KAPITEL 3

Reguläre Ausdrücke und Sprachen

Wir beginnen dieses Kapitel mit der Einführung einer Notation namens »reguläre Ausdrücke«. Bei diesen Ausdrücken handelt es sich um eine andere Art von sprachdefinierender Notation, für die wir in Abschnitt 1.1.2 ein kurzes Beispiel anführten. Man kann sich reguläre Ausdrücke auch als »Programmiersprache« vorstellen, in der einige wichtige Anwendungen wie Suchmaschinen oder Compilerkomponenten beschrieben werden. Reguläre Ausdrücke sind mit nichtdeterministischen endlichen Automaten eng verwandt und können als »benutzerfreundliche« Alternative zur NEA-Notation zur Beschreibung von Softwarekomponenten betrachtet werden.

In diesem Kapitel zeigen wir nach der Definition regulärer Ausdrücke, dass diese Ausdrücke zur Definition aller regulären Sprachen und keiner anderen Sprachen taugen. Wir erörtern die Art und Weise, in der reguläre Ausdrücke in verschiedenen Softwaresystemen eingesetzt werden. Anschließend untersuchen wir die algebraischen Gesetze, die für reguläre Ausdrücke gelten. Obwohl sie den algebraischen Gesetzen der Arithmetik stark ähneln, gibt es einige bedeutende Unterschiede zwischen der Algebra regulärer Ausdrücke und der Algebra arithmetischer Ausdrücke.

3.1 Reguläre Ausdrücke

Wir wenden unsere Aufmerksamkeit nun von maschinenähnlichen Beschreibungen von Sprachen – deterministischen und nichtdeterministischen endlichen Automaten – ab und einer algebraischen Beschreibung zu, nämlich den »regulären Ausdrücken«. Wir werden feststellen, dass reguläre Ausdrücke genau die gleichen Sprachen definieren können, die von den verschiedenen Arten von Automaten beschrieben werden: die regulären Sprachen. Im Gegensatz zu Automaten stellen reguläre Ausdrücke jedoch ein deklaratives Verfahren zur Beschreibung der Zeichenreihen zur Verfügung, die akzeptiert werden sollen. Daher dienen reguläre Ausdrücke als Eingabesprache vieler Systeme, die Zeichenreihen verarbeiten. Beispiele hierfür sind unter anderem:

- Suchbefehle wie der Unix-Befehl `grep` und äquivalente Befehle zur Suche nach Zeichenreihen, die in Webbrowsern oder Textformatiersystemen eingesetzt werden. Diese Systeme verwenden eine regulären Ausdrücken ähnliche Notation zur Beschreibung von Mustern, die der Benutzer in einer Datei sucht. Verschiedene

Suchsysteme wandeln den regulären Ausdruck in einen DEA oder NEA um und simulieren diesen Automaten beim Durchsuchen der gegebenen Datei.

- Generatoren lexikalischer Analysekomponenten, wie z. B. Lex oder Flex. Wie Sie wissen, handelt es sich bei einer lexikalischen Analysekomponente um die Compilerkomponente, die das Quellprogramm in logische Einheiten (so genannte *Token*) zerlegt. Diese logischen Einheiten können ein oder mehrere Zeichen umfassen, die eine gemeinsame Bedeutung haben. Beispiele für Token sind Schlüsselwörter (z. B. `while`), Bezeichner (z. B. ein beliebiger Buchstabe, dem null oder mehr Buchstaben und/oder Ziffern folgen) und Operatorzeichen wie +, - und <=. Ein Generator einer lexikalischen Analysekomponente akzeptiert Beschreibungen der Form der Token, die im Grunde genommen reguläre Ausdrücke sind, und erzeugt einen DEA, der erkennt, welcher Token in der Eingabe als Nächstes folgt.

3.1.1 Die Operatoren regulärer Ausdrücke

Reguläre Operatoren beschreiben Sprachen. Als einfaches Beispiel sei der reguläre Ausdruck $\mathbf{01^* + 10^*}$ angeführt, der die Sprache beschreibt, die aus allen Zeichenreihen besteht, die sich aus einer einzelnen führenden Null und einer beliebigen Anzahl von nachfolgenden Einsen oder aus einer einzelnen führenden Eins und einer beliebigen Anzahl von nachfolgenden Nullen zusammensetzen. Wir erwarten von Ihnen noch nicht, dass Sie reguläre Ausdrücke interpretieren können, und daher müssen Sie unsere Aussage über die Sprache dieses Ausdrucks jetzt einfach erst einmal glauben. Wir werden in Kürze alle Symbole definieren, die in diesem Ausdruck verwendet werden, damit Sie sich selbst davon überzeugen können, dass unsere Interpretation dieses regulären Ausdrucks korrekt ist. Bevor wir die Notation regulärer Ausdrücke beschreiben, müssen wir drei Operationen auf Sprachen vorstellen, die von den Operatoren regulärer Ausdrücke repräsentiert werden. Es handelt sich um folgende Operationen:

1. Die *Vereinigung* zweier Sprachen, L und M, angegeben durch $L \cup M$, ist die Menge aller Zeichenreihen, die entweder in L oder M oder in beiden Sprachen enthalten sind. Wenn z. B. $L = \{001, 10, 111\}$ und $M = \{\varepsilon, 001\}$, dann gilt $L \cup M = \{\varepsilon, 10, 001, 111\}$.

2. Die *Verkettung* (oder Konkatenation) der Sprachen L und M ist die Menge aller Zeichenreihen, die gebildet werden, indem eine Zeichenreihe aus L mit einer beliebigen Zeichenreihe aus M verkettet wird. Rufen Sie sich Abschnitt 1.5.2 in Erinnerung, in dem wir die Verkettung eines Zeichenreihenpaars definiert haben. Eine Zeichenreihe wird an eine andere angehängt, um das Ergebnis der Verkettung zu bilden. Wir geben die Verkettung von Sprachen entweder durch einen Punkt oder durch gar keinen Operator an, auch wenn der Konkatenations- oder Verkettungsoperator häufig »Punkt« genannt wird. Wenn beispielsweise $L = \{001, 10, 111\}$ und $M = \{\varepsilon, 001\}$, dann gilt $L.M$ oder einfach $LM = \{001, 10, 111, 001001, 10001, 111001\}$. Die ersten drei Zeichenreihen von LM sind die mit ε verketteten Zeichenreihen aus L. Da ε die Identität der Verkettung ist, entsprechen die resultierenden Zeichenreihen den Zeichenreihen aus L. Die letzten drei Zeichenreihen von LM werden jedoch gebildet, indem die einzelnen Zeichenreihen von L mit der zweiten Zeichenreihe von M, also 001, verket-

tet werden. Beispielsweise ergibt die Verkettung der Zeichenreihe 10 aus L mit der Zeichenreihe 001 aus M die Zeichenreihe 10001 in LM.

3. Die *Kleenesche Hülle* [1] (oder einfach *Hülle* oder *Stern* genannt) einer Sprache L wird durch L^* angegeben und beschreibt die Menge aller Zeichenreihen, die durch die Verkettung einer beliebigen Anzahl von Zeichenreihen aus L (wobei Wiederholungen zulässig sind, d. h. dieselbe Zeichenreihe kann mehrmals verwendet werden) gebildet werden. Wenn z. B. $L = \{0, 1\}$, dann enthält L^* alle Zeichenreihen aus Nullen und Einsen. Wenn $L = \{0, 11\}$, dann enthält L^* alle Zeichenreihen aus Nullen und Einsen, in denen die Einsen paarweise auftreten, also z. B. 011, 11110 und ε, nicht jedoch 01011 oder 101. Formal ausgedrückt ist L^* die unendliche Vereinigung $\bigcup_{i \geq 0} L^i$, wobei $L^0 = \{\varepsilon\}$, $L^1 = L$ und L^i für $i > 1$ gleich $LL ... L$ (die Verkettung von i Exemplaren von L) ist.

Beispiel 3.1 Da das Konzept der Hülle einer Sprache nicht banal ist, wollen wir einige Beispiele betrachten. Sei $L = \{0, 11\}$. Unabhängig davon, um was für eine Sprache es sich bei L handelt, gilt stets $L^0 = \{\varepsilon\}$; L hoch null repräsentiert die Auswahl von null Zeichenreihen aus L. $L^1 = L$ steht also für die Auswahl einer Zeichenreihe aus L. Die ersten beiden Terme der Erweiterung von L^* ergeben somit $\{\varepsilon, 0, 11\}$.

Als Nächstes betrachten wir L^2. Wir wählen zwei Zeichenreihen aus L aus, und da Wiederholungen zulässig sind, gibt es vier Möglichkeiten. Diese vier Kombinationen ergeben $L^2 = \{00, 011, 110, 1111\}$. Analog hierzu ist L^3 die Menge der Zeichenreihen, die sich durch Verkettung von drei aus L gewählten Zeichenreihen ergeben, also

$$\{000, 0011, 0110, 01111, 1100, 11011, 11110, 111111\}$$

Zur Berechnung von L^* müssen wir L^i für jedes i berechnen und alle diese Sprachen vereinigen. L^i enthält 2^i Elemente. Obwohl jede Sprache L^i endlich ist, ergibt die Vereinigung der unendlichen Anzahl von Termen L^i im Allgemeinen wie in unserem Beispiel eine unendliche Sprache.

Nehmen wir nun an, L sei die Menge aller aus Nullen bestehenden Zeichenreihen. Beachten Sie, dass L hier unendlich ist, im Gegensatz zum vorigen Beispiel, in dem L eine endliche Sprache war. Es ist allerdings nicht schwierig, L^* zu ermitteln. Wie immer gilt: $L^0 = \{\varepsilon\}$. $L^1 = L$. L^2 ist die Menge der Zeichenreihen, die gebildet werden, indem man eine Zeichenreihe aus Nullen mit einer anderen Zeichenreihe aus Nullen verkettet. Auch das Ergebnis ist eine Zeichenreihe aus Nullen. Jede Zeichenreihe aus Nullen kann als Verkettung zweier Zeichenreihen aus Nullen dargestellt werden. (Denken Sie daran, dass ε eine »Zeichenreihe aus Nullen« ist; in der Verkettung zweier Zeichenreihen können wir in jedem Fall diese Zeichenreihe verwenden.) Folglich gilt $L^2 = L$. Analog gilt $L^3 = L$. Folglich ist in dem speziellen Fall, in dem die Sprache L die Menge aller aus Nullen bestehenden Zeichenreihen umfasst, die unendliche Vereinigung $L^* = L^0 \cup L^1 \cup L^2 \cup ...$ gleich L.

Als letztes Beispiel wollen wir die Sprache $\varnothing^* = \{\varepsilon\}$ betrachten. Beachten Sie, dass $\varnothing^0 = \{\varepsilon\}$, während $\varnothing^i$ für jedes $i > 1$ leer ist, da wir aus einer leeren Menge keine Zeichenreihe auswählen können. In der Tat stellt $\varnothing$ eine der beiden einzigen Sprachen dar, deren Hülle *nicht* unendlich ist. ■

1. Der Begriff »Kleenesche Hülle« geht auf S. C. Kleene zurück, von dem die Notation der regulären Ausdrücke und dieser Operator stammen.

Verwendung des Sternoperators

Der Sternoperator wurde im Abschnitt 1.5.2 zum ersten Mal erwähnt, in dem wir ihn auf ein Alphabet anwendeten, z. B. Σ^*. Dieser Operator bildete sämtliche Zeichenreihen, deren Symbole aus dem Alphabet Σ stammten. Der Kleene-Operator (oder Hüllen-Operator) ist im Grunde genommen dasselbe, obwohl ein subtiler Typunterschied zu beachten ist.

Angenommen, L sei die Sprache, die aus Zeichenreihen der Länge 1 besteht, und für jedes in Σ enthaltene Zeichen a gäbe es eine Zeichenreihe a in L. Dann haben L und Σ zwar dasselbe »Aussehen«, sind jedoch unterschiedlichen Typs. L ist eine Menge von Zeichenreihen, und Σ ist eine Menge von Symbolen. Andererseits bezeichnet L^* dieselbe Sprache wie Σ^*.

3.1.2 Reguläre Ausdrücke bilden

Jegliche Art von Algebra setzt auf gewissen elementaren Ausdrücken auf, für gewöhnlich Konstanten und/oder Variablen. Durch die Anwendung bestimmter Operatoren auf diese elementaren Ausdrücke und auf bereits gebildete Ausdrücke können dann weitere Ausdrücke gebildet werden. Für gewöhnlich ist zudem eine Methode zur Gruppierung der Operatoren und ihrer Operanden erforderlich, wie z. B. das Klammern. Die bekannte arithmetische Algebra beginnt beispielsweise mit Konstanten (z. B. ganze und reelle Zahlen) und Variablen und ermöglicht die Bildung komplexerer Ausdrücke unter Verwendung der arithmetischen Operatoren wie + und x.

Die Arithmetik regulärer Ausdrücke folgt diesem Muster insofern, als dass Konstanten und Variablen zur Bezeichnung von Sprachen und Operatoren für die in Abschnitt 3.1.1. beschriebenen Operationen – Vereinigung, Punkt und Stern – verwendet werden. Wir können die regulären Ausdrücke wie folgt induktiv beschreiben. In dieser Definition legen wir nicht nur fest, was zulässige reguläre Ausdrücke sind, sondern wir beschreiben für jeden regulären Ausdruck E auch die von diesem Ausdruck repräsentierte Sprache, die wir mit $L(E)$ angeben.

INDUKTIONSBEGINN: Er besteht aus drei Teilen:

1. Die Konstanten ε und $\emptyset$ sind reguläre Ausdrücke, die die Sprache $\{\varepsilon\}$ bzw. $\emptyset$ beschreiben. Das heißt, $L(\varepsilon) = \{\varepsilon\}$ und $L(\emptyset) = \emptyset$.
2. Wenn a ein beliebiges Symbol ist, dann ist **a** ein regulärer Ausdruck. Dieser Ausdruck beschreibt die Sprache $\{a\}$. Das heißt, $L(\mathbf{a}) = \{a\}$. Beachten Sie, dass wir einen Ausdruck, der einem Symbol entspricht, durch Fettschrift angeben. Die Zuordnung, d. h. dass **a** sich auf a bezieht, sollte offensichtlich sein.
3. Variablen, die für gewöhnlich als kursive Großbuchstaben wie L angegeben werden, repräsentieren eine Sprache.

INDUKTIONSSCHRITT: Der Induktionsschritt umfasst vier Teile, jeweils einen für die drei Operatoren und einen für die Einführung von Klammern.

1. Wenn E und F reguläre Ausdrücke sind, dann ist $E + F$ ein regulärer Ausdruck, der die Vereinigung von $L(E)$ und $L(F)$ angibt. Das heißt, $L(E + F) = L(E) \cup L(F)$.
2. Wenn E und F reguläre Ausdrücke sind, dann ist EF ein regulärer Ausdruck, der die Verkettung von $L(E)$ und $L(F)$ angibt. Das heißt, $L(EF) = L(E)L(F)$.

Beachten Sie, dass der Punkt optional für den Verkettungsoperator angegeben werden kann, der für eine Operation auf Sprachen oder als Operator in regulären Ausdrücken verwendet werden kann. Beispielsweise ist **0.1** ein regulärer Ausdruck, der gleichbedeutend ist mit **01** und die Sprache {01} beschreibt. Wir werden den Punkt als Verkettungsoperator in regulären Ausdrücken vermeiden[2].

3. Wenn E ein regulärer Ausdruck ist, dann ist E^* ein regulärer Ausdruck, der die Hülle von $L(E)$ angibt. Das heißt $L(E^*) = (L(E))^*$.

4. Wenn E ein regulärer Ausdruck ist, dann ist auch (E) – der in Klammern gesetzte Bezeichner – ein regulärer Ausdruck, der die gleiche Sprache wie E beschreibt. Formal ausgedrückt: $L((E)) = L(E)$.

Ausdrücke und ihre Sprachen

Streng genommen ist ein regulärer Ausdruck E einfach ein Ausdruck und keine Sprache. Wir sollten $L(E)$ verwenden, wenn wir auf die Sprache Bezug nehmen wollen, die von E beschrieben wird. Es ist jedoch übliche Praxis, einfach »E« zu sagen, wenn eigentlich »$L(E)$« gemeint ist. Wir werden diese Konvention hier verwenden, solange klar ist, dass von einer Sprache und nicht von einem regulären Ausdruck die Rede ist.

Beispiel 3.2 Wir sollen einen regulären Ausdruck formulieren, der die Menge der Zeichenreihen beschreibt, die aus in abwechselnder Reihenfolge stehenden Nullen und Einsen bestehen. Wir wollen zuerst einen regulären Ausdruck für die Sprache entwickeln, die lediglich die Zeichenreihe 01 beinhaltet. Wir können dann mithilfe des Sternoperators einen Ausdruck bilden, der für alle Zeichenreihen der Form 0101 ... 01 steht.

Die Grundregel für reguläre Ausdrücke besagt, dass **0** und **1** Ausdrücke sind, die die Sprachen {0} bzw. {1} repräsentieren. Wenn wir die beiden Ausdrücke verketten, erhalten wir einen regulären Ausdruck für die Sprache {01}. Dieser Ausdruck lautet **01**. Als allgemeine Regel gilt, dass wir als regulären Ausdruck zur Beschreibung einer Sprache, die nur aus der Zeichenreihe w besteht, die Zeichenreihe w selbst als regulären Ausdruck einsetzen können. Beachten Sie, dass im regulären Ausdruck die in w enthaltenen Symbole normalerweise in Fettschrift dargestellt werden, aber diese Änderung der Schriftart soll es Ihnen lediglich erleichtern, Ausdrücke von Zeichenreihen zu unterscheiden, und hat keine weitere Bewandtnis.

Um alle Zeichenreihen zu erhalten, die aus null oder mehr Vorkommen von 01 bestehen, verwenden wir den regulären Ausdruck $(\mathbf{01})^*$. Beachten Sie, dass wir **01** in Klammer setzen, um Verwechslungen mit dem Ausdruck **01*** zu vermeiden, der die Sprache beschreibt, die alle Zeichenreihen umfasst, die sich aus einer führenden Null und einer beliebigen Anzahl von nachfolgenden Einsen zusammensetzen. Warum dieser Ausdruck so interpretiert wird, wird in Abschnitt 3.1.3 erklärt; hier soll der Hinweis genügen, dass der Sternoperator Vorrang vor dem Punktoperator hat und das Argument des Sternoperators daher zuerst ausgewählt wird, bevor eine Verkettung durchgeführt wird.

2. In regulären Ausdrücken unter Unix hat der Punkt sogar eine völlig andere Bedeutung: Hier repräsentiert er ein beliebiges ASCII-Zeichen.

Allerdings ist $L((\mathbf{01})^*)$ nicht genau die Sprache, die gewünscht ist. Sie umfasst nur solche Zeichenreihen aus in abwechselnder Reihenfolge stehender Nullen und Einsen, die mit null beginnen und mit 1 enden. Wir müssen zudem die Möglichkeit betrachten, dass eine Eins am Anfang und/oder eine Null am Ende steht. Ein Ansatz besteht darin, drei oder mehr reguläre Ausdrücke zu bilden, die die anderen drei Möglichkeiten abdecken. Das heißt, $(\mathbf{10})^*$ repräsentiert die Zeichenreihen, die mit einer Eins beginnen und einer Null enden, während $\mathbf{0}(\mathbf{10})^*$ für Zeichenreihen steht, die sowohl mit einer Null beginnen als auch mit einer Null enden, und $\mathbf{1}(\mathbf{01})^*$ Zeichenreihen beschreibt, die sowohl mit einer Eins beginnen als auch mit einer Eins enden. Der gesamte reguläre Ausdruck lautet:

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Beachten Sie, dass wir den Operator + verwendet haben, um die Vereinigung der vier Sprachen zu bilden, die insgesamt sämtliche Zeichenreihen umfasst, in denen Nullen und Einsen einander abwechseln.

Es gibt jedoch noch einen anderen Ansatz, der einen völlig anders aussehenden regulären Ausdruck ergibt und überdies etwas prägnanter ist. Wir beginnen wieder mit dem Ausdruck $(\mathbf{01})^*$. Wir können eine optionale Eins am Anfang hinzufügen, wenn wir Links mit dem Ausdruck $\varepsilon + \mathbf{1}$ verketten. Analog können wir mit dem Ausdruck $\varepsilon + \mathbf{0}$ eine optionale Null am Ende hinzufügen. Unter Verwendung der Definition des Operators + ergibt sich beispielsweise:

$$L(\varepsilon + \mathbf{1}) = \mathrm{L}(\varepsilon) \cup \mathrm{L}(\mathbf{1}) = \{\varepsilon\} \cup \{\mathbf{1}\} = \{\varepsilon, \mathbf{1}\}$$

Wenn wir diese Sprache mit einer beliebigen Sprache L verketten, dann erhalten wir durch die Verkettung mit ε alle in L enthaltenen Zeichenreihen, während die Verkettung von 1 mit jeder Zeichenreihe w in L die Zeichenreihe $1w$ ergibt. Folglich lautet ein anderer Ausdruck zur Beschreibung der Menge der Zeichenreihen, die abwechselnd Nullen und Einsen enthalten:

$$(\varepsilon + \mathbf{1})\ (\mathbf{01})^*(\varepsilon + \mathbf{0})$$

Beachten Sie, dass die Ausdrücke mit ε jeweils in Klammern gesetzt werden müssen, damit die Operatoren korrekt ausgewertet werden. ■

3.1.3 Auswertungsreihenfolge der Operatoren regulärer Ausdrücke

Wie in jeder anderen Algebra gelten für die Operatoren regulärer Ausdrücke festgelegte »Vorrangregeln«, das heißt, dass die Operatoren in einer bestimmten Reihenfolge ihren Operanden zugeordnet werden. Dieses Konzept der Vorrangregel ist uns von gewöhnlichen arithmetischen Ausdrücken bekannt. Wir wissen beispielsweise, dass im Ausdruck $xy + z$ das Produkt xy vor der Summe berechnet wird, und daher ist er gleichbedeutend mit dem Klammerausdruck $(xy) + z$ und nicht mit dem Ausdruck $x(y + z)$. Zwei gleiche Operatoren werden in der Arithmetik von links nach rechts ausgewertet, sodass $x - y - z$ gleichbedeutend ist mit $(x - y) - z$ und nicht mit $x - (y - z)$. Für die Operatoren regulärer Ausdrücke gilt die folgende Auswertungsreihenfolge:

1. Der Sternoperator hat die höchste Priorität. Das heißt, er wird nur auf die kleinste Sequenz von Symbolen auf seiner linken Seite angewandt, die einen wohl geformten regulären Ausdruck darstellen.

2. Der Verkettungsoperator bzw. der Operator »Punkt« folgt an zweiter Stelle in der Auswertungsreihenfolge. Nachdem alle Sterne ihren Operanden zugeordnet wurden, werden die Verkettungsoperatoren ihren Operanden zugeordnet. Das heißt, alle unmittelbar aufeinander folgenden Ausdrücke werden gruppiert. Da der Verkettungsoperator assoziativ ist, ist es gleichgültig, in welcher Reihenfolge aufeinander folgende Verkettungen gruppiert werden. Falls aber eine Gruppierung erwünscht ist, sollten die Ausdrücke von links nach rechts gruppiert werden; z. B. wird **012** gruppiert in (**01**)**2**.
3. Zuletzt werden alle Vereinigungsoperatoren (+) ihren Operanden zugeordnet. Da die Vereinigung assoziativ ist, spielt es auch hier keine Rolle, in welcher Reihenfolge aufeinander folgende Vereinigungen gruppiert werden; eine Gruppierung sollte aber auch in diesem Fall von links nach rechts erfolgen.

Natürlich kann es vorkommen, dass wir die Operanden anders gruppieren möchten, als durch die Auswertungsreihenfolge der Operatoren vorgegeben. In diesem Fall können wir die Operanden mithilfe runder Klammern nach Belieben gruppieren. Zudem schadet es nichts, die Operanden in Klammern zu setzen, die gruppiert werden sollen, auch wenn die gewünschte Gruppierung durch die Vorrangregeln bereits impliziert wird.

3.1.4 Übungen zum Abschnitt 3.1

Übung 3.1.1 Schreiben Sie reguläre Ausdrücke für die folgenden Sprachen:

* a) Die Menge der Zeichenreihen über dem Alphabet $\{a, b, c\}$, die mindestens ein a und mindestens ein b enthalten

b) Die Menge der Zeichenreihen aus Nullen und Einsen, deren zehntes Symbol von rechts eine Eins ist

c) Die Menge der Zeichenreihen aus Nullen und Einsen, die mindestens ein Paar aufeinander folgender Einsen enthalten

! Übung 3.1.2 Formulieren Sie reguläre Ausdrücke für folgende Sprachen:

* a) Die Menge aller Zeichenreihen aus Nullen und Einsen, derart dass alle Paare aufeinander folgender Nullen vor allen Paaren aufeinander folgender Einsen stehen

b) Die Menge der Zeichenreihen aus Nullen und Einsen, deren Anzahl von Nullen durch 5 teilbar ist

!! Übung 3.1.3 Formulieren Sie reguläre Ausdrücke für folgende Sprachen:

a) Die Menge aller Zeichenreihen aus Nullen und Einsen, die die Teilzeichenreihe 101 nicht enthalten

b) Die Menge aller Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen, die so geformt sind, dass jedes Präfix weder zwei Nullen mehr als Einsen noch zwei Einsen mehr als Nullen enthält

c) Die Menge aller Zeichenreihen aus Nullen und Einsen, deren Anzahl von Nullen durch 5 teilbar und deren Anzahl von Einsen gerade ist

! Übung 3.1.4 Beschreiben Sie die Sprachen der folgenden regulären Ausdrücke in Worten:

* a) $(\mathbf{1} + \varepsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$

c) $(\mathbf{0} + \mathbf{10})^*\mathbf{1}^*$

***! Übung 3.1.5** Wir haben in Beispiel 3.1 darauf hingewiesen, dass $\emptyset$ eine der beiden Sprachen ist, deren Hülle endlich ist. Wie lautet die andere Sprache?

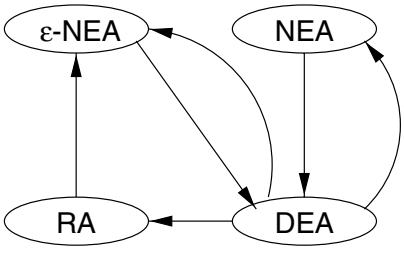
3.2 Endliche Automaten und reguläre Ausdrücke

Obwohl der in regulären Ausdrücken verwendete Ansatz zur Beschreibung von Sprachen sich grundlegend von dem endlicher Automaten unterscheidet, repräsentieren diese beiden Notationen die gleiche Menge von Sprachen, die wir als »reguläre Sprachen« bezeichnet haben. Wir haben bereits gezeigt, dass deterministische endliche Automaten und die beiden Typen nichtdeterministischer Automaten – mit und ohne ε-Übergänge(n) – die gleiche Sprachklasse akzeptieren. Um zu zeigen, dass reguläre Ausdrücke dieselbe Klasse definieren, müssen wir zeigen, dass

1. jede Sprache, die durch einen dieser Automaten definiert wird, auch durch einen regulären Ausdruck definiert wird. Für diesen Beweis können wir annehmen, dass die Sprache von einem DEA akzeptiert wird.
2. jede Sprache, die durch einen regulären Ausdruck definiert wird, auch durch einen dieser Automaten definiert wird. Für diesen Teil des Beweises ist es am einfachsten, wenn wir zeigen, dass es einen NEA mit ε-Übergängen gibt, der dieselbe Sprache akzeptiert.

Abbildung 3.1 zeigt die Äquivalenzen, die wir bewiesen haben oder beweisen werden. Ein Pfeil von Klasse X zu Klasse Y bedeutet, dass wir bewiesen haben, dass jede von Klasse X definierte Sprache auch von Klasse Y definiert wird. Da der Graph stark zusammenhängend ist (d. h. wir können von jedem der vier Knoten zu jedem anderen Knoten gelangen), sehen wir, dass alle vier Klassen tatsächlich gleich sind.

Abbildung 3.1: Plan des Beweises der Äquivalenz von vier verschiedenen Notationen für reguläre Sprachen



3.2.1 Von DEAs zu regulären Ausdrücken

Die Bildung eines regulären Ausdrucks, der die Sprache eines beliebigen DEA definiert, ist überraschenderweise nicht so einfach. In groben Zügen erklärt, müssen wir Ausdrücke formulieren, die Mengen von Zeichenreihen beschreiben, die als Beschriftungen bestimmter Pfade im Übergangsdiagramm des DEA auftreten. Allerdings dürfen diese Pfade nur eine begrenzte Teilmenge von Zuständen passieren. In einer induktiven Definition dieser Ausdrücke beginnen wir mit den einfachsten, die Pfade beschreiben, die überhaupt *keine* Zustände passieren dürfen (d. h. es sind einzelne Knoten oder einzelne Pfeile), und bilden dann die Ausdrücke, die die Pfade durch zunehmend größere Mengen von Zuständen führen. Schließlich lassen wir zu, dass die Pfade durch eine beliebige Anzahl von Zuständen führen; d. h. die Ausdrücke, die wir am Schluss erzeugen, repräsentieren alle möglichen Pfade. Der Gedankengang wird im Beweis des folgenden Satzes deutlich.

Satz 3.4 Wenn für einen beliebigen DEA A $L = L(A)$ gilt, dann gibt es einen regulären Ausdruck R, derart dass $L = L(R)$.

Beweis: Angenommen, A verfügt für eine ganze Zahl n über die Zustände $\{1, 2, ..., n\}$. Gleichgültig, wie die Zustände von A geartet sind, für eine endliche Zahl n gibt es immer n Zustände, und indem wir die Zustände umbenennen, können wir darauf so Bezug nehmen, als handele es sich um die ersten n positiven ganzen Zahlen. Unsere erste und schwierigste Aufgabe besteht darin, eine Menge von regulären Ausdrücken zu konstruieren, die die zunehmend umfangreicher werdende Menge von Pfaden im Übergangsdiagramm von A beschreiben.

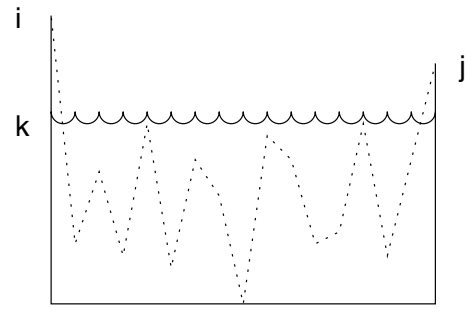
Wir wollen $R_{ij}^{(k)}$ als Namen für den regulären Ausdruck verwenden, dessen Sprache die Menge der Zeichenreihen w umfasst, sodass w die Beschriftung eines Pfades von Zustand i zum Zustand j in A ist und der Pfad keine dazwischenliegenden Knoten berührt, deren Nummer größer als k ist. Beachten Sie, dass Anfangs- und Endpunkt des Pfades nicht »dazwischenliegend« sind; es gibt also keine Beschränkung, die besagt, dass i und/oder j kleiner gleich k sein müssen.

Abbildung 3.2 veranschaulicht die Anforderungen, die für die durch $R_{ij}^{(k)}$ beschriebenen Pfade gelten. In dieser Abbildung repräsentiert die vertikale Richtung die Zustände, die von unten nach oben von 1 bis n an der vertikalen Achse angetragen sind. Die horizontale Achse repräsentiert die Bewegung entlang des Pfades. Beachten Sie, dass i und j auch kleiner oder gleich k sein könnten, obwohl wir in diesem Diagramm sowohl i als auch j größer als k dargestellt haben. Beachten Sie zudem, dass der Pfad zweimal durch Knoten k verläuft, jedoch, mit Ausnahme der Endpunkte, nie einen Knoten berührt, dessen Nummer größer als k ist.

Zur Bildung der Ausdrücke $R_{ij}^{(k)}$ verwenden wir die folgende induktive Definition, wobei wir bei $k = 0$ beginnen und schließlich $k = n$ erreichen. Beachten Sie, dass im Fall $k = n$ die dargestellten Pfade keinerlei Beschränkungen unterliegen, da es keine Zustände größer n gibt.

Induktionsbeginn: $k = 0$. Da alle Zustände beginnend mit 1 aufsteigend nummeriert sind, gilt als Beschränkung für Pfade, dass der Pfad keine dazwischenliegenden Knoten enthalten darf. Es gibt nur zwei Arten von Pfaden, die eine solche Bedingung erfüllen:

Abbildung 3.2: Ein Pfad, dessen Beschriftung in der Sprache des regulären Ausdrucks $R_{ij}^{(k)}$ enthalten ist



1. Ein Pfeil von Knoten (Zustand) i zu Knoten j
2. Ein Pfad mit der Länge 0, der nur aus dem Knoten i besteht

Wenn $i \neq j$, dann ist nur der Fall (1) möglich. Wir müssen den DEA A überprüfen und jene Eingabesymbole a finden, nach deren Eingabe ein Übergang vom Zustand i in den Zustand j erfolgt.

a) Existiert kein solches Symbol a, dann gilt $R_{ij}^{(0)} = \emptyset$.

b) Gibt es genau ein solches Symbol, dann gilt $R_{ij}^{(0)} = \mathbf{a}$.

c) Wenn es Symbole $a_1, a_2, \ldots, a_k$ gibt, die als Beschriftung der Pfeile auf dem Pfad vom Zustand i zum Zustand j dienen, dann gilt $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

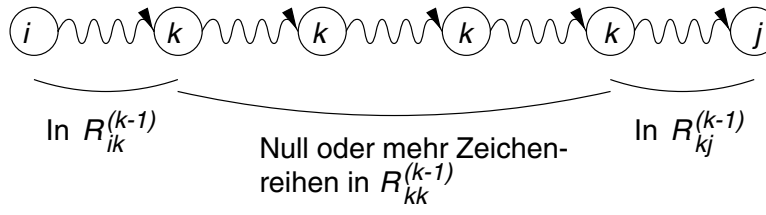
Wenn jedoch $i = j$ ist, dann sind nur der Pfad der Länge 0 und alle Schleifen im Knoten i zulässig. Der Pfad der Länge 0 wird durch den regulären Ausdruck ε repräsentiert, da auf diesem Pfad keine Symbole liegen. Folglich fügen wir ε zu den oben unter a) und c) angegebenen Ausdrücken hinzu. Das heißt, im Fall (a) [keine Symbole a] lautet der Ausdruck ε , im Fall (b) [ein Symbol a] erhalten wir nun den Ausdruck $\varepsilon + \mathbf{a}$ und im Fall (c) [mehrere Symbole] lautet der Ausdruck jetzt $\varepsilon + \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

INDUKTIONSSCHRITT: $k > 0$. Angenommen, es gibt einen Pfad vom Zustand i zum Zustand j, der durch keinen Zustand mit einer höheren Nummer als k führt. Hier sind zwei mögliche Fälle zu betrachten:

1. Zustand k liegt überhaupt nicht auf dem Pfad. In diesem Fall ist die Beschriftung dieses Pfades in der Sprache von $R_{ij}^{(k-1)}$ enthalten.
2. Der Pfad führt mindestens einmal durch Zustand k. In diesem Fall können wir den Pfad in mehrere Teilstücke aufspalten, wie in Abbildung 3.3 dargestellt. Das erste Segment führt vom Zustand i zum Zustand k, ohne k zu passieren, und das letzte Segment führt vom Zustand k direkt zum Zustand j, ohne k zu passieren. Alle anderen Segmente in der Mitte führen von k nach k, ohne k zu passieren. Beachten Sie, dass es keine »mittleren« Segmente gibt, wenn der Pfad den Zustand k nur einmal durchläuft, sondern lediglich einen Pfad von i nach k und einen Pfad von k nach j. Die Menge der Beschriftungen für alle Pfade dieses Typs werden durch den regulären Ausdruck $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$

beschrieben. Das heißt, der erste Ausdruck repräsentiert den Teil des Pfades, der zum ersten Mal zum Zustand *k* führt, der zweite repräsentiert den Teil, der nullmal, einmal oder mehrmals von *k* nach *k* führt, und der dritte Ausdruck repräsentiert den Teil des Pfades, der *k* zum letzten Mal verlässt und zum Zustand *j* führt.

Abbildung 3.3: Ein Pfad von *i* nach *j* kann an jedem Punkt, an dem er durch Zustand *k* führt, in Segmente aufgespaltet werden



Wenn wir die Ausdrücke für die Pfade der oben beschriebenen beiden Typen kombinieren, erhalten wir den Ausdruck

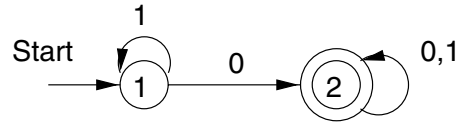
$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)}) * R_{kj}^{(k-1)}$$

für die Beschriftungen aller Pfade vom Zustand *i* zum Zustand *j*, die durch keinen Zustand mit einer höheren Nummer als *k* führen. Wenn wir diese Ausdrücke in der Reihenfolge zunehmender oberer Indizes *k* bilden, dann sind alle Ausdrücke an der richtigen Stelle verfügbar, denn jeder Ausdruck $R_{ij}^{(k)}$ hängt nur von Ausdrücken mit kleineren oberen Indizes ab.

Schließlich erhalten wir $R_{ij}^{(n)}$ für alle *i* und *j*. Wir können annehmen, dass Zustand 1 der Startzustand ist, auch wenn die akzeptierenden Zustände aus jeder beliebigen Menge von Zuständen bestehen könnten. Der reguläre Ausdruck für die Sprache des Automaten ergibt sich dann aus der Summe (Vereinigung) aller Ausdrücke $R_{1j}^{(n)}$, wobei *j* ein akzeptierender Zustand ist. ■

Beispiel 3.5 Wir wollen den in Abbildung 3.4 gezeigten DEA in einen regulären Ausdruck umwandeln. Dieser DEA akzeptiert alle Zeichenreihen, die mindestens eine Null enthalten. Dies geht daraus hervor, dass der Automat vom Startzustand 1 in den akzeptierenden Zustand 2 übergeht, sobald er die Eingabe 0 erhält. Der Automat bleibt danach für alle weiteren Eingabesequenzen im Zustand 2.

Abbildung 3.4: Ein DEA, der alle Zeichenreihen akzeptiert, die mindestens eine Null enthalten



Nachfolgend sind die Anfangsausdrücke in der Konstruktion von Satz 3.4 aufgeführt:

$R_{11}^{(0)}$	$\varepsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\varnothing$
$R_{22}^{(0)}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

$R_{11}^{(0)}$ enthält den Term ε, weil Anfangs- und Endzustand gleich sind, nämlich Zustand 1. Der Ausdruck enthält den Term 1, weil für die Eingabe 1 ein Pfeil (Schleife) vom Zustand 1 zum Zustand 1 führt. $R_{12}^{(0)}$ ist $\mathbf{0}$, weil ein Pfeil mit der Beschriftung 0 vom Zustand 1 zum Zustand 2 führt. Hier ist der Term ε nicht gegeben, weil Anfangs- und Endzustand verschieden sind. Als drittes Beispiel sei $R_{21}^{(0)} = \varnothing$ angeführt; dieser Ausdruck entspricht der leeren Menge, weil kein Pfeil vom Zustand 2 zum Zustand 1 führt.

Wir müssen jetzt den Induktionsschritt durchführen und komplexere Ausdrücke erstellen, die zuerst Pfaden Rechnung tragen, die durch Zustand 1 führen, und anschließend Pfaden, die durch die Zustände 1 und 2 führen. Damit erhalten wir alle Pfade. Die Ausdrücke $R_{ij}^{(1)}$ werden nach der Regel berechnet, die im Induktionsschritt von Satz 3.4 angegeben wurde:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

In Tabelle 3.1 sind zuerst die Ausdrücke aufgeführt, die durch direkte Substitution von Werten in die obige Formel berechnet wurden, und in der nachfolgenden Spalte die vereinfachten Ausdrücke, die nachweislich dieselbe Sprache repräsentieren wie die komplizierteren Ausdrücke.

Tabelle 3.1: Reguläre Ausdrücke für Pfade, die nur Zustand 1 passieren dürfen

	Direkte Substitution	**Vereinfacht**
$R_{11}^{(1)}$	$\varepsilon + \mathbf{1} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^* \mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\varnothing + \varnothing(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\varnothing$
$R_{22}^{(1)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + \varnothing(\varepsilon + \mathbf{1})^* \mathbf{0}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

Betrachten Sie beispielsweise $R_{12}^{(1)}$. Der entsprechende Ausdruck lautet $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, den wir nach (3.1) durch Substitution von $i = 1$ und $j = 2$ erhalten.

Zum Verständnis der Vereinfachung ist die allgemeine Grundregel zu beachten, die besagt: Wenn R ein beliebiger regulärer Ausdruck ist, dann gilt $(\varepsilon + R)^* = R^*$. Gerechtfertigt wird dies dadurch, dass beide Seiten der Gleichung die Sprache beschreiben, die aus beliebigen Verkettungen von null oder mehr Zeichenreihen aus

$L(R)$ bestehen. In unserem Fall liegt $(\varepsilon + \mathbf{1})^* = \mathbf{1}^*$ vor. Beachten Sie, dass beide Ausdrücke eine beliebige Anzahl von Einsen angeben. Zudem gilt: $(\varepsilon + \mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Wieder ist feststellbar, dass beide Ausdrücke »eine beliebige Anzahl von Einsen« angeben. Folglich ist der ursprüngliche Ausdruck $R_{12}^{(1)}$ äquivalent mit $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. Dieser Ausdruck beschreibt die Sprache, die die Zeichenreihe 0 und alle Zeichenreihen enthält, die aus einer Null mit einer beliebigen Anzahl voranstehender Einsen bestehen. Diese Sprache wird auch durch den einfacheren Ausdruck **1*0** repräsentiert.

Die Vereinfachung von $R_{11}^{(1)}$ ähnelt der Vereinfachung von $R_{12}^{(1)}$, die wir gerade betrachtet haben. Die Vereinfachung von $R_{21}^{(1)}$ und $R_{22}^{(1)}$ hängt von zwei Regeln zur Bedeutung von $\varnothing$ ab. Für jeden regulären Ausdruck R gilt:

1. $\varnothing R = R\varnothing = \varnothing$. Das heißt, $\varnothing$ ist ein Nulloperator der Verkettung. Wenn die leere Menge mit einem beliebigen Ausdruck, der links oder rechts stehen kann, verkettet wird, dann ist das Ergebnis die leere Menge. Diese Regel leuchtet ein, weil eine Zeichenreihe nur dann Teil des Ergebnisses einer Verkettung ist, wenn sie Zeichenreihen beider Argumente der Verkettung enthält. Wenn eines der Argumente gleich $\varnothing$ ist, trägt es keine Zeichenreihe zur Verkettung bei und folglich gibt es kein Verkettungsergebnis, das aus Zeichenreihen beider Argumente besteht.

2. $\varnothing + R = R + \varnothing = R$. Das heißt, $\varnothing$ ist der Identitätsoperator der Vereinigung. Wenn die leere Menge mit einem Ausdruck vereinigt wird, ist das Ergebnis immer dieser Ausdruck.

Infolgedessen kann ein Ausdruck wie $\varnothing\,(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$ durch $\varnothing$ ersetzt werden. Damit sollten die letzten beiden Vereinfachungen klar sein.

Wir wollen nun die Ausdrücke $R_{ij}^{(2)}$ berechnen. Durch Anwendung der Induktionsregel mit $k = 2$ erhalten wir:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

Wenn wir die vereinfachten Ausdrücke aus Tabelle 3.1 in (3.2) einsetzen, erhalten wir die in Tabelle 3.2 aufgeführten Ausdrücke. Diese Tabelle enthält zudem Vereinfachungen, die nach den oben für Tabelle 3.1 beschriebenen Regeln gewonnen wurden.

Tabelle 3.2: Reguläre Ausdrücke für Pfade, die durch beliebige Zustände führen können

	Direkte Substitution	**Vereinfacht**
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\,\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\varnothing$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\ \ (\varepsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\varnothing + (\varepsilon + \mathbf{0} + \mathbf{1})\,(\varepsilon + \mathbf{0} + \mathbf{1})^*\,\varnothing$	$\varnothing$
$R_{22}^{(2)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*\,(\varepsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Der letzte mit dem Automaten aus Abbildung 3.4 äquivalente reguläre Ausdruck wird konstruiert, indem die Vereinigung aller Ausdrücke gebildet wird, die als ersten Zustand den Startzustand und als zweiten Zustand einen akzeptierenden Zustand angeben. In diesem Beispiel, in dem 1 der Startzustand und 2 der einzige akzeptie-

rende Zustand ist, brauchen wir lediglich den Ausdruck $R_{12}^{\{2\}}$. Dieser Ausdruck ergibt $\mathbf{1^*0(0 + 1)^*}$. Die Interpretation dieses Ausdrucks ist einfach. Er beschreibt die Sprache, die aus allen Zeichenreihen besteht, die mit null oder mehr Einsen beginnen und denen eine Null und anschließend eine beliebige Zeichenreihe aus Nullen und Einsen folgen. Anders ausgedrückt, die Sprache umfasst alle Zeichenreihen aus Nullen und Einsen, die mindestens eine Null enthalten. ■

3.2.2 DEA durch die Eliminierung von Zuständen in reguläre Ausdrücke umwandeln

Die in Abschnitt 3.2.1 beschriebene Methode zur Umwandlung eines DEA in einen regulären Ausdruck funktioniert immer. Wie Sie vielleicht bemerkt haben, hängt sie nicht davon ab, dass der Automat deterministisch ist; sie könnte genauso gut auf einen NEA oder sogar einen ε-NEA angewandt werden. Die Bildung des regulären Ausdrucks ist jedoch aufwändig. Wir müssen für einen Automaten mit n Zuständen nicht nur etwa n^3 Ausdrücke konstruieren, sondern die Länge des Ausdrucks kann in jedem der n Induktionsschritte durchschnittlich um den Faktor 4 wachsen, falls sich die Ausdrücke nicht vereinfachen lassen. Folglich können die Ausdrücke selbst eine Größenordnung von 4^n Symbolen erreichen.

Mit einem ähnlichen Ansatz lässt es sich vermeiden, an gewissen Punkten dieselben Arbeitsschritte wiederholt ausführen zu müssen. Beispielsweise wird in der Konstruktion von Theorem 3.4 in allen Ausdrücken mit dem oberen Index $(k + 1)$ derselbe Teilausdruck $R_{kk}^{(k)}$ verwendet, folglich muss dieser Ausdruck n^2-mal niedergeschrieben werden.

Der Ansatz zur Bildung regulärer Ausdrücke, den wir nun kennen lernen werden, beinhaltet das Eliminieren von Zuständen. Wenn wir einen Zustand s eliminieren, dann werden damit auch sämtliche Pfade, die durch s verliefen, aus dem Automaten entfernt. Soll sich die Sprache des Automaten nicht ändern, dann müssen wir die Beschriftungen von Pfaden, die von einem Zustand q über s zu einem Zustand p führten, Pfeilen hinzufügen, die direkt von q nach p führen. Da die Beschriftung dieses Pfeils nun unter Umständen Zeichenreihen statt einzelner Symbole sind und da es sogar eine unendliche Anzahl solcher Zeichenreihen geben kann, können wir diese Zeichenreihen nicht einfach als Beschriftung auflisten. Glücklicherweise gibt es eine einfache, endliche Möglichkeit, alle derartigen Zeichenreihen darzustellen: reguläre Ausdrücke.

Wir werden daher Automaten betrachten, die als Beschriftungen reguläre Ausdrücke verwenden. Die Sprache des Automaten ist die Vereinigung aller Pfade, die vom Startzustand zu einem akzeptierenden Zustand der Sprache führen, die gebildet wird, indem die Sprachen der auf dem Pfad liegenden regulären Ausdrücke miteinander verkettet werden. Beachten Sie, dass diese Regel konsistent ist mit der Definition der Sprache eines jeden der verschiedenen Automaten, die wir bislang betrachtet haben. Man kann sich jedes Symbol a oder ε, falls zulässig, auch als regulären Ausdruck vorstellen, dessen Sprache aus einer einzigen Zeichenreihe, nämlich $\{a\}$ oder $\{\varepsilon\}$, besteht. Wir können diese Beobachtung als Grundlage eines Verfahrens zum Eliminieren von Zuständen betrachten, das wir als Nächstes beschreiben.

Abbildung 3.5 zeigt einen generischen Zustand s, der eliminiert werden soll. Wir nehmen an, dass der Automat, zu dem der Zustand s gehört, über die s vorausgehenden Zustände $q_1, q_2, ..., q_k$ und die s nachfolgenden Zustände $p_1, p_2, ..., p_m$ verfügt. Es ist möglich, dass einige q auch p sind, aber wir unterstellen, dass s nicht zu den Zustän-

den q oder p gehört, auch wenn, wie in Abbildung 3.5 dargestellt, von s ein Pfeil zurück zu s führt. Wir zeigen zudem einen regulären Ausdruck an jedem Pfeil von einem der q-Zustände zu s. Der Ausdruck Q_i dient als Beschriftung des von q_i ausgehenden Pfeils. Analog zeigen wir den regulären Ausdruck P_j, der den Pfeil von s nach p_j für alle j beschriftet. An s befindet sich ein Pfeil mit der Beschriftung S, der zurück zu s führt. Schließlich gibt es für alle i und j den regulären Ausdruck R_{ij} an dem Pfeil, der von q_i nach p_j führt. Beachten Sie, dass einige dieser Pfeile im Automaten möglicherweise nicht vorhanden sind. Ist dies der Fall, dann nehmen wir an, dass der Ausdruck an dem Pfeil gleich ∅ ist.

Abbildung 3.5: Ein Zustand s, der eliminiert werden soll

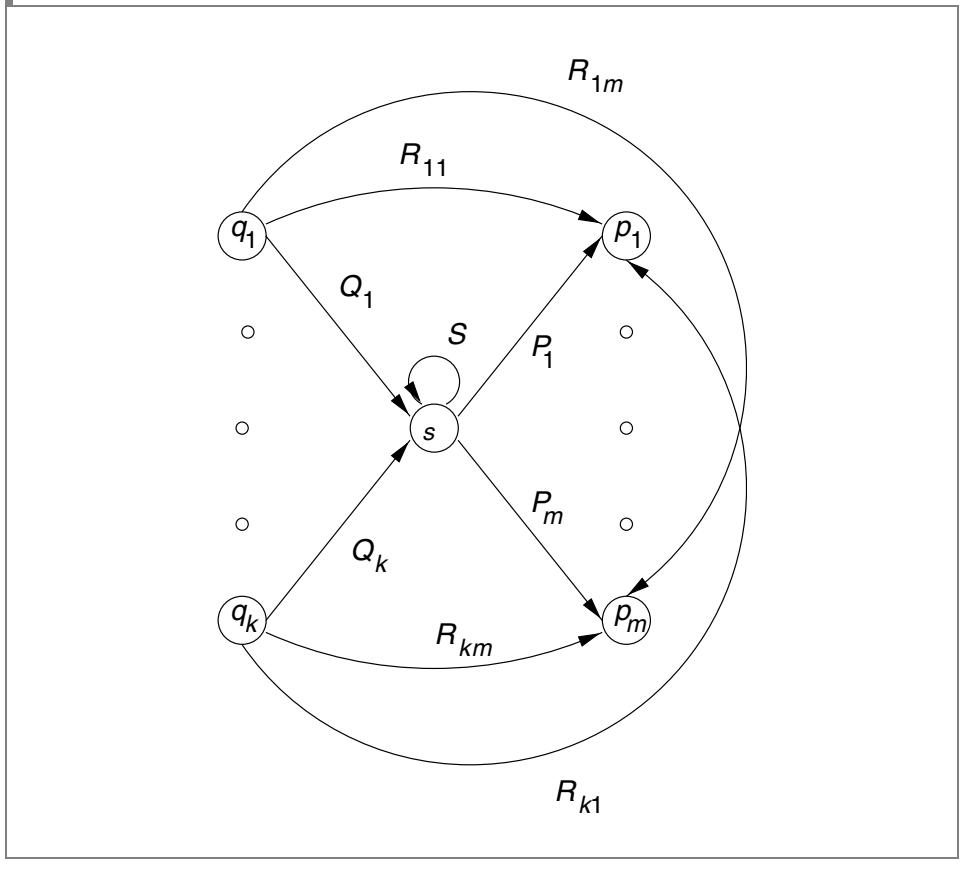
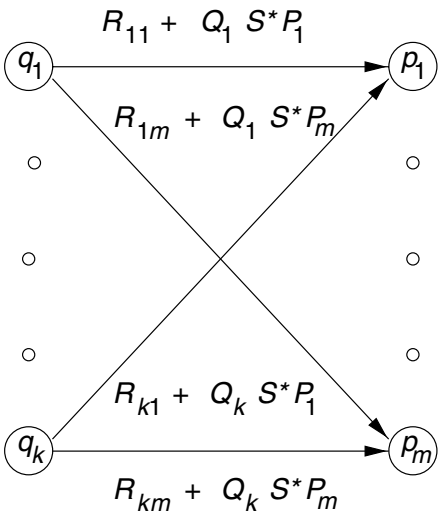


Abbildung 3.6 zeigt, was passiert, wenn wir den Zustand s eliminieren. Alle Pfeile, die vom oder zum Zustand s führen, wurden gelöscht. Zur Kompensierung führen wir für jeden Vorgängerzustand q_i von s sowie für jeden Nachfolgerzustand p_j von s einen regulären Ausdruck ein, der all jene Pfade repräsentiert, die bei q_i beginnen, zu s führen, s null oder mehrere Male in einer Schleife berühren und schließlich zu p_j führen. Der Ausdruck für diese Pfade lautet $Q_i S^* P_j$. Dieser Ausdruck wird (mit dem Mengenvereinigungsoperator) zu dem Pfeil, der von q_i nach p_j führt, hinzugefügt. Falls es keinen Pfeil $q_i \rightarrow p_j$ gibt, dann führen wir mit dem regulären Ausdruck ∅ einen ein.

Abbildung 3.6: Ergebnis der Eliminierung von Zustand *s* aus Abbildung 3.5

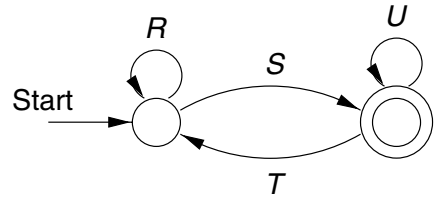


Folgende Strategie wird zur Bildung eines regulären Ausdrucks auf der Grundlage eines endlichen Automaten angewandt:

1. Wende für jeden akzeptierenden Zustand q das oben beschriebene Reduktionsverfahren an, um einen äquivalenten Automaten zu erzeugen, dessen Pfeile mit regulären Ausdrücken beschriftet sind. Eliminiere alle Zustände, mit Ausnahme von q und dem Startzustand q_0.

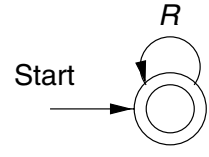
2. Wenn $q \neq q_0$, dann sollten wir einen zwei Zustände umfassenden Automaten erhalten, der wie der Automat in Abbildung 3.7 aussieht. Der reguläre Ausdruck für die akzeptierenden Zeichenreihen kann auf verschiedene Arten beschrieben werden. Eine Möglichkeit lautet $(R + SU^*T)^*SU^*$. Zur Erläuterung: Wir können vom Startzustand beliebig oft zurück zum Startzustand gehen, indem wir einer Folge von Pfaden folgen, deren Beschriftungen entweder in $L(R)$ oder in $L(SU^*T)$ enthalten sind. Der Ausdruck SU^*T repräsentiert Pfade, die über einen in $L(S)$ enthaltenen Pfad zum akzeptierenden Zustand führen, möglicherweise unter Verwendung einer Folge von Pfaden, deren Beschriftungen in $L(U)$ enthalten sind, mehrmals zum akzeptierenden Zustand zurückführen und dann über einen Pfad, dessen Beschriftungen in $L(T)$ enthalten sind, zum Startzustand zurückkehren. Wir müssen dann zum akzeptierenden Zustand gelangen, wobei wir nicht mehr zum Startzustand zurückkehren dürfen, indem wir einem Pfad folgen, dessen Beschriftungen in $L(S)$ enthalten sind. Sobald der akzeptierende Zustand erreicht ist, können wir beliebig oft zu ihm zurückkehren, indem wir einem Pfad folgen, dessen Beschriftungen in $L(U)$ enthalten sind.

Abbildung 3.7: Ein generischer, zwei Zustände umfassender Automat



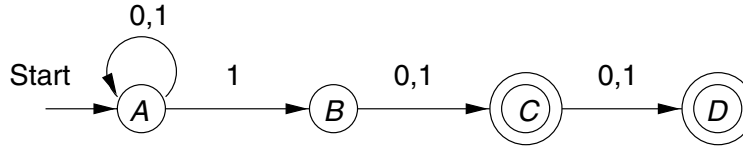
3. Falls der Startzustand zugleich ein akzeptierender Zustand ist, müssen wir zudem eine Zustandseliminierung am ursprünglichen Automaten durchführen, die jeden Zustand außer dem Startzustand beseitigt. Ergebnis dieser Eliminierung ist ein Automat, der über einen Zustand verfügt und wie der in Abbildung 3.8 dargestellte aussieht. Der reguläre Ausdruck, der die Zeichenreihen beschreibt, die dieser Automat akzeptiert, lautet R^*.

Abbildung 3.8: Ein generischer Automat mit einem Zustand



4. Der gewünschte reguläre Ausdruck ist die Summe (Vereinigung) aller Ausdrücke, die aus den reduzierten Automaten nach den Regeln (2) und (3) für jeden akzeptierenden Zustand abgeleitet wurden.

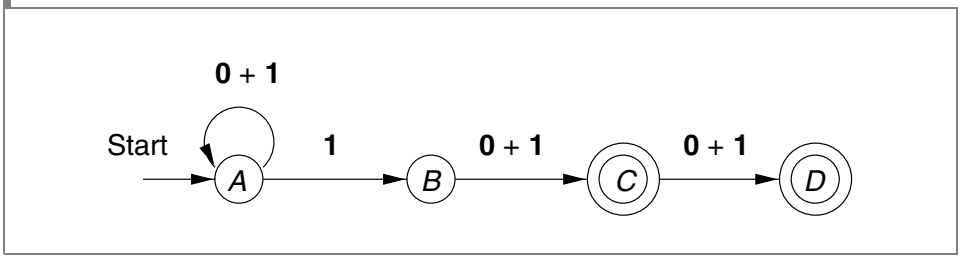
Abbildung 3.9: Ein NEA, der Zeichenreihen akzeptiert, an deren zweiter oder dritter Position vor dem Ende eine 1 steht



Beispiel 3.6 Wir wollen den NEA aus Abbildung 3.9 betrachten, der alle Zeichenreihen aus Nullen und Einsen akzeptiert, in denen entweder an der zweiten oder an der dritten Position vor dem Ende eine Eins steht. Unser erster Schritt besteht darin, diesen Automaten in einen Automaten mit regulären Ausdrücken als Beschriftungen

umzuwandeln. Da keine Zustandseliminierung durchgeführt wurde, müssen wir lediglich die Beschriftungen »0,1« durch die äquivalenten Ausdrücke **0** + **1** ersetzen. Das Ergebnis ist in Abbildung 3.10 dargestellt.

Abbildung 3.10: Der Automat aus Abbildung 3.9 mit regulären Ausdrücken als Beschriftungen



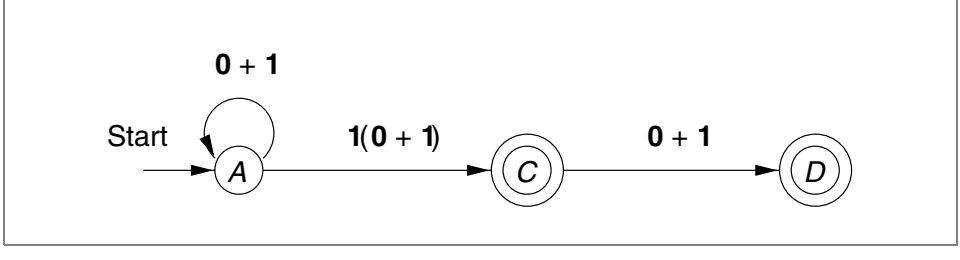
Wir wollen zuerst Zustand *B* eliminieren. Da es sich bei diesem Zustand weder um einen akzeptierenden Zustand noch um den Startzustand handelt, kommt er in den reduzierten Automaten nicht vor. Folglich sparen wir uns etwas Arbeit, wenn wir ihn zuerst eliminieren, bevor wir die beiden reduzierten Automaten entwickeln, die den beiden akzeptierenden Zuständen entsprechen.

Zustand *B* hat einen Vorgänger namens *A* und einen Nachfolger namens *C*. Diese Zustände werden durch die folgenden regulären Ausdrücke (vgl. Abbildung 3.5) beschrieben: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0} + \mathbf{1}$, $R_{11} = \varnothing$ (da kein Pfeil von *A* nach *C* führt) und $S = \varnothing$ (weil kein Pfeil von Zustand *B* zurück zu *B* führt). Daraus ergibt sich der Ausdruck, der als Beschriftung des neuen Pfeils von *A* nach *C* dient: $\varnothing + \mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$.

Um diesen Ausdruck zu vereinfachen, eliminieren wir zuerst die voranstehende $\varnothing$, da die leere Menge in einer Vereinigung ignoriert werden kann. Der Ausdruck lautet nun $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$. Beachten Sie, dass der reguläre Ausdruck $\varnothing^*$ gleichbedeutend ist mit dem regulären Ausdruck ε, da $L(\varnothing^*) = \{\varepsilon\} \cup L(\varnothing) \cup L(\varnothing)\,L(\varnothing) \cup \ldots$

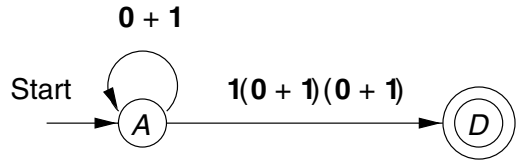
Da alle Terme außer dem ersten leer sind, erkennen wir, dass $L(\varnothing^*) = \{\varepsilon\}$, was dasselbe wie $L(\varepsilon)$ ist. Folglich ist $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$ gleichbedeutend mit $\mathbf{1}(\mathbf{0} + \mathbf{1})$, was dem Ausdruck entspricht, der in Abbildung 3.11 die Beschriftung des Pfeils $A \rightarrow C$ bildet.

Abbildung 3.11: Eliminierung von Zustand B



Nun müssen wir verzweigen und die Zustände *C* und *D* in jeweils getrennten Reduktionen eliminieren. Zur Eliminierung von Zustand *C* gehen wir ähnlich vor, wie oben zur Eliminierung von Zustand *B* beschrieben, und der resultierende Automat ist in Abbildung 3.12 dargestellt.

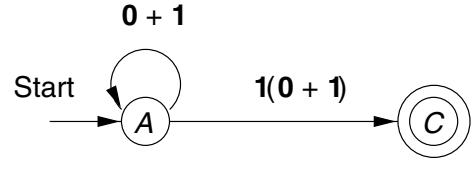
Abbildung 3.12: Ein Automat mit den Zuständen *A* und *D*



In der Notation des generischen Automaten mit zwei Zuständen aus Abbildung 3.7 lauten die regulären Ausdrücke für den Automaten in Abbildung 3.12: $R = \mathbf{0} + \mathbf{1}$, $S = \mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$, $T = \emptyset$ und U = $\emptyset$. Der Ausdruck U^* kann durch ε ersetzt werden, d. h. in einer Verkettung eliminiert werden. Begründen lässt sich dies damit, dass $\emptyset^* = \varepsilon$, wie oben erläutert. Zudem ist der Ausdruck SU^*T gleichbedeutend mit $\emptyset$, da T, einer der Terme in der Verkettung, gleich $\emptyset$ ist. Der allgemeine Ausdruck $(R + SU^*T)^*SU^*$ lässt sich in diesem Fall daher zu R^*S oder $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$ vereinfachen. Informell ausgedrückt, beschreibt dieser Automat eine Sprache, die alle Zeichenreihen aus Nullen und Einsen enthält, an deren drittletzter Stelle eine 1 steht. Diese Sprache stellt einen Teil der Zeichenreihen dar, die der Automat aus Abbildung 3.9 akzeptiert.

Wir müssen nun wieder bei Abbildung 3.11 beginnen, und den Zustand *D* statt *C* eliminieren. Da *D* keine Nachfolger hat, können wir aus Abbildung 3.5 ersehen, dass sich keine Pfadänderungen ergeben, und der Pfeil von *C* nach *D* wird zusammen mit dem Zustand *D* eliminiert. Der resultierende zwei Zustände umfassende Automat ist in Abbildung 3.13 dargestellt.

Abbildung 3.13: Zwei Zustände umfassender Automat, der aus der Elimination von *D* resultiert



Dieser Automat unterscheidet sich von dem in Abbildung 3.12 dargestellten Automaten nur durch die Beschriftung des vom Startzustand zum akzeptierenden Zustand führenden Pfeils. Wir können daher die Regel für zwei Zustände umfassende Automaten anwenden und den Ausdruck vereinfachen, womit wir $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ erhalten. Dieser Ausdruck repräsentiert den zweiten Typ von Zeichenreihe, den der Automat akzeptiert: Zeichenreihen mit einer Eins an der zweiten Position vor dem Ende.

Jetzt müssen wir die beiden Ausdrücke nur noch addieren, um den Ausdruck für den gesamten in Abbildung 3.9 dargestellten Automaten zu erhalten. Dieser Ausdruck lautet:

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

■

Die Reihenfolge der Eliminierung von Zuständen festlegen

Wie wir in Beispiel 3.6 gesehen haben, wird ein Zustand, der weder ein Start- noch ein Akzeptanzzustand ist, aus allen abgeleiteten Automaten eliminiert. Folglich besteht einer der Vorteile des Zustandseliminierungsverfahrens gegenüber der mechanischen Erzeugung von regulären Ausdrücken, die wir in Abschnitt 3.2.1 beschrieben haben, darin, dass wir zuerst alle Zustände, die weder Start- noch akzeptierender Zustand sind, eliminieren und damit ein für alle Mal tilgen können. Wir müssen lediglich dann erneut reduzieren, wenn wir akzeptierende Zustände eliminieren müssen.

Auch dann können wir einige Arbeitsschritte zusammenfassen. Wenn es beispielsweise drei akzeptierende Zustände p, q und r gibt, können wir p eliminieren und dann verzweigen, um entweder q oder r zu eliminieren, und somit den Automaten für den akzeptierenden Zustand r bzw. q erzeugen. Wir können dann erneut mit allen drei akzeptierenden Zuständen beginnen und sowohl q als auch r eliminieren, um den Automaten für p zu erhalten.

3.2.3 Reguläre Ausdrücke in Automaten umwandeln

Wir werden nun den in Abbildung 3.1 skizzierten Plan vervollständigen, indem wir zeigen, dass jede Sprache $L = L(R)$ für einen regulären Ausdruck R auch $L = L(E)$ für einen ε-NEA E ist. Der Beweis besteht aus einer strukturellen Induktion über den Ausdruck R. Wir beginnen, indem wir zeigen, wie der Automat für die Anfangsausdrücke ε und $\emptyset$ konstruiert wird. Wir zeigen anschließend, wie diese Automaten zu größeren Automaten zusammengefasst werden, die die Vereinigung, Verkettung und Hülle der von den kleineren Automaten akzeptierten Sprachen akzeptieren.

Alle dabei konstruierten Automaten sind ε-NEAs mit einem einzigen akzeptierenden Zustand.

Satz 3.7 Jede Sprache, die durch einen regulären Ausdruck definiert wird, wird auch durch einen endlichen Automaten definiert.

Beweis: Angenommen, $L = L(R)$ für einen regulären Ausdruck R. Wir zeigen, dass $L = L(E)$ für einen ε-NEA E mit

1. genau einem akzeptierenden Zustand
2. keinen Pfeilen zum Startzustand
3. keinen Pfeilen, die vom akzeptierenden Zustand ausgehen

Der Beweis wird durch strukturelle Induktion über R erbracht, wobei wir der induktiven Definition von regulären Ausdrücken, die in Abschnitt 3.1.2. vorgestellt wurde, folgen.

Induktionsbeginn: Er gliedert sich in drei Teile, die in Abbildung 3.14 dargestellt sind. In Teil (a) wird gezeigt, wie der Ausdruck ε zu behandeln ist. Es lässt sich einfach zeigen, dass die Sprache des Automaten $\{\varepsilon\}$ ist, da der einzige Pfad vom Startzustand zu einem akzeptierenden Zustand mit ε beschriftet ist. Teil (b) zeigt die Konstruktion für $\emptyset$. Offensichtlich gibt es keine Pfade vom Startzustand zum akzeptierenden Zustand, und daher ist $\emptyset$ die Sprache dieses Automaten. Schließlich zeigt Teil (c) den Automaten für einen regulären Ausdruck **a**. Die Sprache dieses Automaten besteht aus der einen Zeichenreihe a oder, anders ausgedrückt, $L(\mathbf{a})$. Es ist leicht nachprüfbar, dass diese Automaten die Bedingungen (1), (2) und (3) der Induktionshypothese erfüllen.

Abbildung 3.14: Beginn der Konstruktion eines Automaten aus einem regulären Ausdruck

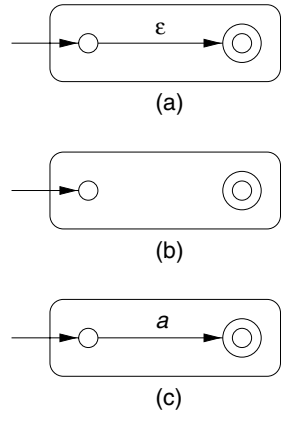
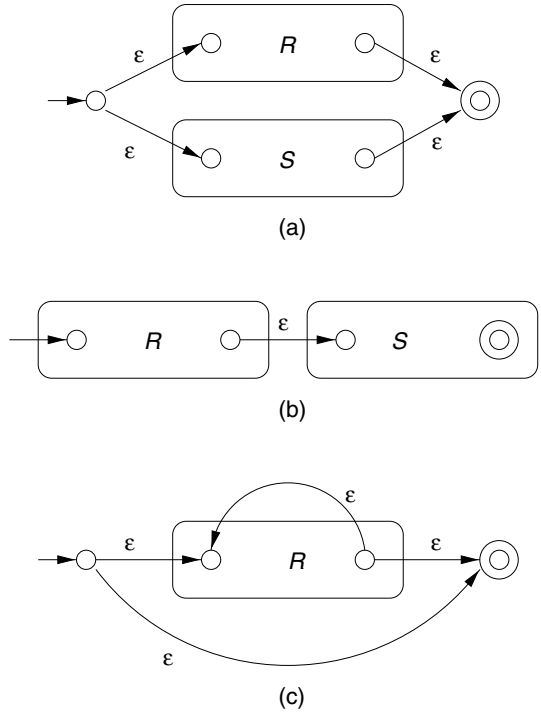


Abbildung 3.15: Der Induktionsschritt in der Konstruktion eines ε-NEA aus einem regulären Ausdruck



INDUKTIONSSCHRITT: Diese drei Teile der Induktion sind in Abbildung 3.15 dargestellt. Wir nehmen an, dass die Aussage des Satzes für die unmittelbaren Teilausdrücke eines gegebenen regulären Ausdrucks wahr ist; das heißt, dass die Sprachen dieser Teilausdrücke auch Sprachen von ε-NEAs mit einem einzigen akzeptierenden Zustand sind. Es sind vier Fälle zu unterscheiden:

1. Der Ausdruck lautet $R + S$ für beliebige kleinere Ausdrücke R und S. Auf diesen Fall trifft der Automat in Abbildung 3.15 (a) zu. Das heißt, ausgehend von einem neuen Startzustand können wir sowohl zum Startzustand des Automaten für R als auch zu dem des Automaten für S gehen. Wir erreichen den akzeptierenden Zustand des jeweiligen Automaten, indem wir einem Pfad folgen, der mit einer Zeichenreihe aus $L(R)$ bzw. $L(S)$ beschriftet ist. Sobald wir den akzeptierenden Zustand von R bzw. S erreicht haben, können wir einem der ε-Pfeile zum akzeptierenden Zustand des neuen Automaten folgen. Folglich lautet die Sprache des Automaten aus Abbildung 3.15 (a) $L(R) \cup L(S)$.

2. Der Ausdruck lautet RS für beliebige kleinere Ausdrücke R und S. Der Automat für die Verkettung ist in Abbildung 3.15 (b) dargestellt. Beachten Sie, dass der Startzustand des ersten Automaten zum Startzustand der Gesamtheit wird und dass der akzeptierende Zustand des zweiten Automaten zum akzeptierenden Zustand der Gesamtheit wird. Begründen lässt sich dies damit, dass die einzigen Pfade, die vom Start- zum akzeptierenden Zustand führen, zuerst den Automaten für R passieren, wo sie einem Pfad folgen müssen, der mit einer in $L(R)$ enthaltenen Zeichenreihe beschriftet ist, und dann den Automaten für S durchqueren, wo sie einem Pfad folgen müssen, der mit einer in $L(S)$ enthaltenen Zeichenreihe beschriftet ist. Daher hat der in Abbildung 3.15 (b) dargestellte Automat genau die Pfade, die mit Zeichenreihen aus $L(R)L(\mathrm{S})$ beschriftet sind.

3. Der Ausdruck lautet R^* für einen kleineren Ausdruck R. In diesem Fall verwenden wir den Automaten aus Abbildung 3.15 (c). Dieser Automat hat zwei mögliche Pfade zur Auswahl:

   a) Direkt vom Startzustand zum akzeptierenden Zustand entlang eines mit ε beschrifteten Pfades. Dieser Pfad ermöglicht es, ε zu akzeptieren, das für jeden beliebigen Ausdruck R in $L(R^*)$ enthalten ist.

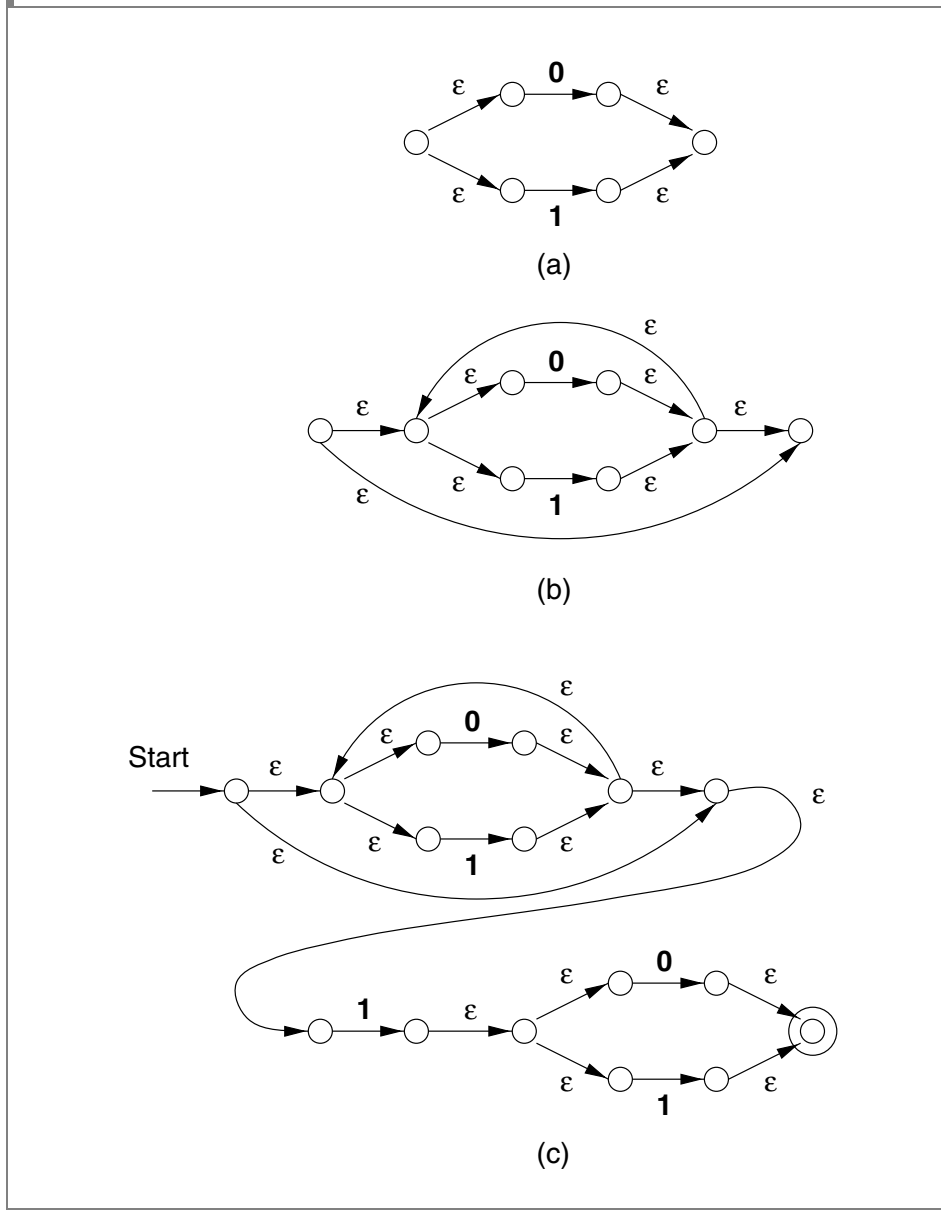
   b) Zum Startzustand des Automaten für R, einmal oder mehrmals durch diesen Automaten und dann zum akzeptierenden Zustand. Diese Menge von Pfaden erlaubt es, Zeichenreihen zu akzeptieren, die in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$ etc. enthalten sind, womit alle Zeichenreihen in $L(R^*)$ abgedeckt sind, womöglich mit Ausnahme von ε, das durch den unter (3a) beschriebenen direkten Pfad zum akzeptierenden Zustand abgedeckt wird.

4. Der Ausdruck lautet (R) für einen kleineren Ausdruck R. Der Automat für R dient auch als Automat für (R), da Klammern die durch den Automaten definierte Sprache nicht ändern.

Es ist eine einfache Beobachtung, dass der konstruierte Automat die drei in der Induktionsannahme gegebenen Bedingungen erfüllt – ein akzeptierender Zustand, keine auf den Startzustand hinweisenden oder von dem akzeptierenden Zustand ausgehenden Pfeile. ■

Beispiel 3.8 Wir wollen nun den regulären Ausdruck $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ in einen ε-NEA umwandeln. Der erste Schritt besteht in der Konstruktion eines Automaten für $\mathbf{0} + \mathbf{1}$. Wir verwenden zwei Automaten, die gemäß Abbildung 3.14 (c) aufgebaut wurden, einen mit der Beschriftung **0** auf dem Pfeil und der andere mit der Beschriftung **1**. Diese beiden Automaten werden dann mit Hilfe der Vereinigungskonstruktion aus Abbildung 3.15 (a) kombiniert. Das Ergebnis ist in Abbildung 3.16 (a) zu sehen.

Abbildung 3.16: Für Beispiel 3.8 erzeugter Automat



Als Nächstes wenden wir die Sternkonstruktion aus Abbildung 3.15 (c) auf Abbildung 3.16 (a) an. Dieser Automat ist in Abbildung 3.16 (b) dargestellt. Die letzten beiden Schritte beinhalten die Anwendung der Verkettungskonstruktion aus Abbildung 3.15 (b). Zuerst verbinden wir den Automaten aus Abbildung 3.16 (b) mit einem anderen Automaten, der nur die Zeichenreihe 1 akzeptieren soll. Dieser Automat stellt eine weitere Anwendung der Grundkonstruktion aus Abbildung 3.14 (c) dar, wobei der Pfeil die Beschriftung **1** trägt. Beachten Sie, dass wir einen neuen Automaten erstellen müssen, damit 1 erkannt wird; der Automat aus Abbildung 3.16 (a), der **1** akzeptierte, darf hier nicht verwendet werden. Bei dem dritten Automaten in der Verkettung handelt es sich um einen anderen Automaten für $\mathbf{0} + \mathbf{1}$. Wieder müssen wir eine Kopie des Automaten aus Abbildung 3.16 (a) erstellen; hier darf nicht die Kopie verwendet werden, die Bestandteil von Abbildung 3.16 (b) ist. Der vollständige Automat ist in Abbildung 3.16 (c) dargestellt.

Beachten Sie, dass dieser ε-NEA, wenn die ε-Übergänge eliminiert werden, genauso aussieht wie der Automat aus Abbildung 3.13, der auch die Zeichenreihen akzeptiert, an deren zweitletzter Position eine 1 steht. ■

3.2.4 Übungen zum Abschnitt 3.2

Übung 3.2.1 Die Übergangstabelle eines DEA sehe wie folgt aus:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(0)}$. *Hinweis*: Stellen Sie sich den Zustand q_i so vor, als handle es sich um den Zustand mit der ganzzahligen Nummer i.

* b) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(1)}$. Versuchen Sie, die Ausdrücke so weit wie möglich zu vereinfachen.

c) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(2)}$. Versuchen Sie, die Ausdrücke so weit wie möglich zu vereinfachen.

d) Formulieren Sie einen regulären Ausdruck für die Sprache des Automaten.

* e) Konstruieren Sie das Übergangsdiagramm für den DEA, und geben Sie einen regulären Ausdruck für dessen Sprache an, indem Sie Zustand q_2 eliminieren.

Übung 3.2.2 Wiederholen Sie Übung 3.2.1 für den folgenden DEA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Beachten Sie, dass Lösungen zu den Teilen (a), (b) und (e) für diese Übung (auf den Internetseiten dieses Buches) *nicht* verfügbar sind.

Übung 3.2.3 Wandeln Sie folgenden DEA in einen regulären Ausdruck um, und verwenden Sie hierbei die in Abschnitt 3.2.2 beschriebene Technik zur Zustandselimination:

	0	1
$\rightarrow * p$	s	p
q	p	s
r	r	q
s	q	r

Übung 3.2.4 Wandeln Sie die folgenden regulären Ausdrücke in NEAs mit ε-Übergängen um:

* a) $\mathbf{01}^*$.

b) $(\mathbf{0} + \mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0} + \mathbf{1})^*$.

Übung 3.2.5 Eliminieren Sie die ε-Übergänge aus Ihrem ε-NEA aus Übung 3.2.4. Eine Lösung zu Teil (a) erscheint auf den Webseiten zu diesem Buch.

! **Übung 3.2.6** Sei $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ ein ε-NEA, derart dass es keine zu q_0 hinführenden und keine von q_f ausgehenden Übergänge gibt. Beschreiben Sie für jede der folgenden Modifikationen von A die akzeptierte Sprache als Modifikation von $L = L(A)$:

* a) Der Automat, der aus A konstruiert wird, indem ein ε-Übergang von q_f nach q_0 hinzugefügt wird

* b) Der Automat, der aus A konstruiert wird, indem ε-Übergänge von q_0 zu jedem Zustand hinzugefügt werden, der von q_0 aus erreichbar ist (auf einem Pfad, dessen Beschriftungen Symbole aus Σ sowie ε enthalten können)

c) Der Automat, der aus A konstruiert wird, indem von jedem Zustand ε-Übergänge nach q_f hinzugefügt werden, von dem aus q_f auf irgendeinem Pfad erreichbar ist

d) Der Automat, der aus A konstruiert wird, indem die unter (b) und (c) geforderten Modifikationen ausgeführt werden

***!! Übung 3.2.7** Die Konstruktionen von Satz 3.7, mit denen wir einen regulären Ausdruck in einen ε-NEA umgewandelt haben, lassen sich auf verschiedene Weise vereinfachen. Folgende drei Vereinfachungen sind beispielsweise möglich:

1. Vereinigungsoperator: Statt einen neuen Start- und einen neuen akzeptierenden Zustand zu bilden, werden die beiden Startzustände in einen Zustand zusammengeführt, der über alle Übergänge aus beiden Startzuständen verfügt. Analog werden die beiden akzeptierenden Zustände zusammengeführt, wobei alle Übergänge in die beiden akzeptierenden Zustände dann zum zusammengeführten Zustand führen.
2. Verkettungsoperator: Der akzeptierende Zustand des ersten Automaten wird mit dem Startzustand des zweiten zusammengeführt.
3. Sternoperator: Es werden einfach ε-Übergänge vom akzeptierenden Zustand zum Startzustand und in der umgekehrten Richtung hinzugefügt.

Jede dieser Vereinfachungen ergibt für sich genommen eine korrekte Konstruktion: d. h. der für einen beliebigen Ausdruck resultierende ε-NEA akzeptiert die durch den Ausdruck definierte Sprache. Welche der Vereinfachungen (1), (2), (3) lassen sich kombinieren, sodass man aus diesen Kombinationen wiederum einen korrekten Automaten für jeden regulären Ausdruck erhält?

***!! Übung 3.2.8** Geben Sie einen Algorithmus an, der einen DEA A als Eingabe erhält und die Anzahl der Zeichenreihen mit der Länge n (für eine gegebene Zahl n, die unabhängig von der Anzahl der Zustände von A ist) berechnet, die von A akzeptiert werden. Ihr Algorithmus sollte sowohl in Bezug auf n als auch die Anzahl der Zustände von A polynomial sein. *Hinweis:* Setzen Sie die Technik ein, die von der Konstruktion im Beweis von Satz 3.4 nahe gelegt wird.

3.3 Anwendungen regulärer Ausdrücke

Reguläre Ausdrücke, die ein »Bild« des Musters darstellen, das erkannt werden soll, sind das geeignete Mittel für Anwendungen, die nach Mustern im Text suchen. Die regulären Ausdrücke werden dann im Hintergrund in deterministische oder nichtdeterministische Automaten kompiliert, die simuliert werden, um ein Programm zu erzeugen, das Muster im Text erkennt. In diesem Abschnitt betrachten wir zwei wichtige Klassen auf regulären Ausdrücken basierender Anwendungen: lexikalische Analysekomponenten und Textsuche.

3.3.1 Reguläre Ausdrücke in Unix

Bevor wir die Anwendungen betrachten, stellen wir die Unix-Notation für erweiterte reguläre Ausdrücke vor. Diese Notation stellt uns eine Reihe zusätzlicher Hilfsmittel zur Verfügung. Tatsächlich enthalten die Unix-Erweiterungen bestimmte Leistungs-

merkmale, z. B. die Fähigkeit, Zeichenreihen zu benennen und wieder aufzurufen, die mit einem Muster übereingestimmt haben, sodass sogar die Erkennung gewisser nichtregulärer Sprachen möglich ist. Wir werden diese Leistungsmerkmale hier nicht behandeln. Stattdessen stellen wir lediglich die Kürzel vor, die eine prägnante Darstellung komplexer regulärer Ausdrücke erlauben.

Die erste Erweiterung der Notation regulärer Ausdrücke betrifft die Tatsache, dass die meisten praktischen Anwendungen mit dem ASCII-Zeichensatz arbeiten. In unseren Beispielen wurden in der Regel kleine Alphabete verwendet, z. B. {0, 1}. Da es nur zwei Symbole gab, konnten wir knappe Ausdrücke wie $\mathbf{0} + \mathbf{1}$ für »beliebiges Zeichen« angeben. Wenn es allerdings 128 Zeichen gibt, dann würde derselbe Ausdruck eine Liste all dieser Zeichen beinhalten und seine Angabe wäre daher äußerst mühsam. Reguläre Ausdrücke im Unix-Stil erlauben es uns, mithilfe von *Zeichenklassen* umfangreiche Mengen von Zeichen so prägnant wie möglich darzustellen. Folgende Regeln gelten für Zeichenklassen:

- Das Symbol . (Punkt) steht für »ein beliebiges Zeichen«.
- Die Folge $[a_1 a_2 \ldots a_k]$ steht für den regulären Ausdruck

  $$a_1 + a_2 + \ldots + a_k$$

  Mit dieser Notation können wir uns die Hälfte der Zeichen sparen, da wir die Pluszeichen nicht angeben müssen. Beispielsweise können wir die vier Zeichen, die in den C-Vergleichsoperatoren vorkommen, darstellen durch `[<>=!]`.
- In eckigen Klammern können wir einen Bereich von Zeichen in der Form *x-y* angeben, um alle Zeichen von *x* bis *y* in der ASCII-Reihenfolge auszuwählen. Da die Ziffern ebenso wie die Groß- und Kleinbuchstaben einen ordnungserhaltenden numerischen Code besitzen, können wir viele Zeichenklassen, die für uns von Belang sind, mit wenigen Tastatureingaben darstellen. Beispielsweise können die Ziffern angegeben werden durch `[0-9]`, die Großbuchstaben durch `[A-Z]` und die Menge aller Buchstaben und Ziffern durch `[A-Za-z0-9]`. Wenn wir ein Minuszeichen in die Liste der Zeichen aufnehmen wollen, dann können wir es als erstes oder letztes Zeichen angeben, sodass es nicht mit seiner Bedeutung als Operator zur Definition eines Zeichenbereichs verwechselt wird. Die Menge der Ziffern plus Komma, Plus- und Minuszeichen, die zur Bildung von Dezimalzahlen dient, kann beispielsweise beschrieben werden durch `[-+.0-9]`. Eckige Klammern und andere Zeichen, die eine besondere Bedeutung in der Unix-Notation für reguläre Ausdrücke haben, können als Zeichen dargestellt werden, indem man ihnen einen umgekehrten Schrägstrich (\) voranstellt.
- Für einige der gebräuchlichsten Zeichenklassen sind spezielle Schreibweisen definiert. Zum Beispiel:

  a) `[:digit:]` steht für die Menge der zehn Ziffern und ist gleichbedeutend mit `[0-9]`.[3]

3. Die Notation `[:digit:]` hat den Vorteil, dass sie auch bei Verwendung eines anderen Zeichencodes als ASCII, in dem die Ziffern keine aufeinander folgenden numerischen Codes besitzen, die Zeichen `[0123456789]` repräsentiert, während der Ausdruck `[0-9]` in diesem Fall die Zeichen repräsentieren würde, die einen Zeichencode zwischen 0 und 9 haben.

b) `[:alpha:]` steht ebenso wie `[A-Za-z]` für die Menge der alphabetischen Zeichen.

c) `[:alnum:]` steht für die Menge aller Ziffern und Buchstaben (alphabetische und numerische Zeichen) genau wie `[A-Za-z0-9]`.

Zudem werden in der Unix-Notation für reguläre Ausdrücke einige Operatoren verwendet, die bislang noch nicht vorgekommen sind. Keiner dieser Operatoren stellt eine Erweiterung der Klasse der Sprachen dar, die sich durch reguläre Ausdrücke beschreiben lassen, aber sie erleichtern es gelegentlich, die gewünschte Sprache darzustellen:

1. Der Operator | wird an Stelle des Pluszeichens (+) zur Angabe einer Vereinigung verwendet.
2. Der Operator `?` bedeutet »null oder ein Vorkommen von«. Daher hat der Ausdruck *R?* in Unix dieselbe Bedeutung wie der Ausdruck $\varepsilon + R$ in der in diesem Buch verwendeten Notation für reguläre Ausdrücke.
3. Der Operator + bedeutet »ein oder mehr Vorkommen von«. Daher ist der Unix-Ausdruck *R*+ ein Kürzel für den Ausdruck RR^* in unserer Notation.
4. Der Operator {*n*} bedeutet »*n* Exemplare von«. Daher ist der Unix-Ausdruck *R*{5} ein Kürzel für den Ausdruck *RRRRR* in unserer Notation.

Beachten Sie, dass auch in der Unix-Notation für reguläre Ausdrücke Teilausdrücke durch runde Klammern gruppiert werden können, wie die in Abschnitt 3.1.2 beschriebenen regulären Ausdrücke, und dass dieselbe Operatorvorrangfolge gilt (wobei ?, + und {*n*}, was die Auswertungsreihenfolge betrifft, wie * behandelt werden). Der Sternoperator hat in Unix dieselbe Bedeutung wie die in diesem Buch angegebene (in Unix aber nicht als oberer Index geschrieben).

3.3.2 Lexikalische Analyse

Eine der ältesten Anwendungen regulärer Ausdrücke bestand in der Spezifikation einer Compilerkomponente, die als »lexikalische Analyse« bezeichnet wird. Diese Komponente liest das Quelltextprogramm ein und erkennt alle *Token*, d. h. jene Teilzeichenreihen aus aufeinander folgenden Zeichen, die eine logische Einheit bilden. Schlüsselwörter und Bezeichner sind übliche Beispiele für Token, stellen aber nur eine kleine Auswahl dar.

> **Umfassende Informationen zur Unix-Notation für reguläre Ausdrücke**
>
> Leser, die an einer kompletten Liste der in der Unix-Notation für reguläre Ausdrücke verfügbaren Operatoren und Kürzel interessiert sind, sollten die Manualseiten zu den Kommandos konsultieren. Die verschiedenen Unix-Versionen unterscheiden sich leicht, aber mit einem Kommando wie `man grep` werden Sie die Notation für das grundlegende Kommando `grep` angezeigt bekommen. »Grep« steht für »Global (search for) Regular Expression and Print«, übersetzt: »Globale (Suche nach) regulärem Ausdruck und Ausgabe«.

Der Unix-Befehl `lex` und dessen GNU-Version `flex` akzeptieren eine Liste regulärer Ausdrücke im Unix-Stil, denen jeweils ein in eckige Klammern gesetzter Codeabschnitt folgt, der angibt, was die Analysekomponente tun soll, wenn sie ein Vorkommen des betreffenden Token findet. Eine derartige Programmfunktion wird *lexical-analyzer generator* (übersetzt: Generator einer lexikalischen Analysekomponente) genannt, weil sie eine Beschreibung einer lexikalischen Analysekomponente als Eingabe erhält und daraus eine Funktion generiert, die wie eine lexikalische Analysekomponente arbeitet.

Kommandos wie `lex` und `flex` haben sich als äußerst nützlich erwiesen, weil die Notation der regulären Ausdrücke genau so mächtig ist, wie es zur Beschreibung von Token erforderlich ist. Diese Kommandos können mithilfe eines Verfahrens zur Umwandlung von regulären Ausdrücken in DEAs eine effiziente Funktion generieren, die Quelltextprogramme in Token aufschlüsselt. Mit ihnen wird die Implementierung einer lexikalischen Analyse zu einer in wenigen Stunden zu bewältigenden Aufgabe, während die manuelle Entwicklung der lexikalischen Analyse vor der Einführung dieser auf regulären Ausdrücken basierenden Hilfsmittel Monate dauern konnte. Wenn die lexikalische Analyse aus irgendeinem Grund modifiziert werden muss, lässt sich dies häufig durch die Änderung einiger weniger regulärer Ausdrücke erledigen, statt eine bestimmte Stelle in schwer verständlichem Code zu suchen und zu korrigieren.

Beispiel 3.9 Listing 3.17 zeigt einen Ausschnitt einer Eingabe für den Befehl `lex`, der einige Token aus der Programmiersprache C beschreibt. Die erste Zeile behandelt das Schlüsselwort `else`, und als Aktion wird festgelegt, dass eine symbolische Konstante (in diesem Beispiel `ELSE`) zur weiteren Verarbeitung an den Parser zurückgegeben wird. Die zweite Zeile enthält einen regulären Ausdruck, der Bezeichner beschreibt: ein Buchstabe, dem null oder mehr Buchstaben und/oder Ziffern folgen. Die Aktion besteht zunächst darin, den Bezeichner in die Symboltabelle einzutragen, sofern er dort noch nicht vorhanden ist, d. h. `lex` fügt den gefundenen Token in einen Puffer ein, und daher weiß das Programm genau, welcher Bezeichner gefunden wurde. Die lexikalische Analyse gibt schließlich die symbolische Konstante `ID` zurück, die in diesem Beispiel Bezeichner repräsentiert.

Listing 3.17: Beispiel für eine `lex`-Eingabe

```
else                  {return(ELSE);}
[A-Za-z][A-Za-z0-9]*  {Code, der den gefundenen Bezeichner
                       in die Symboltabelle einträgt;
                       return(ID);
                      }
>=                    [return(GE);}
=                     {return(EQ);}
...
```

Der dritte Eintrag in Listing 3.17 betrifft das Zeichen >=, einen zwei Zeichen umfassenden Operator. Als letztes Beispiel zeigen wir den Eintrag für das Zeichen =, einen aus einem Zeichen bestehenden Operator. In der Praxis wären Einträge für alle Schlüsselwörter, alle Operatorzeichen und Satzzeichen, wie Kommas und Klammern, und Familien von Konstanten wie Zahlen und Zeichenreihen vorhanden. Viele dieser Einträge sind sehr einfach; sie stellen nur eine Folge von einem oder mehreren speziellen Zeichen dar. Andere ähneln eher Bezeichnern, deren Beschreibung die gesamte Leistungsstärke der Notation regulärer Ausdrücke erfordert. Ganze Zahlen, Gleitkommazahlen, Zeichenreihen und Kommentare sind weitere Beispiele für Mengen von Zeichenreihen, die von der Verarbeitung regulärer Ausdrücke durch Befehle wie `lex` profitieren. ■

Die Umwandlung einer Sammlung von Ausdrücken, wie die in Listing 3.17 dargestellten, in einen Automaten erfolgt ungefähr so, wie wir es formal in den vorangegangenen Abschnitten beschrieben haben. Wir beginnen, indem wir einen Automaten für die Vereinigung aller Ausdrücke entwickeln. Dieser Automat kann im Grunde genommen nur aussagen, dass irgendein Token erkannt worden ist. Wenn wir allerdings die Konstruktion von Satz 3.7 für die Vereinigung der Ausdrücke nachvollziehen, dann gibt der Zustand des ε-NEA genau darüber Aufschluss, welcher Token erkannt wurde.

Problematisch ist hier nur, dass mehrere Token zugleich erkannt werden können. Beispielsweise entspricht die Zeichenreihe `else` nicht nur dem Schlüsselwort `else`, sondern auch dem Ausdruck für Bezeichner. Die Standardlösung besteht darin, dass der zuerst angegebene Ausdruck die Priorität erhält. Wenn Schlüsselwörter wie `else` also reserviert (nicht als Bezeichner verwendbar) sein sollen, dann geben wir sie einfach vor dem Ausdruck für Bezeichner an.

3.3.3 Textmuster finden

In Abschnitt 2.4.1 haben wir erstmals die Vorstellung beschrieben, dass Automaten eingesetzt werden könnten, um effizient nach einer Menge von Wörtern zu suchen, die in einer umfangreichen Textsammlung wie dem Internet enthalten sind. Obwohl die Werkzeuge und die Technologie hierfür noch nicht so weit entwickelt sind wie für die lexikalische Analyse, ist die Notation regulärer Ausdrücke eine wertvolle Hilfe zur Beschreibung der Suche nach interessanten Textmustern. Wie bei der lexikalischen Analyse kann man von der Fähigkeit, von der natürlichen, beschreibenden Notation regulärer Ausdrücke zu einer effizienten (auf Automaten basierenden) Implementierung zu gelangen, erheblich profitieren.

Ein allgemeines Problem, zu dessen Lösung sich die auf regulären Ausdrücken basierenden Techniken als nützlich erwiesen, besteht in der Beschreibung vage definierter Klassen von Textmustern. Die Vagheit der Beschreibung garantiert praktisch, dass wir das Muster anfangs nicht korrekt beschreiben – möglicherweise finden wir die ganz korrekte Beschreibung niemals. Durch den Einsatz der Notation regulärer Ausdrücke wird es einfach, das Muster mühelos abstrakt zu beschreiben und die Beschreibung rasch zu ändern, wenn etwas nicht stimmt. Ein »Compiler« für reguläre Ausdrücke ist hilfreich, um die formulierten Ausdrücke in ausführbaren Code umzuwandeln.

Wir wollen ein umfangreicheres Beispiel für diese Art von Problemen studieren, das sich in vielen Internetanwendungen stellt. Angenommen, wir möchten eine umfangreiche Menge von Webseiten nach Adressen durchsuchen. Wir möchten viel-

leicht einfach eine Mailingliste erstellen. Oder wir versuchen vielleicht, Unternehmen nach ihrem Standort zu klassifizieren, damit wir Abfragen wie »Finde das nächste Restaurant, das in einem Radius von zehn Autominuten von meinem aktuellen Aufenthaltsort liegt« beantworten können.
Wir werden uns im Folgenden auf das Erkennen amerikanischer Straßenangaben konzentrieren. Wie sieht eine solche Straßenangabe aus? Das müssen wir herausfinden, und wenn wir während des Testens der Software erkennen, dass wir einen Fall vergessen haben, dann müssen wir die Ausdrücke abändern, um dem Fehlenden Rechnung zu tragen. Wir beginnen mit der Feststellung, dass amerikanische Straßenangaben wahrscheinlich mit dem Wort »Street« oder dessen Abkürzung »St« enden. Es kommen allerdings auch die Bezeichnungen »Avenue« und »Road« sowie die entsprechenden Abkürzungen vor. Folglich können wir die Endung unseres regulären Ausdrucks etwa wie folgt formulieren:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

Wir haben im obigen Ausdruck die Unix-Notation verwendet und als Operator für die Mengenvereinigung daher den vertikalen Strich statt + angegeben. Beachten Sie zudem, dass den Punkten ein umgekehrter Schrägstrich vorangestellt ist, da der Punkt in Unix-Ausdrücken ein Sonderzeichen ist, das für »ein beliebiges Zeichen« steht. In diesem Fall wollen wir nur, dass der Punkt die Abkürzungen abschließt.

Einer Straßenbezeichnung wie `Street` muss der Straßenname voranstehen. In der Regel handelt es sich beim Namen um einen Großbuchstaben, dem einige Kleinbuchstaben folgen. Wir können dieses Muster mit dem Unix-Ausdruck **[A-Z][a-z]*** beschreiben. Manche Straßennamen bestehen allerdings aus mehreren Wörtern, z. B. Rhode Island Avenue in Washington DC. Nachdem wir erkannt haben, dass Angaben dieser Form nicht erkannt würden, ändern wir unsere Beschreibung von Straßennamen wie folgt ab:

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

Der obige Ausdruck beginnt mit einer Gruppe, die aus einem Großbuchstaben und null oder mehr Kleinbuchstaben besteht. Anschließend folgen null oder mehr Gruppen, die aus einem Leerzeichen, einem weiteren Großbuchstaben und null oder mehr Kleinbuchstaben bestehen. Das Leerzeichen ist in Unix-Ausdrücken ein gewöhnliches Zeichen. Damit der obige Ausdruck in einer Unix-Kommandozeile aber nicht so aussieht, als handele es sich um zwei durch ein Leerzeichen voneinander getrennte Ausdrücke, müssen wir den gesamten Ausdruck in Anführungszeichen setzen. Die Anführungszeichen sind nicht Teil des eigentlichen Ausdrucks.

Nun müssen wir die Hausnummer in die Adresse aufnehmen. Die meisten Hausnummern bestehen aus einer Folge von Ziffern. Manchen ist jedoch ein Buchstabe nachgestellt, wie in »123A Main St.«. Daher umfasst der Ausdruck, den wir für die Hausnummer formulieren, einen optionalen Großbuchstaben am Ende: `[0-9]+[A-Z]?`. Beachten Sie, dass wir den Unix-Operator + für »eine oder mehr« Ziffern und den Operator ? für »null oder einen« Großbuchstaben verwenden. Der gesamte Ausdruck, den wir zur Beschreibung der Straßenangabe entwickelt haben, lautet:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

Dieser Ausdruck sollte einigermaßen funktionieren. In der Praxis werden wir jedoch erkennen, dass Folgendes fehlt:

1. Straßen, die nicht als Street, Avenue oder Road bezeichnet werden. Beispielsweise fehlen hier die Bezeichnungen »Boulevard«, »Place«, »Way« und deren Abkürzungen
2. Straßennamen, die ganz oder teilweise aus Zahlen bestehen, wie »42nd Street«
3. Postfächer und andere Zustelladressen
4. Straßennamen, die nicht mit einer Bezeichnung wie »Street« enden. Ein Beispiel hierfür ist El Camino Real im Silicon Valley. Da dieser Name spanisch ist und »die königliche Straße« bedeutet, wäre »El Camino Real Street« redundant. Wir müssen also komplette Angaben wie »2000 El Camino Real« berücksichtigen.
5. Alle möglichen sonderbaren Sachen, die wir uns kaum vorstellen können. Sie etwa?

Nachdem wir einen Compiler für reguläre Ausdrücke haben, ist es viel leichter, langsam ein Modul zu entwickeln, das sämtliche Straßenangaben erkennt, als wenn wir jede Änderung direkt in einer konventionellen Programmiersprache vornehmen müssten.

3.3.4 Übungen zum Abschnitt 3.3

! **Übung 3.3.1** Geben Sie einen regulären Ausdruck an, der Telefonnummern in allen möglichen Varianten beschreibt. Berücksichtigen Sie internationale Nummern ebenso wie die Tatsache, dass die Vorwahlen und die lokalen Anschlussnummern in verschiedenen Ländern eine unterschiedliche Anzahl von Stellen besitzen.

!! **Übung 3.3.2** Geben Sie einen regulären Ausdruck an, der Gehaltsangaben repräsentiert, wie sie in Stellenanzeigen erscheinen. Beachten Sie, dass das Gehalt pro Stunde, pro Monat oder pro Jahr angegeben werden kann. Die Gehaltsangabe kann möglicherweise ein Währungssymbol oder eine andere Einheit wie »K« enthalten. In der näheren Umgebung der Gehaltsangabe befinden sich möglicherweise Wörter, die diese als solche identifizieren. Tipp: Sehen Sie sich Stellenanzeigen in der Zeitung oder Online-Joblistings an, um sich eine Vorstellung davon zu machen, welche Muster unter Umständen angebracht sind.

! **Übung 3.3.3** Am Ende von Abschnitt 3.3.3 haben wir einige mögliche Verbesserungen für Ausdrücke angeführt, die Straßenangaben beschreiben. Modifizieren Sie den dort entwickelten Ausdruck, sodass er alle aufgeführten Möglichkeiten abdeckt.

3.4 Algebraische Gesetze für reguläre Ausdrücke

In Beispiel 3.5 wurde verdeutlicht, dass es möglich sein muss, reguläre Ausdrücke zu vereinfachen, damit die Ausdrücke handhabbar bleiben. Wir haben dort einige Ad-hoc-Argumente dafür angeführt, warum ein Ausdruck durch einen anderen ersetzt werden kann. In allen Fällen ging es im Grunde genommen immer darum, dass die Ausdrücke in dem Sinn *äquivalent* waren, als sie dieselbe Sprache definierten. In die-

sem Abschnitt stellen wir eine Gruppe von algebraischen Gesetzen vor, nach denen sich die Frage der Äquivalenz von Ausdrücken auf einer etwas abstrakteren Ebene behandeln lässt. Statt bestimmte reguläre Ausdrücke zu untersuchen, betrachten wir Paare von regulären Ausdrücken mit Variablen als Argumente. Zwei Ausdrücke mit Variablen sind *äquivalent*, wenn wir die Variablen durch beliebige Sprachen ersetzen können und die beiden Ausdrücke auch dann stets dieselbe Sprache beschreiben.

Es folgt ein Beispiel für dieses Verfahren aus der Algebra der Arithmetik. Die Aussage 1 + 2 = 2 + 1 ist eine Sache. Dies ist ein Beispiel für das Kommutativgesetz der Addition, das sich leicht überprüfen lässt, indem der Additionsoperator auf beide Seiten angewandt und das Ergebnis 3 = 3 berechnet wird. Das *Kommutativgesetz der Addition* sagt allerdings mehr aus: Es besagt, $x + y = y + x$, wobei x und y Variablen sind, die durch zwei beliebige Zahlen ersetzt werden können. Das heißt, wir können zwei Zahlen in beliebiger Reihenfolge addieren und erhalten stets das gleiche Ergebnis, gleichgültig welche Zahlen wir verwenden.

Wie für arithmetische Ausdrücke gilt auch für reguläre Ausdrücke eine Reihe von Gesetzen. Viele dieser Gesetze ähneln den Gesetzen der Arithmetik, wenn man die Vereinigung von Mengen der Addition und die Verkettung der Multiplikation gleichsetzt. Es gibt jedoch einige Bereiche, in denen diese Analogie nicht stimmt, und es gibt auch einige Gesetze, die für reguläre Ausdrücke gelten und die keine Entsprechung in der Arithmetik haben; dies gilt insbesondere für den Sternoperator. Die nächsten Unterabschnitte bilden einen Katalog der wichtigsten Gesetze. Wir schließen diesen Abschnitt mit einer Erörterung der Verfahren, mit denen man prüfen kann, ob ein angenommenes Gesetz für reguläre Ausdrücke tatsächlich ein Gesetz ist, d. h. ob es für alle Sprachen gilt, die für die Variablen eingesetzt werden können.

3.4.1 Assoziativität und Kommutativität

Kommutativität ist die Eigenschaft eines Operators, die besagt, dass wir die Reihenfolge der Operanden vertauschen können, ohne dass sich das Ergebnis dadurch ändert. Oben wurde ein Beispiel aus der Arithmetik angeführt: $x + y = y + x$. *Assoziativität* ist die Eigenschaft eines Operators, die es zulässt, dass Operanden unterschiedlich gruppiert werden, wenn der Operator zweimal angewandt wird. Beispielsweise lautet das Assoziativgesetz der Multiplikation $(x \times y) \times z = x \times (y \times z)$. Die folgenden drei Gesetze ähnlicher Art gelten für reguläre Ausdrücke:

- $L + M = M + L$. Dieses Gesetz, das *Kommutativgesetz der Mengenvereinigung*, besagt, dass zwei Sprachen in beliebiger Reihenfolge vereinigt werden können.
- $(L + M) + N = L + (M + N)$. Dieses Gesetz, das *Assoziativgesetz der Mengenvereinigung*, besagt, dass drei Sprachen vereinigt werden können, indem man entweder zuerst die Vereinigung der ersten beiden oder zuerst die Vereinigung der letzten beiden bildet. Beachten Sie, dass wir aus der Kombination dieses Gesetzes mit dem Kommutativgesetz der Mengenvereinigung schließen können, dass jede beliebige Sammlung von Sprachen in beliebiger Reihenfolge und Gruppierung vereinigt werden kann, ohne dass sich das Ergebnis der Vereinigung ändert. Eine Zeichenreihe ist also genau dann in $L_1 \cup L_2 \cup ... \cup L_k$ enthalten, wenn sie in einer oder mehreren der Sprachen L_i enthalten ist.
- $(LM)N = L(MN)$. Dieses Gesetz, das *Assoziativgesetz der Verkettung*, besagt, dass drei Sprachen verkettet werden können, indem entweder die ersten beiden zuerst oder die letzten beiden zuerst verkettet werden.

In dieser Liste fehlt das »Gesetz« $LM = ML$, das besagen würde, dass die Verkettung kommutativ ist. Dieses Gesetz ist allerdings falsch.

Beispiel 3.10 Betrachten Sie die regulären Ausdrücke **01** und **10**. Diese Ausdrücke beschreiben die Sprachen {01} bzw. {10}. Da sich diese Sprachen unterscheiden, kann kein allgemeines Gesetz $LM = LM$ gelten. Wäre es gültig, könnten wir den regulären Ausdruck **0** für L und den Ausdruck **1** für M einsetzen und fälschlicherweise folgern, dass **01** = **10**. ■

3.4.2 Einheiten und Annihilatoren

Die *Einheit* bezüglich eines Operators ist ein Wert, für den gilt, dass bei Anwendung des Operators auf die Einheit und einen anderen Wert der andere Wert das Ergebnis bildet. Beispielsweise ist 0 die Einheit bezüglich der Addition, da $0 + x = x + 0 = x$, und 1 ist die Einheit bezüglich der Multiplikation, da $1 \times x = x \times 1 = x$. Der *Annihilator* bezüglich eines Operators ist ein Wert, für den gilt, dass bei Anwendung des Operators auf den Annihilator und einen anderen Wert der Annihilator das Ergebnis bildet. Beispielsweise ist 0 der Annihilator bezüglich der Multiplikation, da $0 \times x = x \times 0 = 0$. Es gibt keinen Annihilator für die Addition.

Es gibt drei Gesetze für reguläre Ausdrücke, die diese Konzepte betreffen und nachfolgend aufgeführt sind.

- $\emptyset + L = L + \emptyset = L$. Dieses Gesetz stellt fest, dass $\emptyset$ die Einheit bezüglich der Mengenvereinigung ist. Das heißt, $\emptyset$ vermittelt in Verbindung mit der Vereinigung die Identitätsfunktion, weil L vereinigt mit $\emptyset$ identisch L ist. Dieses Gesetz wird deshalb das Identitätsgesetz der Vereinigung genannt.
- $\varepsilon L = L\varepsilon = L$. Dieses Gesetz stellt fest, dass ε die Einheit bezüglich der Verkettung ist. Das heißt, ε vermittelt in Verbindung mit der Verkettung die Identitätsfunktion, weil L verkettet mit ε identisch L ist. Dieses Gesetz wird deshalb das Identitätsgesetz der Verkettung genannt.
- $\emptyset L = L\emptyset = \emptyset$. Dieses Gesetz stellt fest, dass $\emptyset$ der Annihilator der Verkettung ist.

Diese Gesetze sind mächtige Werkzeuge zur Vereinfachung von Ausdrücken. Wenn beispielsweise die Vereinigung verschiedener Ausdrücke vorliegt, von denen einige gleich $\emptyset$ sind oder zu $\emptyset$ vereinfacht wurden, dann können die betreffenden Ausdrücke aus der Vereinigung eliminiert werden. Analog gilt, wenn eine Verkettung verschiedener Ausdrücke vorliegt, von denen einige gleich ε sind oder zu ε vereinfacht wurden, dann können die betreffenden Ausdrücke aus der Verkettung eliminiert werden. Schließlich gilt, wenn eine beliebige Anzahl von Ausdrücken verkettet wird und nur einer dieser Ausdrücke gleich $\emptyset$ ist, dann kann die gesamte Verkettung durch $\emptyset$ ersetzt werden.

3.4.3 Distributivgesetze

Ein Distributivgesetz betrifft zwei Operatoren und behauptet, dass ein Operator auf jedes Argument des anderen Operators einzeln angewandt werden kann. Das gebräuchlichste Beispiel aus der Arithmetik ist das Distributivgesetz der Multiplikation bezüglich der Addition, das heißt: $x \times (y + z) = x \times y + x \times z$. Da die Multiplikation kommutativ ist, ist es gleichgültig, ob die Multiplikation links oder rechts von der

Summe steht. Es gibt jedoch ein analoges Gesetz für reguläre Ausdrücke, das wir in zwei Formen angeben müssen, da die Verkettung nicht kommutativ ist. Diese Gesetze lauten:

- $L(M + N) = LM + LN$. Dieses Gesetz wird als linkes Distributivgesetz der Verkettung bezüglich der Vereinigung bezeichnet.
- $(M + N)L = ML + NL$. Dieses Gesetz wird als rechtes Distributivgesetz der Verkettung bezüglich der Vereinigung bezeichnet.

Wir wollen nun das linke Distributivgesetz beweisen. Das andere Gesetz wird auf ähnliche Weise bewiesen. Der Beweis bezieht sich ganz allgemein auf Sprachen. Er hängt nicht davon ab, ob die Sprachen durch reguläre Ausdrücke beschrieben werden.

Satz 3.11 Wenn L, M und N Sprachen sind, dann gilt:

$$L(M \cup N) = LM \cup LN$$

Beweis: Der Beweis ähnelt einem anderen Beweis zu einem Distributivgesetz, der in Satz 1.10 vorgestellt wurde. Wir müssen zeigen, dass eine Zeichenreihe w genau dann in $L(M \cup N)$ enthalten ist, wenn sie in $LM \cup LN$ enthalten ist.

(Nur-wenn-Teil) Wenn w in $L(M \cup N)$ enthalten ist, dann gilt $w = xy$, wobei x Element von L und y Element von M oder von N ist. Wenn y in M enthalten ist, dann ist xy in LM enthalten und daher auch in $LM \cup LN$.

Ähnlich gilt: Wenn y in N enthalten ist, dann ist xy in LN enthalten und daher auch in $LM \cup LN$.

(Wenn-Teil) Angenommen, w sei in $LM \cup LN$ enthalten. Dann ist w Element von LM oder von LN. Sei zunächst w in LM. Dann ist $w = xy$, wobei x Element von L und y Element von M ist. Da y Element von M ist, ist y auch in $M \cup N$ enthalten. Folglich ist xy in $L(M \cup N)$ enthalten. Wenn w nicht in LM enthalten ist, dann ist w mit Sicherheit in LN enthalten. Ein ähnliches Argument wie zuvor zeigt, dass w in $L(M \cup N)$ enthalten ist. ■

Beispiel 3.12 Betrachten Sie den regulären Ausdruck $\mathbf{0}+\mathbf{01}^*$. Wir können den Ausdruck $\mathbf{0}$ aus der Vereinigung »herauslösen«, aber zuerst müssen wir erkennen, dass der Ausdruck $\mathbf{0}$ als Verkettung von $\mathbf{0}$ mit ε dargestellt werden kann. Das heißt, wir wenden das Identitätsgesetz der Verkettung an, um $\mathbf{0}$ durch $\mathbf{0}\varepsilon$ zu ersetzen, wodurch wir den Ausdruck $\mathbf{0}\varepsilon + \mathbf{01}^*$ erhalten. Nun können wir das linke Distributivgesetz anwenden, um diesen Ausdruck durch $\mathbf{0}(\varepsilon + \mathbf{1}^*)$ zu ersetzen. Wenn wir weiter erkennen, dass ε in $L(\mathbf{1}^*)$ enthalten ist, dann wissen wir, dass $\varepsilon + \mathbf{1}^* = \mathbf{1}^*$, und können den ursprünglichen Ausdruck vereinfachen zu $\mathbf{01}^*$. ■

3.4.4 Das Idempotenzgesetz

Ein Operator wird als *idempotent* bezeichnet, wenn seine Anwendung auf zwei Operanden mit dem gleichen Wert eben diesen Wert als Ergebnis hat. Die üblichen arithmetischen Operatoren sind nicht idempotent. Im Allgemeinen gilt $x + x \neq x$ und $x \times x \neq x$, obwohl es einige Werte für x gibt, bei denen die Gleichheit gilt (z. B. $0 + 0 = 0$). Vereinigung und Durchschnitt sind jedoch gebräuchliche Beispiele für idempotente Operatoren. Für reguläre Ausdrücke können wir daher folgendes Gesetz formulieren:

- $L + L = L$. Dieses Gesetz, das Idempotenzgesetz für die Vereinigung, besagt, dass die Vereinigung von zwei identischen Ausdrücken eben diesen Ausdruck ergibt, d. h. wir können die Vereinigung der beiden Ausdrücke durch ein Exemplar dieses Ausdrucks ersetzen.

3.4.5 Gesetze bezüglich der Hüllenbildung

Es gibt eine Reihe von Gesetzen, die den Sternoperator und seine Unix-Varianten + und ? betreffen. Wir werden diese Gesetze im Folgenden aufführen und erklären, warum sie gültig sind.

- $(L^*)^* = L^*$. Dieses Gesetz besagt, dass durch die Anwendung des Sternoperators auf einen Ausdruck, der bereits abgeschlossen ist, die Sprache nicht verändert wird. Die Sprache von $(L^*)^*$ besteht aus allen Zeichenreihen, die durch Verkettung der in der Sprache von L^* enthalten Zeichenreihen gebildet werden. Diese Zeichenreihen setzen sich aber wiederum aus Zeichenreihen der Sprache L zusammen. Folglich ist jede in $(L^*)^*$ enthaltene Zeichenreihe auch eine Verkettung von Zeichenreihen aus L und somit in der Sprache von L^* enthalten.
- $\emptyset^* = \varepsilon$. Die Hülle von $\emptyset$ enthält nur die Zeichenreihe ε, wie wir in Beispiel 3.6 erörtert haben.
- $\varepsilon^* = \varepsilon$. Es lässt sich leicht überprüfen, dass die einzige Zeichenreihe, die durch die Verkettung einer beliebigen Anzahl von leeren Zeichenreihen gebildet werden kann, die leere Zeichenreihe selbst ist.
- $L^+ = LL^* = L^*L$. Sie wissen, dass L^+ definiert ist als $L + LL + LLL + \ldots$. Zudem gilt $L^* = \varepsilon + L + LL + LLL + \ldots$. Daraus folgt:

  $$LL^* = L\varepsilon + LL + LLL + LLLL \ldots$$

  Wenn wir uns in Erinnerung rufen, dass $L\varepsilon = L$, dann erkennen wir, dass die unendlichen Ausdrücke für LL^* und für L^+ identisch sind. Der Beweis, dass $L^+ = L^*L$ gilt, ist ähnlich zu führen.[4]
- $L^* = L^+ + \varepsilon$. Dies ist leicht zu beweisen, da L^+ jeden Term aus L^* enthält, mit Ausnahme von ε. Beachten Sie, dass der Zusatz + ε nicht erforderlich ist, wenn die Sprache L die Zeichenreihe ε bereits enthält; für diesen Sonderfall gilt: $L^+ = L^*$.
- $L? = \varepsilon + L$. Diese Regel ist nicht anders als die Definition des ?-Operators.

3.4.6 Gesetze für reguläre Ausdrücke entdecken

Jedes der obigen Gesetze wurde formal oder informell bewiesen. Man könnte nun eine unendliche Vielzahl für reguläre Ausdrücke geltender Gesetze vermuten. Gibt es eine allgemeine Methodologie, die Beweise gültiger Gesetze erleichtert? Es zeigt sich, dass die Gültigkeit eines Gesetzes sich auf die Frage der Gleichheit zweier Sprachen reduzieren lässt. Interessanterweise ist diese Technik eng mit den Operatoren für

4. Beachten Sie, dass also jede Sprache L und ihre Hülle L^* kommutativ sind (bezüglich der Verkettung): $LL^* = L^*L$. Diese Regel widerspricht der Tatsache nicht, dass die Verkettung im Allgemeinen nicht kommutativ ist.

reguläre Ausdrücke verknüpft und kann nicht auf Ausdrücke, die andere Operatoren (wie z. B. den Durchschnitt) enthalten, erweitert werden.

Um dieses Verfahren zu demonstrieren, wollen wir ein vermutetes Gesetz betrachten:

$$(L + M)^* = (L^*M^*)^*$$

Dieses Gesetz besagt Folgendes: Wenn L und M Sprachen sind und der Sternoperator auf deren Vereinigung angewandt wird, dann erhalten wir dieselbe Sprache wie durch die Anwendung des Sternoperators auf die Sprache L^*M^*, das heißt der Anwendung des Sternoperators auf alle Zeichenreihen, die aus null oder mehr Elementen von L gefolgt von null oder mehr Elementen von M bestehen.

Zum Beweis dieses Gesetzes nehmen wir zuerst an, dass die Zeichenreihe w in der Sprache von $(L + M)^*$ enthalten sei.[5] Wir können dann sagen, w = $w_1w_2 \ldots w_k$ für ein k, wobei jede Zeichenreihe w_i in L oder M enthalten ist. Daraus folgt, dass jede Zeichenreihe w_i in der Sprache von L^*M^* enthalten ist. Dies lässt sich wie folgt begründen: Wenn w_i in L enthalten ist, dann wähle die Zeichenreihe w_i aus L; diese Zeichenreihe ist auch in L^* enthalten. Wähle keine Zeichenreihen aus M; das heißt, wähle ε aus M^*. Wenn w_i in M enthalten ist, dann ist die Argumentation ähnlich. Nachdem bewiesen wurde, dass jedes w_i in L^*M^* enthalten ist, folgt daraus, dass w in der Hülle dieser Sprache enthalten ist.

Um den Beweis zu vervollständigen, müssen wir zudem die Umkehrung beweisen, dass Zeichenreihen, die in $(L^*M^*)^*$ enthalten sind, auch in $(L + M)^*$ enthalten sind. Wir übergehen diesen Teil des Beweises, da unser Ziel nicht im Beweis dieses Gesetzes besteht, sondern in der Hervorhebung der folgenden wichtigen Eigenschaft regulärer Ausdrücke:

Jeder reguläre Ausdruck mit Variablen kann als *konkreter* regulärer Ausdruck (ein Ausdruck ohne Variablen) betrachtet werden, wenn man sich jede Variable als ein bestimmtes Symbol vorstellt. Beispielsweise können im Ausdruck $(L + M)^*$ die Variablen L und M durch die Symbole a bzw. b ersetzt werden, womit wir den regulären Ausdruck $(\mathbf{a} + \mathbf{b})^*$ erhalten.

Die Sprache des konkreten Ausdrucks gibt uns Aufschluss über die Form der Zeichenreihen der Sprache, die aus dem ursprünglichen Ausdruck gebildet wird, wenn wir die Variablen durch Sprachen ersetzen. In unserer Analyse von $(L + M)^*$ haben wir so beobachtet, dass jede Zeichenreihe w, die aus einer Folge von Elementen aus L oder M besteht, in der Sprache $(L + M)^*$ enthalten ist. Wir kommen zu diesem Schluss, wenn wir die Sprache des konkreten Ausdrucks $(\mathbf{a} + \mathbf{b})^*$ betrachten, bei der es sich offensichtlich um die Menge aller aus den Zeichen a und b bestehenden Zeichenreihen handelt. Wir könnten jede in L enthaltene Zeichenreihe für ein Vorkommen von a einsetzen, und wir könnten jede in M enthaltene Zeichenreihe für ein Vorkommen von b einsetzen, wobei durchaus verschiedene Zeichenreihen für die verschiedenen Vorkommen von a und b verwendet werden können. Wenn wir diese Substitutionen an allen in $(\mathbf{a} + \mathbf{b})^*$ enthaltenen Zeichenreihen vornehmen, dann erhalten wir alle Zeichenreihen, die gebildet werden, indem Zeichenreihen aus L und/oder M in beliebiger Reihenfolge miteinander verkettet werden.

5. Der Einfachheit halber werden wir die regulären Ausdrücke und ihre Sprachen als identisch behandeln und im Text nicht ausdrücklich durch den Vorsatz »die Sprache von« vor jedem regulären Ausdruck unterscheiden.

Die obige Aussage mag offensichtlich erscheinen, aber wie im Exkurs »Über reguläre Ausdrücke hinausgehende Erweiterungen des Tests können fehlschlagen« hervorgehoben wird, ist sie nicht einmal wahr, wenn andere Operatoren zu den drei Operatoren regulärer Ausdrücke hinzugefügt werden. Wir beweisen im nächsten Satz das allgemeine Prinzip für reguläre Ausdrücke.

Satz 3.13 Sei E ein regulärer Ausdruck mit den Variablen L_1, L_2, ..., L_m. Bilde den konkreten regulären Ausdruck C, indem jedes Vorkommen von L_j durch das Symbol a_j mit $j = 1, 2, \ldots, m$ ersetzt wird. Dann gilt für beliebige Sprachen L_1, L_2, ..., L_m, dass jede Zeichenreihe w aus $L(E)$ dargestellt werden kann als $w = w_1w_2 \ldots w_k$, wobei jede Zeichenreihe w_i in einer der Sprachen L_{j_i} und die Zeichenreihe $a_{j_1}a_{j_2}\ldots a_{j_k}$ in der Sprache $L(C)$ enthalten ist. Weniger formal ausgedrückt: Wir können L(E) konstruieren, indem wir jede in $L(C)$ enthaltene Zeichenreihe, die wir hier $a_{j_1}a_{j_2}\ldots a_{j_k}$ nennen, nehmen und für jede der Symbole a_{j_i} eine Zeichenreihe aus der zugehörigen Sprache L_{j_i} einsetzen.

Beweis: Der Beweis ist eine strukturelle Induktion über den Ausdruck E.

Induktionsbeginn: Die Basisfälle sind gegeben, wenn E *aus* ε, $\emptyset$ oder einer Variablen L besteht. In den ersten beiden Fällen gibt es nichts zu beweisen, da der konkrete Ausdruck C mit E identisch ist. Wenn E eine Variable L ist, dann gilt $L(E) = L$. Der konkrete Ausdruck C ist einfach **a**, wobei a das L entsprechende Symbol ist. Folglich ist $L(C) = \{a\}$. Wenn wir das Symbol a in dieser einen Zeichenreihe durch eine beliebige Zeichenreihe aus L ersetzen, dann erhalten wir die Sprache L, die gleich $L(E)$ ist.

Induktionsschritt: Es sind drei Fälle zu unterscheiden, die vom letzten Operator von E abhängen. Nehmen wir zuerst an, $E = F + G$, d. h. die Vereinigung ist der letzte Operator. Seien C und D die konkreten Ausdrücke, die aus F bzw. G gebildet werden, indem konkrete Symbole für die Sprachvariablen in diese Ausdrücke eingesetzt werden. Beachten Sie, dass sowohl in F als auch in G alle Vorkommen einer bestimmten Variablen durch dasselbe Symbol ersetzt werden müssen. Wir erhalten dann für E den konkreten Ausdruck $C + D$, und $L(C + D) = L(C) + L(D)$.

Angenommen, w ist eine in $L(E)$ enthaltene Zeichenreihe, wenn die Sprachvariablen von E durch bestimmte Sprachen ersetzt werden. Dann ist w Element von $L(F)$ oder von $L(G)$. Nach der Induktionsannahme erhalten wir w, indem wir mit einer konkreten Zeichenreihe aus $L(C)$ oder $L(D)$ beginnen und für die Symbole Zeichenreihen der entsprechenden Sprachen substituieren. Folglich kann die Zeichenreihe w in jedem Fall konstruiert werden, indem wir mit einer konkreten Zeichenreihe aus $L(C + D)$ beginnen und dieselben Ersetzungen von Symbolen durch Zeichenreihen durchführen.

Wir müssen zudem die Fälle betrachten, in denen E gleich FG oder F^* ist. Die Argumentation ähnelt allerdings der oben für die Vereinigung beschriebenen, und wir überlassen die Vervollständigung dieses Beweises dem Leser. ■

3.4.7 Test eines für reguläre Ausdrücke geltenden Gesetzes der Algebra

Wir können nun den Test zur Beantwortung der Frage, ob ein Gesetz für reguläre Ausdrücke gültig ist, formulieren und beweisen. Der Test zur Beantwortung der Frage, ob

$E = F$ wahr ist, wobei E und F zwei reguläre Ausdrücke mit der gleichen Menge von Variablen sind, lautet:

1. Wandle E und F in konkrete reguläre Ausdrücke C und D um, indem jede Variable durch ein konkretes Symbol ersetzt wird.
2. Prüfe, ob $L(C) = L(D)$. Falls dies zutrifft, dann ist $E = F$ wahr und somit ein gültiges Gesetz, andernfalls ist das »Gesetz« falsch. Beachten Sie, dass wir den Test zur Beantwortung der Frage, ob zwei reguläre Ausdrücke dieselbe Sprache definieren, erst in Abschnitt 4.4 behandeln werden. Wir können jedoch Ad-hoc-Methoden einsetzen, um die Gleichheit von Sprachpaaren zu entscheiden, für die wir uns interessieren. Rufen Sie sich dazu auch in Erinnerung: Wenn zwei Sprachen nicht gleich sind, dann genügt zum Beweis die Angabe eines Gegenbeispiels, nämlich einer einzigen Zeichenreihe, die zu einer der beiden Sprachen gehört, aber nicht zur anderen.

Satz 3.14 Der obige Test identifiziert für reguläre Ausdrücke geltende Gesetze korrekt.

Beweis: Wir werden zeigen, dass $L(E) = L(F)$ genau dann für alle Sprachen gilt, die für die Variablen von E und F eingesetzt werden, wenn $L(C) = L(D)$.

(Nur-wenn-Teil) Angenommen, $L(E) = L(F)$ gilt für alle möglichen Sprachen, die für die Variablen eingesetzt werden können. Wähle für jede Variable L das konkrete Symbol a, das L in den Ausdrücken C und D ersetzt. Dann gilt für diesen Fall $L(C) = L(E)$ und $L(D) = L(F)$. Da $L(E) = L(F)$ gegeben ist, folgt $L(C) = L(D)$.

(Wenn-Teil) Angenommen, $L(C) = L(D)$. Nach Satz 3.13 werden $L(E)$ und $L(F)$ jeweils konstruiert, indem die konkreten Symbole der Zeichenreihen in $L(C)$ und $L(D)$ durch Zeichenreihen der Sprache, der diese Symbole zugeordnet sind, ersetzt werden. Wenn die Zeichenreihen von $L(C)$ und $L(D)$ gleich sind, dann sind auch die beiden Sprachen, die auf diese Weise konstruiert werden, gleich, d. h. $L(E) = L(F)$. ■

Beispiel 3.15 Betrachten Sie den Gesetzesvorschlag $(L + M)^* = (L^*M^*)^*$. Wenn wir die Variablen L und M durch die konkreten Symbole a und b ersetzen, dann erhalten wir die regulären Ausdrücke $(\mathbf{a} + \mathbf{b})^* = (\mathbf{a}^*\mathbf{b}^*)^*$. Es lässt sich mühelos nachweisen, dass beide Ausdrücke die Sprache beschreiben, die alle aus den Symbolen a und b bestehenden Zeichenreihen umfasst. Folglich stehen die beiden konkreten Ausdrücke für dieselbe Sprache, und das Gesetz gilt.

Betrachten Sie $L^* = L^*L^*$ nun als weiteres Beispiel für ein Gesetz. Die konkreten Sprachen lauten $\mathbf{a}^*$ bzw. $\mathbf{a}^*\mathbf{a}^*$, und jede dieser Sprachen umfasst die Menge aller aus dem Symbol a bestehenden Zeichenreihen. Auch hier stellen wir fest, dass das gefundene Gesetz gültig ist, d. h. die Verkettung einer abgeschlossenen Sprache mit sich selbst ergibt eben diese Sprache.

Als letztes Beispiel betrachten Sie das angebliche Gesetz $L + ML = (L + M)L$. Wenn wir die Symbole a und b für die Variablen L und M wählen, dann erhalten wir die konkreten Ausdrücke $\mathbf{a} + \mathbf{ba} = (\mathbf{a} + \mathbf{b})\mathbf{a}$. Die Sprachen dieser Ausdrücke sind allerdings nicht gleich. Beispielsweise ist die Zeichenreihe aa in der Sprache des zweiten, nicht aber des ersten enthalten. Folglich ist dieses angebliche Gesetz falsch. ■

Über reguläre Ausdrücke hinausgehende Erweiterungen des Tests können fehlschlagen

Lassen Sie uns eine erweiterte Algebra für reguläre Ausdrücke betrachten, die den Schnittmengenoperator enthält. Wie wir in Satz 4.8 sehen werden, wird interessanterweise die Menge der beschreibbaren Sprachen durch das Hinzufügen des Operators $\cap$ zu den drei Operatoren für reguläre Ausdrücke nicht erweitert. Allerdings wird dadurch der Test für die Gültigkeit algebraischer Gesetze ungültig.

Betrachten Sie das »Gesetz« $L \cap M \cap N = L \cap M$; das heißt, die Schnittmenge von beliebigen drei Sprachen ist identisch mit der Schnittmenge der ersten beiden Sprachen. Dieses Gesetz ist offenkundig falsch. Sei beispielsweise $L = M = \{a\}$ und $N = \emptyset$. Der auf der Konkretisierung von Variablen basierende Test würde den Unterschied hier nicht aufdecken. Das heißt, wenn wir L, M und N durch die Symbole a, b und c ersetzen würden, dann würden wir testen, ob $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Da beide Seiten gleich der leeren Menge sind, wären die Sprachen gleich, und der Test würde implizieren, dass das »Gesetz« gilt.

3.4.8 Übungen zum Abschnitt 3.4

Übung 3.4.1 Prüfen Sie die Gültigkeit folgender Identitäten regulärer Ausdrücke:

* a) $R + S = S + R$

b) $(R + S) + T = R + (S + T)$

c) $(RS)T = R(ST)$

d) $R\,(S + T) = RS + RT$

e) $(R + S)\,T = RT + ST$

* f) $(R^*)^* = R^*$

g) $(\varepsilon + R)^* = R^*$

h) $(R^*S^*)^* = (R + S)^*$

! **Übung 3.4.2** Beweisen Sie die Wahrheit oder Falschheit der folgenden Aussagen über reguläre Ausdrücke:

* a) $(R + S)^* = R^* + S^*$

b) $(RS + R)^*R = R(SR + R)^*$

* c) $(RS + R)^*RS = (RR^*S)^*$

d) $(R + S)^*S = (R^*S)^*$

e) $S(RS + S)^*R = RR^*S(RR^*S)^*$

Übung 3.4.3 In Beispiel 3.6 haben wir den folgenden regulären Ausdruck entwickelt:

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

Entwickeln Sie unter Verwendung der Distributivgesetze zwei verschiedene, einfachere, äquivalente Ausdrücke.

Übung 3.4.4 Am Anfang von Abschnitt 3.4.6 haben wir einen Teil des Beweises, dass $(L^*M^*)^* = (L + M)^*$, dargestellt. Vervollständigen Sie den Beweis, indem Sie zeigen, dass in $(L^*M^*)^*$ enthaltene Zeichenreihen auch in $(L + M)^*$ enthalten sind.

! **Übung 3.4.5** Vervollständigen Sie den Beweis von Satz 3.13, indem Sie die Fälle behandeln, in denen der reguläre Ausdruck E die Form FG oder die Form F^* hat.

3.5 Zusammenfassung von Kapitel 3

- *Reguläre Ausdrücke* : Diese algebraische Notation beschreibt genau dieselben Sprachen wie endliche Automaten: die regulären Sprachen. Als Operatoren für reguläre Ausdrücke sind die Vereinigung, die Verkettung (auch Konkatenation oder »Punktoperator« genannt) und der Sternoperator (oder Kleenesche Hülle) definiert.
- *Reguläre Ausdrücke in der Praxis* : Systeme wie Unix und verschiedene Unix-Befehle verwenden eine erweiterte Notation für reguläre Ausdrücke, die viele Kürzel für gebräuchliche Ausdrücke bietet. Zeichenklassen erlauben es, Symbolmengen einfach darzustellen, während Operatoren, wie »ein oder mehr Vorkommen von« und »höchstens ein Vorkommen von« die regulären Operatoren für reguläre Ausdrücke ergänzen.
- *Äquivalenz von regulären Ausdrücken und endlichen Automaten*: Wir können einen DEA durch eine induktive Konstruktion in einen regulären Ausdruck umwandeln. In einer solchen Konstruktion werden Ausdrücke für Beschriftungen von Pfaden konstruiert, die zunehmend umfangreichere Mengen von Zuständen durchlaufen. Alternativ können wir ein Verfahren zur Eliminierung von Zuständen einsetzen, um den regulären Ausdruck für einen DEA zu bilden. In der umgekehrten Richtung können wir aus einem regulären Ausdruck rekursiv einen ε-NEA konstruieren und diesen ε-NEA dann bei Bedarf in einen DEA umwandeln.
- *Die Algebra regulärer Ausdrücke* : Viele der algebraischen Gesetze der Arithmetik gelten für reguläre Ausdrücke, obwohl es sehr wohl Unterschiede gibt. Vereinigung und Verkettung sind assoziativ, aber nur die Vereinigung ist kommutativ. Die Verkettung ist bezüglich der Vereinigung distributiv. Die Vereinigung ist idempotent.
- *Algebraische Identitäten testen* : Wir können feststellen, ob eine Äquivalenz regulärer Ausdrücke, die Variablen als Argumente enthalten, wahr ist, indem wir die Variablen durch verschiedene Konstanten ersetzen und testen, ob die resultierenden Sprachen gleich sind.

LITERATURANGABEN ZU KAPITEL 3

Das Konzept der regulären Ausdrücke und der Beweis ihrer Äquivalenz mit endlichen Automaten ist das Werk von S. C. Kleene [3]. Die Konstruktion eines ε-NEA aus einem regulären Ausdruck, die hier vorgestellt wurde, wird als »McNaughton-Yamada-Konstruktion« bezeichnet und stammt aus [4]. Der Test zur Überprüfung der Äquivalenz von regulären Ausdrücken, in dem Variablen als Konstanten behandelt werden, wurde von J. Gischer [2] beschrieben. Dieser Bericht zeigte, dass das Hinzufügen verschiedener anderer Operatoren, wie Durchschnitt oder ordnungserhaltende Durchmischung (siehe Übung 7.3.4) den Test fehlschlagen lässt, obwohl diese Operatoren die Klasse der darstellbaren Sprachen nicht erweitern.

Schon vor der Entwicklung von Unix untersuchte K. Thompson die Verwendung regulärer Ausdrücke in Kommandos wie `grep`, und sein Algorithmus zur Verarbeitung solcher Kommandos findet sich in [5]. Die frühe Unix-Entwicklung führte zur Entwicklung verschiedener anderer Kommandos, die von der Notation erweiterter regulärer Ausdrücke stark Gebrauch machen, z. B. das Kommando `lex` von M. Lesk. Eine Beschreibung dieses Kommandos und anderer auf regulären Ausdrücken basierenden Techniken sind in [1] enthalten.

1. A. V. Aho, R. Sethi und J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA.

2. J. L. Gischer [1984]. STAN-CS-TR-84-1033.

3. S. C. Kleene [1956]. »Representation of events in nerve nets and finite automata«, in C. E. Shannon und J. McCarthy, *Automata Studies*, Princeton Univ. Press, S. 3–42.

4. R. McNaughton und H. Yamada [Jan. 1960]. »Regular expressions and state graphs for automata«, *IEEE Trans. Electronic Computers* **9**:1, S. 39–47.

5. K. Thompson [Juni 1968]. »Regular expression search algorithm«, *Comm. ACM* **11**:6, S. 419–422.