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.

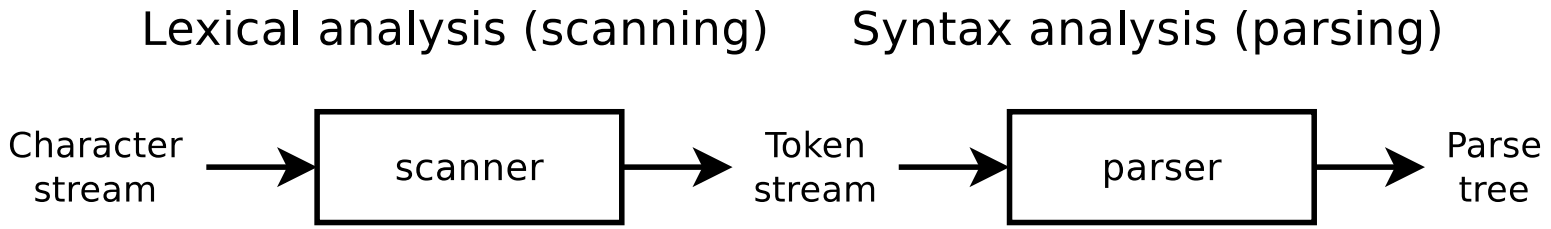
The PLCC tool set is designed to make it easy to build a scanner and parser for a large collection of programming languages. While it is not "industrial strength" like many of the standard tool sets, it is easy to learn. The target language for PLCC is Java, so all of the source code that PLCC generates 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
```

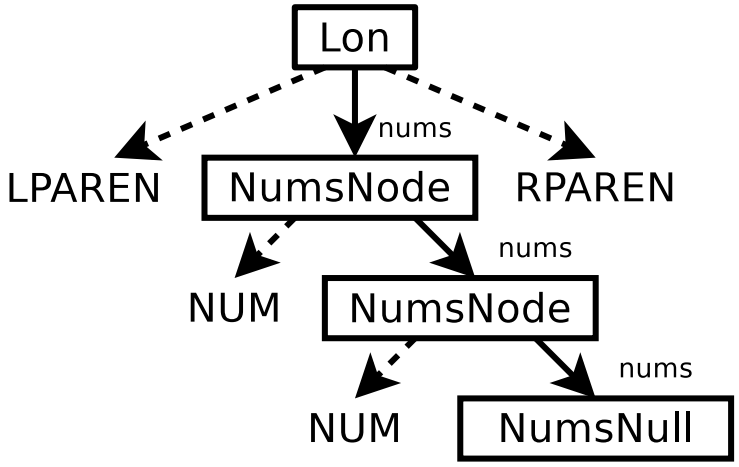
In this example, the root of the parse tree is a `Lon` object whose `nums` field is an instance of `NumsNode`. This instance in turn has a `nums` field that is also an instance of `NumsNode`. And finally, this instance has a `nums` field that is an instance of `NumsNull`. A `NumsNull` object has no fields. 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)

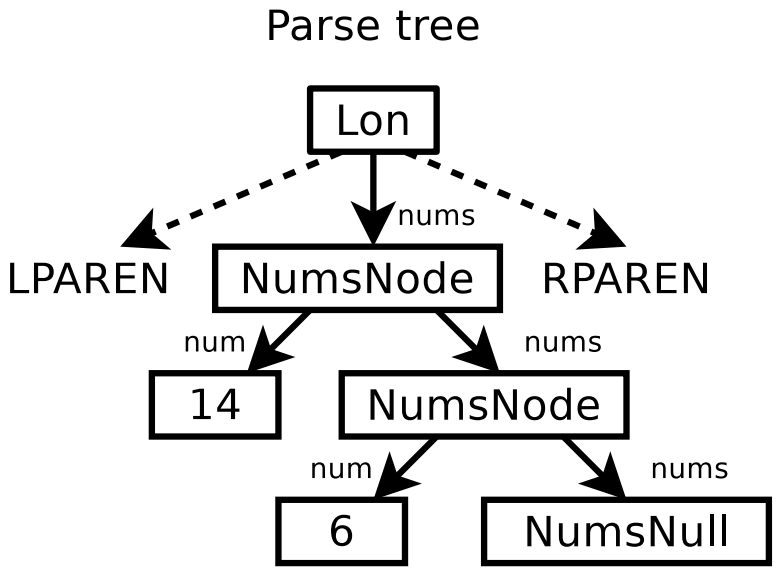
```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

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.

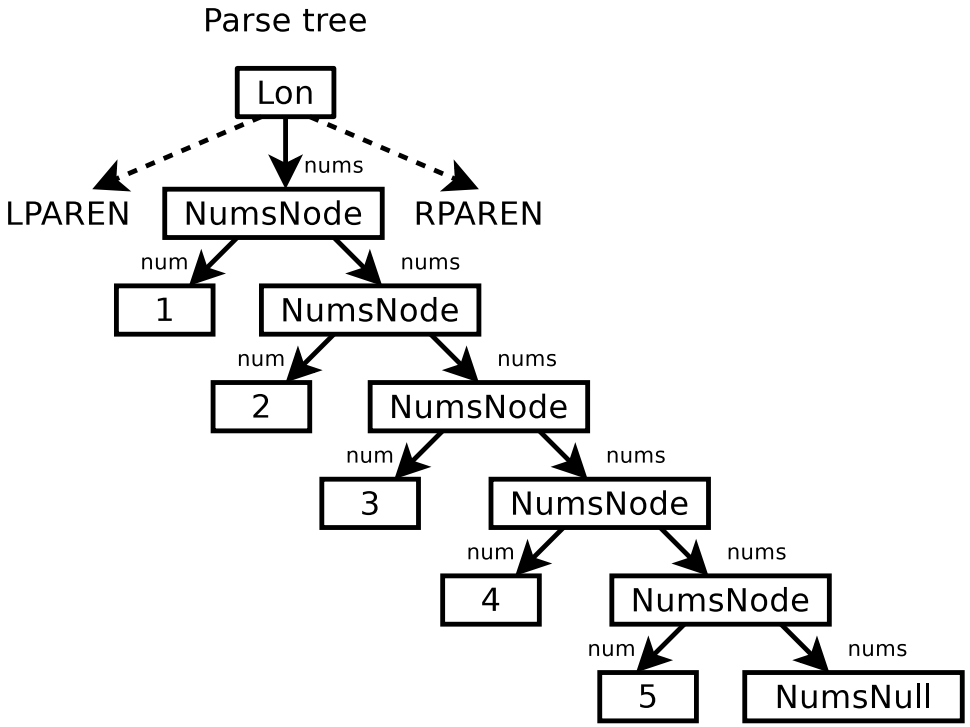
```
<nums>:NumsNode   ::= <NUM> <nums>
<nums>:NumsNull   ::=
```

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 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)`:

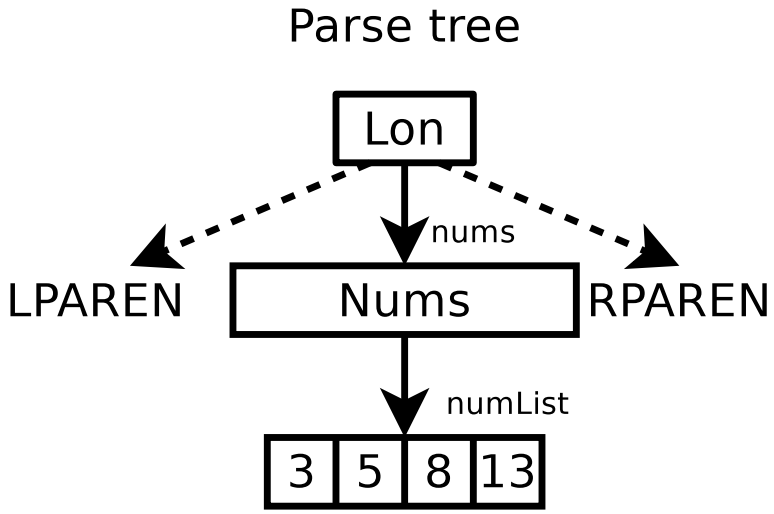
```
--> (3 5 8 13)
<lon>
| LPAREN "("
| <nums>
| | NUM "3"
| | NUM "5"
| | NUM "8"
| | NUM "13"
| RPAREN ")"
```

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;
    ...
}
```