

Theoretische Informatik III & Compilerbau

— WS 2012 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

31. Dezember 2012

Inhaltsverzeichnis

1	Einführung und Motivation	4
1.1	Grundlegende Definitionen	4
1.2	Literatur	7
1.2.1	Verwendete Programmiersprachen	7
2	Reguläre Ausdrücke	8
2.1	Einige Definitionen	8
2.2	Algebraische Vereinfachung regulärer Ausdrücke	11
3	Der Scanner-Generator <i>JFlex</i>	15
3.1	Struktur einer <i>JFlex</i> -Spezifikation	15
3.2	Reguläre Ausdrücke in <i>JFlex</i>	18
3.3	Weitere Optionen	22
3.4	Ein komplexeres Beispiel: Noten-Berechnung	23
3.4.1	Zustände	26
4	Endliche Automaten	30
4.1	Deterministische endliche Automaten	30
4.2	Nicht-deterministische endliche Automaten	35
4.3	Äquivalenz von EA und NEA	38
4.3.1	Implementing the Conversion of NFA to DFA	43
4.4	Übersetzung regulärer Ausdrücke in NEA	44
4.4.1	Implementing the Conversion of Regular Expressions into Finite State Machines	47
4.5	Übersetzung eines EA in einen regulären Ausdruck	53
4.5.1	Implementing the Conversion of FSMs into Regular Expressions	57
5	Minimierung endlicher Automaten	60
5.1	Implementing the Minimization of Finite Automata in SETLX	63
5.2	The Theorem of Nerode	65
6	Die Theorie regulärer Sprachen	68
6.1	Abschluss-Eigenschaften regulärer Sprachen	68
6.2	Erkennung leerer Sprachen	70
6.3	Äquivalenz regulärer Ausdrücke	71
6.4	Implementation in SETLX	72
6.5	Grenzen regulärer Sprachen	73
7	Kontextfreie Sprachen	77
7.1	Kontextfreie Grammatiken	77
7.1.1	Ableitungen	79
7.1.2	Parse-Bäume	82
7.1.3	Mehrdeutige Grammatiken	82
7.2	Top-Down-Parser	86

7.2.1	Umschreiben der Grammatik	86
7.2.2	Implementing a Top Down Parser in SETLX	91
7.2.3	Implementing a Recursive Decent Parser that Works Backwards	95
7.2.4	Implementing a Recursive Decent Parser that Uses an EBNF Grammar	97
8	Antlr	101
8.1	Ein Parser für arithmetische Ausdrücke	101
8.2	Ein Parser zur Auswertung arithmetischer Ausdrücke	104
8.3	Erzeugung abstrakter Syntax-Bäume	107
8.3.1	Implementierung des Parsers	108
9	LL(k)-Sprachen	111
9.1	Links-Faktorisierung	112
9.2	<i>First</i> und <i>Follow</i>	115
9.3	LL(1)-Grammatiken	119
9.4	Implementierung	120
9.4.1	Die Berechnung der ε -erzeugenden Variablen	121
9.4.2	Die Berechnung der Funktion <i>First</i> ()	122
9.4.3	Die Berechnung der Funktion <i>Follow</i> ()	123
9.4.4	Berechnung der Parse-Tabelle	124
9.5	LL(k)-Grammatiken	126
9.5.1	Berechnung von <i>First</i> () und <i>Follow</i> ()	128
9.5.2	Computing the Parse Table	130
9.6	Implementing the Computation of LL(k) Parse Tables	130
10	Behandlung von Nicht-LL(k)-Sprachen in ANTLR	135
10.1	Unterstützung von LL(*)-Sprachen	135
10.1.1	Motivation	135
10.1.2	Die Theorie der LL(*)-Sprachen	139
10.2	Semantische Prädikate	141
10.3	Syntaktische Prädikate und Backtracking	145
10.3.1	Das Dangling-Else-Problem	145
11	Interpreter	149
12	Grenzen kontextfreier Sprachen	160
12.1	Beseitigung nutzloser Symbole	160
12.2	Parse-Bäume als Listen	162
12.3	Das Pumping-Lemma für kontextfreie Sprachen	165
12.4	Anwendungen des Pumping-Lemmas	166
12.4.1	Die Sprache $L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$ ist nicht kontextfrei	166
13	Earley-Parser	168
13.1	Der Algorithmus von Earley	168
13.2	Implementierung	173
13.3	Korrektheit und Vollständigkeit	180
13.4	Analyse der Komplexität	185
14	LR-Parser	187
14.1	Bottom-Up-Parser	187
14.2	Shift-Reduce-Parser	189
14.3	SLR-Parser	196
14.3.1	Shift-Reduce- und Reduce-Reduce-Konflikte	203
14.4	Kanonische LR-Parser	205
14.5	LALR-Parser	209

14.6	Vergleich von SLR-, LR- und LALR-Parsern	211
14.6.1	$SLR\text{-Sprache} \subsetneq LALR\text{-Sprache}$	212
14.6.2	$LALR\text{-Sprache} \subsetneq \text{kanonische LR-Sprache}$	212
14.6.3	Bewertung der verschiedenen Methoden	214
15	Der Parser-Generator <i>JavaCup</i>	215
15.0.4	Generierung eines CUP-Scanner mit Hilfe von <i>Flex</i>	219
15.1	Shift-Reduce und Reduce-Reduce-Konflikte	221
15.2	Operator-Präzedenzen	222
15.2.1	Das <i>Dangling-Else</i> -Problem	228
15.3	Auflösung von Reduce-Reduce-Konflikte	231
15.3.1	Mehrdeutigkeits-Konflikte	231
15.3.2	Look-Ahead-Konflikte	232
15.3.3	Mysteriöse Reduce-Reduce-Konflikte	233
16	Types and Type Checking	237
16.1	Eine Beispielsprache	238
16.2	Grundlegende Begriffe	240
16.3	A Type Checking Algorithm	243
16.4	Implementierung eines Typ-Checkers für TTL	246
16.5	Inklusions-Polymorphismus	254
17	Assembler	256
17.1	Einführung in JVM-Assembler	256
17.2	Die Instruktionen der JVM	257
18	Entwicklung eines einfachen Compilers	269
18.1	Die Programmiersprache <i>Integer-C</i>	269
18.2	Entwicklung von Scanner und Parsers	271
18.3	Darstellung der Assembler-Befehle	278
18.4	Die Code-Erzeugung	279
18.4.1	Übersetzung arithmetischer Ausdrücke	280
18.4.2	Übersetzung von Boole'schen Ausdrücken	284
18.4.3	Übersetzung der Befehle	289
18.4.4	Zusammenspiel der Komponenten	293

Kapitel 1

Einführung und Motivation

Die theoretische Informatik, mit der wir uns im Rahmen dieser Vorlesung beschäftigen werden, hat eine sehr wichtige Anwendung, den *Compilerbau*: Zwischen der Maschinen-Sprache, die von dem Prozessor eines Computers unmittelbar abgearbeitet werden kann, und einer höheren Programmiersprache wie *Java* besteht ein großer Unterschied: Während die Maschinsprache nur einige einfache Instruktionen umfasst, mit deren Hilfe elementare Operationen wie die Addition zweier Zahlen oder das Verschieben einer Zahl aus einem Register in den Speicher bewirkt werden können, enthält eine Programmiersprache wie *Java* Befehle, die beliebig abstrakt sein können. Ziel der vorliegenden Vorlesung ist aufzuzeigen, wie ein Programm, das in einer höheren Programmiersprache entwickelt worden ist, in ein Maschinen-Programm übersetzt werden kann. Im Modulplan ist das Modul *Theoretische Informatik III* mittlerweile in vier Units aufgeteilt:

1. Die Unit “*Automaten I*” beschäftigt sich mit *regulären Sprachen* und der Umsetzung dieser Sprachen mit Hilfe endlicher Automaten.
2. In der Unit “*Automaten II*” stehen die *kontext-freien Sprachen* im Vordergrund. Die Syntax der meisten Programmiersprachen lässt sich durch kontext-freie Sprachen darstellen.
3. Die Unit “*Compilerbau*” zeigt, wie ein gegebenes Programm aus einer Hochsprache in die Maschinsprache eines Rechners übersetzt werden kann.
4. In der Unit “*Compilerbau-Werkzeuge*” lernen wir verschiedene Werkzeuge kennen, mit deren Hilfe sich *Scanner* und *Parser* entwickeln lassen.

In meiner Vorlesung werde ich die Inhalte der vier Units vermischen. Hintergrund ist, dass ich die Werkzeuge *JFlex* und *JavaCup*, mit denen Scanner und Parser generiert werden können, möglichst früh einführen möchte. Dadurch erhalten Sie als Studierende schon sehr früh einen Eindruck vom praktischen Nutzen der theoretischen Informatik.

1.1 Grundlegende Definitionen

Der zentrale Begriff dieser Vorlesung ist der Begriff der *formalen Sprache*. 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*:

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

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. $\Sigma = \{\mathbf{a}, \dots, \mathbf{z}, \mathbf{A}, \dots, \mathbf{Z}\}$ ist das Alphabet, das der englischen Sprache zu Grunde liegt.
3. Die Menge $\Sigma_{\text{ASCII}} = \{0, 1, \dots, 127\}$ 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 $c_1, \dots, c_n \in \Sigma$, so schreiben wir also

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

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. \square

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 $\Sigma = \{\mathbf{a}, \dots, \mathbf{z}\}$. Setzen wir

$$w := \text{beispiel},$$

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

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 . Wir fangen bei der Nummerierung der Buchstaben mit 1 an, $w[1]$ ist also der erste Buchstabe 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 $w_1 + w_2$ oder gelegentlich noch kürzer als $w_1 w_2$.

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

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

Nun können wir bereits den ersten zentralen Begriff einführen:

Definition 3 (Formale Sprache)

Ist Σ ein Alphabet, so bezeichnen wir eine Teilmenge $L \subseteq \Sigma^*$ als eine *formale Sprache*. \square

Im weiteren Verlauf der Vorlesung wird uns die Frage beschäftigen, wie wir formale Sprachen so beschreiben können, dass wir diese Beschreibungen auf dem Rechner verarbeiten können.

Beispiele:

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

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

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

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

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

$$\text{value} : L_{\mathbb{N}} \rightarrow \mathbb{N}$$

durch Induktion nach der Länge des Wortes. Diese Abbildung bezeichnen wir auch als eine *Interpretation* der Worte der Sprache $L_{\mathbb{N}}$. Wir definieren $\text{value}(w)$ als die Zahl, die durch das Wort w dargestellt wird:

- (a) $\text{value}(0) = 0, \text{value}(1) = 1,$
- (b) $|w| > 0 \rightarrow \text{value}(w + 0) = 2 \cdot \text{value}(w),$
- (c) $|w| > 0 \rightarrow \text{value}(w + 1) = 2 \cdot \text{value}(w) + 1.$

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:

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

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:

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

3. Es sei $\Sigma_{\text{ASCII}} = \{0, \dots, 127\}$ 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) $x, y \in \Sigma_{\text{ASCII}}^*,$
- (c) die Anwendung der durch p dargestellten Funktion 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, aber immer noch möglich. Es gibt keinen Algorithmus, 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. Allerdings umfasst dieses Buch wesentlich mehr Theorie als wir im Rahmen dieser Vorlesung besprechen können.

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

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

1.2.1 Verwendete Programmiersprachen

Die in der Vorlesung vorgestellten Werkzeuge basieren auf der Ihnen bekannten Sprache *Java*. Gelegentlich werde ich Ihnen auch Algorithmen vorstellen, deren Umsetzung in *Java* recht mühsam ist. Diese Algorithmen werde ich daher in der Sprache SETLX vorstellen. Hierbei handelt es sich um eine Mengen-basierte Programmiersprache, die sich hervorragend zum *Prototyping* eignet, weil sich komplexe Algorithmen sehr kompakt in SETLX darstellen lassen. Ich erwarte nicht von Ihnen, dass Sie selber in SETLX programmieren können. Sie können die SETLX-Programme daher einfach als Pseudo-Code ansehen. Im Gegensatz zu Algorithmen, die nur als Pseudo-Code angegeben sind, haben die SETLX-Programme aber den Vorteil, dass Sie diese Programme ausprobieren können. Sie finden den dazu notwendigen Interpreter im Netz auf der folgenden Seite:

<http://wwwlehre.dhbw-stuttgart.de/~stroetma/SetlX/setlX.php>

Zu der Sprache SETLX gibt es ein Tutorial, das Sie sich unter der Adresse

<http://wwwlehre.dhbw-stuttgart.de/~stroetma/SetlX/tutorial.pdf>

aus dem Netz laden können.

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. Viele moderne Skript-Sprachen (insbesondere die Sprache *Perl*) wären ohne reguläre Ausdrücke undenkbar. Alle modernen Hoch-Sprachen (z.B. *Java*, *C#*, \dots) enthalten umfangreiche Bibliotheken zur Unterstützung regulärer Ausdrücke. Darüber hinaus gibt es eine Reihe von UNIX-Werkzeugen wie **grep** oder **sed**, die auf der Verwendung regulärer Ausdrücke basieren.

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:

$$L_1 \cdot L_2 := \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\} \quad \square$$

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

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

definiert. Dann gilt

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

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

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

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 $n \in \mathbb{N}$:

$$L^* := \bigcup_{n \in \mathbb{N}} L^n. \quad \square$$

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

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

Hierbei bezeichnet a^n das Wort der Länge n , das nur den Buchstaben a enthält, es gilt also

$$a^n = \underbrace{a \cdots a}_n. \quad \diamond$$

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

$$\begin{aligned} \text{I.A. } n = 0: \quad s^0 &:= \varepsilon. \\ \text{I.S. } n \mapsto n + 1: \quad s^{n+1} &:= s^n s, \quad \text{wobei } s^n s \text{ für die Konkatenation der Strings } s^n \text{ und } s \text{ steht.} \end{aligned}$$

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 RegExp_Σ . Die Definition dieser Menge verläuft induktiv. Gleichzeitig mit der Menge RegExp_Σ definieren wir eine Funktion

$$L : \text{RegExp}_\Sigma \rightarrow 2^{\Sigma^*},$$

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

Definition 7 (reguläre Ausdrücke) Die Menge RegExp_Σ der *regulären Ausdrücke* über dem Alphabet Σ wird wie folgt induktiv definiert:

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

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. $\varepsilon \in \text{RegExp}_\Sigma$

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

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

Beachten Sie, dass 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. $c \in \Sigma \rightarrow c \in \text{RegExp}_\Sigma$.

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

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

Aus den regulären Ausdrücken r_1 und r_2 kann also mit dem Infix-Operator “.” 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 “.” nicht in dem Alphabet Σ auftritt, es gilt also “.” $\notin \Sigma$.

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

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:

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

Wir nehmen “*” $\notin \Sigma$ an.

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

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

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

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

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

1. Der Postfix-Operator “*” bindet am stärksten.
2. Der Infix-Operator “.” bindet schwächer als “*” und stärker als “+”.
3. Der Infix-Operator “+” bindet am schwächsten.

Damit wird also der reguläre Ausdruck

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

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

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

festgelegt.

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

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

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

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

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

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

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

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 $(b + c)^*$) gefolgt von dem Buchstaben a , wiederum gefolgt von einer beliebigen Anzahl der Buchstaben b und c .

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

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

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

$$L(r_4) = \left\{ w \in \Sigma^* \mid \#\{i \in \mathbb{N} \mid w[i] = a\} = 1 \right\} \cup \left\{ w \in \Sigma^* \mid \#\{i \in \mathbb{N} \mid w[i] = b\} = 1 \right\}. \quad \diamond$$

Bemerkung: Die oben festgelegte Syntax regulärer Ausdrücke ist die Syntax, die in theoretischen Abhandlungen über formale Sprachen gebräuchlich ist, siehe beispielsweise [HMU06]. Sie weicht allerdings von der Syntax ab, die in Sprachen wie *Perl* oder *Java* gebräuchlich ist. Wir werden diese Unterschiede später noch diskutieren.

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? \diamond

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

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

$$4. \emptyset \cdot r \doteq r \cdot \emptyset \doteq \emptyset$$

$$5. \varepsilon \cdot r \doteq r \cdot \varepsilon \doteq r$$

$$6. \emptyset + r \doteq r + \emptyset \doteq r$$

$$7. (r_1 + r_2) \cdot r_3 \doteq r_1 \cdot r_3 + r_2 \cdot r_3$$

$$8. r_1 \cdot (r_2 + r_3) \doteq r_1 \cdot r_2 + r_1 \cdot r_3$$

$$9. r + r \doteq r, \text{ denn}$$

$$L(r + r) = L(r) \cup L(r) = L(r).$$

$$10. (r^*)^* \doteq r^*$$

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

$$L(r^*) \subseteq L((r^*)^*)$$

gilt. Um die Umkehrung

$$L((r^*)^*) \subseteq L(r^*)$$

zu beweisen, betrachten wir zunächst die Worte $w \in L((r^*)^*)$. Wegen

$$L((r^*)^*) = \bigcup_{n \in \mathbb{N}} L(r^*)^n$$

gilt $w \in L((r^*)^*)$ genau dann, wenn es ein $n \in \mathbb{N}$ gibt, so dass es Wörter $u_1, \dots, u_n \in L(r^*)$ gibt, so dass

$$w = u_1 \cdots u_n.$$

Wegen $u_i \in L(r^*)$ gibt es für jedes $i \in \{1, \dots, n\}$ eine Zahl $m(i) \in \mathbb{N}$, so dass für $j = 1, \dots, m(i)$ 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

$$w \in L(r^*)$$

und damit ist die Inklusion $L((r^*)^*) \subseteq L(r^*)$ 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 angegebenen Gleichungen vollständig ist. Diese Regel lautet

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

Wir werden diese Regel im Folgenden als *Salomaa-Regel* bezeichnen. Die Korrektheit der Salomaa-Regel ist der Inhalt des folgenden Lemmas.

Lemma 8 Es seien r , s und t reguläre Ausdrücke, für welche die Gleichung

$$r \doteq r \cdot s + t$$

gilt. Außerdem gelte $\varepsilon \notin L(s)$. Dann gilt

$$r \doteq t \cdot s^*.$$

Beweis: Nach Voraussetzung wissen wir, dass

$$L(r) = L(r) \cdot L(s) + L(t) \tag{2.1}$$

gilt. Wir müssen die Gleichung

$$L(r) = L(t) \cdot L(s^*) \tag{2.2}$$

nachweisen. Wir spalten diesen Nachweis in zwei Teile auf und zeigen zunächst, dass

$$L(r) \subseteq L(t) \cdot L(s^*) \tag{2.3}$$

gilt. Es sei also $x \in L(r)$ gegeben. Wir zeigen durch Induktion über die Länge von x , dass daraus $x \in L(t) \cdot L(s^*)$ folgt. Nach der Voraussetzung (2.1) gibt es zwei Möglichkeiten:

$$x \in L(r) \cdot L(s) \quad \text{oder} \quad x \in L(t).$$

Wir betrachten die Fälle getrennt und beginnen mit dem einfacheren Fall.

1. $x \in L(t)$.

Wegen $\varepsilon \in L(s^*)$ folgt dann sofort

$$x = x\varepsilon \in L(t) \cdot L(s^*).$$

2. $x \in L(r) \cdot L(s)$.

Dann läßt sich x in zwei Teilstrings y und z aufspalten, so dass gilt

$$x = yz, \quad y \in L(r) \quad \text{und} \quad z \in L(s).$$

Da $\varepsilon \notin L(s)$ vorausgesetzt ist, folgt $|z| > 0$ und damit gilt $|y| < |x|$, so dass wir auf y die Induktions-Voraussetzung anwenden können und damit

$$y \in L(t) \cdot L(s^*)$$

schließen können. Für x heißt dies

$$x = yz \in L(t) \cdot L(s^*) \cdot L(s) \subseteq L(t) \cdot L(s^*)$$

und damit ist der Beweis von $L(r) \subseteq L(t) \cdot L(s^*)$ abgeschlossen.

Als nächstes zeigen wir, dass

$$L(t \cdot s^*) \subseteq L(r) \tag{2.4}$$

gilt. Es sei also ein String $x \in L(t \cdot s^*)$ gegeben und wir müssen nachweisen, dass $x \in L(r)$ liegt. Wenn $x \in L(t \cdot s^*)$ ist, dann muss x die Form

$$x = yz_1 \cdots z_m \quad \text{mit } y \in L(t) \text{ und } z_i \in L(s) \text{ für alle } i = 1, \dots, m$$

haben. Wir zeigen durch Induktion über $m \in \mathbb{N}$, dass

$$x = yz_1 \cdots z_m \in L(r) \quad \text{für alle } m \in \mathbb{N} \text{ gilt.}$$

I.A.: $m = 0$.

Dann gilt $x = y \in L(t)$ und aus der Voraussetzung (2.1) folgt $y \in L(r)$.

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

Nach Induktions-Voraussetzung gilt bereits

$$yz_1 \cdots z_m \in L(r).$$

Nach Gleichung (2.1) haben wir dann

$$yz_1 \cdots z_m z_{m+1} \in L(r) \cdot L(s) \subseteq L(r)$$

und damit ist die Induktion abgeschlossen.

Damit ist die Gleichung $L(r) = L(t) \cdot L(s^*)$ nun vollständig bewiesen. \square

Bemerkung: Der Beweis der Tatsache, dass die oben angegebenen Gleichungen zusammen mit der Salomaa-Regel ausreichen, um jede gültige Gleichung zweier regulären Ausdrücke nachzuweisen, geht über den Rahmen der Vorlesung hinaus. Der Beweis findet sich in einem Papier von Arto Salomaa [Sal66].

Aufgabe 2: Zeigen Sie die folgende Gleichung mit Hilfe algebraischer Umformungen und der Salomaa-Regel:

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

Lösung: Wir weisen die Gleichung mit Hilfe der Salomaa-Regel

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

nach. Wir definieren zunächst

$$t := 1 \cdot 0, \quad s := 1 \cdot 0 \quad \text{und} \quad r := 1 \cdot (0 \cdot 1)^* \cdot 0.$$

Als nächstes zeigen wir, dass $r \doteq r \cdot s + t$ gilt. Einsetzen der oben definierten Werte liefert

$$1 \cdot (0 \cdot 1)^* \cdot 0 \stackrel{!}{=} 1 \cdot (0 \cdot 1)^* \cdot 0 \cdot (1 \cdot 0) + 1 \cdot 0. \quad (\star)$$

Um (\star) nachzuweisen, formen wir die rechte Seite dieser Gleichung wie folgt um:

$$\begin{aligned} & 1 \cdot (0 \cdot 1)^* \cdot 0 \cdot (1 \cdot 0) + 1 \cdot 0 \\ \doteq & 1 \cdot (0 \cdot 1)^* \cdot (0 \cdot 1) \cdot 0 + 1 \cdot 0 \quad (\text{Assoziativ-Gesetz}) \\ \doteq & 1 \cdot ((0 \cdot 1)^* \cdot (0 \cdot 1) + \varepsilon) \cdot 0 \quad (\text{Distributiv-Gesetz}) \\ \doteq & 1 \cdot (0 \cdot 1)^* \cdot 0 \quad (\text{Regel Nummer 13}) \end{aligned}$$

Das ist aber genau die linke Seite von Gleichung (\star) , so dass wir damit (\star) bewiesen haben. Weiterhin gilt $\varepsilon \notin L(1 \cdot 0)$. Damit sind die Voraussetzungen der Salomaa-Regel erfüllt und wir können schließen, dass die Gleichung

$$r \doteq t \cdot s^*$$

gültig ist. Einsetzen der Werte von r , s und t liefert dann die Gleichung

$$1 \cdot (0 \cdot 1)^* \cdot 0 \doteq (1 \cdot 0) \cdot (1 \cdot 0)^*$$

und das ist die Behauptung. \square

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* [Kle09] 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 *Lex* [Les75] bzw. *Flex* [Nic93].

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 16 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 8 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 16 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

`regex " {" action "}"`

Hierbei ist `regex` ein regulärer Ausdruck und `action` ist ein Fragment von Java-Code, das ausgeführt wird, wenn der Ausdruck `regex` 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:

$$\text{OpSyms} := \{ \text{"."}, \text{"*"}, \text{"+"}, \text{"?"}, \text{"!"}, \text{"~"}, \text{"|"}, \text{"("}, \text{")"}, \text{"["}, \text{"]"}, \text{"{"}, \text{"}"}, \text{"<"}, \text{">"}, \text{"/"}, \text{"\"}, \text{"^"}, \text{"\$"}, \text{"\""} \}$$

Zusätzlich bezeichne *WsSym* die Menge der ASCII-Symbole, die White-Space darstellen. Diese Menge enthält also Blanks, Zeilenumbrüche und Tabulatoren. Damit können wir nun die Menge *Regex* der von JFlex unterstützen regulären Ausdrücke induktiv definieren.

1. $c \in \text{Regex}$ falls $c \in \Sigma_{\text{ASCII}} \setminus (\text{OpSyms} \cup \text{WsSyms})$

Alle Buchstaben c aus dem ASCII-Alphabet, die keine Operator-Symbol und keine White-Space-Symbole sind, können als reguläre Ausdrücke verwendet werden. Der reguläre Ausdruck c spezifiziert genau den Buchstaben c .

2. $\text{"."} \in \text{Regex}$

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

3. $\backslash x \in \text{Regex}$ falls $x \in \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{n}, \mathbf{r}, \mathbf{t}, \mathbf{v}\}$

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

- (a) $\backslash \mathbf{a}$ entspricht dem Steuerzeichen **Ctrl-G** (*alert*).
- (b) $\backslash \mathbf{b}$ entspricht dem Steuerzeichen **Ctrl-H** (*backspace*).
- (c) $\backslash \mathbf{f}$ entspricht dem Steuerzeichen **Ctrl-L** (*form feed*).
- (d) $\backslash \mathbf{n}$ entspricht dem Steuerzeichen **Ctrl-J** (*newline*).
- (e) $\backslash \mathbf{r}$ entspricht dem Steuerzeichen **Ctrl-M** (*carriage return*).
- (f) $\backslash \mathbf{t}$ entspricht dem Steuerzeichen **Ctrl-I** (*tabulator*).
- (g) $\backslash \mathbf{v}$ entspricht dem Steuerzeichen **Ctrl-K** (*vertical tabulator*).

4. $\backslash abc \in \text{Regex}$ falls $a, b, c \in \{0, \dots, 7\}$

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 $\backslash 040$ für das Leerzeichen " " , denn das Leerzeichen hat dezimal den *Ascii*-Code 32 und in dem Oktalsystem schreibt sich diese Zahl als $40_{(8)}$.

5. $\backslash xab \in \text{Regex}$ falls $a, b \in \{0, \dots, 9, \mathbf{a}, \dots, \mathbf{f}, \mathbf{A}, \dots, \mathbf{F}\}$

Bei der Syntax $\backslash xab$ sind a und b hexadezimale Ziffern. Dann wird durch $\backslash xab$ das Zeichen spezifiziert, das im ASCII-Code an der durch die Hexadezimalzahl ab spezifizierten Stelle steht. Beispielsweise steht der Ausdruck $\backslash \mathbf{x5A}$ für den Buchstaben "Z" denn dieser Buchstabe hat dezimal den *Ascii*-Code 90 und im Hexadezimalsystem schreibt sich diese Zahl als $0\mathbf{x5A}$.

6. $\backslash uabcd \in \text{Regexp}$ falls $a, b, c, d \in \{0, \dots, 9, \text{a}, \dots, \text{f}, \text{A}, \dots, \text{F}\}$

Bei der Syntax $\backslash xbc$ sind a, b, c und d hexadezimale Ziffern. Dann wird durch $\backslash uabcd$ das Zeichen spezifiziert, das im *Unicode* die durch den Ausdruck

$$a \cdot 16^3 + b \cdot 16^2 + c \cdot 16 + d$$

gegebene Position hat. For example, $\backslash x2200$ specified the unicode character “V”.

7. $\backslash o \in \text{Regexp}$ falls $o \in \text{OpSyms}$

Die Operator-Symbole können durch Voranstellen eines Backslashes spezifiziert werden. Wollen wir beispielsweise das Zeichen “*” erkennen, so können wir dafür den regulären Ausdruck “*” verwenden.

8. $r_1 r_2 \in \text{Regexp}$ falls $r_1, r_2 \in \text{Regexp}$

Die Konkatenation zweier regulärer Ausdrücke wird in *JFlex* ohne den Infix-Operator “.” 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.

9. $r_1 | r_2 \in \text{Regexp}$ falls $r_1, r_2 \in \text{Regexp}$

Für die Alternative zweier regulärer Ausdrücke wird in *JFlex* an Stelle des Infix-Operators “+” der Operator “|” verwendet.

10. $r^* \in \text{Regexp}$ falls $r \in \text{Regexp}$

Der Postfix-Operator “*” bezeichnet den Kleene-Abschluss.

11. $r^+ \in \text{Regexp}$ falls $r \in \text{Regexp}$

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

$$r^+ \doteq r r^*.$$

12. $r? \in \text{Regexp}$ falls $r \in \text{Regexp}$

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.

13. $r\{n\} \in \text{Regexp}$ falls $n \in \mathbb{N}$

Der Ausdruck “ $r\{n\}$ ” legt fest, dass r genau n mal auftritt. Der reguläre Ausdruck “ $\text{a}\{4\}$ ” beschreibt also den String “aaaa”.

14. $r\{m, n\} \in \text{Regexp}$ falls $m, n \in \mathbb{N}$ und $m \leq n$.

Der Ausdruck “ $r\{m, n\}$ ” legt fest, dass r mehrmal auftritt und zwar mindestens m mal und höchstens n mal. Der reguläre Ausdruck “ $\text{a}\{3, 5\}$ ” beschreibt also die folgenden Strings:

“aaa”, “aaaa” und “aaaaa”.

15. $\sim r$ falls $r \in \text{Regexp}$

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

Bei der Verwendung des Operators “ \sim ” gibt es eine wichtige Einschränkung: Der Operator “ \sim ” darf nur auf der äußersten Ebene eines regulären Ausdrucks verwendet werden. Eine Konstruktion der Form

$$\sim r_1 | r_2$$

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

16. $r\$$ falls $r \in \text{Regexp}$

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.

17. r_1/r_2 falls $r_1, r_2 \in \text{Regexp}$

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.

18. $(r) \in \text{Regexp}$ falls $r \in \text{Regexp}$

Genau wie im letzten Kapitel auch können reguläre Ausdrücke geklammert werden. Für die Präzedenzen der Operatoren gilt: Die Postfix-Operatoren “*”, “?”, “+”, “{ n ” und “{ m,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* (Englisch: *character class*) 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.

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 `/* ... */` 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 Gesetzes 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 “ \sim ” an. Ist r ein regulärer Ausdruck, so spezifiziert der Ausdruck

$$\sim r$$

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

$$"/ * " \sim " * /"$$

spezifiziert werden. Dieser Ausdruck ist wie folgt zu lesen: Ein mehrzeiliger Kommentar beginnt mit dem String “ $/ *$ ” und endet mit dem ersten Auftreten des Strings “ $* /$ ”.

Intern wird der Upto-Operator mit Hilfe des Negations-Operators implementiert. Der Ausdruck

$$\sim r \quad \text{wird auf den Ausdruck} \quad !([\sim]^* r [\sim]^*) r$$

zurück geführt. Dabei steht “ $!([\sim]^* r [\sim]^*)$ ” 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

$$[:\text{digit}:]$$

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

$$\text{Character.isJavaIdentifierStart}(c)$$

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 angeben 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 `ychar` 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.

5. `%ignorecase`

Falls eine *JFlex*-Spezifikation diese Option enthält, dann spielt Groß- und Kleinschreibung keine Rolle. Beispielsweise trifft der reguläre Ausdruck `[a-z]+` dann auch auf den String `"ABC"` zu.

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

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

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 `Ytoken` hat. Die Klasse `Ytoken` wird allerdings 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 `Ytoken` dann undefiniert. Um zu vermeiden, dass der Rückgabe-Wert der Methode `yylex()` den Typ `Ytoken` bekommt, könnten wir den Scanner auch als `standalone` deklarieren, aber dann hätten wir keine Möglichkeit mehr, die Methode `main()` anzugeben. Diese Methode benötigen wir 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 Doppelpunkt 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 läßt 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.
- (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

```
/* ... */
```

Wenn wir hier einen regulären Ausdruck angeben möchten, der weder den Upto-Operator noch den Negations-Operator verwendet¹, so müßten wir den folgenden regulären Ausdruck verwenden:

$$\backslash\backslash*([\^*]|\backslash*+[\^*/])*\backslash*+/\tag{3.1}$$

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. $\backslash\backslash*$

Hierdurch wird der String “/*”, der den Kommentar einleitet, spezifiziert.

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

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

```
/* 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.

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

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 Decoment
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.

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

```

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 "&nbsp;"          { 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

- 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:
 - 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.
 - Darauf folgt eine Liste der Namen der deklarierten Zustände. Die Namen werden durch Leerzeichen getrennt.
- 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 *inklusive* 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 vorgegebene 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.

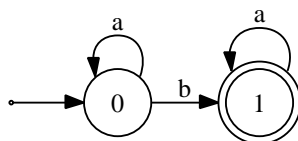


Abbildung 4.1: Ein einfacher endlicher Automat zur Erkennung der durch den regulären Ausdruck $a^* \cdot b \cdot a^*$ definierten Sprache.

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

$$a^* \cdot b \cdot a^*$$

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 9 (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. $\delta : Q \times \Sigma \rightarrow Q \cup \{\Omega\}$
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. $\Sigma = \{a, b\}$,
3. $\delta = \{ \langle 0, a \rangle \mapsto 0, \langle 0, b \rangle \mapsto 1, \langle 1, a \rangle \mapsto 1, \langle 1, b \rangle \mapsto \Omega \}$,
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

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

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

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

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 10 (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:

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

□

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

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

Satz 11 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 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 \dagger einen *neuen* Zustand, also einen Zustand, der noch nicht in der Zustands-Menge Q vorkommt. Wir nennen \dagger auch den *toten* Zustand. Dann definieren wir

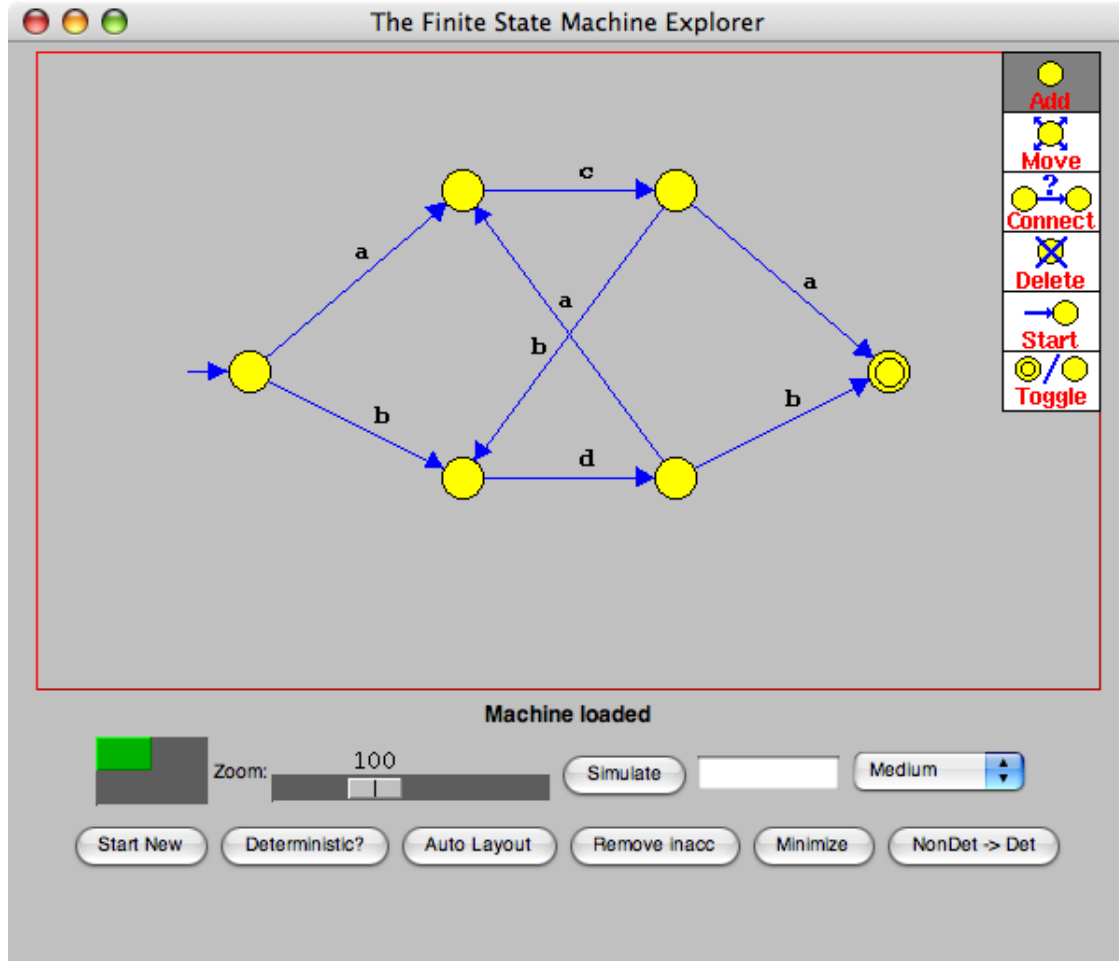


Abbildung 4.2: Das Werkzeug FSME.

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

der tote Zustand \dagger wird also der Menge Q hinzugefügt.

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

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

$$(a) \delta(q, c) \neq \Omega \rightarrow \hat{\delta}(q, c) = \delta(q, c),$$

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.

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

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 \dagger .

$$(c) \hat{\delta}(\dagger, c) = \dagger \quad \text{für alle } c \in \Sigma,$$

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 \dagger und bleibt bei allen folgenden Eingaben in diesem Zustand. Damit ist klar, dass die von A und \hat{A} akzeptierten Sprachen identisch sind. \square

Aufgabe 5: Entwickeln Sie einen endlichen Automaten, der die durch den regulären Ausdruck

$$r := (a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$$

spezifizierte Sprache erkennt.

Lösung: Der reguläre Ausdruck beschreibt genau die Wörter, die aus den Buchstaben des Alphabets $\Sigma = \{a, b\}$ gebildet sind, bei denen der drittletzte Buchstabe ein “b” ist. Demzufolge muss der Automat in seinem Zustand den Wert der drei letzten Buchstaben abspeichern. Da es für die letzten drei Buchstaben insgesamt acht verschiedene Kombinationsmöglichkeiten gibt, benötigen wir auch acht verschiedene Zustände, die wir mit den Zahlen $\{0, 1, 2, \dots, 7\}$ durchnummerieren. Im Folgenden beschreiben wir diese acht Zustände:

Zustand 0: In diesem Zustand wurden als letztes die drei Buchstaben “aaa” gelesen.

Bei den anderen Zuständen geben wir die letzten drei in dem jeweiligen Zustand gelesenen Buchstaben ohne weiteren Kommentar an.

Zustand 1: “aab”.

Zustand 2: “aba”.

Zustand 3: “abb”.

Zustand 4: “bab”.

Zustand 5: “bba”.

Zustand 6: “bbb”.

Zustand 7: “baa”.

Offensichtlich sind die Zustände 4, 5, 6 und 7 akzeptierend, denn hier ist jeweils der drittletzte gelesene Buchstabe ein “b”. Als nächstes überlegen wir, wie die Zustands-Übergangs-Funktion δ aussehen muss.

- Wir betrachten zunächst den Zustand 0. Wenn die letzten drei gelesenen Buchstaben den Wert “aaa” haben und wir als nächstes den Buchstaben “a” lesen, so haben anschließend die letzten drei Buchstaben wieder den Wert “aaa”. Damit ist klar, dass

$$\delta(0, a) = 0$$

gilt. Lesen wir hingegen im Zustand 0 den Buchstaben “b”, so sind die letzten drei gelesenen Buchstaben durch “aab” gegeben, was dem Zustand 1 entspricht. Daher haben wir

$$\delta(0, b) = 1.$$

- Wir betrachten nun den Zustand 1. Wenn die letzten drei gelesenen Buchstaben den Wert “aab” haben und wir als nächstes den Buchstaben “a” lesen, so haben anschließend die letzten drei Buchstaben den Wert “aba”, was dem Zustand 2 entspricht. Damit ist klar, dass

$$\delta(1, a) = 2$$

gilt. Lesen wir hingegen ein “b”, so haben nun die letzten drei Buchstaben den Wert “abb”, was dem Zustand 3 entspricht. Also gilt

$$\delta(1, b) = 3$$

Die restliche Berechnung der Zustands-Übergangs-Funktion verläuft nach dem für die ersten beiden Zustände demonstrierten Verfahren und wird daher an dieser Stelle nicht weiter erklärt, das Ergebnis der Rechnung ist in Abbildung 4.3 auf Seite 35 gezeigt. Wir müssen lediglich noch erklären, was der Start-Zustand des endlichen Automaten ist. Zu Beginn hat der Automat noch keinen Buchstaben gelesen. Das heißt insbesondere, dass die letzten drei Buchstaben alle von dem Buchstaben “b” verschieden sind. Damit können wir aber den Zustand 0 als Start-Zustand nehmen, denn die Frage, ob wir als letztes den String “aaa” gelesen haben oder ob wir noch gar nichts gelesen haben ist irrelevant, wenn es nur darum geht zu entscheiden, ob der drittletzte gelesene Buchstabe ein “b” ist.

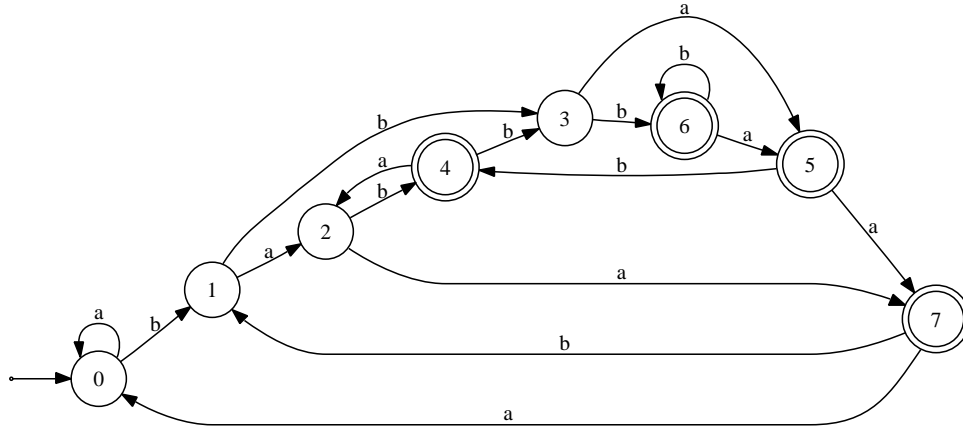


Abbildung 4.3: Ein endlicher Automat für die Sprache, die durch den regulären Ausdruck $(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$ definiert wird.

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. Der in der letzten Aufgabe entwickelte und in Abbildung 4.3 gezeigte Automat belegt diese Behauptung. 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.

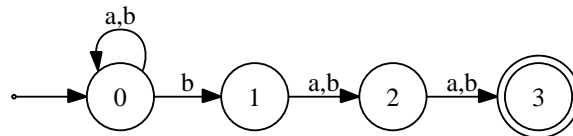


Abbildung 4.4: Ein Nicht-deterministischer Automat für die Sprache, die durch den regulären Ausdruck $(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$ definiert wird.

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

$$(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$$

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 “ $abab$ ”:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$$

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:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$$

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 “ $bbbb$ ” gelesen wird:

$$q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 \xrightarrow{b} \Omega$$

Hier ist der Automat zu früh in den Zustand q_1 gewechselt, was bei der Eingabe des letzten Zeichens 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 “ $bbbb$ ” erkennen können:

$$q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3$$

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 -*Transitions*: Darunter verstehen wir einen Zustands-Übergang, bei dem kein Zeichen der Eingabe gelesen wird. Wir schreiben einen solchen spontanen Übergang vom Zustand q_1 in den Zustand q_2 als

$$q_1 \xrightarrow{\varepsilon} q_2.$$

Definition 12 (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 Funktion auf $Q \times (\Sigma \cup \{\varepsilon\})$, die jedem Paar $\langle q, a \rangle$ aus $Q \times (\Sigma \cup \{\varepsilon\})$ eine Menge $\delta(q, a) \subseteq Q$ zuordnet, es gilt also

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q.$$

Falls $a \in \Sigma$ ist, interpretieren wir $\delta(q, a)$ als die Menge der Zustände, in denen der Automat sein kann, wenn im Zustand q das Symbol a gelesen wurde. Die Menge $\delta(q, \varepsilon)$ ist die Menge der Zustände, die der Automat aus dem Zustand q mit einem ε -Übergang erreichen kann.

4. $q_0 \in Q$ ist der Start-Zustand.

5. $F \subseteq Q$ ist die Menge der akzeptierenden Zustände.

Falls $q_2 \in \delta(q_1, \varepsilon)$ ist, dann sagen wir, dass der Automat eine ε -Transition von dem Zustand q_1 in den Zustand q_2 hat. Wir schreiben dies als

$$q_1 \xrightarrow{\varepsilon} q_2.$$

Falls $c \in \Sigma$ ist und $q_2 \in \delta(q_1, c)$ gilt, so schreiben wir

$$q_1 \xrightarrow{c} q_2. \quad \square$$

Beispiel: Für den in Abbildung 4.4 auf Seite 35 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. $\Sigma = \{a, b\}$.
3. $\delta = \{ \langle q_0, a \rangle \mapsto \{q_0\}, \langle q_0, b \rangle \mapsto \{q_0, q_1\}, \langle q_0, \varepsilon \rangle \mapsto \{\}, \langle q_1, a \rangle \mapsto \{q_2\}, \langle q_1, b \rangle \mapsto \{q_2\}, \langle q_1, \varepsilon \rangle \mapsto \{\},$
 $\langle q_2, a \rangle \mapsto \{q_3\}, \langle q_2, b \rangle \mapsto \{q_3\}, \langle q_2, \varepsilon \rangle \mapsto \{\} \}.$

Die Zustands-Übergangs-Funktion δ kann übersichtlicher auch durch die Transitionen

$$\begin{array}{llll} q_0 \xrightarrow{a} q_0, & q_0 \xrightarrow{b} q_0, & q_0 \xrightarrow{b} q_1, & q_1 \xrightarrow{a} q_2, \\ q_1 \xrightarrow{b} q_2, & q_2 \xrightarrow{a} q_3 & \text{und} & q_2 \xrightarrow{b} q_3 \end{array}$$

angegeben werden.

4. Der Start-Zustand ist q_0 .
5. $F = \{q_3\}$, der einzige akzeptierende Zustand ist also q_3 . \square

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

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

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

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

Hier werden die ε -Transitionen erfaßt.

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

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

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:

$$L(A) = \{w \in \{a, b\}^* \mid |w| \geq 3 \wedge w[|w| - 2] = b\} \quad \diamond$$

Aufgabe 6: Geben Sie einen NEA A an, so dass $L(F)$ aus genau den Wörtern der Sprache $\{a, b\}^*$ besteht, die den Teilstring “aba” enthalten. \diamond

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

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

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 deterministischen Automaten $\det(A)$ sind also **Mengen** von Zuständen des ursprünglichen nicht-deterministischen 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 (Englisch: ε -closure). Die Funktion

$$ec : Q \rightarrow 2^Q$$

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.: $p \in ec(q) \wedge r \in \delta(p, \varepsilon) \rightarrow r \in ec(q)$.

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

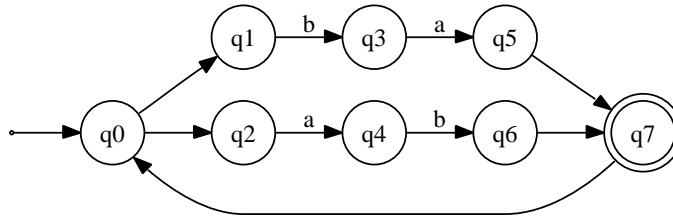


Abbildung 4.5: Nicht-deterministischer Automat mit ε -Transitionen.

Beispiel: Abbildung 4.5 zeigt einen nicht-deterministischen endlichen Automaten mit ε -Transitionen. Die ε -Transitionen sind in der Abbildung die Pfeile, die nicht mit einem Buchstaben beschriftet sind. 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. $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,

$$8. \text{ec}(q_7) = \{q_7, q_0, q_1, q_2\}. \quad \square$$

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

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

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

$$\delta^*(q_1, c) := \bigcup \{ \text{ec}(q_2) \mid q_2 \in \delta(q_1, c) \}.$$

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 $\text{ec}(q_2)$.
2. Anschließend vereinigen wir alle diese Mengen $\text{ec}(q_2)$.

Beispiel: In Fortführung des obigen Beispiels erhalten wir beispielsweise:

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

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

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

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

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

denn vom Zustand q_3 geht der Automat beim Lesen von \mathbf{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. \diamond

Die Funktion δ^* überführt einen Zustand in eine Menge von Zuständen. Da der zu entwickelnde endliche Automat $\text{det}(A)$ 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

$$\Delta : 2^Q \times \Sigma \rightarrow 2^Q$$

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

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

Diese Formel ist einfach zu 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:

$$1. \Delta(\{q_0, q_1, q_2\}, \mathbf{a}) = \{q_4\},$$

$$2. \Delta(\{q_0, q_1, q_2\}, \mathbf{b}) = \{q_3\},$$

$$3. \Delta(\{q_3\}, \mathbf{a}) = \{q_7, q_0, q_1, q_2\},$$

4. $\Delta(\{q_3\}, \mathbf{b}) = \{\}$,
5. $\Delta(\{q_4\}, \mathbf{a}) = \{\}$,
6. $\Delta(\{q_4\}, \mathbf{b}) = \{q_7, q_0, q_1, q_2\}$.

◇

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

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

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

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

Aufgabe 7: Transformieren Sie den in Abbildung 4.4 auf Seite 35 gezeigten nicht-deterministischen Automaten A in einen deterministischen Automaten $\det(A)$. ◇

Lösung: Wir berechnen zunächst die verschiedenen möglichen Zustands-Mengen:

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 der nicht-deterministische Automaten A beim Lesen von \mathbf{a} in den Zustand q_0 über. Also gilt

$$\Delta(S_0, \mathbf{a}) = \Delta(\{q_0\}, \mathbf{a}) = \{q_0\} = S_0.$$

3. Von dem Zustand q_0 geht A beim Lesen von \mathbf{b} in den Zustand q_0 oder q_1 über. Also gilt

$$S_1 := \Delta(S_0, \mathbf{b}) = \Delta(\{q_0\}, \mathbf{b}) = \{q_0, q_1\}.$$

4. Wir haben $\delta(q_0, \mathbf{a}) = \{q_0\}$ und $\delta(q_1, \mathbf{a}) = \{q_2\}$. Daher folgt

$$S_2 := \Delta(S_1, \mathbf{a}) = \Delta(\{q_0, q_1\}, \mathbf{a}) = \{q_0, q_2\}.$$

5. Wir haben $\delta(q_0, \mathbf{b}) \in \{q_0, q_1\}$ und $\delta(q_1, \mathbf{b}) = \{q_2\}$. Daher folgt

$$S_4 := \Delta(S_1, \mathbf{b}) = \Delta(\{q_0, q_1\}, \mathbf{b}) = \{q_0, q_1, q_2\}$$

6. $S_3 := \Delta(S_2, \mathbf{a}) = \Delta(\{q_0, q_2\}, \mathbf{a}) = \{q_0, q_3\}$.

7. $S_5 := \Delta(S_2, \mathbf{b}) = \Delta(\{q_0, q_2\}, \mathbf{b}) = \{q_0, q_1, q_3\}$.

8. $S_6 := \Delta(S_4, \mathbf{a}) = \Delta(\{q_0, q_1, q_2\}, \mathbf{a}) = \{q_0, q_2, q_3\}$.
9. $S_7 := \Delta(S_4, \mathbf{b}) = \Delta(\{q_0, q_1, q_2\}, \mathbf{b}) = \{q_0, q_1, q_2, q_3\}$.
10. $\Delta(S_3, \mathbf{a}) = \Delta(\{q_0, q_3\}, \mathbf{a}) = \{q_0\} = S_0$.
11. $\Delta(S_3, \mathbf{b}) = \Delta(\{q_0, q_3\}, \mathbf{b}) = \{q_0, q_1\} = S_1$.
12. $\Delta(S_5, \mathbf{a}) = \Delta(\{q_0, q_1, q_3\}, \mathbf{a}) = \{q_0, q_2\} = S_2$.
13. $\Delta(S_5, \mathbf{b}) = \Delta(\{q_0, q_1, q_3\}, \mathbf{b}) = \{q_0, q_1, q_2\} = S_4$.
14. $\Delta(S_6, \mathbf{a}) = \Delta(\{q_0, q_2, q_3\}, \mathbf{a}) = \{q_0, q_3\} = S_3$.
15. $\Delta(S_6, \mathbf{b}) = \Delta(\{q_0, q_2, q_3\}, \mathbf{b}) = \{q_0, q_1, q_3\} = S_5$.
16. $\Delta(S_7, \mathbf{a}) = \Delta(\{q_0, q_1, q_2, q_3\}, \mathbf{a}) = \{q_0, q_2, q_3\} = S_6$.
17. $\Delta(S_7, \mathbf{b}) = \Delta(\{q_0, q_1, q_2, q_3\}, \mathbf{b}) = \{q_0, q_1, q_2, q_3\} = S_7$.

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

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

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

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

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

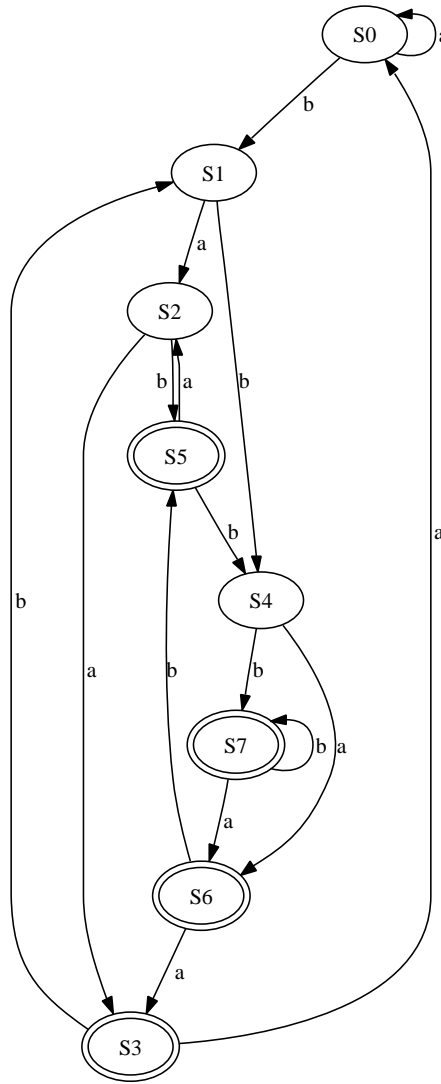
$$\det(A) := \langle \hat{Q}, \Sigma, \Delta, S_0, \hat{F} \rangle.$$

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.

Vergleichen wir den in Abbildung 4.6 gezeigten Automaten mit dem in Abbildung 4.3 gezeigten Automaten, so stellen wir fest, dass diese Automaten bis auf die Benennung der Zustände identisch sind. Es gilt

$$S_0 \approx 0, \quad S_1 \approx 1, \quad S_2 \approx 2, \quad S_3 \approx 7, \quad S_4 \approx 3, \quad S_5 \approx 4, \quad S_6 \approx 5, \quad \text{und} \quad S_7 \approx 6.$$

Aufgabe 8: Transformieren Sie den in Abbildung 4.5 auf Seite 38 gezeigten endlichen Automaten in einen äquivalenten deterministischen endlichen Automaten. \diamond

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

4.3.1 Implementing the Conversion of NFA to DFA

It is straightforward to implement the theory developed so far using the programming language **SETLX**. Figure 4.7 on page 43 shows a SETLX program that converts a given non-deterministic finite state machine **nfa** into a deterministic finite state machine. We discuss this program line by line.

```

1  var delta;                                // declare delta as global variable
2  nfa2dfa := procedure(nfa) {
3      [states, sigma, delta, q0, f] := nfa;
4      newStart := epsilonClosure(q0);
5      newStates := { newStart };
6      while (true) {
7          more := { capitalDelta(m, c) : m in newStates, c in sigma };
8          if (more <= newStates) { break; }
9          newStates += more;
10     }
11     newFinal := { m in newStates | m * f != {} };
12     return [newStates, sigma, capitalDelta, newStart, newFinal];
13 };
14 epsilonClosure := procedure(s) {
15     result := { s };
16     while (true) {
17         newStates := {} +/ { delta(q, "") : q in result };
18         if (newStates <= result) { break; }
19         result += newStates;
20     }
21     return result;
22 };
23 capitalDelta := procedure(m, c) {
24     return {} +/ { deltaStar(q, c) : q in m };
25 };
26 deltaStar := procedure(s, c) {
27     return {} +/ { epsilonClosure(q) : q in delta(s, c) };
28 };

```

Abbildung 4.7: A SETLX program to convert an NFA into a DFA.

1. In line 1, we declare the variable **delta** as a global variable using the keyword **var**.
2. The function **nfa2dfa** takes as input a non-deterministic finite automaton **nfa**. In line 3, this automaton is split up into its components.
3. In line 4 we compute the start state of the deterministic automaton as the ε -closure of the state q_0 . This computation is done using the function **epsilonClosure** that is defined below.
4. Next, we compute the set of all states of the deterministic automaton in line 5. This set is stored in the variable **newStates**. Initially, this set contains only the new start state computed in the previous line.
 Given a set of states **newStates**, we compute in line 7 all those states that can be reached from the given state using the function Δ and some character **c** from the alphabet Σ . If we do not find any new states, the loop is terminated with the **break** statement in line 8. Otherwise, we add the new states to those states already found.
5. The new set of final states is defined in line 12 as the set of all those sets of states that have a non-empty intersection with the set **f** of final states of the non-deterministic automaton.

6. The function **epsilonClosure** computes the ε -closure of a given state **s**. The transition function **delta** has to be provided as a second argument. The idea of the computation is to initialize the set $ec(s)$ with $\{s\}$ and then to add more states into this set by testing for all states found so far, whether there is an ε transition to a state that has not yet been discovered.
7. The function **capitalDelta** takes as argument a set **m** of states of the deterministic automaton and a character **c** and computes $\Delta(m, c)$. As the operator “+ /” computes the union of all sets that are elements of its argument, this function does indeed compute the expression

$$\bigcup \{ \delta^*(q, c) \mid q \in m \}.$$

We have to take some care as the expression

$$+ / \{ \}$$

is undefined. The reason is that for a set $s = \{x_1, x_2, \dots, x_n\}$ the value of the expression “+ / s ” is defined as

$$s_1 + s_2 + \dots + s_n.$$

If the elements s_i of the set s are numbers, then they are added. If they are sets, the union of the sets s_i is computed. The problem is the following: If s is empty, then we don’t know whether s is the empty set of numbers or whether it is the empty set of sets and therefore in this case “+ / s ” is undefined. By writing “ $\{ \} + / s$ ” we insert the empty set $\{ \}$ into the set s before evaluating the expression “+ / s ”. Then it is guaranteed that s is not empty and the problem is solved.

8. The function **deltaStar** takes a state s of the non-deterministic finite automaton **nfa** and computes all states that can be reached from the state s when the character c is read. This function satisfies the specification

$$\delta^*(s, c) := \bigcup \{ ec(q) \mid q \in \delta(s, c) \}.$$

4.4 Übersetzung regulärer Ausdrücke in NEA

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

$$L(A(r)) = L(r)$$

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 $start(A(r))$, so gilt also:

$$\forall p \in Q : \left(start(A(r)) \notin \delta(p, \varepsilon) \wedge \forall c \in \Sigma : start(A(r)) \notin \delta(p, c) \right).$$

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

$$\delta(accept(A(r)), \varepsilon) = \{ \} \wedge \forall c \in \Sigma : \delta(accept(A(r)), c) = \{ \}.$$

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

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

Beachten Sie, dass dieser Automat keinerlei Transitionen hat, für die Zustands-Übergangs-Funktion δ gilt also sowohl

$$\forall q \in Q : \forall c \in \Sigma : \delta(q, c) = \{\} \quad \text{als auch} \quad \forall q \in Q : \delta(q, \varepsilon) = \{\}.$$

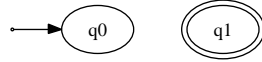
Abbildung 4.8: Der Automat $A(\emptyset)$.

Abbildung 4.8 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 Funktion δ liefert für alle Argumente die leere Menge, der Automat hat also keinerlei Zustands-Übergänge und akzeptiert daher nur die leere Sprache. Damit gilt $L(\emptyset) = \{\}$.

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

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

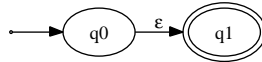
Abbildung 4.9: Der Automat $A(\varepsilon)$.

Abbildung 4.9 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

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

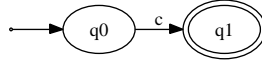
Abbildung 4.10: Der Automat $A(c)$.

Abbildung 4.10 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.

4. Um den Automaten $A(r_1 \cdot r_2)$ für die Konkatenation $r_1 \cdot r_2$ 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 $A(r_2)$ umbenennen. 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) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\} \rangle$,
- (c) $Q_1 \cap Q_2 = \{\}$.

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

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

Die Notation $\{\langle q_2, \varepsilon \rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2$ ist so zu verstehen, dass die so spezifizierte Funktion δ alle Übergänge enthält, die durch die Zustands-Übergangs-Funktionen δ_1 und δ_2 spezifiziert sind. Dazu kommt dann noch der ε -Übergang von q_2 nach q_3 . Formal könnten wir daher die Funktion δ auch wie folgt spezifizieren:

$$\delta(q, c) := \begin{cases} \{q_3\} & \text{falls } q = q_2 \text{ und } c = \varepsilon, \\ \delta_1(q, c) & \text{falls } q \in Q_1 \text{ und } \langle q, c \rangle \neq \langle q_2, \varepsilon \rangle, \\ \delta_2(q, c) & \text{falls } q \in Q_2. \end{cases}$$

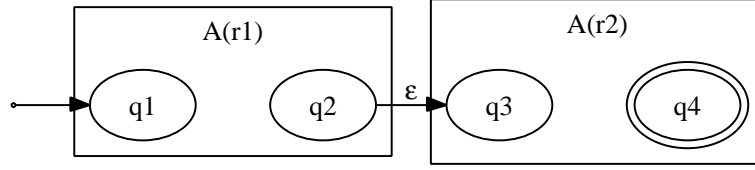
Abbildung 4.11: Der Automat $A(r_1 \cdot r_2)$.

Abbildung 4.11 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 Automat dann etwas kleiner wird. Bei den praktischen Übungen werden wir diese Zustände daher 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) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_2, \{q_4\} \rangle$,
- (c) $Q_1 \cap Q_2 = \{\}$.

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

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

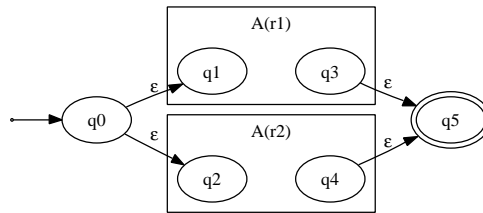
Abbildung 4.12: Der Automat $A(r_1 + r_2)$.

Abbildung 4.12 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-Übergängen 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 $A(r^*)$ aus dem Automaten $A(r)$ konstruieren: Der Automat $A(r^*)$ ist durch den Ausdruck

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

gegeben.

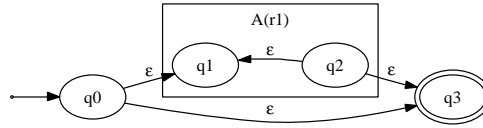


Abbildung 4.13: Der Automat $A(r^*)$.

Abbildung 4.13 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-Übergängen 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 9: Konstruieren Sie einen nicht-deterministischen endlichen Automaten, der die durch den regulären Ausdruck

$$(a + b) \cdot a^* \cdot b$$

spezifizierte Sprache erkennt.

Aufgabe 10: Konstruieren Sie einen nicht-deterministischen endlichen Automaten, der die durch den regulären Ausdruck

$$a^* \cdot b^*$$

spezifizierte Sprache erkennt. Überlegen Sie sich, warum Sie in diesem Beispiel bei der Konstruktion zur Berechnung des Kleene-Abschlusses die Zustände, die in dem im Text beschriebenen Algorithmus die Namen q_0 und q_1 bzw. q_2 und q_3 haben, nicht identifizieren dürfen.

4.4.1 Implementing the Conversion of Regular Expressions into Finite State Machines

In this section, we develop a SETLX program that transforms a regular expression into a non-deterministic finite state machine. In order to do so, we first have to decide how to represent regular expressions. As we have not yet developed enough theory to implement a parser for regular expressions, we will represent complex regular

expressions as terms. In SETLX, a term has the form

$$F(t_1, \dots, t_N).$$

Here, F needs to be a *functor*, which is just an uninterpreted function symbols. In SETLX, functors have to start with a capital letter. The arguments t_1, \dots, t_n are either terms, strings, numbers, sets, or lists. In general, we define a function

$$\text{rep} : \text{RegExp} \rightarrow \text{SETLX}$$

that maps a regular expression r into a value of the programming language SETLX. The idea is that for a given regular expression r , $\text{rep}(r)$ is the SETLX term that represents the regular expression r . We define $\text{rep}(r)$ by induction on the definition of the regular expression r .

1. $\text{rep}(\emptyset) := 0$.

The regular expression \emptyset is represented as the number 0.

2. $\text{rep}(\varepsilon) := ""$.

The regular expression ε is represented as the empty string.

3. $\text{rep}(c) := "c"$.

A single character c is represented by the string containing just this character.

4. $\text{rep}(r_1 \cdot r_2) := \text{Cat}(\text{rep}(r_1), \text{rep}(r_2))$,

so the concatenation $r_1 r_2$ of two regular expressions r_1 and r_2 is represented by a term with the functor “Cat”, where the arguments of this functor are the representations of the regular expressions r_1 and r_2 .

5. $\text{rep}(r_1 + r_2) := \text{Or}(r_1, r_2)$,

so the disjunction $r_1 + r_2$ of two regular expressions r_1 and r_2 is represented by a term with the functor “Or”, where the arguments of this functor are the representations of the regular expressions r_1 and r_2 .

6. $\text{rep}(r^*) := \text{Star}(\text{rep}(r))$,

so the Kleene closure r^* of the regular expression r is represented by a term with the functor “Star”, where the argument of this functor is the representations of the regular expressions r .

```

1  regexp2NFA := procedure(r, sigma) {
2      match (r) {
3          case 0 : return genEmptyNFA(sigma);
4          case "": return genEpsilonNFA(sigma);
5          case c | isString(c) && #c == 1:
6              return genCharNFA(sigma, c);
7          case Cat(r1, r2):
8              return concatenate(regexp2NFA(r1, sigma), regexp2NFA(r2, sigma));
9          case Or(r1, r2):
10             return disjunction(regexp2NFA(r1, sigma), regexp2NFA(r2, sigma));
11         case Star(r0):
12             return kleene(regexp2NFA(r0, sigma));
13         default: abort("$r$ is not a suitable regular expression");
14     }
15 };

```

Abbildung 4.14: Converting a regular expression into a non-deterministic finite state machine.

This leads to the implementation of the function `regexp2NFA` that is shown in Figure 4.14 on page 48. The function `regexp2NFA` gets a regular expression r as its first input. The second input is the alphabet Σ . The

implementation branches in line 2 on the form of the regular expression r . Altogether, there are six different cases.

1. If $r = 0$, then r represents the regular expression \emptyset . In this case, the function `genEmptyNFA` computes the finite state machine that is shown in Figure 4.8.

The implementation of the function `genEmptyNFA` is shown in Figure 4.15 on page 49: First, we generate two new states `q0` and `q1`. The transition function `delta` is defined in line 4 as the function mapping every pair $\langle q, c \rangle$ to the empty set. Finally, line 5 returns the 5-tuple

$$\langle \{q0, q1\}, \Sigma, \delta, q0, \{q1\} \rangle$$

as the result. This corresponds one-to-one with the finite state machine shown in Figure 4.8.

```

1  genEmptyNFA := procedure(sigma) {
2      q0 := getNewState();
3      q1 := getNewState();
4      delta := [q, c] |-> {};
5      return [ {q0, q1}, sigma, delta, q0, { q1 } ];
6  };

```

Abbildung 4.15: Generating the NFA shown in Figure 4.8.

2. If $r = ""$, then r represents the regular expression ε . In this case, the function `genEpsilonNFA` computes the finite state machine that is shown in Figure 4.9.

The implementation of the function `genEpsilonNFA` is shown in Figure 4.16 on page 50: First, we generate two new states `q0` and `q1`. The transition function `delta` is defined in lines 4–11 using the auxilliary function `closure`. Essentially, the function `closure` takes a string, parses this string, interprets the string as a function, and finally returns this function as the result. Essentially, the function `delta` that is defined here, has the following form

```

procedure(q, c) {
    if (q == q0 && c == "") {
        return { q1 };
    } else {
        return {};
    }
};

```

Unfortunately, we can not define `delta` in this simple form. The reason is that the current version of SETLX does not yet support true *closures*: If we would assign this function to `delta`, then, when we later call this function `delta` in a different context, the variables `q0` and `q1` would not relate to the values of the variables `q0` and `q1` that are defined in the function `genEpsilonNFA`.

The trick to solve this problem is to generate a string containing the values of `q0` and `q1` with the help of string interpolation. This string is then parsed as a function and this function is finally assigned to `delta`. The parsing is done in the function `closure` which is shown in Figure 4.22 on page 53.

3. If $r = "c"$, then r represents a single character c . In this case, the function `genCharNFA` computes the finite state machine that is shown in Figure 4.10.

The implementation of the function `genCharNFA` is shown in Figure 4.17 on page 50: After generating two new states `q0` and `q1`, the transition function `delta` is defined in lines 4–11 using the same trick that has been discussed above. In this case, the true form of the function `delta` is as follows:

```

1  genEpsilonNFA := procedure(sigma) {
2      q0 := getNewState();
3      q1 := getNewState();
4      delta := closure(
5          "procedure(q, c) {
6              if (q == \"$q0$\" && c == "\\") {
7                  return { \"$q1$\" };
8              } else {
9                  return {};
10             }
11         }");
12     return [ {q0, q1}, sigma, delta, q0, { q1 } ];
13 };

```

Abbildung 4.16: Generating the NFA shown in Figure 4.9.

```

    procedure(q, d) {
        if (q == q0 && d == c) {
            return { q1 };
        } else {
            return {};
        }
    };

```

```

1  genCharNFA := procedure(sigma, c) {
2      q0 := getNewState();
3      q1 := getNewState();
4      delta := closure(
5          "procedure(q, d) {
6              if (q == \"$q0$\" && d == \"$c$\") {
7                  return { \"$q1$\" };
8              } else {
9                  return {};
10             }
11         }");
12     return [ {q0, q1}, sigma, delta, q0, { q1 } ];
13 };

```

Abbildung 4.17: Generating the NFA shown in Figure 4.10.

4. If $r = \text{Cat}(r_1, r_2)$, then r represents the concatenation $r_1 \cdot r_2$ of the regular expressions r_1 and r_2 . In this case, we recursively compute two finite state machines that recognize the strings corresponding to these regular expressions. These two finite state machines are then combined into a new finite state machine using the function `catenate`. The function `catenate` works as specified in Figure 4.11.

The implementation of the function `catenate` is shown in Figure 4.18 on page 51: The function `catenate` receives two finite state machines `f1` and `f2` as arguments. These finite state machines are split into their components in line 2 and line 3. The function `arb(s)` extracts an arbitrary element from the set s . This function is called in line 4 and 5 to extract the accepting states `q2` and `q4`. This works since we know that

the sets of accepting states **f1** and **f2** contain exactly one state each, so the function **arb** doesn't really have a choice which state to pick.

The most important purpose of the function **catenate** is to construct a new transition function δ such that

$$\delta = \delta_1 \cup \delta_2 \cup \{ \langle q_2, \varepsilon \rangle \mapsto q_3 \}$$

holds. This is done by defining the function **delta** essentially as the following procedure:

```

procedure(q, c) {
  if (q == q2 && c == "") {
    return { q3 };
  } else if (q in m1) {
    d1 := delta1;
    return d1(q, c);
  } else if (q in m2) {
    d2 := delta2;
    return d2(q, c);
  } else {
    return {};
  }
};

```

As above, we have to use the function **closure** to preserve the values of the local variables **q2**, **q3**, **q1**, **q1**, **delta1**, and **delta2** outside of their context.

```

1  catenate := procedure(f1, f2) {
2    [m1, sigma, delta1, q1, a1] := f1;
3    [m2, _, delta2, q3, a2] := f2;
4    q2 := arb(a1);
5    q4 := arb(a2);
6    delta := closure(
7      "procedure(q, c) {
8        if (q == \"$q2$\" && c == \"\") {
9          return { \"$q3$\" };
10       } else if (q in $m1$) {
11         d1 := $delta1$;
12         return d1(q, c);
13       } else if (q in $m2$) {
14         d2 := $delta2$;
15         return d2(q, c);
16       } else {
17         return {};
18       }
19     }");
20    return [ m1 + m2, sigma, delta, q1, a2 ];
21  };

```

Abbildung 4.18: Generating the NFA shown in Figure 4.11.

5. The remaining two cases correspond to the diagrams shown in Figure 4.12 and Figure 4.13. As these cases are quite similar to the last case, there is no need for a detailed discussion.

```

1  disjunction := procedure(f1, f2) {
2      [m1, sigma, delta1, q1, a1] := f1;
3      [m2,      _, delta2, q2, a2] := f2;
4      q3 := arb(a1);
5      q4 := arb(a2);
6      q0 := getNewState();
7      q5 := getNewState();
8      delta := closure(
9          "procedure(q, c) {
10             if (q == \"$q0$\" && c == "\\") {
11                 return { \"$q1$\", \"$q2$\" };
12             } else if (q in { \"$q3$\", \"$q4$\" } && c == "\\") {
13                 return { \"$q5$\" };
14             } else if (q in $m1$) {
15                 d1 := $delta1$;
16                 return d1(q, c);
17             } else if (q in $m2$) {
18                 d2 := $delta2$;
19                 return d2(q, c);
20             } else {
21                 return {};
22             }
23         }");
24      return [ { q0, q5 } + m1 + m2, sigma, delta, q0, { q5 } ];
25  };

```

Abbildung 4.19: Generating the NFA shown in Figure 4.12.

```

1  kleene := procedure(f) {
2      [m, sigma, delta0, q1, a] := f;
3      q2 := arb(a);
4      q0 := getNewState();
5      q3 := getNewState();
6      delta := closure(
7          "procedure(q, c) {
8             if (q == \"$q0$\" && c == "\\") {
9                 return { \"$q1$\", \"$q3$\" };
10             } else if (q == \"$q2$\" && c == "\\") {
11                 return { \"$q1$\", \"$q3$\" };
12             } else if (q in $m$) {
13                 d := $delta0$;
14                 return d(q, c);
15             } else {
16                 return {};
17             }
18         }");
19      return [ { q0, q3 } + m, sigma, delta, q0, { q3 } ];
20  };

```

Abbildung 4.20: Generating the NFA shown in Figure 4.13.

```

1  var gStateCount;
2  gStateCount := -1;
3
4  getNewState := procedure() {
5      gStateCount += 1;
6      return "q" + gStateCount;
7  };

```

Abbildung 4.21: Generating a new state.

```

1  closure := procedure(s) {
2      return evalTerm(parse(s));
3  };

```

Abbildung 4.22: The function closure.

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

$$L(r(A)) = L(A)$$

gilt. Der Automat A habe die Form

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

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:

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

Die Definition der regulären Ausdrücke erfolgt über einen Trick: Wir definieren für $k = 0, \dots, n+1$ 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

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

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

$$p \xrightarrow{w}_k q$$

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 $p \xrightarrow{w}_k q$ erfolgt durch eine Induktion nach der Länge des Wortes w :

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

$$(a) \quad p \xrightarrow{\varepsilon}_k p,$$

denn mit dem leeren Wort kann von p aus nur der Zustand p erreicht werden.

$$(b) \delta(p, c) = q \Rightarrow p \xrightarrow{c}_k q,$$

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.

$$\text{I.S.: } p \xrightarrow{c}_k q \wedge q \notin Q_k \wedge q \xrightarrow{w}_k r \Rightarrow p \xrightarrow{cw}_k r.$$

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 $k = 0, \dots, n+1$ die regulären Ausdrücke $r^{(k)}(p_1, p_2)$ definieren. Wir werden diese regulären Ausdrücke so definieren, dass hinterher

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

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 $p_1 \xrightarrow{w}_0 p_2$ nur dann gelten, wenn w aus einem einzigen Buchstaben besteht. Es sei

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

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

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

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

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

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,

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

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 $p_1 \xrightarrow{w}_k p_2$.

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

- $p_1 \xrightarrow{w_1}_k q_k$,
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.

- $q_k \xrightarrow{s_i} q_k$ für alle $i = \{1, \dots, l\}$,
von dem Zustand q_k wechselt der Automat beim Lesen der Teilstrings s_i wieder in den Zustand q_k .
- $q_k \xrightarrow{w_2} p_2$,
schließlich wechselt der Automat von dem Zustand q_k in den Zustand p_2 , wobei der Rest w_2 gelesen wird.

Daher definieren wir

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

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 $(r^{(k)}(q_k, q_k))^*$ 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

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

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, es gilt also

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

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

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

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

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

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

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 11: 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) $r^{(0)}(0, 0) = \mathbf{a} + \varepsilon$,
- (b) $r^{(0)}(0, 1) = \mathbf{b}$,
- (c) $r^{(0)}(1, 0) = \emptyset$,
- (d) $r^{(0)}(1, 1) = \mathbf{a} + \varepsilon$.

2. Für $k = 1$ haben wir:

(a) Für $r^{(1)}(0, 0)$ finden wir:

$$\begin{aligned} r^{(1)}(0, 0) &= r^{(0)}(0, 0) + r^{(0)}(0, 0) \cdot (r^{(0)}(0, 0))^* \cdot r^{(0)}(0, 0) \\ &= r^{(0)}(0, 0) \cdot (r^{(0)}(0, 0))^* \end{aligned}$$

wobei wir im letzten Schritt die für reguläre Ausdrücke allgemeingültige Umformungen

$$\begin{aligned} r + r \cdot r^* \cdot r &= r \cdot (\varepsilon + r^* \cdot r) \\ &= r \cdot r^* \end{aligned}$$

verwendet haben. Setzen wir für $r^{(0)}(0, 0)$ den oben gefundenen Ausdruck $\mathbf{a} + \varepsilon$ ein, so erhalten wir

$$r^{(1)}(0, 0) = (\mathbf{a} + \varepsilon) \cdot (\mathbf{a} + \varepsilon)^*.$$

Wegen $(\mathbf{a} + \varepsilon) \cdot (\mathbf{a} + \varepsilon)^* = \mathbf{a}^*$ haben wir insgesamt

$$r^{(1)}(0, 0) = \mathbf{a}^*.$$

(b) Für $r^{(1)}(0, 1)$ finden wir:

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

(c) Für $r^{(1)}(1, 0)$ finden wir:

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

(d) Für $r^{(1)}(1, 1)$ finden wir

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

3. Für $k = 2$ erhalten wir:

(a) Für $r^{(2)}(0, 0)$ finden wir

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

(b) Für $r^{(2)}(0, 1)$ finden wir

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

(c) Für $r^{(2)}(1, 0)$ finden wir

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

(d) Für $r^{(2)}(1, 1)$ finden wir

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

Da der Zustand 0 der Start-Zustand ist und der Zustand 1 der einzige akzeptierende Zustand ist, können wir nun den regulären Ausdruck $r(A)$ angeben:

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

Dieses Ergebnis, das wir 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 unmittelbar erkennt und der beschriebene Algorithmus hat den Vorteil, dass er sich implementieren läßt. \square

4.5.1 Implementing the Conversion of FSMs into Regular Expressions

Figure 4.23 on page 58 shows how to implement the conversion of a finite state machine into a regular expression. We discuss the details of this implementation next.

1. The function `dfa2RegExp` takes a deterministic finite state machine `dfa` as input. For every accepting state p of the given `dfa`, it calculates the regular expression $r(q_0, p)$, which describes those strings that transform the finite state machine from the start state q_0 into the state p . If $\{p_1, \dots, p_k\}$ is the set of all accepting states of `dfa`, then the regular expression

$$r(q_0, p_1) + \dots + r(q_0, p_k)$$

describes the language accepted by `dfa`. This regular expression is computed in line 3 via the two functions `regexpSum` and `rpq`.

2. The function `regexpSum` takes as input a set s of regular expressions. If s has the form

$$\{r_1, \dots, r_k\},$$

then the regular expression

$$r_1 + \dots + r_k$$

is returned. There are two special cases:

- (a) If s is empty, then the regular expression \emptyset is returned.
- (b) If s contains just one element, that is if s has the form $s = \{r\}$, then r is returned.

3. The function `rpq` is called with 5 arguments:

```

1  dfa2RegExp := procedure(dfa) {
2      [states, sigma, delta, q0, accepting] := dfa;
3      return regexpSum({ rpq(q0, p, sigma, delta, states) : p in accepting });
4  };
5  regexpSum := procedure(s) {
6      match (s) {
7          case {}:
8              return 0;
9          case { r }:
10             return r;
11         case { r | rs }:
12             return Or(r, regexpSum(rs));
13     }
14 };
15 rpq := procedure(p1, p2, sigma, delta, allowed) {
16     match (allowed) {
17         case {}:
18             allChars := { c in sigma | delta(p1, c) == p2 };
19             sum := regexpSum(allChars);
20             if (p1 == p2) {
21                 if (allChars == {}) {
22                     return "";
23                 } else {
24                     return Or("", sum);
25                 }
26             } else {
27                 return sum;
28             }
29         case { qk | restAllowed }:
30             rkp1p2 := rpq(p1, p2, sigma, delta, restAllowed);
31             rkp1qk := rpq(p1, qk, sigma, delta, restAllowed);
32             rkqkqk := rpq(qk, qk, sigma, delta, restAllowed);
33             rkqkp2 := rpq(qk, p2, sigma, delta, restAllowed);
34             return Or(rkp1p2, Cat(Cat(rkp1qk, Star(rkqkqk)), rkqkp2));
35     }
36 };

```

Abbildung 4.23: Converting a DFA into a regular expression.

- (a) The first two arguments p_1 and p_2 are states of a finite state machine. The idea is that the call

$\text{rpq}(p_1, p_2, \text{sigma}, \text{delta}, \text{states})$

computes the regular expression $r(p_1, p_2)$, which is the expression that describes the set of those strings s that take the finite state machine from the state p to the state q .

- (b) **sigma** is the alphabet of the finite state machine.
(c) **delta** is the transition function of the finite state machine.
(d) **allowed** is the set of all states that are allowed in the transition of the fsm from the state p_1 into the state p_2 .

The function **rpq** is defined by recursion on its last argument.

- (a) The base case of the recursion is the case where the set **allowed** is empty. Then the regular expression

returned by the function `rpq` must only specify those strings that take the fsm from state p_1 to state p_2 without visiting any other state in between. In line 18, the function computes the set of all characters c that take the fsm from the state p_1 into the state p_2 directly, i.e. that satisfy

$$\delta(p_1, c) = p_2.$$

If this set is given as $\{c_1, \dots, c_k\}$, then the variable `sum` gets the value

$$c_1 + \dots + c_k.$$

Of course, if the set $\{c_1, \dots, c_k\}$ is empty, the sum $c_1 + \dots + c_k$ has to be interpreted as the regular expression \emptyset . Now there are two cases:

- i. $p_1 = p_2$. In this case, the empty string ε transforms the state p_1 into p_2 and therefore in this case the result returned is

$$c_1 + \dots + c_k + \varepsilon.$$

- ii. $p_1 \neq p_2$. Then the result is given as

$$c_1 + \dots + c_k.$$

- (b) In the recursive case, we arbitrarily pick a state q_k from the set `allowed` of states that may be used to move from state p_1 to p_2 . The state is removed from the set `allowed` to produce the set `restAllowed`. Then, the regular expression returned has the form

$$r(p_1, p_2) + r(p_1, q_k) \cdot (r(q_k, q_k))^* \cdot r(q_k, p_2).$$

Here, $r(p_1, p_2)$ is computed recursively as the regular expression that takes the fsm from the state p_1 to the state p_2 using only the states from the set `restAllowed`. The regular expressions $r(p_1, q_k)$, $r(q_k, q_k)$, and $r(q_k, p_2)$ are defined in a similar way. The reasoning for returning this result is as follows: In order to get from state p_1 to state p_2 , there are two possibilities:

- If we do not need to visit the state q_k when transforming the fsm from state p_1 into state p_2 , then the regular expression

$$r(p_1, p_2)$$

already describes the transition.

- If we do need to visit the state q_k , then we move from p_1 to p_2 in three steps:
 - We move from p_1 to q_k .
 - Next, we can move from q_k to q_k as many times as we wish.
 - Finally, we move from q_k to p_2 .

These three steps are summarized by the regular expression

$$r(p_1, q_k) \cdot (r(q_k, q_k))^* \cdot r(q_k, p_2).$$

Historisches Stephen C. Kleene (1909 – 1994) hat 1956 gezeigt, dass endliche Automaten und reguläre Ausdrücke gleich mächtig sind [Kle56]: Wir haben in diesem Kapitel gesehen, wie wir zu jedem regulären Ausdruck r einen endlichen Automaten konstruieren können, der genau die Strings akzeptiert, die durch den regulären Ausdruck r beschrieben werden. Umgekehrt gilt, dass es zu jedem endlichen Automaten einen äquivalenten regulären Ausdruck gibt.

Kapitel 5

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 nehmen also an, dass der Ausdruck $\delta(q, c)$ für jeden Zustand $q \in Q$ und jeden Buchstaben $c \in \Sigma$ als Ergebnis einen Zustand aus Q liefert. 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 durchführen zu können, müssen wir etwas ausholen. Zunächst erweitern wir die Funktion

$$\delta : Q \times \Sigma \rightarrow Q$$

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

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

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

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

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

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 betrachteten endlichen Automaten vom Start-Zustand aus erreichbar sind.

Im Allgemeinen können wir einen Automaten dadurch minimieren, dass wir bestimmte Zustände identifizieren. Betrachten wir beispielsweise den in Abbildung 5.1 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 dürfen und einfach alle anderen Zustände als äquivalent zu betrachten.

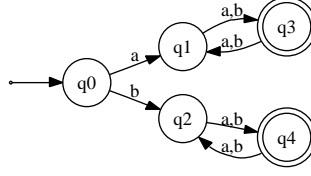


Abbildung 5.1: Ein endlicher Automat mit äquivalenten Zuständen.

Definition 13 (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 unterscheidbar sind. Die Behauptung ist nun die, dass wir alle solchen Zustände identifizieren können. Die Identifikation zweier Zustände p_1 und p_2 erfolgt dadurch, 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

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

gilt. Diese Idee liefert einen Algorithmus, der aus zwei Schritten besteht:

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:

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

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 Zustände $\delta(p, c)$ und $\delta(q, c)$ bereits unterscheidbar sind, fügen wir dieses Paar zur Menge V hinzu:

```

while (  $\exists \langle p, q \rangle \in Q \times Q : \exists c \in \Sigma : \langle \delta(p, c), \delta(q, c) \rangle \in V \wedge \langle p, q \rangle \notin V$  ) {
    choose  $\langle p, q \rangle \in Q \times Q$  such that  $\langle \delta(p, c), \delta(q, c) \rangle \in V \wedge \langle p, q \rangle \notin V$  {
         $V := V \cup \{ \langle p, q \rangle, \langle q, p \rangle \}$ ;
    }
}
    
```

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 5.1 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 q_4 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		\times	\times	\times	\times
q_1	\times			\times	\times
q_2	\times			\times	\times
q_3	\times	\times	\times		
q_4	\times	\times	\times		

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

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

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

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

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

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

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:

$$(a) \quad q_1 \sim q_2$$

$$(b) \quad q_3 \sim q_4$$

Abbildung 5.2 zeigt den entsprechenden reduzierten endlichen Automaten.

Aufgabe 12: Konstruieren Sie den minimalen deterministischen endlichen Automaten, der die Sprache $L(a \cdot (b \cdot a)^*)$ erkennt. Gehen Sie dazu in folgenden Schritten vor:

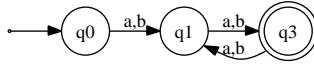


Abbildung 5.2: Der reduzierte endliche Automat.

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.

5.1 Implementing the Minimization of Finite Automata in SetlX

Figure 5.3 on page 63 shows the function `minimize` that takes a deterministic finite state machine `fa` as input. The function eliminates all states from `fa` that are not reachable from the start state and then tries to minimize equivalent states as discussed in the previous section. It returns a finite state machine that accepts the same language as `fa` and that, furthermore, is guaranteed to have as few states as possible. The implementation works as discussed below.

```

1  minimize := procedure(fa) {
2      [states, sigma, delta, q0, accepting] := fa;
3      reachable := { q0 };
4      while (true) {
5          newFound := { delta(q, c) : q in reachable, c in sigma };
6          if (newFound <= reachable) { break; }
7          reachable += newFound;
8      }
9      separable := (states-accepting) >< accepting + accepting >< (states-accepting);
10     while (true) {
11         newSep := { [q1, q2] in states >< states
12                     | exists (c in sigma | [delta(q1, c), delta(q2, c)] in separable)
13                     };
14         if (newSep <= separable) { break; }
15         separable += newSep;
16     }
17     equivalent := states >< states - separable;
18     equivClasses := { { p in states | [p, q] in equivalent } : q in states };
19     newQ0 := { m in equivClasses | q0 in m };
20     newAccept := { m in equivClasses | arb(m) in accepting };
21     newDelta := closure(
22         "procedure(m, c) {
23             d := $delta$;
24             q := d(arb(m), c);
25             return arb({ x : x in $equivClasses$ | q in x });
26         }");
27     return [equivClasses, sigma, newDelta, newQ0, newAccept];
28 };
```

Abbildung 5.3: Minimizing a finite state machine.

1. First, all states *reachable* from the start state q_0 are computed. Here, a state p is *reachable* from the state q_0 iff there is a string s such that $\delta(q_0, s) = p$.
 - (a) Since $\delta(q_0, \varepsilon) = q_0$, the state q_0 is reachable from q_0 and therefore we initialize the set **reachable** with the start state q_0 .
 - (b) The set **newFound** is the set of all states that can be reached by reading one character in a state that is already known to be reachable.
 - (c) If the set **newFound** does not contain any states that have not already been seen, we have found all reachable states. This is checked in line 6.
 - (d) If the test in line 6 instead shows that we have found new reachable states, then these states are added to the set of known reachable states in line 7.
2. Next, we try to find all pairs of states that are *separable*. Here, a pair $\langle p, q \rangle$ is called *separable* if there is a string s such that either $\delta(p, s)$ is accepting while $\delta(q, s)$ is not accepting, or $\delta(p, s)$ is not accepting while $\delta(q, s)$ is accepting. Given two states p and q we call p and q separable iff the pair $\langle p, q \rangle$ is separable.

- (a) Initially, we know that a pair $\langle p, q \rangle$ is separable if either p is a member of the set **accepting** of accepting states while q is not a member of **accepting** or it is the other way around: $p \notin \text{accepting}$ and $q \in \text{accepting}$.

Therefore, the set **separable** is initialized as the set

$$(\text{states} \setminus \text{accepting}) \times \text{accepting} \cup \text{accepting} \times (\text{states} \setminus \text{accepting}).$$

This expression is coded in SETLX in line 9. Note that the set difference $a \setminus b$ of two sets a and b is written as $a - b$ in SETLX, while the *cartesian product* $a \times b$ of a and b is written as $a > < b$. Remember that $a \times b$ is defined as

$$a \times b := \{ \langle x, y \rangle \mid x \in a \wedge y \in b \}.$$

- (b) Next, if the states $\delta(q_1, c)$ and $\delta(q_2, c)$ are already known to be separable, then the states q_1 and q_2 are also separable. The reasoning is as follows: As $\delta(q_1, c)$ and $\delta(q_2, c)$ are separable, there is a string s such that

$$\delta(\delta(q_1, c), s) \text{ is accepting} \quad \text{while} \quad \delta(\delta(q_2, c), s) \text{ is not accepting}$$

or the other way around. As we have $\delta(\delta(q_1, c), s) = \delta(q_1, cs)$ and $\delta(\delta(q_2, c), s) = \delta(q_2, cs)$ this can be rewritten as

$$\delta(q_1, cs) \text{ is accepting} \quad \text{while} \quad \delta(q_2, cs) \text{ is not accepting}$$

or the other way around. By the definition of two states being separable this implies that q_1 and q_2 are separable. This explains the definition of the variable **newSep** in line 11. If we do find new pairs of separable states, these are added to the set of pairs already known to be separable. This process is repeated until we do not find any more separable pairs.

3. Next, we have to identify those states that are *equivalent*: Two states p and q are called *equivalent* if and only if they are not separable.

There are several way to identify equivalent states. The easiest way is to collect the associated *equivalence classes*, where an equivalence class contains all thoses states that are equivalent to each other.

- (a) Therefore, the set of states of the minimized finite state machine is the set of equivalence classes of states of the given finite state machine **fa**. For example, if the set of states of **fa** is

$$\{ q_0, q_1, q_2, q_3, q_4, q_5 \}$$

and the state q_1 is equivalent to q_2 and, furthermore, the states q_3, q_4 , and q_5 are pairwise equivalent, then the set of equivalence classes is given as

$$\{ \{q_0\}, \{q_1, q_2\}, \{q_3, q_4, q_5\} \}.$$

This set of equivalence classes is the new set of states.

- (b) Of course, the new start state is the set of equivalent states that contains the start state q_0 of the given finite state machine \mathbf{fa} .
- (c) A set of equivalent states is accepting if any of its member states is an accepting state. Of course, if a set of equivalent states contains an accepting state, then the other states in this equivalence class have to be accepting also, since otherwise these states would be separable from the accepting state and therefore could not be equivalent.
- (d) In order to compute the new state transition function, we have to construct a function that takes a set of equivalent states of the old finite state machine \mathbf{fa} and turns this set into a new set of states that are again equivalent. If M is a set of equivalent states and c is a character from Σ , in order to define $\Delta(M, c)$ we proceed as follows:
 - We pick an arbitrary state q from M and compute $\delta(q, c)$.
 - We return the set of all states that are equivalent to the state $\delta(q, c)$.

We have to check that this is well defined and does not depend on the choice of the state q . The reasoning is as follows: If both q_1 and q_2 are elements of M , then these two states have to be equivalent. But then the states $\delta(q_1, c)$ and $\delta(q_2, c)$ have to be equivalent also. After all, if $\delta(q_1, c)$ and $\delta(q_2, c)$ are separable, then q_1 and q_2 are separable too and therefore can not be equivalent.

In order to ease our notation, let us write $p_1 \sim p_2$ if the states p_1 and p_2 are equivalent. Then $\Delta(M, c)$ can be defined as follows:

$$\Delta(M, c) = \{p \mid p \sim \delta(\mathbf{arb}(M), c)\}.$$

Here, $\mathbf{arb}(M)$ returns an arbitrary state from the set M . This definition of $\Delta(M, c)$ is implemented in the definition of `newDelta`. However, instead of checking whether p is equivalent to $\delta(\mathbf{arb}(M), c)$ we pick the equivalence class x that contains $\delta(\mathbf{arb}(M), c)$ since this equivalence class will automatically contain all those states p that are equivalent to $\delta(\mathbf{arb}(M), c)$.

5.2 The Theorem of Nerode

A language is called a *regular* language iff there is a finite state machine A recognizing the language. We have already seen that a language is regular iff it is accepted by a finite state machine. In this section we discuss a theorem that can be used to prove that a given language is not a regular language. The main idea is to extend the notion of *separability* from states to strings.

Definition 14 (separable) Given an alphabet Σ and a formal language $L \subseteq \Sigma^*$, a pair $\langle s, t \rangle \in \Sigma^* \times \Sigma^*$ is called *separable with respect to L* iff there is a string $w \in \Sigma^*$ such that

$$(sw \in L \wedge tw \notin L) \vee (sw \notin L \wedge tw \in L).$$

In this case, w is the *witness of the separability* of $\langle s, t \rangle$. If the pair $\langle s, t \rangle$ is separable with respect to L and if the language L is obvious from the context, then in order to shorten our notation, we call s and t separable. \square

Example: Take $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and define L as the language of all strings of the form $a^n b^n$ where n is a natural number, i. e. define

$$L := \{\mathbf{a}^n \mathbf{b}^n \mid n \in \mathbb{N}\}.$$

Then the strings $s := \mathbf{aaab}$ and $t := \mathbf{bba}$ are separable and $w := \mathbf{bb}$ is a witness of separability because

$$sw = \mathbf{aaabbb} \in L \quad \text{but} \quad \mathbf{bbabb} \notin L. \quad \diamond$$

Exercise 13: Assume Σ is an alphabet and $L \subseteq \Sigma^*$ is a formal language. Define a relation \sim_L on Σ^* as follows:

$$s_1 \sim_L s_2 \quad \text{iff} \quad s_1 \text{ and } s_2 \text{ are } \underline{\text{not}} \text{ separable with respect to } L.$$

Prove that the relation \sim_L is an equivalence relation! \diamond

The following Theorem has been proven by Anil Nerode in 1958 [Ner58]. It can be used to show that certain languages are not regular.

Theorem 15 (Nerode) If $L \subseteq \Sigma^*$ is a formal language and $S = \{s_1, \dots, s_n\} \subseteq \Sigma^*$ is a set of strings that are pairwise separable and, furthermore, A is a deterministic finite state machine recognizing the language L , then A has at least n different states.

Proof: Assume $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ is a DFA accepting the language L , i. e. $L(A) = L$. Define states

$$q_i := \delta(q_0, s_i) \quad \text{for all } i \in \{1, \dots, n\}.$$

The claim is that all these states are pairwise different, that is we have

$$\forall i, j \in \{1, \dots, n\} : i \neq j \rightarrow q_i \neq q_j.$$

Therefore, assume that we have $i \neq j$. By our assumption, the states s_i and s_j are separable. Then there is a witness $w \in \Sigma^*$ such that

$$(s_i w \in L \wedge s_j w \notin L) \vee (s_i w \notin L \wedge s_j w \in L).$$

These are two cases which we consider separately.

1. $s_i w \in L \wedge s_j w \notin L$.

Since the DFA A accepts the language L , we know that

$$\delta(q_0, s_i w) \in F \wedge \delta(q_0, s_j w) \notin F.$$

On the other hand, we have

$$\delta(q_0, s_i w) = \delta(\delta(q_0, s_i), w) = \delta(q_i, w) \quad \text{and} \quad \delta(q_0, s_j w) = \delta(\delta(q_0, s_j), w) = \delta(q_j, w).$$

From this we can conclude

$$\delta(q_i, w) \in F \wedge \delta(q_j, w) \notin F.$$

This is only possible if $q_i \neq q_j$.

2. $s_i w \notin L \wedge s_j w \in L$.

If we exchange the roles of i and j , this case is reduced to the previous case and we can again conclude that $q_i \neq q_j$.

We have just shown that $q_i \neq q_j$ as long as $i \neq j$. Therefore the states $\{q_1, \dots, q_n\}$ are all pairwise different and the finite state machine A needs to have at least n different states. \square

Example: We prove that the language

$$L := \{a^k b^k \mid k \in \mathbb{N}\}$$

is not regular. The proof is done by contradiction. Assume L is regular. Then there is a DFA

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

that recognizes L . Next, pick an arbitrary natural number n and consider the following set of strings:

$$S := \{a^1, a^2, \dots, a^n\}.$$

S contains n strings and we claim that these strings are all pairwise separable. In order to see this, take $i, j \in \{1, \dots, n\}$ such that $i \neq j$ and consider the strings a^i and a^j . The witness b^i separates these strings because

$$a^i b^i \in L \quad \text{but} \quad a^j b^i \notin L$$

since $j \neq i$. The theorem of Nerode shows that the DFA A needs to have at least n states. However, the number n is arbitrary! We can pick it as 1, 2, 100, or even 2^{100} . Therefore, the set of states of A can not be finite and we conclude that there is no finite state machine A such that $L = L(A)$. \square

Exercise 14: Prove that the language

$$\{\mathbf{a}^{k^2} \mid k \in \mathbb{N}\}$$

is not regular. ◇

Exercise 15: Prove that the language

$$\{\mathbf{a}^p \mid p \in \mathbb{N} \wedge p \text{ is prime}\}$$

is not regular.

Hint: It is known that there the number of primes is infinite and that there are gaps of arbitrary size between the prime numbers, so given an arbitrary natural number k , there is a pair of primes $\langle p_1, p_2 \rangle$ such

$$p_1 + k < p_2$$

and none of the natural numbers between p_1 and p_2 is prime. ◇

Kapitel 6

Die Theorie regulärer Sprachen

Ist Σ ein Alphabet, so bezeichnen 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 $\Sigma^* \setminus L$ einer regulären Sprache ist wieder eine reguläre Sprache.

Als Anwendung der Abschluss-Eigenschaften zeigen wir anschließend, wie die Äquivalenz zweier regulärer Ausdrücke 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

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

nicht regulär ist.

6.1 Abschluss-Eigenschaften regulärer Sprachen

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

Satz 16 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 17 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 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

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

durch die Gleichung

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

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:

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

Damit gilt für alle $s \in \Sigma^*$:

$$\begin{aligned} & s \in L(A) \\ \Leftrightarrow & \delta(\langle q_1, q_2 \rangle, s) \in F \\ \Leftrightarrow & \langle \delta_1(q_1, s), \delta_2(q_2, s) \rangle \in F_1 \times F_2 \\ \Leftrightarrow & \delta_1(q_1, s) \in F_1 \wedge \delta_2(q_2, s) \in F_2 \\ \Leftrightarrow & s \in L(A_1) \wedge s \in L(A_2) \\ \Leftrightarrow & s \in L(A_1) \cap L(A_2) \\ \Leftrightarrow & s \in L_1 \cap L_2 \end{aligned}$$

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

$$\wedge : \text{RegExp} \times \text{RegExp} \rightarrow \text{RegExp}$$

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 18 Ist L eine reguläre Sprache über dem Alphabet Σ , so ist auch das Komplement von L , die Sprache $\Sigma^* \setminus L$ 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

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

Offenbar gilt

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

und daraus folgt die Behauptung. \square

Korollar 19 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Mengen-Differenz $L_1 \setminus L_2$ eine reguläre Sprache.

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

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

denn ein Wort w ist genau dann in $L_1 \setminus L_2$, 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 $\Sigma^* \setminus L_2$ regulär ist. Da der Durchschnitt zweier regulärer Sprachen wieder regulär ist, ist damit auch $L_1 \setminus L_2$ regulär. \square

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

6.2 Erkennung leerer Sprachen

In diesem Abschnitt untersuchen wir für einen gegebenen 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 iterativ erfolgen:

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

Dieser Schritt wird solange wiederholt, bis wir der Menge R keine neuen Zustände mehr hinzufügen können.

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.

Bemerkung: Ist die reguläre Sprache L nicht durch einen endlichen Automaten A , sondern durch einen regulären Ausdruck r gegeben, so läßt sich durch einen einfachen rekursiven Algorithmus, der dem Aufbau des regulären Ausdrucks folgt, entscheiden, ob $L(r)$ leer ist.

1. $L(\emptyset) = \{\}$.
2. $L(\varepsilon) \neq \{\}$.
3. $L(c) \neq \{\}$ für alle $c \in \Sigma$.
4. $L(r_1 \cdot r_2) = \{\} \Leftrightarrow L(r_1) = \{\} \vee L(r_2) = \{\}$.
5. $L(r_1 + r_2) = \{\} \Leftrightarrow L(r_1) = \{\} \wedge L(r_2) = \{\}$.
6. $L(r^*) \neq \{\}$.

6.3 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:

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

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 20 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 $L(r_2) \setminus L(r_1)$ und $L(r_1) \setminus L(r_2)$ beide verschwinden, denn es gilt:

$$\begin{aligned} L(r_1) = L(r_2) &\Leftrightarrow L(r_1) \subseteq L(r_2) \wedge L(r_2) \subseteq L(r_1) \\ &\Leftrightarrow L(r_1) \setminus L(r_2) = \{\} \wedge L(r_2) \setminus L(r_1) = \{\} \end{aligned}$$

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 19 sind die Sprachen $L(r_1) \setminus L(r_2)$ und $L(r_2) \setminus L(r_1)$ regulär und wir haben gesehen, wie wir endliche Automaten A_{12} und A_{21} so konstruieren können, dass

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

gilt. Damit gilt nun

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

und diese Frage ist nach Abschnitt 6.2 entscheidbar. □

6.4 Implementation in SetlX

Figure 6.1 on page 72 shows how to implement the algorithm sketched in the previous section. The details are discussed below.

```

1  regExpEquiv := procedure(r1, r2, sigma) {
2      fsm1 := regexp2DFA(r1, sigma);
3      fsm2 := regexp2DFA(r2, sigma);
4      r1MinusR2 := fsmComplement(fsm1, fsm2);
5      r2MinusR1 := fsmComplement(fsm2, fsm1);
6      return isEmpty(r1MinusR2) && isEmpty(r2MinusR1);
7  };
8  regexp2DFA := procedure(r, sigma) {
9      nfa := regexp2NFA(r, sigma);
10     return nfa2dfa(nfa);
11 };
12 fsmComplement := procedure(f1, f2) {
13     [states1, sigma, delta1, q1, a1] := f1;
14     [states2, _, delta2, q2, a2] := f2;
15     states := states1 >< states2;
16     delta := closure("procedure(q, c) {
17         [q1, q2] := q;
18         d1 := $delta1$;
19         d2 := $delta2$;
20         return [d1(q1, c), d2(q2, c)];
21     }");
22     return [states, sigma, delta, [q1, q2], a1 >< (states2 - a2)];
23 };
24 isEmpty := procedure(fsm) {
25     [states, sigma, delta, q0, accepting] := fsm;
26     reachable := { q0 };
27     while (true) {
28         newFound := { delta(q, c) : q in reachable, c in sigma };
29         if (newFound <= reachable) { break; }
30         reachable += newFound;
31     }
32     return reachable * accepting == {};
33 };

```

Abbildung 6.1: An algorithm to compare regular expressions.

1. The function `regExpEquiv` is called with three arguments:

- (a) `r1` and `r2` are regular expressions that have to be compared. These regular expressions are represented as terms in the same way as in Figure 4.14 in the definition of the function `regExp2NFA`.
- (b) `sigma` is the alphabet.

The implementation of `regExpEquiv` is straightforward:

- (a) `r1` and `r2` are converted into deterministic finite state machines `fsm1` and `fsm2`, respectively.
- (b) Next, we construct the finite state machines `r1MinusR2` and `r2MinusR1`.
`r1MinusR2` accepts all strings that are accepted by `fsm1` but are rejected by `fsm2`, while `r2MinusR1` accepts all strings that are accepted by `fsm2` but are rejected by `fsm1`. Therefore

$$L(\mathbf{r1MinusR2}) = L(r_1) \setminus L(r_2) \quad \text{and} \quad L(\mathbf{r2MinusR1}) = L(r_2) \setminus L(r_1).$$

- (c) The regular expressions `r1` and `r2` are equivalent iff

$$L(r_1) \subseteq L(r_2) \quad \text{and} \quad L(r_2) \subseteq L(r_1)$$

holds. This is the case if and only if

$$L(r_1) \setminus L(r_2) = \{\} \quad \text{and} \quad L(r_2) \setminus L(r_1) = \{\}$$

and these conditions are checked using the function `isEmpty`.

2. The function `regExp2DFA` takes a regular expression `r` together with an alphabet `sigma` and constructs a deterministic finite state machine that accepts the language described by `r`. This is done by first converting `r` into a non-deterministic finite state machine `nfa` via the function `regExp2NFA`. The non-deterministic finite state machine `nfa` is then converted into a deterministic finite state machine with the help of the function `nfa2dfa`.

The function `regExp2NFA` has already been shown in Figure 4.14 on page 48, while the function `nfa2dfa` is shown in Figure 4.7 on page 43.

3. The function `fsmComplement` has two arguments `f1` and `f2`. These arguments are deterministic finite state machines. The function returns a new finite state machine `F` that accepts the language

$$L(f_1) \setminus L(f_2).$$

The finite state machine `F` simulates the two finite state machines `f1` and `f2` in parallel. Therefore, the states of `F` are pairs of the form $\langle p_1, p_2 \rangle$ where p_1 is a state of `f1` while p_2 is a state of `f2`. The transition function δ of `F` is a composition of the transition function δ_1 of `f1` and δ_2 of `f2` that is defined as follows:

$$\delta(\langle q_1, q_2 \rangle, c) := \langle \delta_1(q_1, c), \delta_2(q_2, c) \rangle.$$

A state $\langle p_1, p_2 \rangle$ is accepting iff p_1 is an accepting state of `f1` but p_2 is not an accepting state of `f2`.

4. The input of the function `isEmpty` is a deterministic finite state machine `fsm`. The function checks whether the language accepted by `fsm` is the empty set. To this end, it computes the set of all states that are reachable from the start state. If any of these states is accepting, then the language of `fsm` is not empty.

6.5 Grenzen regulärer Sprachen

Wir haben im letzten Kapitel bereits das Theorem von Nerode kennengelernt und konnten mit Hilfe dieses Theorems zeigen, dass bestimmte Sprachen nicht regulär sind. Wir werden nun ein weiteres Theorem kennenlernen, das ebenfalls genutzt werden kann um nachzuweisen, dass eine gegebene Sprache nicht regulär ist.

Theorem 21 (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 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:

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

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:

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

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, \dots, c_m die einzelnen Buchstaben sind. Wir betrachten die Berechnung, die der Automat A bei der Eingabe s durchführt. Diese Berechnung hat die Form

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

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

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

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

$$\text{card}(\{0, 1, \dots, n\}) = n + 1$$

wissen wir, dass schon in der Liste

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

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

$$q_0 \xrightarrow{u} q_k. \tag{6.1}$$

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

$$q_k \xrightarrow{v} q_k. \tag{6.2}$$

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

$$q_k \xrightarrow{w} q_m. \tag{6.3}$$

Aus $q_k \xrightarrow{v} q_k$ folgt

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

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

$$q_k \xrightarrow{v^h} q_k \tag{6.4}$$

Aus den Gleichungen (6.1), (6.3) und 6.4) folgt nun

$$q_0 \xrightarrow{uv^h w} q_m$$

und da q_m ein akzeptierender Zustand ist, haben wir damit $uv^h w \in L$ gezeigt. \square

Das Pumping-Lemma kann benutzt werden um nachzuweisen, dass bestimmte Sprachen nicht regulär sind. Der nächste Satz gibt ein Beispiel.

Satz 22 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:

$$L = \{ ({}^k)^k \mid k \in \mathbb{N} \}$$

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 feste Zahl n , so dass sich alle Strings $s \in L$, für die $|s| \geq n$ 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. Mit Hilfe der durch das Pumping-Lemma gegebenen festen Zahl n definieren wir den String s als

$$s := ({}^n)^n.$$

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

$$uw \in L \tag{6.5}$$

gilt. Um den Beweis besser formalisieren zu können, führen wir eine Funktion

$$\text{count} : \Sigma^* \times \Sigma \rightarrow \mathbb{N}$$

ein. Für einen String t und einen Buchstaben c soll $\text{count}(t, c)$ zählen, wie oft der Buchstabe c in dem String t vorkommt. Für die Sprache L gilt offenbar

$$t \in L \Rightarrow \text{count}(t, "(") = \text{count}(t, ")").$$

Einerseits haben wir nun

$$\begin{aligned} \text{count}(uw, "(") &= \text{count}(uvw, "(") - \text{count}(v, "(") \\ &= \text{count}(s, "(") - \text{count}(v, "(") \\ &= \text{count}(({}^n)^n, "(") - \text{count}(({}^k, "(") \\ &= n - k \\ &< n \end{aligned}$$

Andererseits gilt

$$\begin{aligned}
 \text{count}(uw, "(") &= \text{count}(uvw, "(") - \text{count}(v, "(") \\
 &= \text{count}(s, "(") - \text{count}(v, "(") \\
 &= \text{count}(\binom{n}{n}, "(") - \text{count}(\binom{k}{k}, "(") \\
 &= n
 \end{aligned}$$

Insgesamt sehen wir

$$\text{count}(uw, "(") < \text{count}(uw, ")")$$

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. \square

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 wesentlich mächtiger als das Konzept der regulären Sprachen ist. Mit diesem Konzept wird es dann möglich sein, die meisten Programmier-Sprachen zu beschreiben.

Aufgabe 16: Die Sprache L_{square} beinhaltet alle Wörter der Form a^n für die n eine Quadrat-Zahl ist, es gilt also

$$L_{\text{square}} = \{a^m \mid \exists k \in \mathbb{N} : m = k^2\}$$

Zeigen Sie mit Hilfe des Pumping-Lemmas, dass die Sprache L_{square} keine reguläre Sprache ist.

Hinweis: Nutzen Sie aus, dass der Abstand zwischen den Quadrat-Zahlen beliebig groß wird. \diamond

Exercise 17: The language L is defined as

$$L := \{a^m b^n c^n \mid m, n \in \mathbb{N}\} \cup \{b^m c^n \mid m, n \in \mathbb{N}\}.$$

(a) Prove that L is not regular.

(b) Prove that L satisfies the pumping lemma. \diamond

Historical Remark The pumping lemma is a special case of a general theorem that has been proved by Bar-Hillel, Perles and Shamir [BHPS61].

Kapitel 7

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.

7.1 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 “(”, “[”, “{” oder die entsprechenden schließenden Klammern,
- vordefinierte Schlüsselwörter wie “if”, “while”, “typedef”, “struct”, 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 Mengen ist notwendig, um später die Präzedenzen der Operatoren korrekt darstellen zu können. Die Grundbausteine der arithmetischen Ausdrücke sind Variablen, Zahlen, die Operator-Symbole “+”, “-”, “*”, “/”, und die Klammer-Symbole “(” und “)”. Aufbauend auf diesen Symbolen verläuft die induktive Definition der Mengen *Factor*, *Product* und *ArithExpr* wie folgt:

1. Jede Zahlenkonstante ist ein Faktor:

$$C \in \text{Number} \Rightarrow C \in \text{Factor}.$$

2. Jede Variable ist ein Faktor:

$$V \in \text{Variable} \Rightarrow V \in \text{Factor}.$$

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:

$$A \in \text{ArithExpr} \Rightarrow "(" A ")" \in \text{Factor}.$$

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:

$$F \in \text{Factor} \Rightarrow F \in \text{Product}.$$

5. Ist P ein Produkt und ist F ein Faktor, so sind die Strings $P "*" F$ und $P "/" F$ ebenfalls Produkte:

$$P \in \text{Product} \wedge F \in \text{Factor} \Rightarrow P "*" F \in \text{Product} \wedge P "/" F \in \text{Product}.$$

6. Jedes Produkt ist gleichzeitig auch ein arithmetischer Ausdruck

$$P \in \text{Product} \Rightarrow P \in \text{ArithExpr}.$$

7. Ist A ein arithmetischer Ausdruck und ist P ein Produkt, so sind auch die Strings $A "+" P$ und $A "-" P$ arithmetische Ausdrücke:

$$A \in \text{ArithExpr} \wedge P \in \text{Product} \Rightarrow A "+" P \in \text{ArithExpr} \wedge A "-" P \in \text{ArithExpr}.$$

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:

$$\text{ArithExpr} \rightarrow \text{ArithExpr} "+" \text{Product}$$

$$\text{ArithExpr} \rightarrow \text{ArithExpr} "-" \text{Product}$$

$$\text{ArithExpr} \rightarrow \text{Product}$$

$$\text{Product} \rightarrow \text{Product} "*" \text{Factor}$$

$$\text{Product} \rightarrow \text{Product} "/" \text{Factor}$$

$$\text{Product} \rightarrow \text{Factor}$$

$$\text{Factor} \rightarrow "(" \text{ArithExpr} ")"$$

$$\text{Factor} \rightarrow \text{Variable}$$

$$\text{Factor} \rightarrow \text{Number}$$

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 syntaktischen Variablen. 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-Terminale stehen für sich selbst.
2. Token wie *Number* oder *Variable* ist zusätzlich ein Wert zugeordnet. Im Falle von *Number* ist dies eine Zahl, im Falle von *Variable* ist dies ein String, der den Namen der Variablen wiedergibt.

Üblicherweise werden Grammatik-Regeln in einer kompakteren Notation als der oben vorgestellten wiedergegeben, indem alle Regeln für ein Nicht-Terminal zusammengefaßt werden. Für unser Beispiel sieht das dann wie folgt aus:

$$\begin{aligned} \text{ArithExpr} &\rightarrow \text{ArithExpr} \text{ "+" } \text{Product} \mid \text{ArithExpr} \text{ "-" } \text{Product} \mid \text{Product} \\ \text{Product} &\rightarrow \text{Product} \text{ "*" } \text{Factor} \mid \text{Product} \text{ "/" } \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow \text{"(" ArithExpr "("} \mid \text{Number} \end{aligned}$$

Hier werden also die einzelnen Alternativen einer Regel durch das Metazeichen \mid getrennt. Nach dem obigen Beispiel geben wir jetzt die formale Definition für den Begriff der kontextfreien Grammatik.

Definition 23 (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 *syntaktische Variablen* oder auch *Nicht-Terminal* bezeichnen.

In dem obigen Beispiel gilt

$$V = \{\text{ArithExpr}, \text{Product}, \text{Factor}\}.$$

2. T ist eine Menge von Namen, die wir als *Terminal* bezeichnen. Die Mengen T und V sind disjunkt, es gilt also

$$T \cap V = \emptyset. \text{ In dem obigen Beispiel gilt}$$

$$T = \{\text{Number}, \text{"+"}, \text{"-"}, \text{"*"}, \text{"/"}, \text{"("}, \text{")"}\}$$

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

$$\text{ArithExpr} \rightarrow \text{ArithExpr} \text{ "+" } \text{Product}$$

geschrieben. Formal steht diese Regel für das Paar

$$\langle \text{ArithExpr}, [\text{ArithExpr}, \text{"+"}, \text{Product}] \rangle.$$

4. S ist ein Element der Menge V , das wir als das *Start-Symbol* bezeichnen. In dem obigen Beispiel ist *ArithExpr* das Start-Symbol. \square

7.1.1 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. $(A \rightarrow \gamma) \in R$ 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 $A \rightarrow \gamma$. 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:

$$\begin{aligned} \text{ArithExpr} &\Rightarrow \text{ArithExpr} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Product} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Product} \text{ "*" } \text{Factor} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Factor} \text{ "*" } \text{Factor} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Number} \text{ "*" } \text{Factor} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Number} \text{ "*" } \text{Number} \text{ "+" } \text{Product} \\ &\Rightarrow \text{Number} \text{ "*" } \text{Number} \text{ "+" } \text{Factor} \\ &\Rightarrow \text{Number} \text{ "*" } \text{Number} \text{ "+" } \text{Number} \end{aligned}$$

Damit haben wir also gezeigt, dass

$$\text{ArithExpr} \Rightarrow^* \text{Number} \text{ "*" } \text{Number} \text{ "+" } \text{Number}$$

oder genauer

$$\text{ArithExpr} \Rightarrow^8 \text{Number} \text{ "*" } \text{Number} \text{ "+" } \text{Number}$$

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:

$$L(G) := \{w \in T^* \mid S \Rightarrow^* w\}.$$

Beispiel: Die Sprache

$$L = \{(n)^n \mid n \in \mathbb{N}\}$$

wird von der Grammatik

$$G = \langle \{S\}, \{“(”, “)”\}, R, S \rangle$$

erzeugt, wobei die Regeln R wie folgt gegeben sind:

$$\begin{array}{lcl} S & \rightarrow & “(” S “)” \\ & | & \varepsilon \end{array}$$

Beweis: Wir zeigen zunächst, dass sich jedes Wort $w \in L$ aus dem Start-Symbol S ableiten lässt:

$$w \in L \implies S \Rightarrow^* w.$$

Es sei also $w_n = ({}^n)^n$. Wir zeigen durch Induktion über n , dass $w_n \in L(G)$ ist.

I.A.: $n = 0$.

Es gilt $w_0 = \varepsilon$. Offenbar haben wir

$$S \Rightarrow \varepsilon,$$

denn die Grammatik enthält die Regel $S \rightarrow \varepsilon$. Also gilt $w_0 \in L(G)$.

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

Der String w_{n+1} hat die Form $w_{n+1} = "(" w_n ")"$, wobei der String w_n natürlich ebenfalls in L liegt. Also gibt es nach I.V. eine Ableitung von w_n :

$$S \Rightarrow^* w_n.$$

Insgesamt haben wir dann die Ableitung

$$S \Rightarrow "(" S ")" \Rightarrow^* "(" w_n ")" = w_{n+1}.$$

Also gilt $w_{n+1} \in L(G)$.

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

$$S \Rightarrow \varepsilon.$$

Folglich 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:

$$S \Rightarrow "(" S ")" \Rightarrow^n w$$

Daraus folgt

$$w = "(" v ")" \wedge S \Rightarrow^n v.$$

Nach I.V. gilt dann $v \in L$. Damit gibt es $k \in \mathbb{N}$ mit $v = ({}^k)^k$. Also haben wir

$$w = "(" v ")" = (({}^k)^k) = ({}^{k+1})^{k+1} \in L. \quad \square$$

Aufgabe 18: Wir definieren für $w \in \Sigma^*$ und $c \in \Sigma$ die Funktion

$$\text{count}(w, c),$$

die zählt, wie oft der Buchstabe c in dem Wort w vorkommt, durch Induktion über w .

I.A.: $w = \varepsilon$.

Wir setzen

$$\text{count}(\varepsilon, c) := 0.$$

I.S.: $w = dv$ mit $d \in \Sigma$.

Dann wird $\text{count}(dv, c)$ durch eine Fall-Unterscheidung definiert:

$$\text{count}(dv, c) := \begin{cases} \text{count}(v, c) + 1 & \text{falls } c = d; \\ \text{count}(v, c) & \text{falls } c \neq d. \end{cases} \quad \diamond$$

Aufgabe 19: Wir setzen $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ und definieren die Sprache L als die Menge der Wörter $w \in \Sigma^*$, in denen die Buchstaben \mathbf{a} und \mathbf{b} mit der selben Häufigkeit vorkommen:

$$L := \{w \in \Sigma^* \mid \text{count}(w, \mathbf{a}) = \text{count}(w, \mathbf{b})\}$$

Geben Sie eine Grammatik G an, so dass $L = L(G)$ gilt und beweisen Sie Ihre Behauptung! \diamond

Hausaufgabe 20: Wieder sei $\Sigma = \{\mathbf{a}, \mathbf{b}\}$. Wir definieren die Menge L als die Menge der Strings s , die sich nicht in der Form $s = ww$ schreiben lassen:

$$L = \{s \in \Sigma^* \mid \neg(\exists w \in \Sigma^* : s = ww)\}.$$

Geben Sie eine kontextfreie Grammatik G an, die diese Sprache erzeugt. \diamond

7.1.2 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, \dots, X_n beschriftet, so enthält die Grammatik G eine Regel der Form

$$A \rightarrow X_1 X_2 \dots X_n.$$

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 7.1 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 7.1 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, das 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 7.2 zeigt den abstrakten Syntax-Baum den wir erhalten, wenn wir den in Abbildung 7.1 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.

7.1.3 Mehrdeutige Grammatiken

Die zu Anfang des Abschnitts 7.1 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:

$$G = \langle \{Expr\}, \{Number, “+”, “-”, “*”, “/”, “(”, “)”\}, R, Expr \rangle,$$

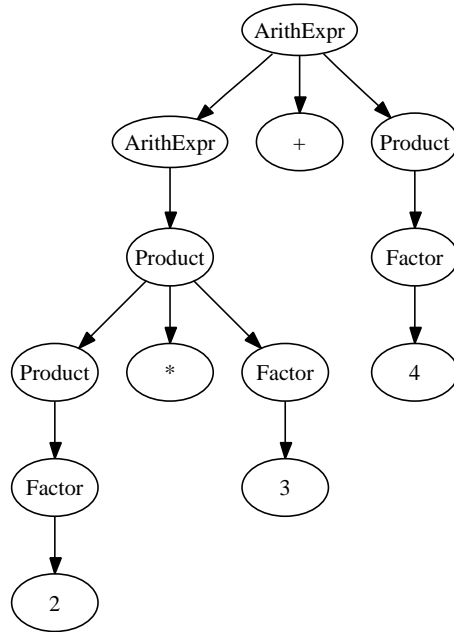


Abbildung 7.1: Ein Parse-Baum für den String “2*3+4”.

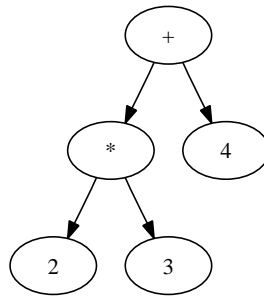


Abbildung 7.2: Ein abstrakter Syntax-Baum für den String “2*3+4”.

wobei die Regeln R wie folgt gegeben sind:

$$\begin{array}{l}
 \text{Expr} \rightarrow \text{Expr } "+" \text{ Expr} \\
 \quad | \text{Expr } "-" \text{ Expr} \\
 \quad | \text{Expr } "*" \text{ Expr} \\
 \quad | \text{Expr } "/" \text{ Expr} \\
 \quad | "(" \text{Expr } ")" \\
 \quad | \text{Number}
 \end{array}$$

Um zu zeigen, dass der String “2*3+4” in der von dieser Sprache erzeugten Grammatik liegt, geben wir die folgende Ableitung an:

$$\begin{array}{l}
 \text{Expr} \Rightarrow \text{Expr } "+" \text{ Expr} \\
 \Rightarrow \text{Expr } "*" \text{ Expr } "+" \text{ Expr} \\
 \Rightarrow 2 \text{ } "*" \text{ Expr } "+" \text{ Expr} \\
 \Rightarrow 2 \text{ } "*" \text{ } 3 \text{ } "+" \text{ Expr} \\
 \Rightarrow 2 \text{ } "*" \text{ } 3 \text{ } "+" \text{ } 4
 \end{array}$$

Diese Ableitung entspricht dem abstrakten Syntax-Baum, der in Abbildung 7.2 gezeigt ist. Es gibt aber noch eine andere Ableitung des Strings “2*3+4” mit dieser Grammatik:

$$\begin{aligned}
 \text{Expr} &\Rightarrow \text{Expr “*” Expr} \\
 &\Rightarrow \text{Expr “*” Expr “+” Expr} \\
 &\Rightarrow 2 \text{ “*” Expr “+” Expr} \\
 &\Rightarrow 2 \text{ “*” } 3 \text{ “+” Expr} \\
 &\Rightarrow 2 \text{ “*” } 3 \text{ “+” } 4
 \end{aligned}$$

Dieser Ableitung entspricht der abstrakte Syntax-Baum, der in Abbildung 7.3 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

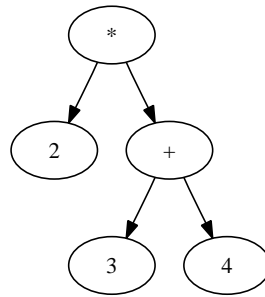


Abbildung 7.3: Ein anderer abstrakter Syntax-Baum für den String “2*3+4”.

zuletzt angegebene Grammatik mehrdeutig 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 Behauptung findet sich beispielsweise in dem Buch von Hopcroft, Motwani und Ullman [HMU06] auf Seite 415.

Beispiel: Es sei $\Sigma = \{a, b\}$. Die Sprache L enthalte alle die Wörter aus Σ^* , bei denen die Buchstaben a und b mit der gleichen Häufigkeit auftreten, es gilt also

$$L = \{w \in \Sigma^* \mid \text{count}(w, a) = \text{count}(w, b)\}.$$

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:

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

Diese Grammatik ist allerdings mehrdeutig: Betrachten wir beispielsweise den String “abab”, so stellen wir fest, dass sich dieser prinzipiell auf zwei Arten ableiten lässt:

$$\begin{aligned}
 S &\Rightarrow aSbS \\
 &\Rightarrow abS \\
 &\Rightarrow abaSbS \\
 &\Rightarrow ababS \\
 &\Rightarrow abab
 \end{aligned}$$

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:

$$\begin{aligned}
 S &\Rightarrow aSbS \\
 &\Rightarrow aSb \\
 &\Rightarrow abSaSb \\
 &\Rightarrow abaSb \\
 &\Rightarrow abab
 \end{aligned}$$

Abbildung 7.4 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”.

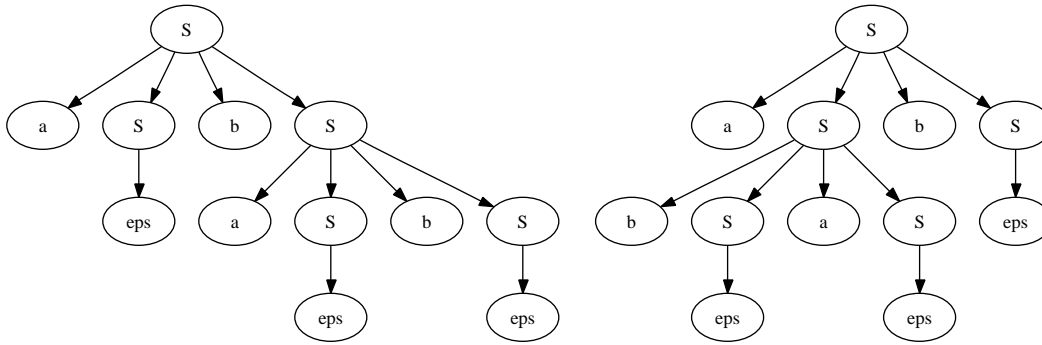


Abbildung 7.4: Zwei strukturell verschiedene Parse-Bäume für den String “abab”.

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:

$$\begin{aligned}
 S &\rightarrow US \mid VS \mid \varepsilon \\
 U &\rightarrow aXb \\
 V &\rightarrow bYa \\
 X &\rightarrow UX \mid \varepsilon \\
 Y &\rightarrow VY \mid \varepsilon
 \end{aligned}$$

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 w) wie folgt:

$$u \preceq w \stackrel{\text{def}}{\iff} \exists v \in \Sigma^* : uv = w$$

Sodann bemerken wir, dass von den syntaktischen Variablen X und Y die folgenden Sprachen erzeugt werden:

$$\begin{aligned}
 L(X) &= \{w \in \Sigma^* \mid w \in L \wedge \forall u \preceq w : \text{count}(u, b) \leq \text{count}(u, a)\} \quad \text{und} \\
 L(Y) &= \{w \in \Sigma^* \mid w \in L \wedge \forall u \preceq w : \text{count}(u, a) \leq \text{count}(u, b)\}.
 \end{aligned}$$

Ein String der Sprache $L(X)$ kann also kein “b” enthalten, das zu einem “a” gehört, das vor diesem String steht und analog kann ein String der Sprache $L(Y)$ kein “a” enthalten, das 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 $L(U)$. Auf dieses “b” kann dann noch ein weiterer Teilstring folgen, der gleich viele “a”s und “b”s enthält. Ein

solcher Teilstring liegt offensichtlich ebenfalls in der Sprache L und kann daher von S mittels der Regel

$$S \rightarrow US$$

erzeugt werden. Im zweiten Fall fängt der String mit einem “b” an. Dieser Fall ist analog zum ersten Fall. \square

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 $\Sigma = \{a, b, c, d\}$ die Sprache

$$L = \{a^m b^m c^n d^n \mid m, n \in \mathbb{N}\} \cup \{a^m b^n c^n d^m \mid m, n \in \mathbb{N}\}$$

kontextfrei ist, aber jede Grammatik G mit der Eigenschaft $L = L(G)$ ist notwendigerweise mehrdeutig. Das Problem ist, dass für gewisse große Zahlen $n \in \mathbb{N}$ ein String der Form

$$a^n b^n c^n d^n$$

immer zwei strukturell verschiedene Parse-Bäume besitzt. Ein Beweis dieser Behauptung findet sich in dem Buch von Hopcroft und Ullman auf Seite 100 [HU79].

7.2 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

$$A \rightarrow X_1 X_2 \cdots X_n$$

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 aufgespalten in

$$w = w_1 w_2 \cdots w_n \quad \text{mit } w_i \in L(X_i) \text{ für alle } i = 1, \dots, n.$$

Leider funktioniert dieses Verfahren dann nicht, wenn die Grammatik *links-rekursiv* ist, das heißt, dass eine Regel die Form

$$A \rightarrow A\beta$$

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

$$A \rightarrow X_1 X_2 \cdots X_n$$

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. Die einfachste Lösung erhalten wir, wenn wir uns klar machen, dass kontext-freie Grammatiken nicht unbedingt die bequemste Art darstellen, eine Sprache zu beschreiben. Wir werden daher den Begriff der *erweiterten Backus-Naur-Form* EBNF einführen. Hierbei handelt es sich um eine Verallgemeinerung des Begriffs der kontext-freien Grammatik, mit dem es in der Praxis leichter ist, Parser zu implementieren.

Im Rahmen dieses Kapitels werden wir alle oben genannten Verfahren an Hand der Grammatik für arithmetische Ausdrücke ausführlich diskutieren.

7.2.1 Umschreiben der Grammatik

Ist A ein Nicht-Terminal, das durch die beiden Regeln

$$\begin{array}{lcl} A & \rightarrow & A\beta \\ & | & \gamma \end{array}$$

beschrieben wird, so hat eine Ableitung von A , bei der zunächst immer die Variable A ersetzt wird, die Form

$$A \Rightarrow A\beta \Rightarrow A\beta\beta \Rightarrow A\beta\beta\beta \Rightarrow \dots \Rightarrow A\beta^n \Rightarrow \gamma\beta^n.$$

Damit sehen wir, dass die durch das Nicht-Terminal A beschriebene Sprache $L(A)$ aus allen den Strings besteht, die sich aus dem Ausdruck $\gamma\beta^n$ ableiten lassen:

$$L(A) = \{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\}.$$

Diese Sprache kann offenbar auch durch die folgenden Regeln für A beschrieben werden:

$$\begin{array}{lcl} A & \rightarrow & \gamma B \\ B & \rightarrow & \beta B \\ & | & \varepsilon \end{array}$$

Hier haben wir die Hilfs-Variable B eingeführt. Die Ableitungen, die von dem Nicht-Terminal B ausgehen, haben die Form

$$B \Rightarrow \beta B \Rightarrow \beta\beta B \Rightarrow \dots \Rightarrow \beta^n B \Rightarrow \beta^n.$$

Folglich beschreibt das Nicht-Terminal B die Sprache

$$L(B) = \{w \in \Sigma \mid \exists n \in \mathbb{N} : \beta^n \Rightarrow w\}.$$

Damit ist klar, dass auch mit der oben angegebenen Grammatik

$$L(A) = \{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\}$$

gilt. Um die Links-Rekursion aus der in Abbildung 7.5 auf Seite 88 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

$$\begin{array}{lcl} A & \rightarrow & A\beta_1 \\ & | & A\beta_2 \\ & \vdots & \vdots \\ & | & A\beta_k \\ & | & \gamma_1 \\ & \vdots & \vdots \\ & | & \gamma_l \end{array}$$

beschrieben wird. Wir können diesen Fall durch Einführung zweier Hilfs-Variablen B und C auf den ersten Fall zurückführen:

$$\begin{array}{lcl} A & \rightarrow & AB \mid C \\ B & \rightarrow & \beta_1 \mid \dots \mid \beta_k \\ C & \rightarrow & \gamma_1 \mid \dots \mid \gamma_l \end{array}$$

Dann können wir die Grammatik umschreiben, indem wir eine neue Hilfs-Variablen, nennen wir sie L für Liste, einführen und erhalten

$$\begin{array}{lcl} A & \rightarrow & CL \\ L & \rightarrow & BL \mid \varepsilon. \end{array}$$

Die Hilfs-Variablen B und C können nun wieder eliminiert werden und dann bekommen wir die folgende Grammatik:

$$\begin{aligned}
A &\rightarrow \gamma_1 L \mid \gamma_2 L \mid \cdots \mid \gamma_l L \\
L &\rightarrow \beta_1 L \mid \beta_2 L \mid \cdots \mid \beta_k L \mid \varepsilon
\end{aligned}$$

$$\begin{aligned}
\text{Expr} &\rightarrow \text{Expr "+" Product} \\
&\mid \text{Expr "-" Product} \\
&\mid \text{Product} \\
\text{Product} &\rightarrow \text{Product "*" Factor} \\
&\mid \text{Product "/" Factor} \\
&\mid \text{Factor} \\
\text{Factor} &\rightarrow "(" \text{Expr} ")" \\
&\mid \text{Number}
\end{aligned}$$

Abbildung 7.5: Links-rekursive Grammatik für arithmetische Ausdrücke.

Wenden wir dieses Verfahren auf die in Abbildung 7.5 gezeigte Grammatik für arithmetische Ausdrücke an, so erhalten wir die in Abbildung 7.6 gezeigte Grammatik.

$$\begin{aligned}
\text{Expr} &\rightarrow \text{Product ExprRest} \\
\text{ExprRest} &\rightarrow "+" \text{Product ExprRest} \\
&\mid "-" \text{Product ExprRest} \\
&\mid \varepsilon \\
\text{Product} &\rightarrow \text{Factor ProductRest} \\
\text{ProductRest} &\rightarrow "*" \text{Factor ProductRest} \\
&\mid "/" \text{Factor ProductRest} \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow "(" \text{Expr} ")" \\
&\mid \text{NUMBER}
\end{aligned}$$

Abbildung 7.6: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

Die Variablen *ExprRest* und *ProductRest* können wie folgt interpretiert werden:

1. *ExprRest* beschreibt eine Liste der Form

$$op \text{ Product } \cdots op \text{ Product},$$

wobei $op \in \{ "+", "-" \}$ gilt.

2. *ProductRest* beschreibt eine Liste der Form

$$op \text{ Factor } \cdots op \text{ Factor},$$

wobei $op \in \{ "*", "/" \}$ gilt.

Aufgabe 21:

- (a) Die folgende Grammatik beschreibt reguläre Ausdrücke:

$$\begin{array}{lcl}
 \text{RegExp} & \rightarrow & \text{RegExp "+" RegExp} \\
 & | & \text{RegExp RegExp} \\
 & | & \text{RegExp "*" } \\
 & | & "(" \text{RegExp} ")" \\
 & | & \text{LETTER}
 \end{array}$$

Diese Grammatik verwendet nur die syntaktische Variable $\{\text{RegExp}\}$ 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. \diamond

$$\begin{array}{lcl}
 S & \rightarrow & A \text{ "x"} \\
 & | & \text{"y"} \\
 A & \rightarrow & B \text{ "y"} \\
 & | & \text{"x"} \\
 B & \rightarrow & S \text{ "z"}
 \end{array}$$

Abbildung 7.7: Wechselseitige Links-Rekursion.

Bei den bisher diskutierten Beispielen war die Links-Rekursion der Grammatik unmittelbar anzusehen. Es gibt allerdings Fälle, in denen die Links-Rekursion durch wechselseitige Rekursion begründet ist. Abbildung 7.7 auf Seite 89 zeigt ein Beispiel: Bei der dort angegebenen Grammatik erstreckt sich die Links-Rekursion über drei Stufen: Ein S kann mit einem A beginnen, das mit einem B beginnen kann, welches dann wieder mit einem S beginnt. Eine Links-Rekursion der Form

$$A \rightarrow A\beta$$

bezeichnen wir als *unmittelbare Links-Rekursion*, jede kompliziertere Form der Links-Rekursion wird als *allgemeine Links-Rekursion* bezeichnet. Um im allgemeine Links-Rekursion zu eliminieren, gehen wir wie folgt vor:

1. Zunächst numerieren wir die syntaktischen Variablen der Grammatik in willkürlicher Weise durch. Im Folgenden seien die Variablen mit A_1, \dots, A_n bezeichnet. Durch diese Numerierung wird implizit eine Ordnung \succ auf den syntaktischen Variablen definiert, wir setzen

$$A_1 \succ A_2 \succ \dots \succ A_i \succ A_{i+1} \succ \dots \succ A_n.$$

2. Das Ziel ist nun, die Grammatik so umzuschreiben, dass für jede Grammatik-Regel der Form

$A \rightarrow B\gamma$ die Ungleichung $A \succ B$

gilt. Dieses Ziel wird durch den folgenden Algorithmus erreicht:

```

for (i = 1; i <= n; ++i) {
  for (j = 1; j < i; ++j) {
    forall  $(A_i \rightarrow A_j\gamma) \in R$  {
      let  $A_j \rightarrow \delta_1 \mid \dots \mid \delta_k$  be all  $A_j$ -productions
      replace  $A_i \rightarrow A_j\gamma$  by  $A_i \rightarrow \delta_1\gamma \mid \dots \mid \delta_k\gamma$ 
    }
  }
  eliminate immediate left recursion for variable  $A_i$ 
}

```

Um den Algorithmus zu verstehen, führen wir zunächst einen neuen Begriff ein: Falls die Grammatik eine Regel der Form

$$A \rightarrow B\gamma$$

enthält, wobei B eine syntaktische Variable ist, dann sagen wir, dass A unmittelbar von B abhängt. Die Idee bei dem oben angegebenen Algorithmus ist nun, dass nach dem i -ten Durchlaufen der äußeren **for**-Schleife die Variablen A_1, \dots, A_i nur noch unmittelbar von solchen Variablen abhängen, die in der Aufzählung A_1, \dots, A_n auf diese Variablen folgen. Formal gilt nach dem i -ten Durchlauf der äußeren **for**-Schleife für alle Indizes $k \in \{1, \dots, i\}$: Falls die Grammatik eine Regel der Form

$$A_k \rightarrow A_l\beta$$

enthält, dann muss $l > k$ gelten. Es ist leicht zu sehen, dass diese Invariante tatsächlich gilt: Vor dem i -ten Durchlauf gilt die Invariante für die Indizes der Menge $\{1, \dots, i-1\}$. Die Variable A_i selber kann dann noch unmittelbar von den Variablen A_1, \dots, A_n abhängen. In der inneren **for**-Schleife erreichen wir, dass nacheinander die unmittelbaren Abhängigkeiten von A_1, \dots, A_{i-1} aufgelöst werden. Anschließend kann A_i höchstens noch unmittelbar von A_i abhängen. Um diese Abhängigkeit gegebenenfalls aufzulösen, führen wir die bereits früher diskutierte Transformation zur Elimination unmittelbarer Links-Rekursion durch. Anschließend hängt A_i höchstens noch unmittelbar von A_{i+1}, \dots, A_n ab. Läuft die Schleife bis zum Ende durch, ist damit dann die Links-Rekursion vollständig eliminiert, denn dann kann jede Variable nur von solchen Variablen abhängen, die in der Aufzählung A_1, \dots, A_n hinter ihr stehen. Folglich ist kein Zyklus der Form

$$A_i \Rightarrow A_j\beta \Rightarrow \dots \Rightarrow A_i\gamma$$

mehr möglich.

Beispiel: Wir demonstrieren das Verfahren an der in Abbildung 7.7 gezeigten Grammatik. Dazu ordnen wir zunächst die Variablen in der Form

$$S, A, B$$

an, mit der Notation des oben angegebenen Algorithmus gilt also $A_1 := S$, $A_2 := A$ und $A_3 := B$.

1. $i = 1$: Da $\{1, \dots, i-1\} = \{\}$ gilt, wird in diesem Fall die innere **for**-Schleife nicht ausgeführt. Wir müssen lediglich die unmittelbare Links-Rekursion in der Variablen S entfernen. Da die Grammatik aber für S keine unmittelbare Links-Rekursion enthält, ist in diesem Fall nichts zu tun.
2. $i = 2$: In diesem Schritt müssen wir in der inneren **for**-Schleife sicherstellen, dass A nicht unmittelbar von S abhängt. Da A in der gegebenen Grammatik nicht unmittelbar von S abhängt, ist bei der inneren **for**-Schleife wieder nichts zu tun.

Weiter müssen wir die unmittelbare Rekursion aus allen Regeln für A eliminieren. Da es für die Variable A keine unmittelbare Rekursion gibt, ist an dieser Stelle wieder nichts zu tun.

3. $i = 3$: In diesem Fall kommen für die innere **for**-Schleife zwei Werte von j in Frage, die wir nacheinander behandeln müssen:

- (a) $j = 1$: Hier müssen wir sicherstellen, dass B nicht unmittelbar von S abhängt. Bei der Regel

$$B \rightarrow S \text{ "z"}$$

ist dies aber der Fall. Wir ersetzen daher das S auf der rechten Seite dieser Regel durch die beiden rechten Seiten der Regeln für S und erhalten für B nun die Regeln

$$B \rightarrow A \text{ "x" "z"} \quad \text{und} \quad B \rightarrow \text{"y" "z"}.$$

- (b) $j = 2$: Nun müssen wir sicherstellen, dass B nicht unmittelbar von A abhängt. Bei der Regel

$$B \rightarrow A \text{ "x" "z"}$$

ist dies aber der Fall. Wir ersetzen daher das A auf der rechten Seite dieser Regel durch die beiden rechten Seiten der Regeln für A und erhalten für B nun insgesamt die folgenden Regeln für B :

$$B \rightarrow B \text{ "y" "x" "z"}, \quad B \rightarrow \text{"x" "x" "z"} \quad \text{und} \quad B \rightarrow \text{"y" "z"}.$$

Diese Regeln enthalten nun nur noch unmittelbare Links-Rekursion, die wir mit dem früher beschriebenen Verfahren eliminieren. Wir erhalten dann für B die Regeln

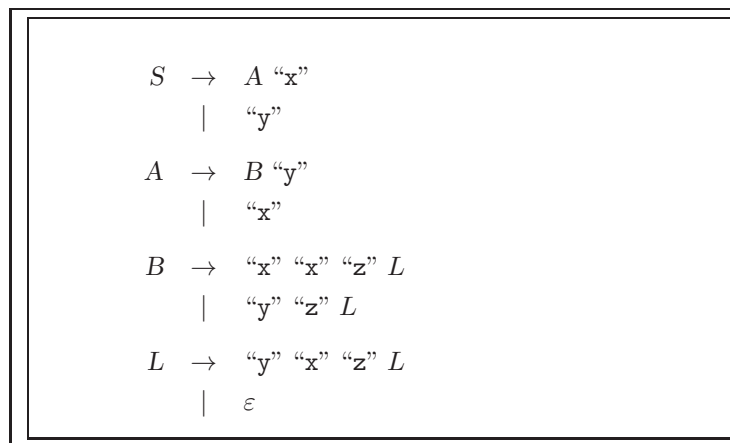
$$B \rightarrow \text{"x" "x" "z"} L \quad \text{und} \quad B \rightarrow \text{"y" "z"} L,$$

wobei die neu eingeführte Variable L durch die Regeln

$$L \rightarrow \text{"y" "x" "z"} L \quad \text{und} \quad L \rightarrow \varepsilon$$

definiert ist.

Abbildung 7.8 zeigt die resultierende Grammatik.



$$\begin{array}{lcl}
 S & \rightarrow & A \text{ "x" } \\
 & | & \text{"y"} \\
 A & \rightarrow & B \text{ "y" } \\
 & | & \text{"x"} \\
 B & \rightarrow & \text{"x" "x" "z"} L \\
 & | & \text{"y" "z"} L \\
 L & \rightarrow & \text{"y" "x" "z"} L \\
 & | & \varepsilon
 \end{array}$$

Abbildung 7.8: Grammatik ohne Links-Rekursion.

7.2.2 Implementing a Top Down Parser in SetlX

Now we are ready to implement a parser for arithmetic expressions. Figure 7.9 on page 92 shows an implementation of a recursive decent parser in SETLX.

1. The main function is `myParse`¹. This function takes a string s representing an arithmetic expression. This string is tokenized using the function `tokenizeString`. The function `tokenizeString` turns a string into a list of tokens. For example, the expression

¹ We had to name the function `myParse` instead of `parse` as SETLX already implements a function with the name `parse`. This function parses strings as SETLX expressions. The function `parse` returns a term representing the abstract syntax tree corresponding to the parsed expression.

```

1  myParse := procedure(s) {
2      [result, rl] := parseExpr(tokenizeString(s));
3      return result;
4  };
5
6  parseExpr := procedure(tl) {
7      [product, rl] := parseProduct(tl);
8      return parseExprRest(product, rl);
9  };
10 parseExprRest := procedure(sum, tl) {
11     match (tl) {
12         case ["+" | rl] : [product, ql] := parseProduct(rl);
13                             return parseExprRest(sum + product, ql);
14         case ["-" | rl] : [product, ql] := parseProduct(rl);
15                             return parseExprRest(sum - product, ql);
16         default:         return [sum, tl];
17     }
18 };
19 parseProduct := procedure(tl) {
20     [factor, rl] := parseFactor(tl);
21     return parseProductRest(factor, rl);
22 };
23 parseProductRest := procedure(product, tl) {
24     match (tl) {
25         case ["*" | rl] : [factor, ql] := parseFactor(rl);
26                             return parseProductRest(product * factor, ql);
27         case ["/" | rl] : [factor, ql] := parseFactor(rl);
28                             return parseProductRest(product / factor, ql);
29         default:         return [product, tl];
30     }
31 };
32 parseFactor := procedure(tl) {
33     match (tl) {
34         case ["(" | rl] : [expr, ql] := parseExpr(rl);
35                             return [expr, ql[2..]];
36         default : return [tl[1], tl[2..]];
37     }
38 };
39
40 tokenizeString := procedure(s) {
41     tokenList := [];
42     scan (s) {
43         regex '0|[1-9][0-9]*' as [ number   ]: tokenList += [ int(number) ];
44         regex '[-+*/()]'      as [ operator ]: tokenList += [ operator   ];
45         regex '[\t\v\n]+'      : // skip
46     }
47     return tokenList;
48 };

```

Abbildung 7.9: A top down parser for arithmetic expressions.

```
tokenizeString("(1 + 2) * 3");
```

returns the result

```
["(", 1, "+", 2, ")", "*", 3].
```

The list of tokens is the parsed by the function `parseExpr`. That function returns a pair:

- (a) The first component is the value of the arithmetic expression.
- (b) The second component is the list of those tokens that have not been consumed when parsing the expression. Of course, on a successful parse this list should be empty.

2. The function `parseExpr` implements the grammar rule

$$Expr \rightarrow Product\ ExprRest.$$

It takes a token list `tl` as input. It will return a pair of the form

```
[v, rl],
```

where v is the value of the arithmetic expression that has been parsed, while `rl` is the list of the remaining tokens. For example, the expression

```
parseExpr(["(", 1, "+", 2, ")", "*", 3, ")", "*", 2])
```

returns the result

```
[9, [")", "*", 2]].
```

Here, the part `["(", 1, "+", 2, ")", "*", 3]` has been parsed and evaluated as the number 9 and `[")", "*", 2]` is the list of tokens that have not yet been processed.

In order to parse an arithmetic expression, the function first parses a *Product* and then it tries to parse the remaining tokens as an *ExprRest*. The function `parseExprRest` that is used to parse an *ExprRest* needs two arguments:

- (a) The first argument is the value of the product that has been parsed by the function `parseProduct`.
- (b) The second argument is the list of tokens that can be used.

To understand the mechanics of `parseExpr`, consider the evaluation of

```
[1, "*", 2, "+", 3].
```

Here, the function `parseProduct` will return the result

```
[2, ["+", 3]],
```

where 2 is the result of parsing the token list `[1, "*", 2]`, while `["+", 3]` is the part of the input token list that is not used by `parseProduct`. Next, the list `["+", 3]` needs to be parsed as the rest of an expression and 3 needs to be added to 2.

3. The function `parseExprRest` takes a number and a list of tokens. It implements the grammar rule

$$\begin{array}{lcl} ExprRest & \rightarrow & "+" Product\ ExprRest \\ & | & "-" Product\ ExprRest \\ & | & \varepsilon \end{array}$$

Therefore, it checks whether the first token is either "+" or "-". If the token is "+", it parses a *Product*, adds the result of this product to the **sum** of values parsed already and proceeds to parse the rest of the tokens.

The case that the first token is "-" is similar to the previous case. If the next token is neither "+" nor "-", then it could be either the token ")" or else it might be the case that the list of tokens is already exhausted. In either case, the rule

$$\text{ExprRest} \rightarrow \varepsilon$$

is used. Therefore, in that case we have not consumed any tokens and therefore the input arguments are already the result.

4. The function `parseProduct` implements the rule

$$\text{Product} \rightarrow \text{Factor ExprRest}.$$

The implementation is similar to the implementation of `parseExpr`.

5. The function `parseProductRest` implements the rules

$$\begin{array}{lcl} \text{ProductRest} & \rightarrow & "*" \text{Factor ProductRest} \\ & | & "/" \text{Factor ProductRest} \\ & | & \varepsilon \end{array}$$

The implementation is similar to the implementation of `parseExprRest`.

6. The function `parseProductRest` implements the rules

$$\begin{array}{lcl} \text{Factor} & \rightarrow & "(" \text{Expr} ")" \\ & | & \text{NUMBER} \end{array}$$

Therefore, we first check whether the next token is "(" because in that case, we have to use the first grammar rule, otherwise we use the second.

7. The last function `tokenizeString` transforms a string into a list of tokens. To this end it uses the `scan` mechanism that is already built into SETLX. For example, in line 43 it is checked whether the next part of the input string is matched by the regular expression `0|[1-9][0-9]*`. If this is the case, the matching part is chopped off the string and converted into a number which is then added to the list of tokens seen so far.

In line 44 we recognize the operator symbols and the parenthesis. Note that we had to put the operator "-" first here since otherwise it would have been mistaken as a range operator.

Line 45 is needed to skip white space.

The parser shown in Figure 7.9 does not contain any error handling. Appropriate error handling will be discussed once we have covered the theory of top down parsers.

7.2.3 Implementing a Recursive Decent Parser that Works Backwards

If a grammar is left recursive but not right recursive then, instead of rewriting the grammar, we can just try to read the grammar rules backwards. Figure 7.10 on page 95 shows an recursive decent parser for arithmetic expressions that works backwards.

```

1  parseExpr := procedure(tl) {
2      [fp, product] := parseProduct(tl);
3      [rp, op] := split(fp);
4      if (op in ["+", "-"]) {
5          [fp, expr] := parseExpr(rp);
6          match (op) {
7              case "+": return [fp, expr + product];
8              case "-": return [fp, expr - product];
9          }
10     }
11     return [fp, product];
12 };
13 parseProduct := procedure(tl) {
14     [fp, factor] := parseFactor(tl);
15     [rp, op] := split(fp);
16     if (op in ["*", "/"]) {
17         [fp, product] := parseProduct(rp);
18         match (op) {
19             case "*": return [fp, product * factor];
20             case "/": return [fp, product / factor];
21         }
22     }
23     return [fp, factor];
24 };
25 parseFactor := procedure(tl) {
26     [fp, op] := split(tl);
27     if (op == ")") {
28         [fp, expr] := parseExpr(fp);
29         [fp, op] := split(fp);
30         assert(op == "(", "parse error in $parseFactor(tl)$");
31         return [fp, expr];
32     }
33     assert(isNumber(op), "parse error in $parseFactor(tl)$");
34     return [fp, op];
35 };
36 split := procedure(l) {
37     if (#l > 0) {
38         return [l[1 .. #l-1], l[#l]];
39     }
40     return [[], ""];
41 };

```

Abbildung 7.10: A recursive decent parser working backwards.

1. The function `parseArithExpr` implements the following grammar rules:

$$\text{ArithExpr} \rightarrow \text{ArithExpr} "+" \text{Product} \mid \text{ArithExpr} "-" \text{Product} \mid \text{Product}.$$

According to these rules, an *ArithExpr* always ends with a *Product*. Therefore, the first thing to do is to parse a *Product* from the end of the token list. This is done using the procedure `parseProduct`. Invoking this procedure consumes some of the tokens from the token list `t1` and returns the list `fp` of those tokens that have not been consumed together with the *product* that has been recognized. Then, there are three cases:

- (a) If the token immediately preceding the *product* is the symbol “+”, then the parser tries to recognize an arithmetic expression using a recursive invocation of the procedure `parseExpr`. If this works and returns the result *expr*, then the end result is the sum *expr* + *product*, which is returned in line 7 together with the token that have not been consumed.
- (b) If the token immediately preceding the *product* is “-”, then everything works as in the first case, but instead the parser returns *expr* - *product*.
- (c) Otherwise, the parser has hit an opening parenthesis. In this case, the parser just returns the *product* together with the remaining tokens.

2. The function `parseProduct` tries to parse a product using the following rules:

$$\begin{array}{lcl}
 \textit{Product} & \rightarrow & \textit{Product} \text{ “*” } \textit{Factor} \\
 & | & \textit{Product} \text{ “/” } \textit{Factor} \\
 & | & \textit{Factor}.
 \end{array}$$

This time, the parser tries to recognize a *factor* at the end of the token list `t1`. If this *factor* is preceded by either the token “*” or “/”, the parser tries to recognize the *product* that must be preceding this operator. In that case, depending on the operator, the parser either returns *product* * *factor* or *product* / *factor*.

If the factor is not preceded by either “*” or “/” it must be preceded by an opening parenthesis. In this case, the parser just returns the *factor*.

3. The function `parseFactor` implements the following grammar rules:

$$\begin{array}{lcl}
 \textit{Factor} & \rightarrow & \text{“(” } \textit{ArithExpr} \text{ “)”} \\
 & | & \textit{Number}.
 \end{array}$$

4. The method `split` is an auxiliary method that takes a list *l* as input. If this list is not empty, this method returns a pair: The first component of this list is the list of all elements of *l* but the last element, while the second component is the last element of the list *l*.

7.2.4 Implementing a Recursive Decent Parser that Uses an EBNF Grammar

The previous two solutions to parse an arithmetical expressions were not completely satisfying: The reason is that we did not really fix the problem but rather cured the symptoms. The real problem is that context free grammars are not that convenient to describe programming languages. Let us extend the power of context free languages slightly by admitting regular expression on the right hand side of grammar rules. These new type of grammars are known as *extended Backus Naur form* grammars, which is abbreviated as EBNF grammars. An EBNF grammar admits the operators “*”, “?”, “+”, and “|” on the right hand side of a grammar rule. The meaning of these operators is the same as when these operators are used in regular expressions.

It can be shown that the languages described by EBNF grammars are still context free languages. Therefore, these operators do not change the descriptive power of grammars. However, it is often much more convenient to describe a language using an EBNF grammar rather than using a context free grammar. Figure 7.11 displays an EBNF grammar for arithmetical expressions. We have extended this grammar to allow for the exponentiation operator “**”. In order to support this operator, we had to introduce a new syntactical variable, which we called *Base*. In an arithmetical expression, the *Base* serves as the base of a power. The exponent can be an arbitrary *Factor*. This way, an expression of the form

2 ** 3 ** 4 is parsed as 2 ** (3 ** 4)

and therefore the operator “**” is right associative. Furthermore, we have added the function symbols “exp” and “ln” to be able to support the exponential function and the natural logarithm. The grammar also supports variables. The reason is that we want to implement a program for symbolic differentiation. We want to implement a function that takes a string representing an arithmetical expression and then does the following:

1. In the first step, the string is translated into an abstract syntax tree.
2. In the second step, this tree is differentiated symbolically with respect to a given variable.

Obviously, the grammar in Figure 7.11 is more concise than the context free grammar discussed at the beginning of this chapter. For example, the first rule clearly expresses that an arithmetical expression is a list of products that are separated by the operators “+” and “-”.

<i>Expr</i>	→	<i>Product</i> ((“+” “-”) <i>Product</i>) [*]
<i>Product</i>	→	<i>Product</i> ((“*” “/”) <i>Factor</i>) [*]
<i>Factor</i>	→	<i>Base</i> (“**” <i>Factor</i>)?
<i>Base</i>	→	“(” <i>Expr</i> “)”
		“exp” “(” <i>Expr</i> “)”
		“ln” “(” <i>Expr</i> “)”
		NUMBER
		VARIABLE

Abbildung 7.11: EBNF grammar for arithmetical expressions.

Figure 7.12 shows a parser that implements this grammar.

1. The function `parseArithExpr` recognizes a **product** in line 2. The value of this **product** is stored in the variable **result** together with the list **rl** of those tokens that have not been consumed yet. If the list **rl** is not empty and the first token in this list is either the operator “+” or the operator “-”, then the function `parseArithExpr` tries to recognize more products. These are added to or subtracted from the **result** computed so far in line 7 or 8. If there are no more products to be parsed, the **while** loop terminates and the function returns the **result** together with the remaining tokens.

2. The function `parseProduct` recognizes a **factor** in line 2. The value of this **factor** is stored in the variable **result** together with the list **rl** of those tokens that have not been consumed yet. If the list **rl** is not empty and the first token in this list is either the operator “*” or the operator “/”, then the function `parseProduct` tries to recognize more factors. The **result** computed so far is multiplied with or divided by these factors in line 19 or 20. If there are no more products to be parsed, the **while** loop terminates and the function returns the **result** together with the list **rl** of tokens that have not been used.
3. The function `parseFactor` recognizes a **factor**. In any case, a factor starts with a base. Optionally, a factor can be a power. This is the case if the base is followed by the exponentiation operator “**”.
4. The function `parseBase` recognizes a call of the exponential function, a call of the natural logarithm, a parenthesized expression, a number, or a variable. Fortunately, the first token of the token list tells us which case we have.

The program in Figure 7.12 generates an abstract syntax tree. This syntax tree is represented as a term in SETLX. Note that in SETLX it is possible to use operators as functors. For example, if we have the expression

$$s + t$$

and at least one of the arguments s or t is a term, then $s + t$ is a term, too. This enables us to write

```
result := result + arg;
```

in line 7 of Figure 7.12. Finally, Figure 7.13 shows the implementation of the function `diff` that can be used for symbolic differentiation. The argument t of this function is supposed to be a term, the second argument x is interpreted as the name of a variable. For example, line 7 and 8 of Figure 7.13 implement the product rule. We have

$$\frac{d}{dx}(u \cdot v) = \frac{du}{dx} \cdot v + u \cdot \frac{dv}{dx}.$$

The right hand side of this equation is returned in line 8: **t1** correspond to u and **t2** corresponds to v . The other rules for differentiation are implemented in a similar way.

Historisches Die Sprache ALGOL [Bac59, NBB⁺60] war die erste Programmier-Sprache, deren Syntax auf einer kontextfreien Grammatik basiert.

```

1  parseArithExpr := procedure(tl) {
2      [result, rl] := parseProduct(tl);
3      while (#rl > 1 && rl[1] in ["+", "-"]) {
4          op := rl[1];
5          [arg, rl] := parseProduct(rl[2..]);
6          match (op) {
7              case "+": result := result + arg;
8              case "-": result := result - arg;
9          }
10     }
11     return [result, rl];
12 };
13 parseProduct := procedure(tl) {
14     [result, rl] := parseFactor(tl);
15     while (#rl > 1 && rl[1] in ["*", "/"]) {
16         op := rl[1];
17         [arg, rl] := parseFactor(rl[2..]);
18         match (op) {
19             case "*": result := result * arg;
20             case "/": result := result / arg;
21         }
22     }
23     return [result, rl];
24 };
25 parseFactor := procedure(tl) {
26     [atom, rl] := parseBase(tl);
27     match (rl) {
28         case [ "**" | ql ]: [factor, rl] := parseFactor(ql);
29                             return [atom ** factor, rl];
30         default:           return [atom, rl];
31     }
32 };
33 parseBase := procedure(tl) {
34     match (tl) {
35         case [ "exp", "(" | rl ]: [expr, ql] := parseArithExpr(rl);
36                                     assert(ql[1] == ")", "Parse Error");
37                                     return [ Exp(expr), ql[2..] ];
38         case [ "ln", "(" | rl ]: [expr, ql] := parseArithExpr(rl);
39                                     assert(ql[1] == ")", "Parse Error");
40                                     return [ Ln(expr), ql[2..] ];
41         case [ "(" | rl ]: [expr, ql] := parseArithExpr(rl);
42                                     assert(ql[1] == ")", "Parse Error");
43                                     return [expr, ql[2..] ];
44         case [ Number(n) | rl ]: return [Number(n), rl];
45         case [ Var(v) | rl ]:   return [Var(v), rl];
46         default:               abort("parse error in parseBase($tl$)");
47     }
48 };

```

Abbildung 7.12: A parser for the grammar in Figure 7.11.

```

1  diff := procedure(t, x) {
2      match (t) {
3          case t1 + t2 :
4              return diff(t1, x) + diff(t2, x);
5          case t1 - t2 :
6              return diff(t1, x) - diff(t2, x);
7          case t1 * t2 :
8              return diff(t1, x) * t2 + t1 * diff(t2, x);
9          case t1 / t2 :
10             return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / (t2 * t2);
11         case f ** Number(n):
12             return Number(n) * diff(f, x) * f ** Number(n-1);
13         case f ** g :
14             return diff( Exp(g * Ln(f)), x);
15         case Ln(a) :
16             return diff(a, x) / a;
17         case Exp(a) :
18             return diff(a, x) * Exp(a);
19         case Var(x) : // x is defined above as second argument to diff
20             return 1;
21         case Var(y) : // y is undefined, therefore matches any other variable
22             return 0;
23         case Number(n):
24             return 0;
25         default:
26             abort("error in diff($t$, $x$)");
27     }
28 };

```

Abbildung 7.13: A function for symbolic differentiation.

Kapitel 8

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 [Par07]. Der Name steht für *another tool for language recognition*¹. Sie finden dieses Werkzeug im Internet unter der folgenden Adresse

<http://www.antlr.org>

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.

8.1 Ein Parser für arithmetische Ausdrücke

Wir beginnen mit einem Parser für arithmetische Ausdrücke, die entsprechend der in Abbildung 7.5 auf Seite 88 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 7.6 auf Seite 88 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 widerspiegelt. 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.

Zusätzlich können Sie die rechten Seiten einer Grammatik-Regel durch Verwendung von Klammern strukturieren. Außerdem können Sie innerhalb solcher Klammern 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 8.1 auf Seite 102 gezeigten kompakten Form. Grammatiken dieser erweiterten Form werden als EBNF-Grammatiken bezeichnet. Die Abkürzung EBNF steht dabei für *extended Backus Naur form*.

Die in der Abbildung gezeigte Grammatik ist wie folgt zu lesen:

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.

$Expr$	\rightarrow	$Product \ (("+" \mid "-") \ Product) *$
$Expr$	\rightarrow	$Factor \ (("*" \mid "/") \ Factor) *$
$Factor$	\rightarrow	$"(" \ Expr \ ")"$
	$ $	$Number$

Abbildung 8.1: Erweiterte Grammatik für arithmetische Ausdrücke.

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

$$p_1 \pm p_2 \pm \dots \pm p_n.$$

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:

$$f_1 * f_2 * \dots * f_n.$$

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 8.2 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

$$('0' \dots '9')*$$

steht hier für eine beliebige, eventuell auch leere, Folge aus Ziffern. In *Flex* hätten wir statt dessen den

²Es ist Konvention, aber nicht vorgeschrieben, dass die Namen von Terminalen nur aus Großbuchstaben bestehen.

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    : '0'|('1'..'9')('0'..'9')*;
12 WS       : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 8.2: ANTLR-Spezifikation eines Parsers für arithmetische Ausdrücke.

Um aus der in Abbildung 8.2 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 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 eine 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 8.3 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 8.3: 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 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.

8.2 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 8.4 zeigt diese Grammatik. Gegenüber der vorher gezeigten Grammatik für arithmetische Ausdrücke gibt es die folgenden Erweiterungen:

```

1  grammar program;
2
3  program    : stmtnt+ ;
4
5  stmtnt     : 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'|('1'..'9')('0'..'9')*;
20 WS : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 8.4: 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. **stmtnt** 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 8.5 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**” spezifiziert wird, an den Anfang der erstellten Parser-Datei.

```

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    : stmtnt+ ;
12
13 stmtnt : 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'|('1'..'9')('0'..'9')*;
40 WS : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 8.5: Ein Interpreter zur Auswertung von Ausdrücken.

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`” spezifiziert 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

$$stmnt \rightarrow ID '=' expr ';'$$

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

$$expr \text{ returns } [int \text{ result}]$$

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

$$expr : product (('+' | '-') product)* ;$$

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.

8.3 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 8.6 zeigt die entsprechend erweiterte EBNF-Grammatik.

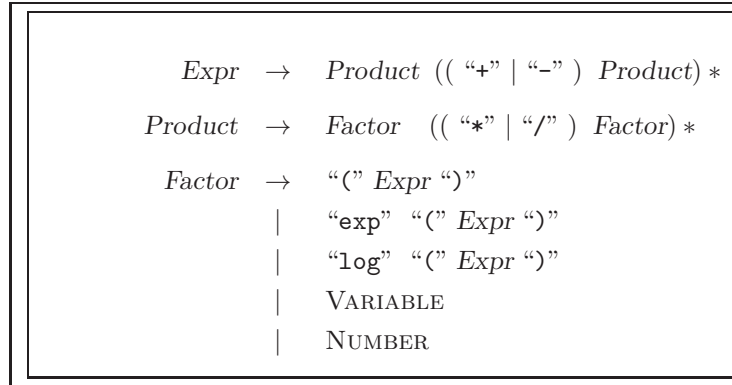


Abbildung 8.6: EBNF-Grammatik für arithmetische Ausdrücke mit Exponential-Funktion und Logarithmus.

```

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'|('1'..'9')('0'..'9')*;
28 WS : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 8.7: Die ANTLR-Spezifikation der Grammatik.

8.3.1 Implementierung des Parsers

Abbildung 8.7 zeigt die ANTLR-Implementierung der Grammatik aus Abbildung 8.6.

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 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 $(\dots)^*$ eingeschlossen ist.

Beispielsweise wird ein Ausdruck der Form

$$p_1 + p_2 + p_3$$

übersetzt in ein Java-Objekt der Form

$$\text{Sum}(\text{Sum}(p_1, p_2), p_3).$$

3. Die weiteren Grammatik-Regeln erzeugen in analoger Weise *Java*-Objekte.

Zum Abschluß zeigt Abbildung 8.8 noch die Einbindung des Parsers. Gegenüber dem in Abbildung 8.3 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 8.8: Ein Treiber für den Parser.

Aufgabe 22: 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 9

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 an Hand der nächsten k Token entscheidet, welche Regel verwendet wird. Falls beispielsweise $k = 1$ ist, wird also nur das nächste Token zur Entscheidung herangezogen. Dieses Token bezeichnen wir dann als *Lookahead-Token*. Wir betrachten zunächst den 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, die beispielsweise in den Regeln folgenden Regeln zur Beschreibung arithmetischer Ausdrücke auftritt:

$$\begin{array}{lcl} Expr & \rightarrow & Expr \text{ “+” } Product \\ & | & Expr \text{ “-” } Product \\ & | & Product \end{array}$$

Wir hatten im Abschnitt 7.2.1 gezeigt, wie die Links-Rekursion aus einer Grammatik eliminiert werden kann.

Zusätzlich haben wir gesehen, dass es bei der Verwendung von EBNF-Grammatiken oft in natürlicher Weise möglich, die Links-Rekursion durch die Verwendung der Postfix-Operatoren “*” und “+” zu vermeiden. Beispielsweise können wir die obigen Regeln für arithmetische Ausdrücke zu der EBNF-Regel

$$Expr \rightarrow Product \left((\text{ “+” } | \text{ “-” }) Product \right)^*$$

umschreiben.

2. Das zweite Problem erkennen wir, wenn wir die folgende Grammatik-Regel für Gleichungen und Ungleichungen betrachten:

$$\begin{array}{lcl} boolExpr & \rightarrow & expr \text{ “=” } expr \\ & | & expr \text{ “<” } expr \end{array}$$

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 dieses Problem verschwindet.

9.1 Links-Faktorisierung

Ist A ein Nicht-Terminal und gibt es zwei verschiedene Regeln, mit denen A abgeleitet werden kann, beispielsweise

$$A \rightarrow \beta \quad \text{und} \quad A \rightarrow \gamma,$$

so muss es bei der 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 9.1 zeigt eine vollständige Grammatik, mit der Gleichungen und Ungleichungen arithmetischer Ausdrücke beschrieben werden können.

<i>boolExpr</i>	\rightarrow	<i>expr</i> “==” <i>expr</i>
		<i>expr</i> “!=” <i>expr</i>
		<i>expr</i> “<=” <i>expr</i>
		<i>expr</i> “>=” <i>expr</i>
		<i>expr</i> “>” <i>expr</i>
		<i>expr</i> “<” <i>expr</i>
<i>expr</i>	\rightarrow	<i>product</i> <i>exprRest</i>
<i>exprRest</i>	\rightarrow	“+” <i>product</i> <i>exprRest</i>
		“-” <i>product</i> <i>exprRest</i>
		ε
<i>product</i>	\rightarrow	<i>factor</i> <i>productRest</i>
<i>productRest</i>	\rightarrow	“*” <i>factor</i> <i>productRest</i>
		“/” <i>factor</i> <i>productRest</i>
		ε
<i>factor</i>	\rightarrow	“(” <i>expr</i> “)”
		NUMBER
		IDENTIFIER

Abbildung 9.1: Grammatik ohne Links-Rekursion für Gleichungen und Ungleichungen.

Es ist nicht möglich einen LL(1)-Parser zu implementieren, der Boole’sche Ausdrücke mit Hilfe dieser Regeln erkennen kann, denn alle Regeln für *boolExpr* beginnen mit *expr*. Wir können die Grammatik aber durch *Links-Faktorisierung* (Englisch: *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 so die Regeln

$$\begin{aligned}
 \text{boolExpr} &\rightarrow \text{expr } \text{boolExprRest} \\
 \text{boolExprRest} &\rightarrow \begin{array}{l} \text{“==” expr} \quad | \quad \text{“!=” expr} \quad | \quad \text{“<=” expr} \\ \text{“>=” expr} \quad | \quad \text{“<” expr} \quad | \quad \text{“>” expr.} \end{array}
 \end{aligned}$$

Mit diesen Regeln reicht nun ein Token als Look-Ahead aus, denn die verschiedenen Alternativen für *boolExprRest* unterscheiden sich in dem ersten Token des Rumpfs der Regel. Verwenden wir statt einer einfachen

Grammatik eine EBNF-Grammatik, so lassen sich die obigen Regeln kürzer in der Form

$$\text{boolExpr} \rightarrow \text{expr} (\text{"="} \mid \text{"!="} \mid \text{"<="} \mid \text{">="} \mid \text{"<"}) \text{expr}$$

darstellen.

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:

$$\begin{array}{l} A \rightarrow \alpha \beta_1 \\ \quad | \quad \alpha \beta_2 \\ \quad \vdots \\ \quad | \quad \alpha \beta_m \\ \quad | \quad \gamma_1 \\ \quad \vdots \\ \quad | \quad \gamma_n \end{array}$$

Bei dieser Darstellung sei vorausgesetzt, dass die Strings β_1, \dots, β_m keinen Präfix haben, der allen β_i gemeinsam ist 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:

$$\begin{array}{ll} A \rightarrow \alpha B & B \rightarrow \beta_1 \\ | \quad \gamma_1 & | \quad \beta_2 \\ \vdots & \vdots \\ | \quad \gamma_n & | \quad \beta_m \end{array}$$

Um alle gemeinsamen Präfixe auszuklammern muss dieses Verfahren unter Umständen mehrfach durchgeführt werden. Die nächste Aufgabe gibt dafür ein Beispiel.

Aufgabe 23: Geben Sie eine Links-Faktorisierung für die folgenden Grammatik-Regeln an.

$$\begin{array}{l} A \rightarrow \text{"a"} \text{"b"} U \text{"d"} \\ \quad | \quad \text{"a"} V \text{"b"} \text{"d"} \\ \quad | \quad \text{"a"} \text{"b"} W \\ \quad | \quad \text{"x"} U \\ \quad | \quad \text{"x"} V \end{array}$$

Lösung: Zunächst eliminieren wir das gemeinsame Präfix "a" und führen dazu die neue syntaktische Variable B ein. Wir erhalten:

$$\begin{array}{l} A \rightarrow \text{"a"} B \\ \quad | \quad \text{"x"} U \\ \quad | \quad \text{"x"} V \\ B \rightarrow \text{"b"} U \text{"d"} \\ \quad | \quad V \text{"b"} \text{"d"} \\ \quad | \quad \text{"b"} W \end{array}$$

Nun eliminieren wir das Präfix “x” aus beiden letzten Regeln für A . Wir führen dazu die neue syntaktische Variable C ein. Dann erhalten wir:

$$\begin{array}{lcl} A & \rightarrow & \text{“a” } B \\ & | & \text{“x” } C \\ C & \rightarrow & U \\ & | & V \\ B & \rightarrow & \text{“b” } U \text{ “d”} \\ & | & V \text{ “b” “d”} \\ & | & \text{“b” } W \end{array}$$

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:

$$\begin{array}{lcl} A & \rightarrow & \text{“a” } B \\ & | & \text{“x” } C \\ C & \rightarrow & U \\ & | & V \\ B & \rightarrow & \text{“b” } D \\ & | & V \text{ “b” “d”} \\ D & \rightarrow & U \text{ “d”} \\ & | & W \end{array}$$

□

Bemerkung: Bei dem Parser-Generator ANTLR läßt sich der Lookahead als Option einstellen. Abbildung 9.2 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 9.2: 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 9.3 zeigt die resultierende Grammatik, die sich nun von ANTLR auch mit nur einem Token Lookahead verarbeiten läßt.

```

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 9.3: Die Grammatik aus Abbildung 9.2 in faktorisierte Form.

9.2 First und Follow

Nicht für jede links-faktorierte Grammatik läßt sich ein LL(1)-Parser bauen. Betrachten wir die folgenden Regeln:

$$\begin{aligned}
 A &\rightarrow B \mid C \\
 B &\rightarrow \text{“a” } U \\
 C &\rightarrow \text{“a” } V
 \end{aligned}$$

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 24 (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 $nullabel(A)$ wenn die Variable A als ε -erzeugend nachgewiesen ist. \square

Beispiele:

1. Bei der in Abbildung 9.1 auf Seite 112 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:

$$S \rightarrow A B C$$

$$\begin{aligned}
A &\rightarrow \text{"a"} B \mid A \text{"b"} \mid B C \\
B &\rightarrow \text{"a"} B \mid A \text{"b"} \mid C C \\
C &\rightarrow A B C \mid \varepsilon
\end{aligned}$$

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 erkennen wir S als ε -erzeugend, denn die erste Regel lautet

$$S \rightarrow A B C$$

und hier sind alle Variablen auf der rechten Seite der Regel ε -erzeugend.

Definition 25 ($First()$) 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:

$$First(A) := \{t \in T \mid \exists \gamma \in (V \cup T)^* : A \Rightarrow^* t\gamma\}.$$

Die Definition der Funktion $First()$ kann wie folgt auf Strings aus $(V \cup T)^*$ erweitert werden:

1. $First(\varepsilon) = \{\}$.
2. $First(t\beta) = \{t\}$ falls $t \in T$.
3. $First(A\beta) = \begin{cases} First(A) \cup First(\beta) & \text{falls } A \Rightarrow^* \varepsilon; \\ First(A) & \text{sonst.} \end{cases}$

□

Beispiel: Bei der in Abbildung 9.1 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

$$factor \rightarrow \text{"("} expr \text{"}") \mid NUMBER \mid IDENTIFIER,$$

dass jeder von $Factor$ abgeleitete String entweder mit einer öffnenden Klammer, einer Zahl oder einem Bezeichner beginnt:

$$First(factor) = \{ \text{"("} , NUMBER, IDENTIFIER \}.$$

2. Analog folgt aus den Regeln

$$productRest \rightarrow \text{"*"} factor productRest \mid \text{" /"} factor productRest \mid \varepsilon,$$

dass ein $productRest$ entweder mit dem Zeichen "*" oder " /" beginnt:

$$First(productRest) = \{ \text{"*"} , \text{" /"} \}$$

3. Die Regel für die Variable $product$ lautet

$$product \rightarrow factor productRest.$$

Da die Variable $factor$ nicht ε erzeugend ist, sehen wir, dass die Menge $First(product)$ mit der Menge $First(factor)$ übereinstimmt:

$$First(product) = \{ \text{"("} , NUMBER, IDENTIFIER \}.$$

4. Aus den Regeln

$$exprRest \rightarrow \text{"+"} product exprRest \mid \text{"-"} product exprRest \mid \varepsilon$$

können wir $First(exprRest)$ wie folgt berechnen:

$$First(exprRest) = \{ \text{"+"} , \text{"-"} \}.$$

5. Weiter folgt aus der Regel

$$\text{expr} \rightarrow \text{product } \text{exprRest}$$

und der Tatsache, dass *product* nicht ε -erzeugend ist, dass die $\text{First}(\text{expr})$ mit der Mengen $\text{First}(\text{product})$ übereinstimmt:

$$\text{First}(\text{expr}) = \{ \text{"("}, \text{NUMBER}, \text{IDENTIFIER} \}.$$

6. Schließlich folgt aus den Regeln für die syntaktische Variable *boolExpr* sowie der Tatsache, dass die syntaktische Variable *expr* nicht ε -erzeugend ist, dass $\text{First}(\text{boolExpr})$ mit $\text{First}(\text{expr})$ identisch ist:

$$\text{First}(\text{boolExpr}) = \{ \text{"("}, \text{NUMBER}, \text{IDENTIFIER} \}.$$

□

Definition 26 (Follow()) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik und $A \in V$. Bei der Berechnung von $\text{Follow}()$ 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 \hat{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:

$$\hat{S} \rightarrow S \text{"\$"}.$$

Das Terminal "\$" steht hierbei für das Ende der Eingabe (EOF, *end of file*). Weiter definieren wir

$$\hat{T} := T \cup \{ \text{"\$"} \}.$$

Die so veränderte Grammatik bezeichnen wir als die *augmentierte* Grammatik. Dann definieren wir $\text{Follow}(A)$ als die Menge aller der Token t , die in einer Ableitung auf A folgen können:

$$\text{Follow}(A) := \{ t \in \hat{T} \mid \exists \beta, \gamma \in (V \cup \hat{T})^* : \hat{S} \Rightarrow^* \beta A t \gamma \}.$$

Wenn sich aus dem Start-Symbol \hat{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 $\text{Follow}(A)$. □

Beispiel: Wir untersuchen wieder die in Abbildung 9.1 gezeigte Grammatik für arithmetische Ausdrücke.

1. Aufgrund der neu hinzugefügten Regel

$$\hat{S} \rightarrow \text{boolExpr } \text{"\$"}.$$

muss die Menge $\text{Follow}(\text{boolExpr})$ das Zeichen "\$" enthalten. Da die syntaktische Variable *boolExpr* sonst nirgends in der Grammatik vorkommt, haben wir

$$\text{Follow}(\text{boolExpr}) = \{ \text{"\$"} \}.$$

2. Die Grammatik-Regeln für die syntaktische Variable *boolExpr* zeigen uns zunächst, dass die Menge $\text{Follow}(\text{expr})$ die Zeichen "=", "!", "<=", ">=", "<", ">" enthält. Da *expr* auch am Ende dieser Regeln steht, folgt weiter, dass alle Elemente aus $\text{Follow}(\text{boolExpr})$ auch auf *expr* folgen können, wir haben also auch

$$\text{"\$"} \in \text{Follow}(\text{expr}).$$

Aufgrund der Regel

$$\text{factor} \rightarrow \text{"(" } \text{expr } \text{"})"}$$

muss die Menge $\text{Follow}(\text{expr})$ außerdem das Zeichen ")" enthalten. Also haben wir insgesamt

$$\text{Follow}(\text{expr}) = \{ \text{"="}, \text{"!="}, \text{">="}, \text{"<="}, \text{">"}, \text{"<"}, \text{"\$"}, \text{"})"} \}.$$

3. Aufgrund der Regel

$$\text{expr} \rightarrow \text{product } \text{exprRest}$$

wissen wir, dass alle Terminale, die auf ein *expr* folgen können, auch auf ein *exprRest* folgen können, womit wir schon mal wissen, dass $\text{Follow}(\text{exprRest})$ 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

$$\text{Follow}(\text{exprRest}) = \{ "=", "!= ", ">=", "<=", ">", "<", "\$", ")" \}.$$

4. Die Regeln

$$\text{exprRest} \rightarrow "+" \text{ product exprRest} \mid "-" \text{ product exprRest}$$

zeigen, dass auf ein product alle Elemente aus $\text{First}(\text{exprRest})$ 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

$$\text{Follow}(\text{product}) = \{ "+", "-", "=", "!= ", ">=", "<=", ">", "<", "\$", ")" \}.$$

5. Die Regel

$$\text{product} \rightarrow \text{factor productRest}$$

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

$$\text{Follow}(\text{productRest}) = \{ "+", "-", "=", "!= ", ">=", "<=", ">", "<", "\$", ")" \}.$$

6. Die Regeln

$$\text{productRest} \rightarrow "*" \text{ factor productRest} \mid "/" \text{ factor productRest}$$

zeigen, dass auf ein factor alle Elemente aus $\text{First}(\text{productRest})$ 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

$$\text{Follow}(\text{factor}) = \{ "*", "/", "+", "-", "=", "!= ", ">=", "<=", ">", "<", "\$", ")" \}.$$

□

Das letzte Beispiel zeigt, dass die Berechnung des Prädikats $\text{nullable}()$ und die Berechnung der Mengen $\text{First}(A)$ und $\text{Follow}(A)$ 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 $\text{nullable}()$ und den beiden Funktionen $\text{First}()$ und $\text{Follow}()$ die folgenden Beziehungen:

1. $\forall t \in T : \neg \text{nullable}(t)$.
2. $k = 0 \Rightarrow \text{nullable}(A)$.
3. $(\forall i \in \{1, \dots, k\} : \text{nullable}(Y_i)) \Rightarrow \text{nullable}(A)$.
Setzen wir hier $k = 0$ so sehen wir, dass 2. ein Spezialfall von 3. ist.
4. $\text{First}(Y_1) \subseteq \text{First}(A)$.
5. $(\forall j \in \{1, \dots, i-1\} : \text{nullable}(Y_j)) \Rightarrow \text{First}(Y_i) \subseteq \text{First}(A)$.
6. $\text{Follow}(A) \subseteq \text{Follow}(Y_k)$.
7. $(\forall j \in \{i+1, \dots, k\} : \text{nullable}(Y_j)) \Rightarrow \text{Follow}(A) \subseteq \text{Follow}(Y_i)$.
Setzen wir hier $i = k$ so sehen wir, dass 6. ein Spezialfall von 7. ist.
8. $\forall i \in \{1, \dots, k-1\} : \text{First}(Y_{i+1}) \subseteq \text{Follow}(Y_i)$.
9. $(\forall j \in \{i+1, \dots, l-1\} : \text{nullable}(Y_j)) \Rightarrow \text{First}(Y_l) \subseteq \text{Follow}(Y_i)$.
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.

9.3 $LL(1)$ -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 27 ($LL(1)$ -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

$$A \rightarrow \alpha \quad \text{und} \quad A \rightarrow \beta$$

gibt, die folgenden Bedingungen erfüllt sind:

1. $\neg(\alpha \Rightarrow^* \varepsilon \wedge \beta \Rightarrow^* \varepsilon)$.
Die Rumpfe zweier verschiedener Regeln der selben Variablen dürfen nicht beide das leere Wort ableiten.
2. $First(\alpha) \cap First(\beta) = \{\}$.
Die Ableitungen der Rumpfe zweier verschiedener Regeln der selben Variablen dürfen nicht mit dem selben Token beginnen.
3. $(\beta \Rightarrow^* \varepsilon) \rightarrow First(\alpha) \cap Follow(A) = \{\}$.
Wenn β den leeren String ableitet, dann müssen die Mengen $First(\alpha)$ und $Follow(A)$ disjunkt sein. □

Wir diskutieren nun die Idee, die hinter der obigen Definition steht.

1. Falls das leere Wort sowohl über die Regel

$$A \rightarrow \alpha \quad \text{als auch über} \quad A \rightarrow \beta$$

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 $Follow(A)$ 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 $First(\alpha)$, so verwenden wir die Regel

$$A \rightarrow \alpha.$$

Analog verwenden wir die Regel

$$A \rightarrow \beta,$$

wenn das Lookahead-Token ein Element der Menge $First(\beta)$ ist.

Dieses Rezept funktioniert natürlich nur, wenn die Mengen $First(\alpha)$ und $First(\beta)$ 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 $Follow(A)$, so verwenden wir die Regel

$$A \rightarrow \beta.$$

Damit diese Regel nicht im Widerspruch zu den unter Punkt 2. genannten Regeln steht, benötigen wir die Bedingungen

$$(\beta \Rightarrow^* \varepsilon) \rightarrow \text{First}(\alpha) \cap \text{Follow}(A) = \{\}.$$

Insgesamt versuchen wir also dann mit einer Regel $A \rightarrow \alpha$ zu reduzieren, wenn eine der beiden folgenden Bedingungen erfüllt sind. In diesen Bedingungen bezeichnet *lat* das Lookahead-Token.

1. $\text{lat} \in \text{First}(\alpha)$ oder
2. $\alpha \Rightarrow^* \varepsilon$ und $\text{lat} \in \text{Follow}(\alpha)$.

Bemerkung: Falls eine Grammatik G links-rekursiv ist und die links-rekursiven Regeln nicht überflüssig sind, dann ist klar, dass G keine LL(1)-Grammatik sein kann.

9.4 Implementierung

Abbildung 9.4 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 9.5, 9.6 und 9.7 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 9.4: Die Struktur der Klasse **Grammar**.

```

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 9.5: Die Methode `nullable()`.

9.4.1 Die Berechnung der ε -erzeugenden Variablen

Wir diskutieren zunächst die Berechnung aller ε -erzeugenden Variablen in der in Abbildung 9.5 gezeigten Methode `computeNullable()`. Die Grundidee ist folgende: Falls es für die Variable A eine Regel der Form

$$A \rightarrow Y_1 \cdots Y_k$$

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 $A \rightarrow Y_1 \cdots Y_k$ folglich die Form $A \rightarrow \varepsilon$ hat, dann ist die Menge $\{1, \dots, k\}$ leer und damit sind alle Y_i für $i \in \{1, \dots, k\}$ ε -erzeugend.

In der Methode `computeNullable()` haben wir eine äußere `while`-Schleife, die sich in der Abbildung 9.5 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

$$A \rightarrow Y_1 \cdots Y_k$$

hat, dann entspricht `items` der Liste $[Y_1, \dots, Y_k]$. 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.

9.4.2 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 9.6: Die Methode *computeFirst()*.

Abbildung 9.6 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

$$A \rightarrow Y_1 \cdots Y_k,$$

so wird `head` in Zeile 6 mit A initialisiert und in Zeile 7 wird `items` die Liste $[Y_1, \dots, Y_k]$ 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, \dots, 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}, \dots, Y_k die Berechnung von *First*(A) nicht mehr beeinflussen, 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, \dots, Y_{i-1} sind ε -erzeugend.

In diesem Fall enthält $First(A)$ alle Terminale aus $First(Y_i)$.

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 $First(A)$ ein neues Terminal hinzufügen, dann kann es sein, dass wir auch in weiteren $First$ -Mengen, die $First(A)$ 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.

9.4.3 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             }
23         }
24     }

```

Abbildung 9.7: 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 $A \rightarrow Y_1 \dots Y_k$ gelten.

1. $(\forall j \in \{i + 1, \dots, k\} : nullable(Y_j)) \Rightarrow Follow(A) \subseteq Follow(Y_i)$.

Diese Beziehung wird in den Zeilen 9 bis 22 umgesetzt. Beachten Sie, dass die Beziehung

$$Follow(A) \subseteq Follow(Y_k).$$

ein Spezialfall dieses Falls ist, der von der Implementierung ebenfalls mit abgedeckt wird.

2. $(\forall j \in \{i + 1, \dots, l - 1\} : nullable(Y_j) \wedge Y_l \in T) \Rightarrow Y_l \in Follow(Y_i)$

Diese Beziehung wird in den Zeilen 33 bis 39 umgesetzt.

3. $(\forall j \in \{i + 1, \dots, l - 1\} : nullable(Y_j) \wedge Y_l \in V) \Rightarrow First(Y_l) \subseteq Follow(Y_i)$

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 9.8: Die Methode *computeFollow()*, zweiter Teil.

9.4.4 Berechnung der Parse-Tabelle

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

$$parseTable : V \times T \rightarrow 2^R,$$

eine Funktion, so dass der Aufruf $parseTable(A, t)$ einer syntaktischen Variable A und einem Token t die Menge aller Regeln der Form

$$A \rightarrow \alpha$$

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 $A \rightarrow \alpha$ eine Regel der Grammatik und ist t ein Token aus der Menge $First(\alpha)$, dann ist diese Regel ein Element der Menge $parseTable(A, t)$:

$$(A \rightarrow \alpha) \in R \wedge t \in First(\alpha) \Rightarrow (A \rightarrow \alpha) \in parseTable(A, t).$$

2. Ist $A \rightarrow \alpha$ eine Regel der Grammatik, wobei α ε -erzeugend ist, und ist t ein Token aus der Menge $Follow(A)$, dann ist diese Regel ein Element der Menge $parseTable(A, t)$:

```

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 9.9: Berechnung der Parse-Tabelle.

$$(A \rightarrow \alpha) \in R \wedge \alpha \Rightarrow^* \varepsilon \wedge t \in \text{Follow}(A) \Rightarrow (A \rightarrow \alpha) \in \text{parseTable}(A, t).$$

Abbildung 9.9 zeigt eine Berechnung der Parse-Tabelle.

1. Die `for`-Schleife in den Zeilen 2 bis 10 initialisiert $\text{parseTable}(A, t)$ 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 $A \rightarrow \alpha$ für alle Token t aus der Menge $\text{First}(\alpha)$ in $\text{parseTable}(A, t)$ ein.
3. Falls $\alpha \Rightarrow^* \varepsilon$ gilt, so fügen wir in den Zeilen 20 bis 22 für jedes Token t aus $\text{Follow}(A)$ die Regel $A \rightarrow \alpha$ in die Menge $\text{parseTable}(A, t)$ ein.

Eine Grammatik ist genau dann eine $LL(1)$ -Grammatik, wenn die Mengen $\text{parseTable}(A, t)$ für jede syntaktische Variable A und jedes Token t maximal eine Regel enthält:

$$G \text{ ist } LL(1) \quad \text{g.d.w.} \quad \forall A \in V : \forall t \in T : \text{card}(\text{parseTable}(A, t)) \leq 1.$$

Falls die Menge $\text{parseTable}(A, t)$ 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.

9.5 $LL(k)$ -Grammatiken

Viele interessante Grammatiken sind keine $LL(1)$ -Grammatiken. Abbildung 9.10 zeigt ein Beispiel. Bei dieser Grammatiken werden für arithmetische Ausdrücke auch Funktionsaufrufe der Form

$$f(a_1, \dots, a_n)$$

zugelassen. Dabei ist f ein Funktionszeichen, das syntaktisch nicht von einem Identifier zu unterscheiden ist. Dadurch gibt es zwischen den beiden Regeln

$$factor \rightarrow \text{IDENTIFIER} \quad \text{und} \quad factor \rightarrow \text{IDENTIFIER} "(" \text{argList} ")"$$

einen Konflikt: Soll ein $factor$ geparkt 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

$$factor \rightarrow \text{IDENTIFIER} "(" \text{argList} ")"$$

heranzuziehen, andernfalls muss die Regel

$$factor \rightarrow \text{IDENTIFIER}$$

verwendet werden.

$expr$	\rightarrow	$product \ exprRest$
$exprRest$	\rightarrow	$"+" \ product \ exprRest$
	$ $	$"-" \ product \ exprRest$
	$ $	ε
$product$	\rightarrow	$factor \ productRest$
$productRest$	\rightarrow	$"*" \ factor \ productRest$
	$ $	$/" \ factor \ productRest$
	$ $	ε
$factor$	\rightarrow	$"(" \ expr \ ")"$
	$ $	$Number$
	$ $	$IDENTIFIER$
	$ $	$IDENTIFIER "(" \ argList \ ")"$
$argList$	\rightarrow	$expr \ argsRest$
	$ $	ε
$argsRest$	\rightarrow	$"," \ expr \ argsRest$
	$ $	ε

Abbildung 9.10: 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 konkurrenzfähig sind zu Parsern, die von Bottom-Up-Parser-Generatoren wie *Bison* oder *CUP* erzeugt werden. Wir skizzieren die Grundzüge dieser

Theorie. Als erstes verallgemeinern wir die Definition der Funktion $First()$.

Definition 28 ($First(k, \alpha)$) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$First : \mathbb{N} \times (V \cup T)^* \rightarrow 2^{T^*},$$

so dass $First(k, \alpha)$ 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:

$$First(k, \alpha) := \{x \in T^* \mid \exists y \in T^* : \alpha \Rightarrow^* xy \wedge |x| = k\} \cup \{x \in T^* \mid \alpha \Rightarrow^* x \wedge |x| < k\}. \quad \square$$

Beispiel: Streichen wir zur Vereinfachung in der in Abbildung 9.1 gezeigte Grammatik für arithmetische Ausdrücke die Regel

$$factor \rightarrow \text{NUMBER}$$

sowie die Regeln für $boolExpr$ und kürzen wir weiter das Token IDENTIFIER als ID ab, so erhalten wir beispielsweise:

1. $First(2, expr) = \{ \text{ID}, "(\text{ID}, "((", \text{ID}"+", \text{ID}-", \text{ID}*", \text{ID}/" \}$,
2. $First(2, exprRest) = \{ \varepsilon, "+" \text{ID}, "+(", "- \text{ID}, "-(", \}$,
3. $First(2, product) = \{ \text{ID}, "(\text{ID}, "((", \text{ID}*", \text{ID}/" \}$,
4. $First(2, productRest) = \{ \varepsilon, "*" \text{ID}, "*"(", "/" \text{ID}, "/"(\}$,
5. $First(2, factor) = \{ \text{ID}, "(\text{ID}, "((" \}$.

Definition 29 ($Follow(k, A)$) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Wir definieren eine Funktion

$$Follow : \mathbb{N} \times V \rightarrow 2^{T^*},$$

so dass $Follow(k, A)$ 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-Symbol S ausgeht, auf A folgen können. Formal lautet die Definition:

$$Follow(k, A) := \{x \in T^* \mid \exists \alpha, \gamma \in (T \cup V)^* : S \Rightarrow^* \alpha A \gamma \wedge x \in First(k, \gamma)\}. \quad \square$$

Beispiel: Setzen wir das letzte Beispiel sinngemäß fort, so erhalten wir:

1. $Follow(2, expr) = \{ \varepsilon, ")", ")))" \}$,
2. $Follow(2, exprRest) = \{ \varepsilon, ")", ")))" \}$,
3. $Follow(2, product) = \{ \varepsilon, "+" \text{ID}, "+(", "- \text{ID}, "-(", ")", ")))" \}$,
4. $Follow(2, productRest) = \{ \varepsilon, "+" \text{ID}, "+(", "- \text{ID}, "-(", ")", ")))" \}$,
5. $Follow(2, factor) = \{ \varepsilon, ")", ")))", "+" \text{ID}, "+(", "- \text{ID}, "-(", "*" \text{ID}, "*"(", "/ \text{ID}, "/"(\}$.

Definition 30 (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

$$A \rightarrow \beta \quad \text{und} \quad A \rightarrow \gamma$$

aus der Menge R die Bedingung

$$\forall \sigma, \tau \in Follow(k, A) : First(k, \beta\sigma) \cap First(k, \gamma\tau) = \{\}$$

erfüllt ist. □

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 $Follow(k, A)$ sind. Für eine Regel

$$A \rightarrow \beta$$

und einen String $\sigma \in Follow(k, A)$ enthält die Menge $First(k, \beta\sigma)$ alle die Strings der Länge $\leq k$, die in einer Ableitung von A , welche die Regel $A \rightarrow \beta$ 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 bestimmten 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.

9.5.1 Berechnung von $First()$ und $Follow()$

In diesem Abschnitt zeigen wir, wie die Funktionen $First(k, \alpha)$ und $Follow(k, A)$ berechnet werden können. Dazu benötigen wir verschiedene Hilfsfunktionen, die wir vorab definieren.

1. Die Funktion $prefix(k, w)$ 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:

$$prefix(k, w) = \begin{cases} w[1:k] & \text{falls } k < |w|; \\ w & \text{sonst.} \end{cases}$$

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 Präfix der Länge k :

$$v +_k w = prefix(k, vw).$$

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

$$\begin{aligned} & \{\varepsilon, \text{"a"}, \text{"ab"}, \text{"abc"}\} +_2 \{\varepsilon, \text{"x"}, \text{"yx"}\} \\ &= \{\varepsilon, \text{"a"}, \text{"ab"}, \text{"x"}, \text{"yx"}, \text{"ax"}, \text{"ay"}\}. \end{aligned}$$

□

Die Berechnung von $First(k, \alpha)$ für $\alpha \in (V \cup T)^*$ wird auf die Berechnung von $First(k, X)$ mit $X \in V \cup T$ zurück geführt, denn es gilt

$$First(k, X_1 X_2 \cdots X_n) = First(k, X_1) +_k First(k, X_2) +_k \cdots +_k First(k, X_n).$$

Für ein Terminal $t \in T$ gilt offenbar

$$First(k, t) = t.$$

Die Berechnung der Mengen $First(k, A)$ für eine syntaktische Variable $A \in V$ erfolgt iterativ über folgenden Algorithmus:

1. Zunächst werden alle Mengen $First(k, A)$ mit der leeren Menge initialisiert:

$$First(k, A) := \{\}.$$

2. Anschließend wird für jede Grammatik-Regel der Form

$$A \rightarrow \alpha$$

die Menge $First(k, A)$ wie folgt erweitert:

$$First(k, A) := First(k, A) \cup First(k, \alpha).$$

3. Der zweite Schritt wird in einer Schleife solange durchgeführt, bis sich keine der Mengen $First(k, A)$ mehr durch die Hinzunahme von $First(k, \alpha)$ ändert.

Sind die Mengen $First(k, A)$ berechnet, so können wir anschließend die Mengen $Follow(k, A)$ für alle syntaktischen Variablen berechnen. Auch die Berechnung der Mengen $Follow(k, A)$ ist iterativ. Sie erfolgt nach dem folgenden Schema:

1. Zunächst werden alle Mengen $Follow(k, A)$ mit der leeren Menge initialisiert:

$$Follow(k, A) = \{\}.$$

Anschließend setzen wir für das Start-Symbol S der Grammatik

$$Follow(k, S) = \{\$ \}.$$

Hier steht “\$” für das Ende der Eingabe. Die Idee ist, dass hinter dem Start-Symbol keine weitere Eingabe mehr kommen kann. Beachten Sie, dass in diesem Fall der String “\$” nicht aus k Zeichen besteht, sondern nur aus einem Zeichen.

2. Für jede Grammatik-Regel der Form

$$A \rightarrow Y_1 Y_2 \cdots Y_l,$$

für die Y_l eine syntaktische Variable ist, erweitern wir die Menge $Follow(k, Y_l)$ wie folgt:

$$Follow(k, Y_l) := Follow(k, Y_l) \cup Follow(k, A),$$

denn alles, was auf ein A folgen kann, kann auch auf ein Y_l folgen.

3. Für jede Grammatik-Regel der Form

$$A \rightarrow Y_1 Y_2 \cdots Y_i Y_{i+1} \cdots Y_l,$$

und jeden Index $i \in \{1, \dots, l-1\}$, für den Y_i eine syntaktische Variable ist, erweitern wir die Menge $Follow(k, Y_i)$ wie folgt:

$$Follow(k, Y_i) := Follow(k, Y_i) \cup (First(k, Y_{i+1} \cdots Y_l) +_k Follow(k, A)).$$

Der Grund, warum wir hier noch die Menge $Follow(k, A)$ anhängen ist der, dass die Strings aus der Menge $First(k, Y_{i+1} \cdots Y_l)$ eventuell kürzer als k sind. In diesem Fall müssen noch die Präfixe von $Follow(k, A)$ angehängt werden.

Bemerkung: Beachten Sie, dass der zweite Schritt ein Spezialfall des dritten Schritts ist, denn wenn wir im dritten Schritt $i := l$ setzen, dann ist der String $Y_{i+1} \cdots Y_l$ leer und somit enthält die Menge $First(k, Y_{i+1} \cdots Y_l)$ dann nur den leeren String ε , so dass der Ausdruck

$$(First(k, Y_{i+1} \cdots Y_l) +_k Follow(k, A)) \quad \text{zu} \quad Follow(k, A)$$

vereinfacht werden kann. Bei der Implementierung werden wir daher nur den dritten Schritt umsetzen.

4. Der zweite und der dritte Schritt werden in einer Schleife solange durchgeführt, bis sich keine der Mengen $Follow(k, A)$ mehr ändert.

9.5.2 Computing the Parse Table

Once we have computed the sets $First(k, A)$ and $Follow(k, A)$ for all variables A and some fixed value $k \in \mathbb{N}$, the next step is to compute the *parse table*. Formally, given a context free grammar $G = \langle V, T, R, S \rangle$ and a positive natural number k , we have a function

$$parseTable_k : V \times T^k \rightarrow 2^R$$

that takes a variable $A \in V$ and a token string $s \in T^*$ such that $|s| \leq k$ and it computes the set of all those grammar rules, that can be used to parse an A when the next k lookahead tokens are given as the string s . For any given $k \in \mathbb{N}$ with $k > 0$ this function is defined as

$$(A \rightarrow \alpha) \in parseTable_k(A, s) \stackrel{\text{def}}{\iff} s \in First(k, \alpha) +_k Follow(k, A).$$

The reasoning behind this definition is as follows: When we try to parse an A using the grammar rule

$$A \rightarrow \alpha,$$

and knowing the string s consisting of the next k tokens, then s might be part of the string derived from α or, if the string derived from α has a length shorter than k , then a suffix of the string s might also be a part of $Follow(k, A)$.

Once we have computed the parse table it is easy to check whether the grammar G is an $LL(k)$ grammar: G is an $LL(k)$ -grammar if and only if we have

$$card(parseTable(A, s)) \leq 1.$$

Here, for a given set M , the expression $card(M)$ denotes the number of elements of M . The reasoning for demanding the set $parseTable(A, S)$ to contain at most one element is that given a variable A and a string s of lookahead tokens the parse table must give us at most one rule which can be used to proceed. If the parse table entry for A and s instead contains more than one rule we don't know which of these rules should be applied and we would have to try all rules. This would effectively result in backtracking which, in general, is computationally much more expensive than $LL(k)$ -parsing.

9.6 Implementing the Computation of $LL(k)$ Parse Tables

The computation of the functions $First$ and $Follow$ is essentially set based. Therefore it is straightforward to implement these function in SETLX.

Figure 9.11 shows the implementation of the function $First(k, A)$ in SETLX. The function $computeFirst(k, rules)$ takes two parameters: The first parameter k specifies the number of lookahead tokens, while the second parameter $rules$ is the set of all rules of the given grammar. The function $computeFirst(k, rules)$ computes a binary relation **first** representing the function $First$, i.e. **first** is a set of pairs of the form $[A, s]$ such that A is a variable of the grammar and s is a set of token strings of length k . We will have

$$[A, s] \in \mathbf{first} \iff s = First(k, A).$$

Therefore, in effect the function $First$ is represented by the binary relation **first**. This binary relation is computed by a fixpoint iteration. Initially, the set $First(k, A)$ is empty for every variable A . This initialization of the binary relation **first** is done by the function `initializeMap` shown in Figure 9.12 on page 131. This function just iterates over the set of all rules of the grammar. For every rule of the form $A \rightarrow \alpha$ it initializes $First(k, A)$ to the empty set.

Next, the while loop in line 4 – 16 of Figure 9.11 performs a fixpoint iteration. In this fixpoint iteration, we compute for any rule

$$a \rightarrow body$$

in our grammar the value of $First(k, body)$. This is done via the function `firstList` shown in Figure 9.13 on page 132. These values are then added to the current value of $First(k, a)$. In case that we have actually found a new string in this set, the variable **change** is set to true and the **while** loop keeps going.

The function `firstList` shown in Figure 9.13 extends the function $First$ from variables to strings containing variables and tokens. It uses the function `unionK` that is defined in Figure 9.16. Given two sets of tokens M and

```

1  computeFirst := procedure(k, rules) {
2      first := initializeMap(rules, {});
3      change := true;
4      while (change) {
5          change := false;
6          for (rule in rules) {
7              match (rule) {
8                  case Rule(a, body):
9                      old := first[a];
10                     new := firstList(k, body, first);
11                     if (new <= old) { continue; }
12                     change := true;
13                     first[a] := old + new;
14             }
15         }
16     }
17     return first;
18 };

```

Abbildung 9.11: Computing $First(k, A)$.

```

1  initializeMap := procedure(rules, s) {
2      map := {};
3      for (rule in rules) {
4          match (rule) {
5              case Rule(a, _):
6                  map[a] := s;
7          }
8      }
9      return map;
10 };

```

Abbildung 9.12: Initializing a binary relation.

N we have that

$$\text{unionK}(M, N, k) = M +_k N.$$

The definition of $firstList(k, \alpha)$ is by induction on α .

1. If α is empty, then

$$firstList(k, \alpha) = \{\varepsilon\}.$$

Since the implementation represent strings as lists, ε is represented as the empty list `[]`.

2. If $\alpha = A\beta$ where A is a variable and β is the remaining list of variables and tokens, then

$$firstList(k, A\beta) = First(A) +_k First(\beta).$$

3. If $\alpha = t\beta$ where t is a token and β is the remaining list of variables and tokens, then

$$firstList(k, t\beta) = \{t\} +_k First(\beta).$$

Figure 9.14 on page 132 shows the implementation of the function $Follow(k, A)$. This function is implemented via the function `computeFollow`. This function gets four arguments.

```

1  firstList := procedure(k, alpha, first) {
2      match (alpha) {
3          case []:
4              return { [] };
5          case [ Var(v) | r ]:
6              firstV := first[v];
7              firstR := firstList(k, r, first);
8              result := unionK(firstV, firstR, k);
9              return unionK(firstV, firstR, k);
10         case [ Token(t) | r ]:
11             firstR := firstList(k, r, first);
12             return unionK({ [t] }, firstR, k);
13     }
14 };

```

Abbildung 9.13: The implementation of $\text{firstList}(k, \alpha)$.

```

1  computeFollow := procedure(k, rules, s, first) {
2      follow := initializeMap(rules, {});
3      follow[s] := { [ "$" ] };
4      change := true;
5      while (change) {
6          change := false;
7          for (rule in rules) {
8              match (rule) {
9                  case Rule(a, body):
10                     for (i in [1 .. #body]) {
11                         match (body[i]) {
12                             case Var(yi):
13                                 old := follow[yi];
14                                 tail := firstList(k, body[i+1 ..], first);
15                                 new := unionK(tail, follow[a], k);
16                                 if (new <= old) { continue; }
17                                 change := true;
18                                 follow[yi] := old + new;
19                             case Token(_):
20                                 continue;
21                         }
22                     }
23             }
24         }
25     }
26     return follow;
27 };

```

Abbildung 9.14: Computing $\text{Follow}(k, A)$.

1. k specifies the length of the follow string that we want to compute.
2. `rules` is the set of rules of the grammar.

3. s is the start symbol of the grammar.
4. **first** is a binary relation coding the function *First*, i.e. for any syntactical variable A we have **first** $[A] = First(k, A)$.

The function **computeFollow** represents this function via the binary relation **follow** which is initialized so that initially $Follow(k, A)$ is the empty set. Since s is the start symbol, we know that the *end-of-file* token “\$” is a member of the follow set of s . Therefore, line 3 of Figure 9.14 sets **follow** $[s]$ to contain the list [“\$”].

After that, the binary relation **follow** is computed via a fixpoint iteration. Whenever we have a rule of the form

$$A \rightarrow Y_1 \cdots Y_i Y_{i+1} \cdots Y_n,$$

such that Y_i is a variable we update the set $Follow(k, Y_i)$ by adding the tokens from the set

$$First(k, Y_{i+1} \cdots Y_n) +_k Follow(A).$$

This update of these follow sets is done as long as any of these sets changes.

```

1  parseTable := procedure(f, k) {
2      [rules, s] := readGrammar(f);
3      pt      := {};
4      first   := computeFirst(k, rules);
5      follow  := computeFollow(k, rules, s, first);
6      for (rule in rules) {
7          match (rule) {
8              case Rule(a, alpha):
9                  for (s in unionK(firstList(k, alpha, first), follow[a], k)) {
10                     rulesAS := pt[[a, s]];
11                     if (rulesAS == om) {
12                         rulesAS := {};
13                     }
14                     pt[[a,s]] := rulesAS + { rule };
15                 }
16             }
17         }
18     return pt;
19 };

```

Abbildung 9.15: Computing the parse table.

Figure 9.15 shows the computation of the parse table. The function **parseTable** gets two arguments:

1. f is the name of a file containing a grammar.
2. k is the number of lookahead tokens.

The implementation works as follows:

1. Initially, the given file f is opened and the grammar contained in this file is parsed using the function **readGrammar**. For reasons of space, the implementation of this function is not shown in these lecture notes but it can be found on my web page.

The function **readGrammar** returns a pair: The first component of this pair is the set of grammar rules, while the second component is the start symbol of the grammar.

2. The resulting parse table is represented as a binary relation. We use the variable **pt** to store this relation. Initially, the parse table is empty. Therefore, **pt** is initialized as the empty set in line 3.

3. Next, both the functions *First* and *Follow* are computed in line 4 and 5. These functions are represented through the binary relations **first** and **follow**.
4. Then, the **for** loop in line 6 iterates over all rules of the grammar. Given a rule $A \rightarrow \alpha$, the loop computes all prefixes s of the set

$$First(k, \alpha) +_k Follow(k, A).$$

Next, for any prefix s in the set $First(k, \alpha) +_k Follow(k, A)$, the rule $A \rightarrow \alpha$ is added to the set $parseTable(A, s)$. Since the function $parseTable$ is represented as the binary relation **pt**, the value of $parseTable(A, s)$ is represented by **pt**[[**a**, **s**]]. The effect of this is to implementat the equivalence

$$(A \rightarrow \alpha) \in parseTable_k(A, s) \stackrel{\text{def}}{\iff} s \in First(k, \alpha) +_k Follow(k, A).$$

Note that have have to take care of the case that **pt**[[**a**, **s**]] is still undefined. Since SETLX represents an undefined value as **om**, we check in line 11 whether **pt**[[**a**, **s**]] == **om**. In this case, we have to initialize it with the empty set.

```

1  prefixK := procedure(s, k) {
2      if (#s <= k) {
3          return s;
4      }
5      return s[1..k];
6  };
7  addK := procedure(u, v, k) {
8      return prefixK(u + v, k);
9  };
10 unionK := procedure(s, t, k) {
11     return { addK(u, v, k) : u in s, v in t };
12 };

```

Abbildung 9.16: The implementation of some auxilliary functions.

Finally, Figure 9.16 shows the implementation of some auxilliary functions.

1. **prefixK**(s, k) takes a list s and computes the maximal prefix of s that contains at most k elements.
2. **addK**(u, v, k) takes two lists u and v as input. These lists are concatenated and then the prefix of lenght k of this concatenation is returned.
3. **unionK**(s, t, k) takes two sets of lists s and t and computes the set $s +_k t$.

Kapitel 10

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.

10.1 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.

10.1.1 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; }
```

Abbildung 10.1 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

¹<https://javacc.dev.java.net/>

```

1  grammar DeclOrDef;
2
3  options {                // Suppress these lines
4      k = 8;                // to let ANTLR create
5  }                        // an  $LL(k)$ -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: '0' | ('1'..'9') ('0'..'9')*;

```

Abbildung 10.1: Eine Grammatik für Deklarationen und Definitionen

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

```

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: '0' | ('1'..'9') ('0'..'9')*;

```

Abbildung 10.2: Eine faktorisierte Grammatik für Deklarationen und Definitionen.

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.

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 10.2 zeigt, wie sich die Unterscheidung von Deklarationen und Definitionen durch Faktorisierung der Grammatik-Regeln lösen läßt.

Klassen versus Schnittstellen

Abbildung 10.3 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 10.3 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 10.4 ist dies gezeigt. Die einfachste Möglichkeit besteht allerdings wieder darin, *Antlr* einen $LL(*)$ -Parser erzeugen zu lassen. Diese erreichen wir, in dem wir

```
options { k = 5; }
```

durch

```
options { k = *; }
```

```

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 10.3: Grammatik für Klassen und Schnittstellen.

ersetzen, oder aber die Spezifikation der Look-Ahead-Tiefe k ganz weglassen.

```

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 10.4: Eine faktorisierte Grammatik für Klassen und Schnittstellen.

10.1.2 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 10.5 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 .
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.

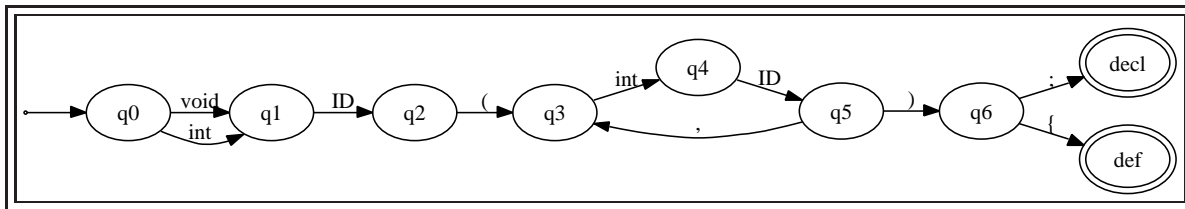


Abbildung 10.5: Ein deterministischer endlicher Automat zur Unterscheidung zwischen Deklarationen und Definitionen.

Implementierung der LL(*)-Sprachen

Wir diskutieren jetzt, wie ANTLR bei der Erstellung des endlichen Automaten für die Auswahl einer Regel vorgeht. Ist

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$$

eine Grammatik-Regel, bei der eine Look-Ahead-Tiefe von 1 nicht ausreicht um zu bestimmen, welche der k Regeln $A \rightarrow \alpha_i$ mit $i = 1 \dots k$ 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 10.1 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 10.6 und 10.7 gezeigt.

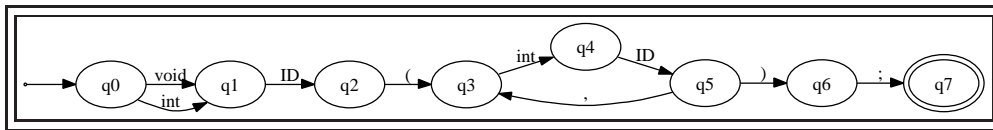


Abbildung 10.6: Ein endlicher Automat zur Erkennung von Deklarationen.

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

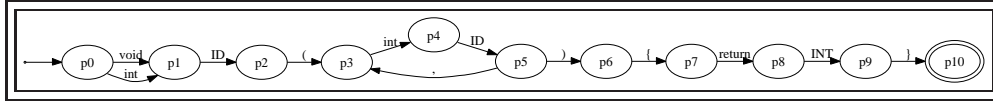


Abbildung 10.7: Ein endlicher Automat zur Erkennung von Definitionen.

den Start-Zuständen q_0 und p_0 gibt. Für diesen, zunächst nicht-deterministischen Automaten führen wir die Teilmengen-Konstruktion durch und erhalten dann wieder einen deterministischen endlichen Automaten. Der resultierende Automat ist in Abbildung 10.8 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 10.1 definierte Sprache eine LL($*$)-Sprache ist.

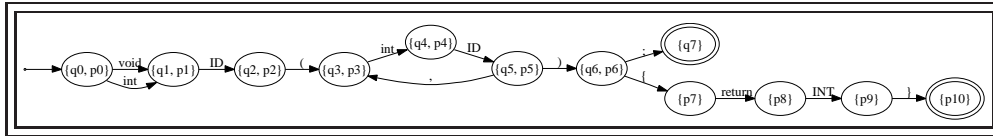


Abbildung 10.8: Der zusammengesetzte endliche Automat.

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 10.1 wie in Abbildung 10.9 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 andernfalls 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.

10.2 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 Möglichkeit gestattet, mit Hilfe des Schlüsselworts `typedef` neue Typ-Bezeichner zu definieren. Ein Beispiel sehen Sie in Abbildung 10.10: 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 Tatsache, dass `uint` 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  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 10.9: Rekursive Grammatik für Deklarationen und Definitionen

```

1  int (uint);
2
3  typedef unsigned int uint;
4
5  uint(x);

```

Abbildung 10.10: 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 10.11 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

$$decl \rightarrow type \text{ "(" identifier "}"$$

als Deklaration interpretiert werden, andererseits zeigt die Grammatik-Regel

$$stmnt \rightarrow identifier \text{ "(" identifier "}" ,$$

dass auch eine Interpretation als Funktions-Aufruf möglich ist.

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

```

1  grammar cGrammar;
2
3  declOrType
4      : decl
5      | typeDef
6      | stmtnt
7      ;
8
9  decl: type identifier ';'
10     | type '(' identifier ')' ';'
11     ;
12
13 typeDef: 'typedef' type identifier ';' ;
14
15 stmtnt: identifier '(' identifier ')' ';' ;
16
17 type: 'unsigned'
18     | 'int'
19     | 'unsigned' 'int'
20     | typeid
21     ;
22
23 typeid: ID;
24 identifier: ID;
25
26 ID : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
27 WS : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 10.11: Eine Grammatik für Deklarationen und Befehle.

verschiedene Arten ableitbar ist.

Abbildung 10.12 zeigt, wie die Mehrdeutigkeit der Grammatik mit Hilfe eines semantischen Prädikats umgangen werden kann.

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

typeid → ID

wird nun nur dann angewendet, wenn dies der Fall ist. Andernfalls wird ein ID mit Hilfe der Regel

identifier → ID

geparst.

```

1
2  grammar cGrammar2;
3
4  @header {
5      import java.util.Set;
6      import java.util.TreeSet;
7  }
8
9  @members {
10     Set<String> mTypeNames = new TreeSet<String>();
11 }
12
13 declOrType
14     : decl
15     | typeDef
16     | stmtnt
17     ;
18
19 decl: type identifier ';'
20     | type '(' identifier ')' ';'
21     ;
22
23 typeDef: 'typedef' type identifier ';' { mTypeNames.add($identifier.text); };
24
25 stmtnt: identifier '(' identifier ')' ';'
26
27 type: 'unsigned'
28     | 'int'
29     | 'unsigned' 'int'
30     | typeid
31     ;
32
33 typeid: { mTypeNames.contains(input.LT(1).getText()); }? ID;
34
35 identifier: ID;
36
37 ID      : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
38 WS      : (' '\t'|\n'|\r') { skip(); };

```

Abbildung 10.12: Eine Grammatik mit semantischen Prädikat.

Die allgemeine Form eines semantischen Prädikats in ANTLR ist

$$A \rightarrow \{Code\}? Y_1 \cdots Y_k.$$

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.

10.3 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 $i = 3, \dots, n$ 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 meistens vermieden werden kann.

10.3.1 Das Dangling-Else-Problem

Als motivierendes Beispiel betrachten wir das sogenannte *Dangling-Else-Problem*, das in der in Abbildung 10.13 auf Seite 146 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 folgenden 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.

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.

```

1  grammar dangling;
2
3  prog  : stmtnt+;
4
5  stmtnt : 'if' '(' boolExp ')' stmtnt 'else' stmtnt
6          | 'if' '(' boolExp ')' stmtnt
7          | 'while' '(' boolExp ')' stmtnt
8          | '{' stmtnt* '}'
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 10.13: Fragment einer Grammatik für die Sprache C

Lösung des Dangling-Else-Problem durch Backtracking

Abbildung 10.14 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 *stmtnt* einen backtrackenden Parser erzeugen soll. Hier ist es nun wichtig, dass die Regel

$$stmtnt \rightarrow \text{"if"} \text{"(" } boolExp \text{"("} stmtnt \text{"else"} stmtnt$$

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

$$stmtnt \rightarrow \text{"if"} \text{"(" } boolExp \text{"("} stmtnt$$

zur Anwendung. Dadurch wird sichergestellt, dass jedes Auftreten des Schlüsselworts **else** dem unmittelbar vorangehenden **if** zugeordnet wird.

Lösung des Dangling-Else-Problem durch ein syntakisches Prädikat

Abbildung 10.15 zeigt, wie sich das Dangling-Else-Problem durch die Verwendung eines syntaktischen Prädikats lösen lässt. In Zeile 5 und 6 haben wir geschrieben:

```

stmtnt : ('if' '(' boolExp ')' stmtnt 'else' stmtnt)=>
        'if' '(' boolExp ')' stmtnt 'else' stmtnt

```

Der Ausdruck `("if" "(" boolExp ")" stmtnt "else" stmtnt)=>` ist hier ein sogenanntes *syntakisches Prädikat*. Bevor der Parser versucht, ein *expr* mit der Regel

```
1  grammar danglingBacktrack;
2
3  prog  : stmtnt+;
4
5  stmtnt
6      options {
7          backtrack = true;
8      }
9      : 'if' '(' boolExp ')' stmtnt 'else' stmtnt
10     | 'if' '(' boolExp ')' stmtnt
11     | 'while' '(' boolExp ')' stmtnt
12     | '{' stmtnt* '}'
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 10.14: Lösen des Dangling-Else-Problems durch Backtracking.

$$stmtnt \rightarrow \text{“if” “(” } boolExp \text{ “)” } stmtnt \text{ “else” } stmtnt$$

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  : stmtnt+;
4
5  stmtnt : ('if' '(' boolExp ')' stmtnt 'else' stmtnt)=>
6          'if' '(' boolExp ')' stmtnt 'else' stmtnt
7          | 'if' '(' boolExp ')' stmtnt
8          | 'while' '(' boolExp ')' stmtnt
9          | '{' stmtnt* '}'
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 10.15: Lösen des Dangling-Else-Problems durch ein syntaktisches Prädikat.

Die allgemeine Form eines syntaktischen Prädikats ist

$$\begin{array}{lcl}
 A & \rightarrow & "(" \alpha_1 ")" \Rightarrow \beta_1 \\
 & | & "(" \alpha_2 ")" \Rightarrow \beta_2 \\
 & \vdots & \\
 & | & "(" \alpha_{n-1} ")" \Rightarrow \beta_{n-1} \\
 & | & "(" \alpha_n ")" \Rightarrow \beta_n
 \end{array}$$

Hierbei sind α_i und β_i für $i = 1, \dots, n$ Folgen von Terminalen und syntaktischen 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 erkannt wird, die Regel $A \rightarrow \beta_i$ anzuwenden.

Vergleichen wir die Möglichkeiten, die syntaktische Prädikate bieten, mit den durch Backtracking gebotenen Möglichkeiten, so stellen wir fest, dass Backtracking zwar einfacher anzuwenden ist, denn es ist lediglich über die Option einzuschalten, dass andererseits der Parser aber durch die Verwendung von syntaktischen 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 syntaktischen Prädikaten oder Backtracking nur dann Gebrauch zu machen, wenn durch eine gründliche Analyse der Grammatik sichergestellt ist, dass durch die Verwendung solcher Prädikate keine unbeabsichtigten Ambivalenzen der Grammatik verschleiert werden.

Kapitel 11

Interpreter

In diesem Kapitel erstellen wir mit Hilfe des Parser-Generators ANTLR einen Interpreter für eine einfache Programmiersprache. Abbildung 11.2 zeigt die EBNF-Grammatik dieser Programmier-Sprache. Die Befehle dieser Sprache sind Zuweisungen, Print-Befehle, `if`-Abfragen, sowie `while`-Schleifen. Abbildung 11.1 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 11.1: 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 11.3 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 11.4 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 im wesentlichen die Deklaration der abstrakten Methode `execute()`. Diese Methode können wir später benutzen, um einzelne Befehle ausführen. Zusätzlich speichern wir hier das Flag `isInteractive` als statische Variable. Mit diesem Flag steuern wir, ob der Interpreter interaktiv in einer Kommandozeile betrieben wird, oder ob im Batchmode eine Datei abgearbeitet werden soll. Außerdem haben wir in der Klasse `Statement` noch die statische Methode `prompt()`. Diese wird nur dann benutzt, wenn der Interpreter interaktiv von der Kommandozeile aus betrieben wird. In diesem Fall gibt die Methode einen Prompt aus.

```

stmntList → statement+

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 11.2: EBNF-Grammatik für eine einfache Programmier-Sprache.

Abbildung 11.5 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 11.9 auf Seite 154 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 11.3: Die EBNF-Grammatik nach Beseitigung der Links-Rekursion.

```

1  public abstract class Statement {
2      static boolean isInteractive = false;
3
4      public abstract void execute();
5
6      static void prompt() {
7          if (isInteractive) {
8              System.out.print("SL> ");
9              System.out.flush();
10         }
11     }
12 }

```

Abbildung 11.4: Die abstrakte Klasse `Statement`

Abbildung 11.6 zeigt die Implementierung der Klasse `Read`. Diese Klasse stellt eine Zuweisung der Form

```
var = read();
```

```
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 }
```

Abbildung 11.5: 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          System.out.print("> "); // write prompt
9          System.out.flush();
10         Scanner scanner = new Scanner(System.in);
11         Double value    = scanner.nextDouble();
12         Expr.sValueTable.put(mLhs.mName, value);
13     }
14 }
```

Abbildung 11.6: Die Klasse `Read`.

dar. Daher besitzt diese Klasse eine Member-Variable `mVar`, in welcher das Ergebnis der Lese-Operation abgespeichert wird. Die Methode `execute()` gibt zunächst den String `>` als Eingabe-Aufforderung aus. Anschließend wird ein Objekt der in *Java* vordefinierten Klasse `Scanner` erzeugt. Diese Klasse stellt die Methode `nextDouble()` zur Verfügung, mit deren Hilfe eine Fließkomma-Zahl eingelesen werden kann. Diese wird dann in der Symboltabelle unter dem Namen der Variablen, der auf der linken Seite der Zuweisung steht, abgespeichert.

```

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> stmtList) {
8          mCond      = cond;
9          mStmntList = stmtList;
10     }
11     public void execute() {
12         while (mCond.eval()) {
13             for (Statement stmt: mStmntList) {
14                 stmt.execute();
15             }
16         }
17     }
18 }

```

Abbildung 11.7: Die Klasse `While`.

Von den übrigen Klassen zur Darstellung von Befehlen diskutieren wir noch die Klasse `While`, die in Abbildung 11.7 gezeigt wird. Diese Klasse stellt einen Befehl der Form

`while (b) { 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.

```

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 }

```

Abbildung 11.8: Klassen `BoolExpr` und `Equal`

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` dargestellt werden, die beide von der Klasse `BoolExpr` abgeleitet sind. Abbildung 11.8 zeigt die Klassen `BoolExpr` und `Equal`. Die Klasse `BoolExpr` hat die beiden Member-Variablen `mLhs` und `mRhs` und repräsentiert die Gleichung

$$mLhs == mRhs.$$

Um diese Gleichung auszuwerten, werden rekursiv die linke und die rechte Seite der Gleichung, die in `mLhs` und `mRhs` gespeichert sind, ausgewertet. Anschließend wird das Ergebnis dieser Auswertung zurück gegeben. Die Klasse `LessThan` ist analog zur Klasse `Equal` aufgebaut und wird daher nicht gezeigt.

```

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 11.9: 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 11.9 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, die wir jetzt der Reihe nach diskutieren. Abbildung 11.10 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.

Abbildung 11.11 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.

```

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 }

```

Abbildung 11.10: 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 }

```

Abbildung 11.11: Die Klasse Variable.

Abbildung 11.12 zeigt das Treiber-Programm, das den von ANTLR erzeugten Parser einbindet. Das Programm soll später wahlweise in der Form

```
java SLInterpreter file1 ... filen
```

oder einfach als

```
java SLInterpreter
```

aufgerufen werden. Im ersten Fall bezeichnet $file_i$ für $i = 1, \dots, n$ jeweils eine Datei, die ein auszuführendes enthält. Im zweiten Fall, oder 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 `InputReader` wird dazu benötigt, um von der Standardeingabe zu lesen und das Gelesene anschließend als `StringBufferInputStream` zurück zu geben. Wir werden diese Klasse gleich genauer diskutieren. Sowohl die Methode `parseFile()` als auch die Methode `parseInteractive()` dienen beide nur dazu, ein Objekt der Klasse `ANTLRInputStream` zu erzeugen. Im ersten Fall wird dieses Objekt aus der zu lesenden Datei erzeugt, im zweiten Fall benutzen wir dazu den noch zu diskutierenden `InputReader`. In beiden Fällen wird schließlich die Methode `parseAndExecute()` aufgerufen, deren Aufgabe es ist, die einzelnen Befehle zu erkennen und auszuführen.

```
1  import org.antlr.runtime.*;
2  import java.util.List;
3  import java.io.*;
4
5  public class SLInterpreter {
6      public static void main(String[] args) throws Exception {
7          for (String file: args) {
8              if (!file.equals("-")) {
9                  parseFile(file);
10             } else {
11                 parseInteractive();
12             }
13         }
14         if (args.length == 0) {
15             parseInteractive();
16         }
17     }
18     private static void parseFile(String fileName) throws Exception {
19         try {
20             ANTLRStringStream ss = new ANTLRFileStream(fileName);
21             parseAndExecute(ss);
22         } catch (IOException e) {
23             System.err.println("File " + fileName + " could not be read.");
24         }
25     }
26     private static void parseInteractive() throws Exception {
27         Statement.isInteractive = true;
28         Statement.prompt();
29         while (true) {
30             InputStream      stream = InputReader.getStream();
31             ANTLRInputStream input  = new ANTLRInputStream(stream);
32             parseAndExecute(input);
33         }
34     }
35     private static void parseAndExecute(ANTLRStringStream stream)
36         throws Exception
37     {
38         SimpleLexer      lexer = new SimpleLexer(stream);
39         CommonTokenStream ts    = new CommonTokenStream(lexer);
40         SimpleParser      parser = new SimpleParser(ts);
41         parser.program();
42     }
43 }
```

Abbildung 11.12: Die Klasse SLInterpreter.

Abbildung 11.13 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR. Der Parser liest in Zeile 9 eine nicht-leere Folge von Befehlen, die sofort nach dem Einlesen ausgeführt werden. Die übrigen Grammatik-Regeln erzeugen jeweils einen abstrakten Syntax-Baum der erkannten Eingabe. So liefert beispielsweise die Regel für die syntaktische Variable `statement` als Ergebnis ein Objekt der abstrakten Klasse `Statement` zurück. Wir diskutieren nur den Fall, dass es sich bei dem Statement um eine `while`-Schleife handelt, denn die anderen Fälle sind analog. Zunächst wird in Zeile 13 in der Variablen `stmts` eine leere Liste von `Statements` angelegt. Diese Liste enthält später alle Befehle, die im Rumpf der `while`-Schleife stehen. Anschließend wird in Zeile 24 das Schlüsselwort `“while”` zusammen mit der Bedingung erkannt. Dann werden der Reihe nach alle Befehle, die sich im Rumpf der Schleife befinden, der Liste `stmts` hinzugefügt. Nach dem Lesen der schließenden geschweiften Klammer wird als Rückgabewert ein Objekt der Klasse `While` erzeugt und zurückgegeben.

Die Spezifikation der Token ist in Abbildung 11.14 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.

Abbildung 11.15 zeigt die Klasse `InputReader`, die dann benutzt wird, wenn der Interpreter interaktiv betrieben wird. Hier müssen wir uns zunächst überlegen, woran der Interpreter erkennen soll, dass der Benutzer ein Kommando eingegeben hat. Ein Ansatz wäre, dass der Parser nach jedem Zeilenumbruch überprüft, ob der Benutzer ein vollständiges Kommando eingegeben hat. Dieser Ansatz scheitert aber daran, dass manche Kommandos sich über mehrere Zeilen erstrecken. Daher wartet der Interpreter darauf, dass nacheinander zwei Zeilenumbrüche eingegeben werden. Dann wird die bis dahin gelesene Eingabe in Zeile 20 in einem Feld von Bytes zusammengefasst und als Objekt der Klasse `ByteArrayInputStream` zurück gegeben.

Aufgabe 24:

1. Erweitern Sie den oben diskutierten Interpreter um `for`-Schleifen und bereichern Sie die Beispiel-Programmiersprache 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.
2. 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.

3. 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.

```

1  grammar Simple;
2
3  @header {
4      import java.util.List;
5      import java.util.ArrayList;
6  }
7
8  program
9      : (s = statement { $s.stmnt.execute(); Statement.prompt(); })+
10     ;
11  statement returns [Statement stmnt]
12     @init {
13         List<Statement> stmnts = new ArrayList<Statement>();
14     }
15     : v = VAR '=' e = expr ';'
16       { $stmnt = new Assignment(new Variable($v.text), $e.result); }
17     | v = VAR '=' 'read' '(' ')' ';'
18       { $stmnt = new Read(new Variable($v.text)); }
19     | 'print' '(' r = expr ')' ';'
20       { $stmnt = new Print($r.result); }
21     | 'if' '(' b = boolExpr ')' '{'
22       (l = statement { stmnts.add($l.stmnt); })* '}'
23       { $stmnt = new IfThen($b.result, stmnts); }
24     | 'while' '(' b = boolExpr ')' '{'
25       (l = statement { stmnts.add($l.stmnt); })* '}'
26       { $stmnt = new While($b.result, stmnts); }
27     ;
28  boolExpr returns [BoolExpr result]
29     : l = expr ( '==' r = expr { $result = new Equal( $l.result, $r.result); }
30       | '<' r = expr { $result = new LessThan($l.result, $r.result); }
31       )
32     ;
33  expr returns [Expr result]
34     : p = product { $result = $p.result; }
35     ( ('+' q = product) { $result = new Sum( $result, $q.result); }
36     | ('-' q = product) { $result = new Difference($result, $q.result); }
37     )*
38     ;
39  product returns [Expr result]
40     : f = factor { $result = $f.result; }
41     ( ('*' g = factor) { $result = new Product( $result, $g.result); }
42     | ('/' g = factor) { $result = new Quotient($result, $g.result); }
43     )*
44     ;
45  factor returns [Expr result]
46     : '(' expr ')' { $result = $expr.result; }
47     | v = VAR { $result = new Variable($v.text); }
48     | n = NUMBER { $result = new MyNumber($n.text); }
49     ;

```

Abbildung 11.13: ANTLR-Spezifikation der Grammatik.

```

50
51  VAR      : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
52  NUMBER   : '0'..'9' | ('1'..'9')('0'..'9')+;
53
54  MULTI_COMMENT : '/*' (~('*'|'/'|'/'))* '*' '/' { skip(); };
55  LINE_COMMENT  : '//' ~('\n')* { skip(); };
56  WS           : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 11.14: ANTLR-Spezifikation der Token.

```

1  public final class InputReader {
2      private static BufferedReader br = null;
3      private static String      EOL = "\n";
4
5      public static InputStream getStream() throws EOFException {
6          if (br == null) {
7              br = new BufferedReader(new InputStreamReader(System.in));
8          }
9          String input      = "";
10         String line       = null;
11         int   endlAdded = 0;
12         try {
13             while (true) {
14                 line = br.readLine();
15                 input += line + EOL;
16                 endlAdded += EOL.length();
17                 if (line == null) {
18                     throw new EOFException("EndOfFile");
19                 } else if (line.length() == 0 && input.length() > endlAdded) {
20                     byte[] byteArray =
21                         input.substring(0,input.length()-EOL.length()).getBytes();
22                     return new ByteArrayInputStream(byteArray);
23                 }
24             }
25         } catch (EOFException eof) {
26             throw eof;
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30         return null;
31     }
32 }

```

Abbildung 11.15: Die Klasse InputReader.

Kapitel 12

Grenzen kontextfreier Sprachen

In diesem Kapitel diskutieren wir die Grenzen kontextfreier Sprachen und leiten dazu das sogenannte “*große Pumping-Lemma*” her, mit dessen Hilfe wir beispielsweise zeigen können, dass die Sprache L_{square} , die durch

$$L_{\text{square}} = \{ww \mid w \in \Sigma^*\}$$

definiert wird, für das Alphabet $\Sigma = \{a, b\}$ keine kontextfreie Sprache ist.

12.1 Beseitigung nutzloser Symbole

In diesem Abschnitt zeigen wir, wie wir nutzlose Symbole aus einer kontextfreien Grammatik entfernen können. Ist $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik, so nennen wir eine syntaktische Variable $A \in V$ *nützlich*, wenn es Strings $w \in T^*$ und $\alpha, \beta \in (V \cup T)^*$ 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. Analog heißt ein Terminal $t \in T$ *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 $w \in L(G)$ auftritt. Variablen und Terminale, die nicht nützlich sind, bezeichnen wir als *nutzlose 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 31 (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, aus einer erzeugenden Variable lässt sich also immer mindestens ein Wort aus T^* herleiten. Die Notation $A \Rightarrow^* w$ drückt aus, dass das Wort w aus der Variablen A in endlich vielen Schritten abgeleitet werden kann. \square

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

¹ *Bison* ist ein Parser-Generator für die Sprache C.

$$A \rightarrow w \quad \text{mit } w \in T^*,$$

so ist die Variable A offenbar erzeugend.

2. Enthält die Grammatik $G = \langle V, T, R, S \rangle$ eine Regel der Form

$$A \rightarrow \alpha$$

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 $G = \langle \{S, A, B, C\}, \{x\}, R, S \rangle$ und die Menge der Regeln R sei wie folgt gegeben:

$$S \rightarrow ABC \mid A,$$

$$A \rightarrow AA \mid x,$$

$$B \rightarrow AC,$$

$$C \rightarrow BA.$$

Aufgrund der Regel

$$A \rightarrow x$$

ist zunächst A erzeugend. Aufgrund der Regel

$$S \rightarrow A$$

ist dann auch S erzeugend. Die Variablen B und C sind hingegen nicht erzeugend und damit sicher nutzlos. \square

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 $G = \langle \{S, A, B\}, \{x, y\}, R, S \rangle$, deren Regeln durch

$$S \rightarrow Ay,$$

$$A \rightarrow AA \mid x \quad \text{und}$$

$$B \rightarrow Ay$$

gegeben sind. Die erzeugenden Variablen 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

$$S \Rightarrow^* \alpha X \beta$$

gilt. 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

$$X \rightarrow \alpha$$

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 $G = \langle \{S, A, B, C\}, \{x, y\}, R, S \rangle$, deren Regeln durch

$$\begin{aligned} S &\rightarrow BC \mid A, \\ A &\rightarrow AA \mid x, \\ B &\rightarrow y \quad \text{und} \\ C &\rightarrow CC \end{aligned}$$

gegeben sind. Die beiden Regeln

$$S \rightarrow BC \quad \text{und} \quad S \rightarrow A$$

zeigen, dass alle Variablen erreichbar sind. Weiter sehen wir, dass die Variablen A , B und S erzeugend sind, denn es gilt

$$A \Rightarrow^* x, \quad S \Rightarrow^* x \quad \text{und} \quad B \Rightarrow^* y.$$

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

$$S \rightarrow BC$$

entfernt und damit ist dann B nicht mehr erreichbar und somit ebenfalls nutzlos.

Bemerkung: 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 um ein Wort $w \in T^*$ abzuleiten, denn wenn die Variable B in der Ableitung

$$A \Rightarrow^* w$$

auftritt, dann folgt aus der Erreichbarkeit von A auch die Erreichbarkeit von B . Wenn also eine Variable A gleichzeitig erreichbar und erzeugend ist und wir andererseits eine Variable B als nicht erreichbar erkannt haben, dann kann B getrost entfernt werden, denn das kann an der Tatsache, dass A erzeugend ist, nichts ändern. \diamond

Bemerkung: Der nachfolgende Beweis des Pumping-Lemmas ist logisch von der Beseitigung der nutzlosen Symbole unabhängig. Das war in einer früheren Version dieses Skriptes anders, denn bei dem damals verwendeten Beweis war es notwendig, die Grammatik vorher in *Chomsky-Normal-Form* zu transformieren. Die Beseitigung der nutzlosen Symbole ist ein Teil dieser Transformation. Der gleich folgende Beweis setzt nicht mehr voraus, dass die Grammatik in Chomsky-Normal-Form vorliegt. Da die Technik der Beseitigung nutzloser Symbole aber auch einen praktischen Wert hat, habe ich diesen Abschnitt trotzdem beibehalten.

12.2 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 12.2 auf Seite 163 zeigt einen Parse-Baum für den String “2*3+4”, der mit Hilfe der in Abbildung 12.1 gezeigten Grammatik erstellt wurde.

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:

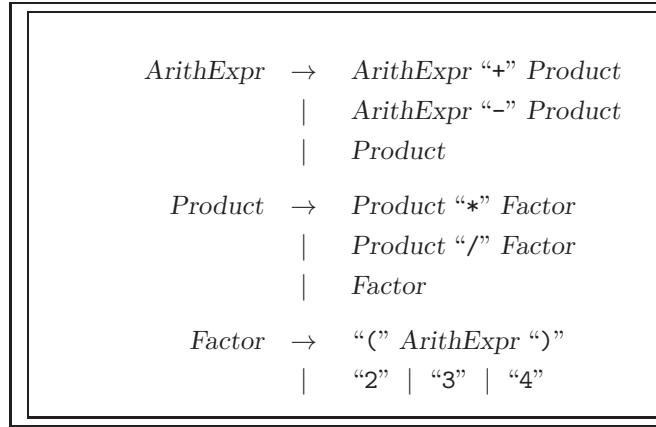


Abbildung 12.1: Grammatik für arithmetische Ausdrücke.

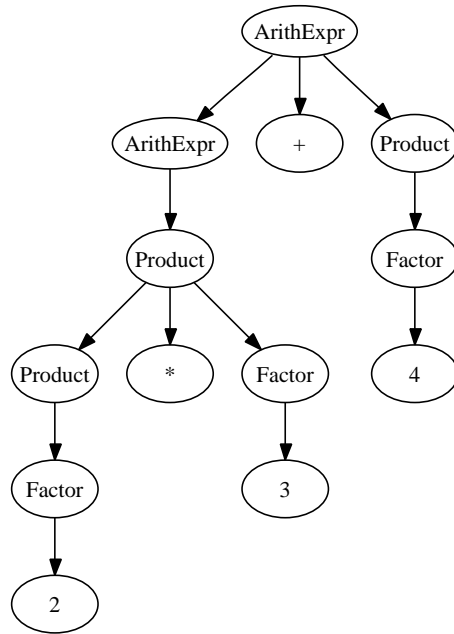


Abbildung 12.2: Ein Parse-Baum für den String “2*3+4”.

[[ArithExpr, ArithExpr, Product, Product, Factor, “2”],
 [ArithExpr, ArithExpr, Product, “*”],
 [ArithExpr, ArithExpr, Product, Factor, “3”],
 [ArithExpr, “+”],
 [ArithExpr, Product, Factor, “4”]
].

Die nun folgende Definition der Funktion *parseTree()* formalisiert die Berechnung des Parse-Baums. ◇

Definition 32 (*parseTree*)

Ist $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik, ist $w \in T^*$, $A \in V$ und gibt es eine Ableitung

$$A \Rightarrow^* w,$$

so definieren wir *parseTree*($A \Rightarrow^* w$) induktiv als Liste von Listen:

1. Falls $(A \rightarrow t_1 t_2 \cdots t_m) \in R$ mit $A \in V$ und $t_1 \cdots t_m = w$ ist, so setzen wir

$$\text{parseTree}(A \Rightarrow^* t) = [[A, t_1], [A, t_2], \dots, [A, t_m]].$$

2. Falls $A \Rightarrow^* w$ gilt, weil

$$(A \rightarrow B_1 B_2 \cdots B_m) \in R, \quad B_i \Rightarrow^* w_i \text{ für alle } i = 1, \dots, m, \quad \text{und } w = w_1 w_2 \cdots w_m$$

gilt, so definieren wir (unter Benutzung der SETLX-Notation für die Definition von Listen)

$$\text{parseTree}(A \Rightarrow^* w) = [[A] + l : i \in [1, \dots, m], l \in \text{parseTree}(B_i \Rightarrow^* w_i)].$$

Damit diese Definition auch tatsächlich alle Fälle abdeckt, müssen wir noch den Fall diskutieren, dass eines der Symbole B_i ein Terminal ist: In diesem Fall setzen wir

$$\text{parseTree}(B_i \Rightarrow^* B_i) := [[B_i]].$$

Wir definieren die *Breite* b einer Grammatik als die größte Anzahl von Symbolen, die auf der rechten Seite einer Grammatik-Regel der Form

$$A \rightarrow \alpha$$

auftreten.

Lemma 33 (Beschränktheits-Lemma) 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

$$\text{parseTree}(A \Rightarrow^* w)$$

ist, so gilt für die Länge des Wortes w die Abschätzung

$$|w| \leq b^{n-1}.$$

Beweis: Wir führen den Beweis durch Induktion nach der Länge n der längsten Liste in $\text{parseTree}(A \Rightarrow^* w)$.

I.A. $n = 2$: Wenn alle Listen in $\text{parseTree}(A \Rightarrow^* w)$ nur zwei Elemente haben, dann besteht die Ableitung aus genau einem Schritt und daher muss es eine Regel der Form

$$A \rightarrow t_1 t_2 \cdots t_m \quad \text{mit } w = t_1 t_2 \cdots t_m \text{ und } t_i \in T \text{ für alle } i = 1, \dots, m$$

in der Grammatik G geben. Es gilt dann

$$\text{parseTree}(A \Rightarrow^* w) = [[A, t_1], [A, t_2], \dots, [A, t_m]].$$

Daraus folgt

$$|w| = |t_1 t_2 \cdots t_m| = m \leq b = b^1 = b^{2-1},$$

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

$$\text{parseTree}(B_i \Rightarrow^* w_i)$$

für alle $i = 1, \dots, m$ höchstens die Länge n . Nach Induktions-Voraussetzung wissen wir also, dass

$$|w_i| \leq b^{n-1} \quad \text{für alle } i = 1, \dots, m$$

gilt. Daher haben wir

$$|w| = |w_1| + \cdots + |w_m| \leq b^{n-1} + \cdots + b^{n-1} = m \cdot b^{n-1} \leq b \cdot b^{n-1} = b^n = b^{(n+1)-1}. \quad \square$$

12.3 Das Pumping-Lemma für kontextfreie Sprachen

Satz 34 (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 folglich eine Länge von höchstens n Buchstaben.
2. $vx \neq \varepsilon$,
die Teilstrings v und x können also nicht beide gleichzeitig leer sein.
3. $\forall h \in \mathbb{N} : uv^hwx^hy \in L$.
die Strings v und x können beliebig oft repliziert (*gepumpt*) werden.

Beweis: Da L eine kontextfreie Sprache ist, gibt es eine kontextfreie Grammatik $G = \langle V, T, R, S \rangle$, 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

$$n := b^{k+1}.$$

Sei nun $s \in L$ mit $|s| \geq n$. Wir wählen nun einen Parse-Baum τ aus, der einerseits den String s ableitet und der andererseits unter allen Parse-Bäumen, die den String s aus S ableiten, die minimale Anzahl von Knoten hat. Für diesen Parse-Baum τ betrachten wir nun die Listen aus

$$\tau = \text{parseTree}(S \Rightarrow^* s).$$

Falls alle Listen hier eine Länge kleiner-gleich $k+1$ hätten, so würde aus den Beschränktheits-Lemma 33 folgen, dass

$$|s| \leq b^{(k+1)-1} = b^k < b^{k+1} = n, \quad \text{also} \quad |s| < n$$

gilt, im Widerspruch zu der Voraussetzung $|s| \geq n$. Also muss es in τ eine Liste geben, die mindestens die Länge $k+2$ hat. Wir wählen die längste Liste unter den Listen in τ aus. Diese Liste hat dann die Form

$$[A_1, \dots, A_l, t] \quad \text{mit } A_i \in V \text{ für alle } i \in \{1, \dots, l\}, \quad t \in T, \quad \text{sowie } l \geq k+1.$$

Wegen $l \geq k+1$ können nicht alle Variablen A_1, \dots, A_l voneinander verschieden sein, denn es gibt ja nur insgesamt k verschiedene syntaktische Variablen. Wir finden daher in der Menge $\{l, l-1, \dots, l-k\}$ zwei verschiedene Indizes i und j , so dass die Variablen A_i und A_j gleich sind und definieren $A := A_i = A_j$. Von den beiden Indizes i und j bezeichne i den kleineren Index, es gelte also $i < j$. Die Ableitung von s aus 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. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxy \Rightarrow^* uvvwxxy$, also

$$S \Rightarrow^* uv^2wx^2y.$$

3. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \Rightarrow^* uv^3Ax^3y \Rightarrow^* \dots \Rightarrow^* uv^hAx^hy \Rightarrow^* uv^hwx^hy$, also

$$uv^hwx^hy \in L \quad \text{für beliebige } h \in \mathbb{N}.$$

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. Dann würde wegen $A_i = A_j$ also

$$A_i \Rightarrow^+ A_i$$

gelten, wobei das Zeichen $+$ and dem Pfeil \Rightarrow anzeigt, dass diese Ableitung mindestens einen Schritt enthält. In diesem Fall wäre aber der Parse-Baum τ nicht minimal, denn wir könnten die Ableitungs-Schritte, die A_i in A_i überführen, einfach weglassen. Damit ist die Annahme $vx = \varepsilon$ widerlegt und es muss $vx \neq \varepsilon$ gelten.

Als letztes zeigen wir, dass die Ungleichung $|vwx| \leq n$ gilt. Wir haben

$$A = A_i \Rightarrow^* vA_jx \Rightarrow^* vwx.$$

Weil einerseits $i \in \{l-k, l-(k-1), \dots, l\}$ gilt und andererseits die der Ableitung $A \Rightarrow^* vwx$ zugeordnete Liste aus $\text{parseTree}(S \Rightarrow^* w)$ von maximaler Länge war, wissen wir, dass die Länge der längsten Liste in

$$\text{parseTree}(A \Rightarrow^* vwx)$$

kleiner-gleich $k+2$ ist. Nach dem Beschränktheits-Lemma 33 folgt damit für die Länge von vwx die Abschätzung

$$|vwx| \leq b^{(k+2)-1} = b^{k+1} = n. \quad \square$$

Bemerkung: Das Pumping-Lemma wird in der deutschsprachigen Literatur gelegentlich als “*Schleifensatz*” bezeichnet.

12.4 Anwendungen des Pumping-Lemmas

Wir zeigen nun, wie mit Hilfe des Pumping-Lemmas der Nachweis erbracht werden kann, dass bestimmte Sprachen nicht kontextfrei sind.

12.4.1 Die Sprache $L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$ ist nicht kontextfrei

Wir weisen nun nach, dass die Sprache

$$L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$$

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:

$$s := \mathbf{a}^n \mathbf{b}^n \mathbf{c}^n.$$

Dieser String hat die Länge $3 \cdot n \geq n$ und erfüllt damit die Voraussetzung über die Länge. Damit finden wir also eine Zerlegung $s = uvwx$ mit den obigen Eigenschaften. Da der Teilstring vwx eine Länge kleiner oder gleich n hat, können in diesem String nicht gleichzeitig die Buchstaben “a” und “c” vorkommen. Wir betrachten die nach dieser Erkenntnis noch möglichen Fälle getrennt.

1. Fall: In dem String vwx kommen nur die Buchstaben “a” und “b” vor, der Buchstabe “c” kommt nicht vor:

$$\text{count}(vwx, \mathbf{c}) = 0.$$

Da $vx \neq \varepsilon$ ist, folgt

$$\text{count}(vx, \mathbf{a}) + \text{count}(vx, \mathbf{b}) > 0.$$

Wir nehmen zunächst an, dass $\text{count}(vx, \mathbf{a}) > 0$ gilt, der Fall $\text{count}(vx, \mathbf{b}) > 0$ ist analog zu behandeln. Dann erhalten wir einerseits

$$\begin{aligned} \text{count}(uwy, \mathbf{c}) &= \text{count}(s, \mathbf{c}) - \text{count}(vx, \mathbf{c}) \\ &= \text{count}(s, \mathbf{c}) - 0 \\ &= \text{count}(\mathbf{a}^n \mathbf{b}^n \mathbf{c}^n, \mathbf{c}) \\ &= n. \end{aligned}$$

Zählen wir nun die Häufigkeit, mit welcher der Buchstabe “a” in dem String uwy auftritt, so erhalten wir

$$\begin{aligned} \text{count}(uwy, \mathbf{a}) &= \text{count}(s, \mathbf{a}) - \text{count}(vx, \mathbf{a}) \\ &< \text{count}(s, \mathbf{a}) \\ &= \text{count}(\mathbf{a}^n \mathbf{b}^n \mathbf{c}^n, \mathbf{a}) \\ &= n. \end{aligned}$$

Damit haben wir dann aber

$$\text{count}(uwy, \mathbf{a}) < n = \text{count}(uwy, \mathbf{c})$$

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 25: Zeigen Sie, dass die Sprache

$$L = \{ \mathbf{a}^{k^2} \mid k \in \mathbb{N} \}$$

nicht kontextfrei ist.

Hinweis: Argumentieren Sie über die Länge der betrachteten Strings.

Aufgabe 26: Das Alphabet Σ sei durch die Festlegung $\Sigma := \{ \mathbf{a}, \mathbf{b} \}$ definiert. Zeigen Sie, dass die Sprache

$$L = \{ tt \mid t \in \Sigma^* \}$$

nicht kontextfrei ist.

Aufgabe 27: Zeigen Sie, dass die Sprache

$$L = \{ \mathbf{a}^p \mid p \text{ ist Primzahl} \}$$

nicht kontextfrei ist.

Kapitel 13

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]. Neben dem Algorithmus von Earley gibt es noch den Cocke-Younger-Kasami-Algorithmus, in der Literatur auch als CYK-Algorithmus bekannt, der unabhängig von John Cocke [CS70], Daniel H. Younger [You67] und Tadao Kasami [Kas65] entdeckt wurde. Der CYK-Algorithmus hat allerdings eine Laufzeit von n^3 und ist nur anwendbar, wenn die Grammatik in Chomsky-Normalform vorliegt. Damit ist der CYK-Algorithmus in der Praxis wertlos. Der von Earley angegebene Algorithmus kann auf beliebige kontextfreie Grammatiken angewendet werden und er hat bei einer eindeutigen Grammatik im schlimmsten Fall eine quadratische Komplexität. In vielen praktisch relevanten Fällen ist die Laufzeit des Algorithmus sogar linear.

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.

13.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist.

Definition 35 (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

$$\langle A \rightarrow \alpha \bullet \beta, k \rangle$$

dann als ein *Earley-Objekt*, falls folgendes gilt:

1. $(A \rightarrow \alpha \beta) \in R$ und
2. $k \in \{0, 1, \dots, n\}$.

□

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

$$Q_0, Q_1, \dots, Q_n.$$

Die Interpretation von

$$\langle A \rightarrow \alpha \bullet \beta, k \rangle \in Q_j \quad \text{mit } j \geq k$$

ist dann wie folgt:

1. Der Parser versucht die Regel $A \rightarrow \alpha\beta$ auf den Teilstring $x_{k+1} \dots x_n$ anzuwenden und am Anfang dieses Teilstrings ein A mit Hilfe der Regel $A \rightarrow \alpha\beta$ zu erkennen.
2. Am Anfang des Teilstrings $x_{k+1} \dots x_j$ hat der Parser bereits α erkannt, es gilt also

$$\alpha \Rightarrow^* x_{k+1} \dots x_j.$$

3. Folglich versucht der Parser am Anfang des Teilstrings $x_{j+1} \dots x_n$ ein β erkennen.

Der Algorithmus von Earley verwaltet für $i = 0, 1, \dots, n$ Mengen Q_i von Earley-Objekten, die den Zustand beschreiben, in dem der Parser ist, wenn der Teilstring $x_1 \dots x_j$ verarbeitet ist. Zu Beginn des Algorithmus wird der Grammatik ein neues Start-Symbol \hat{S} sowie die Regel $\hat{S} \rightarrow S$ hinzugefügt. Die Menge Q_0 wird definiert als

$$Q_0 := \{ \langle \hat{S} \rightarrow \bullet S, 0 \rangle \},$$

denn der Parser soll ja das Start-Symbol S am Anfang des Strings $x_1 \dots x_n$ erkennen. Die restlichen Mengen Q_j sind für $j = 1, \dots, n$ 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 $\langle A \rightarrow \beta \bullet a\gamma, k \rangle$ enthält, wobei a ein Terminal ist, so versucht der Parser, die rechte Seite der Regel $A \rightarrow \beta a\gamma$ zu erkennen und hat bis zur Position j bereits den Teil β erkannt. Folgt auf dieses β nun, wie in der Regel $A \rightarrow \beta a\gamma$ 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

$$\langle A \rightarrow \beta a \bullet \gamma, k \rangle$$

dem Zustand Q_{j+1} hinzugefügt:

$$\langle A \rightarrow \beta \bullet a\gamma, k \rangle \in Q_j \wedge x_{j+1} = a \Rightarrow Q_{j+1} := Q_{j+1} \cup \{ \langle A \rightarrow \beta a \bullet \gamma, k \rangle \}.$$

2. Vorhersage-Operation

Falls der Zustand Q_j ein Earley-Objekt der Form $\langle A \rightarrow \beta \bullet C\delta, k \rangle$ 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 $C \rightarrow \gamma$ der Grammatik das Earley-Objekt $\langle C \rightarrow \bullet \gamma, j \rangle$ zu der Menge Q_j hinzu:

$$\langle A \rightarrow \beta \bullet C\delta, k \rangle \in Q_j \wedge (C \rightarrow \gamma) \in R \Rightarrow Q_j := Q_j \cup \{ \langle C \rightarrow \bullet \gamma, j \rangle \}.$$

3. Vervollständigungs-Operation

Falls der Zustand Q_i ein Earley-Objekt der Form $\langle C \rightarrow \gamma \bullet, j \rangle$ enthält und weiter der Zustand Q_j ein Earley-Objekt der Form $\langle A \rightarrow \beta \bullet C\delta, k \rangle$ 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 $\langle A \rightarrow \beta C \bullet \delta, k \rangle$ hinzu:

$$\langle C \rightarrow \gamma \bullet, j \rangle \in Q_i \wedge \langle A \rightarrow \beta \bullet C\delta, k \rangle \in Q_j \Rightarrow Q_i := Q_i \cup \{ \langle A \rightarrow \beta C \bullet \delta, k \rangle \}.$$

Der Algorithmus von Earley um einen String der Form $s = x_1 \dots x_n$ zu parsen funktioniert so:

1. Wir initialisieren die Zustände Q_i wie folgt:

$$Q_0 := \{ \langle \hat{S} \rightarrow \bullet S, 0 \rangle \},$$

$$Q_i := \{ \} \quad \text{für } i = 1, \dots, n.$$

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

$$C \rightarrow \varepsilon,$$

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 $\langle \hat{S} \rightarrow S\bullet, 0 \rangle$ enthält, dann war das Parsen erfolgreich und der String $x_1 \cdots x_n$ liegt in der von der Grammatik erzeugten Sprache.

Beispiel: Abbildung 13.1 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.

$\begin{aligned} E &\rightarrow E \text{“+”} P \mid P \\ P &\rightarrow P \text{“*”} F \mid F \\ F &\rightarrow \text{“1”} \mid \text{“2”} \mid \text{“3”} \end{aligned}$

Abbildung 13.1: Eine vereinfachte Grammatik für arithmetische Ausdrücke.

1. Wir initialisieren Q_0 als

$$Q_0 = \{ \langle \hat{S} \rightarrow \bullet E, 0 \rangle \}.$$

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 $\langle \hat{S} \rightarrow \bullet E, 0 \rangle$ an. Dadurch werden der Menge Q_0 zunächst die beiden Earley-Objekte

$$\langle E \rightarrow \bullet E \text{“+”} P, 0 \rangle \quad \text{und} \quad \langle E \rightarrow \bullet P, 0 \rangle$$

hinzugefügt. Auf das Earley-Objekt $\langle E \rightarrow \bullet P, 0 \rangle$ können wir die Vorhersage-Operation ein weiteres Mal anwenden und erhalten dann die beiden neuen Earley-Objekte

$$\langle P \rightarrow \bullet P \text{“+”} F, 0 \rangle \quad \text{und} \quad \langle P \rightarrow \bullet F, 0 \rangle.$$

Wenden wir auf das Earley-Objekt $\langle P \rightarrow \bullet F, 0 \rangle$ die Vorhersage-Operation an, so erhalten wir schließlich noch die folgenden Earley-Objekte in Q_0 :

$$\langle F \rightarrow \bullet \text{“1”}, 0 \rangle, \quad \langle F \rightarrow \bullet \text{“2”}, 0 \rangle, \quad \text{und} \quad \langle F \rightarrow \bullet \text{“3”}, 0 \rangle.$$

Insgesamt enthält Q_0 nun die folgenden Earley-Objekte:

- (a) $\langle \hat{S} \rightarrow \bullet E, 0 \rangle$,
- (b) $\langle E \rightarrow \bullet E "+" P, 0 \rangle$
- (c) $\langle E \rightarrow \bullet P, 0 \rangle$,
- (d) $\langle P \rightarrow \bullet P "*" F, 0 \rangle$,
- (e) $\langle P \rightarrow \bullet F, 0 \rangle$,
- (f) $\langle F \rightarrow \bullet "1", 0 \rangle$,
- (g) $\langle F \rightarrow \bullet "2", 0 \rangle$,
- (h) $\langle F \rightarrow \bullet "3", 0 \rangle$.

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:

$$Q_1 = \{ \langle F \rightarrow "1" \bullet, 0 \rangle \}.$$

2. Nun setzen wir $i = 1$ und wenden zunächst auf Q_1 die Vervollständigungs-Operation an. Aufgrund des Earley-Objekts $\langle F \rightarrow "1" \bullet, 0 \rangle$ in Q_1 suchen wie in Q_0 ein Earley-Objekt, bei dem die Markierung " \bullet " vor der Variablen F steht. Wir finden das Earley-Objekt $\langle P \rightarrow \bullet F, 0 \rangle$. Daher fügen wir nun Q_1 das Earley-Objekt

$$\langle P \rightarrow F \bullet, 0 \rangle$$

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) $\langle P \rightarrow F \bullet, 0 \rangle$,
- (b) $\langle P \rightarrow P \bullet "*" F, 0 \rangle$,
- (c) $\langle E \rightarrow P \bullet, 0 \rangle$,
- (d) $\langle E \rightarrow E \bullet "+" P, 0 \rangle$,
- (e) $\langle \hat{S} \rightarrow E \bullet, 0 \rangle$.

Als nächstes wenden wir auf diese Earley-Objekte die Vorhersage-Operation an. Da das Markierungs-Zeichen " \bullet " 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

$$\langle E \rightarrow E \bullet "+" P, 0 \rangle$$

enthält, fügen wir in Q_2 das Earley-Objekt

$$\langle E \rightarrow E "+" \bullet P, 0 \rangle$$

ein.

3. Nun setzen wir $i = 2$ und wenden zunächst auf Q_2 die Vervollständigungs-Operation an. Zu diesem Zeitpunkt gilt

$$Q_2 = \{ \langle E \rightarrow E "+" \bullet P, 0 \rangle \}.$$

Da in dem einzigen Earley-Objekt, das hier auftritt, das Markierungs-Zeichen " \bullet " 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

$$\langle P \rightarrow \bullet F, 2 \rangle \quad \text{und} \quad \langle P \rightarrow \bullet P "*" F, 2 \rangle.$$

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) $\langle F \rightarrow \bullet "1", 2 \rangle$,
- (b) $\langle F \rightarrow \bullet "2", 2 \rangle$,
- (c) $\langle F \rightarrow \bullet "3", 2 \rangle$.

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

$$Q_3 = \{ \langle F \rightarrow "2" \bullet, 2 \rangle \}.$$

4. Wir setzen $i = 3$ und wenden auf Q_3 die Vervollständigungs-Operation an. Dadurch fügen wir

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) $\langle P \rightarrow P \bullet "*" F, 2 \rangle$,
- (b) $\langle E \rightarrow E "+" P \bullet, 0 \rangle$,
- (c) $\langle E \rightarrow E \bullet "+" P, 0 \rangle$
- (d) $\langle \hat{S} \rightarrow E \bullet, 0 \rangle$.

Als letztes wenden wir die Lese-Operation an. Da der nächste zu lesende Buchstabe das Zeichen "*" ist, erhalten wir

$$Q_4 = \{ \langle P \rightarrow P "*" \bullet F, 2 \rangle \}.$$

5. Wir setzen $i = 4$. Die Vervollständigungs-Operation liefert keine neuen Earley-Objekte. Die Vorhersage-Operation liefert folgende Earley-Objekte:

- (a) $\langle F \rightarrow \bullet "1", 4 \rangle$,
- (b) $\langle F \rightarrow \bullet "2", 4 \rangle$,
- (c) $\langle F \rightarrow \bullet "3", 4 \rangle$.

Da das nächste Zeichen die Ziffer "3" ist, liefert die Lese-Operation für Q_5 :

$$Q_5 = \langle F \rightarrow "3" \bullet, 4 \rangle.$$

6. Wir setzen $i = 5$. Die Vervollständigungs-Operation liefert nacheinander die folgenden Earley-Objekte:

- (a) $\langle P \rightarrow P "*" F \bullet, 2 \rangle$,
- (b) $\langle E \rightarrow E "+" P \bullet, 0 \rangle$,
- (c) $\langle P \rightarrow P \bullet "*" F, 2 \rangle$,
- (d) $\langle E \rightarrow E \bullet "+" P, 0 \rangle$
- (e) $\langle \hat{S} \rightarrow E \bullet, 0 \rangle$.

Da die Menge Q_5 das Earley-Objekt $\langle \hat{S} \rightarrow E \bullet, 0 \rangle$ 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 13.1 angegebenen Grammatik liegt.

13.2 Implementierung

Im folgenden präsentieren wir eine einfache Implementierung des von Earley angegebenen Algorithmus. Abbildung 13.2 auf Seite 173 zeigt die Klasse `EarleyParser`. Diese Klasse enthält vier Member-Variablen:

1. `mGrammar` ist die Grammatik, mit der geparkt werden soll.
2. `mString` ist der zu parsende String.
Um den Parser möglichst einfach zu halten, wird der hier entwickelte 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 der Mengen $[Q_0, Q_1, \dots, Q_n]$. Hierbei ist $n = mLength$.

```

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 13.2: Struktur der Klasse `EarleyParser`

Abbildung 13.2 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

$$\langle \hat{S} \rightarrow \bullet S, 0 \rangle$$

abgelegt, das von der Methode `startItem()` berechnet wird. Der Rest der Arbeit wird an die Methode `parse()` delegiert, die in Abbildung 13.3 auf Seite 174 gezeigt ist. Die äußere `for`-Schleife iteriert über die Mengen Q_0, Q_1, \dots, 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 vergrößert¹. 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 13.3: 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 13.4: Implementierung der Vervollständigungs-Operation

Abbildung 13.4 auf Seite 174 zeigt die Implementierung der Vervollständigungs-Operation. Das Argument i gibt an, welche der Mengen Q_i vervollständigt werden soll. Die Methode gibt als Ergebnis einen Wahrheitswert zurück. Dieser ist **true** falls bei der Vervollständigungs-Operation neue Earley-Objekte in die Menge Q_i

¹ Wenn die Grammatik keine ε -Regeln (das sind Regeln der Form $A \rightarrow \varepsilon$) 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.

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:

$$\langle C \rightarrow \gamma \bullet, j \rangle \in Q_i \wedge \langle A \rightarrow \beta \bullet C \delta, k \rangle \in Q_j \Rightarrow Q_i := Q_i \cup \{ \langle A \rightarrow \beta C \bullet \delta, k \rangle \}$$

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

$$\langle C \rightarrow \gamma \bullet, j \rangle$$
 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

$$\langle A \rightarrow \beta \bullet C \delta, k \rangle. \quad (*)$$
 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

$$\langle A \rightarrow \beta C \bullet \delta, k \rangle,$$
 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 13.5: Implementierung der Vorhersage-Operation

Abbildung 13.5 auf Seite 176 zeigt die Methode `predict()`, welche die Vorhersage-Operation implementiert. Diese Operation hatten wir im letzten Abschnitt durch die folgende Formel spezifiziert:

$$\langle A \rightarrow \beta \bullet C\delta, k \rangle \in Q_i \wedge (C \rightarrow \gamma) \in R \Rightarrow Q_i := Q_i \cup \{ \langle C \rightarrow \bullet \gamma, i \rangle \}$$

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

$$\langle A \rightarrow \beta \bullet C\delta, k \rangle$$

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 $\langle A \rightarrow \beta \bullet C\delta, k \rangle$ 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 $C \rightarrow \gamma$, also alle Regeln, bei denen die Variable C auf der linken Seite des Pfeils steht, als Earley-Objekte der Form

$$\langle C \rightarrow \bullet \gamma, i \rangle$$

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 13.6: Implementierung der Lese-Operation

Abbildung 13.6 zeigt die Implementierung der Lese-Operation. Wir hatten diese Operation wie folgt spezifiziert:

$$\langle A \rightarrow \beta \bullet a \gamma, k \rangle \in Q_i \wedge x_{i+1} = a \Rightarrow Q_{i+1} := Q_{i+1} \cup \{ \langle A \rightarrow \beta a \bullet \gamma, k \rangle \}.$$

Die Umsetzung dieser Spezifikation 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 Nummerierung der einzelnen Zeichen mit 0. In der Notation der Spezifikation der Operation gilt also

$$\text{mString.charAt}(i) = x_{i+1}.$$

4. Ist dieses Zeichen a , so werden in der **for**-Schleife alle Earley-Objekte der Menge Q_i darauf überprüft, ob sie die Form

$$\langle A \rightarrow \beta \bullet a \gamma, k \rangle$$

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

$$\langle A \rightarrow \beta a \bullet \gamma, k \rangle$$

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 13.7: Die Klasse `EarleyItem`, erster Teil.

Die Abbildungen 13.7 und 13.8 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 verglichen. Die erste Komponente, bei der sich die zu unterscheidenden Earley-Objekte unterscheiden, bestimmt den

Unterschied.

```

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 13.8: Die Klasse `EarleyItem`, zweiter Teil.

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

$$\langle A \rightarrow \alpha \bullet C \beta, k \rangle$$

hat, ob also hinter dem “•” die Variable C steht.

3. Die Methode `scan(a)` prüft, ob das Earley-Objekt von der Form

$$\langle A \rightarrow \alpha \bullet a \beta, k \rangle$$

ist, ob also hinter dem Markierungs-Zeichen das Zeichen a folgt.

4. Die Methode `nextVar()` prüft, ob das Earley-Objekt die Form

$$\langle A \rightarrow \alpha \bullet C \delta, k \rangle$$

hat. Es wird also überprüft, ob auf das Markierungs-Zeichen eine Variable folgt. Gegebenenfalls wird diese zurück gegeben, sonst liefert die Methode `null` als Ergebnis.

5. Die Methode `moveDot()` bewegt das Markierungs-Zeichen um eine Stelle nach rechts, ein Earley-Objekt der Form

$$\langle A \rightarrow \alpha \bullet X \delta, k \rangle$$

wird also zu

$$\langle A \rightarrow \alpha X \bullet \delta, k \rangle$$

transformiert. Dabei kann X sowohl für eine Variable als auch für ein Literal stehen.

Der Aufruf der Methode `moveDot()` für ein Earley-Objekt $\langle A \rightarrow \alpha \bullet \beta, k \rangle$ 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 13.9 zeigt die von unserem Earley-Parser berechneten Mengen für die Eingabe $1+2*3$, falls die in Abbildung 13.1 gezeigte Grammatik verwendet wird.

13.3 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 $\langle \hat{S} \rightarrow S \bullet, 0 \rangle$ 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 $\langle \hat{S} \rightarrow S \bullet, 0 \rangle$ 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 36 Es sei $s = x_1 x_2 \cdots x_n \in T^*$ der String, auf den wir den Algorithmus von Earley anwenden. Weiter gelte

$$\langle A \rightarrow \alpha \bullet \beta, k \rangle \in Q_i \quad \text{mit } i \in \{0, \dots, n\}.$$

Dann gilt

```

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 13.9: Ausgabe des Earley-Parsers für die Eingabe “1+2*3” und die Grammatik aus Abbildung 13.1.

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i.$$

Beweis: Wir führen den Beweis durch Induktion über die Anzahl l der Berechnungs-Schritte, die der Algorithmus durchgeführt hat, um $\langle A \rightarrow \alpha \bullet \beta, k \rangle \in Q_i$ nachzuweisen.

I.A.: $l = 0$. Zu Beginn enthält die Menge Q_0 nur das Earley-Objekt

$$\langle \widehat{S} \rightarrow \bullet S, 0 \rangle$$

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.: $0, \dots, l \mapsto l + 1$. Wir müssen eine Fallunterscheidung nach der Art der Operation durchführen, mit der das Earley-Objekt $E = \langle A \rightarrow \alpha \bullet \beta, k \rangle$ erzeugt worden ist.

- (a) E ist durch eine Vorhersage-Operation in Q_i eingefügt worden. Daher enthält Q_i ein Earley-Objekt der Form

$$F = \langle A' \rightarrow \alpha' \bullet A \delta, k' \rangle.$$

Dann muss die Grammatik eine Regel der Form $A \rightarrow \beta$ enthalten, mit der die Vorhersage-Operation das Earley-Objekt

$$\langle A \rightarrow \bullet \beta', i \rangle$$

erzeugt hat. Damit gilt also $\alpha = \varepsilon$, $\beta = \beta'$ 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 $\langle A \rightarrow \alpha' \bullet x_i \beta, k \rangle \in Q_{i-1}$, 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

$$E = \langle A \rightarrow \alpha' C \bullet \beta, k \rangle$$

und es gibt ein Earley-Objekt

$$\langle C \rightarrow \delta \bullet, j \rangle \in Q_i$$

und ein weiteres Earley-Objekt

$$\langle A \rightarrow \alpha' \bullet C \beta, k \rangle \in Q_j.$$

Also haben wir $\alpha = \alpha' C$. Aus $\langle A \rightarrow \alpha' \bullet C \beta, k \rangle \in Q_j$ folgt nach Induktions-Voraussetzung

$$\alpha' \Rightarrow^* x_{k+1} \cdots x_j$$

und aus $\langle C \rightarrow \delta\bullet, j \rangle \in Q_i$ folgt nach Induktions-Voraussetzung

$$\delta \Rightarrow^* x_{j+1} \cdots x_i,$$

so dass wir insgesamt die folgende Ableitung haben:

$$\begin{aligned} \alpha &= \alpha' C \\ &\Rightarrow^* x_{k+1} \cdots x_j C \\ &\Rightarrow x_{k+1} \cdots x_j \delta \\ &\Rightarrow^* x_{k+1} \cdots x_j x_{j+1} \cdots x_i \\ &= x_{k+1} \cdots x_i \end{aligned}$$

Damit ist $\alpha \Rightarrow^* x_{k+1} \cdots x_i$ nachgewiesen. □

Korollar 37 (Korrektheit)

Wenden wir den Algorithmus von Earley auf den String $s = x_1 \cdots x_n$ an und gilt $\langle \hat{S} \rightarrow S\bullet, 0 \rangle \in Q_n$, so folgt $s \in L(G)$.

Beweis: Setzen wir $A := \hat{S}$, $\alpha := S$, $\beta := \varepsilon$, $k := 0$ 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

$$\langle \hat{S} \rightarrow S\bullet, 0 \rangle \in Q_n,$$

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 38 Angenommen, wir haben die folgenden Voraussetzungen:

1. $\langle A \rightarrow \alpha \bullet \beta \gamma, k \rangle \in Q_i$,
2. $\beta \Rightarrow^* x_{i+1} \cdots x_l$.

Dann gilt auch

$$\langle A \rightarrow \alpha \beta \bullet \gamma, k \rangle \in Q_l.$$

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 $\langle A \rightarrow \alpha \bullet \beta \gamma, k \rangle \in Q_i$ die Form

$$\langle A \rightarrow \alpha \bullet x_{i+1} \cdots x_l \gamma, k \rangle \in Q_i.$$

Wenden wir hier $(l - i)$ mal die Lese-Operation an, so erhalten wir

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_l \bullet \gamma, k \rangle \in Q_l$$

und wegen $\beta = x_{i+1} \cdots x_l$ ist das äquivalent zu

$$\langle A \rightarrow \alpha \beta \bullet \gamma, k \rangle \in Q_l.$$

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 $D \rightarrow \delta$ 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

$$\delta \Rightarrow^* x_{j+1} \cdots x_h \tag{13.1}$$

und

$$\mu \Rightarrow^* x_{h+1} \cdots x_l \tag{13.2}$$

für ein geeignetes $h \in \{j, \dots, l+1\}$. Also können wir zunächst auf das Earley-Objekt

$$\langle A \rightarrow \alpha \bullet \beta \gamma, k \rangle \in Q_i$$

$(j - i)$ mal die Lese-Operation anwenden. Damit erhalten wir

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j \bullet D \mu \gamma, k \rangle \in Q_j. \tag{13.3}$$

Auf dieses Earley-Objekt wenden wir die Vorhersage-Operation an und erhalten

$$\langle D \rightarrow \bullet \delta, j \rangle \in Q_j. \tag{13.4}$$

Aus (13.4) und (13.1) folgt mit der Induktions-Voraussetzung

$$\langle D \rightarrow \delta \bullet, j \rangle \in Q_h. \tag{13.5}$$

Aus (13.3) und (13.5) folgt mit der Vervollständigungs-Operation

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j D \bullet \mu \gamma, k \rangle \in Q_h. \tag{13.6}$$

Aus (13.6) und (13.2) folgt mit der Induktions-Voraussetzung

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j D\mu \bullet \gamma, k \rangle \in Q_l. \quad (13.7)$$

Wegen $\beta = x_{i+1} \cdots x_j D\mu$ haben wir damit

$$\langle A \rightarrow \alpha \beta \bullet \gamma, k \rangle \in Q_l$$

gezeigt und das ist gerade die Behauptung. \square

Korollar 39 (Vollständigkeit)

Falls $s = x_1 \cdots x_n \in L(G)$ ist, dann gilt $\langle \hat{S} \rightarrow S \bullet, 0 \rangle \in Q_n$.

Beweis: Der Algorithmus von Earley initialisiert die Menge Q_0 mit dem Earley-Objekt

$$\langle \hat{S} \rightarrow \bullet S, 0 \rangle.$$

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. \square

13.4 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:

$$\text{Wenn } \langle A \rightarrow \alpha \bullet \beta, k \rangle \in Q_i, \text{ dann } k \in \{0, \dots, i\}.$$

Die Komponente $A \rightarrow \alpha \bullet \beta$ 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

$$i \in \{0, \dots, n\}.$$

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 13.5) 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

$$\langle A \rightarrow \bullet \gamma, i \rangle,$$

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 $\mathcal{O}(1) \cdot \mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

2. Bei der Implementierung der Lese-Operation, die in Abbildung 13.6 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 13.4 auf Seite 174 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 Element 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 $i = 0, \dots, n$ 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 $\mathcal{O}(n^2)$ 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 14

LR-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 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.

14.1 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

$$\begin{aligned} E &\rightarrow E \text{ “+” } P \mid P \\ P &\rightarrow P \text{ “*” } F \mid F \\ F &\rightarrow \text{“1”} \mid \text{“2”} \mid \text{“3”} \end{aligned}$$

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:

$$\begin{aligned} 1 + 2 * 3 &\Leftarrow F + 2 * 3 && (\text{Regel: } F \rightarrow \text{“1”}) \\ &\Leftarrow P + 2 * 3 && (\text{Regel: } P \rightarrow F) \\ &\Leftarrow E + 2 * 3 && (\text{Regel: } E \rightarrow P) \\ &\Leftarrow E + F * 3 && (\text{Regel: } F \rightarrow \text{“2”}) \\ &\Leftarrow E + P * 3 && (\text{Regel: } P \rightarrow F) \\ &\Leftarrow E + P * F && (\text{Regel: } F \rightarrow \text{“3”}) \\ &\Leftarrow E + P && (\text{Regel: } P \rightarrow P \text{ “*” } F) \\ &\Leftarrow E && (\text{Regel: } E \rightarrow E \text{ “+” } P) \end{aligned}$$

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:

$$\begin{aligned}
 E &\Rightarrow E + P \\
 &\Rightarrow E + P * F \\
 &\Rightarrow E + P * 3 \\
 &\Rightarrow E + F * 3 \\
 &\Rightarrow E + 2 * 3 \\
 &\Rightarrow P + 2 * 3 \\
 &\Rightarrow F + 2 * 3 \\
 &\Rightarrow 1 + 2 * 3
 \end{aligned}$$

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 *reverse rightmost derivation* steht und ausdrückt, dass solche Parser eine Rechts-Ableitung rückwärts 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

$$E + P * 3,$$

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

$$E + E * 3$$

erhalten. Die einzigen Reduktionen, die wir jetzt noch durchführen können, führen über die Zwischenergebnisse $E + E * F$ und $E + E * P$ zu dem String

$$E + E * E,$$

der sich dann aber mit der oben angegebenen Grammatik nicht mehr reduzieren 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.

14.2 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

$$P = \langle Q, q_0, action, goto \rangle$$

beschrieben. Dabei gilt:

1. Q ist die Menge der *Zustände* des Shift-Reduce-Parsers.

Zunächst werden wir die einzelnen Zustände rein abstrakt sehen. Später, wenn wir die Theorie der SLR-Parser diskutieren, werden wir sehen, dass ein Zustand die Information speichert, welche Variable der Parser gerade zu erkennen versucht und welcher Teil der rechten Seite einer Regel bereits erkannt worden ist.

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

$$Action := \{ \langle \text{shift}, q \rangle \mid q \in Q \} \cup \{ \langle \text{reduce}, r \rangle \mid r \in R \} \cup \{ \text{accept} \} \cup \{ \text{error} \},$$

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:

$$action : Q \times T \rightarrow Action.$$

4. *goto* ist eine Funktion, die jedem Zustand $q \in Q$ und jeder syntaktischen Variablen $v \in V$ einen neuen Zustand aus Q zuordnet:

$$goto : Q \times V \rightarrow Q.$$

Ein Shift-Reduce-Parser arbeitet nun mit den folgenden Daten-Strukturen.

1. Einem Stack *states*, auf dem Zustände aus der Menge Q abgelegt werden:

$$states \in Stack(Q).$$

2. Einem Stack *symbols*, auf dem Grammatik-Symbole, also Terminale und Variablen-Symbole abgelegt werden:

$$symbols \in Stack(T \cup V).$$

Zur Vereinfachung der folgenden Überlegungen nehmen wir an, dass die Menge T der Terminale das spezielle Symbol “EOF” enthält und dass dieses Symbol das Ende des zu parsenden Strings spezifiziert aber sonst in dem String nicht auftritt.

Figure 14.1 on page 190 shows the implementation of the SETLX procedure *parseSR* that implements shift-reduce-parsing. This function assumes that the function *action* is coded as a binary relation that is stored in the global variable *actionTable*. The function *goto* is also represented as a binary relation. It is stored in the global variable *gotoTable*. The function *parseSR* is called with one argument *tl*. This is the list of tokens that have to be parsed. The last element of this list is the special token “EOF” denoting the end of file. The invocation *parseSR*(*tl*) returns **true** if the token list *tl* can be parsed successfully and **false** otherwise. The implementation of *parseSR* works as follows:

1. The variable *index* points to the next token in the token list that is to be read. Therefore, this variable is initialized to 1.
2. The variable *symbols* stores the stack of symbols. The top of this stack is at the end of this list. Initially, the stack of symbols is empty.

```

1  parseSR := procedure(tl) {
2      index := 1;          // point to next token
3      symbols := [];       // stack of symbols
4      states := ["s0"];    // stack of states, 0 is start state
5      while (true) {
6          q := states[#states];
7          t := tl[index];
8          p := actionTable[[q,t]];
9          match (p) {
10             case om:
11                 return false;
12             case Shift(s):
13                 symbols += [t];
14                 states += [s];
15                 index += 1;
16             case Rule(head, body):
17                 symbols := pop(symbols, #body);
18                 states := pop(states, #body);
19                 symbols += [head];
20                 state := states[#states];
21                 states += [ gotoTable[[state, head]] ];
22             case Accept():
23                 return true;
24             }
25         }
26     };
27     pop := procedure(l, n) { return l[1 .. #l - n]; };

```

Abbildung 14.1: Implementation of a shift-reduce parser in SETLX

3. The variable *states* is the stack of states. The start state is assumed to be the state “s0”. Therefore this variable is initialized to contain only this state.
4. The main loop of the parser
 - sets the variable *q* to the current state,
 - initializes *t* to the next token, and then
 - sets *p* by looking up the appropriate action in the action table. Therefore *p* is equal to $action(q, t)$.

The following actions depend on this value of $action(q, t)$.

- (a) $action(q, t)$ is undefined.

If $action(q, t)$ is undefined, the parser has found an error and returns **false**.

- (b) $action(q, t) = \langle \text{shift}, s \rangle$.

This action is represented in SETLX as the term **Shift**(*s*). In this case, the token *t* is pushed onto the symbol stack in line 13, while the state *s* is pushed onto the stack of states. Furthermore, the variable *index* is incremented to point to the next unread token.

- (c) $action(q, t) = \langle \text{reduce}, A \rightarrow X_1 \cdots X_n \rangle$.

This action is represented as the SETLX term **Rule**(*A*, [*X*₁, \dots , *X*_{*n*}]). In this case, we use the grammar rule

$$r = (A \rightarrow X_1 \cdots X_n)$$

to reduce the symbol stack. The SETLX variable *head* represents the left hand side A of this rule, while the list $[X_1, \dots, X_n]$ is represented by the SETLX variable *body*.

In this case we know that the symbols X_1, \dots, X_n are on top of the symbol stack. As we are going to reduce the symbol stack with the rule r , we remove n symbols from the symbol stack and replace these symbols with the variable A .

Furthermore, we have to remove n states from the stack of states. After that, we set *state* to the state that is then on top of the stack of states. Next, the new state $goto(state, A)$ is put on top of the stack of states in line 19.

(d) $action(q, t) = \text{accept}$.

In this case parsing is successful and therefore the function returns **true**.

In order to make the function *parseSR* work we have to provide an implementation of the functions *action* and *goto*. The tables 14.1 and 14.2 show these functions for the grammar given in Figure 14.2. For this grammar, there are 16 different states, which have been baptized as s_0, s_1, \dots, s_{15} . The tables use two different abbreviations:

1. $\langle shift, s_i \rangle$ is short for $\langle \text{shift}, s_i \rangle$.
2. $\langle rdc, r_i \rangle$ is short for $\langle \text{reduce}, r_i \rangle$, where r_i denotes the grammar rule number i . Here, we have numbered the rules as follows:
 - (a) $r_1 = (Expr \rightarrow Expr \text{ "+" } Product)$
 - (b) $r_2 = (Expr \rightarrow Expr \text{ "-" } Product)$
 - (c) $r_3 = (Expr \rightarrow Product)$
 - (d) $r_4 = (Product \rightarrow Product \text{ "*" } Factor)$
 - (e) $r_5 = (Product \rightarrow Product \text{ "/" } Factor)$
 - (f) $r_6 = (Product \rightarrow Factor)$
 - (g) $r_7 = (Factor \rightarrow \text{"(" } Expr \text{ ")"})$
 - (h) $r_8 = (Factor \rightarrow \text{NUMBER})$

The corresponding grammar is shown in Figure 14.2. The coding of the functions *action* and *goto* is shown in the Figures 14.3, 14.4, and 14.5 on the following pages.

$Expr$	\rightarrow	$Expr \text{ "+" } Product$
	$ $	$Expr \text{ "-" } Product$
	$ $	$Product$
$Product$	\rightarrow	$Product \text{ "*" } Factor$
	$ $	$Product \text{ "/" } Factor$
	$ $	$Factor$
$Factor$	\rightarrow	$\text{"(" } Expr \text{ ")"}$
	$ $	NUMBER

Abbildung 14.2: A grammar for arithmetical expressions.

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 14.1: The function *action()*.

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 14.2: The function *goto()*.

```

1  var actionTable;
2
3  actionTable := {};
4
5  r1 := Rule("E", ["E", "+", "P"]);      r4 := Rule("P", ["P", "*", "F"]);
6  r2 := Rule("E", ["E", "-", "P"]);      r5 := Rule("P", ["P", "/", "F"]);
7  r3 := Rule("E", ["P"]);                r6 := Rule("P", ["F"]);
8
9  r7 := Rule("F", ["(", "E", ")"]);
10 r8 := Rule("F", ["int"]);
11
12 actionTable[["s0", "("]] := Shift("s5");
13 actionTable[["s0", "int"]] := Shift("s2");
14
15 actionTable[["s1", "EOF"]] := r6;      actionTable[["s2", "EOF"]] := r8;
16 actionTable[["s1", "+"]] := r6;      actionTable[["s2", "+"]] := r8;
17 actionTable[["s1", "-"]] := r6;      actionTable[["s2", "-"]] := r8;
18 actionTable[["s1", "*"]] := r6;      actionTable[["s2", "*"]] := r8;
19 actionTable[["s1", "/"]] := r6;      actionTable[["s2", "/"]] := r8;
20 actionTable[["s1", ")"]] := r6;      actionTable[["s2", "("]] := r8;
21                                     actionTable[["s2", ")"]] := r8;
22
23 actionTable[["s3", "EOF"]] := r3;
24 actionTable[["s3", "+"]] := r3;
25 actionTable[["s3", "-"]] := r3;
26 actionTable[["s3", "*"]] := Shift("s12");
27 actionTable[["s3", "/"]] := Shift("s11");
28 actionTable[["s3", ")"]] := r3;
29
30 actionTable[["s4", "EOF"]] := Accept();
31 actionTable[["s4", "+"]] := Shift("s8");
32 actionTable[["s4", "-"]] := Shift("s9");
33
34 actionTable[["s5", "("]] := Shift("s5");
35 actionTable[["s5", "int"]] := Shift("s2");
36
37 actionTable[["s6", "+"]] := Shift("s8");
38 actionTable[["s6", "-"]] := Shift("s9");
39 actionTable[["s6", ")"]] := Shift("s7");
40
41 actionTable[["s7", "EOF"]] := r7;
42 actionTable[["s7", "+"]] := r7;
43 actionTable[["s7", "-"]] := r7;
44 actionTable[["s7", "*"]] := r7;
45 actionTable[["s7", "/"]] := r7;
46 actionTable[["s7", ")"]] := r7;

```

Abbildung 14.3: Action table coded in SETLX, first part.

```

1  actionTable[["s8", "(" ]] := Shift("s5");
2  actionTable[["s8", "int"]] := Shift("s2");
3
4  actionTable[["s9", "(" ]] := Shift("s5");
5  actionTable[["s9", "int"]] := Shift("s2");
6
7  actionTable[["s10", "EOF"]] := r2;
8  actionTable[["s10", "+" ]] := r2;
9  actionTable[["s10", "-" ]] := r2;
10 actionTable[["s10", "*" ]] := Shift("s12");
11 actionTable[["s10", "/" ]] := Shift("s11");
12 actionTable[["s10", ")" ]] := r2;
13
14 actionTable[["s11", "(" ]] := Shift("s5");
15 actionTable[["s11", "int"]] := Shift("s2");
16
17 actionTable[["s12", "(" ]] := Shift("s5");
18 actionTable[["s12", "int"]] := Shift("s2");
19
20 actionTable[["s13", "EOF"]] := r4;
21 actionTable[["s13", "+" ]] := r4;
22 actionTable[["s13", "-" ]] := r4;
23 actionTable[["s13", "*" ]] := r4;
24 actionTable[["s13", "/" ]] := r4;
25 actionTable[["s13", ")" ]] := r4;
26
27 actionTable[["s14", "EOF"]] := r5;
28 actionTable[["s14", "+" ]] := r5;
29 actionTable[["s14", "-" ]] := r5;
30 actionTable[["s14", "*" ]] := r5;
31 actionTable[["s14", "/" ]] := r5;
32 actionTable[["s14", ")" ]] := r5;
33
34 actionTable[["s15", "EOF"]] := r1;
35 actionTable[["s15", "+" ]] := r1;
36 actionTable[["s15", "-" ]] := r1;
37 actionTable[["s15", "*" ]] := Shift("s12");
38 actionTable[["s15", "/" ]] := Shift("s11");
39 actionTable[["s15", ")" ]] := r1;

```

Abbildung 14.4: Action table coded in SETLX, second part.

```
1  var gotoTable;
2  gotoTable    := {};
3
4  gotoTable[["s0", "E"]] := "s4";
5  gotoTable[["s0", "P"]] := "s3";
6  gotoTable[["s0", "F"]] := "s1";
7
8  gotoTable[["s5", "E"]] := "s6";
9  gotoTable[["s5", "P"]] := "s3";
10 gotoTable[["s5", "F"]] := "s1";
11
12 gotoTable[["s8", "P"]] := "s15";
13 gotoTable[["s8", "F"]] := "s1";
14
15 gotoTable[["s9", "P"]] := "s10";
16 gotoTable[["s9", "F"]] := "s1";
17
18 gotoTable[["s11", "F"]] := "s14";
19 gotoTable[["s12", "F"]] := "s13";
```

Abbildung 14.5: Goto table coded in SETLX.

14.3 SLR-Parser

In diesem Abschnitt zeigen wir, wie wir für eine gegebene kontextfreie Grammatik G die im letzten Abschnitt verwendeten Funktionen

$$action : Q \times T \rightarrow Action \quad \text{and} \quad goto : Q \times V \rightarrow Q$$

definieren können. Dazu klären wir als erstes, wie die Menge Q der Zustände zu definieren ist. Wir werden die Zustände 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 den Begriff einer markierten Regel. In der englischen Originalliteratur wird hier unglücklicherweise der nichtssagende Begriff “*item*” verwendet.

Definition 40 (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 darstellen können. Um diese Idee zu veranschaulichen, betrachten wir ein konkretes Beispiel: Wir gehen von der in Abbildung 14.2 auf Seite 191 gezeigten Grammatik aus, wobei wir diese Grammatik noch um ein neues Start-Symbol \hat{S} und die Regel

$$\hat{S} \rightarrow Expr$$

erweitern. Der Start-Zustand enthält offenbar die markierte Regel

$$\hat{S} \rightarrow \varepsilon \bullet Expr,$$

denn am Anfang versuchen wir ja, das Start-Symbol \hat{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. $Factor \rightarrow \bullet \text{NUMBER}$,

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 41 ($\text{closure}(\mathcal{M})$) 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

$$A \rightarrow \alpha \bullet B\beta$$

eine markierte Regel aus der Menge \mathcal{K} , wobei B eine syntaktische Variable ist, und ist andererseits

$$B \rightarrow \gamma$$

eine Grammatik-Regel der zu Grunde liegenden Grammatik G , so ist auch die markierte Regel

$$B \rightarrow \bullet \gamma$$

ein Element der Menge \mathcal{K} . Als Formel schreibt sich dies wie folgt:

$$(A \rightarrow \alpha \bullet B\beta) \in \mathcal{K} \wedge (B \rightarrow \gamma) \in R \Rightarrow (B \rightarrow \bullet \gamma) \in \mathcal{K}$$

Die so definierte Menge \mathcal{K} ist eindeutig bestimmt und wird im Folgenden mit $\text{closure}(\mathcal{M})$ bezeichnet. \square

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 $\mathcal{K} := \text{closure}(\mathcal{M})$ iterativ erfolgen:

1. Zunächst setzen wir $\mathcal{K} := \mathcal{M}$.
2. Anschließend suchen wir alle Regeln der Form

$$A \rightarrow \alpha \bullet B\beta$$

aus der Menge \mathcal{K} , für die B eine syntaktische Variable ist und fügen dann für alle Regeln der Form $B \rightarrow \gamma$ die neue markierte Regel

$$B \rightarrow \bullet \gamma$$

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 14.2 auf Seite 191 gezeigten Grammatik aus und betrachten die Menge

$$\mathcal{M} := \{ \text{Product} \rightarrow \text{Product} \text{ “*” } \bullet \text{Factor} \}$$

Für die Menge $\text{closure}(\mathcal{M})$ finden wir dann

$$\begin{aligned} \text{closure}(\mathcal{M}) = \{ & \text{Product} \rightarrow \text{Product} \text{ “*” } \bullet \text{Factor}, \\ & \text{Factor} \rightarrow \bullet \text{ “(” Expr “)” }, \\ & \text{Factor} \rightarrow \bullet \text{NUMBER} \\ & \} . \end{aligned}$$

\square

Unser Ziel ist es, für eine gegebene kontextfreie Grammatik $G = \langle V, T, R, S \rangle$ einen Shift-Reduce-Parser

$$P = \langle Q, q_0, \text{action}, \text{goto} \rangle$$

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

$$\Gamma := \{A \rightarrow \alpha \bullet \beta \mid (A \rightarrow \alpha\beta) \in R\}$$

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 $\text{closure}()$ abgeschlossen ist, wenn also $\text{closure}(\mathcal{M}) = \mathcal{M}$ gilt. Wir definieren daher

$$Q := \{\mathcal{M} \in 2^\Gamma \mid \text{closure}(\mathcal{M}) = \mathcal{M}\}.$$

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 \hat{S} zu einer Grammatik

$$\hat{G} = \langle V \cup \{\hat{S}\}, T, R \cup \{\hat{S} \rightarrow S, \hat{S}\} \rangle.$$

Diese Grammatik bezeichnen wir als die augmentierte Grammatik. Die Verwendung der augmentierten Grammatik vereinfacht die nun folgende Definition des Start-Zustands. Wir setzen nämlich:

$$q_0 := \text{closure}(\{\hat{S} \rightarrow \bullet S\}).$$

Als nächstes konstruieren wir die Funktion $\text{goto}()$. Die Definition lautet:

$$\text{goto}(\mathcal{M}, B) := \text{closure}(\{A \rightarrow \alpha B \bullet \beta \mid (A \rightarrow \alpha \bullet B\beta) \in \mathcal{M}\}).$$

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 $A \rightarrow \alpha B \beta$ zu erkennen und dass dabei bereits der Teilstring α erkannt wurde. Dieser Zustand wird durch die markierte Regel

$$A \rightarrow \alpha \bullet B\beta$$

beschrieben. Wird nun ein B erkannt, so kann der Parser von dem Zustand, der die Regel $A \rightarrow \alpha \bullet B\beta$ enthält in einen Zustand, der die Regel $A \rightarrow \alpha B \bullet \beta$ enthält, übergehen. Daher erhalten wir die oben angegebene Definition der Funktion $\text{goto}(\mathcal{M}, B)$. Für die gleich folgende Definition der Funktion $\text{action}(\mathcal{M}, t)$ ist es nützlich, die Definition der Funktion goto auf Terminale zu erweitern. Für Terminale t setzen wir:

$$\text{goto}(\mathcal{M}, t) := \text{closure}(\{A \rightarrow \alpha t \bullet \beta \mid (A \rightarrow \alpha \bullet t\beta) \in \mathcal{M}\}).$$

Als letztes spezifizieren wir, wie die Funktion $\text{action}(\mathcal{M}, t)$ für eine Menge von markierten Regeln \mathcal{M} und ein Token t berechnet wird. Bei der Definition von $\text{action}(\mathcal{M}, t)$ unterscheiden wir vier Fälle.

1. Falls \mathcal{M} eine markierte Regel der Form $A \rightarrow \alpha \bullet t\beta$ enthält, setzen wir

$$\text{action}(\mathcal{M}, t) := \langle \text{shift}, \text{goto}(\mathcal{M}, t) \rangle,$$

denn in diesem Fall versucht der Parser ein A mit Hilfe der Regel $A \rightarrow \alpha t \beta$ 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 $A \rightarrow \alpha \bullet t\beta$ in den Zustand $A \rightarrow \alpha t \bullet \beta$ über, der von der Funktion $\text{goto}(\mathcal{M}, t)$ berechnet wird. Insgesamt haben wir also

$$\text{action}(\mathcal{M}, t) := \langle \text{shift}, \text{goto}(\mathcal{M}, t) \rangle \quad \text{falls} \quad (A \rightarrow \alpha \bullet t\beta) \in \mathcal{M}.$$

2. Falls \mathcal{M} eine markierte Regel der Form $A \rightarrow \alpha \bullet$ enthält und wenn zusätzlich $t \in \text{Follow}(A)$ gilt, dann setzen wir

$$\text{action}(\mathcal{M}, t) := \langle \text{reduce}, A \rightarrow \alpha \rangle,$$

denn in diesem Fall versucht der Parser ein A mit Hilfe der Regel $A \rightarrow \alpha$ 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 $t \in \text{Follow}(A)$, so kann der Parser die Regel $A \rightarrow \alpha$ anwenden und den Symbol-Stack mit dieser Regel reduzieren. Wir haben also

$$\text{action}(\mathcal{M}, t) := \langle \text{reduce}, A \rightarrow \alpha \rangle \quad \text{falls } (A \rightarrow \alpha \bullet) \in \mathcal{M}, A \neq \hat{S} \text{ und } t \in \text{Follow}(A).$$

3. Falls \mathcal{M} die markierte Regel $\hat{S} \rightarrow S \bullet$ enthält und wir den zu parsenden String vollständig gelesen haben, dann setzen wir

$$\text{action}(\mathcal{M}, \text{EOF}) := \text{accept},$$

denn in diesem Fall versucht der Parser, \hat{S} mit Hilfe der Regel $\hat{S} \rightarrow S$ 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

$$\text{action}(\mathcal{M}, \text{EOF}) := \text{accept}, \quad \text{falls } (\hat{S} \rightarrow S \bullet) \in \mathcal{M}.$$

4. In den restlichen Fällen setzen wir

$$\text{action}(\mathcal{M}, t) := \text{error}.$$

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

$$A \rightarrow \alpha \bullet t \beta,$$

zum anderen enthält \mathcal{M} eine Regel der Form

$$C \rightarrow \gamma \bullet \quad \text{mit } t \in \text{Follow}(C).$$

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 $A \rightarrow \alpha t \bullet \beta$ übergehen soll, oder ob statt dessen der Symbol-Stack mit der Regel $C \rightarrow \gamma$ reduziert werden muss.

Auch zwischen der ersten und der dritten Regel kann es einen Konflikt geben. Ein solcher Konflikt wird ebenfalls als Shift-Reduce-Konflikt bezeichnet,

2. Eine *Reduce-Reduce-Konflikt* liegt vor, wenn die Menge \mathcal{M} zwei verschiedene markierte Regeln der Form

$$C_1 \rightarrow \gamma_1 \bullet \quad \text{und} \quad C_2 \rightarrow \gamma_2 \bullet$$

enthält und wenn gleichzeitig $t \in \text{Follow}(C_1) \cap \text{Follow}(C_2)$ 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 geparkt 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 14.2 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

$$\text{start} \rightarrow \text{expr}$$

ein. Als erstes berechnen wir die Menge der Zustände Q . Wir hatten dafür oben die folgende Formel angegeben:

$$Q := \{\mathcal{M} \in 2^\Gamma \mid \text{closure}(\mathcal{M}) = \mathcal{M}\}.$$

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 Abkürzungen ein:

$$S := \text{start}, E := \text{expr}, P := \text{product}, F := \text{factor}, N := \text{NUMBER}.$$

Wir beginnen mit dem Start-Zustand:

$$\begin{aligned} s_0 &:= \text{closure}(\{ S \rightarrow \bullet E \}) \\ &= \{ S \rightarrow \bullet E, \\ &\quad E \rightarrow \bullet E \text{ "+" } P, E \rightarrow \bullet E \text{ "-" } P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P \text{ "*" } F, P \rightarrow \bullet P \text{ "/" } F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet \text{"(" } E \text{ ")"}, F \rightarrow \bullet N \} \end{aligned}$$

Als nächstes berechnen wir $goto(s_0, F)$. Wir bezeichnen den resultierenden Zustand mit s_1 .

$$\begin{aligned} s_1 &:= goto(s_0, F) \\ &= \text{closure}(\{ P \rightarrow F \bullet \}) \\ &= \{ P \rightarrow F \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_0, N)$.

$$\begin{aligned} s_2 &:= goto(s_0, N) \\ &= \text{closure}(\{ F \rightarrow N \bullet \}) \\ &= \{ F \rightarrow N \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_0, P)$.

$$\begin{aligned} s_3 &:= goto(s_0, P) \\ &= \text{closure}(\{ P \rightarrow P \bullet \text{"*" } F, P \rightarrow P \bullet \text{"/" } F, E \rightarrow P \bullet \}) \\ &= \{ P \rightarrow P \bullet \text{"*" } F, P \rightarrow P \bullet \text{"/" } F, E \rightarrow P \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_0, E)$.

$$\begin{aligned} s_4 &:= goto(s_0, E) \\ &= \text{closure}(\{ S \rightarrow E \bullet, E \rightarrow E \bullet \text{"+" } P, E \rightarrow E \bullet \text{"-" } P \}) \\ &= \{ S \rightarrow E \bullet, E \rightarrow E \bullet \text{"+" } P, E \rightarrow E \bullet \text{"-" } P \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_0, \text{"("})$.

$$\begin{aligned} s_5 &:= goto(s_0, \text{"("}) \\ &= \text{closure}(\{ F \rightarrow \text{"(" } \bullet E \text{ ")" } \}) \\ &= \{ F \rightarrow \text{"(" } \bullet E \text{ ")" } \\ &\quad E \rightarrow \bullet E \text{"+" } P, E \rightarrow \bullet E \text{"-" } P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P \text{"*" } F, P \rightarrow \bullet P \text{"/" } F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet \text{"(" } E \text{ ")"}, F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_5, E)$.

$$\begin{aligned} s_6 &:= goto(s_5, E) \\ &= \text{closure}(\{ F \rightarrow \text{"(" } E \bullet \text{ ")"}, E \rightarrow E \bullet \text{"+" } P, E \rightarrow E \bullet \text{"-" } P \}) \\ &= \{ F \rightarrow \text{"(" } E \bullet \text{ ")"}, E \rightarrow E \bullet \text{"+" } P, E \rightarrow E \bullet \text{"-" } P \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_6, \text{"}")$.

$$\begin{aligned} s_7 &:= goto(s_6, \text{""}) \\ &= \text{closure}(\{ F \rightarrow \text{"(" } E \text{ ")" } \bullet \}) \\ &= \{ F \rightarrow \text{"(" } E \text{ ")" } \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_4, "+")$.

$$\begin{aligned} s_8 &:= goto(s_4, "+") \\ &= closure(\{E \rightarrow E "+" \bullet P\}) \\ &= \{ E \rightarrow E "+" \bullet P \\ &\quad P \rightarrow \bullet P "*" F, P \rightarrow \bullet P "/" F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_4, "-")$.

$$\begin{aligned} s_9 &:= goto(s_4, "-") \\ &= closure(\{E \rightarrow E "-" \bullet P\}) \\ &= \{ E \rightarrow E "-" \bullet P \\ &\quad P \rightarrow \bullet P "*" F, P \rightarrow \bullet P "/" F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_9, P)$.

$$\begin{aligned} s_{10} &:= goto(s_9, P) \\ &= closure(\{E \rightarrow E "-" P \bullet, P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F\}) \\ &= \{ E \rightarrow E "-" P \bullet, P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_3, "/")$.

$$\begin{aligned} s_{11} &:= goto(s_3, "/") \\ &= closure(\{P \rightarrow P "/" \bullet F\}) \\ &= \{ P \rightarrow P "/" \bullet F, F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_3, "*")$.

$$\begin{aligned} s_{12} &:= goto(s_3, "*") \\ &= closure(\{P \rightarrow P "*" \bullet F\}) \\ &= \{ P \rightarrow P "*" \bullet F, F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_{12}, F)$.

$$\begin{aligned} s_{13} &:= goto(s_{12}, F) \\ &= closure(\{P \rightarrow P "*" F \bullet\}) \\ &= \{ P \rightarrow P "*" F \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $goto(s_{11}, F)$.

$$\begin{aligned} s_{14} &:= goto(s_{11}, F) \\ &= closure(\{P \rightarrow P "/" F \bullet\}) \\ &= \{ P \rightarrow P "/" F \bullet \}. \end{aligned}$$

Als letztes berechnen wir $goto(s_8, P)$.

$$\begin{aligned} s_{15} &:= goto(s_8, P) \\ &= closure(\{E \rightarrow E "+" P \bullet, P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F\}) \\ &= \{ E \rightarrow E "+" P \bullet, P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F \}. \end{aligned}$$

Weitere Rechnungen führen nicht mehr auf neue Zustände. Berechnen wir beispielsweise $goto(s_8, "(")$, so finden wir

$$\begin{aligned} &goto(s_8, "(") \\ &= closure(\{F \rightarrow "(" \bullet E ")"\}) \\ &= \{ F \rightarrow "(" \bullet E ")" \\ &\quad E \rightarrow \bullet E "+" P, E \rightarrow \bullet E "-" P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P "*" F, P \rightarrow \bullet P "/" F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N \} \\ &= s_5. \end{aligned}$$

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

$$P \rightarrow P \bullet \text{ “*” } F$$

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

$$E \rightarrow E \text{ “+” } P \bullet.$$

Diese Regel sagt, dass der Symbol-Stack mit der Grammatik-Regel $E \rightarrow E \text{ “+” } P$ 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

$$Follow(E) = \{ \text{“+”}, \text{“-”}, \text{“)”}, \text{“$”} \}$$

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

$$\begin{aligned} action(s_1, \text{“+”}) &= action(\{P \rightarrow F \bullet\}, \text{“+”}) \\ &= \langle reduce, P \rightarrow F \rangle, \end{aligned}$$

denn wir haben “+” $\in Follow(P)$.

2. Als nächstes berechnen wir $action(s_4, \text{“+”})$. Es gilt

$$\begin{aligned} action(s_4, \text{“+”}) &= action(\{S \rightarrow E \bullet, E \rightarrow E \bullet \text{ “+” } P, E \rightarrow E \bullet \text{ “-” } P\}, \text{“+”}) \\ &= \langle shift, closure(\{E \rightarrow E \text{ “+” } \bullet P\}) \rangle \\ &= \langle shift, s_8 \rangle. \end{aligned}$$

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 14.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 14.1 und 14.2 übereinstimmen.

Aufgabe 28: Berechnen Sie die Menge der SLR-Zustände für die in Abbildung 14.6 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.

Hinweis: Damit Sie später Ihre Ergebnis vergleichen können, ist es sinnvoll, die Namen der Zustände wie folgt festzulegen:

1. $s_0 := closure(\{S \rightarrow \bullet C\})$,
2. $s_1 := goto(s_0, C)$,
3. $s_2 := goto(s_0, \text{“!”})$,
4. $s_3 := goto(s_0, L)$,
5. $s_4 := goto(s_0, I)$,
6. $s_5 := goto(s_0, D)$,
7. $s_6 := goto(s_5, \text{“|”})$,

<i>Conjunction</i>	\rightarrow	<i>Conjunction</i> "&" <i>Disjunction</i>
		<i>Disjunction</i>
<i>Disjunction</i>	\rightarrow	<i>Disjunction</i> " " <i>Literal</i>
		<i>Literal</i>
<i>Literal</i>	\rightarrow	"!" IDENTIFIER
		IDENTIFIER

Abbildung 14.6: Eine Grammatik für Boole'sche Ausdrücke in konjunktiver Normalform.

8. $s_7 := goto(s_6, L)$,
9. $s_8 := goto(s_2, I)$,
10. $s_9 := goto(s_1, "&")$,
11. $s_{10} := goto(s_9, D)$.

14.3.1 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 14.7 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 Abkürzung 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.

E	\rightarrow	$E \text{ "+" } E$
		$E \text{ "*" } E$
		N

Abbildung 14.7: Eine Grammatik mit Shift-Reduce-Konflikten.

Wir berechnen zunächst den Start-Zustand s_0 .

$$\begin{aligned} s_0 &= \text{closure}(\{S \rightarrow \bullet E\}) \\ &= \{S \rightarrow \bullet E, E \rightarrow \bullet E \text{ "+" } E, E \rightarrow \bullet E \text{ "*" } E, E \rightarrow \bullet N\}. \end{aligned}$$

Als nächstes berechnen wir $s_1 := goto(s_0, E)$:

$$\begin{aligned} s_1 &= goto(s_0, E) \\ &= \text{closure}(\{S \rightarrow E \bullet, E \rightarrow E \bullet \text{ "+" } E, E \rightarrow E \bullet \text{ "*" } E\}) \\ &= \{S \rightarrow E \bullet, E \rightarrow E \bullet \text{ "+" } E, E \rightarrow E \bullet \text{ "*" } E\} \end{aligned}$$

Nun berechnen wir $s_2 := goto(s_1, "+")$:

$$\begin{aligned} s_2 &= goto(s_1, "+") \\ &= \text{closure}(\{E \rightarrow E \text{ "+" } \bullet E, \}) \\ &= \{E \rightarrow E \text{ "+" } \bullet E, E \rightarrow \bullet E \text{ "+" } E, E \rightarrow \bullet E \text{ "*" } E, E \rightarrow \bullet N\} \end{aligned}$$

Als nächstes berechnen wir $s_3 := \text{goto}(s_2, E)$:

$$\begin{aligned} s_3 &= \text{goto}(s_2, E) \\ &= \text{closure}(\{E \rightarrow E \text{ “+” } E \bullet, E \rightarrow E \bullet \text{ “+” } E, E \rightarrow E \bullet \text{ “*” } E\}) \\ &= \{E \rightarrow E \text{ “+” } E \bullet, E \rightarrow E \bullet \text{ “+” } E, E \rightarrow E \bullet \text{ “*” } E\} \end{aligned}$$

Hier tritt bei der Berechnung von $\text{action}(s_3, \text{“*”})$ ein Shift-Reduce-Konflikt auf, denn einerseits verlangt die markierte Regel

$$E \rightarrow E \bullet \text{ “*” } E,$$

dass das Token “*” auf den Stack geschoben wird, andererseits haben wir

$$\text{Follow}(E) = \{ \text{“+”}, \text{“*”}, \text{“$”} \},$$

so dass, falls das nächste zu lesende Token den Wert “*” hat, der Symbol-Stack mit der Regel

$$E \rightarrow E \text{ “+” } E \bullet$$

reduziert werden sollte.

Aufgabe 29: Bei der in Abbildung 14.7 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. \square

Wir untersuchen als nächstes eine Grammatik, die keine SLR-Grammatik ist, weil Reduce-Reduce-Konflikte auftreten. Wir betrachten dazu die in Abbildung 14.8 gezeigte Grammatik. Diese Grammatik ist eindeutig, denn es gilt

$$L(S) = \{ \text{“xy”}, \text{“yx”} \}$$

und der String “xy” lässt sich nur mit der Regel $S \rightarrow A \text{ “x” } A \text{ “y”}$ herleiten, während sich der String “yx” nur mit der Regel $S \rightarrow B \text{ “y” } B \text{ “x”}$ erzeugen lässt. Um zu zeigen, dass diese Grammatik Shift-Reduce-Konflikte enthält, berechnen wir den Start-Zustand eines SLR-Parsers für diese Grammatik.

$$\begin{aligned} S &\rightarrow A \text{ “x” } A \text{ “y”} \\ &\quad | \quad B \text{ “y” } B \text{ “x”} \\ A &\rightarrow \varepsilon \\ B &\rightarrow \varepsilon \end{aligned}$$

Abbildung 14.8: Eine Grammatik mit einem Reduce-Reduce-Konflikt.

$$\begin{aligned} s_0 &= \text{closure}(\{\hat{S} \rightarrow \bullet S\}) \\ &= \{\hat{S} \rightarrow \bullet S, S \rightarrow \bullet A \text{ “x” } A \text{ “y”}, S \rightarrow \bullet B \text{ “y” } B \text{ “x”}, A \rightarrow \bullet \varepsilon, B \rightarrow \bullet \varepsilon\} \\ &= \{\hat{S} \rightarrow \bullet S, S \rightarrow \bullet A \text{ “x” } A \text{ “y”}, S \rightarrow \bullet B \text{ “y” } B \text{ “x”}, A \rightarrow \varepsilon \bullet, B \rightarrow \varepsilon \bullet\}, \end{aligned}$$

denn $A \rightarrow \bullet \varepsilon$ ist das Selbe wie $A \rightarrow \varepsilon \bullet$. In diesem Zustand gibt es einen Reduce-Reduce-Konflikt zwischen den beiden markierten Regeln

$$A \rightarrow \bullet \varepsilon \quad \text{und} \quad B \rightarrow \varepsilon \bullet.$$

Dieser Konflikt tritt bei der Berechnung von

$$\text{action}(s_0, \text{"x"})$$

auf, denn wir haben

$$\text{Follow}(A) = \{\text{"x"}, \text{"y"}\} = \text{Follow}(B)$$

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 $\text{Follow}(A)$ als auch der Menge $\text{Follow}(B)$.

Es ist interessant zu bemerken, dass die obige Grammatik die $LL(1)$ -Eigenschaft hat, denn es gilt

$$\text{First}(A \text{"x"} A \text{"y"}) = \{\text{"x"}\}, \quad \text{First}(B \text{"y"} B \text{"x"}) = \{\text{"y"}\}$$

und daraus folgt sofort

$$\text{First}(A \text{"x"} A \text{"y"}) \cap \text{First}(B \text{"y"} B \text{"x"}) = \{\text{"x"}\} \cap \{\text{"y"}\} = \{\}$$

Dieses Beispiel zeigt, dass SLR-Grammatiken im Allgemeinen nicht ausdrückstärker sind als $LL(1)$ -Grammatiken. In der Praxis zeigt sich jedoch, dass viele Grammatiken, die nicht die $LL(1)$ -Eigenschaft haben, SLR-Grammatiken sind.

14.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 14.8 gezeigten Grammatik auftritt, kann wie folgt gelöst werden: In dem Zustand

$$\begin{aligned} s_0 &= \text{closure}(\{\hat{S} \rightarrow \bullet S\}) \\ &= \{\hat{S} \rightarrow \bullet S, S \rightarrow \bullet A \text{"x"} A \text{"y"}, S \rightarrow \bullet B \text{"y"} B \text{"x"}, A \rightarrow \varepsilon \bullet, B \rightarrow \varepsilon \bullet\} \end{aligned}$$

kommen die markierten Regeln $A \rightarrow \varepsilon \bullet$ und $B \rightarrow \varepsilon \bullet$ von der Berechnung des Abschlusses der Regeln

$$S \rightarrow \bullet A \text{"x"} A \text{"y"} \quad \text{und} \quad S \rightarrow \bullet B \text{"y"} B \text{"x"}.$$

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 $\text{Follow}(A)$ bzw. $\text{Follow}(B)$ enthalten ist, denn jetzt berücksichtigen wir den Kontext, in dem die syntaktische Variable auftaucht. Damit können wir die Funktion $\text{action}(s_0, \text{"x"})$ und $\text{action}(s_0, \text{"y"})$ wie folgt definieren:

$$\text{action}(s_0, \text{"x"}) = \langle \text{reduce}, A \rightarrow \varepsilon \rangle \quad \text{und} \quad \text{action}(s_0, \text{"y"}) = \langle \text{reduce}, B \rightarrow \varepsilon \rangle.$$

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 42 (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. $(A \rightarrow \alpha\beta) \in R$.
2. $L \subseteq T$.

Wir schreiben die erweiterte markierte Regel $\langle A, \alpha, \beta, L \rangle$ als

$$A \rightarrow \alpha \bullet \beta : L.$$

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

$$A \rightarrow \alpha \bullet \beta : t.$$

□

Anschaulich interpretieren wir die e.m.R. $A \rightarrow \alpha \bullet \beta : L$ als einen Zustand, in dem folgendes gilt:

1. Der Parser versucht, ein A mit Hilfe der Grammatik-Regel $A \rightarrow \alpha\beta$ zu erkennen.
2. Dabei wurde bereits α erkannt. Damit die Regel $A \rightarrow \alpha\beta$ 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-Tokens*.

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 43 (*closure*(\mathcal{M})) Es sei \mathcal{M} eine Menge erweiterter markierter Regeln. Dann definieren wir den *Abschluss* von \mathcal{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

$$A \rightarrow \alpha \bullet B\beta : L$$

eine e.m.R. aus der Menge \mathcal{K} , wobei B eine syntaktische Variable ist, und ist andererseits

$$B \rightarrow \gamma$$

eine Grammatik-Regel der zu Grunde liegenden Grammatik G , so ist auch die e.m.R.

$$B \rightarrow \bullet \gamma : \bigcup \{ \text{First}(\beta t) \mid t \in L \}$$

ein Element der Menge \mathcal{K} .

Die so definierte eindeutig bestimmte Menge \mathcal{K} wird wieder mit *closure*(\mathcal{M}) bezeichnet. □

Bemerkung: Gegenüber der alten Definition ist nur die Berechnung der Menge der Folge-Tokens hinzu gekommen. Der Kontext, in dem das B auftritt, das mit der Regel $B \rightarrow \gamma$ erkannt werden soll, ist zunächst durch den String β gegeben, der in der Regel $A \rightarrow \alpha \bullet B\beta : L$ auf das B folgt. Möglicherweise leitet β den leeren String ε ab. In diesen Fall spielen auch die Folge-Tokens aus der Menge L eine Rolle, denn falls $\beta \Rightarrow^* \varepsilon$ 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 $\mathcal{K} := \text{closure}(\mathcal{M})$ iterativ erfolgen. Abbildung 14.9 zeigt die Berechnung von *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 *closure*(\mathcal{M}) aufnehmen, bei der Menge der Folge-Tokens 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.

Bemerkung: Der Ausdruck $\bigcup \{ \text{First}(\beta t) \mid t \in L \}$ 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

$$\bigcup \{ \text{First}(\beta t) \mid t \in L \} = \begin{cases} \text{First}(\beta) \cup L & \text{falls } \beta \Rightarrow^* \varepsilon; \\ \text{First}(\beta) & \text{sonst.} \end{cases}$$

Die Berechnung von *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:

$$\text{goto}(\mathcal{M}, X) := \text{closure} \left(\{ A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X\beta : L) \in \mathcal{M} \} \right).$$

```

1  procedure closure( $\mathcal{M}$ ) {
2       $\mathcal{K} := \mathcal{M}$ ;
3       $\mathcal{K}^- := \{\}$ ;
4      while ( $\mathcal{K}^- \neq \mathcal{K}$ ) {
5           $\mathcal{K}^- := \mathcal{K}$ ;
6           $\mathcal{K} := \mathcal{K} \cup \{(B \rightarrow \bullet \gamma : \bigcup \{\text{First}(\beta t) \mid t \in L\}) \mid$ 
7               $(A \rightarrow \alpha \bullet B \beta : L) \in \mathcal{K} \wedge (B \rightarrow \gamma) \in R \}$ ;
8      }
9      return  $\mathcal{K}$ ;
10 }

```

Abbildung 14.9: Berechnung von $\text{closure}(\mathcal{M})$

Genau wie bei der Theorie der SLR-Parser augmentieren wir unsere Grammatik G , indem wir der Menge der Variable eine neue Start-Variable \hat{S} und der Menge der Regeln die neue Regel $\hat{S} \rightarrow S$ hinzufügen. Dann hat der Start-Zustand die Form

$$q_0 := \text{closure}(\{\hat{S} \rightarrow \bullet S : \text{EOF}\}),$$

denn auf das Start-Symbol muss das Datei-Ende “EOF” folgen. Als letztes zeigen wir, wie die Definition der Funktion $\text{action}()$ geändert werden muss. Wir spezifizieren die Berechnung dieser Funktion durch die folgenden bedingten Gleichungen.

1. $(A \rightarrow \alpha \bullet t \beta : L) \in \mathcal{M} \implies \text{action}(\mathcal{M}, t) := \langle \text{shift}, \text{goto}(\mathcal{M}, t) \rangle.$
2. $(A \rightarrow \alpha \bullet : L) \in \mathcal{M} \wedge A \neq \hat{S} \wedge t \in L \implies \text{action}(\mathcal{M}, t) := \langle \text{reduce}, A \rightarrow \alpha \rangle.$
3. $(\hat{S} \rightarrow S \bullet : \text{EOF}) \in \mathcal{M} \implies \text{action}(\mathcal{M}, \text{EOF}) := \text{accept}.$
4. Sonst: $\text{action}(\mathcal{M}, t) := \text{error}.$

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 $\text{action}(\mathcal{M}, t)$ dann vor, wenn es zwei e.m.R.s

$$(A \rightarrow \alpha \bullet t \beta : L_1) \in \mathcal{M} \quad \text{und} \quad (B \rightarrow \gamma \bullet : L_2) \in \mathcal{M} \quad \text{mit } t \in L_2$$

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 $B \rightarrow \gamma$ 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 $t \in \text{Follow}(B)$ gilt und die Menge L_2 ist in der Regel kleiner als die Menge $\text{Follow}(B)$.

Ein *Reduce-Reduce-Konflikt* liegt vor, wenn es zwei e.m.R.s

$$(A \rightarrow \alpha \bullet : L_1) \in \mathcal{M} \quad \text{und} \quad (B \rightarrow \beta \bullet : L_2) \in \mathcal{M} \quad \text{mit } L_1 \cap L_2 \neq \{\}$$

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 $\text{Follow}(A) \cap \text{Follow}(B)$ 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 2. und der 3. Regel möglich. Da die 3. Regel als ein Spezialfall der 2. Regel angesehen werden kann, sprechen wir dann ebenfalls

von einem Reduce-Reduce-Konflikt.

Beispiel: Wir greifen das Beispiel der in Abbildung 14.8 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände. Um die Schreibweise zu vereinfachen, schreiben wir an Stelle von “EOF” kürzer “\$”.

1. $s_0 := \text{closure}\left(\{\widehat{S} \rightarrow \bullet S : \$\}\right)$
 $= \{\widehat{S} \rightarrow \bullet S : \$, S \rightarrow \bullet A \text{“x”} A \text{“y”} : \$, S \rightarrow \bullet B \text{“y”} B \text{“x”} : \$, A \rightarrow \bullet : \text{“x”}, B \rightarrow \bullet : \text{“y”}\}.$
2. $s_1 := \text{goto}(s_0, A)$
 $= \text{closure}\left(\{S \rightarrow A \bullet \text{“x”} A \text{“y”} : \$\}\right)$
 $= \{S \rightarrow A \bullet \text{“x”} A \text{“y”} : \$\}.$
3. $s_2 := \text{goto}(s_0, S)$
 $= \text{closure}\left(\{\widehat{S} \rightarrow S \bullet : \$\}\right)$
 $= \{\widehat{S} \rightarrow S \bullet : \$\}.$
4. $s_3 := \text{goto}(s_0, B)$
 $= \text{closure}\left(\{S \rightarrow B \bullet \text{“y”} B \text{“x”} : \$\}\right)$
 $= \{S \rightarrow B \bullet \text{“y”} B \text{“x”} : \$\}.$
5. $s_4 := \text{goto}(s_3, \text{“y”})$
 $= \text{closure}\left(\{S \rightarrow B \text{“y”} \bullet B \text{“x”} : \$\}\right)$
 $= \{S \rightarrow B \text{“y”} \bullet B \text{“x”} : \$, B \rightarrow \bullet : \text{“x”}\}.$
6. $s_5 := \text{goto}(s_4, B)$
 $= \text{closure}\left(\{S \rightarrow B \text{“y”} B \bullet \text{“x”} : \$\}\right)$
 $= \{S \rightarrow B \text{“y”} B \bullet \text{“x”} : \$\}.$
7. $s_6 := \text{goto}(s_5, \text{“x”})$
 $= \text{closure}\left(\{S \rightarrow B \text{“y”} B \text{“x”} \bullet : \$\}\right)$
 $= \{S \rightarrow B \text{“y”} B \text{“x”} \bullet : \$\}.$
8. $s_7 := \text{goto}(s_1, \text{“x”})$
 $= \text{closure}\left(\{S \rightarrow A \text{“x”} \bullet A \text{“y”} : \$\}\right)$
 $= \{S \rightarrow A \text{“x”} \bullet A \text{“y”} : \$, A \rightarrow \bullet : \text{“y”}\}.$
9. $s_8 := \text{goto}(s_7, A)$
 $= \text{closure}\left(\{S \rightarrow A \text{“x”} A \bullet \text{“y”} : \$\}\right)$
 $= \{S \rightarrow A \text{“x”} A \bullet \text{“y”} : \$\}.$
10. $s_9 := \text{goto}(s_8, \text{“y”})$
 $= \text{closure}\left(\{S \rightarrow A \text{“x”} A \text{“y”} \bullet : \$\}\right)$
 $= \{S \rightarrow A \text{“x”} A \text{“y”} \bullet : \$\}.$

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 $A \rightarrow \varepsilon$ und $B \rightarrow \varepsilon$ gefunden, weil

$$\text{Follow}(A) \cap \text{Follow}(B) = \{\text{“x”}, \text{“y”}\} \neq \{\}$$

gilt. Dieser Konflikt ist nun verschwunden, denn zwischen den e.m.R.s

$$A \rightarrow \bullet : \text{“x”} \quad \text{und} \quad B \rightarrow \bullet : \text{“y”}$$

gibt es wegen “x” \neq “y” keinen Konflikt. Es ist leicht zu sehen, dass auch bei den anderen Zustände keine Konflikte auftreten.

Aufgabe 30: Berechnen Sie die Menge der Zustände eines LR-Parser für die folgende Grammatik:

$$\begin{aligned}
 E &\rightarrow E \text{ “+” } P \\
 &\quad | \quad P \\
 P &\rightarrow P \text{ “+” } F \\
 &\quad | \quad F \\
 F &\rightarrow \text{ “(” } E \text{ “)” } \\
 &\quad | \quad \text{Number}
 \end{aligned}$$

Untersuchen Sie außerdem, ob es bei dieser Grammatik Shift-Reduce-Konflikte oder Reduce-Reduce-Konflikte gibt.

Bemerkung: Die Theorie der kanonischen LR-Parser geht auf Donald E. Knuth zurück, der diese Theorie 1965 entwickelt hat [Knu65].

14.5 LALR-Parser

Die Zahl der Zustände eines LR-Parser 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 14.10 gezeigte Grammatik, die ich dem *Drachenbuch* [ASUL06] entnommen habe. (Das “Drachenbuch” ist das Standardwerk im Bereich Compilerbau.)

$$\begin{aligned}
 \hat{S} &\rightarrow S \\
 S &\rightarrow C C \\
 C &\rightarrow \text{“x” } C \\
 &\quad | \quad \text{“y”}
 \end{aligned}$$

Abbildung 14.10: Eine Grammatik aus dem Drachenbuch.

Abbildung 14.11 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

$$s_6 = \left\{ S \rightarrow \text{“x”} \bullet C : \text{“$”}, C \rightarrow \bullet \text{“x” } C : \text{“$”}, C \rightarrow \bullet \text{“y”} : \text{“$”} \right\},$$

und andererseits haben wir

$$s_3 = \left\{ S \rightarrow \text{“x”} \bullet C : \{ \text{“x”}, \text{“y”} \}, C \rightarrow \bullet \text{“x” } C : \{ \text{“x”}, \text{“y”} \}, C \rightarrow \bullet \text{“y”} : \{ \text{“x”}, \text{“y”} \} \right\}.$$

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

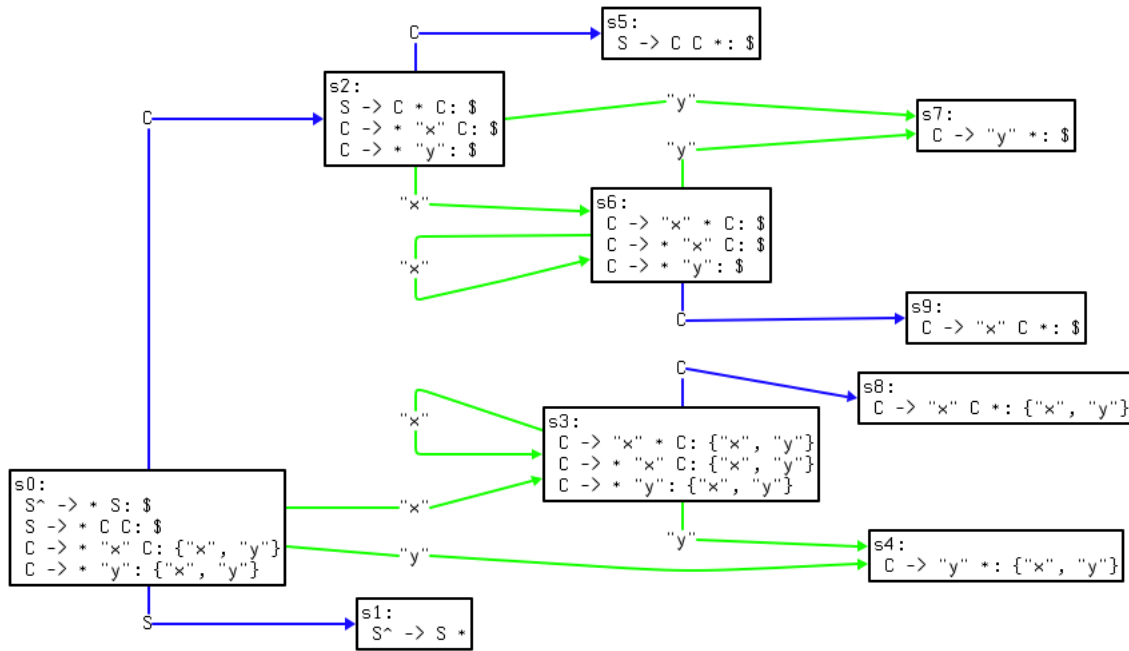


Abbildung 14.11: LR-Goto-Graph für die Grammatik aus Abbildung 14.10.

$$\text{goto}(s_3, C) = s_8 \quad \text{und} \quad \text{goto}(s_6, C) = s_9$$

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 14.12 gezeigten Goto-Graphen.

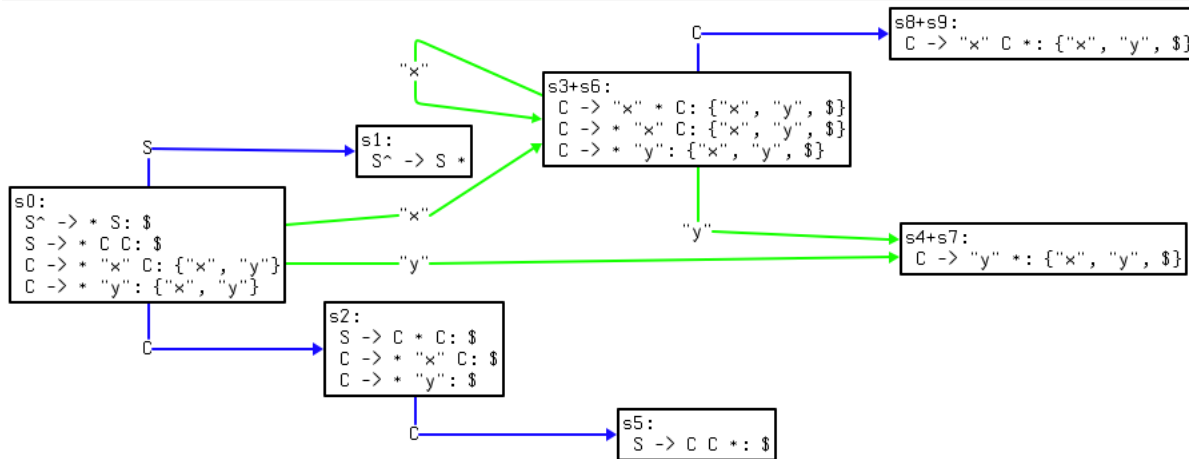


Abbildung 14.12: Der LALR-Goto-Graph für die Grammatik aus Abbildung 14.10.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 14.10 gezeigten Grammatik gemacht gaben, verallgemeinern und formalisieren zu können, definieren wir eine Funktion $\text{core}()$, die den Kern einer Menge von e.m.R.s berechnet und also diese Menge in eine Menge markierter Regeln überführt:

$$\text{core}(\mathcal{M}) := \{A \rightarrow \alpha \bullet \beta \mid (A \rightarrow \alpha \bullet \beta : L) \in \mathcal{M}\}.$$

Die Funktion $\text{core}()$ 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

$$goto(\mathcal{M}, X) := \text{closure}\left(\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X \beta : L) \in \mathcal{M}\}\right).$$

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:

$$\text{core}(\mathcal{M}_1) = \text{core}(\mathcal{M}_2) \Rightarrow \text{core}(goto(\mathcal{M}_1, X)) = \text{core}(goto(\mathcal{M}_2, X)).$$

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

$$\mathcal{M} \uplus \mathcal{N} := \{A \rightarrow \alpha \bullet \beta : K \cup L \mid (A \rightarrow \alpha \bullet \beta : K) \in \mathcal{M} \wedge (A \rightarrow \alpha \bullet \beta : L) \in \mathcal{N}\}.$$

Diese Definition verallgemeinern wir zu einer Operation \uplus , die auf einer Menge von Mengen von e.m.R.s definiert ist: Ist \mathcal{J} eine Menge von Mengen von e.m.R.s, die alle den gleichen Kern haben, gilt also

$$\mathcal{J} = \{\mathcal{M}_1, \dots, \mathcal{M}_k\} \quad \text{mit} \quad \text{core}(\mathcal{M}_i) = \text{core}(\mathcal{M}_j) \quad \text{für alle } i, j \in \{1, \dots, k\},$$

so definieren wir

$$\uplus \mathcal{J} := \mathcal{M}_1 \uplus \dots \uplus \mathcal{M}_k.$$

Es sei nun Δ die Menge aller Zustände eines LR-Parasers. Dann ist die Menge der Zustände des entsprechenden LALR-Parasers durch die erweiterte Vereinigung der Menge aller der Teilmengen von Δ gegeben, deren Elemente den gleichen Kern haben:

$$\Omega := \left\{ \uplus \mathcal{J} \mid \mathcal{J} \in 2^\Delta \wedge \forall \mathcal{M}, \mathcal{N} \in \mathcal{J} : \text{core}(\mathcal{M}) = \text{core}(\mathcal{N}) \wedge \text{und } \mathcal{J} \text{ maximal} \right\}.$$

Die Forderung “ \mathcal{J} maximal” drückt in der obigen Definition aus, dass in \mathcal{J} tatsächlich alle Mengen aus Δ zusammengefasst sind, die den selben Kern haben. Die so definierte Menge Ω ist die Menge der LALR-Zustände.

Als nächstes überlegen wir, wie sich die Berechnung von $goto(\mathcal{M}, X)$ ändern muss, wenn \mathcal{M} ein Element der Menge Ω der LALR-Zustände ist. Zur Berechnung von $goto(\mathcal{M}, X)$ berechnen wir zunächst die Menge

$$\text{closure}\left(\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X \beta : L) \in \mathcal{M}\}\right).$$

Das Problem ist, dass diese Menge im Allgemeinen kein Element der Menge Ω ist, denn die Zustände in Ω entstehen ja durch die Zusammenfassung mehrerer LR-Zustände. Die Zustände, die bei der Berechnung von Ω zusammengefasst werden, haben aber alle den selben Kern. Daher enthält die Menge

$$\left\{ q \in \Omega \mid \text{core}(q) = \text{core}(\text{closure}(\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X \beta : L) \in \mathcal{M}\})) \right\}$$

genau ein Element und dieses Element ist der Wert von $goto(\mathcal{M}, X)$. Folglich können wir

$$goto(\mathcal{M}, X) :=$$

$$\text{arb}\left(\left\{ q \in \Omega \mid \text{core}(q) = \text{core}(\text{closure}(\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X \beta : L) \in \mathcal{M}\})) \right\}\right)$$

setzen. Die hier verwendete Funktion $\text{arb}()$ dient dazu, ein beliebiges Element aus einer Menge zu extrahieren. Da die Menge, aus der hier das Element extrahiert wird, genau ein Element enthält, ist $goto(\mathcal{M}, X)$ wohldefiniert. Die Berechnung des Ausdrucks $\text{action}(\mathcal{M}, t)$ ändert sich gegenüber der Berechnung für einen LR-Parser nicht.

14.6 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:

$$\text{SLR-Sprache} \subsetneq \text{LALR-Sprache} \subsetneq \text{kanonische LR-Sprache} \quad (\star)$$

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.

14.6.1 *SLR-Sprache* \subsetneq *LALR-Sprache*

Die Zustände eines LALR-Parers enthalten gegenüber den Zuständen eines SLR-Parers 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 204 gesehen, dass die Grammatik

$$S \rightarrow A \text{ "x" } A \text{ "y" } \mid B \text{ "y" } B \text{ "x" }, \quad A \rightarrow \varepsilon, \quad B \rightarrow \varepsilon$$

keine SLR-Grammatik ist. Später hatten wir gesehen, dass diese Grammatik von einem kanonischen LR-Parser geparkt werden kann. Wir zeigen nun, dass diese Grammatik auch von einem LALR-Parser geparkt 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. $s_0 = \{\widehat{S} \rightarrow \bullet S : \$, S \rightarrow \bullet A \text{ "x" } A \text{ "y" } : \$, S \rightarrow \bullet B \text{ "y" } B \text{ "x" } : \$, A \rightarrow \bullet : \text{ "x" }, B \rightarrow \bullet : \text{ "y" }\},$
2. $s_1 = \{S \rightarrow A \bullet \text{ "x" } A \text{ "y" } : \$\},$
3. $s_2 = \{\widehat{S} \rightarrow S \bullet : \$\},$
4. $s_3 = \{S \rightarrow B \bullet \text{ "y" } B \text{ "x" } : \$\},$
5. $s_4 = \{S \rightarrow B \text{ "y" } \bullet B \text{ "x" } : \$, B \rightarrow \bullet : \text{ "x" }\},$
6. $s_5 = \{S \rightarrow B \text{ "y" } B \bullet \text{ "x" } : \$\},$
7. $s_6 = \{S \rightarrow B \text{ "y" } B \text{ "x" } \bullet : \$\},$
8. $s_7 = \{S \rightarrow A \text{ "x" } \bullet A \text{ "y" } : \$, A \rightarrow \bullet : \text{ "y" }\},$
9. $s_8 = \{S \rightarrow A \text{ "x" } A \bullet \text{ "y" } : \$\},$
10. $s_9 = \{S \rightarrow A \text{ "x" } A \text{ "y" } \bullet : \$\}.$

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-Parers überein. Daraus folgt, dass es auch bei den LALR-Zuständen keine Konflikte gibt, denn beim Übergang von kanonischen LR-Parern zu LALR-Parern haben wir lediglich Zustände mit gleichem Kern zusammengefasst, die Definition der Funktionen *goto()* und *action()* blieb unverändert.

14.6.2 *LALR-Sprache* \subsetneq *kanonische LR-Sprache*

Wir hatten LALR-Parser dadurch definiert, dass wir verschiedene Zustände eines kanonischen LR-Parers 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 14.13 zeigt eine solche Grammatik, die ich dem Drachenbuch entnommen habe.

S	\rightarrow	$\text{"v"} A \text{"y"}$
	$ $	$\text{"w"} B \text{"y"}$
	$ $	$\text{"v"} B \text{"z"}$
	$ $	$\text{"w"} A \text{"z"}$
A	\rightarrow	"x"
B	\rightarrow	"x"

Abbildung 14.13: 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. $s_0 = \text{closure}(\widehat{S} \rightarrow \bullet S : \$) = \{ \begin{array}{l} \widehat{S} \rightarrow \bullet S : \$, \\ S \rightarrow \bullet \text{"v"} A \text{"y"} : \$, \\ S \rightarrow \bullet \text{"v"} B \text{"z"} : \$, \\ S \rightarrow \bullet \text{"w"} A \text{"z"} : \$, \\ S \rightarrow \bullet \text{"w"} B \text{"y"} : \$ \end{array} \}$
2. $s_1 = \text{goto}(s_0, S) = \{ \widehat{S} \rightarrow S \bullet : \$ \}$
3. $s_2 = \text{goto}(s_0, \text{"v"}) = \{ \begin{array}{l} S \rightarrow \text{"v"} \bullet B \text{"z"} : \$, \\ S \rightarrow \text{"v"} \bullet A \text{"y"} : \$, \\ A \rightarrow \bullet \text{"x"} : \text{"y"}, \\ B \rightarrow \bullet \text{"x"} : \text{"z"} \end{array} \}$
4. $s_3 = \text{goto}(s_0, \text{"w"}) = \{ \begin{array}{l} S \rightarrow \text{"w"} \bullet A \text{"z"} : \$, \\ S \rightarrow \text{"w"} \bullet B \text{"y"} : \$, \\ A \rightarrow \bullet \text{"x"} : \text{"z"}, \\ B \rightarrow \bullet \text{"x"} : \text{"y"} \end{array} \}$
5. $s_4 = \text{goto}(s_2, \text{"x"}) = \{ A \rightarrow \text{"x"} \bullet : \text{"y"}, B \rightarrow \text{"x"} \bullet : \text{"z"} \}$
6. $s_5 = \text{goto}(s_3, \text{"x"}) = \{ A \rightarrow \text{"x"} \bullet : \text{"z"}, B \rightarrow \text{"x"} \bullet : \text{"y"} \}$
7. $s_6 = \text{goto}(s_2, A) = \{ S \rightarrow \text{"v"} A \bullet : \$ \}$
8. $s_7 = \text{goto}(s_6, \text{"y"}) = \{ S \rightarrow \text{"v"} A \text{"y"} \bullet : \$ \}$
9. $s_8 = \text{goto}(s_2, B) = \{ S \rightarrow \text{"v"} B \bullet : \$ \}$
10. $s_9 = \text{goto}(s_8, \text{"z"}) = \{ S \rightarrow \text{"v"} B \text{"z"} \bullet : \$ \}$
11. $s_{10} = \text{goto}(s_3, A) = \{ S \rightarrow \text{"w"} A \bullet : \$ \}$
12. $s_{11} = \text{goto}(s_{10}, \text{"z"}) = \{ S \rightarrow \text{"w"} A \text{"z"} \bullet : \$ \}$
13. $s_{12} = \text{goto}(s_3, B) = \{ S \rightarrow \text{"w"} B \bullet : \$ \}$
14. $s_{13} = \text{goto}(s_{12}, \text{"y"}) = \{ S \rightarrow \text{"w"} B \text{"y"} \bullet : \$ \}$

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

$$A \rightarrow \text{"x"} \quad \text{als auch mit} \quad B \rightarrow \text{"x"}$$

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

$$\text{core}(s_4) = \{A \rightarrow \text{"x"}\bullet, B \rightarrow \text{"x"}\bullet\} = \text{core}(s_5).$$

Bei der Berechnung der LALR-Zustände werden diese beiden Zustände zu einem Zustand $s_{\{4,5\}}$ zusammen gefaßt. Dieser neue Zustand hat die Form

$$s_{\{4,5\}} = \{A \rightarrow \text{"x"}\bullet : \{\text{"y"}, \text{"z"}\}, B \rightarrow \text{"x"}\bullet : \{\text{"y"}, \text{"z"}\}\}.$$

Hier gibt es offensichtlich einen Reduce-Reduce-Konflikt, denn einerseits haben wir

$$\text{action}(s_{\{4,5\}}, \text{"y"}) = \langle \text{reduce}, A \rightarrow \text{"x"} \rangle,$$

andererseits gilt aber auch

$$\text{action}(s_{\{4,5\}}, \text{"y"}) = \langle \text{reduce}, B \rightarrow \text{"x"} \rangle.$$

14.6.3 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. Das Werkzeug *Bison* ist ein Parser-Generator für C, C++ und neuerdings auch *Java*, während *JavaCup* einen Parser in der Sprache *Java* erzeugt. Aus historischen Gründen erzeugen diese beiden Werkzeuge allerdings nur LALR-Parser, so dass Sie de facto keine Wahl haben.

Kapitel 15

Der Parser-Generator *JavaCup*

LALR-Parser erlauben es, auch links-rekursive Grammatiken in natürlicher Weise zu parsen. Da die meisten von Ihnen in der Praxis 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>

Um einen mit *Cup* erzeugten Parser übersetzen zu können, benötigen Sie zusätzlich noch die folgende Datei:

<http://www2.cs.tum.edu/projects/cup/java-cup-11a-runtime.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 15.1 auf Seite 216 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 15.4 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, \dots, t_n ;`

Durch diese Deklaration werden die Symbole t_1, \dots, 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 15.1: 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, \dots, t_n ;`

Hierbei spezifiziert *type* den Typ, der den Terminalen t_1, \dots, 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 220 in Abbildung 15.4 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 ":" := body1 "{" action1 ":"
      "|" body2 "{" action2 ":"
      :
      "|" bodyn "{" actionn ":"
      ";
```

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-Terminals 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 15.1 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/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 Klasse 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 15.2 auf Seite 218 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 15.2: 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 15.3 auf Seite 219 zeigt eine solche Klasse. Wir erzeugen dort in Zeile 4 einen Parser, indem wir den Konstruktor der Klasse `parser` mit einem Scanner als Argument initialisieren. 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 assoziiert 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 15.3: Die Klasse Calculator.

15.0.4 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 15.4 auf Seite 220 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 15.2 auf Seite 218 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.

(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.

```

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\v\n\r]  { /* skip white space */ }
34
35 [^]          { throw new Error("Illegal character '" + yytext() +
36                                "' at line " + yyline +
37                                ", column " + yycolumn); }

```

Abbildung 15.4: Ein Scanner für arithmetische Ausdrücke

(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

```
new 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 15.3 gesehen, wie diese Klasse an den Parser angebunden wird.

15.1 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 15.5: CUP-Spezifikation eines Parsers für arithmetische Ausdrücke

Wir betrachten nun ein weiteres Beispiel. Abbildung 15.5 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 Zahl 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

$$R_1 := \left(\text{expr} \rightarrow \text{expr} \text{ “*” } \text{expr} \bullet \right)$$

und der markierten Regel

$$R_2 := \left(\text{expr} \rightarrow \text{expr} \bullet \text{ “+” } \text{expr} \right)$$

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:

```
l1r_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

$$\text{expr} \rightarrow \text{expr} \text{ “*” } \text{expr}$$

zu reduzieren, oder der Parser ist gerade dabei, die rechte Seite der Regel

$$\text{expr} \rightarrow \text{expr} \text{ “+” } \text{expr}$$

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.

15.2 Operator-Präzedenzen

Es ist mit CUP möglich, Shift-Reduce-Konflikte durch die Angabe von *Operator-Präzedenzen* aufzulösen. Abbildung 15.6 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 wird und die Potenzbildung bezeichnet, *rechts-assoziativ*, der Ausdruck

$$4^3^2 \quad \text{wird also als} \quad 4^{(3^2)} \quad \text{und nicht als} \quad (4^3)^2$$

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 15.6 gezeigten Grammatik ordnet CUP 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 15.6: 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
$E \rightarrow E \text{ “+” } E$	1
$E \rightarrow E \text{ “-” } E$	1
$E \rightarrow E \text{ “*” } E$	2
$E \rightarrow E \text{ “/” } E$	2
$E \rightarrow E \text{ “^” } E$	3
$E \rightarrow \text{ “(” } E \text{ “)” } E$	—
$E \rightarrow N$	—

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

$$r_1 = (A \rightarrow \alpha \bullet o \beta : L_1) \quad \text{und} \quad r_2 = (B \rightarrow \gamma \bullet : L_2) \quad \text{mit} \quad o \in L_2$$

vorkommen, so gibt es bei der Berechnung von

$$\text{action}(s, o)$$

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 fünf Fälle unterschieden:

- (a) $p(o) > p(B \rightarrow \gamma)$: In diesem Fall bindet der Operator o stärker. Daher wird das Token o in diesem Fall auf den Stack geschoben:

$$\text{action}(s, o) = \langle \text{shift}, \text{goto}(s, o) \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$1+2*3$$

mit den Grammatik-Regeln

$$E \rightarrow E \text{ “+” } E \mid E \text{ “*” } E \mid \text{NUMBER}$$

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:

$$\left\{ \begin{array}{l} E \rightarrow E \bullet \text{ “*” } E : \{ \$, \text{ “*” }, \text{ “+” } \}, \\ E \rightarrow E \bullet \text{ “+” } E : \{ \$, \text{ “*” }, \text{ “+” } \}, \\ E \rightarrow E \text{ “+” } E \bullet : \{ \$, \text{ “*” }, \text{ “+” } \} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “*” gelesen wird, so darf der bisher gelesene String “1+2” nicht mit der Regel $E \rightarrow E \text{ “+” } E$ reduziert werden, denn wir wollen die 2 ja zunächst mit 3 multiplizieren. Statt dessen muss das Zeichen “*” auf den Stack geschoben werden.

- (b) $p(o) < p(B \rightarrow \gamma)$: 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

$$\text{action}(s, o) = \langle \text{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$1*2+3$$

mit den Grammatik-Regeln

$$E \rightarrow E \text{ “+” } E \mid E \text{ “*” } E \mid \text{NUMBER}$$

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:

$$\left\{ \begin{array}{l} E \rightarrow E \bullet “*” E : \{ \$, “*”, “+” \}, \\ E \rightarrow E \bullet “+” E : \{ \$, “*”, “+” \}, \\ E \rightarrow E “*” E \bullet : \{ \$, “*”, “+” \} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “+” gelesen wird, so soll der bisher gelesene String “1*2” mit der Regel $E \rightarrow E “*” E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

- (c) $p(o) = p(B \rightarrow \gamma)$ und der Operator o ist links-assoziativ: Dann wird zunächst mit der Regel r_2 reduziert, wir haben also

$$action(s, o) = \langle \text{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$1*2*3$$

mit den Grammatik-Regeln

$$E \rightarrow E “+” E \mid E “*” E \mid \text{NUMBER}$$

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:

$$\left\{ \begin{array}{l} E \rightarrow E \bullet “*” E : \{ \$, “*”, “+” \}, \\ E \rightarrow E \bullet “+” E : \{ \$, “*”, “+” \}, \\ E \rightarrow E “*” E \bullet : \{ \$, “*”, “+” \} \end{array} \right\}.$$

Wenn in diesem Zustand als nächstes Zeichen ein “*” gelesen wird, so soll der bisher gelesene String “1*2” mit der Regel $E \rightarrow E “*” E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

- (d) $p(o) = p(B \rightarrow \gamma)$ und der Operator o ist rechts-assoziativ: In diesem Fall wird o auf den Stack geschoben:

$$action(s, o) = \langle \text{shift}, goto(s, o) \rangle.$$

Wenn wir diesen Fall verstehen wollen, reicht es aus, den String

$$2^3^4$$

mit den Grammatik-Regeln

$$E \rightarrow E \wedge E \mid \text{NUMBER}$$

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) $p(o) = p(B \rightarrow \gamma)$ und der Operator o hat keine Assoziativität: In diesem Fall liegt ein Syntax-Fehler vor:

$$action(s, o) = \text{error}.$$

Diesen Fall verstehen Sie, wenn Sie versuchen, einen String der Form

$$1 < 1 < 1$$

mit den Grammatik-Regeln

$$E \rightarrow E “<” E \mid E “+” E \mid \text{NUMBER}$$

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.

Bemerkung: Beachten Sie, dass auch in diesem Fall der Shift-Reduce-Konflikt aufgelöst wird, denn den Syntax-Fehler erhalten Sie erst beim Parsen, während die Erstellung des Parsers selber fehlerfrei (sprich: ohne verbleibende Konflikte) verläuft.

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-Konflikte aufgelöst worden sind. Wir betrachten exemplarisch zwei Zustände in dieser Datei.

1. Der Zustand Nummer 14 hat die in Abbildung 15.7 gezeigte Form. Hier gibt es unter anderem einen Shift-Reduce-Konflikt zwischen den beiden markierten Regeln

$$E \rightarrow E \bullet “+” E \quad \text{und} \quad E \rightarrow E \bullet “*” E \bullet,$$

denn die erste Regel verlangt nach einem Shift, während die zweite Regel eine Reduktion fordert. Da die Regel $E \rightarrow E \bullet “*” E$ die selbe Präzedenz wie der Operator “*” und dieser eine höhere Präzedenz als “+” hat, wird beispielsweise beim Lesen des Zeichens “+” mit der Regel $E \rightarrow E \bullet “*” E$ reduziert. Wird hingegen das Zeichen “^” gelesen, so wird dieses geshiftet, denn dieses Zeichen hat eine höhere Priorität als die Regel $E \rightarrow E \bullet “*” E$. Weiterhin gibt es einen Shift-Reduce-Konflikt zwischen den beiden markierten

```

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 15.7: Der Zustand Nummer 14.

Regeln

$$E \rightarrow E \bullet “*” E \quad \text{und} \quad E \rightarrow E \bullet “*” E \bullet.$$

Hier haben beide Regeln die gleiche Präzedenz. Daher entscheidet die Assoziativität. Da der Operator “*” links-assoziativ ist, wird mit der Regel $E \rightarrow E \bullet “*” E$ reduziert, falls das nächste Zeichen ein Multiplikations-Operator “*” ist.

2. Der Zustand Nummer 18 hat die in Abbildung 15.8 gezeigte Form. Zunächst gibt es hier einen Shift-Reduce-Konflikt zwischen den Regeln

$$E \rightarrow E \bullet “^” E \quad \text{und} \quad E \rightarrow E \bullet “^” E \bullet,$$

wenn das nächste Token der Operator “ \wedge ” ist. Da der Operator die selbe Präzedenz hat wie die Regel, entscheidet die Assoziativität. Nun ist der Operator “ \wedge ” 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 15.8: 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

$$E \rightarrow E \bullet \text{ “+” } E \quad \text{und} \quad E \rightarrow E \text{ “}\wedge\text{” } E \bullet,$$

der auftritt, wenn das nächste Token ein “+” ist. Da die Regel $E \rightarrow E \text{ “}\wedge\text{” } E$ die Präzedenz 3 hat, die größer ist als die Präzedenz 1 des Operators “+” wird dieser Konflikt dadurch aufgelöst, dass mit der Regel $E \rightarrow E \text{ “}\wedge\text{” } E$ reduziert wird.

15.2.1 Das *Dangling-Else*-Problem

```

1  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
2  terminal NUMBER, ID;
3
4  nonterminal stmt, stmtList, expr;
5
6  stmt ::= IF LPAREN expr RPAREN stmt
7         | IF LPAREN expr RPAREN stmt ELSE stmt
8         | WHILE LPAREN expr RPAREN stmt
9         | LBRACE stmtList RBRACE
10        | ID ASSIGN expr SEMI
11        ;
12
13  stmtList ::= stmtList stmt
14             | /* epsilon */
15             ;
16
17  expr ::= NUMBER
18        ;

```

Abbildung 15.9: Fragment einer Grammatik für die Sprache C

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 15.9 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 15.10 ausschnittsweise gezeigten Shift-Reduce-Konflikt.

```

1  Warning : *** Shift/Reduce conflict found in state #10
2    between stmt ::= IF LPAREN expr RPAREN stmt (*)
3    and      stmt ::= IF LPAREN expr RPAREN stmt (*) ELSE stmt
4    under symbol ELSE
5    Resolved in favor of shifting.

```

Abbildung 15.10: Ein Shift-Reduce-Konflikt.

Der Konflikt entsteht bei der Berechnung von $action(\text{state \#10}, \text{else})$ zwischen den beiden markierten Regeln

$$Statement \rightarrow \text{“if” “(” EXPR “)” } Statement \bullet \text{ und}$$

$$Statement \rightarrow \text{“if” “(” EXPR “)” } Statement \bullet \text{“else” } Statement.$$

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 15.9 gezeigte Grammatik mehrdeutig ist, denn ein *Statement* der Form

$$\text{if (a = b) if (c = d) s = t; else u = v;}$$

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 15.9 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-Präcedenzen 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` assoziiert wird. Um die normalerweise bei Konflikten 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 15.11 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. Der Nachteil ist allerdings, dass bei diesem Vorgehen die Grammatik stark aufgebläht wird. Vermutlich aus diesem Grunde findet sich in den späteren Auflagen 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 Präcedenz zuordnen. Abbildung 15.12 zeigt die Umsetzung dieser Idee.

```

1  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
2  terminal NUMBER, ID;
3
4  nonterminal stmtnt, matchedStmtnt, unmatchedStmtnt, stmtntList, expr;
5
6  stmtnt ::= matchedStmtnt
7           | unmatchedStmtnt
8           ;
9
10 matchedStmtnt ::= IF LPAREN expr RPAREN matchedStmtnt ELSE matchedStmtnt
11                 | WHILE LPAREN expr RPAREN matchedStmtnt
12                 | LBRACE stmtntList RBRACE
13                 | ID ASSIGN expr SEMI
14                 ;
15
16 unmatchedStmtnt ::= IF LPAREN expr RPAREN stmtnt
17                   | IF LPAREN expr RPAREN matchedStmtnt ELSE unmatchedStmtnt
18                   | WHILE LPAREN expr RPAREN unmatchedStmtnt
19                   ;
20
21 stmtntList ::= stmtntList stmtnt
22              | /* epsilon */
23              ;
24
25 expr ::= NUMBER
26        ;

```

Abbildung 15.11: Eine eindeutige Grammatik für C-Befehle.

- (a) Zunächst haben wir in den Zeilen 6 und 7 die Terminale **IF** und **ELSE** als nicht-assoziative 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 15.10 gezeigten Zustand ist.
- (b) In Zeile 9 haben wir der Regel

$$\text{stmtnt} ::= \text{IF LPAREN expr RPAREN stmtnt}$$

explizit mit Hilfe der nachgestellten Option

$$\%prec \text{ IF}$$

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 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 Programmiersprache **C**, bei der es 15 verschiedene Operator-Präzedenzen gibt, überfordern die meisten Benutzer.

```

1  terminal LPAREN, RPAREN, IF, ELSE, WHILE, LBRACE, RBRACE, ASSIGN, SEMI;
2  terminal NUMBER, ID;
3
4  nonterminal stmt, stmtList, expr;
5
6  precedence nonassoc IF;
7  precedence nonassoc ELSE;
8
9  stmt ::= IF LPAREN expr RPAREN stmt          %prec IF
10         | IF LPAREN expr RPAREN stmt ELSE stmt
11         | WHILE LPAREN expr RPAREN stmt
12         | LBRACE stmtList RBRACE
13         | ID ASSIGN expr SEMI
14         ;
15
16  stmtList ::= stmtList stmt
17              | /* epsilon */
18              ;
19
20  expr ::= NUMBER
21         ;

```

Abbildung 15.12: Auflösung des Shift-Reduce-Konflikts mit Hilfe von Operator-Präzedenzen.

15.3 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. *Mysteriöse 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.

15.3.1 Mehrdeutigkeits-Konflikte

Abbildung 15.13 zeigt die *Cup*-Spezifikation einer Grammatik, die offensichtlich mehrdeutig ist, denn das Token `NUMBER` kann sowohl als ein `a` als auch als ein `b` geparkt werden.

Cup liefert bei dieser Grammatik die folgende Fehlermeldung:

```

Warning : *** Reduce/Reduce conflict found in state #4
          between b ::= NUMBER (*)
          and      a ::= NUMBER (*)

```

```

1  terminal    NUMBER;
2  nonterminal s, a, b;
3
4  s ::= a
5     | b
6     ;
7  a ::= NUMBER
8     ;
9  b ::= NUMBER
10    ;

```

Abbildung 15.13: Eine mehrdeutige Grammatik.

```

under symbols: {EOF}
Resolved in favor of the second production.

```

Die einzige Möglichkeit diesen Konflikt aufzulösen besteht darin, die Grammatik so umzuschreiben, dass die Mehrdeutigkeit verschwindet.

15.3.2 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 15.14 zeigt eine Grammatik¹, die zwar eindeutig ist, die aber keine LR-Grammatik ist.

```

1  A : B 'u' 'v'
2     | C 'u' 'w'
3     ;
4  B : 'x'
5     ;
6  C : 'x'
7     ;

```

Abbildung 15.14: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

Berechnen wir die LR-Zustände dieser Grammatik, so finden wir unter anderem den folgenden Zustand:

$$\{B \rightarrow \text{"x"} \bullet : \text{"u"}, C \rightarrow \text{"x"} \bullet : \text{"u"}\}$$

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

$$A \rightarrow B \text{"u"} \text{"v"}$$

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

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.

$$A \rightarrow C \text{ "u" "w"}$$

und folglich ist das "x" als C zu lesen.

```

1  A : B 'v'
2    | C 'w'
3    ;
4  B : 'x' 'u'
5    ;
6  C : 'x' 'u'
7    ;

```

Abbildung 15.15: Eine zu der in Abbildung 15.14 äquivalente LR(1)-Grammatik.

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: Wenn der Kontext "u", der sowohl auf B als auch auf C folgt, mit in die Regeln für B und C aufgenommen wird, dann verschwindet der Konflikt. Abbildung 15.15 zeigt die modifizierte Grammatik.

15.3.3 Mystериöse 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 15.16 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 15.17 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 15.16: 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

$$s_7 = \text{goto}(s_2, \text{ID}).$$

2. Der Zustand s_8 entsteht, wenn das Token ID im Zustand s_0 gelesen wird, wir haben

$$s_8 = \text{goto}(s_0, \text{ID}).$$

Die Idee zur Auflösung des Konflikts ist, dass wir den Zustand s_2 so ändern, dass die Kerne von $\text{goto}(s_2, \text{ID})$ und $\text{goto}(s_0, \text{ID})$ 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 15.18 ab, indem wir eine zusätzliche Grammatik-Regel

$$\text{return_spec} \rightarrow \text{ID BOGUS}$$

eingeführen. 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 ändern sich nun die LR-Zustände. Abbildung 15.19 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

```

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 15.17: LR-Zustände der in Abbildung 15.16 gezeigten Grammatik.

ist, dass diese Zustände in einem LALR-Parser-Generator nicht mehr zusammengefasst werden. Dadurch gibt es dann auch keinen Konflikt mehr.

```

1  def : param_spec return_spec ', '
2      ;
3  param_spec
4      : type
5      | name_list ':' type
6      ;
7  return_spec
8      : type
9      | name ':' type
10     | ID BOGUS          // this never happens
11     ;
12  type: ID
13      ;
14  name: ID
15      ;
16  name_list
17      : name
18      | name ', ' name_list
19      ;

```

Abbildung 15.18: Auflösung des mysteriösen Reduce-Reduce-Konflikts.

```

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 -> <*> ID BOGUS: [','],
13        return_spec -> <*> name ': ' type: [','],
14        return_spec -> <*> type: [','],
15        type -> <*> ID: [',']
16      }
17 s7 = { name -> ID <*>: [',', ':'],
18        type -> ID <*>: [ID]
19      }
20 s8 = { name -> ID <*>: [':'],
21        return_spec -> ID <*> BOGUS: [','],
22        type -> ID <*>: [',']
23      }

```

Abbildung 15.19: Einige Zustände der in Abbildung 15.18 gezeigten Grammatik.

Aufgabe 31: Nachstehend sehen Sie Regeln für eine Grammatik, die Listen von Zahlen beschreibt:

$$\begin{aligned} List &\rightarrow \text{NUMBER } \text{“,”} List \\ &\quad | \text{NUMBER} \end{aligned}$$

Diese Grammatik ist rechts-rekursiv. Alternativ ist auch eine links-rekursive Grammatik möglich:

$$\begin{aligned} List &\rightarrow List \text{“,”} \text{NUMBER} \\ &\quad | \text{NUMBER} \end{aligned}$$

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

$$x_1, x_2, \dots, x_{n-1}, x_n$$

in den beiden Fällen geparkt wird. Geben Sie insbesondere an, wie der Stack des Shift-Reduce-Parsers aussieht, wenn die Teilliste

$$x_1, x_2, \dots, x_{k-1}, x_k \quad \text{mit } k \leq n$$

gelesen worden ist.

Kapitel 16

Types and Type Checking

There are two fundamentally different approaches to typing. Either the user is required to declare the types of variables or instead the program has to check the types of objects at runtime. In the first case, the programming language is called *statically typed*, in the second case it is called *dynamically typed*.

1. Statically typed languages like *Java* or *C* require that the user declares the types of functions and variables. The compiler is then able to check that these variables will indeed have the declared type at runtime. This approach has the following advantages:
 - (a) A number of runtime errors can be excluded. For example, if a variable is declared as a `float` in *C*, we are guaranteed that the program will not try to store a string in this variable.
 - (b) The program does not need to check the type of variables at runtime and can thus be more efficient.
 - (c) Adding type information serves as a form of documentation that can be checked automatically. This enhances the readability of typed programs.
 - (d) Typed programs are easier to maintain as many violations of interfaces between different parts of a typed program will actually manifest itself as type errors. Therefore, the compiler is able to detect these violations.
2. Dynamically typed languages like *Perl*, *Python*, or *JavaScript* do not require the programmer to declare any types. Rather, the types are checked at runtime. For example, if a program in a dynamically typed language contains an expression of the form

$$x + y,$$

the compiler generates code that checks the type of x and y at runtime. Then, if x and y are discovered to be integers, the program performs an integer addition. However, if x and y happen to be strings, these strings are concatenated. Dynamic typing has the following advantages:

- (a) Programs written in dynamically typed languages are typically shorter than the corresponding programs in statically typed languages.
- (b) If the types used in an algorithm are very complex, then using an untyped language is sometimes the only way to code an algorithm in the intended way. The reason is that in some cases the expressiveness of current type system is not sufficient to be able to code certain algorithms conveniently.

The disadvantages of statically typed languages are the advantages of dynamically typed languages and vice versa. Therefore, dynamically typed languages are often used for prototyping, while statically typed languages are used for production systems.

In practice, most statically typed languages offer some escape mechanism to cover the cases where the type system gets too unwieldy. For example, in *C* the programmer can declare a variable x to have type `void*` and then cast this pointer to any other pointer type. In *Java*, a programmer can declare a variable x as having type `Object`. This variable can then hold any object and when this variable is used the user needs to cast it to the appropriate type.

In this chapter we are going to show how the compiler is able to type check a statically typed program. To this end, we first introduce a very simple statically typed programming language and then we develop a type checker for this language.

16.1 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 Ausdrücke 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 16.1: Ein TTL-Beispiel-Programm

Abbildung 16.1 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 `List(X)`:

$$nil \in list(X)$$

- (b) Induktions-Schritt: Falls `a` ein Element der Menge `X` und `l` ein Element der Menge `list(X)` ist, dann ist der Term `cons(a, l)` ebenfalls ein Element der Menge `list(X)`:

$$a \in X \wedge l \in list(X) \rightarrow cons(a, l) \in list(X).$$

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

`concat(nil, nil)` und `concat(cons(x, nil), cons(y, cons(z, nil)))`

angegeben und es wird behauptet, dass diese den Typ `list(int)` haben. Die Aufgabe der Typ-Überprüfung besteht darin, diese Aussage zu verifizieren.

```

1  grammar ttl;
2
3  program    : typeDef* signature* typedTerm*
4              ;
5
6  typeDef    : 'type' FCT ':= ' type ('+' type)* ';'
7              | 'type' FCT '(' PARAM (',' PARAM)* ')' ':= ' type ('+' type)* ';'
8              ;
9
10 type       : FCT '(' type (',' type)* ')'
11             | FCT
12             | PARAM
13             ;
14
15 signature  : 'signature' FCT ':' type ('* ' type)* '->' type ';'
16             | 'signature' FCT ':' type ';'
17             ;
18
19 term       : FCT '(' term (',' term)* ')'
20             | FCT
21             ;
22
23 typedTerm  : term ':' type ';'
24             ;
25
26 PARAM      : ('A'..'Z') ('a'..'z' | '_' | '0'..'9')*;
27 FCT        : ('a'..'z') ('a'..'z' | '_' | '0'..'9')*;

```

Abbildung 16.2: Eine EBNF-Grammatik für die getypte Beispielsprache

Die genaue Syntax der Sprache TTL wird durch die in Abbildung 16.2 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 Typ-Konstruktoren, Variablen und Funktionsnamen 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.

16.2 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.
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

$$\text{map}(K, V), \quad \text{double}, \quad \text{oder} \quad \text{list}(\text{int}).$$

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

$$\mathbb{K} = \{\text{map}, \text{list}, \text{int}, \text{double}\} \quad \text{und} \quad \mathbb{P} = \{K, V\}$$

setzen. Zusätzlich muss noch eine Funktion

$$\text{arity} : \mathbb{K} \rightarrow \mathbb{N}$$

gegeben sein, die für jeden Typ-Konstruktor festlegt, wieviel Argumente er erwartet. In dem obigen Beispiel hätten wir

$$\text{arity}(\text{map}) = 2, \quad \text{arity}(\text{list}) = 1, \quad \text{arity}(\text{int}) = 0 \quad \text{und} \quad \text{arity}(\text{double}) = 0.$$

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 $\text{arity}(c) = 0$, so ist auch c ein Typ:

$$c \in \mathbb{K} \wedge \text{arity}(c) = 0 \Rightarrow c \in \mathcal{T}.$$

3. Ist f ein n -stelliger Typ-Konstruktor und sind τ_1, \dots, τ_n Typen, so ist auch $f(\tau_1, \dots, \tau_n)$ ein Typ:

$$f \in \mathbb{K} \wedge \text{arity}(f) = n \wedge n > 0 \wedge \tau_1 \in \mathcal{T} \wedge \cdots \wedge \tau_n \in \mathcal{T} \Rightarrow f(\tau_1, \dots, \tau_n) \in \mathcal{T}.$$

Examples: If the type constructors map , int , list , and double have the arities given above and if, furthermore, K and V are type parameters, then the following are types:

1. int ,
2. double ,
3. $\text{list}(\text{double})$,
4. $\text{list}(V)$,

5. $\text{map}(K, \text{list}(\text{double}))$. ◇

Definition 44 (Parameter Substitution) A parameter substitution is a finite set of pairs of the form

$$\sigma = \{\langle X_1, \tau_1 \rangle, \dots, \langle X_n, \tau_n \rangle\}$$

such that

1. X_i is a type parameter for all $i \in \{1, \dots, n\}$,
2. τ_i is a type for all $i \in \{1, \dots, n\}$,
3. the type parameters occurring in σ are pairwise distinct, that is we have

$$i \neq j \rightarrow X_i \neq X_j \quad \text{for all } i, j \in \{1, \dots, n\}.$$

If $\sigma = \{\langle X_1, \tau_1 \rangle, \dots, \langle X_n, \tau_n \rangle\}$ is a parameter substitution, then σ is written as

$$\sigma = [X_1 \mapsto \tau_1, \dots, X_n \mapsto \tau_n].$$

In this case, the domain of σ is defined as

$$\text{dom}(\sigma) = \{X_1, \dots, X_n\}.$$

The set of all parameter substitutions is denoted as SUBST. In the following, parameter substitutions are just called substitutions. □

Substitutions can be *applied* to types. If τ is a type and $\vartheta = [X_1 \mapsto \tau_1, \dots, X_n \mapsto \tau_n]$ is a substitution, then $\tau\vartheta$ is the type that we get if we replace all occurrences of X_i in τ by τ_i . The formal definition follows.

Definition 45 (Application of a Substitution)

If τ is a type and $\vartheta = [X_1 \mapsto \tau_1, \dots, X_n \mapsto \tau_n]$ is a substitution, then the application of ϑ to τ (written $\tau\vartheta$) is defined by induction on τ :

1. If τ is a type parameter, there are two cases:

(a) $\tau = X_i$ for some $i \in \{1, \dots, n\}$. Then

$$X_i\vartheta := \tau_i.$$

(b) $\tau = Y$ where Y is a type parameter such that $Y \notin \{X_1, \dots, X_n\}$. Then

$$Y\vartheta := Y.$$

2. Otherwise τ must have the form $\tau = f(\sigma_1, \dots, \sigma_m)$. Then $\tau\vartheta$ is defined as

$$f(\sigma_1, \dots, \sigma_m)\vartheta := f(\sigma_1\vartheta, \dots, \sigma_m\vartheta). \quad \square$$

Examples: Define the substitution ϑ as

$$\vartheta := [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(\text{int})].$$

Then we have the following:

1. $X_3\vartheta = X_3$,
2. $\text{list}(X_2)\vartheta = \text{list}(\text{list}(\text{int}))$,
3. $\text{map}(X_1, \text{set}(X_2))\vartheta = \text{map}(\text{double}, \text{set}(\text{list}(\text{int})))$.

Next, we show how substitutions can be composed.

Definition 46 (Composition of Substitutions) If

$$\vartheta = [X_1 \mapsto \sigma_1, \dots, X_m \mapsto \sigma_m] \quad \text{and} \quad \eta = [Y_1 \mapsto \tau_1, \dots, Y_n \mapsto \tau_n]$$

are substitutions such that $\text{dom}(\vartheta) \cap \text{dom}(\eta) = \{\}$, then we define the composition $\vartheta\eta$ of ϑ and η as

$$\vartheta\eta := [X_1 \mapsto \sigma_1\eta, \dots, X_m \mapsto \sigma_m\eta, Y_1 \mapsto \tau_1, \dots, Y_n \mapsto \tau_n]. \quad \square$$

Example: Define

$$\begin{aligned}\vartheta &:= [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(X_3)] \quad \text{and} \\ \eta &:= [X_3 \mapsto \text{map}(\text{int}, \text{double}), X_4 \mapsto \text{char}].\end{aligned}$$

Then we have

$$\vartheta\eta = [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(\text{map}(\text{int}, \text{double})), X_3 \mapsto \text{map}(\text{int}, \text{double}), X_4 \mapsto \text{char}]. \quad \diamond$$

Exercise 32: How would we have to change the definition of the composition of ϑ and η if we drop the condition $\text{dom}(\vartheta) \cap \text{dom}(\eta) = \{\}$?

The following theorem is a consequence of the previous definition.

Theorem 47 (Associativity of Composition)

If τ is a type and ϑ and η are substitutions such that $\text{dom}(\vartheta) \cap \text{dom}(\eta) = \{\}$ holds, then we have

$$(\tau\vartheta)\eta = \tau(\vartheta\eta).$$

□

This theorem is proved by an easy induction on τ .

Next, we formalize the notion of a term t being of type τ .

Definition 48 ($t : \tau$)

For a term t and a type τ the relation $t : \tau$ (read: t has type τ) is defined by induction on t .

1. If c is a nullary function symbol with signature $c : \tau$ and if ϑ is any parameter substitution, then

$$c : \tau\vartheta.$$

2. Assume that

(a) f is an n -ary function symbol such that $n > 0$.

(b) f has the signature

$$f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho.$$

(c) ϑ is a substitution such that we have $t_i : \sigma_i\vartheta$.

Then $f(t_1, \dots, t_n) : \varrho\vartheta$.

Examples: The following examples assume that the types and signatures are given as in Figure 16.1 on page 238.

1. We have $\text{nil} : \text{list}(\text{int})$, because nil has the signature

$$\text{nil} : \text{list}(X).$$

Therefore, defining

$$\vartheta = [X \mapsto \text{int}]$$

gives $\text{list}(X)\vartheta = \text{list}(\text{int})$ showing the claim.

2. Next, we show

$$\text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int}).$$

The signature of concat is

$$\text{concat} : \text{list}(T) \times \text{list}(T) \rightarrow \text{list}(T).$$

Define

$$\vartheta = [T \mapsto \text{int}].$$

We have already seen that

$$\text{nil} : \text{list}(\text{int})$$

holds. Because of $\text{list}(T)\vartheta = \text{list}(\text{int})$ this shows the claim. \diamond

16.3 A Type Checking Algorithm

Assume a term $t = f(t_1, \dots, t_n)$ and a type τ is given and we want to check whether $t : \tau$ holds. Furthermore, assume that the signature of f is given as

$$f : \sigma_1 \times \dots \times \sigma_n \rightarrow \varrho.$$

According to the definition of $t : \tau$ the term t has type τ iff there is a parameter substitution ϑ such that $\varrho\vartheta = \tau$ and $t_i : \sigma_i\vartheta$. Therefore

$$f(t_1, \dots, t_n) : \tau \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\varrho\vartheta = \tau \wedge \forall i \in \{1, \dots, n\} : t_i : \sigma_i\vartheta)$$

The problem of type checking is to compute the substitution ϑ or to show that ϑ can not exist.

Examples: We discuss the previous examples (that is $\text{nil} : \text{list}(\text{int})$ and $\text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int})$) again using the formula

$$f(t_1, \dots, t_n) : \tau \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\varrho\vartheta = \tau \wedge \forall i \in \{1, \dots, n\} : t_i : \sigma_i\vartheta).$$

1. Lets prove $\text{nil} : \text{list}(\text{int})$ again. We have the signature

$$\text{nil} : \text{list}(X).$$

Therefore, we have

$$\text{nil} : \text{list}(\text{int}) \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\text{list}(X)\vartheta = \text{list}(\text{int})).$$

Obviously, we can define

$$\vartheta = [X \mapsto \text{int}].$$

Then, the equation $\text{list}(X)\vartheta = \text{list}(\text{int})$ is true and we conclude that $\text{nil} : \text{list}(\text{int})$ holds.

2. Next, we prove $\text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int})$. The signature of concat is given as

$$\text{concat} : \text{list}(T) \times \text{list}(T) \rightarrow \text{list}(T).$$

Therefore, our claim is equivalent to

$$\begin{aligned} & \text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int}) \\ \Leftrightarrow & \exists \vartheta \in \text{SUBST} : (\text{list}(T)\vartheta = \text{list}(\text{int}) \wedge \text{nil} : \text{list}(T)\vartheta \wedge \text{nil} : \text{list}(T)\vartheta). \end{aligned}$$

The equation $\text{list}(T)\vartheta = \text{list}(\text{int})$ is solved by the substitution

$$\vartheta = [T \mapsto \text{int}]$$

and therefore the correctness of

$$\text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int})$$

is reduced to the correctness of

$$\text{nil} : \text{list}(T)\vartheta,$$

As $\text{list}(T)\vartheta = \text{list}(\text{int})$ and $\text{nil} : \text{list}(\text{int})$ holds, the proof is complete. \square

The previous examples show that the correctness of an expression $t : \tau$ can be reduced to the solution of a set of *type equations* of the form $\varrho\vartheta = \tau$. We formalize this observation by defining a function

$$\text{typeEqs} : \text{Term} \times \text{Type} \rightarrow \text{set}(\text{Equation}).$$

For a term t and a type τ

$$\text{typeEqs}(t, \tau)$$

yields a set of type equations. These type equations will be solvable if and only if $t : \tau$ is correct. The function typeEqs is defined as follows:

$$(f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho) \implies \text{typeEqs}(f(t_1, \dots, t_n), \tau) := \{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i).$$

This definition has to be read as follows: If the function f has the signature $f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho$, then $f(t_1, \dots, t_n) : \tau$ is true if and only if the set of *type equations*

$$\{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i)$$

has a solution ϑ . We formalize the notion of a *type equation* next.

Definition 49 (Type Equation) *If σ and τ are types, then the expression*

$$\sigma \doteq \tau$$

is called a type equation. A system of type equations is a set of type equations. A substitution ϑ solves a type equation $\sigma \doteq \tau$ if and only if $\sigma\vartheta = \tau\vartheta$. A substitution ϑ solves a system of type equations E iff it solves every type equation in E . \square

In order to implement type checking we still need an algorithm to solve systems of type equations. The crucial observation is that type equations can be solved by unification. We do not need the general form of unification because if we have a type equation of the form $\varrho = \tau$ then only the left hand side ϱ will contain type parameters. We use the algorithm given by Martelli and Montanari [MM82]. This algorithm works with pairs of the form

$$\langle E, \vartheta \rangle$$

where E is a set of type equations and ϑ is a substitution. We start with the pair

$$\langle E, [] \rangle.$$

Here, the substitution is empty, while E is the set of type equations we want to solve. These pairs will be gradually transformed until we arrive at a pair of the form

$$\langle \{\}, \vartheta \rangle$$

such that ϑ is the solution of the system of type equations E . We use the following rules to rewrite the pairs.

1. If X is a type parameter, then

$$\langle E \cup \{X \doteq \tau\}, \vartheta \rangle \rightsquigarrow \langle E[X \mapsto \tau], \vartheta[X \mapsto \tau] \rangle.$$

This works as follows: If E contains a type equation of the form $X \doteq \tau$, then we can remove this type equation from E if we incorporate this equation in the substitution ϑ by transforming ϑ into the new substitution $\vartheta[X \mapsto \tau]$. Of course, we also have to apply the substitution $[X \mapsto \tau]$ to the remaining type equations in E .

2. If f is an n -ary type constructor we have the rule

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \langle E \cup \{\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n\}, \vartheta \rangle.$$

Therefore, a type equation of the form $f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)$ is replaced by the n type equations $\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n$.

A special case of this occurs if $n = 0$. It reads

$$\langle E \cup \{c \doteq c\}, \vartheta \rangle \rightsquigarrow \langle E, \vartheta \rangle.$$

Here c is a nullary type constructor. This rule just says that trivial type equations can be dropped.

3. A system of type equations of the form $E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}$ has no solution if f and g are different. Therefore, we have

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega \quad \text{if } f \neq g.$$

Here Ω denotes unsolvability. If the solution of $\text{typeEqs}(t, \tau)$ yields Ω , then the set of type equations E can not be solved and $t : \tau$ is wrong. \diamond

Example: We demonstrate the algorithm by solving the type equation

$$\text{map}(X_2, X_3) \doteq \text{map}(\text{char}, \text{list}(\text{int})).$$

We proceed as follows:

$$\begin{aligned} & \langle \{\text{map}(X_2, X_3) \doteq \text{map}(\text{char}, \text{list}(\text{int}))\}, [] \rangle \\ \rightsquigarrow & \langle \{X_2 \doteq \text{char}, X_3 \doteq \text{list}(\text{int})\}, [] \rangle \\ \rightsquigarrow & \langle \{X_3 \doteq \text{list}(\text{int})\}, [X_2 \mapsto \text{char}] \rangle \\ \rightsquigarrow & \langle \{\}, [X_2 \mapsto \text{char}, X_3 \mapsto \text{list}(\text{int})] \rangle \end{aligned}$$

In this case, the algorithm is successful and the resulting substitution

$$[X_2 \mapsto \text{char}, X_3 \mapsto \text{list}(\text{int})]$$

is a solution of the type equation given above.

16.4 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 16.2 auf Seite 239 präsentiert. Die Abbildungen 16.3, 16.4 und 16.5 auf den folgenden Seiten zeigen die *JavaCup*-Spezifikation eines Parsers für diese Grammatik. Der zugehörige Scanner ist in Abbildungen 16.7 auf Seite 250 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-Konstrukturen. 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 16.6 gezeigten Spezifikation erzeugt.

Die Klasse `MartelliMontanari` enthält die in Abbildung 16.8 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:

$$\langle E \cup \{X \doteq \tau\}, \vartheta \rangle \rightsquigarrow \langle E[X \mapsto \tau], \vartheta[X \mapsto \tau] \rangle.$$

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-Konstrukturen auf beiden Seiten verschieden sind, es wird also der Fall

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega.$$

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

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \langle E \cup \{\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n\}, \vartheta \rangle.$$

umformen.

Als letztes diskutieren wir die Berechnung der Typ-Gleichungen, die in der Methode `typeEqs()` der Klasse `Term` implementiert ist. Ein Term $f(s_1, \dots, s_n)$ hat genau dann den Typ τ , falls die Funktion f eine Signatur

$$f : \sigma_1 \times \dots \times \sigma_n \rightarrow \varrho$$

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:

$$f(s_1, \dots, s_n) : \tau \Leftrightarrow \exists \vartheta \in \text{SUBST} : \varrho\vartheta = \tau \wedge s_1 : \sigma_1\vartheta \wedge \dots \wedge s_n : \sigma_n\vartheta.$$

1. Um die obige Definition von $t : \tau$ umzusetzen, suchen wir für einen Term der Form $f(s_1, \dots, s_n)$ zunächst in Zeile 2 nach der Signatur des Funktions-Zeichens f . Diese Signaturen sind in 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  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 16.3: *JavaCup*-Spezifikation der TTL-Grammatik, 1. Teil

```

1  typeSum      ::= typeSum:l PLUS type:t {: l.add(t); RESULT = 1; :}
2              | type:t
3              {:
4                  List<Type> l = new LinkedList<Type>();
5                  l.add(t);
6                  RESULT = 1;
7              :}
8              ;
9  signature    ::= SIGNATURE FUNCTION:f COLON argTypes:a
10                  ARROW type:t SEMICOLON
11                  {: RESULT = new Signature(f, a, t); :}
12              | SIGNATURE FUNCTION:f COLON type:t SEMICOLON
13                  {: List a = new LinkedList<Type>();
14                  RESULT = new Signature(f, a, t);
15                  :}
16              ;
17  signatures   ::= signatures:l signature:s {: l.add(s); RESULT = 1; :}
18              | signature:s
19              {: List<Signature> l = new LinkedList();
20              l.add(s);
21              RESULT = 1;
22              :}
23              ;
24  argTypes     ::= argTypes:l TIMES type:t {: l.add(t); RESULT = 1; :}
25              | type:t
26              {: List<Type> l = new LinkedList();
27              l.add(t); RESULT = 1;
28              :}
29              ;
30  varList      ::= varList:l COMMA PARAMETER:v
31                  {: l.add(new Parameter(v)); RESULT = 1; :}
32              | PARAMETER:v {: List<Parameter> l = new LinkedList();
33                  l.add(new Parameter(v));
34                  RESULT = 1;
35                  :}
36              ;
37  term         ::= FUNCTION:f LEFT_PAR termList:l RIGHT_PAR
38                  {: RESULT = new Term(f, l); :}
39              | FUNCTION:f {: List<Term> l = new LinkedList<Term>();
40                  RESULT = new Term(f, l);
41                  :}
42              ;
43  termList     ::= termList:l COMMA term:t
44                  {: l.add(t); RESULT = 1; :}
45              | term:t {: List<Term> l = new LinkedList();
46                  l.add(t); RESULT = 1;
47                  :}
48              ;

```

Abbildung 16.4: *JavaCup*-Spezifikation der TTL-Grammatik, 2. Teil

```

1  typedTerm    ::= term:t COLON type:s SEMICOLON
2                {: RESULT = new TypedTerm(t, s); :}
3                ;
4
5  typedTerms   ::= typedTerms:l typedTerm:t
6                {: l.add(t); RESULT = l; :}
7                | typedTerm:t
8                {: List<TypedTerm> l = new LinkedList<TypedTerm>();
9                  l.add(t);
10                 RESULT = l;
11                 :}
12                ;

```

Abbildung 16.5: *JavaCup*-Spezifikation der TTL-Grammatik, 3. Teil

```

1  Program = Program(List<TypeDef>   typeDefs,
2                    List<Signature> signatures,
3                    List<TypedTerm> typedTerms);
4
5  TypeDef = SimpleTypeDef(String name, List<Type> typeSum)
6            + ParamTypeDef(String   name,
7                           List<String> parameters,
8                           List<Type>  typeSum);
9
10 Type = Parameter(String name)
11       + CompositeType(String name, List<Type> argTypes);
12
13 substitute: Type * Parameter * Type -> Type;
14
15 Signature = Signature(String name, List<Type> argList, Type result);
16
17 Term = Term(String function, List<Term> termList);
18
19 typeEqs: Term * Type * Map<String, Signature> -> List<Equation>;
20
21 TypedTerm = TypedTerm(Term term, Type type);
22
23 Substitution = Substitution(List<Parameter> variables, List<Type> types);
24
25 Equation = Equation(Type lhs, Type rhs);
26
27 substitute: Equation * Parameter * Term -> Equation;
28
29 MartelliMontanari =
30   MartelliMontanari(List<Equation> equations, Substitution theta);

```

Abbildung 16.6: Definition der benötigten *Java*-Klassen mit Hilfe von EP.

2. Ansonsten speichern wir in Zeile 8 die Argument-Typen $\sigma_1, \dots, \sigma_n$ in der Variablen `argTypes`. Die Signatur von f muss natürlich genau so viele Argument-Typen haben, wie das Funktions-Zeichen f Argumente

```

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_0-9]* { 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 16.7: JFlex-Spezifikation des Scanners

hat. Dies wird in Zeile 9 überprüft.

3. Falls bis hierhin keine Probleme aufgetreten sind, erzeugen wir nun die Typ-Gleichungen nach der Formel

$$(f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho) \implies \text{typeEqs}(f(t_1, \dots, t_n), \tau) := \{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i).$$

Dazu erstellen wir in Zeile 17 zunächst die Typ-Gleichungen

$$\varrho \doteq \tau$$

```

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 16.8: Die Methode `solve()` aus der Klasse `MartelliMontanari`.

und berechnen dann in der Schleife in den Zeilen 19 – 22 rekursiv die Typ-Gleichungen, die sich aus den Forderungen

$$s_i : \sigma_i \quad \text{für } i = 1, \dots, n$$

ergeben. Als Ergebnis erhalten wir eine Menge von Typ-Gleichungen, die genau dann lösbar sind, wenn der Term $f(s_1, \dots, s_n)$ den Typ τ hat.

Als letztes diskutieren wir die Klasse `Program`. Abbildung 16.10 zeigt den Konstruktor dieser Klasse. Dieser Konstruktor bekommt als Argumente

- eine Liste von Typ-Definitionen `typeDefs`,
- eine Liste von Signaturen `signatures` und
- 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

```

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 16.9: Berechnung der Typ-Gleichungen

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. Falls die untersuchte Typ-Definition **td** eine einfache Typ-Definition der Form

$$\tau := f_1(\sigma_1^{(1)}, \dots, \sigma_{n_1}^{(1)}) + \dots + f_k(\sigma_1^{(k)}, \dots, \sigma_{n_k}^{(k)})$$

ist, so wird der Typ τ von den Funktionen f_1, \dots, f_k erzeugt. Die i -te Funktion f_i hat die Signatur

$$f_i : \sigma_1^{(i)} \times \dots \times \sigma_{n_i}^{(i)} \rightarrow \tau.$$

```

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 16.10: Der Konstruktor der Klasse **Program**.

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 16.11 gezeigt. Diese Methode iteriert in der Schleife, die in Zeile 2 beginnt, über alle getypten Terme $t : \tau$, die dem Konstruktor der Klasse **Program** über-

geben wurden. Dazu wird in Zeile 6 zu jedem Term t und Typ τ die Menge der Typ-Gleichungen $\text{typeEqs}(t, \tau)$ berechnet. Dazu wird der Methode `typeEqs()`, die die Funktion $\text{typeEqs}()$ 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 16.11: Die Methode `typeCheck()` in der Klasse `Program`.

16.5 Inklusions-Polymorphismus

Interpretieren wir Typen als Mengen von Werten, so stellen wir fest, dass es bei Typen eine Inklusions-Hierarchiy gibt. In der Sprache *Java* hat jeder Wert vom Typ `String` auch gleichzeitig den Typ `Object`, es gilt also

`String` \subseteq `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 16.12.

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 Kombiatiion 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 16.12: 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 Autoren 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 17

Assembler

A compiler translates programs written in a high level language like C or Java into some low level representation. This low level representation can be either machine code or some form of assembler code. The compiler that we are going to develop in the next chapter generates a particular form of assembler code known as JVM assembler. This assembler code is related to the byte code generated by a Java compiler. The very readable book of Tanenbaum on computer architecture [Tan05] describes a processor that is capable of executing the corresponding byte code.

17.1 Einführung in Jvm-Assembler

Der Bytecode, in den Java übersetzt wird, ist mit dem Ziel der Portabilität entwickelt worden: Die abstrakte Java *Virtual Machine* (kurz JVM), die diesen Bytecode abarbeitet, lässt sich auf den meisten Prozessoren einigermaßen effizient simulieren. Damit dies möglich ist, darf die JVM keine Annahmen machen, die für bestimmte Prozessoren falsch sind. Daraus erklärt sich, dass die JVM dem Benutzer keine Register zur Verfügung stellt. Daher holen die einzelnen Instruktionen ihre Argumente immer von einem Stack und legen auch das Ergebnis auf diesem Stack ab. Der wird dabei durch ein einziges Register **SP**, den sogenannten Stack-Pointer realisiert. Dieses Register zeigt immer auf das zuletzt im Stack abgelegte Wort.

Wollen wir zum Beispiel ein Byte am Bildschirm ausgeben, so müssen wir dieses Byte vorher auf dem Stack ablegen. Dazu stellt die JVM den Befehl **bipush** zur Verfügung. Dieser Befehl hat die Syntax

bipush *Byte*.

Dabei steht *Byte* für eine ganze Zahl, die sich mit 8 Bits darstellen lässt und die gemäß dem ASCII-Code als Zeichen interpretiert wird. In dem JVM-Assembler kann die Zahl *Byte* sowohl dezimal als auch hexadezimal angegeben werden, wobei Hexadezimalzahlen an dem Präfix “0x” erkannt werden.

Für die Ausgabe gibt es im JVM-Assembler die Instruktion **out**, die ein Byte als ASCII-Zeichen am Bildschirm ausgibt. Das Zeichen wird anschließend vom Stack entfernt indem der Stack-Pointer dekrementiert wird. Der Befehl zur Ausgabe hat die Syntax

out

Der Befehl erhält kein Argument, denn das Byte, das am Bildschirm ausgegeben wird, wird ja vom Stack genommen.

Damit können wir nun unser erstes Programm in JVM-Assembler erstellen. Einem altem Brauch folgend erstellen wir zunächst ein Assembler-Programm, das den Text “Hello World!” gefolgt von einem Zeilenumbruch am Bildschirm ausgibt. Abbildung 17.1 auf Seite 257 zeigt dieses Programm. Die erste Zeile dieses Programms enthält die Direktive “**.main**”, mit der wir den Beginn des Hauptprogramms markieren. Dann folgen abwechselnd immer ein **bipush**-Befehl, der ein Byte auf dem Stack ablegt, und ein **out**-Befehl, der dieses Byte vom Stack herunternimmt und ausgibt. Der Befehl “**halt**” in Zeile 28 hält schließlich den Prozessor an. Das Programm wird mit der Direktive “**.end-main**” abgeschlossen. Diese Direktive signalisiert das Ende der Methode **main()**. Komplexere Programme enthalten neben der Methode **main()** noch weitere Methoden, die im Programm-Text auf die Methode **main()** folgen würden.

```

1  .main
2  bipush  0x48  // ascii(0x48) = 'H'
3  out
4  bipush  0x65  // ascii(0x65) = 'e'
5  out
6  bipush  0x6c  // ascii(0x6c) = 'l'
7  out
8  bipush  0x6c  // ascii(0x6c) = 'l'
9  out
10 bipush  0x6f  // ascii(0x6f) = 'o'
11 out
12 bipush   32  // ascii(32)   = ' '
13 out
14 bipush  0x57  // ascii(0x57) = 'W'
15 out
16 bipush  0x6f  // ascii(0x6f) = 'o'
17 out
18 bipush  0x72  // ascii(0x72) = 'r'
19 out
20 bipush  0x6c  // ascii(0x6c) = 'l'
21 out
22 bipush  0x64  // ascii(0x64) = 'd'
23 out
24 bipush  0x21  // ascii(0x21) = '!'
25 out
26 bipush  0x0a  // ascii(0x0a) = '\n'
27 out
28 halt
29 .end-main

```

Abbildung 17.1: “Hello World” in JVM-Assembler.

17.2 Die Instruktionen der JVM

Wir stellen nun einige der Instruktionen vor, die von der JVM zur Verfügung gestellt werden. Dazu erläutern wir zunächst die Aufteilung des Speichers. Der Arbeits-Speicher der JVM ist in drei Bereiche unterteilt:

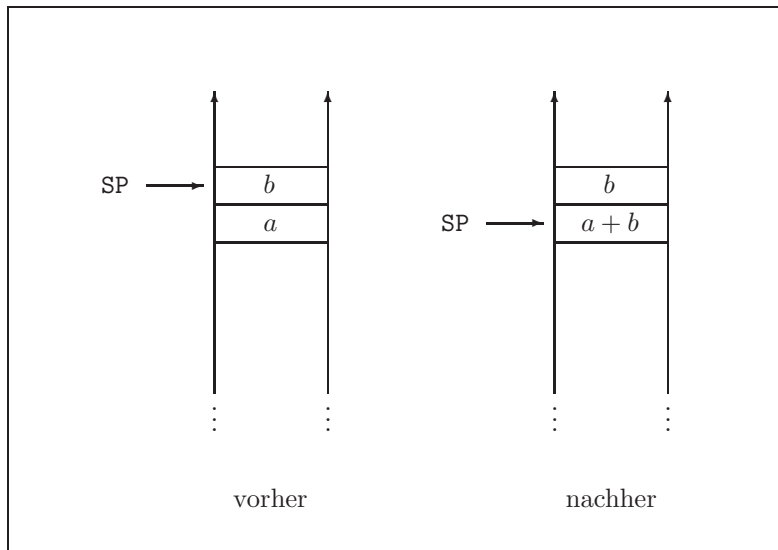
1. Der Programm-Speicher enthält das Programm als Folge von einzelnen Bytes.
2. Der Stack enthält die Operanden der auszuführenden Befehle. Außerdem liegen hier die lokalen Variablen und die Argumente von Prozeduren, die aber jetzt nicht Prozeduren heißen, sondern *Methoden*.

Das Register SP zeigt auf das obere Ende des Stacks. Weiter gibt es noch das Register LV (*local variables*), das auf das erste Argument der als Letzte aufgerufenen Methode zeigt. Wir werden die Funktion des Registers LV näher erläutern, wenn wir den Aufruf von Methoden diskutieren.

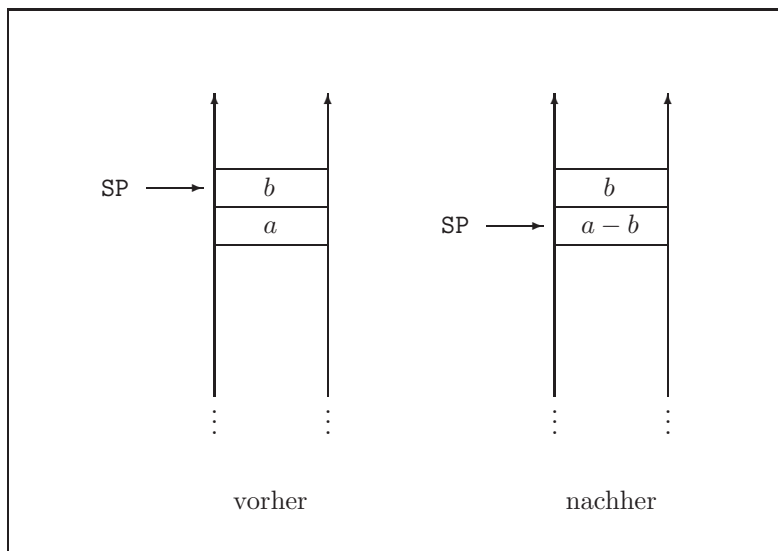
3. Der *Konstanten-Pool* enthält einerseits die Definition von Konstanten und andererseits die Adressen der Methoden im Programm-Speicher.

Wir stellen im Folgenden die Instruktionen der JVM vor, die wir später benutzen werden. Da wir nur Programme übersetzen werden, bei denen alle Variablen den Typ `int` haben, reicht es aus, einen kleinen Teil der insgesamt von der JVM unterstützten Assembler-Befehle zu diskutieren. Wir beginnen mit den arithmetischen und logischen Befehlen.

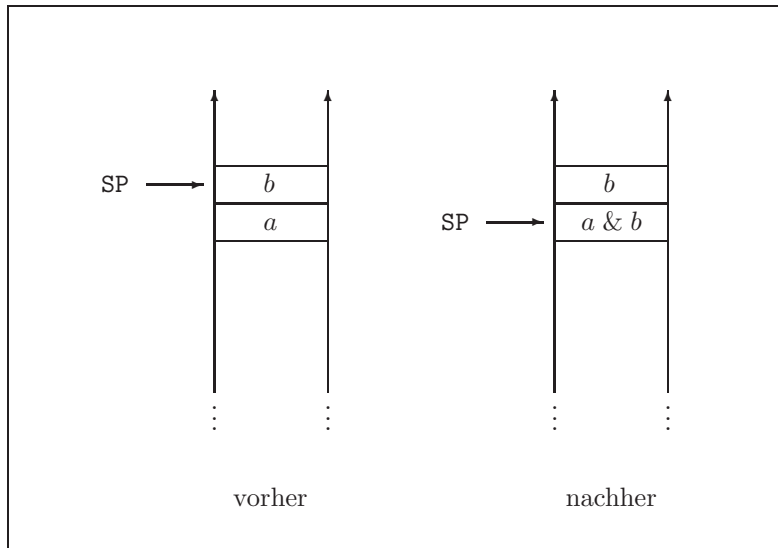
Der Befehl “iadd” addiert die beiden Werte, die zuoberst auf dem Stack liegen. Konzeptuell werden diese Werte dann vom Stack entfernt und das Ergebnis wird auf den Stack gelegt. Abbildung 17.2 zeigt die Wirkung des Befehls `iadd` graphisch. Beachten Sie, dass der Wert b , der vorher zuoberst auf dem Stack liegt, auch nach der Ausführung des Befehls `iadd` auf dem Stack liegen bleibt. Allerdings zeigt der Stack-Pointer jetzt auf den Wert, der unter b liegt, so dass der Wert b *de facto* nicht mehr vom Benutzer verwendet werden kann, denn es gibt keinen Befehl, der den Stack-Pointer erhöht ohne dass dabei ein neuer Wert auf dem Stack abgelegt wird.

Abbildung 17.2: Wirkung des Befehls `iadd`.

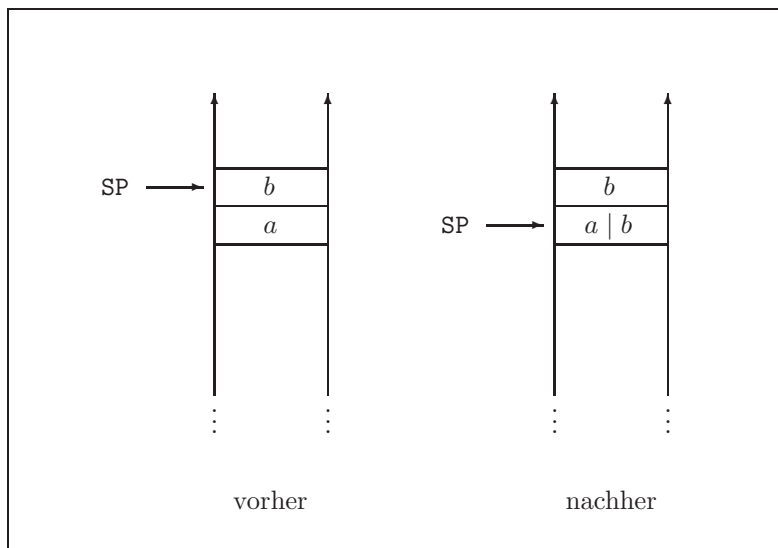
Der Befehl “isub” subtrahiert den Wert, der oben auf dem Stack liegt, von dem darunter liegenden Wert. Konzeptuell werden diese Werte dann vom Stack entfernt und das Ergebnis wird auf den Stack gelegt. Abbildung 17.3 zeigt die Wirkung des Befehls `isub` graphisch.

Abbildung 17.3: Wirkung des Befehls `isub`.

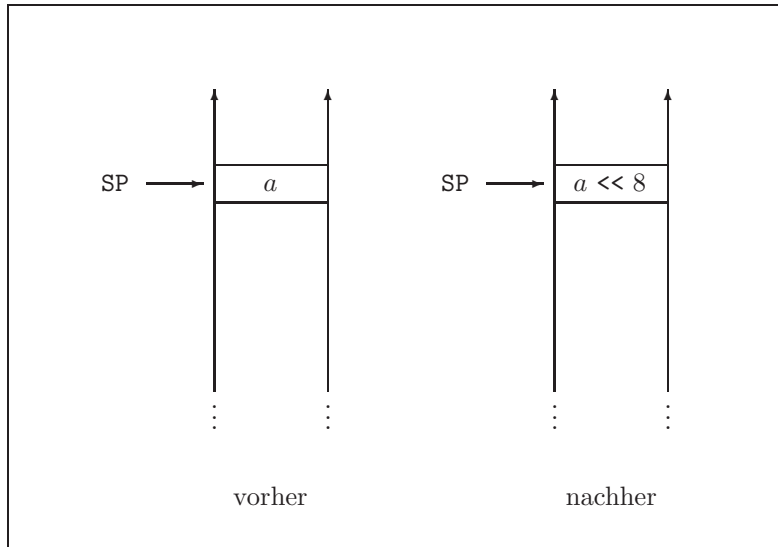
Der Befehl **“iand”** bildet eine bitweise Und-Verknüpfung der beiden Werte, die oben auf dem Stack liegen. Konzeptuell werden diese Werte dann vom Stack entfernt und das Ergebnis wird auf den Stack gelegt. Abbildung 17.4 zeigt die Wirkung des Befehls **iand** graphisch.

Abbildung 17.4: Wirkung des Befehls **iand**.

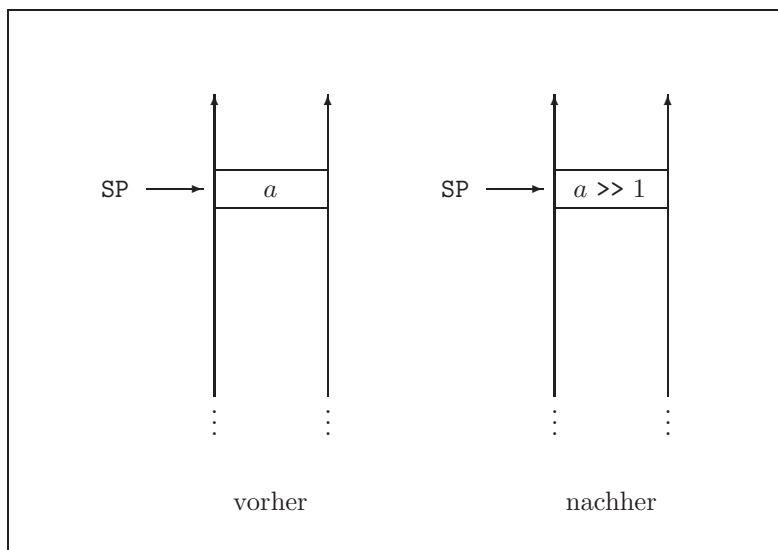
Der Befehl **“ior”** bildet eine bitweise Oder-Verknüpfung der beiden Werte, die oben auf dem Stack liegen. Konzeptuell werden diese Werte dann vom Stack entfernt und das Ergebnis wird auf den Stack gelegt. Abbildung 17.5 zeigt die Wirkung des Befehls **ior** graphisch.

Abbildung 17.5: Wirkung des Befehls **ior**.

Der Befehl **“sll8”** schiebt die Bits des obersten Worts auf dem Stack um 8 Bits nach links. Dabei gehen die obersten 8 Bits verloren und an die Stelle der untersten 8 Bits werden Nullen geschoben. Der String **“sll8”** ist als Abkürzung für **“shift left logical by 8”** zu lesen. Abbildung 17.6 zeigt die Wirkung des Befehls **sll8** auf den Stack.

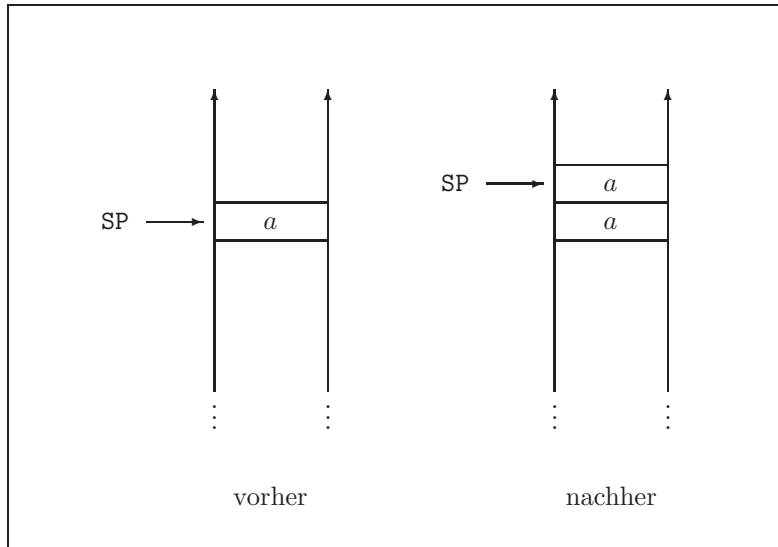
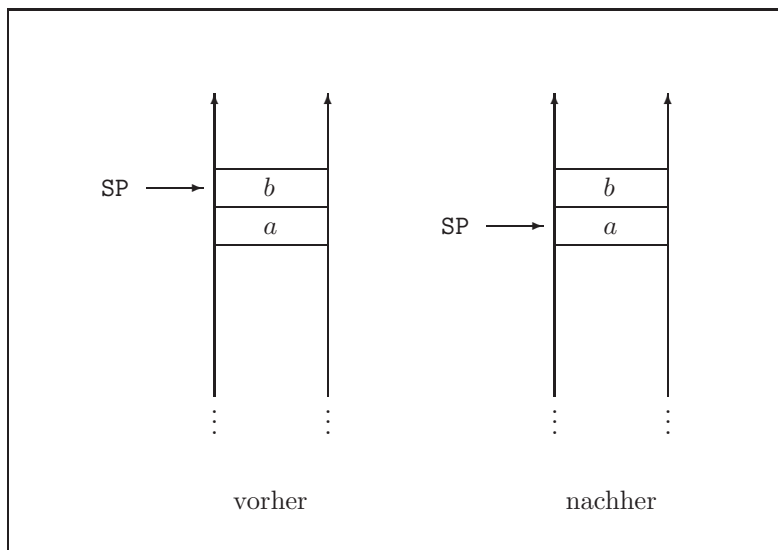
Abbildung 17.6: Wirkung des Befehls `sll8`.

Der Befehl “`sra1`” schiebt die Bits des obersten Wortes auf dem Stack um 1 Bit nach rechts. Dabei bleibt das Vorzeichen-Bit des Wortes unverändert. Deshalb sprechen wir auch von einem arithmetischen Shift. Der String “`sra1`” ist die Abkürzung von “`shift right arithmetic by 1`”. Abbildung 17.7 zeigt die Wirkung des Befehls `sra1` auf den Stack.

Abbildung 17.7: Wirkung des Befehls `sra1`.

Der Befehl “`dup`” legt den Wert, der oben auf dem Stack liegt, ein weiteres Mal auf den Stack, so dass der Wert jetzt zweimal dort liegt. Einen solchen Befehl benötigen wir dann, wenn wir ein Argument mehrfach in einer Berechnung verwenden wollen, denn die arithmetischen und logischen Befehle, die wir bisher diskutiert haben, überschreiben ja das erste Argument. Abbildung 17.8 zeigt die Wirkung des Befehls `dup` graphisch.

Der Befehl “`pop`” entfernt den obersten Wert auf dem Stack, indem der Stack-Pointer dekrementiert wird. Abbildung 17.9 zeigt die Wirkung des Befehls `pop` graphisch.

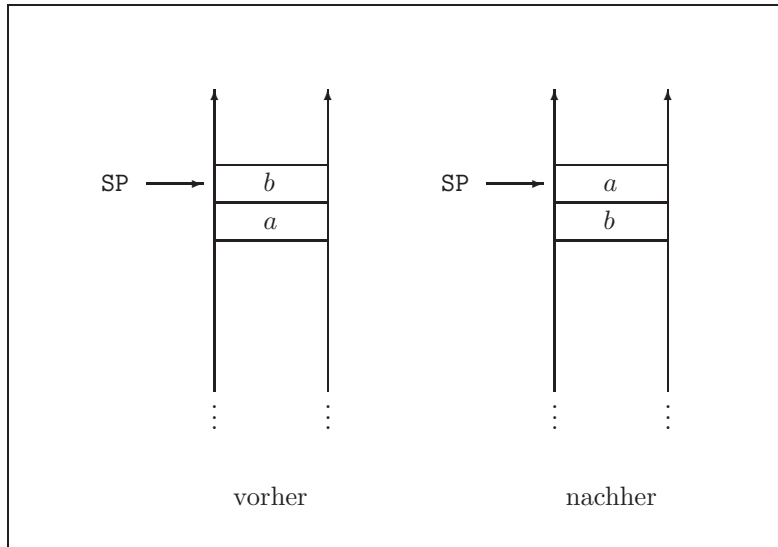
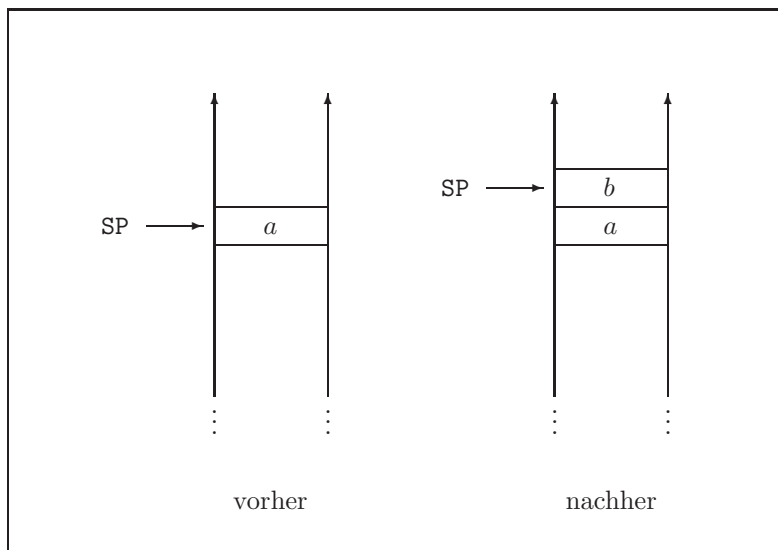
Abbildung 17.8: Wirkung des Befehls `dup`.Abbildung 17.9: Wirkung des Befehls `pop`.

Der Befehl `“swap”` vertauscht die beiden Werte, die oben auf dem Stack liegen. Abbildung 17.10 zeigt die Wirkung des Befehls `swap` graphisch.

Der Befehl `“nop”` ist der sogenannte *Beamtenbefehl*, den er macht nichts. Der Name `“nop”` steht daher für `“no operation”`.

Der Befehl `“bipush b”` legt das als Argument angegebene Byte oben auf den Stack. Abbildung 17.11 zeigt die Wirkung des Befehls `bipush` graphisch.

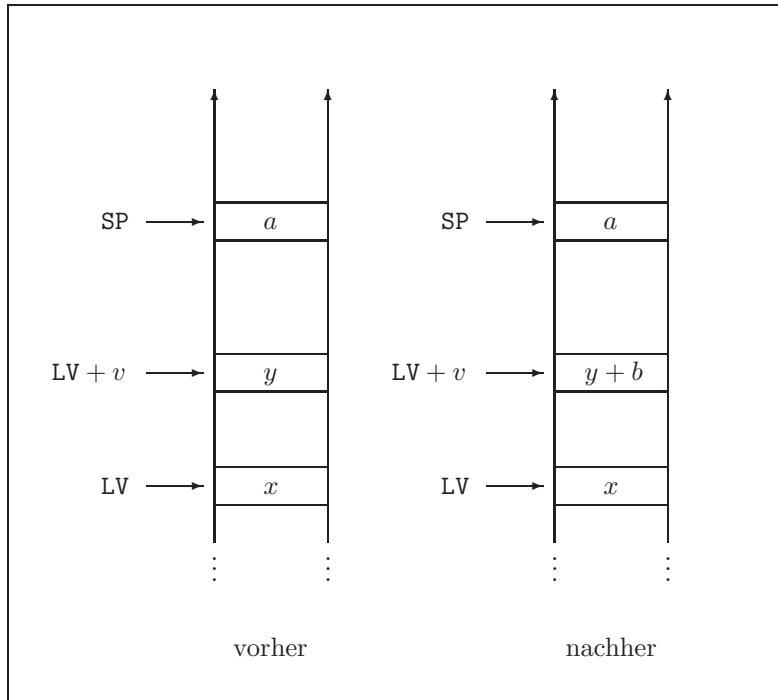
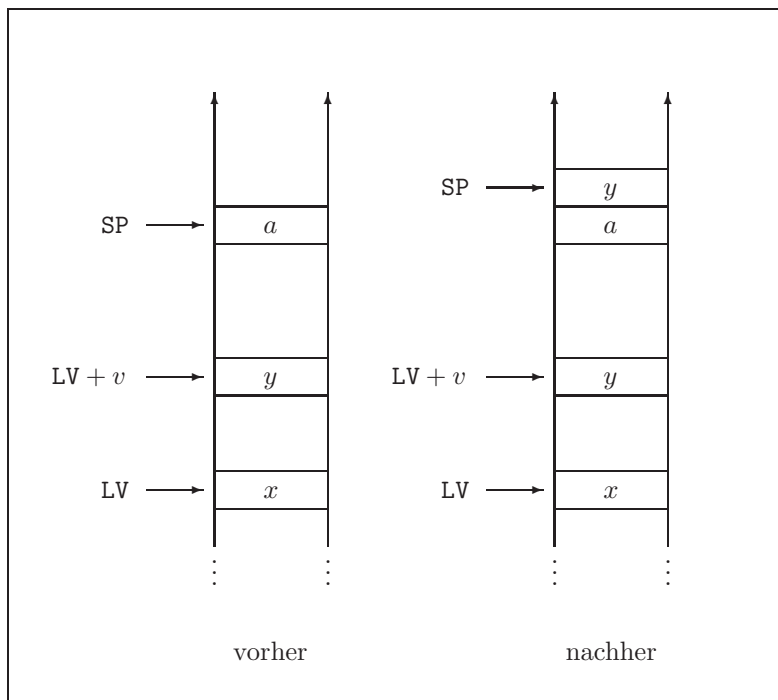
Der Befehl `“iinc v b”` inkrementiert die lokale Variable `v` um das Byte `b`. Der Stack wird dabei nicht verändert. Im Bytecode ist das Argument `v` ein Byte, das als Index in die Tabelle der lokalen Variablen interpretiert wird. Das Register `LV` zeigt auf die erste lokale Variable und folglich wird die Variable an der Stelle

Abbildung 17.10: Wirkung des Befehls `swap`.Abbildung 17.11: Wirkung des Befehls `bipush b`.

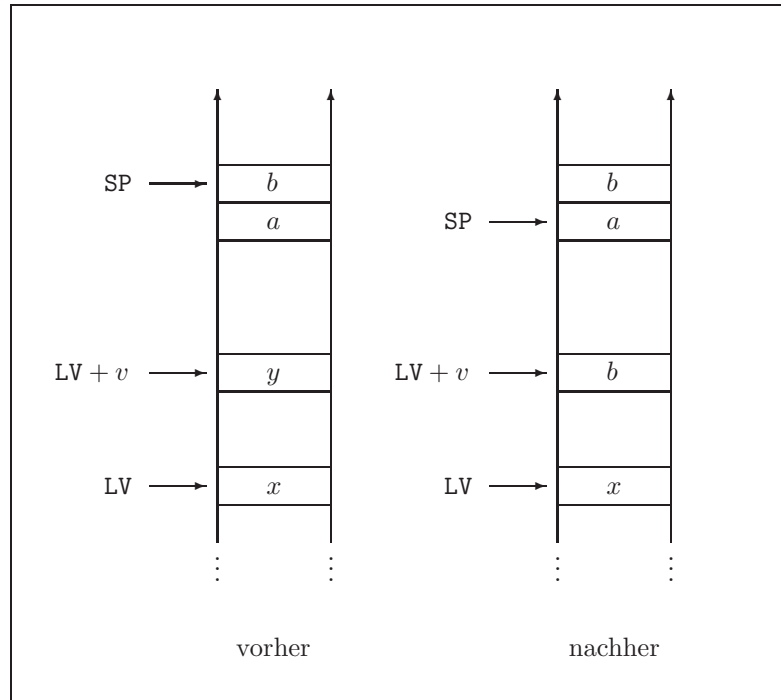
$LV + v$ um das Byte b inkrementiert. Dabei ist b vorzeichenbehaftet und stellt er eine Zahl aus dem Intervall $\{-128, \dots, 127\}$ dar. Abbildung 17.12 zeigt die Wirkung des Befehls `iinc` graphisch.

Der Befehl “`iload v`” liest die lokale Variable v und legt diese oben auf den Stack. Die lokale Variable v muss dazu am Anfang der Methode, in der sie verwendet wird, deklariert werden. Im Bytecode wird v dann durch ein Byte kodiert, das als Index in die Tabelle der lokalen Variablen interpretiert wird. Abbildung 17.13 zeigt die Wirkung des Befehls `iload` graphisch.

Der Befehl “`istore v`” liest den oben auf dem Stack liegenden Wert und weist diesen Wert der lokalen Variablen v zu. Der Stack-Peinter wird dabei dekrementiert. Im Bytecode ist das Argument v ein Byte, das als Index in die Tabelle der lokalen Variablen interpretiert wird. Abbildung 17.14 zeigt die Wirkung des Befehls `istore` graphisch.

Abbildung 17.12: Wirkung des Befehls `iinc v b`.Abbildung 17.13: Wirkung des Befehls `iload v`.

Der Befehl `ldc_w i` liest die Konstante i aus dem Konstanten-Pool und legt diese oben auf den Stack. Dabei steht i für den Namen einer Konstanten, die zu Beginn des Assembler-Programms deklariert sein muss. Im Bytecode wird die Konstante i durch zwei Bytes kodiert, die als Index in die Tabelle der Konstanten interpretiert werden. Der Name `ldc_w` steht für `load constant wide`. Der Name hat den Zusatz `wide`

Abbildung 17.14: Wirkung des Befehls `istore v`.

weil das Argument i aus zwei Bytes besteht. Abbildung 17.15 zeigt die Wirkung des Befehls `ldc_w` graphisch. CPP bezeichnet hier den Zeiger auf den Anfang des Konstanten-Pools. Der Konstanten-Pool ist eine Tabelle, in der Konstanten abgelegt werden. Das Argument i gibt den Index an, unter dem die Konstante in der Tabelle abgelegt ist.

Der Befehl “out” gibt das unterste Byte des oben auf dem Stack liegenden Wortes aus und dekrementiert den Stack-Pointer. Abbildung 17.16 zeigt die Wirkung des Befehls `out` graphisch.

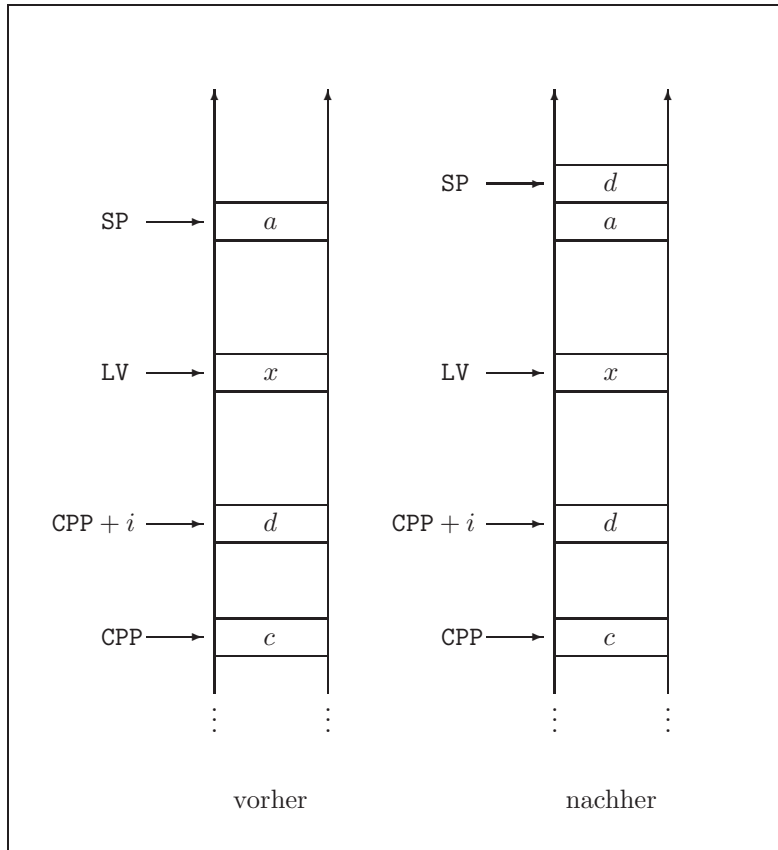
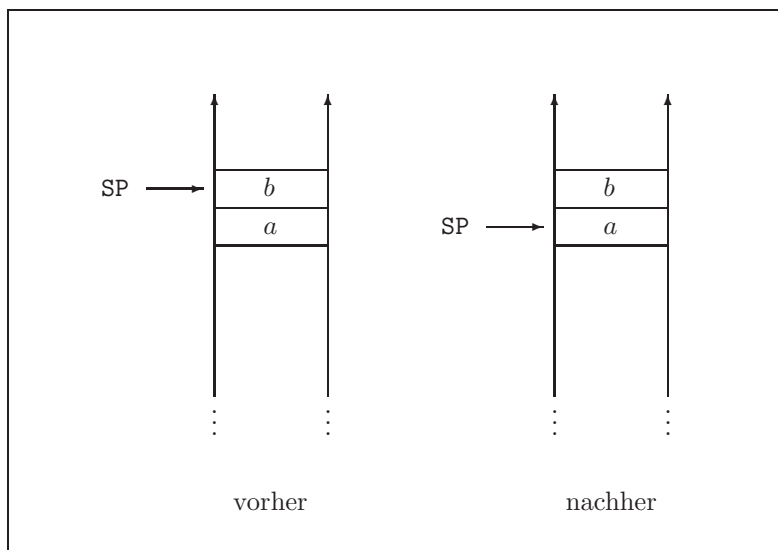
Der Befehl “in” liest ein Byte von außen ein und legt dieses Byte oben auf dem Stack ab. Abbildung 17.17 zeigt die Wirkung des Befehls `in` graphisch.

Der Befehl “goto o” springt zu dem Label o . Das Label o bezeichnet dabei ein Sprungziel, das in dem Assembler-Programm deklariert ist.

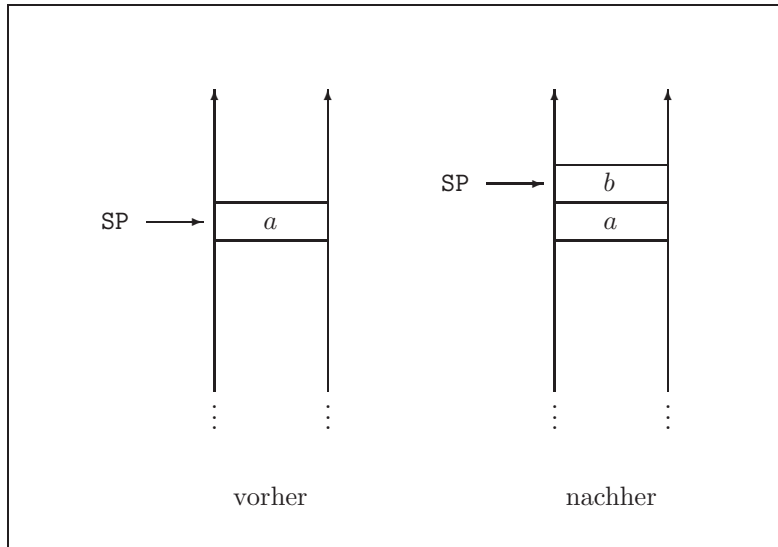
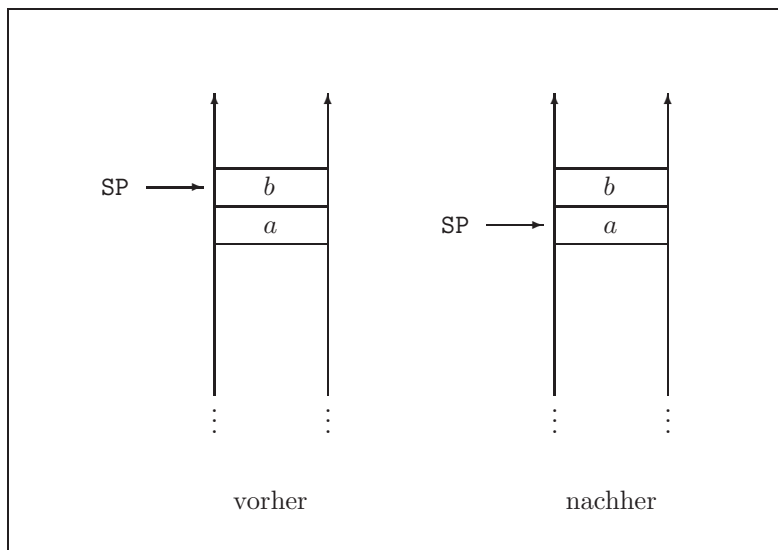
Der Befehl “ifeq o” prüft, ob das oben auf dem Stack liegende Wort den Wert 0 hat. Wenn dieses Wort den Wert 0 hat, dann verzweigt das Programm zu dem Label o , andernfalls geht es mit dem nächsten Bytecode-Befehl weiter. Die Abkürzung “ifeq” steht für “if equal”. Die Wirkung des Befehls auf den Stack wird in Abbildung 17.18 gezeigt.

Der Befehl “iflt o” prüft, ob das oben auf dem Stack liegende Wort negativ ist. Falls der Wert negativ ist, dann verzweigt das Programm zu dem Label o . Die Abkürzung “iflt” steht für “if less than”. Die Wirkung des Befehls auf den Stack wird in Abbildung 17.18 gezeigt.

Der Befehl “if_icmpeq o” prüft, ob die beiden oben auf dem Stack liegenden Worte gleich sind. In diesem Fall verzweigt das Programm zu dem Label o , andernfalls geht es mit dem nächsten Befehl weiter. Die Abkürzung “if_icmpeq” steht für “if integer compare equal”. Die Wirkung des Befehls auf den Stack wird in Abbildung 17.19 gezeigt.

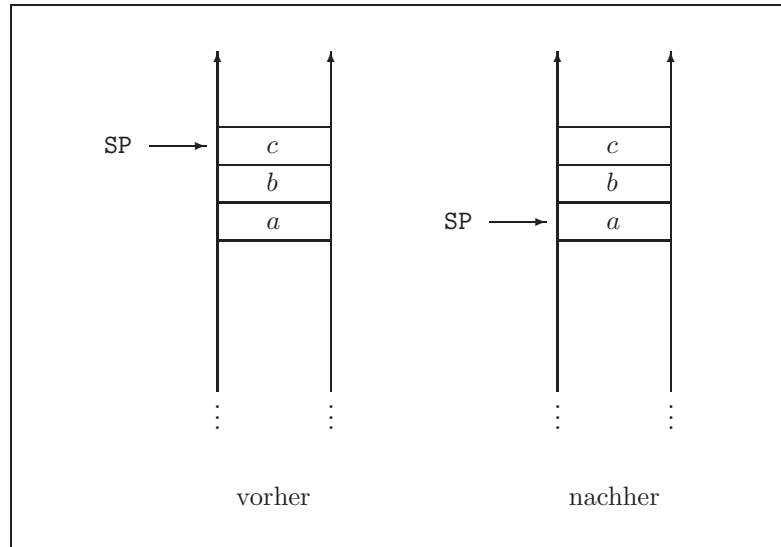
Abbildung 17.15: Wirkung des Befehls `ldc_w i`.Abbildung 17.16: Wirkung des Befehls `out`.

Der Befehl `“invokevirtual m”` dient dazu, die Methode *m* aufzurufen. Wenn der Befehl `invokevirtual` aufgerufen wird, ist die Situation so, wie im linken Teil der Abbildung 17.20 auf Seite 267 gezeigt. Auf dem Stack liegen die Argumente, mit denen die Methode aufgerufen werden soll. In der Abbildung sind diese Argumente grau schattiert. Das erste Argument, das in der Abbildung mit `OBJREF` bezeichnet wird, hat eine besondere

Abbildung 17.17: Wirkung des Befehls `in`.Abbildung 17.18: Wirkung der Befehle `ifeq` und `iflt` auf den Stack.

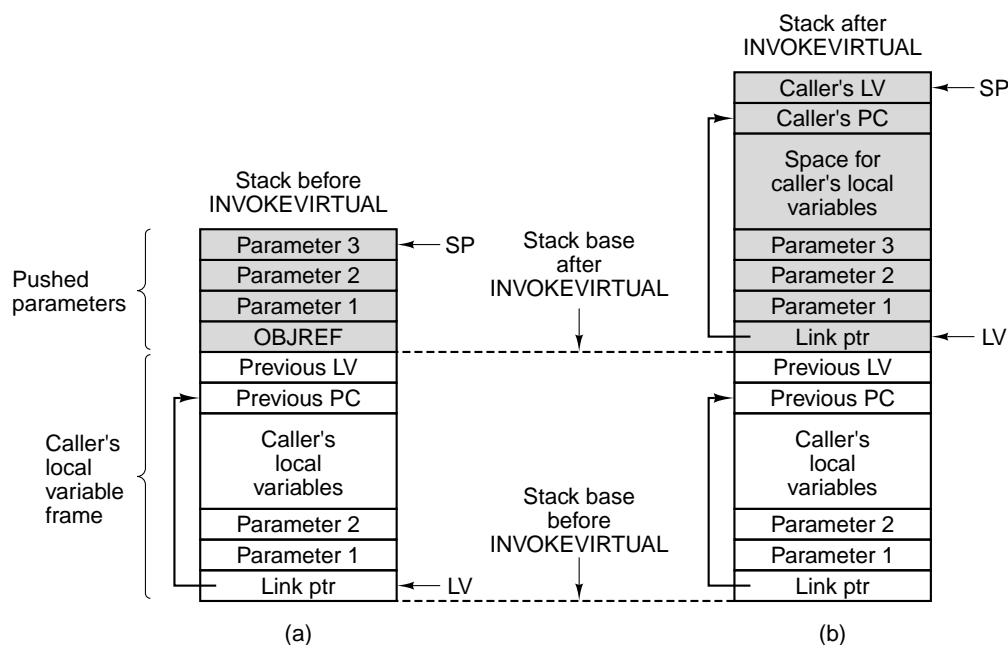
Funktion. In der JVM gibt dieses Argument die Adresse des Objektes an, dessen Methode aufgerufen werden soll. Bei einer statischen Methode hat dieses Argument nur noch die Funktion eines Platzhalters auf dem Stack, der Wert, der hier abgelegt ist, spielt dann keine Rolle. Bevor der Bytecode der aufgerufenen Methode abgearbeitet werden kann, sind einige Dinge zu erledigen:

1. Wir müssen auf dem Stack Platz schaffen für die lokalen Variablen der aufzurufenden Prozedur. Dazu müssen wir wissen, wie viele lokale Variablen es gibt und wie viele Parameter die Prozedur hat. Diese Information ist in dem Bytecode der aufgerufenen Methode hinterlegt: Die ersten zwei Bytes enthalten die Anzahl der Argumente, die nächsten beiden Bytes enthalten die Anzahl der lokalen Variablen.
2. Der Wert des Programm-Zählers muss gesichert werden, damit wir bei der Beendigung der Methode an die Stelle zurückspringen können, von der aus die Methode aufgerufen worden ist.
3. Der aktuelle Wert des Registers LV, das den Beginn der lokalen Variablen der aufrufenden Prozedur angibt,

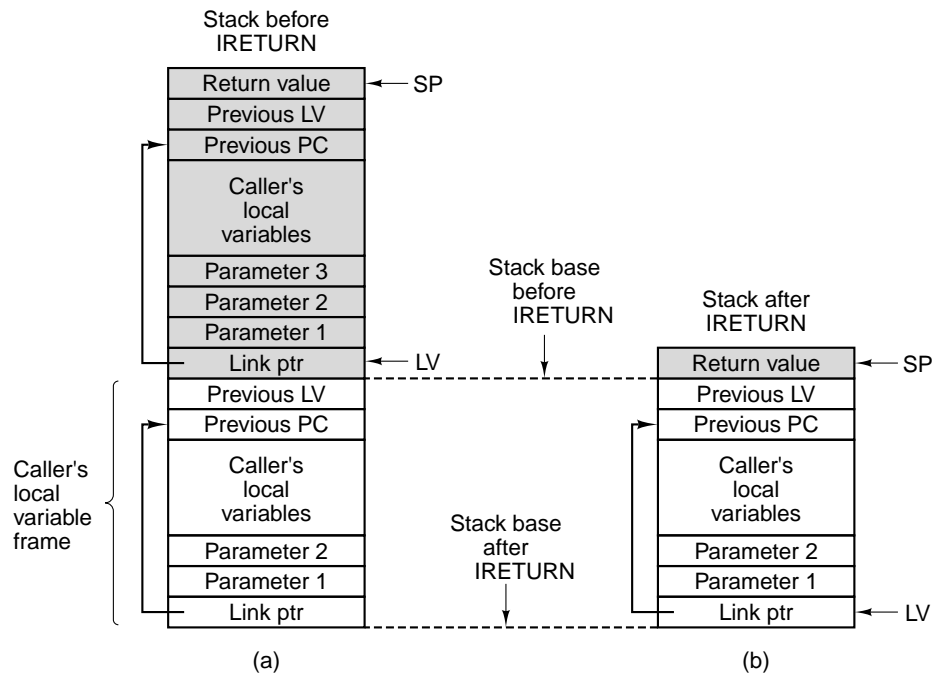
Abbildung 17.19: Wirkung des Befehls `if_icmpeq` auf den Stack.

muss gesichert werden, denn bei der Beendigung der Prozedur muss dieses Register wieder seinen alten Wert haben.

Wir erreichen dies, indem wir zunächst durch Erhöhen des Stack-Pointer Platz für die lokalen Variablen auf dem Stack schaffen. Wir erhöhen den Stack-Pointer dabei gleich so, dass wir auch noch Platz haben, um die aktuellen Werte von PC und LV abzuspeichern. Anschließend setzen wir das Register LV so, dass es auf das Argument OBJREF zeigt. An diese Stelle schreiben wir dann einen so genannten Link-Zeiger, der auf die Stelle im Stack zeigt, wo der alte Wert des Programm-Zählers abgelegt ist. Unmittelbar über diese Stelle legen wir dann noch den alten Wert des Registers LV. Die rechte Seite der Abbildung 17.20 zeigt die Situation, die dann entstanden ist.

Abbildung 17.20: Wirkung des Befehls `“invokevirtual m”` auf den Stack.

Der Befehl **ireturn** dient dazu, die Kontrolle von einer aufgerufenen Methode an die aufrufende Methode zurückzugeben. Wenn der Befehl **ireturn** aufgerufen wird, ist die Situation so, wie im linken Teil der Abbildung 17.21 auf Seite 268 gezeigt. Der Stack-Pointer zeigt auf den Rückgabe-Wert. (Bei der JVM muss jede Methode einen Wert zurückgeben.) Unmittelbar unter diesem Wert liegen die Werte, welche die Register LV und PC beim Aufruf der Methode hatten. Darunter liegen die lokalen Variablen der aufgerufenen Methode inklusive der Argumente, mit denen die Methode aufgerufen wurde. Alle diese Werte werden durch den Aufruf von **ireturn** vom Stack entfernt und der Rückgabe-Wert wird jetzt an die Stelle im Stack gelegt, an der beim Aufruf der Methode das Dummy-Argument **OBJREF** abgelegt worden ist. Das Register LV wird auf den Wert zurückgesetzt, den es vor dem Aufruf der Methode hatte. Schließlich wird PC so gesetzt, dass es auf den Bytecode zeigt, der dem Aufruf der Methode unmittelbar folgt.

Abbildung 17.21: Wirkung des Befehls **ireturn** auf den Stack.

Kapitel 18

Entwicklung eines einfachen Compilers

In diesem Kapitel konstruieren wir einen Compiler, der ein Fragment der Sprache C in eine vereinfachte Version von Java-Byte-Code übersetzt. Das von dem Compiler übersetzte Fragment der Sprache C 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.
Wir werden den Scanner mit Hilfe des Werkzeugs *JFlex* entwickeln.
2. Der Parser liest die Folge von Token und produziert als Ergebnis einen abstrakten Syntax-Baum.
Wir werden den Parser mit Hilfe von *JavaCup* generieren.
3. Der Typ-Checker überprüft den abstrakten Syntax-Baum auf Typ-Fehler.
Da die von uns übersetzte Sprache nur einen einzelnen Datentyp enthält, erübrigt sich diese Phase für den von uns entwickelten Compiler.
4. In realen Compilern erfolgt nun eine *Optimierungsphase*, die wir aber nicht betrachten.
5. Der Code-Generator übersetzt schließlich den Parse-Baum in eine Folge von JVM-Assembler-Befehlen.

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 JVM-Code ähnelt. Ein solcher Code wird als *Zwischen-Code* bezeichnet. Es bleibt dann die Aufgabe eines sogenannten *Backends*, daraus ein Assembler-Programm für eine gegebene Prozessor-Architektur zu 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.

18.1 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 18.1 zeigt die Grammatik der Quellsprache in erweiterter Backus-Naur-Form (EBNF). Die Grammatik für *Integer-C* verwendet die folgenden beiden Terminale:

1. ID steht für eine Folge von Ziffern, Buchstaben und dem Unterstrich, die mit einem Buchstaben beginnt.
Eine 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

```

program → function*

function → "int" ID "(" paramList ")" "{" decl* (stmt ";"*) "}"

paramList → ("int" ID ("," "int" ID)*)?

decl → "int" ID ","

stmt → "{" stmtList "}"
      | ID "=" expr
      | "if" "(" boolExpr ")" stmt
      | "if" "(" boolExpr ")" stmt "else" stmt
      | "while" "(" boolExpr ")" stmt
      | "return" expr
      | expr

expr → expr "+" expr
      | expr "-" expr
      | expr "*" expr
      | expr "/" expr
      | "(" expr ")"
      | NUMBER
      | ID
      | ID "(" (expr ("," expr)*)? ")"

boolExpr → expr "==" expr
          | expr "!=" expr
          | expr "<=" expr
          | expr ">=" expr
          | expr "<" expr
          | expr ">" expr
          | "!" boolExpr
          | boolExpr "&&" boolExpr
          | boolExpr "||" boolExpr

```

Abbildung 18.1: Eine Grammatik für *Integer-C*

Befehlen (*stmt*) folgt, die voneinander durch Semikolons getrennt werden. Der Aufbau der einzelnen Befehle ist dann ähnlich wie bei der Sprache SL, für die wir im Kapitel 11 einen Interpreter entwickelt haben.

Die in Abbildung 18.1 gezeigte Grammatik ist mehrdeutig:

1. Die Grammatik hat das *Dangling-Else-Problem*.

Da wir im Kapitel 15 bereits gesehen haben, wie dieses Problem professionell gelöst werden kann, nehme ich mir hier die Freiheit, JAVACUP mit der Option

```
"-expect 1
```

aufzurufen und dadurch die Fehlermeldung zu unterdrücken, denn wir hatten ja bereits gesehen, dass 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 18.2: Ein einfaches INTEGER-C-Programm.

Abbildung 18.2 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.

18.2 Entwicklung von Scanner und Parsers

Scanner und Parser werden mit Hilfe von *JFlex* und *JAVACUP* in der gleichen Art und Weise entwickelt, wie wir das bereits mehrfach in dieser Vorlesung gesehen haben. Abbildung 18.3 zeigt die Implementierung des Scanners. Der Scanner erkennt die verwendeten Operatoren und Schlüsselwörter, sowie Variablen und natürliche Zahlen. Gegenüber den bisher gesehenen *JFlex*-Scannern gibt es hier keine erwähnenswerten Unterschiede. Daher werden wir den Scanner nicht weiter diskutieren.

Bevor wir die Implementierung des Parsers diskutieren können, müssen wir angeben, durch welche Klassen wir den Syntax-Baum darstellen wollen. Wir beschreiben diese Klassen mit Hilfe der in Abbildung 18.4 auf Seite 273 gezeigten Spezifikation. Diese Spezifikation ist wie folgt zu lesen:

1. In line 1, the equation

```
Program = Program(List<Function> functionList)
```

specifies that the class `Program` has one member variable called `mFunctionList`. This variable has the type `List<Function>`. Therefore, an object of class `Program` is essentially a list of objects of class `Function`.

2. Similarly, in line 3 to 6 the equation

```
Function = Function(String      name,
                    List<String> parameterList,
                    List<Declaration> mDeclarations,
                    List<Statement> body);
```

specifies that the class `Function` has four attributes:

- (a) `mName` is a `String` storing the name of the function,

```

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     private Symbol symbol(int type, Object value) {
12         return new Symbol(type, yychar, yychar + yylength(), value);
13     }
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.GT        ); }
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 "[a-zA-Z][a-zA-Z_0-9]* { return symbol(sym.IDENTIFIER, yytext()); }
43 "0|[1-9][0-9]*       { return symbol(sym.NUMBER, new Integer(yytext())); }
44 "[\t\v\n\r]         { /* skip white space */ }
45 "//" [\n]*           { /* skip comments   */ }
46
47 "[^" { throw new Error("Illegal character '" + yytext() +
48                        "' at line " + yyline + ", column " + yycolumn); }

```

Abbildung 18.3: 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 18.4: Spezifikation der Klassen zur Darstellung des Syntax-Baums.

- (b) `mParameterList` is the list of formal parameters of the function,
- (c) `mDeclarations` is the list of local variable declarations occurring in the body of the function, while
- (d) `mBody` is the list of statements that make up the definition of the function.

3. Next, the equation

```

Statement = Block(List<Statement> statementList)
            + Assign(String var, Expr expr)
            + IfThen(BoolExpr boolExpr, Statement statement)
            + IfThenElse(BoolExpr condition, Statement then, Statement else)
            + While(BoolExpr condition, Statement statement)
            + Return(Expr expr)
            + ExprStatement(Expr expr);

```

tells us that there is an abstract class **Statement** and that the classes **Block**, **Assign**, \dots , **ExprStatement** are derived from this class.

- (a) **Block** is a class representing a list of statements enclosed in the curly braces “{” and “}”. This list of statements is stored in the member variable **mStatementList**.
 - (b) **Assign** is a class representing an assignment statement. Therefore, this class has two attributes:
 - **mVar** is the name of the variable on the left hand side of the assignment and
 - **mExpr** is the expression on the right hand side of the assignment.
 - (c) **IfThen** is a class representing an **if-then** statement without a trailing **else** clause. This class has two member variables:
 - **mBoolExpr** is the Boolean expression controlling whether the
 - **mStatement** is to be executed.
 - (d) **IfThenElse** is a class representing an **if-then-else** statement. This class has three member variables:
 - **mBoolExpr** is the Boolean expression controlling the execution.
 - **mThen** is the statement that is executed if **mBoolExpr** evaluates as **true**.
 - **mElse** is the statement that is executed if **mBoolExpr** evaluates as **false**.
 - (e) **While** is a class representing a **while** loop. This class has two member variables:
 - **mBoolExpr** is the Boolean expression controlling the loop.
 - **mStatement** is a statement that is executed as long as **mBoolExpr** evaluates to **true**.
 - (f) **Return** is a class representing a **return** statement. This class has the member variable **mExpr**. Evaluation of this expression yields the value to be returned.
 - (g) **ExprStatement** is a class representing an expression that is to be evaluated as a statement.
4. The equation in line 16 specifies that the class **Declaration** has one member variable with name **mVar**. This variable stores the name of the variable that is declared in the variable declaration associated with the corresponding object of class **Declaration**.
 5. Similarly, the equations defining **Expr** and **BoolExpr** specify the representation of arithmetical and Boolean expressions.

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 18.5 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 18.6 und 18.7 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.

Observe that the actions only construct the syntax tree. Everything else is then delegated to appropriate methods in the classes representing the syntax tree.

```

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 18.5: 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 18.6: Der erste Teil der Java-CUP-Grammatik.

```

1  expr ::= expr:l PLUS  expr:r      {: RESULT = new Sum(      1, r); :}
2      |  expr:l MINUS expr:r      {: RESULT = new Difference(1, r); :}
3      |  expr:l TIMES expr:r      {: RESULT = new Product(   1, r); :}
4      |  expr:l SLASH expr:r      {: RESULT = new Quotient(  1, 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(      1, r); :}
25      |  expr:l NEQUALS expr:r  {: RESULT = new Inequation(      1, r); :}
26      |  expr:l LE      expr:r  {: RESULT = new LessOrEqual(   1, r); :}
27      |  expr:l GE      expr:r  {: RESULT = new GreaterOrEqual(1, r); :}
28      |  expr:l LT      expr:r  {: RESULT = new LessThan(      1, r); :}
29      |  expr:l GT      expr:r  {: RESULT = new GreaterThan(   1, r); :}
30      |  NOT boolExpr:e      {: RESULT = new Negation(      e  ); :}
31      |  boolExpr:l AND boolExpr:r {: RESULT = new Conjunction(  1, r); :}
32      |  boolExpr:l OR  boolExpr:r {: RESULT = new Disjunction(  1, r); :}
33     ;

```

Abbildung 18.7: Der zweite Teil der Java-CUP-Grammatik.

18.3 Darstellung der Assembler-Befehle

Die Abbildungen 18.8 und 18.9 zeigen eine mögliche Übersetzung des Programms zur Berechnung der Summe $\sum_{i=1}^n i$ aus Abbildungen 18.2 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. Auch die Pseudo-Befehle, wie beispielsweise die Befehle `“.constant”` und `“.end-constant”` werden durch eine Klasse dargestellt. Abbildung 18.10 zeigt die Spezifikation dieser Klassen.

```

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 18.8: Eine Übersetzung des Programms aus Abbildung 18.2 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 “1” vorangestellt, so dass das Sprungziel als String interpretiert werden kann. Um die Generierung von Sprungzielen zu verstehen, betrachten wir die Klasse `LABEL`, die in Abbildung 18.11 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.

Zeile 9 zeigt die Umwandlung eines `LABEL`-Objektes in einen String, die zur Ausgabe der Assembler-Kommandos benutzt wird. Der eindeutigen Zahl wird der Buchstabe “1” vor- und der Doppelpunkt “:” nachgestellt, damit die Ausgabe der Syntax des JVM-Assemblers entspricht.

```

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 18.9: Übersetzung des Programms aus Abbildung 18.2 in Java-Assembler, 2. Teil.

2. Neben den eigentlichen Assembler-Kommandos zeigt Abbildung 18.10 noch die Definition von Klassen wie beispielsweise der Klasse MAIN, deren Implementierung in Abbildung 18.12 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.

18.4 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.

```

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 18.10: Darstellung der Assembler-Befehle als Klassen.

18.4.1 Übersetzung arithmetischer Ausdrücke

Die Übersetzung eines arithmetischen Ausdrucks *expr* soll Code erzeugen, durch dessen Ausführung das Ergebnis der Auswertung auf den Stack gelegt wird. Zu diesem Zweck deklariert die abstrakte Klasse *Expr*, die in Abbildung 18.13 gezeigt ist, die Methode

```
public abstract List<AssemblerCmd> compile();
```

die als Ergebnis 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.

```

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 18.11: Die Klasse LABEL.

```

1  public class MAIN extends AssemblerCmd {
2      public MAIN() {}
3      public String toString() { return ".main"; }
4  }

```

Abbildung 18.12: Die Klasse MAIN.

```

1  import java.util.*;
2
3  public abstract class Expr {
4      public abstract List<AssemblerCmd> compile();
5  }

```

Abbildung 18.13: Die Klasse Expr.

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 18.14](#) 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

$$-128 \leq c < 128$$

```

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 18.14: Die Klasse `Variable`.

erfüllt sind, dann kann die Konstante mit Hilfe des Befehl

```
bipush c
```

auf den Stack gelegt werden.

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.

Abbildung 18.15 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
```

```

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 18.15: Die Klasse MyNumber.

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:

$$\text{compile}(\text{lhs} \text{ "+" } \text{rhs}) = \text{lhs.compile}() + \text{rhs.compile}() + [\text{iadd}],$$

wobei der Operator "+" hier die Verkettung von Listen bezeichnet. Abbildung 18.16 zeigt die Umsetzung dieser Überlegung. Die Übersetzung von Ausdrücken der Form

$$\text{lhs} - \text{rhs}, \quad \text{lhs} * \text{rhs} \quad \text{und} \quad \text{lhs} / \text{rhs}$$

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 Form $f(e_1, \dots, e_n)$ kann übersetzt werden, indem zunächst die Ausdrücke e_1, \dots, 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:

$$\text{compile}(f(e_1, \dots, e_n)) = [\text{ldc_w OBJREF}] + \text{compile}(e_1) + \dots + \text{compile}(e_n) + [\text{invokevirtual } f]$$

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:

$$\text{compile}(\text{getchar}()) = [\text{in}].$$

¹ Die Befehle `imul` und `idiv` gehören zwar nicht zu dem ursprünglich von Tanenbaum in [Tan05] definierten Sprach-Umfang der JVM, in dem Buch [Str07] wird aber gezeigt, dass sich auch diese Befehle im Micro-Code des Prozessors *Mic-1* implementieren lassen.

```

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 18.16: Die Klasse Sum.

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:

$$compile(putchar(e)) = e.compile() + [out, bipush 1].$$

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 18.17 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. 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.

18.4.2 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

$$lhs == rhs$$

```

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 18.17: Die Klasse FunctionCall.

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. 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.

```

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 18.18: Die Klasse Equation.

Damit hat der erzeugte Code insgesamt die folgende Form

$$\begin{aligned}
 \text{compile}(\text{lhs} == \text{rhs}) &= \text{lhs.compile()} \\
 &+ \text{rhs.compile()} \\
 &+ [\text{isub}] \\
 &+ [\text{ifeq true}] \\
 &+ [\text{bipush 0}] \\
 &+ [\text{goto next}] \\
 &+ [\text{true:}] \\
 &+ [\text{bipush 1}] \\
 &+ [\text{next:}]
 \end{aligned}$$

Diese Gleichung ist in der Methode *compile()* eins zu eins umgesetzt worden.

Übersetzung von negierten Gleichungen

Die Übersetzung einer negierten Gleichung der Form

$$\text{lhs} != \text{rhs}$$

verläuft analog zu der Übersetzung einer Gleichung, denn wir müssen hier nur die Rollen von 0 und 1 vertauschen. Daher lautet die Spezifikation

```
compile(lhs != rhs) = lhs.compile()
                    + rhs.compile()
                    + [ isub ]
                    + [ ifeq false ]
                    + [ bipush 1 ]
                    + [ goto next ]
                    + [ false: ]
                    + [ bipush 0 ]
                    + [ next: ]
```

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

$$lhs \leq rhs$$

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:

```
compile(lhs <= rhs) = lhs.compile()
                    + rhs.compile()
                    + [ isub ]
                    + [ bipush 1 ]
                    + [ isub ]
                    + [ iflt true ]
                    + [ bipush 0 ]
                    + [ goto next ]
                    + [ true: ]
                    + [ bipush 1 ]
                    + [ next: ]
```

Die Umsetzung dieser Idee ist in Abbildung 18.19 zu sehen. Die Übersetzung von Ungleichungen der Form

$$lhs < rhs, \quad lhs > rhs, \quad \text{und} \quad lhs \geq rhs$$

verläuft im Wesentlichen analog und wird daher nicht weiter diskutiert.

Übersetzung von Konjunktionen

Die Übersetzung einer Konjunktion der Form

$$lhs \ \&\& \ rhs$$

```

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 18.19: Die Klasse `LessOrEqual`.

kann wie folgt spezifiziert werden:

$$\begin{aligned}
 compile(lhs \&\& rhs) &= lhs.compile() \\
 &+ rhs.compile() \\
 &+ [iand]
 \end{aligned}$$

Abbildung 18.20 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.

Eine Disjunktion wird in analoger Weise auf den Assembler-Befehl `ior` zurück geführt.

```

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 18.20: Die Klasse Conjunction.

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:

```

compile(!expr)  =  [ bipush 1 ]
                  +  expr.compile()
                  +  [ isub ]

```

Abbildung 18.21 zeigt die Umsetzung dieser Idee.

18.4.3 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

$$x = \text{expr}$$

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:

```

compile(x=expr) =  expr.compile()
                  +  [ istore x ]

```

```

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 18.21: Die Klasse `Negation`.

Die Idee wird in der in Abbildung [18.22](#) gezeigten Klasse `Assign` umgesetzt.

```

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 18.22: Die Klasse `Assign`.

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:

$$\begin{aligned} \text{compile}(\text{expr};) &= \text{expr.compile()} \\ &+ [\text{pop}] \end{aligned}$$

Abbildung 18.23 zeigt die Klasse `ExprStatement`, in der diese Überlegung umgesetzt wird.

```

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 18.23: 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:

$$\begin{aligned} \text{compile}(\text{if}(\text{expr}) \text{ statement}) &= \text{expr.compile()} \\ &+ [\text{ifeq else}] \\ &+ \text{statement.compile()} \\ &+ [\text{else:}] \end{aligned}$$

Diese Spezifikation ist in der Abbildung 18.24 umgesetzt worden.

Die Übersetzung eines Verzweigungs-Befehls der Form

if (expr) thenStmnt else elseStmnt

erfolgt in analoger Art und Weise. Diesmal lautet die Spezifikation:

$$\begin{aligned} \text{compile}(\text{if}(\text{expr}) \text{ thenStmnt else elseStmnt}) &= \text{expr.compile()} \\ &+ [\text{ifeq else}] \\ &+ \text{thenStmnt.compile()} \\ &+ [\text{goto next}] \\ &+ [\text{else:}] \\ &+ \text{elseStmnt.compile()} \\ &+ [\text{next:}] \end{aligned}$$

Diese Spezifikation ist in der Abbildung 18.25 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
10     public List<AssemblerCmd> compile() {
11         List<AssemblerCmd> result = mBoolExpr.compile();
12         LABEL      elseLabel = new LABEL();
13         AssemblerCmd ifeq      = new IFEQ(elseLabel.getLabel());
14         result.add(ifeq);
15         result.addAll(mStatement.compile());
16         result.add(elseLabel);
17         return result;
18     }
19     public BoolExpr getBoolExpr() {
20         return mBoolExpr;
21     }
22     public Statement getStatement() {
23         return mStatement;
24     }
25 }

```

Abbildung 18.24: Die Klasse IfThen.java

Die Übersetzung einer Schleife

Die Übersetzung einer **while**-Schleife der Form

while (*cond*) *statement*

orientiert sich an der folgenden Spezifikation:

$$\begin{aligned}
 \text{compile}(\text{while } (cond) \text{ stmnt}) &= [\text{loop:}] \\
 &+ \text{cond.compile()} \\
 &+ [\text{ifeq next}] \\
 &+ \text{stmnt.compile()} \\
 &+ [\text{goto loop}] \\
 &+ [\text{next:}]
 \end{aligned}$$

Die Umsetzung dieser Spezifikation sehen Sie in [Abbildung 18.26](#).

Übersetzen einer Liste von Befehlen

Eine in geschweiften Klammern eingeschlossene Liste von Befehlen der Form

$\{ \text{stmnt}_1; \dots \text{stmnt}_n; \}$

wird dadurch übersetzt, dass die Listen, die bei der Übersetzung der einzelnen Befehle stmnt_i entstehen, aneinander gehängt werden:

$$\text{compile}(\{ \text{stmnt}_1; \dots \text{stmnt}_n; \}) = \text{compile}(\text{stmnt}_1) + \dots + \text{compile}(\text{stmnt}_n).$$

Diese Idee ist in der Klasse **Block** realisiert worden. [Abbildung 18.27](#) zeigt diese Klasse.

```

1  public class IfThenElse extends Statement {
2      private BoolExpr  mExpr;
3      private Statement mThen;
4      private Statement mElse;
5
6      public IfThenElse(BoolExpr expr, Statement thenStmnt, Statement elseStmnt) {
7          mExpr = expr;
8          mThen = thenStmnt;
9          mElse = elseStmnt;
10     }
11     public List<AssemblerCmd> compile() {
12         List<AssemblerCmd> result = mExpr.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 getExpr() {
26         return mExpr;
27     }
28     public Statement getThen() {
29         return mThen;
30     }
31     public Statement getElse() {
32         return mElse;
33     }
34 }

```

Abbildung 18.25: Die Klasse IfThenElse.

18.4.4 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 18.28 zeigt die Klasse `Function`, allerdings ohne die Implementierung der Methode `compile()`, die wir aus Platzgründen in die Abbildung 18.29 ausgelagert haben.

Die Klasse `Function` enthält vier Member-Variablen:

1. `mName` gibt den Namen der Funktion an.

```

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 18.26: Die Klasse While.

```

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 stmtnt: mStatementList) {
10             result.addAll(stmtnt.compile());
11         }
12         return result;
13     }
14     public List<Statement> getStatementList() {
15         return mStatementList;
16     }
17 }

```

Abbildung 18.27: 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 18.28: Die Klasse `Function`.

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.

Die eigentliche Arbeit der Klasse `Function` wird in der Methode `compile()`, die in Abbildung 18.29 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          v1
4          ⋮
5          vk
6      .end-var
7          s1
8          ⋮
9          sn
10         halt
11     .end-main

```

Hier bezeichnen v_1, \dots, v_k die in der Funktion definierten lokalen Variablen. und s_1, \dots, 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  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 stmt: mBody) {
17             result.addAll(stmt.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 stmt: mBody) {
35             result.addAll(stmt.compile());
36         }
37         AssemblerCmd endMethod = new END_METHOD();
38         result.add(endMethod);
39     }
40     return result;
41 }

```

Abbildung 18.29: Die Methode *compile()*.

```

1      .method mName (p1, ..., pl)
2      .var
3          v1
4          ⋮
5          vk
6      .end-var

```

```

7          s1
8          ⋮
9          sn
10         .end-method

```

Hier bezeichnen p_1, \dots, p_l die formalen Parameter, die der Funktion übergeben werden, v_1, \dots, v_k bezeichnen wieder die lokalen Variablen und s_1, \dots, s_n sind die Assemblerbefehle des Rumpfes der Funktion.

```

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 18.30: Die Klasse `Program`.

Zum Abschluss diskutieren wir die Klasse `Program`, die in [Abbildung 18.30](#) 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.

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 18.2 gezeigte Funktion zur Berechnung der Summe $\sum_{i=1}^n i$ mit dem Compiler, so erhalten wir die in den Abbildungen 18.8 und 18.9 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 hinaus.

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.
- [CS70] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
- [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.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The JavaTM 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.
- [Kas65] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA, 1965.
- [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.
- [Kle09] Gerwin Klein. JFlex User's Manual: Version 1.4.3. Technical report, 2009. Available at: <http://jflex.de/jflex.pdf>.
- [Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

- [Les75] Michael E. Lesk. Lex – A lexical analyzer generator. Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.
- [LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, Sebastopol, 2nd edition, 1992.
- [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] Anil Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.
- [Nic93] G. T. Nicol. *Flex: The Lexical Scanner Generator, for Flex Version 2.3.7*. FSF, 1993.
- [Par07] Terence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.
- [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.
- [You67] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3 . *Information and Control*, 10(2):189–208, 1967.