1 Lexer/Parser-Generierung mit ANTLR

1.1 Einführung und Grundlagen

ANTLR (Another Tool for Language Recognition) ist ein Generator für Lexer und Parser. In der aktuellen Version 4 setzt es auf einen adaptiven $LL(*)$-Algorithmus (auch $ALL(*)$ genannt).

ANTLR v4 Eigenschaften

- **Adaptive $LL(*)$:** Die Grammatik wird zur Laufzeit analysiert. Der Parser nutzt Heuristiken, um Konflikte und Mehrdeutigkeiten automatisch aufzulösen.
- **Direkte Linksrekursion:** Wird im Gegensatz zu klassischen LL-Parsern unterstützt (wird intern transformiert). Indirekte Linksrekursion ist weiterhin nicht erlaubt.
- **Trennung von Grammatik und Code:** Semantische Aktionen (eingebetteter Java-Code) werden vermieden. Stattdessen werden Parse-Trees automatisch erstellt und über *Listener* oder *Visitor* traversiert.

1.2 Struktur einer Grammatik

Eine ANTLR-Grammatikdatei (Endung '.g4') definiert Lexer- und Parser-Regeln.

- **Parser-Regeln:** Beginnen mit einem **Kleinbuchstaben** (z.B. `stat`, `expr`). Sie definieren die syntaktische Struktur.
- **Lexer-Regeln (Token):** Beginnen mit einem **Großbuchstaben** (z.B. `ID`, `INT`). Sie definieren die lexikalischen Einheiten (Zeichenfolgen).

1.2.1 Operatoren und Syntax

Die Notation orientiert sich an EBNF (Extended Backus-Naur Form).

Operator	Bedeutung
`x y`	Konkatenation (x gefolgt von y)
`x | y`	Alternative (x oder y)
`x*`	0 oder mehr Wiederholungen (Kleene-Stern)
`x+`	1 oder mehr Wiederholungen
`x?`	0 oder 1 Mal (optional)
`~[a-z]`	Negation (alle Zeichen außer a-z)
`.*?`	**Non-Greedy Match**: Matcht so wenig Zeichen wie möglich (wichtig für Kommentare!)

Non-Greedy Beispiel

Um beispielsweise '/* ... */' Kommentare korrekt zu parsen, darf der '*'-Operator nicht gierig (greedy) sein, da er sonst bis zum allerletzten '*/' der Datei lesen würde.
Lösung: `'/*' .*? '*/'` (stoppt beim ersten Vorkommen von '*/').

1.3 Generierte Artefakte und Parse-Trees

ANTLR erzeugt aus der Grammatik Java-Klassen (oder andere Zielsprachen), die den Input in einen abstrakten Parse-Tree (AST) verwandeln.

Wichtige Klassen:

- `Parser`: Überprüft die syntaktische Struktur.

- `Lexer`: Zerlegt den Input-Stream in Tokens.
- `ParserRuleContext`: Repräsentiert innere Knoten im Baum (Regeln). Speichert Start-/End-Tokens und Referenzen auf Kinder.
- `TerminalNode`: Repräsentiert Blätter im Baum (Tokens/Literale).

1.4 Traversierung: Listener vs. Visitor

ANTLR v4 bietet zwei Mechanismen, um den Parse-Tree zu verarbeiten.

1.4.1 1. Listener Pattern

Der Listener ist der Standardmechanismus. ANTLR generiert einen `ParseTreeWalker`, der den Baum mittels Tiefensuche (DFS) automatisch traversiert.

- **Funktionsweise:** Der Listener reagiert auf Ereignisse. Beim Betreten eines Knotens wird `enterX()` aufgerufen, beim Verlassen `exitX()`.
- **Vorteil:** Vollautomatisch, kein Boilerplate-Code für die Traversierung nötig.
- **Nachteil:** Keine Kontrolle über den Ablauf (Reihenfolge ist fix), keine Rückgabewerte aus Methoden, Kommunikation zwischen Knoten schwierig (oft über Instanzvariablen oder einen Stack).

1.4.2 2. Visitor Pattern

Muss explizit mit `-visitor` generiert werden. Hier steuert der Entwickler die Traversierung selbst.

- **Funktionsweise:** Man implementiert `visitX(Context ctx)`. Um Kinder zu besuchen, muss man explizit `visit(ctx.child)` aufrufen.
- **Vorteil:** Volle Kontrolle (Überspringen von Zweigen, geänderte Reihenfolge), Methoden können Werte zurückgeben (z.B. `Integer` für einen Taschenrechner).
- **Nachteil:** Man muss die Traversierung (den Aufruf von `visit` für Kinder) selbst schreiben.

Vergleich für die Prüfung

Nutzen Sie **Listener**, wenn Sie den gesamten Baum standardmäßig abarbeiten wollen (z.B. Code-Formatierung, einfache Übersetzung).
Nutzen Sie **Visitor**, wenn Sie Ergebnisse berechnen (Interpreter), den Kontrollfluss steuern oder kontextabhängig traversieren müssen.

1.4.3 Labeling von Alternativen

Damit der Visitor/Listener spezifische Methoden für Alternativen generiert (statt einer riesigen Methode mit ‘if/else‘), können Alternativen mit ‘#‘ benannt werden.

Beispiel:

```
expr : expr '*' expr  # MulDiv
     | expr '+' expr  # AddSub
     | INT            # Int
     ;
```

Erzeugt Methoden: `visitMulDiv`, `visitAddSub`, `visitInt`.

1.5 Fortgeschrittene Themen

1.5.1 Assoziativität und Präzedenz

In ANTLR v4 wird Operator-Präzedenz implizit durch die Reihenfolge der Alternativen bestimmt.

- **Präzedenz:** Regeln, die weiter oben stehen, binden stärker (haben höhere Priorität). Beispiel: '*' vor '+' definieren.
- **Assoziativität:** Standard ist links-assoziativ ($1 + 2 + 3 \rightarrow (1 + 2) + 3$). Rechts-Assoziativität (z.B. für Exponenten 2^{3^4}) wird mit `<assoc=right>` annotiert.

1.5.2 Dangling Else Problem

Das Problem der Mehrdeutigkeit bei 'if expr then if expr then stat else stat': Gehört das 'else' zum ersten oder zweiten 'if'?
Lösung in ANTLR: ANTLR parst **greedy** (gierig). Es bindet das 'else' an das nächstmögliche (innerste) offene 'if'. Dies entspricht dem Standardverhalten der meisten Programmiersprachen.

1.5.3 Semantische Prädikate

Manchmal reicht die Grammatik allein nicht aus (z.B. wenn Parsing von Laufzeitdaten abhängt).
Syntax: `{Bedingung}?`
Dies ist ein boolescher Ausdruck in der Zielsprache (Java).

- Ist die Bedingung `true`, wird die Alternative/Regel aktiviert.
- Ist sie `false`, wird die Alternative ignoriert (als ob sie nicht in der Grammatik stünde) und der Parser versucht eine andere Möglichkeit.

Beispiel (Datenabhängiges Parsen): Eine Zahl n gibt an, wie viele folgende Zahlen gelesen werden sollen.

```
sequence[int n] locals [int i = 1;]
   : ( {$i <= $n}? INT {$i++;} )* ;
```

Hier deaktiviert das Prädikat die Schleife, sobald $i > n$ ist.

1.6 Fehlerbehandlung

ANTLR generiert Parser mit eingebauter Fehlerbehandlung.

- **Token ignorieren:** Wenn ein Token unerwartet ist, aber das darauf folgende passt.
- **Token einfügen:** Wenn ein Token fehlt, fügt der Parser ein fiktives Token ein, um weiterzumachen (Single Token Insertion).
- Ziel ist es, möglichst viele Fehler in einem Durchlauf zu finden (Error Recovery), statt beim ersten Fehler abzubrechen.