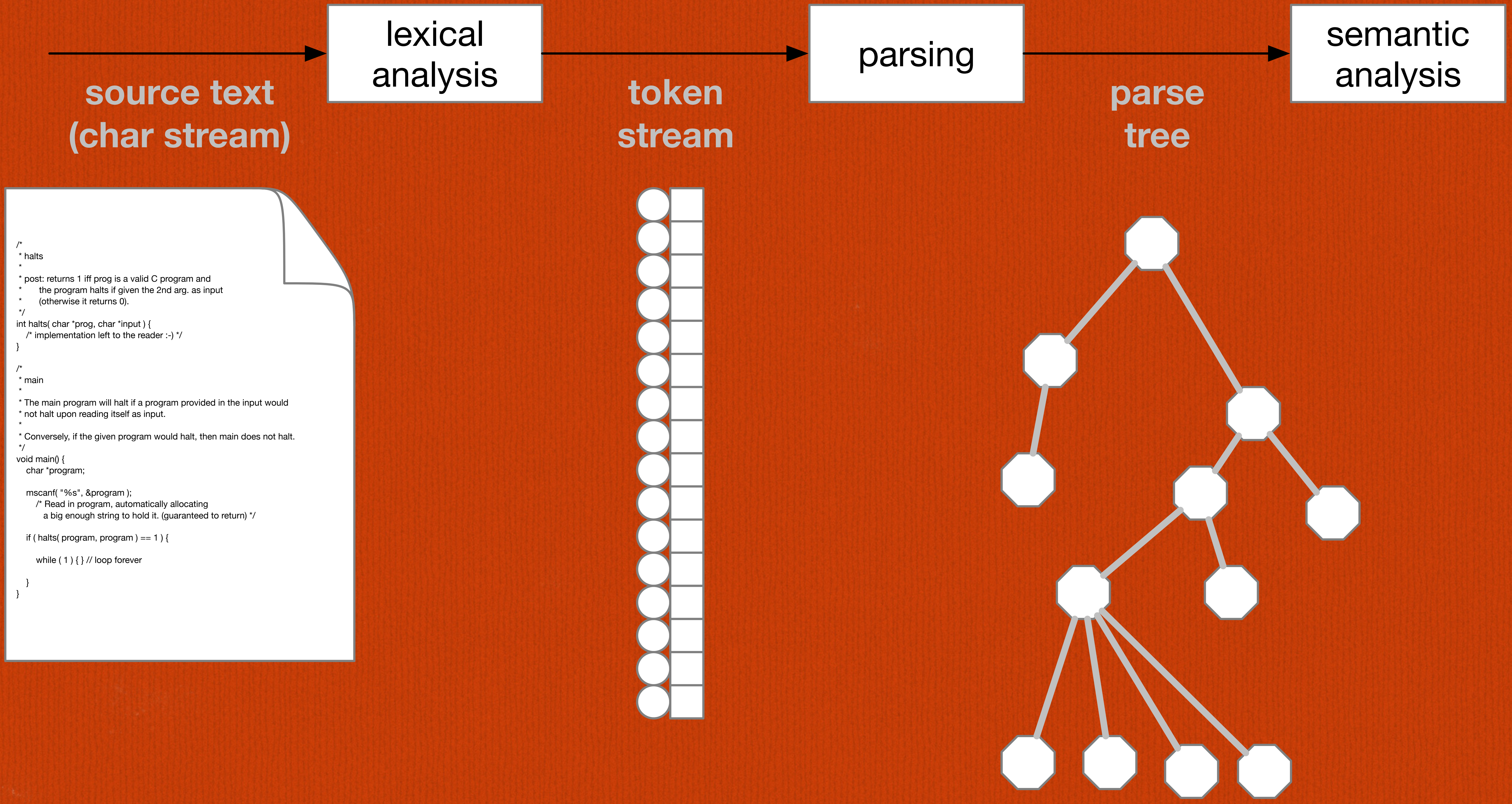
PLCC Overview

CCSCNE 2023 PLCC Workshop

Tim Fossum, Stoney Jackson, Jim Heliotis

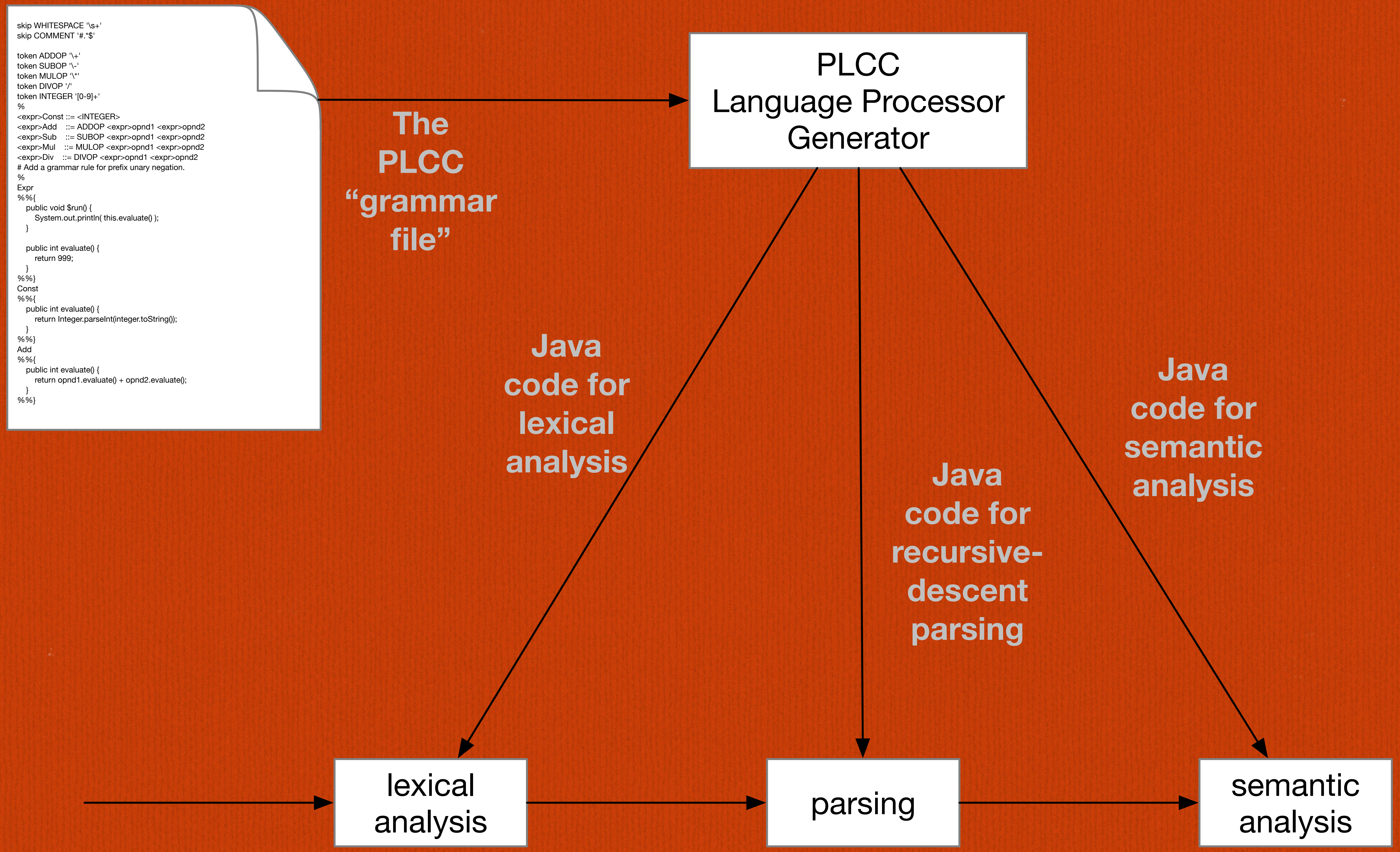
The Language Processing Model



```
/*
 * halts
 *
 * post: returns 1 iff prog is a valid C program and
 *      the program halts if given the 2nd arg. as input
 *      (otherwise it returns 0).
 */
int halts( char *prog, char *input ) {
   /* implementation left to the reader :-) */
}

/*
 * main
 *
 * The main program will halt if a program provided in the input would
 * not halt upon reading itself as input.
 *
 * Conversely, if the given program would halt, then main does not halt.
 */
void main() {
   char *program;

   mscanf( "%s", &program );
      /* Read in program, automatically allocating
         a big enough string to hold it. (guaranteed to return) */

   if ( halts( program, program ) == 1 ) {

      while ( 1 ) { } // loop forever

   }
}
```

Can be
Execution (interpretation)
Byte code
Machine code
Code metrics
Soundness checks
Type checking
Pattern matching

The Language Processing Model



```
skip WHITESPACE '\s+'
skip COMMENT '#.*$'

token ADDOP '\+'
token SUBOP '\-'
token MULOP '\*'
token DIVOP '/'
token INTEGER '[0-9]+'
%
<expr>Const ::= <INTEGER>
<expr>Add    ::= ADDOP <expr>opnd1 <expr>opnd2
<expr>Sub    ::= SUBOP <expr>opnd1 <expr>opnd2
<expr>Mul    ::= MULOP <expr>opnd1 <expr>opnd2
<expr>Div    ::= DIVOP <expr>opnd1 <expr>opnd2
# Add a grammar rule for prefix unary negation.
%
Expr
%%{
  public void $run() {
    System.out.println( this.evaluate() );
  }

  public int evaluate() {
    return 999;
  }
%%}
Const
%%{
  public int evaluate() {
    return Integer.parseInt(integer.toString());
  }
%%}
Add
%%{
  public int evaluate() {
    return opnd1.evaluate() + opnd2.evaluate();
  }
%%}
```

The PLCC “grammar file”
PLCC Language Processor Generator
Java code for lexical analysis
Java code for recursive-descent parsing
Java code for semantic analysis
lexical analysis
parsing
semantic analysis

Break for Tool Setup

- □ At this point the organizers will help you organize yourselves into teams.
- □ Each team will then set up a computer to access the materials of the workshop.
- □ Once you are ready, please create a plain text file **team.txt** that contains the names and emails of the team's members.
    - □ This will also help you get used to the editor you have chosen to use.

PLCC File Syntax

- □ **plcc.py** (via **plcc** and **plccmk scripts**) is used to create language processors.
- □ It reads a file containing a specification of
  1. The tokens of your language
  2. The grammar of your language
  3. Additional support code (Java) to help process the resulting abstract syntax tree
- □ By default the **plcc** expects the file to be named **`grammar`**.
- □ Our files will have distinct names and will end in "**`.plcc`**".

Example: Parenthesized Number Lists

- In these notes we will fashion a language processor that reads expressions of the form

  (*integer integer* …)

  and echoes them back out, with any extra white space removed.
- The processor will be developed in three stages.
  - Lexical scanning stage
  - Parsing stage
  - Semantic stage

(Quick look at result)

"rep": read-evaluate-print loop

```
$ rep
--> (1 2 3)
( 1 2 3 )
--> (23         59          )
( 23 59 )
--> (8)
( 8 )
--> ()
( )
--> ^D
```

control-D: UNIX end-of-file

Team Exercises

- Teams will be assigned separate exercises to complete for each stage.

Section 1 of 3: Language Tokens

□ Below is what we specify for the language processor’s lexical scanner.

```
skip WHITESPACE '\s+'

token LPAREN '\('

token RPAREN '\)'

token NUMBER '\d+'
```

How to Generate the System and Run the System

```
    skip WHITESPACE '\s+'

    token LPAREN '\('

    token RPAREN '\)'

    token NUMBER '\d+'
```

For `rep`*, the parse tree root node is displayed if the parse is successful.*

Note that there was no `toString` *method defined for* `NumSeq`*.*

```
$ plccmk -c numlistA.plcc
: (Some info prints here.)
$ scan
65
   1: NUMBER '65'
(
   2: LPAREN '('
)
   3: RPAREN ')'
)
   4: RPAREN ')'
hio
   5: !ERROR("h")
   5: !ERROR("i")
   5: !ERROR("o")
^D
$
```

Activity 1

□ Each activity is in a separate directory and contains an **assignment.txt** file, some starter code in a **plcc** file, and test input.

□ "**cd**" to the **Activity1** directory.

□ You are to add some additional tokens to the lexical specification.

□ Be aware that token matches are attempted in the order they appear in the specification!

1/3

Section 2 of 3: Grammar (LL(1))

□ Grammar rules are similar to BNF.

□ They are built from two kinds of elements:

  □ Terminals/tokens, and

  □ Non-terminals/variables/rule-names.

<name> ::= *token <name> <token> <name>*

□ Angle brackets are used to mandate storage of the element's information and structure in the parse tree.

2/3

Section 2 of 3: Grammar (LL(1))

- □ *Terminal*: an identifier defined in the token section (section 1)
    - □ possibly surrounded by < … >, if the token's string name is needed
- □ Non-terminal, left hand side – rule name: *<name>*
    - □ *<name>*:*sub_name* if there are multiple rules for *name*
- □ A non-terminal on the right-hand side refers to another rule: *<name>*
    - □ *<name>alt* - required if *<name>* appears multiple times on a single rule's right-hand side (disambiguation in code)

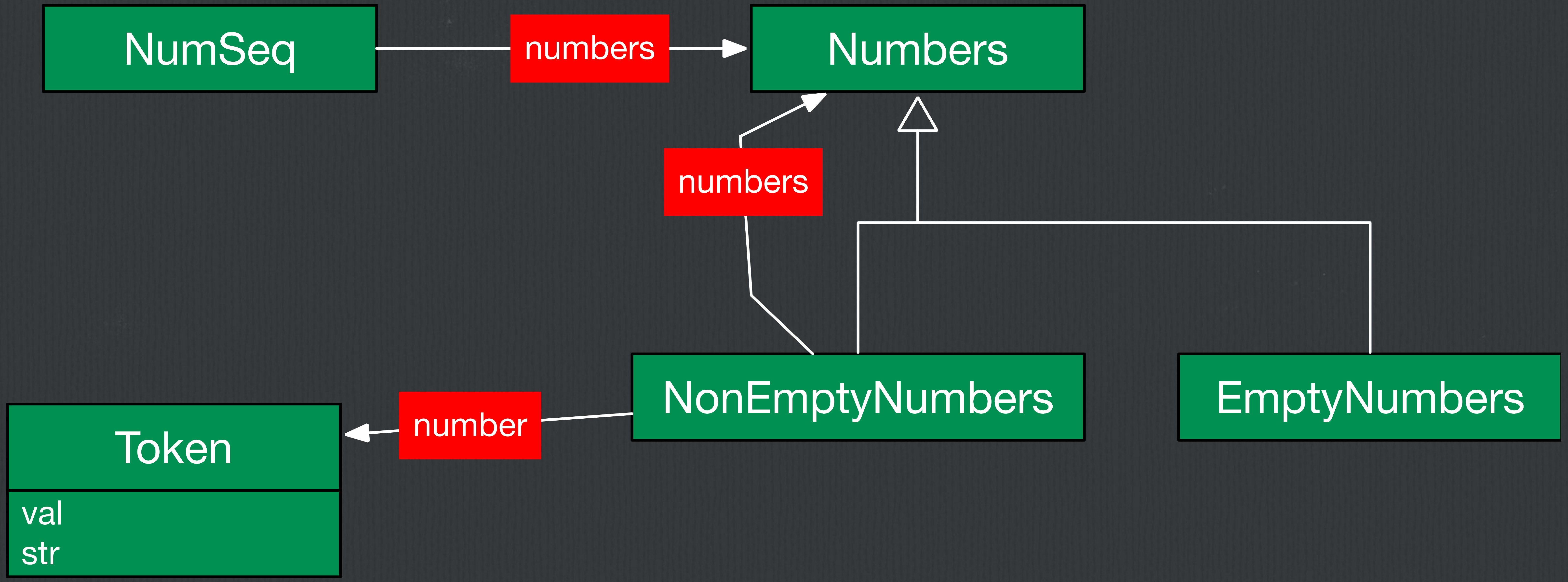
3/3

Section 2 of 3: Grammar (LL(1))

```
# Tokens
skip WHITESPACE '\s+'
token NUMBER '\d+'
token LPAREN '\('
token RPAREN '\)'
%
# BNF
<numSeq> ::= LPAREN <numbers> RPAREN
<numbers>:NonEmptyNumbers ::= <NUMBER> <numbers>
<numbers>:EmptyNumbers ::=
%
```

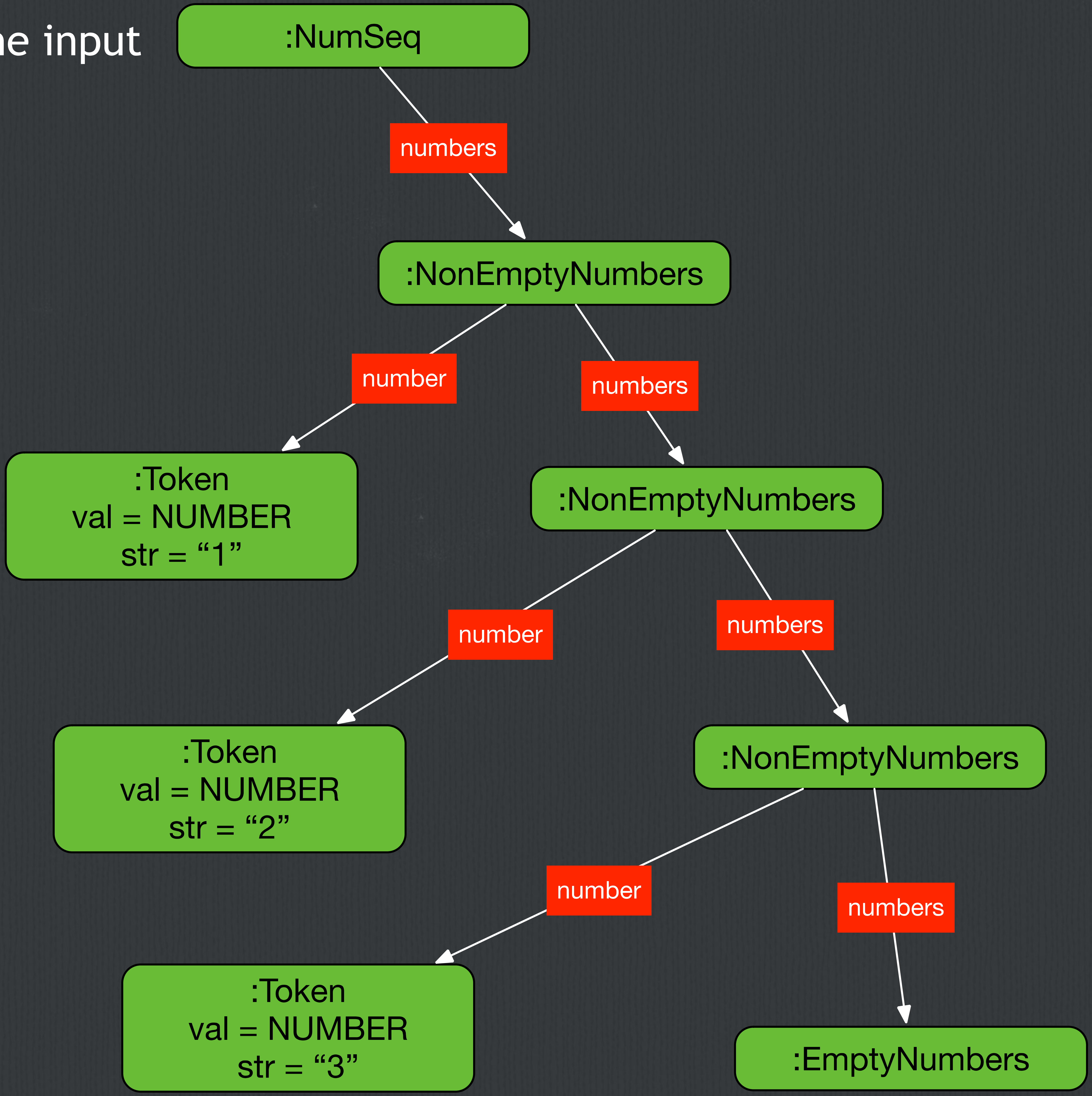
Java Class Correspondence

```
<numSeq> ::= LPAREN <numbers> RPAREN
<numbers>:NonEmptyNumbers ::= <NUMBER> <numbers>
<numbers>:EmptyNumbers ::=
```



The parse tree built from the input

(1 2 3)



How to Generate the System and Run the System

```
<numSeq> ::= LPAREN <numbers> RPAREN
<numbers>:NonEmptyNumbers ::= <NUMBER> <numbers>
<numbers>:EmptyNumbers ::=
```

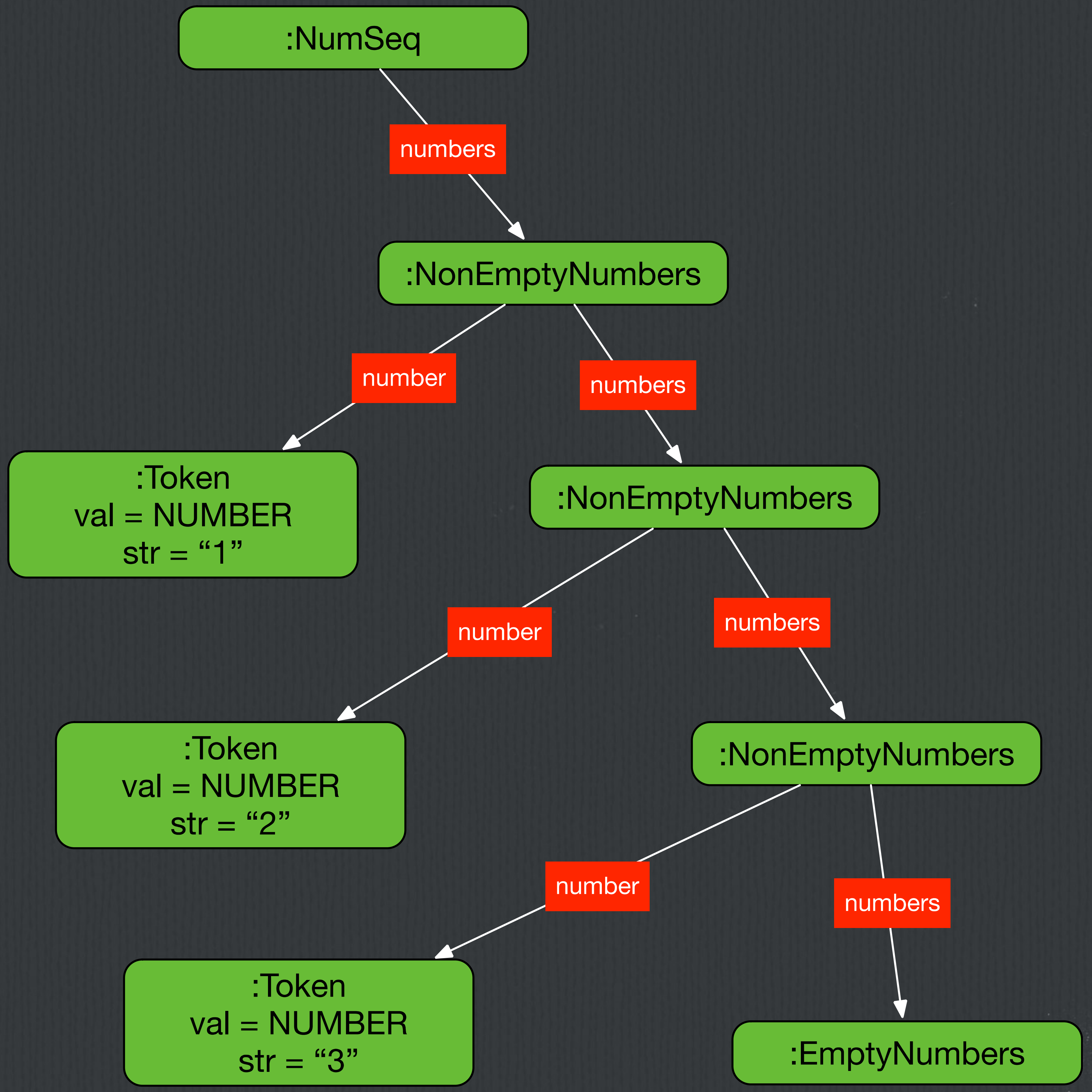
For `rep`*, the parse tree root node is displayed if the parse is successful.*

Note that there was no `toString` *method defined for* `NumSeq`*.*

```
$ plccmk -c numlistAB-rec.plcc
: (Some info prints here.)
$ parse
( 1 2   3)
OK
--> ^D
$ rep
--> ( 1 2 3 )
NumSeq@372f7a8d
--> ()
NumSeq@2f92e0f4
--> (456)
NumSeq@28a418fc
--> (1(2 3))
%%% Parse error: expected token
RPAREN, got LPAREN
$
```

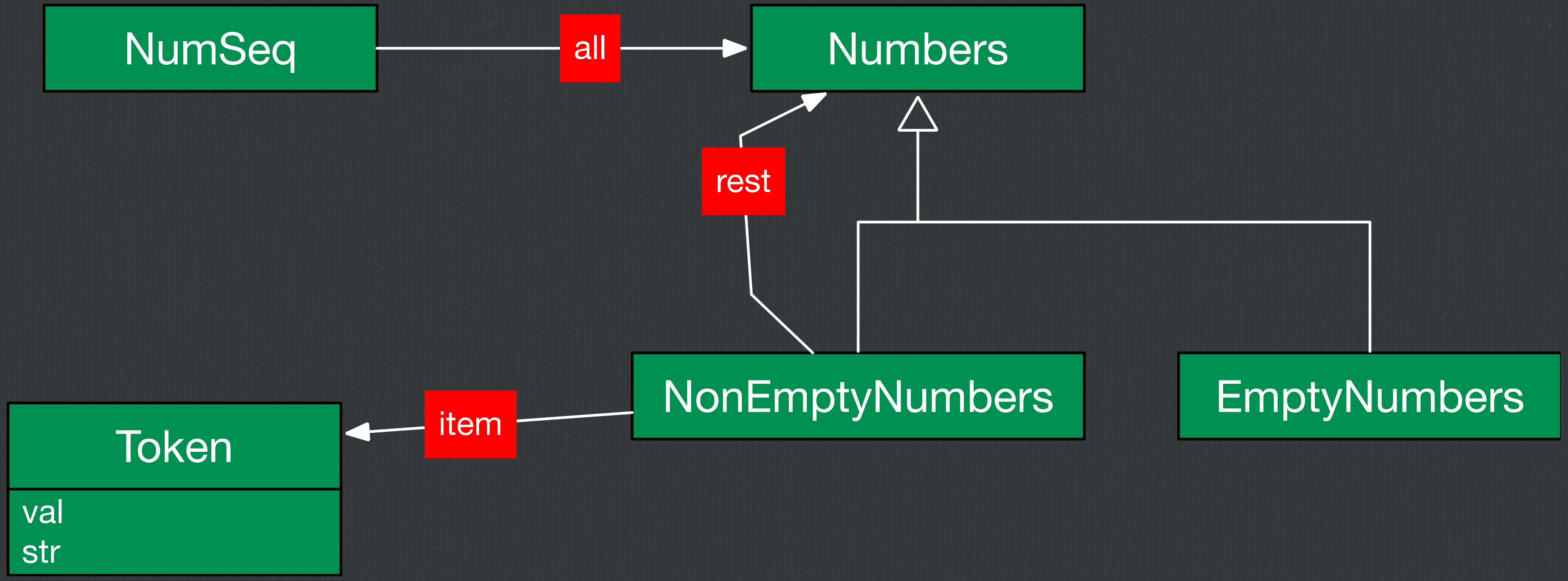
Executing the parse Command with Tracing

```
$ parse -t
--> (1 2 3)
   1: <numSeq>
   1: | LPAREN "("
   1: | <numbers>:NonEmptyNumbers
   1: | | NUMBER "1"
   1: | | <numbers>:NonEmptyNumbers
   1: | | | NUMBER "2"
   1: | | | <numbers>:NonEmptyNumbers
   1: | | | | NUMBER "3"
   1: | | | | <numbers>:EmptyNumbers
   1: | RPAREN ")"
OK
--> ^D
$
```



RHS Variable Renaming (not necessary for this grammar)

```
<numSeq> ::= LPAREN <numbers>all RPAREN
<numbers>:NonEmptyNumbers ::= <NUMBER>item <numbers>rest
<numbers>:EmptyNumbers ::=
```



Activity 2

□ "**cd**" to the **Activity2** directory.

□ You will study the classic *dangling else* problem.

□ It involves modifying a grammar, and the language it models, to fix the problem.

Section 3 of 3: Defining Semantics

```
    :
    :
§2  <numSeq> ::= LPAREN <numbers> RPAREN
    <numbers>   **= <NUMBER>
    %
§3  NumSeq
    %%{
        @Override
        public void $run() {
            System.out.println(
               "This is the root." );
        }
    %%}
```

- After parsing, the run-time system will call `$run()` on the root node of the parse tree.

I will now add features
To the (root) class NumSeq.

Here is a new method.

- If writing a new class, its <u>entire</u> definition should be put in section 3. (next slide)

Syntax of Semantics Section

- Adding Elements to an Existing Class (generated by PLCC from the grammar)

```
Expression
%%{
  public int cachedValue;
  public abstract int evaluate();
%%}
```

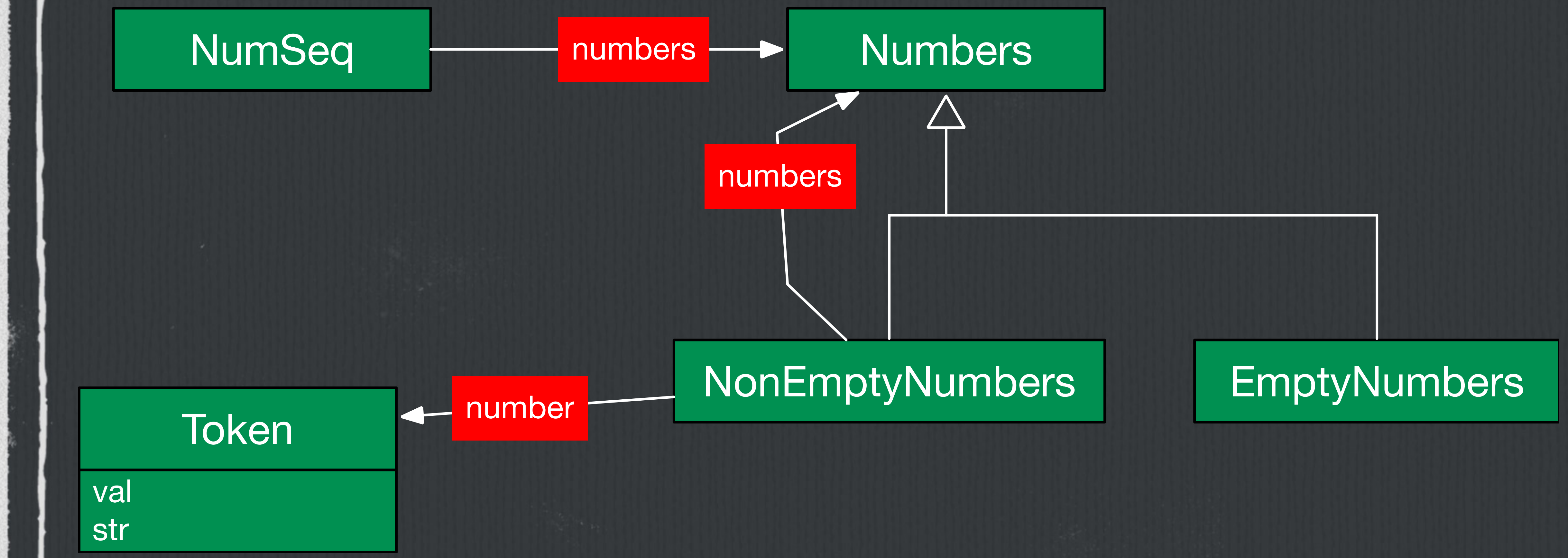
- Adding a new Class

```
Binding
%%{
  public class Binding {
    public String name;
    public Val value;
  }
%%}
```

(Packages not supported.)

Defining Semantics to Echo the List

```
NumSeq
%%{

    @Override
    public void $run() {
        System.out.println(
            "(" + numbers + ")" );
    }
%%}
```



```
NonEmptyNumbers
%%{
    @Override
    public String toString() {
        return " " + number.str + numbers;
    }
%%}

EmptyNumbers
%%{
    @Override
    public String toString() {
        return " ";
    }
%%}
```

How to Generate and Run the New System

```
$ plccmk -c numlistABC-rec.plcc
Nonterminals (* indicates start symbol):
 *<numSeq>
  <numbers>

Abstract classes:
  Numbers

Java source files created:
  NumSeq.java
  Numbers.java
  NonEmptyNumbers.java
  EmptyNumbers.java
$ rep
--> (1 2 3)
( 1 2 3 )
--> (23         59           )
( 23 59 )
--> (8)
( 8 )
--> ()
( )
--> ^D
```

Rep Tool with Tracing Option

```
$ rep -t
--> (1 2 3)
   1: <numSeq>
   1: | LPAREN "("
   1: | <numbers>:NonEmptyNumbers
   1: | | NUMBER "1"
   1: | | <numbers>:NonEmptyNumbers
   1: | | | NUMBER "2"
   1: | | | <numbers>:NonEmptyNumbers
   1: | | | | NUMBER "3"
   1: | | | | <numbers>:EmptyNumbers
   1: | RPAREN ")"
( 1 2 3 )
```

Exercise 3

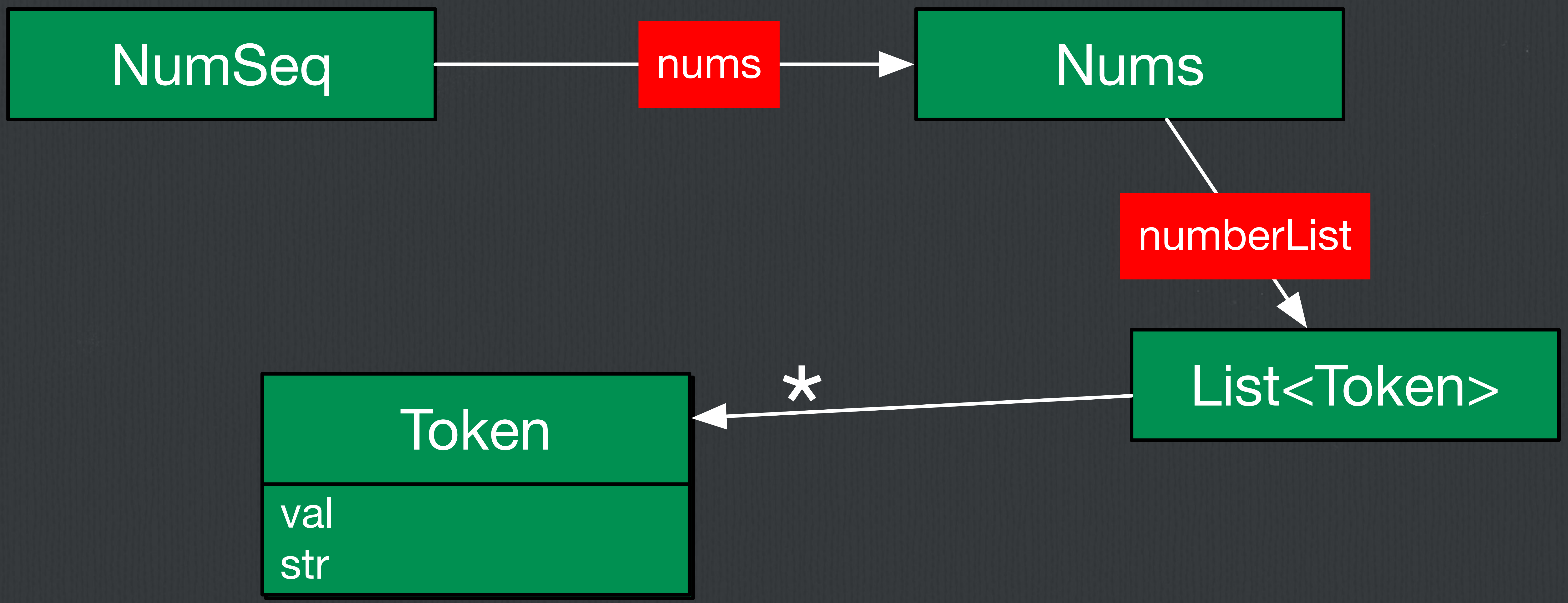
□ "**cd**" to the **Activity3** directory.

□ There is a grammar there that recognizes a prefix expression.

□ You will complete the semantics section, which evaluates the expression.

PLCC Alternative: Repeating Rule (employs Java Lists)

```
<numSeq> ::= LPAREN <nums> RPAREN
<nums> **= <NUMBER>
```



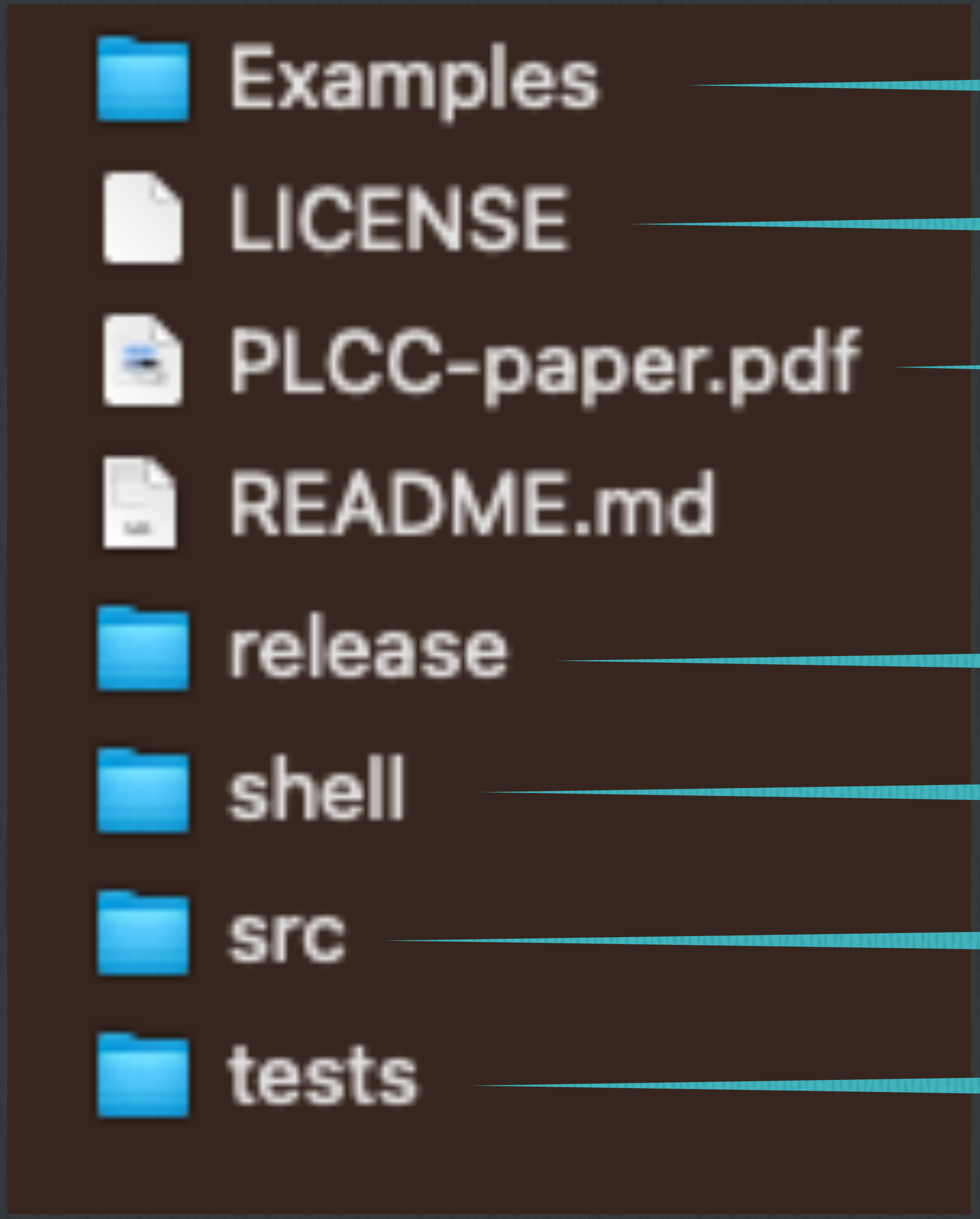
Rep-t on Iterative Version

```
$ plccmk -c numlistABC-rep.plcc
:
:
$ rep -t
--> (1 2 3)
   1: <numSeq>
   1: | LPAREN "("
   1: | <numbers>
   1: | | NUMBER "1"
   1: | | NUMBER "2"
   1: | | NUMBER "3"
   1: | RPAREN ")"
( 1 2 3 )
```

Exercise 4

- This is a complete system that you can study and run
- "**cd**" to the **Activity4** directory.
- There is a grammar and semantics there for processing a post fix expression.
- This is a bit more difficult.
  - The order of tokens is a bit less predictable.
  - The end of the expression is unknown.

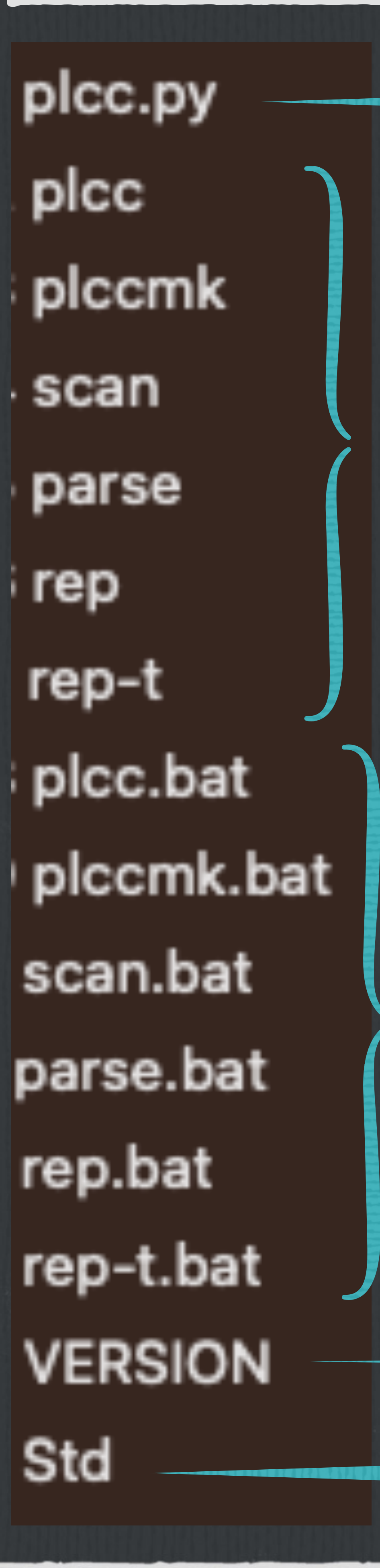
Contents of the PLCC Package



Examples	A set of a few basic plcc grammar files
LICENSE	GNU General Public Licence
PLCC-paper.pdf	Original 2014 paper
README.md	
release	Tool for release management
shell	Docker-based development shell
src	The Python source and executable scripts
tests	For validation of new releases (WIP)

The PLCC Package `src` Directory

This should be the value of the `LIBPLCC` environment variable.



```
plcc.py
plcc
plccmk
scan
parse
rep
rep-t
plcc.bat
plccmk.bat
scan.bat
parse.bat
rep.bat
rep-t.bat
VERSION
Std
```

plcc.py: The Python program that processes PLCC grammar files

plcc, plccmk, scan, parse, rep, rep-t: Script files for UNIX™-based systems

plcc.bat, plccmk.bat, scan.bat, parse.bat, rep.bat, rep-t.bat: Batch files for Windows™ systems

VERSION: contains version number in plain text

Std: Run-time support classes, plus class templates used by `plcc.py`

Features Not Covered Today

- The repetition rule
  - You may include an element that separates those that are being repeated:

```
<numList> **= <NUMBER> +COMMA
```

- Complete Java classes can also be added to the semantics section.
  - See Environment code example.
- There is an `include` directive that can be used in the semantics section to allow placement of Java code in separate files.

Details of PLCC's Grammar Rules

- *lhs ::= term...*
- *lhs:*
  - *<id>*
  - *<id>:sub_id*
- *term:*
  - *<id>*
  - *<id>id2*
  - *TOKEN*
  - *<TOKEN>*
  - *<TOKEN>id2*

➛The syntax of each rule

➛Left-hand side can be

- ➛a class name, which is defined by this rule
- ➛a class and a subclass name, the latter of which is defined by this rule (required when class is used > once on LHS)

➛Each term in the sequence on the right side can be

- ➛a class name whose identifier has the name = *id*
- ➛a class name named *id2* with an identifier name = *id2* for the instance
- ➛a token name, which is not saved in a variable
- ➛a token name, which is saved as an instance of class `Token` and has the name = *token*
- ➛a token name, which is saved as an instance of `Token` but has the name = *id2*

id string begins with lower case in the grammar but its class's name begins with upper case.