

# Compiler 1: Grundlagen

## Lexer/Parser-Generierung mit ANTLR



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

WS 2013/14

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt



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



Inhalt und Beispiele stammen aus:

*The Definitive ANTLR 4 Reference*

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



The Definitive  
**A**NTLR 4  
Reference



Terence Parr

Edited by Susanneth Dautsien Pflaizer



## ANTLR 4.x

- ▶ Adaptive LL(\*) Compiler Generator
  - ▶ Eingabe: Grammatik in EBNF (und mehr!)
  - ▶ Ausgabe: Erkennen für Sprache mittels rekursivem Abstieg
- ▶ Erzeugt (akt. Version 4.1) Erkennen in ...
  - ▶ **Java**, C#, C++
  - ▶ 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



- ▶ 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(★)
  - ▶ 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



## 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



```
grammar Hello; ← Definiere Grammatik "Hello" (der Dateiname muss Hello.g4 sein)
r : 'hello' ID ; ← Erkenne Schlüsselwort hello gefolgt von einem Bezeichner
ID : [a-z]+ ; ← Erkenne Bezeichner in Kleinbuchstaben
WS : [ \r\t\n]+ -> skip ; ← Erkenne Leerzeichen und ignoriere sie
```



- ▶ ANTLR herunterladen

```
$ cd ~/xyz  
$ wget http://antlr4.org/download/antlr-4.1-complete.jar
```

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

```
$ export CLASSPATH=".:~/xyz/antlr-4.1-complete.jar:$CLASSPATH"
```

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

```
$ alias antlr4='java -jar ~/xyz/antlr-4.1-complete.jar'  
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

# TestRig

## Generischer Testtreiber für die Grammatik

- ▶ ANTLR aufrufen und generierte Dateien übersetzen

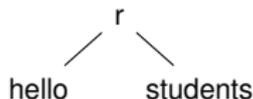
```
$ antlr4 Hello.g4  
$ javac *.java
```

- ▶ 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



<code>/* comments */ // more comments</code>	←	Kommentare (Java-Style)
<code>[lexer parser] grammar &lt;Name&gt;;</code>	←	Spezifikation Grammatikart und -name
<code>options { ... }</code>	←	Optionen, z.B. Zielsprache
<code>import ...</code>	←	Import anderer Grammatiken
<code>tokens { ... }</code>	←	(optionale) Tokenspezifikation
<code>@header { ... }</code> <code>@members { ... }</code>	←	Aktionen: Fügen Code in den Parser ein
<code>rule1: alternativel         alternativ2       ;</code>	←	Parser-Regeln
<code>RULE2: ...</code>	←	Lexer-Regeln (Tokens)



```
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, ...



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

- ▶ Hintereinanderschreiben von Grammatiksymbolen entspricht der Konkatination.
- ▶ 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.

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

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

```
grammar ArrayInit;
```

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

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

```
value : init  
      | INT  
      ;
```

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

oder eine Zahl.

```
INT : [0-9]+ ;
```

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



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 erzeugt einen abstrakten Parse-Baum, bestehend aus

- ▶ **ParserRuleContext**-Objekten
  - ▶ “Kontext”: hier Speicherung aller Information über erkannte Phrase.
  - ▶ Eigene Subklasse für jede Regel (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)



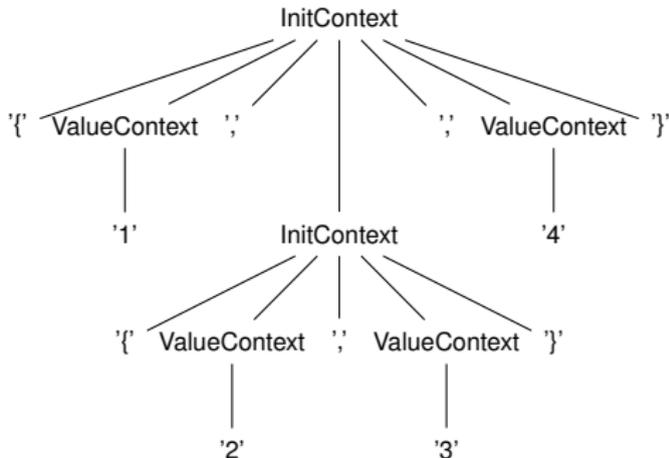
```
grammar ArrayInit;
```

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

```
value : init  
      | INT  
      ;
```

```
INT : [0-9]+ ;
```

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



# Parser

## Generierter Code



```
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() { ... } } ← Zugriff auf Kinder im AST
        ...
    }
    public final ValueContext value() throws RecognitionException { ... }
    ...
}
```

# Parser Integration



```
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(); ← init ist der Name der Regel
        System.out.println(tree.toStringTree(parser));
    }
}
```

→ Lexer und Parser mit Tokenstream verbinden  
Methode für Startsymbol aufrufen

- ▶ 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)



- ▶ 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 Defaultimplementierung (mit leeren Eventmethoden)  
`ArrayInitBaseListener`

# Listener

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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

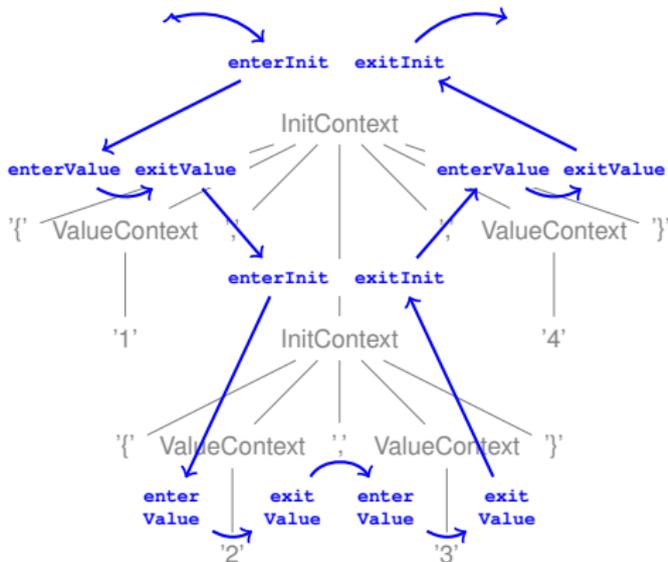
```
grammar ArrayInit;
```

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

```
value : init  
       | INT  
       ;
```

```
INT : [0-9]+ ;
```

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



# Listener

## Anwendungsbeispiel



- ▶ 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

```
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)

```
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

```
grammar Expr;
```

```
import ExprLexerRules;
```

Andere Grammatik importieren (hier: die Lexer-Regeln)

```
prog: stat+ ;
```

```
stat: expr NEWLINE
```

```
| ID '=' expr NEWLINE
```

```
| NEWLINE
```

```
# printExpr
```

```
# assign
```

```
# blank
```

Benennung der Unterregel → Name des Kinds im AST

```
expr: expr op=('*' | '/' ) expr
```

```
| expr op=('+ ' | '- ' ) expr
```

```
| INT
```

```
| ID
```

```
| '(' expr ')'
```

```
;
```

```
# MulDiv
```

```
# AddSub
```

```
# int
```

```
# id
```

```
# parens
```

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

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

# Expression-Grammatik

## Importierte Lexer-Regeln

Deklaration, dass diese Grammatik nur Lexer-Regeln enthält

```
lexer grammar <ExprLexerRules;
```

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

Benennung der Tokens

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

# Visitor-Pattern

## Anwendungsbeispiel

- ▶ 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

```
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



```
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

```
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);
    }
}
```

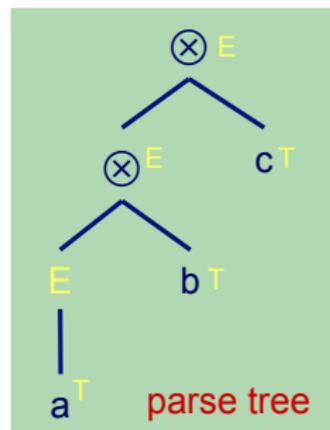


- ▶ Links- und Rechtsassoziativität
- ▶ Operatorpräzedenzen
- ▶ Das “Dangling Else”-Problem
- ▶ Semantische Prädikate
- ▶ Fehlerbehandlung

# Linksassoziativität

## ANTLR v3: Formulierung der Grammatik

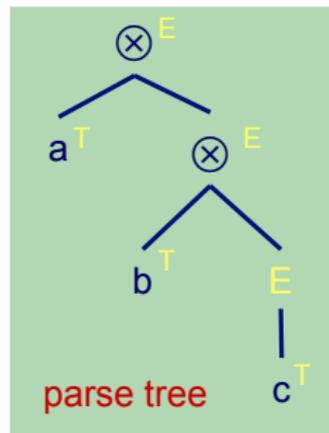
- ▶ Linksassoziativer Operator  $\otimes$ :  
 $a \otimes b \otimes c = (a \otimes b) \otimes c$
- ▶ Produktion (linksrekursiv!)  
 $E ::= E \otimes T \mid T$
- ▶ In EBNF  
 $E ::= T(\otimes T)^*$



# Rechtsassoziativität

## ANTLR v3: Formulierung in der Grammatik

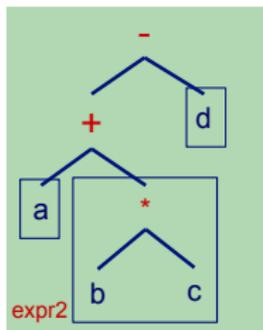
- ▶ Rechtsassoziativer Operator  $\otimes$ :  
 $a \otimes b \otimes c = a \otimes (b \otimes c)$
- ▶ Produktion (linksrekursiv!)  
 $E ::= T \otimes E \mid T$
- ▶ In EBNF (? = 0- oder 1-mal)  
 $E ::= T(\otimes E)?$



# Operatorpräzedenz 1

## ANTLR v3: Formulierung in der Grammatik

- ▶ 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
```

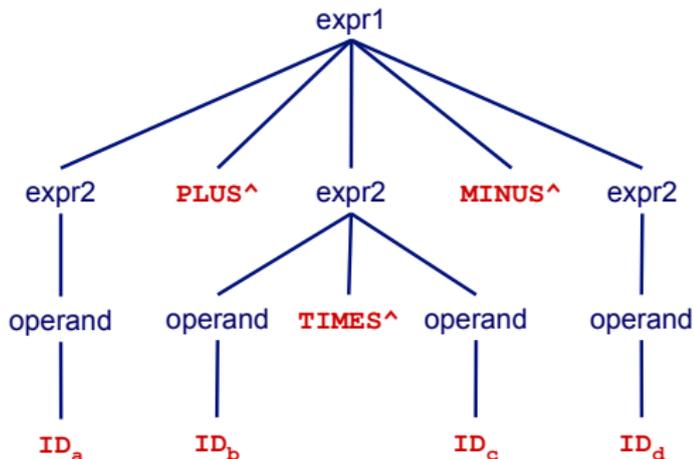
# Operatorpräzedenz 2

$$a + b \times c - d$$

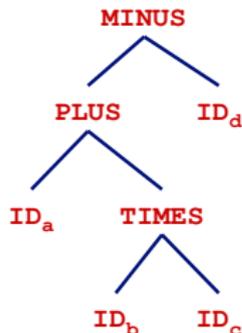


```
expr1 : expr2 ((PLUS^ | MINUS^) expr2)*  
expr2 : operand (TIMES^ operand)*  
operand : IDENTIFIER
```

parse tree:



constructed AST:





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

↓  
Präzedenz

- ▶ 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](http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm)

# Hängendes `else`

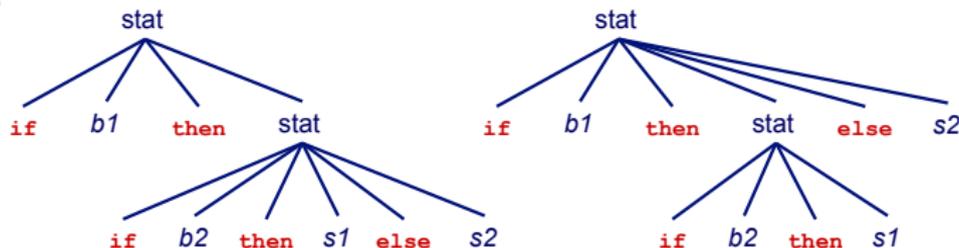
## *dangling else*

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.





- ▶ Zur **Laufzeit** ausgewertete Prädikate qualifizieren Alternativen
  - ▶ Formuliert als Konstrukte der Zielsprache
  - ▶ **TRUE**: Alternative ist möglich
  - ▶ **FALSE**: Alternative ist nicht möglich
  
- ▶ Vorteil: Mächtigerer Parser
  - ▶ Ist  $\epsilon(1)$  Funktionsaufruf oder Zugriff auf Array-Element?
  - ▶ Leicht entscheidbar, wenn Typ von  $\epsilon$  während Parsen abgerufen werden kann
  
- ▶ Nachteil: Nun keine zielsprachenunabhängige Grammatik mehr

# Semantische Prädikate 2

## Beispiel

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

```
grammar Data;
```

```
file : group+ ;
```

```
group: INT sequence[$INT.int] ;
```

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

```
sequence[int n]
```

```
locals [int i = 1;]
```

```
  : ( { $i <= $n } ? INT { $i ++ ; } ) *
```

Definition der Regel mit Parameter n und lokaler Variable i

```
INT : [0-9]+ ;
```

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

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

```
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.
- ▶ Nun deaktiviert `i = 3 <= 2` `false` die (einzige) Alternative von `sequence` deaktiviert.
- ▶ Der Parser kehrt zu `group` und dann `file` zurück, und beginnt mit dem Parsen einer neuen Gruppe.



- ▶ 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();  
}
```

- ▶ 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



- ▶ ANTLR erlaubt Entwickler ...
  - ▶ ... sich auf **Spezifikation** des Compiler zu konzentrieren
  - ▶ Übernimmt dann **Implementation** des Compilers
- ▶ Gleiche **Syntax** zur Spezifikation von
  - ▶ Lexer/Scanner
  - ▶ 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>