Parser mit ANTLR generieren

Carsten Gips (HSBI)

TL;DR
Mit ANTLR kann aus einer Grammatik ein LL(*)-Parser generiert werden. Die Parser-Regeln in der Grammatik fangen dabei mit einem **Kleinbuchstaben** an (Erinnerung: Lexer-Regel starten mit einem Großbuchstaben).
Regeln haben einen Namen (linke Seite) und eine Produktion (rechte Seite). Dabei können beliebige Abfolgen von Lexer- und Parser-Regeln auf der rechten Seite einer Parser-Regel auftauchen. Die Token müssen jeweils matchen, die Parser-Regeln werden in einen Aufruf der jeweiligen generierten Funktion übersetzt.
Parser-Regeln können aus mehreren Alternativen bestehen, diese werden per | separiert. Dabei hat bei Mehrdeutigkeiten die erste passende Alternative Vorrang. Wie bei Lexer-Regeln können Teile per ? ein- oder keinmal vorkommen, per * beliebig oft oder per + ein- oder mehrfach.
ANTLR erlaubt im Gegensatz zu allgemeinen LL-Parsern direkte Links-Rekursion. (Indirekte Links-Rekursion funktioniert allerdings nicht.)
Der von ANTLR generierte Parser erzeugt auf der Eingabe einen Parse-Tree, der die Strukturen der Grammatik widerspiegelt: Die Token bilden die Blätter und jede erfolgreich durchlaufene Parser-Regel bildet einen entsprechenden Knoten im Baum.
Für die Traversierung des Parse-Tree kann man die generierten Listener- oder Visitor-Klassen nutzen. Beim Einsatz der Listener nutzt man die vorgegebene Klasse `ParseTreeWalker`, die mit dem Parse-Tree und dem Listener den Baum per Tiefensuche traversiert und immer die jeweiligen `enterRegel`- und `exitRegel`-Methoden aufruft. Beim Visitor muss die Traversierung selbst erledigt werden, hier steht die aus der Klassenhierarchie geerbte Methode `visit` als Startpunkt zur Verfügung. In dieser Methode wird basierend auf dem Knotentyp die in den Visitor-Klassen implementierte `visitRegel`-Methode aufgerufen und man muss darauf achten, die Kindknoten durch passende Aufrufe zu traversieren. Sowohl bei den generierten Listener- als auch den Visitor-Klassen kann man die leeren Defaultmethoden bei Bedarf selbst überschreiben. Für den Zugriff auf die Regel-Elemente werden die sogenannten Kontextobjekte als Parameter übergeben.
Benannte Alternativen und Regel-Elemente sind nützlich, weil für die benannten Alternativen zusätzliche Kontextklassen erzeugt werden, über die dann auf die Bestandteile der Alternativen zugegriffen werden kann. Außerdem werden zusätzlich passende `enterAlternative`- und `exitAlternative`- bzw. `visitAlternative`-Methoden generiert. Für benannte Regel-Elemente wird ein entsprechend benanntes Attribut im Kontextobjekt angelegt, welches **`public`** sichtbar ist.

Videos

- VL Parser mit ANTLR (YT)
- Demo ANTLR Parser (YT)
- VL Parser mit ANTLR (HSBI)

1 Hello World

```
grammar Hello;

start : stmt* ;

stmt  : ID '=' expr ';' | expr ';' ;

expr  : term ('+' term)* ;
term  : atom ('*' atom)* ;

```

```
10  atom   : ID | NUM ;
11
12  ID     : [a-z][a-zA-Z]* ;
13  NUM    : [0-9]+ ;
14  WS     : [ \t\n]+ -> skip ;
```

Konsole: Hello (grun, Parse-Tree)

1.1 Starten des Parsers

1. Grammatik übersetzen und Code generieren: `antlr Hello.g4`
2. Java-Code kompilieren: `javac *.java`
3. Parser ausführen:
   - `grun Hello start -tree` oder `grun Hello start -gui` (Grammatik "Hello", Startregel "start")
   - Alternativ mit kleinem Java-Programm:

```
1  import org.antlr.v4.runtime.CharStreams;
2  import org.antlr.v4.runtime.CommonTokenStream;
3  import org.antlr.v4.runtime.tree.ParseTree;
4
5  public class Main {
6      public static void main(String[] args) throws Exception {
7          HelloLexer lexer = new HelloLexer(CharStreams.fromStream(System
               .in));
8          CommonTokenStream tokens = new CommonTokenStream(lexer);
9          HelloParser parser = new HelloParser(tokens);
10
11         ParseTree tree = parser.start();  // Start-Regel
12         System.out.println(tree.toStringTree(parser));
13     }
14 }
```

1.2 Startregeln

- `start` ist eine Parser-Regel => Eine Parser-Regel pro Grammatik wird benötigt, damit man den generierten Parser am Ende auch starten kann …
- Alle Regeln mit kleinem Anfangsbuchstaben sind Parser-Regeln
- Alle Regeln mit großem Anfangsbuchstaben sind Lexer-Regeln

1.3 Formen der Subregeln

```
1  stmt   : ID '=' expr ';' ;
```

Um die Regel `stmt` anwenden zu können, müssen alle Elemente auf der rechten Seite der Regel erfüllt werden. Dabei müssen die Token wie `ID`, = und `;` matchen und die Subregel `expr` muss erfüllt werden können. Beachten Sie das abschließende Semikolon am Ende einer ANTLR-Regel!

```
1  stmt  : ID '=' expr ';' | expr ';' ;
```

Alternativen werden durch ein | getrennt. Hier muss genau eine Alternative erfüllt werden. Falls nötig, trennt man die Alternativen durch Einschließung in runden Klammern vom Rest der Regel ab: `r : a (b | c)d ;`.

```
1  expr  : term ('+' term)* ;
```

Der durch den * gekennzeichnete Teil kann beliebig oft vorkommen oder auch fehlen. Bei einem + müsste der Teil mind. einmal vorkommen und bei einem ? entsprechend einmal oder keinmal.

Auch hier kann man die Operatoren durch ein zusätzliches ? auf non-greedy umschalten (analog zu den Lexer-Regeln).

(vgl. github.com/antlr/antlr4/blob/master/doc/parser-rules.md)

1.4 Reihenfolge in Grammatik definiert Priorität

Falls mehr als eine Parser-Regel die selbe Input-Sequenz matcht, löst ANTLR diese Mehrdeutigkeit auf, indem es die erste Alternative nimmt, die an der Entscheidung beteiligt ist.

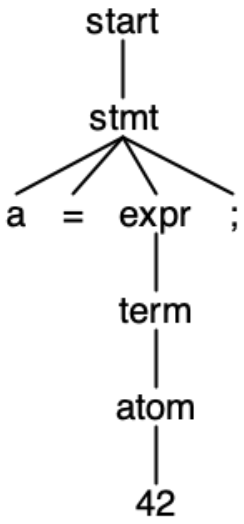
```
1  start : stmt ;
2
3  stmt  : expr | ID  ;
4  expr  : ID   | NUM ;
```

Bei der Eingabe “foo” würde die Alternative `ID` in der Regel `expr` “gewinnen”, weil sie in der Grammatik vor der Alternative `ID` in der Regel `stmt` kommt und damit Vorrang hat.

1.5 Parse-Tree

Betrachten wir erneut die obige Grammatik.

Die Eingabe von “`a = 42;`” führt zu folgendem Parse-Tree:



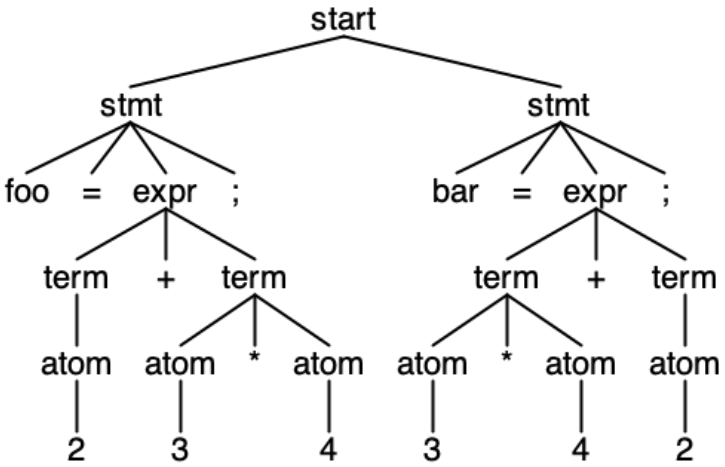
Diese Eingabe führt zur Erkennung der Token `[ID, WS, =, WS, NUM, ;]`, wobei die `WS`-Token verworfen werden und der Parser den Tokenstream `[ID, =, NUM, ;]` erhält.

Die Startregel hat auf der rechten Seite kein oder mehrere `stmt`-Regeln. Die `stmt`-Regel fordert auf der rechten Seite entweder die Token `ID`und = sowie die Regel `expr` gefolgt vom Token `;`, oder die Regel `expr` gefolgt vom Token `;`. In unserem Beispiel kann für das “a” das Token `ID` produziert werden, das “=” matcht ebenfalls. Die “42” wird erklärt, indem für `expr` ein `term` und dort ein `atom` aufgerufen wird. Für das `atom` muss entweder ein Token `ID` oder `NUM` als nächstes Token kommen - hier wird die “42” wird als Token `NUM` verarbeitet. Da die weiteren Regelteile in `term` und `expr` optional sind, haben wir damit ein `expr` erfüllt und das nachfolgende `;`-Token schließt die erste Alternative der Regel `stmt` erfolgreich ab.

Im entstehenden Parse-Tree sind diese Abläufe und grammatikalischen Strukturen direkt erkennbar. Jede erfolgreich durchlaufene Parserregel wird zu einem Knoten im Parse-Tree. Die Token werden als Terminale (Blätter) in den Baum eingehängt.

Anmerkung: Der Parse-Tree ist das Ergebnis der Parsers-Phase im Compiler und dient damit als Input für die folgenden Compilerstufen. In der Regel benötigt man die oft recht komplexen Strukturen aber später nicht mehr und vereinfacht den Baum zu einem *Abstract Syntax Tree* (AST). Im Beispiel könnte man den Zweig `stmt - expr - term - atom - 42` zu `stmt - 42` vereinfachen.

Betrachten wir nun die Eingabe `foo = 2+3*4; bar = 3*4+2;`. Diese führt zu folgendem Parse-Tree:



Wie man sehen kann, sind in der Grammatik die üblichen Vorrangregeln für die Operationen + und * berücksichtigt - die Multiplikation wird in beiden Fällen korrekt "unter" der Addition im Baum eingehängt.

1.6 To EOF not to EOF?

Startregeln müssen nicht unbedingt den gesamten Input "konsumieren". Sie müssen per Default nur eine der Alternativen in der Startregel erfüllen.

Betrachten wir noch einmal einen leicht modifizierten Ausschnitt aus der obigen Grammatik:

```
1  start : stmt ;
```

Die Startregel wurde so geändert, dass sie nur noch genau ein Statement akzeptieren soll.

In diesem Fall würde die Startregel bei der Eingabe "aa; bb;" nur den ersten Teil "aa;" konsumieren (als Token `ID`) und das folgende "bb;" ignorieren. Das wäre in diesem Fall aber auch kein Fehler.

Wenn der gesamte Eingabestrom durch die Startregel erklärt werden soll, dann muss das vordefinierte Token `EOF` am Ende der Startregel eingesetzt werden:

```
1  start : stmt EOF;
```

Hier würde die Eingabe "aa; bb;" zu einem Fehler führen, da nur der Teil "aa;" durch die Startregel abgedeckt ist (Token `ID`), und der Rest "bb;" zwar sogar ein gültiges Token wären (ebenfalls `ID` und `;`), aber eben nicht mehr von der Startregel akzeptiert. Durch das `EOF` soll die Startregel aber den gesamten Input konsumieren und erklären, was hier nicht geht und entsprechend zum Fehler führt.

(vgl. github.com/antlr/antlr4/blob/master/doc/parser-rules.md)

2 Expressions und Vorrang (Operatoren)

Betrachten wir noch einmal den Ausschnitt für die Ausdrücke (*Expressions*) in der obigen Beispielgrammatik:

```
1 expr  : term ('+' term)* ;
2 term  : atom ('*' atom)* ;
3 atom  : ID ;
```

Diese typische, etwas komplex anmutende Struktur soll sicher stellen, dass die Vorrangregeln für Addition und Multiplikation korrekt beachtet werden, d.h. dass `2+3*4` als `2+(3*4)` geparst wird und nicht fälschlicherweise als `(2+3)*4` erkannt wird.

Zusätzlich muss bei LL-Parsern Links-Rekursion vermieden werden: Die Parser-Regeln werden in Funktionsaufrufe übersetzt, d.h. bei einer Links-Rekursion würde man die selbe Regel immer wieder aufrufen, ohne ein Token aus dem Token-Strom zu entnehmen.

ANTLR (ab Version 4) kann mit beiden Aspekten automatisch umgehen:

- ANTLR kann direkte Linksrekursion automatisch auflösen. Die Regel `r : r T U | V ;` kann also in ANTLR verarbeitet werden.
- ANTLR besitzt einen Mechanismus zur Auflösung von Mehrdeutigkeiten. Wie oben geschrieben, wird bei der Anwendbarkeit von mehreren Alternativen die erste Alternative genutzt.

Damit lässt sich die typische Struktur für Expression-Grammatiken deutlich lesbarer gestalten:

```
1 expr  : expr '*' expr
2       | expr '+' expr
3       | ID
4       ;
```

Die Regel `expr` ist links-rekursiv, was normalerweise bei LL-Parsern problematisch ist. ANTLR löst diese Links-Rekursion automatisch auf (vgl. github.com/antlr/antlr4/blob/master/doc/left-recursion.md).

Da bei Mehrdeutigkeit in der Grammatik, also bei der Anwendbarkeit mehrerer Alternativen stets die erste Alternative genommen wird, lassen sich die Vorrangregeln durch die Reihenfolge der Alternativen in der `expr`-Regel implementieren: Die Multiplikation hat Vorrang von der Addition, und diese hat wiederum Vorrang von einer einfachen `ID`.

2.1 Direkte vs. indirekte Links-Rekursion

ANTLR kann nur *direkte* Links-Rekursion auflösen. Regeln wie `r : r T U | V ;` stellen in ANTLR also kein Problem dar.

Indirekte Links-Rekursion erkennt ANTLR dagegen nicht:

```
1 r : s T U | V ;
2 s : r W X ;
```

Hier würden sich die Regeln `r` und `s` gegenseitig aufrufen und kein Token aus dem Tokenstrom entfernen, so dass der generierte LL-Parser hier in einer Endlosschleife stecken bleiben würde. Mit indirekter Links-Rekursion kann ANTLR nicht umgehen.

2.2 Konflikte in Regeln

Wenn mehrere Alternativen einer Regel anwendbar sind, entscheidet sich ANTLR für die erste Alternative.

Wenn sich mehrere Tokenregeln überlappen, "gewinnt" auch hier die zuerst definierte Regel.

```
1  def : 'func' ID '(' ')' block ;
2
3  FOR : 'for' ;
4  ID  : [a-z][a-zA-Z]* ;
```

Hier werden ein implizites Token `'func'` sowie die expliziten Token `FOR` und `ID` definiert. Dabei sind die Lexeme für `'func'` und `FOR` auch in `ID` enthalten. Dennoch werden `'func'` und `FOR` erkannt und nicht über `ID` gematcht, weil sie *vor* der Regel `ID` definiert sind.

Tatsächlich sortiert ANTLR die Regeln intern um, so dass alle Parser-Regeln *vor* den Lexer-Regeln definiert sind. Die impliziten Token werden dabei noch vor den expliziten Token-Regeln angeordnet. Im obigen Beispiel hat also `'func'` eine höhere Priorität als `FOR`, und `FOR` hat eine höhere Priorität als `ID`. Aus diesem Grund gibt es die Konvention, die Parser-Regeln in der Grammatik vor den Lexer-Regeln zu definieren - dies entspricht quasi der Anordnung, die ANTLR bei der Verarbeitung sowieso erzeugen würde.

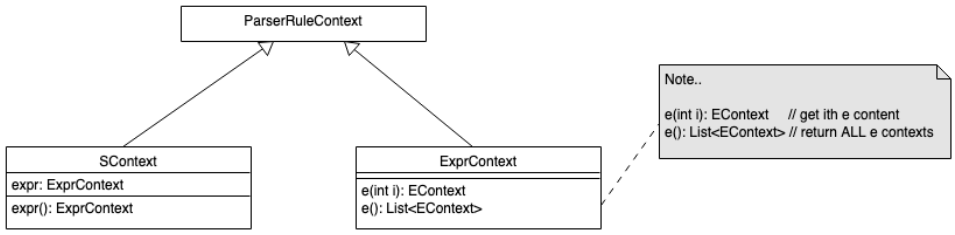
Aus diesem Grund würde auch eine Umsortierung der obigen Grammatik funktionieren:

```
1  FOR : 'for' ;
2  ID  : [a-z][a-zA-Z]* ;
3
4  def : 'func' ID '(' ')' block ;
```

Intern würde ANTLR die Parser-Regel `def` wieder vor den beiden Lexer-Regeln anordnen, und zwischen den Parser-Regeln und den Lexer-Regeln die impliziten Token (hier `'func'`).

3 Kontext-Objekte für Parser-Regeln

```
1  s    : expr         {List<EContext> x = $expr.ctx.e();}
2       ;
3  expr : e '*' e ;
```



Jede Regel liefert ein passend zu dieser Regel generiertes Kontext-Objekt zurück. Darüber kann man das/die Kontextobjekt(e) der Sub-Regeln abfragen.

Die Regel `s` liefert entsprechend ein `SContext`-Objekt und die Regel `expr` liefert ein `ExprContext`-Objekt zurück.

In der Aktion fragt man das Kontextobjekt über `ctx` ab, in den Listener- und Visitor-Methoden erhält man die Kontextobjekte als Parameter.

Für einfache Regel-Aufrufe liefert die parameterlose Methode nur ein einziges Kontextobjekt (statt einer Liste) zurück.

Anmerkung: ANTLR generiert nur dann *Felder* für die Regel-Elemente im Kontextobjekt, wenn diese in irgendeiner Form referenziert werden. Dies kann beispielsweise durch Benennung (Definition eines Labels, siehe nächste Folie) oder durch Nutzung in einer Aktion (siehe obiges Beispiel) geschehen.

4 Benannte Regel-Elemente oder Alternativen

```
stat  : 'return' value=e ';'    # Return
      | 'break' ';'             # Break
      ;
```

```
public static class StatContext extends ParserRuleContext { ... }
public static class ReturnContext extends StatContext {
    public EContext value;
    public EContext e() { ... }
}
public static class BreakContext extends StatContext { ... }
```

Mit `value=e` wird der Aufruf der Regel `e` mit dem Label `value` belegt, d.h. man kann mit `$e.text` oder `$value.text` auf das `text`-Attribut von `e` zugreifen. Falls es in einer Produktion mehrere Aufrufe einer anderen Regel gibt, **muss** man für den Zugriff auf die Attribute eindeutige Label vergeben.

Analog wird für die beiden Alternativen je ein eigener Kontext erzeugt.

5 Arbeiten mit ANTLR-Listeners

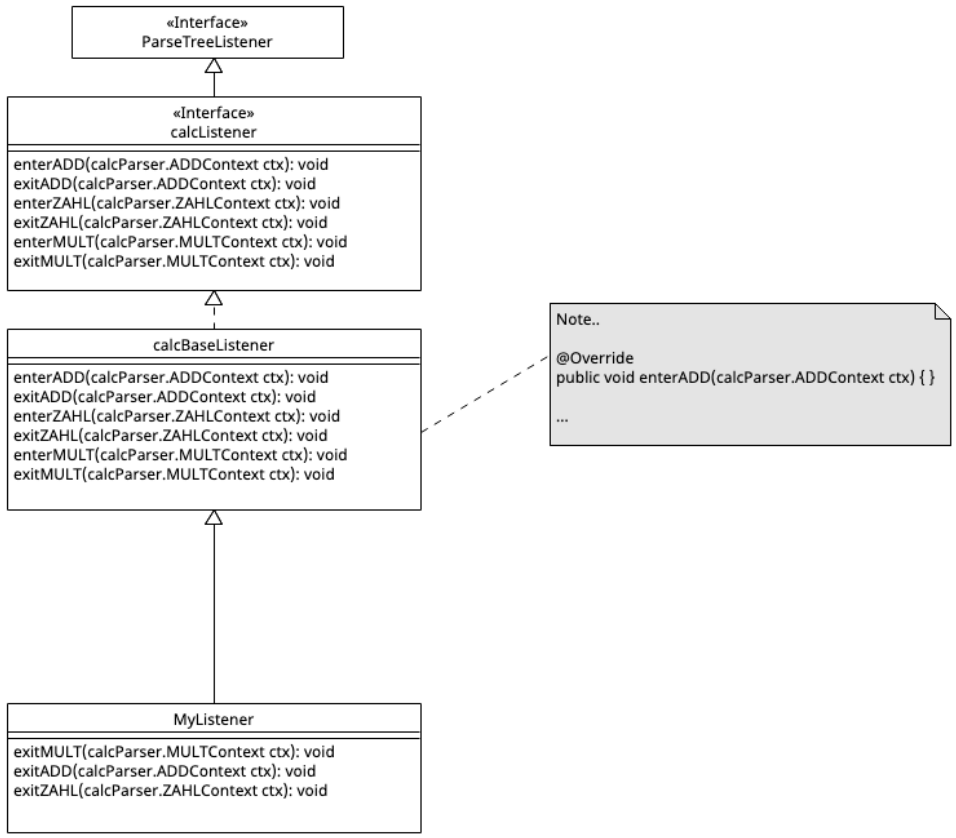
ANTLR (generiert auf Wunsch) zur Grammatik passende Listener (Interface und leere Basisimplementierung). Beim Traversieren mit dem Default-`ParseTreeWalker` wird der Parse-Tree mit Tiefensuche abgelaufen und

jeweils beim Eintritt in bzw. beim Austritt aus einen/m Knoten der passende Listener mit dem passenden Kontext-Objekt aufgerufen.

Damit kann man die Grammatik “für sich” halten, d.h. unabhängig von einer konkreten Zielsprache und die Aktionen über die Listener (oder Visitors, s.u.) ausführen.

```
1  expr : e1=expr '*' e2=expr      # MULT
2       | e1=expr '+' e2=expr      # ADD
3       | DIGIT                    # ZAHL
4       ;
```

ANTLR kann zu dieser Grammatik `calc.g4` einen passenden Listener (Interface `calcListener`) generieren (Option `-listener` beim Aufruf von `antlr`). Weiterhin generiert ANTLR eine leere Basisimplementierung (Klasse `calcBaseListener`):



(Nur “interessante” Methoden gezeigt.)

Von dieser Basisklasse leitet man einen eigenen Listener ab und implementiert die Methoden, die man benötigt.

```
1  public static class MyListener extends calcBaseListener {
2      public void exitMULT(calcParser.MULTContext ctx) {
3          ...
4      }
5      public void exitADD(calcParser.ADDContext ctx) {
```

```
6           ...
7       }
8       public void exitZAHL(calcParser.ZAHLContext ctx) {
9           ...
10      }
11  }
```

Anschließend baut man das alles in eine Traversierung des Parse-Trees ein:

```
1   public class TestMyListener {
2       public static class MyListener extends calcBaseListener {
3           ...
4       }
5
6       public static void main(String[] args) throws Exception {
7           calcLexer lexer = new calcLexer(CharStreams.fromStream(System.in));
8           CommonTokenStream tokens = new CommonTokenStream(lexer);
9           calcParser parser = new calcParser(tokens);
10
11          ParseTree tree = parser.s();    // Start-Regel
12
13          ParseTreeWalker walker = new ParseTreeWalker();
14          MyListener eval = new MyListener();
15          walker.walk(eval, tree);
16      }
17  }
```

Beispiel: TestMyListener.java und calc.g4

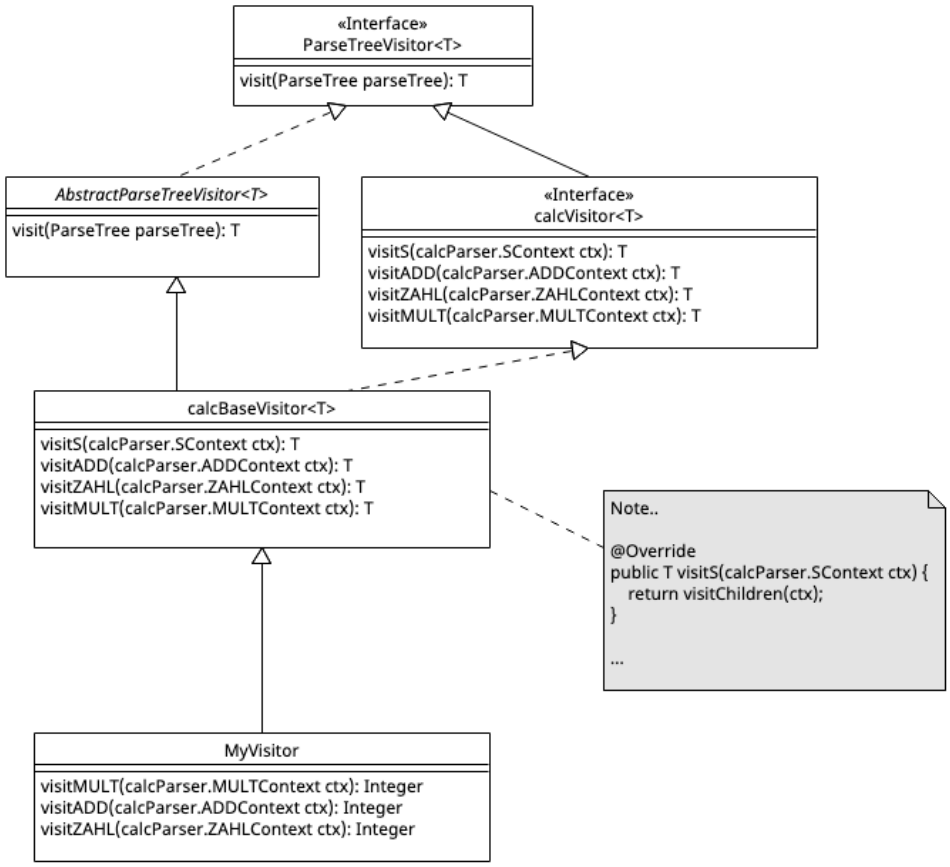
6 Arbeiten mit dem Visitor-Pattern

ANTLR (generiert ebenfalls auf Wunsch) zur Grammatik passende Visitoren (Interface und leere Basisimplementierung).

Hier muss man im Gegensatz zu den Listeners allerdings selbst für eine geeignete Traversierung des Parse-Trees sorgen. Dafür hat man mehr Freiheiten im Vergleich zum Einsatz von Listeners, insbesondere im Hinblick auf Rückgabewerte.

```
1   expr : e1=expr '*' e2=expr      # MULT
2        | e1=expr '+' e2=expr      # ADD
3        | DIGIT                    # ZAHL
4        ;
```

ANTLR kann zu dieser Grammatik einen passenden Visitor (Interface `calcVisitor<T>`) generieren (Option `-visitor` beim Aufruf von `antlr`). Weiterhin generiert ANTLR eine leere Basisimplementierung (Klasse `calcBaseVisitor<T>`):



(Nur “interessante” Methoden gezeigt.)

Von dieser Basisklasse leitet man einen eigenen Visitor ab und überschreibt die Methoden, die man benötigt. Wichtig ist, dass man selbst für das “Besuchen” der Kindknoten sorgen muss (rekursiver Aufruf der geerbten Methode `visit()`).

```
1  public static class MyVisitor extends calcBaseVisitor<Integer> {
2      public Integer visitMULT(calcParser.MULTContext ctx) {
3          return ...
4      }
5      public Integer visitADD(calcParser.ADDContext ctx) {
6          return ...
7      }
8      public Integer visitZAHL(calcParser.ZAHLContext ctx) {
9          return ...
10     }
11 }
```

Anschließend baut man das alles in eine manuelle Traversierung des Parse-Trees ein:

```
1  public class TestMyVisitor {
2      public static class MyVisitor extends calcBaseVisitor<Integer> {
3          ...
4      }
5
```

```
6       public static void main(String[] args) throws Exception {
7           calcLexer lexer = new calcLexer(CharStreams.fromStream(System.in));
8           CommonTokenStream tokens = new CommonTokenStream(lexer);
9           calcParser parser = new calcParser(tokens);
10
11          ParseTree tree = parser.s();    // Start-Regel
12
13          MyVisitor eval = new MyVisitor();
14          eval.visit(tree);
15      }
16  }
```

Beispiel: TestMyVisitor.java und calc.g4

7 Eingebettete Aktionen und Attribute

```
1  s    : expr                      {System.err.println($expr.v);}
2       ;
3
4  expr returns [int v]
5       : e1=expr '*' e2=expr       {$v = $e1.v * $e2.v;}
6       ;
```

Auch die Parser-Regeln können mit eingebetteten Aktionen ergänzt werden, die in die (für die jeweilige Regel) generierte Methode eingefügt werden und bei erfolgreicher Anwendung der Parser-Regel ausgeführt werden.

Über `returns [int v]` fügt man der Regel `expr` ein Attribut `v` (Integer) hinzu, welches man im jeweiligen Kontext abfragen bzw. setzen kann (agiert als Rückgabewert der generierten Methode). Auf diesen Wert kann in den Aktionen mit `$v` zugegriffen werden.

Anmerkung: Durch den Einsatz von eingebetteten Aktionen und Attributen wird die Grammatik abhängig von der Zielsprache des generierten Lexers/Parsers!

8 Ausblick

Damit haben wir die sprichwörtliche “Spitze des Eisbergs” gesehen. Mit ANTLR sind noch viele weitere Dinge möglich. Bitte nutzen Sie aktiv die Dokumentation auf github.com/antlr/antlr4.

9 Wrap-Up

Parser mit ANTLR generieren: Parser-Regeln werden mit **Kleinbuchstaben** geschrieben

- Regeln können Lexer- und Parser-Regeln “aufrufen”
- Regeln können Alternativen haben
- Bei Mehrdeutigkeit: Vorrang für erste Alternative

- ANTLR erlaubt direkte Links-Rekursion
- ANTLR erzeugt Parse-Tree
- Benannte Alternativen und Regel-Elemente
- Traversierung des Parse-Tree: Listener oder Visitoren, Zugriff auf Kontextobjekte

10 Zum Nachlesen

- Parr (2014)

Lernziele

- k2: Aufbau der Parser-Regeln
- k3: Alternativen und optionale/mehrfache Regelteile in Parser-Regeln
- k3: Vorrang von Alternativen (bei Mehrdeutigkeiten)
- k3: Benannte Alternativen und Regel-Elemente
- k2: Aufbau des Parse-Tree
- k3: Umgang mit Kontext-Objekten
- k3: Traversierung des Parse-Tree mit den generierten Listenern oder Visitors

Challenges

Lexer und Parser mit ANTLR: Programmiersprache Lox

Betrachten Sie folgenden Code-Schnipsel in der Sprache "Lox":

```
fun fib(x) {
    if (x == 0) {
        return 0;
    } else {
        if (x == 1) {
            return 1;
        } else {
            fib(x - 1) + fib(x - 2);
        }
    }
}

var wuppie = fib(4);
```

Erstellen Sie für diese fiktive Sprache einen Lexer+Parser mit ANTLR. Implementieren Sie mit Hilfe des Parse-Trees und der Listener oder Visitoren einen einfachen Pretty-Printer.
(Die genauere Sprachdefinition finden Sie bei Bedarf unter craftinginterpreters.com/the-lox-language.html.)

Quellen

Parr, T. 2014. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf. https://learning.oreilly.com/library/view/the-definitive-antlr/9781941222621/.

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

> **Last modified:** f7ac9d2 (reformat using shorter lines, 2025-08-09)