Automaten

Ein Automat hat im weitesten Sinne die Eigenschaft, sein Verhalten selbst zu steuern. Eine Grundlage für diese Eigenschaft bilden Speicherelemente über die Automaten im Gegensatz zu Schaltnetzen verfügen. Diese Speicherelemente (Flip-Flops) verwalten innere Zustände des Automaten. Damit ist der Ausgang vom Eingang *und* vom inneren Zustand abhängig, während Schaltnetze eben *nur* vom Eingang abhängen. Es ist möglich und sinnvoll, den Automaten als algebraische Struktur aufzufassen. In der Literatur werden Automaten (z.B. Transduktoren) so als 6 - Tupel wie folgt definiert.

Ein Automat A (X, Y, S, s_0, δ, ω) ist definiert durch :

- X ist das Eingabealphabet (endliche, nicht leere Menge von Symbolen)
- Y ist das Ausgabealphabet (endliche, nicht leere Menge von Symbolen)
- S ist eine endliche, nichtleere Menge von Zuständen
- s_0 ist der Anfangszustand mit $s_0 \in S$
- δ (delta) ist die Zustandsübergangsfunktion $\delta : SxX \rightarrow S$
- ω (omega) ist die Ausgabefunktion

Ausgehend von dieser algebraischen Definition wird der Begriff des *endlichen deterministischen Automaten (finite and deterministic state machine)* abgeleitet. Wenn der Automat (nur) über eine endliche Anzahl von Zuständen verfügt und die Zustandsübergänge eindeutig sind, handelt es sich um diese Art Automat.

Arten von Automaten

In verschiedenen Bereichen des täglichen Lebens werden Automaten benötigt. Gebräuchliche und einfach zu erfassende sind der Warenautomat oder auch Automaten, welche Eingaben erkennen und entsprechend reagieren. Geht man von endlichen Automaten aus, gibt es zwei Gruppen die in unterschiedlichen Kontexten Verwendung finden.

Akzeptoren

Akzeptoren werden in der Regel zur Spracherkennung eingesetzt. Sie signalisieren durch ihren jeweiligen Zustand die Richtigkeit eines Wortes. Ausgabealphabet und Ausgabefunktion entfallen, da nur akzeptierte Zustände erwartet werden. Der Automat liest also *sequentiell* die Zeichen des Übergabewortes. Eine Azeptanz liegt genau dann vor, wenn sich der Automat nach dem Lesen des letzten Zeichens in einem akzeptierten Zustand befindet.

Eine mögliche Definition lautet A (X, S, s_0, δ, E) ist definiert durch :

- X ist das Eingabealphabet (endliche, nicht leere Menge von Symbolen)
- S ist eine endliche, nichtleere Menge von Zuständen

- s_0 ist der Anfangszustand mit $s_0 \in S$
- δ (delta) ist die Zustandsübergangsfunktion $\delta : SxX \rightarrow S$
- E ist eine Menge von Endzuständen mit $E \subseteq S$

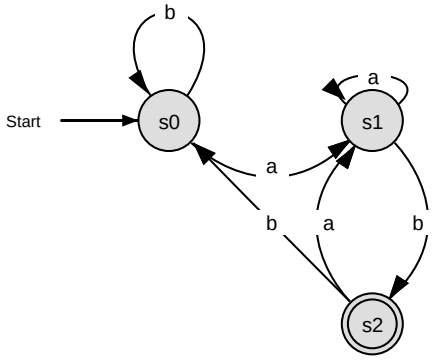
Transduktoren

Auch hier unterscheidet man zwei Gruppen. Die erste Gruppe bezeichnet die *Moore-Automaten*. Diese sind dadurch gekennzeichnet, dass die Ausgabe des Automaten *nur* vom aktuellen Zustand abhängig ist. Die andere Gruppe beinhaltet die *Automaten*, deren Ausgabe vom Zustand *und* der Eingabe abhängig ist. Ein Fahrscheinautomat, der eine gültige Karte ausgibt, nachdem ausreichend Geld eingeworfen wurde, wäre also ein Moore Automat, da nur der Zustand relevant ist, bei dem der Nutzer den Schein erhält. Soll dagegen jedesmal ausgegeben werden, wieviel Geld noch fehlt, bzw. was bezahlt wurde, handelt es sich um einen Mealy - Automaten. Die Prüfung einer Zeichenkette auf Richtigkeit (mit einer entsprechenden Ausgabefunktion) entspräche ebenso einem Moore Automaten. In diesem Sinne können Akzeptoren als besondere Moore Automaten (ohne Ausgabe aufgefasst) werden.

Beispiele

(a) Ein relativ einfaches Beispiel zum Anfang. Auf einer Eingangsleitung werden die Buchstaben a und b gesendet. Ein akzeptierter Zustand soll dann angenommen werden, wenn die Folge *ab* auftritt. Daraus ergeben sich folgende Übergangstabelle und folgendes Übergangsdiagramm.

Übergangsdiagramm



Festlegung der Zustände

X = {a,b}

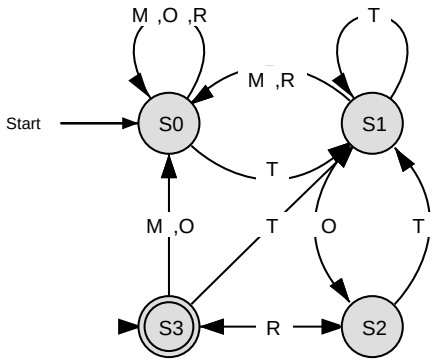
S_0 = Startzustand
S_1 = a
S_2 = ab

Übergangstabelle

	a	b
S_0	S_1	S_0
S_1	S_1	S_2
S_2	S_1	S_0

(b) Auf einer Eingangsleitung werden die Buchstaben M,T,O,R gesendet. Die Aufgabe des Automaten besteht darin, zu überprüfen, ob die Zeichenkette TOR enthält. Ist dieser Zustand erreicht, soll ein TRUE (1) geliefert werden. Es handelt sich hierbei um das Modell eines Moore Automaten da eine Ausgabefunktion vorgesehen ist.

Übergangsdiagramm



Festlegung der Zustände

X = {M, T, O, R}
Y = {1,0}

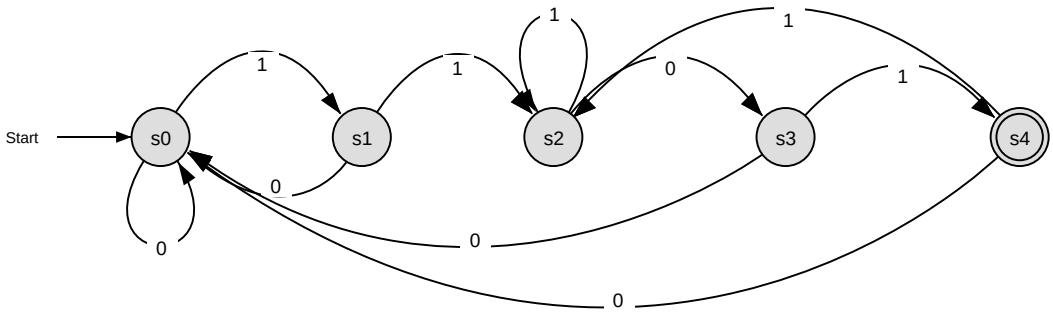
S_0 = Startzustand
S_1 = T
S_2 = TO
S_3 = TOR

Übergangstabelle

	M	O	T	R	Ausgang
S_0	S_0	S_0	S_1	S_0	0
S_1	S_0	S_2	S_1	S_0	0
S_2	S_0	S_0	S_1	S_3	0
S_3	S_0	S_0	S_1	S_3	1

(c) Das folgende Beispiel ist ebenfalls ausgehend vom Modell des *Moore Automaten* erfassbar. Es soll die Zahl 13, welche in binärer Form vorliegt, erkannt werden. Einschließlich des Startzustandes ist hierfür die Bereitstellung von 5 Zuständen erforderlich.

Übergangsdiagramm



Festlegung der Zustände

X = {1,0}
Y = {1,0}

S_0 = Startzustand
S_1 = 1
S_2 = 11
S_3 = 110
S_4 = 1101

Übergangstabelle

	0	1	Ausgang
S_0	S_0	S_1	0
S_1	S_0	S_2	0
S_2	S_3	S_2	0
S_2	S_0	S_4	0
S_3	S_0	S_2	1

(d) Das nächste etwas komplexere Beispiel eines *Moore Automaten* stellt der sogenannte Rauschfilter dar. Er dient der Verarbeitung einer Folge von 1-en und 0-en in der Weise Sequenzen zu glätten. Ist eine 1 von 0-en umgeben, wird Sie durch eine 0 ersetzt, ist eine 0 von 1-en umgeben, wird sie durch eine 1 ersetzt. In der Literatur wird der praktische Bezug zu Bildern hergestellt. Hier werden Schwarz - Weiß Bilder durch eine Folge von 0-en und 1-en digitalisiert. Treten vereinzelte 0-en oder 1-en auf, werden diese als Fehler gewertet und geglättet bzw. gefiltert.

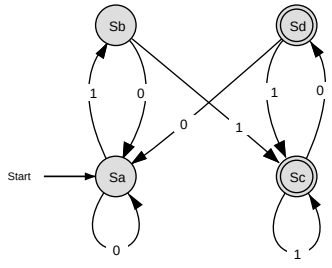
Der entsprechende Automat besitzt 2 akzeptierte Endzustände. Die Ausgabe in diesen Fällen soll eine 1 sein. Im Falle der Nichtakzeptanz liefert die Ausgabefunktion eine Null. Der Automat wird über 4 Zustände erfasst.

Übergangsdiagramm

Festlegung der Zustände

X = {1,0}
Y = {1,0}

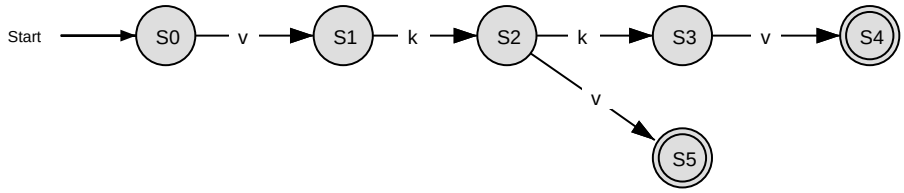
S_a = es wurde eine Folge von mind. 2 0-en gesehen *oder* 2 0-en gefolgt von einer 10
S_b = es wurde eine Folge von 0-en gefolgt von einer 1 gesehen
S_c = es wurde eine Folge von mind. 2 1-en gesehen *oder* 2 1-en gefolgt von einer 01
S_d = es wurde eine Folge von 1-en gefolgt von einer 0 gesehen



Übergangstabelle

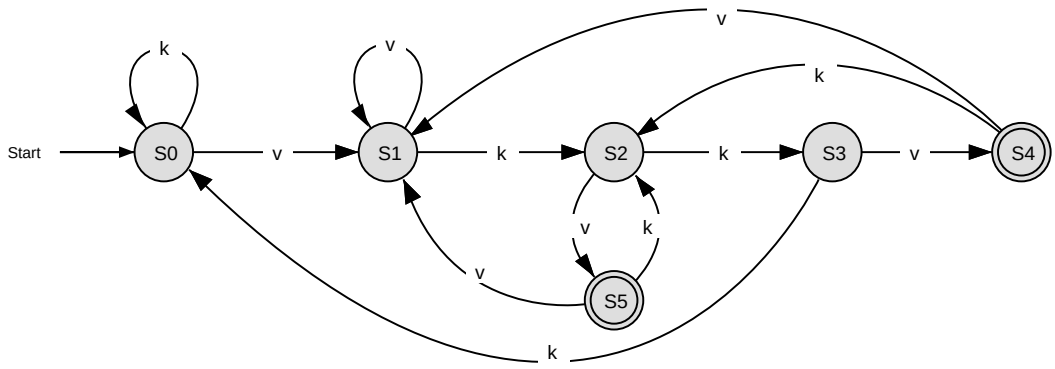
	0	1	Ausgang
S_a	S_a	S_b	0
S_b	S_a	S_c	0
S_c	S_d	S_c	1
S_d	S_a	S_c	1

(e) Der Einsatz eines erkennenden Automaten kann auch am Beispiel der Silbentrennung [1] veranschaulicht werden. Die Regeln zur Trennung sind sehr komplex und vollständig schwer als Programm realisierbar. Beschränkt man sich jedoch auf die Regeln *v-kv* und *vk-kv*, wobei *v* einen beliebigen Vokal und *k* einen beliebigen Konsonanten darstellt, läßt sich der Automat mit seinen Zustandsübergängen, leicht implementieren [2] . Die Besonderheit besteht im Auftreten zweier Endzustände.



Überträgt man dieses Vereinfachte Diagramm in eine Übergangstabelle und entwickelt den Übergangsgraphen ergibt sich folgendes Bild. Dieses kann nun zur Simulation nach *Autoedit* übertragen werden.

Übergangsdiagramm



Festlegung der Zustände

X = {v,k}
Y = {1,0}

S_0 = Startzustand
S_1 = v
S_2 = vk
S_3 = vkk
S_4 = vkkv
S_5 = vkv

Übergangstabelle

	k	v	Ausgang
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_3	S_5	0
S_3	S_0	S_4	0
S_4	S_2	S_1	1
S_5	S_2	S_1	1

Eine Umsetzung des Automaten als Klasse in Python hat folgendes Aussehen. Hier werden im wesentlichen nur die Zustandsübergänge durch einfache Verzweigungen

[1] vgl. Informatik Niedersachsen
[2] Lösungsansatz Silbentrenner

abgefangen.

```
1  class Pruefautomat:
2
3      def __init__(self):
4        self.zustand = 0
5
6      def uebergang (self, zustand, zeichen):
7
8         if  zustand == 0 :
9                 if  zeichen == 'k': return 0
10                elif zeichen == 'v': return 1
11        elif zustand == 1 :
12                if  zeichen == 'k': return 2
13                elif zeichen == 'v': return 1
14        elif zustand == 2 :
15                if  zeichen == 'k': return 3
16                elif zeichen == 'v': return 5
17        elif zustand == 3 :
18                if  zeichen == 'k': return 0
19                elif zeichen == 'v': return 4
20        elif zustand == 4 :
21                if  zeichen == 'k': return 2
22                elif zeichen == 'v': return 1
23        elif zustand == 5 :
24                if  zeichen == 'k': return 2
25                elif zeichen == 'v': return 1
26
27     def verarbeite (self, zeichen):
28        self.zustand = self.uebergang(self.zustand, zeichen)
29
```

Aufgaben

(a) Schreiben Sie ein Programm in Python, welches einen endlichen Automaten simuliert, welcher dem Erkennen von Vokalen in einem Wort dient. Sind alle Vokale enthalten, soll das Programm eine entsprechende Meldung liefern. Verwenden Sie als Struktur eine Klasse.

(b) Entwickeln Sie Übergangstabelle und Übergangsgraph für einen Moore Automaten, der dem Test auf Teilbarkeit durch 4 dienen soll. Die zu testenden Daten oder Eingabeworte sollen Binärzahlen sein. Dabei kann sich der Test auf die Binärzahlen beschränken, die auf 100 enden.

(c) Erstellen Sie Übergangstabelle und Übergangsgraph für einen Moore - Automaten, der die Ausgabe 1 liefert, sobald auf der Eingangsleitung 3 aufeinanderfolgende 1-en in einer beliebigen Folge von 0-en und 1-en auftreten.

(d) Entwickeln Sie den Übergangsgraphen für einen sogenannten Gleichheitsprüfer. Die auf der Eingangsleitung ankommende Folge von Nullen und Einsen soll nur dann

akzeptiert werden, wenn die Kette eine gerade Anzahl von 1-en und 0-en hat.

(e) Entwickeln Sie ein Programm, welches der Silbentrennung dient und sich auf die Klasse *Pruefautomat* abstützt. Testen Sie die Funktionalität der Klasse an geeigneten Beispielen.

(f) Simulieren Sie die Automaten mit dem Programm *Autoedit*, welches unter *http://genesis-x7.de* [3] kostenlos gezogen werden kann.

Beispiel für einen Mealy - Automaten

Der Hauptunterschied zwischen einem Moore und einem Mealy - Automaten bestand im unterschiedlichen Ausgabeverhalten beider Typen. Während die Ausgabe des Moore - Automaten vom inneren Zustand abhängt ist beim Mealy Automaten, wie oben beschrieben, zusätzlich die Eingabe entscheidend. Beispiele sind z.B. Fahrkarten - oder Kaffeeautomaten. Letztere waren Gegenstand einer Prüfungsaufgabe in Mecklenburg und hatte etwa folgenden Inhalt.

Der Kaffeeautomat verfügt über einen Eingabeschlitz in den nur 1-Euro Münzen eingeworfen werden können. Der maximale Wert für den Einwurf beträgt 2 Euro. Darüber hinaus befinden sich am Automaten 2 Tasten. Eine Taste Kaffee klein (KK) mit der Aufschrift 1 Euro und eine Taste Kaffee groß (KG) mit der Aufschrift 2 Euro. Der Automat verfügt über ein Ausgabefach. Da es sich um einen Mealy Automaten handelt, hängt die Ausgabe von dem erreichten Zustand (eingeworfenes Geld) und der Eingabe ab. Ein potenzieller Denkfehler besteht in der Tatsache, dass im Falle einer Ausgabe der Automat nicht nur den gewünschten Kaffee liefert, sondern eben auch Geld abzieht, wodurch sich wiederum der jeweilige Zustand ändert. Insgesamt ergibt sich so folgendes Modell.

Festlegung der Zustände

$X = \{E_1, E_2, E_3\}$
$Y = \{Y_0, Y_1, Y_2\}$

S_0 = 0 Euro eingeworfen
S_1 = 1 Euro eingeworfen
S_2 = 2 Euro eingeworfen

E_1 = Einwurf 1 Euro
E_2 = Druck Taste KG
E_3 = Druck Taste KK

Y_0 = keine Ausgabe
Y_1 = Ausgabe Kaffee klein
Y_2 = Ausgabe Kaffee groß

Übergangstabelle

	E_1	E_2	E_3
S_0	$S_1\|Y_0$	$S_0\|Y_0$	$S_0\|Y_0$
S_1	$S_2\|Y_0$	$S_1\|Y_0$	$S_0\|Y_1$
S_2	$S_2\|Y_0$	$S_0\|Y_2$	$S_1\|Y_1$

[3] Programm Autoedit

Aufgabe

Konstruieren Sie ein vergleichbares Beispiel für einen Mealy - Automaten und stellen Sie das entsprechende Modell auf.

Zusammenhang Automat und Sprache

Grundlage einer Sprache bildet das *Alphabet*. Das Alphabet wird in der Literatur oft mit dem Symbol $\sum$ (Sigma) dargestellt und meint die Menge aller Zeichen. Eine Folge von Zeichen stellt das *Wort* dar. Ein Wort der Länge 0, also das leere Wort wird durch ϵ beschrieben.
Gebräuchlich ist ebenfalls die Bezeichnung $\sum^*$ für die Menge aller Worte über einem Alphabet $\sum$.

Um Worte zu erzeugen bzw. Zeichenketten zu generieren, benötigt man eine *Grammatik G*, welche ein sogenanntes Ersetzungssystem darstellt. Gemeint ist hiermit, dass mit Hilfe der Grammatik ausgehend von einem Startbuchstaben und einer Menge von Regeln Wörter erzeugt werden können, wobei die Menge aller Wörter, die erzeugt werden können, die zur Grammatik gehörende *Sprache L(G)* darstellt.

Eine formale Definition einer Grammatik $(N, \sum, R, S)$ liest sich wie folgt.

- N ist die Menge aller Nichtterminalsysmbole
- $\sum$ ist die Menge aller Terminalsymbole
- R bezeichnet eine endliche Menge von Regeln zur Erzeugung der Sprache L(G)
- $S \in N$ bezeichnet das Startsymbol

Beispiele

(a)

G = {N, $\sum$, R, <Satz>}
N = {<Satz>, <Subjekt>, <Prädikat>, <Objekt>}
$\sum$ = {Bruno, knuffelt, Britta}

R =
{ (1) <Satz> → <Subjekt><Prädikat><Objekt>
(2) <Subjekt> → Bruno
(3) <Prädikat> → knuffelt
(4) <Objekt> → Britta }

Ableitung :
<Satz> ⟶ <Subjekt><Prädikat><Objekt> ⟶ Bruno knuffelt Britta

L(G) = {Bruno knuffelt Britta}

(b)

G = { N, $\sum$, R, S }
N = { S }
$\sum$ = {a,b}

R =
{ (1) S → aS
(2) S → b }

Ableitung :
S ⟶ aS ⟶ aaS .... aaab

L(G) = $\{a^n b | n \in N\}$

(c)

G = { N, $\sum$, R, S }
N = { S }
$\sum$ = {(,)}

R =
{ (1) S → ()
(2) S → (S)
(3) S → SS }

Ableitung :
S ⟶ (S) .... ⟶ (SS)

L(G) = {Menge der regulären Klammerterme }

Die Menge der unterschiedlichen erzeugenden Grammatiken lässt sich nach der sogenannten Chomsky Hierarchie, benannt nach Noam Chomsky, in verschiedene Typen einteilen. Die für uns relevanten sind die sogenannten Typ 2 und Typ 3 Grammatiken.

Aufgaben

(a) Gegeben ist die Grammatik $G = \{\{S\}, \{a, b\}, \{S \to ab, S \to SS\}, S\}$. Geben Sie die Sprache an, die durch diese Grammatik erzeugt wird.

(b) Erläutern Sie die Typen der Chomsky Hierarchie an geeigneten Beispielen und weisen Sie nach, das es sich bei Aufgabe (a) um eine Grammatik vom Typ 3 (regulär) handelt, indem Sie eine äquivalente Grammatik erstellen.

(c) Gegeben ist die Grammatik

$G = \{\{S, A\}, \{a, b\}, \{S \to \epsilon, S \to aS, S \to bA, A \to bA, A \to \epsilon\}, S\}$.

Welche Wörter {aaabb, abb, aba}gehören zur Sprache, die durch die Grammatik erzeugt wird. Begründen Sie ihre Entscheidung durch entsprechende Ableitungsregeln.

Von welchem Typ ist die Grammatik nach Chomsky ?

Reguläre Grammatiken

Die am stärksten eingeschränkte Grammatik nach der Chomsky Hierarchie ist die sogennante reguläre Grammatik. Die Regeln haben die Form $A \rightarrow aB$ oder $A \rightarrow a$ oder $A \rightarrow Ba$. Sprachen die durch solche Grammatiken erzeugt werden, heißen *reguläre Sprachen*.

Der Zusammenhang zu Automaten stellt sich wie folgt dar. Akzeptoren erkennen reguläre Sprachen. Wird eine Sprache, als Menge von Wörtern erkannt, so lassen sich zu dieser Sprache Grammatikregeln finden.

Chomsky Hierarchie

Trotz der eindeutigen Definition einer Grammatik gibt es im Hinblick auf deren Ausprägung Unterschiede. Generell werden neben der Vereinbarung von Terminalsysbolen Regeln aufgestellt, die aus einer linken und rechten Seite ($r_1 \rightarrow r_2$) bestehen. Durch die Ersetzung der linken Seite durch die rechtsstehende Regel sind die Wörter, der durch die Grammatik bestimmten Sprache, ableitbar.

Noam Chomsky teilte die Grammatiken in Typen ein (Typ 0-3)

Typ 0		Jede Grammatik ist automatisch vom Typ 0. Es gibt keine Einschränkungen.
Typ 1	kontextsensitiv	Die Länge der Regel der linken Seite ist kürzer oder gleich rechts $r_1 \rightarrow r_2$ mit $\|r_1\| < \|r_2\|$.
Typ 2	kontextfrei	Die Grammatik ist vom Typ 1, r_1 ist eine einzelne Variable, es liegt links bei der Ersetzung kein notwendiger Kontext vor.
Typ 3	regulär	Die Grammatik ist vom Typ 2 und die rechte Seite der Regel besteht entweder aus einem Terminalzeichen oder aus einem Terminalzeichen gefolgt von einer Variablen (siehe oben).

In der Literatur wird darüber hinaus das leere Wort, wenn es Bestandteil der Sprache ist, wie folgt zugeordnet. $S \rightarrow \epsilon$ ist als Regel zulässig. Für kontextfreie und reguläre Grammatiken gelten darüber hinaus Regeln der Form $A \rightarrow \epsilon$ [4].
Die klare Zuordnung zu einer Sprache ermöglicht Typableitungen. So ist die für Compiler wichtige EBNF [5] Grundlage für das Scannen und Parsen von Termen. Das Programm eines einfachen Taschenrechners basiert im Endeffekt auf der Korrektheit der übergebenen Terme auf der Basis einer Grammatik.

Die Zuordnung einzelner Grammatiken zu den Typen erfolgt im Allgemeinen durch Automaten. Eine Zuordnung wäre wie folgt vorzunehmen :

Typ 0 $\longrightarrow$ Turingmaschine
Typ 1 $\longrightarrow$ Lineare Automaten
Typ 2 $\longrightarrow$ Kellerautomaten
Typ 3 $\longrightarrow$ endliche Automaten (Akzeptor)

Endlicher Automat und reguläre Grammatik

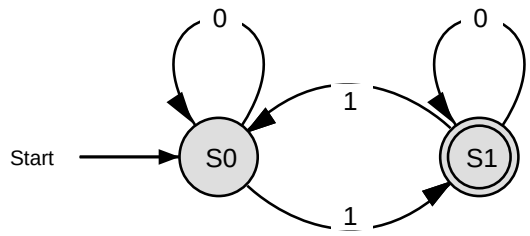
Der Nachweis der Regularität einer Grammatik kann über einen endlichen Automaten vorgenommen werden. Gibt es einen Akzeptor, der die Menge der Wörter einer Sprache welche durch eine Grammatik erzeugt wurde akzeptiert, so ist diese Grammatik regulär.

[4] vgl. Schöning S. 18
[5] Erweiterte Backus Naur Form

Beispiel

Gegeben sei ein Automat, welcher genau in einen akzeptierten Zustand übergeht, wenn das eingegebene Wort eine ungerade Anzahl von Einsen enthält.

Übergangsdiagramm



Festlegung der Zustände

X = {1,0}
Y = {1,0}

S_0 = gerade Anzahl von Einsen
S_1 = ungerade Anzahl von Einsen

Übergangstabelle

	0	1
S_0	S_0	S_1
S_1	S_1	S_0

Grammatik

Dem Eingabealphabet $X = \{1, 0\}$ entspricht die Menge der Terminalzeichen $\{0, 1\}$. Der Menge der Zustände $\{S_0, S_1\}$ entspricht die Menge der Nichtterminalzeichen $\{S, A\}$[6]. Daraus ergibt sich die folgende Grammatik.

Übergangstabelle

$$S_0 \rightarrow S_1$$
$$S_1 \rightarrow S_0$$
$$S_0 \rightarrow S_0$$
$$S_1 \rightarrow S_1$$

$$S \rightarrow 1A$$
$$A \rightarrow 1S|\epsilon$$
$$S \rightarrow 0S$$
$$A \rightarrow 0A$$

Aufgabe [7]

Gegeben sei die Grammatik :

$G = \{\{S, A, B\}, \{a, b\}, \{S \rightarrow aB, A \rightarrow aA \mid bB, B \rightarrow aA \mid bB \mid \epsilon\}, S\}$.

Ermitteln Sie den Zustandsgraphen des endlichen Automaten.

[6] vgl. Baumann S.222
[7] vgl. Baumann S.223

Kontextfreie Grammatiken

Als Standardbeispiel für eine kontextfreie Sprache, welche auf einer kontextfreien Grammatik basiert, gilt :

$$L(G) = \{a^n b^n | n \geq 1\} = \{aabb, ..., aaaabbbb, ..\}.$$

Will man nachweisen, dass die Sprache nicht regulär ist geht man wie folgt vor. Man nimmt an, die Sprache sei *regulär*. Wenn dem so wäre, gäbe es einen endlichen Automaten, der alle Wörter der Sprache akzeptiert. Dieser Automat hat dann eine endliche Anzahl von Zuständen n. Wählt man nun ein Wort aus der Sprache, welches länger ist als n, pumpt das Wort also auf, so muss der Automat eine Schleife aufweisen, um eben dieses Wort zu erkennen. Würde eine solche Schleife aber im Bereich der a's oder b's auftreten, würde der Automat auch Wörter akzeptieren, die *nicht* zur Sprache gehören. Insgesamt akzeptiert der Automat dann alle Wörter der regulären Sprache *und* Wörter, die nicht zur Sprache gehören. Dies wäre ein Widerspruch !

Weitaus formaler und mathematisch abstrakt wird dieser Zusammenhang im sogenannten *Pumping Lemma* (Hilfssatz) formuliert. Eine gute Erklärung findet sich hierzu [8] in der Literatur.

Klammerterme

Zum Nachweis der Kontextfreiheit einer Grammatik bedient man sich der sogenannten Kellerautomaten. Das Grundprinzip dieser Automaten basiert auf dem Stack. Für eine mögliche Implementierung eines Kellerautomaten werden hier alle Attribute und Methoden durch die Klasse Stack zur Verfügung gestellt.

Eine typische kontextfreie Grammatik ist die der bereits weiter oben beschriebenen regulären Klammerterme.

$G = \{\{S\}, \{(,)\}, \{S \to (), S \to (S), S \to SS\}, S\}$

Der Nachweis der Kontextfreiheit erfolgt hier über einen Kellerautomaten. Das Grundprinzip stellt sich wie folgt dar :

- Lege ein Startsysmbol auf den Stack
- Lese das Eingabezeichen - bestimme die (mit einem Terminalsymbol)
- beginnende, passende Ersetzungsregel
- Lese diese aus und lege Sie anschließend auf den Stack
- Stimmt das Eingabezeichen mit dem Kopf des Stack überein incrementiere
- den Eingabeindex und lösche den Kopf.
- $\rightsquigarrow$ ..... usw.

[8] vgl. Schoening S. 39

Eine konkrete Belegung für einen korrekten Klammerterm wäre (()()()) $\rightsquigarrow$

Algorithmus zum Nachweis der Korrektheit des Terms [9]

Eingabe	Stack	Regel
	S#	S $\rightarrow (S)$
(()()())	(S)#	S $\rightarrow SS$
()()())	SS)#	S $\rightarrow SS$
()()())	SSS)#	S $\rightarrow ()$
()()())	()SS)#	S $\rightarrow ()$
()())	()S)#	S $\rightarrow ()$
())	())#	
		... $\rightsquigarrow$ korrekter Term

Aufgabe (Z*)

Stellen Sie die Funktionsweise des Kellerautomaten für 2 weitere Beispiele (mindestens 6 Schritte) dar. Implementieren Sie einen Kellerautomaten zur Untersuchung von Klammertermen. Es reicht aus, wenn Terme der Form (()()()), (()((()))) erkannt werden.

Stützen Sie sich dabei auf die Klasse Stack, von der entsprechende Methoden geerbt werden sollen.

[9] vgl. Baumann S. 242

Alan M. Turing
(1912 - 1954)

Turingmaschine

Die Zugehörigkeit eines Wortes zu einer Sprache, deren Grammatik nach Chomsky Typ 0 entspricht kann über eine *Turingmaschine* erfolgen. Die Turingmaschine hat aber einen weitaus größeren Anwendungsbereich.

1936 wurde durch Alan Turing das Modell einer Rechenmaschine vorgestellt. Mit ihr sollte es möglich sein, ausgehend von einer einfachen Konzeption alles zu berechnen, was berechenbar ist bzw. wofür eine Lösungsstrategie existiert [Churchsche These]. Damit hätte die Turingmaschine die gleiche Funktionalität wie der komplizierteste Computer der Welt. Darüber hinaus ist das Konzept der Turingmaschine denkbar einfach.

Konzept

Ein beliebig langes Band wird mit Symbolen gefüllt, die einem Bandalphabet entstammen (z.B. a oder b, 1 oder 0). Ein Schreib -und Leseknopf hat die Aufgabe, ein Zeichen vom Band zu lesen (*input*) oder ein Zeichen auf das Band zu schreiben (output). Der Schreib - bzw. Leseknopf kann bei den meisten Modellen 3 'Bewegungen 'ausführen, 1 Schritt nach rechts (R), ein Schritt nach links (L), Stop (S). Die jeweilige Bewegung kann zu einem Zustandübergang führen. Insgesamt kann die Turingmaschine ausgehend von einem definierten Startzustand unendlich viele innere Zustände annehmen.
Die Programmierung der Maschine erfolgt im Kontext des zu beschreibenden Problems durch eine Menge an vorgegebenen Regeln. Diese Regeln haben einen vorgegebenen Aufbau der Form (Zustand, Input $\mapsto$ Output, Bewegung, Neuzustand)

Beispiele für die Anwendung der Turingmaschine

Eines der bekanntesten Beispiele für die Anwendung der Turingmaschine ist das durch Tibor Rado [10] aufgestellte Problem der später so genannten *Fleißigen Biber*. Es handelt sich hierbei um eine Funktion die einer festen Anzahl von inneren Zuständen eine Menge von Einsen zuordnet, welche eine stoppende Turingmaschine auf ein leeres Band schreiben kann.

n = 1 (Anzahl = 1)

Zustand	Band				Regel
	0	0	0	0	$1,0 \mapsto 1,1,S$
		↑			
Z_1	0	1	0	0	

[10] vgl. W. Urban

n = 2 (Anzahl = 2)

Zustand	Band								Regel
Z_1	0	0	0	0 ↑	0	0	0	0	$1,0 \mapsto 2,1,L$ $1,1 \mapsto 2,1,R$
Z_2	0	0	0 ↑	1	0	0	0	0	$2,0 \mapsto 1,1,R$ $2,1 \mapsto *,1,S$
Z_1	0	0	1	1 ↑	0	0	0	0	
Z_2	0	0	1	1	0 ↑	0	0	0	
Z_1	0	0	1	1	1	0 ↑	0	0	
Z_2	0	0	1	1	1 ↑	1	0	0	

Erhöht man die Anzahl der Zustände wächst die Anzahl der Schritte. Bereits für n = 5 zeichnet sich ein exponentielles Wachstum ab. Der letzte Biber wurde in Deutschland 1984 mit 4089 Einsen entdeckt. Die dazu notwendigen ca. 47 Mio Rechenschritte wurden nach 3 Tagen auf einem Hochleistungscomputer simuliert. Die Anzahl der notwendigen Schritte wächst so stark, dass heute allgemein eine *Nichtberechenbarkeit* für das Problem angenommen wird.

Im Internet sind weitere Beispiele abrufbar, das grundlegende Prinzip ist jeweils identisch. Um eine Turingmaschine zu implementieren, ist es naheliegend, die Maschine als abstrakten Datentyp bzw. als Objekt aufzufassen. Die Attribute sind dabei eine Bandbelegung, ein Startzustand, eine initiale Bandposition und eine Menge von Regeln, welche zum Beipiel über eine Liste verwaltet werden können. Die im Namensraum durch die Instanzierung gebundenen Methoden können eine Suchmethode sein, welche die passende Regel findet und eine Methode welche darauf aufbauend mit der entsprechenden Regel den Zustandsübergang realisiert und die Bandbelegung entsprechend ändert.

Aufgaben

Implementieren Sie eine *Klasse turing* mit der es möglich ist, eine Turingmaschine zu simulieren.

Überprüfen Sie die Funktionalität ihrer Klasse an einem eigenen Beispiel.

Tibor Rado und Shen Lin fanden eine Turingmaschine für n = 3. Simulieren Sie den Ablauf, der durch folgende Regeln bestimmt wird.

R =
$[\ 1,0 \mapsto 2,1,R;\ 1,1 \mapsto 3,1,L;\ 2,0 \mapsto 1,1,L;\ 2,1 \mapsto 2,1,R;\ 3,0 \mapsto 2,1,L;\ 3,1 \mapsto 3,1,S\ ]$

Klasse Turingmaschine

```
1
2  import time
3
4  class turing :
5
6     # Turingmaschine zur Simulation der fleissigen Biber nach Tibor Rado
7     # copyright R. Herpel@<g-ymnasium.de> Alle Rechte vorbehalten
8
9     def __init__(self) :
10
11       self.belegung = ''
12       for i in range (0,30) : self.belegung += '0'
13
14       self.ausgabe = ""
15       self.zustand = '1'
16       self.band_pos = 4
17       self.aktueller_befehl = '␣␣␣␣␣'
18
19       # Zustand - Input / Output Bewegung Folgezustand
20       self.regelwerk = [('101L2'),('111R2'),('201R1'),('211S_')]
21       self.prglaenge = len (self.regelwerk)
22
23
24    def suche (self, zustand, input):
25       for i in range (0, self.prglaenge) :
26          if self.regelwerk [i] [0] == zustand and self.regelwerk [i] [1] == input :
27            return self.regelwerk [i]
28
29
30    def zyklus (self) :
31
32       while self.aktueller_befehl [3] != 'S':
33         aktuelles_zeichen = self.belegung [self.band_pos]
34         self.aktueller_befehl = self.suche (self.zustand, aktuelles_zeichen)
35
36         if aktuelles_zeichen == self.aktueller_befehl [1] and \
37           self.zustand == self.aktueller_befehl [0] :
38
39            self.ausgabe = self.belegung [:self.band_pos] \
40                        + self.aktueller_befehl [2] \
41                        + self.belegung [self.band_pos +1:]
42            if  self.aktueller_befehl [3] == 'R': self.band_pos += 1
43            elif self.aktueller_befehl [3] == 'L': self.band_pos -= 1
44            self.zustand = self.aktueller_befehl [4]
45
46            time.sleep (0.5)
47            self.belegung = self.ausgabe
48            print str(self.aktueller_befehl) + '\t->\t'+ str (self.ausgabe)
49
50 if __name__=="__main__":
51
52   test = turing ()
53   test.zyklus ()
```

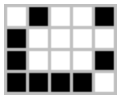

John Horton Conway

Das Spiel des Lebens (Game of Life)

Die von John von Neumann 1952 gestellte Frage, ob Automaten zur Selbstreproduktion fähig seien, kann als Ausgangspunkt für John Horton Conways 2- dimensionale zelluläre Automaten gesehen werden. Der Automat ist eine Ebene, die aus lauter einzelnen Zellen (Quadrate) aufgebaut ist. Jede dieser Zellen stellt einen endlichen Automaten dar, der 2 Zustände annehmen kann, lebend oder tot. Das Überleben der Zelle hängt entscheidend vom Zustand der Nachbarn ab. J. Conway stellt hier vergleichsweise einfache Regeln auf.

Eine Zelle stirbt an Vereinsamung wenn sie weniger als 2 Nachbarn besitzt, an Überbevölkerung, wenn sie mehr als drei Nachbarn besitzt und wird geboren, wenn 3 Nachbarn leben. Mit jedem Schritt wird im gesamten zellulären Automaten über Leben und Tod entschieden und die Welt neu generiert.

Die dadurch entstehenden verschiedensten Gebilde (statisch, oszillierend oder zyklisch) beschäftigten Scharen von Informatikern. Raumschiffe und Gleiter, wie nachfolgend abgebildet, stellen dabei zum Beispiel eine immer wiederkehrende Gruppe von Objekten dar.

Eine Abwandlung der Regeln erweiterte das Betätigungsfeld und läßt auch Anwendungen außerhalb der Informatik zu.

Die Implementierung des Problems ist überschaubar. Neben der Generierung einzelner ansprechbarer Felder die in den Zustand 1 bzw. 0 gesetzt werden können, ist es notwendig nach jedem Schritt *alle* Spielfelder neu zu bewerten. Dazu ist es notwendig die 8 Felder um die Zelle zu prüfen und entsprechend der Regeln eine Entscheidung in Bezug auf das Überleben zu treffen. Um das eigentliche Problem zu implementieren sollten Regler und Menüs entfallen. Ein Anfangszustand kann zufällig generiert werden.

Aufgaben

Implementieren und spezifizieren Sie eine *Klasse Leben* mit der es möglich ist, den zellulären Automaten zu simulieren. Erzeugen Sie eine Kundenklasse *Objekt* die entsprechende Objekte für eine mögliche Untersuchung zur Verfügung stellt.

Finden Sie Objekte und Regeln und erweitern Sie die Implementierung.

Klasse Leben

```
from Tkinter import *
from random import *

class Leben:

   def __init__(self, faktor, breite,hoehe):

      self.fenster = Tk()
      self.fenster.title("Game␣of␣Life")
      self.b = breite
      self.h = hoehe

      self.listeΓ = ((self.b*self.h)+1)*[0]
      self.listeΓ = ((self.b*self.h)+1)*[0]

      self.cv = Canvas(self.fenster,width=faktor*self.b+2,height=faktor*self.h+2)
      self.cv.pack()

      button = Button(self.fenster, text="GOL", command=self.go)
      button.pack (side="left")

      button = Button(self.fenster, text="Exit", command=self.aus)
      button.pack (side="left")

      self.za = ((self.b*self.h)+1)*[0]

      for x in range(1,self.b+1):
         for y in range(1,self.h+1):
            self.za [(y-1)*self.b+x] = self.cv.create_rectangle(
               faktor*(x-1)+2, faktor*(y-1)+2, faktor*x+2, faktor*y+2,
               fill = "white",outline="black")

   def set_za (self, x, y, boolean, color):

      index  = (y-1)* self.b + x
      self.cv.itemconfigure(self.za [index], fill = color)
      self.listeΓ [index] = boolean
      self.listeΓ [index] = boolean

   def go (self) :
      while 1:
       for x in range (2,self.b):
        for y in range (2,self.h):
          # Pruefung der Faelle
          summe = 0
          for i in range (-2,1,1) :
            for j in range (-1,2,1) :
              if i == - 1 and j == 0 : pass
              else               : summe += self.listeΓ [(y+i)*self.b + x + j]

          if  summe == 3 : self.listeΓ[(y-1)*self.b + x] = 1
          elif summe > 3 : self.listeΓ[(y-1)*self.b + x] = 0
          elif summe < 2 : self.listeΓ[(y-1)*self.b + x] = 0
```

```
57
58          for x in range (2,self.b):
59            for y in range (2,self.h):
60             if  self.listeΓ [(y-1)* self.b + x] == 0 : self.set_za (x,y,0,"white")
61             elif self.listeΓ [(y-1)* self.b + x] == 1 : self.set_za (x,y,1,"black")
62          self.cv.update()
63
64      def aus (self): self.fenster.destroy ()
65
66  if __name__=="__main__":
67
68      b = 120
69      h = 90
70      m = Leben(10,b,h)
71      for i in range (2, randint (2,b*h)):
72         x = randint (1,b)
73         y = randint (1,h)
74         if x in (1,b) or y in (1,h) : pass
75         else      : m.set_za (x,y,1,"black")
76
77      m.fenster.mainloop()
```

Klasse Objekte

```
4  from Life import *
5
6  class GOL (Life):
7
8      def __init__(self, breite, hoehe, factor):
9
10        Life.__init__(self, factor, breite, hoehe)
11        self.breite = breite
12        self.hoehe = hoehe
13
14
15     def figur (self,x, y, liste):
16
17        for i in range (0, len(liste)):
18          self.set_za (x+liste[i][0],y+liste[i][1],1,"black")
19
20
21 if __name__=="__main__":
22
23     test = GOL(100,30,10)
24     """
25     # Raumschiffe und Gleiter
26     test.figur (1,1,[(2,1),(3,2),(3,3),(2,3),(1,3)]) # Gleiter
```

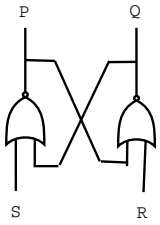
Technische Umsetzung der Automatentheorie

Speicherelemente

Automaten haben die Aufgabe Zustandsübergänge zu realisieren. Der aktuelle Zustand muss zu diesem Zweck zwischengespeichert werden, wobei die Speicherung über die Rückkopplung von Ausgangssignalen erfolgt. Die Speicherelemente werden als *Flipflops* bezeichnet, es handelt sich hierbei um Schaltelemente, die über 2 stabile Zustände verfügen, weshalb auch die Bezeichnung *bistabiles Kippelement* gebräuchlich ist. Eine Änderung der Zustände erfolgt über die Eingangssignale. Am Beispiel verschiedener *Flipflops* wird diese Funktionalität verdeutlicht.

Speicherelemente

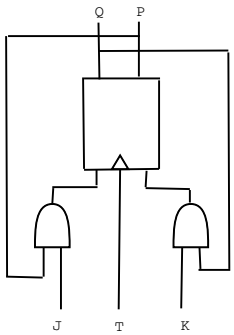
- Reset - Set - Flipflop



S	R	Q_{alt}	Q	
0	0	1	1	
0	0	0	0	$\rightsquigarrow$ Halten
0	1	1	0	
0	1	0	0	$\rightsquigarrow$ Reset
1	0	1	1	
1	0	0	1	$\rightsquigarrow$ Set
1	1	?	?	

Für S = R = 0 bleibt der gespeicherte Zustand erhalten, für S = 1 und R = 0 wird der Speicher auf 1 gesetzt, für S = 0 und R = 1 erfolgt eine Rücksetzung auf 0. Soll ein Übergang von S = R = 0 nach S = R = 1 erfolgen, sind je nach Reihenfolge der gesetzten Eingangssignale mehrere Ausgänge möglich. Es kommt im übertragenen Sinn zu einem Wettlauf *race condition*. Welcher stabile Zustand im Endeffekt erreicht wird, bleibt offen, man bezeichnet dies als *metastabiles* Verhalten. Aus diesem Grund ist das Setzen der Eingangssignale S = R = 1 verboten.

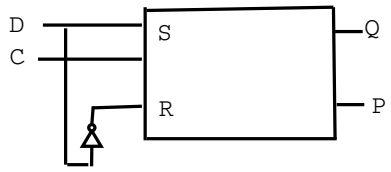
- J - K - Flipflop



S	R	Q_{alt}	Q	
0	0	1	1	
0	0	0	0	$\rightsquigarrow$ Halten
0	1	1	0	
0	1	0	0	$\rightsquigarrow$ Reset
1	0	1	1	
1	0	0	1	$\rightsquigarrow$ Set
1	1	1	0	
1	1	0	1	

Bis auf die Eingabe J = K = 1 entspricht die Funktionalität des JK Flipflops der des RS Flipflops und stimmt vom Aufbau bis auf den Taktgeber und die Rückkopplungen überein. Diese bewirken eine Verhinderung des metastabilen Verhaltens. Bei der Aufstellung der Synthesetabelle ergeben sich 4 *don't care* Zustände. Aus diesen Gründen wird der JK - Flipflop als häufiges Speicherelement für Automaten verwendet.

- D - Flipflop



C	D	Q_{alt}	Q	
0	0	0	0	$\rightsquigarrow$ Durchlassen
0	0	1	0	
0	1	0	1	
0	1	1	1	
1	0	0	0	$\rightsquigarrow$ Halten
1	0	1	1	
1	1	0	0	
1	1	1	1	

Auch für das D *Delay* - Flipflop bildet das RS Flipflop die Grundlage. Der verbotene Zustand S = R = 1 wird durch die Invertierung des Eingabesignals ausgeschlossen. Damit ist nur noch der Eingang D relevant. Die Schaltung verfügt darüber hinaus über einen Taktgeber C. Wird hier eine 1 geschaltet, werden alle Eingaben auf D *durchgelassen*. Man spricht vom D - Latch. Im Fall C = 0 wird der Altzustand gehalten, der Wert, welcher sich im Speicher befindet, übernommen. Die Verzögerung bis zum Erreichen des Wertes C = 1 gibt dem Schaltelement den Namen.

Literaturverzeichnis

(1) Uwe Schöning - *Theoretische Informatik - kurzgefasst* - Spektrum 2000

(2) Rüdiger Baumann - *Informatik für die SEK II* - Klett 1993

(3) Wolfgang Urban - *http://www.hib-wien.at/leute/wurban*

(4) *http://www.matheprisma.uni-wuppertal.de/Module/Turing/*