



Compilerbau

Einführung in ANTLR

Prof. Dr. Franz-Karl Schmatzer
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
- ANTLR Installation
- Grammatik Syntax
- Lexer Regeln
- Beispiel-Grammatik
- Token Spezifikation
- Regel Syntax

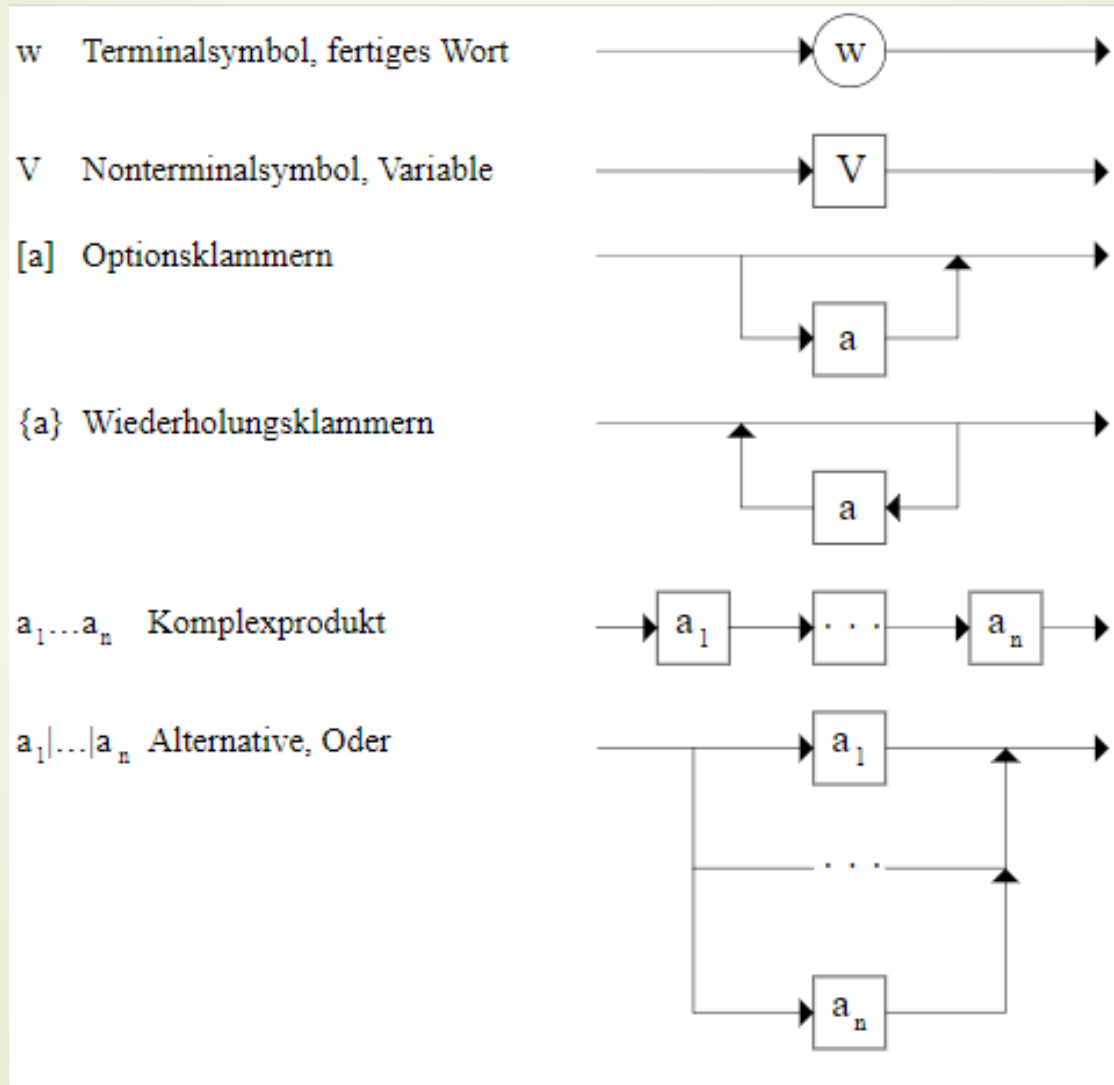
ANTLR Überblick

- 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 Parsergenerierungsprojekt
- 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
 - Eine 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/>) bzw.
 - <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>
 - Installationsanweisungen befolgen
 - Ausführen mit
 - > `java org.antlr.v4.Tool Hello.g4`
 - ANTLR Parser Generator Version 4.12
 - > Files generiert:

 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

ANTLR Development Tools

- IntelliJ Plugin, NetBeans Plugin, Visual Studio Code plugin, Eclipse Plugin usw.
- Implementieren der ANTLR-IDE
- Webeseite 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 grammar is defined as follows:

```
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+;
```

The IDE also shows the generated syntax diagram for the grammar. The diagram consists of several components:

- LETTER**: A box containing the regular expression `[a-zA-Z]`.
- DIGIT**: A box containing the regular expression `[0-9]`.
- SIGN**: A box containing the regular expression `'+' | '-'`.
- WS**: A box containing the regular expression `[\t]+`.
- NEWLINE**: A box containing the regular expression `['\r'? '\n']+`.
- ID**: A box containing the regular expression `LETTER (LETTER | DIGIT)*`.
- INT**: A box containing the regular expression `SIGN? DIGIT+`.

The syntax diagram is a visual representation of the grammar rules, showing how they are combined to form the final expressions. The diagram is organized into a hierarchy, with the root node being the `INT` rule, which is composed of the `SIGN` and `DIGIT` rules. The `DIGIT` rule is further composed of the `DIGIT` rule itself, and the `LETTER` rule is composed of the `LETTER` rule itself.

Beispiel einer Grammatik

```
grammar B1;  
  
// Parser Rules  
start : (ID | INT)+ ;  
  
// Lexer Rules  
ID : LETTER (LETTER| DIGIT)*;  
INT : SIGN? DIGIT+;  
LETTER : [a-zA-Z] ;  
DIGIT : [0-9];  
SIGN : '+' | '-';  
WS : [ \t\r\n]+ -> skip;
```


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

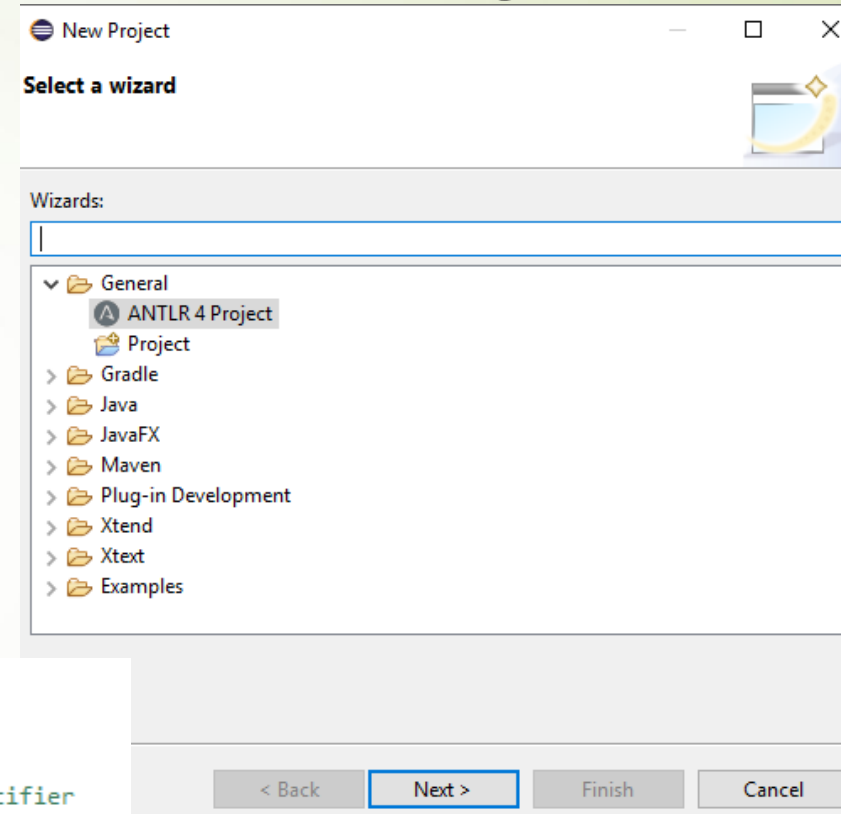
Kommentare wie in Java.

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 editor window, titled 'Hello.g4', contains the following grammar definition:

```
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 |
```

Below the editor, the 'Parse Tree' tab is active, showing the parse tree for the input 'Hello::r'. The tree structure is as follows:

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

The input 'Hello::r' is shown in the input field, and the parsed tokens 'hello' and 'par' are displayed below it.