Specifying Syntax

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

A *programming language* also defines a language in the theory se the strings in a programming language are called *programs*, and t called *tokens* (which we discussed in Slide Set 0). The *syntax* of language is a set of rules used to specify the programs in the langu gramming languages use a context-free grammar (or something clos ify what sequences of tokens belong to the language.

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

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

Backus-Naur Form (BNF)

BNF is a meta-language used to specify a context-free grammar. modern programming language uses some sort of BNF notation to de We work first with examples of languages that define two simple d lists and trees. Remember that the “alphabet” of a programming la of tokens, so any definition of the syntax of a language must first spe In Slide Set 0, we showed how to specify tokens using PLCC.

Our first example is to define a language whose “programs” are lis 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 formula

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

In these formulas, the token names `LPAREN` and `RPAREN` stand for ‘`(`’ and right parenthesis ‘`)`’, respectively. The token name `NUM` (unsigned) decimal number. [Think about what regular expression these.] Conforming to PLCC specifications, we use all-UPPERCA 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 l
<nonterm-symbol> where nonterm-symbol is an identifier (
digits, and underscores). PLCC requires that the first character of th
a lowercase letter. A <nonterm-symbol> expression is called a *n*
the example above, the nonterminals are <lon> and <nums>.

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

Notes: The term *syntactic category* is sometimes used instead of th
minal, and the term *terminal* is sometimes used instead of *token na*
using a token name such as LPAREN, some languages use BNF for
the corresponding actual character string such as '('. In the exampl
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 enc
reading. These shortcuts are usually called Extended BNF, or sim
example, instead of writing two different formulas with <nums> on
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 }$\star$

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

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

In these notes, we casually use the term *token* to refer both to name of the abstraction (like '`NUM`') used in a BNF rule and to i ing *lexeme* (like '`23`') used in a program. In most instances, you s 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 la syntactically correct).

A programmer's job is to construct syntactically correct progr gramming language, illustrating the first use of a grammar. T called *programming*.

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

Given a grammar, we want to generate an algorithm to carry out p
gram [written in Java, or C, or whatever] that carries out this algor
unsuprisingly, a *parser*. In this course, all of our parsers are Java pro

Parsing (continued):

Given a BNF grammar, we describe here an algorithm (written in En
that parses a sentence in the language using what's called a *leftmost*
algorithm returns "success" if the parse is successful, "failure" other

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

Parsing (continued):

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

   (b) If the leftmost unmatched term in the sentential form is a
   choose a rule from the grammar with this nonterminal as its L
   the nonterminal with the (possibly empty) RHS of the chose
   rule to choose depends on finding a rule that is most likely t
   derivation. For some grammars, there is only one choice: t
   are said to be *predictive*. All of the grammars we use in this
   dictive.] If no rule can apply, return failure.
4. If the unmatched sentence is empty, return success: the target sen
   successfully parsed. Otherwise, return failure.

It is possible, for some grammars, that Step 3 loops indefinitely. H
of the grammars that we encounter in this course, this step exits in a
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 wi
minal in the grammar (the *start symbol* – in this case it's `<lon>`) as the sententia

sentential form	unmatched sentence	algorithm s
<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>**
⇒ (14 6)		3a
⇒ *success!*		4

In the above derivation, the leftmost unmatched token name or nonterminal is sho
The underlined parts are the matched lexemes for the unmatched token name as
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 t
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.
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* **<nu**
⇒ (14 **NUM** <nums> RPAREN	?? NUM doesn't match
⇒ (14 **<nums>** RPAREN	(6) – *try second* <
⇒ (14 **RPAREN**	?? RPAREN doesn't ma
⇒ *failure!*	

We can conclude that the string '`( 14 ( 6 )`' does not conform
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> RPA
```

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

Here are some examples of trees that you can check for syntactic co 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> RPA
```

In a given sentential form, the nonterminal `<tree>` can be replace hand side of *either* rule having `<tree>` as its left-hand side. For ex 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> RPAR`
⇒ `...`

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 leftı
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 a
out a parse of the sentence, producing either success or failure. Bu
from the BNF specification of a language is conceptually simple:
to write a program to carry out the steps of the parsing algorithm.
is the algorithm's essential *nondeterminism*: given a nonterminal
form, how do we replace this nonterminal with the right-hand side o
grammar rule having this nonterminal as its left-hand side? (For so
there may even be more than one "correct" rule that leads to a suc
such grammars are said to be *ambiguous*.)

Parsers (continued)

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

There are several industrial-strength tool sets in common use that c
like grammar specification into a parser. Many of these tool sets
as the target language: input to such a tool set is a grammar spe
the output is a set of target language programs that, when compil
parser. Learning how to use some of these tool sets can be a dau
the generated parsers can be obscure. Furthermore, most of these to
into two separate parts: a lexical analysis (scanning) part and a s
(parsing) part.

Parsers (continued)

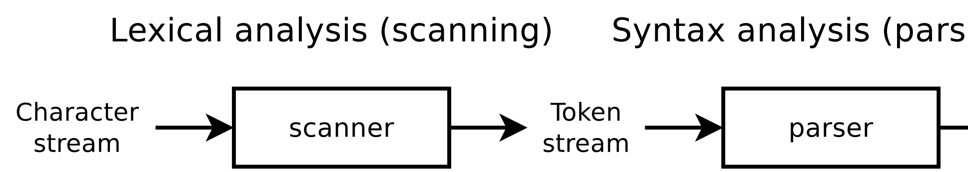
The PLCC tool set is designed to make it easy to build a scanner a
large collection of programming languages. While it is not "industria
many of the standard tool sets, it is easy to learn. The target langu
is Java, so all of the source code that PLCC generates consists of
Java programs. To use PLCC, you only need to be comfortable read
programs in Java.

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

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

Parsers (continued)

The following diagram shows how the output of a scanner (a progr
ments lexical analysis) becomes input to a parser (a program that imp
analysis). The output of a parser is a *parse tree* – which we describe
later.



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

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

Parsers (continued)

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

From each BNF grammar rule for a language, PLCC generates rule such as

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

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

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

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

Parsers (continued)

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

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

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

(Any Java class names are OK for these annotations, but good naming conventio and the names must be unique among the Java class names that PLCC generates to separate the nonterminal from its annotated Java class name. The RHS entries rules are unchanged.

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

The `NumsNode` class has a field named `nums`, which is an instance of the `Num` the first letter of `<nums>`). The `NumsNull` has no fields.

Parsers (continued)

Here is a complete text file representation of Language `LON`, inclu
and syntax specification sections, suitable for processing by `plccr`
we have included the annotations for the two `<nums>` rules as de
previous slide. Also notice that a line containing a single '`%`' separ
specification from the syntax specification (BNF rules).

```
# Language specification for a list of numb
# 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 (`%`)
the *lexical specification* of the language. For this language, these
whitespace (including spaces, tabs, and newlines) should be skipped
a string of one or more decimal digits, and that `LPAREN` and `RPA`
characters '`(`' and '`)`', respectively.

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

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 ::=
%
```

Observe that the `LPAREN` and `RPAREN` token specifications use r
sions having a backslash '\'. This is because parentheses have a sp
in regular expressions, and the backslash turns off this special me
example, the regular expression `'\('` really matches the left paren
(You should take this opportunity to review how to write and interp
pressions in the Java `Pattern` class.)

To complete this example, assume that you have created a director
and that in this directory you have created a file named `gramma`
lines appearing above. In your `LON` directory, run the `plccmk` scrip

Parsers (continued)

```
plccmk
```

Your script output should appear as follows:

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

Abstract classes:
  Nums

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

Parsers (continued)

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

(The above text shows just the syntax section of the `grammar` file examining.) In your `LON` directory, change to the subdirectory nam `plccmk` command has created this subdirectory and populated it w code generated by `plcc.py`. In this subdirectory you will find 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 the line beginning with `<lon>` results in the file `Lon.java` being `Java` subdirectory. As you can see from looking at the Java code in directory for the `NumsNode` and `NumsNull` classes, both of these the `Nums` abstract class. This is because the `<nums>` nonterminal LHS of two grammar rules.

Parsers (continued)

For every BNF grammar rule, PLCC creates a Java class uniquely the rule. For BNF rules that have the same nonterminal appearing multiple rules, PLCC creates an abstract class based on the nonterm the annotated class names become derived classes of this abstract c

The first BNF nonterminal in a language's syntax specification sect *symbol* for the language. Given a program in the language, the pars instance of the class of the start symbol, which is the root of the pa program.

Consider Language `LON`, whose `grammar` file syntax specifica shown below. The start symbol is `<lon>`, and the corresponding `Lon`. Given a program in Language `LON`, the output of the parser is the `Lon` class. We will see shortly how to interpret this instance 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 gener parser with class name `Parse` that uses the Java classes created from processi syntax specification sections. To run this parser in language directory `LON`, for `Parse` class file using '`-cp Java`' on the `java` command line so the Java int where to find the `.class` files. This program displays a prompt '`--> `', after w standard input, "programs" in Language `LON` to parse. (Standard input typically keyboard, sometimes called the *console*.) If the parse for a particular program suc program displays the string `"OK"`. If the parse fails, it displays an error message i parse failed.

For example, in the `LON` directory (after having run `plccmk`), enter the followin

```
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 equival `java -cp Java Parse`.

Parsers (continued)

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

When invoked with command-line arguments, the `Parse` program given as filename arguments and processes them as if they were ent dard input.

Three command-line arguments have special meaning when runni program:

- The '`-n`' command-line argument disables displaying the '`-->` reading programs from standard input.
- The '`-t`' command-line argument toggles displaying a *parse tra* are parsed. A parse trace is a text representation of the parse tre the parser: an example of such a parse trace appears on Slide 1.3 not displaying the parse trace.
- The '`-v`' command-line argument toggles displaying the command-line files when processing them in left-to-right orde not displaying the name.

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

Parsers (continued)

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

PLCC also generates an interactive parser/evaluator called `Rep` the `Java` subdirectory along with the `Parse` program. `Rep` exe Eval-Print" loop that displays a prompt, *Reads* program input from and parses it – producing a parse tree, *Evaluates* the parse tree (program), and *Prints* the result obtained from "running" the progr this result is a representation of the "value" of the program's parse tı of the Java class defined by the start symbol of the language's BNF

How to determine the "value" of a parse tree is the essence of *sem* which we describe in detail for each of the languages we consider in some cases, this value may be just a re-cast version of the program 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()` val tree Java instance. For Language `LON`, this value is a Java `Stri` `Lon@xxxx`, where the `xxxx` part is the hexadecimal address of w resides in memory. Observe that `Lon` is the Java class that defines tl of the `LON` BNF rules.

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

Parsers (continued)

Here's a sample interaction using this trace feature for Language
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 n
instance of `NumsNode`. This instance in turn has a `nums` field tha
stance of `NumsNode`. And finally, this instance has a `nums` field tha
of `NumsNull`. A `NumsNull` object has no fields. The trace a
each token is matched by the parser, along with the lexeme from the
that matched the token. The line '`Lon@372f7a8d`' shows the resu
which is a `Lon` object representing the root of the parse tree. The
part represents the location in memory where the `Lon` object reside

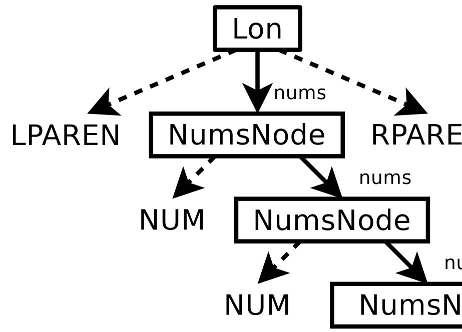
Parsers (continued)

Using this trace feature, we can construct a visual representation o
whose root is an object of type `Lon`. The diagrams below show
output on the left and the resulting parse tree on the right. No
are represented by boxes. The dashed lines point to tokens that
conceptual parse tree but that do *not* correspond to fields in the no
this is the case when the BNF entries on its RHS are tokens that are
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 J
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*
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 param
`Nums`.

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

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

Parsers (continued)

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

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

Parsers (continued)

To remedy this situation, we want to define a field in the NumsN
captures the NUM token obtained by the parser, which in turn allow
the token's lexeme. We already use angle brackets for all *nontermin*
of grammar rules, and these nonterminals automatically become fie
class. So to capture *tokens* on the RHS, we use angle brackets for
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 toke
converted to lowercase) of type Token (since NUM is the name of
grammar file).

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

Parsers (continued)

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
signature of the NumsNode constructor, as shown here (compare w

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

Parsers (continued)

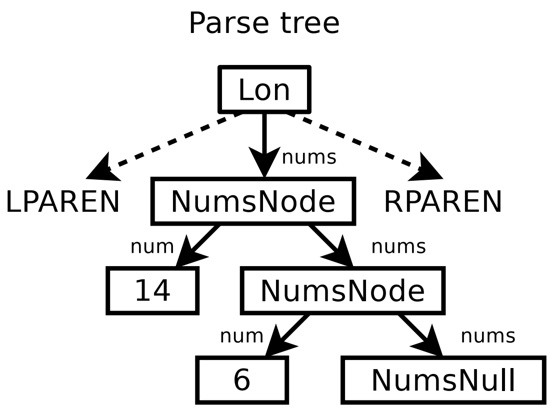
```
<lon>             ::= LPAREN <nums> RPAREN
<nums>:NumsNode ::= <NUM> <nums>
<nums>:NumsNull ::=
```

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

The following diagram shows the parse tree for the string

```
( 14 6 )
```

using this revised grammar. Notice that each NumsNode object now has a nur
containing decimal digits.



Parsers (continued)

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

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

But there is a problem with the Interior constructor signature.
allow multiple field names or constructor formal parameters with t
specifically, in this case, there cannot be two field names with th
Here is what PLCC has to say about this when given the above gra
has been folded for clarity):

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

Parsers (continued)

The solution is to use different identifiers for these field names ar responding constructor formal parameters. PLCC allows duplica (in angle brackets) to be annotated – much like we have seen for nonterminal names – with alternate names that avoid this conflict.

In the case we are considering, we can resolve this issue in th `<tree>:Interior` grammar rule by using the identifier `lef` `<tree>` field and the identifier `right` for the second `<tree>` f here:

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tre
                    Interior(Token symbol, Tree left, T
```

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 <tre
                     Interior(Token symbol, Tree left, T
```

When processed by PLCC, we get the following interaction using t 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 fo

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

Parsers (continued)

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 `Nums`

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

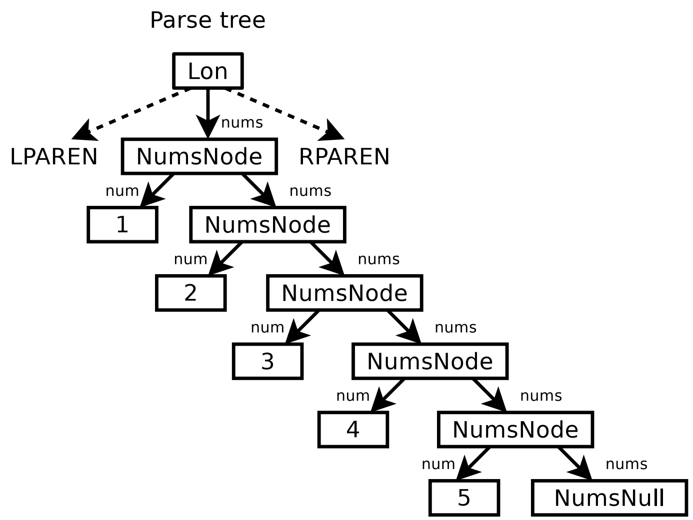
Clearly, when parsing the `<nums>` grammar rules, you get `NumsNode` instances but just one final `NumsNull` instance. RHS of the first `<nums>` rule has `<nums>` as its last entry (which rule is *right recursive*), this rule results in a recursive parsing loo when there are no more `NUM` tokens in the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

Parsers (continued)

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 d as the number of elements in the list grows. Clearly a list of nun entries would have 100 `NumsNode` nodes in the tree, weighted sigr right. How can we structure our tree nodes to avoid this?

Parsers (continued)

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

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

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

Note: The use of '**' in the notation we have just introduced sho
Kleene star repetition notation used in EBNF as well as in regular e

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

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

Parsers (continued)

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

Here is an (edited) example of a parse trace for the list of numbers

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

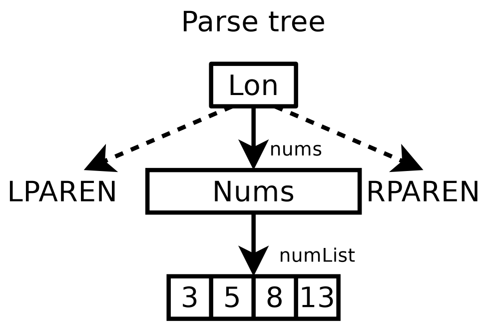
Compare this with the parse trace using the previous grammar that d
**= construct. The previous parse trace drifts to the right as additio
are encountered. Using the **= construct, the parse trace becomes

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

Parsers (continued)

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

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



This parse tree has fewer nodes, and the individual `NUM` token valu
aged together into a single `numList` structure.

Observe that an RHS entry of the form `<UPPERCASE>` in a BNF
corresponds to a leaf node in the parse tree, and an RHS entr
`<lowercase>` corresponds to a subtree of the parse tree. Keep thi
you examine the BNF grammar rules for a particular language and th
ing Java source files.

Parsers (continued)

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

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

This default behavior resides in the Java class `_Start`. This class c
method named `$run()` that simply displays the `toString()`
stance (see the Java code for `_Start.java` in the Java subdirecto
`LON`). Since the Java class `Lon` extends the `_Start` class, evauatin
method of a `Lon` object defaults to evaluating the `$run()` method o
which gives us the behavior we have already seen. `Rep` simply cal
method on the `Lon` instance obtined by parsing the program.

Parsers (continued)

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

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

Fortunately, PLCC allows us to add methods to PLCC-generated
including the added methods in the `grammar` file. Every time p
these added methods are incorporated into the Java source files auto

Parsers (continued)

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

For example, if we want to define the `$run()` method in the `Lon`
file, we put the following lines into the `grammar` file *in the sem*
following the syntax section. The semantics section is separated f
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.`
the two lines containing `%%%` bracket the Java code to be added. Thi
be used to add Java code to *any* PLCC-generated Java file arising
grammar lines. PLCC inserts this added code at the end of the cla
the Java code generated automatically by PLCC for the class. In
`Lon` class, the above Java code appears at the end of the automati
`Lon.java` class code.

Parsers (continued)

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

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

```
nums.numList
```

So we can display the `NUM` entries by iterating over this list, displa (as Java `Strings`) as we do so. Here is the completed version of th fication that adds the `$run()` method to the `Lon.java` file to achi result. Since the parse tree does not include the tokens for parenth adds back the parentheses in the output to make the output look "pre

```
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 the

```
# 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(")");
    }
%%%
```

Parsers (continued)

Continuing with our list-of-numbers example, suppose we want ou
quire 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*
in a repeating grammar rule. The separator must be a bare token n
brackets) at the end of the rule, preceded by a '+' character. So
separate the items in our lists by a comma (with token name COMMA
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, th
ates *exactly the same parse trees* for this language as for the previou

Parsers (continued)

**Slide set 1a provides a summary of PLCC features. You shou
yourself with this section in preparation for material in subsec
this course.**

Static Properties of Variables

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

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

A programming language that is designed solely for the purpose of evaluating ex
an *expression-based language*. Many of the languages we construct in these not
based, especially early on. Scheme, ML, and Haskell are examples of exp
guages used in practice. Expression-based languages do their looping principally

A programming language whose language constructs are designed to "do somet
signing the value of an expression to a variable or displaying the value of an expr
output) is called an *imperative language*. In an imperative language, a language c
somthing" is called a *statement*; Imperative languages are therefore often called *st*
Java, and Python are examples of imperative languages used in practice. Imp
do their looping principally using a form of "goto".

Expression-based languages get their power from defining and applying *function*
describing such languages is *functional*.

Static Properties of Variables (continued)

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

At run-time, how can you find the value bound to a variable? Ther
approaches:

- if the location of the value bound to a variable can be determine
  variable appears *in the text of a program*, we call it *static bindin*
- if the location of the value bound to a varible can only be deter
  the variable is accessed *during program execution*, we call it *dy*

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

Static Properties of Variables (continued)

We consider only static bindings for now, so the program text tells u
mine variable bindings.

For a given variable, the *scope* of the variable is the region of cod
variable's binding can be determined. Consider the following Java p

```
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 o
   }
}
```

In the above code, the scope of `y` is *global*, from its de
`public static` variable to the end of the class. The scope of
global, known only within (and throughout) instances of the class F
of `f` and `x` are *local*, from their declarations to the end of the `mair`
The scope of `args` is also local, from the beginning of the method b

Static Properties of Variables (continued)

It is possible for one symbol to have multiple bindings depending on where it occu
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 va
is from its point of declaration to the end of the *block* in which it is defined, as sho
comments. In this case, we say that the definition of `x` in the block *shadows* the g
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, identifiers *occur bound* in the main procedure.

However, the variable y is not defined anywhere in the procedure main. In this the identifier y *occurs free* in the main method, but it *occurs bound* in the enclos

The Lambda Calculus – OPTIONAL SECTION

The following grammar defines a formal language called "the lam This language plays an important role in the foundations of compute ilar to Turing Machines). The PROC token is the string proc, and LPAREN, RPAREN, LBRACE, RBRACE, and DOT tokens are straigh the examples below.

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

Consider the sentential form (remember what that means?) in thi tained from the second grammar rule, where s replaces <SYMBOL>

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

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

The Lambda Calculus (continued)

Occurs Free, Occurs Bound (informal definitions):

A symbol x *occurs free* in an expression E if x appears somewhere
that is not bound by any declaration of x in E. A symbol x *occurs l*
appears in E in such a way that is bound by a declaration of x in E. I
the same symbol to occur both bound and free in different parts of
(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 bou
                               ; 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
```

The Lambda Calculus (continued)

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

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

- *Rule 1*:
  `E` is of the form `proc(y){E′}` where `x` occurs bound in `E′`
  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
  ```

The Lambda Calculus (continued)

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>
<exp>:App  ::= DOT <exp>rator LPAREN <exp>rand RPAR
%
```

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

Static Properties of Variables (continued)

Most imperative programming languages are *block structured* and u
ing, another term for static scope rules. A *block* is a region of code
one or more variable declarations and continuing to the end of the cc
declarations are active. In C, C++, and Java, blocks are delimited by
of braces '{...}'.

In some languages, blocks may be *nested*, in which case variable bi
blocks may be *shadowed* by bindings in inner blocks. Consider, f
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
with the same name in the *innermost* block that defines the variable.
does not allow the same variable to be defined both in an outer block
block.)

Static Properties of Variables (continued)

Let's return to our C++ example. The following picture shows th
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 l
outwards (up) until a variable declaration with the same variable nar

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

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