Kapitel 1: Mathematische Grundlagen

1.1 Mengen, Multimengen, Tupel, Funktionen, Halbordnungen

Eine ***Menge*** ist eine Zusammenfassung von (endlich oder unendlich vielen) verschiedenen Dingen unserer Umwelt oder Vorstellungswelt, welche ***Elemente*** dieser Menge genannt werden. Wir schreiben $x \in M$, um auszusagen, dass das Ding x Element der Menge M ist. Andere Sprechweisen: x ist in M enthalten, oder M ***enthält*** x .
Um auszudrücken, dass x nicht in M enthalten ist, schreiben wir $x \notin M$. Die Schreibweise $x_1,...,x_n \in M$ steht für $x_1 \in M$ und … und $x_n \in M$, und $\{x_1,...,x_n\}$ ist die Menge, die genau die Elemente $x_1,...,x_n$ enthält.
Beispiele für Mengen sind:

- $\mathbb{N}$: Die Menge der *natürlichen Zahlen* 1,2,3,…
- $\mathbb{N}_0$: Die Menge der *Kardinalzahlen* (natürlichen Zahlen einschließlich der Null): 0,1,2,3,…
- $\mathbb{Z}$: Menge der *ganzen* Zahlen …, -3, -2, -1, 0, 1, 2, 3, …
- $\mathbb{Q}$, $\mathbb{R}$: Menge der *rationalen* bzw. *reellen* Zahlen
- $\varnothing$ oder { }: *leere Menge*
- $\mathbb{B}$ oder `boolean = {true, false}` oder {1,0} oder {tt, ff} oder {w, f} oder {L,O}: Menge der Wahrheitswerte
- {Adam, Eva}: Menge der ersten Menschen
- {A, B, C, …, Z}: Menge der Großbuchstaben im lateinischen Alphabet

In den Programmiersprachen Groovy und Java gibt es die folgenden vordefinierten Mengen:

- `Integer` oder `int`: {-2147483648 … 2147483647}
- `Short`: {-32768, …, 32767}
- `Byte`: {-128, …, -127} Menge der ganzen Zahlen zwischen -128 und 127
- `Long`: {-9223372036854775808, …,-9223372036854775808} (Postfix „L“)
- `BigInteger`: „große“ natürliche Zahlen (Postfix „g“, z.B. 45g)
- Dezimalzahlen, Gleitkommazahlen,
- Characters und Strings, Boolean

Die Menge M_1 ist eine *Teilmenge* von M_2 ($M_1 \subseteq M_2$), wenn jedes Element von M_1 auch Element von M_2 ist. Die Mengen M_1 und M_2 sind *gleich* ($M_1 = M_2$), wenn sie die gleichen Elemente enthalten (*Extensionalitätsaxiom*). Für jede (endliche) Menge M bezeichnet $|M|$ die *Kardinalität*, d.h. die Anzahl ihrer Elemente. Neben der Aufzählung ihrer Elemente können Mengen durch eine charakterisierende Eigenschaft gebildet werden (*Komprehensionsaxiom*). Beispiel: `byte` $= \{x \in \mathbb{Z} \mid -128 \leq x \text{ und } x \leq 127\}$. Auf diese Weise können auch unendliche Mengen gebildet werden.
Die unbeschränkte Verwendung der Mengenkomprehension kann zu Schwierigkeiten führen (Russells Paradox: „die Menge aller Mengen, die sich nicht selbst enthalten“), daher erlaubt man in der axiomatischen Mengenlehre nur gewisse Eigenschaften.
Auf Mengen sind folgende Operationen definiert:

- Durchschnitt: $M_1 \cap M_2 = \{x \mid x \in M_1 \wedge x \in M_2\}$.
- Vereinigung: $M_1 \cup M_2 = \{x \mid x \in M_1 \vee x \in M_2\}$.

- Differenz: $M_1 - M_2 = \{x \mid x \in M_1 \wedge x \notin M_2\}$.

Beispiele: $\mathbb{N} \cup \mathbb{N}_0 = \mathbb{N}_0$, $\mathbb{N} \cap \mathbb{N}_0 = \mathbb{N}$, $\mathbb{N} - \mathbb{N}_0 = \varnothing$, $\mathbb{N}_0 - \mathbb{N} = 0$. Durchschnitt und Vereinigung sind kommutativ und assoziativ. Daher kann man diese Operationen auf beliebige Mengen von Mengen ausweiten:

- $\bigcap \{M_1, M_2, ..., M_n\} = \{x \mid x \in M_1 \wedge x \in M_2 \wedge ... \wedge x \in M_n\}$.
- $\bigcup_{i \in I} M_i = \{x \mid x \in M_i \ \ für\ ein\ i \in I\}$.

Von Mengen kommt man zu „Mengen höherer Ordnung“ durch die *Potenzmengenbildung:* Wenn M eine Menge ist, so bezeichnet $\wp(M)$ oder 2^M die Menge aller Teilmengen von M. Cantor bewies, dass die Potenzmenge einer Menge immer mehr Elemente enthält als die Menge selbst. Speziell gilt für jede endliche Menge M: Wenn $|M| = n$, so ist $|\wp(M)| = 2^n$.
Beispiel: $\wp(\varnothing) = \{\varnothing\}$, $\wp(\{\varnothing\}) = \{\varnothing, \{\varnothing\}\}$ und
$\wp(\{A, B, C\}) = \{\varnothing, \{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$.

Multimengen

Während Mengen die grundlegenden Daten der Mathematik sind, hat man es in der Informatik oft mit Multimengen zu tun, bei denen Elemente „mehrfach“ vorkommen können. Beispiele sind

- eine Tüte mit roten, gelben und grünen Gummibärchen,
- die Vornamen der Studierenden dieser Vorlesung,
- die Multimenge der Buchstaben eines bestimmten Wortes, usw.

Multimengen können notiert werden, indem man zu jedem Element die entsprechende Vielfachheit angibt, z.B. ist {A:3, B:1, N:2} die Multimenge der Buchstaben im Wort BANANA. Formal können Multimengen definiert werden als Funktionen von einer Grundmenge in die Menge $\mathbb{N}_0$ der Kardinalzahlen, siehe unten.

Folgen

Aus Mengen lassen sich durch *Konkatenation* Tupel und Folgen bilden. Der einfachste Fall ist dabei die *Paarbildung* mit dem kartesischen Produkt. Wenn M_1 und M_2 Mengen sind, so bezeichnet $M_1 \times M_2 = \{(x, y) \mid x \in M_1 \wedge y \in M_2\}$ die Menge aller *Paare* von Elementen, deren erster Bestandteil ein Element aus M_1 und deren zweiter eines aus M_2 ist. Da die runden Klammern für vielerlei Zwecke verwendet werden, verwendet man manchmal zur Kennzeichnung von Paaren auch spitze oder, besonders in Programmiersprachen, eckige Klammern.
Beispiele: $\mathbb{N} \times \mathbb{B} = \{(1,tt),(1,ff),(2,tt),(2,ff),(3,tt),\ldots\}$, $\mathbb{N} \times \varnothing = \varnothing$,
$\mathbb{N} \times \{0\} = \{(1,0),(2,0),(3,0),\ldots\}$, $\mathbb{N} \times \mathbb{N}_0 = \{(1,0),(1,1),(1,2),\ldots,(2,0),(2,1),\ldots\}$
Eine Verallgemeinerung ist das n-stellige kartesische Produkt, mit dem *n-Tupel* gebildet werden:

$$M_1 \times M_2 \times \cdots \times M_n = \{(x_1, x_2, ..., x_n) \mid x_1 \in M_1 \wedge x_2 \in M_2 \wedge ... \wedge x_n \in M_n\}$$

2-Tupel sind also Paare, statt 3-, 4- oder 5-Tupel sagt man auch Tripel, Quadrupel, Quintupel usw. Die zur Produktbildung umgekehrten Operationen, mit denen man aus einem Produkt die einzelnen Bestandteile wieder erhält, bezeichnet man als Projektionen:

$$\Pi_i(x_1, x_2, ..., x_n) = x_i$$

Falls alle M_i gleich sind $(M_1 = M_2 = \cdots = M_n = M)$, so schreiben wir statt $M \times M \times \cdots \times M$ auch M^n und nennen $(x_1, x_2, ..., x_n)$ eine *Folge* oder *Sequenz* der Länge n

über M. In Programmiersprachen heißen Folgen auch *Arrays*, *Felder* oder *Reihungen*. Wichtige Spezialfälle sind $n=1$ und $n=0$. Im ersten Fall ist die einelementige Folge (*x*) etwas anderes als das Element *x*. Im zweiten Fall ist die leere Folge () unabhängig von der verwendeten Grundmenge. Achtung: Die leere Folge ist nicht zu verwechseln mit der leeren Menge!

M^n enthält nur Sequenzen einer bestimmten fest vorgegebenen Länge *n*. Unter M^* verstehen wir die Menge, die alle beliebig langen Folgen über M enthält:

$$M^* = \bigcup \{M^i \mid i \in N_0\} \ .$$

Wenn wir nur nichtleere Folgen betrachten wollen, schreiben wir M^+ :

$$M^+ = M^* - M^0 \ .$$

Eine *Liste* in der Programmiersprache Groovy ist ein Element von M^*, wobei M die Menge aller Objekte (Zahlen, Buchstaben, Listen, …) ist.
Beispiele: [1, 3, 5, 7] oder [17.5, "GP", 1.234f, 'a', 1e99]

Hier sind einige Groovy-Tatsachen zu Listen (http://groovy.codehaus.org/JN1015-Collections). In Java können diese Listen mit dem `public interface List` nachgebildet werden (http://java.sun.com/j2se/1.4.2/docs/api/java/util/List.html).

```
assert [0, "a", 3.14].class == java.util.ArrayList
assert [0, 4, 7] + [11] == [0, 4, 7, 11]
assert [0, 4, 7] - [4] == [0, 7]
assert [0, "a", 3.14] * 2 == [0, "a", 3.14, 0, "a", 3.14]
assert [0, 4, 7] != [0, 7, 4]
assert [] != [0]
assert [] != [[]]
assert [[],[]] != [[]]
assert [[],[]].size == 2
assert [0,4,7][2] == 7
x=[0,4,7]; assert x[2] == 7
x=[0,4,7]; x[2]=3; assert x == [0,4,3]
x=[0,4,7]; x[2]=3; x[5]=2; assert x == [0, 4, 3, null, null, 2]
x=[0,4,7]; assert x.contains(4)
x=[0,4,7]; assert (7 in x)
x=[0,4,7]; x.each{println(it + " sq = " + (it*it))}
```

Mengen können als spezielle (ungeordnete) Listen aufgefasst werden, bei denen jedes Element nur einmal vorkommt und die Reihenfolge egal ist:

```
Set x=[0,4,7], y= [7,4,0]; assert x == y
Set z=[0,4,7,4,0]; assert z == y
```

Relationen und Funktionen

Eine *Relation* zwischen zwei Mengen M_1 und M_2 ist eine Teilmenge von $M_1 \times M_2$. Wenn zum Beispiel M={Anna, Beate} und J={Claus, Dirk, Erich}, so ist *liebt*={(Anna, Claus), (Beate, Dirk), (Beate, Erich)} eine Relation zwischen M und J. Relationen schreibt man meist in Infixnotation, d.h., statt (Beate, Dirk)$\in$*liebt* schreibt man (Beate *liebt* Dirk). Falls $M_1 = M_2 = M$, so sagen wir, dass die Relation über M definiert ist. Typische Beispiele sind die Relationen $\leq$ und = über den natürlichen Zahlen, oder die Verbindungsrelation zwischen Städten im Streckennetz der Air Berlin. Eine Relation R heißt (links-)total, wenn es zu jedem $x \in M_1$ ein $y \in M_2$ mit (xRy) gibt. Sie heißt (rechts-)eindeutig, wenn es zu jedem

$x \in M_1$ höchstens ein $y \in M_2$ mit (xRy) gibt. Eine eindeutige Relation nennt man *Abbildung* oder *partielle Funktion*, eine totale und eindeutige Relation heißt *Funktion*. Bei Funktionen schreiben wir $f : M_1 \longrightarrow M_2$ und $f(x) = y$ für $f \subseteq M_1 \times M_2$ und $(x, y) \in f$. Die Menge der $x \in M_1$, für die es ein $y \in M_2$ mit (xRy) gibt, heißt der *Definitionsbereich* oder *Urbildbereich* (domain) der Abbildung; die Menge der $y \in M_2$, für die es ein $x \in M_1$ mit (xRy) gibt, heißt der *Wertebereich* oder *Bildbereich* (range) der Funktion oder Abbildung.
Eine Funktion mit endlichem Definitionsbereich lässt sich angeben durch Auflistung der Menge der Paare (x,y) mit $f(x) = y$. In Groovy nennt man eine solche Funktion *Map* und notiert sie [x_1:y_1, x_2:y_2,....,x_n:y_n], also z.B.

```
[1:5, 2:10, 3:15] oder
["Name":"Anton", "id":573328], aber auch
[3.14:'a', "a":2010, 10:10, 7e5:0]
```

Beachte: `[1:5, 2:10, 3:15] == [2:10, 3:15, 1:5]`

Hier wieder einige Groovy Tatsachen über Maps (http://groovy.codehaus.org/JN1035-Maps):

```
assert [A:3, B:1, N:2]
assert [A:3, B:1, N:2].B == 1
assert [A:3, B:1, N:2] == [B:1, A:3, N:2]
assert [A:3, B:1, N:2] + [C:5] == [A:3, B:1, N:2, C:5]
assert [A:3, B:1, N:2] + [A:1] == [A:1, B:1, N:2]
// [A:3, B:1, N:2] - [A:1] ist undefiniert
// [97:"a", 98:"b", 99:"c"].98 == 'b' ist ein Syntaxfehler
assert [97:"a", 98:"b", 99:"c"].get(98) == "b"
assert [97:"a", 98:"b", 99:"c"].get(100) == null
x=[:]; x[97]="a"; x[98]="b"; assert x == [97:"a", 98:"b"]
```

Genau wie oben lassen sich auch die Begriffe Relation und Funktion verallgemeinern. Eine *n*-stellige Relation zwischen den Mengen $M_1, M_2, \cdots, M_n$ ist eine Teilmenge von $M_1 \times M_2 \times \cdots \times M_n$. Einstellige Relationen heißen auch *Prädikate*. Auch für Prädikate schreiben wir (P*x*) anstatt von (*x*) ∈ P. Eine *n*-stellige Funktion f von $M_1, M_2, \cdots, M_n$ nach M ist eine (n+1)-stellige Relation zwischen $M_1, M_2, \cdots, M_n$ und M, so dass für jedes *n*-Tupel $(x_1, x_2, ..., x_n) \in M_1 \times M_2 \times \cdots \times M_n$ genau ein $y \in M$ existiert mit $(x_1, x_2, ..., x_n, y) \in f$.

Eine Funktion $f : M \times M \times \cdots \times M \longrightarrow M$ heißt (*n*-stellige) *Operation* auf M. Beispiele für zweistellige Operationen sind + und * auf $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ und $\mathbb{R}$. Die Differenz – ist auf $\mathbb{Z}$, $\mathbb{Q}$ und $\mathbb{R}$ eine Operation, auf $\mathbb{N}$ ist sie nur partiell (nicht total). Die Division ist in jedem Fall nur partiell. Typische Prädikate auf natürlichen Zahlen sind `prim` oder `even`.

Wir haben Funktionen als spezielle Relationen definiert, Es gibt auch die Auffassung, dass der Begriff „Funktion“ grundlegender sei als der Begriff „Relation“, und dass Relationen eine spezielle Art von Funktionen sind. Sei $M_1 \subseteq M_2$. Dann ist die *charakteristische Funktion* $\zeta : M_2 \longrightarrow \mathbb{B}$ von M_1 in M_2 definiert durch $\zeta(x) =$ `true` falls $x \in M_1$ und $\zeta(x) =$ `false` falls $x \notin M_1$. Mit Hilfe der charakteristischen Funktion kann jede Relation zwischen den Mengen $M_1, M_2, \cdots, M_n$ als Funktion von $M_1, M_2, \cdots, M_n$ nach $\mathbb{B}$ aufgefasst werden. Diese Auffassung findet man häufig in Programmiersprachen, bei denen Prädikate als boolesche Funktionen realisiert werden.

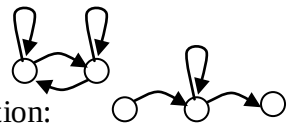
Ordnungen

Die Relation R über M heißt

- *reflexiv*, wenn für alle $x \in M$ gilt, dass xRx.
- *irreflexiv*, wenn für kein $x \in M$ gilt dass xRx.
- *transitiv*, wenn für alle $x, y, z \in M$ mit xRy und yRz gilt, dass xRz.
- *symmetrisch*, wenn für alle $x, y \in M$ mit xRy gilt, dass yRx.
- *antisymmetrisch*, wenn für alle $x, y \in M$ mit xRy und yRx gilt $x = y$.

Eine reflexive, transitive, symmetrische Relation heißt *Äquivalenzrelation*.
Eine reflexive, transitive, antisymmetrische Relation heißt *Halbordnung* oder *partielle Ordnung*. Eine irreflexive, transitive, antisymmetrische Relation heißt *strikte Halbordnung*.
Eine partielle Ordnung heißt *totale* oder *lineare* Ordnung, wenn für alle $x, y \in M$ gilt, dass xRy oder yRx. Bei einer totalen Ordnung lassen sich alle Elemente „der Reihe nach" anordnen. Die Relation $\leq$ ist eine totale Ordnung auf natürlichen und reellen Zahlen, nicht aber auf komplexen Zahlen. Ein einfacheres Beispiel für eine Halbordnung über Zahlen ist die Relation „ist Teiler von".

Beispiel für reflexive, aber nicht antisymmetrische Relation:
Beispiel für eine antisymmetrische, aber nicht reflexive Relation:



1.2 Induktive Definitionen und Beweise

Für fast alle in der Informatik wichtigen Datentypen besteht ein direkter Zusammenhang zwischen ihrer rekursiven Definition (ihrem rekursiven Aufbau) und induktiven Beweisen von Eigenschaften dieser Daten. Wir wollen uns diese Dualität am Beispiel der natürlichen Zahlen betrachten.
Die natürlichen Zahlen lassen sich durch die folgenden so genannten ***Peano-Axiome*** definieren:

- 1 ist eine natürliche Zahl.
- Für jede natürliche Zahl gibt es genau eine natürliche Zahl als Nachfolger.
- Verschiedene natürliche Zahlen haben auch verschiedene Nachfolger.
- 1 ist nicht der Nachfolger irgendeiner natürlichen Zahl.
- Sei P eine Menge natürlicher Zahlen mit folgenden Eigenschaften:
  - 1 ist in P
  - Für jede Zahl in P ist auch ihr Nachfolger in P.

  Dann enthält P alle natürlichen Zahlen.

Das letzte dieser Axiome ist das so genannte *Induktionsaxiom*. Es wird oft in der Form gebraucht, dass P eine Eigenschaft natürlicher Zahlen ist:

- Sei P eine Eigenschaft natürlicher Zahlen, so dass P(1) gilt und aus P(i) folgt P(i+1). Dann gilt P für alle natürlichen Zahlen.

Der erste Mathematiker, der einen formalen Beweis durch vollständige Induktion angab, war der italienische Geistliche Franciscus Maurolicus (1494 -1575). In seinem 1575 veröffentlichten Buch „Arithmetik" benutzte Maurolicus die vollständige Induktion unter

anderem dazu, zu zeigen, dass alle Quadratzahlen sich als Summe der ungeraden Zahlen bis zum doppelten ihrer Wurzel ergeben:

*1 + 3 + 5 + ... + (2n-1)=n*n*

Beweis: Sei P die Menge natürlicher Zahlen, die diese Gleichung erfüllen. Um zu beweisen, dass P alle natürlichen Zahlen enthält, müssen wir zeigen

- *1=1*1*
- *Wenn 1 + 3 + 5 + ... + (2n-1)=n*n,*
  dann 1 + 3 + 5 + ... + (2n-1)+(2(n+1)-1)=(n+1)(n+1)*

Die Wahrheit dieser Aussagen ergibt sich durch einfaches Ausrechnen.
Hier ist ein geringfügig komplizierteres Beispiel zum selber machen: Die Summe der Kubikzahlen bis *n* ist das Quadrat der Summe der Zahlen bis *n*.

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i \right)^2$$

Ein wichtiger Gesichtspunkt beim Induktionsaxiom ist, dass die natürlichen Zahlen als „induktiv aufgebaut“ dargestellt werden gemäß den folgenden Regeln:

- $1 \in \mathbb{N}$.
- Wenn $i \in \mathbb{N}$, dann auch $i+1 \in \mathbb{N}$.
- Außer den so erzeugten Objekten enthält $\mathbb{N}$ keine weiteren Zahlen.

Mit anderen Worten, jede Zahl wird erzeugt durch die endlich-oft-malige Anwendung der Operation (+1) auf die Zahl 1.
Das Induktionsaxiom erlaubt es, Funktionen über den natürlichen Zahlen rekursiv zu definieren. Eine rekursive Definition nimmt dabei auf sich selbst Bezug. Solche Definitionen können leicht schief gehen („das Gehalt berechnet sich immer aus dem Gehalt des letzten Jahres plus 3%“ oder „Freiheit ist immer die Freiheit der Andersdenkenden“ oder „GNU is short for »GNU is Not Unix«“). Ein Begriff, der durch solch eine zirkuläre Definition erklärt wird, ist nicht *wohldefiniert*. Eine Funktion ist nur dann wohldefiniert, wenn sich der Funktionswert eindeutig aus den Argumenten ergibt. Das Prinzip der vollständigen Induktion erlaubt es nun, eine Funktion über den natürlichen Zahlen dadurch zu deklarieren, dass man den Funktionswert für n=1 angibt, und indem man zeigt, wie sich der Funktionswert für (n+1) aus dem Funktionswert für n berechnen lässt. Wenn man nämlich für P die Aussage „der Funktionswert ist eindeutig bestimmt“ einsetzt, so besagt dass Induktionsprinzip, dass dann der Funktionswert für alle natürlichen Zahlen eindeutig bestimmt ist. Zum Beispiel lässt sich die Fakultätsfunktion n!=1*2*…*n ohne „Pünktchen“ dadurch definieren, dass wir festlegen

- 1! = 1
- Wenn n!=x, dann ist (n+1)!=(n+1)*x

Eine andere Schreibweise der zweiten Zeile ist

- (n+1)!=(n+1)*n!

Da in dieser Formel „n“ nur ein Stellvertreter für eine beliebige Zahl ist, können wir auch schreiben

- Wenn n>1, dann ist n!=n*(n-1)!

Diese Schreibweise ist sehr nahe an der Schreibweise in Programmiersprachen, beispielsweise (in Groovy):

- `def fac(n){ if (n==1) return(1) else return(n*fac(n-1)) }`
- `def fac(n){ (n==1)? 1 : n*fac(n-1) }`

In der vorgestellten Fassung erlaubt es das Induktionsprinzip nur, bei der Definition von f(n) auf den jeweils vorherigen Wert f(n-1) zurückzugreifen. Eine etwas allgemeinere Fassung ist das *Prinzip der transfiniten Induktion*:

- Sei P eine Eigenschaft natürlicher Zahlen, so dass für alle $x \in \mathbb{N}$ gilt:
  Falls P(y) für alle y<x, so auch P(x). Dann gilt P für alle natürlichen Zahlen.

Dieses Prinzip gilt nicht nur für die natürlichen Zahlen, sondern für beliebige fundierte Ordnungen (in denen es keine unendlich langen absteigenden Ketten gibt). Der Induktionsanfang ergibt sich dadurch, dass es keine kleinere natürliche Zahl als 1 gibt und für die 1 daher nichts vorausgesetzt werden kann. Im Induktionsschritt erlaubt uns dieses Prinzip, auf beliebige vorher behandelte kleinere Zahlen zurückzugreifen. Das Standardbeispiel sind hier die Fibonacci-Zahlen (nach Leonardo di Pisa, filius Bonacci, 1175-1250, der das Dezimalsystem in Europa einführte)

$$fib(n) = \begin{cases} 1, & falls\ n \le 2 \\ fib(n-1) + fib(n-2), & sonst \end{cases}$$

oder, in der programmiersprachlichen Fassung (in Groovy),

```
def fib(n){ (n<=2)? 1 : fib(n-1) + fib(n-2) }
```

Die Werte dieser Funktion sind, der Reihe nach, 1,1,2,3,5,8,13,21,… und sollen das Bevölkerungswachstum von Kaninchenpaaren nachbilden. Als Beispiel für einen Beweis, der auf mehrere Vorgänger zurückgreift, zeigen wir die Formel von Binet:

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Als Lemma benötigen wir, dass für $\xi = \frac{1+\sqrt{5}}{2}$ gilt $\xi + 1 = \xi^2$, und ebenso für $\varsigma = \frac{1-\sqrt{5}}{2}$.
Das sieht man durch einfaches Ausrechnen, ebenso die Gültigkeit der Aussage für n=1,2.
Damit können wir als Induktionsannahme voraussetzen, dass $fib(n-1) = \frac{1}{\sqrt{5}}\left[\xi^{n-1} - \varsigma^{n-1}\right]$ und

$fib(n-2) = \frac{1}{\sqrt{5}}\left[\xi^{n-2} - \varsigma^{n-2}\right]$. Mit $fib(n) = fib(n-1) + fib(n-2)$ ergibt sich

$fib(n) = \frac{1}{\sqrt{5}}\left[\xi^{n-1} + \xi^{n-2} - \varsigma^{n-1} - \varsigma^{n-2}\right]$, d.h.,

$fib(n) = \frac{1}{\sqrt{5}}\left[(\xi+1)\xi^{n-2} - (\varsigma+1)\varsigma^{n-2}\right] = \frac{1}{\sqrt{5}}\left[\xi^2\xi^{n-2} - \varsigma^2\varsigma^{n-2}\right] = \frac{1}{\sqrt{5}}\left[\xi^n - \varsigma^n\right]$, was zu zeigen war.

Die Berechnung der Fibonacci-Zahlen mit der Formel von Binet geht erheblich schneller als mittels der rekursiven Definition:

```
def binet(n){
 ((((1 + Math.sqrt(5))/2)**n - ((1-Math.sqrt(5))/2)**n)/Math.sqrt(5))}
```

`binet(50)` ergibt sofort `1.258626902500002E10`

Im Allgemeinen ist es nicht immer möglich, solch eine geschlossene (nichtrekursive) Formel für eine rekursiv definierte Funktion zu finden.

Wir haben das Induktionsprinzip für natürliche Zahlen und der fundierten Ordnungsrelation < angewendet. Dies ist nur ein Spezialfall des folgenden allgemeinen Prinzips für induktiv erzeugte Mengen.

Das sind Mengen, die definiert werden durch

- die explizite Angabe gewisser Elemente der Menge,
- Regeln zur Erzeugung weiterer Elemente aus schon vorhandenen Elementen der Menge

sowie der expliziten oder impliziten Annahme, dass die Menge nur die so erzeugten Elemente enthält.

Für induktiv erzeugte Mengen gilt folgendes allgemeine Induktionsprinzip:

Sei P eine Eigenschaft, die Elemente der Menge haben können oder nicht, so dass

- P für alle explizit angegebenen Elemente der Menge gilt, und
- P für alle gemäß den Bildungsregeln erzeugten Elemente gilt, falls es für die bei der Erzeugung verwendeten Elemente gilt.

Dann gilt P für alle Elemente der Menge.

Beispiele für dieses Erzeugungsprinzip werden später betrachtet.

1.3 Alphabete, Wörter, Bäume, Graphen

Unter einem *Alphabet* **A** versteht man eine endliche Menge von *Zeichen* $\mathbf{A}=\{a_1, \ldots, a_n\}$. Das bekannteste Beispiel ist sicher das lateinische Alphabet mit den Zeichen A, B, C, …, Z. Aber bereits davon gibt es verschiedene Varianten, man denke nur an das deutsche Alphabet mit Umlauten ä, ö, ü und der Ligatur ß. Im Laufe der Zeit haben sich bei den Völkern Hunderte von Alphabeten gebildet, von Keilschriften und Hieroglyphen bis hin zu Runen- und Geheimschriften (http://www.schriftgrad.de/). Das chinesische Alphabet umfasst etwa 56000 Zeichen, im Alltag kann man mit 6.000 Schriftzeichen schon relativ gut auskommen; der chinesische Zeichensatz für Computer enthält 7.445 Schriftzeichen. Der ASCII-Zeichensatz enthält 128 bzw. (in der erweiterten Form Latin-1) 256 Druckzeichen, siehe Tabelle. Der Unicode- oder UCS-Zeichensatz umfasst etwa 100.000 Zeichen http://de.wikipedia.org/wiki/Unicode. Man beachte, dass in manchen Alphabeten das Leerzeichen als ein Zeichen enthalten ist; als Ersatzdarstellung wählt man häufig eine Unterstrich-Variante. Ein für die Informatik wichtiges Alphabet ist die Menge $\mathbb{B}$ der Wahrheitswerte.

									9 Tab						
32 leer	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 □

Eine (endliche) Folge $w \in \mathbf{A}^*$ von Zeichen über einem Alphabet **A** heißt *Wort* oder *Zeichenreihe* (string) über **A**. Normalerweise schreibt man, wenn es sich um Wörter handelt, statt $w = (a_1,a_2,\ldots,a_n)$ kurz $w = "a_1a_2\ldots a_n"$, manchmal werden die Anführungszeichen auch weggelassen. Die leere Zeichenreihe wird mit dem Symbol ε oder mit "" bezeichnet. Über Wörtern ist die Konkatenation (Hintereinanderschreibung) als Operation definiert: Wenn $v = a_1a_2\ldots a_n$ und $w = b_1b_2\ldots b_m$, dann ist $v \circ w = a_1a_2\ldots a_nb_1b_2\ldots b_m$. Da die Operation $\circ$ assoziativ ist $((u \circ v) \circ w = u \circ (v \circ w))$, wird das Operationssymbol $\circ$ manchmal auch einfach weggelassen.Die leere Zeichenreihe ε ist bezüglich $\circ$ ein neutrales Element $(w \circ \varepsilon = \varepsilon \circ w = w)$.

Eine Menge mit einer assoziativen Operation und einem neutralem Element nennt man auch *Monoid*; da die Menge $\mathbf{A}^*$ (mit der Operation $\circ$ und dem neutralen Element ε) keinen weiteren Einschränkungen unterliegt, heißt sie auch der *freie Monoid* über **A** (wenn $a_1a_2\ldots a_n = b_1b_2\ldots b_m$, mit ($a_i, b_i \in A$), so ist n=m und $a_1= b_1$ und… und $a_n= b_n$).

Die Menge der Wörter über einem gegebenen Alphabet lässt sich auch induktiv erzeugen:

- $\varepsilon \in \mathbf{A}^*$
- Wenn $a \in \mathbf{A}$ und $w \in \mathbf{A}^*$, so ist $(a \circ w) \in \mathbf{A}^*$.

Hierbei bezeichnet $(a \circ w)$ diejenige Zeichenreihe, die als erstes Zeichen a enthält und danach das Wort w. Alternativ dazu hätten wir Wörter induktiv durch das Anfügen (append) von Zeichen an Zeichenreihen erzeugen können. Diese Charakterisierung der Menge der Zeichenreihen erlaubt es, induktive Beweise zu führen und rekursive Funktionen über Wörtern zu definieren.

Sei first(w) die partielle Funktion, die zu einem nichtleeren Wort dessen erstes Zeichen liefert, und rest(w) die Funktion, die das Wort ohne das erste Zeichen liefert. Programmiersprachlich etwa

```
def first(w){w[0]}
def rest(w){w.substring(1)}
```

Hier sind ein paar rekursiv definierte Funktionen über Wörtern.

```
def laenge(w){if(w=="") return(0) else return(1+laenge(rest(w)))}
```

liefert die Länge eines Wortes. In Groovy/Java schreibt man `""` für die leere Zeichenreihe ε, und + für das Konkatenationssymbol $\circ$.

```
def invertiere(w){
  if(w=="") return(w) else return(invertiere(rest(w))+first(w))}
```

liefert das umgedrehte Wort, also etwa `'negaldnurG'` zu `'Grundlagen'`.

```
def ersetze(w,a,b){
  if(w=="") return(w) else
  if(first(w)==a) return(b+ersetze(rest(w),a,b)) else
    return(first(w)+ersetze(rest(w),a,b))}
```

ersetzt jedes `a` in `w` durch `b`, z.B. ergibt `ersetze("hallo",'l','r')=="harro"`.
Wir werden später noch ähnliche solche Funktionen kennen lernen.
Wenn $w=u \circ v$, so sagen wir, dass u ein *Anfangswort* von w ist. Wenn $w=v1 \circ u \circ v2$, so nennen wir u ein *Teilwort* von w. Auch die Anfangswortrelation lässt sich leicht induktiv definieren:

```
def anfangswort(w,u){
  if(u=="") return(true)
  else if(first(w)!= first(u)) return(false)
  else return(anfangswort(rest(w), rest(u)))}
```

Auf Alphabeten ist häufig eine totale Ordnungsrelation erklärt; meist wird diese durch die Reihenfolge der Aufschreibung der Zeichen unterstellt. Wenn **A** ein Alphabet mit einer totalen Ordnungsrelation $\leq$ ist, so kann $\leq$ zur *lexikographischen Ordnung* auf $\mathbf{A}^*$ ausgeweitet werden:
Sei $x= x_1x_2\ldots x_n$ und $y= y_1y_2\ldots y_m$. Dann gilt $x \leq y$, wenn

- x ein Anfangswort von y ist, oder
- es gibt ein Anfangswort z von x und y ($x=z\circ x'$, $y=z\circ y'$) und first(x') <first(y').

($a < b$ bedeutet $a \leq b$ und nicht $a = b$)
Beispiele für lexikographisch geordnete Wörter über dem lateinischen Alphabet sind "ANTON" < "BERTA", "AACHEN" < "AAL", "AAL" < "AALBORG" und ε < "A".

Es ist nicht schwer zu sehen, dass die lexikographische Ordnung eine totale Ordnung ist. Die Buchstaben des deutschen Alphabets sind nicht linear geordnet (a und ä stehen nebeneinander, ß ist nicht eingeordnet), daher entspricht die Reihenfolge der Wörter in einem deutschen Lexikon nicht der lexikographischen Ordnung.

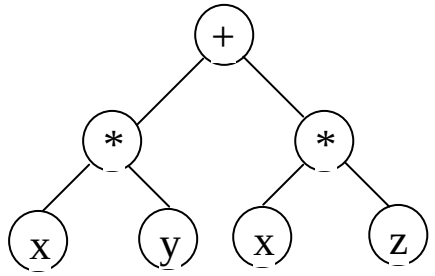
Bäume

Bäume sind – neben Tupeln, Folgen und Wörtern – eine weitere in der Informatik sehr wichtige Datenstruktur. In der induktiven Definition von Zeichenreihen besteht ein Wort w aus der Konkatenation von first(w) mit rest(w). Ein Binärbaum ist dadurch gekennzeichnet, dass es zwei verschiedene „Reste" gibt: den linken und den rechten Unterbaum. Daraus ergibt sich folgende induktive Definition der Menge der Binärbäume über einem gegebenen Alphabet **A**:

- $\varepsilon \in \mathbf{A}^\wedge$
- Wenn $a \in \mathbf{A}$ und $l \in \mathbf{A}^\wedge$ und $r \in \mathbf{A}^\wedge$, so ist $(a,l,r) \in \mathbf{A}^\wedge$.

a heißt *Wurzel*, l und r sind *Unterbäume* des Baumes (a,l,r). Die Wurzeln von l und r heißen die *Kinder* oder *Nachfolger* von a. Ein Baum y ist *Teilbaum* eines Baumes x, wenn $x=y$ oder y Teilbaum eines Unterbaumes von x ist. Wenn y nichtleerer Teilbaum von x ist, so sagen wir, die Wurzel von y ist ein *Knoten* von x. Ein Knoten ohne Nachfolger (d.h. ein Teilbaum der Gestalt $(a, \varepsilon, \varepsilon)$) heißt *Blatt.*

Als Beispiel für Bäume betrachten wir Formelbäume über dem Alphabet (x,y,z,+,*). Die Formel x*y + x*z (mit „Punkt-vor-Strich-Regelung") kann durch den Baum (+,(*,(x, ε , ε),(y, ε , ε)),(*,(x, ε , ε),(z, ε , ε))) repräsentiert werden. Übersichtlicher ist eine graphische Darstellung:

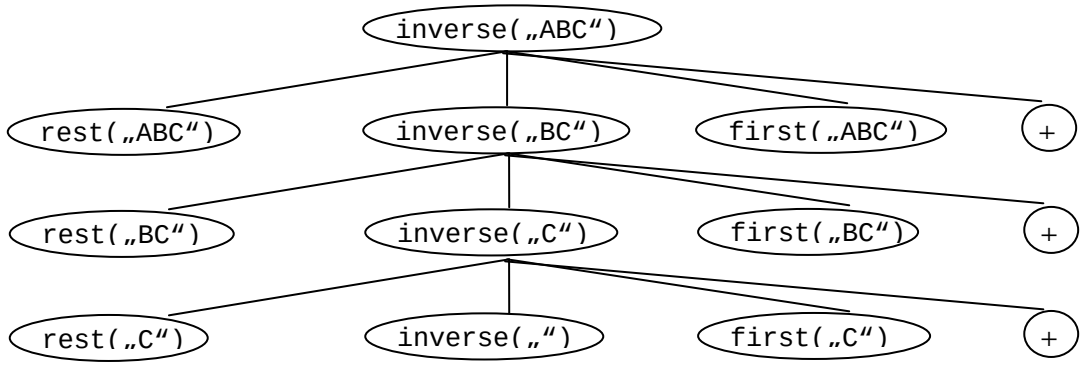


Wir werden später verschiedene Algorithmen, die auf Bäumen basieren, kennen lernen.
Es ist klar, dass sich die obige Definition direkt auf Binärbäume über einer beliebigen Grundmenge verallgemeinern lässt. Eine weitere nahe liegende Erweiterung sind *n-äre Bäume*, bei denen jeder Knoten entweder keinen oder n Nachfolger hat. Wenn wir erlauben, dass jeder Knoten eine beliebige (endliche) Zahl von Nachfolgern haben kann, sprechen wir von *endlich verzweigten Bäumen*.

Aufrufbäume

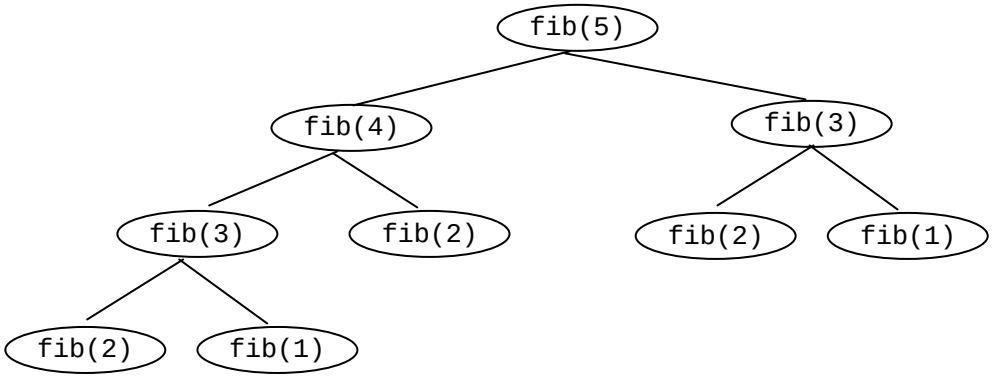
Eine spezielle Art von (endlich verzweigten) Bäumen sind die *Aufrufbäume* einer rekursiven Funktion. Die Wurzel eines Aufrufbaumes ist der Name der Funktion mit den Eingabewerten. Die Nachfolger jeden Knotens sind die bei der Auswertung aufgerufenen Funktionen mit ihren Eingabewerten.

Beispiel:



(In diesem Baum haben wir die Funktionen „==“ und „if“ nicht weiter berücksichtigt.)
Beim *verkürzten Aufrufbaum* lässt man alle Knoten weg außer denen, die die rekursive Funktion selbst betreffen.

Beispiel:



Unter der *Aufrufkomplexität* einer rekursiven Funktion verstehen wir die Anzahl der Knoten im verkürzten Aufrufbaum. Die Zeit, die benötigt wird, um eine rekursive Funktion zu berechnen, hängt im Wesentlichen von der Aufrufkomplexität ab. Als Beispiel betrachten wir die Aufrufkomplexität der Fibonacci-Funktion. Aus obigem Beispiel ist sofort klar:

$$fibComp(n) = \begin{cases} 1, & falls\ n \le 2 \\ 1 + fibComp(n-1) + fibComp(n-2), & sonst \end{cases}$$

Die rekursive Formulierung hilft leider noch nicht, die Aufrufkomplexität abzuschätzen. Per Induktion nach *n* zeigen wir: $fibComp(n) = 2 * fib(n) - 1$. Für *n*=1,2 ist dies klar, für *n*>2 gilt

$$fibComp(n) = 1 + fibComp(n-1) + fibComp(n-2) =^{I.V.} 1 + 2 * fib(n-1) - 1 + 2 * fib(n-2) - 1$$
$$= 2 * (fib(n-1) + fib(n-2)) - 1 = 2 * fib(n) - 1$$

Mit der früher bewiesenen Gleichung von Binet erhalten wir

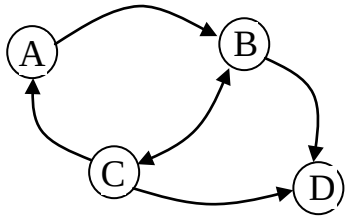
$$fibComp(n) = \frac{2}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] - 1.$$

Eine Wertetabelle für einige Zahlenwerte ist nachfolgend angegeben. Daraus folgt: wenn in einer Sekunde 10.000 Aufrufe erfolgen, benötigt die Rechnung für n=100 etwa 2,2 Milliarden Jahre! (üblicherweise ist vorher der Speicher erschöpft oder ein Zahlbereichsüberlauf eingetreten,).

n	3	5	10	20	30	40	50	60	70	80	90	100
fibComp(n)	3	9	109	13529	$1.6*10^6$	$2*10^8$	$2.5*10^{10}$	$3*10^{12}$	$3.8*10^{14}$	$4.6*10^{16}$	$5.7*10^{18}$	$7*10^{20}$

Graphen

Während ein Wort in der Informatik nur eine spezielle Art von Folgen ist, versteht man unter einem *Graphen* nur eine spezielle Art von Relationen: Ein Graph ist die bildliche Darstellung einer binären Relationen über einer endlichen Grundmenge. Die Elemente der Grundmenge werden dabei in Kreisen (*Knoten*) gezeigt. Zwischen je zwei Knoten zeichnet man einen Pfeil (eine *Kante*), falls das betreffende Paar von Elementen in der Relation enthalten ist. Beispiel:



Dies ist die Relation {(A,B),(B,C),(C,B),(C,A),(B,D),(C,D)}. Für symmetrische Relationen weisen die Pfeile immer in beide Richtungen; man spricht hier von *ungerichteten Graphen*. Eine Alternative zur obigen Definition besteht darin, einen Graphen als Tupel (V,E) zu definieren, wobei V eine endliche Menge von Knoten (vertices) und E eine endliche Menge von Kanten (edges) ist, so dass zu jeder Kante genau ein *Anfangs*- und ein *Endknoten* gehört. Knoten, die nicht Endknoten sind, heißen *Quelle*, Knoten, die nicht Anfangsknoten sind, heißen *Senke* im Graphen. Knoten, die weder Anfangs- noch Endknoten sind, heißen *isoliert*. Eine dritte Art der Definition von Graphen ist durch die so genannte Adjazenzmatrix: Nach dieser Auffassung ist ein Graph eine endliche Matrix (Tabelle) mit booleschen Werten. Die Zeilen und Spalten der Tabelle sind dabei mit der Grundmenge beschriftet; ein Eintrag gibt an, ob das entsprechende Paar (Zeile, Spalte) in der Relation enthalten ist oder nicht.

	A	B	C	D
A		X		
B			X	X
C	X	X		X
D				

Im Gegensatz zu Bäumen können Graphen Zyklen enthalten, daher existiert keine einfache induktive Definition. Umgekehrt können endlich verzweigte Bäume als zyklenfreie Graphen mit nur einer Quelle betrachtet werden.