CCSC
CONSORTIUM FOR COMPUTING SCIENCES IN COLLEGES
ccsc.org

CCSC-NE 2025
State University of NY at Brockport

PLCC Overview

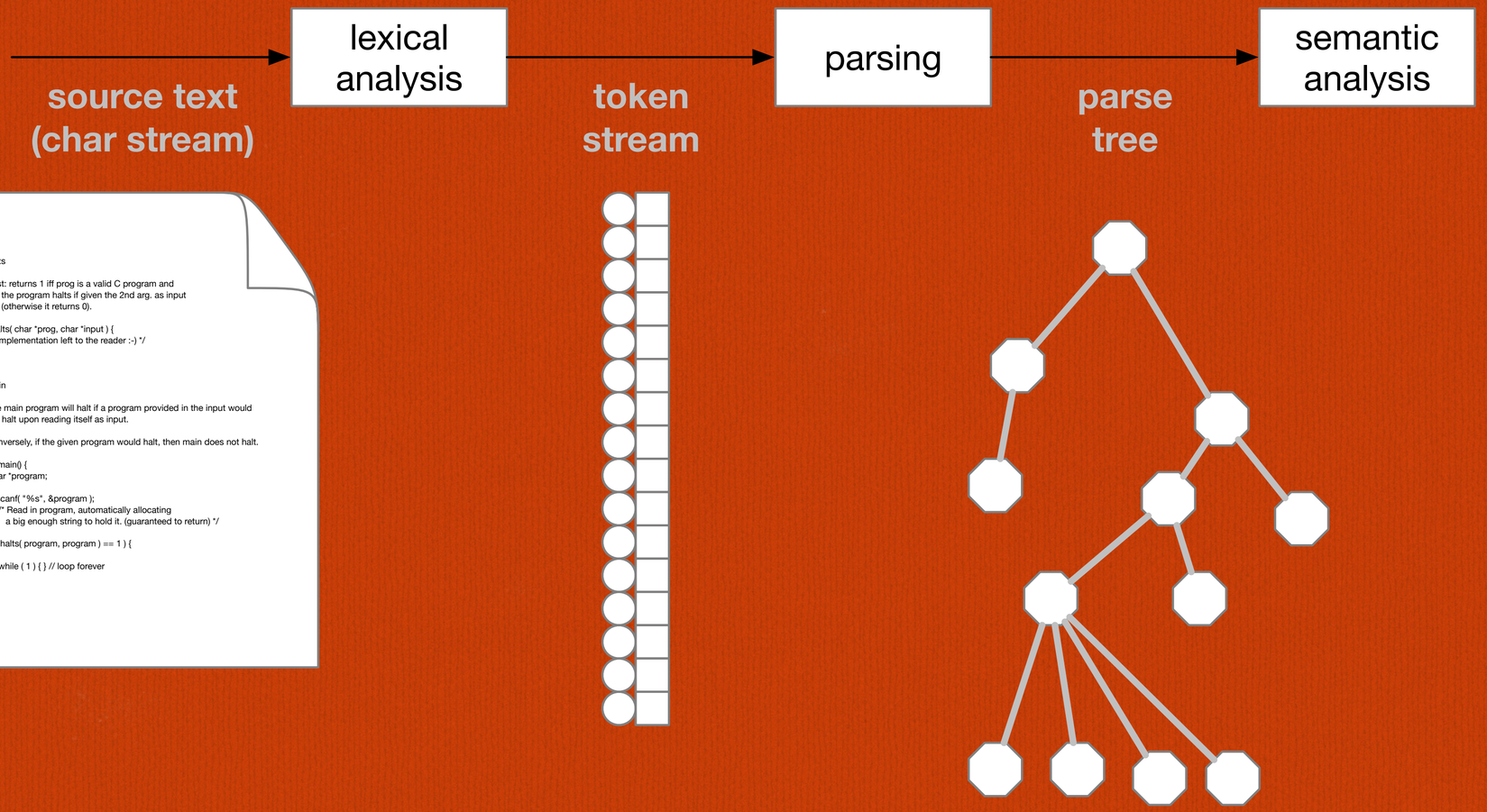
CCSCNE 2025 PLCC Workshop
Stoney Jackson, Jim Heliotis
Tim Fossum

WNE
WESTERN NEW ENGLAND UNIVERSITY

Potsdam
STATE UNIVERSITY OF NEW YORK

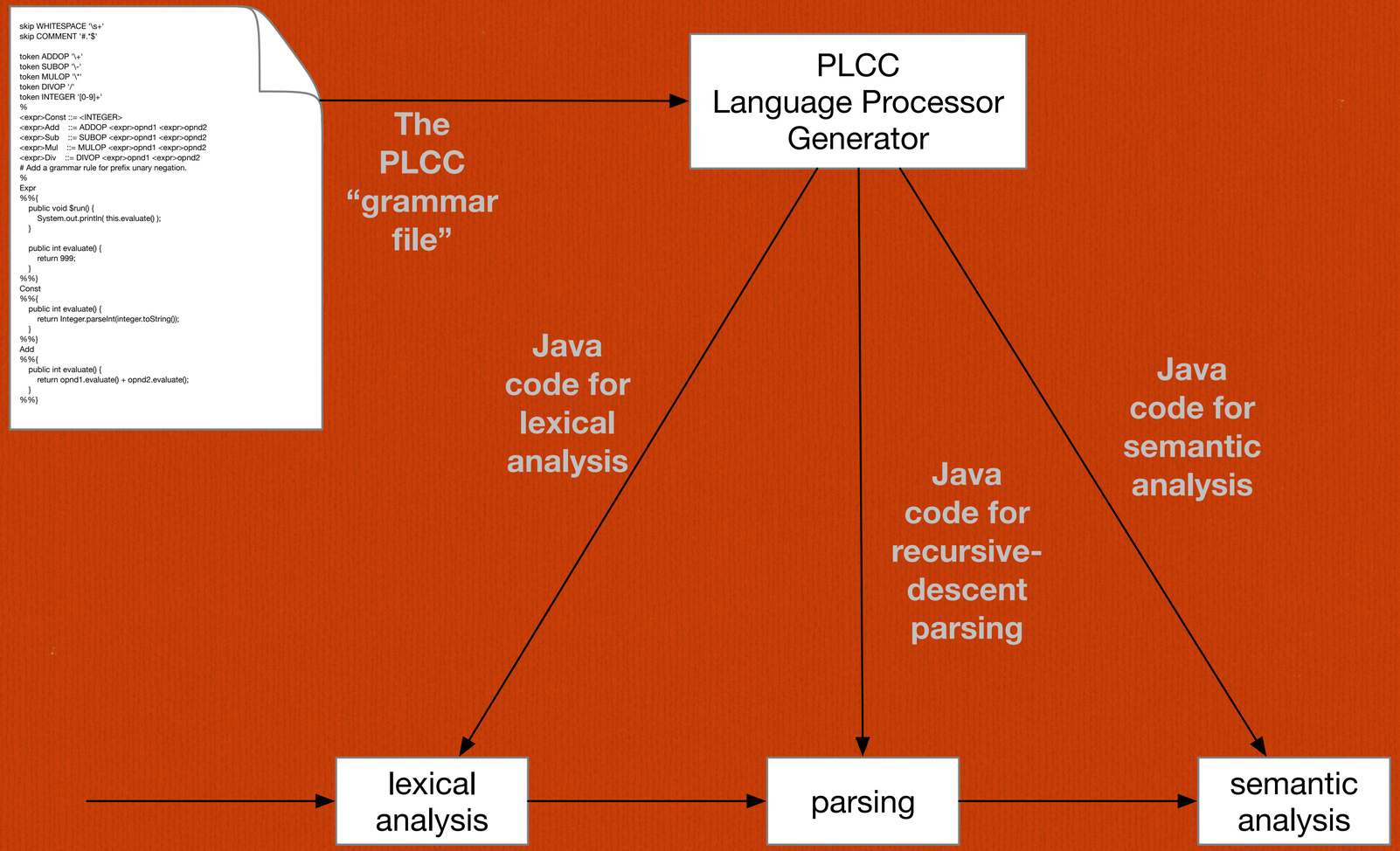
ROCHESTER INSTITUTE OF TECHNOLOGY
R·I·T
1829

The Language Processing Model



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

PLCC Language Processing Model



Form Teams

- The organizers will now help you organize yourselves into teams.
- Each team needs at least one laptop on which the pre-workshop instructions have been completed.
  - https://github.com/ourPLCC/CCSCNE-2025/blob/main/PRE-WORKSHOP.md.

Start a Codespace On Your Fork

- [ ] Navigate to the fork of ourPLCC/CCSCNE-2025 you created in the pre-workshop instructions.
- [ ] Select Code → Codespaces, and select the Codespace you created in the pre-workshop instructions.

Your GitHub Codespace Environment (1 of 2)

- Access your repository's files using the pane on the left.
- Edit files in the pane at the top right.
  - The standard editor is VSCode. It automatically saves as you type.
- Run commands (cd, rep, plccmk, …) using the bash terminal at the bottom right.

Your GitHub Codespace Environment (2 of 2)

- Confirm your environment by executing these commands in the bash shell.
  - `python --version`
  - `java --version`
  - `plcc --version`

Workshop Phase 1
Basic Operation
Lexical Scanning

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
- In **plcc** the filename defaults to **grammar** but you can use other file names and extensions as you wish.
  - Our files will have distinct names. Those names will end in "**`.plcc`**".

Example: Parenthesized Number Lists

- In these notes we will fashion a very simple language processor that reads source code 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 — to collect tokens from the source code
  - Parsing stage — to reorganize the source tokens into a parse tree
  - Semantic stage — to output the list again

(Quick look at result)

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

"rep": read-evaluate-print loop

IMPORTANT: *control-D: UNIX end-of-file*

Language Tokens

- Below is what we specify for the language processor's lexical scanner.

```
skip WHITESPACE '\s+'

token LPAREN '\('

token RPAREN '\)'

token NUMBER '\d+'
```

The token names must be all upper case.

How to Generate the System and Run the System

```
skip WHITESPACE '\s+'

token LPAREN '\('

token RPAREN '\)'

token NUMBER '\d+'
```

```
$ 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!

Workshop Phase 2
Parsing

"Grammar for the grammar"

- 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.

Grammar Definition Details

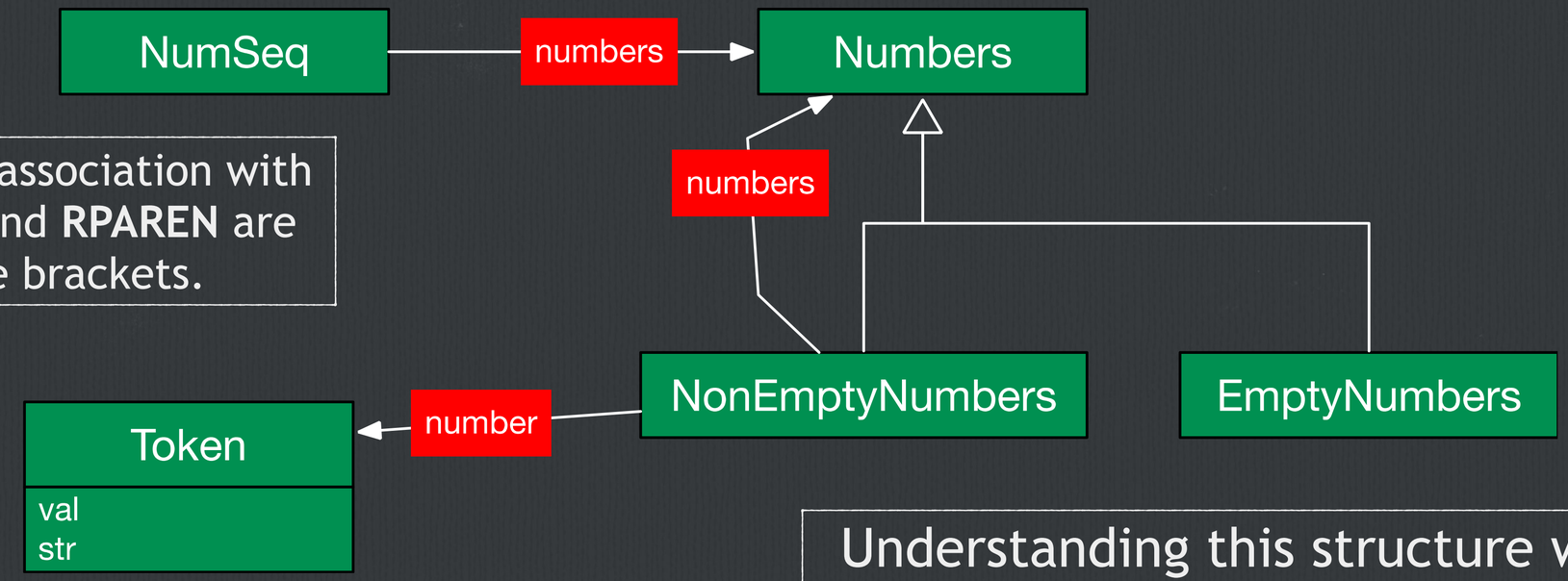
- *Terminal*: an identifier defined in the token section (section 1)
  - possibly surrounded by < … >, if the token's string name is needed
- Non-terminal, left hand side — rule name: *<name>*
  - *<name>*:*Sub_name* if there are multiple rules for *name*
- A non-terminal on the right-hand side refers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

Grammar for Number List Example

```
# Tokens
skip WHITESPACE '\s+'
token NUMBER '\d+'
token LPAREN '\('
token RPAREN '\)'
%
# BNF
<numSeq> ::= LPAREN <numbers> RPAREN    # 1st rule ⇒ tree root
<numbers>:NonEmptyNumbers ::= <NUMBER> <numbers>
<numbers>:EmptyNumbers ::=
%
```

Java Class Correspondence

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



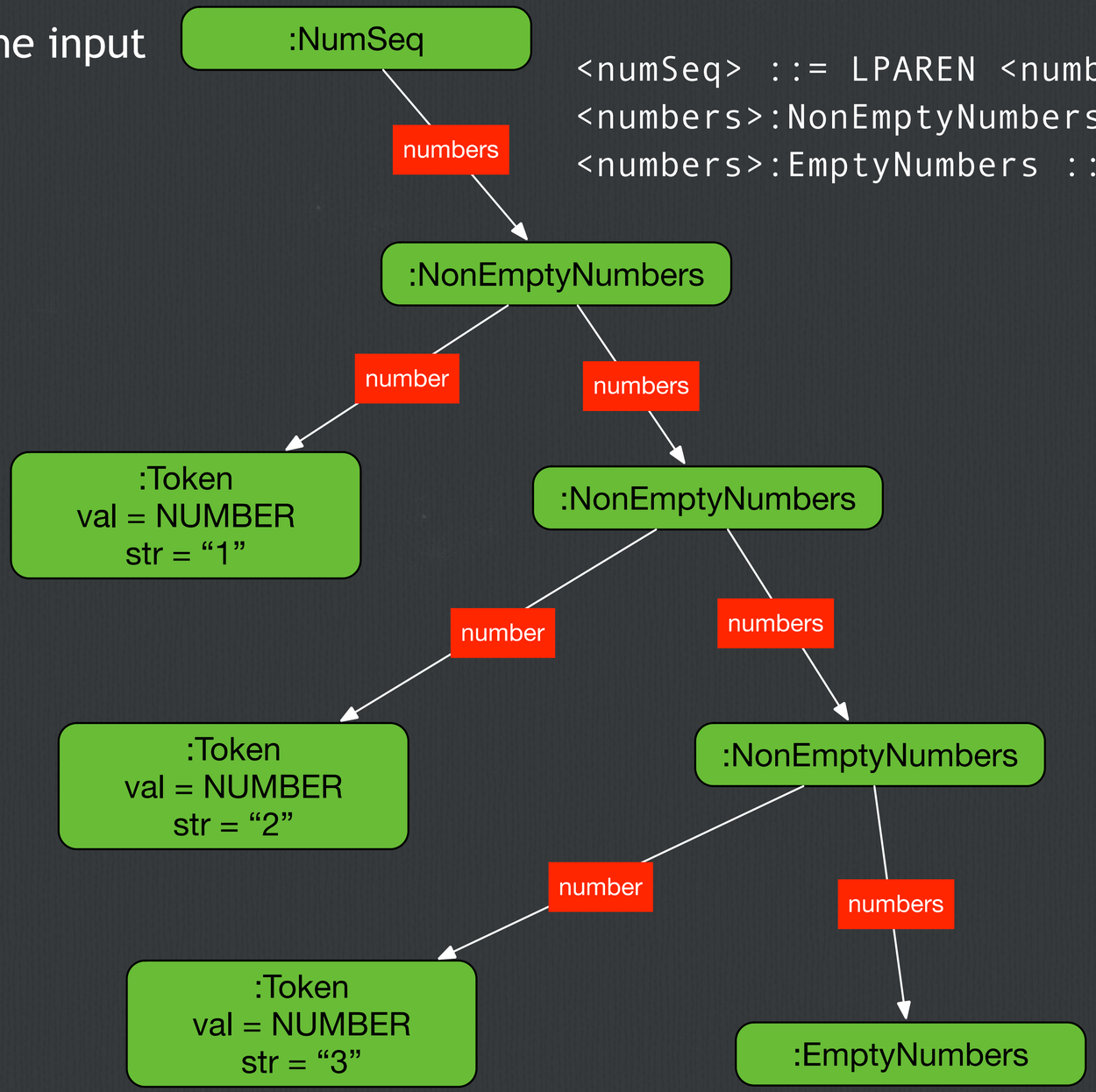
The **NumSeq** class has no association with **Token** because **LPAREN** and **RPAREN** are not wrapped in angle brackets.

Understanding this structure will be more important in Phase 3.

The parse tree built from the input

(1 2 3)

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



How to Generate the System and Run the System

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

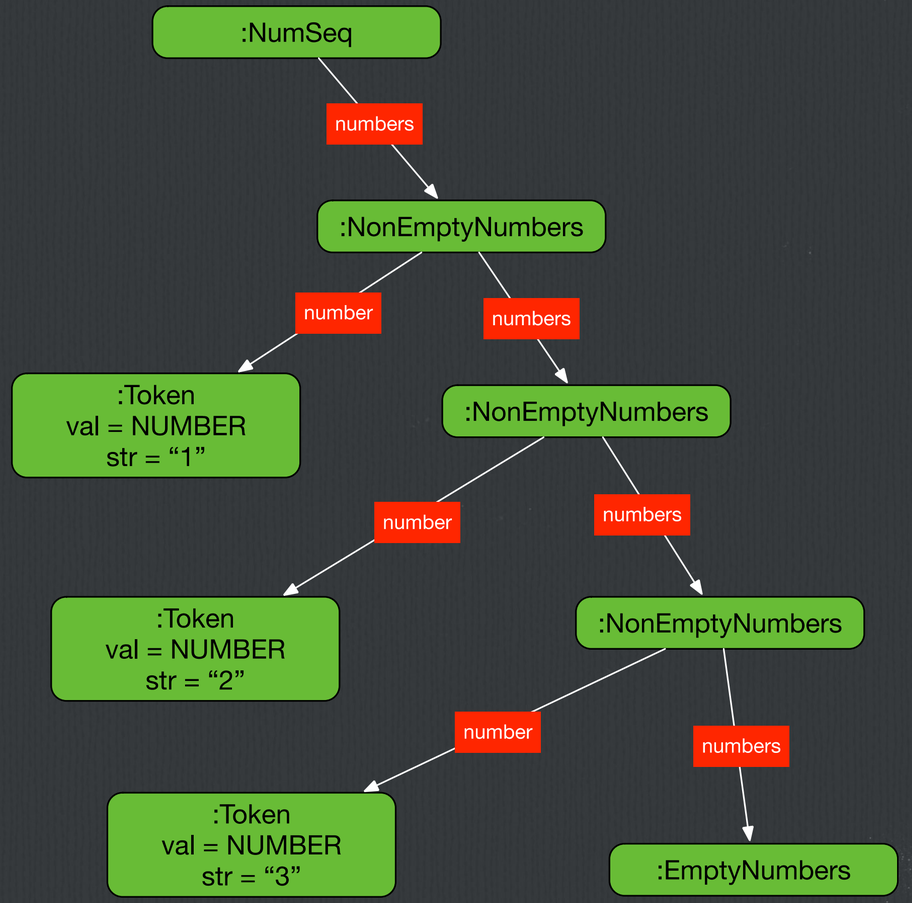
For `rep`*, the parse tree root node is displayed if the parse is successful.*

There was no `$run()` *method defined for* NumSeq. *(TBD)*

```
$ 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: Numbers cannot begin with LPAREN
$
```

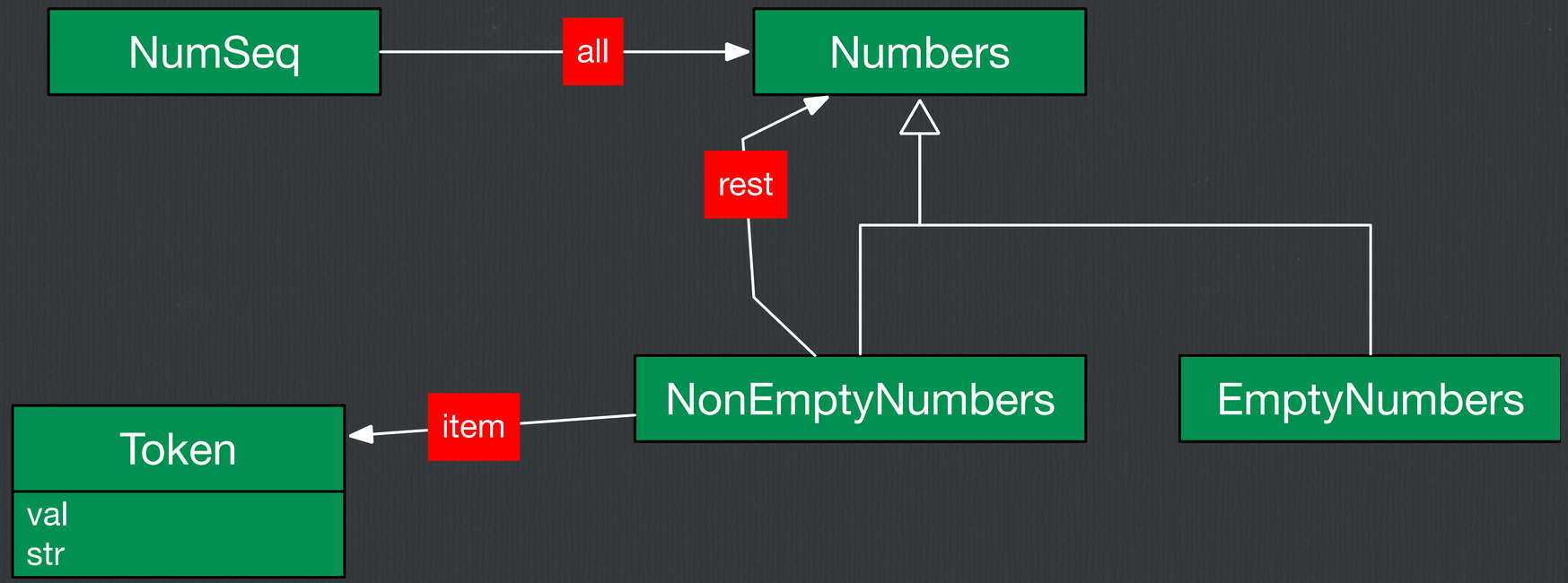
Executing the parse Command with Tracing

```
$ 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.
- Your job is to modify the language's grammar to fix the problem.

Workshop Phase 3
Semantic Analysis

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 code
To the (root) class NumSeq.

Here is a new method.

- If writing a new class, its entire 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

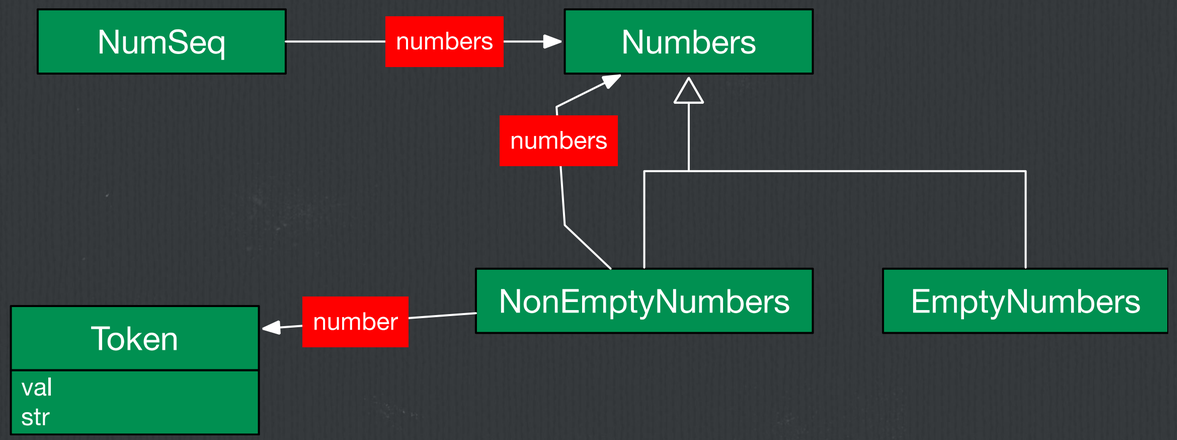
```
Binding
%%{
  public class Binding {
    public String name;
    public Val value;
  }
%%}
```

(Defining new modules and packages is 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 )
```

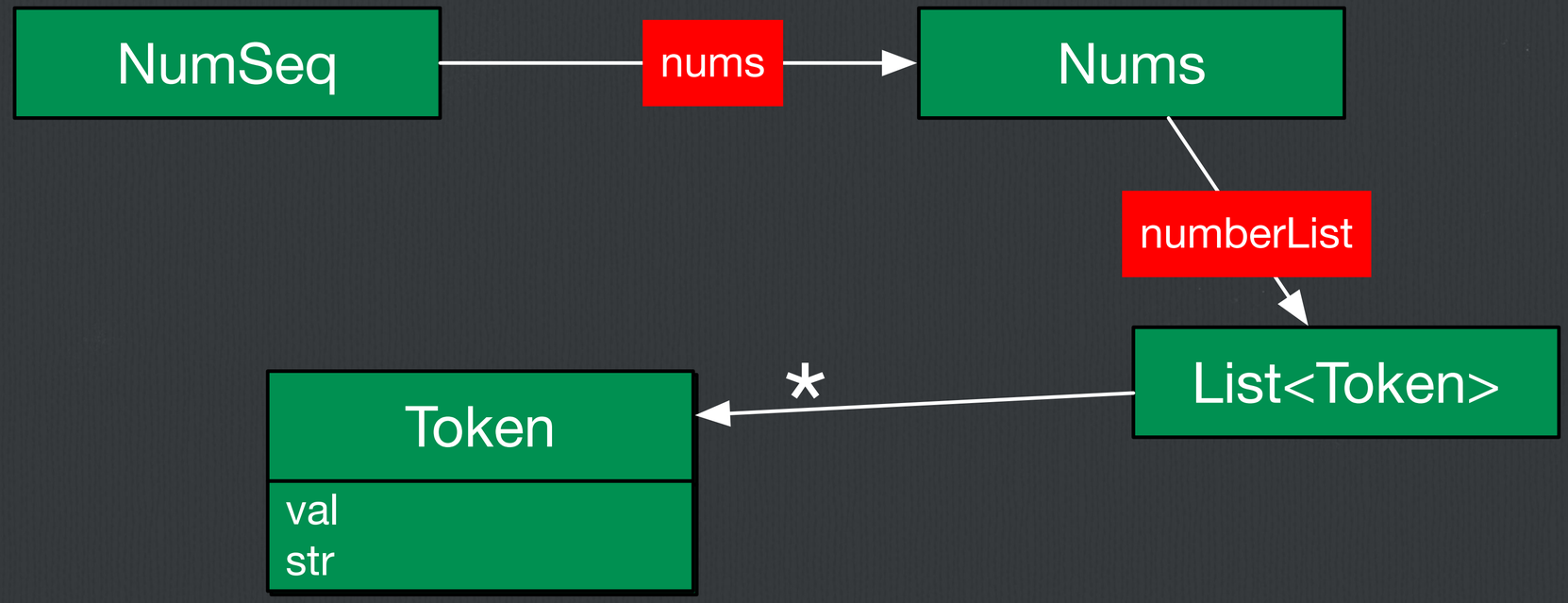
The only difference from the Part 2 trace is that "OK" has changed to "(1 2 3)".

Activity 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 )
```

Epilogue

Activity 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.

Learn More

- Our main repository for PLCC is at https://github.com/ourPLCC/plcc.
- The README.md file therein is a complete but brief manual for installing and using the PLCC system.

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…* ➛The syntax of each rule
- *lhs*: ➛Left-hand side can be
  - <*id*> ➛a class name, which is defined by this rule
  - <*id*>:*sub_id* ➛a class and a subclass name, the latter of which is defined by this rule (required when class is used more than once on LHS)
- *term*: ➛Each term in the sequence on the right side can be
  - <*id*> ➛a class named *id* whose field in the *lhs* class is named *id*
  - <*id*>*id2* ➛a class named *id* whose field in the *lhs* class is named *id2*
  - *TOKEN* ➛a token name, which is not saved in a variable
  - <*TOKEN*> ➛a token name, which is saved as an instance of class `Token` and whose field in the *lhs* class is named *token*
  - <*TOKEN*>*id2* ➛a token name, which is saved as an instance of `Token` but whose field in the *lhs* class is named *id2*

id string begins with lower case in the grammar. The field it creates is identically named but its class's name begins with upper case.

To save your work for your teammate(s) 1/2

- “cd” to the root directory of your repository.
  - `cd /workspaces/CCSCNE-2025`
- Type these commands:
  - `git stage .`
  - `git commit -m “activities done”`
  - `git push`

To save your work for your teammate(s) 2/2

- [ ] Stop the Codespace by clicking 

in the lower left and then selecting Stop Current Codespace.
- [ ] Give your teammates the URL to your fork so that they have access to your team’s work.