

Formale Sprachen und ihre Anwendungen im Compilerbau

— WS 2009 & SS 2010 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

21. Mai 2010

Inhaltsverzeichnis

1	Einführung und Motivation	4
1.1	Grundlegende Definitionen	4
1.2	Überblick über die Vorlesung	6
1.3	Literatur	8
2	Reguläre Ausdrücke	9
2.1	Einige Definitionen	9
2.2	Algebraische Vereinfachung regulärer Ausdrücke	12
3	Die Werkzeuge <i>Flex</i> und <i>JFlex</i>	15
3.1	Das Werkzeug <i>Flex</i>	15
3.1.1	Ein einfaches Beispiel	16
3.1.2	Übersetzung des Beispiels	18
3.1.3	Fallstricke	18
3.1.4	Reguläre Ausdrücke in <i>Flex</i>	18
3.1.5	Ein Beispiel	21
3.1.6	Start-Zustände	24
3.1.7	Zusammenfassung	27
3.2	<i>JFlex</i>	28
3.2.1	Struktur einer <i>JFlex</i> -Spezifikation	28
3.2.2	Reguläre Ausdrücke in <i>JFlex</i>	30
3.2.3	Weitere Optionen	31
3.2.4	Noten-Berechnung mit <i>JFlex</i>	32
3.2.5	Zustände in <i>JFlex</i>	34
4	Endliche Automaten	35
4.1	Deterministische endliche Automaten	35
4.2	Nicht-deterministische endliche Automaten	39
4.3	Äquivalenz von EA und NEA	43
4.4	Übersetzung regulärer Ausdrücke in NEA	48
4.5	Übersetzung eines EA in einen regulären Ausdruck	51
4.6	Minimierung endlicher Automaten	55
5	Die Theorie regulärer Sprachen	60
5.1	Abschluss-Eigenschaften regulärer Sprachen	60
5.2	Erkennung leerer Sprachen	62
5.3	Äquivalenz regulärer Ausdrücke	63
5.4	Grenzen regulärer Sprachen	64

6	Kontextfreie Sprachen	68
6.1	Kontextfreie Grammatiken	68
6.1.1	Ableitungen	71
6.1.2	Parse-Bäume	73
6.1.3	Mehrdeutige Grammatiken	74
6.2	Top-Down-Parser	76
6.2.1	Umschreiben der Grammatik	76
6.2.2	Implementierung eines Top-Down-Parser	79
6.2.3	Implementierung eines rückwärts arbeitenden Top-Down-Parser	84
6.2.4	Erzeugung und Auswertung eines Syntax-Baums	87
7	Antlr — <i>Another Tool for Language Recognition</i>	94
7.1	Ein Parser für arithmetische Ausdrücke	94
7.2	Ein Parser zur Auswertung arithmetischer Ausdrücke	97
7.3	Erzeugung abstrakter Syntax-Bäume	100
7.3.1	Implementierung des Parsers	102
8	Bison	104
8.1	Der Scanner	104
8.2	Der Parser	109
9	Keller-Automaten	113
9.1	Definition eines Keller-Automaten	113
9.2	Von kontextfreien Sprachen zu Keller-Automaten	116
9.3	Von Keller-Automaten zu kontextfreien Sprachen	123
10	Grenzen kontextfreier Sprachen	127
10.1	Die Chomsky-Normalform	127
10.1.1	Beseitigung nutzloser Symbole	128
10.1.2	Beseitigung von ε -Regeln	129
10.1.3	Beseitigung von Unit-Regeln	130
10.1.4	Überführung in Chomsky-Normalform	133
10.2	Parse-Bäume als Listen	133
10.3	Das Pumping-Lemma für kontextfreie Sprachen	135
10.4	Anwendungen des Pumping-Lemmas	137
10.4.1	Die Sprache $L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$	137
11	LL(k)-Sprachen	139
11.1	Links-Faktorisierung	139
11.1.1	<i>First</i> und <i>Follow</i>	142
11.1.2	LL(1)-Grammatiken	149
11.1.3	LL(k)-Grammatiken	153
12	Behandlung von Nicht-LL(1)-Sprachen in Antlr	156
12.1	Unterstützung von LL(*)-Sprachen	156
12.1.1	Motivation	156
12.1.2	Die Theorie der LL(*)-Sprachen	159
12.1.3	Behandlung von LL(*)-Grammatiken in ANTLR	166
12.2	Semantische Prädikate	167
12.3	Syntaktische Prädikate und Backtracking	170
12.3.1	Das Dangling-Else-Problem	170
13	Interpreter	175

14 Earley-Parser	184
14.1 Der Algorithmus von Earley	184
14.2 Implementierung	189
14.3 Korrektheit und Vollständigkeit	198
14.4 Analyse der Komplexität	201
15 LALR-Parser	203
15.1 Bottom-Up-Parser	203
15.2 Shift-Reduce-Parser	205
15.2.1 Implementierung eines <i>Shift-Reduce-Parsers</i>	208
15.3 SLR-Parser	218
15.3.1 Shift-Reduce- und Reduce-Reduce-Konflikte	225
15.4 Kanonische LR-Parser	228
15.5 LALR-Parser	231
15.6 Vergleich von SLR-, LR- und LALR-Parsern	234
15.6.1 <i>SLR-Sprache</i> \subsetneq <i>LALR-Sprache</i>	234
15.6.2 <i>LALR-Sprache</i> \subsetneq <i>kanonische LR-Sprache</i>	235
15.6.3 Bewertung der verschiedenen Methoden	236
15.7 Erzeugung von LALR-Parsern mit <i>Bison</i>	236
15.8 Die Behandlung von Konflikten mit <i>Bison</i>	241
15.8.1 Operator-Präzedenzen	241
15.8.2 Das <i>Dangling-Else</i> -Problem	247
16 Der Parser-Generator <i>JavaCup</i>	251
16.0.3 Generierung eines CUP-Scanner mit Hilfe von <i>Flex</i>	254
16.0.4 Präzedenz und Assoziativität	257
17 Typ-Überprüfung	259
17.1 Eine Beispielsprache	260
17.2 Grundlegende Begriffe	261
17.3 Ein Algorithmus zur Typ-Überprüfung	265
17.4 Implementierung eines Typ-Checkers für TTL	268
18 Entwicklung eines einfachen Compilers	276
18.1 Die Programmiersprache <i>Integer-C</i>	276
18.2 Entwicklung von Scanner und Parsers	278
18.3 Darstellung der Assembler-Befehle	285
18.4 Die Code-Erzeugung	287
18.4.1 Übersetzung arithmetischer Ausdrücke	287
18.4.2 Übersetzung von Boole'schen Ausdrücken	292
18.4.3 Übersetzung der Befehle	297
18.4.4 Zusammenspiel der Komponenten	301

Kapitel 1

Einführung und Motivation

Die Theorie der formalen Sprachen beschäftigt sich mit der Frage, wie Sprachen beschaffen sein müssen, damit diese von einem Computer verarbeitet werden können. Diese Frage hat unter anderem die folgenden Anwendungen:

1. Das Design der Syntax von Programmier-Sprachen und die Entwicklung der dazugehörigen Compiler.
2. Das Design sogenannter *anwendungs-spezifischer Sprachen* (engl. *domain specific languages*). Ein typisches Beispiel einer anwendungs-spezifischen Sprache ist HTML.
3. Die Analyse und Verarbeitung natürlicher Sprache.

Große Teile der Theorie formaler Sprachen gehen auf Noam Chomsky (geb. 1928) [Cho56] zurück, der die Theorie der *kontextfreien Grammatiken* für die Analyse natürlicher Sprachen entwickelt hat.

4. Die Untersuchung der theoretischen Grenzen der Berechenbarkeit.

Neben verschiedenen Sprachklassen werden wir Automatenmodelle untersuchen, mit deren Hilfe die jeweiligen Sprachklassen erkannt werden können. Dabei werden wir erkennen, dass bestimmte naheliegende Fragen über formale Sprachen algorithmisch nicht entscheidbar sind.

1.1 Grundlegende Definitionen

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

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

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

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 Programmiersprachen 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

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

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

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

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

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

Definition 3 (Formale Sprache)

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

Beispiele:

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

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

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 Funktion

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

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

- (a) $\text{value}(0) = 0, \text{value}(1) = 1,$
- (b) $|w| > 0 \rightarrow \text{value}(w0) = 2 \cdot \text{value}(w),$

$$(c) \quad |w| > 0 \rightarrow \text{value}(w1) = 2 \cdot \text{value}(w) + 1.$$

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

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

1.2 Überblick über die Vorlesung

Wir werden in dieser Vorlesung die folgenden Themen behandeln:

1. Reguläre Ausdrücke

Reguläre Ausdrücke bieten eine Möglichkeit, einfache und effizient implementierbare formale Sprachen zu definieren. Die meisten modernen Programmiersprachen und insbesondere alle modernen Skriptsprachen (*Tcl*, *Perl*, *Python*, *Ruby*, ...) unterstützen die Verwendung regulärer Ausdrücke.

2. Anwendungen regulärer Ausdrücke

Es gibt eine Reihe von *Unix*-Werkzeugen, die auf der Verwendung regulärer Ausdrücke basieren. Wir werden das Werkzeug **flex** im Detail besprechen.

flex steht für *fast lexical analyser generator*. **flex** ist ein Werkzeug, mit dem es möglich ist, *Scanner* automatisch zu erzeugen. Ein *Scanner* ist ein Werkzeug, das einen gegebenen Text in eine Folge von Wörtern zu zerlegen. Ein Scanner wird beispielsweise in den meisten Compilern zur Vorverarbeitung der Programme eingesetzt.

3. Endliche Automaten

Die Erkennung reguläre Ausdrücke wird mit Hilfe sogenannter endlicher Automaten realisiert, diese bieten also eine Möglichkeit um reguläre Ausdrücke effizient zu implementieren.

4. Kontextfreie Sprachen

Reguläre Ausdrücke sind nicht ausreichend, um Programme beschreiben zu können, denn mit regulären Sprachen lassen sich rekursive Strukturen nicht erfassen. *Kontextfreie Sprachen* sind ein Konzept zur Beschreibung formaler Sprachen, mit dem sich auch rekursive Strukturen beschreiben lassen. Die Erfahrung zeigt, dass kontextfreie Sprachen ausreichend sind, um die Konstrukte moderner Programmiersprachen zu beschreiben. Da sich kontextfreie Sprachen darüber hinaus auch effizient implementieren lassen, werden moderne Programmiersprachen über kontextfreie Sprachen definiert.

5. ANTLR

ANTLR ist ein Werkzeug, mit dessen Hilfe sich aus einer kontextfreien Grammatik automatisch ein sogenannter *Top-Down-Parser* erzeugen lässt.

6. Bison

BISON erzeugt ähnlich wie ANTLR aus einer Grammatik einen Parser. Im Gegensatz zu ANTLR erzeugt Bison einen *Bottom-Up-Parser*.

7. Keller-Automaten

Ein *Bottom-Up-Parser* für eine kontextfreie Sprache wird mit Hilfe eines *Keller-Automaten* implementiert. Ein Keller-Automat ist eine Verallgemeinerung eines endlichen Automaten. Wir zeigen, wie die von Keller-Automaten erkannten Sprachen mit den kontextfreien Sprachen zusammenhängen.

8. Grenzen kontextfreier Sprachen

In diesem Kapitel beweisen wir einen Satz mit dessen Hilfe wir nachweisen können, dass bestimmte formale Sprachen keine kontextfreien Sprachen sind.

9. Theorie der LL(k)-Sprachen

Der Parser-Generator ANTLR basiert auf der Theorie der LL(k)-Sprachen.

10. Theorie der LL(*)-Sprachen

Die Theorie der LL(*)-Sprachen ist eine Erweiterung der Theorie der LL(k)-Sprachen, die teilweise in ANTLR umgesetzt ist.

11. Interpreter

Wir zeigen, wie sich die bisher entwickelte Theorie praktisch umsetzen lässt und entwickeln dazu einen Interpreter für eine einfache Programmiersprache.

12. Theorie der LALR-Sprachen

Die Klasse der LALR-Sprachen ist die Klasse der Sprachen, für die sich mit Hilfe von Werkzeugen wie *Bison* ein Parser erzeugen lässt.

13. Der Parser-Generator *JavaCup*

JavaCup ist ein LALR-Parser-Generator für *Java*.

14. Typ-Überprüfung

Bevor ein Programm übersetzt werden kann, muss sichergestellt werden, dass die Typen der Variablen konsistent deklariert sind. Diese Aufgabe wird anspruchsvoll, sobald generische Typen verwendet werden. Wir beleuchten die dafür benötigte Theorie.

15. Entwicklung eines einfacher Compiler

Hier entwickeln wir einen Compiler für ein Fragment der Sprache C. Zielsprache ist hierbei *Java ByteCode*.

16. Register-Zuordnung

Wir zeigen, wie wir den Variablen eines Programms Register zuordnen können und entwickeln einen Compiler, der Code für einen RISC-Prozessor erzeugt.

1.3 Literatur

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

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

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

2. *lex & yacc* [LMB92]

Das Buch beschreibt den Scanner-Generator `lex` und den Parser-Generator `yacc`. Diese beiden Generatoren sind die Vorläufer von `flex` und `bison`, die wir in unserer Vorlesung behandeln. Aus Sicht des Anwenders sind die Werkzeuge `flex` und `bison` weitgehend kompatibel mit `lex` und `yacc`.

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

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

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

Kapitel 2

Reguläre Ausdrücke

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

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

leicht spezifizieren.

2.1 Einige Definitionen

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

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

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

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

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

Definition 5 (Potenz einer Sprache) Es sei Σ eine Alphabet, $L \subseteq \Sigma^*$ eine formale Sprache und $n \in \mathbb{N}$ eine natürliche 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. $L^1 = \{\varepsilon\} \cdot \{\mathbf{ab}, \mathbf{ba}\} = \{\mathbf{ab}, \mathbf{ba}\}$,
3. $L^2 = \{\mathbf{ab}, \mathbf{ba}\} \cdot \{\mathbf{ab}, \mathbf{ba}\} = \{\mathbf{abab}, \mathbf{abba}, \mathbf{baab}, \mathbf{baba}\}$.

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

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

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

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

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

$$\mathbf{a}^n = \underbrace{\mathbf{a} \cdots \mathbf{a}}_n$$

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

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

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

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

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

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

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

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

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

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

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

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.

¹Für 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 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.

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

Aufgabe 1:

- 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.
- 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.
- Wieder sei $\Sigma = \{0, 1\}$. Geben Sie einen regulären Ausdruck für die Sprache $L \subseteq \Sigma^*$ der Wörter an, in denen der Teilstring 101 nicht auftritt.
- Welche Sprache wird durch den regulären Ausdruck $(1 + \varepsilon)(00^*1)^*0^*$ beschrieben?

2.2 Algebraische Vereinfachung regulärer Ausdrücke

Es gelten die folgenden Gesetze:

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

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

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

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

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

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

Diese Gleichung folgt aus der Assoziativität der Vereinigung.

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

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

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

Daraus folgt

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

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

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

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

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

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

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

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

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

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

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

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

Aufgabe 2: Vereinfachen Sie den folgenden regulären Ausdruck:

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

Kapitel 3

Die Werkzeuge *Flex* und *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, beispielsweise “hallo”,
 - (c) einzelne Buchstaben, 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.

Flex 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.

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

Die von *Flex* erzeugten Scanner sind C-Programme. Für das Java-Umfeld gibt es ein Äquivalent unter dem Namen *JFlex*.

3.1 Das Werkzeug *Flex*

Wir beginnen mit einem einfachen Beispiel und diskutieren anschließend einige Fallstricke von *Flex*, über die Anfänger häufig stolpern.

3.1.1 Ein einfaches Beispiel

Eine Eingabe-Datei für *Flex* besteht aus vier Abschnitten, die aufeinander folgen.

1. Der *Deklarations-Abschnitt* enthält die Deklarationen von Variablen und Hilfsfunktionen. Zusätzlich kann dieser Teil auch Kommentare und `include`-Anweisungen enthalten. Der Deklarations-Abschnitt wird durch den String “%{” eingeleitet und durch den String “%}” ausgeleitet. In dem in Abbildung 3.1 gezeigten Beispiel erstreckt sich der Deklarations-Abschnitt von Zeile 1 bis Zeile 7.

Die von *Flex* erzeugten Scanner sind C-Programme. Bei der Sprache C ist es erforderlich, dass Funktionen und Variablen vor ihrer Benutzung deklariert werden. Daher werden Variablen, die später im Regel-Abschnitt benutzt werden sollen, im Deklarations-Abschnitt deklariert.

Der Deklarations-Abschnitt ist optional: Falls im Regel-Abschnitt keine Variablen verwendet werden, dann kann der Deklarations-Abschnitt auch entfallen.

2. Der *Definitions-Abschnitt* enthält die Definitionen von sogenannten *regulären Definitionen*. Hierbei handelt es sich um Abkürzungen für komplexere reguläre Ausdrücke. Der Definitions-Abschnitt beginnt hinter dem Deklarationsteil und erstreckt sich bis zum ersten Auftreten des Strings “%%”. Der Definitions-Abschnitt kann leer sein. In dem in Abbildung 3.1 gezeigten Beispiel ist dies der Fall, denn dort folgt der String “%%” unmittelbar auf den String “%}”.

3. Der *Regel-Abschnitt* folgt auf den Definitions-Abschnitt und besteht aus Regeln der Form

`regex {cmds}`

Hier steht *regex* für einen regulären Ausdruck. Dieser muss am Zeilenanfang stehen. Jedemal, wenn der generierte Scanner in seiner Eingabe einen String erkennt, der diesem regulären Ausdruck entspricht, werden die in *cmds* angegebenen *Aktionen* ausgeführt. Genauer handelt es sich bei *cmds* um eine oder mehrere Anweisungen der Sprache C.

Der Regel-Abschnitt wird durch ein Auftreten des Strings “%%” beendet. In dem in Abbildung 3.1 gezeigten Beispiel erstreckt sich der Regel-Abschnitt von Zeile 9 bis Zeile 11.

4. Der *Programm-Abschnitt* enthält die Definition der Funktion `main()`, sowie eventuell die Definition weiterer Hilfsfunktionen. In Abbildung 3.1 erstreckt sich der Programm-Abschnitt von Zeile 13 bis Zeile 18.

Wir diskutieren jetzt das in Abbildung 3.1 gezeigte Beispiel im Detail. In diesem Beispiel wird ein Scanner spezifiziert, dessen Aufgabe es ist, alle Auftreten des Strings “Stephan” durch “Stefan” zu ersetzen. Zusätzlich gibt der Scanner am Ende aus, wieviele Ersetzungen tatsächlich durchgeführt worden sind.

1. Im Deklarations-Abschnitt binden wir die Datei “`stdio.h`” ein, damit wir später die Funktion `printf()` benutzen können. Zusätzlich deklarieren wir die Variable `numberChanges` und initialisieren sie mit dem Wert 0. In dieser Variable zählen wir die Anzahl der durchgeführten Ersetzungen.
2. Da die in diesem Beispiel verwendeten regulären Ausdrücke trivial sind, enthält das Beispiel keinen Definitions-Abschnitt.
3. Das Beispiel enthält in Zeile 10 und 11 jeweils eine Regel:
 - (a) In Zeile 10 ist der reguläre Ausdruck durch den String “Stephan” gegeben. Da dieser Ausdruck keinerlei Operatoren enthält, besteht die dadurch spezifizierte Sprache genau aus dem String “Stephan”. Die ausgeführte Aktion besteht aus zwei Kommandos:
 - i. Zunächst geben wir den String “Stefan” aus, denn wir wollen den String “Stephan” durch den String “Stefan” ersetzen.

```

1  %{
2  /* This trivial lexer replaces occurrences of the
3     string "Stephan" with "Stefan".
4  */
5  #include <stdio.h>
6  int numberChanges = 0; // number of occurrences changed
7  %{
8  %%
9
10 Stephan { printf("Stefan"); ++numberChanges; }
11 .      { printf("%s", yytext);          }
12 %%
13
14 int main() {
15     yylex();
16     printf("\nNumber of changes: %d\n", numberChanges);
17     return 0;
18 }

```

Abbildung 3.1: Eine einfache *Flex*-Spezifikation

- ii. Anschließend wird der Zähler `numberChanges` inkrementiert, denn wir haben ja nun eine Ersetzung durchgeführt.
- (b) In Zeile 11 besteht der reguläre Ausdruck nur aus dem Punkt `“.”`. Dieser reguläre Ausdruck spezifiziert ein beliebiges Zeichen, das von dem Zeilenumbruch `“\n”` verschieden ist. Dieses Zeichen geben wir mit dem Befehl

```
printf("%s", yytext);
```

aus. Hier benutzen wir die in *Flex* vordefinierte Variable `“yytext”`, die genau den Text enthält, der mit dem regulären Ausdruck erkannt worden ist. In diesem Fall besteht dieser Text immer aus genau einem Zeichen.

An dieser Stelle fragen Sie sich vielleicht, was passiert, wenn der Scanner auf einen Zeilenumbruch stößt. Ein Zeilenumbruch würde weder von der ersten noch von der zweiten Regel erfasst. Die Konvention bei *Flex* ist, dass alle Zeichen, die von keiner Regel erfasst werden, unverändert ausgegeben werden. Aus diesem Grunde hätten wir die zweite Regel ebenfalls weglassen können.

Die regulären Ausdrücke der beiden Regeln überlappen sich, denn beispielsweise kann der erste Buchstabe von `“Stephan”` auch durch den regulären Ausdruck `“.”` erkannt werden. Damit könnten im Prinzip die Aktionen beider Regeln ausgeführt werden. Falls zwei verschiedene Regeln angewendet werden können, geht *Flex* nach folgender Konvention vor:

- (a) Zunächst gewinnt die Regel, die auf den längeren Text passt. In dem Beispiel wird also bei jedem Auftreten von `“Stephan”` die erste Regel angewendet, denn diese passt auf den gesamten Text, während die zweite Regel nur auf einen einzigen Buchstaben passt.
 - (b) Ist der Text für zwei Regeln gleich lang, so entscheidet die Reihenfolge, in der die Regeln in der *Flex*-Spezifikation auftreten: Die Regel, die früher auftritt, gewinnt.
4. Im Programm-Abschnitt definieren wir die Funktion `main()`. Diese besteht im wesentlichen aus dem Aufruf der Funktion `yylex()`, die von *Flex* automatisch erzeugt wird. Diese Funktion startet den erzeugten Lexer. Dieser Lexer liest seine Eingabe Zeichen für Zeichen ein und überprüft nach jedem gelesenen Zeichen, ob eine der im Regel-Abschnitt angegebenen Regeln

anwendbar ist und führt gegebenenfalls die Aktionen dieser Regel aus. Nach dem Aufruf von `yylex()` geben wir noch die Anzahl der durchgeführten Ersetzungen aus.

3.1.2 Übersetzung des Beispiels

Wir speichern die in Abbildung 3.1 gezeigte *Flex*-Spezifikation in einer Datei mit dem Namen `“change.1”` und rufen anschließend *Flex* mit dem Befehl

```
flex change.1
```

auf. Dieser Aufruf erzeugt das C-Programm `“lex.yy.c”`, das den generierten Scanner enthält. Dieses Programm übersetzen wir mit dem Befehl

```
gcc -c lex.yy.c -o lex.yy.o
```

in die Objekt-Datei `“lex.yy.o”`. Um daraus ein ausführbares Programm zu erzeugen, binden wir die *Flex*-Bibliothek `“fl”` (`flex library`) mit dem Befehl

```
gcc -o change lex.yy.o -lfl
```

ein. Dabei entsteht die ausführbare Datei `“change”`. Haben wir eine weitere Datei `“test.txt”`, so können wir den erzeugten Scanner durch den Aufruf

```
./change < test.txt
```

testen. Bei diesem Aufruf liest der erzeugte Scanner seine Eingabe von der Datei `“test.txt”` und gibt die Ergebnisse am Bildschirm aus.

3.1.3 Fallstricke

Bei der Erstellung von *Flex*-Spezifikationen ist auf die folgenden Punkte besonders zu achten.

1. Die Strings `“%%”`, mit denen der Definitions-Abschnitt von dem Regel-Abschnitt und der Regel-Abschnitt von dem Programm-Abschnitt getrennt werden, müssen am Anfang einer ansonsten leeren Zeile stehen.
2. Bei den Regeln muss der reguläre Ausdruck am Anfang der Zeile stehen.
3. Die Kommandos, die auf eine Regel folgen, müssen in der selben Zeile beginnen wie der zugehörige reguläre Ausdruck. Es ist sinnvoll, diese Kommandos immer in geschweifte Klammern einzuschließen. Dann müssen wir lediglich darauf achten, dass die öffnende Klammer `“{”` in der selben Zeile steht wie der zugehörige reguläre Ausdruck.

3.1.4 Reguläre Ausdrücke in *Flex*

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 *Flex* eine Reihe von Abkürzungen an, mit der komplexe reguläre Ausdrücke kompakter beschrieben werden können. Den regulären Ausdrücken von *Flex* 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:

OpSyms :=

{ `“.”`, `“*”`, `“+”`, `“?”`, `“|”`, `“ (“`, `“)”`, `“ [“`, `“]“`, `“ {“`, `“ }“`, `“ <“`, `“ >“`, `“ /“`, `“ \“`, `“ ^“`, `“ $“`, `“ ”“` }

Damit können wir nun die Menge *Regexps* der von *Flex* unterstützen regulären Ausdrücke induktiv definieren.

1. $c \in \text{Regexp}$ falls $c \in \Sigma_{\text{ASCII}} \setminus \text{OpSyms}$

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

2. $\backslash x \in \text{Regexp}$ 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*).

3. $\backslash abc \in \text{Regexp}$ 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.

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

Die Operator-Symbole können durch Voranstellen eines Backslashes spezifiziert werden.

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

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

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

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

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

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

8. $r^+ \in \text{Regexp}$ 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:

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

9. $r? \in \text{Regexp}$ 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 *Flex* nicht unterstützt wird.

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

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

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

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

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

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

13. 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 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 Variable “yytext”, in der der erkannte Text aufgesammelt wird, bekommt nur den Text zugewiesen, der dem regulären Ausdruck r_1 entspricht. Der Text, der dem regulären Ausdruck r_2 entspricht, wird dann von der nächsten Regel gematcht. In der angelsächsischen Literatur wird r_2 als *trailing context* bezeichnet.

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

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

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

$[aeiou]$

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

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

Die Menge aller kleinen lateinischen Buchstaben lässt sich durch

$[a-z]$

spezifizieren, es gilt also

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

Die Menge aller lateinischen Buchstaben zusammen mit dem Unterstrich kann durch

$[a-zA-Z_]$

beschrieben werden. *Flex* 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. Innerhalb von Bereichen verlieren die meisten Operator-Symbole ihre Sonderbedeutung und können ohne Backslash geschrieben werden. Innerhalb eines Bereiches zählen nur die Symbole

“_”, “^”, “]” und “\”

als Operator-Symbole. Damit erkennt der Bereich

$[+?*]$

also genau die Postfix-Operatoren “+”, “?” und “*”.

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.

3.1.5 Ein Beispiel

In diesem Abschnitt diskutieren wir eine Anwendung von *Flex*. Es geht dabei um die Auswertung von Klausuren. Bei der Korrektur einer Klausur lege ich eine Datei an, die das in dem in Abbildung 3.2 beispielhaft gezeigte Format besitzt.

```
1  Klausur: Algorithmen und Datenstrukturen
2  Kurs:    TIT07AIX
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 3.2: 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 *Flex*-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 3.3 zeigt ein *Flex*-Programm, das die Aufgabe der Notenberechnung löst. Wir diskutieren dieses Programm jetzt Zeile für Zeile.

1. In dem Deklarations-Abschnitt, der sich von Zeile 1 bis Zeile 9 erstreckt, binden wir zunächst die Header-Dateien “`stdlib.h`” und “`stdio.h`” ein. Anschließend deklarieren wir Variablen und Funktionen:

```

1  %{
2  /* This lexer computes grades. */
3  #include <stdlib.h>
4  #include <stdio.h>
5  int    sumPoints, maxPoints;
6  int    lineNumber = 1;
7  void   errorMsg();
8  float  note();
9  %}
10 ZAHL    0|[1-9][0-9]*
11 NAME    [A-Za-zöäüÖÄÜß]+[ ]+[A-Za-zöäüÖÄÜß]+
12 %%
13 [A-Za-z]+:.*\n { ++lineNumber;                }
14 {NAME}:/:      { printf("%s", yytext);
15                  sumPoints = 0;                }
16 :[ \t]+        { printf("%s", yytext);          }
17 {ZAHL}         { sumPoints += atoi(yytext);    }
18 -             { /* skip hyphens */            }
19 [ \t]          { /* skip white space */        }
20 ^[ \t]*\n      { ++lineNumber;                }
21 \n            { printf(" %3.1f\n", note());
22                  ++lineNumber;
23                  }
24 .             { errorMsg();                    }
25 %%
26 float note() {
27     return 7.0 - 6.0 * sumPoints / maxPoints;
28 }
29 void errorMsg() {
30     printf("invalid character '%s' at line %d\n", yytext, lineNumber);
31 }
32 int main(int argc, char* argv[]) {
33     maxPoints = atoi(argv[1]);
34     yylex();
35     return 0;
36 }

```

Abbildung 3.3: Ein *Flex*-Programm zur Berechnung von Noten

- (a) Die Variable `sumPoints` speichert die Summe aller Punkte, die ein Student in der Klausur erreicht hat und `maxPoints` speichert die Anzahl der Punkte, bei der die Note 1,0 erreicht wird.
 - (b) In der Variablen `lineNumber` speichern wir die Zeilennummer. Dies ist nützlich um später aussagekräftige Fehlermeldungen für den Fall geben zu können, wenn in der Eingabedatei etwas unerwartetes gefunden wird.
 - (c) Die in Zeile 7 deklarierte Funktion `errorMsg()` kann verwendet werden um Fehlermeldungen auszugeben. Diese Funktion wird im Programm-Abschnitt in den Zeilen 29 – 31 definiert.
 - (d) Die in Zeile 8 deklarierte Funktion `note()` dient der Berechnung von Noten. Diese Funktion wird in den Zeilen 26 – 28 definiert.
2. In dem Definitions-Abschnitt werden zwei Abkürzungen definiert:

- (a) Zeile 10 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 11 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.

3. Der Regel-Abschnitt erstreckt sich von Zeile 13 – 24.

- (a) Die Regel in Zeile 13 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. Da wir die Kopfzeilen nicht weiter verarbeiten wollen, inkrementieren wir lediglich den Zähler `lineNumber`, denn wir haben ja gerade einen Zeilenumbruch gelesen.
- (b) Die Regel in Zeile 14 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 `printf`-Befehls aus und setzen anschließend die Variable `sumPoints` auf 0. Dies ist erforderlich, weil diese Variable ja vorher noch die Punkte eines anderen Studenten enthalten könnte.
- (c) Die nächste Regel in Zeile 16 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 17 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 mit Hilfe der Bibliotheks-Funktion `atoi()` in eine Zahl umwandeln. Anschließend wird diese Zahl dann zu der Summe der Punkte hinzuaddiert.
Die Verwendung der Funktion `atoi()` ist übrigens der Grund, weshalb wir die Header-Datei “`stdlib.h`” einbinden müssen.
- (e) Für nicht bearbeitete Aufgaben enthält die Eingabe-Datei einen Bindestrich “-”. Diese Bindestriche werden durch die Regel in Zeile 18 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 19 Leerzeichen und Tabulatoren, die nicht auf einen Doppelpunkt folgen.
- (g) Die Regel in Zeile 20 dient dazu Zeilen einzulesen, die nur aus Leerzeichen und Tabulatoren bestehen. Hier muss lediglich der Zähler “`lineNumber`” inkrementiert werden.
- (h) Wenn wir 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 21 die erzielte Note und geben sie mit einer Stelle hinter dem Komma aus. Zusätzlich wird wieder der Zähler “`lineNumber`” inkrementiert.
- (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 24 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 sowie die Zeilennummer, in der dieses Zeichen gefunden wurde.

4. Der Programm-Abschnitt erstreckt sich von Zeile 26 bis zum Ende der Datei. Bemerkenswert ist hier lediglich die Implementierung der Funktion `main()`. Um die Notenberechnung durchführen zu können ist es erforderlich, dass wir dem Programm die maximale erreichbare Punktzahl mitteilen, denn diese Punktzahl geht nicht aus der Tabelle hervor. Wir übergeben diese Zahl der Funktion `main()` als erstes Argument. Dieses Argument liegt in `argv[1]` zunächst als String vor. Wir konvertieren es mit Hilfe der Funktion `atoi()` in eine Zahl, die wir in der globalen Variablen `maxPoints` abspeichern.

Das obige Beispiel löst eine vergleichsweise einfache Aufgabe. Natürlich könnte diese Aufgabe auch durch ein ganz normales C-Programm gelöst werden. Dieses C-Programm wäre allerdings länger als das oben gezeigte *Flex*-Programm und es wäre in jedem Fall deutlich schwieriger zu verstehen, denn reguläre Ausdrücke sind eine sehr prägnante Möglichkeit um die Syntax einzelner Token zu beschreiben.

3.1.6 Start-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

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

Der reguläre Ausdruck, der diese Art von Kommentaren spezifiziert, ist wie folgt:

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

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 “ $\backslash*+[^{*/}]$ ” einen String, der aus beliebig vielen “*”-Zeichen besteht, auf die dann aber noch ein Zeichen folgen muss, dass sowohl von “/” als auch von “*” verschieden ist.

Der Ausdruck “ $[^{*}]\backslash*+[^{*/}]$ ” 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  \\\[^\*|\*+\[^\*/\])\*+\/ { /* skip multi line comments */ }
7  \\\/.* { /* skip single line comments */ }
8
9  %%
10 int main() {
11     yylex();
12 }
```

Abbildung 3.4: Entfernung von Kommentaren aus einem C-Programm.

Abbildung 3.4 zeigt ein *Flex*-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. Um in ähnlichen Fällen ein lesbareres Programm schreiben zu können, gibt es in *Flex* sogenannte *Start-Zustände* (engl. *start conditions*), mit deren Hilfe komplizierte reguläre Ausdrücke vermieden werden können. Wir diskutieren diese Zustände an Hand eines Beispiels: Wir wollen eine HTML-Datei in eine Text-Datei konvertieren. Die *Flex*-Spezifikation, die in Abbildung 3.5 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.

Wir diskutieren jetzt die *Flex*-Spezifikation aus Abbildung 3.5 im Detail.

1. Im Definitions-Teil deklarieren wir in Zeile 1 die beiden Zustände **header** und **script** als *exklusive* Start-Zustände. Die allgemeine Syntax einer solchen Deklaration ist wie folgt:
 - (a) Am Zeilen-Anfang einer Zustands-Deklaration steht der String “%x” oder “%s”. Der String “%x” spezifiziert *exklusive* Zustände, der String “%s” spezifiziert *inklusive* Zustände. Den Unterschied zwischen diesen beiden Zustandsarten erklären wir später.
 - (b) Darauf folgt eine Liste der Namen der deklarierten Zustände. Die Namen werden durch Leerzeichen getrennt.
2. In Zeile 3 haben wir den String “<head>” in doppelte Hochkommata eingeschlossen. Dadurch verlieren die Operator-Symbole “<” ihre Bedeutung. Dies ist eine allgemeine Möglichkeit, um Operator-Symbole in *Flex* spezifizieren zu können. Wollen wir beispielsweise den String “a*”

```

1  %x header script
2  %%
3  "<head>"           { BEGIN(header);           }
4  "<script\" [^>\n]+>" { BEGIN(script);           }
5  "<\" [^>\n]+>"      { /* skip html tags */      }
6  "&nbsp;";           { printf(" ");           }
7
8  "<head>\"</head>\" { BEGIN(INITIAL);           }
9  "<head>.\|\\n      { /* skip anything else */ }
10
11 "<script>\"</script>\" { BEGIN(INITIAL); }
12 "<script>.\|\\n      { /* skip anything else */ }
13 %%
14 int main() { yylex(); }

```

Abbildung 3.5: *Flex*-Spezifikation zur Transformation von HTML in Text.

wörtlich erkennen, so können wir an Stelle von “a*” auch klarer

“a*”

schreiben.

Wird der String “<head>” erkannt, so wird in Zeile 3 die Aktion “BEGIN(header)” ausgeführt. Damit wechselt der Scanner aus dem Default-Zustand “INITIAL”, 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 “<INITIAL>” markiert.

Die Regeln, die mit dem Zustand “**header**” markiert sind, finden wir weiter unten in den Zeilen 8 und 9.

3. In Zeile 4 wechseln wir entsprechend in den Zustand “**script**” wenn wir ein öffnendes Script-Tag sehen.
4. In Zeile 5 werden alle restlichen Tags gelesen. Da die Aktion hier leer ist, werden diese Tags entfernt.
5. In Zeile 6 ersetzen wir die Zeichenreihe “ ” durch ein Blank. Wollten wir das Skript wirklich einsetzen, so hätten wir hier noch analoge Zeilen zur Verarbeitung von Umlauten und Sonderzeichen.
6. Zeile 8 beginnt mit der Zustands-Spezifikation “**header**”. Daher ist diese Regel nur dann aktiv, wenn der Scanner in dem Zustand “**head**” ist. Diese Regel sucht nach dem schließenden Tag “</head>”. Wird dieses Tag gefunden, so wechselt der Scanner zurück in den Default-Zustand INITIAL, in dem nur die Regeln verwendet werden, die nicht mit einem Zustand markiert sind.
7. Zeile 9 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.
8. Die Zeilen 11 und 12 enthalten entsprechende Regeln für den Zustand “**script**”.

Ist der Automat im Default-Zustand “INITIAL” und liest ein Zeichen, das nicht durch die obigen Regeln verarbeitet wird, so wird dieses Zeichen unverändert ausgegeben. Dadurch erreichen wir, dass der in der HTML-Datei enthaltene Text ausgegeben wird.

Aufgabe 3: Implementieren Sie eine *Flex*-Spezifikation, die aus einem C-Programm alle Kommentare entfernt. Das Programm soll mit Hilfe von Zuständen arbeiten.

3.1.7 Zusammenfassung

Flex ist ein mächtiges Werkzeug um effiziente Scanner zu erzeugen, das in den Werkzeugkasten jedes Informatikers gehört. Die obige Darstellung ist eher als Appetitanreger gedacht, denn für eine ausführliche Darstellung fehlt uns jetzt die Zeit. Wir werden das Thema *Flex* allerdings später noch einmal aufgreifen und zwar dann, wenn wir den Parser-Generator *Bison* diskutieren.

Falls Sie später einmal *Flex* selber einsetzen wollen, dann finden Sie weitere Informationen in dem Buch von Levine, Mason und Brown [LMB92], sowie in den Manual-Seiten und im Unix-Info-System.

Historisches Das Werkzeug *Lex*, das der Vorläufer von *Flex* ist, wurde 1975 vom Michael E. Lesk in den Bell Laboratories entwickelt [Les75].

3.2 JFlex

Das Werkzeug *JFlex* [Kle05] ist ein Scanner-Generator, der an Stelle eines C-Programms ein *Java*-Programm erzeugt. *JFlex* ist weitgehend analog zu *Flex* aufgebaut. 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.

3.2.1 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 3.6 auf Seite 29 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 3.6 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 3.6 erstreckt sich der *Aktions-Teil* von Zeile 17 bis Zeile 19.

Wir diskutieren nun die in Abbildung 3.6 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 3.7 auf Seite 29 gezeigt aussehen. Wir diskutieren die Spezifikation des Scanners aus Abbildung 3.6 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 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 fest, dass ein Scanner für Unicode erzeugt werden soll. Zum Scannen von Text-Dateien sollte diese Option immer gewählt werden.

```

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 3.6: Eine einfache Scanner-Spezifikation für *JFlex*

```

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

```

Abbildung 3.7: Eine Eingabe-Datei für den in Abbildung 3.6 spezifizierten Scanner.

- 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.

- 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.

- 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. Da die Aktion

hier nur aus einem Kommentar besteht, wird dieses 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 3.6 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 erzeugten 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.

3.2.2 Reguläre Ausdrücke in *JFlex*

Bei der Entwicklung von *JFlex* war ein Ziel, dass *JFlex* sich möglichst wenig von *Flex* unterscheiden sollte, damit Benutzer, die bereits mit *Flex* gearbeitet haben, sich ohne großen zusätzlichen Aufwand in *JFlex* zurecht finden. Bei der Spezifikation regulärer Ausdrücke wurde aber dennoch die Chance ergriffen, die Syntax zu verbessern. Im Vergleich zu der Syntax von *Flex* gibt es die folgenden Unterschiede:

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. *JFlex* bietet zusätzlich den Negations-Operator “!” als Präfix-Operator an. 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, nämlich alles außer dem String “*/”.

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

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

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

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

$$!(\neg r_1 \vee \neg r_2)$$

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.

3. Weiterhin bietet *JFlex* den *Upto*-Operator “~” 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

$$"/\ast" \sim "\ast/"$$

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

Wenn wir diesen regulären Ausdruck mit dem regulären Ausdruck

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

vergleichen, den wir im Kapitel über *Flex* für die Erkennung mehrzeiliger Kommentare hergeleitet hatten, dann erkennen wir, wie mächtig der Upto-Operator “~” ist.

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

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

zurück geführt. Dabei steht “!([^\wedge] \ast r [^\wedge] \ast)” für beliebigen Text, der r nicht enthält. Darauf folgt dann Text, der durch r beschrieben wird.

3.2.3 Weitere Optionen

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

1. %char

Mit dieser Option wird das Mitzählen von Zeichen aktiviert. Wird diese Option angegeben, so steht die Variable `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 `yline` 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 `ycolumn` zur Verfügung gestellt.

4. %cup

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

3.2.4 Noten-Berechnung mit *JFlex*

In diesem Abschnitt zeigen wir, wie die im letzten Abschnitt gezeigte Notenberechnung aussieht, wenn wir statt *Flex* das Programm *JFlex* benutzen. Abbildung 3.8 zeigt die *JFlex*-Spezifikation, aus der sich automatisch ein *Java*-Programm zur Noten-Berechnung erstellen läßt.

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 aber von *JFlex* selber gar nicht definiert, denn *JFlex* erwartet, dass diese Klasse von dem Parser, an dem der Scanner angeschlossen wird, definiert wird. Da wir gar keinen Parser anschließen wollen, wäre der Typ **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 aber, um die dem Programm als Argument übergebene Punktzahl einlesen zu können.
4. In den Zeilen 8 und 9 deklarieren wir die Variablen **mMaxPoints** und **mSumPoints** als Member-Variablen der erzeugten Klasse.
5. Die Methoden `note()` und `errorMsg()` haben sich gegenüber der *Flex*-Variante kaum geändert.
6. 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 dann durch den Aufruf in Zeile 22 gestartet.
7. Der Rest der Spezifikation unterscheidet sich nun nicht mehr wesentlich von der entsprechenden *Flex*-Spezifikation.

```

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[1]+"\"");
25         } catch (java.io.IOException e) {
26             System.out.println("IO error scanning file \""+argv[1]+"\"");
27             System.out.println(e);
28         } catch (Exception e) {
29             System.out.println("Unexpected exception:");
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 3.8: Berechnung von Noten mit Hilfe von *JFlex*.

3.2.5 Zustände in *JFlex*

Auch in *JFlex* können wir mit Zuständen arbeiten. Abbildung 3.9 zeigt ein *JFlex*-Programm, mit dessen Hilfe wir den Text aus einer HTML-Datei extrahieren können. Es entspricht weitgehend dem in Abbildung 3.5 auf Seite 26 erstellten *Flex*-Programms. Zur Deklaration der verschiedenen Zustände wird das Schlüsselwort “%xstate” verwendet.

```
1  package Converter;
2
3  %%
4
5  %class Html2Txt
6  %standalone
7  %line
8  %unicode
9
10 %xstate header script
11 %%
12
13 "<head>"           { yybegin(header);          }
14 "<script" [^>\n]+>" { yybegin(script);          }
15 "<" [^>\n]+>"      { /* skip html tags */      }
16 "[\n]+"           { System.out.print("\n");    }
17 "&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 3.9: Transformation einer HTML-Datei in eine reine Text-Datei

Aufgabe 4: 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 ausgibt. 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 *Flex* 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 regulären 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.

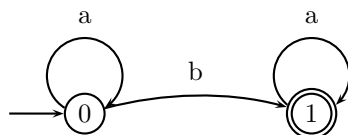


Abbildung 4.1: Ein einfacher endlicher Automat zur Erkennung der durch den regulären Ausdruck $a^* \cdot b \cdot a^*$ definierten Sprache.

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

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

Die einzelnen Komponenten haben die folgende Bedeutung:

1. Q ist die endliche *Menge der Zustände*.
2. Σ ist das *Eingabe-Alphabet*, also die Menge der Buchstaben, die der Automat als Eingabe verarbeitet.
3. $\delta : Q \times \Sigma \rightarrow Q \cup \{\Omega\}$
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 9 (akzeptierte Sprache, $L(A)$) Für einen endlichen Automaten $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ definieren wir die von A *akzeptierte Sprache* $L(A)$ wie folgt:

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

□

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 5: 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 10 Zu jedem endlichen deterministischen Automaten A gibt es einen vollständigen deterministischen Automaten \hat{A} , der die selbe Sprache akzeptiert wie der Automat A , es gilt also:

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

Beweis: Der Automat A habe die Form

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

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

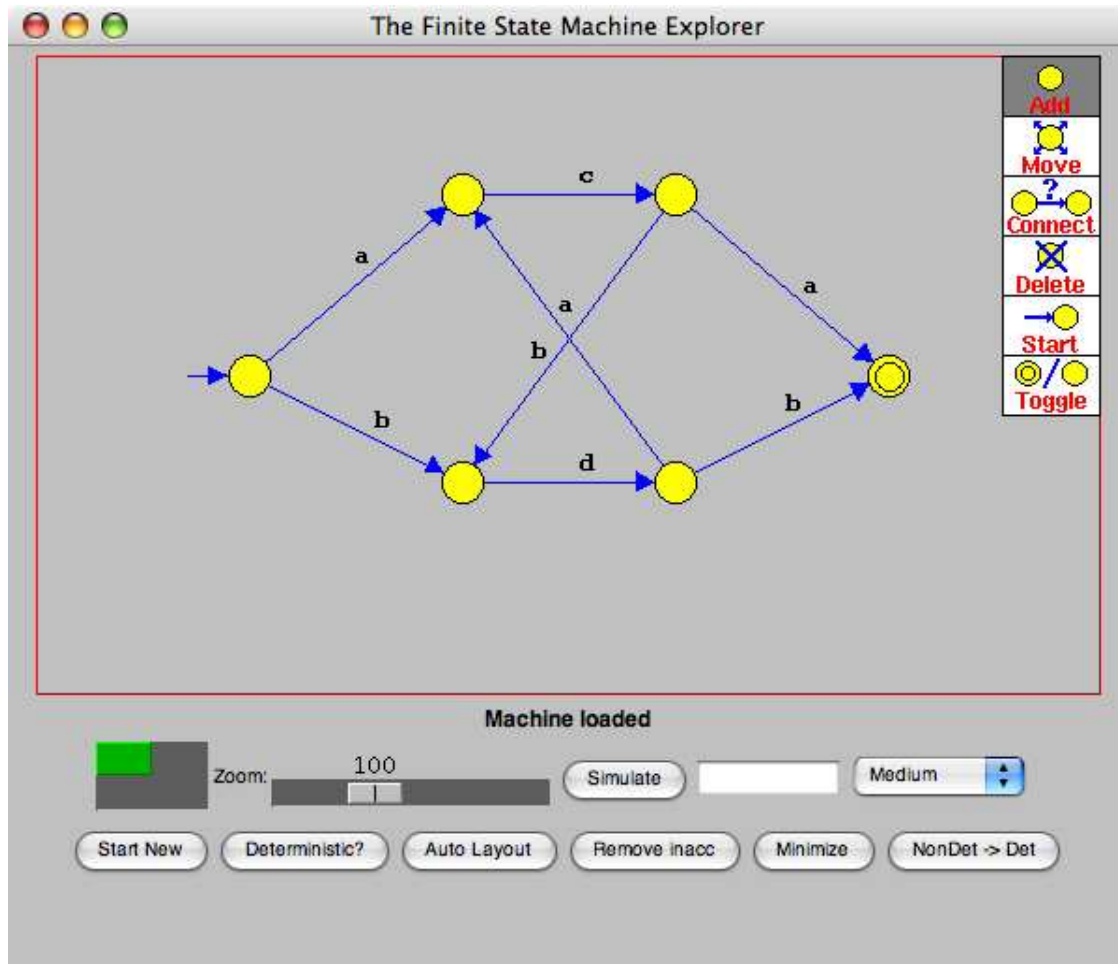


Abbildung 4.2: Das Werkzeug FSME.

Folge-Zustand in Q gibt, wenn also

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

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

Die formale Definition von \hat{A} verläuft wie folgt. Es bezeichne \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

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

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) $\hat{\delta}(\dagger, c) = \dagger$ 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

4.2 Nicht-deterministische endliche Automaten

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

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

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

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

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

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

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**":

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

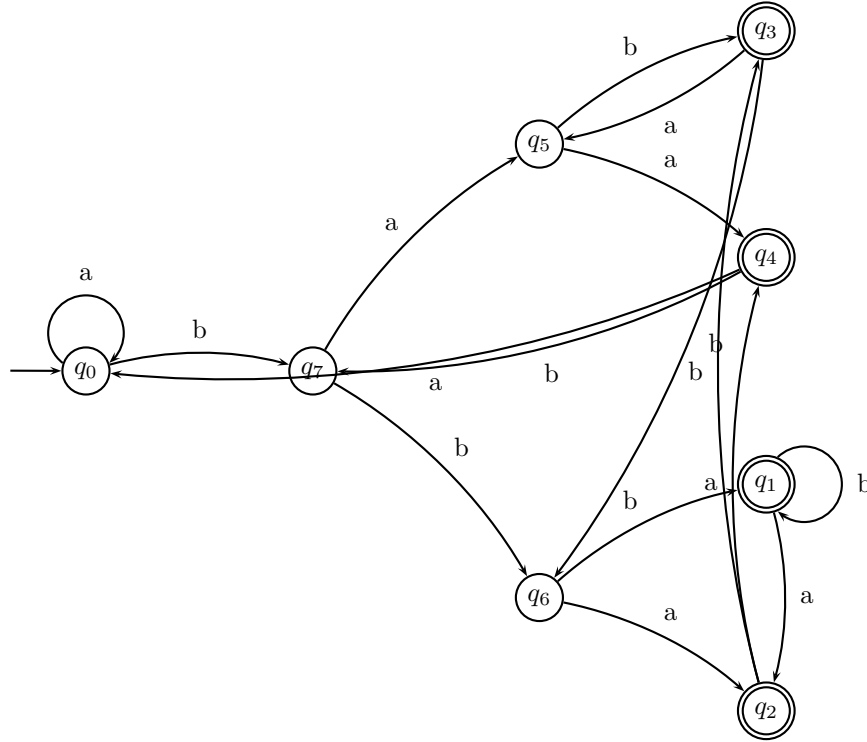


Abbildung 4.3: Ein endlicher Automat für die Sprache, die durch den regulären Ausdruck $(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$ definiert wird.

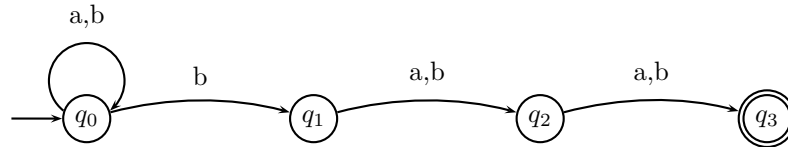


Abbildung 4.4: Ein Nicht-deterministischer Automat für die Sprache, die durch den regulären Ausdruck $(a + b)^* \cdot b \cdot (a + b) \cdot (a + b)$ definiert wird.

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

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 $“bbbbb”$ 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 Zei-

chens zum Tode des Automaten führt. Wäre der Automat beim Lesen des zweiten Buchstabens **b** noch im Zustand q_0 geblieben, so hätte er das Wort “**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 -*Transitionen*: 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 11 (NEA) Ein *nicht-deterministischer endlicher Automat* (abgekürzt NEA) ist ein 5-Tupel

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

so dass folgendes gilt:

1. Q ist die endliche Menge von Zuständen.
2. Σ ist das Eingabe-Alphabet.
3. δ ist eine Relation auf $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$, es gilt also

$$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q.$$

4. $q_0 \in Q$ ist der Start-Zustand.
5. $F \subseteq Q$ ist die Menge der akzeptierenden Zustände. □

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

Beispiel: Für den in Abbildung 4.4 auf Seite 40 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. $\delta = \{ \langle q_0, a, q_0 \rangle, \langle q_0, b, q_0 \rangle, \langle q_0, b, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_1, b, q_2 \rangle, \langle q_2, a, q_3 \rangle, \langle q_2, b, q_3 \rangle \}$.
4. Der Start-Zustand ist q_0 .
5. $F = \{q_3\}$, der einzige akzeptierende Zustand ist also q_3 . □

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

$$\langle q, s \rangle$$

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

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

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

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

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

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

Wir 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.

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

4.3 Äquivalenz von EA und NEA

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

$$A_1 = \langle Q, \Sigma, \delta, q_0, F \rangle$$

so in einen deterministischen endlichen Automaten A_2 übersetzen läßt, dass die von beiden Automaten erkannte Sprache gleich ist, dass also

$$L(A_1) = L(A_2)$$

gilt. Die Idee ist, dass der Automat A_2 die Menge aller der Zustände berechnet, in denen sich der Automat A_1 befinden könnte. Um diese Konstruktion angeben zu können, definieren wir zwei Hilfs-Funktionen. Wir beginnen mit dem sogenannten ε -Abschluss (engl. 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.: $p \in ec(q) \wedge \langle p, \varepsilon, r \rangle \in \delta \rightarrow r \in ec(q)$.

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

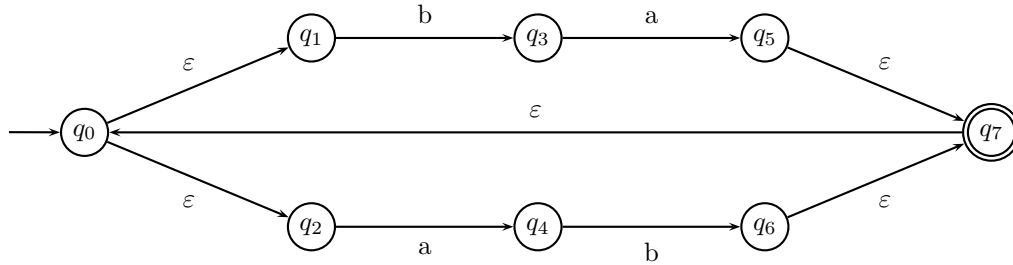


Abbildung 4.5: Nicht-deterministischer Automat mit ε -Transitionen.

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

1. $ec(q_0) = \{q_0, q_1, q_2\}$,
2. $ec(q_1) = \{q_1\}$,
3. $ec(q_2) = \{q_2\}$,
4. $ec(q_3) = \{q_3\}$,
5. $ec(q_4) = \{q_4\}$,
6. $ec(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
7. $ec(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,

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

□

Um den am Anfang des Abschnitts angegebenen nicht-deterministischen Automaten A_1 in einen deterministischen Automaten umwandeln zu können, transformieren wir die Relation 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_1 sich befinden kann, wenn er im Zustand q zunächst den Buchstaben c liest und anschließend eventuell noch einen oder auch mehrere ε -Transitionen durchführt. Formal erfolgt die Definition von δ^* durch die Formel

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

Diese Formel ist wie folgt zu lesen:

1. Zunächst berechnen wir die Menge

$$\{q_2 \in Q \mid \langle q_1, c, q_2 \rangle \in \delta\}.$$

Dies ist die Menge aller Zustände q_2 , die von q_1 aus durch Lesen des Buchstabens c erreicht werden können.

2. Anschließend berechnen wir für jeden der Zustände q_2 den ε -Abschluss $\text{ec}(q_2)$ und vereinigen die Mengen, die wir so erhalten.

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. □

Die Funktion δ^* überführt einen Zustand in eine Menge von Zuständen. Da der zu entwickelnde endliche Automat A_2 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_1 sich befinden kann, wenn er sich zunächst in einem Zustand aus der Menge M befunden hat, dann der Buchstabe c gelesen wurde und anschließend eventuell noch ε -Transitionen ausgeführt werden. Die formale Definition lautet

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

Diese Formel ist 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_1 in einen deterministischen endlichen Automaten überführen zu können. Wir definieren

$$A_2 = \langle 2^Q, \Sigma, \Delta^*, ec(q_0), \hat{F} \rangle.$$

1. Die Menge der Zustände von A_2 besteht aus der Menge aller Teilmengen der Zustände von A_1 , 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_1 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 auftreten.
2. An dem Eingabe-Alphabet ändert sich nichts, denn der neue Automat A_2 soll ja die selbe Sprache erkennen wie der Automat A_1 .
3. Die oben definierte Funktion Δ^* gibt an, wie sich Zustands-Mengen bei Eingabe eines Zeichens ändern.
4. Der Start-Zustand des Automaten A_2 ist die Menge aller der Zustände, die von dem Start-Zustand q_0 des Automaten A_1 durch ε -Transitionen erreichbar sind.
5. Wir definieren die Menge \hat{F} der akzeptierenden Zustände, als die Menge der Teilmengen von Q , die einen akzeptierenden Zustands enthalten, wir setzen also

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

Beispiel: Wir zeigen, wie sich der in Abbildung 4.4 auf Seite 40 gezeigte nicht-deterministische Automat A_1 in einen deterministischen Automaten A_2 transformieren läßt.

1. Der Start-Zustand des deterministischen Automaten, den wir mit S_0 bezeichnen wollen, besteht aus der Menge, die nur den Knoten q_0 enthält:

$$S_0 = {q_0}.$$

2. Von dem Zustand q_0 geht F_1 beim Lesen von \mathbf{a} in den Zustand q_0 über. Also gilt

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

3. Von dem Zustand q_0 geht F_1 beim Lesen von \mathbf{b} in den Zustand q_0 oder q_1 über. Also gilt

$$\Delta^*({q_0}, \mathbf{b}) = {q_0, q_1}.$$

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

$$\Delta^*({q_0, q_1}, \mathbf{a}) = {q_0, q_2}.$$

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

$$\Delta^*(\{q_0, q_1\}, \mathbf{b}) = \{q_0, q_1, q_2\}$$

6. $\Delta^*(\{q_0, q_2\}, \mathbf{a}) = \{q_0, q_3\}$.

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

8. $\Delta^*(\{q_0, q_1, q_2\}, \mathbf{a}) = \{q_0, q_2, q_3\}$.

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

10. $\Delta^*(\{q_0, q_3\}, \mathbf{a}) = \{q_0\}$.

11. $\Delta^*(\{q_0, q_3\}, \mathbf{b}) = \{q_0, q_1\}$.

12. $\Delta^*(\{q_0, q_1, q_3\}, \mathbf{a}) = \{q_0, q_2\}$.

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

14. $\Delta^*(\{q_0, q_2, q_3\}, \mathbf{a}) = \{q_0, q_3\}$.

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

16. $\Delta^*(\{q_0, q_1, q_2, q_3\}, \mathbf{a}) = \{q_0, q_2, q_3\}$.

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

Damit haben wir alle Zustände des deterministischen Automaten. Wir definieren

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

$$S_5 = \{q_0, q_1, q_3\}, S_6 = \{q_0, q_2, q_3\}, S_7 = \{q_0, q_1, q_2, q_3\}$$

und setzen schließlich

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

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

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

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

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

Damit können wir nun einen deterministischen endlichen Automaten A_2 angeben, der die selbe Sprache akzeptiert wie der nicht-deterministische Automat A_1 :

$$A_2 := \langle \widehat{Q}, \Sigma, \Delta^*, S_0, \widehat{F} \rangle.$$

Abbildung 4.6 zeigt den Automaten A_2 . Wir erkennen, dass dieser Automat 8 verschiedene Zustände besitzt. Der ursprünglich gegebene nicht-deterministische Automat A_1 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 A_2 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_1 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

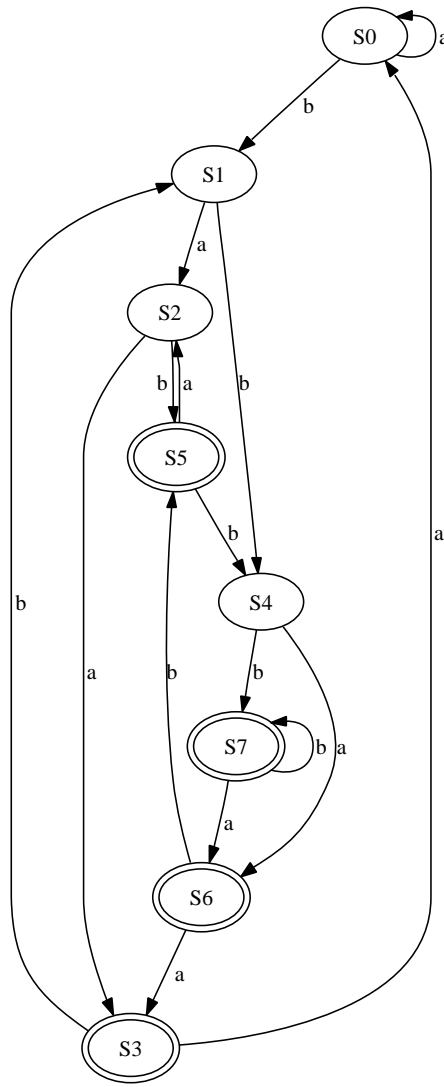


Abbildung 4.6: Der deterministische Automat A_2 .

von A_2 alle Mengen von 2^Q , die q_0 nicht enthalten, wodurch die Zahl der Zustände gegenüber der maximal möglichen Anzahl halbiert wird.

Aufgabe 7: Transformieren Sie den in Abbildung 4.5 auf Seite 43 gezeigten endlichen Automaten in einen äquivalenten deterministischen endlichen Automaten.

4.4 Übersetzung regulärer Ausdrücke in NEA

In diesem Abschnitt konstruieren wir zu einem gegebenen regulären Ausdruck r einen 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:

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

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:

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

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

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

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

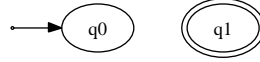


Abbildung 4.7: Der Automat $A(\emptyset)$.

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

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

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

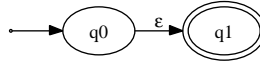


Abbildung 4.8: Der Automat $A(\varepsilon)$.

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

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

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

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

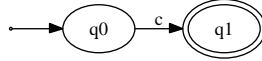


Abbildung 4.9: Der Automat $A(c)$.

4. Um den Automaten $A(r_1 \cdot r_2)$ definieren zu können, nehmen wir zunächst an, dass die Zustände der Automaten $A(r_1)$ und $A(r_2)$ verschieden sind. Dies können wir immer erreichen, indem wir die Zustände des Automaten $A(r_2)$ 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) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\} \rangle$.

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

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

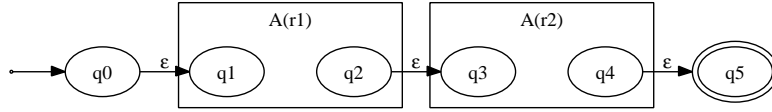


Abbildung 4.10: Der Automat $A(r_1 \cdot r_2)$.

Abbildung 4.10 zeigt den Automaten $A(r_1 \cdot r_2)$. 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 $A(r_1 \cdot r_2)$,
- (b) q_5 ist der einzige akzeptierende Zustand des Automaten $A(r_1 \cdot r_2)$.

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

- (a) Von dem neuen Start-Zustand q_0 gibt es eine ε -Transition zu dem Start-Zustand q_1 des Automaten $A(r_1)$.
- (b) Von dem akzeptierenden Zuständen q_2 des Automaten $A(r_1)$ gibt es eine ε -Transition zu dem Start-Zustand q_3 des Automaten $A(r_2)$.
- (c) Von dem akzeptierenden Zuständen q_4 des Automaten $A(r_2)$ gibt es eine ε -Transition zu dem akzeptierenden Zustand q_5 des Automaten $A(r_1 \cdot r_2)$.

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

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

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

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

Abbildung 4.11 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 $A(r_1 + r_2)$,

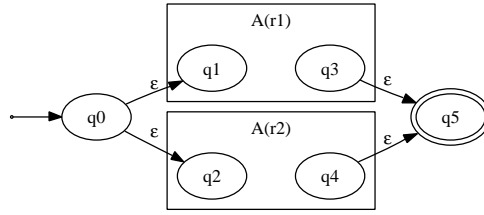


Abbildung 4.11: Der Automat $A(r_1 + r_2)$.

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

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

- (a) Von dem neuen Start-Zustand q_0 gibt es jeweils eine ε -Transition zu den Start-Zuständen q_1 und q_2 der Automaten $A(r_1)$ und $A(r_2)$.
 - (b) Von den akzeptierenden Zuständen q_3 und q_4 der Automaten $A(r_1)$ und $A(r_2)$ gibt es jeweils eine ε -Transition zu dem akzeptierenden Zustand q_5 .
6. Um den Automaten $A(r^*)$ für den Kleene-Abschluss r^* definieren zu können, schreiben wir $A(r)$ als

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

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

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

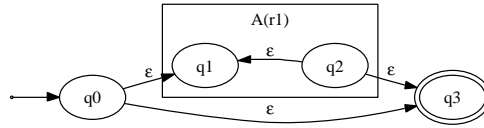


Abbildung 4.12: Der Automat $A(r^*)$.

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

- (a) q_0 ist der Start-Zustand des Automaten $A(r^*)$,
- (b) q_3 ist der einzige akzeptierende Zustand des Automaten $A(r^*)$.

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

- (a) Von dem neuen Start-Zustand q_0 gibt es jeweils eine ε -Transition zu den Zuständen q_1 und q_3 .
- (b) Von q_2 gibt es eine ε -Transition zurück zu dem Zustand q_1 .
- (c) Von q_2 gibt es eine ε -Transition zu dem Zustand q_3 .

Aufgabe 8: Bilden Sie einen nicht-deterministischen endlichen Automaten, der die durch den regulären Ausdruck

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

spezifizierte Sprache erkennt.

4.5 Übersetzung eines EA in einen regulären Ausdruck

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

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

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

$$Q_k := \{q_i \mid i \in \{0, \dots, n\} \wedge i \geq k\} = \{q_k, \dots, q_n\}$$

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.: $p \xrightarrow{\varepsilon}_k p$,

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

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

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 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,

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

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

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

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.
 - $q_k \xrightarrow{s_i}_k q_k$ 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 .
 - $q_k \xrightarrow{w_2}_k p_2$,
schließlich wechselt der Automat von dem Zustand q_k in den Zustand p_2 , wobei der Rest w_2 gelesen wird.

Daher definieren wir

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

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.

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

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

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 9: 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 Gleichung

$$r + r \cdot r^* \cdot r = r \cdot r^*$$

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:

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

(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

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

(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

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

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

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

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

4.6 Minimierung endlicher Automaten

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

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

minimiert werden kann. Wir suchen also 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 verarbeitet. 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.

Finden wir in einem endlichen Automaten zwei Zustände p und q , so dass für alle Buchstaben $c \in \Sigma$ gilt

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

so sind die beiden Zustände offenbar gleichwertig und einer der beiden Zustände ist überflüssig. Als einfaches Beispiel betrachten wir den endlichen Automaten, der wie folgt gegeben ist:

1. $Q = \{0, 1, 2, 3\}$,
2. $\Sigma = \{a, b\}$,
3. $\delta(0, a) = 1$, $\delta(0, b) = 2$, $\delta(1, a) = 0$, $\delta(2, a) = 0$, $\delta(1, b) = 3$, $\delta(2, b) = 3$,
4. $q_0 = 0$,
5. $F = \{3\}$.

Hier sind die Zustände 1 und 2 offenbar gleichwertig, denn wird ein a gelesen, kommen wir in jedem Fall in den Zustand 0, während ein b uns jedesmal in den Zustand 3 befördert. Daher ist einer dieser Zustände überflüssig und wir können den endlichen Automaten einfacher wie folgt definieren:

1. $Q = \{0, 1, 3\}$,
2. $\Sigma = \{a, b\}$,
3. $\delta(0, a) = 1$, $\delta(0, b) = 1$, $\delta(1, a) = 0$, $\delta(1, b) = 3$,
4. $q_0 = 0$,
5. $F = \{3\}$.

Dieser Automat enthält offenbar einen Zustand weniger als der ursprüngliche Automat. Wir könnten nun versuchen, einen Äquivalenz-Begriff für Zustände in Analogie zu dem obigen Beispiel zu entwickeln und zwei Zustände p und q dann als äquivalent betrachten, wenn für jeden Buchstaben die Zustände $\delta(p, c)$ und $\delta(q, c)$ identisch sind. Für einen endlichen Automaten

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

definieren wir $reduce(A, p \mapsto q)$ als den endlichen Automaten, der aus A in zwei Schritten entsteht:

1. Als erstes wird der Zustand p aus der Menge Q entfernt.
2. Dann wird die Zustands-Übergangs-Funktion δ zu einer Funktion $\delta_{p \mapsto q}$ wie folgt abgeändert:

$$\delta_{p \mapsto q}(r, c) = \begin{cases} \delta(r, c) & \text{falls } \delta(r, c) \neq p; \\ q & \text{falls } \delta(r, c) = p. \end{cases}$$

Insgesamt gilt also

$$reduce(A, p \mapsto q) = \langle Q \setminus \{p\}, \Sigma, \delta_{p \mapsto q}, q_0, F \setminus \{p\} \rangle.$$

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.

Es zeigt sich, dass es zur Minimierung nicht ausreicht, gleichwertige Zustände zu ersetzen. Statt dessen gehen wir jetzt andersherum vor und überlegen uns, wann zwei Zustände auf keinen Fall identifiziert werden können. Abbildung 4.13 zeigt einen deterministischen endlichen Automaten mit der Zustands-Menge $\{0, 1, 2, 3, 4\}$. Es zeigt sich, dass es in dieser Menge kein Paar von Zuständen gibt, die gleichwertig sind. Trotzdem werden wir später sehen, dass die Zustände 1 und 2 sowie die Zustände 3 und 4 in einem gewissen Sinne äquivalent sind.

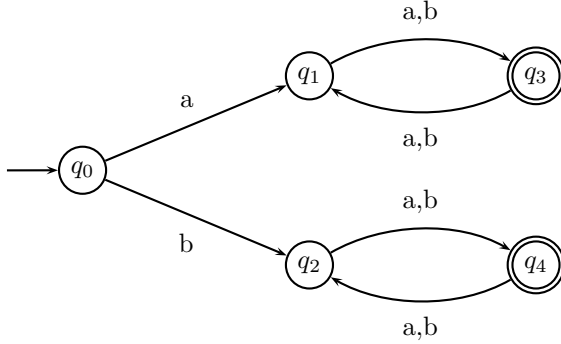


Abbildung 4.13: Ein endlicher Automat mit äquivalenten Zustände, die nicht gleichwertig sind.

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

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

□

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

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

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

Es bleibt die Frage zu klären, wie wir feststellen können, welche Zustände unterscheidbar sind. Eine Möglichkeit besteht darin, eine Menge V von Paaren von Zustände anzulegen. Wir fügen das Paar $\langle p, q \rangle$ in die Menge V ein, wenn wir erkannt haben, dass p und q unterscheidbar sind. Wir erkennen p und q als unterscheidbar, wenn es einen Buchstaben $c \in \Sigma$ und zwei Zustände s und t gibt, so dass gilt

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

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

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

```

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

1. Im ersten Schritt erkennen wir, dass die beiden akzeptierenden Zustände 3 und 4 von allen nicht-akzeptierenden Zuständen unterscheidbar sind. Also sind die Paare $\langle 0, 3 \rangle$, $\langle 0, 4 \rangle$, $\langle 1, 3 \rangle$, $\langle 1, 4 \rangle$, $\langle 2, 3 \rangle$ und $\langle 2, 4 \rangle$ unterscheidbar. Damit hat die Tabelle nun die folgende Gestalt:

	0	1	2	3	4
0				\times	\times
1				\times	\times
2				\times	\times
3	\times	\times	\times		
4	\times	\times	\times		

2. Als nächstes erkennen wir, dass die Zustände 0 und 1 unterscheidbar sind, denn es gilt

$$\delta(0, a) = 1, \quad \delta(1, a) = 3 \quad \text{und} \quad 1 \not\sim 3.$$

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

$$\delta(0, b) = 2, \quad \delta(2, b) = 4 \quad \text{und} \quad 2 \not\sim 4.$$

Tragen wir $0 \not\sim 1$ und $0 \not\sim 2$ in die Tabelle ein, so hat diese jetzt die folgende Gestalt:

	0	1	2	3	4
0		\times	\times	\times	\times
1	\times			\times	\times
2	\times			\times	\times
3	\times	\times	\times		
4	\times	\times	\times		

3. Nun finden wir keine weiteren Paare von unterscheidbaren Zuständen mehr, denn wenn wir das Paar $\langle 1, 2 \rangle$ betrachten, sehen wir

$$\delta(1, a) = 3 \quad \text{und} \quad \delta(2, a) = 4,$$

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

$$\delta(1, b) = 3 \quad \text{und} \quad \delta(2, b) = 4,$$

ein neues unterscheidbares Paar. Jetzt bleiben noch die beiden Zustände 3 und 4. Hier finden wir

$$\delta(3, c) = 1 \quad \text{und} \quad \delta(4, c) = 2 \quad \text{für alle } c \in \{a, b\}$$

und da die Zustände 1 und 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) $1 \sim 2$

(b) $3 \sim 4$

Abbildung 4.14 zeigt den entsprechenden reduzierten endlichen Automaten.

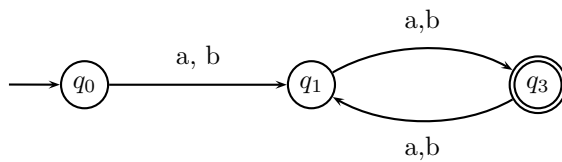


Abbildung 4.14: Der reduzierte endliche Automat.

Aufgabe 10: Konstruieren Sie den minimalen deterministischen endlichen Automaten, der die Sprache $L(a \cdot (b \cdot a)^*)$ erkennt. Gehen Sie dazu in folgenden Schritten vor:

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

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

Kapitel 5

Die Theorie regulärer Sprachen

Ist Σ ein Alphabet, so 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.

5.1 Abschluss-Eigenschaften regulärer Sprachen

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

wobei die Zustands-Übergangs-Funktion

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

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 $s \in \Sigma^*$:

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

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 15 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 16 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.

5.2 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 induktiv erfolgen:

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

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

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

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

5.3 Äquivalenz regulärer Ausdrücke

Definition 17 (\doteq) Zwei reguläre Ausdrücke r_1 und r_2 sind genau dann äquivalent (geschrieben $r_1 \doteq r_2$), wenn die durch r_1 und r_2 spezifizierten Sprachen identisch sind:

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

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

Satz 18 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 16 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 5.2 entscheidbar. \square

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

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

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

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

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

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

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

5.4 Grenzen regulärer Sprachen

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

Satz 19 (Pumping-Lemma) Es sei L eine reguläre Sprache. Dann gibt es eine natürliche Zahl $n \in \mathbb{N}$, so dass sich alle Strings $s \in L$, die länger als n sind, so in drei Teile u , v und w aufspalten lassen, dass die folgenden drei Bedingungen gelten:

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

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

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

Beweis: Da L eine reguläre Sprache ist, gibt es einen deterministischen endlichen 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 mehr als n Buchstaben besteht. Konkret gelte

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

wobei c_1, \dots, c_m die einzelnen Buchstaben sind. Wir betrachten eine Berechnung des Automaten

A , die das Wort s akzeptiert. 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 ist nach Voraussetzung größer als 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

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

können wir sogar sagen, dass 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

$$q_0 \xrightarrow{u} q_k. \tag{5.1}$$

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

$$q_k \xrightarrow{v} q_k. \tag{5.2}$$

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

$$q_k \xrightarrow{w} q_m. \tag{5.3}$$

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 $h \in \mathbb{N}$

$$q_k \xrightarrow{v^h} q_k \tag{5.4}$$

Aus den Gleichungen (5.1), (5.3) und 5.4) 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 20 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 Zahl n , so dass sich alle Strings $s \in L$, für die $|s| > n$ gilt, so in drei Teile u , v und w aufspalten lassen, dass

$$s = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{und} \quad \forall h \in \mathbb{N} : uv^h w \in L$$

gilt. Wir setzen

$$s := ({}^{n+1})^{n+1}.$$

Offenbar gilt $|s| = 2 \cdot (n+1) = 2 \cdot n + 2 > n$. Wir finden also jetzt drei Strings u , v und w , für die gilt:

$$({}^{n+1})^{n+1} = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{und} \quad \forall h \in \mathbb{N} : uv^h w \in L.$$

Wegen $|uv| \leq n$ und $v \neq \varepsilon$ wissen wir, dass der String v aus einer positiven Zahl öffnender runder Klammern bestehen muss:

$$v = ({}^k \quad \text{für ein } k \in \mathbb{N} \text{ mit } k > 0.$$

Setzen wir in der Formel $\forall h \in \mathbb{N} : uv^h w \in L$ für h den Wert 0 ein, so sehen wir, dass

$$uw \in L \tag{5.5}$$

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

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

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

$$\begin{aligned} \text{count}(uw, "(") &= \text{count}(uvw, "(") - \text{count}(v, "(") \\ &= \text{count}(s, "(") - \text{count}(v, "(") \\ &= n + 1 - k \end{aligned}$$

Andererseits gilt

$$\begin{aligned} \text{count}(uw, ")") &= \text{count}(uvw, ")") - \text{count}(v, ")") \\ &= \text{count}(s, ")") - \text{count}(v, ")") \\ &= n + 1 - 0 \end{aligned}$$

Wegen $k > 0$ folgt also

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

und damit kann der String uw im Widerspruch zum Pumping-Lemma nicht in der Sprache L liegen. 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 11: 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, dass die Sprache L_{square} keine reguläre Sprache ist.

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

Historisches Das Pumping-Lemma geht auf einen allgemeineren Satz zurück, der von Bar-Hillel, Perles und Shamir bewiesen wurde [BHPS61].

Kapitel 6

Kontextfreie Sprachen

Im letzten Kapitel haben wir gesehen, dass reguläre Sprachen nicht in der Lage sind, Klammern zu zählen. Damit sind sie offenbar nicht ausdrucksstark genug, um Programmier-Sprachen zu beschreiben, wir brauchen ein mächtigeres Konzept. In diesem Kapitel stellen wir daher die *kontextfreien* Sprachen vorher. Diese basieren auf dem Konzept der *kontextfreien Grammatik*, das wir gleich besprechen.

6.1 Kontextfreie Grammatiken

Kontextfreie Sprachen dienen zur Beschreibung von Programmier-Sprachen, insofern handelt es sich bei den kontextfreien Sprachen genau wie bei den regulären Sprachen auch um formale Sprachen. Allerdings wollen wir später beim Einlesen eines Programms nicht nur entscheiden, ob das Programm korrekt ist, sondern wir wollen darüber hinaus den Programm-Text *strukturieren*. Den Vorgang des *Strukturierens* bezeichnen wir auch als *parsen* und das Programm, das diese Strukturierung vornimmt, wird als *Parser* bezeichnet. Als Eingabe erhält ein Parser üblicherweise nicht den Text eines Programms, sondern statt dessen eine Folge sogenannter *Token*. Diese Token werden von einem Scanner erzeugt, der mit Hilfe regulärer Ausdrücke den Programmtext in einzelne Wörter aufspaltet, die wir in diesem Zusammenhang als *Token* bezeichnen. Beispielsweise spaltet der Scanner des C-Compilers ein C-Programm in die folgenden Token auf:

- Operator-Symbole, wie “+”, “+=”, “<”, “<=” etc.,
- vordefinierte Schlüsselwörter wie “if”, “while”, etc.,
- Namen für Variablen und Funktionen,
- Namen für Typen wie “int”, “char” oder auch benutzerdefinierte Typnamen,
- Konstanten für Zahlen (Fließkommazahlen und ganze Zahlen),
- String-Konstanten,
- 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. Wir müssen bei dieser Definition allerdings zusätzlich die Mengen *Product* und *Factor* definieren. Die Menge *Product* enthält arithmetische Ausdrücke, die Produkte und Quotienten darstellen und die Menge *Factor* enthält einzelne Faktoren. Wir setzen außerdem

voraus, dass durch den Scanner Zahlen als Token *Number* zurück gegeben werden, wobei der konkrete Wert dann an dieses Token angeheftet wird.

1. Ist A ein arithmetischer Ausdruck und ist P ein Produkt, so ist auch der String A “+” P ein arithmetischer Ausdruck:

$$A \in \text{ArithExpr} \wedge P \in \text{Product} \rightarrow A \text{ “+” } P \in \text{ArithExpr}.$$

Ein Wort zur Notation: Während in der obigen Formel A und P Variablen sind, die für beliebige Strings stehen, ist der String “+” wörtlich zu interpretieren und wurde deshalb in Gänsefüßchen eingeschlossen. Die Gänsefüßchen sind natürlich nicht Teil des arithmetischen Ausdrucks sondern dienen lediglich der Notation.

2. Genauso haben wir $A \in \text{ArithExpr} \wedge P \in \text{Product} \rightarrow A$ “-” $P \in \text{ArithExpr}$.
3. Jedes Produkt ist gleichzeitig auch ein arithmetischer Ausdruck

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

4. Ist P ein Produkt und ist F ein Faktor, so ist der String P “*” F ebenfalls ein Produkt:

$$P \in \text{Product} \wedge F \in \text{Factor} \rightarrow P \text{ “*” } F \in \text{Product}.$$

5. Ist P ein Produkt und ist F ein Faktor, so ist auch der String P “/” F ein Produkt:

$$P \in \text{Product} \wedge F \in \text{Factor} \rightarrow P \text{ “/” } F \in \text{Product}.$$

6. F ein Faktor, so ist F gleichzeitig auch ein Produkt:

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

7. Ist A ein arithmetischer Ausdruck und schließen wir diesen Ausdruck in Klammern ein, so erhalten wir einen Ausdruck, den wir als Faktor benützen können:

$$A \in \text{ArithExpr} \rightarrow \text{“(” } A \text{ “)”} \in \text{Factor}.$$

8. Jede Zahlenkonstante ist ein Faktor:

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

Die obige Definition werden wir jetzt in Form von *Grammatik-Regeln* noch einmal wesentlich kompakter wiedergeben:

$$\begin{aligned} \text{ArithExpr} &\rightarrow \text{ArithExpr “+” Product} \\ \text{ArithExpr} &\rightarrow \text{ArithExpr “-” Product} \\ \text{ArithExpr} &\rightarrow \text{Product} \\ \text{Product} &\rightarrow \text{Product “*” Factor} \\ \text{Product} &\rightarrow \text{Product “/” Factor} \\ \text{Product} &\rightarrow \text{Factor} \\ \text{Factor} &\rightarrow \text{“(” ArithExpr “)”} \\ \text{Factor} &\rightarrow \text{Number} \end{aligned}$$

Die Ausdrücke auf der linken Seite einer Grammatik-Regel bezeichnen wir als *syntaktische Variablen* oder auch als *Nicht-Terminal*. In der Literatur finden Sie hierfür auch den Begriff *syntaktische Kategorie*. In dem Beispiel sind *ArithExpr*, *Product* und *Factor* die Nicht-Terminal. Die restlichen Ausdrücke, in unserem Fall also *Number* und die Zeichen “+”, “-”, “*”, “/”, “(” und “)” bezeichnen wir als *Terminal* oder auch *Token*. Dies sind also genau die Zeichen, die nicht auf der linken Seite einer Grammatik-Regel stehen. Bei den Nicht-Terminalen gibt es zwei Arten:

1. Operator-Symbole und Trennzeichen wie beispielsweise “/” und “(”. Diese Nicht-Terminalen stehen für sich selbst.

2. Token wie *Number* ist zusätzlich ein Wert zugeordnet.

Üblicherweise werden Grammatik-Regeln in einer kompakteren Notation angegeben, indem alle Regeln für ein Nicht-Terminal wie folgt zusammengefaßt werden:

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

Hier werden also die einzelnen Alternativen einer Regel durch das Metazeichen \mid getrennt. Nach den obigen Beispielen geben wir jetzt die formale Definition.

Definition 21 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* G ist ein 4-Tupel

$$G = \langle V, T, R, S \rangle,$$

so dass folgendes gilt:

1. V ist eine Menge von Namen, die wir als *syntaktischen Variablen* oder auch *Nicht-Terminal* bezeichnen.

In dem obigen Beispiel gilt

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

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

$$T \cap V = \emptyset.$$

In dem obigen Beispiel gilt

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

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:

$$X \in V.$$

- (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. □

6.1.1 Ableitungen

Als nächstes wollen wir festlegen, welche Sprache durch eine gegebene Grammatik G definiert wird. Dazu definieren wir zunächst den Begriff eines *Ableitungs-Schrittes*. Es sei

1. $G = \langle V, T, R, S \rangle$ eine Grammatik,
2. $A \in V$ eine syntaktische Variable,
3. $\alpha A \beta \in (V \cup T)^*$ ein String aus Terminalen und syntaktischen Variablen, der die Variable A enthält,
4. $(A \rightarrow \gamma) \in R$ 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:

$$\begin{array}{l} S \rightarrow \varepsilon \\ | \quad "(S)" \end{array}$$

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

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

Der Beweis erfolgt durch Induktion über die Länge n des Wortes w .

I.A. $n = 0$: Dann gilt $w = \varepsilon$. Offenbar gilt

$$S \Rightarrow \varepsilon,$$

denn die Grammatik enthält die Regel $S \rightarrow \varepsilon$.

I.S. $n \mapsto n + 1$: Dann hat w die Form $w = "(v)"$, der String v ist kürzer als w und liegt in L . Also gibt es nach I.V. eine Ableitung von v :

$$S \Rightarrow^* v.$$

Dann bekommen wir insgesamt die folgende Ableitung

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

Als nächstes zeigen wir, dass jedes Wort w , dass sich aus S ableiten lässt, ein Element der Sprache L ist. Wir führen den Beweis durch Induktion über die Anzahl n der Ableitungs-Schritte:

I.A. $n = 1$: Die einzige Ableitung eines aus Terminalen aufgebauten Strings, die nur aus einem Schritt besteht, ist

$$S \Rightarrow \varepsilon.$$

Dann muss $w = \varepsilon$ gelten und wegen $\varepsilon = (^0)^0 \in L$ haben wir $w \in L$.

I.S. $n \mapsto n + 1$: Wenn die Ableitung aus mehr als einem Schritt besteht, dann muss die Ableitung die folgende Form haben:

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

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

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

□

Aufgabe 12: 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. $w = \varepsilon$. Dann setzen wir

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

I.S. $w = d v$ mit $d \in \Sigma$. Dann wird $\text{count}(d v, c)$ durch eine Fall-Unterscheidung definiert:

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

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!

Aufgabe 13: Wieder sei $\Sigma = \{\mathbf{a}, \mathbf{b}\}$. Wir definieren die Menge L als die Menge der Strings s , die sich nicht in der Forms $s = ww$ 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.

6.1.2 Parse-Bäume

Mit Hilfe einer Grammatik G können wir nicht nur erkennen, ob ein gegebener String s ein Element der von der Grammatik erzeugten Sprache $L(G)$ ist, wir können den String auch strukturieren indem wir einen *Parse-Baum* aufbauen. Ist eine Grammatik

$$G = \langle V, T, R, S \rangle$$

gegeben, so ist ein Parse-Baum für diese Grammatik ein Baum, der den folgenden Bedingungen genügt:

1. Jeder innere Knoten (also jeder Knoten, der kein Blatt ist), ist mit einem Nicht-Terminal beschriftet.
2. Jedes Blatt ist mit einem Terminal oder mit ε beschriftet.
3. Ist ein innerer Knoten mit einer Variablen A beschriftet und sind die Kinder dieses Knotens mit den Symbolen X_1, X_2, \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 6.1 zeigt einen Parse-Baum für das Wort “2*3+4”, der mit der oben angegebenen Grammatik für arithmetische Ausdrücke abgeleitet worden ist.

Da Bäume der in Abbildung 6.1 gezeigten Art sehr schnell zu groß werden, vereinfachen wir diese Bäume mit Hilfe der folgenden Regeln:

1. Ist n ein innerer Knoten, der mit der Variablen A beschriftet ist und gibt es unter den Kindern dieses Knotens genau ein Kind, dass mit einem Nicht-Terminal o beschriftet ist, so entfernen wir dieses Kind und beschriften den Knoten statt dessen mit dem Terminal o .

Diese Regel wird für den Fall verallgemeinert, dass ein innerer Knoten mehrere Kinder hat, die mit Terminalen o_1, \dots, o_k beschriftet sind. In diesem Fall wird der Knoten mit der Folge der Terminalen beschriftet und die mit diesen Terminalen beschrifteten Kinder werden entfernt.

2. Hat ein innerer Knoten nur ein Kind, so ersetzen wir diesen Knoten durch sein Kind.

Den Baum, den wir auf diese Weise erhalten, nennen wir den *abstrakten Syntax-Baum*. Abbildung 6.2 zeigt den abstrakten Syntax-Baum den wir erhalten, wenn wir den in Abbildung 6.1 gezeigten Parse-Baum nach diesen Regeln vereinfachen. Die in diesem Baum gespeicherte Struktur ist genau das, was wir brauchen um den arithmetischen Ausdruck “2*3+4” auszuwerten.

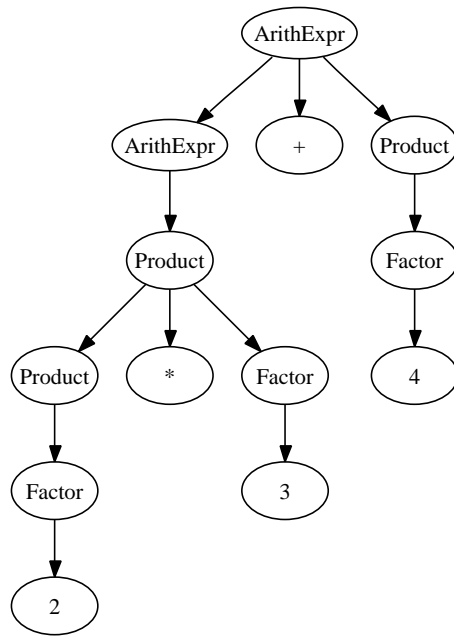


Abbildung 6.1: Ein Parse-Baum für den String “2*3+4”.

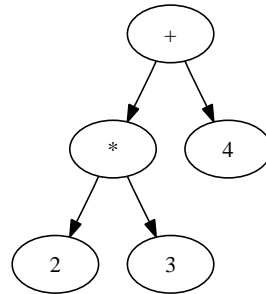


Abbildung 6.2: Ein abstrakter Syntax-Baum für den String “2*3+4”.

6.1.3 Mehrdeutige Grammatiken

Die zu Anfang des Abschnitts 6.1 angegebene Grammatik erscheint durch ihre Unterscheidung der syntaktischen Kategorien *ArithExpr*, *Product* und *Factor* unnötig kompliziert. Wir stellen eine einfachere Grammatik G vor, welche dieselbe Sprache beschreibt:

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

wobei die Regeln R wie folgt gegeben sind:

$$\begin{array}{lcl} Expr & \rightarrow & Expr \text{ “+” } Expr \\ & | & Expr \text{ “-” } Expr \\ & | & Expr \text{ “*” } Expr \\ & | & Expr \text{ “/” } Expr \\ & | & “(” Expr “)” \\ & | & Number \end{array}$$

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

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

Diese Ableitung entspricht dem abstrakten Syntax-Baum, der in Abbildung 6.2 gezeigt ist. Es gibt aber noch eine andere Ableitung des Strings “2*3+4” mit dieser Grammatik:

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

Dieser Ableitung entspricht der abstrakte Syntax-Baum, der in Abbildung 6.3 gezeigt ist. Bei dieser Ableitung wird der String “2*3+4” offenbar als Produkt aufgefasst, was der Konvention widerspricht, dass der Operator “*” stärker bindet als der Operator “+”. Würden wir den String an Hand des letzten Syntax-Baums auswerten, würden wir offenbar ein falsches Ergebnis bekommen!

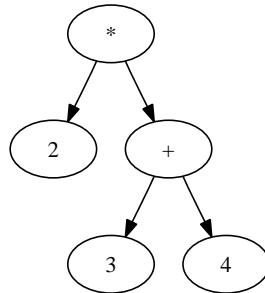


Abbildung 6.3: Ein anderer abstrakter Syntax-Baum für den String “2*3+4”.

Die Ursache dieses Problems ist die Tatsache, dass die zuletzt angegebene Grammatik mehrdeutig ist. Eine solche Grammatik ist zum Parsen ungeeignet.

6.2 Top-Down-Parser

In diesem Abschnitt stellen wir ein Verfahren vor, mit dem sich viele Grammatiken leicht parsen lassen. Die Grundidee ist einfach: Um einen String w mit Hilfe einer Grammatik-Regel der Form

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

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, dass 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. Wir können versuchen, den Parser mit Hilfe von *Backtracking* zu implementieren: Bei diesem Verfahren werden nacheinander alle Regeln ausprobiert, die beim Parsen in Frage kommen. Solche Parser lassen sich sehr einfach mit Hilfe der Sprache *Prolog* implementieren, denn bei der Programmier-Sprache *Prolog* ist *Backtracking* Bestandteil des Sprache.

Wir werden die ersten beiden oben genannten Verfahren an Hand der Grammatik für arithmetische Ausdrücke diskutieren.

6.2.1 Umschreiben der Grammatik

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

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

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

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

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

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

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 6.4 auf Seite 78 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

$$\begin{array}{lcl} A & \rightarrow & C L \\ L & \rightarrow & B L \mid \varepsilon. \end{array}$$

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

$$\begin{array}{lcl} A & \rightarrow & \gamma_1 L \mid \gamma_2 L \mid \dots \mid \gamma_l L \\ L & \rightarrow & \beta_1 L \mid \beta_2 L \mid \dots \mid \beta_k L \mid \varepsilon \end{array}$$

Wenden wir dieses Verfahren auf die in Abbildung 6.4 gezeigte Grammatik für reguläre Ausdrücke an, so erhalten wir die in Abbildung 6.5 gezeigte Grammatik. Die Variablen *ExprRest* und *ProductRest* können wie folgt interpretiert werden:

1. *ExprRest* beschreibt eine Liste der Form

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

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

2. *ProductRest* beschreibt eine Liste der Form

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

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

$Expr$	\rightarrow	$Expr \text{ "+" } Product$
	$ $	$Expr \text{ "-" } Product$
	$ $	$Product$
$Product$	\rightarrow	$Product \text{ "*" } Factor$
	$ $	$Product \text{ "/" } Factor$
	$ $	$Factor$
$Factor$	\rightarrow	$\text{"(" } Expr \text{ ")"}$
	$ $	$Number$

Abbildung 6.4: Links-rekursive Grammatik für arithmetische Ausdrücke.

$Expr$	\rightarrow	$Product \ ExprRest$
$ExprRest$	\rightarrow	$\text{"+" } Product \ ExprRest$
	$ $	$\text{"-" } Product \ ExprRest$
	$ $	ε
$Product$	\rightarrow	$Factor \ ProductRest$
$ProductRest$	\rightarrow	$\text{"*" } Factor \ ProductRest$
	$ $	$\text{"/" } Factor \ ProductRest$
	$ $	ε
$Factor$	\rightarrow	$\text{"(" } Expr \text{ ")"}$
	$ $	$NUMBER$

Abbildung 6.5: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

Aufgabe 14:

- (a) Die folgende Grammatik beschreibt reguläre Ausdrücke:

$RegExp$	\rightarrow	$RegExp \text{ "}" } RegExp$
		$RegExp \text{ "}" } RegExp$
		$RegExp \text{ "*" }$
		$\text{"(" } RegExp \text{ ")"}$
		"."
		LETTER

Diese Grammatik verwendet nur die syntaktische Variable $\{RegExp\}$ und die folgenden Terminale

$\text{"}"$, "*" , "." , "(" , ")" , LETTER.

Da die Grammatik mehrdeutig ist, ist diese Grammatik zum Parsen ungeeignet. Transformieren Sie diese Grammatik in eine eindeutige Grammatik. Orientieren Sie sich an der Grammatik für arithmetische Ausdrücke und führen Sie geeignete neue syntaktische Variablen ein.

- (b) Entfernen Sie die Links-Rekursion aus der in Teil (a) dieser Aufgabe erstellten Grammatik.

6.2.2 Implementierung eines Top-Down-Parser

Wir behandeln jetzt die Implementierung eines Top-Down-Parsers in *Java*. Bevor wir mit der Implementierung des eigentlichen Parsers beginnen können, benötigen wir einen *Scanner*, manchmal wird der auch als *Tokenizer* bezeichnet, der die Aufgabe hat, einen String aus ASCII-Zeichen zu lesen und in eine Liste aus *Terminalen* (manchmal sagen wir auch *Tokens*) zu verwandeln. In unserem Fall sind die Terminale die Operator-Symbole, die beiden Klammern und die Zahlen. Wir werden diesen Scanner mit Hilfe von *JFlex* entwickeln. Abbildung 6.6 zeigt die Implementierung dieses Scanners.

1. In Zeile 1 importieren wir alle Klassen aus dem Paket `Java.util`, denn wir benötigen später die Klasse `ArrayList` und das Interface `List`.
2. Die erzeugte Klasse bekommt in Zeile 5 den Namen `Scanner`.
3. In Zeile 10 legen wir die Member-Variable `mTokenList` an. In dieser Member-Variable sammeln wir die Liste aller Token auf, die wir lesen.
4. Code, der in den Strings `"%eof{"` und `"%eof}"` eingeschlossen ist, wird dann ausgeführt, wenn *JFlex* das Ende der zu lesenden Datei erreicht hat. Dann enthält die Variable `mTokenList` alle gelesenen Token. Wir legen daher in Zeile 15 ein Objekt der Klasse `ExprParser` an, dem wir die Liste der erkannten Tokens übergeben.
5. Anschließend rufen wir von diesem Objekt die Methode `parseExpr()` auf. Aufgabe dieser Methode ist das Parsen und Auswerten der arithmetischen Ausdrucks. Da dabei eventuell eine Ausnahme vom Typ `ParseException` auftreten kann, wird das Parsen in einen `try-catch`-Block eingefaßt.
6. Das eigentliche Scannen findet nun in den Zeilen 24 – 31 statt:

```

1  import java.util.*;
2
3  %%
4
5  %class Scanner
6  %standalone
7
8  %unicode
9
10 %{\
11     List<Token> mTokenList = new ArrayList<Token>();
12 %}
13
14 %eof{
15     ExprParser parser = new ExprParser(mTokenList);
16     try {
17         Integer    result = parser.parseExpr();
18         System.out.println("Ergebnis: " + result);
19     } catch (ParseError e) {}
20 %eof}
21
22 %%
23
24 [1-9][0-9]* { mTokenList.add(new NumberToken(yytext())); }
25 "+"        { mTokenList.add(new Token("+"));           }
26 "-"        { mTokenList.add(new Token("-"));           }
27 "*"        { mTokenList.add(new Token("*"));           }
28 "/"        { mTokenList.add(new Token("/"));           }
29 "("        { mTokenList.add(new Token("("));           }
30 ")"        { mTokenList.add(new Token(")"));           }
31 [ \t\n\r]  { /* skip */                                }
32 .          { System.out.println("Unexpected char: \" +
33                                     yytext() + "\"");    }

```

Abbildung 6.6: *JFlex*-Spezifikation des Scanners.

- (a) In Zeile 24 erkennen wir natürliche Zahlen,
- (b) in den Zeilen 25 – 30 werden Operatoren und Klammer-Symbole erkannt,
- (c) in Zeile 31 werden Leerzeichen, Tabulatoren und Zeilenumbrüche aus der Eingabe herausgefiltert und
- (d) alle weiteren Zeichen werden in Zeile 32 als Fehler gemeldet.

Die einzelnen Token, die vom Scanner erkannt werden, sind Objekte der Klasse `Token`, die in Abbildung 6.7 gezeigt ist. Die Member-Variable `mImage` speichert für jedes Token den Text, der diesem Token zugeordnet ist. Falls es sich bei dem Token um eine Zahl handelt, wird zusätzlich noch der Wert der Zahl abgespeichert.

Jetzt können wir uns dem eigentlichen Parser zuwenden, dessen Implementierung in den Abbildungen 6.8 und 6.9 auf den Seiten 82 und 83 gezeigt ist.

1. Die Klasse `ExprParser` enthält zwei Member-Variablen:

- (a) `mTokenList` ist die Liste der Token, die vom Scanner geliefert werden.

```

1  public class Token {
2      private String mImage;
3
4      public Token(String image) {
5          mImage = image;
6      }
7      public String getImage() { return mImage; }
8  };
9
10 class NumberToken extends Token {
11     private Integer mNumber;
12
13     NumberToken(String number) {
14         super(number);
15         mNumber = new Integer(number);
16     }
17     Integer getNumber() { return mNumber; }
18 }

```

Abbildung 6.7: Die Klasse `Token`.

- (b) `mIndex` gibt den Index des nächsten zu verarbeitenden Tokens an. Jedesmal, wenn ein Token vom Parser erkannt wurde, wird dieser Index inkrementiert. Ist das Parsen erfolgreich, so zeigt der Index am Ende auf das Listenende.

Diese Variablen werden im Konstruktor initialisiert.

2. Die Methode `parseExpr()` implementiert die Grammatik-Regel

$$Expr \rightarrow Product \ ExprRest.$$

Daher wird zunächst versucht, ein *Product* zu erkennen und anschließend wird versucht, einen *ExprRest* zu erkennen.

- (a) Falls das Ende der Token-Liste erreicht ist, wird das berechnete Ergebnis zurückgegeben.
- (b) Das berechnete Ergebnis wird ebenfalls zurückgegeben, wenn als nächstes Token ein schließende Klammer “)” in der Eingabe gefunden wird.
- (c) In allen anderen Fällen liegt ein Syntax-Fehler vor, der eine Ausnahme auslöst. Diese Fälle werden in Zeile 21 – 23 abgeprüft.
- (d) Das `return`-Statement in Zeile 24 kann nie erreicht werden. Es ist nur vorhanden, weil der *Java*-Compiler sonst einen fehlenden Rückgabewert reklamieren würde.

3. Die Methode `parseExprRest()` implementiert die Grammatik-Regeln

$$\begin{array}{lcl}
 ExprRest & \rightarrow & "+" \ Product \ ExprRest \\
 & | & "-" \ Product \ ExprRest \\
 & | & \varepsilon
 \end{array}$$

Zunächst wird mit Hilfe der Methode `check()` überprüft, ob das nächste Token ein “+”- oder ein “-”-Zeichen ist. Falls es sich um keines dieser Zeichen handelt, kann nur noch die letzte Regel in Frage kommen.

```

1  import java.util.*;
2
3  public class ExprParser {
4      private List<Token> mTokenList; // list of all tokens
5      private int         mIndex;     // index to the next token
6
7      public ExprParser(List<Token> tokenList) {
8          mTokenList = tokenList;
9          mIndex     = 0;
10     }
11
12     public Integer parseExpr() throws ParseError {
13         Integer product = parseProduct();
14         Integer result  = parseExprRest(product);
15         if (mIndex == mTokenList.size()) {
16             return result;
17         }
18         if (mTokenList.get(mIndex).getImage().equals("(")) {
19             return result;
20         }
21         if (mIndex < mTokenList.size()) {
22             throw new ParseError("Parse Error");
23         }
24         return result;
25     }
26     private Integer parseExprRest(Integer sum) throws ParseError {
27         if (check("+")) {
28             Integer product = parseProduct();
29             return parseExprRest(sum + product);
30         }
31         if (check("-")) {
32             Integer product = parseProduct();
33             return parseExprRest(sum - product);
34         }
35         return sum;
36     }
37     private Integer parseProduct() throws ParseError {
38         Integer factor = parseFactor();
39         return parseProductRest(factor);
40     }

```

Abbildung 6.8: Top-Down-Parser für arithmetische Ausdrücke, 1. Teil

4. Analog implementieren die Methoden *parseProduct()* und *parseProductRest()* die folgenden Grammatik-Regeln:

$$\begin{array}{ll}
 \textit{Product} & \rightarrow \textit{Factor ProductRest} \\
 \textit{ProductRest} & \rightarrow \begin{array}{l} \text{"*"} \textit{Factor ProductRest} \\ \text{" /"} \textit{Factor ProductRest} \\ \varepsilon \end{array}
 \end{array}$$

```

41     private Integer parseProductRest(Integer product) throws ParseError {
42         if (check("*")) {
43             Integer factor = parseFactor();
44             return parseProductRest(product * factor);
45         }
46         if (check("/")) {
47             Integer factor = parseFactor();
48             return parseProductRest(product / factor);
49         }
50         return product;
51     }
52     private Integer parseFactor() throws ParseError {
53         if (check("(")) {
54             Integer expr = parseExpr();
55             if (!check(")")) {
56                 throw new ParseError("Parse Error, ')' expected");
57             }
58             return expr;
59         }
60         return parseNumber();
61     }
62     private boolean check(String token) {
63         if (mIndex == mTokenList.size()) {
64             return false;
65         }
66         if (mTokenList.get(mIndex).getImage().equals(token)) {
67             ++mIndex;
68             return true;
69         }
70         return false;
71     }
72     private Integer parseNumber() throws ParseError {
73         if (mIndex == mTokenList.size()) {
74             throw new ParseError("Parse Error, number expected");
75         }
76         Token token = mTokenList.get(mIndex);
77         if (!(token instanceof NumberToken)) {
78             throw new ParseError("Parse Error, number expected");
79         }
80         ++mIndex;
81         NumberToken numberToken = (NumberToken) token;
82         return numberToken.getNumber();
83     }
84 }

```

Abbildung 6.9: Top-Down-Parser für arithmetische Ausdrücke, 2. Teil.

5. Die Methode *parseFactor()* implementiert schließlich die Grammatik-Regeln

$$\begin{array}{lcl}
 \textit{Factor} & \rightarrow & \text{“(” Expr “)”} \\
 & | & \text{NUMBER}
 \end{array}$$

```

1  public class ParseError extends Exception {
2      public ParseError(String errorMsg) { super(errorMsg); }
3  }

```

Abbildung 6.10: Die Klasse `ParseError`.

6. Die Methode *check(o)* überprüft, ob das nächste zu lesende Token den Operator *o* darstellt. Dazu muss zunächst überprüft werden, ob überhaupt noch Token vorhanden sind, die bisher noch nicht verarbeitet wurden. Wenn dies der Fall ist und der Vergleich erfolgreich ist, wurde das Token erkannt und der Zähler `mIndex` wird inkrementiert, da das entsprechende Token nun erfolgreich gelesen wurde.
7. Die Methode *parseNumber* überprüft schließlich, ob das nächste zu lesende Token eine Zahl darstellt. dazu wird zunächst überprüft, ob überhaupt noch Token vorhanden sind. Ist dies der Fall, so wird der Typ des Tokens überprüft. Falls das Token kein `NumberToken` ist, wird eine Ausnahme ausgelöst. Ansonsten wird der Wert des Tokens als Zahl zurückgegeben.

Zum Abschluss zeigt die Abbildung 6.10 die Definition der Klasse `ParseError`. Diese Klasse wird benötigt um sinnvolle Fehlermeldungen erzeugen zu können.

Der soeben diskutierte Parser für arithmetische Ausdrücke ist in der vorliegenden Form für die Praxis noch ungeeignet, denn die Fehlermeldungen reichen nicht aus, um in komplexeren Ausdrücken einen Fehler tatsächlich lokalisieren zu können.

Wir werden später noch genauer analysieren, welche Bedingungen eine Grammatik erfüllen muss, damit es möglich ist, auf die oben dargestellte Weise einen Top-Down-Parser zu entwickeln.

6.2.3 Implementierung eines rückwärts arbeitenden Top-Down-Parser

Wir zeigen nun, wie wir die in Abbildung 6.4 gezeigte Grammatik ohne Entfernung der Links-Rekursion mit einem Top-Down-Parser implementieren können. Die entscheidende Idee ist hier, die Token-Liste rückwärts abzuarbeiten. Wir benutzen wieder den in Abbildung 6.6 gezeigten Scanner.

Die Abbildungen 6.11 und 6.12 zeigen die Implementierung des rückwärts-arbeitenden Top-Down-Parser für arithmetische Ausdrücke.

1. Die Klasse `ExprParser` enthält die Member-Variablen `mTokenList` und `mIndex`. Erstere ist die Liste alle Token, Letztere bezeichnet den Index des Tokens, was als nächstes untersucht wird. Da die Liste der Token nun von hinten nach vorne abgearbeitet wird, wird diese Variable in Zeile 9 so initialisiert, dass sie auf das letzte gelesen Token zeigt. Da in Java Felder mit mit 0 beginnen, hat der Index des letzten gelesenen Tokens den Wert `mTokenList.size() - 1`.
2. Die drei Grammatik-Regeln

$$\begin{array}{lcl}
 \textit{Expr} & \rightarrow & \textit{Expr} \text{ "+" } \textit{Product} \\
 & | & \textit{Expr} \text{ "-" } \textit{Product} \\
 & | & \textit{Product}
 \end{array}$$

werden durch die Methode *parseExpr()* implementiert. Diese Regeln zeigen, dass eine *Expr* auf jeden Fall mit einem *Product* endet. Daher parsen wir in Zeile 12 zunächst ein *Product* und speichern den Wert dieses Produktes in der Variablen `product`. Falls vor dem Produkt ein "+"- oder ein "-"-Zeichen kommt, lesen wir dieses Zeichen und versuchen danach durch einen rekursiven Aufruf der Methode *parseExpr()*, den Ausdruck, der vor diesem Zeichen steht, zu parsen. Der dabei erhaltene Wert wird in der Variablen `expression` gespeichert.

```

1  public class ExprParser {
2      private List<Token> mTokenList; // list of all tokens
3      private int         mIndex;      // index of next token to consume
4
5      public ExprParser(List<Token> tokenList) {
6          mTokenList = tokenList;
7          mIndex      = mTokenList.size() - 1;
8      }
9      public Integer parseExpr() throws ParseError {
10         Integer product = parseProduct();
11         if (check("+")) {
12             Integer expression = parseExpr();
13             return expression + product;
14         }
15         if (check("-")) {
16             Integer expression = parseExpr();
17             return expression - product;
18         }
19         return product;
20     }
21     private Integer parseProduct() throws ParseError {
22         Integer factor = parseFactor();
23         if (check("*")) {
24             Integer product = parseProduct();
25             return product * factor;
26         }
27         if (check("/")) {
28             Integer product = parseProduct();
29             return product / factor;
30         }
31         return factor;
32     }
33     private Integer parseFactor() throws ParseError {
34         if (check("(")) {
35             Integer expression = parseExpr();
36             if (!check("(")) {
37                 throw new ParseError("Parse Error, '(' expected");
38             }
39             return expression;
40         }
41         return parseNumber();
42     }

```

Abbildung 6.11: Ein rückwärts-arbeitender Top-Down-Parser für arithmetische Ausdrücke, 1. Teil.

Abhängig davon, ob wir ein “+”- oder “-”-Zeichen gelesen haben, geben wir dann die Summe `expression + product` oder die Differenz `expression - product` als Ergebnis zurück.

Falls vor dem ersten Produkt kein “+”- oder “-”-Zeichen steht, besteht der Ausdruck nur aus einem Produkt und wir geben den Wert dieses Produktes zurück.

```

43     private boolean check(String token) {
44         if (mIndex < 0) {
45             return false; // all tokens consumed
46         }
47         if (mTokenList.get(mIndex).getImage().equals(token)) {
48             --mIndex;
49             return true;
50         }
51         return false;
52     }
53     private Integer parseNumber() throws ParseError {
54         Token token = mTokenList.get(mIndex);
55         if (!(token instanceof NumberToken)) {
56             throw new ParseError("Parse Error, number expected");
57         }
58         --mIndex;
59         NumberToken numberToken = (NumberToken) token;
60         return numberToken.getNumber();
61     }
62 }

```

Abbildung 6.12: Ein rückwärts-arbeitender Top-Down-Parser für arithmetische Ausdrücke, 2. Teil.

3. Die drei Grammatik-Regeln

$$\begin{array}{lcl}
 \textit{Product} & \rightarrow & \textit{Product} \text{ ``*'' } \textit{Factor} \\
 & | & \textit{Product} \text{ ``/''' } \textit{Factor} \\
 & | & \textit{Factor}
 \end{array}$$

werden auf analoge Weise in der Methode *parseProduct()* umgesetzt.

4. Die Methode *parseFactor()* implementiert die beiden Regeln

$$\begin{array}{lcl}
 \textit{Factor} & \rightarrow & \text{ ``('' } \textit{Expr} \text{ ``)''} \\
 & | & \textit{Number}
 \end{array}$$

- Die Methode *check()* ändert sich gegenüber der ursprünglichen Implementierung, da die Token-Liste ja jetzt rückwärts abgearbeitet wird. Deswegen testen wir nun in Zeile 44, ob *mIndex* negativ ist, denn dann hätten wir alle Token abgearbeitet. Andernfalls vergleichen wir das durch *mIndex* spezifizierte Token mit dem String, der als Argument übergeben wird. Falls dieser Vergleich erfolgreich ist, wird der Zeiger *mIndex* nun dekrementiert.
- Die Methode *parseNumber()* überprüft, ob das Token, das an der durch den Zeiger *mIndex* spezifizierten Position steht, eine Zahl ist und gibt diese gegebenenfalls zurück.

6.2.4 Erzeugung und Auswertung eines Syntax-Baums

Die in den letzten beiden Abschnitten vorgestellten Parser haben unmittelbar eine Zahl als Ergebnis berechnet. Normalerweise berechnet der Parser zunächst nur einen abstrakten Syntax-Baum, der dann anschließend von einem weiteren Programm ausgewertet wird. Als Beispiel werden wir einen Parser für arithmetische Ausdrücke entwickeln, der diese zunächst als abstrakte Syntax-Bäume abspeichert. Dadurch können wir dann ein Programm entwickeln, dass arithmetische Ausdrücke symbolisch nach einer Variablen differenziert. Abbildung 6.13 zeigt eine Grammatik, die gegenüber der ursprünglichen Grammatik aus Abbildung 6.4 so erweitert worden ist, dass nun auch Funktionsaufrufe der Form

$$\exp(t) \quad \text{und} \quad \ln(t)$$

für einen beliebigen Ausdruck t zugelassen sind. Außerdem können arithmetische Ausdrücke nun auch Variablen enthalten.

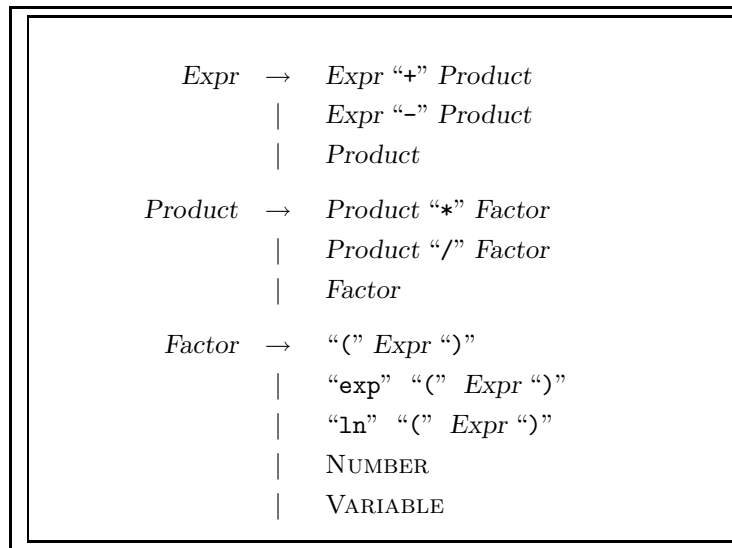


Abbildung 6.13: Erweiterte Grammatik für arithmetische Ausdrücke.

Als erstes entwickeln wir wieder einen Scanner für die so erweiterte Syntax. Abbildung 6.14 zeigt einen *JFlex*-Scanner. Gegenüber dem ersten in Abbildung 6.6 gezeigten Scanner gibt es die folgenden Unterschiede:

1. Der Aufruf der Methode `parseExpr()` in Zeile 18 liefert nun keine Zahl sondern einen abstrakten Syntax-Baum als Ergebnis zurück. Solche Syntax-Bäume werden wir durch die abstrakte Klasse `Expr` und die davon abgeleiteten Klassen modellieren. Wir werden diese Klassen später noch im Detail diskutieren.
2. Die Klasse `Expr` deklariert eine Methode `diff(x)`, die als Argument einen String x erhält, der als Variable interpretiert wird, nach welcher der Ausdruck abgeleitet werden soll. Abbildung 6.15 zeigt die Klasse `Expr`.
3. In Zeile 29 und 30 erkennen wir die Schlüsselwörter `"exp"` und `"ln"`.
4. In Zeile 31 erkennen wir einzelne Buchstaben als Variablen. Wir haben von der Klasse `Token` nun zusätzlich die in Abbildung 6.16 gezeigte Klasse `VariableToken` abgeleitet. Diese neue Klasse dient dazu, solche Token zu repräsentieren, die Variablen darstellen.

Zur Darstellung eines abstrakten Syntax-Baums leiten wir von der Klasse `Expr` die folgenden Klassen ab:

```

1  import java.util.*;
2
3  %%
4
5  %class Scanner
6  %standalone
7
8  %unicode
9
10 %{\
11     List<Token> mTokenList = new ArrayList<Token>();
12 %}
13
14 %eof{
15     System.out.println("Length: " + mTokenList.size());
16     ExprParser parser = new ExprParser(mTokenList);
17     try {
18         Expr expr      = parser.parseExpr();
19         Expr derivation = expr.diff("x");
20         System.out.println("d " + expr + " / dx = " + derivation);
21     } catch (ParseError e) {
22         System.out.println(e);
23     }
24 %eof}
25
26 %%
27
28 [1-9][0-9]* { mTokenList.add(new NumberToken(yytext())); }
29 "exp"       { mTokenList.add(new Token("exp"));           }
30 "ln"        { mTokenList.add(new Token("ln"));            }
31 [a-z]       { mTokenList.add(new VariableToken(yytext())); }
32 "+"         { mTokenList.add(new Token("+"));             }
33 "-"         { mTokenList.add(new Token("-"));             }
34 "*"         { mTokenList.add(new Token("*"));             }
35 "/"         { mTokenList.add(new Token("/"));             }
36 "("         { mTokenList.add(new Token("("));             }
37 ")"         { mTokenList.add(new Token(")"));             }
38 [ \t\n\r]   { /* skip */                                  }
39 .           { System.out.println("Unexpected char: \"" +
40                                     yytext() + "\"");      }

```

Abbildung 6.14: Scanner für erweiterte arithmetische Ausdrücke

```

1  public abstract class Expr {
2      public abstract Expr diff(String var);
3  }

```

Abbildung 6.15: Die abstrakte Klasse *Expr*.

1. Die Klasse *Sum* repräsentiert eine Summe.

```

1  class VariableToken extends Token {
2
3      VariableToken(String number) {
4          super(number);
5      }
6  }

```

Abbildung 6.16: Die Klasse *VariableToken*.

2. Die Klasse *Difference* repräsentiert eine Differenz.
3. Die Klasse *Product* repräsentiert ein Produkt.
4. Die Klasse *Quotient* repräsentiert einen Quotienten.
5. Die Klasse *Exponential* repräsentiert einen Aufruf der Exponential-Funktion $\exp()$.
6. Die Klasse *Logarithm* repräsentiert einen Aufruf der Logarithmus-Funktion $\ln()$.
7. Die Klasse *Number* repräsentiert eine natürliche Zahl.
8. Die Klasse *Variable* repräsentiert eine Variable.

```

1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Sum(mLhs.diff(x), mRhs.diff(x));
11     }
12     public String toString() {
13         return "(" + mLhs.toString() + " + " + mRhs.toString() + ")";
14     }
15 }

```

Abbildung 6.17: Die Klasse *Sum* zur Darstellung von Summen.

Wir diskutieren die in Abbildung 6.17 gezeigte Klasse *Sum* nun im Detail.

1. Die Klasse *Sum* stellt eine Summe der Form

$$lhs + rhs$$

dar. Die in Zeile 2 und 3 deklarierten Member-Variablen *mLhs* und *mRhs* repräsentieren die Summanden *lhs* und *rhs*.

2. Der Konstruktor initialisiert die Member-Variablen *mLhs* und *mRhs*.
3. Da die Ableitung einer Summe $f+g$ nach einer Variablen x durch die Summe der Ableitungen gegeben ist, denn es gilt

$$\frac{d}{dx}(f + g) = \frac{d}{dx}f + \frac{d}{dx}g,$$

können wir die Methode *diff* dadurch implementieren, dass wir die Summanden einzeln ableiten und die Ableitungen zu einer neuen Summe zusammensetzen.

4. Schließlich enthält die Klasse eine Methode *toString()* mit der die Summe ausgegeben werden kann. Beachten Sie, dass das Ergebnis in Klammern gesetzt wird. Dies ist deshalb notwendig, weil die Summe ja Teil eines komplexen Ausdrucks sein kann, beispielsweise Teil eines Produktes. Beispielsweise ist die Summe $2 + 3$ Teil des Produktes $(2 + 3) * 4$ und hier müssen offenbar Klammern gesetzt werden.

Als nächstes diskutieren wir die in Abbildung 6.18 gezeigte Klasse *Product*.

1. Die Klasse *Product* stellt ein Produkt der Form

$$lhs * rhs$$

dar. Die in Zeile 2 und 3 deklarierten Member-Variablen *mLhs* und *mRhs* repräsentieren die Faktoren *lhs* und *rhs*.

2. Der Konstruktor initialisiert die Member-Variablen *mLhs* und *mRhs*.
3. Die Ableitung eines Produktes $f * g$ nach einer Variablen x ist durch die Produkt-Regel gegeben, es gilt

$$\frac{d}{dx}(f * g) = \frac{d}{dx}f * g + f * \frac{d}{dx}g.$$

Wir leiten also zunächst die beiden Faktoren einzeln ab und setzen das Ergebnis entsprechend der Produkt-Regel aus diesen Ableitungen zusammen.

4. Schließlich enthält die Klasse eine Methode *toString()* mit der das Produkt ausgegeben werden kann. Auch hier ist es notwendig Klammern zu setzen, denn sonst würde beispielsweise ein Ausdruck der Form $x/(y * z)$ falsch interpretiert.

Die Implementierung der restlichen Klassen verläuft analog und wird aus Platzgründen nicht weiter diskutiert.

```

1  public class Product extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Product(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         Expr lhsDiff = mLhs.diff(x);
11         Expr rhsDiff = mRhs.diff(x);
12         return new Sum(new Product(lhsDiff, mRhs),
13                        new Product(mLhs, rhsDiff));
14     }
15     public String toString() {
16         return "(" + mLhs.toString() + " * " + mRhs.toString() + ")";
17     }
18 }

```

Abbildung 6.18: Die Klasse *Product* zur Darstellung von Produkten.

```

1  import java.util.*;
2
3  public class ExprParser {
4      private List<Token> mTokenList; // list of all tokens
5      private int         mIndex;      // index to the next token
6
7      public ExprParser(List<Token> tokenList) {
8          mTokenList = tokenList;
9          mIndex      = mTokenList.size() - 1;
10     }
11     public Expr parseExpr() throws ParseError {
12         Expr product = parseProduct();
13         if (check("+")) {
14             Expr expression = parseExpr();
15             return new Sum(expression, product);
16         }
17         if (check("-")) {
18             Expr expression = parseExpr();
19             return new Difference(expression, product);
20         }
21         return product;
22     }
23     private Expr parseProduct() throws ParseError {
24         Expr factor = parseFactor();
25         if (check("*")) {
26             Expr product = parseProduct();
27             return new Product(product, factor);
28         }
29         if (check("/")) {
30             Expr product = parseProduct();
31             return new Quotient(product, factor);
32         }
33         return factor;
34     }

```

Abbildung 6.19: Implementierung der Klasse *ExprParser*, 1. Teil.

In den Abbildungen 6.19, 6.20 und 6.21 zeigen wir schließlich die Implementierung des eigentlichen Parsers. Der Parser ist ein rückwärts arbeitender Top-Down-Parser, der analog zu den in den Abbildungen 6.11 und 6.12 gezeigten Parser arbeitet. Es gibt zwei wesentliche Unterschiede:

1. Da die Aufgabe des Parsers jetzt darin besteht, einen abstrakten Syntax-Baum zu erstellen, ist der Rückgabewert der verschiedenen Parse-Methoden nun nicht mehr **Integer** sondern **Expr**. Dementsprechend werden die Ergebnisse durch Aufrufe der entsprechenden Konstruktor-Methoden der von der Klasse **Expr** abgeleiteten konkreten Methoden berechnet.
2. Da die Grammatik nun auch Exponentialfunktion und Logarithmus sowie Variablen zulässt, wurde die Methode *parseFactor()* entsprechend geändert.

Historisches Die Sprache ALGOL [Bac59, NBB⁺60] war die erste Programmier-Sprache, deren Syntax auf einer kontextfreien Grammatik basiert.

```

35     private Expr parseFactor() throws ParseError {
36         if (check("(")) {
37             Expr expression = parseExpr();
38             if (!check("(")) {
39                 throw new ParseError("Parse Error, '(' expected");
40             }
41             if (check("exp")) {
42                 return new Exponential(expression);
43             }
44             if (check("ln")) {
45                 return new Logarithm(expression);
46             }
47             return expression;
48         }
49         if (checkNumber()) {
50             return parseNumber();
51         }
52         return parseVariable();
53     }
54
55     private Expr parseNumber() {
56         Token token = mTokenList.get(mIndex);
57         --mIndex;
58         NumberToken numberToken = (NumberToken) token;
59         return new Number(numberToken.getNumber());
60     }
61
62     private Expr parseVariable() {
63         Token token = mTokenList.get(mIndex);
64         --mIndex;
65         VariableToken varToken = (VariableToken) token;
66         return new Variable(varToken.getImage());
67     }

```

Abbildung 6.20: Implementierung der Klasse *ExprParser*, 2. Teil.

```

68     private boolean check(String token) {
69         if (mIndex < 0) {
70             return false; // all tokens consumed
71         }
72         if (mTokenList.get(mIndex).getImage().equals(token)) {
73             --mIndex;
74             return true;
75         }
76         return false;
77     }
78
79     private boolean checkNumber() {
80         if (mIndex < 0) {
81             return false; // all tokens consumed
82         }
83         if (mTokenList.get(mIndex) instanceof NumberToken) {
84             --mIndex;
85             return true;
86         }
87         return false;
88     }
89 }

```

Abbildung 6.21: Implementierung der Klasse *ExprParser*, 3. Teil.

Kapitel 7

Antlr — *Another Tool for Language Recognition*

Es gibt eine Reihe von Werkzeugen, die in der Lage sind, einen Top-Down-Parser automatisch zu erzeugen. Das Werkzeug, was vermutlich am weitesten verbreitet ist, trägt den Namen ANTLR, was für *another tool for language recognition* steht. Sie finden dieses Werkzeug im Internet unter der folgenden Adresse

<http://www.antlr.org>

Es fehlt uns die Zeit, um ANTLR vollständig zu diskutieren, dafür gibt es beispielsweise das Buch von Terence Parr [Par07], der auch der Entwickler ist. Stattdessen führen wir ANTLR an Hand verschiedener Beispiele ein, an Hand deren wir die wichtigsten Eigenschaften des Werkzeugs darstellen können.

7.1 Ein Parser für arithmetische Ausdrücke

Wir beginnen mit einem Parser für arithmetische Ausdrücke, die entsprechend der in Abbildung 6.4 auf Seite 78 gezeigten Grammatik aufgebaut sind. Nun ist diese Grammatik linksrekursiv und für einen Top-Down-Parser (und solche werden von ANTLR generiert) in der vorliegenden Form nicht geeignet. Im letzten Kapitel hatten wir diskutiert, wie die Links-Rekursion aus dieser Grammatik beseitigt werden kann und dabei die in Abbildung 6.5 auf Seite 78 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 nun auch die Postfix-Operatoren “*”, “+” und “?” verwendet werden dürfen. Die Bedeutung dieser Operatoren ist die gleiche wie bei regulären Ausdrücken. Zusätzlich können die rechten Seiten einer Grammatik-Regel durch Verwendung von Klammern strukturiert werden und der Infix-Operator “|”, der für eine Auswahl zwischen zwei Alternativen steht, ist ebenfalls zugelassen. Damit schreibt sich dann die Grammatik für arithmetische Ausdrücke wie in Abbildung 7.1 auf Seite 95 gezeigt.

Die in der Abbildung gezeigte Grammatik ist wir folgt zu lesen:

1. Ein arithmetischer Ausdruck startet mit einem Produkt, auf das dann eine Liste weiterer Produkte folgen kann, die entweder durch eine Minus- oder ein Plus-Zeichen abgetrennt sind. Damit hat ein arithmetischer Ausdruck also die Form

$$p_1 \pm p_2 \pm \dots \pm p_n.$$

2. Analog ist ein Produkt eine Liste von Faktoren, die durch die Zeichen “*” oder “/” voneinander getrennt sind.

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

Abbildung 7.1: Erweiterte Grammatik für arithmetische Ausdrücke.

3. Ein Faktor ist entweder ein geklammerter arithmetischer Ausdruck, oder es ist eine einfache Zahl.

```

1  grammar expr;
2
3  expr      : product (('+'|'-') product)* ;
4
5  product   : factor (('*'|'/') factor)* ;
6
7  factor    : '(' expr ')';
8           | NUMBER
9           ;
10
11 NUMBER : ('0'..'9')+;
12 WS    : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 7.2: ANTLR-Spezifikation eines Parsers für arithmetische Ausdrücke.

Diese Grammatik läßt sich nun mit Hilfe von ANTLR wie in Abbildung 7.2 gezeigt implementieren. Wir diskutieren diese Umsetzung Zeile für Zeile.

1. In Zeile 1 geben wir mit dem Schlüsselwort **grammar** an, dass **expr** das Start-Symbol unserer Grammatik ist.
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,¹ während syntaktische Variablen immer mit einem kleinen Buchstaben beginnen. Daher wissen wir, dass der String “NUMBER” in Zeile 8 ein Terminal bezeichnet.

¹Es ist Konvention, aber nicht vorgeschrieben, dass die Namen von Terminalen nur aus Großbuchstaben bestehen.

5. In Zeile 11 wird die lexikalische Struktur der mit **NUMBER** bezeichneten Terminale durch einen regulären Ausdruck definiert. Der reguläre Ausdruck

`('0'..'9')+`

steht hier für eine Folge aus Ziffern, die mindestens die Länge 1 hat. In *Flex* hätten wir statt dessen den regulären Ausdruck

`[0-9]+`

verwendet.

6. Zeile 12 definiert schließlich das Token **WS**, wobei der Name als Abkürzung für *white space* gemeint 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 aufgerufen 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.

Um aus der in Abbildung 7.2 gezeigten Grammatik einen Parser erzeugen zu können, übersetzen wir diese Datei mit dem folgenden Befehl:

```
java org.antlr.Tool expr.g
```

Das setzt natürlich voraus, dass die Umgebungs-Variable **CLASSPATH** so gesetzt ist, dass das Paket `org.antlr.Tool` auch gefunden wird. Die Ausführung dieses Befehls 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 höchstens dann zurückgreifen müssen, wenn wir die Funktionsweise des erzeugten Parsers analysieren möchten.

4. `expr_*.g`

Hier wird von ANTLR eine Spezifikation aller verwendeten Token erstellt, die für uns nicht weiter interessant ist.

Um den erzeugten Parser aufrufen zu können, benötigen wir noch ein Treiber-Programm. Abbildung 7.3 zeigt ein solches Programm.

1. In Zeile 1 importieren wir das Paket `org.antlr.runtime`.
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. Den Scanner wandeln wir in Zeile 8 in einen `TokenStream` und aus diesem können wir dann in Zeile 9 einen Parser erzeugen.
4. Der Aufruf des Parsers erfolgt in Zeile 10 indem wir den Namen des Start-Symbols als Methode verwenden.
5. Da die verschiedenen Konstruktoren diverse Ausnahmen auslösen können, deklarieren wir in Zeile 5, dass die Methode `main` eine Ausnahme auslösen kann.

```

1  import org.antlr.runtime.*;
2
3  public class ParseExpr {
4
5      public static void main(String[] args) throws Exception {
6          ANTLRInputStream input = new ANTLRInputStream(System.in);
7          exprLexer          lexer = new exprLexer(input);
8          CommonTokenStream ts   = new CommonTokenStream(lexer);
9          exprParser          parser = new exprParser(ts);
10         parser.expr();
11     }
12 }

```

Abbildung 7.3: Treiber-Programm für den von ANTLR erzeugten Parser.

Übersetzen wir dieses Programm, so können wir den Parser durch den Befehl

```
echo "2 * 3 + (5 - 4) / 2" | java ParseExpr
```

testen. Dieser Befehl liefert keinerlei Ausgabe und zeigt lediglich, dass der Ausdruck mit der angegebenen Grammatik auch erkannt werden konnte. Geben wir statt dessen den Befehl

```
echo "2 * + 3" | java ParseExpr
```

ein, so erhalten wir die Fehlermeldung:

```
line 1:4 no viable alternative at input '+'
```

Diese Fehlermeldung sagt aus, dass es in Zeile 1 in der vierten Spalte (Spalten werden mit 0 beginnend gezählt) ein Problem gibt, denn der Parser kann dort mit dem Zeichen “+” nichts anfangen.

7.2 Ein Parser zur Auswertung arithmetischer Ausdrücke

Das letzte Beispiel ist noch nicht sehr spektakulär, weil die vom Parser erkannten Ausdrücke nicht ausgewertet werden. Wir präsentieren jetzt ein komplexeres Beispiel, bei dem arithmetische Ausdrücke ausgewertet und die erhaltenen Ergebnisse in Variablen gespeichert werden können. Wir gehen dabei in zwei Schritten vor und präsentieren zunächst eine reine Grammatik in ANTLR-Notation. Anschließend erweitern wir diese mit Aktionen, in denen die Ausdrücke ausgewertet werden können.

Abbildung 7.4 zeigt diese Grammatik. Gegenüber der vorher gezeigten Grammatik für arithmetische Ausdrücke gibt es die folgenden Erweiterungen:

1. 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. **stmt** 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:

```

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'..'9')+;
20 WS : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 7.4: Eine Grammatik für die Auswertung von Ausdrücken.

```

x = 2 * 3; y = 4 * 5; z = x * x + y * y;
z / 3;

```

Wir wollen nun einen ganz einfachen Interpreter entwickeln, der eine Folge von solchen Zuweisungen auswertet und die bei der Auswertung berechneten Zwischen-Ergebnisse in Variablen abspeichert, auf die in folgenden Ausdrücken Bezug genommen werden kann. Weiterhin soll jeder Ausdruck, der keiner Variablen zugewiesen wird, ausgewertet und ausgegeben werden. Abbildung 7.5 zeigt die Realisierung eines solchen Interpreters mit ANTLR.

1. Da wir die Werte der einzelnen Variablen in einer Tabelle abspeichern müssen, importieren wir in den Zeilen 3 – 5 die Klasse `java.util.TreeMap`, denn diese Klasse implementiert Tabellen effizient als binäre Bäume.

Allgemein setzt ANTLR all den Code, der durch das Schlüsselwort “**@header**” spezifiziert wird, an den Anfang der erstellten Parser-Datei.

2. In den Zeilen 7 – 9 definieren wir zusätzliche Member-Variablen für die erzeugte Klasse **programParser**. In unserem Fall definieren wir hier die Tabelle, die später die Werte der Variablen enthält als Abbildung, die Strings ganze Zahlen zuordnet.

Allgemein setzt ANTLR all den Code, der durch das Schlüsselwort “**@members**” spezifiziert wird, an den Anfang der erstellten Parser-Datei. 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 geschweiften Klammern eingefaßt sind.

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

```

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'..'9')+;
40 WS : (' '\t'|\n'|\r') { skip(); };

```

Abbildung 7.5: Ein Interpreter zur Auswertung von Ausdrücken.

$stmtnt \rightarrow ID '=' expr ';'$

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.

4. in Zeile 19 definieren wir mit der Zeile

`expr returns [int result]`

dass die Methode, die eine *expr* parst, als Ergebnis ein `int` zurück gibt. Auf dieses Ergebnis können wir mit dem Namen `result` zurückgreifen, was wir in Ausdrücken wie `$expr.result` ja auch schon getan haben.

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 Berechnung zuweisen.

6. In den Zeilen 36 und 37 greifen wir auf die Strings, die den Token `ID` und `INT` entsprechen, mit Hilfe der für Token vordefinierten Variable `text` zurück.

7.3 Erzeugung abstrakter Syntax-Bäume

Bei der Auswertung arithmetischer Ausdrücke im letzten Abschnitt hatten wir Glück und konnten das Ergebnis eines Ausdrucks unmittelbar mit Hilfe von semantischen Aktionen berechnen. Bei komplexeren Problemen ist es in der Regel erforderlich, zunächst einen abstrakten Syntax-Baum zu erzeugen. Die eigentliche Berechnung findet dann erst nach dem Parsen auf dem Syntax-Baum statt. Wir wollen dieses Verfahren an einem Beispiel demonstrieren. Bei dem Beispiel geht es um die symbolische Differentiation arithmetischer Ausdrücke. Ist beispielsweise der arithmetische Ausdruck

$$x * \ln(x)$$

gegeben, so findet sich für die Ableitung dieses Ausdrucks nach der Variable *x* mit Hilfe der Produkt-Regel das Ergebnis

$$1 * \ln(x) + x * \frac{1}{x}.$$

Da die arithmetischen Ausdrücke nun zusätzlich zu den Operatoren, die die vier Grundrechenarten beschreiben, auch noch Funktionszeichen für die Exponential-Funktion und den natürlichen Logarithmus enthalten sollen, müssen wir die Grammatik aus dem letzten Abschnitt erweitern. Abbildung 7.6 zeigt die entsprechend erweiterte Grammatik. Das neu hinzugekommene Terminal `VARIABLE` steht nun für eine Variable wie beispielsweise “*x*”.

<i>Expr</i>	→	<i>Product</i> ((“+” “-”) <i>Product</i>) *
<i>Product</i>	→	<i>Factor</i> ((“*” “/”) <i>Factor</i>) *
<i>Factor</i>	→	“(” <i>Expr</i> “)”
		“exp” “(” <i>Expr</i> “)”
		“log” “(” <i>Expr</i> “)”
		VARIABLE
		NUMBER

Abbildung 7.6: EBNF-Grammatik für arithmetische Ausdrücke mit Exponential-Funktion und Logarithmus.

7.3.1 Implementierung des Parsers

Abbildung 7.7 zeigt die ANTLR-Implementierung der Grammatik aus Abbildung 7.6. Wir verwenden zur Darstellung der abstrakten Syntax-Bäume die selben Klassen, die wir schon im vorigen Kapitel vorgestellt hatten.

```
1  grammar expr;
2
3  expr returns [Expr result]
4      : p = product { $result = $p.result; }
5      | ('+' q = product) { $result = new Sum($result, $q.result); }
6      | ('-' q = product) { $result = new Difference($result, $q.result); }
7      )*
8      ;
9
10 product returns [Expr result]
11     : f = factor { $result = $f.result; }
12     (
13         ('*' g = factor) { $result = new Product($result, $g.result); }
14         | ('/' g = factor) { $result = new Quotient($result, $g.result); }
15     )*
16     ;
17
18 factor returns [Expr result]
19     : '(' expr ')' { $result = $expr.result; }
20     | 'exp' '(' expr ')' { $result = new Exponential($expr.result); }
21     | 'log' '(' expr ')' { $result = new Logarithm($expr.result); }
22     | VAR { $result = new Variable($VAR.text); }
23     | NUM { $result = new Number($NUM.text); }
24     ;
25
26 VAR : ('a'..'z'|'A'..'Z')+;
27 NUM : ('0'..'9')+;
28 WS  : (' '|'\t'|\n'|\r') { skip(); };
```

Abbildung 7.7: Die ANTLR-Spezifikation der Grammatik.

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 7.8 noch die Einbindung des Parsers. Gegenüber dem in Abbildung 7.3 gezeigten Treiber gibt es nur einen wesentlichen Unterschied: In Zeile 9 wird nun ein Objekt der Klasse *Expr* erzeugt. Für dieses Objekt rufen wir dann in Zeile 10 die Methode *diff()* auf, die die symbolische Ableitung berechnet.

```
1  import org.antlr.runtime.*;
2
3  public class Differentiate {
4      public static void main(String[] args) throws Exception {
5          ANTLRInputStream input = new ANTLRInputStream(System.in);
6          ExprLexer lexer = new ExprLexer(input);
7          CommonTokenStream ts = new CommonTokenStream(lexer);
8          ExprParser parser = new ExprParser(ts);
9          Expr expr = parser.expr();
10         Expr diff = expr.diff("x");
11         System.out.println("d (" + expr + ")/dx = " + diff);
12     }
13 }
```

Abbildung 7.8: Ein Treiber für den Parser.

Aufgabe 15: Auf meiner Webseite finden Sie unter

[http://www.lehre.dhbw-stuttgart.de/
~stroetma/Formale-Sprachen/Aufgaben/Grammar2HTML/c-grammar.g](http://www.lehre.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/](http://www.lehre.dhbw-stuttgart.de/~stroetma/Formale-Sprachen/Aufgaben/Grammar2HTML/)

verschiedene Klassen, deren *toString*-Methode HTML-Ausgabe erzeugt.

Kapitel 8

Bison

Wir wollen in diesem Kapitel einen ersten Einblick in die Verwendung des Parser-Generators *Bison* geben. Die von *Bison* erzeugten Parser sind sogenannte *LALR(1)-Parser*. Die Theorie dieser Parser wurde von Franklin DeRemer [DeR71, DP82] entwickelt und baut auf der Theorie der *LR(k)-Parser* auf, die von Donald Knuth [Knu65] begründet wurde. Da diese Theorie wesentlich komplexer ist als die Theorie der *LL(1)-Parser*, können wir diese Theorie erst in einem später Kapitel darstellen und müssen uns in diesem Kapitel mit einer oberflächlichen Betrachtung von *Bison* begnügen. Dieses Vorgehen funktioniert, weil die Theorie der LALR(1)-Parser solange unwichtig ist, wie die zu parsende Grammatik die LALR(1)-Eigenschaft hat. Bei den einfachen Beispielen, die wir in diesem Kapitel betrachten werden, ist dies der Fall. Für die Praxis ist ein LALR(1)-Parser jedoch oft dann zu schwach, wenn die Sprache vorgegeben ist und nicht mehr verändert werden kann. Falls Sie selber eine Sprache entwerfen wollen, so können Sie durch das Einfügen geeigneter Schlüsselwörter immer erreichen, dass die Sprachen die LALR(1)-Eigenschaft besitzt.

Unser Ziel in diesem Abschnitt ist es, für die in Abbildung 8.1 gezeigte Grammatik einen Parser mit Hilfe des Parser-Generators *Bison* zu erstellen. Die von *Bison* erstellten Parser sind wahlweise C- oder C++-Programme. Diese Grammatik ermöglicht es, einen arithmetischen Ausdruck auszuwerten und das Ergebnis dieser Auswertung einer Variablen zuzuweisen. Dabei können die arithmetischen Ausdrücke neben den Operatoren, die für die Grundrechenarten stehen, auch den Aufruf unärer Funktionen, wie der Funktion *sqrt()* zur Berechnung der Quadratwurzel, enthalten. Mit Hilfe dieser Grammatik werden wir einen Kommando-Zeilen-basierten Taschenrechner entwickeln.

8.1 Der Scanner

Bevor mit der Entwicklung des Parsers beginnen können, benötigen wir einen Scanner, der die Eingabe in unterschiedliche Token zerlegt. Der Parser-Generator *Bison* arbeitet mit dem Scanner-Generator *Flex* zusammen. Abbildung 8.2 zeigt die *Flex*-Spezifikation eines Scanners, der für unsere Zwecke geeignet ist. Der Scanner gibt vier verschiedene Typen von Token zurück.

1. Der Typ **NUMBER** steht für Fließkomma-Zahlen.
2. Der Typ **NAME** steht für den Namen einer Variablen.
3. Der Typ **FUNC** steht für den Namen einer unären Funktion.

Syntaktisch besteht kein Unterschied zwischen **NAME** und **FUNC**, denn beide bestehen aus einer Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt. Der Scanner trifft die Unterscheidung an Hand einer Tabelle, in der alle vordefinierten Funktions-Namen eingetragen sind.

4. Schließlich gibt es noch die Operator- und Klammer-Symbole, die vom Scanner als einzelne Zeichen zurück gegeben werden.

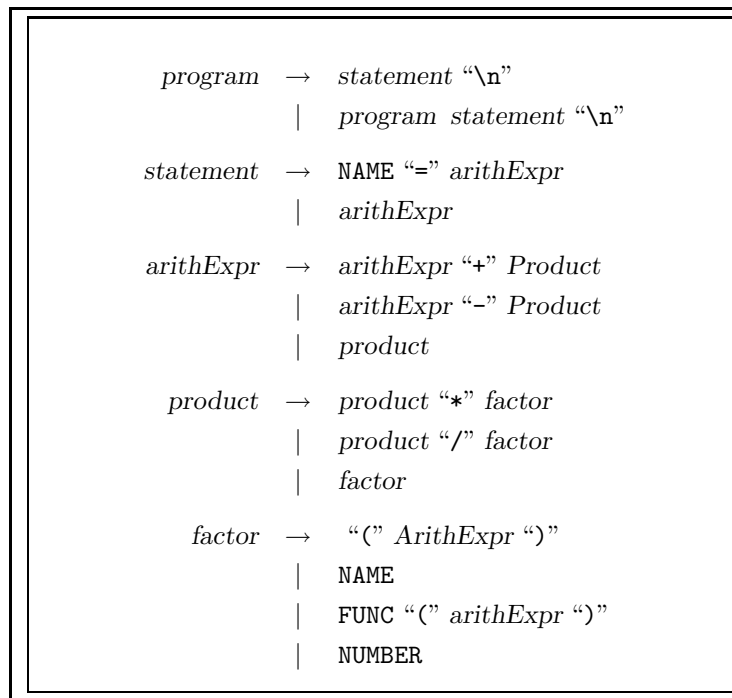


Abbildung 8.1: Eine Grammatik für den Taschenrechner.

Wir diskutieren nun die Details der Abbildung 8.2.

1. In den Zeilen 2 wird die Dateien “**calc.h**” eingebunden. Diese Datei enthält die Definition der Daten-Struktur **SymbolTable**, sowie die Deklaration verschiedener Variablen und Konstanten. Abbildung 8.3 auf Seite 106 zeigt diese Datei, die wir später im Detail diskutieren werden.
2. Die in Zeile 3 eingebundene Datei “**calc.tab.h**” enthält Makro-Definitionen für die verschiedenen Token, die vom Scanner zurück gegeben werden. Zusätzlich wird hier die Datenstruktur **YYSTYPE** definiert. Diese Datei wird von dem Werkzeug *Bison* erzeugt. Die Datei ist in Abbildung 8.4 auf Seite 108 gezeigt. Wir werden diese Datei später genauer analysieren.
3. Die Zeilen 5 und 6 enthalten zwei reguläre Definitionen. *Fraction* steht für eine Zahl, die auch einen Fließpunkt enthalten kann und *Exponent* steht für einen String wie “**e+12**”, der später als Exponent einer Fließkomma-Zahl interpretiert wird.
4. Die Regel in Zeile 8 erkennt eine Fließkomma-Zahl. Die Variable **yylval** bekommt den Wert dieser Zahl zugewiesen. Diese Variable ist von dem in der Datei “**calc.tab.h**” definierten Typ **YYSTYPE**. Dieser Typ ist dort in Zeile 1 – 4 als **union** von einem **double** und einem Zeiger auf den Typ **SymbolTable** definiert, er enthält also wahlweise eine Zahl oder einen Zeiger. Da wir jetzt eine Zahl gelesen haben, die wir **yylval** zuweisen wollen, greifen wir auf **yylval** über die Notation

yylval.value

zu um dem Compiler zu signalisieren, dass **yylval** in diesem Fall als Fließkomma-Zahl zu interpretieren ist.

Außerdem geben wir das Token **NUMBER** zurück, da wir ja gerade eine Zahl gelesen haben. Dieses Token ist in der Datei “**calc.tab.h**” als Makro definiert. Die Rückgabe eines Tokens haben wir bei unseren bisherigen *Flex*-Beispielen noch nicht gesehen, denn bislang hatten

```

1  %{
2  #include "calc.h"      // Deklarationen Benutzer-Funktionen
3  #include "calc.tab.h"  // von Bison erzeugt
4  %}
5  Fraction [0-9]+|([0-9]*\.[0-9]+)
6  Exponent [eE] [-+]?[0-9]+
7  %%
8  {Fraction}{Exponent}? { /* floating point numbers */
9                          yylval.value = atof(yytext);
10                         return NUMBER;
11                         }
12  [ \t\r]                { /* ignore white space */ }
13  [A-Za-z][A-Za-z0-9]* { /* variable names and fuction names */
14                          SymbolTable* sp = lookUpSymbol(yytext);
15                          yylval.symPtr = sp;
16                          if (sp->funcPtr != 0) {
17                              return FUNC;
18                          } else {
19                              return NAME;
20                          }
21                      }
22  [\n=\-+*/()]           { /* operator symbols */
23                          return yytext[0];
24                      }
25  %%
26  void yyerror(char* msg) {
27      printf("%s at '%s'\n", msg, yytext);
28  }

```

Abbildung 8.2: Der Scanner für den Taschenrechner.

wir keinen Parser, mit dem *Flex* kommunizieren musste. Wenn *Flex* und *Bison* zusammenarbeiten, dann muss jede *Flex*-Regel, die ein Token erkannt hat, das später noch vom Parser gebraucht wird, dieses auch zurückgeben. Ein Token wird dabei durch eine natürliche Zahl dargestellt, wobei es folgende Konventionen gibt:

- (a) Das Token **EOF** (*end of file*) wird durch die Zahl 0 repräsentiert.
- (b) Token, die nur aus einem einzigen Buchstaben bestehen, werden durch den entsprechenden ASCII-Code repräsentiert. Beispielsweise hat das Zeichen ‘*’ den ASCII-Code 42 und damit hat das entsprechende Token den Wert 42.
- (c) Alle übrigen Token werden durch natürliche Zahlen größer als 256 repräsentiert. Dabei wird von *Bison* für jedes dieser Token mittels “**#define**” in der Datei `calc.tab.h` ein Makro definiert, das zu der entsprechenden Zahl expandiert wird.

In der in Abbildung 8.4 gezeigten Datei “`calc.tab.h`” sind beispielsweise die Token **NAME**, **FUNC** und **NUMBER** als die Zahlen 257, 258 und 259 definiert.

Generell verläuft die Kommunikation zwischen dem von *Flex* erzeugten Scanner und dem von *Bison* erzeugten Parser über drei Kanäle:

- (a) Über den Rückgabewert der Funktion `yylex()` wird das Token spezifiziert. Die Funktion `yylex()` ist die Funktion, die in der *Flex*-Eingabe-Datei spezifiziert wird. Jeder Aufruf dieser Funktion liefert ein Token zurück.

- (b) Die Variable `yylval` speichert den Wert, der diesem Token zugeordnet ist. Diese Variable hat den Typ `YYSTYPE`. Dieser Typ wird von *Bison* in der Datei `calc.tab.h` definiert.
 - (c) Die Variable `yytext` enthält den Text, der dem letzten gelesenen Token entspricht.
5. Die Regel in Zeile 12 entfernt Leerzeichen, Tabulatoren und Wagen-Rückläufe. Da der Parser sich nicht für diese Zeichen interessiert, steht hier kein `return`-Befehl und damit wird auch kein Token zurückgegeben. Daher sucht der Scanner stattdessen das nächste Token.
 6. Die Regel in Zeile 13 erkennt Variablen und Funktions-Namen. Das Problem ist hier, dass Variablen und Funktions-Namen rein syntaktisch gar nicht unterschieden werden können, denn beide bestehen aus Buchstaben und Ziffern. Daher muss der Scanner mit Hilfe der später im Parser definierten Funktion

```
SymbolTable* lookUpSymbol(char* symbol)
```

überprüfen, ob ein gerade gelesener Name ein Funktions-Name oder aber ein Variablen-Name ist. Die vordefinierten Funktions-Namen wie `"sqrt"`, `"exp"`, etc. sind in einer internen Symbol-Tabelle abgespeichert. In dieser Tabelle werden auch die Variablen-Namen abgelegt. Die Funktion `lookUpSymbol(name)` überprüft für einen gegebenen Namen `name`, ob zu diesem Namen in der Tabelle bereits ein Eintrag existiert. Wenn dies der Fall ist, wird ein Zeiger auf den vorhandenen Eintrag zurückgegeben, ansonsten wird der String `name` in der Tabelle als Variablen-Name eingetragen. Anschließend wird ein Zeiger auf diesen Eintrag zurück gegeben.

Der Zeiger `sp` zeigt also in Zeile 14 in jedem Fall auf ein Objekt vom Typ `SymbolTable`. Dieser Datentyp ist in der Datei `"calc.h"` als Alias für ein `struct` definiert. Dieser `struct` hat drei Einträge.

- (a) Der erste Eintrag `name` gibt den Namen der Variablen oder der Funktion an.
- (b) Der zweite Eintrag mit Namen `funcPtr` ist ein Zeiger auf eine Funktion vom Typ

```
double f(double x).
```

Ist dieser Zeiger von 0 verschieden, so hat der Scanner einen Funktions-Namen erkannt, sonst handelt es sich um einen Variablen-Namen. Entsprechend wird in Zeile 17 entweder das Token `FUNC` zurückgegeben oder es wird in Zeile 19 das Token `NAME` zurückgegeben.

- (c) Der dritte Eintrag der `struct` hat den Namen `value` und gibt für Variablen den momentanen Wert an.
7. Zeile 22 erkennt Token, die nur aus einem Zeichen bestehen. Dies sind die arithmetischen Operatoren `"+"`, `"-"`, `"*"` und `"/"`, die beiden Klammer-Symbole `"("` und `)"`, das Gleichheitszeichen `"="`, das wir als Zuweisungs-Operator verwenden, sowie der Zeilen-Umbruch `"\n"`, der dem Parser das Ende eines arithmetischen Ausdrucks signalisiert. Der Scanner gibt in allen diesen Fällen das gelesene ASCII-Zeichen als Token zurück.
 8. Am Ende der Scanner-Datei definieren wir die Funktion `yyerror()`, mit der später vom Parser Fehlermeldungen ausgegeben werden können. Da diese Funktion von der Variablen `yytext` Gebrauch macht, wird die Funktion `yyerror()` im Scanner definiert. Die Variable `yytext` enthält den Text des zuletzt vom Scanner erkannten Tokens.

Die Datei `"calc.h"` definiert Variablen und Funktionen, die später sowohl vom Scanner als auch vom Parser verwendet werden.

1. Zunächst wird in Zeile 3 – 7 die `struct symtab` definiert und der Name `SymbolTable` wird zum Alias für diese `struct` erklärt. Diese `struct` speichert später die Einträge der Symbol-Tabelle. Diese Einträge sind entweder Variablen-Namen mit dem der Variablen zugeordneten

```

1  #define NSYMS 1000    // maximum number of symbols
2
3  typedef struct symtab {
4      char*   name;
5      double (*funcPtr)();
6      double value;
7  } SymbolTable;
8
9  SymbolTable symtab[NSYMS];
10
11 SymbolTable* lookUpSymbol();
12 void yyerror(char* msg);

```

Abbildung 8.3: Die Header-Datei “`calc.h`”.

Werten `value` oder aber Namen von Funktionen mit einem Zeiger `funcPtr` auf diese Funktion.

- Die eigentliche Symbol-Tabelle wird dann in Zeile 9 als Feld mit `NSYMS` Einträgen definiert. Die Zahl `NSYMS` ist vorher in Zeile 1 als 1000 definiert worden.
- Schließlich werden noch die beiden Funktionen `lookUpSymbol()` und `yyerror()` deklariert. Die Definition der Funktion `lookUpSymbol()` finden wir später in der Datei `calc.y`, die in den Abbildungen 8.6 und 8.7 gezeigt wird, während wir die Definition der Funktion `yyerror()` bereits in dem Scanner `calc.l` in Abbildung 8.2 gesehen haben.

```

1  typedef union {
2      double      value;
3      SymbolTable* symPtr;
4  } YYSTYPE;
5  #define NAME      257
6  #define FUNC      258
7  #define NUMBER    259
8
9  extern YYSTYPE yylval;

```

Abbildung 8.4: Die Datei “`calc.tab.h`”.

Die Datei “`calc.tab.h`” wird automatisch von *Bison* erzeugt, wenn wir *Bison* mit der Option “`-d`” wie folgt aufrufen:

```
bison -d calc.y
```

(Die Datei `calc.y` enthält die Definition der Grammatik und wird später besprochen.) Die Datei `calc.tab.h` definiert die verschiedenen Token und den Typ `YYSTYPE`. Damit hat es Folgendes auf sich: Jedes Token ist zunächst als eine natürliche Zahl definiert. Zusätzlich wird aber jedem Token noch ein Wert zugeordnet. Dieser Wert wird in der Variablen `yylval` gespeichert. Nun ist es so, dass unterschiedlichen Token auch unterschiedliche Arten von Werten zugeordnet werden. So wird einem Token vom Typ `NUMBER` sinnvollerweise ein Wert vom Typ `double` zugeordnet, aber einer Variablen wird statt dessen ein Zeiger auf die Symbol-Tabelle zugeordnet. Dort liegt dann für jede Variablen ein `struct` und wenn dieser Variablen schon ein Wert zugewiesen wurde, dann ist dieser

Wert dort abgelegt. Da es also je nach Art des Tokens verschiedene Werte gibt, wird `YYSTYPE` als `union` definiert.

8.2 Der Parser

Jetzt können wir die Spezifikation des Parsers angeben. Generell hat eine *Bison*-Spezifikation die folgende, an *Flex* angelehnte Struktur: Die einzelnen Teile sind dabei durch die Interpunktions-

1	%{	
2		<i>Deklarations-Teil</i>
3	%}	
4		<i>Definitions-Teil</i>
5	%%	
6		<i>Grammatik-Regeln</i>
7	%%	
8		<i>C-Funktionen</i>

Abbildung 8.5: Struktur einer Grammatik-Spezifikation für *Bison*

Strings “%{”, “%}” und zweimal “%%” getrennt und haben die folgende Bedeutung:

1. Der *Deklarations-Teil* enthält Include-Direktiven, sowie Deklarationen von Funktionen und Variablen. Dieser Teil wird später wörtlich an den Anfang des für den Parser generierten C-Codes kopiert.
2. Der *Definitions-Teil* enthält die Definition der Token sowie der Typen von Nicht-Terminalen. (Genau wie den verschiedenen Token Werte unterschiedlicher Typen zugeordnet werden können, können auch den Nicht-Terminalen Werte verschiedener Typen zugeordnet werden.)
3. Die *Grammatik-Regeln* spezifizieren die Grammatik, für die ein Parser erzeugt werden soll.
4. Die C-Funktionen enthalten die Definition der Funktion *main()* sowie die Definitionen weiterer Funktionen, die in Scanner oder Parser benötigt werden.

Die Abbildungen 8.6 und 8.7 zeigen den Inhalt der Datei *calc.y*, die die Grammatik aus Abbildung 8.1 in *Bison*-gerechter Form darstellt. Wir diskutieren zunächst den Deklarations-Teil, den Definitions-Teil und die Grammatik-Spezifikation.

1. Die Zeilen 2 – 6 binden verschiedene Header-Dateien ein.
2. In den Zeilen 9 – 12 legen wir fest, dass der Daten-Typ `YYSTYPE` eine `union` mit den beiden Feldern `value` und `symPtr` ist. Wir erinnern daran, dass jedes Token und auch jedes Nicht-Terminal einen Wert vom Typ `YYSTYPE` hat.
 - (a) Das Terminal `NUMBER` und die syntaktischen Variablen `arithExpr`, `product` und `factor` haben den Wert `double`. Dies wird in Zeile 15 und 16 festgelegt, denn `value` ist in Zeile 10 als `double` deklariert.
 - (b) Die Terminals `NAME` und `FUNC` haben den Typ `SymbolTable*`, denn `symPtr` ist in Zeile 11 als `SymbolTable*` deklariert.
3. Die Deklaration des Typs eines Terminals hat generell die Form

`token <field> Token`

Hierbei bezeichnet *field* den Namen eines Feldes der `union`, durch die `YYSTYPE` definiert wird.

```

1  %{
2  #include "calc.h"
3  #include <string.h>
4  #include <math.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  %}
8
9  %union {
10     double      value;
11     SymbolTable* symPtr;
12 }
13
14 %token <symPtr> NAME FUNC
15 %token <value>  NUMBER
16 %type <value>  arithExpr product factor
17
18 %%
19 lines      : statement '\n'
20             | lines statement '\n'
21             ;
22
23 statement  : NAME '=' arithExpr      { $1->value = $3;      }
24             | arithExpr              { printf("%g\n", $1); }
25             ;
26
27 arithExpr  : arithExpr '+' product   { $$ = $1 + $3; }
28             | arithExpr '-' product  { $$ = $1 - $3; }
29             | product
30             ;
31
32 product    : product '*' factor      { $$ = $1 * $3; }
33             | product '/' factor     { $$ = $1 / $3; }
34             | factor
35             ;
36
37 factor     : '(' arithExpr ')'       { $$ = $2;          }
38             | NAME                   { $$ = $1->value;      }
39             | FUNC '(' arithExpr ')' { $$ = ($1->funcPtr)($3); }
40             | NUMBER
41             ;
42
43 %%

```

Abbildung 8.6: Grammatik-Spezifikation für den Taschenrechner, Teil 1.

4. Die Deklaration des Typs eines Nicht-Terminals hat die Form

`type <field> NonTerminal.`

Es ist eine Konvention, dass in *Bison*-Spezifikationen Terminale nur aus Groß-Buchstaben bestehen, während Nicht-Terminale nur aus Klein-Buchstaben aufgebaut sind.

5. Die Syntax von Grammatik-Regeln für *Bison* ist erfreulich kompakt. Die Grammatik-Regeln

$$A \rightarrow \alpha \mid \beta \mid \gamma$$

schreiben wir in *Bison* wie folgt:

```

1      A :  $\alpha$ 
2          |  $\beta$ 
3          |  $\gamma$ 
4      ;

```

Aus dem Pfeil “ \rightarrow ” wird also ein Doppelpunkt “:” und außerdem werden die Regeln einer Gruppe durch ein Semikolon abgeschlossen¹. Genau wie bei ANTLR kann eine Grammatik-Regel noch *semantische Aktionen* enthalten. Dies ist C-Code, der in geschweiften Klammern eingeschlossen ist. Typischerweise stehen solche semantische Optionen am Ende einer Regel.

6. Zeile 23 behandelt eine Zuweisung. Falls eine Zuweisung erkannt wurde, muss diese anschließend ausgeführt werden. Dazu dient die semantische Aktion

```
{ $1->value = $3; }
```

Um zu verstehen, wie diese semantische Aktion funktioniert, müssen wir wissen, dass die Terminale und Nicht-Terminal im Rumpf einer Regel mit 1 beginnend durchnummeriert werden. In der Regel

```
statement : NAME '=' arithExpr
```

bekommt `NAME` also die Nummer 1, das Token “=” bekommt die Nummer 2 und das Nicht-Terminal `arithExpr` bekommt die Nummer 3. Dem Terminal `NUMBER` und dem Nicht-Terminal `arithExpr` werden vom Parser Werte zugeordnet. Auf diese Werte können wir zurück greifen, wenn wir der entsprechenden Nummer, die das Token oder Nicht-Terminal identifiziert, ein Dollar-Zeichen “\$” voranstellen. Die Variable “\$1” enthält als den Wert, der dem Token `NAME` zugeordnet ist, während die Variable “\$3” den Wert des Nicht-Terminals `arithExpr` enthält. Dem Token `NAME` wird ein Zeiger auf die Symbol-Table zugeordnet, während dem Nicht-Terminal `arithExpr` eine Zahl vom Typ `double` zugeordnet ist. Der Zeiger, der `NAME` zugeordnet ist, zeigt auf eine Struktur, die ein Feld `value` enthält und in dieses Feld schreiben wir den Wert des geparsen arithmetischen Ausdrucks.

7. Zeile 27 zeigt exemplarisch die Behandlung einer Addition. Hier sehen wir, wie der Parser die Werte, die den Nicht-Terminalen zugeordnet sind, berechnet. Der Wert eines Nicht-Terminals, das auf der linken Seite einer Regel steht, kann mit der Notation “\$\$” referenziert werden. In Zeile 27 setzen wir diesen Wert auf die Summe der gelesenen Werte.

Gleichzeitig sehen sie in Zeile 27 auch, dass die verwendete Grammatik-Regel links-rekursiv ist. Bei LALR(1)-Parsern ist Links-Rekursion unproblematisch, sie ist sogar effizienter als die Rechts-Rekursion, die bei Top-Down-Parsern wie ANTLR benutzt werden muss.

Abbildung 8.7 zeigt die Implementierung der C-Funktionen, die von unserem Parser benutzt werden.

1. Die Funktion `lookUpSymbol()` bekommt als Argument einen String `symbol` übergeben. Dieser String ist entweder der Name einer Variablen oder der Name eine Funktion. Die in `calc.h` definierte globale Variable `syntab` bezeichnet ein Feld von Einträgen, die den Typ `SymbolTable` haben. Aufgabe von `lookUpSymbol()` ist es, den Namen `symbol` in diesem Feld zu suchen, oder aber einen neuen Eintrag für diesen Namen anzulegen. Dazu läuft die `for`-Schleife in Zeile 8 – 19 über alle Einträge in dem Feld `syntab`. Für jeden Eintrag wird

¹In den neueren Versionen von *Bison* ist das Semikolon optional. Es ist aber guter Stil, es trotzdem zu verwenden.

```

44 // look up a symbol table entry, add if not present
45 SymbolTable* lookUpSymbol(char* symbol) {
46     SymbolTable* symbolPtr;
47     for (symbolPtr = symtab; symbolPtr < &symtab[NSYMS]; ++symbolPtr) {
48         // Have we found the symbol?
49         if (symbolPtr->name && !strcmp(symbolPtr->name, symbol)) {
50             return symbolPtr;
51         }
52         // Have we reached the end of the table?
53         if (!symbolPtr->name) {
54             symbolPtr->name = strdup(symbol);
55             return symbolPtr;
56         }
57         // otherwise continue to next table entry
58     }
59     yyerror("Too many symbols");
60     exit(1); // abort
61 }
62
63 void addfunc(char* name, double (*func)(double x)) {
64     SymbolTable* symbolPtr = lookUpSymbol(name);
65     symbolPtr->funcPtr      = func;
66 }
67
68 int main() {
69     addfunc("sqrt", sqrt);
70     addfunc("exp",  exp);
71     addfunc("log",  log);
72     yyparse();
73     return 0;
74 }

```

Abbildung 8.7: C-Funktionen für den Taschenrechner

geprüft, ob der Name dieses Eintrags mit dem Argument `symbol` übereinstimmt. Wenn dies der Fall ist, wird ein Zeiger auf diesen Eintrag zurück gegeben. Andernfalls überprüfen wir in Zeile 14, ob der entsprechende Eintrag noch unbenutzt ist. In diesem Fall tragen wir den Namen an der entsprechenden Stelle ein und geben einen Zeiger auf den neuen Eintrag zurück.

2. Die Funktion `addfunc` dient dazu, Funktionen in das Feld `symtab` einzutragen. Die Funktion `addfunc` bekommt dazu den Namen der einzutragenden Funktion, sowie einen Zeiger auf die einzutragende Funktion als Argument.

Zunächst sucht die Funktion durch den Aufruf von `lookUpSymbol` einen freien Platz in der Symbol-Tabelle. Dort wird dann der Name der Funktion und der Zeiger auf die Funktion eingetragen.

3. Die Funktion `main()` initialisiert die Symbol-Tabelle `symtab`, indem sie die vordefinierten Funktionen `sqrt()`, `exp()` und `log()` einträgt. Anschließend wird der Parser durch den Aufruf von `yyparse()` gestartet. Hier könnten wir problemlos auch alle trigonometrischen Funktionen sowie deren Umkehrfunktionen in die Tabelle eintragen.

Kapitel 9

Keller-Automaten

In diesem Kapitel stellen wir die *Keller-Automaten* (engl. *push-down automata*) vor. Wir werden sehen, dass dieses Automaten-Modell zu dem Konzept der kontextfreien Sprachen äquivalent ist: Eine Sprache L ist genau dann kontextfrei, wenn es einen nicht-deterministischen Keller-Automaten gibt, der die Sprache L erkennt. Insofern verhalten sich Keller-Automaten zu den kontextfreien Sprachen genauso, wie sich die endlichen Automaten zu den regulären Sprachen verhalten. Im Vergleich zur Theorie der endlichen Automaten hat die Theorie der Keller-Automaten allerdings einen Schönheitsfehler: Während die deterministischen endlichen Automaten zu den nicht-deterministischen endlichen Automaten äquivalent sind, sind die nicht-deterministischen Keller-Automaten mächtiger als die deterministischen Keller-Automaten, denn es gibt kontextfreie Sprachen, die sich zwar mit einem nicht-deterministischen Keller-Automaten erkennen lassen, für die es aber keinen deterministischen Keller-Automaten gibt, der diese Sprache erkennt. Allerdings werden wir sehen, dass die meisten für die Praxis interessanten kontextfreien Sprachen bereits von deterministischen Keller-Automaten erkannt werden.

9.1 Definition eines Keller-Automaten

Im Allgemeinen werden Keller-Automaten als endliche Automaten definiert, die zusätzlich mit einem Kellerspeicher (engl. *stack*) ausgestattet sind, auf dem Werte abgelegt werden können. Der Zustand eines Automaten setzt sich dann aus zwei Komponenten zusammen:

1. Dem Inhalt des Kellerspeichers und
2. dem Zustand des zugrunde liegenden endlichen Automaten.

Es zeigt sich aber, dass der Kellerspeicher als Gedächtnis bereits ausreichend ist, eine weitere Komponente wird nicht benötigt. Wir werden uns in unseren Betrachtungen zunächst auf diese Art von Keller-Automaten beschränken.

Definition 22 (Keller-Automat) Ein *Keller-Automat* A ist ein 4-Tupel

$$A = \langle \Sigma, \Gamma, \delta, S_0 \rangle,$$

wobei die Komponenten die folgende Bedeutung haben:

1. Σ ist das *Eingabe-Alphabet*, der Keller-Automat verarbeitet also Wörter aus Σ^* .
2. Γ ist das *Keller-Alphabet*.
In der Praxis ist Γ häufig eine Obermenge von Σ .
3. δ ist die *Übergangs-Funktion*. Als Argument bekommt diese Funktion entweder einen Buchstaben aus dem Eingabe-Alphabet Σ und einen Buchstaben aus dem Keller-Alphabet Γ , oder das Zeichen

ε und einen Buchstaben aus Γ . Als Ausgabe liefert diese Funktion eine Menge von Strings aus dem Keller-Alphabet, es gilt also

$$\delta : (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \text{Pow}(\Gamma^*).$$

Hier bezeichnet $\text{Pow}(\Gamma^*)$ die Potenz-Menge von Γ^* , also die Menge aller Teilmengen von Γ^* und Γ^* bezeichnet die Menge der Wörter über Γ .

4. S_0 ist das Start-Symbol, das ein Element des Keller-Alphabets Γ ist:

$$S_0 \in \Gamma.$$

Zu Beginn der Rechnung eines Keller-Automaten liegt im Kellerspeicher des Keller-Automaten genau dieses Start-Symbol und sonst nichts. \square

Wie funktioniert nun ein Keller-Automat? Ein Wort $w \in \Sigma^*$ wird von dem Keller-Automaten wie folgt verarbeitet.

1. Zunächst wird in dem leeren Kellerspeicher das Start-Symbol S_0 abgelegt.
2. Anschließend wird das Wort Buchstabe für Buchstabe gelesen, wobei sich der Kellerspeicher des Automaten wie folgt verändern kann:
 - (a) Ist C der Buchstabe, der im Kellerspeicher (also auf dem Stack) ganz oben liegt, und gilt $\alpha \in \delta(\varepsilon, C)$, dann wird der Buchstabe C aus dem Kellerspeicher entfernt und die Buchstaben aus α werden so eingekellert dass der erste Buchstabe aus α oben auf dem Stack liegt. Da α auch das leere Wort sein kann, kann der Stack dabei auch schrumpfen. In diesem Fall wird kein Buchstabe des Wortes w gelesen.
 - (b) Ist C der Buchstabe, der im Kellerspeicher (also auf dem Stack) ganz oben liegt, ist b der nächste zu lesende Buchstabe aus dem Wort w und gilt $\alpha \in \delta(b, C)$, dann wird der Buchstabe b gelesen, der Buchstabe C wird aus dem Kellerspeicher entfernt und anschließend werden die Buchstaben aus α eingekellert.

Keller-Automaten sind nicht-deterministisch, denn beispielsweise kann für $\alpha_1 \neq \alpha_2$ sowohl

$$\alpha_1 \in \delta(b, C), \quad \text{als auch} \quad \alpha_2 \in \delta(b, C) \quad \text{gelten,}$$

und dann hat der Automat die Wahl, ob er α_1 oder α_2 einkellert. Genauso kann es sein, dass gleichzeitig

$$\alpha_1 \in \delta(\varepsilon, C) \quad \text{und} \quad \alpha_2 \in \delta(b, C)$$

gilt. In diesem Fall hat der Automat die Möglichkeit, entweder das Zeichen C im Kellerspeicher durch das Wort α_1 zu ersetzen, ohne dabei einen Buchstaben zu lesen, oder der Automat kann den Buchstaben b lesen und das Zeichen C durch das Wort α_2 ersetzen.

3. Sind alle Buchstaben des Wortes w gelesen, und ist gleichzeitig der Kellerspeicher geleert worden, dann akzeptiert der Keller-Automat das Wort w .

Formal definieren wir die *Konfiguration* eines Keller-Automaten $A = \langle \Sigma, \Gamma, \delta, S_0 \rangle$, als ein Paar

$$\langle v, \alpha \rangle \in \Sigma^* \times \Gamma^*,$$

dass aus einem Wort v des Eingabe-Alphabets und einem weiteren Wort α des Keller-Alphabets besteht. Die Idee ist, dass v den Teil des Eingabewortes darstellt, der noch nicht gelesen wurde, während α den Zustand des Kellerspeicher beschreibt. Wir definieren nun für zwei Konfigurationen $\langle u, \alpha \rangle$ und $\langle v, \beta \rangle$ des Keller-Automaten A eine Relation

$$\langle u, \alpha \rangle \vdash_A \langle v, \beta \rangle$$

durch die folgenden beiden Klauseln:

1. $\alpha \in \delta(\varepsilon, C) \rightarrow \langle w, C\beta \rangle \vdash_A \langle w, \alpha\beta \rangle$.

Hier hat der Keller-Automat A einen ε -Übergang, bei dem das Stack-Symbol C durch den String α ersetzt wird. Das Wort w bleibt in diesem Fall unverändert.

2. $\alpha \in \delta(b, C) \rightarrow \langle bu, C\beta \rangle \vdash_A \langle u, \alpha\beta \rangle$.

In diesem Fall hat der Keller-Automat A für den Buchstaben b und das Stack-Symbol einen Übergang, bei dem b gelesen und das Stack-Symbol C durch den String α ersetzt wird.

Wir bezeichnen den transitiven und reflexiven Abschluss der Relation \vdash_A mit \vdash_A^* . Bei Diskussionen, in denen klar ist, um welchen Keller-Automaten A es sich handelt, lassen wir den Index A auch weg und schreiben einfach \vdash^* an Stelle von \vdash_A^* . Damit sind wir nun in der Lage, die von einem Keller-Automaten

$$A = \langle \Sigma, \Gamma, \delta, S_0 \rangle$$

akzeptierte Sprache $L(A)$ zu definieren:

$$L(A) := \{w \in \Sigma^* \mid \langle w, S_0 \rangle \vdash_A^* \langle \varepsilon, [] \rangle\},$$

wobei wir hier mit $[]$ den leeren Kellerspeicher bezeichnen. Ein Wort w wird also genau dann von dem Keller-Automaten A akzeptiert, wenn der Keller-Automat beim Lesen dieses Wortes seinen Kellerspeicher leeren kann.

Beispiel: Wir definieren den Keller-Automat $A = \langle \Sigma, \Gamma, \delta, S \rangle$ wie folgt:

1. $\Sigma := \{a, b\}$.
2. $\Gamma := \{a, b, S\}$.
3. $\delta(\varepsilon, a) := \{\}$, $\delta(\varepsilon, b) := \{\}$, $\delta(\varepsilon, S) := \{aS a, bS b, a, b, \varepsilon\}$,
 $\delta(a, a) := \{\varepsilon\}$, $\delta(a, b) := \{\}$, $\delta(a, S) := \{\}$,
 $\delta(b, a) := \{\}$, $\delta(b, b) := \{\varepsilon\}$, $\delta(b, S) := \{\}$.

Beachten Sie, dass es einen subtilen Unterschied zwischen den Fällen

$$\delta(a, a) := \{\varepsilon\} \quad \text{und} \quad \delta(a, b) := \{\}$$

gibt: Falls der nächste Buchstaben des zu verarbeitenden Wortes ein “a” ist und falls außerdem ein “a” oben auf dem Stack liegt, dann wird das “a” aus dem Wort gelesen und das “a” auf dem Stack wird durch das leere Wort ε ersetzt und damit entfernt.

Ist hingegen der nächste Buchstaben des zu verarbeitenden Wortes ein “a” und liegt ein “b” oben auf dem Stack, so gibt es für den Automaten keinen Übergang und damit kann der Buchstabe “a” nicht gelesen werden. In diesem Fall sagen wir auch, dass der Automat *stirbt*.

4. Das Start-Symbol ist S .

Ist w ein Wort, so bezeichnen wir mit w^r das Wort, das aus w entsteht, wenn wir w von hinten lesen. Der erste Buchstabe von w ist also der letzte Buchstabe von w^r , der zweite Buchstabe von w ist der vorletzte Buchstabe von w^r und allgemein gilt für ein Wort der Länge n

$$|w| = n \rightarrow \forall i \in \{1, \dots, n\} : w^r[i] = w[n + 1 - i].$$

Hier bezeichnet $|w|$ die Länge des Wortes und $w[i]$ bezeichnet den i -ten Buchstaben. Wörter, für die $w^r = w$ gilt, werden als *Palindrome* bezeichnet. Ein einfaches Beispiel ist das Wort

aabaa.

Wir behaupten, dass der oben definierte Automat A genau die Palindrome erkennt, es gilt also

$$L(A) = \{w \in \Sigma^* \mid w = w^r\}$$

Wir zeigen, wie der Automat das Wort

aabaa

verarbeitet.

1. Der Automat startet mit der Konfiguration

$$\langle aabaa, S \rangle$$

2. Da das oberste Symbol auf dem Stack ein S ist, kann der Automat die Tatsache, dass

$$aSa \in \delta(\varepsilon, S)$$

ausnutzen und einen ε -Übergang durchführen. Dabei wird dann die folgende Konfiguration erreicht:

$$\langle aabaa, aSa \rangle$$

3. Jetzt ist der erste Buchstabe des zu lesenden Wortes ein a und gleichzeitig liegt ein a auf dem Stack. Also verwenden wir

$$\varepsilon \in \delta(a, a)$$

und entfernen damit sowohl das erste “ a ” aus dem zu lesenden Wort als auch das “ a ” auf dem Stack und erhalten die Konfiguration

$$\langle abaa, Sa \rangle.$$

4. Wir benutzen wieder $aSa \in \delta(\varepsilon, S)$ und erhalten die Konfiguration

$$\langle abaa, aSaa \rangle.$$

5. Nun verwenden wir $\varepsilon \in \delta(a, a)$ und erhalten

$$\langle baa, Saa \rangle.$$

6. Diesmal benutzen wir $b \in \delta(\varepsilon, S)$ und erhalten

$$\langle baa, baa \rangle.$$

7. Jetzt verwenden wir einmal $\varepsilon \in \delta(b, b)$ und zweimal $\varepsilon \in \delta(a, a)$ und haben dann die Konfiguration

$$\langle \varepsilon, [] \rangle.$$

Damit ist gezeigt, dass $aabaa \in L(A)$ gilt. □

9.2 Von kontextfreien Sprachen zu Keller-Automaten

Die Menge $L_P(\Sigma)$ der Palindrome über einem Alphabet Σ

$$L_P(\Sigma) := \{w \in \Sigma^* \mid w^r = w\}$$

kann auch durch eine kontextfreie Grammatik beschrieben werden. Definieren wir die kontextfreie Grammatik G durch

$$G = \langle \{S\}, \{a, b\}, R, S \rangle,$$

wobei die Regeln durch

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon,$$

so kann gezeigt werden, dass gilt:

$$L(G) = L_P(\{a, b\}).$$

Vergleichen wir die Definition der Grammatik mit der Definition des Keller-Automaten aus dem letzten Abschnitt, so erkennen wir ein Muster, dass wir jetzt allgemein formulieren.

Definition 23 ($A(G)$) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik. Dann definieren wir den von G erzeugten Keller-Automaten $A(G)$ wie folgt:

$$A(G) = \langle T, V \cup T, \delta, S \rangle,$$

wobei die Übergangs-Funktion δ durch die folgenden Klauseln definiert wird.

1. Für jede Regel $A \rightarrow \beta$ aus der Grammatik G gilt $\beta \in \delta(\varepsilon, A)$, denn wir definieren

$$\delta(\varepsilon, A) := \{\beta \mid (A \rightarrow \beta) \in R\} \quad \text{für alle } A \in V.$$

2. Für jeden Buchstaben $b \in T$ gilt $\varepsilon \in \delta(b, b)$, denn wir definieren

$$\delta(b, c) := \begin{cases} \{\varepsilon\} & \text{falls } b = c; \\ \{\} & \text{sonst.} \end{cases}$$

□

3. In allen anderen Fällen liefert die Funktion δ als Ergebnis die leere Menge.

$$(a) \quad \forall b \in T : \delta(\varepsilon, b) = \{\},$$

$$(b) \quad \forall b \in T, A \in V : \delta(b, A) = \{\}.$$

Bemerkung: Der im letzten Abschnitt angegebene Automat A ist genau der von der oben angegebenen Grammatik G erzeugte Automat $A(G)$.

Um zeigen zu können, dass der Automat $A(G)$ genau die Sprache akzeptiert, die von der Grammatik G beschrieben wird, benötigen wir den Begriff der *Links-Ableitung*: Informal ist das ein Ableitungsschritt, bei dem die linkeste Variable durch eine Regel ersetzt wird. Formal definieren wir diesen Begriff wie folgt:

1. Es sei $G = \langle T, V, R, S \rangle$ eine kontextfreie Grammatik. Weiter sei
2. $A \in V$ eine syntaktische Variable,
3. $uA\beta \in (V \cup T)^*$ sei ein String aus Variablen und Terminalen mit $u \in T^*$ und
4. $(A \rightarrow \gamma) \in R$ sei eine Regel der Grammatik G .

Dann schreiben wir

$$uA\beta \Rightarrow_1 u\gamma\beta.$$

und sagen, dass das Wort $u\gamma\beta$ durch einen Schritt einer *Links-Ableitung* aus dem Wort $uA\beta$ hervorgegangen ist. Der entscheidende Unterschied gegenüber einem normalen Ableitungs-Schritt ist der, dass alle Zeichen, die links von der ersetzten Variablen A stehen, Terminale sein müssen: $u \in T^*$. Ist allgemein w ein Wort einer Sprache $L(G)$ und ist S das Start-Symbol, dann erhalten wir eine *Links-Ableitung* von w , wenn wir eine Ableitung wählen, bei der in jedem Schritt die linkeste Variable durch die rechte Seite einer Grammatik-Regel ersetzt wird.

Beispiel: Eine Grammatik $G = \langle \{E, P, F\}, \{“+”, “*”, “1”, “2”, “3”\}, R, E \rangle$ für arithmetische Ausdrücke habe die folgenden Regeln R :

$$E \rightarrow E “+” P \mid P, \quad P \rightarrow P “*” F \mid F, \quad F \rightarrow “1” \mid “2” \mid “3”.$$

Der String “1+2*3” hat dann die folgende Links-Ableitung:

$$\begin{aligned}
E &\Rightarrow_1 E \text{ "+" } P \\
&\Rightarrow_1 P \text{ "+" } P \\
&\Rightarrow_1 F \text{ "+" } P \\
&\Rightarrow_1 \text{"1"} \text{ "+" } P \\
&\Rightarrow_1 \text{"1"} \text{ "+" } P \text{ "*" } F \\
&\Rightarrow_1 \text{"1"} \text{ "+" } F \text{ "*" } F \\
&\Rightarrow_1 \text{"1"} \text{ "+" } \text{"2"} \text{ "*" } F \\
&\Rightarrow_1 \text{"1"} \text{ "+" } \text{"2"} \text{ "*" } \text{"3"}
\end{aligned}$$

Die folgende Ableitung ist hingegen keine Links-Ableitung, denn im zweiten Schritt wird zum Beispiel nicht die linkeste Variable E sondern statt dessen die Variable P ersetzt:

$$\begin{aligned}
E &\Rightarrow E \text{ "+" } P \\
&\Rightarrow E \text{ "+" } P \text{ "*" } F \\
&\Rightarrow P \text{ "+" } P \text{ "*" } F \\
&\Rightarrow P \text{ "+" } P \text{ "*" } \text{"3"} \\
&\Rightarrow F \text{ "+" } P \text{ "*" } \text{"3"} \\
&\Rightarrow \text{"1"} \text{ "+" } P \text{ "*" } \text{"3"} \\
&\Rightarrow \text{"1"} \text{ "+" } F \text{ "*" } \text{"3"} \\
&\Rightarrow \text{"1"} \text{ "+" } \text{"2"} \text{ "*" } \text{"3"}
\end{aligned}$$

Bemerkung: Falls wir ein Wort w aus dem Start-Symbol S einer Grammatik $G = \langle V, T, R, S \rangle$ ableiten können, so können wir die Ableitungs-Schritte immer so umsortieren, dass wir hinterher eine Links-Ableitung haben. Daher gilt

$$L(G) = \{w \in T^* \mid S \Rightarrow_1^* w\}.$$

Satz 24 Ist $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik, so gilt

$$L(A(G)) = L(G),$$

der von G erzeugte Automat erkennt also genau die Sprache, die durch G beschrieben wird.

Beweis: Wir zeigen zunächst, dass

$$L(G) \subseteq L(A(G))$$

gilt. Wir nehmen also an, dass $w \in L(G)$ ist und müssen zeigen, dass der Automat $A(G)$ das Wort w akzeptiert. Wir nehmen nun eine Ableitung des Wortes w , bei der immer die linkeste Variable ersetzt wird. Eine solche Ableitung hat die Form

$$S = u_1 A_1 \beta_1 \Rightarrow_1 u_2 A_2 \beta_2 \Rightarrow_1 \cdots \Rightarrow_1 u_n A_n \beta_n \Rightarrow_1 w,$$

wobei gilt

$$1. \ u_1 = \varepsilon, A_1 = S, \beta_1 = \varepsilon.$$

$$2. \ u_i \in T^* \quad \text{für alle } i = 1, \dots, n$$

links von den syntaktischen Variablen A_i stehen also nur Terminale.

3. $A_i \in V$ für alle $i = 1, \dots, n$,

A_i ist also die syntaktische Variable in dem String $u_i A_i \beta_i$, die am weitesten links steht.

Wir zeigen nun durch Induktion über i , dass es für alle $i = 1, \dots, n$ Strings $v_i \in T^*$ gibt, so dass

$$\langle w, S \rangle \vdash^* \langle v_i, A_i \beta_i \rangle \text{ und } u_i v_i = w$$

gilt, der String v_i ist also der Rest, der von dem String w übrig bleibt, wenn das Präfix u_i entfernt wird.

I.A. $i = 1$: Die Start-Konfiguration ist

$$\langle w, S \rangle = \langle w, u_1 A_1 \beta_1 \rangle$$

und damit muss gelten

$$u_1 = \varepsilon, \quad A_1 = S, \quad \text{und} \quad \beta_1 = \varepsilon.$$

Also setzen wir $v_1 := w$ und haben trivialerweise

$$\langle w, S \rangle \vdash^* \langle w, S \rangle = \langle v_1, u_1 A_1 \beta_1 \rangle \quad \text{und} \quad u_1 v_1 = w.$$

I.S. $i \mapsto i + 1$: Nach Induktions-Voraussetzung wissen wir, dass

$$\langle w, S \rangle \vdash^* \langle v_i, A_i \beta_i \rangle \quad \text{und} \quad u_i v_i = w \tag{9.1}$$

gilt. Wegen

$$u_i A_i \beta_i \xRightarrow{1} u_{i+1} A_{i+1} \beta_{i+1} \quad \text{und} \quad u_i \in T^* \tag{9.2}$$

muss es eine Regel

$$(A_i \rightarrow \gamma_i) \in R$$

geben, so dass bei dem Ableitungs-Schritt A_i durch γ_i ersetzt worden ist. Damit gilt also

$$u_{i+1} A_{i+1} \beta_{i+1} = u_i \gamma_i \beta_i. \tag{9.3}$$

Wegen (9.2) muss u_i ein Präfix von u_{i+1} sein. Also gibt es einen String $t_{i+1} \in T^*$, so dass

$$u_i t_{i+1} = u_{i+1}$$

gilt. Schneiden wir in Gleichung (9.3) auf beiden Seiten den Präfix u_i ab, so erhalten wir

$$\gamma_i \beta_i = t_{i+1} A_{i+1} \beta_{i+1}. \tag{9.4}$$

Aufgrund der Linksableitung

$$S \xRightarrow{1}^* u_{i+1} A_{i+1} \beta_{i+1} \xRightarrow{1}^* w,$$

muss u_{i+1} ein Präfix von w sein und daher gibt es genau ein $v_{i+1} \in T^*$, so dass

$$u_{i+1} v_{i+1} = w$$

gilt. Setzen wir hier einerseits $u_{i+1} = u_i t_{i+1}$ und andererseits $w = u_i v_i$ ein, so finden wir

$$u_i t_{i+1} v_{i+1} = u_i v_i.$$

Schneiden wir auf beiden Seiten dieser Gleichung u_i ab, so folgt

$$t_{i+1} v_{i+1} = v_i.$$

Nach Konstruktion des Keller-Automaten $A(G)$ folgt aus $(A_i \rightarrow \gamma_i) \in R$

$$\gamma_i \in \delta(\varepsilon, A_i)$$

und nach Definition eines Keller-Automaten gilt

$$\langle v_i, A_i \beta_i \rangle \vdash \langle v_i, \gamma_i \beta_i \rangle. \quad (9.5)$$

Wir haben oben gesehen, dass $v_i = t_{i+1} v_{i+1}$ gilt und Gleichung (9.4) zeigt $\gamma_i \beta_i = t_{i+1} A_{i+1} \beta_{i+1}$. Also können wir (9.5) auch als

$$\langle v_i, A_i \beta_i \rangle \vdash \langle t_{i+1} v_{i+1}, t_{i+1} A_{i+1} \beta_{i+1} \rangle \quad (9.6)$$

schreiben. Nun gilt nach Konstruktion von $A(G)$ für alle Buchstaben $b \in T$

$$\varepsilon \in \delta(b, b).$$

Damit haben wir

$$\langle t_{i+1} v_{i+1}, t_{i+1} A_{i+1} \beta_{i+1} \rangle \vdash^* \langle v_{i+1}, A_{i+1} \beta_{i+1} \rangle$$

und zusammen mit (9.1) und 9.6) folgt daraus

$$\langle w, S \rangle \vdash^* \langle v_{i+1}, A_{i+1} \beta_{i+1} \rangle,$$

was für den Induktions-Schritt zu zeigen war und die Induktion ist abgeschlossen.

Setzen wir oben $i = n$, so haben wir gezeigt, dass

$$\langle w, S \rangle \vdash^* \langle v_n, A_n \beta_n \rangle \quad \text{und} \quad u_n v_n = w \quad (9.7)$$

gilt und wissen außerdem, dass

$$u_n A_n \beta_n \xRightarrow{1} w$$

gilt. Also gibt es eine Regel $A_n \rightarrow \gamma_n$ in R und es muss

$$u_n \gamma_n \beta_n = w$$

gelten. Daraus folgt aber wegen $u_n v_n = w$

$$\gamma_n \beta_n = v_n.$$

Nach Konstruktion von $A(G)$ gilt $\gamma_n \in \delta(\varepsilon, A_n)$ und wir haben

$$\langle v_n, A_n \beta_n \rangle \vdash \langle v_n, \gamma_n \beta_n \rangle = \langle v_n, v_n \rangle.$$

Da nach Konstruktion von $A(G)$ für alle Buchstaben $b \in T$ gilt $\varepsilon \in \delta(b, b)$, haben wir

$$\langle v_n, v_n \rangle \vdash^* \langle \varepsilon, [] \rangle$$

und damit ist insgesamt

$$\langle w, S \rangle \vdash^* \langle \varepsilon, [] \rangle$$

gezeigt und wir sehen, dass $w \in L(A(G))$ gilt.

Um den Beweis abzuschließen, zeigen wir nun die umgekehrte Richtung und weisen

$$L(A(G)) \subseteq L(G)$$

nach. Wir nehmen also an, dass $w \in L(A(G))$ gilt und weisen nach, dass daraus $w \in L(G)$ folgt. Erinnern wir uns an die Definition der von einem Keller-Automaten akzeptierten Sprache, sowie an die Definition von $L(G)$, so sagt die Voraussetzung $w \in L(A(G))$ dass aus

$$\langle w, S \rangle \vdash^* \langle \varepsilon, [] \rangle \quad \text{die Konklusion} \quad S \Rightarrow^* w$$

folgt. Wir müssen also die Beziehung

$$\langle w, S \rangle \vdash^* \langle \varepsilon, [] \rangle \rightarrow S \Rightarrow^* w$$

nachweisen. Um diese Behauptung zeigen zu können, verallgemeinern wir sie: Wir werden für jede

syntaktische Variable X und jedes Wort $w \in T^*$ zeigen, dass

$$\langle w, X \rangle \vdash^* \langle \varepsilon, [] \rangle \rightarrow X \Rightarrow^* w$$

gilt. Der Beweis dieser Behauptung erfolgt durch Induktion über die Anzahl n der Berechnungsschritte des Keller-Automaten $A(G)$.

I.A. $n = 1$: X kann sowohl eine syntaktische Variable als auch ein Terminal sein. Wir behandeln diese beiden Fälle getrennt:

(a) $X \in V$, also ist X eine syntaktische Variable.

Die Zustands-Übergänge des Keller-Automaten $A(G)$, bei denen ein Buchstabe des Wortes gelesen wird, setzen alle voraus, dass auf dem Stack der selbe Buchstabe liegt. Falls der Keller-Automat nur einen Schritt braucht, um von der Konfiguration $\langle w, X \rangle$ zur Konfiguration $\langle \varepsilon, [] \rangle$, so kann also kein Buchstabe gelesen worden sein. Daher muss $w = \varepsilon$ gelten und andererseits muss die Grammatik G eine Regel der Form

$$X \rightarrow \varepsilon$$

enthalten. Daraus folgt dann aber sofort

$$X \Rightarrow \varepsilon$$

und wegen $w = \varepsilon$ ist das die Behauptung.

(b) $X \in T$, also ist X ein Terminal.

Nach Konstruktion des Keller-Automaten $A(G)$ folgt aus der Voraussetzung

$$\langle w, X \rangle \rightarrow \langle \varepsilon, [] \rangle,$$

dass das Wort w nur aus dem Buchstaben X besteht: $w = X$. Dann gilt trivialerweise

$$X \Rightarrow^* w.$$

I.S. $1, \dots, n \mapsto n + 1$: Da nun bei der Rechnung

$$\langle w, X \rangle \vdash^* \langle \varepsilon, [] \rangle$$

mehr als ein Schritt durchgeführt wird, kann X nur eine syntaktische Variable sein. Wir nehmen also an, dass der Automat die Rechnung

$$\langle w, X \rangle \vdash^* \langle \varepsilon, [] \rangle$$

in $n + 1$ Schritten durchführt. Da X eine Variable ist, muss der erste Schritt der Rechnung des Keller-Automaten $A(G)$ die Form

$$\langle w, X \rangle \vdash \langle w, \gamma \rangle$$

haben, wobei $X \rightarrow \gamma$ eine Regel der Grammatik sein muss. Wir zerlegen die rechte Seite γ dieser Regel in ihre Buchstaben und schreiben

$$\gamma = y_1 y_2 \cdots y_k, \quad \text{mit } y_i \in T \cup V.$$

Da insgesamt

$$\langle w, X \rangle \vdash^* \langle \varepsilon, [] \rangle$$

gilt, wissen wir

$$\langle w, y_1 y_2 \cdots y_k \rangle \vdash^* \langle \varepsilon, [] \rangle,$$

wobei der Keller-Automat insgesamt n Schritte durchführt. Offenbar entfernt der Keller-Automat $A(G)$ nacheinander die Zeichen $y_1 y_2 \cdots y_k$ vom Stack und liest dabei das Wort w . Damit können wir die Rechnung des Keller-Automaten wie folgt aufspalten:

$$\langle w, y_1 y_2 \cdots y_k \rangle \vdash^* \langle w_2, y_2 \cdots y_k \rangle \vdash^* \cdots \vdash^* \langle w_i, y_i \cdots y_k \rangle \vdash^* \cdots \vdash^* \langle \varepsilon, [] \rangle,$$

Dabei bezeichnet w_i den Teil des Wortes w , der noch nicht gelesen ist, wenn die Zeichen

$y_1 \cdots y_{i-1}$ vom Stack entfernt worden sind. Offenbar können wir das Wort w so in Teilstrings $x_1 \cdots x_k$ zerlegen, dass x_i der Teilstring von w ist, der von dem Automaten gelesen wird, um das Zeichen y_i vom Stack zu entfernen, es gilt also

$$w = x_1 x_2 \cdots x_k \quad \text{und} \quad w_i = x_i x_{i+1} \cdots x_k.$$

Daraus folgt aber, dass das Lesen von x_i das Symbol y_i vom Stack entfernt:

$$\langle \underbrace{x_i x_{i+1} \cdots x_k}_{w_i}, y_i y_{i+1} \cdots y_k \rangle \vdash^* \langle \underbrace{x_{i+1} \cdots x_k}_{w_{i+1}}, y_{i+1} \cdots y_k \rangle \quad (9.8)$$

Da der Keller-Automat immer nur das oberste Zeichen auf dem Stack anschauen kann, können die Zeichen $y_{i+1} \cdots y_k$ bei dieser Rechnung keine Rolle spielen, und genauso spielt auch der Teil des Strings, der bei dieser Rechnung nicht gelesen wird, keine Rolle. Damit folgt aus (9.8), dass

$$\langle x_i, y_i \rangle \vdash^* \langle \varepsilon, [] \rangle \quad (9.9)$$

gilt und für diese Rechnung benötigt der Keller-Automat genauso viele Berechnungs-Schritte wie in (9.8) und das sind höchstens n Schritte. Damit können wir auf (9.9) die Induktions-Voraussetzung anwenden und es gilt

$$y_i \Rightarrow^* x_i \quad (9.10)$$

Damit haben wir insgesamt folgende Ableitung von w :

$$\begin{aligned} X &\Rightarrow \gamma = y_1 y_2 \cdots y_k \\ &\Rightarrow^* x_1 y_2 \cdots y_k \\ &\quad \vdots \\ &\Rightarrow^* x_1 \cdots x_i y_{i+1} y_{i+1} \cdots y_k \\ &\Rightarrow^* x_1 \cdots x_i x_{i+1} y_{i+2} \cdots y_k \\ &\quad \vdots \\ &\Rightarrow^* x_1 \cdots x_k = w \end{aligned}$$

und haben damit die Induktion und den Beweis abgeschlossen. \square

Aufgabe 16: Die Grammatik G sei wie folgt gegeben:

$$G = \langle \{S, A\}, \{x, y\}, R, S \rangle,$$

wobei die Regeln R wie folgt gegeben sind:

$$\begin{aligned} S &\rightarrow xSy \mid A \\ A &\rightarrow yAx \mid S \mid \varepsilon \end{aligned}$$

Konstruieren Sie einen Keller-Automaten, der die Sprache $L(G)$ akzeptiert.

9.3 Von Keller-Automaten zu kontextfreien Sprachen

Nach dem letzten Abschnitt wissen wir, dass das Konzept der Keller-Automaten mindestens so mächtig ist, wie das Konzept der kontextfreien Sprachen. Wir zeigen in diesem Abschnitt die umgekehrte Richtung und weisen nach, dass alle Sprachen, die von Keller-Automaten erkannt werden können, auch durch eine Grammatik beschrieben werden können.

Satz 25 Es sei $A = \langle \Sigma, \Gamma, \delta, S \rangle$ ein Keller-Automat. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass die Mengen Σ und Γ disjunkt sind. Dies können wir immer erreichen, indem wir die Symbole aus Γ umbenennen. Dann konstruieren wir eine Grammatik

$$G(A) := \langle \Gamma, \Sigma, R, S \rangle,$$

indem wir die Menge R der Regeln wie folgt definieren:

1. Ist $b \in \Sigma$, $X \in \Gamma$ und gilt $\gamma \in \delta(b, X)$, dann enthält R die Regel

$$X \rightarrow b\gamma.$$

2. Ist $X \in \Gamma$ und gilt $\gamma \in \delta(\varepsilon, X)$, dann enthält R die Regel

$$X \rightarrow \gamma.$$

Insgesamt setzen wir also

$$R := \{X \rightarrow b\gamma \mid b \in \Sigma \wedge X \in \Gamma \wedge \gamma \in \delta(b, X)\} \cup \{X \rightarrow \gamma \mid X \in \Gamma \wedge \gamma \in \delta(\varepsilon, X)\}.$$

Dann gilt

$$L(G(A)) = L(A).$$

Beweis: Wir zerlegen den Beweis in zwei Teile und zeigen zunächst, dass

$$L(G(A)) \subseteq L(A)$$

gilt. Diese Behauptung ist äquivalent zu

$$S \Rightarrow^* w \rightarrow \langle w, S \rangle \vdash^* \langle \varepsilon, [] \rangle$$

Wir zeigen eine etwas allgemeinere Behauptung. Wir zeigen durch Induktion nach n , dass für alle $X \in \Gamma$ und $w \in \Sigma^*$ folgendes gilt:

$$X \Rightarrow^n w \rightarrow \langle w, X \rangle \vdash^* \langle \varepsilon, [] \rangle$$

Hier drückt die Notation $X \Rightarrow^n w$ aus, dass das Wort w aus der Variablen X in n Schritten abgeleitet werden kann.

I.A. $n = 1$. Dann gibt es in der Grammatik $G(A)$ eine Regel der Form

$$X \rightarrow w$$

Es gibt zwei Möglichkeiten, wie diese Regel entstanden sein kann.

- (a) $w = b\gamma$ mit $b \in \Sigma$ und $\gamma \in \delta(b, X)$.

Einerseits gilt $w \in \Sigma^*$, andererseits folgt aus $\gamma \in \delta(b, X)$, dass $\gamma \in \Gamma^*$ gelten muss. Da γ ein Suffix von w ist, gilt auch $\gamma \in \Sigma^*$, also insgesamt

$$\gamma \in \Sigma^* \cap \Gamma^*.$$

Nun haben wir aber vorausgesetzt, dass das Eingabe-Alphabet Σ und das Stack-Alphabet Γ disjunkt sind. Damit ist die obige Gleichung nur möglich, falls

$$\gamma = \varepsilon$$

gilt. Also haben wir insgesamt

$$w = b \in \Sigma \quad \text{und} \quad \varepsilon \in \delta(b, X).$$

Daraus folgt aber sofort

$$\langle b, X \rangle \vdash \langle \varepsilon, [] \rangle.$$

(b) $w = \gamma$ und $\gamma \in \delta(\varepsilon, X)$.

Einerseits gilt dann $w \in \Sigma^*$, andererseits folgt aus $\gamma \in \delta(\varepsilon, X)$, dass $\gamma \in \Gamma^*$ gelten muss und genau wie im ersten Fall können wir daraus auf

$$\gamma = \varepsilon$$

schließen. Daraus folgt dann

$$\langle \varepsilon, X \rangle \vdash \langle \varepsilon, [] \rangle.$$

I.S. $n \mapsto n + 1$. Wieder gibt es zwei Möglichkeiten für die erste verwendete Regel.

(a) Die erste bei der Ableitung $X \Rightarrow^{n+1} w$ verwendete Regel hat die Form

$$X \rightarrow b\gamma \quad \text{mit } b \in \Sigma \text{ und } \gamma \in \delta(b, X).$$

Insgesamt haben wir dann

$$X \Rightarrow b\gamma \Rightarrow^n w, \quad \text{also } w = bv \text{ mit } \gamma \Rightarrow^n v.$$

Dann haben γ und v die Form

$$\gamma = Y_1 \cdots Y_k, \quad v = u_1 \cdots u_k \quad \text{und} \quad Y_i \Rightarrow^* u_i \quad \text{für alle } i = 1, \dots, k.$$

Auf $Y_i \Rightarrow^* u_i$ können wir die Induktions-Voraussetzung anwenden und sehen, dass

$$\langle u_i, Y_i \rangle \vdash^* \langle \varepsilon, [] \rangle \quad \text{für alle } i = 1, \dots, k$$

gilt. Setzen wir diese Rechnungen zusammen, so haben wir

$$\langle u_1 \cdots u_k, Y_1 \cdots Y_k \rangle \vdash^* \langle \varepsilon, [] \rangle, \quad \text{also} \quad \langle v, \gamma \rangle \vdash^* \langle \varepsilon, [] \rangle.$$

Damit finden wir insgesamt

$$\langle w, X \rangle = \langle bv, X \rangle \vdash \langle v, \gamma \rangle \vdash^* \langle \varepsilon, [] \rangle$$

und das war die Behauptung.

(b) Die erste bei der Ableitung $X \Rightarrow^{n+1} w$ verwendete Regel hat die Form

$$X \rightarrow \gamma \quad \text{mit } \gamma \in \delta(\varepsilon, X).$$

Insgesamt haben wir dann

$$X \Rightarrow \gamma \Rightarrow^n w.$$

Dann haben γ und w die Form

$$\gamma = Y_1 \cdots Y_k, \quad w = u_1 \cdots u_k \quad \text{und} \quad Y_i \Rightarrow^* u_i \quad \text{für alle } i = 1, \dots, k.$$

Die Ableitung $Y_i \Rightarrow^* u_i$ kann höchstens aus n Schritten bestehen und daher können wir die Induktions-Voraussetzung anwenden und sehen, dass

$$\langle u_i, Y_i \rangle \vdash^* \langle \varepsilon, [] \rangle \quad \text{für alle } i = 1, \dots, k$$

gilt. Setzen wir diese Rechnungen zusammen, so haben wir

$$\langle u_1 \cdots u_k, Y_1 \cdots Y_k \rangle \vdash^* \langle \varepsilon, [] \rangle, \quad \text{also} \quad \langle w, \gamma \rangle \vdash^* \langle \varepsilon, [] \rangle.$$

Damit finden wir insgesamt

$$\langle w, X \rangle \vdash \langle w, \gamma \rangle \vdash^* \langle \varepsilon, [] \rangle$$

und wieder ist die Behauptung gezeigt.

Damit ist der Induktions-Beweis abgeschlossen.

Um den Beweis abzuschließen, zeigen wir nun, dass

$$L(A) \subseteq L(G(A))$$

gilt. Dazu zeigen wir, dass für alle $X \in \Gamma$ folgendes gilt:

$$\langle w, X \rangle \vdash^n \langle \varepsilon, [] \rangle \rightarrow X \Rightarrow^n w.$$

Diesen Beweis führen wir durch Induktion nach n .

I.A. $n = 1$. Dann ist w entweder das leere Wort oder w besteht aus einem einzigen Buchstaben $b \in \Sigma$. Wir untersuchen die beiden Fälle getrennt.

- (a) $w = b \in \Sigma$. Dann gilt $\varepsilon \in \delta(b, X)$. Nach Konstruktion der Grammatik $G(A)$ gibt es eine Regel

$$X \rightarrow b\varepsilon \quad \text{also} \quad X \rightarrow b.$$

Daraus folgt sofort $X \Rightarrow b = w$.

- (b) $w = \varepsilon$. Dann gilt $\varepsilon \in \delta(\varepsilon, X)$. Nach Konstruktion der Grammatik $G(A)$ gibt es dann eine Regel

$$X \rightarrow \varepsilon.$$

Daraus folgt sofort $X \Rightarrow \varepsilon = w$.

I.S. $n \mapsto n + 1$. Wieder gibt es zwei Fälle, die wir getrennt betrachten.

- (a) Die Rechnung $\langle w, X \rangle \vdash^n \langle \varepsilon, [] \rangle$ fängt mit dem Übergang $\gamma \in \delta(b, X)$ an. Dann muss $w = bv$ gelten und wir haben

$$\langle w, X \rangle = \langle bv, X \rangle \vdash \langle v, \gamma \rangle \vdash^n \langle \varepsilon, [] \rangle.$$

Dann hat γ die Form $\gamma = Y_1 \cdots Y_k$, v lässt sich zerlegen als $v = u_1 \cdots u_k$ und es ist u_i gerade der Teil des Wortes v , der gelesen wird, um die Variable Y_i vom Stack zu entfernen, es gilt also

$$\langle u_i, Y_i \rangle \vdash^* \langle \varepsilon, [] \rangle.$$

Diese Rechnung kann höchstens n Schritte benötigen. Also haben wir nach Induktions-Voraussetzung

$$Y_i \Rightarrow^* u_i \quad \text{für alle } i = 1, \dots, k.$$

Wegen $\gamma \in \delta(b, X)$ enthält die Grammatik $G(A)$ die Regel $X \rightarrow b\gamma$ und damit haben wir insgesamt

$$X \Rightarrow b\gamma = bY_1 \cdots Y_k \Rightarrow^* bu_1 \cdots u_k = bv = w.$$

- (b) Die Rechnung $\langle w, X \rangle \vdash^n \langle \varepsilon, [] \rangle$ fängt mit dem Übergang $\gamma \in \delta(\varepsilon, X)$ an. Dann haben wir

$$\langle w, X \rangle \vdash \langle w, \gamma \rangle \vdash^n \langle \varepsilon, [] \rangle.$$

Dann hat γ die Form $\gamma = Y_1 \cdots Y_k$, w lässt sich zerlegen als $w = u_1 \cdots u_k$ und es ist u_i gerade der Teil des Wortes w , der gelesen wird, um die Variable Y_i vom Stack zu entfernen, es gilt also

$$\langle u_i, Y_i \rangle \vdash^* \langle \varepsilon, [] \rangle.$$

Diese Rechnung kann höchstens n Schritte benötigen. Also haben wir nach Induktions-Voraussetzung wieder $Y_i \Rightarrow^* u_i$ für alle $i = 1, \dots, k$. Wegen $\gamma \in \delta(\varepsilon, X)$ enthält die Grammatik $G(A)$ die Regel $X \rightarrow \gamma$ und damit haben wir insgesamt

$$X \Rightarrow \gamma = Y_1 \cdots Y_k \Rightarrow^* u_1 \cdots u_k = w. \quad \square$$

Aufgabe 17: Der Keller-Automat A sei wie folgt definiert:

$$A = \langle \Sigma, \Gamma, \delta, S \rangle$$

Dabei gelte:

1. $\Sigma := \{\mathbf{a}, \mathbf{b}\}$.
2. $\Gamma := \{A, B, S\}$.
3. $\delta(\varepsilon, A) := \{\}$, $\delta(\varepsilon, B) := \{\}$, $\delta(\varepsilon, S) := \{ASA, BSB, A, B, \varepsilon\}$,
 $\delta(\mathbf{a}, A) := \{\varepsilon\}$, $\delta(\mathbf{a}, B) := \{\}$, $\delta(\mathbf{a}, S) := \{\}$,
 $\delta(\mathbf{b}, A) := \{\}$, $\delta(\mathbf{b}, B) := \{\varepsilon\}$, $\delta(\mathbf{b}, S) := \{\}$.

Verwenden Sie das im Beweis von Satz 25 verwendete Verfahren um eine Grammatik zu konstruieren, die die selbe Sprache beschreibt, die von dem Automaten $A(G)$ akzeptiert wird.

Kapitel 10

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.

10.1 Die Chomsky-Normalform

In diesem Abschnitt zeigen wir, wie eine kontextfreie Sprache in die sogenannte *Chomsky-Normalform* überführt werden kann. Die Überführung einer Sprache in Chomsky-Normalform erfolgt in mehreren Schritten.

1. Zunächst beseitigen wir *nutzlose Symbole*.

Ist $G = \langle V, T, R, S \rangle$ eine Grammatik, so nennen wir eine syntaktische Variable $A \in V$ *nützlich*, wenn es einen String $w \in T^*$ sowie Strings $\alpha, \beta \in (V \cup T)^*$ gibt, so dass

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* w$$

gilt. Eine syntaktische Variable ist also genau dann nützlich, wenn diese Variable in der Herleitung eines Wortes $w \in L(G)$ benötigt wird.

Ein Terminal t ist *nützlich*, wenn es Wörter $w_1, w_2 \in T^*$ gibt, so dass

$$S \Rightarrow^* w_1 t w_2.$$

Ein Terminal t ist also genau dann nützlich, wenn es in einem Wort der Sprache $L(G)$ auftritt.

Naheliegenderweise bezeichnen wir Variablen und Terminale, die nicht nützlich sind, als *nutzlose Symbole*.

2. Anschließend zeigen wir, dass alle ε -Regeln aus der Grammatik eliminiert werden können. Dabei bezeichnen wir eine Grammatik-Regel der Form

$$A \rightarrow \varepsilon$$

als eine ε -Regel.

3. Schließlich zeigen wir, wie alle Grammatik-Regeln der Form

$$A \rightarrow B$$

mit $A, B \in V$ eliminiert werden können. Solche Grammatik-Regeln werden als *Unit-Regeln* bezeichnet.

10.1.1 Beseitigung nutzloser Symbole

Die Erkennung nutzloser Symbole ist eine Überprüfung die in dem Werkzeug *Bison* eingebaut ist, weil das Auftreten nutzloser Symbole oft einen Hinweis darauf gibt, dass die Grammatik noch nicht die Sprache beschreibt, die spezifiziert werden soll. Insofern ist die jetzt vorgestellte Technik auch von praktischem Interesse. Wir beginnen mit zwei Definitionen.

Definition 26 (erzeugende Variable) Eine syntaktische Variable $A \in V$ einer Grammatik $G = \langle V, T, R, S \rangle$ ist eine *erzeugende Variable*, wenn es ein Wort $w \in T^*$ gibt, so dass

$$A \Rightarrow^* w$$

gilt, eine erzeugende Variable erzeugt also mindestens ein Wort aus T^* . Die Notation $A \Rightarrow^* w$ drückt dabei aus, dass aus der Variablen A in endlich vielen Schritten das Wort w abgeleitet werden kann. \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

$$A \rightarrow w \quad \text{mit } w \in T^*,$$

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 nutzlos sein. Als einfaches Beispiel betrachten wir die Grammatik $G = \langle \{S, A, B\}, \{x, y\}, R, S \rangle$, deren Regeln durch

$$S \rightarrow Ay$$

$$A \rightarrow AA \mid x$$

$$B \rightarrow Ay$$

gegeben sind. Die erzeugenden Variable sind in diesem Falle A , B und S . Die Variable B ist trotzdem nutzlos, denn von S aus ist diese Variable gar nicht *erreichbar*. Formal definieren wir für eine Grammatik $G = \langle V, T, R, S \rangle$ eine syntaktische Variable X als *erreichbar*, wenn es Wörter $\alpha, \beta \in (V \cup T)^*$ gibt, so dass gilt:

$$S \Rightarrow^* \alpha X \beta.$$

Für eine gegebene Grammatik G läßt sich die Menge der Variablen, die erreichbar sind, mit dem folgenden induktiven Algorithmus berechnen.

1. Das Start-Symbol S ist erreichbar.
2. Enthält die Grammatik G eine Regel der Form

$$X \rightarrow \alpha$$

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

$$\begin{aligned} S &\rightarrow BC \mid A \\ A &\rightarrow AA \mid x \\ B &\rightarrow y \\ C &\rightarrow CC \end{aligned}$$

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.

10.1.2 Beseitigung von ε -Regeln

Ist G_1 eine Grammatik, so dass die Sprache $L(G_1)$ den leeren String nicht enthält, so können wir eine Grammatik G_2 finden, die keine Regeln der Form

$$A \rightarrow \varepsilon$$

enthält und die die selbe Sprache beschreibt wie die Grammatik G_1 , es gilt also $L(G_1) = L(G_2)$. Regeln der Form $A \rightarrow \varepsilon$ bezeichnen wir als ε -Regeln. Bei bestimmten Algorithmen stören solche ε -Regeln, deswegen wollen wir jetzt ein Verfahren entwickeln, mit dem ε -Regeln eliminiert werden

können. Eine vollständige Formalisierung des Verfahrens würde die eigentliche Idee verschleiern, deswegen diskutieren wir das Verfahren an Hand eines Beispiels.

Beispiel: Die Regeln der im folgenden angegebenen Grammatik G_1 beschreiben arithmetische Ausdrücke, in denen Zahlen addiert oder subtrahiert werden. Wir verwenden dabei die syntaktischen Variablen $Expr$ und $ExprRest$ sowie das Terminal **NUMBER**:

$$\begin{array}{ll} Expr & \rightarrow \text{NUMBER } ExprRest \\ ExprRest & \rightarrow \begin{array}{l} "+" \text{ NUMBER } ExprRest \\ "-" \text{ NUMBER } ExprRest \\ \varepsilon \end{array} \end{array}$$

Eliminieren wir die ε -Regeln aus dieser Grammatik, so erhalten wir die folgende Grammatik:

$$\begin{array}{ll} Expr & \rightarrow \begin{array}{l} \text{NUMBER } ExprRest \\ \text{NUMBER} \end{array} \\ ExprRest & \rightarrow \begin{array}{l} "+" \text{ NUMBER } ExprRest \\ "+" \text{ NUMBER} \\ "-" \text{ NUMBER } ExprRest \\ "-" \text{ NUMBER} \end{array} \end{array}$$

Wir sehen, dass die Zahl der Grammatik-Regeln bei diesem Verfahren stark anwächst. Das Verfahren ist daher nicht von praktischem Interesse. Wir benötigen es lediglich zum späteren Nachweis der Tatsache, dass für alle kontextfreien Sprachen, die das leere Wort nicht enthalten, eine Grammatik in *Chomsky-Normalform* angegeben werden kann.

10.1.3 Beseitigung von Unit-Regeln

Regeln der Form

$$A \rightarrow B,$$

bei der die rechte Seite nur aus einer einzigen Variablen besteht, heißen *Unit-Regeln*. Diese Unit-Regeln können wie folgt eliminiert werden: Sind alle Regeln, bei denen die Variable B auf der linken Seite steht, durch

$$B \rightarrow \beta_1 \mid \dots \mid \beta_n$$

gegeben, so können wir die Regel $A \rightarrow B$ durch die Regeln

$$A \rightarrow \beta_1 \mid \dots \mid \beta_n$$

ersetzen und erhalten dabei eine Grammatik, die die selbe Sprache beschreibt.

Beispiel: Wir betrachten die folgende Grammatik für arithmetische Ausdrücke.

$$\begin{array}{ll} ArithExpr & \rightarrow \begin{array}{l} ArithExpr \text{ "+" } Product \\ Product \end{array} \\ Product & \rightarrow \begin{array}{l} Product \text{ "*" } Factor \\ Factor \end{array} \\ Factor & \rightarrow \begin{array}{l} "(" ArithExpr ")" \\ \text{NUMBER} \end{array} \end{array}$$

Diese Grammatik enthält die folgenden beiden Unit-Regeln:

$$\begin{array}{ll} ArithExpr & \rightarrow Product \\ Product & \rightarrow Factor \end{array}$$

Die Regel $Factor \rightarrow \text{NUMBER}$ ist keine Unit-Regel, denn **NUMBER** ist ein Terminal und keine syntaktische Variable.

Wir eliminieren zunächst die Unit-Regel $\text{ArithExpr} \rightarrow \text{Product}$. Wir erhalten dann die folgende Grammatik:

$$\begin{aligned}\text{ArithExpr} &\rightarrow \text{ArithExpr} \text{ "+" } \text{Product} \\ &\quad | \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{Factor} \\ \text{Product} &\rightarrow \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER}\end{aligned}$$

Hier haben wir eine neue Unit-Regel bekommen und zwar die Regel

$$\text{ArithExpr} \rightarrow \text{Factor}$$

Eliminieren wir diese Unit-Regel, so erhalten wir:

$$\begin{aligned}\text{ArithExpr} &\rightarrow \text{ArithExpr} \text{ "+" } \text{Product} \\ &\quad | \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER} \\ \text{Product} &\rightarrow \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{Factor} \\ \text{Factor} &\rightarrow \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER}\end{aligned}$$

Eliminieren wir nun die Unit-Regel $\text{Product} \rightarrow \text{Factor}$, so erhalten wir:

$$\begin{aligned}\text{ArithExpr} &\rightarrow \text{ArithExpr} \text{ "+" } \text{Product} \\ &\quad | \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER} \\ \text{Product} &\rightarrow \text{Product} \text{ "*" } \text{Factor} \\ &\quad | \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER} \\ \text{Factor} &\rightarrow \text{"(" ArithExpr ")" } \\ &\quad | \text{NUMBER}\end{aligned}$$

Wie man sieht, wächst die Grammatik durch die Eliminierung von Unit-Regeln stark an. Das oben skizzierte Verfahren kann sehr mühsam sein, wenn es einen *Zyklus* von Unit-Regeln gibt, wie das bei den folgenden Grammatik-Regeln der Fall ist:

$$\begin{aligned}A &\rightarrow x \mid B \\ B &\rightarrow y \mid C \\ C &\rightarrow z \mid A\end{aligned}$$

Eliminieren wir die Regel $A \rightarrow B$, so erhalten wir:

$$\begin{aligned}A &\rightarrow x \mid y \mid C \\ B &\rightarrow y \mid C \\ C &\rightarrow z \mid A\end{aligned}$$

Eliminieren wir nun die Regel $B \rightarrow C$, so erhalten wir:

$$\begin{aligned} A &\rightarrow x \mid y \mid C \\ B &\rightarrow y \mid z \mid A \\ C &\rightarrow z \mid A \end{aligned}$$

Eliminieren wir jetzt die Regel $C \rightarrow A$, so erhalten wir:

$$\begin{aligned} A &\rightarrow x \mid y \mid C \\ B &\rightarrow y \mid z \mid A \\ C &\rightarrow z \mid x \mid y \mid C \end{aligned}$$

Hier können wir die Regel $C \rightarrow C$ ersatzlos streichen und anschließend die Unit-Regel $A \rightarrow C$ eliminieren. Das liefert:

$$\begin{aligned} A &\rightarrow x \mid y \mid z \\ B &\rightarrow y \mid z \mid A \\ C &\rightarrow z \mid x \mid y \end{aligned}$$

Eliminieren wir hier noch die Unit-Regel $B \rightarrow A$, so erhalten wir:

$$\begin{aligned} A &\rightarrow x \mid y \mid z \\ B &\rightarrow y \mid z \mid x \\ C &\rightarrow z \mid x \mid y \end{aligned}$$

10.1.4 Überführung in Chomsky-Normalform

Definition 27 (Chomsky-Normalform) Eine Grammatik G ist in *Chomsky-Normalform* wenn alle Grammatik-Regeln eine der beiden folgenden Formen haben:

1. $A \rightarrow BC$, wobei B und C Nicht-Terminale sind, oder
2. $A \rightarrow x$, wobei x ein Terminal bezeichnet.

Außerdem darf die Grammatik keine nutzlosen Symbole haben. □

Noam Chomsky hat 1959 gezeigt [Cho59], dass jede Grammatik G , deren Sprache $L(G)$ das leere Wort nicht enthält, in Chomsky-Normalform überführt werden kann. Um dieses Verfahren durchführen zu können, führen wir zunächst die folgenden Schritte durch:

1. Wir eliminieren die ε -Regeln.
2. Danach entfernen wir die Unit-Regeln.
3. Schließlich beseitigen wir die nutzlosen Symbole.

Falls nun eine Grammatik-Regel die Form

$$A \rightarrow x$$

hat, so muss x ein Terminal sein, denn sonst hätten die Grammatik ja noch Unit-Regeln. Haben wir eine Grammatik-Regel der Form

$$A \rightarrow \alpha,$$

so können wir zunächst dafür sorgen, dass in α nur syntaktische Variablen auftreten, denn wenn wir eine Grammatik-Regel der Form

$$A \rightarrow \alpha t \beta$$

mit einem Terminal t haben, so führen wir eine neue syntaktische Variable C_t ein, die über die Grammatik-Regel

$$C_t \rightarrow t$$

definiert wird und ändern die Grammatik-Regel $A \rightarrow \alpha t \beta$ ab in die neue Regel

$$A \rightarrow \alpha C_t \beta.$$

Nach diesem Schritt haben alle Grammatik-Regeln die Form

$$A \rightarrow B_1 B_2 \cdots B_n,$$

wobei $n \geq 2$ ist und die Symbole B_1, \dots, B_n syntaktische Variablen sind. Sollte nun $n > 2$ sein, so führen wir eine neue syntaktische Variable C ein und ersetzen die obige Grammatik-Regel durch die beiden Grammatik-Regeln

$$A \rightarrow B_1 C \quad \text{und} \quad C \rightarrow B_2 \cdots B_n.$$

Offensichtlich können wir jede Grammatik durch Iterierung dieses Schrittes in Chomsky-Normalform bringen.

10.2 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

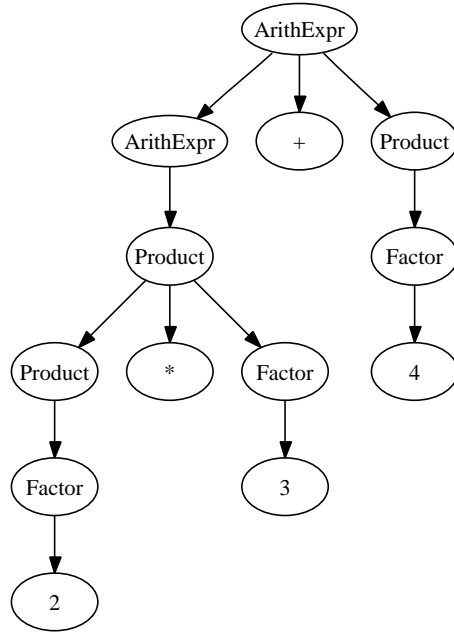


Abbildung 10.1: Ein Parse-Baum für den String “2*3+4”.

wird am ehesten an Hand eines Beispiels klar. Abbildung 10.1 zeigt einen Parse-Baum für den String “2*3+4”.

Fassen wir diesen Parse-Baum als Liste seiner Zweige auf, wobei jeder Zweig eine Liste von Grammatik-Symbolen ist, so erhalten wir die folgende Liste:

[[ArithExpr, ArithExpr, Product, Product, Factor, “2”],
 [ArithExpr, ArithExpr, Product, “*”],
 [ArithExpr, ArithExpr, Product, Factor, “3”],
 [ArithExpr, “+”],
 [ArithExpr, Product, Factor, “4”]
].

Die formale Definition eines Parse-Baums als Liste geben wir jetzt der Einfachheit halber nur für solche Grammatiken, die in Chomsky-Normalform sind.

Definition 28 (parseTree) Ist $G = \langle V, T, R, S \rangle$ eine Grammatik in Chomsky-Normalform, ist $w \in T^*$, $A \in V$ und gibt es eine Ableitung

$$A \Rightarrow^* w,$$

so definieren wir $parseTree(A \Rightarrow^* w)$ induktiv:

1. Falls $(A \rightarrow t) \in R$ mit $A \in V$ und $t \in T$, so setzen wir

$$parseTree(A \Rightarrow^* t) = [[A, t]].$$

2. Falls $A \Rightarrow^* w$ gilt, weil

$$(A \rightarrow BC) \in R, \quad B \Rightarrow^* w_1, \quad C \Rightarrow^* w_2 \quad \text{mit } w = w_1 w_2$$

gilt, so setzen wir

$$\begin{aligned} parseTree(A \Rightarrow^* w) &= [[A] + l \mid l \in parseTree(B \Rightarrow^* w_1)] \\ &+ [[A] + l \mid l \in parseTree(C \Rightarrow^* w_2)]. \end{aligned}$$

Lemma 29 Die Grammatik $G = \langle V, T, R, S \rangle$ sei in Chomsky-Normalform und es 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

$$|w| \leq 2^n.$$

Beweis: Wir führen den Beweis durch Induktion nach n .

I.A. $n = 2$: Dann besteht die Ableitung nur aus einem Schritt und daher muss es eine Regel der Form

$$A \rightarrow t \quad \text{mit } w = t \in T$$

in der Grammatik G geben. Es gilt dann

$$\text{parseTree}(A \Rightarrow^* w) = [[A, t]].$$

Daraus folgt

$$|w| = |t| = 1 = 2^0 \leq 2^2.$$

I.S. $n \mapsto n + 1$: Da die Ableitung nun aus mehr als einem Schritt besteht und die Grammatik in Chomsky-Normalform ist, hat die Ableitung die Form

$$A \Rightarrow BC \Rightarrow^* uv = w.$$

Außerdem haben die Listen in

$$\text{parseTree}(B \Rightarrow^* u) \quad \text{und} \quad \text{parseTree}(C \Rightarrow^* v)$$

dann eine Länge von höchstens n . Nach Induktions-Voraussetzung wissen wir also, dass

$$|u| \leq 2^n \quad \text{und} \quad |v| \leq 2^n$$

gilt. Daher haben wir

$$|w| = |uv| = |u| + |v| \leq 2^n + 2^n = 2^{n+1}. \quad \square$$

10.3 Das Pumping-Lemma für kontextfreie Sprachen

Satz 30 (Pumping-Lemma) Es sei L eine kontextfreie Sprache. Dann gibt es ein $n \in \mathbb{N}$, so dass jeder String $s \in L$, dessen Länge größer oder gleich n ist, in der Form

$$s = uvwxy$$

geschrieben werden kann, so dass außerdem folgendes gilt:

1. $|vwx| \leq n$,
der mittlere Teil des Strings hat also eine Länge von höchstens n Buchstaben.
2. $vx \neq \varepsilon$,
die Teilstrings v und x können also nicht beide gleichzeitig leer sein.
3. $\forall k \in \mathbb{N} : uv^kwx^ky \in L$.

Dieser Bedingung verdankt das Pumping-Lemma seinen Namen, denn die Strings v und x können beliebig oft *gepumpt* werden. \square

Beweis: Wir zeigen den Satz zunächst für den Fall, dass die Sprache L das leere Wort nicht enthält. Dazu konstruieren wir mit dem im letzten Abschnitt gezeigten Verfahren eine Grammatik $G = \langle V, T, R, S \rangle$ in Chomsky-Normalform, so dass

$$L = L(G)$$

gilt. Wir nehmen an, dass die Grammatik G insgesamt m syntaktische Variablen enthält, also haben wir $m := |V|$ und definieren

$$n := 2^{m+2}.$$

Sei nun $s \in L$ mit $|s| \geq n$. Wir betrachten die Listen aus

$$\text{parseTree}(S \Rightarrow^* s).$$

Falls alle Listen hier eine Länge kleiner oder gleich $m + 1$ hätten, so würde aus Lemma 29 folgen, dass

$$|s| \leq 2^{m+1} < n$$

ist, im Widerspruch zu der Voraussetzung $|s| \geq n$. Also muss es in $\text{parseTree}(S \Rightarrow^* s)$ eine Liste geben, die mindestens die Länge $m + 1$ hat. Diese Liste hat dann die Form

$$[A_1, \dots, A_k, t] \quad \text{mit } A_i \in V \text{ für alle } i \in \{1, \dots, k\}, \quad t \in T, \quad \text{sowie } k \geq m + 1.$$

Wegen $k \geq m + 1$ können nicht alle Variablen A_1, \dots, A_k voneinander verschieden sein, denn es gibt ja nur insgesamt m verschiedene syntaktische Variablen. Wir finden daher in der Menge $\{k, k - 1, \dots, k - m\}$ zwei Indices i, j mit $i \neq j$ und $A_i = A_j =: A$. Ohne Beschränkung der Allgemeinheit gelte $i < j$. Die Ableitung von s hat dann die folgende Form

$$S \Rightarrow^* uA_iy \Rightarrow^* uvA_jxy \Rightarrow^* uvwxy = s.$$

Insbesondere gilt also

$$S \Rightarrow^* uAy, \quad A \Rightarrow^* vAx, \quad \text{und} \quad A \Rightarrow^* w.$$

Damit haben wir dann aber folgendes:

1. $S \Rightarrow^* uAy \Rightarrow^* uwy$, also

$$S \Rightarrow^* uv^0wx^0y.$$

2. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxy \Rightarrow^* uvvwxxy$, also

$$S \Rightarrow^* uv^2wx^2y.$$

3. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \Rightarrow^* uv^3Ax^3y \Rightarrow^* \dots \Rightarrow^* uv^kAx^ky \Rightarrow^* uv^kwx^ky$.

Wir müssen jetzt noch zeigen, dass $vx \neq \varepsilon$ gilt. Die Ableitung

$$A \Rightarrow^* vAx$$

ist eigentlich die Ableitung

$$A_i \Rightarrow^* vA_jx$$

und enthält daher mindestens einen Ableitungsschritt. Wir führen den Nachweis indirekt und nehmen $v = x = \varepsilon$ an. Da die Grammatik G nach Voraussetzung in Chomsky-Normalform ist, hat der erste Schritt dieser Ableitung dann die Form

$$A \Rightarrow BC \Rightarrow^* A.$$

Dann muss aber entweder

$$B \Rightarrow^* \varepsilon \quad \text{oder} \quad C \Rightarrow^* \varepsilon$$

gelten und das geht nicht, da eine Grammatik in Chomsky-Normalform keine ε -Regeln enthält.

Als nächstes zeigen wir, dass die Ungleichung $|vwx| \leq n$ gilt. Wir haben

$$A = A_i \Rightarrow^* vA_jx \Rightarrow^* vwx.$$

Wegen $i \in \{k-m, k-(m-1), \dots, k\}$, wissen wir, dass die Länge der längsten Liste in

$$\text{parseTree}(A \Rightarrow^* vwy)$$

kleinergleich $m+2$ ist. Nach dem Lemma 29 folgt damit für die Länge von vwy die Abschätzung

$$|vwy| \leq 2^{m+2} = n.$$

Zum Schluß müssen wir noch festlegen, was in dem Fall zu tun ist, wenn $\varepsilon \in L$ ist. In diesem Fall betrachten wir einfach die Sprache $L' := L \setminus \{\varepsilon\}$. Diese Sprache ist kontextfrei, denn wir können aus der Grammatik der Sprache L einfach alle ε -Regeln mit dem im letzten Abschnitt gezeigten Verfahren entfernen. Die resultierende Grammatik beschreibt dann die Sprache L' . Für die Sprache L' haben wir das Pumping-Lemma oben bewiesen. Wir müssen nun für das n , das wir oben finden, nur fordern, dass es größer als 0 ist, denn dann erfüllt das leere Wort die Voraussetzung des Pumping-Lemmas $|s| \geq n$ nicht und damit gilt das Pumping-Lemma dann auch für alle Wörter aus L . \square

10.4 Anwendungen des Pumping-Lemmas

Wir zeigen nun, wie mit Hilfe des Pumping-Lemmas der Nachweis erbracht werden kann, dass bestimmte Sprachen nicht kontextfrei sind.

10.4.1 Die Sprache $L = \{a^k b^k c^k \mid k \in \mathbb{N}\}$

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

$$uvw \in L.$$

Wir definieren nun den String s wie folgt:

$$s := a^n b^n c^n.$$

Dieser String hat die Länge $3 \cdot n$ und erfüllt damit die Voraussetzung über die Länge. Damit finden wir also eine Zerlegung $s = uvwxy$ mit den obigen Eigenschaften. Da der Teilstring vw eine Länge kleiner oder gleich n hat, können in diesem String nicht gleichzeitig die Buchstaben “a” und “c” vorkommen. Wir betrachten diese Fälle getrennt.

1. Fall: In dem String vwx kommt der Buchstabe “c” 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 18: Zeigen Sie, dass die Sprache

$$L = \{ \mathbf{a}^{n^2} \mid n \in \mathbb{N} \}$$

nicht kontextfrei ist.

Hinweis: Argumentieren Sie über die Länge der betrachteten Strings.

Kapitel 11

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 immer das linkeste Nicht-Terminal expandiert. Die Zahl k in LL(k) bedeutet, dass der Parser immer an Hand der nächsten k Token entscheidet, welche Regel als verwendet wird. Falls $k = 1$ ist, wird also nur das nächste Token zur Entscheidung herangezogen. Dieses Token bezeichnen wir dann als *Look-Ahead-Token*. Um die Darstellung übersichtlich zu halten, beschränken wir uns auf die Theorie der LL(1)-Parser.

Bevor wir die Theorie der LL(1)-Parser darstellen, machen wir auf zwei Probleme aufmerksam, die wir bei der Erstellung von Top-Down-Parsern lösen müssen.

1. Das erste Problem ist Links-Rekursion. Betrachten wir die dazu die folgenden Grammatik-Regeln für arithmetische Ausdrücke:

$$\begin{array}{lcl} \text{Expr} & \rightarrow & \text{Expr} \text{ "+" } \text{Product} \\ & | & \text{Expr} \text{ "-" } \text{Product} \\ & | & \text{Product} \end{array}$$

Die ersten beiden Regeln sind links-rekursiv und daher mit einem Top-Down-Parser in der vorliegenden Form nicht verwendbar. Wir hatten dieses Problem bereits im Abschnitt 6.2.1 besprochen und gezeigt, wie die Links-Rekursion in einer Grammatik eliminiert werden kann.

2. Das zweite Problem erkennen wir, wenn wir die folgende Grammatik für Boole'sche Ausdrücke betrachten:

$$\begin{array}{lcl} \text{boolExpr} & \rightarrow & \text{expr} \text{ "==" } \text{expr} \\ & | & \text{expr} \text{ "<" } \text{expr} \end{array}$$

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 nun ein Verfahren vor, mit dem sich Probleme der Form, dass zwei Grammatik-Regeln den gleichen Anfang haben, lösen lassen.

11.1 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 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 11.1 zeigt die vollständige Grammatik.

<i>boolExpr</i>	\rightarrow	<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 11.1: Grammatik für arithmetische Ausdrücke.

Es ist nicht möglich einen LL(1)-Parser zu implementieren, der Boole'sche Ausdrücke nach diesen beiden Regeln erkennt, denn beide Alternativen fangen mit einer *expr* an. Wir können die Grammatik aber durch *Links-Faktorisierung* (engl. *left factoring*) so umschreiben, das ein Token als Look-Ahead ausreicht, indem wir den Teil aus den beiden Grammatik-Regeln ausklammern, der am Anfang der beiden Regeln identisch ist. In dem obigen Beispiel führen wir dann für den verbleibenden Rest das neue Nicht-Terminal *boolExprRest* ein und erhalten die Regeln

$$\begin{aligned} \text{boolExpr} &\rightarrow \text{expr } \text{boolExprRest} \\ \text{boolExprRest} &\rightarrow \begin{array}{l} \text{“==” } \text{expr} \\ \text{“<” } \text{expr.} \end{array} \end{aligned}$$

Mit diesen Regeln reicht nun ein Token als Look-Ahead aus, denn die beiden Alternativen für *boolExprRest* unterscheiden sich in dem ersten Token des Rumpfs der Regel.

Um den allgemeinen Fall der Links-Faktorisierung diskutieren zu können, nehmen wir an, dass *A* ein Nicht-Terminal ist, das durch insgesamt $m + n$ Regeln definiert wird, wobei der Rumpf der ersten m Regeln immer mit α anfängt, wobei α ein String aus Terminalen und Nicht-Terminalen ist. Die Regeln haben also die folgende Form:

$$\begin{array}{lcl}
A & \rightarrow & \alpha \beta_1 \\
& | & \alpha \beta_2 \\
& & \vdots \\
& | & \alpha \beta_m \\
& | & \gamma_1 \\
& & \vdots \\
& | & \gamma_n
\end{array}$$

Bei dieser Darstellung sei vorausgesetzt, dass die Strings β_1, \dots, β_m keinen gemeinsamen Präfix haben und dass α auch kein Präfix einer der Strings γ_i ist. Bei der Links-Faktorisierung dieser Regeln klammern wir einerseits den gemeinsamen Präfix α aus und führen andererseits eine neue syntaktische Variable B ein, die den auf α folgenden Rest bezeichnet. Wir erhalten dann die folgenden Regeln:

$$\begin{array}{lcl}
A & \rightarrow & \alpha B \\
& | & \gamma_1 \\
& & \vdots \\
& | & \gamma_n
\end{array}
\qquad
\begin{array}{lcl}
B & \rightarrow & \beta_1 \\
& | & \beta_2 \\
& & \vdots \\
& | & \beta_m
\end{array}$$

Um alle gemeinsamen Präfixe auszuklammern muss dieses Verfahren unter Umständen mehrfach durchgeführt werden. Die nächste Aufgabe gibt dafür ein Beispiel.

Aufgabe 19: Geben Sie eine Links-Faktorisierung für die folgenden Grammatik-Regeln an.

$$\begin{array}{lcl}
A & \rightarrow & \text{"a"} \text{"b"} U \text{"d"} \\
& | & \text{"a"} V \text{"b"} \text{"d"} \\
& | & \text{"a"} \text{"b"} W \\
& | & \text{"x"} U \\
& | & \text{"x"} V
\end{array}$$

Lösung: Zunächst eliminieren wir das gemeinsame Präfix "a" und führen dazu die neue syntaktische Variable B ein. Wir erhalten:

$$\begin{array}{lcl}
A & \rightarrow & \text{"a"} B \\
& | & \text{"x"} U \\
& | & \text{"x"} V
\end{array}
\qquad
\begin{array}{lcl}
B & \rightarrow & \text{"b"} U \text{"d"} \\
& | & V \text{"b"} \text{"d"} \\
& | & \text{"b"} W
\end{array}$$

Nun eliminieren wir das Präfix "x" aus beiden letzten Regeln für A . Dabei erhalten wir:

$$\begin{array}{lcl}
A & \rightarrow & \text{"a"} B \\
& | & \text{"x"} C
\end{array}
\qquad
\begin{array}{lcl}
C & \rightarrow & U \\
& | & V
\end{array}
\qquad
\begin{array}{lcl}
B & \rightarrow & \text{"b"} U \text{"d"} \\
& | & V \text{"b"} \text{"d"} \\
& | & \text{"b"} W
\end{array}$$

Als letztes eliminieren wir das Präfix “b”, das in zwei der Regeln für die syntaktische Variable B auftritt. Wir erhalten das Ergebnis:

$$\begin{array}{lcl}
 A & \rightarrow & \text{“a” } B \\
 & | & \text{“x” } C \\
 C & \rightarrow & U \\
 & | & V \\
 B & \rightarrow & \text{“b” } D \\
 & | & V \text{ “b” “d”} \\
 D & \rightarrow & U \text{ “d”} \\
 & | & W
 \end{array}$$

□

Bemerkung: Der Parser-Generator ANTLR führt automatisch für jede Grammatik eine Links-Faktorisierung durch. Es ist daher bei Verwendung von ANTLR nicht notwendig, die Grammatik zu faktorisieren. Andere Parser-Generatoren, beispielsweise *JavaCC*, sind nicht so komfortabel. Dort ist der Benutzer gezwungen, die Links-Faktorisierung von Hand durchzuführen.

11.1.1 First und Follow

Nicht für jede links-faktorierte Grammatik läßt sich ein LL(1)-Parser bauen. Betrachten wir die folgenden Regeln:

$$\begin{array}{lcl}
 A & \rightarrow & B \mid C \\
 B & \rightarrow & \text{“a” } U \\
 C & \rightarrow & \text{“a” } V
 \end{array}$$

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 31 (ε -erzeugend) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik und A sei eine syntaktische Variable, also $A \in V$. Dann heißt die Variable A ε -erzeugend genau dann, wenn

$$A \Rightarrow^* \varepsilon$$

gilt, also dann, wenn sich aus der Variablen A das leere Wort ableiten läßt. Wir schreiben *nullable(A)* wenn die Variable A als ε -erzeugend nachgewiesen ist. □

Beispiele:

1. Bei der in Abbildung 11.1 auf Seite 140 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 ist nun aufgrund der Regel

$$S \rightarrow A B C$$

selbst S ε -erzeugend.

Definition 32 (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 11.1 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

$$expr \rightarrow product \ exprRest$$

und der Tatsache, dass *product* nicht ε -erzeugend ist, dass die $First(expr)$ mit der Mengen $First(product)$ übereinstimmt:

$$First(expr) = \{ "(", NUMBER, IDENTIFIER \}.$$

6. Schließlich folgt aus den beiden Regeln

$$boolExpr \rightarrow expr \ "==" \ expr \quad \text{und} \quad boolExpr \rightarrow expr \ "<" \ expr$$

sowie der Tatsache, dass *expr* nicht ε -erzeugend ist, dass $First(boolExpr)$ mit $First(expr)$ identisch ist:

$$First(boolExpr) = \{ "(", NUMBER, IDENTIFIER \}.$$

□

Definition 33 (*Follow()*) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik und $A \in V$. Bei der Berechnung von *Follow()* wird die Grammatik zunächst abgeändert, indem wir das Symbol "\$" als neues Symbol zu der Menge T der Terminale hinzufügen. Zu den Variablen wird das neue Symbol \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:

$$\hat{S} \rightarrow S \ "$"$$

Das Terminal "\$" steht hierbei für das Ende der Eingabe (EOF, *end of file*). Weiter definieren wir

$$\hat{T} := T \cup \{ "\$" \}.$$

Dann definieren wir $Follow(A)$ als die Menge aller der Token t , die in einer Ableitung auf A folgen können:

$$Follow(A) := \{ t \in \hat{T} \mid \exists \beta, \gamma \in (V \cup \hat{T})^* : \hat{S} \Rightarrow^* \beta A t \gamma \}.$$

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 $Follow(A)$. □

Beispiel: Wir untersuchen wieder die in Abbildung 11.1 gezeigte Grammatik für arithmetische Ausdrücke.

1. Aufgrund der neu hinzugefügten Regel

$$\hat{S} \rightarrow boolExpr \ "$"$$

muss die Menge $Follow(boolExpr)$ das Zeichen "\$" enthalten. Da die syntaktische Variable *boolExpr* sonst nirgends in der Grammatik vorkommt, haben wir

$$Follow(boolExpr) = \{ "\$" \}.$$

2. Die beiden Grammatik-Regeln

$$boolExpr \rightarrow expr \ "==" \ expr \quad \text{und} \quad boolExpr \rightarrow expr \ "<" \ expr$$

zeigen uns zunächst, dass die Menge $Follow(expr)$ die Zeichen "==" und "<" enthält. Da *expr* auch am Ende dieser Regeln steht, folgt weiter, dass alle Elemente aus $Follow(boolExpr)$ auch auf *expr* folgen können, wir haben also auch

$$"\$" \in Follow(expr).$$

Aufgrund der Regel

$$factor \rightarrow "(" \ expr \ ")"$$

muss die Menge $Follow(expr)$ außerdem das Zeichen ")" enthalten. Also haben wir insgesamt

$$\text{Follow}(\text{expr}) = \{ "=", "<", "\$", ")" \}.$$

3. Aufgrund der Regel

$$\text{expr} \rightarrow \text{product exprRest}$$

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 *expr* 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

$$\text{Follow}(\text{factor}) = \{ "*", "/", "+", "-", "=", "<", "\$", ")" \}. \quad \square$$

Das letzte Beispiel zeigt, dass die Berechnung des Prädikats *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 *nullable()* und den beiden Funktionen $\text{First}()$ und $\text{Follow}()$ die folgenden Beziehungen:

1. $\forall t \in T : \neg \text{nullable}(t).$
2. $k = 0 \Rightarrow \text{nullable}(A) = \text{true}.$
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.

Die Abbildungen 11.2, 11.3 und 11.4 zeigen die Implementierung dreier Methoden zur Implementierung des Prädikats *nullable()* und der Funktionen *First()* und *Follow()*.

Die Berechnung der ε -erzeugenden Variablen

```

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 11.2: Die Methode *nullable()*.

Wir diskutieren zunächst die Berechnung aller ε -erzeugenden Variablen in der in Abbildung 11.2 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 11.2 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.

Die Berechnung der Funktion *First()*

Abbildung 11.3 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.

```

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 11.3: Die Methode *computeFirst()*.

Die Berechnung der Funktion *Follow()*

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. $(\forall j \in \{i + 1, \dots, k\} : \text{nullable}(Y_j)) \Rightarrow \text{Follow}(A) \subseteq \text{Follow}(Y_i)$.

Diese Beziehung wird in den Zeilen 9 bis 22 umgesetzt. Beachten Sie, dass die Beziehung

$$\text{Follow}(A) \subseteq \text{Follow}(Y_k).$$

ein Spezialfall dieses Falls ist, der von der Implementierung ebenfalls mit abgedeckt wird.

2. $(\forall j \in \{i + 1, \dots, l - 1\} : \text{nullable}(Y_j) \wedge Y_l \in T) \Rightarrow Y_l \in \text{Follow}(Y_i)$

Diese Beziehung wird in den Zeilen 33 bis 39 umgesetzt.

3. $(\forall j \in \{i + 1, \dots, l - 1\} : \text{nullable}(Y_j) \wedge Y_l \in V) \Rightarrow \text{First}(Y_l) \subseteq \text{Follow}(Y_i)$

Diese Beziehung wird in den Zeilen 39 bis 45 umgesetzt.

```

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     }
25 }

```

Abbildung 11.4: Die Methode *computeFollow()*, erster Teil.

11.1.2 LL(1)-Grammatiken

Wir können nun die Frage beantworten, für welche Grammatiken ein Top-Down-Parser erzeugt werden kann, der immer mit einem Token Look-Ahead auskommt.

Definition 34 (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 Rümpfe zweier verschiedener Regeln der selben Variablen dürfen nicht beide das leere Wort ableiten.

2. $\text{First}(\alpha) \cap \text{First}(\beta) = \{\}$.

Die Ableitungen der Rümpfe zweier verschiedener Regeln der selben Variablen dürfen nicht mit dem selben Token beginnen.

3. $(\beta \Rightarrow^* \varepsilon) \rightarrow \text{First}(\alpha) \cap \text{Follow}(A) = \{\}$.

Wenn β den leeren String ableitet, dann müssen die Mengen $\text{First}(\alpha)$ und $\text{Follow}(A)$ disjunkt sein. \square

Wir diskutieren nun die Idee, die hinter der obigen Definition steht.

1. Falls das leere Wort sowohl über die Regel

$$A \rightarrow \alpha \quad \text{als auch über} \quad B \rightarrow \beta$$

```

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 11.5: Die Methode *computeFollow()*, zweiter Teil.

ableitbar wäre, so wissen wir nicht, welche Regel wir anwenden sollen, wenn für die Variable A dass leere Wort abgeleitet werden muss.

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 Look-Ahead-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 Look-Ahead-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 Look-Ahead-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

$$(\beta \Rightarrow^* \varepsilon) \rightarrow First(\alpha) \cap Follow(A) = \{\}.$$

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 Look-Ahead-Token.

1. $lat \in First(\alpha)$ oder
2. $\alpha \Rightarrow^* \varepsilon$ und $lat \in Follow(\alpha)$.

```

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         for (MyToken t : first) {
17             Set<SimpleRule> ruleSet = mParseTable.get(head).get(t);
18             ruleSet.add(r);
19         }
20         if (allNullable(body, 0, body.size() - 1)) {
21             Set<MyToken> follow = mFollow.get(head);
22             enter(follow, head, r);
23         }
24     }
25 }

```

Abbildung 11.6: 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 $parseTable(A, t)$:

$$(A \rightarrow \alpha) \in R \wedge \alpha \Rightarrow^* \varepsilon \wedge t \in Follow(A) \Rightarrow (A \rightarrow \alpha) \in parseTable(A, t).$$

Abbildung 11.6 zeigt eine Berechnung der Parse-Tabelle.

1. Die **for**-Schleife in den Zeilen 2 bis 10 initialisiert $parseTable(A, t)$ mit der leeren Menge.
2. In den Zeilen fügen wir die Regel $A \rightarrow \alpha$ für alle Token t aus der Menge $First(\alpha)$ in $parseTable(A, t)$ ein.
3. Falls $\alpha \Rightarrow^* \varepsilon$ gilt, so fügen wir in den Zeilen 20 bis 22 für jedes Token t aus $Follow(A)$ die Regel $A \rightarrow \alpha$ in die Menge $parseTable(A, t)$ ein.

Eine Grammatik ist genau dann eine $LL(1)$ -Grammatik, wenn die Mengen $parseTable(A, t)$ für jede syntaktische Variable A und jedes Token t maximal eine Regel enthält:

$$G \text{ ist } LL(1) \quad \text{g.d.w.} \quad \forall A \in V : \forall t \in T : card(parseTable(A, t)) \leq 1.$$

Falls die Menge $parseTable(A, t)$ 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.

11.1.3 LL(k)-Grammatiken

Viele interessante Grammatiken sind keine $LL(1)$ -Grammatiken. Abbildung 11.7 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

$$factor \rightarrow IDENTIFIER \quad \text{und} \quad factor \rightarrow IDENTIFIER "(" argList ")"$$

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

$$factor \rightarrow IDENTIFIER "(" argList ")"$$

heranzuziehen, andernfalls muss die Regel

$$factor \rightarrow IDENTIFIER$$

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 11.7: Arithmetische Ausdrücke mit Funktions-Aufrufen.

Im allgemeinen Fall kann das Verfahren so erweitert werden, dass k Token bei der Entscheidung, welche Regel zu verwenden ist, als Lookahead herangezogen werden. Bei ANTLR ist es beispielsweise möglich, den Lookahead k zu spezifizieren. Die Praxis zeigt, dass die so erzeugten Parser sehr wohl zu Parsern, die von Bottom-Up-Parser-Generatoren wie *Bison* oder *CUP* erzeugt werden, konkurrenzfähig sind. Wir skizzieren die Grundzüge dieser Theorie. Als erstes verallgemeinern wir

die Definition der Funktion $First()$.

Definition 35 ($First(k, \alpha)$) 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 11.1 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 36 ($Follow(k, A)$) 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 37 (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

$$\forall \sigma, \tau \in \text{Follow}(k, A) : \text{First}(k, \beta\sigma) \cap \text{First}(k, \gamma\tau) = \{\}$$

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 $\text{Follow}(k, A)$ sind. Für eine Regel

$$A \rightarrow \beta$$

und einen String $\sigma \in \text{Follow}(k, A)$ enthält die Menge $\text{First}(k, \beta\sigma)$ 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.

Kapitel 12

Behandlung von Nicht-LL(1)-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.

12.1 Unterstützung von $LL(*)$ -Sprachen

Wir werden in diesem Abschnitt zunächst den Begriff der $LL(*)$ -Sprachen definieren. Hierbei handelt es sich um eine Sprachenklasse, die wesentlich mächtiger ist, als die Klasse der $LL(k)$ -Sprachen. Der Parser-Generator ANTLR [Par07] kann für alle $LL(*)$ -Sprachen einen Parser erzeugen und ist damit wesentlich mächtiger als andere Top-Down-Parser-Generatoren wie beispielsweise *JavaCC*¹, die nur für Sprachen der Klasse $LL(k)$ Parser erzeugen können. Zunächst zeigen wir an Hand praktischer Beispiele die Unzulänglichkeit der Klasse $LL(k)$, anschließend geben wir eine formale Definition der Klasse $LL(*)$.

12.1.1 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:

```
void hugo(int a, int b);
```

Eine Definition dieser Funktion könnte die folgende Form haben:

```
void hugo(int a, int b) { return 17; }
```

¹<https://javacc.dev.java.net/>

```

1  grammar decl_or_def;
2
3  options {                // Suppress these lines
4      k = 8;                // to let ANTLR create
5  }                          // an LL(*)-parser.
6
7  decl_or_def
8      : type ID '(' args ')' ';'
9      | type ID '(' args ')' '{' body '}'
10     ;
11
12  type: 'void'
13      | 'int'
14      ;
15
16  args: arg (',' arg)* ;
17
18  arg : 'int' ID ;
19
20  body: 'return' INT;
21
22  ID : ('A'..'Z' 'a'..'z')+;
23  INT: ('1'..'9') ('0'..'9')*;

```

Abbildung 12.1: Eine Grammatik für Deklarationen und Definitionen

Abbildung 12.1 zeigt eine (vereinfachte) Grammatik in ANTLR-SYNTAX, die sowohl Deklarationen als auch Definitionen von Funktionen zuläßt. Das Start-Symbol dieser Grammatik ist `decl_or_def`. Beide Regeln zur Ableitung der syntaktischen Variable `decl_or_def` beginnen mit der Token-Folge

type ID '('.

Daher ist klar, dass wir mindestens 3 Token Look-Ahead-Token benötigen, um zwischen einer Deklaration und einer Definition unterscheiden zu können. In der oben gezeigten ANTLR-Spezifikation der Grammatik ist für den Look-Ahead k der Wert 8 spezifiziert worden. Lassen wir von ANTLR einen Parser für diese Grammatik erzeugen, so erhalten wir die folgende Fehlermeldung:

```

ANTLR Parser Generator Version 3.0 (May 17, 2007) 1989-2007
warning(200): method.g:8:7: Decision can match input such as

```

```

    "'void'..'int' ID '(' 'int' ID ',' 'int' ID"

```

```

using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input

```

An dieser Stelle können wir erkennen, dass die durch die angegebene Grammatik spezifizierte Sprache keine $LL(k)$ -Sprache ist, denn ob eine Deklaration oder eine Definition einer Funktion vorliegt, können wir erst dann erkennen, wenn wir das Zeichen hinter der schließenden Klammer `)` sehen:

1. Falls dieses Zeichen ein Semikolon ist, liegt eine Deklaration vor.
2. Falls das Zeichen eine öffnende geschweifte Klammer ist, haben wir die Definition einer Funktion.

Da nun zwischen den runden Klammern beliebig viele Argumente kommen können, können wir keinen festen Wert k für den Look-Ahead angeben, der ausreicht, um die Entscheidung zwischen den beiden Alternativen der Grammatik anzugeben. Kommentieren wir die Zeilen 3 bis 5 aus, so kann ANTLR einen Parser erzeugen, denn ohne Spezifikation von k versucht ANTLR automatisch, einen $LL(*)$ -Parser an Stelle eines $LL(k)$ -Parsers zu konstruieren, was in dem vorliegenden Fall auch gelingt.

```

1  grammar decl_or_def_factor;
2
3  options {
4      k = 1;
5  }
6
7  decl_or_def_factor
8      : type ID '(' args ')' ';' | '{' body '}'
9      ;
10
11  type: 'void'
12      | 'int'
13      ;
14
15  args: arg (',' arg)* ;
16
17  arg : 'int' ID ;
18
19  body: 'return' INT;
20
21  ID : ('A'..'Z' 'a'..'z')+;
22  INT: ('1'..'9') ('0'..'9')*;

```

Abbildung 12.2: Eine faktorisierte Grammatik für Deklarationen und Definitionen

Wir können das oben dargestellte Problem auch durch Faktorisierung lösen. Dadurch, dass *Antlr* die Möglichkeit bietet, auf der rechten Seite einer Regel Klammern zu setzen, bleibt die resultierende Grammatik sehr übersichtlich. Abbildung 12.2 zeigt, wie sich das konkrete Problem durch Faktorisierung lösen läßt.

Klassen versus Schnittstellen

Abbildung 12.3 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 12.3 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, aber die einfachste Möglichkeit ist wieder, *Antlr* einen $LL(*)$ -parser erzeugen zu lassen. Diese erreichen wir, in dem wir

```

1  grammar class_or_interface;
2
3  options {
4      k = 1;
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 12.3: Grammatik für Klassen und Schnittstellen.

```
options { k = 1; }
```

durch

```
options { k = *; }
```

ersetzen, oder aber die Spezifikation der Look-Ahead-Tiefe k ganz weglassen.

12.1.2 Die Theorie der LL(*)-Sprachen

Die Grundidee, um für Sprachen wie beispielsweise die in Abbildung 12.1 spezifizierte Sprache einen Parser zu entwickeln besteht darin, einen deterministischen endlichen Automaten zu erzeugen, dessen Aufgabe es ist, von den beiden 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 12.4 zeigt die Definition eines deterministischen endlichen Automaten, der zwischen Funktionen und Deklarationen unterscheidet.

Die Interpretation der Zustände ist wie folgt:

1. Zustand 0 ist der Start-Zustand.
2. Im Zustand 1 hat der Automat ein 'void' oder 'int' gelesen.
3. Im Zustand 2 hat der Automat zusätzlich den Namen der Funktion gelesen.
4. Im Zustand 3 hat der Automat zusätzlich eine öffnende Klammer gelesen.

Menge der Zustände:	$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.
Start-Zustand:	0
Zustands-Übergänge:	$\delta(0, \text{'void'}) = 1$ $\delta(0, \text{'int'}) = 1$ $\delta(1, \text{ID}) = 2$ $\delta(2, \text{'('}) = 3$ $\delta(3, \text{'int'}) = 4$ $\delta(4, \text{ID}) = 5$ $\delta(5, \text{' ,'}) = 3$ $\delta(5, \text{' '}) = 6$ $\delta(6, \text{' ;'}) = 7$ $\delta(6, \text{'{'}) = 8$
Akzeptierende Zustände:	7 (Deklaration) 8 (Definition)

Abbildung 12.4: Deterministischer endlicher Automat zur Unterscheidung zwischen Deklarationen und Definitionen

5. \vdots
6. Im Zustand 5 gibt es die Möglichkeit, entweder weitere Argumente zu lesen, oder aber eine schließende Klammer. Der erste Fall tritt ein, wenn ein `' '` gelesen wird, der zweite Fall liegt vor, wenn das nächste Zeichen eine schließende Klammer ist.
7. Im Zustand 6 fällt die Entscheidung: Wird ein Semikolon gelesen, so geht der Automat in den Zustand 7, bei einer öffnenden geschweiften Klammer geht der Automat in den Zustand 8.
8. Folglich sind die Zustände 7 und 8 die akzeptierenden Zustände des Automaten. Im Zustand 7 wird anschließend der Parser die Regel

`decl_or_def -> type ID '(' args ')' ';' ;`

verwenden, im Zustand 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 lesen kann.

Formale Definition der LL(*)-Sprachen

Es $G = \langle V, T, R, S \rangle$ ein kontext-freie Grammatik. Für das Start-Symbol S dieser Grammatik gibt es mehrere Grammatik-Regeln und unser Ziel ist es, einen endlichen Automaten zu entwickeln, der die richtige Grammatik-Regel für S auswählen kann. Die verschiedenen Regeln, in deren Kopf das Start-Symbol S steht, haben die Form

$$\begin{array}{lcl}
 S & \rightarrow & alt_1 \\
 S & \rightarrow & alt_2 \\
 & \vdots & \\
 S & \rightarrow & alt_n
 \end{array}$$

Hierbei bezeichnet alt_i die rechte Seite der i -ten Regel. Diese rechte Seite ist eine Liste von Token und syntaktischen Variablen. Unser Ziel ist es, einen deterministischen endlichen Automaten zu entwickeln, der in der Lage ist, bei einem gegebenen Eingabe-String die richtige Alternative auszuwählen, mit der dieser String geparkt werden kann. Dazu werden wir zunächst einen nicht-deterministischen Automaten entwickeln, den wir dann über die im letzten Semester diskutierte Teilmengen-Konstruktion in einen deterministischen endlichen Automaten transformieren können.

Normalerweise ist die Grammatik, für die wir einen endlichen Automaten entwickeln, nur ein kleiner Teil einer größeren Grammatik, die eventuell weitere Regeln und ein anderes Start-Symbol enthält. Diese größere Grammatik enthalte eine Regel mit mehreren Alternativen, für die wir eine Auswahl treffen wollen. Wir lassen dann in der größeren Grammatik alle Variablen und Regeln weg, die nichts mit der Variablen, für die wir die Auswahl treffen wollen, zu tun haben.

Die Zustände des zu entwickelnden Automaten sind Paare der Form

$$\langle k, l \rangle \quad \text{mit } k \in \mathbb{N} \text{ und } l \in \{0, 1, \dots, n\}.$$

Die Idee ist hier, dass wir die Zustände zunächst mit natürlichen Zahlen k durchnummerieren. Die zweite Zahl l dient dann dazu zu spezifizieren, welche Alternative unter den n Regeln ausgewählt werden soll. Die Zahl l , die wir auch als *Index* des Zustands bezeichnen, hat den Wert 0 wenn die Entscheidung in dem betreffenden Zustand noch nicht gefallen ist.

Zu Beginn unseres Algorithmus ordnen wir jeder von S verschiedenen syntaktischen Variablen einen Start- und einen Endzustand zu. Diese Zustände müssen alle verschieden sein und haben den Index 0. Bezeichnen wir die Menge der Zustände mit Z , so haben wir also eine Funktion

$$start : V \setminus \{S\} \rightarrow Z$$

die jeder vom Start-Symbol S verschiedenen syntaktischen Variablen x den Start-Zustand $start(x)$ zuordnet sowie eine Funktion

$$end : V \setminus \{S\} \rightarrow Z$$

die jeder vom Start-Symbol S verschiedenen syntaktischen Variablen x den End-Zustand $end(x)$ zuordnet. Desweiteren definieren wir den Start-Zustand (willkürlich) durch

$$s_0 := \langle 0, 0 \rangle.$$

Als nächstes betrachten wir alle Regeln

$$X \rightarrow Y_1 \cdots Y_k$$

und führen für jede solche Regel neue Zustände z_0, z_1, \dots, z_k ein. Falls es sich bei der obigen Regel um die Regel

$$S \rightarrow alt_i$$

handelt, indizieren wir diese Zustände mit i und zeichnen den letzten Zustand als akzeptierenden Zustand aus. Andernfalls werden die Zustände alle mit 0 indiziert. Die Idee ist, dass der Automat im Zustand z_0 ist, wenn er ein x parsen soll, aber bisher noch keine Token verarbeitet hat. Ist der Automat im Zustand z_i , so wurden bereits

$$y_1 \cdots y_i$$

erkannt. Wir notieren dann die Regel mit den Zuständen in der Form

$$X \xrightarrow{z_0} Y_1 \xrightarrow{z_1} Y_2 \xrightarrow{z_2} \cdots \xrightarrow{z_{k-1}} Y_k \xrightarrow{z_k}.$$

Jetzt gibt es zwei Fälle.

1. Falls Y_i eine syntaktische Variable ist, so definieren wir für die Zustands-Übergangs-Funktion

$$\delta(z_{i-1}, \varepsilon) = start(Y_i) \quad \text{und} \quad \delta(end(Y_i), \varepsilon) = z_i \quad \text{für alle } i = 1, \dots, k.$$

Die Interpretation ist dabei Folgende: Will der Automat ein x unter Anwendung der Grammatik-Regel

$$X \rightarrow Y_1 \cdots Y_k$$

erkennen und hat schon $Y_1 \cdots Y_{i-1}$ erkannt, so geht der Automat anschließend in den Zustand, in dem er ein Y_i erkennen kann. Hat er dann ein Y_i erkannt, so kann er aus diesem Zustand in den Zustand z_{i+1} wechseln, in dem er als nächstes ein Y_{i+1} zu erkennen versucht.

2. Falls Y_i ein Token ist, definieren wir naheliegenderweise

$$\delta(z_{i-1}, Y_i) = z_i.$$

Weiter setzen wir

$$\delta(\text{start}(X), \varepsilon) = z_0 \quad \text{und} \quad \delta(z_k, \varepsilon) = \text{end}(X).$$

Die erste Gleichung drückt aus, dass der Automat von dem Zustand, in dem er ein X erkennen soll, unmittelbar in den Zustand z_0 wechseln kann, in dem er dann die rechte Seite $Y_1 \cdots Y_k$ der Regel $X \rightarrow Y_1 \cdots Y_k$ erkennen soll. Die zweite Gleichung drückt aus, dass der Automat, falls er die rechte Seite $Y_1 \cdots Y_k$ der Regel $X \rightarrow Y_1 \cdots Y_k$ erkannt hat, natürlich auch ein x erkannt hat.

Auch für die Alternativen der Regel

$$\begin{array}{ll} S & \rightarrow \text{alt}_1 \\ S & \rightarrow \text{alt}_2 \\ & \vdots \\ S & \rightarrow \text{alt}_n \end{array}$$

werden für jede dieser Regeln und jeden Punkt innerhalb einer der Alternativen Zustände z_j eingeführt, die alle mit dem Index i indiziert sind. Den jeweils letzten Zustand der i -ten Regel bezeichnen wir mit a_i . Diese Zustände sind die *akzeptierenden* Zustände des endlichen Automaten. Damit ist die Konstruktion des nicht-deterministischen endlichen Automaten abgeschlossen. \square

Bemerkung: Da wir im letzten Abschnitt einen nicht-deterministischen Automaten konstruiert haben, ist δ keine Funktion, sondern eine Relation. Damit ist die Schreibweise

$$\delta(z, a) = u$$

eigentlich inkorrekt. Mathematisch korrekt, aber umständlicher wäre statt dessen die Notation

$$\langle z, a, u \rangle \in \delta.$$

Wir verwenden trotzdem die Schreibweise $\delta(z, a) = u$, da diese suggestiver ist. Wir sollten uns allerdings klar machen, dass für zwei Zustände z_1 und z_2 durchaus

$$\delta(z_0, \varepsilon) = z_1, \quad \delta(z_0, \varepsilon) = z_2 \quad \text{und} \quad z_1 \neq z_2$$

gelten kann. \square

Der nächste Schritt besteht darin, aus dem nicht-deterministischen Automaten über die Teilmengen-Konstruktion einen deterministischen Automaten zu erzeugen. Der Start-Zustand des deterministischen Automaten besteht jetzt aus der Menge aller Zustände, die durch ε -Übergänge von dem Start-Zustand s_0 des nicht-deterministischen Automaten erreicht werden können. Ist weiter M ein Zustand des deterministischen Automaten, so ist M eine Menge von Zuständen des nicht-deterministischen Automaten. Die Zustands-Übergangs-Funktion Δ des deterministischen Automaten definieren wir durch

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

wobei die Funktion δ^* als

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

definiert ist. Dabei bezeichnet $\text{ec}(q)$ die Menge aller Zustände, die vom Zustand q durch ε -Übergänge erreichbar sind.

Wir bezeichnen nun eine Menge M von Zuständen des nicht-deterministischen Automaten als *homogen* genau dann, wenn alle Zustände in dieser Menge den selben Index haben. Eine

Grammatik ist eine $LL(*)$ -Grammatik genau dann, wenn jede Menge des deterministischen endlichen Automaten, in der ein akzeptierender Zustand des nicht-deterministischen Automaten liegt, homogen ist. In diesem Fall haben also alle Zustände der Menge den selben Index, den wir dann auch als den Index des deterministischen Zustands bezeichnen. Abschließend definieren wir: Eine Sprache L ist eine $LL(*)$ -Sprache, wenn es für L eine $LL(*)$ -Grammatik gibt.

Haben wir eine $LL(*)$ -Grammatik und haben gemäß der obigen Konstruktion einen Automaten erzeugt, so lassen wir den endlichen Automaten so lange laufen, bis ein entscheidender Zustand erreicht worden ist. Der Index dieses Zustands gibt dann den Index der Regel an, welche beim Parsen zu verwenden ist. \square

Beispiel: Wir illustrieren die Definition an einem Beispiel. Wir greifen dazu die Grammatik aus dem ersten Abschnitt dieses Kapitels wieder auf, die für die Sprache \mathbf{C} Funktions-Deklarationen und Funktions-Definitionen beschreibt. Wir vereinfachen diese Grammatik und lassen die Typ-Bezeichner weg. Dann erhalten wir die Abbildung 12.5 gezeigte Grammatik.

```

1  simple
2      : ID '(' args ')' ';'
3      | ID '(' args ')' '{' '}'
4      ;
5
6  args: ID
7      | ID ',' args
8      ;
9
10 ID : ('A'..'Z' 'a'..'z')+;

```

Abbildung 12.5: Grammatik für Deklarationen und Definitionen ohne Typen

Wir konstruieren zunächst den nicht-deterministischen endlichen Automaten gemäß dem oben angegebenen Verfahren. Der Start-Zustand ist $\langle 0, 0 \rangle$ und die Definition der weiteren Zustände erfolgt, indem wir diese an die Grammatik-Regeln wie folgt annotieren:

1. $simple \rightarrow_{\langle 0,1 \rangle} ID_{\langle 1,1 \rangle} \text{ "(" }_{\langle 2,1 \rangle} args_{\langle 3,1 \rangle} \text{ ")" }_{\langle 4,1 \rangle} \text{ ";" }_{\langle 5,1 \rangle}$
2. $simple \rightarrow_{\langle 0,2 \rangle} ID_{\langle 1,2 \rangle} \text{ "(" }_{\langle 2,2 \rangle} args_{\langle 3,2 \rangle} \text{ ")" }_{\langle 4,2 \rangle} \text{ "{" }_{\langle 5,2 \rangle} \text{ "}" }_{\langle 6,2 \rangle}$
3. Für die syntaktische Variable $args$ definieren wir
 $start(args) = \langle 7, 0 \rangle$ und $end(args) = \langle 14, 0 \rangle$.
4. $args \rightarrow_{\langle 8,0 \rangle} ID_{\langle 9,0 \rangle}$
5. $args \rightarrow_{\langle 10,0 \rangle} ID_{\langle 11,0 \rangle} \text{ "," }_{\langle 12,0 \rangle} args_{\langle 13,0 \rangle}$

Damit ist die Menge der Zustände des nicht-deterministischen Automaten wie folgt:

$$\{ \langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 2, 1 \rangle, \langle 3, 1 \rangle, \langle 4, 1 \rangle, \langle 5, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 2 \rangle, \langle 2, 2 \rangle, \langle 3, 2 \rangle, \langle 4, 2 \rangle, \langle 5, 2 \rangle, \langle 6, 2 \rangle, \langle 7, 0 \rangle, \langle 8, 0 \rangle, \langle 9, 0 \rangle, \langle 10, 0 \rangle, \langle 11, 0 \rangle, \langle 12, 0 \rangle, \langle 13, 0 \rangle, \langle 14, 0 \rangle \}$$

Die beiden akzeptierenden Zustände sind die Zustände

$\langle 5, 1 \rangle$ und $\langle 6, 2 \rangle$.

Als nächstes geben wir die Zustands-Übergangsfunktion δ an:

1. $\delta(\langle 0, 0 \rangle, \varepsilon) = \langle 0, 1 \rangle$
2. $\delta(\langle 0, 0 \rangle, \varepsilon) = \langle 0, 2 \rangle$
3. $\delta(\langle 0, 1 \rangle, \text{ID}) = \langle 1, 1 \rangle$
4. $\delta(\langle 1, 1 \rangle, "(") = \langle 2, 1 \rangle$
5. $\delta(\langle 2, 1 \rangle, \varepsilon) = \langle 7, 0 \rangle$
6. $\delta(\langle 14, 0 \rangle, \varepsilon) = \langle 3, 1 \rangle$
7. $\delta(\langle 3, 1 \rangle, ")") = \langle 4, 1 \rangle$
8. $\delta(\langle 4, 1 \rangle, ";") = \langle 5, 1 \rangle$
9. $\delta(\langle 0, 2 \rangle, \text{ID}) = \langle 1, 2 \rangle$
10. $\delta(\langle 1, 2 \rangle, "(") = \langle 2, 2 \rangle$
11. $\delta(\langle 2, 2 \rangle, \varepsilon) = \langle 7, 0 \rangle$
12. $\delta(\langle 14, 0 \rangle, \varepsilon) = \langle 3, 2 \rangle$
13. $\delta(\langle 3, 2 \rangle, ")") = \langle 4, 2 \rangle$
14. $\delta(\langle 4, 2 \rangle, "{") = \langle 5, 2 \rangle$
15. $\delta(\langle 5, 2 \rangle, "}") = \langle 6, 2 \rangle$
16. $\delta(\langle 7, 0 \rangle, \varepsilon) = \langle 8, 0 \rangle$
17. $\delta(\langle 7, 0 \rangle, \varepsilon) = \langle 10, 0 \rangle$
18. $\delta(\langle 8, 0 \rangle, \text{ID}) = \langle 9, 0 \rangle$
19. $\delta(\langle 9, 0 \rangle, \varepsilon) = \langle 14, 0 \rangle$
20. $\delta(\langle 10, 0 \rangle, \text{ID}) = \langle 11, 0 \rangle$
21. $\delta(\langle 11, 0 \rangle, ",") = \langle 12, 0 \rangle$
22. $\delta(\langle 12, 0 \rangle, \varepsilon) = \langle 7, 0 \rangle$
23. $\delta(\langle 14, 0 \rangle, \varepsilon) = \langle 13, 0 \rangle$
24. $\delta(\langle 13, 0 \rangle, \varepsilon) = \langle 14, 0 \rangle$

Damit ist der nicht-deterministische Automat vollständig definiert. Wir berechnen nun den zugehörigen deterministischen Automaten.

1. Der Startzustand ist $S_0 := \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle\}$.
2. $S_1 := \delta(S_0, \text{ID}) = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle\}$.
3. $S_2 := \delta(S_1, "(") = \{\langle 2, 1 \rangle, \langle 2, 2 \rangle\}$.
4. $S_3 := \delta(S_2, \varepsilon) = \{\langle 7, 0 \rangle, \langle 8, 0 \rangle, \langle 10, 0 \rangle\}$.
5. $S_4 := \delta(S_3, \text{ID}) = \{\langle 9, 0 \rangle, \langle 14, 0 \rangle, \langle 13, 0 \rangle, \langle 11, 0 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle\}$.
6. $S_5 := \delta(S_4, ";") = \{\langle 12, 0 \rangle, \langle 7, 0 \rangle, \langle 8, 0 \rangle, \langle 10, 0 \rangle\}$.
7. $\delta(S_5, \text{ID}) = S_4$.

8. $S_6 := \delta(S_4, \text{"}) = \{\langle 4, 1 \rangle, \langle 4, 2 \rangle\}.$
9. $S_7 := \delta(S_6, \text{";"}) = \{\langle 5, 1 \rangle\}.$
10. $S_8 := \delta(S_6, \text{"{"}) = \{\langle 5, 2 \rangle\}.$
11. $S_9 := \delta(S_8, \text{"}) = \{\langle 6, 2 \rangle\}.$

Jetzt sehen wir, dass alle Zustands-Mengen, die einen akzeptierenden Zustand des nicht-deterministischen Automaten enthalten, *homogen* sind. Die Zustände S_7 und S_9 sind homogen und die Sprache ist folglich eine LL(*)-Sprache. \square

12.1.3 Behandlung von LL(*)-Grammatiken in Antlr

Das im letzten Abschnitt beschriebene Verfahren beschreibt die Theorie der LL(*)-Parser. Diese Theorie wird von ANTLR allerdings nicht so umgesetzt, wie das im letzten Abschnitt beschrieben ist. Es gibt bei der Übersetzung von LL(*)-Grammatiken eine Einschränkung, die wir jetzt diskutieren.

Probleme bei rekursiven Grammatik-Regeln

Ändern wir die EBNF-Grammatik aus Abbildung 12.1 wie in Abbildung 12.6 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.
```

```
1  grammar decl_or_def;
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 12.6: Rekursive Grammatik für Deklarationen und Definitionen

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.

Expansion von Grammatik-Regeln

Die Regeln der Grammatik $G = \langle \{A, B\}, \{ \text{“u”}, \text{“x”}, \text{“y”}, \text{“z”} \}, R, A \rangle$ seien wie folgt:

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

Diese Grammatik ist, wie Sie mit einer länglichen Rechnung selbst überprüfen können, keine $LL(*)$ -Grammatik. Die Grammatik lässt sich aber durch Substitution der Regel

$$B \rightarrow \text{"u"}$$

in eine $LL(*)$ Grammatik umformen: Ersetzen wir in den rechten Seiten der Regeln, die die Variable A ableiten, die Variable B durch die rechte Seite der Regel für B , so erhalten wir die Grammatik

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

Diese Grammatik ist nun eine $LL(*)$ -Grammatik. ANTLR führt Substitutionen der oben beschriebenen Art so weit wie möglich automatisch durch und kann daher auch solche Grammatiken behandeln, die keine $LL(*)$ -Grammatiken im Sinne der im letzten Abschnitt gegebenen Definitionen sind. Das Verfahren der Substitutionen funktioniert allerdings dann nicht mehr, wenn die Grammatik rekursiv ist. In diesem Fall expandiert ANTLR alle Variablen bis zu einer gewissen Tiefe. Diese Tiefe ist auf 4 voreingestellt und kann mit Hilfe der Kommandozeilen-Option `-Xm` auf einen anderen Wert gesetzt werden. Die Tatsache, dass es bei einer rekursiven Grammatik nicht möglich ist, alle Variablen zu expandieren, ist der Grund, dass ANTLR solche Grammatiken nicht als $LL(*)$ -Grammatiken erkennen kann.

12.2 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 12.7: 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  int (uint);
2
3  typedef unsigned int uint;
4
5  uint(x);

```

Abbildung 12.7: Typ-Bezeichner und Variablen sind syntaktisch nicht unterscheidbar!

Wir betrachten nun eine Grammatik, mit der sich Deklarationen, Typ-Definitionen und einfa-

che Befehle in C beschreiben lassen. Wir haben die Grammatik so weit wie möglich vereinfacht um das Problem der Mehrdeutigkeit klarer herausheben zu können. Abbildung 12.8 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.

```

1  grammar cGrammar;
2
3  declOrType
4      : decl
5        | typeDef
6        | stmnt
7      ;
8
9  decl: type identifier
10     | type '(' identifier ')'
11     ;
12
13  typeDef: TYPEDEF type identifier;
14
15  stmnt: identifier '(' identifier ')' ;
16
17  type: UNSIGNED
18       | INT
19       | UNSIGNED INT
20       | typeid
21     ;
22
23  typeid: ID;
24
25  identifier: ID;
26
27  TYPEDEF : 'typedef';
28  INT     : 'int';
29  UNSIGNED: 'unsigned';
30  ID      : ('a'..'z') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
31
32  WS      : (' ' | '\t' | '\n' | '\r') { skip(); };

```

Abbildung 12.8: Eine Grammatik für Deklarationen und Befehle.

Übersetzen wir diese Grammatik mit ANTLR, so erhalten wir die folgende Warnung:

```
warning(200): cGrammar.g:4:5: Decision can match input such as "ID '(' ID ')'"
using multiple alternatives: 1, 3
As a result, alternative(s) 3 were disabled for that input
error(201): cGrammar.g:4:5: The following alternatives can never be matched: 3
```

ANTLR hat die Mehrdeutigkeit der Grammatik korrekt erkannt und liefert auch gleich das Beispiel, das auf zwei verschiedene Arten ableitbar ist.

```

1  grammar cGrammar2;
2
3  @header {
4      import java.util.Set;
5      import java.util.TreeSet;
6  }
7
8  @members {
9      Set<String> mTypeNames = new TreeSet<String>();
10 }
11
12 declOrType
13     : decl
14     | typeDef
15     | stmtnt
16     ;
17
18 decl: type identifier
19     | type '(' identifier ')'
20     ;
21
22 typeDef: TYPEDEF type identifier { mTypeNames.add($identifier.text); };
23
24 stmtnt: identifier '(' identifier ')' ;
25
26 type: UNSIGNED
27     | INT
28     | UNSIGNED INT
29     | typeid
30     ;
31
32 typeid: { mTypeNames.contains(input.LT(1).getText()); }? ID;
33
34 identifier: ID;
35
36 TYPEDEF : 'typedef';
37 INT     : 'int';
38 UNSIGNED: 'unsigned';
39 ID      : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
40
41 WS      : (' '|'\t'|\n'|\r') { skip(); };

```

Abbildung 12.9: Eine Grammatik mit semantischen Prädikat.

Abbildung 12.9 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 von `getText()`. Wir überprüfen, ob der String in der Menge der bereits definierten Typ-Bezeichner liegt. Die Grammatik-Regel

$typeid \rightarrow ID$

wird nun nur dann angewendet, wenn dies der Fall ist. Andernfalls wird ein ID mit Hilfe der Regel

$identifier \rightarrow ID$

geparst.

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.

12.3 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 nicht auftritt.

12.3.1 Das Dangling-Else-Problem

Als motivierendes Beispiel betrachten wir das sogenannte *Dangling-Else-Problem*, das in der in Abbildung 12.10 auf Seite 171 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.

```

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 12.10: Fragment einer Grammatik für die Sprache C

Um das Problem zu lösen, gibt es mehrere Möglichkeiten:

1. Wir können die Grammatik so umschreiben, dass die Mehrdeutigkeit verschwindet.

Die entsprechend umgeschriebene Grammatik ist allerdings fast doppelt so groß und erheblich komplexer als die ursprüngliche Grammatik und dadurch schwerer verständlich. Wir werden diesen Weg daher in diesem Kapitel nicht weiterverfolgen.

2. Wir können das in ANTLR verfügbare Backtracking aktivieren.
3. Schließlich können wir das Problem durch Verwendung eines syntaktischen Prädikats lösen.

Lösung des Dangling-Else-Problem durch Backtracking

Abbildung 12.11 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.

Lösung des Dangling-Else-Problem durch ein syntaktisches Prädikat

Abbildung 12.12 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 *syntaktisches Prädikat*. Bevor der Parser versucht, ein *expr* mit der Regel

$$stmtnt \rightarrow \text{"if"} \text{ "(" } boolExp \text{ ")" } stmtnt \text{ "else"} stmtnt$$

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.

Die allgemeine form eines syntaktischen Prädikats ist

$$\begin{array}{lcl} A & \rightarrow & \text{"(" } \alpha_1 \text{ ")}=> \beta_1 \\ & | & \text{"(" } \alpha_2 \text{ ")}=> \beta_2 \\ & \vdots & \\ & \vdots & \text{"(" } \alpha_{n-1} \text{ ")}=> \beta_{n-1} \\ & \vdots & \text{"(" } \alpha_n \text{ ")}=> \beta_n \end{array}$$

Hierbei sind α_i und β_i für $i = 1, \dots, n$ 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.

```

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 12.11: Lösen des Dangling-Else-Problems durch Backtracking.

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 ist lediglich über die Option einzuschalten, dass 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 dadurch keine unbeabsichtigten Ambivalenzen der Grammatik verschleiert werden.

```

1  grammar danglingSyntactic;
2
3  prog  : stmt+;
4
5  stmt  : ('if' '(' boolExp ') ' stmt 'else' stmt)?=>
6          'if' '(' boolExp ') ' stmt 'else' stmt
7          | 'if' '(' boolExp ') ' stmt
8          | 'while' '(' boolExp ') ' stmt
9          | '{' stmt* '}'
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 12.12: Lösen des Dangling-Else-Problems durch ein syntaktisches Prädikat.

Kapitel 13

Interpreter

Ziel dieses Kapitels ist zu zeigen wie einfach es ist, mit Hilfe eines Parser-Generators einen Interpreter für eine Programmiersprache zu erstellen. Abbildung 13.2 zeigt die Grammatik einer einfachen Programmier-Sprache. Die Befehle dieser Sprache sind Zuweisungen, Print-Anweisungen, `if`-Abfragen, sowie `while`-Schleifen. Abbildung 13.1 zeigt ein Beispiel-Programm, das dieser Grammatik entspricht.

```
1  s = 0;
2  i = 0;
3  while (i < 6 * 6) {
4      i = i + 1;
5      if (0 < i) {
6          print(i);
7          print(s);
8      }
9      s = s + i;
10 }
11 print(s);
```

Abbildung 13.1: Ein Programm zur Berechnung der Summe $\sum_{i=0}^n i$.

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 dient dazu, Anweisungen unserer Programmier-Sprache darzustellen. Wir werden von der Klasse **Statement** später die Klassen **Assignment**, **Print**, **IfThen** und **While** ableiten. Abbildung 13.3 zeigt die Implementierung der abstrakten Klasse **Statement**. Die Klasse enthält lediglich die Deklaration einer abstrakten Methode `execute()`. Diese Methode können wir benutzen, um einzelne Befehle ausführen.

Abbildung 13.4 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

<i>stmntList</i>	→	<i>statement stmntList</i>
		ε
<i>statement</i>	→	<i>variable</i> “=” <i>expr</i> “;”
		“print” “(” <i>expr</i> “)” “;”
		“if” “(” <i>boolExpr</i> “)” “{” <i>stmntList</i> “}”
		“while” “(” <i>boolExpr</i> “)” “{” <i>stmntList</i> “}”
<i>boolExpr</i>	→	<i>expr</i> “==” <i>expr</i>
		<i>expr</i> “<” <i>expr</i>
<i>expr</i>	→	<i>expr</i> “+” <i>product</i>
		<i>expr</i> “-” <i>product</i>
		<i>product</i>
<i>product</i>	→	<i>product</i> “*” <i>factor</i>
		<i>product</i> “/” <i>factor</i>
		<i>factor</i>
<i>factor</i>	→	“(” <i>expr</i> “)”
		<i>variable</i>
		<NUMBER>
<i>variable</i>	→	<IDENTIFIER>

Abbildung 13.2: Grammatik für eine einfache Programmier-Sprache.

```

1  public abstract class Statement {
2      public abstract void execute();
3  }

```

Abbildung 13.3: Die abstrakte Klasse **Statement**

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 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 gespeichert sind. Die Tabelle ist als statische Variable in der Klasse **Expr** definiert.

```

1  public class Assignment extends Statement {
2      Variable mLhs;
3      Expr     mRhs;
4
5      public Assignment(Variable lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public void execute() {
10         Expr.sValueTable.put(mLhs.mName, mRhs.eval());
11     }
12     public String toString() {
13         return mLhs + " = " + mRhs + ";";
14     }
15 }

```

Abbildung 13.4: Die Klasse Assignment.

```

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 stmtnt: mStmntList) {
14                 stmtnt.execute();
15             }
16         }
17     }
18     public String toString() {
19         String result = "while (" + mCond + ") {\n";
20         for (Statement stmtnt: mStmntList) {
21             result += stmtnt + "\n";
22         }
23         result += "}";
24         return result;
25     }
26 }

```

Abbildung 13.5: Die Klasse While.

Von den übrigen Klassen zur Darstellung von Befehlen diskutieren wir noch die Klasse `While`, die in Abbildung 13.5 gezeigt wird. Diese Klasse stellt einen Befehl der Form

`while (s) { stmts }`

dar, wobei b ein Boole'scher Ausdruck ist, während *stmts* 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 `mStmtList`. Zur Auswertung eines solchen Befehls führen wir solange alle Befehle in der Liste `mStmtList` 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     public String toString() {
16         return mLhs + " == " + mRhs;
17     }
18 }
```

Abbildung 13.6: 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 13.6 zeigt die Klassen `BoolExpr` und `Equal`. 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 13.7: 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 13.7 gezeigt ist.

1. Die Klasse definiert die statische Variable **sValueTable**. Diese Variable beinhaltet eine Hash-tabelle, in der für jede Variable, der ein Wert zugewiesen wurde, der aktuelle Wert dieser Variablen gespeichert ist.
2. Weiter deklariert die Klasse die abstrakte Methode *eval()*, mit der ein Ausdruck ausgewertet werden kann.

Von der Klasse **Expr** werden die Klassen **Sum**, **Difference**, **Product**, **Quotient**, **MyNumber** und **Variable** abgeleitet. Abbildung 13.8 zeigt die Klasse **Sum**. Da diese Klasse eine Summe der Form

$$l + r$$

darstellt, hat diese Klasse zwei Member-Variablen **mLhs** und **mRhs** um die beiden Summanden *l* und *r* darzustellen. Die Methode *eval* wertet diese beiden Member-Variablen getrennt aus und addiert das Ergebnis. Die Klassen **Difference**, **Product** und **Quotient** sind analog zur Klasse **Sum** aufgebaut und werden daher nicht weiter diskutiert.

```
1  public class Sum extends Expr {
2      Expr mLhs;
3      Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Double eval()    {
10         return mLhs.eval() + mRhs.eval();
11     }
12     public String toString() {
13         return mLhs.toString() + " + " + mRhs.toString();
14     }
15 }
```

Abbildung 13.8: Die Klasse **Sum**

Abbildung 13.9 zeigt die Implementierung der Klasse **Variable**. Die Methode *eval()* wertet eine Variable dadurch aus, dass sie unter dem Namen der Variablen in der Hash-tabelle **sValueTable** den zugeordneten Wert nachschlägt.

```

1  public class Variable extends Expr {
2      String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public Double eval() {
8          return sValueTable.get(mName);
9      }
10     public String toString() {
11         return mName;
12     }
13 }

```

Abbildung 13.9: Die Klasse Variable.

```

1  import org.antlr.runtime.*;
2  import java.util.List;
3
4  public class SLInterpreter {
5
6      public static void main(String[] args) throws Exception {
7          ANTLRInputStream input = new ANTLRInputStream(System.in);
8          simpleLanguageLexer lexer = new simpleLanguageLexer(input);
9          CommonTokenStream ts = new CommonTokenStream(lexer);
10         simpleLanguageParser parser = new simpleLanguageParser(ts);
11         List<Statement> program = parser.simpleLanguage();
12         System.out.println("\nProgram listing:\n");
13         for (Statement stmt: program) {
14             System.out.println(stmt);
15         }
16         System.out.println("\nProgram trace:\n");
17         for (Statement stmt: program) {
18             stmt.execute();
19         }
20     }
21 }

```

Abbildung 13.10: Die Klasse SLInterpreter.

Abbildung 13.10 zeigt das Treiber-Programm, das den von ANTLR erzeugten Parser einbindet. Nachdem zunächst die Standardeingabe in einen für ANTLR passenden Eingabe-Strom umgeformt wird, kann in Zeile 8 der Lexer erzeugt werden, der dann den vom Parser benötigten Token-Strom bereit stellt. Das eigentliche Programm steht in Zeile 11 als Liste von Statements zur Verfügung. Dieses wird in den Zeilen 12 – 15 zunächst ausgegeben und in Zeile 17 – 19 werden die Befehle dann der Reihe nach abgearbeitet.

Abbildung 13.11 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR. Da ANTLR die sogenannte *erweiterte Backus-Naur-Form* (EBNF) bei der Spezifikation der Grammatik unterstützt, konnte die Grammatik wie in Abbildung 13.12 gezeigt vereinfacht werden. Dies war auch deswegen erforderlich, weil die ursprüngliche Grammatik Links-Rekursion enthielt. Beach-

```

22  grammar simpleLanguage;
23
24  @header {
25      import java.util.List;
26      import java.util.ArrayList;
27  }
28
29  simpleLanguage returns [List<Statement> program]
30      : { $program = new ArrayList<Statement>(); }
31      (s = statement { program.add($s.stmnt); })+
32      ;
33  statement returns [Statement stmnt]
34      : v = VAR '=' e = expr ';'
35      { $stmnt = new Assignment(new Variable($v.text), $e.result); }
36      | 'print' '(' r = expr ')' ';'
37      { $stmnt = new Print($r.result); }
38      | { List<Statement> stmnts = new ArrayList<Statement>(); }
39      'if' '(' b = boolExpr ')' '{'
40      (l = statement { stmnts.add($l.stmnt); })* '}'
41      { $stmnt = new IfThen($b.result, stmnts); }
42      | { List<Statement> stmnts = new ArrayList<Statement>(); }
43      'while' '(' b = boolExpr ')' '{'
44      (l = statement { stmnts.add($l.stmnt); })* '}'
45      { $stmnt = new While($b.result, stmnts); }
46      ;
47  boolExpr returns [BoolExpr result]
48      : l = expr ( '=' r = expr { $result = new Equal( $l.result, $r.result); }
49      |
50      | '<' r = expr { $result = new LessThan($l.result, $r.result); }
51      )
52      ;
53  expr returns [Expr result]
54      : p = product { $result = $p.result; }
55      (
56      ('+' q = product) { $result = new Sum( $result, $q.result); }
57      | ('-' q = product) { $result = new Difference($result, $q.result); }
58      ) *
59      ;
60  product returns [Expr result]
61      : f = factor { $result = $f.result; }
62      (
63      ('*' g = factor) { $result = new Product( $result, $g.result); }
64      | ('/' g = factor) { $result = new Quotient($result, $g.result); }
65      ) *
66      ;
67  factor returns [Expr result]
68      : '(' expr ')' { $result = $expr.result; }
69      | v = VAR { $result = new Variable($v.text); }
70      | n = NUMBER { $result = new MyNumber($n.text); }
71      ;

```

Abbildung 13.11: ANTLR-Spezifikation der Grammatik.

ten Sie, dass die EBNF-Grammatik wesentlich natürlicher ist als die Grammatik, die wir erhalten würden, wenn wir die Links-Rekursion auf konventionellem Wege eliminieren würden.

<i>program</i>	→	<i>statement</i> ⁺
<i>statement</i>	→	<i>variable</i> “=” <i>expr</i> “;”
		“ print ” “(” <i>expr</i> “)” “;”
		“ if ” “(” <i>boolExpr</i> “)” “{” <i>statement</i> ⁺ “}”
		“ while ” “(” <i>boolExpr</i> “)” “{” <i>statement</i> ⁺ “}”
<i>boolExpr</i>	→	<i>expr</i> (“==” <i>expr</i> “<” <i>expr</i>)
<i>expr</i>	→	<i>product</i> ((“+” <i>product</i>) (“-” <i>product</i>))*
<i>product</i>	→	<i>factor</i> ((“*” <i>factor</i>) (“/” <i>factor</i>))*
<i>factor</i>	→	“(” <i>expr</i> “)”
		VARIABLE
		NUMBER

Abbildung 13.12: EBNF-Grammatik für die Programmier-Sprache SL.

Die Spezifikation der Token ist in Abbildung 13.13 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.

```

70  VAR      : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
71  NUMBER   : '0'..'9' | ('1'..'9')('0'..'9')+;
72  WS       : (' '|'\t'|\n'|\r') { skip(); };
73  COMMENT  : '/*' ~( '*/' )+ '*/'    { skip(); };

```

Abbildung 13.13: ANTLR-Spezifikation der Token.

Aufgabe 20:

1. Erweitern Sie den oben diskutierten Interpreter um **for**-Schleifen.
2. Reichern Sie unsere Beispiel-Programmier-Sprache um die logischen Operatoren “&&” für das logische *Und*, “||” für das logische *Oder* und “!” für die Negation an. Dabei soll der Operator “!” am stärksten und der Operator “||” am schwächsten binden.
3. Erweitern Sie die Syntax der arithmetischen Ausdrücke so, dass auch vordefinierte mathematische Funktionen wie `exp()` oder `ln()` benutzt werden können.

Hinweis: Wenn Sie das Paket `java.lang.reflect` benutzen, kommen Sie mit einer zusätzlichen Klasse aus und können damit alle in `java.lang.Math` definierten Methoden implementieren.

4. Erweitern Sie den Interpreter so, dass auch benutzerdefinierte Funktionen möglich werden.

Hinweis: Jetzt reicht es nicht mehr, die Belegungen der Variablen in einer global definierten Hashtabelle zu verwalten. Stattdessen müssen Sie den Methoden `eval()` und `execute()` als zusätzliches Argument eine Liste von Hashtabelle mitgeben, in der die momentane Belegung der Variablen gespeichert wird. Dabei entspricht jedem Element der Liste ein (geschachtelter) Funktions-Aufruf.

Kapitel 14

Earley-Parser

In diesem Kapitel stellen wir ein effizientes Verfahren vor, mit dem es möglich ist, für eine beliebige vorgegebene kontextfreie Grammatik

$$G = \langle V, \Sigma, R, S \rangle \quad \text{und einen String } s \in \Sigma^*$$

zu entscheiden, ob s ein Element der Sprache $L(G)$ ist, ob also $s \in L(G)$ gilt. Der Algorithmus, denn wir gleich angeben werden, wurde 1970 von Jay Earley publiziert [Ear70]. Das Kapitel gliedert sich in die folgenden Abschnitte.

1. Zunächst skizzieren wir die dem Algorithmus zu Grunde liegende Theorie.
2. Danach geben wir eine einfache Implementierung des Algorithmus in *Java* an.
3. Anschließend beweisen wir die Korrektheit und Vollständigkeit des Algorithmus.
4. Zum Abschluß des Kapitels untersuchen wir die Komplexität.

14.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist.

Definition 38 (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

$$\langle A \rightarrow \alpha \bullet \beta, j \rangle$$

dann als ein *Earley-Objekt*, falls folgendes gilt:

1. $(A \rightarrow \alpha \beta) \in R$ und
2. $j \in \{0, 1, \dots, n\}$. □

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

$$\langle A \rightarrow \alpha \bullet \beta, j \rangle \in Q_k \quad \text{mit } k \geq j$$

ist dann wie folgt:

1. Der Parser hat bereits x_0, \dots, x_j verarbeitet.

2. Der Parser versucht, am Anfang des Strings x_{j+1}, \dots, x_n ein A mit Hilfe der Regel $A \rightarrow \alpha\beta$ zu parsen.
3. Das α wurde bereits erkannt, es gilt

$$\alpha \Rightarrow^* x_{j+1} \cdots x_k.$$

4. Folglich muss der Parser am Anfang des Teilstrings $x_{k+1} \cdots x_n$ nun β parsen.

Der Algorithmus von Earley verwaltet also für $i = 0, 1, \dots, n$ Mengen Q_i von Earley-Objekten. 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 $x_1 \cdots x_n$ erkennen. Die restlichen Mengen Q_i sind für $i = 1, \dots, n$ zunächst leer. Die Mengen Q_i werden nun durch die folgende drei Operationen so lange wie möglich erweitert:

1. *Lese-Operation*

Falls der Zustand Q_i 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 i bereits den Teil β erkannt. Folgt auf dieses β nun, wie in der Regel $A \rightarrow \beta a\gamma$ vorgesehen, an der Position $i + 1$ das Terminal a , so muss der Parser nach der Position $i + 1$ nur noch γ erkennen. Daher wird in diesem Fall das Earley-Objekt

$$\langle A \rightarrow \beta a \bullet \gamma, k \rangle$$

dem Zustand Q_{i+1} hinzugefügt:

$$\langle A \rightarrow \beta \bullet a\gamma, k \rangle \in Q_i \wedge a \in \Sigma \wedge x_{i+1} = a \Rightarrow Q_{i+1} := Q_{i+1} \cup \{\langle A \rightarrow \beta a \bullet \gamma, k \rangle\}.$$

2. *Vorhersage-Operation*

Falls der Zustand Q_i 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_i 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 $\langle C \rightarrow \bullet \gamma, i \rangle$ zu der Menge Q_i hinzu:

$$\langle A \rightarrow \beta \bullet C\delta, k \rangle \in Q_i \wedge C \in V \wedge (C \rightarrow \gamma) \in R \Rightarrow Q_i := Q_i \cup \{\langle C \rightarrow \bullet \gamma, i \rangle\}.$$

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 $s = x_1 \cdots x_n$ 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 14.1 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 14.1: 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 wir 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

$$\langle P \rightarrow F \bullet, 2 \rangle$$

in Q_3 ein. Hier können wir eine weiteres Mal die Vervollständigungs-Operation anwenden. Durch iterierte Anwendung der Vervollständigungs-Operation erhalten wir zusätzlich die folgenden Earley-Objekte:

- (a) $\langle P \rightarrow P \bullet "*" F, 2 \rangle$,
- (b) $\langle E \rightarrow E "+" P \bullet, 0 \rangle$,
- (c) $\langle E \rightarrow E \bullet "+" P, 0 \rangle$
- (d) $\langle E \rightarrow P \bullet, 0 \rangle$,
- (e) $\langle \hat{S} \rightarrow E \bullet, 0 \rangle$.

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 14.1 angegebenen Grammatik liegt.

14.2 Implementierung

Im folgenden präsentieren wir eine einfache Implementierung des von Earley angegebenen Algorithmus. Abbildung 14.2 auf Seite 189 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 $[Q_0, Q_1, \dots, Q_n]$.

```
1  public class EarleyParser {
2
3      private Grammar mGrammar;
4      private String  mString;
5      private int     mLength;
6
7      private List<Set<EarleyItem>> mStateList;
8
9      public EarleyParser(Grammar grammar, String string) {
10         mGrammar = grammar;
11         mString  = string;
12         mLength  = mString.length();
13         mStateList = new ArrayList<Set<EarleyItem>>(mLength + 1);
14         for (int i = 0; i <= mLength; ++i) {
15             Set<EarleyItem> qi = new TreeSet<EarleyItem>();
16             if (i == 0) {
17                 EarleyItem start = mGrammar.startItem();
18                 qi.add(start);
19             }
20             mStateList.add(qi);
21         }
22         parse();
23     }
24     ...
25 }
```

Abbildung 14.2: Struktur der Klasse `EarleyParser`

Abbildung 14.2 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 14.3 auf Seite 190 gezeigt ist. Die äußere `for`-Schleife ite-

riert 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 vergrößert.¹ Anschließend wird von der Methode `scan()` die Lese-Operation durchgeführt, welche die Menge Q_{i+1} initialisiert.

```

1  void parse() {
2      for (int i = 0; i <= mLength; ++i) {
3          boolean change = false;
4          do {
5              change = complete(i);
6              change = change || predict(i);
7          } while (change);
8          scan(i);
9      }
10 }

```

Abbildung 14.3: 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 14.4: Implementierung der Vervollständigungs-Operation

Abbildung 14.4 auf Seite 190 zeigt die Implementierung der Vervollständigungs-Operation. Das Argument i gibt an, welche der Mengen Q_i vervollständigt werden soll. Die Methode gibt als

¹Wenn 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.

Ergebnis einen Wahrheitswert zurück. Dieser ist **true** falls bei der Vervollständigungs-Operation neue Earley-Objekte in die Menge Q_i eingefügt werden. Bleibt die Menge Q_i bei der Operation unverändert, so wird **false** zurück gegeben. Die Vervollständigungs-Operation war im letzten Abschnitt wie folgt definiert worden:

$$\langle C \rightarrow \gamma \bullet, j \rangle \in Q_i \wedge \langle A \rightarrow \beta \bullet C \delta, k \rangle \in Q_j \Rightarrow Q_i := Q_i \cup \{ \langle A \rightarrow \beta C \bullet \delta, k \rangle \}$$

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 Q_i :

- (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 14.5: Implementierung der Vorhersage-Operation

Abbildung 14.5 auf Seite 192 zeigt die Methode `predict()`, welche die Vorhersage-Operation implementiert. Diese Operation hatten wir im letzten Abschnitt durch die folgende Formel spezifiziert:

$$\langle A \rightarrow \beta \bullet C\delta, k \rangle \in Q_i \wedge C \in V \wedge (C \rightarrow \gamma) \in R \Rightarrow Q_i := Q_i \cup \{ \langle C \rightarrow \bullet \gamma, i \rangle \}$$

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 14.6: Implementierung der Lese-Operation

Abbildung 14.6 zeigt die Implementierung der Lese-Operation. Wir hatten diese Operation wie folgt spezifiziert:

$$\langle A \rightarrow \beta \bullet a\gamma, k \rangle \in Q_i \wedge a \in \Sigma \wedge x_{i+1} = a \Rightarrow Q_{i+1} := Q_{i+1} \cup \{ \langle A \rightarrow \beta a \bullet \gamma, k \rangle \}.$$

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 Numerierung 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 14.7: Die Klasse `EarleyItem`, erster Teil.

Die Abbildungen 14.7 und 14.8 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.

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.

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 14.9 zeigt die von unserem Earley-Parser berechneten Mengen für die Eingabe `1+2*3`, falls die in Abbildung 14.1 gezeigte Grammatik verwendet wird.

```

31     public Boolean isComplete() {
32         return getBeta().getItemList().size() == 0;
33     }
34     public Boolean sameVar(Variable C) {
35         if (mBeta.getItemList().size() > 0) {
36             Item first = mBeta.getItemList().get(0);
37             if (first instanceof Variable) {
38                 Variable v = (Variable) first;
39                 return v.equals(C);
40             }
41         }
42         return false;
43     }
44     public Boolean scan(char a) {
45         if (mBeta.getItemList().size() > 0) {
46             Item first = mBeta.getItemList().get(0);
47             if (first instanceof Literal) {
48                 Literal v = (Literal) first;
49                 return a == v.getChar();
50             }
51         }
52         return false;
53     }
54     public Variable nextVar() {
55         if (mBeta.getItemList().size() > 0) {
56             Item first = mBeta.getItemList().get(0);
57             if (first instanceof Variable) {
58                 return (Variable) first;
59             }
60         }
61         return null;
62     }
63     public EarleyItem moveDot() {
64         List<Item> alphaList = new ArrayList<Item>();
65         List<Item> betaList = new ArrayList<Item>();
66         alphaList.addAll(mAlpha.getItemList());
67         betaList.addAll(mBeta.getItemList());
68         Item next = betaList.get(0);
69         alphaList.add(next);
70         betaList.remove(0);
71         Body alpha = new Body(alphaList);
72         Body beta = new Body(betaList);
73         return new EarleyItem(mVariable, alpha, beta, mIndex);
74     }
75 }

```

Abbildung 14.8: Die Klasse EarleyItem, zweiter Teil.

Q0:
<\$Start -> (*) expr, 0>
<expr -> (*) product, 0>
<expr -> (*) expr '+' product, 0>
<factor -> (*) '1', 0>
<factor -> (*) '2', 0>
<factor -> (*) '3', 0>
<product -> (*) factor, 0>
<product -> (*) product '*' factor, 0>

Q1:
<\$Start -> expr(*), 0>
<expr -> expr(*) '+' product, 0>
<expr -> product(*), 0>
<factor -> '1'(*), 0>
<product -> factor(*), 0>
<product -> product(*) '*' factor, 0>

Q2:
<expr -> expr '+'(*) product, 0>
<factor -> (*) '1', 2>
<factor -> (*) '2', 2>
<factor -> (*) '3', 2>
<product -> (*) factor, 2>
<product -> (*) product '*' factor, 2>

Q3:
<\$Start -> expr(*), 0>
<expr -> expr(*) '+' product, 0>
<expr -> expr '+' product(*), 0>
<factor -> '2'(*), 2>
<product -> factor(*), 2>
<product -> product(*) '*' factor, 2>

Q4:
<factor -> (*) '1', 4>
<factor -> (*) '2', 4>
<factor -> (*) '3', 4>
<product -> product '*'(*) factor, 2>

Q5:
<\$Start -> expr(*), 0>
<expr -> expr(*) '+' product, 0>
<expr -> expr '+' product(*), 0>
<factor -> '3'(*), 4>
<product -> product(*) '*' factor, 2>
<product -> product '*' factor(*), 2>

Abbildung 14.9: Ausgabe des Earley-Parsers für die Eingabe “1+2*3” und die Grammatik aus Abbildung 14.1.

14.3 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 $s = x_1x_2 \cdots x_n$ 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 39 Es sei $s = x_1x_2 \cdots x_n \in T^*$ 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

$$\alpha \Rightarrow^* x_{k+1} \cdots x_i.$$

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

$$\langle \hat{S} \rightarrow \bullet S, 0 \rangle$$

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) $A = \hat{S}$,
- (b) $\alpha = \varepsilon$,
- (c) $\beta = S$ und
- (d) $k = 0$.

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 $\alpha = \varepsilon$ 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. \square

Korollar 40 (Korrektheit)

Wenden wir den Algorithmus von Earley auf den String $s = x_1 \cdots x_n$ an und gilt $\langle \widehat{S} \rightarrow S \bullet, 0 \rangle \in Q_n$, so folgt $s \in L(G)$.

Beweis: Setzen wir $A := \widehat{S}$, $\alpha := S$, $\beta := \varepsilon$, $k := 0$ und $i := n$, so folgt das Ergebnis unmittelbar, wenn wir das letzte Lemma auf die Voraussetzung anwenden. \square

Das Korollar zeigt: Produziert der Algorithmus von Earley das Earley-Objekt

$$\langle \widehat{S} \rightarrow S \bullet, 0 \rangle \in Q_n,$$

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 41 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

$$\delta \Rightarrow^* x_{j+1} \cdots x_h \tag{14.1}$$

und

$$\mu \Rightarrow^* x_{h+1} \cdots x_l \tag{14.2}$$

für ein geeignetes $h \in \{j+1, \dots, l+1\}$. 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

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j \bullet D\mu\gamma, k \rangle \in Q_j. \tag{14.3}$$

Auf dieses Earley-Objekt wenden wir die Vorhersage-Operation an und erhalten

$$\langle D \rightarrow \bullet \delta, j \rangle \in Q_j. \tag{14.4}$$

Aus (14.4) und (14.1) folgt mit der Induktions-Voraussetzung

$$\langle D \rightarrow \delta \bullet, j \rangle \in Q_h. \tag{14.5}$$

Aus (14.3) und (14.5) folgt mit der Vervollständigungs-Operation

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j D \bullet \mu\gamma, k \rangle \in Q_h. \tag{14.6}$$

Aus (14.6) und (14.2) folgt mit der Induktions-Voraussetzung

$$\langle A \rightarrow \alpha x_{i+1} \cdots x_j D\mu \bullet \gamma, k \rangle \in Q_l. \tag{14.7}$$

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 42 (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

14.4 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 14.5) 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 14.6 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 14.4 auf Seite 190 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, 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 15

LALR-Parser

In diesem Kapitel diskutieren wir den theoretischen Hintergrund des Parser-Generators *Bison*. Dazu stellen wir im nächsten Abschnitt das Konzept der *Bottom-Up-Parser* vor. Im folgenden Abschnitt zeigen wir, wie Bottom-Up-Parser implementiert werden können. Wir stellen *Shift-Reduce-Parser* vor, die eigentlich nichts anderes sind als deterministische Keller-Automaten. Ein Shift-Reduce-Parser arbeitet mit Hilfe einer Tabelle, in der hinterlegt ist, wie der Parser in einem bestimmten Zustand die Eingaben verarbeiten muss. Die Theorie, wie eine solche Tabelle sinnvoll mit Informationen gefüllt werden kann, entwickeln wir dann in dem folgenden Abschnitt: Zunächst diskutieren wir die *SLR-Parser* (*simple LR-Parser*). Dies ist die einfachste Klasse von Shift-Reduce-Parsern. Das Konzept der SLR-Parser ist leider für die Praxis nicht mächtig genug. Daher verfeinern wir dieses Konzept und erhalten so die Klasse der *kanonischen LR-Parser*. Da die Tabellen für LR-Parser in der Praxis häufig zu groß werden, vereinfacht man diese Tabellen etwas und erhält dann das Konzept der *LALR-Parser*, das von der Mächtigkeit zwischen dem Konzept der *SLR-Parser* und dem Konzept der *LR-Parser* liegt.

15.1 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 *rightmost derivation* steht und ausdrückt, dass solche Parser eine Rechts-Ableitung konstruieren.

Bei der Implementierung eines LR-Parsers stellen sich zwei Fragen:

1. Welche Teilstrings ersetzen wir?
2. Welche Regeln verwenden wir dabei?

Die Beantwortung dieser Fragen ist im Allgemeinen nicht trivial. Zwar gehen wir die Strings immer von links nach rechts durch, aber damit ist noch nicht unbedingt 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.

15.2 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.
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:

$$states \in Stack\langle Q \rangle.$$

2. Einem Stack *symbols*, auf dem Grammatik-Symbole, also Terminale und Variablen-Symbole abgelegt werden:

$$symbols \in Stack\langle T \cup V \rangle.$$

Zur Vereinfachung der folgenden Überlegungen nehmen wir an, dass die Menge T der Terminale das spezielle Symbol “\$” enthält und dass dieses Symbol das Ende des zu parsenden Strings spezifiziert aber sonst in dem String nicht auftritt. Im Scanner wird dieses Symbol später den Namen EOF für “*end of file*” haben. Weiter nehmen wir an, dass die folgenden beiden Methoden zu Verfügung stehen, um Eingabe-Symbole zu lesen.

1. *peekNextToken* : $TokenStream \rightarrow T$

Die Funktion gibt das nächste zu lesende Token aus der Eingabe zurück. Dieses Token wird aber nicht aus der Eingabe entfernt.

2. *removeNextToken* : $TokenStream \rightarrow void$

Diese Funktion entfernt das nächste Token aus der Eingabe, gibt aber kein Ergebnis zurück.

Falls die Funktion *peekNextToken()* mehrmals aufgerufen wird, ohne dass zwischendurch die Methode *removeNextToken()* aufgerufen wird, dann wird immer das selbe Token zurück gegeben. Damit können wir nun das Verhalten eines Shift-Reduce-Parsers beschreiben:

1. Der Symbol-Stack *symbols* ist am Anfang leer:

$symbols := []$.

2. Zu Beginn enthält der Zustands-Stack *states* den Start-Zustand q_0 :

$states := [q_0]$.

3. Anschließend werden die folgenden Aktionen solange durchlaufen, wie Eingabe-Symbole zu lesen sind.

- (a) Zunächst speichern wir das Symbol, was im Stack *states* oben liegt, in der Variablen q :

$q := states.top()$.

- (b) Weiter sei t das nächste, noch ungelesene Token der Eingabe:

$t := peekNextToken()$.

- (c) Die folgenden Aktionen hängen von dem Wert der Funktion $action(q, t)$ ab:

- i. $action(q, t) = \langle \text{shift}, s \rangle$.

In diesem Fall wird das Token t auf den Symbol-Stack abgelegt

$symbols.push(t)$

und auf den Zustands-Stack legen wir den neuen Zustand s :

$states.push(s)$.

Anschließend wird das Token t durch den Aufruf von

$removeNextToken()$

aus der Eingabe entfernt.

- ii. $action(q, t) = \langle \text{reduce}, r \rangle$.

Dann ist r eine Grammatik-Regel der Form

$r = (A \rightarrow X_1 \cdots X_n)$

In diesem Fall liegen die Symbole X_1, \dots, X_n auf dem Symbol-Stack. Da wir den Symbol-Stack mit der Grammatik-Regel r reduzieren wollen, werden diese Symbole nun vom Symbol-Stack entfernt und anschließend wird statt dessen das Symbol A auf dem Symbol-Stack abgelegt:

$symbols.pop(n); symbols.push(A)$.

Das Token t verbleibt in diesem Fall in der Eingabe, die Funktion $removeNextToken()$ wird also nicht aufgerufen.

Genauso wie wir n Elemente vom Symbol-Stack entfernen, entfernen wir auch n Elemente vom Zustands-Stack.

$states.pop(n);$

Anschließend bestimmen wir den Zustand s , der nach Entfernung von n Zuständen auf dem Zustands-Stack liegt

$s := states.top()$

und legen danach den Zustand $goto(s, A)$ auf dem Zustands-Stack ab:

$states.push(goto(s, A))$.

- iii. $action(q, t) = \text{accept}$.

In diesem Fall wird das Parsen mit Erfolg beendet.

- iv. $action(q, t) = \text{error}$.

Falls dieser Fall eintritt, wurde ein Syntax-Fehler gefunden.

```

1  symbols := [];
2  states  := [ q0 ];
3  while (not end-of-file) {
4      q := states.top();
5      t := peekNextToken();
6      a := action(q,t);
7      switch {
8      when a = <shift,s>:
9          symbols.push(t);
10         states .push(s);
11         removeNextToken();
12     when a = <reduce,A → X1...Xn>:
13         symbols.pop(n);
14         symbols.push(A);
15         states .pop(n);
16         s := states.top();
17         states.push(goto(s, A));
18     when a = accept:
19         print("Parzen erfolgreich");
20         return;
21     when a = error:
22         error;
23     }
24 }

```

Abbildung 15.1: Algorithmus eines Shift-Reduce-Parsers als Pseudo-Code

Abbildung 15.1 zeigt die Implementierung dieses Algorithmus als Pseudo-Code. Um den Pseudo-Code zu verstehen, nehmen wir an, der zu parsende String habe die Form

$$t_1 \cdots t_n t_{n+1},$$

wobei $t_{n+1} = \text{“\$”}$ gilt. Hier spezifiziert “\$” das Ende des zu parsenden Strings, der eigentlich zu erkennende String besteht also nur aus den Terminalen $t_1 \cdots t_n$. Wir betrachten nun die Situation, in der die Terminalen $t_1 \cdots t_j$ bereits gelesen wurden, während die Terminalen $t_{j+1} \cdots t_n$ sich noch im Eingabe-Strom befinden. Die Methode `peekNextToken()` würde also beim nächsten Aufruf das Terminal t_{j+1} zurück liefern. Zu diesem Zeitpunkt befinden sich dann die Symbole X_1, \dots, X_k auf dem Symbol-Stack, es gilt also

$$symbols = [X_1, \dots, X_k],$$

wobei X_k das zuoberst auf dem Stack liegende Symbol bezeichnet. Dann kann nach Konstruktion des oben gezeigten Algorithmus aus dem Symbol-String $X_1 \cdots X_k$ der Terminal-String $t_1 \cdots t_j$ abgeleitet werden:

$$X_1 \cdots X_k \Rightarrow^* t_1 \cdots t_j. \quad (*)$$

Die Behauptung (*) kann leicht durch eine Induktion über die Anzahl der Durchläufe der **while**-Schleife nachgewiesen werden. Gleichzeitig hat der Stack *states* dann die Form

$$states = [q_0, q_1, \dots, q_k].$$

Es wird unser Ziel sein die Zustände q_i so zu konstruieren, dass der Zustand q_k alle Informationen über den Symbol-Stack $[X_1, \dots, X_k]$ speichert, die benötigt werden, um die Entscheidung, ob ein gegebenes Terminal auf den Symbol-Stack geschoben wird (*shift*), oder ob statt dessen der Symbol-Stack durch Anwendung einer Regel reduziert wird (*reduce*).

Wir bemerken noch, dass ein Shift-Reduce-Parser, der nach dem obigen Schema aufgebaut ist, den Vorteil hat, dass der zu parsende String nicht vollständig abgespeichert werden muss, denn die Entscheidungen, die beim Parsen getroffen werden müssen, basieren immer nur auf dem nächsten zu lesenden Zeichen. Diese Eigenschaft bedingt, dass Shift-Reduce-Parser mit vergleichsweise wenig Speicherplatz auskommen können. Dieser Umstand hat in den Anfangsjahren der Entwicklung von Übersetzern wesentlich zur Verbreitung von Shift-Reduce-Parser beigetragen.

15.2.1 Implementierung eines *Shift-Reduce-Parsers*

Eine Implementierung des in Abbildung 15.1 gezeigten Algorithmus ist in den Abbildungen 15.2 und 15.3 auf den Seiten 209 und 210 gezeigt. Wir diskutieren zunächst die in Abbildung 15.2 gezeigte Struktur der Klasse *ShiftReduceParser*. Aus didaktischen Gründen habe ich darauf verzichtet, die Implementierung möglichst effizient zu gestalten, denn eine solche Implementierung wäre deutlich komplexer.

1. Die Klasse enthält die folgenden Member-Variablen, die alle am Anfang der Klasse deklariert sind.
 - (a) `mTokenList` enthält die Liste aller zu parsenden Tokens. Diese Liste wird von einem Scanner zur Verfügung gestellt.
 - (b) `mIndex` gibt den Index des nächsten aus der Liste `mTokenList` zu lesenden Tokens an.
 - (c) `mSymbols` ist der Symbol-Stack.
 - (d) `mStates` ist der Zustands-Stack.
 - (e) `mParseTable` kodiert die beiden Funktionen *action()* und *goto()* in Form von Tabellen.
2. Die Methode *main()* hat die Aufgabe, den Scanner aufzurufen und von diesem eine Liste aller Token erzeugen zu lassen.
 - (a) Dazu wird zunächst ein Scanner erzeugt, der von der Standard-Eingabe liest.
 - (b) Die Liste aller Token wird in der in Zeile 11 erzeugten `ArrayList<Symbol>` gespeichert.
 - (c) Das eigentliche Lesen der Token geschieht dann in der *do-while*-Schleife, die alle vorhandenen Token liest. Anschließend wird noch ein spezielles *End-Of-File-Token* am Ende der Liste eingefügt.

Der Scanner, den wir hier verwenden, wurde von dem Werkzeug *JFlex* erzeugt.

Die Liste aller gelesenen Token wird in Zeile 21 dem Konstruktor der Klasse *ShiftReduceParser* übergeben.

3. Der Konstruktor der Klasse *ShiftReduceParser* hat die Aufgabe, alle Member-Variablen zu initialisieren.
 - (a) `mTokenList` wird mit der als Argument übergebenen Token-Liste initialisiert.
 - (b) `mIndex` wird am Anfang auf 0 gesetzt. Damit zeigt dieser Index dann auf das erste zu lesende Token.
 - (c) Anschließend werden Symbol-Stack und Zustands-Stack angelegt.
 - (d) Zum Schluss füllt der Konstruktor *ParseTable()* die Tabellen, in denen die Funktionen *action()* und *goto()* abgespeichert sind.
4. Die Methode *parse()* ist aus Platzgünden aus Abbildung 15.2 ausgeklammert worden und wird später diskutiert.
5. Die Methode *peekNextToken()* liefert das Token zurück, auf das der Zeiger `mIndex` zeigt. Zu beachten ist, dass der Zeiger `mIndex` dabei nicht erhöht wird.

```

1  public class ShiftReduceParser {
2      private List<Symbol> mTokenList;
3      private int mIndex; // index of next token
4      private Stack<Symbol> mSymbols;
5      private Stack<State> mStates;0
6      private ParseTable mParseTable;
7      private Map<Pair<State, Symbol>, Action> mActionTable;
8      private Map<Pair<State, Symbol>, State> mGotoTable;
9
10     public static void main(String args[]) throws Error, IOException {
11         Scanner scanner = new Scanner(System.in);
12         ArrayList<Symbol> tokenList = new ArrayList<Symbol>();
13         Symbol symbol;
14         do {
15             symbol = scanner.yylex();
16             if (symbol != null) {
17                 tokenList.add(symbol);
18             } else {
19                 tokenList.add(new Symbol(Symbol.EOF, null));
20             }
21         } while (symbol != null); // end of file
22         ShiftReduceParser parser = new ShiftReduceParser(tokenList);
23         if (parser.parse()) {
24             System.out.println("Parsing successful!");
25         } else {
26             System.out.println("Syntax error!");
27         }
28     }
29     public ShiftReduceParser(List<Symbol> input) {
30         mTokenList = input;
31         mIndex = 0;
32         mSymbols = new Stack<Symbol>();
33         mStates = new Stack<State>();
34         mParseTable = new ParseTable();
35     }
36     public boolean parse() { ... }
37     private Symbol peekNextToken() { return mTokenList.get(mIndex); }
38     private void removeNextToken() { ++mIndex; }
39     private State startState() { return new State(0); }
40     private Action action(State s, Symbol t) {
41         Pair<State, Symbol> p = new Pair<State, Symbol>(s, t);
42         return mParseTable.mActionTable.get(p);
43     }
44     private State gotoState(State state, Symbol symbol) {
45         Pair<State, Symbol> p = new Pair<State, Symbol>(state, symbol);
46         return mParseTable.mGotoTable.get(p);
47     }
48 }

```

Abbildung 15.2: Ein Shift-Reduce-Parser für arithmetische Ausdrücke.

6. Die Methode *removeNextToken()* inkrementiert den Zeiger *mIndex*, damit anschließend das nächste Token gelesen werden kann.
7. Die Methode *action(s, t)* berechnet für einen gegebenen Zustand *s* und ein Token *t* den Funktionswert dadurch, dass der Wert in der Tabelle *mActionTable* nachgeschlagen wird. Dazu müssen die beiden Argumente erst zu einem Paar zusammengefasst werden.
8. Die Methode *gotoState(state, symbol)* berechnet für einen gegebenen Zustand *state* und eine syntaktische Variable *symbol* den Funktionswert durch Nachschlagen in der Tabelle *mGotoTable*.

```

1  public boolean parse() {
2      State start = startState();
3      mStates.push(start);
4      while (true) {
5          State q = mStates.peek();
6          Symbol t = peekNextToken();
7          Action p = action(q,t);
8          if (p == null) {
9              return false;
10         }
11         if (p instanceof Shift) {
12             mSymbols.push(t);
13             mStates .push(((Shift) p).getState());
14             removeNextToken();
15         } else if (p instanceof Reduce) {
16             Rule      rule = ((Reduce) p).getRule();
17             Symbol    head = rule.getHead();
18             List<Symbol> body = rule.getBody();
19             for (Symbol s: body) {
20                 mSymbols.pop();
21                 mStates .pop();
22             }
23             mSymbols.push(head);
24             State s = mStates.peek();
25             mStates.push( gotoState(s, head) );
26         } else if (p instanceof Accept) {
27             return true;
28         }
29     }
30 }

```

Abbildung 15.3: Die Methode *parse()* des Shift-Reduce-Parsers.

Als nächstes diskutieren wir die in Abbildung 15.3 gezeigte Implementierung der Methode *parse()*. Diese Methode setzt den in Abbildung 15.1 gezeigten Pseudo-Code um.

1. Zunächst wird der Start-Zustand auf den Zustands-Stack gelegt.
2. Es folgt eine Endlos-Schleife, die sich von Zeile 4 bis zum Ende der Methode erstreckt und die wir jetzt im Detail diskutieren.
 - (a) *q* ist der Zustand, der oben auf dem Zustands-Stack liegt.

- (b) t ist das nächste zu lesende Token.
- (c) p ist der Funktionswert $action(q, t)$. Die Funktion wird $action()$ in einer Tabelle abgespeichert. Um diese Tabelle nicht zu groß werden zu lassen, werden dort die Einträge, die den Wert **error** haben weggelassen. Für diese Einträge gibt die Funktion dann den Wert **null** zurück.
Generell werden die Werte der Methode $action()$ als Elemente der abstrakten Klasse *Action* dargestellt. Von dieser Klasse werden drei Klassen abgeleitet:
 - i. Die Klasse *Shift* stellt Paare der Form $\langle \text{shift}, s \rangle$ dar. Diese Klasse enthält eine Member-Variable **mState**, in welcher der Zustand s gespeichert ist.
 - ii. Die Klasse *Reduce* stellt Paare der Form $\langle \text{reduce}, r \rangle$ dar. Diese Klasse enthält eine Member-Variable **mRule**, in welcher die Regel r gespeichert ist.
 - iii. Die Klasse *Accept* stellt das Ergebnis **accept** dar.
 - iv. Das Ergebnis **error** wird durch den Wert **null** signalisiert.
- (d) Die Methode $parse()$ gibt in Zeile 9 den Wert **false** zurück, wenn $action(q, t)$ den Wert **null** hat.
- (e) Falls die Funktion $action(q, t)$ als Ergebnis den Wert $\langle \text{shift}, s \rangle$ hat, so wird das Token t auf den Symbol-Stack gelegt und aus der Token-Liste entfernt. Der Zustand s wird auf den Zustands-Stack gelegt.
- (f) Falls die Funktion $action(q, t)$ als Ergebnis den Wert $\langle \text{reduce}, r \rangle$ hat und die rechte Seite der Regel r aus n Symbolen besteht, so werden einerseits n Symbole vom Symbol-Stack genommen und andererseits werden n Zustände vom Zustands-Stack entfernt. Anschließend wird die linke Seite der Regel auf dem Symbol-Stack abgelegt. Dann wird ein neuer Zustand mit Hilfe der Funktion $goto()$ ausgerechnet und auf dem Symbol-Stack abgelegt.
- (g) Falls die Funktion $action(q, t)$ als Ergebnis den Wert **accept** hat, wird das Parsen mit Erfolg abgeschlossen.

Die bisher gezeigten Methoden waren von der zu parsenden Grammatik unabhängig. Die Grammatik hat nur Einfluss auf den Konstruktor $ParseTable()$, denn dort werden die Tabellen gefüllt, mit deren Hilfe die Funktionen $action()$ und $goto()$ realisiert werden. Die Tabellen 15.1 und 15.2 zeigen diese Tabellen für die in Abbildung 15.4 angegebene Grammatik. Bei dieser Grammatik ergeben sich insgesamt 16 verschiedene Zustände, die wir auf die Namen s_0, s_1, \dots, s_{15} taufen. Wir haben in der Tabelle 15.1 zwei verschiedene Abkürzungen verwendet:

1. $\langle \text{shift}, s_i \rangle$ steht für $\langle \text{shift}, s_i \rangle$.
2. $\langle \text{rdc}, r_i \rangle$ steht für $\langle \text{reduce}, r_i \rangle$, wobei r_i für die i -te Regel bezeichnet. Dabei haben wir die in Abbildung 15.4 gezeigten Regeln wie folgt durchnummeriert:
 - (a) $r_1 = (\text{Expr} \rightarrow \text{Expr} \text{ "+" } \text{Product})$
 - (b) $r_2 = (\text{Expr} \rightarrow \text{Expr} \text{ "-" } \text{Product})$
 - (c) $r_3 = (\text{Expr} \rightarrow \text{Product})$
 - (d) $r_4 = (\text{Product} \rightarrow \text{Product} \text{ "*" } \text{Factor})$
 - (e) $r_5 = (\text{Product} \rightarrow \text{Product} \text{ "/" } \text{Factor})$
 - (f) $r_6 = (\text{Product} \rightarrow \text{Factor})$
 - (g) $r_7 = (\text{Factor} \rightarrow \text{"(" Expr ")"})$
 - (h) $r_8 = (\text{Factor} \rightarrow \text{NUMBER})$

Die Einträge, die in Tabelle 15.1 frei geblieben sind, haben den Wert **error**. Entsprechendes gilt auch für die freien Einträge in der Tabelle 15.2.

Die Abbildungen 15.5 bis 15.8 zeigen nun die Implementierung der Klasse $parseTable()$, die sich unmittelbar aus den oben gezeigten Tabellen ergibt und daher hier nicht weiter diskutiert wird. Wie diese Tabellen berechnet werden ist Gegenstand der folgenden Abschnitte.

$Expr$	\rightarrow	$Expr \text{ “+” } Product$
	$ $	$Expr \text{ “-” } Product$
	$ $	$Product$
$Product$	\rightarrow	$Product \text{ “*” } Factor$
	$ $	$Product \text{ “/” } Factor$
	$ $	$Factor$
$Factor$	\rightarrow	$\text{ “(” } Expr \text{ “)” }$
	$ $	$NUMBER$

Abbildung 15.4: Eine Grammatik für arithmetische Ausdrücke.

Zustand	EOF	+	-	*	/	()	NUMBER
s_0						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_1	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$		$\langle rdc, r_6 \rangle$	
s_2	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	
s_3	$\langle rdc, r_3 \rangle$	$\langle rdc, r_3 \rangle$	$\langle rdc, r_3 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_3 \rangle$	
s_4	accept	$\langle shft, s_8 \rangle$	$\langle shft, s_9 \rangle$					
s_5						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_6		$\langle shft, s_8 \rangle$	$\langle shft, s_9 \rangle$				$\langle shft, s_7 \rangle$	
s_7	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$		$\langle rdc, r_7 \rangle$	
s_8						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_9						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{10}	$\langle rdc, r_2 \rangle$	$\langle rdc, r_2 \rangle$	$\langle rdc, r_2 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_2 \rangle$	
s_{11}						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{12}						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{13}	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$		$\langle rdc, r_4 \rangle$	
s_{14}	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$		$\langle rdc, r_5 \rangle$	
s_{15}	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_1 \rangle$	

Tabelle 15.1: Tabelle der Funktion $action()$.

<i>Zstd</i>	<i>Expr</i>	<i>Product</i>	<i>Factor</i>
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 15.2: Tabelle der Funktion *goto()*.

```

1  public class ParseTable {
2      Map<Pair<State, Symbol>, Action> mActionTable;
3      Map<Pair<State, Symbol>, State> mGotoTable;
4      public ParseTable() {
5          mActionTable = new TreeMap<Pair<State, Symbol>, Action>();
6          State s0 = new State(0);
7          State s1 = new State(1);
8          State s2 = new State(2);
9          State s3 = new State(3);
10         State s4 = new State(4);
11         State s5 = new State(5);
12         State s6 = new State(6);
13         State s7 = new State(7);
14         State s8 = new State(8);
15         State s9 = new State(9);
16         State s10 = new State(10);
17         State s11 = new State(11);
18         State s12 = new State(12);
19         State s13 = new State(13);
20         State s14 = new State(14);
21         State s15 = new State(15);
22         Symbol t0 = new Symbol( 0, "EOF");
23         Symbol t1 = new Symbol( 1, "(");
24         Symbol t2 = new Symbol( 2, ")");
25         Symbol t3 = new Symbol( 3, "*");
26         Symbol t4 = new Symbol( 4, "+");
27         Symbol t5 = new Symbol( 5, "-");
28         Symbol t6 = new Symbol( 6, "/" );
29         Symbol t7 = new Symbol( 7, "NUMBER");
30         Symbol expr = new Symbol(Symbol.EXPR, "expr" );
31         Symbol factor = new Symbol(Symbol.FACTOR, "factor" );
32         Symbol product = new Symbol(Symbol.PRODUCT, "product" );
33         // expr -> expr '+' product
34         List<Symbol> body2 = new ArrayList<Symbol>();
35         Symbol head2 = expr;
36         body2.add(expr);
37         body2.add(t4);
38         body2.add(product);
39         Rule rule2 = new Rule(head2, body2);
40         // expr -> expr '-' product
41         List<Symbol> body3 = new ArrayList<Symbol>();
42         Symbol head3 = expr;
43         body3.add(expr);
44         body3.add(t5);
45         body3.add(product);
46         Rule rule3 = new Rule(head3, body3);

```

Abbildung 15.5: Die Klasse ParseTable, 1. Teil

```

47      // expr -> product
48      List<Symbol> body4 = new ArrayList<Symbol>();
49      Symbol head4 = expr;
50      body4.add(product);
51      Rule rule4 = new Rule(head4, body4);
52      // factor -> '(' expr ')',
53      List<Symbol> body5 = new ArrayList<Symbol>();
54      Symbol head5 = factor;
55      body5.add(t1);
56      body5.add(expr);
57      body5.add(t2);
58      Rule rule5 = new Rule(head5, body5);
59      // factor -> NUMBER
60      List<Symbol> body6 = new ArrayList<Symbol>();
61      Symbol head6 = factor;
62      body6.add(t7);
63      Rule rule6 = new Rule(head6, body6);
64      // product -> factor
65      List<Symbol> body7 = new ArrayList<Symbol>();
66      Symbol head7 = product;
67      body7.add(factor);
68      Rule rule7 = new Rule(head7, body7);
69      // product -> product '*' factor
70      List<Symbol> body8 = new ArrayList<Symbol>();
71      Symbol head8 = product;
72      body8.add(product);
73      body8.add(t3);
74      body8.add(factor);
75      Rule rule8 = new Rule(head8, body8);
76      // product -> product '/' factor
77      List<Symbol> body9 = new ArrayList<Symbol>();
78      Symbol head9 = product;
79      body9.add(product);
80      body9.add(t6);
81      body9.add(factor);
82      Rule rule9 = new Rule(head9, body9);

```

Abbildung 15.6: Die Klasse ParseTable, 2. Teil.

```

83      mActionTable.put(new Pair<State, Symbol>(s0, t1), new Shift(s2));
84      mActionTable.put(new Pair<State, Symbol>(s0, t7), new Shift(s12));
85      mActionTable.put(new Pair<State, Symbol>(s1, t0), new Accept());
86      mActionTable.put(new Pair<State, Symbol>(s1, t4), new Shift(s4));
87      mActionTable.put(new Pair<State, Symbol>(s1, t5), new Shift(s6));
88      mActionTable.put(new Pair<State, Symbol>(s2, t1), new Shift(s2));
89      mActionTable.put(new Pair<State, Symbol>(s2, t7), new Shift(s12));
90      mActionTable.put(new Pair<State, Symbol>(s3, t2), new Shift(s11));
91      mActionTable.put(new Pair<State, Symbol>(s3, t4), new Shift(s4));
92      mActionTable.put(new Pair<State, Symbol>(s3, t5), new Shift(s6));
93      mActionTable.put(new Pair<State, Symbol>(s4, t1), new Shift(s2));
94      mActionTable.put(new Pair<State, Symbol>(s4, t7), new Shift(s12));
95      mActionTable.put(new Pair<State, Symbol>(s5, t0), new Reduce(rule2));
96      mActionTable.put(new Pair<State, Symbol>(s5, t2), new Reduce(rule2));
97      mActionTable.put(new Pair<State, Symbol>(s5, t3), new Shift(s9));
98      mActionTable.put(new Pair<State, Symbol>(s5, t4), new Reduce(rule2));
99      mActionTable.put(new Pair<State, Symbol>(s5, t5), new Reduce(rule2));
100     mActionTable.put(new Pair<State, Symbol>(s5, t6), new Shift(s10));
101     mActionTable.put(new Pair<State, Symbol>(s6, t1), new Shift(s2));
102     mActionTable.put(new Pair<State, Symbol>(s6, t7), new Shift(s12));
103     mActionTable.put(new Pair<State, Symbol>(s7, t0), new Reduce(rule3));
104     mActionTable.put(new Pair<State, Symbol>(s7, t2), new Reduce(rule3));
105     mActionTable.put(new Pair<State, Symbol>(s7, t3), new Shift(s9));
106     mActionTable.put(new Pair<State, Symbol>(s7, t4), new Reduce(rule3));
107     mActionTable.put(new Pair<State, Symbol>(s7, t5), new Reduce(rule3));
108     mActionTable.put(new Pair<State, Symbol>(s7, t6), new Shift(s10));
109     mActionTable.put(new Pair<State, Symbol>(s8, t0), new Reduce(rule4));
110     mActionTable.put(new Pair<State, Symbol>(s8, t2), new Reduce(rule4));
111     mActionTable.put(new Pair<State, Symbol>(s8, t3), new Shift(s9));
112     mActionTable.put(new Pair<State, Symbol>(s8, t4), new Reduce(rule4));
113     mActionTable.put(new Pair<State, Symbol>(s8, t5), new Reduce(rule4));
114     mActionTable.put(new Pair<State, Symbol>(s8, t6), new Shift(s10));
115     mActionTable.put(new Pair<State, Symbol>(s9, t1), new Shift(s2));
116     mActionTable.put(new Pair<State, Symbol>(s9, t7), new Shift(s12));
117     mActionTable.put(new Pair<State, Symbol>(s10, t1), new Shift(s2));
118     mActionTable.put(new Pair<State, Symbol>(s10, t7), new Shift(s12));

```

Abbildung 15.7: Die Klasse ParseTable, 3. Teil.

```

119         mActionTable.put(new Pair<State, Symbol>(s11, t0), new Reduce(rule5));
120         mActionTable.put(new Pair<State, Symbol>(s11, t2), new Reduce(rule5));
121         mActionTable.put(new Pair<State, Symbol>(s11, t3), new Reduce(rule5));
122         mActionTable.put(new Pair<State, Symbol>(s11, t4), new Reduce(rule5));
123         mActionTable.put(new Pair<State, Symbol>(s11, t5), new Reduce(rule5));
124         mActionTable.put(new Pair<State, Symbol>(s11, t6), new Reduce(rule5));
125         mActionTable.put(new Pair<State, Symbol>(s12, t0), new Reduce(rule6));
126         mActionTable.put(new Pair<State, Symbol>(s12, t2), new Reduce(rule6));
127         mActionTable.put(new Pair<State, Symbol>(s12, t3), new Reduce(rule6));
128         mActionTable.put(new Pair<State, Symbol>(s12, t4), new Reduce(rule6));
129         mActionTable.put(new Pair<State, Symbol>(s12, t5), new Reduce(rule6));
130         mActionTable.put(new Pair<State, Symbol>(s12, t6), new Reduce(rule6));
131         mActionTable.put(new Pair<State, Symbol>(s13, t0), new Reduce(rule7));
132         mActionTable.put(new Pair<State, Symbol>(s13, t2), new Reduce(rule7));
133         mActionTable.put(new Pair<State, Symbol>(s13, t3), new Reduce(rule7));
134         mActionTable.put(new Pair<State, Symbol>(s13, t4), new Reduce(rule7));
135         mActionTable.put(new Pair<State, Symbol>(s13, t5), new Reduce(rule7));
136         mActionTable.put(new Pair<State, Symbol>(s13, t6), new Reduce(rule7));
137         mActionTable.put(new Pair<State, Symbol>(s14, t0), new Reduce(rule8));
138         mActionTable.put(new Pair<State, Symbol>(s14, t2), new Reduce(rule8));
139         mActionTable.put(new Pair<State, Symbol>(s14, t3), new Reduce(rule8));
140         mActionTable.put(new Pair<State, Symbol>(s14, t4), new Reduce(rule8));
141         mActionTable.put(new Pair<State, Symbol>(s14, t5), new Reduce(rule8));
142         mActionTable.put(new Pair<State, Symbol>(s14, t6), new Reduce(rule8));
143         mActionTable.put(new Pair<State, Symbol>(s15, t0), new Reduce(rule9));
144         mActionTable.put(new Pair<State, Symbol>(s15, t2), new Reduce(rule9));
145         mActionTable.put(new Pair<State, Symbol>(s15, t3), new Reduce(rule9));
146         mActionTable.put(new Pair<State, Symbol>(s15, t4), new Reduce(rule9));
147         mActionTable.put(new Pair<State, Symbol>(s15, t5), new Reduce(rule9));
148         mActionTable.put(new Pair<State, Symbol>(s15, t6), new Reduce(rule9));
149         mGotoTable = new TreeMap<Pair<State, Symbol>, State>();
150         mGotoTable.put(new Pair<State, Symbol>(s0, expr), s1);
151         mGotoTable.put(new Pair<State, Symbol>(s0, factor), s13);
152         mGotoTable.put(new Pair<State, Symbol>(s0, product), s8);
153         mGotoTable.put(new Pair<State, Symbol>(s2, expr), s3);
154         mGotoTable.put(new Pair<State, Symbol>(s2, factor), s13);
155         mGotoTable.put(new Pair<State, Symbol>(s2, product), s8);
156         mGotoTable.put(new Pair<State, Symbol>(s4, factor), s13);
157         mGotoTable.put(new Pair<State, Symbol>(s4, product), s5);
158         mGotoTable.put(new Pair<State, Symbol>(s6, factor), s13);
159         mGotoTable.put(new Pair<State, Symbol>(s6, product), s7);
160         mGotoTable.put(new Pair<State, Symbol>(s9, factor), s14);
161         mGotoTable.put(new Pair<State, Symbol>(s10, factor), s15);
162     }
163 }

```

Abbildung 15.8: Die Klasse ParseTable, 4. Teil.

15.3 SLR-Parser

In diesem Abschnitt zeigen wir, wie wir für eine gegebene kontextfreie Grammatik G die im letzten Abschnitt verwendeten Funktionen

$$action : Q \times T \rightarrow Action$$

definieren können. Dazu klären wir als erstes, wie die Menge Q der Zustände zu definieren ist. Wir werden die Zustände nun so definieren, dass sie die Information enthalten, welche Regel der Shift-Reduce-Parser anzuwenden versucht, welche Teile der Syntax er bereits erkannt hat und was er noch erwartet. Zu diesem Zweck definieren wir:

Definition 43 (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 verschaulichen, betrachten wir ein konkretes Beispiel: Wir gehen von der in Abbildung 15.4 auf Seite 212 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. $Factor \rightarrow \bullet NUMBER$,

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 44 ($\text{closure}(\mathcal{M})$) 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 15.4 auf Seite 212 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

$$\hat{G} = \langle V \cup \{\hat{S}\}, T, \hat{S}, R \cup \{\hat{S} \rightarrow S\} \rangle.$$

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 beliebige 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

$$action(\mathcal{M}, t) := \langle \mathbf{shift}, goto(\mathcal{M}, t) \rangle \quad \text{falls} \quad (A \rightarrow \alpha \bullet t\beta) \in \mathcal{M}.$$

2. Falls \mathcal{M} eine markierte Regel der Form $A \rightarrow \alpha \bullet$ enthält und wenn zusätzlich $t \in Follow(A)$ gilt, dann setzen wir

$$action(\mathcal{M}, t) := \langle \mathbf{reduce}, A \rightarrow \alpha \rangle,$$

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 $t \in Follow(A)$, so kann der Parser die Regel $A \rightarrow \alpha$ anwenden und den Symbol-Stack mit dieser Regel reduzieren. Wir haben also

$$action(\mathcal{M}, t) := \langle \mathbf{reduce}, A \rightarrow \alpha \rangle \quad \text{falls} \quad (A \rightarrow \alpha \bullet) \in \mathcal{M}, A \neq \hat{S} \text{ und } t \in Follow(A).$$

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

$$action(\mathcal{M}, \text{EOF}) := \mathbf{accept},$$

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

$$action(\mathcal{M}, \text{EOF}) := \mathbf{accept}, \quad \text{falls} \quad (\hat{S} \rightarrow S \bullet) \in \mathcal{M}.$$

4. In den restlichen Fällen setzen wir

$$action(\mathcal{M}, t) := \mathbf{error}.$$

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

$$C \rightarrow \gamma \bullet \quad \text{mit } t \in Follow(C).$$

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.

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 $t \in Follow(C_1) \cap Follow(C_2)$ 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. Da die syntaktische Variable $expr$

auf der rechten Seite von Grammatik-Regeln auftritt, definieren wir *start* als neues Start-Symbol und fügen in der Grammatik die Regel

$$start \rightarrow expr$$

ein. Als erstes berechnen wir die Menge der Zustände Q . Wir hatten dafür oben die folgende Formel angegeben:

$$Q := \{ \mathcal{M} \in 2^\Gamma \mid closure(\mathcal{M}) = \mathcal{M} \}.$$

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:

$$S := start, E := expr, P := product, F := factor, N := NUMBER.$$

Wir beginnen mit dem Start-Zustand:

$$\begin{aligned} s_0 &:= closure(\{ S \rightarrow \bullet E \}) \\ &= \{ S \rightarrow \bullet E, \\ &\quad E \rightarrow \bullet E \text{ "+" } P, E \rightarrow \bullet E \text{ "-" } P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P \text{ "*" } F, P \rightarrow \bullet P \text{ "/" } F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet \text{"(" } E \text{ ")"}, F \rightarrow \bullet N \} \end{aligned}$$

Als nächstes berechnen wir *goto*(s_0, F). Wir bezeichnen den resultierenden Zustand mit s_1 .

$$\begin{aligned} s_1 &:= goto(s_0, F) \\ &= closure(\{ P \rightarrow F \bullet \}) \\ &= \{ P \rightarrow F \bullet \}. \end{aligned}$$

Als nächstes berechnen wir *goto*(s_0, N).

$$\begin{aligned} s_2 &:= goto(s_0, N) \\ &= closure(\{ F \rightarrow N \bullet \}) \\ &= \{ F \rightarrow N \bullet \}. \end{aligned}$$

Als nächstes berechnen wir *goto*(s_0, P).

$$\begin{aligned} s_3 &:= goto(s_0, P) \\ &= closure(\{ P \rightarrow P \bullet \text{ "*" } F, P \rightarrow P \bullet \text{ "/" } F, E \rightarrow P \bullet \}) \\ &= \{ P \rightarrow P \bullet \text{ "*" } F, P \rightarrow P \bullet \text{ "/" } F, E \rightarrow P \bullet \}. \end{aligned}$$

Als nächstes berechnen wir *goto*(s_0, E).

$$\begin{aligned} s_4 &:= goto(s_0, E) \\ &= closure(\{ S \rightarrow E \bullet, E \rightarrow E \bullet \text{ "+" } P, E \rightarrow E \bullet \text{ "-" } P \}) \\ &= \{ S \rightarrow E \bullet, E \rightarrow E \bullet \text{ "+" } P, E \rightarrow E \bullet \text{ "-" } P \}. \end{aligned}$$

Als nächstes berechnen wir *goto*($s_0, \text{"("}$).

$$\begin{aligned} s_5 &:= goto(s_0, \text{"("}) \\ &= closure(\{ F \rightarrow \text{"("} \bullet E \text{ ")" } \}) \\ &= \{ F \rightarrow \text{"("} \bullet E \text{ ")"}, \\ &\quad E \rightarrow \bullet E \text{ "+" } P, E \rightarrow \bullet E \text{ "-" } P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P \text{ "*" } F, P \rightarrow \bullet P \text{ "/" } F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet \text{"("} E \text{ ")"}, F \rightarrow \bullet N \} \end{aligned}$$

Berechnen wir *goto*($s_0, \text{"("}$), so finden wir

$$\text{goto}(s_0, "(") = \{\}.$$

Da dieser Zustand beim Parsen offenbar nicht benutzt werden kann, bekommt er keinen eigenen Namen. Als nächstes berechnen wir $\text{goto}(s_5, E)$.

$$\begin{aligned} s_6 &:= \text{goto}(s_5, E) \\ &= \text{closure}(\{ F \rightarrow "(" E \bullet ")" , E \rightarrow E \bullet "+" P, E \rightarrow E \bullet "-" P \}) \\ &= \{ F \rightarrow "(" E \bullet ")" , E \rightarrow E \bullet "+" P, E \rightarrow E \bullet "-" P \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_6, "(")$.

$$\begin{aligned} s_7 &:= \text{goto}(s_6, "(") \\ &= \text{closure}(\{ F \rightarrow "(" E "(" \bullet " \}) \\ &= \{ F \rightarrow "(" E "(" \bullet " \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_4, "+")$.

$$\begin{aligned} s_8 &:= \text{goto}(s_4, "+") \\ &= \text{closure}(\{ E \rightarrow E "+" \bullet P \}) \\ &= \{ E \rightarrow E "+" \bullet P \\ &\quad P \rightarrow \bullet P "*" F, P \rightarrow \bullet P "/" F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E "(" \bullet " , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_4, "-")$.

$$\begin{aligned} s_9 &:= \text{goto}(s_4, "-") \\ &= \text{closure}(\{ E \rightarrow E "-" \bullet P \}) \\ &= \{ E \rightarrow E "-" \bullet P \\ &\quad P \rightarrow \bullet P "*" F, P \rightarrow \bullet P "/" F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E "(" \bullet " , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_9, P)$.

$$\begin{aligned} s_{10} &:= \text{goto}(s_9, P) \\ &= \text{closure}(\{ E \rightarrow E "-" P \bullet , P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F \}) \\ &= \{ E \rightarrow E "-" P \bullet , P \rightarrow P \bullet "*" F, P \rightarrow P \bullet "/" F \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_3, "/")$.

$$\begin{aligned} s_{11} &:= \text{goto}(s_3, "/") \\ &= \text{closure}(\{ P \rightarrow P "/" \bullet F \}) \\ &= \{ P \rightarrow P "/" \bullet F, F \rightarrow \bullet "(" E "(" \bullet " , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_3, "*")$.

$$\begin{aligned} s_{12} &:= \text{goto}(s_3, "*") \\ &= \text{closure}(\{ P \rightarrow P "*" \bullet F \}) \\ &= \{ P \rightarrow P "*" \bullet F, F \rightarrow \bullet "(" E "(" \bullet " , F \rightarrow \bullet N \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_{12}, F)$.

$$\begin{aligned} s_{13} &:= \text{goto}(s_{12}, F) \\ &= \text{closure}(\{ P \rightarrow P "*" F \bullet \}) \\ &= \{ P \rightarrow P "*" F \bullet \}. \end{aligned}$$

Als nächstes berechnen wir $\text{goto}(s_{11}, F)$.

$$\begin{aligned} s_{14} &:= \text{goto}(s_{11}, F) \\ &= \text{closure}(\{ P \rightarrow P "/" F \bullet \}) \\ &= \{ P \rightarrow P "/" F \bullet \}. \end{aligned}$$

Als letztes berechnen wir $goto(s_8, P)$.

$$\begin{aligned} s_{15} &:= goto(s_8, P) \\ &= closure(\{E \rightarrow E \text{ "+" } P\bullet, P \rightarrow P\bullet \text{ "*" } F, P \rightarrow P\bullet \text{ "/" } F\}) \\ &= \{E \rightarrow E \text{ "+" } P\bullet, P \rightarrow P\bullet \text{ "*" } F, P \rightarrow P\bullet \text{ "/" } F\}. \end{aligned}$$

Weitere Rechnungen führen nicht mehr auf neue Zustände. Berechnen wir beispielsweise $goto(s_8, "(")$, so finden wir

$$\begin{aligned} &goto(s_8, "(") \\ &= closure(\{F \rightarrow "(" \bullet E ")"\}) \\ &= \{F \rightarrow "(" \bullet E ")", \\ &\quad E \rightarrow \bullet E \text{ "+" } P, E \rightarrow \bullet E \text{ "-" } P, E \rightarrow \bullet P, \\ &\quad P \rightarrow \bullet P \text{ "*" } F, P \rightarrow \bullet P \text{ "/" } F, P \rightarrow \bullet F, \\ &\quad F \rightarrow \bullet "(" E ")" , F \rightarrow \bullet N\} \\ &= s_5. \end{aligned}$$

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 $\text{"+"} \in Follow(P)$.

2. Als nächstes berechnen wir $action(s_4, \text{"+"})$. Es gilt

$$\begin{aligned} action(s_4, \text{"+"}) &= action(\{S \rightarrow E\bullet, E \rightarrow E\bullet \text{"+" } P, E \rightarrow E\bullet \text{"-" } P\}, \text{"+"}) \\ &= \langle shift, closure(\{E \rightarrow E \text{"+" } \bullet P\}) \rangle \\ &= \langle shift, s_8 \rangle. \end{aligned}$$

<i>Conjunction</i>	→	<i>Conjunction</i> “&” <i>Disjunction</i>
		<i>Disjunction</i>
<i>Disjunction</i>	→	<i>Disjunction</i> “ ” <i>Literal</i>
		<i>Literal</i>
<i>Literal</i>	→	“!” IDENTIFIER
		IDENTIFIER

Abbildung 15.9: Eine Grammatik für Boole’sche Ausdrücke in konjunktiver Normalform.

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 15.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 15.1 und 15.2 übereinstimmen.

Aufgabe 21: Berechnen Sie die Menge der SLR-Zustände für die in Abbildung 15.9 gezeigte Grammatik und geben Sie die Funktionen *action()* und *goto()* an. Kürzen Sie die Namen der syntaktischen Variablen und Terminale mit *S*, *C*, *D*, *L* und *I* ab, wobei *S* für das neu eingeführte Start-Symbol steht.

Hinweis: Damit Sie später Ihr Ergebnis mit meinem Ergebnis vergleichen können, ist es sinnvoll, die Namen der Zustände wie folgt festzulegen:

1. $s_0 := \text{closure}(\{S \rightarrow \bullet C\})$,
2. $s_1 := \text{goto}(s_0, C)$,
3. $s_2 := \text{goto}(s_0, “!”)$,
4. $s_3 := \text{goto}(s_0, L)$,
5. $s_4 := \text{goto}(s_0, I)$,
6. $s_5 := \text{goto}(s_0, D)$,
7. $s_6 := \text{goto}(s_5, “|”)$,
8. $s_7 := \text{goto}(s_6, L)$,
9. $s_8 := \text{goto}(s_2, I)$,
10. $s_9 := \text{goto}(s_1, “\&”)$,
11. $s_{10} := \text{goto}(s_9, D)$.

15.3.1 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 15.10 gezeigte Grammatik ist mehrdeutig, denn sie legt nicht fest, ob der Operator “+” stärker oder schwächer bindet als der Operator “*”: Interpretieren wir das Nicht-Terminal *N* als eine 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.

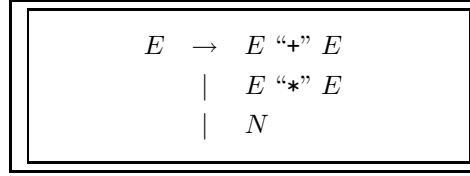


Abbildung 15.10: Eine Grammatik mit Shift-Reduce-Konflikten.

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 $s_1 := \text{goto}(s_0, E)$:

$$\begin{aligned} s_1 &= \text{goto}(s_0, E) \\ &= \text{closure}(\{S \rightarrow E \bullet, E \rightarrow E \bullet \text{ “+” } E, E \rightarrow E \bullet \text{ “*” } E\}) \\ &= \{S \rightarrow E \bullet, E \rightarrow E \bullet \text{ “+” } E, E \rightarrow E \bullet \text{ “*” } E\} \end{aligned}$$

Nun berechnen wir $s_2 := \text{goto}(s_1, \text{“+”})$:

$$\begin{aligned} s_2 &= \text{goto}(s_1, \text{“+”}) \\ &= \text{closure}(\{E \rightarrow E \text{ “+” } \bullet E, \}) \\ &= \{E \rightarrow E \text{ “+” } \bullet E, E \rightarrow \bullet E \text{ “+” } E, E \rightarrow \bullet E \text{ “*” } E, E \rightarrow \bullet N\} \end{aligned}$$

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

$$\text{Follow}(E) = \{ \text{“+”}, \text{“*”} \},$$

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 22: Bei der in Abbildung 15.10 gezeigten Grammatik treten noch weitere Shift-Reduce-Konflikte auf. Berechnen Sie alle Zustände und geben Sie dann die restlichen Shift-Reduce-Konflikte an.

Bemerkung: Es ist nicht weiter verwunderlich, dass wir bei der oben angegebenen Grammatik einen Konflikt gefunden haben, denn diese Grammatik ist nicht eindeutig. Demgegenüber ist kann gezeigt werden, dass jede SLR-Grammatik eindeutig sein muss. Folglich ist eine mehrdeutige Grammatik niemals eine SLR-Grammatik. Die Umkehrung dieser Aussage gilt jedoch nicht. Dies werden wir im nächsten Beispiel sehen. \square

Wir untersuchen als nächstes eine Grammatik, die keine SLR-Grammatik ist, weil Reduce-Reduce-Konflikte auftreten. Wir betrachten dazu die in Abbildung 15.11 gezeigte Grammatik.

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

Abbildung 15.11: Eine Grammatik mit einem Reduce-Reduce-Konflikt.

Wir werden später sehen, dass diese Grammatik eindeutig ist. Wir berechnen zunächst den Start-Zustand eines SLR-Parsers für diese Grammatik.

$$\begin{aligned}
s_0 &= \text{closure}(\{\hat{S} \rightarrow \bullet S\}) \\
&= \{\hat{S} \rightarrow \bullet S, S \rightarrow \bullet A \text{ "x" } A \text{ "y" }, S \rightarrow \bullet B \text{ "y" } B \text{ "x" }, A \rightarrow \bullet \varepsilon, B \rightarrow \bullet \varepsilon\} \\
&= \{\hat{S} \rightarrow \bullet S, S \rightarrow \bullet A \text{ "x" } A \text{ "y" }, S \rightarrow \bullet B \text{ "y" } B \text{ "x" }, A \rightarrow \varepsilon \bullet, B \rightarrow \varepsilon \bullet\},
\end{aligned}$$

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.

15.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 15.11 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 45 (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

$$A \rightarrow \alpha \bullet \beta : t. \quad \square$$

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-Token*.

Mit erweiterten markierten Regeln arbeitet sich ganz ähnlich wie mit markierten Regeln, allerdings müssen wir die Definitionen der Funktionen $\text{closure}()$, goto und $\text{action}()$ etwas modifizieren. Wir beginnen mit der Funktion $\text{closure}()$.

Definition 46 ($\text{closure}(\mathcal{M})$) 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 $\text{closure}(\mathcal{M})$ bezeichnet. \square

Bemerkung: Gegenüber der alten Definition ist nur die Berechnung der Menge der Folge-Token hinzu gekommen. Der Kontext, in dem das B auftritt, das mit der Regel $B \rightarrow \gamma$ 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-Token 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. \square

Für eine gegebene e.m.R.-Menge \mathcal{M} kann die Berechnung von $\mathcal{K} := \text{closure}(\mathcal{M})$ iterativ erfolgen. Abbildung 15.12 zeigt die Berechnung von $\text{closure}(\mathcal{M})$. Der wesentliche Unterschied gegenüber der früheren Berechnung von $\text{closure}()$ ist, dass wir bei den e.m.R.s, die wir für eine Variable B mit in $\text{closure}(\mathcal{M})$ aufnehmen, bei der Menge der Folge-Token den Kontext berücksichtigen, in dem B auftritt. Dadurch gelingt es, die Zustände des Parsers präziser zu beschreiben, als dies bei markierten Regeln der Fall ist.

```

1  procedure closure( $\mathcal{M}$ ) {
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 15.12: Berechnung von $\text{closure}(\mathcal{M})$

Die Berechnung von $\text{goto}(\mathcal{M}, t)$ 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:

$$\text{goto}(\mathcal{M}, X) := \text{closure}\left(\left\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X\beta : L) \in \mathcal{M}\right\}\right).$$

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 $action()$ geändert werden muss. Wir spezifizieren die Berechnung dieser Funktion durch die folgenden bedingten Gleichungen.

1. $(A \rightarrow \alpha \bullet t\beta : L) \in \mathcal{M} \implies action(\mathcal{M}, t) := \langle \text{shift}, goto(\mathcal{M}, t) \rangle.$
2. $(A \rightarrow \alpha \bullet : L) \in \mathcal{M} \wedge A \neq \widehat{S} \wedge t \in L \implies action(\mathcal{M}, t) := \langle \text{reduce}, A \rightarrow \alpha \rangle.$
3. $(\widehat{S} \rightarrow S \bullet : \text{EOF}) \in \mathcal{M} \implies action(\mathcal{M}, \text{EOF}) := \text{accept}.$
4. Sonst: $action(\mathcal{M}, t) := \text{error}.$

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 $action(\mathcal{M}, t)$ dann vor, wenn es zwei e.m.R.s

$$(A \rightarrow \alpha \bullet t\beta : L_1) \in \mathcal{M} \quad \text{und} \quad (B \rightarrow \gamma \bullet : L_2) \in \mathcal{M} \quad \text{mit} \quad t \in L_2$$

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.

Ein *Reduce-Reduce-Konflikt* liegt vor, wenn es zwei e.m.R.s

$$(A \rightarrow \alpha \bullet : L_1) \in \mathcal{M} \quad \text{und} \quad (B \rightarrow \beta \bullet : L_2) \in \mathcal{M} \quad \text{mit} \quad L_1 \cap L_2 \neq \{\}$$

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: Neben den oben angegebenen Konflikten ist auch ein Konflikt zwischen der 1. und der 3. Regel möglich. Da die 3. Regel als ein Spezialfall der 2. Regel angesehen werden kann, sprechend wir dann ebenfalls von einem Shift-Reduce-Konflikt. Darüber hinaus sind noch Konflikte zwischen der 2. und der 3. Regel möglich. Derartige Konflikte werden als Reduce-Reduce-Konflikte bezeichnet.

Beispiel: Wir greifen das Beispiel der in Abbildung 15.11 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände. Um die Schreibweise zu vereinfachen, schreiben wir an Stelle von “EOF” kürzer “\$”.

1. $s_0 := \text{closure}\left(\{\widehat{S} \rightarrow \bullet S : \$\}\right)$
 $= \{\widehat{S} \rightarrow \bullet S : \$, S \rightarrow \bullet A \text{“x”} A \text{“y”} : \$, S \rightarrow \bullet B \text{“y”} B \text{“x”} : \$, A \rightarrow \bullet : \text{“x”}, B \rightarrow \bullet : \text{“y”}\}.$
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 $\text{“x”} \neq \text{“y”}$ keinen Konflikt. Es ist leicht zu sehen, dass auch bei den anderen Zustände keine Konflikte auftreten.

Bemerkung: Die Theorie der kanonischen LR-Parser geht auf Donald E. Knuth zurück, der diese Theorie 1965 entwickelt hat [Knu65].

15.5 LALR-Parser

Die Zahl der Zustände eines LR-Parsers ist oft erheblich größer als die Zahl der Zustände, die ein SLR-Parser der selben Grammatik hätte. Anfangs, als der zur Verfügung stehenden Hauptspeicher 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 15.13 gezeigt Grammatik, die ich dem *Drachenbuch* [ASUL06] entnommen habe. (Das “Drachenbuch” ist das Standardwerk im Bereich Compilerbau.)

Abbildung 15.14 zeigt den sogenannten *LR-Goto-Graphen* für diese Grammatik. Die Knoten dieses Graphen sind die Zustände. Betrachten wir den LR-Goto-Graphen, so stellen wir fest, dass die Zustände s_6 und s_3 sich nur in den Mengen der Folge-Token unterscheiden, denn es gilt einerseits

$$s_6 = \left\{ S \rightarrow \text{“x”} \bullet C : \text{“$”}, C \rightarrow \bullet \text{“x”} C : \text{“$”}, C \rightarrow \bullet \text{“y”} : \text{“$”} \right\},$$

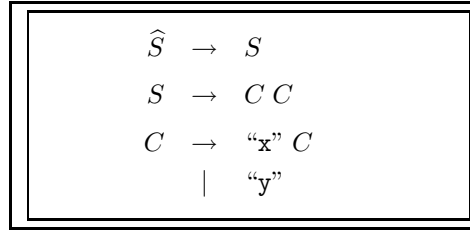


Abbildung 15.13: Eine Grammatik aus dem Drachenbuch.

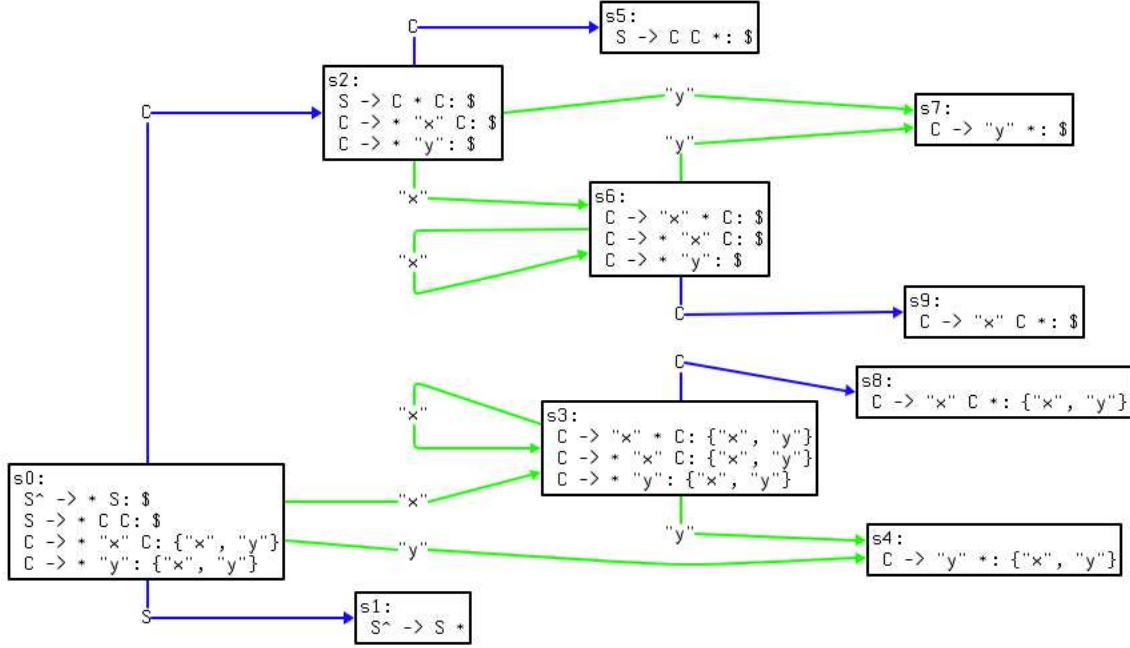


Abbildung 15.14: LR-Goto-Graph für die Grammatik aus Abbildung 15.13.

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

$$goto(s_3, C) = s_8 \quad \text{und} \quad goto(s_6, C) = s_9$$

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 15.15 gezeigten Goto-Graphen.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 15.13 gezeigten Grammatik gemacht haben, verallgemeinern und formalisieren zu können, definieren wir eine Funktion $core()$, die den Kern einer Menge von e.m.R.s berechnet und also diese Menge in eine Menge

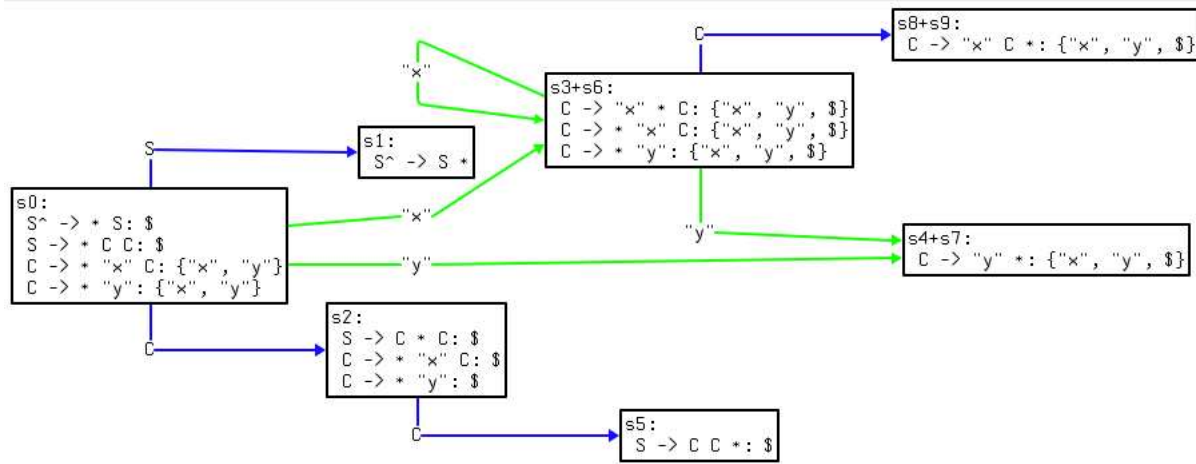


Abbildung 15.15: Der LALR-Goto-Graph für die Grammatik aus Abbildung 15.13.

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 $\text{goto}()$ für eine Menge \mathcal{M} von erweiterten markierten Regeln und ein Symbol X durch

$$\text{goto}(\mathcal{M}, X) := \text{closure}\left(\{A \rightarrow \alpha X \bullet \beta : L \mid (A \rightarrow \alpha \bullet X \beta : L) \in \mathcal{M}\}\right).$$

definiert. Offenbar spielt die Menge der Folge-Tokens bei der Berechnung von $\text{goto}(\mathcal{M}, X)$ keine Rolle, formal gilt für zwei e.m.R.-Mengen \mathcal{M}_1 und \mathcal{M}_2 und ein Symbol X die Formel:

$$\text{core}(\mathcal{M}_1) = \text{core}(\mathcal{M}_2) \Rightarrow \text{core}(\text{goto}(\mathcal{M}_1, X)) = \text{core}(\text{goto}(\mathcal{M}_2, X)).$$

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-Parsers. Dann ist die Menge der Zustände des entsprechenden LALR-Parsers durch die erweiterte Vereinigung der Menge aller der Teilmengen von Δ gegeben, deren Elemente den gleichen Kern haben:

$$\left\{ \uplus \mathcal{J} \mid \mathcal{J} \in 2^\Delta \wedge \forall \mathcal{M}, \mathcal{N} \in \mathcal{J} : \text{core}(\mathcal{M}) = \text{core}(\mathcal{N}) \right\}.$$

Die so definierte Menge ist die Menge der LALR-Zustände. Die Berechnung der Funktionen $\text{goto}()$ und $\text{accept}()$ ändert sich gegenüber den LR-Parsern nicht.

15.6 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:

$$SLR\text{-Sprache} \subsetneq LALR\text{-Sprache} \subsetneq \text{kanonische LR-Sprache} \quad (\star)$$

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.

15.6.1 *SLR-Sprache* \subsetneq *LALR-Sprache*

Die Zustände eines LALR-Parasers enthalten gegenüber den Zuständen eines SLR-Parasers 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 227 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. $s_0 = \{\hat{S} \rightarrow \bullet S : \$, S \rightarrow \bullet A \text{ "x" } A \text{ "y" } : \$, S \rightarrow \bullet B \text{ "y" } B \text{ "x" } : \$, A \rightarrow \bullet : \text{ "x" }, B \rightarrow \bullet : \text{ "y" }\},$
2. $s_1 = \{S \rightarrow A \bullet \text{ "x" } A \text{ "y" } : \$\},$
3. $s_2 = \{\hat{S} \rightarrow S \bullet : \$\},$
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-Parasers überein. Daraus folgt, dass es auch bei den LALR-Zuständen keine Konflikte gibt, denn beim Übergang von kanonischen LR-Parsern zu LALR-Parsern haben wir lediglich Zustände mit gleichem Kern zusammengefasst, die Definition der Funktionen *goto()* und *action()* blieb unverändert.

15.6.2 LALR-Sprache \subsetneq kanonische LR-Sprache

Wir hatten LALR-Parser dadurch definiert, dass wir verschiedene Zustände eines kanonischen LR-Parasers 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 15.16 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 15.16: Eine kanonische LR-Grammatik, die keine LALR-Grammatik ist.

Wir berechnen zunächst die Menge der Zustände eines kanonischen LR-Parasers für diese Grammatik. Wir erhalten dabei die folgende Mengen von erweiterten markierten Regeln:

1. $s_0 = \{ \hat{S} \rightarrow \bullet S : \$, \\ S \rightarrow \bullet \text{"v"} A \text{"y"} : \$, \\ S \rightarrow \bullet \text{"v"} B \text{"z"} : \$, \\ S \rightarrow \bullet \text{"w"} A \text{"z"} : \$, \\ S \rightarrow \bullet \text{"w"} B \text{"y"} : \$ \},$
2. $s_1 = \{ \hat{S} \rightarrow S \bullet : \$ \}$
3. $s_2 = \{ S \rightarrow \text{"v"} \bullet B \text{"z"} : \$, \\ S \rightarrow \text{"v"} \bullet A \text{"y"} : \$, \\ A \rightarrow \bullet \text{"x"} : \text{"y"}, \\ B \rightarrow \bullet \text{"x"} : \text{"z"} \},$
4. $s_3 = \{ S \rightarrow \text{"w"} \bullet A \text{"z"} : \$, \\ S \rightarrow \text{"w"} \bullet B \text{"y"} : \$, \\ A \rightarrow \bullet \text{"x"} : \text{"z"}, \\ B \rightarrow \bullet \text{"x"} : \text{"y"} \},$
5. $s_4 = \{ A \rightarrow \text{"x"} \bullet : \text{"y"}, B \rightarrow \text{"x"} \bullet : \text{"z"} \},$
6. $s_5 = \{ A \rightarrow \text{"x"} \bullet : \text{"z"}, B \rightarrow \text{"x"} \bullet : \text{"y"} \},$
7. $s_6 = \{ S \rightarrow \text{"v"} A \bullet \text{"y"} : \$ \},$
8. $s_7 = \{ S \rightarrow \text{"v"} A \text{"y"} \bullet : \$ \},$

9. $s_8 = \{S \rightarrow \text{"v"} B \bullet \text{"z"} : \$\},$
10. $s_9 = \{S \rightarrow \text{"v"} B \text{"z"} \bullet : \$\},$
11. $s_{10} = \{S \rightarrow \text{"w"} A \bullet \text{"z"} : \$\},$
12. $s_{11} = \{S \rightarrow \text{"w"} A \text{"z"} \bullet : \$\},$
13. $s_{12} = \{S \rightarrow \text{"w"} B \bullet \text{"y"} : \$\},$
14. $s_{13} = \{S \rightarrow \text{"w"} B \text{"y"} \bullet : \$\}.$

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

$$\text{action}(s_{\{4,5\}}, \text{"y"}) = \langle \text{reduce}, B \rightarrow \text{"x"} \rangle.$$

15.6.3 Bewertung der verschiedenen Methoden

Für die Praxis sind SLR-Parser nicht ausreichend, denn es gibt eine Reihe praktisch relevanter Sprach-Konstrukte, für die sich kein SLR-Parser erzeugen läßt. Kanonische LR-Parser sind wesentlich mächtiger, benötigen allerdings oft deutlich mehr Zustände. Hier stellen LALR-Parser einen Kompromiß dar: Einerseits sind LALR-Sprachen fast so ausdrucksstark wie kanonische LR-Sprachen, andererseits liegt der Speicherbedarf von LALR-Parsern in der gleichen Größen-Ordnung wie der Speicherbedarf von SLR-Parsern. In den heute in der Regel zur Verfügung stehenden Hauptspeichern lassen sich allerdings auch kanonische LR-Parser mühelos unterbringen, so dass es eigentlich keinen zwingenden Grund mehr gibt, statt eines LR-Parsers einen LALR-Parser einzusetzen.

Andererseits wird niemand einen LALR-Parser oder einen kanonischen LR-Parser von Hand programmieren wollen. Statt dessen werden Sie später einen Parser-Generator wie *Bison* oder *JavaCup* einsetzen, der Ihnen einen Parser generiert. Aus historischen Gründen erzeugen diese Werkzeuge allerdings nur LALR-Parser, so dass Sie de facto keine Wahl haben.

15.7 Erzeugung von LALR-Parsern mit *Bison*

Die von dem Parser-Generator *Bison* erzeugten Parser sind LALR-Parser. *Bison* bietet die Möglichkeit, die erzeugten Mengen von e.m.R.s textuell oder grafisch darzustellen. Abbildung 15.17 zeigt, wie die Grammatik aus Abbildung 15.13 als Eingabe-Datei für *Bison* formatiert werden kann. Da es

```

1  %%
2  S : C C;
3
4  C : 'x' C
5     | 'y'
6     ;
7  %%

```

Abbildung 15.17: Die Grammatik aus Abbildung 15.13 im *Bison*-Format.

uns hier nur um die Erzeugung der Zustands-Mengen geht, enthält die Datei weder Deklarationen, die zur Anbindung eines Scanners erforderlich wären, noch semantische Aktionen.

Übersetzen wir die in Abbildung 15.17 gezeigte Datei mit dem Befehl

```
bison -v -g cc.y
```

so werden von *Bison* zwei Dateien erzeugt:

1. Die Option “-v” (*verbose*) bewirkt, dass die Datei `cc.output` erzeugt wird. Diese Datei enthält die Zustände des LALR-Parsers und wird in Abbildung 15.18 gezeigt.
2. Die Option “-g” (*graphic*) erzeugt die Datei `cc.vcg`. Die Datei-Endung `.vcg` steht als Abkürzung für *visualization of compiler graphs*. Diese Datei stellt den Goto-Graphen der Grammatik im *VCG-Format* dar und läßt sich mit Hilfe geeigneter Werkzeuge grafisch darstellen. Ein solches Werkzeug ist `aiSee`, das Sie im Internet unter der Adresse

<http://www.aisee.com/>

finden. Dort können Sie eine Demoversion kostenlos herunterladen.

Wir diskutieren nun die in Abbildung 15.18 gezeigte Datei `cc.output` Zeile für Zeile.

1. Die Zeilen 1 – 5 zeigen die zu Grunde liegende Grammatik. Die 0-te Regel hat hier die Form

$$\text{\$accept: S \$end}$$

und entspricht im wesentlichen unserer Regel

$$\widehat{S} \rightarrow S,$$

wobei der String “`\$end`” für das Ende der Eingabe steht. Bei *Bison* wird also das Datei-Ende-Zeichen EOF mit zu der Regel hinzugenommen.

Wie Sie sehen, sind die einzelnen Grammatik-Regeln mit 0 beginnend nummeriert. Auf diese Nummerierung wird Bezug genommen.

2. Die Zeilen 7 – 10 listen alle Terminale auf, die in der Grammatik verwendet werden. Die Zeile

```
'x' (120) 2
```

ist beispielsweise wie folgt zu lesen:

- (a) ‘x’ ist ein Terminal
- (b) mit dem ASCII-Code 120 und
- (c) dieses Terminal tritt in der Regel 2 auf.

Neben den vom Benutzer in der Grammatik definierten Terminalen “x” und “y” führt *Bison* noch die beiden Terminale “`\$end`” und “`error`” ein. Ersteres bezeichnet das Ende der Eingabe, letzteres steht für ein Token, das nicht in der Eingabe-Sprache vorkommt.

```

1  Grammar
2      0 $accept: S $end
3      1 S: C C
4      2 C: 'x' C
5      3   | 'y'
6  Terminals, with rules where they appear
7  $end (0) 0
8  'x' (120) 2
9  'y' (121) 3
10 error (256)
11 Nonterminals, with rules where they appear
12 $accept (5)
13     on left: 0
14 S (6)
15     on left: 1, on right: 0
16 C (7)
17     on left: 2 3, on right: 1 2
18 state 0
19     0 $accept: . S $end
20     'x' shift, and go to state 1
21     'y' shift, and go to state 2
22     S go to state 3
23     C go to state 4
24 state 1
25     2 C: 'x' . C
26     'x' shift, and go to state 1
27     'y' shift, and go to state 2
28     C go to state 5
29 state 2
30     3 C: 'y' .
31     $default reduce using rule 3 (C)
32 state 3
33     0 $accept: S . $end
34     $end shift, and go to state 6
35 state 4
36     1 S: C . C
37     'x' shift, and go to state 1
38     'y' shift, and go to state 2
39     C go to state 7
40 state 5
41     2 C: 'x' C .
42     $default reduce using rule 2 (C)
43 state 6
44     0 $accept: S $end .
45     $default accept
46 state 7
47     1 S: C C .
48     $default reduce using rule 1 (S)

```

Abbildung 15.18: Die Datei cc.output.

3. Die Zeilen 11 – 17 listen die syntaktischen Variable, die hier als Nicht-Terminal bezeichnet werden, auf und geben die Regeln an, wo diese Variablen auftreten. Auch den Variablen werden Zahlen zugeordnet. Diese Zahlen spielen jedoch nur in dem Code des von Bison erzeugten Parsers eine Rolle.
4. Der Rest der Datei zeigt die Zustände und spezifiziert gleichzeitig die Funktionen *goto()* und *action()*. Bei den Zuständen wird allerdings nicht die gesamte Menge der e.m.R. angegeben, sondern es wird nur eine Menge M^- angegeben, aus der sich mit Hilfe der Funktion *closure()* die gesamte Menge nach der Formel

$$M = \text{closure}(M^-)$$

berechnen läßt. Außerdem werden die Mengen der Folge-Token nicht angegeben. Wir betrachten nun zwei der Zustände im Detail:

- (a) Der Zustand “**state 1**”, der in der Zeile 25 spezifiziert wird, entspricht der Menge

$$s_1 = \text{closure}(\{C \rightarrow \text{“x”} \bullet C\})$$

Hier ist zu beachten, dass an Stelle des Markierungs-Symbols “•” ein Punkt “.” verwendet wird.

In Zeile 26 wird spezifiziert, dass

$$\text{goto}(s_1, \text{“x”}) = s_1 \quad \text{und} \quad \text{action}(s_1, \text{“x”}) = \langle \text{shift}, s_1 \rangle$$

gilt. Analog ist Zeile 27 als

$$\text{goto}(s_1, \text{“y”}) = s_2 \quad \text{und} \quad \text{action}(s_1, \text{“x”}) = \langle \text{shift}, s_2 \rangle$$

zu lesen und in Zeile 28 haben wir

$$\text{goto}(s_1, C) = s_5.$$

- (b) Der Zustand “**state 2**”, der in der Zeile 30 spezifiziert wird, entspricht der Menge

$$s_2 = \text{closure}(\{C \rightarrow \text{“y”} \bullet : \{\text{“x”}, \text{“y”}, \$\}\}).$$

Zeile 31 ist als

$$\text{action}(s_2, t) = \langle \text{reduce}, C \rightarrow \text{“x” } C \rangle \quad \text{für alle Token } t$$

zu lesen.

Neben der eben diskutierten textuellen Darstellung bietet *Bison* noch die Option, die Zustände des LALR-Parsers grafisch darzustellen. Dazu erzeugt *Bison* zunächst die Datei `cc.vcg`, die dann beispielsweise mit dem Werkzeug `aieee` visualisiert werden kann. Abbildung 15.19 zeigt diese Visualisierung.

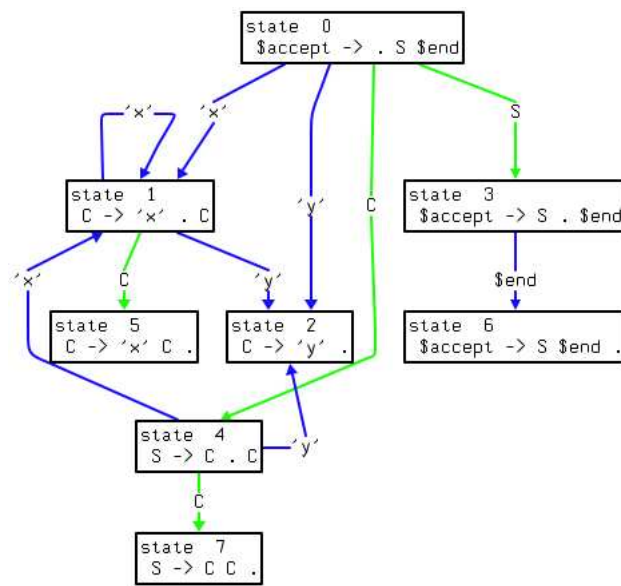


Abbildung 15.19: Von *Bison* erzeugte grafische Darstellung des Goto-Graphen der Grammatik aus Abbildung 15.13.

15.8 Die Behandlung von Konflikten mit *Bison*

Bison kann auch dann noch Grammatiken erzeugen, wenn bei der Konstruktion der Zustände Shift-Reduce- oder Reduce-Reduce-Konflikte auftreten. Wir besprechen an Hand zweier typischer Situationen, wie *Bison* in solchen Fällen vorgeht.

15.8.1 Operator-Präzedenzen

```
1  %token N
2
3  %%
4
5  E : E '+' E
6    | E '*' E
7    | N
8    ;
```

Abbildung 15.20: *Bison*-Darstellung der Grammatik aus Abbildung 15.10.

Wir beginnen mit der in Abbildung 15.10 gezeigten Grammatik für arithmetische Ausdrücke. Abbildung 15.20 zeigt, wie diese Grammatik sich für *Bison* darstellen läßt. Hier haben wir in Zeile 1 festgelegt, dass “N” als Nicht-Terminal zu interpretieren ist. Übersetzen wir diese Grammatik mit dem Befehl

```
bison -v conflict.y,
```

so erhalten wir die Warnung

```
conflict.y: conflicts: 4 shift/reduce.
```

Bei der Übersetzung dieser Grammatik sind also insgesamt 4 Shift-Reduce-Konflikte aufgetreten. Wir wollen diese Konflikte jetzt genauer analysieren und inspizieren zu diesem Zweck die von *Bison* erzeugte Datei `conflict.output`. Die relevanten Teile dieser Datei sind in Abbildung 15.21 wiedergegeben. Wir diskutieren diese Datei jetzt im Detail.

1. Am Anfang der Datei werden alle Konflikte aufgelistet. Bei der von *Bison* übersetzten Grammatik gibt es in den Zuständen “state 6” und “state 7” jeweils zwei Shift-Reduce-Konflikte.
2. Zustand “state 6” besteht aus den Regeln

$$E \rightarrow E \bullet \text{ “+” } E, \quad E \rightarrow E \text{ “+” } E \bullet \quad \text{und} \quad E \rightarrow E \bullet \text{ “*” } E.$$

Hier gibt es zwei Shift-Reduce-Konflikte.

- (a) Bei der Berechnung von `action(state 6, “+”)` gibt es einen Shift-Reduce-Konflikt, denn die markierte Regel $E \rightarrow E \bullet \text{ “+” } E$ verlangt, dass das Token “+” auf den Stack geschoben wird, während die markierte Regel $E \rightarrow E \text{ “+” } E \bullet$ fordert, dass der Symbolstack mit der Grammatik-Regel $E \rightarrow E \text{ “+” } E$ reduziert wird.
Leider werden die Mengen der Folge-Token bei *Bison* nicht mit ausgegeben, so dass die Tatsache, dass das Token “+” tatsächlich ein Folge-Token der markierten Regel $E \rightarrow E \text{ “+” } E$ ist, aus der von *Bison* produzierten Ausgabe nicht ersichtlich ist.
- (b) Bei der Berechnung von `action(state 6, “*”)` gibt es einen Shift-Reduce-Konflikt, denn die markierte Regel $E \rightarrow E \bullet \text{ “*” } E$ verlangt, dass das Token “*” auf den Stack geschoben wird, während die markierte Regel $E \rightarrow E \text{ “+” } E \bullet$ fordert, dass der Symbolstack mit der Grammatik-Regel $E \rightarrow E \text{ “+” } E$ reduziert wird.

```

1  State 6 conflicts: 2 shift/reduce
2  State 7 conflicts: 2 shift/reduce
3
4  Grammar
5      0 $accept: E $end
6      1 E: E '+' E
7      2   | E '*' E
8      3   | N
9
10     :
11
12  state 6
13      1 E: E . '+' E
14      1 | E '+' E .
15      2 | E . '*' E
16      '+' shift, and go to state 4
17      '*' shift, and go to state 5
18      '+' [reduce using rule 1 E]
19      '*' [reduce using rule 1 E]
20      $default reduce using rule 1 E
21  state 7
22      1 E: E . '+' E
23      2 | E . '*' E
24      2 | E '*' E .
25      '+' shift, and go to state 4
26      '*' shift, and go to state 5
27      '+' [reduce using rule 2 E]
28      '*' [reduce using rule 2 E]
29      $default reduce using rule 2 E

```

Abbildung 15.21: Darstellung der Shift-Reduce-Konflikte durch *Bison*.

In den Zeilen 16 – 19 sehen wir, wie diese Konflikte aufgelöst werden: *Bison* bevorzugt bei Shift-Reduce-Konflikten einen Shift. Die auch möglichen Reduktionen sind daher in den Zeilen 18 und 19 mit den eckigen Klammern “[” und “]” eingefasst worden um kenntlich zu machen, dass diese Reduktionen nicht angewendet werden.

3. Die Shift-Reduce-Konflikte, die in dem Zustand “**state 7**” auftreten, sind analog zu den Konflikten im Zustand “**state 6**” und werden daher nicht im Detail diskutiert.

Es ist in *Bison* möglich, Shift-Reduce-Konflikte durch die Angabe von *Operator-Präzedenzen* aufzulösen. Abbildung 15.22 zeigt die *Bison*-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

$3 - 2 - 1$ wird also als $(3 - 2) - 1$ und nicht als $3 - (2 - 1)$

gelesen. Demgegenüber ist der Operator “^”, der die Potenzbildung bezeichnet, *rechts-assoziativ*, der Ausdruck

$4^3 \cdot 2$ wird also als $4^{(3^2)}$ und nicht als $(4^3)^2$

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 15.22 gezeigten Grammatik ordnet *Bison* den Operatoren die Bindungsstärke nach der folgenden Tabelle zu:

Operator	Bindungsstärke
“+”	1
“-”	1
“*”	2
“/”	2
“^”	3

```

1  %token N
2
3  %left  "+" "-"
4  %left  "*" "/"
5  %right "^"
6
7  %%
8
9  E : E "+" E
10   | E "-" E
11   | E "*" E
12   | E "/" E
13   | E "^" E
14   | "(" E ")"
15   | N
16   ;

```

Abbildung 15.22: Auflösung der Shift-Reduce-Konflikte durch Operator-Präzedenzen.

Wie erläutern nun, wie diese Bindungsstärken benutzt werden, um Shift-Reduce-Konflikte aufzulösen. *Bison* 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
$E \rightarrow \text{ “(” } E \text{ “)”}$	—
$E \rightarrow N$	—

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

$$action(s, o)$$

zunächst einen Shift-Reduce-Konflikt. Ist nun o ein Operator, für den eine Präzedenz $p(o)$ festgelegt worden ist und hat außerdem die Regel r_2 , mit der reduziert werden würde, die Präzedenz $p(r_2)$ so wird der Shift-Reduce-Konflikt in Abhängigkeit von der relativen Größe dieser beiden Zahlen aufgelöst. Hier werden drei Fälle unterschieden:

- (a) $p(o) > p(r_2)$: In diesem Fall bindet der Operator o stärker. Daher wird das Token o in diesem Fall auf den Stack geschoben:

$$action(s, o) = \langle \text{shift}, goto(s, o) \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$s = 1+2*3$$

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) $p(o) < p(r_2)$: Jetzt bindet der Operator, der in der Regel r_2 auftritt, stärker als der Operator o . Daher wird in diesem Fall zunächst mit der Regel r_2 reduziert, wir haben also

$$action(s, o) = \langle \text{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$s = 1*2+3$$

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 soll der bisher gelesene String "1*2" mit der Regel $E \rightarrow E \text{"*" } E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

- (c) $p(o) = p(r_2)$ und der Operator o ist links-assoziativ: Dann wird zunächst mit der Regel r_2 reduziert, wir haben also

$$action(s, o) = \langle \text{reduce}, r_2 \rangle.$$

Dass diese Regel sinnvoll ist, sehen wir, wenn wir beispielsweise den Eingabe-String

$$s = 1*2*3$$

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 soll der bisher gelesene String “1*2” mit der Regel $E \rightarrow E \text{ “*” } E$ reduziert werden, denn wir wollen die 1 ja zunächst mit 2 multiplizieren.

- (d) $p(o) = p(r_2)$ und der Operator o ist rechts-assoziativ: In diesem Fall wird o auf den Stack geschoben:

$$action(s, o) = \langle \text{shift}, goto(s, o) \rangle.$$

Wenn wir diesen Fall versethen wollen, reicht es aus, den String

$$s = 2^3^4$$

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) $p(o) = p(r_2)$ und der Operator o hat keine Assoziativität-assoziativ: 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

$$E \rightarrow E \text{ “<” } E \mid E \text{ “+” } E \mid \text{NUMBER}$$

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.

Um in *Bison* einen Operator o als nicht-assoziativ zu deklarieren, schreiben Sie:

$$\%nonassoc \ o$$

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 *Bison* erzeugte Datei “calc-precedence.output” zeigt im Detail, wie die Shift-Reduce-Konflikt Konflikte aufgelöst worden sind. Wir betrachten exemplarisch zwei Zustände in dieser Datei.

1. Der Zustand “state 12” hat die in Abbildung 15.23 gezeigte Form. Hier gibt es zunächst

einen Shift-Reduce-Konflikt zwischen den beiden markierten Regeln

$$E \rightarrow E \bullet \text{ "+" } E \quad \text{und} \quad E \rightarrow E \text{ "+" } E \bullet,$$

denn die erste Regel verlangt nach einem Shift, während die zweite Regel eine Reduktion fordert. Da die Regel $E \rightarrow E \text{ "+" } E$ die selbe Präzedenz wie der Operator "+" hat, spielt nun die Assoziativität eine Rolle. Da dieser Operator links-assoziativ ist, wird mit dieser Regel reduziert. Dies ist Teil des in Zeile 14 spezifizierten Falls.

```

1  state 12
2
3      1 E: E . "+" E
4      1 | E "+" E .
5      2 | E . "-" E
6      3 | E . "*" E
7      4 | E . "/" E
8      5 | E . "^" E
9
10     "*"  shift, and go to state 8
11     "/"  shift, and go to state 9
12     "^"  shift, and go to state 10
13
14     $default  reduce using rule 1 (E)

```

Abbildung 15.23: Der Zustand "state 12"

Der Zustand "state 12" enthält noch weitere Shift-Reduce-Konflikte. Beispielsweise besteht zwischen den beiden Regeln

$$E \rightarrow E \bullet \text{ "*" } E \quad \text{und} \quad E \rightarrow E \text{ "+" } E \bullet,$$

ein Shift-Reduce-Konflikt bei der Berechnung von $\text{action}(\text{state 12}, \text{"*"})$. Da der Operator "*" die Priorität 2 hat, während die Regel $E \rightarrow E \text{ "+" } E$ nur die Priorität 1 hat, wird dieser Konflikt wie in Zeile 10 gezeigt durch einen Shift aufgelöst.

2. Der Zustand "state 16" hat die in Abbildung 15.24 gezeigte Form. Zunächst gibt es hier einen Shift-Reduce-Konflikt zwischen den Regeln

$$E \rightarrow E \bullet \text{ "^" } E \quad \text{und} \quad E \rightarrow E \text{ "^" } E \bullet,$$

wenn das nächste Token der Operator "^" ist. Da der Operator die selbe Präzedenz hat wie die Regel, entscheidet wieder die Assoziativität. Nun ist der Operator "^" rechts-assoziativ, daher wird in diesem Fall geshiftet.

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

$$E \rightarrow E \bullet \text{ "+" } E \quad \text{und} \quad E \rightarrow E \text{ "^" } E \bullet,$$

der auftritt, wenn das nächste Token ein "+" ist. Da die Regel $E \rightarrow E \text{ "^" } E$ die Präzedenz 3 hat, die größer ist als die Präzedenz 1 des Operators "+" wird dieser Konflikt dadurch aufgelöst, dass mit der Regel $E \rightarrow E \text{ "^" } E$ reduziert wird.

```

1  state 16
2
3      1 E: E . "+" E
4      2 | E . "-" E
5      3 | E . "*" E
6      4 | E . "/" E
7      5 | E . "^" E
8      5 | E "^" E .
9
10     "^"  shift, and go to state 10
11
12     $default  reduce using rule 5 (E)

```

Abbildung 15.24: Der Zustand “state 16”

15.8.2 Das *Dangling-Else*-Problem

```

1  %token ID EXPR
2  %%
3
4  Statement      : "if" "(" EXPR ")" Statement
5                  | "if" "(" EXPR ")" Statement "else" Statement
6                  | "while" "(" EXPR ")" Statement
7                  | "{" StatementList "}"
8                  | ID "=" EXPR ";",
9                  ;
10
11 StatementList  : /* epsilon */
12                  | StatementList Statement
13                  ;

```

Abbildung 15.25: 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 15.25 zeigt eine Grammatik, die einen Teil der Syntax von Befehlen der Sprache C beschreibt. Um uns auf das wesentliche konzentrieren zu können, habe ich dort “EXPR” als Terminal definiert, denn wie arithmetische Ausdrücke mit Hilfe von *Bison* behandelt werden können, haben wir ja schon im letzten Abschnitt gesehen. 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 *Bison*, so erhalten wir den in Abbildung 15.26 ausschnittsweise gezeigten Shift-Reduce-Konflikt.

Der Konflikt entsteht bei der Berechnung von `action(state 19, else)` zwischen den beiden markierten Regeln

$$Statement \rightarrow \text{“if” “(” EXPR “)” Statement} \bullet \quad \text{und}$$

$$Statement \rightarrow \text{“if” “(” EXPR “)” Statement} \bullet \text{ “else” Statement}.$$

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 15.25 gezeigte Grammatik mehrdeutig ist, denn ein *Statement* der Form

```

1  state 19
2
3      1 Statement: "if" "(" EXPR ")" Statement .
4      2          | "if" "(" EXPR ")" Statement . "else" Statement
5
6      "else"  shift, and go to state 21
7
8      "else"    [reduce using rule 1 (Statement)]
9      $default  reduce using rule 1 (Statement)

```

Abbildung 15.26: Zustand der Grammatik aus Abbildung 15.25, bei dem der Shift-Reduce-Konflikt auftritt.

```
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 15.25 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-Präedenzen 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. Das einzige, was dann noch stört, ist von dem Shift-Reduce-Konflikt erzeugte Warnung. Diese kann mit Hilfe der Option

```
%expect n
```

mit der angegeben wird, dass wir genau n Konflikte erwarten, unterdrückt werden. Das führt zu der in Abbildung 15.27 gezeigten Grammatik.

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.

```

1  %expect 1
2  %token ID EXPR
3  %%
4
5  Statement      : "if" "(" EXPR ")" Statement
6                  | "if" "(" EXPR ")" Statement "else" Statement
7                  | "while" "(" EXPR ")" Statement
8                  | "{" StatementList "}"
9                  | ID "=" EXPR ";"
10                 ;
11
12  StatementList : /* epsilon */
13                | StatementList Statement
14                ;

```

Abbildung 15.27: Unterdrückung von Warnungen durch `expect`.

- (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 15.28 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.

```

1  %token ID EXPR
2  %%
3
4  Statement      : MatchedStmnt
5                  | UnMatchedStmnt
6                  ;
7
8  MatchedStmnt   : "if" "(" EXPR ")" MatchedStmnt "else" MatchedStmnt
9                  | "while" "(" EXPR ")" MatchedStmnt
10                 | "{" StatementList "}"
11                 | ID "=" EXPR ";"
12                 ;
13
14  UnMatchedStmnt : "if" "(" EXPR ")" Statement
15                  | "if" "(" EXPR ")" MatchedStmnt "else" UnMatchedStmnt
16                  | "while" "(" EXPR ")" UnMatchedStmnt
17                  ;
18
19  StatementList : /* epsilon */
20                | StatementList Statement
21                ;

```

Abbildung 15.28: Eine eindeutige Grammatik für C-Befehle.

Aus theoretischer Sicht ist das Umschreiben der Grammatik der sauberste Weg. Aus diesem Grund haben die Entwickler der Sprache *Java* in der ersten Version der Spezifikation dieser

Sprache [GJS96] diesen Weg auch beschritten. Sie finden die *Java*-Grammatik im Netz unter

http://java.sun.com/docs/books/jls/first_edition/html/19.doc.html

Der Nachteil ist allerdings, dass bei diesem Vorgehen die Grammatik stark aufgebläht wird. Vermutlich aus diesem Grunde findet sich in der zweiten Auflage der Sprach-Spezifikation eine Grammatik, bei der das *Dangling-Else*-Problem wieder auftritt.

3. Die letzte Möglichkeit um das *Dangling-Else*-Problem zu lösen, besteht darin, dass wir “if” und “else” als Operatoren auffassen, denen wir eine Präzedenz zuordnen. Abbildung 15.29 zeigt die Umsetzung dieser Idee.

- (a) Zunächst haben wir in den Zeilen 3 und 4 “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 15.26 gezeigten Zustand ist.

- (b) In Zeile 8 haben wir der Regel

$$\text{Statement} \rightarrow \text{“if” “(” EXPR “)” Statement}$$

explizit mit Hilfe der nachgestellten Option

```
%prec "if"
```

die Präzedenz des Operators “if” zugewiesen. Dies ist notwendig, weil der letzte Operator, der in dieser Regel auftritt, die schließende runde Klammer “)” 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.

```
1  %token ID EXPR
2
3  %nonassoc "if"
4  %nonassoc "else"
5
6  %%
7
8  Statement      : "if" "(" EXPR ")" Statement           %prec "if"
9                  | "if" "(" EXPR ")" Statement "else" Statement
10                 | "while" "(" EXPR ")" Statement
11                 | "{" StatementList "}"
12                 | ID "=" EXPR ";"
13                 ;
14
15  StatementList : /* epsilon */
16                 | StatementList Statement
17                 ;
```

Abbildung 15.29: Auflösung des Shift-Reduce-Konflikts mit Hilfe von Operator-Präzedenzen.

Kapitel 16

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 für *Java* das, was der Parser-Generator *Bison* für die Sprache *C* ist. Wir werden die Version 0.11a verwenden. Sie finden diese Version im Netz unter

<http://www2.cs.tum.edu/projects/cup/java-cup-11a.jar>

Eine *Cup*-Spezifikation besteht aus fünf Teilen, die allerdings im Gegensatz zu einer *Bison*-Spezifikation nicht durch Trennzeichen von einander abgesetzt werden.

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 16.1 auf Seite 252 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 16.4 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          UMINUS, LPAREN, RPAREN;
5  terminal Integer NUMBER;
6
7  nonterminal      expr_list, expr_part;
8  nonterminal Integer expr;
9
10 precedence left  PLUS, MINUS;
11 precedence left  TIMES, DIVIDE, MOD;
12 precedence right UMINUS;
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       | NUMBER:n      { : RESULT = n; : }
27       | MINUS expr:e   { : RESULT = - e; : } %prec UMINUS
28       | LPAREN expr:e RPAREN { : RESULT = e; : }
29       ;

```

Abbildung 16.1: 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 256 in Abbildung 16.4 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. In den Zeilen 10 bis 12 ordnen wir den Operatoren *Präzedenzen* und *Assoziativitäten* zu. Diese werden dazu benötigt, um *Shift-Reduce-Konflikte* aufzulösen. Das funktioniert genau wie bei *Bison*: Zunächst werden die Operatoren, die am schwächsten binden, angegeben. In Abbildung 16.1 haben wir also spezifiziert, dass die Operatoren “+” und “-” am schwächsten binden. Anschließend kommen die Operatoren “*”, “/” und “%” und der Operator “UMINUS”, der für ein unäres Minus-Zeichen steht, bindet am stärksten. Bei der Lektüre des in Abbildung 16.4 gezeigten Scanners fällt auf, dass der Scanner an keiner Stelle das Token `UMINUS` zurück gibt. Dieses Token ist nur ein Hilfstoken, dass benötigt wird, um eine Präzedenz-Klasse zu definieren, die über den Präzedenzen der echten Operatoren liegt, denn damit können wir später in Zeile 27 erklären, dass die Grammatik-Regel

$$\text{expr} \rightarrow \text{“-” expr}$$

ein Präzedenz hat, die höher ist als die Präzedenz aller anderen Operatoren. Die Präzedenzen und Assoziativitäten haben in einer CUP-Grammatik die gleiche Bedeutung wie bei einer *Bison*-Grammatik, die Details werden wir im nächsten Abschnitt noch einmal diskutieren.

6. 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) `bodyi` ist der Rumpf der *i*-ten Grammatik-Regel, der aus einer Liste von Terminalen und syntaktischen Variablen besteht.
- (c) `actioni` 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 *Bison*-Syntax 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 *Bison* die geschweiften Klammern “{” und “}” ausgereicht haben.
- (c) Um auf die Werte, die einem Terminal oder einer syntaktischen Variablen zugeordnet sind, zuzugreifen, hatten wir bei *Bison* die Notation “\$1”, “\$2”, ... 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 an Stelle des Strings “\$\$” nun das Schlüsselwort “RESULT”. Betrachten wir als Beispiel die Regel

```
expr ::= expr:e1 PLUS expr:e2 { : RESULT = e1 + e2; : };
```

Mit `expr:e1` sucht der Parser nach einem arithmetischen Ausdruck, dessen Wert in der Variablen `e1` gespeichert wird. Anschließend wird ein Plus-Zeichen gelesen und darauf folgt wieder ein arithmetischer Ausdruck, dessen Wert jetzt in `e2` gespeichert wird. Der Wert des insgesamt gelesenen Ausdrucks wird dann durch die Zuweisung

```
RESULT = e1 + e2;
```

berechnet und der linken Seite der Grammatik-Regel zugewiesen.

- (d) Ein weiterer Unterschied zwischen CUP und *Bison* besteht darin, dass Nicht-Terminale nicht mehr durch den ihnen zugeordneten String repräsentiert werden können. Daher können wir den String "PLUS" nicht durch den String "+" ersetzen. An dieser Stelle sind CUP-Spezifikationen leider nicht ganz so gut lesbar wie *Bison*-Spezifikationen.

Um aus der in Abbildung 16.1 gezeigten CUP-Spezifikation einen Parser zu erzeugen, müssen wir diese zunächst mit dem Befehl

```
java java_cup.Main calc.cup
```

übersetzen. Damit dies funktioniert müssen Sie die Variable `CLASSPATH` so setzen, dass die Klasse `java_cup.Main` gefunden werden kann. Eine Möglichkeit, den Aufruf zu vereinfachen, besteht unter Unix darin, dass Sie sich eine Datei `cup` mit folgendem Inhalt irgendwo in Ihrem Pfad ablegen:

```
#!/bin/bash
CLASSPATH=/Users/stroetma/Software/JavaCUP-New/java-cup-11a.jar
java -cp $CLASSPATH java_cup.Main $@
```

In dieser Datei müssen Sie die Variable `CLASSPATH` natürlich so anpassen, dass die Datei

```
java-cup-11a.jar
```

gefunden werden kann. Danach können Sie *JavaCup* einfacher mit dem Befehl

```
cup calc.cup
```

aufrufen. Dieser Befehl erzeugt verschiedene *Java*-Dateien.

1. Die Datei `parser.java` enthält die Klasse `parser`, die den eigentlichen Parser enthält. Wenn Sie diese Klasse später übersetzen wollen, müssen Sie dafür sorgen, dass die Datei `java-cup-11a.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 16.2 auf Seite 255 zeigt diese Klasse.

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 16.3 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.

16.0.3 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 16.4 auf Seite 256 zeigt einen solchen Scanner, den wir jetzt im Detail diskutieren.

```

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 16.2: Die Klasse `sym`.

```

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 16.3: Die Klasse `Calculator`.

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 16.2 auf Seite 255 zeigt diese von CUP erzeugte Klasse.

```

1  import java_cup.runtime.*;
2
3  %%
4
5  %char
6  %line
7  %column
8  %cup
9
10 % {
11     private Symbol symbol(int type) {
12         return new Symbol(type, yychar, yychar + yylength());
13     }
14
15     private Symbol symbol(int type, Object value) {
16         return new Symbol(type, yychar, yychar + yylength(), value);
17     }
18 }
19
20 %%
21
22 ";"          { return symbol( sym.SEMI    ); }
23 "+"          { return symbol( sym.PLUS    ); }
24 "-"          { return symbol( sym.MINUS   ); }
25 "*"          { return symbol( sym.TIMES   ); }
26 "/"          { return symbol( sym.DIVIDE  ); }
27 "%"          { return symbol( sym.MOD     ); }
28 "("          { return symbol( sym.LPAREN  ); }
29 ")"          { return symbol( sym.RPAREN  ); }
30
31 [1-9][0-9]*|0 { return symbol(sym.NUMBER, new Integer(yytext())); }
32
33 [ \t\n]      { /* skip white space */ }
34
35 [^]          { throw new Error("Illegal character '" + yytext() +
36                                "' at line " + yyline +
37                                ", column " + yycolumn); }

```

Abbildung 16.4: Ein Scanner für arithmetische Ausdrücke

(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.

(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 Methoden verwenden wir die Funktion `yylength()`. Diese Funktion gibt die Länge des Strings zurück, der dem zuletzt erkannten Tokens entspricht.

5. In den Zeilen 22 bis 29 erkennen wir die arithmetischen Operatoren und die Klammer-Symbole. Wir verwenden dabei die in der Klasse `sym` definierten Konstanten.
6. In Zeile 31 erkennen wir mit dem regulären Ausdruck “[1-9][0-9]*|0” eine natürliche Zahl. Diese verwandeln wir durch Aufruf des Konstruktors

`Integer(String s)`

in ein Objekt vom Typ `Integer`, wobei der Text, der der Zahl entspricht, von der Funktion `yytext()` geliefert wird. Anschließend geben wir ein Symbol zurück, in dem dieses Objekt als zugehöriger Wert abgespeichert wird.

7. In Zeile 33 überlesen wir Leerzeichen, Tabulatoren und Zeilen-Umbrüche. Das Überlesen geschieht dadurch, dass wir in diesem Fall kein Symbol an den Parser zurück geben, denn die semantische Aktion enthält keinen `return`-Befehl.
8. Falls ein beliebiges anderes Zeichen gelesen wird, geben wir mit der Regel, die in Zeile 35 beginnt, eine Fehlermeldung aus. Dabei greifen wir auf die Variablen `yyline` und `yycolumn` zurück, damit der Fehler lokalisiert werden kann.

Wir werden im nächsten Abschnitt sehen, wie dieser Scanner an einen CUP-Parser angeschlossen werden kann.

16.0.4 Präzedenz und Assoziativität

Der in Abbildung 16.1 gezeigten Spezifikation liegt die in Abbildung 16.5 auf Seite 258 gezeigte Grammatik zu Grunde. Diese Grammatik ist sehr kompakt, hat aber ein Problem, denn Sie ist mehrdeutig. 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. Die Konvention *Punkt-vor-Strich* sagt uns, dass die letzte der beiden Interpretationen gemeint ist. Diese Konvention können wir in einer CUP-Spezifikation durch die Angabe von Präzedenzen realisieren.

Würden wir die Präzedenz-Deklarationen in den Zeilen 10 bis 12 der Abbildung 16.1 weglassen, so würde die Mehrdeutigkeit der Grammatik dazu führen, dass CUP bei der Erzeugung des Parsers auf eine Reihe von Shift-Reduce-Konflikten stoßen. Unter anderem würden wir die folgende Fehlermeldung erhalten:

```
*** 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

$$R_1 := \left(\text{expr} \rightarrow \text{expr} \text{ “*” } \text{expr} \right)$$

und der markierten Regel

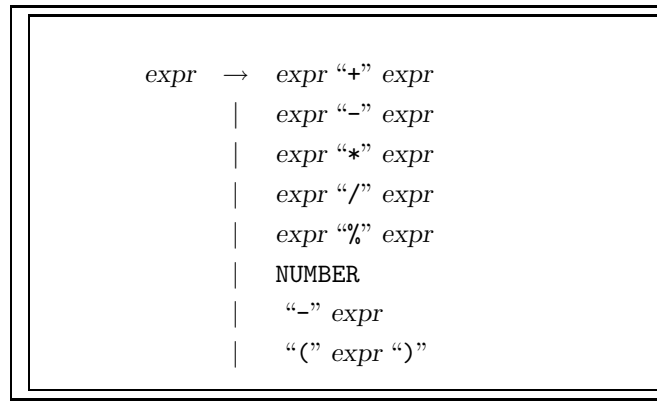


Abbildung 16.5: Eine mehrdeutige Grammatik für arithmetische Ausdrücke.

$$R_2 := \left(expr \rightarrow expr \star \text{ "+" } expr \right)$$

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:

```
lalr_state [12]: {
  [expr ::= expr (*) MOD expr ,    {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) MINUS expr ,   {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) DIVIDE expr ,  {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr TIMES expr (*) ,   {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) TIMES expr ,   {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
  [expr ::= expr (*) PLUS expr ,    {EOF PLUS MINUS TIMES DIVIDE MOD RPAREN}]
}
```

Im Unterschied zu *Bison* werden hier auch die Folge-Token-Mengen mit ausgegeben. Damit können wir jetzt den Shift-Reduce-Konflikt interpretieren: Im Zustand 12 ist der Parser entweder dabei, die Eingabe mit der Regel

$$expr \rightarrow expr \text{ "*" } expr$$

zu reduzieren, oder der Parser ist gerade dabei, die rechte Seite der Regel

$$expr \rightarrow expr \text{ "+" } expr$$

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. Solche Shift-Reduce-Konflikte können von CUP genauso wie bei *Bison* durch die Angabe von Operator-Präzedenzen aufgelöst werden. Der Mechanismus ist der selbe wie bei *Bison*.

Operator-Präzedenzen sind ein mächtiges Mittel um eine Grammatik zu strukturieren. Sie sollten allerdings mit Vorsicht eingesetzt werden, denn Sprachen wie die Programmier-Sprache C, bei der es 15 verschiedene Operator-Präzedenzen gibt, überfordern die meisten Benutzer.

Kapitel 17

Typ-Überprüfung

Von den Skriptsprachen wie *TCL*, *Perl*, *Python*, *RUBY*, etc. abgesehen verfügen die meisten heute verwendeten Programmier-Sprachen über ein *Typ-System*. Eine Sprache, die über ein *Typ-System* verfügt, wird als *getypte* Sprache bezeichnet. Bei einer getypten Sprache ist der Benutzer verpflichtet, die Typen aller verwendeten Variablen anzugeben. Der Compiler kann mit diesen Typ-Informationen dann sicherstellen, dass keine unzulässigen Operationen durchgeführt werden. Auf diese Weise können eine ganze Reihe von Fehlern schon vor der Ausführung des Programms vom Compiler entdeckt werden. Beispielsweise kann der Compiler erkennen, ob eine Methode mit Argumenten aufgerufen werden, deren Typ dem bei der Definition der Methode festgelegten Argument-Typen entspricht. Getypte Sprachen haben gegenüber den ungetypten Sprachen allerdings den Nachteil, dass die Programme etwas länger werden, denn es kommen die Typ-Deklarationen hinzu. Dieser Nachteil wird durch die verbesserte Wartbarkeit getypter Programme mehr als wett gemacht: Einfache Schreibfehler, aber auch einige konzeptuelle Fehler können von einem guten Typ-System automatisch aufgespürt werden.

```
1  class Hand {
2      Integer mValue;
3      String  mSuit;
4
5      Hand(Integer value, String suit) {
6          mValue = value;
7          mSuit  = suit;
8      }
9      :
10 }
11
12 public class TypeError {
13     public static void main(String[] args) {
14         Hand h1 = new Hand( 14, "spades");
15         Hand h2 = new Hand("Ace", "spades");
16     }
17 }
```

Abbildung 17.1: Falscher Aufruf einer Methode

Abbildung 17.1 zeigt ein einfaches Beispiel für einen Typ-Fehler: In Zeile 15 wird der Konstruktor der lokalen Klasse `Hand` mit zwei Strings als Argumenten aufgerufen. Der tatsächlich in Zeile 5

definierte Konstruktor benötigt als erstes Argument aber keinen String, sondern eine ganze Zahl. Dieser Fehler wird vom JAVA-Compiler gefunden und wir bekommen die in Abbildung 17.2 gezeigte Fehlermeldung. Bei dem obigen winzigen Programm ist der Fehler offensichtlich, wären die Klasse `Hand` und `TypeError` aber von verschiedenen Personen in verschiedenen Dateien definiert worden, dann ist die vom Compiler durchgeführte Typ-Überprüfung ein sehr wichtiges Hilfsmittel um Fehler zu vermeiden. Insgesamt wird durch die Typ-Überprüfung die Wartbarkeit wesentlich erhöht, so dass bei großen Software-Projekten praktisch ausschließlich getypte Sprachen eingesetzt werden. Wir wollen uns daher in diesem Kapitel mit der Typ-Überprüfung beschäftigen.

```

1  TypeError.java:15: cannot find symbol
2  symbol   : constructor Hand(java.lang.String,java.lang.String)
3  location: class Hand
4      Hand h2 = new Hand("Ace", "s");
5                      ^

```

Abbildung 17.2: Fehlermeldung des *Java*-Compilers

17.1 Eine Beispielsprache

Wir stellen jetzt die Sprache TTL (*typed term language*) vor. Dabei handelt es sich um eine sehr einfache Beispielsprache, die es dem Benutzer ermöglicht

- Typen zu definieren,
- Funktionen zu deklarieren und
- Terme anzugeben,

für die dann die Typ-Korrektheit nachgewiesen wird. Die Sprache TTL ist sehr einfach gehalten, damit wir uns auf die wesentlichen Ideen der Typ-Überprüfung konzentrieren können. Daher können wir in TTL auch keine wirklichen Programme schreiben, sondern einzig und allein überprüfen, ob Terme wohlgetypt sind.

```

1  type list(X) := nil + cons(X, list(X));
2
3  signature concat: list(T) * list(T) -> list(T);
4  signature x: int;
5  signature y: int;
6  signature z: int;
7
8  concat(nil, nil): list(int);
9  concat(cons(x, nil), cons(y, cons(z, nil))): list(int);

```

Abbildung 17.3: Ein TTL-Beispiel-Programm

Abbildung 17.3 zeigt ein einfaches Beispiel-Programm. Die Schlüsselwörter habe ich unterstrichen. Wir diskutieren dieses Programm jetzt im Detail.

1. In Zeile 1 definieren wir den *generischen* Typ `list(X)`. Das `X` ist hier der Typ-Parameter, der später durch einen konkreten Typ wie z.B. `int`, `string` oder `list(string)` ersetzt werden kann.

Semantisch ist die Zeile als induktive Definition zu lesen, durch die eine Menge von Termen definiert wird, wobei X eine gegebene Menge bezeichnet. nil und $cons$ werden in diesem Zusammenhang als Funktions-Zeichen verwendet. Formal hat die induktive Definition die folgende Gestalt:

- (a) Induktions-Anfang: Der Term nil ist ein Element der Menge $List(X)$:

$$nil \in list(X)$$

- (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 $list(X)$:

$$a \in X \wedge l \in list(X) \rightarrow cons(a, l) \in list(X).$$

- 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.
- In den Zeilen 4 – 6 legen wir fest, dass die Variablen x , y und z jeweils den Typ `int` haben.
- In Zeile 8 und 9 werden schließlich die beiden Terme

$$concat(nil, nil) \quad \text{und} \quad concat(cons(x, nil), cons(y, cons(z, nil)))$$

angegeben und es wird behauptet, dass diese den Typ `list(int)` haben. Die Aufgabe der Typ-Überprüfung besteht darin, diese Aussage zu verifizieren.

Die genaue Syntax der Sprache TTL wird durch die in Abbildung 17.4 gezeigte Grammatik definiert. Diese Grammatik verwendet neben den Zeichen-Reihen, die in doppelten Anführungs-Zeichen gesetzt sind, die folgenden Terminale:

- FUNCTION bezeichnet entweder einen Typ-Konstruktor wie `list`, eine Variable wie x oder y oder einen Funktionsnamen wie `concat`. Syntaktisch werden diese dadurch erkannt, dass sie mit einem Kleinbuchstaben beginnen.
- 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.

17.2 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:

- Die Funktion f erwartet n Argumente.
- Das i -te Argument hat den Typ σ_i .
- Das von der Funktion berechnete Ergebnis hat den Typ ϱ .

```

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'|'A'..'Z'|'_'|'0'..'9')*;
27 FCT       : ('a'..'z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

```

Abbildung 17.4: Eine EBNF-Grammatik für die getypte Beispielsprache

Als nächstes definieren wir, was wir unter einem Typ verstehen wollen. Anschaulich sind das Ausdrücke wie

$\text{map}(K, V)$, double , oder $\text{list}(\text{int})$.

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:

$$f \in \mathbb{K} \wedge \text{arity}(f) = n \wedge n > 0 \wedge \tau_1 \in \mathcal{T} \wedge \dots \wedge \tau_n \in \mathcal{T} \Rightarrow f(\tau_1, \dots, \tau_n) \in \mathcal{T}.$$

Definition 47 (Parameter-Substitution) Eine Parameter-Substitution ist eine endliche Menge von Paaren der Form

$$\sigma = \{\langle X_1, \tau_1 \rangle, \dots, \langle X_n, \tau_n \rangle\}.$$

Dabei gilt:

1. Die X_i sind Typ-Parameter.
2. Die τ_i sind Typen.
3. Für $i \neq j$ ist $X_i \neq X_j$, die Typ-Parameter sind also paarweise verschieden.

Ist $\sigma = \{\langle X_1, \tau_1 \rangle, \dots, \langle X_n, \tau_n \rangle\}$ eine Parameter-Substitution, so schreiben wir

$$\sigma = [X_1 \mapsto \tau_1, \dots, X_n \mapsto \tau_n].$$

Außerdem definieren wir den Domain einer Parameter-Substitution als

$$\text{dom}(\sigma) = \{X_1, \dots, X_n\}.$$

Die Menge aller Parameter-Substitution bezeichnen wir mit SUBST. Im folgenden werden wir der Kürze halber Parameter-Substitutionen einfach als Substitutionen bezeichnen. \square

Substitutionen werden für uns dadurch interessant, dass wir sie auf Typen *anwenden* können. Ist τ ein Typ und ϑ eine Substitution, so ist $\tau\vartheta$ der Typ, der aus τ dadurch entsteht, dass jedes Vorkommen eines Typ-Parameters X_i durch den zugehörigen Typ τ_i ersetzt wird. Die formale Definition folgt.

Definition 48 (Anwendung einer Substitution)

Es sei τ ein Typ und es sei $\vartheta = [X_1 \mapsto \tau_1, \dots, X_n \mapsto \tau_n]$ eine Substitution. Wir definieren die Anwendung von ϑ auf τ (Schreibweise $\tau\vartheta$) durch Induktion über den Aufbau des Typs τ :

1. Falls τ ein Typ-Parameter, gibt es zwei Fälle:

(a) $\tau = X_i$ für ein $i \in \{1, \dots, n\}$. Dann definieren wir

$$X_i\vartheta := \tau_i.$$

(b) $\tau = Y$, wobei Y ein Typ-Parameter ist, so dass $Y \notin \{X_1, \dots, X_n\}$ gilt. Dann definieren wir

$$Y\vartheta := Y.$$

2. Andernfalls muß τ die Form $\tau = f(\sigma_1, \dots, \sigma_m)$ haben. Dann können wir $\tau\vartheta$ durch

$$f(\sigma_1, \dots, \sigma_m)\vartheta := f(\sigma_1\vartheta, \dots, \sigma_m\vartheta).$$

definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $\sigma_i\vartheta$ bereits definiert. \square

Beispiele: Wir definieren eine Substitution ϑ durch

$$\vartheta := [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(\text{int})].$$

Dann gilt:

1. $X_3\vartheta = X_3$,

2. $\text{list}(X_2)\vartheta = \text{list}(\text{list}(\text{int}))$,
3. $\text{map}(X_1, \text{set}(X_2))\vartheta = \text{map}(\text{double}, \text{set}(\text{list}(\text{int})))$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 49 (Komposition von Substitutionen) *Es seien*

$$\vartheta = [X_1 \mapsto \sigma_1, \dots, X_m \mapsto \sigma_m] \quad \text{und} \quad \eta = [Y_1 \mapsto \tau_1, \dots, Y_n \mapsto \tau_n]$$

zwei Substitutionen mit $\text{dom}(\vartheta) \cap \text{dom}(\eta) = \{\}$. Dann definieren wir die Komposition $\vartheta\eta$ von ϑ und η als

$$\vartheta\eta := [X_1 \mapsto \sigma_1\eta, \dots, X_m \mapsto \sigma_m\eta, Y_1 \mapsto \tau_1, \dots, Y_n \mapsto \tau_n]. \quad \square$$

Beispiel: Wir setzen

$$\vartheta := [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(X_3)] \quad \text{und}$$

$$\eta := [X_3 \mapsto \text{map}(\text{int}, \text{double}), X_4 \mapsto \text{char}].$$

Dann gilt:

$$\begin{aligned} \vartheta\eta &= [X_1 \mapsto \text{double}, X_2 \mapsto \text{list}(\text{map}(\text{int}, \text{double})), \\ &\quad X_3 \mapsto \text{map}(\text{int}, \text{double}), X_4 \mapsto \text{char}]. \end{aligned} \quad \square$$

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 50 (Assoziativ-Gesetz für Substitutionen)

Ist τ ein Typ und sind ϑ und η Substitutionen mit $\text{dom}(\vartheta) \cap \text{dom}(\eta) = \{\}$, so gilt

$$(\tau\vartheta)\eta = \tau(\vartheta\eta). \quad \square$$

Der Satz kann durch Induktion über den Aufbau des Typs τ bewiesen werden.

Als nächstes formalisieren wir die Sprechweise, dass ein Term t vom Typ τ ist.

Definition 51 ($t : \tau$) *Für einen Term t und einen Typ τ definieren wir die Relation $t : \tau$ (lese: t ist vom Typ τ) durch Induktion über t .*

1. *Ist c ein 0-stelliges Funktions-Zeichen mit der Signatur $c : \tau$ und ist ϑ eine beliebige Parameter-Substitution, so gilt*

$$c : \tau\vartheta.$$

2. *Es gelte:*

(a) *f sei ein n -stelliges Funktions-Zeichen mit $n > 0$.*

(b) *f habe die Signatur*

$$f : \sigma_1 \times \dots \times \sigma_n \rightarrow \varrho.$$

(c) *ϑ sei eine Substitution so dass die Terme t_i den Typ $\sigma_i\vartheta$ haben.*

Dann hat der Term $f(t_1, \dots, t_n)$ den Typ $\varrho\vartheta$.

Beispiele: Die folgenden Beispiele beziehen sich auf die in Abbildung 17.3 definierten Typen und Funktionszeichen.

1. Es gilt $nil : \text{list}(\text{int})$, denn das Funktionszeichen nil hat die Signatur

$$nil : \text{list}(X).$$

Setzen wir

$$\vartheta = [X \mapsto \text{int}],$$

so folgt $\text{list}(X)\vartheta = \text{list}(\text{int})$ und das zeigt die Behauptung.

2. Als nächstes zeigen wir, dass

$$\text{concat}(nil, nil) : \text{list}(\text{int})$$

gilt. Die Signatur der Funktion concat ist

$$\text{concat} : \text{list}(T) \times \text{list}(T) \rightarrow \text{list}(T).$$

Wieder definieren wir

$$\vartheta = [T \mapsto \text{int}]$$

Wir haben oben schon gesehen, dass

$$nil : \text{list}(\text{int})$$

gilt. Wegen $\text{list}(T)\vartheta = \text{list}(\text{int})$ folgt daraus die Behauptung.

17.3 Ein Algorithmus zur Typ-Überprüfung

Nehmen wir an, es ist ein Term $t = f(t_1, \dots, t_n)$ und ein Typ τ gegeben und es soll geprüft werden, ob der Term t vom Type τ ist. Nehmen wir an, dass das Funktionszeichen f die Signatur

$$f : \sigma_1 \times \dots \times \sigma_n \rightarrow \varrho$$

hat. Nach der obigen Definition von $t : \tau$ hat t genau dann den Typ τ , wenn es eine Parameter-Substitution ϑ gibt, so dass einerseits die Anwendung dieser Substitution auf ϱ den Typ τ liefert ($\varrho\vartheta = \tau$) und andererseits die Teilterme t_i den Typ $\sigma_i\vartheta$ haben. Wir formalisieren dies wie folgt:

$$f(t_1, \dots, t_n) : \tau \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\varrho\vartheta = \tau \wedge \forall i \in \{1, \dots, n\} : t_i : \sigma_i\vartheta)$$

Das Problem, das wir bei der Überprüfung von $t : \tau$ lösen müssen, besteht also darin, eine Parameter-Substitution ϑ zu finden, so dass einerseits

$$\varrho\vartheta = \tau$$

gilt und dass andererseits für alle Teilterme t_i die Beziehung

$$t_i : \sigma_i\vartheta$$

erfüllt ist.

Beispiele: Wir behandeln die bei der Definition des Begriffs $t : \tau$ diskutierten Beispiele jetzt noch einmal im Licht der Formel

$$f(t_1, \dots, t_n) : \tau \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\varrho \vartheta = \tau \wedge \forall i \in \{1, \dots, n\} : t_i : \sigma_i \vartheta)$$

1. Als erstes betrachten wir den Term nil und weisen nach, dass $\text{nil} : \text{list}(\text{int})$ gilt. Dazu fassen wir nil als 0-stelliges Funktionszeichens auf. Da nil einer der *Konstruktoren* des Typs $\text{list}(X)$ ist, hat nil die Signatur

$$\text{nil} : \text{list}(X).$$

Also hat die obige Typ-Bedingung die Form

$$\text{nil} : \text{list}(\text{int}) \Leftrightarrow \exists \vartheta \in \text{SUBST} : (\text{list}(X) \vartheta = \text{list}(\text{int}))$$

Zum Nachweis von $\text{nil} : \text{list}(\text{int})$ benötigen wir also eine Substitution ϑ für die

$$\text{list}(X) \vartheta = \text{list}(\text{int})$$

gilt. Offenbar leistet die Substitution

$$\vartheta = [X \mapsto \text{int}]$$

das Gewünschte und damit haben wir $\text{nil} : \text{list}(\text{int})$ nachgewiesen.

2. Als nächstes betrachten wir den Term $\text{concat}(\text{nil}, \text{nil})$ und weisen nach, dass dieser Term ebenfalls den Typ $\text{list}(\text{int})$ hat. Die Signatur der Funktion concat ist

$$\text{concat} : \text{list}(T) \times \text{list}(T) \rightarrow \text{list}(T).$$

Daher lautet die Typ-Bedingung diesmal

$$\begin{aligned} \text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int}) &\Leftrightarrow \\ \exists \vartheta \in \text{SUBST} : (\text{list}(T) \vartheta &= \text{list}(\text{int}) \wedge \text{nil} : \text{list}(T) \vartheta \wedge \text{nil} : \text{list}(T) \vartheta). \end{aligned}$$

Die Gleichung $\text{list}(T) \vartheta = \text{list}(\text{int})$ wird von der Substitution

$$\vartheta = [T \mapsto \text{int}]$$

gelöst und daher ist die Frage der Korrektheit von

$$\text{concat}(\text{nil}, \text{nil}) : \text{list}(\text{int})$$

nun auf die Frage nach der Korrektheit von

$$\text{nil} : \text{list}(T) \vartheta,$$

also von $\text{nil} : \text{list}(\text{int})$ reduziert. Letzteres haben wir aber unter Punkt 1. schon gezeigt. \square

Die beiden Beispiele zeigen, dass die Frage der Korrektheit einer *Typ-Erklärung* der Form $t : \tau$ auf die Lösung einer Menge von *Typ-Gleichungen* der Form $\varrho \vartheta = \tau$ zurück geführt werden können. Wir formalisieren diese Beobachtung, indem wir eine Funktion

$$\text{typeEqs} : \text{Term} \times \text{Type} \rightarrow \text{set}(\text{Equation})$$

definieren. Die Idee ist, dass für einen Term t und einen Typ τ der Aufruf

$$\text{typeEqs}(t, \tau)$$

eine Menge von Typ-Gleichungen zurückliefert. Diese Gleichungen sind genau dann lösbar, wenn die Typ-Erklärung $t : \tau$ korrekt ist. Die Funktion typeEqs können wir durch die folgende bedingte Gleichung definieren:

$$(f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho) \implies \text{typeEqs}(f(t_1, \dots, t_n), \tau) := \{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i).$$

Diese Gleichung ist wie folgt zu lesen: Falls die Funktion f die Signatur $f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \varrho$ hat, dann ist die Frage der Korrektheit der Typ-Erklärung $f(t_1, \dots, t_n) : \tau$ äquivalent zu der Frage, ob das Typ-Gleichungs-System

$$\{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i)$$

durch eine geeignete *Substitution* ϑ gelöst werden kann. Diesen Begriff definieren wir formal wie folgt.

Definition 52 (Syntaktische Gleichung) *Unter einer syntaktischen Gleichung verstehen wir in diesem Abschnitt ein Konstrukt der Form $\sigma \doteq \tau$, wobei σ und τ Typen sind. Weiter definieren wir ein syntaktisches Gleichungs-System als eine Menge von syntaktischen Gleichungen. Eine Substitution ϑ löst eine syntaktische Gleichung $\sigma \doteq \tau$ genau dann, wenn $\sigma\vartheta = \tau\vartheta$ gilt.* \square

Um die Typ-Überprüfung implementieren zu können, brauchen wir jetzt nur noch ein Verfahren, um Typ-Gleichungs-Systeme lösen zu können. Ein solches Verfahren haben Sie schon im ersten Semester kennengelernt, denn Typ-Gleichungs-Systeme lassen sich durch die selbe *Unifikation* lösen, die Sie schon von der Sprache *Prolog* kennen. Als konkreten Algorithmus zur Unifikation verwenden wir die von Martelli und Montanari [MM82] angegebenen Regeln. Allerdings können wir dieses Verfahren jetzt deshalb noch etwas vereinfachen, weil bei den Typ-Gleichungen der Form $\varrho = \tau$ nur auf der linken Seite, also in ϱ , Typ-Parameter auftauchen, die wir substituieren können. Bei dem Verfahren von Martelli und Montanari arbeiten wir mit Paaren der Form

$$\langle E, \vartheta \rangle.$$

Dabei ist E die Menge der zu lösenden Typ-Gleichungen und ϑ ist die Substitution, die wir berechnen wollen. Wir starten mit dem Paar

$$\langle E, [] \rangle,$$

bei dem die Substitution noch leer ist, während in E die Menge der zu lösenden Gleichungen zusammengefasst sind. Wir formen diese Paare nun nach den folgenden Regeln um:

1. Falls X ein Typ-Parameter ist, so können wir die folgende Reduktion durchführen:

$$\langle E \cup \{X \doteq \tau\}, \vartheta \rangle \rightsquigarrow \langle E[X \mapsto \tau], \vartheta[X \mapsto \tau] \rangle$$

Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $X \doteq \tau$, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution ϑ in die Substitution $\vartheta[X \mapsto \tau]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[X \mapsto \tau]$ angewendet.

2. Ist f ein n -stelliger Typ-Konstruktor, so gilt

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \langle E \cup \{\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n\}, \vartheta \rangle.$$

Eine syntaktische Gleichung der Form $f(\sigma_1, \dots, \sigma_n) \doteq f(\tau_1, \dots, \tau_n)$ wird also ersetzt durch die n syntaktischen Gleichungen $\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n$.

Ein Spezialfall dieser Regel ist

$$\langle E \cup \{c \doteq c\}, \vartheta \rangle \rightsquigarrow \langle E, \vartheta \rangle.$$

Hier steht c für einen 0-stelligen Typ-Konstruktor. Beispielsweise ist `int` ein 0-stelliger Typ-Konstruktor. Diese Gleichung besagt, dass triviale Gleichungen entfernt werden können.

3. Das Gleichungs-System $E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}$ hat keine Lösung, falls die Typ-Konstruktoren f und g verschieden sind, wir schreiben

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega.$$

Hier steht Ω für einen Wert, der das Scheitern des Verfahrens dokumentiert. Falls wir bei der Lösung von $\text{typeEqs}(t, \tau)$ das Ergebnis Ω erhalten, dann trifft die Behauptung $t : \tau$ nicht zu und der Term t ist nicht vom Typ τ .

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$\text{map}(X_2, X_3) \doteq \text{map}(\text{char}, \text{list}(\text{int}))$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{aligned} & \langle \{\text{map}(X_2, X_3) \doteq \text{map}(\text{char}, \text{list}(\text{int}))\}, [] \rangle \\ \rightsquigarrow & \langle \{X_2 \doteq \text{char}, X_3 \doteq \text{list}(\text{int})\}, [] \rangle \\ \rightsquigarrow & \langle \{X_3 \doteq \text{list}(\text{int})\}, [X_2 \mapsto \text{char}] \rangle \\ \rightsquigarrow & \langle \{\}, [X_2 \mapsto \text{char}, X_3 \mapsto \text{list}(\text{int})] \rangle \end{aligned}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[X_2 \mapsto \text{char}, X_3 \mapsto \text{list}(\text{int})]$$

als Lösung der oben gegebenen syntaktischen Gleichung.

17.4 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 17.4 auf Seite 262 präsentiert. Die Abbildungen 17.5, 17.6 und 17.7 auf den folgenden Seiten zeigen die *JavaCup*-Spezifikation eines Parsers für diese Grammatik. Der zugehörige Scanner ist in Abbildungen 17.9 auf Seite 272 gezeigt. Der Scanner unterscheidet in den Zeilen 34 und 35 zwischen Namen, die mit einem großen Buchstaben beginnen und solchen Namen, die mit einem kleinen Buchstaben beginnen. Erstere bezeichnen Typ-Parameter, letztere bezeichnen sowohl Funktionen als auch Typ-Konstruktoren. Die *JavaCup*-Spezifikation des Scanners benutzt die *JavaCup*-Version 0.11a¹, die auch generische Klassen unterstützt. Der von *JavaCup* generierte Parser baut einen abstrakten Syntax-Baum auf. Die zugehörigen Klassen wurden mit Hilfe des Klassen-Generators EP aus der in Abbildung 17.8 gezeigten Spezifikation erzeugt.

Die Klasse `MartelliMontanari` enthält die in Abbildung 17.10 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.

¹Leider ist die Version nicht zu der Version 0.10 kompatibel. Die Inkompatibilität ist zwar für unsere Beispiele bedeutungslos, es gibt aber ein Problem mit den Programmen, die wir in der Rechnertechnik-Vorlesung eingesetzt haben, denn sowohl der IJVM-Assembler als auch der Mikro-Assembler benutzen die *JavaCup*-Version 0.10 und laufen nicht mehr, wenn sie statt dessen die Version 0.11 im CLASSPATH finden.

```

1  import java_cup.runtime.*;
2  import java.util.*;
3
4  terminal          TYPE, SIGNATURE, LEFT_PAR, RIGHT_PAR;
5  terminal          COMMA, COLON, SEMICOLON, ASSIGN, ARROW, PLUS, TIMES;
6  terminal String    FUNCTION, PARAMETER;
7
8  nonterminal Program      program;
9  nonterminal Term        term;
10 nonterminal List<Term>   termList;
11 nonterminal List<Parameter> varList;
12 nonterminal Type        type;
13 nonterminal List<Type>   typeList;
14 nonterminal List<Type>   typeSum;
15 nonterminal TypeDef      typeDef;
16 nonterminal List<TypeDef> typeDefList;
17 nonterminal Signature    signature;
18 nonterminal List<Signature> signatures;
19 nonterminal List<Type>   argTypes;
20 nonterminal TypedTerm    typedTerm;
21 nonterminal List<TypedTerm> typedTerms;
22
23 program      ::= typeDefList:typDefs signatures:signList typedTerms:termList
24               {: RESULT = new Program(typDefs, signList, termList); :}
25               ;
26 typeDefList ::= typeDefList:l typeDef:t {: l.add(t); RESULT = l; :}
27               | typeDef:t
28               {: List<TypeDef> l = new LinkedList<TypeDef>();
29                l.add(t); RESULT = l;
30                :}
31               ;
32 typeDef      ::= TYPE FUNCTION:f ASSIGN typeSum:s SEMICOLON
33               {: RESULT = new SimpleTypeDef(f, s); :}
34               | TYPE FUNCTION:f LEFT_PAR varList:a RIGHT_PAR ASSIGN
35                 typeSum:s SEMICOLON
36               {: RESULT = new ParamTypeDef(f, a, s); :}
37               ;
38 type         ::= FUNCTION:f LEFT_PAR typeList:t RIGHT_PAR
39               {: RESULT = new CompositeType(f, t); :}
40               | FUNCTION:f {: RESULT = new CompositeType(f); :}
41               | PARAMETER:v {: RESULT = new Parameter(v); :}
42               ;
43 typeList     ::= typeList:l COMMA type:t {: l.add(t); RESULT = l; :}
44               | type:t {: List<Type> l = new LinkedList<Type>();
45                          l.add(t);
46                          RESULT = l;
47                          :}
48               ;

```

Abbildung 17.5: *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 17.6: *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 17.7: *JavaCup*-Spezifikation der TTL-Grammatik, 3. Teil

```

1  Program = Program(List<TypeDef>   typedDefs,
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 17.8: Definition der benötigten *Java*-Klassen mit Hilfe von EP.

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-

```

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 17.9: *JFlex*-Spezifikation des Scanners

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

```

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 17.10: Die Methode *solve()* aus der Klasse *MartelliMontanari*.

auf der rechten Seite ein zusammengesetzter Typ stehen. Dies wird in Zeile 16 überprüft.

Der restliche Code beschäftigt sich nun mit dem Fall, dass auf beiden Seiten der syntaktischen Gleichung ein zusammengesetzter Typ steht.

4. Die Zeilen 22 – 25 behandeln den Fall, dass die Typ-Konstruktoren auf beiden Seiten verschieden sind, es wird also der Fall

$$\langle E \cup \{f(\sigma_1, \dots, \sigma_m) \doteq g(\tau_1, \dots, \tau_n)\}, \vartheta \rangle \rightsquigarrow \Omega.$$

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.

```

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 17.11: Berechnung der Typ-Gleichungen

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.
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 hat. Dies wird in Zeile 9 überprüft.
3. Falls bis hierhin keine Probleme aufgetreten sind, erzeugen wir nun die Typ-Gleichungen nach der Formel

$$(f : \sigma_1 \times \dots \times \sigma_n \rightarrow \varrho) \implies \text{typeEqs}(f(t_1, \dots, t_n), \tau) := \{\varrho = \tau\} \cup \bigcup_{i=1}^n \text{typeEqs}(t_i, \sigma_i).$$

Dazu erstellen wir in Zeile 17 zunächst die Typ-Gleichungen

$$\varrho\vartheta \doteq \tau$$

und berechnen dann in der Schleife in den Zeilen 19 – 22 rekursiv die Typ-Gleichungen, die sich aus den Forderungen

$$s_i : \sigma_i \quad \text{für } i = 1, \dots, n$$

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.

Kapitel 18

Entwicklung eines einfachen Compilers

In diesem Kapitel konstruieren wir einen Compiler, der ein Fragment der Sprache **C** in den selben *Java-Byte-Code* übersetzt, den wir schon in der Rechnertechnik-Vorlesung kennengelernt haben. Das von dem Compiler übersetzte Fragment bezeichnen wir als *Integer-C*, denn es steht dort nur der Datentyp `int` zur Verfügung.

Ein Compiler besteht prinzipiell aus den folgenden Komponenten:

1. Der Scanner liest die zu übersetzende Datei ein und zerlegt diese in eine Folge von Token.
Als Scanner werden wir das Werkzeug *JFlex* einsetzen.
2. Der Parser liest die Folge von Token und produziert als Ergebnis einen abstrakten Syntax-Baum.
Wir werden zum Parsen *JavaCup* benutzen.
3. Der Typ-Checker überprüft den abstrakten Syntax-Baum auf Typ-Fehler.
Da die von uns übersetzte Sprache nur einen einzelnen Datentyp enthält, werden wir bei unserem Compiler auf die Implementierung eines Typ-Checkers verzichten, denn das prinzipielle Vorgehen haben wir ja bereits im letzten Kapitel diskutiert und implementiert.
4. Der Code-Generator übersetzt den Parse-Baum in eine Folge von JVM-Assembler-Befehlen.
5. In einer anschließend stattfindenden Optimierungsphase kann optional noch versucht werden, die Effizienz des erzeugten Code zu verbessern. Im Rahmen dieser Vorlesung bleibt uns dafür leider kein Platz mehr.

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 einen 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. Die dabei verwendeten Methoden greifen auf interessante graphentheoretische Algorithmen zurück, die über den Rahmen dieser Vorlesung hinausgehen.

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

$$\begin{aligned}
\text{program} &\rightarrow (\text{function})^+ \\
\text{function} &\rightarrow \text{"int"} \text{ ID } \text{"(" paramList ")} \text{" {" decl }^* (\text{stmtnt ";"})^* \text{"} \\
\text{paramList} &\rightarrow (\text{"int"} \text{ ID } \text{" ," "int"} \text{ ID})^* \text{"} \\
\text{decl} &\rightarrow \text{"int"} \text{ ID } \text{" ;"} \\
\text{stmtnt} &\rightarrow \text{" {" stmtntList "}" } \\
&\quad | \text{ ID " = " expr } \\
&\quad | \text{" if " "(" boolExpr ")" stmtnt } \\
&\quad | \text{" if " "(" boolExpr ")" stmtnt " else " stmtnt } \\
&\quad | \text{" while " "(" boolExpr ")" stmtnt } \\
&\quad | \text{" return " expr } \\
&\quad | \text{ expr } \\
\text{expr} &\rightarrow \text{expr " + " expr} \\
&\quad | \text{expr " - " expr} \\
&\quad | \text{expr " * " expr} \\
&\quad | \text{expr " / " expr} \\
&\quad | \text{" (" expr ")"} \\
&\quad | \text{NUMBER} \\
&\quad | \text{ID} \\
&\quad | \text{ID " (" (expr ("," expr))^* ")" } \\
\text{boolExpr} &\rightarrow \text{expr " == " expr} \\
&\quad | \text{expr " != " expr} \\
&\quad | \text{expr " <= " expr} \\
&\quad | \text{expr " >= " expr} \\
&\quad | \text{expr " < " expr} \\
&\quad | \text{expr " > " expr} \\
&\quad | \text{" ! " boolExpr} \\
&\quad | \text{boolExpr " \&\& " boolExpr} \\
&\quad | \text{boolExpr " || " boolExpr}
\end{aligned}$$

Abbildung 18.1: Eine Grammatik für *Integer-C*

die Grammatik der Quellsprache in erweiterter Backus-Naur-Form (EBNF). Diese Form ist bei der Notation von Listen etwas kürzer, denn wir können die von den regulären Ausdrücken her bekannten Postfix-Operatoren “*”, “+” und “?” verwenden.

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. Ein ID bezeichnet entweder eine Variable oder eine Funktion.
2. NUMBER steht für eine Folge von Ziffern, die als Dezimalzahl interpretiert wird.

Nach der Grammatik ist ein Programm eine Liste von Funktionen. Eine Funktion besteht aus der Deklaration der Signatur, worauf in geschweiften Klammern eine Liste von Deklarationen (*decl*)

und Befehlen (*stmt*) folgt, die voneinander durch Semikolons getrennt werden. Die in Abbildung 18.1 gezeigte Grammatik ist mehrdeutig:

1. Die Grammatik hat das *Dangling-Else-Problem*.

Dieses Problem werden wir später dadurch lösen, dass wir JAVACUP mit der Option

“-expect 1

aufrufen. Diese Option ist analog zu der Option “%expect 1” bei *Bison*: Wir teilen dem Parser mit, dass wir einen Shift/Reduce-Konflikt erwarten und dass wir damit zufrieden sind, wenn dieser Konflikt zugunsten des Shifts aufgelöst wird, denn das ist genau das, was per default passiert.

2. Für die bei arithmetischen und Boole’schen Ausdrücken verwendeten Operatoren müssen Präzedenzen festgelegt werden.

```
1  int sum(int n) {
2      int s;
3      s = 0;
4      while (n != 0) {
5          s = s + n;
6          n = n - 1;
7      };
8      return s;
9  }
10
11 int main() {
12     int n;
13     n = getchar();
14     n = n - 48;
15     putchar(sum(n) + 48);
16     putchar(10);
17 }
```

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 *JAVA-CUP* 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.

Bevor wir die Implementierung des Parsers diskutieren können, müssen wir angeben, durch welche Klassen wir den Syntax-Baum darstellen wollen. Wir erzeugen diese Klassen mit Hilfe von EP aus der in Abbildung 18.4 gezeigten Spezifikation. Die Definition dieser Klassen folgt 1 zu 1 der bereits angegebenen Grammatik.

Damit können wir nun die *Java-CUP*-Datei angeben, mit der wir den Syntaxbaum erzeugen. Wir haben diese Datei aus Platzgründen in drei Teile aufgespalten:

1. Abbildung 18.5 zeigt die Spezifikation der Terminale, Nicht-Terminale und Operator-Präzedenzen. Bei den Präzedenzen ist es wichtig 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.

```

1  import java_cup.runtime.*;
2  %%
3  %char
4  %line
5  %column
6  %cup
7  %{
8      private Symbol symbol(int type) {
9          return new Symbol(type, yychar, yychar + yylength());
10     }
11
12     private Symbol symbol(int type, Object value) {
13         return new Symbol(type, yychar, yychar + yylength(), value);
14     }
15 }
16 %%
17 "+"          { return symbol( sym.PLUS      ); }
18 "-"          { return symbol( sym.MINUS     ); }
19 "*"          { return symbol( sym.TIMES     ); }
20 "/"          { return symbol( sym.SLASH     ); }
21 "("          { return symbol( sym.LPAREN    ); }
22 ")"          { return symbol( sym.RPAREN    ); }
23 "{"          { return symbol( sym.LBRACE    ); }
24 "}"          { return symbol( sym.RBRACE    ); }
25 ","          { return symbol( sym.COMMA     ); }
26 ";"          { return symbol( sym.SEMICOLON ); }
27 "="          { return symbol( sym.ASSIGN    ); }
28 "=="         { return symbol( sym.EQUALS    ); }
29 "!="         { return symbol( sym.NEQUALS   ); }
30 "<"          { return symbol( sym.LT        ); }
31 ">"          { return symbol( sym.LT        ); }
32 "<="         { return symbol( sym.LE        ); }
33 ">="         { return symbol( sym.GE        ); }
34 "&&"         { return symbol( sym.AND       ); }
35 "||"         { return symbol( sym.OR        ); }
36 "!"          { return symbol( sym.NOT       ); }
37 "int"        { return symbol( sym.INT       ); }
38 "return"     { return symbol( sym.RETURN    ); }
39 "if"         { return symbol( sym.IF        ); }
40 "else"       { return symbol( sym.ELSE     ); }
41 "while"      { return symbol( sym.WHILE    ); }
42
43 [a-zA-Z][a-zA-Z_0-9]* { return symbol(sym.IDENTIFIER, yytext()); }
44 [0-9]|[1-9][0-9]*    { return symbol(sym.NUMBER, new Integer(yytext())); }
45 [ \t\n]              { /* skip white space */ }
46 "//" [^\n]*          { /* skip comments   */ }
47
48 [^] { throw new Error("Illegal character '" + yytext() +
49                        "' at line " + yyline + ", column " + yycolumn); }

```

Abbildung 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.

```

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. Wir werden diese Klassen mit Hilfe von EP erzeugen, wobei wir auf für Pseudo-Befehle, wie beispielsweise die “.constant” und “.end-constant” jeweils eine Klasse erzeugen. Die von EP generierten Klassen werden anschließend von Hand angepasst. Abbildung 18.10 zeigt die EP-Definition. Einige Details dieser Klassen erläutern wir jetzt:

```
1  .constant
2  OBJREF 64
3  .end-constant
4
5  .main
6  .var
7      n
8  .end-var
9      in
10     istore n
11     iload n
12     bipush 48
13     isub
14     istore n
15     ldc_w OBJREF
16     iload n
17     invokevirtual sum
18     bipush 48
19     iadd
20     out
21     bipush 1
22     pop
23     bipush 10
24     out
25     bipush 1
26     pop
27     halt
28 .end-main
```

Abbildung 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

```

1  .method sum(n)
2  .var
3      s
4  .end-var
5      bipush 0
6      istore s
7  11:
8      iload n
9      bipush 0
10     isub
11     iflt 13
12     bipush 1
13     goto 14
14  13:
15     bipush 0
16  14:
17     bipush 0
18     if_icmpeq 12
19     iload s
20     iload n
21     iadd
22     istore s
23     iload n
24     bipush 1
25     isub
26     istore n
27     goto 11
28  12:
29     iload s
30     ireturn
31 .end-method

```

Abbildung 18.9: Übersetzung des Programms aus Abbildung 18.2 in Java-Assembler, 2. Teil.

bezeichnen.

Zeile 9 zeigt die Umwandlung eines LABEL-Objektes in einen String, die später zur Ausgabe der Assembler-Kommandos benutzt wird. Der eindeutigen Zahl wird der Buchstabe “1” vor- und der Doppelpunkt “:” nachgestellt, damit die Ausgabe später der Syntax des IJVM-Assemblers entspricht.

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.

```

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 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.

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 Methode

```
public abstract List<AssemblerCmd> compile();
```

```

1  public class LABEL extends AssemblerCmd {
2      private static Integer sLabelCount = 0;
3      private Integer mLabel;
4
5      public LABEL() {
6          mLabel = ++sLabelCount;
7      }
8      public Integer getLabel() { return mLabel; }
9      public String toString() { return "l" + mLabel + ":"; }
10 }

```

Abbildung 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.

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.

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 18.13 gezeigte Form. Die Klasse **Variable** verwaltet eine Member-Variable mit dem Namen **mName**, die den Namen der Variablen angibt. Die Methode *compile()* legt zunächst in Zeile 10 eine neue Liste von Assembler-Kommandos an und erzeugt dann in Zeile 11 das Assembler-Kommando

```
iload mName
```

das als einziges Kommando in diese Liste eingefügt wird. Anschließend kann die Liste als Ergebnis zurück gegeben werden.

Übersetzung einer Konstanten

Bei der Übersetzung einer Konstanten gibt es zwei Fälle:

1. Falls die Konstante c durch ein Byte dargestellt werden kann, wenn also die Ungleichungen

$$-128 \leq c < 128$$

erfüllt sind, dann kann die Konstante mit Hilfe des Befehl

```
bipush c
```

auf den Stack gelegt werden.

```

1  public class Variable extends Expr {
2      private String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd iload = new ILOAD(mName);
10         result.add(iload);
11         return result;
12     }
13     public String getName() { return mName; }
14 }

```

Abbildung 18.13: Die Klasse `Variable`.

2. Andernfalls wird die Konstante zunächst im Konstantenpool unter einem eindeutigen Namen *n* abgelegt und kann dann mit dem Befehl

```
ldc_w n
```

auf den Stack geladen werden.

Ein einfaches Schema zur Namensgenerierung besteht darin, dass wir vor den Dezimalwert der Konstante einfach den Buchstaben “c” setzen und den resultierenden String als Namen verwenden. Die Konstante 234 würde also im Konstanten-Pool unter dem String “c234” abgelegt.

```

1  public class MyNumber extends Expr {
2      private Integer mNumber;
3      static Set<Integer> sConstants = new TreeSet<Integer>();
4
5      public MyNumber(Integer number) {
6          mNumber = number;
7      }
8      public List<AssemblerCmd> compile() {
9          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
10         if (-128 <= mNumber && mNumber < 128) {
11             result.add(new BIPUSH(mNumber));
12         } else {
13             sConstants.add(mNumber);
14             AssemblerCmd ldc = new LDC_W("c" + mNumber);
15             result.add(ldc);
16         }
17         return result;
18     }
19     public Integer getNumber() { return mNumber; }
20 }

```

Abbildung 18.14: Die Klasse `MyNumber`.

Abbildung 18.14 zeigt die Implementierung der Klasse `MyNumber`, die eine Konstante darstellt. Die Konstante selbst wird in der in Zeile 4 deklarierten Member-Variablen `mNumber` gespeichert. Zusätzlich enthält die Klasse aber noch die in Zeile 5 deklarierte statische Variable `sConstants`. Hier handelt es sich um die Menge aller Konstanten, die später in den Konstanten-Pool eingetragen werden müssen.

Die Methode `compile()` überprüft zunächst, ob die Konstante so klein ist, dass sie durch ein Byte dargestellt werden kann. In diesem Fall wird die Auswertung der Konstante in Zeile 13 durch den Assembler-Befehl

```
bipush mNumber
```

übersetzt. Andernfalls wird die Konstante in Zeile 15 dem Konstanten-Pool hinzugefügt und die Auswertung der Konstante wird dann in Zeile 16 durch den Assembler-Befehl

```
ldc_w "c" + mNumber
```

übersetzt.

Übersetzung zusammengesetzter Ausdrücke

Um einen Ausdruck der Form

lhs “+” *rhs*

zu übersetzen, muss zunächst Code erzeugt werden, der die Ausdrücke *lhs* und *rhs* rekursiv übersetzt. Wird dieser Code ausgeführt, so liegen auf dem Stack anschließend die Werte von *lhs* und *rhs*. Durch den Befehl `iadd` werden diese nun addiert. Die Übersetzung kann also wie folgt spezifiziert werden:

$$\text{compile}(\text{lhs} \text{ “+” } \text{rhs}) = \text{lhs.compile}() + \text{rhs.compile}() + [\text{iadd}],$$

wobei der Operator “+” hier die Verkettung von Listen bezeichnet. Abbildung 18.15 zeigt die Umsetzung dieser Überlegung.

```

1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
11         result.addAll(mLhs.compile());
12         result.addAll(mRhs.compile());
13         result.add(new IADD());
14         return result;
15     }
16     public Expr getLhs()      { return mLhs; }
17     public Expr getRhs()      { return mRhs; }
18 }

```

Abbildung 18.15: Die Klasse `Sum`.

Die Übersetzung von Ausdrücken der Form

$$lhs - rhs, \quad lhs * rhs \quad \text{und} \quad lhs / rhs$$

verläuft nach dem selben Schema. Statt des Befehls `iadd` verwenden wir hier die entsprechenden Befehle `isub`, `imul` und `idiv`.¹

Übersetzung von Funktions-Aufrufen

Ein Funktions-Aufruf der $f(e_1, \dots, e_n)$ 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 allerdings vor dem Code, der die einzelnen Ausdrücke e_i auswertet, 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:

$$\text{compile}(\text{getchar}()) = [\text{in}].$$

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:

$$\text{compile}(\text{putchar}(e)) = e.\text{compile}() + [\text{out}].$$

Anschließend legen wir noch mit dem Befehl `bipush` eine 1 auf den Stack. Der Grund dafür ist, dass jeder Funktions-Aufruf ein Ergebnis auf den Stack legen muss, auch dann wenn die Funktion eigentlich gar kein Ergebnis produziert.

Abbildung 18.16 zeigt die Implementierung der Klasse `FunctionCall`, die einen Funktions-Aufruf repräsentiert. Die Klasse hat zwei Member-Variablen.

1. `mName` ist der Name der aufgerufenen Funktion.
2. `mArgs` ist die Liste der Argumente, mit der die Funktion aufgerufen wird.

In der Methode `compile()` wird zunächst überprüft, ob es sich bei dem zu übersetzenden Funktions-Aufruf um einen Aufruf der Funktionen `getchar()` oder `putchar()` handelt und diese Fälle werden dann in der oben beschriebenen Weise abgehandelt. Bei der Übersetzung von `putchar()` ist noch darauf zu achten, dass die Methode ein Ergebnis auf den Stack schreiben muss. Daher wird nach dem `out`-Befehl in Zeile 21 noch ein `bipush`-Befehl angefügt. Andernfalls fügen wir in Zeile 22 an den Anfang der Ergebnisliste einen Befehl, der die Konstante `OBJREF` auf den Stack legt. Anschließend iterieren wir über die Argumente des Funktions-Aufrufs und fügen für jedes Argument den Code an die Liste von Assembler-Befehlen an, die dieses Argument übersetzt. Zum Schluß wird in Zeile 27 noch der Befehl `invokevirtual` an das Ende der Liste angefügt.

¹Die 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 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 18.16: Die Klasse FunctionCall.

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

$$lhs == rhs$$

übersetzen können, müssen wir uns überlegen, was der erzeugte Code überhaupt erreichen soll. Eine naheliegende Forderung ist, dass am Ende auf dem Stack eine 1 abgelegt wird, wenn die Werte der beiden Ausdrücke *lhs* und *rhs* übereinstimmen. Andernfalls soll auf dem Stack eine 0 abgelegt werden. Die Übersetzung kann unter diesen Annahmen wie folgt ablaufen:

```

1  public class Equation extends BoolExpr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Equation(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = mLhs.compile();
11         result.addAll(mRhs.compile());
12         LABEL          trueLabel = new LABEL();
13         LABEL          nextLabel = new LABEL();
14         AssemblerCmd isub      = new ISUB();
15         AssemblerCmd ifeq      = new IFEQ(trueLabel.getLabel());
16         AssemblerCmd bipush0   = new BIPUSH(0);
17         AssemblerCmd gotoNext  = new GOTO(nextLabel.getLabel());
18         AssemblerCmd bipush1   = new BIPUSH(1);
19         result.add(isub);
20         result.add(ifeq);
21         result.add(bipush0);
22         result.add(gotoNext);
23         result.add(trueLabel);
24         result.add(bipush1);
25         result.add(nextLabel);
26         return result;
27     }
28     public Expr    getLhs()    { return mLhs; }
29     public Expr    getRhs()    { return mRhs; }
30 }

```

Abbildung 18.17: Die Klasse `Equation`.

1. Zunächst erzeugen wir in den Zeilen 10 und 11 den Code zur Auswertung von *lhs* und *rhs*. Wenn dieser Code abgearbeitet worden ist, liegen die Werte von *lhs* und *rhs* auf dem Stack.
2. Anschließend subtrahieren wir diese Werte mit dem Befehl `isub`. Wenn die Werte gleich waren, liegt jetzt eine 0 auf dem Stack.
3. Dies können wir mit dem Befehl `ifeq` überprüfen. Liegt eine 0 auf dem Stack, so schreiben wir statt dessen eine 1 auf den Stack, andernfalls schreiben wir eine 0 auf den Stack.

Damit hat der erzeugte Code insgesamt die folgende Form

$$\begin{aligned}
 \text{compile}(\text{lhs} == \text{rhs}) &= \text{lhs.compile}() \\
 &+ \text{rhs.compile}() \\
 &+ [\text{isub}] \\
 &+ [\text{ifeq true}] \\
 &+ [\text{bipush 0}] \\
 &+ [\text{goto next}] \\
 &+ [\text{true:}] \\
 &+ [\text{bipush 1}] \\
 &+ [\text{next:}]
 \end{aligned}$$

Diese Gleichung ist in der Methode *compile()* eins zu eins umgesetzt worden.

Übersetzung von negierten Gleichungen

Die Übersetzung einer negierten Gleichung der Form

$$lhs \neq rhs$$

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 18.18 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.

```

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 18.18: Die Klasse `LessOrEqual`.

Übersetzung von Konjunktionen

Die Übersetzung einer Konjunktion der Form

lhs **&&** *rhs*

kann wie folgt spezifiziert werden:

$$\begin{aligned}
 \text{compile}(\textit{lhs} \ \&\& \ \textit{rhs}) &= \textit{lhs.compile}() \\
 &+ \textit{rhs.compile}() \\
 &+ [\textit{iand}]
 \end{aligned}$$

Abbildung 18.19 zeigt die Umsetzung dieser Gleichung.

Die obige Umsetzung entspricht allerdings nicht dem, was in der Sprache C tatsächlich passiert. Dort wird die Auswertung eines Ausdrucks der Form

lhs **&&** *rhs*

abgebrochen, sobald das Ergebnis der Auswertung feststeht. Liefert die Auswertung von *lhs* als

```

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 18.19: Die Klasse `Conjunction`.

Ergebnis eine 0, so wird der Ausdruck *rhs* nicht mehr auswertet. 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.

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 18.20 zeigt die Umsetzung dieser Idee.

```

1  public class Negation extends BoolExpr {
2      private BoolExpr mExpr;
3
4      public Negation(BoolExpr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          AssemblerCmd bipush1 = new BIPUSH(1);
10         AssemblerCmd isub    = new ISUB();
11         result.add(bipush1);
12         result.addAll(mExpr.compile());
13         result.add(isub);
14         return result;
15     }
16     public BoolExpr getExpr() { return mExpr; }
17 }

```

Abbildung 18.20: Die Klasse `Negation`.

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:

$$\begin{aligned} \text{compile}(x=\text{expr}) &= \text{expr.compile()} \\ &+ [\text{istore } x] \end{aligned}$$

Die Idee wird in der in Abbildung 18.21 gezeigten Klasse `Assign` umgesetzt.

Übersetzung von Ausdrücken als Befehlen

Die Übersetzung eines Ausdrucks, der als Befehl verwendet wird, birgt eine Tücke: Die Übersetzung des Ausdrucks selber hinterläßt auf dem Stack einen Wert. Dieser muss aber bei Beendigung des Befehls vom Stack entfernt werden! Daher müssen wir den Befehl `pop` an das Ende der Liste der Assembler-Befehle anfügen, die bei der Übersetzung des Ausdrucks erzeugt werden. Die Übersetzung eines Befehls vom Typ `ExprStatement` wird also wie folgt spezifiziert:

$$\begin{aligned} \text{compile}(\text{expr};) &= \text{expr.compile()} \\ &+ [\text{pop}] \end{aligned}$$

Abbildung 18.22 zeigt die Klasse `ExprStatement`, in der diese Überlegung umgesetzt wird.

```

1  public class Assign extends Statement {
2      private String mVar;
3      private Expr  mExpr;
4
5      public Assign(String var, Expr expr) {
6          mVar = var;
7          mExpr = expr;
8      }
9      public List<AssemblerCmd> compile() {
10         List<AssemblerCmd> result = mExpr.compile();
11         AssemblerCmd      storeCmd = new ISTORE(mVar);
12         result.add(storeCmd);
13         return result;
14     }
15     public String getVar() { return mVar; }
16     public Expr  getExpr() { return mExpr; }
17 }

```

Abbildung 18.21: Die Klasse Assign.

```

1  public class ExprStatement extends Statement {
2      private Expr mExpr;
3
4      public ExprStatement(Expr expr) {
5          mExpr = expr;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = mExpr.compile();
9          AssemblerCmd      popCmd = new POP();
10         result.add(popCmd);
11         return result;
12     }
13     public Expr  getExpr() { return mExpr; }
14 }

```

Abbildung 18.22: Die Klasse ExprStatement.

Die Übersetzung von Verzweigungs-Befehlen

Als nächstes überlegen wir, wie ein Verzweigungs-Befehl der Form

if (expr) statement

übersetzt werden kann. Offenbar muss zunächst der Boole'sche Ausdruck *expr* übersetzt werden. Die Auswertung dieses Ausdrucks wird auf dem Stack entweder eine 1 oder eine 0 hinterlassen, je nachdem, ob die Bedingung des Tests wahr oder falsch wahr. Mit dem Befehl *ifeq* können wir überprüfen, welcher dieser beiden Fälle vorliegt. Das führt zu der folgenden Spezifikation:

$$\begin{aligned}
 \text{compile}(\text{if (expr) statement}) &= \text{expr.compile()} \\
 &+ [\text{ifeq else}] \\
 &+ \text{statement.compile()} \\
 &+ [\text{else:}]
 \end{aligned}$$

Diese Spezifikation ist in der Abbildung 18.23 umgesetzt worden.

```
1  public class IfThen extends Statement {
2      private BoolExpr  mBoolExpr;
3      private Statement mStatement;
4
5      public IfThen(BoolExpr boolExpr, Statement statement) {
6          mBoolExpr = boolExpr;
7          mStatement = statement;
8      }
9
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 18.23: Die Klasse IfThen.java

Die Übersetzung eines Verzweigungs-Befehls der Form

if (condition) thenStmnt else elseStmnt

erfolgt in analoger Art und Weise. Diesmal lautet die Spezifikation:

$$\begin{aligned} \text{compile}(\text{if } (condition) \text{ thenStmnt else elseStmnt}) &= \text{cond.compile()} \\ &+ [\text{ifeq else}] \\ &+ \text{thenStmnt.compile()} \\ &+ [\text{goto next}] \\ &+ [\text{else:}] \\ &+ \text{elseStmnt.compile()} \\ &+ [\text{next:}] \end{aligned}$$

Diese Spezifikation ist in der Abbildung 18.24 umgesetzt worden.

Die Übersetzung einer Schleife

Die Übersetzung einer while-Schleife der Form

while (cond) statement

```

1  public class IfThenElse extends Statement {
2      private BoolExpr mCondition;
3      private Statement mThen;
4      private Statement mElse;
5
6      public IfThenElse(BoolExpr condition, Statement thenStmt, Statement elseStmt) {
7          mCondition = condition;
8          mThen      = thenStmt;
9          mElse      = elseStmt;
10     }
11     public List<AssemblerCmd> compile() {
12         List<AssemblerCmd> result = mCondition.compile();
13         LABEL      elseLabel = new LABEL();
14         LABEL      nextLabel = new LABEL();
15         AssemblerCmd ifeq     = new IFEQ(elseLabel.getLabel());
16         AssemblerCmd gotoNext = new GOTO(nextLabel.getLabel());
17         result.add(ifeq);
18         result.addAll(mThen.compile());
19         result.add(gotoNext);
20         result.add(elseLabel);
21         result.addAll(mElse.compile());
22         result.add(nextLabel);
23         return result;
24     }
25     public BoolExpr getCondition() {
26         return mCondition;
27     }
28     public Statement getThen() {
29         return mThen;
30     }
31     public Statement getElse() {
32         return mElse;
33     }
34 }

```

Abbildung 18.24: Die Klasse IfThenElse.

orientiert sich an der folgenden Spezifikation:

$$\begin{aligned}
 \text{compile}(\text{while } (cond) \text{ stmt}) &= [\text{loop: }] \\
 &+ \text{cond.compile}() \\
 &+ [\text{ifeq next}] \\
 &+ \text{stmt.compile}() \\
 &+ [\text{goto loop}] \\
 &+ [\text{next: }]
 \end{aligned}$$

Die Umsetzung dieser Spezifikation sehen Sie in Abbildung 18.25.

Übersetzen einer Liste von Befehlen

Eine in geschweiften Klammern eingeschlossene Liste von Befehlen der Form

$$\{ \text{stmt}_1; \dots \text{stmt}_n; \}$$

```

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 18.25: Die Klasse `While`.

wird dadurch übersetzt, dass die Listen, die bei der Übersetzung der einzelnen Befehle *stmt_n*_i entstehen, aneinander gehängt werden:

$$\text{compile}(\{ \text{stmt}_1; \dots \text{stmt}_n; \}) = \text{compile}(\text{stmt}_1) + \dots + \text{compile}(\text{stmt}_n).$$

Diese Idee ist in der Klasse `Block` realisiert worden. Abbildung 18.26 zeigt diese Klasse.

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 die vollständiges Programm.

Wir beginnen mit der Diskussion der Klasse `Function`. Abbildung 18.27 zeigt die Klasse `Function`, allerdings ohne die Implementierung der Methode *compile()*, die wir aus Platzgründen in die Abbildung 18.28 ausgelagert haben.

```

1  public class Block extends Statement {
2      private List<Statement> mStatementList;
3
4      public Block(List<Statement> statementList) {
5          mStatementList = statementList;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
9          for (Statement stmtnt: mStatementList) {
10             result.addAll(stmtnt.compile());
11          }
12          return result;
13      }
14      public List<Statement> getStatementList() {
15          return mStatementList;
16      }
17  }

```

Abbildung 18.26: Die Klasse Block.

```

1  public class Function {
2      private String          mName;
3      private List<String>     mParameterList;
4      private List<Declaration> mDeclarations;
5      private List<Statement>  mBody;
6
7      public Function(String      name,
8                          List<String> parameterList,
9                          List<Declaration> declarations,
10                         List<Statement> body)
11      {
12          mName          = name;
13          mParameterList = parameterList;
14          mDeclarations  = declarations;
15          mBody          = body;
16      }
17      public List<AssemblerCmd> compile() { ... }
18      public String          getName()          { return mName;          }
19      public List<String>     getParameterList() { return mParameterList; }
20      public List<Statement> getBody()           { return mBody;         }
21  }

```

Abbildung 18.27: Die Klasse Function.

Die Klasse `Function` enthält vier Member-Variablen:

1. `mName` gibt den Namen der Funktion an.
2. `mParameterList` ist die Liste der Parameter, mit der die Funktion aufgerufen wird.
3. `mDeclarations` ist die Liste der Variablen-Deklarationen.
4. `mBody` ist die Liste von Befehlen, die im Rumpf der Funktion ausgeführt werden.

```
1  public List<AssemblerCmd> compile() {
2      List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
3      AssemblerCmd nl = new NEWLINE();
4      result.add(nl);
5      if (mName.equals("main")) {
6          AssemblerCmd main = new MAIN();
7          AssemblerCmd var  = new VAR();
8          result.add(main);
9          result.add(var);
10         for (Declaration decl: mDeclarations) {
11             AssemblerCmd varDef = new VAR_DEF(decl.getVar());
12             result.add(varDef);
13         }
14         AssemblerCmd endVar = new END_VAR();
15         result.add(endVar);
16         for (Statement 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 18.28: Die Methode `compile()`.

Die eigentliche Arbeit der Klasse **Funktion** wird in der Methode *compile()*, die in Abbildung 18.28 gezeigt ist, geleistet. Es sind zwei Fälle zu unterscheiden:

1. Falls die zu übersetzende Funktion den Namen “main” hat, so hat der erzeugte Code die folgende Form:

```

1      .main
2      .var
3           $v_1$ 
4           $\vdots$ 
5           $v_k$ 
6      .end-var
7           $s_1$ 
8           $\vdots$ 
9           $s_n$ 
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      .method mName( $p_1, \dots, p_l$ )
2      .var
3           $v_1$ 
4           $\vdots$ 
5           $v_k$ 
6      .end-var
7           $s_1$ 
8           $\vdots$ 
9           $s_n$ 
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.

Zum Abschluss diskutieren wir die Klasse **Program**, die in Abbildung 18.29 gezeigt wird. Diese Klasse verwaltet in der Member-Variablen **mFunctionList**, die Liste aller zu übersetzenden Funktionen. Neben dem Code für die einzelnen Funktionen hat diese Klasse die Aufgabe, die Deklaration der verwendeten Konstanten an den Anfang der erzeugten Assembler-Datei zu schreiben. Damit diese möglich ist, müssen aber erst alle Funktionen übersetzt werden, denn bevor die Funktionen nicht alle übersetzt worden sind, ist ja noch gar nicht klar, welche Konstanten überhaupt verwendet worden sind.

Bei der Übersetzung der Funktionen ist darauf zu achten, dass zuerst die Funktion *main()* übersetzt wird, denn diese muss am Anfang der erzeugten Assembler-Datei stehen. In der C-Datei ist die Funktion *main()* aber die letzte Funktion, denn in der Sprache C müssen alle Funktionen vor ihrer Verwendung deklariert worden sein.

```

1  public class Program {
2      private List<Function> mFunctionList;
3
4      public Program(List<Function> functionList) {
5          mFunctionList = functionList;
6      }
7      public List<AssemblerCmd> compile() {
8          List<AssemblerCmd> fctList = new LinkedList<AssemblerCmd>();
9          int indexMain = mFunctionList.size() - 1;
10         Function main = mFunctionList.get(indexMain);
11         fctList.addAll(main.compile());
12         for (int i = 0; i < indexMain; ++i) {
13             Function f = mFunctionList.get(i);
14             fctList.addAll(f.compile());
15         }
16         AssemblerCmd constant = new CONSTANT();
17         AssemblerCmd endConstant = new END_CONSTANT();
18         List<AssemblerCmd> result = new LinkedList<AssemblerCmd>();
19         result.add(constant);
20         result.add(new CONSTANT_DEF("OBJREF", 64));
21         for (Integer c: MyNumber.sConstants) {
22             result.add(new CONSTANT_DEF("c" + c, c));
23         }
24         result.add(endConstant);
25         result.addAll(fctList);
26         return result;
27     }
28     public List<Function> getFunctionList() {
29         return mFunctionList;
30     }
31 }

```

Abbildung 18.29: Die Klasse `Program`.

Wie übersetzen in Zeile 11 als erstes die Funktion `main()`. Anschließend werden in der Schleife, die sich von Zeile 12 bis 15 erstreckt, die restlichen Funktionen übersetzt. Der erzeugte Code befindet sich dann in der Liste `fctList`. Nach dieser Übersetzung sind dann alle im Programm verwendeten Konstanten bekannt. Wir erinnern uns, dass diese in der statischen Variablen `sConstants` in der Klasse `MyNumber` abgespeichert worden sind. In der Schleife in den Zeilen 21 – 23 werden diese nun in Pseudo-Assembler-Befehle übersetzt, die in die Liste `result` geschrieben werden. An diese Liste wird dann das Ergebnis der Übersetzung der Funktionen angehängt.

Übersetzen wir die in Abbildung 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.
- [Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.
- [Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.
- [DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.
- [DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.
- [Ear68] Jay Clark Earley. *An efficient context-free parsing algorithm*. PhD thesis, Pittsburgh, PA, USA, 1968.
- [Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
- [Ers58] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.
- [Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O’Reilly & Associates, 2nd edition, 2002.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The 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.
- [Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, pages 3–40. Princeton University Press, 1956.
- [Kle05] Gerwin Klein. JFlex User’s Manual. Technical report, 2005. Available at: <http://jflex.de/jflex.pdf>.
- [Knu65] Donald E. Knuth. On the translation of languages from left to 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] A. Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.
- [Par07] Terence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.
- [PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.
- [Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.
- [Str07] Karl Stroetmann. *Computer-Architektur: Modellierung, Entwicklung und Verifikation mit Verilog*. Oldenbourg-Verlag, 2007.
- [Tan05] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Upper Saddle River, NJ, 5th edition, 2005.