ANTLR-Studio zum Entwickeln von ANTLR-Grammatik benutzen

Prashant Deva

ANTLR ist ein Tool, mit dem man Parser, Lexer und Baum-Parser generieren kann. Natürlich kann man diese auch selbst in seiner bevorzugten Programmiersprache schreiben, aber das würde eine sehr lange Zeit in Anspruch nehmen, ganz zu schweigen von der Anzahl der Fehler, die gemacht werden, zusammen mit der Tatsache, dass die Grammatik des Parsers/Lexers sehr schwer zu verstehen wäre.

Daher ist es weitaus angenehmer, eine Grammatik für die zu parsende Sprache zu schreiben und ein Parser-Generierungs-Tool den entsprechenden Code für einen entwickeln zu lassen. Es gibt auch andere Tools, wie Lex, Yacc usw., die Parser/Lexer für einen entwickeln können, doch ANTLR zeichnet sich dadurch aus, dass der generierte Code sehr einfach zu lesen ist und eine fortgeschrittene IDE im ANTLR-Studio hat, die das Schreiben von Grammatik sehr einfach macht.

Das Schreiben von ANTLR-Grammatik wird gewöhnlich als ein langsamer, mühsamer und schwieriger Prozess, der nur von 'echten' Programmierern gemacht wird, betrachtet. Und warum sollte es nicht so sein? Man schaue sich nur mal die ganzen Sachen an, die man 'manuell' vornehmen muss. Als Allererstes gab es für diese Art von Arbeiten keine IDE, also nahm man das gute alte Notepad, wenn man Windows benutzte oder Emacs, wenn man Linux benutze und schrieb seine Grammatik auf. Natürlich gibt es dabei kein Syntax-Highlighting, Auto-Complete oder irgendeinen anderen Schnickschnack, der in den normalen Java/C++-IDEs vorhanden ist. Zu dieser Tatsache

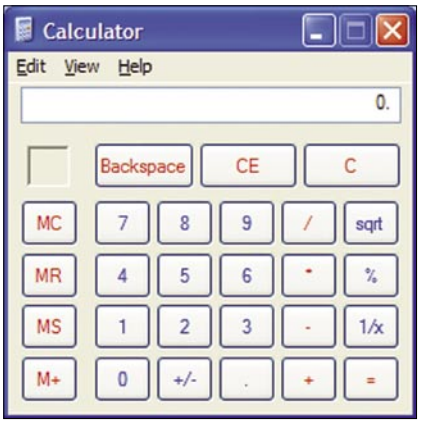

Abbildung 1. *Windows Rechner, obwohl sein Interface zunächst simpel aussieht, kann einem in die Quere kommen, wenn man tatsächlich damit arbeitet*

Über den Autor

Prashant Deva ist der Entwickler von ANTLR-Studio und der Gründer von Placid System.

Kontakt:
pdeva@placidsystems.com

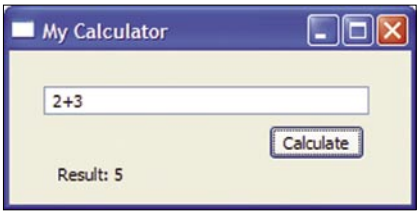

Abbildung 2. *Unser Rechner, einfaches Interface, das es ermöglicht, den ganzen Ausdruck auf einmal zu sehen*

kommt hinzu, dass man die Grammatik-Datei jedes Mal manuell erstellen muss, wenn man einen Fehler macht und es wirklich keine exakte Methode gibt, die Grammatik zu debuggen und man ebenso den Parser/Lexer in der bevorzugten Programmiersprache (z.B. Java/C/C++) erstellen kann, letzten Endes bekommt man alle Vorteile einer IDE, auch wenn man vielleicht einige tausend Zeilen Code extra schreiben muss!

Aber all dieses ändert sich mit ANTLR-Studio, einer IDE für ANTLR, die sich komplett in die Eclipse-Entwicklungsumgebung integriert. ANTLR-Studio wurde mit dem Gedanken an das Problem angelegt, dass Grammatik-Entwicklung, die von

Listing 1. *Der Parser*

```
class CalcParser extends Parser;
options{
      buildAST=true;
}
expr
      : mexpr (PLUS^ mexpr)*
      ;
mexpr
      : a:atom (STAR^ atom)*
      ;
atom: INT
      ;
```

Listing 2. *Der Baum-Parser*

```
expr returns [int r]
{
      int a,b;
      r=0;
}
      : #(PLUS a=expr b=expr)
        {r = a+b;}
      | #(STAR a=expr b=expr)
        {r = a*b;}
      | i:INT   {r =
        Integer.parseInt
        (i.getText());}
      ;
```

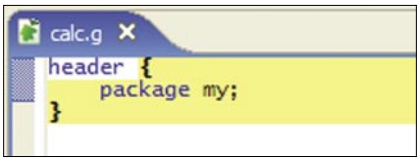

Abbildung 3. *ANTLR-Studio fügt automatisch den Namen des Packages ein, wenn man eine neue Grammatik-Datei erstellt*

der hauptsächlichen Entwicklung getrennt ist, mit sich bringt. So entwickelt es sich zu großen Längen, um eine im höchsten Maße integrierte Praxis zu bieten. Und um dies zu überprüfen werden wir eine einfache Anwendung erstellen, zu der wir ANTLR im ANTLR-Studio verwenden.

Der Rechner

Obwohl der Windows-Rechner auf den ersten Blick sehr friedlich aussieht, sind viele Dinge daran falsch. Als erstes, wenn man irgendwo einen mathematischen Ausdruck geschrieben findet, kann man diese nicht einfach kopieren und in den Rechner einfügen. Man muss den vollständigen Audruck von Hand abtippen. Falls es außerdem ein sehr langer Ausdruck ist, vergisst man manchmal vielleicht einfach, an welcher Stelle man ist, da der Rechner nicht den kompletten Ausdruck anzeigt, sondern nur das letzte Ergebnis des Ausdrucks, den man bisher eingegeben hat. Wenn

Abbildung 4. *Das blaue Puzzlestück-Symbol ruft den Lexer-Assistenten auf*

man also durch einen Fehler eine falsche Zahl eingegeben hat, während man an diesem langen Ausdruck getippt hat, kann man es niemals herausfinden und das Ergebnis wird falsch sein, ohne dass man es überhaupt weiß.

Also werden wir einen neuen Rechner in Eclipse mit dem ANTLR-Studio entwerfen, wie es auf Abbildung 2 gezeigt wird. Man kann in die Textbox einen vollständigen mathematischen Ausdruck eintippen oder kopieren/einfügen und den *Calculate*-Button drücken, um das Ergebnis zu erhalten. Um die Sache für dieses Tutorial einfach zu halten, werden wir nur die Operatoren für Multiplikation und Addition in den Ausdrücken erlauben.

Wenn der *Calculate*-Button geklickt wurde, parst der Rechner den Ausdruck in der Textbox und macht daraus einen Baum. Anschließend geht er den Baum ab, um den Ausdruck zu berechnen. Die Java-Portion dieser Anwendung ist belanglos und irrelevant für unser Tutorial, also werden wir uns nur auf die ANTLR-Grammatik konzentrieren, die den Inhalt des Codes enthält.

Listing 3. *Code für den Parser-Aufruf*

```
try {
            StringReader reader = new StringReader("2+3;");
            CalcLexer lexer = new CalcLexer(reader);
            CalcParser parser = new CalcParser(lexer);
// Den eingegebenen Ausdruck parsen
            parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
// Den resultierenden Baum in LISP-Notation ausgeben
            System.out.println(t.toStringTree());
            CalcTreeWalker walker = new CalcTreeWalker();

// Den vom Parser erzeugten Baum durchlaufen
            int r = walker.expr(t);
            return r;
        }
        catch(TokenStreamException e) {
            System.err.println("exception: "+e);
        }
        catch(RecognitionException e) {
            System.err.println("exception: "+e);
        }
}
```

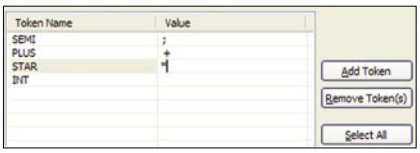

Abbildung 5. *ANTLR-Studio fügt automatisch den Namen der im Parser verwendeten Zeichen ein, man muss nur 'die Lücken ausfüllen'*

Der Parser

Im Package-Explorer macht man einen Rechtsklick auf das Package, in dem man die Grammatik platzieren will und wählt 'New–>ANTLR Grammar' (Neu–>ANTLR-Grammatik) aus. Man gibt den Namen der Grammatik-Datei im sich dann öffnenden Fenster ein und drückt 'Ok'. Eine neue Grammatik-Datei wurde erstellt, die automatisch die Header-Sektion mit der Package-Deklaration enthält. Nun gibt man die folgende Parser-Grammatik in den Editor ein. Man beachte, dass das TypeOnce-Feature des Editors, welches automatisch Vervollständigungen anzeigt, ohne dass man [Strg+Leertaste] drücken muss, die Eingabe der Grammatik sehr schnell macht.

Wenn man die ANTLR-Syntax kennt, sollte dies nach einer sehr einfachen Grammatik aussehen. Die erste Zeile deklariert unseren Parser. Es ist wie das Deklarieren einer Klasse in Java. Tatsächlich wird dies in eine Klasse mit dem selben Namen (CalcParser) entwickelt werden, wenn ANTLR diese Grammatik kompiliert. Als nächstes deklarieren wir die Optionen für diese Regel, die ANTLR anweist, einen Abstrakten Syntax-Baum oder AST zu erstellen, indem es den Parser benutzt.

Die nächsten paar Zeilen definieren eine expr-Regel, die eine mexpr enthält, gefolgt von 0 oder mehreren +Zeichen und noch eine mexpr, gefolgt von einem Semikolon. Das *-Symbol steht für 0 oder mehr. Ebenso definiert die mexpr-Regel ein atom. Ein atom ist einfach definiert, ein INT-Zeichen zu sein, das für irgendein Integer steht. Wir werden diese Zeichen später in unserem Lexer definieren.

Falls sie sich wundern, das komisch aussehende ^-Symbol in zwei der Regeln wird für die Baum-Konstruktion verwendet. Es zeigt dem ANTLR an, dass das Zeichen als Wurzel des Baumes verwendet wird. Zum Beispiel wird der, der Baum, der für die Regel expr gestaltet ist, PLUS als seine Wurzel haben, mit mexpr als Kind.

Abbildung 6. *Man muss nicht länger manuell komplexe Regeln zur Definition von Identifikatoren schreiben. Der Lexer-Assistent erlaubt sogar, den Identifikator auf der Stelle zu testen!*

Der Lexer

Nun, anstatt den Lexer selbst zu tippen, werden wir ANTLR-Studios guten Lexer-Assistenten verwenden, um dies für uns zu erledigen. Man klickt auf das blaue Puzzle-Symbol auf der oberen rechten Ecke der Symbolleiste. Auf der ersten Seite muss man den Namen des Lexers eintippen, nennen wir ihn CalcLexer. Weiter geht es mit Next. Man kann sehen, dass ANTLR-Studio bereits die Namen der Zeichen, die man im Parser verwendet hat, eingetragen hat. Man muss nur festlegen, wofür sie stehen. Zum Beispiel gibt man ; neben SEMI, + neben PLUS und * neben STERN ein. Man entfernt das INT-Zeichen.

Wir werden damit auf der nächsten Seite arbeiten. Auf Next klicken. Auf der *Add Identifier*-Seite klickt man auf Add. Nun gibt man INT als Zeichennamen ein. Man wählt *One or more* (eins oder mehr) und *Any Digit* (jede Zahl) aus. Damit haben wir einen Identifikator definiert, der Ints erkennt. Man kann sogar eine Nummer in die Box darunter eingeben, um den Identifikator zu testen.

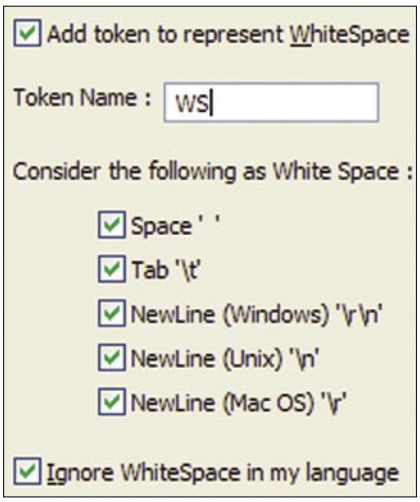

Abbildung 7. *Der Lexer-Assistent macht die Handhabung von nicht-druckbaren Zeichen (Whitespace) so einfach wie abhaken einer Checkbox!*

Auf Next klicken und die *Add token to represent WhiteSpace*-Checkbox (Hinzufügen eines Platzhalters für Nicht-druckbare Zeichen) markieren. WS als Zeichennamen eingeben und *Ignore WhiteSpace in my language* (Ignoriere nicht-druckbare Zeichen in meiner Sprache) auswählen.

Da wir keine Kommentare oder Zeichenketten in unserer Sprache bearbeiten müssen, kann man einfach weiter machen und in den folgenden beiden Schritten auf Next drücken.

Und unser Lexer ist komplett, ohne dass man ein einziges Wort im Editor tippen musste! Das war einfach, oder?

Der Baum-Parser

Nun werden wir unseren Baum-Parser eingeben, der den Ergebnisbaum abgeht und die aktuelle Kalkulation durchführt. Man gibt dazu folgendes im Editor ein.

Der Baum-Parser braucht nur eine Regel. Er gibt eine int r zurück, die das Ergebnis der Kalkulation enthält. Erst definieren wir 2 ints, a & b. Der Rest der Regel definiert, wie der Baum abgearbeitet wird und die Berechnungen durchgeführt werden. Zum Beispiel teilt die erste Alternative dort dem Baum-Abarbeiter mit, dass der Baum das PLUS-Symbol gefolgt von zwei Ausdrücken enthält. Wir weisen das Ergebnis beider exprs jeweils den Variablen a und b zu und setzen das Ergebnis in r ein, indem wir sie addieren. Es ist zu beachten, dass alles in den {}-Klammern für Java-Code steht. ANTLR wird alles zwischen diesen Klammern genauso, wie es dort steht, in die Methoden einfügen, die es für jede Regel generiert.

Der Java-Code

Und unsere Grammatik ist vollständig! Nun werden wir den folgenden Code schreiben, der den Parser aufruft und die Berechnung im Event-Handler für den Calculate-Button (Berechnen) durchführt.

Also, da haben wir nun einen Rechner, der viel freundlicher ist als viele andere Rechner. Versucht nun selbst, die Unterstützung für mathematische Operationen hinzuzufügen. ■