



# Compilerbau

Einführung in ANTLR

Prof. Dr. Franz-Karl Schmatzer  
[schmatzf@dhbw-loerrach.de](mailto:schmatzf@dhbw-loerrach.de)

# Literatur

- Terence Parr, *The Definitive ANTLR Reference*, Pragmatic Bookshelf 2007
- Terence Parr, *Language Implementation Patterns*, Pragmatic Bookshelf 2010
- Terence Parr, *The Definitive ANTLR 4 Reference*, Pragmatic Bookshelf 2012
- <https://www.antlr.org/>

# Inhaltsverzeichnis

- ANTLR Überblick
- Einsatzgebiete
- Grammatik Syntax
- Lexer Regeln
- Beispiel-Grammatik
- Token Specification
- Regel Syntax
- Actions

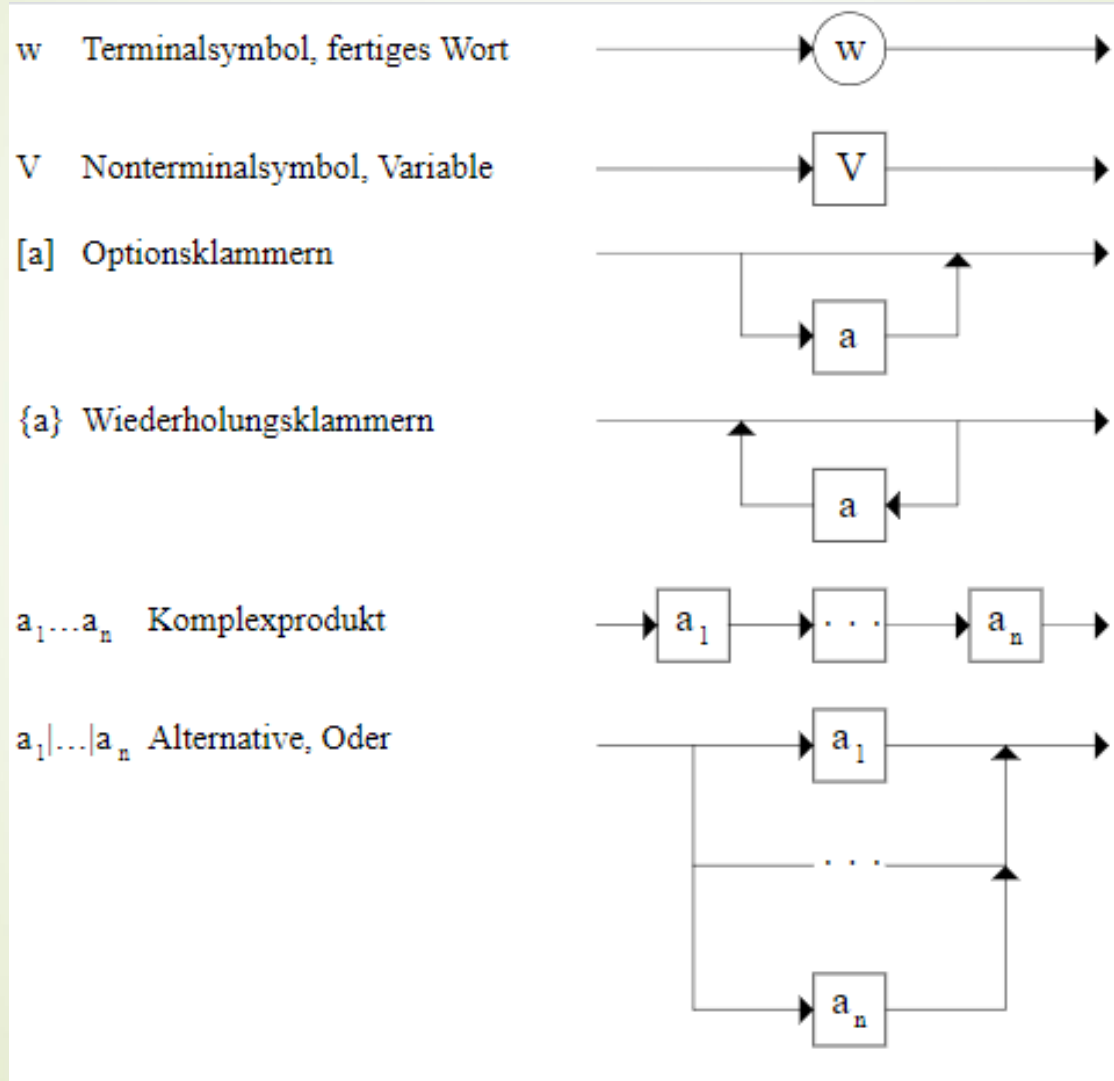
# ANTLR Überblick

- ANother Tool for Language Recognition
  - von Terence Parr in Java geschrieben
- Einfacher zu handhaben als andere Tools
- ANTLRWorks als Plugin oder standalone
  - Grafischer Grammatik- Editor und Debugger
  - von Jean Bovet auf Swing-Basis
- Es lassen sich damit
  - “reale” Programmiersprachen entwickeln oder auch
  - domain-specific Sprachen (DSLs)
- <http://www.antlr.org>
  - Hier gibt es: ANTLR und ANTLRWorks
  - Beide sind frei und open source

# ANTLR Überblick

- Verwendet EBNF-Grammatik
  - Erweiterte Backus-Naur Form
  - Optionale and wiederkehrende Elemente sind modellierbar
  - unterstützt Teilregeln
- Unterstützt viele Ausgabesprachen
  - Default: Java
  - Optional: Ruby, Python, Objective-C, C, C++ and C#
- Plug-ins für IntelliJ und Eclipse

# Grafische Darstellung von EBNF



# ANTLR Überblick

- Unterstützt LL(\*)
  - **LL(k)** Parser: Top-Down Parser
    - parst von links nach rechts
    - erstellt eine Linksableitung der Eingabe
    - Kann k-Token vorausschauen
  - LL-Parser können keine Regeln mit Linksrekursion händeln
- Unterstützt Prädikate
  - Damit lassen sich Zweideutigkeiten in einer Ableitung auflösen

# ANTLR Überblick

Die 3 Haupteinsatzgebiete

## 1. “Validierer“

Erzeugt Code, mit dem man eine Eingabe überprüfen kann, ob diese den Regeln der Grammatik gehorchen.

## 2. “Prozessor“

Erzeugt Code, mit dem eine Eingabe validieren und prozessieren kann

Kann Kalkulation und Update von Datenbanken durchführen

## 3. “Übersetzer“

Erzeugt Code, mit dem eine Eingabe validieren und in ein anderes Format (Programmiersprache, Bytecode) transformieren kann.



# ANTLR Einsatzgebiete

## Programmiersprachen

- Boo (<http://boo.codehaus.org>)
- Code-Blocks C++
- Json-XML Übersetzer
- R-Parser
- XRuby (<http://xruby.com>)

## Andere Projekte

- Hibernate - Übersetzer von HQL zu SQL
- Jazillian - Übersetzer von COBOL, C und C++ in Java

# ANTLR Geschichte

- 1988 startete PCCTS als Parsergenerierungs – Projekt
- anfangs DEA – basierend
  - bis 1990 ANTLR zweimal komplett neu geschrieben
- seit 1990 LL(k)









# Definitionen

- Lexer
  - Der Zeichen-Eingabestrom wird in Token zerlegt.
- Parser
  - Token werden gelesen und prozessiert (und optional ein Syntaxbaum erstellt)
- Syntaxbäume (AST) (abstract Syntax Tree)
  - Ein Baumdarstellung der geparsen Eingabe.
  - Kann einfacher bearbeitet werden als ein Strom von Token.
  - Kann sehr effizient mehrfach durchlaufen werden.
- Tree Parser
  - Prozessiert ein AST
- StringTemplate
  - Ein Bibliothek, welche Templates zur Verfügung stellt
  - Textausgabe (z.B. Java source code)

# Implementieren mit ANTLR I

- ANTLR produziert auf Basis einer Grammatik T mit dem File-Namen T.g4 einen Lexer und Parser.
  - Installation von ANTLR (<https://www.antlr.org/>)
    - antlr-4.8-complete.jar Download
    - CLASSPATH setzen (auf das obige jar-File )
    - Ausführen mit
    - alias antlr4='java org.antlr.v4.Tool'
- >antlr4 Hello.g4

ANTLR Parser Generator Version 4.8

->  Hello.interp	29.04.2020 17:53	INTERP-Datei
 Hello.tokens	29.04.2020 17:53	TOKENS-Datei
 HelloBaseListener.java	29.04.2020 17:53	JAVA-Datei
 HelloLexer.interp	29.04.2020 17:53	INTERP-Datei
 HelloLexer.java	29.04.2020 17:53	JAVA-Datei
 HelloLexer.tokens	29.04.2020 17:53	TOKENS-Datei
 HelloListener.java	29.04.2020 17:53	JAVA-Datei
 HelloParser.java	29.04.2020 17:53	JAVA-Datei

# Implementieren mit ANTLR II

- Files:
- Files mit den Tokens: \*.tokens
- Java-Code für den Lexer und Parser
- Java-Code als Listener und BaseListener
- Kompilieren des Codes mit javac
- `> javac *.java`
- Nun hat man den Byte-Code der Java-Klassen

# Implementieren mit ANTLR III

- ANTLR bietet eine Testumgebung für die erstellten Leser und Parser.
  - `alias grun = 'java org.antlr.v4.runtime.misc.TestRig'`
  - Mit dem alias kann man dann:
    - `grun <Grammatik> S -tokens # S: Startregel die Tokens erhalten`
    - `grun <Grammatik> S -tree # den Ableitungsbaum als Liste`
    - `grun <Grammatik> S -gui # den Ableitungsbaum als Grafik`

# Beispiel Hello.g4

```
grammar Hello ;                                // Define a grammar called Hello
r : 'hello' ID ;                               // match keyword hello followed by an identifier
ID : [ a - z ] + ;                             // match lower-case identifiers
WS : [ \t \r \n ] + -> skip ;                 // skip spaces, tabs, newlines, \r (Windows)
```

Erstellen Sie die Grammatik und legen Sie es im File Hello.g4 ab.

Übersetzen Sie diese Grammatik: `antlr4 Hello.g4`

und Kompilieren: `javac *.Java`

Nun mit grun arbeiten:

-> `grun Hello r -tokens`

-> `grun Hello r -tree`

-> `grun Hello r -gui`

Die Eingabe von grun ist jedesmal "hello DHBW". String eingeben und abschließen mit ctrl D (Linux) bzw. Ctrl Z (Windows)

# IDE mit Eclipse

- Eclipse herunterladen
- Implementieren der ANTLR-IDE für Eclipse
- Implementieren der ANTLRDT Tools für Eclipse
- Webseite mit der Installationsanweisung:
- <https://www.antlr.org/tools.html>



# Eclipse Plugin

The screenshot displays the Eclipse IDE interface with a grammar definition file named `b1.g4`. The editor shows the following code:

```
2 * Define a grammar called Hello
4 lexer grammar b1;
5   LETTER: [a-zA-Z] ;
6   DIGIT:  [0-9];
7   SIGN:   '+' | '-';
8   WS:     [ \t ]+;
9   NEWLINE: [ '\r'? '\n' ]+;
10  ID: LETTER (LETTER | DIGIT)*;
11  INT:  SIGN? DIGIT+;
```

Below the editor, the **Syntax Diagram** tab is active, showing the visual representation of the grammar rules:

- LETTER**: A single box labeled `[a-zA-Z]`.
- DIGIT**: A single box labeled `[0-9]`.
- SIGN**: A box containing two sub-boxes, `+` and `-`, connected by a vertical line.
- WS**: A box labeled `[ \t ]` with a large loop on the right side, indicating one or more occurrences.
- NEWLINE**: A box labeled `[ '\r'? '\n' ]` with a large loop on the right side, indicating one or more occurrences.
- ID**: A box labeled `LETTER` followed by a large loop containing two sub-boxes, `LETTER` and `DIGIT`, connected by a vertical line.
- INT**: A box labeled `SIGN` followed by a box labeled `DIGIT` with a large loop on the right side, indicating one or more occurrences.

The right side of the IDE shows the **Task List** and **Outline** views. The **Outline** view lists the grammar rules: `LETTER`, `DIGIT`, and `SIGN`.

# Beispiel einer Grammatik

**lexer grammar b1;**

*LETTER:*        *[a-zA-Z] ;*

*DIGIT:*        *[0-9];*

*SIGN:*        *'+' | '-';*

*WS:*        *[ \t]+;*

*NEWLINE:*   *['\r'? '\n']+;*

*ID:*        *LETTER (LETTER | DIGIT)\*;*

*INT:*        *SIGN? DIGIT+;*

# Erstellen der Regeln - Metazeichen

- ANTLR unterstützt EBNF, dh. BNF mit

- Wiederholung
- Optionale Operatoren
- Teilregeln

- Metazeichen der Sprache

.. ~ ? | + \* ( ) { } [ ] .

- Metazeichen

	Zeichenfolge	abc	trifft auf die Zeichenfolge abc zu
..	Bereich	A..Z	Zeichen von A bis Z
~	Ausschluss	~a	nicht A
?	Optional	a?	optional a
	Alternative	a   bc	a oder bc
+	ein oder mehrmals	x+	x, oder xx oder xxx ...
*	0 oder mehrmals	x*	kein x oder x oder xx oder ..
()	Unterregeln	(a   cb)+	
.	Alle Zeichen	.	Trifft auf alle Zeichen zu

# Aufstellen der Grammatik für den Lexer

- Für jedes Token muss eine Regel angegeben werden
- Der Name muss mit einem Großbuchstaben anfangen
  - Typisch wird der gesamte Name in Großbuchstaben geschrieben
- Ein Token kann vergeben werden für
  - ein einzelnes Zeichen der Eingabe
  - eine Zeichenfolge der Eingabe
  - ein oder mehrere Zeichen oder Zeichenbereiche
- Man kann auf andere Lexer-Regeln verweisen
- "fragment" lexer Regel
  - ergibt kein Token
  - Ist nur eine Referenz auf andere Lexer-Regeln
- Subregeln und Optionale Parameter
- Kommentare wie in Java
  - `//` Einzeilige Kommentare
  - `/*` mehrzeilige Kommentare `*/`

# Beispiel Lexer-Regeln

- Anwenden der **fragment**-Regel

INT: **DIGIT**+; //references the DIGIT helper rule

**fragment DIGIT:** [0-9]; // not a token by itself

---

INT wird ein Token

DIGIT wird kein Token

# Allgemeiner Regel- Aufbau

- Der allgemeine Aufbau der Regeln für den Lexer

```
</lexer> grammar <name>;
```

```
/* Optionale Parameter */
```

```
<Options-Spec>
```

```
<Token-spec>
```

```
<Attribute-scopes>
```

```
<Actions>
```

```
/* Pflicht Parameter */
```

```
RULE: ... | ... | ... ;
```

<name> muss gleich  
dem File-Namen  
<name>.g4 entsprechen

2 Grammatik-Typen:  
Lexer und parser.

Wenn keine Grammatik  
gegeben wird lexer und  
parser kombiniert

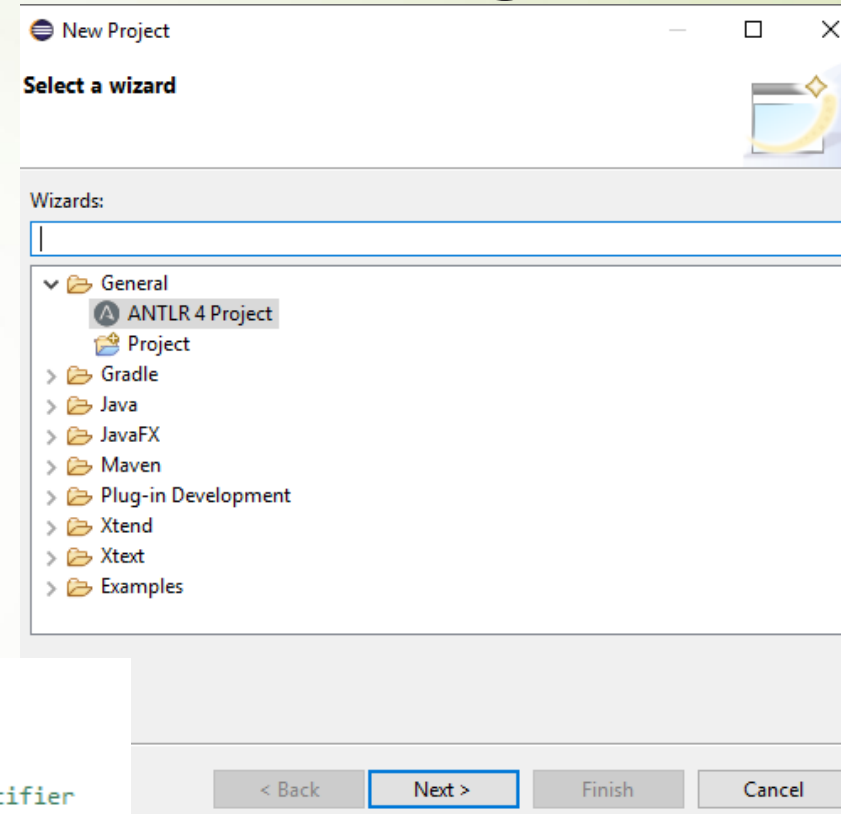
Kommentare wie in Java.

Die Klassen die ANTLR erzeugt, haben für jede Regel eine Methode.

# ANTLR-Arbeiten mit dem Plugin I

- Anlegen eines Antlr-Projekts:
  - File->New->project
  - Auswahl ANTLR 4 Projekt
  - Next
  - Projektname angeben
  - finish- drücken
  - Es wird eine Hello.g4 Grammatik erzeugt

```
1 /**
2  * Define a grammar called Hello
3  */
4 grammar Hello;
5 r : 'hello' ID ;           // match keyword hello followed by an identifier
6
7 ID : [a-z]+ ;             // match lower-case identifiers
8
9 WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
10
11 |
```



# ANTLR-Arbeiten mit dem Plugin II

- Kompilieren des Antlr-Projekts:
  - Run->Run
- Klicken auf die Grammatik, dann wird unter Syntax-Diagramm die EBNF-Form der Grammatik angezeigt
- Klicken auf den Parse-Tree zeigt noch nichts an. Man muss zuerst die Regel auswählen, die man als Startpunkt wählen will.
- Eingabe des zu parsenden Ausdrucks im Fenster unter rechts.
- Im Fenster links wird der geparste Baum angezeigt.



# ANTLR-Arbeiten mit dem Plugin III

- Eingabe des zu parsenden Ausdrucks muss schnell erfolgen, da der Parser sofort anfängt die Eingabe zu interpretieren.

The screenshot displays the ANTLR4 IDE interface. The top pane shows the grammar file `Hello.g4` with the following content:

```
1 /**
2  * Define a grammar called Hello
3  */
4 grammar Hello;
5 r : 'hello' ID ;           // match keyword hello followed by an identifier
6
7 ID : [a-z]+ ;             // match lower-case identifiers
8
9 WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
10
11 |
```

The bottom pane is divided into two sections. The left section, titled `Hello::r`, shows the input `hello par` with `hello` highlighted in red. The right section displays the parse tree structure:

```
graph TD
    r((r)) --- hello[hello]
    r --- par[par]
```

# Aufgaben

- Erstellen Sie die Grammatik b1 von zuvor und zeigen Sie das Syntax-Diagramm an.
- Erstellen Sie eine Grammatik für Integerwerte
- Erstellen Sie eine Grammatik für Integerwerte und Bezeichner, die nur aus Klein- und Großbuchstaben bestehen sollen.
- Erstellen Sie eine Grammatik für Integerwerte, Bezeichner und Floatingpointwerte
- Geben Sie die Syntax-Diagramme an und Parsen Sie die Eingabe verschiedener Werte.
- Sie können auch die Workbench von ANTLR nutzen.  
(<http://labantlr.org/>)

# Allgemeine Schritte

- Aufstellen einer Grammatik in einer IDE oder mit einem Editor
- Übersetzen mit antlr4 und die Klassenfiles erzeugen.
- Überprüfen, ob die Grammatik korrekt ist.
  - Es werden dabei Klassen generiert (Lexer, Parser)
  - Diese sind konform zu der vorher spezifizierten Grammatik
  - Diese Klassen liegen in der gewählten Sprache vor (Default: Java)
- Erstellen einer Applikation, welche diese Klassen verwendet.
  - ANTLR separiert die Applikation von der Grammatik und bietet zwei Schnittstellen für die Nutzung an:
    - Listener-Schnittstelle
    - Visitor-Schnittstelle

# Beispiel für eine Sprache – Expr.g4

- Arithmetische Operationen
  - Operatoren Plus, Minus Multiplikation und Division mit der üblichen arithmetischen Reihenfolge z.B  $3+4*5-1$ .
  - Klammerung von Ausdrücke z.B  $(3+4)*5-(3+4)$ .
  - Variablen Definition ( $a=3$ ,  $b=4$ ,  $c=2*a*b$ )
  - Nur Integerwerte
- Definition der Sprache
  - wird im File Expr.g4 abgelegt
  - Grammar Expr;  
<<regeln>>;
- Wie sehen die Regeln aus?

# Beispiel für eine Sprache – Expr.g4

## ➤ Allgemeiner Aufbau

```
prog:      stat+; // +: 1 und mehr gibt es in prog
stat:      expr NEWLINE
          | ID '=' expr NEWLINE
          | NEWLINE
          ;
```

## ➤ Das heißt unser Programm besteht aus einzelne statements (stat)

## ➤ Der Aufbau von stat :

- Ein Ausdruck (expr) gefolgt von einer neuen Zeile (NEWLINE) oder (|) Eine Variable (ID) gleich (=) einem Ausdruck (expr) gefolgt von einer neuen Zeile (NEWLINE) oder (|) eine neue Zeile (NEWLINE)

## ➤ Nun müssen wir näher spezifizieren, was wir unter einem Ausdruck expr verstehen.

## Beispiel für eine Sprache – Expr.g4

- Wir haben ( '+' , '-' ) und ( '\*' , '/' ),
- um die Vorfahrtsregeln (Operatorpräzedenz) zu wahren, nimmt ANTLR die erste Regel als Vorrang vor den anderen Regeln.

```
expr:      expr ( '*' | '/' ) expr
          | expr ( '+' | '-' ) expr
          | INT
          | ID
          | '(' expr ')'
          ;
```

Nun müssen wir noch die Token spezifizieren (Große Buchstaben)

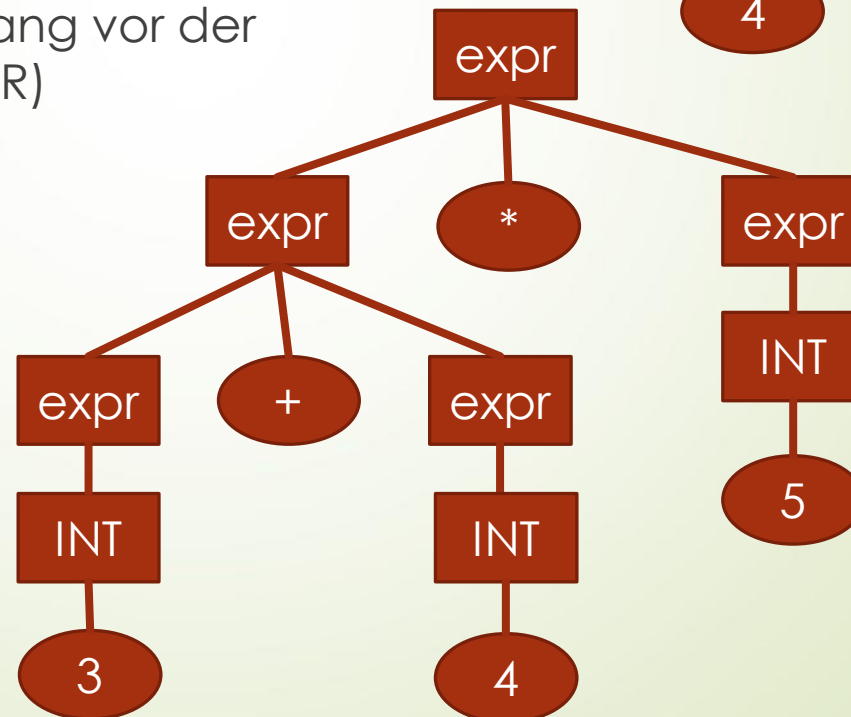
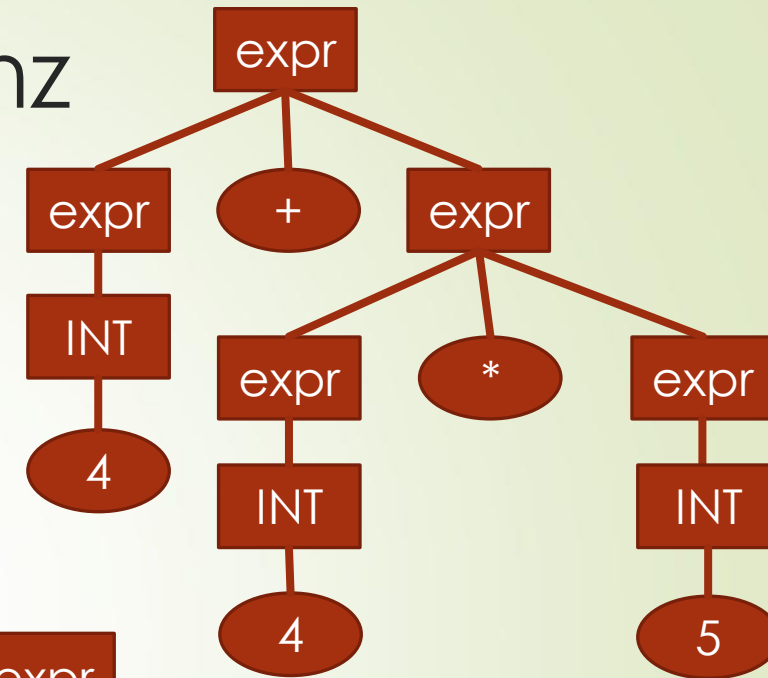
```
ID:      [a-zA-Z]+ ;
INT:      [0-9]+;
NEWLINE:  '\r'? '\n'; //return
WS:      [ '\t' ]+ ->skip ; // Whitespace entfernen
```

# Operatorpräzedenz

➤ Regel:

expr:        expr ('\*' | '/') expr  
              | expr ('+' | '-') expr  
              | INT

- Beispiel 3+4\*5
- Ableitungsbaum nicht eindeutig
- Erste Regel hat Vorrang vor der zweiten Regel (ANTLR)



# Beispiel für eine Sprache - Expr.g4

```
grammar Expr;
prog:    stat+;
stat:    expr NL | ID '=' expr NL | NL ;
expr:    expr ('*' | '/') expr | expr ('+' | '-') expr | INT | ID | '(' expr ')' ;

ID: [a-zA-Z]+;
INT: [0-9]+;
NL: '\r'? '\n'; // newlines
WS: [ \t]+ ->skip; // skip spaces, tabs
```



# Beispiel Eingabe/Ausgabe

- Beispiel für die Ausgabe mit `grun Expr prog -gui t.expr`

**t.expr**

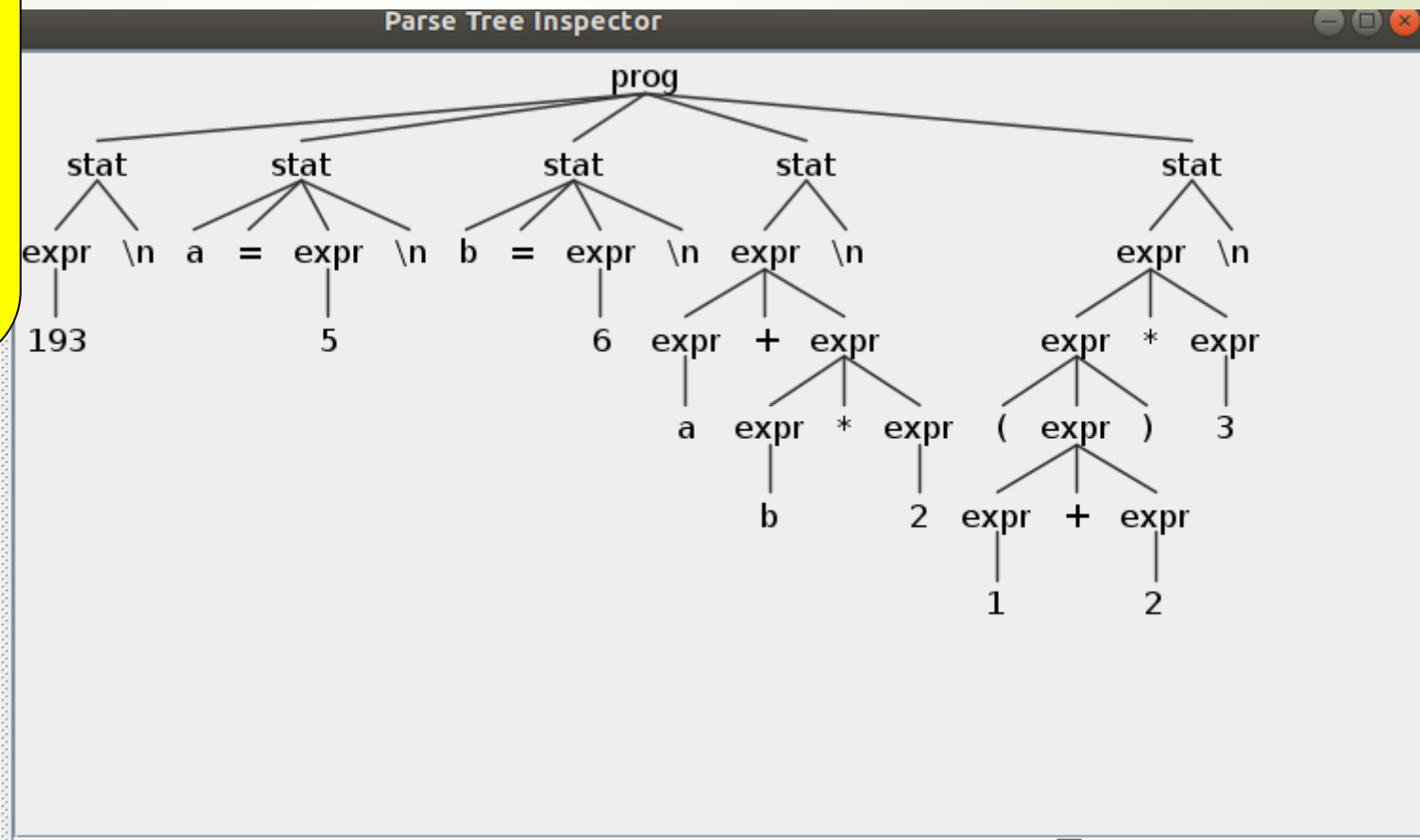
193

a = 5

b = 6

a+b\*2

(1+2)\*3



# Aufgabe Expression

- Erstellen Sie die Grammatik für arithmetische Ausdrücke und geben Sie den Operatorbaum für die Eingabe `t.expr` aus.

# Beispiel

## Files die erzeugt werden

- Expr.g4 Grammatik
- Expr.tokens File mit den Token
- ExprLexer.java Java-Klasse des Lexers
- ExprParser.java Java-Klasse des Parsers
- ... /andere Files
- Entwickeln einer Applikation, welche die java-Klasse einbindet. (Expr.java)
- Dazu das Listener-Interface nutzen.

# Lexer Regeln

- Für jedes Token muss eine Regel angegeben werden
- Der Name muss mit einem Großbuchstaben anfangen
  - Typisch wird der gesamte Name in Großbuchstaben geschrieben
- Ein Token kann vergeben werden für
  - ein einzelnes Zeichen der Eingabe
  - eine Zeichenfolge der Eingabe
  - ein oder mehrere Zeichen oder Zeichenbereiche
    - **Man kann Kardinalitäten wie (?, \* und +) vergeben**
- Man kann auf andere Lexer-Regeln verweisen
- “fragment” lexer Regel
  - ergibt kein Token
  - Ist nur eine Referenz auf andere Lexer-Regeln

# Lexer Regeln - Beispiel

FLOAT: DIGIT+ '.' DIGIT\* | '.' DIGIT\* ; // 1.234 und .2345

STRING: '"'.\*? '"'; //matches anything in "... " einfache Regel

// .\* nimmt alle Zeichen ohne Rücksicht auf andere Regeln. Das ? begrenzt das auf ein Minimalmenge (non-greedy), indem es die anderen Regeln in der Grammatik noch berücksichtigt.

fragment

DIGIT : [0-9]; // match a single digit // DIGIT wird nicht zu einem Token.

# Leerzeichen & Kommentare

- Wird durch Lexer-Regeln behandelt

- mögliche Optionen

  - Z.B. Wegwerfen:

-> skip ;

- Beispiele

Vorsicht NEWLINE nicht wegwerfen oder auf einen eigenen Kanal schreiben, wenn man sie später als Terminatoren für Statements benötigt.

**WHITESPACE:**

**WHITESPACE:**

**NEWLINE:**

**COMMENT:**

`(' ' | '\t' | ' \r' | '\n')+ -> skip ; oder`

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

`(' \r'? '\n')+ -> skip ;`

`/* .*? */ -> skip ;`

# Echte Grammatiken

- Parsing Komma-Separierte Werte in einer Datei (CSV Files)
- Das File besteht aus einer Headerzeile und nachfolgende Zeilen. Die Zeilen enthalten Werte, die Komma separiert sind und am Schluss mit einem Newline enden.
- Wie könnte die Grammatik aufgebaut sein?

# Aufgabe CSV

Erstellen Sie die Grammatik und legen Sie es im File CSV.g4 ab.

Übersetzen Sie diese Grammatik: `antlr4 CSV.g4`

und Kompilieren: `javac CSV*.Java`

Nun mit grun arbeiten:

-> `grun CSV file -tokens data.csv`

-> `grun CSV file -tree data.csv`

-> `grun CSV file -gui data.csv`

data.csv

Name,Alter,Wohnort

Maier,26,Basel

Friedrich,33,Freiburg

Berthold,,Bern



# Lösung CSV-Grammatik

## ► Mögliche Grammatik

```
grammar CSV;
```

```
file : hdr row+ ;
```

```
hdr : row ;
```

```
row : field (',' field)* '\r'? '\n' ;
```

```
field : TEXT | STRING | ;
```

```
TEXT : ~[, \n \r']+ ;
```

```
STRING : '"' ( '"' | ~'"')* '"' ; // quote-quote is an escaped quote
```

# Grammatik mit Actions

- Actions sind:

- Programmcode, welcher in der zugehörigen Programmiersprache verfasst ist und innerhalb von geschweiften Klammern gesetzt wird.

```
decl: type ID ';' {System.out.println("found a decl");}  
type: 'int' | 'float' ;
```

- Man kann auch auf die Attribute der Tokens und die Regeln zugreifen.

```
decl: type ID ';' {System.out.println("var "+$ID.text+" "+$type.text+"");}  
| t=ID id=ID ';' {System.out.println("var "+$id.text+" "+$t.text+"");};
```

- Genaue Beschreibung unter:

<https://github.com/antlr/antlr4/blob/master/doc/actions.md>

# Grammatik mit Actions

## Token Attribute

- Alle Token haben eine Menge an vordefinierten Attributen.
- Attribute sind z.B. Token-Typ oder Text, der von einem Token erkannt wird.
- Mit Actions kann man darauf zugreifen via `$label.Attribute`, wobei `label` der Name des Token entspricht.
- Beispiel:

```
r : INT {int x = $INT.line;}
```

```
( ID {if ($INT.line == $ID.line) ...;} )? a=FLOAT b=FLOAT {if ($a.line == $b.line) ...;};
```

- `a` und `b` werden hier im Action-code für `FLOAT` genutzt, da der Token mehrmals vorkommt. Der Token `INT` ist in der Regel eindeutig. Daher kann man ihn direkt verwenden.

# Token Attribute

Attribute	Type	Description
text	String	The text matched for the token; translates to a call to <code>getText</code> . Example: <code>\$ID.text</code> .
type	int	The token type (nonzero positive integer) of the token such as <code>INT</code> ; translates to a call to <code>getType</code> . Example: <code>\$ID.type</code> .
line	int	The line number on which the token occurs, counting from 1; translates to a call to <code>getLine</code> . Example: <code>\$ID.line</code> .
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to <code>getCharPositionInLine</code> . Example: <code>\$ID.pos</code> .
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to <code>getTokenIndex</code> . Example: <code>\$ID.index</code> .
channel	int	The token's channel number. The parser tunes to only one channel, effectively ignoring off-channel tokens. The default channel is 0 ( <code>Token.DEFAULT_CHANNEL</code> ), and the default hidden channel is <code>Token.HIDDEN_CHANNEL</code> . Translates to a call to <code>getChannel</code> . Example: <code>\$ID.channel</code> .
int	int	The integer value of the text held by this token; it assumes that the text is a valid numeric string. Handy for building calculators and so on. Translates to <code>Integer.valueOf(text-of-token)</code> . Example: <code>\$INT.int</code> .

# Actions

## Hello Beispiel

### Nur Ausgabe

```
grammar Act1;  
r : 'hello' ID  
  {System.out.println("found hello!");};  
ID : [a-z]+ ;  
WS : [\t\r\n]+ -> skip ;
```

### Zugriff auf ein Token

```
grammar Act2;  
r : 'hello' ID  
  {System.out.println("found ID =  
    "+$ID.text;)};  
ID : [a-z]+ ;  
WS : [ \t\r\n]+ -> skip ;
```

Implementieren Sie diese beiden Grammatiken in ANTLR, kompilieren Sie ihn und führen den Code aus.

# Grammatik mit Actions

## Parser Attribute

- ANTLR predefinedes a number of read-only attributes associated with parser rule references that are available to actions.
- Actions can access rule attributes only for references that precede the action.
- Die Syntax ist: `$r.attr` für eine Regel `r` oder Zugriff über einen Label, der auf eine Regel verweist.
- Beispiel für den direkten Zugriff:

```
returnStat : 'return' expr {System.out.println("matched " + $expr.text);};
```

`$expr.text` returns the complete text matched by a preceding invocation of rule `expr`

- Zugriff über ein Label:

```
returnStat : 'return' e=expr {System.out.println("matched " + $e.text);};
```

# Actions

## Beispiel einfaches Rechnen

```
grammar Act3;

stmt : expr NEWLINE {System.out.println("Das Ergebnis ist: "+$expr.value);};

expr returns [int value]:

    m=atom {$value=$m.value;}

    ('+' j=atom {$value += $j.value;} | '-' k=atom {$value -= $k.value;}) *;

atom returns [int value]:

    INTEGER {$value = Integer.parseInt($INTEGER.text);};

INTEGER: '0'..'9'+;

NEWLINE: '\r'? '\n';

WS : [ \t ]+ -> skip ; // skip spaces, tabs
```

# Actions

## Beispiel einfaches Rechnen

```

grammar Act4;

@header {import java.util.HashMap;}

@members {/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();}

prog: stmt+ ;

stmt : expr NEWLINE{System.out.println($expr.value);} | ID '=' expr NEWLINE
{memory.put($ID.text, new Integer($expr.value));System.out.println($expr.value);} | NEWLINE;

expr returns [int value]:

i=atom {value=$i.value;} ('+' i=atom {$value += $i.value;} | '-' i=atom {$value -= $i.value;}) *;

atom returns [int value]: INTEGER {$value = Integer.parseInt($INTEGER.text);} |
ID {Integer v = (Integer)memory.get($ID.text);
if ( v!=null ) $value = v.intValue();else System.err.println("undefined variable "+$ID.text);};
INTEGER: '0'..'9'+;
ID : ('a'..'z'|'A'..'Z')+ ;
NEWLINE: '\r'? '\n';
WS : [ \t]+ -> skip ; // skip spaces, tabs

```