„Theorie“ bedeutet meist, dass die Bestandteile und Eigenschaften von Systemen auf das Elementare reduziert werden, um deren Prinzipien, Zusammenhänge, Möglichkeiten und Grenzen möglichst einfach und klar zu beschreiben und zu verstehen. Das gilt auch für die Theoretische Informatik. Zusätzlich kommt hier allerdings hinzu, dass theoretische Erkenntnisse direkte Anwendungen finden, sodass die in diesem Buch behandelten Themen an den Universitäten weitgehend in der praktischen Informatik angesiedelt sind. Es gilt in diesem Bereich also besonders der schöne Spruch *„Nichts ist praktischer als eine gute Theorie“*.

In der Theoretischen Informatik werden zwei Modellklassen benutzt:

- *Maschinenmodelle* – die Automaten – beschreiben das Verhalten zustandsabhängiger Systeme, die auf Eingaben unterschiedlich reagieren können, je nachdem, in welchen *Zustand* sie von den vorhergehenden Eingaben versetzt worden sind. Diese Modelle gehen auf die Arbeiten von *Alan Turing* (1912-1954) zurück.
- *Grammatiken* beschreiben die Struktur *formaler Sprachen*, die von den Automaten akzeptiert werden. Da sich Computer als Automaten beschreiben lassen, beschäftigt sich dieser Bereich u. a. mit der Entwicklung und Übersetzung von Computersprachen. Die Klassifizierung formaler Sprachen stammt von *Noam Chomsky* (geb. 1928).

Wir wollen in diesem Buch die Modelle der Automaten und formalen Sprachen kennen lernen und auf unterschiedliche Gebiete anwenden. Dabei gehen wir anfangs von einfachen Anwendungsfällen in anderen Bereichen aus, konzentrieren uns dann aber zunehmend auf die Informatik selbst. Die dabei gewonnenen Kenntnisse und Erfahrungen ermöglichen es später, auch zu Fragen der Berechen- und Entscheidbarkeit Stellung zu nehmen.

# 1 Endliche Automaten mit Ausgabe

## *1.1 Autokorrektur und Smileys* ☺

In vielen Textverarbeitungsprogrammen kann man eine *Autokorrektur* so einstellen, dass schon während der Eingabe typische Fehler wie Buchstabendreher („*dre*“ statt „*der*“) oder nach alter Rechtschreibung eingegebene Worte („*daß*“ statt „*dass*“) korrigiert werden. Wir wollen uns stattdessen die so genannten *Smileys* vornehmen, also Zeichenfolgen, die z. B. „Lachgesichter“ darstellen sollen. Da diese Darstellungen nur sehr rudimentäre „Gesichter“ erzeugen, werden sie durch deutlichere Zeichen ersetzt:

| Zeichenfolge ... | wird umgewandelt in ... |
|---|---|
| :-) | ☺ |
| :) | ☺ |
| :-( | ☹ |
| :( | ☹ |

Wir haben es hier mit einem typischen Einsatzgebiet endlicher Automaten zu tun – der *Mustererkennung*. Innerhalb einer Zeichenfolge – hier: dem Strom der Eingabezeichen – sollen bestimmte Teilfolgen erkannt und ggf. ausgetauscht werden. Diese Aufgabe soll durch eine „Maschine" erledigt werden, die

- den Eingabezeichenstrom liest und – bis auf die Smileys – unverändert wieder ausgibt,
- erkennt, ob die Eingabefolgen gültige Smileys darstellen,
- und ggf. die Smiley-Zeichenfolgen durch die richtigen Zeichen ersetzt.

Das System nennen wir einen *Smiley-Erkenner*.

Die von der gesuchten Maschine – einem *Automaten* – gelesenen Zeichen stellen also für diese *Eingabezeichen* dar. Da zu einem Smiley mehrere Zeichen gehören (mindestens zwei), kann das System nicht aufgrund des gerade gelesenen Eingabezeichens entscheiden, ob ein Smiley oder ein anderes Zeichen vorliegt. Vielmehr hängt diese Entscheidung vom gerade gelesenen und den vorausgegangenen Zeichen ab. Die gesuchte Maschine muss sich also „merken" können, welche für das Problem relevanten Zeichen schon gelesen wurden.

Wir können dieses Verhalten so interpretieren, dass der Automat von den gelesenen Eingabezeichen in unterschiedliche *Zustände* überführt wird. Befindet sich der Automat in einem bestimmten Zustand, so wird er von dem nächsten gelesenen Zeichen in einen von mehreren Folgezuständen überführt. Der Zustand gibt hier also an, welche für das Problem relevanten Zeichen bisher empfangen wurden. Nach einigen Eingabezeichen kann entschieden werden, ob eine der gesuchten Eingabefolgen vorliegt. Entsprechend wird ggf. ein Smiley ausgegeben und/oder durch Übergang in den Anfangszustand oder den richtigen Folgezustand weiter gesucht.

Für nicht zu umfangreiche Probleme bildet die Form des *Zustandsdiagramms* oder *Transitionsgraphen* eine übersichtliche und für das weitere Vorgehen hilfreiche Art, dieses Verhalten aufzuschreiben (und damit exakt die Arbeitsweise des Automaten festzulegen). Jeder Zustand wird benannt, durch einen Kreis symbolisiert und bildet so einen *Knoten* des Zustandsdiagramms. Der *Anfangszustand* wird mit einem kleinen Pfeil zusätzlich gekennzeichnet. *Endzustände* erkennt man an einem Doppelkreis – wenn sie denn auftreten. Typische Endzustände sind *Fehlerzustände*, denn nach Fehlern sollte nicht einfach weitergemacht werden, als ob nichts geschehen wäre.

Von jedem Zustand aus führen Pfeile - die *Kanten* des Transitionsgraphen - zu den möglichen Folgezuständen. Diese Kanten werden jeweils mit den Eingabezeichen beschriftet, die den entsprechenden Übergang auslösen. Hierbei sind Mehrfachnennungen möglich, falls unterschiedliche Eingabezeichen das System in denselben Folgezustand überführen (die Zeichen werden dann mit „*v*“ (oder) verknüpft). Produziert das System auch Ausgabezeichen, so müssen diese ebenfalls an der entsprechenden Kante neben dem Eingabezeichen notiert werden.

Um unseren Automaten nicht zu kompliziert werden zu lassen, wollen wir als Eingabezeichen nur die in Smileys vorkommenden Zeichen und zusätzlich das „*x*“ zulassen, das für alle anderen Zeichen steht. Diese Zeichen zusammen bilden das *Eingabealphabet* E des Automaten, das als Zeichenmenge von geschweiften Klammern (Mengenklammern) eingerahmt wird.

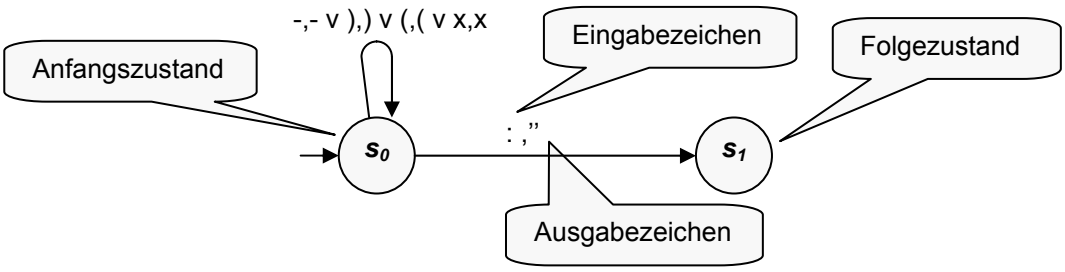
$$E = \{ : , - , ) , ( , x \}$$

Nennen wir den Anfangszustand unseres Automaten $s_0$ (*s* von *state*), so müssen in unserem Beispiel von diesem Zustand fünf Kanten ausgehen, da es fünf verschiedene Eingabezeichen gibt. „Interessant“ ist von diesen Zeichen allerdings nur der Doppelpunkt, denn alle Smileys beginnen mit diesem. Entsprechend lassen alle anderen Eingabezeichen den Automaten im Anfangszustand, was durch eine Kante angezeigt wird, die von diesem in diesen wieder zurückführt. Nur der Doppelpunkt überführt die Maschine in einen anderen Zustand, den wir $s_1$ nennen wollen.

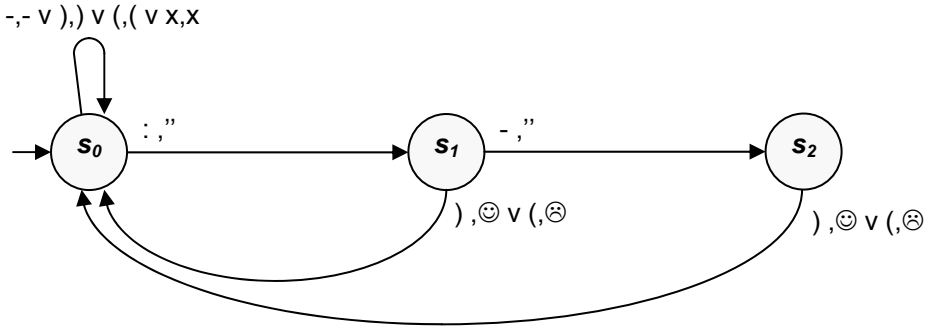
Wie sieht es nun mit den *Ausgabezeichen* aus? Der Automat soll ja Nicht-Smiley-Zeichen unverändert ausgeben. Er reproduziert also einfach „uninteressante“ Zeichen. Ein- und Ausgabezeichen sind gleich. Nur beim Doppelpunkt darf gar nichts ausgegeben werden, denn der Automat muss warten, ob ein vollständiger Smiley kommt – oder nicht. Dieses Verhalten wird durch das „Ausgabezeichen“ ” (nichts: zwei Hochkommas) erreicht. Somit haben wir auch das *Ausgabealphabet* A gefunden, das aus allen Eingabezeichen, den Smileys sowie dem Zeichen für „nichts“ besteht.

$$A = \{ : , - , ) , ( , x , ☺ , ☹ , ” \}$$

Der erste Teil des Zustandsdiagramms sieht dann so aus:



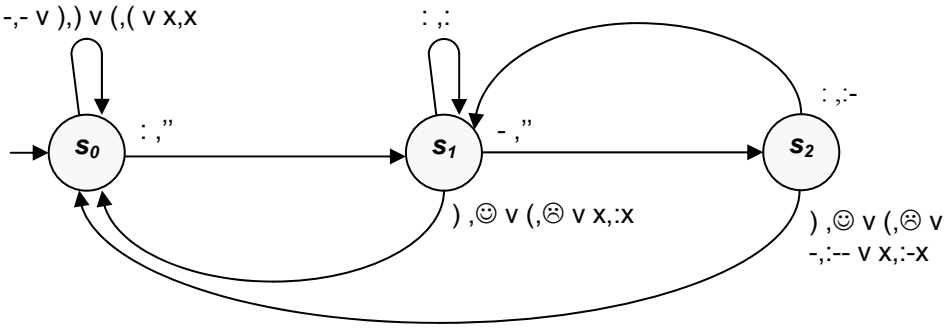
Es ist meist eine gute Idee, als Nächstes die Abläufe bei den „interessanten“ Eingabefolgen zu definieren, hier also für Smiley-Zeichenfolgen. Die beginnen alle mit dem schon bekannten Doppelpunkt. Wir setzen sie danach einfach fort und wechseln bei jedem „richtigen“ Zeichen in den nächsten Zustand. (In diesem Fall folgt nur einer: $s_2$) Ist eine Smiley-Zeichenfolge vollständig, dann kehren wir unter Ausgabe des richtigen Smiley-Zeichens in den Anfangszustand zurück.




Wir haben damit auch die *Zustandsmenge* S des Automaten gefunden: $S = \{ s_0 , s_1 , s_2 \}$

Und was geschieht weiter?

Für jeden Zustand und jedes Eingabezeichen muss definiert werden, in welchen Folgezustand der Automat übergeht und welches Ausgabezeichen er dabei produziert. Damit ist die Beschreibung eines Automaten durch einen Transitionsgraphen *vollständig*: es können keine undefinierten Situationen auftreten. In unserem Fall müssen wir also vereinbaren, was geschehen soll, wenn die Smiley-Zeichenfolgen unvollständig oder falsch eingegeben werden, wenn sie gar keine Smiley-Zeichenfolgen sind. In diesen Fällen muss der Automat die bisher nicht ausgegebenen Zeichen (den Doppelpunkt und ggf. die „Nase“ des Smileys) zusammen mit dem neuen „falschen“ Zeichen ausgeben. Dabei kehrt er in den Anfangszustand zurück – es sei denn, das neue Zeichen ist wieder ein Doppelpunkt: dann führt die Kante in den Zustand $s_1$, denn es könnte ja noch ein Smiley folgen!



Was haben wir damit gewonnen?

Zumindest eine vollständige Beschreibung der Problemlösung, die wir ohne das Hilfsmittel des Zustandsgraphen rein sprachlich nur sehr viel umständlicher finden würden. Vor allem aber können wir durch reines Abzählen der Kanten feststellen, ob noch undefinierte Situationen vorhanden sind: geht von jedem Zustand für jedes Eingabezeichen eine Kante weg, dann gibt es keine Ungewissheiten mehr! (Ob die gefundene Lösung richtig ist, können wir damit allerdings nicht feststellen.) Im Folgenden werden wir dann auch sehen, dass wir mit dem Transitionsgraphen schon über die vollständige Lösung verfügen, also danach rein schematisch eine entsprechend arbeitende Maschine (hier: ein Programm) entwickeln können.

Der Genauigkeit halber wollen wir anmerken, dass es sich bei den beschriebenen Automaten um *Mealy-Automaten* handelt. Diese Maschinen produzieren bei jedem Arbeitsschritt genau ein Ausgabezeichen. Deshalb benötigten wir auch das „leere“ Ausgabezeichen. Eigentlich müssen wir dann für die „verzögert“ ausgegebenen Zeichenfolgen (z. B. „*:-x*“) besondere einzelne Ausgabezeichen einführen, die erst später in einem zweiten Schritt (von einer anderen Maschine) in die „richtigen“ Zeichenfolgen umgesetzt werden. Da dies immer problemlos möglich ist, sollen unsere Automaten gleich Ausgabe-Zeichenfolgen produzieren, die eventuell leer sein können.