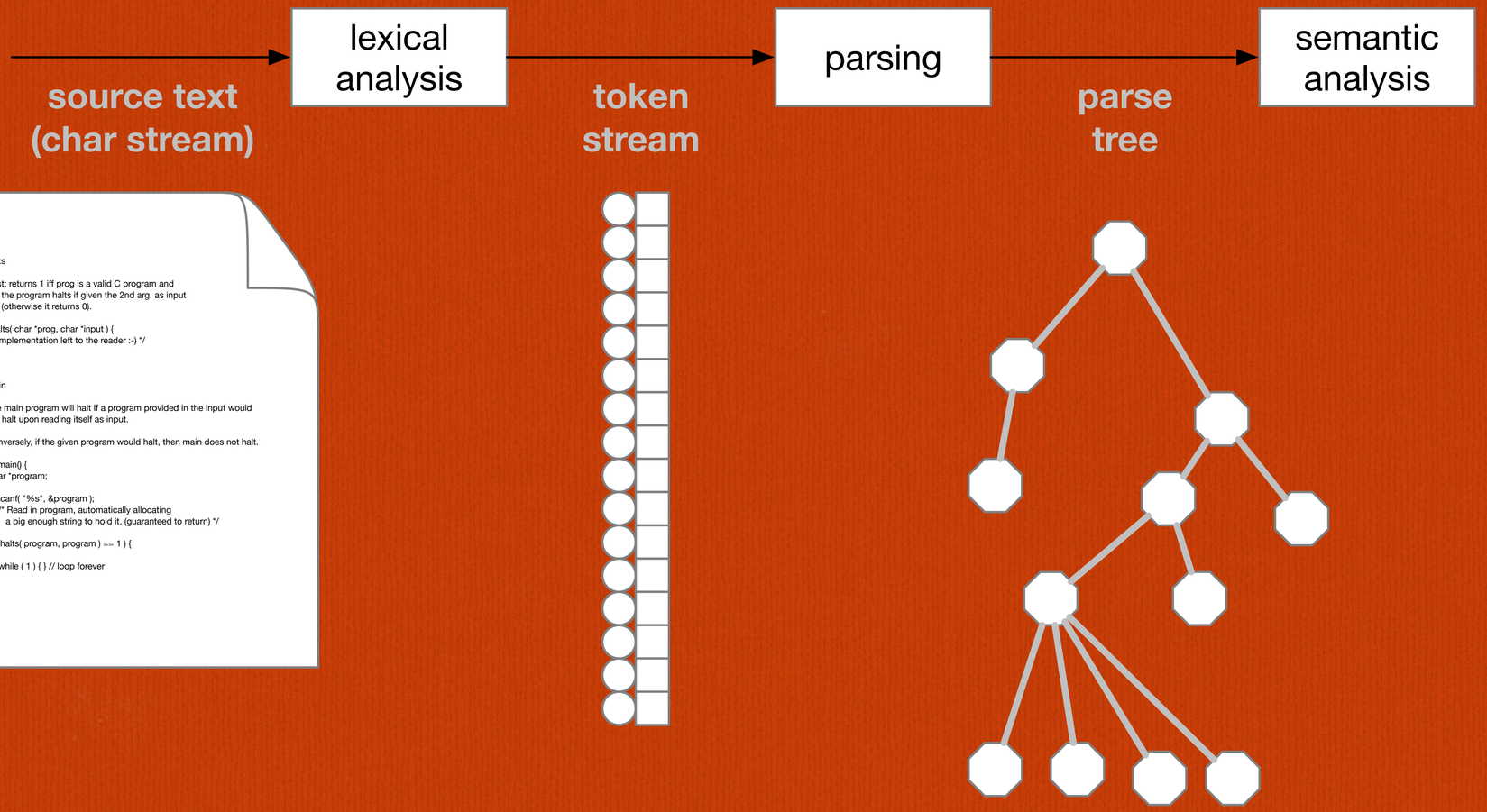
PLCC Overview

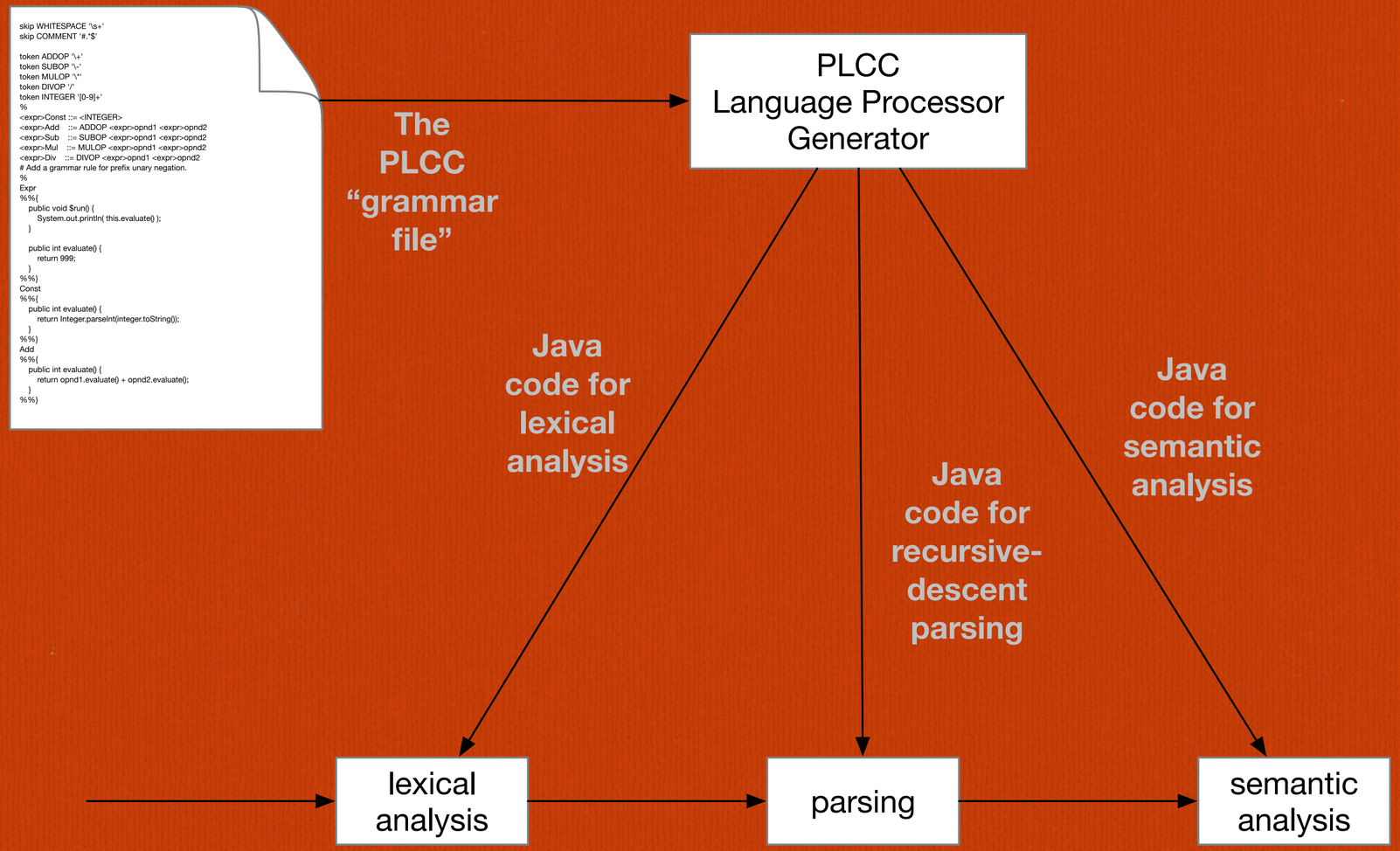
SIGCSE 2024 PLCC Workshop
Stoney Jackson, Jim Heliotis
Tim Fossum

The Language Processing Model



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

The Language Processing Model

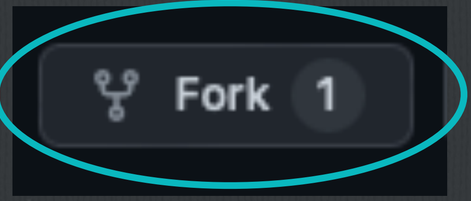


Break for Tool Setup

- The organizers will now help you organize yourselves into teams.
- If you have already done the setup on your computer there is no need to do more now.
- There should be one computer that has done the setup for the workshop.
  - Setup summary is on the next slide.
  - Details at https://github.com/ourPLCC/sigcse2024/blob/main/Docs/Pre-Workshop-Setup/README.md (but step 8 is different)

Break for Tool Setup

- Each team: Set up one computer to access the workshop materials.
  - Navigate the browser to https://github.com/ourPLCC/sigcse2024.
  - Push the "Fork" button to make a copy of the repository.
    - Share the new fork's URL with your teammates.
  - Prepend "https://gitpod.io/#" to that new URL and get that page.
  - Log in with your GitHub account.
  - Select the defaults on the next web page that appears.



Your GitPod Environment (1 of 2)

- Access your repository's contents using the pane on the left.
- Edit files in the pane at the top right.
  - The standard editor is VS Code.
- Run commands (git, plcc, …) using the bash terminal at the bottom right.
  - (Wait for it to complete the Git command that clones the repo.)

Your GitPod Environment (2 of 2)

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

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

"rep": read-evaluate-print loop

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+'
```

```
$ 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 **Part1** 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!

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.

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

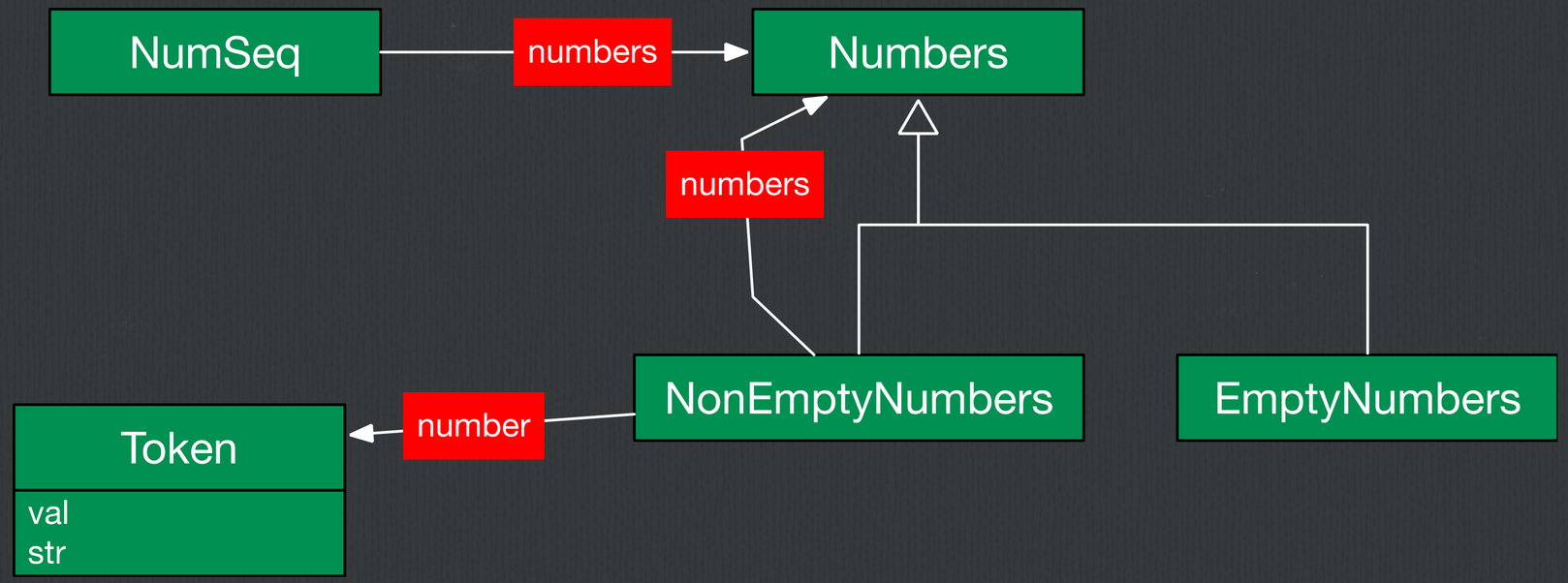
- *Terminal*: an identifier defined in the token section (section 1)
  - possibly surrounded by < … >, if the token's string name is needed
- Non-terminal, left hand side — rule name: *<name>*
  - *<name>*`:Sub_name` if there are multiple rules for *name*
- A non-terminal on the right-hand side refers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

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 ::=
%
```

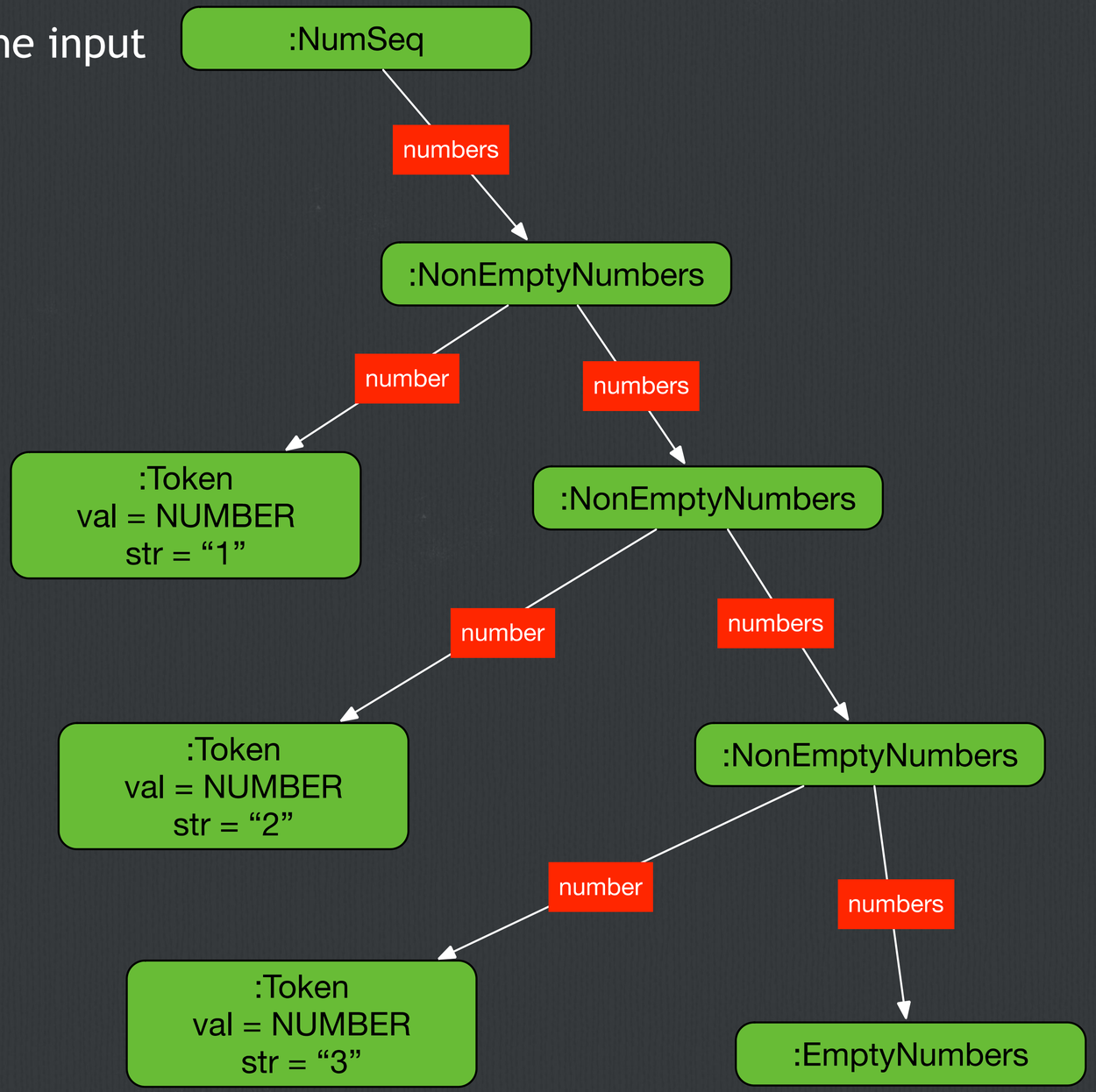
Java Class Correspondence

```
<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 ::=
```

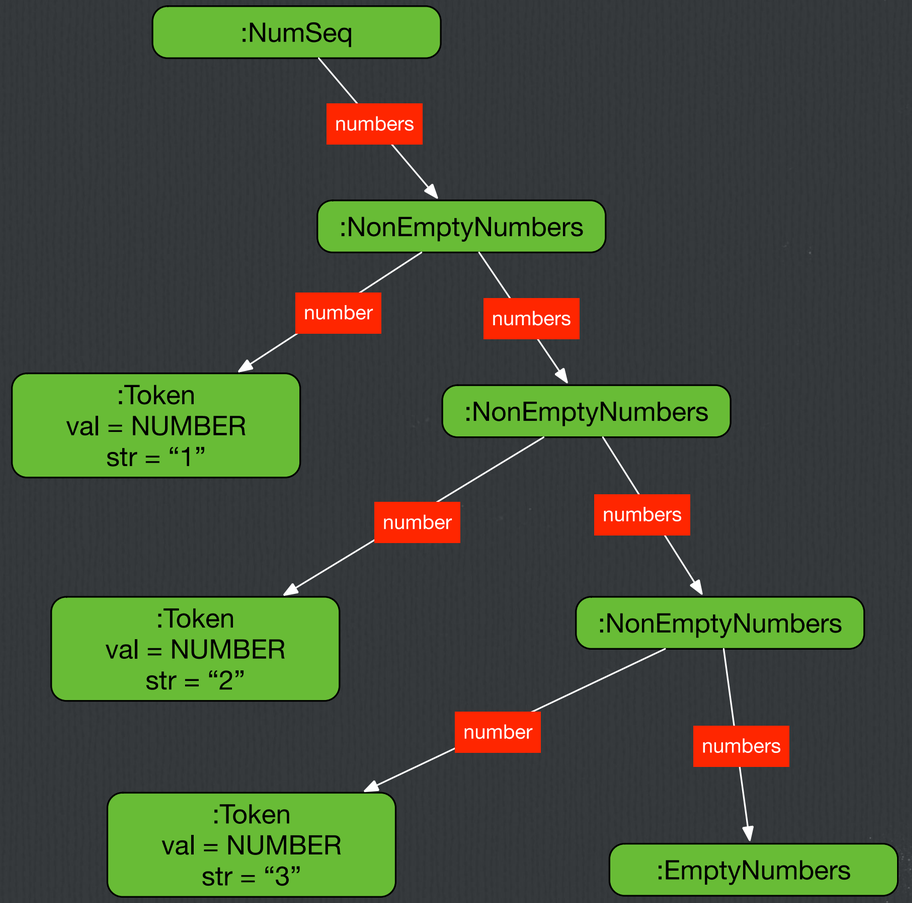
For `rep`*, the parse tree root node is displayed if the parse is successful.*

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: Numbers cannot begin with LPAREN
$
```

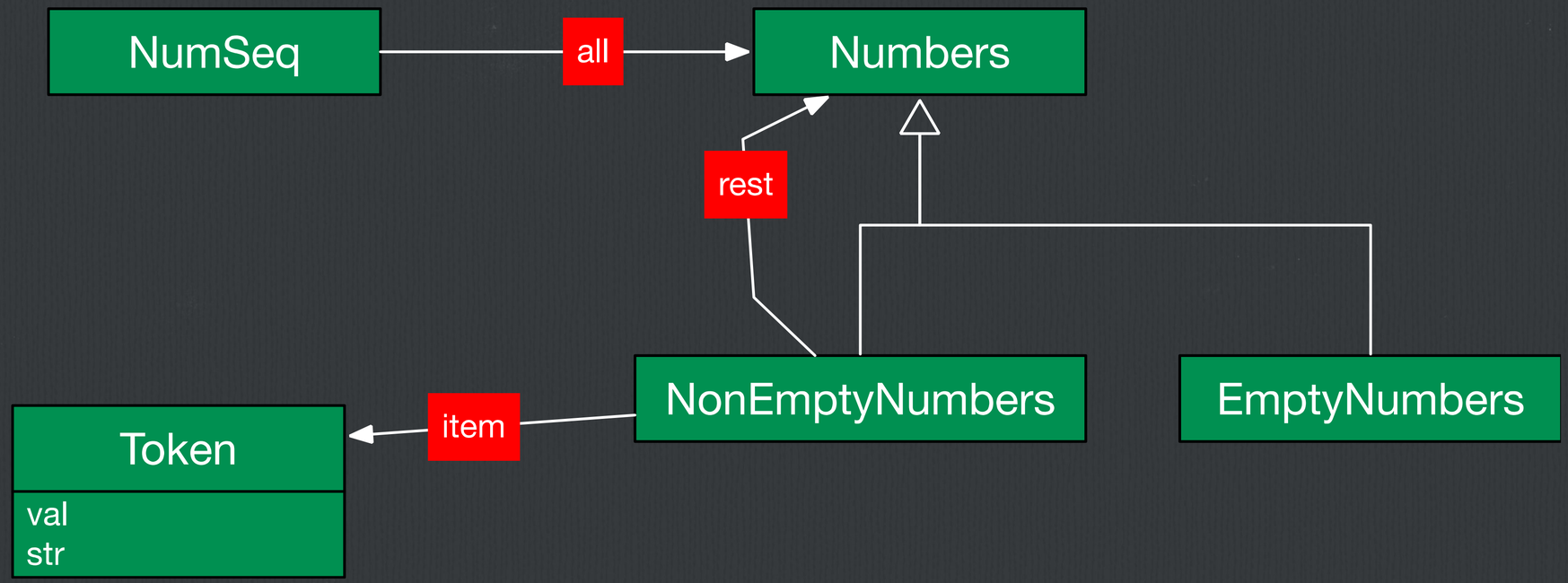
Executing the parse Command with Tracing

```
$ 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 **Part2** 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." );
        }
    %%}
```

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

Here is a new method.

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

- 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

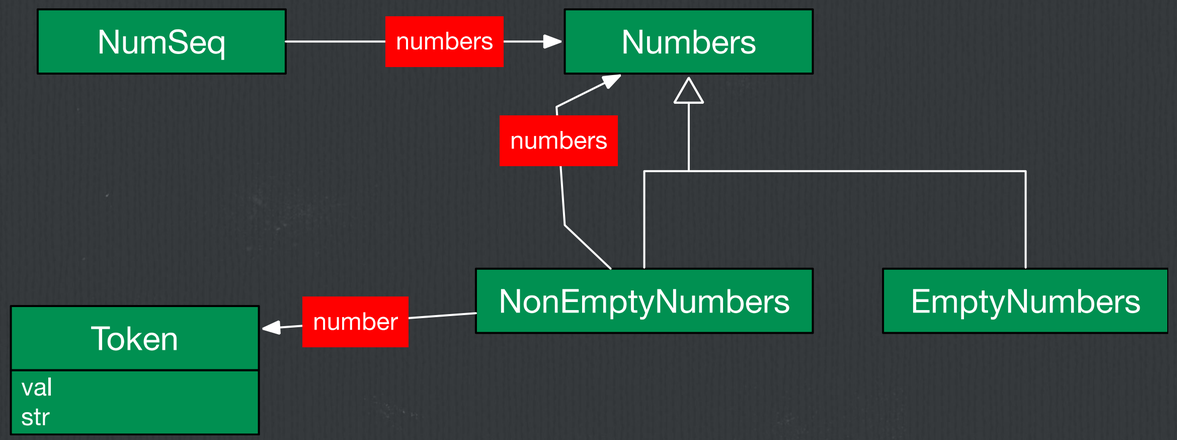
```
Binding
%%{
  public class Binding {
    public String name;
    public Val value;
  }
%%}
```

(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 )
```

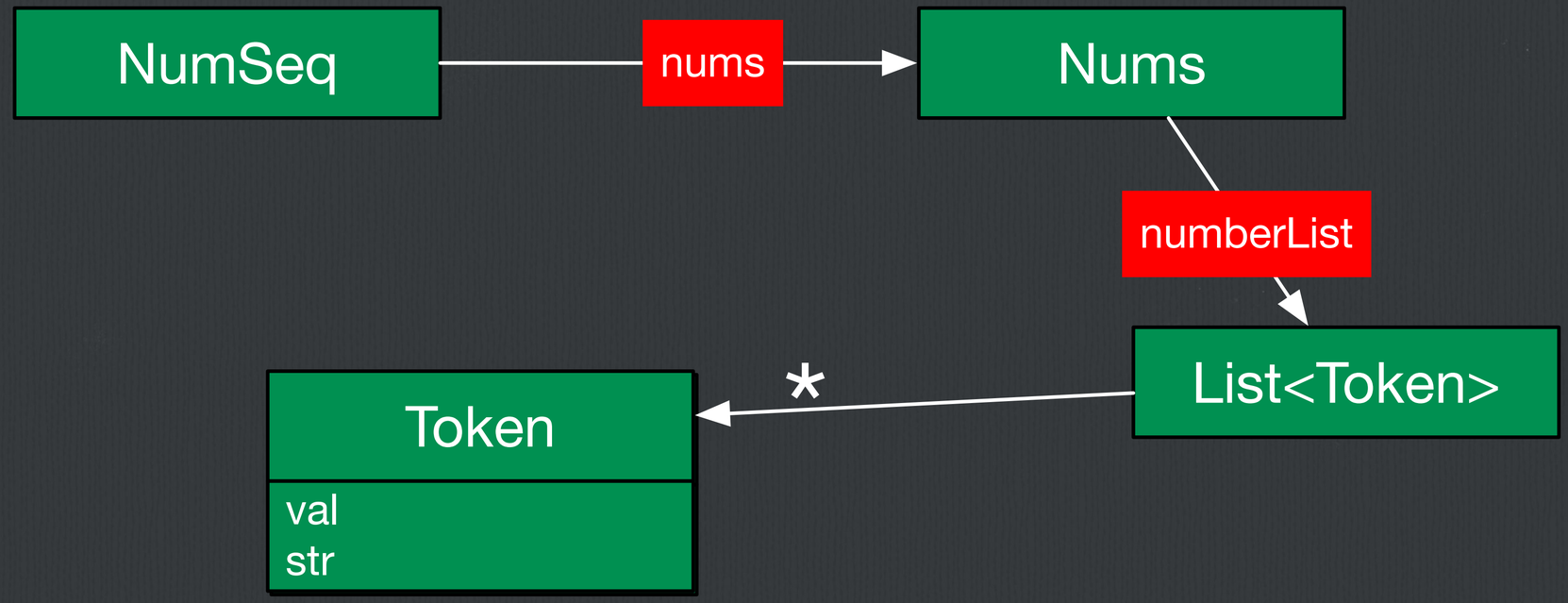
The only difference from the previous trace is that "OK" has changed to "(1 2 3)".

Activity 3

- "**cd**" to the **Part3** 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 )
```

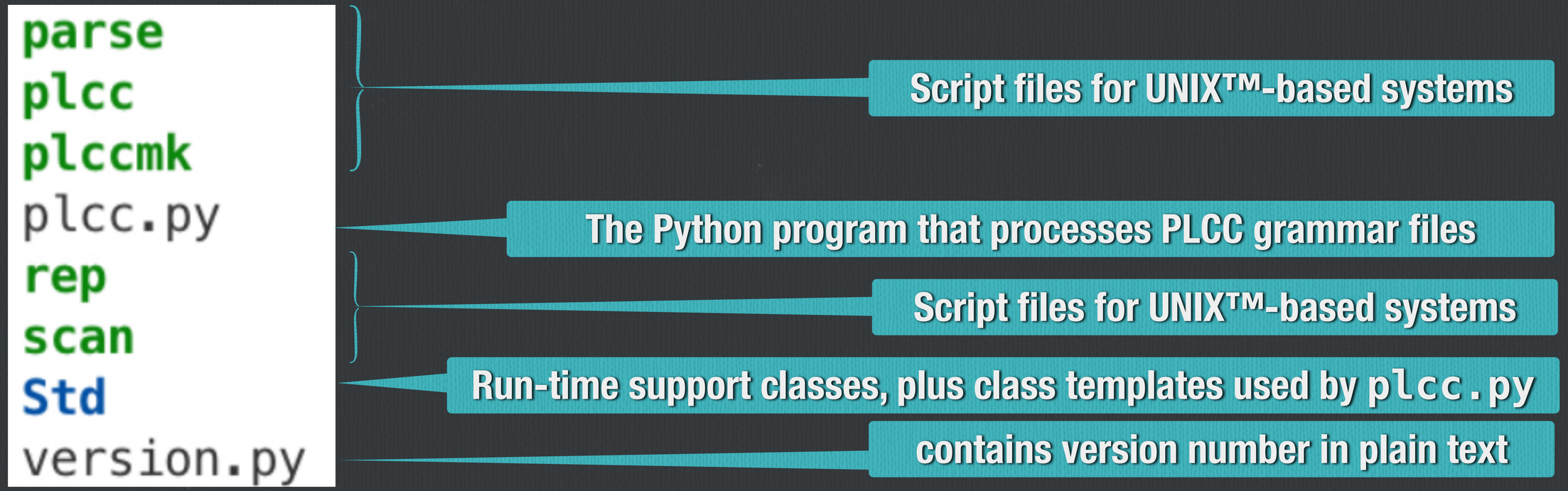
Activity 4

- This is a complete system that you can study and run
- "**cd**" to the **Part4** 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

The PLCC Package `src` Directory

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



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.