ANTLR
'ANother Tool for Language Recognition'

24. März 2009

Gerhard Schlemm

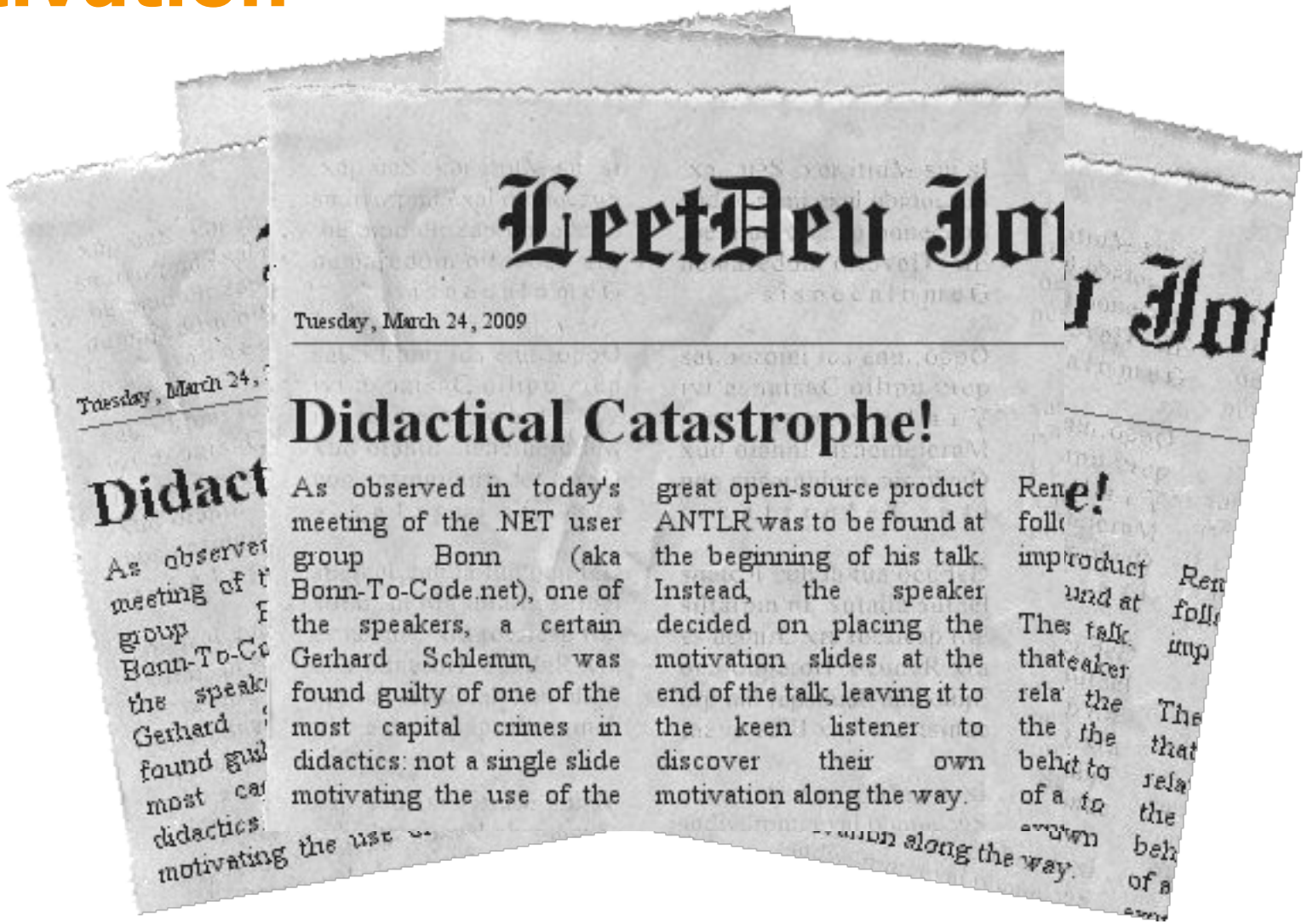
gerhard.schlemm@comma-soft.com
http://www.infonea.de, XING-Profil

TOP 100 2008
COMMA SOFT AG

Agenda

- Motivation
- Was ist ANTLR?
- Lexer, Parser, Tree Walker
- Code-Generierung
- Demo
- Domain Specific Language (DSL)
- Fazit

Motivation


LeetDev Jo
Tuesday, March 24, 2009
Didactical Catastrophe!
As observed in today's meeting of the .NET user group Bonn (aka Bonn-To-Code.net), one of the speakers, a certain Gerhard Schlemm, was found guilty of one of the most capital crimes in didactics: not a single slide motivating the use of the great open-source product ANTLR was to be found at the beginning of his talk. Instead, the speaker decided on placing the motivation slides at the end of the talk, leaving it to the keen listener to discover their own motivation along the way.

Motivation

- Motivation am Ende des Vortrags beziehungsweise selbst zu entdecken!
- ANTLR - nur ein Tool für den Compiler-Bau?
- Was möchten Sie mit ANTLR tun?
  - Wie können Sie ANTLR auf der Ebene der technischen Softwareentwicklung einsetzen?
  - Kann ANTLR im Umgang mit einer kundenspezifischen Domain Specific Language (DSL) behilflich sein?

Was ist ANTLR?

- antlr.org: ... 'is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions'
- Kurz: ANTLR ist ein Werkzeug zur **Erkennung** und **Verarbeitung** formaler Sprachen
- Technik: Code-Generierung für viele Plattformen (Programmiersprachen) ergänzt durch Laufzeitbibliotheken
  - ANTLR selbst ist in Java entwickelt

Erkennen und Verarbeiten (1)

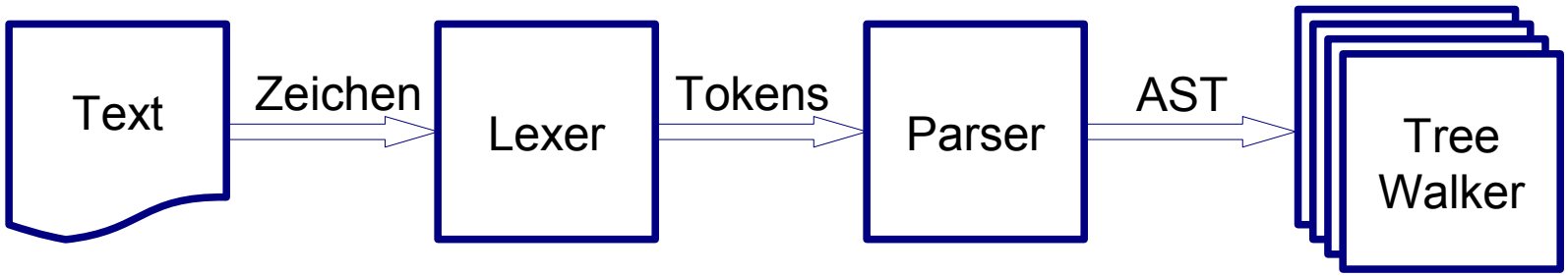
- **Erkennen**
  - Benötigt werden das Vokabular und
  - die Strukturierungsregeln der Sprache.
  - Diese Spezifikation diese beiden Aspekte werden im Zusammenspiel als Grammatik bezeichnet
- **Verarbeiten**
  - ANTLR bietet viele Möglichkeiten, Verarbeitung auf einer erkannten Struktur auszuführen
  - Verarbeitung durch fachspezifischen Code in C#, an geeigneten Stellen hinterlegt

Erkennen und Verarbeiten (2)

- **Verarbeiten** (cont.)
  - Viele Varianten möglich, von einfach bis komplex
    - Einfach: Verarbeitungsschritte direkt in Grammatik schreiben und bei Erkennung einer Strukturregel ausführen
    - Komplex: ANTLR aussagekräftige und kompakte Struktur erzeugen lassen (AST) und diesen ggf. mehrfach verarbeiten (Verarbeitungsphasen)

Lexer, Parser und Tree Walker

- ANTLR generiert drei Arten von Komponenten
  - **Lexer** – lexikographische Analyse
  - **Parser** – Struktur erkennen und Inhalt in abstrakten Syntaxbaum (AST) überführen
  - AST mehrfach mit **Tree Walker** verarbeiten



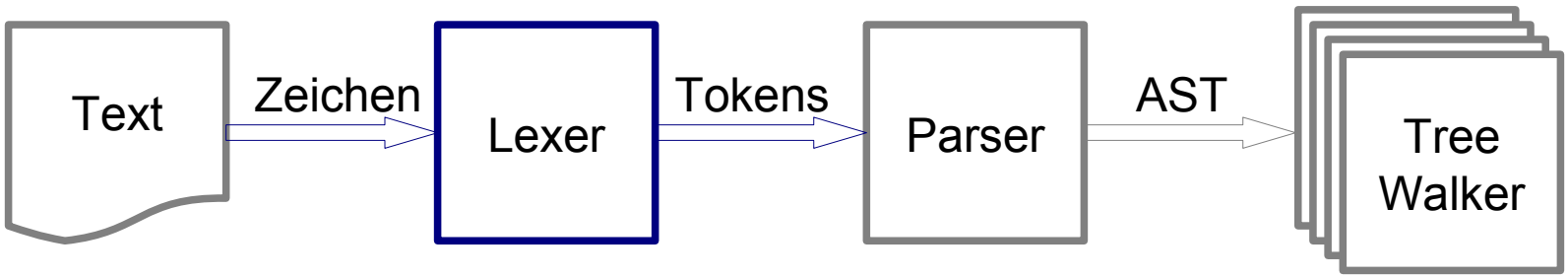
Lexer, Parser und Tree Walker

- Andere Szenarien
  - Nur Lexer
  - Lexer und Parser ohne AST
    Regeln werden sofort bei Erkennung ausgeführt
  - Lexer + Parser + Tree Walker(s)
  - Bäume deklarativ transformieren

  - Demo zeigt Lexer + Parser + Tree Walkers

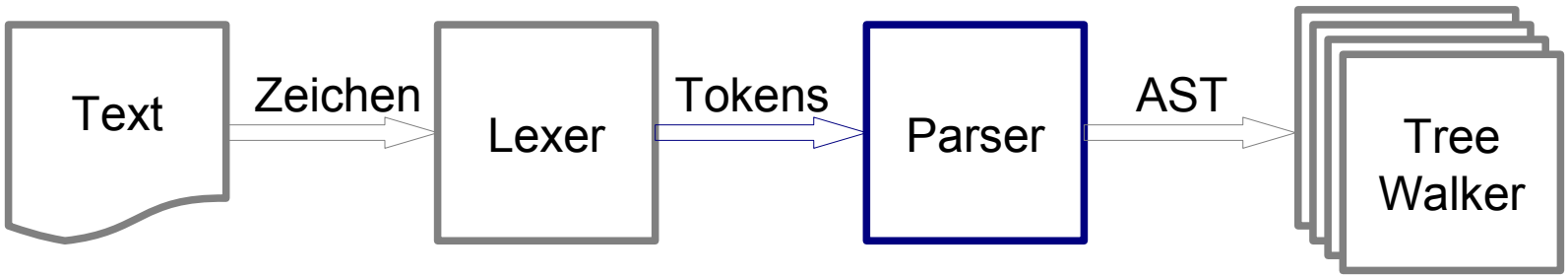
Lexer

- Überführt Zeichenfolge -> Tokenfolge
- Lexikographische Analyse
  - Der Lexer kennt das Vokabular der Sprache und zerlegt die Zeichenfolge als eine Folge von **Tokens**. Jedes Token symbolisiert ein Wort im Vokabular.
  - Beachte: Vokabular hat Struktur, z.B. der Zahlenwert 71.9
- I.d.R. kein eigener Code erforderlich



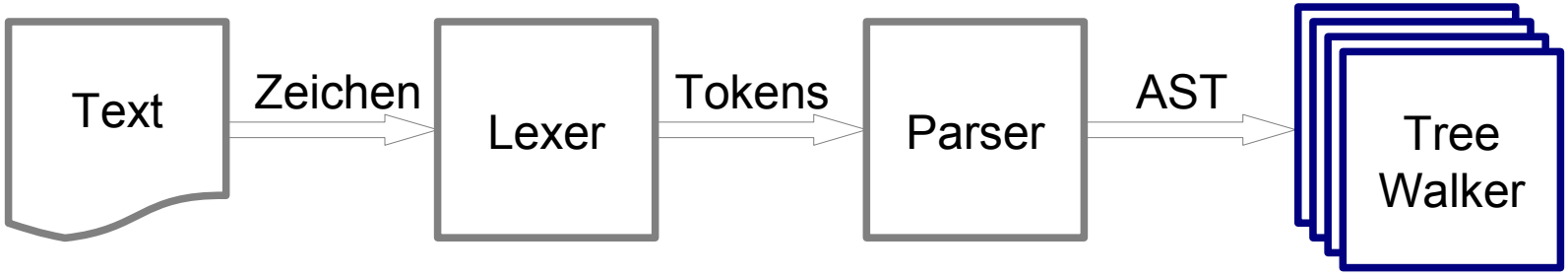
Parser

- Überführt Tokenfolge in Datenstruktur
- Erkennen der grammatikalischen Struktur anhand der angegebenen **Regeln**
  - Dabei Aufbau einer kompakten, aussagekräftigen Datenstruktur, häufig ein Abstrakter Syntaxbaum (AST) zur weiteren Verarbeitung
- I.d.R. kein eigener Code erforderlich

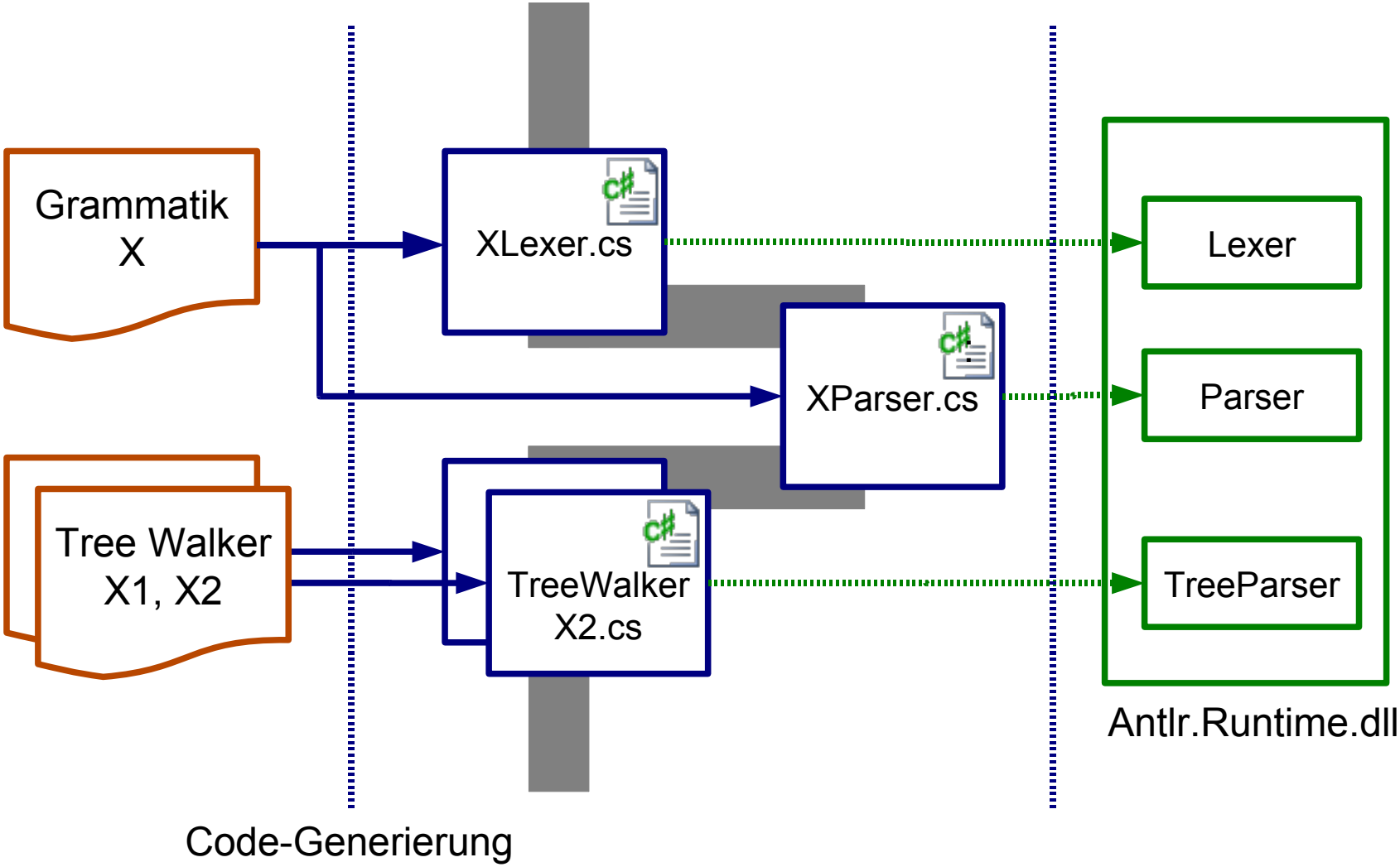


Tree Walker

- Verarbeitung der Informationen in dem vom Parser erzeugten Abstrakten Syntaxbaum (AST)
  - Bei komplexen Problemen sind mehrere sich ergänzende Verarbeitungsschritte normal - jedem solchen Pass könnte ein Tree Walker zugeordnet sein (SoC)
  - Tree Walker ermöglicht es, AST regelgesteuert zu durchwandern und dabei gezielt zu verarbeiten



Code-Generierung (1)



Code-Generierung (2)

- ANTLR generiert aus Grammatiken Code für Lexer, Parser und Tree Walker

- In diesen generierten C#-Code fließen eigene Anweisungen zur Verarbeitung ein
  - Wo wird der eigene Code spezifiziert?

Code-Generierung (3)

- Annotation der Regeln durch **Aktionen** (eigener C#-Code) in der Grammatik
- Code wird in von ANTLR generierte C#-Datei (Lexer, Parser oder Tree Walker) eingewoben

- Vordefinierte Punkte zum Injizieren eigenen Codes sind vielfältig
  - vor, nach und in einer Regel

Code-Generierung (4)

- Arbeiten an einer Grammatik mit eingebetten Aktionen ist unergonomisch
  - Code so gut wie möglich aus der Grammatik trennen
- Auslagerung eigenen Codes in Basisklassen (Lexer, Parser, Tree Walker) möglich
- Arbeit mit partiellen Klassen

- Designziel: So viel deklarativ wie möglich

DEMO

- Story: Arithmetische Ausdrücke (Integer-Werte) mit beliebig vielen Parametern
  - erfassen
  - verarbeiten
    - Berechnen der Funktion über die Parameter
    - Erzeugen einer dynamischen Methode
  - Beispiel: 7+4*(a+3)

DEMO

- Lexer
- Parser
  - Stufe 1: Nur Erkennen
  - Stufe 2: Abstrakten Syntaxbaum (AST) erzeugen
- Tree Walker
  - AST verarbeiten
    - Pass 1: Parameternamen ermitteln
    - Pass 2: Dynamische Methode für arithmetischen Ausdruck generieren

DEMO - LEXER

- Vokabular der Sprache umfasst
  - Parameter (Bezeichner aus Buchstaben)
  - Integer-Konstanten
  - Operatoren für Addition und Multiplikation
  - Strukturierende Klammern

DEMO - LEXER

```
lexer grammar ExpressionLexer;
options {
language=CSharp2;
}
PARAMETER       :       ('a'..'z'|'A'..'Z')+ ;
INTEGER         :       '0'..'9'+;
ADDITION        :       '+';
MULTIPLICATION  :       '*';
OPENBRACE       :       '(';
CLOSEBRACE      :       ')';
```

DEMO - LEXER

- Eingabe "7+4*(a+3)" führt zu Tokenfolge
  - INTEGER (text="7")
  - ADDITION
  - INTEGER (text="4")
  - MULTIPLICATION
  - OPENBRACE
  - PARAMETER (text="a")
  - ADDITION
  - INTEGER (text="3")
  - CLOSEBRACE

DEMO - LEXER

- Damit scheint noch nicht viel gewonnen ...
- Bei komplexeren Vokabularen ist es aber aufwendig und fehleranfällig, eigene performante Lexer zu schreiben

- Deshalb diese Aufgabe besser den Experten überlassen, die ihr Wissen per Code-Generator bereitstellen

DEMO

- Lexer
- Parser
  - Stufe 1: Nur Erkennen
  - Stufe 2: Abstrakten Syntaxbaum (AST) erzeugen
- Tree Walker
  - AST verarbeiten
    - Pass 1: Parameternamen ermitteln
    - Pass 2: Dynamische Methode für arithmetischen Ausdruck generieren

DEMO - PARSER

```
parser grammar ExpressionParser;

options {
language=CSharp2;
tokenVocab=ExpressionLexer; /* verwende Vokabular des erstellten Lexers */
}

expression:
  multiplicationExpression (ADDITION multiplicationExpression)*;

multiplicationExpression:
  atom (MULTIPLICATION atom)* ;

atom:
  INTEGER |
  PARAMETER |
  OPENBRACE expression CLOSEBRACE ;
```

DEMO - PARSER

- Drei Regeln:
  - atom: Ein Atom ist eine Integer-Konstante, ein Parameter oder ein geklammerter Ausdruck
  - expression: Verkettung von Additionen
  - multiplicationExpression: Verkettung von Multiplikation

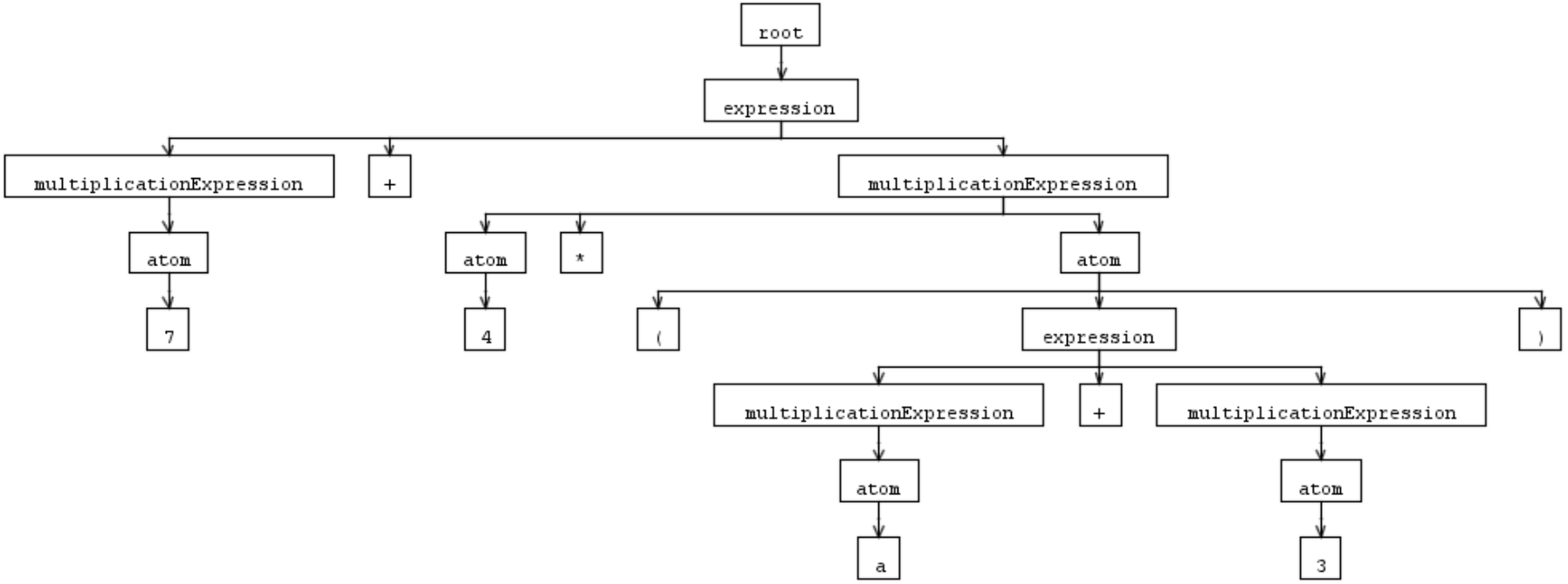
- Abhängigkeiten der Regeln untereinander spiegeln Operatorpräzedenz wieder

DEMO - PARSER

- Notation der Regeln in der Extended Backus-Naur Form (EBNF)
  - nicht schön, aber kompakt und erlernbar

DEMO - PARSER

- "7+4*(a+3)" bisher nur geparst
- Kein Resultat (bis auf ParseTree / Validität)



DEMO - PARSER

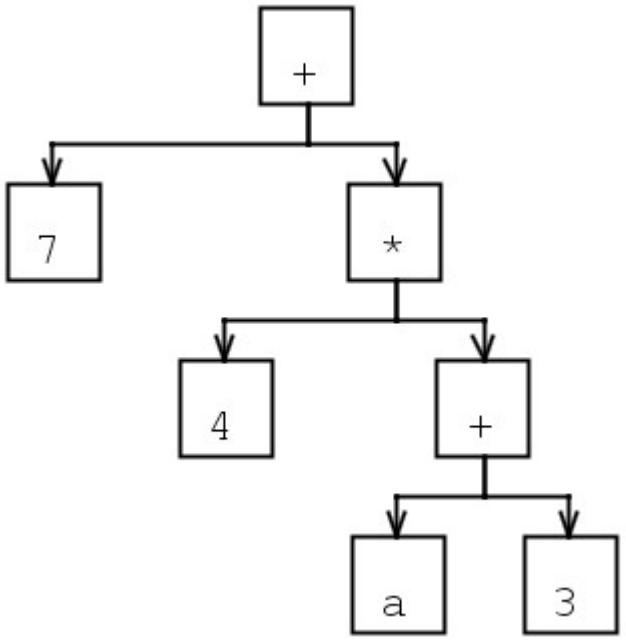
- 7+4*(X+3) per Hand verarbeiten
  - formal nicht korrekt, stark vereinfacht
  - Ausgehend von der Startregel (e) Regeln anwenden, bis Ausdruck (Tokenfolge) gebildet ist
  - e=>m+m=>a+m=>7+m=>7+a*a=> 7+4*a=>7+4*(e)=>7+4*(m+m) => 7+4*(a+m)=>7+4*(X+m) => 7+4*(X+a) =>7+4*(X+3)

DEMO - PARSER

```
parser grammar ExpressionParser;

options {
language=CSharp2;
tokenVocab=ExpressionLexer; /* verwende Vokabular des erstellten Lexers */
output=AST;
}

expression:
  multiplicationExpression (ADDITION^ multiplicationExpression)*;

multiplicationExpression:
  atom (MULTIPLICATION^ atom)* ;

atom: INTEGER |
  PARAMETER |
  OPENBRACE! expression CLOSEBRACE! ;
```

DEMO - PARSER

- Abstract Syntax Tree (AST) als Ergebnis



DEMO - PARSER

- Ergebnis: AST wurde rein deklarativ erzeugt
  - Unabhängig von der Ziel-Programmiersprache, also portabel, noch kein eigener (C#-)Code

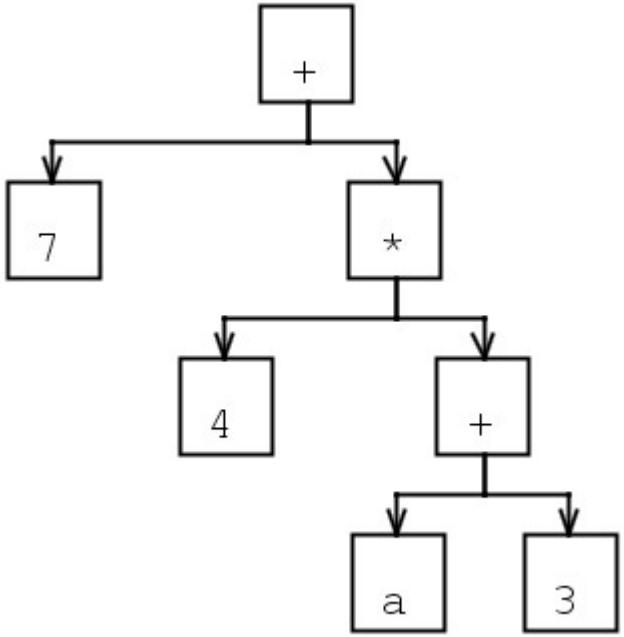
- Zwei Syntaxvarianten bei AST-Erzeugung
  - Einfache Syntax: ^ und ! direkt in der Regel
  - Mächtige Syntax: -> ("rewrite")
    - Reihenfolge ändern, bedingte Transformation, künstliche Tokens einfügen, ...

DEMO

- Lexer
- Parser
  - Stufe 1: Nur Erkennen
  - Stufe 2: Abstrakten Syntaxbaum (AST) erzeugen
- Tree Walker
  - AST verarbeiten
    - Pass 1: Parameternamen ermitteln
    - Pass 2: Dynamische Methode für arithmetischen Ausdruck generieren

TREE WALKER

- Brauche ich ANTLR für die Verarbeitung des AST? Oder implementiere ich ein Visitor-Pattern selbst?



TREE WALKER

- Was bringt mir der Einsatz von ANTLR?
  - AST wird nach Regeln aufgespannt
  - ANTLR kann gut mit Regeln umgehen
  - Ansatz: Überlasse es ANTLR, den AST anhand der Regeln zu durchwandern und regelgesteuert den eigenen Code auszuführen!
  - Eigenen Code sparen - deklarativ arbeiten

TREE WALKER

```
tree grammar JustASkeleton;

options {
        tokenVocab=ExpressionLexer;
        language=CSharp2;
        ASTLabelType=CommonTree;
}

expr:
   ^(ADDITION expr expr) { your code here }
   | ^(MULTIPLICATION expr expr) { your code here }
   | INTEGER { your code here }
   | PARAMETER { your code here }
   ;
```

TREE WALKER

- Pass 1: Parameter im AST bestimmen

- Regel wird erweitert um in-Parameter
  - Liste der Parameter im Ausdruck
- Regel wird erweitert um eigenen Code
  - Trifft ANTLR beim Parsen des AST auf die Variante der Strukturregel, die einen Parameter identifiziert, wird dieser in die Liste eingefügt

TREE WALKER

```
tree grammar FindParameters;

options {
        tokenVocab=ExpressionLexer;
        language=CSharp2;
        ASTLabelType=CommonTree;
}

expr [List<string> parameters]:
   ^(ADDITION expr[parameters] expr[parameters])
   | ^(MULTIPLICATION expr[parameters] expr[parameters])
   | INTEGER
   | PARAMETER { parameters.Add($PARAMETER.text); }
   ;
```

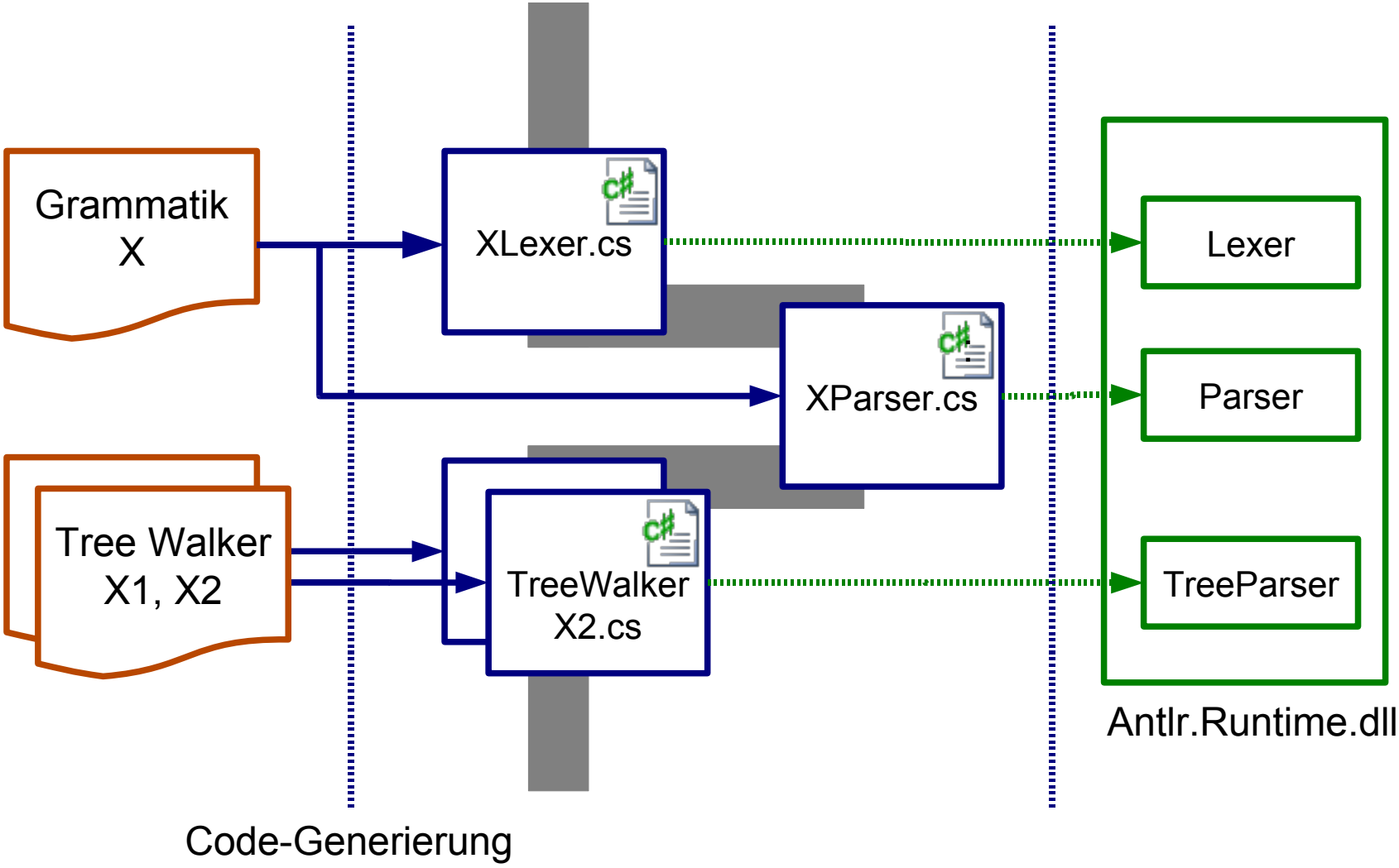
TREE WALKER

- Pass 2: Aus arithmetischem Ausdruck System.Linq.Expression erzeugen
  - Diese kann dann kompiliert und verwendet werden ;-) (analog zur DLR)
- Wie Pass1, aber
  - Regel wird mit out-Parameter annotiert,
    Regel kann als Funktion aufgefasst werden
  - out-Parameter vom Typ System.Linq.Expression
  - Expression-Baum wird mit Abarbeitung der Regeln zusammengestellt

TREE WALKER

```
tree grammar BuildFunction;

options {
language=CSharp2; ...
}

expr[IDictionary<string, ParameterExpression> parameters]
  returns [Expression result]:
  ^(ADDITION left=expr[parameters] right=expr[parameters])
  { $result = Expression.Add(left, right); }
  | ^(MULTIPLICATION left=expr[parameters] right=expr[parameters])
  { $result = Expression.Multiply(left, right); }
  | INTEGER
  { $result = Expression.Constant(Int32.Parse($INTEGER.text)); }
  | PARAMETER
  { $result = parameters[$PARAMETER.text]; };
```

DEMO-RÜCKBLICK



DEMO-RÜCKBLICK

- Vom textuellen Input zum Verarbeitungsergebnis durch drei sauber entkoppelte Phasen
  - Lexer, Parser, Tree Walker
- Solange wie möglich deklarativ bleiben
  - Eigenen Code erst in AST-Verarbeitungsphase (Tree Walker) einbringen
  - Auch Tree Walker hat deklarativen Teil

Zurück in die "echte" Welt

- Was ist mit ANTLR-Grammatiken wirklich erfassbar?
- Wie kann ich ANTLR verwenden?
  - ANTLR löst uralte Probleme
    - Compilerbau, technisch motiviert
  - Aber ANTLR ist nicht veraltet
    - Untertitel ANTLR-Buch: "Building Domain Specific Languages"
    - DSL ist ein sehr aktuelles Thema!

ANTLR - Grammatiken

- Programmiersprachen
  - C#
  - Java 1.6
  - ECMA Script
  - C++
  - Python
  - Fortran
  - Objective, Gnu, Ansi C
  - Eiffel
  - IL
- W3C/Web et al
  - CSS 2.1, 3.0
  - XPath, XQuery
  - HTML
  - VRML
  - OCL (Teil von UML)
- Datenbanken
  - SQL (verschiedene)
- Aber auch ...
  - CIM, VHDL, ...

ANTLR - Showcases

- Hibernate (HQL)
- Genome ORM
- IntelliJ
- Mac OS X Project Builder
- BEA Web Logic (JSP)
- Sybase PowerBuilder
- Adobe Flex Builder 3
- Spring.NET
- ...

ANTLR und DSL (1)

- Technischer Einsatz in der Domäne Softwareentwicklung
  - Eigene Programmiersprachen, Compiler, Interpreter, ...
  - Konfiguration
    - Beispiele : IoC-Container, Markup, Reports, Formatierung, ...
  - Beschränkt auf Einsatz in der internen Toolchain der Softwareentwicklung?

ANTLR und DSL (2)

- Weitere Einsatzmöglichkeiten
  - Platinenlayout, UML, Robotik -> geht über Softwareentwicklung hinaus
- Software wird für Problemdomänen der Kunden entwickelt
  - Entwicklern und Kunden sollten eine gemeinsame, domänenspezifische Sprache finden:
    die sogenannte "Ubiquitous Language"
  - Ermöglicht prägnante, kompakte, unmissverständliche Formulierungen

ANTLR und DSL (3)

- Warum sollte eine Konfigurationsdatei nicht in einer spezifischen "Ubiquitous Language" formuliert sein?
  - ... und warum hier nicht ANTLR verwenden?
  - Anpassung durch den Kunden

Fazit

- Positiv
  - Sehr mächtig, hoher Reifegrad
  - Bis zu einer bestimmten Komplexität schnell erlernbar
  - Gut dokumentiert (hervorragendes Buch)
  - Weitgehende Anpassungsmöglichkeiten
  - Multi-Plattform
- Negativ
  - Benötigt eigene Runtime (BSD-Lizenz)

Fragen?

Weitere Themenfolien

Links

- antlr.org mit umfangreichem Wiki
  antlr.org, Siehe dort: Quick Start
- Buch: "The definitive ANTLR Reference"
  http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference
  ANTLR und DLR
  http://www.bitwisemag.com/2/DLR-Build-Your-Own-Language
  http://www.sapphiresteel.com/Antlr-3-and-the-Dynamic-Language
- fodey.com Newspaper Generator
  http://www.fodey.com/generators/newspaper/snippet.asp

Weitere Features

- Prädikate
- Subrules
- Templates
- Hohe Anpassbarkeit und Eingriffsmöglichkeiten
  - Eigene Typen für Tokens, Knoten im AST
  - Fehlerbehandlung
  - …
- Bäume deklarativ transformieren
- Backtracking

ANTLR-Zielplattformen

- Java
- C++
- C# 1.1, 2.0
- Objective C
- Python
- Ruby
- Perl6
- Perl
- JavaScript
- ActionScript
- Ada95

In Arbeit

- C# 3.0
- Emacs ELisp
- PhP
- Oberon
- Dephi

IDE-Support

- ANTLRWorks als eigene IDE (Java only)
  - Praktisch zum Debuggen und Lernen,
    bis der erste C#-Code ins Spiel kommt
- Andere IDEs werden weniger gut unterstützt
  - Sehr rudimentäres Visual Studio Add-In
    (Syntax-Highlighting), kein MSBuild-Task
  - Eclipse PlugIn

Andere Produkte im .NET-Bereich

- OSLO (CTP)
- GOLD Parser
- Grammatica
- Managed Package Parser Generator, Managed Babel System, Visual Studio SDK
  - Ausgelegt für die Integration neuer Sprachen (Intellisense etc) GPPG