Formale Sprachen und ihre Anwendungen

Karl Stroetmann

27. Oktober 2007

Inhaltsverzeichnis

1 Einführung **3**
- 1.1 Grundlegende Definitionen . . . 3
- 1.2 Überblick über die Vorlesung . . . 5
- 1.3 Literatur . . . 6

2 Reguläre Ausdrücke **8**
- 2.1 Einige Definitionen . . . 8
- 2.2 Algebraische Vereinfachung regulärer Ausdrücke . . . 11

3 Die Werkzeuge *Flex* und *JavaCC* **13**
- 3.1 Das Werkzeug *Flex* . . . 13
  - 3.1.1 Ein einfaches Beispiel . . . 14
  - 3.1.2 Übersetzung des Beispiels . . . 16
  - 3.1.3 Fallstricke . . . 16
  - 3.1.4 Reguläre Ausdrücke in *Flex* . . . 16
  - 3.1.5 Ein Beispiel . . . 19
  - 3.1.6 Start-Zustände . . . 22
  - 3.1.7 Zusammenfassung . . . 24
- 3.2 *JavaCC* . . . 25
  - 3.2.1 Ein Beispiel . . . 25
  - 3.2.2 Reguläre Ausdrücke in *JavaCC* . . . 29
  - 3.2.3 Zustände in *JavaCC* . . . 30

4 Endliche Automaten **33**
- 4.1 Deterministische endliche Automaten . . . 33
- 4.2 Nicht-deterministische endliche Automaten . . . 37
- 4.3 Äquivalenz von EA und NEA . . . 40
- 4.4 Übersetzung regulärer Ausdrücke in NEA . . . 44
- 4.5 Übersetzung eines EA in einen regulären Ausdruck . . . 48
- 4.6 Minimierung endlicher Automaten . . . 52

5 Reguläre Sprachen **57**
- 5.1 Abschluss-Eigenschaften regulärer Sprachen . . . 57
- 5.2 $L(A) = \{\}$? . . . 59
- 5.3 Äquivalenz regulärer Ausdrücke . . . 60
- 5.4 Grenzen regulärer Sprachen . . . 60

6 Kontextfreie Sprachen **64**
- 6.1 Kontextfreie Grammatiken . . . 64
  - 6.1.1 Ableitungen . . . 67
  - 6.1.2 Parse-Bäume . . . 69
  - 6.1.3 Mehrdeutige Grammatiken . . . 70

6.2 Top-Down-Parser 71
6.2.1 Umschreiben der Grammatik 72
6.2.2 Implementierung eines Top-Down-Parser 74
6.2.3 Implementierung eines rückwärts arbeitenden Top-Down-Parser 79

7 Parser-Generatoren **82**
7.1 *JavaCC* 82
7.1.1 Ein einfaches Beispiel 82
7.1.2 Implementierung eines Interpreters 92
7.2 *Bison* 92
7.2.1 Der Scanner 92
7.2.2 Der Parser 96

Kapitel 1

Einführung

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 von Programmier-Sprachen und die Entwicklung der dazugehörigen Compiler.

   Die Theorie der formalen Sprachen bildet die theoretische Grundlage für die Entwicklung von Compilern.

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 erkennem, das 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*:

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

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

Beispiele:

1. $\Sigma = \{0, 1\}$ ist ein Alphabet, mit dem wir beispielsweise natürliche Zahlen im Binärsystem darstellen können.

2. $\Sigma = \{\mathtt{a}, \cdots, \mathtt{z}, \mathtt{A}, \cdots, \mathtt{Z}\}$ ist das Alphabet, das der englischen Sprache zu Grunde liegt.

3. Die Menge $\Sigma_{\text{ASCII}} = \{0, 1, \cdots, 127\}$ wird als das ASCII-Alphabet bezeichnet. Hierbei werden die Zahlen als Buchstaben, Ziffern, Sonderzeichen, und Kontrollzeichen interpretiert. Beispielsweise repräsentieren die Zahlen von 65 bis 91 die Buchstaben '`A`' bis '`Z`'.

   In Anwendungen wird oft auch das Alphabet $\{0, 1, \cdots, 255\}$ betrachtet, wobei die ersten 128 Zeichen nach dem ASCII-Alphabet interpretiert werden.

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

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

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

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

Beispiele:

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

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

   so sind w_1 und w_2 Worte, es gilt also

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

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

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

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

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

$$\mathit{append} : \Sigma^* \times \Sigma^* \to \Sigma^*$$

entsteht. Diese Funktion wird durch Induktion nach der Länge des ersten Arguments definiert.

1. $\mathit{append}(\varepsilon, w) := w$,
2. $\mathit{append}([c] + v, w) := [c] + \mathit{append}(v, w)$ falls $c \in \Sigma$.

Die Konkatenation von w_1 und w_2 wird der Kürze halber als $w_1 w_2$ geschrieben:

$$w_1 w_2 := \mathit{append}(w_1, w_2).$$

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

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

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

Beispiele:

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

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

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

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

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

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

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

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

   (b) $\mathit{value}(w\mathtt{0}) = 2 \cdot \mathit{value}(w)$,

   (c) $\mathit{value}(w\mathtt{1}) = 2 \cdot \mathit{value}(w) + 1$.

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

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

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

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

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

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

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

4. Es sei $\Sigma := \Sigma_{\text{ASCII}} \cup \{\dagger\}$ das Alphabet, das aus dem ASCII-Alphabet dadurch hervorgeht, dass das wir das zusätzliche Zeichen † hinzufügen. Die sogenannte universelle Sprache L_u bestehe aus allen Wörten 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 einfache und praktisch leicht handhabbare Möglichkeit um formale Sprachen zu definieren. Die meisten modernen Programmiersprachen und insbesondere alle modernen Skriptsprachen (*Tcl*, *Perl*, *Python*, *Ruby*, $\cdots$) 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 Ausdrücken sind nicht ausreichend, um Programme beschreiben zu können. Kontextfreie Sprachen sind ein Konzept zur Beschreibung formaler Sprachen, mit dem sich einerseits die Konstrukte heutiger Programmiersprachen gut beschreiben lassen und das andererseits immer noch einfach genug ist, um effizient implementiert werden zu können.

5. Parser-Generatoren

   In diesem Kapitel stellen wir die Parser-Generatoren *Bison* und *Antlr* vor. Wir zeigen exemplarisch, wie mit ANTLR ein Interpreter für eine einfache Programmiersprache entwickelt werden kann.

6. Keller-Automaten

   Keller-Automaten dienen der Implementierung kontextfreier Sprachen.

7. Turing-Maschinen

   Das Konzept der Turing-Maschinen ist das allgemeinste denkbare Maschinen-Konzept, dass sich noch mit Hardware realisieren läßt. Jede Funktion, die überhaupt irgendwie berechnet werden kann, ist mit Hilfe einer Turing-Maschine berechenbar.

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. *The Definitive ANTLR Reference* [Par07]

   Hier wird der Parser- und Scanner-Generator ANTLR beschrieben.

Außerdem möchte ich noch auf das Buch "*Mastering Regular Expressions*" von Friedl [Fri02] hinweisen, in dem die Anwendung regulärer Ausdrücke in Skript-Sprachen ausführlich 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.

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:

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

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

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

definiert. Dannn gilt

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

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

1. $n = 0$:
   $L^0 := \{\varepsilon\}$.
2. $n \mapsto n + 1$:
   $L^{n+1} = L^n \cdot L$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   Außerdem nehmen wir an, dass das Zeichen ε nicht in dem Alphabet Σ auftritt, es gilt also $\varepsilon \notin \Sigma$.

3. $c \in \Sigma \to c \in \mathtt{RegExp}_\Sigma$.

   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. $r_1 \in \mathtt{RegExp}_\Sigma \wedge r_2 \in \mathtt{RegExp}_\Sigma \to r_1 + r_2 \in \mathtt{RegExp}_\Sigma$

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 $+ \notin \Sigma$ an.

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

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

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

   Wir nehmen $\cdot \notin \Sigma$ an.

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

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

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

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

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

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

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

   Wir nehmen $(\notin \Sigma$ und $) \notin \Sigma$ an. □

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

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

Damit wird also der reguläre Ausdruck

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

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

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

festgelegt.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9. $r + r \doteq r$, denn

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

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

    Wir haben

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

    und daraus folgt allgemein $L(r) \subseteq L(r^*)$. Ersetzen wir in dieser Beziehung r durch r^*, so sehen wir, dass

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

    gilt. Um die Umkehrung

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

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

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

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

$$w = u_1 \cdots u_n.$$

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

$$u_i = v_{1,i} \cdots v_{m(i),i}$$

gilt. Insgesamt gilt dann

$$w = v_{1,1} \cdots v_{m(1),1} v_{1,2} \cdots v_{m(2),2} \cdots v_{1,n} \cdots v_{m(n),n}.$$

Also ist w eine Konkatenation von Wörtern der Sprache $L(r)$ und das heißt

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

und damit ist die Inklusion $L\big((r^*)^*\big) \subseteq L(r^*)$ gezeigt.

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

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

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

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

Kapitel 3

Die Werkzeuge *Flex* und *JavaCC*

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 verschiedene Scanner-Generatoren. Wir werden uns zunächst das Werkzeug *JavaCC* genauer ansehen. Der Name *JavaCC* steht für *Java Compiler Compiler*, es handelt sich also um ein Werkzeug, das aus der Spezifikation eines Compilers einen Compiler erzeugen kann. Dieses Werkzeug läßt sich damit auch als Scanner-Generator einsetzen.

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

   regexp {*cmds*}

   Hier steht *regexp* für einen regulären Ausdruck. Dieser muss am Zeilenanfang stehen. Jedesmal, 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, das 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 Buchstaben 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.l`" und rufen anschließend *Flex* mit dem Befehl

```
flex change.l
```

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:

$$\mathit{OpSyms} := \{\ \text{“.”}, \text{“*”}, \text{“+”}, \text{“?”}, \text{“|”}, \text{“(”}, \text{“)”}, \text{“[”}, \text{“]”}, \text{“\{”}, \text{“\}”}, \text{“<”}, \text{“>”}, \text{“/”}, \text{“\textbackslash”}, \text{“\^{}”}, \text{“\$”}, \text{“"”}\ \}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

   Bei der Syntax $\backslash abc$ sind a, b und c oktale Ziffern und abc muss als Zahl im Oktal-System interpretierbar sein. Dann wird durch $\backslash abc$ das Zeichen spezifiziert, das im ASCII-Code an der durch die Oktalzahl abc spezifizierten Stelle steht.

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

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

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

   Die Konkatenation zweier regulärer Ausdrücke wird in *Flex* ohne den Infix-Operator "$\cdot$" geschrieben.

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

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

7. $r\texttt{*} \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

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

8. $r\texttt{+} \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

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

   $$r\texttt{+} \simeq rr*.$$

9. $r\texttt{?} \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

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

   $$r\texttt{?} \simeq r|\varepsilon.$$

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

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

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

11. `^`r falls $r \in \mathit{Regexp}$

    Der Ausdruck `^`r legt fest, dass der reguläre Ausdruck r am Anfang einer Zeile stehen muss.

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

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

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

    Der Ausdruck r_1`/`r_2 legt fest, dass auf den durch r_1 spezifizierten Text ein Text folgen muss, der der Spezifikation r_2 genügt. Im Unterschied zur einfachen Konkatenation von r_1 und r_2 wird durch den 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. $(r) \in \mathit{Regexp}$ falls $r \in \mathit{Regexp}$

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

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]` $\simeq$ `a|e|i|o|u`

Die Menge aller kleinen lateinischen Buchstaben läßt sich durch

```
[a-z]
```

spezifizieren, es gilt also

`[a-z]` $\simeq$ `a|b|c|`$\cdots$`|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  Daniel 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

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

die Note errechnet, wobei die Variable `maxPoints` die maximale erreichbare Punktzahl angibt. 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 maximal möglichen Punkte.

(b) In der Variablen `lineNumber` speichern wir die Zeilennummer. Dies ist nützlich um später aussagekräftige Fehlermeldungen geben zu können.

(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 an-

stelle des regulären Ausdrucks

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

kürzer "{ZAHL}" schreiben. Beachten Sie, dass der Name einer 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 Dopplepunkt 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 ließt 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 ließt die Leerzeichen und Tabulatoren ein, die auf den Doppelpunkt folgen und gibt diese aus. Dadurch erreichen wir, dass die Ausgabe der Noten genauso formatiert wird wie die Eingabe-Datei.

(d) Die Regel in Zeile 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 Punktte 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:

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

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

1. `\/\*`

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

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

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

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

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

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

3. `\*+\/`

   Dieser reguläre Ausdruck spezifiziert das Ende des Kommentars. Es kann aus einer beliebigen positiven Anzahl von "`*`"-Zeichen bestehen, auf die dann noch ein "`/`" folgt. Wenn wir hier

nur den Ausdruck "`\*\/`" verwenden würden, dann könnten wir Kommentare der Form

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

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

Reguläre Ausdrücke, wie der in (3.1) gezeigte Ausdruck zur Erkennung mehrzeiliger Kommentare, sind für die Praxis zu kompliziert. Daher gibt es in *Flex* sogenannte *Start-Zustände* (engl. *start conditions*), mit deren Hilfe *Flex*-Spezifikationen verständlicher strukturiert 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 3.4 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.

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

Abbildung 3.4: *Flex*-Spezifikation zur Transformation von HTML in Text.

Wir diskutieren jetzt die *Flex*-Spezifikation aus Abbildung 3.4 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*`" 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 den der Scanner startet, in den oben deklarierten Zustand `header`. Da dieser Zustand als *exklusiver* Zustand deklariert worden ist, können jetzt nur noch solche Regeln angewendet werden, die mit dem Prefix "`<header>`" beginnen. Wäre der Zustand als *inklusiver* Zustand deklariert worden, so könnten auch solche Regeln verwendet werden, die nicht mit einem Zustand markiert sind. Solche Regeln sind implizit mit dem Zustand "`<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: Implementieren Sie eine *Flex*-Spezifikation, die aus einem `C`-Programm alle Kommentare entfernt.

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 *JavaCC*

Die Sprache `C` ist in vielen Bereichen von der Sprache *Java* verdrängt worden. Daher ist es nur natürlich, dass auch für *Java* Scanner-Generatoren entwickelt worden sind. Hier gibt es mittlerweile eine ganze Menge verschiedener Werkzeuge. Von diesen hat das Werkzeuge *JavaCC* [Kod04] einen offiziellen Character, weil es wie *Java* von der Firma *Sun Microsystems* entwickelt worden ist und *JavaCC* ein registriertes Warenzeichen der Firma *Sun* ist. Daher habe ich von den verschiedenen zur Auswahl stehenden Werkzeugen zunächst *JavaCC* ausgewählt.

Eigentlich ist *JavaCC* mehr als nur ein Scanner-Generator, denn mit *JavaCC* können Sie auch einen Parser erzeugen. Diesen Aspekt werden wir in einem späteren Kapitel noch besprechen. Zunächst begnügen wir uns aber damit, *JavaCC* als Scanner-Generator einzusetzen.

3.2.1 Ein Beispiel

Wir demonstrieren die Funktionalität von *JavaCC* anhand des Beispiels der Notenberechnung, wir setzen das Beispiel aus Abbildung 3.3 also nun noch einmal mit *JavaCC* um. Abbildung 3.5 zeigt die Datei `Klausur.jj`, die eine Eingabe-Spezifikation für das Werkzeug *JavaCC* ist.

Eine *JavaCC*-Eingabe-Spezifikation für einen Scanner besteht aus zwei Teilen:

1. Die *Klassen-Definition* wird durch die Schlüsselwörter "`PARSER_BEGIN`" und "`PARSER_END`" eingeschlossen. In der Abbildung erstreckt sich dieser Teil von Zeile 1 – 21.

2. Die *Token-Definition* definiert verschiedene Tokens durch reguläre Ausdrücke. In unserem Beispiel beginnt sie mit dem Schlüsselwort "`TOKEN`" in Zeile 23, dem eine Doppelpunkt und eine öffnende geschweifte Klammer "{" folgt. Sie endet in Zeile 45 mit der schließenden geschweiften Klammer "}".

Wir diskutieren die Eingabe-Spezifikation jetzt Zeile für Zeile.

1. In Zeile 1 folgt hinter dem Schlüsselwort "`PARSER_BEGIN`" der in Klammern eingeschlossene Name der Datei, die die Spezifikation enthält, wobei die Datei-Endung weggelassen wird. Die Datei-Endung lautet immer "`.jj`".

2. Anschließend folgt eine Klassen-Definition. Der Name dieser Klasse muss ebenfalls mit dem Datei-Namen übereinstimmen.

   In unserem Fall enthält die Klasse zunächst die Definitionen verschiedener statischer Variablen.

   (a) Die in Zeile 4 definierte Variable `sName` enthält später den Namen des Studenten, dessen Note berechnet werden soll. Wir initialisieren diese Variable aber zunächst mit `null`. Dadurch haben wir später die Möglichkeit zu prüfen, ob bereits Punkte eines Studenten gelesen worden sind oder ob bisher erst die Kopfzeilen der Eingabe-Datei gelesen wurden.

   (b) Die in Zeile 5 definierte Variable `sSumPoints` gibt die Summe aller Punkte an, die ein Student erzielt hat.

   (c) Die in Zeile 6 definierte Variable `sMaxPoints` gibt die maximal erreichbare Punktezahl an.

   Anschließend wird die Methode `main` definiert. Diese Methode ist erforderlich, damit der erzeugte Scanner aufgerufen werden kann.

   (a) Als erstes erzeugen wir einen *Token-Manager*. Da der Konstruktor für einen Token-Manager als Eingabe ein Objekt vom Type `SimpleCharStream` erwartet, konvertieren wir den Eingabestrom "`System.in`" zunächst in ein solches Objekt und erzeugen dann in Zeile 10 den Token-Manager, als Objekt der Klasse `KlausurTokenManager`.

```
1  PARSER_BEGIN(Klausur)
2
3  public class Klausur {
4      static String  sName      = null;
5      static Integer sSumPoints = 0;
6      static Integer sMaxPoints = 0;
7
8      public static void main(String args[]) throws ParseException {
9          SimpleCharStream    stream  = new SimpleCharStream(System.in);
10         KlausurTokenManager manager = new KlausurTokenManager(stream);
11         Token               t;
12         sMaxPoints = new Integer(args[0]);
13         do {
14             t = manager.getNextToken();
15         } while (t.kind != 0); // end of file gives t.kind = 0
16     }
17     static Double note() {
18         return 7.0 - 6.0 * sSumPoints / sMaxPoints;
19     }
20 }
21 PARSER_END(Klausur)
22
23 TOKEN: {
24     <KOPF:    (<LETTER>)+ ":" (~["\n"])* "\n">
25   | <NAME:    (<LETTER>)+ " " (<LETTER>)+>
26               {
27                   Klausur.sName      = image.toString();
28                   Klausur.sSumPoints = 0;
29                   System.out.print(Klausur.sName);
30               }
31   | <COLON:   ":" ([ " ", "\t" ])+>
32               {
33                   System.out.print(image);
34               }
35   | <ZAHL:    "0" | ["1"-"9"] (["0"-"9"])*>
36               {
37                   Klausur.sSumPoints += new Integer(image.toString());
38               }
39   | <HYPHEN:  "-">
40   | <EOL:     ([" ", "\t"])* "\n">
41               {
42                   if (Klausur.sName != null) {
43                       System.out.printf(" %3.1f\n", Klausur.note());
44                   }
45               }
46   | <WHITE:   [" ", "\t"]>
47   | <#LETTER: ["a"-"z", "A"-"Z", "ö", "ä", "ü", "Ö", "Ä", "Ü", "ß"]>
48 }
```

Abbildung 3.5: Ein *JavaCC*-Programm zur Notenberechnung

(b) Die Methode `main` bekommt die maximale Punktzahl als Argument übergeben und speichert diese in Zeile 12 in der Variablen `sMaxPoints` ab.

(c) Anschließend lesen wir in der `while`-Schleife solange Tokens, bis wir das *End-Of-File*-Token lesen. Dieses Token können wir daran erkennen, dass die Member-Variable `t.kind` den Wert 0 hat. Diese Schleife bezeichnen wir im Folgenden als die *Scanner-Schleife*.

Jedesmal, wenn der Scanner ein Token erkennt, werden die mit dem Token spezifizierten Aktionen ausgeführt, mehr dazu später.

Neben der Methode *main*(), die in jeder Parser-Klasse vorhanden sein muss, enthält die Klasse `Klausur` noch die Definition der statischen Methode `note`(), mit der später die Note berechnet wird. Hier wird die selbe Formel verwendet, die wir auch schon in dem entsprechenden *Flex*-Beispiel benutzt haben.

3. Nun folgen nach dem Schlüsselwort `Token` und einem Doppelpunkt die einzelnen Token-Definitionen. Diese werden insgesamt von geschweiften Klammern eingefaßt und die einzelnen Token-Spezifikationen werden durch das Zeichen "`|`" voneinander getrennt. Eine einzelne Token-Spezifikationen hat die Form

   <*Name*:*RegExp*> { *CmdList* }

   Die Bedeutung der einzelnen Komponenten ist wie folgt:

   (a) *Name* steht für den Namen eines Tokens. Dieser besteht aus Großbuchstaben.

   (b) *RegExp* bezeichnet einen regulären Ausdruck. Die Syntax der regulären Ausdrücke weicht von der bei *Flex* gebräuchlichen Syntax stark ab. Insbesondere haben Leerzeichen und Tabulatoren keine Bedeutung, so dass es möglich ist, komplexe reguläre Ausdrücke lesbar zu formatieren. Wir werden die genaue Syntax regulärer Ausdrücke im nächsten Unterabschnitt im Detail diskutieren,

   Name und *RegExp* werden zusammen in spitzen Klammern "`<`" und "`>`" eingeschlossen und durch einen Doppelpunkt getrennt.

   (c) *CmdList* steht für eine Liste von Befehlen. Diese Befehle werden ausgeführt, sobald Text erkannt wird, der von dem regulären Ausdruck *RegExp* erkannt wird. Syntaktisch handelt es sich bei den Befehlen um einen in geschweiften Klammern eingeschlossenen Block von *Java*-Code.

   Wir analysieren nun die einzelnen Token-Definitionen aus Abbildung 3.5.

   (a) Die Definition des Tokens `KOPF` in Zeile 24 spezifiziert die Kopfzeilen einer Ergebnis-Datei, die beispielsweise die folgende Form haben:

   ```
   Klausur: Algorithmen und Datenstrukturen
   Kurs:    TIT07AIX
   ```

   Der reguläre Ausdruck, der in Zeile 24 auf den Doppelpunkt folgt, besteht aus vier Komponenten:

   i. `(<LETTER>)+`
   Durch die spitzen Klammern wird das Token `LETTER` referenziert. Dieses Token wird weiter unten in Zeile 47 definiert und steht für einen beliebigen Buchstaben einschließlich eines Umlauts oder "`ß`". Das Token `LETTER` ist in runden Klammern eingeschlossen, auf die noch der Operator "`+`" folgt. Dieser Operator steht genau wie bei *Flex* für eine beliebige positive Anzahl von Wiederholungen. Im Unterschied zu *Flex* muss das Argument, auf dass sich der Operator "`+`" bezieht, aber in runden Klammern eingeschlossen sein. Insgesamt steht der Ausdruck "`(<LETTER>)+`" also für eine beliebige positive Anzahl von Buchstaben.

ii. `":"`
In doppelten Hochkommata "`"`" eingeschlossener Text wird wörtlich interpretiert. Der wesentliche Unterschied zu *Flex*, der hier sichtbar wird, ist die Tatsache, dass innerhalb regulärer Ausdrücke in *JavaCC* Leerzeichen, Tabulatoren und Zeilenumbrüche zulässig sind. Diese werden ignoriert. So wird das Leerzeichen, dass den Ausdruck "`":"`" von dem Ausdruck "`(<LETTER>)+`" trennt, ignoriert. Ich habe dieses Leerzeichen zur Verbesserung der Lesbarkeit eingefügt. Das wäre bei einer *Flex*-Spezifikation so nicht möglich.

iii. `(~["\n"])*`
Dieser Ausdruck beschreibt eine beliebige Anzahl von Zeichen, die von einem Zeilen-Umbruch "`\n`" verschieden sind. Zunächst haben wir hier den in eckigen Klammern eingeschlossenen Ausdruck "`\n`". Eckige Klammern geben, genau wie bei *Flex*, eine Menge von Zeichen an. Im Unterschied zu *Flex* müssen die Zeichen aus dieser Menge allerdings alle in doppelten Hochkommata "`"`" eingefaßt werden. In dem obigen Fall enthält die Menge nur ein einziges Zeichen. **Vor** den eckigen Klammern finden wir aber das Zeichen "`~`". Dieses Zeichen bewirkt eine Komplementbildung: Der Ausdruck "`~["\n"]`" steht also für alle Zeichen, die von einem Zeilen-Umbruch **verschieden** sind.
Hier weicht die Syntax ebenfalls von der bei *Flex* gebräuchlichen Syntax ab, den bei *Flex* wird zur Komplementbildung der Operator "`^`" verwendet und dieser steht dann als erstes Zeichen **nach** der öffnenden eckigen Klammer.
Der ganze Ausdruck ist dann noch in runden Klammern eingefaßt, auf die der Operator "`*`" folgt. Genau wie bei *Flex* steht dieser Operator für eine beliebige Anzahl von Wiederholungen. Im Unterschied zu *Flex* muss das Argument, auf dass sich der Operator "`*`" bezieht, aber in runden Klammern eingeschlossen sein.

iv. `"\n"`
Dieser Ausdruck steht für einen Zeilen-Umbruch.

Hinter dem Token `KOPF` steht kein *Java*-Block. Daher wird der Text, der hier erkannt wurde, einfach ignoriert.

(b) Die Definition von `NAME` sagt, dass eine Name aus zwei Gruppen von Buchstaben besteht, die durch ein Leerzeichen getrennt werden. Wichtig ist hier, dass das Leerzeichen in dem regulären Ausdruck in doppelte Hochkommata "`"`" eingefaßt ist, denn (wie oben bereits erwähnt) können reguläre Ausdrücke bei *JavaCC* durch das Einfügen von Leerzeichen, Tabulatoren und Zeilenumbrüchen zur Verbesserung der Lesbarkeit formatiert werden.

Wir diskutieren nun den *Java*-Block in den Zeilen 26 – 30: Der Text, der durch den regulären Ausdruck erkannt wird, ist in der Variablen `image` abgelegt. Diese Variable entspricht also der *Flex*-Variablen `yytext`. Die Variable `image` hat aus Effizienzgründen den Typ `StringBuffer` und kann mit der Methode *toString*() in einen `String` umgewandelt werden. Mit diesem String initialisieren wir die statische Variable `sName`. Beachten Sie, dass der Klassenname "`Klausur`" der statischen Variablen vorangestellt werden muss, denn der *Java*-Block ist später Teil der Klasse `KlausurTokenManager` und nicht der Klasse `Klausur`.

(c) Die Definition von `COLON` spezifiziert einen Doppelpunkt, auf den Leerzeichen und Tabulatoren folgen. Beachten Sie hier das Komma in der Mengen-Definition

```
[" ", "\t"]
```

Bei *JavaCC* müssen die einzelnen Zeichen einer Mengen-Definition durch Kommata getrennt werden.

(d) Das Token `ZAHL` steht für eine ganze Zahl. Dies ist entweder das Zeichen "`0`" oder eine beliebige Folge von Ziffern, die nicht mit dem Zeichen "`0`" beginnt.

Dieses Beispiel zeigt, dass auch bei *JavaCC* innerhalb einer Mengen-Definition Bereiche gebildet werden können. Dazu wird genau wie bei *Flex* das Minus-Zeichen "`-`" verwendet.

(e) Das Token `HYPHEN` steht für ein einzelnes Minus-Zeichen "`-`".

(f) Das Token `EOL` steht für Leerzeichen und Tabulatoren, die am Zeilenende auftreten. Der abschließende Zeilenumbruch gehört ebenfalls noch zu dem Token. Der Name `EOL` ist als Abkürzung für *end of line* gedacht.

(g) Das Token `WHITE` steht für Leerzeichen und Tabulatoren.

(h) Das Token `LETTER` ist durch das Voranstellen des Zeichens "`#`" als *privat* deklariert worden. Ein solches Token kann nur bei der Definition anderer Token verwendet werden, der Scanner gibt später nie ein Token zurück, das als privat deklariert wurde. Durch die Verwendung privater Token können reguläre Ausdrücke abgekürzt werden. Die *regulären Definitionen* in *Flex* haben die selbe Funktion.

Um das Beispiel zu übersetzen, geben wir die folgenden Befehle ein:

1. `javacc Klausur.jj`

   Dieser Befehl erzeugt die Datei `Klausur.java`. Zusätzlich werden noch die Dateien

   `KlausurConstants.java`, `KlausurTokenManager.java` und `Token.java`

   generiert.

2. `javac Klausur.java`

   Damit werden die erzeugten *Java*-Dateien übersetzt.

3. `java Klausur 60 < ergebnis`

   Hierdurch wird der erzeugte Scanner mit dem Argument 60 aufgerufen. Als Eingabe verarbeitet der Scanner dann die Datei `ergebnis`.

3.2.2 Reguläre Ausdrücke in *JavaCC*

Die Syntax der regulären Ausdrücke ist für *JavaCC* wie folgt:

1. In doppelten Hochkommata eingeschlossen Strings stehen für sich selbst. Beispiel:

   ```
   "else"
   ```

   steht für den String "`else`". Genau wie bei *Flex* haben Operator-Symbole innerhalb eines solchen Strings keine Bedeutung.

2. Bereiche können mit den eckigen Klammern "`[`" und "`]`" gebildet werden, im Unterschied zu *Flex* müssen die einzelnen Zeichen allerdings in Hochkommata eingeschlossen werden. Beispiel:

   ```
   ["0" - "9"]
   ```

   Dieser Ausdruck steht für eine beliebige Ziffer.

3. Die Komplementbildung erfolgt für Bereiche durch das Zeichen "`~`", dass nun **vor** der öffnenden eckigen Klammer steht. Beispiel:

   ```
   ~["a" - "z"]
   ```

   Dieser Ausdruck steht für ein Zeichen, dass **kein** kleiner Buchstabe ist.

4. Der Kleene-Abschluss wird mit dem Postfix-Operator "`*`" gebildet. Im Unterschied zu *Flex* muss das Argument allerdings in runden Klammern eingeschlossen sein. Beispiel:

```
(["0"-"9"])*
```

Dieser Ausdruck steht für einen String, der nur aus Ziffern besteht.

5. Genau wie in *Flex* gibt es die Postfix-Operatoren "`?`" und "`+`", deren Argumente allerdings zusätzlich in runden Klammern eingeschlossen sein müssen.

6. Eine Auswahl zwischen verschiedenen Alternativen erfolgt genau wie bei *Flex* mit Hilfe des Operators "`|`". Beispiel:

   ```
   "0" | ["1"-"9"] (["0"-"9"])*
   ```

   Dieser Ausdruck steht für eine einzelne 0 oder eine Zahl, die mit einer postiven Ziffer anfängt, auf die dann beliebig viele andere Ziffern folgen.

Die Operatoren "`^`" und "`$`", mit denen in *Flex* der Beginn bzw. das Ende einer Zeile spezifiziert werden können, gibt es in *JavaCC* nicht. Auch der Operator "`/`", mit dem *trailing context* spezifiziert werden kann, fehlt in *JavaCC*. Schließlich ist auch der Punkt "`.`", der in *Flex* für ein beliebiges von "`\n`" verschiedenes Zeichen steht, in *JavaCC* kein Operator. Um ein beliebiges Zeichen zu spezifizieren kann in *JavaCC* der Ausdruck

```
~[]
```

verwendet werden, denn dieser Ausdruck spezifiziert das Komplement eines leeren Bereichs und das sind alle Zeichen!

3.2.3 Zustände in *JavaCC*

Auch in *JavaCC* gibt es Zustände. Abbildung 3.6 zeigt ein *JavaCC*-Programm, mit dessen Hilfe Kommentare der Form

```
/* ··· */
```

aus einer `C`-Datei entfernt werden können. Im Unterschied zu *Flex* werden Zustände in *JavaCC* nicht deklariert, es gibt also kein Pendant zu "`%x`" bzw. "`%s`". Außerdem sind in *JavaCC* alle Zustände automatisch exklusiv. Der Default-Zustand heißt jetzt "`DEFAULT`".
Das Programm in Abbildung 3.6 ist wie folgt aufgebaut.

1. Nach der Erzeugung eines Token-Managers finden wir in den Zeilen 8 – 10 die *Scanner-Schleife*.

2. Anschließend finden wir in dem Programm zwei Gruppen von Token-Definitionen.

   (a) Die Zeilen 15 – 18 enthalten die Regeln, die im Zustand `DEFAULT` verwendet werden.

   - Die Regel mit dem Namen `START` erkennt den String "`/*`", mit dem ein mehrzeiliger Kommentar eingeleitet wird. Diese Regel wechselt in den Zustand `ML_COMMENT`.
   - Die Regel mit dem Namen `CHAR` erkennt das Komplement der leeren Zeichenmenge, also ein beliebiges Zeichen. Dieses Zeichen wird unverändert ausgegeben. Der Folge-Zustand ist wieder der Zustand `DEFAULT`, es findet also kein Zustandswechsel statt. Daher könnten wir bei dieser Regel den Rest

     ```
     : DEFAULT
     ```

     auch weglassen, denn die Angabe eines Folge-Zustands ist optional.

   (b) Die Zeilen 19 – 22 enthalten die Regeln, die im Zustand `ML_COMMENT` zum Tragen kommen.

   - Die Regel mit dem Namen `STOP` erkennt den String "`*/`", mit dem ein mehrzeiliger Kommentar beendet wird und wechselt in den Zustand `DEFAULT`.
   - Die Regel mit dem Namen `CHAR` erkennt ein beliebiges Zeichen, was dann einfach verschluckt wird.

```
1  PARSER_BEGIN(DeComment)
2
3  public class DeComment {
4      public static void main(String args[]) throws ParseException {
5          SimpleCharStream      stream  = new SimpleCharStream(System.in);
6          DeCommentTokenManager manager = new DeCommentTokenManager(stream);
7          Token t;
8          do {
9              t = manager.getNextToken();
10         } while (t.kind != 0); // end of file gives t.kind = 0
11     }
12 }
13 PARSER_END(DeComment)
14
15 <DEFAULT> TOKEN: {
16     <START: "/*"> { /* change state here */  } : ML_COMMENT
17 |   <CHAR:  ~[]>  { System.out.print(image); } : DEFAULT
18 }
19 <ML_COMMENT> TOKEN: {
20     <STOP: "*/"> {} : DEFAULT
21 |   <EAT:  ~[]>  {} : ML_COMMENT
22 }
```

Abbildung 3.6: Entfernung mehrzeiliger Kommentare

Wir erkennen an dem Beispiel, dass die allgemeine Syntax einer Regel die folgende Form hat:

<*Name*:*Regexp*> { *CmdList* } : *State*

Hierbei gilt:

1. *Name* ist der Name, unter dem das Token später angesprochen werden kann. Solange wir *JavaCC* nur als Scanner verwenden, hat dieser Name keine Bedeutung. Im Allgemeinen wird *JavaCC* aber als *Parser-Generator* eingesetzt und dann können wir in den Grammatik-Regeln auf die Namen zurück greifen.

2. *Regexp* ist der reguläre Ausdruck, der erkannt werden soll.

3. *CmdList* ist die Gruppe von Befehlen, die ausgeführt werden, sobald der reguläre Ausdruck erkannt worden ist.

4. *State* ist der Zustand, in den der Automat wechseln soll, nachdem die Kommandos abgearbeitet sind. Die Angabe des Folge-Zustands ist optional. Wird kein Folge-Zustand angegeben, so bleibt der Scanner in dem Zustand, in dem er gerade ist.

Aufgabe: Entwickeln Sie einen Assembler für die im Rechnertechnik-Skript beschriebene Assembler-Sprache. Das Programm soll eine Assembler-Datei in eine binäre Datei übersetzen.

Hinweise:

1. Verwenden Sie verschiedene Zustände um die einzelnen Komponenten eines Assembler-Befehls zu lesen.

2. Passen Sie die Klasse `Assembler`, die in dem Rechnertechnik-Skript vorgestellt wird, an Ihren Bedarf an und verwandeln Sie diese Klasse in eine konkrete Klasse. Die abgeleiteten Klassen sind nicht mehr erforderlich.

3. Um eine 32-Bit-Zahl binär auszugeben, können Sie die folgende Methode verwenden:

```
public void writeBinary(OutputStream writer, int code) throws IOException
{
    int b0   = (code >>  0) & 255;
    int b1   = (code >>  8) & 255;
    int b2   = (code >> 16) & 255;
    int b3   = (code >> 24) & 255;
    writer.write(b0);
    writer.write(b1);
    writer.write(b2);
    writer.write(b3);
}
```

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

4.1 Deterministische endliche Automaten

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

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

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

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

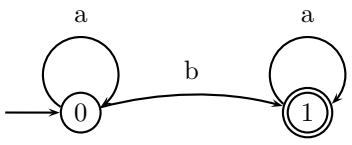


Abbildung 4.1: Ein einfacher endlicher Automat zur Erkennung der durch den regulären Ausdruck "$\texttt{a}^* \cdot \texttt{b} \cdot \texttt{a}^*$" definierten Sprache.

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

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

beschrieben werden. Der Automat besteht aus den beiden Zuständen 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 diesem Zustand. 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 den Buchstaben "b" liest, der Folgezustand ist dann undefiniert.

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

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

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

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

Die einzelnen Komponenten haben die folgende Bedeutung:

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

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

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

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

wobei gilt:

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

Um die von einem endlichen Automaten akzeptierte Sprache formal definieren zu können, führen wir den Begriff der *Konfiguration* (engl. *instantaneous description)*) eines endlichen Automaten ein. Darunter verstehen wir ein Paar

$$\langle q, s\rangle,$$

wobei q ein Zustand ist und s ein String, und zwar genau der Teil der Eingabe, der noch nicht von dem EA gelesen worden ist. Weiter definieren wir für Konfigurationen eine Relation $\rightsquigarrow$. Es soll

$$\langle q_1, cs\rangle \rightsquigarrow \langle q_2, s\rangle$$

genau dann gelten, wenn der Automat aus dem Zustand q_1 beim Lesen des Buchstabens c in den Zustand q_2 übergeht, also

$$\langle q_1, cs\rangle \rightsquigarrow \langle q_2, s\rangle \quad \text{g.d.w.} \quad \delta(q_1, c) = q_2.$$

In diesem Fall nennen wir $\langle q_2, s\rangle$ die *Folge-Konfiguration* von $\langle q_1, cs\rangle$. Falls $\delta(q_1, c) = \Omega$ ist, dann ist die Folge-Konfiguration von $\langle q_1, cs\rangle$ undefiniert:

$$\langle q_1, cs\rangle \rightsquigarrow \Omega \quad \text{g.d.w.} \quad \delta(q_1, c) = \Omega.$$

Wir definieren nun den transitiven Abschluss der Relation $\rightsquigarrow^*$. Anschaulich bedeutet

$$\langle q_1, s_1 s_2\rangle \rightsquigarrow^* \langle q_2, s_2\rangle,$$

dass der EA ausgehend von dem Zustand q_1 den String s_1 liest und dabei in den Zustand q_2 übergeht. Die formale Definition erfolgt induktiv.

I.A.: $\langle q, s\rangle \rightsquigarrow^* \langle q, s\rangle$.

I.S.: $\langle q_1, s_1\rangle \rightsquigarrow^* \langle q_2, s_2\rangle \;\wedge\; \langle q_2, s_2\rangle \rightsquigarrow \langle q_3, s_3\rangle \;\Rightarrow\; \langle q_1, s_1\rangle \rightsquigarrow^* \langle q_3, s_3\rangle$.

Wenn der EA von der Konfiguration in $\langle q_1, s_1\rangle$ endlich vielen Schritten in die Konfiguration $\langle q_2, s_2\rangle$ und von dieser Konfiguration dann in einem Schritt in die Konfiguration $\langle q_3, s_3\rangle$ übergehen kann, dann kann der Automat in endlich vielen Schritten von der Konfiguration $\langle q_1, s_1\rangle$ in die Konfiguration $\langle q_3, s_3\rangle$ übergehen.

Damit können wir die von einem Automaten A akzeptierte Sprache $L(A)$ wie folgt definieren. Es sei q_0 der Start-Zustand des EA A und die Menge der akzeptierenden Zustände sei F. Dann setzen wir

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

Die akzeptierte Sprache besteht also aus allen Strings s, so dass der EA aus der Konfiguration $\langle q_0, s\rangle$ nach endlich vielen Schritten in eine Konfiguration $\langle p, \varepsilon\rangle$ gelangt, bei der einerseits der Zustand p ein Element der Menge der akzeptierenden Zustände ist und bei der andererseits der restliche, noch ungelesene String leer ist, so dass also der gesamte String s gelesen worden ist.

Aufgabe 1: 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://www.belgarath.org/java/fsme.html`

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

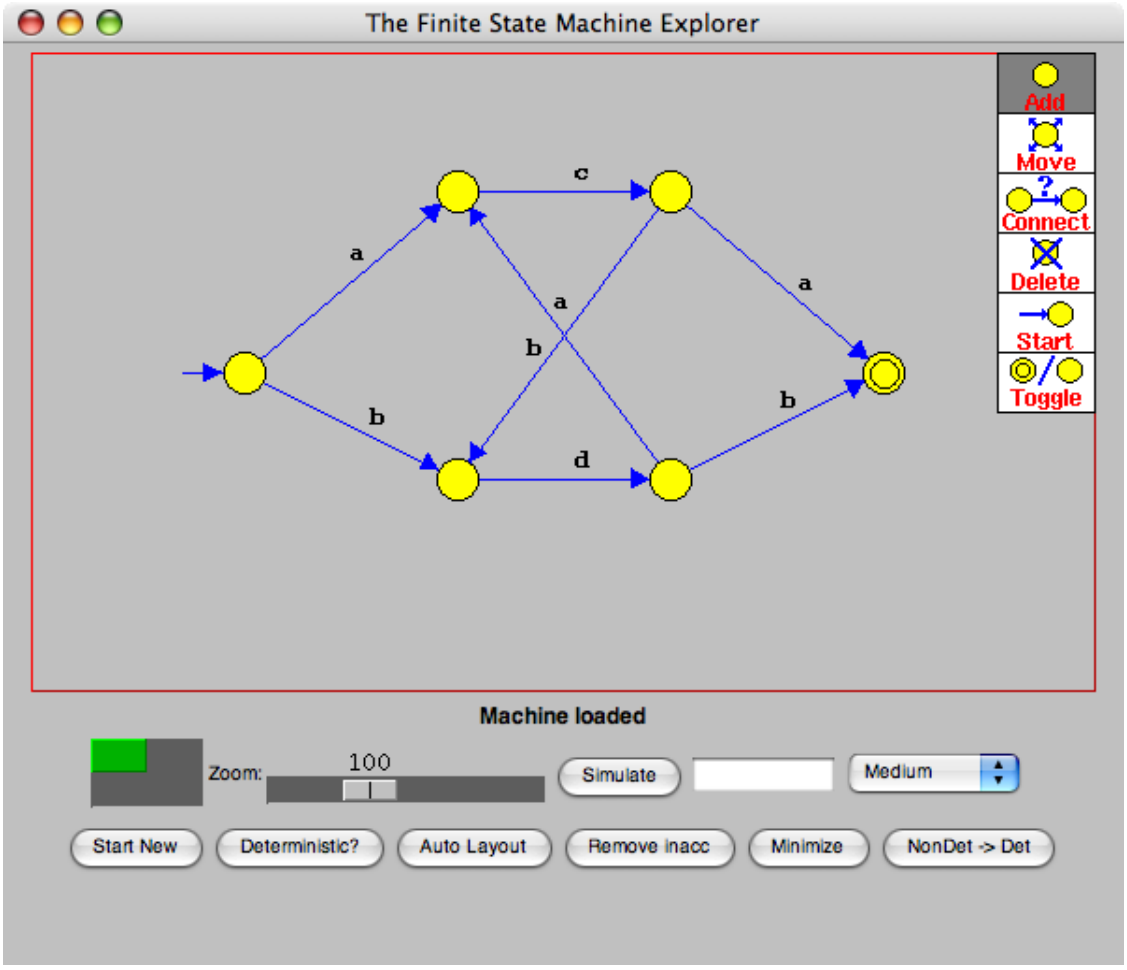


Abbildung 4.2: Das Werkzeug FSME.

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

Satz 9 Zu jedem endlichen deterministischen Automaten A gibt es einen vollständigen deterministischen Automaten $\hat{A}$, der die selbe Sprache akzeptiert wie der Automat A, es gilt also:

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

Beweis: Der Automat A habe die Form

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

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

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

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

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

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

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

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

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

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

   (b) $\delta(q, c) = \Omega \to \hat{\delta}(q, c) = \dagger$,
   wenn die Zustands-Übergangs-Funktion δ für den Zustand q und den Buchstaben c als Ergbenis Ω liefert und also undefiniert ist, dann produziert $\hat{\delta}$ als Ergebnis den toten Zustand $\dagger$.

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

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

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

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

4.2 Nicht-deterministische endliche Automaten

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

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

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

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

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

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

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

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

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

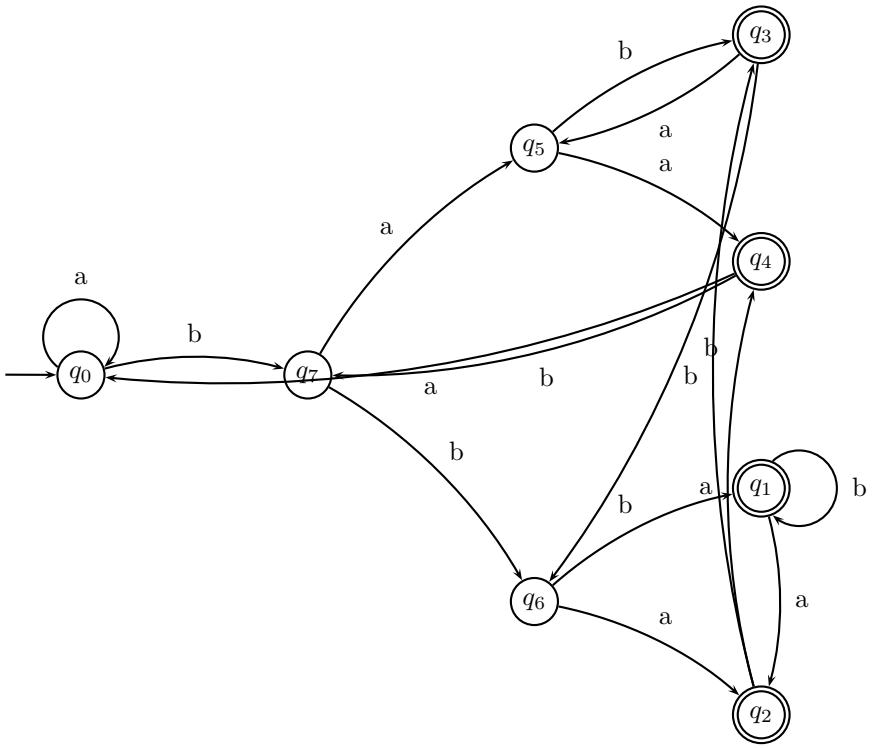


Abbildung 4.3: Ein endlicher Automat für die Sprache, die durch den regulären Ausdruck "$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$" definiert wird

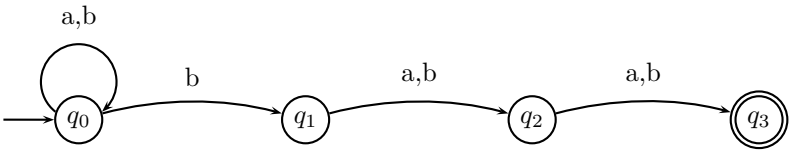


Abbildung 4.4: Ein Nicht-deterministischer Automat für die Sprache, die durch den regulären Ausdruck "$(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})$" definiert wird.

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 ein mögliche *Berechnung* des Automaten zu der Eingabe "`abab`":

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

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

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

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

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

Hier ist der Automat zu früh in den Zustand q_1 gewechselt, was bei der Eingabe des letzten Zeichens zum Tode des Automaten führt. Wäre der Automat beim Lesen des zweiten Buchstabens b noch im Zustand q_0 geblieben, so hätte er das Wort "bbbbb" erkennen können:

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

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

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

Definition 10 (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 ist der Start-Zustand.
5. F 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 38 gezeigten nicht-deterministischen endlichen Automaten A gilt

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

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

5. $F = \{q_3\}$, der einzige akzeptierende Zustand ist also q_3. □

Der Begriff der *Konfiguration* eines NEA ist (genau wie die Konfiguration eines EA) definiert als ein Paar

$$\langle q, s \rangle$$

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

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

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

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

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

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

Wir beichnen den transitiven Abschluss der Relation $\rightsquigarrow$ wieder mit $\rightsquigarrow^*$. Die formale Definition ist die selbe wie im Falle deterministischer endlicher Automaten und wird deshalb hier nicht noch einmal angegeben. Die von einem nicht-deterministischer endlicher Automaten A akzeptierte Sprache $L(A)$ ist definiert als

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

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

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

Aufgabe 2: Geben Sie einen NEA F an, so dass $L(F)$ aus genau den Wörtern der Sprache $\{a, b\}^*$ besteht, die den Teilstring "`aba`" enthalten.

4.3 Äquivalenz von EA und NEA

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

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

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

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

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

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

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

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

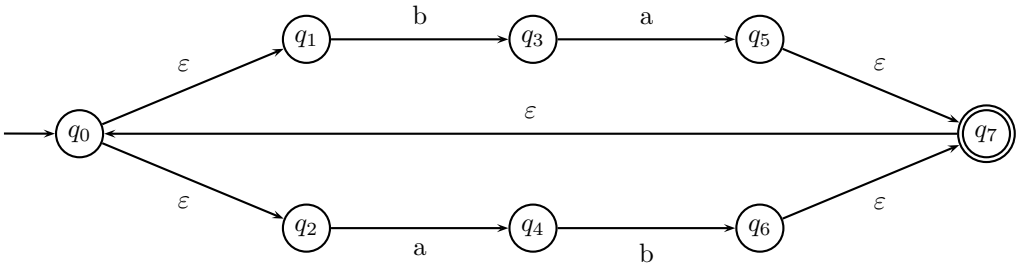


Abbildung 4.5: Nicht-deterministischer Automat mit ε-Transitionen.

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

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

Um den am Anfang des Abschnitts angegebenen nicht-deterministischen Automaten A_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

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

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 $ec(q_2)$ und vereinigen die Mengen, die wir so erhalten.

Beispiel: In Fortführung des obigen Beispiels erhalten wir beispielsweise:

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

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

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

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

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

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

Die Funktion δ^* überführt einen Zustand in eine Menge von Zuständen. Da der zu entwickelnde endliche Automat 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

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

so, dass $\Delta^*(M, c)$ für eine Menge von Zuständen M und einen Buchstaben c die Menge aller der Zustände berechnet, in denen der Automat A_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 einfachzu verstehen: Für jeden Zustand $q \in M$ berechnen wir zunächst die Menge aller Zustände, in denen sich der Automat nach Lesen von c und eventuellen ε-Transitionen befinden kann. Die so erhaltenen Mengen vereinigen wir.

Beispiel: In Fortführung des obigen Beispiels erhalten wir beispielsweise:

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

Wir haben nun alles Material zusammen, um den nicht-deterministischen endlichen Automaten A_1 in einen deterministischen endlichen Automaten überführen zu können. Wir definieren

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

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

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

Beispiel: Wir zeigen, wie sich der in Abbildung 4.4 auf Seite 38 gezeigte nicht-deterministische Automat A_1 in einen deterministischen Automaten A_2 transformieren läßt. Der Start-Zustand des deterministischen Automaten, den wir mit S_0 bezeichnen wollen, besteht aus der Menge, die nur den Knoten q_0 enthält:

$$S_0 = \{q_0\}.$$

Um die Notation zu vereinfachen, interpretieren wir die Relation

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

als Funktion

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

in dem wir vereinbaren, dass gilt:

$$p \in \delta(q, c) \Leftrightarrow \langle q, c, p\rangle \in \delta \quad \text{und} \quad p \in \delta(q, \varepsilon) \Leftrightarrow \langle q, \varepsilon, p\rangle \in \delta.$$

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

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

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

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

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

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

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

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

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

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

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

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

9. $\Delta^*(\{q_0, q_3\}, \mathtt{a}) = \{q_0\}$.
10. $\Delta^*(\{q_0, q_3\}, \mathtt{b}) = \{q_0, q_1\}$.
11. $\Delta^*(\{q_0, q_1, q_3\}, \mathtt{a}) = \{q_0, q_2\}$.
12. $\Delta^*(\{q_0, q_1, q_3\}, \mathtt{b}) = \{q_0, q_1, q_2\}$.
13. $\Delta^*(\{q_0, q_2, q_3\}, \mathtt{a}) = \{q_0, q_3\}$.
14. $\Delta^*(\{q_0, q_2, q_3\}, \mathtt{b}) = \{q_0, q_1, q_3\}$.
15. $\Delta^*(\{q_0, q_1, q_2, q_3\}, \mathtt{a}) = \{q_0, q_2, q_3\}$.
16. $\Delta^*(\{q_0, q_1, q_2, q_3\}, \mathtt{b}) = \{q_0, q_1, q_2, q_3\}$.

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 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 3: Transformieren Sie den in Abbildung 4.5 auf Seite 41 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:

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

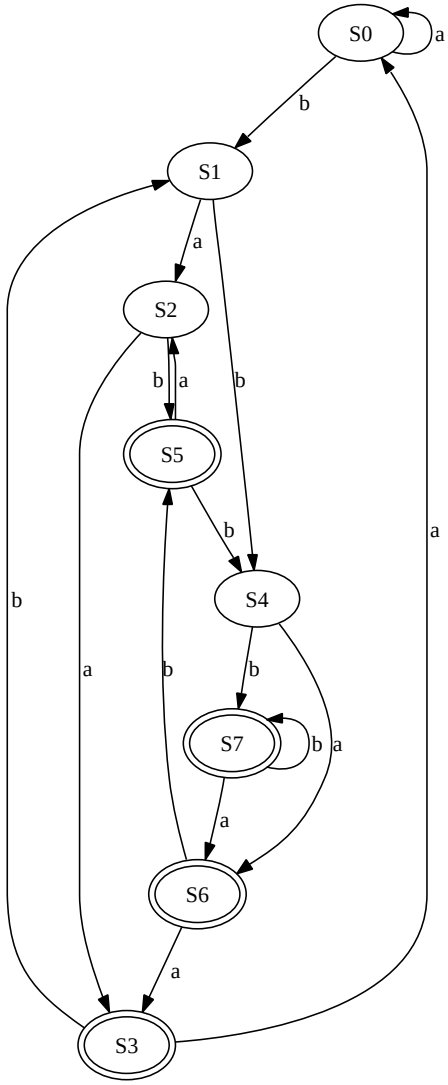


Abbildung 4.6: Der deterministische Automat A_2.

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

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

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

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

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

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

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

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

   Abbildung 4.7 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

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

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

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

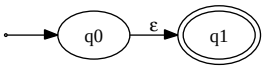


Abbildung 4.8: Der Automat $A(\varepsilon)$.

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

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

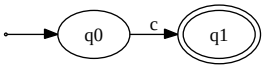


Abbildung 4.9: Der Automat $A(c)$.

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.

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

   (a) $A(r_1) = \langle Q_1, \Sigma, \delta_1, q_1, \{q_2\} \rangle$,

   (b) $A(r_2) = \langle Q_2, \Sigma, \delta_2, q_3, \{q_4\} \rangle$.

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

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

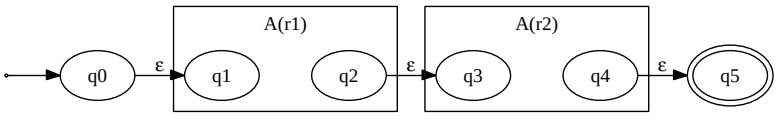


Abbildung 4.10: Der Automat $A(r_1 \cdot r_2)$.

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-Übergänge 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

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

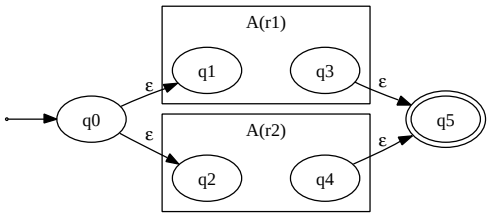


Abbildung 4.11: Der Automat $A(r_1 + r_2)$.

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

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

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

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

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

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

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

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

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

ist gegeben durch

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

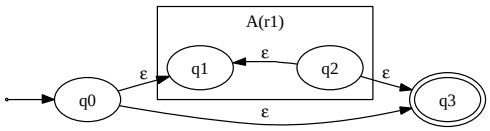


Abbildung 4.12: Der Automat $A(r^*)$.

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

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

spezifizierte Sprache erkennt.

4.5 Übersetzung eines EA in einen regulären Ausdruck

Wir runden die Theorie ab indem wir zeigen, dass sich zu jedem deterministischen endlichen Automaten A ein regulärer Ausdruck r 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

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

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

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

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

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

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

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

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

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

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

I.A.: $p \overset{\varepsilon}{\mapsto}_k p$,

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

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

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

Damit können wir nun für alle $k = 0, \cdots, n+1$ die regulären Ausdrücke $r^{(k)}(p_1, p_2)$ definieren. Die Idee dieser Definition ist durch die folgende Formel gegeben:

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

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

I.A.: $k = 0$. Dann gilt $Q_0 = Q$, die Menge Q_0 enthält also alle Zustände und damit dürfen wir, wenn wir vom Zustand p_1 in den Zustand p_2 übergehen, zwischendurch überhaupt keine Zustände besuchen.

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

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

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

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

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

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

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

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

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

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

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

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

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 $p_1 \overset{w}{\mapsto}_k p_2$.

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

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

Daher definieren wir

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Aufgabe 5: Konstruieren Sie für den in Abbildung 4.1 gezeigten endlichen Automaten einen äquivalenten regulären Ausdruck.

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

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

(b) Für $r^{(2)}(0,1)$ finden wir

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

(c) Für $r^{(2)}(1,0)$ finden wir

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

(d) Für $r^{(2)}(1,1)$ finden wir

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

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

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

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

4.6 Minimierung endlicher Automaten

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

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

minimiert werden kann. 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 durchfüren zu können, müssen wir etwas ausholen. Zunächst erweitern wir die Funktion

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

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

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

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

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

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

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

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. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
3. $\delta(0, \mathtt{a}) = 1,\ \delta(0, \mathtt{b}) = 2,\ \delta(1, \mathtt{a}) = 0,\ \delta(2, \mathtt{a}) = 0,\ \delta(1, \mathtt{b}) = 3,\ \delta(2, \mathtt{b}) = 3$,
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. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
3. $\delta(0, \mathtt{a}) = 1,\ \delta(0, \mathtt{b}) = 1,\ \delta(1, \mathtt{a}) = 0,\ \delta(1, \mathtt{b}) = 3$,
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 $\mathit{reduce}(A, p \mapsto q)$ 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

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

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.

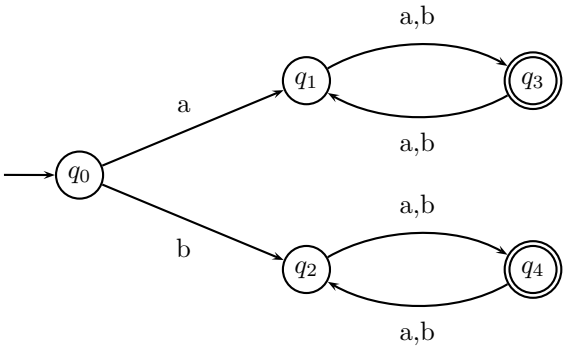


Abbildung 4.13: Ein endlicher Automat mit äquivalenten Zustände, die nicht gleichwertig sind.

Definition 11 (Unterscheidbar) Zwei Zustände $p_1, p_2 \in Q$ sind *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. $\delta(p_2, s) \notin F$ und $\delta(p_2, s) \in F$. □

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

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

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

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

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

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

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

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

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

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

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. Zu Beginn ist die Tabelle leer:

	0	1	2	3	4
0					
1					
2					
3					
4					

2. 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	$\times$	$\times$	$\times$		

3. 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	$\times$	$\times$	$\times$		

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

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

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

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

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

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

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.

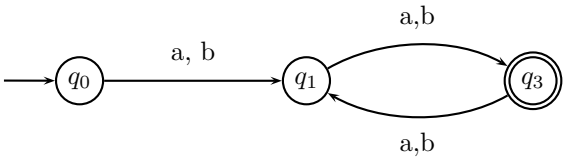


Abbildung 4.14: Der reduzierte endliche Automat.

Aufgabe 6: Konstruieren Sie den minimalen deterministischen endlichen Automaten, der die Sprache $L\big(a \cdot (b \cdot a)^*\big)$ erkennt. Gehen Sie dazu in folgenden Schritten vor:

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

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

Kapitel 5

Reguläre Sprachen

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

$$L = L(r)$$

gilt. Wir wollen in diesem Kapitel einige Eigenschaften regulärer Sprachen diskutieren. Wir werden sehen, dass reguläre Sprachen bestimmte Abschluss-Eigenschaften haben:

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

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

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

nicht regulär ist.

5.1 Abschluss-Eigenschaften regulärer Sprachen

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

Satz 12 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 13 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 folgt, 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. Als Baumaterial verwenden wir die Automaten A_1 und A_2. Wir nehmen an, dass

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

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

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

wobei die Zustands-Übergangs-Funktion

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

durch die Gleichung

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

definiert wird. Der so definierte endliche Automat A simuliert gleichzeitig die beiden Automaten A_1 und A_2 indem er parallel berechnet, in welchem Zustand jeweils A_1 und A_2 ist. Damit das möglich ist, bestehen die Zustände von A aus Paaren $\langle p_1, p_2 \rangle$, so dass p_1 ein Zustand von A_1 und p_2 en 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:

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

Damit gilt für alle $s \in \Sigma^*$:

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

Damit haben wir insgesamt gezeigt, dass

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

gilt und das war zu zeigen. □

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

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

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

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

gilt: Zunächst berechnen wir zu r_1 und r_2 äquivalente nicht-deterministische endliche Automaten, überführen diese Automaten dann in einen vollständigen deterministischen Automaten, bilden

wie oben gezeigt das kartesische Produkt dieser Automaten und gewinnen schließlich aus diesem Automaten einen regulären Ausdruck zurück. Der so gewonnene reguläre Ausdruck wäre allerdings so groß, dass diese Funktion in der Praxis nicht implementiert wird, denn bei der Überführung eines nicht-deterministischen in einen deterministischen Automaten wächst der Automat stark an und der reguläre Ausdruck, der sich aus einem Automaten ergibt, kann schon bei verhältnismäßig kleinen Automaten sehr unübersichtlich werden.

Satz 14 Ist L eine reguläre Sprache über dem Alphabet Σ, so ist auch das Komplement von L, die Sprache $\Sigma^* \backslash L$ eine reguläre Sprache.

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

$$L = L(A).$$

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

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

Dann definieren wir

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

Offenbar gilt

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

und daraus folgt die Behauptung. □

Korollar 15 Sind L_1 und L_2 reguläre Sprachen, so ist auch die Mengen-Differenz $L_1 \backslash L_2$ eine reguläre Sprache.

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

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

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

5.2 $L(A) = \{\}$?

In diesem Abschnitt untersuchen wir für einen gegeben deterministischen endlichen Automaten

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

die Frage, ob die von A erkannte Sprache leer ist, ob also $L(A) = \{\}$ gilt. Dazu fassen wir den endlichen Automaten als einen Graphen auf: Die Knoten dieses Graphen sind die Zustände von A und zwischen zwei Zuständen q_1 und q_2 gibt es genau dann eine Kante, die q_1 mit q_2 verbindet, wenn es einen Buchstaben $c \in \Sigma$ gibt, so dass $\delta(q_1, c) = q_2$ gilt. Dann berechnen wir die Menge R der von dem Start-Zustand q_0 erreichbaren Zustände induktiv:

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

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

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

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

5.3 Äquivalenz regulärer Ausdrücke

Definition 16 ($\approx$) Zwei reguläre Ausdrücke r_1 und r_2 sind genau dann äquivalent (geschrieben $r_1 \approx r_2$), wenn die durch r_1 und r_2 spezifizierten Sprachen identisch sind:

$$r_1 \approx r_2 \stackrel{\text{def}}{\Longleftrightarrow} L(r_1) = L(r_2). \qquad \square$$

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

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

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

gilt, entscheidbar.

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

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

Seien nun A_1 und A_2 endliche Automaten mit

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

Im letzten Kapite haben wir gesehen, wie wir solche Automaten konstruieren können. Nach dem Korollar 15 sind die Sprachen $L(r_1)\backslash L(r_2)$ und $L(r_2)\backslash L(r_1)$ regulär. Damit gibt es endliche Automaten A_{12} und A_{21}, so dass

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

gilt. Damit gilt nun

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

und diese Frage ist nach Abschnitt 5.2 entscheidbar. $\square$

5.4 Grenzen regulärer Sprachen

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

Satz 18 (Pumping-Lemma) Es sei L eine reguläre Sprache. Dann gibt es eine natürliche Zahl $n \in \mathbb{N}$, so dass sich alle Strings $s \in L$, 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 endliche Automaten

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

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

$$n := card(Q).$$

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

$$s = c_1 c_2 \cdots c_m,$$

wobei c_1, $\cdots$, c_m die einzelnen Buchstaben sind. Wir betrachten eine Berechnung des Automaten A, die das Wort s akzeptiert. Diese Berechnung hat die Form

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

und es gilt $q_m \in F$. Die Länge m des Wortes s ist nach Voraussetzung größer als n. Daher können in der Liste

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

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

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

können wir sogar sagen, dass in der Liste

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Aus den Gleichungen (5.1), (5.3) und 5.4) folgt nun

$$q_0 \overset{uv^hw}{\mapsto} q_m$$

und da q_m ein akzeptierender Zustand ist, haben wir damit $uv^hw \in L$ gezeigt. □

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

Satz 19 Das Alphabet Σ sei durch $\Sigma = \{$ "(" , ")" $\}$ definiert, es enthält also die beiden Klammer-Symbole "(" und ")". Die Sprache L sei die Menge aller Strings, die aus k öffnenden runden Klammern gefolgt von k schließenden runden Klammern besteht:

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

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

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

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:

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

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 $\forall h \in \mathbb{N} : uv^hw \in L)$ für h den Wert 0 ein, so sehen wir, dass

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

gilt. Der String uw enthält genauso viele schließende runde Klammern wie der String $s = uvw$, aber uv enthält offenbar k öffnende runde Klammern weniger als s. Folglich enthält uw weniger öffnende Klammern als schließende Klammern und kann daher nicht in der Menge L liegen. Dieser Widerspruch zu (5.5) zeigt, dass die Sprache L nicht regulär sein kann. □

Bemerkung: Der letzte Satz zeigt uns, dass wir mit Hilfe von regulären Ausdrücken noch nicht einmal Klammern zählen können. Der Begriff der regulären Ausdrücke ist damit offensichtlich zu schwach um die Sysntax gängiger Programmier-Sprache adäquat zu beschreiben. Im nächsten Kapitel werden wir daher ein Konzept kennen lernen, das wesentlichen mächtiger als das Konzept der regulären Sprachen ist.

Aufgabe 7: Die Sprache L_{square} beinhaltet alle Wörter der Form $\mathtt{a}^n$ für die n eine Quadrat-Zahl ist, es gilt also

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

Zeigen Sie, dass die Sprache L_{square} keine reguläre Sprache ist.
Hinweis: Nutzen Sie aus, dass der Abstand zwischen den Quadrat-Zahlen beliebig groß wird.

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

Kapitel 6

Kontextfreie Sprachen

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

6.1 Kontextfreie Grammatiken

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

- Operator-Symbole, wie "`+`", "`+=`", "`<`", "`<=`" etc.,
- 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:

   $$A \in \mathit{ArithExpr} \wedge P \in \mathit{Product} \rightarrow A \text{ "+" } P \in \mathit{ArithExpr}.$$

   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. Diese sind natürlich nicht Teil des arithmetischen Ausdrucks sondern dienen lediglich der Notation.

2. Genauso haben wir $A \in \mathit{ArithExpr} \wedge P \in \mathit{Product} \rightarrow A$ "`-`" $P \in \mathit{ArithExpr}$.

3. Jedes Product ist ein arithmetischer Ausdruck

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

4. Ist P ein Produkt und ist F ein Faktor, so ist der String P "`*`" F ein Produkt:

   $$P \in \mathit{Product} \wedge F \in \mathit{Factor} \rightarrow P \text{ "*" } F \in \mathit{Product}.$$

5. Ist P ein Produkt und ist F ein Faktor, so ist der String P "`/`" F ein Produkt:

   $$P \in \mathit{Product} \wedge F \in \mathit{Factor} \rightarrow P \text{ "/" } F \in \mathit{Product}.$$

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

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

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:

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

8. Jede Zahlenkonstante ist ein Faktor:

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

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

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

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

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

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

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

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

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

Definition 20 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* ist ein 4-Tupel $\langle V, T, R, S\rangle$, so dass folgendes gilt:

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

   In dem obigen Beispiel gilt

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

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

   $$T = \{\mathit{Number},\ \text{“+”}\ ,\ \text{“-”}\ ,\ \text{“*”}\ ,\ \text{“/”}\ ,\ \text{“(”}\ ,\ \text{“)”}\ \}$$

3. R ist die Menge der Regeln. Formal ist eine Regel ein Paar:

   (a) Die erste Komponente dieses Paares ist eine syntaktische Variable.

   (b) Die zweite Komponente ist eine endlichen Folge von syntaktischen Variablen und Terminalen. Da endliche Folgen nichts anderes sind als Strings, ist die zweite Komponente damit ein Element der Menge

   $$(V \cup T)^*.$$

   Insgesamt gilt also für die Menge der Regeln R

   $$R \subseteq V \times (V \cup T)^*$$

   Ist $\langle X, \alpha\rangle$ eine Regel, so schreiben wir diese Regel als

   $$X \rightarrow \alpha.$$

   Beispielsweise haben wir oben die erste Regel als

   $$\mathit{ArithExpr} \rightarrow \mathit{ArithExpr}\ \text{“+”}\ \mathit{Product}$$

   geschrieben. Formal steht diese Regel für das Paar

   $$\big\langle \mathit{ArithExpr}, [\mathit{ArithExpr},\ \text{“+”}\ , \mathit{Product}]\big\rangle.$$

4. S ist ein Element der Menge V, das wir als das *Start-Symbol* bezeichnen.

   In dem obigen Beispiel ist *ArithExpr* das Start-Symbol. □

6.1.1 Ableitungen

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

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

Dann kann der String $\alpha A \beta$ durch einen Ableitungs-Schritt in den String $\alpha \gamma \beta$ überführt werden, wir ersetzen also ein Auftreten der syntaktische Variable A durch die rechte Seite der Regel $A \to \gamma$. Diesen Ableitungs-Schritt schreiben wir als

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta.$$

Geht die verwendete Grammatik G aus dem Zusammenhang klar hervor, so wird der Index $_G$ weggelassen und wir schreiben kürzer $\Rightarrow$ an Stelle von $\Rightarrow_G$. Der transitive und reflexive Abschluss der Relation $\Rightarrow_G$ wird mit $\Rightarrow_G^*$ bezeichnet. Wollen wir ausdrücken, dass die Ableitung des Strings w aus dem Nicht-Terminal A aus n Ableitungs-Schritten besteht, so schreiben wir

$$A \Rightarrow^n w.$$

Wir geben ein Beispiel:

$$\begin{array}{lcl}
\mathit{ArithExpr} & \Rightarrow & \mathit{ArithExpr}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Product}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Product}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Factor}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Factor}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Product} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Factor} \\
& \Rightarrow & \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}
\end{array}$$

Damit haben wir also gezeigt, dass

$$\mathit{ArithExpr} \Rightarrow^* \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}$$

oder genauer

$$\mathit{ArithExpr} \Rightarrow^8 \mathit{Number}\ \text{“*”}\ \mathit{Number}\ \text{“+”}\ \mathit{Number}$$

gilt. Ersetzen wir hier das Terminal *Number* durch verschiedene Zahlen, so haben wir damit beispielsweise gezeigt, dass der String

$$2 * 3 + 4$$

ein arithmetischer Ausdruck ist. Allgemein definieren wir die durch eine Grammatik G definierte Sprache $L(G)$ als die Menge aller Strings, die einerseits nur aus Terminalen bestehen und die sich andererseits aus dem Start-Symbol S der Grammatik ableiten lassen:

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

Beispiel: Die Sprache

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

wird von der Grammatik

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

erzeugt, wobei die Regeln R wie folgt gegeben sind:

$$\begin{array}{lcl} S & \to & \varepsilon \\ & \mid & \text{“(” } S \text{ “)”} \end{array}$$

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

$$w \in L \;\to\; S \Rightarrow^* w.$$

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 $S \to \varepsilon$.

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

$$S \Rightarrow \text{ “(” } S \text{ “)” } \Rightarrow^* \text{ “(” } v \text{ “)” } = w.$$

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

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

$$S \Rightarrow \varepsilon.$$

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

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

$$S \Rightarrow \text{ “(” } S \text{ “)” } \Rightarrow^n w$$

Daraus folgt

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

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

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

Aufgabe 8: Wir definieren für $w \in \Sigma^*$ und $c \in \Sigma$ die Funktion

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

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

I.A. $w = \varepsilon$. Dann setzen wir

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

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

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

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

$$L := \big\{w \in \Sigma^* \mid \mathit{count}(w, \mathtt{a}) = \mathit{count}(w, \mathtt{b})\big\}$$

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

Aufgabe 9: Wieder sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Wir definieren die Menge L als die Menge der Strings s, die sich nicht in der Forms $s = ww$ schreiben lassen:

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

Die erste korrekte Lösung dieser Aufgabe, die mich per Email erreicht, prämiere ich mit einer Flache Wein.

6.1.2 Parse-Bäume

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

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

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

1. Jeder innere Knoten (also jeder Knoten, der kein Blatt ist), ist mit einem Nicht-Terminal beschriftet.

2. Jedes Blatt ist mit einem Terminal oder mit ε beschriftet.

3. Ist ein innerer Knoten mit einer Variablen A beschriftet und sind die Kinder dieses Knotens mit den Symbolen X_1, X_2, $\cdots$ X_n beschriftet, so enthält die Grammatik G eine Regel der Form

   $$A \to X_1 X_2 \cdots X_n.$$

Die Blätter des Parse-Baums ergeben dann, wenn wir sie von links nach rechts lesen, ein Wort, das von der Grammatik G abgeleitet wird. Abbildung 6.1 zeigt einen Parse-Baum für das Wort "`2*3+4`", der mit der oben angegebenen Grammatik für arithmetische Ausdrücke abgeleitet worden ist.

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

1. Ist n ein innerer Knoten, der mit der Variablen A beschriftet ist und gibt es unter den Kindern dieses Knotens genau ein Kind, dass mit einem 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, $\cdots$, o_k beschriftet sind. In diesem Fall wird der Knoten mit der Folge der Terminale 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 dass, was wir brauchen um den arithmetischen Ausdruck "`2*3+4`" auszuwerten.

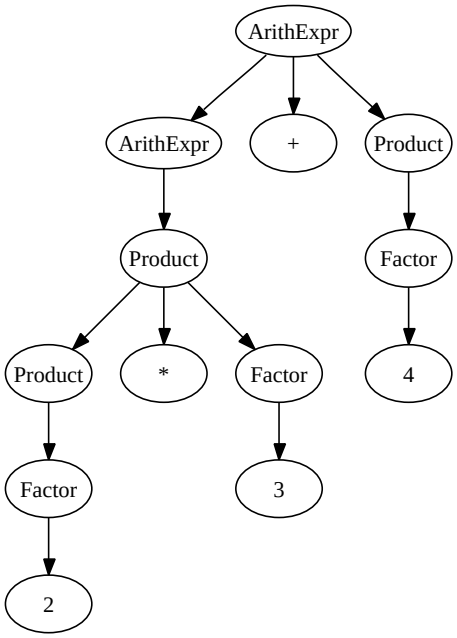


Abbildung 6.1: Ein Parse-Baum für den String "`2*3+4`".

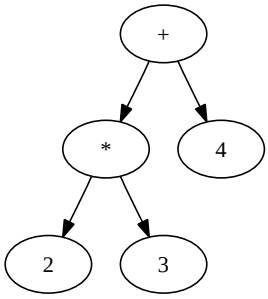


Abbildung 6.2: Ein abstrakter Syntax-Baum für den String "`2*3+4`".

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:

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

wobei die Regeln R wie folgt gegeben sind:

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

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

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \mathit{Expr}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \texttt{4}
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \mathit{Expr}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \mathit{Expr} \\
& \Rightarrow & \texttt{2}\ \text{“}\texttt{*}\text{”}\ \texttt{3}\ \text{“}\texttt{+}\text{”}\ \texttt{4}
\end{array}
$$

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!

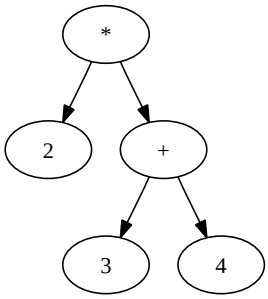


Abbildung 6.3: Ein anderer abstrakter Syntax-Baum für den String "`2*3+4`".

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

$$A \to X_1 X_2 \cdots X_n$$

zu parsen, versuchen wir, zunächst ein X_1 zu parsen. Dabei zerlegen wir den String in $w = w_1 r_1$ so, dass $w_1 \in L(X_1)$ liegt. Dann versuchen wir, in dem Rest-String r_1 ein X_2 zu parsen und zerlegen dabei r_1 so, dass $r_1 = w_2 r_2$ mit $w_2 \in L(X_2)$ gilt. Zum Schluss haben wir dann den String w aufgespaltet in

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

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

$$A \to A\beta$$

hat, denn dann würden wir um ein A zu parsen sofort wieder rekursiv versuchen, ein A zu parsen und wären damit in einer Endlos-Schleife. Es gibt zwei Möglichkeiten, mit dem Problem umzugehen:

1. Wir können die Grammatik so umschreiben, dass sie danach nicht mehr links-rekursiv ist.
2. Wir können versuchen, den String *rückwärts* zu parsen, d.h. bei einer Regel der Form

   $$A \to X_1 X_2 \cdots X_n$$

   versuchen wir als erstes, ein X_n am Ende eines zu parsenden Strings w zu entdecken und arbeiten den String w dann von hinten ab.

Wir werden beide Möglichkeiten an Hand der Grammatik für arithmetische Ausdrücke diskutieren.

6.2.1 Umschreiben der Grammatik

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

$$\begin{array}{lcl} A & \to & A\beta \\ & \mid & \gamma \end{array}$$

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:

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

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

$$\begin{array}{lcl} A & \to & \gamma B \\ B & \to & \beta B \\ & \mid & \varepsilon \end{array}$$

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

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

Folglich beschreibt das Nicht-Terminal B die Sprache

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

Damit ist klar, dass auch mit der oben angegeben Grammatik

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

gilt. Um die Links-Rekursion aus der in Abbildung 6.4 auf Seite 73 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:

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

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

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

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

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

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \text{“}\texttt{+}\text{”}\ \mathit{Product} \\
& | & \mathit{Expr}\ \text{“}\texttt{-}\text{”}\ \mathit{Product} \\
& | & \mathit{Product} \\
\mathit{Product} & \rightarrow & \mathit{Product}\ \text{“}\texttt{*}\text{”}\ \mathit{Factor} \\
& | & \mathit{Product}\ \text{“}\texttt{/}\text{”}\ \mathit{Factor} \\
& | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \text{“}\texttt{(}\text{”}\ \mathit{Expr}\ \text{“}\texttt{)}\text{”} \\
& | & \mathit{Number}
\end{array}
$$

Abbildung 6.4: Links-rekursive Grammatik für arithmetische Ausdrücke.

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

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

   wobei $\mathit{op} \in \{$ “+” , “-” $\}$ gilt.

2. *ProductRest* beschreibt eine Liste der Form

op Factor $\cdots$ *op Factor*,

wobei *op* $\in$ { “`*`” , “`/`” } gilt.

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*

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

Aufgabe 10:

1. Geben Sie eine Grammatik für die in Kapitel 2 definierten regulären Ausdrücke an. Die Grammatik soll die im Kapitel 2 definierten Präzedenzen der Operatoren, “`.`”, “`|`” und “`*`” wiederspiegeln.

2. Entfernen Sie die Links-Rekursion aus dieser 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 einen String aus Terminalen zu verwandeln. In userem Fal sind die Terminale die Operator-Symbole, die beiden Klammern und die Zahlen. Wir werden diesen Scanner mit Hilfe von *JavaCC* entwickeln. Abbildung 6.6 zeigt eine *JavaCC*-Eingabe-Spezifikation, die wir jetzt diskutieren.

1. Zunächst erzeugen wir in den Zeilen 8 und 9 einen Token-Manager, der von der Standard-Eingabe liest.

2. Die gelesenen Token sammeln wir in der Liste `tokenList` auf, die wir in Zeile 10 definieren.

3. Diese Liste wird dann in der Schleife in den Zeilen 12 – 15 gefüllt.

4. Anschließend erzeugen wir in Zeile 16 den eigentlichen Parser, der die auszuwertende Tokenliste als Argument übergeben bekommt.

5. Der Aufruf der Methode `parseExpr`() in Zeile 17 startet dann den Parser, der den erzeugten Ausdruck zusätzlich auswertet.

```
1  PARSER_BEGIN(RDParser)
2
3  import java.util.*;
4
5  public class RDParser {
6
7      public static void main(String args[]) throws ParseException, ParseError {
8          SimpleCharStream     stream    = new SimpleCharStream(System.in);
9          RDParserTokenManager manager   = new RDParserTokenManager(stream);
10         ArrayList<Token>     tokenList = new ArrayList<Token>();
11         Token token;
12         do {
13             token = manager.getNextToken();
14             tokenList.add(token);
15         } while (token.kind != RDParserConstants.EOF); // end of file
16         ExprParser parser = new ExprParser(tokenList);
17         double     result = parser.parseExpr();
18         System.out.println("result: " + result);
19     }
20 }
21 PARSER_END(RDParser)
22
23 // forget about white space
24 SKIP: { " " | "\t" | "\r" | "\n" }
25
26 TOKEN: {
27     <NUMBER:   ("0" | ["1"-"9"] (["0"-"9"])*) ("." (["0"-"9"])*)?>
28   | <OPERATOR: "+" | "-" | "*" | "/" | "(" | ")">
29 }
```

Abbildung 6.6: *JavaCC*-Spezifikation des Scanners.

6. In Zeile 24 spezifizieren wir über das Schlüsselwort `SKIP` diejenigen Token, die vom Scanner aus der Eingabe entfernt werden sollen. Konkret eliminieren wir Leerzeichen, Tabulatoren, Wagenrückläufe und Zeilenumbrüche.

7. Schließlich werden in den Zeilen 26 – 29 die Token definiert, die von Scanner zurück gegeben werden.

   (a) `<NUMBER>` steht für ein Gleitkommazahl.

   (b) `<OPERATOR>` spezifiziert Operator- und Klammer-Symbole.

   JavaCC erzeugt eine Datei mit dem Namen `RDParserConstants.java`, in der Konstanten für die verschiedenen Token-Arten definiert werden. Diese Datei wir in Abbildung 6.7 gezeigt. Ob ein gegebenes Token t eine Zahl ist können wir dann beispielsweise durch den Ausdruck

   t`.kind == RDParserConstants.Number`

   testen.

Jetzt können wir uns dem eigentlichen Parser zuwenden, dessen Implementierung in den Abbildungen 6.8 und 6.9 auf den Seiten 77 und 78 gezeigt ist.

1. Zunächst definieren wir in den Zeilen 3 und 4 die Member-Variablen `mTokenList` und `mIndex`. Die Variable `mTokenList` ist eine Liste aller Token, die der Scanner gefunden hat. Diese Liste

```
1  public interface RDParserConstants {
2      int EOF      = 0;
3      int NUMBER   = 5;
4      int OPERATOR = 6;
5      int DEFAULT  = 0;
6
7      String[] tokenImage = {
8          "<EOF>",
9          "\" \"",
10         "\"\\t\"",
11         "\"\\r\"",
12         "\"\\n\"",
13         "<NUMBER>",
14         "<OPERATOR>",
15     };
16 }
```

Abbildung 6.7: Die Datei `RDParserConstants.java`.

wird dem Konstruktor der Klasse `ExprParser` als Argument übergeben. Die Variable `mIndex` zeigt auf das erste Token in dieser Liste, das vom Parser noch nicht verbraucht worden ist. Diese Variable wird daher im Konstruktor mit 0 initialisiert.

Anschließend haben wir für jedes Nicht-Terminal der Grammatik eine Methode, die versucht, dieses Nicht-Terminal zu parsen und dabei den Wert zurück gibt.

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

   $$\mathit{Expr} \rightarrow \mathit{Product}\ \mathit{ExprRest}.$$

   Die Umsetzung ist wie folgt: Um eine *Expr* zu erkennen, versuchen wir als erstes mit dem Aufruf *parseProduct*() ein Produkt zu erkennen. Ist dies erfolgreich so wird als nächstes versucht, einen *ExprRest* zu parsen, also eine Liste der Form

   $$\mathit{op}\ \mathit{Product}\ \cdots \mathit{op}\ \mathit{Product} \quad \text{mit } \mathit{op} \in \{\ \text{“+”}\ ,\ \text{“-”}\ \}.$$

3. Um einen *ExprRest* zu erkennen überprüfen wir mit der Methode *check*(), ob das nächste Zeichen das Symbol “+” oder “-” ist. Falls dem so ist, versuchen wir anschließend ein Produkt zu parsen, sowie einen eventullen weiteren *ExprRest*, der auf dieses Produkt folgt. Dadurch setzen wir effektiv die folgenden Grammatik-Regeln um:

   $$\begin{array}{rcl} \mathit{ExprRest} & \rightarrow & \text{“+”}\ \mathit{Product}\ \ \mathit{ExprRest} \\ & | & \text{“-”}\ \mathit{Product}\ \ \mathit{ExprRest} \\ & | & \varepsilon \end{array}$$

   Es ist wichtig zu verstehen, wie in diesem Fall der Wert berechnet wird. Ein *ExprRest* wird genau dann geparst, wenn vorher bereits ein *Product* geparst worden ist. Der dabei erhaltene Wert wird der Methode *parseExprRest*() als Argument mitgegeben. Wird anschließend beispielsweise das Zeichen “+” gefolgt von einem *Product* gelesen, so wird der Wert dieses Produktes zu dem Wert des ersten Produktes addiert und dann werden, ausgehend von dieser Summe, durch den rekursiven Aufruf von *parseExprRest*() weitere Produkte hinzuaddiert.

4. Die restlichen Grammatik-Regeln werden nun in analoger Weise umgesetzt.

5. Die Methode *check* überprüft, ob das als Argument übergebene Token das nächste Token in der Tokenliste ist. Dazu wird zuerst geprüft, ob überhaupt noch Token da sind, oder ob der

```
1  public class ExprParser {
2      private ArrayList<Token> mTokenList; // list of all tokens
3      private int              mIndex;     // index to the next token
4
5      public ExprParser(ArrayList<Token> tokenList) {
6          mTokenList = tokenList;
7          mIndex     = 0;
8      }
9      public double parseExpr() throws ParseError {
10         double product = parseProduct();
11         double result  = parseExprRest(product);
12         if (mTokenList.get(mIndex).image.equals(")")) {
13             return result;
14         }
15         if (mTokenList.get(mIndex).kind != RDParserConstants.EOF) {
16             throw new ParseError("Parse Error");
17         }
18         return result;
19     }
20     private double parseExprRest(double sum) throws ParseError {
21         if (check("+")) {
22             double product = parseProduct();
23             return parseExprRest(sum + product);
24         }
25         if (check("-")) {
26             double product = parseProduct();
27             return parseExprRest(sum - product);
28         }
29         return sum;
30     }
31     private double parseProduct() throws ParseError {
32         double factor = parseFactor();
33         return parseProductRest(factor);
34     }
35     private double parseProductRest(double product) throws ParseError {
36         if (check("*")) {
37             double factor = parseFactor();
38             return parseProductRest(product * factor);
39         }
40         if (check("/")) {
41             double factor = parseFactor();
42             return parseProductRest(product / factor);
43         }
44         return product;
45     }
```

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

Index `mIndex` bereits auf das Ende der Token-Liste zeigt. Ansonsten holen wir mit

```
mTokenList.get(mIndex)
```

das Token, auf das der Index zeigt. Der String, der zu diesem Token korrespondiert, ist in der

```
46      private double parseFactor() throws ParseError {
47          if (check("(")) {
48              double expr = parseExpr();
49              if (check(")") == false) {
50                  throw new ParseError("Parse Error, ')' expected");
51              }
52              return expr;
53          }
54          return parseNumber();
55      }
56      private boolean check(String token) {
57          if (mIndex == mTokenList.size()) {
58              return false;
59          }
60          if (mTokenList.get(mIndex).image.equals(token)) {
61              ++mIndex;
62              return true;
63          }
64          return false;
65      }
66      private double parseNumber() throws ParseError {
67          Token token = mTokenList.get(mIndex);
68          if (token.kind != RDParserConstants.NUMBER) {
69              throw new ParseError("Parse Error, number expected");
70          }
71          ++mIndex;
72          return new Double(token.image);
73      }
74  }
```

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

Member-Variablen `image` gespeichert. Wir vergleichen diesen String mit dem übergebenen Argument. Ist dieser Vergleich erfolgreich, so haben wir das gesuchte Token gefunden und inkrementieren den Index.

6. Die Methode *parseNumber* überprüft, ob das nächste Token in der Tokenliste vom Typ `NUMBER` ist. Falls ja, wird der diesem Token zugeordnete Wert als Zahl vom Typ `Double` zurück gegeben und der Index wird erhöht. Andernfalls wird eine Ausnahme geworfen.

Zum Abschluss zeigt die Abbildung 6.10 die Definition der Klasse `ParseError`.

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

Abbildung 6.10: Die Klasse `ParseError`.

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.

6.2.3 Implementierung eines rückwärts arbeitenden Top-Down-Parser

Wir zeigen nun, wir 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.

```
1  import java.util.*;
2
3  public class ExprParser {
4      private ArrayList<Token> mTokenList; // list of all tokens
5      private int              mIndex;     // index to the next token
6
7      public ExprParser(ArrayList<Token> tokenList) {
8          mTokenList = tokenList;
9          mIndex     = mTokenList.size() - 2;
10     }
11     public double parseExpr() throws ParseError {
12         double product = parseProduct();
13         if (check("+")) {
14             double expression = parseExpr();
15             return expression + product;
16         }
17         if (check("-")) {
18             double expression = parseExpr();
19             return expression - product;
20         }
21         return product;
22     }
23     private double parseProduct() throws ParseError {
24         double factor = parseFactor();
25         if (check("*")) {
26             double product = parseProduct();
27             return product * factor;
28         }
29         if (check("/")) {
30             double product = parseProduct();
31             return product / factor;
32         }
33         return factor;
34     }
35     private double parseFactor() throws ParseError {
36         if (check(")")) {
37             double expression = parseExpr();
38             if (!check("(")) {
39                 throw new ParseError("Parse Error, '(' expected");
40             }
41             return expression;
42         }
43         return parseNumber();
44     }
```

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

```
45      private boolean check(String token) {
46          if (mIndex < 0) {
47              return false;  // all tokens consumed
48          }
49          if (mTokenList.get(mIndex).image.equals(token)) {
50              --mIndex;
51              return true;
52          }
53          return false;
54      }
55      private double parseNumber() throws ParseError {
56          Token token = mTokenList.get(mIndex);
57          if (token.kind != RDParserConstants.NUMBER) {
58              throw new ParseError("Parse Error, number expected");
59          }
60          --mIndex;
61          return new Double(token.image);
62      }
63  }
```

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

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, Letzterer 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 nun so initialisiert, dass sie auf das letzte gelesen Token zeigt. Da der Scanner hinter die gelesenen Token noch ein spezielles `EOF`-Token, welches das Ende der Eingabe signalisiert, anhängt, hat der Index des letzten gelesenen Tokens den Wert `mTokenList.size()` − 2.

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. Abhängig davon, ob wir ein “+”- oder “-”-Zeichen gelsen 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.

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

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

5. Die Methode *check*() ändert sich gegenüber der ursprünglichen Implementierung, da die Token-Liste ja jetzt rückwärts abgearbeitet wird. Deswegen testen wir nun in Zeile 46, ob `mIndex` negativ ist, denn dann hätten wir alle Token abgearbeitet. Andernfalls vergleichen wir das durch `mIndex` spezifizierte Token mit dem String, der als Argument übergeben wird. Falls dieser Vergleich erfolgreich ist, wird der Zeiger `mIndex` nun dekrementiert.

6. Die Methode *parseNumber*() überprüft, ob das Token, das an der durch den Zeiger `mIndex` spezifizierten Position steht, eine Zahl ist und gibt diese gegebenenfalls zurück.

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

Kapitel 7

Parser-Generatoren

In diesem Kapitel werden wir zwei verschiedene Werkzeuge vorstellen, mit deren Hilfe Parser automatisch erzeugt werden können. Wir beginnen mit dem Parser-Generator *JavaCC*, mit dem wir Parser in der Sprache *Java* erstellen können. Im zweiten Abschnitt zeigen wir dann, wie mit Hilfe von *Bison* Parser in der Sprache `C` erzeugt werden.

7.1 *JavaCC*

In diesem Abschnitt zeigen wir, wie mit dem Parser-Generator *JavaCC* ein Parser in der Sprache *Java* erzeugt werden kann. Das Werkzeug *JavaCC* erzeugt einen *Top-Down-Parser*, der in etwa die selbe Struktur hat wie die Top-Down-Parser, die wir im letzten Kapitel vorgestellt haben. Damit haben die von *JavaCC* gegenüber den von *Bison* erzeugten Parser den Vorteil, dass sie im Prinzip lesbar und wartbar sind. Die von *Bison* erzeugten Parser basieren auf einer Tabelle und sind auch für das geübte Auge unlesbar.

Wir diskutieren zunächst ein kleineres Beispiel, die Auswertung arithmetischer Ausdrücke und erläutern dabei die Struktur einer *JavaCC*-Eingabe-Spezifikation. Anschließend wenden wir uns dann einem Beispiel zu, das etwas komplexer ist: Wir zeigen, wie sich ein Interpreter für eine sehr einfache Programmiersprache mit Hilfe von *JavaCC* implementieren läßt.

7.1.1 Ein einfaches Beispiel

Wir greifen in diesem Unterabschnitt das Parsen arithmetischer Ausdrücke wieder auf. Die Parser, die von *JavaCC* generiert werden, sind Top-Down-Parser. Daher kann *JavaCC* nur für solche Grammatiken verwendet werden, die keine Links-Rekursion enthalten. Wir starten deshalb mit der in Abbildung 7.1 auf Seite 83 gezeigten Grammatik und zeigen, wie sich für diese Grammatik mit Hilfe von *JavaCC* ein Parser erstellen läßt. Diese Grammatik ist fast identisch zu der in Abbildung 6.5 auf Seite 74 gezeigten Grammatik, der einzige Unterschied ist der, dass es für das Nicht-Terminal *factor* die zusätzliche Regel

$$\mathit{factor} \rightarrow \mathit{IDENTIFIER}$$

gibt, die die Benutzung von Variablen in arithmetischen Ausdrücken ermöglicht. Um das Beispiel etwas abwechselungsreicher zu machen, ist unser Ziel dieses Mal nicht die Auswertung eines arithmetischen Ausdrucks, sondern wir wollen statt dessen einen arithmetischen Ausdruck symbolisch nach einer gegeben Variablen ableiten. Daher müssen wir nun in unserem Parser einen abstrakten Syntax-Baum aufbauen, der den gelesenen arithmetischen Ausdruck repräsentiert. Zu diesem Zweck definieren wir zunächst eine abstrakte Klasse `Expr`, deren Implementierung in der Abbildung 7.2 auf Seite 83 gezeigt wird.

Die Klasse `Expr` deklariert die abstrakte Methode *diff*(), mit der wir einen Ausdruck nach einer Variablen differenzieren können, die als Argument übergeben wird. Von der Klasse `Expr` leiten wir

$$
\begin{array}{rcl}
\mathit{expr} & \rightarrow & \mathit{product}\ \ \mathit{exprRest} \\
\mathit{exprRest} & \rightarrow & \text{“}\texttt{+}\text{”}\ \ \mathit{product}\ \ \mathit{exprRest} \\
 & | & \text{“}\texttt{-}\text{”}\ \ \mathit{product}\ \ \mathit{exprRest} \\
 & | & \varepsilon \\
\mathit{product} & \rightarrow & \mathit{factor}\ \ \mathit{productRest} \\
\mathit{productRest} & \rightarrow & \text{“}\texttt{*}\text{”}\ \ \mathit{factor}\ \ \mathit{productRest} \\
 & | & \text{“}\texttt{/}\text{”}\ \ \mathit{factor}\ \ \mathit{productRest} \\
 & | & \varepsilon \\
\mathit{factor} & \rightarrow & \text{“}\texttt{(}\text{”}\ \ \mathit{expr}\ \ \text{“}\texttt{)}\text{”} \\
 & | & \mathit{NUMBER} \\
 & | & \mathit{IDENTIFIER}
\end{array}
$$

Abbildung 7.1: Grammatik für arithmetische Ausdrücke.

```
1  public abstract class Expr {
2      public abstract Expr diff(Variable x);
3  }
```

Abbildung 7.2: Die abstrakte Klasse `Expr`.

konkrete Klassen ab, die die einzelnen Knoten in einem Parse-Baum repräsentieren. Insgesamt haben wir die folgenden abgeleiteten Klassen:

1. `Sum` repräsentiert Ausdrücke der Form $l + r$.
2. `Difference` repräsentiert Ausdrücke der Form $l - r$.
3. `Product` repräsentiert Ausdrücke der Form $l * r$.
4. `Quotient` repräsentiert Ausdrücke der Form l/r.
5. `Variable` repräsentiert Variablen.
6. `MyNumber` repräsentiert Zahlen.

Wir diskutieren exemplarisch die Implementierungen der Klassen `Product` und `MyNumber`.

Abbildung 7.3 zeigt die Implementierung der Klasse `Product`. Die Klasse hat zwei Member-Variablen *mLhs* und *mRhs*, die beide arithmetische Ausdrücke repräsentieren. Die Klasse `Product` repräsentiert dann das Produkt

$$\mathit{mLhs} * \mathit{mRhs}.$$

Neben dem Konstruktor, der diese Variablen initialisiert, enthält die Klasse noch zwei Methoden:

1. Die Methode *diff*(x) berechnet die Ableitung des Produktes $\mathit{mLhs} \cdot \mathit{mRhs}$ nach der Produkt-Regel
$$\frac{d}{d\,x}\big(\mathit{mLhs} \cdot \mathit{mRhs}\big) = \big(\frac{d}{d\,x}\mathit{mLhs}\big) \cdot \mathit{mRhs} + \mathit{mLhs} \cdot \big(\frac{d}{d\,x}\mathit{mRhs}\big).$$
2. Die Methode *toString*() wandelt den arithmetischen Ausdruck in einen String um, damit wir das Ergebnis unserer Rechnung auch ausgeben können. Hier ist darauf zu achten, dass Klammern zu setzen sind, denn schließlich könnten die Member-Variablen *mLhs* und *mRhs*

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

Abbildung 7.3: Die Klasse `Product`.

ja Summen, zum Beispiel $1 + 2$ und $3 + 4$, darstellen und dieses Produkt müssen wir als

$$(1 + 2) * (3 + 4) \quad \text{und nicht als} \quad 1 + 2 * 3 + 4$$

ausgeben.

Die Klassen `Sum`, `Difference` und `Quotient` sind analog zu der Klasse `Product` aufgebaut und werden daher nicht diskutiert. Der Vollständigkeit halber zeigen wir die Implementierung dieser Klassen in den Abbildungen 7.4, 7.5 und 7.6.

```
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 Sum diff(Variable 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 7.4: Die Klasse `Sum`.

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

Abbildung 7.5: Die Klasse `Difference`.

```
1   public class Quotient extends Expr {
2       Expr mLhs;
3       Expr mRhs;
4
5       public Quotient(Expr lhs, Expr rhs) {
6           mLhs = lhs;
7           mRhs = rhs;
8       }
9       public Quotient diff(Variable x) {
10          return new Quotient(new Difference(new Product(mLhs.diff(x), mRhs),
11                                             new Product(mLhs, mRhs.diff(x))),
12                              new Product(mRhs, mRhs));
13      }
14      public String toString() {
15          return "(" + mLhs.toString() + ") / (" + mRhs.toString() + ")";
16      }
17  }
```

Abbildung 7.6: Die Klasse `Quotient`.

Abbildung 7.7 zeigt die Implementierung der Klasse `Variable`. Der Name der Variablen wird in der Member-Variablen *mName* gespeichert. Bei der Implementierung der Methode *diff*() überprüfen wir, ob der Name der Variablen, nach der differenziert wird, identisch ist mit dem Namen der Variablen, die differenziert wird. Falls dies so ist, ist das Ergebnis die Zahl 1, andernfalls ist das Ergebnis die Zahl 0. Abbildung 7.8 zeigt die Implementierung der Klasse `MyNumber`. Diese Klasse ist analog zu der Klasse `Variable` aufgebaut und wird daher nicht weiter diskutiert.

```
1  public class Variable extends Expr {
2      String mName;
3
4      public Variable(String name) {
5          mName = name;
6      }
7      public Expr diff(Variable x) {
8          if (mName.equals(x.mName)) {
9              return new MyNumber("1");
10         }
11         return new MyNumber("0");
12     }
13     public String toString() {
14         return mName;
15     }
16 }
```

Abbildung 7.7: Die Klasse `Variable`.

```
1  public class MyNumber extends Expr {
2      Double mValue;
3
4      public MyNumber(String value) {
5          mValue = new Double(value);
6      }
7      public Expr diff(Variable x) {
8          return new MyNumber("0");
9      }
10     public String toString() {
11         return mValue.toString();
12     }
13 }
```

Abbildung 7.8: Die Klasse `MyNumber`.

Abbildung 7.9 auf Seite 87 zeigt die *JavaCC*-Eingabe-Spezifikation, mit der wir einen Parser für arithmetische Ausdrücke erzeugen kann. Eine *JavaCC*-Eingabe-Spezifikation besteht aus drei Teilen.

1. Der erste Teil ist die *Java*-Übersetzungs-Einheit. Dieser Teil definiert die Klasse, die hinterher den erzeugten Parser enthält. In der Abbildung erstreckt sich dieser Teil von Zeile 1 bis zur Zeile 11. Syntaktisch wird dieser Teil von den Schlüsselwörtern `PARSER_BEGIN` und `PARSER_END` eingeklammert.

   Die *Java*-Übersetzungs-Einheit enthält `import`-Direktiven und die Definition einer Klasse, die den selben Namen haben muss, wie die Datei. Dieser Name muss auch hinter den Schlüsselwörtern `PARSER_BEGIN` und `PARSER_END` angegeben werden.

   In dem obigen Beispiel enthält die Klasse nur die Methode *main*(). In der später generierten *Java*-Datei finden sich dann hinterher noch ein Konstruktor, der mit einem Objekt der Klasse

   `java.io.InputStream` oder der Klasse `CharStream`

   aufgerufen werden kann. Dieser Konstruktor wird in Zeile 4 benutzt um ein Objekt der

```
1  PARSER_BEGIN(ExprParser)
2  public class ExprParser {
3      public static void main(String args[]) throws ParseException {
4          ExprParser parser = new ExprParser(System.in);
5          Expr       expr   = parser.expr();
6          Variable   x      = new Variable("x");
7          Expr       deriv  = expr.diff(x);
8          System.out.println(deriv);
9      }
10 }
11 PARSER_END(ExprParser)
12
13 Expr expr() : {
14   Expr p, r;
15 } {
16   p = product() r = exprRest(p) { return r; }
17 }
18 Expr exprRest(Expr s) : {
19   Expr p, a, r;
20 } {
21   "+" p=product() { a = new Sum       (s, p); } r=exprRest(a) { return r; }
22 | "-" p=product() { a = new Difference(s, p); } r=exprRest(a) { return r; }
23 | /* epsilon */                                               { return s; }
24 }
25 Expr product() : {
26   Expr f, r;
27 } {
28     f = factor() r = productRest(f) { return r; }
29 }
30 Expr productRest(Expr p) : {
31   Expr f, a, r;
32 } {
33   "*" f=factor() { a = new Product (p, f); } r=productRest(a) { return r; }
34 | "/" f=factor() { a = new Quotient(p, f); } r=productRest(a) { return r; }
35 | /* epsilon */                                               { return p; }
36 }
37 Expr factor() : {
38   Expr  r; Token t;
39 } {
40   "(" r = expr() ")" { return r; }
41 | t = <NUMBER>       { return new MyNumber(t.image); }
42 | t = <IDENTIFIER>   { return new Variable(t.image); }
43 }
44 SKIP: { " " | "\t" | "\r" | "\n" }
45 TOKEN: {
46     <NUMBER:     ("0" | ["1"-"9"] (["0"-"9"])*) ("." (["0"-"9"])*)?>
47   | <IDENTIFIER: (["a"-"z", "A"-"Z"])+>
48 }
```

Abbildung 7.9: Die *JavaCC*-Eingabe-Spezifikation

Klasse `ExprParser` zu erzeugen. Neben dem Konstruktor erzeugt *JavaCC* in der generierten *Java*-Datei für jedes Nicht-Terminal eine Methode, die die Aufgabe hat, das entsprechende Nicht-Terminal zu parsen. Die in Abbildung 7.1 auf Seite 83 gezeigte und in Abbildung 7.9 implementierte Grammatik hat das Start-Symbol *expr*. Daher rufen wir in Zeile 5 die Methode *expr*() auf, um einen arithmetischen Ausdruck zu parsen. Diese Methode liefert als Ergebnis den abstrakten Syntax-Baum zurück, der durch ein Objekt der Klasse `Expr` repräsentiert wird. Durch den Aufruf der Methode *diff*() wird dieser Ausdruck in Zeile 7 nach der Variablen "`x`" differenziert und das dabei erhaltene Ergebnis wird ausgegeben.

2. Der zweite Teil der *JavaCC*-Eingabe-Spezifikation enthält die Grammatik-Regeln, die allerdings noch mit *semantischen Aktionen* angereichert werden können. In der Abbildung 7.9 erstreckt sich dieser Teil von Zeile 13 bis Zeile 43. Die Zeilen 13 bis 17 zeigen die Implementierung der Grammatik-Regel

   $$expr \rightarrow product \;\; exprRest.$$

   Generell hat eine Grammatik-Regel die Form

   ReturnType name "`(`" *ParameterList* "`)`" "`:`" "`{`"
   JavaBlock
   "`}`" "`{`"
   ProductionList
   "`}`"

   Die einzelnen Komponenten haben dabei die folgende Bedeutung:

   (a) *ReturnType* ist der Rückgabe-Typ der Methode *name*().
   In der Zeile 13 ist der Rückgabe-Typ die Klasse `Expr`, den jeder Aufruf der Methode *expr*() gibt als Ergebnis den abstrakten Syntax-Baum des gelesenen arithmetischen Ausdrucks zurück, und diese Syntax-Bäume wollen wir ja durch die Klasse *Expr* darstellen.

   (b) *name* ist der Name des Nicht-Terminals, das durch diese Regel definiert wird.

   (c) *ParameterList* ist eine Liste von Parametern, die zusammen mir ihrem Typ definier werden, genau wie bei der Parameter-Liste einer *Java*-Methode. In Zeile 13 ist diese Liste leer, aber Zeile 18 zeigt eine Grammatik-Regel, bei der die Parameter-Liste den Wert "`Expr s`" hat.

   (d) *JavaBlock* ist ein Block, der Deklarationen von Variablen enthalten kann. Zusätzlich kann dort auch noch *Java*-Code stehen, mit dem diese Variablen initialisiert werden. In den Zeilen 13 – 15 enthält dieser Block die Deklaration der Variablen `p` und `r`.
   Wenn keine Variablen zu deklarieren sind, darf der *JavaBlock* auch leer sein, aber die gesschweiften Klammern müssen in jedem Fall vorhanden sein.

   (e) *ProductionList* ist eine Liste der verschiedenen *Produktionen*, mir denen das Nicht-Terminal *name* abgeleitetet werden kann. Unter einer *Produktion* verstehen wir hier einfach die rechte Seite einer Grammatik-Regel. Die einzelnen Elemente der Liste von Produktionen werden durch den Operator "`|`" getrennt. In dem Beispiel in 16 gibt es nur eine Produktion, daher wird dort der Operator "`|`" nicht verwendet. Die Struktur einer Produktion wird weiter unten erläutert. In den Zeilen 18 – 24 werden die Grammatik-Regeln

   $$\begin{array}{lcl} exprRest & \rightarrow & \text{"+"}\; product \;\; exprRest \\ & | & \text{"-"}\; product \;\; exprRest \\ & | & \varepsilon \end{array}$$

   implementiert. Hier haben wir insgesamt drei Produktionen.

Eine Produktion ist eine Liste aus sogenannten *Rumpf-Einheiten*. In dieser Liste wird kein Trennzeichen verwendet, die einzelnen Rumpf-Einheiten werden einfach hintereinander geschrieben. Wir diskutieren vier verschiedene Arten von Rumpf-Einheiten:

(a) *Aufruf eines Nicht-Terminals*, wobei das Ergebnis des Aufrufs optional noch einer Variablen zugewiesen werden kann, die dann aber vorher in dem *JavaBlock* deklariert sein muss. Ein Beispiel eines solchen Aufrufs finden wir in Zeile 16:

```
p = product()
```

Bei dem Aufruf können der Methode, die das Nicht-Terminal parst, auch Variablen übergeben werden. In Zeile 16 gibt es dazu das Beispiel

```
r = exprRest(p).
```

(b) *Semantische Aktionen* sind Jave-Code, der in gesschweiften Klammern eingefasst ist. Solche semantischen Aktionen werden beispielsweise benötigt, wenn die Methode, die ein Nicht-Terminal parst, einen abstrakten Syntax-Baum als Ergebnis zurück geben soll.

(c) *Wörtlich spezifizierte Terminale* sind Terminale, die in doppelten Anführungsstrichen angegeben werden. Beispielsweise wird in Zeile 21 das Terminal "`+`" wörtlich spezifiziert.

(d) *Token-Kategorien* beschreiben Klassen von Terminalen. Diese Token-Kategorien müssen in den spitzen Klammern "`<`" und "`>`" eingeschlossen werden und bestehen per Konvention nur aus Groß-Buchstaben. Außerdem müssen die verwendeten Token-Kategorien im dritten Teil der *JavaCC*-Eingabe-Spezifikation definiert werden.

In Zeile 41 verwenden wir beispielsweise die Token-Kategorie "`NUMBER`", die weiter unten in Zeile 46 definiert wird.

3. Der dritte Teil der *JavaCC*-Eingabe-Spezifikation definiert den Scanner. In Abbildung 7.9 erstreckt sich dieser Teil von Zeile 44 – 48.

(a) In dem Beispiel haben wir in Zeile 44 festgelegt, dass Leerzeichen, Tabulatoren, Wagenrückläufe und Zeilenumbrüche vom Scanner überlesen werden sollen.

(b) In Zeile 46 spezifizieren wir das Terminal `NUMBER` zur Erkennung von Fließkomma-Zahlen.

(c) In Zeile 47 spezifizieren wir das Terminal `IDENTIFIER` zur Erkennung von Variablen-Namen.

(d) Bemerkenswert ist, dass wir für die verwendeten Operator-Symbole wie "`+`" und "`-`" kein zusätzliches Terminal mehr spezifizieren müssen. Für den Scanner reicht es aus, dass diese Operator-Symbole als wörtlich spezifizierte Terminale in den Grammatik-Regeln auftreten.

Um zu verstehen, wie *JavaCC* aus der obigen Eingabe-Spezifikation eine Parser erzeugt, werfen wir einen Blick auf die von *JavaCC* erzeugte Klasse `ExprParser`. Konkret betrachten wir, wie die Zeilen

```
17  Expr exprRest(Expr s) : { Expr p, a, r; } {
18        "+" p = product() { a = new Sum       (s, p); } r = exprRest(a)
19                { return r; }
20      | "-" p = product() { a = new Difference(s, p); } r = exprRest(a)
21                { return r; }
22      | /* epsilon */ { return s; }
23  }
```

```
1  static final public Expr exprRest(Expr s) throws ParseException {
2      Expr p, a, r;
3      switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
4      case 1:
5          jj_consume_token(1);
6          p = product();
7          a = new Sum(s, p);
8          r = exprRest(a);
9          return r;
10         break;
11     case 2:
12         jj_consume_token(2);
13         p = product();
14         a = new Difference(s, p);
15         r = exprRest(a);
16         return r;
17         break;
18     default:
19         return s;
20     }
21 }
```

Abbildung 7.10: Die Implementierung der Methode *exprRest*().

der Eingabe-Spezifikation umgesetzt werden. *JavaCC* generiert aus diesen Zeilen in etwa die in Abbildung 7.10 gezeigte Methode, die wir jetzt im Detail diskutieren.

1. Da hier ein *exprRest* geparst werden soll, wurde bereits vorher ein (oder auch mehrere) Produkte geparst, deren Summe gespeichert sind in dem Parameter `s`, der als Argument übergeben wird.

2. Nach der Deklaration der Variablen enthält diese Methode eine `switch`-Anweisung, die von dem Ausdruck

   ```
   (jj_ntk==-1)?jj_ntk():jj_ntk
   ```

   gesteuert wird. Die Variable `jj_ntk` (gelesen: *jj next token*) enthält eine ganze Zahl, die das nächste Token spezifiziert. Diese Zahl hat den Wert -1, wenn das nächste Token noch nicht gelesen worden ist. In diesem Fall liest der Aufruf der Methode `jj_ntk`() das nächste Token und liefert eine Zahl zurück, die dieses Token eindeutig spezifiziert. Dem Token "`+`" wird dabei die Zahl 1 zugeordnet, dem Token "`-`" wird die Zahl 2 zugeordnet.

   Ist also ein Token vorhanden, so wird die `switch`-Anweisung durch dieses Token gesteuert, andernfalls wird das nächste Token gelesen und dieses Token steuert dann die `switch`-Anweisung.

3. Falls das Token "`+`" gelesen wurde, so wird dieses Token durch den Aufruf von `jj_consume_token` in Zeile 5 gefressen. Anschließend versucht der Parser durch den Aufruf von *product*() in Zeile 6 ein Produkt zu parsen. Der dabei produzierte abstrakte Syntax-Baum wird in der Variablen `p` gespeichert. Dann wird in Zeile 7 ein Objekt vom Typ `Sum` erzeugt, das den Ausdruck `s + p` repräsentiert, das eben geparste Produkt wir also zu der Summe der bisher geparsten Produkte hinzuaddiert. Mit dieser neuen Summe `a` wird nun die Methode *exprRest*() rekursiv aufgerufen und das dabei erhaltene Ergebnis wird zurück gegeben.

4. Der Fall, dass das Token "-" gelesen wurde, wird in analoger Weise in den Zeilen 11 – 17 behandelt.

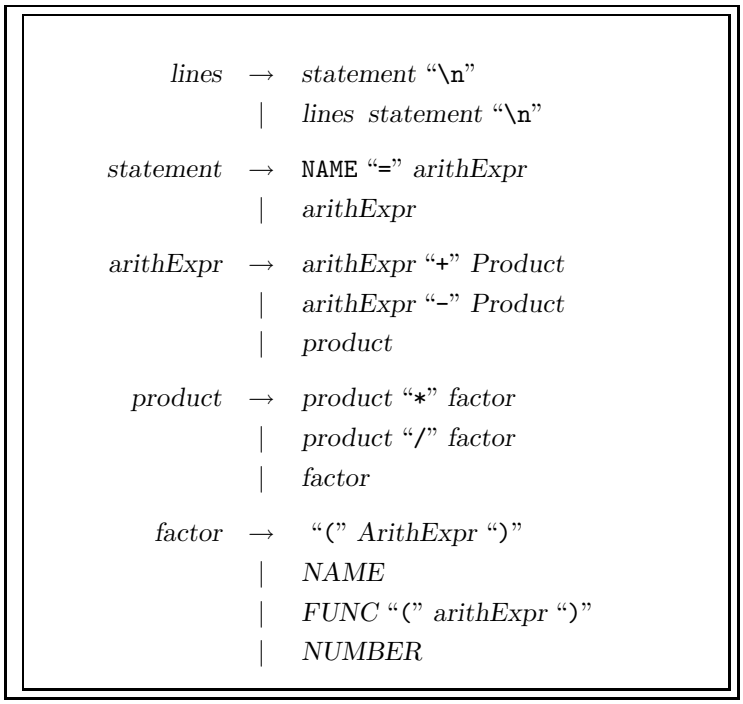
5. Fall weder ein "+" noch ein "-" gelesen vorhanden ist, wird die Summe der bisher gelesenen Produkte als Ergebnis zurück gegeben.

Wir haben jetzt gesehen, dass der von *JavaCC* automatisch erzeugte Parser in etwa die selbe Struktur hat wie der Top-Down-Parser für arithmetische Ausdrücke, den wir im letzten Kapitel von Hand erzeugt haben. Die *JavaCC*-Eingabe-Spezifikation ist aber wesentlich kompakter und übersichtlicher als das *Java*-Programm, das wir im Kapitel 6 entwickelt haben. Dies zeigt die Nützlichkeit von *JavaCC*.

7.1.2 Implementierung eines Interpreters

7.2 *Bison*

Unser Ziel in diesem Abschnitt ist es, für die in Abbildung 7.11 gezeigte Grammatik einen Parser zu erstellen.



lines	→	*statement* "`\n`"
	|	*lines statement* "`\n`"
statement	→	`NAME` "`=`" *arithExpr*
	|	*arithExpr*
arithExpr	→	*arithExpr* "`+`" *Product*
	|	*arithExpr* "`-`" *Product*
	|	*product*
product	→	*product* "`*`" *factor*
	|	*product* "`/`" *factor*
	|	*factor*
factor	→	"`(`" *ArithExpr* "`)`"
	|	*NAME*
	|	*FUNC* "`(`" *arithExpr* "`)`"
	|	*NUMBER*

Abbildung 7.11: Eine Grammatik für den Taschenrechner.

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.

7.2.1 Der Scanner

Bevor mit der Entwicklung des Parsers beginnen können, benötigen wir einen Scanner, der die eingabe in unterschiedliche Token zerlegt. Abbildung 7.12 zeigt die *Flex*-Spezifikation eines solchen Scanners. 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.

```
1  %{
2  #include "calc.h"
3  #include "calc.tab.h"
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]                  { /* 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 7.12: Der Scanner für den Taschenrechner.

Wir diskutieren nun die Details der Abbildung 7.12.

1. In den Zeilen 2 wird die Dateien "`calc.h`" eingebunden. Diese Datei enthält die Definition der Daten-Struktur `SymbolTable`, sowie die Deklaraton verschiedener Variablen und Konstanten. Abbildung 7.13 zeigt diese Datei.

2. Die in Zeile 3 eingebundene "`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 7.13 gezeigt.

   Wir werden die Dateien "`calc.h`" und "`calc.tab.h`" später im Detail diskutieren.

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 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. Es ist Aufgabe des Parsers, die entsprechenden Makros zu generieren. In der Datei "`calc.tab.h`" sind beispielsweise die Token `NAME`, `FUNC` und `NUMBER` als die Zahlen 257, 258 und 259 definiert.

5. Die Regel in Zeile 12 entfernt Leerzeichen und Tabulatoren. Da der Parser sich nicht für Leerzeichen und Tabulatoren interessiert, wird hier kein Token zurückgegeben, sondern der Scanner sucht stattdessen das nächste Token.

6. Die Regel in Zeile 13 erkennt Variablen und Funktions-Namen. Das Problem ist hier, dass sowohl Variablen als auch Funktions-Namen als aus Buchstaben und Ziffern bestehend definiert sind und damit lexikalisch gar nicht zu unterscheiden sind. 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*(n) überprüft für einen gegeben Namen n, ob zu diesem Namen in der Tabelle bereits ein Eintrag existiert. Wenn dies der Fall ist, wird ein Zeiger auf den Eintrag zurückgegeben, ansonsten liefert die Funktion einen 0-Zeiger als Ergebnis. Der Zeiger zeigt 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. Der erste Eintrag `name` gibt den Namen der Variablen oder der Funktion an. 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, andernfalls wurde eine Variable erkannt. Entsprechend wird entweder das Token `FUNC` oder das Token `NAME` zurückgegeben.

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, 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 sie im Scanner definiert. Die Variable `yytext` enthält den Text des zuletzt vom Scanner erkannten Tokens.

```
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 7.13: Die Header-Datei "`calc.h`".

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 Sysmbol-Tabelle. Diese Einträge sind entweder Variablen-Namen mit den der Variablen zugeordneten Werten `value` oder aber Namen von Funktionen mit einem Zeiger `funcPtr` auf diese Funktion.

2. Die eigentliche Sysmbol-Tabelle wird dann in Zeile 9 als Feld mit `NSYMS` Einträgen definiert. Die Zahl `NSYMS` ist vorher in Zeile 1 als 1000 definiert worden.

3. Schließlich weerden noch die beiden Funktionen *lookUpSymbol*() und *yyerror*() deklariert.

```
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 7.14: Die Datei "`calc.tab.h`".

Die Datei "`calc.tab.h`" wird automatisch von *Bison* erzeugt und 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 Sysmbol-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.

7.2.2 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 Strings "`%{`", "`%}`"

```
1  %{
2      Deklarations-Teil
3  %}
4      Definitions-Teil
5  %%
6      Grammatik-Regeln
7  %%
8      C-Funktionen
```

Abbildung 7.15: Struktur einer Grammatik-Spezifikation für *Bison*

und zweimal "`%%`" getrennt und haben die folgende Bedeutung:

1. Der *Deklarations-Teil* enthält Include-Direktiven Deklarationen von Funktionen und Variablen. Dieser Teil wird später wörtlich in den generierten `C`-Code 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.
4.

```
%{
#include "calc.h"
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
%}

%union {
    double       value;
    SymbolTable* symPtr;
}

%token <symPtr> NAME FUNC
%token <value>  NUMBER
%type <value> arithExpr product factor

%%
lines     : statement '\n'
          | lines statement '\n'
          ;

statement : NAME '=' arithExpr     { $1->value = $3;     }
          | arithExpr              { printf("%g\n", $1); }
          ;

arithExpr : arithExpr '+' product  { $$ = $1 + $3; }
          | arithExpr '-' product  { $$ = $1 - $3; }
          | product
          ;

product   : product '*' factor     { $$ = $1 * $3; }
          | product '/' factor     { $$ = $1 / $3; }
          | factor
          ;

factor    : '(' arithExpr ')'      { $$ = $2;                }
          | NAME                   { $$ = $1->value;         }
          | FUNC '(' arithExpr ')' { $$ = ($1->funcPtr)($3); }
          | NUMBER
          ;

%%
```

Abbildung 7.16: Grammatik-Spezifikation für den Taschenrechner, Teil 1.

```
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          // Is it already here?
49          if (symbolPtr->name && !strcmp(symbolPtr->name, symbol)) {
50              return symbolPtr;
51          }
52          // Is it free?
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      // extern double sqrt(), exp(), log();
70      addfunc("sqrt", sqrt);
71      addfunc("exp", exp);
72      addfunc("log", log);
73      yyparse();
74      return 0;
75  }
```

Abbildung 7.17: C-Funktionen für den Taschenrechner

Literaturverzeichnis

[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.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, 2nd edition, 2002.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[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.

[Kod04] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javaCC tutorial. *IEEE Software*, 21(4):70–77, 2004.

[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.

[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.

[Par07] Terence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.