4 Parser und Übersetzer

Ein *Parser* ist ein Programm, das eine Zeichenreihe in ihre syntaktischen Bestandteile zerlegt und erkennt, ob die Zeichenreihe vorgegebenen Syntaxregeln entspricht oder nicht. Ein *Übersetzer* ist ein Programm, das eine Zeichenreihe einer bestimmten Sprache in eine bestimmte Zeichenreihe einer anderen Sprache überführt. Die Zeichenreihe kann beispielsweise ein Programmtext sein, der den Syntaxregeln der verwendeten Programmiersprache genügen muss, damit er in Maschinencode übersetzt werden kann.

Parsen und Übersetzen sind jedoch keineswegs auf den Bereich der Compiler für Programmiersprachen beschränkt, sondern treten erstaunlich oft in der alltäglichen Programmierpraxis auf. Etwa beim Einlesen von Dateien, die in bestimmten Formaten vorliegen, beim Prüfen und Ausführen von Eingabekommandos, bei der Auswertung von arithmetischen Ausdrücken und vielem anderen mehr.

Die vorgegebenen Syntaxregeln liegen im allgemeinen in Form einer Grammatik vor. Mit der Recursive-Descent-Methode wird die Grammatik ziemlich unmittelbar in einen Parser und, wenn noch Übersetzungsaktionen hinzugenommen werden, in einen Übersetzer überführt.

Zur Beschreibung der Syntax von sehr einfachen Sprachen, etwa Kommandosprachen, dienen reguläre Ausdrücke. Entsprechend einfach ist im Prinzip auch der Parser; die Konstruktion des Parsers aus dem regulären Ausdruck ist jedoch mit einem gewissen Aufwand verbunden.

Wir werden diese Konstruktion in den folgenden Abschnitten durchgehen und dabei wichtige Konzepte wie reguläre Sprachen, nichtdeterministische Automaten, kontextfreie Grammatiken sowie die Recursive-Descent-Methode kennen lernen.

4.1 Regulärer Ausdruck, reguläre Sprache

Sprache, Syntax

Definition: Sei A ein Alphabet. Mit A^* bezeichnet man die Menge aller Wörter über A. Jede Teilmenge L von A^*, d.h. jede Menge von Wörtern über A, heißt *Sprache* über A.

Sprachen können endlich viele oder unendlich viele Wörter enthalten. Endliche Sprachen lassen sich einfach durch Aufzählung ihrer Wörter angeben. Um eine unendliche Sprache

angeben zu können, benötigt man eine endliche Beschreibung der Sprache. Eine solche existiert nur, wenn die Sprache nach gewissen Regeln aufgebaut ist. Die Menge dieser Regeln wird als *Syntax* der Sprache bezeichnet; sie stellt eine endliche Beschreibung der Sprache dar.

Beispiel: Sei $A = \{\mathrm{a}, \mathrm{b}\}$. Beispiele für Sprachen über A sind:

L_1	=	$\{\mathrm{a}, \mathrm{bb}, \mathrm{aab}, \mathrm{aba}, \mathrm{abb}\}$	(endliche Sprache)
L_2	=	$\{\mathrm{a}, \mathrm{aa}, \mathrm{aaa}, \mathrm{aba}, \mathrm{aaaa}, \mathrm{aaba}, \mathrm{abaa}, \ldots\}$	(unendliche Sprache)
alle Wörter, die mit a anfangen und mit a aufhören			(Syntax von L_2)

Die Syntax einer Sprache kann informell angegeben werden, wie in diesem Beispiel, aber auch formal. Es gibt unterschiedliche Möglichkeiten, die Syntax von Sprachen formal zu beschreiben. Die wichtigsten sind Grammatik, erkennender Automat und regulärer Ausdruck.

Regulärer Ausdruck

Die einfachsten unendlichen Sprachen, für die endliche Beschreibungen existieren, sind die regulären Sprachen. Die Bezeichnung weist darauf hin, dass die Wörter der Sprache sehr regulär aufgebaut sind, nämlich alle nach ein und demselben Muster, einem sogenannten regulären Ausdruck.

Ein *regulärer Ausdruck* x ist eine Formel, die eine gewisse Teilmenge von A^* erzeugt, die *reguläre Sprache* $L(x)$.

Beispiel: Sei $A = \{\mathrm{a}, \mathrm{b}\}$. Der reguläre Ausdruck

$$x \quad = \quad \mathrm{a} \mid \mathrm{a}(\mathrm{a} \mid \mathrm{b})^*\mathrm{a}$$

erzeugt die Menge aller Wörter über A, die mit a anfangen und mit a aufhören.

Der Ausdruck x ist so zu interpretieren: Die erzeugte Sprache $L(x)$ besteht aus allen Wörtern, die nur aus dem Zeichen a bestehen oder die am Anfang ein a haben, gefolgt von 0, 1 oder mehreren a's oder b's, gefolgt von a.

Reguläre Ausdrücke über einem Alphabet A sind wie folgt induktiv definiert. Das Zeichen % ist ein spezielles Zeichen, das nicht im Alphabet A vorkommt.

Definition: (Regulärer Ausdruck)

a) % ist ein regulärer Ausdruck;

b) für alle $a \in A$ ist a ein regulärer Ausdruck;

c) sind x und y reguläre Ausdrücke, so sind auch $(x \mid y), (xy)$ und x^* reguläre Ausdrücke.

Nach der obigen Definition hätte der reguläre Ausdruck aus dem Beispiel wie folgt dargestellt sein müssen:

$$x \quad = \quad (\mathrm{a} \mid ((\mathrm{a}(\mathrm{a} \mid \mathrm{b})^*)\mathrm{a})).$$

Um die Lesbarkeit zu verbessern, werden folgende Klammereinsparungsregeln vereinbart: Die Operationen „gefolgt von“ und | sind jeweils assoziativ; * bindet am stärksten, „gefolgt von“ am zweitstärksten, | am schwächsten. Damit wird der Ausdruck zu

$$x \quad = \quad \mathrm{a} \mid \mathrm{a}(\mathrm{a} \mid \mathrm{b})^*\mathrm{a}.$$

Reguläre Sprache

Ein regulärer Ausdruck ist eine Formel, die eine (endliche oder unendliche) Sprache beschreibt. Die von einem regulären Ausdruck z erzeugte reguläre Sprache $L(z)$ ergibt sich induktiv über den Aufbau von z:

Definition: Seien x, y reguläre Ausdrücke. Dann ist

$L(\%)$	$=$	$\emptyset$	(leere Sprache)
$L(a)$	$=$	$\{a\}$	für alle $a \in A$
$L(x \mid y)$	$=$	$L(x) \cup L(y)$	(Vereinigung)
$L(xy)$	$=$	$L(x)L(y)$	(Produkt)
$L(x^*)$	$=$	$L(x)^*$	(Abschluss)

Die Definition von Produkt und Abschluss von Sprachen ist im Anhang gegeben. Das spezielle Zeichen % wird gebraucht, damit die leere Sprache durch einen regulären Ausdruck erzeugt werden kann. Die Sprache $\{\varepsilon\}$, die nur aus dem leeren Wort besteht, wird von dem Ausdruck $\%^*$ erzeugt.

Die von einem regulären Ausdruck erzeugte Sprache ist eindeutig bestimmt. Umgekehrt können jedoch verschiedene reguläre Ausdrücke dieselbe Sprache erzeugen.

Beispiel: Die regulären Ausdrücke $x = \mathrm{a}(\mathrm{a} \mid \mathrm{b})$ und $y = \mathrm{aa} \mid \mathrm{ab}$ erzeugen beide die Sprache $L(x) = L(y) = \{\mathrm{aa}, \mathrm{ab}\}$.

Es ist wichtig, zwischen den Begriffen „regulärer Ausdruck“ und „reguläre Sprache“ zu unterscheiden. Ein regulärer Ausdruck ist, wie wir gesehen haben, eine Formel, ein Muster für Wörter. Um zu den entsprechenden Wörtern zu kommen, muss die Formel nach den oben angegebenen Regeln ausgewertet werden. Das Ergebnis der Auswertung ist die erzeugte reguläre Sprache. Eine Sprache heißt regulär, wenn es einen regulären Ausdruck gibt, der sie erzeugt.

4.2 Erkennung von regulären Sprachen

Gegeben sei ein regulärer Ausdruck x und ein Wort w. Es stellt sich die Frage, ob w zu $L(x)$ gehört. Um dies zu entscheiden, wird aus dem regulären Ausdruck x ein endlicher Automat $N(x)$ konstruiert. Der Automat erhält das Wort w als Eingabe. Beginnend in einem Startzustand, arbeitet er das Wort w zeichenweise ab und vollführt bei jedem Zeichen einen Zustandsübergang. Endet er in einem besonders gekennzeichneten Endzustand, so akzeptiert er das Wort w. Der Automat wird so konstruiert, dass er genau alle jene Wörter akzeptiert, die zu $L(x)$ gehören.

Nichtdeterministischer endlicher Automat

Definition: Ein *nichtdeterministischer endlicher Automat* ist ein Quintupel

$$N = (Z, A, d, s_0, F).$$

Hierbei ist

Z	eine endliche Menge von *Zuständen*,
A	das *Eingabealphabet*,
d	die *Übergangsrelation* $d \subseteq Z \times A \times Z$,
$s_0 \in Z$	der *Anfangszustand*,
$F \subseteq Z$	die Menge der *Endzustände*.

Ein *deterministischer endlicher Automat* ist ein Spezialfall des nichtdeterministischen endlichen Automaten, bei dem die Übergangsrelation eine *Übergangsfunktion* ist:

$$d : Z \times A \to Z$$

Ein Element (s, a, s') der Übergangsrelation d gibt für das Paar bestehend aus Zustand s und Eingabezeichen a den Zustand s' als möglichen Folgezustand an.[1]

Beim deterministischen endlichen Automaten ist der Folgezustand für alle $(s, a) \in Z \times A$ eindeutig bestimmt. Beim nichtdeterministischen endlichen Automaten kann es zu einem $(s, a) \in Z \times A$ mehrere mögliche Folgezustände geben, oder auch gar keinen.

Die Tatsache, dass es beim nichtdeterministischen endlichen Automaten i.a. keinen eindeutig bestimmten Folgezustand, sondern mehrere mögliche Folgezustände gibt, ist eine gedanklich schwierige (und erst recht technisch schwer umsetzbare) Konstruktion. Eine Möglichkeit der Vorstellung ist, dass der Automat wählt, welchen von den

[1] Eine andere Darstellung der Übergangsrelation d eines nichtdeterministischen endlichen Automaten ist die einer *Übergangsfunktion* f, die jedem Paar aus Zustand und Eingabezeichen eine *Menge* von möglichen Folgezuständen zuordnet:
$f : Z \times A \to \mathcal{P}(Z)$ mit $s' \in f(s, a) \Leftrightarrow (s, a, s') \in d$.

möglichen Folgezuständen er annimmt. Eine andere Vorstellung ist, dass der Automat entsprechend viele Kopien von sich herstellt, die jeweils in den verschiedenen möglichen Folgezuständen weiterarbeiten.

Anschaulich lässt sich ein nichtdeterministischer endlicher Automat durch seinen Zustandsgraphen darstellen.

Definition: Der *Zustandsgraph* eines nichtdeterministischen endlichen Automaten (Z, A, d, s_0, F) ist ein Graph $G = (V, E, h)$. Hierbei ist $V = Z$, d.h. die Knoten des Graphen sind die Zustände des Automaten; Anfangszustand und Endzustände werden geeignet gekennzeichnet.

Ein Zustand r ist mit einem Zustand s durch eine Kante (r, s) verbunden, wenn s Folgezustand von r unter einem Zeichen $a \in A$ ist.

Die Abbildung $h : E \to \mathcal{P}(A)$ ist eine *Kantenbeschriftung*; jede Kante (r, s) ist mit der Menge aller Zeichen beschriftet, unter denen Zustand r nach Zustand s übergeht.

Beispiel: In Bild 4.1a ist der Zustandsgraph eines nichtdeterministischen Automaten abgebildet, hierbei sind die Mengenklammern um die Beschriftungen der Kanten weggelassen.

Die Zustandsmenge ist $Z = \{0, ..., 3\}$, der Anfangszustand ist $s_0 = 0$, die Menge der Endzustände ist $F = \{2, 3\}$, und das Eingabealphabet ist $A = \{\mathrm{a}, \mathrm{b}\}$. Die Übergangsrelation d enthält folgende Zustandsübergänge:

$(0, \mathrm{a}, 1)$
$(0, \mathrm{a}, 3)$
$(1, \mathrm{a}, 1)$
$(1, \mathrm{b}, 1)$
$(1, \mathrm{a}, 2)$
$(3, \mathrm{a}, 3)$

Definition: Sei $e_0 \dots e_{n-1}$ mit $e_i \in E, n \in \mathbb{N}_0$ ein Pfad im Zustandsgraphen G eines nichtdeterministischen Automaten. Die *Beschriftung des Pfades* ergibt sich als Produkt[2] der Markierungen seiner Kanten:

$$h(e_0 \dots e_{n-1}) \quad = \quad h(e_0) \dots h(e_{n-1}).$$

Definition: Ein nichtdeterministischer endlicher Automat N *erkennt* (*akzeptiert*) das Wort w, wenn es in seinem Zustandsgraphen einen Pfad vom Anfangszustand zu einem Endzustand gibt, dessen Beschriftung das Wort w enthält. Die Menge aller Wörter, die N erkennt, ist die von N *erkannte Sprache* $L(N)$.

[2]Das Produkt von Mengen von Wörtern ist im Anhang definiert

Beispiel: Im Zustandsgraphen des nichtdeterministischen endlichen Automaten ist in Bild 4.1b ein Pfad vom Anfangszustand zu einem Endzustand hervorgehoben. Die Beschriftung des Pfades ergibt sich als $\{\text{aaa}, \text{aba}\}$, sie enthält das Wort $w = \text{aba}$. Das Wort aba wird also von dem Automaten erkannt.

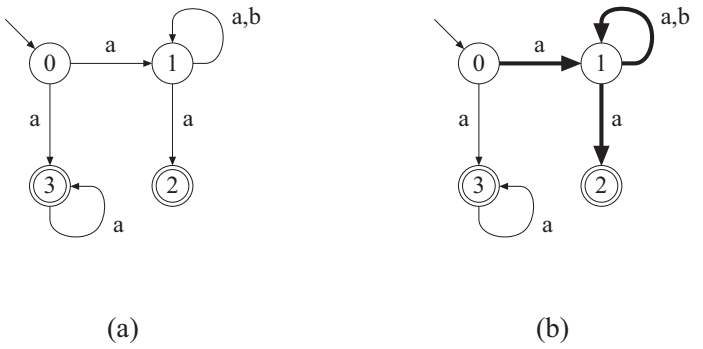


Bild 4.1: *Zustandsgraph eines nichtdeterministischen endlichen Automaten (a), Pfad vom Anfangszustand zu einem Endzustand (b)*

In unserer Vorstellung von der Arbeitsweise eines nichtdeterministischen endlichen Automaten ergibt sich der in diesem Beispiel hervorgehobene Pfad dadurch, dass der Automat ausgehend vom Startzustand immer die richtigen Zustandsübergänge wählt, die mit dem Eingabewort w zu einem Endzustand führen – wie ein Schlafwandler, der seine Schritte unbewusst immer richtig setzt.

Die andere Vorstellung ist die, dass der Automat entsprechend viele Kopien von sich herstellt, wenn mehrere Zustandsübergänge möglich sind, und dass zumindest eine der Kopien während der Verarbeitung des Eingabewortes w den Pfad zum Endzustand beschreitet.

Nichtdeterministischer endlicher Automat mit ε-Übergängen

Nichtdeterministische endliche Automaten lassen sich häufig leichter konstruieren, wenn die Übergangsrelation so erweitert wird, dass auch Zustandsübergänge möglich sind, ohne dass ein Zeichen des Eingabewortes abgearbeitet wird. Ein solcher Zustandsübergang wird als ε-Übergang bezeichnet. Es lässt sich zeigen, dass es zu jedem nichtdeterministischen endlichen Automaten mit ε-Übergängen einen normalen nichtdeterministischen endlichen Automaten ohne ε-Übergänge gibt, der dieselbe Sprache erkennt, und umgekehrt [AHU 74]. Die beiden Maschinenmodelle sind also äquivalent. Wir werden im Folgenden bei nichtdeterministischen endlichen Automaten ε-Übergänge zulassen.

Formal ist die Übergangsrelation d eines nichtdeterministischen endlichen Automaten mit ε-Übergängen definiert als

$$d \subseteq Z \times A' \times Z$$

mit $A' = A \cup \{\varepsilon\}$.

Beispiel: In Bild 4.2 ist ein nichtdeterministischer endlicher Automat mit ε-Übergängen dargestellt.

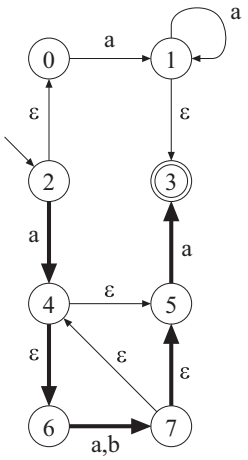


Bild 4.2: *Zustandsgraph eines nichtdeterministischen endlichen Automaten mit ε-Übergängen*

Simulation eines nichtdeterministischen endlichen Automaten

Es ist im allgemeinen nicht offensichtlich, ob ein gegebener nichtdeterministischer endlicher Automat ein bestimmtes Wort w erkennt, d.h. ob es in seinem Zustandsgraphen einen Pfad gibt, dessen Beschriftung das Wort w enthält. Dies lässt sich jedoch herausfinden, indem eine Breitensuche in dem Zustandsgraphen des Automaten durchgeführt wird. Dieses Verfahren wird als *Simulation* des Automaten bezeichnet.

Die Simulation eines nichtdetermistischen endlichen Automaten kann man sich als eine Art „Mensch-ärgere-dich-nicht“-Spiel vorstellen. Der Zustandsgraph des Automaten ist das Spielbrett. Zustände werden markiert, indem nach bestimmten Regeln Spielsteine auf die Felder gesetzt werden. Es gibt nur einen Spieler; dieser liest die Zeichen des zu untersuchenden Wortes w.

Zu Beginn der Simulation wird ein Spielstein auf den Anfangszustand gesetzt. Es folgen dann abwechselnd zwei Arten von Zügen:

a) Spielsteine, die einen Epsilon-Übergang machen können, werden geklont. D.h. wenn ein freies Feld durch einen Epsilon-Übergang mit einem Feld verbunden ist, auf dem ein Spielstein steht, so wird auch auf das freie Feld ein Spielstein gesetzt (Bild 4.3).

b) Wird das Zeichen a gelesen, so rücken alle Spielsteine, die einen Übergang mit dem Zeichen a machen können, auf das entsprechende Feld vor. Spielsteine, die auf mehrere Felder vorrücken können, werden geklont (Bild 4.4). Alle Spielsteine, die keinen Übergang mit dem Zeichen a machen können, werden entfernt (Bild 4.5).

Das Spiel ist „gewonnen“, wenn nach Zug a) ein Spielstein auf einem Endzustand steht. Der Automat hat dann das Wort w, das aus der Folge der bis hierhin gelesenen Zeichen besteht, erkannt.

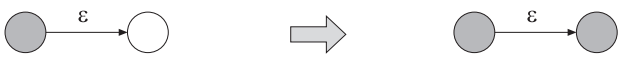


Bild 4.3: *Epsilon-Übergang*

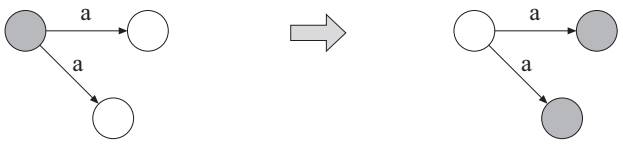


Bild 4.4: *Zustandsübergang bei gelesenem Zeichen a*

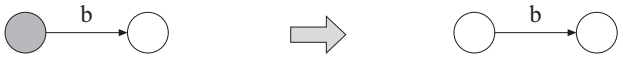


Bild 4.5: *Zustandsübergang bei gelesenem Zeichen a*

Es folgt die formale Beschreibung des Verfahrens zur Simulation eines nichtdeterministischen endlichen Automaten.

1) Initialisierung:

a) Markiere den Anfangszustand;
b) markiere alle Zustände, die mit Epsilon-Übergängen erreichbar sind.

2) Für jedes gelesene Eingabezeichen:
   a) Markiere alle Folgezustände unter dem Eingabezeichen;
   b) markiere alle Zustände, die mit Epsilon-Übergängen erreichbar sind.

Ist am Ende ein Endzustand markiert, so hat der Automat das Eingabewort erkannt.

Konstruktion eines nichtdeterministischen endlichen Automaten aus einem regulären Ausdruck

Zu jedem regulären Ausdruck z gibt es einen nichtdeterministischen endlichen Automaten $N(z)$, der die von z erzeugte reguläre Sprache $L(z)$ erkennt. $N(z)$ wird induktiv über den Aufbau von z konstruiert [AHU 74].

Die elementaren Automaten für $a \in A$ und für das spezielle Zeichen % sind in Bild 4.6 dargestellt:

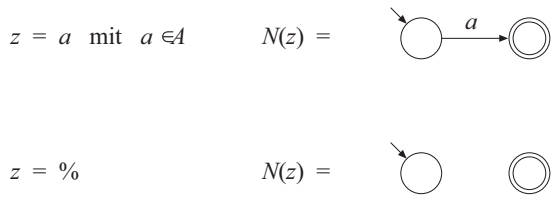


Bild 4.6: *Elementare Automaten für die regulären Ausdrücke $z = a$ mit $a \in A$ und $z = \%$*

Wenn also der reguläre Ausdruck z nur aus einem einzelnen Alphabetzeichen a besteht, dann ist der zugehörigen Automat $N(z)$ sehr einfach: er hat einen Anfangszustand, einen Endzustand und einen Zustandsübergang, nämlich vom Anfangszustand zum Endzustand unter dem Zeichen a. Der Automat erkennt also die Menge $\{a\}$, und dies ist auch genau die Sprache, die von dem regulären Ausdruck a erzeugt wird.

Wenn der reguläre Ausdruck z nur aus dem Zeichen % besteht, dann hat der zugehörige Automat nur einen Anfangs- und einen Endzustand ohne Zustandsübergang. Der Automat kann also nie in den Endzustand kommen; die erkannte Sprache ist daher die leere Menge, und dies ist genau die Sprache, die von % erzeugt wird.

Seien nun x und y reguläre Ausdrücke und $N(x)$ und $N(y)$ die zugehörigen Automaten. Dann werden die Automaten für $x \mid y$, xy und x^* wie in Bild 4.7 dargestellt konstruiert:

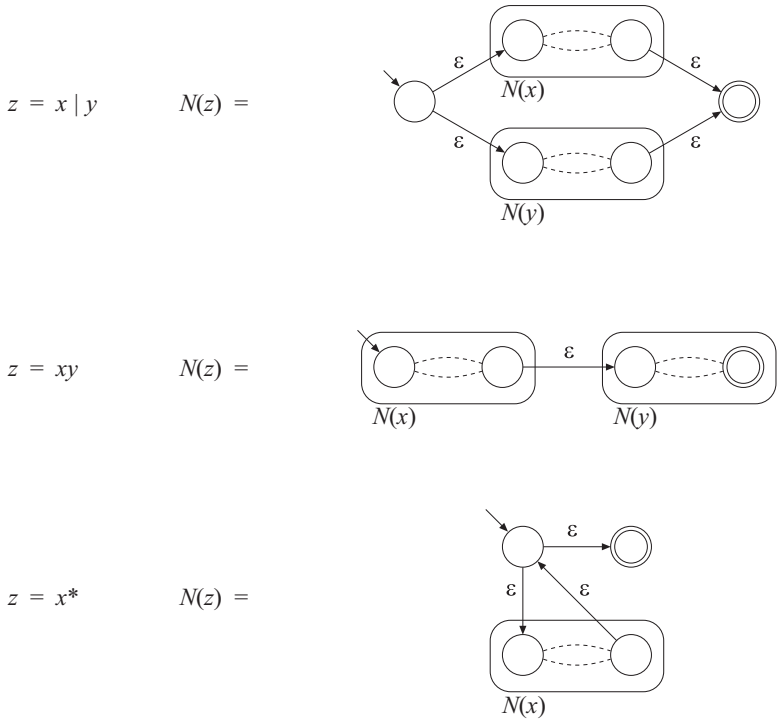


Bild 4.7: *Automaten $N(z)$ für $z = x \mid y$, $z = xy$ und $z = x^*$*

Die auf diese Weise konstruierten Automaten enthalten viele Epsilon-Übergänge. Die Anzahl von Zuständen ist dadurch größer als eigentlich notwendig. Auf der anderen Seite haben die Automaten jedoch die Eigenschaft, dass jeder Zustand maximal zwei Folgezustände hat. Dies vereinfacht die Implementation.

Beispiel: Folgendes Bild 4.8 zeigt den nach dieser Methode konstruierten nichtdeterministischen endlichen Automaten $N(x)$, der zu dem regulären Ausdruck $x = (\mathrm{a} \mid \mathrm{b})^*\mathrm{a}$ gehört.

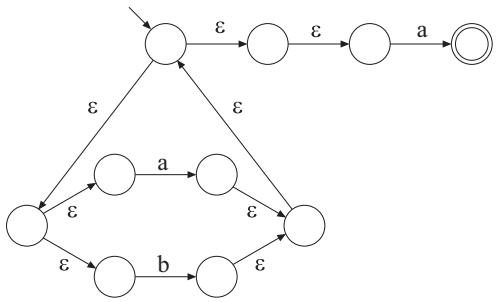


Bild 4.8: *Nichtdeterministischer Automat für* $(a \mid b)^* a$

Recursive-Descent-Übersetzung von regulären Ausdrücken

Um für einen beliebigen regulären Ausdruck x und ein beliebiges Wort w über einem Alphabet A zu entscheiden, ob w von x erzeugt wird, sind zwei Schritte nötig:

1) Konstruktion eines nichtdeterministischen endlichen Automaten $N(x)$ aus dem regulären Ausdruck x
2) Simulation des nichtdeterministischen endlichen Automaten $N(x)$ mit dem Eingabewort w

Wenn der Automat das Wort erkennt, wird es von dem regulären Ausdruck erzeugt und sonst nicht.

Es wird jetzt noch ein Verfahren gebraucht, das den Aufbau eines regulären Ausdrucks analysiert und nach der oben beschriebenen Methode den zugehörigen nichtdeterministischen endlichen Automaten konstruiert. In klassischer Weise wird dies von einem Recursive-Descent-Übersetzer geleistet, der einen regulären Ausdruck in den zugehörigen nichtdeterministischen Automaten übersetzt. Das folgende Bild 4.9 illustriert diese Vorgehensweise.

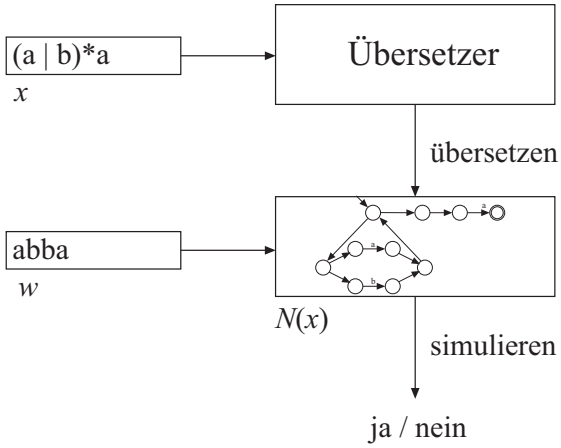


Bild 4.9: *Prüfen, ob das Wort w von dem regulären Ausdruck x erzeugt wird*

4.3 Recursive-Descent-Übersetzung

Die Methode der Übersetzung durch rekursiven Abstieg (engl.: *recursive descent*) wird im Folgenden anhand von Beispielen dargestellt. Das erste Beispiel ist bewusst sehr einfach gewählt, um das Prinzip der Recursive-Descent-Methode darzustellen.

Im ersten Beispiel soll ein Parser für einstellige ganze Zahlen mit oder ohne negatives Vorzeichen erstellt werden. Ein Parser ist ein Programm, das eine Zeichenreihe in ihre syntaktischen Bestandteile zerlegt und erkennt, ob die Zeichenreihe den vorgegebenen Syntaxregeln entspricht oder nicht. Somit ist ein Parser ein sehr einfacher Übersetzer, der alle syntaktisch korrekt aufgebauten Zeichenreihen in den Wert *true* übersetzt und ansonsten eine Fehlermeldung ausgibt.

Syntax

Die Syntaxregeln werden formal durch eine Grammatik angegeben. Der Begriff der Grammatik ist im Kapitel Grundlagen im Anhang definiert.

Die Zahlen sollen zunächst nur aus einer Ziffer bestehen. Wir benutzen folgende Grammatik $G = (V, T, P, S)$; es sind

V = $\{number, sign, digit\}$ die Menge der Nichtterminalzeichen,

T = $\{0, ..., 9, -\}$ die Menge der Terminalzeichen,

P : *number* $\rightarrow$ *sign digit*

sign $\rightarrow$ $-$ | ε

digit $\rightarrow$ 0 | ... | 9

die Produktionen und

S = *number* das Startsymbol.

Syntaktisch korrekte Zahlen entsprechend dieser Grammatik sind also beispielsweise $3, -5, -0$ oder 6. Bild 4.10 zeigt den Ableitungsbaum für die Zahl -5.

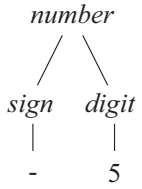


Bild 4.10: *Ableitungsbaum für* -5

Recursive-Descent-Methode

Bei der *Recursive-Descent-Methode* wird für jedes Nichtterminalzeichen eine Funktion geschrieben. In der Funktion wird die rechte Seite der zugehörigen Produktion behandelt, und zwar jedes Nichtterminalzeichen durch einen entsprechenden Funktionsaufruf und jedes Terminalzeichen durch Abarbeiten des entsprechenden Zeichens vom Eingabewort.

Hat eine Produktion auf der rechten Seite mehrere Alternativen, so wird in der Funktion eine entsprechende Fallunterscheidung durchgeführt. Grundlage der Entscheidung, welche Alternative angewendet wird, ist das nächste abzuarbeitende Zeichen des Eingabewortes (der „Lookahead“). Die Grammatik muss so gestaltet sein, dass aufgrund des Lookaheads immer eine Entscheidung möglich ist.

Die Funktion für das Nichtterminalzeichen *number* ist sehr einfach: Da es nur die Produktion *number* $\rightarrow$ *sign digit* gibt, besteht die Funktion aus einem Funktionsaufruf für *sign* gefolgt von einem Funktionsaufruf für *digit*.

```
void number()
{
    sign();
    digit();
}
```

Die Funktion für das Nichtterminalzeichen *sign* muss die beiden Alternativen der Produktion unterscheiden. Wenn das nächste Zeichen des Eingabewortes ein Minuszeichen ist, wird die erste Alternative angewendet, anderenfalls die zweite.

Die Funktion *lookahead* liefert das nächste Zeichen des Eingabewortes; die Funktion *consume* arbeitet das Zeichen ab. Die genaue Implementation dieser Funktionen ist weiter unten angegeben.

```
void sign()
{
    String z=lookahead();
    if (z.equals("-"))
        consume(z);
    else;    // Epsilon-Produktion: tue nichts
}
```

Die Funktion für das Nichtterminalzeichen *digit* behandelt alle Alternativen in gleicher Weise und ist daher recht einfach. Die Funktion *isDigit* liefert *true*, wenn der Parameter z eines der Zeichen „0“, ..., „9“ ist. Wenn keine Ziffer gefunden wird, löst die Funktion einen Fehler aus. Der Fehler wird in einer übergeordneten Funktion abgefangen und ausgewertet.

```
void digit()
{
    String z=lookahead();
    if (isDigit(z))
        consume(z);
    else
        throw new RuntimeException("Ziffer erwartet");
}
```

Klassen *Parser* und *SimpleNumberParser*

Die angegebenen drei Funktionen sind eingebettet in eine Klasse *SimpleNumberParser*, die von der Basisklasse *Parser* abgeleitet ist. Die Klasse *Parser* enthält das zu analysierende Eingabewort v sowie einige grundlegende Funktionen, so etwa *lookahead*, *consume*, und *isDigit*, ferner *trymatch*(a) (prüft ob das Zeichen a am Anfang des Eingabewortes steht und arbeitet es gegebenenfalls ab) und *match*(a) (macht dasselbe, aber erzeugt einen Fehler, wenn das Zeichen a nicht am Anfang des Eingabewortes steht).

```
class Parser
{
    protected static final String ALPHA="abcdefghijklmnopqrstuvwxyz";
    protected static final String NUM="0123456789";
    protected String v;    // zu analysierendes Eingabewort
    public String errorMessage;
    public int errorPosition;


    protected String lookahead()
    {
        if (v.length()>0)
            return v.substring(0,1);    // erstes Zeichen
        return "";
    }


    protected void consume(String a)
    {
        v=v.substring(1);    // Rest außer dem ersten Zeichen
    }


    protected boolean trymatch(String a)
    {
        if (v.startsWith(a))
        {
            consume(a);
            return true;
        }
        return false;
    }


    protected void match(String a)
    {
        if (!trymatch(a))
            throw new RuntimeException("Zeichen "+a+" erwartet");
    }


    protected boolean isIn(String a, String set)
    {
        if (a.length()>0)
            return set.indexOf(a)>=0;
        return false;
    }
```

```
    protected boolean isLetter(String a)
    {
        return isIn(a, ALPHA);
    }



    protected boolean isDigit(String a)
    {
        return isIn(a, NUM);
    }

}    // end class Parser
```

Es folgt die Klasse *SimpleNumberParser*. Sie enthält außer den drei Funktionen für die Nichtterminalzeichen die Methode *parse*. In *parse* werden ausgelöste Fehler abgefangen und die entsprechende Fehlermeldung und die Position des Fehlers ausgewertet.

```
class SimpleNumberParser extends Parser
{
    public void parse(String w)
    {
        v=w;
        errorMessage="";
        try
        {
            number();
            if (v.length()>0)
                throw new RuntimeException("Überzählige Zeichen");
        }
        catch (RuntimeException e)
        {
            errorMessage=e.getMessage();
        }
        errorPosition=w.length()-v.length();
    }



    private void number()
    {
        sign();
        digit();
    }
```

```
    private void sign()
    {
        String z=lookahead();
        if (z.equals("-"))
            consume(z);
        else;    // Epsilon-Produktion: tue nichts
    }

    private void digit()
    {
        String z=lookahead();
        if (isDigit(z))
            consume(z);
        else
            throw new RuntimeException("Ziffer erwartet");
    }

}    // end class SimpleNumberParser
```

Auswertung von Ausdrücken

Im zweiten Beispiel soll ein Übersetzer zur Auswertung von Additions-/Subtraktions-Ausdrücken erstellt werden. Die Ausdrücke sind als Strings gegeben und sollen in ihren Wert „übersetzt“ werden. Zunächst sollen die Zahlen in den Ausdrücken nur aus einzelnen Ziffern bestehen (Beispiel: Der Ausdruck „$9-3-4$“ liefert das Ergebnis 2).

Die Grammatik $G = (V, T, P, S)$ für die Ausdrücke ist wie folgt gegeben. Es sind

V = $\{expr, number\}$ die Menge der Nichtterminalzeichen,

T = $\{0, 1, ..., 9, +, -\}$ die Menge der Terminalzeichen,

P : $expr \rightarrow expr + number \mid expr - number \mid number$

$number \rightarrow 0 \mid 1 \mid \ldots \mid 9$

die Produktionen und

S = $expr$ das Startsymbol.

In Bild 4.11a ist der Ableitungsbaum für den Ausdruck „$9-3-4$“ dargestellt. Der Ausdruck wird korrekt von links nach rechts zusammengefasst. Bild 4.11 zeigt den entsprechenden Datenfluss bei der Auswertung.

Leider verträgt sich die Auswertung von links nach rechts nicht ohne weiteres mit der Recursive-Descent-Methode. Denn die Umsetzung der ersten Produktion würde zu folgendem Programm führen:

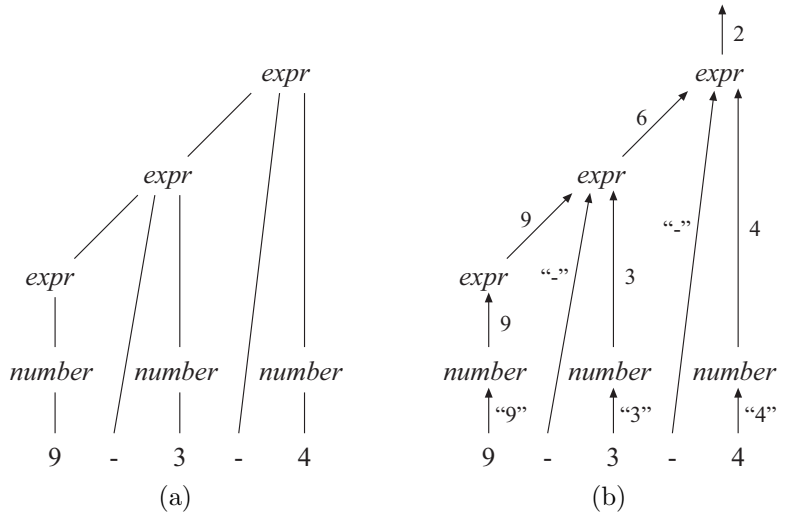


Bild 4.11: *(a) Ableitungsbaum und (b) Datenflussgraph für den Ausdruck „9 − 3 − 4“*

```
void expr()
{
    expr();
    match("+");
    number();
}
```

Hier tritt in der ersten Anweisung bereits wieder ein rekursiver Aufruf von *expr* auf, ohne dass ein Zeichen des Eingabewortes verarbeitet wurde. Dies führt zu einer unendlichen Kette von Aufrufen von *expr* und damit irgendwann zum Abbruch des Programms mit der Fehlermeldung „stack overflow“.

Derartige Linksrekursionen müssen also vermieden werden. Es gibt die Möglichkeit, die Grammatik entsprechend umzuformen.

Beseitigung von Linksrekursionen

Die Grammatik G soll so umgeformt werden, dass sie keine linksrekursiven Produktionen enthält.

Allgemeine Vorgehensweise: Jede Produktion der Form

$$X \rightarrow Xw \mid u$$

wird ersetzt durch die Produktionen

$$X \rightarrow uR$$

$$R \rightarrow wR \mid \varepsilon$$

Hierbei ist R ein neu eingeführtes Nichtterminalzeichen.

Angewandt auf die Grammatik G führt dies zu folgenden Produktionen (*expr* entspricht X, *number* entspricht u, + *number* bzw. - *number* entspricht w, *rest* steht für das neu eingeführte Nichtterminalzeichen R):

$$expr \rightarrow number\ rest$$

$$rest \rightarrow +\ number\ rest \mid -\ number\ rest \mid \varepsilon$$

Leider hat die Beseitigung der Linksrekursionen in der Grammatik auch die Auswertungsreihenfolge geändert. Der Ableitungsbaum für den Ausdruck „9-3-4“ zeigt, dass von rechts nach links zusammengefasst wird (Bild 4.12a). Dennoch ist es möglich, den Ausdruck korrekt auszuwerten, jedoch nur, indem auch Datenfluss von oben nach unten im Baum stattfindet. Bild 4.12b zeigt den entsprechenden Datenflussgraphen.

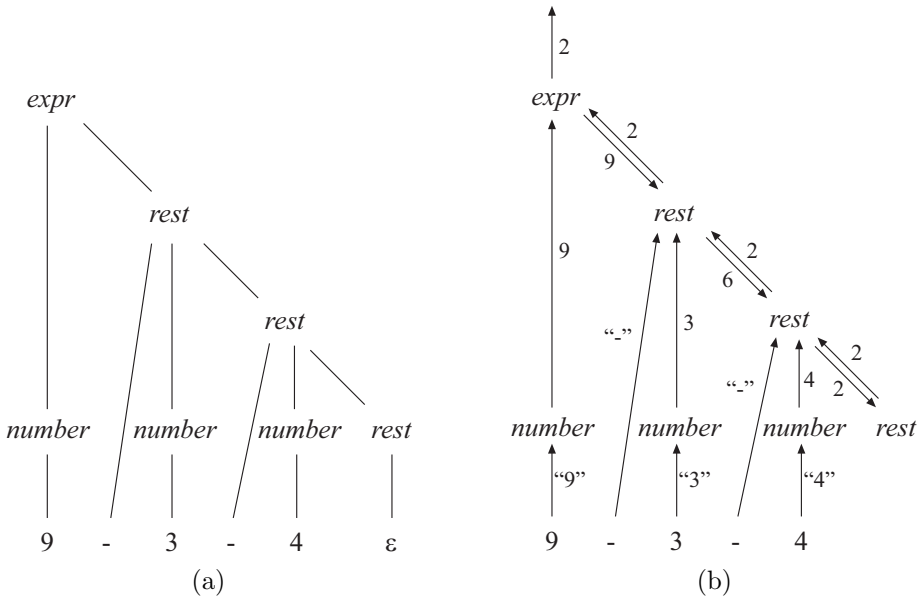


Bild 4.12: *Neuer Ableitungsbaum und Datenflussgraph für den Ausdruck „9 − 3 − 4“*

Übersetzungsaktionen

Aus der Analyse des Datenflussgraphen werden Übersetzungsaktionen hergeleitet, die

zu den Produktionen hinzugefügt werden. Im Unterschied zum Parser erzeugen die den Nichtterminalzeichen entsprechenden Funktionen nun jeweils einen Wert und bekommen einen Wert als Parameter übergeben.[3]

Betrachten wir beispielsweise das erste Vorkommen von *rest* im Datenflussgraphen von Bild 4.12b. Die entsprechende Funktion *rest* bekommt die 9 als Parameter übergeben. Sie ruft *number* auf und subtrahiert den Rückgabewert 3 von 9. Das Ergebnis 6 übergibt sie als Parameter an den rekursiven Aufruf von *rest*. Den Rückgabewert 2 dieses rekursiven Aufrufs gibt sie ihrerseits als Rückgabewert zurück.

Das folgende Bild 4.13 zeigt den Ableitungsbaum für eine allgemeine Produktion $P_0 \to P_1P_2P_3$ und den zugehörigen Datenflussgraphen der Übersetzungsaktionen (die Verallgemeinerung für $P_0 \to P_1 \dots P_n$ mit beliebigem n ist offensichtlich).

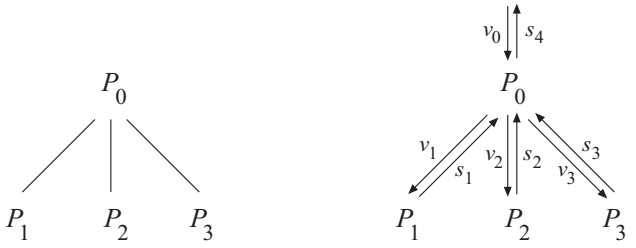


Bild 4.13: *Ableitungsbaum und Datenflussgraph für die allgemeine Produktion $P_0 \to P_1P_2P_3$*

Die entsprechenden Übersetzungsaktionen werden zu der Produktion $P_0 \to P_1P_2P_3$ nach folgendem Schema hinzugefügt:

Produktion	**Übersetzungsaktionen**	
$P_0 \to$		$s_0 = v_0$
P_1	$v_1 = f_1(s_0)$	$s_1 = P_1(v_1)$
P_2	$v_2 = f_2(s_0, s_1)$	$s_2 = P_2(v_2)$
P_3	$v_3 = f_3(s_0, s_1, s_2)$	$s_3 = P_3(v_3)$
	$v_4 = f_4(s_0, s_1, s_2, s_3)$	$s_4 = v_4$

Die v_i und s_i sind Variablen. Die Variable v_0 ist der Parameter der Funktion P_0 und die Variable s_4 der Rückgabewert der Funktion P_0. Man nennt v_0 das *vererbte Attribut* und s_4 das *synthetisierte Attribut* der Produktion. Die Funktionen f_i entsprechen den

[3] Der Begriff „Wert“ ist nicht wie in diesem Beispiel auf einen Zahlenwert beschränkt, sondern der Wert kann auch ein String sein oder eine Referenz auf ein beliebiges Objekt.

Übersetzungsaktionen, die mit den Variablen v_0, ..., v_{i-1} angestellt werden. Die $P_i(v_i)$ sind Funktionsaufrufe, die zu den Zeichen P_i gehören. Ist P_i ein Terminalzeichen, so wird die Funktion $match(P_i)$ aufgerufen.

Konkret angewandt auf die oben angegebene Produktion *rest* → − *number rest* ergibt sich das folgende Schema, indem die entsprechenden Zeichen für P_i und die gewünschten Übersetzungsaktionen für f_i eingesetzt werden. Nicht benötigte Variablen sind weggelassen worden.

Produktion	**Übersetzungsaktionen**	
rest →		$s_0 = v_0$
-		$match(\text{„-“})$
number		$s_2 = number()$
rest	$v_3 = s_0 - s_2$	$s_3 = rest(v_3)$
	$v_4 = s_3$	$s_4 = v_4$

Welche Übersetzungsaktionen konkret einzusetzen sind, lässt sich wie oben gesehen am einfachsten durch Analyse des Datenflussgraphen eines Beispielausdrucks bestimmen.

So ergeben sich die Berechnungen, die bei Anwendung der Produktion *rest* → − *number rest* auszuführen sind: die Funktion *rest* vermindert den Wert, den sie von oben bekommt ($v_0 = s_0 = 9$), um den Wert, den sie von *number* bekommt ($s_2 = 3$). Hieraus ergibt sich die Übersetzungsaktion $v_3 = s_0 - s_2$. Das Ergebnis ($v_3 = 6$) übergibt sie als Parameter einem weiteren Aufruf von *rest*. Den Wert, den sie von diesem Aufruf von *rest* zurückerhält ($s_3 = 2$), gibt sie ihrerseits zurück ($s_4 = 2$).

Die direkte Implementation der Übersetzungsaktionen liefert folgendes Programm:

```
int rest(int v0){  s0=v0;
                   match("-");
                   s2=number();
v3=s0-s2;          s3=rest(v3);
v4=s3;             return v4;}
```

Die Implementation lässt sich vereinfachen, indem anstelle der Variablen gleich die entsprechenden Ausdrücke eingesetzt werden, etwa wie folgt:

```
int rest(int v0)
{
    match("-");
    return rest(v0-number());
}
```

Soweit ist in der Funktion *rest* nur die Produktion *rest* → − *number rest* implementiert, nicht jedoch die Alternativen *rest* → +*number rest* und *rest* → ε. Welche der Produktionen anzuwenden ist, muss bei Ausführung der Funktion anhand des nächsten Zeichens

des Eingabewortes entschieden werden: Je nach dem, ob das erste Zeichen ein Minus- oder ein Pluszeichen oder keines von beiden ist, ist die jeweilige Alternative anzuwenden.

Die Funktion *trymatch* versucht das angegebene Zeichen abzuarbeiten und liefert im Erfolgsfall *true* zurück. Die vollständige Implementation der Funktion *rest* lautet somit wie folgt:

```
int rest(int v0)
{
    if (trymatch("+"))
        return rest(v0+number());
    if (trymatch("-"))
        return rest(v0-number());
    return v0;
}
```

Übersetzer für Integer-Ausdrücke

Der vollständige Übersetzer enthält noch die Funktionen *expr* und *number*. Die Funktion *number* ist hier so implementiert, dass auch mehrstellige Zahlen verarbeitet werden.

```
class SimpleExprParser extends Parser
{

    public int parse(String w)
    {
        int i;
        v=w;
        errorMessage="";
        try
        {
            i=expr();
            if (v.length()>0)
                throw new RuntimeException("Überzählige Zeichen");
        }
        catch (RuntimeException e)
        {
            errorMessage=e.getMessage();
            i=0;
        }
        errorPosition=w.length()-v.length();
        return i;
    }
```

```
    private int expr()
    {
        return rest(number());
    }


    private int rest(int v0)
    {
        if (trymatch("+"))
            return rest(v0+number());
        if (trymatch("-"))
            return rest(v0-number());
        return v0;
    }


    private int number()
    {
        String y="";
        String z=lookahead();
        while (isDigit(z))
        {
            consume(z);
            y+=z;
            z=lookahead();
        }
        if (y.length()>0)
            return Integer.parseInt(y);
        else
            throw new RuntimeException("Ziffer erwartet");
    }

}    // end class SimpleExprParser
```

Die Implementation der Hilfsfunktionen *trymatch*, *lookahead*, *consume* und *isDigit* findet sich in der Basisklasse *Parser*.

Multiplikation, Division und Klammern

Für die Auswertung von arithmetischen Ausdrücken einschließlich Multiplikation, Division sowie Klammern muss die Grammatik um entsprechende Produktionen erweitert werden. Die Produktionen sind so gestaltet, dass bei der Übersetzung die Regel „Punktrechnung vor Strichrechnung“ berücksichtigt wird. Außerdem sind positives und negatives Vorzeichen realisiert.

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{term}\ \mathit{exprrest} \\
\mathit{exprrest} & \rightarrow & +\mathit{term}\ \mathit{exprrest} \mid -\mathit{term}\ \mathit{exprrest} \mid \varepsilon \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{termrest} \\
\mathit{termrest} & \rightarrow & *\ \mathit{factor}\ \mathit{termrest} \mid /\mathit{factor}\ \mathit{termrest} \mid \varepsilon \\
\mathit{factor} & \rightarrow & +\mathit{factor} \mid -\mathit{factor} \mid (\mathit{expr}) \mid \mathit{number}
\end{array}
$$

Zusammenfassung

Die Recursive-Descent-Methode bildet die rekursive Struktur der Grammatik durch entsprechende rekursive Aufrufe von Funktionen nach. Die Funktionen entsprechen hierbei den Nichtterminalzeichen der Grammatik. Der Ableitungsbaum eines zu untersuchenden Wortes wird nicht explizit erzeugt, sondern implizit durch die Folge der Funktionsaufrufe durchlaufen.

Die Recursive-Descent-Methode ist nicht für jede beliebige kontextfreie Grammatik anwendbar. Näheres zur Theorie hierzu liefert die Literatur über Compilerbau, etwa [GE 99]. In der Praxis aber ist sie eine sehr gute und leicht zu handhabende Methode des Übersetzerbaus.

4.4 Übersetzung regulärer Ausdrücke

Ziel ist es, aus einem regulären Ausdruck z in systematischer Weise einen nichtdeterministischen endlichen Automaten $N(z)$ zu konstruieren, der die von z erzeugte reguläre Sprache $L(z)$ erkennt. Hierzu wird zuerst eine Grammatik für reguläre Ausdrücke angegeben. Basierend auf dieser Grammatik wird mithilfe der Recursive-Descent-Methode ein Übersetzer erzeugt, der einen regulären Ausdruck in den entsprechenden Automaten übersetzt. Die Übersetzungsaktionen entsprechen den Schritten, die im Abschnitt Konstruktion eines nichtdeterministischen endlichen Automaten aus einem regulären Ausdruck angegeben sind.

Grammatik

Zur Beschreibung regulärer Ausdrücke verwenden wir eine Grammatik mit folgenden Produktionen:

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{term} \mid \mathit{term}\ []\ \mathit{expr} \\
\mathit{term} & \rightarrow & \mathit{factor} \mid \mathit{factor}\ \mathit{term} \\
\mathit{factor} & \rightarrow & \mathit{atom}\ \mathit{stars} \\
\mathit{atom} & \rightarrow & \mathit{literal} \mid (\mathit{expr}) \\
\mathit{stars} & \rightarrow & {}^{*}\mathit{stars} \mid \varepsilon \\
\mathit{literal} & \rightarrow & \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \%
\end{array}
$$

Das Zeichen | kommt einerseits in regulären Ausdrücken vor, ist also ein Terminalzeichen der Grammatik, andererseits ist es auch ein Metazeichen der Grammatik. In seiner Rolle als Terminalzeichen ist es hier durch [] ersetzt worden. Die Grammatik berücksichtigt die Präzedenzregeln bei der Auswertung regulärer Ausdrücke: * bindet am stärksten, | bindet am schwächsten. Das in den regulären Ausdrücken verwendete Alphabet besteht hier nur aus den Zeichen a, b und c.

Es folgt das Programm.

Klassen *State* und *Nfa*

Zunächst folgt die Klasse *State*, die einen Zustand des Automaten darstellt. Bedingt durch die spezielle Konstruktion des Automaten ist die Klasse *State* sehr einfach aufgebaut.

Der Automat hat genau einen Endzustand. Dieser ist der einzige Zustand, der keinen Folgezustand hat. Alle anderen Zustände des Automaten haben einen oder zwei Folgezustände[4] . Hat ein Zustand zwei Folgezustände, so sind beide Zustandsübergänge mit ε markiert.

Elemente der Klasse sind ein Array t mit zwei Einträgen, das Verweise auf die beiden möglichen Folgezustände enthält, sowie ein String a, der das Zeichen enthält, mit dem der Zustand in den Folgezustand übergeht. Für die Simulation des Automaten wird die boolesche Variable *marked* benötigt.

```
class State
{
    public State[] t;        // Folgezustände
    public String symbol;    // Markierung des Ausgangspfeils,
    // bei zwei Ausgangspfeilen sind beide mit Epsilon markiert
    public boolean marked=false;

    public State(String a)
    {
        t=new State[2];
        symbol=a;
    }

    public State()
    {
        this("§");    // Epsilon
    }
```

[4]In der Implementation gilt dies auch für den Anfangszustand des Automaten für das spezielle Zeichen %

```
    public void mark(boolean b)
    {
        marked=b;
    }

}    // end class State
```

Die folgende Klasse *Nfa* (*nondeterministic finite automaton*) repräsentiert einen nichtdeterministischen endlichen Automaten. Sie enthält einen Konstruktor zur Erzeugung eines Elementarautomaten für ein Alphabetzeichen a. Derselbe Konstruktor wird verwendet, um den Elementarautomaten für das spezielle Zeichen % zu erzeugen. Da das Zeichen % nicht im Alphabet vorkommt, kann der Zustandsübergang mit dem Zeichen % nie ausgeführt werden, so dass der Automat wie gewünscht die leere Sprache akzeptiert.

Die Klasse enthält ferner Methoden zur Parallelschaltung und Hintereinanderschaltung des Automaten mit einem anderen Automaten sowie eine Methode für die Anwendung des Stern-Operators.

```
class Nfa
{
    public State anfz, endz;    // Anfangs- und Endzustand

    public Nfa()
    {
    }


    public Nfa(String a)
    // Elementarautomat für Zeichen a
    {
        anfz=new State(a);
        endz=new State();
        anfz.t[0]=endz;
    }


    public Nfa concat(Nfa y)
    {
        endz.t[0]=y.anfz;
        endz=y.endz;
        return this;
    }
```

```
    public Nfa parallel(Nfa y)
    {
        State s=new State(), t=new State();
        s.t[0]=anfz;
        s.t[1]=y.anfz;
        endz.t[0]=t;
        y.endz.t[0]=t;
        anfz=s;
        endz=t;
        return this;
    }

    public Nfa star()
    {
        State s=new State(), t=new State();
        s.t[0]=anfz;
        s.t[1]=t;
        endz.t[0]=anfz;
        endz.t[1]=t;
        anfz=s;
        endz=t;
        return this;
    }

}    // end class Nfa
```

Klasse *RegExprParser*

Die folgende, von der Klasse *Parser* abgeleitete Klasse *RegExprParser* enthält die Funktionen, die den Produktionen der Grammatik entsprechen, sowie eine Funktion *parse*, die *expr* aufruft und die beim Auftreten von Syntaxfehlern ausgelösten *RuntimeExceptions* abfängt. Die Funktion *parse* liefert den erzeugten Automaten zurück. Ein möglicher Aufruf des Übersetzers ist

```
RegExprParser p=new RegExprParser();
Nfa z=p.parse("a(a|b)*a");
System.out.println(p.errorMessage);
```

Es folgt die Klasse *RegExprParser*.

```
class RegExprParser extends Parser
{
    protected boolean isLetter(String a)
    {
        return isIn(a, "abc%");
    }
```

```
private Nfa literal()
{
    String a=lookahead();
    if (isLetter(a))
    {
        consume(a);
        return new Nfa(a);
    }
    throw new RuntimeException("Buchstabe oder % erwartet");
}


private Nfa atom()
{
    if (trymatch("("))
    {
        Nfa x=expr();
        match(")");
        return x;
    }
    return literal();
}


private Nfa factor()
{
    return stars(atom());
}


private Nfa stars(Nfa x)
{
    if (trymatch("*"))
        return stars(x.star());
    return x;
}


private Nfa term()
{
    Nfa x=factor();
    String a=lookahead();
    if (isLetter(a) || a.equals("("))
        return x.concat(term());
    return x;
}
```

```
    private Nfa expr()
    {
        Nfa x=term();
        if (trymatch("|"))
            return x.parallel(expr());
        return x;
    }

    public Nfa parse(String a)
    {
        Nfa z;
        v=a;
        errorMessage="OK";
        try
        {
            z=expr();
            if (v.length()>0)
                throw new RuntimeException("Überzählige Zeichen");
        }
        catch (RuntimeException e)
        {
            errorMessage=e.getMessage();
            z=null;
        }
        errorPosition=a.length()-v.length();
        return z;
    }

}    // end class RegExprParser
```

Klassen *NfaSimulator* und *Queue*

Die folgende Klasse *NfaSimulator* enthält die Funktionen *epsclosure* und *move*, die benötigt werden, um den Automaten zu simulieren. Die Funktion *epsclosure* realisiert den Schritt „Markiere alle Zustände, die durch Epsilon-Übergänge erreichbar sind“, die Funktion *move* realisiert den Schritt „Markiere alle Folgezustände unter dem Eingabezeichen a“ in der Simulation eines nichtdeterministischen endlichen Automaten. Die markierten Zustände werden im Wechsel in zwei Queues gespeichert und von der jeweils anderen Funktion abgearbeitet. Um den Automaten z beispielsweise mit dem Eingabewort „abba“ zu simulieren, wird der Simulator wie folgt aufgerufen:

```
NfaSimulator t=new NfaSimulator(z);
boolean accepted=t.run("abba");
```

Es folgt die Klasse *NfaSimulator*.

```
class NfaSimulator extends Nfa
{
    private Queue p, q;



    public NfaSimulator(Nfa z)
    {
        anfz=z.anfz;
        endz=z.endz;
    }



    public boolean run(String s)
    // simuliert den Automaten mit Eingabewort s
    {
        String a;
        boolean accepted=false;
        p=new Queue();
        q=new Queue();
        anfz.mark(true);
        q.add(anfz);
        s+="$";        // Begrenzer-Zeichen an String s anhängen

        for(int i=0; i<s.length(); i++)
        {
            accepted=epsclosure();
            a=s.substring(i, i+1);
            move(a);
        }
        return accepted;
    }



    private boolean epsclosure()
    {
        State s, t;
        boolean accepted=false;
        while (!q.isEmpty())
        {
            s=(State)q.remove();
            p.add(s);
            accepted=accepted || s==endz;
```

```
            if (s.symbol.equals("§"))  // Epsilon
                for (int i=0; i<2; i++)
                {
                    t=s.t[i];
                    if (t!=null)
                        if (!t.marked)
                        {
                            t.mark(true);
                            q.add(t);
                        }
                }
        }
        return accepted;
    }


    private void move(String a)
    {
        State s;
        while (!p.isEmpty())
        {
            s=(State)p.remove();
            s.mark(false);
            if (s.symbol.equals(a))
            {
                s.t[0].mark(true);
                q.add(s.t[0]);
            }
        }
    }

}    // end class NfaSimulator
```

Zum Schluss folgt eine mögliche Implementation der von *NfaSimulator* verwendeten Queue-Datenstruktur als verkettete Liste. Die Listenelemente sind vom hier zunächst definierten Typ *Item*.

```
class Item
{
    public Object obj;
    public Item nxt;

}    // end class Item
```

```
class Queue
{
    private Item top;
    private Item bot;


    public boolean isEmpty()
    {
        return top==null;
    }


    public void add(Object p)
    {
        Item b=new Item();
        if (isEmpty())
            top=b;
        else
            bot.nxt=b;
        bot=b;
        bot.obj=p;
    }


    public Object remove()
    {
        if (isEmpty())
            throw new RuntimeException("Queue leer");
        Item b=top;
        top=b.nxt;
        return b.obj;
    }

}    // end class Queue
```

Zusammenfassung

Die prinzipielle Struktur regulärer Ausdrücke ist durch die Grammatik für reguläre Ausdrücke vorgegeben. Aus dieser Grammatik haben wir mit der Recursive-Descent-Methode einen Übersetzer für reguläre Ausdrücke konstruiert. Ziel der Übersetzung eines bestimmten regulären Ausdrucks ist ein zugehöriger nichtdeterministischer endlicher Automat, der die von dem regulären Ausdruck erzeugte Sprache erkennt. Durch Simulation dieses Automaten können Eingabewörter daraufhin geprüft werden, ob sie von dem regulären Ausdruck erzeugt werden oder nicht.

4.5 String-Matching-Automaten

Wir nehmen das Thema Textsuche noch einmal (kurz) auf. Es zeigt sich, dass sich das Textsuch-Problem (engl.: *string matching problem*) auf die Erkennung einer regulären Sprache mit einem nichtdeterministischen endlichen Automaten zurückführen lässt. Ein entsprechend konstruierter Automat wird als *String-Matching-Automat* bezeichnet.

Konstruktion von String-Matching-Automaten

Sei $A = \{a_0, ..., a_{k-1}\}$ ein Alphabet und $p = p_0 \dots p_{m-1}$ ein Muster sowie ferner t ein Text über A. Die Aufgabe eines Textsuchverfahrens besteht darin, alle Vorkommen des Musters p im Text t zu suchen.

Offenbar erhält man einen solches Verfahren, indem man für den regulären Ausdruck A^*p einen zugehörigen nichtdeterministischen endlichen Automaten konstruiert und diesen mit dem Text t als Eingabe simuliert.[5] Es ist lediglich noch eine Ausgabefunktion hinzuzufügen, die jedesmal ein Vorkommen des Musters im Text meldet, wenn der Automat einen Endzustand erreicht.

Statt der schematischen Methode zur Konstruktion eines Automaten aus einem regulären Ausdruck wird folgende vereinfachte Konstruktion angewendet:

Der Anfangszustand geht mit allen Zeichen aus A in sich selbst über, außerdem geht vom Anfangszustand eine Kette von Zustandsübergängen mit den Zeichen p_0, p_1, ..., p_{m-1} aus. Der letzte Zustand dieser Kette ist der Endzustand.

Wir bezeichnen einen solchen vereinfacht konstruierten nichtdeterministischen endlichen Automaten als *String-Matching-Automaten*.

Beispiel: Das folgende Bild 4.14 zeigt den vereinfacht konstruierten nichtdeterministischen endlichen Automaten für das Muster $p = \text{aaba}$.

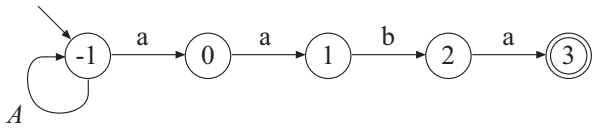


Bild 4.14: *String-Matching-Automat für $p = aaba$*

Simulation von String-Matching-Automaten

Die Simulation eines nichtdeterministischen endlichen Automaten ist im allgemeinen nicht sehr effizient, denn es müssen bei jedem eingelesenen Zeichen alle markierten

[5] In dem regulären Ausdruck steht A für $(a_0 \mid a_1 \mid \dots \mid a_{k-1})$.

Zustände behandelt werden. Wenn etwa das Muster a^m und der Eingabetext a^n ist, sind nach kurzer Zeit m Zustände markiert, d.h. es sind bei jedem gelesenen Zeichen m Zustandsübergänge zu behandeln. Somit ergibt sich eine Komplexität der Simulation von $\Theta(n \cdot m)$, also keine Verbesserung gegenüber dem naiven Algorithmus.

Ein effizienter Algorithmus ergibt sich durch Konstruktion eines zu dem String-Matching-Automaten äquivalenten deterministischen Automaten. Dieser verarbeitet jedes gelesene Eingabezeichen in konstanter Zeit, so dass sich eine Zeitkomplexität von $\Theta(n)$ ergibt.

Der Speicherbedarf eines entsprechenden deterministischen Automaten beträgt jedoch $\Theta(m \cdot k)$. Denn die Anzahl der Zustände des Automaten ist etwa gleich m, und die Tabelle für die möglichen Zustandsübergänge hat in jedem Zustand k Einträge (m Länge des Musters, k Größe des Alphabets). Entsprechend hohe Zeitkomplexität hat auch das Verfahren zur Konstruktion des deterministischen Automaten.

Da es sich bei String-Matching-Automaten um sehr spezielle nichtdeterministische endliche Automaten handelt, stellt sich die Frage, ob sich ein solcher Automat nicht auf andere Art effizienter simulieren lässt als ein allgemeiner nichtdeterministischer endlicher Automat. Tatsächlich ist dies der Fall; sowohl der Knuth-Morris-Pratt-Algorithmus als auch der Shift-And-Algorithmus basieren auf einer geschickten Simulation eines String-Matching-Automaten.

Shift-And-Algorithmus

Der Shift-And-Algorithmus stellt eine Simulation eines String-Matching-Automaten dar. Die Funktionsweise ist die folgende:

Die bei der normalen Simulation eines nichtdeterministischen endlichen Automaten folgende Zustandsmarkierung bei Eingabe eines Zeichens $a \in A$ ergibt sich, indem in einem ersten Schritt alle Markierungen um 1 nach rechts geschoben werden und der Anfangszustand erneut markiert wird. Korrekt markiert sind jetzt allerdings außer dem Anfangszustand nur diejenigen Zustände, zu denen tatsächlich ein mit a bezeichneter Pfeil hinführt. Im zweiten Schritt werden daher alle anderen Markierungen gelöscht.

Implementierung mit Bitvektoren

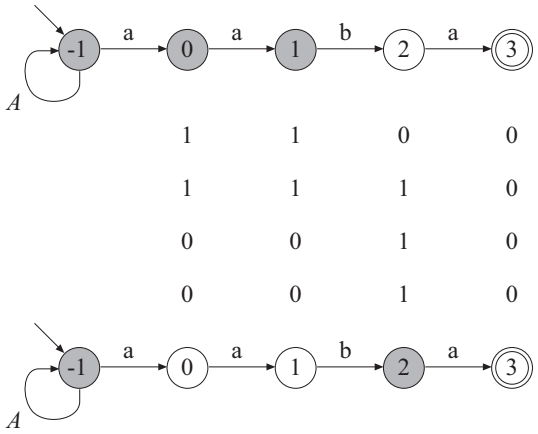
Die Zustandsmarkierung des Automaten lässt sich in einem Bitvektor der Länge m darstellen. Hierbei ist Bit i gleich 1, wenn Zustand i markiert ist, und sonst 0. Der Anfangszustand ist immer markiert, er braucht daher nicht in dem Bitvektor repräsentiert zu sein. Daher wird er hier mit -1 nummeriert, so dass die Nummerierung der folgenden Zustände zur Indizierung des Bitvektors passt.

In Schritt 1 wird der Bitvektor um eine Position nach rechts geschoben, wobei an Position 0 eine 1 nachgezogen wird (entsprechend der Markierung des Zustands -1).

In Schritt 2 erfolgt das Ausblenden der nunmehr falsch markierten Bitstellen, indem ein logisches Und mit einem Bitvektor durchgeführt wird, der an den richtigen Stellen

Einsen trägt. Dieser Bitvektor ist der *charakteristische Vektor* für das Eingabezeichen a; er trägt eine 1 an Position i, wenn $p_i = a$ ist.

Beispiel: Das folgende Bild 4.15 zeigt die Markierung des Automaten nach dem Einlesen des Wortes aa. Darunter ist der zugehörige Bitvektor angegeben. Die Markierung des Automaten bei Einlesen des Zeichens b ergibt sich durch Rechtsschieben des Bitvektors und Nachziehen einer 1 sowie anschließender Und-Verknüpfung mit dem charakteristischen Bitvektor von b.



1	1	0	0	ursprüngliche Markierung
1	1	1	0	geshiftete Markierung, 1 nachgezogen
0	0	1	0	& charakteristischer Bitvektor für b
0	0	1	0	= neue Markierung

Bild 4.15: *Neue Zustandsmarkierungen nach Einlesen des Zeichens b*

Wenn die verwendeten Bitvektoren in ein Maschinenwort passen, lassen sich die Schiebe- und die Und-Operation in jeweils einem Schritt durchführen. Dies ist pro gelesenem Textzeichen zu rechnen, so dass sich für die Suchphase insgesamt eine Komplexität von $\Theta(n)$ ergibt.

In einem Vorlauf muss für jedes im Muster vorkommende Zeichen der charakteristische Vektor berechnet werden; dies benötigt Zeit $\Theta(m)$. Die charakteristischen Vektoren aller anderen Zeichen des Alphabets sind 0.

Knuth-Morris-Pratt-Algorithmus

Auch der Knuth-Morris-Pratt-Algorithmus stellt eine Simulation eines String-Matching-Automaten dar. Der Ansatz ist der folgende:

Es wird nur die jeweils am weitesten nach rechts vorgerückte Markierung in der normalen Simulation eines nichtdeterministischen endlichen Automaten betrachtet.

Interessanterweise sind durch diese Markierung alle anderen Markierungen eindeutig bestimmt.

Die Positionen der jeweils anderen Markierungen lassen sich im voraus aus der Struktur des Musters berechnen. Denn wenn ein Zustand markiert ist, weiß man, welches die zuletzt eingelesenen Zeichen gewesen sein müssen, die zur Markierung dieses Zustands geführt haben. Dann weiß man auch, welche der vorhergehenden Zustände markiert sind.

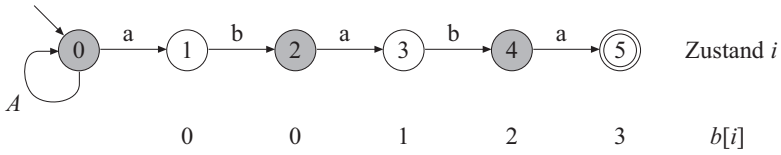
In folgendem Beispiel etwa ist der Zustand 4 markiert. Somit sind die zuletzt eingelesenen Zeichen abab gewesen. D.h. die beiden letzten eingelesenen Zeichen sind ab gewesen, also ist auch Zustand 2 markiert. Zustand 0 ist immer markiert.

Die Positionen der Markierungen werden in einem Array b gehalten, das die folgende Information enthält:

Wenn Zustand i markiert ist, dann ist $b[i]$ der nach links gesehen nächste markierte Zustand.

Beispiel: Bild 4.16 zeigt den nichtdeterministischen endlichen Automaten für das Muster $p =$ ababa. Unter jedem Zustand i ist der Wert $b[i]$ angegeben.

Zustand 4 trägt die am weitesten nach rechts vorgerückte Markierung. Der nächste markierte Zustand ist $b[4] = 2$, der darauf folgende ist $b[2] = 0$.



$b[i]$ = nächster markierter Zustand (nach links gesehen), wenn Zustand i markiert ist

Bild 4.16: *Werte des Arrays b*

Die Simulation geschieht nun, indem jeweils geprüft wird, ob von dem am weitesten rechts befindlichen markierten Zustand i aus ein Zustandsübergang mit dem gelesenen Zeichen a möglich ist. Wenn dies der Fall ist, wird dieser Zustandsübergang vollzogen und das nächste Zeichen gelesen. Wenn nicht, wird die Markierung gelöscht und stattdessen Zustand $b[i]$ markiert. Von diesem Zustand aus wird ein neuer Versuch unternommen, mit dem Zeichen a einen Zustandsübergang zu machen usw.

Spätestens im Anfangszustand gelingt der Zustandsübergang, da vom Anfangszustand aus unter jedem Eingabezeichen ein Zustandsübergang möglich ist.

Suchen von mehreren Mustern

Eine Verallgemeinerung des Textsuchproblems besteht darin, gleichzeitig nach mehreren Mustern zu suchen. Im Algorithmus von Aho und Corasick [AC 75] wird dieses Problem ebenfalls durch Simulation eines String-Matching-Automaten gelöst. Anhand des in [AC 75] angegebenen Beispiels wird das Verfahren im Folgenden skizziert.

Sei $P = \{\text{he}, \text{she}, \text{his}, \text{hers}\}$ die Menge der Muster. Der zugehörige String-Matching-Automat ist in Bild 4.17 abgebildet. Nach Eingabe der Zeichenfolge sh ergibt die Simulation des Automaten die dargestellte Zustandsmarkierung.

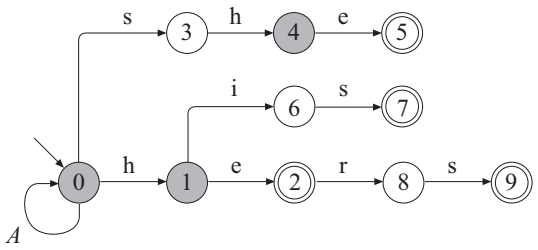


Bild 4.17: *String-Matching-Automat für mehrere Muster*

Genau wie beim Knuth-Morris-Pratt-Algorithmus wird die am weitesten nach rechts vorgerückte Markierung betrachtet. Wird diese Markierung gelöscht, weil von diesem Zustand aus mit dem eingelesenen Zeichen kein Zustandsübergang möglich ist, so wird versucht, mit der nächsten Markierung den entsprechenden Zustandsübergang durchzuführen usw. Spätestens beim Anfangszustand gelingt der Zustandsübergang, da vom Anfangszustand aus unter jedem Eingabezeichen ein Zustandsübergang möglich ist.

Im Beispiel von Bild 4.17 kann die Markierung von Zustand 4 unter dem Eingabezeichen i nicht vorrücken und wird daher gelöscht. Nun wird versucht, mit dem nächsten markierten Zustand, hier 1, unter dem Eingabezeichen i einen Zustandsübergang zu machen. Dies führt hier zum Erfolg, die Markierung rückt auf Zustand 6 vor.

Auch hier bestimmt die am weitesten rechts befindliche Markierung eindeutig alle anderen Markierungen. Befindet sich wie im Beispiel die am weitesten nach rechts vorgerückte Markierung auf Zustand 4, so sind die letzten eingelesenen Zeichen sh gewesen. Somit ist auch Zustand 1 markiert. Zustand 0 ist immer markiert.

Es kann also ein Array b im voraus berechnet werden, das die folgende Information enthält: Wenn Zustand i markiert ist, so ist $b[i]$ der nächste markierte Zustand.

Im Beispiel ist $b[4] = 1$ und $b[1] = 0$; $b[0]$ wird auf -1 gesetzt.

Die Struktur des Automaten entspricht einem sogenannten *Trie*. Ein Trie ist ein Baum

mit Kantenmarkierung, dessen an der Wurzel beginnenden Pfade mit den Präfixen einer Menge von Wörtern markiert sind, wobei keine verschiedenen Pfade die gleiche Markierung tragen.

4.6 Aufgaben

Aufgabe 1: Schreiben Sie einen Recursive-Descent-Parser, der logische Formeln der Art

$$\neg(a \vee b) \wedge \neg\neg c \quad \Leftrightarrow \quad 0 \;\Rightarrow\; a \wedge \neg c$$

auf korrekten syntaktischen Aufbau überprüft.

Für die logischen Symbole sollen folgende Ersatzsymbole verwendet werden:

$\neg$	:	`!`
$\wedge$	:	`*`
$\vee$	:	`+`
$\Rightarrow$	:	`=>`
$\Leftrightarrow$	:	`<=>`

Diese Tabelle gibt gleichzeitig die Präzedenz der logischen Operatoren wieder: $\neg$ bindet am stärksten, $\Leftrightarrow$ bindet am schwächsten.

Entwerfen Sie eine Grammatik für logische Formeln, die diese Präzedenz der Operatoren berücksichtigt. Orientieren Sie sich dabei an der Grammatik für Integer-Ausdrücke aus Abschnitt 4.3.

Schreiben Sie einen Recursive-Descent-Übersetzer, der eine logische Formel der obigen Art einliest und in ihren Wert übersetzt, wobei eine Belegung der Variablen a, b und c mit den Wahrheitswerten *true* oder *false* zugrunde gelegt ist.