

## Part III. Technical Spaces

### 20. Analysis and Model Management in the Technical Space Grammarware and Treeware (Context-Free Syntax Analysis)

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und  
Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

<http://st.inf.tu-dresden.de>

Version 19-1.1, 21.12.19

- 1) Parsing
- 2) Regular Tree Grammars
- 3) Tree Construction
- 4) Text Algebrae
- 5) Port Graph Algebrae
- 6) Pseudocode



**DRESDEN  
concept**  
Exzellenz aus  
Wissenschaft  
und Kultur

- ▶ Obligatory:
  - <http://wwwantlr.org>
- ▶ Optional:
  - Cocktail [www.cocolab.de](http://www.cocolab.de), die Compiler-Toolbox für die schnellsten Compiler der Welt (kommerziell, Demoverversionen erhältlich)
  - TaTa Tree Grammars <http://tata.gforge.inria.fr/> and all the tree theory
- ▶ Oana Andrei, Helene Kirchner. A Port Graph Calculus for Autonomic Computing and Invariant Verification. A. Corradini. TERMGRAPH 2009, 5th International Workshop on Computing with Terms and Graphs, Satellite Event of ETAPS 2009, Mar 2009, York, United Kingdom. Electronic Notes in Theoretical Computer Science, Elsevier. Preprint <inria-00418560>, <https://hal.inria.fr/inria-00418560>
- ▶

# .1. Parser Generators in the Technical Space

## Grammarware

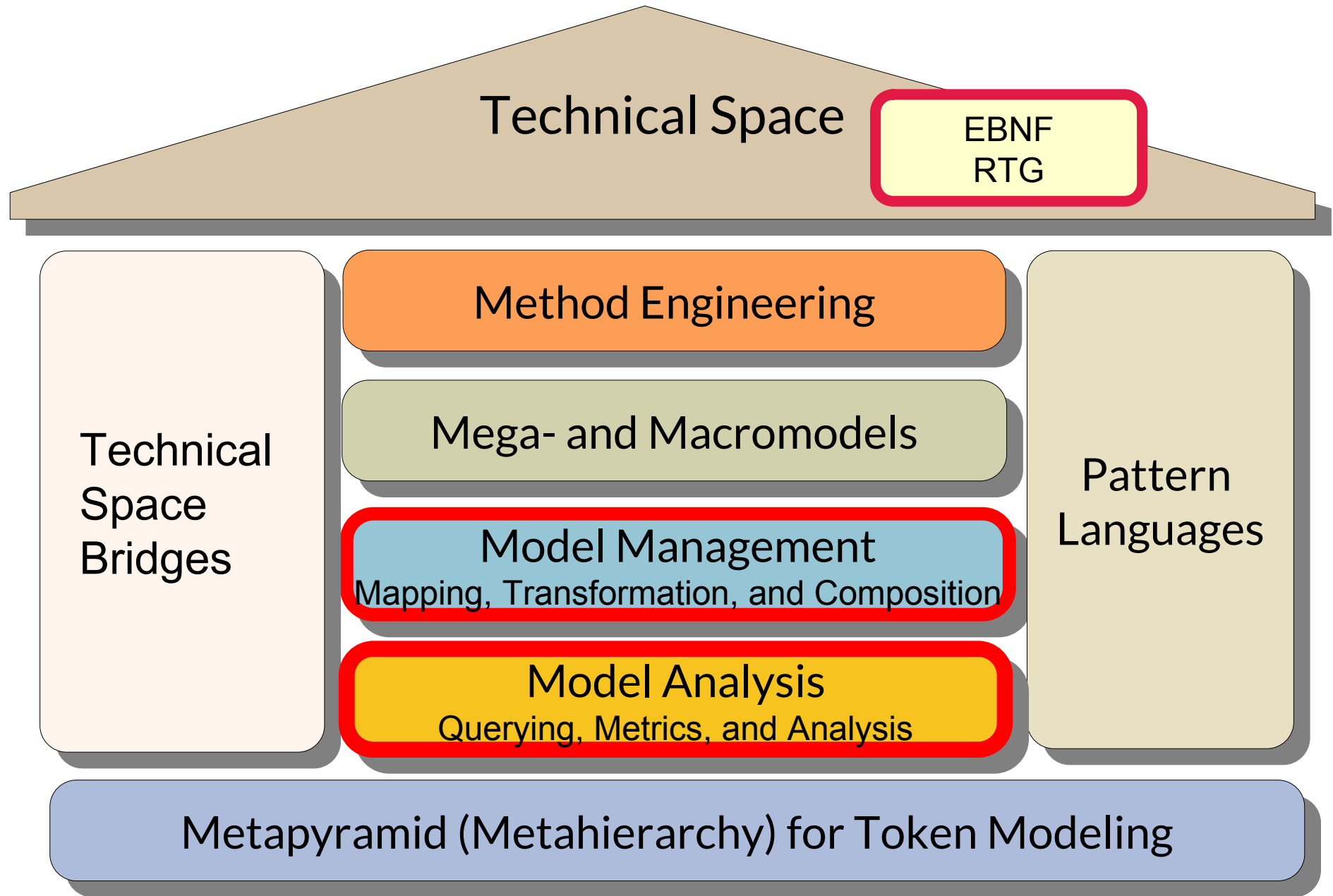
- 1) Parsing as checker for instance-of
  - 2) Antlr as example
  - 3) Example pocket computer
- ▶ Analyzing the structure of linear lists
  - ▶ And transforming them to trees



# Q10: The House of a Technical Space

4

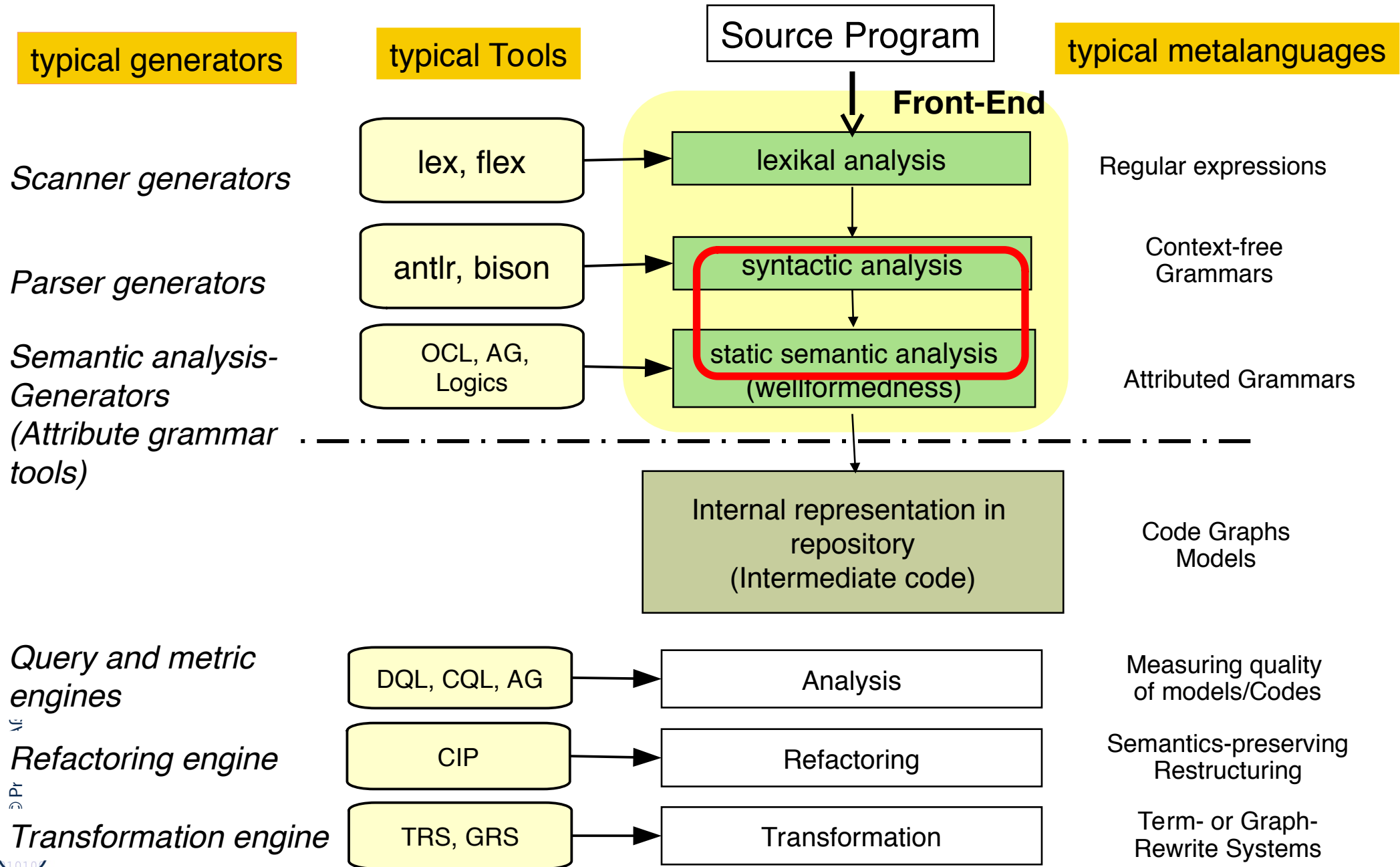
Model-Driven Software Development in Technical Spaces (MOST)



# Q7b: Phases of a Source Code Importers into a Repository and the Generating Tools

5

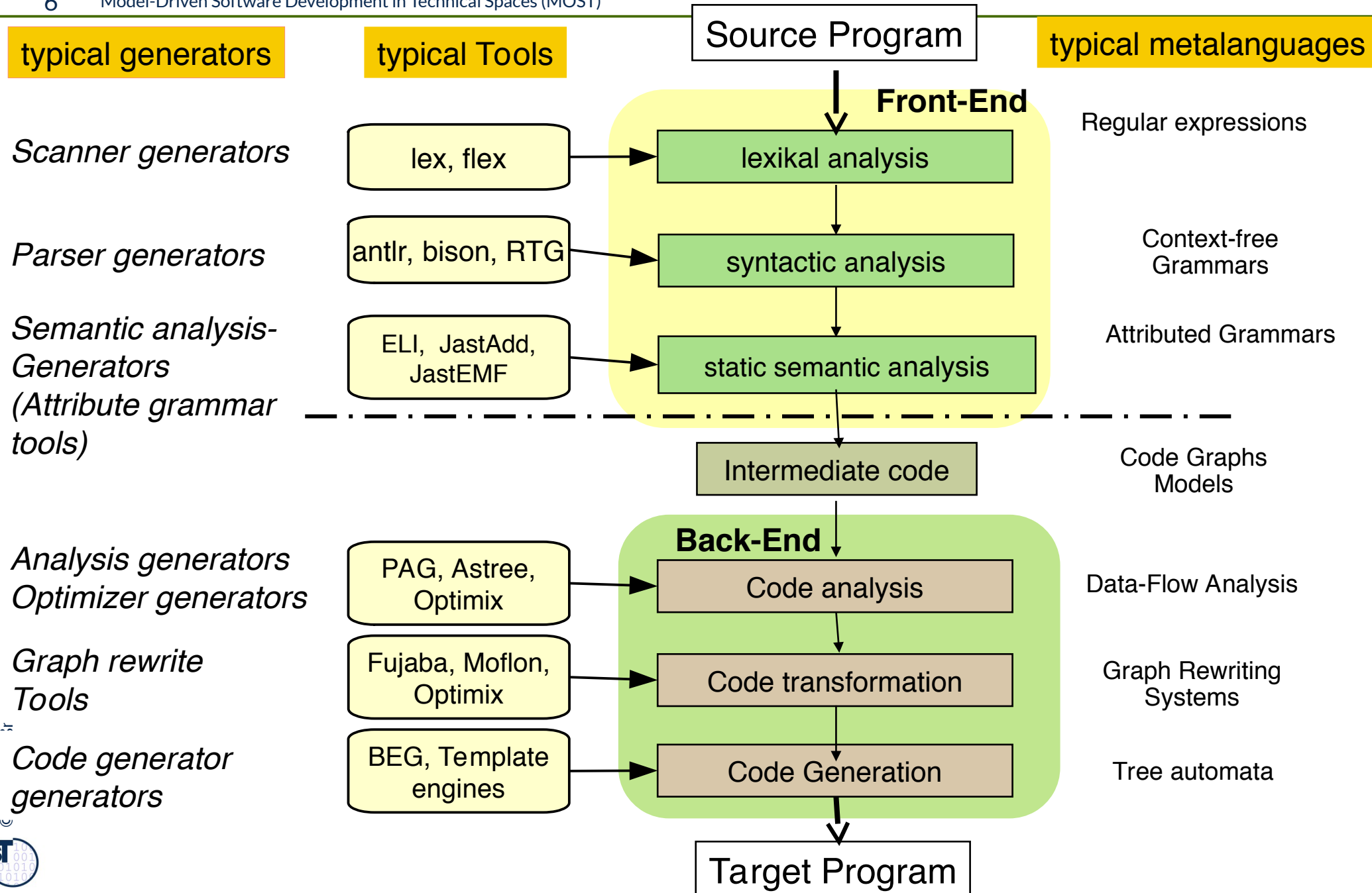
Model-Driven Software Development in Technical Spaces (MOST)



# Q8: Phases of Compilers and Software Tools and Generators

6

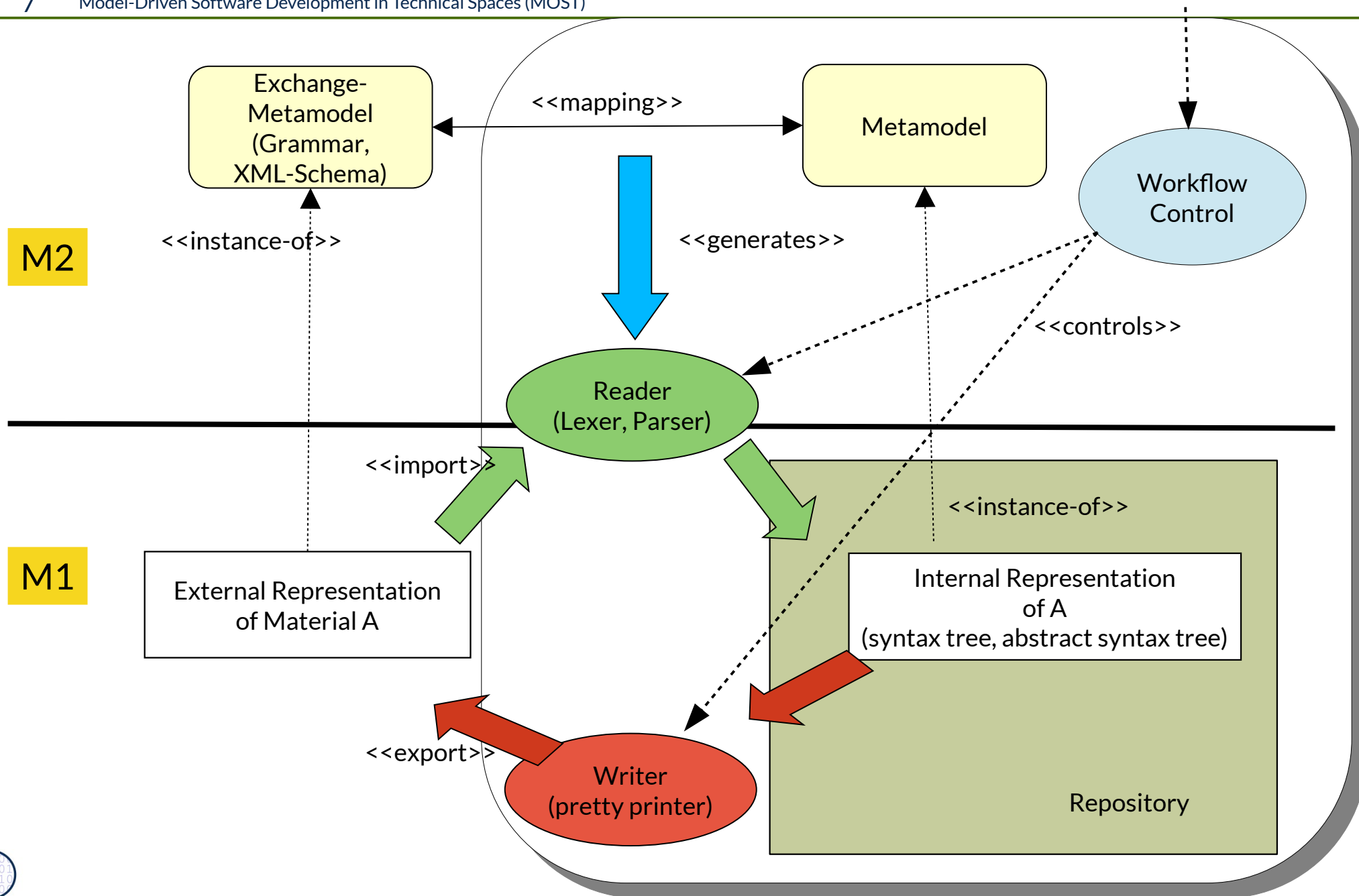
Model-Driven Software Development in Technical Spaces (MOST)



# Rpt.: Use of Generated Importers and Exporters in Modelling Tools

7

Model-Driven Software Development in Technical Spaces (MOST)



# Problem 1 of Parsing

- ▶ Parsing a program, model or document, or a material means to **recognize its context-free structure in the linear stream of characters**
  - Parsers are usually the first phases of a tool when it *imports a material*
- ▶ Parsers parse according to the *concrete syntax grammar* containing
  - Whitespace handling
  - Block handling (brackets)
  - Comment handling
- ▶ From a context-free grammar, a **parse automaton with parse rules** can be derived:
  - Address ::= Streetname StreetNumber Location
  - Location ::= Postcode Town Country
- ▶ Generates the parse rules
  - Streetname StreetNumber Location → Address
  - Postcode Town Country → Location
- ▶ The parser reads in all tokens until it can decide which rule to reduce



# String/Text Parsing with Grammars

9

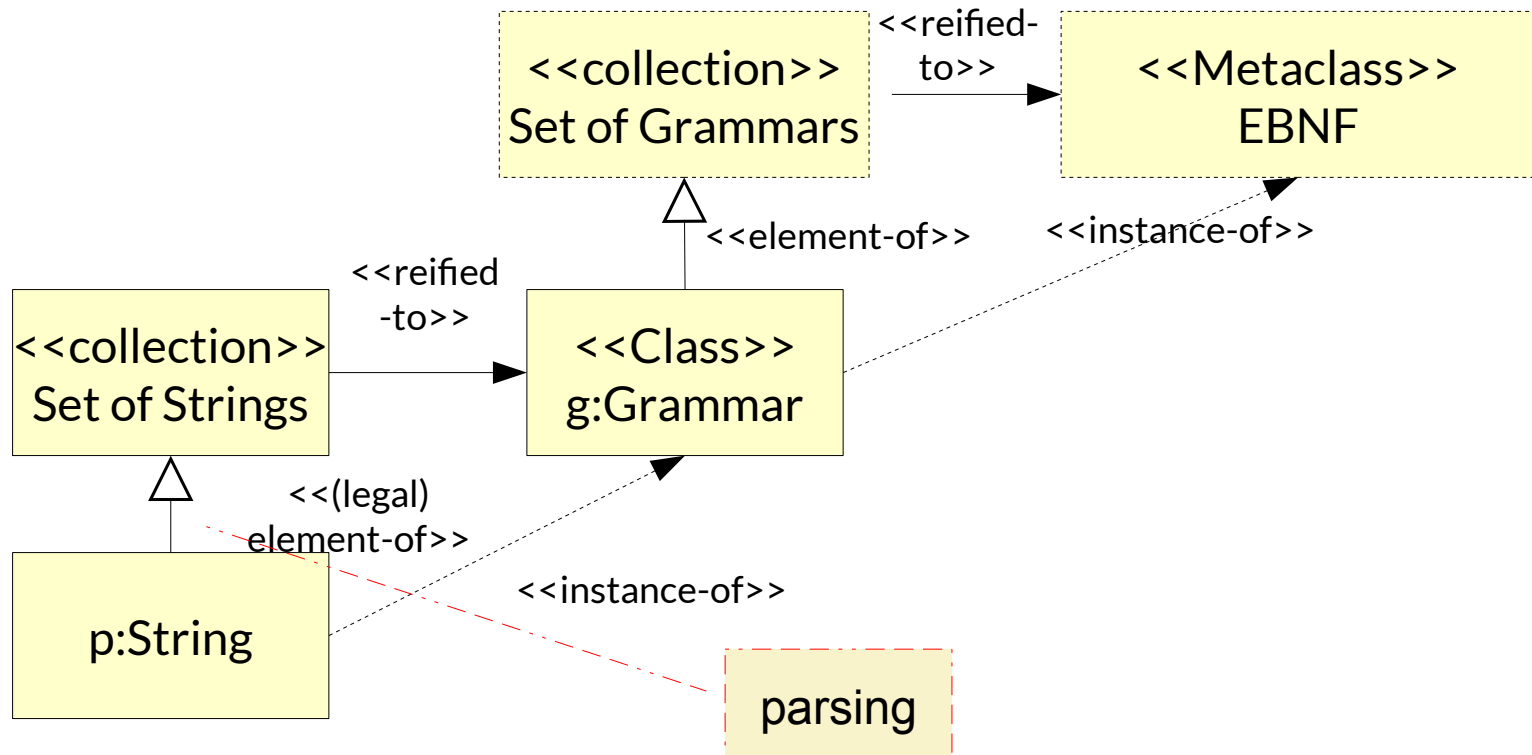
Model-Driven Software Development in Technical Spaces (MOST)

- ▶ A grammar can be used to generate a parser for strings (texts) that tests the legality of a string with the grammar
- ▶ The parser checks <<instance-of>> for the string p with regard to the grammar g

M3

M2

M1



# EBNF Rules for String Grammars

Symbol	Meaning	Example
Name (Nonterminal)	Identifier (for type or variable)	$A = B + C$
"text"	Token (text terminal)	$B ::= \text{"Town"} + R$
$=, ::=$	Consists of	$X ::= X1 + X2 + X3$
$+$ , also juxtaposition	Sequence	$X ::= X1 X2 X3$
@	Key (unique identifier)	$P = @PersonNr + N + Address$
$[... ...]$	Selection (alternative)	$P = [ P1   P2 ]$
$n \{ ... \} m$	Iteration, at least n upto m times	$B = 1 \{ C \} 10$
$n^*$	Iteration of n - arbitrarily many times	$Children ::= Name^*$
$n +$	Iteration of n at least once	$PastEmployers ::= Name +$
$( ... )$	Optional	$Address ::= Street + ( PostBox )$
$A // \text{" "}$	Sequence of A with intermittennd '"	$C = D // \text{" "}$
$* ... *$	Comment	$X = B + C * \text{text}^*$
$< a > b$	Modifier (Kommentar)	$< old > A < new > A$
SYN	Synonym für Name	$SecondName SYN SurName$

# Example: Address Grammar

- ▶ “::=” means “is-composed-of” or “is-decomposed-to”
- ▶ Every rule declares a whole-part decomposition
- ▶ Grammar declares the structure of a part list

```
Address ::= Person Company Location.  
Person ::= Title Name  
Title ::= „Dr.“ | „Prof. Dr.“ | „Mr.“ | „Ms.“  
Name ::= FirstName* LastName  
Company ::= String  
Location ::= Street StreetNumber Postcode Town  
FirstName ::= String  
LastName ::= String  
Postcode ::= 5{Digit}5  
Street ::= String  
StreetNumber ::= Integer [ String ]  
Town ::= String
```

# Example: ANTLR [www.antlr.org](http://www.antlr.org)

- ▶ Since the 90s, many parser generators have been built for C/C++
  - Cocktail's lalr, ell, lark [www.cocolab.de](http://www.cocolab.de)
  - Fnc2 (INRIA)
  - flex und bison (GNU)
  - Eli is a fast compiler generator toolset <http://eli.sf.net>
- ▶ For Java, ANTLR is popular
  - Parser class LL(k): Left-recursive grammar rules, k-lookahead for decisions
  - Generated Parser with algorithm “recursive descent”
  - [http://www.bearcave.com/software/antlr/antlr\\_expr.html](http://www.bearcave.com/software/antlr/antlr_expr.html)



- parameter\_declaration
- identifier\_list
- initializer
- initializer\_list
- type\_name
- abstract\_declarator
- direct\_abstract\_declarator
- typedef\_name

## ▼ Statement

- statement
- labeled\_statement
- expression\_statement
- compound\_statement
- statement\_list
- selection\_statement
- iteration\_statement
- jump\_statement

## ► Expression

## ► Lexer

```

compound_statement
: RCURLY declaration_list? statement_list? LCURLY
;

statement_list
: statement+
;

selection_statement
: 'if' LPAREN expression RPAREN statement ('else' statement)?
| 'switch' LPAREN expression RPAREN statement
;

iteration_statement
: 'while' LPAREN expression RPAREN statement
| 'do' statement 'while' LPAREN expression RPAREN SEMI
| 'for' LPAREN expression SEMI expression SEMI expression
;

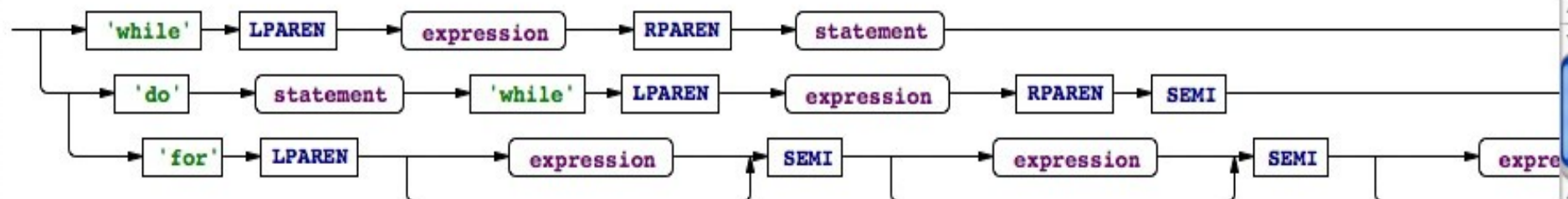
jump_statement
: 'goto' identifier SEMI
| 'continue' SEMI
| 'break' SEMI
| 'return' expression SEMI
;

```

Enter rule name:

- st
- struct\_or\_union\_specifier
- storage\_class\_specifier
- struct\_or\_union
- struct\_declaration\_list
- struct\_declaration
- struct\_declarator\_list
- struct\_declarator
- statement
- statement\_list
- string

Zoom

☐ Show NFA

Syntax Diagram

Interpreter

Debugger

Console

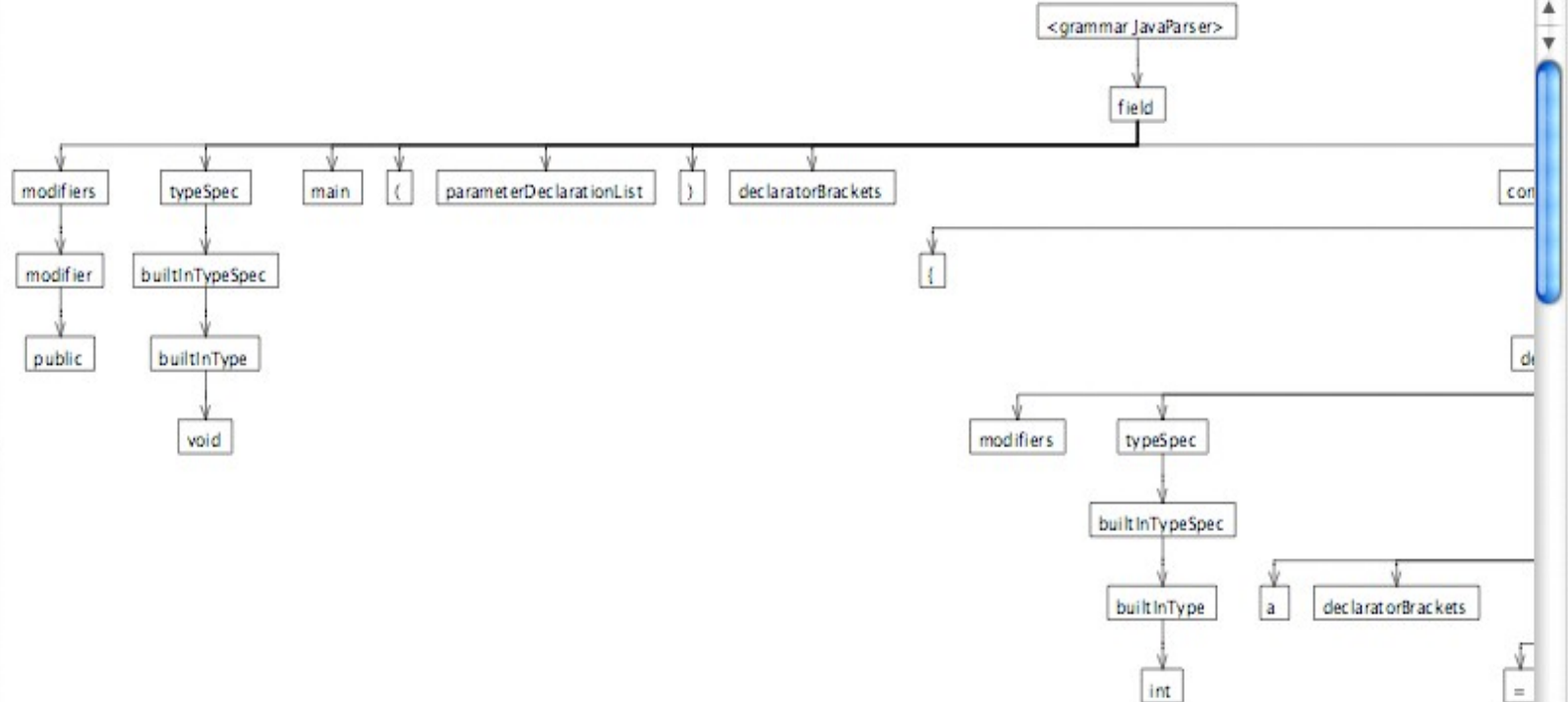


P handler  
P expression  
P expressionList  
P assignmentExpression  
P conditionalExpression

```
// the mother of all expressions  
expression  
: assignmentExpression  
;  
;
```

field

```
public void main() {  
    int a = 2+3;  
}
```



Zoom

Syntax Diagram

Interpreter

Debugger

Console



/Users/bovet/Development/Research/depot/antlr/examples-v3/java/java/java.g

interfaceBodyDeclaration  
interfaceMemberDecl  
interfaceMethodOrFieldDecl  
interfaceMethodOrFieldRest  
methodDeclaratorRest  
voidMethodDeclaratorRest  
interfaceMethodDeclaratorRest  
interfaceGenericMethodDecl  
voidInterfaceMethodDeclaratorRest  
constructorDeclaratorRest  
constantDeclarator  
variableDeclarators  
variableDeclarator  
variableDeclaratorRest  
constantDeclaratorsRest  
constantDeclaratorRest  
variableDeclaratorId  
variableInitializer  
arrayInitializer  
modifier

```
variableDeclaratorId  
: Identifier ('[' ']*'  
;  
  
variableInitializer  
: arrayInitializer  
| expression  
;  
  
arrayInitializer  
: '{' (variableInitializer ',' variableInitializer)* (''  
;  
  
modifier  
: annotation  
| 'public'  
| 'protected'  
| 'private'  
| 'static'  
| 'abstract'  
| 'final'  
| 'native'  
| 'synchronized'  
| 'transient'  
| 'volatile'  
| 'strictfp'  
;  
packageOrTypeName
```

Break on: ☐ All ☐ Location ☒ Consume ☐ LT ☐ Exception

Input

```
public class Sample {  
    public void main() {  
        System.out.println("Hello, world");  
    }  
}
```

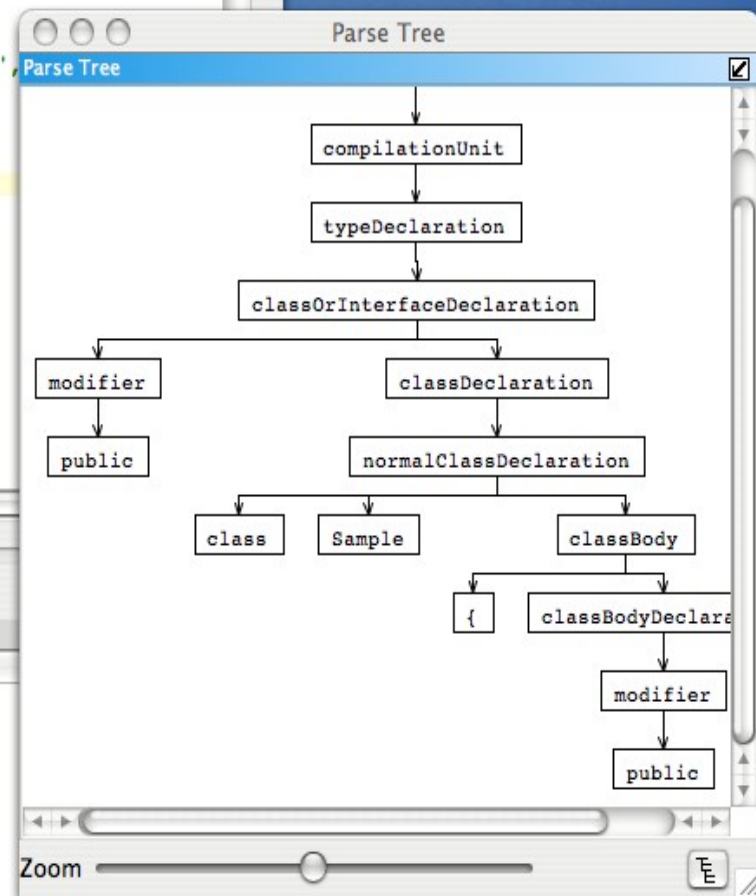
Stack

#	Rule
0	compilationUnit
1	typeDeclaration
2	classOrInterfaceDeclaration
3	classDeclaration
4	normalClassDeclaration
5	classBody
6	classBodyDeclaration
7	modifier

Input Output Parse Tree AST Stack Events

Syntax Diagram Interpreter Debugger Console

148 rules (2 warnings) 254:9 Warnings reported in console

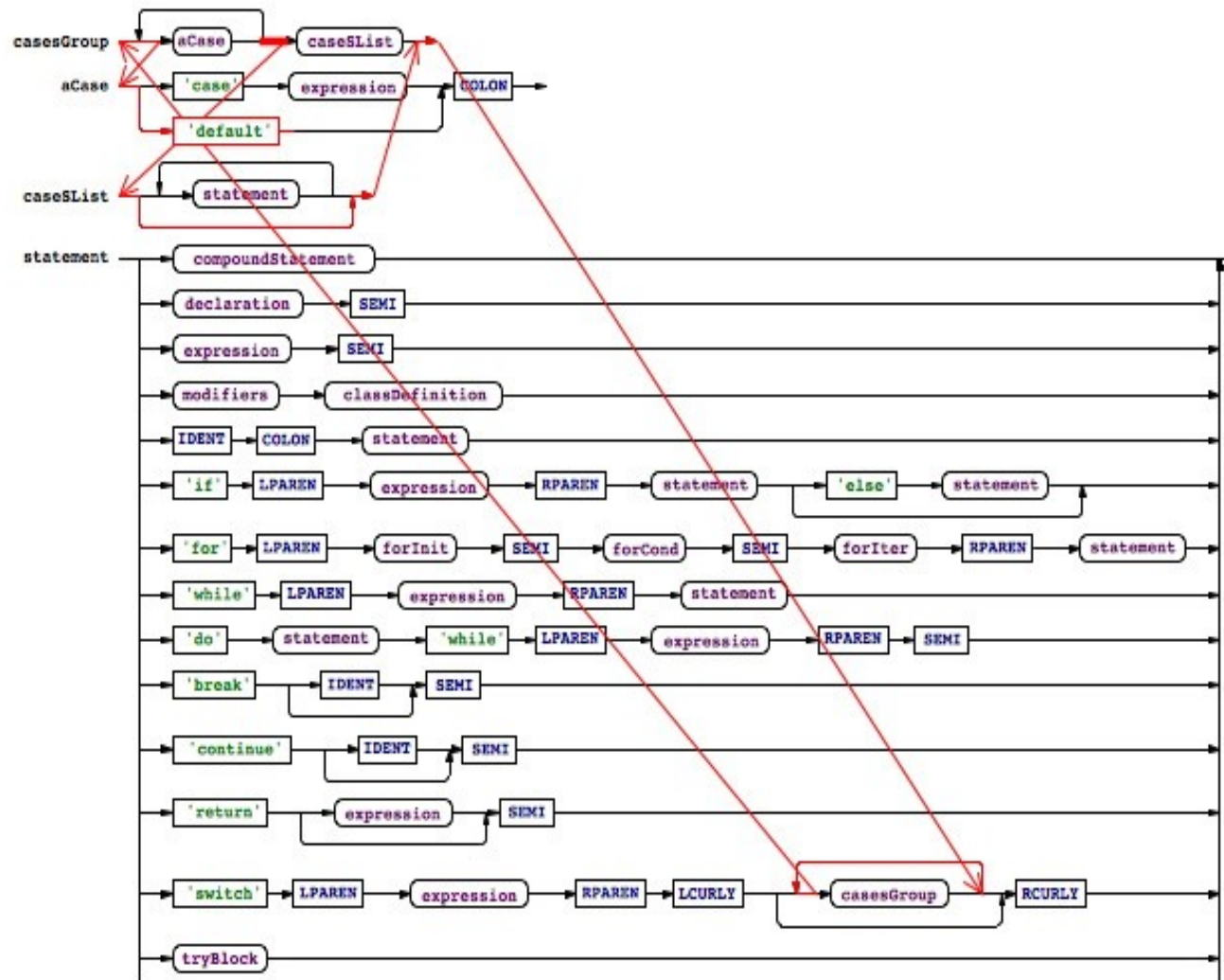




Zoom

☐ Show NFA

(1/2) Decision can match input such as "default" using multiple alternatives

Alternatives: ☐ 1 ☒ 2

Syntax Diagram

Interpreter

Debugger

Console



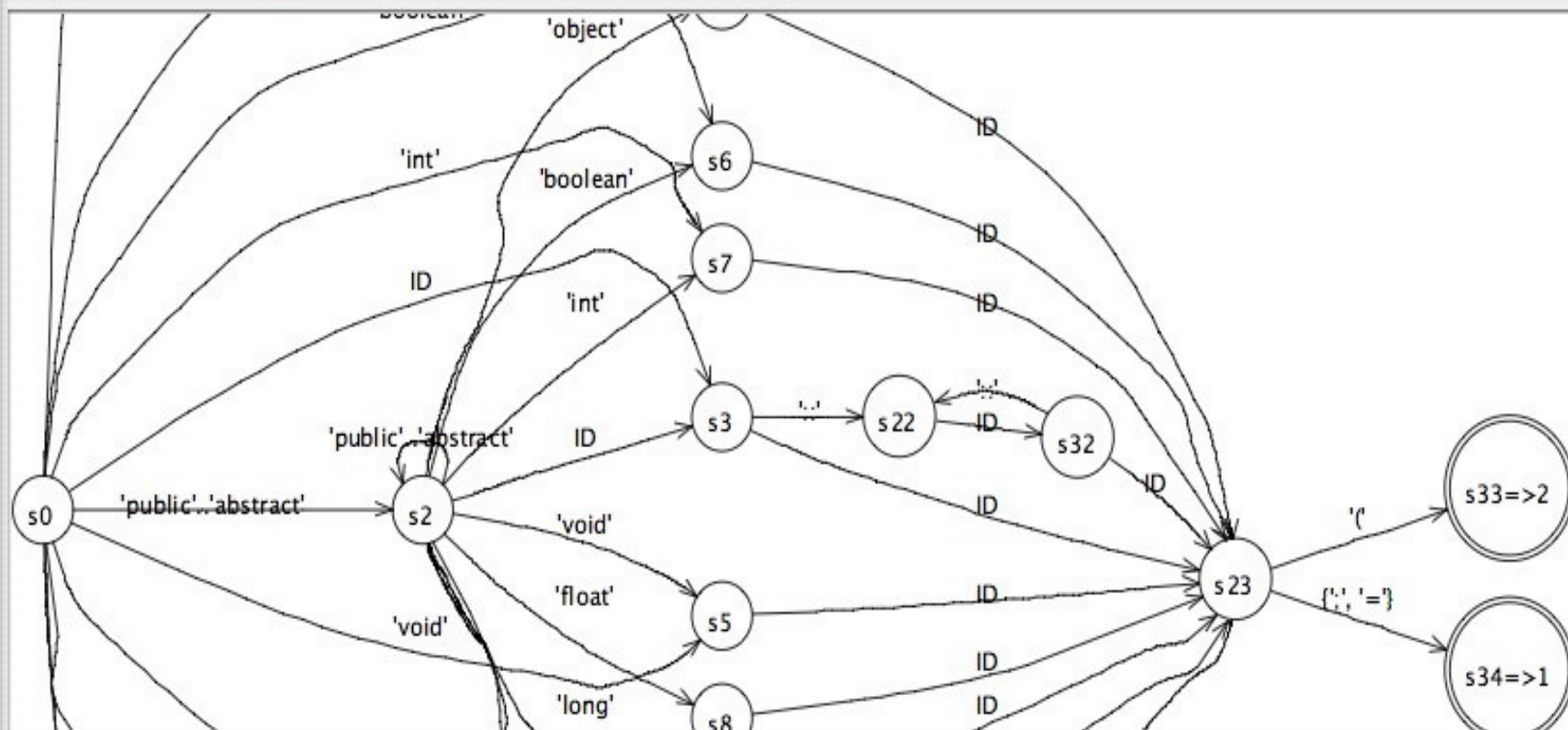


- P compilationUnit
- P packageDefinition
- P importDefinition
- P typeDefinition
- P classDefinition
- P interfaceDefinition
- P methodDefinition
- P formalArgs

```

classDefinition[MantraAST mod]
scope {
  String name;
}
: 'class' ID ('extends' sup=classname)? ('implements' i+=classname (',' i+=classname)*)?
{$classDefinition::name = $ID.text;}
{
  variableDefinition
  methodDefinition
}*
  
```

Zoom



Syntax Diagram

Interpreter

Debugger

Console

Decision 10 of "classDefinition"

59 rules (1 warnings)

56:5

## 20.1.2 Program Interpretation While Parsing

# An ANTLR Grammar for the Input Language of Pocket Calculator

- ▶ Pocket calculator interpretes the program to calculate one attribute \$value
  - Interpretation needs non-terminal attributes, which are stored on the stack of the parser
- ▶ Usually, the parse automaton with the parse rules is not shown, because it is rather complex
- ▶ Debugging a generated parser is no fun

```

grammar Expr;
@header {
package test;
import java.util.HashMap;
}
@lexer::header {package test;}
@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
prog:  stat+ ;

stat:  expr NEWLINE {System.out.println($expr.value);}
      | ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}
      | NEWLINE
      ;
expr returns [int value]
:  e=multExpr {$value = $e.value;}
  ( '+' e=multExpr {$value += $e.value;}
  | '-' e=multExpr {$value -= $e.value;}
  )*
  ;
multExpr returns [int value]
:  e=atom {$value = $e.value;} ('*' e=atom {$value *=
$value.value;}) *
  ;
atom returns [int value]
:  INT {$value = Integer.parseInt($INT.text);}
  | ID
    {
      Integer v = (Integer)memory.get($ID.text);
      if ( v!=null ) $value = v.intValue();
      else System.err.println("undefined variable "+$ID.text);
    }
  | '(' e=expr ')' {$value = $e.value;}
  ;
// lexical rules
ID  :  ('a'..'z'|'A'..'Z')+ ;
INT :  '0'..'9'+ ;
NEWLINE: '\r'? '\n' ;
WS   :  (' '|'\t')+ {skip();} ;

```

# Control of a Generated Java Parser

```
import org.antlr.runtime.*;
public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ExprLexer lexer = new ExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExprParser parser = new ExprParser(tokens);
        parser.prog();
    }
}
```

/Users/bovet/ Grammars/Demo/Expr.g

prog  
stat  
expr  
multExpr  
atom  
ID  
INT  
NEWLINE  
WS

```

grammar Expr;

@header {
package test;
import java.util.HashMap;
}

@lexer::header {package test;}

@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}

prog: stat+ ;

stat: expr NEWLINE {System.out.println($expr.value);}
    | ID '=' expr NEWLINE
      {memory.put($ID.text, new Integer($expr.value));}
    | NEWLINE
    ;

expr returns [int value]
: e=multExpr {$value = $e.value;}
( '+' e=multExpr {$value += $e.value;}
  '-' e=multExpr {$value -= $e.value;}
)*

```

prog Line Endings: Unix (LF) Ignore rules: WS Guess

2+3\*4

Zoom

Syntax Diagram Interpreter Debugger Console

9 rules 1:1 Writable

/Users/bovet/ Grammars/Demo/Expr.g

prog  
stat  
expr  
multExpr  
atom  
ID  
INT  
NEWLINE  
WS

```

expr returns [int value]
: e=multExpr {$value = $e.value;}
( '+' e=multExpr {$value += $e.value;}
| '-' e=multExpr {$value -= $e.value;}
)*

multExpr returns [int value]
: e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})

atom returns [int value]
: INT {$value = Integer.parseInt($INT.text);}
| ID
{
  Integer v = (Integer)memory.get($ID.text);
  if ( v!=null ) $value = v.intValue();
  else System.err.println("undefined variable "+$ID.text);
}
| '(' e=expr ')' {$value = $e.value;}

ID : ('a'..'z'|'A'..'Z')+;
INT : '0'..'9'+;
NEWLINE: '\r'? '\n';

```

Break on: ☐ All ☒ Location ☐ Consume ☐ LT ☐ Exception

Input: 2

Parse Tree:

```

graph TD
  root --> prog
  prog --> stat
  stat --> expr
  expr --> multExpr
  multExpr --> atom
  atom --> 2

```

Stack:

#	Rule
0	prog
1	stat
2	expr
3	multExpr
4	atom

Zoom: [Slider]

Input Output Parse Tree AST Stack Events

Syntax Diagram Interpreter Debugger Console

9 rules 35:13 Writable



Debugger window showing the source code and the parse tree for the expression "2 + 3 \* 4".

**Source Code:**

```

grammars/Expr.g

@header {
package test;
import java.util.HashMap;
}

@lexer::header {package test;}

@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}

prog: stat+ ;

stat: expr NEWLINE {System.out.println($expr.value);}
    | ID '=' expr NEWLINE
    {memory.put($ID.text, new Integer($expr.value));}
    | NEWLINE ;

expr returns [int value]
: e=multExpr {$value = $e.value;}
| '+' e=multExpr {$value += $e.value;}
| '-' e=multExpr {$value -= $e.value;}
)*

```

**Parse Tree:**

```

graph TD
    root --> prog
    prog --> stat
    stat --> expr
    expr --> multExpr1[multExpr]
    expr --> plus["+"]
    expr --> multExpr2[multExpr]
    multExpr1 --> atom1[atom]
    atom1 --> 2
    multExpr2 --> atom2[atom]
    atom2 --> 3
    multExpr2 --> star["*"]
    star --> atom3[atom]
    atom3 --> 4

```

**Debugger Controls:**

- Break on: ☐ All ☒ Location ☐ Consume ☐ LT ☐ Exception
- Break on Terminate
- Stack: # Rule
- Zoom: [Slider]
- Input Output Parse Tree AST Stack Events
- Syntax Diagram Interpreter **Debugger** Console

**Status Bar:** 9 rules 15:15 Writable



# Applications of String Grammars

Everything that has to do with *ordered strings*:

- ▶ Protocol checking in component-based systems (protocol automata and grammars)
- ▶ Document processing
- ▶ Matching text patterns and data mining in files, emails, streams
- ▶ Communication in multi-agent systems

## 20.2 Regular Tree Grammars

- String grammars are for structuring sequences
- RTG are for specifying trees, syntax trees and abstract syntax trees
- A RTG does not care about concrete syntax

- ▶ String Grammars assume:
  - Sequence of words
  - Implicit syntax tree, because non-terminals specify it implicitly
- ▶ Regular Tree Grammars specify the tree directly, with tree node constructors
- ▶ ENBF-rule for Tree Grammar Rule:

TreeNode → constructor '(' Treenode // ',' ')'

- ▶ Example:

Model → ModelElements \*

// Regular Tree Grammar from Stratego

**regular tree grammar TIL**

start Program

**productions**

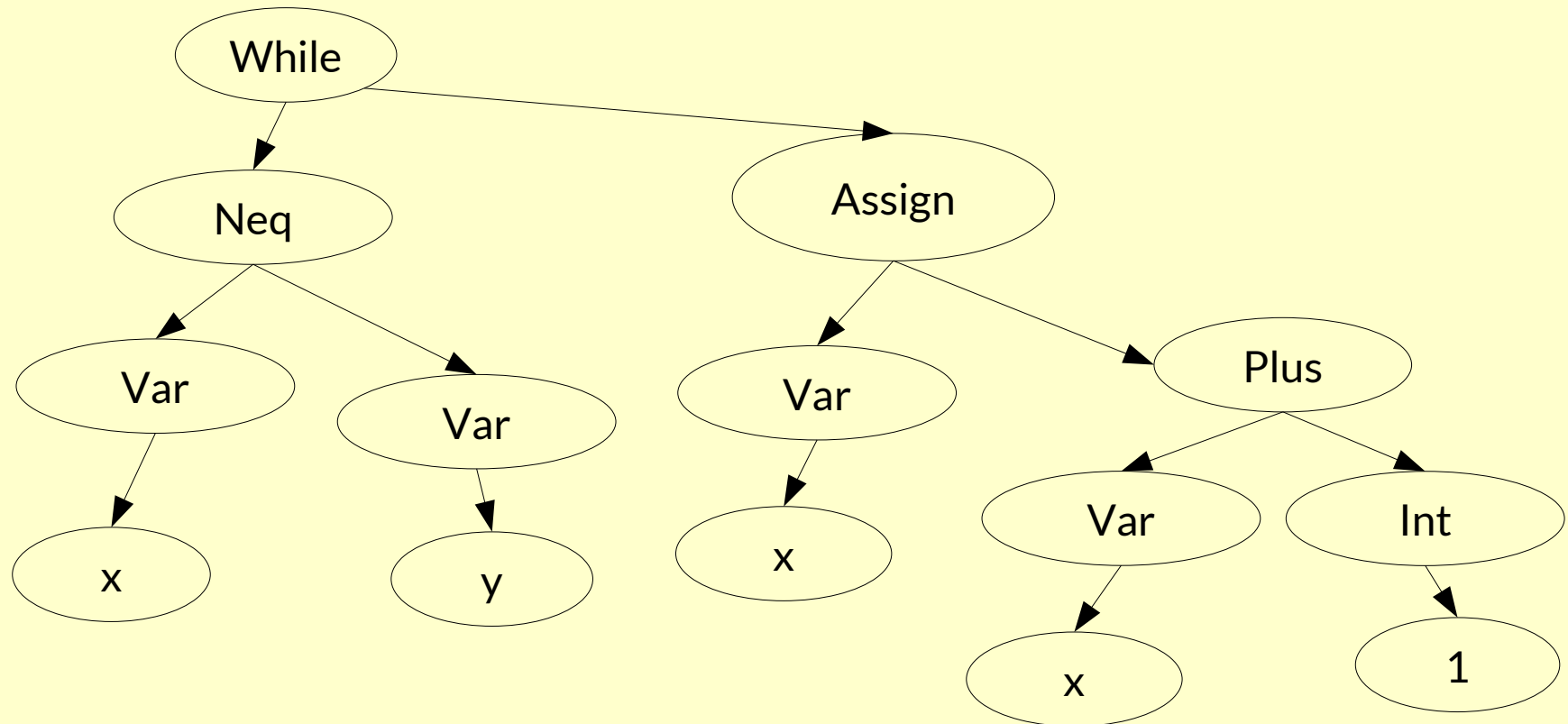
```
Program    -> Program(ListStarOfStat0)
Stat       -> ProcCall(Id,ListStarOfExp0)
Exp        -> FunCall(Id,ListStarOfExp0)
Stat       -> For(Id,Exp,Exp,ListStarOfStat0)
Stat       -> While(Exp,ListStarOfStat0)
Stat       -> IfElse(Exp,ListStarOfStat0,ListStarOfStat0)
Stat       -> IfThen(Exp,ListStarOfStat0)
Stat       -> Block(ListStarOfStat0)
Stat       -> Assign(Id,Exp)
Stat       -> DeclarationTyped(Id,Type)
Stat       -> Declaration(Id)
ListStarOfStat1 -> Stat // ','
ListStarOfStat0 -> Stat *
Type       -> TypeName(Id)
Exp        -> Or(Exp,Exp) | And(Exp,Exp)
Exp        -> Geq(Exp,Exp) | Eq(Exp,Exp) | Neq(Exp,Exp)
Exp        -> Gt(Exp,Exp) | Lt(Exp,Exp) | Leq(Exp,Exp)
Exp        -> Sub(Exp,Exp) | Add(Exp,Exp)
Exp        -> Mod(Exp,Exp) | Div(Exp,Exp) | Mul(Exp,Exp)
Exp        -> String(String)
Exp        -> Int(Int) | Var(Id)
Exp        -> False() | True()
StrChar    -> <string>
String     -> <string>
Int        -> <string>
```

# Correct Instance?

28

Model-Driven Software Development in Technical Spaces (MOST)

```
// Example: applying TIL grammar to a fragment  
ExecuteGrammar(TIL,  
  While(Neq(Var(x),Var(y)), Assign(Var(x),Plus(Var(x),Int(1) ) )  
)
```

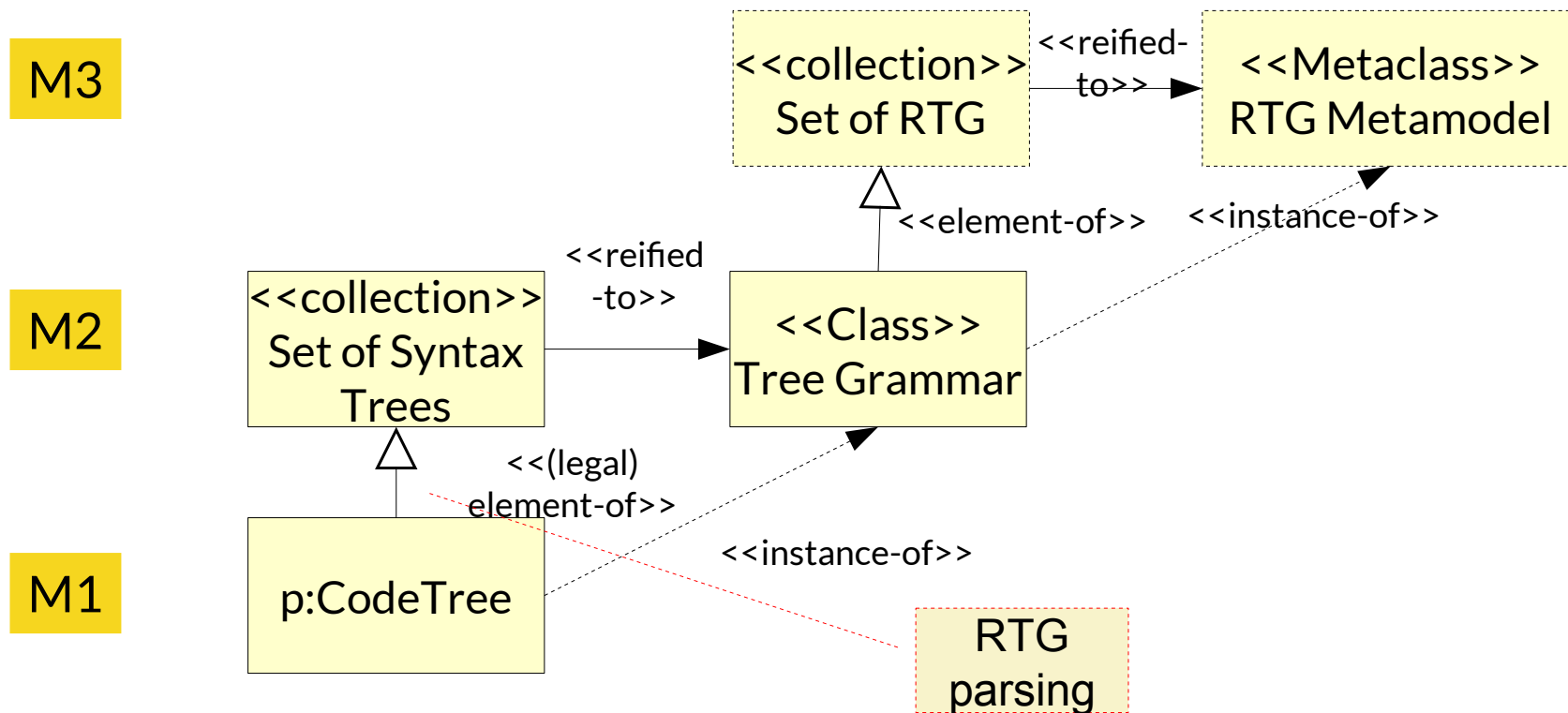


# Tree Parsing with RTG

29

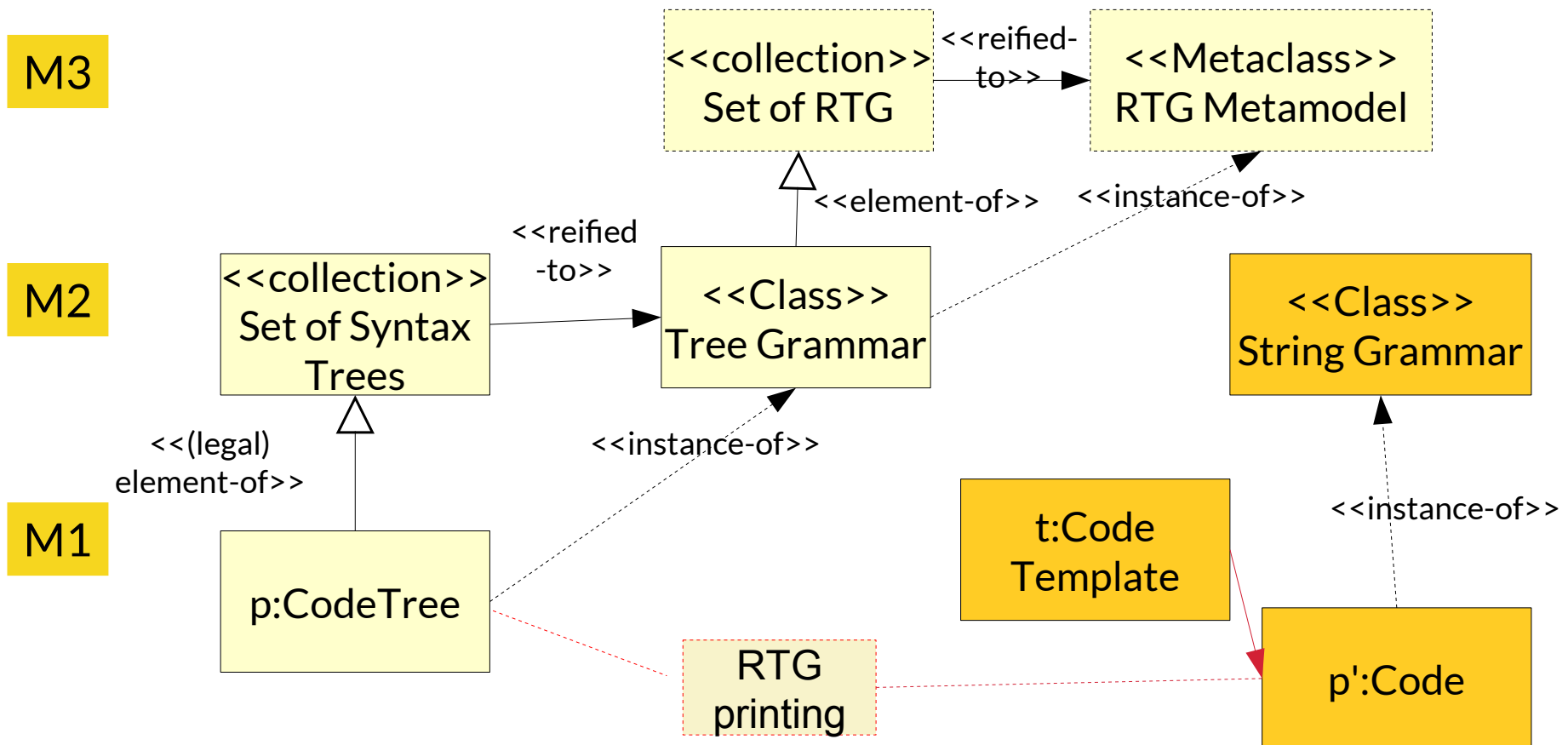
Model-Driven Software Development in Technical Spaces (MOST)

- ▶ An RTG can be used to generate a **tree parser** that tests the legality of a code tree with a tree grammar



# Tree Pretty-Printing with RTG

- ▶ An RTG can be used to generate a **tree pretty-printer** that prints the nodes of a tree recursively
- ▶ If  $p$  is a code tree,  $p'$  can be its pretty-printed copy, derived of a code template  $t$
- ▶ Exercise: write a pretty-printer for the RTG TIL



## 20.3. Tree Construction as a Mapping between Parse Grammar and Tree Grammar

- Full parser also build syntax trees – Design Pattern Builder

# Tree Construction While Parsing

- ▶ Parsing recognizes the tree structure of a text – **however, the syntax tree must be built**
- ▶ After parsing, the parser creates an (**abstract**) *syntax tree*, i.e., builds up a tree with regard to a ***regular tree grammar of the abstract syntax***
  - Recognized nonterminals have to be mapped
  - Tokens, keywords, comments, layouts have to be omitted
  - **Tree building:** Treenodes have to be allocated
- ▶ This **CS-AS mapping (from concrete to abstract syntax)** is created by hand in *side actions* of the parser
- ▶ For simple languages, parsers and tree constructors are no longer written by hand, but generated from *grammars in EBNF*
  - **Parser** recognizes the structure of the text (“Zerteiler des Textes”)
  - **Tree builder** generates an abstract syntax tree
  - **CS-AS-mapping** creates AS nodes after recognition of CS nonterminals



# Constructing a Tree Grammar fitting to the String Grammar of Office DSL

```
//*****  
// Copyright (c) 2006-2010  
// Software Technology Group, Dresden University of Technology  
//  
// All rights reserved. This program and the accompanying materials  
// are made available under the terms of the Eclipse Public License v1.0  
// which accompanies this distribution, and is available at  
// http://www.eclipse.org/legal/epl-v10.html  
//  
// Contributors:  
//   Software Technology Group - TU Dresden, Germany  
//   - initial API and implementation  
// *****/  
SYNTAXDEF office  
FOR <http://emftext.org/office>  
START OfficeModel  
OPTIONS {  
    licenceHeader = "../org.dropsbox/licence.txt";  
    generateCodeFromGeneratorModel = "true";  
    disableLaunchSupport = "true";  
    disableDebugSupport = "true";  
}  
RULES {  
    OfficeModel ::= "officemodel" name[] "{" elements:Element* "}" ;  
  
    Elements ::= Office | Employee;  
    Office ::= "office" name[];  
  
    Employee ::= "employee" name[]  
                "works" "in" worksIn[]  
                "works" "with"  
                worksWith[] ("," worksWith[])* ;  
}
```

# .CS Grammar Plus Mapping to RTG (Abstract Syntax Tree)

34

Model-Driven Software Development in Tech

- ▶ CS-AS mapping works via side actions of the grammar rules
- ▶ Tree is built while returning from recursive descent

```
//*****  
// Copyright (c) 2006-2015 under EPL  
// Software Technology Group, Dresden University of Technology  
// http://www.eclipse.org/legal/epl-v10.html  
//  
//*****/  
SYNTAXDEF office FOR <http://emftext.org/office>  
TREENODES { // RTG  
    START NodeOfficeModel  
    NodeOfficeModel →  
NodeOfficeModel(name:String,elements:Element *)  
    Element → Office(name:String) |  
                Employee(name:String, worksIn:String,  
worksWith:String *)  
}  
START OfficeModel  
RULES {  
    OfficeModel returns [NodeOfficeModel root]  
        ::= "officemodel" name[] "{" elements:Element * "  
    { root = NodeOfficeModel()  
      root.name = name; root.elements = assemble elements; };  
    Elements returns [Element retval]  
        ::= Office { retval = Office.val; }  
        | Employee { retval = Employee.val; };  
    Office returns [Element retval]  
        ::= "office" name[] { retval = Office(name); };  
    Employee returns [Element retval]  
        ::= "employee" name[] "works" "in" worksIn[]  
                "works" "with"  
                worksWith[] ("," worksWith[])*  
    { retval = Employee(name,worksIn,assemble worksWith);  
    };  
}
```

# Modeling Tools need Several Languages and DSL

- ▶ Bidirektional mapping between technical space “Grammarware” and another one, e.g., “Treeware”, “Link-TreeWare”, “XMLWare”, or “Modelware”

How can an MDSD Tool work flexibly  
with several *textual* languages?

Generating parsers and tree builders from string grammars and  
RTG

... and generate from the RTG ..

Pretty printers (Code generators)

# Example: EMFText: EMOF and RTG

- ▶ EMFText uses the parser generator ANTLR to generate parsers
- ▶ The EMOF metamodels have a primary tree that can be written down as RTG
- ▶ Mapping concrete to abstract syntax:
  - EBNF Grammar and the (implicit) RTG of the corresponding EMF metamodel are mapped *automatically* to each other (language mapping)
- ▶ For pretty printer generation, EMFText uses template-based code generation for the (implicit) RTG

## 10.4 Text and Tree Algebrae

# Composition with Composition Systems



A Venn diagram consisting of three overlapping circles. The top-left circle is light green and labeled 'Component Model'. The top-right circle is light yellow and labeled 'Composition Technique'. The bottom circle is a medium green and labeled 'Composition Language'. The circles overlap in various combinations, representing the integration of these three concepts.

**Component Model**

**Composition Technique**

**Composition Language**

# Composition with Algebrae in Mathematics

**Component Model:**

Set as Carrier

**Composition Technique:**

Algebra Operators  
(union, unify, etc.)

**Composition Language:**

Functional Language,  
Lambda-Calculus

# One-sorted Algebra on Texts

- ▶ A **one-sorted algebra** is a set of operators on a carrier set (Trägermenge) of a type (a sort). Example: Texts, sequences of lines of characters
- ▶ Operations
  - E.g., the parser splits texts into lines, separated by newline characters
  - A text template is expanded by a padding (Füllsel)
- ▶ The UNIX Programmers Workbench is built on an algebra on texts:
  - `diff: Text x Text → edit-sequence` (for a transformation)
  - `cmp: Text x Text → Boolean`
  - `patch: Text x edit-sequence → Text`
  - `diff3: mine:Text x older:Text x yours:Text → edit-sequence`
  - `split: Text x Split-char → Text*`
  - `match/grep: Text x Pattern → Text*`
  - `check-property: Text x Pattern → Boolean`
  - `is-consistent: Text x Text → Boolean`
  - `format: Text → Text`
  - `expand: Text-template x padding:Text* → Text`



# CSV: A One-Sorted Algebra on Ascii-Tables

- ▶ Tables consist of sequences of lines, split into columns by a column-separator (TAB, COMMA, | )
  - .csv-tables (comma separated values)
  - html-tables, tex-tables
- ▶ rdb is a command tool suite on an algebra on tables:
  - Diff: `table x table → edit-sequence`
  - Cmp: `File x File → Boolean`
  - Patch: `table x edit-sequence → table`
  - Diff3: `mine:table x older:table x yours:table → edit-sequence`
  - split: `table x Splitzeichen → table*`
  - match: `table x Pattern → table*`
  - check-property: `table x Pattern → Boolean`
  - is-consistent: `table x table → Boolean`
  - join, sort, group-by...
  - format: `table → table`
  - expand: `table-template x table* → table`

- ▶ Operations in a simple algebra on trees:
  - `Diff: tree x tree → edit-sequence`
  - `Cmp: File x File → Boolean`
  - `Patch: tree x edit-sequence → tree`
  - `Diff3: mine:tree x older:tree x yours:tree → edit-sequence`
  - `split: tree x node → upper:tree x lower:tree`
  - `match: tree x Pattern → tree*`
  - `check-property: tree x Pattern → Boolean`
  - `is-consistent: tree x tree → Boolean`
  - `Unify: tree x tree → tree`
  - `format: tree → tree`
  - `expand: tree-template x padding:tree* → tree`

## 20.5 Port-Graph Algebrae on Fragments

Invasive Software Composition is a general, typed templating technique for all languages

- ... based on port-graph algebrae
- ... with Graybox Components
- ... preview onto the summer (CBSE course)

# “Invasive” Composition (Typed Templating) with Port-Graph Algebrae

**Component Model:**

**Fragment Components and  
their Ports (Slots and  
Hooks)**

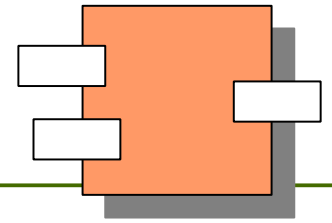
**Composition Technique:**

**Hook Transformation**

**Composition Language:**

**Standard Languages**

# Invasive Composition as Hook Transformations



45

Model-Driven Software Development in Technical Spaces (MOST)

Fragment Component:  
Molecule in a Port-Graph

Change point

Port

Composer

Invasively transformed tags

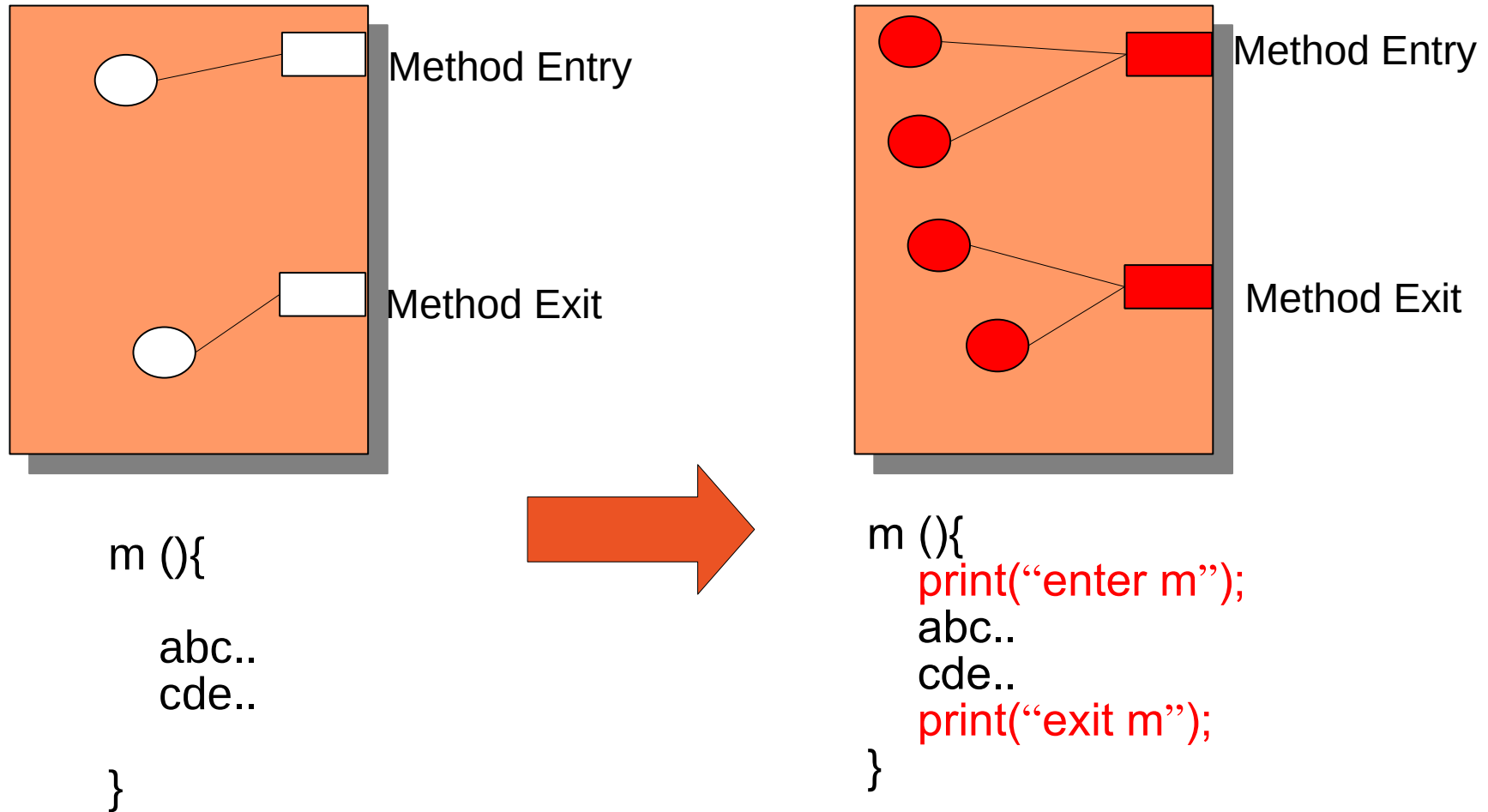
- ▶ A **port graph** is a graph in which each node (**molecule**) has a set of *ports*
- ▶ A **fragment component** is a molecule with ports (slots, hooks, query points) related to change points

**Invasive Composition**  
**adapts** and **extends**  
**fragment components**  
**at ports (slots, hooks, query-points)**  
**by composition operators**

# Binding Implicit Hooks with Fragments

46

Model-Driven Software Development in Technical Spaces (MOST)



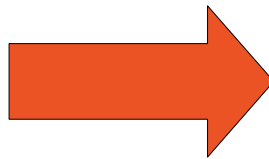
```
component.findHook(„MethodEntry“).extend("print(\\\"enter m\\\")");  
component.findHook(„MethodExit“).extend("print(\\\"exit m\\\")");
```

# Partial Parsing of Fragment Components

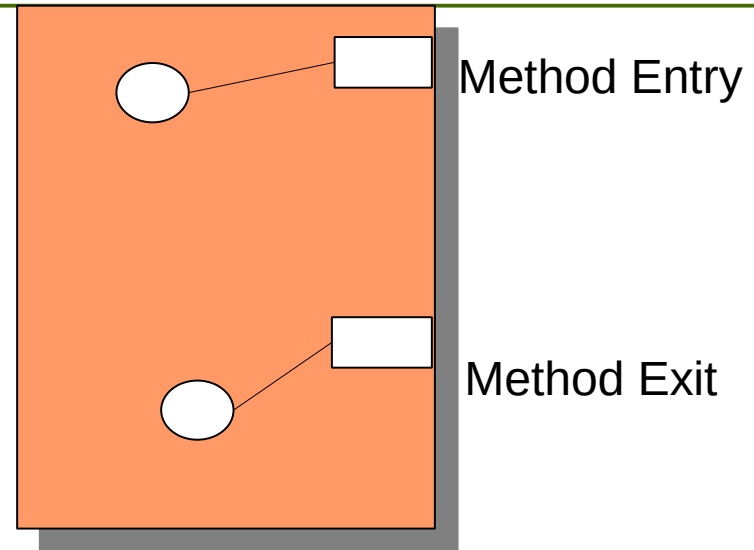
47

Model-Driven Software Development in Technical Spaces (MOST)

```
m (){  
    abc..  
    cde..  
}
```



```
m (){  
    // Method Entry  
    abc..  
    cde..  
    // Method Exit  
}
```

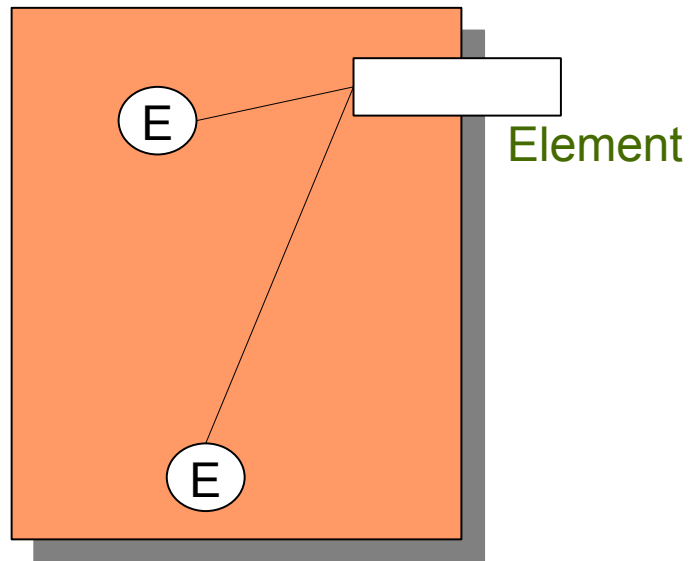


```
Component = compositionSystem.partialParser(„m (){ abc.. cde.. }“);
```

# Binding Declared Hooks with Fragments

48

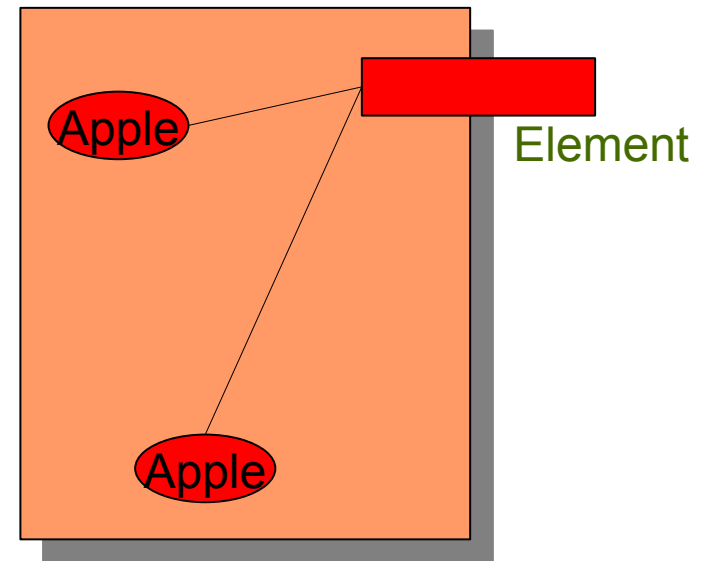
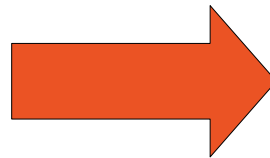
Model-Driven Software Development in Technical Spaces (MOST)



```
List(E) list;
```

```
....  
list.add(new E());
```

```
...
```



```
List(Apple) list;
```

```
....  
list.add(new Apple());
```

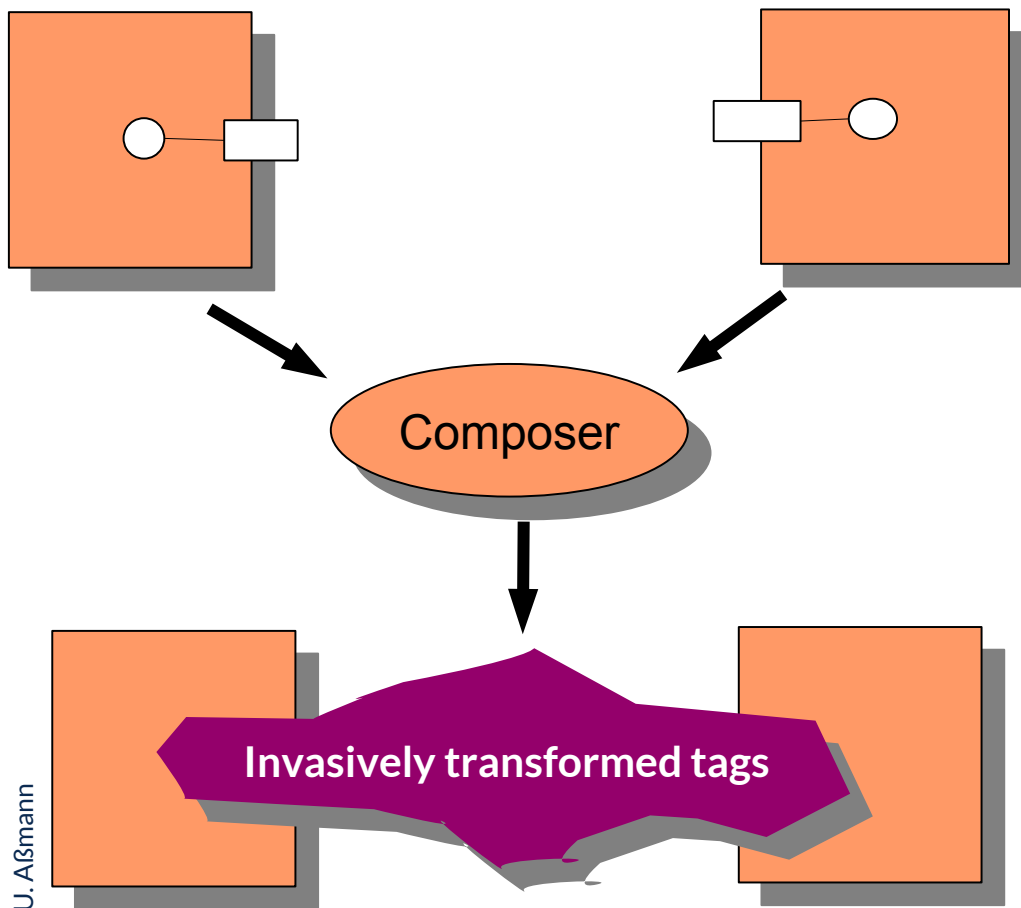
```
...
```

```
box.findHook(„Element“).bind(“Apple”);
```



# Invasive Composition as Hook Transformations

- ▶ Invasive Composition works uniformly on
  - For all languages
  - For declared hooks and implicit hooks
- ▶ Allows for unification of
  - Inheritance
  - Views
  - Aspect weaving
  - Parameterization
  - Role model merging



## Simple composition operators

- ▶ **bind** hook (parameterize)
  - For generic programming
- ▶ **rename** component, rename hook
- ▶ **remove** value from hook (unbind)
- ▶ **extend** component or hook extensions
- ▶ **copy** fragment component

## Compound composition operators

- ▶ **inheritance** from component
  - For object-oriented programming
- ▶ **view** of component
  - view-based programming
- ▶ **connect** hook 1 and 2
  - For connector-based programming
- ▶ **distribute** component over other component
  - For aspect weaving

## 20.6 Pseudocode and Markup Languages

<http://en.wikipedia.org/wiki/Pseudocode>

- ▶ **Pseudocode** consists of *structured text with keywords and blocks*, z. B. **seq**, **endseq**, **if**, **then**, **else**, **endif**, **while**, **endwhile**, **call**, **action**, **stop**,...
  - Natural text is enclosed as comment, but ignored
- ▶ For pseudocode, grammars can be constructed:
  - Syntax checking with ***island parsing***
  - An ***island grammar*** contains
    - “islands” for the keywords and structure
    - “water” for the free-form text
- ▶ Tool support:
  - Code generation (code templates and comments)
  - Documentation generation (structograms, LaTeX document generation)

- ▶ In pseudocode, structure can be recognized (as islands in the free-form water)
- ▶ Pseudocode can recognize names and do a name analysis:
  - Title of procedures, classes, and processes
  - Types from the data dictionary, Local names
- ▶ Pseudocode can define macros

```
process empfangen_Patient 1.3.1
for &Patient
    with >Bestelldatum = Datum in &Termine und >Beschwerden
    if Name*des Patienten* in &Patient
    else "aktualisieren_Patient 1.1"
    if keine >Beschwerden und >Bestelldatum unguelteig
        then „vergeben_Termin 1.2“
    else Uebernahme Patientendaten aus &Patient
    alle Unterlagen fuer Arzt aufbereiten
    <Aufnahme Name*des Patienten* in &Warteliste
    if @Bestdat+Zeit = Kalenderdatum + Uhrzeit
    then Terminpatient Platz m+1*
        vorhergehender Terminpatient m*
    else Platz n+1*n Anzahl aller Patienten im Wartezimmer*
```

# Structural Skelecon of Pseudocode (2)

```
action empfangen_Patient
  while (Patienten oder Praxisoeffnung)
    seq Eingabe >Bestelldatum, >Beschwerden
    if (@Bestdat+Uhrzeit enth. &Termine)
      then Bestellpatient
    else if (@Gebdatum+Name enth. &Patient)
      then ziehen Patientenakte
      else call aktualisieren_Patientendaten
    endif
    if (>Beschwerden <> 0*vorhanden*)
      then Unbestellter_Patient
      else call vergeben_Termin
    endif endif
    Aufbereiten aller Unterlagen fuer Arzt endseq
    if (Bestellpatient)
      then <Aufnahme Platz m+1 in &Warteliste
      else <Aufnahme Platz n+1 in &Warteliste
    endif endwhile
  stop
```

- ▶ Markup languages structure pseudocode with **markup tags**.

```
\documentclass{article}

\title{My first Document}
\author{John Doe}
\usepackage[english]{babel}

\begin{document}

\maketitle

Hello World! My name is John Doe.

\emph{Next paragraph has to be written.}

\end{document}
```

► LaTeX-distributions have good style packages for pseudocode:

- `algorithms.sty`
- `\usepackage{algpseudocode}`
- `\usepackage{algorithmicx}`
- `listings.sty`

► See also ELAN, the semi-natural programming language

- <http://de.wikipedia.org/wiki/ELAN>
- Part of OS L3, predecessor of L4

```
PACKET stack handling DEFINES push,pop,init
stack:
  LET max = 1000;
  ROW max INT VAR stack;
  INT VAR stack pointer;
  PROC init stack:
    stack pointer := 0
  END PROC init stack;
  PROC push (INT CONST dazu wert):
    stack pointer INCR 1;
    IF stack pointer > max
      THEN errorstop ("stack overflow")
    ELSE stack [stack pointer] := dazu wert
    END IF
  END PROC push;

  PROC pop (INT VAR von wert):
    IF stack pointer = 0
      THEN errorstop ("stack empty")
    ELSE von wert := stack [stack pointer];
      stack pointer DECR 1
    END IF
  END PROC pop

END PACKET stack handling;
```

- <http://os.inf.tu-dresden.de/L3/usrman/node10.html>



- ▶ Parser generators belong to the tool set of a software engineer
- ▶ Parsers can parse
  - Texts (lines of rows)
  - CSV relations (lines of delimiter-separated tuples)
  - Pseudocode with island grammars
- ▶ The parser only parses the context-free structure of the programmes, document, or model;
- ▶ Syntax trees are built from a mapping of concrete to abstract syntax
- ▶ Context conditions, integrity and wellformedness constraints are delayed to the *static semantic analysis* on the syntax tree

# The End

- ▶ Why is a parser often delivering several results (parses)?
- ▶ How can you disambiguate ambiguous rules?
- ▶ Why is string parsing not the same as tree parsing? What is the difference of concrete and abstract syntax trees?
- ▶ Explain the difference of
  - A string grammar vs a tree grammar
  - A concrete syntax grammar (CS grammar) vs an abstract syntax grammar (AS grammar)