

In diesem Kapitel beschreiben wir zuerst die Aufgabe der lexikalischen Analyse. Dann führen wir reguläre Ausdrücke als Hilfsmittel zur formalen Spezifikation dieser Aufgabe ein. Schließlich zeigen wir, wie endliche Automaten für die Realisierung eingesetzt werden können. Wir erläutern, wie man zu einem regulären Ausdruck einen nichtdeterministischen endlichen Automaten erzeugt, der genau die von dem regulären Ausdruck beschriebene Sprache akzeptiert, wie dieser deterministisch gemacht werden kann, und wie die Anzahl der Zustände eines gegebenen deterministischen Automaten gegebenenfalls verringert werden kann. Diese drei Schritte liefern ein Generierungsverfahren für lexikalische Analysatoren (Scanner). Am Ende diskutieren wir einige praktische Erweiterungen und erläutern, wie ein Scanner um einen Sieber erweitert werden kann.

---

## 2.1 Die Aufgabe der lexikalischen Analyse

Nehmen wir an, das Quellprogramm sei in einer Datei abgelegt. Es besteht aus einer Folge von Zeichen. Die lexikalische Analyse liest diese Folge ein und zerlegt sie in eine Folge von lexikalischen Einheiten, die *Symbole* genannt werden. Der Scanner liest die Eingabe von links nach rechts. Bei verschränkter Arbeitsweise von Scanner, Sieber und Parser ruft der Parser die Kombination Scanner-Sieber auf, um das nächste Symbol zu erhalten. Der Scanner beginnt die Analyse mit dem Zeichen, welches auf das Ende des zuletzt gefundenen Symbols folgt, und sucht den längsten Präfix der restlichen Eingabe, der ein Symbol der Sprache ist. Eine Darstellung dieses Symbols reicht er an den Sieber weiter, der feststellt, ob dieses Symbol für den Parser relevant ist oder ignoriert werden soll. Ist es nicht relevant, so stößt der Sieber den Scanner erneut an. Andernfalls gibt er eine eventuell veränderte Darstellung des Symbols an den Parser zurück.

Der Scanner muss i. A. in der Lage sein, unendlich viele oder zumindest sehr viele verschiedene Symbole zu erkennen. Die Menge aller Symbole werden deshalb in

endlich viele Klassen eingeteilt. In eine *Symbolklasse* werden dabei sinnvollerweise Symbole verwandter Struktur bzw. ähnlicher syntaktischer Funktion zusammen gefasst. Wir unterscheiden:

- Das Alphabet ist der Vorrat an Zeichen, die in dem Programmtext vorkommen dürfen. Das aktuelle Alphabet bezeichnen wir mit  $\Sigma$ .
- Ein *Symbol* ist ein Wort über dem Alphabet  $\Sigma$ . Beispiele sind etwa  $xyz12$ ,  $125$ ,  $class$ , „ $abc$ “.
- Eine *Symbolklasse* ist eine Menge von Symbolen. Beispiele sind etwa die Menge der Bezeichner (identifier), die Menge der *int*-Konstanten und die der konstanten Zeichenketten. Diese bezeichnen wir mit den Namen *Id*, *Intconst* bzw. *String*.
- Die *Darstellung eines Symbols* fasst alle vorliegenden Informationen eines gefundenen Symbols zusammen, die für eine nachgeordnete Phase des Übersetzers erforderlich sind. Der Scanner könnte etwa das Wort  $xyz12$  als Paar (*Id*, „ $xyz12$ “), bestehend aus dem Namen der Klasse und dem gefundenen Symbol an den Sieber weitergeben. Der Sieber könnte das Wort „ $xyz12$ “ durch eine Interndarstellung des Bezeichners ersetzen, etwa eine eindeutige Kennzahl, bevor er das Symbol an den Parser weiterreicht.

## 2.2 Reguläre Ausdrücke und endliche Automaten

Zuerst führen wir einige Grundbegriffe ein. Mit  $\Sigma$  bezeichnen wir ein beliebiges *Alphabet*, d. h. eine endliche, nichtleere Menge von Zeichen. Ein *Wort*  $x$  über  $\Sigma$  der Länge  $n$  ist eine Folge von  $n$  Zeichen aus  $\Sigma$ . Das *leere Wort*  $\varepsilon$  ist die leere Folge von Zeichen, d. h. die Folge der Länge 0. Einzelne Zeichen aus  $\Sigma$  fassen wir als Worte der Länge 1 auf. Für  $n \geq 0$  bezeichnet  $\Sigma^n$  die Menge der Worte der Länge  $n$ . Insbesondere ist  $\Sigma^0 = \{\varepsilon\}$  und  $\Sigma^1 = \Sigma$ . Die Menge aller Worte bezeichnen wir mit  $\Sigma^*$ . Entsprechend bezeichnet  $\Sigma^+$  die Menge der *nicht-leeren* Worte, d. h.

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \quad \text{und} \quad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$

Mehrere Worte können zu einem Gesamtwort zusammen gesetzt werden. Die *Konkatenation* der Worte  $x$  und  $y$  hängt die Folge der Zeichen  $y$  hinten an die Folge der Zeichen  $x$  an, d. h.

$$x \cdot y = x_1 \dots x_m y_1 \dots y_n,$$

sofern  $x = x_1 \dots x_m$ ,  $y = y_1 \dots y_n$  für  $x_i, y_j \in \Sigma$ . Haben  $x$  und  $y$  die Längen  $n$  bzw.  $m$ , dann liefert die Konkatenation von  $x$  und  $y$  ein Wort der Länge  $n + m$ . Die Konkatenation ist eine binäre Operation auf der Menge  $\Sigma^*$ . Im Gegensatz etwa zur Addition auf Zahlen ist die Konkatenation von Worten nicht *kommutativ*. Das heißt, dass das Wort  $x \cdot y$  i. a. verschieden von dem Wort  $y \cdot x$  ist. Wie die Addition auf Zahlen ist die Konkatenation von Worten aber *assoziativ*, d. h.

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \quad \text{für alle } x, y, z \in \Sigma^*$$

Das leere Wort  $\varepsilon$  ist das *neutrale* Element bezüglich der Konkatenation von Worten, d. h.

$$x \cdot \varepsilon = \varepsilon \cdot x = x \quad \text{für alle } x \in \Sigma^*.$$

Im Folgenden schreiben wir oft einfach  $xy$  für  $x \cdot y$ .

Für ein Wort  $w = xy$  mit  $x, y \in \Sigma^*$  nennen wir  $x$  ein *Präfix* und  $y$  ein *Suffix* von  $w$ . Präfixe und Suffixe sind spezielle *Teilworte*. Allgemein ist das Wort  $y$  ein Teilwort des Wortes  $w$ , falls  $w = xyz$  für Worte  $x, z \in \Sigma^*$ . Präfixe, Suffixe oder ganz allgemein Teilworte von  $w$  heißen *echt*, falls sie verschieden von  $w$  sind.

Teilmengen von  $\Sigma^*$  werden (formale) *Sprachen* genannt. Auf Sprachen benötigen wir einige Operationen. Nehmen wir an,  $L, L_1, L_2 \subseteq \Sigma^*$  seien Sprachen. Die *Vereinigung*  $L_1 \cup L_2$  besteht aus allen Worten aus  $L_1$  oder  $L_2$ :

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ oder } w \in L_2\}.$$

Die *Konkatenation*  $L_1 \cdot L_2$  (oder kurz  $L_1 L_2$ ) besteht aus allen Worten, die sich durch Konkatenation eines Wortes aus  $L_1$  mit einem Wort aus  $L_2$  ergeben:

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

Das *Komplement*  $\overline{L}$  der Sprache  $L$  besteht aus allen Worten aus  $\Sigma^*$ , die nicht in  $L$  enthalten sind:

$$\overline{L} = \Sigma^* - L.$$

Für  $L \subseteq \Sigma^*$  bezeichnet  $L^n$  die  $n$ -fache Konkatenation von  $L$ ,  $L^*$  die Vereinigung aller Konkatenationen und  $L^+$  die Vereinigung aller nichtleeren Konkatenationen von  $L$ , d. h.

$$\begin{aligned} L^n &= \{w_1 \dots w_n \mid w_1, \dots, w_n \in L\} \\ L^* &= \{w_1 \dots w_n \mid \exists n \geq 0. w_1, \dots, w_n \in L\} = \bigcup_{n \geq 0} L^n \\ L^+ &= \{w_1 \dots w_n \mid \exists n > 0. w_1, \dots, w_n \in L\} = \bigcup_{n \geq 1} L^n \end{aligned}$$

Die Operation  $(\_)^*$  heißt *Kleene-Stern*.

### Reguläre Sprachen und reguläre Ausdrücke

Als Symbolklassen, deren Elemente ein Scanner identifizieren kann, bieten sich nichtleere *reguläre* Sprachen an. Jede reguläre Sprache, die nicht gleich der leeren Menge ist, lässt sich aus einelementigen Sprachen durch die Operationen Vereinigung, Konkatenation und Kleene-Stern konstruieren. Formal wird die Menge aller *regulären Sprachen* über einem Alphabet  $\Sigma$  induktiv definiert durch:

- Die leere Menge  $\emptyset$  und die Menge  $\{\varepsilon\}$ , die nur aus dem leeren Wort besteht, sind regulär.

- Die Mengen  $\{a\}$  für alle  $a \in \Sigma$  sind regulär über  $\Sigma$ .
- Sind  $R_1$  und  $R_2$  reguläre Sprachen über  $\Sigma$ , so auch  $R_1 \cup R_2$  und  $R_1 R_2$ .
- Ist  $R$  regulär über  $\Sigma$ , dann auch  $R^*$ .

Gemäß dieser Definition lässt sich jede reguläre Sprache durch einen regulären Ausdruck spezifizieren. *Reguläre Ausdrücke* über  $\Sigma$  und die regulären Sprachen, die von ihnen beschrieben werden, sind ebenfalls induktiv definiert:

- $\emptyset$  ist ein regulärer Ausdruck über  $\Sigma$ , der die reguläre Sprache  $\emptyset$  beschreibt.  
 $\varepsilon$  ist ein regulärer Ausdruck über  $\Sigma$  und beschreibt die reguläre Sprache  $\{\varepsilon\}$ .
- Für jedes  $a \in \Sigma$  ist  $a$  ein regulärer Ausdruck über  $\Sigma$ , der die reguläre Sprache  $\{a\}$  beschreibt.
- Sind  $r_1$  und  $r_2$  reguläre Ausdrücke über  $\Sigma$ , welche die regulären Sprachen  $R_1$  bzw.  $R_2$  beschreiben, dann sind  $(r_1 \mid r_2)$  und  $(r_1 r_2)$  reguläre Ausdrücke über  $\Sigma$ , die die regulären Sprachen  $R_1 \cup R_2$  bzw.  $R_1 R_2$  beschreiben.
- Ist  $r$  ein regulärer Ausdruck über  $\Sigma$ , der die reguläre Sprache  $R$  beschreibt, dann ist  $r^*$  ein regulärer Ausdruck über  $\Sigma$ , der die reguläre Sprache  $R^*$  beschreibt.

In praktischen Anwendungen wird oft  $r^?$  als Abkürzung für  $(r \mid \varepsilon)$  geschrieben und gegebenenfalls auch  $r^+$  für den Ausdruck  $(r r^*)$ .

Bei der Definition regulärer Ausdrücke haben wir angenommen, dass das Symbol für die leere Menge bzw. das leere Wort nicht in  $\Sigma$  enthalten sind – genauso wenig wie die Klammern  $(, )$  sowie die Operatorsymbole  $\mid$  und  $*$  und gegebenenfalls  $?, +$ . Diese Zeichen gehören zu dem Beschreibungsmechanismus für reguläre Ausdrücke und nicht zu den Sprachen, die durch die regulären Ausdrücke beschrieben werden. Sie werden deshalb auch *Metazeichen* genannt. Der Vorrat an darstellbaren Zeichen ist jedoch beschränkt. Jedes Programmsystem, welches Beschreibungen regulärer Sprachen durch reguläre Ausdrücke akzeptiert, muss deshalb das Problem lösen, dass Metazeichen mit Zeichen aus  $\Sigma$  zusammenfallen können.

Eine Möglichkeit, Metazeichen von Zeichen zu unterscheiden, sind *Escape-Zeichen*. In vielen gängigen Spezifikationssprachen für reguläre Sprachen wird dazu das Zeichen  $\backslash$  verwendet. Soll ein Metazeichen wie der senkrechte Strich  $\mid$  auch im Alphabet vorkommen, wird jedem Vorkommen dieses Zeichens als Alphabetszeichen in dem regulären Ausdruck ein  $\backslash$  vorangestellt. Ein senkrechter Strich des Alphabets wird dann durch  $\backslash \mid$  repräsentiert.

Um Klammern einzusparen, legen wir die folgenden Operator-Präcedenzen fest: Der  $?$ -Operator hat die höchste Präcedenz, gefolgt von dem Kleene-Stern  $(\_)*$ , gegebenenfalls dem Operator  $(\_)^+$ , dann der Konkatenation und schließlich dem *Alternativzeichen*  $\mid$ .

**Beispiel 2.2.1** Die folgende Tabelle listet einige regulären Ausdrücke zusammen mit den Sprachen auf, die von ihnen beschrieben werden, und einigen, manchmal auch allen ihren Elementen:

regulärer Ausdruck	beschriebene Sprache	Elemente der Sprache
$a \mid b$	$\{a, b\}$	$a, b$
$ab^*a$	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbbba, \dots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \dots$
$abba$	$\{abba\}$	$abba \quad \square$

Reguläre Ausdrücke, welche die leere Menge als Symbol enthalten, lassen sich durch wiederholte Ausnutzung der folgenden Gleichheiten vereinfachen:

$$\begin{aligned}
 r \mid \emptyset &= \emptyset \mid r = r \\
 r \emptyset &= \emptyset r = \emptyset \\
 \emptyset^* &= \emptyset? = \epsilon
 \end{aligned}$$

Dabei soll das Gleichheitszeichen zwischen zwei regulären Ausdrücken bedeuten, dass sie die gleiche Sprache bezeichnen. Wir erhalten:

**Lemma 2.2.1** Zu jedem regulären Ausdruck  $r$  über  $\Sigma$  lässt sich ein regulärer Ausdruck  $r'$  konstruieren, der die gleiche Sprache bezeichnet wie  $r$  und die folgenden Eigenschaften besitzt:

1. Bezeichnet  $r$  die leere Sprache, dann ist  $r'$  der reguläre Ausdruck  $\emptyset$ .
2. Bezeichnet  $r$  eine nichtleere Sprache, dann kommt in  $r'$  das Symbol  $\emptyset$  nicht mehr vor.  $\square$

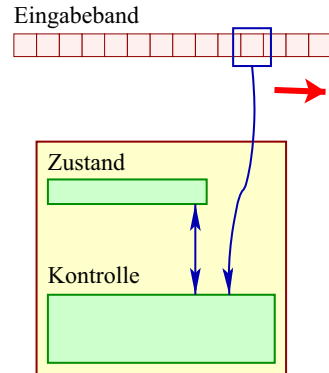
In unseren Anwendungen kommen nur reguläre Ausdrücke vor, die nichtleere Sprachen beschreiben. Deshalb wird kein Symbol zur Darstellung der leeren Sprache benötigt. Auf das leere Wort kann dagegen nicht so leicht verzichtet werden. Zum Beispiel möchte man spezifizieren, dass ein Vorzeichen *optional* ist, also vorhanden ist oder fehlen kann. In den Spezifikationssprachen, die in Scannern zum Einsatz kommen, wird jedoch auch für das leere Wort oft kein eigenes Zeichen bereit gestellt: in allen praktischen Fällen reicht hier die Verwendung des  $?$ -Operators aus. Um die Vorkommen des Symbols  $\varepsilon$  aus einem regulären Ausdruck zu beseitigen, können die folgenden Gleichheiten eingesetzt werden:

$$\begin{aligned}
 r \mid \varepsilon &= \varepsilon \mid r = r? \\
 r \varepsilon &= \varepsilon r = r \\
 \varepsilon^* &= \varepsilon? = \varepsilon
 \end{aligned}$$

Wir erhalten:

**Lemma 2.2.2** Zu jedem regulären Ausdruck  $r$  über  $\Sigma$  kann ein regulärer Ausdruck  $r'$  (möglicherweise mit Vorkommen von  $?$ ) konstruiert werden, der die gleiche Sprache beschreibt, aber zusätzlich die folgenden Eigenschaften hat:

**Abb. 2.1** Schematische Darstellung eines endlichen Automaten



1. Beschreibt  $r$  die Menge  $\{\varepsilon\}$ , dann ist  $r'$  gleich  $\varepsilon$ .
2. Beschreibt  $r$  eine Menge, die verschieden von  $\{\varepsilon\}$  ist, dann enthält  $r'$  kein Vorkommen des Symbols für  $\varepsilon$ .  $\square$

### Endliche Automaten

Während reguläre Ausdrücke zur Spezifikation von Symbolklassen eingesetzt werden, basiert die Implementierung von Scannern auf endlichen Automaten. *Endliche Automaten* sind Akzeptoren für reguläre Sprachen. Sie verwalten eine Zustandsvariable, die nur endlich viele verschiedene Werte, die *Zustände* des Automaten, annehmen kann. Wie die Abb. 2.1 zeigt, verfügt ein endlicher Automat weiterhin konzeptuell über einen Lesekopf, mit dem er das Eingabeband von links nach rechts überstreichen kann. Das dynamische Verhalten des Automaten wird durch eine *Übergangsrelation*  $\Delta$  beschrieben.

Formal repräsentieren wir einen *nichtdeterministischen endlichen Automaten* (mit  $\varepsilon$ -Übergängen) (NEA) als ein Tupel  $M = (Q, \Sigma, \Delta, q_0, F)$ , wobei

- $Q$  eine endliche Menge von *Zuständen* ist,
- $\Sigma$  ein endliches Alphabet, das *Eingabealphabet*, ist,
- $q_0 \in Q$  der *Anfangszustand* ist,
- $F \subseteq Q$  die Menge der *Endzustände* ist, und
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$  die *Übergangsrelation* ist.

Ein Übergang  $(p, x, q) \in \Delta$  gibt an, dass  $M$  aus seinem aktuellen Zustand  $p$  in den Zustand  $q$  wechseln kann. Ist  $x \in \Sigma$ , muss  $x$  das nächste Zeichen in der Eingabe sein und nach dem Lesen von  $x$  der Eingabekopf um ein Zeichen weiterbewegt werden. Ist  $x = \varepsilon$ , wird bei dem Übergang kein Zeichen der Eingabe gelesen: der Eingabekopf bleibt in der alten Position. Einen solchen Übergang nennt man einen  $\varepsilon$ -Übergang.

Besonders wichtig sind endliche Automaten ohne  $\varepsilon$ -Übergänge, die außerdem in jedem Zustand und für jedes Zeichen genau eine Übergangsmöglichkeit besitzen. Ein solcher Automat heißt *deterministischer endlicher Automat* (DEA). Bei einem DEA ist die Übergangsrelation  $\Delta$  eine *Funktion*  $\Delta : Q \times \Sigma \rightarrow Q$ .

Wir erläutern die Arbeitsweise eines DEA im Vergleich zu einem als Scanner eingesetzten endlichen Automaten. Die Verhaltensweise des Scanners setzen wir dabei in Kästen ab. Ein deterministischer endlicher Automat soll Eingabeworte daraufhin prüfen, ob sie in einer gegebenen Sprache sind oder nicht. Er akzeptiert ein Wort, wenn er nach Lesen des ganzen Wortes in einem Endzustand angekommen ist.

Ein als Scanner eingesetzter deterministischer endlicher Automat zerlegt ein Eingabewort dagegen in eine Folge von Teilworten, die *Symbole* der gegebenen Sprache sind. Jedes Symbol bringt ihn von seinem Anfangszustand in einen Endzustand.

Der deterministische endliche Automat wird in seinem Anfangszustand gestartet. Sein Lesekopf steht dabei am Anfang des Eingabebandes.

Bei Einsatz eines deterministischen endlichen Automaten als Scanner steht er auf dem ersten noch nicht konsumierten Zeichen.

Dann macht er eine Folge von Schritten. Abhängig von dem aktuellen Zustand und dem nächsten Eingabezeichen ändert der DEA in jedem Schritt seinen Zustand und setzt seinen Lesekopf auf das jeweils nächste Zeichen. Der Automat akzeptiert das Eingabewort, wenn die Eingabe erschöpft ist und der aktuelle Zustand ein Endzustand ist.

Der Scanner führt ganz analog eine Folge von Schritten aus. Er meldet das Vorkommen eines Symbols oder einen Fehler. Hat er aus dem aktuellen Zustand keinen Übergang mehr in Richtung auf einen Endzustand, kehrt er zu dem letzten Zeichen der Eingabe zurück, nach dessen Lesen er in einem Endzustand für eine Symbolklasse war. Diese Klasse, zusammen mit dem Präfix der Eingabe bis zu dieser Stelle liefert er als Beschreibung des Symbols zurück. Dann startet der Scanner neu. Wurde für das aktuell gesuchte Symbol dagegen kein Endzustand durchlaufen, liegt ein Fehler vor.

Unser Ziel ist, aus der Spezifikation einer regulären Sprache eine Implementierung der Sprache abzuleiten, d. h. wir wollen zu einem regulären Ausdruck  $r$  einen deterministischen endlichen Automaten konstruieren, der die von  $r$  beschriebene Sprache akzeptiert. In einem ersten Zwischenschritt wird zu  $r$  ein NEA konstruiert, der die von  $r$  beschriebene Sprache akzeptiert.

**Tab. 2.1** Die Übergangsrelation eines endlichen Automaten zum Erkennen von vorzeichenlosen *int*- und *float*-Konstanten. Die erste Spalte repräsentiert die identischen Spalten für die Ziffern  $i = 0, \dots, 9$ , die fünfte diejenige für  $+$  und  $-$

$T_M$	$i$	.	$E$	$+, -$	$\varepsilon$
0	$\{1, 2\}$	$\{3\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{4\}$
2	$\{2\}$	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{4\}$	$\emptyset$	$\{5\}$	$\emptyset$	$\{7\}$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\{6\}$
6	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

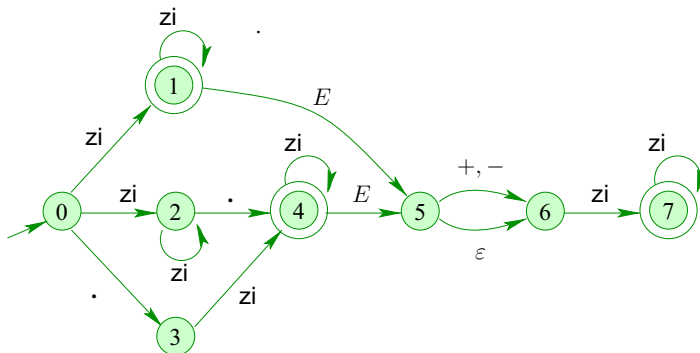
Ein endlicher Automat  $M = (Q, \Sigma, \Delta, q_0, F)$  startet in seinem Anfangszustand  $q_0$  und führt dann für ein gegebenes Eingabewort nichtdeterministisch eine *Berechnung*, d. h. eine Folge von *Schritten* durch. Führt eine Berechnung in einen Endzustand, wird das Eingabewort *akzeptiert*. Das zukünftige Verhalten des endlichen Automaten wird alleine durch seinen Zustand  $q \in Q$  und die restliche Eingabe  $w \in \Sigma^*$  bestimmt. Das Paar  $(q, w)$  bildet die aktuelle *Konfiguration* des Automaten. Ein Paar  $(q_0, w)$  ist eine *Anfangskonfiguration*, während Paare  $(q, \varepsilon)$  mit  $q \in F$  *Endkonfigurationen* sind.

Die *Schritt*-Relation  $\vdash_M$  ist eine binäre Relation zwischen Konfigurationen. Für  $q, p \in Q$ ,  $a \in \Sigma \cup \{\varepsilon\}$  und  $w \in \Sigma^*$  gilt  $(q, aw) \vdash_M (p, w)$  genau dann, wenn  $(q, a, p) \in \Delta$  und  $a \in \Sigma \cup \{\varepsilon\}$  sind.  $\vdash_M^*$  bezeichnet die reflexive, transitive Hülle der Relation  $\vdash_M$ . Die von dem endlichen Automaten  $M$  *akzeptierte Sprache* ist dann definiert als

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \varepsilon) \text{ mit } q_f \in F\}.$$

**Beispiel 2.2.2** In Tab. 2.1 ist die Übergangsrelation eines endlichen Automaten  $M$  in Form einer zweidimensionalen Matrix  $T_M$  dargestellt. Die Zustände des Automaten sind bezeichnet mit den Zahlen  $0, \dots, 7$ . Das Alphabet ist die Menge  $\{0, \dots, 9, ., E, +, -\}$ . Jede Zeile der Tabelle beschreibt die Übergänge für einen der Zustände des Automaten, die Spalten entsprechen den Elementen aus  $\Sigma \cup \{\varepsilon\}$ . Der Eintrag  $T_M[q, x]$  enthält die Menge der Zustände  $p$  mit  $(q, x, p) \in \Delta$ . Der Zustand 0 ist der Anfangszustand, während  $\{1, 4, 7\}$  die Menge der Endzustände ist. Der Automat erkennt vorzeichenlose *int*- und *float*-Konstanten. Mit *int*-Konstanten kann der akzeptierende Zustand 1 erreicht werden, während mit *float*-Konstanten die akzeptierenden Zuständen 4 oder 6 erreicht werden können.  $\square$





**Abb. 2.2** Übergangsdiagramm zu dem endlichen Automaten aus Beispiel 2.2.2. Das Zeichen  $zi$  steht für die Menge  $\{0, 1, \dots, 9\}$ . Eine mit  $zi$  markierte Kante ersetzt mit  $0, 1, \dots, 9$  markierte Kanten mit gleichem Eingangs- und Ausgangsknoten

Jeder endliche Automat  $M$  lässt sich graphisch durch ein (endliches) *Übergangsdiagramm* darstellen. Ein Übergangsdiagramm ist ein endlicher, gerichteter, kantenmarkierter Graph. Die Menge der Knoten dieses Graphen ist gegeben durch die Menge der Zustände des Automaten  $M$ , während die Menge der Kanten durch die Menge der Übergänge von  $M$  gegeben ist: ein Übergang  $(p, x, q)$  entspricht dann einer Kante von  $p$  nach  $q$ , die mit  $x$  beschriftet ist. In dem Übergangsdiagramm werden der *Startknoten* (dargestellt durch einen eingehenden Pfeil) und die *Endknoten* (graphisch doppelt umrandet dargestellt) gesondert markiert. Für ein Wort  $w \in \Sigma^*$  ist ein  $w$ -Weg in diesem Graphen ein Weg von einem Knoten  $q$  zu einem Knoten  $p$ , so dass  $w$  die Konkatenation der Kantenmarkierungen ist. Die von  $M$  akzeptierte Sprache besteht damit genau aus allen Worten  $w \in \Sigma^*$ , für die es einen  $w$ -Weg in dem Zustandsdiagramm von  $q_0$  zu einem Knoten  $q \in F$  gibt.

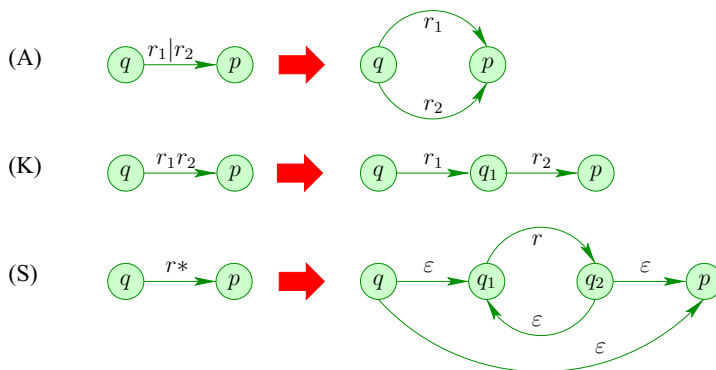
**Beispiel 2.2.3** Das Übergangsdiagramm zu dem endlichen Automaten von Beispiel 2.2.2 zeigt Abb. 2.2.  $\square$

### Akzeptoren

Der nächste Satz garantiert, dass zu jedem regulären Ausdruck ein nichtdeterministischer Automat konstruiert werden kann.

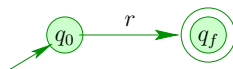
**Satz 2.2.1** Zu jedem regulären Ausdruck  $r$  über einem Alphabet  $\Sigma$  gibt es einen nichtdeterministischen endlichen Automaten  $M_r$  mit Eingabealphabet  $\Sigma$ , so dass  $L(M_r)$  die von  $r$  beschriebene reguläre Sprache ist.

Im Folgenden geben wir ein Verfahren an, das zu einem regulären Ausdruck  $r$  über dem Alphabet  $\Sigma$  das Übergangsdiagramm eines nichtdeterministischen endlichen Automaten konstruiert. Konzeptuell startet die Konstruktion mit einer Kante von



**Abb. 2.3** Die Regeln zur Konstruktion eines endlichen Automaten für einen regulären Ausdruck

einem Anfangszustand und zu einem Endzustand, die mit  $r$  markiert ist:



Dann wird  $r$  gemäß seiner syntaktischen Struktur zerlegt. Dazu dienen die Regeln aus Abb. 2.3. Sie werden solange angewendet, bis alle Kanten mit  $\emptyset$ ,  $\epsilon$  oder Zeichen aus  $\Sigma$  markiert sind. Dann werden die Kanten, die mit  $\emptyset$  beschriftet sind, entfernt.

Die Anwendung einer Regel ersetzt eine Kante, auf deren Beschriftung das Muster der linken Seite passt, durch eine entsprechende Kopie des Teilgraphen der rechten Seite. Für jeden Operator ist genau eine Regel zuständig. Die Anwendung einer Regel entfernt eine Kante mit einem regulären Ausdruck  $r$  und fügt neue Kanten ein, die mit den Argumentausdrücken des äußersten Konstruktors in  $r$  beschriftet sind. Im Falle der Regel für den Kleene-Stern werden zusätzlich neue  $\epsilon$ -Kanten eingefügt. Wenn wir als Zustände des endlichen Automaten natürliche Zahlen wählen, lässt sich diese Konstruktion durch das folgende Programmstück implementieren:

```

trans  $\leftarrow \emptyset$ ;
count  $\leftarrow 1$ ;
generate(0, r, 1);
return (count, trans);

```

In der Menge *trans* werden global die Übergänge des erzeugten Automaten gesammelt, während der globale Zähler *count* die größte natürliche Zahl vermerkt, die als Zustand verwendet wurde. Ein Aufruf der Prozedur **generate** für  $(p, r', q)$  fügt die Menge der Übergänge eines endlichen Automaten für den regulären Ausdruck  $r'$  mit Startzustand  $p$  und Endzustand  $q$  in die Menge *trans* ein, wobei neue Zustände jeweils durch Inkrementierung des Zählers *count* gewonnen werden. Diese Prozedur ist rekursiv über der Struktur des regulären Ausdrucks  $r'$  definiert:

```

void generate(int  $p$ , Exp  $r'$ , int  $q$ ) {
    switch ( $r'$ ) {
        case ( $r_1 \mid r_2$ ) : generate( $p, r_1, q$ );
                           generate( $p, r_2, q$ ); return;
        case ( $r_1.r_2$ ) :   int  $q_1 \leftarrow ++count$ ;
                           generate( $p, r_1, q_1$ );
                           generate( $q_1, r_2, q$ ); return;
        case  $r_1^*$  :       int  $q_1 \leftarrow ++count$ ;
                           int  $q_2 \leftarrow ++count$ ;
                            $trans \leftarrow trans \cup \{(p, \varepsilon, q_1), (q_2, \varepsilon, q), (q_2, \varepsilon, q_1)\}$ 
                           generate( $q_1, r_1, q_2$ ); return;
        case  $\emptyset$  :      return;
        case  $x$  :           $trans \leftarrow trans \cup \{(p, x, q)\}$ ; return;
    }
}

```

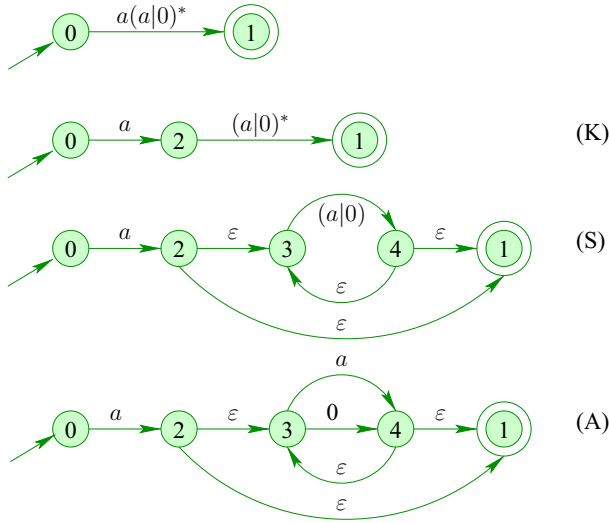
Hier soll **Exp** den Typ regulärer Ausdrücke über dem Alphabet  $\Sigma$  bezeichnen. Als Implementierungssprache dient eine JAVA-artige Programmiersprache. Um elegant mit strukturierten Daten wie regulären Ausdrücken umgehen zu können, wurde die *switch*-Anweisung um die Möglichkeit des *Pattern-Matching* erweitert. Das bedeutet, dass Muster nicht nur zur Unterscheidung verschiedener Alternativen eingesetzt werden, sondern auch zur Identifizierung von Teilstrukturen.

Der Prozeduraufruf  $\text{generate}(0, r, 1)$  terminiert nach  $n$  Regelanwendungen, wenn  $n$  die Anzahl der Operator- und Symbolvorkommen in dem regulären Ausdruck  $r$  ist. Ist  $l$  der Zählerstand nach dem Aufruf, dann benötigt der generierte Automat  $\{0, \dots, l\}$  als Menge von Zuständen, wobei 0 der Startzustand und 1 der einzige Endzustand ist. Die Menge seiner Übergänge sind in der Menge *trans* gesammelt. Der Automat  $M_r$  kann damit in linearer Zeit berechnet werden.

**Beispiel 2.2.4** Der reguläre Ausdruck  $a(a \mid 0)^*$  über dem Alphabet  $\{a, 0\}$  beschreibt die Menge der Worte aus  $\{a, 0\}^*$ , die mit einem  $a$  beginnen. Die Konstruktion des Übergangsdiagramms eines NEA, der diese Sprache akzeptiert, zeigt Abb. 2.4.  $\square$

## Die Teilmengenkonstruktion

Für die praktische Implementierung ziehen wir *deterministische* endliche Automaten nichtdeterministischen Automaten vor. Weil ein deterministischer endlicher Automat  $M$  keine Übergänge unter  $\varepsilon$  kennt und für jedes Paar  $(q, a)$  mit  $q \in Q$  und  $a \in \Sigma$  genau einen Nachfolgezustand besitzt, gibt es für jeden Zustand  $q$  von  $M$  und jedes Wort  $w \in \Sigma^*$  genau einen  $w$ -Weg im Übergangsdiagramm von  $M$ , der in  $q$  startet. Wählen wir  $q$  als den Anfangszustand von  $M$ , dann ist  $w$  im Sprachschatz



**Abb. 2.4** Konstruktion eines Übergangsdiagramms für den regulären Ausdruck  $a(a|0)^*$

von  $M$  genau dann enthalten, wenn dieser Weg in einen Endzustand von  $M$  führt. Glücklicherweise gilt Satz 2.2.2.

**Satz 2.2.2** Zu jedem nichtdeterministischen endlichen Automaten kann ein deterministischer endlicher Automat konstruiert werden, der die gleiche Sprache akzeptiert.  $\square$

**Beweis.** Der Beweis ist konstruktiv und liefert uns den zweiten Schritt des Generierungsverfahrens für Scanner. Er benutzt die *Teilmengenkonstruktion*. Sei  $M = (Q, \Sigma, \Delta, q_0, F)$  ein NEA. Ziel der Teilmengenkonstruktion ist die Konstruktion eines DEA  $\mathcal{P}(M) = (\mathcal{P}(Q), \Sigma, \mathcal{P}(\Delta), \mathcal{P}(q_0)\mathcal{P}(F))$ , der die gleiche Sprache akzeptiert wie  $M$ . Für ein Wort  $w \in \Sigma^*$  sei  $\text{states}(w) \subseteq Q$  die Menge aller Zustände  $q \in Q$ , für die es einen  $w$ -Weg vom Anfangszustand  $q_0$  nach  $q$  gibt. Der DEA  $\mathcal{P}(M)$  ist gegeben durch:

$$\begin{aligned}
 \mathcal{P}(Q) &= \{\text{states}(w) \mid w \in \Sigma^*\} \\
 \mathcal{P}(q_0) &= \text{states}(\varepsilon) \\
 \mathcal{P}(F) &= \{\text{states}(w) \mid w \in L(M)\} \\
 \mathcal{P}(\Delta)(S, a) &= \text{states}(wa) \\
 &\quad \text{für } S \in \mathcal{P}(Q) \text{ und } a \in \Sigma, \text{ sofern } S = \text{states}(w)
 \end{aligned}$$

Wir überzeugen uns davon, dass unsere Definition der Übergangsfunktion  $\mathcal{P}(\Delta)$  vernünftig ist. Dazu vergewissern wir uns, dass für Worte  $w, w' \in \Sigma^*$  mit  $\text{states}(w) = \text{states}(w')$  auch  $\text{states}(wa) = \text{states}(w'a)$  gilt für alle  $a \in \Sigma$ . Daraus folgt insbesondere, dass  $M$  und  $\mathcal{P}(M)$  die gleichen Sprachen akzeptieren.

Wir benötigen eine systematische Weise, um die Zustände und Übergänge von  $\mathcal{P}(M)$  zu konstruieren. Wenn wir die Menge der Zustände von  $\mathcal{P}(M)$  kennen, können wir die Menge der Endzustände von  $\mathcal{P}(M)$  ermitteln. Es gilt nämlich:

$$\mathcal{P}(F) = \{A \in \mathcal{P}(M) \mid A \cap F \neq \emptyset\}$$

Für eine Menge  $A \subseteq Q$  definieren wir die Menge der  $\varepsilon$ -Folgezustände von  $A$  als

$$\text{FZ}_\varepsilon(S) = \{p \in Q \mid \exists q \in S. (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$$

Diese Menge besteht aus allen Zuständen, die von Zuständen aus  $S$  im Übergangsdiagramm von  $M$  durch  $\varepsilon$ -Wege erreichbar sind. Dieser Abschluss kann durch die folgende Funktion berechnet werden:

```

set  $\langle \text{state} \rangle$  closure(set  $\langle \text{state} \rangle$   $S$ ) {
  set  $\langle \text{state} \rangle$   $result \leftarrow \emptyset$ ;
  list  $\langle \text{state} \rangle$   $W \leftarrow \text{list\_of}(S)$ ;
   $state$   $q, q'$ ;
  while ( $W \neq []$ ) {
     $q \leftarrow \text{hd}(W)$ ;  $W \leftarrow \text{tl}(W)$ ;
    if ( $q \notin result$ ) {
       $result \leftarrow result \cup \{q\}$ ;
      forall ( $q' : (q, \varepsilon, q') \in \Delta$ )
         $W \leftarrow q' :: W$ ;
    }
  }
}
return  $result$ ;

```

In der Menge  $result$  werden die von  $A$  aus erreichbaren Zustände des nichtdeterministischen Automaten gesammelt. Die Liste  $W$  enthält alle diejenigen Elemente aus  $result$ , deren  $\varepsilon$ -Übergänge noch nicht betrachtet wurden. Solange  $W$  nicht leer ist, wird der erste Zustand  $q$  aus  $W$  extrahiert. Dazu werden die Hilfsfunktionen  $\text{hd}$  und  $\text{tl}$  verwendet, die das erste Element bzw. den Rest einer Liste liefern. Ist  $q$  bereits in  $result$  enthalten, muss nichts getan werden. Andernfalls wird  $q$  in die Menge  $result$  eingefügt. Dann werden alle Übergänge  $(q, \varepsilon, q')$  für  $q$  in  $\Delta$  betrachtet und die entsprechenden Nachfolgezustände  $q'$  zu  $W$  hinzugefügt.

Mit dem Abschlussoperator  $\text{FZ}_\varepsilon(\_)$  lässt sich der Anfangszustand  $\mathcal{P}(q_0)$  des Teilmengenautomaten berechnen:

$$\mathcal{P}(q_0) = S_\varepsilon = \text{FZ}_\varepsilon(\{q_0\})$$

Um die Menge aller Zustände  $\mathcal{P}(M)$  zusammen mit der Übergangsfunktion  $\mathcal{P}(\Delta)$  von  $\mathcal{P}(M)$  zu konstruieren, werden die Menge  $Q' \subseteq \mathcal{P}(M)$  der bereits gefundenen Zustände und die Menge  $\Delta' \subseteq \mathcal{P}(\Delta)$  der bereits gefundenen Übergänge verwaltet. Am Anfang ist  $Q' = \{\mathcal{P}(q_0)\}$  und  $\Delta' = \emptyset$ .

Für einen Zustand  $S \in Q'$  und jedes  $a \in \Sigma$  werden sein *Nachfolgezustand*  $S'$  unter  $a$  und  $Q'$  und der Übergang  $(S, a, S')$  zu  $\Delta$  hinzu gefügt. Den Nachfolgezustand  $S'$  zu  $S$  unter einem Zeichen  $a \in \Sigma$  erhält man, indem man die Nachfolgezustände aller Zustände  $q \in S$  unter  $a$  zusammenfasst und alle  $\varepsilon$ -Folgezustände hinzufügt:

$$S' = \text{FZ}_\varepsilon(\{p \in Q \mid \exists q \in S : (q, a, p) \in \Delta\})$$

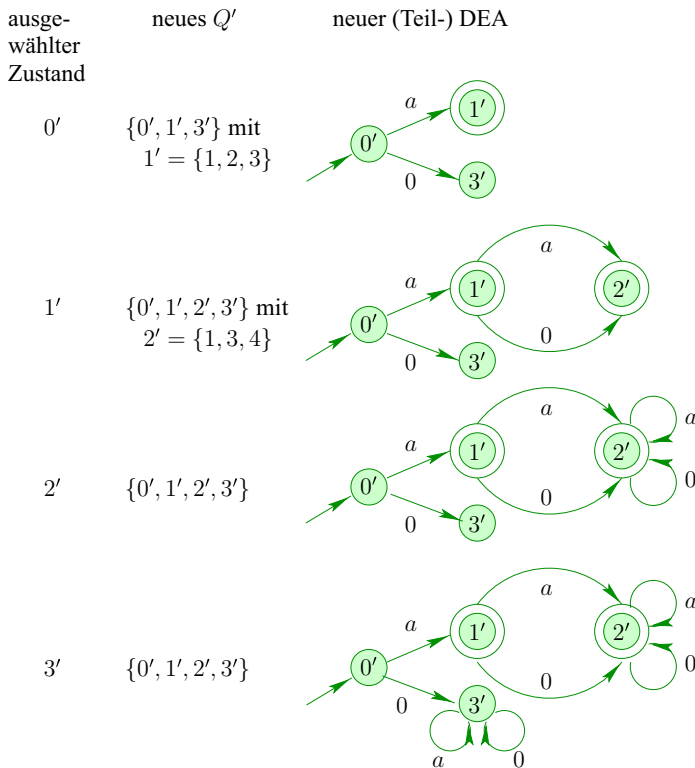
Zur Berechnung dieser Menge dient die Funktion `nextState()`:

```
set  $\langle \text{state} \rangle$  nextState(set  $\langle \text{state} \rangle$   $S$ , symbol  $x$ ) {
  set  $\langle \text{state} \rangle$   $S' \leftarrow \emptyset$ ;
  state  $q, q'$ ;
  forall ( $q' : q \in S, (q, x, q') \in \Delta$ )  $S' \leftarrow S' \cup \{q'\}$ ;
  return closure( $S'$ );
}
```

Die Erweiterungen von  $Q'$  und  $\Delta'$  werden so lange ausgeführt, bis alle Nachfolgezustände der Zustände in  $Q'$  unter Zeichen aus  $\Sigma$  bereits in der Menge  $Q'$  enthalten sind. Technisch heißt das, dass die Menge aller Zustände *states* und die Menge aller Übergänge *trans* des Teilmengenumautomaten iterativ durch die folgende Schleife berechnet werden können:

```
list  $\langle \text{set } \langle \text{state} \rangle \rangle$   $W$ ;
set  $\langle \text{state} \rangle$   $S_0 \leftarrow \text{closure}(\{q_0\})$ ;
states  $\leftarrow \{S_0\}$ ;  $W \leftarrow [S_0]$ ;
trans  $\leftarrow \emptyset$ ;
set  $\langle \text{state} \rangle$   $S, S'$ ;
while ( $W \neq []$ ) {
   $S \leftarrow \text{hd}(W)$ ;  $W \leftarrow \text{tl}(W)$ ;
  forall ( $x \in \Sigma$ ) {
     $S' \leftarrow \text{nextState}(S, x)$ ;
    trans  $\leftarrow \text{trans} \cup \{(S, x, S')\}$ ;
    if ( $S' \notin \text{states}$ ) {
      states  $\leftarrow \text{states} \cup \{S'\}$ ;
       $W \leftarrow W \cup \{S'\}$ ;
    }
  }
}
```

□



**Abb. 2.5** Die Teilmengenkonstruktion für den NEA aus Beispiel 2.2.4

**Beispiel 2.2.5** Die Teilmengenkonstruktion, angewendet auf den endlichen Automaten aus Beispiel 2.2.4 könnte in den in Abb. 2.5 beschriebenen Schritten ablaufen. Die Zustände des zu konstruierenden DEA wurden mit den gestrichenen natürlichen Zahlen  $0', 1', \dots$  bezeichnet. Der Anfangszustand  $0'$  bezeichnet die Menge  $0' = \{0\}$ . Die Zustände in  $Q'$ , deren Nachfolger bereits ermittelt wurden, sind unterstrichen. Der Zustand  $3'$  repräsentiert die leere Menge von Zuständen, d. h. den *Fehlerzustand*. Er kann nicht mehr verlassen werden. Er ist der Nachfolgezustand eines Zustandes  $q$  unter  $a$ , wenn es keinen Übergang unter  $a$  aus  $q$  heraus gibt.  $\square$

### Minimierung

Die in den beiden Schritten aus regulären Ausdrücken erzeugten deterministischen endlichen Automaten sind i. A. nicht die *kleinstmöglichen*, welche die Ausgangssprache akzeptieren. Möglicherweise gibt es mehrere Zustände, die das gleiche *Akzeptanzverhalten* haben. Zustände  $p$  und  $q$  haben das gleiche Akzeptanzverhalten, wenn der Automat für jedes Eingabewort entweder aus  $p$  und  $q$  in einen Endzustand geht oder aus  $p$  und  $q$  in einen Nichtendzustand geht.

Sei  $M = (Q, \Sigma, \Delta, q_0, F)$  ein deterministischer endlicher Automat, bei dem sämtliche Zustände erreichbar sind. Um den Begriff gleichen Akzeptanzverhaltens zu formalisieren, erweitern wir die Übergangsfunktion  $\Delta : Q \times \Sigma \rightarrow Q$  des DEA  $M$  zu einer Übergangsfunktion  $\Delta^* : Q \times \Sigma^* \rightarrow Q$ , die jedem Paar  $(q, w) \in Q \times \Sigma^*$  den eindeutigen Zustand zuordnet, in dem der  $w$ -Weg aus  $q$  im Übergangsdiagramm von  $M$  endet. Die Funktion  $\Delta^*$  ist induktiv über die Länge von Worten definiert durch:

$$\Delta^*(q, \varepsilon) = q \quad \text{und} \quad \Delta^*(q, aw) = \Delta^*(\Delta(q, a), w)$$

für alle  $q \in Q$ ,  $w \in \Sigma^*$  und  $a \in \Sigma$ . Dann haben die Zustände  $p, q \in Q$  das gleiche Akzeptanzverhalten, wenn

$$\Delta^*(p, w) \in F \quad \text{genau dann, wenn} \quad \Delta^*(q, w) \in F$$

In diesem Fall schreiben wir  $p \sim_M q$ . Die Relation  $\sim_M$  ist eine Äquivalenzrelation auf  $Q$ . Den DEA  $M$  nennen wir *minimal*, falls es keinen DEA mit weniger Zuständen gibt, der die gleiche Sprache akzeptiert wie  $M$ . Es gilt:

**Satz 2.2.3** Zu jedem deterministischen endlichen Automaten  $M$  kann ein minimaler deterministischer endlicher Automaten  $M'$  konstruiert werden, der die gleiche Sprache akzeptiert wie  $M$ . Dieser minimale deterministische Automat ist (bis auf Umbenennung der Zustände) eindeutig.

**Beweis.** Für einen deterministischen endlichen Automaten  $M = (Q, \Sigma, \Delta, q_0, F)$  wollen wir einen deterministischen endlichen Automaten  $M' = (Q', \Sigma, \Delta', q'_0, F')$  definieren, der minimal ist.

Ohne Beschränkung der Allgemeinheit können wir annehmen, dass sämtliche Zustände vom Startzustand aus erreichbar sind. Als Menge der Zustände des deterministischen endlichen Automaten  $M'$  wählen wir die Menge der Äquivalenzklassen von Zuständen des Automaten  $M$  unter  $\sim_M$ . Für einen Zustand  $q \in Q$  sei  $[q]_M$  die Äquivalenzklasse des Zustands  $q$  bzgl. der Relation  $\sim_M$ , d. h.

$$[q]_M = \{p \in Q \mid q \sim_M p\}$$

Dann ist die Menge der Zustände von  $M'$  gegeben durch:

$$Q' = \{[q]_M \mid q \in Q\}$$

Entsprechend sind der Anfangszustand und die Menge der Endzustände von  $M'$  definiert durch:

$$q'_0 = [q_0]_M \quad F' = \{[q]_M \mid q \in F\}$$



und die Übergangsfunktion von  $M$  für  $q' \in Q'$  und  $a \in \Sigma$  liefert:

$$\Delta'(q', a) = [\Delta(q, a)]_M \quad \text{für ein } q \in Q \text{ mit } q' = [q]_M.$$

Man überzeugt sich, dass die neue Übergangsfunktion  $\Delta'$  wohldefiniert ist, d. h. dass für  $[q_1]_M = [q_2]_M$  auch  $[\Delta(q_1, a)]_M = [\Delta(q_2, a)]_M$  gilt für alle  $a \in \Sigma$ . Weiterhin zeigt man, dass

$$\Delta^*(q, w) \in F \quad \text{genau dann, wenn} \quad (\Delta')^*([q]_M, a) \in F'$$

gilt für alle  $q \in Q$  und  $w \in \Sigma^*$ . Daraus folgt, dass  $L(M) = L(M')$  gilt. Wir behaupten, dass der DEA  $M'$  minimal ist. Um dies zu zeigen, betrachten wir einen weiteren DEA  $M'' = (Q'', \Sigma, \Delta'', q_0'', F'')$  mit  $L(M'') = L(M')$ , dessen Zustände sämtlich erreichbar sein sollen. Nehmen wir für einen Widerspruch an, es gebe einen Zustand  $q \in Q''$  und Wörter  $u_1, u_2 \in \Sigma^*$  geben so dass  $(\Delta'')^*(q_0'', u_1) = (\Delta'')^*(q_0'', u_2) = q$ , aber  $(\Delta')^*([q_0]_M, u_1) \neq (\Delta')^*([q_0]_M, u_2)$  gilt. Für  $i = 1, 2$ , sei  $p_i \in Q$  ein Zustand mit  $(\Delta')^*([q_0]_M, u_i) = [p_i]_M$ . Da  $[p_1]_M \neq [p_2]_M$  gilt, können insbesondere  $p_1$  und  $p_2$  nicht äquivalent sein. Andererseits gilt jedoch für alle Wörter  $w \in \Sigma^*$ , dass

$$\begin{aligned} \Delta^*(p_1, w) \in F & \text{ gdw. } (\Delta')^*([p_1]_M, w) \in F' \\ & \text{ gdw. } (\Delta'')^*(q, w) \in F'' \\ & \text{ gdw. } (\Delta')^*([p_2]_M, w) \in F' \\ & \text{ gdw. } \Delta^*(p_2, w) \in F \end{aligned}$$

Folglich müssten die Zustände  $p_1, p_2$  – entgegen unserer Annahme – äquivalent sein. Weil es zu jedem Zustand  $[p]_M$  des DEA  $M'$  ein Wort  $u$  gibt mit  $(\Delta')^*([q_0]_M, u) = [p]_M$ , folgern wir, dass es eine surjektive Abbildung der Zustände von  $M''$  auf die Zustände von  $M'$  geben muss. Dann besitzt  $M''$  allerdings mindestens genauso viele Zustände wie  $M'$ . Der DEA  $M'$  ist deshalb der gewünschte minimale deterministische endliche Automat.  $\square$

Die praktische Konstruktion von  $M'$  erfordert, dass die Äquivalenzklassen  $[q]_M$  der Relation  $\sim_M$  berechnet werden. Ist *jeder* Zustand ein Endzustand oder *kein* Zustand ein Endzustand, dann sind alle Zustände äquivalent, d. h.  $Q = [q_0]_M$  ist der einzige Zustand von  $M'$ .

Nehmen wir im Folgenden an, dass diese beiden Fälle nicht vorliegen, d. h.  $Q \neq F \neq \emptyset$ . Dann verwaltet das Verfahren eine *Partition*  $\Pi$  auf der Menge  $Q$  der Zustände des DEA  $M$ . Eine Partition auf der Grundmenge  $Q$  ist eine Menge von nicht-leeren Teilmengen von  $Q$ , deren Vereinigung  $Q$  ist.

Eine Partition  $\Pi$  nennen wir *stabil* unter der Übergangsrelation  $\Delta$ , falls es für alle  $q' \in \Pi$  und alle  $a \in \Sigma$  ein  $p' \in \Pi$  gibt mit:

$$\{\Delta(q, a) \mid q \in q'\} \subseteq p'$$

In einer stabilen Partition führen alle Übergänge aus einer Menge der Partition in genau eine Menge der Partition.

In der Partition  $\Pi$  werden die Mengen von Zuständen verwaltet, von denen wir annehmen, dass sie gleiches Akzeptanzverhalten haben. Stellt sich heraus, dass eine Menge  $q' \in \Pi$  Zustände mit unterschiedlichem Akzeptanzverhalten enthält, wird die Menge  $q'$  aufgeteilt. Unterschiedliches Akzeptanzverhalten zweier Zustände  $q_1$  und  $q_2$  wird erkannt, wenn für ein  $a \in \Sigma$  die Nachfolgezustände  $\Delta(q_1, a)$  und  $\Delta(q_2, a)$  in unterschiedlichen Mengen aus  $\Pi$  liegen. Die Partition ist also nicht stabil. Einen solchen Aufteilungsschritt nennen wir eine *Verfeinerung* von  $\Pi$ . Die sukzessive Verfeinerung der Partition  $\Pi$  endet, wenn keine weitere Aufteilung notwendig ist, d. h.  $\Pi$  unter der Übergangsrelation  $\Delta$  stabil ist.

Eine Konstruktion des minimalen deterministischen endlichen Automaten geht deshalb so vor. Am Anfang wird die Partition  $\Pi$  mit  $\Pi = \{F, Q \setminus F\}$  initialisiert. Nehmen wir an, die gegenwärtige Partition  $\Pi$  der Menge  $Q$  der Zustände von  $M'$  ist noch nicht stabil unter  $\Delta$ . Dann gibt es eine Menge  $q' \in \Pi$  und ein  $a \in \Sigma$  so, dass die Menge  $\{\Delta(q, a) \mid q \in q'\}$  in keiner der Mengen  $p' \in \Pi$  ganz enthalten ist. Eine solche Menge  $q'$  wird dann in die Partition  $\Pi'$  aufgeteilt, die aus allen nicht-leeren Elementen der Menge

$$\{\{q \in q' \mid \Delta(q, a) \in p'\} \mid p' \in \Pi\}$$

besteht. Die Partition  $\Pi'$  von  $q'$  besteht also aus allen nichtleeren Teilmengen von Zuständen aus  $q'$ , die unter  $a$  in dieselben Mengen  $p' \in \Pi$  führen. Dann wird in  $\Pi$  die Menge  $q'$  durch die Partition  $\Pi'$  von  $q'$  ersetzt, d. h. die Partition  $\Pi$  wird zu der Partition  $(\Pi \setminus \{q'\}) \cup \Pi'$  verfeinert.

Ist nach einer Folge solcher Verfeinerungsschritte die Partition  $\Pi$  stabil, ist die Menge der Zustände von  $M'$  berechnet. Dann gilt:

$$\Pi = \{[q]_M \mid q \in Q\}$$

In jedem Verfeinerungsschritt erhöht sich die Anzahl der Mengen in der Partition  $\Pi$ . Da eine Partition der Menge  $Q$  höchstens so viele Mengen enthalten kann wie  $Q$  Elemente besitzt, terminiert der Algorithmus nach endlich vielen Verfeinerungsschritten.  $\square$

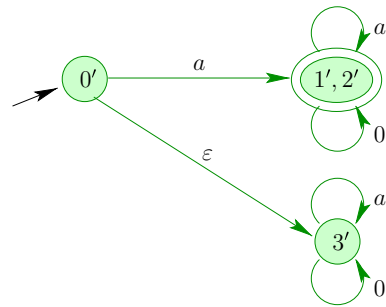
**Beispiel 2.2.6** Wir illustrieren unser Verfahren durch Minimierung des deterministischen endlichen Automaten aus Beispiel 2.2.5. Am Anfang ist die Partition  $\Pi$  gegeben durch:

$$\{\{0', 3'\}, \{1', 2'\}\}$$

Diese Partition ist nicht stabil. Vielmehr muss die erste Menge  $\{0', 3'\}$  zerlegt werden in die Partition  $\Pi' = \{\{0'\}, \{3'\}\}$ . Die entsprechende Verfeinerung der Partition  $\Pi$  liefert die Partition:

$$\{\{0'\}, \{3'\}, \{1', 2'\}\}$$

**Abb. 2.6** Der minimale deterministische endliche Automat aus Beispiel 2.2.6



Diese Partition ist stabil unter  $\Delta$ . Sie liefert deshalb die Zustände des minimalen deterministischen endlichen Automaten. Das Übergangsdiagramm des so konstruierten deterministischen endlichen Automaten zeigt Abb. 2.6.  $\square$

## 2.3 Eine Sprache zur Spezifikation der lexikalischen Analyse

Mit regulären Ausdrücken steht ein Beschreibungsformalismus zur Spezifikation einzelner Symbolklassen für die lexikalische Analyse zur Verfügung. Allein ist er allerdings für viele praktische Zwecke zu unhandlich.

**Beispiel 2.3.1** Der folgende reguläre Ausdruck beschreibt die in den Beispielen 2.2.2 und 2.2.3 durch endliche Automaten akzeptierte Sprache der vorzeichenlosen *int*-Konstanten.

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

Eine entsprechende Beschreibung der *float*-Konstanten würde sich bereits über drei Zeilen erstrecken.  $\square$

In den folgenden Abschnitten werden einige Erweiterungen des Beschreibungsformalismus erläutert, die den Komfort erhöhen, aber die Mächtigkeit, d. h. die beschreibbare Sprachklasse nicht erweitern.

### 2.3.1 Zeichenklassen

Eine Spezifikation der lexikalischen Analyse sollte es erlauben, Mengen von Zeichen zu *Klassen* zusammenzufassen, wenn diese in Symbolen ausgetauscht werden können, ohne dass dadurch die entstehenden Symbole in verschiedene Symbolklassen eingeordnet würden. Dies ist besonders dann hilfreich, wenn das verwendete

Alphabet sehr groß ist, z. B. beliebige *Unicode*-Zeichen enthält. Beispiele für häufig vorkommende Zeichenklassen sind:

$$\begin{aligned} \text{bu} &= a - z A - Z \\ \text{zi} &= 0 - 9 \end{aligned}$$

Die ersten beiden Zeichenklassendefinitionen definieren Mengen von Zeichen durch Angabe von *Intervallen* im zugrundeliegenden Zeichencode, z. B. ASCII. Beachten Sie, dass hier zur Spezifikation von Intervallen als weiteres Metazeichen – benötigt wird. Jetzt lässt sich elegant z. B. eine Definition der Symbolklasse der Bezeichner angeben:

$$\text{Id} = \text{bu}(\text{bu} \mid \text{zi})^*$$

In unseren Zeichenklassendefinitionen kommen wir mit drei Metazeichen aus, nämlich ‘=’, ‘–’ und dem Leerzeichen. Bei der *Verwendung* der Bezeichner für Zeichenklassen muss der Beschreibungsformalismus sicher stellen, dass die neu eingeführten Bezeichner ebenfalls als Metazeichen erkannt werden! In unserem Beispiel verwenden wir dazu einen besonderen Font. In der Praxis stehen die definierten Bezeichner in besonderen Klammern, z. B.  $\{\dots\}$ .

**Beispiel 2.3.2** Der reguläre Ausdruck für vorzeichenlose *int*- und *float*-Konstanten vereinfacht sich durch die Zeichenklassendefinition  $\text{zi} = 0 - 9$  zu:

$$\begin{aligned} &\text{zi zi}^* \\ &\text{zi zi}^* E(+ \mid -)? \text{zi zi}^* \mid \text{zi}^* (. \text{zi} \mid \text{zi}.) \text{zi}^* (E(+ \mid -)? \text{zi zi}^*)? \end{aligned}$$

□

### 2.3.2 Nichtrekursive Klammerung

Programmiersprachen enthalten lexikalische Einheiten, welche durch die sie begrenzenden Klammern charakterisiert sind, z. B. Zeichenketten (strings) und Kommentare. Im Falle der Kommentare können die Klammern durchaus aus mehreren Zeichen zusammengesetzt sein: (\* und \*) bzw. /\* und \*/ oder // und \n (Zeilenwechsel). Zwischen den öffnenden und schließenden Klammern können nahezu beliebige Worte stehen. Dies ist nicht sehr einfach zu beschreiben. Eine abkürzende Schreibweise dafür ist:

$$r_1 \text{ until } r_2$$

Seien  $L_1, L_2$  die durch  $r_1$  bzw.  $r_2$  beschriebenen Sprachen, wobei  $L_2$  das leere Wort nicht enthält. Dann ist die durch den *until*-Ausdruck beschriebene Sprache gegeben durch

$$L_1 \overline{\Sigma^* L_2 \Sigma^*} L_2$$

Ein Kommentar, der mit `//` beginnt und bis zum Zeilenende geht, kann dann beschrieben werden durch:

`// until \n`

---

## 2.4 Die Generierung eines Scanners

Abschnitt 2.2 stellte Verfahren vor, um zu einem regulären Ausdruck einen deterministischen endlichen Automaten bzw. einen minimalen deterministischen endlichen Automaten zu konstruieren. Im Folgenden erläutern wir die notwendigen Erweiterungen dieser Verfahren, die zur Generierung von Scannern oder Siebern erforderlich sind.

### 2.4.1 Zeichenklassen

Zeichenklassen wurden eingeführt, um die regulären Ausdrücke zu vereinfachen. Sie erlauben es ebenfalls, die entstehenden Automaten zu verkleinern. Mit Hilfe der Klassendefinitionen

$$\begin{aligned} \text{bu} &= a - z \\ \text{zi} &= 0 - 9 \end{aligned}$$

lassen sich z. B. die 26 Übergänge zwischen zwei Zuständen eines Automaten unter Buchstaben durch einen Übergang unter `bu` zu ersetzen. Dies vereinfacht den Automaten für den regulären Ausdruck

$$\text{Id} = \text{bu}(\text{bu} \mid \text{zi})^*$$

beträchtlich. Die Implementierung verwaltet dann eine Abbildung  $\chi$ , die jedem Zeichen  $a$  seine zugehörige Klasse zuordnet. Damit die Funktion  $\chi$  konstruiert werden kann, muss jedes Zeichen in genau einer Klasse auftreten. Für Zeichen, die nicht explizit in einer Zeichenklasse vorkommen und für solche, die in einer Symboldefinition explizit auftreten, wird deshalb implizit eine eigene Klasse definiert. Ein Problem tritt auf, wenn Zeichenklassen spezifiziert werden, die nicht disjunkt sind. In diesem Fall wird der Generator implizit die Liste der spezifizierten Zeichenklassen durch eine disjunkte Verfeinerung der Klassen in der Liste ersetzen. Nehmen wir an, es wurden die Klassen  $z_1, \dots, z_k$  spezifiziert. Dann wird für jeden Durchschnitt  $\tilde{z}_1 \cap \dots \cap \tilde{z}_k$ , der nicht leer ist, eine eigene Zeichenklasse eingeführt. Dabei bezeichnet  $\tilde{z}_i$  entweder  $z_i$  oder das Komplement von  $z_i$ . Sei  $D$  die Menge dieser neuen Zeichenklassen. Jede Zeichenklasse  $z_i$  entspricht dann einer geeigneten Alternative  $d_i = (d_{i1} \mid \dots \mid d_{ir_i})$  von Zeichenklassen aus  $D$ . In dem regulären Ausdruck wird dann jedes Vorkommen der Zeichenklasse  $z_i$  durch  $d_i$  ersetzt.

**Beispiel 2.4.1** Nehmen wir an, wir hätten die beiden Klassen

$$\begin{aligned} \text{bu} &= a - z \\ \text{buzi} &= a - z0 - 9 \end{aligned}$$

eingeführt, um damit die Symbolklasse  $\text{Id} = \text{bu buzi}^*$  zu definieren. Dann spaltet der Generator eine dieser Zeichenklassen auf in:

$$\begin{aligned} \text{zi}' &= \text{buzi} \setminus \text{bu} \\ \text{bu}' &= \text{bu} \cap \text{buzi} = \text{bu} \end{aligned}$$

Das Vorkommen von buzi in dem regulären Ausdruck wird deshalb durch  $(\text{bu}' \mid \text{zi}')$  ersetzt.  $\square$

## 2.4.2 Eine Implementierung des *until*-Konstrukts

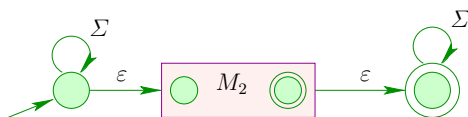
Nehmen wir an, der Scanner soll Symbole erkennen, deren Symbolklasse durch den Ausdruck  $r = r_1 \text{ until } r_2$  beschrieben wird. Nachdem er ein Wort der Sprache für  $r_1$  erkannt hat, muss der Scanner ein Wort der Sprache für  $r_2$  finden und dann anhalten. Diese letzte Aufgabe ist eine Verallgemeinerung des Problems der Mustererkennung auf Zeichenketten (string pattern matching). Es gibt dazu Algorithmen, die für reguläre Muster die Mustererkennung in linearer Zeit in der Größe der zu durchmusternden Eingabe vornehmen. Diese werden z.B. in dem UNIX-Programm EGREP verwendet. Sie konstruieren einen endlichen Automaten für diese Aufgabe. Entsprechend geben wir eine Konstruktion an, die einen DEA für  $r$  konstruiert.

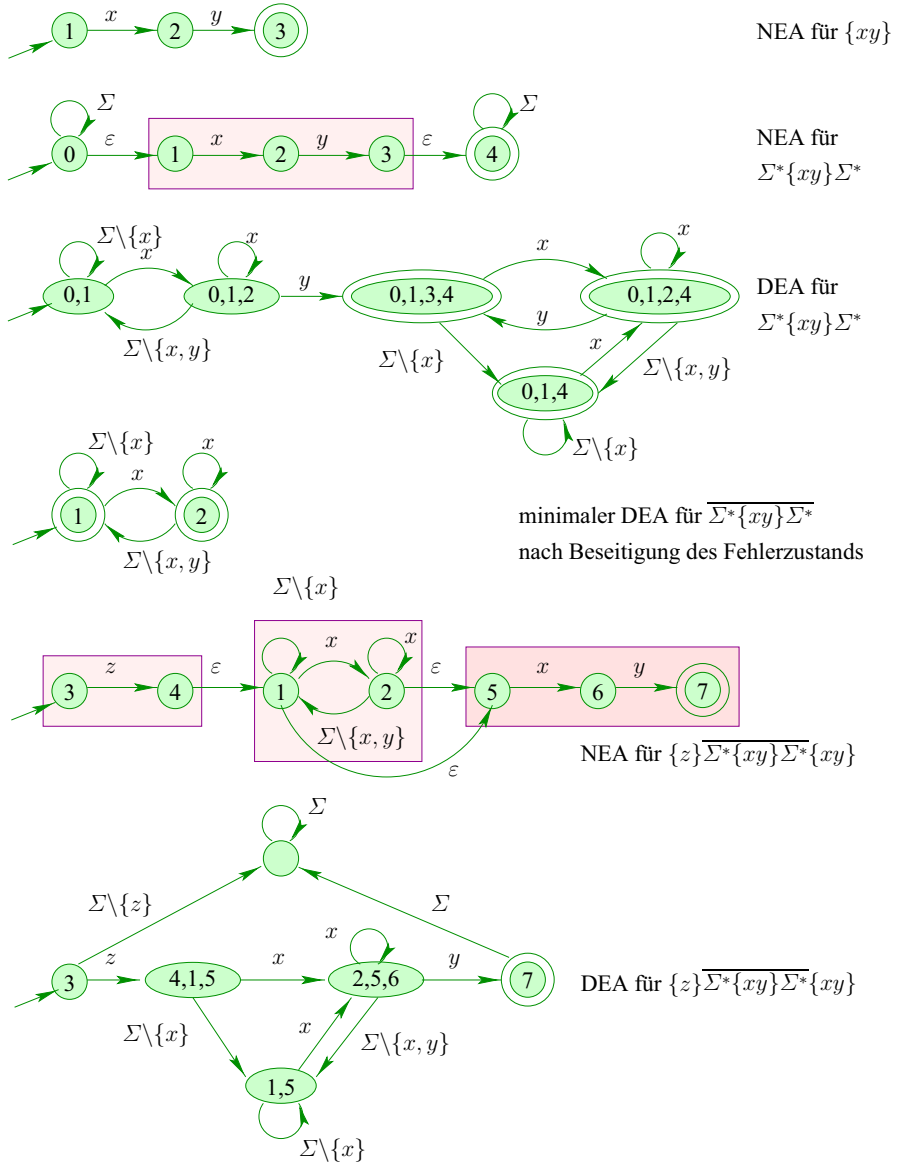
Seien  $L_1, L_2$  die Sprachen, die durch die Ausdrücke  $r_1$  und  $r_2$  beschrieben werden. Die Sprache  $L$ , die durch den Ausdruck  $r_1 \text{ until } r_2$  beschrieben wird, ist:

$$L = L_1 \overline{\Sigma^* L_2 \Sigma^*} L_2$$

Wir gehen von den Automaten für die Sprachen  $L_1$  und  $L_2$  aus und wenden die Standardkonstruktionen für die benötigten Operationen auf den Sprachen an. Die Konstruktion besteht aus den folgenden sieben Schritten. In Abb. 2.7 können diese Schritte an einem einfachen Beispiel nachvollzogen werden.

1. Der erste Schritt konstruiert zu den regulären Ausdrücken  $r_1, r_2$  endliche Automaten  $M_1$  bzw.  $M_2$  mit  $L(M_1) = L_1$  und  $L(M_2) = L_2$ . Von dem endlichen Automaten  $M_2$  wird eine Kopie in Schritt 2 und eine weitere in Schritt 6 benötigt.

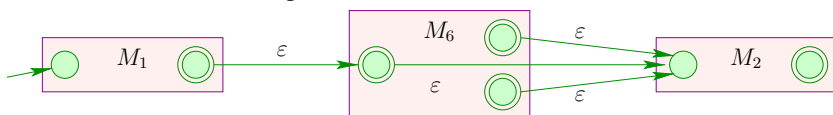




**Abb. 2.7** Die Entwicklung des deterministischen endlichen Automaten für  $z$  until  $xy$  mit  $x, y, z \in \Sigma$

2. Dann wird ein endlicher Automat  $M_3$  für  $\Sigma^*L_2\Sigma^*$  konstruiert, wobei die erste Kopie von  $M_2$  benutzt wird. Der endliche Automat  $M_3$  akzeptiert (nichtdeterministisch) alle Worte über  $\Sigma$ , die ein Teilwort aus  $L_2$  enthalten.

3. Der endliche Automat  $M_3$  wird mit Hilfe der Teilmengenkonstruktion in einen deterministischen endlichen Automaten  $M_4$  umgewandelt.
4. Dann wird ein deterministischer endlicher Automat  $M_5$  konstruiert, der die Sprache zu  $\overline{\Sigma^* L_2 \Sigma^*}$  akzeptiert. Dazu werden in  $M_4$  die Mengen der Endzustände und der Nichtendzustände vertauscht. Jeder Zustand, der vorher Endzustand war, ist es jetzt nicht mehr, jeder Zustand von  $M_4$ , der vorher kein Endzustand war, ist es jetzt geworden. Insbesondere akzeptiert  $M_5$  das leere Wort, da gemäß unserer Annahme  $\varepsilon \notin L_2$ . Deshalb ist der Anfangszustand von  $M_5$  auch ein Endzustand.
5. Der deterministische endliche Automat  $M_5$  wird in einen minimalen deterministischen endlichen Automaten  $M_6$  umgewandelt. Da aus keinem Endzustand von  $M_4$  ein Endzustand von  $M_5$  erreicht werden kann, sind alle Endzustände von  $M_4$  äquivalent und tot. Dieser Fehlerzustand wird ebenfalls entfernt.
6. Aus den endlichen Automaten  $M_1, M_2$  für  $L_1$  bzw.  $L_2$  und  $M_6$  wird ein endlicher Automat  $M_7$  für die Sprache  $L_1 \overline{\Sigma^* L_2 \Sigma^*} L_2$  konstruiert.



Von jedem Endzustand von  $M_6$ , also auch vom Anfangszustand von  $M_6$ , geht ein  $\varepsilon$ -Übergang zum Anfangszustand von  $M_2$ . Von dort führen Wege unter allen Worten  $w \in L_2$  in den Endzustand von  $M_2$ , welcher der einzige Endzustand von  $M_7$  ist.

7. Der endliche Automat  $M_7$  wird in einen deterministischen endlichen Automaten  $M_8$  umgewandelt, der gegebenenfalls noch minimiert wird.

### 2.4.3 Folgen regulärer Ausdrücke

Gegeben sei eine Folge

$$r_0, \dots, r_{n-1}$$

regulärer Ausdrücke für die Symbolklassen, die der Scanner erkennen soll. Ein Scanner zu dieser Folge kann durch die folgenden Schritte generiert werden.

1. In einem ersten Schritt werden endliche Automaten  $M_i = (Q_i, \Sigma, \Delta_i, q_{0,i}, F_i)$  für die regulären Ausdrücke  $r_i$  erzeugt, wobei die  $Q_i$  paarweise disjunkt seien.
2. Die endlichen Automaten  $M_i$  werden zu einem endlichen Automaten  $M = (\Sigma, Q, \Delta, q_0, F)$  zusammengefügt, indem man einen neuen Anfangszustand  $q_0$  hinzufügt zusammen mit  $\varepsilon$ -Übergängen zu den Anfangszuständen  $q_{0,i}$  der Automaten  $M_i$ . Der endliche Automat  $M$  ist deshalb gegeben durch:

$$Q = \{q_0\} \cup Q_0 \cup \dots \cup Q_{n-1} \quad \text{für ein} \quad q_0 \notin Q_0 \cup \dots \cup Q_{n-1}$$

$$F = F_0 \cup \dots \cup F_{n-1}$$

$$\Delta = \{(q_0, \varepsilon, q_{0,i}) \mid 0 \leq i \leq n-1\} \cup \Delta_0 \cup \dots \cup \Delta_{n-1}.$$



Der endliche Automat  $M$  für die Sequenz akzeptiert deshalb die *Vereinigung* der Sprachen, die von den endlichen Automaten  $M_i$  akzeptiert werden. Je nachdem, welcher Endzustand erreicht wird, lässt sich zusätzlich ablesen, zu welcher der spezifizierten Symbolklassen die gelesene Eingabe gehört.

3. Auf den endlichen Automaten  $M$  wird die Teilmengenkonstruktion angewendet. Das Ergebnis ist der deterministische endliche Automat  $\mathcal{P}(M)$ . Ein Wort  $w$  wird der  $i$ -ten Symbolklasse zugeordnet, wenn es zu der Sprache von  $r_i$ , aber zu keiner der Sprachen der regulären Ausdrücke  $r_j$ ,  $j < i$ , gehört. Ausdrücke mit kleinerem Index werden damit gegenüber Ausdrücken mit größerem Index bevorzugt.

Zu welcher Symbolklasse das Wort  $w$  gehört, kann mit Hilfe des Teilmengenautomaten  $\mathcal{P}(M)$  berechnet werden. Das Wort  $w$  gehört genau dann zu der  $i$ -ten Symbolklasse, wenn es in einen Zustand  $q' \subseteq Q$  des Teilmengenautomaten  $\mathcal{P}(M)$  führt mit

$$q' \cap F_i \neq \emptyset \quad \text{und} \quad q' \cap F_j = \emptyset \quad \text{für alle } j < i.$$

Die Menge aller dieser Zustände  $q'$  nennen wir  $F'_i$ .

4. Eventuell wird der deterministische endliche Automat  $\mathcal{P}(M)$  anschließend *minimiert*. Bei der Minimierung muss jedoch darauf geachtet werden, dass Zustände aus  $F'_i$  und aus  $F'_j$  für  $i \neq j$  niemals identifiziert werden. Entsprechend wird der Minimierungsalgorithmus mit der Partition:

$$\Pi = \left\{ F'_0, F'_1, \dots, F'_{n-1}, \mathcal{P}(Q) \setminus \bigcup_{i=0}^{n-1} F'_i \right\}$$

gestartet.

**Beispiel 2.4.2** Seien die Einzelzeichenklassen

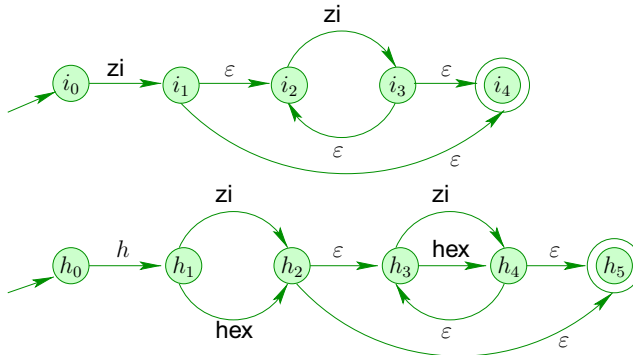
$$\begin{aligned} \text{zi} &= 0 - 9 \\ \text{hex} &= A - F \end{aligned}$$

gegeben. Die Folge regulärer Definitionen

$$\begin{aligned} &\text{zi zi}^* \\ &h(\text{zi} \mid \text{hex})(\text{zi} \mid \text{hex})^* \end{aligned}$$

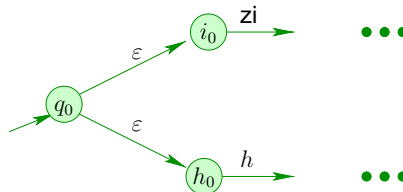
für die Symbolklassen `Intconst` und `Hexconst` wird in den folgenden Schritten bearbeitet:

- Für diese regulären Ausdrücke werden endliche Automaten erzeugt:

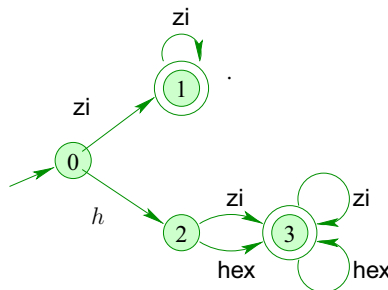


Der Endzustand  $i_4$  steht für Symbole der Klasse `Intconst`, während der Endzustand  $h_5$  Symbole der Klasse `Hexconst` bezeichnet.

- Die beiden endlichen Automaten werden mithilfe eines neuen Anfangszustandes  $q_0$  zusammengefügt:



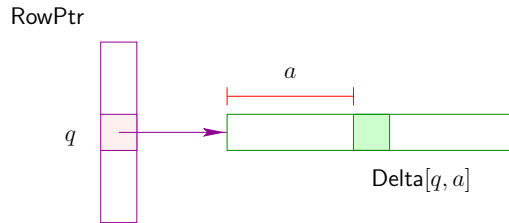
- Dieser endliche Automat wird dann deterministisch gemacht:



Zusätzlich wird ein Zustand 4 benötigt, der Fehlerzustand, welcher der leeren Menge von ursprünglichen Zuständen entspricht. Der Übersichtlichkeit halber haben wir diesen Zustand und alle Übergänge in diesen Zustand in dem Übergangsdiagramm weggelassen.

- Die Minimierung im letzten Schritt ändert den deterministischen endlichen Automaten nicht.

Nach der Konstruktion des deterministischen endlichen Automaten enthält der neue Endzustand 1 den alten Endzustand  $i_4$  und signalisiert deshalb Symbole der Symbolklasse `Intconst`. Der Endzustand 3 enthält  $h_5$  und signalisiert deshalb die Symbolklasse `Hexconst`. Der generierte Scanner sucht stets das längste Präfix der ver-



**Abb. 2.8** Darstellung der Übergangsfunktion eines deterministischen endlichen Automaten

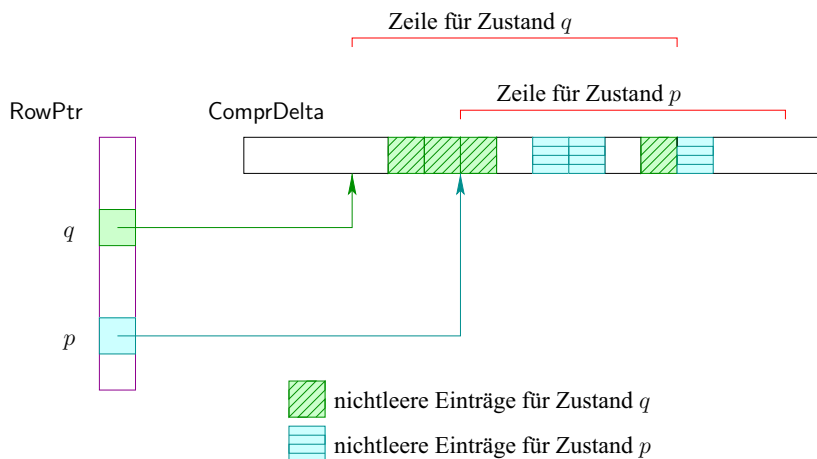
bleibenden Eingabe, das in einen Endzustand führt. Der Scanner wird also aus dem Endzustand 1 heraus einen Übergang machen, wenn dies möglich ist, d. h. wenn eine Ziffer folgt. Folgt keine Ziffer, muss der Scanner zum Endzustand 1 zurückkehren und den Lesezeiger zurücksetzen.  $\square$

#### 2.4.4 Die Implementierung eines Scanners

Das Herzstück des Scanners ist ein deterministischer endlicher Automat. Die Übergangsfunktion dieses Automaten kann durch ein zweidimensionales Feld  $\Delta$  implementiert werden. Dieses Feld wird mit dem aktuellen Zustand und der Zeichenklasse des nächsten Eingabezeichens indiziert und liefert den neuen Zustand, in den der Automat nach Lesen dieses Zeichens übergeht. Während der Zugriff auf  $\Delta[q, a]$  schnell ist, kann dagegen die Größe des Feldes  $\Delta$  gegebenenfalls Probleme bereiten. Oft enthält der deterministische endliche Automat jedoch viele Übergänge in den Fehlerzustand *error*. Diesen Zustand wählen wir deshalb als *Standardwert* (Default) für die Einträge in  $\Delta$ . Es genügt dann, nur solche Übergänge zu repräsentieren, die *nicht* in den Fehlerzustand *error* führen. Zur Repräsentation eines solchen schwach besetzten Feldes kann man verschiedene von Kompressionsverfahren anwenden. Diese sparen meist sehr viel an Platz – auf Kosten geringfügig erhöhter Zugriffszeit. Da die leeren Einträge aber zu Übergängen in den Fehlerzustand gehören, welche für die Analyse und die Fehlererkennung wichtig sind, muss die zugehörige Information weiterhin verfügbar sein.

Betrachten wir einen solchen Komprimierungsalgorithmus. Statt durch das Feld  $\Delta$  repräsentieren wir die Übergangsfunktion durch ein Feld  $\text{RowPtr}$ , welches mit Zuständen indiziert wird und dessen Komponenten Adressen der Zeilen von  $\Delta$  sind (Abb. 2.8).

Noch haben wir nichts gewonnen, sondern nur beim Zugriff Geschwindigkeit eingebüßt. Die Zeilen, auf die in  $\text{RowPtr}$  verwiesen wird, sind oft fast leer. Deshalb werden die einzelnen Zeilen in einem gemeinsamen eindimensionalen Feld  $\text{ComprDelta}$  so übereinander gelegt, dass nichtleere Einträge nicht miteinander kollidieren. Für die jeweils nächste abzulegende Zeile kann etwa die *first-fit*-Strategie angewendet werden. Die Zeile wird dann so lange über das Feld  $\Delta$  verschoben,



**Abb. 2.9** Komprimierte Darstellung der Übergangsfunktion eines deterministischen endlichen Automaten

ben, bis keine nichtleeren Einträge dieser Zeile mehr mit nichtleeren Einträgen bereits abgelegter Zeilen von Delta kollidieren. In  $\text{RowPtr}[q]$  wird dann der Index in  $\text{ComprDelta}$  abgespeichert, ab dem die  $q$ -te Zeile von Delta abgelegt ist, siehe Abb. 2.9.

Allerdings hat der dargestellte Automat jetzt die Fähigkeit verloren, undefinierte Übergänge zu erkennen: Ist etwa  $\Delta(q, a)$  undefiniert (d. h. gleich dem Fehlerzustand), könnte  $\text{ComprDelta}[\text{RowPtr}[q] + a]$  dennoch einen nichtleeren Eintrag enthalten, der aus der verschobenen Zeile eines Zustands  $p \neq q$  stammt. Deshalb wird ein weiteres Feld  $\text{Valid}$  der gleichen Länge wie  $\text{ComprDelta}$  hinzugenommen, welches angibt, zu welchen Zuständen die Einträge in  $\text{ComprDelta}$  gehören. Das heißt,  $\text{Valid}[\text{RowPtr}[q] + a] = q$  gilt genau dann, wenn  $\Delta(q, a)$  definiert ist. Die Übergangsfunktion des deterministischen endlichen Automaten kann dann durch eine Funktion  $\text{next}()$  wie folgt implementiert werden:

```

State next (State q, CharClass a) {
    if (Valid[RowPtr[q] + a]  $\neq$  q) return error;
    return ComprDelta[RowPtr[q] + a];
}

```

## 2.5 Der Sieber

Ein Scannergenerator ist ein vielseitig einsetzbares Instrument. In verschiedensten Bereichen gibt es Anwendungen für generierte Scanner, also die Aufgabe, einen Eingabestrom mit Hilfe regulärer Ausdrücke zu zerlegen. Über die reine Zerteilung

des Eingabestroms hinaus bietet ein Scanner oft die Möglichkeit an, die erkannten Symbole weiter zu verarbeiten.

Um diese erweiterte Funktionalität zu spezifizieren, wird jeder Symbolklasse zusätzlich eine semantische Aktion zugeordnet. Ein Sieber kann damit als Folge von Paaren der Form:

$$\begin{array}{ll} r_0 & \{\text{action}_0\} \\ \dots & \\ r_{n-1} & \{\text{action}_{n-1}\} \end{array}$$

spezifiziert werden. Dabei ist  $r_i$  ein (gegebenenfalls erweiterter) regulärer Ausdruck über Zeichenklassen für die  $i$ -te Symbolklasse, und  $\text{action}_i$  bezeichnet die semantische Aktion, die bei Erkennen eines Symbols dieser Klasse auszuführen ist.

Soll aus der Spezifikation eine Sieberkomponente in einer bestimmten Programmiersprache generiert werden, werden die semantischen Aktionen ebenfalls in dieser Programmiersprache ausgedrückt. Für die Aufgabe, eine Repräsentation des gefundenen Symbols zurück zu liefern, bieten sich in unterschiedlichen Programmiersprachen unterschiedliche Lösungen an. In C ist es z. B. üblich, einen *int*-Wert zurück zu liefern, der die Symbolklasse codiert, während alle weiteren Bestandteile in geeigneten globalen Variablen abzulegen sind. Etwas komfortabler könnte der Sieber in einer objekt-orientierten Programmiersprache wie JAVA realisiert werden. Hier kann eine Oberklasse Token eingeführt werden, deren Unterklassen  $C_i$  den einzelnen Symbolklassen entsprechen. Die letzte Anweisung in  $\text{action}_i$  sollte dann eine *return*-Anweisung sein, die ein Objekt der Klasse  $C_i$  zurückliefert, dessen Attribute alle Eigenschaften des identifizierten Symbols beinhalten. In einer funktionalen Programmiersprache wie OCAML kann ein Datentyp *token* bereitgestellt werden, dessen Konstruktoren  $C_i$  den verschiedenen Symbolklassen entsprechen. Die semantische Aktion  $\text{action}_i$  besteht aus einem Ausdruck vom Typ *token*, dessen Wert  $C_i(\dots)$  das identifizierte Symbol der Klasse  $C_i$  repräsentiert.

Semantische Aktionen sollten in die Lage versetzt werden, auf den Text des aktuellen Symbols zuzugreifen. In einigen generierten Scannern wird dieser deshalb in einer *globalen* Variable *yytext* zur Verfügung gestellt. Weitere globale Variablen enthalten Informationen über die *Position* des aktuellen Symbols innerhalb der Eingabe. Diese sind für die Erzeugung sinnvoller Fehlermeldungen wichtig. Semantische Aktionen sollten auch in der Lage sein, das gegebene Symbol nicht zurück zu liefern, sondern stattdessen ein weiteres Symbol aus der Eingabe anzufordern. Z. B. soll ein Kommentar überlesen oder nach der Ausführung einer Übersetzer-Direktive erst das nächste Symbol zurück geliefert werden. In einem Generator für C oder JAVA wird in der entsprechenden Aktion auf eine *return*-Anweisung verzichtet.

Aus einer solchen Spezifikation wird eine Funktion *yylex()* generiert, die bei jedem Aufruf ein weiteres Symbol zurückliefert. Nehmen wir an, für die Folge von regulären Ausdrücken  $r_0, \dots, r_{n-1}$  wäre eine Funktion *scan()* generiert worden, die das nächste Symbol in der Eingabe als Wort in der globalen Variable *yytext* ablegt und die Nummer  $i$  der Symbolklasse dieses Worts zurückliefert. Dann ist die Funktion *yylex()* gegeben durch:

```

Token yylex() {
    while(true)
        switch scan() {
            case 0      :  action0; break;
                        ...
            case n - 1  :  actionn-1; break;
            default     :  return error();
        }
    }
}

```

Hier ist die Funktion `error()` für den Fall zuständig, dass während der Identifizierung des nächsten Symbols ein Fehler auftritt. Enthält eine Aktion `actioni` keine *return*-Anweisung, springt die Programmausführung am Ende der Aktion an den Anfang der *switch*-Anweisung zurück und liest damit das nächste Symbol aus der Eingabe ein. Enthält sie dagegen eine *return*-Anweisung, wird damit die *switch*-Anweisung, die *while*-Schleife und der aktuelle Aufruf der Funktion `yylex()` verlassen.

### 2.5.1 Scannerzustände

Gelegentlich ist es nützlich, in Abhängigkeit von einem Kontext *unterschiedliche* Symbolklassen zu erkennen. Dazu stellen viele Scanner-Generatoren *Scannerzustände* zur Verfügung. Durch Lesen eines Symbols kann der Scanner von einem Zustand in einen anderen Zustand wechseln.

**Beispiel 2.5.1** Mit Hilfe von Scannerzuständen lässt sich elegant das Überlesen von Kommentaren implementieren. Dazu wird zwischen einem Zustand *normal* und einem Zustand *comment* unterschieden. Im Zustand *normal* werden die Symbole aus den Symbolklassen erkannt, die für die Programmausführung wichtig sind. Zusätzlich gibt es eine Symbolklasse *CommentInit* für das Anfangssymbol eines Kommentars, also z. B. `/*`. Als semantische Aktion für das Symbol `/*` wird auf den Zustand *comment* umgeschaltet. In dem Zustand *comment* wird nur das Endsymbol für Kommentare `*/` erkannt. Alle anderen Zeichen der Eingabe werden überlesen. Als semantische Aktion für das Endezeichen eines Kommentars wird auf den Zustand *normal* zurückgegangen.

Der aktuelle Scannerzustand kann z. B. in einer globalen Variablen `yystate` verwaltet werden. Die Zuweisung `yystate ← state` setzt dann den aktuellen Zustand auf den neuen Zustand `state`. Die Spezifikation eines Scanners mit Scannerzuständen hat die Form:

$$\begin{array}{ll}
 A_0 : & class\_list_0 \\
 & \dots \\
 A_{r-1} : & class\_list_{r-1}
 \end{array}$$

wobei  $class\_list_j$  jeweils die Folge der mit Aktionen versehenen regulären Ausdrücke für den Zustand  $A_j$  ist. Für die Zustände *normal* und *comment* aus Beispiel 2.5.1 erhalten wir etwa:

```
normal :
    /* { yystate ← comment; }
       ... // weitere Symbolklassen
comment :
    */ { yystate ← normal; }
    . { }
```

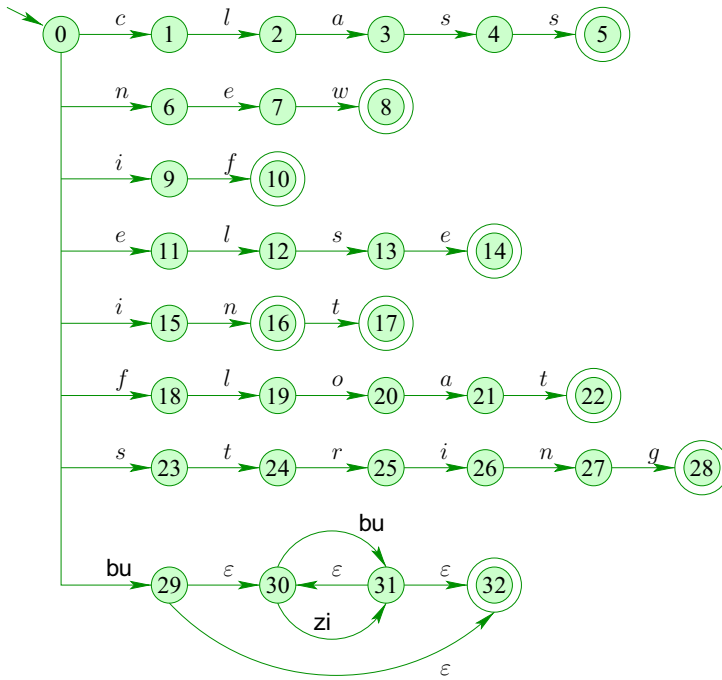
Das Zeichen `.` steht hier für ein beliebiges Eingabesymbol. Weil keine der semantischen Aktionen für Beginn, Inhalt oder Ende eines Kommentars eine *return*-Anweisung enthält, wird für den gesamten Kommentar kein Symbol zurück geliefert. □

Scannerzustände beeinflussen allein die Auswahl der Symbolklassen, aus denen Symbole erkannt werden. Um Scannerzustände zu berücksichtigen, kann deshalb das Verfahren zur Generierung der Funktion `yylex()` auf die Konkatenation der Folgen  $class\_list_j$  angewendet werden. Die einzige Funktion, die abgeändert werden muss, ist die Funktion `scan()`. Zur Bestimmung des nächsten Symbols verwendet diese Funktion nicht mehr *einen* deterministischen endlichen Automaten, sondern verfügt über einen gesonderten deterministischen endlichen Automaten  $M_j$  für jede Teilfolge  $class\_list_j$ . In Abhängigkeit von dem jeweiligen Scannerzustand  $A_j$  wird erst der zugehörige Automat  $M_j$  ausgewählt und dann zur Identifizierung des nächsten Symbols verwendet.

## 2.5.2 Die Erkennung von Schlüsselwörtern

Für die Verteilung der Aufgaben zwischen Scanner und Sieber und für die Funktionalität des Siebers gibt es viele Möglichkeiten, deren Vorteile bzw. Nachteile nicht ganz leicht zu beurteilen sind. Ein Beispiel für solche Alternativen ist die Erkennung von Schlüsselwörtern.

Nach der Aufgabenverteilung im letzten Kapitel soll der Sieber die reservierten Bezeichnungen oder Schlüsselwörter (keywords) identifizieren. Eine Möglichkeit dazu besteht darin, für jede reservierte Bezeichnung eine eigene Symbolklasse bereit zu stellen. Abbildung 2.10 zeigt einen endlichen Automaten, der einige Schlüsselwörter in Endzuständen erkennt. Rein formal gesehen haben die reservierten Bezeichnungen in C, JAVA oder OCAML jedoch oft die gleiche Struktur wie Bezeichner. Alternativ zur Beschreibung dieser Symbole durch eigene reguläre Ausdrücke und lässt sich ihre Identifizierung in die semantische Nachbearbeitung verlagern. Die Funktion `scan()` wird bei Vorliegen eines Schlüsselworts zuerst nur das Vorliegen eines Bezeichners melden. Die semantische Aktion für Bezeichner



**Abb. 2.10** Ein endlicher Automat zur Erkennung von Bezeichnern und den Schlüsselwörtern `class`, `new`, `if`, `else`, `int`, `float`, `string`.

muss später überprüfen, ob und wenn ja welches Schlüsselwort vorliegt. Diese Aufgabenverteilung hält die Mengen der Zustände und der Übergänge des Scannerautomaten klein. Allerdings muss eine effiziente Möglichkeit des Erkennens von Schlüsselwörtern bereit gestellt werden.

Als Interndarstellung werden Bezeichnern in Übersetzern oft eindeutige *int*-Werte zugeordnet. Zur Berechnung dieser Interndarstellung verwaltet der Sieber typischerweise eine *Hashtabelle*. Diese Tabelle unterstützt den effizienten Vergleich eines Wortes mit den bereits in die Tabelle eingetragenen Bezeichnern. Liegen die reservierten Bezeichner vor der lexikalischen Analyse der Eingabe in dieser Tabelle vor, kann sie der Sieber innerhalb der Klasse der Bezeichner in etwa mit dem gleichen Aufwand identifizieren, der bei der Nachbearbeitung anderer Bezeichner auftritt.



## 2.6 Übungen

### 1. Kleene-Stern

Sei  $\Sigma$  ein Alphabet und  $L, M \subseteq \Sigma^*$ . Zeigen Sie:

- (a)  $L \subseteq L^*$ .
- (b)  $\varepsilon \in L^*$ .
- (c) Falls  $u, v \in L^*$ , dann auch  $uv \in L^*$ .
- (d)  $L^*$  ist die kleinste Menge mit den Eigenschaften (1) – (3), d. h. wenn für eine Menge  $M$  gilt:  
 $L \subseteq M$ ,  $\varepsilon \in M$  und  $(u, v \in M \Rightarrow uv \in M)$ , dann ist  $L^* \subseteq M$ .
- (e) Falls  $L \subseteq M$ , dann auch  $L^* \subseteq M^*$ .
- (f)  $(L^*)^* = L^*$ .

### 2. Symbolklassen

FORTRAN erlaubt die implizite Deklaration von Bezeichnern nach ihrem Anfangsbuchstaben. Bezeichner, die mit einem der Buchstaben  $i, j, k, l, m, n$  beginnen, stehen für eine *int*-Variable oder einen *int*-Funktionswert, alle übrigen Bezeichner stehen für *float*-Werte.

Geben Sie Definitionen für die Symbolklassen FloatId und IntId an.

### 3. Erweiterte reguläre Ausdrücke

Erweitern Sie die Konstruktion eines endlichen Automaten zu einem regulären Ausdruck aus Abb. 2.3 so, dass sie direkt reguläre Ausdrücke  $r^+$  und  $r^?$  verarbeitet.  $r^+$  steht für  $rr^*$  und  $r^?$  für  $(r \mid \varepsilon)$ .

### 4. Erweiterte reguläre Ausdrücke (Forts.)

Erweitern Sie die Konstruktion eines endlichen Automaten zu einem regulären Ausdruck um eine Behandlung *zählender* Iteration, d. h. um reguläre Ausdrücke der Form:

$r\{u - o\}$       mindestens  $u$  und höchstens  $o$  aufeinanderfolgende  
Exemplare von  $r$

$r\{u-\}$       mindestens  $u$  aufeinanderfolgende Exemplare von  $r$

$r\{-o\}$       höchstens  $o$  aufeinanderfolgende Exemplare von  $r$

### 5. Deterministische Automaten

Machen Sie den endlichen Automaten aus Abb. 2.10 deterministisch.

### 6. Zeichenklassen und Symbolklassen

Gegeben seien die folgenden Definitionen von Zeichenklassen:

$$\text{bu} = a - z$$

$$\text{zi} = 0 - 9$$

$$\text{bzi} = 0 \mid 1$$

$$\text{ozi} = 0 - 7$$

$$\text{hzi} = 0 - 9 \mid A - F$$

und die Symbolklassendefinitionen

$b \text{ bzi}^+$   
 $o \text{ ozi}^+$   
 $h \text{ hzi}^+$   
 $\text{zi}^+$   
 $\text{bu (bu | zi)}^*$

- (a) Geben Sie die Einteilung in Einzelzeichenklassen an, die ein Scannergenerator berechnen würde.
  - (b) Beschreiben Sie den generierten endlichen Automaten unter Benutzung dieser Einzelzeichenklasseneinteilung.
  - (c) Machen Sie diesen endlichen Automaten deterministisch.
7. **Reservierte Bezeichner**  
 Konstruieren Sie einen deterministischen Automaten zu dem endlichen Automaten aus Abb. 2.10.
8. **Tabellenkompression**  
 Komprimieren Sie die Tabellen der von Ihnen erstellten deterministischen endlichen Automaten mittels des Verfahrens aus Abschn. 2.2.
9. **Verarbeitung römischer Zahlen**  
 (a) Geben Sie einen regulären Ausdruck für römische Zahlen an.  
 (b) Erzeugen Sie daraus einen deterministischen endlichen Automaten.  
 (c) Ergänzen Sie diesen Automaten um die Berechnung des Dezimalwerts einer römischen Zahl. Mit jedem Zustandsübergang darf der Automat eine Wertzuweisung an *eine* Variable  $w$  durchführen. Der Wert ergibt sich aus einem Ausdruck über  $w$  und Konstanten.  $w$  wird mit 0 initialisiert. Geben Sie zu jedem Zustandsübergang eine geeignete Wertzuweisung an, so dass in jedem Endzustand  $w$  den Wert der erkannten Zahl enthält.
10. **Generierung eines Scanners**  
 Generieren Sie aus einer Scanner-Spezifikation in OCAML eine OCAML-Funktion `yylex`. Verwenden Sie dabei nach Möglichkeit funktionale Konstrukte.  
 (a) Stellen Sie eine Funktion `skip` bereit, mit der das identifizierte Symbol übersprungen wird.  
 (b) Erweitern Sie Ihren Generator um Scannerzustände. Stellen Sie dazu eine Funktion `next` bereit, die als Argument den Nachfolgezustand erhält.

---

## 2.7 Literaturhinweise

Die konzeptionelle Trennung in Scanner und Sieber wurde von F. DeRemer vorgeschlagen [15]. Die Generierung von Scannern aus regulären Ausdrücken wird in vielen sogenannten Übersetzergeneratoren unterstützt. Johnson u. a. [29] beschreiben ein solches System. Das entsprechende Dienstprogramm unter UNIX, LEX,

wurde von M. Lesk entwickelt [42]. FLEX wurde von Vern Paxson geschrieben. Das in diesem Kapitel beschriebene Konzept lehnt sich an den Scannergenerator JFLEX für JAVA an.

Kompressionsmethoden für schwach besetzte Tabellen, wie sie bei der Scanner- und der Parsergenerierung erzeugt werden, werden in [61] und [11] analysiert und verglichen.

Übersetzerbau

Band 2: Syntaktische und semantische Analyse

Wilhelm, R.; Seidl, H.; Hack, S.

2012, XII, 247 S., Softcover

ISBN: 978-3-642-01134-4