

## Teil II. Elementare Analysewerkzeuge

1

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 13-1.0, 05.12.13

Softwareentwicklungswerkzeuge (SEW) © Prof. Uwe Aßmann

## 20. Parser-Generatoren im Technikraum Grammarware

2

- 1) Grundlagen
- 2) Beispiel Taschenrechner

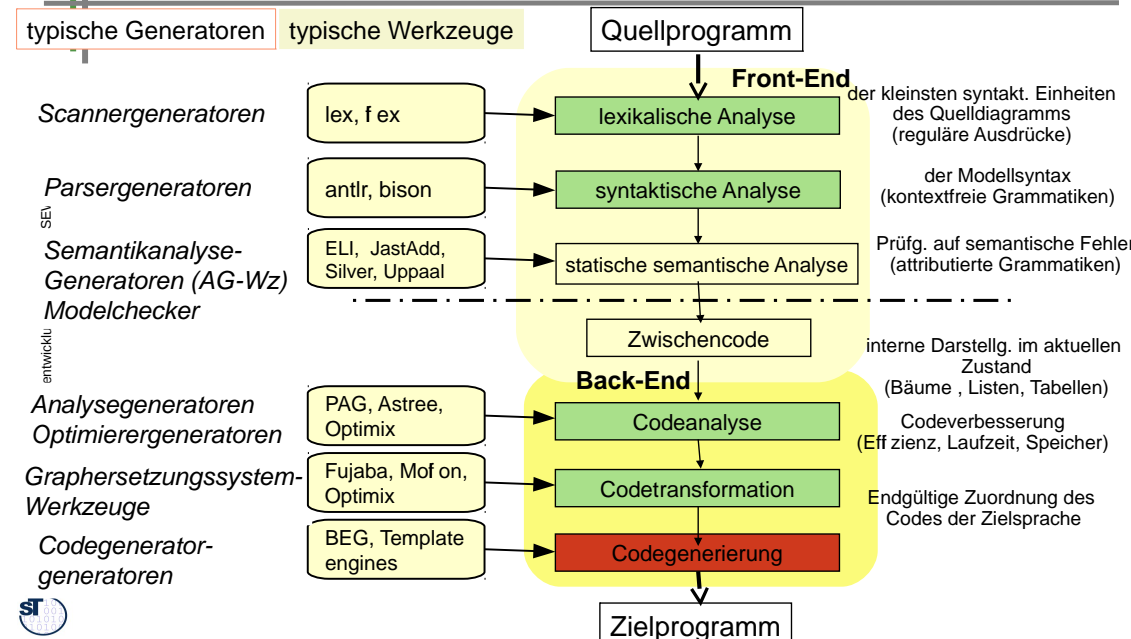
Softwareentwicklungswerkzeuge (SEW) © Prof. Uwe Aßmann

## Literatur

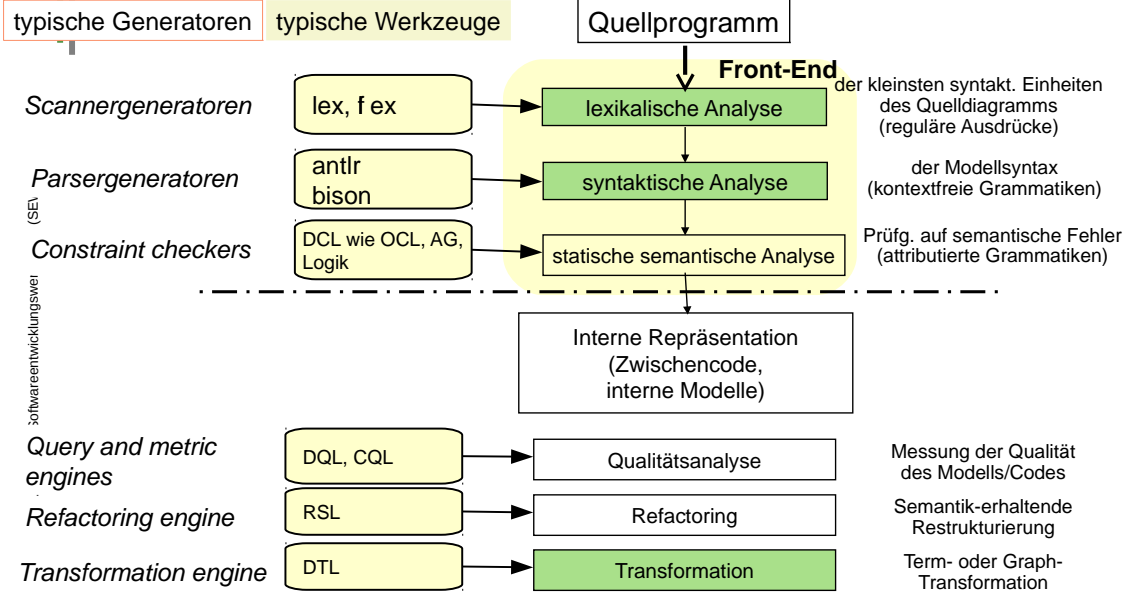
- ▶ Obligatorisch:
- ▶ <http://www.antlr.org>
- ▶ Zusätzlich:
  - Cocktail [www.cocolab.de](http://www.cocolab.de), die Compiler-Toolbox für die schnellsten Compiler der Welt (kommerziell, Demoversionen erhältlich)

## Phasen eines Werkzeugs und ihre erzeugenden Werkzeuge

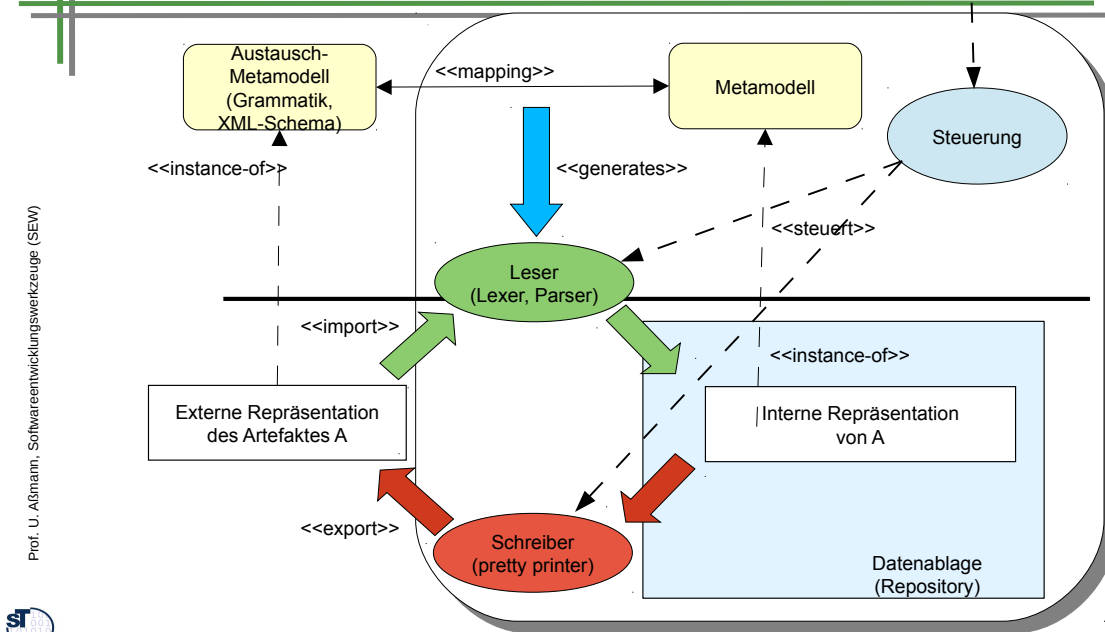
4



## 5 Phasen eines Importers in ein Repository und die erzeugenden Werkzeuge



## 6 Wdh: Nutzung von generierten Parsern zum Import in Repositorien



## 7 Problem

- Parsen eines Programms, Modells oder Artefakts bedeutet, seine kontextfreie Syntax zu erkennen
- Parser sind die ersten Phasen eines Werkzeugs, denn es muss ein Artefakt importieren und damit ihn parsen
- Parsen erzeugt einen *Syntaxbaum*
- Parser wurden ursprünglich von Hand geschrieben (Compilerbau), heute generiert man sie aus *Grammatiken in EBNF*
  - **Parser** erkennt die Struktur des Textes anhand der Grammatik ("Zerteiler")
  - **Baumaufbauer (Baumkonstruktor)** erzeugt einen abstrakten Syntaxbaum. Normalerweise wird er mit dem Parser verschränkt.

Wie arbeite ich flexibel mit mehreren Programmiersprachen oder DSL?

## 8 Antwort

- Bidirektionale Abbildung zwischen Technikum "Grammarware" und einem anderen Technikum, wie z.B. "Treeware" oder "Modelware"

In dem ich aus Grammatiken Parser (Zerteiler und Baumkonstruktoren) generiere  
und  
zusätzlich Prettyprinter (Codegenerator)

## Beispiel EMFText

- ▶ Nutzt Parser-Generator ANTLR zur Generierung von Parsern
  - Parser und Metamodell werden aufeinander abgebildet (mapping), um konkrete auf abstrakte Syntax abzubilden
- ▶ Nutzt schablonengesteuerte Codegenerierung zur Erzeugung von Text und Programmen (siehe später).

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

- ▶ In den 90er Jahren gab es für C viele Parsergeneratoren
  - Cocktail's lalr, ell, lark [www.cocolab.de](http://www.cocolab.de)
  - fnc2
  - flex und bison (gnu)
- ▶ Für Java ist ANTLR populär geworden
  - LL(k)
  - Generierter Parser mit Algorithmus "rekursiver Abstieg"
  - Etwas "gefärbte" Seite mit Geschichte [http://www.bearcave.com/software/antlr/antlr\\_expr.html](http://www.bearcave.com/software/antlr/antlr_expr.html)

The screenshot shows the ANTLRWorks IDE interface for the EMFText grammar. The top window displays the grammar rules, including `compound_statement`, `statement_list`, `selection_statement`, `iteration_statement`, and `jump_statement`. A dialog box prompts for an "Enter rule name:". The bottom window shows a syntax diagram for the `while` loop, illustrating the flow from the `while` keyword through `LPAREN`, `expression`, `RPAREN`, and `statement` to the next `while` keyword. The status bar at the bottom indicates 129 rules and 452:23 lines.

The screenshot shows the ANTLRWorks IDE interface for the JavaParser grammar. The top window displays the grammar rules, including `handler`, `expression`, `expressionList`, `assignmentExpression`, and `conditionalExpression`. The bottom window shows a syntax diagram for the `field` declaration, illustrating the flow from `modifiers` through `typeSpec` to `main`, `LPAREN`, `parameterDeclarationList`, `RPAREN`, and `declaratorBrackets`. The status bar at the bottom indicates 132 rules and 528:1 lines.



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

```

interfaceBodyDeclaration
interfaceMemberDecl
interfaceMethodOrFieldDecl
interfaceMethodOrFieldRest
methodDeclaratorRest
voidMethodDeclaratorRest
interfaceMethodDeclaratorRest
voidInterfaceMethodDeclaratorRest
constructorDeclaratorRest
constantDeclarator
variableDeclarators
variableDeclarator
variableDeclaratorRest
constantDeclaratorRest
variableDeclaratorRest
variableDeclaratorId
arrayInitializer
modifier
annotation
public
protected
private
static
abstract
final
native
synchronized
transient
volatile
strictfp

```

Parse Tree

```

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

```

Stack

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

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

/Users/bovet/ Grammars/java.g

Zoom (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

/Users/bovet/mantra.g

```

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

```

Decision 10 of "classDefinition"

Syntax Diagram Interpreter Debugger Console

59 rules (1 warnings) 56:5

## 13.2 Ein Taschenrechner

```

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

```

## Ansteuerung

```

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();
    }
}

```

The screenshot shows the ANTLR IDE interface. The top pane displays the source code from slide 17. The bottom pane shows a syntax tree for the input '2+3\*4'. The tree structure is as follows:

```

graph TD
    prog --> stat
    stat --> expr
    expr --> multExpr1
    expr --> plus
    expr --> multExpr2
    multExpr1 --> atom1
    atom1 --> 2
    multExpr2 --> atom2
    atom2 --> 3
    multExpr2 --> plus
    multExpr2 --> atom3
    atom3 --> 4

```

The status bar at the bottom indicates '9 rules 1.1 Writable'.

The screenshot shows the ANTLR IDE interface with the parse tree view selected. The parse tree structure is as follows:

```

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

```

The status bar at the bottom indicates '9 rules 35:13 Writable'. The 'Stack' pane on the right shows the current stack of rules: prog, stat, expr, multExpr, atom.

The screenshot shows a Java IDE window titled "/Users/bovet/ Grammars/Demo/Expr.g". The main editor displays a grammar definition for expressions. The grammar includes a header with package and import statements, a lexer header, and a members section with a map for memory. The main grammar rules are:

```

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=multiExpr {$value = $e.value;}
| "+" e=multiExpr {$value += $e.value;}
| "-" e=multiExpr {$value -= $e.value;}
;

```

Below the editor, the "Parse Tree" view is active, showing a tree structure for the input "14". The root node is "root", which contains "prog", which contains "stat", which contains "expr". The "expr" node branches into "multiExpr", "+", and "multiExpr". Each "multiExpr" node further branches into "atom", which contains the integers "2" and "4".

## Was haben wir gelernt?

- ▶ Parsergeneratoren gehören heute zum Werkzeugesatz jeden Softwareingenieurs
- ▶ Neben Cocktail gibt es freie Initiativen, z.B. ANTLR
- ▶ Leider erfasst der Parser nur die kontextfreie Struktur des Programms oder Dokuments; Kontextbedingungen und Integritätsbedingungen bleiben der *statischen semantischen Analyse* vorbehalten.

## The End