Test-data generation for XText

A two-phase approach to generating good test data

Lukas Härtel
lukashaertel@uni-koblenz.de

University of Koblenz and Landau
Germany

Background

- Based on work done for the 2013/2014 SLE course at UKOLD; see Lukas & Johannes Härtel

  softlang.wikidot.com/course:sle1314-results

- Related Material can be found at the Software Languages Teams portal

  softlang.wikidot.com/course:sle

Background

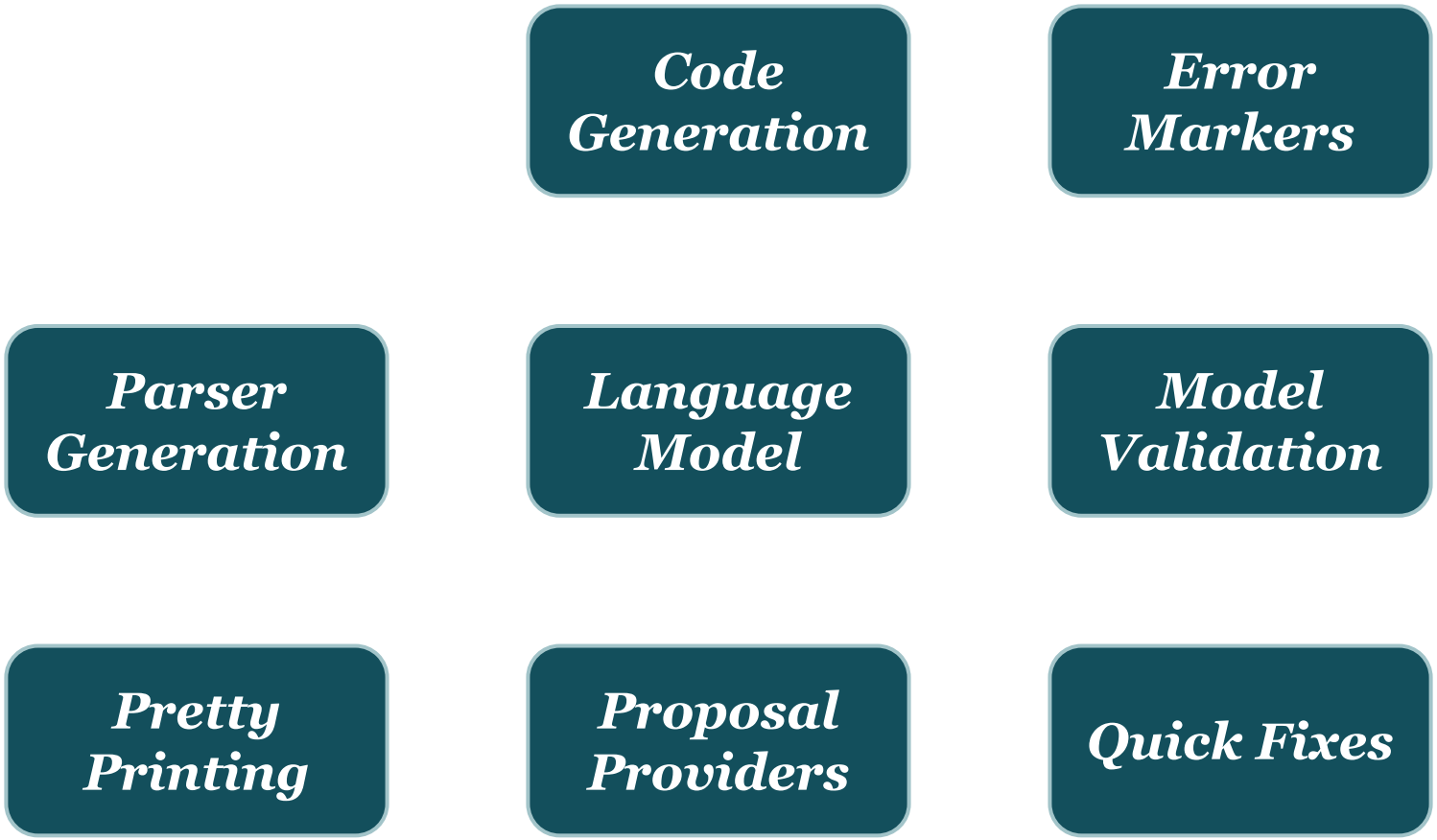
- Eclipse Modeling framework EMF
  - Infrastructure for model based tools
  - Defines a class-diagram-like meta-model for meta-modeling
  - Tightly integrated with the Eclipse IDE

- XText
  - Framework for writing domain specific languages
  - Generates language models, parsers, and tool integration for Eclipse
  - Based on the EMF

Motivation

- Test-data generation is *The missing Piece* for the XText tooling framework
- DSL developers need to think of their own test cases → Is *coverage* achieved?
- Performance testing needs big test sets

Motivation


Code Generation
Error Markers
Parser Generation
Language Model
Model Validation
Pretty Printing
Proposal Providers
Quick Fixes

Motivation

Test-data Generation

Code Generation

Error Markers

Parser Generation

Language Model

Model Validation

Pretty Printing

Proposal Providers

Quick Fixes

Method

- Grammar models and configuration models are extracted
- Grammar is customized for generation
- Adjusted grammars productions are iterated
- Productions are fed into a post-processor

Method - *Structure*

- What artefacts are involved
- What's the flow of model
- How are the models transformed

Method - *Structure*

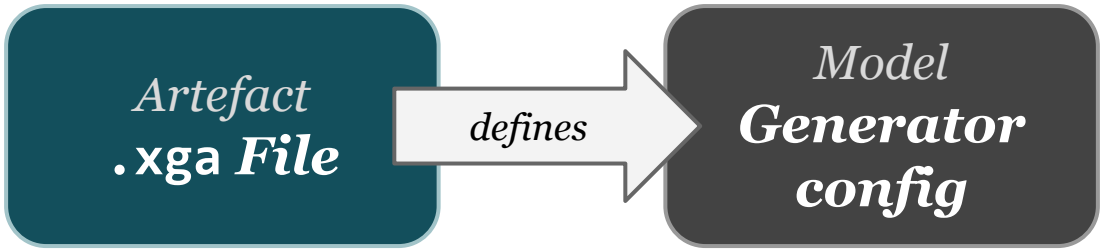
Defines the grammar and the model

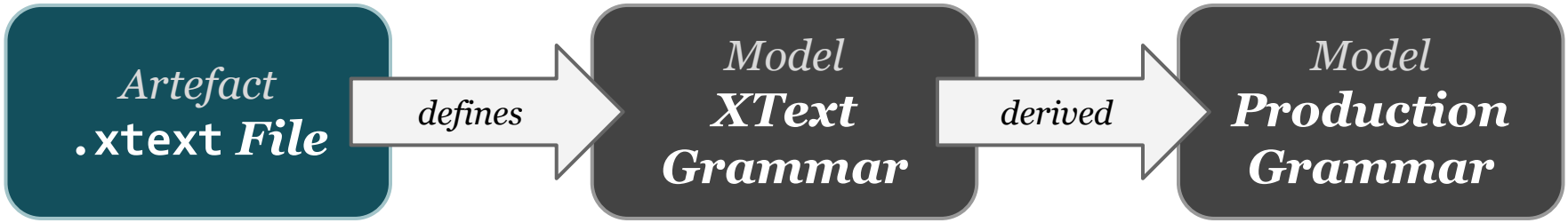
Configures the test-data generation

```
grammar sle.fsml.FSML with org.eclipse.xtext.common.Terminals

generate fSML "http://www.fsml.sle/FSML"

/**
 * A FSM is a collection of multiple states
 */
FSM:
    states+=FSMState*;

/**
 * A state can be optional, has a name and is
 * composed of multiple transitions
 */
FSMState:
    (initial?="initial")? 'state' name=ID
    '{'
    transitions+=FSMTransition*
    '}';


/**
 * A transition has an input, an optional
 * action and an optional target state
 */
FSMTransition:
    input=ID
    ('/' action=ID)?
    ('->' target=[FSMState|ID])? ';';
```

```
sle.fsml.FSML(0..10)
{
    // Replace the one keyword in the rule
    // FSMState with the one specified here
    replace "initial"{0,1} in FSMState with "<initial>"

    // Only one ID here so no problem
    replace ID in FSMState with "<state name>"

    // In FSMTransition there are three ID calls,
    // specify their positions to make clear where to replace
    replace ID/0 in FSMTransition with "<input value>"
    replace ID/1 in FSMTransition with "<action value>"
    replace ID/2 in FSMTransition with "<state reference>"

    // We want at least two states in the FSM, makes sense doesn't it
    adjust multiplicity in FSM with {2,3}


    // Lets also have some transitions
    adjust multiplicity in FSMState with {2,3}

    // Also always an action and a targeted transition
    adjust multiplicity/0 in FSMTransition with {1, 1}
//  adjust multiplicity/1 in FSMTransition with {1, 1}
}
```

Method - *Structure*

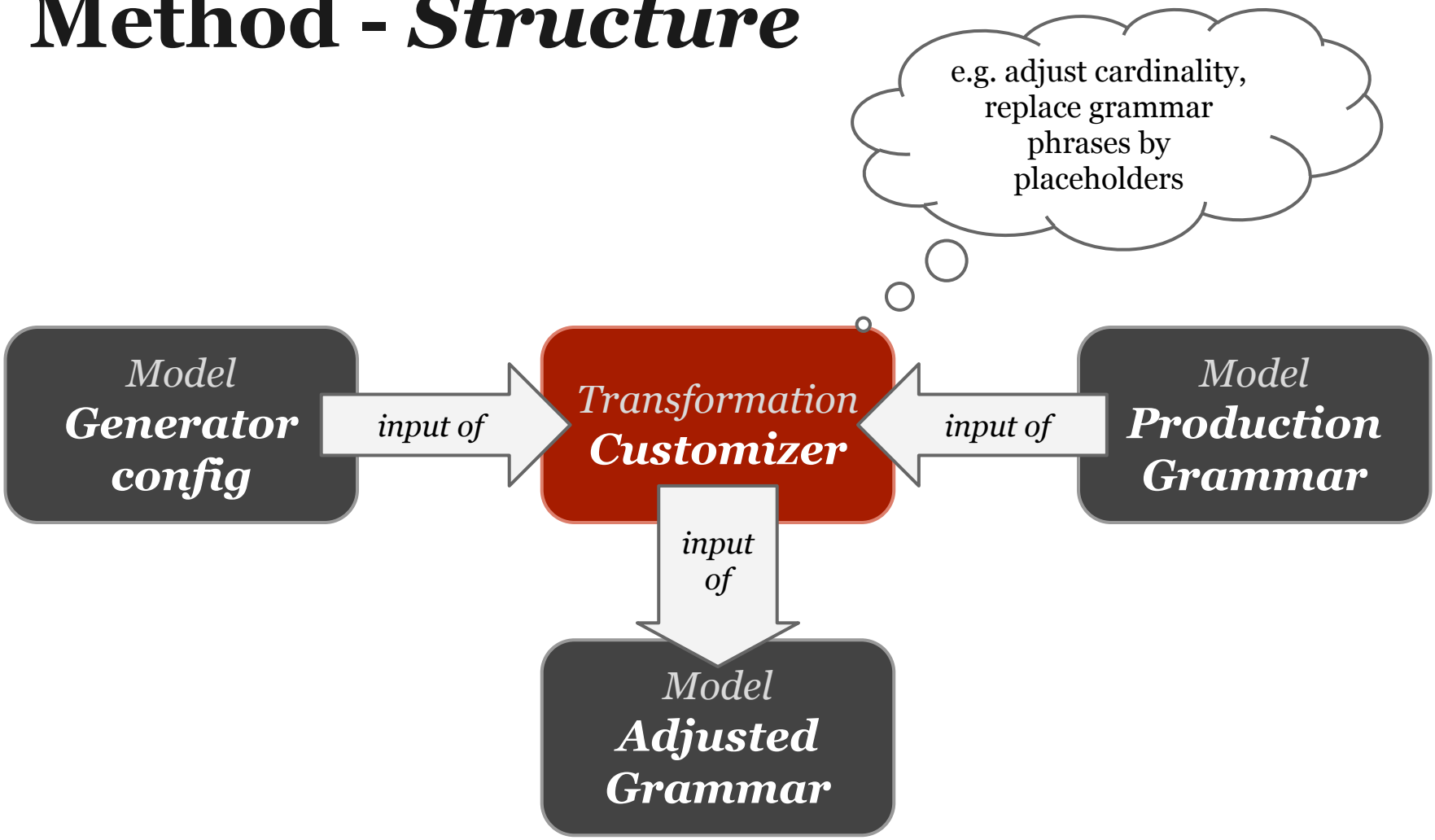




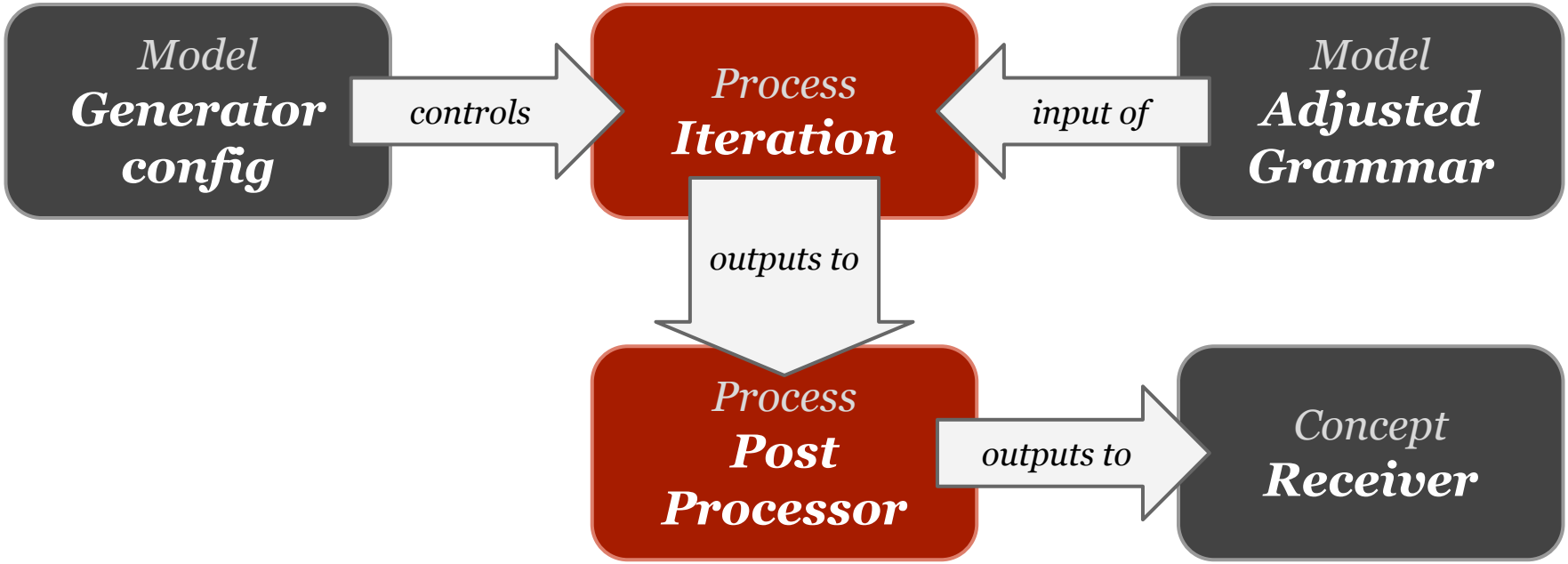
Artefacts are parsed by XText

Transformation is hand-written

Method - *Structure*



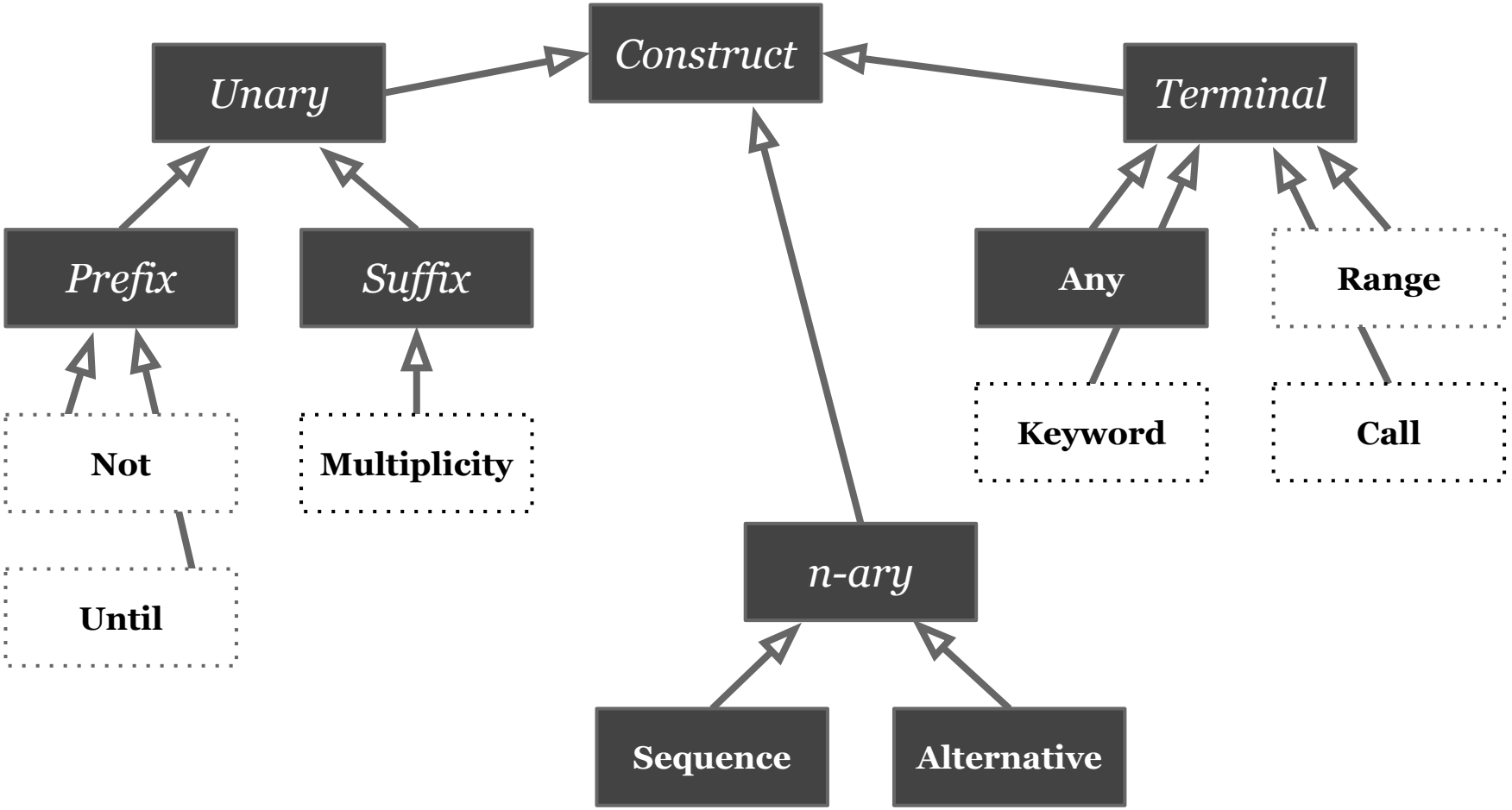
Method - *Structure*



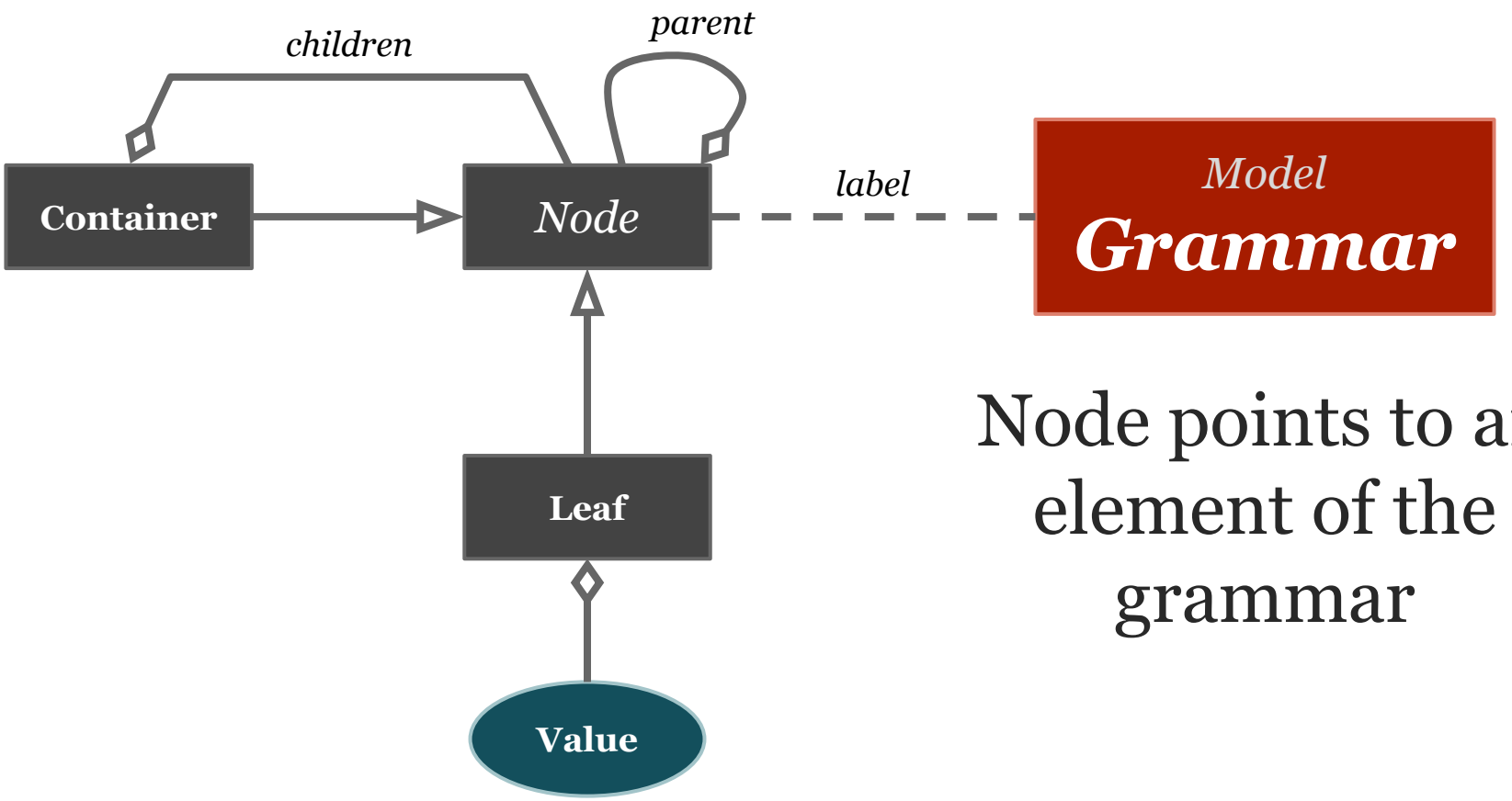
Method - *Iteration*

- What are the grammatical constructs
- What is the resulting test-data
- How are they iterated

Method - *Iteration*

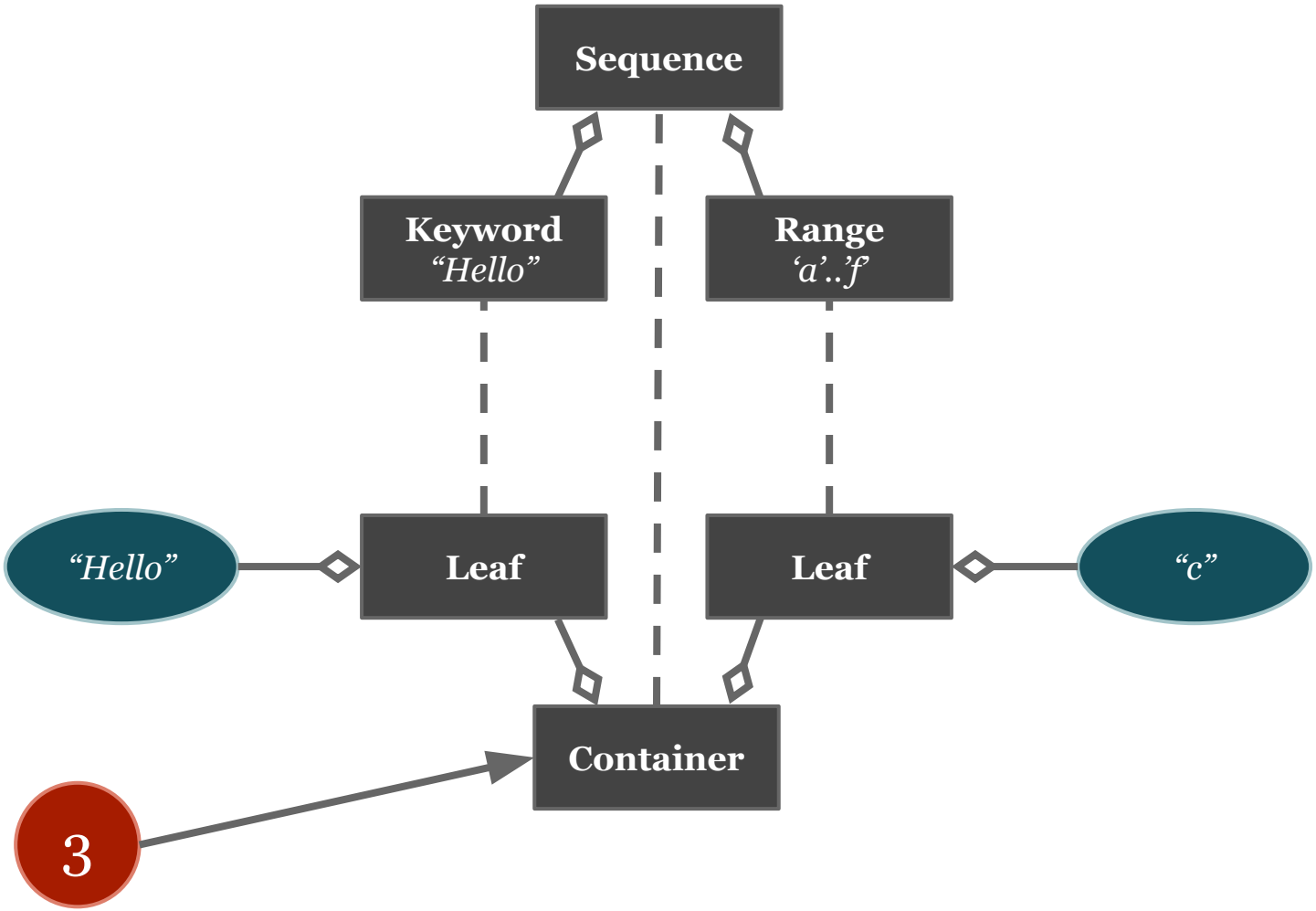


Method - *Iteration*

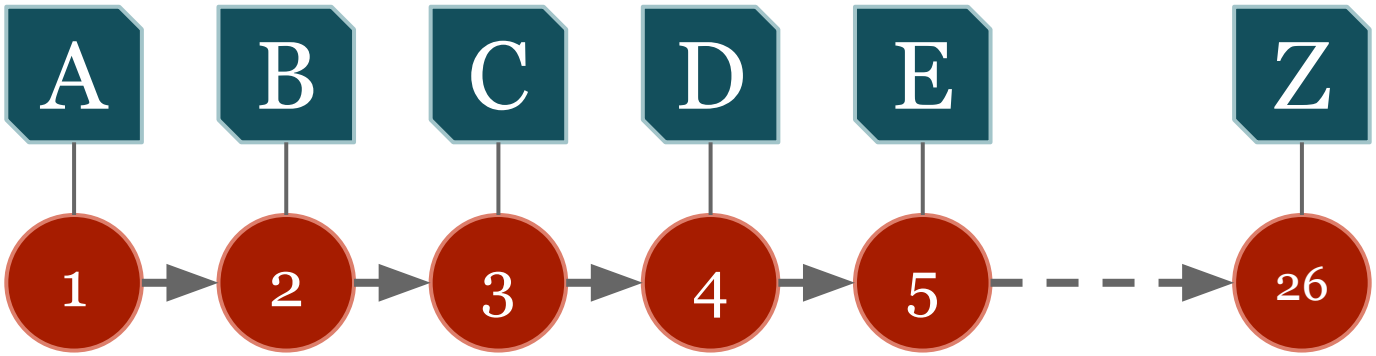


Node points to an element of the grammar

Method - *Iteration*



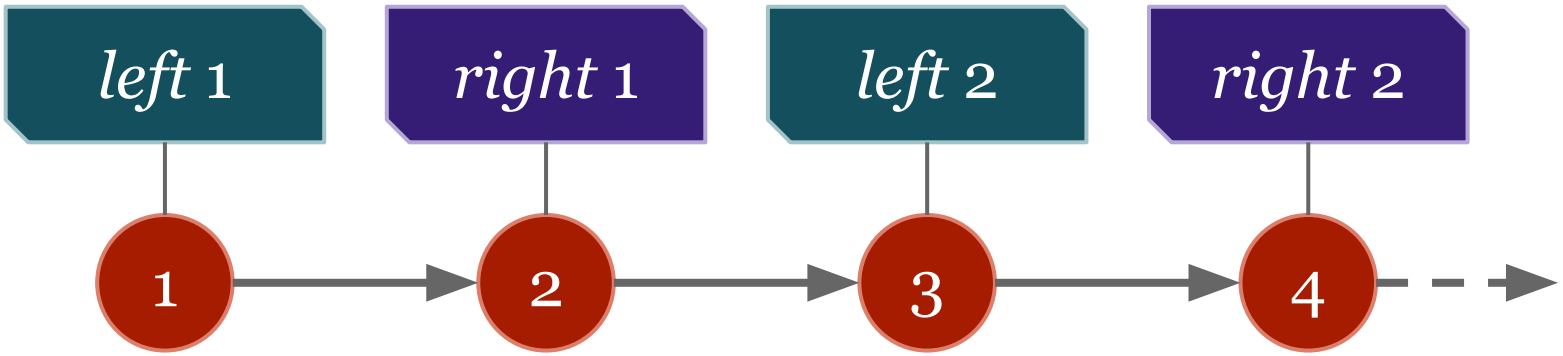
.



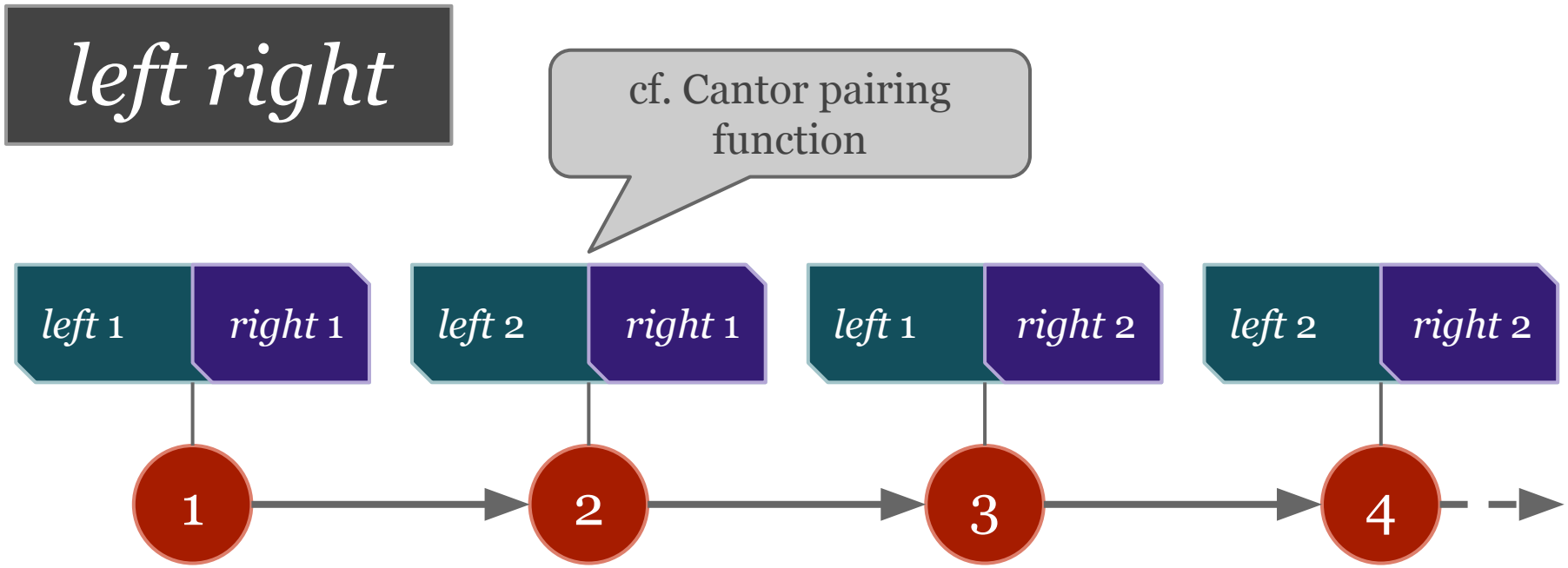
Range *c1* .. *c2*
similar

Any Character

left | right



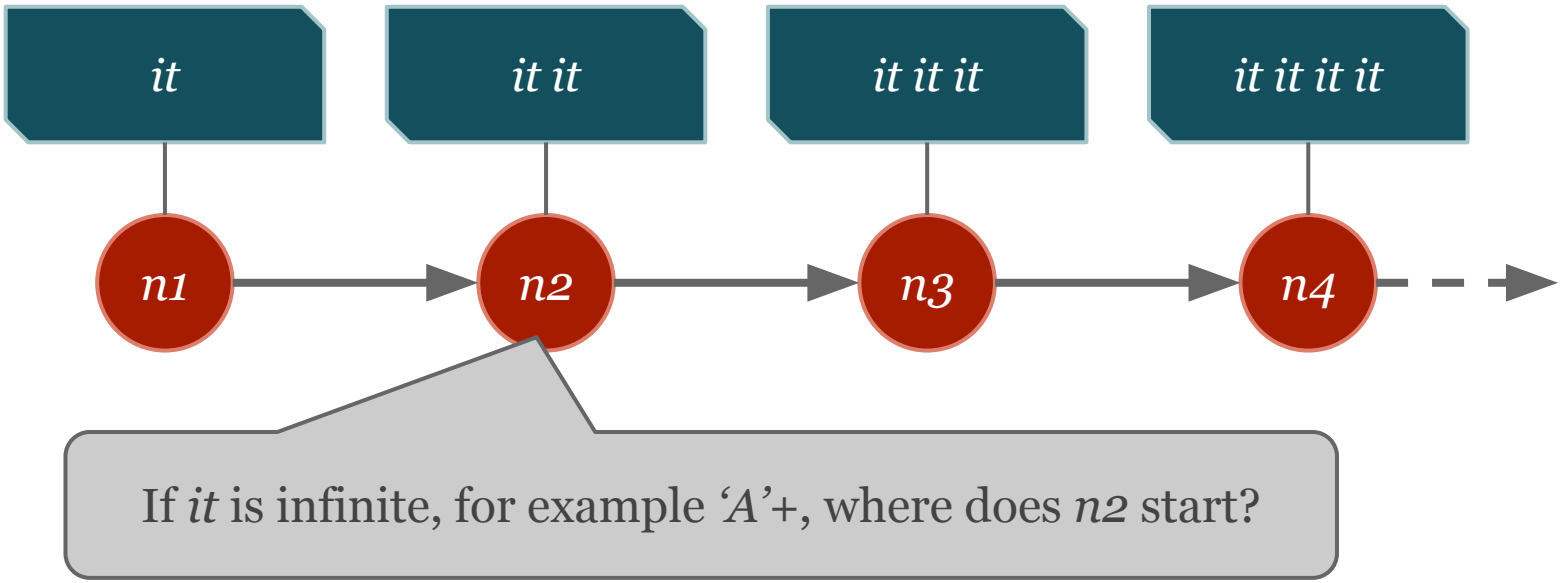
Alternative Composition



Sequential Composition

Lukas Härtel - University of Koblenz and Landau

it+

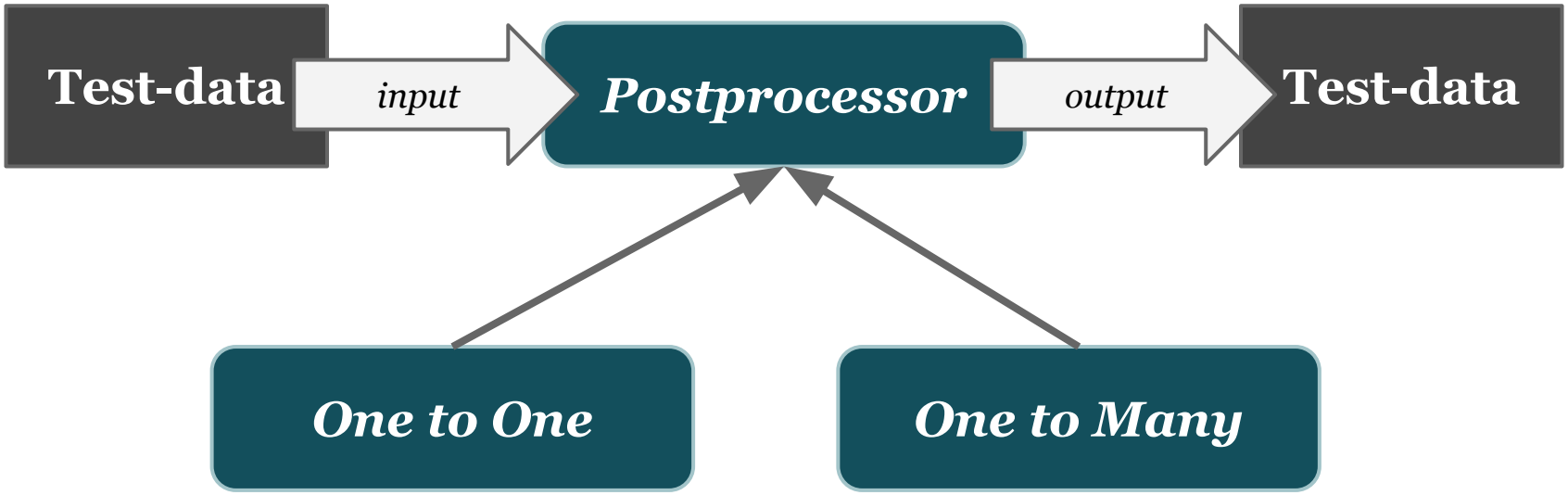


Plus Operator

Method - *Post Processing*

- How are the post processors formulated
- How to deal with multiple ways to transform an input
- What are the characteristics

Method - *Post Processing*

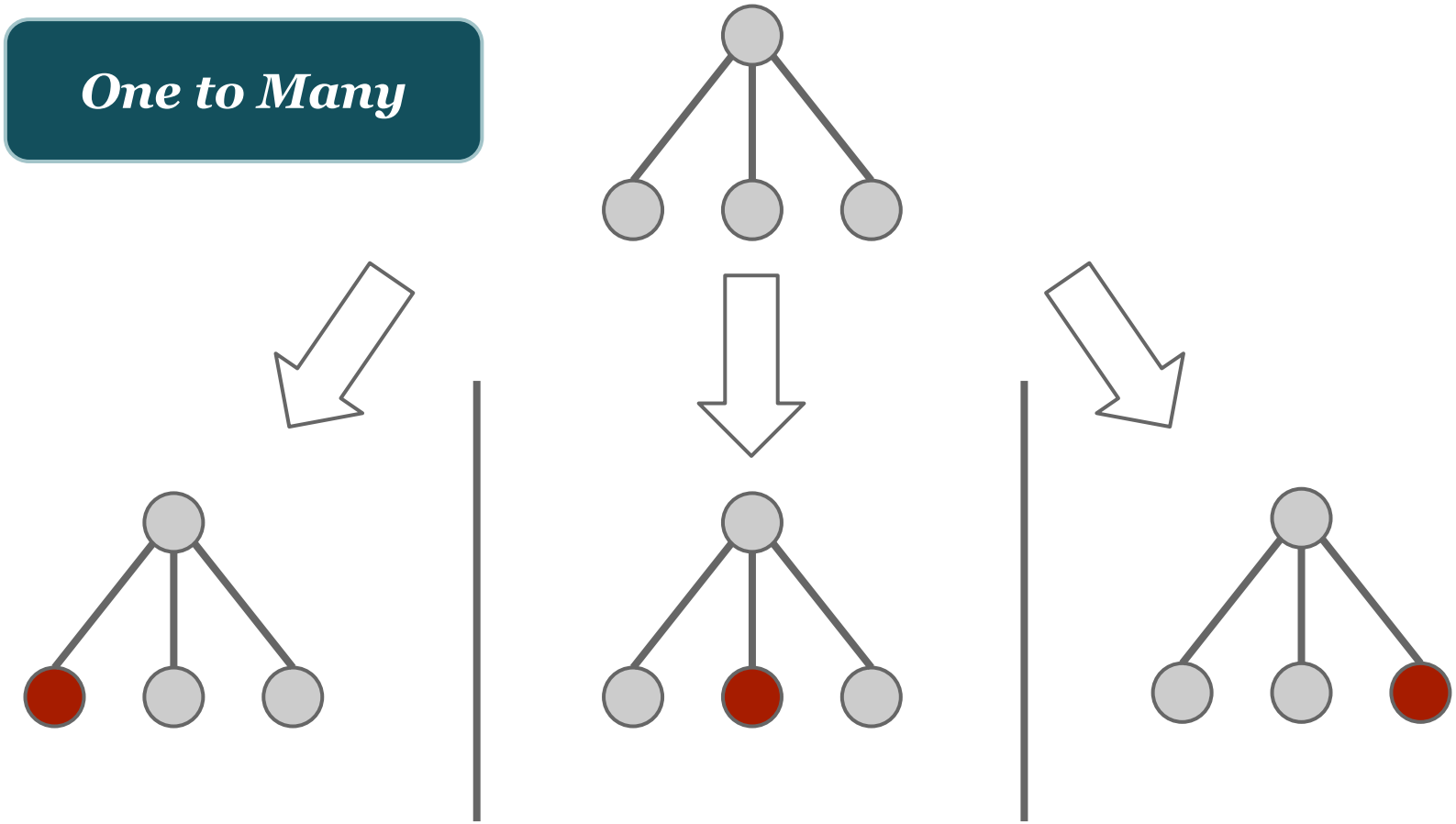


Map one parse-tree to one resulting parse-tree

Transform one parse-tree to many output parse-trees

Method - *Post Processing*

One to Many

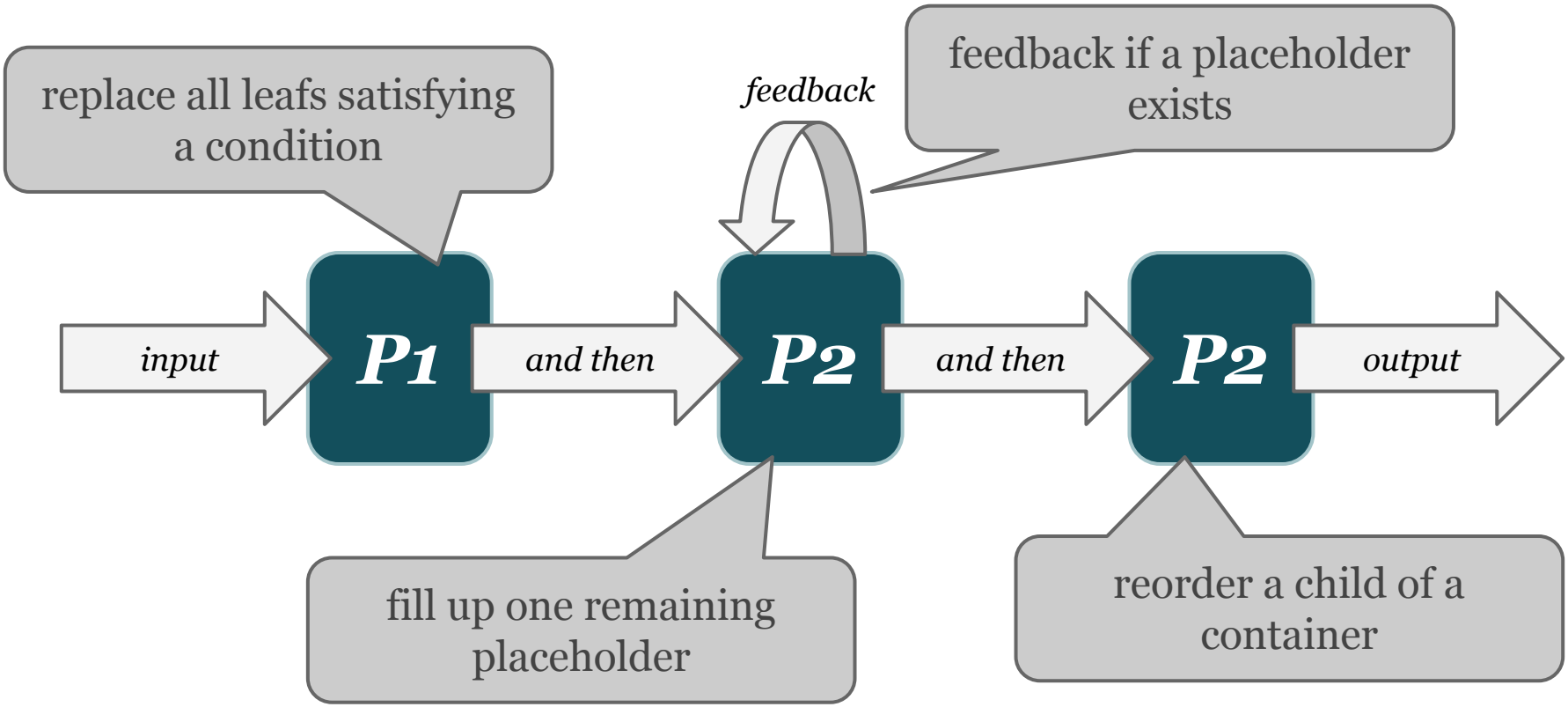


```
class ReplaceInitial extends TransformOne {

    override protected select(Leaf it) {
        value == "<initial>"
    }

    override protected transform(Leaf it) {
        new Leaf(label, "initial")
    }

}

class RemoveRemainingInitial extends RemoveAll {

    override protected remove(Leaf it) {
        value == "<initial>"
    }
}
```

One to many: for every position make one new parse-tree

One to one: remove all positions in input and return

Method - *Post Processing*



Conclusion

- Fast generation of large test sets is possible
- Smaller valid-data solutions can be written down easily
- No direct modification of the grammar is needed
- Nice integration with Eclipse

Conclusion - Future Work

- Formalization of post-processors
- DSL for post-processors composition
- Studies on more complicated test-data generation scenarios
- Fix some stability issues

Discussion

Thank you.