Induction, Recursion, and Scope

A data type is a set of values and operations on those values. In object-oriented languages, a data type is represented as a class: the data type values are instances of this class, and the data type operations are the methods defined in the class.

Specification of values by Induction – Examples

A subset $S$ of the natural numbers:

- **Basis Case:** $1 \in S$
- **Induction rule:** whenever $x$ belongs to $S$, $3 + x$ belongs to $S$
- **Uniqueness rule:** A number $x$ is in set $S$ only if it can be constructed using these two rules.

*Note:* $S$ is the set of natural numbers $n$ such that $n \mod 3 = 1$.

A list of numbers:

- **Basis Case:** () is a (empty) list of numbers
- **Induction rule:** if ($nums$) is a list of numbers (with $nums$ possibly empty) and $n$ is a number, then ($n$ $nums$) is a list of numbers.
- **Uniqueness rule:** A list of numbers can only be constructed using these two rules.

*Examples:* The following are examples of lists of numbers:

()  
(1 3 7)  
(2)
Backus-Naur Form (BNF)

BNF is a meta-language that can be used to define the values of a data type.

Here is a BNF description of the set $S$ described on slide 1.1, using two BNF formulas:

$$
<\text{onemodthree}> ::= \text{ONE} \\
<\text{onemodthree}> ::= \text{THREE PLUS} <\text{onemodthree}>
$$

In these formulas, the token names $\text{ONE}$ and $\text{THREE}$ stand for the decimal numbers 1 and 3, respectively, and the token name $\text{PLUS}$ stands for the operator symbol $+$. The values defined by these BNF rules are strings of the following form: 1, 3+1, 3+3+1, and so forth. (Their corresponding arithmetic values are 1, 4, 7, and so forth.)

Here is a BNF description of the list of numbers example described on slide 1.1, using three BNF formulas:

$$
<\text{lon}> ::= \text{LPAREN} <\text{nums}> \text{RPAREN} \\
<\text{nums}> ::= \text{NUM} <\text{nums}> \\
<\text{nums}> ::= 
$$

In these formulas, the token names $\text{LPAREN}$ and $\text{RPAREN}$ stand for left parenthesis '(' and right parenthesis ')', respectively. The token name $\text{NUM}$ stands for any decimal number. (Think about what regular expressions would match these.) We will always use all-uppercase letters for our token names.
Every BNF formula has the form

\[ LHS ::= RHS \]

The \textit{LHS} (Left-Hand Side) of a BNF formula always has the form \texttt{<nonterm-symbol>} where \texttt{nonterm-symbol} is an identifier, normally written in lowercase. A \texttt{<nonterm-symbol>} expression is called a \textit{nonterminal}. In the example above, the nonterminals are \texttt{<lon>} and \texttt{<nums>}.

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

\textbf{Notes:} The term \textit{syntactic category} is sometimes used instead of the term \textit{nonterminal}, and the term \textit{terminal} is sometimes used instead of \textit{token name}. Instead of using a token name such as \texttt{LPAREN}, some BNF formulas just use the corresponding actual character string such as “(".
List of numbers example (copied from previous slide):

\[
\begin{align*}
<\text{lon}> &::= \text{LPAREN} \ <\text{nums}> \ \text{RPAREN} \\
<\text{nums}> &::= \text{NUM} \ <\text{nums}> \\
<\text{nums}> &::= \\
\end{align*}
\]

BNF has some shortcuts that we will not use but that you may run into in your reading. These shortcuts are usually called Extended BNF, or simply EBNF. For example, instead of writing two different formulas with \(<\text{nums}>\) on the LHS, one can use \textit{alternation} notation “\(\mid\)”:

\[
<\text{nums}> ::= \text{NUM} \ <\text{nums}> \mid \epsilon
\]

Note: ‘\(\epsilon\)’ means the empty string.

One could also use the \textit{Kleene star} notation to define \(<\text{nums}>\):

\[
<\text{nums}> ::= \{ \text{NUM} \}^*
\]
The purpose of a BNF grammar is to define the set of “legal” strings that conform to the grammar rules. A “legal” string is one that can be derived from the grammar using the grammar rules, in a way that we will define next. A legal string is called a *sentence*. 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 strings that conform to the grammar rules.

Given a BNF grammar, the set of all of its syntactically correct strings is called the *language* of the grammar. So if we had a grammar for the Java programming language, for example, then the set of strings that can be derived from this grammar would be the set of all syntactically correct Java programs.

A grammar can be used:

- to construct syntactically correct strings, or
- to check to see if a particular target string 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 is called *programming*.

A compiler’s job is, in part, to check a program for syntactic correctness, illustrating the second use of a grammar. This is called *parsing*.
Given a BNF grammar, this is how to parse a target string using a leftmost derivation:

1. Find the start symbol of the grammar, which is usually the first LHS nonterminal in the set of grammar rules. The RHS of this rule is the initial sentential form (sentence) of the derivation. (We call it a sentential form because it may still have some token names or nonterminal symbols in it. Once all of these have been removed, using the steps given below, the result will be a sentence.) Set the unmatched string to the target string.

2. Repeat Step 3 until the sentential form is a sentence – i.e., consists only of tokens (no nonterminals, no token names).

3. (a) If the leftmost unmatched term in the sentential form is a token name, match it with the leftmost token in the unmatched string. [Here, match means that the token name describes exactly the part of the string to be matched. For example, the token name LPAREN matches the token " ( ", and NUM matches the token "42". The term lexical analysis refers to this process.] If there is no match, exit this algorithm with a parse failure. If there is a match, replace the leftmost token name in the sentential form with its matching token and remove the matched token from the unmatched string.

(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 RHS of the chosen rule. [Which rule to choose will depend on finding a rule that is most likely to complete the derivation. For some grammars, there will be only one choice: these grammars are said to be predictive. All of the grammars we will use in this course will be predictive.] If no rule can apply, exit this algorithm with a parse failure.

(c) If the leftmost term in the sentential form is $\epsilon$ – which stands for the empty string – remove it from the sentential form.

4. If the unmatched string is empty, the target string has been successfully parsed. Otherwise, exit this algorithm with a parse failure.
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 \(<\text{lon}>\) ) as the sentential form:

<table>
<thead>
<tr>
<th>sentential form</th>
<th>unmatched tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{lon}&gt;)</td>
<td>( 14 6 )</td>
</tr>
<tr>
<td>⇒ (\text{LPAREN} \ &lt;\text{nums}&gt; \ \text{RPAREN})</td>
<td>( 14 6 )</td>
</tr>
<tr>
<td>⇒ ( &lt;\text{nums}&gt; \ \text{RPAREN} )</td>
<td>14 6 )</td>
</tr>
<tr>
<td>⇒ ( NUM &lt;\text{nums}&gt; \ \text{RPAREN} )</td>
<td>14 6 )</td>
</tr>
<tr>
<td>⇒ ( 14 &lt;\text{nums}&gt; \ \text{RPAREN} )</td>
<td>6 )</td>
</tr>
<tr>
<td>⇒ ( 14 NUM &lt;\text{nums}&gt; \ \text{RPAREN} )</td>
<td>6 )</td>
</tr>
<tr>
<td>⇒ ( 14 6 &lt;\text{nums}&gt; \ \text{RPAREN} )</td>
<td>)</td>
</tr>
<tr>
<td>⇒ ( 14 6 ɛ \ \text{RPAREN} )</td>
<td>)</td>
</tr>
<tr>
<td>⇒ ( 14 6 \ \text{RPAREN} )</td>
<td>)</td>
</tr>
<tr>
<td>⇒ ( 14 6 )</td>
<td>done!</td>
</tr>
</tbody>
</table>

In the above derivation, the leftmost unmatched token name or nonterminal is shown in **boldface**.

A derivation ends when there are no token names or nonterminals in the sentential form and no unmatched tokens.
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 tokens
<lon>                               ( 14 ( 6 )
⇒ LPAREN <nums> RPAREN              ( 14 ( 6 )
⇒ ( <nums> RPAREN                  14 ( 6 )
⇒ ( NUM <nums> RPAREN              14 ( 6 )
⇒ ( 14 <nums> RPAREN               ( 6 )
⇒ ( 14 NUM <nums> RPAREN           ?? NUM doesn’t match " ( "
⇒ ( 14 e RPAREN                   ( 6 )
⇒ ( 14 RPAREN                     ?? RPAREN doesn’t match " ( "
```

Derivation fails!

We can conclude that the string ‘( 14 ( 6 ) )’ does not conform to the grammar specifications, so it is not a “list of numbers”.
Data Specification

Consider a definition of a nested lists of symbols:

\[
\begin{align*}
\texttt{<slist>} &::= \texttt{LPAREN <sexp> RPAREN} \\
\texttt{<sexp>} &::= \texttt{<slist> <sexp>} \\
\texttt{<sexp>} &::= \texttt{SYMBOL <sexp>} \\
\texttt{<sexp>} &::= \\
\end{align*}
\]

Assume that the token name \texttt{SYMBOL} matches any nonempty string of lowercase letters.

Here is a partial parse of \texttt{(a lst (with (nested) stuff))}, again with the leftmost token name or nonterminal displayed in boldface:

\[
\begin{align*}
\texttt{<slist>} &\Rightarrow \texttt{LPAREN <sexp> RPAREN} \\
&\Rightarrow ( \texttt{<sexp> RPAREN} \\
&\Rightarrow ( \texttt{SYMBOL <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a SYMBOL <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst <slist> <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst LPAREN <sexp> RPAREN <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst ( <sexp> RPAREN <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst ( SYMBOL <sexp> RPAREN <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst ( with <sexp> RPAREN <sexp> RPAREN} \\
&\Rightarrow ( \texttt{a lst ( with <slist> <sexp> RPAREN <sexp> RPAREN} \\
&\Rightarrow \texttt{...}
\end{align*}
\]
Data Specification (continued)

Other data types: trees

\[
\langle \text{tree} \rangle ::= \text{NUMBER} \\
\langle \text{tree} \rangle ::= \text{LPAREN} \text{SYMBOL} \langle \text{tree} \rangle \langle \text{tree} \rangle \text{RPAREN}
\]

Examples of trees:

\[
3 \\
( \text{bar} 1 ( \text{foo} 1 2 ) ) \\
( \text{bar} ( \text{biz} 3 4 ) ( \text{foo} 1 2 ) )
\]

Constructing strings that belong to the language

Instead of starting with a target string and attempting to parse the target, we can use the grammar rules to construct examples of strings that belong to the language. To do so, start with the grammar start symbol to get an initial sentential form and repeatedly replace a nonterminal (usually leftmost) in this sentential form with one of the corresponding RH sides having the nonterminal as its LHS, and replace a token name (usually leftmost) in this sentential form with a corresponding string. Do this until the sentential form has no remaining nonterminals or token names.

For example, we can replace \text{LPAREN} with “(”; replace \text{NUMBER} with 47, or replace \text{SYMBOL} with \text{xyz}.
Data Specification (continued)

BNF rules are *context-free*: this means that for a nonterminal `<foo>`, *any* right-hand side of a rule having `<foo>` as its left-hand side can replace `<foo>` in a construction.

Example construction of a tree:

```
<tree>
  ⇒ LPAREN SYMBOL <tree> <tree> RPAREN
  ⇒ ( bar <tree> <tree> RPAREN                 [two construction steps collapsed into one]
  ⇒ ( bar NUMBER <tree> RPAREN
  ⇒ ( bar 1 <tree> RPAREN
  ⇒ ( bar 1 LPAREN SYMBOL <tree> <tree> RPAREN RPAREN
  ⇒ ...
```
Recursively Specified Programs

Recall our grammar for list of numbers:

\[
\begin{align*}
<\mathit{lon}> & := \text{LPAREN } <\mathit{nums}> \text{ RPAREN} \\
<\mathit{nums}> & := \text{NUM } <\mathit{nums}> \\
<\mathit{nums}> & := \\
\end{align*}
\]

How can we write methods to operate on a list of numbers as defined by this grammar? In order to do so, we will need to represent a list of numbers in a way that a program can process. Instead of using an \textit{ad hoc} approach that might not generalize to other grammars, we will use \texttt{plcc}. \texttt{plcc} takes a computer-readable description of the grammar and produces Java classes that can be used to create objects representing lists of numbers.
Recursively Specified Programs (continued)

Here is a text file representation of this grammar suitable for processing by plcc:

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

These grammar rules look almost identical to the ones at the top of the page, except it has comments (that begin with the # symbol), and the duplicate nonterminals on the LHS are annotated with Java class names which we have called NumsNode, and NumsNull.
The top lines of the file, up to the first percent (%) line, is called 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.
Recursively Specified Programs (continued)

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

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, you can 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
  ...
```
Recursively Specified Programs (continued)

\[
<\text{lon}> ::= \text{LPAREN } <\text{nums}> \text{ RPAREN}
<\text{nums}> : \text{NumsNode} ::= <\text{NUM}> <\text{nums}>
<\text{nums}> : \text{NumsNull} ::= \\
\]

The above text shows the grammar rules in the grammar file.

In your LON directory, change to a subdirectory named Java. This subdirectory was created and populated by the plccmk command. 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 \(<\text{lon}>\) 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 \(<\text{nums}>\) nonterminal appears on more than one grammar rule lines.

Every grammar rule line has a corresponding Java class that is uniquely associated with the grammar rule. In this example, the Java classes are Lon, NumsNode, and NumsNull.
Recursively Specified Programs (continued)

A list of numbers can now be created as an instance of the Lon class. The Lon class constructor takes a single parameter of type Nums, which must be an instance of one of its derived classes: NumsNode or NumsNull.

The NumsNode class constructor takes two parameters: the first of type Token (corresponding to the <NUM> item in the second grammar rule line) and the second of type Nums (corresponding to the <nums> item in the second grammar rule line). The NumsNull constructor takes no parameters; it represents the end of a list being parsed.

Here are some examples of lists of numbers that can now be created:

```java
Nums empty = new NumsNull();
new Lon(empty); // the empty list ()
new Lon(new NumsNode("12", empty)); // the list (12)
new Lon(new NumsNode("12", new NumsNode("6", empty))); // (12, 6)
```

To simplify things, this code uses an alternative NumsNode constructor that takes a String instead of a Token as its first parameter. This constructor is added to the NumsNode class in the grammar file.

The LON subdirectory of the class Code directory has a grammar file that you should examine. The Test.java program is included in the grammar file; this program is used to construct examples of Lon objects and to print them. Here is an example:

```java
public class Test {
    public static void main(String [] args) {
        Nums empty = new NumsNull();
        Lon lon1 = new Lon(empty);
        System.out.println(lon1);
        Lon lon2 = new Lon(new NumsNode("12", empty));
        System.out.println(lon2);
        Lon lon3 = new Lon(new NumsNode("12", new NumsNode("6", empty)));
        System.out.println(lon3);
    }
}
```
Recursively Specified Programs (continued)

When you compile and run Test.java, you will get output that will look something like this:

Lon@42e816
Lon@9304b1
Lon@190d11

The problem is that the classes Lon, Nums, NumsNode, NumsNull do not have toString methods defined. (Examine the Java files for these class to see that the toString methods are missing.) So let’s fix things.

Go up a level to your LON directory and edit the grammar file. At the bottom of the grammar file we are going to add code that will be appended (by the plcc program) to each of the Java files for Lon, Nums, etc., and that will implement the toString method for each of these classes. For example, this is what you should enter for Lon:

 Lon
  {%{public String toString() {
      return "( " + nums.toString() + ")";
  }
%} }

If you look at the Java/Lon.java file, you will see a field named nums of type Nums. So this toString method simply prints the string value of nums surrounded by parentheses.

The ‘%{%’ and ‘%}’ lines delimit the lines of code that will be added to the original Lon.java source file. If you make the above change, run plccmk, and then take a look at the Java/Lon.java file, you will see that this file now has a toString method.
Recursively Specified Programs (continued)

Now when you run Test.java, you will get output that will look something like this:

    ( NumsNull@42e816)
    ( Nums@9304b1)
    ( Nums@190d11)

The parentheses are there, but not the numbers.

Of course, to fix this you will need to write toString methods for the other classes: NumsNode, and NumsNull. This is the rest of the code you will need to add to the grammar file:

```java
NumsNode
{public String toString() {
    return num + " " + nums;
}
}

NumsNull
{public String toString() {
    return "";
}
}
```
Recursively Specified Programs (continued)

Instead of laboriously creating complex instances of Lon using constructors, as we have shown in the Test.java program, we would like to automate the process. Fortunately, the plcc program creates the appropriate Java programs to carry this out.

To see this, be sure you are in your Java subdirectory that has the Test.java program. Examine the Lon.java program again, but this time look at the static parse method with the Scan parameter. This method reads tokens from the Scan scn object (this is not the same as the java.util.Scanner class) and walks through the RHS of the <lon> grammar rule, matching the parentheses tokens and creating a Nums object by calling the parse method on the Nums class. It then creates and returns the appropriate Lon object by calling its constructor.

The file Rep.java grabs input from System.in, sets up a Scan tokenizer, and enters a loop that: (1) prints a prompt and waits for user input; (2) parses the user input into a Lon object; and (3) prints the Lon object in String format. The file is called Rep because it Reads user input, Evaluates it by parsing it, and Prints the result.

Now when you run Rep, you can enter input strings that conform to the grammar rules, and the parser will create Lon objects (just like we had to do by hand) on the fly and print them. If you enter a syntactically incorrect input, the parser will complain.

Try this with the following input strings:

()  
(12)  
(12 6)
The Rep program creates an object of type Lon in its read-eval-print loop. Suppose we want to print the *length* of a Lon object instead of just printing a copy of the object. For example, if the Lon object is obtained by parsing the string "(42 3 15)", we would want to print the length of 3.

So let’s define a length method that, when applied to a Lon object, returns the number of items (NUMs) in the list.

A Lon object is parsed from the RHS of the grammar rule

\[
\text{<lon>} ::= \text{LPAREN} \text{<nums>} \text{RPAREN}
\]

so a Lon object has an instance variable named nums which by default references an object of type Nums (uppercase the first letter of nums to get Nums). To implement a length method for the Lon class, all we need to do is to return the length of the Nums object referred to by nums:

```java
public int length() {
    return nums.length();
}
```
The `Nums` abstract class is extended by both `NumsNode` and `NumsNull` objects. From the grammar specification lines

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

(or looking at the Java files directly) you can see that a `NumsNull` object has no instance variables but that `NumsNode` object has two instance variables: a `num` variable which refers to a `Token` object (lowercase all of the letters in the token name `NUM`), and a `nums` variable which refers to a `Nums` object. The length of a `NumsNull` object is zero, but the length of a `NumsNode` object is equal to one (for the `num`) plus the length of the object referred to (recursively) by `nums`. (A `NumsNode` object is essentially a node in a linked list, where the `num` variable represents the data corresponding to this node and the `nums` variable is a link to the next item in the list.)
We need to define a length method in both the NumsNull class and the NumsNode class. For NumsNull, we define

```java
public int length() {
    return 0;
}
```

and for NumsNode, we define

```java
public int length() {
    return 1 + nums.length();
}
```

In the Lon class, if we define toString() to return "length=\+nums.length()", we should expect the following results from running Rep:

- () → length=0
- (42 3) → length=2
- (15) → length=1
Recursively Specified Programs (continued)

In order to get things to compile, we need to declare a `length` method in the `Nums` abstract class. (We never construct a `Nums` object directly, but only by constructing instances of `NumsNode` and `NumsNull`.) We do this in the semantics section of the `grammar` file, as shown here:

```plaintext
Nums
%%{
    public abstract int length();
%%}
```

This is a declaration, not a definition, because it does not have any code associated with it.

The actual definitions of the `length` methods for each of the `NumsNode` and `NumsNull` classes should appear in the `grammar` file, with the name of the class appearing on one line and the code for the methods appearing between `%%{` and `%%%` on subsequent lines. For example, with the `NumsNode` class, you would add the following lines to your `grammar` file:

```plaintext
NumsNode
%%{
    public int length() {
        return 1 + nums.length();
    }
%%}
```

The order in which these sections of code appear in the `grammar` file is not important.
Recursively Specified Programs (continued)

You should be able to see how to define a method `sum` that returns the sum of the numbers in a list of numbers and to make `Rep` print the sum of the list instead of the length of the list. Basically, it would be defined much the same as `length`, except that `sum` would take the `num` variable (it’s a `Token`), convert the `String` representation of this to an integer using `Integer.parseInt`, and add it (recursively) to the running sum.
Recursively Specified Programs (continued)

We observed that an object that extends the Nums class essentially represents a linked list of numbers (Tokens, actually, each of whose toString() values represents a decimal number), and processing such an object consists of handling a NumsNull object – the end of the list, or a NumsNode object – a node in the list. So a Nums object is just a repeated list of numbers (zero or more).

As we mentioned briefly before, we can use the Kleene Star Extended BNF (EBNF) construction to represent repetition in a grammar. In our grammar file, we use a special notation to represent repetition in the style of Kleene Star. To simplify things, we only allow the entire RHS of a grammar rule to be repeated in this way. Here is an example using repetition to define a list <nums> of numbers:

\[
<\text{lon}> ::= \text{LPAREN} <\text{nums}> \text{RPAREN} \\
\quad \text{Lon(Nums nums)} \\
<\text{nums}> ::= <\text{NUM}> \\
\quad \text{Nums(List<Token> numList)}
\]

Notice that Nums is now a regular class, not an abstract class that other classes will extend. The nums instance variable in the Lon class still refers to a Nums object. But the Nums class now has a numList instance variable that refers to a List of Tokens. Note that the numList variable name is derived from the NUM token name by converting it to lowercase and appending the string "List".

The Rep program still parses a list of numbers as a Lon object, but the repeated nature of a Nums object is now packaged directly in a Java List. The plcc program handles the creation of the List instance variables automatically when it sees the *= line in the grammar.
We normally process List objects using a loop. For example, with the List<Token> numList variable in the Nums class, the following method returns the sum of the numbers in the List:

```java
Nums
%%{
    public int sum() {
        int a = 0;
        for (Token t : numList) {
            a += Integer.parseInt(t.toString());
        }
        return a;
    }
}%%
```
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 syntactic construct that has a value at run-time. A variable, by itself, is therefore an expression, but other syntactic 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 constructed solely for the purpose of evaluating expressions is called an expression-based language. Many of the languages we will construct will be expression-based. Scheme, ML, and Haskell are examples of expression-based languages used in practice. A programming language that is constructed for the purpose of “doing something” with expressions (such as assigning the value of an expression to a variable or printing the value of an expression to standard output) is called an imperative language. C, Java, and Python are examples of imperative languages used in practice.

Expression-based languages get their power from defining and applying functions, so another term describing such languages is functional.
Static Properties of Variables (continued)

Determining 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 bindings of these variables.

At run-time, how can you find the binding of a variable? There are two basic approaches: if the binding of a variable can be found by code that is created at compile-time, we call it static binding; otherwise 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.
Static Properties of Variables (continued)

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:

```java
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
        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 `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:

```java
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 will be

```
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, which as shown in the program comments. In this case, we say that the definition of `x` in the block puts a *hole* in the scope for the global `int x`. 
Static Properties of Variables (continued)

Consider the following grammar for a language called the Lambda Calculus that has only variable references, procedure definitions with one parameter, and procedure applications (also called “invocations” or “calls”). The Lambda Calculus is of interest theoretically, but it has no practical value as a programming language.

\[
\begin{align*}
\text{<exp>} & \ ::= \ \text{<SYMBOL>} \\
\text{<exp>} & \ ::= \ \text{PROC} \ \text{LPAREN} \ \text{<SYMBOL>} \ \text{RPAREN} \ \text{LBRACK} \ \text{<exp>} \ \text{RBRACK} \\
\text{<exp>} & \ ::= \ \text{DOT} \ \text{<exp>} \ \text{LPAREN} \ \text{<exp>} \ \text{RPAREN}
\end{align*}
\]

Here PROC is the token "proc", LPAREN is the token "(" , and so forth. The SYMBOL token is a string of letters, digits, and underscores, beginning with a letter. This language is essentially the same as a classic language called the Lambda Calculus, but we use slightly different notation. We will, nonetheless, call our simple language the Lambda Calculus. The nonterminal <exp> stands for a Lambda Calculus expression.

Here are some examples of Lambda Calculus expressions, derived from the three rules given above:

\[
\begin{align*}
p \\
\text{proc}(a) \ \{x\} \\
.f(y)
\end{align*}
\]
Consider the following sentential form (remember what that means?) obtained from the second grammar rule, where $s$ replaces $\langle\text{SYMBOL}\rangle$:

$$
\text{proc}(s) \{ \text{<exp>} \}
$$

The occurrence of the symbol $s$ in this expression is called a *variable declaration* that *binds* all occurrences of $s$ that appear in $\langle\text{exp}\rangle$ unless some intervening declaration of the same symbol $s$ occurs in $\langle\text{exp}\rangle$. We say that the expression $\langle\text{exp}\rangle$ is the *scope* of the variable declaration.

### 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.)

- $\text{proc}(x) \{ x \}$ ; $x$ occurs bound
- $\text{proc}(x) \{ y \}$ ; $y$ occurs free
- $\text{.proc}(x) \{ x \} (x)$ ; first $x$ is bound, second is free
- $\text{.proc}(x) \{ x \} (y)$ ; $y$ occurs free
- $\text{proc}(y) \{ \text{.proc}(x) \{ x \} (y) \}$ ; $y$ occurs bound
- $\text{proc}(x) \{ \text{.proc}(y) \{ x \} (y) \}$ ; $y$ occurs free
- $\text{.t (u)}$ ; $t$ and $u$ occur free
Formal definitions of \textit{occurs free} and \textit{occurs bound}:

For a Lambda Calculus expression $E$, a symbol $x$ \textit{occurs free} in $E$ if

- \textit{Rule 1}:
  $E$ is a \texttt{<SYMBOL>} and $E$ is the same as $x$.
  
  \begin{verbatim}
  x ; x is free
  \end{verbatim}

- \textit{Rule 2}:
  $E$ is of the form \texttt{proc(y){E'}} where $y$ is different from $x$ and $x$ occurs free in $E'$
  
  \begin{verbatim}
  proc(y){x} ; x is free
  \end{verbatim}

- \textit{Rule 3}:
  $E$ is of the form $\texttt{.E1(E2)}$ and $x$ occurs free in $E1$ or $E2$
  
  \begin{verbatim}
  proc(y){x}(y) ; x is free
  \end{verbatim}

  \begin{verbatim}
  proc(y){y}(x) ; x is free
  \end{verbatim}
Static Properties of Variables (continued)

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

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

- **Rule 2:**
  $E$ is of the form $\cdot E_1(E_2)$ and $x$ occurs bound in $E_1$ or $E_2$
  
  $\cdot \text{proc}(y)\{\text{proc}(x)\{x\}\}\{y\}\quad ; \quad x \text{ is bound}$
  
  $\cdot \text{proc}(y)\{x\}\{\text{proc}(y)\{y\}\}\quad ; \quad y \text{ is bound}$
Static Properties of Variables (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> RBRACE
<exp>:App ::= DOT <exp>rator LPAREN <exp>rand RPAREN

%
**Static Properties of Variables** (continued)

**Code for occurs free:**

Exp

```java
%%{
    public abstract boolean occursFree(String s);
%%}
```

Var

```java
%%{
    public boolean occursFree(String s) {
        return s.equals(sym.toString());
    }
%%}
```

Proc

```java
%%{
    public boolean occursFree(String s) {
        return !(s.equals(sym.toString())) && exp.occursFree(s);
    }
%%}
```

App

```java
%%{
    public boolean occursFree(String s) {
        return rator.occursFree(s) || rand.occursFree(s);
    }
%%}
```
Static Properties of Variables (continued)

Code for occurs bound:

Exp
{%
    public abstract boolean occursBound(String s);
%
}

Var
{%
    public boolean occursBound(String s) {
        return false;
    }
%
}

Proc
{%
    public boolean occursBound(String s) {
        return exp.occursBound(s) || (s.equals(sym.toString()) && exp.occursFree(s));
    }
%
}

App
{%
    public boolean occursBound(String s) {
        return rator.occursBound(s) || rand.occursBound(s);
    }
%
}
Static Properties of Variables (continued)

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.

We described the following problem earlier, in the context of commonly used programming languages: for a given variable in an expression, to what value is that variable bound? Most modern programming languages are block structured and use lexical binding, which is 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 this declaration is 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:

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

This code prints 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:

```cpp
{ 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 \( y \):

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