

Evaluating variables that appear as actual parameter expressions (`VarExp`) also do not produce any side-effects, but such variables may have denoted values that can change. We therefore use reference semantics for actual parameter expressions that are variables.

In the presence of side-effects, call-by-reference and call-by-name may give different results. Consider evaluating the following expression:

```
let
  x = 1
  f = proc(t,u)
        {
          set t = add1(t) ;
          u
        }
in
  .f(x, +(x,5))
```

In call-by-reference, when we evaluate the application `.f(x, +(x,5))`, the formal parameter `t` in the definition of `f` denotes the same reference that `x` denotes (initially containing 1), whereas the formal parameter `u` denotes a new reference to the value 6. Modifying `t` in the body of `f` changes the expressed value of `x` (because `t` and `x` denote the same reference) but does not change the expressed value of `u`. Thus this expression evaluates to 6.

Language NAME (continued)

```
let
  x = 1
  f = proc(t,u)
        {
          set t = add1(t) ;
          u
        }
in
  .f(x, +(x,5))
```

Consider now what happens when we evaluate `.f(x, +(x,5))` using call-by-name: The formal parameter `t` still denotes the same reference that `x` denotes (initially containing 1), but the formal parameter `u` denotes the (un-evaluated) expression `+(x,5)`.

The `set` operation in the body of this procedure increments the formal parameter `t`; but since `t` denotes the same reference as `x`, the value of `x` changes too, to two. When we then evaluate the formal parameter `u` at the end of the `proc` body, we evaluate the expression `+(x,5)` denoted by `u` *in the environment of the caller*. Since this expression gets evaluated after the `set`, and the value of `x` is now 2, the value of the expression `+(x,5)` (and thus the value returned by the procedure application) is `+(2,5)` or 7. Thus the entire expression evaluates to 7.

Language NAME (continued)

Consider the following definition:

```
define while = proc(test?, do, ans)
  letrec loop = proc()
    if test? then {do ; .loop()} else ans
  in .loop()
```

Using call-by-name, the expression

```
let x = 0 sum = 0 in
  .while(
    <=?(x,10),
    { set sum=+(sum,*(x,x)) ; set x = add1(x) },
    sum
  )
```

returns the sum

$$\sum_{x=1}^{10} x^2 = 385$$

Using call-by-reference as in the language `REF`, the expression never terminates because the actual parameter expression `<=?(x,10)` is evaluated only once, to 1 (true) when `x` is initially 0, and so the `test?` parameter is bound permanently to (a reference to) 1. Evaluating `test?` repeatedly always returns 1 (true), so the "loop" never terminates.

We proceed to implement call-by-name. We take our call-by-reference implementation as a starting point.

If an actual parameter is a literal expression (such as `4`), we bind the formal parameter to (a reference to) the literal value. If an actual parameter is a procedure, we bind the formal parameter to (a reference to) the procedure's closure in the calling environment. If an actual parameter is an identifier, we bind the formal parameter to the same reference as the actual parameter, using reference semantics.

If an actual parameter is any other kind of expression, we bind the formal parameter to a `Ref` object that captures the expression in the environment in which it was called and that can be evaluated, when needed, by the called procedure. We call such an object a *thunk*.

A thunk amounts to a parameterless procedure that consists of an expression and an environment in which the expression is to be evaluated. It looks just like a closure, except that there is no formal parameter list.

```
ThunkRef(Exp exp, Env env)
```

A `ThunkRef` is a `Ref`, since we want to de-reference (`deRef`) it whenever we refer to the corresponding actual parameter. We bind a formal parameter to a thunk reference only during procedure application. Thunks will otherwise not play a role in expression semantics.

To change from call-by-reference to call-by-name, we need to change the default `evalRef` behavior of the `Exp` objects so that `evalRef` returns a thunk for most expressions *except for* `LitExp`, `ProcExp`, and `VarExp`. This means that we define `evalRef` in the `Exp` class with its default behavior as follows:

```
public Ref evalRef(Env env) {
    return new ThunkRef(this, env);
}
```

For a `LitExp` and a `ProcExp`, a thunk is not necessary, so we return an ordinary `ValRef` as in the `REF` language:

```
public Ref evalRef(Env env) {
    return new ValRef(eval(env));
}
```

Finally, for a `VarExp`, we simply use reference semantics as in the `REF` language:

```
public Ref evalRef(Env env) {
    return env.applyEnvRef(var);
}
```

Language NAME (continued)

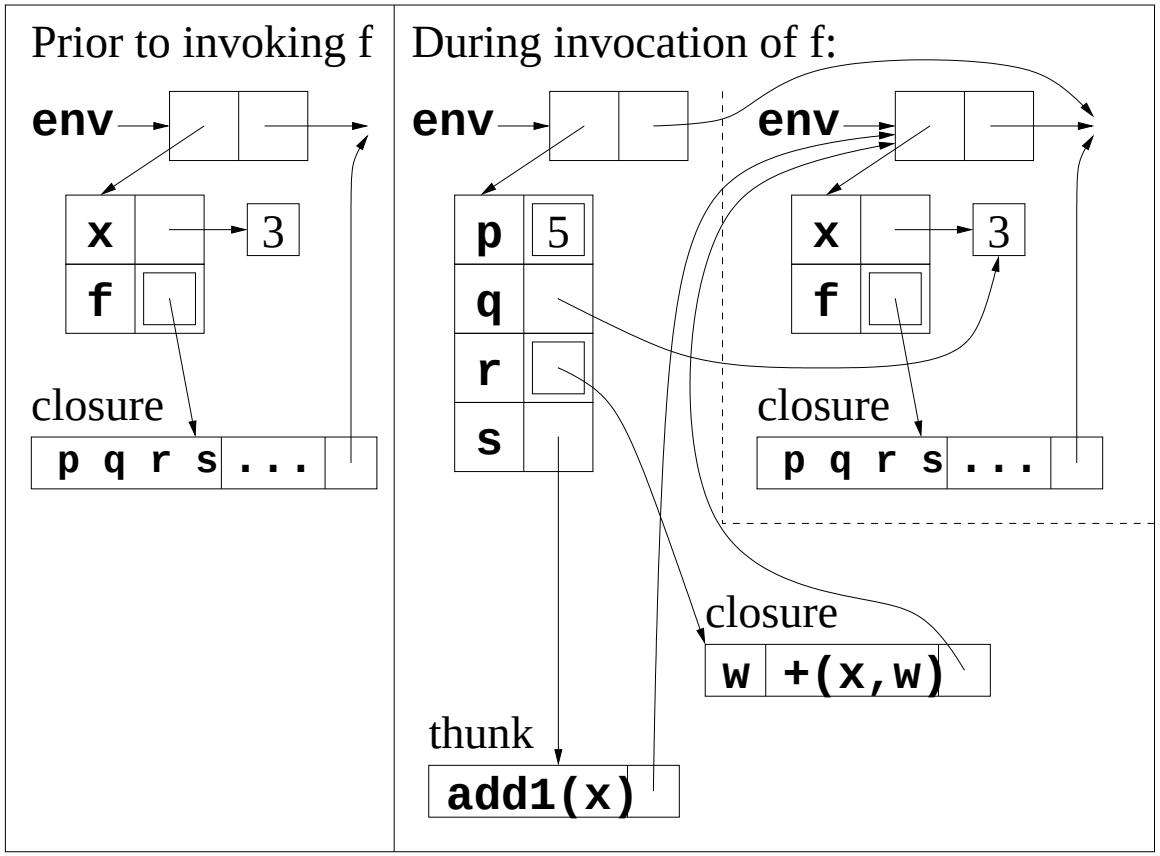
The `ThunkRef` class is straight-forward:

```
ThunkRef
%%%
public class ThunkRef extends Ref {

    public Exp exp;
    public Env env;

    public ThunkRef(Exp exp, Env env) {
        this.exp = exp;
        this.env = env;
    }

    public Val deRef() {
        return exp.eval(env);
    }

    public Val setRef(Val v) {
        throw new RuntimeException("cannot modify a read-only expression");
    }
}
%%%
```

Observe that the `setRef` method throws an exception. This method is only used to evaluate `set` expressions, and it doesn't make sense to have something other than a `VarExp` on the LHS of a `set`.

The following illustration may help you to understand how these bindings work. The example shows all four possible cases of actual parameter expressions: literal, variable, procedure, and other:

```
let x = 3
    f = proc(p,q,r,s) ...
in .f(5, x, proc(w) +(x,w), add1(x))
```



Language NEED

The call-by-need parameter passing mechanism is the same as call-by-name, except that a thunk is called at most once, and its value is remembered (*memoized*).

Suppose a procedure with formal parameter `x` is invoked with actual parameter `set z = add1(z)`, using call-by-need semantics. As with call-by-name, the formal parameter `x` is bound to the thunk with `set z = add1(z)` as its body, in an enclosing environment that we will assume has `z` bound to (a reference to) the value 8. When `x` is referenced in the body of calling procedure, its corresponding thunk is dereferenced, producing a result of 9 for `set z = add1(z)`. The thunk now remembers (*memoizes*) the value 9, and any further references to the formal parameter `x` in the body of the procedure will continue to evaluate to 9 without making any further changes to the variable `z`.

If call-by-name had been used in the above example, additional evaluations of `x` in the body of the procedure would result in evaluating the body of the thunk for each such evaluation, further modifying `z` and yielding values 10, 11, 12, and so forth.

In both call-by-need and call-by-name (and unlike call-by-reference), if the formal parameter is never referenced in the body of the procedure, the thunk is never evaluated. Compared to call-by-name, call-by-need reduces the overhead of repeatedly evaluating a thunk when evaluating the corresponding formal parameter.

Language NEED (continued)

Implementing call-by-need is easy, starting from the call-by-name interpreter. The principal change is to have a `Val` field named `val` in the `ThunkRef` class that is used to memoize the value of the body of the thunk. This field is initialized to `null` when the thunk is created. When the thunk's `deRef` method is invoked, it checks to see if the `val` field has been memoized (*i.e.*, is non-null). If so, the `deRef` method simply returns the memoized value. Otherwise, it evaluates the body of the thunk, saves the value in the `val` field (thereby memoizing it), and then returns that value; subsequent `deRef` calls simply use the resulting memoized value.

In the NEED language, the `ThunkRef` constructor initializes the `val` field to `null`, indicating that the thunk has not been memoized. This field is modified when the thunk's `deRef` method is called.

Language NEED (continued)

Here are the appropriate changes to `ThunkRef` ...

```
ThunkRef
%%%
public class ThunkRef extends Ref {

    public Exp exp;
    public Env env;
    public Val val;

    public ThunkRef(Exp exp, Env env) {
        this.exp = exp;
        this.env = env;
        this.val = null;
    }

    public Val deRef() {
        if (val == null)
            val = exp.eval(env);
        return val;
    }

...

}
%%%
```

Language NEED (continued)

You might have noticed in the `code` file that an instance of the class `ValRORef` is constructed by the `evalRef` methods in the `LitExp` and `ProcExp` classes. This is a slight change from the `NAME` language, where the instances were `ValRefs`. The `RO` part stands for "Read Only". We do this because it doesn't make sense for a literal or procedure to be modified.

Consider, for example, the following code:

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

One's intuition would be to think of the procedure application `.f(3)` as saying:

```
set 3=add1(3)
```

But of course this doesn't make any sense.

We have already seen that the `setRef` method in the `ThunkRef` class throws an exception. What we are now doing is to have this same behavior for *any* actual parameter expression except for a variable (where call-by-reference is the default).

Language NEED (continued)

The following example illustrates the difference between call-by-name and call-by-need:

```
let
  x = 3
  p = proc(t) {t;t;t}
in
  .p(set x=add1(x))
```

With call-by-name, when we apply the procedure `p`, its formal parameter `t` is bound to a thunk containing the expression `set x=add1(x)`. Each time we evaluate the formal parameter `t` in the body of the procedure `p`, its thunk is dereferenced, resulting in evaluation of the expression `set x=add1(x)`. So since we evaluate `t` three times in the body of `p`, the expression `set x=add1(x)` gets evaluated three times, incrementing `x` from 3 to 6. Consequently, the entire expression evaluates to 6.

With call-by-need, the first time we evaluate `t` in the body of `p`, its corresponding actual parameter expression `set x=add1(x)` is evaluated, which has the side-effect of incrementing the value of `x` to 4 and evaluates to 4. However, the thunk memoizes the expressed value of 4, so any further references we make to `t` evaluate to 4. Consequently, the entire expression evaluates to 4.

Language NEED (continued)

Here's another example illustrating the difference between call-by-reference and call-by-name/need. Examine the definition of `seq`, which seems to recurse infinitely but doesn't with call-by-name (why?).

```
define pair = proc(x,y)
  proc(t) if t then y else x
define first = proc(p) .p(0)
define rest = proc(p) .p(1)
define nth = proc(n,lst)    % zero-based
  if n then .nth(sub1(n),.rest(lst)) else .first(lst)
define seq = proc(n) .pair(n,.seq(add1(n)))
define natno = .seq(0)      % all the natural numbers!!
%% The above never terminates with call-by-reference.
%% With call-by-name or call-by-need, we get:
.first(natno)               % => 0
.first(.rest(natno))        % => 1
.first(.rest(.rest(natno))) % => 2, and so forth ...
.nth(100,natno)             % => 100
```

Order of evaluation

Let's examine the following example:

```
let
  x = 3
in
  let
    y = {set x = add1(x)}
    z = {set x = add1(x)}
  in
    z
```

Consider the inner `let`. We know that the right-hand side expressions (here written inside curley braces for clarity) are evaluated before their values are bound to the left-hand variables. Our language specifies the order in which the right-hand side expressions are evaluated, namely left-to-right.

In the absence of side-effects, *i.e.* in our early interpreters without `set`, the order of evaluation of the RHS expressions wouldn't matter. However, when side-effects are possible, as in our interpreters such as `SET` and `REF`, the order of evaluation does matter.

In the above example, if the second `set` were to be evaluated first, then `z` becomes 4 and `y` becomes 5, so the entire expression evaluates to 4 – the value of `z`. If the order of evaluation is reversed, the entire expression evaluates to 5.

Order of evaluation (continued)

A similar situation exists when evaluating actual parameter expressions, as shown by this example, assuming call-by-value semantics.

```
let
  x = 3
  p = proc(t,u) t
in
  .p(set x = add1(x), set x = add1(x))
```

When the actual parameters are evaluated left-to-right as specified in our languages, `t` would be bound to 4 and `u` to 5, so the entire expression evaluates to 4. If the evaluation order had been right-to-left, the entire expression would evaluate to 5.

Order of evaluation (continued)

You can see that both `evalRands` and `evalRandsRef` use `for-each` loops (also called enhanced `for` loops) to traverse and evaluate the expressions in the list of actual parameters. The traversal is guaranteed by the Java API specification to be "natural", in the sense that the elements of the list are visited in ascending item number order. Here is the code for `evalRands` in the `Rands` class:

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

Our order of evaluation semantics depends on the behavior of Java's `for-each` order of evaluation mechanism, which is guaranteed to be left-to-right. If we had chosen right-to-left semantics, we could, instead, explicitly traverse the `expList` from last to first if we wished.

The point is that, to make any language semantics well-defined and unambiguous, it is necessary to specify the order of evaluation. Unless the language specification clearly addresses the issue of order of evaluation, the language implementor can choose any evaluation order. *Let the buyer beware!*

Order of evaluation (continued)

Both Java and Python specify that actual parameter expressions are evaluated left-to-right. However, the C language specification explicitly states that the order in which actual parameter expressions are evaluated is *undefined* – which means that the evaluation order is implementation dependent. In the following C program, the output is 3 (using the GCC compiler on the date this file was last modified), which shows that the the operand expressions `foo(3)` and `foo(5)` are evaluated right-to-left. Your mileage may vary!

```
#include <stdio.h>
#include <stdlib.h>
static int xx = 0;
void foo2(int x1, int x2) {
    return;
}
int foo(int x) {
    xx = x;
    return x;
}
int main(int argc, char ** argv) {
    foo2(foo(3), foo(5));
    printf("xx=%d\n", xx);
    return 0;
}
```

Order of evaluation (continued)

Order of evaluation does not matter in languages without side-effects, which makes functional languages immune to order of evaluation issues. See `http://en.wikipedia.org/wiki/Evaluation_strategy` for more information about order of evaluation.

Another way to avoid order of evaluation problems is to require that all procedures have at most one formal parameter. In languages that use this approach, coupled with with call-by-need, there is never an "order of evaluation" issue because there is never more than one actual parameter to evaluate.

While you may think that a language with procedures having only one formal parameter might be limited, it's possible for such a language to behave like having multiple formal parameters using an approach called "Currying", as employed in the Haskell programming language – named after Haskell Curry. The following slide gives an example.

Order of evaluation (continued)

Here is an example without currying:

```
let
  x = 3
  y = 5
  p = proc(t,u) +(t,u)
in
  .p(x,y) % => 8
```

Here is semantically equivalent code that has exactly one formal parameter per `proc`:

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

In the second example above, `x` must be evaluated first, so that the procedure `.p(x)` can then be applied to the value of `y`.

Aliasing

Side-effecting languages that use call-by-reference suffer from another danger. Consider, for example, the following program using the REF language semantics:

```
let
  addplus1 = proc(x,y) {set x = add1(x) ; +(x,y)}
in
  .addplus1(3,3)
```

It's clear that this program returns 7. But what about the following program?

```
let
  a = 3
  addplus1 = proc(x,y) {set x = add1(x) ; +(x,y)}
in
  .addplus1(a,a)
```

Aliasing (continued)


```
let
  a = 3
  addplus1 = proc(x,y) {set x = add1(x) ; +(x,y)}
in
  .addplus1(a,a)
```

Using call-by-reference, when `addplus1` is applied to the actual parameters `a` and `a`, both formal parameters `x` and `y` of `addplus1` refer to the *same cell* as `a`. Therefore the `set x = add1(x)` expression is equivalent to the expression `set a = add1(a)` which increments `a` to 4, and the next expression `+(x,y)` is essentially equivalent to the expression `+(a,a)` which now evaluates to 8. Thus the value of the program is 8.

Aliasing occurs when two different formal parameters refer to the same actual parameter. As this example shows, aliasing can lead to unexpected side-effects and should be avoided. **Of course, the best way to avoid problems such as order of evaluation ambiguities and aliasing is to avoid using langauges with side-effects!**