

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 21-1.2, 08.01.22

- 1) Parsing
- 2) Regular Tree Grammars
- 3) Tree Construction
- 4) Text Algebrae
- 5) Controlled Natural Languages (CNL)
- 6) Pseudocode and Markup Languages

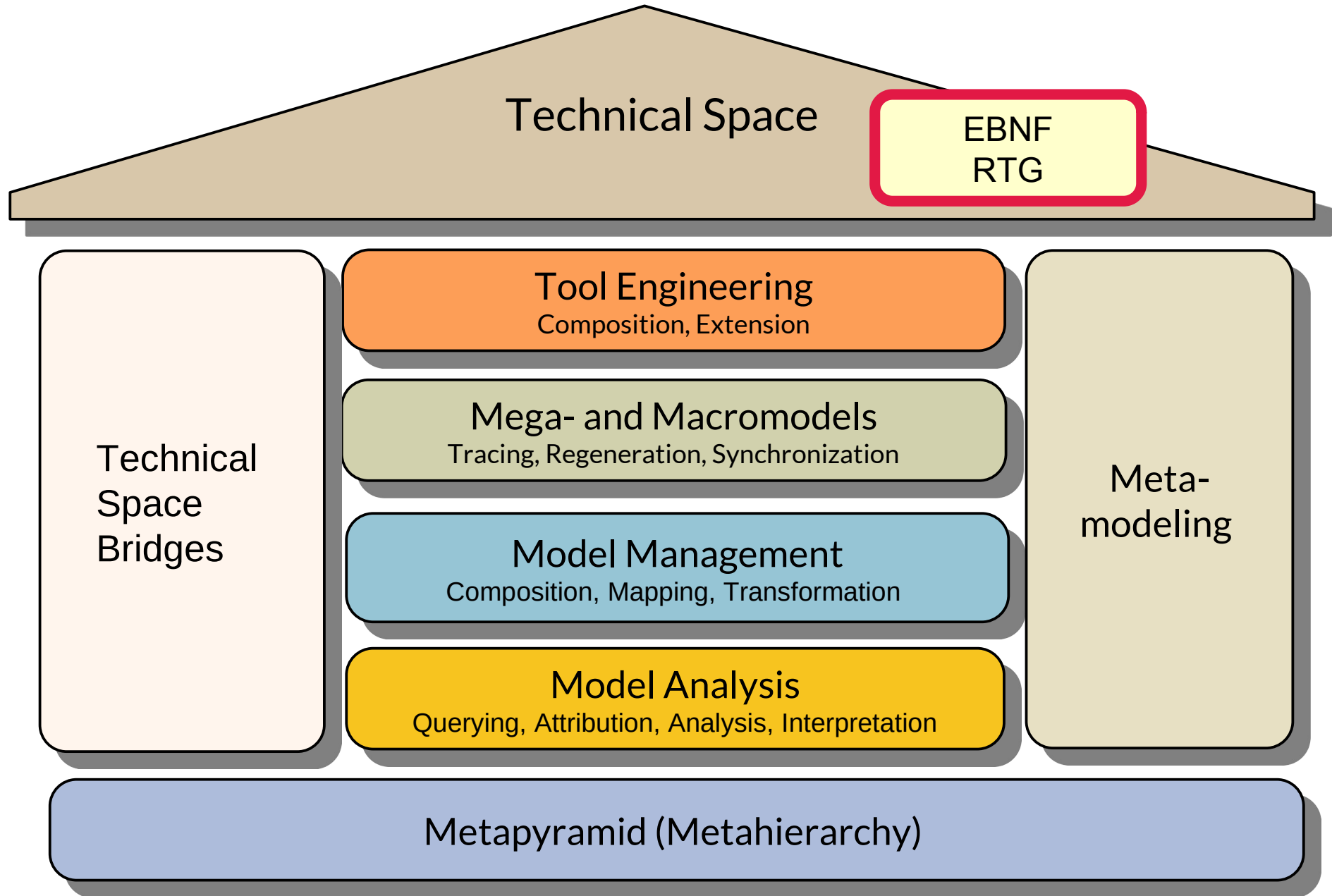
- ▶ Obligatory:
 - [http://www.antlr.org](http://wwwantlr.org)
- ▶ Optional:
 - Cocktail www.cocolab.de, die Compiler-Toolbox für die schnellsten Compiler der Welt (kommerziell, Demoversionen erhältlich)
 - TaTa Tree Grammars <http://tata.gforge.inria.fr/> and all the tree theory
 - Oana Andrei, Helene 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>

20.1. 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

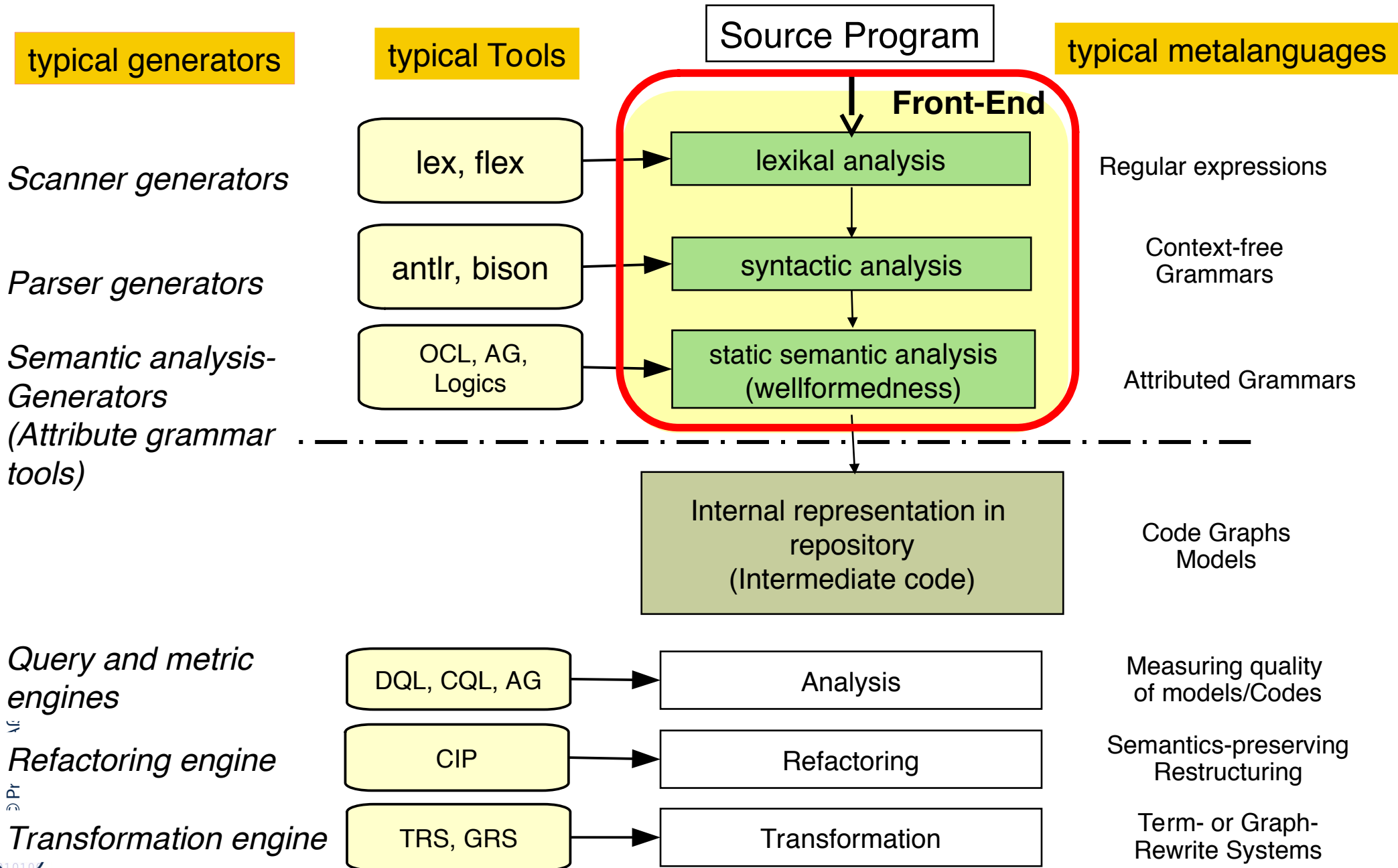
Q10: The House of a Technical Space



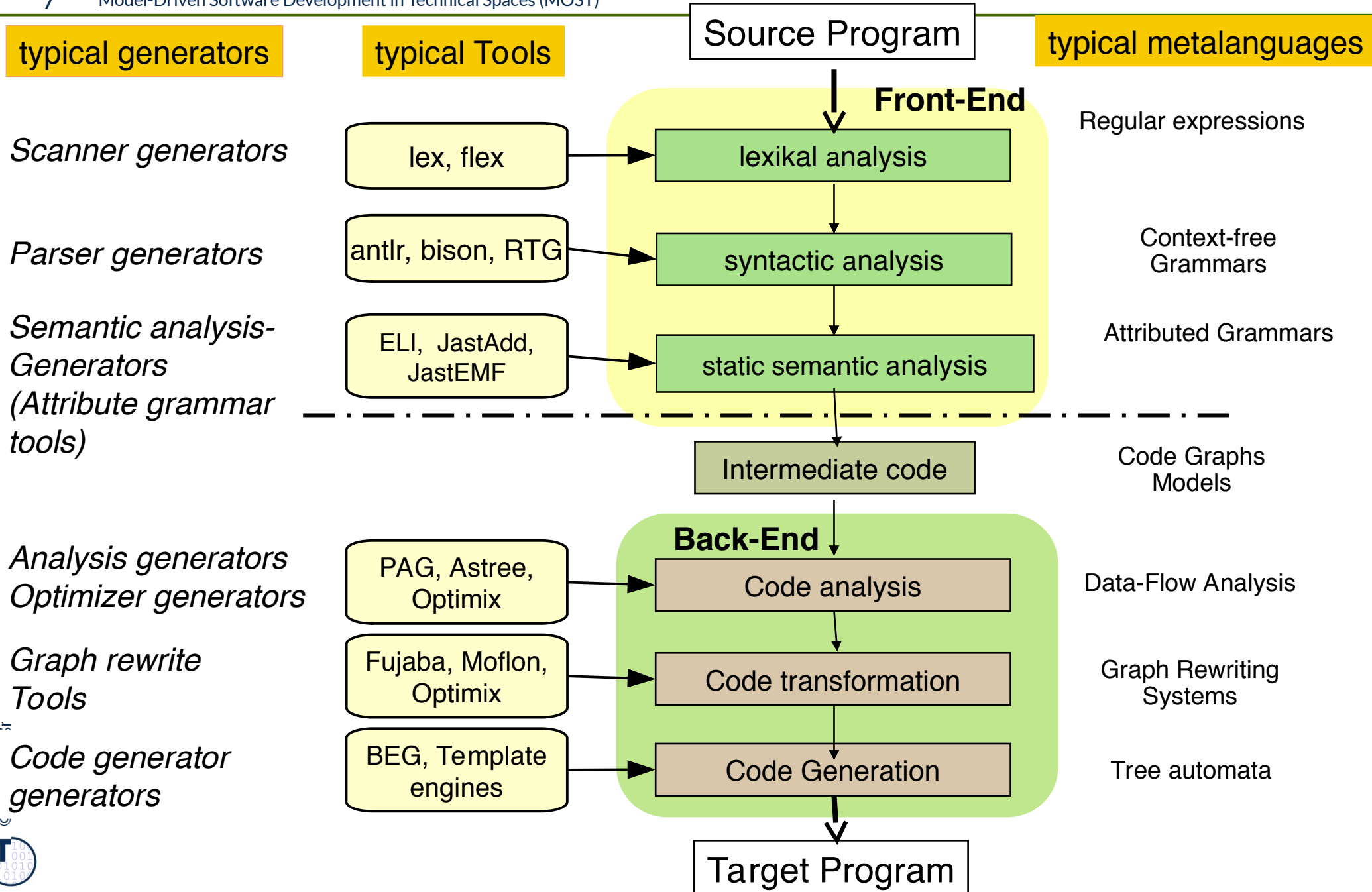
Q11: Overview of Technical Spaces in the Classical Metahierarchy

	Gramm arware (String s)	Text- ware	Table-ware		Treewar e (trees)	Link-Tree- ware		Graph ware/ Model ware			Role- Ware	CROM- Ware	Ontology -ware
	today				today								
	Strings	Text	Text- Table	Relationa l Algebra	NF2	XML	Link trees	MOF	Eclipse	CDI F	MetaEdit+	Context- role graphs	OWL-Ware
M 3	EBNF	EBNF		CWM (common warehouse model)	NF2- language	XSD	JastAd d, Silver	MOF	Ecore, EMOF	ERD	GOPPR	CROM	RDFS OWL
M 2	Gramma r of a language	Gramm ar with line delimit ers	csv- head er	Relationa l Schema	NF2- Schema	XML Schema , e.g. xhtml	Specific RAG	UML- CD, -SC, OCL	UML, many others	CDI F- lang uage s	UML, many others	CROM	HTML XML MOF UML DSL
M 1	String, Program	Text in lines	csv Table	Relation s	NF2-tree relation	XML- Docum ents	Link- Syntax- Trees	Classes, Progra ms	Classes, Program s	CDI F- Mod els	Classes, Programs	CROM models	Facts (T- Box)
M 0	Objects	Sequenc es of lines	Seque nces of rows	Sets of tuples	trees	dynami c semanti cs in browse r		Object nets	Hierarch ical graphs	Obje ct nets	Object nets	Context- Object-Role Nets	A-Box (RDF- Graphs)

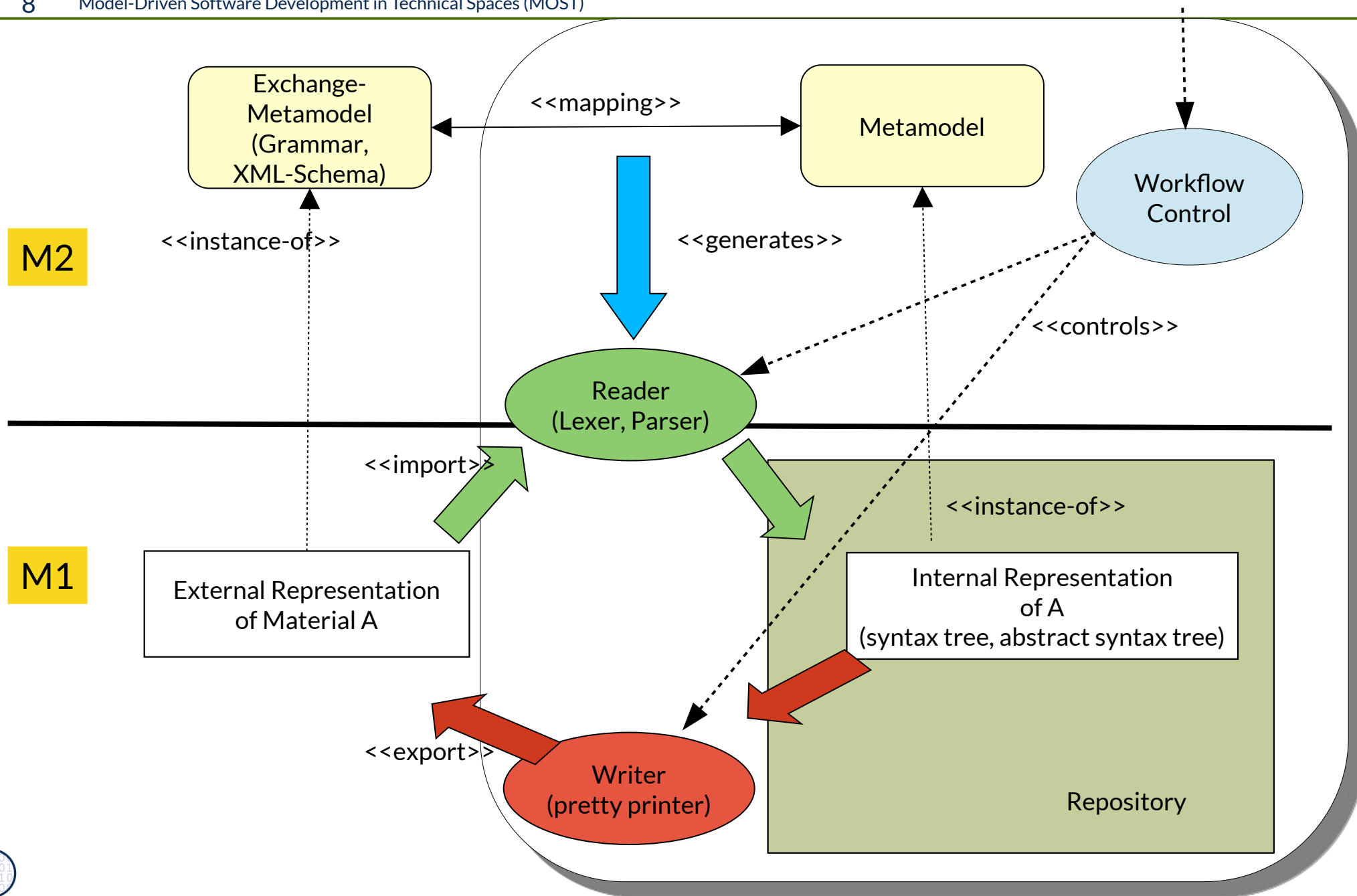
Q7: Phases of a Source Code Importers into a Repository and the Generating Tools



Q9: Phases of Compilers and Software Tools and Generators



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



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

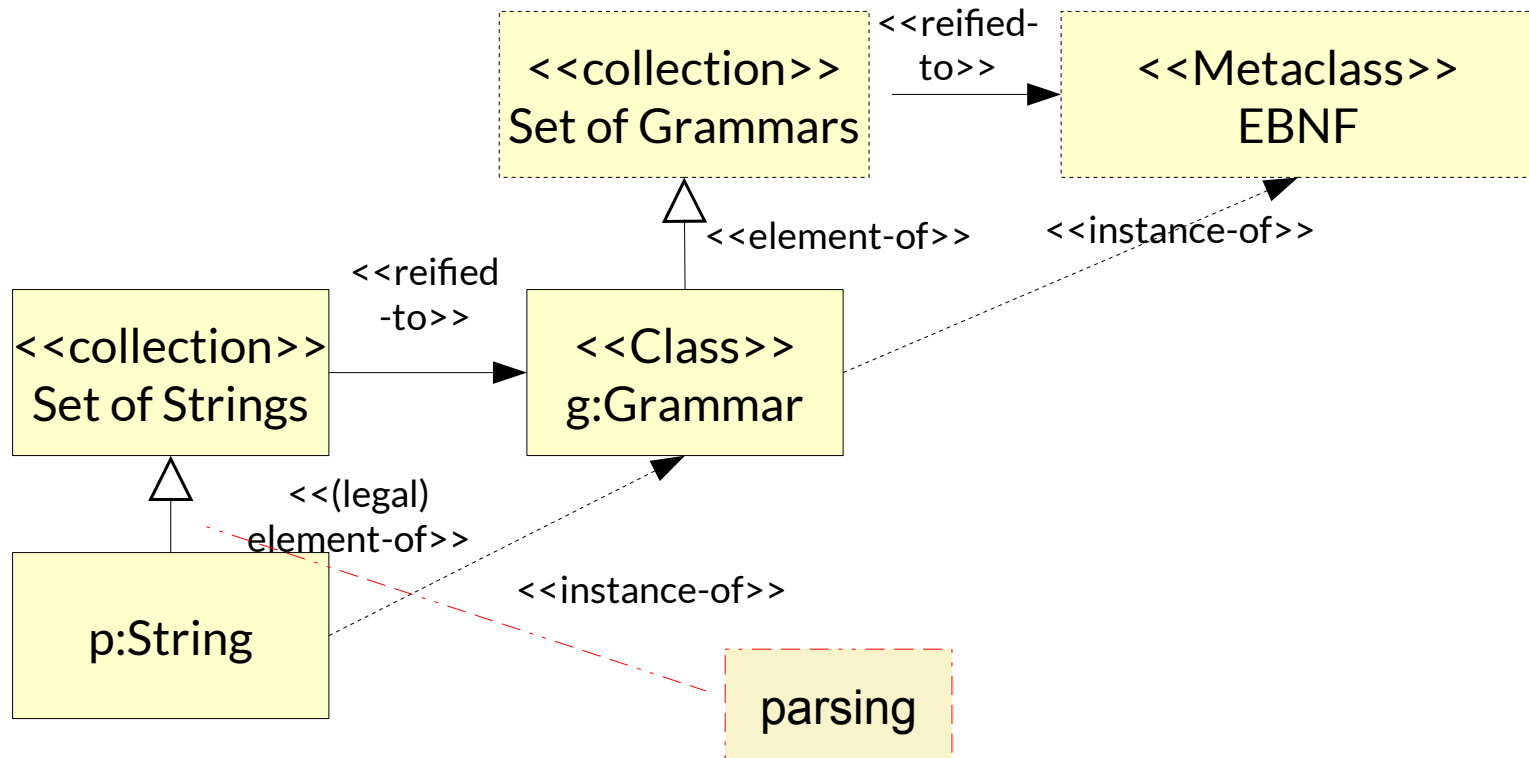
String/Text Parsing with Grammars

- ▶ 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)	$P = [P1 P2]$
$n \{ \dots \} m$	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 intermittend "	$C = D // \text{" "}$
* ... *	Comment	$X = B + C *text*$
$\langle a \rangle b$	Modifier (Kommentar)	$\langle old \rangle A \langle new \rangle A$
SYN	Synonym für Name	SecondName SYN SurName

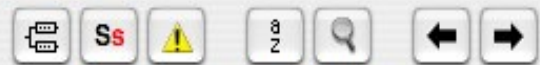
Example: Address Grammar

- ▶ “::=” 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

- ▶ Since the 90s, many parser generators have been built for C/C++
 - Cocktail's lalr, ell, lark 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



- 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
;

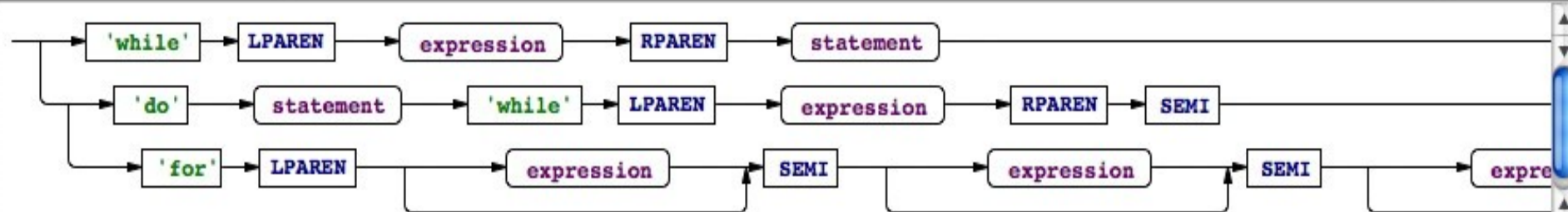
iteration_statement
: 'while' LPAREN expression RPAREN statement
| 'do' statement 'while' LPAREN expression RPAREN statement
| 'for' LPAREN expression SEMI expression SEMI expression
;

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

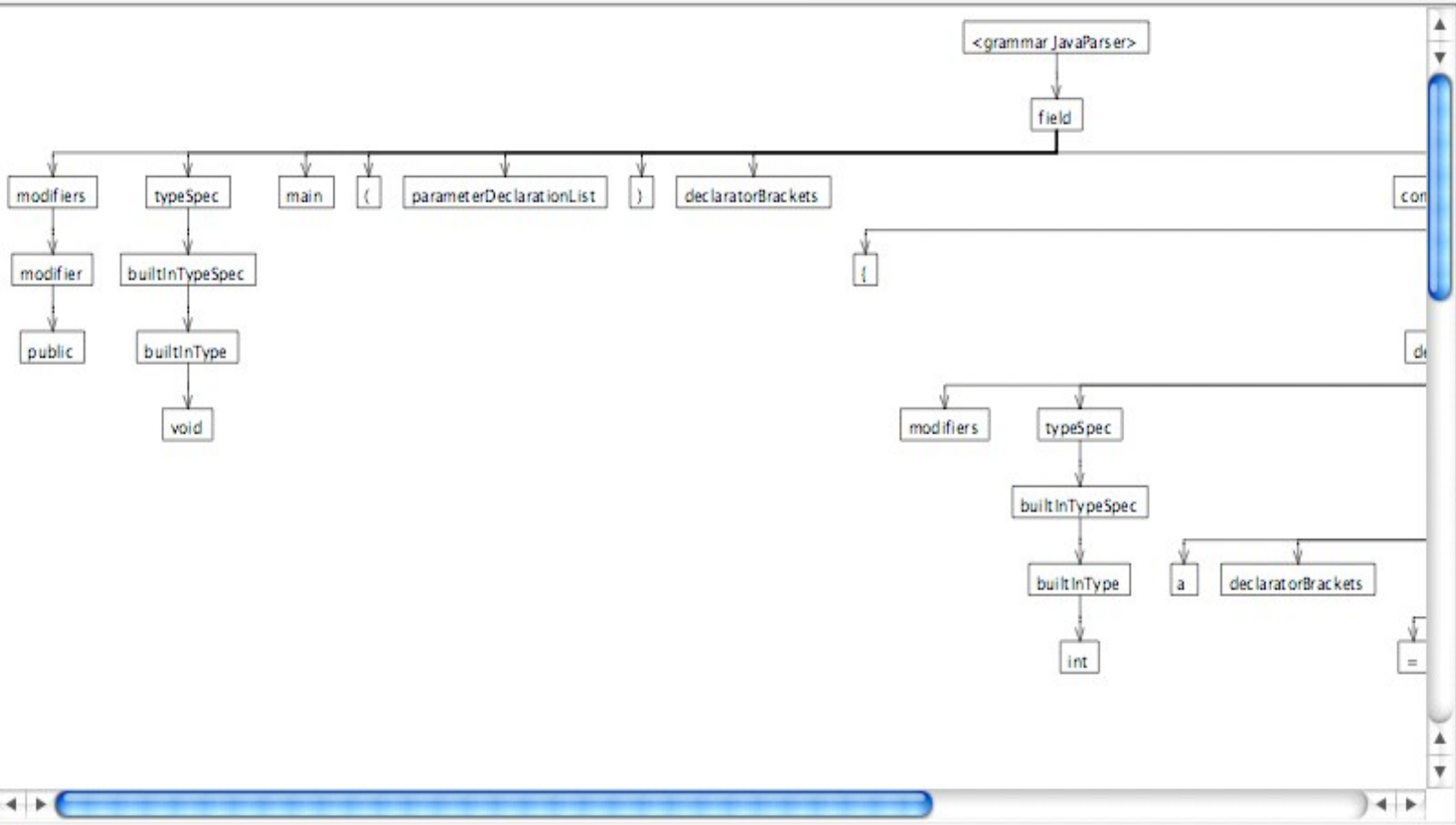


- handler
- expression
- expressionList
- assignmentExpression
- conditionalExpression

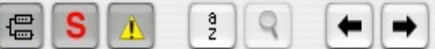
```
// the mother of all expressions  
expression  
: assignmentExpression  
;  
;
```

field

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



Syntax Diagram Interpreter Debugger Console



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

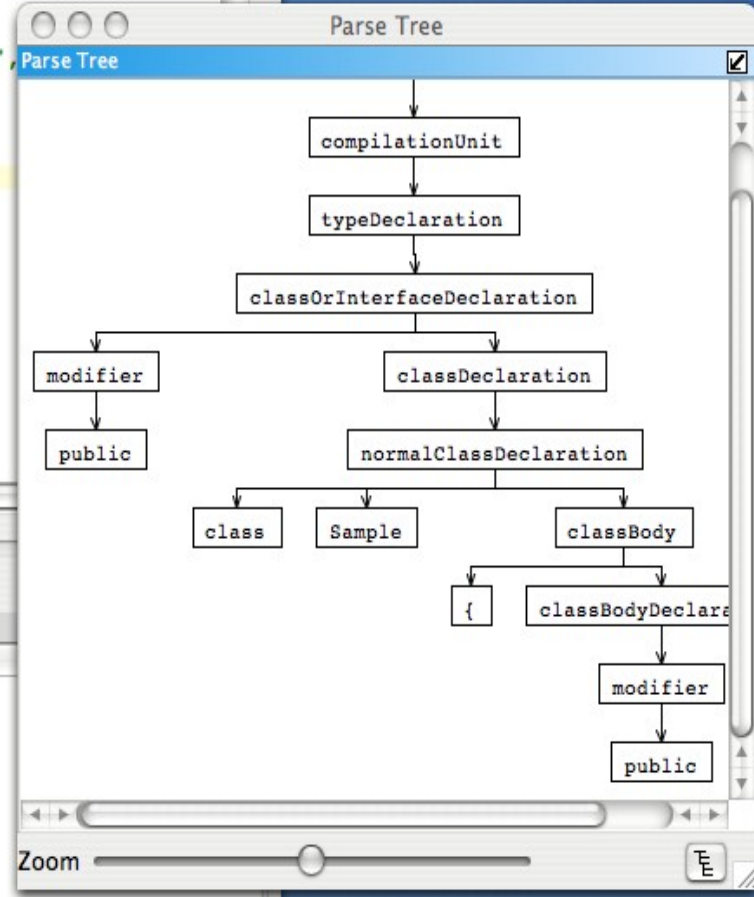
```
variableDeclaratorId
: Identifier ('[' ']')*
;

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");
    }
}
```

#	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

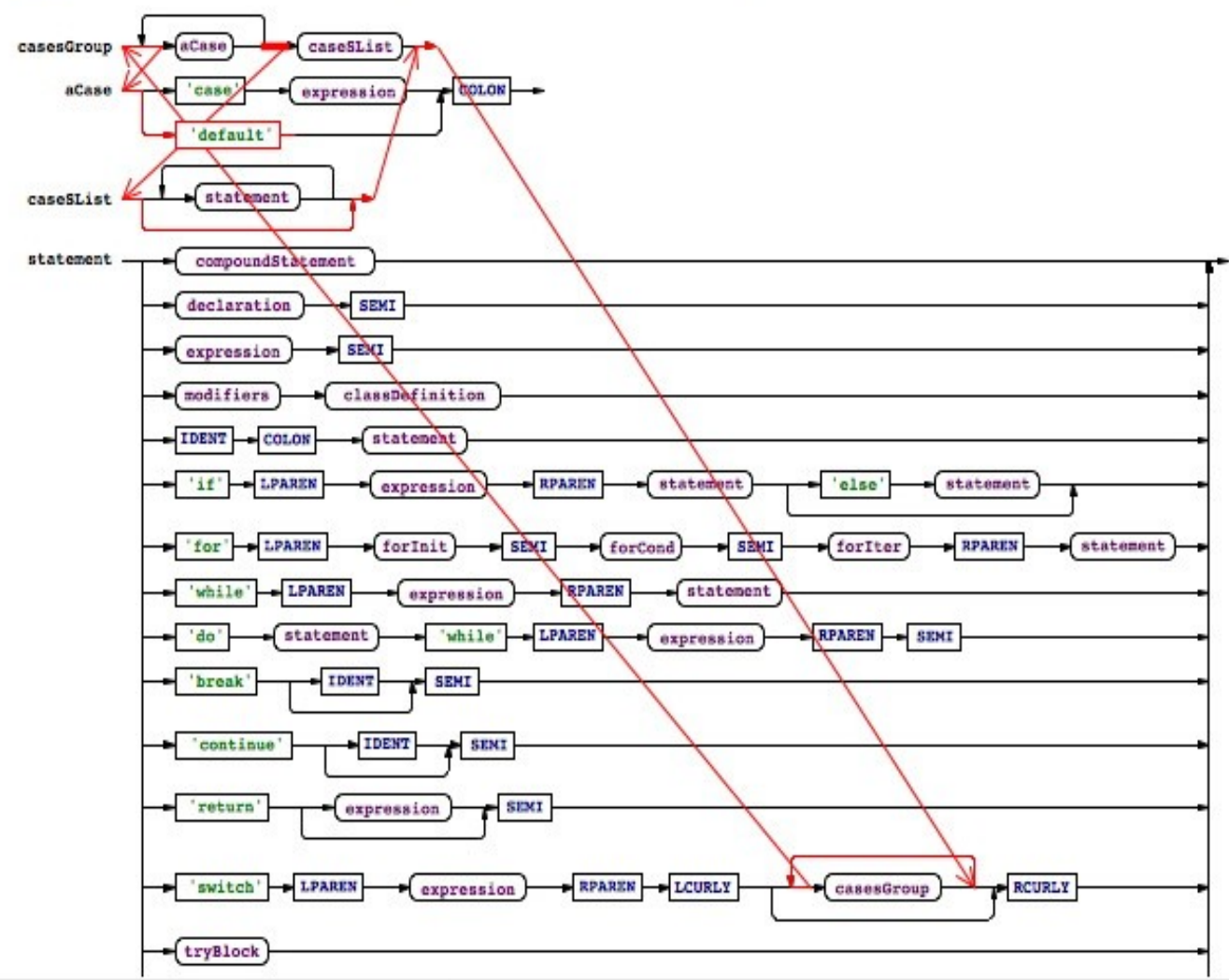




Zoom Show NFA

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

Alternatives: 1 2

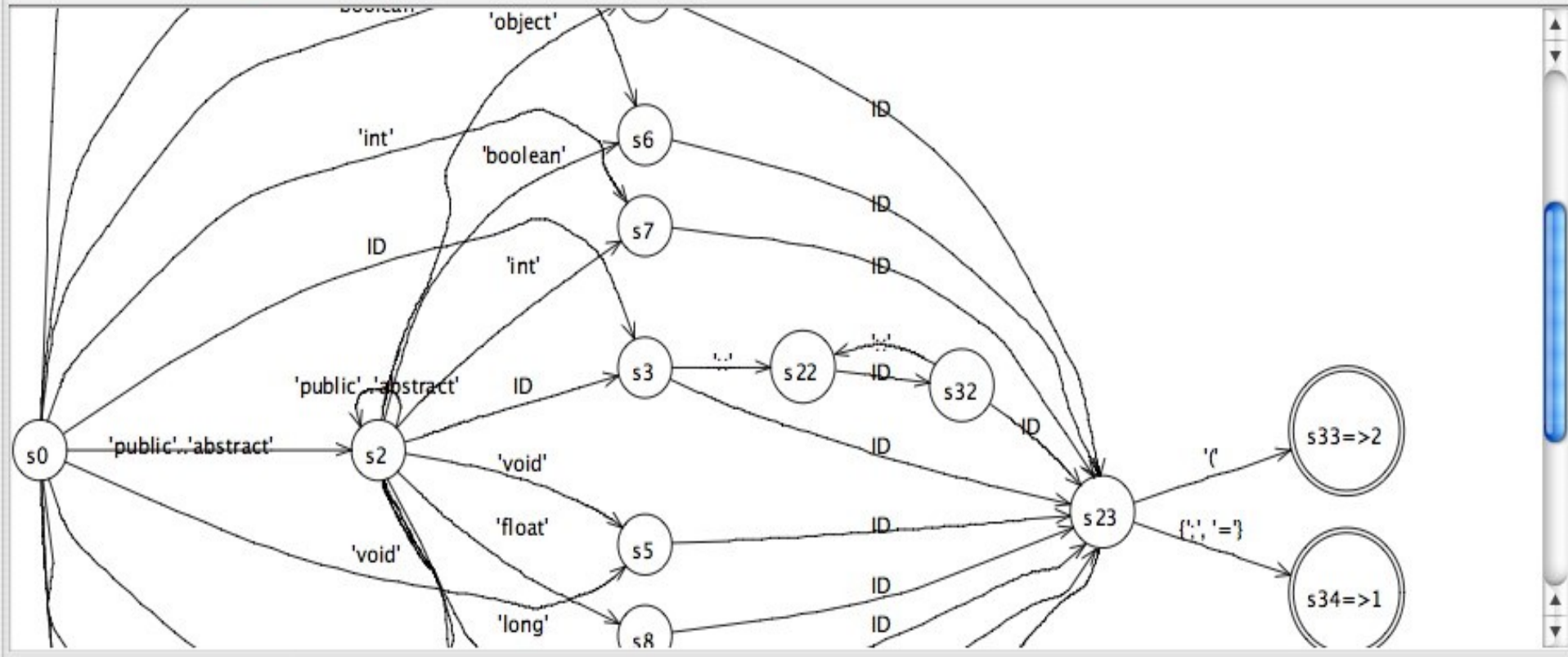
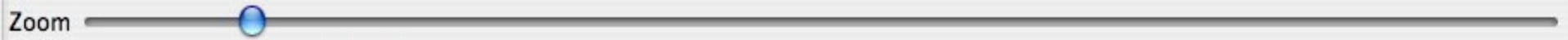


Syntax Diagram Interpreter Debugger Console



- 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
  }*
```



Syntax Diagram Interpreter Debugger Console **Decision 10 of "classDefinition"**

20.1.2 Program Interpretation While Parsing

An ANTLR Grammar for the Input Language of Pocket Calculator

- ▶ ANTLR knows several *actions* (*snippets*, *fragments*) in a rule
 - *semantic actions*: modifying attributes of nodes or temporary values
 - *emissions*: emitting tokens of an output alphabet, e.g., for code generation
 - *motions*: emitting an action on a shared environment (“motion grammars”)
 - e.g., for controlling a robot
 - Motion grammars usually influence parsing also!

Actions Calculating Attributes

- ▶ 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 *=
$value;})*
  ;
atom returns [int value]
:  INT {$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);
        // Link lexer tokens with parser parser
        ExprParser parser = new ExprParser(tokens);
        // run the parser parser
        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

```

graph TD
    Root["<grammar Expr>"] --> prog
    prog --> stat
    stat --> expr
    expr --> multExpr1[multExpr]
    expr --> plus["+"]
    expr --> multExpr2[multExpr]
    multExpr1 --> atom1[atom]
    atom1 --> 2
    multExpr2 --> atom2[atom]
    multExpr2 --> star["*"]
    multExpr2 --> atom3[atom]
    atom2 --> 3
    atom3 --> 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';
WS : ' ' | '\t' | '\f' | '\r';

```

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

```

Zoom

Stack

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

Input
Output
Parse Tree
AST
Stack
Events

Syntax Diagram
Interpreter
Debugger
Console

9 rules
35:13
Writable

/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;}
}*

```

Break on: All Location Consume LT Exception

Output

14

Parse Tree

```

graph TD
    root --> prog
    prog --> stat
    stat --> expr
    stat --> empty1[ ]
    expr --> multExpr1[multExpr]
    expr --> plus1[+]
    expr --> multExpr2[multExpr]
    multExpr1 --> atom1[atom]
    atom1 --> 2[2]
    multExpr2 --> atom2[atom]
    atom2 --> 3[3]
    multExpr2 --> plus2[+]
    plus2 --> atom3[atom]
    atom3 --> 4[4]

```

Zoom

Stack

#	Rule

Input
Output
Parse Tree
AST
Stack
Events

Syntax Diagram
Interpreter
Debugger
Console

9 rules
15:15
Writable

Applications of String Grammars

Applications can be 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 (RTG)

- 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 **explicitly**, 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
```

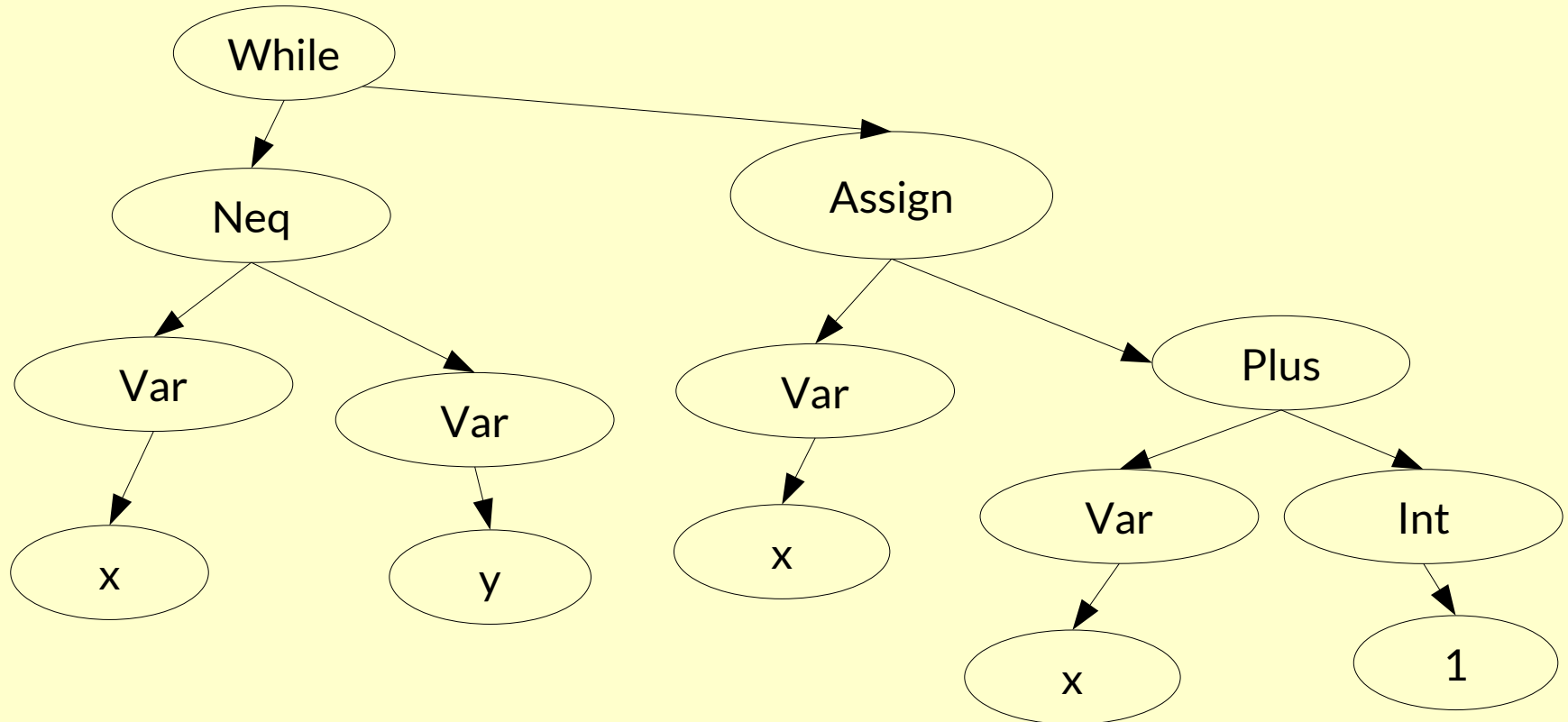
```
start Program
```

```
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>
Id         -> <string>
```

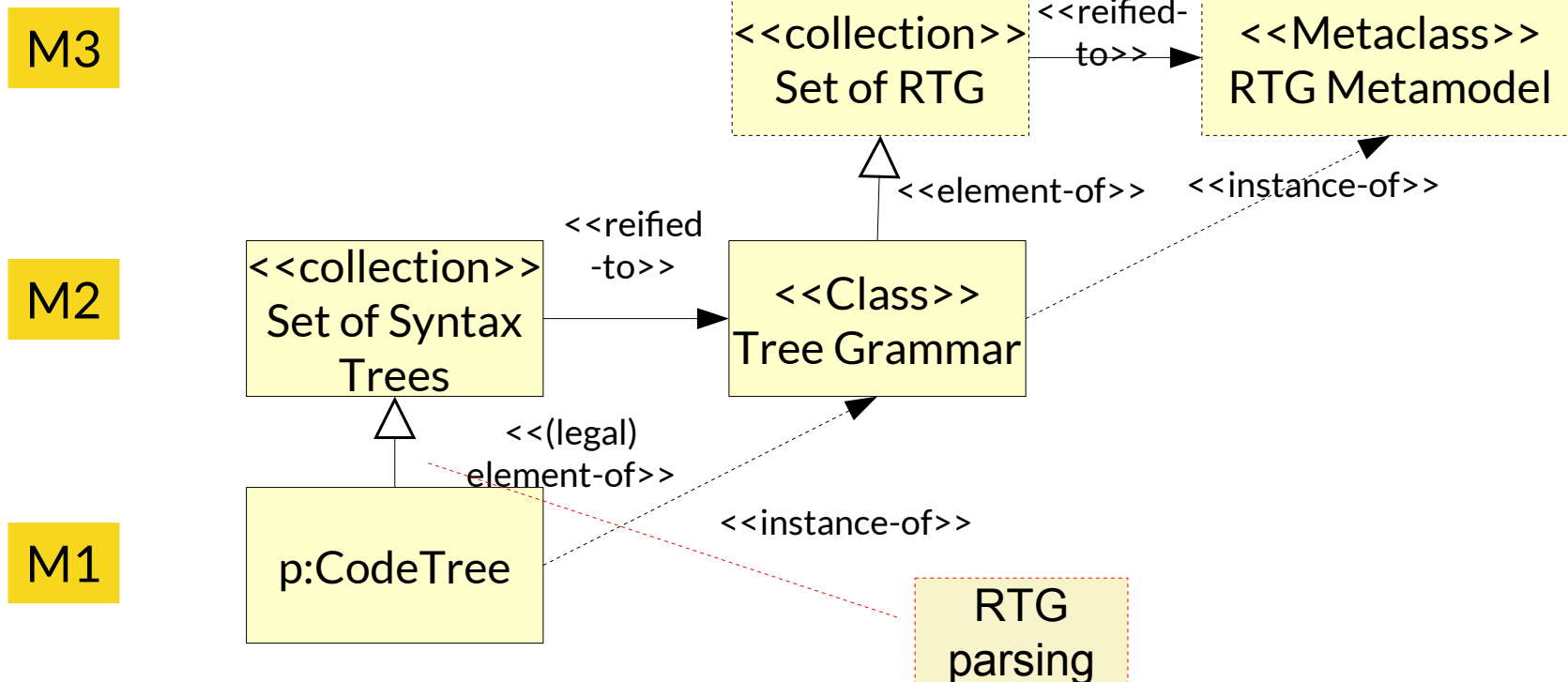
Correct Instance?

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



Tree Parsing with RTG

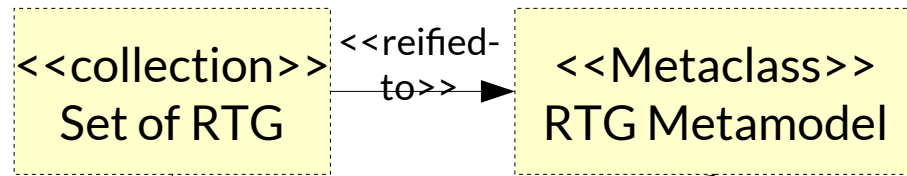
- ▶ An RTG can be used to generate a **tree parser** that tests the legality of a code tree with a tree grammar



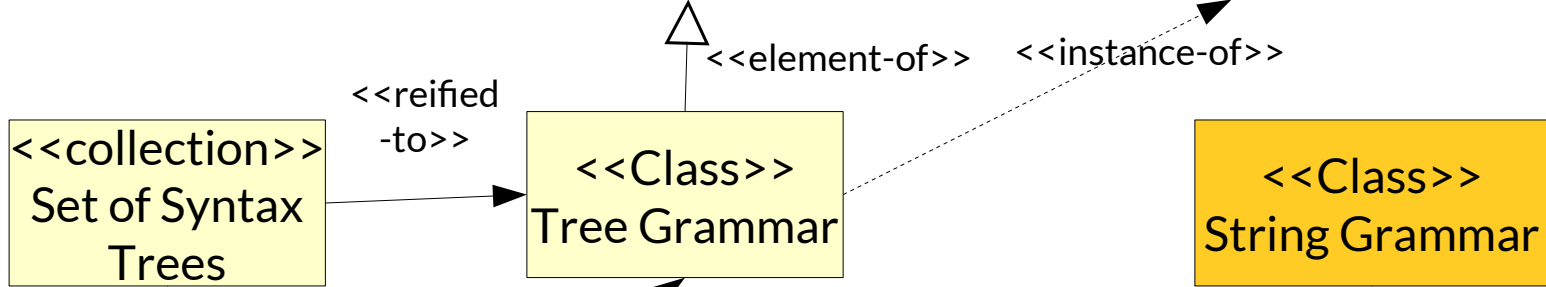
Tree Pretty-Printing with RTG

- ▶ 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

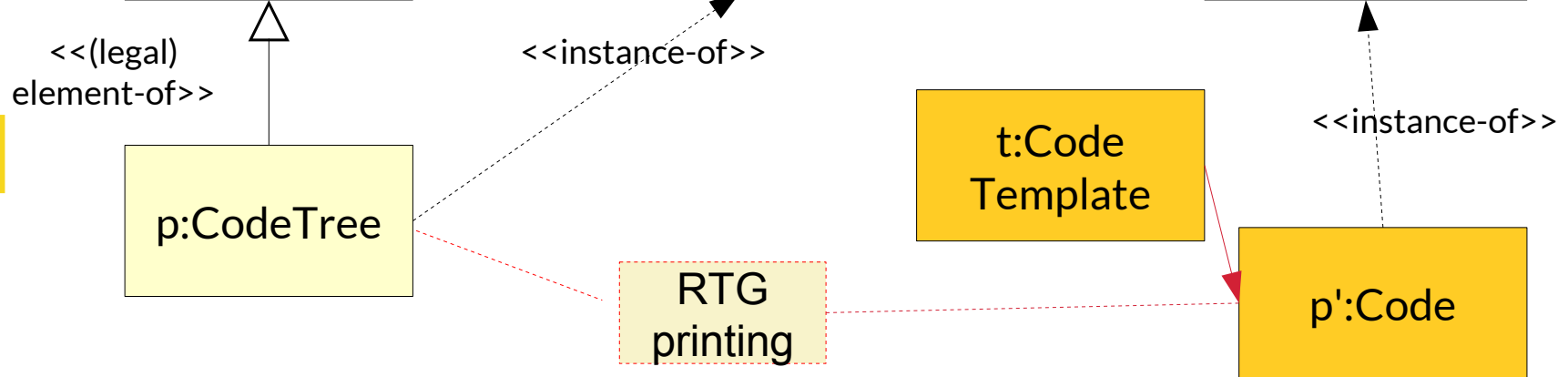
M3



M2



M1

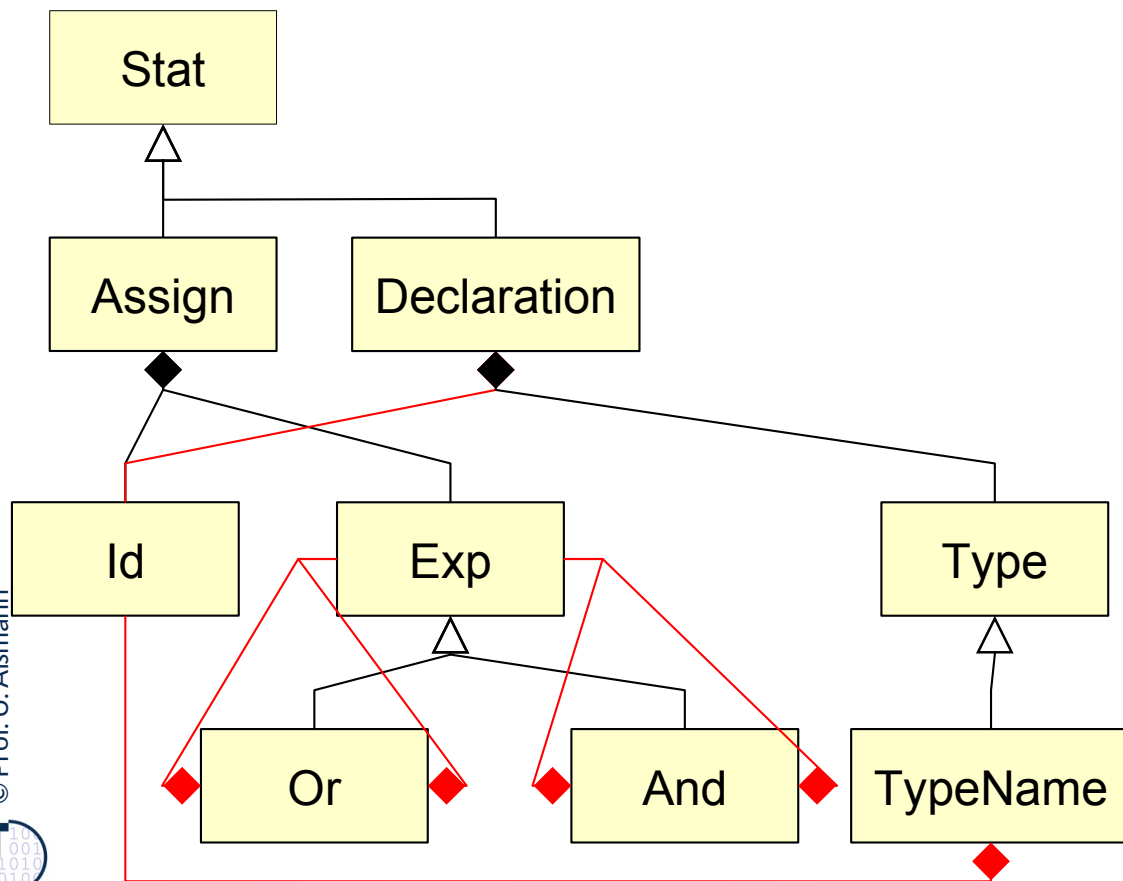


20.2.2. Metamodels and Tree Grammars

- Full parser also build syntax trees – Design Pattern Builder

Grammars and Metamodels are Very Similar

- ▶ The nonterminals of a grammar form the *name space* of the grammar
- ▶ Definition of a nonterminal (on left side)
- ▶ Use of a nonterminal (on right side) induces relation to definition
- ▶ Metamodels are graphs (including the red edges); grammars are *link trees* (red edges are overlaid)



Stat	-> Assign(Id,Exp)
Stat	-> Declaration(Id,Type)
Type	-> TypeName(Id)
Exp	-> Or(Exp,Exp) And(Exp,Exp)
Id	-> String

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 (concrete syntax, CS)
 - **However, the syntax tree must be built as “action” while parsing**
- ▶ After parsing, the parser creates an (**abstract**) *syntax tree (AST)*, i.e., builds up a tree with regard to a *regular tree grammar of the abstract syntax (AS)*
 - Recognized nonterminals have to be mapped
 - Tokens, keywords, comments, layouts have to be omitted
 - **Tree building:** Treenodes have to be allocated and composed
- ▶ This **CS-AS mapping (from concrete to abstract syntax)** is created by hand in *side actions* of the parser
 - as action snippet in the rules
- ▶ 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)

- ▶ CS-AS mapping works via side actions of the grammar rules (“transducer”)
- ▶ 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 abstracte 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

for composition of texts and trees

Composition with Composition Systems



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`

Excursion: CSV and Digital Preservation

- ▶ <https://digital-preservation.github.io/>
- ▶ Digital preservation “Bewahrung des digitalen Kulturerbes”
- ▶ csv schema language
<https://digital-preservation.github.io/csv-schema/csv-schema-1.1.html>
- ▶ <https://jsonformatter.org/xml-parser> online xml parser

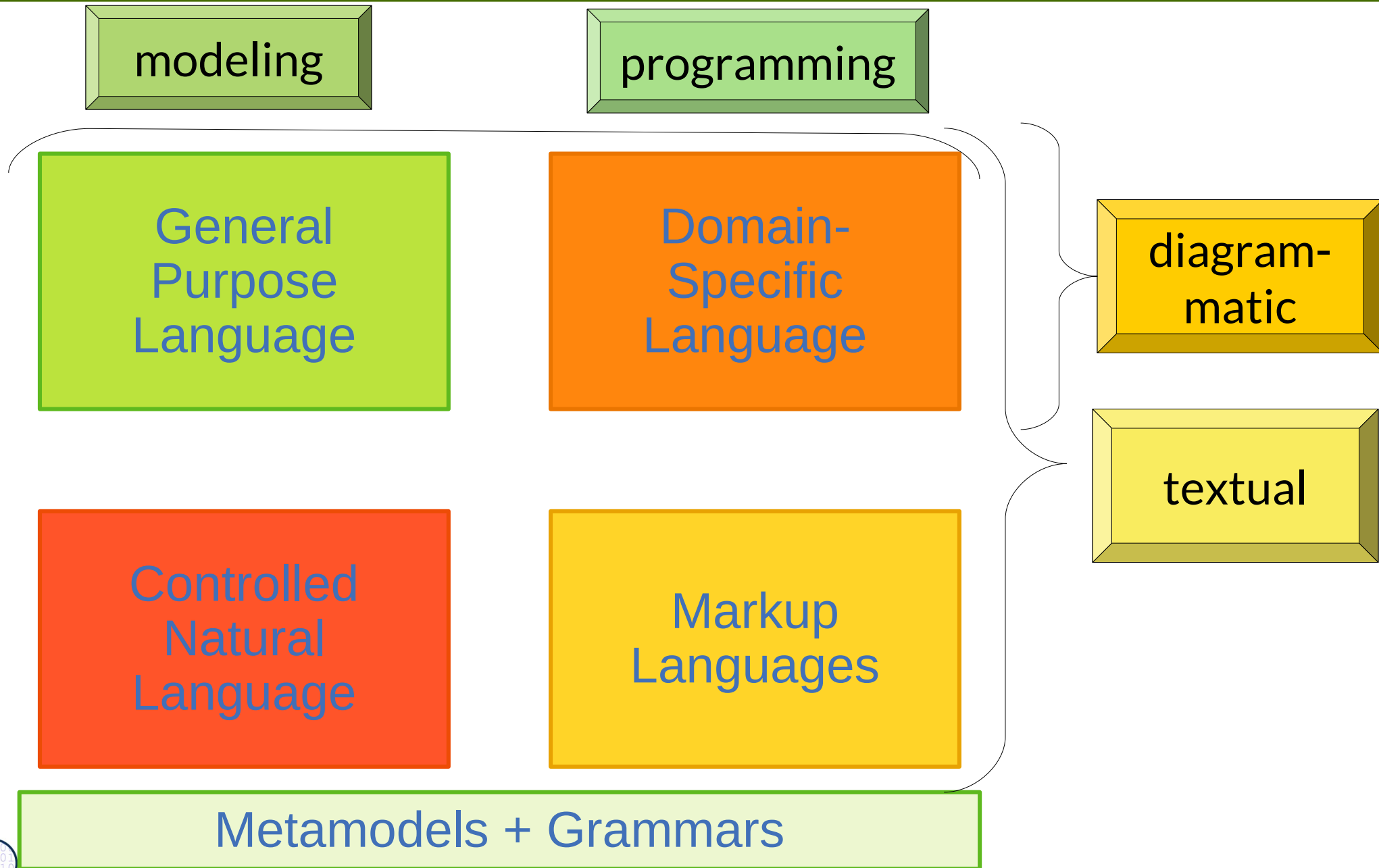
Tree Algebra

- ▶ 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 Controlled Natural Languages (CNL)

- Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguistics*, 40(1):121--170, 2014.
- Anne Cregan, Rolf Schwitter, and Thomas Meyer. Sydney owl syntax - towards a controlled natural language syntax for owl 1.1. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*, volume 258. CEUR-WS, 2007.

Q16: Languages in Software Factories

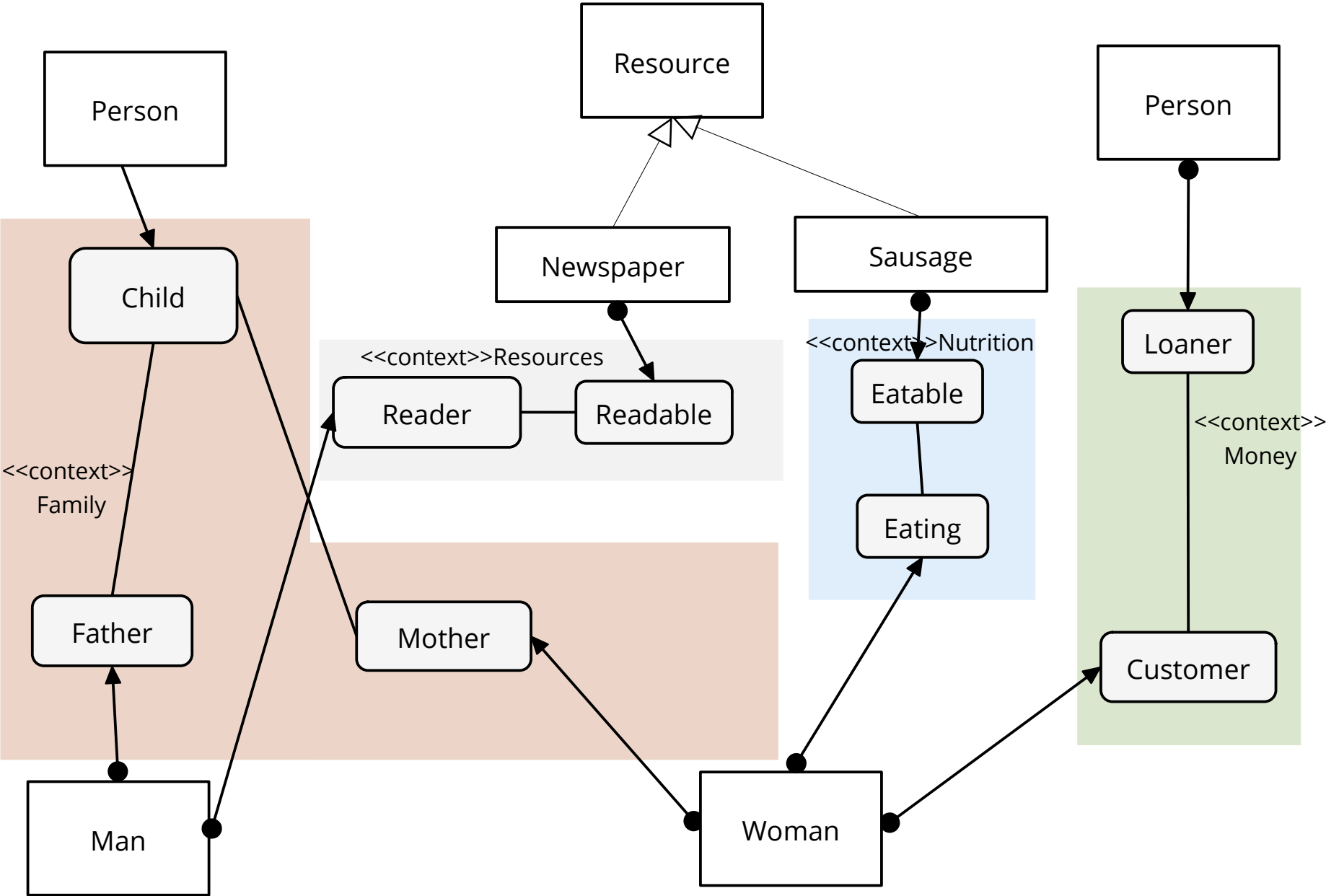


Controlled Natural Languages (CNL)

- ▶ Def.: A **controlled natural language** is a restricted natural language that can be described by a formal grammar, either a
 - context-free grammar (e.g., specified by ANTLR)
 - regular tree grammar (e.g., specified by Spoofax)
 - context-sensitive grammar or attributed grammar (e.g., specified by JastAdd, see later).

- ▶ Advantages:
 - Simple, comprehensive syntax
 - In contrast to a natural language, a CNL can be processed easily by tools, because syntax trees for naturally looking texts can be built

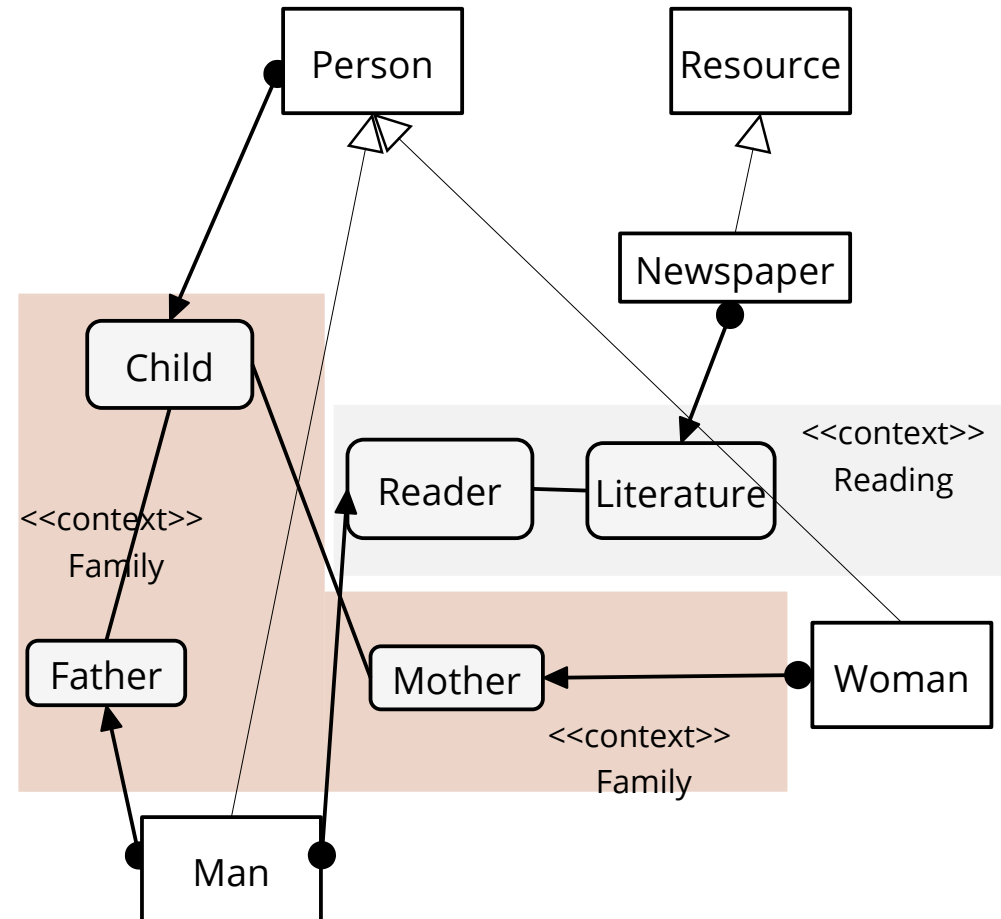
Families and Banks in Natural and Role Types (CROM)



Families and Banks in Natural and Role Types, Specified in Sydney OWL Syntax

- ▶ Models can be read or specified in controlled natural language
- ▶ [Cregan] introduces a CNL for OWL ontologies, also useful for CROM
- ▶ Example: Families in CROM:

Every Man is a Person.
Every Woman is a Person.
Every Person may play a Child (in a Family).
Every man:Man may play a Father (in a Family).
Every Child is related to a Mother in a Family
Every Child is related to a Father in a Family.
A man:Main may play a Reader (in Reading).
Every Reader is related to a Literature in Reading.
Every Newspaper is a Resource.
Every Newspaper may play a Literature (in Reading).



PENS Classification

- ▶ [Kuhn] Languages can be classified with regard to 4 criteria and school grades.
- ▶ E.g., English is $S^2E^5N^5S^2$.

	Precision	Expressiveness	Naturalness	Simpliicty
English	1	5	5	1
Controlled English	4	3	5	4
Domain-specific language	4	3	5	4
UML Class diagrams	3	3	4	4

20.6 Markup Languages and Pseudocode

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

Languages used for documentation

- ▶ **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)

Examples for Pseudocode

- ▶ 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 ungueltig
    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
```


LATEX, XML and Pseudocode

- ▶ 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}
```

Support for Pseudocode

▶ 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>

Summary

- ▶ 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)

20.A.1 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)**

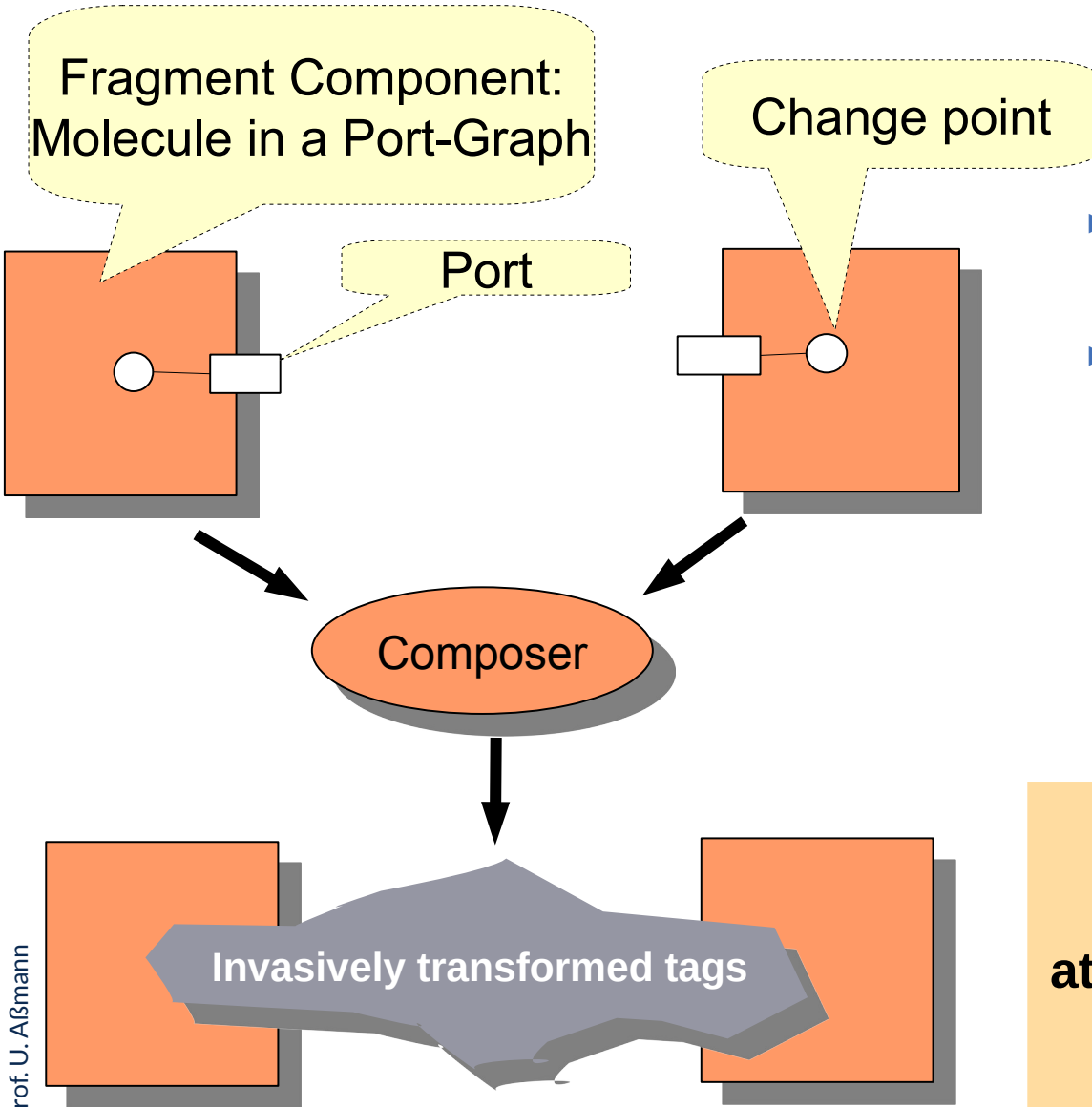
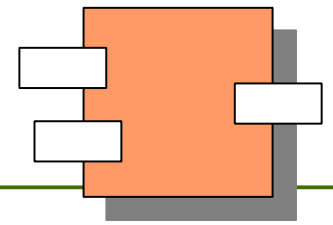
Composition Technique:

Hook Transformation

Composition Language:

Standard Languages

Invasive Composition as Hook Transformations



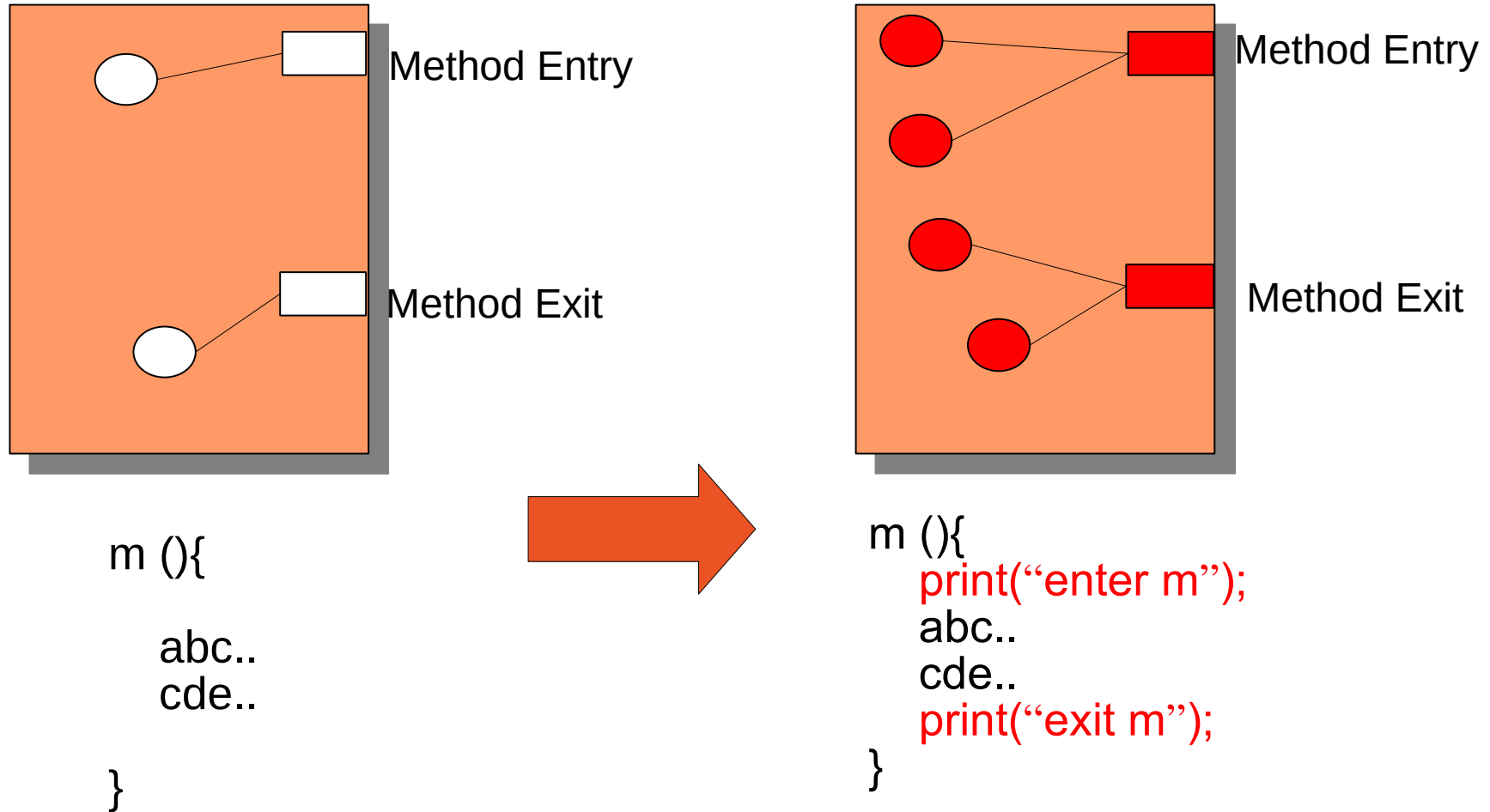
- ▶ A **port graph** is a graph in which each node (**molecule**) has a set of *ports*
- ▶ A **fragment component** is a molecule with ports (slots, hooks, query points) related to change points

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

Binding Implicit Hooks with Fragments

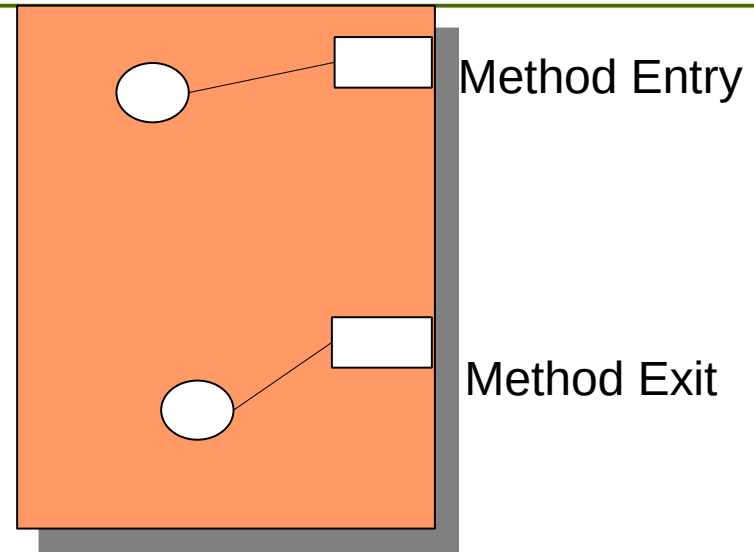
65

Model-Driven Software Development in Technical Spaces (MOST)

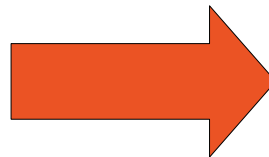


```
component.findHook(„MethodEntry“).extend("print(\nenter m\n");");  
component.findHook(„MethodExit“).extend("print(\nexit m\n");");
```


Partial Parsing of Fragment Components



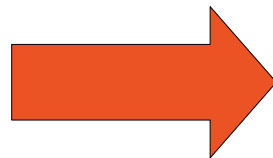
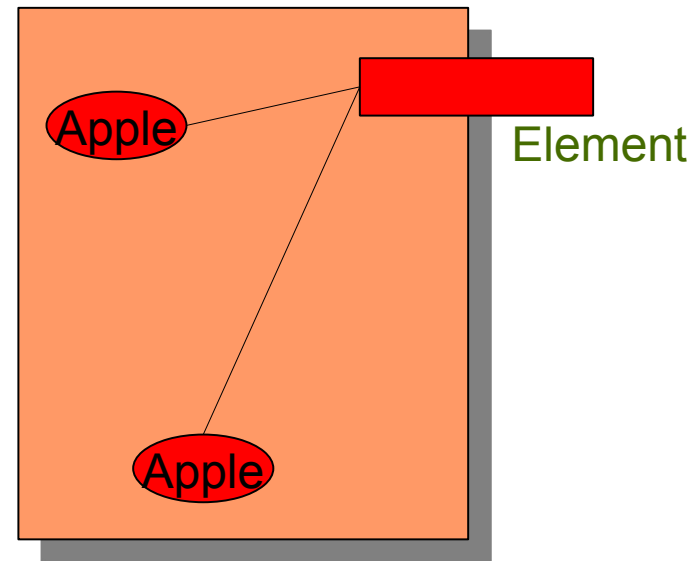
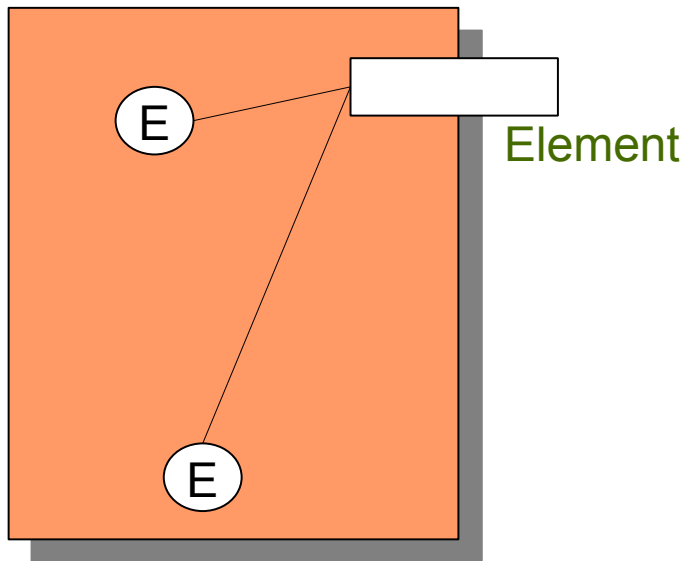
```
m (){\n  abc..\n  cde..\n}
```



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

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

Binding Declared Hooks with Fragments



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

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

```
...
```

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

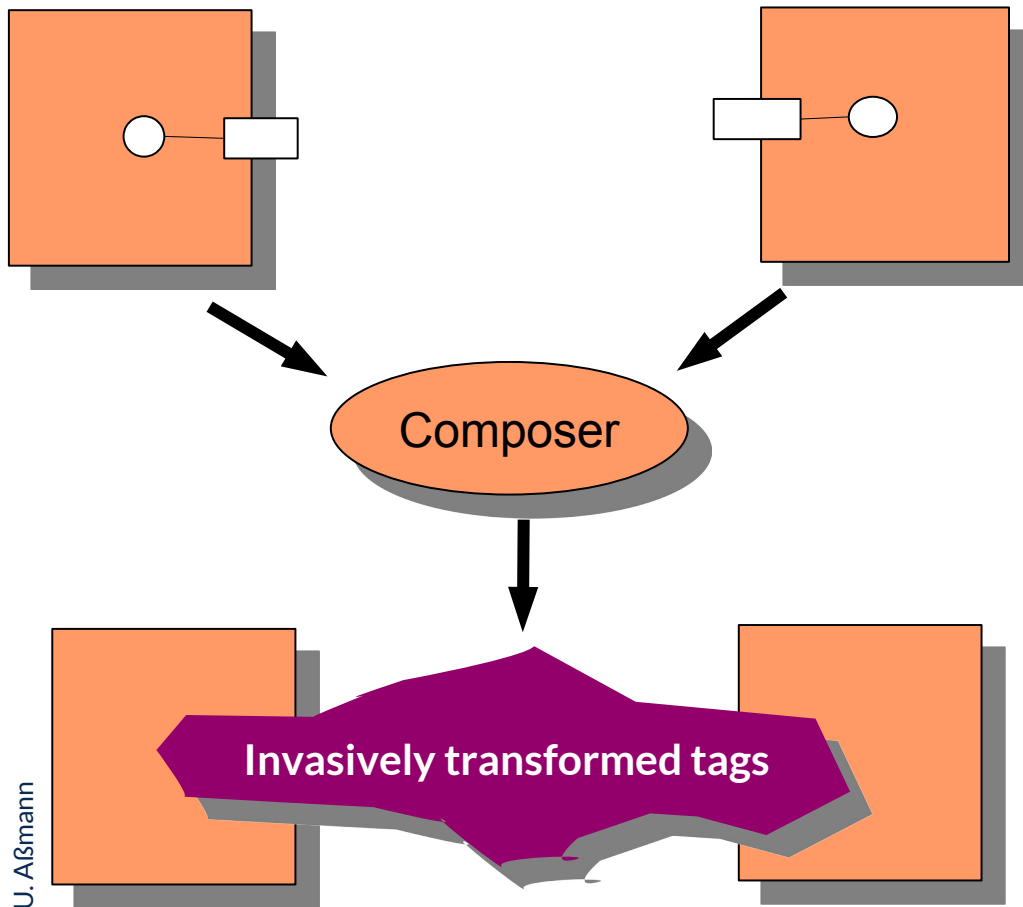
```
....  
list.add(new Apple());
```

```
...
```

```
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



Operators in a Port-Graph Algebra

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



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 21-1.2, 08.01.22

- 1) Parsing
- 2) Regular Tree Grammars
- 3) Tree Construction
- 4) Text Algebrae
- 5) Controlled Natural Languages (CNL)
- 6) Pseudocode and Markup Languages

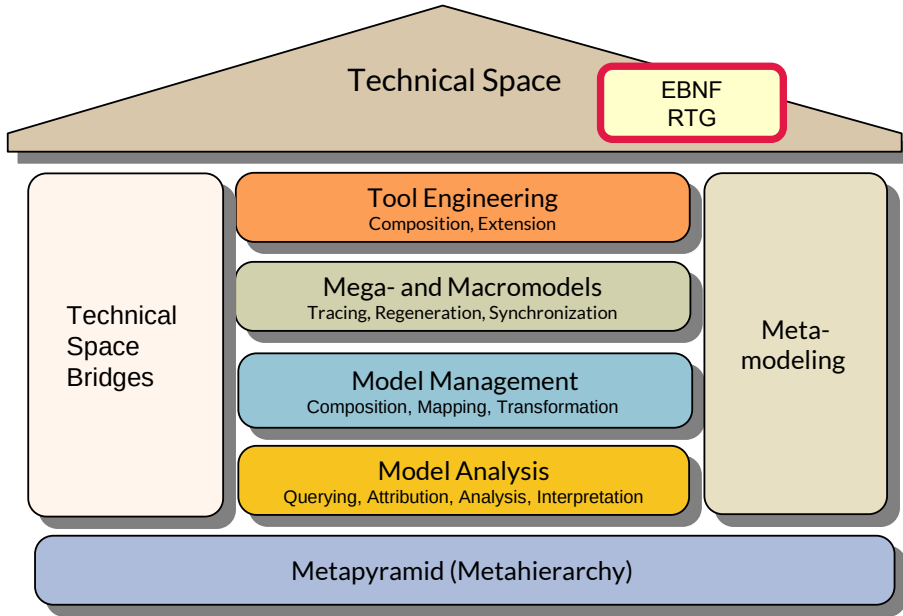
- ▶ Obligatory:
 - <http://www.antlr.org>
- ▶ Optional:
 - Cocktail www.cocolab.de, die Compiler-Toolbox für die schnellsten Compiler der Welt (kommerziell, Demoverionen erhältlich)
 - TaTa Tree Grammars <http://tata.gforge.inria.fr/> and all the tree theory
 - Oana Andrei, Helene Kirchner. A Port Graph 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>



20.1. 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

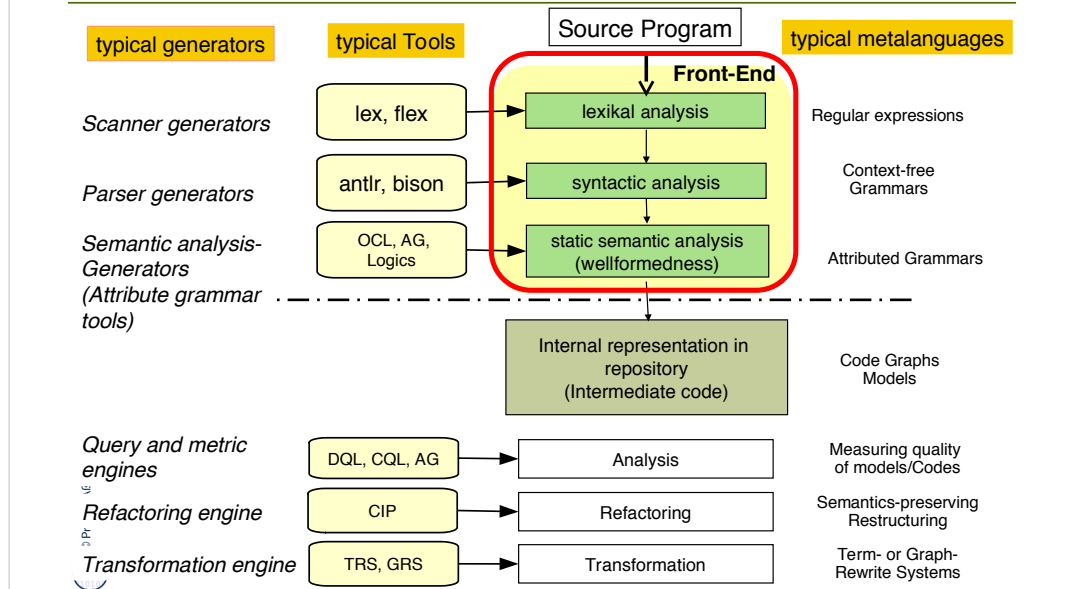


Q11: Overview of Technical Spaces in the Classical Metahierarchy

	Gramm arware (String s)	Text- ware	Table-ware		Treewar e (trees)	Link-Tree- ware		Graph ware/ Model ware			Role- Ware	CROM- Ware	Ontology -ware
		today			today								
	Strings	Text	Text- Table	Relationa l Algebra	NF2	XML	Link trees	MOF	Eclipse	CDI F	MetaEdit+	Context- role graphs	OWL-Ware
M 3	EBNF	EBNF		CWM (common warehou se model)	NF2- language	XSD	JastAd d, Silver	MOF	Ecore, EMOF	ERD	GOPPR	CROM	RDFS OWL
M 2	Gramma r of a language	Gramm ar with line delimit ers	csv- head er	Relation al Schema	NF2- Schema	XML Schema , e.g. xhtml	Specific RAG	UML- CD, -SC, OCL	UML, many others	CDI F- lang uage s	UML, many others	CROM	HTML XML MOF UML DSL
M 1	String, Program	Text in lines	csv Table	Relation s	NF2-tree relation	XML- Docum ents	Link- Syntax- Trees	Classes, Progra ms	Classes, Program s	CDI F- Mod els	Classes, Programs	CROM models	Facts (T- Box)
M 0	Objects	Sequenc es of lines	Sequenc es of rows	Sets of tuples	trees	dynam ic semanti cs in brow ser		Object nets	Hierarch ical graphs	Obj ect nets	Object nets	Context- Object-Role Nets	A-Box (RDF- Graphs)

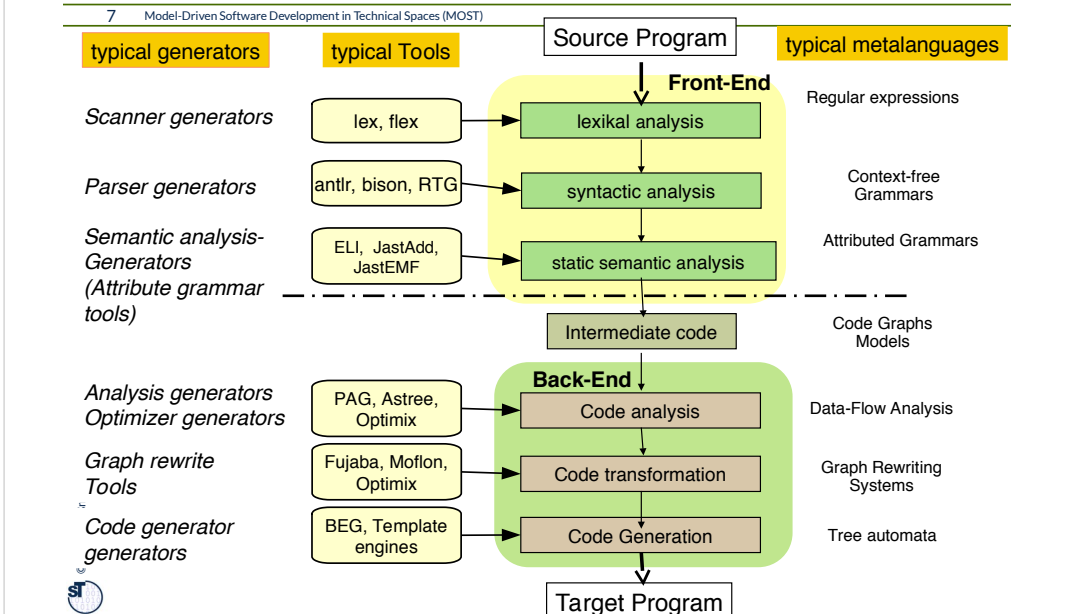
Q7: Phases of a Source Code Importers into a Repository and the Generating Tools

6 Model-Driven Software Development in Technical Spaces (MOST)



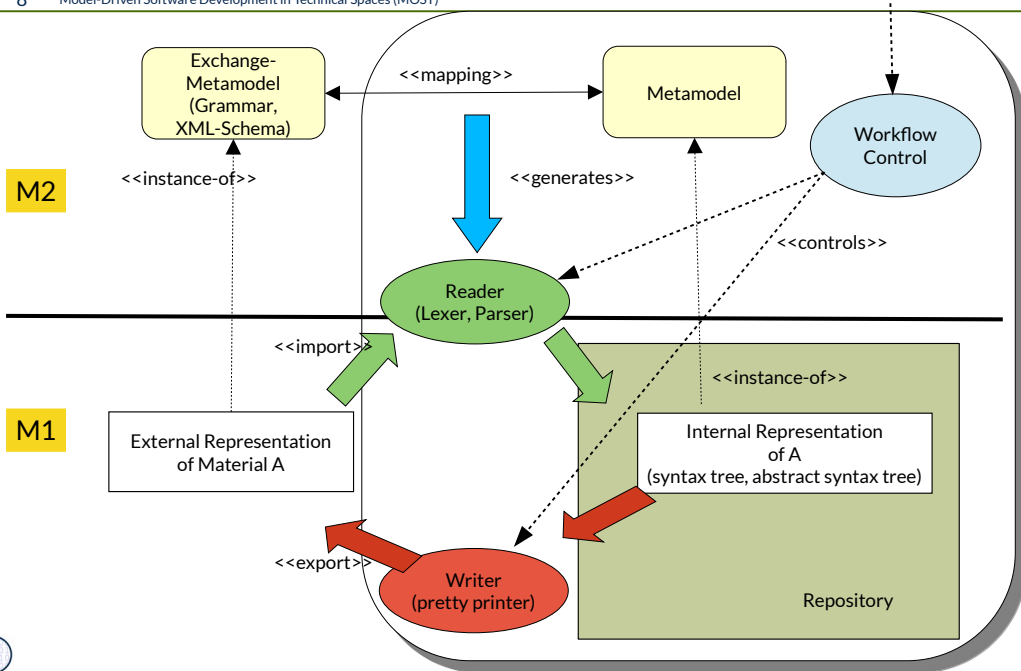
Beispiel LLVM bauen

Q9: Phases of Compilers and Software Tools and Generators



from Introductory Chapter “Architecture”

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

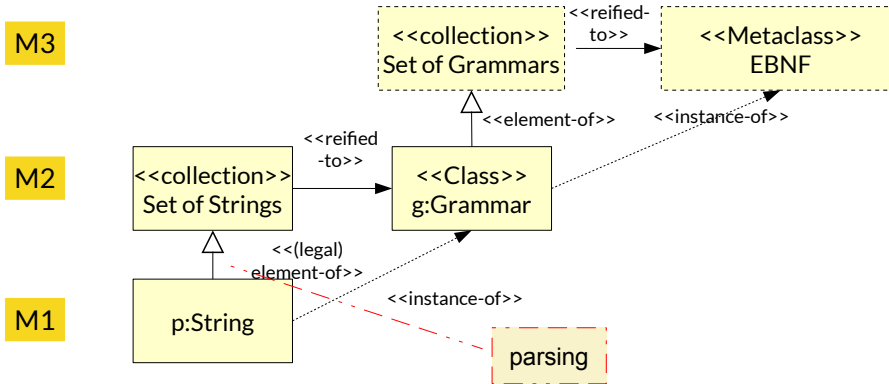


Problem 1 of Parsing

- ▶ 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

- ▶ 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



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)	$P = [P1 P2]$
$n \{ \dots \} m$	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	$X = B + C *text*$
< 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

- ▶ Since the 90s, many parser generators have been built for C/C++
 - Cocktail's lalr, ell, lark 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

/Users/bovet/ Demo/objc.g

- 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
;
Enter rule name:
iteration_statement
: 'while' LPAREN expression RPAREN statement
| 'do' statement struct_or_union_specifier
| 'for' LPAREN storage_class_specifier? RPAREN statement
;
jump_statement
: 'goto' identifier struct_declaration_list
| 'continue' SEMI struct_declaration_list
| 'break' SEMI struct_declarator_list
| 'return' expression struct_declarator_list
;
statement
: statement_list
;
string

```

Zoom Show NFA

Syntax Diagram Interpreter Debugger Console

129 rules 452:23

© Prof. U. AGmann

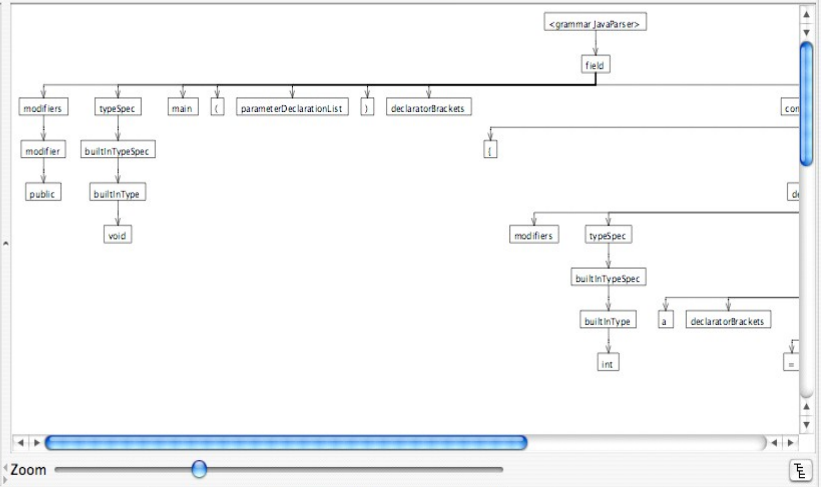


- handler
- expression
- expressionList
- assignmentExpression
- conditionalExpression

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

field

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



Zoom

Syntax Diagram Interpreter Debugger Console

132 rules 528:1

© 2004-11 APD Systems

ANTLR v3 IDE showing the parsing of a Java class. The main window displays the source code and the parse tree.

```
variableDeclaratorId
    : Identifier ('{' '}')*
    ;

variableInitializer
    : arrayInitializer
    | expression
    ;

arrayInitializer
    : '{' (variableInitializer (',' variableInitializer))* ('}'
    ;

modifier
    : annotation
    | 'public'
    | 'protected'
    | 'private'
    | 'static'
    | 'abstract'
    | 'final'
    | 'synchronized'
    | 'transient'
    | 'volatile'
    | 'strictfp'
    ;
```

```
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

Parse Tree

```
graph TD
    compilationUnit --> typeDeclaration
    typeDeclaration --> classOrInterfaceDeclaration
    classOrInterfaceDeclaration --> modifier
    classOrInterfaceDeclaration --> classDeclaration
    modifier --> public
    classDeclaration --> normalClassDeclaration
    normalClassDeclaration --> class
    normalClassDeclaration --> Sample
    normalClassDeclaration --> classBody
    classBody --> classBodyDeclaration
    classBodyDeclaration --> modifier
    modifier --> public
```

Input

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

Output

Parse-Tree

AST

Stack

Events

Syntax Diagram

Interpreter

Debugger

Console

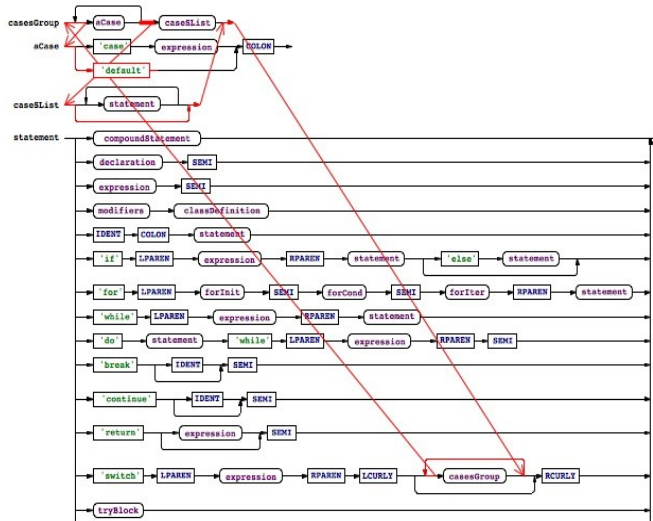
148 rules (2 warnings) 254.9 Warnings reported in console



Zoom Show NFA

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

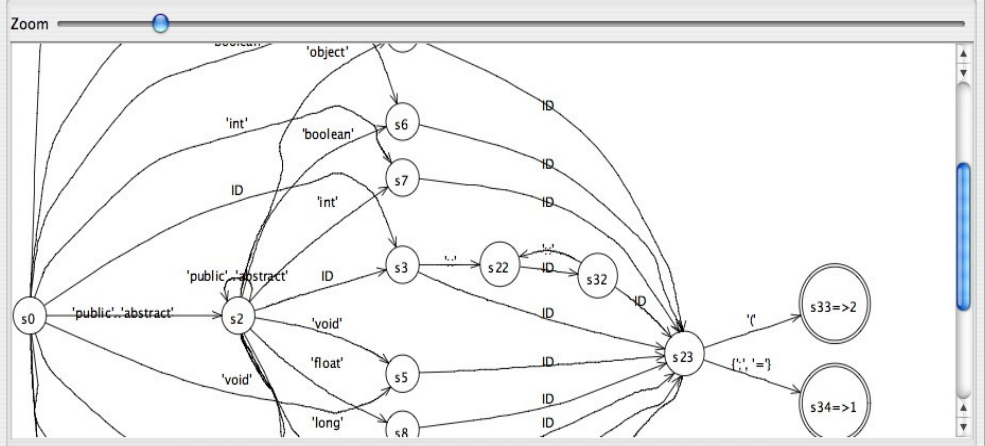
Alternatives: 1 2



Syntax Diagram Interpreter Debugger Console

132 rules (6 warnings) 433:1

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



© Prof. U. Altmann



20.1.2 Program Interpretation While Parsing

- ▶ ANTLR knows several *actions* (*snippets*, *fragments*) in a rule
 - *semantic actions*: modifying attributes of nodes or temporary values
 - *emissions*: emitting tokens of an output alphabet, e.g., for code generation
 - *motions*: emitting an action on a shared environment (“motion grammars”)
 - e.g., for controlling a robot
 - Motion grammars usually influence parsing also!

Actions Calculating Attributes

- ▶ 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 = $.value;}
  | ( '+' e=multExpr {$value += $.value;}
    | '-' e=multExpr {$value -= $.value;}
    )*
  ;

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

// lexical rules
ID  : ('a'..'z'|'A'..'Z')+ ;
INT : '0'..'9'+ ;
NEWLINE: '\r'? '\n' ;
WS   : (' '\t')+ {skip();} ;
```



```
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);
        // Link lexer tokens with parser parser
        ExprParser parser = new ExprParser(tokens);
        // rone the parser parser
        parser.prog();
    }
}
```

The screenshot displays a Java IDE window titled `/Users/bovet/ Grammars/Demo/Expr.g`. The editor shows a grammar definition for expressions in Java. The grammar rules are as follows:

```
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;}
;
}
```

Below the editor, the IDE shows the execution of the grammar on the input `2+3*4`. The resulting syntax tree is:

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

The IDE interface includes a toolbar with options for `Syntax Diagram`, `Interpreter`, `Debugger`, and `Console`. The status bar at the bottom indicates `9 rules 1:1 Writable`.

The screenshot displays a parser debugger interface with the following components:

- Source Code:** Located at the top, showing the implementation of grammar rules. The rule `atom` is highlighted in yellow. The code includes methods for `expr`, `multExpr`, and `atom`, along with their respective return types and implementations.
- Parse Tree:** A tree diagram on the left side of the debugger window showing the current state of the parse. The root node is `root`, which contains `prog`, `stat`, `expr`, and `multExpr`. The `multExpr` node contains `atom`, which in turn contains the integer `2`.
- Stack:** A table on the right side of the debugger window showing the current stack of frames. It lists the rule names and their indices: `prog` (0), `stat` (1), `expr` (2), `multExpr` (3), and `atom` (4).
- Debugger Controls:** At the bottom, there are buttons for `Input`, `Output`, `Parse Tree`, `AST`, `Stack`, and `Events`. There is also a `Break on` section with options for `All`, `Location`, `Consume`, `LT`, and `Exception`.

The screenshot shows a Java IDE window titled "/Users/bovet/ Grammars/Demo/Expr.g". The editor displays a grammar definition for expressions. The code includes package declarations, imports, and several methods: `prog`, `stat`, `expr`, `multExpr`, `atom`, and `expr returns`. A yellow highlight is placed on the line `prog: stat+ ;`. Below the editor, the "Parse Tree" view is active, showing a tree structure for the input "2+3*4". The root node is "root", which branches into "prog", "stat", and an empty node. "stat" branches into "expr" and another empty node. "expr" branches into "multExpr", "+", and "multExpr". The left "multExpr" branches into "atom", which contains the value "2". The right "multExpr" branches into "atom", "*", and "atom". The left "atom" contains "3" and the right "atom" contains "4". The IDE interface includes a toolbar with navigation and execution icons, a "Break on:" section with "Location" checked, and a status bar at the bottom showing "9 rules 15:15 Writable".

Applications of String Grammars

Applications can be 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 (RTG)

- 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 **explicitly**, with tree node constructors
- ▶ ENBF-rule for Tree Grammar Rule:
TreeNode → constructor '(' Treenode // ';')
- ▶ Example:
Model → ModelElements *

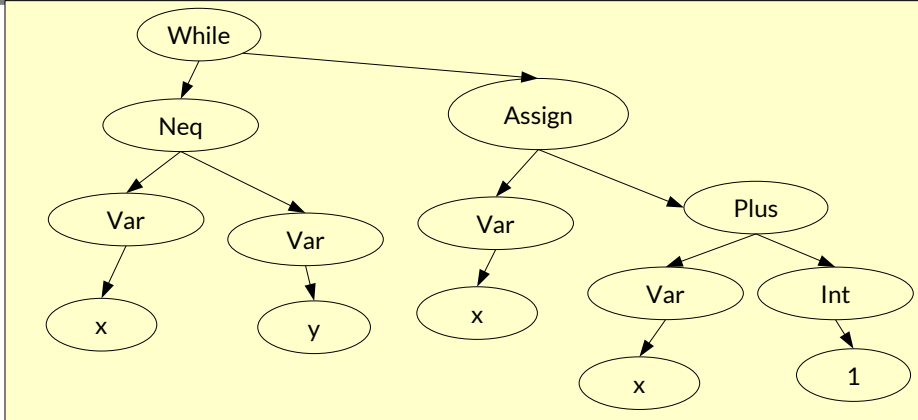
// Regular Tree Grammar from Stratego regular tree grammar TIL

```
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>
Id          -> <string>
```



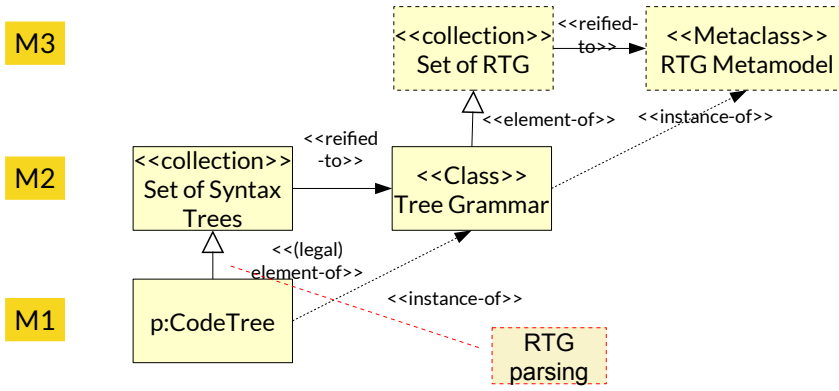
Correct Instance?

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



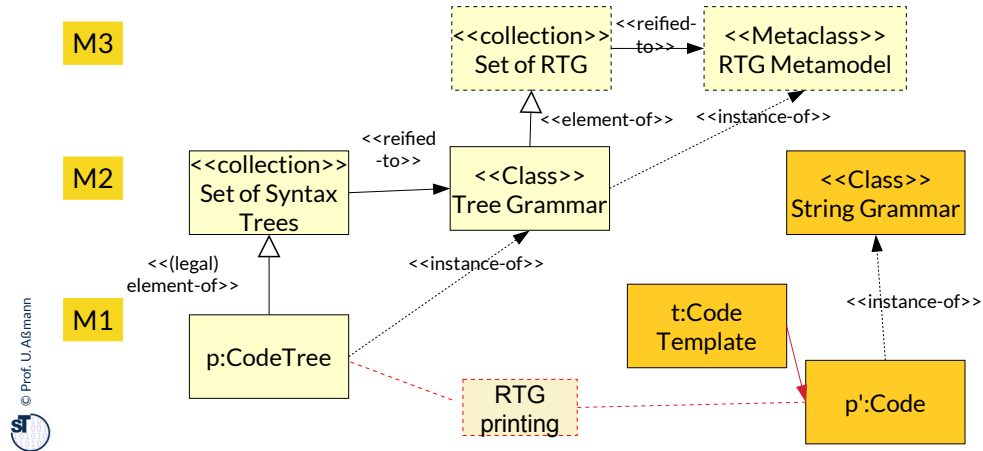
Tree Parsing with RTG

- ▶ 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



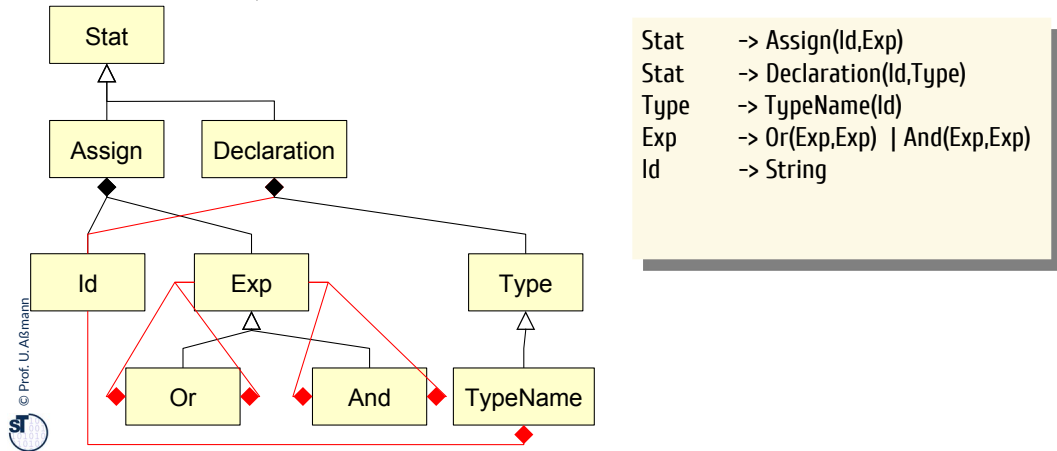


20.2.2. Metamodels and Tree Grammars

- Full parser also build syntax trees - Design Pattern Builder

Grammars and Metamodels are Very Similar

- ▶ The nonterminals of a grammar form the *name space* of the grammar
- ▶ Definition of a nonterminal (on left side)
- ▶ Use of a nonterminal (on right side) induces relation to definition
- ▶ Metamodels are graphs (including the red edges); grammars are *link trees* (red edges are overlaid)





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 (concrete syntax, CS)
 - **However, the syntax tree must be built as “action” while parsing**
- ▶ After parsing, the parser creates an **(abstract) syntax tree (AST)**, i.e., builds up a tree with regard to a **regular tree grammar of the abstract syntax (AS)**
 - Recognized nonterminals have to be mapped
 - Tokens, keywords, comments, layouts have to be omitted
 - **Tree building:** Treenodes have to be allocated and composed
- ▶ This **CS-AS mapping (from concrete to abstract syntax)** is created by hand in *side actions* of the parser
 - as action snippet in the rules
- ▶ 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)

- ▶ CS-AS mapping works via side actions of the grammar rules (“transducer”)
- ▶ 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

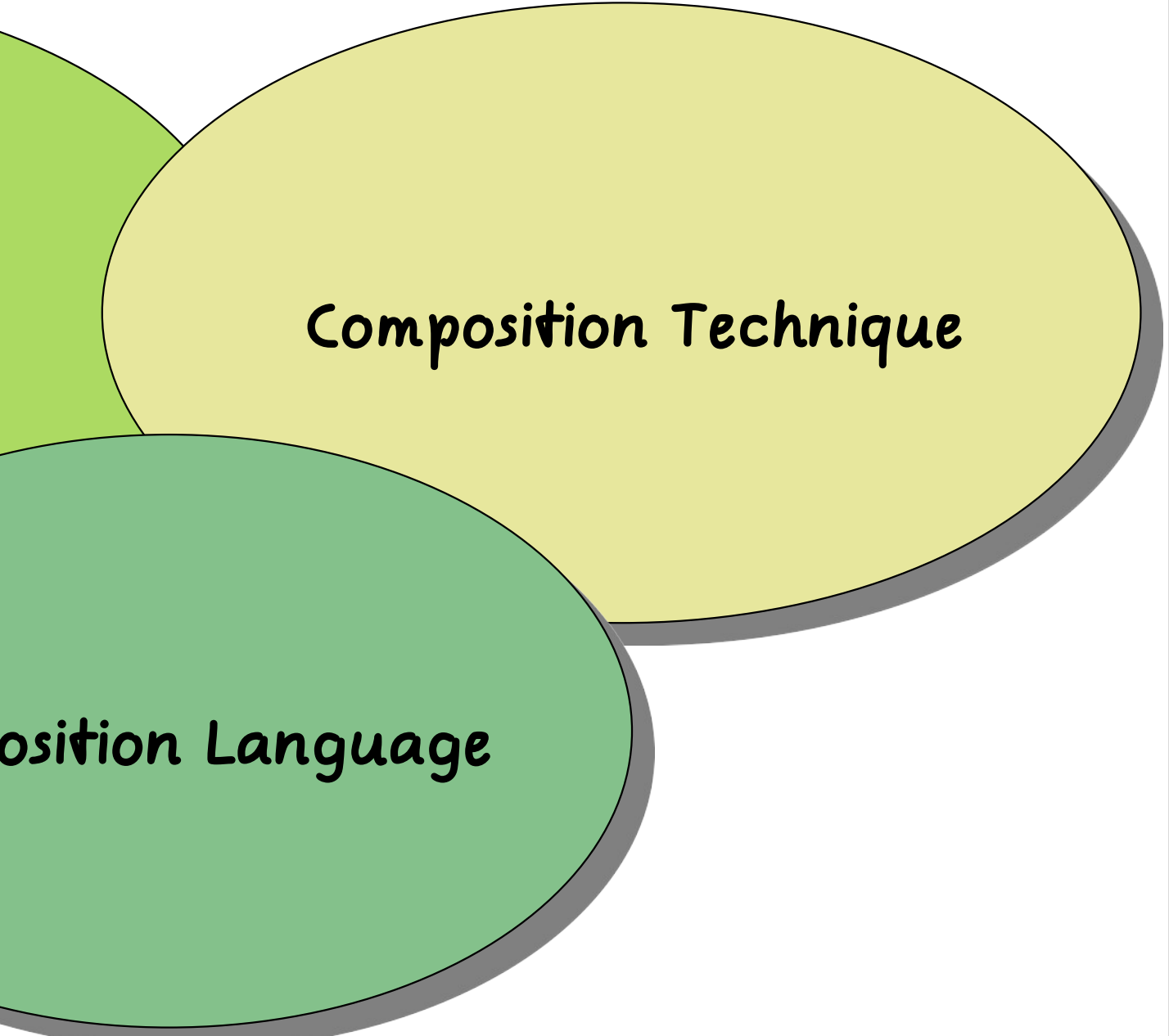


10.4 Text and Tree Algebrae

for composition of texts and trees

Composition Systems

(OST)



Composition Technique

Composition Language

Prolog in Mathematics

(OST)

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`

Excursion: CSV and Digital Preservation

- ▶ <https://digital-preservation.github.io/>
- ▶ Digital preservation “Bewahrung des digitalen Kulturerbes”
- ▶ csv schema language
<https://digital-preservation.github.io/csv-schema/csv-schema-1.1.html>
- ▶ <https://jsonformatter.org/xml-parser> online xml parser

- ▶ 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`

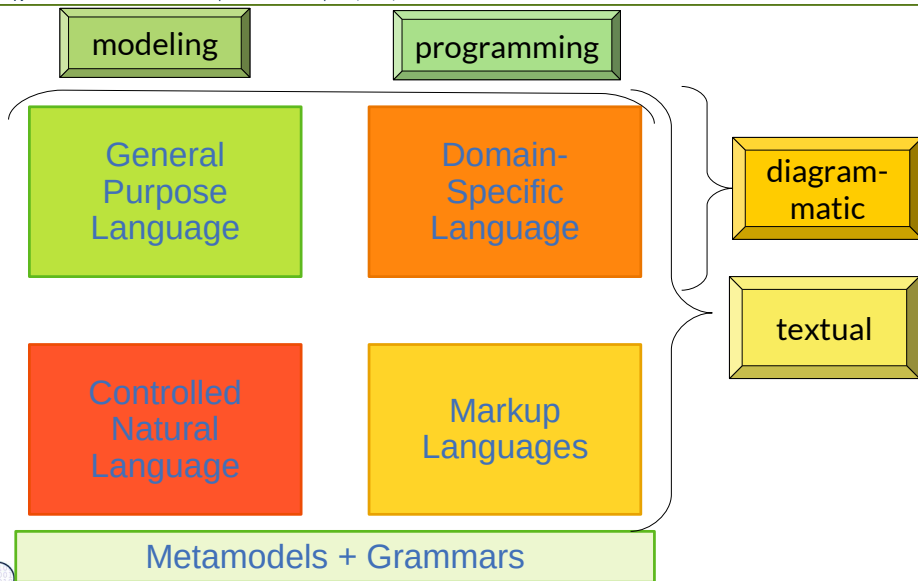


20.5 Controlled Natural Languages (CNL)

- Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguistics*, 40(1):121--170, 2014.
- Anne Cregan, Rolf Schwitter, and Thomas Meyer. Sydney owl syntax - towards a controlled natural language syntax for owl 1.1. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*, volume 258. CEUR-WS, 2007.

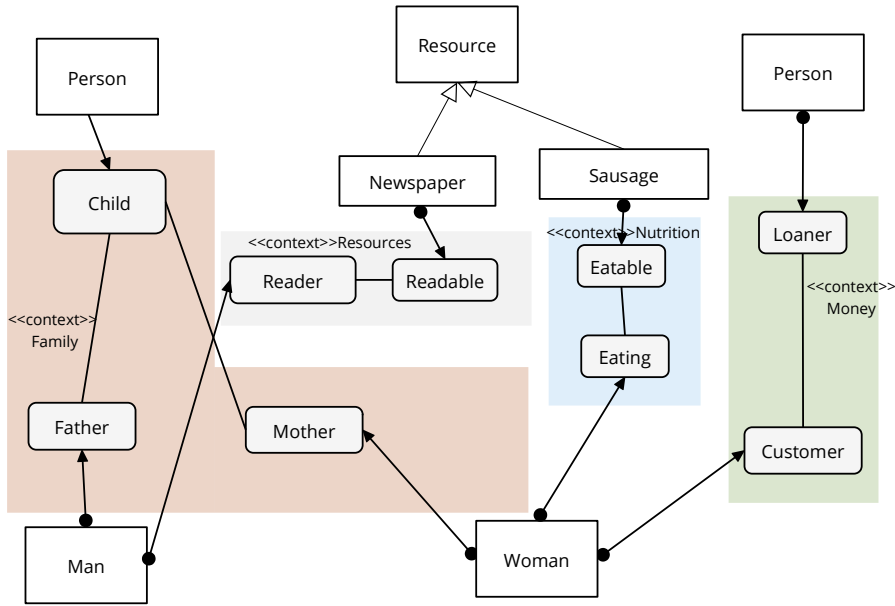
Q16: Languages in Software Factories

49 Model-Driven Software Development in Technical Spaces (MOST)



- ▶ Def.: A **controlled natural language** is a restricted natural language that can be described by a formal grammar, either a
 - context-free grammar (e.g., specified by ANTLR)
 - regular tree grammar (e.g., specified by Spoofax)
 - context-sensitive grammar or attributed grammar (e.g., specified by JastAdd, see later).
- ▶ Advantages:
 - Simple, comprehensive syntax
 - In contrast to a natural language, a CNL can be processed easily by tools, because syntax trees for naturally looking texts can be built

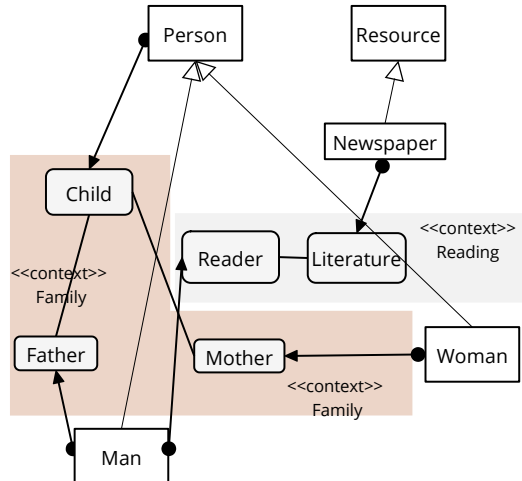
Families and Banks in Natural and Role Types (CROM)



Families and Banks in Natural and Role Types, Specified in Sydney OWL Syntax

- ▶ Models can be read or specified in controlled natural language
- ▶ [Cregan] introduces a CNL for OWL ontologies, also useful for CROM
- ▶ Example: Families in CROM:

Every Man is a Person.
Every Woman is a Person.
Every Person may play a Child (in a Family).
Every man:Man may play a Father (in a Family).
Every Child is related to a Mother in a Family
Every Child is related to a Father in a Family.
A man:Main may play a Reader (in Reading).
Every Reader is related to a Literature in Reading.
Every Newspaper is a Resource.
Every Newspaper may play a Literature (in Reading).



PENS Classification

- ▶ [Kuhn] Languages can be classified with regard to 4 criteria and school grades.
- ▶ E.g., English is S²E⁵N⁵S².

	Precision	Expressiveness	Naturalness	Simpliicty
English	1	5	5	1
Controlled English	4	3	5	4
Domain-specific language	4	3	5	4
UML Class diagrams	3	3	4	4





20.6 Markup Languages and Pseudocode

<http://en.wikipedia.org/wiki/Pseudocode>
Languages used for documentation

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)

Examples for Pseudocode

- ▶ 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 ungueltig
    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>

Summary

- ▶ 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 programs, 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)

20.A.1 Port-Graph Algebrae on Fragments

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

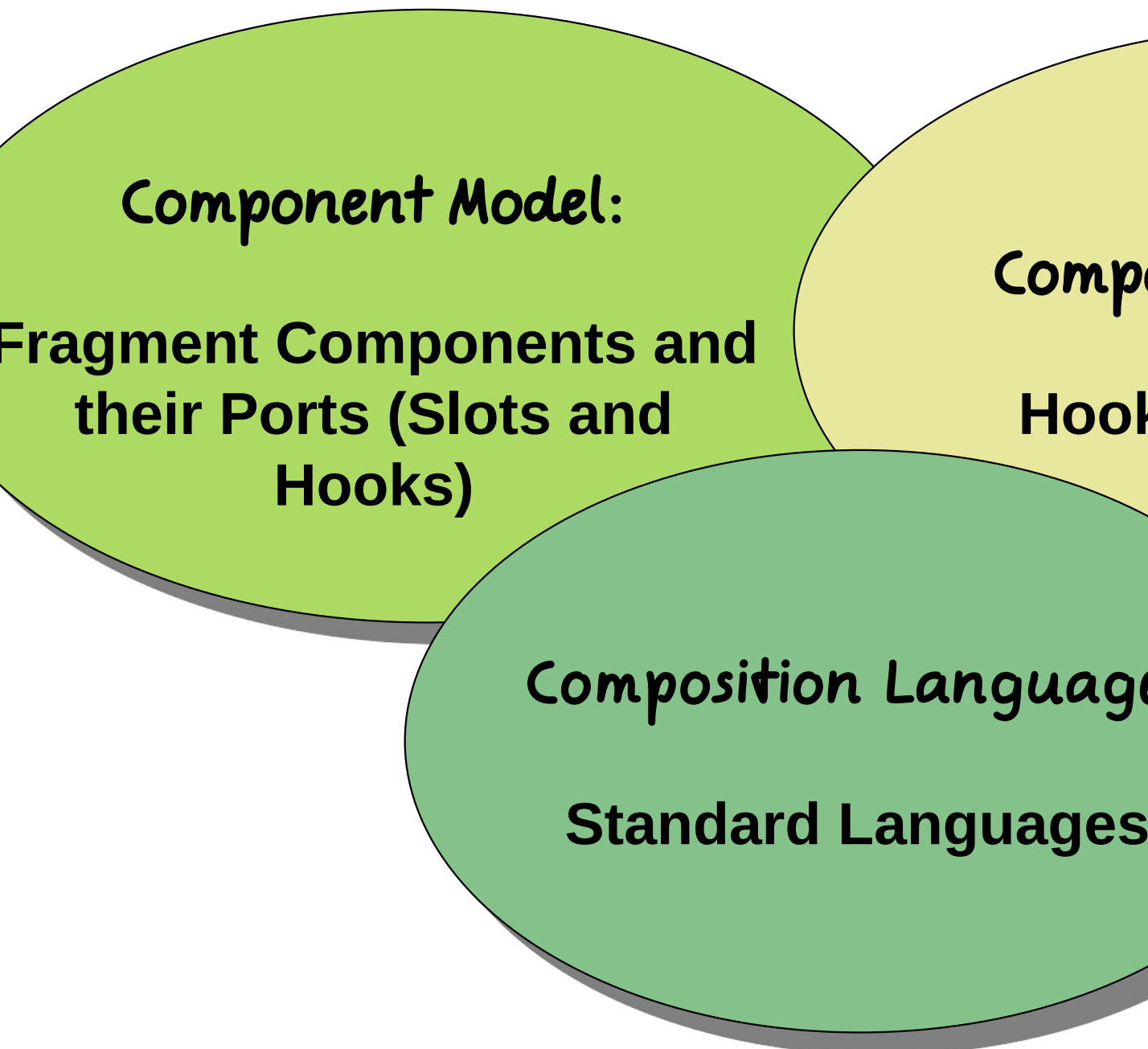
... based on port-graph algebrae

... with Graybox Components

... preview onto the summer (CBSE course)

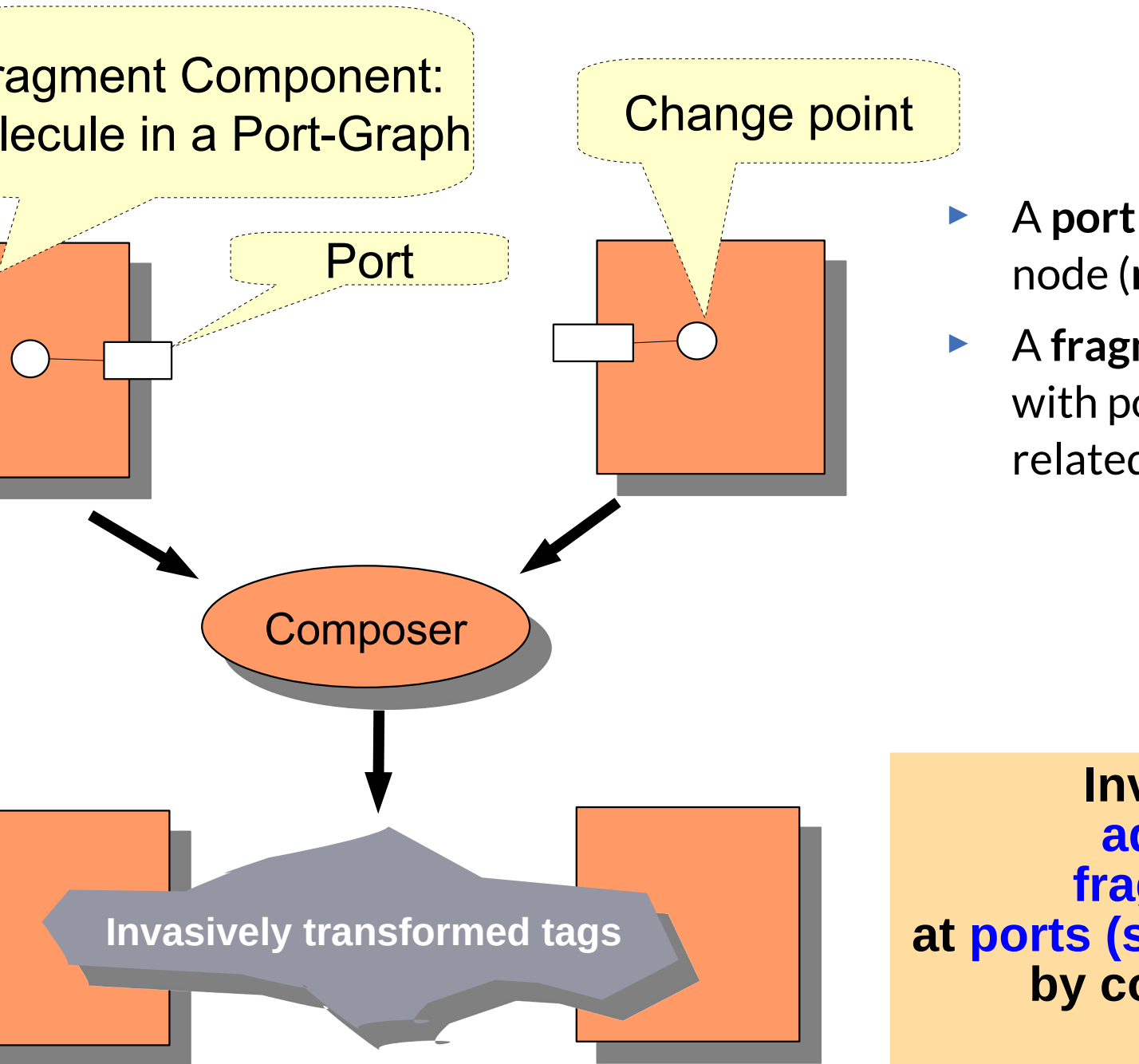
“Invasive” Composition (Typed Template Algebrae)

Model-Driven Software Development in Technical Spaces (MOST)

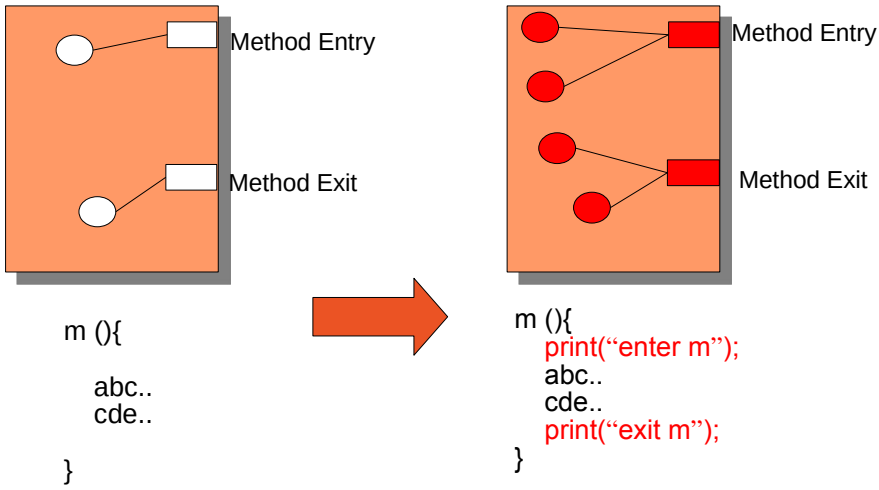


Invasive Composition as Hook Transform

Model-Driven Software Development in Technical Spaces (MOST)

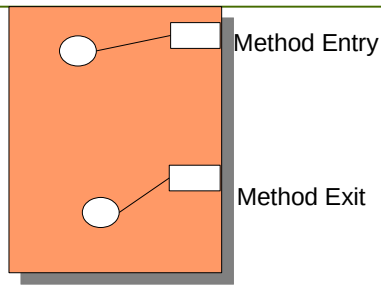


Binding Implicit Hooks with Fragments



```
component.findHook(„MethodEntry“).extend(“print(“\nenter m”);”);  
component.findHook(„MethodExit“).extend(“print(“\nexit m”);”);
```

Partial Parsing of Fragment Components



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

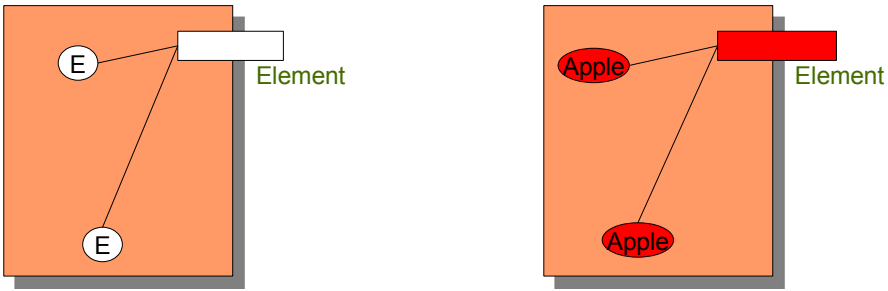


```
m(){  
  // Method Entry  
  abc..  
  cde..  
  // Method Exit  
}
```

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



Binding Declared Hooks with Fragments



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

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

```
...
```

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

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

```
...
```

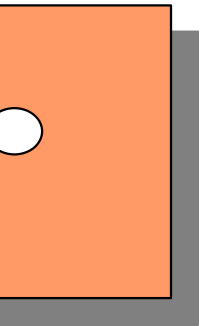
```
box.findHook(„Element“).bind(“Apple”);
```



s Hook Transformations

(OST)

- ▶ 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