Formale Sprachen und ihre Anwendungen im Compilerbau

— WS 2010 & SS 2011 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

19. Mai 2011

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

In den beiden Vorlesungen "*Formale Sprachen*" und "*Compilerbau*" werden wir die folgenden Themen behandeln:

1. Reguläre Ausdrücke

   Reguläre Ausdrücke dienen dazu, Mengen von Strings kompakt beschreiben zu können. Sie bilden die Grundlage von Skriptsprachen wie *Perl* [WS92] oder *Python* [Lut09].

2. *JFlex*

   JFlex [Kle05] ist eine Scanner-Generator, mit dessen Hilfe wir später einen Scanner für den Compiler, den wir in dieser Vorlesung entwickeln werden, erzeugen können.

3. Endliche Automaten

   Endliche Automaten ermöglichen es uns, reguläre Ausdrücke zu erkennen.

4. Reguläre Sprachen

   In diesem Kapitel untersuchen wir den Zusammenhang von regulären Ausdrücken und endlichen Automaten.

5. Kontextfreie Sprachen

   Die meisten modernen Programmier-Sprachen lassen sich als sogenannte *kontextfreie Sprachen* darstellen.

6. Antlr

   Das Werkzeug Antlr [Par07] ist ein sehr leistungsfähiger Parser-Generator, der sogenannte *Top-Down-Parser* erzeugt.

7. *JavaCup*

   Das Werkzeug *JavaCup* ist ein weiterer Parser-Generator, mit dessen Hilfe sich *Bottom-Up-Parser* erzeugen lassen

8. Keller-Automaten

   Keller-Automaten spielen in der Theorie der kontextfreien Sprachen eine analoge Rolle wir endliche Automaten in der Theorie der regulären Sprachen.

9. Grenzen kontextfreier Sprachen

10. Compilerbau

    Im zweiten Teil des Skriptes besprechen wir, wie mit Hilfe der im ersten Teil vorgestellten Theorie ein Compiler für eine einfache Programmier-Sprache erstellt werden kann.

1.1 Grundlegende Definitionen

Als erstes müssen wir klären, was wir unter einer *formalen Sprache* verstehen wollen. Dazu folgen jetzt einige Definitionen.

Definition 1 (Alphabet) Ein *Alphabet* Σ ist eine endliche, nicht-leere Menge von *Buchstaben*:

$$\Sigma = \{c_1, \cdots, c_n\}.$$

Statt von Buchstaben sprechen wir gelegentlich auch von *Symbolen*. □

Beispiele:

1. $\Sigma = \{0, 1\}$ ist ein Alphabet, mit dem wir beispielsweise natürliche Zahlen im Binärsystem darstellen können.
2. $\Sigma = \{\mathtt{a}, \cdots, \mathtt{z}, \mathtt{A}, \cdots, \mathtt{Z}\}$ ist das Alphabet, das der englischen Sprache zu Grunde liegt.
3. Die Menge $\Sigma_{\text{ASCII}} = \{0, 1, \cdots, 127\}$ wird als das ASCII-Alphabet bezeichnet. Hierbei werden die Zahlen als Buchstaben, Ziffern, Sonderzeichen, und Kontrollzeichen interpretiert. Beispielsweise repräsentieren die Zahlen von 65 bis 91 die Buchstaben '`A`' bis '`Z`'.

Definition 2 (Wort) Ein Wort eines Alphabets Σ ist eine Liste von Buchstaben aus Σ. In der Theorie der formalen Sprachen werden solche Listen ohne eckigen Klammern und ohne Kommata geschrieben. Sind $c_1, \cdots, c_n \in \Sigma$, so schreiben wir also

$$w = c_1 \cdots c_n \quad \text{an Stelle von} \quad w = [c_1, \cdots, c_n]$$

für das Wort, das aus den Buchstaben c_1 bis c_n besteht. Für die leere Liste, die wir auch als das leere Wort bezeichnen, schreiben wir ε.

Die Menge aller Wörter eines Alphabets wird mit Σ^* bezeichnet. Im Kontext von Programmier-Sprachen werden Wörter auch als *Strings* bezeichnet. □

Beispiele:

1. Es sei $\Sigma = \{0, 1\}$. Setzen wir

   $$w_1 = 01110 \quad \text{und} \quad w_2 = 11001,$$

   so sind w_1 und w_2 Worte, es gilt also

   $$w_1 \in \Sigma^* \quad \text{und} \quad w_2 \in \Sigma^*.$$

2. Es sei $\Sigma = \{\mathtt{a}, \cdots, \mathtt{z}\}$. Setzen wir

   $$w = \mathtt{beispiel},$$

   so gilt $w \in \Sigma^*$.

Unter der *Länge* eines Wortes w verstehen wir die Länge der Liste w. Wir schreiben die Länge eines Wortes als $|w|$. Auf die einzelnen Buchstaben eines Wortes greifen wir mit Hilfe eckiger Klammern zurück: Für ein Wort w und eine positive natürliche Zahl $i \leq |w|$ bezeichnet $w[i]$ den i-ten Buchstaben des Wortes w. Um später leichter mit Wörtern arbeiten zu können, definieren wir die *Konkatenation* zweier Wörter w_1 und w_2 als das Wort w, das entsteht, wenn wir das Wort w_2 hinten an das Wort w_1 anfügen. Wir schreiben die Konkatenation von w_1 und w_2 als $w_1 w_2$.

Beispiel: Ist $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ und gilt $w_1 = \mathtt{01}$ und $w_2 = \mathtt{10}$, so haben wir beispielsweise

$$w_1 w_2 = \mathtt{0110} \quad \text{und} \quad w_2 w_1 = \mathtt{1001}.$$

Definition 3 (Formale Sprache)
Ist Σ ein Alphabet, so bezeichnen wir eine Teilmenge $L \subseteq \Sigma^*$ als eine *formale Sprache*. □

Beispiele:

1. Es sei $\Sigma = \{\texttt{0}, \texttt{1}\}$. Wir definieren

   $$L_{\mathbb{N}} = \{\texttt{1}w \mid w \in \Sigma^*\} \cup \{\texttt{0}\}$$

   Dann ist $L_{\mathbb{N}}$ die Sprache, die aus allen Wörtern besteht, die sich im Binärsystem als natürliche Zahlen interpretieren lassen. Dies sind alle Wörter aus Σ^*, die mit dem Buchstaben `1` beginnen sowie das Wort `0`, das nur aus dem Buchstaben `0` besteht. Es gilt also beispielsweise

   $$\texttt{100} \in L_{\mathbb{N}}, \quad \text{aber} \quad \texttt{010} \notin L_{\mathbb{N}}.$$

   Auf der Menge $L_{\mathbb{N}}$ definieren wir eine Funktion

   $$\mathit{value} : L_{\mathbb{N}} \to \mathbb{N}$$

   durch Induktion nach der Länge des Wortes. Dabei soll $\mathit{value}(w)$ die Zahl ergeben, die durch das Wort w dargestellt wird.

   (a) $\mathit{value}(\texttt{0}) = 0$, $\mathit{value}(1) = 1$,

   (b) $|w| > 0 \to \mathit{value}(w\texttt{0}) = 2 \cdot \mathit{value}(w)$,

   (c) $|w| > 0 \to \mathit{value}(w\texttt{1}) = 2 \cdot \mathit{value}(w) + 1$.

2. Es sei wieder $\Sigma = \{0, 1\}$. Wir definieren die Sprache $L_{\mathbb{P}}$ als die Menge aller der Wörter aus $L_{\mathbb{N}}$, die Primzahlen darstellen:

   $$L_{\mathbb{P}} := \{w \in L_{\mathbb{N}} \mid \mathit{value}(w) \in \mathbb{P}\}$$

   Hier bezeichnet $\mathbb{P}$ die Menge der Primzahlen, also die Menge aller natürlichen Zahlen größer als 1, die nur durch 1 und sich selbst teilbar sind:

   $$\mathbb{P} = \big\{p \in \mathbb{N} \;\big|\; \{t \in \mathbb{N} \mid \exists k \in \mathbb{N} : k \cdot t = p\} = \{1, p\}\big\}.$$

3. Es sei $\Sigma_{\text{ASCII}} = \{0, \cdots, 127\}$ das ASCII-Alphabet. Es bezeichne L_C die Menge aller `C`-Funktionen, die eine Deklaration der Form

   `char*` f`(char*` x`);`

   haben. L_C enthält also die `C`-Funktionen, die als Argument einen String verarbeiten und als Ergebnis einen String zurück liefern.

4. Es sei $\Sigma := \Sigma_{\text{ASCII}} \cup \{\dagger\}$ das Alphabet, das aus dem ASCII-Alphabet dadurch hervorgeht, dass das wir das zusätzliche Zeichen $\dagger$ hinzufügen. Die sogenannte universelle Sprache L_u bestehe aus allen Wörtern der Form

   $$p\dagger x\dagger y$$

   für die gilt

   (a) $p \in L_C$,

   (b) die Anwendung der Funktion p auf den String x terminiert und liefert den String y.

Die obigen Beispiele zeigen, dass der Begriff der formalen Sprache sehr breit gefaßt ist. Während es sehr einfach ist, Wörter der Sprache $L_{\mathbb{N}}$ zu erkennen, ist das bei Wörtern der Sprachen $L_{\mathbb{P}}$ und L_C schon schwieriger. Wir werden später sehen, dass es keinen Algorithmus gibt, mit dem es möglich ist zu entscheiden, ob ein Wort w ein Element der universellen Sprache L_u ist.

1.2 Literatur

Neben dem Skript eignet sich die folgende Literatur zur Nachbereitung der Vorlesung:

1. *Introduction to Automata Theory, Languages, and Computation* [HMU06]

   Dieses Buch ist das Standardwerk zu dem Thema und enthält sämtliche theoretischen Resultate, die wir in der Vorlesung kennen lernen werden.

2. *The Definitive* ANTLR *reference* [PQ95]

   Das Buch beschreibt den Parser-Generator ANTLR.

3. *Compilers — Principles, Techniques and Tools* [ASUL06]

   Das Standardwerk zum Compilerbau, in dem auch viele Aspekte der Theorie der formalen Sprachen dargestellt werden.

Außerdem möchte ich noch auf das Buch "*Mastering Regular Expressions*" von Friedl [Fri02] hinweisen, in dem die Anwendung regulärer Ausdrücke in Skript-Sprachen diskutiert wird. Wir werden im Rahmen der Vorlesung allerdings keine Zeit finden, um darauf einzugehen.

Kapitel 2

Reguläre Ausdrücke

Reguläre Ausdrücke sind Terme, die einfache formale Sprachen spezifizieren. Mit Hilfe eines regulären Ausdrucks lassen sich

1. die Auswahl zwischen mehreren Alternativen,
2. Wiederholungen, und
3. Verkettung

leicht spezifizieren.

2.1 Einige Definitionen

Bevor wir die Definition der regulären Ausdrücke geben können, benötigen wir einige Definitionen.

Definition 4 (Produkt von Sprachen) Es sei Σ ein Alphabet und $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ seien formale Sprachen. Dann definieren wir das *Produkt* der Sprachen L_1 und L_2, geschrieben $L_1 \cdot L_2$, als die Menge aller Konkatenationen $w_1 w_2$, für die w_1 ein Wort aus L_1 und w_2 ein Wort aus L_2 ist:

$$L_1 \cdot L_2 := \left\{ w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2 \right\} \qquad \square$$

Beispiel: Es sei $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und L_1 und L_2 seien als

$$L_1 = \{\mathtt{ab}, \mathtt{bc}\} \quad \text{und} \quad L_2 = \{\mathtt{ac}, \mathtt{cb}\}$$

definiert. Dann gilt

$$L_1 \cdot L_2 = \{\mathtt{abac}, \mathtt{abcb}, \mathtt{bcac}, \mathtt{bccb}\}.$$

Definition 5 (Potenz einer Sprache) Es sei Σ eine Alphabet, $L \subseteq \Sigma^*$ eine formale Sprache und $n \in \mathbb{N}$ eine natürlich Zahl. Wir definieren die *n-te Potenz* von L, wir schreiben L^n, durch Induktion nach n.

I.A.: $n = 0$:

$L^0 := \{\varepsilon\}$.

Hier steht ε für das leere Wort, schreiben wir Worte als Listen von Buchstaben, so gilt also $\varepsilon = []$.

I.S.: $n \mapsto n + 1$:

$L^{n+1} = L^n \cdot L$

Beispiel: Es sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ und $L = \{\mathtt{ab}, \mathtt{ba}\}$. Dann gilt:

1. $L^0 = \{\varepsilon\}$,
2. $L^1 = \{\varepsilon\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{ab}, \mathtt{ba}\}$,
3. $L^2 = \{\mathtt{ab}, \mathtt{ba}\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \mathtt{baba}\}$.

Definition 6 (Kleene-Abschluss) Es sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine formale Sprache. Dann definieren wir den *Kleene-Abschluss* von L, geschrieben L^*, als die Vereinigung der Potenzen L^n für alle $n \in \mathbb{N}$:

$$L^* = \bigcup_{n \in \mathbb{N}} L^n$$ □

Beispiel: Es sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ und $L = \{\mathtt{a}\}$. Dann gilt

$$L^* = \{\mathtt{a}^n \mid n \in \mathbb{N}\}.$$

Hierbei bezeichnet $\mathtt{a}^n$ das Wort der Länge n, das nur den Buchstaben $\mathtt{a}$ enthält, es gilt also

$$\mathtt{a}^n = \underbrace{\mathtt{a} \cdots \mathtt{a}}_{n}$$

Formal definieren wir für einen beliebigen String s und eine natürliche Zahl $n \in \mathbb{N}$ den Ausdruck s^n durch Induktion über n:

I.A. $n = 0$: $s^0 := \varepsilon$.

I.S. $n \mapsto n + 1$: $s^{n+1} := s^n s$,

wobei $s^n s$ für die Konkatenation der Strings s^n und s steht.

Das letzte Beispiel zeigt, dass der Kleene-Abschluss einer endlichen Sprache unendlich sein kann. Offenbar ist der Kleene-Abschluss einer Sprache L dann unendlich, wenn L wenigstens ein Wort mit einer Länge größer als 0 enthält.

Wir geben nun die Definition der regulären Ausdrücke über einem Alphabet Σ. Wir bezeichnen die Menge aller regulären Ausdrücke mit $\mathtt{RegExp}_\Sigma$. Die Definition dieser Menge verläuft induktiv. Gleichzeitig mit der Menge $\mathtt{RegExp}_\Sigma$ definieren wir eine Funktion

$$L : \mathtt{RegExp}_\Sigma \to 2^{\Sigma^*},$$

die jedem regulären Ausdruck r eine formale Sprache $L(r) \subseteq \Sigma^*$ zuordnet.[1]

Definition 7 (reguläre Ausdrücke) Die Menge $\mathtt{RegExp}_\Sigma$ der *regulären Ausdrücke* über dem Alphabet Σ wird wie folgt induktiv definiert:

1. $\emptyset \in \mathtt{RegExp}_\Sigma$

   Der reguläre Ausdruck $\emptyset$ bezeichnet die leere Sprache, es gilt also

   $$L(\emptyset) := \{\}.$$

   Zur Vermeidung von Verwirrung nehmen wir dabei an, dass das Zeichen $\emptyset$ nicht in dem Alphabet Σ auftritt, es gilt also $\emptyset \notin \Sigma$.

2. $\varepsilon \in \mathtt{RegExp}_\Sigma$

   Der reguläre Ausdruck ε bezeichnet die Sprache, die nur das leere Wort ε enthält:

   $$L(\varepsilon) := \{\varepsilon\}$$

   Beachten Sie, das die beiden Auftreten von ε in der obigen Gleichung verschiedene Dinge bezeichnen. Das Auftreten auf der linken Seite der Gleichung bezeichnet einen regulären Ausdruck, während das Auftreten auf der rechten Seite das leere Wort bezeichnet.

[1] Für eine Menge M bezeichnet 2^M die *Potenzmenge* von M, also die Menge aller Teilmengen von M.

3. $c \in \Sigma \rightarrow c \in \texttt{RegExp}_\Sigma$.

   Jeder Buchstabe aus dem Alphabet Σ fungiert also gleichzeitig als regulärer Ausdruck. Dieser Ausdruck beschreibt die Sprache, die aus genau dem Wort c besteht:

   $$L(c) := \{c\}$$

   Bemerken Sie, dass wir an dieser Stelle Buchstaben mit Wörtern der Länge Eins identifizieren.

4. $r_1 \in \texttt{RegExp}_\Sigma \wedge r_2 \in \texttt{RegExp}_\Sigma \rightarrow r_1 + r_2 \in \texttt{RegExp}_\Sigma$

   Aus den regulären Ausdrücken r_1 und r_2 kann mit dem Infix-Operator "+" ein neuer regulärer Ausdruck gebildet werden. Dieser so gebildete Ausdruck beschreibt die Vereinigung der Sprachen von r_1 und r_2:

   $$L(r_1 + r_2) := L(r_1) \cup L(r_2).$$

   Wir nehmen an, dass das Zeichen "+" nicht in dem Alphabet Σ auftritt, es gilt also "+" $\notin \Sigma$.

5. $r_1 \in \texttt{RegExp}_\Sigma \wedge r_2 \in \texttt{RegExp}_\Sigma \rightarrow r_1 \cdot r_2 \in \texttt{RegExp}_\Sigma$

   Aus den regulären Ausdrücken r_1 und r_2 kann also mit dem Infix-Operator "$\cdot$" ein neuer regulärer Ausdruck gebildet werden. Dieser so gebildete Ausdruck beschreibt das Produkt der Sprachen von r_1 und r_2:

   $$L(r_1 \cdot r_2) := L(r_1) \cdot L(r_2).$$

   Wir nehmen wieder an, dass das Zeichen "$\cdot$" nicht in dem Alphabet Σ auftritt, es gilt also "$\cdot$" $\notin \Sigma$.

6. $r \in \texttt{RegExp}_\Sigma \rightarrow r^* \in \texttt{RegExp}_\Sigma$

   Aus dem regulären Ausdrücken r kann mit dem Postfix-Operator "*" ein neuer regulärer Ausdruck gebildet werden. Dieser so gebildete Ausdruck steht für den Kleene-Abschluss der durch r beschriebenen Sprache:

   $$L(r^*) := \big(L(r)\big)^*.$$

   Wir nehmen "*" $\notin \Sigma$ an.

7. $r \in \texttt{RegExp}_\Sigma \rightarrow (r) \in \texttt{RegExp}_\Sigma$

   Reguläre Ausdrücke können geklammert werden. Dadurch ändert sich die Sprache natürlich nicht:

   $$L\big((r)\big) := L(r)\ .$$

   Wir nehmen dabei an, dass die Klammer-Symbole "(" und ")" nicht in dem Alphabet Σ auftreten, es gilt also "(" $\notin \Sigma$ und ")" $\notin \Sigma$. □

Um Klammern zu sparen vereinbaren wir die folgenden Operator-Präzedenzen:

1. Der Postfix-Operator "*" bindet am stärksten.
2. Der Infix-Operator "$\cdot$" bindet schwächer als "*" und stärker als "+".
3. Der Infix-Operator "+" bindet am schwächsten.

Damit wird also der reguläre Ausdruck

$$a + b \cdot c^* \quad \text{implizit geklammert als} \quad a + \big(b \cdot (c^*)\big).$$

Beispiele: Bei den Beispielen ist das Alphabet Σ durch die Definition

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$$

festgelegt.

1. $r_1 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})$

   Der Ausdruck r_1 beschreibt alle Wörter, die aus genau zwei Buchstaben bestehen:

   $$L(r_1) = \{w \in \Sigma^* \mid |w| = 2\}$$

2. $r_2 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})^*$

   Der Ausdruck r_2 beschreibt alle Wörter, die aus wenigstens einem Buchstaben bestehen.

   $$L(r_2) = \{w \in \Sigma^* \mid |w| \geq 1\}$$

3. $r_3 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^*$

   Der Ausdruck r_3 beschreibt alle Wörter, in denen der Buchstabe `a` genau einmal auftritt. Ein solches Wort besteht aus einer beliebigen Anzahl der Buchstaben `b` und `c` (dafür steht der Teilausdruck $(\mathtt{b}+\mathtt{c})^*$) gefolgt von dem Buchstaben `a`, wiederum gefolgt von einer beliebigen Anzahl der Buchstaben `b` und `c`.

   $$L(r_3) = \left\{w \in \Sigma^* \;\middle|\; \#\{i \in \mathbb{N} \mid w[i] = \mathtt{a}\} = 1\right\}$$

4. $r_4 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^* + (\mathtt{a} + \mathtt{c})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{c})^*$

   Der Ausdruck r_4 beschreibt alle die Wörter, die entweder genau ein `a` oder genau ein `b` enthalten.

   $$L(r_4) = \left\{w \in \Sigma^* \;\middle|\; \#\{i \in \mathbb{N} \mid w[i] = \mathtt{a}\} = 1\right\} \cup \left\{w \in \Sigma^* \;\middle|\; \#\{i \in \mathbb{N} \mid w[i] = \mathtt{b}\} = 1\right\}$$

Aufgabe 1:

(a) Es sei $\Sigma = \{a, b, c\}$. Geben Sie einen regulären Ausdruck für die Sprache $L \subseteq \Sigma^*$ an, deren Wörter mindestens ein a und mindestens ein b enthalten.

(b) Es sei $\Sigma = \{0, 1\}$. Geben Sie einen regulären Ausdruck für die Sprache $L \subseteq \Sigma^*$ an, für die das drittletzte Zeichen eine 1 ist.

(c) Wieder sei $\Sigma = \{0, 1\}$. Geben Sie einen regulären Ausdruck für die Sprache $L \subseteq \Sigma^*$ der Wörter an, in denen der Teilstring 110 nicht auftritt.

(d) Welche Sprache wird durch den regulären Ausdruck $(1 + \varepsilon) \cdot (0 \cdot 0^* \cdot 1)^* \cdot 0^*$ beschrieben?

2.2 Algebraische Vereinfachung regulärer Ausdrücke

Es gelten die folgenden Gesetze:

1. $r_1 + r_2 \doteq r_2 + r_1$

   Mit dem Zeichen $\doteq$ meinen wir hier, dass die Sprachen, die durch die regulären Ausdrücke beschrieben werden, gleich sind, die obere Gleichung ist also nur eine Kurzschreibweise für

   $$L(r_1 + r_2) = L(r_2 + r_1).$$

   Der Beweis dieser Gleichung folgt aus der Definition und der Kommutativität der Vereinigung von Mengen:

   $$L(r_1 + r_2) = L(r_1) \cup L(r_2) = L(r_2) \cup L(r_1) = L(r_2 + r_1).$$

2. $(r_1 + r_2) + r_3 \doteq r_1 + (r_2 + r_3)$

   Diese Gleichung folgt aus der Assoziativität der Vereinigung.

3. $(r_1 \cdot r_2) \cdot r_3 \doteq r_1 \cdot (r_2 \cdot r_3)$

   Diese Gleichung folgt aus der Tatsache, dass die Konkatenation von Worten assoziativ ist, für beliebige Wörter u, v und w gilt

$$(uv)w = u(vw).$$

Daraus folgt

$$\begin{aligned} L\big((r_1 \cdot r_2) \cdot r_3\big) &= \big\{xw \mid x \in L(r_1 \cdot r_2) \wedge w \in L(r_3)\big) \\ &= \big\{(uv)w \mid u \in L(r_1) \wedge v \in L(r_2) \wedge w \in L(r_3)\big) \\ &= \big\{u(vw) \mid u \in L(r_1) \wedge v \in L(r_2) \wedge w \in L(r_3)\big) \\ &= \big\{uy \mid u \in L(r_1) \wedge y \in L(r_2 \cdot r_3)\big) \\ &= L\big(r_1 \cdot (r_2 \cdot r_3)\big). \end{aligned}$$

4. $\emptyset \cdot r \doteq r \cdot \emptyset \doteq \emptyset$
5. $\varepsilon \cdot r \doteq r \cdot \varepsilon \doteq r$
6. $\emptyset + r \doteq r + \emptyset \doteq r$
7. $(r_1 + r_2) \cdot r_3 \doteq r_1 \cdot r_3 + r_2 \cdot r_3$
8. $r_1 \cdot (r_2 + r_3) \doteq r_1 \cdot r_2 + r_1 \cdot r_3$
9. $r + r \doteq r$, denn
$$L(r + r) = L(r) \cup L(r) = L(r).$$
10. $(r^*)^* \doteq r^*$

   Wir haben
$$L(r^*) = \bigcup_{n \in \mathbb{N}} L(r)^n$$
   und daraus folgt allgemein $L(r) \subseteq L(r^*)$. Ersetzen wir in dieser Beziehung r durch r^*, so sehen wir, dass
$$L(r^*) \subseteq L\big((r^*)^*\big)$$
   gilt. Um die Umkehrung
$$L\big((r^*)^*\big) \subseteq L(r^*)$$
   zu beweisen, betrachten wir zunächst die Worte $w \in L\big((r^*)^*\big)$. Wegen
$$L\big((r^*)^*\big) = \bigcup_{n \in \mathbb{N}} L(r^*)^n$$
   gilt $w \in L\big((r^*)^*\big)$ genau dann, wenn es ein $n \in \mathbb{N}$ gibt, so dass es Wörter $u_1, \cdots, u_n \in L(r^*)$ gibt, so dass
$$w = u_1 \cdots u_n.$$
   Wegen $u_i \in L(r^*)$ gibt es für jedes $i \in \{1, \cdots, n\}$ eine Zahl $m(i) \in \mathbb{N}$, so dass für $j = 1, \cdots, m(i)$ Wörter $v_{i,j} \in L(r)$ gibt, so dass
$$u_i = v_{1,i} \cdots v_{m(i),i}$$
   gilt. Insgesamt gilt dann
$$w = v_{1,1} \cdots v_{m(1),1} v_{1,2} \cdots v_{m(2),2} \cdots v_{1,n} \cdots v_{m(n),n}.$$
   Also ist w eine Konkatenation von Wörtern der Sprache $L(r)$ und das heißt
$$w \in L\big(r^*\big)$$
   und damit ist die Inklusion $L\big((r^*)^*\big) \subseteq L(r^*)$ gezeigt.
11. $\emptyset^* \doteq \varepsilon$

12. $\varepsilon^* \doteq \varepsilon$
13. $r^* \doteq \varepsilon + r^* \cdot r$
14. $r^* \doteq (\varepsilon + r)^*$

Leider gibt es kein System von Gleichungen, aus denen man alle anderen Gleichungen für reguläre Ausdrücke ableiten kann. Es gibt aber eine Ableitungs-Regel, die zusammen mit den oben angegeben Gleichungen vollständig ist. Diese Regel lautet

$$\frac{r \doteq r \cdot s + t \quad \varepsilon \notin L(s)}{r \doteq t \cdot s^*}$$

Der Beweis dieser Behauptung geht weit über den Rahmen der Vorlesung hinaus, er findet sich in einem Papier von Arto Salomaa [Sal66].

Aufgabe 2: Vereinfachen Sie den folgenden regulären Ausdruck:

$$(0 + 1)^* \cdot 1 \cdot (0 + 1) + (0 + 1)^* \cdot 1 \cdot (0 + 1) \cdot (0 + 1).$$

Kapitel 3

Der Scanner-Generator *JFlex*

Ein *Scanner* ist ein Werkzeug, das einen gegebenen Text in Gruppen einzelner *Token* aufspaltet. Beispielsweise spaltet der Scanner, der für einen `C`-Compiler eingesetzt wird, den Programmtext in die folgenden Token auf:

1. Schlüsselwörter wie "`if`", "`while`", etc.
2. Operator-Symbole wie "`+`", "`+=`", "`<`", "`<=`", etc.
3. Konstanten, wobei es in der Sprache `C` drei Arten von Konstanten gibt:
   (a) Zahlen, beispielsweise "`123`" oder "`1.23e2`",
   (b) Strings, die in doppelten Hochkommata eingeschlossen sind, beispielsweise "`"hallo"`",
   (c) einzelne Buchstaben, die in Quotes eingeschlossen sind, beispielsweise "`'a'`".
4. Namen, die als Bezeichner für Variablen, Funktionen, oder Typ-Definitionen fungieren.
5. Kommentare.
6. Sogenannte *White-Space-Zeichen*. Hierzu gehören Leerzeichen, horizontale und vertikale Tabulatoren, Zeilenumbrüche und Seitenvorschübe.

Das Werkzeug *JFlex* [Kle05] ist ein sogenannter Scanner-Generator, also ein Werkzeug, das aus einer Spezifikation verschiedener Token automatisch einen Scanner generiert. Die einzelnen Token werden dabei durch reguläre Ausdrücke definiert. Im Netz finden Sie dieses Programm unter der Adresse

```
http://jflex.de
```

Da das Programm selber in *Java* implementiert ist, kann es auf allen Plattformen eingesetzt werden, auf denen *Java* zur Verfügung steht.

In nächsten Abschnitt besprechen wir die Struktur einer *JFlex*-Eingabe-Datei und zeigen wie *JFlex* aufgerufen wird. Anschließend zeigen wir, wie reguläre Ausdrücke in der Eingabe-Sprache von *JFlex* spezifiziert werden können. Das Kapitel wird durch ein Beispiel abgerundet, bei dem wir mit Hilfe von *JFlex* ein Programm erzeugen, mit dessen Hilfe die Ergebnisse einer Klausur ausgewertet werden können.

Die von *JFlex* erzeugten Scanner sind *Java*-Programme. Für die Sprachen `C` und `C++` gibt es ein Äquivalent unter dem Namen *Flex*.

3.1 Struktur einer *JFlex*-Spezifikation

Eine *JFlex*-Spezifikation besteht aus drei Abschnitten, die durch den String "`%%`", der am Anfang einer Zeile stehen muss, von einander getrennt werden.

1. Der erste Teil ist der *Benutzer-Code*. Er besteht aus `package`-Deklarationen und `import`-Befehlen, die wörtlich an den Anfang der erzeugten Scanner-Klasse kopiert werden. Zusätzlich kann der Benutzer-Code noch die Definition lokaler *Java*-Klassen enthalten. Allerdings ist es sinnvoller, solche Klassen in separaten Dateien definieren.

   Abbildung 3.1 auf Seite 15 zeigt ein Beispiel für eine *JFlex*-Spezifikation für einen Scanner, der Zahlen erkennen und aufaddieren soll. In diesem Fall besteht der Benutzer-Code nur aus der `package`-Deklaration in Zeile 1.

2. Der zweite Teil ist *Options-Teil*. Dieser Teil enthält die Spezifikation verschiedener Optionen, sowie eventuell die Deklaration von Variablen und Methoden der erzeugten Scanner-Klasse.

   In Abbildung 3.1 erstreckt sich dieser Teil von Zeile 3 bis Zeile 15.

3. Der dritte Teil ist der *Aktions-Teil*. Hier werden die Strings, die der Scanner erkennen soll, mit Hilfe von regulären Ausdrücken spezifiziert. Zusätzlich wird festgelegt, wie der Scanner diese Strings verarbeiten soll.

   In Abbildung 3.1 erstreckt sich der *Aktions-Teil* von Zeile 17 bis Zeile 19.

```
1  package count;
2  %%
3
4  %class Count
5  %standalone
6  %unicode
7
8  %{
9      int mCount = 0;
10 %}
11
12 %eof{
13     System.out.println("Total: " + mCount);
14 %eof}
15
16 %%
17
18 [1-9][0-9]*  { mCount += new Integer(yytext()); }
19 .|\n         { /* skip */                        }
```

Abbildung 3.1: Eine einfache Scanner-Spezifikation für *JFlex*

```
1  Hier sind 3 Äpfel und 5 Birnen.
2  Und hier sind 5 Bananen.
3  Wieviel Stücke Obst sind in diesem Text versteckt?
```

Abbildung 3.2: Eine Eingabe-Datei für den in Abbildung 3.1 spezifizierten Scanner.

Wir diskutieren nun die in Abbildung 3.1 gezeigte *JFlex*-Spezifikation. Der in dieser Abbildung gezeigte Scanner hat die Aufgabe, alle natürlichen Zahlen, die in einer Datei vorkommen, aufzuaddieren. Eine Eingabe-Datei für den zu entwickelnden Scanner könnte wie in Abbildung 3.2 auf Seite 15 gezeigt aussehen. Wir diskutieren die Spezifikation des Scanners aus Abbildung 3.1 jetzt Zeile für Zeile.

1. Zeile 1 spezifiziert, dass der erzeugte Scanner zu dem Paket `count` gehören soll. Im allgemeinen Fall würden hier auch noch `import`-Spezifikationen stehen, aber dieses Beispiel ist so einfach, dass keine Imports erforderlich sind.

2. Zeile 4 spezifiziert, dass die erzeugte Scanner-Klasse den Namen `Count` haben soll.

3. Zeile 5 legt durch die Option "`%standalone`" fest, dass der erzeugte Scanner nicht Teil eines Parsers ist, sondern als unabhängiges Programm eingesetzt werden soll. Daher wird *JFlex* die Klasse `Count` mit einer Methode `main`() ausstatten. Diese Methode wird alle Dateien, die ihr als Argument übergeben werden, scannen.

   Einen Scanner, der selber mit einer `main`-Methode ausgestattet ist, werden wir im Folgenden als *Stand-Alone-Scanner* bezeichnen.

4. Zeile 6 legt durch die Option "`%unicode` fest, dass ein Scanner für Unicode erzeugt werden soll. Zum Scannen von Text-Dateien sollte diese Option immer gewählt werden.

5. In den Zeilen 8 bis 10 deklarieren wir mit Hilfe der Schlüsselwörter "`%{`" und "`%}`", die Variable `mCount` als zusätzliche Member-Variable des erzeugten Scanners. An dieser Stelle können wir neben Member-Variablen auch zusätzliche Methoden definieren.

   Es ist zu beachten, dass die Schlüsselwörter "`%{`" und "`%}`" am Anfang einer Zeile stehen müssen.

6. In den Zeilen 12 bis 14 spezifizieren wir mit Hilfe der Schlüsselwörter "`%eof{`" und "`%eof}`" einen Befehl, der ausgeführt werden soll, wenn das Ende der Datei erreicht ist. Dies ist nur bei einem Stand-Alone-Scanner notwendig. In dem Beispiel geben wir hier die ermittelte Summe aller Zahlen aus.

   Wieder ist zu beachten, dass die Schlüsselwörter "`%eof{`" und "`%eof}`" am Zeilen-Anfang stehen müssen.

7. Die Zeilen 18 und 19 enthalten die Regeln unseres Scanners. Eine Regel hat die Form

   regexp "{" *action* "}"

   Hierbei ist *regexp* ein regulärer Ausdruck und *action* ist ein Fragment von *Java*-Code, das ausgeführt wird, wenn der Ausdruck *regexp* im Text erkannt wird.

   In Zeile 18 spezifiziert der reguläre Ausdruck "`[1-9][0-9]*`" eine natürliche Zahl. Der zu dieser Zahl korrespondierende String wird von der Funktion `yytext`() zurück gegeben und mit Hilfe des Konstruktors für die Klasse `Integer` in eine Zahl umgewandelt.

   In Zeile 19 spezifiziert der reguläre Ausdruck "`.|\n`" ein beliebiges Zeichen. Der reguläre Ausdruck "`.`" spezifiziert dabei ein Zeichen, dass von einem Zeilenumbruch verschieden ist, während "`\n`" für einen Zeilenumbruch steht. Der Operator "`|`" steht für die Alternative. Im letzten Kapitel hatten wir dafür den Operator "+" verwendet. Sie sehen, dass die Syntax, mit der reguläre Ausdrücke in *JFlex* spezifiziert werden können, von der im zweiten Kapitel gegebenen Darstellung abweicht. Wir werden diese Syntax später im Detail diskutieren.

   Die Aktion besteht hier nur aus einem Kommentar. Daher wird das durch den regulären Ausdruck "`.|\n`" erkannte Zeichen einfach überlesen. Diese Regel ist notwendig, denn ein Stand-Alone-Scanner gibt jedes Zeichen, das nicht von einer Regel erkannt wird, auf der Standard-Ausgabe aus.

Aus der in Abbildung 3.1 gezeigten *JFlex*-Spezifikation können wir mit dem Befehl

```
jflex -d count count.jflex
```

die Datei `Count.java` erzeugen. Die Option "`-d`" spezifiziert dabei, dass diese Datei in dem Verzeichnis `count` erstellt wird. Das ist notwendig, weil der Scanner die Package-Spezifikation

```
package count;
```

enthält. Die so erzeugte Datei `Count.java` können wir mit dem Befehl

```
javac count/Count.java
```

übersetzen. Den Scanner können wir dann über den Befehl

```
java count.Count input.txt
```

aufrufen, wobei `input.txt` den Namen der zu scannenden Datei angibt.

3.2 Reguläre Ausdrücke in *JFlex*

Im letzten Kapitel haben wir reguläre Ausdrücke mit einer minimalen Syntax definiert. Dies ist nützlich, wenn wir später die Äquivalenz von den durch regulären Ausdrücken spezifizierten Sprachen mit den Sprachen, die von endlichen Automaten erkannt werden können, beweisen wollen. Für die Praxis ist eine reichhaltigere Syntax wünschenswert. Daher bietet die Eingabe-Sprache von *JFlex* eine Reihe von Abkürzungen an, mit der komplexe reguläre Ausdrücke kompakter beschrieben werden können. Den regulären Ausdrücken von `JFlex` liegt das Ascii-Alphabet zu Grunde, wobei zwischen den Zeichen, die als Operatoren dienen können, und den restlichen Zeichen unterschieden wird. Die Menge *OpSyms* der Operator-Symbole ist wie folgt definiert:

$$\begin{aligned} \mathit{OpSyms} := \{\ &\text{“}\texttt{.}\text{”},\ \text{“}\texttt{*}\text{”},\ \text{“}\texttt{+}\text{”},\ \text{“}\texttt{?}\text{”},\ \text{“}\texttt{!}\text{”},\ \text{“}\texttt{\~{}}\text{”},\ \text{“}\texttt{|}\text{”},\ \text{“}\texttt{(}\text{”},\ \text{“}\texttt{)}\text{”},\ \text{“}\texttt{[}\text{”},\ \text{“}\texttt{]}\text{”},\ \text{“}\texttt{\{}\text{”},\ \text{“}\texttt{\}}\text{”},\\ &\text{“}\texttt{<}\text{”},\ \text{“}\texttt{>}\text{”},\ \text{“}\texttt{/}\text{”},\ \text{“}\texttt{\textbackslash}\text{”},\ \text{“}\texttt{\^{}}\text{”},\ \text{“}\texttt{\$}\text{”},\ \text{“}\texttt{"}\text{”}\ \} \end{aligned}$$

Damit können wir nun die Menge *Regexp* der von *JFlex* unterstützen regulären Ausdrücke induktiv definieren.

1. $c \in \mathit{Regexp}$ falls $c \in \Sigma_{\text{Ascii}} \backslash \mathit{OpSyms}$

   Alle Buchstaben c aus dem Ascii-Alphabet, die keine Operator-Symbol sind, können als reguläre Ausdrücke, verwendet werden. Diese Ausdrücke spezifizieren genau diesen Buchstaben.

2. “`.`”$\in \mathit{Regexp}$

   Der reguläre Ausdruck “`.`” spezifiziert ein Zeichen, das von einem Zeilenumbruch “`\n`” verschieden ist.

3. $\backslash x \in \mathit{Regexp}$ falls $x \in \{\texttt{a}, \texttt{b}, \texttt{f}, \texttt{n}, \texttt{r}, \texttt{t}, \texttt{v}\}$

   Die Syntax $\backslash x$ ermöglicht es, Steuerzeichen zu spezifizieren. Im einzelnen gilt:

   (a) `\a` entspricht dem Steuerzeichen `Ctrl-G` (*alert*).

   (b) `\b` entspricht dem Steuerzeichen `Ctrl-H` (*backspace*).

   (c) `\f` entspricht dem Steuerzeichen `Ctrl-L` (*form feed*).

   (d) `\n` entspricht dem Steuerzeichen `Ctrl-J` (*newline*).

   (e) `\r` entspricht dem Steuerzeichen `Ctrl-M` (*carriage return*).

   (f) `\t` entspricht dem Steuerzeichen `Ctrl-I` (*tabulator*).

   (g) `\v` entspricht dem Steuerzeichen `Ctrl-K` (*vertical tabulator*).

4. $\backslash abc \in \mathit{Regexp}$ falls $a, b, c \in \{\texttt{0}, \cdots \texttt{7}\}$

   Bei der Syntax $\backslash abc$ sind a, b und c oktale Ziffern und abc muss als Zahl im Oktal-System interpretierbar sein. Dann wird durch $\backslash abc$ das Zeichen spezifiziert, das im Ascii-Code an der durch die Oktalzahl abc spezifizierten Stelle steht. Beispielsweise steht der Ausdruck “`\040`” für das Leerzeichen “ ”, denn das Leerzeichen hat dezimal den *Ascii*-Code 32 und in dem Oktalsystem schreibt sich diese Zahl als $40_{(8)}$.

5. $\backslash o \in \mathit{Regexp}$ falls $o \in \mathit{OpSyms}$

   Die Operator-Symbole können durch Voranstellen eines Backslashs spezifiziert werden. Wollen wir beispielsweise das Zeichen "`*`" erkennen, so können wir dafür den regulären Ausdruck "`\*`" verwenden.

6. $r_1 r_2 \in \mathit{Regexp}$ falls $r_1, r_2 \in \mathit{Regexp}$

   Die Konkatenation zweier regulärer Ausdrücke wird in *JFlex* ohne den Infix-Operator "$\cdot$" geschrieben. Der Ausdruck $r_1 r_2$ steht also für einen String s der sich in die Form $s = uv$ so aufspalten läßt, dass u durch r_1 und v durch r_2 spezifiziert wird.

7. $r_1 | r_2 \in \mathit{Regexp}$ falls $r_1, r_2 \in \mathit{Regexp}$

   Für die Addition zweier regulärer Ausdrücke wird in *JFlex* an Stelle des Infix-Operators "+" der Operator "`|`" verwendet.

8. $r* \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

   Der Postfix-Operator "`*`" bezeichnet den Kleene-Abschluss.

9. $r+ \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

   Der Ausdruck "r`+`" ist eine Variante des Kleene-Abschlusses, bei der gefordert wird, dass r mindestens einmal auftritt. Daher gilt die folgende Äquivalenz:

   $$r+ \doteq rr*.$$

10. $r? \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

    Der Ausdruck "r`?`" legt fest, dass r einmal oder keinmal auftritt. Es gilt die folgende Äquivalenz:

    $$r? \doteq r|\varepsilon.$$

    Hier ist allerdings zu beachten, dass der Ausdruck "ε" von *JFlex* nicht unterstützt wird.

11. $r\{n\} \in \mathit{Regexp}$ falls $n \in \mathbb{N}$

    Der Ausdruck "r{n}" legt fest, dass r genau n mal auftritt. Der reguläre Ausdruck "`a{4}`" beschreibt also den String "`aaaa`".

12. $\hat{}r$ falls $r \in \mathit{Regexp}$

    Der Ausdruck $\hat{}r$ legt fest, dass der reguläre Ausdruck r am Anfang einer Zeile stehen muss.

    Bei der Verwendung des Operators "`^`" gibt es eine wichtige Einschränkung: Der Operator "`^`" darf nur auf der äußersten Ebene eines regulären Ausdrucks verwendet werden. Eine Konstruktion der Form

    $$\hat{}r_1|r_2$$

    ist also verboten, denn hier tritt der Operator innerhalb einer Alternative auf.

13. $r\$$ falls $r \in \mathit{Regexp}$

    Der Ausdruck $r\$$ legt fest, dass der reguläre Ausdruck r am Ende einer Zeile stehen muss.

    Für die Verwendung des Operators "`$`" gibt es eine ähnliche Einschränkung wie bei dem Operator "`^`": Der Operator "`$`" darf nur auf der äußersten Ebene eines regulären Ausdrucks verwendet werden. Eine Konstruktion der Form

    $$r_1|r_2\$$

    ist verboten, denn hier tritt der Operator innerhalb einer Alternative auf.

14. r_1/r_2 falls $r_1, r_2 \in \mathit{Regexp}$

    Der Ausdruck r_1/r_2 legt fest, dass auf den durch r_1 spezifizierten Text ein Text folgen muss, der der Spezifikation r_2 genügt. Im Unterschied zur einfachen Konkatenation von r_1 und r_2 wird durch den regulären Ausdruck r_1/r_2 aber der selbe Text spezifiziert, der durch r_1 spezifiziert wird. Der Operator "`/`" liefert also nur eine zusätzliche Bedingung, die für eine erfolgreiche Erkennung des regulären Ausdrucks erfüllt sein muss. Die Methode "`yytext()`", die den erkannten Text zurück gibt, liefert nur den Text zurück, der dem regulären Ausdruck r_1 entspricht. Der Text, der dem regulären Ausdruck r_2 entspricht, kann dann von der nächsten Regel bearbeitet werden. In der angelsächsischen Literatur wird r_2 als *trailing context* bezeichnet.

15. $(r) \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

    Genau wie im letzten Kapitel auch können reguläre Ausdrücke geklammert werden. Für die Präzedenzen der Operatoren gilt: Die Postfix-Operatoren "`*`", "`?`", "`+`" und "$\{n\}$" binden am stärksten, der Operator "`|`" bindet am schwächsten.

Alle bis hierher vorgestellten Operatoren können auch in dem für die Sprache `C` verfügbaren Werkzeug *Flex* verwendet werden. *JFlex* unterstützt zusätzlich den Negations-Operator "`!`" und den *Upto*-Operator "`~`". Wir besprechen diese Operatoren später.

Die Spezifikation der regulären Ausdrücke ist noch nicht vollständig, denn es gibt in *JFlex* noch die Möglichkeit, sogenannte *Bereiche* zu spezifizieren. Ein *Bereich* spezifiziert eine Menge von Buchstaben in kompakter Weise. Dazu werden die eckigen Klammern benutzt. Beispielsweise lassen sich die Vokale durch den regulären Ausdruck

```
[aeiou]
```

spezifizieren. Dieser Ausdruck ist als Abkürzung zu verstehen, es gilt:

`[aeiou]` $\doteq$ `a|e|i|o|u`

Die Menge aller kleinen lateinischen Buchstaben läßt sich durch

```
[a-z]
```

spezifizieren, es gilt also

`[a-z]` $\doteq$ `a|b|c|`$\cdots$`|x|y|z`.

Die Menge aller lateinischen Buchstaben zusammen mit dem Unterstrich kann durch

```
[a-zA-Z_]
```

beschrieben werden. *JFlex* gestattet auch, das Komplement einer solchen Menge zu bilden. Dazu ist es lediglich erforderlich, nach der öffnenden eckigen Klammer das Zeichen "`^`" zu verwenden. Beispielsweise beschreibt der Ausdruck

```
[^0-9]
```

alle Ascii-Zeichen, die keine Ziffern sind.

Beispiele: Um die Diskussion anschaulicher zu machen, präsentieren wir einige Beispiele regulärer Ausdrücke.

1. `[a-zA-Z][a-zA-Z0-9_]*`

   Dieser reguläre Ausdruck spezifiziert die Worte, die aus lateinischen Buchstaben, Ziffern und dem Unterstrich "`_`" bestehen und die außerdem mit einem lateinischen Buchstaben beginnen.

2. `\/\/.*`

   Hier wird ein `C`-Kommentar beschrieben, der sich bis zum Zeilenende erstreckt.

3. `0|[1-9][0-9]*`

   Dieser Ausdruck beschreibt natürliche Zahlen. Hier ist es wichtig darauf zu achten, dass eine natürliche Zahl nur dann mit der Ziffer `0` beginnt, wenn es sich um die Zahl `0` handelt.

Unsere bisherige Diskussion regulärer Ausdrücke ist noch nicht vollständig. Gegenüber dem Werkzeug *Flex* bietet *JFlex* die folgenden zusätzlichen Möglichkeiten.

1. Reguläre Ausdrücke können zur Formatierung Leerzeichen und Tabulatoren enthalten. Folglich verändert das Einfügen von Leerzeichen und Tabulatoren die Semantik eines regulären Ausdrucks nicht. Soll ein Leerzeichen oder ein Tabulator erkannt werden, so ist dies durch die regulären Ausdrücke "`[ ]`" bzw. "`\t`" möglich.

2. Innerhalb von doppelten Hochkommata verlieren alle Operator-Symbole bis auf "\" ihre Bedeutung. So beschreibt der reguläre Ausdruck

   ```
   "/*"
   ```

   beispielsweise den Anfang eines mehrzeiligen `C`-Kommentars. Sowohl "`/`" als auch "`*`" sind eigentlich Operator-Symbole, aber innerhalb der Anführungszeichen stehen diese Zeichen für sich selbst.

3. *JFlex* bietet zusätzlich den Negations-Operator "`!`" als Präfix-Operator an. Ist r ein regulärer Ausdruck, so steht der reguläre Ausdruck "`!`r" für das Komplement der durch r beschriebenen Sprache.

   Die Präzedenz dieses Operators ist geringer als die Präzedenz der Postfix-Operatoren "`+`", "`*`" und "`?`", aber höher als die Präzedenz des Konkatenations-Operators. Damit wird der Ausdruck

   ```
   !a*b    also als    (!(a*))b
   ```

   geklammert. Wie nützlich dieser Operator ist, zeigt sich bei der Spezifikation von mehrzeiligen `C`-Kommentaren. Ein regulärer Ausdruck, der mehrzeilige `C`-Kommentare erkennt, läßt sich mit dem Operator "`!`"in der Form

   ```
   "/*" !([^]* "*/" [^]*) "*/"
   ```

   schreiben. Wir diskutieren diesen Ausdruck im Detail.

   (a) Zunächst müssen wir natürlich die Zeichenreihe "`/*`" erkennen. Dafür ist der Ausdruck "`"/*"`" zuständig. Dieser Ausdruck ist in den doppelten Anführungszeichen "`"`" eingeschlossen, damit das Zeichen "`*`" nicht als Operator interpretiert wird.

   (b) Das Innere eines Kommentars darf die Zeichenreihe "`*/`" nicht enthalten. Der Ausdruck "`[^]`" spezifiziert das Komplement der leeren Menge, also ein beliebiges Zeichen. Damit steht der Ausdruck

   ```
   [^]* "*/" [^]*
   ```

   für einen Text, der irgendwo innen drin den String "`*/`" enthält, wobei vorher und nachher beliebige Zeichen stehen können. Die Negation dieses Ausdrucks beschreibt dann genau das, was in einem Kommentar der Form `/*` $\cdots$ `*/` innen drin stehen darf: Beliebiger Text, der den String "`*/`" nicht als Zeichenfolge enthält.

   (c) Am Ende wird der Kommentar dann durch den String "`*/`" abgeschlossen.

   Der Negations-Operator kann auch dazu benutzt werden, denn Durchschnitt zweier regulärer Ausdrücke r_1 und r_2 zu definieren, denn aufgrund des deMorgan'schen Gesetztes der Aussagen-Logik gilt für beliebige aussagen-logische Formeln f_1 und f_2

   $$f_1 \wedge f_2 \leftrightarrow \neg(\neg f_1 \vee \neg f_2).$$

   Damit können wir den Durchschnitt der regulären Ausdrücke r_1 und r_2 als

`!(!`r_1`|!`r_2`)`

definieren. Dieser Ausdruck beschreibt also gerade solche Strings, die sowohl in der durch den regulären Ausdruck r_1 spezifizierten Sprache, als auch in der durch r_2 spezifizierten Sprache liegen.

Die Erfahrung zeigt, dass die endlichen Automaten, die mit Hilfe des Negations-Operators konstruiert werden, sehr groß werden können. Daher sollte dieser Operator mit Vorsicht benutzt werden.

4. Weiterhin bietet *JFlex* den *Upto*-Operator "`~`" an. Ist r ein regulärer Ausdruck, so spezifiziert der Ausdruck

   `~`r

   den kürzesten String, der mit r endet. Mit diesem Operator kann ein mehrzeiliger `C`-Kommentar sehr prägnant durch den Ausdruck

   `"/*" ~"*/"`

   spezifiziert werden. Dieser Ausdruck ist wie folgt zu lesen: Ein mehrzeiliger Kommentar beginnt mit dem String "`/*`" und endet mit dem ersten Aufreten des Strings "`*/`".

   Intern wird der Upto-Operator mit Hilfe des Negations-Operators implementiert. Der Ausdruck `~`r wird auf den Ausdruck `!([^]*` r `[^]*)` r zurück geführt. Dabei steht "`!([^]*` r `[^]*)`" für beliebigen Text, der r nicht enthält. Darauf folgt dann Text, der durch r beschrieben wird.

5. In *JFlex* können innerhalb von Bereichen bestimmte in *Java* vordefinierte *Zeichen-Klassen* benutzt werden um reguläre Ausdrücke zu spezifizieren. So spezifiziert der reguläre Ausdruck

   `[:digit:]`

   beispielsweise eine Ziffer. Insgesamt sind die folgenden Zeichen-Klassen vordefiniert, die jeweils auf Methoden der Klasse `java.lang.Character` zurückgeführt werden:

   (a) `[:jletter:]` wird zurückgeführt auf die Methode `isJavaIdentifierStart`().
   Der reguläre Ausdruck beschreibt also genau die Zeichen c, für welche der Aufruf

   `Character.isJavaIdentifierStart(`c`)`

   als Ergebnis `true` zurück liefert.

   (b) `[:jletterdigit:]` wird zurückgeführt auf die Methode `isJavaIdentifierPart`().

   (c) `[:letter:]` wird zurückgeführt auf die Methode `isLetter`().

   (d) `[:digit:]` wird zurückgeführt auf die Methode `isDigit`().

   (e) `[:uppercase:]` wird zurückgeführt auf die Methode `isUppercase`().

   (f) `[:lowercase:]` wird zurückgeführt auf die Methode `isLowercase`().

3.3 Weitere Optionen

In diesem Abschnitt wollen wir die wichtigsten Optionen vorstellen, die im Options-Teil eine *JFlex*-Spezifikation angegeben werden können. Alle Optionen beginnen mit dem Zeichen "`%`", das am Zeilen-Anfang stehen muss.

1. `%char`

   Mit dieser Option wird das Mitzählen von Zeichen aktiviert. Wird diese Option angegeben, so steht die Variable `yychar` zur Verfügung. Diese Variable ist vom Typ `int` und gibt an, wieviele Zeichen bereits gelesen worden sind.

2. `%line`

   Mit dieser Option wird das Mitzählen von Zeilen aktiviert. Wird diese Option angegeben, so steht die Variable `yyline` zur Verfügung. Diese Variable ist vom Typ `int` und gibt an, wieviele Zeilen bereits gelesen worden sind.

3. `%column`

   Mit dieser Option wird mitgezählt, wieviele Zeichen seit dem letzten Zeilen-Umbruch gelesen worden sind. Diese Information wird in der Variablen `yycolumn` zur Verfügung gestellt.

4. `%cup`

   Diese Option spezifiziert, dass ein Scanner für den Parser-Generator CUP erstellt werden soll. Wir werden diese Option später benutzen, wenn wir mit *JFlex* und *Cup* einen Parser erzeugen.

3.4 Ein komplexeres Beispiel: Noten-Berechnung

In diesem Abschnitt diskutieren wir eine Anwendung von *JFlex*. Es geht dabei um die Auswertung von Klausuren. Bei der Korrektur einer Klausur lege ich eine Datei an, die das in dem in Abbildung 3.3 beispielhaft gezeigte Format besitzt.

```
1  Klausur: Algorithmen und Datenstrukturen
2  Kurs:    TIT09AID
3
4  Aufgaben:            1. 2. 3. 4. 5. 6.
5  Max Müller:          9 12 10  6  6  0
6  Dietmar Dumpfbacke:  4  4  2  0  -  -
7  Susi Sorglos:        9 12 12  9  9  6
```

Abbildung 3.3: Klausurergebnisse

1. Die erste Zeile enthält nach dem Schlüsselwort `Klausur` den Titel der Klausur.
2. Die zweite Zeile gibt den Kurs an.
3. Die dritte Zeile ist leer.
4. Die vierte Zeile gibt die Nummern der einzelnen Aufgaben an.
5. Danach folgt eine Tabelle. Jede Zeile dieser Tabelle listet die Punkte auf, die ein Student erzielt hat. Der Name des Studenten wird dabei am Zeilenanfang angegeben. Auf den Namen folgt ein Doppelpunkt und daran schließen sich dann Zahlen an, die angeben, wieviele Punkte bei den einzelnen Aufgaben erzielt wurden. Wurde eine Aufgabe nicht bearbeitet, so steht in der entsprechenden Spalte ein Bindestrich "-".

Das *JFlex*-Programm, das wir entwickeln werden, berechnet zunächst die Summe `sumPoints` aller Punkte, die ein Student erzielt hat. Aus dieser Summe wird dann nach der Formel

$$\texttt{note} = 7 - 6 \cdot \frac{\texttt{sumPoints}}{\texttt{maxPoints}}$$

die Note errechnet, wobei die Variable `maxPoints` die Punktzahl angibt, die für die Note 1,0 benötigt wird. Diese Zahl ist ein Argument, das dem Programm beim Start übergeben wird.

Abbildung 3.4 zeigt die *JFlex*-Spezifikation, aus der sich automatisch ein *Java*-Programm zur Noten-Berechnung erstellen läßt.

```
1  package Klausur;
2  %%
3  %class Noten
4  %int
5  %line
6  %unicode
7  %{
8      public int mMaxPoints = 0;
9      public int mSumPoints = 0;
10     public double note() {
11         return 7.0 - 6.0 * mSumPoints / mMaxPoints;
12     }
13     public void errorMsg() {
14         System.out.printf("invalid character '%s' at line %d\n",
15                           yytext(), yyline);
16     }
17     public static void main(String argv[]) {
18         Noten scanner = null;
19         try {
20             scanner = new Noten( new java.io.FileReader(argv[0]) );
21             scanner.mMaxPoints = new Integer(argv[1]);
22             scanner.yylex();
23         } catch (java.io.FileNotFoundException e) {
24             System.out.println("File not found : \""+argv[0]+"\"");
25         } catch (java.io.IOException e) {
26             System.out.println("IO error scanning file \""+argv[0]+"\"");
27             System.out.println(e);
28         } catch (Exception e) {
29             System.out.println("Second argument (maxpoints) missing?");
30             e.printStackTrace();
31         }
32     }
33 %}
34
35 ZAHL = 0|[1-9][0-9]*
36 NAME = [A-Za-zöäüÖÄÜß]+[ ][A-Za-zöäüÖÄÜß]+
37 %%
38
39 [A-Za-z]+:.*\n { /* skip header                          */ }
40 {NAME}/:       { System.out.print(yytext());
41                  mSumPoints = 0;                           }
42 :[ \t]+        { System.out.print(yytext());               }
43 {ZAHL}         { mSumPoints += new Integer(yytext());      }
44 -              { /* skip hyphens                          */ }
45 [ \t]          { /* skip white space                      */ }
46 ^[ \t]*\n      { /* skip empty line                       */ }
47 \n            { System.out.printf(" %3.1f\n", note());    }
48 .             { errorMsg();                               }
```

Abbildung 3.4: Berechnung von Noten mit Hilfe von *JFlex*.

1. Die `package`-Deklaration in Zeile 1 legt fest, dass die erzeugte Klasse Teil des Pakets `Klausur` ist.

2. Zeile 3 legt den Namen der Klasse fest.

3. In Zeile 4 spezifizieren wir durch die Option "`int`", dass die Scanner-Methode *yylex()* den Rückgabe-Wert `int` hat.

   Dies ist deswegen erforderlich, weil in einem Scanner, der nicht als `standalone` deklariert ist, dieser Rückgabe-Wert den Typ `Yytoken` hat. Die Klasse `Yytoken` wird aber von *JFlex* selber gar nicht definiert, denn *JFlex* erwartet, dass diese Klasse von dem Parser, an dem der Scanner angeschlossen wird, definiert wird. Da wir gar keinen Parser anschließen wollen, wäre der Typ `Yytoken` dann undefiniert. Um zu vermeiden, dass der Rückgabe-Wert der Methode `yylex()` den Typ `Yytoken` bekommt, könnten wir den Scanner auch als `standalone` deklarieren, aber dann hätten wir keine Möglichkeit mehr, die Methode `main()` anzugeben. Diese Methode benötigen wir aber, um die dem Programm als Argument übergebene Punktzahl einlesen zu können.

4. In den Zeilen 8 und 9 deklarieren wir die Variablen `mMaxPoints` und `mSumPoints` als Member-Variablen der erzeugten Klasse.

5. In der Methode `main()` erzeugen wir zunächst in Zeile 20 den Scanner und setzen dann in Zeile 21 die Variable `mMaxPoints` auf den beim Aufruf übergebenen Wert. Der Scanner wird durch den Aufruf in Zeile 22 gestartet.

6. In dem Definitions-Abschnitt werden zwei Abkürzungen definiert:

   (a) Zeile 35 enthält die Definition von `ZAHL`. Mit dieser Definition können wir später anstelle des regulären Ausdrucks

   ```
   0|[1-9][0-9]*
   ```

   kürzer "`{ZAHL}`" schreiben. Beachten Sie, dass der Name einer Abkürzung bei der Verwendung der Abkürzung in geschweiften Klammern eingefaßt werden muss.

   (b) Zeile 36 enthält die Definition von `NAME`. In dem regulären Ausdruck

   ```
   [A-Za-zöäüÖÄÜß]+[ ][A-Za-zöäüÖÄÜß]+
   ```

   wird festgelegt, dass ein Name großen und kleinen lateinischen Buchstaben sowie Umlauten besteht und das Vor- und Nachname durch ein Leerzeichen getrennt werden.

7. Der Regel-Abschnitt erstreckt sich von Zeile 39 – 48.

   (a) Die Regel in Zeile 39 dient dazu, die beiden Kopfzeilen der zu verarbeitenden Datei zu lesen. Diese Zeilen bestehen jeweils aus einem Wort, auf das ein Dopplepunkt folgt. Dahinter steht beliebiger Text, der mit einem Zeilenumbruch endet.

   (b) Die Regel in Zeile 40 liest den Namen eines Studenten, dem ein Doppelpunkt folgen muss. Da wir den Doppelpunkt mit dem Operator "`/`" von dem Namen abtrennen, ist der Doppelpunkt nicht Bestandteil des von dieser Regel gelesenen Textes. Dadurch können wir den Doppelpunkt in der nächsten Regel noch benutzen.

   Wenn wir einen Namen gelesen haben, geben wir diesen mit Hilfe eines `print`-Befehls aus. Sie sehen hier, dass wir mit Hilfe der Methode `yyprintf()` auf durch den regulären Ausdruck erkannten Text zugreifen können.

   Anschließend setzen wir die Variable `mSumPoints` auf 0. Dies ist erforderlich, weil diese Variable ja vorher noch die Punkte eines anderen Studenten enthalten könnte.

   (c) Die nächste Regel in Zeile 42 ließt die Leerzeichen und Tabulatoren ein, die auf den Doppelpunkt folgen und gibt diese aus. Dadurch erreichen wir, dass die Ausgabe der Noten genauso formatiert wird wie die Eingabe-Datei.

(d) Die Regel in Zeile 43 dient dazu, die Punkte, die der Student bei einer Aufgabe erreicht hat, einzulesen. Da die Zahl zunächst nur als String zur Verfügung steht, müssen wir diesen String in eine Zahl umwandeln. Dazu benutzen wir den Konstruktor der Klasse `Integer`. Anschließend wird diese Zahl dann zu der Summe der Punkte hinzuaddiert.

Die Verwendung der Funktion `atoi`() ist übrigens der Grund, weshalb wir die Header-Datei "`stdlib.h`" einbinden müssen.

(e) Für nicht bearbeitete Aufgaben enthält die Eingabe-Datei einen Bindestrich "`-`". Diese Bindestriche werden durch die Regel in Zeile 44 eingelesen und ignoriert. Daher ist das Kommando dieser Regel (bis auf den Kommentar) leer.

(f) In der gleichen Weise überlesen wir mit der Regel in Zeile 45 Leerzeichen und Tabulatoren, die nicht auf einen Doppelpunkt folgen.

(g) Die Regel in Zeile 46 dient dazu Zeilen einzulesen, die nur aus Leerzeichen und Tabulatoren bestehen.

(h) Wenn wir nun einen einzelnen Zeilenumbruch lesen, dann muss dieser von einer Zeile stammen, die die Punkte eines Studenten auflistet. In diesem Fall berechnen wir mit der Regel in Zeile 47 die erzielte Note und geben sie mit einer Stelle hinter dem Komma aus.

(i) Die bis hierhin vorgestellten Regeln ermöglichen es, eine syntaktisch korrekte Eingabe-Datei zu verarbeiten. Für den Fall, dass die Eingabe-Datei Syntaxfehler enthält, ist es sinnvoll, eine Fehlermeldung auszugeben, denn sonst könnte es passieren, dass auf Grund eines einfachen Tippfehlers eine falsche Note berechnet wird. Daher enthält Zeile 48 eine Default-Regel, die immer dann greift, wenn keine der anderen Regeln zum Zuge gekommen ist. Diese Regel liest ein einzelnes Zeichen und gibt eine Fehlermeldung aus. Diese Fehlermeldung enthält das gelesene Zeichen.

3.4.1 Zustände

Viele syntaktische Konstrukte lassen sich zwar im Prinzip mit regulären Ausdrücken beschreiben, aber die Ausdrücke, die benötigt werden, sind sehr unübersichtlich. Ein gutes Beispiel hierfür ist der reguläre Ausdruck zur Spezifikation von mehrzeilige `C`-Kommentaren, also Kommentaren der Form

`/*` $\cdots$ `*/`

Wenn wir hier einen regulären Ausdruck angeben möchten, der weder den Upto-Operator noch den Negations-Operator verwendet[1], so müßten wir den folgenden regulären Ausdruck verwenden:

```
\/\*([^*]|\*+[^*/])*\*+\/
```
(3.1)

Zunächst ist dieser Ausdruck schwer zu lesen. Das liegt vor allem daran, dass die Operator-Symbole "`/`" und "`*`" durch einen Backslash geschützt werden müssen. Aber auch die Logik, die hinter diesem Ausdruck steht, ist nicht ganz einfach. Wir analysieren die einzelnen Komponenten dieses Ausdrucks:

1. `\/\*`

   Hierdurch wird der String "`/*`", der den Kommentar einleitet, spezifiziert.

2. `([^*]|\*+[^*/])*`

   Dieser Teil spezifiziert alle Zeichen, die zwischen dem öffnenden String "`/*`" und einem schließenden String der Form "`*/`" liegen. Wie müssen sicherstellen, dass dieser Teil die Zeichenreihe "`*/`" nicht enthält, denn sonst würden wir in einer Zeile der Form

[1] Die meisten anderen Werkzeuge, die mit regulären Ausdrücken arbeiten, unterstützen weder den Negations-Operator noch den Upto-Operator.

```
/* first */ ++n; /* second */
```

den Befehl "`++n;`" für einen Teil des Kommentars halten. Der erste Teil des obigen regulären Ausdrucks "`[^*]`" steht für ein beliebiges von "`*`" verschiedenes Zeichen. Denn solange wir kein "`*`" lesen, kann der Text auch kein "`*/`" enthalten. Das Problem ist, dass das Innere eines Kommentars aber durchaus das Zeichen "`*`" enthalten kann, es darf nur kein "`/`" folgen. Daher spezifiziert die Alternative "`\*+[^*/]`" einen String, der aus beliebig vielen "`*`"-Zeichen besteht, auf die dann aber noch ein Zeichen folgen muss, dass sowohl von "`/`" als auch von "`*`" verschieden ist.

Der Ausdruck "`[^*]|\*+[^*/]`" spezifiziert jetzt also entweder ein Zeichen, das von "`*`" verschieden ist, oder aber eine Folge von "`*`"-Zeichen, auf die dann noch ein von "`/`" verschiedenes Zeichen folgt. Da solche Folgen beliebig oft vorkommen können, wird der ganze Ausdruck in Klammern eingefaßt und mit dem Quantor "`*`" dekoriert.

3. `\*+\/`

   Dieser reguläre Ausdruck spezifiziert das Ende des Kommentars. Es kann aus einer beliebigen positiven Anzahl von "`*`"-Zeichen bestehen, auf die dann noch ein "`/`" folgt. Wenn wir hier nur den Ausdruck "`\*\/`" verwenden würden, dann könnten wir Kommentare der Form

   ```
   /*** blah ***/
   ```

   nicht mehr erkennen, denn der unter 2. diskutierte reguläre Ausdruck akzeptiert nur Folgen von "`*`", auf die kein "`*`" folgt.

```
1  /**
2    remove C comments from a file
3  */
4
5  %%
6
7  %class Decomment
8  %standalone
9  %unicode
10
11 %%
12
13 \/\*([^*]|\*+[^*/])*\*+\/ { /* skip multi  line comments */ }
14 \/\/.*                    { /* skip single line comments */ }
```

Abbildung 3.5: Entfernung von Kommentaren aus einem C-Programm.

Abbildung 3.5 zeigt ein *JFlex*-Programm, das aus einem `C`-Programm sowohl einzeilige Kommentare der Form "`// ···`" als auch mehrzeilige Kommentare der Form "`/* ··· */` entfernt. Das Programm an sich ist zwar recht kurz, aber der reguläre Ausdruck zur Erkennung mehrzeiliger Kommentare ist sehr kompliziert und damit nur schwer zu verstehen. Es gibt zwei Möglichkeiten, um dieses Programm zu vereinfachen:

1. Wir könnten den oben bereits diskutierten Upto-Operator benutzen.
2. Alternativ können wir auch mit sogenannten *Start-Zuständen* arbeiten. Wir werden die letzte Möglichkeit jetzt vorführen.

Wir diskutieren diese Start-Zustände an Hand eines Beispiels: Wir wollen eine HTML-Datei in eine Text-Datei konvertieren. Die *JFlex*-Spezifikation, die in Abbildung 3.6 gezeigt wird, führt dazu die folgenden Aktionen durch:

1. Zunächst wird der Kopf der HTML-Datei, der in den Tags "`<head>`" und "`</head>`" eingeschlossen ist, entfernt.

2. Die Skripte, die in der HTML-Datei enthalten sind, werden ebenfalls entfernt.

3. Außerdem werden die HTML-Tags entfernt.

Zur Deklaration der verschiedenen Zustände wird das Schlüsselwort "`%xstate`" verwendet.

```
1   package Converter;
2
3   %%
4
5   %class Html2Txt
6   %standalone
7   %line
8   %unicode
9
10  %xstate header script
11  %%
12
13  "<head>"            { yybegin(header);         }
14  "<script"[^>\n]+">" { yybegin(script);         }
15  "<"[^>\n]+">"       { /* skip html tags */     }
16  [\n]+               { System.out.print("\n");  }
17                 { System.out.print(" ");   }
18  &auml;              { System.out.print("ä");   }
19  &ouml;              { System.out.print("ö");   }
20  &uuml;              { System.out.print("ü");   }
21  &Auml;              { System.out.print("Ä");   }
22  &Ouml;              { System.out.print("Ö");   }
23  &Uuml;              { System.out.print("Ü");   }
24  &szlig;             { System.out.print("ß");   }
25
26  <header>"</head>"   { yybegin(YYINITIAL);      }
27  <header>.|\n        { /* skip anything else */ }
28
29  <script>"</script>" { yybegin(YYINITIAL);      }
30  <script>.|\n        { /* skip anything else */ }
```

Abbildung 3.6: Transformation einer HTML-Datei in eine reine Text-Datei

Wir diskutieren jetzt die in Abbildung 3.6 gezeigte *JFlex*-Datei im Detail.

1. In Zeile 10 deklarieren wir die beiden *Zustände* `header` und `script` über das Schlüsselwort "`%xstate`" als *exklusive* Start-Zustände. Die allgemeine Syntax einer solchen Deklaration ist wie folgt:

   (a) Am Zeilen-Anfang einer Zustands-Deklaration steht der String "`%xstate`" oder "`%state`". Der String "`%xstate`" spezifiziert *exklusive* Zustände, der String "`%state`" spezifiziert *inklusive* Zustände. Den Unterschied zwischen diesen beiden Zustandsarten erklären wir später.

   (b) Darauf folgt eine Liste der Namen der deklarierten Zustände. Die Namen werden durch Leerzeichen getrennt.

2. In Zeile 3 haben wir den String "`<head>`" in doppelte Hochkommata eingeschlossen. Dadurch verlieren die Operator-Symbole "`<`" ihre Bedeutung. Dies ist eine allgemeine Möglichkeit, um Operator-Symbole in *JFlex* spezifizieren zu können. Wollen wir beispielsweise den String "`a*`" wörtlich erkennen, so können wir an Stelle von "`a\*`" auch klarer

   ```
   "a*"
   ```

   schreiben.

   Wird der String "`<head>`" erkannt, so wird in Zeile 13 die Aktion "`yybegin(header)`" ausgeführt. Damit wechselt der Scanner aus dem Default-Zustand "`YYINITIAL`", in dem der Scanner startet, in den oben deklarierten Zustand `header`. Da dieser Zustand als *exklusiver* Zustand deklariert worden ist, können jetzt nur noch solche Regeln angewendet werden, die mit dem Prefix "`<header>`" beginnen. Wäre der Zustand als *inklusiver* Zustand deklariert worden, so könnten auch solche Regeln verwendet werden, die nicht mit einem Zustand markiert sind. Solche Regeln sind implizit mit dem Zustand "`<YYINITIAL>`" markiert.

   Die Regeln, die mit dem Zustand "`header`" markiert sind, finden wir weiter unten in den Zeilen 26 und 27.

3. In Zeile 14 wechseln wir entsprechend in den Zustand "`script`" wenn wir ein öffnendes `Script`-Tag sehen.

4. In Zeile 15 werden alle restlichen Tags gelesen. Da die Aktion hier leer ist, werden diese Tags einfach entfernt.

5. In Zeile 16 ersetzen wir die Zeichenreihe "` `" durch ein Blank.

6. In den folgenden Zeilen werden die HTML-Darstellungen von Umlauten durch die entsprechenden Zeichen ersetzt.

7. Zeile 26 beginnt mit der Zustands-Spezifikation "`header`". Daher ist diese Regel nur dann aktiv, wenn der Scanner in dem Zustand "`header`" ist. Diese Regel sucht nach dem schließenden Tag "`</head>`". Wird dieses Tag gefunden, so wechselt der Scanner zurück in den Default-Zustand `YYINITIAL`, in dem nur die Regeln verwendet werden, die nicht mit einem Zustand markiert sind.

8. Zeile 27 enthält ebenfalls eine Regel, die nur im Zustand "`header`" ausgeführt wird. Diese Regel liest ein beliebiges Zeichen, welches nicht weiter verarbeitet wird und daher im Endeffekt verworfen wird.

9. Die Zeilen 29 und 30 enthalten entsprechende Regeln für den Zustand "`script`".

Aufgabe 3: Einige Programmiersprachen unterstützen geschachtelte Kommentare. Nehmen Sie an, dass Sie für eine Programmiersprache, bei der Kommentare der Form

```
/* ··· */
```

unterstützt werden, ein *JFlex*-Programm erstellen sollen, dass diese Kommentare aus einem gegebenen Programm entfernt. Nehmen Sie dabei an, dass solche Kommentare geschachtelt werden dürfen. Nehmen Sie weiter an, dass die Programmiersprache auch Strings enthält, die durch doppelte Anführungszeichen begrenzt werden. Falls die Zeichenfolgen "`/*`" und "`*/`" innerhalb eines Strings auftreten, sollen diese Zeichenfolgen nicht als Begrenzung eines Kommentars gewertet werden.

Kapitel 4

Endliche Automaten

Wir wenden uns nun der Frage zu, wie aus regulären Ausdrücken *Scanner* erzeugt werden können und klären damit die Frage, wie ein Werkzeug wie *JFlex* funktioniert. Dazu führen wir zunächst den Begriff des *deterministischen endlichen Automaten* (abgekürzt EA) ein. Für die Praxis sind deterministische endliche Automaten zu unhandlich, daher wird dieser Begriff zu dem Begriff der *nicht-deterministischen* endlichen Automaten (abgekürzt NEA) erweitert. Wir werden sehen, dass diese beiden Konzepte gleichmächtig sind: Für jeden nicht-deterministischen endlichen Automaten können wir einen deterministischen endlichen Automaten bauen, der das gleiche leistet. Anschließend zeigen wir dann, wie ein regulärer Ausdruck in einen EA übersetzt werden kann. Schließlich runden wir die Theorie ab indem wir zeigen, dass auch jeder EA zu einem regulärer Ausdruck äquivalent ist.

4.1 Deterministische endliche Automaten

Die endlichen Automaten, die wir in diesem Kapitel diskutieren wollen, haben die Aufgabe, einen String einzulesen und sollen dann entscheiden, ob dieser String ein Element der Sprache ist, die durch den Automaten definiert wird. Die Ausgabe dieser Automaten beschränkt sich also auf die beiden Werte `true` und `false`. Der wesentliche Aspekt eines endlichen Automaten ist, dass der Automat intern eine fest vorgegeben Anzahl von Zuständen hat, in denen er sich befinden kann. Die Arbeitsweise eines solchen Automaten ist dann wie folgt:

1. Anfangs befindet sich der Automat in einem speziellen Zustand, der als *Start-Zustand* bezeichnet wird.

2. In jedem Verarbeitungs-Schritt liest der Automat einen Buchstaben b des Eingabe-Alphabets Σ und wechselt in Abhängigkeit von b und dem aktuellen Zustand in den Folgezustand.

3. Eine Teilmenge aller Zustände wird als Menge der *akzeptierenden Zustände* ausgezeichnet. Das eingelesene Wort ist genau dann ein Element der vom EA akzeptierten Sprache, wenn sich der Automat nach dem Einlesen aller Buchstaben in einem akzeptierenden Zustand befindet.

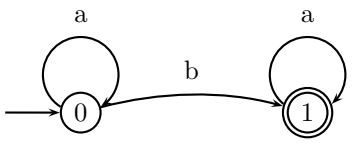


Abbildung 4.1: Ein einfacher endlicher Automat zur Erkennung der durch den regulären Ausdruck "$\texttt{a}^* \cdot \texttt{b} \cdot \texttt{a}^*$" definierten Sprache.

Am einfachsten können endliche Automaten grafisch dargestellt werden. Abbildung 4.1, zeigt einen einfachen endlichen Automaten, der die Strings erkennt, die durch den regulären Ausdruck

$$\mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*$$

beschrieben werden. Der Automat hat die beiden Zustände q_0 und q_1.

1. Zustand q_0 ist der Start-Zustand. In der Abbildung wird das durch den Pfeil, der auf diesen Zustand zeigt, kenntlich gemacht.

   Wenn in diesem Zustand der Buchstabe "a" gelesen wird, dann bleibt der Automat in dem Zustand q_0. Wird hingegen der Buchstabe "b" gelesen, dann wechselt der Automat in den Zustand q_1.

2. Zustand q_1 ist ein akzeptierender Zustand. In der Abbildung ist das dadurch zu erkennen, dass dieser Zustand von einem doppelten Kreis umgeben ist.

   Wenn in diesem Zustand der Buchstabe "a" gelesen wird, dann bleibt der Automat in dem Zustand q_1. In der Abbildung wird nicht gezeigt, was passiert, wenn der Automat im Zustand q_1 den Buchstaben "b" liest, der Folgezustand ist dann undefiniert.

   Allgemein sagen wir, dass ein Automat *stirbt*, wenn er in einem Zustand q einen Buchstaben b liest, für den kein Übergang definiert ist.

Formal definieren wir den Begriff des *endlichen Automaten* durch ein Tupel.

Definition 8 (EA) Ein *endlicher Automat* (abgekürzt EA) ist ein 5-Tupel

$$\langle Q, \Sigma, \delta, q_0, F \rangle.$$

Die einzelnen Komponenten haben die folgende Bedeutung:

1. Q ist die endliche *Menge der Zustände*.
2. Σ ist das *Eingabe-Alphabet*, also die Menge der Buchstaben, die der Automat als Eingabe verarbeitet.
3. $\delta : Q \times \Sigma \to Q \cup \{\Omega\}$

   ist die *Zustands-Übergangs-Funktion*. Für jeden Zustand q und für jeden Buchstaben c des Eingabe-Alphabets Σ gibt $\delta(q, c)$ den Zustand an, in den der Automat wechselt, wenn im Zustand q der Buchstabe c gelesen wird. Falls $\delta(q, c) = \Omega$ ist, dann sagen wir, dass der Automat *stirbt*, wenn im Zustand q der Buchstabe c gelesen wird.
4. $q_0 \in Q$ ist der *Start-Zustand*.
5. $F \subseteq Q$ ist die Menge der akzeptierenden Zustände. □

Beispiel: Der in Abbildung 4.1 gezeigte endliche Automat kann formal wie folgt beschrieben werden:

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle,$$

wobei gilt:

1. $Q = \{0, 1\}$,
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
3. $\delta = \big\{\langle 0, a\rangle \mapsto 0, \langle 0, b\rangle \mapsto 1, \langle 1, a\rangle \mapsto 1, \langle 1, b\rangle \mapsto \Omega\big\}$,
4. $q_0 = 0$,
5. $F = \{1\}$.

Um die von einem endlichen Automaten akzeptierte Sprache formal definieren zu können, verallgemeinern wir die Zustands-Übergangs-Funktion δ zu einer Funktion

$$\delta^* : Q \times \Sigma^* \to Q \cup \{\Omega\},$$

der als zweites Argument ein String übergeben werden kann. Die Definition von $\delta^*(q, w)$ erfolgt durch Induktion nach der Länge $|w|$ des Strings w.

I.A. $|w| = 0$: Dann gilt offenbar $w = \varepsilon$. Wir setzen

$$\delta^*(q, \varepsilon) := q,$$

denn wenn kein Buchstabe gelesen wird, ändert der Automat seinen Zustand auch nicht.

I.S. $|w| = n + 1$: In diesem Fall hat w die Form $w = cv$ mit $c \in \Sigma$, $v \in \Sigma^*$ und $|v| = n$. Wir setzen

$$\delta^*(q, cv) := \begin{cases} \delta^*\big(\delta(q, c), v\big) & \text{falls } \delta(q, c) \neq \Omega; \\ \Omega & \text{sonst.} \end{cases}$$

denn wenn der Automat das Wort cv liest, wird erst der Buchstabe c gelesen. Falls der Automat dabei in den Zustand $\delta(q, c)$ überwechselt, wird nun in diesem Zustand der Rest des Wortes, also v gelesen. Falls $\delta(q, c)$ undefiniert ist, ist natürlich auch $\delta^*(q, cv)$ undefiniert.

Definition 9 (akzeptierte Sprache, $L(A)$) Für einen endlichen Automaten $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ definieren wir die von A *akzeptierte Sprache* $L(A)$ wie folgt:

$$L(A) := \big\{s \in \Sigma^* \mid \delta^*(q_0, s) \in F\big\}. \qquad \square$$

Die akzeptierte Sprache eines endlichen Automaten besteht also aus all den Wörtern s, bei denen der endliche Automat beim Lesen des Wortes s von dem Start-Zustand in einen akzeptierenden Zustand übergeht.

Aufgabe 4: Geben Sie einen EA F an, so dass $L(F)$ aus genau den Wörtern der Sprache $\{a, b\}^*$ besteht, die den Teilstring "`aba`" enthalten.

Bemerkung: Im Internet finden Sie unter der Adresse

`http://fsme.sourceforge.net/`

das Werkzeug FSME, wobei der Name für *finite state machine environment* steht. Mit diesem Werkzeug können Sie das Verhalten eines gegebenen endlichen Automaten untersuchen. Abbildung 4.2 zeigt die graphische Oberfläche dieses Werkzeugs.

Vollständige Endliche Automaten Gelegentlich ist es hilfreich, wenn ein Automat A *vollständig* ist: Darunter verstehen wir einen Automaten

$$A = \langle Q, \Sigma, \delta, q_0, F\rangle,$$

für den die Funktion δ nie den Wert Ω als Ergebnis liefert, es gilt also

$$\delta : Q \times \Sigma \to Q.$$

Satz 10 Zu jedem endlichen deterministischen Automaten A gibt es einen vollständigen deterministischen Automaten $\hat{A}$, der die selbe Sprache akzeptiert wie der Automat A, es gilt also:

$$L(\hat{A}) = L(A).$$

Beweis: Der Automat A habe die Form

$$A = \langle Q, \Sigma, \delta, q_0, F\rangle$$

Die Idee ist, dass wir $\hat{A}$ dadurch definieren, dass wir zu der Menge Q einen neuen, sogenannten *toten* Zustand hinzufügen. Wenn es nun für einen Zustand $q \in Q$ und einen Buchstaben c keinen

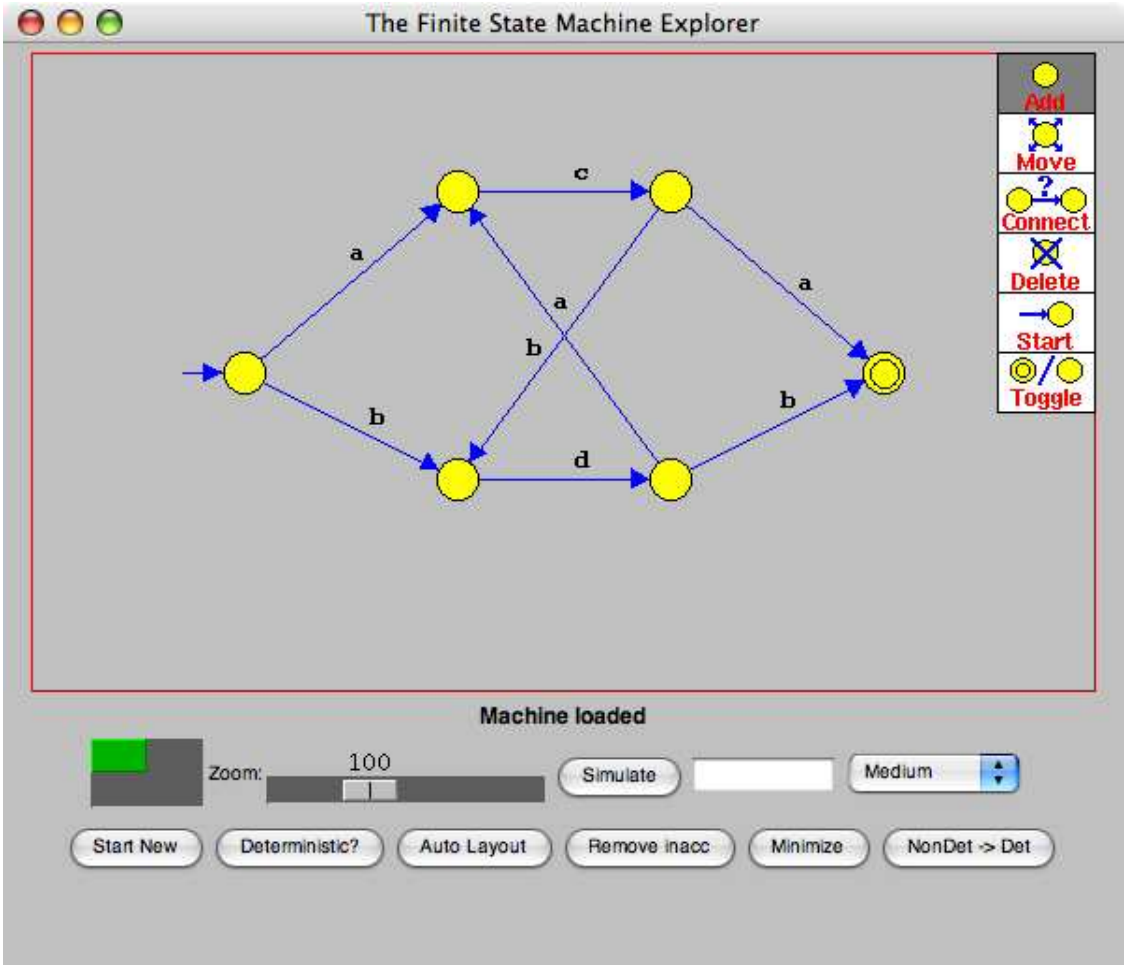


Abbildung 4.2: Das Werkzeug FSME.

Folge-Zustand in Q gibt, wenn also

$$\delta(q, c) = \Omega$$

gilt, dann geht der Automat in den toten Zustand über und bleibt auch bei allen folgenden Eingaben in diesem Zustand.

Die formale Definition von $\hat{A}$ verläuft wie folgt. Es bezeichne † einen *neuen* Zustand, also einen Zustand, der noch nicht in der Zustands-Menge Q vorkommt. Wir nennen † auch den *toten* Zustand. Dann definieren wir

1. $\hat{Q} := Q \cup \{\dagger\}$,

   der tote Zustand † wird also der Menge Q hinzugefügt.

2. $\hat{\delta} : \hat{Q} \times \Sigma \to \hat{Q}$,

   wobei die Werte der Funktion $\hat{\delta}$ wie folgt festgelegt werden:

   (a) $\delta(q, c) \neq \Omega \to \hat{\delta}(q, c) = \delta(q, c)$,
   wenn die Zustands-Übergangs-Funktion δ für den Zustand q und den Buchstaben c definiert ist und also einen Zustand als Ergebnis liefert, dann produziert $\hat{\delta}$ den selben Zustand.

   (b) $\delta(q, c) = \Omega \to \hat{\delta}(q, c) = \dagger$,

wenn die Zustands-Übergangs-Funktion δ für den Zustand q und den Buchstaben c als Ergebnis Ω liefert und also undefiniert ist, dann produziert $\hat{\delta}$ als Ergebnis den toten Zustand †.

(c) $\hat{\delta}(\dagger, c) = \dagger \quad$ für alle $c \in \Sigma$,
denn aus der Unterwelt gibt es kein Entkommen: Ist der Automat einmal in dem toten Zustand angekommen, so kann er in keinen anderen Zustand mehr gelangen, egal welches Zeichen eingelesen wird.

Damit können wir den Automaten $\hat{A}$ angeben:

$$\hat{A} = \langle \hat{Q}, \Sigma, \hat{\delta}, q_0, F \rangle.$$

Falls nun der Automat A einen String s einliest und dabei nicht stirbt, so ist das Verhalten von A und $\hat{A}$ identisch, es werden in beiden Automaten die selben Zustände durchlaufen. Falls der Automat A stirbt, dann geht der Automat $\hat{A}$ ersatzweise in den Zustand † und bleibt bei allen folgenden Eingaben in diesem Zustand. Damit ist klar, dass die von A und $\hat{A}$ akzeptierten Sprachen identisch sind. □

4.2 Nicht-deterministische endliche Automaten

Die im letzten Abschnitt eingeführten deterministischen Automaten sind für manche Anwendungen zu unhandlich, weil die Anzahl der Zustände zu groß wird. Abbildung 4.3 zeigt beispielsweise einen Automaten, der die Sprache akzeptiert, die durch den regulären Ausdruck

$$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$$

spezifiziert wird. Dieser Automat, den ich mit Hilfe des Werkzeugs *FSA*[1] erzeugt habe, enthält 8 Zustände und die grafische Darstellung ist ziemlich unübersichtlich.

Wir können einen solchen Automaten vereinfachen, wenn wir zulassen, dass der endliche Automat seinen Nachfolgezustand aus einer Menge von Zuständen, die vom aktuellen Zustand und dem gelesenen Buchstaben abhängt, frei wählen darf.

Abbildung 4.4 zeigt einen nicht-deterministischen endlichen Automaten, der die durch den regulären Ausdruck

$$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$$

beschriebene Sprache akzeptiert. Der Automat hat insgesamt 4 Zustände mit den Namen q_0, q_1, q_2 und q_3.

1. q_0 ist der Start-Zustand. Wird in diesem Zustand ein `a` gelesen, so bleibt der Automat im Zustand q_0. Wird hingegen der Buchstabe `b` gelesen, so hat der Automat die Wahl: Er kann entweder im Zustand q_0 bleiben, oder er kann in den Zustand q_1 wechseln.

2. Vom Zustand q_1 wechselt der Automat in den Zustand q_2, falls ein `a` oder ein `b` gelesen wurde.

3. Vom Zustand q_2 wechselt der Automat in den Zustand q_3, falls ein `a` oder ein `b` gelesen wurde.

4. Der Zustand q_3 ist der akzeptierende Zustand. Von diesem Zustand gibt es keinen Übergang mehr.

Der Automat aus Abbildung 4.4 ist nicht-deterministisch, weil er im Zustand q_0 bei der Eingabe von b den "richtigen" Nachfolge-Zustand raten muss. Betrachten wir eine mögliche *Berechnung* des Automaten zu der Eingabe "`abab`":

[1] FSA ist Toolbox, die verschiedene Werkzeuge zur Generierung und Manipulation endlicher Automaten enthält. Sie finden diese Software unter der Adresse `http://www.let.rug.nl/~vannoord/Fsa/`.

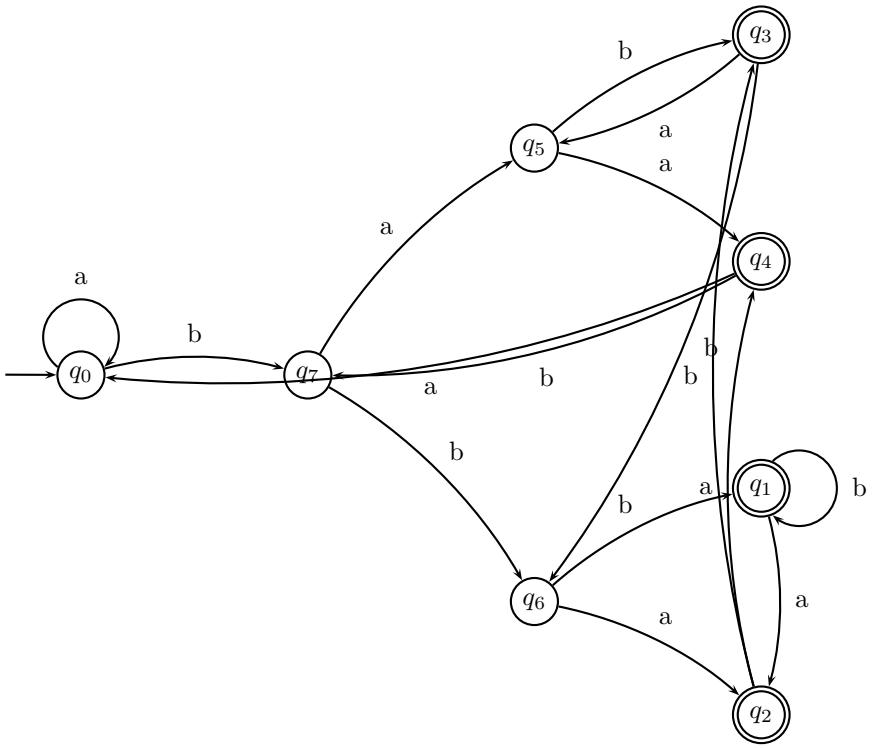


Abbildung 4.3: Ein endlicher Automat für die Sprache, die durch den regulären Ausdruck "$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$" definiert wird.

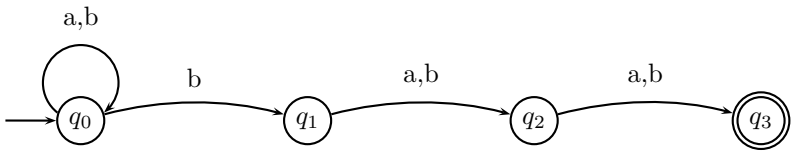


Abbildung 4.4: Ein Nicht-deterministischer Automat für die Sprache, die durch den regulären Ausdruck "$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$" definiert wird.

$$q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_1 \overset{a}{\mapsto} q_2 \overset{b}{\mapsto} q_3$$

Bei dieser Berechnung hat der Automat bei der Eingabe des ersten `b`'s richtig geraten, dass er in den Zustand q_1 wechseln muss. Wäre der Automat hier im Zustand q_0 verblieben, so könnte der akzeptierende Zustand q_3 nicht mehr erreicht werden:

$$q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{a}{\mapsto} q_0 \overset{b}{\mapsto} q_1$$

Hier ist der Automat am Ende der Berechnung im Zustand q_1, der nicht akzeptierend ist. Betrachten wir eine andere Berechnung, bei der das Wort "`bbbbb`" gelesen wird:

$$q_0 \overset{b}{\mapsto} q_0 \overset{b}{\mapsto} q_1 \overset{b}{\mapsto} q_2 \overset{b}{\mapsto} q_3 \overset{b}{\mapsto} \Omega$$

Hier ist der Automat zu früh in den Zustand q_1 gewechselt, was bei der Eingabe des letzten Zei-

chens zum Tode des Automaten führt. Wäre der Automat beim Lesen des zweiten Buchstabens b noch im Zustand q_0 geblieben, so hätte er das Wort "bbbbb" erkennen können:

$$q_0 \overset{b}{\mapsto} q_0 \overset{b}{\mapsto} q_0 \overset{b}{\mapsto} q_1 \overset{b}{\mapsto} q_2 \overset{b}{\mapsto} q_3$$

Auf den ersten Blick scheint es so zu sein, dass das Konzept der nicht-deterministischen endlichen Automaten wesentlich mächtiger ist als das Konzept der deterministischen endlichen Automaten, denn die nicht-deterministischen Automaten müssen ja geradezu hellseherische Fähigkeiten haben, um den richtigen Übergang raten zu können. Wir werden allerdings im nächsten Abschnitt sehen, dass die beiden Konzepte bei der Erkennung von Sprachen die gleiche Mächtigkeit haben. Dazu formalisieren wir den Begriff des nicht-deterministischen endlichen Automaten. Die Definition, die nun folgt, ist noch etwas weiter gefaßt als in der informalen Erklärung, die wir bisher gegeben haben, denn wir erlauben dem Automaten zusätzlich *spontane Übergänge*, sogenannte ε-*Transitionen*: Darunter verstehen wir einen Zustands-Übergang, bei dem kein Zeichen der Eingabe gelesen wird. Wir schreiben einen solchen spontanen Übergäng vom Zustand q_1 in den Zustand q_2 als

$$q_1 \overset{\varepsilon}{\mapsto} q_2.$$

Definition 11 (NEA) Ein *nicht-deterministischer endlicher Automat* (abgekürzt NEA) ist ein 5-Tupel

$$\langle Q, \Sigma, \delta, q_0, F \rangle,$$

so dass folgendes gilt:

1. Q ist die endliche Menge von Zuständen.
2. Σ ist das Eingabe-Alphabet.
3. δ ist eine Relation auf $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$, es gilt also
   $$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q.$$
4. $q_0 \in Q$ ist der Start-Zustand.
5. $F \subseteq Q$ ist die Menge der akzeptierenden Zustände. □

Falls $\langle q_1, \varepsilon, q_2 \rangle \in \delta$ ist, dann sagen wir, dass der Automat eine ε-*Transition* von dem Zustand q_1 in den Zustand q_2 hat.

Beispiel: Für den in Abbildung 4.4 auf Seite 34 gezeigten nicht-deterministischen endlichen Automaten A gilt

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle \quad \text{mit}$$

1. $Q = \{q_0, q_1, q_2, q_3\}$.
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.
3. $\delta = \big\{\langle q_0, \mathtt{a}, q_0\rangle, \langle q_0, \mathtt{b}, q_0\rangle, \langle q_0, \mathtt{b}, q_1\rangle, \langle q_1, \mathtt{a}, q_2\rangle, \langle q_1, \mathtt{b}, q_2\rangle, \langle q_2, \mathtt{a}, q_3\rangle, \langle q_2, \mathtt{b}, q_3\rangle\big\}$.
4. Der Start-Zustand ist q_0.
5. $F = \{q_3\}$, der einzige akzeptierende Zustand ist also q_3. □

Wir definieren den Begriff der *Konfiguration* eines NEA als ein Paar

$$\langle q, s \rangle$$

bestehend aus einem Zustand q und s ein String. Dabei ist q der Zustand, in dem der Automat sich befindet und s ist der Teil der Eingabe, der noch nicht gelesen worden ist. Im Falle von NEA definieren wir die Relation $\rightsquigarrow$ wie folgt: Es gilt

$$\langle q_1, cs \rangle \rightsquigarrow \langle q_2, s \rangle \quad \text{falls} \quad \langle q_1, c, q_2 \rangle \in \delta,$$

es gilt also $\langle q_1, cs\rangle \rightsquigarrow \langle q_2, s\rangle$ genau dann, wenn der Automat aus dem Zustand q_1 beim Lesen des Buchstabens c in den Zustand q_2 übergehen kann. Weiter haben wir

$$\langle q_1, s\rangle \rightsquigarrow \langle q_2, s\rangle \quad \text{falls} \quad \langle q_1, \varepsilon, q_2\rangle.$$

Hier werden die ε-Transitionen erfaßt. Auch ein NEA kann *sterben* und zwar dann, wenn es aus einem Zustand für den gelesen Buchstaben keinen Übergang gibt und wenn für diesen Zustand zusätzlich keine ε-Transition möglich ist:

$$\langle q_1, cs\rangle \rightsquigarrow \Omega \quad \text{g.d.w.} \quad \forall q_2 \in Q : \langle q_1, c, q_2\rangle \notin \delta \wedge \langle q_1, \varepsilon, q_2\rangle \notin \delta.$$

Wir beichnen den transitiven Abschluss der Relation $\rightsquigarrow$ mit $\rightsquigarrow^*$. Die von einem nicht-deterministischen endlicher Automaten A akzeptierte Sprache $L(A)$ ist definiert als

$$L(A) := \big\{ s \in \Sigma^* \mid \exists p \in F : \langle q_0, s\rangle \rightsquigarrow^* \langle p, \varepsilon\rangle \big\},$$

wobei q_0 den Start-Zustand und F die Menge der akzeptierenden Zustände bezeichnet. Ein Wort s liegt also genau dann in der Sprache $L(A)$, wenn von der Konfiguration $\langle q_0, s\rangle$ eine Konfiguration $\langle p, \varepsilon, \rangle$ erreichbar ist, bei der p ein akzeptierender Zustand ist.

Beispiel: Für den in Abbildung 4.4 gezeigten endlichen Automaten A besteht die akzeptierte Sprache $L(A)$ aus allen Worten $w \in \{a, b\}^*$, die mindestens die Länge drei haben und für die der drittletzte Buchstabe ein `b` ist.

Aufgabe 5: Geben Sie einen NEA A an, so dass $L(F)$ aus genau den Wörtern der Sprache $\{a, b\}^*$ besteht, die den Teilstring "`aba`" enthalten.

4.3 Äquivalenz von EA und NEA

In diesem Abschnitt zeigen wir, wie sich ein nicht-deterministischer endlicher Automat

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle$$

so in einen deterministischen endlichen Automaten $det(a)$ übersetzen läßt, dass die von beiden Automaten erkannte Sprache gleich ist, dass also

$$L(A) = L\big(det(A)\big)$$

gilt. Die Idee ist, dass der Automat $det(A)$ die Menge aller der Zustände berechnet, in denen sich der Automat A befinden könnte. Die Zustände des Automaten $det(A)$ sind also **Mengen** von Zuständen des ursprünglichen Automaten A. Eine solche Menge fasst alle die Zustände zusammen, in denen der nicht-deterministische Automat A sich befinden kann. Folglich ist eine Menge M von Zuständen des Automaten A ein akzeptierender Zustand des Automaten $det(A)$, wenn die Menge M einen akzeptierenden Zustand des Automaten A enthält.

Um diese Konstruktion von $det(A)$ angeben zu können, definieren wir zunächst zwei Hilfs-Funktionen. Wir beginnen mit dem sogenannten ε-Abschluss (engl. ε-closure). Die Funktion

$$ec : Q \to 2^Q$$

soll für jeden Zustand $q \in Q$ die Menge $ec(q)$ aller der Zustände berechnen, in die der Automat ausgehend von dem Zustand q mit Hilfe von ε-Transitionen übergehen kann. Formal definieren wir die Menge $ec(Q)$ indem wir induktiv festlegen, welche Elemente in der Menge $ec(q)$ enthalten sind.

I.A.: $q \in ec(q)$.

I.S.: $p \in ec(q) \wedge \langle p, \varepsilon, r \rangle \in \delta \;\rightarrow\; r \in ec(q)$.

Falls der Zustand p ein Element des ε-Abschlusses von p ist und es eine ε-Transition von p zu einem Zustand r gibt, dann ist auch r ein Element des ε-Abschlusses von q.

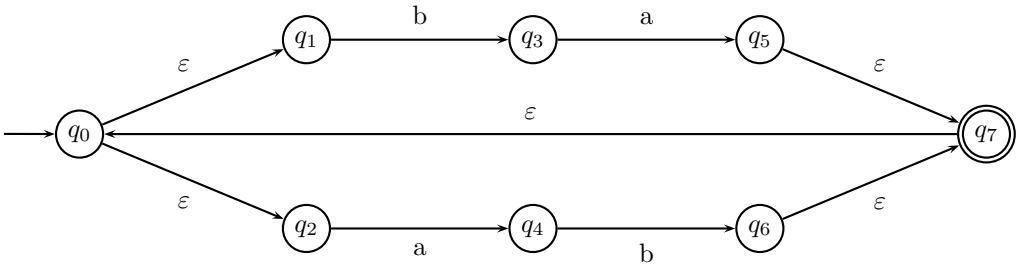


Abbildung 4.5: Nicht-deterministischer Automat mit ε-Transitionen.

Beispiel: Abbildung 4.5 zeigt einen nicht-deterministischen endlichen Automaten mit ε-Transitionen. Wir berechnen für alle Zustände den ε-Abschluss.

1. $ec(q_0) = \{q_0, q_1, q_2\}$,
2. $ec(q_1) = \{q_1\}$,
3. $ec(q_2) = \{q_2\}$,
4. $ec(q_3) = \{q_3\}$,

5. $ec(q_4) = \{q_4\}$,
6. $ec(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
7. $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,
8. $ec(q_7) = \{q_7, q_0, q_1, q_2\}$. □

Um den am Anfang des Abschnitts angegebenen nicht-deterministischen Automaten A in einen deterministischen Automaten $det(A)$ umwandeln zu können, transformieren wir die Relation δ in eine Funktion

$$\delta^* : Q \times \Sigma \to 2^Q,$$

wobei die Idee ist, dass $\delta^*(q, c)$ für einen Zustand q und einen Buchstaben c die Menge aller der Zustände berechnet, in denen der Automat A sich befinden kann, wenn er im Zustand q zunächst den Buchstaben c liest und anschließend eventuell noch einen oder auch mehrere ε-Transitionen durchführt. Formal erfolgt die Definition von δ^* durch die Formel

$$\delta^*(q_1, c) := \bigcup_{q_2 \in Q:\ \langle q_1, c, q_2 \rangle \in \delta} ec(q_2).$$

Diese Formel ist wie folgt zu lesen:

1. Wir berechnen für alle Zustände $q_2 \in Q$, die von dem Zustand q_1 durch Lesen des Buchstabens c erreicht werden können, den ε-Abschluss $ec(q_2)$.
2. Anschließend vereinigen wir alle diese Mengen $ec(q_2)$.

Beispiel: In Fortführung des obigen Beispiels erhalten wir beispielsweise:

1. $\delta^*(q_0, \mathtt{a}) = \{\}$,

   denn vom Zustand q_0 gibt es keine Übergänge mit dem Buchstaben `a`. Beachten Sie, dass wir bei der oben gegebenen Definition der Funktion δ^* die ε-Transitionen erst nach den Buchstaben-Transitionen durchgeführt werden.

2. $\delta^*(q_1, \mathtt{b}) = \{q_3\}$,

   denn vom Zustand q_1 geht der Automat beim Lesen von `b` in den Zustand q_3 über. Für den Zustand q_3 gibt es aber keine ε-Transitionen.

3. $\delta^*(q_3, \mathtt{a}) = \{q_5, q_7, q_0, q_1, q_2\}$,

   denn vom Zustand q_3 geht der Automat beim Lesen von `a` zunächst in den Zustand q_5 über. Von diesem Zustand aus sind dann die Zustände q_7, q_0, q_1 und q_2 durch ε-Transitionen erreichbar. □

Die Funktion δ^* überführt einen Zustand in eine Menge von Zuständen. Da der zu entwickelnde endliche Automat $det(A)$ mit *Mengen von Zuständen* arbeiten wird, benötigen wir eine Funktion, die *Mengen von Zuständen* in *Mengen von Zuständen* überführt. Wir verallgemeinern daher die Funktion δ^* zu der Funktion

$$\Delta^* : 2^Q \times \Sigma \to 2^Q$$

so, dass $\Delta^*(M, c)$ für eine Menge von Zuständen M und einen Buchstaben c die Menge aller der Zustände berechnet, in denen der Automat A sich befinden kann, wenn er sich zunächst in einem Zustand aus der Menge M befunden hat, dann der Buchstabe c gelesen wurde und anschließend eventuell noch ε-Transitionen ausgeführt werden. Die formale Definition lautet

$$\Delta^*(M, c) := \bigcup_{q \in M} \delta^*(q, c).$$

Diese Formel ist einfachzu verstehen: Für jeden Zustand $q \in M$ berechnen wir zunächst die Menge aller Zustände, in denen sich der Automat nach Lesen von c und eventuellen ε-Transitionen befinden kann. Die so erhaltenen Mengen vereinigen wir.

Beispiel: In Fortführung des obigen Beispiels erhalten wir beispielsweise:

1. $\Delta^*(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_4\}$,
2. $\Delta^*(\{q_0, q_1, q_2\}, \mathtt{b}) = \{q_3\}$,
3. $\Delta^*(\{q_3\}, \mathtt{a}) = \{q_7, q_0, q_1, q_2\}$,
4. $\Delta^*(\{q_3\}, \mathtt{b}) = \{\}$,
5. $\Delta^*(\{q_4\}, \mathtt{a}) = \{\}$,
6. $\Delta^*(\{q_4\}, \mathtt{b}) = \{q_7, q_0, q_1, q_2\}$. □

Wir haben nun alles Material zusammen, um den nicht-deterministischen endlichen Automaten A in einen deterministischen endlichen Automaten $det(A)$ überführen zu können. Wir definieren

$$det(A) = \big\langle 2^Q, \Sigma, \Delta^*, ec(q_0), \widehat{F} \big\rangle.$$

1. Die Menge der Zustände von $det(A)$ besteht aus der Menge aller Teilmengen der Zustände von A, ist also gleich der Potenz-Menge 2^Q.

   Wir werden später sehen, dass von diesen Teilmengen nicht alle wirklich benötigt werden: Die Teilmengen fassen ja Zustände zusammen, in denen der Automat A sich ausgehend von dem Start-Zustand nach der Eingabe eines bestimmten Wortes befinden kann. In der Regel können nicht alle Kombinationen von Zuständen auch tatsächlich erreicht werden.

2. An dem Eingabe-Alphabet ändert sich nichts, denn der neue Automat $det(A)$ soll ja die selbe Sprache erkennen wie der Automat A.

3. Die oben definierte Funktion Δ^* gibt an, wie sich Zustands-Mengen bei Eingabe eines Zeichens ändern.

4. Der Start-Zustand des Automaten $det(A)$ ist die Menge aller der Zustände, die von dem Start-Zustand q_0 des Automaten A durch ε-Transitionen erreichbar sind.

5. Wir definieren die Menge $\widehat{F}$ der akzeptierenden Zustände, als die Menge der Teilmengen von Q, die einen akzeptierenden Zustands enthalten, wir setzen also

   $$\widehat{F} := \big\{ M \in 2^Q \mid M \cap F \neq \{\} \big\}.$$

Beispiel: Wir zeigen, wie sich der in Abbildung 4.4 auf Seite 34 gezeigte nicht-deterministische Automat A in einen deterministischen Automaten $det(A)$ transformieren läßt.

1. Da $ec(q_0) = \{q_0\}$ gilt, besteht der Start-Zustand des deterministischen Automaten aus der Menge, die nur den Knoten q_0 enthält:

   $$S_0 := ec(q_0) = \{q_0\}.$$

   Wir bezeichnen den Start-Zustand mit S_0.

2. Von dem Zustand q_0 geht F_1 beim Lesen von `a` in den Zustand q_0 über. Also gilt

   $$\Delta^*(\{q_0\}, \mathtt{a}) = \{q_0\} = S_0.$$

3. Von dem Zustand q_0 geht F_1 beim Lesen von `b` in den Zustand q_0 oder q_1 über. Also gilt

   $$S_1 := \Delta^*(\{q_0\}, \mathtt{b}) = \{q_0, q_1\}.$$

4. Wir haben $\delta(q_0, \mathtt{a}) = \{q_0\}$ und $\delta(q_1, \mathtt{a}) = \{q_2\}$. Daher folgt
$$S_2 := \Delta^*(\{q_0, q_1\}, \mathtt{a}) = \{q_0, q_2\}.$$
5. Wir haben $\delta(q_0, \mathtt{b}) \in \{q_0, q_1\}$ und $\delta(q_1, \mathtt{b}) = \{q_2\}$. Daher folgt
$$S_4 := \Delta^*(\{q_0, q_1\}, \mathtt{b}) = \{q_0, q_1, q_2\}$$
6. $S_3 := \Delta^*(\{q_0, q_2\}, \mathtt{a}) = \{q_0, q_3\}$.
7. $S_5 := \Delta^*(\{q_0, q_2\}, \mathtt{b}) = \{q_0, q_1, q_3\}$.
8. $S_6 := \Delta^*(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_0, q_2, q_3\}$.
9. $S_7 := \Delta^*(\{q_0, q_1, q_2\}, \mathtt{b}) = \{q_0, q_1, q_2, q_3\}$.
10. $\Delta^*(\{q_0, q_3\}, \mathtt{a}) = \{q_0\} = S_0$.
11. $\Delta^*(\{q_0, q_3\}, \mathtt{b}) = \{q_0, q_1\} = S_1$.
12. $\Delta^*(\{q_0, q_1, q_3\}, \mathtt{a}) = \{q_0, q_2\} = S_2$.
13. $\Delta^*(\{q_0, q_1, q_3\}, \mathtt{b}) = \{q_0, q_1, q_2\} = S_4$.
14. $\Delta^*(\{q_0, q_2, q_3\}, \mathtt{a}) = \{q_0, q_3\} = S_3$.
15. $\Delta^*(\{q_0, q_2, q_3\}, \mathtt{b}) = \{q_0, q_1, q_3\} = S_5$.
16. $\Delta^*(\{q_0, q_1, q_2, q_3\}, \mathtt{a}) = \{q_0, q_2, q_3\} = S_6$.
17. $\Delta^*(\{q_0, q_1, q_2, q_3\}, \mathtt{b}) = \{q_0, q_1, q_2, q_3\} = S_7$.

Damit haben wir alle Zustände des deterministischen Automaten. Zur besseren Übersicht fassen wir die Definitionen der einzelnen Zustände des deterministischen Automaten noch einmal zusammen:

$$S_0 = \{q_0\},\ S_1 = \{q_0, q_1\},\ S_2 = \{q_0, q_2\},\ S_3 = \{q_0, q_3\},\ S_4 = \{q_0, q_1, q_2\},$$
$$S_5 = \{q_0, q_1, q_3\},\ S_6 = \{q_0, q_2, q_3\},\ S_7 = \{q_0, q_1, q_2, q_3\}$$

und setzen schließlich

$$\widehat{Q} := \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}.$$

Wir fassen die Übergangs-Funktion Δ^* in einer Tabelle zusammen:

Δ^*	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
a	S_0	S_2	S_3	S_0	S_6	S_2	S_3	S_6
b	S_1	S_4	S_5	S_1	S_7	S_4	S_5	S_7

Als letztes stellen wir fest, dass die Mengen S_3, S_5, S_6 und S_7 den akzeptierenden Zustand q_3 enthalten. Also setzen wir

$$\widehat{F} := \{S_3, S_5, S_6, S_7\}.$$

Damit können wir nun einen deterministischen endlichen Automaten $det(A)$ angeben, der die selbe Sprache akzeptiert wie der nicht-deterministische Automat A:

$$det(A) := \langle \widehat{Q}, \Sigma, \Delta^*, S_0, \widehat{F} \rangle.$$

Abbildung 4.6 zeigt den Automaten $det(A)$. Wir erkennen, dass dieser Automat 8 verschiedene Zustände besitzt. Der ursprünglich gegebene nicht-deterministische Automat A hat 4 Zustände, für die Zustands-Menge des nicht-deterministischen Automaten gilt $Q = \{q_0, q_1, q_2, q_3\}$. Die Potenz-Menge 2^Q besteht aus 16 Elementen. Wieso hat dann der Automat $det(A)$ nur 8 und nicht $2^4 = 16$ Zustände? Der Grund ist, dass von dem Start-Zustand q_0 nur solche Mengen von Zuständen erreichbar sind, die den Zustand q_0 enthalten, denn egal ob `a` oder `b` eingegeben wird, kann der Automat A von q_0 immer wieder in den Zustand q_0 zurück wechseln. Daher muss jede Menge von Zuständen, die von q_0 erreichbar ist, selbst wieder q_0 enthalten. Damit entfallen als Zustände von $det(A)$ alle Mengen von 2^Q, die q_0 nicht enthalten, wodurch die Zahl der Zustände gegenüber der maximal möglichen Anzahl halbiert wird.

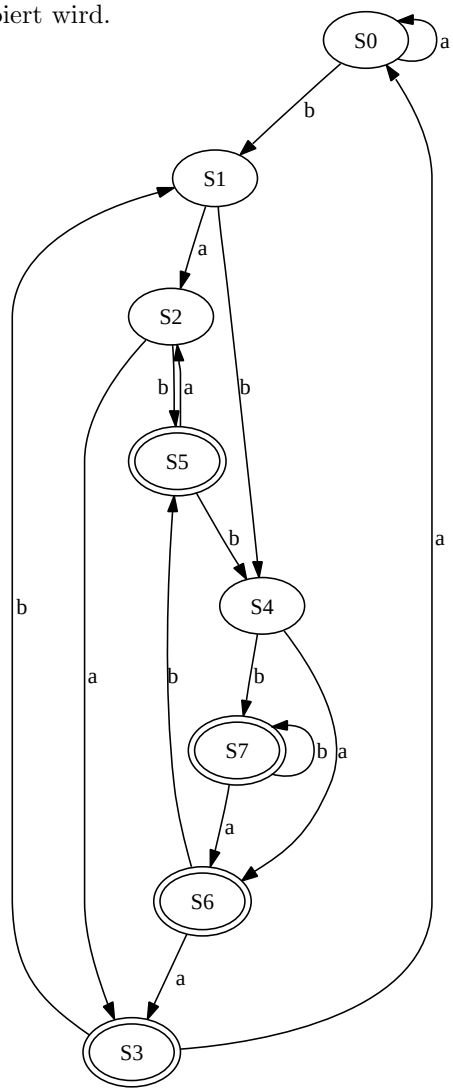


Abbildung 4.6: Der deterministische Automat $det(A)$.

Aufgabe 6: Transformieren Sie den in Abbildung 4.5 auf Seite 37 gezeigten endlichen Automaten in einen äquivalenten deterministischen endlichen Automaten.

4.4 Übersetzung regulärer Ausdrücke in NEA

In diesem Abschnitt konstruieren wir zu einem gegebenen regulären Ausdruck r einen nichtdeterministischen endlichen Automaten $A(r)$, der die durch r spezifizierte Sprache akzeptiert:

$$L\big(A(r)\big) = L(r)$$

Die Konstruktion von $A(r)$ erfolgt durch eine Induktion nach dem Aufbau des regulären Ausdrucks r. Der konstruierte Automat $A(r)$ wird die folgenden Eigenschaften haben, die wir bei der Konstruktion komplexerer Automaten ausnutzen werden:

1. $A(r)$ hat keine Transition zu dem Start-Zustand. Bezeichnen wir den Start-Zustand mit $\mathit{start}\big(A(r)\big)$, so gilt also:

   $$q = \mathit{start}\big(A(r)\big) \;\rightarrow\; \forall p \in Q : \big(\langle p, \varepsilon, q\rangle \notin \delta \;\wedge\; \forall c \in \Sigma : \langle p, c, q\rangle \notin \delta\big).$$

2. $A(r)$ hat genau einen akzeptierenden Zustand, den wir mit $\mathit{accept}\big(A(r)\big)$ bezeichnen. Außerdem gibt es keine Übergänge, die von diesem akzeptierenden Zustand ausgehen:

   $$p = \mathit{accept}\big(A(r)\big) \;\rightarrow\; \forall q \in Q : \big(\langle p, \varepsilon, q\rangle \notin \delta \;\wedge\; \forall c \in \Sigma : \langle p, c, q\rangle \notin \delta\big).$$

Im folgenden nehmen wir an, dass Σ das Alphabet ist, das bei der Konstruktion des regulären Ausdrucks r verwendet wurde. Dann wird $A(r)$ wie folgt definiert.

1. Den Automaten $A(\emptyset)$ definieren wir als

   $$A(\emptyset) = \langle \{q_0, q_1\}, \Sigma, \{\}, q_0, \{q_1\}\rangle$$

   Abbildung 4.7: Der Automat $A(\emptyset)$.

   Abbildung 4.7 zeigt den Automaten, der die durch $\emptyset$ spezifizierte Sprache akzeptiert. Der Automat besteht nur aus dem Start-Zustand q_0 und dem akzeptierenden Zustand q_1. Die Relation δ ist leer, der Automat hat keinerlei Zustands-Übergänge und akzeptiert die leere Sprache, denn $L(\emptyset) = \{\}$.

2. Den Automaten $A(\varepsilon)$ definieren wir als

   $$A(\varepsilon) = \langle \{q_0, q_1\}, \Sigma, \{\langle q_0, \varepsilon, q_1\rangle\}, q_0, \{q_1\}\rangle$$

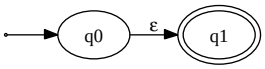
   

   Abbildung 4.8: Der Automat $A(\varepsilon)$.

   Abbildung 4.8 zeigt den Automaten, der die durch ε spezifizierte Sprache akzeptiert. Der Automat besteht nur aus dem Start-Zustand q_0 und dem akzeptierenden Zustand q_1. Von dem Zustand q_0 gibt es eine ε-Transition zu dem Zustand q_1. Damit akzeptiert der Automat das leere Wort und sonst nichts.

3. Für einen Buchstaben $c \in \Sigma$ definieren wir den Automaten $A(c)$ durch

   $$A(c) = \langle \{q_0, q_1\}, \Sigma, \{\langle q_0, c, q_1\rangle\}, q_0, \{q_1\}\rangle$$

   Abbildung 4.9 zeigt den Automaten, der die durch den Buchstaben c spezifizierte Sprache akzeptiert. Der Automat besteht aus dem Start-Zustand q_0 und dem akzeptierenden Zustand q_1. Von dem Zustand q_0 gibt es eine Transition zu dem Zustand q_1, die beim Lesen des Buchstabens c benutzt wird. Damit akzeptiert der Automat das Wort, das nur aus dem Buchstaben c besteht und sonst nichts.

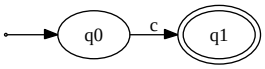


Abbildung 4.9: Der Automat $A(c)$.

4. Um den Automaten $A(r_1 \cdot r_2)$ definieren zu können, nehmen wir zunächst an, dass die Zustände der Automaten $A(r_1)$ und $A(r_2)$ verschieden sind. Dies können wir immer erreichen, indem wir die Zustände des Automaten $A(r_2)$ umbennen. Wir nehmen nun an, dass $A(r_1)$ und $A(r_2)$ die folgende Formen haben:

   (a) $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\rangle$,
   (b) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\rangle$.

   Damit können wir den endlichen Automaten $A(r_1 \cdot r_2)$ aus den beiden Automaten $A(r_1)$ und $A(r_2)$ zusammenbauen: Dieser Automat ist gegeben durch

   $$\langle Q_1 \cup Q_2, \Sigma, \{\langle q_2, \varepsilon, q_3\rangle\} \cup \delta_1 \cup \delta_2, q_0, \{q_4\}\rangle$$

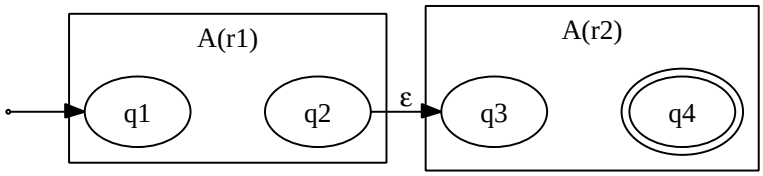


Abbildung 4.10: Der Automat $A(r_1 \cdot r_2)$.

   Abbildung 4.10 zeigt den Automaten $A(r_1 \cdot r_2)$.

   Statt der ε-Transition von q_2 nach q_3 können wir die beiden Zustände q_2 und q_3 auch identifizieren. Das hat den Vorteil, dass der resultierende Autmat dann etwas kleiner wird. Bei den praktischen Übungen werden wir daher diese Zustände immer identifizieren.

5. Um den Automaten $A(r_1+r_2)$ definieren zu können, nehmen wir wieder an, dass die Zustände der Automaten $A(r_1)$ und $A(r_2)$ verschieden sind. Wir nehmen weiter an, dass $A(r_1)$ und $A(r_2)$ die folgende Formen haben:

   (a) $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_3\}\rangle$,
   (b) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_2, \{q_4\}\rangle$.

   Damit können wir den Automaten $A(r_1 + r_2)$ aus den beiden Automaten $A(r_1)$ und $A(r_2)$ zusammenbauen: Dieser Automat ist gegeben durch

   $$\langle \{q_0, q_5\} \cup Q_1 \cup Q_2, \Sigma, \{\langle q_0, \varepsilon, q_1\rangle, \langle q_0, \varepsilon, q_2\rangle, \langle q_3, \varepsilon, q_5\rangle, \langle q_4, \varepsilon, q_5\rangle\} \cup \delta_1 \cup \delta_2, q_0, \{q_5\}\rangle$$

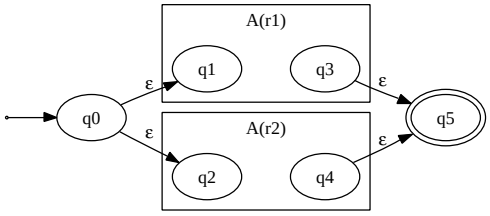


Abbildung 4.11: Der Automat $A(r_1 + r_2)$.

   Abbildung 4.11 zeigt den Automaten $A(r_1+r_2)$. Wir sehen, dass zusätzlich zu den Zuständen der beiden Automaten $A(r_1)$ und $A(r_2)$ noch zwei weitere Zustände hinzukommen:

(a) q_0 ist der Start-Zustand des Automaten $A(r_1 + r_2)$,

(b) q_5 ist der einzige akzeptierende Zustand des Automaten $A(r_1 + r_2)$.

Gegenüber den Zustands-Übergänge der Automaten $A(r_1)$ und $A(r_2)$ kommen noch vier ε-Transitionen hinzu:

(a) Von dem neuen Start-Zustand q_0 gibt es jeweils eine ε-Transition zu den Start-Zuständen q_1 und q_2 der Automaten $A(r_1)$ und $A(r_2)$.

(b) Von den akzeptierenden Zuständen q_3 und q_4 der Automaten $A(r_1)$ und $A(r_2)$ gibt es jeweils eine ε-Transition zu dem akzeptierenden Zustand q_5.

Um den so definierten Automaten zu vereinfachen, können wir einerseits die drei Zustände q_0, q_1 und q_2 und andererseits die drei Zustände q_3, q_4 und q_5 identifizieren.

6. Um den Automaten $A(r^*)$ für den Kleene-Abschluss r^* definieren zu können, schreiben wir $A(r)$ als

$$A(r) = \langle Q, \Sigma, \delta, q_1, \{q_2\}\rangle,$$

Damit können wir den $A(r^*)$ aus dem Automaten $A(r)$ konstruieren: Dieser Automat $A(r^*)$ ist gegeben durch

$$\langle \{q_0, q_3\} \cup Q, \Sigma, \{\langle q_0, \varepsilon, q_1\rangle, \langle q_2, \varepsilon, q_1\rangle, \langle q_0, \varepsilon, q_3\rangle, \langle q_2, \varepsilon, q_3\rangle\} \cup \delta, q_0, \{q_3\}\rangle$$

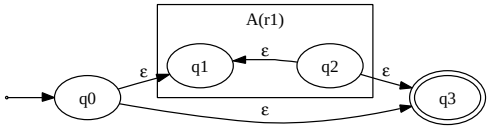


Abbildung 4.12: Der Automat $A(r^*)$.

Abbildung 4.12 zeigt den Automaten $A(r^*)$. Wir sehen, dass zusätzlich zu den Zuständen des Automaten $A(r)$ noch zwei weitere Zustände hinzukommen:

(a) q_0 ist der Start-Zustand des Automaten $A(r^*)$,

(b) q_3 ist der einzige akzeptierende Zustand des Automaten $A(r^*)$.

Zu den Zustands-Übergänge des Automaten $A(r)$ kommen vier ε-Transitionen hinzu:

(a) Von dem neuen Start-Zustand q_0 gibt es jeweils eine ε-Transition zu den Zuständen q_1 und q_3.

(b) Von q_2 gibt es eine ε-Transition zurück zu dem Zustand q_1.

(c) Von q_2 gibt es eine ε-Transition zu dem Zustand q_3.

Achtung: Wenn wir bei diesem Automaten versuchen würden, die Zustände q_0 und q_1 bzw. q_2 und q_3 zu identifizieren, dann funktioniert das in diesem Abschnitt beschriebene Verfahren in bestimmten Fällen nicht mehr. Diese Zustände dürfen also **nicht** identifiziert werden!

Aufgabe 7: Bilden Sie einen nicht-deterministischen endlichen Automaten, der die durch den regulären Ausdruck

$$(\mathtt{a} + \mathtt{b}) \cdot \mathtt{a}^* \cdot \mathtt{b}$$

spezifizierte Sprache erkennt.

4.5 Übersetzung eines EA in einen regulären Ausdruck

Wir runden die Theorie ab indem wir zeigen, dass sich zu jedem deterministischen endlichen Automaten A ein regulärer Ausdruck $r(A)$ angeben läßt, der die selbe Sprache spezifiziert, die von dem Automaten A akzeptiert wird, für den also

$$L\big(r(a)\big) = L(A)$$

gilt. Der Automat A habe die Form

$$A = \langle \{q_0, q_1, \cdots, q_n\}, \Sigma, \delta, q_0, F \rangle.$$

Für jedes Paar von Zuständen $\langle p_1, p_2 \rangle \in Q \times Q$ definieren wir einen regulären Ausdruck $r(p_1, p_2)$. Die Idee bei dieser Definition ist, dass der reguläre Ausdruck $r(p_1, p_2)$ alle die Strings w spezifiziert, die den Automaten A von dem Zustand p_1 in den Zustand p_2 überführen, formal gilt:

$$L\big(r(p_1, p_2)\big) = \big\{w \in \Sigma^* \mid \langle p_1, w\rangle \rightsquigarrow^* \langle p_2, \varepsilon\rangle\big\}$$

Die Definition der regulären Ausdrücke erfolgt über einen Trick: Wir definieren für $k = 0, \cdots, n+1$ reguläre Ausdrücke $r^{(k)}(p_1, p_2)$. Der reguläre Ausdruck beschreibt gerade die Strings, die den Automaten A von dem Zustand p_1 in den Zustand p_2 überführen, ohne dass dabei zwischendurch ein Zustand aus der Menge

$$Q_k := \big\{q_i \mid i \in \{k, \cdots, n\}\big\} = \{q_k, \cdots, q_n\}$$

besucht wird. Die Menge Q_k enthält also nur die Zustände, deren Index größer oder gleich k ist. Formal definieren wir dazu die dreistellige Relation

$$\mapsto_k \subseteq (Q \times \Sigma^* \times Q).$$

Für zwei Zustände $p, q \in Q$ und einen String w soll

$$p \overset{w}{\mapsto}_k q$$

genau dann gelten, wenn der Automat A von dem Zustand p beim Lesen des Wortes w in den Zustand q übergeht, ohne dabei zwischendurch in einen Zustand aus der Menge Q_k zu wechseln. Mit "zwischendurch" ist hier gemeint, dass die Zustände p und q sehr wohl in der Menge Q_k liegen können. Die formale Definition der Relation $p \overset{w}{\mapsto}_k q$ erfolgt durch eine Induktion nach der Länge des Wortes w:

I.A.: Im Induktions-Anfang haben wir zwei Fälle:

(a) $p \overset{\varepsilon}{\mapsto}_k p$,
denn mit dem leeren Wort kann von p aus nur der Zustand p erreicht werden.

(b) $\delta(p, c) = q \Rightarrow p \overset{c}{\mapsto}_k q$,
denn wenn der Automat beim Lesen des Buchstabens c von dem Zustand p direkt in den Zustand q übergeht, dann werden zwischendurch keine Zustände aus Q_k besucht, denn es werden überhaupt keine Zustände zwischendurch besucht.

I.S.: $\delta(p, c) = q \wedge q \neq r \wedge w \neq \varepsilon \wedge q \notin Q_k \wedge q \overset{w}{\mapsto}_k r \Rightarrow p \overset{cw}{\mapsto}_k r$.

Wenn der Automat A von dem Zustand p durch Lesen des Buchstabens c in einen Zustand $q \notin Q_k$ übergeht und wenn der Automat dann von diesem Zustand q beim Lesen von w in den Zustand r übergehen kann, ohne dabei Zustände aus Q_k zu benutzen, dann geht der Automat beim Lesen von cw aus dem Zustand p in den Zustand r über, ohne zwischendurch in Zustände aus Q_k zu wechseln.

Damit können wir nun für alle $k = 0, \cdots, n+1$ die regulären Ausdrücke $r^{(k)}(p_1, p_2)$ definieren. Wir werden diese regulären Ausdrücke so definieren, dass hinterher

$$L\big(r^{(k)}(p_1, p_2)\big) = \big\{w \in \Sigma^* \mid p_1 \overset{w}{\mapsto}_k p_2\big\}$$

gilt. Die Definition der regulären Ausdrücke $r^{(k)}(p_1, p_2)$ erfolgt durch eine Induktion nach k.

I.A.: $k = 0$.

Dann gilt $Q_0 = Q$, die Menge Q_0 enthält also alle Zustände und damit dürfen wir, wenn wir vom Zustand p_1 in den Zustand p_2 übergehen, zwischendurch überhaupt keine Zustände besuchen.

Wir betrachten zunächst den Fall $p_1 \neq p_2$. Dann kann $p_1 \overset{w}{\mapsto}_0 p_2$ nur dann gelten, wenn w aus einem einzigen Buchstaben besteht. Es sei

$$\{c_1, \cdots, c_l\} := \{c \in \Sigma \mid \delta(p_1, c) = p_2\}$$

die Menge aller Buchstaben, die den Zustand p_1 in den Zustand p_2 überführen. Falls diese Menge nicht leer ist, setzen wir

$$r^{(0)}(p_1, p_2) := c_1 + \cdots + c_l.$$

Ist die obige Menge leer, so gibt es keinen direkten Übergang von p_1 nach p_2 und wir setzen

$$r^{(0)}(p_1, p_2) := \emptyset.$$

Wir betrachten jetzt den Fall $p_1 = p_2$. Definieren wir wieder

$$\{c_1, \cdots, c_l\} := \{c \in \Sigma \mid \delta(p_1, c) = p_1\}$$

als die Menge aller Buchstaben, die den Zustand p_1 in sich selbst überführen, so können wir in dem Fall, dass diese Menge nicht leer ist,

$$r^{(0)}(p_1, p_1) := c_1 + \cdots + c_l + \varepsilon,$$

setzen. Ist die obige Menge leer, so gibt es nur den Übergang mit dem leeren Wort von p_1 nach p_1 und wir setzen

$$r^{(0)}(p_1, p_1) := \varepsilon.$$

I.S.: $k \mapsto k + 1$.

Bei dem Übergang von $r^{(k)}(p_1, p_2)$ zu $r^{(k+1)}(p_1, p_2)$ dürfen wir zusätzlich den Zustand q_k benutzen, denn q_k ist das einzige Element der Menge Q_k, das nicht in der Menge q_{k+1} enthalten ist. Wird ein String w gelesen, der den Zustand p_1 in den Zustand p_2 überführt, ohne dabei zwischendurch in einen Zustand aus der Menge Q_{k+1} zu wechseln, so gibt es zwei Möglichkeiten:

(a) Es gilt bereits $p_1 \overset{w}{\mapsto}_k p_2$.

(b) Der String w kann so in mehrere Teile $w_1 s_1 \cdots s_l w_2$ aufgeteilt werden dass gilt

- $p_1 \overset{w_1}{\mapsto}_k q_k$,
  von dem Zustand p_1 gelangt der Automat also beim Lesen von w_1 zunächst in den Zustand q_k, wobei zwischendurch der Zustand q_k nicht benutzt wird.
- $q_k \overset{s_i}{\mapsto}_k q_k \quad$ für alle $i = \{1, \cdots, l\}$,
  von dem Zustand q_k wechselt der Automat beim Lesen der Teilstrings s_i wieder in den Zustand q_k.
- $q_k \overset{w_2}{\mapsto}_k p_2$,
  schließlich wechselt der Automat von dem Zustand q_k in den Zustand p_2, wobei der Rest w_2 gelesen wird.

Daher definieren wir

$$r^{(k+1)}(p_1, p_2) := r^{(k)}(p_1, p_2) + r^{(k)}(p_1, q_k) \cdot \left(r^{(k)}(q_k, q_k)\right)^* \cdot r^{(k)}(q_k, p_2).$$

Dieser Ausdruck kann wie folgt gelesen werden: Um von p_1 nach p_2 zu kommen, ohne den Zustand q_k zu benutzen, kann der Automat entweder direkt von p_1 nach p_2 gelangen, ohne q_k zu benutzen, was dem Ausdruck $r^{(k)}(p_1, p_2)$ entspricht, oder aber der Automat wechselt von p_1 ein erstes Mal in den Zustand q_k, was den Ausdruck $r^{(k)}(p_1, q_k)$ erklärt, wechselt dann beliebig oft von q_k nach q_k, was den Ausdruck $\left(r^{(k)}(q_k, q_k)\right)^*$ erklärt und wechselt schließlich von q_k in den Zustand p_2, wofür der Ausdruck $r^{(k)}(q_k, p_2)$ steht.

Nun haben wir alles Material zusammen, um die Ausdrücke $r(p_1, p_2)$ definieren zu können. Wir setzen

$$r(p_1, p_2) := r^{(n+1)}(p_1, p_2).$$

Dieser reguläre Ausdruck beschreibt die Wörter, die den Automaten von dem Zustand p_1 in den Zustand p_2 überführen, ohne dass der Automat dabei in einen Zustand der Menge Q_{n+1} wechselt. Nun gilt aber

$$Q_{n+1} = \{q_i | i \in \{0, \cdots, n\} \wedge i \geq n + 1\} = \{\},$$

die Menge ist also leer! Folglich werden durch den regulären Ausdruck $r^{(n+1)}(p_1, p_2)$ überhaupt keine Zustände ausgeschlossen: Der Ausdruck beschreibt also genau die Strings, die den Zustand p_1 in den Zustand p_2 überführen.

Um nun einen regulären Ausdruck konstruieren zu können, der die Sprache des Automaten A beschreibt, schreiben wir die Menge F der akzeptierenden Zustände von A als

$$F = \{t_1, \cdots, t_m\}$$

und definieren den regulären Ausdruck $r(A)$ als

$$r(A) := r(q_0, t_1) + \cdots + r(q_0, t_m).$$

Dieser Ausdruck beschreibt genau die Strings, die den Automaten A aus dem Start-Zustand in einen der akzeptierenden Zustände überführen. □

Damit sehen wir jetzt, dass die Konzepte "*deterministischer endlicher Automat*" und "*regulärer Ausdruck*" äquivalent sind.

1. Jeder deterministische endliche Automat kann in einen äquivalenten regulären Ausdruck übersetzt werden.

2. Jeder reguläre Ausdruck kann in einen äquivalenten nicht-deterministischen endlichen Automaten transformiert werden.

3. Ein nicht-deterministischer endlicher Automat läßt sich durch die Teilmengen-Konstruktion in einen endlichen Automaten überführen.

Aufgabe 8: Konstruieren Sie für den in Abbildung 4.1 gezeigten endlichen Automaten einen äquivalenten regulären Ausdruck.

Lösung: Der Automat hat die Zustände 0 und 1. Wir berechnen zunächst die regulären Ausdrücke $r^{(k)}(i,j)$ für alle $i,j \in \{0,1\}$ der Reihe nach für die Werte $k = 0$, 1 und 2:

1. Für $k = 0$ finden wir:
   (a) $r^{(0)}(0,0) = \mathtt{a} + \varepsilon$,
   (b) $r^{(0)}(0,1) = \mathtt{b}$,
   (c) $r^{(0)}(1,0) = \emptyset$,
   (d) $r^{(0)}(1,1) = \mathtt{a} + \varepsilon$.
2. Für $k = 1$ haben wir:
   (a) Für $r^{(1)}(0,0)$ finden wir:
   $$\begin{array}{rcl} r^{(1)}(0,0) & = & r^{(0)}(0,0) + r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,0) \\ & = & r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \end{array}$$
   wobei wir im letzten Schritt die für reguläre Ausdrücke allgemeingültige Gleichung
   $$r + r \cdot r^* \cdot r = r \cdot r^*$$
   verwendet haben. Setzen wir für $r^{(0)}(0,0)$ den oben gefundenen Ausdruck $\mathtt{a} + \varepsilon$ ein, so erhalten wir
   $$r^{(1)}(0,0) = (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^*.$$
   Wegen $(\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* = \mathtt{a}^*$ haben wir insgesamt
   $$r^{(1)}(0,0) = \mathtt{a}^*.$$
   (b) Für $r^{(1)}(0,1)$ finden wir:
   $$\begin{array}{rcl} r^{(1)}(0,1) & = & r^{(0)}(0,1) + r^{(0)}(0,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,1) \\ & = & \mathtt{b} + (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \cdot \mathtt{b} \\ & = & \mathtt{b} + \mathtt{a}^* \cdot \mathtt{b} \\ & = & \mathtt{a}^* \cdot \mathtt{b} \end{array}$$
   (c) Für $r^{(1)}(1,0)$ finden wir:
   $$\begin{array}{rcl} r^{(1)}(1,0) & = & r^{(0)}(1,0) + r^{(0)}(1,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,0) \\ & = & \emptyset + \emptyset \cdot (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon) \\ & = & \emptyset \end{array}$$
   (d) Für $r^{(1)}(1,1)$ finden wir
   $$\begin{array}{rcl} r^{(1)}(1,1) & = & r^{(0)}(1,1) + r^{(0)}(1,0) \cdot \big(r^{(0)}(0,0)\big)^* \cdot r^{(0)}(0,1) \\ & = & (\mathtt{a} + \varepsilon) + \emptyset \cdot (\mathtt{a} + \varepsilon)^* \cdot \mathtt{b} \\ & = & (\mathtt{a} + \varepsilon) + \emptyset \\ & = & \mathtt{a} + \varepsilon \end{array}$$
3. Für $k = 2$ erhalten wir:
   (a) Für $r^{(2)}(0,0)$ finden wir
   $$\begin{array}{rcl} r^{(2)}(0,0) & = & r^{(1)}(0,0) + r^{(1)}(0,1) \cdot \big(r^{(1)}(1,1)\big)^* \cdot r^{(1)}(1,0) \\ & = & \mathtt{a}^* + \mathtt{a}^* \cdot b \cdot (\mathtt{a} + \varepsilon)^* \cdot \emptyset \\ & = & \mathtt{a}^* \end{array}$$

(b) Für $r^{(2)}(0,1)$ finden wir

$$\begin{aligned} r^{(2)}(0,1) &= r^{(1)}(0,1) + r^{(1)}(0,1) \cdot \big(r^{(1)}(1,1)\big)^* \cdot r^{(1)}(1,1) \\ &= \mathtt{a}^* \cdot b + \mathtt{a}^* \cdot b \cdot (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon) \\ &= \mathtt{a}^* \cdot b + \mathtt{a}^* \cdot b \cdot \mathtt{a}^* \\ &= \mathtt{a}^* \cdot b \cdot \mathtt{a}^* \end{aligned}$$

(c) Für $r^{(2)}(1,0)$ finden wir

$$\begin{aligned} r^{(2)}(1,0) &= r^{(1)}(1,0) + r^{(1)}(1,1) \cdot \big(r^{(1)}(1,1)\big)^* \cdot r^{(1)}(1,0) \\ &= \emptyset + (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon)^* \cdot \emptyset \\ &= \emptyset \end{aligned}$$

(d) Für $r^{(2)}(1,1)$ finden wir

$$\begin{aligned} r^{(2)}(1,1) &= r^{(1)}(1,1) + r^{(1)}(1,1) \cdot \big(r^{(1)}(1,1)\big)^* \cdot r^{(1)}(1,1) \\ &= r^{(1)}(1,1) \cdot \big(r^{(1)}(1,1)\big)^* \\ &= (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \\ &= \mathtt{a}^* \end{aligned}$$

Damit können wir den regulären Ausdruck $r(A)$ angeben:

$$r(A) = r^{(2)}(0,1) = \mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*.$$

Dieses Ergebnis, das wir jetzt so mühevoll abgeleitet haben, hätten wir auch durch einen einfachen Blick auf den Automaten erhalten können, aber die oben gezeigte Rechnung formalisiert das, was der geübte Betrachter mit einem Blick sieht und das Verfahren hat den Vorteil, dass es sich implementieren läßt. □

4.6 Minimierung endlicher Automaten

In diesem Abschnitt zeigen wir ein Verfahren, mit dem die Anzahl der Zustände eines deterministischen endlichen Automaten

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle$$

minimiert werden kann. Ohne Beschränkung der Allgemeinheit wollen wir dabei voraussetzen, dass der Automat A vollständig ist. Wir suchen dann einen deterministischen endlichen Automaten

$$A^- = \langle Q^-, \Sigma, \delta^-, q_0, F^- \rangle,$$

der die selbe Sprache akzeptiert wie der Automat A, für den also

$$L(A^-) = L(A)$$

gilt und für den die Anzahl der Zustände der Menge Q^- minimal ist. Um diese Konstruktion durchfüren zu können, müssen wir etwas ausholen. Zunächst erweitern wir die Funktion

$$\delta : Q \times \Sigma \to Q$$

zu einer Funktion $\hat{\delta}$, die als zweites Argument nicht nur einen Buchstaben sondern auch einen String akzeptiert:

$$\hat{\delta} : Q \times \Sigma^* \to Q.$$

Der Funktions-Aufrug $\delta(q, s)$ soll den Zustand p berechnen, in den der Automat A gelangt, wenn der Automat im Zustand q den String s verarbeitet. Die Definition von $\hat{\delta}(q, s)$ erfolgt durch Induktion über die Länge des Strings s:

I.A.: $\hat{\delta}(q, \varepsilon) = q$,

I.S.: $\hat{\delta}(q, cs) = \hat{\delta}\big(\delta(q, c), s\big)$, falls $c \in \Sigma$ und $s \in \Sigma^*$.

Da die Funktion $\hat{\delta}$ eine Verallgemeinerung der Funktion δ ist, werden wir in der Notation nicht zwischen δ und $\hat{\delta}$ unterscheiden und einfach nur δ schreiben.

Offensichtlich können wir in einem endlichen Automaten $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ alle die Zustände $p \in Q$ entfernen, die vom Start-Zustand aus nicht *erreichbar* sind. Dabei heißt ein Zustand p *erreichbar* genau dann, wenn es einen String $w \in \Sigma^*$ gibt, so dass

$$\delta(q_0, w) = p$$

gilt. Wir wollen im folgenden daher voraussetzen, dass alle Zustände des betrachten endlichen Automaten vom Start-Zustand aus erreichbar sind.

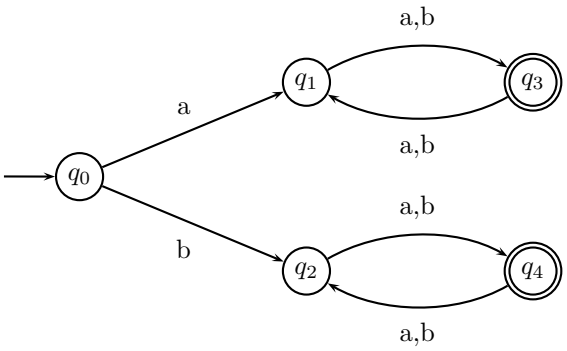


Abbildung 4.13: Ein endlicher Automat mit äquivalenten Zustände, die nicht gleichwertig sind.

Im Allgemeinen können wir einen Automaten dadurch minimieren, dass wir bestimmte Zustände identifizieren. Betrachten wir beispielsweise den in Abbildung 4.13 gezeigten Automaten, so können wir dort die Zustände q_1 und q_2 sowie q_3 und q_4 identifizieren, ohne dass sich dadurch die Sprache des Automaten ändert. Die zentrale Idee bei der Minimierung eines Automaten besteht darin, dass wir uns überlegen, welche Zustände wir auf keinen Fall identifizieren können und einfach alle anderen Zustände identifizieren.

Definition 12 (Unterscheidbar) Gegeben sei ein endlicher Automat $A = \langle Q, \Sigma, \delta, q_0, F\rangle$. Zwei Zustände $p_1, p_2 \in Q$ heißen *unterscheidbar* genau dann, wenn es einen String $s \in \Sigma^*$ gibt, so dass einer der beiden folgenden Fälle vorliegt:

1. $\delta(p_1, s) \in F$ und $\delta(p_2, s) \notin F$.
2. $\delta(p_1, s) \notin F$ und $\delta(p_2, s) \in F$. □

Zwei unterscheidbare Zustände p_1 und p_2 sind offenbar nicht gleichwertig, denn wenn wir diese Zustände identifizieren würden, würde sich die von dem Automaten erkannte Sprache ändern. Wir definieren nun eine Äquivalenz-Relation $\sim$ auf der Menge Q der Zustände. Für zwei Zustände $p_1, p_2 \in Q$ setzen wir

$$p_1 \sim p_2 \quad \text{g.d.w.} \quad \forall s \in \Sigma^* : \delta(p_1, s) \in F \leftrightarrow \delta(p_2, s) \in F,$$

die beiden Zustände p_1 und p_2 sind also bereits dann äquivalent, wenn sie nicht unterscheibar sind. Die Behauptung ist nun die, dass wir alle solchen Zustände identifizieren können. Die Identifikation zweier Zustände p_1 und p_2 erfolgt so, dass wir einerseits den Zustand p_2 aus der Menge Q der Zustände entfernen und andererseits die Funktion δ so abändern, dass an Stelle des Wertes p_2 nun immer p_1 zurück gegeben wird.

Es bleibt die Frage zu klären, wie wir feststellen können, welche Zustände unterscheidbar sind. Eine Möglichkeit besteht darin, eine Menge V von Paaren von Zustände anzulegen. Wir fügen das

Paar $\langle p, q\rangle$ in die Menge V ein, wenn wir erkannt haben, dass p und q unterscheidbar sind. Wir erkennen p und q als unterscheidbar, wenn es einen Buchstaben $c \in \Sigma$ und zwei Zustände s und t gibt, so dass gilt

$$\delta(p, c) = s,\ \delta(q, c) = t \text{ und } \langle s, t\rangle \in V.$$

Dieser Algorithmus läßt sich in einer ersten Version in den folgenden zwei Schritten programieren:

1. Zunächst initialisieren wir V mit alle den Paaren $\langle p, q\rangle$, für die entweder p ein akzeptierender Zustand und q kein akzeptierender Zustand ist, oder umgekehrt q ein akzeptierender Zustand und p kein akzeptierender Zustand ist, denn ein akzeptierender Zustand kann durch den leeren String ε von einem nicht-akzeptierenden Zustand unterschieden werden:

   $$V := \big\{\langle p, q\rangle \in Q \times Q \mid (p \in F \wedge q \notin F) \vee (p \notin F \wedge q \in F)\big\}$$

2. Solange wir ein neues Paar $\langle p, q\rangle \in Q \times Q$ finden, für dass es einen Buchstaben c gibt, so dass die Zustäne $\delta(p, c)$ und $\delta(q, c)$ bereits unterscheidbar sind, fügen wir dieses Paar zur Menge V hinzu:

   ```
   while (∃⟨p, q⟩ ∈ Q × Q : ∃c ∈ Σ : ⟨δ(p, c), δ(q, c)⟩ ∈ V ∧ ⟨p, q⟩ ∉ V) {
      V := V ∪ { ⟨p, q⟩, ⟨q, p⟩ };
   }
   ```

Haben wir alle Paare $\langle p, q\rangle$ von unterscheidbaren Zuständen gefunden, so können wir anschließend alle Zustände p und q identifizieren, die nicht unterscheidbar sind, für die also $\langle p, q\rangle \notin V$ gilt. Es läßt sich zeigen, dass der so konstruierte Automat tatsächlich minimal ist.

Beispiel: Wir betrachten den in Abbildung 4.13 gezeigten endlichen Automaten und wenden den oben skizzierten Algorithmus auf diesen Automaten an. Wir bedienen uns dazu einer Tabelle, deren Spalten und Zeilen mit den verschiedenen Zuständen durchnummeriert sind. Wenn wir erkannt haben, dass die Zustände i und j unterscheidbar sind, so fügen wir in dieser Tabelle in der i-ten Zeile und der j-ten Spalte ein Kreuz $\times$ ein. Da mit den Zuständen i und j auch die Zustände j und i unterscheidbar sind, fügen wir außerdem in der j-ten Zeile und der i-ten Spalte ein Kreuz $\times$ ein.

1. Im ersten Schritt erkennen wir, dass die beiden akzeptierenden Zustände q_3 und ${}_44$ von allen nicht-akzeptierenden Zuständen unterscheidbar sind. Also sind die Paare $\langle q_0, q_3\rangle$, $\langle q_0, q_4\rangle$, $\langle q_1, q_3\rangle$, $\langle q_1, q_4\rangle$, $\langle q_2, q_3\rangle$ und $\langle q_2, q_4\rangle$ unterscheidbar. Damit hat die Tabelle nun die folgende Gestalt:

	q_0	q_1	q_2	q_3	q_4
q_0				$\times$	$\times$
q_1				$\times$	$\times$
q_2				$\times$	$\times$
q_3	$\times$	$\times$	$\times$		
q_4	$\times$	$\times$	$\times$		

2. Als nächstes erkennen wir, dass die Zustände q_0 und q_1 unterscheidbar sind, denn es gilt

   $$\delta(q_0, a) = q_1, \quad \delta(q_1, a) = q_3 \quad \text{und} \quad q_1 \not\sim q_3.$$

   Genauso sehen wir, dass die Zustände 0 und 2 unterscheidbar sind, denn es gilt

   $$\delta(q_0, b) = q_2, \quad \delta(q_2, b) = q_4 \quad \text{und} \quad q_2 \not\sim q_4.$$

   Tragen wir $q_0 \not\sim q_1$ und $q_0 \not\sim q_2$ in die Tabelle ein, so hat diese jetzt die folgende Gestalt:

	q_0	q_1	q_2	q_3	q_4
q_0		×	×	×	×
q_1	×			×	×
q_2	×			×	×
q_3	×	×	×		
q_4	×	×	×		

3. Nun finden wir keine weiteren Paare von unterscheidbaren Zuständen mehr, denn wenn wir das Paar $\langle q_1, q_2 \rangle$ betrachten, sehen wir

   $$\delta(q_1, \mathtt{a}) = q_3 \quad \text{und} \quad \delta(q_2, \mathtt{a}) = q_4,$$

   aber da die Zustände 3 und 4 bisher nicht unterscheidbar sind, liefert dies kein neues unterscheidbares Paar. Genausowenig liefert

   $$\delta(q_1, \mathtt{b}) = q_3 \quad \text{und} \quad \delta(q_2, \mathtt{b}) = q_4,$$

   ein neues unterscheidbares Paar. Jetzt bleiben noch die beiden Zustände q_3 und q_4. Hier finden wir

   $$\delta(q_3, c) = 1 \quad \text{und} \quad \delta(q_4, c) = 2 \quad \text{für alle } c \in \{\mathtt{a}, \mathtt{b}\}$$

   und da die Zustände q_1 und q_2 bisher nicht als unterscheidbar bekannt sind, haben wir keine neuen unterscheidbaren Zustände gefunden. Damit können wir die äquivalenten Zustände aus der Tabelle ablesen, es gilt:

   (a) $q_1 \sim q_2$
   (b) $q_3 \sim q_4$

   Abbildung 4.14 zeigt den entsprechenden reduzierten endlichen Automaten.

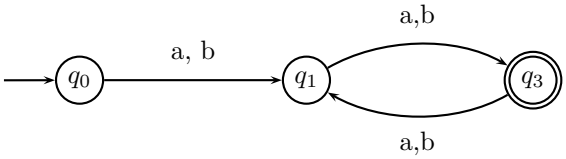


Abbildung 4.14: Der reduzierte endliche Automat.

Aufgabe 9: Konstruieren Sie den minimalen deterministischen endlichen Automaten, der die Sprache $L\big(a \cdot (b \cdot a)^*\big)$ erkennt. Gehen Sie dazu in folgenden Schritten vor:

1. Berechnen Sie einen nicht-deterministischen endlichen Automaten, der diese Sprache erkennt.
2. Transformieren Sie diesen Automaten in einen deterministischen Automaten.
3. Minimieren Sie die Zahl der Zustände dieses Automaten mit dem oben angegebenen Algorithmus.

Historisches Die Äquivalenz der durch reguläre Ausdrücke definierten Sprachen zu den Sprachen, die von endlichen Automaten akzeptiert werden, wurde von Stephen C. Kleene (1909 – 1994) im Jahre 1956 gezeigt [Kle56].

Kapitel 5

Die Theorie regulärer Sprachen

Ist Σ ein Alphabet, so bezeichen wir eine Sprache $L \subseteq \Sigma^*$ dann als eine *reguläre Sprache*, wenn es einen regulären Ausdruck r gibt, so dass L die durch diesen Ausdruck spezifizierte Sprache ist, wenn also

$$L = L(r)$$

gilt. Im letzten Kapitel hatten wir gesehen, dass die regulären Sprachen genau die Sprachen sind, die von einem endlichen Automaten erkannt werden können. In diesem Kapitel zeigen wir zunächst, dass reguläre Sprachen bestimmte Abschluss-Eigenschaften haben:

1. Die Vereinigung $L_1 \cup L_2$ zweier regulärer Sprachen L_1 und L_2 ist ebenfalls eine reguläre Sprache.
2. Der Durchschnitt $L_1 \cap L_2$ zweier regulärer Sprachen L_1 und L_2 ist eine reguläre Sprache.
3. Das Komplement $\Sigma^* \backslash L$ einer regulären Sprache ist wieder eine reguläre Sprache.

Als Anwendung der Abschluss-Eigenschaften zeigen wir anschließend, wie die Äquivalenz zweier regulärer Audrücke geprüft werden kann. Anschließend diskutieren wir die Grenzen regulärer Sprachen. Dazu beweisen wir das *Pumping-Lemma*, mit dem wir beispielsweise zeigen können, dass die Sprache

$$\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$$

nicht regulär ist.

5.1 Abschluss-Eigenschaften regulärer Sprachen

In diesem Abschnitt zeigen wir, dass reguläre Sprachen unter den Boole'schen Operationen *Vereinigung*, *Durchschnitt* und *abgeschlossen* sind. Wir beginnen mit der Vereinigung.

Satz 13 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Vereinigung $L_1 \cup L_2$ eine reguläre Sprache.

Beweis: Da L_1 und L_2 reguläre Sprachen sind, gibt es reguläre Ausdrücke r_1 und r_2, so dass

$$L_1 = L(r_1) \quad \text{und} \quad L_2 = L(r_2)$$

gilt. Wir definieren $r := r_1 + r_2$. Offenbar gilt

$$L(r) = L(r_1 + r_2) = L(r_1) \cup L(r_2) = L_1 \cup L_2.$$

Damit ist klar, dass $L_1 \cup L_2$ eine reguläre Sprache ist. □

Satz 14 Sind L_1 und L_2 reguläre Sprachen, so ist auch der Durchschnitt $L_1 \cap L_2$ eine reguläre Sprache.

Beweis: Während der letzte Satz unmittelbar aus der Definition der regulären Ausdrücke gefolgert werden kann, müssen wir nun etwas weiter ausholen. Im letzten Kapitel haben wir gesehen, dass es zu jedem regulären Ausdruck r einen äquivalenten deterministischen endlichen Automaten A gibt, der die durch r spezifizierte Sprache akzeptiert und wir können außerdem annehmen, dass dieser Automat vollständig ist.

Es seien nun r_1 und r_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 spezifizieren:

$$L_1 = L(r_1) \quad \text{und} \quad L_2 = L(r_2).$$

Dann konstruieren wir zunächst zwei vollständige deterministische endliche Automaten A_1 und A_2, die diese Sprachen akzeptieren, es gilt also

$$L(A_1) = L_1 \quad \text{und} \quad L(A_2) = L_2.$$

Wir werden für die Sprache $L_1 \cap L_2$ einen Automaten A bauen, der diese Sprache akzeptiert. Da es zu jedem Automaten auch einen regulären Ausdruck gibt, der die Sprache beschreibt, die von dem Automaten akzeptiert wird, haben wir damit dann gezeigt, dass die Sprache $L_1 \cap L_2$ regulär ist. Als Baumaterial für den Automaten A, der die Sprache $L_1 \cup L_2$ akzeptiert, verwenden wir natürlich die Automaten A_1 und A_2. Wir nehmen an, dass

$$A_1 = \langle Q_1, \Sigma, \delta_1, q_1, F_1 \rangle \quad \text{und} \quad A_2 = \langle Q_2, \Sigma, \delta_2, q_2, F_2 \rangle$$

gilt und definieren A als eine Art kartesisches Produkt von A_1 und A_2:

$$A := \langle Q_1 \times Q_2, \Sigma, \delta, \langle q_1, q_2 \rangle, F_1 \times F_2 \rangle,$$

wobei die Zustands-Übergangs-Funktion

$$\delta : (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$$

durch die Gleichung

$$\delta\big(\langle p_1, p_2 \rangle, c\big) := \big\langle \delta_1(p_1, c), \delta_2(p_2, c) \big\rangle$$

definiert wird. Der so definierte endliche Automat A simuliert gleichzeitig die beiden Automaten A_1 und A_2 indem er parallel berechnet, in welchem Zustand jeweils A_1 und A_2 ist. Damit das möglich ist, bestehen die Zustände von A aus Paaren $\langle p_1, p_2 \rangle$, so dass p_1 ein Zustand von A_1 und p_2 ein Zustand von A_2 ist und die Funktion δ berechnet den Nachfolgezustand zu $\langle p_1, p_2 \rangle$, indem separat die Nachfolgezustände von p_1 und p_2 berechnet werden. Ein String wird genau dann akzeptiert, wenn sowohl A_1 als auch A_2 einen akzeptierenden Zustand erreicht haben. Daher wird die Menge der akzeptierenden Zustände wie folgt definiert:

$$F := \big\{ \langle p_1, p_2 \rangle \in Q_1 \times Q_2 \mid p_1 \in F_1 \wedge p_2 \in F_2 \big\} = F_1 \times F_2.$$

Damit gilt für alle $s \in \Sigma^*$:

$$\begin{array}{ll} & s \in L(A) \\ \text{g.d.w.} & \delta(\langle q_1, q_2 \rangle, s) \in F \\ \text{g.d.w.} & \langle \delta(q_1, s), \delta(q_2, s) \rangle \in F_1 \times F_2 \\ \text{g.d.w.} & \delta(q_1, s) \in F_1 \wedge \delta(q_2, s) \in F_2 \\ \text{g.d.w.} & s \in L(A_1) \wedge s \in L(A_2) \\ \text{g.d.w.} & s \in L(A_1) \cap L(A_2) \end{array}$$

Damit haben wir insgesamt gezeigt, dass

$$L(A) = L_1 \cap L_2$$

gilt und das war zu zeigen. □

Bemerkung: Prinzipiell wäre es möglich, für reguläre Ausdrücke eine Funktion

$$\wedge : \mathit{RegExp} \times \mathit{RegExp} \to \mathit{RegExp}$$

zu definieren, so dass für den Ausdruck $r_1 \wedge r_2$ die Beziehung

$$L(r_1 \wedge r_2) = L(r_1) \cap L(r_2)$$

gilt: Zunächst berechnen wir zu r_1 und r_2 äquivalente nicht-deterministische endliche Automaten, überführen diese Automaten dann in einen vollständigen deterministischen Automaten, bilden wie oben gezeigt das kartesische Produkt dieser Automaten und gewinnen schließlich aus diesem Automaten einen regulären Ausdruck zurück. Der so gewonnene reguläre Ausdruck wäre allerdings so groß, dass diese Funktion in der Praxis nicht implementiert wird, denn bei der Überführung eines nicht-deterministischen in einen deterministischen Automaten kann der Automat stark anwachsen und der reguläre Ausdruck, der sich aus einem Automaten ergibt, kann schon bei verhältnismäßig kleinen Automaten sehr unübersichtlich werden.

Satz 15 Ist L eine reguläre Sprache über dem Alphabet Σ, so ist auch das Komplement von L, die Sprache $\Sigma^* \backslash L$ eine reguläre Sprache.

Beweis: Wir gehen ähnlich vor wie beim Beweis des letzten Satzes und nehmen an, dass ein vollständiger deterministischer endlicher Automat A gegeben ist, der die Sprache L akzeptiert:

$$L = L(A).$$

Wir konstruieren einen Automaten $\hat{A}$, der ein Wort w genau dann akzeptiert, wenn A dieses Wort nicht akzeptiert. Dazu ist es lediglich erforderlich das Komplement der Menge der akzeptierenden Zustände von A zu bilden. Sei also

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle.$$

Dann definieren wir

$$\hat{A} = \langle Q, \Sigma, \delta, q_0, Q \backslash F \rangle.$$

Offenbar gilt

$$\begin{array}{ll} & w \in L(\hat{A}) \\ \Leftrightarrow & \delta(q_0, w) \in Q \backslash F \\ \Leftrightarrow & \delta(q_0, w) \notin F \\ \Leftrightarrow & w \notin L(A) \end{array}$$

und daraus folgt die Behauptung. □

Korollar 16 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Mengen-Differenz $L_1 \backslash L_2$ eine reguläre Sprache.

Beweis: Es sei Σ das Alphabet, das den Sprachen L_1 und L_2 zu Grunde liegt. Dann gilt

$$L_1 \backslash L_2 = L_1 \cap (\Sigma^* \backslash L_2),$$

denn ein Wort w ist genau dann in $L_1 \backslash L_2$, wenn w einerseits in L_1 und andererseits im Komplement von L_2 liegt. Nach dem letzten Satz wissen wir, dass mit L_2 auch das Komplement $\Sigma^* \backslash L_2$ regulär ist. Da der Durchschnitt zweier regulärer Sprachen wieder regulär ist, ist damit auch $L_1 \backslash L_2$ regulär. □

Insgesamt haben wir jetzt gezeigt, dass reguläre Sprachen unter den Boole'schen Mengen-Operationen abgeschlossen sind.

5.2 Erkennung leerer Sprachen

In diesem Abschnitt untersuchen wir für einen gegeben deterministischen endlichen Automaten

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle$$

die Frage, ob die von A erkannte Sprache leer ist, ob also $L(A) = \{\}$ gilt. Dazu fassen wir den endlichen Automaten als einen Graphen auf: Die Knoten dieses Graphen sind die Zustände von

A und zwischen zwei Zuständen q_1 und q_2 gibt es genau dann eine Kante, die q_1 mit q_2 verbindet, wenn es einen Buchstaben $c \in \Sigma$ gibt, so dass $\delta(q_1, c) = q_2$ gilt. Die Sprache $L(A)$ ist genau dann leer, wenn es in diesem Graphen keinen Pfad gibt, der von dem Start-Zustand q_0 ausgeht und in einem akzeptierenden Zustand endet, wenn also die akzeptierenden Zustände von dem Start-Zustand aus nicht erreichbar sind.

Daher berechnen wir zur Beantwortung der Frage, ob $L(A)$ leer ist, die Menge R der von dem Start-Zustand q_0 erreichbaren Zustände. Diese Berechnung kann am einfachsten induktiv erfolgen:

1. $q_0 \in R$.
2. $p_1 \in R \wedge \delta(p_1, c) = p_2 \;\Rightarrow\; p_2 \in R$.

Die Sprache $L(A)$ ist genau dann leer, wenn keiner der akzeptierenden Zustände erreichbar ist, mit anderen Worten haben wir

$$L(A) = \{\} \;\Leftrightarrow\; R \cap F = \{\}.$$

Damit haben wir einen Algorithmus zur Beantwortung der Frage $L(A) = \{\}$: Wir bilden die Menge alle vom Start-Zustand q_0 erreichbaren Zustände und überprüfen dann, ob diese Menge einen akzeptierenden Zustand enthält.

5.3 Äquivalenz regulärer Ausdrücke

Bei der Diskussion der algebraischen Vereinfachung regulärer Ausdrücke im Kapitel 2 hatten wir zwei reguläre Ausdrücke r_1 und r_2 als äquivalent bezeichnet (geschrieben $r_1 \doteq r_2$), wenn die durch r_1 und r_2 spezifizierten Sprachen identisch sind:

$$r_1 \doteq r_2 \stackrel{\text{def}}{\Longleftrightarrow} L(r_1) = L(r_2).$$

Wir werden in diesem Abschnitt eine Verfahren vorstellen, mit dem wir für zwei reguläre Ausdrücke r_1 und r_2 entscheiden können, ob $r_1 \doteq r_2$ gilt.

Satz 17 Es seien r_1 und r_2 zwei reguläre Ausdrücke. Dann ist die Frage, ob $r_1 \doteq r_2$ gilt, ob also die von den beiden Ausdrücken spezifizierte Sprachen gleich sind und damit

$$L(r_1) = L(r_2)$$

gilt, entscheidbar.

Beweis: Wir geben einen Algorithmus an, der die Frage, ob $L(r_1) = L(r_2)$ gilt, beantwortet. Zunächst bemerken wir, dass die Sprachen $L(r_1)$ und $L(r_2)$ genau dann gleich sind, wenn die Mengen-Differenzen $L(r_2) \backslash L(r_1)$ und $L(r_1) \backslash L(r_2)$ beide verschwinden, denn es gilt:

$$\begin{array}{lcl} L(r_1) = L(r_2) & \Leftrightarrow & L(r_1) \subseteq L(r_2) \;\wedge\; L(r_2) \subseteq L(r_1) \\ & \Leftrightarrow & L(r_1) \backslash L(r_2) = \{\} \;\wedge\; L(r_2) \backslash L(r_1) = \{\} \end{array}$$

Seien nun A_1 und A_2 endliche Automaten mit

$$L(A_1) = L(r_1) \quad \text{und} \quad L(A_2) = L(r_2).$$

Im letzten Kapitel haben wir gesehen, wie wir solche Automaten konstruieren können. Nach dem Korollar 16 sind die Sprachen $L(r_1) \backslash L(r_2)$ und $L(r_2) \backslash L(r_1)$ regulär und wir haben gesehen, wie wir endliche Automaten A_{12} und A_{21} so konstruieren können, dass

$$L(r_1) \backslash L(r_2) = L(A_{12}) \quad \text{und} \quad L(r_2) \backslash L(r_1) = L(A_{21})$$

gilt. Damit gilt nun

$$r_1 \doteq r_2 \;\Leftrightarrow\; L(A_{12}) = \{\} \wedge L(A_{21}) = \{\}$$

und diese Frage ist nach Abschnitt 5.2 entscheidbar. □

Bemerkung: Das oben angegebene Verfahren zur Überprüfung der Äquivalenz zweier regulärer Ausdrücke ist viel zu aufwendig, um von praktischem Nutzen zu sein. Es gibt einen effizienteren Algorithmus um die Äquivalenz zweier regulärer Ausdrücke r_1 und r_2 zu überprüfen:

1. Konstruiere zu r_1 und r_2 nicht-deterministische endliche Automaten A_1 und A_2, welche die durch r_1 und r_2 beschriebenen Sprachen erkennen:

   $$L(A_1) = L(r_1) \quad \text{und} \quad L(A_2) = L(r_2).$$

2. Überführe die nicht-deterministischen endlichen Automaten A_1 und A_2 in deterministische endliche Automaten D_1 und D_2.

3. Minimiere die Automaten D_1 und D_2. Dabei entstehen deterministische Automaten M_1 und M_2 mit den Eigenschaften

   $$L(M_1) = L(r_1) \quad \text{und} \quad L(M_2) = L(r_2).$$

Dann läßt sich zeigen, dass $r_1 \doteq r_2$ genau dann gilt, wenn die Automaten M_1 und M_2 bis auf die Namen der Zustände identisch sind. Der Nachweis der Korrektheit dieses Verfahrens benötigt das Myhill-Nerode-Theorem [Ner58] und geht über den Rahmen der Vorlesung hinaus. Ein Beweis der Korrektheit des oben vorgestellten Algorithmus findet sich in der zweiten Auflage des Lehrbuchs von Hopcroft and Ullman [HU79]. In der dritten Auflage [HMU06] findet sich dieser Beweis nicht mehr.

5.4 Grenzen regulärer Sprachen

Der folgende Satz liefert eine Eigenschaft regulärer Sprachen, mit deren Hilfe wir zeigen können, dass bestimmte Sprachen nicht regulär sein können.

Satz 18 (Pumping-Lemma) Es sei L eine reguläre Sprache. Dann gibt es eine natürliche Zahl $n \in \mathbb{N}$, so dass sich alle Strings $s \in L$, deren Länge größer oder gleich n ist, so in drei Teile u, v und w aufspalten lassen, dass die folgenden drei Bedingungen gelten:

1. $s = uvw$
2. $v \neq \varepsilon$,
3. $|uv| \leq n$,
4. $\forall h \in \mathbb{N} : uv^h w \in L$.

Das Pumping-Lemma für eine reguläre Sprache L kann in einer einzigen Formel zusammen gefaßt werden:

$$\exists n \in \mathbb{N} : \forall s \in L : |s| \geq n \rightarrow \exists u, v, w \in \Sigma^* : s = uvw \wedge v \neq \varepsilon \wedge |uv| \leq n \wedge (\forall h \in \mathbb{N} : uv^h w \in L).$$

Beweis: Da L eine reguläre Sprache ist, gibt es einen deterministischen endliche Automaten

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle,$$

so dass $L = L(A)$ ist. Die Zahl n, deren Existenz in dem Lemma behauptet wird, definieren wir als die Zahl der Zustände dieses Automaten:

$$n := \mathit{card}(Q).$$

Es sei nun ein Wort $s \in L$ gegeben, das aus mindestens n Buchstaben besteht. Konkret gelte

$$s = c_1 c_2 \cdots c_m \quad \text{mit } m \geq n,$$

wobei $c_1, \cdots, c_m$ die einzelnen Buchstaben sind. Wir betrachten die Berechnung, die der Automat

A bei der Eingabe s durchführt. Diese Berechnung hat die Form

$$q_0 \overset{c_1}{\mapsto} q_1 \overset{c_2}{\mapsto} q_2 \overset{c_3}{\mapsto} \cdots \overset{c_m}{\mapsto} q_m$$

und es gilt $q_m \in F$. Die Länge m des Wortes s hat nach Voraussetzung mindestens den Wert n. Daher können in der Liste

$$[q_0, q_1, q_2, \cdots, q_m]$$

nicht alle q_i verschieden sein, denn es gibt ja insgesamt nur n verschiedene Zustände. Wegen

$$card\big(\{0, 1, \cdots, n\}\big) = n + 1$$

können wir sogar sagen, dass in der Liste

$$[q_0, q_1, q_2, \cdots, q_n]$$

mindestens ein Zustand zweimal (oder öfter) auftreten muss. Bezeichnen wir den Index des ersten Auftretens mit k und den Index des zweiten Auftretens mit l, so haben wir also

$$q_k = q_l \wedge k < l \wedge l \leq n.$$

Dann können wir den String s wie folgt in die Strings u, v und w zerlegen:

$$u := c_1 \cdots c_k, \quad v := c_{k+1} \cdots c_l \quad \text{und} \quad w := c_{l+1} \cdots c_m$$

Aus $k < l$ folgt nun $v \neq \varepsilon$ und aus $l \leq n$ folgt $|uv| \leq n$. Weiter wissen wir das Folgende:

1. Beim Lesen von u geht der Automat vom Zustand q_0 in den Zustand q_k über, es gilt

   $$q_0 \overset{u}{\mapsto} q_k. \tag{5.1}$$

2. Beim Lesen von v geht der Automat vom Zustand q_k in den Zustand q_l über und da $q_l = q_k$ ist, gilt also

   $$q_k \overset{v}{\mapsto} q_k. \tag{5.2}$$

3. Beim Lesen von w geht der Automat vom Zustand $q_l = q_k$ in den akzeptierenden Zustand q_m über:

   $$q_k \overset{w}{\mapsto} q_m. \tag{5.3}$$

Aus $q_k \overset{v}{\mapsto} q_k$ folgt

$$q_k \overset{v}{\mapsto} q_k \overset{v}{\mapsto} q_k, \quad \text{also} \quad q_k \overset{v^2}{\mapsto} q_k$$

Da wir dieses Spiel beliebig of wiederholen können, haben wir für alle $h \in \mathbb{N}$

$$q_k \overset{v^h}{\mapsto} q_k \tag{5.4}$$

Aus den Gleichungen (5.1), (5.3) und 5.4) folgt nun

$$q_0 \overset{uv^hw}{\mapsto} q_m$$

und da q_m ein akzeptierender Zustand ist, haben wir damit $uv^hw \in L$ gezeigt. □

Das Pumping-Lemma kann benutzt werden um nachzuweisen, dass bestimmte Sprachen nicht regulär sind. Der nächste Satz gibt ein Beispiel.

Satz 19 Das Alphabet Σ sei durch $\Sigma = \{$ "(" , ")" $\}$ definiert, es enthält also die beiden Klammer-Symbole "(" und ")". Die Sprache L sei die Menge aller Strings, die aus k öffnenden runden Klammern gefolgt von k schließenden runden Klammern besteht:

$$L = \big\{ (^k)^k \mid k \in \mathbb{N} \big\}$$

Dann ist die Sprache L nicht regulär.

Beweis: Wir führen den Beweis indirekt und nehmen an, dass L regulär ist. Nach dem Pumping-Lemma gibt es dann eine Zahl n, so dass sich alle Strings $s \in L$, für die $|s| > n$ gilt, so in drei Teile u, v und w aufspalten lassen, dass

$$s = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{und} \quad \forall h \in \mathbb{N} : uv^h w \in L$$

gilt. Wir setzen

$$s := (^n)^n.$$

Offenbar gilt $|s| = 2 \cdot n \geq n$. Wir finden also jetzt drei Strings u, v und w, für die gilt:

$$(^n)^n = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{und} \quad \forall h \in \mathbb{N} : uv^h w \in L.$$

Wegen $|uv| \leq n$ und $v \neq \varepsilon$ wissen wir, dass der String v aus einer positiven Zahl öffnender runder Klammern bestehen muss:

$$v = (^k \quad \text{für ein } k \in \mathbb{N} \text{ mit } k > 0.$$

Setzen wir in der Formel $\forall h \in \mathbb{N} : uv^h w \in L$ für h den Wert 0 ein, so sehen wir, dass

$$uw \in L \tag{5.5}$$

gilt. Um den Beweis besser formalisieren zu können, führen wir eine Funktion

$$\mathit{count} : \Sigma^* \times \Sigma \to \mathbb{N}$$

ein. Für einen String t und einen Buchstaben c soll $\mathit{count}(t, c)$ zählen, wie oft der Buchstabe c in dem String t vorkommt. Für die Sprache L gilt offenbar

$$t \in L \Rightarrow \mathit{count}(t, \text{“(”}) = \mathit{count}(t, \text{“)”}).$$

Einerseits haben wir nun

$$\begin{array}{lcl}
\mathit{count}(uw, \text{“(”}) & = & \mathit{count}(uvw, \text{“(”}) - \mathit{count}(v, \text{“(”}) \\
& = & \mathit{count}(s, \text{“(”}) - \mathit{count}(v, \text{“(”}) \\
& = & \mathit{count}((^n)^n, \text{“(”}) - \mathit{count}((^k, \text{“(”}) \\
& = & n - k \\
& < & n
\end{array}$$

Andererseits gilt

$$\begin{array}{lcl}
\mathit{count}(uw, \text{“)”}) & = & \mathit{count}(uvw, \text{“)”}) - \mathit{count}(v, \text{“)”}) \\
& = & \mathit{count}(s, \text{“)”}) - \mathit{count}(v, \text{“)”}) \\
& = & \mathit{count}((^n)^n, \text{“)”}) - \mathit{count}((^k, \text{“)”}) \\
& = & n
\end{array}$$

Insgesamt sehen wir

$$\mathit{count}(uw, \text{“(”}) < \mathit{count}(uw, \text{“)”})$$

und damit kann der String uw im Widerspruch zum Pumping-Lemma nicht in der Sprache L liegen, denn alle Wörter der Sprache L enthalten gleich viele öffnende wie geschlossene Klammern. Dieser Widerspruch zum Pumping-Lemma zeigt, dass die Sprache L nicht regulär sein kann. □

Bemerkung: Der letzte Satz zeigt uns, dass wir mit Hilfe von regulären Ausdrücken noch nicht einmal Klammern zählen können. Der Begriff der regulären Ausdrücke ist damit offensichtlich zu schwach um die Syntax gängiger Programmier-Sprache adäquat zu beschreiben. Im nächsten Kapitel werden wir das Konzept der kontextfreien Grammatik kennen lernen, das wesentlichen mächtiger als das Konzept der regulären Sprachen ist. Mit diesem Konzept wird es dann möglich sein, die meisten Programmier-Sprachen zu beschreiben.

Aufgabe 10: Die Sprache L_{square} beinhaltet alle Wörter der Form $\mathtt{a}^n$ für die n eine Quadrat-Zahl ist, es gilt also

$$L_{\text{square}} = \left\{ a^m \mid \exists k \in \mathbb{N} : m = k^2 \right\}$$

Zeigen Sie, dass die Sprache L_{square} keine reguläre Sprache ist.

Hinweis: Nutzen Sie aus, dass der Abstand zwischen den Quadrat-Zahlen beliebig groß wird.

Historisches Das Pumping-Lemma geht auf einen allgemeineren Satz zurück, der von Bar-Hillel, Perles und Shamir bewiesen wurde [BHPS61].

Kapitel 6

Kontextfreie Sprachen

Im letzten Kapitel haben wir gesehen, dass reguläre Sprachen nicht in der Lage sind, Klammern zu zählen. Damit sind sie offenbar nicht ausdrucksstark genug, um Programmier-Sprachen zu beschreiben, wir brauchen ein mächtigeres Konzept. In diesem Kapitel stellen wir daher die *kontextfreien* Sprachen vorher. Diese basieren auf dem Konzept der *kontextfreien Grammatik*, das wir gleich besprechen.

6.1 Kontextfreie Grammatiken

Kontextfreie Sprachen dienen zur Beschreibung von Programmier-Sprachen, insofern handelt es sich bei den kontextfreien Sprachen genau wie bei den regulären Sprachen auch um formale Sprachen. Allerdings wollen wir später beim Einlesen eines Programms nicht nur entscheiden, ob das Programm korrekt ist, sondern wir wollen darüber hinaus den Programm-Text *strukturieren*. Den Vorgang des *Strukturierens* bezeichnen wir auch als *parsen* und das Programm, das diese Strukturierung vornimmt, wird als *Parser* bezeichnet. Als Eingabe erhält ein Parser üblicherweise nicht den Text eines Programms, sondern statt dessen eine Folge sogenannter *Token*. Diese Token werden von einem Scanner erzeugt, der mit Hilfe regulärer Ausdrücke den Programmtext in einzelne Wörter aufspaltet, die wir in diesem Zusammenhang als *Token* bezeichnen. Beispielsweise spaltet der Scanner des `C`-Compilers ein `C`-Programm in die folgenden Token auf:

- Operator-Symbole wie "`+`", "`+=`", "`<`", "`<=`" etc.,
- Klammer-Symbole wie "`(`", "`[`", "`symbol93`", sowie die entsprechenden schließenden Klammern,
- vordefinierte Schlüsselwörter wie "`if`", "`while`", etc.,
- Variablen- und Funktions-Namen wie "`x`", "`y`", "`printf`", etc.,
- Namen für Typen wie "`int`", "`char`" oder auch benutzerdefinierte Typnamen,
- Literale zur Bezeichnung von Konstanten, wie "`1.23`", "`"hallo"`" oder "`'c'`"
- Kommentare,
- *White-Space-Zeichen*, (Leerzeichen, Tabulatoren, Zeilenumbrüche).

Der Parser bekommt dann vom Scanner eine Folge von Tokens und hat die Aufgabe, daraus einen sogenannten *Syntax-Baum* zu bauen. Dazu bedient sich der Parser einer *Grammatik*, die mit Hilfe von *Grammatik-Regeln* angibt, wie die Eingabe zu strukturieren ist. Betrachten wir als Beispiel das Parsen arithmetischer Ausdrücke. Die Menge *ArithExpr* der arithmetischen Ausdrücke können wir induktiv definieren. Um die Struktur arithmetischer Ausdrücke korrekt wiedergeben zu können,

definieren wir gleichzeitig die Mengen *Product* und *Factor*. Die Menge *Product* enthält arithmetische Ausdrücke, die Produkte und Quotienten darstellen und die Menge *Factor* enthält einzelne Faktoren. Die Definition dieser zusätzlichen Megen ist notwendig, um später die Präzedenzen der Operatoren korrekt wiedergeben zu können. Die Grundbausteine der arithmetischen Ausdrücke sind Variablen, Zahlen, die Operator-Symbole und die Klammer-Symbole "`(`" und "`)`".

1. Jede Zahlenkonstante ist ein Faktor:

   $$C \in \mathit{Number} \Rightarrow C \in \mathit{Factor}.$$

2. Jede Variable ist ein Faktor:

   $$V \in \mathit{Variable} \Rightarrow V \in \mathit{Factor}.$$

3. Ist A ein arithmetischer Ausdruck und schließen wir diesen Ausdruck in Klammern ein, so erhalten wir einen Ausdruck, den wir als Faktor benützen können:

   $A \in \mathit{ArithExpr} \Rightarrow$ "`(`" A "`)`" $\in \mathit{Factor}$. Ein Wort zur Notation: Während in der obigen Formel A eine Meta-Variablen ist, die für einen beliebigen arithmetischen Ausdruck steht, sind die Strings "`(`" und "`)`" wörtlich zu interpretieren und deshalb in Gänsefüßchen eingeschlossen. Die Gänsefüßchen sind natürlich nicht Teil des arithmetischen Ausdrucks sondern dienen lediglich der Notation.

4. F ein Faktor, so ist F gleichzeitig auch ein Produkt:

   $$F \in \mathit{Factor} \Rightarrow F \in \mathit{Product}.$$

5. Ist P ein Produkt und ist F ein Faktor, so sind die Strings P "`*`" F und P "`/`" F ebenfalls Produkte:

   $$P \in \mathit{Product} \wedge F \in \mathit{Factor} \Rightarrow P\text{"*"}F \in \mathit{Product} \ \wedge \ P\text{"/"}F \in \mathit{Product}.$$

6. Jedes Produkt ist gleichzeitig auch ein arithmetischer Ausdruck

   $$P \in \mathit{Product} \Rightarrow P \in \mathit{ArithExpr}.$$

7. Ist A ein arithmetischer Ausdruck und ist P ein Produkt, so sind auch die Strings A "`+`" P und A "`-`" P arithmetische Ausdrücke:

   $$A \in \mathit{ArithExpr} \wedge P \in \mathit{Product} \Rightarrow A\text{"+"}P \in \mathit{ArithExpr} \ \wedge \ A\text{"-"}P \in \mathit{ArithExpr}.$$

Die oben angegebenen Regeln definieren die Mengen *Factor*, *Product* und *ArithExpr* durch wechselseitige Rekursion. Diese Definition können wir in Form von sogenannten *Grammatik-Regeln* wesentlich kompakter schreiben:

$$\begin{array}{rcl}
\mathit{ArithExpr} & \rightarrow & \mathit{ArithExpr}\ \text{"+"}\ \mathit{Product} \\
\mathit{ArithExpr} & \rightarrow & \mathit{ArithExpr}\ \text{"-"}\ \mathit{Product} \\
\mathit{ArithExpr} & \rightarrow & \mathit{Product} \\
\mathit{Product} & \rightarrow & \mathit{Product}\ \text{"*"}\ \mathit{Factor} \\
\mathit{Product} & \rightarrow & \mathit{Product}\ \text{"/"}\ \mathit{Factor} \\
\mathit{Product} & \rightarrow & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \text{"("}\ \mathit{ArithExpr}\ \text{")"} \\
\mathit{Factor} & \rightarrow & \mathit{Variable} \\
\mathit{Factor} & \rightarrow & \mathit{Number}
\end{array}$$

Die Ausdrücke auf der linken Seite einer Grammatik-Regel bezeichnen wir als *syntaktische Variablen* oder auch als *Nicht-Terminale*. In der Literatur finden Sie hierfür auch den Begriff *syntaktische Kategorie*. In dem Beispiel sind *ArithExpr*, *Product* und *Factor* die Nicht-Terminale. Die restlichen Ausdrücke, in unserem Fall also *Number* und die Zeichen "`+`", "`-`", "`*`", "`/`", "`(`" und "`)`"

bezeichnen wir als *Terminale* oder auch *Token*. Dies sind also genau die Zeichen, die nicht auf der linken Seite einer Grammatik-Regel stehen. Bei den Nicht-Terminalen gibt es zwei Arten:

1. Operator-Symbole und Trennzeichen wie beispielsweise "/" und "(". Diese Nicht-Terminalen stehen für sich selbst.

2. Token wie *Number* ist zusätzlich ein Wert zugeordnet.

Üblicherweise werden Grammatik-Regeln in einer kompakteren Notation angegeben, indem alle Regeln für ein Nicht-Terminal wie folgt zusammengefaßt werden:

$$\begin{aligned} \textit{ArithExpr} &\rightarrow \textit{ArithExpr}\ \text{“+”}\ \textit{Product} \mid \textit{ArithExpr}\ \text{“-”}\ \textit{Product} \mid \textit{Product} \\ \textit{Product} &\rightarrow \textit{Product}\ \text{“*”}\ \textit{Factor} \mid \textit{Product}\ \text{“/”}\ \textit{Factor} \mid \textit{Factor} \\ \textit{Factor} &\rightarrow \text{“(”}\ \textit{ArithExpr}\ \text{“)”} \mid \textit{Number} \end{aligned}$$

Hier werden also die einzelnen Alternativen einer Regel durch das Metazeichen | getrennt. Nach den obigen Beispielen geben wir jetzt die formale Definition.

Definition 20 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* G ist ein 4-Tupel

$$G = \langle V, T, R, S \rangle,$$

so dass folgendes gilt:

1. V ist eine Menge von Namen, die wir als *syntaktischen Variablen* oder auch *Nicht-Terminale* bezeichnen.

   In dem obigen Beispiel gilt

   $$V = \{\textit{ArithExpr}, \textit{Product}, \textit{Factor}\}.$$

2. T ist eine Menge von Namen, die wir als *Terminale* bezeichnen. Die Mengen T und V sind disjunkt, es gilt also

   $T \cap V = \emptyset$. In dem obigen Beispiel gilt

   $T = \{\textit{Number}, \textit{Variable},\ \text{“+”}\ ,\ \text{“-”}\ ,\ \text{“*”}\ ,\ \text{“/”}\ ,\ \text{“(”}\ ,\ \text{“)”}\ \}$

3. R ist die Menge der Regeln. Formal ist eine Regel ein Paar $\langle A, \alpha \rangle$:

   (a) Die erste Komponente dieses Paares ist eine syntaktische Variable:

   $$A \in V.$$

   (b) Die zweite Komponente ist ein String, der aus syntaktischen Variablen und Terminalen aufgebaut ist:

   $$\alpha \in (V \cup T)^*.$$

   Insgesamt gilt für die Menge der Regeln R damit

   $$R \subseteq V \times (V \cup T)^*$$

   Ist $\langle X, \alpha \rangle$ eine Regel, so schreiben wir diese Regel als

   $X \rightarrow \alpha$. Beispielsweise haben wir oben die erste Regel als

   ArithExpr → *ArithExpr* “+” *Product*

   geschrieben. Formal steht diese Regel für das Paar

   $$\big\langle \textit{ArithExpr}, [\textit{ArithExpr},\ \text{“+”}\ , \textit{Product}] \big\rangle.$$

4. S ist ein Element der Menge V, das wir als das *Start-Symbol* bezeichnen. In dem obigen Beispiel ist *ArithExpr* das Start-Symbol. □

6.1.1 Ableitungen

Als nächstes wollen wir festlegen, welche Sprache durch eine gegebene Grammatik G definiert wird. Dazu definieren wir zunächst den Begriff eines *Ableitungs-Schrittes*. Es sei

1. $G = \langle V, T, R, S \rangle$ eine Grammatik,
2. $A \in V$ eine syntaktische Variable,
3. $\alpha A \beta \in (V \cup T)^*$ ein String aus Terminalen und syntaktischen Variablen, der die Variable A enthält,
4. $(A \to \gamma) \in R$ eine Regel.

Dann kann der String $\alpha A \beta$ durch einen Ableitungs-Schritt in den String $\alpha \gamma \beta$ überführt werden, wir ersetzen also ein Auftreten der syntaktische Variable A durch die rechte Seite der Regel $A \to \gamma$. Diesen Ableitungs-Schritt schreiben wir als

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta.$$

Geht die verwendete Grammatik G aus dem Zusammenhang klar hervor, so wird der Index $_G$ weggelassen und wir schreiben kürzer $\Rightarrow$ an Stelle von $\Rightarrow_G$. Der transitive und reflexive Abschluss der Relation $\Rightarrow_G$ wird mit $\Rightarrow_G^*$ bezeichnet. Wollen wir ausdrücken, dass die Ableitung des Strings w aus dem Nicht-Terminal A aus n Ableitungs-Schritten besteht, so schreiben wir

$$A \Rightarrow^n w.$$

Wir geben ein Beispiel:

$$\begin{array}{rcl}
\mathit{ArithExpr} & \Rightarrow & \mathit{ArithExpr}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Product}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Product}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Factor}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Factor} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}
\end{array}$$

Damit haben wir also gezeigt, dass

$$\mathit{ArithExpr} \Rightarrow^* \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}$$

oder genauer

$$\mathit{ArithExpr} \Rightarrow^8 \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}$$

gilt. Ersetzen wir hier das Terminal *Number* durch verschiedene Zahlen, so haben wir damit beispielsweise gezeigt, dass der String

$$2 * 3 + 4$$

ein arithmetischer Ausdruck ist. Allgemein definieren wir die durch eine Grammatik G definierte Sprache $L(G)$ als die Menge aller Strings, die einerseits nur aus Terminalen bestehen und die sich andererseits aus dem Start-Symbol S der Grammatik ableiten lassen:

$$L(G) := \big\{ w \in T^* \mid S \Rightarrow^* w \big\}.$$

Beispiel: Die Sprache

$$L = \{(^n)^n \mid n \in \mathbb{N}\}$$

wird von der Grammatik

$$G = \big\langle \{S\}, \{\text{ “(” , “)” }\}, R, S \big\rangle$$

erzeugt, wobei die Regeln R wie folgt gegeben sind:

$$\begin{array}{lcl} S & \rightarrow & \text{“(” } S \text{ “)”} \\ & | & \varepsilon \end{array}$$

Beweis: Wir zeigen zunächst, dass sich jedes Wort $w \in L$ aus dem Start-Symbol S ableiten läßt:

$$w \in L \;\rightarrow\; S \Rightarrow^* w.$$

Der Beweis erfolgt durch Induktion über die Länge n des Wortes w.

I.A. $n = 0$: Dann gilt $w = \varepsilon$. Offenbar gilt

$$S \Rightarrow \varepsilon,$$

denn die Grammatik enthält die Regel $S \rightarrow \varepsilon$.

I.S. : Dann hat w die Form $w =$ “(” v “)” , der String v ist kürzer als w und liegt in L. Also gibt es nach I.V. eine Ableitung von v:

$$S \Rightarrow^* v.$$

Dann bekommen wir insgesamt die folgende Ableitung

$$S \Rightarrow \text{ “(” } S \text{ “)” } \Rightarrow^* \text{ “(” } v \text{ “)” } = w.$$

Als nächstes zeigen wir, dass jedes Wort w, dass sich aus S ableiten läßt, ein Element der Sprache L ist. Wir führen den Beweis durch Induktion über die Anzahl n der Ableitungs-Schritte:

I.A. $n = 1$: Die einzige Ableitung eines aus Terminalen aufgebauten Strings, die nur aus einem Schritt besteht, ist

$$S \Rightarrow \varepsilon.$$

Dann muss $w = \varepsilon$ gelten und wegen $\varepsilon = (^0)^0 \in L$ haben wir $w \in L$.

I.S. $n \mapsto n+1$: Wenn die Ableitung aus mehr als einem Schritt besteht, dann muss die Ableitung die folgende Form haben:

$$S \Rightarrow \text{ “(” } S \text{ “)” } \Rightarrow^n w$$

Daraus folgt

$$w = \text{ “(” } v \text{ “)” } \;\wedge\; S \Rightarrow^n v.$$

Nach I.V. gilt dann $v \in L$. Damit gibt es $k \in \mathbb{N}$ mit $v = (^k)^k$. Also haben wir

$$w = \text{ “(” } v \text{ “)” } = ((^k)^k) = (^{k+1})^{k+1} \in L.$$ □

Aufgabe 11: Wir definieren für $w \in \Sigma^*$ und $c \in \Sigma$ die Funktion

$$\mathit{count}(w, c),$$

die zählt, wie oft der Buchstabe c in dem Wort w vorkommt, durch Induktion über w.

I.A. $w = \varepsilon$. Dann setzen wir

$$\mathit{count}(\varepsilon, c) := 0.$$

I.S. $w = d\,v$ mit $d \in \Sigma$. Dann wird $\mathit{count}(d\,v, c)$ durch eine Fall-Unterscheidung definiert:

$$\mathit{count}(d\,v) := \begin{cases} \mathit{count}(v, c) + 1 & \text{falls } c = d; \\ \mathit{count}(v, c) & \text{falls } c \neq d. \end{cases}$$

Wir setzen $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ und definieren die Sprache L als die Menge der Wörter $w \in \Sigma^*$, in denen die Buchstaben `a` und `b` mit der selben Häufigkeit vorkommen:

$$L := \big\{w \in \Sigma^* \mid \mathit{count}(w, \mathtt{a}) = \mathit{count}(w, \mathtt{b})\big\}$$

Geben Sie eine Grammatik G an, so dass $L = L(G)$ gilt und beweisen Sie Ihre Behauptung!

Aufgabe 12: Wieder sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Wir definieren die Menge L als die Menge der Strings s, die sich nicht in der Forms $s = ww$ schreiben lassen:

$$L = \big\{s \in \Sigma^* \mid \neg(\exists w \in \Sigma^* : s = ww)\big\}.$$

Geben Sie eine kontextfreie Grammatik G an, die diese Sprache erzeugt.

6.1.2 Parse-Bäume

Mit Hilfe einer Grammatik G können wir nicht nur erkennen, ob ein gegebener String s ein Element der von der Grammatik erzeugten Sprache $L(G)$ ist, wir können den String auch strukturieren indem wir einen *Parse-Baum* aufbauen. Ist eine Grammatik

$$G = \langle V, T, R, S\rangle$$

gegeben, so ist ein Parse-Baum für diese Grammatik ein Baum, der den folgenden Bedingungen genügt:

1. Jeder innere Knoten (also jeder Knoten, der kein Blatt ist), ist mit einem Nicht-Terminal beschriftet.
2. Jedes Blatt ist mit einem Terminal oder mit ε beschriftet.
3. Ist ein innerer Knoten mit einer Variablen A beschriftet und sind die Kinder dieses Knotens mit den Symbolen X_1, X_2, $\cdots$ X_n beschriftet, so enthält die Grammatik G eine Regel der Form

   $$A \to X_1 X_2 \cdots X_n.$$

Die Blätter des Parse-Baums ergeben dann, wenn wir sie von links nach rechts lesen, ein Wort, das von der Grammatik G abgeleitet wird. Abbildung 6.1 zeigt einen Parse-Baum für das Wort "`2*3+4`", der mit der oben angegebenen Grammatik für arithmetische Ausdrücke abgeleitet worden ist. Da Bäume der in Abbildung 6.1 gezeigten Art sehr schnell zu groß werden, vereinfachen wir diese Bäume mit Hilfe der folgenden Regeln:

1. Ist n ein innerer Knoten, der mit der Variablen A beschriftet ist und gibt es unter den Kindern dieses Knotens genau ein Kind, dass mit einem Terminal o beschriftet ist, so entfernen wir dieses Kind und beschriften den Knoten statt dessen mit dem Terminal o.
2. Hat ein innerer Knoten nur ein Kind, so ersetzen wir diesen Knoten durch sein Kind.

Den Baum, den wir auf diese Weise erhalten, nennen wir den *abstrakten Syntax-Baum*. Abbildung 6.2 zeigt den abstrakten Syntax-Baum den wir erhalten, wenn wir den in Abbildung 6.1 gezeigten Parse-Baum nach diesen Regeln vereinfachen. Die in diesem Baum gespeicherte Struktur ist genau das, was wir brauchen um den arithmetischen Ausdruck "`2*3+4`" auszuwerten.

6.1.3 Mehrdeutige Grammatiken

Die zu Anfang des Abschnitts 6.1 angegebene Grammatik erscheint durch ihre Unterscheidung der syntaktischen Kategorien *ArithExpr*, *Product* und *Factor* unnötig kompliziert. Wir stellen eine einfachere Grammatik G vor, welche dieselbe Sprache beschreibt:

$$G = \big\langle \{\mathit{Expr}\}, \{\mathit{Number},\ \text{“}\mathtt{+}\text{”}\ ,\ \text{“}\mathtt{-}\text{”}\ ,\ \text{“}\mathtt{*}\text{”}\ ,\ \text{“}\mathtt{/}\text{”}\ ,\ \text{“}\mathtt{(}\text{”}\ ,\ \text{“}\mathtt{)}\text{”}\ \}, R, \mathit{Expr}\big\rangle,$$

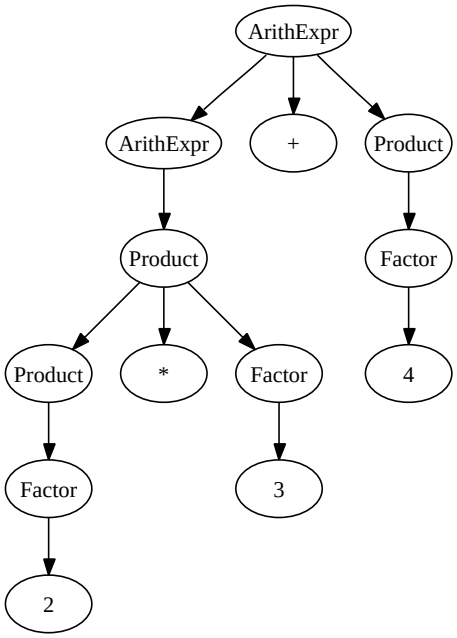


Abbildung 6.1: Ein Parse-Baum für den String "`2*3+4`".

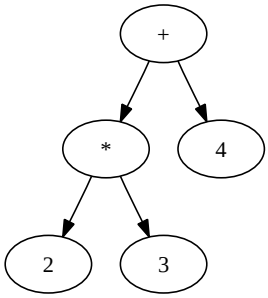


Abbildung 6.2: Ein abstrakter Syntax-Baum für den String "`2*3+4`".

wobei die Regeln R wie folgt gegeben sind:

$$
\begin{array}{rcl}
\textit{Expr} & \rightarrow & \textit{Expr} \text{ “+” } \textit{Expr} \\
& | & \textit{Expr} \text{ “-” } \textit{Expr} \\
& | & \textit{Expr} \text{ “*” } \textit{Expr} \\
& | & \textit{Expr} \text{ “/” } \textit{Expr} \\
& | & \text{“(” } \textit{Expr} \text{ “)”} \\
& | & \textit{Number}
\end{array}
$$

Um zu zeigen, dass der String "`2*3+4`" in der von dieser Sprache erzeugten Grammatik liegt, geben wir die folgende Ableitung an:

$$
\begin{array}{rcl}
\textit{Expr} & \Rightarrow & \textit{Expr} \text{ “+” } \textit{Expr} \\
& \Rightarrow & \textit{Expr} \text{ “*” } \textit{Expr} \text{ “+” } \textit{Expr} \\
& \Rightarrow & \texttt{2} \text{ “*” } \textit{Expr} \text{ “+” } \textit{Expr} \\
& \Rightarrow & \texttt{2} \text{ “*” } \texttt{3} \text{ “+” } \textit{Expr} \\
& \Rightarrow & \texttt{2} \text{ “*” } \texttt{3} \text{ “+” } \texttt{4}
\end{array}
$$

Diese Ableitung entspricht dem abstrakten Syntax-Baum, der in Abbildung 6.2 gezeigt ist. Es gibt aber noch eine andere Ableitung des Strings "`2*3+4`" mit dieser Grammatik:

$$
\begin{aligned}
\mathit{Expr} &\Rightarrow \mathit{Expr}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr} \\
&\Rightarrow \mathit{Expr}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
&\Rightarrow \texttt{2}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
&\Rightarrow \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
&\Rightarrow \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \texttt{4}
\end{aligned}
$$

Dieser Ableitung entspricht der abstrakte Syntax-Baum, der in Abbildung 6.3 gezeigt ist. Bei dieser Ableitung wird der String "`2*3+4`" offenbar als Produkt aufgefasst, was der Konvention widerspricht, dass der Operator "`*`" stärker bindet als der Operator "`+`". Würden wir den String an Hand des letzten Syntax-Baums auswerten, würden wir offenbar ein falsches Ergebnis bekommen! Die Ursache dieses Problems ist die Tatsache, dass die zuletzt angegebene Grammatik mehrdeutig

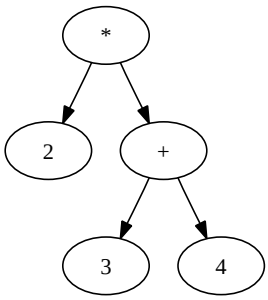


Abbildung 6.3: Ein anderer abstrakter Syntax-Baum für den String "`2*3+4`".

ist. Eine solche Grammatik ist zum Parsen ungeeignet. Leider ist die Frage, ob eine gegebene Grammatik mehrdeutig ist, im Allgemeinen nicht entscheidbar. Ein Beweis dieser Behaupung findet sich beispielsweise in dem Buch von Hopcroft, Motwani und Ullman [HMU06] auf Seite 415.

Beispiel: Es sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Die Sprache L enthalte alle die Wörter aus Σ^*, bei denen die Buchstaben `a` and `b` mit der gleichen Häufigkeit auftreten, es gilt also

$$L = \big\{w \in \Sigma^* \mid \mathit{count}(w, \mathtt{a}) = \mathit{count}(w, \mathtt{b})\big\}.$$

Dann wird die Sprache L durch die kontextfreie Grammatik $G_1 = \langle\{S\}, \Sigma, R_1, S\rangle$ beschrieben, deren Regeln wie folgt gegeben sind:

$$S \rightarrow \mathtt{a}S\mathtt{b}S \mid bS\mathtt{a}S \mid \varepsilon$$

Diese Grammatik ist allerdings mehrdeutig: Betrachten wir beispielsweise den String "`abab`", so stellen wir fest, dass sich dieser prinzipiell auf zwei Arten ableiten läßt:

$$
\begin{aligned}
S &\Rightarrow \mathtt{a}S\mathtt{b}S \\
&\Rightarrow \mathtt{ab}S \\
&\Rightarrow \mathtt{aba}S\mathtt{b}S \\
&\Rightarrow \mathtt{abab}S \\
&\Rightarrow \mathtt{abab}
\end{aligned}
$$

Eine andere Ableitung des selben Strings ergibt sich, wenn wir im zweiten Ableitungs-Schritt nicht das erste S durch ε ersetzen sondern statt dessen das zweite S durch ε ersetzen:

$$
\begin{aligned}
S &\Rightarrow \mathtt{a}S\mathtt{b}S \\
&\Rightarrow \mathtt{a}S\mathtt{b} \\
&\Rightarrow \mathtt{ab}S\mathtt{a}S\mathtt{b}
\end{aligned}
$$

$\Rightarrow$ $\texttt{aba}S\texttt{b}$

$\Rightarrow$ $\texttt{abab}$

Abbildung 6.4 zeigt die Parse-Bäume, die sich aus den beiden Ableitungen ergeben. Wir können erkennen, dass die Struktur dieser Bäume unterschiedlich ist: Im ersten Fall gehört das erste "a" zu dem ersten "b", im zweiten Fall gehört das erste "a" zu dem letzten "b".

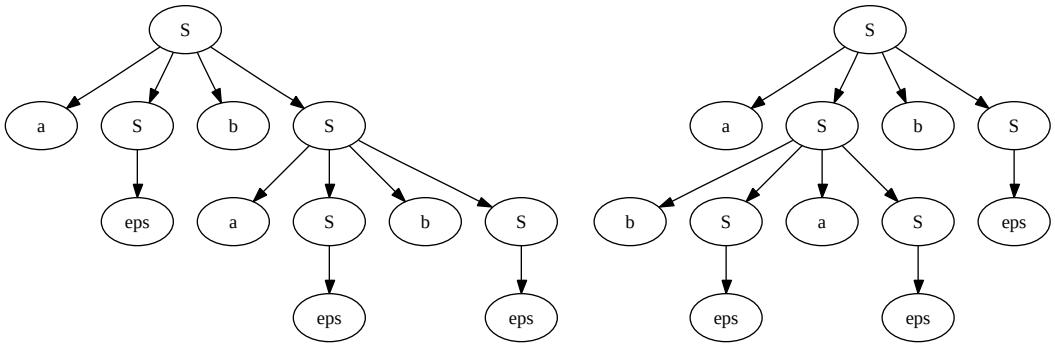


Abbildung 6.4: Zwei strukturell verschiedene Parse-Bäume für den String "abab".

Wir definieren nun eine kontextfreie Grammatik $G_2 = \langle \{S, U, V, X, Y\}, \Sigma, R_2, S \rangle$, deren Regeln wie folgt gegeben sind:

$$S \rightarrow US \mid VS \mid \varepsilon$$

$$U \rightarrow \texttt{a}X\texttt{b}$$

$$V \rightarrow \texttt{b}Y\texttt{a}$$

$$X \rightarrow UX \mid \varepsilon$$

$$Y \rightarrow VY \mid \varepsilon$$

Um die Sprachen, die von den einzelnen Variablen erzeugt werden, klarer beschreiben zu können, definieren wir für zwei Strings u und w die Relation $u \preceq w$ (lese: u ist ein Präfix von v) wie folgt:

$$u \preceq w \quad \stackrel{\text{def}}{\Longleftrightarrow} \exists v \in \Sigma^* : uv = w$$

Sodann bemerken wir, dass von den syntaktischen Variablen X und Y die folgenden Sprachen erzeugt werden:

$$L(X) = \big\{ w \in \Sigma^* \mid w \in L \ \wedge \ \forall u \preceq w : \mathit{count}(u, \texttt{b}) \leq \mathit{count}(u, \texttt{a}) \big\} \quad \text{und}$$

$$L(Y) = \big\{ w \in \Sigma^* \mid w \in L \ \wedge \ \forall u \preceq w : \mathit{count}(u, \texttt{a}) \leq \mathit{count}(u, \texttt{b}) \big\}.$$

Ein String der Sprache $L(X)$ kann also kein "b" enthalten, dass zu einem "a" gehört, das vor diesem String steht und analog kann ein String der Sprache $L(Y)$ kein "a" enthalten, dass zu einem "b" gehört, das vor diesem String steht.

Ein String der Sprache L fängt nun entweder mit "a" oder mit "b" an. Im ersten Fall interpretieren wir das "a" als öffnende Klammer und das "b" als schließende Klammer und suchen nun das "b", das dem "a" am Anfang des Strings zugeordnet ist. Der String, der mit dem "a" anfängt und dem "b" endet, liegt in der Sprache $L(U)$. Der zweite Fall ist analog. □

In dem obigen Beispiel hatten wir Glück und konnten eine Grammatik finden, mit der sich die Sprache eindeutig parsen läßt. Es gibt allerdings auch kontextfreie Sprachen, die inhärent mehrdeutig sind: Es läßt sich beispielsweise zeigen, dass für das Alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ die Sprache

$$L = \left\{ \mathtt{a}^m \mathtt{b}^m \mathtt{c}^n \mathtt{d}^n \mid m, n \in \mathbb{N} \right\} \cup \left\{ \mathtt{a}^m \mathtt{b}^n \mathtt{c}^n \mathtt{d}^m \mid m, n \in \mathbb{N} \right\}$$

kontextfrei ist, aber jede Grammatik G mit der Eigenschaft $L = L(G)$ ist notwendigerweise mehrdeutig. Das Problem ist, dass für große n Strings der Form

$$\mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \mathtt{d}^n$$

immer mehrere strukturell verschiedene Parse-Bäume besitzen. Ein Beweis dieser Behaupung findet sich in dem Buch von Hopcroft und Ullman auf Seite 100 [HU79].

6.2 Top-Down-Parser

In diesem Abschnitt stellen wir ein Verfahren vor, mit dem sich viele Grammatiken leicht parsen lassen. Die Grundidee ist einfach: Um einen String w mit Hilfe einer Grammatik-Regel der Form

$$A \to X_1 X_2 \cdots X_n$$

zu parsen, versuchen wir, zunächst ein X_1 zu parsen. Dabei zerlegen wir den String in $w = w_1 r_1$ so, dass $w_1 \in L(X_1)$ liegt. Dann versuchen wir, in dem Rest-String r_1 ein X_2 zu parsen und zerlegen dabei r_1 so, dass $r_1 = w_2 r_2$ mit $w_2 \in L(X_2)$ gilt. Zum Schluss haben wir dann den String w aufgespaltet in

$$w = w_1 w_2 \cdots w_n \quad \text{mit } w_i \in L(X_i) \text{ für alle } i = 1, \cdots, n.$$

Leider funktioniert dieses Verfahren dann nicht, wenn die Grammatik *links-rekursiv* ist, dass heißt, dass eine Regel die Form

$$A \to A\beta$$

hat, denn dann würden wir um ein A zu parsen sofort wieder rekursiv versuchen, ein A zu parsen und wären damit in einer Endlos-Schleife. Es gibt mehrere Möglichkeiten, um mit dem Problem umzugehen:

1. Wir können die Grammatik so umschreiben, dass sie danach nicht mehr links-rekursiv ist.
2. Wir können versuchen, den String *rückwärts* zu parsen, d.h. bei einer Regel der Form

   $$A \to X_1 X_2 \cdots X_n$$

   versuchen wir als erstes, ein X_n am Ende eines zu parsenden Strings w zu entdecken und arbeiten den String w dann von hinten ab.
3. Wir können versuchen, den Parser mit Hilfe von *Backtracking* zu implementieren: Bei diesem Verfahren werden nacheinander alle Regeln ausprobiert, die beim Parsen in Frage kommen. Solche Parser lassen sich sehr einfach mit Hilfe der Sprache *Prolog* implementieren, denn bei der Programmier-Sprache *Prolog* ist *Backtracking* Bestandteil des Sprache.

Wir werden die oben genannten Verfahren an Hand der Grammatik für arithmetische Ausdrücke diskutieren.

6.2.1 Umschreiben der Grammatik

Ist A ein Nicht-Terminal, das durch die beiden Regeln

$$\begin{array}{lcl} A & \to & A\beta \\ & | & \gamma \end{array}$$

beschrieben wird, so hat eine Ableitung von A, bei der zunächst immer die Variable A ersetzt wird, die Form

$$A \Rightarrow A\beta \Rightarrow A\beta\beta \Rightarrow A\beta\beta\beta \Rightarrow \cdots \Rightarrow A\beta^n \Rightarrow \gamma\beta^n.$$

Damit sehen wir, dass die durch das Nicht-Terminal A beschriebene Sprache $L(A)$ aus alle den

Strings besteht, die sich aus dem Ausdruck $\gamma\beta^n$ ableiten lassen:

$$L(A) = \big\{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\big\}.$$

Diese Sprache kann offenbar auch durch die folgenden Regeln für A beschrieben werden:

$$\begin{array}{lcl} A & \to & \gamma B \\ B & \to & \beta B \\ & \mid & \varepsilon \end{array}$$

Hier haben wir die Hilfs-Variable B eingeführt. Die Ableitungen, die von dem Nicht-Terminal B ausgehen, haben die Form

$$B \Rightarrow \beta B \Rightarrow \beta\beta B \Rightarrow \cdots \Rightarrow \beta^n B \Rightarrow \beta^n.$$

Folglich beschreibt das Nicht-Terminal B die Sprache

$$L(B) = \big\{w \in \Sigma \mid \exists n \in \mathbb{N} : \beta^n \Rightarrow w\big\}.$$

Damit ist klar, dass auch mit der oben angegeben Grammatik

$$L(A) = \big\{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\big\}$$

gilt. Um die Links-Rekursion aus der in Abbildung 6.5 auf Seite 72 gezeigten Grammatik zu entfernen müssen wir das obige Beispiel verallgemeinern. Wir betrachten jetzt den allgemeinen Fall und nehmen an, dass ein Nicht-Terminal A durch Regeln der Form

$$\begin{array}{lcl} A & \to & A\beta_1 \\ & \mid & A\beta_2 \\ & \vdots & \vdots \\ & \mid & A\beta_k \\ & \mid & \gamma_1 \\ & \vdots & \vdots \\ & \mid & \gamma_l \end{array}$$

beschrieben wird. Wir können diesen Fall durch Einführung zweier Hilfs-Variablen B und C auf den ersten Fall zurückführen:

$$\begin{array}{lcl} A & \to & AB \mid C \\ B & \to & \beta_1 \mid \cdots \mid \beta_k \\ C & \to & \gamma_1 \mid \cdots \mid \gamma_l \end{array}$$

Dann können wir die Grammatik umschreiben, indem wir eine neue Hilfs-Variablen, nennen wir sie L für Liste, einführen und erhalten

$$\begin{array}{lcl} A & \to & C\ L \\ L & \to & B\ L \mid \varepsilon. \end{array}$$

Die Hilfs-Variablen B und C können nun wieder eliminiert werden und dann bekommen wir die folgende Grammatik:

$$\begin{array}{lcl} A & \to & \gamma_1\ L \mid \gamma_2\ L \mid \cdots \mid \gamma_l\ L \\ L & \to & \beta_1\ L \mid \beta_2\ L \mid \cdots \mid \beta_k\ L \mid \varepsilon \end{array}$$

Wenden wir dieses Verfahren auf die in Abbildung 6.5 gezeigte Grammatik für arithmetische Ausdrücke an, so erhalten wir die in Abbildung 6.6 gezeigte Grammatik. Die Variablen *ExprRest* und *ProductRest* können wie folgt interpretiert werden:

1. *ExprRest* beschreibt eine Liste der Form

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
& | & \mathit{Expr}\ \text{“}\texttt{-}\text{”}\ \mathit{Product} \\
& | & \mathit{Product} \\
\mathit{Product} & \rightarrow & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& | & \mathit{Product}\ \text{“}\texttt{/}\text{”}\ \mathit{Factor} \\
& | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& | & \mathit{Number}
\end{array}
$$

Abbildung 6.5: Links-rekursive Grammatik für arithmetische Ausdrücke.

op Product $\cdots$ *op Product*,

wobei $op \in \{$ “+” , “-” $\}$ gilt.

2. *ProductRest* beschreibt eine Liste der Form

   op Factor $\cdots$ *op Factor*,

   wobei $op \in \{$ “*” , “/” $\}$ gilt.

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Product}\ \ \mathit{ExprRest} \\
\mathit{ExprRest} & \rightarrow & \text{“}\texttt{+}\text{”}\ \mathit{Product}\ \ \mathit{ExprRest} \\
& | & \text{“}\texttt{-}\text{”}\ \mathit{Product}\ \ \mathit{ExprRest} \\
& | & \varepsilon \\
\mathit{Product} & \rightarrow & \mathit{Factor}\ \ \mathit{ProductRest} \\
\mathit{ProductRest} & \rightarrow & \text{“}\texttt{*}\text{”}\ \mathit{Factor}\ \ \mathit{ProductRest} \\
& | & \text{“}\texttt{/}\text{”}\ \mathit{Factor}\ \ \mathit{ProductRest} \\
& | & \varepsilon \\
\mathit{Factor} & \rightarrow & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& | & \textsc{Number}
\end{array}
$$

Abbildung 6.6: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

Aufgabe 13:

(a) Die folgende Grammatik beschreibt reguläre Ausdrücke:

$$
\begin{array}{rcl}
\mathit{RegExp} & \rightarrow & \mathit{RegExp}\ \text{“|”}\ \mathit{RegExp} \\
& | & \mathit{RegExp}\ \ \mathit{RegExp} \\
& | & \mathit{RegExp}\ \text{“*”} \\
& | & \text{“(”}\ \mathit{RegExp}\ \text{“)”} \\
& | & \textsc{Letter}
\end{array}
$$

Diese Grammatik verwendet nur die syntaktische Variable $\{\mathit{RegExp}\}$ und die folgenden Terminale

“|”, “*”, “(”, “)”, Letter.

Da die Grammatik mehrdeutig ist, ist diese Grammatik zum Parsen ungeeignet. Transformieren Sie diese Grammatik in eine eindeutige Grammatik. Orientieren Sie sich an der Grammatik für arithmetische Ausdrücke und führen Sie geeignete neue syntaktische Variablen ein.

(b) Entfernen Sie die Links-Rekursion aus der in Teil (a) dieser Aufgabe erstellten Grammatik.

6.2.2 Implementierung eines Top-Down-Parser

Wir behandeln jetzt die Implementierung eines Top-Down-Parsers in *Java*. Bevor wir mit der Implementierung des eigentlichen Parsers beginnen können, benötigen wir einen *Scanner*, manchmal wird der auch als *Tokenizer* bezeichnet, der die Aufgabe hat, einen String aus Ascii-Zeichen zu lesen und in eine Liste aus *Terminalen* (manchmal sagen wir auch *Tokens*) zu verwandeln. In unserem Fal sind die Terminale die Operator-Symbole, die beiden Klammern und die Zahlen. Wir werden diesen Scanner mit Hilfe von *JFlex* entwickeln. Abbildung 6.7 zeigt die Implementierung dieses Scanners.

1. In Zeile 1 importieren wir alle Klassen aus dem Paket `Java.util`, denn wir benötigen später die Klasse `ArrayList` und das Interface `List`.

2. Die erzeugte Klasse bekommt in Zeile 5 den Namen Scannner.

3. In Zeile 10 legen wir die Member-Variable `mTokenList` an. In dieser Member-Variable sammeln wir die Liste aller Token auf, die wir lesen.

4. Code, der in den Strings “`%eof{`” und “`%eof}`” eingeschlossen ist, wird dann ausgeführt, wenn *JFlex* das Ende der zu lesenden Datei erreicht hat. Dann enthält die Variable `mTokenList` alle gelesenen Token. Wir legen daher in Zeile 15 ein Objekt der Klasse `ExprParser` an, dem wir die Liste der erkannten Tokens übergeben.

5. Anschließend rufen wir von diesem Objekt die Methode *parseExpr*() auf. Aufgabe dieser Methode ist das Parsen und Auswerten der arithmetischen Ausdrucks. Da dabei eventuell eine Ausnahme vom Typ *ParseError* auftreten kann, wird das Parsen in einen `try-catch`-Block eingefaßt.

6. Das eigentliche Scannen findet nun in den Zeilen 24 – 31 statt:

   (a) In Zeile 24 erkennen wir natürliche Zahlen,

```
1  import java.util.*;
2
3  %%
4
5  %class Scanner
6  %standalone
7
8  %unicode
9
10 %{
11     List<Token> mTokenList = new ArrayList<Token>();
12 %}
13
14 %eof{
15     ExprParser parser = new ExprParser(mTokenList);
16     try {
17         Integer    result = parser.parseExpr();
18         System.out.println("Ergebnis: " + result);
19     } catch (ParseError e) {}
20 %eof}
21
22 %%
23
24 [1-9][0-9]*  { mTokenList.add(new NumberToken(yytext())); }
25 "+"          { mTokenList.add(new Token("+"));           }
26 "-"          { mTokenList.add(new Token("-"));           }
27 "*"          { mTokenList.add(new Token("*"));           }
28 "/"          { mTokenList.add(new Token("/"));           }
29 "("          { mTokenList.add(new Token("("));           }
30 ")"          { mTokenList.add(new Token(")"));           }
31 [ \t\n\r]    { /* skip */                                }
32 .            { System.out.println("Unexpected char: \"" +
33                                     yytext() + "\"");     }
```

Abbildung 6.7: *JFlex*-Spezifikation des Scanners.

(b) in den Zeilen 25 – 30 werden Operatoren und Klammer-Symbole erkannt,

(c) in Zeile 31 werden Leerzeichen, Tabulatoren und Zeilenumbrüche aus der Eingabe herausgefiltert und

(d) alle weiteren Zeichen werden in Zeile 32 als Fehler gemeldet.

Die einzelnen Token, die vom Scanner erkannt werden, sind Objekte der Klasse `Token`, die in Abbildung 6.8 gezeigt ist. Die Member-Variable `mImage` speichert für jedes Token den Text, der diesem Token zugeordnet ist. Falls es sich bei dem Token um eine Zahl handelt, wird zusätzlich noch der Wert der Zahl abgespeichert.

Jetzt können wir uns dem eigentlichen Parser zuwenden, dessen Implementierung in den Abbildungen 6.9 und 6.10 auf den Seiten 76 und 77 gezeigt ist.

1. Die Klasse `ExprParser` enthält zwei Member-Variablen:

   (a) `mTokenList` ist die Liste der Token, die vom Scanner geliefert werden.

```
1  public class Token {
2      private String mImage;
3
4      public Token(String image) {
5          mImage = image;
6      }
7      public String getImage() { return mImage; }
8  };
9
10 class NumberToken extends Token {
11     private Integer mNumber;
12
13     NumberToken(String number) {
14         super(number);
15         mNumber = new Integer(number);
16     }
17     Integer getNumber() { return mNumber; }
18 }
```

Abbildung 6.8: Die Klasse `Token`.

(b) `mIndex` gibt den Index des nächsten zu verarbeitenden Tokens an. Jedesmal, wenn ein Token vom Parser erkannt wurde, wird dieser Index inkrementiert. Ist das Parsen erfolgreich, so zeigt der Index am Ende auf das Listenende.

Diese Variablen werden im Konstruktor initialisiert.

2. Die Methode *parseExpr*() implementiert die Grammatik-Regel

$$Expr \rightarrow Product \;\; ExprRest.$$

Daher wird zunächst versucht, ein *Product* zu erkennen und anschließend wird versucht, einen *ExprRest* zu erkennen.

(a) Falls das Ende der Token-Liste erreicht ist, wird das berechnete Ergebnis zurückgegeben.

(b) Das berechnete Ergebnis wird ebenfalls zurückgegeben, wenn als nächstes Token ein schließende Klammer "`)`" in der Eingabe gefunden wird.

(c) In allen anderen Fällen liegt ein Syntax-Fehler vor, der eine Ausnahme auslöst. Diese Fälle werden in Zeile 21 – 23 abgeprüft.

(d) Das `return`-Statement in Zeile 24 kann nie erreicht werden. Es ist nur vorhanden, weil der *Java*-Compiler sonst einen fehlenden Rückgabewert reklamieren würde.

3. Die Methode *parseExprRest*() implementiert die Grammatik-Regeln

$$\begin{array}{lcl} ExprRest & \rightarrow & \text{"+"} \; Product \; ExprRest \\ & | & \text{"-"} \; Product \; ExprRest \\ & | & \varepsilon \end{array}$$

Zunächst wird mit Hilfe der Methode *check*() überprüft, ob das nächste Token ein "`+`"– oder ein "`-`"–Zeichen ist. Falls es sich um keines dieser Zeichen handelt, kann nur noch die letzte Regel in Frage kommen.

```
1  import java.util.*;
2
3  public class ExprParser {
4      private List<Token> mTokenList; // list of all tokens
5      private int         mIndex;     // index to the next token
6
7      public ExprParser(List<Token> tokenList) {
8          mTokenList = tokenList;
9          mIndex     = 0;
10     }
11
12     public Integer parseExpr() throws ParseError {
13         Integer product = parseProduct();
14         Integer result  = parseExprRest(product);
15         if (mIndex == mTokenList.size()) {
16             return result;
17         }
18         if (mTokenList.get(mIndex).getImage().equals(")")) {
19             return result;
20         }
21         if (mIndex < mTokenList.size()) {
22             throw new ParseError("Parse Error");
23         }
24         return result;
25     }
26     private Integer parseExprRest(Integer sum) throws ParseError {
27         if (check("+")) {
28             Integer product = parseProduct();
29             return parseExprRest(sum + product);
30         }
31         if (check("-")) {
32             Integer product = parseProduct();
33             return parseExprRest(sum - product);
34         }
35         return sum;
36     }
37     private Integer parseProduct() throws ParseError {
38         Integer factor = parseFactor();
39         return parseProductRest(factor);
40     }
```

Abbildung 6.9: Top-Down-Parser für arithmetische Ausdrücke, 1. Teil

4. Analog implementieren die Methoden *parseProduct*() und *parseProductRest*() die folgenden Grammatik-Regeln:

$$
\begin{array}{lcl}
\textit{Product} & \rightarrow & \textit{Factor ProductRest} \\
\textit{ProductRest} & \rightarrow & \text{“}*\text{”}\ \textit{Factor ProductRest} \\
 & | & \text{“}/\text{”}\ \textit{Factor ProductRest} \\
 & | & \varepsilon
\end{array}
$$

```
41      private Integer parseProductRest(Integer product) throws ParseError {
42          if (check("*")) {
43              Integer factor = parseFactor();
44              return parseProductRest(product * factor);
45          }
46          if (check("/")) {
47              Integer factor = parseFactor();
48              return parseProductRest(product / factor);
49          }
50          return product;
51      }
52      private Integer parseFactor() throws ParseError {
53          if (check("(")) {
54              Integer expr = parseExpr();
55              if (!check(")")) {
56                  throw new ParseError("Parse Error, ')' expected");
57              }
58              return expr;
59          }
60          return parseNumber();
61      }
62      private boolean check(String token) {
63          if (mIndex == mTokenList.size()) {
64              return false;
65          }
66          if (mTokenList.get(mIndex).getImage().equals(token)) {
67              ++mIndex;
68              return true;
69          }
70          return false;
71      }
72      private Integer parseNumber() throws ParseError {
73          if (mIndex == mTokenList.size()) {
74              throw new ParseError("Parse Error, number expected");
75          }
76          Token token = mTokenList.get(mIndex);
77          if (!(token instanceof NumberToken)) {
78              throw new ParseError("Parse Error, number expected");
79          }
80          ++mIndex;
81          NumberToken numberToken = (NumberToken) token;
82          return numberToken.getNumber();
83      }
84  }
```

Abbildung 6.10: Top-Down-Parser für arithmetische Ausdrücke, 2. Teil.

5. Die Methode *parseFactor*() implementiert schließlich die Grammatik-Regeln

$$\begin{array}{rcl} \textit{Factor} & \rightarrow & \text{“(”}\ \textit{Expr}\ \text{“)”} \\ & | & \textsc{Number} \end{array}$$

```
1  public class ParseError extends Exception {
2      public ParseError(String errorMsg) { super(errorMsg); }
3  }
```

Abbildung 6.11: Die Klasse `ParseError`.

6. Die Methode *check*(o) überprüft, ob das nächste zu lesende Token den Operator o darstellt. Dazu muss zunächst überprüft werden, ob überhaupt noch Token vorhanden sind, die bisher noch nicht verarbeitet wurden. Wenn dies der Fall ist und der Vergleich erfolgreich ist, wurde das Token erkannt und der Zähler `mIndex` wird inkrementiert, da das entsprechende Token nun erfolgreich gelesen wurde.

7. Die Methode *parseNumber* überprüft schließlich, ob das nächste zu lesende Token eine Zahl darstellt. Dazu wird zunächst überprüft, ob überhaupt noch Token vorhanden sind. Ist dies der Fall, so wird der Typ des Tokens überprüft. Falls das Token kein `NumberToken` ist, wird eine Ausnahme ausgelöst. Ansonsten wird der Wert des Tokens als Zahl zurückgegeben.

Zum Abschluss zeigt die Abbildung 6.11 die Definition der Klasse `ParseError`. Diese Klasse wird benötigt um sinnvolle Fehlermeldungen erzeugen zu können. Der soeben diskutierte Parser für arithmetische Ausdrücke ist in der vorliegenden Form für die Praxis noch ungeeignet, denn die Fehlermeldungen reichen nicht aus, um in komplexeren Ausdrücken einen Fehler tatsächlich lokalisieren zu können. Wir werden später noch genauer analysieren, welche Bedingungen eine Grammatik erfüllen muss, damit es möglich ist, auf die oben dargestellte Weise einen Top-Down-Parser zu entwickeln.

6.2.3 Implementierung eines rückwärts arbeitenden Top-Down-Parser

Wir zeigen nun, wir wir die in Abbildung 6.5 gezeigte Grammatik ohne Entfernung der Links-Rekursion mit einem Top-Down-Parser implementieren können. Die entscheidende Idee ist hier, die Token-Liste rückwärts abzuarbeiten. Wir benutzen wieder den in Abbildung 6.7 gezeigten Scanner. Die Abbildungen 6.12 und 6.13 zeigen die Implementierung des rückwärts-arbeitenden Top-Down-Parser für arithmetische Ausdrücke.

1. Die Klasse `ExprParser` enthält die Member-Variablen `mTokenList` und `mIndex`. Erstere ist die Liste alle Token, Letztere bezeichnet den Index des Tokens, was als nächstes untersucht wird. Da die Liste der Token nun von hinten nach vorne abgearbeitet wird, wird diese Variable in Zeile 9 so initialisiert, dass sie auf das letzte gelesen Token zeigt. Da in *Java* Felder mit mit 0 beginnen, hat der Index des letzten gelesenen Tokens den Wert `mTokenList.size`() − 1.

2. Die drei Grammatik-Regeln

   $$\mathit{Expr} \;\rightarrow\; \mathit{Expr}\text{ “+” }\mathit{Product} \;\mid\; \mathit{Expr}\text{ “−” }\mathit{Product} \;\mid\; \mathit{Product}$$

   werden durch die Methode *parseExpr*() implementiert. Diese Regeln zeigen, dass eine *Expr* auf jeden Fall mit einem *Product* endet. Daher parsen wir in Zeile 12 zunächst ein *Product* und speichern den Wert dieses Produktes in der Variablen `product`. Falls vor dem Produkt ein “+”- oder ein “−”-Zeichen kommt, lesen wir dieses Zeichen und versuchen danach durch einen rekursiven Aufruf der Methode *parseExpr*(), den Ausdruck, der vor diesem Zeichen steht, zu parsen. Der dabei erhaltene Wert wird in der Variablen `expression` gespeichert. Abhängig davon, ob wir ein “+”- oder “−”-Zeichen gelesen haben, geben wir dann die Summe `expression` + `product` oder die Differenz `expression` − `product` als Ergebnis zurück.

   Falls vor dem ersten Produkt kein “+”- oder “−”-Zeichen steht, besteht der Ausdruck nur aus einem Produkt und wir geben den Wert dieses Produktes zurück.

```
1  public class ExprParser {
2      private List<Token> mTokenList; // list of all tokens
3      private int         mIndex;     // index of next token to consume
4
5      public ExprParser(List<Token> tokenList) {
6          mTokenList = tokenList;
7          mIndex     = mTokenList.size() - 1;
8      }
9      public Integer parseExpr() throws ParseError {
10         Integer product = parseProduct();
11         if (check("+")) {
12             Integer expression = parseExpr();
13             return expression + product;
14         }
15         if (check("-")) {
16             Integer expression = parseExpr();
17             return expression - product;
18         }
19         return product;
20     }
21     private Integer parseProduct() throws ParseError {
22         Integer factor = parseFactor();
23         if (check("*")) {
24             Integer product = parseProduct();
25             return product * factor;
26         }
27         if (check("/")) {
28             Integer product = parseProduct();
29             return product / factor;
30         }
31         return factor;
32     }
33     private Integer parseFactor() throws ParseError {
34         if (check(")")) {
35             Integer expression = parseExpr();
36             if (!check("(")) {
37                 throw new ParseError("Parse Error, '(' expected");
38             }
39             return expression;
40         }
41         return parseNumber();
42     }
```

Abbildung 6.12: Ein rückwärts-arbeitender Top-Down-Parser für arithmetische Ausdrücke, 1. Teil.

3. Die drei Grammatik-Regeln

$$\begin{array}{rcl} \textit{Product} & \rightarrow & \textit{Product} \text{ “*” } \textit{Factor} \\ & | & \textit{Product} \text{ “/” } \textit{Factor} \\ & | & \textit{Factor} \end{array}$$

werden auf analoge Weise in der Methode *parseProduct*() umgesetzt.

```
43      private boolean check(String token) {
44          if (mIndex < 0) {
45              return false;  // all tokens consumed
46          }
47          if (mTokenList.get(mIndex).getImage().equals(token)) {
48              --mIndex;
49              return true;
50          }
51          return false;
52      }
53      private Integer parseNumber() throws ParseError {
54          Token token = mTokenList.get(mIndex);
55          if (!(token instanceof NumberToken)) {
56              throw new ParseError("Parse Error, number expected");
57          }
58          --mIndex;
59          NumberToken numberToken = (NumberToken) token;
60          return numberToken.getNumber();
61      }
62  }
```

Abbildung 6.13: Ein rückwärts-arbeitender Top-Down-Parser für arithmetische Ausdrücke, 2. Teil.

4. Die Methode *parseFactor*() implementiert die beiden Regeln

   $$\begin{array}{lcl} \textit{Factor} & \rightarrow & \text{“(”}\ \textit{Expr}\ \text{“)”} \\ & | & \textit{Number} \end{array}$$

5. Die Methode *check*() ändert sich gegenüber der ursprünglichen Implementierung, da die Token-Liste ja jetzt rückwärts abgearbeitet wird. Deswegen testen wir nun in Zeile 44, ob `mIndex` negativ ist, denn dann hätten wir alle Token abgearbeitet. Andernfalls vergleichen wir das durch `mIndex` spezifizierte Token mit dem String, der als Argument übergeben wird. Falls dieser Vergleich erfolgreich ist, wird der Zeiger `mIndex` nun dekrementiert.

6. Die Methode *parseNumber*() überprüft, ob das Token, das an der durch den Zeiger `mIndex` spezifizierten Position steht, eine Zahl ist und gibt diese gegebenenfalls zurück.

6.2.4 Erzeugung und Auswertung eines Syntax-Baums

Die in den letzten beiden Abscnitten vorgestellten Parser haben unmittelbar eine Zahl als Ergebnis berechnet. Normalerweise berechnet der Parser zunächst nur einen abstrakten Syntax-Baum, der dann anschließend von einem weiteren Programm ausgewertet wird. Als Beispiel werden wir einen Parser für arithmetische Ausdrücke entwickeln, der diese zunächst als abstrakte Syntax-Bäume abspeichert. Anschließend können wir diese arithmetischen Ausdrücke symbolisch nach einer Variablen differenzieren. Abbildung 6.14 zeigt eine Grammatik, die gegenüber der ursprünglichen Grammatik aus Abbildung 6.5 so erweitert worden ist, dass nun auch Funktionsaufrufe der Form

$$exp(t) \quad \text{und} \quad ln(t)$$

für einen beliebigen Ausdruck t zugelassen sind. Außerdem können arithmetische Ausdrücke Variablen enthalten.

Expr	→	*Expr* "`+`" *Product*
	|	*Expr* "`-`" *Product*
	|	*Product*
Product	→	*Product* "`*`" *Factor*
	|	*Product* "`/`" *Factor*
	|	*Factor*
Factor	→	"`(`" *Expr* "`)`"
	|	"`exp`" "`(`" *Expr* "`)`"
	|	"`ln`" "`(`" *Expr* "`)`"
	|	NUMBER
	|	VARIABLE

Abbildung 6.14: Erweiterte Grammatik für arithmetische Ausdrücke.

Als erstes entwickeln wir wieder einen Scanner für die so erweiterte Syntax. Abbildung 6.15 zeigt einen *JFlex*-Scanner. Gegenüber dem ersten in Abbildung 6.7 gezeigten Scanner gibt es die folgenden Unterschiede:

1. Der Aufruf der Methode *parseExpr*() in Zeile 18 liefert nun keine Zahl sondern einen abstrakten Syntax-Baum als Ergebnis zurück. Solche Syntax-Bäume werden wir durch die abstrakte Klasse *Expr* und die davon abgeleiteten Klassen modellieren. Wir werden diese Klassen später noch im Detail diskutieren.

2. Die Klasse *Expr* ist abstrakt und deklariert eine Methode *diff*(x), die als Argument einen String x erhält, der als Variable interpretiert wird, nach welcher der Ausdruck abgeleitet werden soll. Abbildung 6.16 zeigt die Klasse *Expr*. Von dieser Klasse werden wir später konkrete Klassen zur Darstellung von Summen, Differenzen, Produkten, Quotienten, Anwendungen der Exponential- und Logarithmus-Funktion, Zahlen und Varablen ableiten.

3. In Zeile 29 und 30 erkennen wir die Schlüsselwörter "`exp`" und "`ln`".

4. In Zeile 31 erkennen wir einzelne Buchstaben als Variablen. Wir haben von der Klasse *Token* nun zusätzlich die in Abbildung 6.17 gezeigte Klasse *VariableToken* abgeleitet. Diese neue Klasse dient dazu, solche Token zu repräsentieren, die Variablen darstellen.

```
import java.util.*;

%%

%class Scanner
%standalone

%unicode

%{
    List<Token> mTokenList = new ArrayList<Token>();
%}

%eof{
    System.out.println("Length: " + mTokenList.size());
    ExprParser parser = new ExprParser(mTokenList);
    try {
        Expr expr       = parser.parseExpr();
        Expr derivation = expr.diff("x");
        System.out.println("d " + expr + " / dx = " + derivation);
    } catch (ParseError e) {
        System.out.println(e);
    }
%eof}

%%

[1-9][0-9]*  { mTokenList.add(new NumberToken(yytext()));   }
"exp"        { mTokenList.add(new Token("exp"));            }
"ln"         { mTokenList.add(new Token("ln"));             }
[a-z]        { mTokenList.add(new VariableToken(yytext())); }
"+"          { mTokenList.add(new Token("+"));              }
"-"          { mTokenList.add(new Token("-"));              }
"*"          { mTokenList.add(new Token("*"));              }
"/"          { mTokenList.add(new Token("/"));              }
"("          { mTokenList.add(new Token("("));              }
")"          { mTokenList.add(new Token(")"));              }
[ \t\n\r]    { /* skip */                                   }
.            { System.out.println("Unexpected char: \"" +
                                  yytext() + "\"");        }
```

Abbildung 6.15: Scanner für erweiterte arithmetische Ausdrücke

```
public abstract class Expr {
    public abstract Expr diff(String var);
}
```

Abbildung 6.16: Die abstrakte Klasse *Expr*.

Zur Darstellung eines abstrakten Syntax-Baums leiten wir von der Klasse *Expr* die folgenden

```
1  class VariableToken extends Token {
2
3      VariableToken(String number) {
4          super(number);
5      }
6  }
```

Abbildung 6.17: Die Klasse *VariableToken*.

Klassen ab:

1. Die Klasse *Sum* repräsentiert eine Summe.
2. Die Klasse *Difference* repräsentiert eine Differenz.
3. Die Klasse *Product* repräsentiert ein Produkt.
4. Die Klasse *Quotient* repräsentiert einen Quotienten.
5. Die Klasse *Exponential* repräsentiert eine Anwendung der Exponential-Funktion *exp*().
6. Die Klasse *Logarithm* repräsentiert eine Anwendung der Logarithmus-Funktion *ln*().
7. Die Klasse *Number* repräsentiert eine natürliche Zahl.
8. Die Klasse *Variable* repräsentiert eine Variable.

```
1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Sum(mLhs.diff(x), mRhs.diff(x));
11     }
12     public String toString() {
13         return "(" + mLhs.toString() + " + " + mRhs.toString() + ")";
14     }
15 }
```

Abbildung 6.18: Die Klasse *Sum* zur Darstellung von Summen.

Wir diskutieren die in Abbildung 6.18 gezeigte Klasse `Sum` nun im Detail.

1. Die Klasse *Sum* stellt eine Summe der Form

   $$lhs + rhs$$

   dar. Die in Zeile 2 und 3 deklarierten Member-Variablen `mLhs` und `mRhs` repräsentieren die Summanden *lhs* und *rhs*, die ihrerseits wieder durch die abstrakte Klasse `Expr` dargestellt werden.

2. Der Konstruktor initialisiert die Member-Variablen `mLhs` und `mRhs`.

3. Da die Ableitung einer Summe $f+g$ nach einer Variablen x durch die Summe der Ableitungen gegeben ist, denn es gilt

   $$\frac{d}{dx}(f+g) = \frac{d}{dx}f + \frac{d}{dx}g,$$

   können wir die Methode *diff* dadurch implementieren, dass wir die Summanden einzeln ableiten und die Ableitungen zu einer neuen Summe zusammensetzen.

4. Schließlich enthält die Klasse eine Methode *toString*() mit der die Summe ausgegeben werden kann. Beachten Sie, dass das Ergebnis in Klammern gesetzt wird. Dies ist deshalb notwendig, weil die Summe ja Teil eines komplexeren Ausdrucks sein kann, beispielsweise Teil eines Produktes. Beispielsweise ist die Summe $2+3$ Teil des Produktes $(2+3)*4$ und hier müssen offenbar Klammern gesetzt werden.

Als nächstes diskutieren wir die in Abbildung 6.19 gezeigte Klasse `Product`.

1. Die Klasse *Product* stellt ein Produkt der Form

   $$lhs * rhs$$

   dar. Die in Zeile 2 und 3 deklarierten Member-Variablen `mLhs` und `mRhs` repräsentieren die Faktoren *lhs* und *rhs*.

2. Der Konstruktor initialisiert die Member-Variablen `mLhs` und `mRhs`.

3. Die Ableitung eines Produktes $f * g$ nach einer Variablen x ist durch die Produkt-Regel gegeben, es gilt

   $$\frac{d}{dx}(f * g) = \frac{d}{dx}f * g + f * \frac{d}{dx}.$$

   Wir leiten also zunächst die beiden Faktoren einzeln ab und setzen das Ergebnis entsprechend der Produkt-Regel aus diesen Ableitungen zusammen.

4. Schließlich enthält die Klasse eine Methode *toString*() mit der das Produkt-Regel ausgegeben werden kann. Auch hier ist is notwendig Klammern zu setzen, denn sonst würde beispielsweise ein Ausdruck der Form $x/(y * z)$ falsch interpretiert.

Die Implementierung der restlichen Klassen verläuft analog und wird aus Platzgründen nicht weiter diskutiert.

In den Abbildungen 6.20, 6.21 und 6.22 zeigen wir schließlich die Implementierung des eigentlichen Parsers. Der Parser ist ein rückwärts arbeitender Top-Down-Parser, der analog zu den in den Abbildungen 6.12 und 6.13 gezeigten Parser arbeitet. Es gibt zwei wesentliche Unterschiede:

1. Da die Aufgabe des Parsers jetzt darin besteht, einen abstrakten Syntax-Baum zu erstellen, ist der Rückgabewert der verschiedenen Parse-Methoden nun nicht mehr `Integer` sondern `Expr`. Dementsprechend werden die Ergebnisse durch Aufrufe der entsprechenden Konstruktor-Methoden der von der Klasse `Expr` abgeleiteten konkreten Methoden berechnet.

2. Da die Grammatik nun auch Exponentialfunktion und Logarithmus sowie Variablen zuläßt, wurde die Methode *parseFactor*() entsprechend geändert.

```
1  public class Product extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Product(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         Expr lhsDiff = mLhs.diff(x);
11         Expr rhsDiff = mRhs.diff(x);
12         return new Sum(new Product(lhsDiff, mRhs),
13                        new Product(mLhs, rhsDiff));
14     }
15     public String toString() {
16         return "(" + mLhs.toString() + " * " + mRhs.toString() + ")";
17     }
18 }
```

Abbildung 6.19: Die Klasse *Product* zur Darstellung von Produkten.

```
1  import java.util.*;
2
3  public class ExprParser {
4      private List<Token> mTokenList; // list of all tokens
5      private int         mIndex;     // index to the next token
6
7      public ExprParser(List<Token> tokenList) {
8          mTokenList = tokenList;
9          mIndex     = mTokenList.size() - 1;
10     }
11     public Expr parseExpr() throws ParseError {
12         Expr product = parseProduct();
13         if (check("+")) {
14             Expr expression = parseExpr();
15             return new Sum(expression, product);
16         }
17         if (check("-")) {
18             Expr expression = parseExpr();
19             return new Difference(expression, product);
20         }
21         return product;
22     }
23     private Expr parseProduct() throws ParseError {
24         Expr factor = parseFactor();
25         if (check("*")) {
26             Expr product = parseProduct();
27             return new Product(product, factor);
28         }
29         if (check("/")) {
30             Expr product = parseProduct();
31             return new Quotient(product, factor);
32         }
33         return factor;
34     }
```

Abbildung 6.20: Implementierung der Klasse *ExprParser*, 1. Teil.

```
35      private Expr parseFactor() throws ParseError {
36          if (check(")")) {
37              Expr expression = parseExpr();
38              if (!check("(")) {
39                  throw new ParseError("Parse Error, '(' expected");
40              }
41              if (check("exp")) {
42                  return new Exponential(expression);
43              }
44              if (check("ln")) {
45                  return new Logarithm(expression);
46              }
47              return expression;
48          }
49          if (checkNumber()) {
50              return parseNumber();
51          }
52          return parseVariable();
53      }
54
55      private Expr parseNumber() {
56          Token token = mTokenList.get(mIndex);
57          --mIndex;
58          NumberToken numberToken = (NumberToken) token;
59          return new Number(numberToken.getNumber());
60      }
61
62      private Expr parseVariable() {
63          Token token = mTokenList.get(mIndex);
64          --mIndex;
65          VariableToken varToken = (VariableToken) token;
66          return new Variable(varToken.getImage());
67      }
```

Abbildung 6.21: Implementierung der Klasse *ExprParser*, 2. Teil.

```
68      private boolean check(String token) {
69          if (mIndex < 0) {
70              return false;  // all tokens consumed
71          }
72          if (mTokenList.get(mIndex).getImage().equals(token)) {
73              --mIndex;
74              return true;
75          }
76          return false;
77      }
78
79      private boolean checkNumber() {
80          if (mIndex < 0) {
81              return false;  // all tokens consumed
82          }
83          if (mTokenList.get(mIndex) instanceof NumberToken) {
84              --mIndex;
85              return true;
86          }
87          return false;
88      }
89  }
```

Abbildung 6.22: Implementierung der Klasse *ExprParser*, 3. Teil.

6.2.5 Implementierung eines backtrackenden Parsers

In der Sprache *Prolog* basiert die Abarbeitung eines Programms auf dem Prinzip des *Backtrackings*. Falls wir darauf achten, dass der Suchraum beschränkt bleibt, können wir Grammatik-Regeln unmittelbar durch Prolog-Klauseln umsetzen. Eine Beschränkung des Suchraums erhalten wir beispielsweise, indem wir die Rekursionstiefe begrenzen. Setzen wir diese Grenze sukzessive so lange hoch bis wir sicher sind, dass eine mögliche Ableitung auch gefunden wird, so erhalten wir ein Verfahren, dass praktikabel ist, wenn der zu parsende Text nicht zu groß wird. Das in Abbildung 6.23 gezeigte Programm demonstriert das Verfahren an Hand der Grammatik für arithmetische Ausdrücke.

1. Das Prädikat `parseExpr/4` wird in der Form

   $$parseExpr(L, D, R, E)$$

   aufgerufen. Für die Parameter gilt dabei:

   (a) L ist eine Liste von Token, die zu parsen ist. Dabei wird allerdings nicht gefordert, dass alle Token tatsächlich auch verwendet werden, der Parser kann also durchaus auch nur einen Teil der Liste L verwenden.

   (b) D ist eine natürliche Zahl, welche die Anzahl der rekursiven Aufrufe begrenzt. L und D sind zusammen die Eingabe-Parameter.

   (c) R enthält nach erfolgreicher Ausführung des Prädikats alle die Token aus der Liste L, die beim parsen noch nicht verwendet wurden.

   (d) E enthält nach erfolgreicher Ausführung das Ergebnis der Berechnung des geparsten Ausdrucks.

```
1  parseExpr(L, Depth, R, E) :-
2          parseProduct(L, Depth, R, E).
3
4  parseExpr(L, Depth, R, E) :-
5          D is Depth - 1,
6          D > 0,
7          parseExpr(L, D, R1, E1),
8          parseToken(R1, '+', R2),
9          parseProduct(R2, Depth, R, P),
10         E is E1 + P.
11
12 parseExpr(L, Depth, R, E) :-
13         D is Depth - 1,
14         D > 0,
15         parseExpr(L, D, R1, E1),
16         parseToken(R1, '-', R2),
17         parseProduct(R2, Depth, R, P),
18         E is E1 - P.
19
20 parseProduct(L, Depth, R, P) :-
21         parseFactor(L, Depth, R, P).
22
23 parseProduct(L, Depth, R, P) :-
24         D is Depth - 1,
25         D > 0,
26         parseProduct(L, D, R1, P1),
27         parseToken(R1, '*', R2),
28         parseFactor(R2, Depth, R, F),
29         P is P1 * F.
30
31 parseProduct(L, Depth, R, P) :-
32         D is Depth - 1,
33         D > 0,
34         parseProduct(L, D, R1, P1),
35         parseToken(R1, '/', R2),
36         parseFactor(R2, Depth, R, F),
37         P is P1 / F.
38
39 parseFactor(L, Depth, R, F) :-
40         parseToken(L, '(', R1),
41         parseExpr(R1, Depth, R2, F),
42         parseToken(R2, ')', R).
43
44 parseFactor(L, _Depth, R, N) :-
45         parseNumber(L, R, N).
```

Abbildung 6.23: Ein *Prolog*-Parser für arithmetische Ausdrücke

Die drei Klauseln der Implementierung dieses Prädikats setzen die Grammatik-Regeln

$$\begin{array}{lcl} \textit{Expr} & \rightarrow & \textit{Product} \\ & | & \textit{Expr} \text{ “+” } \textit{Product} \end{array}$$

| *Expr* "-" *Product*

in der angegebenen Reihenfolge um. Wir erläutern dies am Beispiel der zweiten Prolog-Klausel, die in der Abbildung 6.23 in Zeile 4 beginnt: Um einen arithmetischen Ausdruck mit der Regel

Expr → *Expr* "+" *Product*

zu erkennen, wird zunächst rekursiv versucht, in L einen arithmetischen Ausdruck mit dem Wert $E1$ zu erkennen. Um eine Endlos-Schleife zu vermeiden, wird dabei die erlaubte Suchtiefe um Eins dekrementiert. Ist dieser Versuch erfolgreich und bleibt danach von der ursprünglich gegebenen Liste L eine Restliste R übrig, so wird anschließend versucht, in dieser Restliste das Token "+" zu erkennen. Das Prädikat *parseToken* entfernt dann gegebenenfalls dieses Token aus der Liste $R1$, wobei die Liste $R2$ übrig bleibt. In dieser Liste versucht nun das Prädikat *parseProduct* ein Produkt zu erkennen. Alle Token, die dabei nicht verbraucht werden, werden in der Liste R aufgesammelt und das Produkt selber wird in der Variablen P zur Verfügung gestellt. Abschließend wird das Ergebnis E als Summe von $E1$ und P dargestellt.

Die dritte Klausel setzt in analoger Weise die Grammatik-Regel

Expr → *Expr* "-" *Product*

um. Die Grammatik-Regel

Expr → *Product*

wird durch die erste Regel implementiert.

2. Die Implementierung der Prädikate `parseProduct/4` und `parseFactor/4` verläuft analog und wird daher nicht weiter diskutiert.

3. Abbildung 6.24 zeigt die Implementierung einiger Hilfsprädikate.

   (a) Das Prädikat `parseNumber/3` überprüft, ob das nächste zu lesende Token eine Zahl ist und entfernt diese gegebenenfalls.

   (b) Der Aufruf *parseToken*(L, T, R) überprüft, ob das erste Element der Liste L das Token T ist und entfernt dieses gegebenenfalls aus der Liste L. Diese verkürzte Liste wird dann gegebenenfalls als die Liste R zurück gegeben.

   (c) Das Prädikat *parse*(L, E) bietet den Einstiegs-Punkt in das Programm. Hier ist L die zu verarbeitende Token-Liste und im Erfolgsfall enthält E das Ergebnis. Ein möglicher Aufruf könnte die folgende Form haben:

   ```
   parse( [ 1, '+', 2, '*', 3 ], E ).
   ```

   (d) Das Hilfsprädikat *parseExpr*(L, D, E) steuert schließlich die Tiefensuche:

   - Zunächst wird versucht, die Liste mit der gegebenen Rekursions-Beschränkung so zu parsen, dass keine Token mehr übrig bleiben. Falls dies erfolgreich ist, wird durch den Cut-Operator "!" sichergestellt, dass die Suche abbricht.
   - Andernfalls wird die zulässige Rekursions-Tiefe um 1 erhöht. Ist die neue Tiefe geringer als die Länge der zu parsenden Tokenliste, so wird erneut versucht, die Liste L zu parsen.

Historisches Die Sprache ALGOL [Bac59, NBB+60] war die erste Programmier-Sprache, deren Syntax auf einer kontextfreien Grammatik basiert.

```
1  parseNumber( [ N | R ], R, N ) :-
2          number(N).
3
4  parseToken( [ T | R ], T, R ).
5
6  parse(L, E) :-
7          parse(L, 1, E).
8
9  parse(L, D, E) :-
10         parseExpr(L, D, [], E), !.
11
12 parse(L, D, E) :-
13         D1 is D + 1,
14         length(L, Length),
15         D1 =< Length,
16         parse(L, D1, E).
```

Abbildung 6.24: Einige Hilfsprädikate für den Parser

Kapitel 7

Antlr — *Another Tool for Language Recognition*

Es gibt eine Reihe von Werkzeugen, die in der Lage sind, einen Top-Down-Parser automatisch zu erzeugen. Das Werkzeug, das in meinen Augen am besten ausgereift ist, ist ANTLR. Der Name steht für *another tool for language recognition*[1]. Sie finden dieses Werkzeug im Internet unter der folgenden Adresse

`http://www.antlr.org`

Das Werkzeug ANTLR wird von dem Entwickler Terence Parr in dem Buch

The Definitive ANTLR *Reference: Building Domain Specific Languages*

[Par07] ausführlich beschrieben. Zunächst führen wir ANTLR an Hand verschiedener Beispiele ein, durch die wir die wichtigsten Eigenschaften des Werkzeugs demonstrieren können. In späteren Kapiteln werden wir die Theorie von Top-Down-Parsern untersuchen.

7.1 Ein Parser für arithmetische Ausdrücke

Wir beginnen mit einem Parser für arithmetische Ausdrücke, die entsprechend der in Abbildung 6.5 auf Seite 72 gezeigten Grammatik aufgebaut sind. Nun ist die dort gezeigte Grammatik linksrekursiv und daher für einen Top-Down-Parser (und solche werden von ANTLR generiert) nicht geeignet. Im letzten Kapitel hatten wir gesehen, wie die Links-Rekursion aus dieser Grammatik beseitigt werden kann und dabei die in Abbildung 6.6 auf Seite 72 gezeigte Grammatik erhalten, die allerdings umfangreicher als die ursprüngliche Grammatik ist und die auch die Struktur der arithmetischen Ausdrücke nicht so klar wie die ursprüngliche Grammatik wiederspiegelt. Um solche Strukturen einfach durch eine Grammatik beschreiben zu können, wurde bei ANTLR das Konzept der Grammatik dadurch erweitert, dass auf der rechten Seite einer Grammatik-Regel auch die Postfix-Operatoren "`*`", "`+`" und "`?`" verwendet werden dürfen. Die Bedeutung dieser Operatoren ist die gleiche wie bei regulären Ausdrücken:

"`*`" steht für eine beliebige Anzahl von Wiederholungen, wobei auch 0 Wiederholungen zugelassen sind.

"`+`" steht für eine beliebige positive Anzahl von Wiederholungen, der Ausdruck muss also mindestens einmal vorkommen.

"`?`" steht für einen optionalen Ausdruck.

[1] Der Name ANTLR kann außerdem auch als Abkürzung für anti LR verstanden werden. Hier steht "LR" für *LR-Parser*. Dies ist eine Klasse von Parsern, die nicht *top-down* sondern statt dessen *bottom-up* arbeiten. Wir werden diese Klasse von Parsern später noch ausführlich diskutieren.

Zusätzlich können Sie die rechten Seiten einer Grammatik-Regel durch Verwendung von Klammern strukturieren. Außerdem können Sie den Infix-Operator "|", der für eine Auswahl zwischen zwei Alternativen steht, verwenden. Damit schreibt sich dann die Grammatik für arithmetische Ausdrücke in der in Abbildung 7.1 auf Seite 93 gezeigten kompakten Form. Grammatiken dieser erweiterten Form werden als EBNF-Grammatiken bezeichnet. Die Abkürzung EBNF steht dabei für extended backus naur form.

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Product}\ ((\ \text{“+”} \mid \text{“-”}\ )\ \mathit{Product})* \\
\mathit{Expr} & \rightarrow & \mathit{Factor}\ \ ((\ \text{“*”} \mid \text{“/”}\ )\ \mathit{Factor})* \\
\mathit{Factor} & \rightarrow & \text{“(”}\ \mathit{Expr}\ \text{“)”} \\
 & \mid & \mathit{Number}
\end{array}
$$

Abbildung 7.1: Erweiterte Grammatik für arithmetische Ausdrücke.

Die in der Abbildung gezeigte Grammatik ist wir folgt zu lesen:

1. Ein arithmetischer Ausdruck startet mit einem Produkt, auf das dann eine Liste weiterer Produkte folgen kann, die entweder durch eine Minus- oder ein Plus-Zeichen abgetrennt sind. Damit hat ein arithmetischer Ausdruck also die Form

   $$p_1 \pm p_2 \pm \cdots \pm p_n.$$

   Die p_i stehen hier für die einzelnen Produkte, aus denen die Summe aufgebaut ist.

2. Analog ist ein Produkt eine Liste von Faktoren, die durch die Zeichen "*" oder "/" voneinander getrennt sind. Wird nur der Multiplikations-Operator verwendet, so hat ein Produkt die folgende Form:

   $$f_1 * f_2 * \cdots * f_n.$$

   Die f_i stehen hier für die einzelnen Faktoren des Produkts.

3. Ein Faktor ist entweder ein geklammerter arithmetischer Ausdruck, oder er ist eine einfache Zahl.

Diese Grammatik läßt sich nun mit Hilfe von ANTLR wie in Abbildung 7.2 gezeigt implementieren. Wir diskutieren diese Umsetzung Zeile für Zeile.

1. In Zeile 1 spezifizieren wir mit dem Schlüsselwort `grammar` den Namen der Grammatik. In unserem Fall hat die Grammatik also den Namen `expr`. Der Name der Datei, in dem die Grammatik abgespeichert ist, wird aus dem Namen der Grammatik durch Anhängen der Endung ".g" gebildet, daher muss die gezeigte Grammatik in einer Datei mit dem Namen "`expr.g`" abgespeichert werden.

2. Zeile 3 enthält die Grammatik-Regel für die syntaktische Variable *expr*. Zu beachten ist hier, dass die Terminale "+" und "*" bei ANTLR in einfache Anführungszeichen gesetzt werden müssen. Linke und rechte Seite einer Grammatik-Regel werden durch einen Doppelpunkt getrennt. Jede Grammatik-Regel wird durch ein Semikolon "`;`" beendet.

3. Entsprechend enthalten die Zeilen 5 und 7 die Grammatik-Regeln für die syntaktischen Variablen *product* und *factor*. Dabei sehen wir, dass die verschiedenen Grammatikregeln, welche die selbe syntaktische Variable definieren, durch das Zeichen "|" von einander getrennt werden.

4. Bei ANTLR werden die grammatikalische Struktur und die lexikalische Struktur der Eingabe in der selben Datei definiert. Um syntaktische Variablen und Terminale von einander unterscheiden zu können, wird vereinbart, dass Terminale mit einem Großbuchstaben beginnen,[2] während syntaktische Variablen immer mit einem kleinen Buchstaben beginnen. Daher wissen wir, dass der String "`NUMBER`" in Zeile 8 ein Terminal bezeichnet.

5. In Zeile 11 wird die lexikalische Struktur der mit `NUMBER` bezeichneten Terminale durch einen regulären Ausdruck definiert. Der reguläre Ausdruck

   ```
   ('0'..'9')+
   ```

   steht hier für eine Folge aus Ziffern, die mindestens eine Ziffer enthält. In *Flex* hätten wir statt dessen den regulären Ausdruck

   ```
   [0-9]+
   ```

   verwendet.

6. Zeile 12 definiert schließlich das Token `WS`, wobei der Name als Abkürzung für *white space* zu verstehen ist. Hier werden Leerzeichen, Tabulatoren, Zeilenumbrüche und Wagenrückläufe erkannt. Nach der lexikalischen Spezifikation von `WS` folgt in geschweiften Klammern noch eine semantische Aktion. Die spezielle Aktion "`skip()`", die hier ausgeführt wird, wirft das erkannte Token einfach weg, ohne es an den Parser weiterzureichen.

Zusammenfassend enthalten die Zeilen 3 – 9 also die Definition der grammatikalischen Struktur, während die Zeilen 11 und 12 die lexikalische Struktur definieren.

```
1   grammar expr;
2
3   expr    : product (('+'|'-') product)* ;
4
5   product : factor (('*'|'/') factor)* ;
6
7   factor  : '(' expr ')'
8           | NUMBER
9           ;
10
11  NUMBER : ('1'..'9')('0'..'9')*;
12  WS     : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 7.2: ANTLR-Spezifikation eines Parsers für arithmetische Ausdrücke.

Um aus der in Abbildung 7.2 gezeigten Grammatik einen Parser erzeugen zu können, übersetzen wir diese Datei mit dem folgenden Befehl:

```
java org.antlr.Tool expr.g
```

Das setzt natürlich voraus, dass die Umgebungs-Variable `CLASSPATH` so gesetzt ist, dass das Paket `org.antlr.Tool` auch gefunden wird. Bei der Übersetzung erhalten wir eine Warnung:

```
warning(138): expr.g:0:0: grammar expr: no start rule
              (no rule can obviously be followed by EOF)
```

Diese Warnung sagt aus, dass ANTLR nicht in der Lage war, die Start-Variable der Grammatik automatisch zu erkennen. Diese Warnung erhalten Sie immer dann, wenn die Start-Variable auch auf der rechten Seite einer Regel auftritt. Bei unserer Grammatik ist dies der Fall, denn die

[2] Es ist Konvention, aber nicht vorgeschrieben, dass die Namen von Terminalen nur aus Großbuchstaben bestehen.

Variablen *expr* tritt auf der rechten Seite der ersten Regel für *factor* auf. Falls ANTLR die Start-Variable nicht automatisch erkennen kann, wird die Variable, die auf der linken Seite der ersten Regel steht, als Start-Variable genommen. Da dies in unserem Fall die Variable *expr* ist, könnten wir die obige Warnung ignorieren. Wir können die Warnung auch beseitigen, in dem wir ein neue Variable *start* als Start-Symbol der Grammatik einführen und für diese Variable die Regel

```
start: expr;
```

hinzufügen. ANTLR erzeugt die folgenden Dateien:

1. `exprParser.java`

   Hier handelt es sich um den eigentlichen Parser.

2. `exprLexer.java`

   Hier ist der Scanner implementiert.

3. `expr.tokens`

   Diese Datei ordnet den verwendeten Token natürliche Zahlen zu, auf die wir zurückgreifen können, wenn wir die Funktionsweise des erzeugten Parsers analysieren möchten.

Um den erzeugten Parser aufrufen zu können, benötigen wir noch ein Treiber-Programm. Abbildung 7.3 zeigt ein solches Programm.

```
1  import org.antlr.runtime.*;
2
3  public class ParseExpr {
4
5      public static void main(String[] args) throws Exception {
6          ANTLRInputStream  input  = new ANTLRInputStream(System.in);
7          exprLexer         lexer  = new exprLexer(input);
8          CommonTokenStream ts     = new CommonTokenStream(lexer);
9          exprParser        parser = new exprParser(ts);
10         parser.expr();
11     }
12 }
```

Abbildung 7.3: Treiber-Programm für den von ANTLR erzeugten Parser.

1. In Zeile 1 importieren wir das Paket `org.antlr.runtime`. Dort sind u.a. die Klassen `ANTLRInputStream` und `CommonTokenStream`, auf deren Verwendung wir angewiesen sind, definiert.

2. In Zeile 6 wandeln wir die Standard-Eingabe in einen `ANTLRInputStream` um, der dann in Zeile 7 dazu benutzt werden kann, einen Scanner zu erzeugen.

3. Mit Hilfe des Scanners erzeugen wir in Zeile 8 einen TokenStream und aus diesem können wir dann in Zeile 9 einen Parser erzeugen.

4. Der Aufruf des Parsers erfolgt in Zeile 10, indem wir den Namen der syntaktischen Variable, die wir parsen möchten, als Methode verwenden.

5. Da die Konstruktoren der Klassen `ANTLRInputStream` und `CommonTokenStream` Ausnahmen auslösen können, deklarieren wir in Zeile 5, dass die Methode `main` eine Ausnahme auslösen kann.

Übersetzen wir dieses Programm, so können wir den Parser durch den Befehl

```
echo "2 * 3 + (5 - 4) / 2" | java ParseExpr
```

testen. Dieser Befehl liefert keinerlei Ausgabe und zeigt lediglich, dass der Ausdruck mit der angegebenen Grammatik auch erkannt werden konnte. Geben wir statt dessen den Befehl

```
echo "2 * + 3" | java ParseExpr
```

ein, so erhalten wir die Fehlermeldung:

```
line 1:4 no viable alternative at input '+'
```

Diese Fehlermeldung sagt aus, dass es in Zeile 1 in der vierten Spalte (Spalten werden mit 0 beginnend gezählt) ein Problem gibt, denn der Parser kann dort mit dem Zeichen "+" nichts anfangen.

7.2 Ein Parser zur Auswertung arithmetischer Ausdrücke

Das letzte Beispiel ist noch nicht sehr spektakulär, weil die vom Parser erkannten Ausdrücke nicht ausgewertet werden. Wir präsentieren jetzt ein komplexeres Beispiel, bei dem arithmetische Ausdrücke ausgewertet und die erhaltenen Ergebnisse in Variablen gespeichert werden können. Wir gehen dabei in zwei Schritten vor und präsentieren zunächst eine reine Grammatik in ANTLR-Notation. Anschließend erweitern wir diese mit Aktionen, in denen die Ausdrücke ausgewertet werden können. Abbildung 7.4 zeigt diese Grammatik. Gegenüber der vorher gezeigten Grammatik für arithmetische Ausdrücke gibt es die folgenden Erweiterungen:

```
1  grammar program;
2
3  program   : stmnt+ ;
4
5  stmnt     : ID '=' expr ';'
6            | expr ';'
7            ;
8
9  expr      : product (('+'|'-') product)* ;
10
11 product   : factor (('*'|'/') factor)* ;
12
13 factor    : '(' expr ')'
14           | ID
15           | INT
16           ;
17
18 ID : ('a'..'z'|'A'..'Z')+;
19 INT: ('0'..'9')+;
20 WS : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 7.4: Eine Grammatik für die Auswertung von Ausdrücken.

1. Das Start-Symbol ist jetzt `program`. Es steht für eine Liste von Zuweisungen der folgenden Form:

   var = *expr*;

Hier ist `var` der Name einer Variablen und *expr* ist ein arithmetischer Ausdruck.

2. `stmnt` bezeichnet eine Zuweisung oder einen einzelnen Ausdruck.

3. Das Terminal `ID` bezeichnet den Namen einer Variablen. Ein solcher Name besteht aus einer beliebigen Folge von Buchstaben.

Mit diesem Parser können wir jetzt zum Beispiel die folgende Eingabe parsen:

```
x = 2 * 3; y = 4 * 5; z = x * x + y * y;
z / 3;
```

Wir wollen nun einen ganz einfachen Interpreter entwickeln, der eine Folge von solchen Zuweisungen auswertet und die bei der Auswertung berechneten Zwischen-Ergebnisse in Variablen abspeichert, auf die in folgenden Ausdrücken Bezug genommen werden kann. Weiterhin soll jeder Ausdruck, der keiner Variablen zugewiesen wird, ausgewertet und ausgegeben werden. Abbildung 7.5 zeigt die Realisierung eines solchen Interpreters mit Antlr.

1. Da wir die Werte der einzelnen Variablen in einer Tabelle abspeichern müssen, importieren wir in den Zeilen 3 – 5 die Klasse `java.util.TreeMap`, denn diese Klasse implementiert Tabellen effizient als binäre Bäume.

   Allgemein setzt Antlr all den Code, der durch das Schlüsselwort "`@header`" spezifiert wird, an den Anfang der erstellten Parser-Datei.

2. In den Zeilen 7 – 9 definieren wir zusätzliche Member-Variablen für die erzeugte Klasse `programParser`. In unserem Fall definieren wir hier die Tabelle, die später die Werte der Variablen enthält, als Abbildung, die Strings ganze Zahlen zuordnet.

   Allgemein setzt Antlr all den Code, der durch das Schlüsselwort "`@members`" spezifiert wird, an den Anfang der erstellten Parser-Klasse. Dieses Feature kann sowohl zur Definition von zusätzlichen Klassen-Variablen als auch von Methoden verwendet werden.

   Wir reichern nun die Grammatik-Regeln mit Aktionen an, die durchgeführt werden, wenn der Parser die entsprechende Grammatik-Regel anwendet. Die Aktionen werden von der eigentlichen Grammatik-Regel, für die sie angewendet werden sollen, dadurch abgesetzt, dass sie in den geschweiften Klammern "{" und "}" eingefaßt werden.

3. Wird vom Parser in Zeile 15 eine Zuweisung der Form *var* = *expr* erkannt, so soll der Wert des Ausdrucks *expr* berechnet und das Ergebnis in der Tabelle `varTable` unter dem Namen *var* eingetragen werden. In der Grammatik-Regel

   stmnt → `ID` '=' *expr* ';'

   haben wir einerseits das Token `ID`, auf dessen Namen wir mit `$ID.text` zugreifen können, andererseits haben wir die syntaktische Variable *expr*. Wir werden später dieser syntaktischen Variable die *Java*-Variable `result` als Ergebnis-Variable zuordnen, die den zugehörigen Wert enthält. Dann können wir mit `$expr.result` auf diesen Wert zugreifen.

   In Zeile 14 haben wir einen einzelnen arithmetischen Ausdruck, den wir auswerten und ausgeben.

4. in Zeile 19 definieren wir mit der Zeile

   ```
   expr returns [int result]
   ```

   dass die Methode, die eine *expr* parst, als Ergebnis ein `int` zurück gibt und dass dieses in der Variable mit dem Namen `result` abgespeichert wird.

5. In der Grammatik-Regel

   ```
   expr : product (('+'|'-') product)* ;
   ```

```
1  grammar program;
2
3  @header {
4      import java.util.TreeMap;
5  }
6
7  @members {
8      TreeMap<String, Integer> varTable = new TreeMap<String, Integer>();
9  }
10
11 program   : stmnt+ ;
12
13 stmnt : ID '=' expr ';' { varTable.put($ID.text, $expr.result); }
14       |        expr ';' { System.out.println($expr.result);     }
15       ;
16
17 expr returns [int result]
18     : p = product { $result = $p.result; }
19       (   ('+' q = product) { $result += $q.result; }
20         | ('-' q = product) { $result -= $q.result; }
21       )*
22     ;
23
24 product returns [int result]
25     : f = factor { $result = $f.result; }
26       (
27           ('*' g = factor) { $result *= $g.result; }
28         | ('/' g = factor) { $result /= $g.result; }
29       )*
30     ;
31
32 factor returns [int result]
33     : '(' expr ')' { $result = $expr.result;           }
34     | ID           { $result = varTable.get($ID.text); }
35     | INT          { $result = new Integer($INT.text); }
36     ;
37
38 ID : ('a'..'z'|'A'..'Z')+;
39 INT: ('0'..'9')+;
40 WS : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 7.5: Ein Interpreter zur Auswertung von Ausdrücken.

taucht die syntaktische Variable *product* zweimal auf. Deswegen wäre ein Notation der Form `$product.result` nicht eindeutig. Daher haben wir in Zeile 20 – 23 den verschiedenen Auftreten dieser syntaktische Variablen die Namen p und q zugeordnet, auf die wir dann in den Aktionen zugreifen können. Dabei muss diesen Variablen allerdings ein Dollar-Zeichen vorgestellt werden.

Die Aktionen selber bestehen nun darin, dass wir der Variablen `result` das Ergebnis der jeweiligen Berechnung zuweisen.

6. In den Zeilen 36 und 37 greifen wir auf die Strings, die den Token `ID` und `INT` entsprechen,

mit Hilfe der für Token vordefinierten Variable `text` zurück.

7.3 Erzeugung abstrakter Syntax-Bäume

Bei der Auswertung arithmetischer Ausdrücke im letzten Abschnitt hatten wir Glück und konnten das Ergebnis eines Ausdrucks unmittelbar mit Hilfe von semantischen Aktionen berechnen. Bei komplexeren Problemen ist es in der Regel erforderlich, zunächst einen abstrakten Syntax-Baum zu erzeugen. Die eigentliche Berechnung findet dann erst nach dem Parsen auf dem Syntax-Baum statt. Wir wollen dieses Verfahren an einem Beispiel demonstrieren. Bei dem Beispiel geht es wieder um die symbolische Differentiation arithmetischer Ausdrücke. Ist beispielsweise der arithmetische Ausdruck

$$x * \ln(x)$$

gegeben, so findet sich für die Ableitung dieses Ausdrucks nach der Variable x mit Hilfe der Produkt-Regel das Ergebnis

$$1 * \ln(x) + x * \frac{1}{x}.$$

Da die arithmetischen Ausdrücke nun zusätzlich zu den Operatoren, die die vier Grundrechenarten beschreiben, auch noch Funktionszeichen für die Exponential-Funktion und den natürlichen Logarithmus enthalten sollen, müssen wir die Grammatik aus dem letzten Abschnitt erweitern. Abbildung 7.6 zeigt die entsprechend erweiterte EBNF-Grammatik.

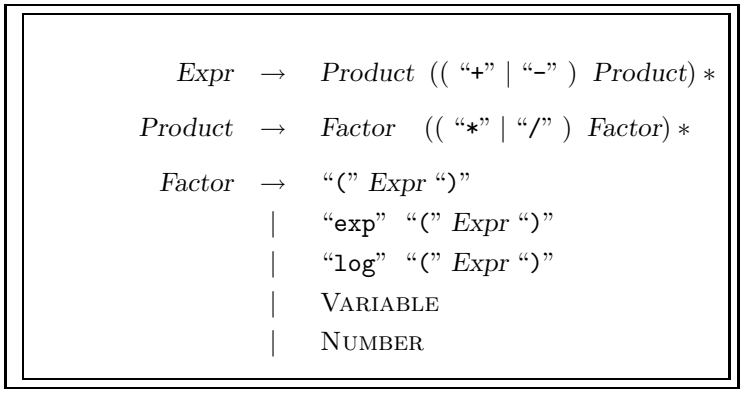


Abbildung 7.6: EBNF-Grammatik für arithmetische Ausdrücke mit Exponential-Funktion und Logarithmus.

7.3.1 Implementierung des Parsers

Abbildung 7.7 zeigt die ANTLR-Implementierung der Grammatik aus Abbildung 7.6. Wir verwenden zur Darstellung der abstrakten Syntax-Bäume die selben Klassen, die wir schon im vorigen Kapitel vorgestellt hatten.

1. In Zeile 3 deklarieren wir durch "`returns [Expr result]`", dass beim Erkennen einer *expr* nun ein Objekt der Klasse `Expr` zurück gegeben werden soll und dass dieses Objekt als Member-Variable mit dem Namen `result` angesprochen werden kann.

2. In Zeile 4 haben wir zunächst ein Produkt erkannt, dass wir unter der Variable p abspeichern. Anschließend weisen wir der Variable `result` dieses Produkt zu. Falls nun später noch ein "`+`"– oder "`-`" Zeichen gefolgt von einem weiteren Ausdruck gelesen wird, so bauen wir aus

```
1  grammar expr;
2
3  expr returns [Expr result]
4      : p = product { $result = $p.result; }
5        (   ('+' q = product) { $result = new Sum(       $result, $q.result); }
6          | ('-' q = product) { $result = new Difference($result, $q.result); }
7        )*
8      ;
9
10 product returns [Expr result]
11     : f = factor { $result = $f.result; }
12       (
13           ('*' g = factor) { $result = new Product( $result, $g.result); }
14         | ('/' g = factor) { $result = new Quotient($result, $g.result); }
15       )*
16     ;
17
18 factor returns [Expr result]
19     : '(' expr ')'       { $result = $expr.result;                  }
20     | 'exp' '(' expr ')' { $result = new Exponential($expr.result); }
21     | 'log' '(' expr ')' { $result = new Logarithm(  $expr.result); }
22     | VAR                { $result = new Variable($VAR.text);       }
23     | NUM                { $result = new Number($NUM.text);         }
24     ;
25
26 VAR : ('a'..'z'|'A'..'Z')+;
27 NUM : ('0'..'9')+;
28 WS  : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 7.7: Die ANTLR-Spezifikation der Grammatik.

dem neu gelesenen Ausdruck und dem alten Wert von `result` den neuen Wert von `result`. Dies kann mehrmals passieren, da dieser Teil der Grammatik-Regel in $(\cdots)*$ eingeschlossen ist.

Beispielsweise wird ein Ausdruck der Form

$$p_1 + p_2 + p_3$$

übersetzt in ein Java-Objekt der Form

$$\texttt{Sum}(\texttt{Sum}(p_1, p_2), p_3)).$$

3. Die weiteren Grammatik-Regeln erzeugen in analoger Weise *Java*-Objekte.

Zum Abschluß zeigt Abbildung 7.8 noch die Einbindung des Parsers. Gegenüber dem in Abbildung 7.3 gezeigten Treiber gibt es nur einen wesentlichen Unterschied: In Zeile 9 wird nun ein Objekt der Klasse `Expr` erzeugt. Für dieses Objekt rufen wir dann in Zeile 10 die Methode *diff*() auf, die die symbolische Ableitung berechnet.

```
1  import org.antlr.runtime.*;
2
3  public class Differentiate {
4      public static void main(String[] args) throws Exception {
5          ANTLRInputStream  input  = new ANTLRInputStream(System.in);
6          exprLexer         lexer  = new exprLexer(input);
7          CommonTokenStream ts     = new CommonTokenStream(lexer);
8          exprParser        parser = new exprParser(ts);
9          Expr expr = parser.expr();
10         Expr diff = expr.diff("x");
11         System.out.println("d (" + expr + ")/dx = " + diff);
12     }
13 }
```

Abbildung 7.8: Ein Treiber für den Parser.

Aufgabe 14: Auf meiner Webseite finden Sie unter

`http://wwwlehre.dhbw-stuttgart.de/`
`~stroetma/Formale-Sprachen/Aufgaben/Grammar2HTML/c-grammar.g`

eine Grammatik für die Sprache `C`.

(a) Entwickeln Sie eine Grammatik, mit der Sie die Syntax der dort angegebenen Regeln beschreiben können.

(b) Entwickeln Sie mit Hilfe von ANTLR ein Programm, dass die angegebene Grammatik liest und als HTML-Datei ausgibt.

Zur Darstellung der Grammatik-Regeln finden Sie in dem Verzeichnis

`~stroetma/Formale-Sprachen/Aufgaben/Grammar2HTML/`

verschiedene Klassen, deren `toString`-Methode `Html`-Ausgabe erzeugt.

Kapitel 8

LL(k)-Sprachen

In diesem Kapitel werden wir die Theorie vorstellen, die Top-Down Parser-Generatoren wie beispielsweise ANTLR zu Grunde liegt. Es handelt sich dabei um die Theorie der LL(k)-Sprachen. Dabei steht das erste L dafür, dass der Parser die Eingabe von links nach rechts parst, das zweite L steht dafür, dass der Parser versucht, eine Links-Ableitung des zu parsenden Wortes zu berechnen. Eine Links-Ableitung ist dabei eine Ableitung, bei der immer die linkeste Variable ersetzt wird. Die Zahl k in LL(k) bedeutet, dass der Parser immer an Hand der nächsten k Token entscheidet, welche Regel als verwendet wird. Falls $k = 1$ ist, wird also nur das nächste Token zur Entscheidung herangezogen. Dieses Token bezeichnen wir dann als *Lookahead-Token*. Wir beginnen mit dem Fall $k = 1$. Bevor wir die Theorie der LL(1)-Parser darstellen, machen wir auf zwei Probleme aufmerksam, die wir bei der Erstellung von Top-Down-Parsern lösen müssen.

1. Das erste Problem ist Links-Rekursion. Betrachten wir die dazu die folgenden Grammatik-Regeln für arithmetische Ausdrücke:

   $$\begin{array}{lcl} \mathit{Expr} & \rightarrow & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\ & | & \mathit{Expr}\ \text{“}\texttt{-}\text{”}\ \mathit{Product} \\ & | & \mathit{Product} \end{array}$$

   Die ersten beiden Regeln sind links-rekursiv und daher mit einem Top-Down-Parser in der vorliegenden Form nicht verwendbar. Wir hatten dieses Problem bereits im Abschnitt 6.2.1 besprochen und gezeigt, wie die Links-Rekursion in einer Grammatik eliminiert werden kann.

   Zusätzlich ist es bei Verwending von EBNF-Grammatiken oft möglich, die Links-Rekursion durch die Verwendung der Postfix-Operatoren “+” und “*” zu vermeiden.

2. Das zweite Problem erkennen wir, wenn wir die folgende Grammatik für Boole'sche Ausdrücke betrachten:

   $$\begin{array}{lcl} \mathit{boolExpr} & \rightarrow & \mathit{expr}\ \text{“}\texttt{==}\text{”}\ \mathit{expr} \\ & | & \mathit{expr}\ \text{“}\texttt{<}\text{”}\ \ \mathit{expr} \end{array}$$

   Diese Grammatik-Regeln sind nicht links-rekursiv, aber für einen Top-Down-Parser, der mit nur einem Token Look-Ahead auskommen soll, ist die Frage, welche der beiden Regeln zum Parsen verwendet werden soll, offenbar nicht zu beantworten. Wir stellen gleich ein Verfahren vor, mit dem sich Grammatiken so transformieren lassen, dass das Problem verschwindet.

8.1 Links-Faktorisierung

Ist A ein Nicht-Terminal und gibt es zwei verschiedene Regeln, mit denen A abgeleitet werden kann, beispielsweise

$$A \to \beta \quad \text{und} \quad A \to \gamma,$$

so muss es bei Verwendung eines LL(1)-Parsers möglich sein, an Hand des Look-Ahead-Tokens zu erkennen, welche Regel benutzt werden soll. In der Praxis gibt es häufig Situationen, wo diese Voraussetzung nicht erfüllt ist. Wir haben oben bereits ein solches Beispiel gesehen. Um das Beispiel zu vervollständigen, benötigen wir noch Regeln zur Ableitung von *expr*. Abbildung 8.1 zeigt die vollständige Grammatik.

boolExpr	→	*expr* "==" *expr*
	|	*expr* "<" *expr*
expr	→	*product exprRest*
exprRest	→	"+" *product exprRest*
	|	"-" *product exprRest*
	|	ε
product	→	*factor productRest*
productRest	→	"*" *factor productRest*
	|	"/" *factor productRest*
	|	ε
factor	→	"(" *expr* ")"
	|	NUMBER
	|	IDENTIFIER

Abbildung 8.1: Grammatik für arithmetische Ausdrücke.

Es ist nicht möglich einen LL(1)-Parser zu implementieren, der Boole'sche Ausdrücke nach diesen beiden Regeln erkennt, denn beide Alternativen fangen mit einer *expr* an. Wir können die Grammatik aber durch *Links-Faktorisierung* (engl. *left factoring*) so umschreiben, das ein Token als Look-Ahead ausreicht, indem wir den Teil aus den beiden Grammatik-Regeln ausklammern, der am Anfang der beiden Regeln identisch ist. In dem obigen Beispiel führen wir dann für den verbleibenden Rest das neue Nicht-Terminal *boolExprRest* ein und erhalten die Regeln

boolExpr	→	*expr boolExprRest*
boolExprRest	→	"==" *expr*
	|	"<" *expr.*

Mit diesen Regeln reicht nun ein Token als Look-Ahead aus, denn die beiden Alternativen für *boolExprRest* unterscheiden sich in dem ersten Token des Rumpfs der Regel.

Um den allgemeinen Fall der Links-Faktorisierung diskutieren zu können, nehmen wir an, dass A ein Nicht-Terminal ist, das durch insgesamt $m + n$ Regeln definiert wird, wobei der Rumpf der ersten m Regeln immer mit α anfängt, wobei α ein String aus Terminalen und Nicht-Terminalen ist. Die Regeln haben also die folgende Form:

$$
\begin{array}{lcl}
A & \rightarrow & \alpha\ \beta_1 \\
& | & \alpha\ \beta_2 \\
& \vdots & \vdots \\
& | & \alpha\ \beta_m \\
& | & \gamma_1 \\
& \vdots & \vdots \\
& | & \gamma_n
\end{array}
$$

Bei dieser Darstellung sei vorausgesetzt, dass die Strings $\beta_1, \cdots, \beta_m$ keinen gemeinsamen Präfix haben und dass α auch kein Präfix einer der Strings γ_i ist. Bei der Links-Faktorisierung dieser Regeln klammern wir einerseits den gemeinsamen Präfix α aus und führen andererseits eine neue syntaktische Variable B ein, die den auf α folgenden Rest bezeichnet. Wir erhalten dann die folgenden Regeln:

$$
\begin{array}{lcl}
A & \rightarrow & \alpha\ B \\
& | & \gamma_1 \\
& \vdots & \vdots \\
& | & \gamma_n
\end{array}
\qquad\qquad
\begin{array}{lcl}
B & \rightarrow & \beta_1 \\
& | & \beta_2 \\
& \vdots & \vdots \\
& | & \beta_m
\end{array}
$$

Um alle gemeinsamen Präfixe auszuklammern muss dieses Verfahren unter Umständen mehrfach durchgeführt werden. Die nächste Aufgabe gibt dafür ein Beispiel.

Aufgabe 15: Geben Sie eine Links-Faktorisierung für die folgenden Grammatik-Regeln an.

$$
\begin{array}{lcl}
A & \rightarrow & \text{“}\texttt{a}\text{”}\ \ \text{“}\texttt{b}\text{”}\ U\ \text{“}\texttt{d}\text{”} \\
& | & \text{“}\texttt{a}\text{”}\ V\ \text{“}\texttt{b}\text{”}\ \ \text{“}\texttt{d}\text{”} \\
& | & \text{“}\texttt{a}\text{”}\ \ \text{“}\texttt{b}\text{”}\ W \\
& | & \text{“}\texttt{x}\text{”}\ U \\
& | & \text{“}\texttt{x}\text{”}\ V
\end{array}
$$

Lösung: Zunächst eliminieren wir das gemeinsame Präfix “a” und führen dazu die neue syntaktische Variable B ein. Wir erhalten:

$$
\begin{array}{lcl}
A & \rightarrow & \text{“}\texttt{a}\text{”}\ B \\
& | & \text{“}\texttt{x}\text{”}\ U \\
& | & \text{“}\texttt{x}\text{”}\ V \\
\\
B & \rightarrow & \text{“}\texttt{b}\text{”}\ U\ \text{“}\texttt{d}\text{”} \\
& | & V\ \text{“}\texttt{b}\text{”}\ \ \text{“}\texttt{d}\text{”} \\
& | & \text{“}\texttt{b}\text{”}\ W
\end{array}
$$

Nun eliminieren wir das Präfix “x” aus beiden letzten Regeln für A. Wir führen dazu die neue syntaktische Variable C ein. Dann erhalten wir:

$$
\begin{array}{lcl}
A & \rightarrow & \text{“}\texttt{a}\text{”}\ B \\
& | & \text{“}\texttt{x}\text{”}\ C \\
\\
C & \rightarrow & U \\
& | & V
\end{array}
$$

$$
\begin{array}{lcl}
B & \rightarrow & \text{“b”}\ U\ \text{“d”} \\
& | & V\ \text{“b”}\ \text{“d”} \\
& | & \text{“b”}\ W
\end{array}
$$

Als letztes eliminieren wir das Präfix “b”, das in zwei der Regeln für die syntaktische Variable B auftritt. Wir nennen die neu eingeführte Variable d und erhalten:

$$
\begin{array}{lcl}
A & \rightarrow & \text{“a”}\ B \\
& | & \text{“x”}\ C \\
\\
C & \rightarrow & U \\
& | & V \\
\\
B & \rightarrow & \text{“b”}\ D \\
& | & V\ \text{“b”}\ \text{“d”} \\
\\
D & \rightarrow & U\ \text{“d”} \\
& | & W
\end{array}
$$

□

Bemerkung: Bei dem Parser-Generator ANTLR läßt sich der Lookahead als Option einstellen. Abbildung 8.2 zeigt ein Beispiel für eine Grammatik, für die sich mit einem Lookahead von $k = 1$ kein Parser erzeugen läßt. Wir haben hier in den Zeilen 3 – 5 den Lookahead als 1 spezifiziert. Ändern wir dort den Lookahead auf 2, so kann ANTLR einen Parser erzeugen.

```
1  grammar Left;
2
3  options {
4      k = 1;
5  }
6
7  a : 'x' b
8    | 'x' c
9    ;
10
11 b : 'y' ;
12
13 c : 'z' ;
14
15 WS: (' '|'\n'|'\r'|'\t')+ { skip(); };
```

Abbildung 8.2: Eine Grammatik mit zu geringem Lookahead.

Da ANTLR die Verwendung von EBNF-Grammatiken unterstützt, können wir die Grammatik aber auch von Hand faktorisieren, indem wir den Teil der Regel, der beiden Grammatik-Regeln gemeinsam ist, ausklammern. Abbildung 8.3 zeigt die resultierende Grammatik, die sich nun von ANTLR verarbeiten läßt.

8.1.1 *First* und *Follow*

Nicht für jede links-faktorisierte Grammatik läßt sich ein LL(1)-Parser bauen. Betrachten wir die folgenden Regeln:

$$A \rightarrow B \mid C$$

```
1  grammar Factored;
2
3  options {
4      k = 1;
5  }
6
7  a : 'x' ( b | c)
8    ;
9
10 b : 'y' ;
11
12 c : 'z' ;
13
14 WS: (' '|'\n'|'\r'|'\t')+ { skip(); };
```

Abbildung 8.3: Die Grammatik aus Abbildung 8.2 in faktorisierter Form.

$$B \rightarrow \text{“a”}\ U$$
$$C \rightarrow \text{“a”}\ V$$

Will der Parser ein A parsen und ist das nächste Token ein “a”, so ist nicht klar, ob der Parser als nächstes die Regel

$$A \rightarrow B \quad \text{oder} \quad A \rightarrow C$$

verwenden soll. Für die obige Grammatik läßt sich daher kein LL(1)-Parser implementieren. Zur Entscheidung, ob sich für eine gegebene Grammatik ein LL(1)-Parser implementieren läßt, benötigen wir die Funktionen *First*() und *Follow*(), die wir gleich definieren werden. Um diese Funktionen implementieren zu können, definieren wir vorher den Begriff einer ε-erzeugenden syntaktischen Variablen.

Definition 21 (ε-erzeugend) Es sei $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik und A sei eine syntaktische Variable, also $A \in V$. Dann heißt die Variable A *ε-erzeugend* genau dann, wenn

$$A \Rightarrow^* \varepsilon$$

gilt, also dann, wenn sich aus der Variablen A das leere Wort ableiten läßt. Wir schreiben *nullable*(A) wenn die Variable A als ε-erzeugend nachgewiesen ist. □

Beispiele:

1. Bei der in Abbildung 8.1 auf Seite 103 gezeigten Grammatik sind offenbar die Variablen *exprRest* und *productRest* ε-erzeugend.

2. Wir betrachten nun ein weniger offensichtliches Beispiel. Die Grammatik G enthalte die folgenden Regeln:

   $$S \rightarrow A\ B\ C$$
   $$A \rightarrow \text{“a”}\ B \mid A\ \text{“b”} \mid B\ C$$
   $$B \rightarrow \text{“a”}\ B \mid A\ \text{“b”} \mid C\ C$$
   $$C \rightarrow A\ B\ C \mid \varepsilon$$

   Zunächst ist offenbar die Variable C ε-erzeugend. Dann sehen wir, dass aufgrund der Regel $B \rightarrow C\ C$ auch B ε-erzeugend ist und daraus folgt dann wegen der Regel $A \rightarrow B\ C$, dass

auch A ε-erzeugend ist. Schließlich ist nun aufgrund der Regel

$$S \to A\ B\ C$$

selbst S ε-erzeugend.

Definition 22 (*First*()) Es sei $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik und $A \in V$. Dann definieren wir *First*(A) als die Menge aller der Token t, mit denen ein von A abgeleitetes Wort beginnen kann:

$$\mathit{First}(A) := \{t \in T \mid \exists \gamma \in (V \cup T)^* : A \Rightarrow^* t\gamma\}.$$

Die Definition der Funktion *First*() kann wie folgt auf Strings aus $(V \cup T)^*$ erweitert werden:

1. $\mathit{First}(\varepsilon) = \{\}$.
2. $\mathit{First}(t\beta) = \{t\}$ falls $t \in T$.
3. $\mathit{First}(A\beta) = \begin{cases} \mathit{First}(A) \cup \mathit{First}(\beta) & \text{falls } A \Rightarrow^* \varepsilon; \\ \mathit{First}(A) & \text{sonst.} \end{cases}$ □

Beispiel: Bei der in Abbildung 8.1 gezeigten Grammatik für arithmetische Ausdrücke können wir die Funktion *First*() für die einzelnen Variablen am besten so berechnen, dass wir mit den Variablen beginnen, die in der Hierarchie ganz unten stehen.

1. Zunächst folgt aus den Regeln

   $$\mathit{factor} \to \text{“(”}\ \mathit{expr}\ \text{“)”} \mid \mathit{NUMBER} \mid \mathit{IDENTIFIER},$$

   dass jeder von *Factor* abgeleitete String entweder mit einer öffnenden Klammer, einer Zahl oder einem Bezeichner beginnt:

   $$\mathit{First}(\mathit{factor}) = \{\ \text{“(”}\ , \mathit{NUMBER}, \mathit{IDENTIFIER}\ \}.$$

2. Analog folgt aus den Regeln

   $$\mathit{productRest} \to \text{“*”}\ \mathit{factor}\ \ \mathit{productRest} \mid \text{“/”}\ \mathit{factor}\ \ \mathit{productRest} \mid \varepsilon,$$

   dass ein *productRest* entweder mit dem Zeichen “*” oder “/” beginnt:

   $$\mathit{First}(\mathit{productRest}) = \{\ \text{“*”}\ ,\ \text{“/”}\ \}$$

3. Die Regel für die Variable *product* lautet

   $$\mathit{product} \to \mathit{factor}\ \ \mathit{productRest}.$$

   Da die Variable *factor* nicht ε erzeugend ist, sehen wir, dass die Menge *First*(*product*) mit der Menge *First*(*factor*) übereinstimmt:

   $$\mathit{First}(\mathit{product}) = \{\ \text{“(”}\ , \mathit{NUMBER}, \mathit{IDENTIFIER}\ \}.$$

4. Aus den Regeln

   $$\mathit{exprRest} \to \text{“+”}\ \mathit{product}\ \ \mathit{exprRest} \mid \text{“-”}\ \mathit{product}\ \ \mathit{exprRest} \mid \varepsilon$$

   können wir *First*(*exprRest*) wie folgt berechnen:

   $$\mathit{First}(\mathit{exprRest}) = \{\ \text{“+”}\ ,\ \text{“-”}\ \}.$$

5. Weiter folgt aus der Regel

   $$\mathit{expr} \to \mathit{product}\ \ \mathit{exprRest}$$

   und der Tatsache, dass *product* nicht ε-erzeugend ist, dass die *First*(*expr*) mit der Mengen *First*(*product*) übereinstimmt:

   $$\mathit{First}(\mathit{expr}) = \{\ \text{“(”}\ , \mathit{NUMBER}, \mathit{IDENTIFIER}\ \}.$$

6. Schließlich folgt aus den beiden Regeln

   $$\mathit{boolExpr} \rightarrow \mathit{expr}\ \text{“==”}\ \mathit{expr} \quad \text{und} \quad \mathit{boolExpr} \rightarrow \mathit{expr}\ \text{“<”}\ \mathit{expr}$$

   sowie der Tatsache, das *expr* nicht ε-erzeugend ist, dass *First*(*boolExpr*) mit *First*(*expr*) identisch ist:

   $$\mathit{First}(\mathit{boolExpr}) = \{\ \text{“(”}\ , \mathit{NUMBER}, \mathit{IDENTIFIER}\ \}.$$ □

Definition 23 (*Follow*()) Es sei $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik und $A \in V$. Bei der Berechnung von *Follow*() wird die Grammatik zunächst abgeändert, indem wir das Symbol “$” als neues Symbol zu der Menge T der Terminale hinzufügen. Zu den Variablen wird das neue Symbol $\widehat{S}$ hinzugefügt, dass auch gleichzeitig das neue Start-Symbol der Grammatik ist. Zu der Menge R der Regeln fügen wir die folgende Regel neu hinzu:

$$\widehat{S} \rightarrow S\ \text{“\$”}\ .$$

Das Terminal “$” steht hierbei für das Ende der Eingabe (EOF, *end of file*). Weiter definieren wir

$$\widehat{T} := T \cup \{\ \text{“\$”}\ \}.$$

Dann definieren wir *Follow*(A) als die Menge aller der Token t, die in einer Ableitung auf A folgen können:

$$\mathit{Follow}(A) := \{t \in \widehat{T} \mid \exists \beta, \gamma \in (V \cup \widehat{T})^* : \widehat{S} \Rightarrow^* \beta A t \gamma\}.$$

Wenn sich aus dem Start-Symbol $\widehat{S}$ also irgendwie ein String $\beta A t \gamma$ ableiten läßt, bei dem das Token t auf die Variable A folgt, dann ist t ein Element der Menge *Follow*(A). □

Beispiel: Wir untersuchen wieder die in Abbildung 8.1 gezeigte Grammatik für arithmetische Ausdrücke.

1. Aufgrund der neu hinzugefügten Regel

   $$\widehat{S} \rightarrow \mathit{boolExpr}\ \text{“\$”}$$

   muss die Menge *Follow*(*boolExpr*) das Zeichen “$” enthalten. Da die syntaktische Variable *boolExpr* sonst nirgends in der Grammatik vorkommt, haben wir

   $$\mathit{Follow}(\mathit{boolExpr}) = \{\ \text{“\$”}\ \}.$$

2. Die beiden Grammatik-Regeln

   $$\mathit{boolExpr} \rightarrow \mathit{expr}\ \text{“==”}\ \mathit{expr} \quad \text{und} \quad \mathit{boolExpr} \rightarrow \mathit{expr}\ \text{“<”}\ \mathit{expr}$$

   zeigen uns zunächst, dass die Menge *Follow*(*expr*) die Zeichen “==” und “<” enthält. Da *expr* auch am Ende dieser Regeln steht, folgt weiter, dass alle Elemente aus *Follow*(*boolExpr*) auch auf *expr* folgen können, wir haben also auch

   $$\text{“\$”}\ \in \mathit{Follow}(\mathit{expr}).$$

   Aufgrund der Regel

   $$\mathit{factor} \rightarrow\ \text{“(”}\ \mathit{expr}\ \text{“)”}$$

   muss die Menge *Follow*(*expr*) außerdem das Zeichen “)” enthalten. Also haben wir insgesamt

   $$\mathit{Follow}(\mathit{expr}) = \{\ \text{“==”}\ ,\ \text{“<”}\ ,\ \text{“\$”}\ ,\ \text{“)”}\ \}.$$

3. Aufgrund der Regel

   $$\mathit{expr} \rightarrow \mathit{product}\ \ \mathit{exprRest}$$

   wissen wir, dass alle Terminale, die auf ein *expr* folgen können, auch auf ein *exprRest* folgen können, womit wir schon mal wissen, dass *Follow*(*exprRest*) die Token “==”, “<” und “)”

enthält. Da *exprRest* sonst nur am Ende der Regeln vorkommt, die *exprRest* definieren, sind das auch schon alle Token, die auf *exprRest* folgen können und wir haben

$$\mathit{Follow}(\mathit{exprRest}) = \{\ \text{“}\texttt{==}\text{”}\ ,\ \text{“}\texttt{<}\text{”}\ ,\ \text{“}\texttt{\$}\text{”}\ ,\ \text{“}\texttt{)}\text{”}\ \}.$$

4. Die Regeln

   $$\mathit{exprRest} \rightarrow\ \text{“}\texttt{+}\text{”}\ \mathit{product}\ \ \mathit{exprRest} \mid\ \text{“}\texttt{-}\text{”}\ \mathit{product}\ \ \mathit{exprRest}$$

   zeigen, dass auf ein *product* alle Elemente aus *First*(*exprRest*) folgen können, aber das ist noch nicht alles: Da die Variable *exprRest* ε-erzeugend ist, können zusätzlich auf *product* auch alle Token folgen, die auf *exprRest* folgen. Damit haben wir insgesamt

   $$\mathit{Follow}(\mathit{product}) = \{\ \text{“}\texttt{+}\text{”}\ ,\ \text{“}\texttt{-}\text{”}\ ,\ \text{“}\texttt{==}\text{”}\ ,\ \text{“}\texttt{<}\text{”}\ ,\ \text{“}\texttt{\$}\text{”}\ ,\ \text{“}\texttt{)}\text{”}\ \}.$$

5. Die Regel

   $$\mathit{product} \rightarrow \mathit{factor}\ \ \mathit{productRest}$$

   zeigt, dass alle Terminale, die auf ein *product* folgen können, auch auf ein *productRest* folgen können. Da *productRest* sonst nur am Ende der Regeln vorkommt, die *productRest* definieren, sind das auch schon alle Token, die auf *productRest* folgen können und wir haben insgesamt

   $$\mathit{Follow}(\mathit{productRest}) = \{\ \text{“}\texttt{+}\text{”}\ ,\ \text{“}\texttt{-}\text{”}\ ,\ \text{“}\texttt{==}\text{”}\ ,\ \text{“}\texttt{<}\text{”}\ ,\ \text{“}\texttt{\$}\text{”}\ ,\ \text{“}\texttt{)}\text{”}\ \}.$$

6. Die Regeln

   $$\mathit{productRest} \rightarrow\ \text{“}\texttt{*}\text{”}\ \mathit{factor}\ \ \mathit{productRest} \mid\ \text{“}\texttt{/}\text{”}\ \mathit{factor}\ \ \mathit{productRest}$$

   zeigen, dass auf ein *factor* alle Elemente aus *First*(*productRest*) folgen können, aber das ist noch nicht alles: Da die Variable *productRest* ε-erzeugend ist, können zusätzlich auf *factor* auch alle Token folgen, die auf *productRest* folgen. Damit haben wir insgesamt

   $$\mathit{Follow}(\mathit{factor}) = \{\ \text{“}\texttt{*}\text{”}\ ,\ \text{“}\texttt{/}\text{”}\ ,\ \text{“}\texttt{+}\text{”}\ ,\ \text{“}\texttt{-}\text{”}\ ,\ \text{“}\texttt{==}\text{”}\ ,\ \text{“}\texttt{<}\text{”}\ ,\ \text{“}\texttt{\$}\text{”}\ ,\ \text{“}\texttt{)}\text{”}\ \}.$$ □

Das letzte Beispiel zeigt, dass die Berechnung des Prädikats *nullable*() und die Berechnung der Mengen *First*(A) und *Follow*(A) für eine syntaktische Variable A eng miteinander verbunden sind. Es sei

$$A \rightarrow Y_1 Y_2 \cdots Y_k$$

eine Grammatik-Regel. Dann bestehen zwischen dem Prädikat `nullable`() und den beiden Funktionen *First*() und *Follow*() die folgenden Beziehungen:

1. $\forall t \in T : \neg\, \mathit{nullable}(t)$.
2. $k = 0 \Rightarrow \mathit{nullable}(A) = \texttt{true}$.
3. $\big(\forall i \in \{1, \cdots, k\} : \mathit{nullable}(Y_i)\big) \Rightarrow \mathit{nullable}(A)$.

   Setzen wir hier $k = 0$ so sehen wir, dass 2. ein Spezialfall von 3. ist.
4. $\mathit{First}(Y_1) \subseteq \mathit{First}(A)$.
5. $\big(\forall j \in \{1, \cdots, i-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_i) \subseteq \mathit{First}(A)$.
6. $\mathit{Follow}(A) \subseteq \mathit{Follow}(Y_k)$.
7. $\big(\forall j \in \{i+1, \cdots, k\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{Follow}(A) \subseteq \mathit{Follow}(Y_i)$.

   Setzen wir hier $i = k$ so sehen wir, dass 6. ein Spezialfall von 7. ist.
8. $\forall i \in \{1, \cdots, k-1\} : \mathit{First}(Y_{i+1}) \subseteq \mathit{Follow}(Y_i)$.

9. $\big(\forall j \in \{i+1, \cdots, l-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_l) \subseteq \mathit{Follow}(Y_i)$.

   Setzen wir hier $l = i + 1$ so sehen wir, dass 8. ein Spezialfall von 9. ist.

Mit Hilfe dieser Beziehungen können *nullable*(), *First*() und *Follow*() iterativ berechnet werden:

1. Zunächst werden die Funktionen *First*(A) und *Follow*(A) für jede syntaktische Variable A mit der leeren Menge initialisiert. Das Prädikat *nullable*(A) wird für jede syntaktische Variable auf `false` gesetzt.

2. Anschließend werden die oben angegebenen Regeln so lange angewendet, wie sich durch die Anwendung Änderungen ergeben.

8.1.2 Implementierung

Abbildung 8.4 zeigt die Struktur der Klasse `Grammar`, in der wir die Berechnung der ε-erzeugenden Variablen sowie der Funktionen *First*() und *Follow*() implementieren wollen. Zunächst enthält diese Klasse in der Member-Variable `mSimpleRules` die Menge aller Grammatik-Regeln. Die Member-Variable `mNullable` enthält später die ε-erzeugenden Variablen, die Member-Variable `mFirst` speichert die Funktion *First*() und die Member-Variable `mFollow` speichert die Funktion *Follow*(). Darüber hinaus enthält die Klasse die Methoden *computeNullable*(), *computeFirst*() und *computeFollow*().

1. *computeNullable*() berechnet die Menge der ε-erzeugenden Variablen.

2. *computeFirst*() berechnet die *First*-Mengen der Variablen.

3. *computeFollow*() berechnet die *Follow*-Mengen der Variablen.

Die Abbildungen 8.5, 8.6 und 8.7 auf den folgenden Seiten zeigen die Implementierung dieser Methoden.

```
1  public class Grammar {
2      private Set<SimpleRule>  mSimpleRules;
3
4      private Set<Variable>               mNullable;
5      private Map<Variable, Set<MyToken>> mFirst;
6      private Map<Variable, Set<MyToken>> mFollow;
7      ...
8
9      public Grammar(List<Rule> rules) { ... }
10
11     void computeParseTable() { ... }
12     void computeNullable() { ... }
13     void computeFirst() { ... }
14     void computeFollow() { ... }
15     ...
16 }
```

Abbildung 8.4:

Die Berechnung der ε-erzeugenden Variablen

Wir diskutieren zunächst die Berechnung aller ε-erzeugenden Variablen in der in Abbildung 8.5 gezeigten Methode `computeNullable()`. Die Grundidee ist folgende: Falls es für die Variable A

```
1  void computeNullable() {
2      boolean change = true;
3      while (change) {
4          change = false;
5          for (SimpleRule r : mSimpleRules) {
6              List<Item> items = r.getBody().getItemList();
7              boolean allNullable = true;
8              for (Item i: items) {
9                  if (i instanceof MyToken) {
10                     allNullable = false;
11                     break;
12                 }
13                 Variable v = (Variable) i;
14                 if (!mNullable.contains(v)) {
15                     allNullable = false;
16                     break;
17                 }
18             }
19             Variable head = r.getHead();
20             if (allNullable && !mNullable.contains(head)) {
21                 mNullable.add(head);
22                 change = true;
23             }
24         }
25     }
26 }
```

Abbildung 8.5: Die Methode *nullable*().

eine Regel der Form

$$A \to Y_1 \cdots Y_k$$

gibt, so dass alle Y_i bereits ε-erzeugend sind, dann ist auch A ε-erzeugend. Beachten Sie, dass dies den Fall $k = 0$ mit einschließt: Wenn $k = 0$ ist und die Grammatik-Regel $A \to Y_1 \cdots Y_k$ folglich die Form $A \to \varepsilon$ hat, dann ist die Menge $\{1, \cdots, k\}$ leer und damit sind alle Y_i für $i \in \{1, \cdots, k\}$ ε-erzeugend.

In der Methode `computeNullable()` haben wir eine äußere `while`-Schleife, die sich in der Abbildung 8.5 von Zeile 3 bis Zeile 25 erstreckt und die solange durchlaufen wird, wie wir neue ε-erzeugende Variablen finden. Diese Schleife wird über die Variable `change` gesteuert: Zu Beginn der äußeren Schleife setzen wir in Zeile 4 `change` auf `false`. Immer, wenn wir eine neue syntaktische Variable als ε-erzeugend erkannt haben, setzen wir `change` auf `true`, so dass dann die Schleife ein weiteres mal durchlaufen wird. Innerhalb der `while`-Schleife beginnt in Zeile 5 eine `for`-Schleife, in der wir über alle Regeln der Grammatik laufen. Die Variable `items` wird in Zeile 6 mit der rechten Seite der Grammatik-Regel initialisiert, wenn die Grammatik-Regel `r` in Zeile 5 die Form

$$A \to Y_1 \cdots Y_k$$

hat, dann entspricht `items` der Liste $[Y_1, \cdots, Y_k]$. In der inneren `for`-Schleife, die sich von Zeile 8 bis Zeile 18 erstreckt, überprüfen wir nun, ob alle Y_i bereits als ε-erzeugend erkannt worden sind. Falls dies der Fall ist, und die Variable A bisher noch nicht als ε-erzeugend erkannt ist, dann ist A ebenfalls ε-erzeugend und wir fügen die syntaktische Variable A zu der Menge `mNullable` hinzu, in der wir alle die syntaktischen Variablen, die ε-erzeugend sind, aufsammeln. Außerdem setzen wir dann in Zeile 22 `change` auf den Wert `true`, denn es könnte ja nun sein, dass wir durch die

Erkenntnis, dass A ε-erzeugend ist, für weitere Variablen darauf schließen können, dass diese ε-erzeugend sind. Am Ende der Schleife enthält die Menge `mNullable` alle syntaktischen Variablen, die ε-erzeugend sind.

Die Berechnung der Funktion *First*()

```
1  void computeFirst() {
2      boolean change = true;
3      while (change) {
4          change = false;
5          for (SimpleRule r : mSimpleRules) {
6              Variable   head  = r.getHead();
7              List<Item> items = r.getBody().getItemList();
8              Set<MyToken> firstSet = mFirst.get(head);
9              for (Item i: items) {
10                 if (i instanceof MyToken) {
11                     if (!firstSet.contains(i)) {
12                         MyToken t = (MyToken) i;
13                         firstSet.add(t);
14                         change = true;
15                     }
16                     break;
17                 }
18                 Variable v = (Variable) i;
19                 Set<MyToken> ts = mFirst.get(v);
20                 if (!firstSet.containsAll(ts)) {
21                     firstSet.addAll(ts);
22                     change = true;
23                 }
24                 if (!mNullable.contains(v)) {
25                     break;
26                 }
27             }
28         }
29     }
30 }
```

Abbildung 8.6: Die Methode *computeFirst*().

Abbildung 8.6 zeigt die Berechnung der Menge *First*(). Die `for`-Schleife, die in Zeile 5 beginnt, iteriert über alle Regeln der Grammatik. Ist `r` eine solche Regel und hat die Form

$$A \to Y_1 \cdots Y_k,$$

so wird `head` in Zeile 6 mit A initialisiert und in Zeile 7 wird `items` die Liste $[Y_1, \cdots, Y_k]$ zugewiesen. Die `for`-Schleife in Zeile 9 iteriert nun über diese Liste. Es gibt folgende Fälle:

1. Y_i ist ein Terminal und Y_1, $\cdots$, Y_{i-1} sind ε-erzeugend.

   In diesem Fall gehört Y_i sicher zur Menge *First*(A) hinzu. Gleichzeitig können dann die Variablen Y_{i+1}, $\cdots$, Y_k die Berechnung von *First*(A) nicht mehr beinflussen, so dass die innere Schleife durch ein `break` abgebrochen werden kann.

   Dieser Fall wird in den Zeilen 10 bis 17 behandelt.

2. Y_i ist eine Variable und $Y_1, \cdots, Y_{i-1}$ sind ε-erzeugend.

   In diesem Fall enthält $\mathit{First}(A)$ alle Terminale aus $\mathit{First}(Y_i)$.

3. Y_i ist eine Variable, aber Y_i ist nicht ε-erzeugend.

   In diesem Fall brechen wir die Iteration über die Y_i ab.

Jedesmal, wenn wir bei dem obigen Verfahren einer Menge $\mathit{First}(A)$ ein neues Terminal hinzufügen, dann kann es sein, dass wir auch in weiteren *First*-Mengen, die $\mathit{First}(A)$ enthalten, dieses Terminal hinzufügen müssen. Daher ist die `for`-Schleife, in der wir über alle Regeln iterieren, noch in eine `while`-Schleife eingebettet, die solange läuft, wie wir neue Terminale für eine *First*-Menge finden.

Die Berechnung der Funktion *Follow*()

```
1   void computeFollow() {
2       boolean change = true;
3       while (change) {
4           change = false;
5           for (SimpleRule r : mSimpleRules) {
6               Variable   head = r.getHead();
7               List<Item> list = r.getBody().getItemList();
8               Set<MyToken> followSetHead = mFollow.get(head);
9               for (int i = 0; i < list.size(); ++i) {
10                  Item item = list.get(i);
11                  if (item instanceof MyToken) {
12                      continue;
13                  }
14                  Variable v = (Variable) item;
15                  if (allNullable(list, i + 1, list.size() - 1)) {
16                      Set<MyToken> followSet = mFollow.get(v);
17                      if (!followSet.containsAll(followSetHead)) {
18                          followSet.addAll(followSetHead);
19                          change = true;
20                      }
21                  }
22              }
```

Abbildung 8.7: Die Methode *computeFollow*(), erster Teil.

Auch bei der Implementierung der Methode *Follow*() iterieren wir über alle Regeln der Grammatik. Der Implementierung liegen die folgenden Beziehungen zu Grunde, die für eine Grammatik-Regel der Form $A \rightarrow Y_1 \cdots Y_k$ gelten.

1. $\big(\forall j \in \{i+1, \cdots, k\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{Follow}(A) \subseteq \mathit{Follow}(Y_i)$.

   Diese Beziehung wird in den Zeilen 9 bis 22 umgesetzt. Beachten Sie, dass die Beziehung

   $$\mathit{Follow}(A) \subseteq \mathit{Follow}(Y_k).$$

   ein Spezialfall dieses Falls ist, der von der Implementierung ebenfalls mit abgedeckt wird.

2. $\big(\forall j \in \{i+1, \cdots, l-1\} : \mathit{nullable}(Y_j) \wedge Y_l \in T\big) \Rightarrow Y_l \in \mathit{Follow}(Y_i)$

   Diese Beziehung wird in den Zeilen 33 bis 39 umgesetzt.

3. $\big(\forall j \in \{i+1, \cdots, l-1\} : \mathit{nullable}(Y_j) \wedge Y_l \in V\big) \Rightarrow \mathit{First}(Y_l) \subseteq \mathit{Follow}(Y_i)$

   Diese Beziehung wird in den Zeilen 39 bis 45 umgesetzt.

```
23                for (int i = 0; i < list.size() - 1; ++i) {
24                    Item itemI = list.get(i);
25                    if (itemI instanceof MyToken) {
26                        continue;
27                    }
28                    Variable     vi       = (Variable) itemI;
29                    Set<MyToken> followVi = mFollow.get(vi);
30                    for (int l = i + 1; l < list.size(); ++l) {
31                        if (allNullable(list, i + 1, l - 1)) {
32                            Item itemL = list.get(l);
33                            if (itemL instanceof MyToken) {
34                                MyToken tokenL = (MyToken) itemL;
35                                if (!followVi.contains(tokenL)) {
36                                    followVi.add(tokenL);
37                                    change = true;
38                                }
39                            } else {
40                                Variable vl = (Variable) itemL;
41                                Set<MyToken> firstVl = mFirst.get(vl);
42                                if (!followVi.containsAll(firstVl)) {
43                                    followVi.addAll(firstVl);
44                                    change = true;
45                                }
46                            }
47                        }
48                    }
49                }
50            }
51        }
52    }
```

Abbildung 8.8: Die Methode *computeFollow*(), zweiter Teil.

8.1.3 LL(1)-Grammatiken

Wir können nun die Frage beantworten, für welche Grammatiken ein Top-Down-Parser erzeugt werden kann, der immer mit einem Token Lookahead auskommt.

Definition 24 (LL(1)-Grammatik) Eine Grammatik G ist eine *LL(1)-Grammatik* genau dann, wenn für jede syntaktische Variable A, für die es in der Grammatik G zwei verschiedene Regeln

$$A \to \alpha \quad \text{und} \quad A \to \beta$$

gibt, die folgenden Bedingungen erfüllt sind:

1. $\neg(\alpha \Rightarrow^* \varepsilon \wedge \beta \Rightarrow^* \varepsilon)$.

   Die Rümpfe zweier verschiedener Regeln der selben Variablen dürfen nicht beide das leere Wort ableiten.

2. $\mathit{First}(\alpha) \cap \mathit{First}(\beta) = \{\}$.

   Die Ableitungen der Rümpfe zweier verschiedener Regeln der selben Variablen dürfen nicht mit dem selben Token beginnen.

3. $(\beta \Rightarrow^* \varepsilon) \to \mathit{First}(\alpha) \cap \mathit{Follow}(A) = \{\}$.

Wenn β den leeren String ableitet, dann müssen die Mengen $\mathit{First}(\alpha)$ und $\mathit{Follow}(A)$ disjunkt sein. □

Wir diskutieren nun die Idee, die hinter der obigen Definition steht.

1. Falls das leere Wort sowohl über die Regel

   $$A \to \alpha \quad \text{als auch über} \quad A \to \beta$$

   ableitbar wäre, so wissen wir nicht, welche Regel wir anwenden sollen, wenn wir ein A ableiten sollen und das nächste Eingabe-Token ein Element der Menge $\mathit{Follow}(A)$ ist.

2. Um ein A zu parsen und zwischen den beiden Regeln für A unterscheiden zu können, verwenden wir das folgende Rezept:

   Parsen wir ein A und ist das Lookahead-Token ein Element der Menge $\mathit{First}(\alpha)$, so verwenden wir die Regel

   $$A \to \alpha.$$

   Analog verwenden wir die Regel

   $$A \to \beta,$$

   wenn das Lookahead-Token ein Element der Menge $\mathit{First}(\beta)$ ist.

   Dieses Rezept funktioniert natürlich nur, wenn die Mengen $\mathit{First}(\alpha)$ und $\mathit{First}(\beta)$ disjunkt sind.

3. Das obige Rezept um ein A zu parsen muss in dem Fall, dass β das leere Wort ableitet, wie folgt erweitert werden.

   Gilt $\beta \Rightarrow^* \varepsilon$ und ist das Lookahead-Token ein Element der Menge $\mathit{Follow}(A)$, so verwenden wir die Regel

   $$A \to \beta.$$

   Damit diese Regel nicht im Widerspruch zu den unter Punkt 2. genannten Regeln steht, benötigen wir die Bedingungen

   $$(\beta \Rightarrow^* \varepsilon) \to \mathit{First}(\alpha) \cap \mathit{Follow}(A) = \{\}.$$

Insgesamt versuchen wir also dann mit einer Regel $A \to \alpha$ zu reduzieren, wenn eine der beiden folgenden Bedingungen erfüllt sind. In diesen Bedingungen bezeichnet *lat* das Lookahead-Token.

1. $\mathit{lat} \in \mathit{First}(\alpha)$ oder
2. $\alpha \Rightarrow^* \varepsilon$ und $\mathit{lat} \in \mathit{Follow}(\alpha)$.

Nach diesen Vorbereitungen können wir nun zu einer LL(1)-Grammatik die *Parse-Tabelle* berechnen. Für eine Grammatik $G = \langle V, T, R, S \rangle$ ist die Parse-Tabelle

$$\mathit{parseTable} : V \times T \to 2^R,$$

eine Funktion, so dass der Aufruf $\mathit{parseTable}(A, t)$ einer syntaktischen Variable A und einem Token t die Menge aller Regeln der Form

$$A \to \alpha$$

zuordnet, die bei einer Ableitung von A in Frage kommen, wenn das nächste zu lesenden Token den Wert t hat. Diese Funktion genügt den folgenden beiden Bedingungen:

1. Ist $A \to \alpha$ eine Regel der Grammatik und ist t ein Token aus der Menge $\mathit{First}(\alpha)$, dann ist diese Regel ein Element der Menge $\mathit{parseTable}(A, t)$:

   $$(A \to \alpha) \in R \;\wedge\; t \in \mathit{First}(\alpha) \;\Rightarrow\; (A \to \alpha) \in \mathit{parseTable}(A, t).$$

```
1   void computeParseTable() {
2       for (Variable v : mVariableSet) {
3           Map<MyToken, Set<SimpleRule>> tokenMap =
4               new TreeMap<MyToken, Set<SimpleRule>>();
5           for (MyToken t : mTokenSet) {
6               Set<SimpleRule> ruleSet = new TreeSet<SimpleRule>();
7               tokenMap.put(t, ruleSet);
8           }
9           mParseTable.put(v, tokenMap);
10      }
11      for (SimpleRule r : mSimpleRules) {
12          Variable   head = r.getHead();
13          List<Item> body = r.getBody().getItemList();
14          Set<MyToken> first = computeFirst(body);
15          enter(first, head, r);
16          if (allNullable(body, 0, body.size() - 1)) {
17              Set<MyToken> follow = mFollow.get(head);
18              enter(follow, head, r);
19          }
20      }
21  }
22
23  void enter(Set<MyToken> tokens, Variable v, SimpleRule r) {
24      for (MyToken t : tokens) {
25          Set<SimpleRule> rules = mParseTable.get(v).get(t);
26          rules.add(r);
27      }
28  }
```

Abbildung 8.9: Berechnung der Parse-Tabelle.

2. Ist $A \to \alpha$ eine Regel der Grammatik, wobei α ε-erzeugend ist, und ist t ein Token aus der Menge $\mathit{Follow}(A)$, dann ist diese Regel ein Element der Menge $\mathit{parseTable}(A, t)$:

$$(A \to \alpha) \in R \ \wedge \ \alpha \Rightarrow^* \varepsilon \ \wedge \ t \in \mathit{Follow}(A) \ \Rightarrow \ (A \to \alpha) \in \mathit{parseTable}(A, t).$$

Abbildung 8.9 zeigt eine Berechnung der Parse-Tabelle.

1. Die `for`-Schleife in den Zeilen 2 bis 10 initialisiert $\mathit{parseTable}(A, t)$ für alle syntaktischen Variablen A und alle Token t mit der leeren Menge.

2. In Zeile 15 fügen wir mit Hilfe der Methode *enter*() die Regel $A \to \alpha$ für alle Token t aus der Menge $\mathit{First}(\alpha)$ in $\mathit{parseTable}(A, t)$ ein.

3. Falls $\alpha \Rightarrow *\varepsilon$ gilt, so fügen wir in den Zeilen 20 bis 22 für jedes Token t aus $\mathit{Follow}(A)$ die Regel $A \to \alpha$ in die Menge $\mathit{parseTable}(A, t)$ ein.

Eine Grammatik ist genau dann eine $LL(1)$-Grammatik, wenn die Mengen $\mathit{parseTable}(A, t)$ für jede syntaktische Variable A und jedes Token t maximal eine Regel enthält:

$$G \text{ ist LL(1)} \quad \text{g.d.w.} \quad \forall A \in V : \forall t \in T : \mathit{card}\big(\mathit{parseTable}(A, t)\big) \leq 1.$$

Falls die Menge $\mathit{parseTable}(A, t)$ leer ist, so heißt dies einfach, dass wir beim Parsen von A nicht auf das Token t stoßen können. Parsen wir also ein A und sehen als erstes Zeichen das Token t, so muss ein Syntax-Fehler vorliegen.

8.1.4 LL(k)-Grammatiken

Viele interessante Grammatiken sind keine $LL(1)$-Grammatiken. Abbildung 8.10 zeigt ein Beispiel. Bei dieser Grammatiken werden für arithmetische Ausdrücke auch Funktionsaufrufe der Form

$$f(a_1, \cdots, a_n)$$

zugelassen. Dabei ist f ein Funktionszeichen, das syntaktisch nicht von einem Identfier zu unterscheiden ist. Dadurch gibt es zwischen den beiden Regeln

$$\textit{factor} \rightarrow \textsc{Identifier} \quad \text{und} \quad \textit{factor} \rightarrow \textsc{Identifier}\ \text{“(”}\ \textit{argList}\ \text{“)”}$$

einen Konflikt: Soll ein *factor* geparst werden und ist das nächste zu lesende Zeichen ein Identifier, so ist nicht klar, welche der beiden Regeln angewendet werden sollen. Die Lösung des Problems besteht in diesem Fall darin, zusätzlich das zweite Zeichen mit heran zu ziehen: Handelt es sich um das Zeichen “(” , so ist offenbar die Regel

$$\textit{factor} \rightarrow \textsc{Identifier}\ \text{“(”}\ \textit{argList}\ \text{“)”}$$

heranzuziehen, andernfalls muss die Regel

$$\textit{factor} \rightarrow \textsc{Identifier}$$

verwendet werden.

expr	→	*product exprRest*
exprRest	→	“+” *product exprRest*
	|	“-” *product exprRest*
	|	ε
product	→	*factor productRest*
productRest	→	“*” *factor productRest*
	|	“/” *factor productRest*
	|	ε
factor	→	“(” *expr* “)”
	|	*Number*
	|	Identifier
	|	Identifier “(” *argList* “)”
argList	→	*expr argsRest*
	|	ε
argsRest	→	“,” *expr argsRest*
	|	ε

Abbildung 8.10: Arithmetische Ausdrücke mit Funktions-Aufrufen.

Im allgemeinen Fall kann das Verfahren so erweitert werden, dass k Token bei der Entscheidung, welche Regel zu verwenden ist, als Lookahead herangezogen werden. Bei Antlr ist es beispielsweise möglich, den Lookahead k zu spezifizieren. Die Praxis zeigt, dass die so erzeugten Parser sehr wohl zu Parsern, die von Bottom-Up-Parser-Generatoren wie *Bison* oder Cup erzeugt werden, konkurrenzfähig sind. Wir skizzieren die Grundzüge dieser Theorie. Als erstes verallgemeinern wir

die Definition der Funktion *First*().

Definition 25 (*First*(k, α)) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$\mathit{First} : \mathbb{N} \times (V \cup T)^* \to 2^{T^*},$$

so dass $\mathit{First}(k, \alpha)$ für eine natürliche Zahl k und einen String α, der aus Terminalen und syntaktischen Variablen besteht, die Menge der Token-Strings berechnet, die höchstens die Länge k haben und die Präfix eines von α abgeleiteten Strings sind. Formal lautet die Definition:

$$\mathit{First}(k, \alpha) := \big\{ x \in T^* \mid \exists y \in T^* : \alpha \Rightarrow^* xy \wedge |x| = k \big\} \cup \big\{ x \in T^* \mid \alpha \Rightarrow^* x \wedge |x| < k \big\}. \quad \square$$

Beispiel: Streichen wir zur Vereinfachung in der in Abbildung 8.1 gezeigte Grammatik für arithmetische Ausdrücke die Regel

$$\mathit{factor} \to \textsc{Number}$$

sowie die Regeln für *boolExpr* und kürzen wir weiter das Token IDENTIFIER als ID ab, so erhalten wir beispielsweise:

1. $\mathit{First}(2, \mathit{expr}) = \big\{$ ID, "(" ID, "((", ID"+", ID"-", ID"*", ID"/" $\big\}$,
2. $\mathit{First}(2, \mathit{exprRest}) = \big\{ \varepsilon$, "+"ID, "+(", "-"ID, "-(", $\big\}$,
3. $\mathit{First}(2, \mathit{product}) = \big\{$ ID, "(" ID, "((", ID"*", ID"/" $\big\}$,
4. $\mathit{First}(2, \mathit{productRest}) = \big\{ \varepsilon$, "*"ID, "*(", "/"ID, "/(" $\big\}$,
5. $\mathit{First}(2, \mathit{factor}) = \big\{$ ID, "(" ID, "((" $\big\}$.

Definition 26 (*Follow*(k, A)) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$\mathit{Follow} : \mathbb{N} \times V \to 2^{T^*},$$

so dass $\mathit{Follow}(k, A)$ für eine natürliche Zahl k und eine syntaktische Variable V die Menge der Token-Strings berechnet, die höchstens die Länge k haben und in einer Ableitung, die vom Start-Sysmbol S ausgeht, auf A folgen können. Formal lautet die Definition:

$$\mathit{Follow}(k, A) := \big\{ x \in T^* \mid \exists \alpha, \gamma \in (T \cup V)^* : S \Rightarrow^* \alpha A \gamma \wedge x \in \mathit{First}(k, \gamma) \big\}. \quad \square$$

Beispiel: Setzen wir das letzte Beispiel sinngemäß fort, so erhalten wir:

1. $\mathit{Follow}(2, \mathit{expr}) = \big\{ \varepsilon$, ")", "))" $\big\}$,
2. $\mathit{Follow}(2, \mathit{exprRest}) = \big\{ \varepsilon$, ")", "))" $\big\}$,
3. $\mathit{Follow}(2, \mathit{product}) = \big\{ \varepsilon$, "+"ID, "+(", "-"ID, "-(", ")", "))" $\big\}$,
4. $\mathit{Follow}(2, \mathit{productRest}) = \big\{ \varepsilon$, "+"ID, "+(", "-"ID, "-(", ")", "))" $\big\}$,
5. $\mathit{Follow}(2, \mathit{factor}) = \big\{ \varepsilon$, ")", "))", "+"ID, "+(", "-"ID, "-(", "*"ID, "*(", "/Id", "/(" $\big\}$.

Definition 27 (Starke $LL(k)$-Grammatik) Eine kontextfreie Grammatik $G = \langle V, T, R, S \rangle$ ist eine *starke $LL(k)$-Grammatik* genau dann, wenn für je zwei verschiedene Grammatik-Regeln

$$A \to \beta \quad \text{und} \quad A \to \gamma$$

aus der Menge R die Bedingung

$$\forall \sigma, \tau \in \mathit{Follow}(k, A) : \mathit{First}(k, \beta\sigma) \cap \mathit{First}(k, \gamma\tau) = \{\}$$

erfüllt ist. $\square$

Erklärung: Um die obige Definition zu verstehen, nehmen wir an, wir wollten ein A parsen. Wenn wir einen $LL(k)$-Parser bauen wollen, dürfen wir die nächsten k Symbole der Eingabe lesen und müssen entscheiden, welche der Regeln von A in Frage kommen. Diese k Symbole können das Resultat der Ableitung von β oder γ sein. Wenn die von β oder γ abgeleiteten Strings kürzer als k sind, so kann es sich aber auch schon um Token handeln, die in einer Ableitung auf A folgen, die also Elemente der Menge $\mathit{Follow}(k, A)$ sind. Für eine Regel

$$A \to \beta$$

und einen String $\sigma \in \mathit{Follow}(k, A)$ enthält die Menge $\mathit{First}(k, \beta\sigma)$ alle die Strings der Länge $\leq k$, die in einer Ableitung von A, welche die Regel $A \to \beta$ benutzt, folgen können. Sind diese Mengen für verschiedene Regeln disjunkt, so läßt sich an Hand der k folgenden Token entscheiden, welche der Regeln angewendet werden muss.

Bemerkung: In der theoretischen Informatik gibt es neben dem Begriff der *starken* $LL(k)$-Grammatik auch noch den Begriff der (einfachen) $LL(k)$-Grammatik. Bei einer solchen $LL(k)$-Grammatik dürfen bei der Auswahl der Regel nicht nur die nächsten k Eingabe-Token berücksichtigt werden, sondern zusätzlich kann der Parser alle bisher gelesenen Token mit zu Rate ziehen. Dadurch kann in bestimten Fällen zu gegebener Variable und gegebenem Lookahead auch dann noch eine Regel ausgewählt werden, wenn das Kriterium der starken $LL(k)$-Grammatik nicht erfüllt ist. Da dieser Begriff einerseits wesentlich komplexer ist als der Begriff der starken $LL(k)$-Grammatik, andererseits das Werkzeug ANTLR auch den Begriff der starken LL(k)-Grammatik implementiert, verzichten wir auf eine formale Darstellung des allgemeineren Begriffs. Die dem allgemeineren Begriff zu Grunde liegende Theorie ist sehr ausführlich in [AU72] dargestellt.

Berechnung von *First*() und *Follow*()

In diesem Abschnitt zeigen wir, wir die Funktionen $\mathit{First}(k, \alpha)$ und $\mathit{Follow}(k, A)$ berechnet werden können. Dazu benötigen wir verschiedene Hilfsfunktionen, die wir vorab definieren.

1. Die Funktion $\mathit{prefix}(k, w)$ berechnet für eine natürliche Zahl k und einen String w den Präfix von w mit der Länge k. Ist die Länge von w kleiner oder gleich k, so wird w zurück gegeben:

   $$\mathit{prefix}(k, w) = \begin{cases} w[1\!:\!k] & \text{falls } k < |w|; \\ w & \text{sonst.} \end{cases}$$

   Hier bezeichnet die Notation $w[1\!:\!k]$ den Teilstring von w, der aus den ersten k Buchstaben von w besteht.

2. Der Operator $+_k$ verkettet zwei Strings und bildet anschließend das Päfix der Länge k:

   $$v +_k w = \mathit{prefix}(k, vw).$$

   Hier bezeichnet vw die Verkettung der Strings v und w.

3. Die Definition des Operators $+_k$ wird auf Mengen von Strings verallgemeinert:

   $$M +_k N := \{v +_k w \mid v \in M \wedge w \in N\}.$$

Beispiel: Wir haben

$$\begin{aligned} &\{\varepsilon,\ \text{“}\texttt{a}\text{”}\ ,\ \text{“}\texttt{ab}\text{”}\ ,\ \text{“}\texttt{abc}\text{”}\ \} +_2 \{\varepsilon,\ \text{“}\texttt{x}\text{”}\ ,\ \text{“}\texttt{yx}\text{”}\ \} \\ = &\{\varepsilon,\ \text{“}\texttt{a}\text{”}\ ,\ \text{“}\texttt{ab}\text{”}\ ,\ \text{“}\texttt{x}\text{”}\ ,\ \text{“}\texttt{yx}\text{”}\ ,\ \text{“}\texttt{ax}\text{”}\ ,\ \text{“}\texttt{ay}\text{”}\ \}. \end{aligned}$$ □

Die Berechnung von $\mathit{First}(k, \alpha)$ für $\alpha \in (V \cup T)^*$ wird auf die Berechnung von $\mathit{First}(k, \alpha)$ für Variablen und Terminale zurück geführt, denn es gilt

$$\mathit{First}(k, X_1 X_2 \cdots X_n) = \mathit{First}(k, X_1) +_k \mathit{First}(k, X_2) +_k \cdots +_k \mathit{First}(k, X_n).$$

Für ein Terminal $t \in T$ gilt offenbar

$$\mathit{First}(k, t) = t.$$

Die Berechnung der Mengen $\mathit{First}(k, A)$ für eine syntaktische Variable $A \in V$ erfolgt iterativ über folgenden Algorithmus:

1. Zunächst werden alle Mengen $\mathit{First}(k, A)$ mit der leeren Menge initialisiert:

   $$\mathit{First}(k, A) := \{\}.$$

2. Anschließend wird für jede Grammatik-Regel der Form

   $$A \to \alpha$$

   die Menge $\mathit{First}(k, A)$ wie folgt erweitert:

   $$\mathit{First}(k, A) := \mathit{First}(k, A) \cup \mathit{First}(k, \alpha).$$

3. Der zweite Schritt wird in einer Schleife solange durchgeführt, bis sich keine der Mengen $\mathit{First}(k, A)$ mehr durch die Hinzunahme von $\mathit{First}(k, \alpha)$ ändert.

Sind die Mengen $\mathit{First}(k, A)$ berechnet, so können wir anschließend die Mengen $\mathit{Follow}(k, A)$ für alle syntaktischen Variablen berechnen. Auch die Berechnung der Mengen $\mathit{Follow}(k, A)$ ist iterativ. Sie erfolgt nach dem folgenden Schema:

1. Zunächst werden alle Mengen $\mathit{Follow}(k, A)$ mit der leeren Menge initialisiert:

   $$\mathit{Follow}(k, A) = \{\}.$$

2. Für jede Grammatik-Regel der Form

   $$A \to Y_1 Y_2 \cdots Y_l,$$

   für die Y_l eine syntaktische Variable ist, erweitern wir die Menge $\mathit{Follow}(k, Y_l)$ wie folgt:

   $$\mathit{Follow}(k, Y_l) := \mathit{Follow}(k, Y_l) \cup \mathit{Follow}(k, A),$$

   denn alles, was auf ein A folgen kann, kann auch auf ein Y_l folgen.

3. Für jede Grammatik-Regel der Form

   $$A \to Y_1 Y_2 \cdots Y_i Y_{i+1} \cdots Y_l,$$

   und jeden Index $i \in \{1, \cdots, l-1\}$, für den Y_i eine syntaktische Variable ist, erweitern wir die Menge $\mathit{Follow}(k, Y_i)$ wie folgt:

   $$\mathit{Follow}(k, Y_i) := \mathit{Follow}(k, Y_i) \cup \big(\mathit{First}(k, Y_{i+1} \cdots Y_l) +_k \mathit{Follow}(k, A)\big).$$

   Der Grund, warum wir hier noch die Menge $\mathit{Follow}(k, A)$ anhängen ist der, dass die Strings aus der Menge $\mathit{First}(k, Y_{i+1} \cdots Y_l)$ eventuell kürzer als k sind. In diesem Fall müssen noch die Präfixe von $\mathit{Follow}(k, A)$ angehängt werden.

4. Der zweite und der dritte Schritt werden in einer Schleife solange durchgeführt, bis sich keine der Mengen $\mathit{Follow}(k, A)$ mehr ändert.

Kapitel 9

Behandlung von Nicht-LL(k)-Sprachen in ANTLR

In diesem Kapitel werden wir die zusätzlichen Möglichkeiten diskutieren, die ANTLR zur Verfügung stellt, um auch für solche Sprachen, die keine LL(k)-Sprachen sind, Parser erzeugen zu können. Im Einzelnen diskutieren wir die folgenden Besonderheiten von ANTLR.

1. Unterstützung von LL($*$)-Sprachen,
2. semantische Prädikate und
3. syntaktische Prädikate nebst Backtracking.

9.1 Unterstützung von LL($*$)-Sprachen

Wir werden in diesem Abschnitt zunächst den Begriff der $LL(*)$-Sprachen [Par07] definieren. Hierbei handelt es sich um eine Sprachenklasse, die wesentlich mächtiger ist, als die Klasse der $LL(k)$-Sprachen. Der Parser-Generator ANTLR kann für alle $LL(*)$-Sprachen einen Parser erzeugen und ist damit erheblich leistungsfähiger als der Top-Down-Parser-Generator *JavaCC*[1], der nur für LL(k)-Sprachen einen Parser erzeugen kann. Um den Begriff der LL($*$)-Sprachen zu motivieren, zeigen wir an Hand praktischer Beispiele die Unzulänglichkeit der Klasse der $LL(k)$-Sprachen, anschließend geben wir eine formale Definition der Klasse der $LL(*)$-Sprachen.

9.1.1 Motivation

Wir zeigen an Hand zweier Beispiele, dass der Begriff der $LL(k)$-Sprachen für die Beschreibung von Programmiersprachen in der Praxis nicht ausreichend ist. Die Beispiele sind der ANTLR-Referenz [Par07] entnommen.

Deklarationen versus Definitionen

Betrachten wir zunächst eine Grammatik, die sowohl die Deklaration als auch die Definition einer Funktion in der Sprache `C` beschreibt. Die Deklaration einer Funktion mit zwei Parametern könnte in `C` beispielsweise wie folgt aussehen:

```
int hugo(int a, int b);
```

Eine Definition dieser Funktion könnte die folgende Form haben:

```
int hugo(int a, int b) { return 17; }
```

[1] https://javacc.dev.java.net/

```
1  grammar DeclOrDef;
2
3  options {              // Suppress these lines
4      k = 8;             // to let ANTLR create
5  }                      // an LL(*)-parser.
6
7  decl_or_def
8      : type ID '(' args ')' ';'
9      | type ID '(' args ')' '{' body '}'
10     ;
11
12 type: 'void'
13     | 'int'
14     ;
15
16 args: arg (',' arg)* ;
17
18 arg : 'int' ID ;
19
20 body: 'return' INT;
21
22 ID : ('A'..'Z'|'a'..'z')+;
23 INT: ('1'..'9') ('0'..'9')*;
```

Abbildung 9.1: Eine Grammatik für Deklarationen und Definitionen

Abbildung 9.1 zeigt eine (vereinfachte) Grammatik in ANTLR-SYNTAX, die sowohl Deklarationen als auch Definitionen von Funktionen zuläßt. Das Start-Symbol dieser Grammatik ist `decl_or_def`. Beide Regeln zur Ableitung der syntaktischen Variable `decl_or_def` beginnen mit der Token-Folge

type ID '('.

Daher ist klar, dass wir mindestens 3 Token Look-Ahead-Token benötigen, um zwischen einer Deklaration und einer Definition unterscheiden zu können. In der oben gezeigten ANTLR-Spezifikation der Grammatik ist für den Look-Ahead k der Wert 8 spezifiziert worden. Lassen wir von ANTLR einen Parser für diese Grammatik erzeugen, so erhalten wir die folgende Fehlermeldung:

```
ANTLR Parser Generator  Version 3.0 (May 17, 2007)  1989-2007
warning(200): method.g:8:7: Decision can match input such as

  "'void'..'int' ID '(' 'int' ID ',' 'int' ID"

using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

An dieser Stelle können wir erkennen, dass die durch die angegebene Grammatik spezifizierte Sprache keine $LL(k)$-Sprache ist, denn ob eine Deklaration oder eine Definition einer Funktion vorliegt, können wir erst dann erkennen, wenn wir das Zeichen hinter der schließenden Klammer ')' sehen:

1. Falls dieses Zeichen ein Semikolon ist, liegt eine Deklaration vor.
2. Falls das Zeichen eine öffnende geschweifte Klammer ist, haben wir die Definition einer Funktion.

Da nun zwischen den runden Klammern beliebig viele Argumente kommen können, können wir keinen festen Wert k für den Look-Ahead angeben, der ausreicht, um die Entscheidung zwischen den beiden Alternativen der Grammatik zu treffen. Kommentieren wir die Zeilen 3 bis 5 aus, so kann ANTLR einen Parser erzeugen, denn ohne Spezifikation von k versucht ANTLR automatisch, einen $LL(*)$-Parser an Stelle eines $LL(k)$-Parsers zu konstruieren, was in dem vorliegenden Fall auch gelingt.

```
1  grammar DeclOrDefFactor;
2
3  options {
4       k = 1;
5  }
6
7  decl_or_def
8      : type ID '(' args ')' (';' | '{' body '}')
9      ;
10
11 type: 'void'
12     | 'int'
13     ;
14
15 args: arg (',' arg)* ;
16
17 arg : 'int' ID ;
18
19 body: 'return' INT;
20
21 ID : ('A'..'Z'|'a'..'z')+;
22 INT: ('1'..'9') ('0'..'9')*;
```

Abbildung 9.2: Eine faktorisierte Grammatik für Deklarationen und Definitionen.

Wir können das oben dargestellte Problem auch durch Faktorisierung lösen. Dadurch, dass *Antlr* die Möglichkeit bietet, auf der rechten Seite einer Regel Klammern zu setzen, bleibt die resultierende Grammatik sehr übersichtlich. Abbildung 9.2 zeigt, wie sich die Unterscheidung von Deklarationen und Definitionen durch Faktorisierung der Grammatik-Regeln lösen läßt.

Klassen versus Schnittstellen

Abbildung 9.3 zeigt eine Grammatik, die wahlweise Klassen- oder Schnittstellen-Definitionen für die Sprache *Java* beschreibt. Die angegebene Grammatik läßt auch Klassen-Definitionen zu, bei denen ein Modifikator mehrfach wiederholt wird. Zwar wäre es prinzipiell möglich die Grammatik so umzuschreiben, dass nur zulässige Folgen von Modifikator akzeptiert würden, aber die resultierende Grammatik wäre erheblich komplexer als die angegebene Grammatik. Es ist wesentlich einfacher, die Folge der Modifikatoren später durch eine geeignete Methode überprüfen zu lassen.

Die in Abbildung 9.3 angegebene Grammatik ist keine $LL(k)$ Grammatik, denn ob eine Klasse oder eine Schnittstelle spezifiziert werden soll wird erst klar, wenn das Schlüsselwort `class` oder `interface` gefunden wird. Die Grammatik spezifiziert, dass diesem Schlüsselwort beliebig viele Modifikatoren vorangehen können, so dass ein endlicher Look-Ahead nicht ausreicht um zwischen den beiden Varianten zu unterscheiden. Auch hier könnten wir das Problem durch Faktorisierung lösen, in Abbildung 9.4 ist dies gezeigt. Die einfachste Möglichkeit besteht allerdings wieder darin, *Antlr* einen $LL(*)$-Parser erzeugen zu lassen. Diese erreichen wir, in dem wir

```
1  grammar ClassOrInterface;
2
3  options {
4      k = 5;
5  }
6
7  class_or_interface
8      : modifier* 'class'     ID '{' '...' '}'
9      | modifier* 'interface' ID '{' '...' '}'
10     ;
11
12 modifier
13     : 'public'
14     | 'package'
15     | 'protected'
16     | 'private'
17     | 'abstract'
18     | 'final'
19     ;
20
21 ID : ('A'..'Z'|'a'..'z')+;
```

Abbildung 9.3: Grammatik für Klassen und Schnittstellen.

```
options { k = 5; }
```

durch

```
options { k = *; }
```

ersetzen, oder aber die Spezifikation der Look-Ahead-Tiefe k ganz weglassen.

9.1.2 Die Theorie der LL(∗)-Sprachen

Die Grundidee, um für Sprachen, für die ein endlicher Look-Ahead-Tiefe nicht ausreicht, einen Parser zu entwickeln besteht darin, einen deterministischen endlichen Automaten zu erzeugen, dessen Aufgabe es ist, von den zur Verfügung stehenden Regeln eine Regel auszuwählen. Wir werden das Verfahren zunächst an einem Beispiel motivieren, anschließend folgt die formale Definition.

Motivation: Regel-Auswahl durch einen endlichen Automaten

In dem Fall der Unterscheidung zwischen Funktionen und Deklarationen kann die Entscheidung, welche der beiden Regeln verwendet werden soll, durch einen endlichen Automaten getroffen werden. Abbildung 9.5 zeigt die Definition eines deterministischen endlichen Automaten, der zwischen Funktionen und Deklarationen unterscheidet. Die Interpretation der Zustände ist wie folgt:

1. Der Zustand q_0 ist der Start-Zustand.
2. Im Zustand q_1 hat der Automat ein '`void`' oder '`int`' gelesen.
3. Im Zustand q_2 hat der Automat zusätzlich den Namen der Funktion gelesen.
4. Im Zustand q_3 hat der Automat zusätzlich eine öffnende runde Klammer "`(`" gelesen.
5. Im Zustand q_4 hat der Automat zusätzlich "`int`" gelesen und wartet als nächstes auf eine Variable ID. Wird diese gelesen, so wechselt der Automat in den Zustand q_5.

```
1  grammar ClassOrInterfaceFactor;
2
3  options {
4      k = 1;
5  }
6
7  class_or_interface_factor
8      : modifier* ('class' | 'interface') ID '{' '...' '}'
9      ;
10
11 modifier
12     : 'public'
13     | 'package'
14     | 'protected'
15     | 'private'
16     | 'abstract'
17     | 'final'
18     ;
19
20 ID : ('A'..'Z'|'a'..'z')+;
```

Abbildung 9.4: Eine faktorisierte Grammatik für Klassen und Schnittstellen.

6. Im Zustand q_5 gibt es die Möglichkeit, entweder weitere Argumente zu lesen, oder aber eine schließende Klammer. Der erste Fall tritt ein, wenn ein Komma "`,`" gelesen wird, der zweite Fall liegt vor, wenn das nächste Zeichen eine schließende Klammer "`)`" ist.

7. Im Zustand q_6 fällt nun die Entscheidung: Wird ein Semikolon "`;`" gelesen, so geht der Automat in den Zustand q_7, der in der Abbildung mit *decl* beschriftet ist, wird statt dessen eine öffnenden geschweiften Klammer "{" gelesen, so geht der Automat in den Zustand q_8, der in der Abbildung mit *def* beschriftet ist.

8. Damit sind die Zustände q_7 und q_8 die akzeptierenden Zustände des Automaten. Im Zustand q_7 wird anschließend der Parser die Regel

   ```
   decl_or_def -> type ID '(' args ')' ';'
   ```

   verwenden, im Zustand q_8 wird stattdessen die Regel

   ```
   decl_or_def -> type ID '(' args ')' '{' body '}'
   ```

   benutzt.

Zu beachten ist, dass die Token, die der endliche Automat liest, alle in den Eingabestrom zurück gelegt werden, damit der eigentliche Parser diese Token später lesen kann.

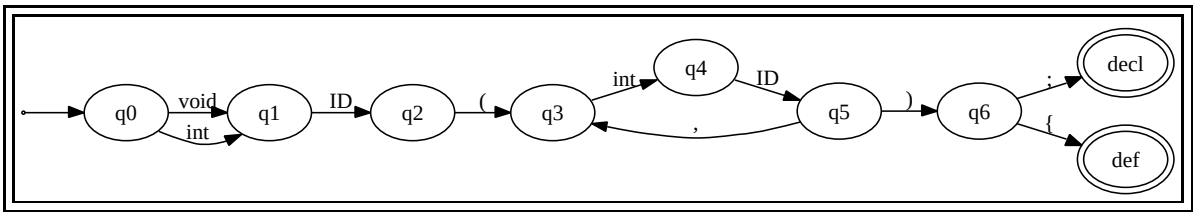


Abbildung 9.5: Ein deterministischer endlicher Automat zur Unterscheidung zwischen Deklarationen und Definitionen.

Implementierung der LL(∗)-Sprachen

Wir diskutieren jetzt, wie ANTLR bei der Erstellung des endlichen Automaten für die Auswahl einer Regel vorgeht. Ist

$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$$

eine Grammatik-Regel, bei der eine Look-Ahead-Tiefe von 1 nicht ausreicht um zu bestimmen, welche der k Regeln $A \to \alpha_i$ mit $i = 1 \cdots, k$ zum Parsen ausgewählt werden soll, so versucht ANTLR die rechten Seiten der Regeln solange zu expandieren, bis lediglich ein regulärer Ausdruck übrig bleibt. Gelingt dies für alle der k verschiedenen Regeln, so können die regulären Ausdrücke anschließend in nicht-deterministische endlichen Automaten umgewandelt werden. Wir indizieren dabei die einzelnen Zustände mit dem Index i der Regel, die diesen Zustand erzeugt hat. Der nächste Schritt besteht nun darin, über die bereits im Kapitel über endliche Automaten diskutierte Teilmengen-Konstruktion aus den verschiedenen nicht-deterministischen Automaten einen deterministischen endlichen Automaten zu erzeugen. Die einzelnen Zustände dieses deterministischen Automaten sind Mengen von Zuständen der verschiedenen nicht-deterministischen Automaten, die den einzelnen Regeln zugeordnet waren. Wir bezeichnen eine Menge solcher Zustände als *homogen*, wenn alle Zustände von der selben Regel stammen. Entscheidend ist nun, ob alle Zustands-Mengen, die einen akzeptierenden Zustand enthalten, homogen sind. Ist dies der Fall, so kann der endliche Automat entscheiden, welche Regel anzuwenden ist, denn der Automat muss nur solange laufen, bis eine Zustands-Menge erreicht ist, die einen akzeptierenden Zustand enthält. Aufgrund der Homogenität der Zustandsmenge können wir dieser Menge dann nämlich eindeutig eine Regel zuordnen. Wir illustrieren die verschiedenen Schritte jetzt am Beispiel der in Abbildung 9.1 gezeigten Grammatik zur Unterscheidung von Deklarationen und Definitionen.

1. Zunächst ersetzen wir auf der rechten Seite der Grammatik-Regel für die syntaktische Variable *decl_or_def* die syntaktischen Variablen *type*, *args* und *body* durch die entsprechenden rechten Seiten der Grammatik-Regeln, die diese Variablen definieren. Wir erhalten dann die folgende Regel:

   ```
   decl_or_def
       : ('void' | 'int') ID '(' arg (',' arg)* ')' ';'
       | ('void' | 'int') ID '(' arg (',' arg)* ')' '{' 'return' INT '}'
       ;
   ```

2. Nun tritt noch die syntaktische Variable *arg* auf der rechten Seite der beiden Regeln für *decl_or_def* auf. Auch diese ersetzen wir und erhalten:

   ```
   decl_or_def
      : ('void' | 'int') ID '(' 'int' ID (',' 'int' ID)* ')' ';'
      | ('void' | 'int') ID '(' 'int' ID (',' 'int' ID)* ')' '{' 'return' INT '}'
      ;
   ```

3. Nun enthalten die Ausdrücke, die auf der rechten Seite der beiden Grammatik-Regeln stehen, keine syntaktischen Variablen mehr und können daher als reguläre Ausdrücke aufgefasst werden. Wir konstruieren für beide Ausdrücke einen nicht-deterministischen endlichen Automaten. Die resultierenden endlichen Automaten sind in den beiden Abbildungen 9.6 und 9.7 gezeigt.

   Aus diesen beiden Automaten bauen wir nun einen neuen Automaten, der zunächst nicht-deterministisch ist. Dies erreichen wir, indem wir einen neuen Start-Zustand s_0 einführen, von dem aus es ε-Übergänge zu den Start-Zuständen q_0 und p_0 gibt. Für diesen, zunächst

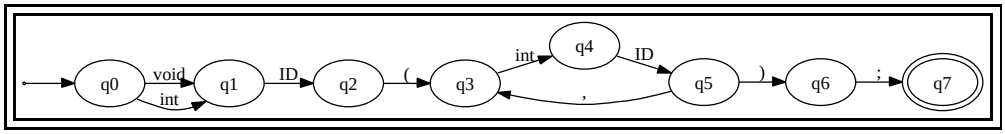


Abbildung 9.6: Ein endlicher Automat zur Erkennung von Deklarationen.

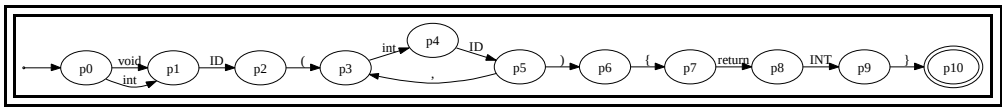


Abbildung 9.7: Ein endlicher Automat zur Erkennung von Definitionen.

nicht-deterministischen Automaten führen wir die Teilmengen-Konstruktion durch und erhalten dann wieder einen deterministischen endlichen Automaten. Der resultierende Automat ist in Abbildung 9.8 gezeigt. Wir sehen hier, dass die Zustands-Mengen, die akzeptierende Zustände enthalten, homogen sind. Die erste solche Menge ist die Menge $\{q_7\}$, die zweite Menge ist $\{p_{10}\}$. Damit ist klar, dass die in Abbildung 9.1 definierte Sprache eine LL(∗)-Sprache ist.

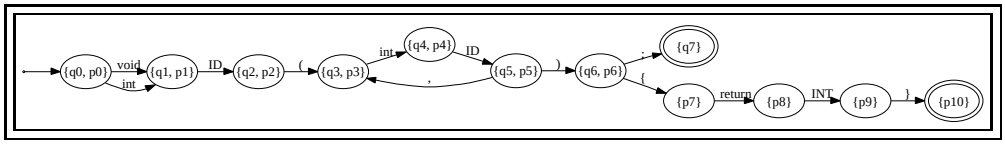


Abbildung 9.8: Der zusammengesetzte endliche Automat.

Probleme bei rekursiven Grammatik-Regeln

Es ist offensichtlich, dass das oben beschriebene Verfahren nicht funktionieren kann, wenn die Grammatik-Regeln rekursiv sind, denn dann ist es nicht möglich, die rechten Seiten der Regeln so zu expandieren, bis nur noch reguläre Ausdrücke übrig bleiben. Ändern wir die EBNF-Grammatik aus Abbildung 9.1 wie in Abbildung 9.9 gezeigt so um, dass die Regeln für die syntaktische Variable *args* rekursiv werden, so liefert ANTLR die folgende Fehlermeldung:

```
error(211): decl_or_def_recursive.g:8:5: [fatal] rule decl_or_def_recursive has
non-LL(*) decision due to recursive rule invocations reachable from alts 1,2.
Resolve by left-factoring or using syntactic predicates or using backtrack=true
option.
```

Momentan kann ANTLR dann eine Grammatik nicht als LL(∗)-Grammatik erkennen, falls die beteiligten Grammatik-Regeln Rekursion enthalten. In diesem Fall untersucht ANTLR zunächst, ob es sich um eine LL(1)-Grammatik handelt und liefert andernfals eine Fehlermeldung. Als Benutzer können Sie diese Schwäche in der Praxis meist dadurch umgehen, dass Sie statt der Rekursion die entsprechenden EBNF-Postfix-Operatoren "`+`" und "`*`" zur Spezifikation von Listen verwenden. In den Fällen, wo dies nicht möglich ist, können Sie wahlweise syntaktischen Look-Ahead spezifizieren, oder aber das später noch diskutierte Backtracking aktivieren.

9.2 Semantische Prädikate

Viele Programmiersprachen lassen sich nicht durch kontextfreie Grammatiken beschreiben, weil oft der Kontext, in dem ein Zeichen auftritt, mit darüber entscheidet, wie das Zeichen zu interpretieren ist. Ein typisches Beispiel liefert die Programmiersprache `C`: Ob eine Zeichenreihe als Variable oder als Typ-Bezeichner interpretiert wird, ist kontext-abhängig. Der Grund liegt darin, dass `C` die

```
1  grammar DeclOrDefRecursive;
2
3  decl_or_def
4      : type ID '(' args ')' ';'
5      | type ID '(' args ')' '{' body '}'
6      ;
7
8  type: 'void'
9      | 'int'
10     ;
11
12 args: arg
13     | arg ',' args
14     ;
15
16 arg : 'int' ID ;
17
18 body: 'return' INT;
```

Abbildung 9.9: Rekursive Grammatik für Deklarationen und Definitionen

Möglichkeit gestattet, mit Hilfe des Schlüsselworts `typedef` neue Typ-Bezeichner zu definieren. Ein Beispiel sehen Sie in Abbildung 9.10: Der in Zeile 1 und Zeile 5 verwendete Typ-Bezeichner `uint` hat in Zeile 1 die Funktion einer Variablen, während er in Zeile 5 als Typ-Bezeichner verwendet wird. Die Tasache, dass `unit` in Zeile 5 als Typ-Bezeichner zu interpretieren ist, ist rein syntaktisch nicht ersichtlich sondern folgt aus der Typ-Definition in Zeile 3.

Bemerkung: Die Syntax von `C`-Typdeklarationen ist aufgrund der Tatsache, dass Sie in `C` auch Zeiger auf Funktionen deklarieren können, sehr komplex. Insbesondere muss zugelassen werden, dass die zu deklarierenden Ausdrücke geklammert werden können. Dadurch ist die resultierende Grammatik nicht mehr kontextfrei.

```
1  int (uint);
2
3  typedef unsigned int uint;
4
5  uint(x);
```

Abbildung 9.10: Typ-Bezeichner und Variablen sind syntaktisch nicht unterscheidbar!

Wir betrachten nun eine Grammatik, mit der sich Deklarationen, Typ-Definitionen und einfache Befehle in C beschreiben lassen. Wir haben die Grammatik so weit wie möglich vereinfacht um das Problem der Mehrdeutigkeit klarer herausheben zu können. Abbildung 9.11 zeigt die Grammatik. Die erste Grammatik-Regel besagt, dass ein `C`-Programm aus Deklarationen (*decl*), Typ-Definitionen (*typeDef*) oder Befehlen (*stmnt*) besteht. Als Befehle haben wir in der vereinfachten Grammatik nur Funktions-Aufrufe zugelassen. Die Grammatik ist mehrdeutig, denn der String

```
uint(x);
```

kann einerseits aufgrund der Regel

decl → *type* “(” *identifier* “)”

als Deklaration interpretiert werden, andererseits zeigt die Grammatik-Regel

stmnt → *identifier* “(” *identifier* “)” ,

dass auch eine Interpretation als Funktions-Aufruf möglich ist.

```
1   grammar cGrammar;
2
3   declOrType
4       : decl
5       | typeDef
6       | stmnt
7       ;
8
9   decl: type identifier
10      | type '(' identifier ')'
11      ;
12
13  typeDef: TYPEDEF type identifier;
14
15  stmnt: identifier '(' identifier ')' ;
16
17  type: UNSIGNED
18      | INT
19      | UNSIGNED INT
20      | typeid
21      ;
22
23  typeid: ID;
24
25  identifier: ID;
26
27  TYPEDEF : 'typedef';
28  INT     : 'int';
29  UNSIGNED: 'unsigned';
30  ID      : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
31
32  WS      : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 9.11: Eine Grammatik für Deklarationen und Befehle.

Übersetzen wir diese Grammatik mit ANTLR, so erhalten wir die folgende Warnung:

```
warning(200): cGrammar.g:4:5: Decision can match input such as "ID '(' ID ')'"
using multiple alternatives: 1, 3
As a result, alternative(s) 3 were disabled for that input
error(201): cGrammar.g:4:5: The following alternatives can never be matched: 3
```

ANTLR hat die Mehrdeutigkeit der Grammatik korrekt erkannt und liefert auch gleich das Beispiel, das auf zwei verschiedene Arten ableitbar ist.

Abbildung 9.12 zeigt, wie die Mehrdeutigkeit der Grammatik mit Hilfe eines semantischen Prädikats umgangen werden kann.

```
1  grammar cGrammar2;
2
3  @header {
4      import java.util.Set;
5      import java.util.TreeSet;
6  }
7
8  @members {
9      Set<String> mTypeNames = new TreeSet<String>();
10 }
11
12 declOrType
13     : decl
14     | typeDef
15     | stmnt
16     ;
17
18 decl: type identifier
19     | type '(' identifier ')'
20     ;
21
22 typeDef: TYPEDEF type identifier { mTypeNames.add($identifier.text); };
23
24 stmnt: identifier '(' identifier ')' ;
25
26 type: UNSIGNED
27     | INT
28     | UNSIGNED INT
29     | typeid
30     ;
31
32 typeid: { mTypeNames.contains(input.LT(1).getText(); }? ID;
33
34 identifier: ID;
35
36 TYPEDEF : 'typedef';
37 INT     : 'int';
38 UNSIGNED: 'unsigned';
39 ID      : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
40
41 WS      : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 9.12: Eine Grammatik mit semantischen Prädikat.

1. Wir merken uns in der in Zeile 9 definierten Member-Variable `mTypeNames` alle Namen, die als Typ-Bezeichner definiert werden.

2. Jedesmal, wenn wir in Zeile 22 einen neuen Typ-Bezeichner definieren, wird dieser der Menge der Typ-Bezeichner hinzugefügt.

3. Die Grammatik-Regel in Zeile 32, mit der wir Typ-Bezeichner erkennen, enthält mit

   ```
   { mTypeNames.contains(input.LT(1).getText()); }?
   ```

den Aufruf eines semantischen Prädikats. Hierin bezeichnet `input.LT(1)` das nächste Eingabe-Token. Den String dieses Tokens erhalten wir durch den Aufruf der Methode `getText()`, die für jedes Token den ursprünglich erkannten Text zurück liefert. Wir überprüfen, ob der erhaltene String in der Menge der bereits definierten Typ-Bezeichner liegt. Die Grammatik-Regel

$$\textit{typeid} \rightarrow \textsc{Id}$$

wird nun nur dann angewendet, wenn dies der Fall ist. Andernfalls wird ein ID mit Hilfe der Regel

$$\textit{identifier} \rightarrow \textsc{Id}$$

geparst.

Die allgemeine Form eines semantischen Prädikats in ANTLR ist

$$A \rightarrow \{\textit{Code}\}?\ Y_1 \cdots Y_k.$$

Hierbei ist $A \rightarrow Y_1 \cdots Y_k$ eine gewöhnliche Grammatik-Regel. Code ist ein Test, der `true` oder `false` ergibt. Falls *Code* zu `true` ausgewertet wird, kann die Regel angewendet werden, andernfalls ist die Regel nicht anwendbar. ANTLR sucht dann automatisch die nächste anwendbare Regel. Auf diese Weise lassen sich auch Grammatiken parsen, die nicht kontextfrei sind.

9.3 Syntaktische Prädikate und Backtracking

ANTLR ist in der Lage, *backtrackende* (Deutsch: rücksetzende) Parser zu erzeugen. Ein backtrackender Parser, der eine Variable A parsen muss, die durch die Grammatik-Regeln

$$A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n$$

definiert wird, versucht zunächst, die erste Regel $A \rightarrow \alpha_1$ zum Parsen zu verwenden. Gelingt dies, ist das Parsen von A erfolgreich beendet. Scheitert der Versuch, ein A mit der ersten Regel zu parsen, wird anschließend versucht, das A über die Regel $A \rightarrow \alpha_2$ zu parsen. Falls auch dieser Versuch scheitert, werden nacheinander die Regel $A \rightarrow \alpha_i$ für $i = 3, \cdots, n$ zum Parsen ausprobiert. Auf diese Weise läßt sich jeder String, der von der Variablen A abgeleitet werden kann, erkennen. Dies funktioniert sogar dann, wenn die verwendete Grammatik mehrdeutig ist. Der Nachteil des Verfahrens ist, dass im schlechtesten Fall ein exponentieller Rechenaufwand benötigt wird. ANTLR bietet aber durch die Verwendung syntaktischer Prädikate die Möglichkeit, dass Backtracking so zu steuern, dass der schlechteste Fall in der Praxis nicht auftritt.

9.3.1 Das Dangling-Else-Problem

Als motivierendes Beispiel betrachten wir das sogenannte *Dangling-Else-Problem*, das in der in Abbildung 9.13 auf Seite 132 gezeigten Grammatik auftritt. Diese Grammatik beschreibt ein Fragment der Sprache `C`. Das Problem ist, dass diese Grammatik mehrdeutig ist, denn der String

```
if (x < y) if (y < z) r = 1; else r = 2;
```

kann hier auf zwei Arten interpretiert werden: Einerseits kann die `else`-Klausel dem Test `y < z` zugerechnet werden, andererseits kann diese Klausel auch dem Test `x < y` zugeordnet werden: Im ersten Fall wäre die Semantik die selbe wie in dem Programm

```
if (x < y) {
   if (y < z) {
       r = 1;
   } else {
       r = 2;
   }
}
```

während die zweite Interpretation dem fogenden Programm-Ausschnitt entspricht:

```
if (x < y) {
    if (y < z) {
        r = 1;
    }
} else {
   r = 2;
}
```

Die Definition der Sprache `C` legt fest, dass eine `else`-Klausel immer dem unmittelbar vorhergehenden `if`-Befehl zugeordnet wird, so dass also die erste der beiden oben angegebenen Interpretationen die nach dem `C`-Standard korrekte Interpretation ist.

```
1   grammar dangling;
2
3   prog  : stmnt+;
4
5   stmnt : 'if' '(' boolExp ')' stmnt 'else' stmnt
6         | 'if' '(' boolExp ')' stmnt
7         | 'while' '(' boolExp ')' stmnt
8         | '{' stmnt* '}'
9         | ID '=' expr ';'
10        ;
11
12  expr  : ID
13        | NUMBER
14        ;
15
16  boolExp
17        : expr '==' expr
18        | expr '<'  expr
19        ;
20
21  ID     : ('a'..'z'|'A'..'Z')+;
22  NUMBER : ('0'..'9')+;
23  WS     : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 9.13: Fragment einer Grammatik für die Sprache `C`

Um das Problem der Mehrdeutigkeit zu lösen, gibt es mehrere Möglichkeiten:

1. Wir können die Grammatik so umschreiben, dass die Mehrdeutigkeit verschwindet.

   Die entsprechend umgeschriebene Grammatik ist allerdings fast doppelt so groß und erheblich komplexer als die ursprüngliche Grammatik und dadurch schwerer verständlich. Wir werden diesen Weg daher in diesem Kapitel nicht weiterverfolgen.

2. Wir können das in Antlr verfügbare Backtracking aktivieren.

3. Schließlich können wir das Problem durch Verwendung eines syntaktischen Prädikats lösen.

Lösung des Dangling-Else-Problem durch Backtracking

Abbildung 9.14 zeigt, wie sich das Dangling-Else-Problem durch Backtracking lösen läßt. Durch den Text

```
options {
    backtrack = true;
}
```

in den Zeilen 6 bis 8 spezifizieren wir, dass ANTLR für die Variable *stmnt* einen backtrackenden Parser erzeugen soll. Hier ist es nun wichtig, dass die Regel

stmnt → "`if`" "`(`" *boolExp* "`)`" *stmnt* "`else`" *stmnt*

an erster Stelle steht, denn dadurch wird der Parser zunächst versuchen, zu jedem `if` auch ein passendes `else` zu finden. Erst wenn dies nicht gelingt kommt die Regel

stmnt → "`if`" "`(`" *boolExp* "`)`" *stmnt*

zur Anwendung.

```
1  grammar danglingBacktrack;
2
3  prog  : stmnt+;
4
5  stmnt
6        options {
7            backtrack = true;
8        }
9        : 'if' '(' boolExp ')' stmnt 'else' stmnt
10       | 'if' '(' boolExp ')' stmnt
11       | 'while' '(' boolExp ')' stmnt
12       | '{' stmnt* '}'
13       | ID '=' expr ';'
14       ;
15
16 expr  : ID
17       | NUMBER
18       ;
19
20 boolExp
21       : expr '==' expr
22       | expr '<'  expr
23       ;
24
25 ID     : ('a'..'z'|'A'..'Z')+;
26 NUMBER : ('0'..'9')+;
27 WS     : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 9.14: Lösen des Dangling-Else-Problems durch Backtracking.

Lösung des Dangling-Else-Problem durch ein syntakisches Prädikat

Abbildung 9.15 zeigt, wie sich das Dangling-Else-Problem durch die Verwendung eines syntakischen Prädikats lösen läßt. In Zeile 5 und 6 haben wir geschrieben:

```
stmnt : ('if' '(' boolExp ')' stmnt 'else' stmnt)=>
        'if' '(' boolExp ')' stmnt 'else' stmnt
```

Der Ausdruck "`('if' '(' boolExp ')' stmnt 'else' stmnt)=>`" ist hier ein sogenanntes *syntakisches Prädikat*. Bevor der Parser versucht, ein *expr* mit der Regel

stmnt → “`if`” “`(`” *boolExp* “`)`” *stmnt* “`else`” *stmnt*

zu parsen, wird zunächst versucht, ob es möglich ist, den in `( )=>` gesetzten Ausdruck zu erkennen. Falls dies gelingt, wird die erste Regel angewendet, sonst versucht der Parser, den zu erkennenden String mit den restlichen Regeln zu parsen.

```
1  grammar danglingSyntactic;
2
3  prog  : stmnt+;
4
5  stmnt : ('if' '(' boolExp ')' stmnt 'else' stmnt)=>
6          'if' '(' boolExp ')' stmnt 'else' stmnt
7        | 'if' '(' boolExp ')' stmnt
8        | 'while' '(' boolExp ')' stmnt
9        | '{' stmnt* '}'
10       | ID '=' expr ';'
11       ;
12
13 expr  : ID
14       | NUMBER
15       ;
16
17 boolExp
18       : expr '==' expr
19       | expr '<'  expr
20       ;
21
22 ID     : ('a'..'z'|'A'..'Z')+;
23 NUMBER : ('0'..'9')+;
24 WS     : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 9.15: Lösen des Dangling-Else-Problems durch ein syntakisches Prädikat.

Die allgemeine Form eines syntakischen Prädikats ist

$$
\begin{array}{lcl}
A & \rightarrow & \text{“(”}\ \alpha_1\ \text{“)=>”}\ \beta_1 \\
 & | & \text{“(”}\ \alpha_2\ \text{“)=>”}\ \beta_2 \\
 & \vdots & \\
 & \vdots & \text{“(”}\ \alpha_{n-1}\ \text{“)=>”}\ \beta_{n-1} \\
 & \vdots & \text{“(”}\ \alpha_n\ \text{“)=>”}\ \beta_n
\end{array}
$$

Hierbei sind α_i und β_i für $i = 1, \cdots, n$ Folgen von Terminalen und syntakischen Variablen, wobei einige der α_i auch fehlen können. Der von ANTLR erzeugte Parser testet dann der Reihe nach, ob sich eines der α_i erkennen läßt und versucht anschließend für den ersten Index i, für den ein α_i erkant wird, die Regel $A \rightarrow \beta_i$ anzuwenden.

Vergleichen wir die Möglichkeiten, die syntakische Prädikate bieten, mit den durch Backtracking gebotenen Möglichkeiten, so stellen wir fest, dass Backtracking zwar einfacher anzuwenden ist, denn ist ist lediglich über die Option einzuschalten, dass der Parser aber durch die Verwendung von syntakischen Prädikaten präziser gesteuert werden kann. Die große Gefahr, die sowohl bei der Verwendung von Backtracking als auch bei der Verwendung von syntaktischen Prädikaten besteht ist die, dass damit eventuell Mehrdeutigkeiten der Grammatik zugedeckt werden, die dem Autor der Grammatik nicht bewußt sind. Es ist daher zu empfehlen von syntakischen Prädikaten oder Backtracking nur dann Gebrauch zu machen, wenn durch eine gründiche Analyse der Grammatik sichergestellt ist, dass durch die Verwendung solcher Prädikate keine unbeabsichtigten Ambivalenzen der Grammatik verschleiert werden.

Kapitel 10

Interpreter

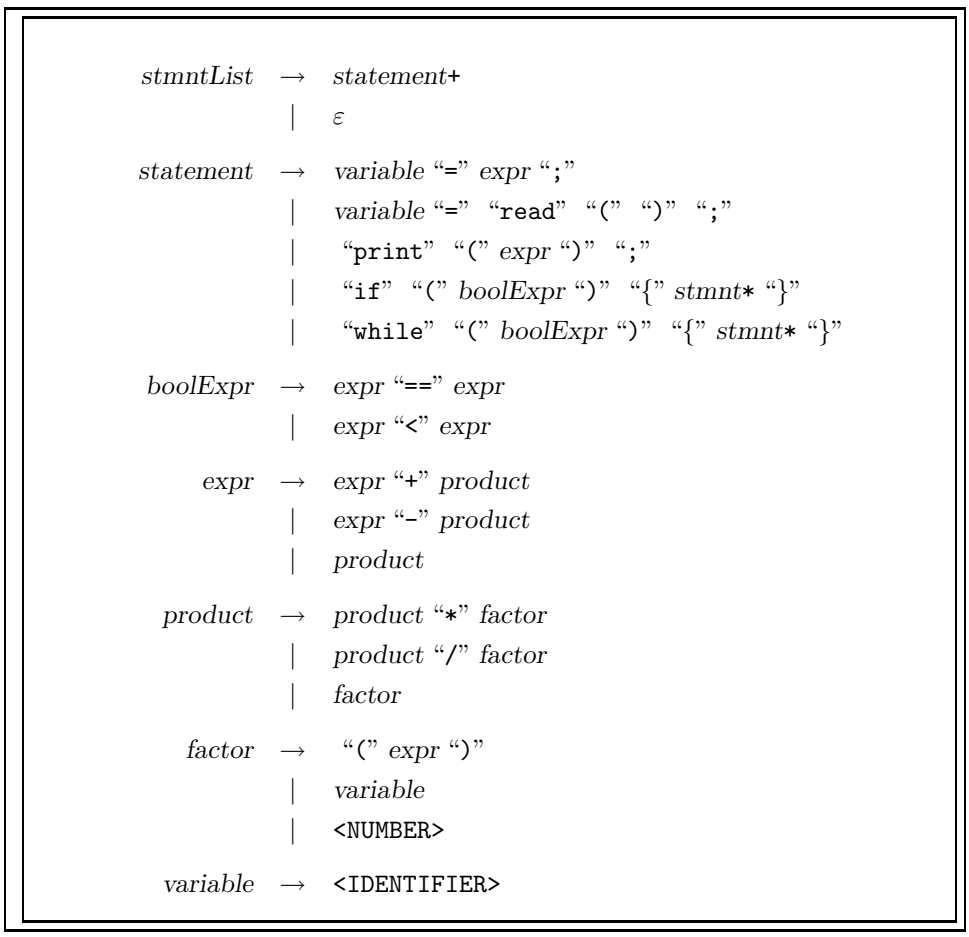
In diesem Kapitels erstellen wir mit Hilfe des Parser-Generators ANTLR einen Interpreter für eine einfache Programmiersprache. Abbildung 10.2 zeigt die EBNF-Grammatik dieser Programmier-Sprache. Die Befehle dieser Sprache sind Zuweisungen, Print-Befehle, `if`-Abfragen, sowie `while`-Schleifen. Abbildung 10.1 zeigt ein Beispiel-Programm, das dieser Grammatik entspricht.

```
1  n = read();
2  s = 0;
3  i = 0;
4  while (i < n * n) {
5      i = i + 1;
6      if (0 < i) {
7          print(i);
8          print(s);
9      }
10     s = s + i;
11 }
12 print(s);
```

Abbildung 10.1: Ein Programm zur Berechnung der Summe $\sum_{i=0}^{n} i$.

Die vorliegende Grammatik ist links-rekursiv und daher für ANTLR zunächst ungeeignet. Eliminieren wir die Links-Rekursion, so halten wir die in Abbildung 10.3 gezeigte Grammatik, die schon in der für ANTLR geeigneten Syntax notiert ist.

Um einen Interpreter für diese Sprache entwickeln zu können, benötigen wir zunächst Klassen, mit denen wir die einzelnen Befehle darstellen können. Wir beginnen mit der abstrakten Klasse `Statement`. Diese Klasse ist in Abbildung 10.4 gezeigt und dient dazu, Anweisungen unserer Programmier-Sprache darzustellen. Wir werden von der Klasse `Statement` später die Klassen `Assignment`, `Print`, `IfThen` und `While` ableiten. Die Klasse `Statement` ist selber abstrakt und enthält lediglich die Deklaration der abstrakten Methode *execute*(). Diese Methode können wir später benutzen, um einzelne Befehle ausführen.



stmntList	→	*statement*`+`
	|	ε
statement	→	*variable* "`=`" *expr* "`;`"
	|	*variable* "`=`" "`read`" "`(`" "`)`" "`;`"
	|	"`print`" "`(`" *expr* "`)`" "`;`"
	|	"`if`" "`(`" *boolExpr* "`)`" "{" *stmnt*`*` "}"
	|	"`while`" "`(`" *boolExpr* "`)`" "{" *stmnt*`*` "}"
boolExpr	→	*expr* "`==`" *expr*
	|	*expr* "`<`" *expr*
expr	→	*expr* "`+`" *product*
	|	*expr* "`-`" *product*
	|	*product*
product	→	*product* "`*`" *factor*
	|	*product* "`/`" *factor*
	|	*factor*
factor	→	"`(`" *expr* "`)`"
	|	*variable*
	|	`<NUMBER>`
variable	→	`<IDENTIFIER>`

Abbildung 10.2: EBNF-Grammatik für eine einfache Programmier-Sprache.

Abbildung 10.5 zeigt die Implementierung der Klasse `Assignment`. Da diese Klasse eine Zuweisung der Form

$$var = expr$$

darstellt, bei der einer Variablen *var* der Wert eines arithmetischen Ausdrucks *expr* zugewiesen wird, hat diese Klasse zwei Member-Variablen um die Variable und den Ausdruck abzuspeichern.

1. Die erste Member-Variable ist `mLhs`. Diese Member-Variable entspricht der Variablen auf der linken Seite des Zuweisungs-Operators "`=`".

2. Die zweite Member-Variable ist `mRhs`. Hier wird der arithmetische Ausdruck, der auf der rechten Seite des Zuweisungs-Operators steht, kodiert. Diese Member-Variable hat den Typ `Expr`. Hierbei handelt es sich um eine abstrakte Klasse zur Darstellung arithmetischer Ausdrücke, von der wir später konkrete Klassen ableiten. Diese Klasse besitzt eine abstrakte Methode *eval*(), mit der ein arithmetischer Ausdruck ausgewertet werden kann.

In der Klasse `Assignment` wertet die Methode *execute*() den Ausdruck, der in der Variablen `mRhs` gespeichert wird, mit Hilfe der für Objekte der Klasse `Expr` zur Verfügung stehenden Methode *eval*() aus und speichert den erhaltenen Wert in der Hashtabelle `sValueTable` unter dem Namen der Variablen ab. Es handelt sich bei dieser Tabelle um eine sogenannte *Symboltabelle*, in der die Werte der einzelnen Variablen abgelegt werden. Die Tabelle ist als statische Variable in der Klasse `Expr` definiert. Diese Klasse wird in Abbildung 10.9 auf Seite 141 gezeigt.

```
1  grammar Pure;
2
3  program
4      : statement+
5      ;
6
7  statement
8      : VAR '=' expr ';'
9      | VAR '=' 'read' '(' ')' ';'
10     | 'print' '(' expr ')' ';'
11     | 'if' '(' boolExpr ')' '{' statement* '}'
12     | 'while' '(' boolExpr  ')' '{' statement* '}'
13     ;
14
15 boolExpr
16     : expr ('==' expr | '<' expr)
17     ;
18
19 expr: product (('+' product | '-' product))*
20     ;
21
22 product
23     : factor (('*' factor | '/' factor))*
24     ;
25
26 factor
27     : '(' expr ')'
28     | VAR
29     | NUMBER
30     ;
```

Abbildung 10.3: Die EBNF-Grammatik nach Beseitigung der Links-Rekursion.

```
1  public abstract class Statement {
2      public abstract void execute();
3  }
```

Abbildung 10.4: Die abstrakte Klasse `Statement`

Abbildung 10.6 zeigt die Implementierung der Klasse `Read`. Diese Klasse stellt eine Zuweisung der Form

var `= read();`

dar. Daher besitzt diese Klasse eine Member-Variable `mVar`, die das Ergebnis der Lese-Operation abzuspeichert. Die Methode *execute*() gibt zunächst den String "`> `" als Eingabe-Aufforderung aus. Anschließend wird mit Hilfe der Methode *nextDouble*() eine Fließkommazahl gelesen, die dann in der Symboltabelle unter dem Namen der Variablen, der auf der linken Seite der Zuweisung steht, abgespeichert wird.

```
1  public class Assignment extends Statement {
2      Variable mLhs;
3      Expr     mRhs;
4
5      public Assignment(Variable lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public void execute() {
10         Expr.sValueTable.put(mLhs.mName, mRhs.eval());
11     }
12     public String toString() {
13         return mLhs + " = " + mRhs + ";";
14     }
15 }
```

Abbildung 10.5: Die Klasse `Assignment`.

```
1  public class Read extends Statement {
2      Variable mLhs;
3
4      public Read(Variable lhs) {
5          mLhs = lhs;
6      }
7      public void execute() {
8          Scanner scanner = new Scanner(System.in);
9          System.out.print("> "); // write prompt
10         System.out.flush();
11         Double value = scanner.nextDouble();
12         Expr.sValueTable.put(mLhs.mName, value);
13     }
14     public String toString() {
15         return mLhs + "= read();";
16     }
17 }
```

Abbildung 10.6: Die Klasse `Read`.

```
1  import java.util.*;
2
3  public class While extends Statement {
4      BoolExpr        mCond;
5      List<Statement> mStmntList;
6
7      public While(BoolExpr cond, List<Statement> stmntList) {
8          mCond      = cond;
9          mStmntList = stmntList;
10     }
11     public void execute() {
12         while (mCond.eval()) {
13             for (Statement stmnt: mStmntList) {
14                 stmnt.execute();
15             }
16         }
17     }
18     public String toString() {
19         String result = "while (" + mCond + ") {\n";
20         for (Statement stmnt: mStmntList) {
21             result += stmnt + "\n";
22         }
23         result += "}";
24         return result;
25     }
26 }
```

Abbildung 10.7: Die Klasse `While`.

Von den übrigen Klassen zur Darstellung von Befehlen diskutieren wir noch die Klasse `While`, die in Abbildung 10.7 gezeigt wird. Diese Klasse stellt einen Befehl der Form

$$\texttt{while}\,(s)\,\{\,\mathit{stmnts}\,\}$$

dar, wobei b ein Boole'scher Ausdruck ist, während *stmnts* eine Liste von Befehlen ist. Der Boole'sche Ausdruck wird in der Member-Variablen `mCond` gespeichert, die Liste von Befehlen findet sich in der Member-Variablen `mStmntList`. Zur Auswertung eines solchen Befehls führen wir solange alle Befehle in der Liste `mStmntList` aus, wie die Auswertung des Boole'schen Ausdrucks b den Wert `true` ergibt.

Die abstrakte Klasse `BoolExpr` dient zur Darstellung Boole'scher Ausdrücke. In unserem Fall sind das Ausdrücke der Form

$$l == r \quad \text{und} \quad l < r,$$

wobei Gleichungen durch die Klasse `Equal` dargestellt werden, während Ungleichung durch die Klasse `LessThan` dargestelt werden, die beide von der Klasse `BoolExpr` abgeleitet sind. Abbildung 10.8 zeigt die Klassen `BoolExpr` und `Equal`. Die Klasse `LessThan` ist analog zur Klasse `Equal` aufgebaut und wird daher nicht gezeigt.

```
1  public abstract class BoolExpr {
2      public abstract Boolean eval();
3  }
4  public class Equal extends BoolExpr {
5      Expr mLhs;
6      Expr mRhs;
7
8      public Equal(Expr lhs, Expr rhs) {
9          mLhs = lhs;
10         mRhs = rhs;
11     }
12     public Boolean eval() {
13         return mLhs.eval() == mRhs.eval();
14     }
15     public String toString() {
16         return mLhs + " == " + mRhs;
17     }
18 }
```

Abbildung 10.8: Klassen `BoolExpr` und `Equal`

```
1  import java.util.*;
2
3  public abstract class Expr {
4      public static HashMap<String, Double>
5           sValueTable = new HashMap<String, Double>();
6
7      public abstract Double eval();
8  }
```

Abbildung 10.9: Die abstrakte Klasse `Expr`.

Schließlich haben wir noch die Klassen, die zur Repräsentation von arithmetischen Ausdrücken benötigt werden. Diese Klassen werden alle von der abstrakten Klasse `Expr` abgeleitet, die in Abbildung 10.9 gezeigt ist.

1. Die Klasse `Expr` definiert die statische Variable `sValueTable`. Diese Variable beinhaltet eine Hashtabelle, in der für jede Variable, der ein Wert zugewiesen wurde, der aktuelle Wert dieser Variablen gespeichert ist.

2. Weiter deklariert die Klasse die abstrakte Methode *eval*(), mit der ein Ausdruck ausgewertet werden kann.

Von der Klasse `Expr` werden die Klassen `Sum`, `Difference`, `Product`, `Quotient`, `MyNumber` und `Variable` abgeleitet. Abbildung 10.10 zeigt die Klasse `Sum`. Da diese Klasse eine Summe der Form

$$l + r$$

darstellt, hat diese Klasse zwei Member-Variablen `mLhs` und `mRhs` um die beiden Summanden l und r darzustellen. Die Methode *eval* wertet diese beiden Member-Variablen getrennt aus und addiert das Ergebnis. Die Klassen `Difference`, `Product` und `Quotient` sind analog zur Klasse `Sum` aufgebaut und werden daher nicht weiter diskutiert.

```
1  public class Sum extends Expr {
2      Expr mLhs;
3      Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Double eval()    {
10         return mLhs.eval() + mRhs.eval();
11     }
12     public String toString() {
13         return mLhs.toString() + " + " + mRhs.toString();
14     }
15 }
```

Abbildung 10.10: Die Klasse `Sum`

```
1  public class Variable extends Expr {
2      String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public Double eval() {
8          return sValueTable.get(mName);
9      }
10     public String toString() {
11         return mName;
12     }
13 }
```

Abbildung 10.11: Die Klasse `Variable`.

Abbildung 10.11 zeigt die Implementierung der Klasse `Variable`. Die Methode *eval*() wertet eine Variable dadurch aus, dass sie unter dem Namen der Variablen in der Hashtabelle `sValueTable` den zugeordneten Wert nachschlägt.

Abbildung 10.12 zeigt das Treiber-Programm, das den von ANTLR erzeugten Parser einbindet. Das Programm soll später in der Form

`java SLInterpreter` $file_1 \cdots file_n$

aufgerufen werden. Hierbei bezeichte $file_i$ für $i = 1, \cdots, n$ eine Datei, die ein zu interpretierendes Programm enthält. Falls anstelle eines Dateinamens der String "`-`" als Argument übergeben wird, sollen die Befehle statt dessen interaktiv eingegeben werden. Die Methode *parseFile*() behandelt dabei den Fall, dass die Befehle aus einer Datei gelesen werden, während die Methode *parseInteractive*() für den Fall der interaktiven Verarbeitung zuständig ist. Die in der Methode *parseInteractive*() verwendete Klasse `MyReader` wird dazu benötigt, um von der Standardeingabe zu lesen und das Gelesene anschließend als `StringBufferInputStream` zurück zu geben.

```
1  import org.antlr.runtime.*;
2  import java.util.List;
3  import java.io.*;
4
5  public class SLInterpreter {
6
7      public static void main(String[] args) throws Exception {
8          for (String file: args) {
9              if (!file.equals("-")) {
10                 parseFile(file);
11             } else {
12                 parseInteractive();
13             }
14         }
15         if (args.length == 0) {
16             parseInteractive();
17         }
18     }
19     private static void parseFile(String fileName) throws Exception {
20         try {
21             ANTLRStringStream ss = new ANTLRFileStream(fileName);
22             parseAndExecute(ss);
23         } catch (IOException e) {
24             System.err.println("File " + fileName + " could not be read.");
25         }
26     }
27     private static void parseInteractive() throws Exception {
28         while (true) {
29             StringBufferInputStream stream = MyReader.getStream();
30             ANTLRInputStream input = new ANTLRInputStream(stream);
31             parseAndExecute(input);
32         }
33     }
34     private static void parseAndExecute(ANTLRStringStream stream)
35         throws Exception
36     {
37         SimpleLexer       lexer  = new SimpleLexer(stream);
38         CommonTokenStream ts     = new CommonTokenStream(lexer);
39         SimpleParser      parser = new SimpleParser(ts);
40         List<Statement>   stmnts = parser.program();
41         for (Statement s: stmnts) {
42             s.execute();
43         }
44     }
45 }
```

Abbildung 10.12: Die Klasse `SLInterpreter`.

```
grammar Simple;

@header {
    import java.util.List;
    import java.util.ArrayList;
}
program returns [List<Statement> program]
    : { $program = new ArrayList<Statement>(); }
      (s = statement { program.add($s.stmnt); })+
    ;
statement returns [Statement stmnt]
    : v = VAR '=' e = expr ';'
      { $stmnt = new Assignment(new Variable($v.text), $e.result); }
    | v = VAR '=' 'read' '(' ')' ';'
      { $stmnt = new Read(new Variable($v.text)); }
    | 'print' '(' r = expr ')' ';'
      { $stmnt = new Print($r.result); }
    | { List<Statement> stmnts = new ArrayList<Statement>(); }
      'if' '(' b = boolExpr ')' '{'
      (l = statement { stmnts.add($l.stmnt); })* '}'
      { $stmnt = new IfThen($b.result, stmnts); }
    | { List<Statement> stmnts = new ArrayList<Statement>(); }
      'while' '(' b = boolExpr() ')' '{'
      (l = statement { stmnts.add($l.stmnt); })* '}'
      { $stmnt = new While($b.result, stmnts); }
    ;
boolExpr returns [BoolExpr result]
    : l = expr ( '==' r = expr { $result = new Equal(   $l.result, $r.result); }
               | '<'  r = expr { $result = new LessThan($l.result, $r.result); }
               )
    ;
expr returns [Expr result]
    : p = product { $result = $p.result; }
      (   ('+' q = product) { $result = new Sum(       $result, $q.result); }
        | ('-' q = product) { $result = new Difference($result, $q.result); }
      )*
    ;
product returns [Expr result]
    : f = factor { $result = $f.result; }
      ( ('*' g = factor) { $result = new Product( $result, $g.result); }
        | ('/' g = factor) { $result = new Quotient($result, $g.result); }
      )*
    ;
factor returns [Expr result]
    : '(' expr ')' { $result = $expr.result;          }
    | v = VAR      { $result = new Variable($v.text); }
    | n = NUMBER   { $result = new MyNumber($n.text); }
    ;
```

Abbildung 10.13: ANTLR-Spezifikation der Grammatik.

Abbildung 10.13 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR. Die Spezifikation der Token ist in Abbildung 10.14 gezeigt. Der Scanner unterscheidet im wesentlichen zwischen Variablen und Zahlen. Variablen beginnen mit einem großen oder kleinen Buchstaben, auf den dann zusätzlich Ziffern und der Unterstrich folgen können. Folgen von Ziffern werden als Zahlen interpretiert. Enthält eine solche Folge mehr als ein Zeichen, so darf die erste Ziffer nicht 0 sein. Darüber hinaus entfernt der Scanner Whitespace und Kommentare.

```
49
50  VAR     : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
51  NUMBER  : '0'..'9' | ('1'..'9')('0'..'9')+;
52
53  MULTI_COMMENT : '/*' (~('*') | '*'+ ~('*'|'/'))* '*'+ '/' { skip(); };
54  LINE_COMMENT  : '//' ~('\n')*                              { skip(); };
55  WS            : (' '|'\t'|'\n'|'\r') { skip(); };
```

Abbildung 10.14: ANTLR-Spezifikation der Token.

Aufgabe 16:

1. Erweitern Sie den oben diskutierten Interpreter um `for`-Schleifen.
2. Reichern Sie unsere Beispiel-Programmier-Sprache um die logischen Operatoren "`&&`" für das logische *Und*, "`||`" für das logische *Oder* und "`!`" für die Negation an. Dabei soll der Operator "`!`" am stärksten und der Operator "`||`" am schwächsten binden.
3. Erweitern Sie die Syntax der arithmetischen Ausdrücke so, dass auch vordefinierte mathematische Funktionen wie *exp*() oder *ln*() benutzt werden können.

   Hinweis: Wenn Sie das Paket *java.lang.reflect* benutzen, kommen Sie mit einer zusätzlichen Klasse aus und können damit alle in `java.lang.Math` definierten Methoden implementieren.

4*. Erweitern Sie den Interpreter so, dass auch benutzerdefinierte Funktionen möglich werden.

   Hinweis: Jetzt müssen Sie zwischen lokalen und globalen Variablen unterscheiden. Daher reicht es nicht mehr, die Belegungen der Variablen in einer global definierten Hashtabelle zu verwalten. Grundsätzlich gibt es zwei Möglichkeiten, die Variablen zu verwalten:

   (a) Sie können die Variablen über einen Stack von Symboltabellen verwalten. Oben auf dem Stack liegt dann immer die Symboltabelle, die gerade aktuell ist. Insgesamt entspricht jedem Element des Stacks dann ein (geschachtelter) Funktions-Aufruf.

   (b) Alternativ können Sie den Methoden *eval*() und *execute*() als zusätzliches Argument eine Symboltabelle mitgeben, in der die momentane Belegung der Variablen gespeichert wird.

Kapitel 11

Grenzen kontextfreier Sprachen

In diesem Kapitel diskutieren wir die Grenzen kontextfreier Sprachen und leiten dazu dass sogenannte "*große Pumping-Lemma*" her, mit dessen Hilfe wir beispielsweise zeigen können, dass die Sprache L_{square}, die durch

$$L_{square} = \{ww \mid w \in \Sigma^*\}$$

definiert wird, für das Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ keine kontextfreie Sprache ist.

11.1 Die verallgemeinerte Chomsky-Normalform

In diesem Abschnitt zeigen wir, wie eine kontextfreie Sprache in die *verallgemeinerte Chomsky-Normalform* (vCNF) überführt werden kann. Die Überführung einer Sprache in verallgemeinerte Chomsky-Normalform erfolgt in mehreren Schritten.

1. Zunächst beseitigen wir *nutzlose Symbole*.

   Ist $G = \langle V, T, R, S\rangle$ eine Grammatik, so nennen wir eine syntaktische Variable $A \in V$ *nützlich*, wenn es einen String $w \in T^*$ sowie Strings $\alpha, \beta \in (V \cup T)^*$ gibt, so dass

   $$S \Rightarrow^* \alpha A \beta \Rightarrow^* w$$

   gilt. Eine syntaktische Variable ist also genau dann nützlich, wenn diese Variable in der Herleitung eines Wortes $w \in L(G)$ verwendet werden kann.

   Ein Terminal t ist *nützlich*, wenn es Wörter $w_1, w_2 \in T^*$ gibt, so dass

   $$S \Rightarrow^* w_1 t w_2$$

   gilt. Ein Terminal t ist also genau dann nützlich, wenn es in einem Wort der Sprache $L(G)$ auftritt. Variablen und Terminale, die nicht nützlich sind, bezeichnen wir als *nutzlose Symbole*.

2. Anschließend zeigen wir, dass alle *ε-Regeln* aus der Grammatik eliminiert werden können. Dabei bezeichnen wir eine Grammatik-Regel der Form

   $$A \to \varepsilon$$

   als eine *ε-Regel*.

3. Schließlich zeigen wir, wie alle Grammatik-Regeln der Form

   $$A \to B$$

   mit $A, B \in V$ eliminiert werden können. Solche Grammatik-Regeln werden als *Unit-Regeln* bezeichnet.

11.1.1 Beseitigung nutzloser Symbole

Die Erkennung nutzloser Symbole ist eine Überprüfung, die in manchen Parser-Generatoren (beispielsweise in *Bison*[1]) eingebaut ist, weil das Auftreten nutzloser Symbole oft einen Hinweis darauf gibt, dass die Grammatik nicht die Sprache beschreibt, die intendiert ist. Insofern ist die jetzt vorgestellte Technik auch von praktischem Interesse. Wir beginnen mit zwei Definitionen.

Definition 28 (erzeugende Variable) Eine syntaktische Variable $A \in V$ einer Grammatik $G = \langle V, T, R, S\rangle$ ist eine *erzeugende* Variable, wenn es ein Wort $w \in T^*$ gibt, so dass

$$A \Rightarrow^* w$$

gilt, eine erzeugende Variable erzeugt also mindestens ein Wort aus T^*. Die Notation $A \Rightarrow^* w$ drückt dabei aus, dass aus der Variablen A in endlich vielen Schritten das Wort w abgeleitet werden kann. □

Offenbar ist eine syntaktische Variable, die nicht erzeugend ist, nutzlos. Die Menge aller erzeugenden Variablen einer Grammatik G kann mit Hilfe der folgenden induktiven Definition gefunden werden.

1. Enthält die Grammatik $G = \langle V, T, R, S\rangle$ eine Regel der Form

   $$A \to w \quad \text{mit } w \in T^*,$$

   so ist die Variable A offenbar erzeugend.

2. Enthält die Grammatik $G = \langle V, T, R, S\rangle$ eine Regel der Form

   $$A \to \alpha$$

   und sind alle syntaktischen Variablen, die in dem Wort α auftreten, bereits als erzeugende Variablen erkannt, so ist auch die syntaktische Variable A erzeugend.

Beispiel: Es sei $G = \langle \{S, A, B, C\}, \{\mathtt{x}\}, R, S\rangle$ und die Menge der Regeln R sei wie folgt gegeben:

$$\begin{array}{lcl} S & \to & ABC \mid A \\ A & \to & AA \mid \mathtt{x} \\ B & \to & AC \\ C & \to & BA \end{array}$$

Aufgrund der Regel

$$A \to \mathtt{x}$$

ist zunächst A erzeugend. Aufgrund der Regel

$$S \to A$$

ist dann auch S erzeugend. Die Variablen B und C sind hingegen nicht erzeugend und damit sicher nutzlos. □

Ist $G = \langle V, T, R, S\rangle$ eine Grammatik und ist E die Menge der erzeugenden Variablen, so können wir alle Variablen, die nicht erzeugend sind, einfach weglassen. Zusätzlich müssen wir natürlich auch die Regeln weglassen, in denen Variablen auftreten, die nicht erzeugend sind. Dabei ändert sich die von der Grammatik G erzeugte Sprache offenbar nicht.

Eine Variable kann gleichzeitig erzeugend und trotzdem nutzlos sein. Als einfaches Beispiel betrachten wir die Grammatik $G = \langle \{S, A, B\}, \{\mathtt{x}, \mathtt{y}\}, R, S\rangle$, deren Regeln durch

$$\begin{array}{lcl} S & \to & A\mathtt{y} \\ A & \to & AA \mid \mathtt{x} \\ B & \to & A\mathtt{y} \end{array}$$

[1] *Bison* ist ein Parser-Generator für die Sprache `C`.

gegeben sind. Die erzeugenden Variable sind in diesem Falle A, B und S. Die Variable B ist trotzdem nutzlos, denn von S aus ist diese Variable gar nicht *erreichbar*. Formal definieren wir für eine Grammatik $G = \langle V, T, R, S\rangle$ eine syntaktische Variable X als *erreichbar*, wenn es Wörter $\alpha, \beta \in (V \cup T)^*$ gibt, so dass gilt:

$$S \Rightarrow^* \alpha X \beta.$$

Für eine gegebene Grammatik G läßt sich die Menge der Variablen, die erreichbar sind, mit dem folgenden induktiven Algorithmus berechnen.

1. Das Start-Symbol S ist erreichbar.
2. Enthält die Grammatik G eine Regel der Form

   $$X \to \alpha$$

   und ist X erreichbar, so sind auch alle Variablen, die in α auftreten, erreichbar.

Offenbar sind Variablen, die nicht erreichbar sind, nutzlos und wir können diese Variablen, sowie alle Regeln, in denen diese Variablen auftreten, weglassen, ohne dass sich dabei die Sprache ändert. Damit haben wir jetzt ein Verfahren, um aus einer Grammatik alle nutzlosen Variablen zu entfernen.

1. Zunächst entfernen wir alle Variablen, die nicht erzeugend sind.
2. Anschließend entfernen wir alle Variablen, die nicht erreichbar sind.

Es ist wichtig zu verstehen, dass die Reihenfolge der obigen Regeln nicht umgedreht werden darf. Dazu betrachten wir die Grammatik $G = \langle \{S, A, B, C\}, \{\mathtt{x}, \mathtt{y}\}, R, S\rangle$, deren Regeln durch

$$\begin{array}{lcl} S & \to & BC \mid A \\ A & \to & AA \mid \mathtt{x} \\ B & \to & \mathtt{y} \\ C & \to & CC \end{array}$$

gegeben sind. Die beiden Regeln

$$S \to BC \quad \text{und} \quad S \to A$$

zeigen, dass alle Variablen erreichbar sind. Weiter sehen wir, dass die Variablen A, B und S erzeugend sind, denn es gilt

$$A \Rightarrow^* \mathtt{x}, \quad S \Rightarrow^* \mathtt{x} \quad \text{und} \quad B \Rightarrow^* \mathtt{y}.$$

Damit sieht es zunächst so aus, als ob nur C nutzlos ist. Das stimmt aber nicht, auch die Variable B ist nutzlos, denn wenn wir die Variable C aus der Grammatik entfernen, dann wird auch die Regel

$$S \to BC$$

entfernt und damit ist dann B nicht mehr erreichbar und somit ebenfalls nutzlos.

An dieser Stelle können wir uns fragen, warum es funktioniert, wenn wir erst alle Variablen entfernen, die nicht erzeugend sind und anschließend dann die nicht erreichbaren Variablen entfernen. Der Grund ist, dass eine Variable B, die nicht erreichbar ist, niemals von einer anderen Variablen A, die erreichbar ist, benötigt wird. Wenn also A erreichbar ist und gleichzeitig erzeugend, dann kann B getrost entfernt werden, denn das kann an der Tatsache, dass A erzeugend ist, nichts ändern.

11.1.2 Beseitigung von ε-Regeln

Ist G_1 eine Grammatik, so dass die Sprache $L(G_1)$ den leeren String nicht enthält, so können wir eine Grammatik G_2 finden, welche keine Regeln der Form

$$A \to \varepsilon$$

enthält und welche die selbe Sprache beschreibt wie die Grammatik G_1, es gilt also $L(G_1) = L(G_2)$. Regeln der Form $A \to \varepsilon$ bezeichnen wir als ε-Regeln. Bei bestimmten Algorithmen stören solche ε-Regeln, deswegen wollen wir jetzt ein Verfahren entwickeln, mit dem ε-Regeln eliminiert werden können. Eine vollständige Formalisierung des Verfahrens würde die eigentliche Idee verschleiern, deswegen diskutieren wir das Verfahren an Hand eines Beispiels.

Beispiel: Die Regeln der im folgenden angegebenen Grammatik G_1 beschreiben arithmetische Ausdrücke, in denen Zahlen addiert oder subtrahiert werden. Wir verwenden dabei die syntaktischen Variablen *Expr* und *ExprRest* sowie das Terminal `NUMBER`:

Expr	$\to$	`NUMBER` *ExprRest*
ExprRest	$\to$	"`+`" `NUMBER` *ExprRest*
	$\mid$	"`-`" `NUMBER` *ExprRest*
	$\mid$	ε

Eliminieren wir die ε-Regeln aus dieser Grammatik, so erhalten wir die folgende Grammatik:

Expr	$\to$	`NUMBER` *ExprRest*
	$\mid$	`NUMBER`
ExprRest	$\to$	"`+`" `NUMBER` *ExprRest*
	$\mid$	"`+`" `NUMBER`
	$\mid$	"`-`" `NUMBER` *ExprRest*
	$\mid$	"`-`" `NUMBER`

Im Allgemeinen ist die Idee, dass alle Regeln der Form

$$B \to \varepsilon$$

aus der Grammatik entfernt werden. Gleichzeitig wird zu jeder Regel der Form

$$A \to \alpha B \gamma,$$

bei denen die Variable B genau einmal auf der rechten Seite der Regel auftritt, eine neue Regel der Form

$$A \to \alpha\gamma$$

hinzugefügt. Haben wir eine Regel, bei der die Variable B zweimal auf der rechten Seite auftritt, die Regel hat dann also die Form

$$A \Rightarrow \alpha B \gamma B \delta,$$

so fügen wir der Grammatik die folgenden drei Regeln hinzu:

1. $A \Rightarrow \alpha B \gamma \delta$,
2. $A \Rightarrow \alpha \gamma B \delta$,
3. $A \Rightarrow \alpha \gamma \delta$.

Im Allgemeinen gilt: Enthält die rechte Seite einer Regel die syntaktische Variable B an n verschiedenen Stellen, so werden der Grammatik $2^n - 1$ neue Regeln hinzugefügt.

Bei der Elimination der ε-Produktionen kann die Zahl der Regeln stark anwachsen. Das Verfahren ist daher nicht von praktischem Interesse. Wir benötigen es lediglich zum späteren Nachweis der Tatsache, dass für alle kontextfreien Sprachen, die das leere Wort nicht enthalten, eine Grammatik in *verallgemeinerter Chomsky-Normalform* angegeben werden kann.

11.1.3 Beseitigung von Unit-Regeln

Regeln der Form

$$A \to B,$$

bei der die rechte Seite nur aus einer einzigen Variablen besteht, heißen *Unit-Regeln*. Diese Unit-Regeln können wie folgt eliminiert werden: Sind alle Regeln, bei denen die Variable B auf der linken Seite steht, durch

$$B \to \beta_1 \mid \cdots \mid \beta_n$$

gegeben, so können wir die Regel $A \to B$ durch die Regeln

$$A \to \beta_1 \mid \cdots \mid \beta_n$$

ersetzen und erhalten dabei eine Grammatik, die die selbe Sprache beschreibt.

Beispiel: Wir betrachten die folgende Grammatik für arithmetische Ausdrücke.

$$\begin{array}{rcl}
\mathit{Expr} & \to & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
& \mid & \mathit{Product} \\
\mathit{Product} & \to & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& \mid & \mathit{Factor} \\
\mathit{Factor} & \to & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& \mid & \texttt{NUMBER}
\end{array}$$

Diese Grammatik enthält die folgenden beiden Unit-Regeln:

$$\begin{array}{rcl}
\mathit{Expr} & \to & \mathit{Product} \\
\mathit{Product} & \to & \mathit{Factor}
\end{array}$$

Die Regel $\mathit{Factor} \to$ `NUMBER` ist keine Unit-Regeln, denn `NUMBER` ist ein Terminal und keine syntaktische Variable. Wir eliminieren zunächst die Unit-Regel $\mathit{Expr} \to \mathit{Product}$. Wir erhalten dann die folgende Grammatik:

$$\begin{array}{rcl}
\mathit{Expr} & \to & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
& \mid & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& \mid & \mathit{Factor} \\
\mathit{Product} & \to & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& \mid & \mathit{Factor} \\
\mathit{Factor} & \to & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& \mid & \texttt{NUMBER}
\end{array}$$

Hier haben wir eine neue Unit-Regel bekommen und zwar die Regel

$$\mathit{Expr} \to \mathit{Factor}$$

Eliminieren wir diese Unit-Regel, so erhalten wir:

$$\begin{array}{rcl}
\mathit{Expr} & \to & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
& \mid & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& \mid & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& \mid & \texttt{NUMBER} \\
\mathit{Product} & \to & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& \mid & \mathit{Factor}
\end{array}$$

$$
\begin{array}{rcl}
\textit{Factor} & \rightarrow & \text{“(” } \textit{Expr} \text{ “)”} \\
& | & \texttt{NUMBER}
\end{array}
$$

Eliminieren wir nun die Unit-Regel $\textit{Product} \rightarrow \textit{Factor}$, so erhalten wir:

$$
\begin{array}{rcl}
\textit{Expr} & \rightarrow & \textit{Expr} \text{ “+” } \textit{Product} \\
& | & \textit{Product} \text{ “*” } \textit{Factor} \\
& | & \text{“(” } \textit{Expr} \text{ “)”} \\
& | & \texttt{NUMBER} \\
\textit{Product} & \rightarrow & \textit{Product} \text{ “*” } \textit{Factor} \\
& | & \text{“(” } \textit{Expr} \text{ “)”} \\
& | & \texttt{NUMBER} \\
\textit{Factor} & \rightarrow & \text{“(” } \textit{Expr} \text{ “)”} \\
& | & \texttt{NUMBER}
\end{array}
$$

Wie man sieht, wächst die Grammatik durch die Eliminierung von Unit-Regeln stark an. Das oben skiziierte Verfahren kann sehr mühsam sein, wenn es einen *Zyklus* von Unit-Regeln gibt, wie das bei den folgenden Grammatik-Regeln der Fall ist:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid B \\
B & \rightarrow & \texttt{y} \mid C \\
C & \rightarrow & \texttt{z} \mid A
\end{array}
$$

Eliminieren wir die Regel $A \rightarrow B$, so erhalten wir:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid \texttt{y} \mid C \\
B & \rightarrow & \texttt{y} \mid C \\
C & \rightarrow & \texttt{z} \mid A
\end{array}
$$

Eliminieren wir nun die Regel $B \rightarrow C$, so erhalten wir:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid \texttt{y} \mid C \\
B & \rightarrow & \texttt{y} \mid \texttt{z} \mid A \\
C & \rightarrow & \texttt{z} \mid A
\end{array}
$$

Eliminieren wir jetzt die Regel $C \rightarrow A$, so erhalten wir:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid \texttt{y} \mid C \\
B & \rightarrow & \texttt{y} \mid \texttt{z} \mid A \\
C & \rightarrow & \texttt{z} \mid \texttt{x} \mid \texttt{y} \mid C
\end{array}
$$

Hier können wir die Regel $C \rightarrow C$ ersatzlos streichen und anschließend die Unit-Regel $A \rightarrow C$ eliminieren. Das liefert:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
B & \rightarrow & \texttt{y} \mid \texttt{z} \mid A \\
C & \rightarrow & \texttt{z} \mid \texttt{x} \mid \texttt{y}
\end{array}
$$

Eliminieren wir hier noch die Unit-Regel $B \rightarrow A$, so erhalten wir:

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
B & \rightarrow & \texttt{y} \mid \texttt{z} \mid \texttt{x} \\
C & \rightarrow & \texttt{z} \mid \texttt{x} \mid \texttt{y}
\end{array}
$$

Ist die Variable A das Start-Symbol der Grammatik, so sind die Variablen B und C durch die Beseitigung der Unit-Regeln nutzlos geworden sind.

Definition 29 (Verallgemeinerte Chomsky-Normalform) Eine Grammatik G ist in *verallgemeinerter Chomsky-Normalform* wenn die Grammatik keine Unit-Regeln und keine ε-Produktionen enthält. Außerdem darf die Grammatik keine nutzlosen Symbole haben. □

Mit Hilfe der in diesem Abschnitt vorgeführten Verfahren läßt sich jede Grammatik G, für die $\varepsilon \notin L(G)$ gilt, zu einer äquivalenten Grammatik umschreiben, die in verallgemeinerter Chomsky-Normalform ist. Dazu führen wir die folgenden Schritte in nachstehender Reihenfolge durch:

1. Wir eliminieren die ε-Regeln.
2. Danach entfernen wir die Unit-Regeln.
3. Schließlich beseitigen wir die nutzlosen Symbole.

Die dabei entstehende Grammatik hat dann offenbar die verallgemeinerte Chomsky-Normalform, denn bei der Beseitigung von Unit-Regeln werden keine ε-Regeln neu eingeführt und durch die Beseitigung von nutzlosen Symbolen entstehen weder ε-Regeln noch Unit-Regeln.

11.2 Parse-Bäume als Listen

Zum Beweis des Pumping-Lemmas für kontextfreie Sprachen benötigen wir eine Abschätzung, bei der wir die Länge eines Wortes w aus einer kontextfreien Sprachen $L(G)$ mit der Höhe des Parse-Baums für W in Verbindung bringen. Es zeigt sich, dass es zum Nachweis dieser Abschätzung hilfreich ist, wenn wir einen Parse-Baum als Liste aller Pfade des Parse-Baums auffassen. Die Idee wird am ehesten an Hand eines Beispiels klar. Abbildung 11.1 zeigt einen Parse-Baum für den String "`2*3+4`".

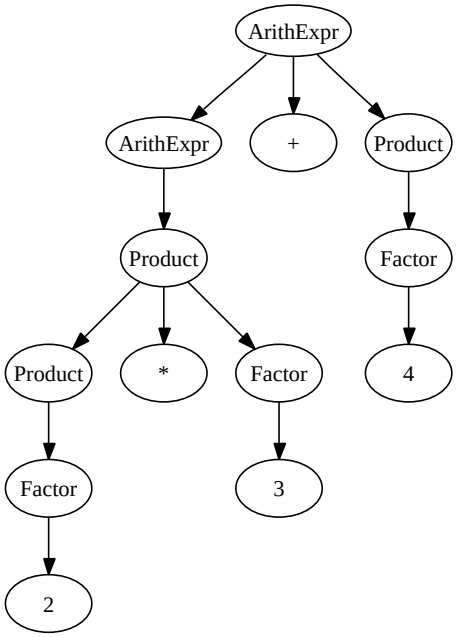


Abbildung 11.1: Ein Parse-Baum für den String "`2*3+4`".

Fassen wir diesen Parse-Baum als Liste seiner Zweige auf, wobei jeder Zweig eine Liste von Grammatik-Symbolen ist, so erhalten wir die folgende Liste:

$$
\begin{array}{ll}
\big[& [Expr, Expr, Product, Product, Factor,\ \text{“2”}\ ], \\
& [Expr, Expr, Product,\ \text{“*”}\ ], \\
& [Expr, Expr, Product, Factor,\ \text{“3”}\ ], \\
& [Expr,\ \text{“+”}\ ], \\
& [Expr, Product, Factor,\ \text{“4”}\ ] \\
\big].
\end{array}
$$

Definition 30 (*parseTree*) Ist $G = \langle V, T, R, S\rangle$ eine Grammatik, ist $w \in T^*$, $A \in V$ und gibt es eine Ableitung

$$A \Rightarrow^* w,$$

so definieren wir $\mathit{parseTree}(A \Rightarrow^* w)$ induktiv:

1. Falls $(A \to t_1 t_2 \cdots t_m) \in R$ mit $A \in V$ und $t_i \in T$ für alle $i = 1, \cdots, m$, so setzen wir

   $$\mathit{parseTree}(A \Rightarrow^* t) = \big[\, [A, t_1],\, [A, t_2],\, \cdots,\, [A, t_m] \,\big].$$

2. Falls $A \Rightarrow^* w$ gilt, weil

   $$(A \to B_1 B_2 \cdots B_m) \in R, \quad B_i \Rightarrow^* w_i \text{ für alle } i = 1, \cdots, m, \quad \text{mit } w = w_1 w_2 \cdots w_m$$

   gilt, so definieren wir (unter Benutzung der SETL-Notation für die Definition von Listen)

   $$\mathit{parseTree}(A \Rightarrow^* w) = \big[\, [A] + l \,:\, i \in [1, \cdots, m],\, l \in \mathit{parseTree}(B_i \Rightarrow^* w_i) \,\big].$$

Wir definieren die *Breite* b einer Grammatik als die größte Anzahl von Symbolen, die auf der rechten Seite einer Grammatik-Regel der Form

$$A \to \alpha$$

auftreten, bei der α kein einzelnes Terminal ist. Mit dieser Definition ist klar, dass die Breite einer nicht-trivialen Grammatik in verallgemeinerter Chomsky-Normalform mindestens den Wert zwei hat, denn eine solche Grammatik enthält ja weder ε-Regeln noch Unit-Regeln. Falls die Grammatik also eine Regel enthält, auf deren rechten Seite eine Variable auftritt, dann muss die rechte Seite dieser Regel mindestens zwei verschiedene Zeichen enthalten.

Lemma 31 Die Grammatik $G = \langle V, T, R, S\rangle$ habe die Breite b. Ferner gelte

$$A \Rightarrow^* w$$

für eine syntaktische Variable $A \in V$ und ein Wort $w \in T^*$. Falls n die Länge der längsten Liste in

$$\mathit{parseTree}(A \Rightarrow^* w)$$

ist, so gilt für die Länge des Wortes w die Abschätzung

$$|w| \leq b^n.$$

Beweis: Wir führen den Beweis durch Induktion nach n.

I.A. $n = 2$: Dann besteht die Ableitung nur aus einem Schritt und daher muss es eine Regel der Form

$$A \to t_1 t_2 \cdots t_m \quad \text{mit } w = t_1 t_2 \cdots t_m \text{ und } t_i \in T \text{ für alle } i = 1, \cdots, m$$

in der Grammatik G geben. Es gilt dann

$$\mathit{parseTree}(A \Rightarrow^* w) = \big[\, [A, t_1],\, [A, t_2],\, \cdots,\, [A, t_m] \,\big].$$

Daraus folgt

$$|w| = |t_1 t_2 \cdots t_m| = m \leq b \leq b^2,$$

wobei die Ungleichung $m \leq b$ aus der Tatsache folgt, dass Länge der Regeln der Grammatik G durch die Breite b beschränkt ist.

I.S. $n \mapsto n+1$: Da die Ableitung nun aus mehr als einem Schritt besteht, hat die Ableitung die Form

$$A \Rightarrow B_1 B_2 \cdots B_m \Rightarrow^* w_1 w_2 \cdots w_m = w.$$

Außerdem haben dann die Listen in

$$\mathit{parseTree}(B_i \Rightarrow^* w_i)$$

für alle $i = 1, \cdots, m$ höchstens die Länge n. Nach Induktions-Voraussetzung wissen wir also, dass

$$|w_i| \leq b^n \quad \text{für alle } i = 1, \cdots, m$$

gilt. Daher haben wir

$$|w| = |w_1| + \cdots + |w_m| \leq b^n + \cdots + b^n = m \cdot b^n \leq b \cdot b^n \leq b^{n+1}.$$ □

11.3 Das Pumping-Lemma für kontextfreie Sprachen

Satz 32 (Pumping-Lemma) Es sei L eine kontextfreie Sprache. Dann gibt es ein $n \in \mathbb{N}$, so dass jeder String $s \in L$, dessen Länge größer oder gleich n ist, in der Form

$$s = uvwxy$$

geschrieben werden kann, so dass außerdem folgendes gilt:

1. $|vwx| \leq n$,

   der mittlere Teil des Strings hat also eine Länge von höchstens n Buchstaben.

2. $vx \neq \varepsilon$,

   die Teilstrings v und x können also nicht beide gleichzeitig leer sein.

3. $\forall h \in \mathbb{N} : uv^h wx^h y \in L$.

   Dieser Bedingung verdankt das Pumping-Lemma seinen Namen, denn die Strings v und x können beliebig oft *gepumpt* werden. □

Beweis: Wir zeigen den Satz zunächst für den Fall, dass die Sprache L das leere Wort nicht enthält. Dazu konstruieren wir mit dem im letzten Abschnitt gezeigten Verfahren eine Grammatik $G = \langle V, T, R, S \rangle$ in verallgemeinerter Chomsky-Normalform, so dass

$$L = L(G)$$

gilt. Wir nehmen an, dass die Grammatik G insgesamt k syntaktische Variablen enthält und außerdem die Breite b hat. Wir definieren

$$n := b^{k+2}.$$

Sei nun $s \in L$ mit $|s| \geq n$. Wir betrachten die Listen aus

$$\mathit{parseTree}(S \Rightarrow^* s).$$

Falls alle Listen hier eine Länge kleiner-gleich $k+1$ hätten, so würde aus Lemma 31 folgen, dass

$$|s| \leq b^{k+1} < n$$

gilt, im Widerspruch zu der Voraussetzung $|s| \geq n$. Also muss es in $\mathit{parseTree}(S \Rightarrow^* s)$ eine Liste geben, die mindestens die Länge $k+2$ hat. Diese Liste hat dann die Form

$$[A_1, \cdots, A_l, t] \quad \text{mit } A_i \in V \text{ für alle } i \in \{1, \cdots, l\}, \quad t \in T, \quad \text{sowie } l \geq k+1.$$

Wegen $l \geq k+1$ können nicht alle Variablen A_1, $\cdots$, A_l voneinander verschieden sein, denn es gibt ja nur insgesamt k verschiedene syntaktische Variablen. Wir finden daher in der Menge $\{l, l-1, \cdots, l-k\}$ zwei Indices i, j mit $i \neq j$ und $A_i = A_j =: A$. Ohne Beschränkung der Allgemeinheit gelte $i < j$. Die Ableitung von s hat dann die folgende Form

$$S \Rightarrow^* uA_iy \Rightarrow^* uvA_jxy \Rightarrow^* uvwxy = s.$$

Insbesondere gilt also

$$S \Rightarrow^* uAy, \quad A \Rightarrow^* vAx, \quad \text{und} \quad A \Rightarrow^* w.$$

Damit haben wir dann aber folgendes:

1. $S \Rightarrow^* uAy \Rightarrow^* uwy$, also
$$S \Rightarrow^* uv^0wx^0y.$$
2. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uvvwxxy$, also
$$S \Rightarrow^* uv^2wx^2y.$$
3. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \Rightarrow^* uv^3Ax^3y \Rightarrow^* \cdots \Rightarrow^* uv^kAx^ky \Rightarrow^* uv^kwx^ky.$

Wir müssen jetzt noch zeigen, dass $vx \neq \varepsilon$ gilt. Die Ableitung

$$A \Rightarrow^* vAx$$

ist eigentlich die Ableitung

$$A_i \Rightarrow^* vA_jx$$

und enthält daher mindestens einen Ableitungsschritt. Wir führen den Nachweis der Behauptung $vx \neq \varepsilon$ indirekt und nehmen $v = x = \varepsilon$ an. Da die Grammatik G nach Voraussetzung keine Unit-Regeln enthält, hat der erste Schritt dieser Ableitung dann die Form

$$A \Rightarrow B_1B_2 \cdots B_m \Rightarrow^* A \quad \text{mit } m \geq 2.$$

Dann muss aber für wenigstens ein $o \in \{1, \cdots, m\}$

$$B_o \Rightarrow^* \varepsilon$$

gelten und das geht nicht, da eine Grammatik in Chomsky-Normalform keine ε-Regeln enthält.

Als nächstes zeigen wir, dass die Ungleichung $|vwx| \leq n$ gilt. Wir haben

$$A = A_i \Rightarrow^* vA_jx \Rightarrow^* vwx.$$

Wegen $i \in \{l-k, l-(k-1), \cdots, l\}$, wissen wir, dass die Länge der längsten Liste in

$$\mathit{parseTree}(A \Rightarrow^* vwx)$$

kleiner-gleich $k+2$ ist. Nach dem Lemma 31 folgt damit für die Länge von vwx die Abschätzung

$$|vwx| \leq b^{k+2} = n.$$

Zum Schluß müssen wir noch festlegen, was in dem Fall zu tun ist, wenn $\varepsilon \in L$ ist. In diesem Fall betrachten wir einfach die Sprache $L' := L \backslash \{\varepsilon\}$. Diese Sprache ist kontextfrei, denn wir können aus der Grammatik der Sprache L einfach alle ε-Regeln mit dem im letzten Abschnitt gezeigten Verfahren entfernen und erhalten dann eine Grammatik, welche die Sprache L' beschreibt. Für die Sprache L' haben wir das Pumping-Lemma oben bewiesen. Wir müssen nun für das n, das wir oben finden, nur fordern, dass es größer als 0 ist, denn dann erfüllt das leere Wort die Voraussetzung des Pumping-Lemmas $|s| \geq n$ nicht und damit gilt das Pumping-Lemma dann auch für alle Wörter aus L. □

Bemerkung: Das Pumping-Lemma wird in der deutschsprachigen Literatur gelegentlich auch als "*Schleifensatz*" bezeichnet.

11.4 Anwendungen des Pumping-Lemmas

Wir zeigen nun, wie mit Hilfe des Pumping-Lemmas der Nachweis erbracht werden kann, dass bestimmte Sprachen nicht kontextfrei sind.

11.4.1 Die Sprache $L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k \mid k \in \mathbb{N}\}$

Wir weisen nun nach, dass die Sprache

$$L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k \mid k \in \mathbb{N}\}$$

nicht kontextfrei ist. Wir führen diesen Nachweis indirekt und nehmen zunächst an, dass L kontextfrei ist. Nach dem Pumping-Lemma gibt es dann eine natürliche Zahl n, so dass jeder String $s \in L$, dessen Länge größer oder gleich n ist, sich in Teilstrings der Form

$$s = uvwxy$$

zerlegen läßt, so dass außerdem folgendes gilt:

1. $|vwx| \leq n$,
2. $vx \neq \varepsilon$,
3. $\forall i \in \mathbb{N} : uv^iwx^iy \in L$.

   Insbesondere können wir hier $i = 0$ wählen und erhalten dann

   $$uwy \in L.$$

Wir definieren nun den String s wie folgt:

$$s := \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n.$$

Dieser String hat die Länge $3 \cdot n$ und erfüllt damit die Voraussetzung über die Länge. Damit finden wir also eine Zerlegung $s = uvwxy$ mit den obigen Eigenschaften. Da der Teilstring vwx eine Länge kleiner oder gleich n hat, können in diesem String nicht gleichzeitig die Buchstaben "$\mathtt{a}$" und "$\mathtt{c}$" vorkommen. Wir betrachten diese Fälle getrennt.

1. Fall: In dem String vwx kommt der Buchstabe "$\mathtt{c}$" nicht vor:

   $$\mathit{count}(vwx, \mathtt{c}) = 0$$

   Da $vx \neq \varepsilon$ ist, folgt

   $$\mathit{count}(vx, \mathtt{a}) + \mathit{count}(vx, \mathtt{b}) > 0.$$

   Wir nehmen zunächst an, dass $\mathit{count}(vx, \mathtt{a}) > 0$ gilt, der Fall $\mathit{count}(vx, \mathtt{b}) > 0$ ist analog zu behandeln. Dann erhalten wir einerseits

   $$\begin{array}{rcl} \mathit{count}(uwy, \mathtt{c}) & = & \mathit{count}(s, \mathtt{c}) - \mathit{count}(vx, \mathtt{c}) \\ & = & \mathit{count}(s, \mathtt{c}) - 0 \\ & = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n, \mathtt{c}) \\ & = & n. \end{array}$$

   Zählen wir nun die Häufigkeit, mit welcher der Buchstabe "$\mathtt{a}$" in dem String uwy auftritt, so erhalten wir

   $$\begin{array}{rcl} \mathit{count}(uwy, \mathtt{a}) & = & \mathit{count}(s, \mathtt{a}) - \mathit{count}(vx, \mathtt{a}) \\ & < & \mathit{count}(s, \mathtt{a}) \\ & = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n, \mathtt{a}) \\ & = & n. \end{array}$$

Damit haben wir dann aber

$$count(uwy, \mathtt{a}) < n = count(uwy, \mathtt{c})$$

und daraus folgt $uwy \notin L$, was im Widerspruch zum Pumping-Lemma steht.

2. Fall: In dem String vwx kommt der Buchstabe "a" nicht vor.
   Dieser Fall läßt sich analog zum ersten Fall behandeln. □

Aufgabe 17: Zeigen Sie, dass die Sprache

$$L = \left\{ \mathtt{a}^{k^2} \,\middle|\, k \in \mathbb{N} \right\}$$

nicht kontextfrei ist.

Hinweis: Argumentieren Sie über die Länge der betrachteten Strings.

Kapitel 12

Earley-Parser

In diesem Kapitel stellen wir ein effizientes Verfahren vor, mit dem es möglich ist, für eine beliebige vorgegebene kontextfreie Grammatik

$$G = \langle V, \Sigma, R, S \rangle \quad \text{und einen String } s \in \Sigma^*$$

zu entscheiden, ob s ein Element der Sprache $L(G)$ ist, ob also $s \in L(G)$ gilt. Der Algorithmus, denn wir gleich angeben werden, wurde 1970 von Jay Earley publiziert [Ear70]. Das Kapitel gliedert sich in die folgenden Abschnitte.

1. Zunächst skizzieren wir die dem Algorithmus zu Grunde liegende Theorie.
2. Danach geben wir eine einfache Implementierung des Algorithmus in *Java* an.
3. Anschließend beweisen wir die Korrektheit und Vollständigkeit des Algorithmus.
4. Zum Abschluß des Kapitels untersuchen wir die Komplexität.

12.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist.

Definition 33 (Earley-Objekt) Gegeben sei eine kontextfreie Grammatik $G = \langle V, \Sigma, R, S \rangle$ und ein String $s = x_1 x_2 \cdots x_n \in \Sigma^*$ der Länge n. Wir bezeichnen ein Paar der Form

$$\langle A \to \alpha \bullet \beta, k \rangle$$

dann als ein *Earley-Objekt*, falls folgendes gilt:

1. $(A \to \alpha\beta) \in R$ und
2. $k \in \{0, 1, \cdots, n\}$. □

Erklärung: Ein Earley-Objekt beschreibt einen Zustand, in dem ein Parser sich befinden kann. Ein Earley-Parser, der einen String $x_1 \cdots x_n$ parsen soll, verwaltet $n + 1$ Mengen von Earley-Objekten. Diese Mengen bezeichnen wir mit

$$Q_0, Q_1, \cdots, Q_n.$$

Die Interpretation von

$$\langle A \to \alpha \bullet \beta, k \rangle \in Q_j \quad \text{mit } j \geq k$$

ist dann wie folgt:

1. Der Parser versucht die Regel $A \to \alpha\beta$ auf den Teilstring $x_{k+1} \cdots x_n$ anzuwenden und am Anfang dieses Teilstrings ein A mit Hilfe der Regel $A \to \alpha\beta$ zu erkennen.

2. Am Anfang des Teilstrings $x_{k+1} \cdots x_j$ hat der Parser bereits α erkannt, es gilt also

   $$\alpha \Rightarrow^* x_{k+1} \cdots x_j.$$

3. Folglich versucht der Parser am Anfang des Teilstrings $x_{j+1} \cdots x_n$ ein β erkennen.

Der Algorithmus von Earley verwaltet für $i = 0, 1, \cdots, n$ Mengen Q_j von Earley-Objekten, die den Zustand beschreiben, in dem der Parser ist, wenn der Teilstring $x_1 \cdots x_j$ verarbeitet ist. Zu Beginn des Algorithmus wird der Grammatik ein neues Start-Symbol $\widehat{S}$ sowie die Regel $\widehat{S} \to S$ hinzugefügt. Die Menge Q_0 wird definiert als

$$Q_0 := \big\{\langle \widehat{S} \to \bullet S, 0\rangle\big\},$$

denn der Parser soll ja das Start-Symbol S am Anfang des Strings $x_1 \cdots x_n$ erkennen. Die restlichen Mengen Q_j sind für $j = 1, \cdots, n$ zunächst leer. Die Mengen Q_j werden nun durch die folgende drei Operationen so lange wie möglich erweitert:

1. *Lese-Operation*

   Falls der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta \bullet a\gamma, k\rangle$ enthält, wobei a ein Terminal ist, so versucht der Parser, die rechte Seite der Regel $A \to \beta a\gamma$ zu erkennen und hat bis zur Position j bereits den Teil β erkannt. Folgt auf dieses β nun, wie in der Regel $A \to \beta a\gamma$ vorgesehen, an der Position $j + 1$ das Terminal a, so muss der Parser nach der Position $j + 1$ nur noch γ erkennen. Daher wird in diesem Fall das Earley-Objekt

   $$\langle A \to \beta a \bullet \gamma, k\rangle$$

   dem Zustand Q_{j+1} hinzugefügt:

   $$\langle A \to \beta \bullet a\gamma, k\rangle \in Q_j \wedge x_{j+1} = a \;\Rightarrow\; Q_{j+1} := Q_{j+1} \cup \big\{\langle A \to \beta a \bullet \gamma, k\rangle\big\}.$$

2. *Vorhersage-Operation*

   Falls der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta \bullet C\delta, k\rangle$ enthält, wobei C eine syntaktische Variable ist, so versucht der Parser im Zustand Q_j den Teilstring $C\delta$ zu erkennen. Dazu muss der Parser an diesem Punkt ein C erkennen. Wir fügen daher für jede Regel $C \to \gamma$ der Grammatik das Earley-Objekt $\langle C \to \bullet\gamma, j\rangle$ zu der Menge Q_j hinzu:

   $$\langle A \to \beta \bullet C\delta, k\rangle \in Q_j \wedge (C \to \gamma) \in R \;\Rightarrow\; Q_j := Q_j \cup \big\{\langle C \to \bullet\gamma, j\rangle\big\}.$$

3. *Vervollständigungs-Operation*

   Falls der Zustand Q_i ein Earley-Objekt der Form $\langle C \to \gamma\bullet, j\rangle$ enthält und weiter der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta \bullet C\delta, k\rangle$ enthält, dann hat der Parser im Zustand Q_j versucht, ein C zu parsen und das C ist im Zustand Q_i erkannt worden. Daher fügen wir dem Zustand Q_i nun das Earley-Objekt $\langle A \to \beta C \bullet \delta, k\rangle$ hinzu:

   $$\langle C \to \gamma\bullet, j\rangle \in Q_i \wedge \langle A \to \beta \bullet C\delta, k\rangle \in Q_j \;\Rightarrow\; Q_i := Q_i \cup \big\{\langle A \to \beta C \bullet \delta, k\rangle\big\}.$$

Der Algorithmus von Earley um einen String der Form $s = x_1 \cdots x_n$ zu parsen funktioniert so:

1. Wir initialisieren die Zustände Q_i wie folgt:

   $$Q_0 := \big\{\langle \widehat{S} \to \bullet S, 0\rangle\big\},$$
   $$Q_i := \big\{\big\} \quad \text{für } i = 1, \cdots, n.$$

2. Anschließend lassen wir in einer Schleife i von 0 bis n laufen und führen die folgenden Schritte durch:

   (a) Wir vergrößern Q_i mit der Vervollständigungs-Operation so lange, bis mit dieser Operation keine neuen Earley-Objekte mehr gefunden werden können.

(b) Anschließend vergrößern wir Q_i mit Hilfe der Vorhersage-Operation. Diese Operation wird ebenfalls so lange durchgeführt, wie neue Earley-Objekte gefunden werden.

(c) Falls $i < n$ ist, wenden wir die Lese-Operation auf Q_i an und initialisierend damit Q_{i+1}.

Falls die betrachtete Grammatik G auch ε-Regeln enthält, also Regeln der Form

$$C \to \varepsilon,$$

dann kann es passieren, dass durch die Anwendung einer Vorhersage-Operation eine neue Anwendung der Vervollständigungs-Operation möglich wird. In diesem Fall müssen Vorhersage-Operation und Vervollständigungs-Operation so lange iteriert werden, bis durch Anwendung dieser beiden Operationen keine neuen Earley-Objekte mehr erzeugt werden können.

3. Falls nach Beendigung des Algorithmus die Menge Q_n das Earley-Objekt $\langle \widehat{S} \to S\bullet, 0\rangle$ enthält, dann war das Parsen erfolgreich und der String $x_1 \cdots x_n$ liegt in der von der Grammatik erzeugten Sprache.

Beispiel: Abbildung 12.1 zeigt eine vereinfachte Grammatik für arithmetische Ausdrücke, die nur aus den Zahlen "1", "2" und "3" und den beiden Operator-Symbolen "+" und "*" aufgebaut sind. Die Menge T der Terminale dieser Grammatik ist also durch

$$T = \{\ \text{“1”}\ ,\ \text{“2”}\ ,\ \text{“3”}\ ,\ \text{“+”}\ ,\ \text{“*”}\ \}$$

gegeben. Wie zeigen, wie sich der String "1+2*3" mit dieser Grammatik und dem Algorithmus von Earley parsen läßt.

$$\begin{array}{lcl} E & \to & E \text{ “+” } P \mid P \\ P & \to & P \text{ “*” } F \mid F \\ F & \to & \text{“1”} \mid \text{“2”} \mid \text{“3”} \end{array}$$

Abbildung 12.1: Eine vereinfachte Grammatik für arithmetische Ausdrücke.

1. Wir initialisieren Q_0 als

$$Q_0 = \{\langle \widehat{S} \to \bullet\, E, 0\rangle\}.$$

Die Mengen Q_1, Q_2, Q_3, Q_4 und Q_5 sind zunächst alle leer. Wenden wir die Vervollständigungs-Operation auf Q_0 an, so finden wir keine neuen Earley-Objekte.

Anschließend wenden wir die Vorhersage-Operation auf das Earley-Objekt $\langle \widehat{S} \to \bullet\, E, 0\rangle$ an. Dadurch werden der Menge Q_0 zunächst die beiden Earley-Objekte

$$\langle E \to \bullet\, E \text{“+”} P, 0\rangle \quad \text{und} \quad \langle E \to \bullet\, P, 0\rangle$$

hinzugefügt. Auf das Earley-Objekt $\langle E \to \bullet\, P, 0\rangle$ können wir die Vorhersage-Operation ein weiteres Mal anwenden und erhalten dann die beiden neuen Earley-Objekte

$$\langle P \to \bullet\, P \text{“+”} F, 0\rangle \quad \text{und} \quad \langle P \to \bullet\, F, 0\rangle.$$

Wenden wir auf das Earley-Objekt $\langle P \to \bullet\, F, 0\rangle$ die Vorhersage-Operation an, so erhalten wir schießlich noch die folgenden Earley-Objekte in Q_0:

$$\langle F \to \bullet \text{“1”}, 0\rangle, \quad \langle F \to \bullet \text{“2”}, 0\rangle, \quad \text{und} \quad \langle F \to \bullet \text{“3”}, 0\rangle.$$

Insgesamt enthält Q_0 nun die folgenden Earley-Objekte:

(a) $\langle \widehat{S} \to \bullet\, E, 0\rangle$,

(b) $\langle E \to \bullet\ E \text{“+”} P, 0\rangle$

(c) $\langle E \to \bullet\ P, 0\rangle$,

(d) $\langle P \to \bullet\ P \text{“*”} F, 0\rangle$,

(e) $\langle P \to \bullet\ F, 0\rangle$,

(f) $\langle F \to \bullet \text{“1”}, 0\rangle$,

(g) $\langle F \to \bullet \text{“2”}, 0\rangle$,

(h) $\langle F \to \bullet \text{“3”}, 0\rangle$.

Jetzt wenden wir die Lese-Operation auf Q_0 an. Da das erste Zeichen des zu parsenden Strings eine “1” ist, hat die Menge Q_1 danach die folgende Form:

$$Q_1 = \big\{\langle F \to \text{“1”}\bullet, 0\rangle\big\}.$$

2. Nun setzen wir $i = 1$ und wenden zunächst auf Q_1 die Vervollständigungs-Operation an. Aufgrund des Earley-Objekts $\langle F \to \text{“1”}\bullet, 0\rangle$ in Q_1 suchen wie in Q_0 ein Earley-Objekt, bei dem die Markierung “•” vor der Variablen F steht. Wir finden das Earley-Objekt $\langle P \to \bullet\ F, 0\rangle$. Daher fügen wir nun Q_1 das Earley-Objekt

   $$\langle P \to F\ \bullet, 0\rangle$$

   hinzu. Hierauf können wir wieder die Vervollständigungs-Operation anwenden und finden (nach mehrmaliger Anwendung der Vervollständigungs-Operation) für Q_1 insgesamt die folgenden Earley-Objekte durch Vervollständigung:

   (a) $\langle P \to F\ \bullet, 0\rangle$,

   (b) $\langle P \to P\ \bullet\ \text{“*”} F, 0\rangle$,

   (c) $\langle E \to P\ \bullet, 0\rangle$,

   (d) $\langle E \to E\ \bullet\ \text{“+”} P, 0\rangle$,

   (e) $\langle \widehat{S} \to E\ \bullet, 0\rangle$.

   Als nächstes wenden wir auf diese Earley-Objekte die Vorhersage-Operation an. Da das Markierungs-Zeichen “•” aber in keinem der in Q_i auftretenden Earley-Objekte vor einer Variablen steht, ergeben sich hierbei keine neuen Earley-Objekte.

   Als letztes wenden wir die Lese-Operation auf Q_1 an. Da in dem String “1+2*3” das Zeichen “+” an der Position 2 liegt ist und Q_1 das Earley-Objekt

   $$\langle E \to E\ \bullet\ \text{“+”} P, 0\rangle$$

   enthält, fügen wir in Q_2 das Earley-Objekt

   $$\langle E \to E \text{“+”}\ \bullet\ P, 0\rangle$$

   ein.

3. Nun setzen wir $i = 2$ und wenden zunächst auf Q_2 die Vervollständigungs-Operation an. Zu diesem Zeitpunkt gilt

   $$Q_2 = \{\langle E \to E \text{“+”}\ \bullet\ P, 0\rangle\}.$$

   Da in dem einzigen Earley-Objekt, das hier auftritt, das Markierungs-Zeichen “•” nicht am Ende der Grammatik-Regel steht, finden wir durch die Vervollständigungs-Operation in diesem Schritt keine neuen Earley-Objekte.

   Als nächstes wenden wir auf Q_2 die Vorhersage-Operation an. Da das Markierungs-Zeichen vor der Variablen P steht, finden wir zunächst die beiden Earley-Objekte

   $$\langle P \to \bullet\ F, 2\rangle \quad \text{und} \quad \langle P \to \bullet\ P \text{“*”} F, 2\rangle.$$

   Da in dem ersten Earley-Objekte das Markierungs-Zeichen vor der Variablen F steht, kann

die Vorhersage-Operation ein weiteres Mal angewendet werden und wir finden noch die folgenden Earley-Objekte:

(a) $\langle F \to \bullet \text{“}\texttt{1}\text{”}, 2\rangle$,

(b) $\langle F \to \bullet \text{“}\texttt{2}\text{”}, 2\rangle$,

(c) $\langle F \to \bullet \text{“}\texttt{3}\text{”}, 2\rangle$.

Als letztes wenden wir die Lese-Operation auf Q_2 an. Da das dritte Zeichen in dem zu lesenden String “1+2*3” die Ziffer “2” ist, hat Q_3 nun die Form

$$Q_3 = \{\langle F \to \text{“}\texttt{2}\text{”}\bullet, 2\rangle\}.$$

4. Wir setzen $i = 3$ und wenden auf Q_3 die Vervollständigungs-Operation an. Dadurch fügen wir
$$\langle P \to F \bullet, 2\rangle$$
in Q_3 ein. Hier können wir eine weiteres Mal die Vervollständigungs-Operation anwenden. Durch iterierte Anwendung der Vervollständigungs-Operation erhalten wir zusätzlich die folgenden Earley-Objekte:

   (a) $\langle P \to P \bullet \text{“}\texttt{*}\text{”} F, 2\rangle$,

   (b) $\langle E \to E \text{“}\texttt{+}\text{”} P \bullet, 0\rangle$,

   (c) $\langle E \to E \bullet \text{“}\texttt{+}\text{”} P, 0\rangle$

   (d) $\langle \widehat{S} \to E \bullet, 0\rangle$.

   Als letztes wenden wir die Lese-Operation an. Da der nächste zu lesende Buchstabe das Zeichen “*” ist, erhalten wir

   $$Q_4 = \{\langle P \to P \text{“}\texttt{*}\text{”} \bullet F, 2\rangle\}.$$

5. Wir setzen $i = 4$. Die Vervollständigungs-Operation liefert keine neuen Earley-Objekte. Die Vorhersage-Operation liefert folgende Earley-Objekte:

   (a) $\langle F \to \bullet \text{“}\texttt{1}\text{”}, 4\rangle$,

   (b) $\langle F \to \bullet \text{“}\texttt{2}\text{”}, 4\rangle$,

   (c) $\langle F \to \bullet \text{“}\texttt{3}\text{”}, 4\rangle$.

   Da das nächste Zeichen die Ziffer “3” ist, liefert die Lese-Operation für Q_5:

   $$Q_5 = \langle F \to \text{“}\texttt{3}\text{”}\bullet, 4\rangle\}.$$

6. Wir setzen $i = 5$. Die Vervollständigungs-Operation liefert nacheinander die folgenden Earley-Objekte:

   (a) $\langle P \to P \text{“}\texttt{*}\text{”} F \bullet, 2\rangle$,

   (b) $\langle E \to E \text{“}\texttt{+}\text{”} P \bullet, 0\rangle$,

   (c) $\langle P \to P \bullet \text{“}\texttt{*}\text{”} F, 2\rangle$,

   (d) $\langle E \to E \bullet \text{“}\texttt{+}\text{”} P, 0\rangle$

   (e) $\langle \widehat{S} \to E \bullet, 0\rangle$.

Da die Menge Q_5 das Earley-Objekt $\langle \widehat{S} \to E \bullet, 0\rangle$ enthält, können wir schließen, dass der String “1+2*3” tatsächlich in der von der Grammatik erzeugten Sprache liegt.

Aufgabe: Zeigen Sie, dass der String “1*2+3” in der Sprache der von der in Abbildung 12.1 angegebenen Grammatik liegt.

12.2 Implementierung

Im folgenden präsentieren wir ein einfache Implementierung des von Earley angegebenen Algorithmus. Abbildung 12.2 auf Seite 163 zeigt die Klasse `EarleyParser`. Diese Klasse enthält vier Member-Variablen:

1. `mGrammar` ist die Grammatik, mit der geparst werden soll.
2. `mString` ist der zu parsende String.

   Um den Parser möglichst einfach zu halten, wird der hier entwicklte Parser nicht auf einen Scanner zurückgreifen, sondern die Terminale unserer Grammatik sind unmittelbar die Buchstaben, aus denen der String besteht.
3. `mLength` gibt die Anzahl der Buchstaben des zu parsenden Strings an.
4. `mStateList` ist die Liste $[Q_0, Q_1, \cdots, Q_n]$.

```
1  public class EarleyParser {
2
3      private Grammar mGrammar;
4      private String  mString;
5      private int     mLength;
6
7      private List<Set<EarleyItem>> mStateList;
8
9      public EarleyParser(Grammar grammar, String string) {
10         mGrammar = grammar;
11         mString  = string;
12         mLength  = mString.length();
13         mStateList = new ArrayList<Set<EarleyItem>>(mLength + 1);
14         for (int i = 0; i <= mLength; ++i) {
15             Set<EarleyItem> qi = new TreeSet<EarleyItem>();
16             if (i == 0) {
17                 EarleyItem start = mGrammar.startItem();
18                 qi.add(start);
19             }
20             mStateList.add(qi);
21         }
22         parse();
23     }
24     ...
25 }
```

Abbildung 12.2: Struktur der Klasse `EarleyParser`

Abbildung 12.2 zeigt zunächst nur den Konstruktor der Klasse `EarleyParser`, der als Argumente die Grammatik und den zu parsenden String erhält. Die wesentliche Aufgabe dieses Konstruktors ist die Initialisierung der Mengen Q_i. Zunächst werden diese alle als leere Mengen angelegt. Außerdem wird in Q_0 noch das Earley-Objekt

$$\langle \widehat{S} \rightarrow \bullet S, 0 \rangle$$

abgelegt, das von der Methode `startItem`() berechnet wird. Der Rest der Arbeit wird an die Methode `parse`() delegiert, die in Abbildung 12.3 auf Seite 164 gezeigt ist. Die äußere `for`-Schleife ite-

riert über die Mengen $Q_0, Q_1, \cdots, Q_n$. Für ein festes i werden anschließend die Vervollständigungs-Operation (Methode `complete`()) und die Vorhersage-Operation (Methode *predict*()) so lange ausgeführt, bis sich die Menge Q_i durch diese Operationen nicht mehr vergrößert.[1] Anschließend wird von der Methode *scan*() die Lese-Operation durchgeführt, welche die Menge Q_{i+1} initialisiert.

```
1  void parse() {
2      for (int i = 0; i <= mLength; ++i) {
3          boolean change = false;
4          do {
5              change = complete(i);
6              change = change || predict(i);
7          } while (change);
8          scan(i);
9      }
10 }
```

Abbildung 12.3: Implementierung der Methode *parse*()

```
1  boolean complete(int i) {
2      boolean change = false;
3      Set<EarleyItem> Qi = mStateList.get(i);
4      boolean added = false;
5      do {
6          Set<EarleyItem> newQi = new TreeSet<EarleyItem>();
7          for (EarleyItem item: Qi) {
8              if (item.isComplete()) {
9                  Variable C = item.getVariable();
10                 int      j = item.getIndex();
11                 Set<EarleyItem> Qj = mStateList.get(j);
12                 for (EarleyItem cItem: Qj) {
13                     if (cItem.sameVar(C)) {
14                         EarleyItem moved = cItem.moveDot();
15                         newQi.add(moved);
16                     }
17                 }
18             }
19         }
20         added = Qi.addAll(newQi);
21         change = change || added;
22     } while (added);
23     return change;
24 }
```

Abbildung 12.4: Impementierung der Vervollständigungs-Operation

Abbildung 12.4 auf Seite 164 zeigt die Implementierung der Vervollständigungs-Operation. Das Argument i gibt an, welche der Mengen Q_i vervollständigt werden soll. Die Methode gibt als

[1] Wenn die Grammatik keine ε-Regeln (das sind Regeln der Form $A \to \varepsilon$) enthält, dann können durch die Anwendung der Vorhersage-Operation keine neuen Anwendungen der Vervollständigungs-Operation möglich werden, so dass wir uns in diesem Fall die `do-while`-Schleife sparen könnten. Eine effizientere Implementierung würde vorher prüfen, ob die Grammatik ε-Regeln enthält und gegebenenfalls diese Schleife einsparen.

Ergebnis einen Wahrheitswert zurück. Dieser ist `true` falls bei der Vervollständigungs-Operation neue Earley-Objekte in die Menge Q_i eingefügt werden. Bleibt die Menge Q_i bei der Operation unverändert, so wird `false` zurück gegeben. Die Vervollständigungs-Operation war im letzten Abschnitt wie folgt definiert worden:

$$\langle C \to \gamma\bullet, j\rangle \in Q_i \wedge \langle A \to \beta \bullet C\delta, k\rangle \in Q_j \;\Rightarrow\; Q_i := Q_i \cup \left\{\langle A \to \beta C \bullet \delta, k\rangle\right\}$$

Die Umsetzung dieser Formel ist nun wie folgt:

1. Die Variable i wird als Parameter `i` übergeben.

2. In der Variablen `change` speichern wir den Rückgabewert der Methode. Dieser Rückgabewert soll genau dann `true` sein, wenn bei der Vervollständigungs-Operation neue Earley-Objekte in die Menge Q_i eingefügt werden. Diese Variable wird daher zunächst mit `false` initialisiert, denn am Anfang haben wir ja noch keine neuen Earley-Objekte gefunden.

3. Wird die Vervollständigungs-Operation einmal ausgeführt und werden dabei neue Earley-Objekte in die Menge Q_i eingefügt, so kann es sein, dass auf die neu eingefügten Earley-Objekte wiederum die Vervollständigungs-Operation angewendet werden kann. Daher wird die Vervollständigungs-Operation innerhalb einer `do-while`-Schleife ausgeführt, die solange läuft, wie durch die Vervollständigungs-Operation neue Earley-Objekte generiert werden. Diese Schleife wird durch die Variable `added` gesteuert: Anfangs wird diese Variable mit `false` initialisiert. Falls die Vervollständigungs-Operation ein neues Earley-Objekt in die Menge Q_i einfügt, dann wird die Variable `added` auf `true` gesetzt.

4. Innerhalb der `do-while`-Schleife initialisieren wir zunächst die Variable `newQi` als leere Menge. In dieser Variablen sammeln wir alle Earley-Objekte auf, die bei der Vervollständigungs-Operation gefunden werden.

5. Nach der Initialisierung von `newQi` iteriert in Zeile 7 - 19 eine `for`-Schleife über alle Earley-Objekte der Menge `Qi`:

   (a) Zunächst wird mit Hilfe der Methode *isComplete*() überprüft, ob bei dem Earley-Objekt das Markierungs-Zeichen "•" am Ende der Grammatik-Regel steht, ob das Earley-Objekt also die Form

   $$\langle C \to \gamma \bullet, j\rangle$$

   hat.

   (b) Wenn dies der Fall ist, liefert der Aufruf `item.getVariable()` als Ergebnis die Variable C und `item.getIndex()` liefert den zugehörigen Index j.

   (c) Anschließend muss überprüft werden, ob die Menge Q_j ein Earley-Objekt enthält, bei dem das Markierungs-Zeichen vor der Variablen C steht. Ein solches Earley-Objekt hat dann die Form

   $$\langle A \to \beta \bullet C\delta, k\rangle. \qquad (*)$$

   Diese Überprüfung wird von der inneren `for`-Schleife durchgeführt, die über alle Earley-Objekte der Menge Q_j iteriert. Der Aufruf `cItem.sameVar(C)` überprüft, ob das Earley-Objekt `cItem` die in $(*)$ angegebene Form hat, ob also das Markierungs-Zeichen • vor einer Variablen steht, die mit der Variablen C identisch ist.

   (d) Wenn dies der Fall ist, berechnet der Aufruf `cItem.moveDot()` das Earley-Objekt

   $$\langle A \to \beta C \bullet \delta, k\rangle,$$

   das anschließend in Zeile 20 der Menge `newQi` hinzugefügt wird.

6. Die Menge der berechneten neuen Earley-Objekte wird durch den Aufruf `Qi.addAll(newQi)` der Menge Q_i hinzugefügt. Eventuell sind die so berechneten Earley-Objekte schon vorher

Elemente der Menge Q_i gewesen. In diesem Fall liefert der Aufruf `Qi.addAll(newQi)` als Ergebnis `false`, weil dann der Menge Q_i keine neuen Elemente hinzugefügt werden. Andernfalls liefert der Aufruf `true`, was zur Folge hat, dass die `do-while`-Schleife ein weiteres Mal durchlaufen wird.

```
1  boolean predict(int i) {
2      boolean change = false;
3      Set<EarleyItem> Qi = mStateList.get(i);
4      boolean added = false;
5      do {
6          Set<EarleyItem> newQi = new TreeSet<EarleyItem>();
7          for (EarleyItem item: Qi) {
8              Variable C = item.nextVar();
9              if (C != null) {
10                 for (Rule rule: mGrammar.getRules()) {
11                     if (C.equals(rule.getVariable())) {
12                         Set<EarleyItem> items = rule.generateRules(i);
13                         newQi.addAll(items);
14                     }
15                 }
16             }
17         }
18         added = Qi.addAll(newQi);
19         change = change || added;
20     } while (added);
21     return change;
22 }
```

Abbildung 12.5: Implementierung der Vorhersage-Operation

Abbildung 12.5 auf Seite 166 zeigt die Methode `predict()`, welche die Vorhersage-Operation implementiert. Diese Operation hatten wir im letzten Abschnitt durch die folgende Formel spezifiziert:

$$\langle A \to \beta \bullet C\delta, k\rangle \in Q_i \wedge (C \to \gamma) \in R \;\Rightarrow\; Q_i := Q_i \cup \left\{\langle C \to \bullet\gamma, i\rangle\right\}$$

Wir diskutieren nun die Details der Umsetzung dieser Formel in der Methode `predict(i)`.

1. Der Parameter `i` gibt wieder an, auf welche der Mengen Q_i die Operation angewendet werden soll.

2. Die Methode liefert als Ergebnis `true` zurück, wenn sich die Menge Q_i durch die Anwendung dieser Operation tatsächlich vergrößert hat, sonst wird `false` zurück gegeben. Dieser Rückgabewert wird in der Variablen `change` gespeichert.

3. Da auf die Earley-Objekte, die bei der Vorhersage-Operation gefunden werden, unter Umständen ebenfalls die Vorhersage angewendet werden kann, findet die eigentliche Anwendung der Vorhersage-Operation innerhalb einer `do-while`-Schleife statt, die durch die Variable `added` gesteuert wird. Diese Variable wird mit `false` initialisiert. Falls durch die Vorhersage-Operation neue Earley-Objekte gefunden werden, wird `added` auf `true` gesetzt, so dass die Schleife dann ein weiteres Mal durchlaufen wird.

4. Zur Durchführung der Vorhersage-Operation benötigen wir zunächst eine Schleife, die über alle Earley-Objekte der Menge Q_i iteriert und überprüft, ob diese Objekte die Form

$$\langle A \to \beta \bullet C\delta, k \rangle$$

mit $C \in V$ haben, so dass sich die Vorhersage-Operation anwenden läßt.

5. Die Durchführung der Vorhersage-Operation startet mit dem Aufruf `item.nextVar()`. Falls nun `item` ein Earley-Objekt der Form $\langle A \to \beta \bullet C\delta, k \rangle$ ist, bei dem also das Markierungs-Zeichen vor einer Variablen steht, dann liefert die Methode *nextVar*() als Ergebnis diese Variable C als Ergebnis, andernfalls wird `null` zurück gegeben.

6. Wenn eine Variable C hinter dem Markierungs-Zeichen gefunden wird, werden anschließend in der `for`-Schleife alle die Regeln der Form $C \to \gamma$, also alle Regeln, bei denen die Variable C auf der linken Seite des Pfeils steht, als Earley-Objekte der Form

   $$\langle C \to \bullet\, \gamma, i \rangle$$

   in der Menge `newQi` aufgesammelt.

   Es ist offensichtlich, dass diese Operation iteriert werden muss, denn γ kann ja seinerseits mit einer Variablen beginnen.

7. Die gefundenen Earley-Objekte werden in Zeile 18 der Menge Q_i hinzugefügt. Falls diese Objekte vorher noch nicht in der Menge Q_i enthalten waren, liefert die Methode `addAll()` als Ergebnis `true` und damit wird die `do-while`-Schleife ein weiteres Mal durchlaufen.

```
1   void scan(int i) {
2       Set<EarleyItem> Qi = mStateList.get(i);
3       if (i < mLength) {
4           char a = mString.charAt(i);
5           for (EarleyItem item: Qi) {
6               if (item.scan(a)) {
7                   EarleyItem next = item.moveDot();
8                   Set<EarleyItem> Qiplus1 = mStateList.get(i + 1);
9                   Qiplus1.add(next);
10              }
11          }
12      }
13  }
```

Abbildung 12.6: Implementierung der Lese-Operation

Abbildung 12.6 zeigt die Implementierung der Lese-Operation. Wir hatten diese Operation wie folgt spezifiziert:

$$\langle A \to \beta \bullet a\gamma, k \rangle \in Q_i \wedge x_{i+1} = a \;\Rightarrow\; Q_{i+1} := Q_{i+1} \cup \big\{ \langle A \to \beta a \bullet \gamma, k \rangle \big\}.$$

Die Umsetzung dieser Spezifkation in der Methode `scan(i)` diskutieren wir jetzt.

1. Das Argument gibt an, auf welche Menge Q_i die Operation angewendet werden soll.

2. Wenn der zu parsende String insgesamt n Zeichen enthält, dann kann die Lese-Operation offenbar nur für $i < n$ angewendet werden, denn bei der Anwendung dieser Operation wird das $(i+1)$-te Zeichen der Eingabe gelesen und das wäre für $i = n$ nicht definiert. Das erklärt den Test in Zeile 3.

3. Der Aufruf `mString.charAt(i)` liefert das $(i+1)$-te Zeichen der Eingabe, denn wir starten unsere Numerierung der einzelnen Zeichen mit 0. In der Notation der Spezifkation der Operation gilt also

   `mString.charAt(i)` $= x_{i+1}$.

4. Ist dieses Zeichen a, so werden in der `for`-Schleife alle Earley-Objekte der Menge Q_i darauf überprüft, ob sie die Form

   $$\langle A \to \beta \bullet a\,\gamma, k\rangle$$

   haben, ob also dem Markierungs-Zeichen das eben gelesene Zeichen a folgt. Diese Überprüfung wird von dem Aufruf `item.scan(a)` durchgeführt.

5. Ist diese Überprüfung erfolgreich, so wird das Markierungs-Zeichen durch den Aufruf `item.moveDot()` mit dem gelesenen Zeichen a vertauscht und das so entstandene Earley-Objekt

   $$\langle A \to \beta\, a \bullet \gamma, k\rangle$$

   wird der Menge Q_{i+1} hinzugefügt.

```
1   public class EarleyItem implements Comparable {
2       Variable mVariable;
3       Body     mAlpha;
4       Body     mBeta;
5       Integer  mIndex;
6
7       public EarleyItem(Variable variable, Body alpha, Body beta,
8                         Integer index)
9       {
10          mVariable = variable;
11          mAlpha    = alpha;
12          mBeta     = beta;
13          mIndex    = index;
14      }
15      public int compareTo(Object rhs) {
16          EarleyItem rhsItem = (EarleyItem) rhs;
17          int result = mVariable.compareTo(rhsItem.getVariable());
18          if (result != 0) {
19              return result;
20          }
21          result = mAlpha.compareTo(rhsItem.getAlpha());
22          if (result != 0) {
23              return result;
24          }
25          result = mBeta.compareTo(rhsItem.getBeta());
26          if (result != 0) {
27              return result;
28          }
29          return mIndex.compareTo(rhsItem.getIndex());
30      }
```

Abbildung 12.7: Die Klasse `EarleyItem`, erster Teil.

Die Abbildungen 12.7 und 12.8 zeigen die Implementierung der Klasse `EarleyItem`. Diese Klasse stellt ein Earley-Objekt der Form

$$\langle A \Rightarrow \alpha \bullet \beta, k\rangle$$

dar. Die Komponenten A, α, β und k werden dabei als die Member-Variablen `mVariable`, `mAlpha`, `mBeta` und `mIndex` abgespeichert. Der Konstruktor initialisiert diese Variablen. Die Methode

`compareTo()` ist notwendig, damit wir Mengen von Earley-Objekte bilden können, denn wir wollen die *Java*-Klasse `TreeSet` benutzen, bei der Mengen intern durch geordnete binäre Bäume repräsentiert werden. Die Verwendung von `TreeSet<`T `>` setzt voraus, dass die Klasse T die Methode `compareTo` definiert und dass diese Methode eine totale Ordnung erzeugt.

Die von uns implementierte Methode `compareTo()` vergleicht die Objekte lexikografisch: Zunächst werden die Variablen verglichen. Sind diese unterschiedlich, so ist das Ergebnis durch den Vergleich dieser Variablen gegeben. Anschließend werden der Reihe nach die Komponenten α, β und zum Schluß der Index k vergleichen. Die erste Komponente, bei der sich die zu unterscheidenden Earley-Objekte unterscheiden, bestimmt den Unterschied.

Die restlichen Methoden funktionieren wie folgt:

1. Die Methode *isComplete*() überprüft, ob das Earley-Objekt die Form

   $$\langle A \Rightarrow \alpha \bullet \varepsilon, k \rangle$$

   hat. Dies ist genau dann der Fall, wenn `mBeta` die Länge 0 hat.

2. Die Methode *sameVar*(C) überprüft, ob das Earley-Objekt die Form

   $$\langle A \to \alpha \bullet C\, \beta, k \rangle$$

   hat, ob also hinter dem "$\bullet$" die Variable C steht.

3. Die Methode *scan*(a) prüft, ob das Earley-Objekt von der Form

   $$\langle A \to \alpha \bullet a\, \beta, k \rangle$$

   ist, ob also hinter dem Markierungs-Zeichen das Zeichen a folgt.

4. Die Methode *nextVar*() prüft, ob das Earley-Objekt die Form

   $$\langle A \to \alpha \bullet C\delta, k \rangle$$

   hat. Es wird also überprüft, ob auf das Markierungs-Zeichen eine Variable folgt. Gegebenenfalls wird diese zurück gegeben.

5. Die Methode *moveDot*() bewegt das Markierungs-Zeichen um eine Stelle nach rechts, ein Earley-Objekt der Form

   $$\langle A \to \alpha \bullet X\delta, k \rangle$$

   wird also zu

   $$\langle A \to \alpha X \bullet \delta, k \rangle$$

   transformiert. Dabei kann X sowohl für eine Variable als auch für ein Literal stehen.

   Der Aufruf der Methode `moveDot`() für ein Earley-Objekt $\langle A \to \alpha \bullet \beta, k \rangle$ setzt voraus, dass $\beta \neq \varepsilon$ ist.

Zusätzlich beinhaltet die Klasse `EarleyItem` noch Getter-Methoden, mit denen von außen auf die Member-Variablen zugegriffen werden kann. Da die Implementierung dieser Methoden trivial ist, geben wir diese aus Platzgründen nicht an.

Die restlichen bei der Implementierung verwendeten Klassen sind analog zu den Klassen aufgebaut, die wir bei der Implementierung der Aufgabe zur Transformation einer Grammatik in eine HTML-Datei verwendet haben und werden daher hier nicht weiter diskutiert. Abbildung 12.9 zeigt die von unserem Earley-Parser berechneten Mengen für die Eingabe `1+2*3`, falls die in Abbildung 12.1 gezeigte Grammatik verwendet wird.

```
31      public Boolean isComplete() {
32          return getBeta().getItemList().size() == 0;
33      }
34      public Boolean sameVar(Variable C) {
35          if (mBeta.getItemList().size() > 0) {
36              Item first = mBeta.getItemList().get(0);
37              if (first instanceof Variable) {
38                  Variable v = (Variable) first;
39                  return v.equals(C);
40              }
41          }
42          return false;
43      }
44      public Boolean scan(char a) {
45          if (mBeta.getItemList().size() > 0) {
46              Item first = mBeta.getItemList().get(0);
47              if (first instanceof Literal) {
48                  Literal v = (Literal) first;
49                  return a == v.getChar();
50              }
51          }
52          return false;
53      }
54      public Variable nextVar() {
55          if (mBeta.getItemList().size() > 0) {
56              Item first = mBeta.getItemList().get(0);
57              if (first instanceof Variable) {
58                  return (Variable) first;
59              }
60          }
61          return null;
62      }
63      public EarleyItem moveDot() {
64          List<Item> alphaList = new ArrayList<Item>();
65          List<Item> betaList  = new ArrayList<Item>();
66          alphaList.addAll(mAlpha.getItemList());
67          betaList .addAll(mBeta .getItemList());
68          Item next = betaList.get(0);
69          alphaList.add(next);
70          betaList.remove(0);
71          Body alpha = new Body(alphaList);
72          Body beta  = new Body(betaList);
73          return new EarleyItem(mVariable, alpha, beta, mIndex);
74      }
75  }
```

Abbildung 12.8: Die Klasse `EarleyItem`, zweiter Teil.

```
Q0:
<$Start -> (*) expr, 0>
<expr -> (*) product, 0>
<expr -> (*) expr '+' product, 0>
<factor -> (*) '1', 0>
<factor -> (*) '2', 0>
<factor -> (*) '3', 0>
<product -> (*) factor, 0>
<product -> (*) product '*' factor, 0>

Q1:
<$Start ->  expr(*), 0>
<expr ->  expr(*) '+' product, 0>
<expr ->  product(*), 0>
<factor ->  '1'(*), 0>
<product ->  factor(*), 0>
<product ->  product(*) '*' factor, 0>

Q2:
<expr ->  expr '+'(*) product, 0>
<factor -> (*) '1', 2>
<factor -> (*) '2', 2>
<factor -> (*) '3', 2>
<product -> (*) factor, 2>
<product -> (*) product '*' factor, 2>

Q3:
<$Start ->  expr(*), 0>
<expr ->  expr(*) '+' product, 0>
<expr ->  expr '+' product(*), 0>
<factor ->  '2'(*), 2>
<product ->  factor(*), 2>
<product ->  product(*) '*' factor, 2>

Q4:
<factor -> (*) '1', 4>
<factor -> (*) '2', 4>
<factor -> (*) '3', 4>
<product ->  product '*'(*) factor, 2>

Q5:
<$Start ->  expr(*), 0>
<expr ->  expr(*) '+' product, 0>
<expr ->  expr '+' product(*), 0>
<factor ->  '3'(*), 4>
<product ->  product(*) '*' factor, 2>
<product ->  product '*' factor(*), 2>
```

Abbildung 12.9: Ausgabe des Earley-Parsers für die Eingabe "`1+2*3`" und die Grammatik aus Abbildung 12.1.

12.3 Korrektheit und Vollständigkeit

In diesem Kapitel beweisen wir zwei Eigenschaften des von Earley angegebenen Algorithmus. Zunächst zeigen wir, dass immer dann, wenn wir den Algorithmus auf einen String $s = x_1 x_2 \cdots x_n$ anwenden und nach Beendigung des Algorithmus das Earley-Objekt $\langle \widehat{S} \to S \bullet, 0 \rangle$ in der Menge Q_n enthalten ist, wir schließen können, dass der String s sich von dem Start-Symbol S ableiten läßt. Diese Eigenschaft bezeichnen wir als die Korrektheit des Algorithmus. Außerdem beweisen wir, dass auch die Umkehrung gilt: Falls der String s in der von der Variablen S erzeugten Sprache liegt, dann ist das Earley-Objekt $\langle \widehat{S} \to S \bullet, 0 \rangle$ nach Beendigung des Algorithmus ein Element der Menge Q_n. Diese Eigenschaft bezeichnen wir als die Vollständigkeit des Algorithmus. Bei allen folgenden Betrachtungen gehen wir davon aus, dass $G = \langle V, T, S, R \rangle$ die verwendete kontextfreie Grammatik bezeichnet.

Das nachfolgende Lemma wird später benötigt, um die Korrektheit zu zeigen. Es formalisiert die Idee, die der Definition eines Earley-Objekts zu Grunde liegt.

Lemma 34 Es sei $s = x_1 x_2 \cdots x_n \in T^*$ der String, auf den wir den Algorithmus von Earley anwenden. Weiter gelte

$$\langle A \to \alpha \bullet \beta, k \rangle \in Q_i \quad \text{mit } i \in \{0, \cdots, n\}.$$

Dann gilt

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i.$$

Beweis: Wir führen den Beweis durch Induktion über die Anzahl l der Berechnungs-Schritte, die der Algorithmus durchgeführt hat, um $\langle A \to \alpha \bullet \beta, k \rangle \in Q_i$ nachzuweisen.

I.A.: $l = 0$. Zu Beginn enthält die Menge Q_0 nur das Earley-Objekt

$$\langle \widehat{S} \to \bullet S, 0 \rangle$$

und alle anderen Mengen Q_i sind leer. Damit müssen wir die Behauptung nur für dieses eine Earley-Objekt nachweisen. Für dieses Earley-Objekt haben die Variablen A, α, β und k folgende Werte:

(a) $A = \widehat{S}$,
(b) $\alpha = \varepsilon$,
(c) $\beta = S$,
(d) $i = 0$,
(e) $k = 0$.

Wir müssen dann zeigen, dass

$$\varepsilon \Rightarrow^* x_1 \cdots x_0$$

gilt. Diese Behauptung folgt aus $x_1 \cdots x_0 = \varepsilon$.

I.S.: $0, \cdots, l \mapsto l+1$. Wir müssen eine Fallunterscheidung nach der Art der Operation durchführen, mit der das Earley-Objekt $E = \langle A \to \alpha \bullet \beta, k \rangle$ erzeugt worden ist.

(a) E ist durch eine Vorhersage-Operation in Q_i eingefügt worden. Daher enthält Q_i ein Earley-Objekt der Form

$$F = \langle A' \to \alpha' \bullet A\, \delta, k' \rangle.$$

Dann muss die Grammatik eine Regel der Form $A \to \beta$ enthalten, mit der die Vorhersage-Operation das Earley-Objekt

$$\langle A \to \bullet \beta, i \rangle$$

erzeugt hat. Damit gilt also $\alpha = \varepsilon$ und $k = i$. Es ist zu zeigen, dass

$$\varepsilon \Rightarrow^* x_{i+1} \cdots x_i$$

gilt. Wegen $x_{i+1} \cdots x_i = \varepsilon$ ist das trivial.

(b) E ist durch eine Lese-Operation in Q_i eingefügt worden. Dann gibt es ein Earley-Objekt der Form $\langle A \to \alpha' \bullet x_i \beta, k\rangle \in Q_{i-1}$, es gilt $\alpha = \alpha' x_i$ und nach Induktions-Voraussetzung gilt

$$\alpha' \Rightarrow^* x_{k+1} \cdots x_{i-1}$$

Wir müssen zeigen, dass

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i$$

gilt. Dies folgt aus

$$\alpha = \alpha' x_i \Rightarrow^* x_{k+1} \cdots x_{i-1} x_i = x_{k+1} \cdots x_i.$$

(c) E ist durch eine Vervollständigungs-Operation in Q_i eingefügt worden. Dann hat E die Form

$$E = \langle A \to \alpha' C \bullet \beta, k\rangle$$

und es gibt ein Earley-Objekt

$$\langle C \to \delta \bullet, j\rangle \in Q_i$$

und ein weiteres Earley-Objekt

$$\langle A \to \alpha' \bullet C\beta, k\rangle \in Q_j.$$

Also haben wir $\alpha = \alpha' C$. Aus $\langle A \to \alpha' \bullet C\beta, k\rangle \in Q_j$ folgt nach Induktions-Voraussetzung

$$\alpha' \Rightarrow^* x_{k+1} \cdots x_j$$

und aus $\langle C \to \delta \bullet, j\rangle \in Q_i$ folgt nach Induktions-Voraussetzung

$$\delta \Rightarrow^* x_{j+1} \cdots x_i,$$

so dass wir insgesamt die folgende Ableitung haben:

$$\begin{array}{lcl} \alpha & = & \alpha' C \\ & \Rightarrow^* & x_{k+1} \cdots x_j C \\ & \Rightarrow & x_{k+1} \cdots x_j \delta \\ & \Rightarrow^* & x_{k+1} \cdots x_j x_{j+1} \cdots x_i \\ & = & x_{k+1} \cdots x_i \end{array}$$

Damit ist $\alpha \Rightarrow^* x_{k+1} \cdots x_i$ nachgewiesen. □

Korollar 35 (Korrektheit)
Wenden wir den Algorithmus von Earley auf den String $s = x_1 \cdots x_n$ an und gilt $\langle \widehat{S} \to S\bullet, 0\rangle \in Q_n$, so folgt $s \in L(G)$.

Beweis: Setzen wir $A := \widehat{S}$, $\alpha := S$, $\beta := \varepsilon$, $k := 0$ und $i := n$, so folgt das Ergebnis unmittelbar, wenn wir das letzte Lemma auf die Voraussetzung anwenden. □

Das Korollar zeigt: Produziert der Algorithmus von Earley das Earley-Objekt

$$\langle \widehat{S} \to S\bullet, 0\rangle \in Q_n,$$

so liegt der String, auf den wir den Algorithmus angewendet haben, tatsächlich in der von der Grammatik erzeugten Sprache. Wir wollen aber auch die Umkehrung dieser Aussage nachweisen: Falls ein String s von der Grammatik erzeugt wird, so wird der Algorithmus von Earley dies auch erkennen. Dazu zeigen wir zunächst das folgende Lemma.

Lemma 36 Angenommen, wir haben die folgenden Voraussetzungen:

1. $\langle A \to \alpha \bullet \beta\gamma, k\rangle \in Q_i$,
2. $\beta \Rightarrow^* x_{i+1} \cdots x_l$.

Dann gilt auch

$$\langle A \to \alpha\beta \bullet \gamma, k\rangle \in Q_l.$$

Beweis: Wir führen den Beweis durch Induktion über die Anzahl der Ableitungs-Schritte in der Ableitung $\beta \Rightarrow^* x_{i+1} \cdots x_l$.

1. Falls die Ableitung die Länge 0 hat, gilt offenbar $\beta = x_{i+1} \cdots x_l$. Damit hat die Voraussetzung $\langle A \to \alpha \bullet \beta\gamma, k\rangle \in Q_i$ die Form

   $$\langle A \to \alpha \bullet x_{i+1} \cdots x_l\gamma, k\rangle \in Q_i.$$

   Wenden wir hier $(l - i)$ mal die Lese-Operation an, so erhalten wir

   $$\langle A \to \alpha x_{i+1} \cdots x_l \bullet \gamma, k\rangle \in Q_l$$

   und wegen $\beta = x_{i+1} \cdots x_l$ ist das äquivalent zu

   $$\langle A \to \alpha\beta \bullet \gamma, k\rangle \in Q_l.$$

   Das ist aber gerade die Behauptung.

2. Wir führen nun den Induktions-Schritt durch, wobei die Ableitung $\beta \Rightarrow^* x_{i+1} \cdots x_l$ eine Länge größer als 0 hat und nehmen an, dass die erste Regel, die bei dieser Ableitung angewendet worden ist, die Form $D \to \delta$ hat. Zur Vereinfachung nehmen wir außerdem an, dass die Ableitungs-Schritte so durchgeführt werden, dass immer die linkeste Variable ersetzt wird. Die Ableitung hat dann insgesamt die Form

   $$\beta = x_{i+1} \cdots x_j D\mu \Rightarrow x_{i+1} \cdots x_j \delta\mu \Rightarrow^* x_{i+1} \cdots x_l.$$

   und wir haben

   $$\delta \Rightarrow^* x_{j+1} \cdots x_h \tag{12.1}$$

   und

   $$\mu \Rightarrow^* x_{h+1} \cdots x_l \tag{12.2}$$

   für ein geeignetes $h \in \{j + 1, \cdots, l + 1\}$. Also können wir zunächst auf das Earley-Objekt

   $$\langle A \to \alpha \bullet \beta\gamma, k\rangle \in Q_i$$

   $(j - i)$ mal die Lese-Operation anwenden. Damit erhalten wir

   $$\langle A \to \alpha x_{i+1} \cdots x_j \bullet D\mu\gamma, k\rangle \in Q_j. \tag{12.3}$$

   Auf dieses Earley-Objekt wenden wir die Vorhersage-Operation an und erhalten

   $$\langle D \to \bullet\, \delta, j\rangle \in Q_j. \tag{12.4}$$

   Aus (12.4) und (12.1) folgt mit der Induktions-Voraussetzung

   $$\langle D \to \delta\, \bullet, j\rangle \in Q_h. \tag{12.5}$$

   Aus (12.3) und (12.5) folgt mit der Vervollständigungs-Operation

   $$\langle A \to \alpha x_{i+1} \cdots x_j D \bullet \mu\gamma, k\rangle \in Q_h. \tag{12.6}$$

Aus (12.6) und (12.2) folgt mit der Induktions-Voraussetzung

$$\langle A \to \alpha x_{i+1} \cdots x_j D\mu \bullet \gamma, k\rangle \in Q_l. \tag{12.7}$$

Wegen $\beta = x_{i+1} \cdots x_j D\mu$ haben wir damit

$$\langle A \to \alpha\beta \bullet \gamma, k\rangle \in Q_l$$

gezeigt und das ist gerade die Behauptung. □

Korollar 37 (Vollständigkeit)
Falls $s = x_1 \cdots x_n \in L(G)$ ist, dann gilt $\langle \widehat{S} \to S\bullet, 0\rangle \in Q_n$.

Beweis: Der Algorithmus von Earley initialisiert die Menge Q_0 mit dem Earley-Objekt $\langle \widehat{S} \to \bullet S, 0\rangle$. Die Voraussetzung $x_1 \cdots x_n \in L(G)$ heißt gerade $S \Rightarrow^* x_1 \cdots x_n$. Also folgt die Behauptung aus dem eben bewiesenen Lemma, wenn wir dort $\alpha := \varepsilon$, $\beta := S$, $\gamma := \varepsilon$, $i := 0$, $k := 0$ und $l := n$ setzen. □

12.4 Analyse der Komplexität

Wir zeigen, dass sich die Anzahl der Schritte, die der Algorithmus von Earley bei einem String der Länge n durch $\mathcal{O}(n^3)$ nach oben abschätzen läßt. Falls die Grammatik eindeutig ist, wenn es also zu jedem String nur einen Parse-Baum gibt, kann dies zu $\mathcal{O}(n^2)$ verbessert werden. Zusätzlich hat Jay Earley in seiner Doktorarbeit [Ear68] gezeigt, dass der Algorithmus in vielen praktisch relevanten Fällen eine lineare Komplexität hat.

Der Schlüssel bei der Berechnung der Komplexität des Algorithmus von Earley ist die Betrachtung der Anzahl der Elemente der Menge Q_i. Es gilt offenbar:

$$\text{Wenn } \langle A \to \alpha \bullet \beta, k\rangle \in Q_i, \text{ dann } k \in \{0, \cdots, i\}.$$

Die Komponente $A \to \alpha \bullet \beta$ hängt nur von der Grammatik und nicht von dem zu parsenden String ab, während der Index i von der Länge des zu parsenden Strings abhängig ist, es gilt

$$i \in \{0, \cdots, n\}.$$

Interessiert nur das Wachstum der Mengen Q_i in Abhängigkeit von der Länge des zu parsenden Strings, so kann daher die Anzahl der Elemente der Menge Q_i durch $\mathcal{O}(n)$ abgeschätzt werden. Wir analysieren nun die Anzahl der Rechenschritte, die bei den einzelnen Operationen durchgeführt werden.

1. Bei der Implementierung der Vorhersage-Operation in der Methode *predict*() (Abbildung 12.5) haben wir drei Schleifen.

   (a) Für die äußere `do-while`-Schleife finden wir, dass diese maximal so oft durchlaufen wird, wie neue Earley-Objekte in die Menge Q_i eingefügt werden. Alle mit der Vorhersage-Operation neu eingefügten Objekte haben aber die Form

   $$\langle A \to \bullet\gamma, i\rangle,$$

   der Index hat hier also immer den Wert i. Die Anzahl solcher Objekte ist nur von der Grammatik und nicht von dem Eingabe-String abhängig. Damit kann die Anzahl dieser Schleifen-Durchläufe durch $\mathcal{O}(1)$ abgeschätzt werden.

   (b) Die äußere `for`-Schleife läuft über alle Earley-Objekte der Menge Q_i und wird daher maximal $\mathcal{O}(n)$-mal durchlaufen.

   (c) Die innere `for`-Schleife läuft über alle Grammatik-Regeln und ist von der Länge des zu parsenden Strings unabhängig. Diese Schleife liefert also nur einen Beitrag, der durch $\mathcal{O}(1)$ abgeschätzt werden kann.

Insgesamt hat ein Aufruf der Methode `predict`() daher die Komplexität $\mathcal{O}(1) \cdot O(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

2. Bei der Implementierung der Lese-Operation, die in Abbildung 12.6 gezeigt ist, haben wir eine `for`-Schleife, die über alle Elemente der Menge Q_i iteriert. Da diese Menge $\mathcal{O}(n)$ Elemente enthält, hat die Lese-Operation ebenfalls die Komplexität $\mathcal{O}(n)$.

3. Die von uns in Abbildung 12.4 auf Seite 164 gezeigte Implementierung der Vervollständigungs-Operation in der Methode `complete`() ist ineffizient, weil wir in der äußeren `for`-Schleife immer über alle Elemente der Menge Q_i iterieren. Effizienter wäre es, wenn wir nur über die beim letzten Schleifen-Durchlauf neu hinzu gekommenen Elemente iterieren würden. Dann würde im schlimmsten Falle für jedes Elemente der Menge Q_i einmal die innere `for`-Schleife, die über alle Elemente der Menge Q_j iteriert, ausgeführt. Da die Anzahl der Elemente von Q_i und Q_j jeweils durch $\mathcal{O}(n)$ abgeschätzt werden können, kann die Komplexität der Vervollständigungs-Operation insgesamt mit $\mathcal{O}(n^2)$ abgeschätzt werden.

Da die einzelnen Operationen für alle Mengen Q_i für $i = 0, \cdots, n$ durchgeführt werden müssen, hat der Algorithmus insgesamt die Komplexität $\mathcal{O}(n^3)$. Damit diese Komplexität auch tatsächlich erreicht wird, müßten wir die Implementierung der Methoden so umändern, dass kein Element der Menge Q_i mehrfach betrachtet wird. Da eine solche Implementierung wesentlich unübersichtlicher wäre, haben wir uns aus didaktischen Gründen mit einer ineffizienteren Implementierung begnügt.

Der Nachweis, dass der Algorithmus bei einer eindeutigen Grammatik die Komplexität $\mathcal{O}\big(n^2\big)$ hat, geht über den Rahmen der Vorlesung hinaus und kann in dem Artikel von Earley [Ear70] nachgelesen werden. In seiner Doktorarbeit hat Jay Earley [Ear68] zusätzlich gezeigt, dass der Algorithmus in vielen praktisch relevanten Fällen nur eine lineare Komplexität hat. Es gibt daher eine Reihe von Parser-Generatoren, die den Algorithmus von Earley umsetzen, z. B. das System *Accent*

`http://accent.compilertools.net`,

mit dessen Hilfe sich `C`-Parser für beliebige Grammatiken erzeugen lassen. Ein Problem bei der Verwendung solcher Systeme besteht in der Praxis darin, dass die Frage, ob eine Grammatik eindeutig ist, unentscheidbar ist. Bei der Definition einer neuen Grammatik kann es leicht passieren, dass die Grammatik aufgrund eines Design-Fehlers nicht eindeutig ist. Bei der Verwendung eines Earley-Parser-Generators können solche Fehler erst zur Laufzeit des erzeugten Parsers bemerkt werden. Bei Systemen wie *Antlr* oder *Bison*, die mit einer eingeschränkteren Klasse von Grammatiken arbeiten, tritt dieses Problem nicht auf, denn die Grammatiken, für die sich mit einem solchen System ein Parser erzeugen läßt, sind nach Konstruktion eindeutig. Ist also eine Grammatik aufgrund eines Design-Fehlers nicht eindeutig, so liegt Sie erst recht nicht in der eingeschränkten Klasse von LR(1)-Grammatiken, die sich beispielsweise mit *Bison* bearbeiten lassen und der Fehler wird bereits bei der Erstellung des Parsers bemerkt.

Kapitel 13

LALR-Parser

Bei der Konstruktion eines Parsers gibt es generell zwei Möglichkeiten: Wir können *Top-Down* oder *Bottom-Up* vorgehen. Den Top-Down-Ansatz haben wir bereits ausführlich diskutiert. In diesem Kapitel beleuchten wir nun den Bottom-Up-Ansatz. Dazu stellen wir im nächsten Abschnitt das allgemeine Konzept vor, das einem *Bottom-Up-Parser* zu Grunde liegt. Im darauf folgenden Abschnitt zeigen wir, wie Bottom-Up-Parser implementiert werden können und stellen als eine Implementierungsmöglichkeit die *Shift-Reduce-Parser* vor. Ein Shift-Reduce-Parser arbeitet mit Hilfe einer Tabelle, in der hinterlegt ist, wie der Parser in einem bestimmten Zustand die Eingaben verarbeiten muss. Die Theorie, wie eine solche Tabelle sinnvoll mit Informationen gefüllt werden kann, entwickeln wir dann in dem folgenden Abschnitt: Zunächst diskutieren wir die *SLR-Parser* (*simple LR-Parser*). Dies ist die einfachste Klasse von Shift-Reduce-Parsern. Das Konzept der SLR-Parser ist leider für die Praxis nicht mächtig genug. Daher verfeinern wir dieses Konzept und erhalten so die Klasse der *kanonischen LR-Parser*. Da die Tabellen für LR-Parser in der Praxis häufig zu groß werden, vereinfacht man diese Tabellen etwas und erhält dann das Konzept der *LALR-Parser*, das von der Mächtigkeit zwischen dem Konzept der *SLR-Parser* und dem Konzept der *LR-Parser* liegt. Im nächsten Kapitel werden wir den Parser-Generator *JavaCup* diskutieren, der ein LALR-Parser ist.

13.1 Bottom-Up-Parser

Die mit *Antlr* erstellten Parser sind sogenannte *Top-Down-Parser*: Ausgehend von dem Start-Symbol der Grammatik wurde versucht, eine gegebene Eingabe durch Anwendung der verschiedenen Grammatik-Regeln zu parsen. Die Parser, die wir nun entwickeln werden, sind *Bottom-Up-Parser*. Bei einem solchen Parser ist die Idee, dass wir von dem zu parsenden String ausgehen und dort Terminale an Hand der rechten Seiten der Grammatik-Regeln zusammen fassen.

Wir versuchen den String "`1 + 2 * 3`" mit der Grammatik, die durch die Regeln

$$
\begin{array}{lcl}
E & \rightarrow & E \text{ “+” } P \mid P \\
P & \rightarrow & P \text{ “*” } F \mid F \\
F & \rightarrow & \text{“1”} \mid \text{“2”} \mid \text{“3”}
\end{array}
$$

gegeben ist, zu parsen. Dazu suchen wir in diesem String Teilstrings, die den rechten Seiten von Grammatikregeln entsprechen, wobei wir den String von links nach rechts durchsuchen. Auf diese Art versuchen wir, einen Parse-Baum rückwärts von unten aufzubauen:

$$
\begin{array}{lcll}
\texttt{1 + 2 * 3} & \Leftarrow & F\ \texttt{+ 2 * 3} & \text{(Regel: } F \rightarrow \text{“}\texttt{1}\text{”)} \\
& \Leftarrow & P\ \texttt{+ 2 * 3} & \text{(Regel: } P \rightarrow F\text{)} \\
& \Leftarrow & E\ \texttt{+ 2 * 3} & \text{(Regel: } E \rightarrow P\text{)} \\
& \Leftarrow & E\ \texttt{+}\ F\ \texttt{* 3} & \text{(Regel: } F \rightarrow \text{“}\texttt{2}\text{”)} \\
& \Leftarrow & E\ \texttt{+}\ P\ \texttt{* 3} & \text{(Regel: } P \rightarrow F\text{)} \\
& \Leftarrow & E\ \texttt{+}\ P\ \texttt{*}\ F & \text{(Regel: } F \rightarrow \text{“}\texttt{3}\text{”)} \\
& \Leftarrow & E\ \texttt{+}\ P & \text{(Regel: } P \rightarrow P\ \text{“}\texttt{*}\text{”}\ F\text{)} \\
& \Leftarrow & E & \text{(Regel: } E \rightarrow E\ \text{“}\texttt{+}\text{”}\ P\text{)}
\end{array}
$$

Im ersten Schritt haben wir beispielsweise die Grammatik-Regel $F \rightarrow$ “1” benutzt, um den String “1” durch F zu ersetzen und dabei dann den String “F + 2 * 3” erhalten. Im zweiten Schritt haben wir die Regel $P \rightarrow F$ benutzt, um F durch P zu ersetzen. Auf diese Art und Weise haben wir am Ende den ursprünglichen String “1 + 2 * 3” auf E zurück geführt. Wir können an dieser Stelle zwei Beobachtungen machen:

1. Wir ersetzen bei unserem Vorgehen immer den am weitesten links stehenden Teilstring, der ersetzt werden kann, wenn wir den anfangs gegebenen String auf das Start-Symbol der Grammatik zurück führen wollen.

2. Schreiben wir die Ableitung, die wir rückwärts konstruiert haben, noch einmal in der richtigen Reihenfolge hin, so erhalten wir:

   $$
   \begin{array}{lcl}
   E & \Rightarrow & E\ \texttt{+}\ P \\
   & \Rightarrow & E\ \texttt{+}\ P\ \texttt{*}\ F \\
   & \Rightarrow & E\ \texttt{+}\ P\ \texttt{* 3} \\
   & \Rightarrow & E\ \texttt{+}\ F\ \texttt{* 3} \\
   & \Rightarrow & E\ \texttt{+ 2 * 3} \\
   & \Rightarrow & P\ \texttt{+ 2 * 3} \\
   & \Rightarrow & F\ \texttt{+ 2 * 3} \\
   & \Rightarrow & \texttt{1 + 2 * 3}
   \end{array}
   $$

   Wir sehen hier, dass bei dieser Ableitung immer die am weitesten rechts stehende syntaktische Variable ersetzt worden ist. Eine derartige Ableitung wird als *Rechts-Ableitung* bezeichnet.

   Im Gegensatz dazu ist es bei den Ableitungen, die ein *Top-Down-Parser* erzeugt, genau umgekehrt: Dort wird immer die am weitesten links stehende syntaktische Variable ersetzt. Die mit einem solchen Parser erzeugten Ableitungen heißen daher *Links-Ableitungen*.

Die obigen beiden Beobachtungen sind der Grund, weshalb die Parser, die wir in diesem Kapitel diskutieren, als *LR-Parser* bezeichnet werden. Das L steht für *left to right* und beschreibt die Vorgehensweise, dass der String immer von links nach rechts durchsucht wird, während das R für *rightmost derivation* steht und ausdrückt, dass solche Parser eine Rechts-Ableitung konstruieren.

Bei der Implementierung eines LR-Parsers stellen sich zwei Fragen:

1. Welche Teilstrings ersetzen wir?

2. Welche Regeln verwenden wir dabei?

Die Beantwortung dieser Fragen ist im Allgemeinen nicht trivial. Zwar gehen wir die Strings immer von links nach rechts durch, aber damit ist noch nicht unbeding klar, welchen Teilstring wir ersetzen, denn die potentiell zu ersetzenden Teilstrings können sich durchaus überlappen. Betrachten wir beispielsweise das Zwischenergebnis

$$E\ \texttt{+}\ P\ \texttt{* 3},$$

das wir oben im fünften Schritt erhalten haben. Hier könnten wir den Teilstring “P” mit Hilfe der Regel

$$E \rightarrow P$$

durch "E" ersetzen. Dann würden wir den String

$$E \texttt{ + } E \texttt{ * 3}$$

erhalten. Die einzigen Reduktionen, die wir jetzt noch durchführen können, führen über die Zwischenergebnisse $E \texttt{ + } E \texttt{ * } F$ und $E \texttt{ + } E \texttt{ * } P$ zu dem String

$$E \texttt{ + } E \texttt{ * } E,$$

der sich dann aber mit der oben angegebenen Grammatik nicht mehr reduziren läßt. Die Antwort auf die oben Fragen, welchen Teilstring wir ersetzen und welche Regel wir verwenden, setzt einiges an Theorie voraus, die wir in den folgenden Abschnitten entwickeln werden.

13.2 Shift-Reduce-Parser

Wollen wir einen Bottom-Up-Parser implementieren, so müssen wir uns zunächst die Frage stellen, welche Datenstrukturen wir bei der Implementierung verwenden wollen. Wenn wir uns dabei für einen Stack entscheiden, dann sprechen wir von einem *Shift-Reduce-Parser*. Ist $G = \langle V, T, R, S\rangle$ eine kontextfreie Grammatik, so wird ein Shift-Reduce-Parser P durch ein 4-Tupel

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto}\rangle$$

beschrieben. Dabei gilt:

1. Q ist die Menge der *Zustände* des Shift-Reduce-Parsers.
2. $q_0 \in Q$ ist der Start-Zustand.
3. *action* ist eine Funktion, die als Argumente einen Zustand $q \in Q$ und ein Terminal $t \in T$ erhält. Das Ergebnis ist ein Element der Menge

   $$\mathit{Action} := \big\{\langle \texttt{shift}, q\rangle \mid q \in Q\big\} \cup \big\{\langle \texttt{reduce}, r\rangle \mid r \in R\big\} \cup \big\{\texttt{accept}\big\} \cup \big\{\texttt{error}\big\},$$

   wobei `shift`, `reduce`, `accept` und `error` hier einfach als Strings interpretiert werden, mit denen die verschiedenen Arten von Ergebnissen der Funktion *action*() unterschieden werden können. Zusammenfassend haben wir also:

   $$\mathit{action} : Q \times T \rightarrow \mathit{Action}.$$

4. *goto* ist eine Funktion, die jedem Zustand $q \in Q$ und jeder syntaktischen Variablen $v \in V$ einen neuen Zustand aus Q zuordnet:

   $$\mathit{goto} : Q \times V \rightarrow Q.$$

Ein Shift-Reduce-Parser arbeitet nun mit den folgenden Daten-Strukturen.

1. Einem Stack *states*, auf dem Zustände aus der Menge Q abgelegt werden:

   $$\mathit{states} \in \mathit{Stack}\langle Q\rangle.$$

2. Einem Stack *symbols*, auf dem Grammatik-Symbole, also Terminale und Variablen-Symbole abgelegt werden:

   $$\mathit{symbols} \in \mathit{Stack}\langle T \cup V\rangle.$$

Zur Vereinfachung der folgenden Überlegungen nehmen wir an, dass die Menge T der Terminale das spezielle Symbol "`$`" enthält und dass dieses Symbol das Ende des zu parsenden Strings spezifiziert aber sonst in dem String nicht auftritt. Im Scanner wird dieses Symbol später den Namen Eof für "*end of file*" haben. Weiter nehmen wir an, dass die folgenden beiden Methoden zu Verfügung stehen, um Eingabe-Symbole zu lesen.

1. $\mathit{peekNextToken} : \mathit{TokenStream} \rightarrow T$

   Die Funktion gibt das nächste zu lesende Token aus der Eingabe zurück. Dieses Token wird aber nicht aus der Eingabe entfernt.

2. *removeNextToken* : *TokenStream* → *void*

   Diese Funktion entfernt das nächste Token aus der Eingabe, gibt aber kein Ergebnis zurück.

Falls die Funktion *peekNextToken*() mehrmals aufgerufen wird, ohne dass zwischendurch die Methode *removeNextToken*() aufgerufen wird, dann wird immer das selbe Token zurück gegeben. Damit können wir nun das Verhalten eines Shift-Reduce-Parsers beschreiben:

1. Der Symbol-Stack *symbols* ist am Anfang leer:

   $$symbols := [].$$

2. Zu Beginn enthält der Zustands-Stack *states* den Start-Zustand q_0:

   $$states := [q_0].$$

3. Anschließend werden die folgenden Aktionen solange durchlaufen, wie Eingabe-Symbole zu lesen sind.

   (a) Zunächst speichern wir das Symbol, was im Stack *states* oben liegt, in der Variablen q:

   $$q := states.top().$$

   (b) Weiter sei t das nächste, noch ungelesene Token der Eingabe:

   $$t := peekNextToken().$$

   (c) Die folgenden Aktionen hängen von dem Wert der Funktion $action(q, t)$ ab:

   i. $action(q, t) = \langle \texttt{shift}, s \rangle$.
   In diesem Fall wird das Token t auf den Symbol-Stack abgelegt
   $$symbols.push(t)$$
   und auf den Zustands-Stack legen wir den neuen Zustand s:
   $$states.push(s).$$
   Anschließend wird das Token t durch den Aufruf von
   $$removeNextToken()$$
   aus der Eingabe entfernt.

   ii. $action(q, t) = \langle \texttt{reduce}, r \rangle$.
   Dann ist r eine Grammatik-Regel der Form
   $$r = (A \to X_1 \cdots X_n)$$
   In diesem Fall liegen die Symbole X_1, $\cdots$, X_n auf dem Symbol-Stack. Da wir den Symbol-Stack mit der Grammatik-Regel r reduzieren wollen, werden diese Symbole n nun vom Symbol-Stack entfernt und anschließend wird statt dessen das Symbol A auf dem Symbol-Stack abgelegt:
   $$symbols.pop(n);\ symbols.push(A).$$
   Das Token t verbleibt in diesem Fall in der Eingabe, die Funktion *removeNextToken*() wird also nicht aufgerufen.
   Genauso wie wir n Elemente vom Symbol-Stack entfernen, entfernen wir auch n Elemente vom Zustands-Stack.
   $$states.pop(n);$$
   Anschließend bestimmen wir den Zustand s, der nach Entfernung von n Zuständen auf dem Zustands-Stack liegt
   $$s := states.top()$$
   und legen danach den Zustand $goto(s, A)$ auf dem Zustands-Stack ab:
   $$states.push(goto(s, A)).$$

```
1     symbols := [];
2     states  := [ q0 ];
3     while (not end-of-file) {
4         q := states.top();
5         t := peekNextToken();
6         a := action(q,t);
7         switch {
8         when a = ⟨shift, s⟩:
9              symbols.push(t);
10             states .push(s);
11             removeNextToken();
12        when a = ⟨reduce, A → X1 ··· Xn⟩:
13             symbols.pop(n);
14             symbols.push(A);
15             states .pop(n);
16             s := states.top();
17             states.push(goto(s, A));
18        when a = accept:
19             print("Parsen erfolgreich");
20             return;
21        when a = error:
22             error;
23        }
24    }
```

Abbildung 13.1: Algorithmus eines Shift-Reduce-Parsers als Pseudo-Code

iii. $action(q, t) =$ `accept`.
   In diesem Fall wird das Parsen mit Erfolg beendet.

iv. $action(q, t) =$ `error`.
   Falls dieser Fall eintritt, wurde ein Syntax-Fehler gefunden.

Abbildung 13.1 zeigt die Implementierung dieses Algorithmus als Pseudo-Code. Um den Pseudo-Code zu verstehen, nehmen wir an, der zu parsende String habe die Form

$$t_1 \cdots t_n t_{n+1},$$

wobei $t_{n+1} =$ "`$`" gilt. Hier spezifiziert "`$`" das Ende des zu parsenden Strings, der eigentlich zu erkennende String besteht also nur aus den Terminalen $t_1 \cdots t_n$. Wir betrachten nun die Situation, in der die Terminale $t_1 \cdots t_j$ bereits gelesen wurden, während die Terminale $t_{j+1} \cdots t_n$ sich noch im Eingabe-Strom befinden. Die Methode *peekNextToken*() würde also beim nächsten Aufruf das Terminal t_{j+1} zurück liefern. Zu diesem Zeitpunkt befinden sich dann die Symbole $X_1, \cdots, X_k$ auf dem Symbol-Stack, es gilt also

$$symbols = [X_1, \cdots, X_k],$$

wobei X_k das zuoberst auf dem Stack liegende Symbol bezeichnet. Dann kann nach Konstruktion des oben gezeigten Algorithmus aus dem Symbol-String $X_1 \cdots X_k$ der Terminal-String $t_1 \cdots t_j$ abgeleitet werden:

$$X_1 \cdots X_k \Rightarrow^* t_1 \cdots t_j. \tag{$*$}$$

Die Behauptung $(*)$ kann leicht durch eine Induktion über die Anzahl der Durchläufe der `while`-Schleife nachgewiesen werden. Gleichzeitig hat der Stack *states* dann die Form

$$states = [q_0, q_1, \cdots, q_k].$$

Es wird unser Ziel sein die Zustände q_i so zu konstruieren, dass der Zustand q_k alle Informationen über den Symbol-Stack $[X_1, \cdots, X_k]$ speichert, die benötigt werden, um die Entscheidung, ob ein gegebenes Terminal auf den Symbol-Stack geschoben wird (*shift*), oder ob statt dessen der Symbol-Stack durch Anwendung einer Regel reduziert wird (*reduce*).

Wir bemerken noch, dass ein Shift-Reduce-Parser, der nach dem obigen Schema aufgebaut ist, den Vorteil hat, das der zu parsende String nicht vollständig abgespeichert werden muss, denn die Entscheidungen, die beim Parsen getroffen werden müssen, basieren immer nur auf dem nächsten zu lesenden Zeichen. Diese Eigenschaft bedingt, dass Shift-Reduce-Parser mit vergleichsweise wenig Speicherplatz auskommen können. Dieser Umstand hat in den Anfangsjahren der Entwicklung von Übersetzern wesentlich zur Verbreitung von Shift-Reduce-Parser beigetragen.

13.2.1 Implementierung eines *Shift-Reduce-Parsers*

Eine Implementierung des in Abbildung 13.1 gezeigten Algorithmus ist in den Abbildungen 13.2 und 13.3 auf den Seiten 183 und 184 gezeigt. Wir diskutieren zunächst die in Abbildung 13.2 gezeigte Struktur der Klasse *ShiftReduceParser*. Aus didaktischen Gründen habe ich darauf verzichtet, die Implementierung möglichst effizient zu gestalten, denn eine solche Implementierung wäre deutlich komplexer.

1. Die Klasse enthält die folgenden Member-Variablen, die alle am Anfang der Klasse deklariert sind.
   (a) `mTokenList` enthält die Liste aller zu parsenden Tokens. Diese Liste wird von einem Scanner zur Verfügung gestellt.
   (b) `mIndex` gibt den Index des nächsten aus der Liste `mTokenList` zu lesenden Tokens an.
   (c) `mSymbols` ist der Symbol-Stack.
   (d) `mStates` ist der Zustands-Stack.
   (e) `mParseTable` kodiert die beiden Funktionen *action*() und *goto*() in Form von Tabellen.
2. Die Methode *main*() hat die Aufgabe, den Scanner aufzurufen und von diesem eine Liste aller Token erzeugen zu lassen.
   (a) Dazu wird zunächst ein Scanner erzeugt, der von der Standard-Eingabe liest.
   (b) Die Liste aller Token wird in der in Zeile 11 erzeugten `ArrayList<Symbol>` gespeichert.
   (c) Das eigentliche Lesen der Token geschieht dann in der `do-while`-Schleife, die alle vorhandenen Token liest. Anschließend wird noch ein spezielles *End-Of-File-Token* am Ende der Liste eingefügt.

   Der Scanner, den wir hier verwenden, wurde von dem Werkzeug *JFlex* erzeugt.

   Die Liste aller gelesenen Token wird in Zeile 21 dem Konstruktor der Klasse `ShiftReduceParser` übergeben.
3. Der Konstruktor der Klasse `ShiftReduceParser` hat die Aufgabe, alle Member-Variablen zu initialisieren.
   (a) `mTokenList` wird mit der als Argument übergebenen Token-Liste initialisiert.
   (b) `mIndex` wird am Anfang auf 0 gesetzt. Damit zeigt dieser Index dann auf das erste zu lesende Token.
   (c) Anschließend werden Symbol-Stack und Zustands-Stack angelegt.
   (d) Zum Schluss füllt der Konstruktor `ParseTable`() die Tabellen, in denen die Funktionen *action*() und *goto*() abgespeichert sind.
4. Die Methode *parse*() ist aus Platzgünden aus Abbildung 13.2 ausgeklammert worden und wird später diskutiert.

```
1  public class ShiftReduceParser {
2      private List<Symbol>  mTokenList;
3      private int           mIndex;    // index of next token
4      private Stack<Symbol> mSymbols;
5      private Stack<State>  mStates;0
6      private ParseTable    mParseTable;
7      private Map<Pair<State, Symbol>, Action> mActionTable;
8      private Map<Pair<State, Symbol>, State>  mGotoTable;
9
10     public static void main(String args[]) throws Error, IOException {
11         Scanner           scanner   = new Scanner(System.in);
12         ArrayList<Symbol> tokenList = new ArrayList<Symbol>();
13         Symbol            symbol;
14         do {
15             symbol = scanner.yylex();
16             if (symbol != null) {
17                 tokenList.add(symbol);
18             } else {
19                 tokenList.add(new Symbol(Symbol.EOF, null));
20             }
21         } while (symbol != null); // end of file
22         ShiftReduceParser parser = new ShiftReduceParser(tokenList);
23         if (parser.parse()) {
24             System.out.println("Parsing successful!");
25         } else {
26             System.out.println("Syntax error!");
27         }
28     }
29     public ShiftReduceParser(List<Symbol> input) {
30         mTokenList  = input;
31         mIndex      = 0;
32         mSymbols    = new Stack<Symbol>();
33         mStates     = new Stack<State>();
34         mParseTable = new ParseTable();
35     }
36     public boolean parse() { ··· }
37     private Symbol peekNextToken() { return mTokenList.get(mIndex); }
38     private void removeNextToken() { ++mIndex;                      }
39     private State startState()     { return new State(0);           }
40     private Action action(State s, Symbol t) {
41         Pair<State, Symbol> p = new Pair<State, Symbol>(s, t);
42         return mParseTable.mActionTable.get(p);
43     }
44     private State gotoState(State state, Symbol symbol) {
45         Pair<State, Symbol> p = new Pair<State, Symbol>(state, symbol);
46         return mParseTable.mGotoTable.get(p);
47     }
48 }
```

Abbildung 13.2: Ein Shift-Reduce-Parser für arithmetische Ausdrücke.

5. Die Methode *peekNextToken*() liefert das Token zurück, auf das der Zeiger `mIndex` zeigt. Zu beachten ist, dass der Zeiger `mIndex` dabei nicht erhöht wird.

6. Die Methode *removeNextToken*() inkrementiert den Zeiger `mIndex`, damit anschließend das nächste Token gelesen werden kann.

7. Die Methode $action(s, t)$ berechnet für einen gegebenen Zustand s und ein Token t den Funktionswert dadurch, dass der Wert in der Tabelle `mActionTable` nachgeschlagen wird. Dazu müssen die beiden Argumente erst zu einem Paar zusammengefasst werden.

8. Die Methode *gotoState*(`state`, `symbol`) berechnet für einen gegebenen Zustand `state` und eine syntaktische Variable `symbol` den Funktionswert durch Nachschlagen in der Tabelle `mGotoTable`.

```
1  public boolean parse() {
2      State start = startState();
3      mStates.push(start);
4      while (true) {
5          State  q = mStates.peek();
6          Symbol t = peekNextToken();
7          Action p = action(q,t);
8          if (p == null) {
9              return false;
10         }
11         if (p instanceof Shift) {
12             mSymbols.push(t);
13             mStates .push(((Shift) p).getState());
14             removeNextToken();
15         } else if (p instanceof Reduce) {
16             Rule         rule = ((Reduce) p).getRule();
17             Symbol       head = rule.getHead();
18             List<Symbol> body = rule.getBody();
19             for (Symbol s: body) {
20                 mSymbols.pop();
21                 mStates .pop();
22             }
23             mSymbols.push(head);
24             State s = mStates.peek();
25             mStates.push( gotoState(s, head) );
26         } else if (p instanceof Accept) {
27             return true;
28         }
29     }
30 }
```

Abbildung 13.3: Die Methode *parse*() des Shift-Reduce-Parsers.

Als nächstes diskutieren wir die in Abbildung 13.3 gezeigte Implementierung der Methode *parse*(). Diese Methode setzt den in Abbildung 13.1 gezeigten Pseudo-Code um.

1. Zunächst wird der Start-Zustand auf den Zustands-Stack gelegt.

2. Es folgt eine Endlos-Schleife, die sich von Zeile 4 bis zum Ende der Methode erstreckt und die wir jetzt im Detail diskutieren.

(a) q ist der Zustand, der oben auf dem Zustands-Stack liegt.

(b) t ist das nächste zu lesende Token.

(c) p ist der Funktionswert $action(q, t)$. Die Funktion wird *action*() in einer Tabelle abgespeichert. Um diese Tabelle nicht zu groß werden zu lassen, werden dort die Einträge, die den Wert `error` haben weggelassen. Für diese Einträge gibt die Funktion dann den Wert `null` zurück.

Generell werden die Werte der Methode *action*() als Elemente der abstrakten Klasse *Action* dargestellt. Von dieser Klasse werden drei Klassen abgeleitet:

i. Die Klasse *Shift* stellt Paare der Form $\langle$`shift`$, s\rangle$ dar. Diese Klasse enthält eine Member-Variable `mState`, in welcher der Zustand s gespeichert ist.

ii. Die Klasse *Reduce* stellt Paare der Form $\langle$`reduce`$, r\rangle$ dar. Diese Klasse enthält eine Member-Variable `mRule`, in welcher die Regel r gespeichert ist.

iii. Die Klasse *Accept* stellt das Ergebnis `accept` dar.

iv. Das Ergebnis `error` wird durch den Wert `null` signalisiert.

(d) Die Methode *parse*() gibt in Zeile 9 den Wert `false` zurück, wenn $action(q, t)$ den Wert `null` hat.

(e) Falls die Funktion $action(q, t)$ als Ergebnis den Wert $\langle$`shift`$, s\rangle$ hat, so wird das Token t auf den Symbol-Stack gelegt und aus der Token-Liste entfernt. Der Zustand s wird auf den Zustands-Stack gelegt.

(f) Falls die Funktion $action(q, t)$ als Ergebnis den Wert $\langle$`reduce`$, r\rangle$ hat und die rechte Seite der Regel r aus n Symbolen besteht, so werden einerseits n Symbole vom Symbol-Stack genommen und andererseits werden n Zustände vom Zustands-Stack entfernt. Anschließend wird die linke Seite der Regel auf dem Symbol-Stack abgelegt. Dann wird ein neuer Zustand mit Hilfe der Funktion *goto*() ausgerechnet und auf dem Symbol-Stack abgelegt.

(g) Falls die Funktion $action(q, t)$ als Ergebnis den Wert `accept` hat, wird das Parsen mit Erfolg abgeschlossen.

Die bisher gezeigten Methoden waren von der zu parsenden Grammatik unabhängig. Die Grammatik hat nur Einfluss auf den Konstruktor *ParseTable*(), denn dort werden die Tabelen gefüllt, mit deren Hilfe die Funktionen *action*() und *goto*() realisiert werden. Die Tabellen 13.1 und 13.2 zeigen diese Tabellen für die in Abbildung 13.4 angegebene Grammatik. Bei dieser Grammatik ergeben sich insgesamt 16 verschiedene Zustände, die wir auf die Namen s_0, s_1, $\cdots$, s_{15} taufen. Wir haben in der Tabelle 13.1 zwei verschiedene Abkürzungen verwendet:

1. $\langle shft, s_i\rangle$ steht für $\langle$`shift`$, s_i\rangle$.

2. $\langle rdc, r_i\rangle$ steht für $\langle$`reduce`$, r_i\rangle$, wobei r_i für die i-te Regel bezeichnet. Dabei haben wir die in Abbildung 13.4 gezeigten Regeln wie folgt durchnummeriert:

   (a) $r_1 = (Expr \rightarrow Expr$ "`+`" $Product)$

   (b) $r_2 = (Expr \rightarrow Expr$ "`-`" $Product)$

   (c) $r_3 = (Expr \rightarrow Product)$

   (d) $r_4 = (Product \rightarrow Product$ "`*`" $Factor)$

   (e) $r_5 = (Product \rightarrow Product$ "`/`" $Factor)$

   (f) $r_6 = (Product \rightarrow Factor)$

   (g) $r_7 = (Factor \rightarrow$ "`(`" $Expr$ "`)`" $)$

   (h) $r_8 = (Factor \rightarrow$ NUMBER$)$

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
 & | & \mathit{Expr}\ \text{“}\texttt{-}\text{”}\ \mathit{Product} \\
 & | & \mathit{Product} \\
\mathit{Product} & \rightarrow & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
 & | & \mathit{Product}\ \text{“}\texttt{/}\text{”}\ \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
 & | & \textsc{Number}
\end{array}
$$

Abbildung 13.4: Eine Grammatik für arithmetische Ausdrücke.

Zustand	EOF	+	-	*	/	(	)	NUMBER
s_0						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_1	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$	$\langle rdc, r_6\rangle$		$\langle rdc, r_6\rangle$	
s_2	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	$\langle rdc, r_8\rangle$	
s_3	$\langle rdc, r_3\rangle$	$\langle rdc, r_3\rangle$	$\langle rdc, r_3\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_3\rangle$	
s_4	*accept*	$\langle shft, s_8\rangle$	$\langle shft, s_9\rangle$					
s_5						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_6		$\langle shft, s_8\rangle$	$\langle shft, s_9\rangle$				$\langle shft, s_7\rangle$	
s_7	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$	$\langle rdc, r_7\rangle$		$\langle rdc, r_7\rangle$	
s_8						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_9						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{10}	$\langle rdc, r_2\rangle$	$\langle rdc, r_2\rangle$	$\langle rdc, r_2\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_2\rangle$	
s_{11}						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{12}						$\langle shft, s_5\rangle$		$\langle shft, s_2\rangle$
s_{13}	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$	$\langle rdc, r_4\rangle$		$\langle rdc, r_4\rangle$	
s_{14}	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$	$\langle rdc, r_5\rangle$		$\langle rdc, r_5\rangle$	
s_{15}	$\langle rdc, r_1\rangle$	$\langle rdc, r_1\rangle$	$\langle rdc, r_1\rangle$	$\langle shft, s_{12}\rangle$	$\langle shft, s_{11}\rangle$		$\langle rdc, r_1\rangle$	

Tabelle 13.1: Tabelle der Funktion *action*().

Die Einträge, die in Tabelle 13.1 frei geblieben sind, haben den Wert `error`. Entsprechendes gilt auch für die freien Einträge in der Tabelle 13.2.
Die Abbildungen 13.5 bis 13.8 zeigen nun die Implementierung der Klasse *parseTable*(), die sich unmittelbar aus den oben gezeigten Tabellen ergibt und daher hier nicht weiter diskutiert wird. Wie diese Tabellen berechnet werden werden wir in den folgenden Abschnitten sehen.

Zstd	*Expr*	*Product*	*Factor*
s_0	s_4	s_3	s_1
s_1			
s_2			
s_3			
s_4			
s_5	s_6	s_3	s_1
s_6			
s_7			
s_8		s_{15}	s_1
s_9		s_{10}	s_1
s_{10}			
s_{11}			s_{14}
s_{12}			s_{13}
s_{13}			
s_{14}			
s_{15}			

Tabelle 13.2: Tabelle der Funktion *goto*().

```
1  public class ParseTable {
2      Map<Pair<State, Symbol>, Action> mActionTable;
3      Map<Pair<State, Symbol>, State>  mGotoTable;
4      public ParseTable() {
5          mActionTable = new TreeMap<Pair<State, Symbol>,  Action>();
6          State s0 = new State(0);
7          State s1 = new State(1);
8          State s2 = new State(2);
9          State s3 = new State(3);
10         State s4 = new State(4);
11         State s5 = new State(5);
12         State s6 = new State(6);
13         State s7 = new State(7);
14         State s8 = new State(8);
15         State s9 = new State(9);
16         State s10 = new State(10);
17         State s11 = new State(11);
18         State s12 = new State(12);
19         State s13 = new State(13);
20         State s14 = new State(14);
21         State s15 = new State(15);
22         Symbol t0 = new Symbol( 0, "EOF");
23         Symbol t1 = new Symbol( 1, "(");
24         Symbol t2 = new Symbol( 2, ")");
25         Symbol t3 = new Symbol( 3, "*");
26         Symbol t4 = new Symbol( 4, "+");
27         Symbol t5 = new Symbol( 5, "-");
28         Symbol t6 = new Symbol( 6, "/");
29         Symbol t7 = new Symbol( 7, "NUMBER");
30         Symbol expr = new Symbol(Symbol.EXPR, "expr" );
31         Symbol factor = new Symbol(Symbol.FACTOR, "factor" );
32         Symbol product = new Symbol(Symbol.PRODUCT, "product" );
33         // expr -> expr '+' product
34         List<Symbol> body2 = new ArrayList<Symbol>();
35         Symbol head2 = expr;
36         body2.add(expr);
37         body2.add(t4);
38         body2.add(product);
39         Rule rule2 = new Rule(head2, body2);
40         // expr -> expr '-' product
41         List<Symbol> body3 = new ArrayList<Symbol>();
42         Symbol head3 = expr;
43         body3.add(expr);
44         body3.add(t5);
45         body3.add(product);
46         Rule rule3 = new Rule(head3, body3);
```

Abbildung 13.5: Die Klasse `ParseTable`, 1. Teil

```
47          // expr -> product
48          List<Symbol> body4 = new ArrayList<Symbol>();
49          Symbol head4 = expr;
50          body4.add(product);
51          Rule rule4 = new Rule(head4, body4);
52          // factor -> '(' expr ')'
53          List<Symbol> body5 = new ArrayList<Symbol>();
54          Symbol head5 = factor;
55          body5.add(t1);
56          body5.add(expr);
57          body5.add(t2);
58          Rule rule5 = new Rule(head5, body5);
59          // factor -> NUMBER
60          List<Symbol> body6 = new ArrayList<Symbol>();
61          Symbol head6 = factor;
62          body6.add(t7);
63          Rule rule6 = new Rule(head6, body6);
64          // product -> factor
65          List<Symbol> body7 = new ArrayList<Symbol>();
66          Symbol head7 = product;
67          body7.add(factor);
68          Rule rule7 = new Rule(head7, body7);
69          // product -> product '*' factor
70          List<Symbol> body8 = new ArrayList<Symbol>();
71          Symbol head8 = product;
72          body8.add(product);
73          body8.add(t3);
74          body8.add(factor);
75          Rule rule8 = new Rule(head8, body8);
76          // product -> product '/' factor
77          List<Symbol> body9 = new ArrayList<Symbol>();
78          Symbol head9 = product;
79          body9.add(product);
80          body9.add(t6);
81          body9.add(factor);
82          Rule rule9 = new Rule(head9, body9);
```

Abbildung 13.6: Die Klasse `ParseTable`, 2. Teil.

```
83          mActionTable.put(new Pair<State, Symbol>(s0, t1), new Shift(s2));
84          mActionTable.put(new Pair<State, Symbol>(s0, t7), new Shift(s12));
85          mActionTable.put(new Pair<State, Symbol>(s1, t0), new Accept());
86          mActionTable.put(new Pair<State, Symbol>(s1, t4), new Shift(s4));
87          mActionTable.put(new Pair<State, Symbol>(s1, t5), new Shift(s6));
88          mActionTable.put(new Pair<State, Symbol>(s2, t1), new Shift(s2));
89          mActionTable.put(new Pair<State, Symbol>(s2, t7), new Shift(s12));
90          mActionTable.put(new Pair<State, Symbol>(s3, t2), new Shift(s11));
91          mActionTable.put(new Pair<State, Symbol>(s3, t4), new Shift(s4));
92          mActionTable.put(new Pair<State, Symbol>(s3, t5), new Shift(s6));
93          mActionTable.put(new Pair<State, Symbol>(s4, t1), new Shift(s2));
94          mActionTable.put(new Pair<State, Symbol>(s4, t7), new Shift(s12));
95          mActionTable.put(new Pair<State, Symbol>(s5, t0), new Reduce(rule2));
96          mActionTable.put(new Pair<State, Symbol>(s5, t2), new Reduce(rule2));
97          mActionTable.put(new Pair<State, Symbol>(s5, t3), new Shift(s9));
98          mActionTable.put(new Pair<State, Symbol>(s5, t4), new Reduce(rule2));
99          mActionTable.put(new Pair<State, Symbol>(s5, t5), new Reduce(rule2));
100         mActionTable.put(new Pair<State, Symbol>(s5, t6), new Shift(s10));
101         mActionTable.put(new Pair<State, Symbol>(s6, t1), new Shift(s2));
102         mActionTable.put(new Pair<State, Symbol>(s6, t7), new Shift(s12));
103         mActionTable.put(new Pair<State, Symbol>(s7, t0), new Reduce(rule3));
104         mActionTable.put(new Pair<State, Symbol>(s7, t2), new Reduce(rule3));
105         mActionTable.put(new Pair<State, Symbol>(s7, t3), new Shift(s9));
106         mActionTable.put(new Pair<State, Symbol>(s7, t4), new Reduce(rule3));
107         mActionTable.put(new Pair<State, Symbol>(s7, t5), new Reduce(rule3));
108         mActionTable.put(new Pair<State, Symbol>(s7, t6), new Shift(s10));
109         mActionTable.put(new Pair<State, Symbol>(s8, t0), new Reduce(rule4));
110         mActionTable.put(new Pair<State, Symbol>(s8, t2), new Reduce(rule4));
111         mActionTable.put(new Pair<State, Symbol>(s8, t3), new Shift(s9));
112         mActionTable.put(new Pair<State, Symbol>(s8, t4), new Reduce(rule4));
113         mActionTable.put(new Pair<State, Symbol>(s8, t5), new Reduce(rule4));
114         mActionTable.put(new Pair<State, Symbol>(s8, t6), new Shift(s10));
115         mActionTable.put(new Pair<State, Symbol>(s9, t1), new Shift(s2));
116         mActionTable.put(new Pair<State, Symbol>(s9, t7), new Shift(s12));
117         mActionTable.put(new Pair<State, Symbol>(s10, t1), new Shift(s2));
118         mActionTable.put(new Pair<State, Symbol>(s10, t7), new Shift(s12));
```

Abbildung 13.7: Die Klasse `ParseTable`, 3. Teil.

```
119             mActionTable.put(new Pair<State, Symbol>(s11, t0), new Reduce(rule5));
120             mActionTable.put(new Pair<State, Symbol>(s11, t2), new Reduce(rule5));
121             mActionTable.put(new Pair<State, Symbol>(s11, t3), new Reduce(rule5));
122             mActionTable.put(new Pair<State, Symbol>(s11, t4), new Reduce(rule5));
123             mActionTable.put(new Pair<State, Symbol>(s11, t5), new Reduce(rule5));
124             mActionTable.put(new Pair<State, Symbol>(s11, t6), new Reduce(rule5));
125             mActionTable.put(new Pair<State, Symbol>(s12, t0), new Reduce(rule6));
126             mActionTable.put(new Pair<State, Symbol>(s12, t2), new Reduce(rule6));
127             mActionTable.put(new Pair<State, Symbol>(s12, t3), new Reduce(rule6));
128             mActionTable.put(new Pair<State, Symbol>(s12, t4), new Reduce(rule6));
129             mActionTable.put(new Pair<State, Symbol>(s12, t5), new Reduce(rule6));
130             mActionTable.put(new Pair<State, Symbol>(s12, t6), new Reduce(rule6));
131             mActionTable.put(new Pair<State, Symbol>(s13, t0), new Reduce(rule7));
132             mActionTable.put(new Pair<State, Symbol>(s13, t2), new Reduce(rule7));
133             mActionTable.put(new Pair<State, Symbol>(s13, t3), new Reduce(rule7));
134             mActionTable.put(new Pair<State, Symbol>(s13, t4), new Reduce(rule7));
135             mActionTable.put(new Pair<State, Symbol>(s13, t5), new Reduce(rule7));
136             mActionTable.put(new Pair<State, Symbol>(s13, t6), new Reduce(rule7));
137             mActionTable.put(new Pair<State, Symbol>(s14, t0), new Reduce(rule8));
138             mActionTable.put(new Pair<State, Symbol>(s14, t2), new Reduce(rule8));
139             mActionTable.put(new Pair<State, Symbol>(s14, t3), new Reduce(rule8));
140             mActionTable.put(new Pair<State, Symbol>(s14, t4), new Reduce(rule8));
141             mActionTable.put(new Pair<State, Symbol>(s14, t5), new Reduce(rule8));
142             mActionTable.put(new Pair<State, Symbol>(s14, t6), new Reduce(rule8));
143             mActionTable.put(new Pair<State, Symbol>(s15, t0), new Reduce(rule9));
144             mActionTable.put(new Pair<State, Symbol>(s15, t2), new Reduce(rule9));
145             mActionTable.put(new Pair<State, Symbol>(s15, t3), new Reduce(rule9));
146             mActionTable.put(new Pair<State, Symbol>(s15, t4), new Reduce(rule9));
147             mActionTable.put(new Pair<State, Symbol>(s15, t5), new Reduce(rule9));
148             mActionTable.put(new Pair<State, Symbol>(s15, t6), new Reduce(rule9));
149             mGotoTable = new TreeMap<Pair<State, Symbol>, State>();
150             mGotoTable.put(new Pair<State, Symbol>(s0, expr), s1);
151             mGotoTable.put(new Pair<State, Symbol>(s0, factor), s13);
152             mGotoTable.put(new Pair<State, Symbol>(s0, product), s8);
153             mGotoTable.put(new Pair<State, Symbol>(s2, expr), s3);
154             mGotoTable.put(new Pair<State, Symbol>(s2, factor), s13);
155             mGotoTable.put(new Pair<State, Symbol>(s2, product), s8);
156             mGotoTable.put(new Pair<State, Symbol>(s4, factor), s13);
157             mGotoTable.put(new Pair<State, Symbol>(s4, product), s5);
158             mGotoTable.put(new Pair<State, Symbol>(s6, factor), s13);
159             mGotoTable.put(new Pair<State, Symbol>(s6, product), s7);
160             mGotoTable.put(new Pair<State, Symbol>(s9, factor), s14);
161             mGotoTable.put(new Pair<State, Symbol>(s10, factor), s15);
162         }
163     }
```

Abbildung 13.8: Die Klasse `ParseTable`, 4. Teil.

13.3 SLR-Parser

In diesem Abschnitt zeigen wir, wie wir für eine gegebene kontextfreie Grammatik G die im letzten Abschnitt verwendeten Funktionen

$$action : Q \times T \rightarrow Action$$

definieren können. Dazu klären wir als erstes, wie die Menge Q der Zustände zu definieren ist. Wir werden die Zustände nun so definieren, dass sie die Information enthalten, welche Regel der Shift-Reduce-Parser anzuwenden versucht, welche Teile der Syntax er bereits erkannt hat und was er noch erwartet. Zu diesem Zweck definieren wir:

Definition 38 (markierte Regel) Eine *markierte Regel* einer Grammatik $G = \langle V, T, R, S\rangle$ ist ein Tripel

$$\langle A, \alpha, \beta\rangle,$$

für das gilt

$$(A \rightarrow \alpha\beta) \in R.$$

Wir schreiben eine markierte Regel $\langle A, \alpha, \beta\rangle$ als

$$A \rightarrow \alpha \bullet \beta.$$ □

Die markierte Regel $A \rightarrow \alpha \bullet \beta$ drückt aus, dass der Parser versucht, mit der Regel $A \rightarrow \alpha\beta$ ein A zu parsen, dabei schon α gesehen haben und als nächstes versucht, β zu erkennen. Das Zeichen $\bullet$ markiert also die Position innerhalb der rechten Seite der Regel, bis zu der wir den String schon erkannt haben. Die Idee ist jetzt, dass wir die Zustände eines SLR-Parsers als Mengen von markierten Regeln darsstellen können. Um diese Idee zu verschaulichen, betrachten wir ein konkretes Beispiel: Wir gehen von der in Abbildung 13.4 auf Seite 186 gezeigten Grammatik aus, wobei wir diese Grammatik noch um ein neues Start-Symbol $\widehat{S}$ und die Regel

$$\widehat{S} \rightarrow Expr$$

erweitern. Der Start-Zustand enthält offenbar die markierte Regel

$$\widehat{S} \rightarrow \varepsilon \bullet Expr,$$

denn am Anfang versuchen wir ja, das Start-Symbol $\widehat{S}$ herzuleiten. Die Komponente ε drückt aus, dass wir bisher noch nichts verarbeitet haben. Neben dieser markierten Regel muss der Start-Zustand dann die markierten Regeln

1. $Expr \rightarrow \varepsilon \bullet Expr$ "+" $Product$,
2. $Expr \rightarrow \varepsilon \bullet Expr$ "-" $Product$ und
3. $Expr \rightarrow \varepsilon \bullet Product$

enthalten. Rechnen wir so weiter, so finden wir, dass der Start-Zustand außerdem noch die folgenden markierten Regeln enthalten muss:

4. $Product \rightarrow \bullet\ Product$ "*" $Factor$,
5. $Product \rightarrow \bullet\ Product$ "/" $Factor$,
6. $Product \rightarrow \bullet\ Factor$,

   denn die markierte Regel $Expr \rightarrow \varepsilon \bullet Product$ zeigt, dass wir eventuell als erstes ein $Product$ parsen müssen und dazu kann jeder der drei obigen Regeln verwendet werden.
7. $Factor \rightarrow \bullet$ "(" $Expr$ ")" ,
8. $Factor \rightarrow \bullet$ NUMBER,

   denn die sechste Regel zeigt, dass wir eventuell als erstes einen $Factor$ parsen müssen.

Damit sehen wir, dass der Start-Zustand aus einer Menge mit 8 markierten Regeln besteht. Die oben praktizierte Art, aus einer gegebenen Regel weitere Regeln abzuleiten, formalisieren wir in dem Begriff des *Abschlusses* einer Menge von markierten Regeln.

Definition 39 (*closure*($\mathcal{M}$)) Es sei $\mathcal{M}$ eine Menge markierter Regeln. Dann definieren wir den *Abschluss* dieser Menge als die kleinste Menge $\mathcal{K}$ markierter Regeln, für die folgendes gilt:

1. $\mathcal{M} \subseteq \mathcal{K}$,

   der Abschluss umfasst also die ursprüngliche Regel-Menge.

2. Ist einerseits

   $$A \to \alpha \bullet B\beta$$

   eine markierte Regel aus der Menge $\mathcal{K}$, wobei B eine syntaktische Variable ist, und ist andererseits

   $$B \to \gamma$$

   eine Grammatik-Regel der zu Grunde liegenden Grammatik G, so ist auch die markierte Regel

   $$B \to \bullet\gamma$$

   ein Element der Menge $\mathcal{K}$. Als Formel schreibt sich dies wie folgt:

   $$(A \to \alpha \bullet B\beta) \in \mathcal{K} \;\wedge\; (B \to \gamma) \in R \;\Rightarrow\; (B \to \bullet\gamma) \in \mathcal{K}$$

Die so definierte Menge $\mathcal{K}$ ist eindeutig bestimmt und wird im Folgenden mit *closure*($\mathcal{M}$) bezeichnet. □

Bemerkung: Wenn Sie sich an den Earley-Algorithmus erinnern, dann sehen Sie, dass bei der Berechnung des Abschlusses die selbe Berechnung wie bei der Vorhersage-Operation des Earley-Algorithmus durchgeführt wird.

Für eine gegebene Menge $\mathcal{M}$ von markierten Regeln, kann die Berechnung von $\mathcal{K} := \mathit{closure}(\mathcal{M})$ iterativ erfolgen:

1. Zunächst setzen wir $\mathcal{K} := \mathcal{M}$.

2. Anschließend suchen wir alle Regeln der Form

   $$A \to \alpha \bullet B\beta$$

   aus der Menge $\mathcal{K}$, für die B eine syntaktische Variable ist und fügen dann für alle Regeln der Form $B \to \gamma$ die neue markierte Regel

   $$B \to \bullet\gamma$$

   in die Menge $\mathcal{K}$ ein.

   Dieser Schritt wird solange iteriert, bis keine neuen Regeln mehr gefunden werden.

Beispiel: Wir gehen von der in Abbildung 13.4 auf Seite 186 gezeigten Grammatik aus und betrachten die Menge

$$\mathcal{M} := \big\{\mathit{Product} \to \mathit{Product}\ \text{“*”} \bullet \mathit{Factor}\big\}$$

Für die Menge *closure*($\mathcal{M}$) finden wir dann

$$\begin{array}{lcl} \mathit{closure}(\mathcal{M}) & = & \big\{ \quad \mathit{Product} \to \mathit{Product}\ \text{“*”} \bullet \mathit{Factor}, \\ & & \quad\ \mathit{Factor} \to \bullet\ \text{“(”}\ \mathit{Expr}\ \text{“)”}, \\ & & \quad\ \mathit{Factor} \to \bullet\ \textsc{Number} \\ & & \big\}. \end{array}$$

□

Unser Ziel ist es, für eine gegebene kontextfreie Grammatik $G = \langle V, T, R, S\rangle$ einen Shift-Reduce-Parser

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto}\rangle$$

zu definieren. Um dieses Ziel zu erreichen, müssen wir uns als ersten überlegen, wie wir die Menge Q der Zustände definieren wollen, denn dann funktioniert die Definition der restlichen Komponenten fast von alleine. Die Idee ist, dass wir die Zustände als Mengen von markierten Regeln definieren. Wir definieren zunächst

$$\Gamma := \big\{A \to \alpha \bullet \beta \mid (A \to \alpha\beta) \in R\big\}$$

als die Menge aller markierten Regeln der Grammatik. Nun ist es allerdings nicht sinnvoll, beliebige Teilmengen von Γ als Zustände zuzulassen: Eine Teilmenge $\mathcal{M} \subseteq \Gamma$ kommt nur dann als Zustand in Betracht, wenn die Menge $\mathcal{M}$ unter der Funktion *closure*() *abgeschlossen* ist, wenn also $\mathit{closure}(\mathcal{M}) = \mathcal{M}$ gilt. Wir definieren daher

$$Q := \big\{\mathcal{M} \in 2^\Gamma \mid \mathit{closure}(\mathcal{M}) = \mathcal{M}\big\}.$$

Die Interpretation der Mengen $\mathcal{M} \in Q$ ist die, dass ein Zustand $\mathcal{M}$ genau die markierten Grammatik-Regeln enthält, die in der durch den Zustand beschriebenen Situation angewendet werden können.

Zur Vereinfachung der folgenden Konstruktionen erweitern wir die Grammatik $G = \langle V, T, R, S\rangle$ durch Einführung eines neuen Start-Symbols $\widehat{S}$ zu einer Grammatik

$$\widehat{G} = \Big\langle V \cup \{\widehat{S}\}, T, \widehat{S}, R \cup \{\widehat{S} \to S\}\Big\rangle.$$

Diese Grammatik bezeichnen wir als die augmentierte Grammatik. Die Verwendung der augmentierten Grammatik vereinfacht die nun folgende Definition des Start-Zustands. Wir setzen nämlich:

$$q_0 := \mathit{closure}\Big(\big\{\widehat{S} \to \bullet S\big\}\Big).$$

Als nächstes konstruieren wir die Funktion *goto*(). Die Definition lautet:

$$\mathit{goto}(\mathcal{M}, B) := \mathit{closure}\Big(\big\{A \to \alpha B \bullet \beta \mid (A \to \alpha \bullet B\beta) \in \mathcal{M}\big\}\Big).$$

Um diese Definition zu verstehen, nehmen wir an, dass der Parser in einem Zustand ist, in dem er versucht, ein A mit Hilfe der Regel $A \to \alpha B \beta$ zu erkennen und dass dabei bereits der Teilstring α erkannt wurde. Dieser Zustand wird durch die markierte Regel

$$A \to \alpha \bullet B\beta$$

beschrieben. Wird nun ein B erkannt, so kann der Parser von dem Zustand, der die Regel $A \to \alpha \bullet B\beta$ enthält in einen Zustand, der die Regel $A \to \alpha B \bullet \beta$ enthält, übergehen. Daher erhalten wir die oben angegebene Definition der Funktion $\mathit{goto}(\mathcal{M}, B)$. Für die gleich folgende Definition der Funktion $\mathit{action}(\mathcal{M}, t)$ ist es nützlich, die Definition der Funktion *goto* auf Terminale zu erweitern. Für beliebige Terminale t setzen wir:

$$\mathit{goto}(\mathcal{M}, t) := \mathit{closure}\Big(\big\{A \to \alpha t \bullet \beta \mid (A \to \alpha \bullet t\beta) \in \mathcal{M}\big\}\Big).$$

Als letztes spezifizieren wir, wie die Funktion $\mathit{action}(\mathcal{M}, t)$ für eine Menge von markierten Regeln $\mathcal{M}$ und ein Token t berechnet wird. Bei der Definition von $\mathit{action}(\mathcal{M}, t)$ unterscheiden wir vier Fälle.

1. Falls $\mathcal{M}$ eine markierte Regel der Form $A \to \alpha \bullet t\beta$ enthält, setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t)\rangle,$$

   denn in diesem Fall versucht der Parser ein A mit Hilfe der Regel $A \to \alpha t \beta$ zu erkennen und hat von der rechten Seite dieser Regel bereits α erkannt. Ist nun das nächste Token im Eingabe-String das Token t, so kann der Parser dieses t lesen und geht dabei von dem Zustand $A \to \alpha \bullet t\beta$ in den Zustand $A \to \alpha t \bullet \beta$ über, der von der Funktion $\mathit{goto}(\mathcal{M}, t)$ berechnet wird. Insgesamt haben wir also

$$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t) \rangle \quad \text{falls} \quad (A \to \alpha \bullet t\beta) \in \mathcal{M}.$$

2. Falls $\mathcal{M}$ eine markierte Regel der Form $A \to \alpha\bullet$ enthält und wenn zusätzlich $t \in \mathit{Follow}(A)$ gilt, dann setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, A \to \alpha \rangle,$$

   denn in diesem Fall versucht der Parser ein A mit Hilfe der Regel $A \to \alpha$ zu erkennen und hat bereits α erkannt. Ist nun das nächste Token im Eingabe-String das Token t und ist darüber hinaus t ein Token, dass auf A folgen kann, gilt also $t \in \mathit{Follow}(A)$, so kann der Parser die Regel $A \to \alpha$ anwenden und den Symbol-Stack mit dieser Regel reduzieren. Wir haben also

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, A \to \alpha \rangle \quad \text{falls } (A \to \alpha\bullet) \in \mathcal{M},\ A \neq \widehat{S} \text{ und } t \in \mathit{Follow}(A).$$

3. Falls $\mathcal{M}$ die markierte Regel $\widehat{S} \to S\bullet$ enthält und wir den zu parsenden String vollständig gelesen haben, dann setzen wir

   $$\mathit{action}(\mathcal{M}, \textsc{Eof}) := \texttt{accept},$$

   denn in diesem Fall versucht der Parser, $\widehat{S}$ mit Hilfe der Regel $\widehat{S} \to S$ zu erkennen und hat also bereits S erkannt. Ist nun das nächste Token im Eingabe-String das Datei-Ende-Zeichen Eof, so liegt der zu parsende String in der durch die Grammatik G spezifizierte Sprache $L(G)$. Wir haben also

   $$\mathit{action}(\mathcal{M}, \textsc{Eof}) := \texttt{accept}, \quad \text{falls } (\widehat{S} \to S\bullet) \in \mathcal{M}.$$

4. In den restlichen Fällen setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \texttt{error}.$$

Zwischen den ersten beiden Regeln kann es Konflikte geben. Wir unterscheiden zwischen zwei Arten von Konflikten.

1. Ein *Shift-Reduce-Konflikt* tritt auf, wenn sowohl der erste Fall als auch der zweite Fall vorliegt. In diesem Fall enthält die Menge $\mathcal{M}$ also zum einen eine markierte Regel der Form

   $$A \to \alpha \bullet t\beta,$$

   zum anderen enthält $\mathcal{M}$ eine Regel der Form

   $$C \to \gamma\bullet \quad \text{mit } t \in \mathit{Follow}(C).$$

   Wenn dann das nächste Token den Wert t hat, ist nicht klar, ob dieses Token auf den Symbol-Stack geschoben und der Parser in einen Zustand mit der markierten Regel $A \to \alpha t \bullet \beta$ übergehen soll, oder ob statt dessen der Symbol-Stack mit der Regel $C \to \gamma$ reduziert werden muss.

2. Eine *Reduce-Reduce-Konflikt* liegt vor, wenn die Menge $\mathcal{M}$ zwei verschiedene markierte Regeln der Form

   $$C_1 \to \gamma_1\bullet \quad \text{und} \quad C_2 \to \gamma_2\bullet$$

   enthält und wenn gleichzeitig $t \in \mathit{Follow}(C_1) \cap \mathit{Follow}(C_2)$ ist, denn dann ist nicht klar, welche der beiden Regeln der Parser anwenden soll, wenn das nächste zu lesende Token den Wert t hat.

Falls einer dieser beiden Konflikte auftritt, dann sagen wir, dass die Grammatik keine SLR-Grammatik ist. Eine solche Grammatik kann mit Hilfe eines SLR-Parser nicht geparst werden. Wir werden später noch Beispiele für die beiden Arten von Konflikten geben, aber zunächst wollen wir eine Grammatik untersuchen, die die SLR-Eigenschaft hat und wollen für diese Grammatik die Funktionen *goto*() und *action*() auch tatsächlich berechnen. Wir nehmen als Grundlage die

in Abbildung 13.4 gezeigte Grammatik. Da die syntaktische Variable *expr* auf der rechten Seite von Grammatik-Regeln auftritt, definieren wir *start* als neues Start-Symbol und fügen in der Grammatik die Regel

$$start \rightarrow expr$$

ein. Als erstes berechnen wir die Menge der Zustände Q. Wir hatten dafür oben die folgende Formel angegeben:

$$Q := \big\{\mathcal{M} \in 2^{\Gamma} \mid closure(\mathcal{M}) = \mathcal{M}\big\}.$$

Diese Menge enthält allerdings auch Zustände, die von dem Start-Zustand über die Funktion *goto*() gar nicht erreicht werden können. Wir berechnen daher nur die Zustände, die sich auch tatsächlich vom Start-Zustand mit Hilfe der Funktion *goto*() erreichen lassen. Damit die Rechnung nicht zu unübersichtlich wird, führen wir die folgenden Abküzungen ein:

$$S := start,\ E := expr,\ P := product,\ F := factor,\ N := \textsc{Number}.$$

Wir beginnen mit dem Start-Zustand:

$$\begin{array}{rcl}
s_0 & := & closure\big(\{\ \ S \rightarrow \bullet E\ \ \}\big) \\
& = & \{\ \ S \rightarrow \bullet E, \\
& & \ \ E \rightarrow \bullet E\ \text{“+”}\ P,\ E \rightarrow \bullet E\ \text{“-”}\ P,\ E \rightarrow \bullet P, \\
& & \ \ P \rightarrow \bullet P\ \text{“*”}\ F,\ P \rightarrow \bullet P\ \text{“/”}\ F,\ P \rightarrow \bullet F, \\
& & \ \ F \rightarrow \bullet\text{“(”}E\ \text{“)”}\ ,\ F \rightarrow \bullet N \qquad\qquad \}
\end{array}$$

Als nächstes berechnen wir $goto(s_0, F)$. Wir bezeichnen den resultierenden Zustand mit s_1.

$$\begin{array}{rcl}
s_1 & := & goto(s_0, F) \\
& = & closure(\{P \rightarrow F\bullet\}) \\
& = & \{P \rightarrow F\bullet\}.
\end{array}$$

Als nächstes berechnen wir $goto(s_0, N)$.

$$\begin{array}{rcl}
s_2 & := & goto(s_0, N) \\
& = & closure(\{F \rightarrow N\bullet\}) \\
& = & \{F \rightarrow N\bullet\}.
\end{array}$$

Als nächstes berechnen wir $goto(s_0, P)$.

$$\begin{array}{rcl}
s_3 & := & goto(s_0, P) \\
& = & closure(\{P \rightarrow P \bullet\ \text{“*”}\ F,\ P \rightarrow P \bullet\ \text{“/”}\ F,\ E \rightarrow P\bullet\}) \\
& = & \{P \rightarrow P \bullet\ \text{“*”}\ F,\ P \rightarrow P \bullet\ \text{“/”}\ F, E \rightarrow P\bullet\}.
\end{array}$$

Als nächstes berechnen wir $goto(s_0, E)$.

$$\begin{array}{rcl}
s_4 & := & goto(s_0, E) \\
& = & closure(\{S \rightarrow E\bullet,\ E \rightarrow E \bullet\ \text{“+”}\ P,\ E \rightarrow E \bullet\ \text{“-”}\ P\ \}) \\
& = & \{\ S \rightarrow E\bullet,\ E \rightarrow E \bullet\ \text{“+”}\ P,\ E \rightarrow E \bullet\ \text{“-”}\ P\ \}.
\end{array}$$

Als nächstes berechnen wir $goto(s_0,\ \text{“(”}\ )$.

$$\begin{array}{rcl}
s_5 & := & goto(s_0,\ \text{“(”}\ ) \\
& = & closure(\{F \rightarrow\ \text{“(”}\ \bullet E\ \text{“)”}\ \}) \\
& = & \{\ F \rightarrow\ \text{“(”}\ \bullet E\ \text{“)”} \\
& & \ \ E \rightarrow \bullet E\ \text{“+”}\ P,\ E \rightarrow \bullet E\ \text{“-”}\ P,\ E \rightarrow \bullet P, \\
& & \ \ P \rightarrow \bullet P\ \text{“*”}\ F,\ P \rightarrow \bullet P\ \text{“/”}\ F,\ P \rightarrow \bullet F, \\
& & \ \ F \rightarrow \bullet\text{“(”}E\ \text{“)”}\ ,\ F \rightarrow \bullet N \qquad\qquad \}.
\end{array}$$

Berechnen wir $goto(s_0,\ \text{“)”}\ )$, so finden wir

$$goto(s_0,\ \text{“)”}\ ) = \{\}.$$

Da dieser Zustand beim Parsen offenbar nicht benutzt werden kann, bekommt er keinen eigenen Namen. Als nächstes berechnen wir $goto(s_5, E)$.

$$\begin{array}{lcl} s_6 & := & goto(s_5, E) \\ & = & closure(\{\ F \rightarrow\ \text{“(”}\ E \bullet\ \text{“)”}\ ,\ E \rightarrow E \bullet\ \text{“+”}\ P,\ E \rightarrow E \bullet\ \text{“-”}\ P\ \}) \\ & = & \{\ F \rightarrow\ \text{“(”}\ E \bullet\ \text{“)”}\ ,\ E \rightarrow E \bullet\ \text{“+”}\ P,\ E \rightarrow E \bullet\ \text{“-”}\ P\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_6,\ \text{“)”}\ )$.

$$\begin{array}{lcl} s_7 & := & goto(s_6,\ \text{“)”}\ ) \\ & = & closure(\{\ F \rightarrow\ \text{“(”}\ E\ \text{“)”} \bullet\ \}) \\ & = & \{\ F \rightarrow\ \text{“(”}\ E\ \text{“)”} \bullet\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_4,\ \text{“+”}\ )$.

$$\begin{array}{lcl} s_8 & := & goto(s_4,\ \text{“+”}\ ) \\ & = & closure(\{E \rightarrow E\ \text{“+”}\ \bullet P\}) \\ & = & \{\ E \rightarrow E\ \text{“+”}\ \bullet P \\ & & \quad P \rightarrow \bullet P\ \text{“*”}\ F,\ P \rightarrow \bullet P\ \text{“/”}\ F,\ P \rightarrow \bullet F, \\ & & \quad F \rightarrow \bullet\text{“(”}E\ \text{“)”}\ ,\ F \rightarrow \bullet N \qquad\qquad \}. \end{array}$$

Als nächstes berechnen wir $goto(s_4,\ \text{“-”}\ )$.

$$\begin{array}{lcl} s_9 & := & goto(s_4,\ \text{“-”}\ ) \\ & = & closure(\{E \rightarrow E\ \text{“-”}\ \bullet P\}) \\ & = & \{\ E \rightarrow E\ \text{“-”}\ \bullet P \\ & & \quad P \rightarrow \bullet P\ \text{“*”}\ F,\ P \rightarrow \bullet P\ \text{“/”}\ F,\ P \rightarrow \bullet F, \\ & & \quad F \rightarrow \bullet\text{“(”}E\ \text{“)”}\ ,\ F \rightarrow \bullet N \qquad\qquad \}. \end{array}$$

Als nächstes berechnen wir $goto(s_9, P)$.

$$\begin{array}{lcl} s_{10} & := & goto(s_9, P) \\ & = & closure(\{E \rightarrow E\ \text{“-”}\ P\bullet,\ P \rightarrow P \bullet\ \text{“*”}\ F,\ P \rightarrow P \bullet\ \text{“/”}\ F\ \}) \\ & = & \{\ E \rightarrow E\ \text{“-”}\ P\bullet,\ P \rightarrow P \bullet\ \text{“*”}\ F,\ P \rightarrow P \bullet\ \text{“/”}\ F\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_3,\ \text{“/”}\ )$.

$$\begin{array}{lcl} s_{11} & := & goto(s_3,\ \text{“/”}\ ) \\ & = & closure(\{P \rightarrow P\ \text{“/”}\ \bullet F\ \}) \\ & = & \{\ P \rightarrow P\ \text{“/”}\ \bullet F,\ F \rightarrow \bullet\text{“(”}E\ \text{“)”},\ F \rightarrow \bullet N\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_3,\ \text{“*”}\ )$.

$$\begin{array}{lcl} s_{12} & := & goto(s_3,\ \text{“*”}\ ) \\ & = & closure(\{P \rightarrow P\ \text{“*”}\ \bullet F\ \}) \\ & = & \{\ P \rightarrow P\ \text{“*”}\ \bullet F,\ F \rightarrow \bullet\text{“(”}E\ \text{“)”},\ F \rightarrow \bullet N\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_{12}, F)$.

$$\begin{array}{lcl} s_{13} & := & goto(s_{12}, F) \\ & = & closure(\{P \rightarrow P\ \text{“*”}\ F\bullet\ \}) \\ & = & \{\ P \rightarrow P\ \text{“*”}\ F\bullet\ \}. \end{array}$$

Als nächstes berechnen wir $goto(s_{11}, F)$.

$$\begin{array}{lcl} s_{14} & := & goto(s_{11}, F) \\ & = & closure(\{P \to P \text{ “/” } F\bullet\}) \\ & = & \{\ P \to P \text{ “/” } F\bullet\ \}. \end{array}$$

Als letztes berechnen wir $goto(s_8, P)$.

$$\begin{array}{lcl} s_{15} & := & goto(s_8, P) \\ & = & closure(\{E \to E \text{ “+” } P\bullet,\ P \to P \bullet \text{ “*” } F,\ P \to P \bullet \text{ “/” } F\ \}) \\ & = & \{\ E \to E \text{ “+” } P\bullet,\ P \to P \bullet \text{ “*” } F,\ P \to P \bullet \text{ “/” } F\ \}. \end{array}$$

Weitere Rechnungen führen nicht mehr auf neue Zustände. Berechnen wir beispielsweise $goto(s_8, \text{ “(” })$, so finden wir

$$\begin{array}{ll} & goto(s_8, \text{ “(” }) \\ = & closure(\{F \to \text{ “(” } \bullet E \text{ “)” }\}) \\ = & \{\ F \to \text{ “(” } \bullet E \text{ “)” } \\ & \quad E \to \bullet E \text{ “+” } P,\ E \to \bullet E \text{ “-” } P,\ E \to \bullet P, \\ & \quad P \to \bullet P \text{ “*” } F,\ P \to \bullet P \text{ “/” } F,\ P \to \bullet F, \\ & \quad F \to \bullet \text{“(”} E \text{ “)” },\ F \to \bullet N \qquad\qquad \} \\ = & s_5. \end{array}$$

Damit ist die Menge der Zustände des Shift-Reduce-Parsers durch

$$Q := \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}\}$$

gegeben. Wir untersuchen als nächstes, ob es Konflikte gibt und betrachten exemplarisch die Menge s_{15}. Aufgrund der markierten Regel

$$P \to P \bullet \text{ “*” } F$$

muss im Zustand s_{15} geshiftet werden, wenn das nächste Token den Wert “*” hat. Auf der anderen Seite beinhaltet der Zustand s_{15} die Regel

$$E \to E \text{ “+” } P \bullet .$$

Diese Regel sagt, dass der Symbol-Stack mit der Grammatik-Regel $E \to E$ “+” P reduziert werden muss, falls in der Eingabe ein Zeichen aus der Menge $Follow(E)$ auftritt. Falls nun “*” $\in Follow(E)$ liegen würde, so hätten wir einen Shift-Reduce-Konflikt. Es gilt aber

$$Follow(E) = \{ \text{ “+” , “-” , “)” , “\$” } \}$$

und daraus folgt “*” $\notin Follow(E)$, so dass hier kein Shift-Reduce-Konflikt vorliegt. Eine Untersuchung der anderen Mengen zeigt, dass dort ebenfalls keine Shift-Reduce- oder Reduce-Reduce-Konflikte auftreten.

Als nächstes berechnen wir die Funktion $action()$. Wir betrachten exemplarisch zwei Fälle.

1. Als erstes berechnen wir $action(s_1, \text{ “+” })$. Es gilt

$$\begin{array}{lcl} action(s_1, \text{ “+” }) & = & action(\{P \to F\bullet\}, \text{ “+” }) \\ & = & \langle reduce, P \to F\rangle, \end{array}$$

   denn wir haben “+” $\in Follow(P)$.

2. Als nächstes berechnen wir $action(s_4, \text{ “+” })$. Es gilt

$$\begin{array}{lcl} action(s_4, \text{ “+” }) & = & action(\{S \to E\bullet,\ E \to E \bullet \text{ “+” } P,\ E \to E \bullet \text{ “-” } P\ \}, \text{ “+” }) \\ & = & \langle shift, closure(\{E \to E \text{ “+” } \bullet P\})\rangle \\ & = & \langle shift, s_8\rangle. \end{array}$$

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 13.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 13.1 und 13.2 übereinstimmen.

Aufgabe 18: Berechnen Sie die Menge der SLR-Zustände für die in Abbildung 13.9 gezeigte Grammatik und geben Sie die Funktionen *action*() und *goto*() an. Kürzen Sie die Namen der syntaktischen Variablen und Terminale mit S, C, D, L und I ab, wobei S für das neu eingeführte Start-Symbol steht.

Conjunction	→	*Conjunction* "`&`" *Disjunction*
	|	*Disjunction*
Disjunction	→	*Disjunction* "`|`" *Literal*
	|	*Literal*
Literal	→	"`!`" IDENTIFIER
	|	IDENTIFIER

Abbildung 13.9: Eine Grammatik für Boole'sche Ausdrücke in konjunktiver Normalform.

Hinweis: Damit Sie später Ihr Ergebnis mit meinem Ergebnis vergleichen können, ist es sinnvoll, die Namen der Zustände wie folgt festzulegen:

1. $s_0 := \mathit{closure}(\{S \to \bullet C\})$,
2. $s_1 := \mathit{goto}(s_0, C)$,
3. $s_2 := \mathit{goto}(s_0, \text{“!”})$,
4. $s_3 := \mathit{goto}(s_0, L)$,
5. $s_4 := \mathit{goto}(s_0, I)$,
6. $s_5 := \mathit{goto}(s_0, D)$,
7. $s_6 := \mathit{goto}(s_5, \text{“|”})$,
8. $s_7 := \mathit{goto}(s_6, L)$,
9. $s_8 := \mathit{goto}(s_2, I)$,
10. $s_9 := \mathit{goto}(s_1, \text{“\&”})$,
11. $s_{10} := \mathit{goto}(s_9, D)$.

13.3.1 Shift-Reduce- und Reduce-Reduce-Konflikte

In diesem Abschnitt untersuchen wir Shift-Reduce- und Reduce-Reduce-Konflikte genauer und betrachten dazu zwei Beispiele. Das erste Beispiel zeigt einen Shift-Reduce-Konflikt. Die in Abbildung 13.10 gezeigte Grammatik ist mehrdeutig, denn sie legt nicht fest, ob der Operator "`+`" stärker oder schwächer bindet als der Operator "`*`": Interpretieren wir das Nicht-Terminal N als eine Abküzung für NUMBER, so können wir mit dieser Grammatik den Ausdruck $1 + 2 * 3$ sowohl als

$$(1 + 2) * 3 \quad \text{als auch als} \quad 1 + (2 * 3)$$

lesen.

$$
\begin{array}{rcl}
E & \to & E \text{ “+” } E \\
& | & E \text{ “*” } E \\
& | & N
\end{array}
$$

Abbildung 13.10: Eine Grammatik mit Shift-Reduce-Konflikten.

Wir berechnen zunächst den Start-Zustand s_0.

$$
\begin{array}{rcl}
s_0 & = & \mathit{closure}\big(\{S \to \bullet E\}\big) \\
& = & \big\{S \to \bullet E,\ E \to \bullet E \text{ “+” } E,\ E \to \bullet E \text{ “*” } E,\ E \to \bullet N\ \big\}.
\end{array}
$$

Als nächstes berechnen wir $s_1 := \mathit{goto}(s_0, E)$:

$$
\begin{array}{rcl}
s_1 & = & \mathit{goto}(s_0, E) \\
& = & \mathit{closure}\big(\{S \to E\bullet,\ E \to E \bullet \text{ “+” } E,\ E \to E \bullet \text{ “*” } E\ \}\big) \\
& = & \{S \to E\bullet,\ E \to E \bullet \text{ “+” } E,\ E \to E \bullet \text{ “*” } E\ \}\big)
\end{array}
$$

Nun berechnen wir $s_2 := \mathit{goto}(s_1, \text{“+”})$:

$$
\begin{array}{rcl}
s_2 & = & \mathit{goto}(s_1, \text{“+”}) \\
& = & \mathit{closure}\big(\{E \to E \text{ “+” } \bullet E,\ \}\big) \\
& = & \{E \to E \text{ “+” } \bullet E,\ E \to \bullet E \text{ “+” } E,\ E \to \bullet E \text{ “*” } E,\ E \to \bullet N\ \}\big)
\end{array}
$$

Als nächstes berechnen wir $s_3 := \mathit{goto}(s_2, E)$:

$$
\begin{array}{rcl}
s_3 & = & \mathit{goto}(s_2, E) \\
& = & \mathit{closure}\big(\{E \to E \text{ “+” } E\bullet,\ E \to E \bullet \text{ “+” } E,\ E \to E \bullet \text{ “*” } E\}\big) \\
& = & \{E \to E \text{ “+” } E\bullet,\ E \to E \bullet \text{ “+” } E,\ E \to E \bullet \text{ “*” } E\ \}\big)
\end{array}
$$

Hier tritt bei der Berechnung von $\mathit{action}(s_3, \text{“*”})$ ein Shift-Reduce-Konflikt auf, denn einerseits verlangt die markierte Regel

$$E \to E \bullet \text{ “*” } E,$$

dass das Token “*” auf den Stack geschoben wird, andererseits haben wir

$$\mathit{Follow}(E) = \{ \text{ “+” , “*” } \},$$

so dass, falls das nächste zu lesende Token den Wert “*” hat, der Symbol-Stack mit der Regel

$$E \to E \text{ “+” } E\bullet$$

reduziert werden sollte.

Aufgabe 19: Bei der in Abbildung 13.10 gezeigten Grammatik treten noch weitere Shift-Reduce-Konflikte auf. Berechnen Sie alle Zustände und geben Sie dann die restlichen Shift-Reduce-Konflikte an.

Bemerkung: Es ist nicht weiter verwunderlich, dass wir bei der oben angegebenen Grammatik einen Konflikt gefunden haben, denn diese Grammatik ist nicht eindeutig. Demgegenüber ist kann gezeigt werden, dass jede SLR-Grammatik eindeutig sein muss. Folglich ist eine mehrdeutige Grammatik niemals eine SLR-Grammatik. Die Umkehrung dieser Aussage gilt jedoch nicht. Dies werden wir im nächsten Beispiel sehen. □

Wir untersuchen als nächstes eine Grammatik, die keine SLR-Grammatik ist, weil Reduce-Reduce-Konflikte auftreten. Wir betrachten dazu die in Abbildung 13.11 gezeigte Grammatik.

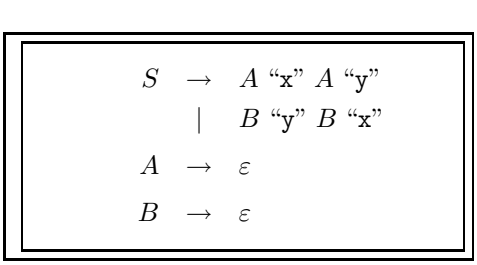


Abbildung 13.11: Eine Grammatik mit einem Reduce-Reduce-Konflikt.

Wir werden später sehen, dass diese Grammatik eindeutig ist. Wir berechnen zunächst den Start-Zustand eines SLR-Parsers für diese Grammatik.

$$
\begin{array}{lcl}
s_0 & = & \mathit{closure}\big(\{\widehat{S} \to \bullet S\}\big) \\
& = & \big\{\widehat{S} \to \bullet S,\ S \to \bullet A\ \texttt{“x”}\ A\ \texttt{“y”}\ ,\ S \to \bullet B\ \texttt{“y”}\ B\ \texttt{“x”}\ ,\ A \to \bullet\varepsilon,\ B \to \bullet\varepsilon\ \big\} \\
& = & \big\{\widehat{S} \to \bullet S,\ S \to \bullet A\ \texttt{“x”}\ A\ \texttt{“y”}\ ,\ S \to \bullet B\ \texttt{“y”}\ B\ \texttt{“x”}\ ,\ A \to \varepsilon\bullet,\ B \to \varepsilon\bullet\ \big\},
\end{array}
$$

denn $A \to \bullet\varepsilon$ ist das Selbe wie $A \to \varepsilon\bullet$. In diesem Zustand gibt es einen Reduce-Reduce-Konflikt zwischen den beiden markierten Regeln

$$A \to \bullet\varepsilon \quad \text{und} \quad B \to \varepsilon \bullet .$$

Dieser Konflikt tritt bei der Berechnung von

$$\mathit{action}(s_0,\ \texttt{“x”}\ )$$

auf, denn wir haben

$$\mathit{Follow}(A) = \big\{\texttt{“x”}, \texttt{“y”}\big\} = \mathit{Follow}(B)$$

und damit ist dann nicht klar, mit welcher dieser Regeln der Parser die Eingabe im Zustand s_0 reduzieren soll, wenn das nächste gelesene Token den Wert “x” hat, denn dieses Token ist sowohl ein Element der Menge $\mathit{Follow}(A)$ als auch der Menge $\mathit{Follow}(B)$.

Es ist interessant zu bemerken, dass die obige Grammatik die $LL(1)$-Eigenschaft hat, denn es gilt

$$\mathit{First}(A\ \texttt{“x”}\ A\ \texttt{“y”}\ ) = \{\ \texttt{“x”}\ \},\quad \mathit{First}(B\ \texttt{“y”}\ B\ \texttt{“x”}\ ) = \{\ \texttt{“y”}\ \}$$

und daraus folgt sofort

$$\mathit{First}(A\ \texttt{“x”}\ A\ \texttt{“y”}\ ) \cap \mathit{First}(B\ \texttt{“y”}\ B\ \texttt{“x”}\ ) = \{\texttt{“x”}\} \cap \{\texttt{“y”}\} = \{\}.$$

Dieses Beispiel zeigt, dass SLR-Grammatiken im Allgemeinen nicht ausdruckstärker sind als LL(1)-Grammatiken. In der Praxis zeigt sich jedoch, dass viele Grammatiken, die nicht die LL(1)-Eigenschaft haben, SLR-Grammatiken sind.

13.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 13.11 gezeigten Grammatik auftritt, kann wie folgt gelöst werden: In dem Zustand

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{\widehat{S} \to \bullet S\}\big) \\ & = & \big\{\widehat{S} \to \bullet S,\ S \to \bullet A\ \text{“}\texttt{x}\text{”}\ A\ \text{“}\texttt{y}\text{”}\ ,\ S \to \bullet B\ \text{“}\texttt{y}\text{”}\ B\ \text{“}\texttt{x}\text{”}\ ,\ A \to \varepsilon\bullet,\ B \to \varepsilon\bullet\ \big\} \end{array}$$

kommen die markierten Regeln $A \to \varepsilon\bullet$ und $B \to \varepsilon\bullet$ von der Berechnung des Abschlusses der Regeln

$$S \to \bullet A\ \text{“}\texttt{x}\text{”}\ A\ \text{“}\texttt{y}\text{”} \quad \text{und} \quad S \to \bullet B\ \text{“}\texttt{y}\text{”}\ B\ \text{“}\texttt{x}\text{”}\ .$$

Bei der ersten Regel ist klar, dass auf das erste A ein “x” folgen muss, bei der zweiten Regel sehen wir, dass auf das erste B ein “y” folgt. Diese Information geht über die Information hinaus, die in den Mengen *Follow*(A) bzw. *Follow*(B) enthalten ist, denn jetzt berücksichtigen wir den Kontext, in dem die syntaktische Variable auftaucht. Damit können wir die Funktion *action*(s_0, “x”) und *action*(s_0, “y”) wie folgt definieren:

$$\mathit{action}(s_0, \text{“}\texttt{x}\text{”}) = \langle \texttt{reduce}, A \to \varepsilon\rangle \quad \text{und} \quad \mathit{action}(s_0, \text{“}\texttt{y}\text{”}) = \langle \texttt{reduce}, B \to \varepsilon\rangle.$$

Durch diese Definition wird der Reduce-Reduce-Konflikt gelöst. Die zentrale Idee ist, bei der Berechnung des Abschlusses den Kontext, in dem eine Regel auftritt, miteinzubeziehen. Dazu erweitern wir zunächst die Definition einer markierten Regel.

Definition 40 (erweiterte markierte Regel) Eine *erweiterte markierte Regel* (abgekürzt: *e.m.R.*) einer Grammatik $G = \langle V, T, R, S\rangle$ ist ein Quadrupel

$$\langle A, \alpha, \beta, L\rangle,$$

wobei gilt:

1. $(A \to \alpha\beta) \in R$.
2. $L \subseteq T$.

Wir schreiben die erweiterte markierte Regel $\langle A, \alpha, \beta, L\rangle$ als

$$A \to \alpha \bullet \beta : L.$$

Falls L nur aus einem Element t besteht, falls also $L = \{t\}$ gilt, so lassen wir die Mengen-Klammern weg und schreiben die Regel als

$$A \to \alpha \bullet \beta : t.$$ □

Anschaulich interpretieren wir die e.m.R. $A \to \alpha \bullet \beta : L$ als einen Zustand, in dem folgendes gilt:

1. Der Parser versucht, ein A mit Hilfe der Grammatik-Regel $A \to \alpha\beta$ zu erkennen.
2. Dabei wurde bereits α erkannt. Damit die Regel $A \to \alpha\beta$ angewendet werden kann, muss nun β erkannt werden.
3. Auf das A folgt ein Token aus der Menge L.

   Die Menge L bezeichnen wir daher als die Menge der *Folge-Token*.

Mit erweiterten markierten Regeln arbeitet sich ganz ähnlich wie mit markierten Regeln, allerdings müssen wir die Definitionen der Funktionen *closure*(), *goto* und *action*() etwas modifizieren. Wir beginnen mit der Funktion *closure*().

Definition 41 (*closure*($\mathcal{M}$)) Es sei $\mathcal{M}$ eine Menge erweiterter markierter Regeln. Dann definieren wir den *Abschluss* von M als die kleinste Menge $\mathcal{K}$ markierter Regeln, für die folgendes gilt:

1. $\mathcal{M} \subseteq \mathcal{K}$,

   der Abschluss umfasst also die ursprüngliche Regel-Menge.

2. Ist einerseits

   $$A \to \alpha \bullet B\beta : L$$

   eine e.m.R. aus der Menge $\mathcal{K}$, wobei B eine syntaktische Variable ist, und ist andererseits

   $$B \to \gamma$$

   eine Grammatik-Regel der zu Grunde liegenden Grammatik G, so ist auch die e.m.R.

   $$B \to \bullet\gamma : \bigcup\{\mathit{First}(\beta t) \mid t \in L\}$$

   ein Element der Menge $\mathcal{K}$.

Die so definierte eindeutig bestimmte Menge $\mathcal{K}$ wird wieder mit $\mathit{closure}(\mathcal{M})$ bezeichnet. □

Bemerkung: Gegenüber der alten Definition ist nur die Berechnung der Menge der Folge-Token hinzu gekommen. Der Kontext, in dem das B auftritt, das mit der Regel $B \to \gamma$ erkannt werden soll, ist zunächst durch den String β gegeben, der in der Regel $A \to \alpha \bullet B\beta : L$ auf das B folgt. Möglicherweise leitet β den leeren String ε ab. In diesen Fall spielen auch die Folge-Token aus der Menge L eine Rolle, denn falls $\beta \to^* \varepsilon$ gilt, kann auf das B auch ein Folge-Token t aus der Menge L folgen. □

Für eine gegebene e.m.R.-Menge $\mathcal{M}$ kann die Berechnung von $\mathcal{K} := \mathit{closure}(\mathcal{M})$ iterativ erfolgen. Abbildung 13.12 zeigt die Berechnung von $\mathit{closure}(\mathcal{M})$. Der wesentliche Unterschied gegenüber der früheren Berechnung von *closure*() ist, dass wir bei den e.m.R.s, die wir für eine Variable B mit in $\mathit{closure}(\mathcal{M})$ aufnehmen, bei der Menge der Folge-Token den Kontext berücksichtigen, in dem B auftritt. Dadurch gelingt es, die Zustände des Parsers präziser zu beschreiben, als dies bei markierten Regeln der Fall ist.

```
1   procedure closure(M) {
2       K  := M;
3       K⁻ := {};
4       while (K⁻ ≠ K) {
5           K⁻ := K;
6           K  := K ∪ {(B → •γ : ⋃{First(βt) | t ∈ L}) |
7                       (A → α • Bβ : L) ∈ K ∧ (B → γ) ∈ R };
8       }
9       return K;
10  }
```

Abbildung 13.12: Berechnung von $\mathit{closure}(\mathcal{M})$

Bemerkung: Der Ausdruck $\bigcup\{\mathit{First}(\beta t) \mid t \in L\}$ sieht komplizierter aus, als er tatsächlich ist. Wollen wir diesen Ausdruck berechnen, so ist es zweckmäßig eine Fallunterscheidung danach durchzuführen, ob β den leeren String ε ableiten kann oder nicht, denn es gilt

$$\bigcup\{\mathit{First}(\beta t) \mid t \in L\} = \begin{cases} \mathit{First}(\beta) \cup L & \text{falls } \beta \Rightarrow^* \varepsilon; \\ \mathit{First}(\beta) & \text{sonst.} \end{cases}$$

Die Berechnung von $\mathit{goto}(\mathcal{M}, t)$ für eine Menge $\mathcal{M}$ von erweiterten Regeln und ein Zeichen X ändert sich gegenüber der Berechnung im Falle einfacher markierter Regeln nur durch das Anfügen der Menge von *Folge-Tokens*, die aber selbst unverändert bleibt:

$$\mathit{goto}(\mathcal{M}, X) := \mathit{closure}\Big(\big\{A \to \alpha X \bullet \beta : L \mid (A \to \alpha \bullet X\beta : L) \in \mathcal{M}\big\}\Big).$$

Genau wie bei der Theorie der SLR-Parser augmentieren wir unsere Grammatik G, indem wir der Menge der Variable eine neue Start-Variable $\widehat{S}$ und der Menge der Regeln die neue Regel $\widehat{S} \to S$ hinzufügen. Dann hat der Start-Zustand die Form

$$q_0 := \mathit{closure}\big(\big\{\widehat{S} \to \bullet S : \textsc{Eof}\big\}\big),$$

denn auf das Start-Symbol muss das Datei-Ende "Eof" folgen. Als letztes zeigen wir, wie die Definition der Funktion *action*() geändert werden muss. Wir spezifizieren die Berechnung dieser Funktion durch die folgenden bedingten Gleichungen.

1. $(A \to \alpha \bullet t\beta : L) \in \mathcal{M} \implies \mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t)\rangle$.
2. $(A \to \alpha\bullet : L) \in \mathcal{M} \;\wedge\; A \neq \widehat{S} \;\wedge\; t \in L \implies \mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, A \to \alpha\rangle$.
3. $(\widehat{S} \to S\bullet : \textsc{Eof}) \in \mathcal{M} \implies \mathit{action}(\mathcal{M}, \textsc{Eof}) := \texttt{accept}$.
4. Sonst: $\mathit{action}(\mathcal{M}, t) := \texttt{error}$.

Falls es bei diesen Gleichungen zu einem Konflikt kommt, weil gleichzeitig die Bedingung der ersten Gleichung als auch die Bedingung der zweiten Gleichung erfüllt ist, so sprechen wir wieder von einem *Shift-Reduce-Konflikt*. Ein Shift-Reduce-Konflikt liegt also bei der Berechnung von $\mathit{action}(\mathcal{M}, t)$ dann vor, wenn es zwei e.m.R.s

$$(A \to \alpha \bullet t\beta : L_1) \in \mathcal{M} \quad \text{und} \quad (B \to \gamma\bullet : L_2) \in \mathcal{M} \quad \text{mit } t \in L_2$$

gibt, denn dann ist nicht klar, ob im Zustand $\mathcal{M}$ das Token t auf den Stack geschoben werden soll, oder ob statt dessen der Symbol-Stack mit der Regel $B \to \gamma$ reduziert werden muss.

Bemerkung: Gegenüber einem SLR-Parser ist die Möglichkeit von Shift-Reduce-Konflikten verringert, denn bei einem SLR-Parser liegt bereits dann ein Shift-Reduce-Konflikt vor, wenn $t \in \mathit{Follow}(B)$ gilt und die Menge L_2 ist in der Regel kleiner als die Menge $\mathit{Follow}(B)$.

Ein *Reduce-Reduce-Konflikt* liegt vor, wenn es zwei e.m.R.s

$$(A \to \alpha\bullet : L_1) \in \mathcal{M} \quad \text{und} \quad (B \to \beta\bullet : L_2) \in \mathcal{M} \quad \text{mit } L_1 \cap L_2 \neq \{\}$$

gibt, denn dann ist nicht klar, mit welcher dieser beiden Regeln der Symbol-Stack reduziert werden soll, wenn das nächste Token ein Element der Schnittmenge $L_1 \cap L_2$ ist.

Bemerkung: Gegenüber einem SLR-Parser ist die Möglichkeit von Reduce-Reduce-Konflikten verringert, denn bei einem SLR-Parser liegt bereits dann ein Reduce-Reduce-Konflikt vor, wenn es ein t in der Menge $\mathit{Follow}(A) \cap \mathit{Follow}(B)$ gibt und die *Follow*-Mengen sind oft größer als die Mengen L_1 und L_2.

Bemerkung: Neben den oben angegebenen Konflikten ist auch ein Konflikt zwischen der 1. und der 3. Regel möglich. Da die 3. Regel als ein Speziallfall der 2. Regel angesehen werden kann, sprechend wir dann ebenfalls von einem Shift-Reduce-Konflikt. Darüber hinaus sind noch Konflikte zwischen der 2. und der 3. Regel möglich. Derartige Konflikte werden als Reduce-Reduce-Konflikte bezeichnet.

Beispiel: Wir greifen das Beispiel der in Abbildung 13.11 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände. Um die Schreibweise zu vereinfachen, schreiben wir an Stelle von "Eof" kürzer "$".

1. $\begin{aligned} s_0 &:= \mathit{closure}\Big(\big\{\widehat{S} \to \bullet S : \$\big\}\Big) \\ &= \big\{\widehat{S} \to \bullet S : \$, S \to \bullet A\text{“}\texttt{x}\text{”}A\text{“}\texttt{y}\text{”} : \$, S \to \bullet B\text{“}\texttt{y}\text{”}B\text{“}\texttt{x}\text{”} : \$, A \to \bullet : \text{“}\texttt{x}\text{”}, B \to \bullet : \text{“}\texttt{y}\text{”}\big\}. \end{aligned}$
2. $\begin{aligned} s_1 &:= \mathit{goto}(s_0, A) \\ &= \mathit{closure}\Big(\big\{S \to A \bullet \text{“}\texttt{x}\text{”}A\text{“}\texttt{y}\text{”} : \$\big\}\Big) \\ &= \big\{S \to A \bullet \text{“}\texttt{x}\text{”}A\text{“}\texttt{y}\text{”} : \$\big\}. \end{aligned}$

3. $s_2 := goto(s_0, S)$
$= closure\Big(\big\{\widehat{S} \to S\bullet : \$\big\}\Big)$
$= \big\{\widehat{S} \to S\bullet : \$\big\}.$

4. $s_3 := goto(s_0, B)$
$= closure\Big(\big\{S \to B \bullet \text{“y”}\, B \text{“x”} : \$\big\}\Big)$
$= \big\{S \to B \bullet \text{“y”}\, B \text{“x”} : \$\big\}.$

5. $s_4 := goto(s_3, \text{“y”})$
$= closure\Big(\big\{S \to B\text{“y”} \bullet B \text{“x”} : \$\big\}\Big)$
$= \big\{S \to B\text{“y”} \bullet B \text{“x”} : \$, B \to \bullet : \text{“x”}\big\}.$

6. $s_5 := goto(s_4, B)$
$= closure\Big(\big\{S \to B\text{“y”}\, B \bullet \text{“x”} : \$\big\}\Big)$
$= \big\{S \to B\text{“y”}\, B \bullet \text{“x”} : \$\big\}.$

7. $s_6 := goto(s_5, \text{“x”})$
$= closure\Big(\big\{S \to B\text{“y”}\, B\text{“x”}\bullet : \$\big\}\Big)$
$= \big\{S \to B\text{“y”}\, B\text{“x”}\bullet : \$\big\}.$

8. $s_7 := goto(s_1, \text{“x”})$
$= closure\Big(\big\{S \to A\text{“x”} \bullet A\text{“y”} : \$\big\}\Big)$
$= \big\{S \to A\text{“x”} \bullet A\text{“y”} : \$, A \to \bullet : \text{“y”}\big\}.$

9. $s_8 := goto(s_7, A)$
$= closure\Big(\big\{S \to A\text{“x”}\, A \bullet \text{“y”} : \$\big\}\Big)$
$= \big\{S \to A\text{“x”}\, A \bullet \text{“y”} : \$\big\}.$

10. $s_9 := goto(s_8, \text{“y”})$
$= closure\Big(\big\{S \to A\text{“x”}\, A\text{“y”}\bullet : \$\big\}\Big)$
$= \big\{S \to A\text{“x”}\, A\text{“y”}\bullet : \$\big\}.$

Als nächstes untersuchen wir, ob es bei den Zuständen Konflikte gibt. Beim Start-Zustand s_0 hatten wir im letzten Abschnitt einen Reduce-Reduce-Konflikt zwischen den beiden Regeln $A \to \varepsilon$ und $B \to \varepsilon$ gefunden, weil

$$\mathit{Follow}(A) \cap \mathit{Follow}(B) = \{\text{“x”}, \text{“y”}\} \neq \{\}$$

gilt. Dieser Konflikt ist nun verschwunden, denn zwischen den e.m.R.s

$$A \to \bullet : \text{“x”} \quad \text{und} \quad B \to \bullet : \text{“y”}$$

gibt es wegen “x” $\neq$ “y” keinen Konflikt. Es ist leicht zu sehen, dass auch bei den anderen Zustände keine Konflikte auftreten.

Aufgabe 20: Berechnen Sie die Menge der Zustände eines LR-Parsers für die folgende Grammatik:

$$\begin{array}{lcl} E & \to & E \text{“+”}\ P \\ & \mid & P \\ P & \to & P \text{“+”}\ F \\ & \mid & F \\ F & \to & \text{“(”}E \text{“)”} \\ & \mid & \mathit{Number} \end{array}$$

Bemerkung: Die Theorie der kanonischen LR-Parser geht auf Donald E. Knuth zurück, der diese Theorie 1965 entwickelt hat [Knu65].

13.5 LALR-Parser

Die Zahl der Zustände eines LR-Parsers ist oft erheblich größer als die Zahl der Zustände, die ein SLR-Parser der selben Grammatik hätte. Anfangs, als der zur Verfügung stehenden Haupt-Speicher der meisten Rechner noch bescheidener dimensioniert waren, als dies heute der Fall ist, hatten LR-Parser daher eine inakzeptable Größe. Eine genaue Analyse der Menge der Zustände von LR-Parsern zeigte, dass es oft möglich ist, bestimmte Zustände zusammen zu fassen. Dadurch kann die Menge der Zustände in den meisten Fällen deutlich verkleinert werden. Wir illustrieren das Konzept an einem Beispiel und betrachten die in Abbildung 13.13 gezeigt Grammatik, die ich dem *Drachenbuch* [ASUL06] entnommen habe. (Das "Drachenbuch" ist das Standardwerk im Bereich Compilerbau.)

$$
\begin{array}{rcl}
\widehat{S} & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & \text{“x”}\ C \\
 & | & \text{“y”}
\end{array}
$$

Abbildung 13.13: Eine Grammatik aus dem Drachenbuch.

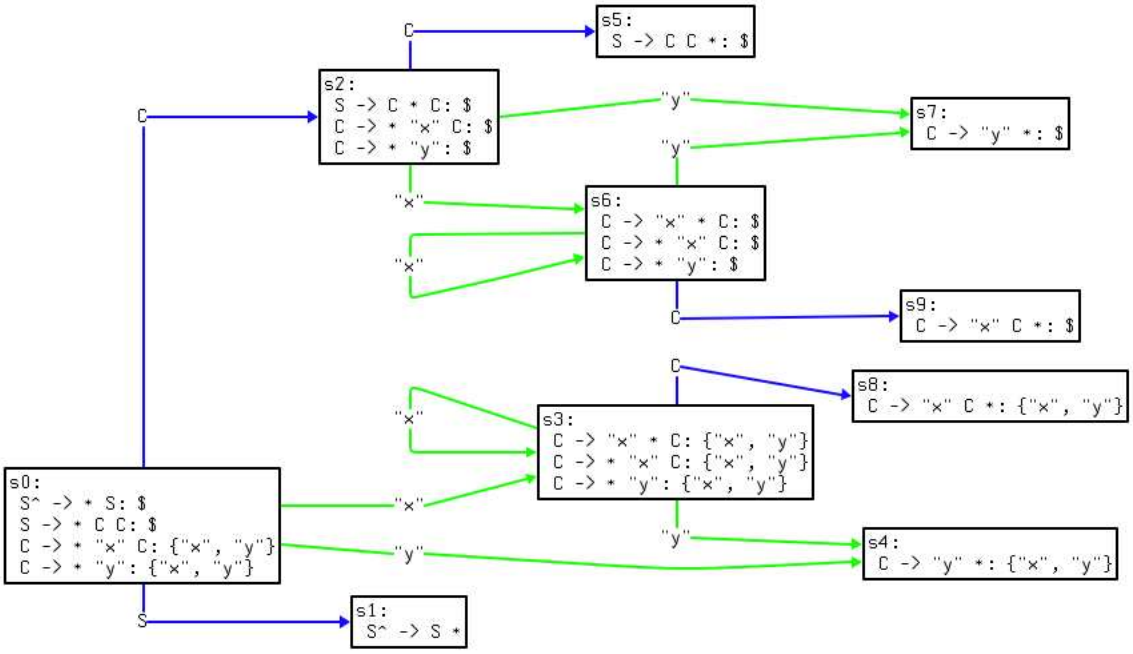


Abbildung 13.14: LR-Goto-Graph für die Grammatik aus Abbildung 13.13.

Abbildung 13.14 zeigt den sogenannten *LR-Goto-Graphen* für diese Grammatik. Die Knoten dieses Graphen sind die Zustände. Betrachten wir den LR-Goto-Graphen, so stellen wir fest, dass die Zustände s_6 und s_3 sich nur in den Mengen der Folge-Token unterscheiden, denn es gilt

einerseits

$$s_6 = \Big\{ S \to \text{“x”} \bullet C : \text{“\$”}, C \to \bullet \text{“x”}\ C : \text{“\$”}, C \to \bullet \text{“y”}\ : \text{“\$”} \Big\},$$

und andererseits haben wir

$$s_3 = \Big\{ S \to \text{“x”} \bullet C : \{\text{“x”}, \text{“y”}\}, C \to \bullet \text{“x”}\ C : \{\text{“x”}, \text{“y”}\}, C \to \bullet \text{“y”}\ : \{\text{“x”}, \text{“y”}\} \Big\}.$$

Offenbar entsteht die Menge s_3 aus der Menge s_6 indem überall “$” durch die Menge {“x”, “y”} ersetzt wird. Genauso kann die Menge s_7 in s_4 und s_9 in s_8 überführt werden. Die entscheidende Erkenntnis ist nun, dass die Funktion *goto*() unter dieser Art von Transformation invariant ist, denn bei der Definition dieser Funktion spielt die Menge der Folge-Token keine Rolle. So sehen wir zum Beispiel, dass einerseits

$$goto(s_3, C) = s_8 \quad \text{und} \quad \text{und } goto(s_6, C) = s_9$$

gilt und dass andererseits der Zustand s_9 in den Zustand s_8 übergeht, wenn wir überall in s_9 das Terminal “$” durch die Menge {“x”, “y”} ersetzen. Definieren wir den *Kern* einer Menge von erweiterten markierten Regeln dadurch, dass wir in jeder Regel die Menge der Folgetoken wegstreichen, und fassen dann Zustände mit dem selben Kern zusammen, so erhalten wir den in Abbildung 13.15 gezeigten Goto-Graphen.

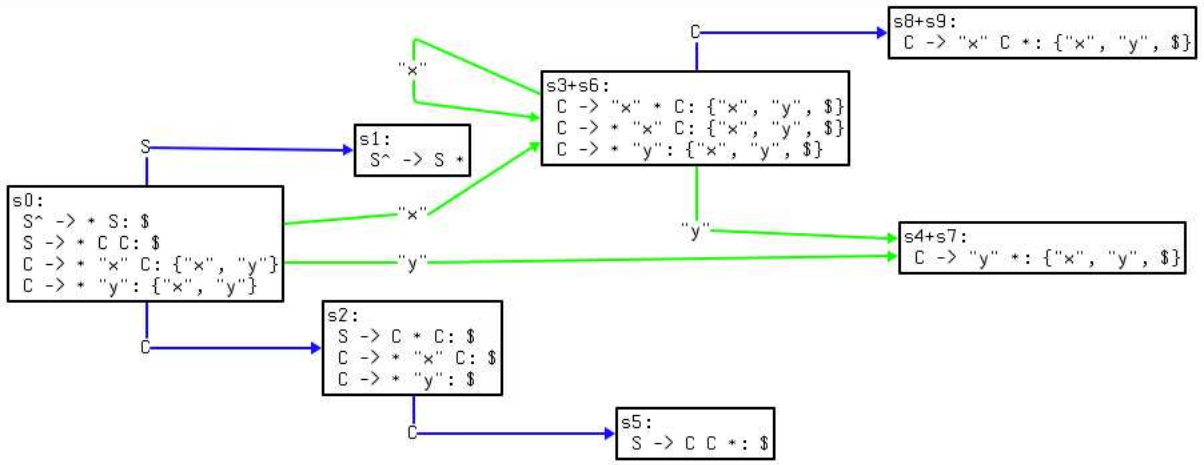


Abbildung 13.15: Der LALR-Goto-Graph für die Grammatik aus Abbildung 13.13.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 13.13 gezeigten Grammatik gemacht gaben, verallgemeinern und formalisieren zu können, definieren wir ein Funktion *core*(), die den Kern einer Menge von e.m.R.s berechnet und also diese Menge in eine Menge markierter Regeln überführt:

$$core(\mathcal{M}) := \{A \to \alpha \bullet \beta \mid (A \to \alpha \bullet \beta : L) \in \mathcal{M}\}.$$

Die Funktion *core*() entfernt also einfach die Menge der Folge-Tokens von den e.m.R.s. Wir hatten die Funktion *goto*() für eine Menge $\mathcal{M}$ von erweiterten markierten Regeln und ein Symbol X durch

$$goto(\mathcal{M}, X) := closure\Big(\big\{A \to \alpha X \bullet \beta : L \mid (A \to \alpha \bullet X\beta : L) \in \mathcal{M}\big\}\Big).$$

definiert. Offenbar spielt die Menge der Folge-Token bei der Berechnung von $goto(\mathcal{M}, X)$ keine Rolle, formal gilt für zwei e.m.R.-Mengen $\mathcal{M}_1$ und $\mathcal{M}_2$ und ein Symbol X die Formel:

$$core(\mathcal{M}_1) = core(\mathcal{M}_1) \ \Rightarrow\ core(goto(\mathcal{M}_1, X)) = core(goto(\mathcal{M}_2, X)).$$

Für zwei e.m.R.-Mengen $\mathcal{M}$ und $\mathcal{N}$, die den gleichen Kern haben, definieren wir die *erweiterte Vereinigung* $\mathcal{M} \uplus \mathcal{N}$ von $\mathcal{M}$ und $\mathcal{N}$ als

$$\mathcal{M} \uplus \mathcal{N} := \{A \to \alpha \bullet \beta : K \cup L \mid (A \to \alpha \bullet \beta : K) \in \mathcal{M}\ \wedge\ (A \to \alpha \bullet \beta : L) \in \mathcal{N}\}.$$

Diese Definition verallgemeinern wir zu einer Operation $\biguplus$, die auf einer Menge von Mengen von e.m.R.s definiert ist: Ist $\mathfrak{I}$ eine Menge von Mengen von e.m.R.s, die alle den gleichen Kern haben, gilt also

$$\mathfrak{I} = \{\mathcal{M}_1, \cdots, \mathcal{M}_k\} \quad \text{mit} \quad \mathit{core}(\mathcal{M}_i) = \mathit{core}(\mathcal{M}_j) \quad \text{für alle } i, j \in \{1, \cdots, k\},$$

so definieren wir

$$\biguplus \mathfrak{I} := \mathcal{M}_1 \uplus \cdots \uplus \mathcal{M}_k.$$

Es sei nun Δ die Menge aller Zustände eines LR-Parsers. Dann ist die Menge der Zustände des entsprechenden LALR-Parsers durch die erweiterte Vereinigung der Menge aller der Teilmengen von Δ gegeben, deren Elemente den gleichen Kern haben:

$$\mathfrak{Q} := \left\{ \biguplus \mathfrak{I} \mid \mathfrak{I} \in 2^{\Delta} \wedge \forall \mathcal{M}, \mathcal{N} \in \mathfrak{I} : \mathit{core}(\mathcal{M}) = \mathit{core}(\mathcal{N}) \wedge \text{und } \mathfrak{I} \text{ maximal} \right\}.$$

Die Forderung "$\mathfrak{I}$ maximal" drückt in der obigen Definition aus, dass in $\mathfrak{I}$ tatsächlich alle Mengen aus Δ zusamengefasst sind, die den selben Kern haben. Die so definierte Menge $\mathfrak{Q}$ ist die Menge der LALR-Zustände. Die Berechnung der Funktionen *goto*() und *accept*() ändert sich gegenüber den LR-Parsern nicht.

13.6 Vergleich von SLR-, LR- und LALR-Parsern

Wir wollen nun die verschiedenen Methoden, mit denen wir in diesem Kapitel Shift-Reduce-Parser konstruiert haben, vergleichen. Wir nennen eine Sprache $\mathcal{L}$ eine *SLR-Sprache*, wenn $\mathcal{L}$ von einem SLR-Parser erkannt werden kann. Die Begriffe *kanonische LR-Sprache* und *LALR-Sprache* werden analog definiert. Zwischen diesen Sprachen bestehen die folgende Beziehungen:

$$\textit{SLR-Sprache} \subsetneq \textit{LALR-Sprache} \subsetneq \textit{kanonische LR-Sprache} \tag{$\star$}$$

Diese Inklusionen sind leicht zu verstehen: Bei der Definition der LR-Parser hatten wir zu den markierten Regeln Mengen von Folge-Token hinzugefügt. Dadurch war es möglich, in bestimmten Fällen Shift-Reduce- und Reduce-Reduce-Konflikte zu vermeiden. Da die Zustands-Mengen der kanonischen LR-Parser unter Umständen sehr groß werden können, hatten wir dann wieder solche Mengen von erweiterten markierten Regeln zusammen gefaßt, für die die Menge der Folge-Token identisch war. So hatten wir die LALR-Parser erhalten. Durch die Zusammenfassung von Regel-Menge können wir uns allerdings in bestimmten Fällen Reduce-Reduce-Konflikte einhandeln, so dass die Menge der LALR-Sprachen eine Untermenge der kanonischen LR-Sprachen ist.

Wir werden in den folgenden Unterabschnitten zeigen, dass die Inklusionen in ($\star$) echt sind.

13.6.1 *SLR-Sprache* $\subsetneq$ *LALR-Sprache*

Die Zustände eines LALR-Parsers enthalten gegenüber den Zuständen eines SLR-Parsers noch Mengen von Folge-Token. Damit sind LALR-Parser mindestens genauso mächtig wie SLR-Parser. Wir zeigen nun, dass LALR-Parser tatsächlich mächtiger als SLR-Parser sind. Um diese Behauptung zu belegen, präsentieren wir eine Grammatik, für die es zwar einen LALR-Parser, aber keinen SLR-Parser gibt. Wir hatten auf Seite 201 gesehen, dass die Grammatik

$$S \;\rightarrow\; A \text{ “}\texttt{x}\text{”} \; A \text{ “}\texttt{y}\text{”} \;\mid B \text{ “}\texttt{y}\text{”} \; B \text{ “}\texttt{x}\text{”} \;, \quad A \;\rightarrow\; \varepsilon, \quad B \;\rightarrow\; \varepsilon$$

keine SLR-Grammatik ist. Später hatten wir gesehen, dass diese Grammatik von einem kanonischen LR-Parser geparst werden kann. Wir zeigen nun, dass diese Grammatik auch von einem LALR-Parser geparst werden kann. Dazu berechnen wir die Menge der LALR-Zustände. Dazu ist zunächst die Menge der kanonischen LR-Zustände zu berechnen. Diese Berechnung hatten wir bereits früher durchgeführt und dabei die folgenden Zustände erhalten:

1. $s_0 = \big\{ \widehat{S} \rightarrow \bullet S : \$, S \rightarrow \bullet A \text{“}\texttt{x}\text{”} A \text{“}\texttt{y}\text{”} : \$, S \rightarrow \bullet B \text{“}\texttt{y}\text{”} B \text{“}\texttt{x}\text{”} : \$, A \rightarrow \bullet : \text{“}\texttt{x}\text{”}, B \rightarrow \bullet : \text{“}\texttt{y}\text{”} \big\}$,

2. $s_1 = \{S \to A \bullet \text{“}\texttt{x}\text{”} A \text{“}\texttt{y}\text{”} : \$\}$,
3. $s_2 = \{\widehat{S} \to S\bullet : \$\}$,
4. $s_3 = \{S \to B \bullet \text{“}\texttt{y}\text{”} B \text{“}\texttt{x}\text{”} : \$\}$,
5. $s_4 = \{S \to B \text{“}\texttt{y}\text{”} \bullet B \text{“}\texttt{x}\text{”} : \$, B \to \bullet : \text{“}\texttt{x}\text{”}\}$,
6. $s_5 = \{S \to B \text{“}\texttt{y}\text{”} B \bullet \text{“}\texttt{x}\text{”} : \$\}$,
7. $s_6 = \{S \to B \text{“}\texttt{y}\text{”} B \text{“}\texttt{x}\text{”} \bullet : \$\}$,
8. $s_7 = \{S \to A \text{“}\texttt{x}\text{”} \bullet A \text{“}\texttt{y}\text{”} : \$, A \to \bullet : \text{“}\texttt{y}\text{”}\}$,
9. $s_8 = \{S \to A \text{“}\texttt{x}\text{”} A \bullet \text{“}\texttt{y}\text{”} : \$\}$,
10. $s_9 = \{S \to A \text{“}\texttt{x}\text{”} A \text{“}\texttt{y}\text{”} \bullet : \$\}$.

Wir stellen fest, dass die Kerne aller hier aufgelisteten Zustände verschieden sind. Damit stimmt bei dieser Grammatik die Menge der Zustände des LALR-Parser mit der Menge der Zustände des kanonischen LR-Parsers überein. Daraus folgt, dass es auch bei den LALR-Zuständen keine Konflikte gibt, denn beim Übergang von kanonischen LR-Parsern zu LALR-Parsern haben wir lediglich Zustände mit gleichem Kern zusammengefasst, die Definition der Funktionen *goto*() und *action*() blieb unverändert.

13.6.2 *LALR-Sprache $\subsetneq$ kanonische LR-Sprache*

Wir hatten LALR-Parser dadurch definiert, dass wir verschiedene Zustände eines kanonischen LR-Parsers zusammen gefaßt haben. Damit ist klar, dass kanonische LR-Parser mindestens so mächtig sind wie LALR-Parser. Um zu zeigen, dass kanonische LR-Parser tatsächlich mächtiger sind als LALR-Parser, benötigen wir eine Grammatik, für die sich zwar ein kanonischer LR-Parser, aber kein LALR-Parser erzeugen läßt. Abbildung 13.16 zeigt eine solche Grammatik, die ich dem Drachenbuch entnommen habe.

$$
\begin{array}{lcl}
S & \to & \text{“}\texttt{v}\text{”}\ A\ \text{“}\texttt{y}\text{”} \\
 & | & \text{“}\texttt{w}\text{”}\ B\ \text{“}\texttt{y}\text{”} \\
 & | & \text{“}\texttt{v}\text{”}\ B\ \text{“}\texttt{z}\text{”} \\
 & | & \text{“}\texttt{w}\text{”}\ A\ \text{“}\texttt{z}\text{”} \\
A & \to & \text{“}\texttt{x}\text{”} \\
B & \to & \text{“}\texttt{x}\text{”}
\end{array}
$$

Abbildung 13.16: Eine kanonische LR-Grammatik, die keine LALR-Grammatik ist.

Wir berechnen zunächst die Menge der Zustände eines kanonischen LR-Parsers für diese Grammatik. Wir erhalten dabei die folgende Mengen von erweiterten markierten Regeln:

1. $s_0 = \mathit{closure}(\widehat{S} \to \bullet\ S : \$) = \{$
$$
\begin{array}{lcl}
\widehat{S} & \to & \bullet\ S : \$, \\
S & \to & \bullet \text{“}\texttt{v}\text{”} A \text{“}\texttt{y}\text{”} : \$, \\
S & \to & \bullet \text{“}\texttt{v}\text{”} B \text{“}\texttt{z}\text{”} : \$, \\
S & \to & \bullet \text{“}\texttt{w}\text{”} A \text{“}\texttt{z}\text{”} : \$, \\
S & \to & \bullet \text{“}\texttt{w}\text{”} B \text{“}\texttt{y}\text{”} : \$\ \},
\end{array}
$$
2. $s_1 = \mathit{goto}(s_0, S) = \{\widehat{S} \to S\bullet : \$\}$

3. $s_2 = goto(s_0,\ \text{“}\texttt{v}\text{”}\ ) = \{\ \begin{array}{lcl} S & \rightarrow & \text{“}\texttt{v}\text{”} \bullet B \text{“}\texttt{z}\text{”} : \$, \\ S & \rightarrow & \text{“}\texttt{v}\text{”} \bullet A \text{“}\texttt{y}\text{”} : \$, \\ A & \rightarrow & \bullet \text{“}\texttt{x}\text{”} : \text{“}\texttt{y}\text{”}, \\ B & \rightarrow & \bullet \text{“}\texttt{x}\text{”} : \text{“}\texttt{z}\text{”}\ \}, \end{array}$

4. $s_3 = goto(s_0,\ \text{“}\texttt{w}\text{”}\ ) = \{\ \begin{array}{lcl} S & \rightarrow & \text{“}\texttt{w}\text{”} \bullet A \text{“}\texttt{z}\text{”} : \$, \\ S & \rightarrow & \text{“}\texttt{w}\text{”} \bullet B \text{“}\texttt{y}\text{”} : \$, \\ A & \rightarrow & \bullet \text{“}\texttt{x}\text{”} : \text{“}\texttt{z}\text{”}, \\ B & \rightarrow & \bullet \text{“}\texttt{x}\text{”} : \text{“}\texttt{y}\text{”}\ \}, \end{array}$

5. $s_4 = goto(s_2,\ \text{“}\texttt{x}\text{”}\ ) = \{A \rightarrow \text{“}\texttt{x}\text{”}\bullet : \text{“}\texttt{y}\text{”},\ B \rightarrow \text{“}\texttt{x}\text{”}\bullet : \text{“}\texttt{z}\text{”}\}$,

6. $s_5 = goto(s_3,\ \text{“}\texttt{x}\text{”}\ ) = \{A \rightarrow \text{“}\texttt{x}\text{”}\bullet : \text{“}\texttt{z}\text{”},\ B \rightarrow \text{“}\texttt{x}\text{”}\bullet : \text{“}\texttt{y}\text{”}\}$,

7. $s_6 = goto(s_2, A) = \{S \rightarrow \text{“}\texttt{v}\text{”} A \bullet \text{“}\texttt{y}\text{”} : \$\}$,

8. $s_7 = goto(s_6,\ \text{“}\texttt{y}\text{”}\ ) = \{S \rightarrow \text{“}\texttt{v}\text{”} A \text{“}\texttt{y}\text{”}\bullet : \$\}$,

9. $s_8 = goto(s_2, B) = \{S \rightarrow \text{“}\texttt{v}\text{”} B \bullet \text{“}\texttt{z}\text{”} : \$\}$,

10. $s_9 = goto(s_8,\ \text{“}\texttt{z}\text{”}\ ) = \{S \rightarrow \text{“}\texttt{v}\text{”} B \text{“}\texttt{z}\text{”}\bullet : \$\}$,

11. $s_{10} = goto(s_3, A) = \{S \rightarrow \text{“}\texttt{w}\text{”} A \bullet \text{“}\texttt{z}\text{”} : \$\}$,

12. $s_{11} = goto(s_{10},\ \text{“}\texttt{z}\text{”}\ ) = \{S \rightarrow \text{“}\texttt{w}\text{”} A \text{“}\texttt{z}\text{”}\bullet : \$\}$,

13. $s_{12} = goto(s_3, B) = \{S \rightarrow \text{“}\texttt{w}\text{”} B \bullet \text{“}\texttt{y}\text{”} : \$\}$,

14. $s_{13} = goto(s_{12},\ \text{“}\texttt{y}\text{”}\ ) = \{S \rightarrow \text{“}\texttt{w}\text{”} B \text{“}\texttt{y}\text{”}\bullet : \$\}$.

Die einzigen Zustände, bei denen es Konflikte geben könnte, sind die Mengen s_4 und s_5, denn hier sind prinzipiell sowohl Reduktionen mit der Regel

$$A \rightarrow \text{“}\texttt{x}\text{”} \quad \text{als auch mit} \quad B \rightarrow \text{“}\texttt{x}\text{”}$$

möglich. Da allerdings die Mengen der Folge-Token einen leeren Durchschnitt haben, gibt es tatsächlich keinen Konflikt und die Grammatik ist eine kanonische LR-Grammatik.

Wir berechnen als nächstes die LALR-Zustände der oben angegebenen Grammatik. Die einzigen Zustände, die einen gemeinsamen Kern haben, sind die beiden Zustände s_4 und s_5, denn es gilt

$$core(s_4) = \{A \rightarrow \text{“}\texttt{x}\text{”}\bullet,\ B \rightarrow \text{“}\texttt{x}\text{”}\bullet\} = core(s_5).$$

Bei der Berechnung der LALR-Zustände werden diese beiden Zustände zu einem Zustand $s_{\{4,5\}}$ zusammen gefaßt. Dieser neue Zustand hat die Form

$$s_{\{4,5\}} = \big\{A \rightarrow \text{“}\texttt{x}\text{”}\bullet : \{\text{“}\texttt{y}\text{”}, \text{“}\texttt{z}\text{”}\},\ B \rightarrow \text{“}\texttt{x}\text{”}\bullet : \{\text{“}\texttt{y}\text{”}, \text{“}\texttt{z}\text{”}\}\big\}.$$

Hier gibt es offensichtlich einen Reduce-Reduce-Konflikt, denn einerseits haben wir

$$action(s_{\{4,5\}}, \text{“}\texttt{y}\text{”}) = \langle reduce, A \rightarrow \text{“}\texttt{x}\text{”}\rangle,$$

andererseits gilt aber auch

$$action(s_{\{4,5\}}, \text{“}\texttt{y}\text{”}) = \langle reduce, B \rightarrow \text{“}\texttt{x}\text{”}\rangle.$$

13.6.3 Bewertung der verschiedenen Methoden

Für die Praxis sind SLR-Parser nicht ausreichend, denn es gibt eine Reihe praktisch relevanter Sprach-Konstrukte, für die sich kein SLR-Parser erzeugen läßt. Kanonische LR-Parser sind wesentlich mächtiger, benötigen allerdings oft deutlich mehr Zustände. Hier stellen LALR-Parser einen Kompromiß dar: Einerseits sind LALR-Sprachen fast so ausdrucksstark wie kanonische LR-Sprachen, andererseits liegt der Speicherbedarf von LALR-Parsern in der gleichen Größen-Ordnung

wie der Speicherbedarf von SLR-Parsern. In den heute in der Regel zur Verfügung stehenden Hauptspeichern lassen sich allerdings auch kanonische LR-Parser mühelos unterbringen, so dass es eigentlich keinen zwingenden Grund mehr gibt, statt eines LR-Parsers einen LALR-Parser einzusetzen.

Andererseits wird niemand einen LALR-Parser oder einen kanonischen LR-Parser von Hand programmieren wollen. Statt dessen werden Sie später einen Parser-Generator wie *Bison* oder *JavaCup* einsetzen, der Ihnen einen Parser generiert. Aus historischen Gründen erzeugen diese Werkzeuge allerdings nur LALR-Parser, so dass Sie de facto keine Wahl haben.

Kapitel 14

Der Parser-Generator *JavaCup*

LALR-Parser erlauben es, auch links-rekursive Grammatiken in natürlicher Weise zu parsen. Da die meisten von Ihnen in der Parxis vermutlich mit *Java* arbeiten, möchte ich Ihnen in diesem Kapitel einen LALR-Parser-Generator vorstellen, den Sie benutzen können, wenn Sie in *Java* programmieren. In diesem Kapitel gebe ich Ihnen daher eine kurze Einführung in die Verwendung von CUP zusammen mit *JFlex*.

Der Parser-Generator CUP [HFA+99] ist ein LALR-Parser-Generator für *Java*. Wir werden die Version 0.11a verwenden. Sie finden diese Version im Netz unter

`http://www2.cs.tum.edu/projects/cup/java-cup-11a.jar`

Eine *Cup*-Spezifikation besteht aus fünf Teilen.

1. Der erste Teil ist optional und enthält gegebenenfalls eine Paket-Deklaration.
2. Der zweite Teil enthält die benötigten Import-Deklarationen.
3. Der dritte Teil deklariert die verwendeten Symbole. Hier werden also die Terminale und die syntaktischen Variablen spezifiziert.
4. Der vierte Teil ist wieder optional und spezifiziert die Präzedenzen von Operator-Symbolen.
5. Der fünfte Teil enthält die Grammatik-Regeln.

Abbildung 14.1 auf Seite 213 zeigt eine *Cup*-Spezifikation, mit deren Hilfe arithmetische Ausdrücke ausgewertet werden können. In dieser *Cup*-Spezifikation sind die Schlüsselwörter unterstrichen.

1. Die gezeigte CUP-Spezifikation enthält keine Paket-Deklarationen.
2. Die Spezifikation beginnt in Zeile 1 mit dem Import der Klassen von `java_cup.runtime`. Dieses Paket muss immer importiert werden, denn dort wird beispielsweise die Klasse `Symbol` definiert, die wir auch später noch in dem in Abbildung 14.4 gezeigten Scanner verwenden werden.

   Würden noch weitere Pakete benötigt, so könnten diese hier ebenfalls importiert werden.
3. In den Zeilen 3 bis 5 werden die Terminale deklariert. Es gibt zwei Arten von Terminalen:

   (a) Terminale, die keinen zusätzlichen Wert haben. Hierbei handelt es sich um die Operator-Symbole, die beiden Klammer-Symbole und das Semikolon. Die Syntax zur Deklaration solcher Terminale ist

   `terminal` $t_1, \cdots, t_n$;

   Durch diese Deklaration werden die Symbole $t_1, \cdots, t_n$ als Terminale deklariert.

```
1  import java_cup.runtime.*;
2
3  terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
4  terminal            LPAREN, RPAREN;
5  terminal Integer    NUMBER;
6
7  nonterminal         expr_list, expr_part;
8  nonterminal Integer expr, prod, fact;
9
10 expr_list ::= expr_list expr_part
11           |  expr_part
12           ;
13
14 expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
15           ;
16
17 expr ::= expr:e PLUS  prod:p {: RESULT = e + p; :}
18       |  expr:e MINUS prod:p {: RESULT = e - p; :}
19       |  prod:p              {: RESULT = p;     :}
20       ;
21
22 prod ::= prod:p TIMES  fact:f {: RESULT = p * f; :}
23       |  prod:p DIVIDE fact:f {: RESULT = p / f; :}
24       |  prod:p MOD    fact:f {: RESULT = p % f; :}
25       |  fact:f               {: RESULT = f;     :}
26       ;
27
28 fact ::= LPAREN expr:e RPAREN {: RESULT = e;   :}
29       |  MINUS  fact:e        {: RESULT = - e; :}
30       |  NUMBER:n             {: RESULT = n;   :}
31       ;
```

Abbildung 14.1: CUP-Spezifikation eines Parsers für arithmetische Ausdrücke

(b) Terminale mit einem zusätzlichen Wert. In diesem Fall muss zusätzlich der Typ dieses zusätzlichen Werts deklariert werden. Die Syntax ist in diesem Fall

`terminal` *type* $t_1, \cdots, t_n$;

Hierbei spezifiziert *type* den Typ, der den Terminalen $t_1, \cdots, t_n$ zugeordnet wird.

Bei der Spezifikation eines Typs ist es wichtig zu beachten, dass zwischen dem Typ und dem ersten Terminal kein Komma steht, denn sonst würde der Typ ebenfalls als Terminal interpretiert.

In Zeile 5 spezifizieren wir beispielsweise, dass das Terminal `NUMBER` einen Wert vom Typ `Integer` hat. Damit das funktioniert, muss der Scanner jedesmal, wenn er ein Terminal `NUMBER` an den Parser zurück geben soll, ein Objekt der Klasse `Symbol` erzeugen, dass die entsprechende Zahl als Wert beinhaltet. In dem auf Seite 217 in Abbildung 14.4 gezeigten Scanner geschieht dies beispielsweise in Zeile 31 dadurch, dass mit Hilfe der Methode `symbol`() der Konstruktor der Klasse `Symbol` aufgerufen wird, dem als zusätzliches Argument der Wert der Zahl übergeben wird.

4. In den Zeilen 7 und 8 werden die syntaktischen Variablen, die wir auch als Nicht-Terminale bezeichnen, deklariert. Die Syntax ist die selbe wie bei der Deklaration der Terminale, nur

dass wir jetzt das Schlüsselwort "`nonterminal`" an Stelle von "`terminal`" verwenden. Auch hier gibt es wieder zwei Fälle: Den in Zeile 7 deklarierten Nicht-Terminalen `expr_list` und `expr_part` wird kein Wert zugeordnet, während wir dem Nicht-Terminal *expr* einen Wert vom Typ `Integer` zuordnen, der sich aus der Auswertung des entsprechenden arithmetischen Ausdrucks ergibt.

5. Der letzte Teil einer CUP-Grammatik-Spezifikation enthält die Grammatik-Regeln. Die allgemeine Form einer CUP-Grammatik-Regel ist

   var "`::=`" $body_1$ "`{:`" $action_1$ "`:}`"
   "`|`" $body_2$ "`{:`" $action_2$ "`:}`"
   ⋮
   "`|`" $body_n$ "`{:`" $action_n$ "`:}`"
   "`;`"

   Dabei gilt

   (a) *var* ist die syntaktische Variable, die von dieser Regel erzeugt wird.

   (b) $body_i$ ist der Rumpf der i-ten Grammatik-Regel, der aus einer Liste von Terminalen und syntaktischen Variablen besteht.

   (c) $action_i$ ist eine durch Semikolons getrennte Folge von *Java*-Anweisungen, die ausgeführt werden, falls der Stack des Shift-Reduce-Parsers, der die Symbole enthält, mit der zugehörigen Regel reduziert wird.

   Bei der Spezifikation einer Grammatik-Regel mit CUP weicht die Syntax von der Syntax, die in ANTLR verwendet wird, an mehreren Stellen ab:

   (a) Statt eines einfachen Doppelpunkts "`:`" wird die Zeichenreihe "`::=`" verwendet, um die zu definierende Variable vom Rumpf der Grammatik-Regel zu trennen.

   (b) Die Kommandos werden bei CUP in den Zeichenreihen "`{:`" und "`:}`" eingeschlossen, während bei ANTLR die geschweiften Klammern "{" und "}" ausgereicht haben.

   (c) Um auf die Werte, die einem Terminal oder einer syntaktischen Variablen zugeordnet sind, zuzugreifen, hatten wir bei ANTLR Zuweisungen verwendet. Stattdessen müssen wir nun jedem Symbol, dessen Wert wir verwenden wollen, eine eigene Variable zuordnen, deren Namen wir getrennt von einem Doppelpunkt hinter das Symbol schreiben. Um der durch die Grammatik-Regel definierten syntaktischen Variablen einen Wert zuzuweisen, verwenden wir das Schlüsselwort "`RESULT`". Betrachten wir als Beispiel die Regel

   ```
   expr ::= expr:e PLUS prod:p {: RESULT = e + p; :};
   ```

   Mit `expr:e` sucht der Parser nach einem arithmetischen Ausdruck, dessen Wert in der Variablen `e` gespeichert wird. Anschließend wird ein Plus-Zeichen gelesen und darauf folgt wieder ein Produkt, dessen Wert jetzt in `p` gespeichert wird. Der Wert des insgesamt gelesenen Ausdrucks wird dann durch die Zuweisung

   ```
   RESULT = e + p;
   ```

   berechnet und der linken Seite der Grammatik-Regel zugewiesen.

   (d) Ein weiterer Unterschied zwischen CUP und ANTLR besteht darin, dass Nicht-Terminale nicht mehr durch den ihnen zugeordneten String repräsentiert werden können. Daher können wir den String "`PLUS`" nicht durch den String "`'+'`" ersetzen. An dieser Stelle sind CUP-Grammatiken leider nicht ganz so gut lesbar wie ANTLR-Grammatiken. Der Grund dafür ist, dass bei CUP zum Scannen ein separates Werkzeug, nämlich *JFlex*, zum Scannen verwendet wird. Demgegenüber wird bei ANTLR der Scanner zusammen

mit dem Parser in der selben Datei spezifiziert und das Werkzeug ANTLR erzeugt aus dieser Datei sowohl einen Parser als auch einen Scanner.

Um aus der in Abbildung 14.1 gezeigten CUP-Spezifikation einen Parser zu erzeugen, müssen wir diese zunächst mit dem Befehl

```
java java_cup.Main calc.cup
```

übersetzen. Damit dies funktioniert müssen Sie die Variable `CLASSPATH` so setzen, dass die Klasse `java_cup.Main` gefunden werden kann. Eine Möglichkeit, den Aufruf zu vereinfachen, besteht unter Unix darin, dass Sie sich eine Datei `cup` mit folgendem Inhalt irgendwo in Ihrem Pfad ablegen:

```
#!/bin/bash
CLASSPATH=/Users/stroetma/Software/JavaCUP-New/java-cup-11a.jar
java -cp $CLASSPATH java_cup.Main $@
```

In dieser Datei müssen Sie die Variable `CLASSPATH` natürlich so anpassen, dass die Datei

```
java-cup-11a.jar
```

gefunden werden kann. Danach können Sie *JavaCup* einfacher mit dem Befehl

cup calc.cup

aufrufen. Dieser Befehl erzeugt verschiedene *Java*-Dateien.

1. Die Datei `parser.java` enthält die Klasse `parser`, die den eigentlichen Parser enthält. Wenn Sie diese Klassse später übersetzen wollen, müssen Sie dafür sorgen, dass die Datei `java-cup-11a-runtime.jar` im `CLASSPATH` liegt.
2. Die Datei `sym.java` enthält die Klasse `sym`, welche die verschiedenen Symbole als statische Konstanten einer Klasse `sym` definiert. Diese Konstanten werden später im von *JFlex* erzeugten Scanner verwendet. Abbildung 14.2 auf Seite 215 zeigt diese Klasse.

```
1  public class sym {
2    public static final int MINUS  = 4;
3    public static final int DIVIDE = 6;
4    public static final int UMINUS = 8;
5    public static final int NUMBER = 11;
6    public static final int MOD    = 7;
7    public static final int SEMI   = 2;
8    public static final int EOF    = 0;
9    public static final int PLUS   = 3;
10   public static final int error  = 1;
11   public static final int RPAREN = 10;
12   public static final int TIMES  = 5;
13   public static final int LPAREN = 9;
14 }
```

Abbildung 14.2: Die Klasse `sym`.

Neben dem Parser wird noch ein Scanner benötigt. Diesen werden wir im nächsten Abschnitt präsentieren. Um mit dem Parser arbeiten zu können, brauchen wir eine Klasse, die eine Methode `main`() enthält. Abbildung 14.3 auf Seite 216 zeigt eine solche Klasse. Wir erzeugen dort in Zeile 4 einen Parser, indem wir den Konstruktor der Klasse `parser` mit einem Scanner als Argument initialsieren. Die per Default von *JFlex* erzeugte Scanner-Klasse hat den Namen `Yylex` und bekommt als Argument entweder ein Objekt vom Typ `java.io.Reader` oder ein Objekt vom Typ

`java.io.InputStream`. Der Parser wird dann durch Aufruf der Methode *parse*() gestartet, wobei eventuelle Ausnahmen noch abgefangen werden müssen. Falls mit dem Start-Symbol der Grammatik ein Wert assoziert ist, so wird dieser Wert von der Methode *parse*() als Ergebnis zurück gegeben, andernfalls wird der Wert `null` zurück gegeben.

```
1  public class Calculator {
2      public static void main(String[] args) {
3          try {
4              parser p = new parser(new Yylex(System.in));
5              p.parse();
6          } catch (Exception e) {}
7      }
8  }
```

Abbildung 14.3: Die Klasse `Calculator`.

14.0.4 Generierung eines Cup-Scanner mit Hilfe von *Flex*

Wir zeigen in diesem Abschnitt, wie wir mit Hilfe von *JFlex* einen Scanner für den im letzten Abschnitt erzeugten Parser erstellen können. Der Scanner, den wir benötigen, muss in der Lage sein, Zahlen und arithmetische Operatoren zu erkennen. Abbildung 14.4 auf Seite 217 zeigt einen solchen Scanner, den wir jetzt im Detail diskutieren.

1. In Zeile 1 importieren wir alle Klassen des Paketes `java_cup.runtime`. Dieses Paket enthält insbesondere die Definition der Klasse `Symbol`, mit der in einem Cup-Parser Terminale und Nicht-Terminale beschrieben werden. Daher muss dieses Paket bei jedem Scanner importiert werden, der an einen von Cup erzeugten Parser angeschlossen werden soll.

2. In den Zeilen 5 bis 7 spezifizieren wir, dass der Scanner die Anzahl der insgesamt gelesenen Zeichen, die Anzahl der gelesenen Zeilen und die Anzahl der in der aktuellen Zeile gelesenen Zeichen automatisch berechnen soll. Dadurch können wir später im Parser Syntax-Fehler präzise lokalisieren.

3. Zeile 8 spezifiziert mit dem Schlüsselwort "`%cup`", dass der Scanner an einen Cup-Parser angeschlossen werden soll.

4. In den Zeilen 11 bis 17 definieren wir zwei Hilfs-Methoden, die Objekte vom Typ `Symbol` erzeugen. Der Scanner muss Objekte von diesem Typ an den Parser zurück liefern. Die in dem Paket `java_cup.runtime` definierte Klasse `Symbol` stellt verschiedene Konstruktoren für diese Klasse zur Verfügung. Wir stellen die wichtigsten Konstruktoren vor.

   (a) `public Symbol(int symbolID);`
   Dieser Konstruktor bekommt als Argument eine natürliche Zahl, die festlegt, welche Art von Symbol definiert werden soll. Diese Zahl bezeichnen wir als *Symbol-Nummer*. Jedes Terminal und jede syntaktische Variable entspricht genau Symbol-Nummer. Die Kodierung der Symbol-Nummern wird von dem Parser-Generator Cup in der Klasse `sym` festgelegt. Abbildung 14.2 auf Seite 215 zeigt diese von Cup erzeugte Klasse.

   (b) `public Symbol(int symbolID, Object value);`
   Dieser Konstruktor bekommt zusätzlich zur Symbol-Nummer einen *Wert*, der im Symbol abgespeichert wird. Dieser Wert hat den Typ `Object`, wodurch der Typ so allgemein wie möglich ist. Dieser Konstruktor wird benutzt, wenn Terminale, die einen Wert haben, wie beispielsweise Zahlen, vom Scanner an den Parser zurück gegeben werden sollen.

```
1  import java_cup.runtime.*;
2
3  %%
4
5  %char
6  %line
7  %column
8  %cup
9
10 %{
11     private Symbol symbol(int type) {
12         return new Symbol(type, yychar, yychar + yylength());
13     }
14
15     private Symbol symbol(int type, Object value) {
16         return new Symbol(type, yychar, yychar + yylength(), value);
17     }
18 %}
19
20 %%
21
22 ";"           { return symbol( sym.SEMI   ); }
23 "+"           { return symbol( sym.PLUS   ); }
24 "-"           { return symbol( sym.MINUS  ); }
25 "*"           { return symbol( sym.TIMES  ); }
26 "/"           { return symbol( sym.DIVIDE ); }
27 "%"           { return symbol( sym.MOD    ); }
28 "("           { return symbol( sym.LPAREN ); }
29 ")"           { return symbol( sym.RPAREN ); }
30
31 [1-9][0-9]*|0 { return symbol(sym.NUMBER, new Integer(yytext())); }
32
33 [ \t\n]       { /* skip white space */ }
34
35 [^]           { throw new Error("Illegal character '" + yytext() +
36                                 "' at line " + yyline +
37                                 ", column "  + yycolumn); }
```

Abbildung 14.4: Ein Scanner für arithmetische Ausdrücke

(c) `public Symbol(int` *`symbolID`*`, int` *`start`*`, int` *`end`*`)`
Dieser Konstruktor hat zusätzlich zur Symbol-Nummer die Argumente `start` und `stop`, die den Anfang und das Ende des erkannten Terminals festlegen. Die Variable `start` gibt die Position des ersten Zeichens im Text an, während `end` die Position des letzten Zeichens des Tokens angibt. Diese Information ist nützlich, um später im Parser Syntax-Fehler besser lokalisieren zu können.

(d) `public Symbol(int` *`symbolID`*`, int` *`start`*`, int` *`end`*`, Objekt` *`value`*`)`
Dieser Konstruktor erhält zusätzlich zur Symbol-Nummer und Position des gelesenen Tokens noch den Wert, den dieses Token hat.

Bei der Implementierung der Hilfs-Methoden `symbol`() verwenden wir die Funktion `yylength`().

Diese Funktion gibt die Länge des Strings zurück, der dem zuletzt erkannten Tokens entspricht.

5. In den Zeilen 22 bis 29 erkennen wir die arithmetischen Operatoren und die Klammer-Symbole. Wir verwenden dabei die in der Klasse `sym` definierten Konstanten.

6. In Zeile 31 erkennen wir mit dem regulären Ausdruck "`[1-9][0-9]*|0`" eine natürliche Zahl. Diese verwandeln wir durch Aufruf des Konstruktors

   ```
   Integer(String s)
   ```

   in ein Objekt vom Typ `Integer`, wobei der Text, der der Zahl entspricht, von der Funktion `yytext()` geliefert wird. Anschließend geben wir ein Symbol zurück, in dem dieses Objekt als zugehöriger Wert abgespeichert wird.

7. In Zeile 33 überlesen wir Leerzeichen, Tabulatoren und Zeilen-Umbrüche. Das Überlesen geschieht dadurch, dass wir in diesem Fall kein Symbol an den Parser zurück geben, denn die semantische Aktion enthält keinen `return`-Befehl.

8. Falls ein beliebiges anderes Zeichen gelesen wird, geben wir mit der Regel, die in Zeile 35 beginnt, eine Fehlermeldung aus. Dabei greifen wir auf die Variablen `yyline` und `yycolumn` zurück, damit der Fehler lokalisiert werden kann.

JFlex erzeugt den Scanner in der Klasse `Yylex`. Wir hatten in Abbildung 14.3 gesehen, wie diese Klasse an den Parser angebunden wird.

14.1 Shift-Reduce und Reduce-Reduce-Konflikte

```
1  import java_cup.runtime.*;
2
3  terminal           PLUS, MINUS, TIMES, DIVIDE, MOD;
4  terminal           UMINUS, LPAREN, RPAREN;
5  terminal Integer   NUMBER;
6
7  nonterminal Integer expr;
8
9  expr ::= expr PLUS   expr
10       |  expr MINUS  expr
11       |  expr TIMES  expr
12       |  expr DIVIDE expr
13       |  expr MOD    expr
14       |  NUMBER
15       |  MINUS expr
16       |  LPAREN expr RPAREN
17       ;
```

Abbildung 14.5: Cup-Spezifikation eines Parsers für arithmetische Ausdrücke

Wir betrachten nun ein weiteres Beispiel. Abbildung 14.5 zeigt eine Grammatik, die offenbar mehrdeutig ist. Mit dieser Grammatik ist beispielsweise nicht klar, ob der String

`1 + 2 * 3` als "`(1 + 2) * 3`" oder als "`1 + (2 * 3)`"

gelesen werden soll. Wir hatten im letzten Kapitel schon gesehen, dass es in einer mehrdeutigen Grammatik immer Shift-Reduce- oder Reduce-Reduce-Konflikte geben muss. Wenn wir versuchen,

diese Grammatik mit CUP zu übersetzen und wenn wir dabei zusätzlich die Option "`-dump`" angeben, erhalten wir eine große Zahle von Shift-Reduce-Konflikten angezeigt. Beispielsweise erhalten wir die folgende Fehlermeldung:

```
Warning : *** Shift/Reduce conflict found in state #12
  between expr ::= expr TIMES expr (*)
  and     expr ::= expr (*) PLUS expr
  under symbol PLUS
  Resolved in favor of shifting.
```

Statt des Zeichens "•" benutzt CUP den String "`(*)`" zur Darstellung der Position in einer markierten Regel. Die obige Fehlermeldung zeigt uns an, dass es zwischen der markierten Regel

$$R_1 \;:=\; \Big(expr \rightarrow expr \text{ “}\texttt{*}\text{” } expr \bullet\Big)$$

und der markierten Regel

$$R_2 \;:=\; \Big(expr \rightarrow expr \bullet \text{ “}\texttt{+}\text{” } expr\Big)$$

einen Shift-Reduce-Konflikt gibt: Die beiden markierten Regeln R_1 und R_2 sind Elemente eines Zustands, der von CUP intern mit der Nummer 12 versehen worden ist. Die Menge aller Zustände kann über die Option "`-dump`" mit ausgegeben werden. Dazu wird CUP in der Form

```
cup -dump calc.cup
```

aufgerufen. Der Zustand mit der Nummer 12 wird dann wie folgt ausgegeben:

```
lalr_state [12]: {
  [expr ::= expr (*) MOD expr ,    {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) MINUS expr ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) DIVIDE expr , {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr TIMES expr (*) ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) TIMES expr ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) PLUS expr ,   {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
}
```

Damit können wir jetzt den Shift-Reduce-Konflikt interpretieren: Im Zustand 12 ist der Parser entweder dabei, die Eingabe mit der Regel

$$expr \rightarrow expr \text{ “}\texttt{*}\text{” } expr$$

zu reduzieren, oder der Parser ist gerade dabei, die rechte Seite der Regel

$$expr \rightarrow expr \text{ “}\texttt{+}\text{” } expr$$

zu erkennen, wobei er bereits eine *expr* erkannt hat und nun als nächstes das Token "`+`" erwartet wird. Da das Token "`+`" auch in der Follow-Menge der Regel R_1 liegen kann, ist an dieser Stelle unklar, ob das Token "`+`" auf den Stack geschoben werden soll, oder ob stattdessen mit der Regel R_1 reduziert werden muss. Bei einem Shift-Reduce-Konflikt entscheidet sich der von CUP erzeugte Parser immer dafür, das Token auf den Stack zu schieben.

14.2 Operator-Präzendenzen

Es ist mit CUP möglich, Shift-Reduce-Konflikte durch die Angabe von *Operator-Präzedenzen* aufzulösen. Abbildung 14.6 zeigt die Spezifikation einer Grammatik zur Erkennung arithmetischer Ausdrücke, die aus Zahlen und den binären Operatoren "`+`", "`-`", "`*`", "`/`" und "`^`" aufgebaut sind. Mit Hilfe der Schlüsselwörter "`%left`" und "`%right`" haben wir festgelegt, dass die Operatoren "`+`", "`-`", "`*`" und "`/`" *links-assoziativ* sind, ein Ausdruck der Form

$$3 - 2 - 1 \quad \text{wird also als} \quad (3 - 2) - 1 \quad \text{und nicht als} \quad 3 - (2 - 1)$$

gelesen. Demgegenüber ist der Operator "^", der in der CUP-Grammatik mit "POW" bezeichnet wir und die Potenzbildung bezeichnet, *rechts-assoziativ*, der Ausdruck

4^3^2 wird also als $4^{(3^2)}$ und nicht als $\left(4^3\right)^2$

interpretiert. Die Reihenfolge, in der die Assoziativität der Operatoren spezifiziert werden, legt die *Präzedenzen*, die auch als *Bindungsstärken* bezeichnet werden, fest. Dabei ist die Bindungsstärke umso größer, je später der Operator spezifiziert wird. In unserem konkreten Beispiel bindet der Exponentiations-Operator "^" also am stärksten, während die Operatoren "+" und "-" am schwächsten binden. Bei der in Abbildung 14.6 gezeigten Grammatik ordnet ANGABE den Operatoren die Bindungsstärke nach der folgenden Tabelle zu:

Operator	Bindungsstärke	Assoziativität
"+"	1	links
"-"	1	links
"*"	2	links
"/"	2	links
"^"	3	rechts

```
1  import java_cup.runtime.*;
2
3  terminal         SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, POW;
4  terminal         UMINUS, LPAREN, RPAREN;
5  terminal Double  NUMBER;
6
7  nonterminal        expr_list, expr_part;
8  nonterminal Double expr;
9
10 precedence left   PLUS, MINUS;
11 precedence left   TIMES, DIVIDE, MOD;
12 precedence right  UMINUS, POW;
13
14 expr_list ::= expr_list expr_part
15             |  expr_part
16             ;
17
18 expr_part ::= expr:e {: System.out.println("result = " + e); :} SEMI
19             ;
20
21 expr ::= expr:e1 PLUS   expr:e2 {: RESULT = e1 + e2; :}
22       |  expr:e1 MINUS  expr:e2 {: RESULT = e1 - e2; :}
23       |  expr:e1 TIMES  expr:e2 {: RESULT = e1 * e2; :}
24       |  expr:e1 DIVIDE expr:e2 {: RESULT = e1 / e2; :}
25       |  expr:e1 MOD    expr:e2 {: RESULT = e1 % e2; :}
26       |  expr:e1 POW    expr:e2 {: RESULT = Math.pow(e1, e2); :}
27       |  NUMBER:n                {: RESULT = n;       :}
28       |  MINUS expr:e           {: RESULT = - e;     :} %prec UMINUS
29       |  LPAREN expr:e RPAREN   {: RESULT = e;       :}
30       ;
```

Abbildung 14.6: Auflösung der Shift-Reduce-Konflikte durch Operator-Präzedenzen.

Wie erläutern nun, wie diese Bindungsstärken benutzt werden, um Shift-Reduce-Konflikte aufzulösen. CUP geht folgendermaßen vor:

1. Zunächst wird jeder Grammatik-Regel eine *Präzedenz* zugeordnet. Die Präzedenz ist dabei die Bindungsstärke des letzten in der Regel auftretenden Operators. Für den Fall, dass eine Regel mehrere Operatoren enthält, für die eine Bindungsstärke spezifiziert wurde, wird zur Festlegung der Bindungsstärke also der Operator herangezogen, der in der Regel am weitesten rechts steht. In unserem Beispiel haben die einzelnen Regeln damit die folgenden Präzedenzen:

Regel	Präzedenz
$E \to E$ "`+`" E	1
$E \to E$ "`-`" E	1
$E \to E$ "`*`" E	2
$E \to E$ "`/`" E	2
$E \to E$ "`^`" E	3
$E \to$ "`(`" E "`)`"	—
$E \to N$	—

   Für die Regeln, die keinen Operator enthalten, für den eine Bindungsstärke spezifiziert ist, bleibt die Präzedenz unspezifiziert.

2. Ist nun s ein Zustand, in dem zwei Regeln r_1 und r_2 der Form

   $$r_1 = (A \to \alpha \bullet o\ \beta : L_1) \quad \text{und} \quad r_2 = (B \to \gamma \bullet : L_2) \quad \text{mit} \quad o \in L_2$$

   vorkommen, so gibt es bei der Berechnung von

   $$\mathit{action}(s, o)$$

   zunächst einen Shift-Reduce-Konflikt. Ist nun o ein Operator, für den eine Präzedenz $p(o)$ festgelegt worden ist und hat außerdem die Regel r_2, mit der reduziert werden würde, die Präzedenz $p(r_2)$ so wird der Shift-Reduce-Konflikt in Abhängigkeit von der relativen Größe dieser beiden Zahlen aufgelöst. Hier werden drei Fälle unterschieden:

   (a) $p(o) > p(r_2)$: In diesem Fall bindet der Operator o stärker. Daher wird das Token o in diesem Fall auf den Stack geschoben:

   $\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle$.

   Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

   ```
   1+2*3
   ```

   mit den Grammatik-Regeln

   $$E \to E \text{ "`+`" } E \mid E \text{ "`*`" } E \mid \textsc{Number}$$

   parsen. Betrachten wir die Situation, bei der der Teilstring "`1+2`" bereits gelesen wurde und nun als nächstes das Token "`*`" verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

   $$\begin{array}{lcl} \{ \quad E & \to & E \bullet \text{"`*`"} E : \{\$, \text{"`*`"}, \text{"`+`"}\}, \\ \quad E & \to & E \bullet \text{"`+`"} E : \{\$, \text{"`*`"}, \text{"`+`"}\}, \\ \quad E & \to & E \text{"`+`"} E \bullet : \{\$, \text{"`*`"}, \text{"`+`"}\} \quad \}. \end{array}$$

   Wenn in diesem Zustand als nächstes Zeichen ein "`*`" gelesen wird, so darf der bisher gelesene String "`1+2`" nicht mit der Regel $E \to E$"`+`"E reduziert werden, denn wir wollen die 2 ja zunächst mit 3 multiplizieren. Statt dessen muss das Zeichen "`*`" auf den Stack geschoben werden.

(b) $p(o) < p(r_2)$: Jetzt bindet der Operator, der in der Regel r_2 auftritt, stärker als der Operator o. Daher wird in diesem Fall zunächst mit der Regel r_2 reduziert, wir haben also

$$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

```
1*2+3
```

mit den Grammatik-Regeln

$$E \to E \text{ “+” } E \mid E \text{ “*” } E \mid \textsc{Number}$$

parsen. Betrachten wir die Situation, bei der der Teilstring “1*2” bereits gelesen wurde und nun als nächstes das Token “+” verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

$$\left\{ \begin{array}{lcl} E & \to & E \bullet \text{“*”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \bullet \text{“+”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \text{“*”} E \bullet : \{\$, \text{“*”}, \text{“+”}\} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “+” gelesen wird, so soll der bisher gelesene String “1*2” mit der Regel $E \to E \text{“*”} E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

(c) $p(o) = p(r_2)$ und der Operator o ist links-assoziativ: Dann wird zunächst mit der Regel r_2 reduziert, wir haben also

$$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

```
1*2*3
```

mit den Grammatik-Regeln

$$E \to E \text{ “+” } E \mid E \text{ “*” } E \mid \textsc{Number}$$

parsen. Betrachten wir die Situation, bei der der Teilstring “1*2” bereits gelesen wurde und nun als nächstes das Token “*” verarbeitet werden soll. Der LALR-Parser ist dann in dem folgenden Zustand:

$$\left\{ \begin{array}{lcl} E & \to & E \bullet \text{“*”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \bullet \text{“+”} E : \{\$, \text{“*”}, \text{“+”}\}, \\ E & \to & E \text{“*”} E \bullet : \{\$, \text{“*”}, \text{“+”}\} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “*” gelesen wird, so soll der bisher gelesene String “1*2” mit der Regel $E \to E \text{“*”} E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

(d) $p(o) = p(r_2)$ und der Operator o ist rechts–assoziativ: In diesem Fall wird o auf den Stack geschoben:

$$\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle.$$

Wenn wir diesen Fall verstehen wollen, reicht es aus, den String

```
2^3^4
```

mit den Grammatik-Regeln

$$E \to E \text{“^”} E \mid \textsc{Number}$$

zu parsen und die Situation zu betrachten, bei der der Teilstring “1^2” bereits verarbeitet wurde und als nächstes Zeichen nun der Operator “^” gelesen wird.

(e) $p(o) = p(r_2)$ und der Operator o hat keine Assoziativität: In diesem Fall liegt ein Syntax-Fehler vor:

$action(s, o) = error$.

Diesen Fall verstehen Sie, wenn Sie versuchen, einen String der Form

```
1 < 1 < 1
```

mit den Grammatik-Regeln

$E \rightarrow E$ "`<`" $E \mid E$ "`+`" $E \mid$ NUMBER

zu parsen. In dem Moment, in dem Sie den Teilstring "`1 < 1`" gelesen haben und nun das nächste Token das Zeichen "`<`" ist, erkennen Sie, dass es ein Problem gibt.

Um in CUP einen Operator o als nicht-assoziativ zu deklarieren, schreiben Sie:

```
precedence nonassoc o
```

In den Fällen, in denen ein Shift-Reduce-Konflikt nicht mit den oben angegebenen Regeln aufgelöst werden kann, wird eine Warnung ausgegeben. In diesem Fall wird der Konflikt dann dadurch aufgelöst, dass das betreffende Token auf den Stack geschoben wird.

Die von CUP mit der Option "`-dump`" erzeugte Ausgabe zeigt im Detail, wie die Shift-Reduce-Konflikt Konflikte aufgelöst worden sind. Wir betrachten exemplarisch zwei Zustände in dieser Datei.

1. Der Zustand Nummer 14 hat die in Abbildung 14.7 gezeigte Form. Hier gibt es unter anderem einen Shift-Reduce-Konflikt zwischen den beiden markierten Regeln

   $E \rightarrow E \bullet$ "`+`" E und $E \rightarrow E$ "`*`" $E\bullet$,

   denn die erste Regel verlangt nach einem Shift, während die zweite Regel eine Reduktion fordert. Da die Regel $E \rightarrow E$ "`*`" E die selbe Präzedenz wie der Operator "`*`" und dieser eine höhere Präzedenz als "`+`" hat, wird beispielsweise beim Lesen des Zeichens "`+`" mit der Regel $E \rightarrow E$ "`*`" E reduziert. Wird hingegen das Zeichen "`^`" gelesen, so wird dieses geshiftet, denn dieses Zeichen hat eine höhere Priorität als die Regel $E \rightarrow E$ "`*`" E. Weiterhin

```
1  lalr_state [14]: {
2    [expr ::= expr (*) PLUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
3    [expr ::= expr TIMES expr (*) , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
4    [expr ::= expr (*) POW expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
5    [expr ::= expr (*) TIMES expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
6    [expr ::= expr (*) MOD expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
7    [expr ::= expr (*) MINUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
8    [expr ::= expr (*) DIVIDE expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
9  }
10
11 From state #14
12   [term 2:REDUCE(with prod 7)] [term 3:REDUCE(with prod 7)]
13   [term 4:REDUCE(with prod 7)] [term 5:REDUCE(with prod 7)]
14   [term 6:REDUCE(with prod 7)] [term 7:REDUCE(with prod 7)]
15   [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 7)]
```

Abbildung 14.7: Der Zustand Nummer 14.

   gibt es einen Shift-Reduce-Konflikt zwischen den beiden markierten Regeln

   $E \rightarrow E \bullet$ "`*`" E und $E \rightarrow E$ "`*`" $E\bullet$.

   Hier haben beide Regeln die gleiche Präzedenz. Daher entscheidet die Assoziativität. Da der Operator "`*`" links-assoziativ ist, wird mit der Regel $E \rightarrow E$ "`*`" E reduziert, falls das nächste Zeichen ein Multiplikations-Operator "`*`" ist.

2. Der Zustand Nummer 18 hat die in Abbildung 14.8 gezeigte Form. Zunächst gibt es hier einen Shift-Reduce-Konflikt zwischen den Regeln

   $$E \to E \bullet \text{ “}\hat{}\text{”} \; E \quad \text{und} \quad E \to E \text{ “}\hat{}\text{”} \; E\bullet,$$

   wenn das nächste Token der Operator “^” ist. Da der Operator die selbe Präzedenz hat wie die Regel, entscheidet die Assoziativität. Nun ist der Operator “^” rechts-assoziativ, daher wird in diesem Fall geshiftet.

```
1  lalr_state [18]: {
2    [expr ::= expr POW expr (*) , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
3    [expr ::= expr (*) PLUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
4    [expr ::= expr (*) POW expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
5    [expr ::= expr (*) TIMES expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
6    [expr ::= expr (*) MOD expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
7    [expr ::= expr (*) MINUS expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
8    [expr ::= expr (*) DIVIDE expr , {SEMI PLUS MINUS TIMES DIVIDE MOD POW RPAREN }]
9  }
10
11 From state #18
12  [term 2:REDUCE(with prod 10)] [term 3:REDUCE(with prod 10)]
13  [term 4:REDUCE(with prod 10)] [term 5:REDUCE(with prod 10)]
14  [term 6:REDUCE(with prod 10)] [term 7:REDUCE(with prod 10)]
15  [term 8:SHIFT(to state 9)] [term 11:REDUCE(with prod 10)]
```

Abbildung 14.8: Der Zustand Nummer 18.

Hier gibt es noch viele andere Shift-Reduce-Konflikte, die aber alle die selbe Struktur haben. Exemplarisch betrachten wir den Shift-Reduce-Konflikt zwischen den Regeln

$$E \to E \bullet \text{ “+”} \; E \quad \text{und} \quad E \to E \text{ “}\hat{}\text{”} \; E\bullet,$$

der auftritt, wenn das nächste Token ein “+” ist. Da die Regel $E \to E$ “^” E die Präzedenz 3 hat, die größer ist als die Präzedenz 1 des Operators “+” wird dieser Konflikt dadurch aufgelöst, dass mit der Regel $E \to E$ “^” E reduziert wird.

14.2.1 Das *Dangling-Else*-Problem

Bei der syntaktischen Beschreibung von Befehlen der Sprache `C` tritt bei der Behandlung von *if-then-else* Konstrukten ein Shift-Reduce-Konflikt auf, den wir jetzt analysieren wollen. Abbildung 14.9 zeigt eine Grammatik, die einen Teil der Syntax von Befehlen der Sprache `C` beschreibt. Um uns auf das wesentliche konzentrieren zu können, sind dort die Ausdrücke einfach nur Zahlen. Das Token “`ID`” steht für eine Variable, die Grammatik beschreibt also Befehle, die aus Zuweisungen, *If-Abfragen*, *If-Else-Abfragen* und *While-Schleifen* aufgebaut sind. Übersetzen wir diese Grammatik mit Cup, so erhalten wir den in Abbildung 14.10 ausschnittsweise gezeigten Shift-Reduce-Konflikt.

Der Konflikt entsteht bei der Berechnung von *action*(`state #10`, `else`) zwischen den beiden markierten Regeln

Statement → “`if`” “`(`” `EXPR` “`)`” *Statement* • und

Statement → “`if`” “`(`” `EXPR` “`)`” *Statement* • “`else`” *Statement*.

Die erste Regel verlangt nach einer Reduktion, die zweite Regel sagt, dass das Token `else` geshiftet werden soll. Das dem Konflikt zu Grunde liegende Problem ist, dass die in Abbildung 14.9 gezeigte Grammatik mehrdeutig ist, denn ein *Statement* der Form

```
if (a = b) if (c = d) s = t; else u = v;
```

```
terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
terminal NUMBER, ID;

nonterminal stmnt, stmntList, expr;

stmnt ::= IF LPAREN expr RPAREN stmnt
        |  IF LPAREN expr RPAREN stmnt ELSE stmnt
        |  WHILE LPAREN expr RPAREN stmnt
        |  LBRACE stmntList RBRACE
        |  ID ASSIGN expr SEMI
        ;

stmntList ::= stmntList stmnt
            |  /* epsilon */
            ;

expr ::= NUMBER
        ;
```

Abbildung 14.9: Fragment einer Grammatik für die Sprache `C`

```
Warning : *** Shift/Reduce conflict found in state #10
  between stmnt ::= IF LPAREN expr RPAREN stmnt (*)
  and     stmnt ::= IF LPAREN expr RPAREN stmnt (*) ELSE stmnt
  under symbol ELSE
  Resolved in favor of shifting.
```

Abbildung 14.10: Ein Shift-Reduce-Konflikt.

kann auf die folgenden beiden Arten gelesen werden:

1. Die erste (und nach der Spezifikation der Sprache `C` auch korrekte) Interpretation besteht darin, dass wir den Befehl wie folgt klammern:

   ```
   if (a = b) {
       if (c = d) {
           s = t;
       } else {
           u = v;
       }
   }
   ```

2. Die zweite Interpretation, die nach der in Abbildung 14.9 gezeigten Grammatik ebenfalls zulässig wäre, würde den Befehl in der folgenden Form interpretieren:

   ```
   if (a = b) {
       if (c = d) {
           s = t;
       }
   } else {
       u = v;
   ```

```
}
```

Diese Interpretation entspricht nicht der Spezifikation der Sprache `C`.

Es gibt drei Möglichkeiten, das Problem zu lösen.

1. Tritt ein Shift-Reduce-Konflikt auf, der nicht durch Operator-Präzedenzen gelöst wird, so ist der Default, dass das nächste Token auf den Stack geschoben wird. In dem konkreten Fall ist dies genau das, was wir wollen, weil dadurch das `else` immer mit dem letzten `if` assoziert wird. Um die normallerweise bei Konfliken von CUP ausgelöste Fehlermeldung zu unterdrücken, müssen wir CUP mit der Option "`-expect`" wie folgt aufrufen:

   ```
   java java_cup.Main -expect 1 -dump dangling.cup
   ```

   Die Zahl `1` gibt hier die Anzahl der Konflikte an, die wir erwarten. So lange die spezifizierte Zahl mit der tatsächlich gefundenen Zahl an Konflikten übereinstimmt, erzeugt CUP einen Parser.

2. Die zweite Möglichkeit besteht darin, die Grammatik so umzuschreiben, dass die Mehrdeutigkeit verschwindet. Die grundsätzliche Idee ist hier, zwischen zwei Arten von Befehlen zu unterscheiden.

   (a) Einerseits gibt es Befehle, bei denen jedem "`if`" auch ein "`else`" zugeordnet ist. Zwischen einem "`if`" und einem "`else`" dürfen nur solche Befehle auftreten.

   (b) Andererseits gibt es Befehle, bei denen dem letzten "`if`" kein "`else`" zugeordnet ist. solche Befehle dürfen nicht zwischen einem "`if`" und einem "`else`" auftreten.

   Abbildung 14.11 zeigt die Umsetzung dieser Idee. Die syntaktische Kategorie *MatchedStmnt* beschreibt dabei die Befehle, bei denen jedem "`if`" ein "`else`" zugeordnet ist, während die Kategorie *UnMatchedStmnt* die restlichen Befehle erfasst.

   Aus theoretischer Sicht ist das Umschreiben der Grammatik der sauberste Weg. Aus diesem Grund haben die Entwickler der Sprache *Java* in der ersten Version der Spezifikation dieser Sprache [GJS96] diesen Weg auch beschritten. Sie finden die *Java*-Grammatik im Netz unter

   ```
   http://java.sun.com/docs/books/jls/first_edition/html/19.doc.html
   ```

   Der Nachteil ist allerdings, dass bei diesem Vorgehen die Grammatik stark aufgebläht wird. Vermutlich aus diesem Grunde findet sich in der zweiten Auflage der Sprach-Spezifikation eine Grammatik, bei der das *Dangling-Else*-Problem wieder auftritt.

3. Die letzte Möglichkeit um das *Dangling-Else*-Problem zu lösen, besteht darin, dass wir "`if`" und "`else`" als Operatoren auffassen, denen wir eine Präzedenz zuordenen. Abbildung 14.12 zeigt die Umsetzung dieser Idee.

   (a) Zunächst haben wir in den Zeilen 6 und 7 die Terminale `IF` und `ELSE` als nichtassoziative Operatoren deklariert, wobei `ELSE` die höhere Präzedenz hat. Dadurch erreichen wir, dass ein `ELSE` auf den Stack geschoben wird, wenn der Parser in dem in Abbildung 14.10 gezeigten Zustand ist.

   (b) In Zeile 9 haben wir der Regel

   stmnt `::=` `IF` `LPAREN` *expr* `RPAREN` *stmnt*

   explizit mit Hilfe der nachgestellten Option

   ```
   %prec IF
   ```

   die Präzedenz des Operators `IF` zugewiesen. Dies ist notwendig, weil der letzte Operator, der in dieser Regel auftritt, die schließende runde Klammer `RPAREN` ist, der wir keine Priorität zugewiesen haben. Der Klammer eine Priorität zuzuweisen wäre einerseits kontraintuitiv, andererseits problematisch, da die Klammer ja auch noch an anderen

```
1  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
2  terminal NUMBER, ID;
3
4  nonterminal stmnt, matchedStmnt, unmatchedStmnt, stmntList, expr;
5
6  stmnt ::= matchedStmnt
7         |  unmatchedStmnt
8         ;
9
10 matchedStmnt ::= IF LPAREN expr RPAREN matchedStmnt ELSE matchedStmnt
11                |  WHILE LPAREN expr RPAREN matchedStmnt
12                |  LBRACE stmntList RBRACE
13                |  ID ASSIGN expr SEMI
14                ;
15
16 unmatchedStmnt ::= IF LPAREN expr RPAREN stmnt
17                  |  IF LPAREN expr RPAREN matchedStmnt ELSE unmatchedStmnt
18                  |  WHILE LPAREN expr RPAREN unmatchedStmnt
19                  ;
20
21 stmntList ::= stmntList stmnt
22              |  /* epsilon */
23              ;
24
25 expr ::= NUMBER
26       ;
```

Abbildung 14.11: Eine eindeutige Grammatik für C-Befehle.

Stellen verwendet werden kann. Mit Hilfe der `%prec`-Deklaration können wir einer Regel unmittelbar die Präzedenz eines Operators zuweisen und so das Problem umgehen.

In dem vorliegenden Fall ist die Präzedenz des Operators `ELSE` höher als die Präzedenz von `IF`, so dass der Shift-Reduce-Konflikt dadurch aufgelöst wird, dass das Token `ELSE` auf den Stack geschoben wird, wodurch eine `else`-Klausel tatsächlich mit der unmittelbar davor stehenden `if`-Klausel verbunden wird, wie es die Definition der Sprache `C` fordert.

Operator-Präzedenzen sind ein mächtiges Mittel um eine Grammatik zu strukturieren. Sie sollten allerdings mit Vorsicht eingesetzt werden, denn Sprachen wie die Programmier-Sprache `C`, bei der es 15 verschiedene Operator-Präzendenzen gibt, überfordern die meisten Benutzer.

```
1  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
2  terminal NUMBER, ID;
3
4  nonterminal stmnt, stmntList, expr;
5
6  precedence nonassoc IF;
7  precedence nonassoc ELSE;
8
9  stmnt ::= IF LPAREN expr RPAREN stmnt            %prec IF
10         |  IF LPAREN expr RPAREN stmnt ELSE stmnt
11         |  WHILE LPAREN expr RPAREN stmnt
12         |  LBRACE stmntList RBRACE
13         |  ID ASSIGN expr SEMI
14         ;
15
16 stmntList ::= stmntList stmnt
17             |  /* epsilon */
18             ;
19
20 expr ::= NUMBER
21        ;
```

Abbildung 14.12: Auflösung des Shift-Reduce-Konflikts mit Hilfe von Operator-Präzedenzen.

14.3 Auflösung von Reduce-Reduce-Konflikte

Im Gegensatz zu Shift-Reduce-Konflikten können Reduce-Reduce-Konflikte nicht durch Operator-Präzedenzen aufgelöst werden. Wir diskutieren in diesem Abschnitt die Möglichkeiten, die wir haben um Reduce-Reduce-Konflikte aufzulösen. Wir beginnen unsere Diskussion damit, dass wir die Reduce-Reduce-Konflikte in verschiedene Kategorien einteilen.

1. *Mehrdeutigkeits-Konflikte* sind Reduce-Reduce-Konflikte, die ihre Ursache in einer Mehrdeutigkeit der zu Grunde liegenden Grammatik haben. Solche Konflikte weisen damit auf ein tatsächliches Problem der Grammatik hin und können nur dadurch gelöst werden, dass die Grammatik umgeschrieben wird.

2. *Look-Ahead-Konflikte* sind Reduce-Reduce-Konflikte, bei denen die Grammatik zwar eindeutig ist, ein Look-Ahead von einem Token aber nicht ausreichend ist um den Konflikt zu lösen.

3. *Mysteröse Konflikte* entstehen erst beim Übergang von den LR-Zuständen zu den LALR-Zuständen durch das Zusammenfassen von Zuständen mit dem gleichen Kern. Diese Konflikte haben ihre Ursache also in der Unzulänglichkeit des LALR-Parser-Generators.

Wir betrachten die letzten beiden Fälle nun im Detail und zeigen Wege auf, wie die Konflikte gelöst werden können.

14.3.1 Look-Ahead-Konflikte

Ein Look-Ahead-Konflikt liegt dann vor, wenn die Grammatik zwar eindeutig ist, aber ein Look-Ahead von einem Token nicht ausreicht um zu entscheiden, mit welcher Regel reduziert werden soll. Abbildung 14.13 zeigt eine Grammatik[1], die zwar eindeutig ist, die aber keine LR-Grammatik ist.

```
1  A : B 'u' 'v'
2    | C 'u' 'w'
3    ;
4
5  B : 'x'
6    ;
7
8  C : 'x'
9    ;
```

Abbildung 14.13: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

Berechnen wir die LR-Zustände dieser Grammatik, so finden wir unter anderem den folgenden Zustand:

$$\{B \rightarrow \text{“}\texttt{x}\text{”} \bullet : \text{“}\texttt{u}\text{”},\ C \rightarrow \text{“}\texttt{x}\text{”} \bullet : \text{“}\texttt{u}\text{”}\}$$

Da die Menge der Folge-Token für beide Regeln gleich sind, haben wir hier einen Reduce-Reduce-Konflikt. Dieser Konflikt hat seine Ursache darin, dass der Parser mit einem Look-Ahead von nur einem Token nicht entscheiden kann, ob ein “x” als ein B oder als ein C zu interpretieren ist, denn dies entscheidet sich erst, wenn das auf “u” folgende Zeichen gelesen wird: Handelt es sich hierbei um ein “v” , so wird insgesamt die Regel

[1] Diese Grammatik habe ich im Netz auf der Seite von Pete Jinks unter der Adresse

`http://www.cs.man.ac.uk/~pjj/cs212/ho/node19.html`

gefunden.

$A \rightarrow B$ “u” “v”

verwendet werden und folglich ist das “x” als ein B zu interpretieren. Ist das zweite Token hinter dem “x” hingegen ein “w” , so ist die zu verwendende Regel

$A \rightarrow C$ “u” “w”

und folglich ist das “x” als C zu lesen.

Das Problem bei dieser Grammatik ist, dass sie versucht, abhängig vom Kontext ein “x” wahlweise als ein B oder als ein C zu interpretieren. Es ist offensichtlich, wie das Problem gelöst werden kann: Wir müssen die Regel für B und C zu einer Regel zusammenfassen. Das führt zu der in Abbildung 14.14 gezeigten Grammatik.

```
1  A : B 'u' 'v'
2    | B 'u' 'w'
3    ;
4
5  B : 'x'
6    ;
```

Abbildung 14.14: Eine LR(1)-Grammatik, die zu der der in Abbildung 14.13 gezeigten Grammatik äquivalent ist.

14.3.2 Mysteriöse Reduce-Reduce-Konflikte

Wir sprechen dann von einem *mysteriösen Reduce-Reduce-Konflikt*, wenn die gegebene Grammatik eine LR(1)-Grammatik ist, sich aber beim Übergang von LR-Zuständen zu LALR-Zuständen Reduce-Reduce-Konflikte ergeben. Die in Abbildung 14.15 gezeigte Grammatik habe ich dem *Bison*-Handbuch entnommen. (*Bison* ist ein LALR-Parser-Generator für die Sprachen `C` und `C++`.)

Übersetzen wir diese Grammatik mit CUP, so erhalten wir unter anderem den folgenden Zustand:

```
lalr_state [1]: {
  [name ::= ID (*) , {COMMA COLON }]
  [type ::= ID (*) , {ID COMMA }]
}
```

Da in beiden Mengen von Folgetoken das Token `COMMA` auftritt, gibt es hier offensichtlich einen Reduce-Reduce-Konflikt. Um diesen Konflikt besser zu verstehen, berechnen wir zunächst die Zustände eines kanonischen LR-Parsers für diese Grammatik. Wir erhalten dann eine Menge von Zuständen, von denen die für den späteren Konflikt ursächlichen Zuständen in Abbildung 14.16 gezeigt sind.

```
1   terminal    ID, COMMA, COLON;
2   nonterminal def, param_spec, return_spec, type, name_list, name;
3
4   def
5       ::= param_spec return_spec COMMA
6        ;
7   param_spec
8       ::= type
9        |  name_list COLON type
10       ;
11  return_spec
12      ::= type
13       |  name COLON type
14       ;
15  type
16      ::= ID
17       ;
18  name
19      ::= ID
20       ;
21  name_list
22      ::= name
23       |  name COMMA name_list
24       ;
```

Abbildung 14.15: Eine CUP-Grammatik mit einem mysteriösen Reduce-Reduce-Konflikt.

Analysieren wir die Zustände, so stellen wir fest, dass beim Übergang von LR-Zuständen zu den LALR-Zuständen die beiden Zustände s_7 und s_8 zu einem Zustand zusammengefasst werden, denn diese beiden Zustände haben den selben Kern. Bei der Zusammenfassung entsteht der Zustand, der von CUP als "`lalr_state [1]`" bezeichnet hat. Die Zustände s_7 und s_8 selber haben noch keinen Konflikt, weil dort die Mengen der Folgetoken disjunkt sind. Der Konflikt tritt erst durch die Vereinigung dieser beiden Mengen auf, denn dadurch ist das Token "`,`" als Folgetoken für beide in dem Zustand enthaltenen Regeln zulässig. Um den Konflikt aufzulösen müssen wir verhindern, dass die beiden Zustände s_7 und s_8 zusammengefasst werden. Dazu analysieren wir zunächst, wo diese Zustände herkommen.

1. Den Zustand s_7 erhalten wir, wenn wir im Zustand s_2 das Token `ID` lesen, denn es gilt

   $$s_7 = goto(s_2, \texttt{ID}).$$

2. Der Zustand s_8 entsteht, wenn das Token `ID` im Zustand s_0 gelesen wird, wir haben

   $$s_8 = goto(s_0, \texttt{ID}).$$

Die Idee zur Auflösung des Konflikts ist, dass wir den Zustand s_2 so ändern, dass die Kerne von $goto(s_2, \texttt{ID})$ und $goto(s_0, \texttt{ID})$ unterschiedlich werden. Die erweiterten markierten Regeln in dem Zustand s_2, die letztlich für den Konflikt verantwortlich sind, sind die Grammatik-Regeln für die syntaktische Variable *return_spec*. Wir ändern diese Regeln nun wie in Abbildung 14.17 ab, indem wir eine zusätzliche Grammatik-Regel

$$\textit{return\_spec} \rightarrow \texttt{ID BOGUS}$$

einführen. Wenn das Terminal `BOGUS` nie vom Scanner erzeugt werden kann, dann ändert sich durch die Hinzunahme dieser Regel die von der Grammatik erzeugte Sprache nicht. Allerdings

```
1   s0 = { S -> <*> def: [$],
2          def -> <*> param_spec return_spec ',': [$],
3          name -> <*> ID: [',', ':'],
4          name_list -> <*> name: [':'],
5          name_list -> <*> name ',' name_list: [':'],
6          param_spec -> <*> name_list ':' type: [ID],
7          param_spec -> <*> type: [ID],
8          type -> <*> ID: [ID]
9        }
10  s2 = { def -> param_spec <*> return_spec ',': [$],
11         name -> <*> ID: [':'],
12         return_spec -> <*> name ':' type: [','],
13         return_spec -> <*> type: [','],
14         type -> <*> ID: [',']
15       }
16  s7 = { name -> ID <*>: [',', ':'],
17         type -> ID <*>: [ID]
18       }
19  s8 = { name -> ID <*>: [':'],
20         type -> ID <*>: [',']
21       }
22
```

Abbildung 14.16: LR-Zustände der in Abbildung 14.15 gezeigten Grammatik.

ändern sich nun die LR-Zustände. Abbildung 14.18 zeigt, wie sich die entsprechenden Zustände ändern. Insbesondere sehen wir, dass der Zustand s_8 nun eine weitere markierte Regel enthält, zu der es in dem Zustand s_7 kein äquivalent gibt. Die Konsequenz ist, dass diese Zustände in einem LALR-Parser-Generator nicht mehr zusammengefasst werden. Dadurch gibt es dann auch keinen Konflikt mehr.

Aufgabe 21: Nachstehend sehen Sie Regeln für eine Grammatik, die Listen von Zahlen beschreibt:

$$\begin{array}{lcl} \mathit{List} & \rightarrow & \textsc{Number}\ \mathit{List} \\ & | & \textsc{Number} \end{array}$$

Diese Grammatik ist rechts-rekursiv. Alternativ ist auch eine links-rekursive Grammatik möglich:

$$\begin{array}{lcl} \mathit{List} & \rightarrow & \mathit{List}\ \textsc{Number} \\ & | & \textsc{Number} \end{array}$$

Überlegen Sie, welche dieser beiden Grammatiken effizienter ist, wenn Sie aus diesen Regeln mit einem LALR-Parser-Generator einen Parser erzeugen wollen. Berechnen Sie dazu für beide Fälle die Zustände und Aktionen und untersuchen Sie, wie eine Liste von natürlichen Zahlen der Form

$$x_1, x_2, \cdots, x_{n-1}, x_n,$$

in den beiden Fällen geparst wird. Geben Sie insbesondere an, wie der Stack des Shift-Reduce-Parsers aussieht, wenn die Teilliste

$$x_1, x_2, \cdots, x_{k-1}, x_k \quad \text{mit } k \leq n$$

gelesen worden ist.

```
def : param_spec return_spec ','
    ;
param_spec
    : type
    |  name_list ':' type
    ;
return_spec
    : type
    | name ':' type
    | ID BOGUS      // this never happens
    ;
type: ID
    ;
name: ID
    ;
name_list
    : name
    | name ',' name_list
    ;
```

Abbildung 14.17: Auflösung des mysteriösen Reduce-Reduce-Konflikts.

```
s0 = { S -> <*> def: [$],
       def -> <*> param_spec return_spec ',': [$],
       name -> <*> ID: [',', ':'],
       name_list -> <*> name: [':'],
       name_list -> <*> name ',' name_list: [':'],
       param_spec -> <*> name_list ':' type: [ID],
       param_spec -> <*> type: [ID],
       type -> <*> ID: [ID]
     }
s2 = { def -> param_spec <*> return_spec ',': [$],
       name -> <*> ID: [':'],
       return_spec -> <*> ID BOGUS: [','],
       return_spec -> <*> name ':' type: [','],
       return_spec -> <*> type: [','],
       type -> <*> ID: [',']
     }
s7 = { name -> ID <*>: [',', ':'],
       type -> ID <*>: [ID]
     }
s8 = { name -> ID <*>: [':'],
       return_spec -> ID <*> BOGUS: [','],
       type -> ID <*>: [',']
     }
```

Abbildung 14.18: Einige Zustände der in Abbildung 14.17 gezeigten Grammatik.

Kapitel 15

Typ-Überprüfung

Von den Skriptsprachen wie Tcl, *Perl*, *Python*, Ruby, etc. abgesehen verfügen die meisten heute verwendeten Programmier-Sprachen über ein *Typ-System*. Eine Sprache, die über ein *Typ-System* verfügt, wird als *getypte* Sprache bezeichnet. Bei einer getypten Sprache ist der Benutzer verpflichtet, die Typen aller verwendeten Variablen anzugeben. Der Compiler kann mit diesen Typ-Informationen dann sicherstellen, dass keine unzulässigen Operationen durchgeführt werden. Auf diese Weise können eine ganze Reihe von Fehlern schon vor der Ausführung des Programms vom Compiler entdeckt werden. Beispielsweise kann der Compiler erkennen, ob eine Methode mit Argumenten aufgerufen werden, deren Typ dem bei der Definition der Methode festgelegten Argument-Typen entspricht. Getypte Sprachen haben gegenüber den ungetypten Sprachen allerdings den Nachteil, dass die Programme etwas länger werden, denn es kommen die Typ-Deklarationen hinzu. Dieser Nachteil wird durch die verbesserte Wartbarkeit getypter Programme mehr als wett gemacht: Einfache Schreibfehler, aber auch einige konzeptuelle Fehler können von einem guten Typ-System automatisch aufgespürt werden.

```
1   class Hand {
2       Integer mValue;
3       String  mSuit;
4
5       Hand(Integer value, String suit) {
6           mValue = value;
7           mSuit  = suit;
8       }
9       ⋮
10  }
11
12  public class TypeError {
13      public static void main(String[] args) {
14          Hand h1 = new Hand( 14, "spades");
15          Hand h2 = new Hand("Ace", "spades");
16      }
17  }
```

Abbildung 15.1: Falscher Aufruf einer Methode

Abbildung 15.1 zeigt ein einfaches Beispiel für einen Typ-Fehler: In Zeile 15 wird der Konstruktor der lokalen Klasse `Hand` mit zwei Strings als Argumenten aufgerufen. Der tatsächlich in Zeile 5

definierte Konstruktor benötigt als erstes Argument aber keinen String, sondern eine ganze Zahl. Dieser Fehler wird vom JAVA-Compiler gefunden und wir bekommen die in Abbildung 15.2 gezeigte Fehlermeldung. Bei dem obigen winzigen Programm ist der Fehler offensichtlich, wären die Klasse `Hand` und `TypeError` aber von verschiedenen Personen in verschiedenen Dateien definiert worden, dann ist die vom Compiler durchgeführte Typ-Überprüfung ein sehr wichtiges Hilfsmittel um Fehler zu vermeiden. Insgesamt wird durch die Typ-Überprüfung die Wartbarkeit wesentlich erhöht, so dass bei großen Software-Projekten praktisch ausschließlich getypte Sprachen eingesetzt werden. Wir wollen uns daher in diesem Kapitel mit der Typ-Überprüfung beschäftigen.

```
1  TypeError.java:15: cannot find symbol
2  symbol  : constructor Hand(java.lang.String,java.lang.String)
3  location: class Hand
4          Hand h2 = new Hand("Ace", "s");
5                        ^
```

Abbildung 15.2: Fehlermeldung des *Java*-Compilers

Wir wollen in diesem Kapitel zeigen, wie sich die Typ-Korrektheit eines Porgramms automatisch überprüfen läßt. Dazu stellen wir imnächsten Abschnitt eine einfache Beispielsprache vor, für die wir die Theorie der Typ-Überprüfung vollständig entwickeln können. Später diskutieren wir die Probleme

15.1 Eine Beispielsprache

Wir stellen jetzt die Sprache TTL (*typed term language* vor. Dabei handelt es sich um eine sehr einfache Beispielsprache, die es dem Benutzer ermöglicht

- Typen zu definieren,
- Funktionen zu deklarieren und
- Terme anzugeben,

für die dann die Typ-Korrektheit nachgewiesen wird. Die Sprache TTL ist sehr einfach gehalten, damit wir uns auf die wesentlichen Ideen der Typ-Überprüfung konzentrieren können. Daher können wir in TTL auch keine wirklichen Programme schreiben, sondern einzig und allein überprüfen, ob Terme wohlgetypt sind.

```
1  type list(X) := nil + cons(X, list(X));
2
3  signature concat: list(T) * list(T) -> list(T);
4  signature x: int;
5  signature y: int;
6  signature z: int;
7
8  concat(nil, nil): list(int);
9  concat(cons(x, nil), cons(y, cons(z, nil))): list(int);
```

Abbildung 15.3: Ein TTL-Beispiel-Programm

Abbildung 15.3 zeigt ein einfaches Beispiel-Programm. Die Schlüsselwörter habe ich unterstrichen. Wir diskutieren dieses Programm jetzt im Detail.

1. In Zeile 1 definieren wir den *generischen* Typ `list`(X). Das X ist hier der Typ-Parameter, der später durch einen konkreten Typ wie z.B. `int`, `string` oder `list(string)` ersetzt werden kann.

   Semantisch ist die Zeile als induktive Definition zu lesen, durch die eine Menge von Termen definiert wird, wobei X eine gegebene Menge bezeichnet. *nil* und *cons* werden in diesem Zusammenhang als Funktions-Zeichen verwendet. Formal hat die induktive Definition die folgende Gestalt:

   (a) Induktions-Anfang: Der Term *nil* ist ein Element der Menge $\mathit{List}(X)$:

   $$\mathit{nil} \in \mathit{list}(X)$$

   (b) Induktions-Schritt: Falls a ein Element der Menge X und l ein Element der Menge $\mathit{list}(X)$ ist, dann ist der Term $\mathit{cons}(a, l)$ ebenfalls ein Element der Menge $\mathit{list}(X)$:

   $$a \in X \wedge l \in \mathit{list}(X) \rightarrow \mathit{cons}(a, l) \in \mathit{list}(X).$$

2. In Zeile 3 deklarieren wir die Funktion *concat* als zweistellige Funktion. Die beiden Argumente haben jeweils den Typ `list`(T) und das Ergebnis hat ebenfalls diesen Typ.

3. In den Zeilen 4 – 6 legen wir fest, dass die Variablen x, y und z jeweils den Typ `int` haben.

4. In Zeile 8 und 9 werden schließlich die beiden Terme

   $$\texttt{concat}(\texttt{nil}, \texttt{nil}) \quad \text{und} \quad \texttt{concat}(\texttt{cons}(\texttt{x}, \texttt{nil}), \texttt{cons}(\texttt{y}, \texttt{cons}(\texttt{z}, \texttt{nil})))$$

   angegeben und es wird behauptet, dass diese den Typ `list(int)` haben. Die Aufgabe der Typ-Überprüfung besteht darin, diese Aussage zu verifizieren.

Die genaue Syntax der Sprache Ttl wird durch die in Abbildung 15.4 gezeigte Grammatik definiert. Diese Grammatik verwendet neben den Zeichen-Reihen, die in doppelten Anführungs-Zeichen gesetzt sind, die folgenden Terminale:

1. Function bezeichnet entweder einen Typ-Konstruktor wie `list`, eine Variable wie `x` oder `y` oder einen Funktionsnamen wie `concat`. Syntaktisch werden diese dadurch erkannt, dass sie mit einem Kleinbuchstaben beginnen.

2. Parameter bezeichnet einen Typ-Parameter wie z.B. das X in `list`(X). Diese beginnen immer mit einem Großbuchstaben.

Bevor wir einen Algorithmus zur Typ-Überprüfung vorstellen können ist es erforderlich, einige grundlegende Begriffe wie den Begriff der Substitution und die Anwendung von Substitutionen auf Typen zu diskutieren.

15.2 Grundlegende Begriffe

Als erstes definieren wir den Begriff der Signatur eines Funktions-Zeichens. Die *Signatur* eines Funktionszeichens legt fest,

- welchen Typ die Argumente der Funktion haben und
- von welchem Typ das Ergebnis der Funktion ist.

Wir geben die Signatur einer Funktion f in der Form

$$f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho$$

an. Damit spezifizieren wir:

1. Die Funktion f erwartet n Argumente.

```
grammar ttl;

program   : typeDef* signature* typedTerm*
          ;

typeDef   : 'type' FCT ':=' type ('+' type)* ';'
          | 'type' FCT '(' PARAM (',' PARAM)* ')' ':=' type ('+' type)* ';'
          ;

type      : FCT '(' type (',' type)* ')'
          | FCT
          | PARAM
          ;

signature : 'signature' FCT ':' type ('*' type)* '->' type ';'
          | 'signature' FCT ':' type ';'
          ;

term      : FCT '(' term (',' term)* ')'
          | FCT
          ;

typedTerm : term ':' type ';'
          ;

PARAM   : ('A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
FCT     : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Abbildung 15.4: Eine EBNF-Grammatik für die getypte Beispielsprache

2. Das i-te Argument hat den Typ σ_i.

3. Das von der Funktion berechnete Ergebnis hat den Typ ϱ.

Als nächstes definieren wir, was wir unter einem Typ verstehen wollen. Anschaulich sind das Ausdrücke wie

$$\mathit{map}(K, V), \quad \mathit{double}, \quad \text{oder} \quad \mathit{list}(\mathit{int}).$$

Formal werden Typen aus Typ-Parametern und Typ-Konstruktoren aufgebaut. In dem obigen Beispiel sind *map*, *double*, *list* und *int* Typ-Konstruktoren, während K und V Typ-Parameter sind. Wir nehmen an, dass einerseits eine Menge $\mathbb{P}$ von Typ-Parametern und andererseits eine Menge von Typ-Konstruktoren $\mathbb{K}$ gegeben sind. In dem letzten Beispiel könnten wir

$$\mathbb{K} = \{\mathit{map}, \mathit{list}, \mathit{int}, \mathit{double}\} \quad \text{und} \quad \mathbb{P} = \{K, V\}$$

setzen. Zusätzlich muss noch eine Funktion

$$\mathit{arity} : \mathbb{K} \to \mathbb{N}$$

gegeben sein, die für jeden Typ-Konstruktor festlegt, wieviel Argumente er erwartet. In dem obigen Beispiel hätten wir

$$\mathit{arity}(\mathit{map}) = 2, \quad \mathit{arity}(\mathit{List}) = 1, \quad \mathit{arity}(\mathit{int}) = 0 \quad \text{und} \quad \mathit{arity}(\mathit{double}) = 0.$$

Dann wird die Menge $\mathcal{T}$ der Typen induktiv definiert:

1. Jeder Typ-Parameter X ist ein Typ:

$$X \in \mathbb{P} \Rightarrow X \in \mathcal{T}.$$

2. Ist c ein Typ-Konstruktor mit $\mathit{arity}(c) = 0$, so ist auch c ein Typ:

   $$c \in \mathbb{K} \wedge \mathit{arity}(c) = 0 \Rightarrow c \in \mathcal{T}.$$

3. Ist f ein n-stelliger Typ-Konstruktor und sind τ_1, $\cdots$, τ_n Typen, so ist auch $f(\tau_1, \cdots, \tau_n)$ ein Typ:

   $$f \in \mathbb{K} \wedge \mathit{arity}(f) = n \wedge n > 0 \wedge \tau_1 \in \mathcal{T} \wedge \cdots \wedge \tau_n \in \mathcal{T} \Rightarrow f(\tau_1, \cdots, \tau_n) \in \mathcal{T}.$$

Definition 42 (Parameter-Substitution) *Eine Parameter-Substitution ist eine endliche Menge von Paaren der Form*

$$\sigma = \big\{\langle X_1, \tau_1\rangle, \cdots, \langle X_n, \tau_n\rangle\big\}.$$

Dabei gilt:

1. *Die X_i sind Typ-Parameter.*
2. *Die τ_i sind Typen.*
3. *Für $i \neq j$ ist $X_i \neq X_j$, die Typ-Parameter sind also paarweise verschieden.*

Ist $\sigma = \big\{\langle X_1, \tau_1\rangle, \cdots, \langle X_n, \tau_n\rangle\big\}$ eine Parameter-Substitution, so schreiben wir

$$\sigma = \big[X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n\big].$$

Außerdem definieren wir den Domain *einer Parameter-Substitution als*

$$\mathit{dom}(\sigma) = \{X_1, \cdots, X_n\}.$$

Die Menge aller Parameter-Substitution bezeichnen wir mit SUBST. *Im folgenden werden wir der Kürze halber* Parameter-Substitutionen *einfach als* Substitutionen *bezeichnen.* □

Substitutionen werden für uns dadurch interessant, dass wir sie auf Typen *anwenden* können. Ist τ ein Typ und ϑ eine Substitution, so ist $\tau\vartheta$ der Typ, der aus τ dadurch entsteht, dass jedes Vorkommen eines Typ-Parameters X_i durch den zugehörigen Typ τ_i ersetzt wird. Die formale Definition folgt.

Definition 43 (Anwendung einer Substitution)
Es sei τ ein Typ und es sei $\vartheta = \big[X_1 \mapsto \tau_1, \cdots, X_n \mapsto \tau_n\big]$ eine Substitution. Wir definieren die Anwendung *von ϑ auf τ (Schreibweise $\tau\vartheta$) durch Induktion über den Aufbau des Typs τ:*

1. *Falls τ ein Typ-Parameter, gibt es zwei Fälle:*
   (a) *$\tau = X_i$ für ein $i \in \{1, \cdots, n\}$. Dann definieren wir*
   $$X_i\vartheta := \tau_i.$$
   (b) *$\tau = Y$, wobei y ein Typ-Parameter ist, so dass $Y \notin \{X_1, \cdots, X_n\}$ gilt. Dann definieren wir*
   $$Y\vartheta := Y.$$
2. *Andernfalls muß τ die Form $\tau = f(\sigma_1, \cdots, \sigma_m)$ haben. Dann können wir $\tau\vartheta$ durch*
   $$f(\sigma_1, \cdots, \sigma_m)\vartheta := f(\sigma_1\vartheta, \cdots, \sigma_m\vartheta).$$
   definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $\sigma_i\vartheta$ bereits definiert. □

Beispiele: Wir definieren eine Substitution ϑ durch

$$\vartheta := \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list(int)}\big].$$

Dann gilt:

1. $X_3\vartheta = X_3$,
2. $\texttt{list}(X_2)\vartheta = \texttt{list}\big(\texttt{list}(\texttt{int})\big)$,
3. $\texttt{map}(X_1, \texttt{set}(X_2))\vartheta = \texttt{map}\big(\texttt{double}, \texttt{set}(\texttt{list}(\texttt{int}))\big)$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 44 (Komposition von Substitutionen) *Es seien*

$$\vartheta = \big[X_1 \mapsto \sigma_1, \cdots, X_m \mapsto \sigma_m\big] \quad \textit{und} \quad \eta = \big[Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n\big]$$

zwei Substitutionen mit dom$(\vartheta) \cap$ dom$(\eta) = \{\}$. Dann definieren wir die Komposition *$\vartheta\eta$ von ϑ und η als*

$$\vartheta\eta := \big[X_1 \mapsto \sigma_1\eta, \cdots, X_m \mapsto \sigma_m\eta,\ Y_1 \mapsto \tau_1, \cdots, Y_n \mapsto \tau_n\big]. \qquad \square$$

Beispiel: Wir setzen

$$\vartheta := \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list}(\texttt{X}_3)\big] \quad \text{und}$$

$$\eta := \big[X_3 \mapsto \texttt{map}(\texttt{int}, \texttt{double}),\ X_4 \mapsto \texttt{char}\big].$$

Dann gilt:

$$\vartheta\eta = \big[X_1 \mapsto \texttt{double},\ X_2 \mapsto \texttt{list}(\texttt{map}(\texttt{int}, \texttt{double})),$$
$$X_3 \mapsto \texttt{map}(\texttt{int}, \texttt{double}),\ X_4 \mapsto \texttt{char}\big]. \qquad \square$$

Aufgabe 22: Überlegen Sie, wie die Definition der Komposition zweier Substitutionen ϑ und η lauten muss, wenn die Bedingung $dom(\vartheta) \cap dom(\eta) = \{\}$ nicht erfüllt ist.

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 45 (Assoziativ-Gesetz für Substitutionen)
Ist τ ein Typ und sind ϑ und η Substitutionen mit dom$(\vartheta) \cap$ dom$(\eta) = \{\}$, so gilt

$$(\tau\vartheta)\eta = \tau(\vartheta\eta). \qquad \square$$

Der Satz kann durch Induktion über den Aufbau des Typs τ bewiesen werden.

Als nächstes formalisieren wir die Sprechweise, dass ein Term t vom Typ τ ist.

Definition 46 ($t : \tau$) *Für einen Term t und einen Typ τ definieren wir die Relation $t : \tau$ (lese: t ist vom Typ τ) durch Induktion über t.*

1. *Ist c ein 0-stelliges Funktions-Zeichen mit der Signatur $c : \tau$ und ist ϑ eine beliebige Parameter-Substitution, so gilt*

   $$c : \tau\vartheta.$$

2. *Es gelte:*

   (a) *f sei ein n-stelliges Funktions-Zeichen mit $n > 0$.*

   (b) *f habe die Signatur*

   $$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho.$$

   (c) *ϑ sei eine Substitution so dass die Terme t_i den Typ $\sigma_i\vartheta$ haben.*

   Dann hat der Term $f(t_1, \cdots, t_n)$ den Typ $\varrho\vartheta$.

Beispiele: Die folgenden Beispiele beziehen sich auf die in Abbildung 15.3 definierten Typen und Funktionszeichen.

1. Es gilt $nil : \texttt{list(int)}$, denn dass Funktionszeichen *nil* hat die Signatur

   $$nil : \texttt{list(X)}.$$

   Setzen wir

   $$\vartheta = [X \mapsto \texttt{int}],$$

   so folgt $\texttt{list}(X)\vartheta = \texttt{list(int)}$ und das zeigt die Behauptung.

2. Als nächstes zeigen wir, dass

   $$concat(nil, nil) : \texttt{list(int)}$$

   gilt. Die Signatur der Funktion *concat* ist

   $$concat : list(T) \times list(T) \to list(T).$$

   Wieder definieren wir

   $$\vartheta = [T \mapsto \texttt{int}]$$

   Wir haben oben schon gesehen, dass

   $$nil : \texttt{list(int)}$$

   gilt. Wegen $\texttt{list(T)}\vartheta = \texttt{list(int)}$ folgt daraus die Behauptung.

15.3 Ein Algorithmus zur Typ-Überprüfung

Nehmen wir an, es ist ein Term $t = f(t_1, \cdots, t_n)$ und ein Typ τ gegeben und es soll geprüft werden, ob der Term t vom Type τ ist. Nehmen wir an, dass das Funktionszeichen f die Signatur

$$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$$

hat. Nach der obigen Definition von $t : \tau$ hat t genau dann den Typ τ, wenn es eine Parameter-Substitution ϑ gibt, so dass einerseits die Anwendung dieser Substitution auf ϱ den Typ τ liefert $(\varrho\vartheta = \tau)$ und andererseits die Teilterme t_i den Typ $\sigma_i\vartheta$ haben. Wir formalisieren dies wie folgt:

$$f(t_1, \cdots, t_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\varrho\vartheta = \tau \wedge \forall i \in \{1, \cdots, n\} : t_i : \sigma_i\vartheta\big)$$

Das Problem, das wir bei der Überprüfung von $t : \tau$ lösen müssen, besteht also darin, eine Parameter-Substitution ϑ zu finden, so dass einerseits

$$\varrho\vartheta = \tau$$

gilt und dass andererseits für alle Teilterme t_i die Beziehung

$$t_i : \sigma_i\vartheta$$

erfüllt ist.

Beispiele: Wir behandeln die bei der Definition des Begriffs $t : \tau$ diskutierten Beispiele jetzt noch einmal im Licht der Formel

$$f(t_1, \cdots, t_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\varrho\vartheta = \tau \wedge \forall i \in \{1, \cdots, n\} : t_i : \sigma_i\vartheta\big)$$

1. Als erstes betrachten wir den Term *nil* und weisen nach, dass $nil : \texttt{list(int)}$ gilt. Dazu fassen wir *nil* als 0-stelliges Funktionszeichens auf. Da `nil` einer der *Konstruktoren* des Typs $\texttt{list}(X)$ ist, hat *nil* die Signatur

   $$nil : \texttt{list(X)}.$$

Also hat die obige Typ-Bedingung die Form

$$\mathit{nil} : \texttt{list}(\texttt{int}) \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \big(\texttt{list}(X)\vartheta = \texttt{list}(\texttt{int})\big)$$

Zum Nachweis von $\mathit{nil} : \texttt{list}(\texttt{int})$ benötigen wir also eine Substitution ϑ für die

$$\texttt{list}(X)\vartheta = \texttt{list}(\texttt{int})$$

gilt. Offenbar leistet die Substitution

$$\vartheta = [X \mapsto \texttt{int}]$$

das Gewünschte und damit haben wir $\mathit{nil} : \texttt{list}(\texttt{int})$ nachgewiesen.

2. Als nächstes betrachten wir den Term $\mathit{concat}(\mathit{nil}, \mathit{nil})$ und weisen nach, dass dieser Term ebenfalls den Typ $\texttt{list}(\texttt{int})$ hat. Die Signatur der Funktion *concat* ist

   $$\mathit{concat} : \mathit{list}(T) \times \mathit{list}(T) \to \mathit{list}(T).$$

   Daher lautet die Typ-Bedingung diesmal

   $$\mathit{concat}(\mathit{nil}, \mathit{nil}) : \texttt{list}(\texttt{int}) \;\Leftrightarrow$$
   $$\exists \vartheta \in \textsc{Subst} : \big(\texttt{list}(T)\vartheta = \texttt{list}(\texttt{int}) \wedge \mathit{nil} : \mathit{list}(T)\vartheta \wedge \mathit{nil} : \mathit{list}(T)\vartheta\big).$$

   Die Gleichung $\texttt{list}(T)\vartheta = \texttt{list}(\texttt{int})$ wird von der Substitution

   $$\vartheta = [T \mapsto \texttt{int}]$$

   gelöst und daher ist die Frage der Korrektheit von

   $$\mathit{concat}(\mathit{nil}, \mathit{nil}) : \texttt{list}(\texttt{int})$$

   nun auf die Frage nach der Korrektheit von

   $$\mathit{nil} : \mathit{list}(T)\vartheta,$$

   also von $\mathit{nil} : \mathit{list(int)}$ reduziert. Letzteres haben wir aber unter Punkt 1. schon gezeigt. □

Die beiden Beispiele zeigen, dass die Frage der Korrektheit einer *Typ-Erklärung* der Form $t : \tau$ auf die Lösung einer Menge von *Typ-Gleichungen* der Form $\varrho\vartheta = \tau$ zurück geführt werden können. Wir formalisieren diese Beobachtung, indem wir eine Funktion

$$\mathit{typeEqs} : \texttt{Term} \times \texttt{Type} \to \texttt{set}(\texttt{Equation})$$

definieren. Die Idee ist, dass für einen Term t und einen Typ τ der Aufruf

$$\mathit{typeEqs}(t, \tau)$$

eine Menge von Typ-Gleichungen zurückliefert. Diese Gleichungen sind genau dann lösbar, wenn die Typ-Erklärung $t : \tau$ korrekt ist. Die Funktion *typeEqs* können wir durch die folgende bedingte Gleichung definieren:

$$\big(f : \sigma_1 \times \cdots \times \sigma_n \to \varrho\big) \implies \mathit{typeEqs}\big(f(t_1, \cdots, t_n), \tau\big) := \{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i).$$

Diese Gleichung ist wie folgt zu lesen: Falls die Funktion f die Signatur $f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$ hat, dann ist die Frage der Korrektheit der Typ-Erklärung $f(t_1, \cdots, t_n) : \tau$ äquivalent zu der Frage, ob das Typ-Gleichungs-System

$$\{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i)$$

durch eine geeignete *Substitution* ϑ gelöst werden kann. Diesen Begriff definieren wir formal wie folgt.

Definition 47 (Syntaktische Gleichung) *Unter einer* syntaktischen Gleichung *verstehen wir in diesem Abschnitt ein Konstrukt der Form $\sigma \doteq \tau$, wobei σ und τ Typen sind. Weiter definieren wir ein* syntaktisches Gleichungs-System *als eine Menge von syntaktischen Gleichungen. Eine Substitution ϑ* löst *eine syntaktische Gleichung $\sigma \doteq \tau$ genau dann, wenn $\sigma\vartheta = \tau\vartheta$ gilt.* □

Um die Typ-Überprüfung implementieren zu können, brauchen wir jetzt nur noch ein Verfahren, um Typ-Gleichungs-Systeme lösen zu können. Ein solches Verfahren haben Sie schon im ersten Semester kennengelernt, denn Typ-Gleichungs-Systeme lassen sich durch die selbe *Unifikation* lösen, die Sie schon von der Sprache *Prolog* kennen. Als konkreten Algorithmus zur Unifikation verwenden wir die von Martelli und Montanari [MM82] angegebenen Regeln. Allerdings können wir dieses Verfahren jetzt deshalb noch etwas vereinfachen, weil bei den Typ-Gleichungen der Form $\varrho = \tau$ nur auf der linken Seite, also in ϱ, Typ-Parameter auftauchen, die wir substituieren können. Bei dem Verfahren von Martelli und Montanari arbeiten wir mit Paaren der Form

$$\langle E, \vartheta \rangle.$$

Dabei ist E die Menge der zu lösenden Typ-Gleichungen und ϑ ist die Substitution, die wir berechnen wollen. Wir starten mit dem Paar

$$\langle E, [] \rangle,$$

bei dem die Substitution noch leer ist, während in E die Menge der zu lösenden Gleichungen zusammen gefasst sind. Wir formen diese Paare nun nach den folgenden Regeln um:

1. Falls X ein Typ-Parameter ist, so können wir die folgende Reduktion durchführen:

   $$\Big\langle E \cup \big\{X \doteq \tau\big\}, \vartheta \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[X \mapsto \tau], \vartheta\big[X \mapsto \tau\big] \Big\rangle$$

   Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $X \doteq \tau$, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution ϑ in die Substitution $\vartheta\big[X \mapsto \tau\big]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[X \mapsto \tau]$ angewendet.

2. Ist f ein n-stelliger Typ-Konstruktor, so gilt

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Big\langle E \cup \big\{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\big\}, \vartheta \Big\rangle.$$

   Eine syntaktische Gleichung der Form $f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)$ wird also ersetzt durch die n syntaktische Gleichungen $\sigma_1 \doteq \tau_1$, $\cdots$, $\sigma_n \doteq \tau_n$.

   Ein Spezialfall dieser Regel ist

   $$\Big\langle E \cup \big\{c \doteq c\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Big\langle E, \vartheta \Big\rangle.$$

   Hier steht c für einen 0-stelligen Typ-Konstruktor. Beispielsweise ist `int` ein 0-stelliger Typ-Konstruktor. Diese Gleichung besagt, dass triviale Gleichungen entfernt werden können.

3. Das Gleichungs-System $E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}$ hat keine Lösung, falls die Typ-Konstruktoren f und g verschieden sind, wir schreiben

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Omega.$$

   Hier steht Ω für einen Wert, der das Scheitern des Verfahrens dokumentiert. Falls wir bei der Lösung von $\mathit{typeEqs}(t, \tau)$ das Ergebnis Ω erhalten, dann trifft die Behauptung $t : \tau$ nicht zu und der Term t ist nicht vom Typ τ.

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$\texttt{map}(X_2, X_3) \doteq \texttt{map}(\texttt{char}, \texttt{list}(\texttt{int}))$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$
\begin{array}{rl}
 & \big\langle \big\{ \texttt{map}(X_2, X_3) \doteq \texttt{map}(\texttt{char}, \texttt{list}(\texttt{int})) \big\}, [] \big\rangle \\
\rightsquigarrow & \big\langle \big\{ X_2 \doteq \texttt{char},\ X_3 \doteq \texttt{list}(\texttt{int}) \big\}, [] \big\rangle \\
\rightsquigarrow & \big\langle \big\{ X_3 \doteq \texttt{list}(\texttt{int}) \big\}, \big[X_2 \mapsto \texttt{char} \big] \big\rangle \\
\rightsquigarrow & \big\langle \{\}, \big[X_2 \mapsto \texttt{char},\ X_3 \mapsto \texttt{list}(\texttt{int}) \big] \big\rangle
\end{array}
$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$\big[X_2 \mapsto \texttt{char},\ X_3 \mapsto \texttt{list}(\texttt{int}) \big]$$

als Lösung der oben gegebenen syntaktischen Gleichung.

15.4 Exkurs: Der Klassen-Generator EP

Die meisten Parser haben die Aufgabe, einen Parse-Baum zu erstellen. In diesem Parsebaum wird jeder Knoten durch ein Objekt einer Klasse dargestellt. Die Klassen dieser Objekte orientieren sich typischerweise an den Symbolen der Grammatik, so dass in der Regel für jedes Symbol auch eine Klasse existiert. Das führt dazu, dass wir bei der Entwicklung eines solchen Parses eine große Zahl von *Java*-Klassen definieren müssen. Das ist zwar nicht weiter schwer, aber doch mit einem hohen Schreibaufwand verbunden. Um hier die Arbeit abzukürzen, können wir einen *Klassen-Generator* einsetzen, die die benötigten Klassen aus einer Spezifikation erzeugt. Da diese Spezifikation meist wesentlich kürzer ist als die *Java*-Klassen, können wir dadurch einiges an Schreibarbeit einsparen. In diesem Abschnitt möchte ich Ihnen ein solches System vorstellen. Es handelt sich um das System EP, dass Sie unter der Adresse

```
http://www.ba-stuttgart.de/stroetmann/EP/ep.tar.gz
```

im Netz finden.

15.4.1 Installation

Wir beschreiben zunächst die Installation dieses Systems. Die nachfolgende Anleitung geht von einem Linux oder Unix-System aus. Wir nehmen an, dass der Klassen-Generator in dem Ordner `Software`, der sich im Home-Verzeichnis des Benutzers befindet, installiert werden soll. Die Installation besteht dann aus folgenden Schritten:

1. Zunächst schieben wir die Datei "`ep.tar.gz`" in das Verzeichnis `~/Software`:

   ```
   mv ep.tar.gz ~/Software
   ```

2. Nun wechseln wir in dieses Verzeichnis:

   ```
   cd ~/Software
   ```

3. Dort dekomprimieren wir die Datei `ep.tar.gz` mit dem Befehl

   ```
   gunzip ep.tar.gz
   ```

   Dabei entsteht die Datei `ep.tar`.

4. Die Datei `ep.tar` wird nun entpackt:

   ```
   untar xf ep.tar
   ```

   Durch diesen Befehl wird im Verzeichnis `Software` ein Unterverzeichnis mit dem Namen `EP` angelegt. Dieses Unterverzeichnis enthält drei Verzeichnisse:

(a) `src` enthält die *Java*-Quellen zusammen mit den Klassen-Dateien. Falls die Version Ihrer *Java*-Umgebung nicht mit der Version übereinstimmt, für die die Klassen-Dateien erzeugt wurden, müssen Sie in diesem Verzeichnis mit dem Befehl

```
javac *.java
```

alle *Java*-Quelldateien neu übersetzen.

(b) `doc` enthält die Dokumentation.

(c) `examples` enthält die Beispiel-Datei "`expr.ep`". In dieser Datei werden Klassen spezifiziert, die arithmetische Ausdrücke beschreiben.

Zusätzlich enthält das Verzeichnis `EP` noch die Datei `antlr-2.7.5.jar`. Hierbei handelt es sich um die Version des Parser-Generator ANTLR [PQ95], die ich bei der Erstellung des Parsers verwendet habe.

5. Um EP verwenden zu können, müssen Sie die Datei `antlr-2.7.5.jar` und das Verzeichnis `src` in den `CLASSPATH` aufnehmen. Das geht in einem *Unix*-basiertem Betriebssystem beispielsweise, indem Sie in der Datei `~/.bashrc` folgende Zeile hinzufügen:

   ```
   export CLASSPATH=$CLASSPATH:~/Software/EP/antlr-2.7.5.jar:~/Software/EP/src
   ```

6. Um die Installation zu testen, wechseln Sie in das Verzeichnis "`examples`" and starten dort EP, indem Sie das Kommando

   ```
   java EP expr
   ```

   ausführen. Dieses Kommando sollte eine Reihe von *Java*-Dateien erzeugen, die dann durch den Befehl

   ```
   javac *.java
   ```

   übersetzt werden können. Anschließend können wir das Kommando

   ```
   java Main
   ```

   ausführen. Dieses Kommando sollte das folgende Ergebnis produzieren:

   ```
   Product(Variable(x), Variable(x))/dx =
        Sum(Product(Number(1.0), Variable(x)),
            Product(Variable(x), Number(1.0))).
   ```

15.5 Eine Beispiel-Datei

Abbildung 15.5 zeigt eine EP-Spezifikation für Klassen, die arithmetische Ausdrücke spezifizieren. Die eigentliche Spezifikation der Klassen findet sich in den Zeilen 1 – 6. Hier wird eine abstrakte Klasse `Expr` definiert, von der insgesamt 6 Klassen abgeleitet werden. Als ein Beispiel betrachten wir die Klasse `Sum`. Der Ausdruck

```
Sum(Expr lhs, Expr rhs)
```

spezifiziert, dass die Klasse `Sum` zwei Member-Variablen enthält, die beide den Typ `Expr` haben. Weiterhin werden die Namen dieser Member-Variablen definiert. Der Name der ersten Member-Variable ist später `mLhs`, die zweite Member-Variable heißt `mRhs`. Dem in der Spezifikation angegebenen Namen wird also ein "`m`" vorangestellt und der darauf folgende Buchstabe wird groß geschrieben.

Neben der Spezifikation der Klassen enthält die in Abbildung 15.5 gezeigte Datei noch die Spezifikation der Signatur einer Methode: In Zeile 8 wird die Methode `diff` spezifiziert. Das implizite Argument dieser Methode ist vom Typ `Expr`, denn es handelt sich ja um eine Methode der Klasse `Expr`, das erste Argument ist vom Type `String` und das Ergebnis ist vom Type `Expr`. Die nachfolgenden bedingten Gleichungen zeigen, wie die Methode *diff*(), die das symbolische

```
1  Expr = Number(Double value)
2       + Variable(String name)
3       + Sum(Expr lhs, Expr rhs)
4       + Difference(Expr lhs, Expr rhs)
5       + Product(Expr lhs, Expr rhs)
6       + Quotient(Expr lhs, Expr rhs);
7
8  diff: Expr * String -> Expr;
9
10 Number(value).diff(x) = Number(0.0);
11
12 v.equals(x) -> Variable(v).diff(x) = Number(1.0);
13 Variable(v).diff(x) = Number(0.0);
14
15 Sum(l, r).diff(x) = Sum(l.diff(x), r.diff(x));
16 Difference(l, r).diff(x) = Difference(l.diff(x), r.diff(x));
17 Product(l, r).diff(x) =
18     Sum(Product(l.diff(x), r), Product(l, r.diff(x)));
19 Quotient(l, r).diff(x) =
20     Quotient( Difference(Product(l.diff(x), r), Product(l, r.diff(x))),
21               Product(r, r) );
```

Abbildung 15.5: Die Spezifikation arithmetischer Ausdrücke.

Differenzieren beschreibt, spezifiziert werden kann. Aus dieser Spezifikation erzeugt EP nun die in Zeile 1 – 6 spezifizierten Klassen. Wir betrachten exemplarisch die Klasse `Sum`, die in Abbildung 15.6 gezeigt ist. Wir diskutieren den erzeugten *Java*-Code jetzt im Detail.

1. In Zeile 1 sehen wir, dass die Klasse `Sum` von der Klasse `Expr` abgeleitet wird.
2. In Zeile 2 und 3 werden die Member-Variablen `mLhs` und `mRhs` definiert.
3. In Zeile 5 – 8 finden wir einen Konstruktor, der die Member-Variablen initialisiert.
4. In Zeile 9 – 11 finden wir die Implementierung der Methode *diff*. Diese Zeile setzt die Zeile 15 aus der Abbildung 15.5 um.
5. Des weiteren enthält die Klasse `Sum` noch eine Methode `equals`(), die automatisch erzeugt wird.
6. Außerdem werden noch Getter-Methoden für die Member-Variablen in den Zeilen 25 – 30 erzeugt.
7. Schließlich zeigt Zeile 31 – 33 eine automatisch erzeugte `toString`-Methode.

Um die von EP erzeugten Klassen zu testen, benötigen wir noch eine Methode `main`(). Abbildung 15.7 zeigt die Klasse `Main`, die eine solche Methode enthält. Übersetzen wir diese Klasse und führen wir das erzeugte Programm aus, so erhalten wir die folgende Ausgabe:

```
Sum(Product(Number(1.0), Variable(x)), Product(Variable(x), Number(1.0)))
```

Wir zeigen nun, wie wir alle drei Werkzeuge *JFlex*, *JavaCup* und Ep gemeinsam verwenden können um ein Programm zum symbolischen Differenzieren erstellen zu können.

1. Abbildung 15.8 zeigt die Spezifikation des Scanners. Gegenüber dem in Abbildung 14.4 gezeigten Scanner kommt hier die Zeile 30 hinzu, in der Variablen erkannt werden.

```
1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Sum(mLhs.diff(x), mRhs.diff(x));
11     }
12     public Boolean equals(Expr rhs) {
13         if (!(rhs instanceof Sum)) {
14             return false;
15         }
16         Sum r = (Sum) rhs;
17         if(!mLhs.equals(r.mLhs)) {
18             return false;
19         }
20         if(!mRhs.equals(r.mRhs)) {
21             return false;
22         }
23         return true;
24     }
25     public Expr getLhs() {
26         return mLhs;
27     }
28     public Expr getRhs() {
29         return mRhs;
30     }
31     public String toString() {
32         return "Sum(" + mLhs.toString() + ", " + mRhs.toString() + ")";
33     }
34 }
```

Abbildung 15.6: Die Klasse `Sum`.

```
1  public class Main {
2      public static void main(String[] args) {
3          Expr x      = new Variable("x");
4          Expr square = new Product(x, x);
5          System.out.println(square.diff("x"));
6      }
7  }
```

Abbildung 15.7: Eine Klasse zum Testen der von EP erzeugten Klassen.

2. Abbildung 15.9 zeigt die Spezifikation der Grammatik. Der wesentliche Unterschied zu der in Abbildung 14.1 gezeigten Grammatik besteht darin, dass mit der syntaktische Variable *expr* nun nicht mehr ein Objekt vom Typ `Integer` assoziiert ist, sondern statt dessen ein

```
1  import java_cup.runtime.*;
2
3  %%
4
5  %char
6  %line
7  %column
8  %cup
9
10 %{
11     private Symbol symbol(int type) {
12         return new Symbol(type, yychar, yychar + yylength());
13     }
14
15     private Symbol symbol(int type, Object value) {
16         return new Symbol(type, yychar, yychar + yylength(), value);
17     }
18 %}
19
20 %%
21
22 "+"          { return symbol( sym.PLUS   ); }
23 "-"          { return symbol( sym.MINUS  ); }
24 "*"          { return symbol( sym.TIMES  ); }
25 "/"          { return symbol( sym.DIVIDE ); }
26 "("          { return symbol( sym.LPAREN ); }
27 ")"          { return symbol( sym.RPAREN ); }
28
29 [1-9][0-9]*  { return symbol(sym.NUMBER,     new Integer( yytext())); }
30 [a-zA-Z]+    { return symbol(sym.IDENTIFIER, new Variable(yytext())); }
31 [ \t\n]      { /* skip white space */ }
```

Abbildung 15.8: *JFlex*-Spezifikation des Scanners

Objekt der Klasse `Expr`. Dieses Objekt repräsentiert einen Syntax-Baum.

3. Abbildung 15.5 aus dem letzten Abschnitt zeigt die Definition der abstrakten Klasse `Expr` und der davon abgeleiteten Klassen mit Hilfe einer EP-Spezifikation.

4. Abbildung 15.10 zeigt die Definition der Klasse Differentiator, in der die Methode *main*() enthalten ist.

   Hier muss die Zeile 7 näher erläutert werden: Die Methode *parse*() gibt ein Objekt vom Typ `Symbol` zurück. Die Klasse `Symbol` ist in dem Paket `java_cup.runtime` definiert und enthält eine als `public` deklarierte Member-Variable `value`. Diese Variable enthält den Wert, welcher dem von der Methode *parse*() erkannten Nichtterminal *expr* zugeordnet ist. Dieser Wert ist ein Element der Klasse `Expr`, der in der Klasse `Symbol` aber als `Object` deklariert ist. Daher müssen wir diesen Wert erst durch den Cast in Zeile 8 in den Typ `Expr` überführen, damit wird dann in Zeile 9 die Methode *diff*() Klasse `Expr` aufrufen dürfen.

```
1  import java_cup.runtime.*;
2
3  terminal           PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
4  terminal Integer   NUMBER;
5  terminal Variable  IDENTIFIER;
6
7  nonterminal Expr   expr;
8
9  precedence left    PLUS, MINUS;
10 precedence left    TIMES, DIVIDE;
11
12 /* The grammar */
13 expr    ::= expr:e1 PLUS   expr:e2 {: RESULT = new Sum(        e1, e2 ); :}
14           |  expr:e1 MINUS  expr:e2 {: RESULT = new Difference( e1, e2 ); :}
15           |  expr:e1 TIMES expr:e2  {: RESULT = new Product(    e1, e2 ); :}
16           |  expr:e1 DIVIDE expr:e2 {: RESULT = new Quotient(   e1, e2 ); :}
17           |  NUMBER:n               {: RESULT = new Number(     n      ); :}
18           |  IDENTIFIER:v           {: RESULT = v;                        :}
19           |  LPAREN expr:e RPAREN   {: RESULT = e;                        :}
20           ;
```

Abbildung 15.9: Grammatik für symbolische Ausdrücke

```
1  import java_cup.runtime.*;
2
3  public class Differentiator {
4      public static void main(String[] args) {
5          try {
6              parser p = new parser(new Yylex(System.in));
7              Symbol s = p.parse();
8              Expr   e = (Expr) s.value;
9              Expr   d = e.diff("x");
10             System.out.println("d/dx " + e + " = " + d);
11         } catch (Exception e) {}
12     }
13 }
```

Abbildung 15.10: Die Klasse `Differentiator`

15.6 Implementierung eines Typ-Checkers für TTL

Zur Illustration der dargestellten Theorie implementieren wir einen Typ-Checker für TTL. Die Grammatik hatten wir ja bereits in Abbildung 15.4 auf Seite 237 präsentiert. Die Abbildungen 15.11, 15.12 und 15.13 auf den folgenden Seiten zeigen die *JavaCup*-Spezifikation eines Parsers für diese Grammatik. Der zugehörige Scanner ist in Abbildungen 15.15 auf Seite 253 gezeigt. Der Scanner unterscheidet in den Zeilen 34 und 35 zwischen Namen, die mit einem großen Buchstaben beginnen und solchen Namen, die mit einem kleinen Buchstaben beginnen. Erstere bezeichnen Typ-Parameter, letztere bezeichnen sowohl Funktionen als auch Typ-Konstruktoren. Die *Java-*

Cup-Spezifikation des Scanners benutzt die *JavaCup*-Version `0.11a`[1], die auch generische Klassen unterstützt. Der von *JavaCup* generierte Parser baut einen abstrakten Syntax-Baum auf. Die zugehörigen Klassen wurden mit Hilfe des im vorhergehenden Abschnitts diskutierten Klassen-Generators EP aus der in Abbildung 15.14 gezeigten Spezifikation erzeugt. Wir werden dieses Werkzeug im nächsten Abschnitt näher diskutieren.

Die Klasse `MartelliMontanari` enthält die in Abbildung 15.16 gezeigte Methode *solve*(), mit der sich ein syntaktisches Gleichungs-System lösen läßt. Wir diskutieren diese Methode jetzt im Detail.

1. Am Anfang wählen wir in Zeilen 3 willkürlich die erste syntaktische Gleichung aus der Menge `mEquations` der zu lösenden syntaktischen Gleichungen aus.

   Das weitere Vorgehen richtet sich dann nach der Art der Gleichung.

2. In den Zeilen 6 – 14 behandeln wir den Fall, dass auf der linken Seite der syntaktischen Gleichung ein Typ-Parameter steht. In diesem Fall formen wir das syntaktische Gleichungs-System nach der folgenden Regel um:

   $$\Big\langle E \cup \big\{X \doteq \tau\big\}, \vartheta \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[X \mapsto \tau], \vartheta\big[X \mapsto \tau\big] \Big\rangle.$$

   Der Typ-Parameter, der in der obigen Formel mit `X` bezeichnet wird, trägt im Programm den Namen `var` und der Typ τ wird im Programm mit `rhs` bezeichnet.

3. In Zeile 15 betrachten wir den Fall, dass auf der linken Seite der syntaktischen Gleichung ein zusammengesetzter Typ steht. Wenn die Typ-Gleichung lösbar sein soll, muss dann auch auf der rechten Seite ein zusammengesetzter Typ stehen. Dies wird in Zeile 16 überprüft.

   Der restliche Code beschäftigt sich nun mit dem Fall, dass auf beiden Seiten der syntaktischen Gleichung ein zusammengesetzter Typ steht.

4. Die Zeilen 22 – 25 behandeln den Fall, dass die Typ-Konstruktoren auf beiden Seiten verschieden sind, es wird also der Fall

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_m) \doteq g(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Omega.$$

   behandelt. Die Methode liefert in diesem Fall statt einer Substitution den Wert `null` zurück.

5. Die Zeilen 27 – 32 behandeln schließlich den Fall, in dem wir das Gleichungs-System nach der Regel

   $$\Big\langle E \cup \big\{f(\sigma_1, \cdots, \sigma_n) \doteq f(\tau_1, \cdots, \tau_n)\big\}, \vartheta \Big\rangle \;\rightsquigarrow\; \Big\langle E \cup \big\{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\big\}, \vartheta \Big\rangle.$$

   umformen.

Als letztes diskutieren wir die Berechnung der Typ-Gleichungen, die in der Methode *typeEqs*() der Klasse `Term` implementiert ist. Ein Term $f(s_1, \cdots, s_n)$ hat genau dann den Typ τ, falls die Funktion f eine Signatur

$$f : \sigma_1 \times \cdots \times \sigma_n \to \varrho$$

hat und es darüber hinaus eine Parameter-Substitution ϑ gibt, so dass $\tau = \varrho\vartheta$ gilt und weiterhin die Terme s_i vom Typ $\sigma_i\vartheta$ sind:

$$f(s_1, \cdots, s_n) : \tau \;\Leftrightarrow\; \exists \vartheta \in \textsc{Subst} : \varrho\vartheta = \tau \wedge s_1 : \sigma_1\vartheta \wedge \cdots \wedge s_n : \sigma_n\vartheta.$$

1. Um die obige Definition von $t : \tau$ umzusetzen, suchen wir für einen Term der Form $f(s_1, \cdots, s_n)$ zunächst in Zeile 2 nach der Signatur des Funktions-Zeichens f. Diese Signaturen sind in

[1] Leider ist die Version nicht zu der Version 0.10 kompatibel. Die Inkompatibilität ist zwar für unsere Beispiele bedeutungslos, es gibt aber ein Problem mit den Programmen, die wir in der Rechnertechnik-Vorlesung eingesetzt haben, denn sowohl der IJVM-Assembler als auch der Mikro-Assembler benutzen die *JavaCup*-Version 0.10 und laufen nicht mehr, wenn sie statt dessen die Version 0.11 im CLASSPATH finden.

```
1  import java_cup.runtime.*;
2  import java.util.*;
3
4  terminal           TYPE, SIGNATURE, LEFT_PAR, RIGHT_PAR;
5  terminal           COMMA, COLON, SEMICOLON, ASSIGN, ARROW, PLUS, TIMES;
6  terminal String    FUNCTION, PARAMETER;
7
8  nonterminal Program         program;
9  nonterminal Term            term;
10 nonterminal List<Term>      termList;
11 nonterminal List<Parameter>  varList;
12 nonterminal Type            type;
13 nonterminal List<Type>      typeList;
14 nonterminal List<Type>      typeSum;
15 nonterminal TypeDef         typeDef;
16 nonterminal List<TypeDef>   typeDefList;
17 nonterminal Signature       signature;
18 nonterminal List<Signature> signatures;
19 nonterminal List<Type>      argTypes;
20 nonterminal TypedTerm       typedTerm;
21 nonterminal List<TypedTerm> typedTerms;
22
23 program     ::= typeDefList:typDefs signatures:signList typedTerms:termList
24                 {: RESULT = new Program(typDefs, signList, termList); :}
25                ;
26 typeDefList ::= typeDefList:l typeDef:t {: l.add(t); RESULT = l; :}
27                |  typeDef:t
28                   {: List<TypeDef> l = new LinkedList<TypeDef>();
29                      l.add(t); RESULT = l;
30                   :}
31                ;
32 typeDef     ::= TYPE FUNCTION:f ASSIGN typeSum:s SEMICOLON
33                 {: RESULT = new SimpleTypeDef(f, s); :}
34                |  TYPE FUNCTION:f LEFT_PAR varList:a RIGHT_PAR ASSIGN
35                       typeSum:s SEMICOLON
36                   {: RESULT = new ParamTypeDef(f, a, s); :}
37                ;
38 type        ::= FUNCTION:f LEFT_PAR typeList:t RIGHT_PAR
39                                {: RESULT = new CompositeType(f, t); :}
40                |  FUNCTION:f  {: RESULT = new CompositeType(f);    :}
41                |  PARAMETER:v {: RESULT = new Parameter(v);        :}
42                ;
43 typeList    ::= typeList:l COMMA type:t {: l.add(t); RESULT = l; :}
44                |  type:t {: List<Type> l = new LinkedList<Type>();
45                             l.add(t);
46                             RESULT = l;
47                          :}
48                ;
```

Abbildung 15.11: *JavaCup*-Spezifikation der TTL-Grammatik, 1. Teil

```
typeSum     ::= typeSum:l PLUS type:t {: l.add(t); RESULT = l; :}
              |  type:t
                 {:
                    List<Type> l = new LinkedList<Type>();
                    l.add(t);
                    RESULT = l;
                 :}
              ;
signature   ::= SIGNATURE FUNCTION:f COLON argTypes:a
                           ARROW type:t SEMICOLON
                 {: RESULT = new Signature(f, a, t); :}
              |  SIGNATURE FUNCTION:f COLON type:t SEMICOLON
                 {: List a = new LinkedList<Type>();
                    RESULT = new Signature(f, a, t);
                 :}
              ;
signatures  ::= signatures:l signature:s {: l.add(s); RESULT = l; :}
              |  signature:s
                 {: List<Signature> l = new LinkedList();
                    l.add(s);
                    RESULT = l;
                 :}
              ;
argTypes    ::= argTypes:l TIMES type:t {: l.add(t); RESULT = l; :}
              |  type:t
                 {: List<Type> l = new LinkedList();
                    l.add(t); RESULT = l;
                 :}
              ;
varList     ::= varList:l COMMA PARAMETER:v
                            {: l.add(new Parameter(v)); RESULT = l; :}
              |  PARAMETER:v {: List<Parameter> l = new LinkedList();
                               l.add(new Parameter(v));
                               RESULT = l;
                            :}
              ;
term        ::= FUNCTION:f LEFT_PAR termList:l RIGHT_PAR
                           {: RESULT = new Term(f, l); :}
              |  FUNCTION:f {: List<Term> l = new LinkedList<Term>();
                               RESULT = new Term(f, l);
                            :}
              ;
termList    ::= termList:l COMMA term:t
                      {: l.add(t); RESULT = l;            :}
              |  term:t {: List<Term> l = new LinkedList();
                           l.add(t); RESULT = l;
                        :}
              ;
```

Abbildung 15.12: *JavaCup*-Spezifikation der TTL-Grammatik, 2. Teil

```
typedTerm   ::= term:t COLON type:s SEMICOLON
                {: RESULT = new TypedTerm(t, s); :}
             ;

typedTerms  ::= typedTerms:l typedTerm:t
                {: l.add(t); RESULT = l; :}
             |  typedTerm:t
                {: List<TypedTerm> l = new LinkedList<TypedTerm>();
                   l.add(t);
                   RESULT = l;
                :}
             ;
```

Abbildung 15.13: *JavaCup*-Spezifikation der TTL-Grammatik, 3. Teil

```
Program = Program(List<TypeDef>   typeDefs,
                  List<Signature> signatures,
                  List<TypedTerm> typedTerms);

TypeDef = SimpleTypeDef(String name, List<Type> typeSum)
        + ParamTypeDef(String       name,
                       List<String> parameters,
                       List<Type>   typeSum);

Type = Parameter(String name)
     + CompositeType(String name, List<Type> argTypes);

substitute: Type * Parameter * Type -> Type;

Signature = Signature(String name, List<Type> argList, Type result);

Term = Term(String function, List<Term> termList);

typeEqs: Term * Type * Map<String, Signature> -> List<Equation>;

TypedTerm = TypedTerm(Term term, Type type);

Substitution = Substitution(List<Parameter> variables, List<Type> types);

Equation = Equation(Type lhs, Type rhs);

substitute: Equation * Parameter * Term -> Equation;

MartelliMontanari =
    MartelliMontanari(List<Equation> equations, Substitution theta);
```

Abbildung 15.14: Definition der benötigten *Java*-Klassen mit Hilfe von EP.

der Symboltabelle `map` hinterlegt. Findet sich für das Funktions-Zeichen keine Signatur, so liegt offenbar ein Fehler vor, der in Zeile 4 ausgegeben wird.

```
1   import java_cup.runtime.*;
2
3   %%
4
5   %char
6   %line
7   %column
8   %cup
9
10  %{
11      private Symbol symbol(int type) {
12          return new Symbol(type, yychar, yychar + yylength());
13      }
14
15      private Symbol symbol(int type, Object value) {
16          return new Symbol(type, yychar, yychar + yylength(), value);
17      }
18  %}
19
20  %%
21
22  "+"                { return symbol( sym.PLUS      ); }
23  "*"                { return symbol( sym.TIMES     ); }
24  "("                { return symbol( sym.LEFT_PAR  ); }
25  ")"                { return symbol( sym.RIGHT_PAR ); }
26  ","                { return symbol( sym.COMMA     ); }
27  ":"                { return symbol( sym.COLON     ); }
28  ";"                { return symbol( sym.SEMICOLON ); }
29  ":="               { return symbol( sym.ASSIGN    ); }
30  "->"               { return symbol( sym.ARROW     ); }
31  "type"             { return symbol( sym.TYPE      ); }
32  "signature"        { return symbol( sym.SIGNATURE ); }
33
34  [a-z][a-zA-Z_0-9]* { return symbol(sym.FUNCTION,  yytext()); }
35  [A-Z][a-zA-Z_0-0]* { return symbol(sym.PARAMETER, yytext()); }
36
37  [ \t\n]            { /* skip white space */ }
38  "//" [^\n]*        { /* skip comments    */ }
39
40  [^] { throw new Error("Illegal character '" + yytext() +
41                        "' at line " + yyline + ", column " + yycolumn); }
```

Abbildung 15.15: *JFlex*-Spezifikation des Scanners

2. Ansonsten speichern wir in Zeile 8 die Argument-Typen $\sigma_1, \cdots, \sigma_n$ in der Variablen `argTypes`. Die Signatur von f muss natürlich genau so viele Argument-Typen haben, wie das Funktions-Zeichen f Argumente hat. Dies wird in Zeile 9 überprüft.

3. Falls bis hierhin keine Probleme aufgetreten sind, erzeugen wir nun die Typ-Gleichungen nach der Formel

$$\big(f : \sigma_1 \times \cdots \times \sigma_n \to \varrho\big) \implies \mathit{typeEqs}\big(f(t_1, \cdots, t_n), \tau\big) := \{\varrho = \tau\} \cup \bigcup_{i=1}^{n} \mathit{typeEqs}(t_i, \sigma_i).$$

```
1  public Substitution solve() {
2      while (mEquations.size() != 0) {
3          Equation eq = mEquations.remove(0);
4          Type lhs = eq.getLhs();
5          Type rhs = eq.getRhs();
6          if (lhs instanceof Parameter) {
7              Parameter var = (Parameter) lhs;
8              List<Equation> newEquations = new LinkedList<Equation>();
9              for (Equation equation: mEquations) {
10                 Equation neq = equation.substitute(var, rhs);
11                 newEquations.add(neq);
12             }
13             mEquations = newEquations;
14             mTheta     = mTheta.substitute(var, rhs);
15         } else if (lhs instanceof CompositeType) {
16             CompositeType compLhs = (CompositeType) lhs;
17             CompositeType compRhs = (CompositeType) rhs;
18             if (!compLhs.getName().equals(compRhs.getName())) {
19                 // different type constructors, no solution
20                 System.err.println("Error: different type constructors\n");
21                 return null;
22             }
23             List<Type> lhsArgs = compLhs.getArgTypes();
24             List<Type> rhsArgs = compRhs.getArgTypes();
25             for (int i = 0; i < lhsArgs.size(); ++i) {
26                 Type sigmaLhs = lhsArgs.get(i);
27                 Type sigmaRhs = rhsArgs.get(i);
28                 mEquations.add(0, new Equation(sigmaLhs, sigmaRhs));
29             }
30         }
31     }
32     return mTheta;
33 }
```

Abbildung 15.16: Die Methode *solve*() aus der Klasse `MartelliMontanari`.

Dazu erstellen wir in Zeile 17 zuächst die Typ-Gleichungen

$$\varrho \doteq \tau$$

und berechnen dann in der Schleife in den Zeilen 19 – 22 rekursiv die Typ-Gleichungen, die sich aus den Forderungen

$$s_i : \sigma_i \quad \text{für } i = 1, \cdots, n$$

ergeben. Als Ergebnis erhalten wir eine Menge von Typ-Gleichungen, die genau dann lösbar sind, wenn der Term $f(s_1, \cdots, s_n)$ den Typ τ hat.

Als letztes diskutieren wir die Klasse `Program`. Abbildung 15.18 zeigt den Konstruktor dieser Klasse. Dieser Konstruktor bekommt als Argumente

- eine Liste von Typ-Definitionen `typeDefs`,
- eine Liste von Signaturen `signatures` und

```
1  public List<Equation> typeEqs(Type tau, Map<String, Signature> map) {
2      Signature sign = map.get(mFunction);
3      if (sign == null) {
4          System.err.println("The function " + mFunction +
5                             " has not been declared!");
6          throw new Error("Undeclared function in " + myString());
7      }
8      List<Type> argTypes = sign.getArgList();
9      if (argTypes.size() != mTermList.size()) {
10         System.err.println("Wrong number of parameters for function " +
11                            mFunction);
12         System.err.println("expected: " + argTypes.size());
13         System.err.println("found:    " + mTermList.size());
14         throw new Error("Wrong number of parameters in " + myString());
15     }
16     List<Equation> result = new LinkedList<Equation>();
17     Equation eq = new Equation(sign.getResult(), tau);
18     result.add(eq);
19     for (int i = 0; i < mTermList.size(); ++i) {
20         Term argI   = mTermList.get(i);
21         Type sigmaI = argTypes.get(i);
22         result.addAll( argI.typeEqs(sigmaI, map) );
23     }
24     return result;
25 }
```

Abbildung 15.17: Berechnung der Typ-Gleichungen

- eine Liste von getypten Termen `typedTerms`, deren Typ-Korrektheit überprüft werden soll.

Die wesentliche Aufgabe des Konstruktors besteht darin, die Member-Variable `mSignatureMap` zu initialisieren. Hierbei handelt es sich um eine Symboltabelle, in der zu jedem Funktions-Namen die zugehörige Signatur abgelegt ist. Dazu werden zunächst in der Schleife in Zeile 9 – 11 alle Signaturen aus der als Argument übergebenen Liste `signatures` in dieser Liste eingetragen. Dies alleine ist allerdings nicht ausreichend, denn durch die Typ-Definitionen werden implizit noch weitere Funktions-Zeichen deklariert. Beispielsweise werden durch die Typ-Definition

```
type  list(X) := nil + cons(X, list(X));
```

implizit die Funktionszeichen *nil* und *cons* definiert. Diese Funktionszeichen haben die folgenden Signaturen:

1. `nil: List(T)`,
2. `cons: T * List(T) -> List(T)`.

Die Aufgabe des Konstruktors der Klasse `Program` besteht also darin, aus den in dem Argument `typeDefs` enthaltenen Typ-Definitionen die Signaturen der implizit deklarierten Funktionszeichen zu generieren. Dazu iteriert die Schleife in Zeile 12 zunächst über alle Typ-Definitionen. Es gibt zwei Arten von Typ-Definitionen: Typ-Definitionen, bei denen der erzeugte Typ noch von einem oder mehreren Parametern abhängt, wie dies z.B. bei der Typ-Definition von `List(T)` der Fall ist, oder solche Typ-Definitionen, bei denen der erzeugte Typ keine Parameter hat. Letztere Typ-Definitionen werden durch die Klasse `SimpleTypeDef` dargestellt, erstere durch die Klasse `CompositeType`. Diese beiden Fälle werden durch die `if`-Abfrage in Zeile 13 unterschieden.

```
1  public Program(List<TypeDef>   typeDefs,
2                 List<Signature> signatures,
3                 List<TypedTerm> typedTerms)
4  {
5      mTypeDefs   = typeDefs;
6      mSignatures = signatures;
7      mTypedTerms = typedTerms;
8      mSignatureMap = new TreeMap<String, Signature>();
9      for (Signature s: mSignatures) {
10         mSignatureMap.put(s.getName(), s);
11     }
12     for (TypeDef td: mTypeDefs) {
13         if (td instanceof SimpleTypeDef) {
14             SimpleTypeDef std = (SimpleTypeDef) td;
15             Type rho = new CompositeType(std.getName(), new LinkedList<Type>());
16             for (Type tau: std.getTypeSum()) {
17                 CompositeType c = (CompositeType) tau;
18                 String name = c.getName();
19                 Signature s = new Signature(name, c.getArgTypes(), rho);
20                 mSignatureMap.put(name, s);
21             }
22         } else {
23             ParamTypeDef ctd = (ParamTypeDef) td;
24             List<Type> paramList = new LinkedList<Type>();
25             for (Parameter v : ctd.getParameters()) {
26                 paramList.add(v);
27             }
28             Type rho = new CompositeType(ctd.getName(), paramList);
29             for (Type tau: ctd.getTypeSum()) {
30                 CompositeType c = (CompositeType) tau;
31                 String name = c.getName();
32                 Signature s = new Signature(name, c.getArgTypes(), rho);
33                 mSignatureMap.put(name, s);
34             }
35         }
36     }
37 }
```

Abbildung 15.18: Der Konstruktor der Klasse `Program`.

1. Falls die untersuchte Typ-Definition `td` eine einfache Typ-Definition der Form

   $$\tau := f_1(\sigma_1^{(1)}, \cdots, \sigma_{n_1}^{(1)}) + \cdots f_k(\sigma_1^{(k)}, \cdots, \sigma_{n_k}^{(k)})$$

   ist, so wird der Typ τ von den Funktionen f_1, $\cdots$, f_k erzeugt. Die i-te Funktion f_i hat die Signatur

   $$f_i : \sigma_1^{(i)} \times \cdots \times \sigma_{n_i}^{(i)} \to \tau.$$

   Diese Signatur wird in Zeile 19 aus dem Ergebnis-Typ τ und den Argument-Typen `c.getArgTypes()` der Funktion f_i zusammengebaut. Die so konstruierte Signatur wird dann in der Symboltabelle `mSignatureMap` abgelegt.

2. Falls es sich bei der untersuchten Typ-Definition um die Definition eines parametrisierten Typs handelt, muss darauf geachtet werden, dass der Ergebnis-Typ der Signatur der Funktionen f_i auch von diesen Parametern abhängt. Zu diesem Zweck wird in Zeile 24 zunächst eine neue Parameter-Liste `paramList` angelegt, in die dann in der Schleife in Zeile 25 — 27 die Parameter des Typs hineinkopiert werden. Der Rest dieses Falls ist analog zum ersten Fall.

Neben dem Konstruktor enthält die Klasse `Program` noch die Methode `typeCheck()`, welche die Typüberprüfung ausführt. Die Implementierung dieser Methode ist in der Abbildung 15.19 gezeigt. Diese Methode iteriert in der Schleife, die in Zeile 2 beginnt, über alle getypten Terme $t : \tau$, die dem Konstruktor der Klasse `Program` übergeben wurden. Dazu wird in Zeile 6 zu jedem Term t und Typ τ die Menge der Typ-Gleichungen $typeEqs(t, \tau)$ berechnet. Dazu wird der Methode `typeEqs()`, die die Funktion *typeEqs*() implementiert, noch die vom Konstruktor berechnete Symboltabelle übergeben. In dieser Symboltabelle sind die Signaturen der verschiedenen Funktions-Zeichen abgespeichert. Die berechneten Typ-Gleichungen werden anschließend mit Hilfe der Methode `solve()` der Klasse `MartelliMontanari` gelöst. Falls die Typ-Gleichungen gelöst werden konnten, hat der Term t tatsächlich den Typ τ. Andernfalls wird eine Fehlermeldung ausgegeben.

```
1   public void typeCheck() {
2       for (TypedTerm tt: mTypedTerms) {
3           System.out.println("\nChecking " + tt.myString());
4           Term t   = tt.getTerm();
5           Type tau = tt.getType();
6           List<Equation> typeEquations = t.typeEqs(tau, mSignatureMap);
7           MartelliMontanari mm = new MartelliMontanari(typeEquations);
8           Substitution theta = mm.solve();
9           if (theta != null) {
10              System.out.println(tt.myString() + " has been verified!");
11              System.out.println(theta);
12          } else {
13              System.out.println(tt.myString() + " type ERROR!!!");
14          }
15      }
16  }
```

Abbildung 15.19: Die Methode *typeCheck()* in der Klasse `Program`.

15.7 Inklusions-Polymorphismus

Interpretieren wir Typen als Mengen von Werten, so stellen wir fest, dass es bei Typen eine Inklusions-Hierarchy gibt. In der Sprache *Java* hat jeder Wert vom Typ `String` auch gleichzeitig den Typ `Object`, es gilt also

$$\texttt{String} \subseteq \texttt{Object}.$$

Allgemein gilt: Ist e ein Ausdruck, der den Typ A hat und ist A weiter eine Klasse, die von der Klasse B abgeleitet ist, so kann der Ausdruck e überall dort verwendet werden, wo ein Ausdruck vom Typ B benötigt wird. Damit kann der Ausdruck e sowohl als ein A, als auch als ein B verwendet werden: e kann also *polymorph* verwendet werden. Im Gegensatz zu dem *parametrischen Polymorphismus*, den wir bisher betrachtet haben, sprechen wir hier von *Inklusions-Polymorphismus*. Die Behandlung von Inklusions-Polymorphismus ist besonders dann interessant, wenn dieser zusammen mit parametrischen Polymorphismus auftritt. Wir zeigen exemplarisch ein Beispiel in Abbildung 15.20.

1. In Zeile 4 wird hier zunächst ein Feld `x` von Strings angelegt.

2. In Zeile 5 definieren wir ein Feld `y` von Objekten und initialisieren dieses Feld mit dem bereits erzeugten Feld `x`. Da jeder String zugleich auch ein Objekt ist, ist dies möglich.

3. In Zeile 6 weisen wir dem zweiten Element des Feldes `y` die Zahl 2 zu. Da `y` ein Feld von Objekten ist und eine Zahl ebenfalls als Objekt angesehen werden kann, sollte auch dies kein Problem sein.

In der Tat läßt sich das Programm fehlerfrei übersetzen. Bei der Ausführung wird aber eine Ausnahme ausgelöst. Der Grund ist, dass mit der Zuweisung in Zeile 5 die Variable `y` eine Referenz auf das Feld von Strings ist, das in Zeile 4 angelegt worden ist. Wenn wir nun versuchen, in dieses Feld eine Zahl einzufügen, dann gibt es eine `ArrayStoreException`. Dieses Beispiel zeigt, dass die Sprache *Java* nicht statisch auf Typ-Sicherheit geprüft werden kann. Es zeigt auch, dass die Behandlung der Kombiation von Inklusions-Polymorphismus und parametrischen Polymorphismus deutlich komplexer ist, als die Behandlung von parametrischem Polymorphismus alleine.

```
1  public class TypeSurprise {
2
3      public static void main(String[] args) {
4          String[] x = { "a", "b", "c" };
5          Object[] y = x;
6          y[1] = new Integer(2);
7          System.out.println(x[1]);
8      }
9  }
```

Abbildung 15.20: Ein Typ-Problem in *Java*.

In dem Papier "*Type Inference for Java 5*" von Mazurak und Zdancewic wird das Typ-System der Sprache *Java* näher untersucht. Die Autporen kommen zu dem Schluss, dass die Frage, ob ein gegebenes *Java*-Programm korrekt getypt ist, unentscheidbar ist. Es ist daher auch nicht verwunderlich, dass sich der *Java*-Compiler bei manchen Programmen mit einem Stack-Overflow verabschiedet, der seine Ursache im Typ-Checker des Compilers hat. Desweiteren ist der folgende Satz aus der "*Java Language Specifiaction*", also der verbindlichen Definition der Sprache *Java*, aufschlussreich:

> *The type inference algorithm should be viewed as a heuristic, designed to perform well in practice. If it fails to infer the desired result, explicit type parameters may be used instead.*

Dies zeigt, dass die Kombination von parametrischem Polymorphismus und Inklusions-Polymorphismus zu sehr komplexen Problemen führen kann, die wir aber aus Zeitgründen nicht tiefer betrachten können.

Kapitel 16

Entwicklung eines einfachen Compilers

In diesem Kapitel konstruieren wir einen Compiler, der ein Fragment der Sprache `C` in den selben *Java*-Byte-Code übersetzt, den wir schon in der Rechnertechnik-Vorlesung kennengelernt haben. Das von dem Compiler übersetzte Fragment bezeichnen wir als *Integer*-`C`, denn es steht dort nur der Datentyp `int` zur Verfügung.

Ein Compiler besteht prinzipiell aus den folgenden Komponenten:

1. Der Scanner liest die zu übersetzende Datei ein und zerlegt diese in eine Folge von Token.

   Als Scanner werden wir das Werkzeug *JFlex* einsetzen.

2. Der Parser liest die Folge von Token und produziert als Ergebnis einen abstrakten Syntax-Baum.

   Wir werden zum Parsen *JavaCup* benutzen.

3. Der Typ-Checker überprüft den abstrakten Syntax-Baum auf Typ-Fehler.

   Da die von uns übersetzte Sprache nur einen einzelnen Datentyp enthält, werden wir bei unserem Compiler auf die Implementierung eines Typ-Checkers verzichten, denn das prinzipielle Vorgehen haben wir ja bereits im letzten Kapitel diskutiert und implementiert.

4. Der Code-Generator übersetzt den Parse-Baum in eine Folge von IJVM-Assembler-Befehlen.

5. In einer anschließend stattfindenden Optimierungsphase kann optional noch versucht werden, die Effizienz des erzeugten Code zu verbessern.

Bei unseren Compiler sind wir an dieser Stelle schon fertig. Bei Compilern, deren Zielcode ein RISC-Assembler-Programm ist, wird normalerweise zunächst auch ein Code erzeugt, der dem IJVM-Code ähnelt. Ein solcher Code wird als *Zwischen-Code* bezeichnet. Es bleibt dann die Aufgabe eines sogenannten *Backends*, daraus ein Assembler-Programm für einen gegebene Prozessor-Architektur zur erzeugen. Die schwierigste Aufgabe besteht hier darin, für die verwendeten Variablen eine Register-Zuordnung zu finden, bei der möglichst alle Variablen in Registern vorgehalten werden können. Dieses Thema ist allerdings so komplex, dass wir es in einem separaten Kapitel diskutieren.

16.1 Die Programmiersprache *Integer*-`C`

Wir stellen nun die Sprache *Integer*-`C` vor, die unser Compiler übersetzen soll. In diesem Zusammenhang sprechen wir auch von der *Quellsprache* unseres Compilers. Abbildung 16.1 zeigt die Grammatik der Quellsprache in erweiterter Backus-Naur-Form (EBNF). Die Grammatik für *Integer*-`C` verwendet die folgenden beiden Terminale:

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & (\textit{function})^{+} \\
\textit{function} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“}\texttt{(}\text{”}\ \textit{paramList}\ \text{“}\texttt{)}\text{”}\ \ \text{“}\{\text{”}\ \textit{decl}^{*}\ (\textit{stmnt}\ \text{“}\texttt{;}\text{”})^{*}\ \text{“}\}\text{”} \\
\textit{paramList} & \rightarrow & (\text{“}\texttt{int}\text{”}\ \textsc{Id}\ (\text{“}\texttt{,}\text{”}\ \ \text{“}\texttt{int}\text{”}\ \textsc{Id})^{*})? \\
\textit{decl} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“}\texttt{;}\text{”} \\
\textit{stmnt} & \rightarrow & \text{“}\{\text{”}\ \textit{stmntList}\ \text{“}\}\text{”} \\
 & | & \textsc{Id}\ \text{“}\texttt{=}\text{”}\ \textit{expr} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“}\texttt{(}\text{”}\ \textit{boolExpr}\ \text{“}\texttt{)}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“}\texttt{(}\text{”}\ \textit{boolExpr}\ \text{“}\texttt{)}\text{”}\ \textit{stmnt}\ \text{“}\texttt{else}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{while}\text{”}\ \ \text{“}\texttt{(}\text{”}\ \textit{boolExpr}\ \text{“}\texttt{)}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{return}\text{”}\ \textit{expr} \\
 & | & \textit{expr} \\
\textit{expr} & \rightarrow & \textit{expr}\ \text{“}\texttt{+}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{-}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{*}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{/}\text{”}\ \textit{expr} \\
 & | & \text{“}\texttt{(}\text{”}\ \textit{expr}\ \text{“}\texttt{)}\text{”} \\
 & | & \textsc{Number} \\
 & | & \textsc{Id} \\
 & | & \textsc{Id}\ \text{“}\texttt{(}\text{”}\ (\textit{expr}\ (\text{“}\texttt{,}\text{”}\ \textit{expr})^{*})?\ \text{“}\texttt{)}\text{”} \\
\textit{boolExpr} & \rightarrow & \textit{expr}\ \text{“}\texttt{==}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{!=}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{<=}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{>=}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{<}\text{”}\ \textit{expr} \\
 & | & \textit{expr}\ \text{“}\texttt{>}\text{”}\ \textit{expr} \\
 & | & \text{“}\texttt{!}\text{”}\ \textit{boolExpr} \\
 & | & \textit{boolExpr}\ \text{“}\texttt{\&\&}\text{”}\ \textit{boolExpr} \\
 & | & \textit{boolExpr}\ \text{“}\texttt{||}\text{”}\ \textit{boolExpr}
\end{array}
$$

Abbildung 16.1: Eine Grammatik für *Integer*-C

1. Id steht für eine Folge von Ziffern, Buchstaben und dem Unterstrich, die mit einem Buchstaben beginnt. Ein Id bezeichnet entweder eine Variable oder den Namen einer Funktion.

2. Number steht für eine Folge von Ziffern, die als Dezimalzahl interpretiert wird.

Nach der oben angegebenen Grammatik ist ein Programm eine Liste von Funktionen. Eine Funktion besteht aus der Deklaration der Signatur, worauf in geschweiften Klammern eine Liste von Deklarationen (*decl*) und Befehlen (*stmt*) folgt, die voneinander durch Semikolons getrennt werden. Der Aufbau der einzelnen Befehle ist dann ähnlich wie beider Sprache Sl, für die wir im Kapitel 10 einen Interpreter entwickelt haben.

Die in Abbildung 16.1 gezeigte Grammatik ist mehrdeutig:

1. Die Grammatik hat das *Dangling-Else-Problem*.

   Da wir im Kapitel 14 bereits gesehen haben, wie dieses Problem professionell gelöst werden kann, nehme ich mir hier die Freiheit, JavaCup mit der Option

   “`-expect 1`

   aufrufen und dadurch die Fehlermeldung zu unterdrücken, denn wir hatten ja bereits gesehen, das Cup per default den durch die Mehrdeutigkeit entstehenden Shift-Reduce-Konflikt in unserem Sinne auflöst.

2. Für die bei arithmetischen und Boole'schen Ausdrücken verwendeten Operatoren müssen Präzedenzen festgelegt werden.

```
1   int sum(int n) {
2       int s; s = 0;
3       while (n != 0) {
4           s = s + n; n = n - 1;
5       };
6       return s;
7   }
8   int main() {
9       int n;
10      n = getchar();
11      n = n - 48;
12      putchar(sum(n) + 48);
13      putchar(10);
14  }
```

Abbildung 16.2: Ein einfaches Integer-C-Programm.

Abbildung 16.2 zeigt ein einfaches Integer-C-Programm. Die Funktion $sum(n)$ berechnet die Summe $\sum_{i=1}^{n} i$ und die Funktion $main()$ liest ein Zeichen von der Tastatur, interpretiert dieses als Ascii-Darstellung einer Ziffer n, berechnet $sum(n)$ und gibt schließlich das Ergebnis als Ascii-Zeichen aus, was allerdings nur dann funktioniert, wenn das Ergebnis nur aus einer Ziffer besteht.

16.2 Entwicklung von Scanner und Parsers

Scanner und Parser werden mit Hilfe von *Jflex* und Java-Cup in der gleichen Art und Weise entwickelt, wie wir das bereits mehrfach in dieser Vorlesung gesehen haben. Abbildung 16.3 zeigt die Implementierung des Scanners.

Bevor wir die Implementierung des Parsers diskutieren können, müssen wir angeben, durch welche Klassen wir den Syntax-Baum darstellen wollen. Wir erzeugen diese Klassen mit Hilfe von Ep aus der in Abbildung 16.4 gezeigten Spezifikation. Die Definition dieser Klassen folgt 1 zu 1 der bereits angegebenen Grammatik.

Damit können wir nun die *Java*-Cup-Datei angeben, mit der wir den Syntaxbaum erzeugen. Wir haben diese Datei aus Platzgründen in drei Teile aufgespalten:

1. Abbildung 16.5 zeigt die Spezifikation der Terminale, Nicht-Terminale und Operator-Präzedenzen. Bei den Präzedenzen ist es sinnvoll diese so zu spezifizieren, dass die arithmetischen Operatoren stärker binden als die logischen Operatoren.

2. Abbildung 16.6 und 16.7 zeigen die eigentliche Grammatik und die Erzeugung des abstrakten Syntaxbaums. In der Grammatik lassen wir auch zu, dass die Liste der Funktionen leer ist. Das vereinfacht die Konstruktion der Liste etwas.

```
1  import java_cup.runtime.*;
2  %%
3  %char
4  %line
5  %column
6  %cup
7  %{
8      private Symbol symbol(int type) {
9          return new Symbol(type, yychar, yychar + yylength());
10     }
11
12     private Symbol symbol(int type, Object value) {
13         return new Symbol(type, yychar, yychar + yylength(), value);
14     }
15 %}
16 %%
17 "+"                   { return symbol( sym.PLUS      ); }
18 "-"                   { return symbol( sym.MINUS     ); }
19 "*"                   { return symbol( sym.TIMES     ); }
20 "/"                   { return symbol( sym.SLASH     ); }
21 "("                   { return symbol( sym.LPAREN    ); }
22 ")"                   { return symbol( sym.RPAREN    ); }
23 "{"                   { return symbol( sym.LBRACE    ); }
24 "}"                   { return symbol( sym.RBRACE    ); }
25 ","                   { return symbol( sym.COMMA     ); }
26 ";"                   { return symbol( sym.SEMICOLON ); }
27 "="                   { return symbol( sym.ASSIGN    ); }
28 "=="                  { return symbol( sym.EQUALS    ); }
29 "!="                  { return symbol( sym.NEQUALS   ); }
30 "<"                   { return symbol( sym.LT        ); }
31 ">"                   { return symbol( sym.LT        ); }
32 "<="                  { return symbol( sym.LE        ); }
33 ">="                  { return symbol( sym.GE        ); }
34 "&&"                  { return symbol( sym.AND       ); }
35 "||"                  { return symbol( sym.OR        ); }
36 "!"                   { return symbol( sym.NOT       ); }
37 "int"                 { return symbol( sym.INT       ); }
38 "return"              { return symbol( sym.RETURN    ); }
39 "if"                  { return symbol( sym.IF        ); }
40 "else"                { return symbol( sym.ELSE      ); }
41 "while"               { return symbol( sym.WHILE     ); }
42
43 [a-zA-Z][a-zA-Z_0-9]* { return symbol(sym.IDENTIFIER, yytext());          }
44 [0-9]|[1-9][0-9]*     { return symbol(sym.NUMBER, new Integer(yytext())); }
45 [ \t\n]               { /* skip white space */ }
46 "//" [^\n]*           { /* skip comments    */ }
47
48 [^] { throw new Error("Illegal character '" + yytext() +
49                       "' at line " + yyline + ", column " + yycolumn); }
```

Abbildung 16.3: Der Scanner für *Integer*-C

```
1  Program = Program(List<Function> functionList);
2
3  Function = Function(String           name,
4                      List<String>     parameterList,
5                      List<Declaration> mDeclarations,
6                      List<Statement>   body);
7
8  Statement = Block(List<Statement> statementList)
9            + Assign(String var, Expr expr)
10           + IfThen(BoolExpr boolExpr, Statement statement)
11           + IfThenElse(BoolExpr condition, Statement then, Statement else)
12           + While(BoolExpr condition, Statement statement)
13           + Return(Expr expr)
14           + ExprStatement(Expr expr);
15
16 Declaration = Declaration(String var);
17
18 Expr = Sum(Expr lhs, Expr rhs)
19      + Difference(Expr lhs, Expr rhs)
20      + Product(Expr lhs, Expr rhs)
21      + Quotient(Expr lhs, Expr rhs)
22      + MyNumber(Integer number)
23      + Variable(String name)
24      + FunctionCall(String name, List<Expr> args);
25
26 BoolExpr = Equation(Expr lhs, Expr rhs)
27          + Inequation(Expr lhs, Expr rhs)
28          + LessOrEqual(Expr lhs, Expr rhs)
29          + GreaterOrEqual(Expr lhs, Expr rhs)
30          + LessThan(Expr lhs, Expr rhs)
31          + GreaterThan(Expr lhs, Expr rhs)
32          + Negation(BoolExpr expr)
33          + Conjunction(BoolExpr lhs, BoolExpr rhs)
34          + Disjunction(BoolExpr lhs, BoolExpr rhs);
```

Abbildung 16.4: Spezifikation der Klassen zur Darstellung des Syntax-Baums.

```
1  // CUP specification for a simple expression evaluator (with actions)
2  import java_cup.runtime.*;
3  import java.util.*;
4
5  /* Terminals (tokens returned by the scanner). */
6  terminal COMMA, PLUS, MINUS, TIMES, SLASH, LPAREN, RPAREN, LBRACE, RBRACE;
7  terminal ASSIGN, EQUALS, LT, GT, LE, GE, NEQUALS, AND, OR, NOT;
8  terminal IF, ELSE, WHILE, RETURN, SEMICOLON;
9  terminal INT;
10 terminal String    IDENTIFIER;
11 terminal Integer   NUMBER;
12
13 /* Non-terminals */
14 nonterminal Program           program;
15 nonterminal List<Function>    functionList;
16 nonterminal Function          function;
17 nonterminal List<String>      paramList, neParamList;
18 nonterminal Declaration       declaration;
19 nonterminal List<Declaration> declarations;
20 nonterminal Statement         statement;
21 nonterminal List<Statement>   statementList;
22 nonterminal Expr              expr;
23 nonterminal List<Expr>        exprList, neExprList;
24 nonterminal BoolExpr          boolExpr;
25
26 precedence left     OR;
27 precedence left     AND;
28 precedence right    NOT;
29 precedence left     PLUS, MINUS;
30 precedence left     TIMES, SLASH;
```

Abbildung 16.5: Deklaration der Terminale, Nicht-Terminale und Operator-Präzedenzen

```
31  program ::= functionList:l {: RESULT = new Program(l); :} ;
32
33  functionList ::= /* epsilon */ {: RESULT = new LinkedList<Function>(); :}
34                 |  functionList:l function:f {: l.add(f); RESULT = l;    :}
35                 ;
36
37  function ::= INT IDENTIFIER:f LPAREN paramList:p RPAREN LBRACE
38               declarations:d statementList:l RBRACE
39               {: RESULT = new Function(f, p, d, l); :}
40             ;
41
42  paramList ::= /* epsilon */ {: RESULT = new LinkedList<String>(); :}
43              |  neParamList:l {: RESULT = l;                        :}
44              ;
45  neParamList ::= INT IDENTIFIER:v
46                  {: List<String> l = new LinkedList<String>();
47                     l.add(v);
48                     RESULT = l;
49                  :}
50                |  neParamList:l COMMA INT IDENTIFIER:v
51                  {: l.add(v); RESULT = l; :}
52                ;
53
54  declaration ::= INT IDENTIFIER:v SEMICOLON {: RESULT = new Declaration(v); :} ;
55
56  declarations ::= /* epsilon */ {: RESULT = new LinkedList<Declaration>(); :}
57                 |  declarations:l declaration:d {: l.add(d); RESULT = l;    :}
58                 ;
59
60  statement ::= LBRACE statementList:l RBRACE {: RESULT = new Block(l);     :}
61              |  IDENTIFIER:v ASSIGN expr:e    {: RESULT = new Assign(v, e); :}
62              |  IF LPAREN boolExpr:b RPAREN statement:s
63                 {: RESULT = new IfThen(b, s); :}
64              |  IF LPAREN boolExpr:b RPAREN statement:t ELSE statement:e
65                 {: RESULT = new IfThenElse(b, t, e); :}
66              |  WHILE LPAREN boolExpr:b RPAREN statement:s
67                 {: RESULT = new While(b, s); :}
68              |  RETURN expr:e    {: RESULT = new Return(e);        :}
69              |  expr:e           {: RESULT = new ExprStatement(e); :}
70              ;
71
72  statementList ::= /* epsilon */ {: RESULT = new LinkedList<Statement>(); :}
73                  |  statement:s SEMICOLON statementList:l
74                     {: List<Statement> r = new LinkedList<Statement>();
75                        r.add(s);
76                        r.addAll(l);
77                        RESULT = r;
78                     :}
79                  ;
```

Abbildung 16.6: Der erste Teil der *Java*-CUP-Grammatik.

```
1   expr ::= expr:l PLUS  expr:r     {: RESULT = new Sum(       l, r); :}
2         |  expr:l MINUS expr:r     {: RESULT = new Difference(l, r); :}
3         |  expr:l TIMES expr:r     {: RESULT = new Product(   l, r); :}
4         |  expr:l SLASH expr:r     {: RESULT = new Quotient(  l, r); :}
5         |  LPAREN expr:e RPAREN    {: RESULT = e;                    :}
6         |  NUMBER:n                {: RESULT = new MyNumber(n);      :}
7         |  IDENTIFIER:v            {: RESULT = new Variable(v);      :}
8         |  IDENTIFIER:n LPAREN exprList:l RPAREN
9            {: RESULT = new FunctionCall(n, l); :}
10        ;
11
12  exprList ::= /* epsilon */ {: RESULT = new LinkedList<Expr>(); :}
13             |  neExprList:l  {: RESULT = l;                      :}
14             ;
15
16  neExprList ::= expr:e
17                 {: List<Expr> l = new LinkedList<Expr>();
18                    l.add(e);
19                    RESULT = l;
20                 :}
21               |  neExprList:l COMMA expr:e {: l.add(e); RESULT = l; :}
22               ;
23
24  boolExpr ::= expr:l EQUALS  expr:r   {: RESULT = new Equation(      l, r); :}
25            |   expr:l NEQUALS expr:r   {: RESULT = new Inequation(    l, r); :}
26            |   expr:l LE      expr:r   {: RESULT = new LessOrEqual(   l, r); :}
27            |   expr:l GE      expr:r   {: RESULT = new GreaterOrEqual(l, r); :}
28            |   expr:l LT      expr:r   {: RESULT = new LessThan(      l, r); :}
29            |   expr:l GT      expr:r   {: RESULT = new GreaterThan(   l, r); :}
30            |   NOT boolExpr:e          {: RESULT = new Negation(      e   ); :}
31            |   boolExpr:l AND boolExpr:r {: RESULT = new Conjunction(   l, r); :}
32            |   boolExpr:l OR  boolExpr:r {: RESULT = new Disjunction(   l, r); :}
33            ;
```

Abbildung 16.7: Der zweite Teil der *Java*-CUP-Grammatik.

16.3 Darstellung der Assembler-Befehle

Die Abbildungen 16.8 und 16.9 zeigen eine mögliche Übersetzung des Programms zur Berechnung der Summe $\sum_{i=1}^{n} i$ aus Abbildungen 16.2 in Java-Assembler. Um solche Assembler-Programme innerhalb des Programms darstellen zu können, implementieren wir für jeden Java-Assembler-Befehl eine eigene Klasse, die diesen Befehl darstellen kann. Wir werden diese Klassen mit Hilfe von Ep erzeugen, wobei wir auch für Pseudo-Befehle, wie beispielsweise die Befehle "`.constant`" und "`.end-constant`" eine Klasse erzeugen. Die von Ep generierten Klassen werden anschließend von Hand angepasst. Abbildung 16.10 zeigt die Ep-Definition. Einige Details dieser Klassen erläutern wir jetzt:

```
1   .constant
2   OBJREF  64
3   .end-constant
4
5   .main
6   .var
7       n
8   .end-var
9       in
10      istore n
11      iload n
12      bipush 48
13      isub
14      istore n
15      ldc_w OBJREF
16      iload n
17      invokevirtual sum
18      bipush 48
19      iadd
20      out
21      bipush 1
22      pop
23      bipush 10
24      out
25      bipush 1
26      pop
27      halt
28  .end-main
```

Abbildung 16.8: Eine Übersetzung des Programms aus Abbildung 16.2 in Java-Assembler, 1. Teil.

1. Wir haben in den Klassen `GOTO`, `IFEQ`, `IFLT` und `IF_ICMPEQ` das Sprungziel als Zahl dargestellt. Später wird bei der Ausgabe eines Sprungziels diesen Zahlen noch der Buchstabe "`l`" vorangestellt, so dass das Sprungziel als String gelesen werden kann. Um die Generierung von Sprungzielen zu verstehen, betrachten wir die Klasse `LABEL`, die in Abbildung 16.11 gezeigt wird. Diese Klasse verfügt über die statische Variable `sLabelCount`, die in Zeile 2 mit 0 initialisiert wird. Der Konstruktor der Klasse `LABEL` erzeugt bei jedem Aufruf ein neues Objekt, dessen Member-Variable `mLabel` einen nur einmal vergebenen Wert hat. Dies wird dadurch erreicht, dass die statische Variable `sLabelCount` nach jedem Anlegen eines neuen Objektes vom Typ `LABEL` inkrementiert wird. Dadurch wird sichergestellt, dass zwei verschiedene Objekte der Klasse `LABEL` später tatsächlich verschiedene Sprungziele bezeichnen.

```
1   .method sum(n)
2   .var
3       s
4   .end-var
5       bipush 0
6       istore s
7    l1:
8       iload n
9       bipush 0
10      isub
11      iflt l3
12      bipush 1
13      goto l4
14   l3:
15      bipush 0
16   l4:
17      bipush 0
18      if_icmpeq l2
19      iload s
20      iload n
21      iadd
22      istore s
23      iload n
24      bipush 1
25      isub
26      istore n
27      goto l1
28   l2:
29      iload s
30      ireturn
31  .end-method
```

Abbildung 16.9: Übersetzung des Programms aus Abbildung 16.2 in Java-Assembler, 2. Teil.

Zeile 9 zeigt die Umwandlung eines `LABEL`-Objektes in einen String, die später zur Ausgabe der Assembler-Kommandos benutzt wird. Der eindeutigen Zahl wird der Buchstabe "`l`" vor- und der Doppelpunkt "`:`" nachgestellt, damit die Ausgabe später der Syntax des IJVM-Assemblers entspricht.

2. Neben den eigentlichen Assembler-Kommandos zeigt Abbildung 16.10 noch die Definition von Klassen wie beispielsweise der Klasse `MAIN`, deren Implementierung in Abbildung 16.12 gezeigt wird. Diese Klasse dient nur dazu, die Direktive "`.main`" auszugeben. Die Klassen `END_MAIN`, `METHOD`, `END_METHOD`, `VAR`, `END_VAR`, `CONSTANT` und `END_CONSTANT` haben eine analoge Funktion.

   Die Klasse `CONSTANT_DEF` dient dazu, Konstanten-Definition darzustellen. Diese werden später in dem Abschnitt zur Definition der Konstanten in der Form

   Name Value

   ausgegeben.

```
1  AssemblerCmd = IADD()
2               + ISUB()
3               + IMUL()
4               + IDIV()
5               + IAND()
6               + IOR()
7               + SRA1()
8               + SLL8()
9               + POP()
10              + DUP()
11              + SWAP()
12              + OUT()
13              + IN()
14              + BIPUSH(Integer number)
15              + LDC_W(String name)
16              + IINC(String var, Integer number)
17              + ILOAD(String var)
18              + ISTORE(String var)
19              + GOTO(Integer label)
20              + IFEQ(Integer label)
21              + IFLT(Integer label)
22              + IF_ICMPEQ_EQ(Integer label)
23              + INVOKE(String name)
24              + IRETURN()
25              + HALT()
26              + CONSTANT()
27              + CONSTANT_DEF(String name, Integer value)
28              + END_CONSTANT()
29              + VAR()
30              + VAR_DEF(String name)
31              + END_VAR()
32              + MAIN()
33              + END_MAIN()
34              + METHOD(String name, List<String> argList)
35              + END_METHOD()
36              + LABEL(Integer label)
37              + NEWLINE();
```

Abbildung 16.10: Darstellung der Assembler-Befehle als Klassen.

16.4 Die Code-Erzeugung

Nun haben wir alles Material zusammen, um die eigentliche Code-Erzeugung diskutieren zu können. Wir gliedern unsere Darstellung, indem wir die Übersetzung arithmetischer Ausdrücke, Boole'schen Ausdrücke, Befehle und Funktionen getrennt behandeln.

16.4.1 Übersetzung arithmetischer Ausdrücke

Die Übersetzung eines arithmetischen Ausdrucks *expr* soll Code erezeugen, durch dessen Ausführung das Ergebnis der Auswertung auf den Stack gelegt wird. Zu diesem Zweck deklariert die abstrakte Klasse `Expr` die Methode

```
public abstract List<AssemblerCmd> compile();
```

```
1  public class LABEL extends AssemblerCmd {
2      private static  Integer sLabelCount = 0;
3      private         Integer mLabel;
4
5      public LABEL() {
6          mLabel = ++sLabelCount;
7      }
8      public Integer getLabel() { return mLabel;             }
9      public String  toString() { return "l" + mLabel + ":"; }
10 }
```

Abbildung 16.11: Die Klasse `LABEL`.

```
1  public class MAIN extends AssemblerCmd {
2      public MAIN() {}
3      public String toString() { return ".main"; }
4  }
```

Abbildung 16.12: Die Klasse `MAIN`.

die als Ergbnis eine Liste von Assembler-Kommandos erzeugt. Werden diese Kommandos ausgeführt, so liegt anschließend der Wert des Ausdrucks auf dem Stack. Wir betrachten jetzt die Übersetzung der verschiedenen arithmetischen Ausdrücke der Reihe nach.

Übersetzung einer Variablen

Um eine Variable v auszuwerten, laden wir diese Variable mit dem Kommando

`iload` v

auf den Stack. Daher hat die Klasse `Variable` die in Abbildung 16.13 gezeigte Form. Die Klasse Variable verwaltet eine Member-Variable mit dem Namen `mName`, die den Namen der Variablen angibt. Die Methode *compile*() legt zunächst in Zeile 10 eine neue Liste von Assembler-Kommandos an und erzeugt dann in Zeile 11 das Assembler-Kommando

```
iload mName
```

das als einziges Kommando in diese Liste eingefügt wird. Anschließend kann die Liste als Ergebnis zurück gegeben werden.

Übersetzung einer Konstanten

Bei der Übersetzung einer Konstanten gibt es zwei Fälle:

1. Falls die Konstante c durch ein Byte dargestellt werden kann, wenn also die Ungleichungen

   $$-128 \leq c < 128$$

   erfüllt sind, dann kann die Konstante mit Hilfe des Befehl

   `bipush` c

   auf den Stack gelegt werden.

```
1  public class Variable extends Expr {
2      private String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd iload = new ILOAD(mName);
10         result.add(iload);
11         return result;
12     }
13     public String getName()  { return mName; }
14 }
```

Abbildung 16.13: Die Klasse `Variable`.

2. Andernfalls wird die Konstante zunächst im Konstantenpool unter einem eindeutigen Namen n abgelegt und kann dann mit dem Befehl

   `ldc_w` n

   auf den Stack geladen werden.

Ein einfaches Schema zur Namensgenerierung besteht darin, dass wir vor den Dezimalwert der Konstante einfach den Buchstaben "`c`" setzen und den resultierenden String als Namen verwenden. Die Konstante 234 würde also im Konstanten-Pool unter dem String "`c234`" abgelegt.

```
1  public class MyNumber extends Expr {
2      private Integer      mNumber;
3      static  Set<Integer> sConstants = new TreeSet<Integer>();
4
5      public MyNumber(Integer number) {
6          mNumber = number;
7      }
8      public List<AssemblerCmd> compile() {
9          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
10         if (-128 <= mNumber && mNumber < 128) {
11             result.add(new BIPUSH(mNumber));
12         } else {
13             sConstants.add(mNumber);
14             AssemblerCmd ldc = new LDC_W("c" + mNumber);
15             result.add(ldc);
16         }
17         return result;
18     }
19     public Integer getNumber() { return mNumber; }
20 }
```

Abbildung 16.14: Die Klasse `MyNumber`.

Abbildung 16.14 zeigt die Implementierung der Klasse `MyNumber`, die eine Konstante darstellt. Die Konstante selbst wird in der in Zeile 4 deklarierten Member-Variablen `mNumber` gespeichert. Zusätzlich enthält die Klasse aber noch die in Zeile 5 deklarierte statische Variable `sConstants`. Hier handelt es sich um die Menge aller Konstanten, die später in den Konstanten-Pool eingetragen werden müssen.

Die Methode *compile*() überprüft zunächst, ob die Konstante so klein ist, dass sie durch ein Byte dargestellt werden kann. In diesem Fall wird die Auswertung der Konstante in Zeile 13 durch den Assembler-Befehl

```
bipush mNumber
```

übersetzt. Andernfalls wird die Konstante in Zeile 15 dem Konstanten-Pool hinzugefügt und die Auswertung der Konstante wird dann in Zeile 16 durch den Assembler-Befehl

```
ldc_w "c" + mNumber
```

übersetzt.

Übersetzung zusammengesetzter Ausdrücke

Um einen Ausdruck der Form

lhs "`+`" *rhs*

zu übersetzen, muss zunächst Code erzeugt werden, der die Ausdrücke *lhs* und *rhs* rekursiv übersetzt. Wird dieser Code ausgeführt, so liegen auf dem Stack anschließend die Werte von *lhs* und *rhs*. Durch den Befehl `iadd` werden diese nun addiert. Die Übersetzung kann also wie folgt spezifiziert werden:

$$compile(lhs \text{ "+" } rhs) = lhs.compile() + rhs.compile() + [\texttt{iadd}],$$

wobei der Operator "`+`" hier die Verkettung von Listen bezeichnet. Abbildung 16.15 zeigt die Umsetzung dieser Überlegung.

```
1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         result.addAll(mLhs.compile());
12         result.addAll(mRhs.compile());
13         result.add(new IADD());
14         return result;
15     }
16     public Expr getLhs()     { return mLhs; }
17     public Expr getRhs()     { return mRhs; }
18 }
```

Abbildung 16.15: Die Klasse `Sum`.

Die Übersetzung von Ausdrücken der Form

$$lhs - rhs, \quad lhs * rhs \quad \text{und} \quad lhs/rhs$$

verläuft nach dem selben Schema. Statt des Befehls `iadd` verwenden wir hier die entsprechenden Befehle `isub`, `imul` und `idiv`.[1]

Übersetzung von Funktions-Aufrufen

Ein Funktions-Aufruf der $f(e_1, \cdots, e_n)$ kann übersetzt werden, indem zunächst die Ausdrücke e_1, $\cdots$, e_n übersetzt werden. Anschließend wird dann die Funktion f mit Hilfe des Kommandos `invokevirtual` aufgerufen. Dabei muss vor den Ausdrücken e_i noch die Konstante OBJREF auf den Stack gelegt werden. Damit hat die Übersetzung im Allgemeinen die folgende Form:

$$compile\big(f(e_1, \cdots, e_n)\big) = [\texttt{ldc\_w}\ \textsc{Objref}] + compile(e_1) + \cdots + compile(e_n) + [\texttt{invokevirtual}\ f]$$

Allerdings müssen wir noch zwei Sonderfälle berücksichtigen. Falls es sich bei der Funktion f um die Funktion *getChar*() handelt, so können wir diese einfach in das Kommando `in` zum Lesen eines Bytes übersetzen:

$$compile\big(\texttt{getchar}()\big) = [\texttt{in}].$$

Wenn der Funktions-Aufruf f die Form $putchar(e)$ hat, so werten wir zunächst den Ausdruck e aus und rufen dann den Befehl `out` auf:

$$compile\big(\texttt{putchar}(e)\big) = e.compile() + [\texttt{out}].$$

Anschließend legen wir noch mit dem Befehl `bipush` eine 1 auf den Stack. Der Grund dafür ist, dass jeder Funktions-Aufruf ein Ergebnis auf den Stack legen muss, auch dann wenn die Funktion eigentlich gar kein Ergebnis produziert.

Abbildung 16.16 zeigt die Implementierung der Klasse `FunctionCall`, die einen Funktions-Aufruf repräsentiert. Die Klasse hat zwei Member-Variablen.

1. `mName` ist der Name der aufgerufenen Funktion.
2. `mArgs` ist die Liste der Argumente, mit der die Funktion aufgerufen wird.

In der Methode *compile*() wird zunächst überprüft, ob es sich bei dem zu übersetzenden Funktions-Aufruf um einen Aufruf der Funktionen *getchar*() oder *putchar*() handelt und diese Fälle werden dann in der oben beschriebenen Weise abgehandelt. Bei der Übersetzung von *putchar*() ist noch darauf zu achten, dass die Methode ein Ergebnis auf den Stack schreiben muss. Daher wird nach dem `out`-Befehl in Zeile 21 noch ein `bipush`-Befehl angefügt. Andernfalls fügen wir in Zeile 22 an den Anfang der Ergebnisliste einen Befehl, der die Konstante OBJREF auf den Stack legt. Anschließend iterieren wir über die Argumente des Funktions-Aufrufs und fügen für jedes Argument den Code an die Liste von Assembler-Befehlen an, die dieses Argument übersetzt. Zum Schluß wird in Zeile 27 noch der Befehl `invokevirtual` an das Ende der Liste angefügt.

[1] Die Befehle `imul` und `idiv` gehören zwar nicht zu dem ursprünglich von Tanenbaum in [Tan05] definierten Sprach-Umfang der IJVM, in dem Buch [Str07] wird aber gezeigt, dass sich auch diese Befehle im Micro-Code des Prozessors *Mic-1* implementieren lassen.

```
1  public class FunctionCall extends Expr {
2      private String     mName;
3      private List<Expr> mArgs;
4
5      public FunctionCall(String name, List<Expr> args) {
6          mName = name;
7          mArgs = args;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         if (mName.equals("getchar")) {
12             AssemblerCmd in = new IN();
13             result.add(in);
14         } else if (mName.equals("putchar")) {
15             for (Expr arg: mArgs) {
16                 result.addAll(arg.compile());
17             }
18             AssemblerCmd out    = new OUT();
19             AssemblerCmd bipush = new BIPUSH(1);
20             result.add(out);
21             result.add(bipush);
22         } else {
23             AssemblerCmd ldcObjref = new LDC_W("OBJREF");
24             result.add(ldcObjref);
25             for (Expr arg: mArgs) {
26                 result.addAll(arg.compile());
27             }
28             AssemblerCmd invoke = new INVOKE(mName);
29             result.add(invoke);
30         }
31         return result;
32     }
33     public String getName()     { return mName; }
34     public List<Expr> getArgs() { return mArgs; }
35 }
```

Abbildung 16.16: Die Klasse `FunctionCall`.

16.4.2 Übersetzung von Boole'schen Ausdrücken

Boole'sche Ausdrücke werden aus Gleichungen und Ungleichungen mit Hilfe der logischen Operatoren "`!`" (Negation), "`&&`" (Konjunktion) und "`||`" (Disjunktion) aufgebaut. Wir beginnen mit der Übersetzung von Gleichungen.

Übersetzung von Gleichungen

Bevor wir eine Gleichung der Form

lhs == *rhs*

übersetzen können, müssen wir uns überlegen, was der erzeugte Code überhaupt erreichen soll. Eine naheliegende Forderung ist, dass am Ende auf dem Stack eine 1 abgelegt wird, wenn die Werte der beiden Ausdrücke *lhs* und *rhs* übereinstimmen. Andernfalls soll auf dem Stack eine 0 abgelegt werden. Die Übersetzung kann unter diesen Annahmen wie folgt ablaufen:

```
1  public class Equation extends BoolExpr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Equation(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = mLhs.compile();
11         result.addAll(mRhs.compile());
12         LABEL        trueLabel = new LABEL();
13         LABEL        nextLabel = new LABEL();
14         AssemblerCmd isub      = new ISUB();
15         AssemblerCmd ifeq      = new IFEQ(trueLabel.getLabel());
16         AssemblerCmd bipush0   = new BIPUSH(0);
17         AssemblerCmd gotoNext  = new GOTO(nextLabel.getLabel());
18         AssemblerCmd bipush1   = new BIPUSH(1);
19         result.add(isub);
20         result.add(ifeq);
21         result.add(bipush0);
22         result.add(gotoNext);
23         result.add(trueLabel);
24         result.add(bipush1);
25         result.add(nextLabel);
26         return result;
27     }
28     public Expr   getLhs()   { return mLhs; }
29     public Expr   getRhs()   { return mRhs; }
30 }
```

Abbildung 16.17: Die Klasse `Equation`.

1. Zunächst erzeugen wir in den Zeilen 10 und 11 den Code zur Auswertung von *lhs* und *rhs*. Wenn dieser Code abgearbeitet worden ist, liegen die Werte von *lhs* und *rhs* auf dem Stack.
2. Anschließend subtrahieren wir diese Werte mit dem Befehl `isub`. Wenn die Werte gleich waren, liegt jetzt eine 0 auf dem Stack.
3. Dies können wir mit dem Befehl `ifeq` überprüfen. Liegt eine 0 auf dem Stack, so schreiben wir statt dessen eine 1 auf den Stack, andernfalls schreiben wir eine 0 auf den Stack.

Damit hat der erzeugte Code insgesamt die folgende Form

$$
\begin{array}{lcl}
\mathit{compile}(\mathit{lhs}\texttt{ == }\mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\
& + & \mathit{rhs}.\mathit{compile}() \\
& + & [\,\texttt{isub}\,] \\
& + & [\,\texttt{ifeq}\ \mathit{true}\,] \\
& + & [\,\texttt{bipush}\ 0\,] \\
& + & [\,\texttt{goto}\ \mathit{next}\,] \\
& + & [\,\mathit{true}\texttt{:}\,] \\
& + & [\,\texttt{bipush}\ 1\,] \\
& + & [\,\mathit{next}\texttt{:}\,]
\end{array}
$$

Diese Gleichung ist in der Methode *compile*() eins zu eins umgesetzt worden.

Übersetzung von negierten Gleichungen

Die Übersetzung einer negierten Gleichung der Form

lhs `!=` *rhs*

verläuft analog zu der Übersetzung einer Gleichung, denn wir müssen hier nur die Rollen von 0 und 1 vertauschen. Daher lautet die Spezifikation

$$
\begin{array}{lcl}
\mathit{compile}(\mathit{lhs}\ \texttt{!=}\ \mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\
& + & \mathit{rhs}.\mathit{compile}() \\
& + & [\ \texttt{isub}\ ] \\
& + & [\ \texttt{ifeq}\ \mathit{false}\ ] \\
& + & [\ \texttt{bipush}\ 1\ ] \\
& + & [\ \texttt{goto}\ \mathit{next}\ ] \\
& + & [\ \mathit{false}\texttt{:}\ ] \\
& + & [\ \texttt{bipush}\ 0\ ] \\
& + & [\ \mathit{next}\texttt{:}\ ]
\end{array}
$$

Dies kann wieder eins zu eins umgesetzt werden. Aus Platzgründen verzichten wir darauf, die Klasse `Inequation` zu präsentieren.

Übersetzung von Ungleichungen

Wir zeigen exemplarisch, wie eine Ungleichung der Form

lhs `<=` *rhs*

übersetzt werden kann. Die Grundidee besteht darin, die folgende mathematische Äquivalenz zu benutzen:

$$k \leq m \ \Leftrightarrow\ k < m+1 \ \Leftrightarrow\ k-m-1 < 0.$$

Auf dem Rechner kann diese Äquivalenz zwar durch einen Überlauf falsch werden, aber diesen Fall werden wir bei der Implementierung vernachlässigen, denn reale Prozessoren implementieren die Relation $\leq$ unmittelbar, so dass dort das Problem eines Überlaufs nicht auftritt.

Die obige Äquivalenz ermöglicht es, die Relation $\leq$ auf einen Test der Negativität zurückzuführen. Dies ist notwendig, denn nur die Relation $x < 0$ kann auf dem Prozessor effektiv getestet werden. Damit können wir nun die Übersetzung wie folgt spezifizieren:

$$
\begin{array}{lcl}
\mathit{compile}(\mathit{lhs}\ \texttt{<=}\ \mathit{rhs}) & = & \mathit{lhs}.\mathit{compile}() \\
& + & \mathit{rhs}.\mathit{compile}() \\
& + & [\ \texttt{isub}\ ] \\
& + & [\ \texttt{bipush}\ 1\ ] \\
& + & [\ \texttt{isub}\ ] \\
& + & [\ \texttt{iflt}\ \mathit{true}\ ] \\
& + & [\ \texttt{bipush}\ 0\ ] \\
& + & [\ \texttt{goto}\ \mathit{next}\ ] \\
& + & [\ \mathit{true}\texttt{:}\ ] \\
& + & [\ \texttt{bipush}\ 1\ ] \\
& + & [\ \mathit{next}\texttt{:}\ ]
\end{array}
$$

Die Umsetzung dieser Idee ist in Abbildung 16.18 zu sehen. Die Übersetzung von Ungleichungen der Form

lhs `<` *rhs*, *lhs* `>` *rhs*, und *lhs* `>=` *rhs*

verfläuft im Wesentlichen analog und wird daher nicht weiter diskutiert.

```
1  public class LessOrEqual extends BoolExpr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public LessOrEqual(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = mLhs.compile();
11         result.addAll(mRhs.compile());
12         LABEL        trueLabel   = new LABEL();
13         LABEL        nextLabel   = new LABEL();
14         AssemblerCmd isub        = new ISUB();
15         AssemblerCmd bipush1     = new BIPUSH(1);
16         AssemblerCmd iflt        = new IFLT(trueLabel.getLabel());
17         AssemblerCmd bipush0     = new BIPUSH(0);
18         AssemblerCmd gotoNext    = new GOTO(nextLabel.getLabel());
19         result.add(isub);
20         result.add(bipush1);
21         result.add(isub);
22         result.add(iflt);
23         result.add(bipush0);
24         result.add(gotoNext);
25         result.add(trueLabel);
26         result.add(bipush1);
27         result.add(nextLabel);
28         return result;
29     }
30     public Expr   getLhs()   { return mLhs; }
31     public Expr   getRhs()   { return mRhs; }
32 }
```

Abbildung 16.18: Die Klasse `LessOrEqual`.

Übersetzung von Konjunktionen

Die Übersetzung einer Konjunktion der Form

lhs `&&` *rhs*

kann wie folgt spezifiziert werden:

$$\begin{aligned} \mathit{compile}(\mathit{lhs}\ \texttt{\&\&}\ \mathit{rhs}) &= \mathit{lhs}.\mathit{compile}() \\ &+ \mathit{rhs}.\mathit{compile}() \\ &+ [\ \texttt{iand}\ ] \end{aligned}$$

Abbildung 16.19 zeigt die Umsetzung dieser Gleichung.

Die obige Umsetzung entspricht allerdings nicht dem, was in der Sprache `C` tatsächlich passiert. Dort wird die Auswertung eines Ausdrucks der Form

lhs `&&` *rhs*

abgebrochen, sobald das Ergebnis der Auswertung feststeht. Liefert die Auswertung von *lhs* als Ergebnis eine 0, so wird der Ausdruck *rhs* nicht mehr ausgewertet. Falls dieser Ausdruck Seiteneffekte hat, ist das Ergebnis der Auswertung dann also verschieden von unserer Auswertung.

```
1  public class Conjunction extends BoolExpr {
2      private BoolExpr mLhs;
3      private BoolExpr mRhs;
4
5      public Conjunction(BoolExpr lhs, BoolExpr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = mLhs.compile();
11         result.addAll(mRhs.compile());
12         AssemblerCmd iand = new IAND();
13         result.add(iand);
14         return result;
15     }
16     public BoolExpr getLhs()   { return mLhs; }
17     public BoolExpr getRhs()   { return mRhs; }
18 }
```

Abbildung 16.19: Die Klasse `Conjunction`.

Eine Disjunktion wird in analoger Weise auf den Assembler-Befehl `ior` zurück geführt.

Übersetzung von Negationen

Die Übersetzung einer Negation der Form `!`*expr* kann nicht so geradlinig behandelt werden wie die Übersetzung von Konjunktionen und Disjunktionen. Das liegt daran, dass es einen Assembler-Befehl `inot`, der den oben auf dem Stack liegenden Wert negiert, nicht gibt. Aber es geht auch anders, denn weil wir die Wahrheitswerte durch 1 und 0 darstellen, können wir die Negation arithmetisch wie folgt spezifizieren:

$$!x = 1 - x.$$

Damit verläuft die Übersetzung einer Negation nach dem folgenden Schema:

$$\begin{array}{lcl} \mathit{compile}(\texttt{!}\mathit{expr}) & = & [\,\texttt{bipush}\,1\,] \\ & + & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{isub}\,] \end{array}$$

Abbildung 16.20 zeigt die Umsetzung dieser Idee.

```
1  public class Negation extends BoolExpr {
2      private BoolExpr mExpr;
3
4      public Negation(BoolExpr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd bipush1 = new BIPUSH(1);
10         AssemblerCmd isub    = new ISUB();
11         result.add(bipush1);
12         result.addAll(mExpr.compile());
13         result.add(isub);
14         return result;
15     }
16     public BoolExpr getExpr()  { return mExpr; }
17 }
```

Abbildung 16.20: Die Klasse `Negation`.

16.4.3 Übersetzung der Befehle

Als nächstes zeigen wir, wie die einzelnen Befehle übersetzt werden können. Dazu legen wir zunächst fest, dass die Ausführung eines Befehls den Stack nicht verändern darf, alle Argumente, die auf dem Stack zwischendurch abgelegt werden, müssen am Ende auch wieder verschwinden!

Übersetzung von Zuweisungen

Wir untersuchen als erstes, wie eine Zuweisung der Form

x = *expr*

übersetzt werden kann. Die Grundidee besteht darin, zunächst den Ausdruck *expr* auszuwerten. Als Folge dieser Auswertung wird dann ein Wert auf dem Stack zurück bleiben, der das Ergebnis dieser Auswertung ist. Diesen Wert können wir mit dem Befehl `istore` unter der Variable x abspeichern. Folglich kann die Übersetzung einer Zuweisung wie folgt spezifiziert werden:

$$\begin{array}{lcl} \mathit{compile}(x\texttt{=}\mathit{expr}) & = & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{istore}\,x\,] \end{array}$$

Die Idee wird in der in Abbildung 16.21 gezeigten Klasse `Assign` umgesetzt.

Übersetzung von Ausdrücken als Befehlen

Die Übersetzung eines Ausdrucks, der als Befehl verwendet wird, birgt eine Tücke: Die Übersetzung des Ausdrucks selber hinterläßt auf dem Stack einen Wert. Dieser muss aber bei Beendigung des Befehls vom Stack entfernt werden! Daher müssen wir den Befehl `pop` an das Ende der Liste der Assembler-Befehle anfügen, die bei der Übersetzung des Ausdrucks erzeugt werden. Die Übersetzung eines Befehls vom Typ `ExprStatement` wird also wie folgt spezifiziert:

$$\begin{array}{lcl} \mathit{compile}(\mathit{expr}\texttt{;}) & = & \mathit{expr}.\mathit{compile}() \\ & + & [\,\texttt{pop}\,] \end{array}$$

Abbildung 16.22 zeigt die Klasse `ExprStatement`, in der diese Überlegung umgesetzt wird.

```
1  public class Assign extends Statement {
2      private String mVar;
3      private Expr   mExpr;
4
5      public Assign(String var, Expr expr) {
6          mVar  = var;
7          mExpr = expr;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result   = mExpr.compile();
11         AssemblerCmd       storeCmd = new ISTORE(mVar);
12         result.add(storeCmd);
13         return result;
14     }
15     public String getVar()  { return mVar;  }
16     public Expr   getExpr() { return mExpr; }
17 }
```

Abbildung 16.21: Die Klasse `Assign`.

```
1  public class ExprStatement extends Statement {
2      private Expr mExpr;
3
4      public ExprStatement(Expr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile() {
8        List<AssemblerCmd> result = mExpr.compile();
9        AssemblerCmd       popCmd = new POP();
10       result.add(popCmd);
11       return result;
12     }
13     public Expr   getExpr()  { return mExpr; }
14 }
```

Abbildung 16.22: Die Klasse `ExprStatement`.

Die Übersetzung von Verzweigungs-Befehlen

Als nächstes überlegen wir, wie ein Verzweigungs-Befehl der Form

`if` (*expr*) *statement*

übersetzt werden kann. Offenbar muss zunächst der Boole'sche Ausdruck *expr* übersetzt werden. Die Auswertung dieses Ausdrucks wird auf dem Stack entweder eine 1 oder eine 0 hinterlassen, je nachdem, ob die Bedingung des Tests wahr oder falsch wahr. Mit dem Befehl `ifeq` können wir überprüfen, welcher dieser beiden Fälle vorliegt. Das führt zu der folgenden Spezifikation:

$$
\begin{array}{lcl}
\mathit{compile}(\texttt{if}\ (\mathit{expr})\ \mathit{statement}) & = & \mathit{expr}.\mathit{compile}() \\
& + & [\ \texttt{ifeq}\ \mathit{else}\ ] \\
& + & \mathit{statement}.\mathit{compile}() \\
& + & [\ \mathit{else}\texttt{:}\ ]
\end{array}
$$

Diese Spezifikation ist in der Abbildung 16.23 umgesetzt worden.

```
1  public class IfThen extends Statement {
2      private BoolExpr  mBoolExpr;
3      private Statement mStatement;
4
5      public IfThen(BoolExpr boolExpr, Statement statement) {
6          mBoolExpr = boolExpr;
7          mStatement = statement;
8      }
9          public List<AssemblerCmd> compile() {
10                 List<AssemblerCmd> result = mBoolExpr.compile();
11                 LABEL        elseLabel = new LABEL();
12                 AssemblerCmd ifeq      = new IFEQ(elseLabel.getLabel());
13                 result.add(ifeq);
14                 result.addAll(mStatement.compile());
15                 result.add(elseLabel);
16                 return result;
17         }
18     public BoolExpr getBoolExpr() {
19         return mBoolExpr;
20     }
21     public Statement getStatement() {
22         return mStatement;
23     }
24 }
```

Abbildung 16.23: Die Klasse `IfThen.java`

Die Übersetzung eines Verzweigungs-Befehls der Form

`if` (*condition*) *thenStmnt* `else` *elseStmnt*

erfolgt in analoger Art und Weise. Diesmal lautet die Spezifikation:

$$
\begin{aligned}
\mathit{compile}(\texttt{if}\ (\mathit{condition})\ \mathit{thenStmnt}\ \texttt{else}\ \mathit{elseStmnt}) \quad &= \quad \mathit{cond.compile}() \\
&+ \quad [\ \texttt{ifeq}\ \mathit{else}\ ] \\
&+ \quad \mathit{thenStmnt.compile}() \\
&+ \quad [\ \texttt{goto}\ \mathit{next}\ ] \\
&+ \quad [\ \mathit{else}\texttt{:}\ ] \\
&+ \quad \mathit{elseStmnt.compile}()\ ] \\
&+ \quad [\ \mathit{next}\texttt{:}\ ]
\end{aligned}
$$

Diese Spezifikation ist in der Abbildung 16.24 umgesetzt worden.

Die Übersetzung einer Schleife

Die Übersetzung einer `while`-Schleife der Form

`while` (*cond*) *statement*

```
1  public class IfThenElse extends Statement {
2      private BoolExpr  mCondition;
3      private Statement mThen;
4      private Statement mElse;
5
6      public IfThenElse(BoolExpr condition, Statement thenStmnt, Statement elseStmnt) {
7          mCondition = condition;
8          mThen      = thenStmnt;
9          mElse      = elseStmnt;
10     }
11         public List<AssemblerCmd> compile() {
12                 List<AssemblerCmd> result = mCondition.compile();
13                 LABEL        elseLabel = new LABEL();
14                 LABEL        nextLabel = new LABEL();
15                 AssemblerCmd ifeq      = new IFEQ(elseLabel.getLabel());
16                 AssemblerCmd gotoNext  = new GOTO(nextLabel.getLabel());
17                 result.add(ifeq);
18                 result.addAll(mThen.compile());
19                 result.add(gotoNext);
20                 result.add(elseLabel);
21                 result.addAll(mElse.compile());
22                 result.add(nextLabel);
23                 return result;
24         }
25     public BoolExpr getCondition() {
26         return mCondition;
27     }
28     public Statement getThen() {
29         return mThen;
30     }
31     public Statement getElse() {
32         return mElse;
33     }
34 }
```

Abbildung 16.24: Die Klasse `IfThenElse`.

orientiert sich an der folgenden Spezifikation:

$$
\begin{array}{lcl}
\mathit{compile}(\texttt{while}\ (\mathit{cond})\ \mathit{stmnt}) & = & [\ \mathit{loop}\texttt{:}\ ] \\
& + & \mathit{cond}.\mathit{compile}() \\
& + & [\ \texttt{ifeq}\ \mathit{next}\ ] \\
& + & \mathit{stmnt}.\mathit{compile}() \\
& + & [\ \texttt{goto}\ \mathit{loop}\ ] \\
& + & [\ \mathit{next}\texttt{:}\ ]
\end{array}
$$

Die Umsetzung dieser Spezifikation sehen Sie in Abbildung 16.25.

Übersetzen einer Liste von Befehlen

Eine in geschweiften Klammern eingeschlossene Liste von Befehlen der Form

$$\{\mathit{stmnt}_1\texttt{;}\ \cdots\ \mathit{stmnt}_n\texttt{;}\}$$

```
1  public class While extends Statement {
2      private BoolExpr  mCondition;
3      private Statement mStatement;
4
5      public While(BoolExpr condition, Statement statement) {
6          mCondition = condition;
7          mStatement = statement;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         LABEL        loopLabel = new LABEL();
12         LABEL        nextLabel = new LABEL();
13         AssemblerCmd bipush0   = new BIPUSH(0);
14         AssemblerCmd if_icmpeq = new IF_ICMPEQ(nextLabel.getLabel());
15         AssemblerCmd gotoLoop  = new GOTO(loopLabel.getLabel());
16         result.add(loopLabel);
17         result.addAll(mCondition.compile());
18         result.add(bipush0);
19         result.add(if_icmpeq);
20         result.addAll(mStatement.compile());
21         result.add(gotoLoop);
22         result.add(nextLabel);
23         return result;
24     }
25     public BoolExpr  getCondition() { return mCondition; }
26     public Statement getStatement() { return mStatement; }
27 }
```

Abbildung 16.25: Die Klasse `While`.

wird dadurch übersetzt, dass die Listen, die bei der Übersetzung der einzelnen Befehle $stmnt_i$ entstehen, aneinander gehängt werden:

$$compile\big(\{stmnt_1\,;\ \cdots\ stmnt_n\,;\}\big) \quad = \quad compile(stmnt_1) + \cdots + compile(stmnt_n).$$

Diese Idee ist in der Klasse `Block` realisiert worden. Abbildung 16.26 zeigt diese Klasse.

16.4.4 Zusammenspiel der Komponenten

Nachdem wir jetzt gesehen haben, wie die einzelnen Teile eines Programms in Listen von Assembler-Befehlen übersetzt werden können, müssen wir noch zeigen, wie die einzelnen Komponenten unseres Programms zusammen spielen. Dazu sind noch zwei Klassen zu diskutieren:

1. Die Klasse `Function` repräsentiert die Definition einer Funktion.
2. Die Klasse `Program` repräsentiert das vollständige Programm.

Wir beginnen mit der Diskussion der Klasse `Function`. Abbildung 16.27 zeigt die Klasse `Function`, allerdings ohne die Implementierung der Methode *compile*(), die wir aus Platzgründen in die Abbildung 16.28 ausgelagert haben.

```
1  public class Block extends Statement {
2      private List<Statement> mStatementList;
3
4      public Block(List<Statement> statementList) {
5          mStatementList = statementList;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          for (Statement stmnt: mStatementList) {
10             result.addAll(stmnt.compile());
11         }
12         return result;
13     }
14     public List<Statement> getStatementList() {
15         return mStatementList;
16     }
17 }
```

Abbildung 16.26: Die Klasse `Block`.

```
1  public class Function {
2      private String            mName;
3      private List<String>      mParameterList;
4      private List<Declaration> mDeclarations;
5      private List<Statement>   mBody;
6
7      public Function(String            name,
8                      List<String>      parameterList,
9                      List<Declaration> declarations,
10                     List<Statement>   body)
11     {
12         mName          = name;
13         mParameterList = parameterList;
14         mDeclarations  = declarations;
15         mBody          = body;
16     }
17     public List<AssemblerCmd> compile() { ... }
18     public String          getName()          { return mName;          }
19     public List<String>    getParameterList() { return mParameterList; }
20     public List<Statement> getBody()          { return mBody;          }
21 }
```

Abbildung 16.27: Die Klasse `Function`.

Die Klasse `Function` enthält vier Member-Variablen:

1. `mName` gibt den Namen der Funktion an.
2. `mParameterList` ist die Liste der Parameter, mit der die Funktion aufgerufen wird.
3. `mDeclarations` ist die Liste der Variablen-Deklarationen.
4. `mBody` ist die Liste von Befehlen, die im Rumpf der Funktion ausgeführt werden.

```
1   public List<AssemblerCmd> compile() {
2       List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
3       AssemblerCmd nl = new NEWLINE();
4       result.add(nl);
5       if (mName.equals("main")) {
6           AssemblerCmd main = new MAIN();
7           AssemblerCmd var  = new VAR();
8           result.add(main);
9           result.add(var);
10          for (Declaration decl: mDeclarations) {
11              AssemblerCmd varDef = new VAR_DEF(decl.getVar());
12              result.add(varDef);
13          }
14          AssemblerCmd endVar = new END_VAR();
15          result.add(endVar);
16          for (Statement stmnt: mBody) {
17              result.addAll(stmnt.compile());
18          }
19          AssemblerCmd halt    = new HALT();
20          AssemblerCmd endMain = new END_MAIN();
21          result.add(halt);
22          result.add(endMain);
23      } else {
24          AssemblerCmd method = new METHOD(mName, mParameterList);
25          AssemblerCmd var    = new VAR();
26          result.add(method);
27          result.add(var);
28          for (Declaration decl: mDeclarations) {
29              AssemblerCmd varDef = new VAR_DEF(decl.getVar());
30              result.add(varDef);
31          }
32          AssemblerCmd endVar  = new END_VAR();
33          result.add(endVar);
34          for (Statement stmnt: mBody) {
35              result.addAll(stmnt.compile());
36          }
37          AssemblerCmd endMethod = new END_METHOD();
38          result.add(endMethod);
39      }
40      return result;
41  }
```

Abbildung 16.28: Die Methode *compile*().

Die eigentliche Arbeit der Klasse `Funktion` wird in der Methode *compile*(), die in Abbildung 16.28 gezeigt ist, geleistet. Es sind zwei Fälle zu unterscheiden:

1. Falls die zu übersetzende Funktion den Namen "`main`" hat, so hat der erzeugte Code die folgende Form:

```
1          .main
2          .var
3             v_1
4             ⋮
5             v_k
6          .end-var
7             s_1
8             ⋮
9             s_n
10            halt
11         .end-main
```

   Hier bezeichnen v_1, $\cdots$, v_k die in der Funktion definierten lokalen Variablen. und s_1, $\cdots$, s_n bezeichnen die einzelnen Assemblerbefehle, die bei der Übersetzung des Rumpfes der Funktion erzeugt werden.

2. Andernfalls hat der erzeugte Code die folgende Form:

```
1          .method mName(p_1, ··· , p_l)
2          .var
3             v_1
4             ⋮
5             v_k
6          .end-var
7             s_1
8             ⋮
9             s_n
10         .end-method
```

   Hier bezeichnen p_1, $\cdots$, p_l die formalen Parameter, die der Funktion übergeben werden, v_1, $\cdots$, v_k bezeichnen wieder die lokalen Variablen und s_1, $\cdots$, s_n sind die Assemblerbefehle des Rumpfes der Funktion.

Zum Abschluss diskutieren wir die Klasse `Program`, die in Abbildung 16.29 gezeigt wird. Diese Klasse verwaltet in der Member-Variablen `mFunctionList` die Liste aller zu übersetzenden Funktionen. Neben dem Code für die einzelnen Funktionen hat diese Klasse die Aufgabe, die Deklaration der verwendeten Konstanten an den Anfang der erzeugten Assembler-Datei zu schreiben. Damit diese möglich ist, müssen aber erst alle Funktionen übersetzt werden, denn bevor die Funktionen nicht alle übersetzt worden sind, ist ja noch gar nicht klar, welche Konstanten überhaupt verwendet worden sind.

Bei der Übersetzung der Funktionen ist darauf zu achten, dass zuerst die Funktion *main*() übersetzt wird, denn diese muss am Anfang der erzeugten Assembler-Datei stehen. In der `C`-Datei ist die Funktion *main*() aber die letzte Funktion, denn in der Sprache `C` müssen alle Funktionen vor ihrer Verwendung deklariert worden sein.

```
1  public class Program {
2      private List<Function> mFunctionList;
3
4      public Program(List<Function> functionList) {
5          mFunctionList = functionList;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> fctList = new LinkedList<AssemblerCmd>();
9          int indexMain = mFunctionList.size() - 1;
10         Function main = mFunctionList.get(indexMain);
11         fctList.addAll(main.compile());
12         for (int i = 0; i < indexMain; ++i) {
13             Function f = mFunctionList.get(i);
14             fctList.addAll(f.compile());
15         }
16         AssemblerCmd constant    = new CONSTANT();
17         AssemblerCmd endConstant = new END_CONSTANT();
18         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
19         result.add(constant);
20         result.add(new CONSTANT_DEF("OBJREF", 64));
21         for (Integer c: MyNumber.sConstants) {
22             result.add(new CONSTANT_DEF("c" + c, c));
23         }
24         result.add(endConstant);
25         result.addAll(fctList);
26         return result;
27     }
28     public List<Function> getFunctionList() {
29         return mFunctionList;
30     }
31 }
```

Abbildung 16.29: Die Klasse `Program`.

Wie übersetzen in Zeile 11 als erstes die Funktion *main*(). Anschließend werden in der Schleife, die sich von Zeile 12 bis 15 erstreckt, die restlichen Funktionen übersetzt. Der erzeugte Code befindet sich dann in der Liste `fctList`. Nach dieser Übersetzung sind dann alle im Programm verwendeten Konstanten bekannt. Wir erinnern uns, dass diese in der statischen Variablen `sConstants` in der Klasse `MyNumber` abgespeichert worden sind. In der Schleife in den Zeilen 21 – 23 werden diese nun in Pseudo-Assembler-Befehle übersetzt, die in die Liste `result` geschrieben werden. An diese Liste wird dann das Ergebnis der Übersetzung der Funktionen angehängt.

Übersetzen wir die in Abbildung 16.2 gezeigte Funktion zur Berechnung der Summe $\sum_{i=1}^{n} i$ mit dem Compiler, so erhalten wir die in den Abbildungen 16.8 und 16.9 gezeigten Assembler-Datei. Vergleichen wir dieses Programm mit dem Assembler-Programm, das wir damals in der Rechnertechnik-Vorlesung entwickelt haben, so fällt auf, dass das vom Compiler erzeugte Programm deutlich länger ist. Es wäre nun Aufgabe eines Code-Optimierers, den erzeugten Code zu verkürzen. Eine Diskussion von Techniken zur Code-Optimierung geht allerdings über den Rahmen der Vorlesung heraus.

Kapitel 17

Register-Zuordnung

Die Übersetzung eines Integer-C-Programms in *Java*-Byte-Code ist deswegen einfach, weil wir uns keine Gedanken darüber machen müssen, wie wir die Variablen auf Register verteilen, denn bei der IJVM gibt es keine für den Benutzer sichtbaren Register, so dass alle Variablen und auch die Zwischenergebnisse auf dem Stack abgelegt werden müssen. In diesem Kapitel ist unser Ziel, als Endergebnis ein Assembler-Programm für den SRP-Prozessor zu erzeugen. Dazu müssen wir uns überlegen, wie wir die einzelnen Variablen auf die verschiedenen Register verteilen. Wir geben zunächst ein Verfahren an, mit dessen Hilfe wir die Anzahl der Register bestimmen können, die für die Speicherung von Zwischenergebnissen, die bei der Auswertung arithmetischer Ausdrücke auftreten, benötigt werden. Die Parameter und lokalen Variablen einer Funktion verteilen wir dann auf die verbleibenden Register oder, falls nicht genügend Register vorhanden sind, allokieren wir die lokalen Variablen auf dem Stack.

17.1 Die Ershov-Zahlen

Das Verfahren, mit dem Register für die Auswertung arithmetischer Ausdrücke zugeordnet werden können, berechnet zunächst für einen gegebenen arithmetischen Ausdruck e die Anzahl $ershov(e)$ der Register, die benötigt werden, um den Ausdruck auszuwerten. Es ist nach dem russischen Informatiker A. P. Ershov benannt, der die Idee des Verfahrens in [Ers58] publiziert hat. Wir spezifizieren die Funktion

$$ershov : \texttt{Expr} \rightarrow \mathbb{N},$$

die für einen gegebenen arithmetischen Ausdruck e die Anzahl der zur Auswertung benötigten Register angibt, durch eine Fallunterscheidung nach dem Aufbau des Ausdrucks.

1. Um eine Konstante auszuwerten, benötigen wir genau ein Register, in dem wir diese Konstante speichern können:

   $$isNumber(c) \rightarrow ershov(c) = 1.$$

2. Analog benötigen wir zur Auswertung einer Variablen genau ein Register, in dem der Wert der Variablen abgelegt wird:

   $$isVariable(v) \rightarrow ershov(v) = 1.$$

3. Zur Auswertung eines Ausdrucks der Form

   $$op\ e,$$

   bei dem der unäre Operator op auf den Ausdruck e angewendet wird, benötigen wir genau so viele Register, wie wir zur Auswertung des Ausdrucks e benötigen, denn wenn das Ergebnis der Auswertung von e in einem Register Rx liegt, so können wir anschließend dieses Register

überschreiben um das Endergebnis der Auswertung abzuspeichern:

$$\mathit{ershov}(\mathit{op}\ e) = \mathit{ershov}(e).$$

Bemerkung: Bei dem RISC-Prozessor, den wir in der Rechnertechnik-Vorlesung besprochen haben, hatten wir als einzigen unären Operator die bitweise Negation, aber prinzipiell sind auch andere unäre Operatoren denkbar.

4. Bei der Auswertung eines binären Ausdrucks der Form

   $$e_1\ \mathit{op}\ e_2$$

   sind drei Fälle zu unterscheiden.

   (a) $\mathit{ershov}(e_1) < \mathit{ershov}(e_2)$.
   In diesem Fall berechnen wir zunächst den Ausdruck e_2 und benötigen dazu $\mathit{ershov}(e_2)$ Register. Das letzte dieser Register, nennen wir es R_y, enthält dann das Ergebnis der Auswertung von e_2, alle anderen Register können für die Berechnung von e_1 verwendet werden. Da $\mathit{ershov}(e_1) < \mathit{ershov}(e_2)$ ist, reichen diese Register auch aus und das Ergebnis von e_1 wird in einem dieser Register, sagen wir mal R_x, abgespeichert. Anschließend führen wir dann noch den Befehl

   $$\mathit{op}\ R_y,\ R_x,\ R_y$$

   aus, bei dem wir das Ergebnis des Ausdrucks $e_1\ \mathit{op}\ e_2$ berechnen und in dem Register R_y abspeichern. Wir sehen, dass wir in diesem Fall zur Berechnung von $e_1\ \mathit{op}\ e_2$ nicht mehr Register benötigen als zur Berechnung von e_2 und halten die folgende Formel fest:

   $$\mathit{ershov}(e_1) < \mathit{ershov}(e_2) \rightarrow \mathit{ershov}(e_1\ \mathit{op}\ e_2) = \mathit{ershov}(e_2).$$

   (b) $\mathit{ershov}(e_1) = \mathit{ershov}(e_2)$.
   In diesem Fall berechnen wir den Ausdruck e_1 und benötigen dafür $\mathit{ershov}(e_1)$ viele Register. Bis auf das letzte dabei benötigte Register, nennen wir es R_x, in dem wir das Ergebnis dieser Rechnung speichern müssen, können wir alle diese Register für die Berechnung des Ausdrucks e_2 wiederverwenden. Folglich benötigen wir für die Berechnung von e_2 also ein weiteres Register, das wir jetzt mit R_y bezeichnen. Das Endergebnis können wir nun mit dem Befehl

   $$\mathit{op}\ R_y,\ R_x,\ R_y$$

   berechnen, so dass wir an dieser Stelle kein weiteres Register mehr benötigen. Wir halten fest:

   $$\mathit{ershov}(e_1) = \mathit{ershov}(e_2) \rightarrow \mathit{ershov}(e_1\ \mathit{op}\ e_2) = \mathit{ershov}(e_1) + 1.$$

   (c) $\mathit{ershov}(e_1) > \mathit{ershov}(e_2)$.
   Dieser Fall ist offenbar analog zum ersten Fall und wir haben:

   $$\mathit{ershov}(e_1) > \mathit{ershov}(e_2) \rightarrow \mathit{ershov}(e_1\ \mathit{op}\ e_2) = \mathit{ershov}(e_1).$$

5. Die Auswertung eines Funktions-Aufrufs der Form

   $$f(e_1, \cdots, e_n)$$

   gelingt in Registern nur dann, wenn die einzelnen Ausdrücke $e_1, \cdots, e_n$ selber keine Funktions-Aufrufe mehr enthalten. Wir werden daher unsere Grammatik so einschränken, dass wir nur solche Funktions-Aufrufe betrachten, bei denen dies der Fall ist. Desweiteren werden wir Funktions-Aufrufe nur auf der rechten Seite einer Zuweisungen zulassen. Dann können wir jedes der Argumente von f getrennt auswerten. Da die Ergebnisse dieser Auswertungen ohnehin auf den Stack geschrieben werden, ist es nicht erforderlich, diese Ergebnisse in Registern zu speichern. Daher haben wir

   $$\mathit{ershov}\big(f(e_1, \cdots, e_n)\big) = \mathit{max}\big(\mathit{ershov}(e_1), \cdots, \mathit{ershov}(e_n)\big).$$

17.2 Implementierung eines Compilers für den SRP

Nach den Vorüberlegungen aus dem letzten Abschnitt entwickeln wir in diesem Abschnitt einen Compiler, der Code für den SRP erzeugt. Wir gehen dabei wieder von der Sprache Integer-`C` aus, für die wir im letzten Kapitel einen Compiler in Byte-Code entwicklet haten. Diese Grammatik hatten wir in Abbildung 16.1 gezeigt.

Die Idee bei der Entwicklung eines Compilers für den SRP ist, dass wir zunächst berechnen, wieviele Register wir benötigen, um die Zwischenergebnisse, die bei der Auswertung arithmetischer Ausdrücke entstehen, zu speichern. Die restlichen Register können dann für die lokalen Variablen genutzt werden. Sollten wir mehr lokale Variablen als freie Register haben, so speichern wir die überzähligen lokalen Variablen auf dem Stack. Wir werden nur die Register `R1` bis `R29` benutzen, denn wir gehen davon aus, dass folgendes gilt:

1. Das Register `R0` speichert die Konstante 0.
2. Das Register `R30` speichert den Stack-Pointer.
3. Das Register `R31` ist für die Berechnung von Sprungadressen und Speicherstellen reserviert.

In der Rechnertechnik-Vorlesung hatten wir eine Unterscheidung in solche Register, die beim Aufruf einer Funktion gesichert werden müssen und solche Register, die nicht gesichert werden, durchgeführt. Diese Unterscheidung lassen wir fallen und vereinbaren statt dessen, dass bei einem Funktionsaufruf alle Register, die von der Funktion verändert werden, gesichert werden müssen. Das vereinfacht die Entwicklung des Compilers etwas.

17.2.1 Diskussion der verschiedenen Klassen

Die Struktur des SRP-Compilers ist ähnlich zu der Struktur des *Java*-Byte-Code-Compilers: Wir erzeugen zunächst einen abstrakten Syntax-Baum und haben dann in allen Klassen eine Methode *compile*(), die Code erzeugt, der den jeweils dargestellten Ausdruck übersetzt. Die Erzeugung des Codes ist allerdings deutlich komplexer als bei dem Byte-Code-Compiler.

Wir beginnen unsere Diskussion des SRP-Compilers mit der Klasse `Function`, die in den Abbildungen 17.1, 17.2, 17.3 und 17.4 gezeigt ist. Abbildung 17.1 auf Seite 291 zeigt die Member-Variablen der Klasse `Function`, den Konstruktor sowie die Methode *compile*(). Die Klasse repräsentiert eine Funktions-Definition der Form

$$\texttt{int}\ f(p_1, \cdots, p_n)\ \{\ \textit{decls}\ \textit{body}\ \}$$

Die Member-Variablen haben folgende Bedeutung:

1. `mName` entspricht dem Funktions-Namen f.
2. `mParameterList` entspricht den Übergabe-Parametern $p_1, \cdots, p_n$.
3. `mDeclarations` ist die Liste der Deklarationen aller lokalen Variablen.
4. `mBody` ist die Liste der Befehle im Rumpf der Funktion.
5. `mNumberUsedRegs` ist die Anzahl der Register, die von dieser Funktion benutzt werden.
6. `mNumberVarsOnStack` ist die Anzahl der lokalen Variablen, die auf dem Stack gespeichert werden müssen, weil in den Registern kein Platz mehr für diese Variablen ist.
7. `mSymbolTable` gibt für jede lokale Variable den Ort an, wo diese lokale Variable gespeichert ist. Zur Spezifikation dieses Ortes haben wir die abstrakte Klasse `Location` eingeführt, die über die Gleichung

   ```
   Location = Memory(Integer offset) + Register(Integer number);
   ```

   definiert ist. Die Klasse `Location` unterscheidet also, ob die Variable in einem Register

```
1  public class Function {
2      private String            mName;
3      private List<String>      mParameterList;
4      private List<Declaration> mDeclarations;
5      private List<Statement>   mBody;
6      private Integer           mNumberUsedRegs;
7      private Integer           mNumberVarsOnStack;
8      private Map<String, Location> mSymbolTable;
9      public  static String sName;
10
11     public Function(String            name,
12                     List<String>      parameterList,
13                     List<Declaration> declarations,
14                     List<Statement>   body)
15     {
16         mName          = name;
17         mParameterList = parameterList;
18         mDeclarations  = declarations;
19         mBody          = body;
20     }
21     public List<AssemblerCmd> compile() {
22         initTable();
23         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
24         result.add(new PROC(mName));
25         result.addAll(saveRegisters());
26         result.addAll(fillRegisters());
27         for (Statement stmnt: mBody) {
28             result.addAll(stmnt.compile(mSymbolTable));
29         }
30         result.add(new NEWLINE());
31         return result;
32     }
```

Abbildung 17.1: Die Klasse `Function`, 1. Teil.

gespeichert ist, oder ob die Variable auf dem Stack liegt. Falls die Variable in einem Register gespeichert wird, dann gibt `mNumber` an, um welches Register es sich handelt. Liegt die Variable auf dem Stack, so wird diese Variable an der Stelle

```
SP + offset
```

gespeichert. Dabei gehen wir davon aus, dass der Stackpointer `SP` immer auf die Rücksprung-Adresse der Funktion zeigt, so dass wir die Adresse der lokalen Variablen tatsächlich immer nach der obigen Formel berechnen können.

8. Zusätzlich hat die Klasse eine statische Variable `sName`, die den Namen der Funktion enthält, die gerade geparst wird.

Der Konstruktor der Klasse `Function` speichert die Member-Variablen, die den abstrakten Syntax-Baum repräsentieren und die bereits vom Parser erzeugt werden können.

Interessanter ist die Methode *compile*(), die diese Funktion übersetzt. Im Gegensatz zu unserer Implementierung des *Java*-Byte-Code-Compilers behandeln wir bei dieser Implementierung die Sonderfälle, die bei den Funktionen *putchar()* und *getchar*() auftreten, nicht sondern gehen davon aus, dass diese Funktionen bereits in einer Bibliothek zur Verfügung gestellt werden und dann

```
33  private List<AssemblerCmd> saveRegisters() {
34      List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
35      for (int i = 1; i <= mNumberUsedRegs; ++i) {
36          result.add(new STOREC(i, mNumberVarsOnStack + i));
37      }
38      int nextFree = mNumberVarsOnStack + mNumberUsedRegs + 1;
39      mSymbolTable.put("$free@" + mName,
40                       new Memory(nextFree));
41      mSymbolTable.put("$numberVarsOnStack@" + mName,
42                       new Memory(mNumberVarsOnStack));
43      mSymbolTable.put("$numberUsedRegs@" + mName,
44                       new Memory(mNumberUsedRegs   ));
45      return result;
46  }
47  private List<AssemblerCmd> fillRegisters() {
48      List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
49      for (int i = 0; i < mParameterList.size(); ++i) {
50          String p = mParameterList.get(i);
51          Location l = mSymbolTable.get(p);
52          if (l instanceof Register) {
53              Integer register = ((Register) l).getNumber();
54              Integer offset   = mParameterList.size() - i;
55              result.add(new LOADC(register, -offset));
56          } else {
57              return result;
58          }
59      }
60      return result;
61  }
```

Abbildung 17.2: Die Klasse `Function`, 2. Teil.

später von einem Linker eingebunden werden. Die Funktion *compile*() arbeitet nun wie folgt:

1. Zunächst berechnen wir für die Parameter und die lokalen Variablen der Funktion, wo diese abgespeichert werden. Diese Berechnung wird von der Methode *initTable*() durchgeführt. Das Ergebnis dieser Rechnung wird in der Variablen `mSymbolTable` abgelegt. Zusätzlich initialisiert diese Methode die Member-Variablen `mNumberUsedRegs` und `mNumberVarsOnStack`.

2. Anschließend erzeugen wir eine Liste von Assembler-Befehlen, die mit der Assembler-Deklaration

   `PROC` *mName*

   beginnt. Als erstes sichert die Prozedur dann alle die Register, die von der Prozedur eventuell verändert werden. Die dafür notwendigen Assembler-Befehle werden von der Funktion *saveRegisters*() erzeugt.

3. Danach werden die Übergabe-Parameter, die wir in Registern speichern wollen, in die entsprechenden Register geladen. Der dafür notwendige Code wird von der Methode *fillRegisters*() erzeugt.

4. Zum Schluss werden die einzelnen Befehle, die im Rumpf der Funktion stehen, übersetzt.

Die Methode *saveRegisters*() erzeugt Code, mit dem die Register, die in der Methode benutzt werden, auf dem Stack gesichert werden. Dazu benutzt Sie den Pseudo-Assembler-Befehl

`storec R`i`, SP + `o,

mit dem das Register `R`i an die Adresse `SP` $+ o$ im Hauptspeicher geschrieben wird. Dieser Pseudo-Befehl wird in die folgenden Assembler-Befehle übersetzt:

```
const R31, o
add   R31, R30, R31
store Ri, R31
```

Anschließend speichern wir die Adresse der ersten freien Stelle auf dem Stack in der Symboltabelle unter dem Namen "$free@" + *name*, wobei `name` für den Namen der Prozedur steht. Genauso werden die Anzahl der Variablen, die wir auf dem Stack haben sowie die Anzahl der benutzten Register in der Symboltabelle abgespeichert.

Die Methode *fillRegisters*() hat die Aufgabe, die Parameter, mit denen die Funktion aufgerufen worden ist, vom Stack in die dafür vorgesehenen Register zu laden. Dazu benutzt diese Methode den Pseudo-Assembler-Befehl

```
loadc Ri, SP + o,
```

mit dem das Register `R`i von der Adresse `SP` $+ o$ im Hauptspeicher geladen wird. Dieser Pseudo-Befehl wird in die folgenden Assembler-Befehle übersetzt:

```
const R31, o
add   R31, R30, R31
load  Ri, R31
```

Bemerkung: Reale Prozessoren haben in der Regel ein Kommando, das dem obigen Pseudobefehl direkt umsetzen kann.

```
62      private void initTable() {
63          mSymbolTable = new TreeMap<String, Location>();
64          int nextAvailable = maxErshov() + 1;
65          for (int i = 0; i < mParameterList.size(); ++i) {
66              String p = mParameterList.get(i);
67              if (nextAvailable < 30) {
68                  mSymbolTable.put(p, new Register(nextAvailable));
69                  ++nextAvailable;
70              } else {
71                  int offset = - mParameterList.size() + i;
72                  mSymbolTable.put(p, new Memory(offset));
73              }
74          }
75          int offset = 0;
76          for (int i = 0; i < mDeclarations.size(); ++i) {
77              String v = mDeclarations.get(i).getVar();
78              if (nextAvailable < 30) {
79                  mSymbolTable.put(v, new Register(nextAvailable));
80                  ++nextAvailable;
81              } else {
82                  ++offset;
83                  mSymbolTable.put(v, new Memory(offset));
84              }
85          }
86          mNumberVarsOnStack = offset;
87          mNumberUsedRegs    = nextAvailable - 1;
88      }
```

Abbildung 17.3: Die Klasse `Function`, 3. Teil.

Die in Abbildungen 17.3 gezeigte Methode *initTable*() berechnet die Symboltabelle, in der

hinterlegt ist, wo die einzelnen Variablen abgespeichert sind. Dazu wird zunächst die Funktion *maxErshov*() aufgerufen. Diese Funktion berechnet, wieviele Register maximal für Zwischenergebnisse benötigt werden. Das erste Register, über das wir frei verfügen können, hat dann den Wert

$$maxErshov() + 1.$$

Als nächstes ordnen wir in den Zeilen 67 bis 77 den Übergabe-Parametern der Funktion Register zu. Falls wir für einen Parameter kein freies Register mehr finden können, berücksichtigen wir in Zeile 73, dass die Parameter einer Funktion ja schon auf dem Stack unter der Rücksprungadresse liegen und tragen die entsprechende Adresse in der Symboltabelle ein.

In den Zeilen 78 bis 87 suchen wir anschließend für die lokalen Variablen einen Platz. Falls noch Register frei sind, ordnen wir den Variablen Register zu, sonst kommen die lokalen Variablen auf den Stack.

```
89      private int maxErshov() {
90          int result = 1;
91          for (Statement stmnt: mBody) {
92              result = Math.max(result, stmnt.maxErshov());
93          }
94          return result;
95      }
96      public String getName() {
97          return mName;
98      }
99      public List<String> getParameterList() {
100         return mParameterList;
101     }
102     public List<Statement> getBody() {
103         return mBody;
104     }
105     public String toString() {
106         return "Function(" + mName + ", " +
107                mParameterList + ", " + mBody + ")";
108     }
109 }
```

Abbildung 17.4: Die Klasse `Function`, 4. Teil.

Abbildung 17.4 zeigt schließlich die Implementierung der Funktion *maxErshov*(), bei der einfach das Maximum über alle Befehle des Rumpfs der Funktion gebildet wird.

Abbildung 17.5 zeigt exemplarisch, wie die Ershov-Zahlen für einzelne Ausdrücke berechnet werden und wie der Code für einen arithmetischen Ausdruck produziert wird. Die Klasse `Sum` repräsentiert einen Ausdruck der Form

$$mLhs + mRhs.$$

Die Ershov-Zahl für diesen Ausdruck wird bereits im Konstruktor ermittelt, wobei die Fallunterscheidung unmittelbar aus der im letzten Abschnitt gegebenen Diskussion folgt.

Die Methode *compile*() bekommt als erstes Argument eine natürliche Zahl *base* und als zweites Argument die Symboltabelle. Das Argument *base* gibt dabei den Index des ersten Registers an, das für Zwischen-Ergebnisse genutzt werden kann.

1. Betrachten wir zunächst den Fall, dass die Zahl der Register, die für die Berechnung von `mLhs` benötigt wird, kleiner ist, als die Zahl der für `mRhs` benötigten Register. In diesem Fall berechnen wir erst `mRhs`. Anschließend berechnen wir `mLhs`. Da wir dafür weniger Re-

```
1  import java.util.*;
2
3  public class Sum extends Expr {
4      private Expr mLhs;
5      private Expr mRhs;
6
7      public Sum(Expr lhs, Expr rhs) {
8          mLhs = lhs;
9          mRhs = rhs;
10         if (mLhs.getErshov() < mRhs.getErshov()) {
11             mErshov = mRhs.getErshov();
12         } else if (mLhs.getErshov() > mRhs.getErshov()) {
13             mErshov = mLhs.getErshov();
14         } else {
15             mErshov = mLhs.getErshov() + 1;
16         }
17     }
18     public List<AssemblerCmd> compile(Integer base,
19                                       Map<String, Location> symbolTable)
20     {
21         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
22         if (mLhs.getErshov() < mRhs.getErshov()) {
23             List<AssemblerCmd> rhsCmds = mRhs.compile(base, symbolTable);
24             List<AssemblerCmd> lhsCmds = mLhs.compile(base, symbolTable);
25             result.addAll(rhsCmds);
26             result.addAll(lhsCmds);
27         } else if (mLhs.getErshov() > mRhs.getErshov()) {
28             List<AssemblerCmd> lhsCmds = mLhs.compile(base, symbolTable);
29             List<AssemblerCmd> rhsCmds = mRhs.compile(base, symbolTable);
30             result.addAll(lhsCmds);
31             result.addAll(rhsCmds);
32         } else {
33             List<AssemblerCmd> lhsCmds = mLhs.compile(base + 1, symbolTable);
34             List<AssemblerCmd> rhsCmds = mRhs.compile(base    , symbolTable);
35             result.addAll(lhsCmds);
36             result.addAll(rhsCmds);
37         }
38         mRegister = base + mErshov;
39         if (mRegister < 30) {
40             result.add(new ADD(mRegister, mLhs.getReg(), mRhs.getReg()));
41         } else {
42             System.err.println("insufficient registers to compile:");
43         }
44         return result;
45     }
46     public Expr getLhs() { return mLhs; }
47     public Expr getRhs() { return mRhs; }
48 }
```

Abbildung 17.5: Die Klasse Sum.java

gister benötigen als für die Berechnung von `mRhs` werden wir das Ergebnis, das wir bei der Berechnung von `mRhs` im letzten Register gesichert haben, nicht überschreiben.

Wir speichern für jeden Ausdruck in der Member-Variablen `mRegister`, die in der abstrakten Klasse `Expr` definiert ist und die über die Funktion *getReg*() abgefragt werden kann, dasjenige Register, in dem das Ergebnis des Ausdrucks abgespeichert wird. Wir setzen diese Variable in Zeile 38 und können dann die Summe durch den Befehl

```
add mRegister, mLhs.getReg(), mRhs.getReg()
```

berechnen. Dieser Fall wird in den Zeilen 22 – 26 abgehandelt.

2. Der umgekehrte Fall, bei dem die Zahl der Register, die für die Berechnung von `mRhs` benötigt wird, kleiner ist, als die Zahl der für `mLhs` benötigten Register kann analog behandelt werden und ist in den Zeilen 27 – 31 gezeigt.

3. Abschließend betrachten wir den Fall, in dem die Zahl der Register, die für die Berechnung von `mLhs` benötigt wird, genauso groß ist wie die Zahl der für `mRhs` benötigten Register. In diesem Fall benutzen wir zur Berechnung von `mLhs` die Register

   $$base + 1,\ \cdots,\ base + 1 + mLhs.getErshov().$$

   Das Ergebnis dieser Rechnung ist dann in dem Register $base + 1 + mLhs.getErshov()$ gespeichert. Wenn wir nun bei der Berechnung von *mRhs* bereits mit dem Register *base* starten, dann ist sicher gestellt, dass dabei das in

   $$base + 1 + mLhs.getErshov()$$

   gespeicherte Ergebnis nicht überschrieben werden kann. Dieser Fall ist der Grund, warum wir bei der Methode *compile*() das zusätzliche Argument *base* benötigen. Der Fall ist in den Zeilen 33 bis 36 gezeigt.

4. Der Fall, dass die Anzahl der Register nicht ausreichend ist, um die Zwischen-Ergebnisse, die bei der Berechnung eines Ausdrucks auftreten können, zu berechnen, wird in Zeile 42 behandelt. Dieser Fall kann, wie in einer Übungsaufgabe gezeigt wird, bei 29 frei verfügbaren Registern de facto nicht auftreten.

Die in Abbildung 17.6 gezeigte Klasse `ReturnStmnt` zeigt die Übersetzung eines Befehls der Form

```
return expr;
```

In der Methode *compile*() wird zunächst der Ausdruck *expr* berechnet und das Ergebnis dieser Rechnung wird in Zeile 13 auf den Stack gelegt. Anschließend werden mit dem von der Methode *restoreRegisters*() erzeugten Code die Register wieder auf die Werte zurück gesetzt, die auf dem Stack gespeichert wurden.

Zum Abschluss diskutieren wir die in Abbildung 17.7 gezeigte Klasse `FunctionCall`, die einen Funktions-Aufruf der Form

$$f(e_1, \cdots, e_n)$$

repräsentiert. Der Konstruktor initialisiert sowohl den Namen als auch die Argumente der Funktion und berechnet anschließend die Ershov-Zahl als Maximum der Ershov-Zahlen der Argumente. Die Aufgabe der Methode *compile* ist es nun, Code zu erzeugen, der diese Argumente berechnet und auf dem Stack ablegt. In der Symboltabelle haben wir unter der Pseudo-Variablen "$free" gespeichert, wo der erste freie Platz auf dem Stack ist. dort legen wir in der Schleife in Zeile 20 – 23 die Argumente ab. Anschließend erhöhen wir in Zeile 26 den Stack-Pointer, so dass er nun auf das letzte auf dem Stack abgelegte Argument zeigt und rufen die Funktion mit einem `call`-Befehl auf. In Zeile 28 laden wir das Ergebnis vom Stack und speichern es in dem dafür vorgesehenen Register. Danach wird der Stack-Pointer wieder auf seinen ursprünglichen Wert zurück gesetzt.

```
1  public class ReturnStmnt extends Statement {
2      private Expr   mExpr;
3          private String mName;
4
5      public ReturnStmnt(Expr expr, String name) {
6          mExpr = expr;
7                  mName = name;
8      }
9      public List<AssemblerCmd> compile(Map<String, Location> symbolTable) {
10         int numVarsOnStack = ((Memory) symbolTable.get("$numberVarsOnStack@" + mName)).getOffset
11         int numUsedRegs    = ((Memory) symbolTable.get("$numberUsedRegs@"    + mName)).getOffset
12         List<AssemblerCmd> result = mExpr.compile(0, symbolTable);
13         result.add(new STOREC(mExpr.getReg(), 1));
14         result.addAll(restoreRegisters(numVarsOnStack, numUsedRegs));
15         result.add(new RETURN());
16         return result;
17     }
18     private List<AssemblerCmd> restoreRegisters(int numVarsOnStack, int numUsedRegs) {
19         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
20         for (int i = 1; i <= numUsedRegs; ++i) {
21             result.add(new LOADC(i, numVarsOnStack + i));
22         }
23         return result;
24     }
25     protected int maxErshov() {
26         return mExpr.getErshov();
27     }
28     public Expr getExpr() {
29         return mExpr;
30     }
31     public String toString() {
32         return "ReturnStmnt(" + mExpr.toString() + ")";
33     }
34 }
```

Abbildung 17.6: Die Klasse `ReturnStmnt`.

17.3 Schlussbetrachtung

Der im letzten Abschnitt dargestellte Compiler geht sehr sorglos mit Registern um. Insbesondere das Sichern und Wiederherstellen der Register auf dem Stack, das von diesem Compiler durchgeführt würde, ist für die Praxis viel zu aufwendig. Um einen halbwegs brauchbaren Code zu erzeugen, sind wenigstens die folgenden Schritte erforderlich:

1. Zunächst muss überprüft werden, welche Variablen an welcher Stelle des Programms *lebendig* sind. Dabei wird eine Variable an einer Stelle des Programms dann als *lebendig* bezeichnet, wenn der Wert, den diese Variable an der Stelle hat, später noch für eine Rechnung benötigt wird.

   Der Grund, warum eine Berechnung der lebendigen Variablen Sinn macht, ist der, dass ein Register, das von einer Variablen belegt wird, die an dieser Stelle des Programms nicht mehr lebendig ist, von einer anderen Variablen genutzt werden kann.

2. Anschließend geschieht nun die Register-Allokation, bei der nun auch verschiedenen Varia-

```
1  public class FunctionCall extends Expr {
2      private String     mName;
3      private List<Expr> mArgs;
4
5      public FunctionCall(String name, List<Expr> args) {
6          mName = name;
7          mArgs = args;
8          mErshov = 1;
9          for (Expr arg: mArgs) {
10             mErshov = Math.max(arg.getErshov(), mErshov);
11         }
12     }
13     public List<AssemblerCmd>
14         compile(Integer base, Map<String, Location> symbolTable)
15     {
16         Location l = symbolTable.get("$free");
17         Memory m = (Memory) l;
18         Integer k = m.getOffset();
19         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
20         for (Expr arg: mArgs) {
21             result.addAll(arg.compile(base, symbolTable));
22             result.add(new STOREC(arg.getReg(), k)); // store Rx, SP + k
23             ++k;
24         }
25         mRegister = 1;
26         result.add(new CONST(31, k + mArgs.size() - 1));
27         result.add(new ADD(30, 30, 31));   // SP = SP + k + n - 1
28         result.add(new CALL(mName));
29         result.add(new LOADC(mRegister, 1));
30         result.add(new CONST(31, - k - mArgs.size()));
31         result.add(new ADD(30, 30, 31));   // SP = SP -(k + n)
32         return result;
33     }
34     public String     getName() { return mName; }
35     public List<Expr> getArgs() { return mArgs; }
36 }
```

Abbildung 17.7: Die Klasse `FunctionCall`.

blen das selbe Register zugeordnet werden kann. Wird zwei verschiedenen Variablen x und y das selbe Register zugewiesen, so muss lediglich gewährleistet werden, dass es keinen Punkt im Programm gibt, bei dem gleichzeitig sowohl x als auch y lebendig sind.

3. Bei dem Sichern und Wiederherstellen der Register auf dem Stack muss ausgehend von der Analyse der Lebendigkeit der Variablen garantiert werden, dass nur solche Register gesichert werden, deren Werte tatsächlich später noch benötigt werden.

Aus Zeitgründen können wir die oben dargestellten Überlegungen leider nicht mehr weiter detailieren.

Kapitel 18

Lebendigkeits-Analyse

Bei komplexeren Funktionen ist es in der Regel nicht möglich, für jede in der Funktion auftretende Variable ein neues Register zu allokieren, denn dafür ist die Zahl der Register insbesondere bei einem CISC-Prozessor nicht ausreichend. Dies ist in der Regel aber auch nicht notwendig, denn zwei verschiedene Variablen können sich dann ein Register teilen, wenn diese Variablen nie gleichzeitig *lebendig* sind. Dabei nennen wir eine Variable an einer bestimmten Position lebendig, wenn auf den Wert dieser Variablen zu einem späteren Zeitpunkt noch zugegriffen werden kann. Betrachten wir zur Veranschaulichung die in Abbildung 18.1 gezeigte C-Funktion. In dieser Funktion werden die vie Variablen a, b, c und n verwendet. Es sieht zunächst so aus, als ob vier Register notwendig wären, um alle Variablen in Registern speichern zu können. Wir werden zeigen, dass drei Register bereits ausreichend sind.

```
1       int f(int n) {
2       int a, b, c;
3       a = 0;
4       c = 0;
5       do {
6           b = a + 1;
7           c = c + b;
8           a = b * 2;
9       } while (a < n);
10      return c;
11  }
```

Abbildung 18.1: Eine einfache C-Funktion.

Als erstes übersetzen wir das in Abbildung 18.1 gezeigte Programm in ein Assembler-ähnliches Programm. Dieses Programm ist in Abbildung 18.2 gezeigt. Wir untersuchen nun, wann die einzelnen Variablen *lebendig* sind. Ein erster Versuch könnte wie folgt aussehen:

1. a wird in Zeile 6 gelesen und ist daher in dieser Zeile lebendig. Da Zeile von Zeile 3 und 4 sowie auch von Zeile 8 und 9 erreicht werden kann, ist a auch in diesen Zeile lebendig. In Zeile 7 ist a nicht lebendig, denn der Wert, den a zu diesem Zeitpunkt hat, wird in Zeile 8 überschrieben. In Zeile 10 ist a ebenfalls nicht lebendig. Insgesamt haben wir

   $$\mathit{live}(\mathtt{a}) = \{3, 4, 6, 8, 9\}$$

   Bemerkung: Zeilen, die keine echten Anweisungen sondern nur Deklarationen enthalten, werden bei der berechnung der Lebendigkeit einer Variablen nicht berücksichtigt. In dem obigen Beispiel sind dies die Zeilen 2 und 5.

```
1  int f(int n) {
2      int a, b, c;
3      a = 0;
4      c = 0;
5    loop:
6      b = a + 1;
7      c = c + b;
8      a = b * 2;
9      if (a < n) goto loop;
10     return c;
11 }
```

Abbildung 18.2: Eine einfache C-Funktion.

2. b wird in Zeile 7 und 8 gelesen und ist daher dort lebendig. Der Wert von b wird in Zeile 6 berechnet. Wir könnten daher vermuten, dass

   $$\mathit{live}(\mathtt{b}) = \{6, 7, 8\}$$

   gilt. Wenn wir so rechnen würden, dann würden sich die Lebendigkeits-Bereiche der Variablen a und b überlappen. Tatsächlich können wir die Variable a aber in dem selben Register wie die Variable b speichern, denn a wird nur bei der Berechnung des Wertes b benötigt, aber sobald dieser Wert berechnet ist, ist der Wert von a irrelevant.

   Das Problem ist, dass die Frage, ob eine Variable lebendig ist, nicht an einer Programmzeile festgemacht werden kann, sondern an dem Übergang von einer Programmzeile zu einer anderen Programmzeile. Wir schreiben daher

   $$\mathit{live}(\mathtt{b}) = \{6 \to 7,\ 7 \to 8\}.$$

   Damit schreibt sich $\mathit{live}(\mathtt{a})$ als

   $$\mathit{live}(\mathtt{a}) = \{3 \to 4,\ 4 \to 6,\ 8 \to 9,\ 9 \to 6\}$$

   Wir stellen nun fest, dass die Mengen $\mathit{live}(\mathtt{a})$ und $\mathit{live}(\mathtt{b})$ disjunkt sind, so dass wir die beiden Variablen tatsächlich in dem selben Register speichern können.

3. c wird in Zeile 7 gelesen und auch geschrieben. Außerdem wird c auch in Zeile 10 gelesen. Die Definition dieser Variablen ist in Zeile 4. Daher haben wir

   $$\mathit{live}(\mathtt{c}) = \{4 \to 6,\ 6 \to 7,\ 7 \to 8,\ 8 \to 9,\ 9 \to 6,\ 9 \to 10\}$$

   Die Argumentation für diese Gleichung ist in Detail wie folgt:

   (a) Da c in Zeile 7 gelesen wird und der Befehl 7 auf den Befehl 6 folgt, gilt

   $$6 \to 7 \in \mathit{live}(\mathtt{c}).$$

   (b) Da c in Zeile 6 lebendig ist und auch nicht neu geschrieben wird, ist c auch in den Zeilen, die unmittelbar vor Zeile 6 kommen können, lebendig. Damit haben wir

   $$9 \to 6 \in \mathit{live}(\mathtt{c}) \quad \text{und} \quad 4 \to 6 \in \mathit{live}(\mathtt{c})$$

   (c) Da c in Zeile 9 lebendig ist und auch nicht neu geschrieben wird und da weiterhin Zeile 9 auf Zeile 8 folgt, ist c auch in Zeile 8 lebendig. Damit haben wir

   $$8 \to 9 \in \mathit{live}(\mathtt{c}).$$

(d) Genauso sehen wir

$$7 \to 8 \in \mathit{live}(\mathtt{c}).$$

4. `n` wird in Zeile 9 gelesen und in Zeile 1 definiert gelesen. Damit finden wir

$$\mathit{live}(\mathtt{c}) = \{1 \to 3,\ 3 \to 4,\ 4 \to 6,\ 6 \to 7,\ 7 \to 8,\ 8 \to 9,\ 9 \to 6\}$$

Nur in Zeile 10 ist also `n` nicht mehr lebendig.

Wir formalisieren nun die in dem obigen Beispiel durchführte Argumentation. Wir fassen dazu das Assembler-ähnliche Programm als einen Graphen auf. Die Zeilen des Programms, die später in einen Assembler-Befehl übersetzt werden, sind die Knoten des Programms. Zwischen zwei Zeilen a und b gibt es genau dann eine Kante von a nach b, wenn bei der Abarbeitung des Programms die Instruktion b auf die Instruktion a unmittelbar folgen kann. Den so erstellten Graph bezeichnen wir als den *Kontrollfluss-Graphen*. In dem obigen Beispiel besteht der Kontrollfluss-Graph aus den folgenden Kanten:

$$\{1 \to 3,\ 3 \to 4,\ 4 \to 6,\ 6 \to 7,\ 7 \to 8,\ 8 \to 9,\ 9 \to 6,\ 9 \to 10\}.$$

Für eine Programm-Zeile n bezeichnen wir mit $\mathit{pred}(n)$ die Menge aller Knoten, von denen eine Kante zum Knoten n hin geht. Der Funktionsname *pred* steht hier für *predecessor*. Weiter bezeichnen wir mit $\mathit{succ}(n)$ die Menge aller Knoten, zu denen eine Kante von n aus hin führt. Der Name *succ* steht für *successor*. In dem obigen Beispiel gilt

$$\mathit{pred}(6) = \{4, 9\} \quad \text{und} \quad \mathit{succ}(9) = \{6, 10\}.$$

Weiter bezeichnen wir für eine Variable x mit $\mathit{def}(x)$ die Menge aller Knoten, in denen der Variablen ein Wert zugewiesen wird. Beispielsweise gilt

$$\mathit{def}(\mathtt{a}) = \{3, 8\} \quad \text{und} \quad \mathit{def}(\mathtt{b}) = \{6\}.$$

Schließlich bezeichnen wir für eine Variablen x mit $\mathit{use}(x)$ die Menge aller Knoten im Kontrollfluss-Graphen, in denen die Variable x gelesen wird. Beispielsweise gilt

$$\mathit{use}(\mathtt{a}) = \{6, 9\} \quad \text{und} \quad \mathit{def}(\mathtt{b}) = \{7, 8\}.$$

Damit kommen wir nun zur zentralen Definition der Lebendigkeit einer Variablen.

Definition 48 (lebendig) Eine Variable x ist lebendig auf einer Kante $k_1 \to k_2$ im Kontrollfluss-Graphen, falls die Kante zu einem Pfad $[k_1, k_2, \cdots, k_n]$ fortgesetzt werden kann, so dass gilt:

1. $k_n \in \mathit{use}(x)$,
   die Variable x wird also am Endpunk des Pfads benutzt.
2. $\{k_2, k_2, \cdots, k_{n-1}\} \cap \mathit{def}(x) = \{\}$,
   die Variable wird also in keinem der Zwischenpunkte des Pfads neu definiert.

Beispiel: Die Variable `c` ist auf der Kante $7 \to 8$ lebendig, denn diese Kante kann zu dem Pfad $[7, 8, 9, 6, 7]$ fortgesetzt werden und auf den Punkten der Menge $\{8, 9, 6\}$ wird `c` nicht neu definiert.

Definition 49 (*liveIn*, *liveOut*) Für eine Variable x und einen Knoten n sagen wir dass $x \in \mathit{liveIn}(n)$ ist, falls es einen Knoten m gibt, so dass x auf der Kante $m \to n$ lebendig ist. Es gilt $x \in \mathit{liveOut}(n)$ wenn es einen Knoten k gibt, so dass x auf der Kante $n \to k$ lebendig ist.

Bemerkung: Anschaulich gilt $x \in \mathit{liveIn}(n)$, falls die Variable in dem Knoten n oder in einem Knoten, der im Kontrollfluss-Graphen auf n folgt, gelesen wird, ohne vorher überschrieben zu werden. Die Bedingung $x \in \mathit{liveOut}(n)$ bedeutet, dass der Wert von x in dem Knoten n (oder einem seiner Vorgänger-Knoten) definiert wird und später noch gelesen wird, ohne vorher überschrieben zu werden.

Die Funktionen $\mathit{liveIn}()$ und $\mathit{liveOut}()$ genügen damit den folgenden beiden Gleichungen:

1. $\mathit{liveIn}(n) = \mathit{use}(n) \cup \big(\mathit{liveOut}(n) \setminus \mathit{def}(n)\big)$,

   in dem Knoten n sind also genau die Variablen *live-in*, die entweder in dem Knoten n gelesen werden, oder aber bereits *live-out* sind und nicht in dem Knoten n neu geschrieben werden.

2. $\mathit{liveOut}(n) = \bigcup_{k \in \mathit{succ}(n)} \mathit{liveIn}(k)$,

   in dem Knoten n sind genau die Variablen *live-out*, die in einem der Nachfolgerknoten *live-in* sind.

Über diese Gleichungen können nun die Funktionen *liveIn*() und *liveOut*() iterativ berechnet werden. Abbildung 18.3 zeigt den Pseudo-Code. Zunächst werden *liveIn*(n) und *liveOut*(n) für jeden Knoten n mit der leeren Menge initialsiert. Die `while`-Schleife in Zeile 5 wird solange iteriert, bis sich die Mengen *liveIn*(n) und *liveOut*(n) nicht mehr ändern.

```
1  for (k in Knoten) {
2      liveIn(k)  := {};
3      liveOut(k) := {};
4  }
5  while (change) {
6      for (n in Knoten) {
7          liveIn(n)  := use(n) ∪ (liveOut(n) - def(n));
8          liveOut(n) := ⋃ { liveIn(k) | k ∈ succ(n) };
9  }
```

Abbildung 18.3: Pseudo-Code zur Berechnung von *liveIn*(n) und *liveOut*(n)

Die Berechnung der Funktion *live*(x) für eine Kante $k \to n$ ergibt sich nun nach der folgenden Formel

$$(k \to n) \in \mathit{live}(x) \quad \text{g.d.w.} \quad n \in \mathit{liveIn}(x) \wedge k \in \mathit{pred}(n).$$

Die Variable x ist also auf der Kante $k \to n$ genau dann lebendig, wenn x eine Element der Menge *liveIn*(n) ist und wenn außerdem die Kante $k \to n$ eine Kante des Kontrollfluss-Graphen ist.

Ist die Funktion *live*(x) für alle Variablen berechnet, so definieren wir, dass zwei Variablen x und y *im Konflikt* miteinander stehen, wenn

$$\mathit{live}(x) \cap \mathit{live}(y) \neq \{\}.$$

gilt. In diesem Fall benötigen wir verschiedene Register für die Variablen x und y. Ist die Schnittmenge *live*(x) $\cap$ *live*(y) hingegen leer, so können wir die Variablen x und y im selben Register speichern.

Literaturverzeichnis

[ASUL06] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling: Volume I: Parsing*. Prentice-Hall, 1972.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959.

[BHPS61] Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonologie, Sprachwissenschaft und Kommunikationsforschung*, 14:113–124, 1961.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.

[Ear68] Jay Clark Earley. *An efficient context-free parsing algorithm*. PhD thesis, Pittsburgh, PA, USA, 1968.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[Ers58] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

[Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, 2nd edition, 2002.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.

[HFA+99] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew W. Appel. CUP - LALR parser generator for Java, 1999. Available at `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 1979.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, pages 3–40. Princeton University Press, 1956.

[Kle05] Gerwin Klein. JFlex User's Manual. Technical report, 2005. Available at: `http://jflex.de/jflex.pdf`.

[Knu65] Donald E. Knuth. On the translation of languages from left to rigth. *Information and Control*, 8(6):607–639, December 1965.

[Lut09] Mark Lutz. *Learning Python*. O'Reilly, 4th edition, 2009.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[NBB+60] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. MacCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language Algol 60. *Numerische Mathematik*, 2:106–136, 1960.

[Ner58] A. Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.

[Par07] Terence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[Str07] Karl Stroetmann. *Computer-Architektur: Modellierung, Entwicklung und Verifikation mit Verilog*. Oldenbourg-Verlag, 2007.

[Tan05] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Upper Saddle River, NJ, 5th edition, 2005.

[WS92] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Assoc., 1992.