Intensivkurs II
Formale Methoden & Programmierung

IIG, Abt. Kognitionswissenschaft
Anna Strasser
Martin Brösamle

Version 1.0.4; 19. November 2003

Bemerkungen vorneweg

Inhalt des Dokuments

Das Skript deckt den ersten Teil der Veranstaltung „Formale Methoden und Programmierung“ im Fachbereich Kongnitionswissenschaft ab. Vor allem geht es um die Auffrischung und Erweiterung mathematischer Grundkenntnisse, sowie die Vermittlung wichtiger Hintergründe aus dem Bereich Informatik und Programmierung.
Im zweiten Teil der Veranstaltung sollen auf diese Basis aufbauend praktische Programmiertechniken erlernt werden. Hierzu wird die Sprache LISP verwendet werden. Für den zweiten Teil der Veranstaltug wird auf ein Skript verzichtet, da für praktisches Programmieren sowie die Programmiersprache LISP ausreichend Material zur Verfügung steht.

Wie liest man dieses Skript?

Zu jedem größeren Themenkomplex gibt es eine Zusammenfassung, in der die wichtigsten Zusammenhänge noch einmal kondensiert dargestellt werden. Auf der einen Seite dienen die Zusammenfassungen der Wiederholung und damit Festigung des soeben gelesenen. Auf der anderen Seite bieten sie demjenigen einen schnellen Überblick, der bereits Vorkenntnisse mitbringt und nur Teile nachlesen möchte.
Generell wird so weit wie möglich auf formale und natürlichsprachliche Präzision geachtet. Einschränkend ist hier zu bemerken, dass bei etlichen Definitionen auf den kompletten formalen Apparat verzichtet wurde, wenn dieser für das Verständnis nicht von Nöten ist. Der interessierte Leser sei für formal korrekte Definitionen auf die entsprechende Fachliteratur verwiesen, wie sie im Literaturverzeichnis angegeben ist.

Inhaltsverzeichnis

►►► Woche 1 ◄◄◄

1 Mathematische Grundlagen und formale Arbeitsweise

1.1 Einige Sprechweisen

Es gibt einige Sprechweisen, die in der Mathematik in einer anderen Weise verwendet werden, als in der Alltagssprache. Dadurch kann es leicht zu Mißverständnissen kommen. So meint etwa das Wort *ein* in der mathematischen Ausdrucksweise *mindestens ein*; ein Satz der Art „Es existiert ein ...“ bedeutet „Es existiert mindestens ein ...“. Der alltagsübliche Gebrauch von *ein* wird in der Mathematik mit *genau ein* ausgedrückt. Die in der Mathematik häufig benuntzte Ausdrucksweise *genau dann, wenn* ist gleichbedeutend mit *dann und nur dann*. Eine Aussage der Art „A gilt genau dann, wenn B gilt“ bedeutet, dass die Aussage B in allen den Fällen gilt, in denen A gilt und in keinen weiteren Fällen. Gilt A lässt sich daraus schließen, dass auch B gilt und umgekehrt.

1.2 Mengen

1.2.1 Intuition

Dieser Abschnitt stellt Mengen als eine Möglichkeit vor, die Zusammenfassung von Dingen zu beschreiben. Dabei ist diese Ausdrucksweise für die Arbeitsweise der Mathematik besonders geeignet und wird auch für die später eingeführte Programmierung benötigt. Eine formale Einführung des Mengenbegriffs würde an dieser Stelle zu weit führen und wird für unsere Zwecke auch nicht benötigt. Weniger komplexe Definitionen enthalten sehr leicht versteckte Zirkularitäten oder Inkonsistenzen, so dass wir uns mit einer intuitiven Vorstellung begnügen wollen.
Eine Menge ist durch die Elemente, die sie enthält, bestimmt. Im weiteren Verlauf wird öfters von Elementen und Objekten die Rede sein. Es ist nicht ohne weiteres möglich, eine saubere Definition zu geben. Von *Element* wird die Rede sein, wenn es um ein Element einer explizit angegebenen Menge geht. Dagegen wird der Ausdruck *Objekt* dann verwendet, wenn es sich um ein Element einer nicht weiter angegebenen Menge handelt; also die Menge, in der es enthalten ist, keine Rolle spielt.

1.2.2 Schreibweisen für Mengen

Es gibt verschiedene Schreibweisen, um konkrete Mengen anzugeben. Wir beginnen mit der üblichen Schreibweise, bei der die Elemente der Menge explizit aufgezählt werden und in geschweifte Klammern eingeschlossen werden.
Beispielsweise ist

$$\{1, 2, 3, 4\}$$

die Menge mit den Elementen 1, 2, 3, 4. Dabei kann die (informale) Punktschreibweise

$$\{-4, -3, \ldots, 20\}$$

benutzt werden, um zu verdeutlichen, dass alle ganzen Zahlen von -4 bis 20 in der Menge enthalten sein sollen. Die Punkte kennzeichnen eine sinngemäße Fortsetzung der begonnenen Folge. Eine besondere Menge ist die leere Menge {}, auch mit $\emptyset$ bezeichnet. Sie enthält keine Elemente.

Bemerkung: *Ein Element kann nicht mehrmals in einer Menge vorkommen. So ist beispielsweise* $\{1.5, 1.8, 1.0, 1.8\}$ *keine wohlgeformte Schreibweise für eine Menge, da das Element 1.8 doppelt aufgeführt ist.*

Einige konkrete Mengen, die recht häufig benutzt werden, sind die Menge der natürlichen Zahlen, bezeichnet mit $\mathbb{N}$, die Menge der ganzen Zahlen $\mathbb{Z}$, die Menge der rationalen Zahlen $\mathbb{Q}$ und schließlich die Menge der reellen Zahlen $\mathbb{R}$.
Ist ein Objekt x in einer Menge M enthalten, d.h. x ist Element von M, dann schreiben wir

$$x \in M.$$

Entsprechend bedeutet

$$x \notin M,$$

dass x nicht in M enthalten ist.

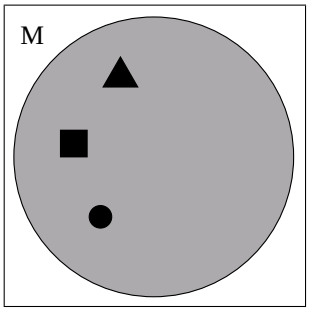

Abbildung 1: Diagrammartige Darstellung einer Menge.

Eine diagrammartige Schreibweise für Mengen zeigt Abbildung 1. Dabei stehen die Symbole in dem Kreis für die Elemente der Menge. In der konvetionellen Schreibweise würde man schreiben:

$$\{\blacktriangle, \blacksquare, \bullet\}$$

Wir werden diese Diagrammdarstellung im Zusammenhang mit Funktionen noch öfter verwenden.

1.2.3 Einige Eigenschaften von Mengen

1 Definition Zwei Mengen M und N sind gleich, geschrieben $M = N$, wenn sie dieselben Elemente enthalten.

Es kommt demnach nicht auf die Nennungsreihenfolge der Elemente an. Etwa bezeichnen $\{1, 2, 3\}$ und $\{3, 1, 2\}$ dieselbe Menge – sie enthalten dieselben Elemente.

2 Definition Eine Menge M ist *Teilmenge* einer Menge N, geschrieben $M \subset N$, wenn jedes Element von M auch Element von N ist.

3 Definition Die *Schnittmenge* zweier Mengen M und N ist die Menge der Elemente, die sowohl in M als auch in N enthalten sind. Wir schreiben $M \cap N$.

4 Definition Zwei Mengen M und N heißen *disjunkt*, wenn sie keine gemeinsamen Elemente enthalten. Es gilt $M \cap N = \emptyset$; der Schnitt ist leer.

1.2.4 Mengen enthalten Mengen

Da Mengen selbst mathematische Objekte sind, können sie Elemente anderer Mengen sein. Von etwas merkwürdig anmutenden Ausdrücken in diesem Zusammenhang sollte man sich nicht verwirren lassen. Die Menge

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

enthält vier Elemente: Die Menge $\{1, 2\}$ sowie die Zahlen 1, 2 und 3. Dagegen enthält die Menge

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

nur drei Elemente: Die beiden Mengen $\{1, 2\}$ und $\{2, 3\}$ sowie die Zahl 1.
Bei den beiden Ausdrücken

$$\{\{2, 3\}, \{\{2, 3\}\}\} \tag{5}$$

$$\{\{2, 3\}, \{2, 3\}\} \tag{6}$$

ist (5) wohlgeformte Mengenschreibweise, nicht jedoch (6). Im letzteren Fall ist zweimal die Menge $\{2, 3\}$ als Element aufgeführt. Bei (5) ist dieses Problem nicht gegeben, da die Elemente $\{2, 3\}$ und $\{\{2, 3\}\}$ ungleich sind.
Ebenfalls ist die leere Menge ungleich der Menge, die nur die leere Menge enthält:

$$\{\} \neq \{\{\}\}$$

1.2.5 Zusammenfassung: Mengen

Ein intuitives Verständnis des Mengenbegriffs wurde dahingehend präzisiert, dass eine Menge durch die in ihr enthaltenen Elemente gegeben ist. Neben wichtigen Eigenschaften von Mengen wurden verschiedene Schreibweisen für Mengen kennengelernt. Hierbei tritt ein wichtiges Prinzip erstmals auf: Verschiedene Schreibweisen können zur Angabe derselben Objekte verwendet werden. Es ist daher wichtig, niemals die Schreibweise mit dem, was sie beschreibt, zu verwechseln.

1.3 Tupel

1.3.1 Definition

Bis jetzt wurden Mengen als eine Möglichkeit vorgestellt, Objekte zu neuen Objekten zusammenzufassen. Dabei haben wir gesehen, dass die Elemente bei der Angabe von Mengen stets nur einmal aufgeführt wurden und dass auf die Reihenfolge keine Rücksicht genommen werden musste. In vielen Fällen ist aber die Reihenfolge wichtig oder Mehrfachnennungen sind nötig.
Bei Punkten in der Ebene, die über zwei Koordinaten beschrieben werden, ist es sehr wohl wichtig, welche der Koordinaten zuerst genannt wird. Für Punkte im zweidimensionalen Raum verwendet man üblicherweise die Schreibweise $(1, 2)$, hier für den Punkt dessen x-Koordinate 1 und dessen y-Koordinate 2 ist.
Tupel sind eine verallgemeinerte Form zur Darstellung geordneter Aufzählungen von Objekten. Dabei wird dieselbe Schreibweise wie für Punkte verwendet, doch gegebenenfalls mehr als zwei Komponenten haben kann, wie in

$$(2, 5, 2, 1).$$

Zu beachten ist wie gesagt die Nennungsreihenfolge, so dass beispielsweise

$$(1, 2) \neq (2, 1).$$

Zum Schluss dieses Abschnittes soll noch eine formale Definition für Tupel gegeben werden.

7 Definition Sei $n \geq 2$ eine natürliche Zahl. Zu n Objekten $x_1, \ldots, x_n$ (die durch Nummerierung von 1 bis n mit einer Reihenfolge versehen sind), bilden wir ein neues Objekt $(x_1, \ldots, x_n)$, das wir das n-Tupel von $x_1, \ldots, x_n$ nennen.

1.3.2 Tupel und Mengen

Tupel und Mengen lassen sich beliebig komplex verschachteln: Mengen und Tupel können in Tupeln zusammengefasst werden und Mengen können Tupel als Elemente haben. Die folgenden Beispiele sind wohlgeformte Schreibweisen für Tupel:

$$(2, 1) \quad (2, 2) \quad ((1, 2), (1, 2)) \quad (\emptyset, \{\}) \quad (\{(5, 6), (8, 1)\}, 3)$$

1.4 Der Gebrauch von Variablen

Variablen sind Stellvertreter-Symbole, die anstelle konkreter Objekte benutzt werden. Für eine Variable kann stets ein Objekt eingesetzt werden; die Variable vertritt das Objekt.
Schauen wir uns noch einmal die formale Definition (7) für Tupel an. In diesem Fall ist n eine Variable, die eine beliebige natürliche Zahl (größer oder gleich 2) repräsentiert. Statt n kann eine beliebige natürliche Zahl (größer oder gleich 2) verwendet werden.

Entsprechend stehen die Variablen x_1 bis x_n für je ein beliebiges Objekt aus einer nicht näher angegebenen Menge. (vgl. Abschnitt 1.2.1)
Stellen wir uns n als die Zahl 8 vor. Wählen wir $n = 8$ so lautet der Satz:

> „Zu 8 Objekten $x_1, \ldots, x_8$ (die durch Nummerierung von 1 bis 8 mit einer Reihenfolge versehen sind), bilden wir ein neues Objekt $(x_1, \ldots, x_8)$, das wir das 8-Tupel von $x_1, \ldots, x_8$ nennen."

In diesem Fall gibt es 8 Variablen x_1, x_2, x_3, x_4, x_5, x_6, x_7 und x_8, von denen jede ein beliebiges Objekt vertritt.
Allerdings muss jedes n stets durch die gleiche Zahl ersetzt werden. Falsch wäre etwa die folgende Variante:

> „Zu 8 Objekten $x_1, \ldots, x_8$ (die durch Nummerierung von 1 bis 5 mit einer Reihenfolge versehen sind), bilden wir ein neues Objekt $(x_1, \ldots, x_9)$, das wir das 122-Tupel von $x_1, \ldots, x_4$ nennen."

Dieser Satz ergibt keinen Sinn und verdeutlicht, warum eine Variable zwar beliebige Objekte vertreten kann, jedoch stets nur eines zur gleichen Zeit.
Auf der anderen Seite kann es sehr wohl vorkommen, dass zwei unterschiedliche Variablen dasselbe Objekt vertreten. Beispielsweise können die Variablen $x_1, \ldots, x_n$ folgendermaßen belegt werden: $x_1 = 1$, $x_2 = 5$, $x_3 = 8$, $x_4 = -20$, $x_5 = 9$, $x_6 = 0$, $x_7 = 5$, und $x_8 = 5$. Das dazugehörige 8-Tupel ist $(1, 5, 8, -20, 9, 0, 5, 5)$.

1.5 Funktionen

1.5.1 Funktionen als Zuordnung

Im Zusammenhang mit Funktionen fallen vielen spontan Parabeln, Geraden und dergleichen ein. All das sind zwar Funktionen, jedoch gibt es wesentlich mehr Funktionen, die nicht in dieser Form verstanden werden können. Es ist sinnvoll, sich Funktionen nicht einfach als Graphen oder Formeln wie etwa x^2 vorzustellen. Wir werden zunächst eine allgemeinere Vorstellung, was eine Funktion ist, entwickeln.

8 Definition Funktionen sind eindeutige Zuordnungen zwischen Mengen. Jedem Element aus der *Definitionsmenge* wird eindeutig ein bestimmter Funktionswert aus der *Wertemenge* zugeordnet.[1]

Was heißt das? Zum Beispiel beschreibt jedes Telefonbuch eine Funktion: Jedem Inhaber eines Telefonanschlusses innerhalb eines gewissen Gebietes wird darin seine Telefonnummer zugeordnet. Wenn man dieses Beispiel im Sinn behält, wird man nicht in die Gefahr geraten, eine Funktion für eine explizite Berechnungsvorschrift zu halten.
Das ist in der Tat der springende Punkt. Die meisten Funktionen, die uns im Alltag begegnen, sind als explizite mathematische Gleichung angegeben, wie zum Beispiel die

[1] Eine formale Definition des Funktionsbegriffs findet sich bei [1]

Normalparabel durch die Gleichung $f(x) = x^2$ gegeben ist. Allerdings darf hier nicht die Schreibweise mit der Funktion selber verwechselt werden. Abbildung 2 zeigt zum Beispiel, wie man eine Funktion über ein Diagramm darstellen kann, ohne eine Formel anzugeben. Das Diagramm ist hierbei nur eine andere Schreibweise.

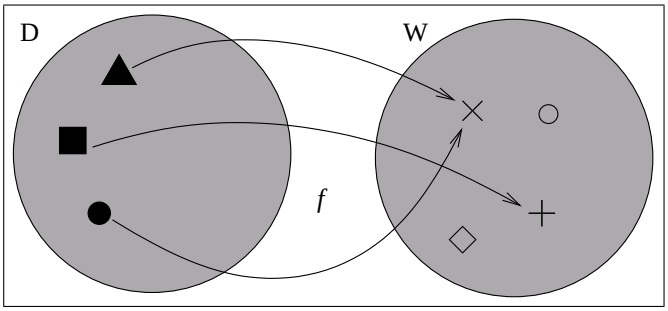

Abbildung 2: Diagrammartige Darstellung einer Funktion. Die Pfeile führen stets von den Elementen der Definitionsmenge D zu ihren Funktionswerten in der Wertemenge W.

Man beachte, dass an *jedem* der Elemente in D *genau ein* Pfeil beginnt; schließlich haben wir eine *eindeutige* Zuordnung gefordert. Dagegen ist es möglich, dass ein Element aus W Funktionswert von zwei Elementen aus D ist, wie es in Abbildung 2 für das Element $\times \in W$ der Fall ist.
Der nächste Abschnitt stellt die gängige Schreibweisevon Funktionen vor und setzt die bisherige Sichtweise von Funktionen mit dieser Schreibweise in Beziehung.

1.5.2 Bezeichungen und Schreibweisen

Wenn f eine Funktion, D ihre Definitionsmenge und W ihre Wertemenge bezeichnen, dann ordnet f jedem Element $x \in D$ eindeutig ein Element $y \in W$ zu. x wird Argument und y Funktionswert genannt. Der Ausdruck $f(x)$ bezeichnet den Funktionswert von x.

Bemerkung: *Auch hier sind die Variablen x und y Vertreter beliebiger Elemente aus den Mengen D respektive W.*

Tupel können als Argument einer Funktion auftreten. Für den Ausdruck

$$f((x_1, \ldots, x_n))$$

soll alternativ auch die Schreibweise

$$f(x_1, \ldots, x_n)$$

möglich sein.
Erinnern wir uns nun an Abbildung 2. Die Definitionsmenge D ist $\{\blacktriangle, \blacksquare, \bullet\}$, die Wertemenge $W = \{\diamondsuit, +, \times, \circ\}$. Die Funktion f lässt sich nicht durch eine sinnfällige Gleichung

beschreiben. Allerdings lassen sich die Funktionswerte für verschiedene Argumentwerte explizit angeben:

$$f(\blacktriangle) = \times f(\blacksquare) = + f(\bullet) = \times \tag{9}$$

1.5.3 Funktion als Gleichung

An dieser Stelle werden wir die bekannten Parabeln und Geraden in den oben beschriebenen Funktionsbegriff einordnen. Neben der Darstellung als Schaubild (vgl. Abbildung 2) oder als explizit angegebene Funktionswerte, kann in vielen Fällen ein mathematischer Ausdruck für den Funktionswert angegeben werden. Dann kann die Funktion auch als Gleichung der Form

$$f(x) = A$$

angegeben werden, wobei A ein Ausdruck ist, der den Funktionswert repräsentiert.
Im Falle

$$f(x) = x^2 - 6 \tag{10}$$

gibt der Ausdruck auf der rechten Seite der Gleichung an, wie der Funktionswert definiert ist: Für ein beliebiges $x \in D$ ist der Funktionswert $f(x)$ stets durch den Ausdruck $x^2 - 6$ eindeutig bestimmt.

1.5.4 + als Funktion

Nachdem der Funktionsbegriff eingehender betrachtet wurde, kann nun eine Einordnung der üblichen arithmetischen Operatoren erfolgen. So ließe sich zum Beispiel der Additionsoperator + für reelle Zahlen als Funktion auffassen, die jedem Tupel (a_1, a_2) eine Zahl a_3 zuordnet, wobei $a_1, a_2, a_3 \in \mathbb{R}$. Dabei ist a_3 die Summe von a_1 und a_2. Es handelt sich bei Operatoren lediglich um eine andere Schreibweise für Funktionsanwendungen:

$$a_1 + a_2 = + (a_1, a_2) = a_3$$

Die Definitionsmenge dieser Funktion ist die Menge aller 2-Tupel reeller Zahlen sind. Die Wertemenge ist $\mathbb{R}$.

1.5.5 Eindeutigkeit und Bijektion

In den vergangenen Abschnitten hatten wir uns insbesondere mit dem Begriff der Eindeutigkeit im Zusammenhang mit Zuordnungen befasst. Eine Funktion f ordnet den Elementen der Definitionsmenge D genau ein Element der Wertemenge W zu. Von einem beliebigen Element $x \in D$ ist zweifelsfrei sein Funktionswert $y \in W$ festzustellen.
Es wird jedoch nicht gefordert, dass auch jedem Element der Wertemenge eindeutig ein Element der Definitionsmenge zugeordnet ist. So kann es für ein Element $y \in W$ der Fall sein, dass nicht eindeutig bestimmt ist, für welches Element $x \in D$ gilt $f(x) = y$.
Beispielsweise ist im Falle der Funktion f in (10) für $f(x) = 10$ nicht eindeutig bestimmt, welchen Wert x hat. Sowohl für $x = 4$ als auch $x = -4$ ist die Gleichung $f(x) = 10$ erfüllt – sowohl 4 als auch -4 haben den Funktionswert 10. Bei der Funktion in Abbildung 2 ist

für $f(x) = \times$ nicht eindeutig feststellbar, ob $x = \blacktriangle$ oder $x = \bullet$. Denn sowohl $f(\blacktriangle) = \times$ als auch $f(\bullet) = \times$.

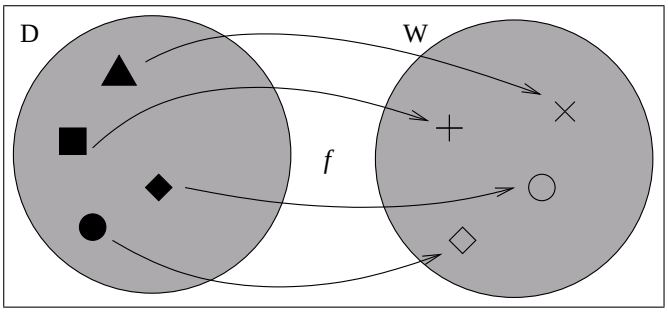

Abbildung 3: Diagrammartige Darstellung einer bijektiven Funktion

Eine interessante Eigenschaft von Funktionen ist die Bijektivität. *Bijektionen* sind Funktionen, die sich durch die Eigenschaft auszeichnen, dass die Umkehrfunktion existiert. D.h. es kann auch von einem Element y der Wertemenge eindeutig festgestellt werden, welches Element x der Definitionsmenge den Funktionswert y hat.
Dies ist zum Beispiel bei der Funktion in Abbildung 3 der Fall. Bildlich drückt sich die Bijektivität der Funktion in der Tatsache aus, dass an jedem Element der Wertemenge genau ein Pfeil endet.

1.5.6 Zusammenfassung: Funktionen

Eine Funktion ordnet jedem Element der Definitionsmenge, dem Argumentwert, genau ein Element der Wertemenge zu, den Funktionswert. Wir haben festgestellt, dass eine Funktion nicht mit der Art und Weise verwechselt werden darf, in der sie beschrieben ist. Ob als Auflistung von Argument-Funktionswert-Paaren wie etwa im Telefonbuch, als Diagramm wie in Abbildung 2 oder als explizite mathematische Gleichung: Stets sind es nur Darstellungsweisen – die Funktion selbst ist nur die Zuordnung der Elemente einer Menge zu den Elementen einer anderen Menge.

1.6 Schlussrichtung

1.6.1 Ein alltägliches Beispiel

Worum es in diesem Abschnitt gehen soll ist das Problem der Schlussrichtung. Es ist wichtig, sich stets darüber im Klaren zu sein, in welcher Beziehung zwei (logische) Aussagen stehen. Die Aussage

Alle Hunde haben ein Fell

bedeutet im Grunde

Wenn etwas ein Hund ist, dann hat es auch ein Fell. (11)

Es wird ausgesagt, dass aus der Tatsache, dass etwas ein Hund ist, gefolgert werden kann, dass dieses etwas ein Fell besitzt. Entgegen der menschlichen Intuition ist Aussage (11) mit Aussage der folgenden Aussage äquivalent, d.h. gleichbedeutend:

Wenn etwas kein Fell hat, dann ist es kein Hund (12)

Die Äquivalenz der Aussagen (11) und (12) ist zunächst nicht einleuchtend und wir vollziehen sie daher anhand von Beispielen nach.
In einer Welt in der mindestens ein felloser Hund existiert, gilt die Aussage (11) nicht. Man kann in dieser Welt nicht von der Tatsache, dass etwas ein Hund ist, auf die Anwesenheit eines Felles schließen. Zugleich gilt in dieser Welt auch Aussage (12) nicht. Denn ein felloser Hund verbietet es, von der Abwesenheit eines Felles darauf zu schließen, dass es sich bei einem Ding nicht um einen Hund handelt.
In einer Welt, in der alle Hunde Felle haben, gelten dagegen beide Aussagen. Aussage (11) trifft auf alle Hunde dieser Welt zu. Für alle anderen Dinge (Tiere) gilt sie ebenfalls, weil es sich nicht um Hunde handelt und daher die Prämisse (Vorbedingung) nicht erfüllt ist. Aussage (12) gilt in dieser Welt, da alle Dinge ohne Fell in dieser Welt auch keine Hunde sind. Alle Dinge mit Fell erfüllen die Prämisse nicht und die Aussage trifft daher auf sie zu.
Es ist stets der Fall, daß entweder beide Aussagen wahr sind oder beide Aussagen falsch sind, womit wir ihre Äquivalenz gezeigt haben.

Bemerkung: *Generell ist das Durchdenken von Beispielen sehr hilfreich, wenn es darum geht, intuitiv schwer zugängliche Sachverhalte zu verstehen. Speziell bei logischen Ausagen ist es oft aufschlussreich , ihre Gültigkeit in verschiedenen hypothetischen Welten durchzuspielen. Wichtig ist, dass man bei den hypothetischen Welten alle Fälle in Betracht zieht. Im Beispiel waren dies die beiden Fälle, in denen fellose Hunde existieren oder eben nicht existieren – weitere Fälle gibt es nicht.*

Verallgemeinert gilt die Äquivalenz auch zwischen den Aussagen

Wenn A gilt, dann gilt auch B (13)

und

Wenn B nicht gilt, dann gilt auch A nicht. (14)

1.6.2 Verwirrende Beispiele

Das Hundebeispiel aus dem vorigen Abschnitt erscheint noch recht übersichtlich, auch wenn die Äquivalenz der Aussagen (13) und (14) sorgfältig nachvollzogen werden muss.

Wason-Task. Ein Beispiel, in dem die oben angesprochene Äquivalenz der Aussagen (13) und (14) verwendet werden muss, ist die Lösung des *Wason-Task*. Dabei geht es um die Beantwortung der Frage, welche der Karten in Abbildung 4 umgedreht werden

müssen, um die Gültikkeit der Regel (15) zu überprüfen.

Wenn auf der einen Seite der Karte ein Vokal steht,
steht auf der anderen Seite der Karte eine gerade Zahl (15)

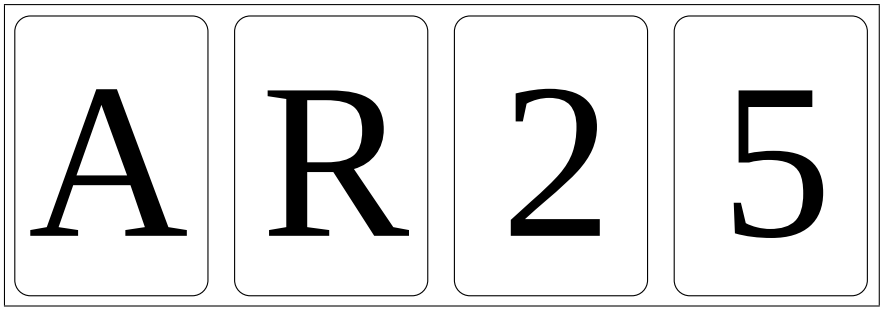

Abbildung 4: Welche der Karten müssen umgedreht werden um zu prüfen, ob die Regel (15) bei diesen vier Karten erfüllt ist?

Bringen Störche Kinder? Zuletzt wollen wir uns einem anderen verwirrendem Beispiel mit alltäglicherem Kontext widmen. Wenn man davon ausgeht, dass Kinder durch Störche verursacht werden, kann man daraus schließen, dass die Geburtenrate mit dem Auftreten von Störchen korrelliert ist.[2] Den gleichen Schluss kann man allerdings auch ziehen, wenn man davon ausgeht, dass Störche durch Geburten von Kindern verursacht werden.[3] Wie wir sehen, kann man angesichts dieser völlig unterschiedlicher Sachverhalte zu ein und demselben Schluss gelangen.
Tatsächlich kann man eine Korrelation zwischen Geburtenrate und dem Auftreten von Störchen feststellen.[4] Nun aber daraus zu schließen, Störche würden die Kinder bringen ist sicherlich so fehl am Platze, wie zu schließen, Störche entstünden bei der Geburt von Kindern als Nebenprodukt.

1.6.3 Logische Operatoren

Für die Verknüpfung von logischen Aussagen gibt es verschiedene *logische Operatoren*, die hier kurz vorgestellt werden. In Abbildung 5 sind die gängigen logischen Operatoren zusammengefasst, wobei A und B für logische Aussagen stehen.

(i) Die *Negation* einer Aussage $\neg A$ ist genau dann wahr, wenn A falsch ist.

[2]Nehmen wir mal an, dass alle Störche gleichviele Kinder pro Zeiteinheit verursachen.
[3]Auch hier nehmen wir an, dass jede Geburte gleichviele Störche erzeugt.
[4]In den Vergangenen 100 Jahren ging sowohl die Zahl der Störche als auch die Zahl der Geburten in Deutschland zurück.

A	B	$\neg A$	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$A \Leftrightarrow B$
W	W	F	W	W	W	W
W	F	F	W	F	F	F
F	W	W	W	F	W	F
F	F	W	F	F	W	W

Abbildung 5: Wahrheitstabelle gängiger logischer Operatoren

(ii) Die *Adjunktion* $\vee$ zweier Aussagen ist genau dann wahr, wenn mindestens eine der beiden Aussagen wahr ist. Sie wird auch als *einschließende Oder-Verknüpfung* bezeichnet.[5]

(iii) Die *Konjunktion* $\wedge$ oder auch *Und-Verknüpfung* zweier Aussagen ist genau dann wahr, wenn beide Aussagen wahr sind.

(iv) Die *Implikation* $A \Rightarrow B$ ist genau dann falsch, wenn A wahr ist und B falsch.

(v) Zwei Aussagen A und B sind logisch *äquivalent*, wenn A wahr ist, genau dann, wenn B war ist. Insbesondere heißt das, dass A aus B folgt und B aus A.

Bemerkung: *Überraschen mag an dieser Stelle, dass die* Implikation $A \Rightarrow B$ *(sprich: Aus A folgt B) genau dann falsch ist, wenn A wahr ist und B falsch. Insbesondere heißt das nämlich, dass eine falsche Aussage eine wahre Aussage impliziert. Der Gedanke dahinter ist, dass aus einer wahren Aussage stets eine wahre Aussage folgen soll, aus einer wahren Aussage jedoch keine falsche Aussage. Ferner kann aus einer falschen Prämisse alles gefolgert werden!*

1.6.4 Zusammenfassung: Schlussrichtung

Die Lehre, die man aus diesem Abschnitt ziehen kann, ist die, dass man auch in alltäglichen, vermeintlich intuitiv zugänglichen Kontexten, sehr leicht falsche Schlüsse zieht. Das ist keine verwerfliche Fehlleistung sondern eine geradezu menschliche: Menschen springen in vielen Fällen unbewusst wenn es um logische Schlüsse geht. Das liegt daran, dass dies in den allermeisten Fällen keine Schwierigkeiten verursacht und wesentlich schneller geht. Für methodisch sauberes wissenschaftliches Arbeiten ist das aber nicht akzeptabel und es ist daher notwendig, Schlüsse sehr sorgfältig und am besten formal zu prüfen.
Aus diesem Grund schlossen wir das Thema Schlussrichtung mit der Präsentation der gängigen logischen Operatoren ab. Sie sind formale Schreibweisen dessen, was in diesem Abschnitt weniger formal angesprochen wurde.

[5]Die ausschließende Oder-Verknüpfung soll an dieser Stelle keine Rolle spielen.

►►► Woche 2 ◄◄◄

1.7 Rekursion

1.7.1 bildlich...

Wer kennt sie nicht, die Frau auf der Waschmittelpackung, die eine Waschmittelpackung hält, auf der eine Frau, die eine Waschmittelpackung hält... Das Beispiel mit der Waschmittelpackung zeichnet sich dadurch aus, dass das Bild sich selbst enthält. Hierbei handelt es sich um Rekursion. Bevor wir uns in einer formalen Weise dem Thema Rekursion widmen, sollen hier einige Betrachtungen an anderen bildlichen Strukturen gemacht werden.
Die Spirale aus Abbildung 6 ist eine Struktur, die einige interessante Eigenschaften hat: Denkt man sich die innerste Windung der Spirale weg, erhält man zwar eine etwas andere Spirale, die aber nach demselben Prinzip konstruiert werden kann. Lediglich der Anfang muss etwas geändert werden.

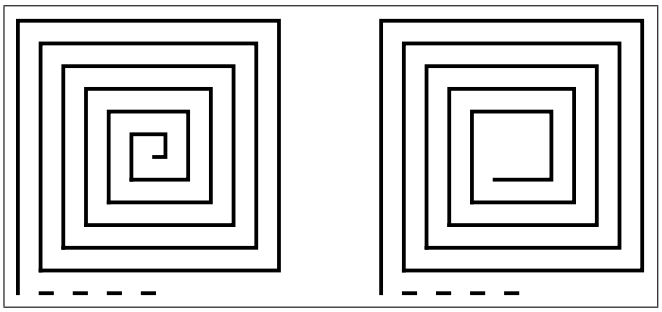

Abbildung 6: Rekursiv konstruierte Spiralen. Bei der rechten wurde innen etwas anders begonnen als bei der linken – dennoch ist das Konstruktionsprinzip dasselbe. Diese Spiralen lassen sich nach einer einfachen Vorschrift prinzipiell ins Undendliche fortsetzten.

Es wäre ohne weiteres möglich, eine beliebig große Spirale zu erstellen, ohne dass eine komplexere Konstruktionsregel notwendig wäre. In Abbildung 7 sind einzelne Teilspiralen unterschiedlich schattiert, so dass die sich wiederholende Struktur gut erkennbar ist.
Die große Spirale ist eigentlich eine elementare Spirale an deren Ende wiederum eine große Spirale ansetzt, die wiederum eine elementare Spirale ist, an deren Ende erneut eine große Spirale ansetzt... Die Tatsache, dass das gleiche Konstruktionsprinzip ineinander verschachtelt wird macht diese Spirale zu einer rekursiv definierten Struktur, ähnlich wie die Waschmittelpackung.
Abbildung 8 zeigt eine weitere rekursive Struktur. Im Unterschied zu den vorigen Beispielen ist die Schneeflocke nicht nur einfach rekursiv sondern mehrfach. Während die Spirale ein Spiralenstück mit *genau einer* Spirale an dessen Ende ist, ist die Schneeflocke

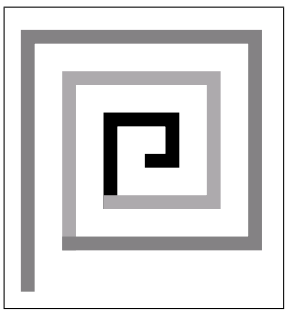

Abbildung 7: Rekursiv konstruierte Spirale, bei der wiederkehrende Elementarspiralen unterschiedlich schattiert sind.

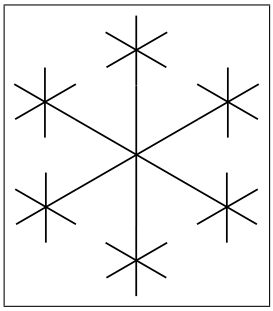

Abbildung 8: rekursiv konstruierte Schneeflocke

ein sechsstraliger Stern an dessen *sechs* Strahlen jeweils eine Schneeflocke angebracht ist. Bei der Flocke liegt eine baumartig verzweigte Struktur vor, während die Spirale lediglich eine Verkettung von (leicht modifizierten) Elementarspiralen ist.

1.7.2 rekursive Funktionen

Ähnlich den rekursiv definierten und bildlich dargestellten Strukturen, die wir im letzten Abschnitt gesehen haben, gibt es auch Funktionen, die rekursiv definiert sind. Rekursion wird typischerweise verwendet, um sich wiederholende, verschachtelte oder hierarchieartige Strukturen oder Ausdrücke zu definieren.
Die Summe der Zahlen 1 bis 20 ließe sich natürlich folgendermaßen aufschreiben:

$$1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20$$

Abgesehen davon, dass dies sehr unbefriedigend ist, würde diese Methode spätestens bei größeren Anzahlen von Summanden völlig unpraktikabel werden. Im Falle der Summe bedient sich die Mathematik der Summenschreibweise, die aber für unsere späteren Zwecke der funktionalen Programmierung nicht geeignet ist. Eine allgemeinere Form, kettenartige mathematische Ausdrücke zu definieren, ist die Rekursion.
Eine Funktion f_{sum}, die die Summe der Zahlen von 1 bis n liefert, wobei n eine natürliche

Zahl ist, ließe sich rekursiv definieren:

$$\begin{aligned} f_{sum}(1) &= 1 \\ f_{sum}(n) &= n + f(n-1) \quad \text{für } n \neq 1,\ n \in \mathbb{N} \end{aligned} \tag{16}$$

Warum funktioniert diese Definition?
Um dies zu verstehen ist es am besten, die Funktionsanwendungen nacheinander durchzuführen und nachzuvollziehen, wie dabei der gewünschte Ausdruck Schritt für Schritt „entsteht“. Dies werden wir nun für $f_{sum}(4)$ tun:

$$\begin{aligned} f_{sum}(4) &= 4 + f_{sum}(4-1) \\ = 4 + f_{sum}(3) &= 4 + 3 + f_{sum}(3-1) \\ = 4 + 3 + f_{sum}(2) &= 4 + 3 + 2 + f_{sum}(2-1) \\ = 4 + 3 + 2 + f_{sum}(1) &= 4 + 3 + 2 + 1 \\ &= 10 \end{aligned} \tag{17}$$

Bei der Auswertung solcher rekursiver Funktionen wird die Funktionsdefinition (16)) mehrmals angewendet. Der Abschnitt 1.4 merkt an, dass eine Variable zwar für verschiedene Werte stehen kann, jedoch immer nur einen zur selben Zeit. Im Falle mehrmaliger Anwendung derselben Funktionsdefinition (wie in (17)) haben jedoch die verwendeten Variablen bei jeder Anwendung andere Werte erhalten. Die einzelnen Funktionsanwendungen haben jeweils eine eigene Belegung der in der Definition verwendeten Variablen. Jede Anwendung der Funktionsdefinition hat einen eigenen *Kontext*, in dem die Variablen belegt werden. Wichtig ist dabei, dass bei jeder Funktionsanwendungen, d.h. in jedem Kontext, jede Variable nur einen Wert erhält. (So wie es im Abschnitt 1.4 gefordert ist).

Bemerkung: *Im Zusammenhang mit LISP wird statt Kontext das Wort* Bindungsumgebung *verwendet. Bindungsumgebungen werden in LISP zur Realisierung von Kontexten bei der Funktionsanwendung verwendet.*

1.7.3 Anschauung der Anwendung rekursiver Funktionen

Um die Anwendung von rekursiven Funktionen besser verstehen zu können, eignet sich die Metapher von aufeinander gestapelten Ebenen. Die Auswertung beginnt auf der höchsten Ebene und resultiert in dem Ausdruck der auf der *rechten Seite* der Definition steht. Enthält dieser Ausdruck eine erneute Anwendung der Funktion, wird die Auswertung an die darunterliegende Ebene deligiert. Diese deligiert rekursive Verwendungen wiederum an die unter ihr liegende Ebene usw. Dies setzt sich fort, bis der Basisfall erreicht ist und das Ergebnis ohne weitere Rekursion ermittelt werden kann. Das Ergebnis der untersten Ebene wird in den Ausdruck der darüberliegenden Ebene eingesetzt, so dass nun ihr Ergebnis der Ebene darüber zur Verfügung steht. Das Ergebnis, das auf diese Weise auf der obersten Ebene erhalten wird ist der gesuchte Funktionswert.

1.7.4 Aufbau rekursiver Funktionen

Typischerweise werden in rekursiven Funktionen mindestens zwei Fälle unterschieden:

1. Der *Basisfall* definiert die Funktion für den Argumentwert, für den der Funktionswert ohne Rekursion gegeben ist.

2. Der *rekursive oder nichttriviale Fall* ist rekursiv definiert, d.h. zur Definition des nichttrivialen Falles wird die Funktion selbst verwendet. Dabei ist darauf zu achten, dass nach endlichmaliger Anwendung des nichttrivialen Falles ein Ausdruck erhalten wird, der nur noch den Basisfall verwendet.

Bei (16)) ist der Basisfall $f_{sum}(1)$, für den das Ergebnis ohne Rekursion ermittelt werden kann. Der rekursive Fall ist definiert als $f_{sum}(n) = n + f(n-1)$ und drückt damit den Funktionswert $f_{sum}(n)$ durch einen Ausdruck aus, der wiederum f_{sum} verwendet. Allerdings ist der Argumentwert bei der rekursiven Verwendung von f_{sum} stets kleiner als der der erhaltene Argumentwert, so dass der Basisfall irgendwann erreicht wird.

1.7.5 Baumrekursion

Wie bei der Schneeflocke bereits anschaulich zu sehen ist, kann Rekursion nicht nur kettenartige Strukturen und Ausdrücke beschreiben, sondern auch baumartige Strukturen und Ausdrücke. Eine Baumrekursion liegt dann vor, wenn in der Definition des nichttrivialen Falles einer Funktion mehrere Male die Funktion selber verwendet wird. Eine numerische Funktion dieser Art ist die Fibonacci-Funktion f_{fib}, die durch

$$\begin{aligned} f_{fib}(0) &= 0 \\ f_{fib}(1) &= 1 \\ f_{fib}(n) &= f_{fib}(n-1) + f_{fib}(n-2) \quad \text{für } n \geq 2,\ n \in \mathbb{N} \end{aligned} \tag{18}$$

gegeben ist.
Zerlegen wir die Auswertung des Ausdrucks $f_{fib}(4)$ in Einzelschritte um diese Definition zu verstehen:

$$\begin{aligned} f_{fib}(4) &= f_{fib}(3) && && + && f_{fib}(2) && \\ & && \downarrow \quad \searrow && && \downarrow \quad \searrow && \\ &= f_{fib}(2) && + f_{fib}(1) && + && f_{fib}(1) && + f_{fib}(0) \\ & && \downarrow\searrow && && && \\ &= f_{fib}(1) + f_{fib}(0) && + 1 && + && 1 && + 0 \\ &= 1 \quad + 0 && + 1 && + && 1 && + 0 \end{aligned}$$

Es ist zu erkennen, dass der Ausdruck $f_{fib}(4)$ eine ganze Flut von rekursiven Funktionsanwendungen nach sich zieht. Jede Anwendung der Funktion f_{fib} im nichttrivialen Fall

zieht zwei rekursive Anwendungen nach sich, die separat behandelt werden. Auf diese Weise entstehen neue Zweige, in denen Teilausdrücke ausgewertet werden. Erst werden alle Zweige ausgewertet und dann das Gesamtergebnis zusammengesetzt. Die Aufteilung in Zweige ist im Schaubild oben durch Pfeile gekennzeichnet und in Abbildung 9 noch einmal verdeutlicht.

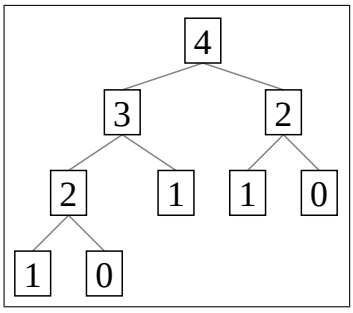

Abbildung 9: Jeder Kasten steht für eine Funktionsanwendung der Funktion f_{fib}. Untergeordnete rekursiven Anwendungen sind über graue Linien mit ihrer übergeordneten Anwendung verbunden.

1.7.6 Exkurs: Iteration

Im Abschnitt 1.7.2 haben wir uns mit der Auswertung von kettenartigen mathematischen Ausdrücken befasst. Eine andere Methode, derartige Berechnungen auszudrücken, ist die Iteration. Sie ist allerdings nicht im mathematischen Sinne zu verstehen, sondern ist eher mit einem Kochrezept vergleichbar. Anstatt durch Funktionsdefinition und Anwendung dieser Definition auf bestimmte Argumente zu einem Ergebnis zu kommen, werden hier Befehle abgearbeitet, die Werte manipulieren.
Dabei werden Variablen nicht als Platzhalter verwendet, die nur einen Wert im jeweiligen Kontext vertreten können. Stattdessen sind Variablen Speicherplätze, denen zu jeder Zeit ein neuer Wert zugewiesen werden kann. Eine Variable verhält sich hier in etwa wie der Speicher eines modernen Taschenrechners, in dem man einen beliebigen Wert ablegen kann, um ihn später wieder abzurufen.
Iteration hat das wiederholte Ausführen von Anweisungen zum Ziel. Die zu wiederholenden Anweisungen befinden sich im *Rumpf* und werden bei jedem Durchlauf der Schleife einmal ausgeführt. Üblicherweise wird bei einer Iteration eine Variable als Zähler verwendet. Diese *Iterationsvariable* wird am Anfang auf einen definierten *Anfangswert* gesetzt. Im *Iterationsschritt* wird die Iterationsvariable nach jedem Schleifendurchlauf um einen definierten Wert geändert. Die Schleife bricht ab, d.h. wird nicht mehr wiederholt, wenn die *Abbruchbedingung* eintritt.
Die Berechnung der Summe der Zahlen von 1 bis 20 würde auf der Basis von Iteration und Befehlsausführung folgendermaßen formuliert werden:

1. Setze die Variable (= Speicherzelle) i auf den Anfangswert 1.

2. Setze die Variable x auf den Wert 0.

3. Wiederhole die folgende Anweisung (Rumpf), bis i den Wert 20 überschritten hat. (Abbruchbedingung)
   Erhöhe nach jeder Ausführung des Rumpfes die Variable i um eins. (Iterationsschritt)

   a) Erhöhe den Wert der Variablen x um den Wert von i.

Die ersten beiden Befehle dienen nur der Belegung der Variablen mit geeigneten Werten. Die dritte Anweisung ist die Schleife. Sie führt die Anweisung 3a 20 mal aus – für $i = 1$ bis $i = 20$. Bei jedem Durchlauf wird der Wert von i zu dem von x addiert und das Ergebnis wieder in x gespeichert. Hat i den Wert 21 erreicht, bricht die Schleife ab; die Variable x enthält die Summe der Zahlen 1 bis 20, das Ergebnis.
Im Gegensatz zur Rekursion, bei der bei vielen rekursiven Ausdrücken relativ komplexe Ausdrücke auftreten, ist hier zu jedem Zeitpunkt nur der Wert von x und i zu speichern, sowie die nächste auszuführende Anweisung. Andererseits kann Baumrekursion nicht unbedingt durch Iteration ausgedrückt werden.

1.7.7 Zusammenfassung: Rekursion

Rekursion ist ein wichtiges Werkzeug um kettenartige Strukturen zu beschreiben. Dabei handelt es sich meist um mathematische Ausdrücke, die durch rekursiv definierte Funktionen beschrieben werden. Jedoch kann Rekursion genauso zur Beschreibung graphischer Elemente oder anderer Strukturen verwendet werden.
Wichtig ist, dass rekursiv definierte Funktionen stets einen Basisfall abdecken, der nicht rekursiv definiert ist und dass der rekursive Fall irgendwann auf den Basisfall zurückgeführt werden kann. Ansonsten wäre die Auswertung der Funktion nicht möglich.
Ferner kann zwischen Einfach-Rekursion und Baumrekursion unterschieden werden, wobei bei ersterer nur eine rekursive Verwendung in der Definition auftritt, bei letzterer jedoch mindestens zwei. Bei der Baumrekursion geschehen mehrere Auswertungen simultan in verschiedenen Zweigen.
Generell kann man sich die Auswertung von rekursiven Funktionen auf verschiedene Ebenen aufgeteilt vorstellen, wobei auf der obersten Ebene begonnen wird und rekursive Funktionsanwendungen auf die darunterliegende Ebene deligiert werden. Jede Ebene stellt ihr Ergebnis der übergeordneten Ebene zur Verfügung.

►►► Woche 3 ◄◄◄

1.8 Funktionen höherer Ordnung

1.8.1 Verknüpfung von Funktionen

Für die Betrachtungen in diesem Paragraphen definieren wir zunächst eine Verschiebungsfunktion für Punkte in der Ebene. Ein Punkt lässt sich als Paar (x, y) von reellen Zahlen darstellen – 2-Tupel werden auch Paare genannt. Dabei sind x und y die Koordinaten in der xy-Ebene des üblicherweise verwendeten Koordinatensystems.
Die Funktion

$$g(x, y) = (x + 3, y - 1) \tag{19}$$

beschreibt eine Verschiebung, wobei für einen Punkt p der Funktionswert oder auch Bildpunkt $p' = f(p)$, um $+3$ entlang der x-Achse und um -1 entlang der y-Achse gegenüber p verschoben liegt. In ähnlicher Weise beschreibt die Funktion

$$(x, y) = (-x, y) \tag{20}$$

eine Achsenspiegelung an der y-Achse.
Die Funktionen g und h bilden beide Punkte auf Punkte ab, d.h. sowohl ihre Definitionsmenge als auch ihre Wertemenge ist die Menge aller Paare reeller Zahlen.
Angenommen man möchte einen Punkt p zunächst an der y-Achse spiegeln und den erhaltenen Bildpunkt entsprechend der Funktion g verschieben. Der gesuchte Punkt ist

$$p'' = g(h(p))$$

Man nennt dies die *Komposition* der Funktionen g und h [h und g]. Stattdessen ist auch die Schreibweise

$$p'' = (g \circ h)\ (p)$$

üblich.

1.8.2 Funktionen als Argumente anderer Funktionen.

In Abschnitt 1.5.4 haben wir festgestellt, dass der Operator + als Funktion von der Menge der reellen Paare in die Menge der reellen Zahlen aufgefasst werden kann: Statt $x+y$ kann man sich zur Verdeutlichung die Schreibweise $+(x, y)$ vorstellen. Entsprechend kann hier der Verknüpfungsoperator $\circ$ als Funktion verstanden werden, die ein Paar von Funktionen (f_1, f_2) auf eine neue Funktion

$$f_{neu} = \circ(f_1, f_2) = f_1 \circ f_2$$

abbildet. Statt $f_1 \circ f_2$ ist auch hier die Schreibweise $\circ(f_1, f_2)$ denkbar.
Hier treten die Funktionen f_1 und f_2 als Argumentwerte der Funktion $\circ$ auf. Funktionen, die andere Funktionen als Argumente erwarten, nennt man *Funktionen höherer Ordnung.*

Die Technik, Funktionen als Argumente anderer Funktionen zu benutzen ist insbesondere in der funktionalen Programmierung sehr wichtig. Das folgende Beispiel dient als Vorbereitung für diese Programmiertechnik.
Dazu sollen die Transformations-Funktionen g und h zu einer neuen Transformations-Funktion t verknüpft werden. Die neue Funktion ist definiert durch

$$t = g \circ h.$$

Schauen wir uns die Auswertungsschritte bei der Anwendung der neuen Funktion t auf den Punkt $p = (2, 9)$ an:

$$\begin{aligned} f(p) &= (g \circ h)(p) \\ &= g(h(p)) \\ &= g(h(2,9)) \\ &= g(-2,9) \\ &= (1,8) \end{aligned}$$

Im nächsten Abschnitt befassen wir uns mit einem anderen Fall, in dem die Verwendung von Funktionen als Argumente äußerst nützlich ist.

1.8.3 Warum es nützlich ist...

(19) und (20) definieren eine Verschiebungsfunktion g respektive eine Achsenspiegelung h. Anstatt diese Funktionen auf nur einen Punkt auf einmal anzuwenden, kann es sinnvoll sein, eine ganze Menge von Punkten zu verschieben oder zu spiegeln. Man könnte nun separate Funktionen definieren; sowohl für die Transformation von einzelnen Punkten als auch für Punktmengen. Dann würden allerdings stets zwei Funktionen für jede Transformation benötigt.
Es ist allerdings möglich, eine Funktion zu definieren, die dieses Problem eleganter löst: Stellen wir uns eine natürliche Zahl n und eine (endliche) Menge $M = \{x_1, \ldots, x_n\}$ von n Elementen vor. Die Funktion m soll eine Funktion f auf jedes Element von M anwenden und soll die Menge der Funktionswerte als Ergebnis haben. Dabei ist es wichtig, dass M Teilmenge der Definitionsmenge von f ist, damit f für alle Elemente in M definiert ist.

$$m(f, \{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$$

Auch hier tritt eine Funktion als Argument einer anderen Funktion auf. Der Vorteil besteht darin, dass sich jede beliebige Funktion auf die Elemente von Mengen anwenden lässt. Im Falle der Punktmengen und Transformationen muss nun nur noch eine Transformations-Funktion für einzelne Punkte definiert werden. Die Variante für Mengen kann aus der Version für einzelne Punkte und m zusammengesetzt werden.

Bemerkung: *Wenn M nicht Teilmenge der Definitionsmenge D einer Funktion f ist, dann ist $m(f, M)$ nicht definiert. Denn dann existiert in M ein Element e, das nicht in*

D enthalten ist. Da $e \notin D$ ist $f(e)$ nicht definiert. Allerdings müßte $f(e)$ in $m(f, M)$ enthalten sein.
Beispielsweise könnte man die Inkrementierungsfunktion $f_{inc}(x) = x + 1$ mit der Definitionsmenge $D = \mathbb{R}$ als Beispiel anschauen. Für die Menge $M_1 = \{1, 2, 3\}$ ist $m(f_{inc}, M_1)$ definiert, da $M_1 \subset D$. Dagegen gilt dies für die Menge $M_2 = \{1, 2, 3, \spadesuit\}$ nicht, denn $\spadesuit \notin \mathbb{R}$ also auch $\spadesuit \notin \mathbb{R}$.

$$m(f_{inc}, M_2) = m(f_{inc}, \{1, 2, 3, \spadesuit\}) = \{f_{inc}(1), f_{inc}(2), f_{inc}(3), f_{inc}(\spadesuit)\} = \{2, 3, 4, \spadesuit + 1\}$$

Dabei ist $\spadesuit + 1$ ist offenbar nicht definert.

1.8.4 λ-Abstraktion und anonyme Funktionen

Bisher wurden Funktionen, bei denen der Funktionswert als Ausdruck formuliert war, stets mit einem Namen versehen, wie zum Beispiel bei $f(x) = x^2$. Es kann allerdings hinderlich sein, für jede Funktion einen neuen Namen zu vergeben, ganz besonders dann, wenn viele verschiedene Funktionen gebraucht werden.
Funktionen können *anonym* definiert werden, indem sie als λ-Abstraktion geschrieben werden.

21 Definition Allgemein wird eine Funktion folgendermaßen als λ-*Abstraktion* geschrieben:

$$\lambda x.A$$

Dabei ist x der *Parameter* und A der *Funktionskörper*.

22 Definition Bei der *Anwendung* der Funktion

$$(\lambda x.A)\ (B) = A'$$

wird das Argument B an den Parameter x gebunden. Der Funktionswert A' wird erhalten, indem für jedes Vorkommen des Parameters x im Funktionskörper A das Argument B eingesetzt wird.

Die Normalparabel f_{par} als λ-Abstraktion sieht zum Beispiel so aus:

$$f_{par} = \lambda x.x^2$$

Hier die Anwendung auf das Argument 7:

$$f_{par}(7) = (\lambda x.x^2)7 = 7^2$$

Die Verschiebungsfunktion g aus (19) kann ebenfalls als λ-Ausdruck formuliert werden:

$$g = \lambda p.(f_x(p) + 3, f_y(p) - 1)$$

Dabei werden die Funktionen f_x und f_y lediglich dazu benötigt, die x- respektive y-Koordinate eines Punktes zu extrahieren.[6]

[6] In modernen Programmiersprachen wurden λ-Ausdrücke so erweitert, dass dieser Umstand nicht mehr gegeben ist. Hier soll allerdings auf diese Erweiterung verzichtet werden; insbesondere weil sie in LISP nicht existiert.

1.8.5 Zusammenfassung: Funktionen höherer Ordnung

Funktionen höherer Ordnung sind ein wichtiges Werkzeug in der funktionalen Programmierung. Sie werden insbesondere zur Konstruktion komplexer Funktionen aus anderen (einfacheren) Funktionen verwendet. Dabei ist es in vielen Fällen hilfreich, anonyme Funktionen mit Hilfe der λ-Abstraktion zu definieren.
Bis hierher haben wir nur die formalen Grundlagen betrachtet. Wie diese Techniken in der Programmiersprache LISP verwendet werden, wird im zweiten Teil erläutert.

►►► Woche 4 ◄◄◄

2 Datenstrukturen und Manipulation

2.1 Daten und ihre Organisation

2.1.1 Motivation: Datenverarbeitung

Computer wurden entwickelt um Daten automatisch und schnell zu verarbeiten. Neben dem heute oft verwendeten Begriff *Informatik* sprach man vor allem früher von *elektronischer Datenverarbeitung (EDV)*. Der eigentliche Vorgang wird durch den alten Begriff sehr gut ausgedrückt: Daten werden elektronisch verarbeitet. Das zweifellos wichtigste elektronische Datenverarbeitungsgerät ist heute der Computer. Er zeichnet sich insbesondere dadurch aus, dank seiner Programmierbarkeit besonders flexibel zu sein. Anstatt (wie früher) unterschiedliche Maschinen für unterschiedliche Berechnungen zu haben (Addiermaschine, Rechenschieber, Logarithmentafeln...), ist man heute in der Lage mit ein und derselben Maschine beliebige Verarbeitungs-Aufgaben durchzuführen. Datenverarbeitung heißt, gezielt elementare Manipulationen an den Daten vorzunehmen, so dass dadurch das gewünschte Ergbnis erzielt wird. Dabei werden meistens Daten aus einer Form in eine andere überführt. Beispielsweise ist das Übersetzen eines Textes Datenverarbeitung. Der Ursprungstext wird dabei aus einer Form, die der Ursprache, in eine andere Form übersetzt, die der Zielsprache. Eine Liste von Personen alphabetisch zu ornden ist ein anderes Beispiel für Datenverarbeitung.
In der elektronischen Datenverarbeitung geschieht im Grunde nichts Anderes; allerdings müssen die Daten zur maschinellen Verarbeitung in einer geeigneten, maschinenlesbaren Form gespeichert werden. Als Vorbereitung werden sich die nächsten Abschnitte kurz mit der Organisation von Daten allgemein befassen, um dann die Organisationsformen kennenzulernen, die in der elektronischen Datenverarbeitung von Bedeutung sind.

2.1.2 die Rolle der Organisationsform

Die Wahl der Organisationsform von Daten hängt von der Aufgaben ab und ist für eine effiziente Ausführung der Aufgabe entscheidend.
Ein Lexikon zum Beispiel enthält alle Stichworte nach einer allgemein bekannten Ordnung, so dass ein schnelles Auffinden der Stichworte möglich ist. Karteikarten werden verwendet, wenn häufig neue Daten hinzukommen oder entfernt werden und der Datenbestand stets zugreifbar sein muss. Dagegen eignet sich für ein Lehrbuch eine thematische Anordung, in Archiven wird chronologisch vorgegangen usw.

2.1.3 Datenstrukturen

Der Begriff *Datenstruktur* bezieht sich einerseits auf die Art und Weise, wie bestimmte Daten gespeichert sind. Aus der Repräsentationsform ergibt sich, welche Operationen

mit Daten dieser Datenstruktur effizient möglich sind.
Eine Analogie bietet das Beispiel von Buch und Karteikarte: Ein Buch besteht aus gebundenen Seiten, während Karteikarten lose abgelegt sind. Daraus ergibt sich, dass bei Karteikarten leicht weitere Daten hinzugefügt oder entfernt werden können. Bei Karteikarten ist so etwas wie eine Operation *Einfügen* möglich, nicht jedoch beim Buch. Andererseits gibt es auch Datenstrukturen, die nicht durch die physische Repräsentationsform charakterisiert werden, sondern durch die Operationen, die sie unterstützen. Dabei ergeben sich die nötigen Operationen aus der Aufgabenklasse, für die eine solche Datenstruktur eingesetzt werden soll. Meistens sind es elementare Aufgabenklassen, die in sehr unterschiedlichen Bereichen immer wieder vorkommen.

2.1.4 Zusammenfassung: Datenstrukturen

Verarbeitung von Daten bedeutet gezielte Manipulation von Daten, in einer Weise, die schließlich zum gewünschten Ergebnis führt. Dabei spielt die Organisationsform der Daten eine entscheidende Rolle. Im Bereich der elektronischen Datenverarbeitung spricht man von *Datenstrukturen*. Einerseits bezieht sich dieser Begriff auf die physische Repräsentation, d.h. bei Computern immer Speicherung der Daten. Von der physischen Organisation der Daten hängt es ab, welche Operationen effizient unterstützt werden. Zum anderen kann sich der Begriff Datenstruktur auch auf Strukturen beziehen, die für eine typische Aufgabe benötigt werden. Dann interessiert nicht die physische Organisation, sondern ein Satz bestimmter Operationen, der für diese Aufgabe benötigt wird.

2.2 Datenstrukturen im Einzelnen

2.2.1 Graphen

Graphen werden eigentlich in der Mathematik verwendet, eignen sich aber zur allgemeinen Beschreibung vieler Datenstrukturen, da diese Spezialfälle von Graphen sind. Daher werden wir zunächst Graphen kennenlernen, um darauf aufbauend die anderen Datenstrukturen durch weitere Einschränkungen zu beschreiben.
Ein Graph ist gegeben durch eine Menge von Knoten und einer Menge von Kanten. Jede Kante verbindet zwei Knoten. Bildlich lassen sich Graphen darstellen, wie es Abbildung 10 zeigt. Der linke Graph kann zum Beispiel auch durch (E_l, K_l, i_l) ausgedrückt werden, wobe $E_l = \{e_1, e_2, e_3, e_4\}$, $K_l = \{k_1, k_2, k_3, k_4, k_5\}$ und

$$\begin{aligned} i_l(k_1) &= (e_1, e_1), \\ i_l(k_2) &= (e_1, e_2), \\ i_l(k_3) &= (e_2, e_3), \\ i_l(k_4) &= (e_2, e_4), \\ i_l(k_5) &= (e_3, e_4). \end{aligned}$$

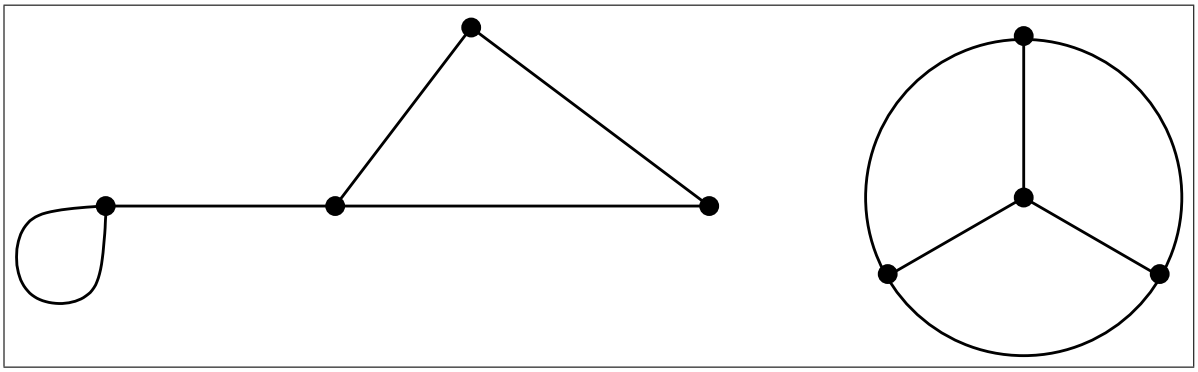

Abbildung 10: Bildliche Darstellung zweier Graphen. Die Punkte stellen Knoten dar, die Verbindungslinien Kanten.

23 Definition Ein Graph G ist gegeben durch ein 3-Tupel

$$G = (E, K, i),$$

wobei E die Menge der Ecken oder *Knoten*, K die Menge der *Kanten* und i die *Inzidenzfunktion*. i ordnet jeder Kante $u \in K$ ein Tupel (x_i, x_j) zu, wobei $x_i, x_j \in E$ die beiden Knoten sind, die durch die Kante verbunden sind. Man nennt x_i und x_j auch die Anfangs- bzw. Endknoten der Kante u.

Bisher hat eine Kante keine Richtung, d.h. insbesondere, dass der Begriff Anfangsknoten synonym mit dem Begriff Endknoten gebraucht wird, es sei denn, wir beziehen uns auf dieselbe Kante.
Es gibt etliche interessante Eigenschaften und Spezialfälle von Grafen, von denen einige hier erläutert werden sollen.

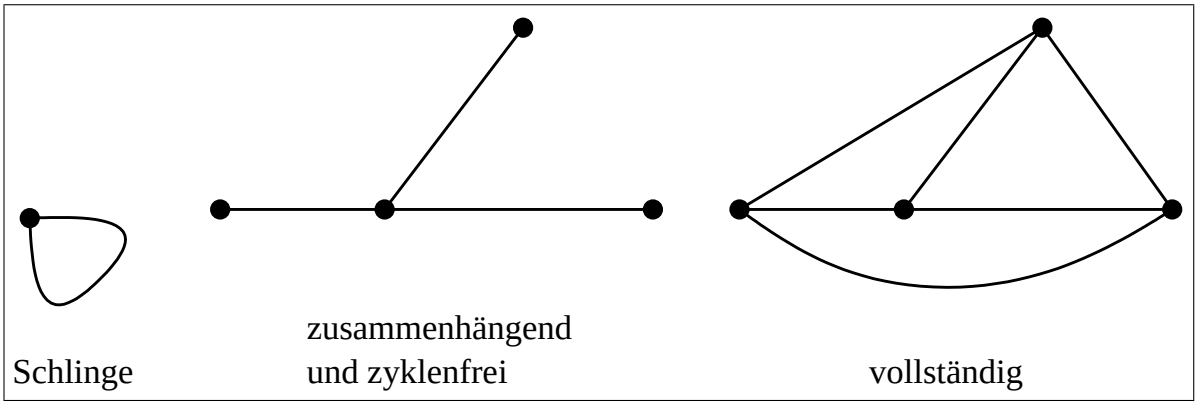

Abbildung 11: Einige Graphen und ihre Eigenschaften.

Weg. Als *Weg* oder als Kantenfolge wollen wir eine Folge von benachbarten Kanten bezeichnen; zwei Kanten sind *benachbart*, wenn ein gemeinsamer Endknoten existiert. *Geschlossen* ist ein Weg, falls Anfangs- und Endpunkt zusammenfallen. Schließlich ist ein *Kreis* ein geschlossener Weg, bei dessen Durchlaufen jede Ecke außer dem Anfangs- und Endpunkt nur einmal besucht wird. Ein *einfacher* Weg oder ein Kantenzug ist ein Weg, bei dessen Durchlaufen jede Kante nur einmal passiert wird.

Schlinge. Eine *Schlinge* ist eine Kante, deren Anfangs- und Endknoten identisch ist.

Zyklenfrei. Ein Graph heißt *zyklenfrei*, wenn es keinen geschlossenen Weg im Graphen gibt. Insbesondere gibt es keine Schlingen.

Zusammenhängend. Ein Graph heißt *zusammenhängend*, wenn man von jedem beliebigen Eckpunkt via die Kanten nach jedem beliebigen zweiten Punkt gelangen kann.

Vollständig. Ein Graph heißt *vollständig*, wenn je zwei (d.h. zwei beliebig ausgewählte) Eckpunkte durch genau eine Kante verbunden sind. Insbesondere gibt es keine Schlingen. Ein vollständiger Graph ist immer zusammenhängend, wie man sofort einsieht.

Gerichtet. Ein *gerichteter* ist ein Graph, bei dem anstatt der Kanten Pfeile (engl. vertex) verwendet werden. Seien k_i und k_j Knoten eines Graphen. Das Tupel (k_i, k_j), das im ungerichteten Graph die Kante zwischen k_i und k_j repräsentiert, repräsentiert im gerichteten Graphen einen Pfeil, der bei k_i beginnt und bei k_j endet.

2.2.2 Bäume

Ein *Baum* ist ein zyklenfreier, zusammenhängender Graph mit einem ausgezeichneten Knoten, der *Wurzel*. Abbildung 12 zeigt einen Baum in der bildlichen Darstellung, die im vorigen Abschnitt auch bereits für Graphen verwendet wurde.
Dabei sind diejenigen Knoten eines Baumes, bei denen nur eine Kante beginnt und die nicht die Wurzel sind, die *Blätter* des Baumes. Alle Knoten, die weder Wurzel noch Blätter sind, heißen innere Knoten. An ihnen beginnen stets mindestens zwei Kanten. Die blattwärts liegenden Nachbarknoten eines Knotens bezeichnet man als die *Nachfolger* oder *Kinder* des Knotens, den wurzelwärts liegenden als den *Vorgänger* oder *Elternknoten*.
Dadurch, dass die Wurzel festgelegt ist, ist in gewisser Weise eine Richtung im Baum festgelegt, die im Diagramm auch durch die Anordnung der Knoten verdeutlicht wird: die Blätter unten und die Wurzel oben in der Mitte. Das ist die übliche Darstellungsweise von Baumstrukturen – wahrscheinlich weil man beim Zeichnen stets mit der Wurzel oben auf dem Blatt beginnt.
Als die *Ordnung* eines Baumes bezeichnet man die maximale Anzahl von Nachfolgern, die ein Knoten des Baumes hat. Die *Höhe* eines Baumes ist die Anzahl der Kanten, die zwischen der Wurzel und dem am weitesten entfernten Blatt liegen. Die Höhe kann

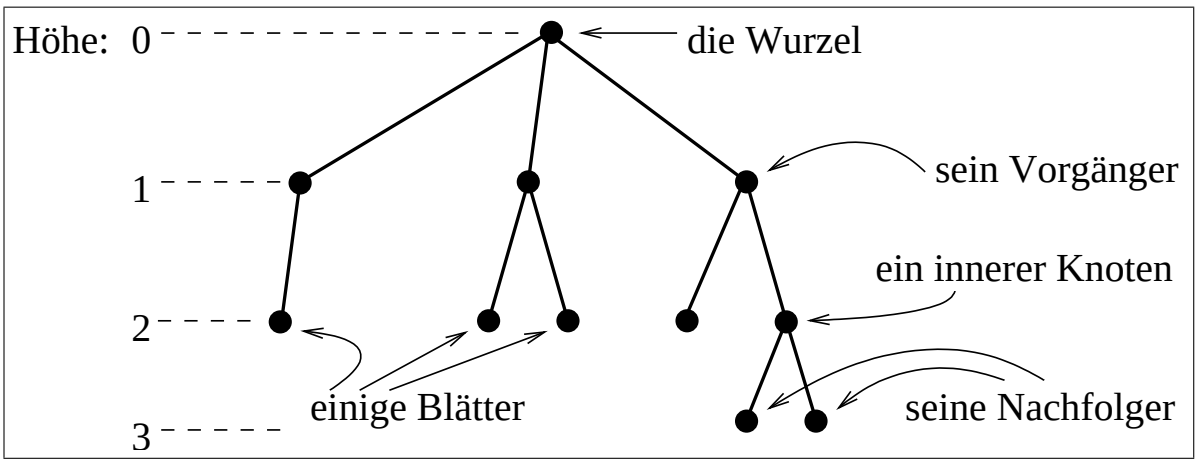

Abbildung 12: Bildliche Darstellung eines Baumes

zusammen mit der Ordnung eines Baumes dazu genutzt werden, die maximale Knotenanzahl zu bestimmen. Aus diesem Grund sind diese beiden Größen in der Informatik wichtige Charakteristika eines Baumes. Für uns ist vor allem die Höhe interessant, weil sie im Baum eines CONS-Diagramms die Verschachtelungstiefe der dargestellten Listenstruktur angibt.
Zuletzt soll der Vollständigkeit halber noch eine formalere Definition für Bäume gegeben werden. Sie ist aus rein formaler Sicht nicht sauber, da Begriffe wie Nachfolger und Wurzel nicht definiert werden. Dennoch soll dies an dieser Stelle nicht formal geschehen, da ein großer formaler Apparat benötigt würde, der jedoch wenig zum Verständnis beiträgt.

24 Definition
Ein Baum der Ordnung d ist rekursiv definiert durch:

1. Der leere Baum ist ein Baum der Ordnung d.
2. Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung d. Die Höhe h ist 0.
3. Sind $t_1, \ldots, t_d$ beliebige, disjunkte Bäume der Ordnung d, so erhält man einen (weiteren) Baum der Ordnung d, indem man die Wurzeln von $t_1, \ldots, t_d$ zu Nachfolgern einer neu geschaffenen Wurzel w macht. Die Höhe h des neuen Baums ist dann $\max\{h(t_1), \ldots, h(t_d)\} + 1$.

2.2.3 Listen

Von ihrer Struktur her sind Listen Bäume der Ordnung 1. Daraus und aus der Tatsache, dass ein Baum zusammenhängend ist, ergibt sich eine kettenartige Struktur. Die Liste beginnt an der Wurzel und endet am einzigen Blatt des Baumes. Wie in Abbildung 13 zu sehen wird bei der Darstellung von Listen nicht die Wurzel oben plaziert, sondern links. Außerdem spricht man nicht mehr von Wurzel und Blatt.

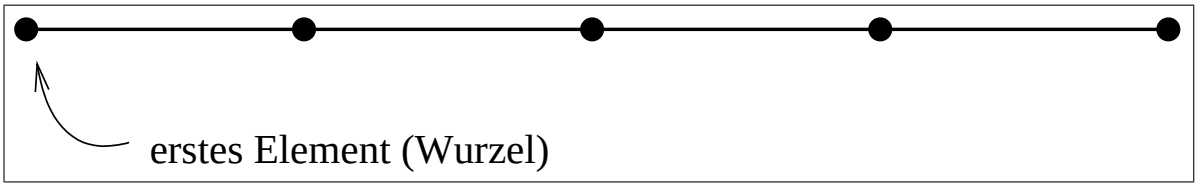

Abbildung 13: Eine Liste lässt sich als Baum der Ordnung 1 auffassen.

Wegen ihrer unverzweigten Struktur, in der jedes Element genau einen Nachfolger hat (mit Ausnahme des letzten), werden Listen auch als *linear* bezeichnet.

2.2.4 Speicherung von Daten in Bäumen und Listen

Bisher haben wir nur verschiedene Strukturen beschrieben, ohne uns Gedanken darüber zu machen, wie diese zur Datenrepräsentation genutzt werden können. Der eigentliche Sinn von Listen und Bäumen ist nicht die Repräsentation der reinen Struktur, sondern die geschickte Organisation bei der Speicherung von Daten.
Dazu verwendet man Bäume und Listen derart, dass jeder Knoten ein Datum speichern kann. Zur Speicherung von Namen in einer Listenstruktur würde jeder Knoten einen Namen speichern (Abbildung 14). Im Falle von Listen bezeichnet man jedes gespeicherte Datum als ein *Element* der Liste.
Bäume eignen sich dann besonders, wenn man eine hierarschische Organisation repräsentieren will. So könnte der Name des obersten Leiters einer Organisation in der Wurzel gespeichert werden, seine direkten Untergebenen in den Nachfolgerknoten, deren Untergebene wieder in deren Nachfolgern...

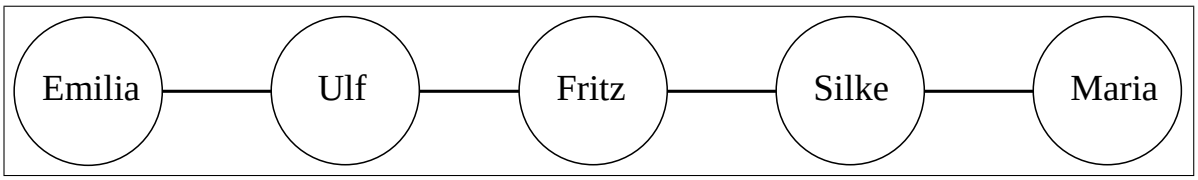

Abbildung 14: Eine Liste, die zur Speicherung von Namen verwendet wird.

Hierbei wird erneut deutlich, dass die Wahl der Repräsentationsform wichtig ist. Zwar ließen sich auch Hierarchien in Listen verwalten, jedoch sind Bäume dafür eindeutig besser geeignet. Unter bestimmten Voraussetzungen ist es sinnvoll, auch zur Speicherung von listenartigen Daten Bäume zu verwenden. Diese Technik führt an dieser Stelle jedoch zu weit und wird nicht in Betracht gezogen.
Ein wichtiger Aspekt bei der Speicherung von Daten in Bäumen und Listen ist, dass die in den Knoten gespeicherte Information nur dann gelesen oder geändert werden kann, wenn man sich von der Wurzel her zu dem betreffenden Knoten durchhangelt. Das hängt mit der normalerweise verwendeten physikalischen Repräsentation von Listen und Bäumen zusammen.

2.2.5 Eigenschaften von Listen

Für die Angabe konkreter Listen werden in der Programmierung verschiedene Schreibweisen verwendet. Wir wollen hier nur die in LISP verwendete kennenlernen – große Unterschiede zwischen verschiedenen Programmiersprachen gibt es ohnehin nicht. Die Liste, die in Abbildung 14 bildlich dargestellt ist, würde in LISP-Schreibweise so geschrieben.

```
(Emilia Ulf Fritz Silke Maria)
```

Ähnlich wie es den leeren Baum gibt (vgl. (24)), gibt es auch die leere Liste:

```
()
```

Listen können wie Mengen oder Tupel beliebig tief verschachtelt werden. Wie bei Tupeln ist die Reihenfolge der Elemente von Bedeutung – wie es die kettenartige Struktur bereits nahelegt – es können auch Elemente mehrfach genannt werden.

```
(() 1 Fritz (Maria) (Maria Ulf) Fritz)
```

2.2.6 CONS-Diagramme

Eine Notation, die sich an den technischen Gegebenheiten der Listenrepräsentation in LISP orientiert ist das CONS-Diagramm. Listen werden in LISP als miteinander verknüpfte CONS-Zellen repräsentiert. Wir wollen an dieser Stelle noch nicht fragen, was CONS-Zellen eigentlich seien. Es geht hier vorerst nur um die Einführung einer Notation, die dieser internen Repräsentation gerecht wird und später dazu dienen wird, bei der Verarbeitung von Listen zu helfen.

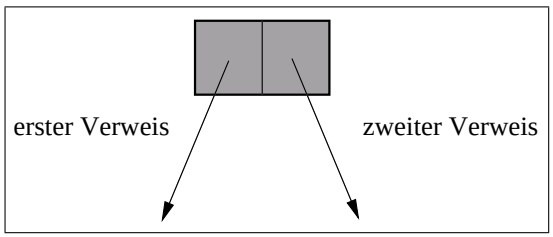

Abbildung 15: Eine CONS-Zelle mit ihren beiden Verweisen.

Jede Cons-Zelle hat zwei Verweise zu anderen Daten (wie etwa Zahlen, Texte, andere CONS-Zellen). Abbildung 15 zeigt, wie eine CONS-Zelle in dieser Diagrammtechnik dargestellt wird. Listen werden in LISP durch Verkettung mehrerer CONS-Zellen repräsentiert. Dabei verweist jede CONS-Zelle mit dem zweiten Verweis auf eine Nachfolgerin, die in gleicher Weise auf die nächste verweist, wie in Abbildung 16 zu sehen ist. Der erste Verweis jeder CONS-Zelle ist dabei bis jetzt unbenutzt. Er soll auf die Elemente der Liste verweisen. Die letzte CONS-Zelle in der Kette hat keine Nachfolgerzelle – so wie das letzte Element auch keinen Nachfolger hat. Anstatt auf eine Liste mit weiteren Elementen (CONS-Zellen) zu verweisen, verweist die letzte Zelle auf die leere Liste. Sie wird in Lisp mit

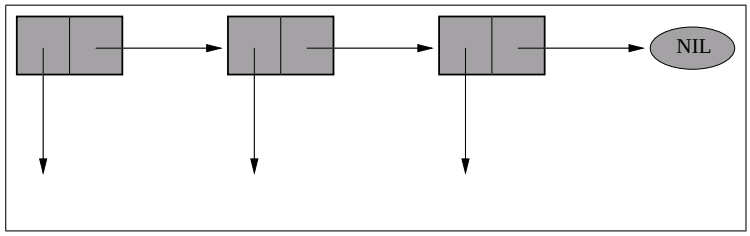

Abbildung 16: Eine Liste kann durch verkettete CONS-Zellen repräsentiert werden.

`NIL`

bezeichnet.

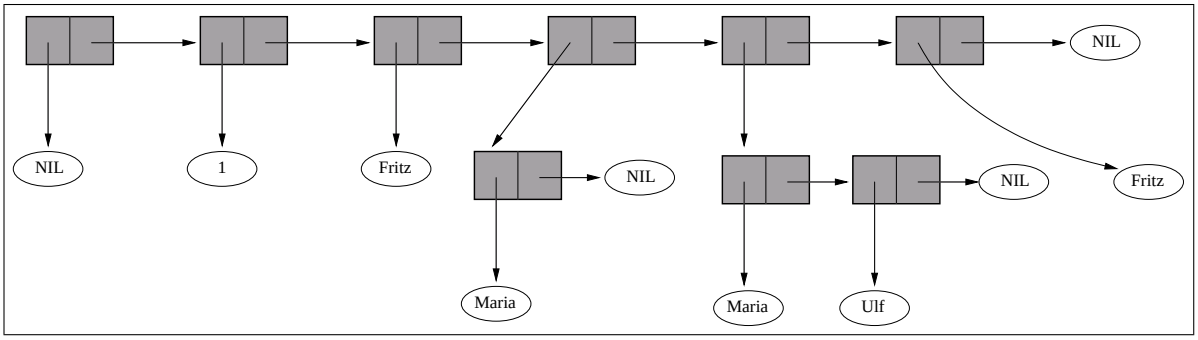

Abbildung 17: Eine verschachtelte Liste als CONS-Diagramm.

Die Liste

```
(() 1 Fritz (Maria) (Maria Ulf) Fritz)
```

ist in Abbildung 17 als CONS-Diagramm gezeigt. Dabei ist durch die unterschiedlichen horizontalen Positionen im Diagramm die Verschachtelung gut zu erkennen. Die Listen in der unteren „Etage“ sind dabei Elemente der übergeordneten Liste, wie etwa die Elemente 1 oder *Fritz*. Das entspricht dem Gebrauch bei Mengen und ihren Elementen.

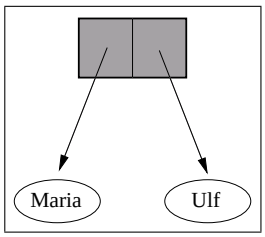

Abbildung 18: Eine Struktur dieser Art heißt *schmutzige Liste*, weil der zweite Verweis nicht auf eine Liste zeigt.

2.2.7 Arrays

Wie am Ende des Abschnitts 2.2.4 erläutert wurde haben Bäume und Listen einen entscheidenden Schwachpunkt: Man kann nicht direkt auf jedes Element zugreifen. Das ist bei großen Datenbeständen für bestimmte Aufgaben nicht akzeptabel.
Ein Array ist eine andere lineare Datenstruktur, die dieses Problem nicht hat. Ein *Array* zeichnet sich dadurch aus, dass es eine im Vorhinein festgelegte Anzahl von Elementen enthält, und die Elemente alle dieselbe Struktur haben, d.h. insbesondere gleich lang sind. Da in einem Array alle Elemente direkt aneinander angrenzen kann jedes Element über seine Position im Array direkt gefunden werden: Für das n-te Element ist klar, an welcher Position es ist.
Der besseren Handhabung wegen sind alle Positionen eines Arrays mit Ordnungsnummern 1 bis $n \in \mathbb{N}$ versehen – jedes Element im Array ist über die Ordnungsnummer der Position, an dem es gespeichert ist, seinen *Index*, zu erreichen. Der Index ist eine leicht benutzbare Möglichkeit, physische Positionsangaben zu machen.
Man erreicht ein bestimmtes Element stets über die Angabe seines Index.
Das ist vergleichbar mit einer Bibliothek mit einem sehr langen Regal, in dem ein vielbändiges Lexikon steht. Angenommen man benötigt den soundsovielten Band und man weiß, wieviel Platz jeder Band im Regal benötigt, dann kann man direkt an die Stelle im Regal laufen, ohne alle Buchrücken anzusehen.

2.2.8 Stack und Queue

Stack und *Queue* sind zwei Datenstrukturen, die sich nicht primär durch ihre physikalische Organisation auszeichnen, sondern durch bestimmte Operationen, die für viele Aufgabentypen wichtig sind.
Eine Queue organisiert die Elemente wie in einer Warteschlange. Elemente werden nur am „Ende“ der Schlange angefügt und am „Anfang“ gelesen oder entfernt. Das bedeutet, dass das zuerst hinzugefügte Element auch als erstes wieder entfernt werden wird. Man spricht daher auch vom *first-in-first-out*-Prinzip oder auch von *FIFO*.
Dagegen ist bei einem *Stack* das zuerst eingefügte Element dasjenige, das zuletzt gelesen und gelöscht wird. Das entspricht dem Tellerwärmer in der Mensa, bei dem stets der oberste Teller entnommen werden kann aber auch nur oben neue Teller auf den Stapel gestellt werden können. Man spricht hier vom *first-in-last-out*-Prinzip oder auch von *FILO*.
Diese beiden Datenstrukturen unterstützen nur die Operationen, die bei ihrer Verwendung üblicherweise nötig sind: Hinzufügen eines Elements mit *push*, Auslesen und anschließendes Entfernen des vordersten bzw. obersten Elementes mit *pop*, Überprüfen der Elementanzahl, *empty*. Andere Operationen sind nicht vorgesehen und für die typischen Aufgaben von Queues und Stacks auch nicht nötig. Diese beiden Datenstrukturen sind daher eher als Kategorien denn als physische Organisationsstruktur zu sehen.

2.2.9 Zusammenfassung: Datenstrukturen im Einzelnen

Nachdem Graphen eingeführt wurden, lernten wir zwei durch Graphen gut beschreibbare Datenstrukturen kennen: Bäume und Listen. Sie zeichnen sich dadurch aus, dass sie durch miteinander verbundenen Knoten repräsentiert werden, wobei jeder Knoten ein Datenelement speichert sowie ein oder mehrere Verbindung zu anderen Knoten. Um ein bestimmtes Element zu erreichen – sei es zum Abrufen oder zum Bearbeiten – muss man stets den Weg über die Verbindungen von anderen Knoten zu dem betreffenden Knoten benutzen. Begonnen wird bei der Wurzel eines Baumes bzw. beim ersten Element einer Liste.
Arrays unterstützen dagegen direkte Zugriffe auf alle Elemente über Indizes, die mit den Positionen der Elemente im Array identifiziert sind. Allerdings müssen alle Elemente des Arrays von gleicher Struktur, d.h. insbesondere gleich lang sein.
Zuletzt wurden die Strukturen Stack und Queue vorgestellt, bei denen es nicht um ihre interne Struktur geht, sondern um die Bewältigung prototypischer Verwaltungs-Aufgaben.

►►► Woche 5 ◄◄◄

2.3 Suche

2.3.1 Suche in Listen

Nehmen wir nun an, wir möchten feststellen, ob sich ein bestimmter Name in einer Liste befindet oder nicht. Bei 5 Elementen, wie in Abbildung 14 kann man als Mensch einfach ein Blick darauf werfen. Geht es aber um Tausende von Einträgen, ist dies nicht mehr praktikabel. Man sollte nach einem System vorgehen, das garantiert, dass der Name gefunden wird, wenn er als Element in der Liste vorkommt, und bei Nichtvorhandensein dies ebenfalls zweifelsfrei feststellt.
Bei Listen bietet es sich an, bei dem ersten Element zu beginnen, zu prüfen, ob es sich um den gesuchten Namen handelt. Falls es nicht der gesuchte Name ist, geht man zum nächsten Element. Damit fährt man fort, bis man das Ende der Liste erreicht hat.
Bei diesem Suchverfahren bewegt man sich nur entlang der Kanten der Listenstruktur. Neben der Notwendigkeit systematisch zu suchen ist es bekanntlich auch gar nicht möglich, irgendein beliebiges Element direkt anzuspringen. (vgl.2.2.4)

2.3.2 Suche in Bäumen

Einen Baum systematisch zu durchsuchen ist etwas komplizierter als eine Liste. Es gibt im Grunde zwei mögliche Vorgehensweisen, die *Breitensuche* und *Tiefensuche* heißen.[7] Bei beiden Verfahren beginnt die Suche an der Wurzel.

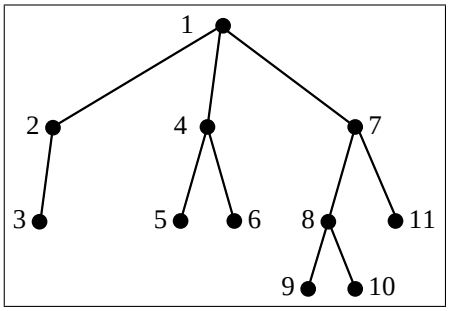

Abbildung 19: Tiefensuche

[7]Wir werden hier nicht auf Bäume eingehen, bei denen die Elemente so abgelegt sind, dass an jedem besuchten Knoten fest steht, in welchem Ast das gesuchte Element zu finden ist. Diese Form der Organisation von Bäumen ist für die Informatik von großer Bedeutung – für die künstliche Intelligenz und die Kongnitionswissenschaft ist das systematische Absuchen der Baumstruktur von größerer Bedeutung.

Tiefensuche Bevor die Schwestern eines Knotens untersucht werden, werden seine Kinder von links nach rechts untersucht. Das erste besuchte Blatt ist also das ganz links liegende. In Abbildung 19 ist die Reihenfolge, in der die Knoten geprüft werden, durch Zahlen dargestellt.

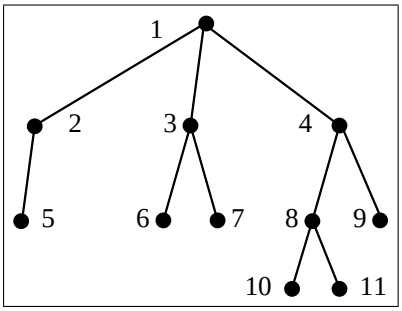

Abbildung 20: Breitensuche

Breitensuche Bei der Breitensuche wird in einem Suchbaum zuerst jeder Knoten auf derselben Ebene konsultiert, bevor auf die nächste Ebene vorgerückt wird. Auf jeder Ebene wird von links nach rechts vorgegangen. Abbildung 20 stellt die Besuchs-Reihenfolge der Knoten durch Zahlen dar.

2.3.3 Suche in Arrays

Die Möglichkeit, direkt über den Index zu jedem beliebigen Element springen zu können, bringt bei ungeordneten Daten keine Zeitersparnis. Es bleibt nur das systematische Durchprobieren aller Indizes, bis man beim gewünschten Element angelangt ist. Gegenüber dem Durchhangeln bei Listen ist also nichts gewonnen.
Ist der Datenbestand aber sortiert, kann man erhebliche Zeitgewinne erzielen: Man beginnt in der Mitte des Arrays also bei dem Index, der halb so groß ist, wie der größte Index im Array. Wenn eine grade Anzahl von Elementen vorliegt, gibt es kein mittleres Element – in diesem Falle rundet man auf eine ganze Zahl. Wenn das mittlere Element das Gesuchte ist, ist die Suche zu Ende. Wenn nun das gesuchte Element kleiner ist, als das mittlere, wählt man die Hälfte, in dem nur kleinere Elemente als das mittlere Element sind. Andernfalls wählt man die anderen Hälfte, in der nur größere Elemente als das mittlere sind. In der gewählten Hälfte sucht man nach dem gleichen Verfahren weiter, bis man bei dem Element angekommen ist oder der durchsuchte Bereich nicht mehr weiter geteilt werden kann. Dieses Verfahren muss wesentlich weniger Elemente anschauen, um ein bestimmtes Element zu finden oder seine Abwesenheit festzustellen. Jetzt, da wir um den Nutzen sortierter Datenbestände wissen, werden wir uns in den folgenen Abschnitte mit der Sortiereung unsortierter Datenbestände widmen.

2.3.4 Zusammenfassung: Suche

Im Zusammenhang mit der Suche in verschiedenen Strukturen wird deutlich, wie stark sich unterschiedliche Repräsentationsformen auf die Durchführung bestimmter Datenverarbeitungsaufgaben auswirken. Deutlich wurde auch, dass je nach Struktur andere Vorgehensweisen nötig waren um die auf abstrakterer Ebene gleiche Operation *Suche* zu realisieren. In vielen Fällen ist es sogar möglich, verschiedene Vorgehensweisen für die gleiche Datenstruktur wählen, wie wir an Breitensuche bzw. Tiefensuche bei Bäumen gesehen haben.
Aus der Notwendigkeit, abhängig von der Datenstruktur bestimmte Operation unterschiedlich zu realisieren, ergeben sich auch Unterschiede in der Effizienz verschiedener Strukturen bei der gleichen Operation.

2.4 Sortierung

2.4.1 Sortierkriterium

Notwendige Voraussetzung der Sortierung von Datenbeständen ist die Existenz eines Sortierkriteriums, bezüglich dessen für beliebige zwei Elemente des Datenbestandes feststeht, welches der beiden größer ist oder ob sie gleich groß sind.
Im Falle eines Lexikons wird die lexikographische Ordnung als Kriterium verwendet; im Falle von Zeitpunkten ihre zeitliche Abfolge. Oftmals ist uns das Kriterium so in Fleisch und Blut übergegangen, dass wir es nicht als separates Konstrukt wahrnehmen, sondern es schlicht als Eigenschaft der Elemente auffassen. Personen lassen sich aber nach ihrem Namen, ihrem Alter, ihrer Körpergröße etc. sortieren. All das sind unterschiedliche Kriterien, nach denen eine Reihenfolge festgelegt ist.

2.4.2 Sortieralgorithmen

Es gibt nun eine Reihe von Möglichkeiten, Datenbestände nach irgendeinem Kriterium zu sortieren. Aus Sicht der Sortierung spielt es keine Rolle, welches Kriterium verwendet wird. Der Sortieralgorithmus kümmert sich darum, welche Elemente zu vergleichen sind und wie sie umsortiert werden müssen.
Was ein Algorithmus genau ist wird in Abschnitt 3.1 genauer erörtert. Für den Moment genügt es, Algorithmus synonym mit Rechenverfahren zu verstehen.

2.4.3 Bubblesort

Der Name „Bubblesort Bubblesort“ rührt von der bildhaften Vorstellung her, dass bei diesem Verfahren leichte Elemente (= kleine Werte) wie Blasen in einer Flüssigkeit nach oben (= vorne im Datenbestand) steigen lässt. Bubblesort beruht auf der folgenden Idee: „Der Datenbestand wird vom Anfang zum Ende durchkämmt und die Elemente dabei paarweise verglichen.“ Wenn beide in der richtigen Reihenfolge stehen (das kleinere zuerst, das größere hinterher), dann wird mit dem nächsten Paar fortgefahren. „Wenn beide in der falschen Reihenfolge stehen, werden sie zuerst vertauscht. Das größte

Element wandert dabei an das Ende des Datenbestandes.“ Dann wird der Prozess wiederholt, aber ohne das letzte Element. Im nächsten Durchgang werden die letzten beiden Elemente ausgelassen usw. Der Datenbestand ist vollständig sortiert, wenn nur noch ein Element zu „sortieren“ ist.

2.4.4 Quicksort

Quicksort ist ein anderes Sortierverfahren, das sich besonders durch seine hohe Effizienz auszeichnet, d.h. im Durchschnitt benötigt Quicksort deutlich weniger Zeit als Bubblesort, um denselben Datenbestand zu sortieren.
Quicksort geht dabei folgendermaßen vor:

1. Wenn der Datenbestand nicht mehr als ein Element enthält, ist er bereits sortiert und die folgenden Schritte sind nicht auszuführen.
2. Wähle ein beliebiges Element aus dem Datenbestand als *Pivotelement* aus.
3. Teile den Datenbestand so in zwei Teilbestände auf, dass in dem einen nur die Elemente größer oder gleich dem Pivotelement sind und in dem anderen nur die, die kleiner sind als das Pivotelement.

   Bemerkung: *Größer oder kleiner bezieht sich immer auf das vom Algorithmus unabhängige Sortierkriterium.*

4. Sortiere beide Teilbestände. Hierfür wird Quicksort rekursiv benutzt.
5. Füge die beiden, nun sortierten, Teilbestände so zusammen, dass der Gesamtbestand sortiert ist.

Es ist wichtig, sich dieses Verfahren anhand einiger Beispiele Schritt für Schritt klarzumachen. Abbildung 21 zeigt die Sortierung eines kleinen Datenbestandes mit Quicksort.

2.4.5 Zusammenfassung: Sortierung

Zwei Dinge sind grundsätzlich zu trennen: Das Sortierkriterium gibt an, in welcher Hinsicht zwei Elemente des Datenbestandes verglichen werden und legt damit die Reihenfolge fest, in der die Elemente im sortierten Datenbestand anzutreffen sein werden. Dagegen beschreibt der Sortieralgorithmus die Vorgehensweise, nach der Sortierung durchgeführt wird. Dabei wird das Sortierkriterium abstrakt behandelt und ist unabhängig vom Algorithmus.

2.5 Laufzeit und Komplexität

2.5.1 Laufzeit und Speicherverbrauch

Bei der Betrachtung der Effizienz von Rechenverfahren geht es um den Aufwand, der für die Abarbeitung des Verfahrens nötig ist. Dabei geht es vorwiegend um die benötigte Zeit

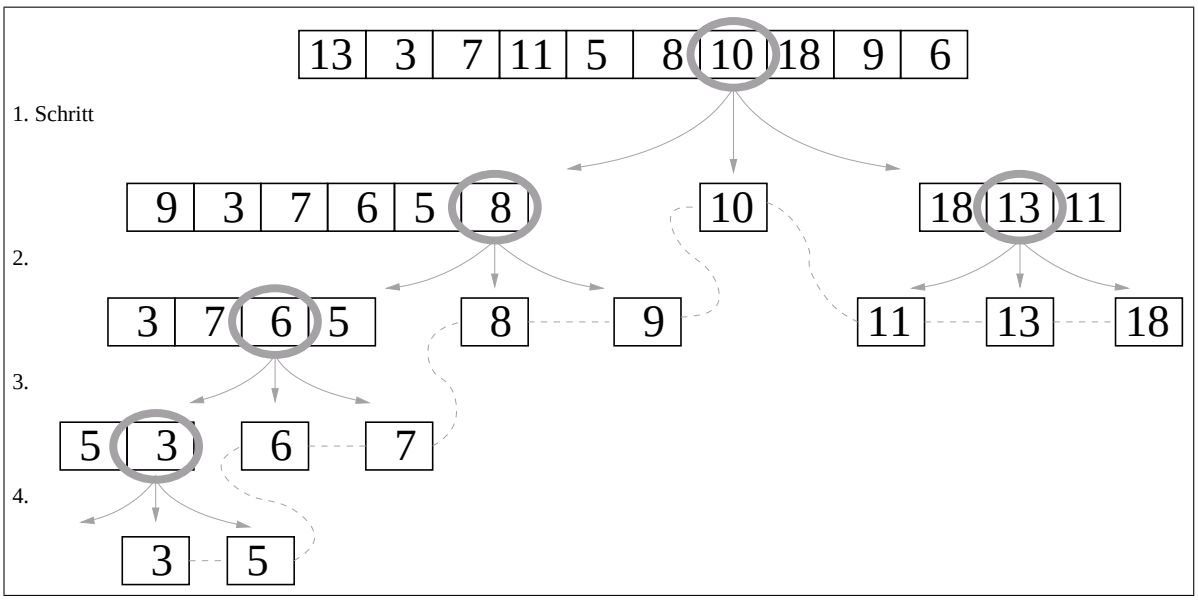

Abbildung 21: Sortierung mit Quicksort: Für jeden Aufteilungsschritt ist das Pivotelement grau eingekreist. Die Pfeile zeigen auf die beiden Teilbestände und das Pivotelement im nächsten Schritt. Dabei wird die rekursive Baumstruktur der Verarbeitung deutlich. Die Verbindung der sortierten Teilbestände ist durch die unterbrochene Linie gezeigt.

einerseits und den benötigten Speicherplatz andererseits. In unserem Fall ist die Rechenzeit der wichtigere Faktor insofern, als wir noch keine Rechenverfahren kennengelernt haben, die nennenswerte Mengen Speicher benötigen.
Bei der Laufzeit eines Algorithmus interessiert weniger die physische Abarbeitungszeit des Algorithmus. Diese ändert sich ohnehin, sobald ein anderer Computertyp, eine andere Programmiersprache verwendet wird oder sonstige Änderungen an den Rahmenbedingungen stattfinden. Es geht vielmehr darum, wie sich die Abarbeitungszeit verändert, wenn der Datenbestand nennenswert vergrößert wird. Um dies besser verständlich zu machen, folgen zunächst einige intuitive Beispiele.

2.5.2 Intuitive Beispiele für einige Komplexitätsklassen

Wir erinnern uns daran, was ein Array ist und daran, dass seine wichtigste Eigenschaft ist, dass auf jedes Element direkt über den Index zugegriffen werden kann. Wie der vorige Abschnitt bereits angedeutet hat, spielt es eine große Rolle, wie sich die Laufzeit zur Größe des Datenbestandes verhält.
Wir wollen uns ansehen, wie es um die Verarbeitungszeit beim Erreichen (zum Beispiel zum Auslesen) eines bestimmten Elementes in einem Array bestellt ist. Bei einem Array mit 100 Elementen dauert das Auslesen des Elementes bei jedem Index eine bestimmte Zeit. Wenn wir ein Array mit 200 Elementen benutzen, ändert sich an der Zeit überhaupt nichts. Die Zeit ist unabhängig von der Größe des Datenbestandes. Man sagt auch, das

Auslesen eines Elementes aus einem Array sei in *konstanter Zeit* möglich.
Schauen wir uns eine andere Situation an. Wir möchten in einem unsortierten Array herausfinden, ob ein bestimmtes Element enthalten ist. Wie in Abschnitt 2.3.3 besprochen wurde, muss das Array Element für Element systematisch durchsucht werden. Ist das Element enthalten, wird es im Durchschnitt gefunden, nachdem die Hälfte der Elemente besucht worden sind.[8] Wenn nun ein doppelt so großes Array durchsucht werden muss, dauert die Suche durchschnittlich die doppelte Zeit. Die Verarbeitungszeit ist abhängig von der Größe des Datenbestandes – sie ist proportional zur Datenbestand-Größe. Das ist etwas grundlegend anderes, als der vorher betrachtete Fall.
Noch ein anderer Fall liegt vor, wenn ein Element im sortierten Array gesucht wird. Den Algorithmus, den wir dazu verwendet haben, halbiert das Array in zwei Hälften, und durchsucht nur die Hälfte, in der das gesuchte Element zu finden sein muss. Für einen Halbierschritt wird immer die gleiche Zeit benötigt. Für ein Array einer bestimmten Größe wird durchschnittlich eine bestimmte Anzahl Halbierschritte benötigt. Wenn hier die Arraygröße verdoppelt wird, muss durchschnittlich nur ein weiterer Schritt vorgenommen werden. Das ist wesentlich weniger, als im Falle des unsortierten Arrays, bei dem eine Verdopplung des Datenbestandes auch eine Verdopelung des Aufwandes bedeutet. Es ist dennoch mehr, als im Falle des Zugriffs per Index, bei dem die Verarbeitungszeit absolut unabhängig von der Arraygröße war. Dieses Verhalten der Laufzeit gegenüber der Größe des Datenbestandes ist logarithmisch.

2.5.3 Komplexitätsklassen

Bei der Betrachtung des Zeitaufwandes von Berechnungen zeigt sich, dass eine Einteilung in sogenannte *Komplexitätsklassen* Sinn macht.
Am Beispiel der systematischen Suche in einem ungeordneten Array (vgl. Abschnitt 2.3.3) lässt sich gut verdeutlichen, worin dieser Qualitätsunterschied besteht und welche Konsequenz er hat. Angenommen der Vergleich eines Elements mit einem anderen auf Computer A dauert 50 Sekunden. Dann benötigt man für 10 Elemente 500 Sekunden, wenn sonstige Operationen nicht ins Gewicht fallen.
Bubblesort benötigt für die Sortierung eines Arrays von 10 Elementen 9 Vergleiche für den ersten Durchlauf. Für den nächsten 8 und so weiter. Wir benötigen 9 Durchläufe, um das ganze Array zu sortieren also insgesamt 45 Vergleiche. Mit dem Computer A von vorhin benötigen wir 2250 Sekunden. Wenn uns das zu lange dauert, könnten wir die Sortierung mit Bubblesort auf dem 50 mal schnelleren Computer B durchführen – er benötigt eine Sekunde für einen Vergleich. Nun dauert die ganze Sortierung nur noch 45 Sekunden, was sogar weniger ist als die Suche auf Computer A.
Nehmen wir nun ein Array mit 100 Elementen. Die systematische Suche benötigt nun 99 Vergleiche d.h. 4950 Sekunden auf Rechner A. Die Sortierung mit Bubblesort benötigt bei 100 Elementen 4950 Vergleiche und damit auf Rechner B 4950 Sekunden – genauso lang, wie die systematische Suche auf Rechner A. Rechner B ist 50 mal schneller als Rechner A und B hat bei 10 Elementen seine Aufgabe sogar wesentlich schneller erledigt

[8] Für das hier angestrebte intuitive Bewusstsein soll der Einfachheit halber davon ausgegangen werden, dass das Element immer enthalten ist.

als Rechner A seine Aufgabe. Dennoch ist bei nur 10 mal größerem Datenbestand der 50-fache Geschwindigkeitsvorteil bereits wieder aufgezehrt. Es geht um das Prinzip, dass egal welchen Vorteil ein Rechner durch seine Geschwindigkeit hat wird dieser Vorteil mit steigender Datenbestand-Größe nichts nützen. Egal wie viel schneller Rechner B gemacht wird, es wird immer ein Array geben, bei dem Rechner A die Aufgabe der Suche schneller erledigt.
Die Beziehung zwischen Größe des Datenbestandes und der Verarbeitungszeit kann in Form einer Funktion ausgedrückt werden, die jeder Datenbestandsgröße eine Laufzeit zuordnet. Bei der Betrachtung dieser Funktionen kommt es nicht auf exakte Werte sondern auf das grundlegende Verhalten der Funktion an. Die systematische Suche besitzt *lineare Zeitkomplexität*, während Bubblesort *quadratische Zeitkomplexität* besitzt.
Die lineare Funktion wird früher oder später immer von der quadratischen Funktion übertroffen werden. Zwar können schnellere Rechner den Effekt abschwächen, jedoch nicht aufheben. Die Rechengeschwindigkeit entspricht einem konstanten Faktor, der in die Laufzeit mit eingeht. Doch ein konstanter Faktor wird es niemals schaffen, eine quadratisches Wachstum so zu mildern, dass es auf Dauer schwächer bleibt, als das lineare Wachstum. Das grundsätzliche Verhalten von Parabeln und Geraden ist in Abbildung 22 bildlich dargestellt.
Diese Beziehung zwischen Funktionen ist der springende Punkt bei der Bildung von Komplexitätsklassen. Bezüglich ihres grundlegenden Verhaltens mit steigender Datenbestandsgröße sind die Funktionen einer Komplexitätsklasse äquivalent. Diese Äquivalenz äußert sich darin, dass alle Funktionen einer Klasse sich gleich zu allen Funktionen einer beliebigen anderen Klasse verhalten: Die Funktionen einer Komplexitätsklasse C_1 wachsen auf lange Sicht entweder *alle* schneller als *alle* Funktionen einer anderen Komplexitätsklasse C_2 oder alle langsamer. Dieses langfristige Wachstumsverhalten ist das Charakteristische einer Komplexitätsklasse.
Dabei ist die Komplexitätsklasse nicht auf eine bestimmte Ressource bezogen, sondern hängt nur von der Funktion ab. Es kann also sowohl die Laufzeit als auch der Speicherverbrauch eines Algorithmus bezüglich der Eingabegröße in Komplexitätsklassen eingeteilt werden.

2.5.4 Zusammenfassung: Laufzeit und Komplexität

Bei der Betrachtung der Effizienz eines Algorithmus geht es vor allem um zwei elementare Ressourcen: Arbeitsspeicher und Verarbeitungszeit. Für beide Ressourcen macht die Bildung von Komplexitätsklassen insofern Sinn, als sich Veränderungen an den Ablaufbedingungen, wie etwa die Modifikation des ausführenden Computers, nur marginal auf den Ressourcenbedarf eines Algorithmus auswirken.
Der Ressourcenverbrauch kann als Funktion der Größe der Eingabedaten gesehen werden. Das prinzipielle Verhalten der Funktion mit steigender Eingabegröße ist entscheidend bei der Bildung der Komplexitätsklasse. Dabei ist die Komplexitätsklasse nicht auf eine bestimmte Ressource bezogen, sondern hängt nur von der Funktion ab.

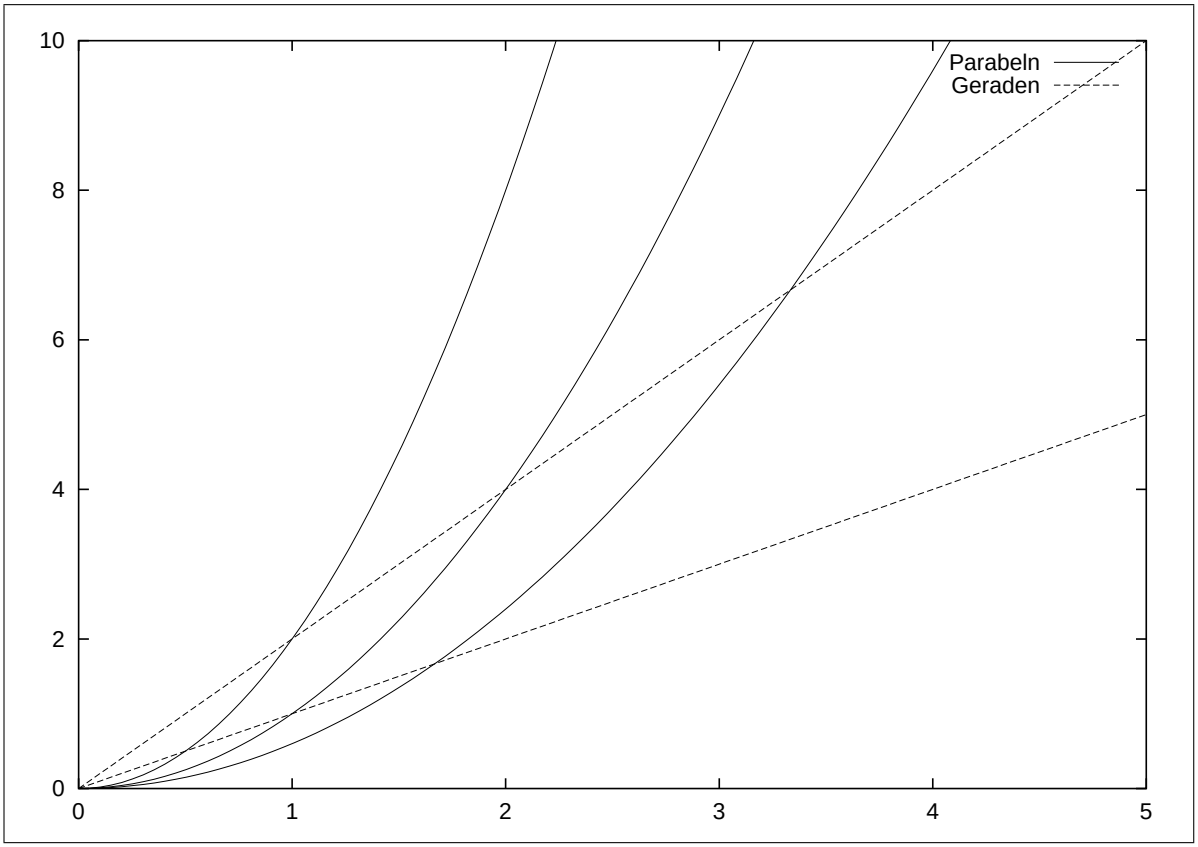

Abbildung 22: Hier ist das unterschiedliche Verhalten von Geraden und Parabeln zu sehen. Unabhängig davon, wie steil eine Gerade und wie flach eine Parabel ansteigt, ist es immer der Fall, dass die Parabel die Gerade irgendwann übertrifft.

►►► Woche 6 ◄◄◄

3 Funktion, Algorithmen, Rechnermodelle

3.1 Funktion und Algorithmus

Die elektronische Datenverarbeitung (EDV), die auch zu Beginn des vorigen Kapitels bereits angerissen wurde, spielt heute eine zentrale Rolle bei der Automatisierung von Erfassung, Verwaltung und Verarbeitung großer Datenmengen. Während sich das vorige Kapitel mit Repräsentation und Organisation von Daten befasste, wird nun ein genauerer Einblick in die eigentliche Verarbeitung von Daten gegeben. Bis jetzt kennen wir das „Arbeitsmaterial“ der EDV, nun werden wir uns genauer mit der Art und Weise beschäftigen, wie dieses Material verarbeitet wird.

3.1.1 Funktionen, Vorschriften und Maschinen

Wie errechnet man den Durchschnitt einer Folge von Zahlen? Und wie erklärt man jemandem, der es nicht weiß, wie man den Durchschnitt – das arithmetische Mittel – einer Folge von Zahlen errechnet?

> Addiere alle Zahlen der Folge und dividiere die Summe durch die Anzahl der Zahlen in der Folge.

Hier haben wir eine Folge von Anweisungen gegeben, wie aus den gegebenen Zahlen das Ergebnis errechnet werden kann. Wir befinden uns auf recht hohem Abstraktionsniveau; so wurden viele Anweisungen nicht exakt beschrieben. Wie addiert man zwei Zahlen? So könnte man weiter fragen, auch das kann prinzipiell als Folge von Anweisungen beschreiben werden.
Bereits im Kapitel über mathematische Grundlagen wurden verschiedene Sichtweisen von Funktionen verwendet. Nun ist neben der formalen Definition des Funktionswertes (durch eine Gleichung oder rekursive Definition etwa) auch die Angabe einer Berechnungsvorschrift hinzugekommen. Die Funktion ist durch eine Art Rezept beschrieben, wie zu einem bestimmten Element der Wertemenge der dazugehörige Funktionswert zu berechnen ist. Der Vorteil einer solchen Vorschrift besteht darin, dass sie, richtig formuliert, durch Maschinen ausgeführt werden kann, wie wir sehen werden.

3.1.2 Rechenmaschinen

Eine Vorstufe des modernen Computers wurde schon sehr früh in Form von mechanischen und elektrischen Rechenmaschinen geschaffen. Addiermaschinen, Rechenschieber etc. sind spezielle Maschinen, die für die Ausführung bestimmter Rechenaufgaben geeignet sind. Zwar benötigt man für verschiedene Berechnungen auch verschiedene Maschinen

doch zeichnen sich auch diese Vertreter der datenverarbeitenden Maschinen durch einige wichtige Merkmale aus.
Die verarbeiteten Daten werden in einer geeigneten Form repräsentiert – bei Computern heißt Rerpäsentation fast immer Speicherung. Der Funktionswert wird durch die Abarbeitung genau festgelegter Verarbeitungsschritte aus dem Argumentwert bestimmt. Ob mithilfe von Mechanik, Elektrotechnik oder Elektronik: Jedesmal wird der Argumentwert in irgendeiner geeigneten Form gespeichert; als Zahnradstellung, als Kugelanzahl etc. Jedesmal werden bestimmte Schritte unternommen, die schließlich dazu führen, dass das Ergebnis – der Funktionswert des Argumentwertes – in geeigneter Repräsentationform ausgegeben werden; als Radpositionen, Kugelanzahlen, Position eines Zeigers auf einer Skala etc.
An diesen Rechenmaschinen ist ein grundlegendes Prinzip bereits gut entwickelt:

1. Der Argumentwert wird in die Maschine eingegeben.

2. Durch genau festgelegte Verarbeitungsschritte wird aus der Eingabe der Funktionswert ermittelt.

3. Der Funktionswert wird ausgegeben.

Man nennt dieses Prinzip das *EVA-Prinzip*: Eingabe, Verarbeitung, Ausgabe. Wie diese einzelnen Vorgänge aussehen, hängt von der Rechenmaschine und den von ihr benutzten Repräsentationen ab.

3.1.3 Universelle Rechenmaschinen – Taschenrechner

Das Problem dieser Rechenmaschinen ist, dass sie stets dieselbe Berechnung durchführen. Entsprechend sind es meist Maschinen für sehr gängige Berechnungen, die immer wieder auftraten: Addition, Multiplikation usw. Bei komplexeren Berechnung ist der menschliche Benutzer genötigt, zwischen verschiedensten Maschinen hin- und herzuwechseln. Um etwa das arithmetische Mittel einiger Zahlen zu berechnen, müßte man erst alle Zahlen mit einem Addierer addieren. Das von ihm ausgegebene Ergebnis übertrüge man in den Dividierer, um es durch die Anzahl der Zahlen zu teilen. Die Ausgabe des Dividierers ist das gesuchte Ergebnis.
Die Überlegung drängt sich auf, auch diese Arbeit durch Maschinen erledigen zu lassen. An dieser Stelle verlässt man das Territorium häufig benötigter Elementaraufgaben: Es bedeutete viel zu viel Aufwand, für jede irgendwo benötigte komplexere Berechnungsaufgabe eine eigene Maschine herzustellen.
Die Lösung des Problems wäre die Konstruktion einer universellen Maschine, die jede Art von Berechnung ausführen kann. Man könnte etwa alle elementaren Grundoperationen in einer Maschine zusammenfassen und sie so kombinieren, dass nicht für jede Berechnung die Ausgabe einer Teilmaschine als Eingabe in eine andere übertragen werden muss. Man könnte die Ausgabe so repräsentieren, dass sie gleichzeitig als Eingabe für eine weitere Berechnung verwendet werden kann.

Ein mittelmoderner Taschenrechner erfüllt diese Anforderung. Der auf dem Display angezeigte Wert kann für eine weitere Berechnung verwendet werden, deren Ergebnis wiederum auf dem Display ausgegeben wird.

3.1.4 Programmierbare Maschinen und Automatische Verarbeitung

Auch wenn ein Taschenrechner schon große Vorteile gegenüber speziellen Rechenmaschinen bietet, so ist es nach wie vor Aufgabe des Benutzers, die verschiedenen Elementarrechnungen so zu kombinieren, dass er das gewünschte Ergebnis erhält. Insbesondere bei komplizierten Berechnungen ist das fehleranfällig, wie jeder weiß, der schon komplexere Berechnungen mit einem durchschnittlichen Taschenrechner durchgeführt hat.
Auch für dieses Problem bietet sich eine Lösung: Anstatt den Menschen all die Kombinationsarbeit zur Ausführung komplexer Berechnungen jedesmal neu ausführen zu lassen, kann eine Berechnungsabfolge gespeichert werden. Sie steht dann zu einem späteren Zeitpunkt wieder zur Verfügung.
Um eine Berechnung auszuführen, müssen lediglich am Anfang alle Eingaben gemacht, die Ausführung der gespeicherte Berechnungsabfolge angestoßen und am Schluss das Endergebnis ausgelesen werden.
Dies wird durch *programmierbare* Maschinen möglich. Anstatt den Menschen die Einzelberechnungen anstoßen zu lassen, aus denen die ganze Berechnung zusammengesetzt ist, gibt es hierfür eine Kontrolleinheit. Sie übernimmt genau das Einleiten der richtigen Aktion zur richtigen Zeit. Um diese Kontrolleinheit für alle Berechnungen gleichermaßen benutzen zu können, ist sie so konstruiert, daß sie beliebige Befehlsfolgen abarbeiten kann.
Dies entspricht der Funktionsweise einer Drehorgel bei der unterschiedlichen Papierstreifen für verschiedene Stücke eingelegt werden können. Ein Loch an einer bestimmten Stelle löst einen Ton aus, ein Loch an anderer Stelle einen anderen. So kann im Prinzip jedes Stück gespielt werden, ohne dass ein Mensch da sein muss, der zur richtigen Zeit die richtigen Töne auslöst.
Ersetzt man gedanklich Töne durch elementare Rechenoperationen, hat man eine gute Intuition, wie eine programmierbare Rechenmaschine funktioniert. Wir benötigen lediglich den richtigen Lochstreifen für jede Berechnung.
Neben Befehlen, die Berechnungsschritte auslösen, sind noch Hilfsbefehle zur Steuerung und zum Einlesen von Daten notwendig. Eine simple Befehlsfolge, wie sie bei einem Lochstreifen gegeben wäre, könnte zum Beispiel nicht ohne weiteres Befehle wiederholen oder bestimmte Sequenzen in Abhängigkeit eienr bestimmten Bedingung ausführen.
Wenn Befehlssequenzen schon von vornherein derartig festgelegt sind, ergeben sich daraus starke Einschränkungen. Zum Beispiel kann ein Programm immer nur die Anzahl Werte einlesen, auf die es bei der Programmierung ausgelegt wurde. Das wäre schon im Falle des arithmetischen Mittels sehr problematisch. Es wäre wünschenswert, dasselbe Programm für eine variable Anzahl Werte zu verwenden.
Hier hilft zum Beispiel ein Befehl, der einen anderen so oft ausführt, bis eine bestimmte Bedingung eintritt. (vgl. Abschnitt 1.7.6) Es gibt noch weitere Steuerbefehle, die wir uns allerdings erst später genau ansehen werden. Zunächst geht es nur darum ihre

Notwendigkeit zu verstehen.

3.1.5 Präzisierung der Vorschrift: Algorithmus

Durch die Programmierbarkeit werden Maschinen um Größenordungen flexibler. Prinzipiell kann nun ein und dieselbe Maschine beliebig komplexe Berechnungen durchführen, ohne dass ein Mensch die Steuerung übernehmen oder die Maschine modifiziert werden müßte.
Um eine Berechnungsvorschrift, wie sie in 3.1.1 gegeben ist, von einer Maschine ausführen zu lassen, muss sie präzise formuliert werden. Letztendlich muss jeder Arbeisschritt in einer maschinenlesbaren Form repräsentiert werden, so dass die Zielmaschine ihn als Befehl ausführen kann. Allerdings hängt es von der Maschine ab, welche Befehle sie genau unterstützt und wie die Befehle repräsentiert werden müssen.
Als Vorstufe der Befehlssequenz für eine bestimmte Maschine wählt man eine von Menschen lesbare aber doch sehr präzise Fassung der Vorschrift. Diese lässt sich relativ leicht in das Befehlsformat der jeweiligen Maschine übertragen, ist jedoch so abstrakt, dass das Verfahren im Fordergrund steht und nicht eine bestimmte Maschine.
Eine derart durch Angabe wohldefinierter Einzelschritte präzisierte Verarbeitungsvorschrift nennt man *Algorithmus*.

25 Definition Ein Algorithmus ist eine präzise, endliche Verarbeitungsvorschrift, die genau festlegt, wie die Probleme einer bestimmten Klasse von Problemen gelöst werden. Ein Algorithmus liefert eine Funktion, die jeder zulässigen Eingabe die Ausgabe zuordnet.

Dabei sind die folgenden Eigenschaften von Bedeutung:

Finitheit: Die Beschreibung des Verfahrens ist von endlicher Länge (statische Finitheit) und zu jedem Zeitpunkt der Abarbeitung des Algorithmus hat der Algorithmus nur endlich viele Ressorucen belegt (dynamische Finitheit). Diese Eigenschaft ist wichtig, weil die ausführende Maschine stets endlich ist.

Terminierung: Algorithmen, die nach Durchführung endlich vieler Schritte (Operationen) zum Stillstand kommen, heißen terminierend.

Effektivität: Die Wirkung einer einzelnen Anweisung eines Algorithmus ist eindeutig festgelegt.

3.1.6 Zusammenfassung: Funktion und Algorithmus

Die elektronische Datenverarbeitung hat die Automatisierung von Berechnungsvorgängen zum Ziel. Rechenmaschinen für elementare Operationen wurden zu universellen Maschinen kombiniert, die prinzipiell jede Berechnung durchführen können. Durch Programmierbarkeit wurde auch die Kontrolle der Einzeloperationen automatisiert. Um Berechnungsvorschriften von solchen Maschinen ausführen zu lassen, müssen sie in ein für die

betreffende Maschine lesbares Programm übersetzt werden. Als Vorstufe des maschinenspezifischen Programmes dient der abstrakter und möglichst maschinenunabhängig formulierte Algorithmus, eine endliche, präzise Verarbeitungsvorschrift. Besonders interessant sind bei Algorithmen die Eigenschaften Finitheit, Terminierung und Effektivität.

3.2 Maschinenmodelle

3.2.1 Formale Grundlagen

Bevor wir uns formale Maschinenmodelle nähern können, müssen einige Grundlagen im Bereich der formalen Sprachen gelegt werden. In diesem Abschnitt ist es weniger von Bedeutung, sich eingehend mit den eingeführten Begriffen auseinanderzusetzen. Sie sind im wesentlichen intuitiv gut zugänglich und werden hier nur eingeführt, weil dies die spätere Definitionsarbeit wesentlich leichter macht.
Eine endliche, nicht-leere Menge, bezeichnet man im Zusammenhang formaler Sprachen oft als *Alphabet*. Die Elemente einer solchen Menge heißen dann *Zeichen* oder *Symbole*.

26 Definition Ein *Wort* über einem gegebenen Alphabet Σ ist jede endliche Folge von Elementen aus Σ. Die Menge aller Wörter über dem Alphabet Σ wird mit Σ^* bezeichnet. Dies schließt das *leere Wort*, mit ε bezeichnet, mit ein.

Als Beispiel schauen wir uns das Alphabet $\Sigma_{ab} = \{a, b\}$ an. Die Menge aller Worte über Σ ist dann

$$\Sigma_{ab}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}.$$

27 Definition Eine *Sprache* über Σ ist jede Teilmenge L von Σ^*.

3.2.2 Endliche Automaten

Die Idee eines endlichen Automaten ist einfach. Der Automat hat endlich viele Zustände. In Abhängigkeit der Eingabe kann der Automat von einem Zustand in einen anderen übergehen. Die Parallele zu einem Getränke-Automaten soll diese Idee verdeutlichen.
Man stellt sich vor, der Automat lese Worte, bestehend aus Zeichen seines *Eingabealphabets*. Nach dem Lesen des Wortes befindet er sich in einem bestimmten Zustand. In der theoretischen Informatik werden endliche Automate für (sprach-)theoretische Überlegungen benutzt – hier geht es lediglich darum, ein Gefühl für formale Maschinenmodelle zu bekommen und sich an die mechanistische Arbeitsweise solcher Maschinen zu gewöhnen. Nun soll die formale Definition gegeben werden. Sie wirkt zunächst sehr verwirrend, dürfte aber mit etwas Übung klar werden.

28 Definition Ein *(deterministischer) endlicher Automat* M ist gegeben durch ein 5-Tupel

$$M = (Z, \Sigma, \delta, z_0, E).$$

Hierbei bezeichnet Z die Menge der Zustände und Σ das *Eingabealphabet*. Es muss gelten $Z \cap \Sigma = \emptyset$, d.h. Z und Σ haben keine gemeinsamen Zeichen. Z und Σ sind Alphabete und

müssen daher endliche, nicht-leere Mengen sein. $z_0 \in Z$ ist der *Startzustand*, $E \subseteq Z$ ist die Menge der *Endzustände*[9]. δ ist die *Überführungsfunktion*, die jedem 2-Tupel (z, α), wobei $z \in Z$ und $\alpha \in \Sigma$, einen Zustand $z' \in Z$ zuordnet.[10]

Durch die Überführungsfunktion ist festgelegt, aus welchem Zustand der Automat in welchen Zustand übergeht, wenn eine bestimmte Eingabe erfolgt.

3.2.3 Beispiel eines endlichen Automaten

Als ein Analogiebeispiel wollen wir uns einen Getränkeautomaten als endlichen automaten vorstellen. Seine Arbeitsweise ist einfach: Zunächst erwartet er Geld, dann die Wahl eines Getränks, dann wird das gewählte Getränk ausgegeben und er erwartet erneut Geld...
Es gibt drei Zustände, die direkt mit den oben genannten Arbeitsschritten identifiziert werden können. Im ersten Zustand erwartet er Geld, im zweiten eine Getränkewahl und im dritten gibt er das Getränk aus und kehrt in den ersten Zustand zurück. Gehen wir davon aus, dass es zwei Getränkesorten gibt. Wir unterteilen den dritten Zustand deshalb in zwei Zustände, für jedes Getränk einen. Nun ist also

$$Z = \{z_0, z_1, z_{2a}, z_{2b}\}.$$

Die Interaktionen mit dem Benutzer sollen die Eingaben sein. Daher ist das Eingabealphabet

$$\Sigma = \{\text{Geld}, \text{Wahl}_a, \text{Wahl}_b, \text{Getränk}_a, \text{Getränk}_b\},$$

wobei Geld der Einwurf von Geld, Wahl_a und Wahl_b die Getränkeauswahl und schließlich Getränk_a und Getränk_b die Ausgabe des Getränkes ist.

Bemerkung: *Die Ausgabe des Getränkes wird zwar vom Automaten angestoßen, ist aber in diesem Modell eine Eingabe. Daran wird deutlich, dass der Vergleich hinkt, jedoch hoffentlich dennoch zum besseren Verständnis beiträgt.*

Die Überführungsfunktion bestimmt nun die Aktionen, die bei den Eingaben geschehen sollen. Bestimmte Eingaben sind in bestimmten Zuständen nicht möglich, wie etwa die Ausgabe des Getränks in Zustand z_0. Die Überführungsfunktion ist dann für die

[9] Endzustände spielen in der theoretischen Informatik eine große Rolle, sind für uns aber nicht von großer Bedeutung.

[10] Es gibt auch nichtdeterministische endliche Automaten, die in der theoretischen Informatik eine Bedeutung haben, jedoch für uns keinen besonderen didaktischen Wert haben. Daher sollen sie hier nicht beschrieben werden.

betreffenden Eingaben nicht definiert.

$$\begin{aligned}
\delta(z_0, \text{Geld}) &= z_1 \\
\delta(z_0, \text{Wahl}_a) &= z_0 \\
\delta(z_0, \text{Wahl}_b) &= z_0 \\
\delta(z_1, \text{Geld}) &= z_1 \\
\delta(z_1, \text{Wahl}_a) &= z_{2a} \\
\delta(z_1, \text{Wahl}_b) &= z_{2b} \\
\delta(z_{2a}, \text{Getränk}_a) &= z_0 \\
\delta(z_{2b}, \text{Getränk}_b) &= z_0
\end{aligned}$$

Der Startzustand ist z_0 und die Menge der Endzustände ist leer. Derselbe Automat ist in Abbildung 23 bildlich dargestellt.

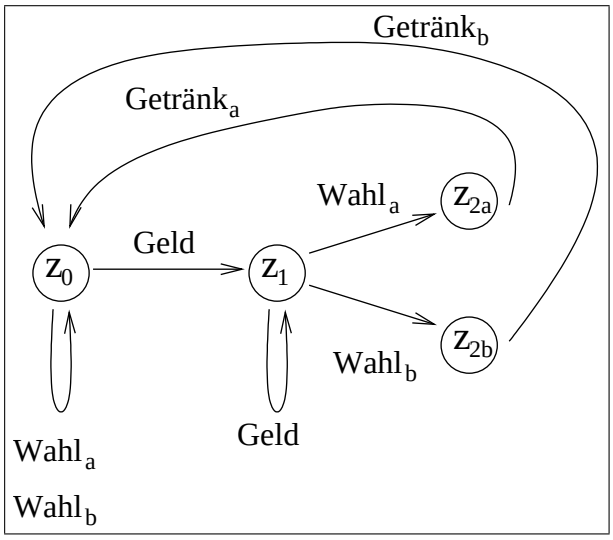

Abbildung 23: Bildliche Darstellung eines (deterministischen) endlichen Automaten. Die Zustände sind durch die bezeichneten Kreise dargestellt; die Überführungsfunktion über Pfeile, wobei die Eingaben an den Pfeilen vermerkt sind.

3.2.4 Turingmaschine – eine allgemeine Rechenmaschine

Ein anderes Maschinenmodell wurde von Alan M. Turing in den 1930-er Jahren vorgestellt. Seine *Turingmaschine* ist in der Lage, jede Berechnung durchzuführen, die auch moderne Computer durchführen können. Man ist mittlerweile überzeugt, dass dies jede intuitiv berechenbare Funktion einschließt, auch wenn der Begriff der intuitiven Berechenbarkeit nicht formal erfasst werden kann.[11]

[11] Die sogenannte *Churchsche These* behauptet, die Klasse der intuitiv berechenbaren Funktionen sei mit der Klasse der durch Turingmaschienen (und moderne Computer) berechenbaren Funktionen

Eine Turingmaschine besteht anschaulich aus einem potentiell unendlichen Band, das in Felder eingeteilt ist. Jedes Feld kann ein Zeichen des *Arbeitsalphabets* speichern. Ein Schreib-Lesekopf befindet sich an einem dieser Felder und kann dieses lesen oder verändern. Alle anderen Felder bleiben unverändert. Der Schreiblesekopf kann in einem Rechenschritt um maximal eine Position nach links oder rechts verschoben werden. Auf diese Weise ist im nächsten Schritt ein anderes Feld das schreib- bzw. lesbare Feld. Neben dem Band besitzt eine Turingmaschine eine endliche Kontrolleinheit. Sie ist für die Steuerung der Maschine zuständig.

3.2.5 Die Idee dahinter

Die ursprüngliche Idee der Turingmaschine ist die, dass sie alle elementaren Operationen unterstützt, die auch ein Mensch beim Rechnen auf Papier durchführen kann. So hatte ein Entwurf ein zweidimensionales Papier auf dem immer nur ein begrenzter Bereich auf einmal gelesen oder beschrieben werden konnte – wie Blickfeld und Hand des Menschen. Es zeigt sich aber, dass eine Maschine mit eindimensionalem Band nicht weniger mächtig ist als eine mit zweidimensionalem „Band“ – alle Berechnungen, die mit zweidimensionalem „Band“ möglich sind, sind auch mit eindimensionalem möglich.
Der Begriff *mächtig* bezieht sich im Falle von Maschinenmodellen auf die Klasse der Berechnungen, die eine Maschine ausführen kann. Wenn zwei verschiedene Maschinen trotz ihrer Unterschiede prinzipiell dieselben Berechnungen durchführen können, sagt man, sie seien gleich mächtig.
In der theoretischen Informatik wird die Menge der Algorithmen mit der Menge der Turingmaschinen identifiziert. Auf ähnliche Weise kann die Architektur einer Turingmaschine mit wichtigen Eigenschaften von Algorithmen in Beziehung gesetzt werden: In der Architektur einer Turingmaschine entspricht das Band dem Speicher, der die prinzipiell beliebig großen Datenmengen aufnimmt, die verarbeitet werden sollen. Die *endliche* Kontrolleinheit entspricht der *endlichen* Vorschrift, wie die Daten zu verarbeiten sind. Diese Elemente lassen sich auch bei einem modernen Computer identifizieren.
Wenn wir uns an das Beispiel des arithmetischen Mittels aus Abschnitt 3.1.1 erinnern, können wir ebenfalls diese Elemente finden. Die Vorschrift, wie man das arithmetische Mittel einer Folge von Zahlen berechnet ist endlich. Dennoch kann die Zahlenfolge theoretisch beliebig lang sein (auch wenn sie irgendwann zu Ende sein muss). Die Zahlen entsprechen den Daten, die im Falle der Turingmaschine auf dem Band gespeichert würden und die Vorschrift würde durch die Kontrolleinheit realisiert.

3.2.6 Turingmaschine – formale Definition

Auch für eine Turingmaschine gibt es eine formale Definition, mit der sich dieser Abschnitt beschäftigt.

---

identisch. Man ist mittlerweile davon überzeugt, dass dies zutrifft. Der interessierte Leser sei an dieser Stelle auf [2] verwiesen.

29 Definition Eine *Turingmaschine* M ist gegeben durch ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E).$$

Hierbei sind:

- Z die endliche *Zustandsmenge*;
- Σ das Eingabealphabet;
- $\Gamma \supset \Sigma$ das Arbeitsalphabet ;
- δ die Überführungsfunktion, die jedem Tupel (z, α), wobei $z \in Z$ und $\alpha \in \Delta$, ein Tupel (z', β, k) zuordnet, wobei $z' \in Z$, $\beta \in \Gamma$ und $k \in \{L, R, N\}$;
- z_0 der Startzustand;
- $\square$ das Blank;
- $E \subseteq Z$ die Menge der Endzustände.

Eine solche Definition schleppt einen riesigen formalen Apparat mit sich herum, der vor allem von der Tatsache herrührt, dass Turingmaschinen zur Präzisierung sonst intuitiver Begriffe verwendet werden und daher keine formalen Lücken gelassen werden sollen. Wir benötigen nicht alle Finessen dieser Definition, wie es bereits bei endlichen Automaten der Fall war.
Nach eingehender Betrachtung wird die Definition jedoch überraschend verständlich werden. Hierzu sollen die im vorigen Abschnitt beschriebenen Elemente der „anschaulichen“ Turingmaschine mit der formalen Definition in Beziehung gesetzt werden.
Die Zustandsmenge, der Startzustand und die Endzustände beziehen sich auf die endliche Kontrolleinheit und sind mit ihren Pendants eines endlichen Automaten vergleichbar. Die Überführungsfunktion drückt, ähnlich wie bei einem endlichen Automaten, aus, wie sich die Turingmaschine in bestimmten Zuständen und bestimmten Zeichen auf dem Band verhält. Das Zeichen unter dem Schreib-Lesekopf entspricht der Eingabe. Die Aktion der Turingmaschine betrifft nicht nur ihren eigenen Zustand, sondern auch etwaige Schreiboperationen an der aktuellen Position Kopfes auf dem Band sowie die Bewegung des Kopfes.
Das Tupel (z', β, b), das in der Definition der Funktion δ verwendet wurde, gibt an, dass sich die Maschine nach der Abarbeitung des Schritts in Zustand z' befinden wird, das Zeichen β auf das Band schreibt und sich der Kopf anschließend um ein Feld nach links L, rechts R oder nicht N bewegen wird.
So bedeutet

$$\delta(z, a) = (z', b, k)$$

dass die Maschine, wenn sie in Zustand z ist und das Zeichen a unter dem Schreib-Lesekopf steht im nächsten Schritt in Zustand z' übergeht, an der alten Kopfposition das Zeichen b schreibt und nun den Kopf entsprechend k bewegt.
Zu Beginn befindet sich der Schreib-Lesekopf auf dem ersten Zeichen (ganz links) der Eingabe, die auf dem Band steht. Die noch nicht besuchten Felder links und rechts der Eingabe sind jeweils mit Blank $\square$ beschrieben.

3.2.7 Eine Beispiel Turingmaschine

Die Turingmaschine, die hier als Beispiel dienen soll, soll eine dezimal geschriebene, nicht negative Ganzzahl um 1 erhöhen. Das Eingabealphabet Σ besteht also aus den Ziffern 0 bis 9, im Arbeitsalphabet kommt noch das Blank hinzu.

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{30}$$

$$\Gamma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \square\} \tag{31}$$

Bei der Erfüllung ihrer Aufgabe soll die Maschine im Prinzip wie beim schriftlichen Addieren von rechts nach links, Stelle um Stelle vorgehen.

1. Zu Beginn befindet sich das erste Zeichen, also die Stelle am weitesten Links unter dem Kopf. Zunächst muss der Kopf sich also an die Einerstelle ganz nach rechts bewegen. Dazu bewegt er sich solange nach rechts, bis er das Blank am rechten Rand erreicht und dann um eins nach links.

2. Nun muss die Stelle unter dem Kopf um eins erhöht werden.

   a) Wenn es sich um eine der Ziffern 0-8, erhöhe sie um eins und fahre bei 3. fort.

   b) Wenn es sich um eine 9 handelt, kommt es zum Übertrag und die nächste Stelle muss ebenfalls erhöht werden. Setze also die momentane Ziffer auf 0, bewege den Kopf um eins nach links und fahre bei 2. fort.

   c) Wenn es sich um Blank handelt, schreibe eine 1 an diese Position und halte an. Dieser Fall tritt nur ein, wenn in der „linkesten“ Stelle ein Übertrag stattfindet, also die Zahl um eine Stelle länger wird.

3. Der Kopf soll am Ende wieder über dem ersten Zeichen der Eingabe sein und muss deshalb solange nach links bewegt werden, bis das Blank am linken Rand erreicht ist. Dann muss er noch um eine Position nach rechts bewegt werden.

Soweit haben wir das Turingmaschinenprogramm natürlichsprachlich ausformuliert, dass wir nun eine exakte Turingmaschine angeben können. Jeder der Hauptschritte entspricht einem Zustand, so dass wir drei Zustände während der Ausführung benötigen und einen Endzustand z_e. Das ist also unsere Zustandsmenge

$$Z = \{z_0, z_1, z_2, z_e\}.$$

Zuletzt müssen wir noch die Überführungsfunktion anschauen. Wenn die Maschine sich in z_0 befindet, soll sie den Kopf auf die erste Stelle der Zahl bewegen. Das entspricht

Schritt 1. Für Zustand z_0 sieht die Überführungsfunktion δ so aus:

$$\begin{aligned}
\delta(z_0, 0) &= (z_0, 0, R) \\
\delta(z_0, 1) &= (z_0, 1, R) \\
\delta(z_0, 2) &= (z_0, 2, R) \\
\delta(z_0, 3) &= (z_0, 3, R) \\
\delta(z_0, 4) &= (z_0, 4, R) \\
\delta(z_0, 5) &= (z_0, 5, R) \\
\delta(z_0, 6) &= (z_0, 6, R) \\
\delta(z_0, 7) &= (z_0, 7, R) \\
\delta(z_0, 8) &= (z_0, 8, R) \\
\delta(z_0, 9) &= (z_0, 9, R) \\
\delta(z_0, \Box) &= (z_1, \Box, L)
\end{aligned}$$

Das Band bleibt in jedem Falle unverändert, was daran zu erkennen ist, dass stets dasselbe Zeichen geschrieben wird, das auch vor der Aktion unter dem Kopf war. Wenn das Blank erreicht ist, bewegt sich der Kopf nach links (über die Einerstelle der Zahl) und geht in z_1 über, womit wir bei Schritt 2 des natürlichsprachlichen Programms angekommen sind.

$$\begin{aligned}
\delta(z_1, 0) &= (z_2, 1, L) \\
\delta(z_1, 1) &= (z_2, 2, L) \\
\delta(z_1, 2) &= (z_2, 3, L) \\
\delta(z_1, 3) &= (z_2, 4, L) \\
\delta(z_1, 4) &= (z_2, 5, L) \\
\delta(z_1, 5) &= (z_2, 6, L) \\
\delta(z_1, 6) &= (z_2, 7, L) \\
\delta(z_1, 7) &= (z_2, 8, L) \\
\delta(z_1, 8) &= (z_2, 9, L) \\
\delta(z_1, 9) &= (z_1, 0, L) \\
\delta(z_1, \Box) &= (z_e, 1, N)
\end{aligned}$$

Falls kein Übertrag stattfand geht sie in Zustand z_2 über, also Schritt 3 des natürlichsprachlichen Programms. Nur im Falle eines Übertrags bleibt die Maschine in Zustand z_1. Wenn eine 9 erhöht wurde wird die nächste Stelle erhöht oder eine 1 auf das leere Band geschrieben. Wenn eine 1 auf das leere Band geschrieben wurde, d.h. die erste Stelle eine 9 war und erhöht wurde, sind wir fertig und halten die Maschine an.

In Zustand z_2 wird der Kopf auf das erste Zeichen zurückbewegt.

$$\begin{aligned}
\delta(z_2, 0) &= (z_2, 0, L) \\
\delta(z_2, 1) &= (z_2, 1, L) \\
\delta(z_2, 2) &= (z_2, 2, L) \\
\delta(z_2, 3) &= (z_2, 3, L) \\
\delta(z_2, 4) &= (z_2, 4, L) \\
\delta(z_2, 5) &= (z_2, 5, L) \\
\delta(z_2, 6) &= (z_2, 6, L) \\
\delta(z_2, 7) &= (z_2, 7, L) \\
\delta(z_2, 8) &= (z_2, 8, L) \\
\delta(z_2, 9) &= (z_1, 9, L) \\
\delta(z_2, \square) &= (z_e, \square, R)
\end{aligned}$$

Der Startzustand ist z_0 und der die Menge der Endzustände $E = \{z_e\}$.
Am besten läßt sich die Funktionsweise der Maschine nachvollziehen, indem man einige Beispiele von Hand auf dem Papier durchspielt.

3.2.8 Zusammenfassung: Maschinenmodelle

Maschinenmodelle werden eigentlich in der theoretischen Informatik verwendet und dienen dort zur Formalisierung von Aussagen über Berechenbarkeit, Sprachklassen und Algorithmen. In unserem Fall soll daran lediglich ein Gefühl dafür entwickelt werden, auf welche Weise Computer arbeiten und wie eine solch abstrakte, formale Maschine mit einer konkreten Vorstellung verbunden werden kann.
Besonderes Augenmerk liegt dabei auf der Turingmaschine, vor allem weil sie im Prinzip die gleiche Klasse von Problemen lösen kann, wie heutige Computer. Dabei formalisiert die Kontrolleinheit den endlichen Algorithmus und das Band den Speicher, der während der Ausführung benutzt wird.

Literatur

[1] Prof. R. Schneider: *Script Analysis I*, 1999/2000,
http://home.mathematik.uni-freiburg.de/rschnei/

[2] Uwe Schöning: *Theoretische Informatik – kurzgefaßt*, Spektrum Akademischer Verlag, 1997

[3] Prof. Ottmann: *Vorlesungsskript Info 1*, 1998/99,
http://ad.informatik.uni-freiburg.de
/lehre/ws9899/informatik-1/skript/java/kapitel2.pdf