Skript

Vorkurs Informatik

Sommersemester 2018

PD Dr. David Sabel
Ronja Düffel
Mario Holldack
Stand: 7. März 2018

Inhaltsverzeichnis

1. Einführung **5**

2. Mengen **7**
2.1. Grundsätzliche Eigenschaften von Mengen 7
2.2. Mengenalgebra 8
2.2.1. Rechenregeln für Schnitt und Vereinigung 11

3. Aussagenlogik **13**
3.1. Syntax und Semantik der Aussagenlogik 14
3.1.1. Syntax 14
3.1.2. Semantik 15
3.2. Erfüllbarkeit, Allgemeingültigkeit und Äquivalenz 17
3.3. Fundamentale Rechenregeln 18
3.4. Quantoren 19
3.4.1. Negation von Aussagen 21

4. Beweistechniken **23**
4.1. Direkter Beweis 23
4.1.1. Abgeschlossenheit einer Zahlenmenge bezüglich einer Verknüpfung 25
4.2. Indirekter Beweis / Beweis durch Kontraposition 26
4.3. Beweis durch Widerspruch 27

5. Induktion und Rekursion **29**
5.1. Vollständige Induktion 29
5.1.1. Wann kann man vollständige Induktion anwenden? 32
5.1.2. Was kann schief gehen? 32
5.2. Rekursion 33
5.2.1. Wozu Rekursion? 34
Türme von Hanoi 36

6. Einführung in die Bedienung von Unix-Systemen **41**
6.1. Unix und Linux 41
6.1.1. Dateien und Verzeichnisse 42
6.1.2. Login und Shell 43
6.1.3. Befehle 43
6.1.4. History und Autovervollständigung 46
6.2. Editieren und Textdateien 47

7. Programmieren und Programmiersprachen **51**
7.1. Programme und Programmiersprachen 51
7.1.1. Imperative Programmiersprachen 52
7.1.2. Deklarative Programmiersprachen 53
7.2. Haskell: Einführung in die Benutzung 54
7.2.1. GHCi auf den Rechnern der RBI 54
7.2.2. GHCi auf dem eigenen Rechner installieren 54
7.2.3. Bedienung des Interpreters 55
7.2.4. Quelltexte erstellen und im GHCi laden 56

7.2.5. Kommentare in Quelltexten . . . 58
7.2.6. Fehler . . . 59

8. Grundlagen der Programmierung in Haskell **61**
8.1. Ausdrücke und Typen . . . 61
8.2. Basistypen . . . 62
8.2.1. Wahrheitswerte: Der Datentyp `Bool` . . . 62
8.2.2. Ganze Zahlen: `Int` und `Integer` . . . 64
8.2.3. Gleitkommazahlen: `Float` und `Double` . . . 65
8.2.4. Zeichen und Zeichenketten . . . 65
8.3. Funktionen und Funktionstypen . . . 66
8.4. Einfache Funktionen definieren . . . 70
8.5. Rekursion . . . 76
8.6. Listen . . . 81
8.6.1. Listen konstruieren . . . 82
8.6.2. Listen zerlegen . . . 84
8.6.3. Einige vordefinierte Listenfunktionen . . . 86
8.6.4. Nochmal Strings . . . 86
8.7. Paare und Tupel . . . 87
8.7.1. Die Türme von Hanoi in Haskell . . . 88

A. Kochbuch **91**
A.1. Erste Schritte . . . 91
A.2. Remote Login . . . 97
A.2.1. Unix-artige Betriebssysteme (Linux, MacOS, etc) . . . 97
A.2.2. Windows . . . 99

1. Einführung

"In der Informatik geht es genau so wenig um Computer wie in der Astronomie um Teleskope"
Edsger W. Dijkstra (1930-2002), Niederländischer Informatiker

Vielmehr ist der Computer lediglich ein Hilfsmittel dessen sich die Informatik bedient. Anstatt selbst zu rechnen, lässt man die Rechenmaschine (Computer) rechnen. Daher befasst sich die Informatik auch nicht mit der Bedienung von Software, sondern u.a. mit dem Entwurf derselbigen.

"Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mithilfe von Computern". (Duden Informatik).

Da wir es in der Informatik mit Computern zu tun haben, die Anweisungen nicht interpretieren können, sondern lediglich ausführen, ist es notwendig präzise zu Arbeiten und zu Formulieren und ganz sicher zu sein, dass eine entworfene Lösung in jeder auftretenden Situation funktioniert. Fehler in Software und Hardware haben in der Vergangenheit immer wieder zu großem wirtschaftlichen Schaden und auch zu Todesfällen geführt.

Hier einige Beispiele:

Der Pentium Prozessor Divisions-Fehler 1994

Für $x = 4195835.0$ und $y = 3145727.0$ und $z = x - \frac{x}{y} \times y$ liefert ein fehlerfreier Prozessor bei exakter Rechnung $z = 0$. Der Pentium Prozessor lieferte als Ergebnis $z = 256$.
Erklärung: Intel verwendete einen speziellen Divisions-Algorithmus, der den Vorteil hatte, dass pro Takt 2 Bits des Quotienten bestimmt werden konnten und den Chip somit schneller machte. Allerdings sollte die Schätzung für die nächste Stelle aus einer Tabelle gelesen werden, die 1066 Einträge haben sollte. Durch eine fehlerhafte `for`-Schleife wurden nur 1061 Einträge auf den Chip aufgenommen, dadurch kam es zu falschen Ergebnissen bei der Gleitkomma-Division.
Laut Intel trete der Fehler lediglich alle 27 000 Jahre einmal auf, IBM, die einen eigenen Chip herstellten, verkündete der Fehler trete ca alle 24 Tage auf. Da der Fehler erst nach einem Jahr entdeckt wurde, obwohl der Chip in vielen Geräten verbaut war, ist IBMs Behauptung schwer zu glauben. Letztendlich kostete dieser Fehler Intel über 400 Millionen Dollar.

Therac-25 Bestrahlungsunfälle

Zwischen 1985 und 1987 kam es zu mehreren Unfällen mit dem medizinischen Bestrahlungsgerät *Therac-25*. Infolge einer Überdosis mussten mehreren Patienten Organe entfernt werden und drei Patienten starben sogar. Mehrere Fehler im Kontrollprogramm des Geräts führten zu den Unfällen.

Ariane 5

Die Ariane 5 Rakete der ESA musste am 4.Juni 1996 eine Minute nach dem Start gesprengt werden, da sie vom Kurs abgekommen war und sich nicht mehr steuern ließ.
Erklärung: Für die Steuerung der Ariane 5 hatte man die Steuerungssoftware der Ariane 4 übernommen. Die Ariane 5 war aber größer und erreichte eine höhere Geschwindigkeit als die Ariane 4. Dadurch kam es in der Kursberechnung zu Zahlen, die nicht vorgesehen waren und es kam zu einem Speicherüberlauf, die Zahlen konnten von der Software nicht mehr korrekt interpretiert werden. Die Steuerungssoftware korrigierte die vermeintlich falsche Flugbahn. Der Schaden betrug etwa 370 Millionen Dollar.

Flughafen Denver, Koffertransportsystem

Im Oktober 1993 sollte der neue Flughafen, ausgestattet mit einem vollautomatischen Koffertransportsystem, eröffnet werden. Wegen Schwierigkeiten mit dem vollautomatischen Gepäcktransportsystem verzögerte sich die Eröffnung um 16 Monate. Statt einer großen Gesamtlösung hatte jeder Flugsteig sein eignes, unterschiedlich stark automatisiertes Gepäcktransportsystem. Lediglich die größte Airline am Flughafen, United, benutzte das automatisierte Gepäcksystem für Abflüge und zahlte ca 1 Mio Dollar Instandhaltungskosten pro Monat. 2005 entschied United das automatisierte System aufzugeben.
Im Nachhinein stellte sich heraus, dass u.a. das Gesamtproblem zu komplex war und zu viele Nachrichten über das Netz verschickt werden mussten, sodass viele Sortier-Anweisungen nicht rechtzeitig ankamen.
Der Schaden wird auf 3,2 Milliarden Dollar geschätzt.

Wir wollen also Probleme und Aufgaben unseres täglichen Lebens mit Hilfe von Computern bearbeiten und lösen. Dafür müssen wir die, meist in Alltagssprache formulierten Wünsche und Vorstellungen eines Kunden modellieren und formalisieren, um dann eine Lösung zu entwerfen, die tatsächlich immer funktioniert, alle Eventualitäten berücksichtigt und im Idealfall verifizierbar ist.

Also benötigen wir formale Konstrukte und Konzepte, sodass Aussagen bewiesen und die Korrektheit von Systemen verifiziert werden kann. Unsere Alltagssprache eignet sich dafür nicht, da sie zu ungenau, und häufig mehrdeutig ist. Wir benötigen also die Möglichkeit Objekte und Konzepte anhand der, für die Problemstellung relevanten Eigenschaften, zusammenzufassen, sodass wir Aussagen über diese machen können und diese beweisen können.

2. Mengen

Wir benötigen also die Möglichkeit Objekte und Konzepte anhand der, für die Problemstellung relevanten Eigenschaften, zusammenzufassen, sodass wir Aussagen über diese machen können und diese beweisen können.

Definition 2.1 (naiver Mengenbegriff nach Cantor)**.**
Eine *Menge M* ist eine Zusammenfassung von wohlunterschiedenen Objekten unserer Anschauung oder unseres Denkens, welche *Elemente* der Menge M genannt werden, zu einem Ganzen.

Menge
Element

Notation 2.2.

Wir schreiben $m \in M$ um auszudrücken, dass m Element der Menge M ist. $\in$

Wir schreiben $m \notin M$ um auszudrücken, dass m *kein* Element der Menge M ist. $\notin$

Mengen werden in geschweiften Klammern $\{\}$ angegeben. Sie können durch Aufzählen der einzelnen Elemente (*extensional*) oder durch Angabe charakteristischer Eigenschaften der Elemente der Menge (*intensional*) definiert werden.

extensional
intensional

Beispiel 2.3.
extensional angegebene Mengen:

- $M_1 := \{1, 2, 3, 4, 5, 6, 7\} = \{1, 2, \dots, 7\} = \{3, 4, 7, 5, 1, 6, 2\} = \{1, 2, 2, 3, 5, 6, 1, 7\}$
- $M_2 := \{$ich, du, er, sie, es, wir, ihr, sie$\}$
- $\mathbb{N} := \{1, 2, 3, \dots\}$(die Menge der natürlichen Zahlen)
- $M_3 := \{1, Ball, \{l, m\}, (Karo, 7), -2\}$

intensional angegebene Mengen:

- $M_1 : \{x | x \in \mathbb{N}, 1 \leq x \leq 7\}$
- $\mathbb{Q} := \{\frac{a}{b} : a, b \in \mathbb{Z}, b \neq 0\}$ (die Menge der rationalen Zahlen)
- $A := \{i | i \in \mathbb{Z}$ und $-3 \leq i \leq 3\}$

Der vertikale Strich " | " in der intensionalen Mengendefinition wird als "sodass" gelesen und häufig auch durch einen Doppelpunkt ":" ersetzt. Statt "und" wird häufig auch nur ein Komma "," geschrieben. Die oben definierte Menge A, ist also die "Menge aller i, sodass i eine ganze Zahl ist, und einen Wert zwischen -3 und 3".

Achtung!
$\{x | 1 \leq x \leq 10\}$ ist keine eindeutige Definition einer Menge, da nicht angegeben ist, ob x eine ganze, oder eine reele Zahl ist. Bei der intensionalen Mengendefinition ist es ganz wichtig anzugeben, aus welcher Menge x stammt. Das kann auch vor dem vertikalen Strich geschehen, so könnte man die oben definierte Menge A auch als $A := \{i \in \mathbb{Z} | -3 \leq i \leq 3\}$ definieren.

2.1. Grundsätzliche Eigenschaften von Mengen

- alle Elemente einer Menge sind *verschieden*. Das heißt, dass kein Wert oder Objekt nur einmal in einer Menge vorkommen kann, nicht mehrfach.

- die Elemente einer Menge haben *keine* feste Reihenfolge. $\{1,2,3\} = \{3,2,1\} = \{1,1,3,2\}$
- Eine Menge kann auf verschiedene Arten beschrieben werden. $U := \{1,3,5,7\} = \{3,5,7,1\} = \{1,1,5,3,5,7\} = \{x \in \mathbb{N} | 1 \leq x \leq 7, x \text{ ist ungerade}\}$
- Mengen können auch "verschiedenartige" Elemente enthalten. Zum Beispiel besteht die Menge $M := \{7, Haus, (Herz, 3), -4, \{l, m, n\}\}$ aus fünf Elementen: den atomaren Werten 7, -4 und $Haus$, sowie dem Tupel $(Herz, 3)$ und der Menge $\{l, m, n\}$.

An dieser Stelle sei erwähnt, dass Cantors Mengenbegriff problematisch ist und zu Widersprüchen führen kann. Der britische Philosoph Bertrand Russell(1872-1970) gab folgendes Beispiel, das als die **Russellsche Antinomie** bekannt ist.

Sei N die Menge aller Mengen M, die sich selbst nicht enthalten,
d.h. $N := \{M | M \text{ ist eine Menge}, M \notin M\}$
Frage: Ist N in sich selbst enthalten? (also gilt $N \in N$?)
Es gibt nur zwei Fälle zu betrachten. Entweder ist N in N enthalten, oder nicht.

Fall 1: $N \in N$. Laut Definition der Menge N gilt dann aber $N \notin N$. Das ist ein Widerspruch.

Fall 2: $N \notin N$. Laut Definition der Menge N gilt dann aber $N \in N$. Auch das ist ein Widerspruch.

Beide Fälle führen also zu einem Widerspruch, obwohl einer der Fälle ja zutreffen muss, denn welchen anderen Fall sollte es noch geben, außer dass $N \in N$ oder $N \notin N$ ist?

Cantors Mengenbegriff ist also mit Vorsicht zu genießen. Wenn man sich allerdings der Problematik bewusst ist, kann man diese "problematischen Mengen" im Alltag des Informatikstudiums umgehen, daher arbeiten wir hier weiterhin mit dem naiven Mengenbegriff.

2.2. Mengenalgebra

Für den Umgang mit Mengen definieren wir folgende Begriffe:

Definition 2.4 (Teilmenge, Obermenge)**.**
Seien A und B Mengen.

Teilmenge $A \subset B$

A ist genau dann eine **Teilmenge** von B (kurz:$A \subset B$), wenn jedes Element von A auch ein Element von B ist.

echte Teilmenge $A \subsetneq B$

A ist genau dann eine **echte Teilmenge** von B (kurz: $A \subsetneq B$), wenn jedes Element von A auch ein Element von B, aber nicht jedes Element von B auch ein Element von A ist.

Obermenge $A \supseteq B$

A ist genau dann eine **Obermenge** von B (kurz: $A \supseteq B$), wenn B eine Teilmenge von A ist, also $B \subseteq A$ gilt.

echte Obermenge $A \supsetneq B$

A ist genau dann eine **echte Obermenge** von B (kurz: $A \supsetneq B$), wenn B eine echte Teilmenge von von A ist, also $B \subsetneq A$ gilt.

Beispiel 2.5.
Betrachten wir die Mengen $C := \{1,3,5\}$ und $D := \{1,2,3,4,5,6\}$. Dann gilt: $C \subseteq D$, denn jedes Element von C ist auch in D enthalten. Es gilt sogar $C \subsetneq D$, denn nicht nur ist jedes Element von C in D enthalten, es gibt auch Elemente in D, die nicht in C enthalten sind, nämlich die Elemente $2, 4$ und 6. Aussreichend, für die Gültigkeit von $C \subsetneq D$ wäre bereits ein einzelnes Element das zwar in D aber nicht in C enthalten ist.

Die Kritierien für eine echte Teilmenge sind also härter, als die für eine Teilmenge. Jede echte Teilmenge einer Menge M ist eine Teilmenge von M, aber nicht jede Teilmenge von M ist auch eine echte Teilmenge.

Analog gelten für die beiden oben definierten Mengen C und D: $D \supset C$, bzw $D \supsetneq C$.

Definition 2.6 (Gleichheit von Mengen)**.**

A und B sind genau dann **gleich** (kurz: $A = B$), wenn sie **dieselben** Elemente enthalten. Anders ausgedrückt gilt $A = B$ genau dann, wenn $A \subseteq B$ und $B \subseteq A$ gilt.

Gleichheit
$A = B$

Beispiel 2.7.
Für die Mengen $C := \{5, 6, 7\}$ und $D := \{6, 5, 7\}$ gilt $C \subseteq D$, denn jedes Element von C ist auch Element von D. Außerdem gilt $D \subseteq C$, denn jedes Element von D ist auch Element von C und ebenfalls gilt $C = D$.

Für die Mengen $E := \{2, 3, 4\}$ und $F := \{3, 4\}$ gilt $F \subseteq E$, denn jedes Element von F ist auch Element von E, allerdings ist $E \nsubseteq F$, denn nicht jedes Element von E ist auch Element von F. Insbesondere gilt das für das Element 2. $2 \in E$, aber $2 \notin F$. Dementsprechend gilt $E \neq F$.

Satz 2.8.
Seien L,M und P Mengen, für die $L \subseteq M$ und $M \subseteq P$ gilt. Dann gilt auch $L \subseteq P$

Beispiel 2.9.
Wir betrachten die Mengen $L := \{2, 3, 4\}$, $M := \{2, 3, 4, 5\}$ und $P := \{2, 3, 4, 5, 6\}$. Für diese Mengen gilt offensichtlich $L \subseteq M$ und $M \subseteq P$, denn alle Elemente in L sind auch in M enthalten und alle Elemente von M sind auch in P enthalten. Wie leicht zu sehen ist, sind ebenfalls alle Elemente von L in P enhalten, also gilt auch $L \subseteq P$.

Achtung! Das ist lediglich ein Beispiel, um den Sachverhalt zu veranschaulichen. Das ist kein Beweis, dass Satz 2.8 tatsächlich stimmt. Mit dem Beispiel zeigen wir lediglich, dass die Aussage für die angegebenen Mengen L, M und P wahr ist, wir zeigen nicht, dass die Aussage für alle beliebigen Mengen gilt.

Definition 2.10 (Schnitt, Vereinigung, Differenz)**.**
Seien A und B Mengen.

Der **Schnitt** von A und B ist die Menge

Schnitt
$A \cap B$

$$A \cap B := \{x | x \in A \text{ und } x \in B\},$$

also die Menge aller Elemente, die sowohl in A als auch in B enthalten sind.

Die **Vereinigung** von A und B ist die Menge

Vereinigung
$A \cup B$

$$A \cup B := \{x | x \in A \text{ oder } x \in B\},$$

also die Menge aller Elemente, die in mindestens einer der Mengen enthalten sind.

Die **Differenz** von A und B ist die Menge

Differenz
$A \setminus B$

$$A \setminus B := \{x | x \in A \text{ und } x \notin B\},$$

also die Menge aller Elemente von A die nicht in B enthalten sind.

Die **symmetrische Differenz** von A und B ist die Menge

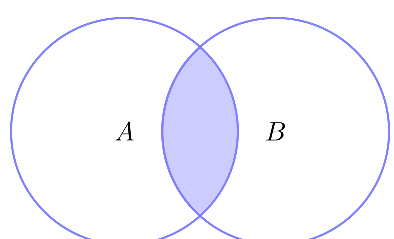
symmetrische Differenz
$A \oplus B$

$$A \oplus B := A \setminus B \cup B \setminus A,$$

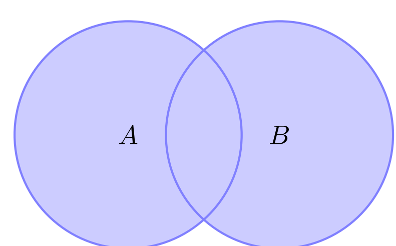
also die Menge aller Elemente, die entweder in A oder in B, aber nicht in beiden gleichzeitig enthalten sind. Statt $A \oplus B$ wird manchmal auch $A \bigtriangleup B$ geschrieben.

Diese Begriffe lassen sich mit *Venn*-Diagrammen veranschaulichen.

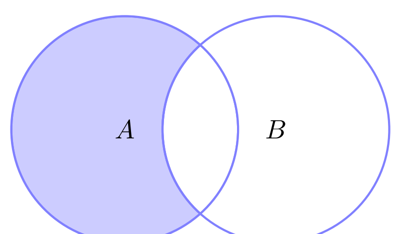
Schnitt $A \cap B$



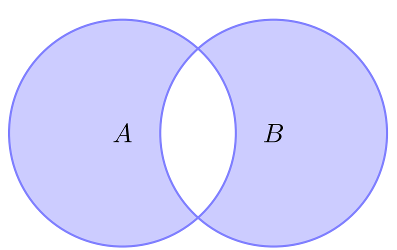
Vereinigung $A \cup B$



Differenz $A \setminus B$



symmetrische Differenz $A \oplus B$



Definition 2.11 (leere Menge, Potenzmenge, disjunkt)**.**
Seien A und B Mengen.

leere Menge $\emptyset$ — Die **leere Menge** ist die eindeutig bestimmte Menge, die kein Element enhält. Wir bezeichnen sie mit $\emptyset$. $\emptyset = \{\}$
Beachte: $\emptyset \neq \{\emptyset\}$, denn $\emptyset$ ist die Menge, die keine Elemente enthält und $\{\emptyset\}$ enthält genau ein Element, nämlich die leere Menge.

Potenzmenge $\mathcal{P}(A)$ — Die **Potenzmenge** von A ist die Menge

$$\mathcal{P}(A) := \{T | T \subseteq A\},$$

also die Menge aller Teilmengen von A.

disjunkt — Wir bezeichnen zwei Mengen A und B als **disjunkt** , wenn $A \cap B = \emptyset$, d.h. wenn sie keine gemeinsamen Elemente haben.

Eigenschaften der leeren Menge
Anhand obiger Definitionen kann man sich folgende Eigenschaften der leeren Menge klar machen.

- Die leere Menge ist Teilmenge jeder Menge: $\emptyset \subseteq A$
- Jede Menge bleibt bei der Vereinigung mit der leeren Menge unverändert: $\emptyset \cup A = A$
- Für jede Menge ist der Durchschnitt mit der leeren Menge die leere Menge: $\emptyset \cap A = \emptyset$
- Die einzige Teilmenge der leeren Menge, ist die leere Menge: aus $A \subseteq \emptyset$ folgt $A = \emptyset$
- Daraus folgt, dass die Potenzmenge der leeren Menge genau ein Element enthält, nämlich die leere Menge selbst: $\mathcal{P}(\emptyset) = \{\emptyset\}$

Beispiel 2.12.
Seien $A := \{1, 2, 3\}$, $B := \{2, 3, 4\}$ und $C := \{4, 5, 6\}$.

$A \cap B = \{2, 3\}$, denn 2 und 3 sind die einzigen Elemente, die sowohl in A als auch in B enthalten sind.

$A \cup B = \{1, 2, 3, 4\}$, denn jedes dieser Elemente kommt in mindestens einer der beiden Mengen A und B vor.

$A \setminus B = \{1\}$, denn 1 ist das einzige Element von A, das nicht in B enthalten ist.

$A \oplus B = \{1, 4\}$ denn 1 und 4 sind die Elemente, die in genau einer der beiden Mengen A und B auftreten (und nicht in beiden). Insbesondere ist $A \setminus B = \{1\}$ (siehe oben) und $B \setminus A = \{4\}$ (4 ist das einzige Element von B, das nicht in A enthalten ist. Die Vereinigung der beiden Mengen $A \setminus B$ und $B \setminus A$ ist $(A \setminus B) \cup (B \setminus A) = \{1, 4\}$.

$A \cap C = \emptyset$, denn A und C haben keine gemeinsamen Elemente. Die Mengen A und C sind *disjunkt*. Daraus folgt auch, dass $A \setminus C = A$ und $C \setminus A = C$ ist.

Die Potenzmenge von A ist: $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$

2.2.1. Rechenregeln für Schnitt und Vereinigung

Satz 2.13.
Seien A,B und C Mengen. Dann gilt:

Idempotenz

Idempotenz:

$$A \cap A = A \text{ und } A \cup A = A.$$

Kommutativität

Kommutativität:

$$A \cap B = B \cap A \text{ und } A \cup B = B \cup A.$$

Assoziativität

Assoziativität:

$$A \cap (B \cap C) = (A \cap B) \cap C \text{ und } A \cup (B \cup C) = (A \cup B) \cup C.$$

Absorption

Absorption:

$$A \cap (A \cup B) = A \text{ und } A \cup (A \cap B) = A.$$

Distributivität

Distributivität:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \text{ und } A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

Definition 2.14 (Mächtigkeit, Kardinalität)**.**

Eine Menge M heißt **endlich**, wenn sie nur endlich viele Elemente enhält, d.h. es gibt eine Zahl $k \in \mathbb{N}$, sodass M genau k viele Elemente enthält.

endliche Menge

Die Anzahl der Elemente einer Menge M bezeichnet man auch als **Mächtigkeit** oder Kardinalität der Menge M (in Zeichen: $|M|$).

Mächtigkeit
Kardinalität
$|M|$

Beispiel 2.15.

- $A := \{4, 2, 7\}, |A| = 3$
- $|\{7, Haus, (Herz, 3), -4, \{l, m, n\}, 9\}| = 6$
- $|\emptyset| = 0$ **aber** $|\{\emptyset\}| = 1$

Satz 2.16.
Seinen L und P endliche Mengen. Es gilt $|L \cup P| = |L| + |P|$ *genau dann, wenn L und P disjunkt sind.*

Beispiel 2.17.
Sei $L := \{2, 3, Haus\}$ und $P := \{Stein, Papier, Schere, 5\}$
Dann ist $L \cap P = \emptyset$, außerdem ist $|L| = 3$ und $|P| = 4$. $L \cup P = \{2, 3, Haus, Stein, Papier, Schere, 5\}$ und $|L \cup P| = 7 = 3 + 4 = |L| + |P|$

Sei $L := \{2, 3, Haus\}$ und $P := \{Stein, Papier, Schere, Haus\}$
Dann ist $L \cap P = \{Haus\}$, außerdem ist $|L| = 3$ und $|P| = 4$. $L \cup P = \{2, 3, Stein, Papier, Schere, Haus\}$ und $|L \cup P| = 6 \neq 3 + 4 = |L| + |P|$.
Was ist hier passiert? Das Element der Schnittmenge ist bei $|L| + |P|$ doppelt gezählt worden, nämlich einmal für die Menge L und einmal für die Menge P.

3. Aussagenlogik

Logik ist die Lehre des vernünftigen Schlussfolgerns, das Teilgebiet der Aussagenlogik befasst sich mit logischen Aussagen und deren Verknüpfung, sowie der Frage, wie man formale Schlüsse ziehen und Beweise durchführen kann. Anwendungsgebiete in der Informatik sind z.B. die Modellierung von Wissen in der künstlichen Intelligenz, die Auswertung von Datenbankanfragen, der Kontrollfluss von Computerprogrammen, Logikbausteine in der technischen Informatik und die Verifikation von Schaltkreisen und Programmen.

Eine logische Aussage ist ein Satz, der entweder *wahr* oder *falsch* (also nie beides(!)) sein kann. Die Zuordnung des Wahrheitswerts bei elementaren Aussagen wie z.B. "*3 ist eine ungerade Zahl*", "*Die Nidda ist der längste Fluss Europas*" oder "*gleiche Ladungen ziehen sich an*", ist nicht Gegenstand der Logik, sondern der entsprechenden Fachwissenschaften. Durch Verknüpfungen wie *und, oder, nicht, wenn... dann...* können elementare Aussagen zu sehr komplexen Aussagen zusammengesetzt werden. Die Aussagenlogik legt fest, wie sich der Wahrheitswert einer zusammengesetzten Aussage ohne zusätzliche Information aus den Wahrheitwerten der Teilaussagen bestimmen läßt.

Definition 3.1 (Logische Aussage)**.**
Eine logische Aussage ist ein Satz, der entweder wahr oder falsch sein kann. Aussage
Notation: Für den Wahrheitswert *wahr* wird häufig auch das Symbol **1** oder **T** (*True*) verwendet, für *falsch* das Symbol **0** oder **F** (*False*).

Beispiel 3.2.

1. Der Satz "*3 ist eine ungerade Zahl.*", der Satz "*Das Auto ist rot.*" und der Satz "*Die Sonne scheint.*" sind alles Aussagen, denn sie sind entweder *wahr* oder *falsch*, aber nie beides gleichzeitig.
2. Auch mathematische Gleichungen $3 + 5 = 7$ bzw. Ungleichungen $3 \leq 5$ sind logische Aussagen. Die erste ist *falsch*, die zweite *wahr*.
3. **Achtung:** Mathematische Terme wie z.B. $3 + 5$ sind jedoch **keine** Aussagen, ebenso wenig wie der Satz "*2 ist eine kleine Zahl*", es sei denn, wir hätten "*klein*" zuvor für Zahlen definiert. Auch Aufforderungen ("*Räum dein Zimmer auf!*") und Fragen ("*Schläfst du schon?*") sind keine Aussagen.
4. Der Satz "*Jede gerade natürliche Zahl > 2 ist die Summe zweier Primzahlen*" ist eine Aussage, denn entweder gibt es eine gerade Zahl die sich nicht als Summe zweier Primzahlen darstellen lässt, dann ist die Aussage *falsch*, oder es gibt sie nicht, dann ist die Aussage *wahr*. Man vermutet zwar, dass die Aussage wahr ist (*Goldbach-Vermutung*), bisher konnte das aber noch nicht bewiesen werden. Der Wahrheitswert der Aussage ist also noch ungewiss. Für die Klassifizierung des Satzes als Aussage ist die Kenntnis des Wahrheitswertes unerheblich. Lediglich die Tatsache, dass es möglich ist einen Wahrheitswert zuzuweisen, ist ausschlaggebend.
5. Der Satz "*Dieser Satz ist falsch.*" ist **keine** Aussage, da er durch den Bezug auf sich selbst weder wahr noch falsch sein kann.

Atomare oder auch elementare Aussagen , sind Aussagen, die nicht weiter zerlegt werden können. Der Satz "*Die Sonne scheint.*" und der Satz "*Die Erde ist eine Scheibe.*" sind atomare Aussagen, atomare Aussage

die zweite ist falsch, ob die erste wahr ist oder nicht lässt sich durch einen Blick aus dem Fenster klären. Aus diesen lassen sich verschiedene zusammengesetze Aussagen bilden, z.B.

1. "*Die Sonne scheint* und *die Erde ist eine Scheibe.*"
2. "*Die Sonne scheint* oder *die Erde ist eine Scheibe.*"
3. "Wenn *die Sonne scheint,* dann *ist die Erde eine Scheibe.*"
4. "Genau dann, wenn *die Sonne scheint, ist die Erde eine Scheibe.*"

Die erste ist *falsch*, da die Erde keine Scheibe ist. Die zweite ist *wahr*, wenn die Sonne scheint, die dritte ist *wahr* wenn die Sonne nicht scheint und die vierte ist *falsch*, es sei denn die Sonne würde nie scheinen.

3.1. Syntax und Semantik der Aussagenlogik

3.1.1. Syntax

Die **Syntax** der Aussagenlogik legt fest, welche Zeichenketten aussagenlogische Formeln sind und schließt damit implizit ungültige aus. Zunächst legen wir fest, welche Zeichen in Formeln der Aussagenlogik vorkommen dürfen.

Definition 3.3 (Aussagenvariablen und Alphabet)**.**

Aussagenvariable

Eine **Aussagenvariable** (kurz: Variable) bezeichnen wir durch einen Großbuchstaben A,B,. . . ,Z und ggf. einen Index $i \in \mathbb{N}$. Die Menge aller Aussagenvariablen bezeichnen wir als AVAR.

$$\text{AVAR} := \{A, B, C, \ldots, Z, A_1, \ldots\}.$$

Das **Alphabet** der Aussagenlogik ist:

$$A_{AL} := \text{AVAR} \cup \{\mathbf{0}, \mathbf{1}, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}.$$

Bemerkung 3.4.

Junktoren

Die Zeichen $\neg, \wedge, \vee, \rightarrow$ und $\leftrightarrow$ werden auch **Junktoren** genannt.

Definition 3.5 (Syntax)**.**
Die Menge AL der aussagenlogischen Formeln (kurz: Formeln) ist die folgendermaßen rekursiv definierte Menge:

Basisregeln:

$(B0)$ $\mathbf{0} \in AL$.

$(B1)$ $\mathbf{1} \in AL$.

(BV) Für jede Variable $X \in$ AVAR gilt: $X \in AL$.

Rekursive Regeln:

$(R1)$ Ist $\varphi \in AL$, dann ist auch $\neg\varphi \in AL$.

$(R2)$ Ist $\varphi \in AL$, und $\psi \in AL$, dann ist auch

$(\varphi \wedge \psi) \in AL$,

$(\varphi \vee \psi) \in AL$,

$(\varphi \rightarrow \psi) \in AL$,

$(\varphi \leftrightarrow \psi) \in AL$.

Beispiel 3.6.
Nach Definition 3.5 ist $(A \leftrightarrow (B \wedge C))$ eine gültige aussagenlogische Formel, während $(A \wedge B \wedge C)$ und $(A \leftarrow B)$ keine gültigen Formeln sind. In ersterer fehlen Klammern, in letzterer ist das Zeichen $\leftarrow$ kein gültiges Zeichen.

3.1.2. Semantik

Die **Semantik** der Aussagenlogik beschreibt die Bedeutung der Junktoren und bestimmt den Wahrheitswert der Aussage abhängig vom Wahrheitswert der Teilaussagen. Wir werden die Semantik zunächst in einer *Wahrheitstabelle* darstellen und dann auf jeden Junktor nocheinmal einzeln und mit einem Beispiel eingehen.

Semantik

Wahrheitstabelle

Wahrheitstabellen haben eine Spalte für jede Variable, sowie eine Spalte für jede Formel die betrachtet werden soll. Jede Zeile der Wahrheitstabelle repräsentiert eine mögliche Belegung der Variablen. Bei zwei Variablen hat die Wahrheitstabelle also vier Zeilen, da es vier unterschiedliche Belegungsmöglichkeiten gibt, bei 3 Variablen 8 Zeilen, bei 4 Variablen 16, bei 5 32 Zeilen, usw. Wahrheitstabellen sind also nur eingeschränkt verwendbar, aber für unsere Zwecke gerade bestens geeignet.

A	$\neg A$
0	1
1	0

A	B	$(A \wedge B)$	$(A \vee B)$	$(A \rightarrow B)$	$(A \leftrightarrow B)$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Definition 3.7 (Semantik der Aussagenlogik)**.**
Seien A und B aussagenlogische Variabeln.

$\neg$ **("nicht")** : $\neg A$ ist genau dann wahr, wenn A falsch ist.
Hierbei handelt es sich also um die *Negation* oder *Verneinung* der Aussage.
Beispiel: Die Aussage "*Das Auto ist nicht rot*" ist genau dann `wahr`, wenn das Auto eine andere Farbe als rot hat. Wenn also die Aussage "*Das Auto ist rot*" `falsch` ist.

$\wedge$ **("und")** : $A \wedge B$ ist genau dann wahr, wenn sowohl A als auch B wahr ist.
Damit die Gesamtaussage wahr ist, müssen also beide Teilaussagen wahr sein.
Beispiel: Die Aussage "*Die Sonne scheint und ich gehe schwimmen*" ist also nur wahr, wenn ich schwimmen gehe und gleichzeitig die Sonne scheint.

$\vee$ **("oder")** : $A \vee B$ ist genau dann wahr, wenn A oder B oder beide (A und B) wahr sind.
In der Alltagssprache wird "oder" häufig für die Aufzählung von Alternativen verwendet ("Tee oder Kaffee", "rot, blau oder grün", "mit oder ohne...") was dazu führt, dass es häufig zunächst verwirrend ist, dass die Aussage auch wahr ist, wenn beide Teilaussagen wahr sind. Denn im Alltag beinhaltet die Verwendung von "oder" meist die Auswahl *einer* Alternative. Implizit interpretiert der Empfänger das "oder" als ein "entweder...oder". In der Logik fordert das *"oder"*, dass *mindestens eine* der Teilaussagen erfüllt ist.
Beispiel: Die Aussage "*An der Goethe Uni darf studieren, wer ein Abitur oder ein Fachabitur oder eine Ausbildung hat*" bedeutet also, dass jeder, der *mindestens eine* der genannten Qualifikationen besitzt, an der Goethe Universität studieren darf.

→ **(“wenn..., dann...”)** : $A \to B$ ist genau dann wahr, wenn A falsch ist, oder A und B wahr sind.
Diese Verbindung zweiter Aussagen nennt man auch *Implikation*. Man sagt, *“Wenn A, dann B”*, oder *“Aus A, folgt B”*. A bezeichnet man auch als *Prämisse* und B nennt man *Konklusion*. Diese Aussage ist wie oben erwähnt nur falsch, wenn A wahr und B falsch ist. Sonst ist sie immer wahr. Insbesondere ist sie immer dann wahr, wenn A falsch ist. Warum das sinnvoll ist, zeigt folgendes Beispiel.

Prämisse
Konklusion

Beispiel 3.8.
Wir betrachten die Aussage

”Für alle natürlichen Zahlen $\neq 2$ gilt: Wenn n eine Primzahl ist, dann ist n ungerade.“

Diese Aussage ist wahr, denn wir wissen, dass alle Primzahlen $\neq 2$ ungerade sind.[1] Somit müssen insbesondere die Aussagen:

⋮
Wenn 7 eine Primzahl ist, dann ist 7 ungerade.
Wenn 8 eine Primzahl ist, dann ist 8 ungerade.
Wenn 9 eine Primzahl ist, dann ist 9 ungerade.
usw.

wahr sein. In der ersten sind Prämisse (“*...ist eine Primzahl*”) und Konklusion (“*...ist ungerade*”) wahr. In der zweiten sind Prämisse und Konklusion flasch und in der dritten ist die Prämisse falsch[2], aber die Konklusion ist wahr. Das entspricht genau den Kombinationen von Wahrheitswerten bei denen die Aussage $A \to B$ wahr ist. Lediglich wenn sich eine Primzahl $\neq 2$ finden ließe, die gerade ist, wäre die Aussage falsch. Die gibt es aber nicht, wie wir oben bereits überlegt haben. Wir können also auch sagen: “*Wenn* $n \neq 2$ *gerade ist, dann ist* n *keine Primzahl*”. Diese Aussage ist *äquivalent* zur ursprünglichen Aussage.

äquivalente Aussage

Formal bedeutet das für die Aussagen A und B:

$$A \to B \text{ ist gleichwertig mit } \neg B \to \neg A.$$

Achtung: $A \to B$ ist **nicht** gleichwertig mit $\neg A \to \neg B$. Denn nur weil 9 keine Primzahl ist, ist sie noch lange nicht gerade. Alle Primzahlen $\neq 2$ sind zwar ungerade, aber **nicht alle** ungeraden Zahlen sind auch Primzahlen!

Man sagt für Aussagen der Form $A \to B$ auch: “A ist hinreichend für B und B ist notwendig für A”. Für $n \in \mathbb{N}, n \neq 2$ ist die Bedingung dass n eine Primzahl ist *hinreichend* dafür, dass n ungerade ist. Dass n ungerade ist, ist allerdings nur eine *notwendige* Bedingung dafür, dass $n \neq 2$ eine Primzahl sein kann. Während jede Primzahl $\neq 2$ ungerade ist, ist nicht zwangsläufig jede ungerade Zahl eine Primzahl.

notwendig und hinreichend

Zum besseren Verständnis noch ein Beispiel ohne mathematische Ausdrücke:

Beispiel 3.9.
Die Aussage “*Wenn ein Tier ein Hund ist, dann ist es ein Säugetier*” besteht aus den beiden Teilaussagen A “*Das Tier ist ein Hund*” und B “*Das Tier ist ein Säugetier*”, wobei Aussage A Aussage B impliziert. Man sagt auch, dass aus der Tatsache, dass das Tier ein Hund ist, folgt, dass das Tier ein Säugetier ist. Die Bedingung, dass das Tier ein Hund ist, ist also *hinreichend* dafür, dass das Tier ein Säugetier ist, denn alle Hunde sind Säugetiere.

[1] Im Augenblick nehmen wir diese Behauptung mal so hin. Wir werden sie später beweisen (siehe Beispiel 4.6).
[2] 9 ist nicht nur durch 1 und sich selbst, sondern auch durch 3 teilbar

Allerdings sind nicht alle Säugetiere Hunde. Um ein Hund zu ein, muss das Tier ein Säugetier sein, aber noch weitere Kriterien erfüllen, z.B. ein Fell haben, bellen, etc. Ein Säugetier zu sein, ist also lediglich ein *notwendiges* Kriterium das erfüllt sein muss, damit das Tier ein Hund ist, aber *kein hinreichendes.*

$\leftrightarrow$ **("genau dann..., wenn...")** : $A \leftrightarrow B$ ist genau dann wahr, wenn $A \to B$ und $B \to A$ gilt. Das ist genau dann der Fall, wenn A und B den gleichen Wahrheitswert haben, wenn also entweder beide *wahr* oder beide *falsch* sind.
$A \leftrightarrow B$ drückt aus, dass die beiden Aussagen A und B *äquivalent* sind [3]. Wenn A gilt, dann gilt auch B und wenn A nicht gilt, dann gilt auch B nicht.
Die Aussage "*Die Klausur ist genau dann bestanden, wenn mindestens 50% der Punkte erreicht wurden*" bedeutet also, dass das Erreichen von mindesten 50% der Punkte eine notwendige und hinreichende Bedingung ist, um die Klausur zu bestehen. Alle Studierenden die in der Klausur mindestens 50% der Punkte erreichen, haben die Klausur bestanden und alle die die Klausur bestanden haben, haben mindestens 50% der Punkte erreicht.

3.2. Erfüllbarkeit, Allgemeingültigkeit und Äquivalenz

Wie zu Beginn des Kapitels bereits gesagt, geht es in der Aussagenlogik nicht um die Wahrheitsgehalte einzelner Aussagen, sondern um die Untersuchung der Wahrheitswerte komplexer Aussagenverbindungen. Vor allem interessieren dabei Aussageverbindungen die, unabhängig der Wahrheitswerte ihrer Einzelaussagen, immer wahr sind, nie wahr sind oder äquivalent sind.

Definition 3.10.
Sei φ eine aussagenlogische Formel.

φ heißt **erfüllbar**, wenn es (mindestens) eine Belegung der Variablen gibt, sodass die Formel den Wahrheitswert 1 hat. Erfüllbarkeit

φ heißt **allgemeingültig**, wenn jede Belegung der Variablen den Wahrheitswert 1 ergibt. Eine allgemeingültige aussagenlogische Formel heißt auch **Tautologie**. allgemeingültig Tautologie

φ heißt **unerfüllbar**, wenn es keine Belegung der Variablen dazu führt, dass die Formel den Wahrheitswert 1 hat. Eine unerfüllbare Formel heißt auch **Kontradiktion**. Kontradiktion

Zwei aussagenlogische Formeln φ und ψ heißen **äquivalent**, wenn für alle Belegungen der Variablen in φ und ψ die Wahrheitswerte von φ und ψ übereinstimmen. Wir schreiben auch $\varphi \equiv \psi$. äquivalent

Beispiel 3.11.

- Die Formel $(A \wedge B)$ ist *erfüllbar*, denn für die Belegung $A = 1$ und $B = 1$ ist der Wahrheitswert der Formel $(A \wedge B) = 1$. Die Formel ist *nicht allgemeingültig*, da z.B. die Belegung $A = 1$ und $B = 0$ den Wahrheitwert $(A \wedge B) = 0$ ergibt.
- Die Formel $(A \wedge \neg A)$ ist *unerfüllbar*, da keine der zur Formel passenden Belegungen zu 1 ausgewertet wird:

A	$\neg$**A**	**(A $\wedge$ $\neg$A)**
0	1	0
1	0	0

[3] in obigem Beispiel könnte man also schreiben: $(A \to B) \leftrightarrow (\neg B \to \neg A)$.

- Die Formel $(A \vee \neg A)$ ist *allgemeingültig*, da jede zur Formel passende Belegung zu 1 ausgewertet wird:

A	$\neg$**A**	(**A** $\vee$ $\neg$**A**)
0	1	1
1	0	1

- Die Formeln $\varphi := (A \wedge (A \vee B))$ und $\psi := A$ sind *äquivalent*, da ihre Wahrheitswerte für alle Belegungen übereinstimmen:

A	**B**	(**A** $\vee$ **B**)	φ	ψ
0	0	0	0	0
0	1	1	0	0
1	0	1	1	1
1	1	1	1	1

3.3. Fundamentale Rechenregeln

Abschließend stellen wir noch einige wichtige Regeln im Umgang mit aussagenlogischen Formeln vor.

Satz 3.12.
Gegeben seien aussagenlogische Formeln A, B und C.

(a). ***Doppelte Negation***

1. $\neg\neg A \equiv A$

(b). ***Kommutativgesetze***

1. $(A \wedge B) \equiv (B \wedge A)$
2. $(A \vee B) \equiv (B \vee A)$

(c). ***Assoziativgesetze***

1. $((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$
2. $((A \vee B) \vee C) \equiv (A \vee (B \vee C))$

(d). ***Distributivgesetze***

1. $((A \wedge B) \vee C) \equiv ((A \vee C) \wedge (B \vee C))$
2. $((A \vee B) \wedge C) \equiv ((A \wedge C) \vee (B \wedge C))$

Wahrheitstabelle zu 1.

A	B	C	$(A \wedge B)$	$(A \vee C)$	$(B \vee C)$	$((A \wedge B) \vee C)$	$((A \vee C) \wedge (B \vee C))$
0	0	0	0	0	0	**0**	**0**
0	0	1	0	1	1	**1**	**1**
0	1	0	0	0	1	**0**	**0**
0	1	1	0	1	1	**1**	**1**
1	0	0	0	1	0	**0**	**0**
1	0	1	0	1	1	**1**	**1**
1	1	0	1	1	1	**1**	**1**
1	1	1	1	1	1	**1**	**1**

Wahrheitstabelle zu **2.**

A	B	C	$(A \vee B)$	$(A \wedge C)$	$(B \wedge C)$	$((A \vee B) \wedge C)$	$((A \wedge C) \vee (B \wedge C))$
0	0	0	0	0	0	**0**	**0**
0	0	1	0	0	0	**0**	**0**
0	1	0	1	0	0	**0**	**0**
0	1	1	1	0	1	**1**	**1**
1	0	0	1	0	0	**0**	**0**
1	0	1	1	1	0	**1**	**1**
1	1	0	1	0	0	**0**	**0**
1	1	1	1	1	1	**1**	**1**

(e). ***De Morgan'sche Gesetze***

1. $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
2. $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$

(f). ***Absorptionsgesetze***

1. $(A \vee (A \wedge B)) \equiv A$
2. $(A \wedge (A \vee B)) \equiv A$

(g). ***Tertium non Datur***

1. $(A \wedge 1) \equiv A$
2. $(A \vee 1) \equiv 1$
3. $(A \wedge 0) \equiv 0$
4. $(A \vee 0) \equiv A$
5. $(A \wedge A) \equiv A$
6. $(A \vee A) \equiv A$

(h). ***Elimination der Implikation***

– $(A \rightarrow B) \equiv (\neg A \vee B)$

(i). ***Elimination der Biimplikation***

– $(A \leftrightarrow B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A)) \equiv ((\neg A \vee B) \wedge (\neg B \vee A))$

3.4. Quantoren

Viele mathematische Aussagen beginnen mit "*Für alle natürlichen Zahlen gilt ...*" oder "*Für alle Vierecke gilt ...*" oder "*Es gibt eine Zahl, sodass ...*".

Allgemein: Sei $A(n)$ eine Aussageform und $\mathcal{M}$ eine Menge von Werten, die n annehmen kann. Dann betrachten wir die folgenden Aussagen:

- Für alle $n \in \mathcal{M}$ gilt $A(n)$ (in Zeichen: $\forall n \in \mathcal{M} : A(n)$)
- Es gibt ein $n \in \mathcal{M}$ für das $A(n)$ gilt. (in Zeichen: $\exists n \in \mathcal{M} : A(n)$)

Die Ausdrücke "*Für alle*" und "*Es gibt*" (bzw. die Zeichen $\forall$ und $\exists$) nennt man *Quantoren*

Quantoren

Abbildung 3.1.: Ein Viereck mit 4 gleichen Winkeln.

Beispiel 3.13.

1. Betrachten wir die Aussage: "*Für alle Zahlen* $n \in \mathbb{N}$ *gilt* $n + 1 \in \mathbb{N}$". Die Menge $\mathcal{M}$ von Werten, die n in diesem Fall annehmen kann, ist $\mathbb{N}$. Das sind alle natürlichen Zahlen, also alle positiven ganzen Zahlen $\{1, 2, 3, \dots\}$. Die Aussage ist wahr, denn die Menge der natürlichen Zahlen ist unendlich groß. Es gibt also für jede natürliche Zahl eine um 1 größere Zahl, die ebenfalls in den natürlichen Zahlen ist. Man kann nicht nach oben aus den natürlichen Zahlen herauszählen, da es keine obere Grenze der natürlichen Zahlen gibt.
2. Die Aussage: "*Für alle Zahlen* $n \in \mathbb{N}$ *gilt* $n - 1 \in \mathbb{N}$" ist falsch, denn für $n = 1$ resultiert die Aussage $1 - 1 \in \mathbb{N}$ und das ist falsch, da 0 keine natürliche Zahl ist. [4]
3. Die Aussage: "*Es gibt ein Viereck das vier gleiche Winkel hat*" ist wahr. Insbesondere hat z.B. das Viereck in Abb. 3.1 vier gleiche Winkel. Tatsächlich haben alle Rechtecke vier gleiche Winkel.[5] Damit die Aussage wahr ist, reicht aber schon eins, denn von mehr ist nicht die Rede.[6] "Es gibt ein ... " bedeutet also immer "Es gibt *mindestens* ein ...".
4. Die Aussage: "*Es gibt ein Viereck dessen Winkelsumme* $400°$ *beträgt*" ist falsch, denn für alle Vierecke gilt, dass sie die Winkelsumme $360°$ haben. Demnach kann es kein Viereck mit Winkelsumme $400°$ geben.

Wir stellen fest: Eine "Für alle"-Aussage kann durch ein einziges Gegenbeispiel widerlegt werden. Eine "Es gibt"-Aussage wird durch eine "Für alle"-Aussage widerlegt. Beachte, dass bei der Argumentation für eine "Es gibt ein"-Aussage die Angabe eines einzigen Beispiels, welches die Aussage erfüllt, ausreichend ist. Bei einer "Für alle"-Aussage dagegen muss die Argumentation über alle Elemente der Menge über die die Aussage gemacht wird geführt werden. Hier **reicht es nicht** Beispiele anzuführen für die die Aussage gilt. Denn nur, weil die Aussage für alle genannten Beispiele gilt, heißt das noch lange nicht, dass sie für alle Elemente der Menge über die die Aussage gemacht ist gilt.[7]

Aussagen über leere Mengen

Achtung: In der Mathematik gilt eine Aussage der Form "*Für alle* $n \in \mathcal{M}$ *gilt ...*" immer als wahr, wenn die Menge $\mathcal{M}$ leer ist. Das ist sinnvoll, da sich in der Menge ja auch kein Gegenbeispiel befindet, mit dem die Aussage widerlegt werden könnte. Im Alltag ist das in der Regel nicht der Fall. Wenn man sagt "*Alle meine Autos sind rot*" werden die meisten Leute davon ausgehen, dass man mindestens ein Auto besitzt.

[4] Je nach Autor kann $\mathbb{N}$ die Zahl 0 enthalten oder auch nicht. Hier definieren wir $\mathbb{N}$ als $\{1, 2, 3, \dots\}$ (siehe auch Beisp 2.3), so wie es auch in der Veranstaltung Mathe 2 definiert ist. Für den Fall, dass 0 in der Menge $\mathbb{N}$ enthalten ist, wird die Aussage durch $n = 0$ widerlegt.

[5] Beachte, dass die Menge aller Rechtecke alle Quadrate mit einschließt, denn jedes Quadrat ist auch ein Rechteck.

[6] Wenn man ausdrücken möchte, dass es nur ein einziges Viereck mit dieser Eigenschaft gibt, dann sagt man "*Es gibt genau ein ...*".

[7] Es sei denn, es kann gezeigt werden, dass es in der Menge keine anderen Elemente als die im Beispiel aufgezählten gibt.

Beispiel 3.14.
Die Aussage "*Für alle eckigen Kreise gilt, dass sie gern Kaffee trinken*" ist wahr, denn da es keine eckigen Kreise gibt, kann es auch keinen eckigen Kreis geben, der nicht gerne Kaffee trinkt.

Häufig werden in Aussagen auch mehrere Quantoren kombiniert. Hier ist die Reihenfolge in der die Quantoren verwendet werden wichtig. Die Aussage "*Für jeden Topf gibt es einen Deckel, der zu dem Topf passt*" bedeutet etwas ganz anderes als die Aussage "*Es gibt einen Deckel, der zu jedem Topf passt*".

3.4.1. Negation von Aussagen

Die Negation einer Aussage ist genau dann wahr, wenn die Aussage selbst falsch ist.

Beispiel 3.15.
1. Die Aussage "Der Ball ist rund *und* ein Spiel dauert 90 Minuten" ist genau dann falsch, wenn der Ball nicht rund ist *oder* ein Spiel nicht 90 Minuten dauert.
2. Die Aussage "Heute gibt es in der Mensa Fisch *oder* Fleisch" ist genau dann falsch, wenn es heute in der Mensa keinen Fisch *und* auch kein Fleisch gibt.
3. Die Aussage "*Alle* Kinder spielen gern" ist genau dann falsch, wenn es (mindestens) ein Kind gibt, das nicht gern spielt. Die Negation der Aussage lautet also "*Es gibt (mindestens) ein* Kind das nicht gern spielt".
4. Die Aussage "Diese Woche *gibt es einen* Tag an dem es keine Nudeln gibt" ist genau dann falsch, wenn es diese Woche jeden Tag Nudeln gibt. Die Negation der Aussage lautet also "Diese Woche gibt es *an allen Tagen* Nudeln".

Wir sehen, dass die Negation "und" mit "oder" und "Für alle" mit "Es gibt" vertauscht.

Aussage	negierte Aussage	ist äquivalent zu der Aussage
$A \wedge B$	$\neg(A \wedge B)$	$\neg A \vee \neg B$
$A \vee B$	$\neg(A \vee B)$	$\neg A \wedge \neg B$
$\exists n \in \mathcal{M} : A(n)$	$\neg(\exists n \in \mathcal{M} : A(n))$	$\forall n \in \mathcal{M} : \neg A(n)$
$\forall n \in \mathcal{M} : A(n)$	$\neg(\forall n \in \mathcal{M} : A(n))$	$\exists n \in \mathcal{M} : \neg A(n)$

4. Beweistechniken

Ein Beweis ist eine **logisch vollständige Begründung** einer Aussage. Solange eine Aussage nicht bewiesen ist, kann es sein, dass sie falsch ist, auch wenn zahlreiche Beispiele die Aussage zu bestätigen scheinen.

Beispiel 4.1 (FERMAT-Zahlen)**.**
Benannt nach dem französischen Mathematiker *Pierre de Fermat*, brechnen sich die Zahlen aus $F_n = 2^{2^n} + 1$.

$$\begin{aligned} F_0 &= 2^{2^0} + 1 &&= 2^1 + 1 = 2 + 1 &&= 3 \\ F_1 &= 2^{2^1} + 1 &&= 2^2 + 1 = 4 + 1 &&= 5 \\ F_2 &= 2^{2^2} + 1 &&= 2^4 + 1 = 16 + 1 &&= 17 \end{aligned}$$

FERMAT vermutete 1637, dass alle F_n Primzahlen sind. Erst im Jahr 1732 konnte EULER beweisen, dass diese Vermutung nicht stimmt. Er zeigte, dass $F_5 = 4\,294\,967\,297$ durch 641 teilbar und somit keine Primzahl ist.

Jeder Beweis ist aus einzelnen, leicht nachvollziehbaren Schritten aufgebaut.

4.1. Direkter Beweis

Um die Aussage $A \to B$ zu beweisen, beginnt man mit der Prämisse A und argumentiert dann unter Verwendung von Definitionen und bereits bewiesenen Aussagen schrittweise, bis man bei der Konklusion B angelangt ist. Um $A \to B$ zu beweisen, argumentieren wir also $A \to A_1$ und $A_1 \to A_2$ usw. bis wir bei $A_i \to B$ angelangt sind.

Beispiel 4.2.
Wir wollen nun die Korrektheit des folgenden Satz mit einem direkten Beweis zeigen.

Satz 4.3.
Die Summe zweier gerader Zahlen ist wiederum eine gerade Zahl

Bevor wir mit dem Beweis loslegen können, müssen wir erstmal klären, wie eine gerade Zahl definiert ist.

Definition 4.4.
Eine Zahl ist genau dann gerade, wenn sie durch 2 teilbar ist.

Jetzt müssen wir noch klarstellen, was wir mit "*teilbar sein*" meinen.

Definition 4.5.
Eine Zahl $a \in \mathbb{Z}$ ist genau dann durch eine andere Zahl $b \neq 0$ teilbar, wenn es eine Zahl $k \in \mathbb{Z}$ gibt, sodass $a = b \cdot k$.[1](Beispiel: Für $a = 24, b = 6$ und $k = 4$ gilt: $24 = 6 \cdot 4$. 24 ist also durch 6 teilbar.)

[1] $\mathbb{Z}$ bezeichnet die Menge der ganzen Zahlen, also $\mathbb{Z} = \{0, -1, 1, -2, 2, -3, 3, \dots\}$.

Umgangssprachlich sagen wir also, dass eine Zahl a gerade ist, wenn sie ohne Rest durch 2 teilbar ist.

Nachdem wir also geklärt haben, was eine gerade Zahl ist, schauen wir nach Prämisse A und Konklusion B. Damit die zwei Teilaussagen der Aussage deutlicher werden, formulieren wir die Aussage um, **ohne** den Sinn der Aussage zu verändern.

$$\underbrace{\text{Wenn zwei gerade Zahlen addiert werden}}_{A}, \underbrace{\text{dann ist das Ergebnis wieder eine gerade Zahl}}_{B}$$

$$A \qquad \rightarrow \qquad B$$

Kommen wir jetzt zum Beweis.

Beweis

Zu zeigen ist also, dass bei der Addition zweier gerader Zahlen, wieder eine gerade Zahl entsteht; und zwar bei der Addition egal welcher beider geraden Zahlen und nicht nur bei der Addition zweier bestimmter gerader Zahlen. Es handelt sich hier also auch um eine "Für alle" Aussage, nicht um eine "Es gibt" Aussage. Es ist also nicht damit getan zwei gerade Zahlen anzugeben, deren Ergebniss wieder eine gerade Zahl ist. Wir werden für alle geraden Zahlen argumentieren müssen.

Seien also a und b zwei beliebige gerade Zahlen. Laut Definition 4.4 sind a und b durch 2 teilbar. Das bedeutet demnach laut Definition 4.5, dass $a = 2 \cdot k$ und $b = 2 \cdot l$ für zwei Zahlen $k, l \in \mathbb{Z}$. Wenn wir also jetzt a und b addieren, dann können wir schreiben:

$$\begin{aligned} a + b &= 2 \cdot k + 2 \cdot l \\ &= 2 \cdot (k + l) \\ &= 2 \cdot m && \text{für } m = k + l \end{aligned}$$

Wenn wir nun argumentieren können, dass $m \in \mathbb{Z}$ ist, haben wir's geschafft, denn dann ist $a + b$ nach Definition 4.5 durch zwei teilbar und nach Definition 4.4 somit gerade.

m ist eine ganze Zahl, da die ganzen Zahlen unter $+$ abgeschlossen sind. Das bedeutet, dass bei der Addition zweier ganzer Zahlen das Ergebnis auch immer eine ganze Zahl ist. Strenggenommen müssten wir diese Abgeschlossenheit jetzt noch beweisen, das ginge aber hier zu weit. Daher nehmen wir es als bereits bewiesene Aussage an.

Zusammenfassung
Ausgehend von der Prämisse, dass a und b gerade Zahlen sind, haben wir unter Verwendung der Definitionen für gerade Zahlen und Teilbarkeit, sowie die als bewiesen angenommene Aussage, dass die Summer zweier ganzer Zahlen wieder eine ganze Zahl ist, schrittweise argumentiert, dass dann die Summe $a + b$ ebenfalls eine gerade Zahl ist.

Betrachten wir anhand einer oben bereits verwendeten Aussage (siehe Beispiel 3.8), einen weiteren direkten Beweis.

Satz 4.6.
Alle Primzahlen bis auf die 2 sind ungerade.

Umformuliert ergibt sich der Satz:

$$\underbrace{\text{Wenn } x \neq 2 \text{ eine Primzahl ist}}_{\text{A}}, \underbrace{\text{dann ist } x \text{ ungerade}}_{B}.$$
$$\text{A} \quad \rightarrow \quad B$$

Beweis
Zu zeigen ist, dass für jede Primzahl $\neq 2$ gilt, dass sie ungerade ist.
Sei x also eine Primzahl. Laut Definition der Primzahlen, ist x somit durch 1 und sich selbst teilbar und sonst durch keine andere Zahl. Insbesondere ist x dann auch nicht durch 2 teilbar, denn das würde nur für $x = 2$ gelten und das ist ja ausgeschlossen. Laut Definition der Teilbarkeit (Def 4.5) ist x also **nicht** durch $2 \cdot k, k \in \mathbb{Z}$ darstellbar und somit laut Definition der geraden Zahlen (Def 4.4) **nicht** gerade. Damit muss x also ungerade sein. □

4.1.1. Abgeschlossenheit einer Zahlenmenge bezüglich einer Verknüpfung

Zahlen kann man auf verschiedene Weise miteinander "verknüpfen". Z. B. kann man sie addieren, subtrahieren, multiplizieren und dividieren. Als Verallgemeinerung schreiben wir hier für eine solche Verknüpfung das Zeichen $\circ$ [2].

Definition 4.7 (Abgeschlossenheit)**.**
Eine Zahlenmenge heißt *abgeschlossen* bezüglich einer Verknüpfung, wenn für alle Zahlen a, b aus der Menge gilt, dass das Ergebnis der Verknüpfung $a \circ b$ auch wieder in der Zahlenmenge ist.

Im Vorkurs dürfen Sie folgende Abgeschlossenheiten als bereits bewiesen annehmen:

- Die natürlichen Zahlen $\mathbb{N}$ sind abgeschlossen bezüglich der Verknüpfungen $+$ und $\cdot$.
- Die ganzen Zahlen $\mathbb{Z}$ sind abgeschlossen bezüglich der Verknüpfungen $+$, - und $\cdot$.

Beispiel 4.8.
Den direkten Beweis können wir z.B. auch nutzen, um die in Satz 2.13 gemachte Behauptung der Distributivität von Schnitt und Vereinigung von Mengen zu zeigen. Zu zeigen ist: Für die Mengen A, B und C gilt:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

Nach Definition 2.6 müssen wir für die Gleichheit der beiden Mengen $A \cup (B \cap C)$ und $(A \cup B) \cap (A \cup C)$ zeigen, dass die erste in der zweiten und die zweite in der ersten Menge enhalten ist.

Wir beginnen damit zu zeigen, dass $A \cup (B \cap C) \subset (A \cup B) \cap (A \cup C)$ ist. Nach Definition 2.4 müssen wir hierfür zeigen, dass jedes Element von $A \cup (B \cap C)$ auch in $(A \cup B) \cap (A \cup C)$ enthalten ist.

Sei $x \in A \cup (B \cap C)$ beliebig gewählt. Für x gilt dann:

$$\begin{array}{rcll} x \in A & \cup & (B \cap C) & \\ \leftrightarrow x \in A & \vee & x \in (B \cap C) & \text{(Def. Vereinigung von Mengen)} \\ \leftrightarrow x \in A & \vee & (x \in B \ \wedge \ x \in C) & \text{(Def. Schnitt von Mengen)} \end{array}$$

[2] Beispiel: $5 \circ 2$ kann also für $5 + 2, 5 - 2, 5 \cdot 2$ oder $5 : 2$ stehen.

Wir müssen also die zwei Fälle $x \in A$ und $x \notin A$ betrachten:

Fall 1: $x \in A$. Aus Definition 2.10 zu Vereinigung und Schnitt von Mengen folgt, dass dann auch $x \in (A \cup B)$ und $x \in (A \cup C)$ ist und somit $x \in (A \cup B) \cap (A \cup C)$.
Fall 2: $x \notin A$. Dann muss nach $x \in A \vee (x \in B \wedge x \in C)$ also $x \in B$ und $x \in C$ sein. Damit ist $x \in (A \cup B) \cap (A \cup C)$.

Somit gilt in beiden Fällen $x \in (A \cup B) \cap (A \cup C)$. Da x beliebig gewählt war, gilt also allgemein für $x \in A \cup (B \cap C)$, dass $x \in (A \cup B) \cap (A \cup C)$ und somit $A \cup (B \cap C) \subset (A \cup B) \cap (A \cup C)$.

Nun müssen wir noch zeigen, dass $(A \cup B) \cap (A \cup C) \subset A \cup (B \cap C)$ ist.

Sei $x \in (A \cup B) \cap (A \cup C)$ beliebig gewählt. Für x gilt dann:

$$\begin{array}{rcll} x \in (A \cup B) & \cap & (A \cup C) & \\ \leftrightarrow x \in (A \cup B) & \wedge & x \in (A \cup C) & \text{(Def. Schnitt von Mengen)} \\ \leftrightarrow (x \in A \vee x \in B) & \wedge & (x \in A \vee x \in C) & \text{(Def. Vereinigung von Mengen)} \end{array}$$

Es gibt wieder zwei Fälle, nämlich $x \in A$ und $x \notin A$ zu betrachten:

Fall 1: $x \in A$. In diesem Fall ist auch $x \in A \cup (B \cap C)$.
Fall 2: $x \notin A$. Dann muss aber $x \in B$ und $x \in C$ sein. Somit ist $x \in B \cap C$ und ebenfalls $x \in A \cup (B \cap C)$.

Da x beliebig gewählt war, gilt somit allgemein für $x \in (A \cup B) \cap (A \cup C)$, dass $x \in A \cup (B \cap C)$ ist. Damit gilt nach der Definition von Teilmengen: $(A \cup B) \cap (A \cup C) \subset A \cup (B \cap C)$ □

4.2. Indirekter Beweis / Beweis durch Kontraposition

Manchmal ist es schwierig von der Prämisse auf die Konklusion zu schließen. Wie wir oben gesehen haben, ist die Aussage $A \rightarrow B$ aber äquivalent zu der Aussage $\neg B \rightarrow \neg A$. Wir können also auch versuchen zu zeigen, dass wenn die Konklusion (Aussage B) nicht gilt, dann auch die Prämisse (Aussage A) nicht gelten kann und damit die Aussage $\neg B \rightarrow \neg A$ zeigen. Da die beiden Aussagen äquivalent sind, hätten wir damit indirekt gezeigt, dass auch $A \rightarrow B$ gilt.

A	B	$A \rightarrow B$	$\neg A$	$\neg B$	$\neg B \rightarrow \neg A$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	0	0	1

Beispiel 4.9.

Satz 4.10.
Wenn a^2 gerade ist, dann ist auch a gerade.

Wenn wir diese Aussage direkt beweisen wollen, wird es schwierig, denn wie sollen wir aus $a^2 = 2 \cdot k, k \in \mathbb{Z}$ argumentieren, dass daraus folgt, dass $a = \sqrt{2 \cdot k}, k \in \mathbb{Z}$ ebenfalls gerade ist?

Also versuchen wir statt $A \rightarrow B$, $\neg B \rightarrow \neg A$ zu beweisen.

	Aussage	negierte Aussage
A	a^2 ist eine gerade Zahl	a^2 ist eine ungerade Zahl
B	a ist eine gerade Zahl	a ist eine ungerade Zahl

Statt “*Wenn* a^2 *eine gerade Zahl ist, dann ist auch a eine gerade Zahl.*” beweisen wir also “*Wenn a eine ungerade Zahl ist, dann ist auch* a^2 *eine ungerade Zahl.*”.

Beweis
Sei a also eine beliebige ungerade Zahl. In Anlehnung an die Definition gerader Zahlen (Def 4.4) lässt sich a dann als $a = 2 \cdot k + 1$ mit $k \in \mathbb{Z}$ darstellen. Wenn wir a nun quadrieren, passiert folgendes:

$$\begin{aligned}
a^2 &= (2 \cdot k + 1)^2 && \text{(quadrieren)} \\
&= (2 \cdot k)^2 + 2 \cdot (2 \cdot k \cdot 1) + 1^2 && \text{(Binomische Formel mit } a = 2 \cdot k \text{ und } b = 1) \\
&= \mathbf{2} \cdot 2 \cdot k^2 + \mathbf{2} \cdot (2 \cdot k \cdot 1) + 1^2 \\
&= \mathbf{2} \cdot (2 \cdot k^2 + 2 \cdot k) + 1^2 && \text{(ausklammern)} \\
&= 2 \cdot l + 1 && \text{mit } l = 2 \cdot k^2 + 2 \cdot k
\end{aligned}$$

Bleibt also zu argumentieren, dass $l \in \mathbb{Z}$ ist. Das ist aber der Fall, denn 2 und k sind ganze Zahlen, also $2, k \in \mathbb{Z}$ und die ganzen Zahlen sind abgeschlossen unter $+$ und $\cdot$ (siehe oben). Somit ist $l = 2 \cdot k^2 + 2 \cdot k$ ebenfalls eine ganze Zahl und $a^2 = 2 \cdot l + 1$ ungerade. □

Damit haben wir indirekt unsere ursprüngliche Aussage “*Wenn* a^2 *gerade ist, dann ist a ebenfalls gerade*” bewiesen.

4.3. Beweis durch Widerspruch

Das Vorgehen beim Beweis durch Widerspruch ist ähnlich wie beim indirekten Beweis. Statt $A \rightarrow B$ zeigen wir, dass $A \wedge \neg B$ zu einem Widerspruch führt und somit falsch ist. Das heißt wir zeigen indirekt, dass $\neg(A \wedge \neg B)$ wahr ist. Diese Aussage ist äquivalent zur Aussage $A \rightarrow B$ (siehe Wahrheitstabelle) und somit haben wir indirekt auch $A \rightarrow B$ gezeigt.

A	B	$A \rightarrow B$	$\neg A \vee B$	$A \wedge \neg B$	$\neg(A \wedge \neg B)$
0	0	1	1	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	0	1

Manchmal gibt es auch keine Prämisse. Man möchte einfach nur eine Aussage C beweisen. Das Vorgehen ist in diesem Fall, anzunehmen, dass die Aussage nicht gilt (also $\neg C$ anzunehmen) und zu zeigen, dass das zu einem Widerspruch führt. Damit hat man gezeigt, dass $\neg C \rightarrow \mathbf{F}$ wahr ist und somit $\neg C$ falsch ist. Im Umkehrschluss muss C aber dann wahr sein, denn wenn $\neg C$ falsch ist, muss $\neg(\neg C)$ wahr sein. $\neg(\neg C)$ ist aber nichts anderes als die ursprüngliche Aussage C.

Satz 4.11.
$\sqrt{2}$ *ist eine irrationale Zahl.*

In diesem Beispiel gibt es keine Prämisse. Wir wollen einen Beweis durch Widerspruch führen und nehmen an, dass die negierte Aussage gilt. Dann hoffen wir durch schrittweises Argumentieren auf einen Widerspruch zu unserer Annahme zu stoßen und damit zu zeigen, dass die negierte Aussage falsch ist. Somit muss die ursprüngliche Aussage wahr sein.

Beweis

Nehmen wir also an, $\sqrt{2}$ sei rational. D. h. $\sqrt{2} \in \mathbb{Q}$.[3] Also können wir $\sqrt{2} = \frac{p}{q}$ schreiben, mit $p, q \in \mathbb{Z}$ und p, q teilerfremd, also $\operatorname{ggT}(p, q) = 1$. Wenn wir jetzt quadrieren, erhalten wir $2 = \frac{p^2}{q^2}$ und durch Umformung $2 \cdot q^2 = p^2$. Da $q^2 \in \mathbb{Z}$, ist p^2 laut Definition der Teilbarkeit (Def 4.5) durch 2 teilbar. Wie soeben bewiesen, ist dann aber auch p durch 2 teilbar und als $p = 2 \cdot k$ mit $k \in \mathbb{Z}$ darstellbar. Also lässt sich die Gleichung $2 \cdot q^2 = p^2$ auch als $2 \cdot q^2 = (2 \cdot k)^2 = 2 \cdot 2 \cdot k^2$ schreiben. Wenn man jetzt auf beiden Seiten durch 2 teilt, erhält man $q^2 = 2 \cdot k^2$. Damit ist q^2 durch 2 teilbar, denn $k^2 \in \mathbb{Z}$ und somit auch q durch 2 teilbar. Das heißt sowohl p als auch q sind durch 2 teilbar. Das allerdings ist eine Widerspruch zu der Annahme, dass p und q teilerfremd sind, denn sie haben 2 als gemeinsamen Teiler. Somit ist die Aussage *"$\sqrt{2}$ ist eine rationale Zahl"* widerlegt und entsprechend ist die Aussage *"$\sqrt{2}$ ist eine irrationale Zahl"* bewiesen. □

[3] Rationale Zahlen sind dadurch gekennzeichnet, dass sie als Bruch zweier ganzer Zahlen darstellbar sind

5. Induktion und Rekursion

Strenggenommen sollte dieses Kapitel ein Unterkapitel des vorherigen Kapitels sein, denn die *vollständige Induktion* ist eine mathematische Beweistechnik. Allerdings ist diese Technik komplexer als die bisher behandelten Beweistechniken daher widmen wir ihr hier ein eigenes Kapitel.

5.1. Vollständige Induktion

Ziel der vollständigen Induktion ist es zu beweisen, dass eine Aussage $A(n)$ für alle $n \in \mathbb{N}_0$[1] gilt. Dabei verwendet man das **Induktionsprinzip**, d.h. man schließt vom Besonderen auf das Allgemeine. (Im Gegensatz zur *Deduktion*, wo man vom Allgemeinen auf das Besondere schließt.) Das Vorgehen ist folgendermaßen:

Induktionsprinzip

1. Für eine gegebene Aussage A zeigt man zunächst, dass die Aussage für ein (meist $n = 0$, oder $n = 1$) oder einige kleine n wahr ist. Diesen Schritt nennt man **Induktionsanfang**. (Häufig findet sich in der Literatur auch *Induktionsbasis* oder *Induktionsverankerung*.)

Induktionsanfang

2. Dann zeigt man im **Induktionsschritt**, dass für jede beliebige Zahl $n \in \mathbb{N}$ gilt: Falls die Aussage $A(n)$ wahr ist, so ist auch die Aussage $A(n + 1)$ wahr. (**Induktionsbehauptung**)

Induktionsschritt
Induktionsbehauptung

Wenn man also gezeigt hat, dass $A(n + 1)$ aus $A(n)$ folgt, dann gilt insbesondere $A(1)$, falls $A(0)$ wahr ist. Damit gilt dann aber auch $A(2)$, da $A(1)$ gilt, $A(3)$ da $A(2)$ gilt, usw. .

Für das erste Beispiel zur vollständigen Induktion werden abkürzende Schreibweisen für Summen und Produkte eingeführt.

Definition 5.1.
Sei $n \in \mathbb{N}$, und seien $a_1, \dots, a_n$ beliebige Zahlen. Dann ist:

- $\sum_{i=1}^{n} a_i := a_1 + a_2 + \cdots + a_n$
  insbesondere ist die leere Summe $\sum_{i=1}^{0} a_i = 0$.

Summe

- $\prod_{i=1}^{n} a_i := a_1 \cdot a_2 \cdot \dots \cdot a_n$
  insbesondere ist das leere Produkt $\prod_{i=1}^{0} a_i = 1$.

Produkt

Beispiel 5.2.

Satz 5.3 (kleiner Gauß)**.**
$A(n)$:Für alle $n \in \mathbb{N}$ gilt:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Induktionsanfang: $n = 0$
Behauptung: $A(0)$: Der Satz gilt für $n = 0$.
Beweis:

$$\sum_{i=1}^{0} i = 0 = \frac{0}{2} = \frac{0(0+1)}{2} = \frac{n(n+1)}{2}$$

[1]Wir schreiben $\mathbb{N}_0$ für die natürlichen Zahlen inklusive der 0 ($\mathbb{N} \cup \{0\}$)

Induktionsschritt: $A(n) \to A(n+1)$
Induktionsvoraussetzung: Es gilt $A(n)$, also $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
Unter dieser Voraussetzung muss nun gezeigt werden, dass der Satz auch für $n+1$ gilt.
Induktionsbehauptung: Es gilt $A(n+1)$:

$$\sum_{i=1}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$$

Beweis:

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= 1 + 2 + \cdots + n + (n+1) \\
&= (\sum_{i=1}^{n} i) + (n+1) && \text{Induktionsvoraussetzung anwenden} \\
&= \frac{n(n+1)}{2} + (n+1) && \text{mit 2 erweitern} \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2} \\
&= \frac{(n+1)((n+1)+1)}{2}
\end{aligned}$$

□

Im Folgenden wird der besseren Lesbarkeit wegen, statt $A(n) \to A(n+1)$ lediglich $n \to n+1$ geschrieben und vorausgesetzt, dass dem Leser klar ist, dass im Fall $n = 0$ die Aussage $A(0)$ bzw. im Fall $n = 1$ die Aussage $A(1)$ gemeint ist.

Beispiel 5.4.

Satz 5.5.
Für alle $n \in \mathbb{N}$ und $q \neq 1$ gilt:

$$\sum_{i=0}^{n} q^i = \frac{1 - q^{n+1}}{1 - q}$$

Induktionsanfang: $n = 0$
Behauptung: Es gilt: $\sum_{i=0}^{0} q^i = \frac{1-q^{0+1}}{1-q}$
Beweis:

$$\sum_{i=0}^{0} q^i = q^0 = 1 = \frac{1-q}{1-q} = \frac{1-q^1}{1-q} = \frac{1-q^{0+1}}{1-q}$$

Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Es gilt $\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}$.
Induktionsbehauptung: Es gilt:

$$\sum_{i=0}^{n+1} q^i = \frac{1 - q^{(n+1)+1}}{1 - q}$$

Beweis:

$$\begin{aligned}
\sum_{i=0}^{n+1} q^i &= q^1 + q^2 + \cdots + q^n + q^{n+1} \\
&= \sum_{i=0}^{n} q^i + q^{n+1} && \text{Induktionsvoraussetzung} \\
&= \frac{1-q^{n+1}}{1-q} + q^{n+1} && \text{erweitern mit } (1-q) \\
&= \frac{1-q^{n+1}}{1-q} + \frac{q^{n+1}\cdot(1-q)}{1-q} \\
&= \frac{1-q^{n+1}+q^{n+1}-q^{n+1}\cdot q}{1-q} \\
&= \frac{1-q\cdot q^{n+1}}{1-q} \\
&= \frac{1-q^{(n+1)+1}}{1-q}
\end{aligned}$$

□

Man kann mit der Beweistechnik der vollständigen Induktion jedoch nicht nur Gleichungen beweisen.

Beispiel 5.6.

Satz 5.7.
Sei $n \in \mathbb{N}$. $n^2 + n$ ist eine gerade (d.h. durch 2 teilbare) Zahl.

Beweis. durch vollständige Induktion.

Induktionsanfang: $n = 0$
Behauptung: $0^2 + 0$ ist eine gerade Zahl.
Beweis: $0^2 + 0 = 0 + 0 = 0$ ist eine gerade Zahl. Das stimmt, denn $0 = 2 \cdot 0, 0 \in \mathbb{Z}$ und damit nach Def 4.5 und Def. 4.4 gerade.
Da die 0 aber so speziell ist, kann man zur Sicherheit und zur Übung den Satz auch noch für $n = 1$ beweisen. Notwendig, ist das allerdings nicht.
Behauptung: $1^2 + 1$ ist eine gerade Zahl.
Beweis: $1^2 + 1 = 1 + 1 = 2$. $2 = 2 \cdot 1, 1 \in \mathbb{Z}$. Damit ist 2 eine gerade Zahl.

Induktionsschritt: $n \to n + 1$
Induktionsvoraussetzung: Für $n \geq 0$ gilt: $n^2 + n$ ist eine gerade Zahl.
Induktionsbehauptung: $(n + 1)^2 + (n + 1)$ ist eine gerade Zahl.
Beweis:

$$\begin{aligned}
(n+1)^2 + (n+1) &= n^2 + 2n + 1 + n + 1 \\
&= n^2 + 3n + 2 \\
&= (n^2 + n) + (2n + 2) \\
&= (n^2 + n) + 2 \cdot (n + 2)
\end{aligned}$$

$(n^2 + n) + 2 \cdot (n + 2)$ ist eine gerade Zahl, da laut Induktionsvoraussetzung $n^2 + n$ eine gerade Zahl ist, und $2 \cdot (n + 1)$ ist ein Vielfaches von 2. Somit ist auch der zweite Summand eine gerade Zahl, und die Summe gerader Summanden ist ebenfalls gerade. Das haben wir in Beispiel 4.2 bewiesen.

□

5.1.1. Wann kann man vollständige Induktion anwenden?

Die vollständige Induktion eignet sich, um Behauptungen zu beweisen, die sich auf Objekte (Zahlen, Geraden, Spielzüge,...) beziehen, die als natürliche Zahlen betrachtet werden können. Mathematisch korrekt ausgedrückt, muss die Objektmenge die sog. *Peano-Axiome* erfüllen. Diese sagen im wesentlichen, dass es ein erstes Element geben muss, jedes Element einen eindeutig bestimmten Nachfolger haben muss und das Axiom der vollständigen Induktion gelten muss.

Aussagen über reelle Zahlen lassen sich beispielsweise nicht mit vollständiger Induktion beweisen.

Oftmals ist man versucht zu meinen, dass Induktion immer dann möglich ist, wenn die Behauptung ein n enthält. Allerdings, ist folgender Satz nicht mit vollständiger Induktion zu beweisen.

Satz 5.8.
Sei $n \in \mathbb{N}$. Dann ist die Folge $a(n) = 1/n$ immer positiv.

Obiger Satz ist zwar wahr, aber wie soll man aus $\frac{1}{n} > 0$ folgern, dass $\frac{1}{n+1} > 0$? Genau das wäre für den Induktionsschritt aber notwendig. Im Induktionsschritt muss gezeigt werden, dass aus $A(n)$, $A(n+1)$ folgt.

5.1.2. Was kann schief gehen?

Das Prinzip der vollständigen Induktion lässt sich auch mit einem Domino-Effekt vergleichen. Die Bahn läuft durch, d.h. alle Dominosteine fallen um, wenn man den ersten Stein umstoßen kann und gesichert ist, dass jeder Stein n seinen Nachfolger $n+1$ umstößt.

In obigem Beispiel lässt sich nicht beweisen, dass jeder Stein n seinen Nachfolger $n+1$ umstößt. Daher ist die vollständige Induktion als Beweismethode für Satz 5.8 ungeeignet.

Der Beweis, dass der erste Stein umgestoßen werden kann, $A(n)$ gilt für ein *bestimmtes* n, der *Induktionsanfang*, ist genau so wichtig, wie der *Induktionsschritt*.

Beispiel 5.9 (fehlender Induktionsanfang)**.**
Im folgenden sei Teilbarkeit wie in Definition 4.5 definiert.
Zum Beispiel lässt sich aus der Aussage $A(5 \textit{ ist durch } 2 \textit{ teilbar})$ logisch korrekt folgern, dass auch $B(7 \textit{ ist durch } 2 \textit{ teilbar})$ gilt. Die Schlussfolgerung ist logisch korrekt, die Aussagen gelten aber nicht, da eben die Voraussetzung nicht gegeben ist. Denn 5 ist nunmal nicht durch 2 teilbar.

Während der Induktionsanfang meist relativ einfach zu beweisen ist, macht der Induktionsschritt häufiger Probleme. Die Schwierigkeit liegt darin, dass ein konstruktives Argument gefunden werden muss, das in Bezug auf die Aufgabenstellung tatsächlich etwas aussagt. Dies ist der Fehler im folgenden Beispiel.

Beispiel 5.10 (fehlerhafte Induktion)**.**
Behauptung: In einen Koffer passen unendlich viele Socken.
Induktionsanfang: $n = 1$
Behauptung: Ein Paar Socken passt in einen leeren Koffer.
Beweis: Koffer auf, Socken rein, Koffer zu. Passt.
Induktionsschritt: $n \to n+1$**:**
Induktionsvoraussetzung: n Paar Socken passen in den Koffer.
Induktionsbehauptung: $n+1$ Paar Sochen passen in den Koffer.
Beweis: n Paar Socken befinden sich im Koffer. Aus Erfahrung weiß man, ein Paar Socken passt immer noch rein. Also sind nun $n+1$ Paar Socken im Koffer. □
Somit ist bewiesen, dass unendlich viele Socken in einen Koffer passen.

Was ist passiert?
Das Argument „aus Erfahrung weiß man, ein Paar Socken passt immer noch rein“, ist in Bezug

auf die Aufgabenstellung nicht konstruktiv. Ein konstruktives Argument hätte sagen müssen, wo genau das extra Paar Socken noch hinpasst.

Ferner muss man darauf achten, dass das n der Aussage $A(n)$ aus der man dann $A(n+1)$ folgert keine Eigenschaften hat, die im Induktionsanfang nicht bewiesen wurden.

Beispiel 5.11 (fehlerhafte Induktion).
Behauptung: Alle Menschen einer Menge M mit $|M| = n$ sind gleich groß.
Induktionsanfang: $n = 1$
Behauptung: In einer Menge von einem Menschen, sind alle Menschen dieser Menge gleich groß.
Beweis: Sei M eine Menge von Menschen mit $|M| = 1$. Da sich genau ein Mensch in M befindet, sind offensichtlich alle Menschen in M gleich groß.
Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Sei $n \in \mathbb{N}$ beliebig. In einer Menge Menschen M', mit $|M'| = n$, haben alle Menschen die gleiche Größe.
Induktionsbehauptung: Ist M eine Menge Menschen mit $|M| = n+1$, so sind alle Menschen in M gleich groß.
Beweis: Sei $M = \{m_1, m_2, \ldots, m_{n+1}\}$ eine Menge von $n+1$ Menschen. Sei $M' = \{m_1, m_2, \ldots, m_n\}$ und $M'' = \{m_2, m_3, \ldots, m_{n+1}\}$. Damit sind M' und M'' Mengen von je n Menschen. Laut Induktionsannahme gilt dann:

1. Alle Menschen in M' haben die gleiche Größe g'.
2. Alle Menschen in M'' haben die gleiche Größe g''.

Insbesondere hat Mensch $m_2 \in M'$ Größe g' und Mensch $m_2 \in M''$ Größe g''. Da aber jeder Mensch nur eine Größe haben kann, muss gelten: $g' = g''$. Wegen $M = M' \cup M''$, haben somit alle Menschen in M die gleiche Größe $g = g' = g''$. □

Was ist passiert?
Der Induktionsschluss funktioniert nur für $n > 1$. Denn nur, wenn es mindestens 2 Menschen mit der gleichen Größe gibt, kann ich m_1, m_2 in M' und m_2, m_{n+1} in M'' einteilen. Im Fall $n = 1$, und $n+1 = 2$, gilt $M' = \{m_1\}$ und $M'' = \{m_2\}$. Dann ist $m_2 \in M''$, jedoch keinesfalls in M'. Die Argumentation im Induktionsschritt fällt in sich zusammen, denn es gibt keinen Grund, warum m_1 und m_2 die gleiche Größe haben sollten. Man hätte also im Induktionsanfang beweisen müssen, dass die Aussage auch für $n = 2$ gilt. Wie leicht einzusehen ist, wird das nicht gelingen, denn zwei willkürlich herausgegriffene Menschen sind keineswegs zwangsläufig gleich groß.

5.2. Rekursion

Rekursion ist eine Technik bei der Funktionen durch sich selbst definiert werden. Bei der rekursiven Definition wird das Induktionsprinzip angewendet. Zunächst wird, meist für kleine Eingaben, der Funktionswert explizit angegeben (*Rekursionsanfang*). Dann wird im *Rekursionsschritt* eine Vorschrift formuliert, wie die Funktionswerte für größere Eingaben mit Hilfe der Funktionswerte kleinerer Eingaben berechnet werden können.

Rekursionsanfang

Rekursionsschritt

Ein bekannter Spruch zur Rekursion lautet:

„Wer Rekursion verstehen will, muss Rekursion verstehen “

Definition 5.12 (Fakultätsfunktion).
Die Funktion $f : \mathbb{N} \to \mathbb{N}$, gegeben durch:

Fakultätsfunktion

$$f(n) := \begin{cases} 1, & \text{falls } n = 0 \qquad \text{Rekursionsanfang} \\ n \cdot f(n-1), & \text{sonst.} \qquad \text{Rekursionsschritt} \end{cases}$$

heißt **Fakultätsfunktion**. Man schreibt für $f(n)$ auch $n!$.

Was genau beschreibt nun diese rekursive Definition? Einen besseren Überblick bekommt man meist, wenn man ein paar konkrete Werte für n einsetzt.

$$\begin{aligned}
&f(0) = 1 && n = 0\\
&f(1) = 1 \cdot f(0) = 1 \cdot 1 = 1 && n = 1\\
&f(2) = 2 \cdot f(1) = 2 \cdot (1 \cdot f(0)) = 2 \cdot (1 \cdot 1) = 2 && n = 2\\
&f(3) = 3 \cdot f(2) = 3 \cdot (2 \cdot f(1)) = 3 \cdot (2 \cdot (1 \cdot f(0))) = 3 \cdot (2 \cdot (1 \cdot 1)) = 6 && n = 3
\end{aligned}$$

Es liegt nahe, dass die Fakultätsfunktion das Produkt der ersten n natürlichen Zahlen beschreibt.

Satz 5.13.
Für die Fakultätsfunktion $f(n)$ gilt: $f(n) = \prod_{i=1}^{n} i$.

Beweis. Hier eignet sich vollständige Induktion zum Beweis des Satzes.

Induktionsanfang: $n = 0$
Behauptung: Der Satz gilt für $n = 0$.
Beweis: Nach Definition 5.12 gilt $f(0) = 1$. Nach Definition 5.1 gilt $\prod_{i=1}^{0} i = 1$. Im Fall von $n = 0$ ist somit $f(0) = 1 = \prod_{i=1}^{0} i = \prod_{i=1}^{n} i$

Induktionsschritt: $n \to n + 1$**:**
Induktionsvoraussetzung: Es gilt $f(n) = \prod_{i=1}^{n} i$ für n. Unter dieser Voraussetzung zeigt man nun, dass
Induktionsbehauptung: $f(n + 1) = \prod_{i=1}^{n+1} i$ gilt.
Beweis:

$$\begin{aligned}
f(n+1) &= (n+1) \cdot f(n) && \text{Definition 5.12}\\
&= (n+1) \cdot \prod_{i=1}^{n} i && \text{Induktionsvoraussetzung}\\
&= (n+1) \cdot n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1\\
&= \prod_{i=1}^{n+1} i
\end{aligned}$$

□

Nicht nur Funktionen lassen sich rekursiv definieren, auch Mengen können rekursiv definiert werden.

Beispiel 5.14.
So lässt sich die Menge der natürlichen geraden Zahlen $\mathbb{N}_G = 0, 2, 4, \dots$ folgendermaßen rekursiv definieren.

Basisregel: $0 \in \mathbb{N}_G$

Rekursive Regel: Ist $x \in \mathbb{N}_G$, so ist auch $x + 2 \in \mathbb{N}_G$.

Die implizite Definition der Menge $\mathbb{N}_G$ lautet: $\mathbb{N}_G = \{x | x = 2n, n \in \mathbb{N}\}$.

5.2.1. Wozu Rekursion?

Schaut man sich die obigen Beispiele an, so kann man sich berechtigter Weise fragen, wozu Rekursion gut sein soll. Betrachten wir ein anderes Beispiel.

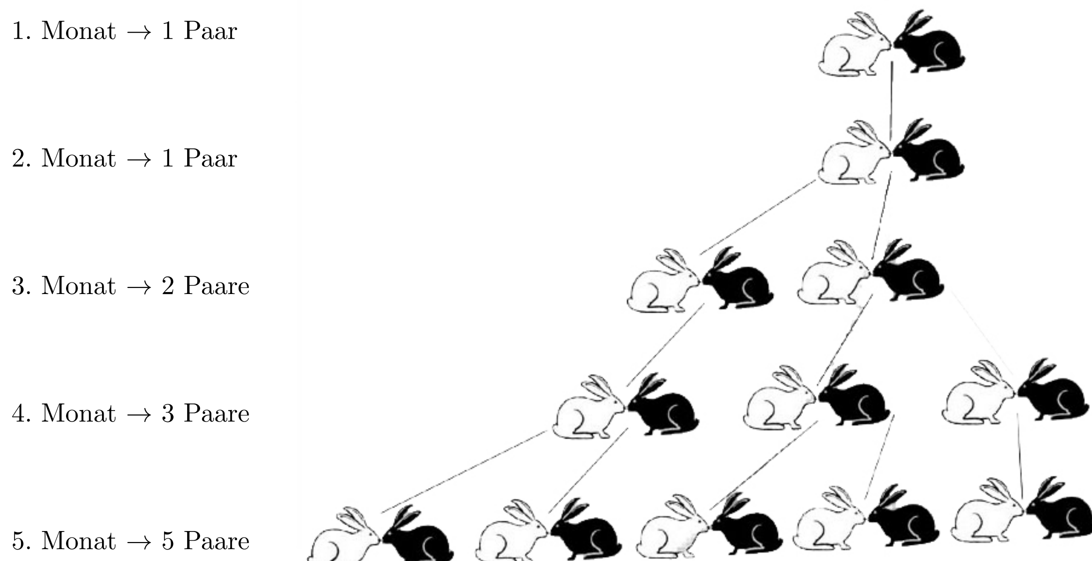


Tabelle 5.1.: Vermehrung der idealen Kaninchenpaare

Beispiel 5.15.
Ein Freund hat uns ein frischgeborenes Kaninchenpaar (♂,♀) geschenkt. Netterweise hat er uns vorgewarnt, dass das Paar in einem Monat geschlechtsreif wird und dann jeden Monat ein neues Kaninchenpaar werfen wird. Mit einem besorgten Blick auf unseren teuren, begrenzten Frankfurter Wohnraum und den schmalen Geldbeutel fragen wir uns:
„Wie viele Kaninchenpaare werden wir in einem Jahr haben, wenn sich die Kaninchen ungehindert vermehren können und keines stirbt? “

Im ersten Monat wächst unser erstes Kaninchenpaar (♂,♀) heran. Somit haben wir zum Ende des 1. Monats immernoch 1 Kaninchenpaar. Nun gebärt dieses Kaninchenpaar am Ende des 2. Monats ein weiteres Kaninchenpaar, also haben wir zu Beginn des 3. Monats bereits 2 Kaninchenpaare. Am Ende des 3. Monats gebärt unser ursprüngliches Kaninchenpaar dann erneut ein Kaninchenpaar, das zweite Kaninchenpaar aber wächst ersteinmal heran. Daher haben wir zu Beginn des 4. Monats insgesamt 3 Kaninchenpaare. Unser ursprüngliches, das Paar welches am Ende des 2. Monats geboren wurde und das Paar das gerade erst geboren wurde. Am Ende des 4. Monats gebären dann sowohl unser urspüngliches Kaninchenpaar, als auch unser Kaninchenpaar das am Ende des 2. Monats geboren wurde je ein Kaninchenpaar. Zu Beginn des 5. Monats sind wir also schon stolze Besitzer von 5 Kaninchenpaaren. Von denen werden alle Paare trächtig, die älter als 2 Monate sind. Das sind 3, also haben wir zu Beginn des 6. Monats 8 Kaninchenpaare, usw... Wir ahnen schon Böses, was in den ersten 2 Monaten so harmlos anfing, wächst uns über den Kopf.

Können wir eine Funktion $fib : \mathbb{N} \to \mathbb{N}$ angeben, die uns sagt, wie viele Kaninchenpaare wir zu Beginn des n-ten Monats haben werden?
Zu Beginn des 1. Monats haben wir 1 Paar, zu Beginn des 2. Monats haben wir immernoch nur ein Paar, zu Beginn des n-ten Monats haben wir immer so viele Kaninchenpaare, wie frisch geboren wurden, zusätzlich zu denen, die wir zu Beginn des vorigen Monats bereits hatten. Da alle Paare, die mindestens 2 Monate alt sind, also alle Paare, die wir vor 2 Monaten bereits hatten, ein Kaninchenpaar gebären, ist die Anzahl der neugeborenen Paare gleich der Anzahl der Paare vor 2 Monaten. Somit ergibt sich:

Definition 5.16 (Fibonacci-Folge).

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

Fibonacci Folge

Zu diesem Schluss kam 1202 bereits der italienische Mathematiker *Leonardo Fibonacci*, als er über eben dieses Problem nachdachte. Nach ihm wurde die durch $fib(n)$ definierte Zahlenfolge benannt.

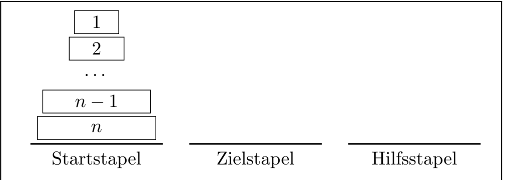
Beispiel 5.17.
Betrachten wir ein weiteres Beispiel:

Türme von Hanoi

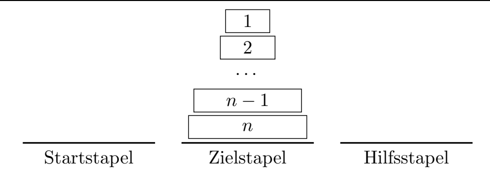
Türme von Hanoi

Bei dem Spiel *Türme von Hanoi*, muss ein Stapel von n, nach Größe sortierter Scheiben, von einem *Startstapel* mithilfe eines *Hilfsstapels* auf einen *Zielstapel* transportiert werden (siehe Abbildung). Dabei darf

- immer nur eine Scheibe bewegt werden und
- es darf nie eine größere auf einer kleineren Scheibe liegen.

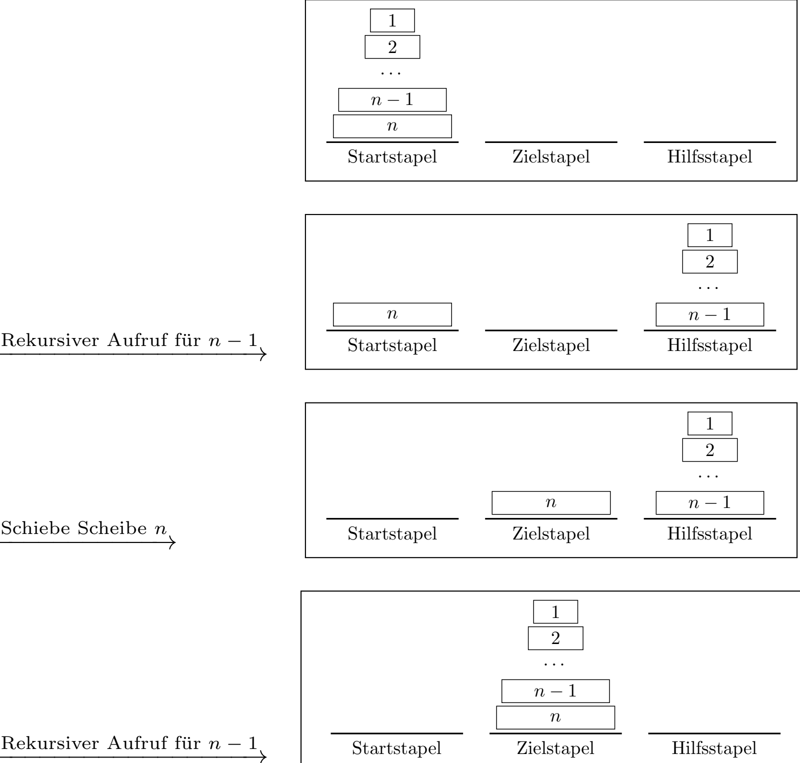


Ausgangssituation



Zielsituation

Für $n = 1$ Scheibe ist die Lösung des Problems noch trivial, auch für $n = 2$ Scheiben ist die Lösung offensichtlich. Für $n = 3$ Scheiben muss man schon etwas „rumprobieren“, um das Problem zu lösen. Für noch mehr Scheiben benötigen wir eine Lösungsstrategie. Bei der Betrachtung des Problems fällt auf, dass wir, um n Scheiben vom Startstapel auf den Zielstapel zu transportieren, zunächst die oberen $n-1$ Scheiben vom Startstapel auf den Hilfsstapel transportieren müssen. Dann brauchen wir lediglich die n-te Scheibe vom Startstapel auf den Zielstapel zu legen und dann die $n-1$ Scheiben vom Hilfsstapel auf den Zielstapel (siehe Abbildung).



Um $n-1$ Scheiben vom Start- zum Hilfsstapel zu bewegen, müssen wir die oberen $n-2$ Scheiben vom Start- zum Zielstapel bewegen. Um das zu tun, müssen wir $n-3$ Scheiben vom Start- auf den Hilfsstapel bewegen, usw. Wir verkleinern also sukzessive die Problemgröße und wählen Start-, Ziel- und Hilfsstapel passend, bis wir lediglich eine Scheibe vom Start- zum Zielstapel transportieren müssen. Das Problem können wir sofort lösen.

So haben wir mithilfe von Rekursion rasch eine Lösung für das Problem gefunden.

Algorithmus 5.18.

```
1  bewege n Scheiben von Start zu Ziel, benutze Hilf
2    falls n>1
3      bewege n−1 Scheiben von Start zu Hilf, benutze Ziel
4    verschiebe Scheibe n von Start auf Ziel
5    falls n>1
6      bewege n−1 Scheiben von Hilf zu Ziel, benutze Start
```

Würden wir nun unser Leben darauf verwetten, dass das immer funktioniert? Vielleicht beweisen wir vorher lieber die Korrektheit.

Korrektheit

Satz 5.19.
Algorithmus 5.18 löst das Problem der „Türme von Hanoi".

Beweis durch vollständige Induktion
Sei $n \in \mathbb{N}$ und sei die Aufgabe einen Stapel mit n Scheiben von Stapel A (start) nach B (ziel) zu transportieren, wobei Stapel C (hilf) als Hilfsstapel verwendet werden darf.

Induktionsanfang: $n = 1$
Behauptung: Der Algorithmus arbeitet korrekt für $n = 1$.
Beweis: Da $n = 1$ ist, führt der Aufruf von `bewege 1 Scheibe von A nach B, benutze C` dazu, dass lediglich Zeile 5 ausgeführt wird. Die Scheibe wird also von *start* auf *ziel* verschoben. Genau das sollte geschehen.
Zur Sicherheit und Übung betrachten wir auch noch $n = 2$
Behauptung: Der Algorithmus arbeitet korrekt für $n = 2$.
Beweis:

```
1  bewege 2 Scheiben von A nach B, benutze C        //Aufruf mit n
2    bewege 1 Scheibe von A nach C, benutze B       //Z.3, 1.Aufruf mit (n−1)
3      verschiebe Scheibe 1 von A nach C            //Aufruf mit (n−1), Z.5
4    verschiebe Scheibe 2 von A nach B              //Aufruf mit n, Z.5
5    bewege 1 Scheibe von C nach B, benutze A       //Aufruf mit n, Z.7
6      verschiebe Scheibe 1 von C nach B            //2.Aufruf mit (n−1), Z.5
```

Die oberste Scheibe wird also von A auf Stapel C gelegt (3), dann wird die untere Scheibe von Stapel A auf B gelegt (4) und zum Schluss die kleinere Scheibe von Stapel C auf B gelegt (6). Somit ist der Stapel mit $n = 2$ Scheiben unter Beachtung der Regeln von A nach B verschoben worden.

Induktionsschritt: $n \to n + 1$
Induktionsvoraussetzung: Der Algorithmus arbeitet korrekt für $n \geq 1$.
Induktionsbehauptung: Wenn der Algorithmus für n korrekt arbeitet, dann auch für $n + 1$.
Beweis:

```
1  bewege n+1 Scheiben von A nach B, benutze C      //Aufruf mit n+1
2    bewege n Scheiben von A nach C, benutze B      //Z.3, Aufruf mit n
3    verschiebe Scheibe n+1 von A auf B             //Z.5
4    bewege n Scheiben von C nach B, benutze A      //Z.7, Aufruf mit n
```

Zuerst werden also die obersten n Scheiben von Stapel A nach Stapel C transportiert (2). Laut Induktionsvoraussetzung arbeitet der Algorithmus für n Scheiben korrekt und transportiert den

Stapel mit n Scheiben von A nach C. Dann wird Scheibe $n+1$ von A nach B verschoben (3), anschließend werden die n Scheiben auf Stapel C auf Stapel B transportiert (4). Das verstößt nicht gegen die Regeln, da

1. die Scheibe $n+1$ größer ist als alle anderen Scheiben, denn sie war zu Beginn die unterste Scheibe. Somit kann die Regel, dass niemals ein größere auf einer kleineren Scheibe liegen darf, nicht verletzt werden, da B frei ist und Scheibe $n+1$ somit zuunterst liegt.
2. der Algorithmus für n Scheiben korrekt arbeitet und somit der Stapel mit n Scheiben korrekt von C nach B verschoben wird.

Damit arbeitet der Algorithmus auch für $n+1$ korrekt. □

Hier ist im Induktionsschritt gleich zweimal die Induktionsvoraussetzung angewendet worden. Einmal, um zu argumentieren, dass die ersten n Scheiben korrekt von A nach C transportiert werden, und dann, um zu argumentieren, dass diese n Scheiben auch korrekt von C nach B transportiert werden können.

Anzahl der Spielzüge

Es drängt sich einem schnell der Verdacht auf, dass das Spiel „Die Türme von Hanoi“ ziemlich schnell ziemlich viele Versetzungen einer Scheibe (oder Spielzüge) benötigt, um einen Stapel zu versetzen. Um zu schauen, ob sich eine Gesetzmäßigkeit feststellen lässt, zählt man zunächst die Spielzüge für kleine Werte von n.

$n=1$	Schiebe 1 von A nach B	1 Zug
$n=2$	$1\curvearrowright$ C, $2\curvearrowright$ B, $1\curvearrowright$ B	3 Züge
$n=3$	$1\curvearrowright$ B, $2\curvearrowright$ C ,$1\curvearrowright$ C, $3\curvearrowright$ B, $1\curvearrowright$ A, $2\curvearrowright$ B, $1\curvearrowright$ B	7 Züge

Nach einigem Nachdenken kommt man auf die Gesetzmäßigkeit:

Satz 5.20.
Um n Scheiben von einem Stapel zu einem anderen zu transportieren, werden mindestens 2^n-1 Spielzüge benötigt.

Beweis durch vollständige Induktion
Induktionsanfang: $n=1$
Behauptung: Um eine Scheibe von einem Stapel auf einen anderen zu transportieren, wird mindesten $2^1-1=2-1=1$ Spielzug benötigt.
Beweis: Setze die Scheibe vom Startstapel auf den Zielstapel. Das entspricht einem Spielzug und man ist fertig.

Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Um n Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens 2^n-1 Spielzüge benötigt.
Induktionsbehauptung: Um $n+1$ Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens $2^{n+1}-1$ Spielzüge benötigt.
Beweis: Um $n+1$ Scheiben von einem Stapel A auf einen Stapel B zu transportieren, transportiert man nach Algorithmus 5.18 zunächt n Scheiben von Stapel A auf Stapel C, dann Scheibe $n+1$ von Stapel A nach Stapel B und zum Schluss die n Scheiben von Stapel C nach Stapel B. Nach der Induktionsvoraussetzung benötigt das Versetzen von n Scheiben von Stapel A auf Stapel C mindestens 2^n-1 Spielzüge, das Versetzen der Scheibe $(n+1)$, 1 Spielzug und das Versetzen der n Scheiben von C nach B nochmals mindestens 2^n-1 Spielzüge (Induktionsvoraussetzung). Das sind insgesamt mindestens:

$$2^n - 1 + \mathbf{1} + 2^n - 1 = 2 \cdot (2^n - 1) + 1 = 2 \cdot 2^n - 2 + 1 = 2^{n+1} - 1$$

Spielzüge. □

6. Einführung in die Bedienung von Unix-Systemen

In diesem Kapitel werden wir vorwiegend die Grundlagen der Bedienung von Unix-Systemen und dabei insbesondere die Benutzung der für Informatikstudierende zur Verfügung stehenden Linux-Rechner an der Goethe-Universität erläutern.

6.1. Unix und Linux

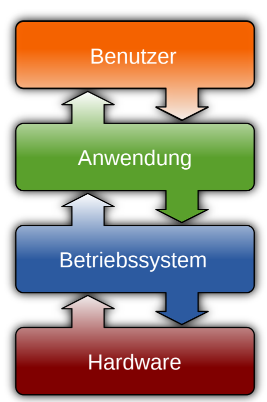


Abbildung 6.1.:

Unix ist ein Betriebssystem und wurde 1969 in den *Bell Laboratories* (später *AT&T*) entwickelt. Als Betriebssytem verwaltet Unix den Arbeitsspeicher, die Festplatten, CPU, Ein- und Ausgabegeräte eines Computers und bildet somit die Schnittstelle zwischen den Hardwarekomponenten und der Anwendungssoftware (z.B. Office) des Benutzers (Abb. 6.1). Da seit den 80er Jahren der Quellcode von Unix nicht mehr frei verfügbar ist und bei der Verwendung von Unix hohe Lizenzgebühren anfallen, wurde 1983 das GNU-Projekt (**G**NU's **N**ot **U**nix) ins Leben gerufen, mit dem Ziel, ein freies Unix-kompatibles Betriebssystem zu schaffen. Dies gelang 1991 mithilfe des von Linus Torvalds programmierten Kernstücks des Betriebssystems, dem Linux-Kernel. GNU Linux ist ein vollwertiges, sehr mächtiges Betriebssystem.

Unix

GNU Linux

Da der Sourcecode frei zugänglich ist, kann jeder seine eigenen Anwendung und Erweiterungen programmieren und diese veröffentlichen. Es gibt vielzählige Linux-Distributionen (Red Hat, SuSe, Ubuntu,...), welche unterschiedliche Software Pakete zur Verfügung stellen. Auf den Rechnern der **R**echner**b**etriebsgruppe **I**nformatik der Goethe Universität (RBI) ist Red Hat Linux installiert.

Linux stellt seinen Benutzern sog. *Terminals* zur Verfügung, an denen gearbeitet werden kann. Ein Terminal ist eine Schnittstelle zwischen Mensch und Computer. Es gibt *textbasierte* und *graphische* Terminals.

Terminal

Textbasierte Terminals stellen dem Benutzer eine Kommandozeile zur Verfügung. Über diese kann der Benutzer, durch Eingabe von Befehlen, mithilfe der Computertastatur, mit Programmen, die über ein CLI (**c**ommand **l**ine **i**nterface) verfügen, interagieren. Einige solcher Programme werden wir gleich kennenlernen. Das Terminal stellt Nachrichten und Informationen der Programme in Textform auf dem Bildschirm dar. Der Nachteil textbasierter Terminals ist für Anfänger meist, dass die Kommandozeile auf eine Eingabe wartet. Man muss den Befehl kennen, den man benutzen möchte, oder wissen, wie man ihn nachschauen kann. Es gibt nicht die Möglichkeit sich mal irgendwo „durchzuklicken“. Der Vorteil textbasierter Terminals ist, dass die Programme mit denen man arbeiten kann häufig sehr mächtig sind. Ein textbasiertes Terminal bietet sehr viel mehr Möglichkeiten, als ein graphisches Terminal.

textbasiertes Terminal
CLI

Graphische Terminals sind das, was die meisten Menschen, die heutzutage Computer benutzen, kennen. Ein graphisches Terminal lässt sich mit einer Computermaus bedienen. Der Benuter bedient die Programme durch Klicken auf bestimmte Teile des Bildschirms, welche durch Icons

graphisches Terminal

GUI

(kleine Bilder) oder Schriftzüge gekennzeichnet sind. Damit das funktionert, benötigen die Programme eine graphische Benutzeroberfläche, auch GUI (**g**raphical **u**ser **i**nterface) genannt. Auf den Rechnern der RBI findet man, unter anderem, die graphischen Benutzeroberflächen Gnome und KDE.

Ein einzelner Rechner stellt sieben, voneinander unabhängige Terminals zur Verfügung. Mit den Tastenkombinationen Strg + Alt + F1 , Strg + Alt + F2 bis Strg + Alt + F7 kann zwischen den sieben Terminals ausgewählt werden. Tastatureigaben werden immer an das angezeigte Terminal weitergeleitet. In der Regel ist lediglich das durch die Tastenkombination Strg + Alt + F7 erreichbare Terminal graphisch. Alle andern Terminals sind textbasiert. Auf den RBI-Rechnern ist das graphische Terminal als das aktive Terminal eingestellt, sodass ein Benutzer der nicht Strg + Alt + F1 , . . . , Strg + Alt + F6 drückt, die textbasierten Terminals nicht zu Gesicht bekommt.

6.1.1. Dateien und Verzeichnisse

Eines der grundlegenden UNIX-Paradigmen ist: „Everything is a file“. Die Zugriffsmethoden für Dateien, Verzeichnisse, Festplatten, Drucker, etc. folgen alle den gleichen Regeln, grundlegende Kommandos sind universell nutzbar. Über die Angabe des Pfades im UNIX-Dateisystem lassen sich Quellen unabhängig von ihrer Art direkt adressieren. So erreicht man beispielsweise mit `/home/hans/protokoll.pdf` eine persönliche Datei des Benutzers „hans“, ein Verzeichnis auf einem Netzwerklaufwerk mit `/usr`, eine Festplattenpartition mit `/dev/sda1/` und sogar die Maus mit `/dev/mouse`.

Dateibaum
Verzeichnis

Das UNIX-Betriebssystem verwaltet einen *Dateibaum*. Dabei handelt es sich um ein virtuelles Gebilde zur Datenverwaltung. Im Dateibaum gibt es bestimmte Dateien, welche *Verzeichnisse* (engl.: *directories*) genannt werden. Verzeichnisse können andere Dateien (und somit auch Verzeichnisse) enthalten. Jede Datei muss einen Namen haben, dabei wird zwischen Groß- und Kleinschreibung unterschieden. `/home/hans/wichtiges` ist ein anderes Verzeichnis als `/home/hans/Wichtiges`. Jede Datei, insbesondere jedes Verzeichnis, befindet sich in einem Verzeichnis, dem *übergeordneten* Verzeichnis (engl.: *parent directory*). Nur das *Wurzelverzeichnis* (engl.:*root directory*) ist in sich selbst enthalten. Es trägt den Namen „/“.

übergeordnetes Verzeichnis
Wurzelverzeichnis

Beispiel 6.1 (Ein Dateibaum)**.**
Nehmen wir an, das Wurzelverzeichnis enthält zwei Verzeichnisse `Alles` und `Besser`. Beide Verzeichnisse enthalten je ein Verzeichnis mit Namen `Dies` und `Das`. In Abbildung 6.2 lässt sich der Baum erkennen. Die Bäume mit denen man es meist in der Informatik zu tun hat, stehen auf dem Kopf. Die Wurzel befindet sich oben, die Blätter unten.

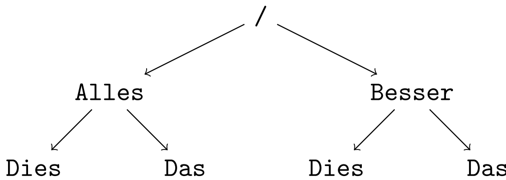


Abbildung 6.2.: Ein Dateibaum

Pfad
absolut
relativ

Die beiden Verzeichnisse mit Namen `Dies` lassen sich anhand ihrer Position im Dateibaum leicht auseinanderhalten. Den Weg durch den Dateibaum zu dieser Position nennt man *Pfad* (engl::*path*). Gibt man den Weg von der Wurzel aus an, so spricht man vom *absoluten* Pfad. Gibt man den Weg vom Verzeichnis aus an, in dem man sich gerade befindet, so spricht man vom *relativen* Pfad. Die absoluten Pfade zu den Verzeichnissen mit Namen `Dies` lauten `/Alles/Dies` und `/Besser/Dies`.

slash

Unter UNIX/LINUX dient der Schrägstrich `/` (engl.:*slash*) als Trennzeichen zwischen Verzeich-

nissen. Im Gegensatz zu Windows, wo dies durch den back-slash \ geschieht. Wenn wir uns im Verzeichnis `/Besser` befinden, so können die Unterverzeichnisse mit *Dies* und `Das` direkt adressiert werden. Das Symbol `..` bringt uns ins übergeordnete Verzeichnis, in diesem Fall das Wurzelverzeichnis. Somit erreichen wir aus dem das Verzeichnis `Alles` aus dem Verzeichnis `Besser` über den relativen Pfad `../Alles`. Befinden wir uns in Verzeichnis `/Alles/Dies` so erreichen wir das Verzeichnis `/Besser/Das` über den relativen Pfad `../../Besser/Das`.

Dateizugriffsrechte

Unter UNIX können auch die Zugriffsrechte einzelner Benutzer auf bestimmte Dateien verwaltet werden. Dabei wird unterschieden zwischen Lese-(*r*ead), Schreib-(*w*rite) und Ausführrechten (*x* execute). Für die Vergabe dieser Rechte, wird zwischen dem Besitzer (*o*wner) der Datei, einer Gruppe (*g*roup) von Benutzern und allen Nutzern, die nicht zu der Gruppe gehören (*o*ther), unterschieden. Um bestimmten Nutzern Zugriffsrechte für bestimmte Dateien zu erteilen, können diese Nutzer zu einer Gruppe zusammengefasst werden. Dann können allen Mitgliedern der Gruppe Zugriffsrechte für diese Dateien erteilt werden.

6.1.2. Login und Shell

Um an einem Terminal arbeiten zu können, muss sich der Benutzer zunächst anmelden. Dies geschieht durch Eingabe eines Benutzernamens und des zugehörigen Passwortes. Diesen Vorgang nennt man „sich *Einloggen*“. Loggt man sich an einem textbasierten Terminal ein, so startet nach dem Einloggen automatisch eine (Unix)-*Shell*. Dies ist die traditionelle Benutzerschnittstelle unter UNIX/Linux. Der Benutzer kann nun über die Kommandozeile Befehle eintippen, welche der Computer sogleich ausführt. Wenn die Shell bereit ist Kommandos entgegenzunehmen, erscheint eine *Eingabeaufforderung* (engl.: *prompt*). Das ist eine Zeile, die Statusinformationen, wie den Benutzernamen und den Namen des Rechners auf dem man eingeloggt ist, enthält und mit einem blinkenden *Cursor* (Unterstrich) endet.

Einloggen
Shell
Eingabe-
aufforderung

Benutzer, die sich an einem graphischen Terminal einloggen, müssen zunächst ein virtuelles textbasiertes Terminal starten, um eine Shell zu Gesicht zu bekommen. Ein virtuelles textbasiertes Terminal kann man in der Regel über das Startmenü, Unterpunkt „Konsole“ oder „Terminal“, gestartet werden. Unter der graphischen Benutzeroberfläche KDE kann man solch ein Terminal auch starten, indem man mit der rechten Maustaste auf den Desktop klickt und im erscheinenden Menü den Eintrag „Konsole“ auswählt (Abb.: A.2).

6.1.3. Befehle

Es gibt unzählige Kommandos die die Shell ausführen kann. Wir beschränken uns hier auf einige, die wir für besonders nützlich halten. Um die Shell aufzuforden den eingetippten Befehl auszuführen, muss die *Return*-Taste (↩) betätigt werden. Im Folgenden sind Bildschirmein- und -ausgaben in `Schreibmaschinenschrift` gegeben und

```
in grau hinterlegten Kästen mit durchgezogenen Linien und runden Ecken zu finden.
```

Später werden wir auch sogenannte Quelltexte darstellen, diese sind

```
in gestrichelten und eckigen Kästen zu finden.
```

Passwort ändern

passwd: ändert das Benutzerpasswort auf dem Rechner auf dem man eingeloggt ist. Nach Eingabe des Befehls, muss zunächt einmal das alte Passwort eingegeben werden. Dannach muss zweimal das neue Passwort eingegeben werden. Dieser Befehl ist ausreichend, wenn der Account lediglich auf einem Rechner existiert, z.B. wenn man Linux auf seinem privaten Desktop oder Laptop installiert hat.

```
> passwd ↩
Changing password for [Benutzername].
(current) UNIX password: ↩
Enter new UNIX password: ↩
Retype new UNIX password: ↩
passwd: password updated successfully
```

Für den RBI-Account wird folgender Befehl benötigt.

Netzwerk-passwort

yppasswd: ändert das Passwort im System und steht dann auf allen Rechnern des Netzwerks zur Verfügung.

```
> passwd ↩
Changing NIS account information for [Benutzername] on [server].
Please enter old password: ↩
Changing NIS password for [Benutzername] on [server].
Please enter new password: ↩
Please retype new password: ↩

The NIS password has been changed on [server].
```

Arbeits-verzeichnis
Home-verzeichnis

pwd (*print working directory*): gibt den Pfad des Verzeichnisses aus, in dem man sich gerade befindet. Dieses Verzeichnis wird häufig auch als *„aktuelles Verzeichnis“*, oder *„Arbeitsverzeichnis“* bezeichnet. Unmittelbar nach dem Login, befindet man sich immer im *Homeverzeichnis*. Der Name des Homeverzeichnis ist der gleiche wie der Benutzername. Hier werden alle persönlichen Dateien und Unterverzeichnisse gespeichert.

```
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
```

whoami : gibt den Benutzernamen aus.

```
> whoami ↩
ronja
```

hostname: gibt den Rechnernamen, auf dem man eingeloggt ist, aus.

```
> hostname ↩
nash
```

Verzeichnis erstellen
Argument

mkdir: (*make directory*): mit diesem Befehl wird ein neues Verzeichnis (Ordner) angelegt. Dieser Befehl benötigt als zusätzliche Information den Namen, den das neue Verzeichnis haben soll. Dieser Name wird dem Befehl als *Argument* übergeben. Zwischen Befehl und Argument muss immer ein Leerzeichen stehen. Der folgende Befehl legt in dem Verzeichnis, in dem sich der Benutzer gerade befindet, ein Verzeichnis mit Namen „Zeug“ an.

```
> mkdir Zeug ↩
```

`cd` (*change directory*): wechselt das Verzeichnis. Wird kein Verzeichnis explizit angegeben, so wechselt man automatisch in das Homeverszeichnis.

`cd ..`: wechselt in das nächsthöhere Verzeichnis. Dabei wird `..` als Argument übergeben.

```
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
> mkdir Wichtiges ↩
> cd Wichtiges ↩
> pwd ↩
 /home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/Wichtiges/
> cd .. ↩
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
```

`ls` (*list*): zeigt eine Liste der Namen der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Dateien die mit einem „.“ anfangen, meist Systemdateien, werden nicht angezeigt.

`ls -a` : zeigt eine Liste der Namen *aller* (engl.: *all*) Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden an. Auch Dateien die mit einem „.“ anfangen, werden angezeigt. Bei dem `-a` handelt es sich um eine *Option* , die dem Befehl übergeben wird. Optionen werden mit einem oder zwei Bindestrichen eingeleitet. Dabei können mehrer Optionen gemeinsam übergeben werden, ohne dass erneut ein Bindestrich eingegeben werden muss. Wird dem Kommando als Argument der absolute oder relative Pfad zu einem Verzeichnis angegeben, so werden die Namen der in diesem Verzeichnis enthaltenen Dateien angezeigt.

Option

```
> ls ↩
Wichtiges  sichtbareDatei1.txt  sichtbareDatei2.pdf
> ls -a ↩
.  ..  Wichtiges  sichtbareDatei1.txt  sichtbareDatei2.pdf
> ls -a Wichtiges ↩
.  ..
```

`ls -l`: zeigt eine Liste der Namen und Zusatzinformationen (`l` für engl.: *long*) der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Die Einträge ähneln dem Folgenden.

```
> ls -l ↩
-rw-r--r-- 1    alice    users   2358  Jul 15   14:23 protokoll.pdf
```

Von rechts nach links gelesen, sagt uns diese Zeile, dass die Datei „protokoll.pdf“ um 14:23 Uhr am 15. Juli diesen Jahres erstellt wurde. Die Datei ist 2358 Byte groß, und gehört der Gruppe „users“, insbesondere gehört sie der Benutzerin „alice“ und es handelt sich um eine (1) Datei. Dann kommen 10 Positionen an denen die Zeichen `-`, `r` oder `w`, stehen. Der Strich (`-`) an der linkesten Position zeigt an, dass es sich hierbei um eine gewöhnliche Datei handelt. Bei einem Verzeichnis würde an dieser Stelle ein `d` (für directory) stehen. Dann folgen die Zugriffsrechte. Die ersten drei Positionen sind für die Zugriffsrechte der Besitzers (engl.: *owner*) der Datei. In diesem Fall darf die Besitzerin alice die Datei lesen (*r*ead) und verändern (*w*rite). Alice darf die Datei aber nicht ausführen (*x* execute). Eine gewöhnliche .pdf-Datei möchte man aber auch nicht ausführen. Die Ausführungsrechte sind wichtig für Verzeichnisse und Programme. Die mittleren drei Positionen geben die Zugriffsrechte der Gruppe (engl.: *group*) an. Die Gruppe users darf hier die Datei lesen, aber nicht schreiben. Die letzen drei Positionen sind für die Zugriffrechte aller andern Personen (engl.: *other*). Auch diesen ist gestattet die Datei zu lesen, sie dürfen sie aber nicht verändern.

Zugriffsrechte

`chmod` (*change mode*): ändert die Zugriffsberechtigungen auf eine Datei. Dabei muss dem Programm die Datei, deren Zugriffsrechte man ändern will, als Argument übergeben werden. Ferner muss man dem Programm mitteilen, wessen (*u*ser,*g*roup,*o*ther, oder *a*ll) Rechte man wie ändern (`+` hinzufügen, `-` wegnehmen) will.

```
> chmod go +w protokoll.pdf ↵
```

Dieser Befehl gibt der Gruppe `g` und allen anderen Nutzern `o` Schreibrechte `+w` für die Datei `protokoll.pdf`. Vermutlich ist es keine gute Idee, der ganzen Welt die Erlaubnis zu erteilen die Datei zu ändern.

```
> chmod o -rw protokoll.pdf ↵
```

nimmt allen anderen Nutzern `o` die Schreibrechte wieder weg `-w` und nimmt ihnen auch die Leserechte `-r`.

Alternativ kann die gewünschte Änderung auch als Oktalzahl eingegeben werden. Für die Erkärung dazu verweisen wir auf das Internet, oder die *man-page*, s.u.

`man` (*manual*): zeigt die Hilfe-Seiten zu dem, als Argument übergebenen, Kommando an.

```
ronja@nash: ~
MKDIR(1)                    User Commands                    MKDIR(1)

NAME
       mkdir - make directories

SYNOPSIS
       mkdir [OPTION]... DIRECTORY...

DESCRIPTION
       Create the DIRECTORY(ies), if they do not already exist.
 Manual page mkdir(1) line 1 (press h for help or q to quit)
```

Abbildung 6.3.: man page des Befehls `mkdir`

6.1.4. History und Autovervollständigung

History

Ein sehr nützliches Hilfmittel beim Arbeiten mit der Shell ist die *history*. Alle Befehle, die man in der Shell eingibt, werden in der history gespeichert. Mit den Cursortasten ↑ und ↓ kann man in der history navigieren. ↑ holt den zuletzt eingegebenen Befehl in die Eingabezeile, ein erneutes Drüchen von ↑ den vorletzen, usw. ↓ arbeitet in die andere Richtung, also z.B. vom vorletzen Befehl zum zuletzt eingegebenen Befehl. Mit den Cursortasten ← und →, kann man innerhalb des Befehls navigieren, um Änderungen vorzunehmen.

Autovervollständigung

Ein weiteres nützliches Hilfmittel ist die *Autovervollständigung*. Hat man den Anfang eines Befehls, oder eines Datei- (oder Verzeichnis-) Namens eingegeben, so kann man den Namen durch Betätigen der *Tab*-Taste ⇆ automatisch vervollständigen lassen, solange der angegebene Anfang eindeutig ist. Ist dies nicht der Fall, so kann man sich mit nochmaliges Betätigen der Tab-Taste ⇆, eine Liste aller in Frage kommenden Vervollständigungen anzeigen lassen (Abb. 6.4).

```
ronja@nash: ~
ronja@nash:~$ pr
pr              preunzip        printafm        printf
precat          prezip          printenv        protoc
preconv         prezip-bin      printerbanner   prove
prename         print           printer-profile prtstat
ronja@nash:~$ pr
```

Abbildung 6.4.: Autovervollständigung für die Eingabe `pr`

6.2. Editieren und Textdateien

Bislang ging es um Dateien, Verzeichnisse und das allgemeine Bedienen einer Shell. Im Fokus dieses Abschnittes stehen die Textdateien. Diese werden für uns relevant, da sie unter anderem den *Code* der geschriebenen Programme beherbergen werden. Ein *Texteditor* ist ein Programm, welches das Erstellen und Verändern von Textdateien erleichtert.

Texteditor

Es gibt unzählige Texteditoren. Unter KDE ist z.B. der Editor `kate` (`http://kate-editor.org/`), unter Gnome der Editor `gedit` (`http://projects.gnome.org/gedit/`) sehr empfehlenswert. Diese können allerdings nur im graphischen Modus ausgeführt werden, dafür ist ihre Bedienung dank vorhandener Mausbedienung recht komfortabel. Beide Editoren unterstützen Syntax-Highlighting für gängige Programmiersprachen. Sie können entweder über die entsprechenden Kommandos aus einer Shell heraus gestartet werden, oder über das Startmenü geöffnet werden. Unter Windows oder MacOS empfiehlt es sich für den Vorkurs einen Editor mit Syntax-Highlighting für die Programmiersprache Haskell zu verwenden, z.B. „Notepad ++“ (`http://notepad-plus-plus.org/`) für Windows und „TextWrangler“ (`http://www.barebones.com/products/textwrangler/`) für Mac OS X . Weitere Allround-Editoren sind Emacs (`http://www.gnu.org/software/emacs/` und XEmacs (`http://www.xemacs.org/`), deren Bedienung allerdings gewöhnungsbedürftig ist.

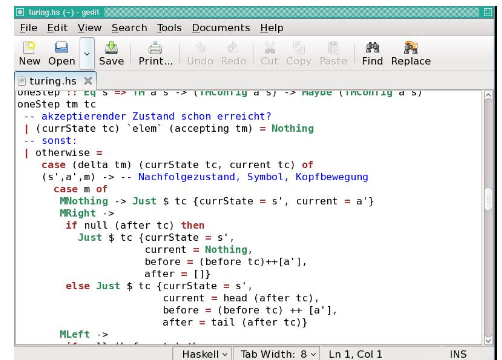
Abbildung 6.5 zeigt einige Screenshots von Editoren, wobei eine Quellcode-Datei der Programmiersprache Haskell geöffnet ist.

Hat man kein graphisches Interface zur Verfügung, so kann man einen Text-basierten Editor verwenden. Ein solcher weit verbreiteter Texteditor heißt `vi` (sprich: [vi: ai]). Dieser steht nicht nur in der RBI zur Verfügung, er ist auf fast jedem Unix-System vorhanden. Wir werden kurz auf seine Bedienung eingehen. Mit dem Befehl `vi` wird der Editor[1] gestartet. Der Befehl `vi /tmp/irgendeinedatei` startet `vi` und öffnet sogleich eine Sicht auf die angegebene Datei. Diese Sicht nennt man *Buffer*, hier findet das Editieren statt. Nach dem Öffnen und nach dem Speichern stimmt der Inhalt des Buffers mit dem der korrespondierenden Datei überein. Der `vi` unterscheidet einige Betriebsmodi, die sich wie folgt beschreiben lassen. Wichtig dabei ist die Position des *Cursors*, auf die sich die meisten Aktionen beziehen.

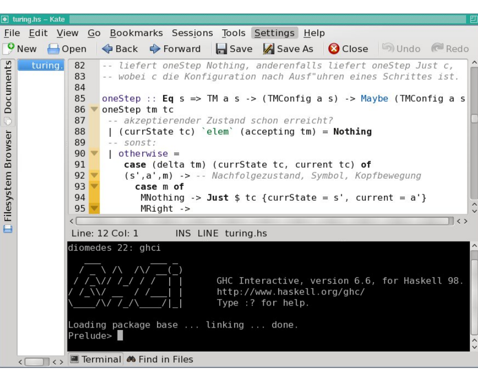
`vi`

1. Im *Befehlsmodus* werden Tastatureingaben als Befehle aufgefasst. Unter einem Befehl kann man sich das vorstellen, was man einem Herausgeber (engl.: editor) zurufen würde, bäte man ihn um Änderungen an einem Text. In diesem Modus befindet sich `vi` nach dem Starten. `vi` versteht Befehle wie „öffne/speichere diese und jene Datei“ (`:e diese`, `:w jene`), „Lösche die Zeile, in der sich der Cursor befindet!“ (`d` , `d`) oder „Tausche den Buchstaben unterm Cursor durch den folgenden aus!“ (`r`). Natürlich ist es auch möglich, den Cursor zu

[1] Oftmals handelt es sich schon um eine verbesserte Variante `vim` (für `vi` *improved*). Diese Unterscheidung soll uns hier nicht kümmern.

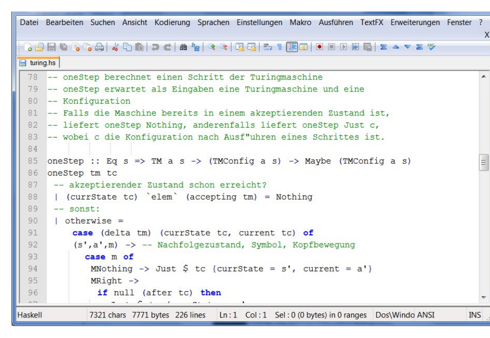


Screenshot `gedit`



Screenshot `kate`

Screenshot `xemacs`



Screenshot Notepad++ (MS Windows)

Abbildung 6.5.: Screenshots verschiedener Editoren

bewegen, etwa mit den Pfeiltasten oder mit mannigfachen anderen Tasten, die Bewegungen in alle möglichen Positionen erlauben. Viele Neider halten `:q!` für den wichtigsten aller `vi`-Befehle. Dieser führt jedoch nur zum Programmabbruch (engl.: quit) ohne vorheriges Speichern.

2. Im `EINFÜGEN`- und `ERSETZEN`-*Modus* erscheinen die eingegebenen Zeichen direkt auf dem Bildschirm, im Buffer. Im ersteren wird Text neu hinzugefügt, im zweiteren werden bereits vorhandene Zeichen überschrieben. In diese Modi gelangt man vom Befehlsmodus aus mit den Tasten [i] bzw. [R]. Zwischen ihnen kann mit der [Einfg]-Taste hin- und hergewechselt werden. Durch Betätigen der [Esc]-Taste gelangt man wieder zurück in den Befehlsmodus.
3. Die `VISUELL`en Modi erlauben das Markieren eines Bereiches des Buffers, um sodann Befehle abzusetzen, die sich auf den markierten Bereich beziehen. Befindet man sich im Befehlsmodus, so gelangt man mit der [v]-Taste in den gewöhnlichen visuellen Modus. Dieser ermöglicht das Markieren eines Textbereiches durch Bewegen des Cursors. Nach Eingabe und Ausführung eines Befehls – etwa [x], „markierten Bereich löschen“ – findet man sich im Befehlsmodus wieder. Auch die [Esc]-Taste führt zurück in den Befehlsmodus.

Wir stellen fest, dass ein Einstieg in `vi` mit dem Lernen einiger Tastenkürzel und Schlüsselworte einhergehen muss. Das lohnt sich aber für alle, die oft mit Text arbeiten. Hat man einmal die Grammatik der Befehle, die `vi` akzeptiert, verstanden, so wird das Editieren von Text zum Kinderspiel und geht schnell von der Hand. Wer das Tippen im Zehnfingersystem beherrscht und einzusetzen vermag, weiss schon, dass sich anfänglicher Mehraufwand auszahlen kann.

Ein Schnelleinstieg in `vi` ist mit Hilfe einer vorbereiteten Anleitung möglich. Diese lässt sich in einer Shell mit dem Befehl `vimtutor` starten. In dieser Anleitung sind einige Funktionen des Editors zum sofortigen Ausprobieren aufbereitet worden. Danach empfiehlt es sich, in einer Kurzreferenz zu stöbern.

7. Programmieren und Programmiersprachen

In diesem Kapitel wird zunächst kurz und knapp erklärt, was eine Programmiersprache ist und wie man die verschiedenen Programmiersprachen grob einteilen kann. Anschließend wird kurz erläutert, wie man den Interpreter GHCi für die funktionale Programmiersprache Haskell (insbesondere auf den Rechnern der RBI) verwendet. Genauere Details zum Programmieren in Haskell werden erst im nächsten Kapitel erläutert.

7.1. Programme und Programmiersprachen

Rechner

Ein Rechner besteht (vereinfacht) aus dem Prozessor (bestehend aus Rechenwerk, Steuerwerk, Registern, etc.), dem Hauptspeicher, Ein- und Ausgabegeräten (Festplatten, Bildschirm, Tastatur, Maus, etc.) und einem Bus-System über den die verschiedenen Bestandteile miteinander kommunizieren (d.h. Daten austauschen).

Maschinenprogramm

Ein *ausführbares* Computerprogramm ist eine Folge *Maschinencodebefehlen*, die man auch als *Maschinenprogramm* bezeichnet. Ein einzelner Maschinencodebefehl ist dabei eine Operation, die der Prozessor direkt ausführen kann (z.B. Addieren zweier Zahlen, Lesen oder Beschreiben eines Speicherregisters), d.h. diese Befehle „versteht“ der Prozessor und kann diese direkt verarbeiten (d.h. ausführen). Die Ausführung eines ganzen Maschinenprogramms besteht darin, die Folge von Maschinencodebefehlen nacheinander abzuarbeiten und dabei den Speicher zu manipulieren (d.h. zu verändern, oft spricht man auch vom „Zustand“ des Rechners und meint damit die gesamte Speicherbelegung).

höhere Programmiersprache

Quellcode

Compiler

Interpreter

Allerdings sind Maschinenprogramme eher schwierig zu erstellen und für den menschlichen Programmierer schwer zu verstehen. Deshalb gibt es sogenannte *höhere Programmiersprachen*, die es dem Programmierer erlauben, besser verständliche Programme zu erstellen. Diese Programme versteht der Computer allerdings nicht direkt, d.h. sie sind nicht ausführbar. Deshalb spricht man oft auch von sogenanntem *Quelltext* oder *Quellcode*. Damit aus dem Quelltext (geschrieben in einer höheren Programmiersprache) ein für den Computer verständliches (und daher ausführbares) Maschinenprogramm wird, ist eine weitere Zutat erforderlich: Entweder kann ein *Compiler* benutzt werden, oder ein *Interpreter*. Ein Compiler ist ein *Übersetzer*: Er übersetzt den Quelltext in ein Maschinenprogramm. Ein *Interpreter* hingegen führt das Programm schrittweise aus (d.h. der Interpreter ist ein Maschinenprogramm und interpretiert das Programm der höheren Programmiersprache). Neben der Übersetzung (durch den Compiler) bzw. der Ausführung (durch den Interpreter) führt ein solches Programm noch weitere Aufgaben durch. Z.B. wird geprüft, ob der Quelltext tatsächlich ein gültiges Programm der Programmiersprache ist. Ist dies nicht der Fall, so gibt ein guter Compiler/Interpreter eine Fehlermeldung aus, die dem Programmierer mitteilt, an welcher Stelle der Fehler steckt. Je nach Programmiersprache und je nach Compiler/Interpreter kann man hier schon Programmierfehler erkennen und mithilfe der Fehlermeldung korrigieren.

Es gibt unzählige verschiedene (höhere) Programmiersprachen. Wir werden gleich auf die Charakteristika eingehen, die Programmiersprachen unterscheiden. Diese Kriterien nennt man auch Programmiersprachenparadigmen oder Programmierstile.

Im Allgemeinen unterscheidet man Programmiersprachen in *imperative* und in *deklarative* Programmiersprachen.

7.1.1. Imperative Programmiersprachen

„Imperativ“ stammt vom lateinischen Wort „imperare“ ab, was „befehlen“ bedeutet. Tatsächlich besteht der Programmcode eines imperativen Programms aus einzelnen Befehlen (auch *Anweisungen* genannt), die nacheinander ausgeführt werden und den Zustand (d.h. Speicher) des Rechners verändern. Dies klingt sehr ähnlich zu den bereits erwähnten Maschinenprogrammen. Der Unterschied liegt darin, dass in höheren Programmiersprachen die Befehle komplexer und verständlicher sind, und dass meistens vom tatsächlichen Speicher abstrahiert wird, indem sogenannte Programmvariablen verwendet werden. Diese sind im Grunde Namen für Speicherbereiche, wobei der Programmierer im Programm nur die Namen verwendet, und die Abbildung der Namen auf den tatsächlichen Speicher durch den Compiler oder den Interpreter geschieht (der Programmierer braucht sich hierum nicht zu kümmern). Z.B. kann X für eine Variable stehen. Verwendet man in einem imperativen Programm die Variable X beispielsweise im *Ausdruck* $X + 5$, so ist die Bedeutung hierfür im Normalfall: Lese den Wert von X (d.h. schaue in die zugehörige Stelle im Speicher) und addiere dann die Zahl 5 dazu. Das Verändern des Speichers geschieht in imperativen Sprachen üblicherweise mithilfe der *Zuweisung*, die oft durch := dargestellt wird. Hinter dem Befehl (bzw. der Zuweisung) $X := 10$ steckt die Bedeutung: Weise dem Speicherplatz, der durch X benannt ist, den Wert 10 zu.

Ein imperatives Programm besteht aus einer Folge solcher Befehle, die bei der Ausführung sequentiell (d.h. nacheinander) abgearbeitet werden. Hierbei werden noch sogenannte *Kontrollstrukturen* verwendet, die den Ablauf des Programmes steuern können. Als Kontrollstrukturen bezeichnet man sowohl *Verzweigungen* als auch *Schleifen*. Verzweigungen sind Wenn-Dann-Abfragen, die je nachdem, ob ein bestimmtes Kriterium erfüllt ist, das eine oder das andere Programm ausführen. Schleifen ermöglichen es, eine Befehlsfolge wiederholt (oder sogar beliebig oft) auszuführen.

prozedurale Programmiersprache

Es gibt verschiedene Unterklassen von imperativen Programmiersprachen, die sich meist dadurch unterscheiden, wie man den Programmcode strukturieren kann. Z.B. gibt es *prozedurale Programmiersprachen*, die es erlauben den Code durch Prozeduren zu strukturieren und zu gruppieren. Eine *Prozedur* ist dabei ein Teilprogramm, das immer wieder (von verschiedenen anderen Stellen des Programmcodes) aufgerufen werden kann. Hierdurch kann man Programmcode sparen, da ein immer wieder verwendetes Programmstück nur einmal programmiert werden muss. Bekannte prozedurale Programmiersprachen sind z.B. *C*, *Fortran* und *Pascal*.

objektorientierte Programmiersprache

Eine weitere Unterklasse der imperativen Programmiersprachen, sind die sogenannten *Objektorientierten Programiersprachen*. In objektorientierten Programmiersprachen werden Programme durch sogenannte *Klassen* dargestellt. Klassen geben das Muster vor, wie Instanzen dieser Klasse (Instanzen nennt man auch *Objekte*) aussehen. Klassen bestehen im Wesentlichen aus *Attributen* und *Methoden*. Attribute legen die Eigenschaften fest, die ein Objekt haben muss (z.B. könnte man sich eine Klasse `Auto` vorstellen, welche die Attribute `Höchstgeschwindigkeit`, `Gewicht` und `Kennzeichen` hat). Methoden definieren, ähnlich wie Prozeduren, Programme, die das Objekt verändern können (z.B. verändern des Kennzeichens). Über die Methoden können Objekte jedoch auch miteinander kommunizieren, indem eine Methode eines Objekts eine andere Methode eines anderen Objekts aufruft. Man sagt dazu auch: „die Objekte versenden Nachrichten untereinander“.

Die Strukturierungsmethode in objektorientierten Programmiersprachen ist die *Vererbung*. Hierdurch kann man Unterklassen erzeugen, dabei übernimmt die Unterklasse sämtliche Attribute und Methoden der Oberklasse und kann noch eigene hinzufügen.

Wird ein objektorientiertes Programm ausgeführt, so werden Objekte als Instanzen von Klassen erzeugt und durch Methodenaufrufe werden Nachrichten zwischen den Objekten ausgetauscht. Objekte werden dabei im Speicher des Rechners abgelegt. Da der Zustand der Objekte bei der Ausführung des System verändert wird, wirken die Methodenaufrufe wie Befehle (die den Zustand des Systems bei der Ausführung des Programms verändern). Daher zählt man objektorientierte Sprachen zu den imperativen Programmiersprachen. Bekannte objektorientierte Programmierspra-

che sind z.B. *Java*, *C++* und *C#*, aber die meisten modernen imperativen Sprachen unterstützen auch die objektorientierte Programmierung (z.B. *Modula-3*, *Python*, *Ruby*, ...).

7.1.2. Deklarative Programmiersprachen

„Deklarativ" stammt vom lateinischen Wort „declarare" was „erklären" oder auch „beschreiben" heißt. Programme in deklarativen Programiersprachen beschreiben das Ergebnis des Programms. Dafür wird jedoch im Allgemeinen nicht genau festgelegt, *wie* das Ergebnis genau berechnet wird. Es wird eher beschrieben, *was* berechnet werden soll. Hierin liegt ein großer Unterschied zu imperativen Sprachen, denn diese geben genau an, wie der Speicher manipuliert werden soll, um dadurch das gewünschte Ergebnis zu erhalten. Programme deklarativer Programmiersprachen beschreiben im Allgemeinen nicht die Speichermanipulationen, sondern bestehen aus (oft mathematischen) *Ausdrücken*. Zur Ausführung des Programms werden diese Ausdrücke *ausgewertet*. In der Schule führt man eine solche Auswertung oft per Hand für arithmetische Ausdrücke durch. Will man z.B. den Wert des Ausdrucks $(5 \cdot 10 + 3 \cdot 8)$ ermitteln, so wertet man den Ausdruck aus (was man in der Schule auch oft als „ausrechnen" bezeichnet), z.B. durch die Rechnung $(5 \cdot 10 + 3 \cdot 8) = (50 + 3 \cdot 8) = (50 + 24) = 74$. In deklarativen Programmiersprachen gibt es wie in imperativen Sprachen auch Variablen, diese meinen aber meistens etwas anderes: Während in imperativen Sprachen Variablen veränderbare Speicherbereiche bezeichnen, so bezeichnen Variablen in deklarativen Programmiersprachen im Allgemeinen bestimmte, feststehende Ausdrücke, d.h. insbesondere ist ihr Wert *unveränderlich*. Z.B. kann man in der deklarativen Sprache Haskell schreiben `let x = 5+7 in x*x`. Hierbei ist die Variable `x` nur ein Name für den Ausdruck `5+7` und ihr Wert ist stets `12`[1].

Da deklarative Programmiersprachen i.A. keine Speichermanipulationen direkt durchführen, sind meist weder eine Zuweisung noch Schleifen zur Programmierung vorhanden (diese manipulieren nämlich den Speicher). Um jedoch Ausdrücke wiederholt auszuwerten, wird *Rekursion* bzw. werden *rekursive Funktionen* verwendet. Diese werden wir später genauer betrachten. An dieser Stelle sei nur kurz erwähnt, dass eine Funktion rekursiv ist, wenn sie sich selbst aufrufen kann.

Deklarative Sprachen lassen sich grob aufteilen in *funktionale Programmiersprachen* und *logische Programmiersprachen*.

Bei logischen Programmiersprachen besteht ein Programm aus einer Mengen von logischen Formeln und Fakten (wahren Aussagen). Zur Laufzeit werden mithilfe logischer Folgerungen (sogenannter Schlussregeln) neue wahre Aussagen hergeleitet, die dann das Ergebnis der Ausführung darstellen. Die bekannteste Vertreterin der logischen Programmiersprachen ist die Sprache *Prolog*.

logische Programmiersprache

Ein Programm in einer *funktionalen Programmiersprache* besteht aus einer Menge von Funktionsdefinitionen (im engeren mathematischen Sinn) und evtl. selbstdefinierten Datentypen. Das Ausführen eines Programms entspricht dem Auswerten eines Ausdrucks, d.h. das Resultat ist ein einziger Wert. In rein funktionalen Programmiersprachen wird der Zustand des Rechners nicht explizit durch das Programm manipuliert, d.h. es treten bei der Ausführung keine sogenannten *Seiteneffekte* (d.h. sichtbare Speicheränderungen) auf. Tatsächlich braucht man zur Auswertung eigentlich gar keinen Rechner, man könnte das Ergebnis auch stets per Hand mit Zettel und Stift berechnen (was jedoch oft sehr mühsam wäre). In rein funktionalen Programmiersprachen gilt das Prinzip der *referentiellen Transparenz*: Das Ergebnis der Anwendung einer Funktion auf Argumente, hängt ausschließlich von den Argumenten ab, oder umgekehrt: Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat.

funktionale Programmiersprache

referentielle Transparenz

Aus dieser Sicht klingen funktionale Programmiersprachen sehr mathematisch und es ist zunächst nicht klar, was man außer arithmetischen Berechnungen damit anfangen kann. Die Mächtigkeit der funktionalen Programmierung ergibt sich erst dadurch, dass die Funktionen nicht nur auf Zahlen, sondern auf beliebig komplexen Datenstrukturen (z.B. Listen, Bäumen, Paaren, usw.)

[1] Der Operator `*` bezeichnet in fast allen Computeranwendungen und Programmiersprachen die Multiplikation.

operieren dürfen. So kann man z.B. Funktionen definieren, die als Eingabe einen Text erhalten und Schreibfehler im Text erkennen und den verbesserten Text als Ausgabe zurückliefern, oder man kann Funktionen schreiben, die Webseiten nach bestimmten Schlüsselwörtern durchsuchen, etc.

Prominente Vertreter von funktionalen Programmiersprachen sind *Standard ML*, *OCaml*, Microsofts *F#* und *Haskell*. Im Vorkurs und in der Veranstaltung „Grundlagen der Programmierung 2“ werden wir die Sprache Haskell behandeln und benutzen.

7.2. Haskell: Einführung in die Benutzung

Die Programmiersprache *Haskell* ist eine pure funktionale Programmiersprache. Der Name „Haskell“ stammt von dem amerikanischen Mathematiker und Logiker Haskell B. Curry. Haskell ist eine relativ neue Programmiersprache, der erste Standard wurde 1990 festgelegt. Damals gab es bereits einige andere funktionale Programmiersprachen. Um eine einheitliche Sprache festzulegen wurde ein Komitee gegründet, das die standardisierte funktionale Programmiersprache Haskell entwarf. Inzwischen wurden mehrere Revisionen des Standards veröffentlicht (1999 und leicht verändert 2003 wurde *Haskell 98* veröffentlicht, im Juli 2010 wurde der *Haskell 2010*-Standard veröffentlicht).

Das aktuelle Haskell-Logo

Die wichtigste Informationsquelle zu Haskell ist die Homepage von Haskell:

`http://www.haskell.org`

Dort findet man neben dem Haskell-Standard zahlreiche Links, die auf Implementierungen der Sprache Haskell (d.h. Compiler und Interpreter), Bücher, Dokumentationen, Anleitungen, Tutorials, Events, etc. verweisen.

GHC,GHCi

Es gibt einige Implementierungen der Sprache Haskell. Wir werden im Vorkurs den am weitesten verbreiteten *Glasgow Haskell Compiler* (kurz GHC) benutzen. Dieser stellt neben einem Compiler für Haskell auch einen Interpreter (den sogenannten GHCi) zur Verfügung. Für den Vorkurs und auch für die Veranstaltung „Grundlagen der Programmierung 2“ ist die Benutzung des Interpreters ausreichend. Die Homepage des GHC ist `http://www.haskell.org/ghc`.

7.2.1. GHCi auf den Rechnern der RBI

Auf den RBI-Rechnern ist der Compiler GHC und der Interpreter GHCi bereits installiert. Den Haskell-Interpreter GHCi findet man unter `/opt/rbi/bin/ghci`, d.h. er kann mit dem Kommando `/opt/rbi/bin/ghci` (ausgeführt in einer Shell) gestartet werden[2].

7.2.2. GHCi auf dem eigenen Rechner installieren

Für die Installation des GHC und GHCi bietet es sich an, die *Haskell Platform* zu installieren, diese beinhaltet neben GHC und GHCi einige nützliche Werkzeuge und Programmbibliotheken für Haskell. Unter

`http://hackage.haskell.org/platform/`

[2]wenn die Umgebungsvariable `PATH` richtig gestetzt ist, genügt auch das Kommando `ghci`

stehen installierbare Versionen für verschiedene Betriebssysteme (MS Windows, Mac OS) zum Download zur Verfügung. Für Linux-basierte Systeme kann für einige Distributionen (Ubuntu, Debian, Fedora, Arch Linux, Gentoo, NixOS) die Haskell Platform über den Paketmanager installiert werden. Die Seite `http://hackage.haskell.org/platform/linux.html` enthält dazu Links (und auch Informationen für weitere Distributionen).

Alternativ (nicht empfohlen) kann der GHC/GHCi alleine installiert werden, er kann von der Homepage `http://www.haskell.org/ghc/` heruntergeladen werden.

Nach der Installation ist es i.A. möglich den GHCi zu starten, indem man das Kommando `ghci` in eine Shell eintippt (unter MS Windows ist dies auch über das Startmenü möglich).

7.2.3. Bedienung des Interpreters

Nachdem wir den Interpreter gestartet haben (siehe Abschnitt 7.2.1) erhalten wir auf dem Bildschirm

```
ghci ⏎
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Wir können nun Haskell-Ausdrücke eingeben und auswerten lassen, z.B. einfache arithmetische Ausdrücke:

```
Prelude>  1+1 ⏎
2
Prelude> 3*4 ⏎
12
Prelude> 15-6*3 ⏎
-3
Prelude> -3*4 ⏎
-12
```

Gibt man ungültige Ausdrücke ein (also Ausdrücke, die keine Haskell-Ausdrücke sind), so erhält man im Interpreter eine Fehlermeldung, z.B.

```
\Prelude> 1+2+3+4+ ⏎  <interactive>:2:9:
    parse error (possibly incorrect indentation or mismatched brackets)
```

Hierbei sollte man die Fehlermeldung durchlesen, bevor man sich auf die Suche nach dem Fehler macht. Sie enthält zumindest die Information, an welcher Stelle des Ausdrucks der Interpreter einen Fehler vermutet: Die Zahlen `1:8` verraten, dass der Fehler in der 2. Zeile und der 9. Spalte (im interaktiven Modus werden die Zeilen stets weitergezählt, daher ist Zeile 1 für den Text `GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help` verbraucht) ist. Tatsächlich ist dort das `+`-Zeichen, dem keine Zahl folgt. Wir werden später noch weitere Fehlermeldungen betrachten.

Neben Haskell-Ausdrücken können wir im Interpreter auch Kommandos zur Steuerung des Interpreters absetzen. Diese werden stets mit einem Doppelpunkt eingeleitet (damit der Interpreter selbst sie von Haskell-Programmen unterscheiden kann). Einige wichtige Kommandos sind:

`:quit`	Verlassen des Interpreters. Der Interpreter wird gestoppt, und es wird zur Shell zurück gekehrt.
`:help`	Der Interpreter zeigt einen Hilfetext an. Insbesondere wird eine Übersicht über die verfügbaren Kommandos gegeben.
`:load` *Dateiname*	Lädt den Haskell-Quellcode der entsprechenden Datei, die Dateiendung von *Dateiname* muss `.hs` lauten.
`:reload`	Lädt die aktuelle geladene Datei erneut (hilfreich, wenn man die aktuell geladene Datei im Editor geändert hat).

7.2.4. Quelltexte erstellen und im GHCi laden

Normalerweise erstellt man den Quelltext eines Haskell-Programms in einem Editor (siehe Kapitel 6), und speichert das Haskell-Programm in einer Datei. Die Datei-Endung muss hierbei `.hs` lauten. Abbildung 7.1 zeigt den Inhalt eines ganz einfachen Programms. Es definiert für den Namen `wert` (eigentlich ist `wert` eine Funktion, die allerdings keine Argumente erhält) die Zeichenfolge „Hallo Welt!“ (solche Zeichenfolgen bezeichnet man auch als *String*).

```
wert = "Hallo Welt!"
```

Abbildung 7.1.: Inhalt der Datei `hallowelt.hs`

Nach dem Erstellen der Datei können wir sie im GHCi mit dem Kommando `:load hallowelt.hs` laden:

```
> ghci ↩
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load hallowelt.hs ↩
[1 of 1] Compiling Main  ( hallowelt.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Das funktioniert aber nur, wenn der GHCi aus dem Verzeichnis heraus gestartet wurde, das die Datei `hallowelt.hs` enthält. Angenommen wir starten den GHCi aus einem Verzeichnis, aber `hallowelt.hs` liegt in einem Unterverzeichnis namens `programme`, so müssen wir dem `:load`-Kommando nicht nur den Dateinamen (`hallowelt.hs`), sondern den Verzeichnispfad mit übergeben (also `:load programme/hallowelt.hs`). Wir zeigen was passiert, wenn man den Verzeichnispfad vergisst, und wie es richtig ist:

```
> ghci ↩
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load hallowelt.hs ↩

<no location info>: can't find file: hallowelt.hs
Failed, modules loaded: none.
Prelude> :load programme/hallowelt.hs ↩
[1 of 1] Compiling Main ( programme/hallowelt.hs, interpreted )
Ok, modules loaded: Main.
```

Nach dem Laden des Quelltexts, können wir die dort definierten Funktionen im Interpreter auswerten lassen. Wir haben nur die Parameter-lose Funktion `wert` definiert, also lassen wir diese mal auswerten:

```
*Main> wert ⏎
"Hallo Welt!"
```

Wir erhalten als Ergebnis der Berechnung gerade den String „Hallo Welt". Wirklich rechnen musste der Interpreter hierfür nicht, da der `wert` schon als dieser String definiert wurde.

Abbildung 7.2 zeigt den Inhalt einer weiteren Quelltextdatei (namens `einfacheAusdruecke.hs`). Dort werden wieder nur Parameter-lose Funktionen definiert, aber rechts vom `=` stehen *Ausdrücke*,

```
zwei_mal_Zwei = 2 * 2

oft_fuenf_addieren = 5 + 5 + 5 + 5 +5 + 5 + 5 + 5 + 5 + 5 + 5

beides_zusammenzaehlen = zwei_mal_Zwei + oft_fuenf_addieren
```

Abbildung 7.2.: Inhalt der Datei `einfacheAusdruecke.hs`

die keine Werte sind. Die Funktion `zwei_mal_Zwei` berechnet das Produkt $2 \cdot 2$, die Funktion `oft_fuenf_addieren` addiert elf mal 5 und die Funktion `beides_zusammenzaehlen` addiert die Werte der beiden anderen Funktionen. Dafür *ruft* sie die beiden anderen Funktionen *auf*.

Funktionsaufruf

Nach dem Laden des Quelltexts, kann man die Werte der definierten Funktionen berechnen lassen:

```
> ghci ⏎
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load einfacheAusdruecke.hs ⏎
[1 of 1] Compiling Main  ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> zwei_mal_Zwei ⏎
4
*Main> oft_fuenf_addieren ⏎
55
*Main> beides_zusammenzaehlen ⏎
59
*Main> 3*beides_zusammenzaehlen ⏎
177
```

Beachte, dass Funktionsnamen in einer Quelltextdatei mit einem Kleinbuchstaben oder dem Unterstrich _ beginnen *müssen*, anschließend können Großbuchstaben und verschiedene Sonderzeichen folgen. Hält man sich nicht an diese Regel, so erhält man beim Laden des Quelltexts eine Fehlermeldung. Ersetzt man z.B. `zwei_mal_Zwei` durch `Zwei_mal_Zwei` so erhält man:

Funktionsnamen

```
Prelude> :load grossKleinschreibungFalsch.hs
[1 of 1] Compiling Main ( grossKleinschreibungFalsch.hs, interpreted )

grossKleinschreibungFalsch.hs:3:1:
    Not in scope: data constructor ‘Zwei_mal_Zwei’
Failed, modules loaded: none.
```

Zur Erklärung der Fehlermeldung: Der GHCi nimmt aufgrund der Großschreibung an, dass `Zwei_mal_Zwei` ein Datenkonstruktor ist (und daher keine Funktion). Was Datenkonstruktoren sind, erläutern wir an dieser Stelle nicht, wichtig ist nur, dass der GHCi die Funktion nicht als eine solche akzeptiert.

7.2.5. Kommentare in Quelltexten

In Quelltexten sollten neben dem eigentlichen Programmcode auch Erklärungen und Erläuterungen stehen, die insbesondere umfassen: Was macht jede der definierten Funktionen? Wie funktioniert die Implementierung, bzw. was ist die Idee dahinter? Man sagt auch: Der Quelltext soll *dokumentiert* sein. Der Grund hierfür ist, dass man selbst nach einiger Zeit den Quelltext wieder verstehen, verbessern und ändern kann, oder auch andere Programmierer den Quelltext verstehen. Um dies zu bewerkstelligen, gibt es in allen Programmiersprachen die Möglichkeit *Kommentare* in den Quelltext einzufügen, wobei diese speziell markiert werden müssen, damit der Compiler oder Interpreter zwischen Quellcode und Dokumentation unterscheiden kann. In Haskell gibt es zwei Formen von Kommentaren:

Zeilenkommentare: Fügt man im Quelltext in einer Zeile zwei Minuszeichen gefolgt von einem Leerzeichen ein, d.h. „`-- `“, so werden alle Zeichen danach bis zum Zeilenende als Kommentar erkannt und dementsprechend vom GHCi ignoriert. Zum Beispiel:

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

Kommentarblöcke: Man kann in Haskell einen ganzen Textblock (auch über mehrere Zeilen) als Kommentar markieren, indem man ihn in durch spezielle Klammern einklammert. Die öffnende Klammer besteht aus den beiden Symbolen `{-` und die schließende Klammer besteht aus `-}`. Zum Beispiel ist im folgenden Programm nur `wert2 = "Hallo Welt"` Programmcode, der Rest ist ein Kommentar:

```
{- Hier steht noch gar keine Funktion,
   da auch die naechste Zeile noch im
   Kommentar ist

wert = "Hallo Welt"

   gleich endet der Kommentar -}

wert2 = "Hallo Welt"
```

7.2.6. Fehler

Jeder Programmierer erstellt Programme, die fehlerhaft sind. Es gibt jedoch verschiedene Arten von Fehlern, die wir kurz erläutern. *Syntaxfehler* entstehen, wenn der Quelltext ein Programm enthält, das syntaktisch nicht korrekt ist. Z.B. kann das Programm Symbole enthalten, die die Programmiersprache nicht erlaubt (z.B. `5!` für die Fakultät von 5, aber die Sprache verfügt nicht über den Operator `!`). Ein anderer syntaktischer Fehler ist das Fehlen von Klammern, oder allgemein die nicht korrekte Klammerung (z.B. `(5+3` oder auch `(4*2))`). Ein andere Klasse von Fehlern sind sogenannte *logische* oder *semantische* Fehler. Ein solcher Fehler tritt auf, wenn das Programm nicht die gewünschte Funktionalität, aber irgendeine andere Funktionalität, implementiert.

Syntaxfehler

Logischer Fehler

Syntaxfehler sind eher leicht zu entdecken (auch automatisch), während logische Fehler eher schwer zu erkennen sind. In die Klasse der semantischen Fehler fallen auch sogenannte *Typfehler*. Ein Typfehler tritt auf, wenn Konstrukte der Sprache miteinander verwendet werden, obwohl sie nicht zueinander passen. Ein Beispiel ist `1+'A'`, da man eine Zahl nicht mit einem Buchstaben addieren kann. Wir werden später Typfehler genauer behandeln.

Typfehler

Man kann Programmierfehler auch danach unterscheiden, *wann* sie auftreten. Hierbei unterscheidet man in *Compilezeitfehler* und *Laufzeitfehler*. Wenn ein Fehler bereits beim Übersetzen des Programms in Maschinensprache entdeckt wird, dann spricht man von einem Compilezeitfehler. In diesem Fall bricht der Compiler die Übersetzung ab, und meldet dem Programmierer den Fehler. Auch der GHCi liefert solche Fehlermeldungen. Betrachte beispielsweise den folgenden Quellcode in der Datei `fehler.hs`

Compilezeitfehler

```
-- 1 und 2 addieren
eineAddition = (1+2)

-- 2 und 3 multiplizieren
eineMultiplikation = (2*3))
```

Laden wir diese Datei im GHCi , so erhalten wir eine Fehlermeldung:

```
Prelude> :load fehler.hs
[1 of 1] Compiling Main             ( fehler.hs, interpreted )

fehler.hs:5:27: parse error on input ‘)’
Failed, modules loaded: none.
```

Es empfiehlt sich, die ausgegebene Fehlermeldung genau zu lesen, denn sie verrät oft, wo sich der Fehler versteckt (in diesem Fall in Zeile 5 und Spalte 27), um welche Art von Fehler es sich handelt (in diesem Fall ein „parse error“, was einem Syntaxfehler entspricht), und welches Symbol zum Fehler geführt hat (in diesem Fall die schließende Klammer).

Ein *Laufzeitfehler* ist ein Fehler, der nicht vom Compiler entdeckt wird, und daher erst beim *Ausführen* des Programms auftritt. Das Programm bricht dann normalerweise ab. Gute Programme führen eine Fehlerbehandlung durch und Vermeiden daher das plötzliche Abbrechen des Programms zur Laufzeit. Ein Beispiel für einen Laufzeitfehler ist die Division durch 0.

Laufzeitfehler

```
Prelude> div 10 0
*** Exception: divide by zero
```

Andere Beispiele sind das Lesen von Dateien, die gar nicht existieren, etc.

Stark und statisch getypte Programmiersprache wie Haskell haben den Vorteil, das viele Fehler bereits vom Compiler entdeckt werden, und daher Laufzeitfehler vermieden werden.

8. Grundlagen der Programmierung in Haskell

In diesem Kapitel werden wir die Programmierung in Haskell genauer kennen lernen. Es sei vorab erwähnt, dass wir im Rahmen dieses Vorkurses nicht den gesamten Umfang von Haskell behandeln können. Wir werden viele wichtige Konzepte von Haskell in diesem Rahmen überhaupt nicht betrachten: Eine genaue Betrachtung des Typenkonzepts fehlt ebenso wie wichtige Datenstrukturen (z.B. Arrays, selbstdefinierte Datentypen, unendliche Datenstrukturen). Auch auf die Behandlung von Ein- und Ausgabe unter Verwendung der `do`-Notation werden wir nicht eingehen.

Ziel des Vorkurses ist vielmehr das Kennenlernen und der Umgang mit der Programmiersprache Haskell, da diese im ersten Teil der Veranstaltung „Grundlagen der Programmierung 2“ weiter eingeführt und verwendet wird.

8.1. Ausdrücke und Typen

Das Programmieren in Haskell ist im Wesentlichen ein Programmieren mit *Ausdrücken*. Ausdrücke werden aufgebaut aus kleineren Unterausdrücken. Wenn wir als Beispiel einfache arithmetische Ausdrücke betrachten, dann sind deren kleinsten Bestandteile Zahlen `1,2,...`. Diese können durch die Anwendung von arithmetischen Operationen `+,-,*` zu größeren Ausdrücken zusammengesetzt werden, beispielsweise `17*2+5*3`. Fast jeder Ausdruck besitzt einen *Wert*, im Falle von elementaren Ausdrücken, wie `1` oder `2`, kann der Wert direkt abgelesen werden. Der Wert eines zusammengesetzten Ausdrucks muss berechnet werden. In Haskell stehen, neben arithmetischen, eine ganze Reihe andere Arten von Ausdrücken bereit um Programme zu formulieren. Die wichtigste Methode, um in Haskell Ausdrücke zu konstruieren, ist die Anwendung von Funktionen auf Argumente. In obigem Beispiel können die arithmetischen Operatoren `+,-,*` als Funktionen aufgefasst werden, die infix (`17*2`) anstelle von präfix (`* 17 2`) notiert werden.

Ausdruck

Wert

Haskell ist eine *streng getypte* Programmiersprache, d.h. jeder Ausdruck und jeder Unterausdruck hat einen *Typ*. Setzt man Ausdrücke aus kleineren Ausdrücken zusammen, so müssen die Typen stets zueinander passen, z.B. darf man Funktionen, die auf Zahlen operieren, nicht auf Strings anwenden, da die Typen nicht zueinander passen.

Typ

Man kann sich im GHCi, den Typ eines Ausdrucks mit dem Kommando `:type` *Ausdruck* anzeigen lassen, z.B. kann man eingeben:

```
Prelude> :type 'C' ⏎
'C' :: Char
```

Man sagt der Ausdruck (bzw. der Wert) `'C'` *hat den Typ* `Char`. Hierbei steht `Char` für *Character*, d.h. dem englischen Wort für Buchstabe. Typnamen wie `Char` beginnen in Haskell stets mit einem Großbuchstaben. Mit dem Kommando `:set +t` kann man den GHCi in einen Modus versetzen, so dass er stets zu jedem Ergebnis auch den Typ anzeigt, das letzte berechnete Ergebnis ist im GHCi immer über den Namen `it` (für „es“) ansprechbar, deshalb wird der Typ des Ergebnisses in der Form `it ::` *Typ* angezeigt. Wir probieren dies aus:

`Char`

```
> ghci ⏎
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load einfacheAusdruecke.hs
[1 of 1] Compiling Main ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +t ⏎
*Main> zwei_mal_Zwei ⏎
4
it :: Integer
*Main> oft_fuenf_addieren ⏎
55
it :: Integer
*Main> beides_zusammenzaehlen ⏎
59
it :: Integer
```

`Integer`

Die Ergebnisse aller drei Berechnungen sind vom Typ `Integer`, der beliebig große ganze Zahlen darstellt. Man kann in Haskell den Typ eines Ausdrucks auch selbst angeben (die Schreibweise ist *Ausdruck*: : *Typ*), der GHCi überprüft dann, ob der Typ richtig ist. Ein Beispiel hierzu ist:

```
*Main> 'C'::Char ⏎
'C'
it :: Char
*Main> 'C'::Integer ⏎

<interactive>:2:1:
    Couldn't match expected type ‘Integer’ with actual type ‘Char’
    In the expression: 'C' :: Integer
    In an equation for ‘it’: it = 'C' :: Integer
```

Da `'C'` ein Zeichen und daher vom Typ `Char` ist, schlägt die Typisierung als Zahl vom Typ `Integer` fehl.

Typinferenz

In den allermeisten Fällen muss der Programmierer den Typ *nicht* selbst angegeben, da der GHCi den Typ selbstständig herleiten kann. Dieses Feature nennt man auch *Typinferenz*. Manchmal ist es jedoch hilfreich, sich selbst die Typen zu überlegen, sie anzugeben, und anschließend den GHCi zum Überprüfen zu verwenden.

Im folgenden Abschnitt stellen wir einige vordefinierte und eingebaute Datentypen vor, die Haskell zur Verfügung stellt.

8.2. Basistypen

8.2.1. Wahrheitswerte: Der Datentyp `Bool`

Die Boolesche Logik (benannt nach dem englischen Mathematiker George Boole) ist eine sehr einfache Logik. Sie wird in vielen Bereichen insbesondere in wohl jeder Programmiersprache verwendet, sie ist zudem *die* Grundlage der Hardware von Rechnern. Wir führen sie hier nur sehr oberflächlich ein. Die Boolesche Logik baut auf sogenannten atomaren Aussagen auf. Für eine solche Aussage kann nur gelten: Die Aussage ist entweder *wahr* oder die Aussage ist *falsch*. Beispiele für solche atomaren Aussagen aus dem natürlichen Sprachgebrauch sind:

- Heute regnet es.
- Mein Auto hat die Farbe blau.
- Fritz ist noch nicht erwachsen.

In Programmiersprachen findet man häufig Aussagen der Form wie `x < 5`, die entweder wahr oder falsch sind.

In Haskell gibt es den Datentyp `Bool`, der die beiden Wahrheitswerte „wahr“ und „falsch“ durch die *Datenkonstruktoren* `True` und `False` darstellt, d.h. wahre Aussagen werden zu `True`, falsche Aussagen werden zu `False` ausgewertet. Datenkonstruktoren (wie `True` und `False`) beginnen in Haskell nie mit einem Kleinbuchstaben, daher fast immer mit einem Großbuchstaben.

`Bool`
Daten-konstruktor

Die Boolesche Logik stellt sogenannte *Junktoren* zur Verfügung, um aus atomaren Aussagen und Wahrheitswerten größere Aussagen (auch aussagenlogische *Formeln* genannt) zu erstellen, d.h. Formeln werden gebildet, indem kleinere Formeln (die auch nur Variablen oder Wahrheitswerte sein können) mithilfe von *Junktoren* verknüpft werden.

Junktor

Die drei wichtigsten Junktoren sind die Negation, die Und-Verknüpfung und die Oder-Verknüpfung. Hier stimmt der natürlichsprachliche Gebrauch von „nicht“, „und“ und „oder“ mit der Bedeutung der Junktoren überein.

Will man z.B. die atomare Aussage A_1: „Heute regnet es.“ negieren, würde man sagen A_2: „Heute regnet es *nicht*“, d.h. man negiert die Aussage, dabei gilt offensichtlich: Die Aussage A_1 ist genau dann wahr, wenn die Aussage A_2 falsch ist, und umgekehrt. In mathematischer Notation schreibt man die Negation als $\neg$ und könnte daher anstelle von A_2 auch $\neg A_1$ schreiben.

$\neg$

Die Bedeutung (Auswertung) von Booleschen Junktoren wird oft durch eine Wahrheitstabelle repräsentiert, dabei gibt man für alle möglichen Belegungen der atomaren Aussagen, den Wert der Verknüpfung an. Für $\neg$ sieht diese Wahrheitstabelle so aus:

A	$\neg A$
wahr	falsch
falsch	wahr

Die Und-Verknüpfung (mathematisch durch das Symbol $\wedge$ repräsentiert) verknüpft zwei Aussagen durch ein „Und“. Z.B. würde man die Verundung der Aussagen A_1: „Heute regnet es“ und A_2: „Ich esse heute Pommes“ natürlichsprachlich durch „Heute regnet es *und* ich esse heute Pommes“ ausdrücken. In der Booleschen Logik schreibt man $A_1 \wedge A_2$. Die so zusammengesetzte Aussage ist nur dann wahr, wenn beide Operanden des $\wedge$ wahr sind, im Beispiel ist die Aussage also nur wahr, wenn es heute regnet und ich heute Pommes esse.

$\wedge$

Die Wahrheitstabelle zum logischen Und ist daher:

A_1	A_2	$A_1 \wedge A_2$
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

Die Oder-Verknüpfung (mathematisch durch das Symbol $\vee$ repräsentiert) verknüpft analog zwei Aussagen durch ein „Oder“, d.h. $A_1 \vee A_2$ entspricht der Aussage „Es regnet heute *oder* ich esse heute Pommes“. Die so gebildete Aussage ist wahr, sobald einer der Operanden wahr ist (aber auch dann, wenn beide wahr sind). D.h.: Wenn es heute regnet, ist die Aussage wahr, wenn ich heute Pommes esse, ist die Aussage wahr, und auch wenn beide Ereignisse eintreten.

$\vee$

Die Wahrheitstabelle zum logischen Oder ist:

A_1	A_2	$A_1 \lor A_2$
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

In Haskell gibt es vordefinierte Operatoren für die Junktoren: `not`, `&&` und `||`, wobei die folgende Tabelle deren Entsprechung zu den mathematischen Junktoren zeigt:

Bezeichnnung	Mathematische Notation	Haskell-Notation
logische Negation	$\neg$ F	`not` F
logisches Und	$F_1 \land F_2$	F_1 `&&` F_2
logisches Oder	$F_1 \lor F_2$	F_1 `||` F_2

Wir probieren die Wahrheitswerte und die Junktoren gleich einmal im GHCi mit einigen Beispielen aus:

```
*Main> not True ⏎
False
it :: Bool
*Main> True && True ⏎
True
it :: Bool
*Main> False && True ⏎
False
it :: Bool
*Main> False || False ⏎
False
it :: Bool
*Main> True || False ⏎
True
it :: Bool
```

8.2.2. Ganze Zahlen: `Int` und `Integer`

`Int`, `Integer`

Für ganze Zahlen sind in Haskell zwei verschiedene Typen eingebaut: Der Typ `Int` hat als Werte die ganzen Zahlen im Bereich zwischen -2^{31} bis $2^{31}-1$. Der zweite Typ `Integer` umfasst die gesamten ganzen Zahlen. Die Darstellung der Werte vom Typ `Int` und der Werte vom Typ `Integer` (im entsprechenden Bereich) ist identisch, d.h. z.B. wenn man `1000` eingibt, so weiß der Interpreter eigentlich nicht, welchen Typ man meint, man kann dies durch Angabe des Typs festlegen, indem man `1000::Int` bzw. `1000::Integer` eingibt. Lässt man den Typ weg, so führt der Interpreter manchmal sogenanntes *Defaulting* durch (wenn es nötig ist) und nimmt automatisch den Typ `Integer` an. Fragen wir den Interpeter doch mal nach dem Typ von `1000`:

```
Prelude> :type 1000 ⏎
1000 :: (Num t) => t
```

Der ausgegebene Typ ist weder `Integer` noch `Int`, sondern `(Num t) => t`. Den Teil vor dem `=>` nennt man *Typklassenbeschränkung*. Wir erklären Typklassen hier nicht genau, aber man kann den

Typ wie folgt interpretieren: `1000` hat den Typ `t`, *wenn* `t` *ein Typ der Typklasse* `Num` *ist*[1]. Sowohl `Int` als auch `Integer` sind Typen der Typklasse `Num`, d.h. der Compiler kann für die *Typvariable* `t` sowohl `Integer` als `Int` einsetzen. Die vordefinierten Operatoren für ganze Zahlen erläutern wir später.

8.2.3. Gleitkommazahlen: `Float` und `Double`

Haskell stellt die Typen `Float` und `Double` (mit doppelter Genauigkeit) für Gleitkommazahlen (auch *Fließkommazahlen* genannt) zur Verfügung. Die Kommastelle wird dabei wie im Englischen üblich mit einem Punkt vom ganzzahligen Teil getrennt, insbesondere ist die Darstellung für Zahlen vom Typ `Float` und vom Typ `Double` identisch. Auch für Gleitkommazahlen gibt es eine Typklasse, die als Typklassenbeschränkung verwendet wird (die Klasse heißt `Fractional`). Für das Defaulting nimmt der GHCi den Typ `Double` an. Man kann analog wie bei ganzen Zahlen experimentieren:

`Float`, `Double`

```
Prelude> :type 10.5 ⏎
10.5 :: (Fractional t) => t
```

Man muss den Punkt nicht stets angeben, denn auch `Float` und `Double` gehören zur Typklasse `Num`. Beachte, dass das Rechnen mit Gleitkommazahlen *ungenau* werden kann, da diese nur eine bestimmte Anzahl von Nachkommastellen berücksichtigen, wobei `Double` doppelt so viele Nachkommastellen berücksichtigt als `Float`. Einige Aufrufe im GHCi, welche die Ungenauigkeit zeigen, sind:

```
Prelude> :set +t ⏎
Prelude> (1.0000001)::Float ⏎
1.0000001
it :: Float
Prelude> (1.00000001)::Float ⏎
1.0
it :: Float
Prelude> 1.000000000000001 ⏎
1.000000000000001
it :: Fractional a => a
Prelude> (1.0000000000000001) ⏎
1.0
it :: Fractional a => a
```

8.2.4. Zeichen und Zeichenketten

Zeichen sind in Haskell eingebaut durch den Typ `Char`. Ein Zeichen wird eingerahmt durch einzelne Anführungszeichen, z.B. `'A'` oder `'&'`. Es gibt einige spezielle Zeichen, die alle mit einem \ („Backslash") eingeleitet werden. Dies sind zum einen sogenannte Steuersymbole zum anderen die Anführungszeichen und der Backslash selbst. Ein kleine Auflistung ist

[1] Typklassen bündeln verschiedene Typen und stellen gemeinsame Operationen auf ihnen bereit. Dadurch, dass die Typen `Int` und `Integer` in der `Num` Typklasse sind (der Typklasse für Zahlen), wird sichergestellt, dass typische Operationen auf Zahlen (Addition, Multiplikation, usw.) für Daten beiden Typs bereitstehen.

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
`'\\'`	Backslash \
`'\''`	einfaches Anführungszeichen '
`'\"'`	doppeltes Anführungszeichen "
`'\n'`	Zeilenumbruch
`'\t'`	Tabulator

String

Zeichenketten bestehen aus einer Folge von Zeichen. Sie sind in Haskell durch den Typ `String` implementiert und werden durch doppelte Anführungszeichen umschlossen, z.B. ist die Zeichenkette `"Hallo"` ein Wert vom Typ `String`. Beachte, dass der Typ `String` eigentlich nur eine Abkürzung für `[Char]` ist, d.h. Strings sind gar nicht primitiv eingebaut (daher eigentlich auch keine Basistypen), sondern nichts anderes als Listen von Zeichen. Die eckigen Klammern um `Char` im Typ `[Char]` besagt, dass der Typ eine Liste von Zeichen ist. Wir werden Listen in einem späteren Abschnitt genauer erörtern.

8.3. Funktionen und Funktionstypen

In Haskell sind einige Funktionen bzw. Operatoren für Zahlen bereits vordefiniert, die arithmetischen Operationen Addition, Substraktion, Multiplikation und Division sind über die Operatoren `+`, `-`, `*` und `/` verfügbar:

Bezeichnung	mathematische Notation(en)	Haskell-Notation
Addition	$a + b$	`a + b`
Subtraktion	$a - b$	`a - b`
Multiplikation	$a \cdot b$, oft auch ab	`a * b`
Division	$a : b$, a/b, $a \div b$	`a / b`

Die Operatoren werden (wie in der Mathematik) als infix-Operationen verwendet, z.B. `3 * 6`, `10.0 / 2.5`, `4 + 5 * 4`. Der GHCi beachtet dabei die Punkt-vor-Strich-Regel, z.B. ist der Wert von `4 + 5 * 4` die Zahl `24` und *nicht* `36` (was dem Ausdruck `(4 + 5) * 4` entspräche). Das Minuszeichen wird nicht nur für die Subtraktion, sondern auch zur Darstellung negativer Zahlen verwendet (in diesem Fall präfix, d.h. vor der Zahl). Manchmal muss man dem Interpreter durch zusätzliche Klammern helfen, damit er „weiß“, ob es sich um eine negative Zahl oder um die Subtraktion handelt. Ein Beispiel hierfür ist der Ausdruck `2 * -2`, gibt man diesen im Interpreter ein, so erhält man eine Fehlermeldung:

```
Prelude> 2 * -2 ⏎

<interactive>:2:1:
Precedence parsing error
   cannot mix '*' [infixl 7] and
   prefix '-' [infixl 6] in the
   same infix expression
```

Klammert man jedoch `(-2)`, so kann der Interpreter den Ausdruck erkennen:

```
Prelude> 2 * (-2) ⏎
-4
Prelude>
```

Zum Vergleich von Werten stellt Haskell die folgenden Vergleichsoperatoren zur Verfügung:

Bezeichnung	mathematische Notation(en)	Haskell-Notation
Gleichheit	$a = b$	a `==` b
Ungleichheit	$a \neq b$	a `/=` b
echt kleiner	$a < b$	a `<` b
echt größer	$a > b$	a `>` b
kleiner oder gleich	$a \leq b$	a `<=` b
größer oder gleich	$a \geq b$	a `>=` b

Die Vergleichsoperatoren nehmen zwei Argumente und liefern einen Wahrheitswert, d.h. sie werten zu `True` oder `False` aus. Beachte, dass der Gleichheitstest `==` und der Ungleichheitstest `/=` nicht nur für Zahlen definiert ist, sondern für viele Datentypen verwendet werden kann. Z.B. kann man auch Boolesche Werte damit vergleichen.

Wir testen einige Beispiele:

```
Prelude> 1 == 3 ⏎
False
Prelude> 3*10 == 6*5 ⏎
True
Prelude> True == False ⏎
False
Prelude> False == False ⏎
True
Prelude> 2*8 /= 64 ⏎
True
Prelude> 2+8 /= 10 ⏎
False
Prelude> True /= False ⏎
True
```

Einige Beispiele für die Vergleiche sind:

```
Prelude> 5 >= 5 ⏎
True
Prelude> 5 > 5 ⏎
False
Prelude> 6 > 5 ⏎
True
Prelude> 4 < 5 ⏎
True
Prelude> 4 < 4 ⏎
False
Prelude> 4 <= 4 ⏎
True
```

Die arithmetischen Operationen und die Vergleichsoperationen sind auch Funktionen, sie besitzen jedoch die Spezialität, dass sie infix verwendet werden. Die meisten Funktionen werden präfix verwendet, wobei die Notation in Haskell leicht verschieden ist, von der mathematischen Notation. Betrachten wir die Funktion, die zwei Zahlen erhält und den Rest einer Division mit Rest berechnet. Sei f diese Funktion. In der Mathematik würde man zur Anwendung der Funktion f auf die

zwei Argumente 10 und 3 schreiben $f(10, 3)$, um anschließend den Wert 1 zu berechnen (denn $10 \div 3 = 3$ Rest 1). In Haskell ist dies ganz ähnlich, jedoch werden die Argumente der Funktion *nicht* geklammert, sondern jeweils durch ein Leerzeichen getrennt. D.h. in Haskell würde man schreiben `f 10 3` oder auch `(f 10 3)`. Der Grund für diese Schreibweise liegt vor allem darin, dass man in Haskell auch *partiell anwenden* darf, d.h. bei der Anwendung von Funktionen auf Argumente müssen nicht immer genügend viele Argumente vorhanden sein. Für unser Beispiel bedeutet dies: Man darf in Haskell auch schreiben `f 10`. Mit der geklammerten Syntax wäre dies nur schwer möglich. `f 10` ist in diesem Fall immer noch eine Funktion, die *ein* weiteres Argument erwartet. In Haskell ist die Funktion zur Berechnung des Restes tatsächlich schon vordefiniert sie heißt dort `mod`. Analog gibt es die Funktion `div`, die den ganzahligen Anteil der Division mit Rest berechnet. Wir probieren dies gleich einmal im Interpreter aus:

partielle Anwendung

`mod`, `div`

```
Prelude> mod 10 3 ⏎
1
Prelude> div 10 3 ⏎
3
Prelude> mod 15 5 ⏎
0
Prelude> div 15 5 ⏎
3
Prelude> (div 15 5) + (mod 8 6) ⏎
5
```

Tatsächlich kann man die Infix-Operatoren wie `+` und `==` auch in Präfix-Schreibweise verwenden, indem man sie in Klammern setzt. Z.B. kann `5 + 6` auch als `(+) 5 6` oder `True == False` auch als `(==) True False` geschrieben werden. Umgekehrt kann man zweistellige Funktionen wie `mod` und `div` auch in Infix-Schreibweise verwenden, indem man den Funktionsnamen in Hochkommata (diese werden durch die Tastenkombination ⇧ + ` eingegeben) umschließt, d.h. `mod 5 3` ist äquivalent zu ``5 `mod` 3``.

In Haskell haben nicht nur Basiswerte einen Typ, sondern jeder Ausdruck hat einen Typ. Daher haben auch Funktionen einen Typ. Als einfaches Beispiel betrachten wir erneut die Booleschen Funktionen `not`, `(&&)` und `(||)`. Die Funktion `not` erwartet einen booleschen Ausdruck (d.h. einen Ausdruck vom Typ `Bool`) und liefert einen Wahrheitswert vom Typ `Bool`. In Haskell wird dies wie folgt notiert: Die einzelnen Argumenttypen und der Ergebnistyp werden durch `->` voneinander getrennt, d.h. `not :: Bool -> Bool`. Der GHCi verrät uns dies auch:

Funktionstyp, `->`

```
Prelude> :type not ⏎
not :: Bool -> Bool
```

Die Operatoren `(&&)` und `(||)` erwarten jeweils zwei boolesche Ausdrücke und liefern als Ergebnis wiederum einen booleschen Wert. Daher ist ihr Typ `Bool -> Bool -> Bool`. Wir überprüfen dies im GHCi:

```
Prelude> :type (&&) ⏎
(&&) :: Bool -> Bool -> Bool
Prelude> :type (||) ⏎
(||) :: Bool -> Bool -> Bool
```

Allgemein hat eine Haskell-Funktion f, die n Eingaben (d.h. Argumente) erwartet, einen Typ der Form

$$f :: \underbrace{Typ_1}_{\text{Typ des 1. Arguments}} \texttt{->} \underbrace{Typ_2}_{\text{Typ des 2. Arguments}} \texttt{->} \ldots \texttt{->} \underbrace{Typ_n}_{\text{Typ des } n\text{. Arguments}} \texttt{->} \underbrace{Typ_{n+1}}_{\text{Typ des Ergebnisses}}$$

Der Pfeil `->` in Funktionstypen ist rechts-geklammert, d.h. z.B. ist `(&&) :: Bool -> Bool -> Bool` äquivalent zu `(&&) :: Bool -> (Bool -> Bool)`. Das passt nämlich gerade zur partiellen Anwendung: Der Ausdruck `(&&) True` ist vom Typ her eine Funktion, die noch einen Booleschen Wert als Eingabe nimmt und als Ausgabe einen Booleschen Wert liefert. Daher kann man `(&&)` auch als Funktion interpretieren, die *eine* Eingabe vom Typ `Bool` erwartet und als Ausgabe eine *Funktion* vom Typ `Bool -> Bool` liefert. Auch dies kann man im GHCi testen:

```
Prelude> :type ((&&) True)
((&&) True) :: Bool -> Bool
Prelude> :type ((&&) True False)
((&&) True False) :: Bool
```

Kehren wir zurück zu den Funktion und Operatoren auf Zahlen. Wie muss der Typ von `mod` und `div` aussehen? Beide Funktionen erwarten zwei ganze Zahlen und liefern als Ergebnis eine ganze Zahl. Wenn wir vom Typ `Integer` als Eingaben und Ausgaben ausgehen, bedeutet dies: `mod` erwartet als erste Eingabe eine Zahl vom Typ `Integer`, als zweite Eingabe eine Zahl vom Typ `Integer` und liefert als Ausgabe eine Zahl vom Typ `Integer`. Daher ist der Typ von `mod` (wie auch von `div`) der Typ `Integer -> Integer -> Integer` (wenn wir uns auf den `Integer`-Typ beschränken). Daher können wir schreiben:

```
mod :: Integer -> Integer -> Integer
div :: Integer -> Integer -> Integer
```

Fragt man den Interpreter nach den Typen von `mod` und `div` so erhält man etwas allgemeinere Typen:

```
Prelude> :type mod ⏎
mod :: (Integral a) => a -> a -> a
Prelude> :type div ⏎
div :: (Integral a) => a -> a -> a
```

Dies liegt daran, dass `mod` und `div` auch für andere ganze Zahlen verwendet werden können. Links vom `=>` steht hier wieder eine sogenannte Typklassenbeschränkung. Rechts vom `=>` steht der eigentliche Typ, der jedoch die Typklassenbeschränkung einhalten muss. Man kann dies so verstehen: `mod` hat den Typ `a -> a -> a` für alle Typen `a` die `Integral`-Typen sind (dazu gehören u.a. `Integer` und `Int`).

Ähnliches gilt für den Gleichheitstest `(==)`: Er kann für jeden Typ verwendet werden, der zur Typklasse `Eq` gehört:

```
Prelude> :type (==) ⏎
(==) :: (Eq a) => a -> a -> Bool
```

Wir fragen die Typen einiger weiterer bereits eingeführter Funktionen und Operatoren im GHCi ab:

```
Prelude> :type (==) ⏎
(==) :: (Eq a) => a -> a -> Bool
Prelude> :type (<) ⏎
(<) :: (Ord a) => a -> a -> Bool
Prelude> :type (<=) ⏎
(<=) :: (Ord a) => a -> a -> Bool
Prelude> :type (+) ⏎
(+) :: (Num a) => a -> a -> a
Prelude> :type (-) ⏎
(-) :: (Num a) => a -> a -> a
```

Bei einer Anwendung einer Funktion auf Argumente müssen die Typen der Funktion stets zu den Typen der Argumente passen. Wendet man z.B. die Funktion `not` auf ein Zeichen (vom Typ `Char`) an, so passt der Typ nicht: `not` hat den Typ `Bool -> Bool` und erwartet daher einen booleschen Ausdruck. Der GHCi bemerkt dies sofort und produziert einen *Typfehler*:

```
Prelude> not 'C' ⏎

<interactive>:2:5:
    Couldn't match expected type ‘Bool’ with actual type ‘Char’
    In the first argument of ‘not’, namely 'C'
    In the expression: not 'C'
    In an equation for ‘it’: it = not 'C'
```

Sind Typklassen im Spiel (z.B. wenn man Zahlen verwendet deren Typ noch nicht sicher ist), so sind die Fehlermeldungen manchmal etwas merkwürdig. Betrachte das folgende Beispiel:

```
Prelude> not 5 ⏎

<interactive>:2:5:
    No instance for (Num Bool) arising from the literal ‘5’
    In the first argument of ‘not’, namely ‘5’
    In the expression: not 5
    In an equation for ‘it’: it = not 5
```

In diesem Fall sagt der Compiler nicht direkt, dass die Zahl nicht zum Typ `Bool` passt, sondern er bemängelt, dass Boolesche Werte nicht der Klasse `Num` angehören, d.h. er versucht nachzuschauen, ob man die Zahl `5` nicht als Booleschen Wert interpretieren kann. Da das nicht gelingt (`Bool` gehört nicht zur Klasse `Num`, da Boolesche Werte keine Zahlen sind), erscheint die Fehlermeldung.

8.4. Einfache Funktionen definieren

Bisher haben wir vordefinierte Funktionen betrachtet, nun werden wir Funktionen selbst definieren. Z.B. kann man eine Funktion definieren, die jede Zahl verdoppelt als:

```
verdopple x = x + x
```

Funktions-definition

Die Syntax für Funktionsdefinitionen in Haskell sieht folgendermaßen aus:

$$\textit{funktion_Name} \quad par_1 \quad \ldots \quad par_n \quad \texttt{=} \quad \textit{Haskell_Ausdruck}$$

Wobei *funktion_Name* eine Zeichenkette ist, die mit einem Kleinbuchstaben oder einem Unterstrich beginnt und den Namen der Funktion darstellt, unter dem sie aufgerufen werden kann. $par_1 \ldots par_n$ sind die *formalen Parameter* der Funktion, in der Regel stehen hier verschiedene Variablen, beispielsweise `x,y,z`. Rechts vom Gleichheitszeichen folgt ein beliebiger Haskell-Ausdruck, der die Funktion definiert und bestimmt welcher Wert berechnet wird, hier dürfen die Parameter $par_1 \ldots par_n$ verwendet werden.

Man darf dem Quelltext auch den Typ der Funktion hinzufügen[2]. Beschränken wir uns auf `Integer`-Zahlen, so nimmt `verdopple` als Eingabe eine `Integer`-Zahl und liefert als Ausgabe ebenfalls eine `Integer`-Zahl. Daher ist der Typ von `verdopple` der Typ `Integer -> Integer`. Mit Typangabe erhält man den Quelltext:

```
verdopple :: Integer -> Integer
verdopple x = x + x
```

Schreibt man die Funktion in eine Datei und lädt sie anschließend in den GHCi, so kann man sie ausgiebig ausprobieren:

```
Prelude> :load programme/einfacheFunktionen.hs ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)
Ok, modules loaded: Main.
*Main> verdopple 5 ⏎
10
*Main> verdopple 100 ⏎
200
*Main> verdopple (verdopple (2*3) + verdopple (6+9)) ⏎
84
```

Haskell stellt `if-then-else`-Ausdrücke zur Verfügung. Diese werden zur *Fallunterscheidung* bzw. *Verzweigung* verwendet: Anhand des Wahrheitswerts einer Bedingung wird bestimmt, welchen Wert ein Ausdruck haben soll. Die Syntax ist

`if-then-else`

$$\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2.$$

Hierbei muss b ein Ausdruck vom Typ `Bool` sein, und die Typen der Ausdrücke e_1 und e_2 müssen identisch sein. Die Bedeutung eines solchen `if-then-else`-Ausdrucks ist: *Wenn* b wahr ist (zu `True` auswertet), *dann* ist der Wert des gesamten Ausdrucks e_1, *anderenfalls* (b wertet zu `False` aus) ist der Wert des gesamten Ausdrucks e_2. D.h. je nach Wahrheitswert von b „verzweigt“ die Funktion zu e_1 bzw. zu e_2[3].

Wir betrachten einige Beispiele zu `if-then-else`:

- `if 4+5 > 7 then 100 else 1000` ist gleich zu `100`, da $4+5 = 9$ und 9 größer als 7 ist.
- `if False then "Hallo" else "Hallihallo"` ist gleich zum String `"Hallihallo"`, da die Bedingung falsch ist.

[2] Man muss dies i.A. jedoch nicht tun, da der GHCi die Typen auch selbst berechnen kann!

[3] Solche Fallunterscheidungen kennt man auch von imperativen Programmiersprachen, wobei sie dort jedoch eine etwas andere Bedeutung haben, da in Haskell e_1 und e_2 Ausdrücke sind und keine Befehle. Insgesamt ist in Haskell `if` b `then` e_1 `else` e_2 wieder ein Ausdruck. Aus diesem Grund gibt es in Haskell (im Gegensatz zu vielen imperativen Programmiersprachen) kein `if-then`-Konstrukt, denn dann wäre der Wert von `if` b `then` e für falsches b nicht definiert!

- `if (if 2 == 1 then False else True) then 0 else 1"` ist gleich zu 0, da der innere `if-then-else`-Ausdruck gleich zu `True` ist.

Man kann mit `if-then-else`-Ausdrücken z.B. auch eine Funktion definieren, die nur gerade Zahlen verdoppelt:

```
verdoppleGerade :: Integer -> Integer
verdoppleGerade x = if even x then verdopple x else x
```

Die dabei benutzte Funktion `even` ist in Haskell bereits vordefiniert, sie testet, ob eine Zahl gerade ist. Die Funktion `verdoppleGerade` testet nun mit `even`, ob das Argument `x` gerade ist, und mithilfe einer Fallunterscheidung (`if-then-else`) wird entweder die Funktion `verdopple` mit `x` aufgerufen, oder (im `else`-Zweig) einfach `x` selbst zurück gegeben. Machen wir die Probe:

```
*Main> :reload ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
Ok, modules loaded: Main.

*Main> verdoppleGerade 50 ⏎
100
*Main> verdoppleGerade 17 ⏎
17
```

Man kann problemlos mehrere `if-then-else`-Ausdrücke verschachteln und damit komplexere Funktionen implementieren. Z.B. kann man eine Funktion implementieren, die alle Zahlen kleiner als 100 verdoppelt, die Zahlen zwischen 100 und 1000 verdreifacht und andere Zahlen unverändert zurück gibt, als:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
               (if x <= 1000 then 3*x else x)
```

Die Haskell-Syntax beachtet die Einrückung, daher kann man die Klammern um das zweite `if-then-else` auch weglassen:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
               if x <= 1000 then 3*x else x
```

Man muss jedoch darauf achten, dass z.B. die rechte Seite der Funktionsdefinition um mindestens ein Zeichen gegenüber dem Funktionsnamen eingerückt ist. Z.B. ist

```
jenachdem :: Integer -> Integer
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

falsch, da das erste `if` nicht eingerückt ist. Der GHCi meldet in diesem Fall einen Fehler:

```
Prelude> :reload ⏎
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )

programme/einfacheFunktionen.hs:9:1:
    parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

Wir betrachten eine weitere Funktion, die zwei Eingaben erhält, zum Einen einen Booleschen Wert b und zum Anderen eine Zahl x. Die Funktion soll die Zahl x verdoppeln, wenn b wahr ist, und die Zahl x verdreifachen, wenn b falsch ist:

```
verdoppeln_oder_verdreifachen :: Bool -> Integer -> Integer
verdoppeln_oder_verdreifachen b x =
  if b then 2*x else 3*x
```

Wir testen die Funktion im GHCi:

```
*Main> verdoppeln_oder_verdreifachen True 10 ⏎
20
*Main> verdoppeln_oder_verdreifachen False 10 ⏎
30
```

Wir können die Funktion `verdoppele` von vorher nun auch unter Verwendendung von `verdoppeln_oder_verdreifachen` definieren:

```
verdoppeln2 :: Integer -> Integer
verdoppeln2 x = verdoppeln_oder_verdreifachen True x
```

Man kann sogar das Argument `x` in diesem Fall weglassen, da wir ja partiell anwenden dürfen, d.h. eine noch elegantere Definition ist:

```
verdoppeln3 :: Integer -> Integer
verdoppeln3 = verdoppeln_oder_verdreifachen True
```

Hier sei nochmals auf die Typen hingewiesen: Den Typ `Bool -> Integer -> Integer` von `verdoppeln_oder_verdreifachen` kann man auch geklammert schreiben als `Bool -> (Integer -> Integer)`. Genau das haben wir bei `verdoppeln3` ausgenutzt, denn das Ergebnis von `verdoppeln3` ist der Rückgabewert der partiellen Anwendung von `verdoppeln_oder_verdreifachen` auf `True` und daher eine *Funktion* vom Typ `Integer -> Integer`. D.h. in Haskell gibt es keine Bedingung, dass eine Funktion als Ergebnistyp einen Basistyp haben muss, es ist (wie z.B. bei `verdoppeln3`) durchaus erlaubt auch Funktionen als Ergebnis zurück zu geben.

Etwas analoges gilt für die Argumente einer Funktion: Bisher waren dies stets Basistypen, es ist aber auch an dieser Stelle erlaubt, *Funktionen* selbst zu übergeben. Betrachte zum Beispiel die folgende Funktion:

Funktionen als Parameter

```
wende_an_und_addiere f x y = (f x) + (f y)
```

Die Funktion `wende_an_und_addiere` erhält drei Eingaben: Eine Funktion `f` und zwei Zahlen `x` und `y`. Die Ausgabe berechnet sie wie folgt: Sie wendet die übergebene Funktion `f` sowohl einmal auf `x` als auch einmal auf `y` an und addiert schließlich die Ergebnisse. Z.B. kann man für `f` die Funktion `verdopple` oder die Funktion `jenachdem` übergeben:

```
*Main> wende_an_und_addiere verdopple 10 20 ⏎
60
*Main> wende_an_und_addiere jenachdem 150 3000 ⏎
3450
```

Wir können allerdings nicht ohne Weiteres jede beliebige Funktion an `wende_an_und_addiere` übergeben, denn die Typen müssen passen. Sehen wir uns den Typ von `wende_an_und_addiere` an:

`wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer ->Integer`

Wenn man an den äußeren `->`-Pfeilen zerlegt, kann man die einzelnen Typen den Argumenten und dem Ergebnis zuordnen:

$$\texttt{wende_an_und_addiere} :: \underbrace{\texttt{(Integer -> Integer)}}_{\text{Typ von }\texttt{f}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von }\texttt{x}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von }\texttt{y}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\substack{\text{Typ des}\\ \text{Ergebnisses}}}$$

D.h. nur Funktionen, die einen Ausdruck vom Typ `Integer` erwarten und eine Zahl liefern können an `wende_an_und_addiere` übergeben werden. Beachte auch, dass man die Klammern im Typ

`(Integer -> Integer) -> Integer -> Integer -> Integer`

nicht weglassen darf, denn der Typ

`Integer -> Integer -> Integer -> Integer -> Integer`

ist implizit rechtsgeklammert, und entspricht daher dem Typ

`Integer -> (Integer -> (Integer -> (Integer -> Integer)))`.

Funktion höherer Ordnung

Funktionen, die andere Funktionen als Argumente akzeptieren oder zurückgeben, bezeichnet man als *Funktionen höherer Ordnung*.

Wir betrachten noch eine Funktion, die zwei Eingaben erhält und den String `"Die Eingaben sind gleich!"` als Ergebnis liefert, wenn die Eingaben gleich sind. Man könnte versucht sein, die Funktion so zu implementieren:

```
sonicht x x = "Die Eingaben sind gleich!"
```

Dies ist allerdings keine gültige Definition in Haskell, der GHCi beanstandet dies auch sofort:

```
einfacheFunktionen.hs:31:9:
    Conflicting definitions for ‘x’
    Bound at: einfacheFunktionen.hs:31:9
              einfacheFunktionen.hs:31:11
    In an equation for ‘sonicht’
Failed, modules loaded: none.
```

Die Parameter einer Funktion müssen nämlich alle verschieden sein, d.h. für `sonicht` darf man nicht zweimal das gleiche `x` für die Definition verwenden. Richtig ist

```
vergleiche x y = if x == y then "Die Eingaben sind gleich!" else ""
```

Die Funktion `vergleiche` gibt bei Ungleichheit den leeren String `""` zurück. Der Typ von `vergleiche` ist `vergleiche :: (Eq a) => a -> a -> String`, d.h. er beinhaltet eine Typklassenbeschränkung (aufgrund der Verwendendung von `==`): Für jeden Typ `a`, der zur Klasse `Eq` gehört, hat `vergleiche` den Typ `a -> a -> String`.

Wir betrachten als weiteres Beispiel die Funktion `zweimal_anwenden`:

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Die Funktion erwartet eine Funktion und ein Argument und wendet die übergebene Funktion zweimal auf das Argument an. Da hier eine fast beliebige Funktion verwendet werden kann, enthält der Typ von `zweimal_anwenden` Typvariablen (das `a`). Für diese kann man einen beliebigen Typen einsetzen, allerdings muss für jedes `a` der selbe Typ eingesetzt werden. Ist der Typ einer Funktion (wie der Typ von `f` in `zweimal_anwenden`) von der Form `a -> a`, so handelt es sich um eine Funktion, deren Ergebnistyp und deren Argumenttyp identisch sein müssen, aber ansonsten nicht weiter beschränkt sind. Wir führen einige Tests durch:

```
 *Main> zweimal_anwenden verdopple 10
40
*Main> zweimal_anwenden (wende_an_und_addiere verdopple 100) 10
640
*Main> zweimal_anwenden (vergleiche True) True

<interactive>:2:30:
    Couldn’t match type ‘Bool’ with ‘[Char]’
    Expected type: String
      Actual type: Bool
    In the first argument of ‘vergleiche’, namely ‘True’
    In the first argument of ‘zweimal_anwenden’, namely
      ‘(vergleiche True)’

<interactive>:2:36:
    Couldn’t match type ‘Bool’ with ‘[Char]’
    Expected type: String
      Actual type: Bool
    In the second argument of ‘zweimal_anwenden’, namely ‘True’
    In the expression: zweimal_anwenden (vergleiche True) True
```

Der letzte Test `zweimal_anwenden (vergleiche True) True` geht schief (hat einen Typfehler), da `(vergleiche True)` den Typ `Bool -> String` hat, und daher nicht für den Typ `(a -> a)` eingesetzt werden kann.

Da in den Typen von Ausdrücken und Funktion Typvariablen erlaubt sind, sagt man Haskell hat ein *polymorphes* Typsystem.

Polymorpher Typ

8.5. Rekursion

Der Begriff Rekursion stammt vom lateinischen Wort „recurrere“ was „zurücklaufen“ bedeutet. Die Technik, die sich hinter der Rekursion versteckt, besteht darin, dass eine Funktion definiert wird, indem sie sich in der Definition selbst wieder aufruft. Ein altbekannter Spruch (Witz) zur Rekursion lautet:

„*Wer Rekursion verstehen will, muss Rekursion verstehen.*“

rekursiv

Man nennt eine Funktion (direkt) *rekursiv*, wenn sie sich selbst wieder aufruft, d.h. wenn die Definition von f von der Form

```
f ... =  ...  f  ...
```

ist. Man kann diesen Begriff noch verallgemeinern, da `f` sich nicht unbedingt direkt selbst aufrufen muss, es geht auch indirekt über andere Funktionen, also in der Art:

```
f ... = ... g ...
g ... = ... f ...
```

In diesem Fall ruft `f` die Funktion `g` auf, und `g` ruft wieder `f` auf. Daher sind `f` und `g` *verschränkt rekursiv*.

Nicht jede rekursive Funktion ist sinnvoll, betrachte z.B. die Funktion

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

Die Auswertung der Funktion wird (egal für welchen Wert von `x`) nicht aufhören, da sich `endlos_eins_addieren` stets erneut aufrufen wird. Etwa in der Form:

`endlos_eins_addieren` 1
$\rightarrow$ `endlos_eins_addieren` (1+1)
$\rightarrow$ `endlos_eins_addieren` ((1+1)+1)
$\rightarrow$ `endlos_eins_addieren` (((1+1)+1)+1)
$\rightarrow$ ...

Rekursionsanfang und Rekursionsschritt

D.h. man sollte im Allgemeinen rekursive Funktionen so implementieren, dass irgendwann *sicher*[4] ein Ende erreicht wird und kein rekursiver Aufruf mehr erfolgt. Dieses Ende nennt man auch *Rekursionsanfang*, den rekursiven Aufruf nennt man auch *Rekursionsschritt*. Wir betrachten als erste sinnvolle rekursive Funktion die Funktion `erste_rekursive_Funktion`:

```
erste_rekursive_Funktion x = if x <= 0 then 0 else
                                x+(erste_rekursive_Funktion (x-1))
```

Der Rekursionsanfang wird gerade durch den `then`-Zweig des `if-then-else`-Ausdrucks abgedeckt: Für alle Zahlen die kleiner oder gleich 0 sind, findet kein rekursiver Aufruf statt, sondern es wird direkt 0 als Ergebnis geliefert. Der Rekursionsschritt besteht darin `erste_rekursive_Funktion` mit `x` erniedrigt um `1` aufzurufen und zu diesem Ergebnis den Wert von `x` hinzuzählen. Man

[4]Tatsächlich ist es manchmal sehr schwer, diese Sicherheit zu garantieren. Z.B. ist bis heute unbekannt, ob die Funktion `f x = if n == 1 then 1 else (if even x then f (x ‘div‘ 2) else f (3*x+1))` für jede natürliche Zahl terminiert, d.h. den Rekursionsanfang findet. Dies ist das sogenannte Collatz-Problem.

kann leicht überprüfen, dass für jede ganze Zahl x der Aufruf `erste_rekursive_Funktion` x irgendwann beim Rekursionsanfang landen muss, daher *terminiert* jede Anwendung von `erste_rekursive_Funktion` auf eine ganze Zahl.

Es bleibt zu klären, was `erste_rekursive_Funktion` berechnet. Wir können zunächst einmal im GHCi testen:

```
*Main> erste_rekursive_Funktion 5
15
*Main> erste_rekursive_Funktion 10
55
*Main> erste_rekursive_Funktion 11
66
*Main> erste_rekursive_Funktion 12
78
*Main> erste_rekursive_Funktion 100
5050
*Main> erste_rekursive_Funktion 1
1
*Main> erste_rekursive_Funktion (-30)
0
```

Man stellt schnell fest, dass für negative Zahlen stets der Wert 0 als Ergebnis herauskommt. Wir betrachten den Aufruf von `erste_rekursive_Funktion 5`. Das Ergebnis `15` kann man per Hand ausrechnen:

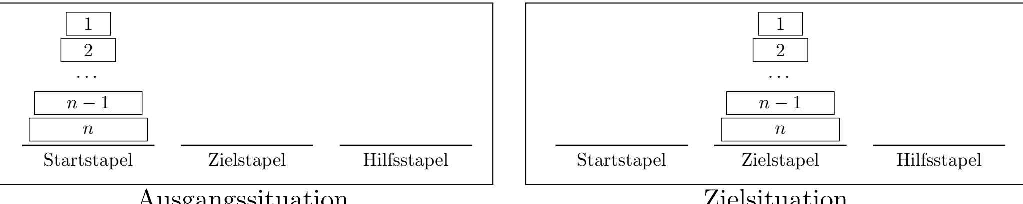
```
  erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15
```

Durch längeres Anschauen der Definition erkennt man, dass für jede nicht-negative Zahl x gerade $x+(x-1)+(x-2)+\ldots+0$ (oder als mathematische Summenformel $\sum_{i=0}^{x} i$) berechnet wird. Man kann dies auch beweisen, was wir allerdings an dieser Stelle nicht tun wollen. Es ist erwähnswert, dass obige Beispielauswertung durch Hinschauen geschehen ist, in der Veranstaltung „Grundlagen der Programmierung 2“ wird dieses Berechnen viel genauer durchgeführt und erklärt.

Der Trick beim Entwurf einer rekursiven Funktion besteht darin, dass man eine Problemstellung *zerlegt*: Der Rekursionsanfang ist der einfache Fall, für den man das Problem direkt lösen kann. Für den Rekursionsschritt löst man nur einen (meist ganz kleinen) Teil des Problems (im obigen Beispiel war das gerade das Hinzuaddieren von `x`) und überlässt den Rest der Problemlösung der Rekursion.

Das sogenannte „Türme von Hanoi“-Spiel lässt sich sehr einfach durch Rekursion lösen. Die Anfangssituation besteht aus einem Turm von n immer kleiner werdenden Scheiben, der auf dem Startfeld steht. Es gibt zwei weitere Felder: Das Zielfeld und das Hilfsfeld. Ein gültiger Zug besteht darin, die oberste Scheibe eines Turmes auf einen anderen Stapel zu legen, wobei stets nur kleinere Scheiben auf größere Scheiben gelegt werden dürfen.

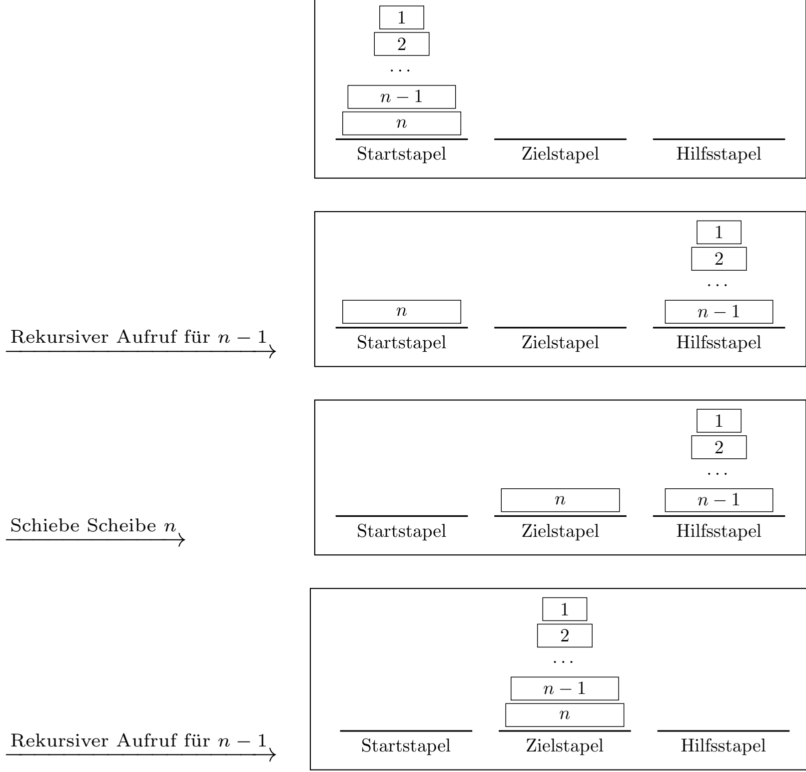
Anschaulich können die Start- und die Zielsituation dargestellt werden durch die folgenden Abbildungen:



Anstatt nun durch probieren alle möglichen Züge usw. zu berechnen, kann das Problem mithilfe von Rekursion sehr einfach gelöst werden:

1. Schiebe mittels Rekursion den Turm der ersten $n - 1$ oberen Scheiben vom Startstapel auf den Hilfsstapel
2. Schiebe die n. Scheibe vom Startstapel auf den Zielstapel
3. Schiebe mittels Rekursion den Turm der $n - 1$ Scheiben vom Hilfsstapel auf den Zielstapel.

Oder mit Bildern ausgedrückt:



Beachte, dass der Rekursionsanfang gerade der Fall $n = 1$ ist, da dann nur eine Scheibe verschoben wird. Die Rekursion terminiert, da bei jedem Aufruf die Probleminstanz (der Turm) um 1 verringert wird. Wir betrachten an dieser Stelle nicht die Implementierung in Haskell und verschieben diese auf einen späteren Abschnitt.

Als weiteres Beispiel betrachten wir die Funktion `n_mal_verdoppeln`. Die Funktion soll eine Zahl `x` genau `n` mal verdoppeln. Da wir die Funktion `verdoppeln` bereits definiert haben, genügt es diese Funktion `n`-mal aufzurufen. Da wir den Wert von `n` jedoch nicht kennen, verwenden wir Rekursion: Wenn `n` die Zahl 0 ist, dann verdoppeln wir gar nicht, sondern geben `x` direkt zurück. Das ist der Rekursionsanfang. Wenn `n` größer als `0` ist, dann verdoppeln wir `x` einmal und müssen anschließend das Ergebnis noch `n-1`-mal verdoppeln, das erledigt der Rekursionsschritt. In Haskell programmiert, ergibt dies die folgende Implementierung:

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n = if n == 0 then x
                       else n_mal_verdoppeln (verdopple x) (n-1)
```

In Haskell kann man auch mehrere Definitionsgleichungen für eine Funktion angeben, diese werden von oben nach unten abgearbeitet, so dass die erste „passende“ Gleichung gewählt wird: Man darf anstelle einer Variablen nämlich auch ein sogenanntes Pattern (Muster) verwenden. Bei Zahlentypen kann man einfach eine Zahl hinschreiben. Damit können wir uns das `if-then-else` sparen:

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x 0 = x
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
```

Der Effekt ist der gleiche: Ist `n` = `0`, so passt die erste Zeile und wird daher verwendet, ist `n` ungleich 0, so passt die erste Zeile nicht, und die zweite Zeile der Definition wird verwendet. Wir testen beide Funktion im Interpreter:

```
*Main> n_mal_verdoppeln2 10 2
40
*Main> n_mal_verdoppeln2 10 3
80
*Main> n_mal_verdoppeln2 10 10
10240
*Main> n_mal_verdoppeln 10 2
40
*Main> n_mal_verdoppeln 2 8
512
```

Gibt man für `n` jedoch eine negative Zahl ein, so hält der Interpreter nicht an. Den Fall hatten wir übersehen. Wir könnten einfach `x` zurückgeben, besser ist es jedoch, eine *Fehlermeldung* zu erzeugen. Hierfür gibt es in Haskell die eingebaute Funktion `error`. Sie erwartet als Argument einen String, der beim Auftreten des Fehlers ausgegeben wird. Die Implementierung mit Fehlerabfang ist:

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x 0 = x
n_mal_verdoppeln3 x n =
  if n < 0 then
    error "in n_mal_verdoppeln3: negatives Verdoppeln ist verboten"
  else
    n_mal_verdoppeln3 (verdopple x) (n-1)
```

Ein Testaufruf im GHCi ist:

```
*Main> n_mal_verdoppeln3 10 (-10)
*** Exception: in n_mal_verdoppeln3: negatives Verdoppeln ist verboten
```

Eine weitere Möglichkeit, die Funktion `n_mal_verdoppeln` zu definieren, ergibt sich durch die Verwendung von sogenannten *Guards* (Wächtern zu deutsch), die ebenfalls helfen ein `if-then-else` zu sparen:

Guard

```
n_mal_verdoppeln4 x 0 = x
n_mal_verdoppeln4 x n
  | n < 0     = error "negatives Verdoppeln ist verboten"
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

Hinter dem Guard, der syntaktisch durch `|` repräsentiert wird, steht ein Ausdruck vom Typ `Bool` (beispielsweise `n < 0`). Beim Aufruf der Funktion werden diese Ausdrücke von oben nach unten ausgewertet, liefert einer den Wert `True`, dann wird die entsprechende rechte Seite als Funktionsdefinition verwendet. `otherwise` ist ein speziell vordefinierter Ausdruck

```
otherwise = True
```

der immer zu `True` auswertet. D.h. falls keines der vorhergehenden Prädikate `True` liefert, wird die auf `otherwise` folgende rechte Seite ausgewertet. Die Fallunterscheidung über n kann auch komplett über Guards geschehen:

```
n_mal_verdoppeln4 x n
  | n < 0     = error "negatives Verdoppeln ist verboten"
  | n == 0    = x
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

Dabei unterscheidet sich die Funktionalität von `n_mal_verdoppeln4` nicht von `n_mal_verdoppeln3`, lediglich die Schreibweise ist verschieden.

Mit Rekursion kann man auch Alltagsfragen lösen. Wir betrachten dazu die folgende Aufgabe:

> In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n?

Wir müssen zur Lösung eine rekursive Funktion aufstellen, welche die Anzahl an Rehen nach n Jahren berechnet, d.h. die Funktion erhält n als Eingabe und berechnet die Anzahl. Der Rekursionsanfang ist einfach: Im ersten Jahr gibt es 10 Rehe. Für den Rekursionsschritt denken wir uns das Jahr n und müssen die Anzahl an Rehen aufgrund der Anzahl im Jahr $n-1$ berechnen. Sei k die Anzahl der Rehe am 1.1. im Jahr $n-1$. Dann gibt es am 1.1. im Jahr n genau $3*k-17$ Rehe, wenn $n-1$ kein zweites Jahr war, und $\frac{3*k-17}{2}$ Rehe, wenn $n-1$ ein zweites Jahr war. Festellen, ob $n-1$ ein zweites Jahr war, können wir, indem wir prüfen, ob $n-1$ eine gerade Zahl ist.

Mit diesen Überlegungen können wir die Funktion implementieren:

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)  then ((3*anzahlRehe (n-1))-17) `div` 2
               else 3*(anzahlRehe (n-1))-17
```

Testen zeigt, dass die Anzahl an Rehen sehr schnell wächst:

```
*Main> anzahlRehe 1
10
*Main> anzahlRehe 2
13
*Main> anzahlRehe 3
11
*Main> anzahlRehe 4
16
*Main> anzahlRehe 5
15
*Main> anzahlRehe 6
28
*Main> anzahlRehe 7
33
*Main> anzahlRehe 8
82
*Main> anzahlRehe 9
114
*Main> anzahlRehe 10
325
*Main> anzahlRehe 50
3626347914090925
```

In Haskell gibt es `let`-Ausdrücke, mit diesen kann man lokal (im Rumpf einer Funktion) den Wert eines Ausdrucks an einen Variablennamen binden (der Wert ist unveränderlich!). Dadurch kann man es sich z.B. ersparen gleiche Ausdrücke immer wieder hinzuschreiben. Die Syntax ist

`let`-Ausdruck

`let`	*Variable1*	=	*Ausdruck1*
	Variable2	=	*Ausdruck2*
	...		
	VariableN	=	*AusdruckN*
`in`	Ausdruck		

Hierbei müssen Variablennamen mit einem Kleinbuchstaben oder mit einem Unterstrich beginnen, und auf die gleiche Einrückung aller Definitionen ist zu achten.

Z.B. kann man `anzahlRehe` dadurch eleganter formulieren:

```
anzahlRehe2 1 = 10
anzahlRehe2 n = let k = (3*anzahlRehe2 (n-1))-17
                in  if even (n-1) then k `div` 2
                    else k
```

8.6. Listen

Eine Liste ist eine Folge von Elementen, z.B. ist `[True, False, False, True, True]` eine Liste von Wahrheitswerten und `[1,2,3,4,5,6]` eine Liste von Zahlen. In Haskell sind nur *homogene* Listen erlaubt, d.h. die Listenelemente ein und derselben Liste müssen alle den gleichen Typ besitzen. Z.B. ist die Liste `[True,'a',False,2]` in Haskell nicht erlaubt, da in ihr Elemente vom Typ `Bool`, vom Typ `Char`, und Zahlen vorkommen. Der GHCi bemängelt dies dementsprechend sofort:

```
Prelude> [True,'a',False,2]

<interactive>:1:6:
  Couldn't match expected type `Bool' against inferred type `Char'
  In the expression: 'a'
  In the expression: [True, 'a', False, 2]
  In the definition of `it': it = [True, 'a', False, ....]
```

Der Typ einer Liste ist von der Form `[a]`, wobei `a` der Typ der *Listenelemente* ist. Einige Beispiele im Interpreter:

```
Prelude> :type [True,False]
[True,False] :: [Bool]
Prelude> :type ['A','B']
['A','B'] :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: (Num t) => [t]
```

Für die Liste `[1,2,3]` ist der Typ der Zahlen noch nicht festgelegt, daher die Typklassenbeschränkung für `Num`. Man kann in Haskell beliebige Dinge (gleichen Typs) in eine Liste stecken, nicht nur Basistypen. Für den Typ bedeutet dies: In `[a]` kann man für `a` einen beliebigen Typ einsetzen. Z.B. kann man eine Liste von Funktionen (alle vom Typ `Integer -> Integer`) erstellen:

```
*Main> :type [verdopple, verdoppleGerade, jenachdem]
[verdopple, verdoppleGerade, jenachdem] :: [Integer -> Integer]
```

Man kann sich diese Liste allerdings nicht anzeigen lassen, da der GHCi nicht weiß, wie er Funktionen anzeigen soll. Man erhält dann eine Fehlermeldung der Form:

```
*Main> [verdopple, verdoppleGerade, jenachdem]

<interactive>:1:0:
    No instance for (Show (Integer -> Integer))
      arising from a use of `print' at <interactive>:1:0-38
    Possible fix:
      add an instance declaration for (Show (Integer -> Integer))
    In a stmt of a 'do' expression: print it
```

Eine andere komplizierte Liste, ist eine Liste, die als Elemente wiederum Listen erhält, z.B. `[[True,False], [False,True,True], [True,True]]`. Der Typ dieser Liste ist `[[Bool]]`: Für `a` in `[a]` wurde einfach `[Bool]` eingesetzt.

8.6.1. Listen konstruieren

Wir haben bereits eine Möglichkeit gesehen, Listen zu erstellen: Man trennt die Elemente mit Kommas ab und verpackt die Elemente durch eckige Klammern. Dies ist jedoch nur sogenannter

syntaktischer Zucker. In Wahrheit sind Listen *rekursiv* definiert, und man kann sie auch rekursiv erstellen. Der „Rekursionsanfang" ist die leere Liste, die keine Elemente enthält. Diese wird in Haskell durch `[]` dargestellt und als „Nil"[5] ausgesprochen. Der „Rekursionsschritt" besteht darin aus einer Liste mit $n-1$ Elementen eine Liste mit n Elementen zu konstruieren, indem ein neues Element vorne an die Liste der $n-1$ Elemente angehängt wird. Hierzu dient der *Listenkonstruktor* : (ausgesprochen „Cons"[6]). Wenn xs eine Liste mit $n-1$ Elementen ist und x ein neues Element ist, dann ist `x:xs` die Liste mit n Elementen, die entsteht, indem man x vorne an xs anhängt. In $x : xs$ sagt man auch, dass x der *Kopf* (engl. head) und xs der *Schwanz* (engl. tail) der Liste ist. Wir konstruieren die Liste `[True,False,True]` rekursiv:

Nil

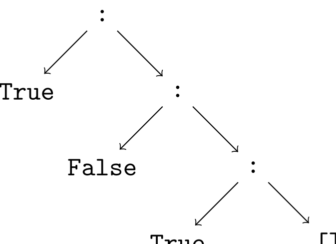
Cons

- `[]` ist die leere Liste
- `True:[]` ist die Liste, die `True` enthält
- `False:(True:[])` ist die Liste, die erst `False` und dann `True` enthält
- Daher lässt sich die gesuchte Liste als `True:(False:(True:[]))` konstruieren.

Tatsächlich kann man diese Liste so im GHCi eingeben:

```
*Main> True:(False:(True:[]))
[True,False,True]
```

Man kann sich eine Liste daher auch als Baum vorstellen, dessen innere Knoten mit : markiert sind, wobei das linke Kind das Listenelement ist und das rechte Kind die Restliste ist. Z.B. für `True:(False:(True:[]))`:



Natürlich kann man : und `[]` auch in Funktionsdefinitionen verwenden. Daher kann man relativ einfach eine rekursive Funktion implementieren, die eine Zahl n als Eingabe erhält und eine Liste erstellt, welche die Zahlen von n bis 1 in dieser Reihenfolge enthält:

```
nbis1 :: Integer -> [Integer]
nbis1 0 = []
nbis1 n = n:(nbis1 (n-1))
```

Der Rekursionsanfang ist für $n = 0$ definiert: In diesem Fall wird die leere Liste konstruiert. Für den Rekursionsschritt wird die Liste ab $(n-1)$ rekursiv konstruiert und anschließend die Zahl n mittels : vorne an die Liste angehängt. Wir testen `nbis1` im GHCi:

[5]Diese Bezeichnung stamm vom lateinischen Wort „nihil" für „Nichts".
[6]Kurzform von „Constructor", also einem Konstruktor.

```
*Main> nbis1 0
[]
*Main> nbis1 1
[1]
*Main> nbis1 10
[10,9,8,7,6,5,4,3,2,1]
*Main> nbis1 100
[100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,
 78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,57,
 56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,
 34,33,32, 31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
 12,11,10,9,8,7,6,5,4,3,2,1]
```

8.6.2. Listen zerlegen

Oft will man auf die Elemente einer Liste zugreifen. In Haskell sind die Funktion `head` und `tail` dafür bereits vordefiniert:

- `head :: [a] -> a` liefert das erste Element einer nicht-leeren Liste
- `tail :: [a] -> [a]` liefert den Schwanz einer nicht-leeren Liste.

Beachte, dass `head` und `tail` für beliebige Listen verwendet werden können, da ihr Typ *polymorph* ist: Er enthält Typvariablen an der Position für den Elementtyp. Einige Aufrufe von `head` und `tail` im GHCi:

```
*Main> head [1,2]
1
*Main> tail [1,2,3]
[2,3]
*Main> head []
*** Exception: Prelude.head: empty list
```

Die in Haskell vordefinierte Funktion `null :: [a] -> Bool` testet, ob eine Liste leer ist. Mit `head`, `tail` und `null` kann man beispielsweise eine Funktion definieren, die das letzte Element einer Liste extrahiert:

```
letztesElement :: [a] -> a
letztesElement xs = if null xs then
                       error "Liste ist leer"
                    else
                      if null (tail xs) then head xs
                      else letztesElement (tail xs)
```

Für eine Liste mit mehr als einem Element ruft sich `letztesElement` rekursiv mit dem Schwanz der Liste auf. Enthält die Liste nur ein Element (der Test hierfür ist `null (tail xs)`), so wird das erste Element dieser Liste als Ergebnis zurück geliefert (dies ist der Rekursionsanfang). Der Fehlerfall, dass die Liste gar keine Elemente enthält, wird direkt am Anfang abgefangen, und eine Fehlermeldung wird generiert. Der Typ der Funktion `letztesElement` ist `[a] -> a`, da sie eine Liste erhält (hierbei ist der konkrete Typ der Elemente egal) und ein Listenelement liefert. Wir testen `letztesElement`:

```
Main> letztesElement [True,False,True]
True
*Main> letztesElement [1,2,3]
3
*Main> letztesElement (nbis1 1000)
1
*Main> letztesElement [[1,2,3], [4,5,6], [7,8,9]]
[7,8,9]
```

Die Programmierung mit `head`, `tail` und `null` ist allerdings nicht wirklich elegant. Haskell bietet hierfür wesentlich schönere Möglichkeiten durch Verwendung sogenannter *Pattern*. Man kann bei einer Funktionsdefinition (ähnlich wie bei den Funktionen auf Zahlen) ein Muster angeben, anhand dessen die Listen zerlegt werden sollen. Hierfür verwendet man die Konstruktoren `[]` und `:` innerhalb der Parameter der Funktion. Wir betrachten als Beispiel die Implementierung von `head`:

Pattern

```
eigenesHead []     = error "empty list"
eigenesHead (x:xs) = x
```

Die Definition von `eigenesHead` ist hierbei so zu interpretieren: Wenn die Eingabe eine Liste ist, die zu dem Muster `(x:xs)` passt (die Liste daher mindestens ein Element hat), dann gebe den zu `x` passenden Teil zurück und wenn die Liste zum Muster `[]` passt (die Liste daher leer ist) dann generiere eine Fehlermeldung. Die Fälle werden bei der Auswertung von oben nach unten geprüft. Analog ist `tail` definiert als.

```
eigenesTail []     = error "empty list"
eigenesTail (x:xs) = xs
```

Eine mögliche Definition für die Funktion `null` ist:

```
eigenesNull []     = True
eigenesNull (x:xs) = False
```

Da die Muster von oben nach unten abgearbeitet werden, kann man alternativ auch definieren

```
eigenesNull2 []     = True
eigenesNull2 xs     = False
```

In diesem Fall passt das zweite Muster (es besteht nur aus der Variablen `xs`) für jede Liste. Trotzdem wird für den Fall der leeren Liste `True` zurückgeliefert, da die Muster von oben nach unten geprüft werden. Falsch wird die Definition, wenn man die beiden Fälle in falscher Reihenfolge definiert:

```
falschesNull xs = False
falschesNull [] = True
```

Da das erste Muster immer passt, wird die Funktion `falschesNull` für jede Liste `False` liefern. Der GHCi ist so schlau, dies zu bemerken und liefert beim Laden der Datei eine Warnung:

```
    Warning: Pattern match(es) are overlapped
             In the definition of ‘falschesNull’: falschesNull [] = ...
Ok, modules loaded: Main.
*Main> falschesNull [1]
False
*Main> falschesNull []
False
*Main> eigenesNull []
True
*Main>
```

Kehren wir zurück zur Definition von `letztesElement`: Durch Pattern können wir eine elegantere Definition angeben:

```
letztesElement2 []     = error "leere Liste"
letztesElement2 (x:[]) = x
letztesElement2 (x:xs) = letztesElement2 xs
```

Beachte, dass das Muster `(x:[])` in der zweiten Zeile ausschließlich für einelementige Listen passt.

8.6.3. Einige vordefinierte Listenfunktionen

In diesem Abschnitt führen wir einige Listenfunktionen auf, die in Haskell bereits vordefiniert sind, und verwendet werden können:

- `length :: [a] -> Int` berechnet die Länge einer Liste (d.h. die Anzahl ihrer Elemente).
- `take :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert die Liste der ersten k Elemente von xs
- `drop :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert xs ohne die der ersten k Elemente.
- `(++) :: [a] -> [a] -> [a]` erwartet zwei Listen und hängt diese aneinander zu einer Liste, kann infix in der Form `xs ++ ys` verwendet werden. Man nennt diese Funktion auch „append“.
- `concat :: [[a]] -> [a]` erwartet eine Liste von Listen und hängt die inneren Listen alle zusammen. Z.B. gilt `concat [xs,ys]` ist gleich zu `xs ++ ys`.
- `reverse :: [a] -> [a]` dreht die Reihenfolge der Elemente einer Liste um.

8.6.4. Nochmal Strings

Wir haben Zeichenketten (Strings) vorher schon eingeführt. Tatsächlich ist die Syntax `"Hallo Welt"` nur syntaktischer Zucker: Zeichenketten werden in Haskell als Listen von Zeichen intern dargestellt, d.h. unser Beispielstring ist äquivalent zur Liste `['H','a','l','l','o',' ','W','e','l','t']`, die man auch mit `[]` und `:` darstellen kann als `'H':('a':('l':('l':('o':(' '(:'(W':('e':(l':('t':[])))))))))`. Alle Funktionen die auf Listen arbeiten, kann man daher auch für Strings verwenden, z.B.

```
*Main> head "Hallo Welt"
'H'
*Main> tail "Hallo Welt"
"allo Welt"
*Main> null "Hallo Welt"
False
*Main> null ""
True
*Main> letztesElement "Hallo Welt"
't'
```

Es gibt allerdings spezielle Funktionen, die nur für Strings funktionieren, da sie die einzelnen Zeichen der Strings „anfassen". Z.B.

- `words :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Worten*
- `unwords :: [String] -> String`: Macht aus einer Liste von Worten einen einzelnen String.
- `lines :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Zeilen*

Z.B. kann man unter Verwendung von `words` und `length` die Anzahl der Worte eines Texts zählen:

```
anzahlWorte :: String -> Int
anzahlWorte text = length (words text)
```

8.7. Paare und Tupel

Paare stellen in Haskell – neben Listen – eine weitere Möglichkeit dar, um Daten zu strukturieren. Ein Paar wird aus zwei Ausdrücken e_1, e_2 konstruiert, indem sie geschrieben werden als (e_1, e_2). Hat e_1 den Typ T_1 und e_2 den Typ T_2, dann ist der Typ des Paares (T_1, T_2). Der Unterschied zu Listen besteht darin, dass Paare eine feste Stelligkeit haben (nämlich 2) und dass die Typen der beiden Ausdrücke nicht gleich sein müssen. Einige Beispiele für Paare in Haskell sind:

```
Main> :type ("Hallo",True)
("Hallo",True) :: ([Char], Bool)
Main> :type ([1,2,3],'A')
([1,2,3],False) :: (Num t) => ([t], Char)
*Main> :type (letztesElement, "Hallo" ++ "Welt")
(letztesElement, "Hallo" ++ "Welt") :: ([a] -> a, [Char])
```

Der Zugriff auf Elemente eines Paares kann über zwei vordefinierte Funktionen erfolgen:

- `fst :: (a,b) -> a` nimmt ein Paar und liefert das linke Element.
- `snd :: (a,b) -> b` liefert das rechte Element eines Paares.

Wie man am Typ der beiden Funktionen sieht (der Typvariablen `a,b` enthält), sind sie polymorph, d.h. sie können für Paare beliebigen Typs verwendet werden. Beispiele für die Anwendung sind:

```
*Main> fst (1,'A')
1
*Main> snd (1,'A')
'A'
*Main>
```

Bei der Definition von `fst` und `snd` kann man wieder Pattern (Muster) verwenden, um die entsprechenden Elemente auszuwählen:

```
eigenesFst (x,y) = x

eigenesSnd (x,y) = y
```

Hier wird das Pattern `(x,y)` zur Selektion des gewünschten Elements verwendet, wobei `( , )` der Paar-Konstruktor ist, `x` ist eine Variable, die das linke Element eines Paares bezeichnet und `y` eine Variable, die das rechte Element bezeichnet.

Tupel stellen eine Verallgemeinerung von Paaren dar: Ein n-Tupel kann n Elemente verschiedenen Typs aufnehmen. Auch hier ist die Stelligkeit fest: Ein n-Tupel hat Stelligkeit n. Das Erzeugen von n-Tupeln verläuft analog zum Erzeugen von Paaren, einige Beispiele sind:

```
*Main> :set +t
*Main> ('A',True,'B')
('A',True,'B')
it :: (Char, Bool, Char)
*Main> ([1,2,3],(True,'A',False,'B'),'B')
([1,2,3],(True,'A',False,'B'),'B')
it :: ([Integer], (Bool, Char, Bool, Char), Char)
```

Der GHCi kann Tupel mit bis zu 15 Elementen drucken, intern können noch größere Tupel verwendet werden. Zur Auswahl bestimmter Elemente von Tupeln sind keine Funktionen vordefiniert, man kann sie aber leicht mit Hilfe von Pattern selbst implementieren:

```
erstes_aus_vier_tupel (w,x,y,z) = w
-- usw.
viertes_aus_vier_tupel (w,x,y,z) = z
```

8.7.1. Die Türme von Hanoi in Haskell

Wir betrachten nun die Implementierung des „Türme von Hanoi“-Spiels in Haskell. Wir implementieren hierfür eine Funktion, die als Ergebnis die *Liste der Züge* berechnet. Der Einfachheit halber nehmen wir an, dass die drei Stapel (Startstapel, Zielstapel, Hilfsstapel) (am Anfang) den Zahlen 1, 2 und 3 versehen sind. Ein Zug ist dann ein Paar (a, b) von zwei (unterschiedlichen) Zahlen a und b aus der Menge {1,2,3} und bedeute: ziehe die oberste Scheibe vom Stapel a auf den Stapel b.

Wir implementieren die Funktion `hanoi` mit 4 Parametern: Der erste Parameter ist die Höhe des Turms n, die nächsten drei Parameter geben die Nummern für die drei Stapel an.

Die Hanoi-Funktion ist nun wie folgt implementiert:

```
-- Basisfall: 1 Scheibe verschieben
hanoi 1 start ziel hilf = [(start,ziel)]

-- Allgemeiner Fall:
hanoi n start ziel hilf =
   -- Schiebe den Turm der Hoehe n-1 von start zu hilf:
   (hanoi (n-1) start hilf ziel)
   ++
   -- Schiebe n. Scheibe von start auf ziel:
   [(start,ziel)]
   ++
   -- Schiebe Turm der Hoehe n-1 von hilf auf ziel:
   ++ (hanoi (n-1) hilf ziel start)
```

Der Basisfall ist der Fall $n = 1$: In diesem Fall ist ein Zug nötig, der die Scheibe vom Startstapel auf den Zielstapel bewegt. Im allgemeinen Fall werden zunächst $n - 1$ Scheiben vom Startstapel auf den Hilfsstapel durch den rekursiven Aufruf verschoben (beachte, dass dabei der Hilfsstapel zum Zielstapel (und umgekehrt) wird). Anschließend wird die letzte Scheibe vom Startstapel zum Zielstapel verschoben, und schließich wird der Turm der Höhe $n - 1$ vom Hilfsstapel auf den Zielstapel verschoben.

Gestartet wird die Funktion mit 1, 2 und 3 als Nummern für die 3 Stapel:

```
start_hanoi n = hanoi n 1 2 3
```

Z.B. kann man nun mit einem Stapel der Höhe 4 testen:

```
*Main> start_hanoi 4
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3),(1,2),
 (3,2),(3,1),(2,1),(3,2),(1,3),(1,2),(3,2)]
```

Wer möchte kann nachprüfen, dass diese Folge von Zügen tatsächlich zum Ziel führt.

A. Kochbuch

A.1. Erste Schritte

Falls Sie an einem Laptop arbeiten, können/müssen Sie Punkte 1 und 3 überspringen.

1. **Login**: Auf dem Bildschirm sollte eine Login-Maske zu sehen sein, ähnlich wie die in Abb. A.1.

Abbildung A.1.: Login-Maske, „(spuele)“ ist in diesem Fall der Rechnername, der ist bei Ihnen anders

   - unter `Login:` muss der Login-Name, den Sie von uns erhalten haben,
   - unter `Password:` muss das Passwort, welches Sie erhalten haben, eingegeben werden.
   - den `Go!`- Button klicken

2. **Shell öffnen**:

   RBI-PC: rechte Maustaste irgendwo auf dem Bildschirm klicken, nicht auf einem Icon. Im sich öffnenden Menü auf „Konsole“ klicken (Abb. A.2). Eine Shell öffnet sich (Abb. A.4).

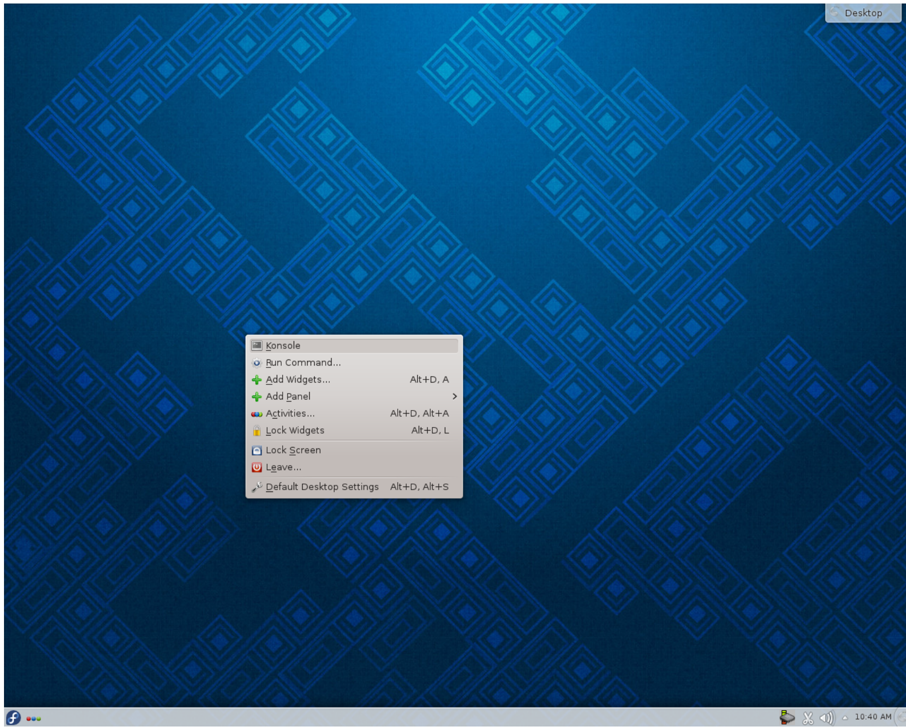


Abbildung A.2.: Terminal öffnen unter KDE

Laptop: die Laptops laufen mit einem Knoppix-System, welches von DVD bzw. USB-Stick aus gebootet wird. Um mit der aktuellen Haskell-Version zu arbeiten, muss man sich per remote login auf einem der RBI-Rechner einloggen. Dazu muss zunächst eine shell geöffnet werden.

Mit der linken Maustaste auf das Startsymbol in der linken unteren Ecke des Bildschirms klicken. Das Menü „Systemwerkzeuge“ wählen, im Untermenü „Konsole“ anklicken (Abb. A.3). Eine Shell öffnet sich (Abb. A.4).

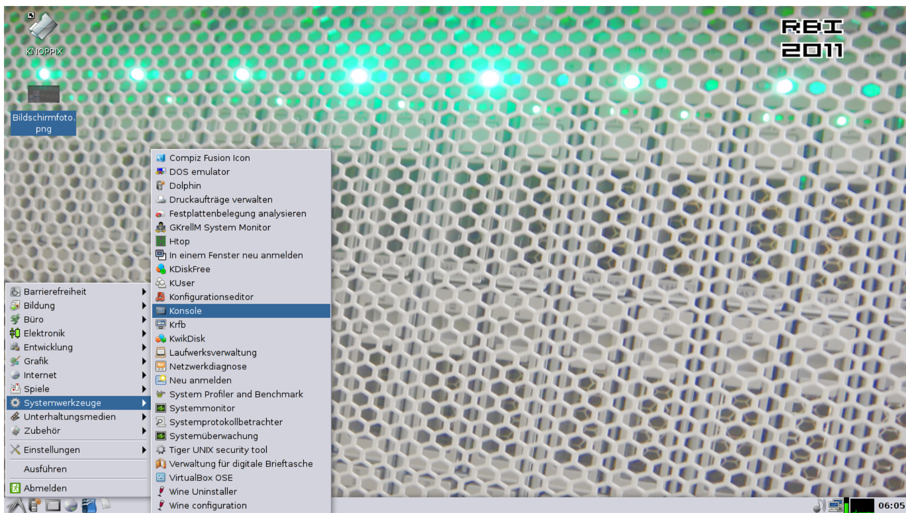


Abbildung A.3.: Terminal öffnen unter Koppix

Abbildung A.4.: Shell; in der Eingabezeile steht `[Benutzername]@[Rechnername]`

Laptop:
- in der Shell `ssh -X [Benutzername]@login.rbi.cs.uni-frankfurt.de` eingeben, ↵ drücken
- evtl. erscheint eine Nachricht, dass der ECDSA Schlüssel unbekannt ist und man wird gefragt, ob man sich tatsächlich verbinden will. In diesem Fall `yes` eingeben und ↵ drücken.
- in der Shell erscheint die Nachricht:
  `[Benutzername]@login.rbi.cs.uni-frankfurt.de's password:`
- Passwort eingeben und ↵ drücken. Es wird nicht angezeigt, dass man irgendetwas eingegeben hat. Es erscheinen keine ∗ und der Cursor bewegt sich nicht.

Nun ist man auf seinem persönlichen RBI-Account eingeloggt und kann vom Laptop aus direkt auf diesem Account arbeiten. Das heißt, alle Dateien die gespeichert werden, befindet sich dann auf diesem Account. Durch Eingabe der Option `-X` erlaubt man eine Übertragung der graphischen Oberfläche. Die Arbeit am Laptop ist nun nicht anders als die Arbeit an einem RBI-Rechner.

3. **Passwort ändern**:

- in der Shell `yppasswd` eingeben, ↵ drücken
- in der Shell erscheint die Nachricht: `Please enter old password:`
- aktuelles Passwort (das das Sie von uns erhalten haben) eingeben. Die eingegebenen Zeichen erscheinen **nicht** auf dem Bildschirm. Es sieht aus, als hätten Sie gar nichts eingegeben. Am Ende ↵ -Taste drücken.
- in der Shell erscheint die Nachricht: `Please enter new password:`
- neues Passwort eingeben. Das Passwort muss aus mindestens 6 Zeichen bestehen. Wieder erscheint die Eingabe nicht auf dem Bildschirm. ↵ -Taste drücken.
- in der Shell erscheint die Nachricht: `Please retype new password:`
- neues Passwort erneut eingeben, ↵ -Taste drücken.
- in der Shell erscheint die Nachricht: `The NIS password has been changed on zeus.`

4. **GHCi öffnen**

   - in der Eingabezeile der Shell **ghci** eintippen, ↩ -Taste drücken. Der Haskell-Interpreter öffnet sich direkt in der Shell und ist bereit Eingaben entgegenzunehmen (Abb. A.5).

```
ronja@nash: ~
ronja@nash:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Abbildung A.5.: GHCi

5. **Ein erster Haskell-Befehl**

   - Geben Sie `putStrLn ''Hello world!''` ein. Dies ist in den meisten Tutorials der meisten Programmiersprachen das Standardbeispiel für ein erstes Programm. Durch Betätigen der Return-Taste ↩ wird der Befehl direkt ausgewertet.

   - in der Shell erscheint `Hello world!` (Abb. A.6)

```
ronja@nash: ~
ronja@nash:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> putStrLn "Hello world!"
Hello world!
Prelude>
```

Abbildung A.6.: Ein erster Haskell-Befehl

6. **Ein erstes Haskell-Programm**

   a) *Öffnen eines Editors*: Geben Sie in der Shell hinter der GHCi-Eingabeaufforderung `:!gedit` ein und betätitgen Sie die ↩ -Taste. Ein gedit-Editor-Fenster öffnet sich. (Abb A.7). Falls Sie einen anderen Editor bevorzugen, geben Sie den entsprechenden Befehl ein (z.B. `xemacs` oder `kate`).

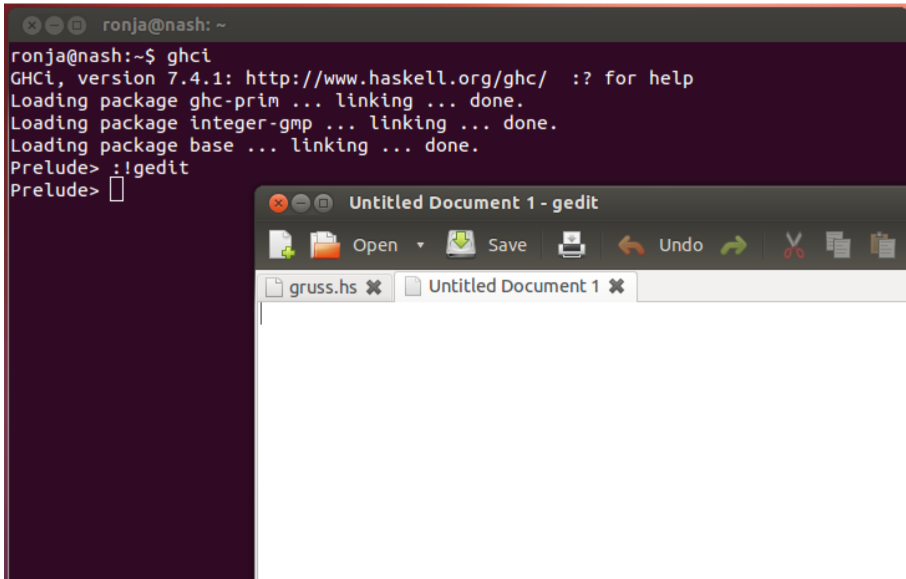


Abbildung A.7.: Öffnen eines Editors

b) *Schreiben und Speichern des Programmcodes*: Geben Sie in dem neu geöffneten Editor den Text `gruss = ''Hallo Welt!''` ein. Speichern Sie, indem Sie mit der Maus im File-Menü den Unterpunkt „Save“ wählen. Wählen Sie einen geeigneten Ordner und Dateinamen. Die Datei *muss* die Endung `.hs` haben. Sobald dem Editor durch Speichern mit der Endung `.hs` signalisiert wurde, dass es sich um einen Haskell-Quelltext handelt, erscheint der Text mehrfarbig. Das nennt man *Syntax-Highlighting*. Text erscheint in unterschiedlichen Farben, je nachdem zu welcher Kategorie der Term gehört, den der Text beschreibt.

Syntax-Highlighting

c) *Programm laden und starten*: Geben Sie im GHCiden Befehl `:load [Dateipfad/Dateiname]` ein und betätigen Sie die ⏎ -Taste. Im Beispiel ist das Programm im Homeverzeichnis im Ordner `Programme` unter dem Dateinamen `gruss.hs` gespeichert. (Abb. A.8).

```
ronja@nash: ~
ronja@nash:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :!gedit
Prelude> :load Programme/gruss.hs
[1 of 1] Compiling Main             ( Programme/gruss.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Abbildung A.8.: Laden des Programms

Das Programm ist nun geladen und kann im GHCi gestartet durch Eingabe von `gruss` gestartet werden. (Abb. A.9).

```
ronja@nash: ~
ronja@nash:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :!gedit
Prelude> :load Programme/gruss.hs
[1 of 1] Compiling Main             ( Programme/gruss.hs, interpreted )
Ok, modules loaded: Main.
*Main> gruss
"Hallo Welt !"
*Main> 
```

Abbildung A.9.: Modulaufruf und Bildschirmausgabe des Programms

7. **Arbeit beenden**

RBI-PC: Die RBI-Rechner bitte **niemals** ausschalten. Sie brauchen sich lediglich Auszuloggen. Dies geschieht, indem Sie in der linken, unteren Bildschirmecke auf das Startsymbol klicken, dann im sich öffnenden Menü den Punkt „Leave“ auswählen und auf „Log out“ klicken (Abb.: A.10). Danach erscheint wieder die Login-Maske auf dem Bildschirm.

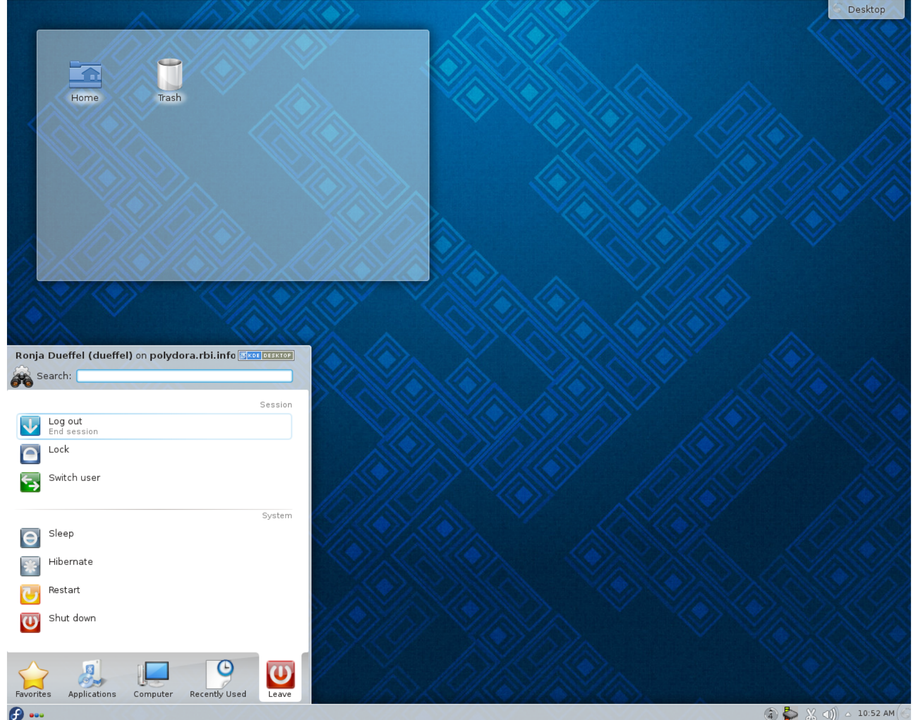


Abbildung A.10.: Ausloggen

Laptop : Den Laptop können Sie, wie Sie es von ihrem Computer zu Hause gewohnt sind, herunterfahren. Dazu auf das Startmenü (unten links) klicken, und „Abmelden“ auswählen (Abb.: A.11). Es öffnet sich ein Dialog, in dem man unterschiedliche Optionen wählen kann. Bitte klicken Sie auf „Herunterfahren“.

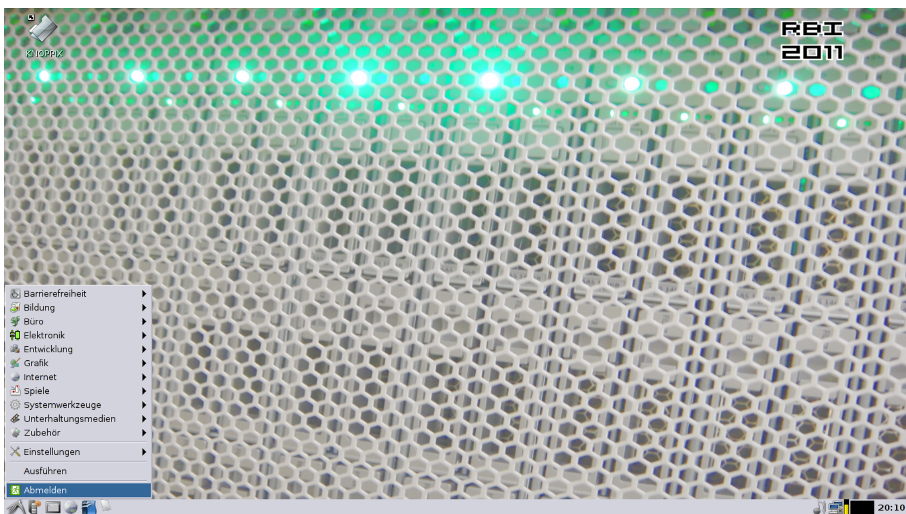


Abbildung A.11.: Herunterfahren

A.2. Remote Login

Auf die Rechner der RBI kann man sich auch über das Internet von einem anderen Rechner (z.B. von zu Hause) aus einloggen. Das ist nützlich, wenn man Programme nicht auf dem eigenen Rechner installieren möchte, oder um Lösungen zu Programmieraufgaben zu testen, denn meist wird gefordert, dass die Programme auf den RBI-Rechnern laufen. Der Rechner, von dem aus man auf einem RBI-Rechner arbeiten möchte, benötigt hierfür ein *ssh-Client*-Programm. Ferner benötigt man ein *scp-Client*-Programm, um Dateien zwischen den Rechnern auszutauschen. Außerdem muss man wissen, auf welchem RBI-Rechner man sich einloggen möchte. Eine Liste der Rechnernamen findet man auf der RBI-Webseite[1].

A.2.1. Unix-artige Betriebssysteme (Linux, MacOS, etc)

Bei allen UNIX-artigen Betriebssystemen sind ssh- und scp-Client-Programme bereits installiert und können über die Shell gestartet werden.

1. **Austausch der Daten** Mit folgendem Kommando kann man Daten aus dem aktuellen Verzeichnis seines eigenen Rechners, in sein RBI-Homeverzeichnis kopieren.

   ```
   > scp [dateiname] [benutzername]@[rechnername].rbi.cs.uni-frankfurt.de:~/ ⏎
   ```

   Um Daten vom RBI-Account in das aktuelle Verzeichnis auf dem eigenen Rechner zu kopieren, benutzt man dieses Kommando:

   ```
   > scp [benutzername]@[rechnername].rbi.cs.uni-frankfurt.de:~/[dateiname] . ⏎
   ```

   Der `.` steht dabei stellvertretend für das aktuelle Verzeichnis. Es ist auch möglich einen relativen oder absoluten Pfad (siehe Kap. 6) zu einem anderen Verzeichnis anzugeben.

2. **Einloggen**

[1] http://www.rbi.informatik.uni-frankfurt.de/rbi/informationen-zur-rbi/raumplane/

```
> ssh [benutzername]@[rechnername].rbi.informatik.uni-frankfurt.de ↵
```

Loggt man sich zum ersten Mal auf diesem Rechner ein, so wir man zunächst gefragt, ob man sich tatsächlich auf diesem unbekannten Rechner einloggen möchte. Bestätigt man dies, so wird man aufgefordert das Passwort einzugeben. Ist das erfolgreich, ist man auf dem RBI-Rechner eingeloggt. Der Name des RBI-Rechners erscheint nun in der Eingabezeile (Abb.: A.12).

```
ronja@nash: ~
ronja@nash:~$ ssh lz_inf@eos.rbi.informatik.uni-frankfurt.de
The authenticity of host 'eos.rbi.informatik.uni-frankfurt.de (141.2.15.135)' ca
n't be established.
RSA key fingerprint is 10:71:22:7e:dd:13:e4:6e:14:26:aa:bf:49:00:32:41.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'eos.rbi.informatik.uni-frankfurt.de,141.2.15.135' (R
SA) to the list of known hosts.
lz_inf@eos.rbi.informatik.uni-frankfurt.de's password:
eos:~>
```

Abbildung A.12.: Auf einem RBI-Rechner einloggen

3. **Programm starten**

Ist man auf dem RBI-Rechner eingeloggt, kann das Programm mit dem Befehl

```
> ghci [meinprogramm.hs] ↵
```

gestartet werden. Bei diesem Aufruf startet der GHCi und läd selbstständig das übergebene Modul.

4. **Verbindung beenden**

Dieser Befehl schließt die Verbindung zu dem RBI-Rechner.

Abbildung A.13 zeigt das ganze am Beispiel unseres „Hallo Welt!“-Programms welches unter dem Namen `gruss.hs` im Verzeichnis `/home/ronja/Programme/` gespeichert ist.

```
ronja@nash: ~
ronja@nash:~$ scp /home/ronja/Programme/gruss.hs lz_inf@eos.rbi.cs.uni-frankfurt
.de:~/
lz_inf@eos.rbi.cs.uni-frankfurt.de's password:
/home/users3/leitung/lz_inf/.bashrc: line 20: /bin/ttytype: No such file or dire
ctory
tset: standard error: Inappropriate ioctl for device

gruss.hs                                      100%   24     0.0KB/s   00:00
ronja@nash:~$ ssh lz_inf@eos.rbi.cs.uni-frankfurt.de
lz_inf@eos.rbi.cs.uni-frankfurt.de's password:
Last failed login: Wed Dec 11 15:56:00 CET 2013 from nash.rbi.informatik.uni-fra
nkfurt.de on ssh:notty
There were 8 failed login attempts since the last successful login.
eos:~> ghci gruss.hs
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( gruss.hs, interpreted )
Ok, modules loaded: Main.
*Main> gruss
"Hallo Welt !"
*Main> :quit
Leaving GHCi.
eos:~> exit
logout
Connection to eos.rbi.cs.uni-frankfurt.de closed.
ronja@nash:~$
```

Abbildung A.13.: Ein Haskell-Programm vom eigenen Rechner auf einem RBI-Rechner starten

A.2.2. Windows

1. **Austausch der Daten** Windows-Nutzer müssen sich zunächst ein scp-Client-Programm herunterladen. Eines der populärsten ist *WinSCP*[2]. Um Daten auszutauschen, muss man sich erst einmal mit dem RBI-Rechner verbinden. Das geschieht über die WinSCP-Login-Maske (Abb.: A.14).

[2]http://winscp.net/eng/download.php

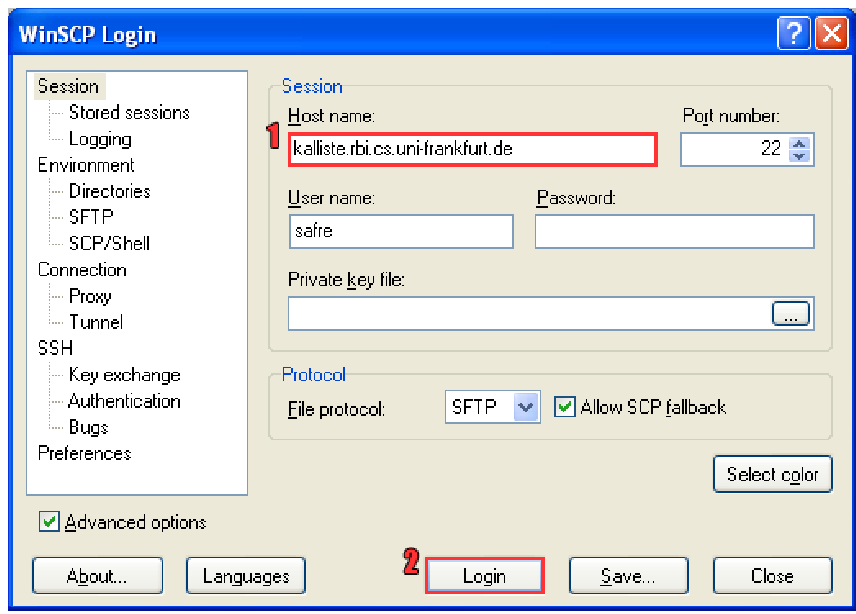


Abbildung A.14.: Das WinSCP-Login-Fenster

Sind die Rechner verbunden, können Daten wie man es vom Dateimanager gewohnt ist, mit drag-and-drop oder über das Menü ausgetauscht werden (Abb. A.15).

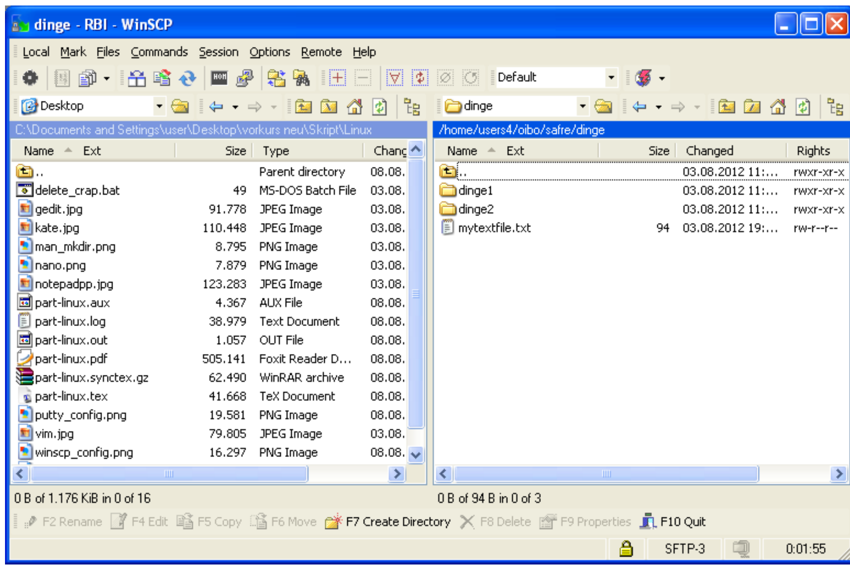


Abbildung A.15.: Das WinSCP-Hauptfenster. Links die Dateien des Windows-Rechners, rechts die Dateien im RBI-Verzeichnis

2. **Einloggen** Um sich von einem Windows-Rechner auf einem RBI-Rechner einzuloggen, benötigt man ein ssh-Client-Programm. *Putty*[3] ist solch ein Programm. Bei der Einstellung des Programms, muss die Adresse des RBI-Rechners angegeben werden, auf dem man sich einloggen möchte (Abb.: A.16).

[3] http://www.chiark.greenend.org.uk/~ sgtatham/putty/download.html

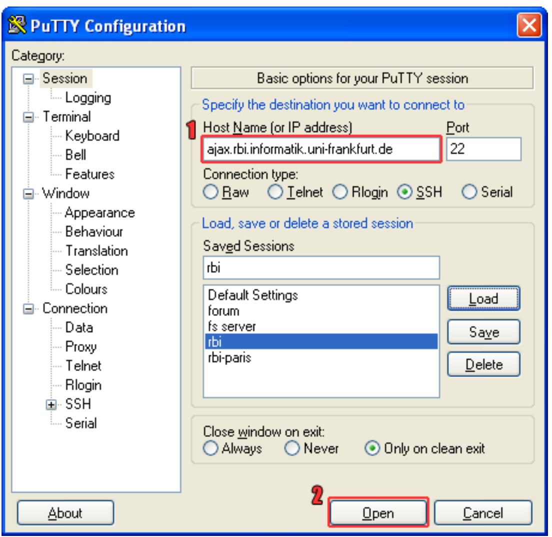


Abbildung A.16.: Einstellung von Putty

Anschließend kann man sich auf dem RBI-Rechner einloggen. Ist der Vorgang erfolgreich, so öffnet sich eine Shell, über die man auf dem RBI-Rechner Kommandos ausführen kann.

3. **Programm starten**

   In der Putty-Shell kann man nun mit folgendem Kommando das Haskell-Programm starten.

```
> ghci [meinprogramm.hs] ⏎
```

4. **Verbindung beenden** Der Befehl `exit` schließt die Verbindung zum RBI-Rechner. Wenn die Trennung erfolgreich war, sollte sich das Putty-Fenster selbstständig schießen.