Grundlagen der Informatik

Vorlesungsskript

Elemente der Theoretischen Informatik

Matrikel 05WM

Wintersemester 2005/06

Uwe Petermann

FB IMN, HTWK Leipzig
G.-Freytag-Str. 42, Leipzig
Netz: uwe@imn.th-leipzig.de

6. Februar 2006

7 Elemente der theoretischen Informatik

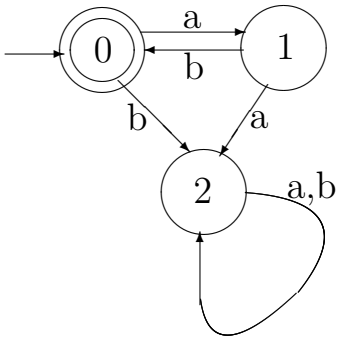
Die theoretische Informatik befaßt sich mit prinzipiellen Fragestellungen:

> Was ist mit Computern machbar?
>
> Präziser: Welche Probleme sind mit Computern und geeigneten Programmen lösbar?
>
> Detaillierter: Welche (mathematischen) Modelle werden gebraucht, um folgende Fragen zu beantworten.
>
> > Was ist ein Computer?
> >
> > Was ist ein Programm?
> >
> > Was ist ein Problem?
> >
> > Was heißt es, ein Problem zu lösen?

Um auf die genannten Fragen Antworten zu geben,
sind mathematische Modelle für
Computer, Programme und Probleme sowie Algorithmen zu entwickeln.

- Probleme können durch Sprachen beschrieben werden.
  - Lösen, im Sinne von Entscheiden, heißt dann,
    die Zugehörigkeit von Wörtern zu Sprachen zu entscheiden.
  - Lösen, im Sinne von Konstruieren, heißt dann,
    für Ausgangswerte (Wörter über einem Alphabet) Ergebniswerte
    (Wörter über einem Alphabet) zu konstruieren.
- Hilfsmittel zur Beschreibung von Sprachen :
  - Automaten (vgl. Definitionen 7.2 und 7.10)
  - Grammatiken (vgl. Definition 7.12)
- Welche Probleme sind überhaupt lösbar?
- Welche Probleme sind praktisch lösbar?

Beispiel: endlicher Automat $A_1 = (Q, \Sigma, q_0, \delta, F)$:



Zustandsmenge: $Q = \{0, 1, 2\}$
Eingabealphabet: $\Sigma = \{a, b\}$
Startzustand: $q_0 = 0$

Zustandsüberführungsfunktion:

δ :	a	b
0	1	2
1	2	0
2	2	2

Endzustandsmenge: $F = \{0\}$

von A akzeptierte Wörter: ϵ, ab, abab, ababab, ...
von A **nicht** akzeptierte Wörter: a, aba, aab, abbb, ...
es gilt: $\delta^*(0, abab) = 0$, $\delta^*(0, a) = 1$, $\delta^*(1, ab) = 2$, ...

Abbildung 31: Graphische Darstellung eines endliches Automaten

---

7.1 Automaten, Sprachen, Grammatiken

7.1.1 Automaten und reguläre Sprachen

Zur Motivation und zu Anwendungen
von endlichen Automaten:

- Automaten als Akzeptoren von formalen Sprachen (vgl. Abb. 31)
  akzeptierte Sprache:
  von außen sichtbares Verhalten eines Systems
  Automat: interne Realisierung des Systems
  - Beispiel:
    Man kann mit Hilfe des Automaten in Abbildung 31 entscheiden, ob ein Wort bestehend aus den Buchstaben A und B eine Verkettung von Wörtern der Form AB ist:
    Mit der Interpretation A - Anmelden, B - Beenden
    ergibt dies ein einfaches Protokoll zur Bedienung eines (Banking-) Systems

  Mit dieser Anwendung wird der in Definition 7.2 eingeführte *deterministische endliche Automat* (DEA) motiviert.

Abbildung 32: Schaltwerk als eine Motivation für endliche Automaten

---

- Beschreibung von Schaltwerken: vgl. [11] S. 86 ff.

  Das Schaltwerk in Abbildung 32 berechnet

  für Eingabewerte $x_1, \ldots, x_l$ und Steuersignale $s_1, \ldots, s_k$

  Ausgaben $y_1, \ldots, y_m$ und

  neue Steuersignale $s'_1, \ldots, s'_k$.

  Letztere werden über Flip-Flop-Schaltungen im nächsten Takt weiter verarbeitet.

  Das Wertetupel $(s_1, \ldots, s_k)$ wird als innerer Zustand des Schaltwerkes aufgefaßt.

  Mit dieser Anwendung ist der in Definition 7.10 eingeführte Begriff des *Mealy-Automaten* motiviert.

  Für $m = 0$ erhält man die in 7.1.1 für die Definition von Sprachen eingeführten Automaten ohne Ausgabe.

Definition 7.1 Für eine nichtleere Menge Σ, dem *Alphabet*, heißt jede (endliche) Folge von Zeichen aus Σ *(endliches) Wort über* Σ.

Die Menge aller Wörter über Σ wird mit Σ^* notiert.

Eine Teilmenge von Σ^* heißt *Sprache über* Σ. □

Sprachen fassen wir als Mengen von Wörtern auf.

Wörter können beliebige Folgen von Elementen aus einer gegebenen, nichtleeren Menge, dem *Alphabet* sein.

Meist betrachtet man nur endliche Wörter, ohne dies besonders hervorzuheben.

Soweit auch unendliche Wörter betrachtet werden sollen, wird dies besonders hervorgehoben.

Abkürzungen für eine kompakte Notation:

Wörter werden als Folgen ihrer Zeichen ohne Trennzeichen notiert. (*abca* ist Wort über Alphabet $\{a, b, c\}$).

Die leere Zeichenreihe schreiben wir als ε.

Gegeben sei ein Alphabet Σ, ein Zeichen $a \in \Sigma$ und eine natürliche Zahl $n \in \mathbf{N}$.

Dann definieren wir a^0 als die leere Zeichenreihe und a^n die Zeichenreihe $\underbrace{a \ldots a}_{n\text{-fach}}$.

Mit Σ^+ wird die Menge der nichtleeren Wörter bezeichnet, d.h. $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Für $n \in \mathbf{N}$ bezeichnet Σ^n die Menge der Wörter über Σ der Länge n.

Beispiel 7.1 Sei $\Sigma = \{a, b, c, d, u, k\}$. Folgende Mengen dienen als Beispiele für Sprachen. Sie sind nach aufsteigender Komplexität ihrer Beschreibung geordnet.

(1) $L_1 = \{ab, ba\}$

(2) $L_2 = \{a(k^m u^n)^p b \mid m, n, p \in \mathbf{N}\}$

(3) $L_3 = \left\{ c\, x_1 c\, x_2 \ldots c\, x_n d^n \;\middle|\; \begin{array}{l} n \in \mathbf{N}, \\ 0 < n, \\ x_1, x_2, \ldots, x_n \in \{a\}^+ \end{array} \right\}$

(4) $L_4 = \left\{ x_1 a\, y\, x_2 b\, x_3 y\, x_4 \;\middle|\; \begin{array}{l} y \in \Sigma^+, \\ x_1 x_1, x_2, x_3, x_4, \in \Sigma^*, \\ ay \not\sqsubseteq x_1 x_2 \end{array} \right\}$

Weitere Notationen:

Zeichen eines Alphabets	a, b, c, ...
Wörter über einem Alphabet	x, y, z, ...
Sprachen	A, B, C, ...

Weitere Operationen und Relationen für Wörter:

Verkettung der Wörter $x = a_1 \dots a_n$ und $y = b_1 \dots b_m$ ist das Wort $xy = a_1 \dots a_n b_1 \dots b_m$.

Teilwort. x ist Teilwort des Wortes y, wenn es Wörter z_1 und z_2 gibt mit $y = z_1 x z_2$. In Zeichen $x \sqsubseteq y$.

Präfix. x ist Präfix des Wortes y, wenn es ein Wort z gibt mit $y = xz$.

Suffix. x ist Suffix des Wortes y, wenn es ein Wort z gibt mit $y = zx$.

Übung: Zeigen Sie: $x \sqsubseteq y$ genau dann, wenn es z_1 und z_2 gibt mit: x ist Suffix von z_1 und z_1 ist Praefix von y sowie
x ist Praefix ist z_2 und z_2 ist Suffix von y.

Definition 7.2

(1) Ein Tupel $A = (Q, \Sigma, q_0, \delta, F)$ heißt *deterministischer endlicher Automat (DEA)*, falls

   (a) Q und Σ endliche, paarweise disjunkte Mengen (genannt: Zustandsmenge und Eingabealphabet) sind,

   (b) $q_0 \in Q$ heißt *Startzustand*.

   (c) $F \subseteq Q$ für die Endzustandsmenge F gilt und

   (d) δ eine Abbildung $\delta : Q \times \Sigma \to Q$ ist. Sie heißt *Zustandsüberführungsfunktion*.

(2) Die *erweiterte Zustandsüberführungsfunktion* $\delta^* : Q \times \Sigma^* \to Q$ ist wie folgt definiert:

   (a) $\delta^*(q, \varepsilon) = q$ und

   (b) $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$.

(3) Die Menge $\{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ heißt von A *akzeptierte Sprache* und wird mit $L(A)$ notiert. A wird auch *Akzeptor* für $L(A)$ genannt.

□

Beispiel 7.2 Modifizieren Sie den Automaten aus der Einleitung so, daß er alle Wörter akzeptiert, bei denen zwischen einem a und einem b jeweils beliebige endliche Folgen von Symbolen u und k vorkommen können.

Beispiel 7.3 Definieren Sie einen DEA, der die rechnerinterne Darstellung von negativen Zahlen des Typs `byte` akzeptiert.

Beispiel 7.4 Definieren Sie einen DEA, der die Bezeichner in einer Programmiersprache akzeptiert.

Beispiel 7.5 vgl. 4.1.1, 4.1.2, 4.1.3 in [11]
Minimalität bzgl. der Anzahl der Zustände wird als Gütemaß für Automaten angenommen.

Je mehr Zustände ein Automat hat, desto mehr Ressourcen (sei es in Hardware (Material) oder in Software (Speicher)) werden für seine Realisierung verbraucht.

Man ist deshalb an einem Automaten interessiert, der eine gewünschte Funktionalität erbringt, d.h. eine Sprache akzeptiert, aber möglichst wenige Zustände hat.

Es gilt, zwei Ursachen für zu große Zustandsmengen auszuschalten:

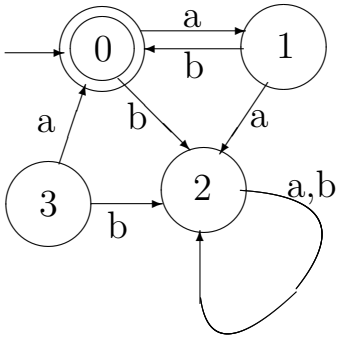
(1) *Zustände, die niemals aktiv werden, also nichts zur Arbeit des Automaten beitragen* (vgl. Def. 7.3 und Abb. 31).

Diese Quelle von Ineffizienz wird mit der Konstruktion nach Satz 7.1 beseitigt.

(2) *Paare von Zuständen, die sich in ihrer Wirkung nicht unterscheiden.*

Definition 7.4 definiert die Äquivalenz von Zuständen.
Ein effizienter Algorithmus findet alle Zustandsäquivalenzen.
Durch Zusammenfassen äquivalenter Zustände kommt man zum Äquivalenzklassenautomat (vgl. Definition 7.5 und das Beispiel in Abb. 33).

endlicher Automat $A_2 = (Q, \Sigma, q_0, \delta, F)$:



Zustandsmenge: $Q = \{0, 1, 2, 3\}$
Eingabealphabet: $\Sigma = \{a, b\}$
Startzustand: $q_0 = 0$
Zustandsüberführungsfunktion:

δ :	a	b
0	1	2
1	2	0
2	2	2
3	0	2

Endzustandsmenge: $F = \{0\}$

Genau ein *unerreichbarer* Zustand: 3

Abbildung 33: Überflüssige Zustände eines endliches Automaten

---

Definition 7.3 ([11], Df. 4.2.1)
Ein Zustand eines DEA, der vom Anfangszustand aus nicht errreicht wird, heißt *überflüssig.* □

Satz 7.1 *([11], Satz. 4.2.2)*

Die Menge der überflüssigen Zustände eines DEA kann in Zeit $O(|Q||\Sigma|)$ berechnet werden.

Beweis. Idee: Tiefe-zuerst-Suche im Graphen, der den DEA darstellt. **q.e.d.**

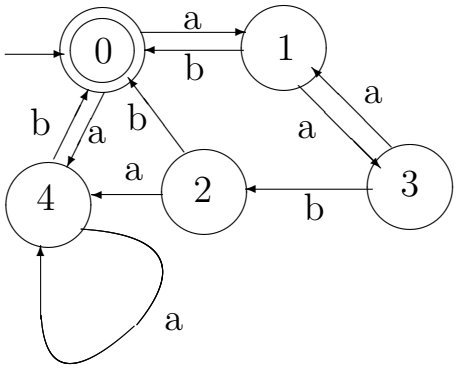
Definition 7.4 ([11], Df. 4.2.3)
Zustände p und q eines DEA heißen *äquivalent* gdw.

$$\forall w \in \Sigma^*.\ \delta^*(p, w) \in\ F \Leftrightarrow \delta^*(q, w) \in\ F$$

Notation für Zustandsäquivalenz: $p \equiv\ q$.

Die Relation $\equiv$ heißt auch Nerode-Relation. □

endlicher Automat $A_3 = (Q, \Sigma, q_0, \delta, F)$:



Zustandsmenge: $Q = \{0, 1, 2, 3, 4\}$
Eingabealphabet: $\Sigma = \{a, b\}$
Startzustand: $q_0 = 0$
Zustandsüberführungsfunktion:

δ :	a	b
0	4	1
1	3	0
2	4	0
3	1	2
4	4	0

Endzustandsmenge: $F = \{0\}$

Zwei *äquivalente* Zustände: 2 und 4

Abbildung 34: Endlicher Automat mit zwei äquivalenten Zuständen

Für alle nicht geordneten Paare von zwei verschiedenen Zuständen p und q werden Wörter gesucht, für $\delta^*(p, w) \in F$ und $\delta^*(q, w) \notin F$ gilt. Solche Wörter bezeugen die Inäquivalenz der jeweils betrachten Zustände.

Nachfolgende Tabelle protokolliert eine Prüfung der Zustandspaare für Automat A_3 auf Inäquivalenz. Für Paare verschiedener Zustände, die den Endzustand enthalten, wird die Inäquivalenz durch das leere Wort bezeugt. Das Wort b bezeugt die Inäquivalenz der Zustände in $\{1, 3\}$, $\{2, 3\}$ und $\{3, 4\}$. Mit dem Zeugen b für die Inäquivalenz von $3, 4$ ergibt sich der Zeuge ab für die Inäquivalenz der Zustände in $\{1, 2\}$ bzw. in $\{1, 4\}$. Die Zustände 2 und 4 sind äquivalent, da $\delta(1, a) = \delta(1, a)$ und $\delta(1, b) = \delta(1, b)$.

Zeuge	Zustandspaar		Symbole a		Symbole b		Wartelisten
ϵ	0	1					
ϵ	0	2					
ϵ	0	3					
ϵ	0	4					
ab	1	2	3	4			
b	1	3			0	2	
ab	1	4	3	4			
b	2	3			0	2	
-	2	4	4	4	0	0	
b	3	4			2	0	$\{1, 2\}, \{1, 4\}$

Definition 7.5 (vgl. [11], Df. 4.2.3)

Äquivalenzklassenautomat: Für die in Definition 7.4 eingeführte Äquivalenzrelation $\equiv$ auf der Menge der Zustände eines DEA $A = (Q, \Sigma, q_0, \delta, F)$ sei $A' = (Q', \Sigma', q_0', \delta', F')$

wie folgt definiert:

(1) $Q' = [Q]/_{\equiv}$

(2) $\Sigma' = \Sigma$

(3) $q_0' = [q_0]/_{\equiv}$

(4) $F' = \{[q]_{\equiv} \mid q \in F\}$

(5) $\delta'([q]_{\equiv}, x) = [\delta(q, x)]_{\equiv}$ □

Satz 7.2 *([11], Df. 4.2.5)*

Äquivalenzklassenautomat A' ist wohldefiniert und akzeptiert gleiche Sprache wie A.

Wir wenden uns nun der Frage zu, wie die Klasse von Sprachen, welche von DEA akzeptiert werden, charakterisiert werden kann.

Es geht um ein vom Automatenbegriff unabhängige Charakterisierung.
Es zeigt sich,
daß dies gerade die Definition 7.8 definierten regulären Sprachen sind.

Diese Klasse enthält alle endlichen Sprachen und ist bezüglich Vereinigung, Verkettung und Iteration abgeschlossen.

Satz 7.3 *([11], Satz 4.6.5)*

Für DEA $A_1 = (Q_1, \Sigma, q_1, \delta_1, F_1)$ und $A_2 = (Q_2, \Sigma, q_2, \delta_2, F_2)$ kann in der Zeit $O(|Q_1||Q_2||\Sigma|)$ ein DEA konstruiert werden, der die Vereinigung $L(A_1) \cup L(A_2)$ akzeptiert.

Beweis.

Konstruktionsidee für den Automaten $A = (Q, \Sigma, q_0, \delta, F)$:

$Q = Q_1 \times Q_2$,
$q_0 = (q_1, q_2)$,
$\forall (q, q') \in Q, a \in \Sigma.\ \delta((q, q'), a) = (\delta_1(q, a), \delta_2(q', a))$
$F = Q_1 \times F_2 \cup F_1 \times Q_2$. **q.e.d.**

Anmerkung:

Der $L(A_1) \cup L(A_2)$ akzeptierende Automat
kann unter Vermeidung überflüssiger Zustände konstruiert werden.
Bei der Konstruktion wird in einer Matrix für $Q_1 \times Q_2$ eingetragen, welche Zustände erreichbar sind.

Für Sprachen werden nun zur Vorbereitung von Definition 7.8 folgende Operationen erklärt.

Definition 7.6

Die *Verkettung* (auch: die Produktsprache, die Konkatenation) der Sprachen L und L' ist die Sprache $LL' = \{xy \mid x \in L, y \in L'\}$. □

Definition 7.7

Der *Kleenesche Abschluß* (kurz: der Abschluß) der Sprache L ist die Sprache $L^* = \cup_{i \in \mathbf{N}} L^i$. Dabei gelten $L^0 = \{\varepsilon\}$ und für alle $i \in \mathbf{N} : L^{i+1} = L(L^i)$.

□

Oft wird auch die Abkürzung L^+ für LL^* benutzt (genannt positiver Abschluß).

Definition 7.8 Eine Sprache, die durch endlich viele Anwendungen der Operationen Verkettung, Vereinigung und Iteration aus endlichen Sprachen über einem Alphabet Σ gebildet werden kann, heißt *reguläre Sprache über* Σ. □

Beispiel 7.6

(1) Die Sprachen 1 und 2 in Beispiel 7.1 sind regulär, die Sprachen 3 und 4 in jenem Beispiel sind es nicht.

(2) Über dem Alphabet $\{a, \ldots, z, 0, \ldots, 9\}$ beschreibt die Sprache

$$L_5 = \{a, \ldots, z\}\,\{a, \ldots, z, 0, \ldots, 9\}^*$$

die Menge der Namen (Bezeichner) in (Standard-)Pascal.

(3) Über dem Alphabet $\{-, +, 0, 1\}$ beschreibt die Sprache

$$L_6 = \{-1, +1, 1\}\{0, 1\}^* \cup \{0\}$$

alle Darstellungen ganzer Zahlen im Dualsystem.

(4) Über dem Alphabet $\{-, +, ., 0, 1, E\}$ beschreibt die Sprache

$$L_7 = \{-1, +1, 1\}\{.\}\{0, 1\}^*\{E\}\{-1, +1\}\{0, 1\}^+$$

alle Darstellungen reeller Zahlen im Dualsystem.

Es wird konstruktiv gezeigt, daß die Klasse der durch DEA akzeptierten Sprachen abgeschlossen bezüglicher der Operationen Vereinigung, Verkettung und Iteration ist (vgl. Sätze 7.3, 7.4 und 7.5).

Da für Sprachen, die nur aus einem Element bestehen, sehr leicht ein diese Sprache akzeptierender Automat angegeben werden kann,

ist mit der Angabe der entsprechenden Konstruktionen gezeigt, daß alle regulären Sprachen durch Automaten akzeptiert werden können.

Der umgekehrte Schluß erfordert eine tiefere Einsicht in die Struktur von regulären Sprachen.

Man kann zeigen, daß für jede reguläre Sprache eine Äquivalenzrelation definiert werden kann, die endlich viele Äquivalenzklassen definiert.

Diese Klassen werden als Zustände eines diese Sprache definierenden Automaten benutzt (vgl. [11] S. 96-98).

Können für gegebene Sprachen A_1 und A_2 Automaten A_3 und A_4 konstruiert werden, die $L(A_1)L(A_2)$ bzw. $L(A_1)^*$ akzeptieren?

Idee: „Hintereinanderschalten“ der Automaten A_1 und A_2.

Die Idee ist nachfolgend für die Automaten A_4 und A_5 illustriert. Der akzeptierende Zustand von A_4 erhält einen zusätzlichen Übergang zum Anfangszustand von A_5.

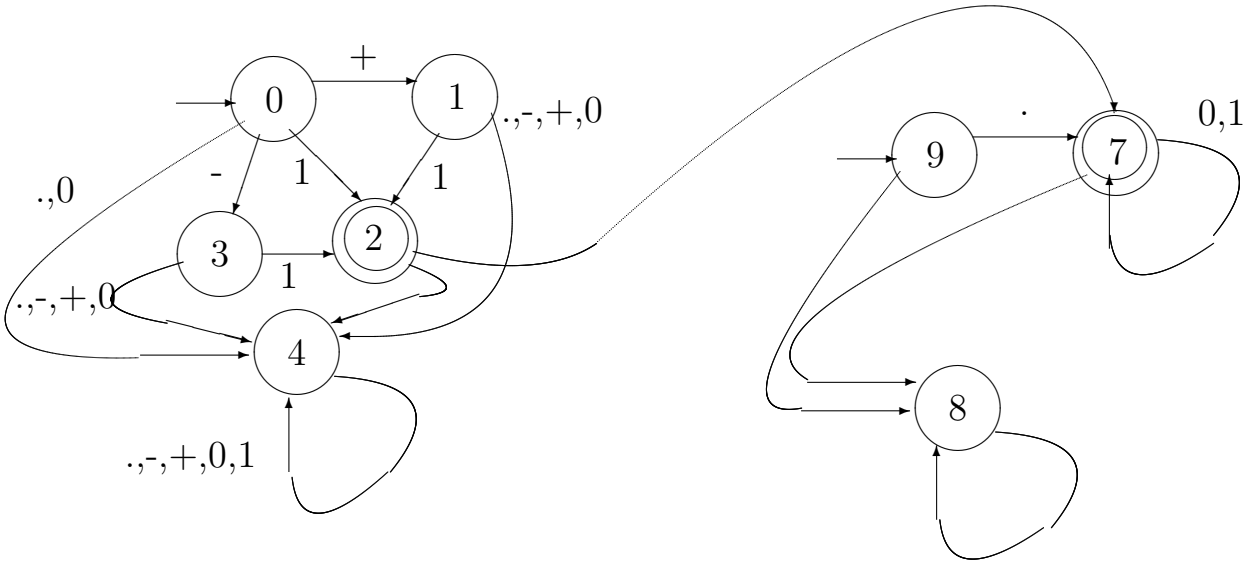
Problem: zunächst entsteht i.A. keine Übergangsfunktion sondern eine Übergangsrelation.
Das führt zum Begriff des nichtdeterministischen endlichen Automaten NEA.

Erst mit einer u.U. aufwendigen Konstruktion kommt man wieder zu einem DEA.

$$L = \{-1, +1, 1\}\,(\{.\}\,\{0,1\}^*)$$

$\{-1, +1, 1\}$ $\qquad$ $\{.\}\,\{0,1\}^*$

$A_4 = (Q_1, \Sigma, q_0 1, \delta_1, F_1)$ $\qquad$ $A_5 = (Q_2, \Sigma, q_0 2, \delta_2, F_2)$



Beispiel 7.7 (nach Satz 4.4.3 aus [11])

Für natürliche Zahl n sei L_n Menge aller Wörter über $\{0, 1\}$, bei denen 1 der n-letzte Buchstabe ist.

DEA für Erkennen von L_n hat mindestens 2^n Zustände.

Beweisidee: Betrachtung der Nerode-Relation (vgl. Definition 7.4) zeigt, daß alle Wörter aus $\{0, 1\}^n$ paarweise inäquivalent sind.

Definition 7.9

(1) Ein Tupel $A = (Q, \Sigma, q_0, \delta, F)$ heißt *nichtdeterministischer endlicher Automat (NEA)*, falls

   (a) Q und Σ endliche, paarweise disjunkte Mengen (genannt: Zustandsmenge und Eingabealphabet) sind,

   (b) $q_0 \in Q$ heißt *Startzustand*.

(c) $F \subseteq Q$ für die Endzustandsmenge F gilt und

(d) δ eine Abbildung $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ ist. Sie heißt *Zustandsüberführungsfunktion.*

(2) Die *erweiterte Zustandsüberführungsfunktion* $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ ist wie folgt definiert:

(a) $\delta^*(q, \varepsilon) = \{q\}$ und

(b) $\delta^*(q, wx) = \cup_{q' \in \delta^*(q,w)} \delta(q', x)$.

(3) Die von A *akzeptierte Sprache* ist $\{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$ und wird mit $L(A)$ notiert. A wird auch *Akzeptor* für $L(A)$ genannt.

□

Beispiel 4.4.2 aus [11]: NEA für String Matching Problem (Erkennen von Zeichenketten).

Beispiel 7.8 (nach Satz 4.4.3 aus [11])

Für natürliche Zahl n sei L_n Menge aller Wörter über $\{0, 1\}$, bei denen 1 der n-letzte Buchstabe ist.

Es gibt einen NEA für Erkennen von L_n mit $n + 1$ Zuständen.

Satz 7.4 *([11], Satz 4.6.10)*

Für DEA $A_1 = (Q_1, \Sigma, q_1, \delta_1, F_1)$ *und* $A_2 = (Q_2, \Sigma, q_2, \delta_2, F_2)$ *kann in der Zeit*

$O((|Q_1| + |Q_2|)|\Sigma|)$ *ein NEA konstruiert werden, der* $L(A_1)L(A_2)$ *akzeptiert.*

Beweis. Konstruktionsidee für den Automaten
$A = (Q, \Sigma, q_0, \delta, F)$:

$Q = Q_1 \cup Q_2$,

$q_0 = q_1$,

$$\forall q \in Q, a \in \Sigma.\ \delta(q, a) = \begin{cases} \{\delta_1(q, a)\} & \dots\ q \in Q_1 \setminus F_1 \\ \{\delta_1(q, a), \delta_2(q_2, a)\} & \dots\ q \in F_1 \\ \{\delta_2(q, a)\} & \dots\ q \in Q_2 \end{cases}$$

$$F = \begin{cases} F_1 \cup F_2 & \dots\ \varepsilon \in L(A_2) \\ F_2 & \dots\ \varepsilon \notin L(A_2) \end{cases}.$$

q.e.d.

Satz 7.5 *([11], Satz 4.6.12)*
Für DEA $A = (Q, \Sigma, q_0, \delta, F)$ kann in der Zeit $O(|Q||\Sigma|)$ ein NEA konstruiert werden, der $L(A)^*$ akzeptiert.

Beweis. Konstruktionsidee:
Schritt 1: Konstruktion eines Automaten $A' = (Q', \Sigma, q_0', \delta', F')$, der $L(A)^+$ akzeptiert.
$Q' = Q$,
$q_0' = q_0$,
$\forall q \in Q, a \in \Sigma.\ \delta'(q,a) = \begin{cases} \{\delta(q,a)\} & \dots \quad q \in Q \setminus F \\ \{\delta(q,a), \delta(q_0,a)\} & \dots \quad q \in F \end{cases}$
$F' = F$.
In zwei weiteren Schritten wird ein Automat A'' konstruiert, der genau das leere Wort akzeptiert. Aus beiden kann ein Automat konstruiert werden, der die Vereinigung $L(A)^* = L(A)^+ \cup \{\varepsilon\}$ akzeptiert. **q.e.d.**

Es bleibt die Frage, wie man für einen NEA einen äquivalenten DEA konstruiert.

Satz 7.6 *([11], Satz 4.4.5) Zu jedem NEA mit n Zuständen gibt es einen äquivalenten DEA mit 2^n Zuständen.*

Beweis.
Schritt 1: Es wird in einem ersten Schritt die Potenzmengenkonstruktion (vgl. [11], Algorithmus 4.4.4) verwendet. Konstruktion des Automaten $A' = (Q', \Sigma, q_0', \delta', F')$:
$Q' = \mathcal{P}(Q)$,
$q_0' = \{q_0\}$,
$\forall q' \in \mathcal{P}(Q), a \in \Sigma.\ \delta'(q', a) = \cup_{q \in q'} \delta(q, a)$
$F' = \{q' \in Q' \mid q' \cap F \neq \emptyset\}$.
Danach können uberflüssige Zustände beseitigt (Schritt 2) und der DEA minimiert werden (Schritt 3). **q.e.d.**

Diskussion: In der Praxis ist es notwendig, die Schritte 1 und 2 zu integrieren. Dies ist zumindest theoretisch möglich. Jedoch kann der entstehende nichtminimale DEA ohne überflüssige Zustände immer noch sehr groß sein.

Der folgende als Pumping-Lemma bezeichnete Satz markiert eine prinzipielle Schranke der Ausdrucksfähigkeit von regulären Sprachen. Der Satz sagt aus, daß es für jede reguläre Sprache L eine natürliche Zahl n gibt, so daß es für jedes Wort z, welches wenigstens die Länge n hat, eine Darstellung von z als uvw mit nichtleerem v gibt, so daß auch alle Wörter der Form uv^iw in die Sprache „hineingepumpt“ werden.

Satz 7.7

Es sei L eine reguläre Sprache über einem Alphabet Σ. Dann gibt es eine natürliche Zahl n, so daß es für jedes Wort $z \in L$ mit $|z| \geq n$ Wörter $u, v, w \in \Sigma^*$ gibt mit

(1) $z = uvw$,

(2) $v \neq \varepsilon$,

(3) $|uv| \leq n$ *und*

(4) $\forall i \in \mathbf{N}.\ uv^iw \in L$

Beweis. Der Beweis beruht auf folgender Idee:
Es sei n die Anzahl der Zustände eines L akzeptierenden DEA.
Dann wird beim Akzeptieren eines Wortes z mit einer Länge größer oder gleich n ein Weg mit mindestens $n + 1$ Zuständen durchlaufen. Das heißt, betrachtet man den Automaten wieder als Graphen, muß dieser Weg einen Zyklus enthalten.
Es sei u das Präfix von z, welches bis zum ersten Erreichen des Zyklus gelesen wird.
Es sei w das Suffix von z, welches nach dem ersten Durchlaufen des Zyklus gelesen wird.
Schließlich sei v derart, daß $z = uvw$.
Man zeigt leicht, daß $\forall i \in \mathbf{N} : uv^iw \in L$. **q.e.d.**

Der Beweis macht deutlich,

daß die *Endlichkeit der Speichermöglichkeiten* des Automaten der Grund für die beschriebene Ausdrucksschwäche ist.

Man kann Beispiele konstruieren, die zeigen,
daß reguläre Sprachen insbesondere nicht ausreichen,
um Klammerstrukturen korrekt zu beschreiben.

Beispiel 7.9 Wir betrachten die Sprache

$$L_3 = \left\{ cx_1cx_2 \ldots cx_nd^n \mid n \in \mathbf{N}, 0 < n, x_1, x_2, \ldots, x_n \in \{a\}^+ \right\}$$

aus Beispiel 7.1. Angenommen, L_3 sei regulär. Dann sei n die natürliche Zahl, die nach dem Pumping-Lemma existiert. Es sei z das Wort aus L_3 mit der Form

$$caca \ldots cad^{n+1}$$

und

$$uvw$$

seine Zerlegung, die nach dem Pumping-Lemma existiert.

Zunächst kann ausgeschlossen werden, daß v Teilwort eines Wortes a ist. Denn dann wäre ein Wort der Form

$$caca \ldots cc \ldots cad^{n+1}$$

oder

$$caca \ldots cacd^{n+1}$$

in L_3 in Widerspruch zur Definition.

Da $|uv| \leq n$ ist, kann v außerdem das Alphabetssymbol d nicht enthalten. Es muß demnach das Symbol c enthalten.

Dann gilt aber bereits

$$uvvw \notin L_3$$

Widerspruch. Folglich ist L_3 nicht regulär.

7.1.2 Anwendung regulärer Ausdrücke zur Beschreibung von Textmustern

In verschiedenen Computersystemen steht ein Programm `grep` (engl.: global regular expression print) zur Verfügung, mit dem in Textdateien nach Mustern gesucht werden kann, die durch reguläre Ausdrücke beschrieben sind. Die vom Programm `grep` benutzte Syntax wird auch in Editoren (z.B. in Emacs oder im Editor des Turbo-Pascal) verwendet, um Textmuster für Such- und andere Funktionen anzugeben. Die Zeichen `*` und `+` bedeuten die oben definierten Formen der Iteration. Allerdings beziehen sie

sich nur auf das vor ihnen stehende Zeichen. `^` bezeichnet den Zeilenanfang, `$` das Zeilenende, `.` ein beliebiges Zeichen. Die eckigen Klammern dienen als Mengenklammern. Mit dem Minuszeichen `-` können Aufzählungen abgekürzt werden. Mit `[a-z0-9]` werden beispielsweise alle Kleinbuchstaben und Ziffern beschrieben. `[^a-z0-9]` bezeichnet das Komplement dieser Zeichenmenge. Ein am Anfang einer Auflistung stehender Zirkumflex bewirkt, daß das Komplement der Menge gebildet wird.

Beispiel: `grep ^alpha.*$ text` sucht in der Datei `text` alle Zeilen, in denen am Zeilenanfang das Wort `alpha` steht.

Bei der Angabe eines Dateinamens können die Platzhalter `*` und `?` verwendet werden.

`grep "[a-z][a-z0-9 ]* *: *real" ex.pas` sucht in dem Programm `ex.pas` alle Vereinbarungen einer Variablen oder eines Parameters vom Typ `real`. Das Suchmuster muß in diesem Fall in Anführungsstriche eingeschlossen werden, weil das Leerzeichen sonst als Trennzeichen zwischen verschiedenen Parametern für das Programm aufgefaßt würde.

7.1.3 Konstruktion von Akzeptoren für reguläre Sprachen

Für die Konstruktion von Übersetzern bzw. Interpretern ist ein Unterprogramm notwendig, welches die lexalischen Analyse ausführt.

Mit den Konstruktionen aus den Sätzen 7.3, 7.4 ,7.5 ist man prinzipiell in der Lage, aus regulären Ausdrücken äquivalente DEA zu konstruieren. Diese wiederum kann in ausführbare Programme übersetzen. Dieses Verfahren ist in Werkzeugen wie `lex` (kommerziell) oder `flex` (freie Software) realisiert. Diese Werkzeuge übersetzen reguläre Ausdrücke in C-Programme.

Algorithmen (Programme), mit deren Hilfe die Zugehörigkeit von Wörtern zu einer Sprache entschieden werden kann, werden als *Akzeptoren* bezeichnet. Hier wird die automatische Konstruktion von Akzeptoren für reguläre Sprachen behandelt.

In Abbildung 35 ist ein Eingabedatei für das Programm `lex` dargestellt.

7.1.4 Der Mealy-Automat

Definition 7.10

(1) Ein Tupel $A = (Q, \Sigma, \Omega, q_0, \delta, \gamma, F)$ heißt *deterministischer Mealy-Automat*, falls

   (a) Q, Σ und Ω endliche, paarweise disjunkte Mengen (genannt: Zustandsmenge, Eingabe- und Ausgabealphabet) sind,

   (b) $q_0 \in Q$ heißt *Startzustand*.

   (c) $F \subseteq Q$ für die Endzustandsmenge F gilt und

   (d) δ, γ Abbildungen $\delta : Q \times \Sigma \to Q$ bzw. $\gamma : Q \times \Sigma \to \Omega$ sind. Sie heißen *Zustandsüberführungs-* bzw. *Ausgabefunktion*.

(2) Die *erweiterte Zustandsüberführungsfunktion* und die *akzeptierte Sprache* sind wie beim deterministischen endlichen Automaten definiert.

(3) Die vom Mealy-Automaten A *berechnete Funktion* ist ist die wie folgt definierte $f : \Sigma^* \to \Omega^*$:

   (a) Zunächst wird die Funktion γ zu $\gamma^* : Q \times \Sigma^* \to \Omega^*$ erweitert.

      i. $\gamma^*(q, \varepsilon) = \varepsilon$ und

      ii. $\gamma^*(q, wx) = \gamma^*(q, w)\gamma(\delta^*(q, w), x)$.

   (b) Nun wird f durch γ^* definiert: $f(w) = \gamma^*(q_0, w)$,

□

7.1.5 Grammatiken und Sprachen

Am Beispiel der Sprache L_3 wurde demonstriert, daß es Sprachen gibt, die nicht regulär sind.
Auf ähnliche Weise kann man zeigen, daß eine Programmiersprache wie Pascal ebenfalls nicht regulär ist.
Dazu genügt es zu beobachten,
daß in Programmiersprachen vielfältige Klammerstrukturen auftreten, z.B. verschachtelte `begin - end` Klammerungen.

Im folgenden werden die Syntaxdefinition und die Syntaxanalyse für Programmiersprachen betrachtet.
Dabei sind die *kontextfreien Sprachen (Grammatiken)* ein wichtiges Hilfsmittel.

Definition 7.11 Ein Quatrupel $G = (T, H, s, R)$ heißt *Grammatik*, falls folgende Eigenschaften erfüllt sind.

(1) H, T und R sind endliche und paarweise disjunkte Mengen. H und T heißen *Terminal-* bzw. *Nichtterminalalphabet*, die Elemente von R *Regeln*, s Startsymbol.

(2) $R \subseteq ((H \cup T)^* \setminus T^*) \times (H \cup T)^*$.

(3) $s \in H$.

Ein Wort y über $T \cup H$ heißt *direkt ableitbar aus* $x \in (T \cup H)^*$, falls es eine Regel $(l, r) \in R$ und Wörter $u, v \in (T \cup H)^*$ gibt mit $x = ulv$ und $y = urv$.
Notation: $x \rightarrow_R y$.
Ein Wort y über $T \cup H$ heißt
ableitbar aus $x \in (T \cup H)^*$,
falls es eine endliche Folge von Wörtern $z_1, \ldots, z_n$ gibt, so daß $z_1 = x$, $z_n = y$ und für alle $i = 1, \ldots, n-1$ gilt $z_i \rightarrow_R z_{i+1}$.
Notation: $x \rightarrow_R^* y$.
Die Menge $L(G) = \{x \mid x \in T^*, s \rightarrow_R^* x\}$ heißt *von G erzeugte Sprache*.
□

Definition 7.12 Eine Grammatik $G = (T, H, s, R)$ heißt *kontextfrei*, falls

$$R \subseteq H \times (H \cup T)^*.$$

□

Beispiele: Alle regulären Sprachen kontextfrei. Die Sprache L_3 ist auch kontextfrei.

Eine wichtige Anwendung der kontextfreien Grammatiken ist die Syntaxdefiniiton von Programmiersprachen. Bei der Definition von Programmiersprachen geht man oft in zwei Schritten vor: 1. Man definiert mit einer kontextfreien Grammatik zunächst eine Obermenge der betrachteten Sprache. 2. Mit Hilfe von Kontextbedingungen wählt man daraus die eigentliche Programmiersprache aus. Typische Kontextbedingungen betreffen die Definiertheit von Variablen und anderen Sprachkonstrukten sowie deren Eigenschaften. Alle verwendeten Namen müssen vereinbart oder vordefiniert sein. Die Vereinbarungen müssen der Verwendung entsprechen.

Zunächst führen wir eine bequemere Notation für die Grammatiken ein, die sogenannte Backus-Naur-Notation (BNF). Es sei eine Grammatik $G = (T, H, s, R)$ gegeben.

Backus-Naur-Notation (BNF) :

Regeln (r, w_1),..., (r, w_n), deren linke Seite das gleiche Nichtterminalsymbol r haben, können in folgender Schreibweise zusammengefaßt werden:

$$r ::= w_1 | \ldots | w_n$$

Wichtig ist die Effizienz der Syntaxanalyse. Die Kosten der Syntaxanalyse sollten linear in der Größe n des zu übersetzenden Programmtextes sein, schlimmstenfalls proportional zu $n\ logn$.

Wie skizzieren jetzt ein Syntaxanalyseverfahren welches folgenden Anforderungen gerecht wird. Ausserdem wird erläutert, welche Anforderungen Grammatiken erfüllen müssen, damit dieses Verfahren benutzt werden kann.

(1) Jeweils nächster Analyseschritt muß sich aus aktuellem Zustand der Analyse (sog. top-down Analyse) und dem gerade eingelesenem Symbol (Schlüsselwort der Programmiersprache) ergeben (engl.: one-symbol-lockahead).

(2) Analyseschritte brauchen nicht rückgängig gemacht zu werden. D.h. keine Suche notwendig!

Beispiel 7.10
Für Grammatik

$$G_8 = (\{x, y, z\}, \{s, t\}, s, \{(s, xt), (t, z), (t, yt)\})$$

wird eine Ableitung des Wortes $xyyz$ und darunter der entsprechende Ablauf der Analyse gezeigt. Es werden jeweils die Folgen angewendeten Regeln und der noch zu lesenden Zeichen angegeben. In der vierten Zeile wird das jeweils gerade eingelesene Zeichen angegeben. In der letzten Zeile ist das bereits analysierte Teilwort angegeben. Eine Regel der Form (h, h') wird so interpretiert, daß die Zeichenreihe x eingelesen wird.

Ableitung:	s	$\rightarrow_R$	xt	$\rightarrow_R$	xyt	$\rightarrow_R$	$xyyt$	$\rightarrow_R$	$xyyz$
benutzte Regel:		(s, xt)		(t, yt)		(t, yt)		(t, z)	
noch zu analysieren:	$xyyz$		yyz		yz		z		ε
gerade gelesen		x		y		y		z	
bereits analysiert	ε		x		xy		xyy		$xyyz$

In jedem Schritt konnte die Entscheidung über den nächsten Schritt anhand des Zustands der Analyse und des gerade eingelesenem Zeichen gemacht werden.

Beispiel 7.11

Für Grammatik $G_9 =$
$(\{x, y, z\}, \{s, r, t\}, s, \{(s, r), (s, t), (r, y), (t, z), (r, xr), (t, xt)\})$
wird der Versuch einer Ableitung und ein Analyseversuch des Wortes $xxxz$ gezeigt. Eine Regel der Form (h, h') wird so interpretiert, daß kein Zeichen eingelesen wird.

Ableitung:	s	$\rightarrow_R$	r	$\rightarrow_R$	xr	$\rightarrow_R$	xxr	$\rightarrow_R$	xxx
benutzte Regel:		(s, r)		(r, xr)		(r, xr)		(r, xr)	
noch zu analysieren:	$xxxz$		$xxxz$		xxz		xz		z
gerade gelesen				x		x		x	
bereits analysiert	ε		ε		x		xx		xxx

Der Versuch endet in einer Sackgasse, weil bereits die erste Entscheidung, nämlich die Regel (s, r) zu verwenden, falsch war. Das wird aber für ein Analyseprogramm erst nach mehreren Schritten sichtbar. Eine Obergrenze für die „Tiefe der Sackgasse“ ist nicht angebbar. Das führt dazu, daß die Komplexität der Analyse exponentiell werden kann.

Übung 7.1
Die Grammatik
$G_{12} =$
$(\{x, n, *, (,), :, =, \underline{if}, \underline{then}\}, \{A, P, D, I\}, I, R))$

hat die in BNF gegebenen Regeln

$$\begin{aligned} I &::= n := A \mid \underline{if}\ A = A\ \underline{then}\ I \\ A &::= x \mid (P) \\ P &::= AD \\ D &::= \mid *AD \end{aligned}$$

Übung 7.2
Die Grammatik

$G_{13} =$
$(\{a, b, c, d, (,), \Leftarrow, \Rightarrow, \Uparrow, \Downarrow, \nearrow, \nwarrow, \searrow, \swarrow\}, \{S, A, B, C, D\}, S, R)$
hat die in BNF gegebenen Regeln

$$\begin{aligned} S &::= A \searrow B \swarrow C \nwarrow D \nearrow \\ A &::= a \mid (A \searrow B \Rightarrow D \nearrow A) \\ B &::= b \mid (B \swarrow C \Downarrow A \searrow B) \\ C &::= c \mid (C \nwarrow D \Leftarrow B \swarrow C) \\ D &::= d \mid (D \nearrow A \Uparrow C \nwarrow D) \end{aligned}$$

Hintergrund für dieses Beispiel: Werden die mit dieser Grammatik ableitbaren Wörter als (geeignet skalierte) Anweisungen an einen Plotter interpretiert, geben Wörter bei denen alle Ableitungen von Terminalsymbolen die gleiche Tiefe haben, Vorschriften für das Zeichnen von Sierpiński-Kurven an.

7.1.6 Syntaxdiagramme

Am Beispiel einer sehr einfachen Grammatik wird nun eine weitere Darstellungsform von Grammatiken erläutert, die sogenannten Syntaxdiagramme. Sie werden z.B. in den von N. Wirth geschriebenen Büchern über Pascal benutzt.

Die Beispielgrammatik hat die Terminalsymbole $x, (,), +$ und die Nichtterminalsymbole A, B, C. A ist gleichzeitig das Startsymbol. Die Regeln haben die Form

$$\begin{array}{lcl} A & ::= & x|(B) \\ B & ::= & AC \\ C & ::= & |+AC \end{array}$$

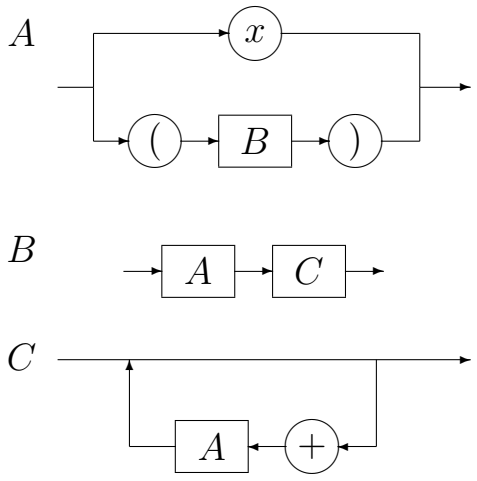
Man überzeugt sich leicht, daß die Kriterien 1 und 2 eingehalten werden.

Man übersetzt nun diese Grammatik in Syntaxdiagramme (vgl. Abbildung 7.1.6). Für jedes Nichtterminal wird ein Syntaxdiagramm angegeben. Die Syntaxdiagramme können als Beschreibung des Syntaxanalyseprozesses ähnlich den Programmablaufplänen betrachtet werden. Terminal- und Nichtterminalsymbole treten als Knoten auf. Zur Unterscheidung werden erstere mit Kreisen (oder Ovalen) umschrieben, letztere mit Kästchen.

Ähnlich wie Ablaufdiagramme die Abfolge der Ausführung von Anweisungen angeben, beschreiben Syntaxdiagramme den Ablauf der Syntaxanalyse. Wird ein Kreis erreicht, muß das angegebene Terminalsymbol gelesen werden. Wird ein Kästchen erreicht, wird nach der Vorschrift des dort benannten Diagramms fortgefahren.

```
ID              [a-z][a-z0-9]*

%%

{ID}            { printf("1 Name:         %s\n",yytext); }
{ID}[,]{ID}     { printf("2 Namen:        %s\n",yytext); }
{ID}(","{ID})*  { printf("viele Namen:    %s\n",yytext); }
a(-(c|d))*e     { printf("gefunden:       %s\n",yytext); }
.               { printf("*** falsch:     %s\n",yytext); }

%%

/*--------------------------------------------------------------------
** Hauptprogramm:   main()
** Zweck:     Versucht, ein Wort der Spraches einzulesen,
**           solange Dateiende nicht erreicht ist.
--------------------------------------------------------------------*/
int main()
{
  while ( yylex() )  {}
  return 0;
}

/*--------------------------------------------------------------------
** Prozedur:        yywrap()
** Zweck:     sagt lex dass nach EOF kein Fortsetzung moeglich.
** Argumente:       keine
** Rueckgabewert:   TRUE
--------------------------------------------------------------------*/
int yywrap()
{   return 1;
}
```

Abbildung 35: Eingabedatei für automatische Konstruktion eines Scanners



$$
\begin{aligned}
A &::= x|(B) \\
B &::= AC \\
C &::= |+AC
\end{aligned}
$$

Abbildung 36: Syntaxdiagramme für eine kontextfreie Sprache

```
    uses crt;
        var ch:char; correct:boolean;
        procedure next;        ...
        procedure test(c:char);        ...
        procedure B; forward;
        procedure A;
        { A ::= x|(B) }
        begin
             case ch of
                  'x' : next;
                  '(' : begin next; B; test(')'); next end;
             else correct := false
             end
        end;
        procedure C; forward;
        procedure B;
        { B ::= AC }
        begin
             A; C
        end;
        procedure C;
        { C ::= |+AC }
        begin
             while ch='+' do
             begin
                  next;
                  A
             end
        end;
begin
{ Initialisierung }
     correct:=true; next;
{ A  ist Startsymbol }
     A;
     ...
end.
```

Abbildung 37: Akzeptor für eine kontextfreie Sprache

Literatur

[1] L. Banachowski and A. Kreczmar. *Elementy analizy algorytmów*. Wydawnictwo Naukowo-Techniczne, 1982.

[2] L. Banachowski, A. Kreczmar, and W. Rytter. *Analysis of Algorithms and Data Structures*. Addison-Wesley, 1991.

[3] H.-P. Gumm and M. Sommer. *Einfürung in die Informatik*. R. Oldenbourg Verlag, 1998.

[4] C. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[5] C. Horn and I. O. Kerner. *Lehr- und Übungsbuch Informatik, Band 1: Grundlagen und Überlick*. Fachbuchverlag Leipzig, 1995.

[6] F. Kröger. *Einführung in die Informatik*. Springer Berlin, 1991.

[7] R. Müller and R. Hopfer. *Turbo–PASCAL*. Fachbuch–Verlag Leipzig, 1991.

[8] Nusser. *Sicherheit im Internet*. xyz, 1998.

[9] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*, volume 70 of *Informatik*. BI Wissenschaftsverlag, 1 edition, 1993.

[10] B. Stroustrup. *The C++ Programming Language*. Addison wesley, 2nd edition, 1995.

[11] I. Wegener. *Theoretische Informatik*. B. G. Teubner, 1993.

[12] N. Wirth. *Revidierter Bericht über die Programmiersprache PASCAL*. Akademie–Verlag Berlin, 1976.

[13] N. Wirth. *Algorithmen und Datenstrukturen*. B. G. Teubner Stuttgart, 1983.

[14] N. Wirth. *Systematisches Programmieren*. B. G. Teubner Stuttgart, 1993.