Compiler 1: Grundlagen

Lexer/Parser-Generierung mit ANTLR

TECHNISCHE UNIVERSITÄT DARMSTADT

WS 2014/15

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR - Another Tool for Language Recognition
- Kurze Einführung in ANTLR 4.x
  - Inkompatibel zu ANTLR 2.x und 3.x!

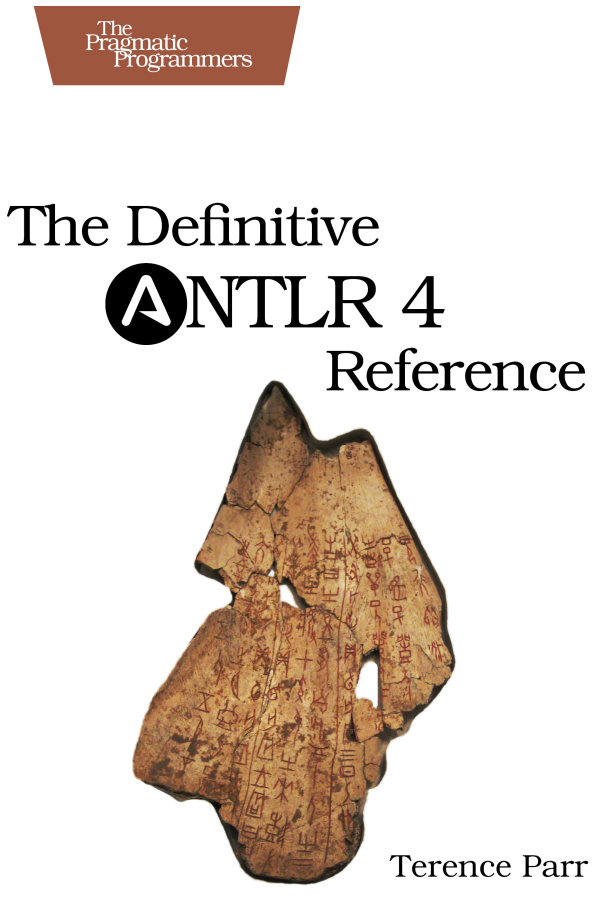
Werbung

TECHNISCHE UNIVERSITÄT DARMSTADT

Inhalt und Beispiele stammen aus:

The Definitive ANTLR 4 Reference

- Terence Parr
- Pragmatic Bookshelf 2013
- Sehr gut lesbar!



ANTLR - Einführung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

ANTLR 4.x

- Adaptive LL(*) Compiler Generator
  - Eingabe: Grammatik in EBNF (und mehr!)
  - Ausgabe: Erkenner für Sprache mittels rekursivem Abstieg
- Erzeugt (akt. Version 4.5) Erkenner in ...
  - Java, C#, C++, JavaScript, Python
  - weitere Sprachen geplant.

ANTLR - Einführung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

ANTLR 4.x

- Adaptive LL(*) Compiler Generator
  - Eingabe: Grammatik in EBNF (und mehr!)
  - Ausgabe: Erkenner für Sprache mittels rekursivem Abstieg
- Erzeugt (akt. Version 4.5) Erkenner in ...
  - Java, C#, C++, JavaScript, Python
  - weitere Sprachen geplant.

Arten von Eingabedaten

- Zeichenströme (bearbeitet mit Lexer)
- Token-Ströme (bearbeitet mit Parser)

ANTLR - Einführung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

ANTLR 4.x

- Adaptive LL(*) Compiler Generator
  - Eingabe: Grammatik in EBNF (und mehr!)
  - Ausgabe: Erkenner für Sprache mittels rekursivem Abstieg
- Erzeugt (akt. Version 4.5) Erkenner in ...
  - Java, C#, C++, JavaScript, Python
  - weitere Sprachen geplant.

Arten von Eingabedaten

- Zeichenströme (bearbeitet mit Lexer)
- Token-Ströme (bearbeitet mit Parser)

Alternative Compiler-Generatoren

- Lexer/Scanner: lex, flex, JFlex
- Parser: yacc/bison, JCup, JavaCC, SableCC, SLADE

Neuerungen in ANTLR 4.x

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR verarbeitet jede gültige Grammatik
  - Versucht Konflikte und Mehrdeutigkeiten automatisch aufzulösen
    - Anhand von Heuristiken für die praktisch relevantesten Stellen
  - Einschränkung: Indirekt links-rekursive Regeln werden nicht unterstützt

Neuerungen in ANTLR 4.x

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR verarbeitet jede gültige Grammatik
  - Versucht Konflikte und Mehrdeutigkeiten automatisch aufzulösen
    - Anhand von Heuristiken für die praktisch relevantesten Stellen
  - Einschränkung: Indirekt links-rekursive Regeln werden nicht unterstützt
- Adaptiver Parser mit variablem Lookahead: ALL($\star$)
  - Die Grammatik wird zur Laufzeit des generierten Parsers analysiert
  - Parser kann Entscheidungen auf konkreten Eingabedaten treffen

Neuerungen in ANTLR 4.x

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR verarbeitet jede gültige Grammatik
  - Versucht Konflikte und Mehrdeutigkeiten automatisch aufzulösen
    - Anhand von Heuristiken für die praktisch relevantesten Stellen
  - Einschränkung: Indirekt links-rekursive Regeln werden nicht unterstützt
- Adaptiver Parser mit variablem Lookahead: ALL($\star$)
  - Die Grammatik wird zur Laufzeit des generierten Parsers analysiert
  - Parser kann Entscheidungen auf konkreten Eingabedaten treffen
- Automatischer Aufbau von Parse Trees
- Automatische Generierung von Visitors und Listeners

Neuerungen in ANTLR 4.x

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR verarbeitet jede gültige Grammatik
  - Versucht Konflikte und Mehrdeutigkeiten automatisch aufzulösen
    - Anhand von Heuristiken für die praktisch relevantesten Stellen
  - Einschränkung: Indirekt links-rekursive Regeln werden nicht unterstützt
- Adaptiver Parser mit variablem Lookahead: ALL($\star$)
  - Die Grammatik wird zur Laufzeit des generierten Parsers analysiert
  - Parser kann Entscheidungen auf konkreten Eingabedaten treffen
- Automatischer Aufbau von Parse Trees
- Automatische Generierung von Visitors und Listeners
- Bevorzuge saubere Trennung zwischen Grammatik und Implementierungssprache
  - Wesentliche Änderung des APIs im Vergleich zu ANTLR v3

Weiterführendes Material

TECHNISCHE UNIVERSITÄT DARMSTADT

Internet

- **`http://www.antlr.org`**
- Dort: Wiki, Thema “FAQ und Getting Started”
- Sehr umfangreiche Materialsammlung
  - Leider unstrukturiert
- Vortrag von Terence Parr: http://www.youtube.com/watch?v=q8p1voEiu8Q

Hello World

TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar Hello;
r  : 'hello' ID ;
ID : [a-z]+ ;
WS : [ \r\t\n]+ -> skip ;
```

Definiere Grammatik "Hello" (der Dateiname muss Hello.g4 sein)

Erkenne Schlüsselwort *hello* gefolgt von einem Bezeichner

Erkenne Bezeichner in Kleinbuchstaben

Erkenne Leerzeichen und ignoriere sie

Setup

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR herunterladen

```
$ cd ~/xyz
$ wget http://antlr4.org/download/antlr-4.5-complete.jar
```

- Umgebungsvariable `CLASSPATH` (für die Java-Werkzeuge) setzen

```
$ export CLASSPATH=".:~/xyz/antlr-4.5-complete.jar:$CLASSPATH"
```

- Aliase für ANTLR und das TestRig erstellen (unixoide Systeme)

```
$ alias antlr4='java -jar ~/xyz/antlr-4.5-complete.jar'
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

TestRig

Generischer Testtreiber für die Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR aufrufen und generierte Dateien übersetzen

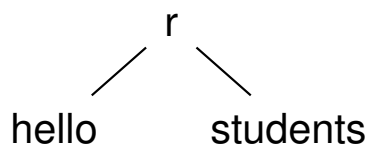
```
$ antlr4 Hello.g4
$ javac *.java
```

- Eingabe in Token zerlegen

```
$ echo "hello students" | grun Hello r -tokens
[@0,0:4='hello',<1>,1:0]
[@1,6:13='students',<2>,1:6]
[@2,15:14='<EOF>',<-1>,2:0]
```

- Eingabe parsen

```
$ echo "hello students" | grun Hello r -gui
```



Struktur einer Grammatik-Datei

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/* comments */ // more comments
```
← Kommentare (Java-Style)

```
[lexer|parser] grammar <Name>;
```
← Spezifikation Grammatikart und -name

```
options { ... }
```
← Optionen, z.B. Zielsprache

```
import ...
```
← Import anderer Grammatiken

```
tokens { ... }
```
← (optionale) Tokenspezifikation

```
@header { ... }
@members { ... }
```
← Aktionen: Fügen Code in den Parser ein

```
rule1: alternative1
     | alternative2
     ;
```
← Parser-Regeln

```
RULE2: ...
```
← Lexer-Regeln (Tokens)

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Eine Regeln besteht aus einem Namen
  - gefolgt von einer Menge an Alternativen.

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Eine Regeln besteht aus einem Namen
  - gefolgt von einer Menge an Alternativen.
- Parserregeln beginnen immer mit einem Kleinbuchstaben
- Lexerrregeln (Tokens) beginnen immer mit einem Großbuchstaben

Aufbau einer Regel


```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Eine Regeln besteht aus einem Namen
  - gefolgt von einer Menge an Alternativen.
- Parserregeln beginnen immer mit einem Kleinbuchstaben
- Lexerrregeln (Tokens) beginnen immer mit einem Großbuchstaben
- Eine Alternative referenziert
  - andere Parserregeln
  - Lexerregeln
  - Literale in einfachen Anführungszeichen

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Eine Regeln besteht aus einem Namen
  - gefolgt von einer Menge an Alternativen.
- Parserregeln beginnen immer mit einem Kleinbuchstaben
- Lexerrregeln (Tokens) beginnen immer mit einem Großbuchstaben
- Eine Alternative referenziert
  - andere Parserregeln
  - Lexerregeln
  - Literale in einfachen Anführungszeichen
- Alternativen können mit einem Label versehen werden.

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Eine Regeln besteht aus einem Namen
  - gefolgt von einer Menge an Alternativen.
- Parserregeln beginnen immer mit einem Kleinbuchstaben
- Lexerrregeln (Tokens) beginnen immer mit einem Großbuchstaben
- Eine Alternative referenziert
  - andere Parserregeln
  - Lexerregeln
  - Literale in einfachen Anführungszeichen
- Alternativen können mit einem Label versehen werden.
- Außerdem: Parameter, Rückgabewerte, eingebettete Aktionen, Optionen, ...

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Hintereinanderschreiben von Grammatiksymbolen entspricht der Konkatenation.

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Hintereinanderschreiben von Grammatiksymbolen entspricht der Konkatenation.
- Unterregeln (subrules) werden durch Klammern eingeleitet
  - Sie können weitere Alternativen getrennt durch | beeinhalten.

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Hintereinanderschreiben von Grammatiksymbolen entspricht der Konkatenation.
- Unterregeln (subrules) werden durch Klammern eingeleitet
  - Sie können weitere Alternativen getrennt durch | beeinhalten.
- Weitere Operatoren sind * (0-n Wdh.), + (1-n Wdh.), ? (0-1 Wdh.).

Aufbau einer Regel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rulename : alternative1                                  # label
         | anotherParserRule LEXERRULE 'literal'         # concat
         | A 'x' (B | C)                                 # subrule
         | D* E+ F? (G H | I)+                           # repeat
         ;
```

- Hintereinanderschreiben von Grammatiksymbolen entspricht der Konkatenation.
- Unterregeln (subrules) werden durch Klammern eingeleitet
  - Sie können weitere Alternativen getrennt durch | beeinhalten.
- Weitere Operatoren sind * (0-n Wdh.), + (1-n Wdh.), ? (0-1 Wdh.).
- Die Operatoren lassen sich auch auf Unterregeln anwenden.

Kurzschreibweisen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
CharClass1 : [a-z] ;
```
← Zeichenklasse: Alle Zeichen zwischen a und z.

```
CharClass2 : 'a'..'z' ;
```
← Alternative Schreibweise für obige Zeichenklasse

```
Negation : ~[abc] ;
```
← Negation: Alle Zeichen außer a, b oder c

```
NonGreedy : '//' .*? '\n' ;
```
← Der *?-Operator ist die “nicht-gierige” (non-greedy) Variante des Kleene-Sterns. Damit werden alle Zeichen (.) bis zum **ersten** Auftreten des folgenden Zeichens (’\n’) erkannt.

Abschnitt 2

Parser und Listener-Interface

ArrayInit

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grammatik, die (geschachtelte) Array-Initialisierer erkennt
  - z.B. `{1,2,3}` oder `{17, {42, 55, 31}, 28, 0}`

```
grammar ArrayInit;

init  :
  '{' value (',' value)* '}' ;

value : init
      | INT
      ;

INT   : [0-9]+ ;
WS    : [ \t\r\n]+ -> skip ;
```

Erkenne komma-separierte Werte zwischen geschweiften Klammern (mindestens einen)

Ein 'value' ist entweder wieder eine Liste, ...

oder eine Zahl.

Erzeugte Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

`antlr4 ArrayInit.g4` erzeugt:

- ArrayInitParser.java
  - `public class ArrayInitParser extends Parser { ... }`
- ArrayInitLexer.java
  - `public class ArrayInitLexer extends Lexer { ... }`
  - Erzeugt Tokenstrom für den Parser.
- ArrayInit.tokens
  ArrayInitLexer.tokens
  - `*.tokens`-Dateien nur für Datenaustausch zwischen Grammatiken
    - (hier nicht notwendig)
- ArrayInitListener.java
  ArrayInitBaseListener.java
  - Listener-Interface (Alternative zum Visitor-Pattern, später mehr dazu)

Parser

TECHNISCHE UNIVERSITÄT DARMSTADT

Parser erzeugt einen abstrakten Parse-Baum, bestehend aus

- **`ParserRuleContext`**-Objekten
  - “Kontext”: hier Speicherung aller Information über erkannte Phrase
  - Eigene Subklasse für jede Regel (bzw. für Alternative mit Label)
  - Attribute für Nichtterminals und benannte Literaltokens
    - Nicht-benannte Literale werden verworfen
  - Entspricht den phrasenspezifischen AST-Knoten in Triangle
  - Enthält u.a. Zugriffsmethoden auf
    - Start- und Endtokens
    - Kinder und Elter im AST
- **`TerminalNode`**-Objekten (repräsentieren die Terminals)

Parser

AST für `{1,{2,3},4}`

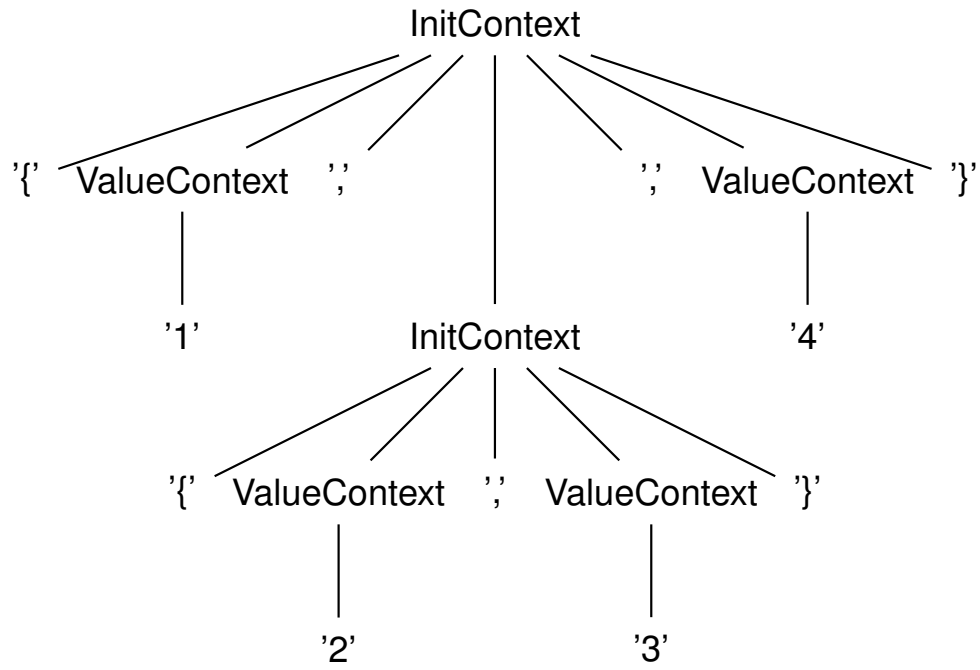
TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar ArrayInit;

init  :
  '{' value (',' value)* '}' ;

value : init
      | INT
      ;

INT   : [0-9]+ ;
WS    : [ \t\r\n]+ -> skip ;
```



Parser

Generierter Code

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ArrayInitParser extends Parser {
  ...
  public static class InitContext extends ParserRuleContext {
    ...
    public List<ValueContext> value() { ... }
    ...
  }
  public final InitContext init() throws RecognitionException { ... }

  public static class ValueContext extends ParserRuleContext {
    ...
    public TerminalNode INT() { ... }
    public InitContext init() { ... }
    ...
  }
  public final ValueContext value() throws RecognitionException { ... }
  ...
}
```

← Zugriff auf Kinder im AST

Parser
Integration

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
public class Test {
  public static void main(String[] args) throws Exception {
    ANTLRInputStream input = new ANTLRInputStream(System.in);

    ArrayInitLexer lexer = new ArrayInitLexer(input);

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    ArrayInitParser parser = new ArrayInitParser(tokens);

    ParseTree tree = parser.init();
    System.out.println(tree.toStringTree(parser));
  }
}
```

init ist der Name der Regel

→ Lexer und Parser mit Tokenstream verbinden
Methode für Startsymbol aufrufen

Verwendung des Parsers

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bislang wird nur erkannt, ob die Eingabe zur Grammatik passt.
- In ANTLR $\leq$ 3: Semantische Aktionen (= Java-Code) in Grammatik einbetten.
  - Aktionen werden beim Erkennen einer Regel ausgeführt.
  - In v4 weiterhin möglich, es wird aber davon abgeraten
    - Da nun Konstrukte der Zielsprache und Grammatik vermischt

Verwendung des Parsers

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bislang wird nur erkannt, ob die Eingabe zur Grammatik passt.
- In ANTLR $\leq$ 3: Semantische Aktionen (= Java-Code) in Grammatik einbetten.
  - Aktionen werden beim Erkennen einer Regel ausgeführt.
  - In v4 weiterhin möglich, es wird aber davon abgeraten
    - Da nun Konstrukte der Zielsprache und Grammatik vermischt

- Traversieren des ASTs
  - Neu und viel bequemer in ANTLRv4
- Automatische Generierung von passenden Interfaces und Implementierungen
  - Listener (traversiert auch Unterbäume automatisch)
  - Visitor (explizite Traversierung von Unterbäumen erforderlich)

Listener

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Listener reagiert auf Ereignisse (*events*).
  Hier:
  - Der AST wird mittels Tiefensuche von links nach rechts traversiert.
  - Wird ein Knoten X besucht, wird die Methode **`enterX(..)`** aufgerufen.
  - Nachdem alle Kinder von X besucht wurden, wird **`exitX(..)`** aufgerufen.

Listener

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Listener reagiert auf Ereignisse (*events*).
  Hier:
  - Der AST wird mittels Tiefensuche von links nach rechts traversiert.
  - Wird ein Knoten X besucht, wird die Methode **`enterX(..)`** aufgerufen.
  - Nachdem alle Kinder von X besucht wurden, wird **`exitX(..)`** aufgerufen.

- Für die Grammatik **`ArrayInit`** ANTLR generiert automatisch
  - das Listener-Interface **`ArrayInitListener`**
  - eine Defaultimplementiertung (mit leeren Eventmethoden)
    `ArrayInitBaseListener`

Listener

Beispiel: `{1,{2,3},4}`

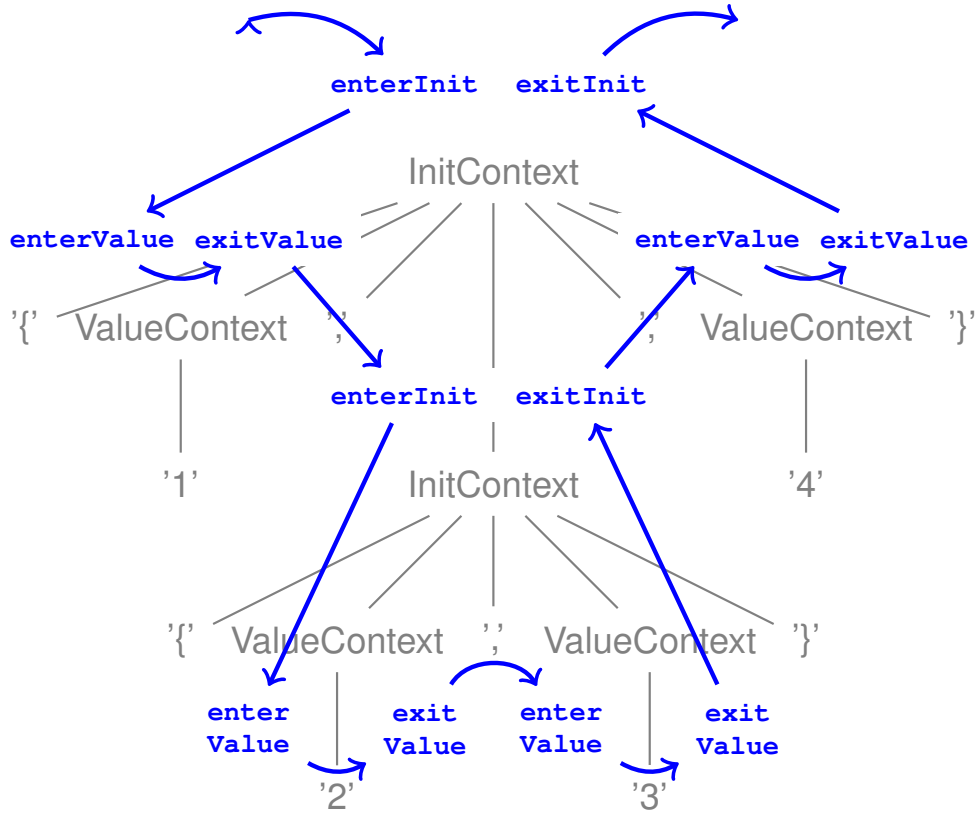
TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar ArrayInit;

init  :
  '{' value (',' value)* '}' ;

value : init
      | INT
      ;

INT   : [0-9]+ ;
WS    : [ \t\r\n]+ -> skip ;
```



Listener

Anwendungsbeispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel: Array-Initialisierer wie `{1, 2, 3}` in Unicode-Strings wie `"\u0001\u0002\u0003"` umwandeln.
  - Ist tatsächlich sinnvoll, letzteres wird effizienter von JVM ausgeführt
- Vorgehensweise:
  - Geschweifte Klammern zu Anführungszeichen machen
  - Elemente in hexadezimale Darstellung umwandeln und jeweils \u davor schreiben

Listener
Code-Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ShortToUnicodeString extends ArrayInitBaseListener {
    public void enterInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }

    public void exitInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }

    public void enterValue(ArrayInitParser.ValueContext ctx) {
        int value = Integer.valueOf(ctx.INT().getText());
        System.out.printf("\\u%04x", value);
    }
}
```

Annahme:
keine geschachtelten Initialisierer

Listener

Anwendungsbeispiel (Integration)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Test {
  public static void main(String[] args) throws Exception {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    ArrayInitLexer lexer = new ArrayInitLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ArrayInitParser parser = new ArrayInitParser(tokens);

    ParseTree tree = parser.init();
    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(new ShortToUnicodeString(), tree);

    System.out.println();
  }
}
```

Abschnitt 3

Expressions und Visitor-Interface

Expression-Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar Expr;
import ExprLexerRules;

prog:    stat+ ;

stat:    expr NEWLINE                # printExpr
     |   ID '=' expr NEWLINE         # assign
     |   NEWLINE                     # blank

expr:    expr op=('*'|'/') expr      # MulDiv
     |   expr op=('+'|'-') expr      # AddSub
     |   INT                         # int
     |   ID                          # id
     |   '(' expr ')'                # parens
     ;
```

Andere Grammatik importieren (hier: die Lexer-Regeln)

Benennung der Unterregel → Name des Kinds im AST

Benennung (Labeling) der Alternativen → getrennte Event/ Visitor-Methoden

Links-rekursive Regel (!) → Rekursion wird von ANTLR intern eliminiert

Expression-Grammatik

Importierte Lexer-Regeln

TECHNISCHE UNIVERSITÄT DARMSTADT

Deklaration, dass diese Grammatik nur Lexer-Regeln enthält

```
lexer grammar ExprLexerRules;

MUL :   '*' ;
DIV :   '/' ;
ADD :   '+' ;
SUB :   '-' ;

ID  :     [a-zA-Z]+ ;
INT :     [0-9]+ ;
NEWLINE: '\r'? '\n' ;
WS  :     [ \t]+ -> skip ;
```

Benennung der Tokens

Visitor-Pattern

Anwendungsbeispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel: Interaktiver Taschenrechner
- Benutzung des von ANTLR generierten Visitor-Patterns
- Unterschied zum Listener-Mechanismus
  - Traversierung von Unterbäumen muß nun explizit ausformuliert werden
  - Damit Eingriff in Traversierung möglich, z.B.
    - Veränderte Reihenfolge
    - Überspringen von Unterbäumen
  - ANTLR Visitor-Methoden können Ergebnis an Aufrufer zurückgeben
    - Haben aber keine Argumente außer besuchtem AST-Knoten

Visitor-Pattern
Anwendungsbeispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel: Interaktiver Taschenrechner
- Benutzung des von ANTLR generierten Visitor-Patterns
- Unterschied zum Listener-Mechanismus
  - Traversierung von Unterbäumen muß nun explizit ausformuliert werden
  - Damit Eingriff in Traversierung möglich, z.B.
    - Veränderte Reihenfolge
    - Überspringen von Unterbäumen
  - ANTLR Visitor-Methoden können Ergebnis an Aufrufer zurückgeben
    - Haben aber keine Argumente außer besuchtem AST-Knoten

- Erzeugen von Visitor statt Listener:
  - `antlr4 -no-listener -visitor Expr.g4`
- Erzeugt
  - Interface **`ExprVisitor`**
  - Leere Default-Implementierung **`ExprBaseVisitor`**
    - Besucht alle Unterbäume, führt aber sonst keine Operationen aus

Visitor-Pattern

Implementierung des Beispiels I

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EvalVisitor extends ExprBaseVisitor<Integer> {
    Map<String, Integer> memory = new HashMap<String, Integer>();

    public Integer visitAssign(ExprParser.AssignContext ctx) {
        String id = ctx.ID().getText();
        int value = visit(ctx.expr());
        memory.put(id, value);
        return value;
    }

    public Integer visitPrintExpr(ExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr());
        System.out.println(value);
        return 0;
    }

    public Integer visitInt(ExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }
```

Visitor-Pattern

Implementierung des Beispiels II

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public Integer visitId(ExprParser.IdContext ctx) {
    String id = ctx.ID().getText();
    if ( memory.containsKey(id) ) return memory.get(id);
    return 0;
}

public Integer visitAddSub(ExprParser.AddSubContext ctx) {
    int left = visit(ctx.expr(0));
    int right = visit(ctx.expr(1));
    if ( ctx.op.getType() == ExprParser.ADD )
        return left + right;
    return left - right;
}

public Integer visitMulDiv(ExprParser.MulDivContext ctx) { ... }

public Integer visitParens(ExprParser.ParensContext ctx) {
    return visit(ctx.expr());
} } // EvalVisitor
```

Möglich durch Benennung der Token und Unterregeln

Visitor-Pattern

Integration

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTree;

public class Calc {
    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);

        ParseTree tree = parser.prog();

        EvalVisitor eval = new EvalVisitor();
        eval.visit(tree);
    }
}
```

Fortgeschrittene Themen

TECHNISCHE UNIVERSITÄT DARMSTADT

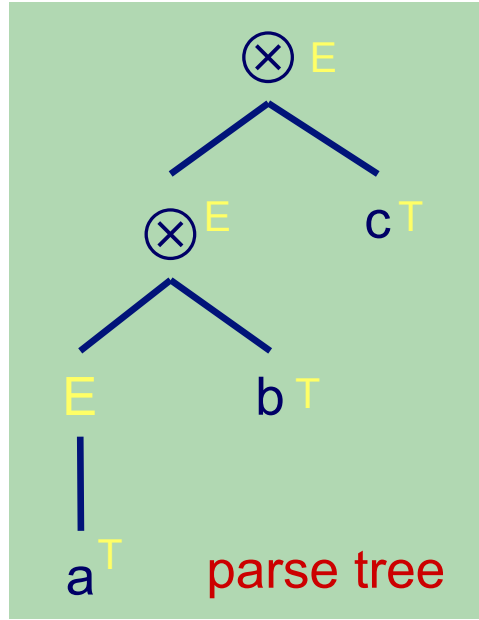
- Links- und Rechtsassoziativität
- Operatorpräzedenzen
- Das “Dangling Else”-Problem
- Semantische Prädikate
- Fehlerbehandlung

Linksassoziativität

ANTLR v3: Formulierung der Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Linksassoziativer Operator $\otimes$:
  $a \otimes b \otimes c = (a \otimes b) \otimes c$
- Produktion (linksrekursiv!)
  $E ::= E \otimes T \,|\, T$
- In EBNF
  $E ::= T(\otimes T)^*$

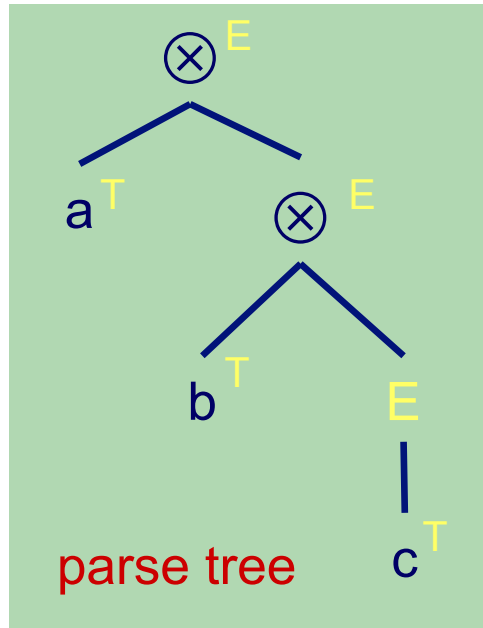


Rechtsassoziativität

ANTLR v3: Formulierung in der Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Rechtsassoziativer Operator $\otimes$:
  $a \otimes b \otimes c = a \otimes (b \otimes c)$
- Produktion (linksrekursiv!)
  $E ::= T \otimes E \mid T$
- In EBNF (? = 0- oder 1-mal)
  $E ::= T(\otimes E)?$

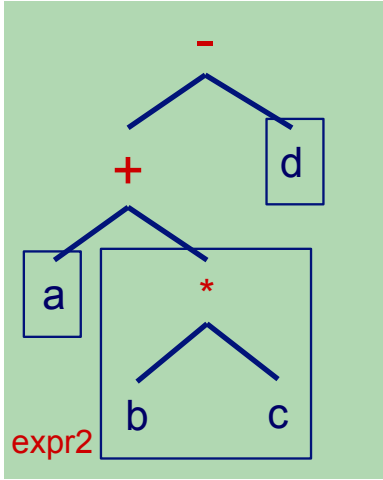


Operatorpräzedenz 1

ANTLR v3: Formulierung in der Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiel:
  $a + b \times c - d$
- . . . sollte geparsed werden als
  $(a + (b \times c)) - d$



- Operator $\times$ hat höhere Präzedenz als $+$ und $-$
- In Grammatik ausdrücken, durch Platzieren von $\times$ “näher an Operanden” als $+$ und $-$

```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```

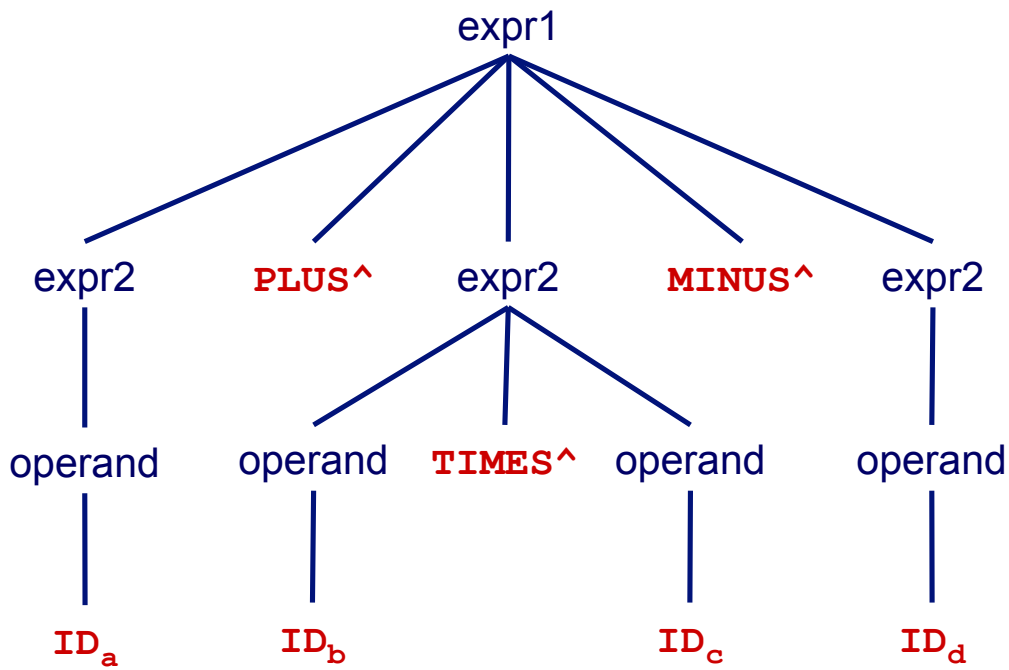
Operatorpräzedenz 2

$a + b \times c - d$

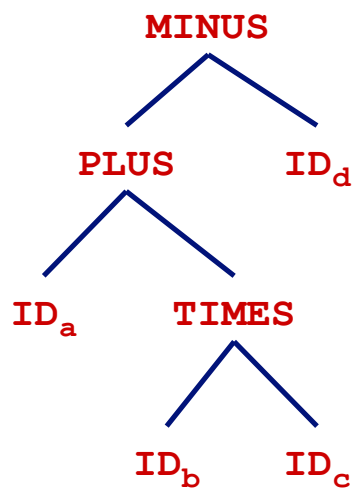
TECHNISCHE UNIVERSITÄT DARMSTADT

```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```

parse tree:



constructed AST:



Assoziativität und Präzedenz in ANTLRv4

TECHNISCHE UNIVERSITÄT DARMSTADT

```
expr:   expr '^'<assoc=right> expr
    |   expr op=('*'|'/') expr
    |   expr op=('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
```

- Operatorpräzedenz wird durch Reihenfolge der Alternativen implizit kodiert
- Rechts-assoziative Operatoren durch Token-Option `assoc=right` kennzeichnen

Assoziativität und Präzedenz in ANTLRv4

TECHNISCHE UNIVERSITÄT DARMSTADT

```
expr:   expr '^'<assoc=right> expr
    |   expr op=('*'|'/') expr
    |   expr op=('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
```

- Operatorpräzedenz wird durch Reihenfolge der Alternativen implizit kodiert
- Rechts-assoziative Operatoren durch Token-Option `assoc=right` kennzeichnen

- ANTLR kann links-rekursive Regeln dieser Art automatisch transformieren.

→ Verfahren: “Precedence climbing”, mehr Details

  - ANTLR v4 Buch
  - **`http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm`**

Hängendes `else`

dangling else

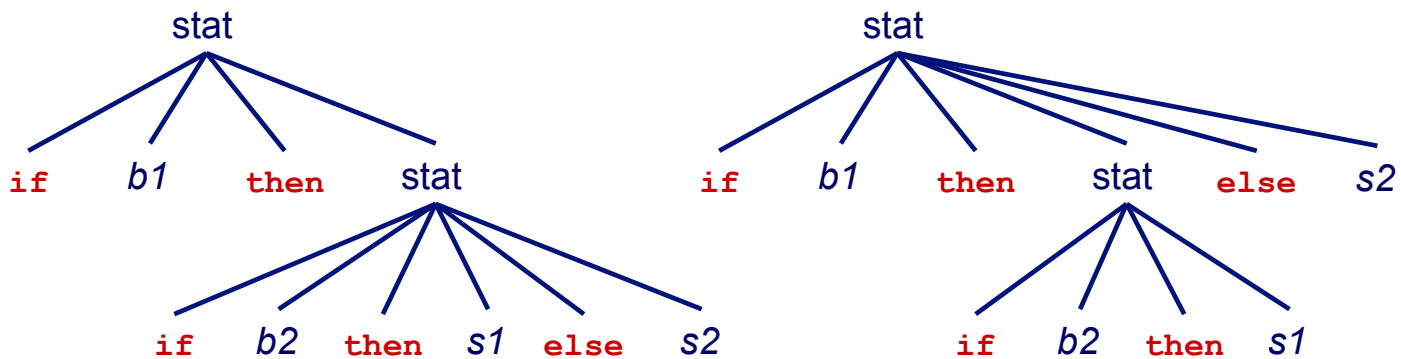
TECHNISCHE UNIVERSITÄT DARMSTADT

Klassisches Problem von Mehrdeutigkeit beim Parsen

```
stat : 'if' expr 'then' stat ('else' stat)?
     | ...      ;
```

```
if b1 then if b2 then s1 else s2
```

Zwei mögliche Parse-Bäume. ANTLR tut automatisch das richtige (1. Baum), da ? greedy ist.



Semantische Prädikate 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Laufzeit ausgewertete Prädikate qualifizieren Alternativen
  - Formuliert als Konstrukte der Zielsprache
  - **`TRUE`**: Alternative ist möglich
  - **`FALSE`**: Alternative ist nicht möglich

Semantische Prädikate 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Laufzeit ausgewertete Prädikate qualifizieren Alternativen
  - Formuliert als Konstrukte der Zielsprache
  - **`TRUE`**: Alternative ist möglich
  - **`FALSE`**: Alternative ist nicht möglich

- Vorteil: Mächtigerer Parser
  - Ist `f(1)` Funktionsaufruf oder Zugriff auf Array-Element?
  - Leicht entscheidbar, wenn Typ von `f` während Parsen abgerufen werden kann
- Nachteil: Nun keine zielsprachenunabhängige Grammatik mehr

Semantische Prädikate 2

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Erkenne Gruppen in einem Zahlenstrom, in dem die jeweils erste Zahl die Länge der Gruppe angibt.

2 4 5 3 1 7 9 5 0 0 0 17 0 $\rightarrow$ (4, 5); (1, 7, 9); (0, 0, 0, 17, 0)

Semantische Prädikate 2

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Erkenne Gruppen in einem Zahlenstrom, in dem die jeweils erste Zahl die Länge der Gruppe angibt.

2 4 5 3 1 7 9 5 0 0 0 17 0 $\rightarrow$ (4, 5); (1, 7, 9); (0, 0, 0, 17, 0)

Problem:

- Die Gruppengröße ist Teil der (variablen!) Eingabedaten
- Kann nicht statisch in der Grammatik ausformuliert werden

Semantische Prädikate

Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar Data;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )*
     ;

INT :   [0-9]+ ;
WS  :   [ \t\n\r]+ -> skip ;
```

Semantische Prädikate

Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
grammar Data;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )*
     ;

INT :   [0-9]+ ;
WS  :   [ \t\n\r]+ -> skip ;
```

Parametrisierte Regel. Das Argument ist der Integerwert des INT-Tokens, welches vorher erkannt wurde.

Definition der Regel mit Parameter n und lokaler Variable i

Das semantische Prädikat. In der Grammatik deklarierte Variablen werden mit vorangestelltem $ zugegriffen.

Eine “normale” Aktion. Der Code wird in die entsprechende Methode des Parser kopiert.

Semantische Prädikate

Effekt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
file : group+ ;
group: INT sequence[$INT.int] ;
sequence[int n] locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* ;
```

Beispiel: 2 4 5 3 ...

- In der Regel `group` wird 2 als `INT` erkannt und an `sequence` übergeben

Semantische Prädikate

Effekt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
file : group+ ;
group: INT sequence[$INT.int] ;
sequence[int n] locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* ;
```

Beispiel: 2 4 5 3 ...

- In der Regel `group` wird 2 als `INT` erkannt und an `sequence` übergeben
- Das semantische Prädikat `1 <= 2` wird zu `true` ausgewertet, so dass 4 als `INT` erkannt wird. Die folgende Aktion inkrementiert `i`

Semantische Prädikate

Effekt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
file : group+ ;
group: INT sequence[$INT.int] ;
sequence[int n] locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* ;
```

Beispiel: 2 4 5 3 ...

- In der Regel `group` wird 2 als `INT` erkannt und an `sequence` übergeben
- Das semantische Prädikat `1 <= 2` wird zu `true` ausgewertet, so dass 4 als `INT` erkannt wird. Die folgende Aktion inkrementiert `i`
- Analog wird 5 erkannt und `i` zu 3 inkrementiert

Semantische Prädikate

Effekt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
file : group+ ;
group: INT sequence[$INT.int] ;
sequence[int n] locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* ;
```

Beispiel: 2 4 5 3 ...

- In der Regel `group` wird 2 als `INT` erkannt und an `sequence` übergeben
- Das semantische Prädikat `1 <= 2` wird zu `true` ausgewertet, so dass 4 als `INT` erkannt wird. Die folgende Aktion inkrementiert `i`
- Analog wird 5 erkannt und `i` zu 3 inkrementiert
- Nun deaktiviert `i <= 2` $\rightarrow$ `false` die (einzige) Alternative von `sequence`

Semantische Prädikate

Effekt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
file : group+ ;
group: INT sequence[$INT.int] ;
sequence[int n] locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* ;
```

Beispiel: 2 4 5 3 ...

- In der Regel `group` wird 2 als `INT` erkannt und an `sequence` übergeben
- Das semantische Prädikat `1 <= 2` wird zu `true` ausgewertet, so dass 4 als `INT` erkannt wird. Die folgende Aktion inkrementiert `i`
- Analog wird 5 erkannt und `i` zu 3 inkrementiert
- Nun deaktiviert `i <= 2` $\rightarrow$ `false` die (einzige) Alternative von `sequence`
- Der Parser kehrt erst zu `group` und dann zu `file` zurück, und beginnt mit dem Parsen einer neuen Gruppe

Fehlerbehandlung

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR-generierte Erkenner behandeln Fehler durch Java-Exceptions.
  - **`RecognitionException`** ist Basisklasse aller ANTLR-Exceptions.
- Standardverhalten der generierten Parserregeln:
  - Fange alle **`RecognitionException`**s,
  - Gebe Fehlermeldung aus,
  - Setze dann das Parsen fort.

```
try {
  ...
} catch (RecognitionException re) {
  _errHandler.reportError(this, re);
  _errHandler.recover(this, re);
} finally {
  exitRule();
}
```

Fortsetzen nach Fehler

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wichtig, z.B. für Compiler
  - Benutzer möchte nicht jeden Fehler einzeln präsentiert bekommen

(Einfache) Möglichkeiten für den Parser im Fehlerfall

- Token ignorieren: `class 3 Color { int x; }`
  - → Wenn das darauf folgende Token passen würde
  - → Ignoriert hier `3`.
- Fehlendes Token annehmen/einfügen: `class { int x; }`
  - → Wenn das aktuelle Token auf das nächste Element der Regel passen würde
  - → Nimmt fiktiven Bezeichner als Klassennamen an

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- ANTLR erlaubt Entwickler . . .
  - . . . sich auf Spezifikation der Eingabesprache zu konzentrieren
- Übernimmt dann Implementation von Lexer/Parser
- Unterstützt bei Implementierung von Passes auf AST
- Gleiche Syntax zur Spezifikation von Lexer/Scanner und Parser
- Trennung von Grammatik und Code in der Zielsprache
  - Grammatik bleibt portabel/wiederverwendbar
  - Anwendung kann in gewohnter Sprache programmiert werden
- Mächtiges *ALL(⋆)* Parsing-Verfahren
- Automatisierte Grammatiktransformationen
- Gut unterstützt und aktive Benutzergemeinschaft
  - Sehr gute graphische IDE ANTLRWorks2:
    `http://tunnelvisionlabs.com/products/demo/antlrworks`