Der Parsergenerator ANTLR

Hans Jürgen Ohlbach

Keywords: ANTLR, LL(*)-Parsing, Java API für den Syntaxbaum

Empfohlene Vorkenntnisse: Formale Sprachen, Typ 2-Sprachen, Kellerautomaten

Inhaltsverzeichnis

1 Einführung

ANTLR ist ein Parser Generator für kontextfreie Sprachen. Er wurde von Terence Parr programmiert und wird seit 1989 permanent weiterentwickelt. ANTLR steht für `ANother Tool for Language Recognition`. Das Programm ist freie Software und läuft auf Java Plattformen. Es kann von http:www.antlr.org bezogen werden.

Eingabe für ANTLR ist eine Grammatik in einer für ANTLR speziellen Syntax. Ausgabe ist eine Reihe von Java Klassen, die den Parsergenerator darstellen. Den Parsergenerator wiederum kann man auf konkrete Wörter anwenden und erhält entweder Fehlermeldungen oder einen Syntaxbaum. Zusätzlich erzeugt ANTLR Schnittestellen, um den Syntaxbaum zu bearbeiten.

Dieser Text kann natürlich keine vollständige Beschreibung von ANTLR geben. Dazu gibt es genügend Dokumentation und Tutorials. Es soll hier nur ein erster Eindruck von ANTLR's Möglichkeiten gegeben werden.

2 Lexer und Parser

In der Theorie der formalen Sprachen gibt es für jede Sprache ein Alphabet Σ, und daraus geformt die Wörter $w \in \Sigma^*$. In realistischen Sprachen, wie z.B. Programmiersprachen, gibt es aber noch etwas dazwischen. In diesen Sprachen gibt es Schlüsselwörter wie `if, then, else, while` usw. Dann gibt es Bezeichner für Variablen, Methoden, Klassen usw. Weiterhin gibt es Teilsprachen für Datentypen, wie z.B. Integer- oder Floating-Point-Typen. All diese Komponenten in einer einzige Grammatik zu spezifizieren, würde sie extrem überladen und unleserlich machen. Stattdessen spezifiziert man diese einfachen Komponenten separat, so dass man einen längeren String in Einzelkomponenten, sog. *Tokens*, zerlegen kann, die man dann dem eigentlichen Parser übergibt.

Beispiel: den String
`if(ab < 3.7) {cd = f(ab);} else {cd = g(ab);}`
würde man in gerne in folgende Sequenz von Tokens zerlegen:
`|if|(|ab|<|3.7|)|{|cd|=|f|(|ab|)|;|}|else|{|cd|=|g|(|ab|)|;|}|`
wobei auch gleich die Leerzeichen entfernt werden. Diese Tokens bilden dann die Terminalsymbole der eigentlichen Sprache.

Alle praktisch verwendbaren Parsergeneratoren, und daher auch ANTLR, folgen diesem Konzept:
1. Man definiert die Teilsprache der Tokens.
2. Der daraus erzeugte sog. *Lexer* zerlegt dann den zu parsenden String in eine Sequenz von *Tokens*.
3. Man definiert die eigentliche Sprache, auf Basis der Tokens als Terminalsymbole.
4. Der daraus erzeugte *Parser* kann dann die Folge von Tokens parsen.

Beispiel: Für ANTLR würde man die Aufteilung für den Lexer und Parser wie in folgendem Beispiel definieren. Das Beispiel spezifiziert arithmetische Terme wie z.B. `12 + (34 * 56)`.

```
expr: expr OP expr |
      '(' expr ')' |
      INT;

OP : '+'|'-'|'*'|'/';
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Die Bezeichner mit Großbuchstaben, OP, INT und WS definieren die Sprache für den Lexer. Insbesondere `INT: [0-9]+`, mit dem regulären Ausdruck `[0-9]+`, definiert Integer als beliebig lange Folge von Ziffern. Damit kann der Lexer Integer wie 12345 als Einheit erkennen und dem Parser übergeben. `WS` steht für „whitespace“ und bewirkt, dass alle Leerzeichen entfernt werden.

Der String

`12 + (34 * 56)` würde damit vom Lexer in die Folge
`|12|+|(|34|*|56|)|` zerlegt werden.

Die Variablen für den Parser beginnen mit Kleinbuchstaben. Im Beispiel ist es nur `expr`. Der Parser sieht dann insbesondere die Integer als Einheit, und braucht sich nicht um deren interne Struktur zu kümmern.

3 LL(*)-Parsing

ANTLR ist ein sog. LL(*)-Parsergenerator für kontextfreie Sprachen.

In LL steht das erste L für *Left to right*, was Top-Down-Parsing entspricht. Das zweite L steht für *Leftmost* und bezieht sich darauf, dass von den u.U. mehreren möglichen Syntaxbäumen der „linkeste“ erzeugt wird. Das ist derjenige, der in der Reihenfolge der Produktionsregeln der Grammatik mit den jeweils ersten Regeln mögliche.

Typische kontextfreie Grammatiken sind in der Regel erst einmal nichtdeterministisch. Das sieht man schon an der kleinen Sprache im obigen Beispiel. Für die Variable `expr` gibt es drei Regeln. Beim top-down Parsing muss sich der Parser dann für eine davon entscheiden. Wählt er die falsche, dann muss er backtracken. Dies ist extrem ineffizient, und sollte daher möglichst vermieden werden, bzw. der Parser sollte in der Lage sein, mit geeigneten Tests zu entscheiden, welche der Möglichkeiten die richtige ist. Eine richtige Entscheidung kann er aber nur treffen, wenn er *vorausschauen* kann. Daher hat man den Begriff LL(k)-Parsing, bzw. LL(k)-Sprachen geprägt. LL(1)-Sprachen sind Sprachen, bei denen der Parser mit *einem* Token Vorausschau entscheiden kann, welches die richtige Regel ist. Entsprechend sind LL(k)-Sprachen Sprachen, bei denen der Parser mit k Token Vorausschau entscheiden kann, was die richtige Regel ist. Damit kann er *deterministisch* arbeiten, und braucht kein Backtracking.

LL(*)-Parsing ist wiederum eine Erweiterung von LL(k)-Parsing, bei dem das k flexibel bestimmt wird. Dafür analysiert man die Grammatikregeln, und versucht, für jede Regel ein Anfangsstück der rechten Seite zu finden, die man mit einem endlichen Automaten auseinanderhalten kann.

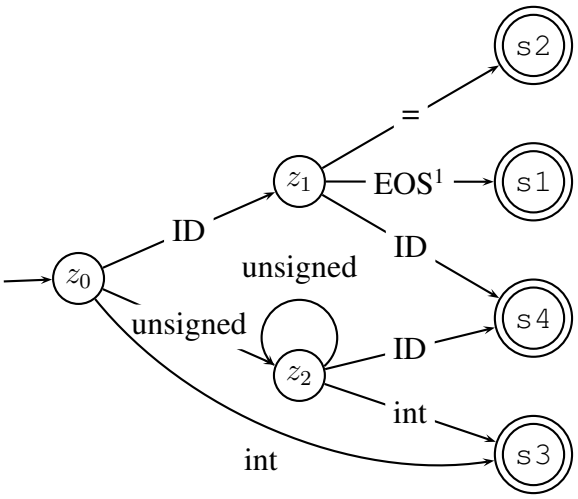
Das folgende Beispiel illustriert die Idee:

```
s : ID                      |  # s1
    ID '=' expr             |  # s2
    'unsigned'* 'int' ID    |  # s3
    'unsigned'* ID ID;         # s4
```

Der *-Operator bedeutet hier, dass beliebig viele, incl. 0 Folgen von 'unsigned' erlaubt sind. `ID` steht für Identifier, und wird vom Lexer zu einem Token verarbeitet. `expr` könnte ein arithmetischer Ausdruck sein, wie im ersten Beispiel. Wörter der Sprache sind z.B. `a` (s1) oder `a = b` (s2) oder `int a` (s3) oder `unsigned unsigned int a` (s3) oder `a b` (s4). Sieht der Parser nur das `a`, dann kann die Regel s1 oder s2 oder s4 zutreffen. Sieht der Parser ein `unsigned`, dann kann die Regel s3

oder s4 zutreffen. Welche es ist, kann man erst entscheiden, wenn man alle `unsigned` übersprungen hat, und das können beliebig viele sein.

Um die richtige Entscheidung zu treffen, erzeugt ein LL(*)-Parser einen deterministischen endlichen Automaten, dessen Endzustände die Information über die richtige Regel enthalten. Im Beispiel würde er folgendermaßen aussehen:



Jeder Endzustand enthält die Information, welche der vier Regeln anwendbar ist.
Z.B. wird `unsigend unsigned int` auf `s3` abgebildet. Wieviele `unsigned` nacheinander kommen ist dabei egal. Daher ist das kein LL(k)-Parsing mit festen k, sondern LL(*) mit flexiblem k. Der Automat wird jedoch nur zur Vorausschau benutzt. Der zu parsende String wird dabei für den eigentlichen Parser nicht verändert.

ANTLR kann jedoch auch mit Fällen umgehen, wo trotz dieser Voranalyse Nichtdeterminismen übrig bleiben. In diesem Fall muss er backtracken, es sei denn, der Programmierer unterstützt die Regelauswahl. Dafür kann er Regeln mit Codestücken definieren.

ANTLR unterscheidet dabei *syntaktische Prädikate*, *semantische Prädikate* und *Mutatoren*. Syntaktische Prädikate erlauben programmierte Tests auf den zu parsenden Reststring. Semantische Prädikate und Mutatoren erlauben Zugriff auf Datenstrukturen der Hostsprache, bei ANTLR ist das Java. Hiermit kann man z.B. während des Parsens Symboltabellen anlegen und abfragen.

Beispiel: Eine Deklaration `int x;` könnte man mit einem Mutator in eine Symboltabelle eintragen, und dann bei einer Zuweisung `x = 3.5` mit einem semantischen Prädikat feststellen, dass das nicht Typ-konform ist. Mit diesen Mechanismen angereichert wird die LL(*)-Sprache sogar *kontextsensitiv*.

Um die genaue Syntax der ANTLR-Grammatik zu lernen sollte man auf die Originaldokumentation zurück greifen. Eine detaillierte Beschreibung des LL(*) Parsings bei ANTLR findet man in [1].

[1] EOS steht für „End Of String“

4 ANTLR benutzen

Die Installationsanweisungen findet man unter
https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

Ist die Installation beendet, kann man ein erstes Beispiel probieren.

In einen File `Arith.g4` schreibt man

```
grammar Arith;

calc : expr          # calculator;

expr: expr OP expr   # Ausdruck
   | '(' expr ')'    # Klammer
   | INT             # int
;
OP : '+'|'-'|'*'|'/';
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Mit dem Kommando

```
antlr4 -visitor Arith.g4
```

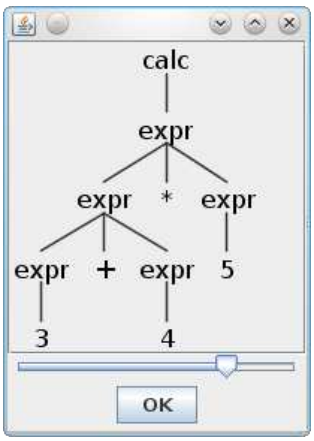
erzeugt man eine Reihe von .java Files, die man mit `javac *.java` compiliert.

antlr4 ist eine Abkürzung:
`alias antlr4='java -jar /usr/local/lib/antlr-4.7-complete.jar'`
oder ähnlich.

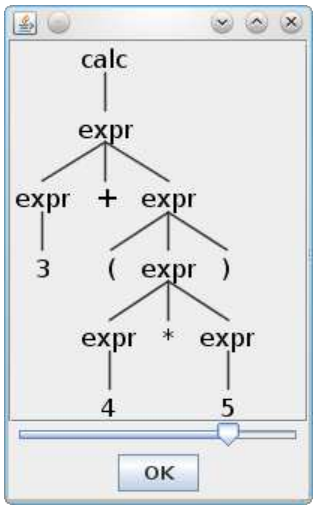
Mit dem Komando
`grun Arith r -gui` kann man jetzt Strings eingeben, die geparst werden.
Dabei ist `grun` die Abkürzung:
`alias grun='java org.antlr.v4.runtime.misc.TestRig'`

Die Ausgabe ist ein Syntaxbaum, wie in folgenden Beispielen:

3 + 4 * 5



3 + (4 * 5)



Der linke Baum illustriert das zweite L in LL(*). `3 + 4 * 5` wird wie `(3 + 4) * 5` geparst, d.h. die *linkeste* Variante wird genommen. Es gibt in ANTLR natürlich auch Möglichkeiten, Operatorpräzedenzen zu definieren. Damit kann man z.B. Punktrechnung vor Strichrechnung erzwingen.

5 Das ANTLR Application Programming Interface (API)

Das ANTLR-API stellt zwei verschiedenen Möglichkeiten zur Verfügung, wie man einen Syntaxbaum bearbeiten kann. Sie basieren auf bekannten Design Patterns, dem Visitor- und dem Listener-Pattern.

5.1 Das Visitor-Pattern

Die arithmetischen Ausdrücke im obigen Beispiel möchte man natürlich auch ausrechnen. Dafür hat ANTLR beim Aufruf von antlr4 eine generische Klasse ArithBaseVisitor erzeugt, die man folgendermaßen vererben kann:

```
public class Calculator extends ArithBaseVisitor<Integer> {
        static int result;
        public Integer visitInt(ArithParser.IntContext ctx) {
                return Integer.valueOf(ctx.INT().getText());}

        public Integer visitKlammer(ArithParser.KlammerContext ctx) {
                return visit(ctx.expr());}

        public Integer visitAusdruck(ArithParser.AusdruckContext ctx) {
                int left = visit(ctx.expr(0));
                int right = visit(ctx.expr(1));
                switch(ctx.OP().getText().charAt(0)) {
                        case '+': result = left + right; break;
                        case '-': result = left - right; break;
                        case '*': result = left * right; break;
                        case '/': result = left / right;}
                return result;}}
```

Für jede Grammatikregel, hinter der ein # gefolgt von einem Wort steht, z.B. `# Ausdruck` programmiert man eine Methode visit..., also visitAusdruck, visitKlammer, visitInt. Die Methoden können auf den Syntaxbaum zugreifen und Berechnungen ausführen.

Das dazugeörige main-Programm könnte so aussehen:

```
public static void main(String[] args) throws Exception {
    CharStream input = CharStreams.fromStream(System.in);
        ArithLexer lexer = new ArithLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ArithParser parser = new ArithParser(tokens);
        ParseTree tree = parser.calc();

        Calculator cal = new Calculator();
        int result = cal.visit(tree);
        System.out.println(result);}
```

Der erste Teil liest die Eingabe, übergibt sie einem neu erzeugten Lexer. Der wandelt sie in einem Token-Stream um, welcher vom Parser geparst wird. Erst dann wird der Calculator aufgerufen, der den erzeugten Syntaxbaum bearbeitet.

5.2 Das Listener-Pattern

Das obige Beispiel demonstriert den Zugriff auf den Syntaxbaum mittels des sog. *Visitor Patterns*. Es gibt noch ein zweites API mittels des sog. *Listener Patterns*

Hierbei definiert man für jede Variable der Grammatik sog. enter- und exit-Methoden. Zusätzlich wird noch eine visitTerminal-Methode definiert, die für jedes Terminalsymbol aufgerufen wird. Der Syntaxbaum wird dann per Tiefensuche durchlaufen, und jeweils die enter- und exit-Methoden aufgerufen.

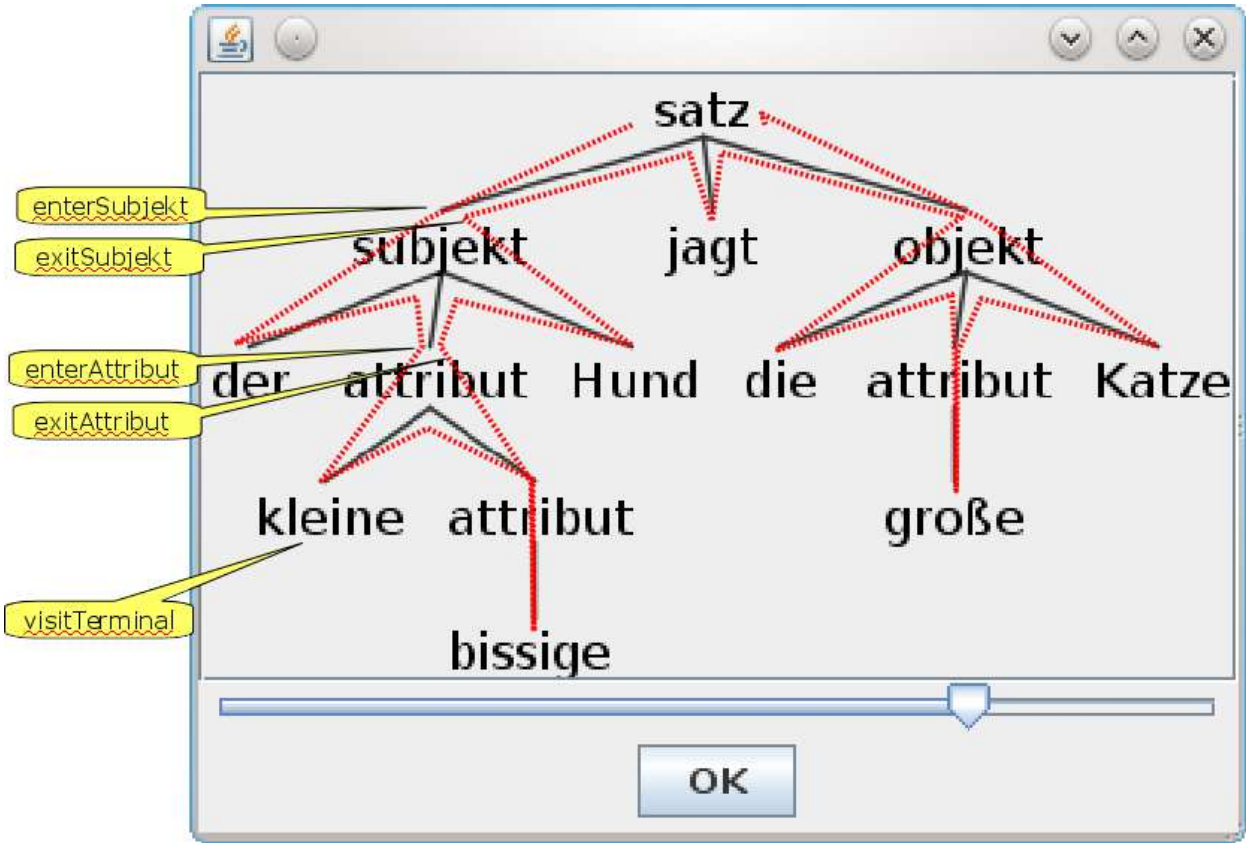
Mit der folgenden Grammatik illustrieren wir den Zugriff auf den Syntaxbaum mittels des Listener-Patterns.

Die Grammatik sei:

```
grammar Hund;

satz:     subjekt Prädikat objekt;
subjekt:  Artikel attribut Substantiv;
objekt:   Artikel attribut Substantiv;
attribut: Adjektiv attribut | Adjektiv;


Artikel:    'der'|'die'|'das';
Adjektiv:   'kleine'|'bissige'|'große';
Substantiv: 'Hund'|'Katze';
Prädikat:   'jagt'|'fängt';

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

Mit ihr kann man z.B. das Wort „der kleine bissige Hund jagt die große Katze“ erzeugen.

Wir programmieren einen *Treewalker*, der den Baum durchläuft, und dabei bestimmte Methoden aufruft.



Die visitTerminal-Methode wird dabei für jedes Terminalsymbol aufgerufen.

Folgende Java-Klasse implementiert damit einen primitiven Übersetzer in Englisch. Er übersetzt Sätze wie „der kleine bissige Hund jagt die große Katze“ Wort für Wort in Englisch.

```
public class Translator extends HundBaseListener {
    static HashMap<String,String> dictionary ß
        = new HashMap<String,String>();
    static {
        dictionary.put("kleine","small");    dictionary.put("große","large")
        dictionary.put("bissige","biting"); dictionary.put("jagt","chases")
        dictionary.put("fängt","catches");

        dictionary.put("Hund","dog");        dictionary.put("Katze","cat");
        dictionary.put("der","the");         dictionary.put("die","the");
        dictionary.put("das","the");}

     @Override
     public void visitTerminal(TerminalNode node)
        {System.out.print(dictionary.get(node.getText())+"␣"); }
```

```
public static void main(String[] args) throws Exception {
        CharStream input = CharStreams.fromStream(System.in);
        HundLexer lexer = new HundLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ParseTreeWalker walker = new ParseTreeWalker();
        HundParser parser = new HundParser(tokens);

        Translator translator = new Translator();
        ParseTree tree = parser.satz();
        walker.walk(translator,tree);
        }}
```

Die einzige Methode, die spezifisch für das Beispiel ist, ist die visitTerminal Methode. Sie wird aufgerufen wenn der Durchlauf durch den Syntaxbaum an den Terminalsymbolen ankommt. Die enter- und exit-Methoden sind für dieses Beispiel nicht notwendig.

Für den Satz
„der kleine bissige Hund jagt die große Katze“ durckt der TreeWalker aus:
„the small biting dog chases the large cat“.

Literatur

[1] Terence Parr and Kathleen S.Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.