Skript

Vorkurs Informatik

Wintersemester 2014/15

Sandra Kiefer
Rafael Franzke
Jens Keppeler
Mario Holldack
Ronja Düffel
Dr. Arne Nägel
Stand: 24. September 2014

Inhaltsverzeichnis

1. Programmieren und Programmiersprachen

In diesem Kapitel gehen wir kurz darauf ein, was der Vorgang des Programmierens eigentlich beinhaltet, was eine Programmiersprache ist und wie Programmiersprachen unterschiedliche Lösungsstrategien unterstützen. Kurz gehen wir auf die Gechsichte der Programmiersprache *C++* ein, mit der wir im folgenden Kapitel erste einführende Schritte unternehmen werden.

Auch, wenn es für den Laien vielleicht so aussehen mag, Programmieren besteht nicht nur daraus Zeichenfolgen in einen Computer einzutippen. Vielmehr ist das lediglich der letzte Schritt eines Prozesses, dessen denkintensivster Teil nicht aus dem Eintippen von Zeichenfolgen besteht. Computerprogramme werden für den Einsatz „im wirklichen Leben“ geschrieben. Das heißt, die Anweisungen und Anforderungen, die ein Programmierer erhält, sind in der Regel in menschlicher Sprache formuliert. Ein Computer ist jedoch, wie der Name schon sagt, eine Rechenmaschine. Zugegebenermaßen eine sehr schnelle, präzise und mächtige Rechenmaschine, aber dennoch nur eine Rechenmaschine. D.h. ein Computer kann arithmetische und logische Operationen ausführen und den Zustand seines Speichers verändern, aber nicht die Bedeutung (*Semantik*) menschlicher Sprache verstehen. Dass es sprachgesteuerte Maschinen gibt, ist der Software zu verdanken, die Sprachimpulse in Maschinenbefehle übersetzt.

Semantik

Ein *ausführbares* Computerprogramm ist eine Folge von *Maschinencodebefehlen*, auch *Maschinenprogramm* genannt. Ein einzelner Maschinencodebefehl ist eine Operation, die der Prozessor direkt ausführen kann (z.B. Addieren zweier Zahlen, Vergleich zweier Werte, Lesen oder Beschreiben eines Speicherregisters). Die Ausführung eines Programms im Sinne des Computers besteht darin, die Folge von Maschinencodebefehlen nacheinander abzuarbeiten und dabei den Speicher zu verändern. Häufig spricht man auch vom „Zustand“ des Rechners und meint damit die gesamte Speicherbelegung. Um ein ausführbares Computerprogramm zu erstellen, muss also letztendlich das in menschlicher Sprache formulierte Problem in Maschinensprache übersetzen werden.

ausführbares Programm

Maschinenprogramm

Ein Maschinenprogramm ist jedoch lediglich eine Folge von 0 und 1 und für einen menschlichen Programmierer schwer zu erstellen und zu verstehen. Daher gibt es sog. *höhere Programmiersprachen*, die es erlauben, für Menschen besser lesbare und verständliche Programme zu erstellen. Dieser, für Menschen lesbare, Text wird auch *Quelltext* oder *Quellcode* (engl.: *source code*) ge-nannt. Damit der Computer die, in der höheren Programmiersprache geschriebenen, Anweisungen im Quelltext ausführen kann, muss entweder ein *Compiler* oder ein *Interpreter* verwendet werden. Ein Compiler ist ein Übersetzungsprogramm, welches den Quelltext in ein Maschinenprogramm übersetzt. Ein Interpreter ist ein Programm, welches die Anweisungen im Quelltext schrittweise ausführt. Es interpretiert die Anweisungen einer höheren Progammiersprache. Während der Übersetzung bzw. Ausführung, wird auch überprüft, ob es sich bei dem Quelltext um ein zulässiges, d.h. gültiges Programm handelt. Ist dies nicht der Fall, dann sollte eine entsprechende Fehlermeldung ausgegeben werden.

höhere Programmiersprachen

Quellcode

Compiler

Interpreter

Zwar sind höhere Programmiersprachen für Menschen besser verständlich als Maschinenbefehle, allerdings sind sie noch weit entfernt von menschlichen Sprachen. Begriffe, Konstrukte und Befehle in menschlicher Sprache enthalten immer Interpretationsspielraum. Ein Computer benötigt aber eindeutige Handlungsanweisungen. Programmieren bedeutet also, eine Lösung für das gestellte Problem zu finden und diese Lösung in einer höheren Programmiersprache zu implementieren. Hierfür werden folgende Schritte, häufig auch mehr als einmal, durchlaufen.

1. Zerlegung des Problems in Teilprobleme
2. Präzisierung der Teilprobleme
3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem
4. Entwurf eines groben Programmgerüsts
5. Schrittweise Verfeinerung
6. Umsetzung in eine Programmiersprache
7. Testen und Korrigieren von Fehlern

Das, was Viele gemeinhin als Programmieren bezeichnen, geschieht demnach frühestens an 6. Stelle. Also seien Sie nicht frustriert, wenn Sie nach 20 Minuten noch kein bisschen Code geschrieben haben.

1.1. Programmiersprachen

Es gibt unzählige höhere Programmiersprachen. Je nachdem welches Programmierkonzept die Struktur und Vorgaben der Sprache dabei unterstützen, unterscheidet man in *deklarative* und *imperative* Progammiersprachen.

1.1.1. Deklarative Programmiersprachen

„Deklarativ“ stammt vom lateinischen Wort „declarare“, was soviel heißt wie „beschreiben“ oder auch „erklären“. Programme in deklarativen Programmierspachen beschreiben das Ergebnis des Programms. Sie erklären vielmehr *was* berechnet werden soll, als *wie* das Ergebnis berechnet werden soll. Sie bestehen aus, häufig mathematischen, Ausdrücken, die ausgewertet werden. Die Auswertung des arithmetischen Ausdrucks $(7 \cdot 5 + 3 \cdot 8)$ beispielsweise erfolgt, wie aus der Schule bekannt, über die schrittweise Auswertung der Teilausdrücke. $(7{\cdot}5{+}3{\cdot}8) = (35{+}3{\cdot}8) = (35{+}24) = 59$.

logische Programmiersprachen

Deklarative Programmiersprachen lassen sich aufteilen in *logische* und *funktionale* Programmiersprachen. Programme in logischen Programmiersprachen bestehen aus logischen Formeln und Aussagen, aus denen mithilfe logischer Schlussfolgerungen neue Aussagen hergeleitet werden. Die wohl bekannteste logische Programmiersprache ist *Prolog*.

funktionale Programmiersprachen

Programme in funktionalen Programmiersprachen bestehen aus Funktionsdefinitionen im engeren mathematischen Sinn und, evtl. selbstdefinierten, Datentypen. Das Resultat eines Programms ist immer ein einziger Wert. So scheinen funktionale Programmiersprachen nicht mehr Möglichkeiten als ein guter Taschenrechner zu bieten. Allerdings lassen sich die Funktionen nicht nur auf Zahlen, sondern auf beliebig komplexe Datenstrukturen (Listen, Bäume, Paare, usw.) anwenden. So ist es möglich Programme zu schreiben, die z.B. einen Text einlesen, Schreibfehler erkennen und als Resultat den korrigierten Text zurückliefern. Bekannte Vertreter der funktionalen Programmiersprachen sind *Scheme*, Microsofts *F#* und *Haskell*. Letztere wird in der Veranstaltung „Grundlagen der Programmierung 2“ verwendet.

1.1.2. Imperative Programmiersprachen

„Imperativ“ stammt vom lateinischen Wort „imperare“ = „befehlen“. Tatsächlich besteht ein imperatives Programm aus einer Abfolge von Befehlen, die nacheinander ausgeführt werden und den Zustand des Speichers verändern. Das klingt zunächst sehr ähnlich wie die bereits erwähnten

Maschinenprogramme. Die Befehle höherer Programmiersprachen sind jedoch komplexer als Maschinenbefehle und in der Regel wird vom tatsächlichen Speicher abstrahiert. Anstatt zu manipulierende Speicherbereiche direkt zu adressieren, kann der Programmierer beschreibende Namen vergeben und diese im Quelltext verwenden. Daher ist der Quelltext wesentlich besser lesbar als der Maschinencode. Man kann imperative Programmiersprachen in *prozedurale* und *objektorientierte* Sprachen unterscheiden.

prozedurale Programmiersprachen

Prozedurale Programmiersprachen erlauben es dem Programmierer den imperativen Quelltext durch Gruppierung zu strukturieren. Je nach Programmiersprachen heißen diese gruppierten Programmteile Unterprogramme, Routinen, Prozeduren oder Funktionen. Sie können Parameter entgegennehmen und Ergebnisse zurückgeben und innerhalb des Programms mehrfach verwendet werden. Typische prozdurale Programmiersprachen sind *C, Fortran, COBOL* und *Pascal*.

objektorientierte Programmiersprachen

Objektorientierte Programmiersprachen erlauben die Definition von Objekten, welche gewisse Eigenschaften und Fähigkeiten besitzen. Der „Bauplan“ der Objekte wird dabei in sog. *Klassen* definiert, welche *Attribute* (Eigenschaften) und *Methoden* (Fähigkeiten) besitzen. Methoden können den *Zustand* (Werte seiner Attribute) eines Objekts verändern. Methoden dienen aber auch dazu, dass Objekte untereinander Nachrichten austauschen. Ein wesentliches Konzept der objektorientierten Programmierung ist die *Vererbung*. Es können *Unterklassen* erzeugt werden, welche sämtliche Attribute und Methoden ihrer Oberklasse übernehmen und eigene hinzufügen können. Mehr zur objektorientierten Programmierung findet sich in Kapitel 2.3. Bekannte objektorientierte Programmiersprachen sind *C++, Java* und *C#*. Die meisten modernen imperativen Sprachen (z.B. *Ruby, Modula-3* und *Python*) unterstützen ebenfalls eine objektorientierte Programmierung.

1.2. Programmieren mit C und C++

Bei C und C++ handelt es sich um zwei eng verwandte Programmiersprachen, die in ihren modernen Varianten auf einen gemeinsamen Vorfahren zurückgehen. Dieser ist das *klassische C*, das von Dennis Richie an den Bell Laboratories entwickelt und implementiert wurde. Die Sprache ist prozedural und erfreut sich, nicht zuletzt seit der Veröffentlichung des Klassikers **(author?)** (KR78), großer Popularität.

Basierend auf der ursprünglichen *C*-Definition entwickelte Bjarne Stroustrup die Sparche *C++* als objektorientierte Erweiterung **(author?)** (Str). Beide Sprachen sind bis heute eng verwandt. Zur Vereinfachung der Darstellung werden wir *C* als Teilmenge von *C++* betrachten[1]. Entsprechend werden Sie im Folgenden auch Beispiele prozeduraler Programmierung finden, die in *C++* verfasst sind.

Beide Sprachen zeichnen sich durch große Maschinennähe und entsprechend hohe Performance aus. Entsprechend sind beide Sprachen heutzutage sowohl bei der Programmierung von Betriebssystemen und als auch bei der Entwicklung von Anwendungen verbreitet. Bei letztgenannten stehen mit *Java* und *C#* zudem wiederum verwandte Sprachen zur Verfügung, die viele Sprachfeatures verwenden, und aus Sicht der Entwickler auch für große Klassenhierarchien komfortabel zu verwenden sind.

[1] Wie man sich z.B. an Hand der Schlüsselworte verdeutlicht, ist dies nicht ganz exakt: Während der Ausdruck `int class = 4;` in *C* gültig ist, ist es kein gültiges *C++*, da in diesem Fall `class` ein zum Sprachstandard gehörendes Schlüsselwort ist.

1.3. Literaturhinweise zu C und C++

Klassiker/Standardwerke:

1. Kernighan, Brian W. ; Ritchie, Dennis M.: *The C programming language.* Englewood Cliffs, NJ : Prentice Hall, 1978 (Prentice-Hall software series)
2. Stroustrup, Bjarne: *The C++ programming language.* Reprinted with corr. Reading, Mass., `http://proquest.tech.safaribooksonline.de/9780133522884`

Zum Einstieg:

1. Folien der Vorlesung `http://www-stud.cs.uni-frankfurt.de/~lz_inf/Vorkurs/WS1415/webseite.html`
2. Willms, Andre: *C++-Programmierung lernen : Anfangen, Anwenden, Verstehen.* 1. Aufl. München [u.a.] : Addison-Wesley, 2008 (Programmer's choice : Klassiker). `http://proquest.safaribooksonline.com/9783827326744`
3. Stroustrup, Bjarne: *Einführung in die Programmierung mit C++.* München [u.a.] : Pearson Studium, 2010 (IT Informatik)
4. Breymann, Ulrich: *Der C++-Programmierer : C++ lernen - professionell anwenden - Lösungen nutzen.* München : Hanser, 2009
   bzw. Breymann, Ulrich: *C++ : Einführung und professionelle Programmierung.* 9., neu bearb. Aufl. München [u.a.] : Hanser, 2007
5. Dieterich, Ernst-Wolfgang: *C++.* 3., überarb. Aufl. München [u.a.] : Oldenbourg, 2000 `http://www.oldenbourg-link.de/isbn/9783486250480`

2. Erste Schritte der Programmierung mit C++

2.1. Schritt für Schritt zum ersten Programm

1. **Benutzer anmelden (nur benötigt für Rechner im RBI-Pool!):** Auf dem Bildschirm sollte eine Login-Maske zu sehen sein, ähnlich wie die in Abb. 2.1.

Abbildung 2.1.: 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 starten:**

   a) **Rechner im RBI-Pool:**

   rechte Maustaste irgendwo auf dem Bildschirm klicken, nicht auf einem Icon. Im sich öffnenden Menü auf „Konsole“ klicken (Abb. 2.2). Eine Shell öffnet sich (Abb. 2.4).

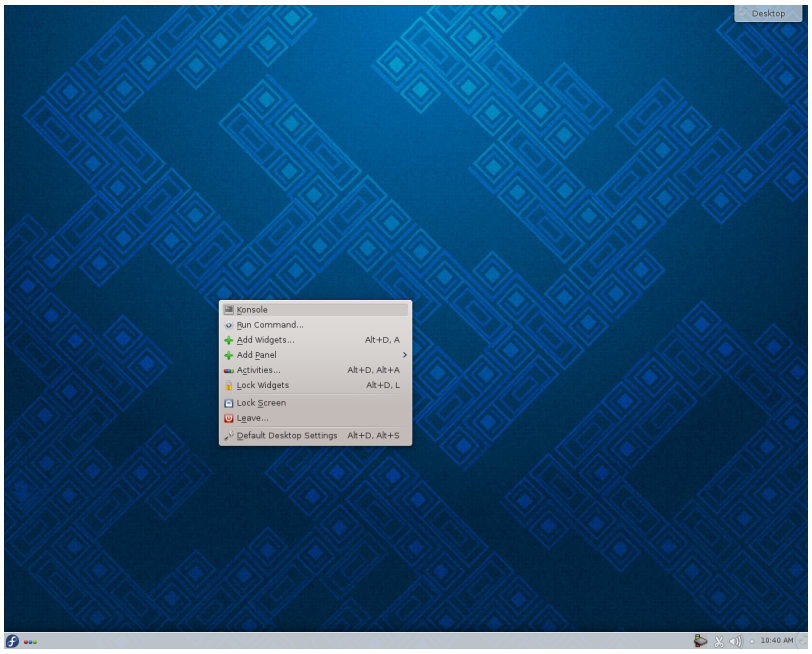


Abbildung 2.2.: Terminal öffnen unter KDE

b) **Laptop:**

- mit der linken Maustaste auf das Startsymbol in der linken unteren Ecke des Bildschirms klicken. Das Menü „Systemwerkzeuge“ wählen, im Untermenü „Konsole“ anklicken (Abb. 2.3). Eine Shell öffnet sich (Abb. 2.4).

Abbildung 2.3.: Terminal öffnen unter Knoppix

- außerdem müssen Sie eine Internetverbindung zu einem Rechner der RBI aufbauen. Geben Sie dazu den folgenden Befehl ein:

  `ssh -X [user]@login.rbi.cs.uni-frankfurt.de`, ↵-Taste drücken

  Ersetzen Sie `[user]` durch Ihren Benutzernamen. Die Option `-X` erzwingt, dass auch die grafische Oberfläche exportiert wird.

Abbildung 2.4.: Shell; in der Eingabezeile steht `[Benutzername]@[Rechnername]`

3. **Passwort ändern:** Bei der ersten Verwendung Ihres Kontos sollten Sie das Passwort, welches Sie von uns erhalten haben, durch ein selbstgewähltes ersetzen. Dies geschieht wie folgt:

   - in der Shell `yppasswd` eingeben, ↵ -Taste 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. **Eclipse starten:**

   - Starten Sie Eclipse: in der Shell `eclipse` eingeben, ↵ -Taste drücken
   - Bestätigen Sie das voreingestellte Arbeitsverzeichnis wie in Abb. 2.5 dargestellt.

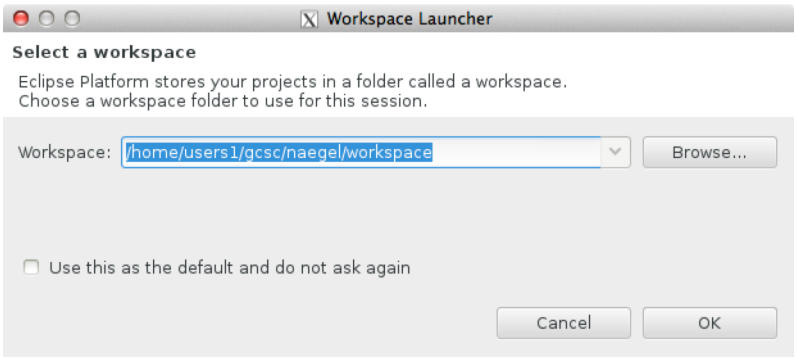


Abbildung 2.5.: Bestätigung des voreingestellten Arbeitsverzeichnis.

   - Nun können Sie Ihr erstes Projekt erstellen:

5. **Erzeugen eines C++ -Projektes in Eclipse:**

   - Wählen Sie aus dem Menü *File* > *New* > *Project*. Daraufhin erscheint die Anzeige in Abb. 2.6:

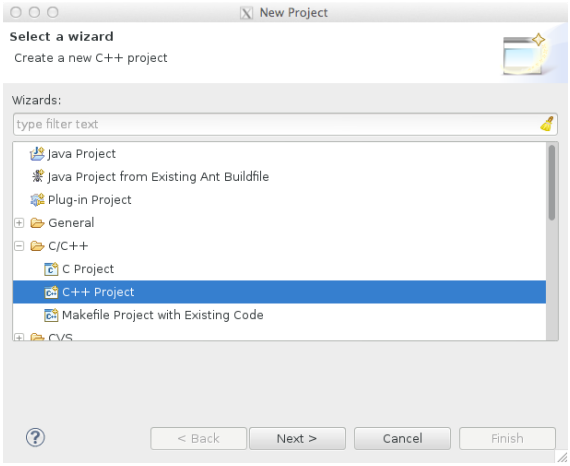


Abbildung 2.6.: Auswahl eines neuen C++ -Projektes.

   - Wählen Sie ein C++ -Projekt und fahren Sie mit *Next* fort.
   - Nun ist die C/C++ -Ansicht aktiviert. In Zukunft können Sie im Menü auch direkt *File* > *New* > *C++ Project* auswählen.
   - Die Maske in Abb. 2.7 erwartet unter *Project name* die Eingabe eines Namens für Ihr Projekt (z.B. MyHelloWorld). Wählen Sie unter *Project type* 'Hello World C++ Project' und als *Toolchain* 'Linux GCC' aus:

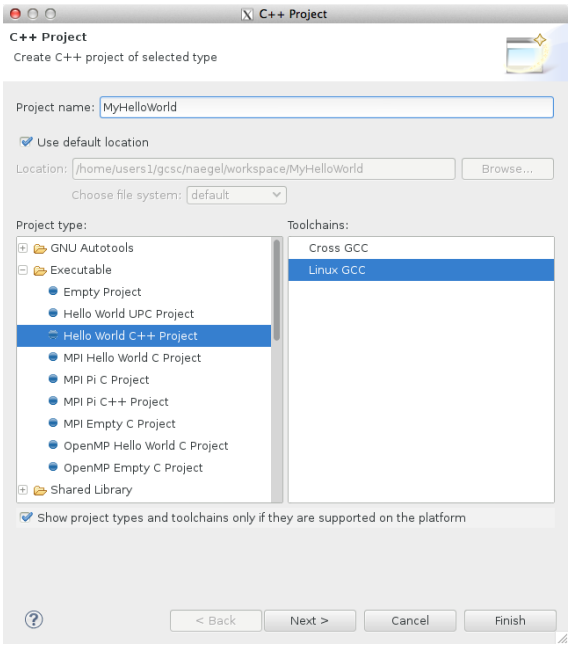


Abbildung 2.7.: Konfiguration eines C++ -Projektes.

   - Mit dem *Finish*-Schalter (rechts unten) stellen Sie das Projekt fertig!

6. **Übersetzen und Ausführen des Programms.**

- Sie erhalten die Ansicht in Abb. 2.8. Das Projekt umfasst bisher eine C++ -Datei (hier: `MyHelloWorld.cpp`).

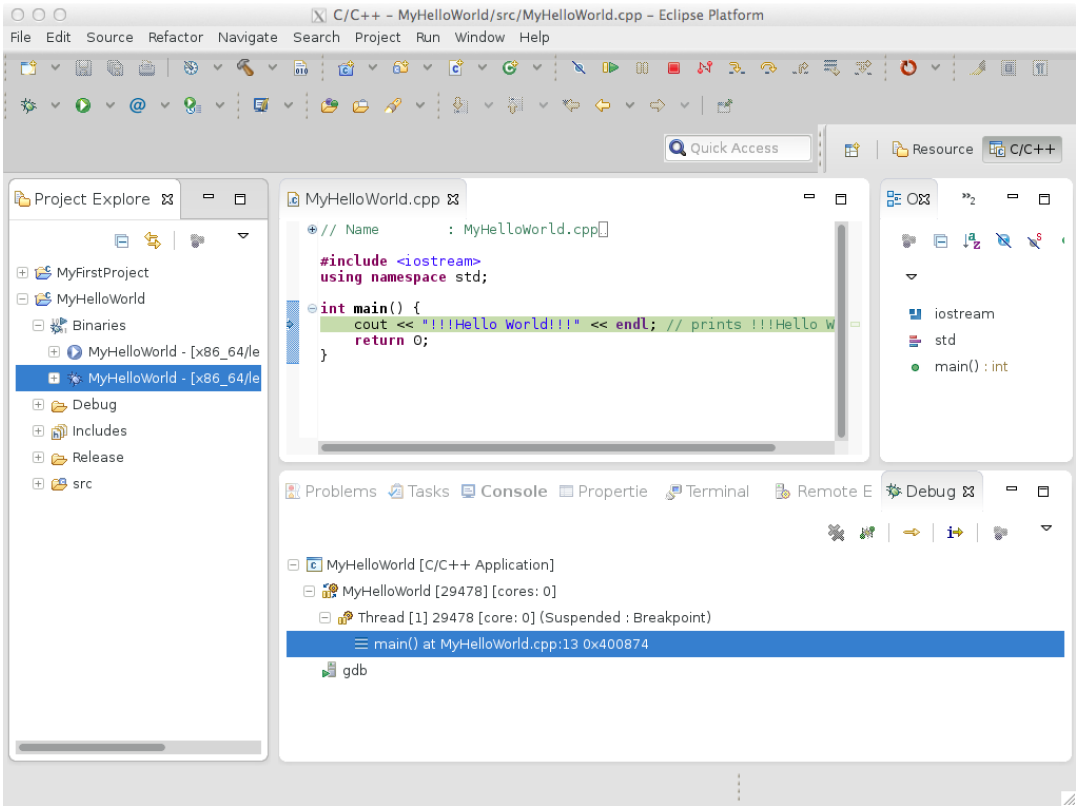


Abbildung 2.8.: Ansicht des erstellten C++ -Projektes.

- Für diese Datei wurde automatisch Quelltext generiert. Dieser sorgt dafür, dass später der Text `!!!Hello World!!!` am Bildschirm ausgegen wird.

- Zunächst müssen Sie den Quelltext jedoch in ein ausführbares Programm übersetzen. Wählen Sie dazu im Menü *Project* > *Build Project* (bzw. klicken Sie alternativ auf das Hammer-Symbol in der Menüleiste). Daraufhin sollte der Übersetzungsprozess starten und im Tab *Console* protokolliert werden.

- Um das Programm zu starten, wählen Sie im Menü *Run* > *Run* (bzw. klicken Sie alternativ auf das grüne Play-Symbol in der Menüleiste).

- Die Programmausgabe `!!!Hello World!!!` sollte nun im Tab *Console* erscheinen. Herzlichen Glückwunsch zu Ihrem ersten Programm!

- Bei Bedarf können Sie weitere weitere Quellcode-Dateien zu Ihrem Projekt hinzufügen. Wählen Sie aus dem Menü *File* > *New* > *Source file* bzw. *File* > *New* > *Header file*.

7. **Abmeldung am Rechner**

Rechner im RBI-Pool: 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.: 2.9). Danach erscheint wieder die Login-Maske auf dem Bildschirm.

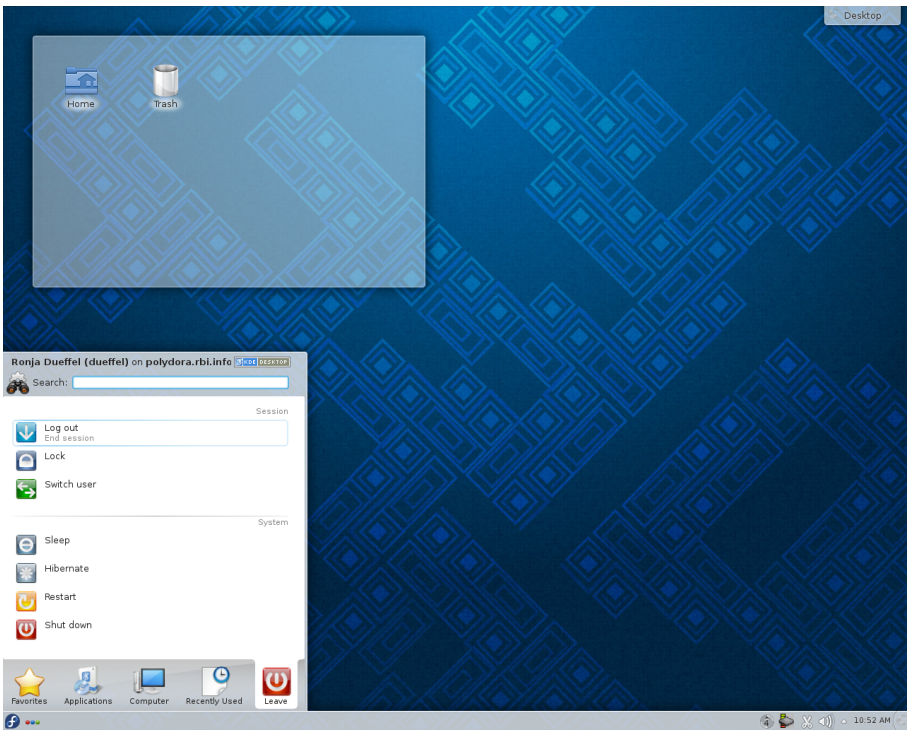


Abbildung 2.9.: Ausloggen

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.: 2.10). Es öffnet sich ein Dialog, in dem man unterschiedliche Optionen wählen kann. Bitte klicken Sie auf „Herunterfahren“.

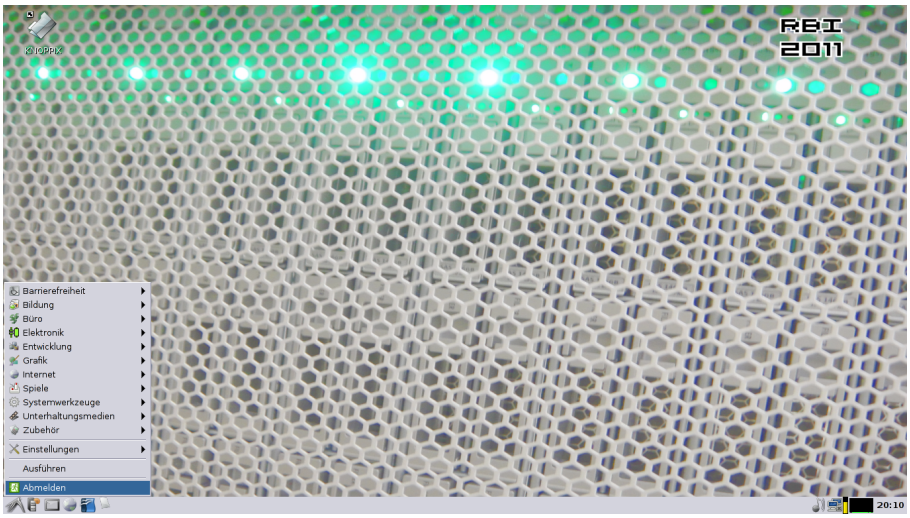


Abbildung 2.10.: Herunterfahren

2.2. Prozedurale Programmierung

C++ unterstützt die Zerlegung von Algorithmen in kleinere, überschaubare Teile, die sog. Prozedurale Programmierung. Damit lassen sich zwei Arten von Programmen schreiben, die in Kapitel 8 theoretisch aufgegriffen werden:

Die *iterativen* Programme arbeiten mit Schleifen und wiederholen Anweisungen und Anweisungsfolgen. Im Gegensatz dazu wird bei der *rekursiven* Programmierung lediglich mit Selbstaufrufen gearbeitet. Eine Funktion definiert sich durch sich selbst, d.h. eine Funktion ruft sich in ihrem Funktionskörper ein- oder mehrmals selber auf. Damit man eine rekursive Funktion in der Programmierung benutzen kann, muss man sicher gehen, dass sie irgendwann auch einmal beendet wird und sich nicht nur endlos selber aufruft. Dafür benötigt die Funktion eine *Abbruchbedingung*. Meist ist das ein Wert bei dem, wenn die Funktion mit diesem Wert aufgerufen wird, kein erneuter Funktionsaufruf gestartet wird, sondern ein Wert zurückgegeben wird. Eine rekursive Funktion terminiert, wenn es eine solche Abbruchbedingunggibt und sich in jedem *Rekursionsschritt* (Selbstaufruf der Funktion) die Problemgröße veringert und irgendwann die Abbruchbedingung erreicht.

Abbruchbedingung

Rekursionsschritt

Beispiel 2.1 (Rekursive Summe)**.**
Wir betrachten zur Illustration die Summe der ersten n natürlichen Zahlen:

$$textsumme(n) := \sum_{i=1}^{n} i = (1 + \ldots + n - 1) + n = \text{summe}(n-1) + n = \frac{n(n+1)}{2}$$

Diese Funktion lässt sich auch rekursiv programmieren, denn die Summe der ersten n ganzen Zahlen, ist nichts anderes als die Summe der ersten $n-1$ ganzen Zahlen $+n$. Die Abbruchbedingung wäre dann der Aufruf `summe(1)`, denn die Summe von 1 ist 1.

Programmbeispiel: summe.cpp

```
#include <iostream>
using namespace std;

//! Summiert Zahlen von 1..n (funktional) rekursiv
int summe_rekursiv(int n)
{
    if (n==1) return (1);
    else return summe_rekursiv(n-1)+n;
}

//! Summiert Zahlen von 1..n imperativ
int summe_iterativ(int n)
{
    int summe = 0;
    for (int i=0; i<=n; ++i) { summe += i; }
    return summe;
}

//! Aufruf & Ausgabe
int main()
{
    int n=137;
    cout << "Summe (rekursiv): " << summe_rekursiv(n) << endl;
    cout << "Summe (iterativ): " << summe_iterativ(n) << endl;
    cout << "Summe (mit Gauss): " << n*(n+1)/2 << endl;  // kleiner Gauss
    return 0;
}
```

Abbildung 2.11.: Summation über die ersten n Ganzzahlen (iterative und rekursiv programmiert).

```
Summe (rekursiv): 9453
Summe (iterativ): 9453
Summe (mit Gauss): 9453
```

Abbildung 2.12.: Ausgabe des Programms in `summe.cpp`.

In Abbildung 2.11 sind die iterativ und die rekursiv programmierte Funktion gemeinsam dargestellt. Die beiden Funktionen berechnen das gleiche. Für die gleichen Eingabewerte liefern sie den gleichen Rückgabewert (Abb. 2.12).

Was genau bei der rekursiven Funktion passiert, sieht man erst, wenn wir noch ein paar Zwischenausgaben hinzufügen (Abb. 2.13).

Programmbeispiel: Ausschnitt aus summe2.cpp

```
int summe_rekursiv(int n)
{
    cout << "Aufruf: summe_rekursiv(" << n << ");" << endl;
    if (n==1) return (1);
    else {
        int erg = summe_rekursiv(n-1) + n;
        cout << "Zwischenergebnis: summe_rekursiv(" << n-1 << ") + " << n <<" = "<< erg << ";"<<
     endl;
        return erg;
    }
}
```

Abbildung 2.13.: Modifizierte Version der rekursiven Summenberechnung.

Ruft man die Funktion nun mit dem Wert 5 auf, erhält man folgende Ausgabe:

```
Summe (rekursiv): Aufruf: summe_rekursiv(5);
Aufruf: summe_rekursiv(4);
Aufruf: summe_rekursiv(3);
Aufruf: summe_rekursiv(2);
Aufruf: summe_rekursiv(1);
Zwischenergebnis: summe_rekursiv(1) + 2 = 3;
Zwischenergebnis: summe_rekursiv(2) + 3 = 6;
Zwischenergebnis: summe_rekursiv(3) + 4 = 10;
Zwischenergebnis: summe_rekursiv(4) + 5 = 15;
15
```

Abbildung 2.14.: Ausgabe des modifizierten Programms.

Die Funktion wird nacheinander für alle Werte von 5 bis 1 aufgerufen, und erhält die Zwischenergebnisse für die Aufrufe in umgekehrter Reihenfolge zurück. Aus den Zwischenergebnissen setzt sich dann das Endergebnis zusammen.

Beispiel 2.2 (Fibonacci-Zahlen)**.**
Die Fibonacci-Zahlen sind eine unendliche Zahlenfolge, die sog. *Fibonacci-Folge*. Der theoretische Hintergrund der Fibonacci-Folge wird in Beispiel 8.16 behandelt. Hier wollen wir zeigen, wie wir eine rekursive Funktion programmieren können, die die n-te Fibonacci-Zahl berechnet. Mathematisch sind die Fibonacci-Zahlen wie folgt rekursiv definiert: Fibonacci Folge

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

Die Funktion lässt sich ziemlich leicht in eine C-Funktion übertragen (Abb. 2.15):

Programmbeispiel: fibonacci.cpp

```
#include <iostream>
using namespace std;

//! Rekursive Berechnung der Fibonacci Zahlen
int fib(int n)
{
    if ((n<=2)) return (1);
    else return (fib(n-1)+fib(n-2));
};

//! Aufruf & Ausgabe
int main()
{
    cout << "fib(5): " << fib(5) << endl;
    cout << "fib(10): " << fib(10) << endl;
    cout << "fib(20): " << fib(20) << endl;
    cout << "fib(35): " << fib(35) << endl;
    return 0;
}
```

Abbildung 2.15.: Rekursive Berechnung der Fibonacci-Zahlen

Wenn wir obiges Programm ausführen, erhalten wir folgende Ausgabe:

```
fib(5): 5
fib(10): 55
fib(20): 6765
fib(35): 9227465
```

Abbildung 2.16.: Ausgabe des Programms zur rekursiven Berechnung der Fibonacci-Zahlen

Allerdings fällt bei der Ausführung auf, dass das Ergebnis des Funktionsaufrufs `fib(35)` sekundenlang auf sich warten lässt. Beim Aufruf von `fib(40)` wartet man bereits minutenlang auf das Ergebnis.

Die Erklärung hierfür liegt in der Rekursion. Jeder Aufruf von `fib(n)` mit $n > 2$ bewirkt zwei weitere Aufrufe der Funktion, welche erneut je zwei Aufrufe bewirken, usw.... So entsteht für den Aufruf `fib(5)` bereits ein Rekursionsbaum mit 9 Funktionsaufrufen (Abb. 2.17).

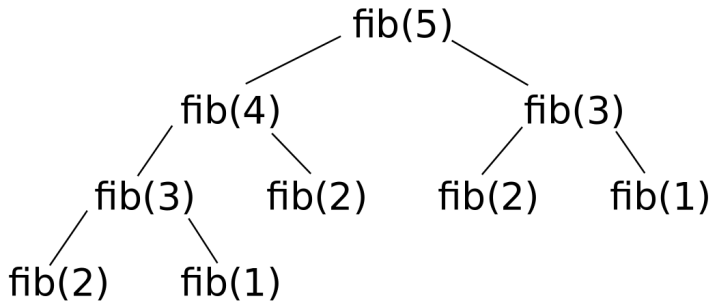


Abbildung 2.17.: Rekursionsbaum für Aufruf von `fib(5)`

Die Berechnung `fib(2)` wird gleich dreimal, die Berechnung von `fib(3)` zweimal ausgeführt. Viele Dinge werden also mehrfach ausgerechnet, das macht das Programm so langsam. Wenn man allerdings die Ergebnisse zwischenspeichert und auf bereits berechnete Zwischenergebnisse zurückgreift, ist die rekursive Implementierung (Programmierung) sehr schnell.

Die rekursive Programmierung ist bei vielen Problemen der einfachere Weg zu einer Lösung, da viele Prozesse eine rekursive Struktur haben und unser Denken auf rekursiven Denkprozessen basiert. Allerdings muss man vorsichtig bei der Programmierung sein. Wie das Beispiel der Fibonacci-Zahlen zeigt, kann unüberlegte Rekursion auch sehr ineffizient sein.

2.3. Objektorientierte Programmierung

Die objektorientierte Programmierung ist ein Programmierstil. Es ist eine Art und Weise Software (Programme) zu entwerfen, die sich an die Prinzipien der Objektorientierung hält. Objektorientierung bedeutet, dass ein System durch das Zusammenspiel kooperierender Objekte beschrieben wird. Dabei ist der Objektbegriff unscharf gefasst. Es muss keine Einheit sein, die man sehen und anfassen kann. Alles, dem bestimmte Eigenschaften und Verhaltensweisen (*Methoden*) zugeordnet werden können und das in der Lage ist, mit anderen Objekten zu interagieren, ist ein Objekt.

Objektorientierung liegt in der Natur des Menschen. Schon früh erkennen Kinder, dass Roller, Dreirad, Fahrrad, Auto oder Laster alle etwas gemeinsam haben. Es sind alles Fortbewegungsmittel. Täglich ordnen wir Gegenstände in Gruppen. Äpfel, Birnen oder Bananen sind Obst, Tomaten, Gurken oder Möhren sind Gemüse. Programmiert man ein Programm zur Lagerverwaltung, so arbeitet man genauso mit Objekten. Waren, Regale, Transportmittel, und viele mehr. Das Leben ist voller Objekte, und Programmierer haben erkannt, dass komplexe Probleme einfacher oder nachhaltiger zu lösen sind, wenn für objektorientierte Probleme, objektorientierte Lösungen gefunden werden können.

Attribut

Methode

Bei der objektorientierten Programmierung können Objekte direkt im Programm abgebildet werden. Dabei beschränkt man sich auf die Eigenschaften und Verhaltensweisen des Objekts, die für die Anwendung des Programms relevant sind. Eigenschaften eines Objekts werden *Attribute* genannt. Typische Eigenschaften des Objekts Mensch wären z.B. Name, Geburtsdatum, Größe, Gewicht oder Haarfarbe, während die Eigenschaften eines Autos Geschwindigkeit, Kraftstoffstand oder Farbe seien könnten. Das Verhalten eines Objekts wird durch seine *Methoden* definiert. Ein Mensch nimmt zu oder wird älter, ein Auto kann beschleunigen oder bremsen. Objekte haben oft Beziehungen untereinder, was ebenfalls als Methode anzusehen ist. Ein Mensch kann z.B. ein Auto fahren. Somit hat der Mensch eine Methode `fahren(Auto)` die auf ein Auto zugreift.

Um nicht für jedes einzelne Objekt immer wieder seine Eigenschaften und sein Verhalten festzulegen, werden gleichartige Objekte zu Gruppen zusammengefasst. Diese Gruppen nennt man

Klasse

Klassen. Klassen sind der Bauplan für Objekte. In ihnen werden alle Attribute und Methoden, welche die Objekte der Klasse ausmachen, definiert. Die konkreten Objekte werden über die Klasse erzeugt. Häufig wird ein konkretes Objekt auch als *Instanz* (von engl.: *instance*) der Klasse bezeichnet (Abb. 2.18). Instanz

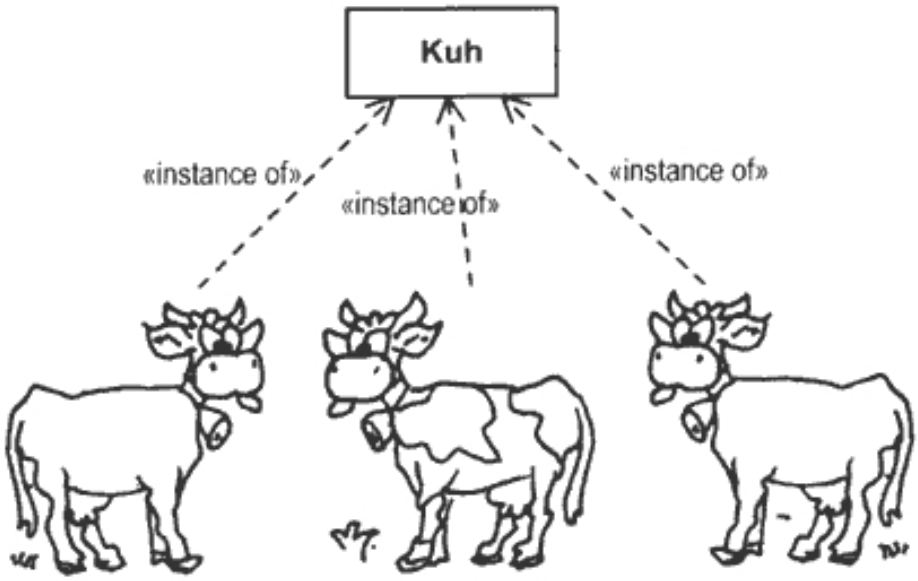


Abbildung 2.18.: Instanzen einer Klasse, v.l.n.r.: Elsa Euter, Vera Vollmilch und Anja v.d. Alm
Quelle: www.hki.uni-koeln.de

Elsa Euter, Vera Vollmilch und Anja v.d. Alm beispielsweise sind Objekte, oder Instanzen der Klasse Kuh. Sie haben natürlich unterschiedliche Attributwerte. Elsa hat ihr eigenes Geburtsdatum, und gibt im Durchschnitt mehr Milch als ihre Kollegin Anja v.d. Alm und hat einen eigenen Namen. Die Werte seiner Attribute definieren den *Zustand* eines Objekts. Zustand

Klassen werden in einer hierarchischen Beziehung aufgebaut. Z.B. könnte es die Klasse “Nutztier” geben. Diese Klasse hat die Kindklassen “Kuh” und “Schwein”. Die Kindklassen erben die Eigenschaften und das Verhalten der allgemeineren Elternklasse Nutztier und haben zusätzlich noch andere Eigenschaften und Methoden, die in der allgemeineren Elternklasse nicht enthalten sind. So benötigt die Klasse “Kuh” eine Methode `gemolken_werden()`. Zwischen Kind- und Elternklasse besteht eine “ist ein” Beziehung. Eine Kuh *ist ein* Nutztier.

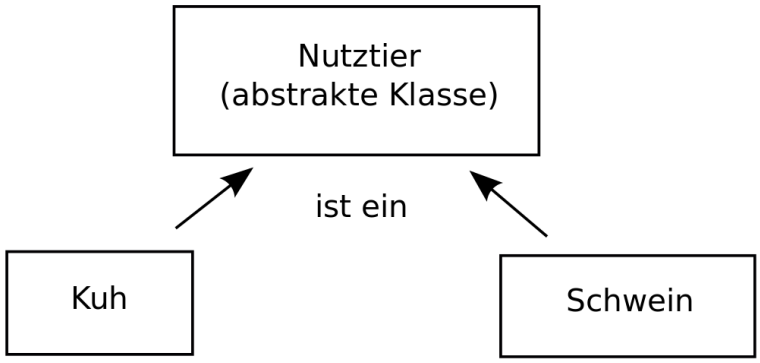


Abbildung 2.19.: Eltern- und Kindklassen

abstrakte Klasse

Die Klasse "Nutztier" könnte in diesem Fall eine *abstrakte Klasse* sein. Abstrakte Klassen sind ein gängiges Konzept in der objektorientierten Programmierung. Sie enthalten selbst nur leere Methoden. Dementsprechend kann keine Instanz von ihnen erzeugt werden. Sie dienen dazu ähnliche Klassen unter einem Oberbegriff zusammenzufassen und gemeinsame Methoden- und Attributnamen zu definieren. Sie müssen abgeleitet werden, um sinnvoll verwendet zu werden. Dann garantieren Sie das Vorhandensein der in Ihnen definierten Attribute und Methoden in allen ihren Kindklassen.

Vorteile der objektorientierten Programmierung sind:

Abstraktion

Abstraktion: Jedes Objekt im System kann als ein abstraktes Modell eines Akteurs betrachtet werden. Bei der Systementwicklung konzentriert sich der Entwickler zunächst darauf, welche Objekte benötigt werden und welche Objekteigenschaften und -fähigkeiten für die Softwarelösung wichtig sind, ohne direkt festzulegen, wie dieses *implementiert* (programmiert) werden müssen. Das verhindert eine frühe Festlegung auf Datenstrukturen etc. bevor das Problem vollständig verstanden wurde.

Kapselung

Kapselung: auch *information hiding* genannt. Objekte können auf Attribute anderer Objekte nur über vordefinierte Methoden zugreifen. Nach außen ist somit lediglich sichtbar *was* implementiert ist, aber nicht *wie*. Dadurch kann die Implementierung eines Objektes geändert werden, ohne die Zusammenarbeit mit anderen Objekten zu beeinträchtigen, solange sich die *Schnittstelle* (nach außen sichtbare Methoden) nicht ändert.

Schnittstelle

Vererbung: durch die Vererbungsbeziehung zwischen Klassen ist es möglich, dass ähnliche Kindklassen eine gemeinsame Struktur nutzen. Dadurch werden Redundanzen vermieden und die konzeptuelle Klarheit wird erhöht.

Wiederverwendbarkeit: Abstraktion, Kapselung und Vererbung erhöhen die Wiederverwendbarkeit objektorientierte Software enorm. Nicht nur können entworfene Klassen und Strukturen in anderen Programmen wiederverwendet werden, auch die Weiterentwicklung von Software wird durch die Modularität deutlich erleichtert. Da im System lediglich die Interaktion von Objekten wichtig ist, ist objektorientierte Software sehr viel übersichtlicher und damit leichter testbar, stabiler und änderbar.

Beispiel 2.3 (Bankkonto)**.**
Stellen wir uns eine Klasse "Bankkonto" vor. Ein Bankkonto hat einen Kontoinhaber, eine Kontonummer und einen Kontostand. Typische Methoden für ein Bankkonto wären `einzahlen(betrag)` und `auszahlen(betrag)`. Diese Methoden verändern den Wert des Attributs Kontostand. Ein Kontoinhaber hat einen Namen, Anschrift, und Geburtsdatum. Durch das Verhalten "umziehen" ändert sich die Anschrift (Abb. 2.20)

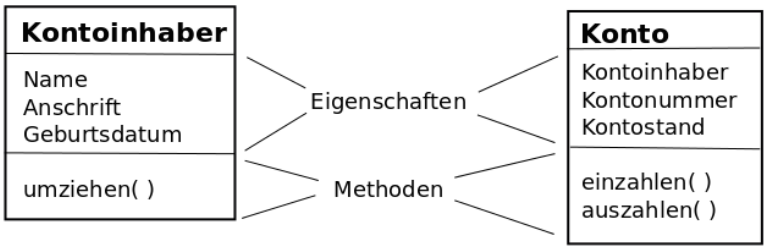


Abbildung 2.20.: Klassendiagramme der Klassen Kontoinhaber und Konto

Die Klasse "Konto" genügt einer realen Bank nicht. Es gibt verschiedene Arten von Konten. Sparkonten, die nicht überzogen werden drürfen und Girokonten, die bis zu einem gewissen Rahmen sehr wohl überzogen werden dürfen. Die Eigenschaften und Methoden unserer Klasse "Konto", sind aber beiden Kontoarten gemeinsam. Dies sind Attribute und Methoden, die sie von der Klasse "Konto" erben (Abb. 2.21)

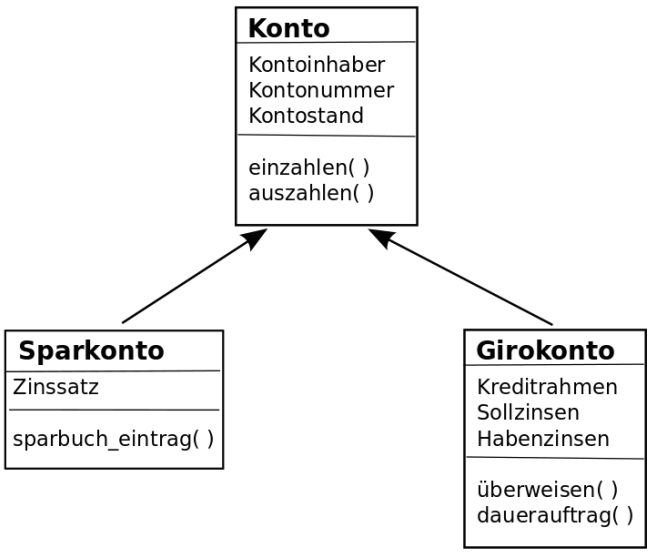


Abbildung 2.21.: Klassenhierarchie Konto

2.3.1. Implementierung

Zunächst führen wir eine Klasse als erweiterte Strukur ein. Innerhalb der Klasse werden Attribute und Methoden definiert. Dabei ist eine Methode eine innerhalb einer Klasse definierte Funktion. Unsere Klasse "Konto" könnte dann wie folgt aussehen (Abb. 2.22 und abb:KontoDef):

Programmbeispiel: Definition in der Headerdatei konto.h

```
#ifndef _KONTO_H_
#define _KONTO_H_

#include <string>   // Benutze std::string
using namespace std;

// Erster Entwurf fuer ein Konto
struct Konto
{
    // Attribute (Daten)
    string inhaber;
    long kontonummer;
    double saldo;

    static int anzKont;     // Klassenvariable aller Instanzen von 'Konto'

    // Methoden (definiert in konto.cpp)
    Konto(string name, long nummer, double betrag);
    ~Konto();

    void einzahlen(double betrag);       // Ein-/Auszahlungen
    void auszahlen(double betrag);


};  // Klassendefinition endet mit Semikolon

#endif /* KONTO_H_ */
```

Abbildung 2.22.: Implementierung der Klasse Konto

Konstruktor

Die Implementierung der Klasse "Konto" erzeugt noch kein Konto an sich. Es wird lediglich definiert, welche Eigenschaften ein Konto hat und was man mit ihm machen kann. Um ein Exemplar der Klasse "Konto" anzulegen, muss der *Konstruktor* aufgerufen werden. Der Konstruktor ist eine spezielle Methode. Diese besitzt keinen Typ und besitzt den gleichen Namen wie die Klasse selbst. Im Konstruktor werden Initialisierungen bei der Erstellung einer Instanz der Klasse vorgenommen. In unserem Beispiel muss jedes Konto einen Kontoinhaber, eine Kontonummer und einen Kontostand (Saldo) haben. Dementsprechend müssen beim Anlegen eines Kontos (Exemplar der Klasse Konto) Werte für diese Attribute übergeben werden. Diese werden dann den entsprechenden Attributen (`inhaber, kontonummer, saldo`) zugewiesen.

Klassenvariable

Ferner haben wir für unsere Konto-Klasse eine Klassenvariable `anzKont` definiert. Eine Klassenvariable wird von allen Instanzen der Klasse geteilt. Dies geschieht über das Schlüsselwort `static`. Mit `<Klassenname>::<Variablenname>` kann auf Klassenvariablen zugegriffen werden. In unserer Klasse soll die Klassenvariable angeben wie viele Konten es aktuell gibt. Dementsprechend muss beim Aufruf des Konstruktors jedesmal der Wert von `Konto::anzKont` erhöht werden. Ferner haben wir die Methoden `einzahlen` und `auszahlen` definiert.

Destruktor

Es ist auch möglich einen *Destruktor* für eine Klasse zu entwerfen. Mit dem Destruktor kann eine Instanz der Klasse gelöscht werden. In C++ wird der Destruktor (ähnlich zum Konstruktor) als spezielle Methode definiert. Sie besitzt weder Typ noch Argumente, der Name entspricht dem Namen der Klasse mit einer vorangestellten Tilde $\sim$. In unserem Fall wollen wir, dass die Klassenvariable `Konto::anzKont` um 1 reduziert wird, wenn ein Konto gelöscht wird.

Programmbeispiel: Deklaration in der Programmdatei konto.cpp

```
#include "konto.h"

//! Konstruktor
Konto::Konto(string name, long nummer, double betrag)
: inhaber(name), kontonummer(nummer), saldo(betrag)
{Konto::anzKont++;}

//! Destruktor
Konto::~Konto()
{Konto::anzKont--;}

//! Gemeinsame (statische) Zaehlvariable
int Konto::anzKont = 0;

//! Einzahlung
void Konto::einzahlen(double betrag)
{saldo += betrag;}

//! Auszahlung
void Konto::auszahlen(double betrag)
{saldo -= betrag;}
```

Abbildung 2.23.: Implementierung der Klasse Konto

Mit der nun entworfenen Klasse können wir nun in einem Programm arbeiten (Abb. 2.24):

Programmbeispiel: Verwendung der Klasse Konto demo_konto.cpp

```
#include <iostream>
#include "konto.h"

using namespace std;

int main()
{
    // Setze Anzahl der Konten zurueck
    Konto::anzKont = 0;

    // Erzeuge zwei Konten
    Konto *k1 = new Konto("Joe", 123, 1000.0);
    Konto *k2 = new Konto("Jane", 456, 10000.0);

    k1->einzahlen(50.79);           // Einzahlung auf Konto 1
    k2->auszahlen(6000.0);          // Auszahlung von Konto 2

    // Ungekapselte (direkte) Abfrage der Daten
    cout << k1->inhaber << "s Kontostand ist: " << k1->saldo << endl;
    cout << k2->inhaber << "s Kontostand ist: " << k2->saldo << endl;

    // Loesche Konten, ueberpruefe Anzahl
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
    delete k1;
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
    delete k2;
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
}
```

Abbildung 2.24.: Verwendung der Klasse Konto

Ausgabe des Programms:

```
Joes Kontostand ist: 1050.79
Janes Kontostand ist: 4000
Anzahl der Konten: 2
Anzahl der Konten: 1
Anzahl der Konten: 0
```

Abbildung 2.25.: Ausgabe des Programms zur Klasse Konto

Bemerkung: In unserem Beipiel dient der Destruktor vornehmlich der Illustration. Das explizite Angeben von Destruktoren ist aber insbesondere wichtig, wenn eine Klasse intern Speicher allokiert. Wenn ein Objekt am Ende seiner Lebenszeit gelöscht wird, muss dann auch dieser Speicher wieder korrekt freigegeben werden.

2.3.2. Datenkapselung

Im Augenblick sind noch alle Attribute unserer Klasse "Konto" direkt von außen zugänglich. Mit dem Aufruf `k2->saldo` kann man direkt Jane's Kontostand abrufen, ohne über die `abfrage`-Methode gehen zu müssen (Abb. 2.25). Das widerspricht aber dem Prinzip der Kapselung, welches in der objektorientierten Programmierung so vorteilhaft ist. Daher gibt es Möglichkeiten den direkten Zugriff zu unterbinden. In C++ erfolgt dies über die Schlüsselworte `public`, `protected` und `private`:

public: Attribute und Methoden sind öffentlich, d.h. sowohl innerhalb als auch außerhalb der Klasse sichtbar (also les- und schreibbar).

protected: Attribute und Methoden sind geschützt, d.h. nur innerhalb Klasse sowie in ggf. abgeleiteten Klassen (s. Abschnitt 2.3.3) sichtbar.

private: Attribute und Methoden sind privat, d.h. nur innerhalb Klasse selbst (s. Abschnitt 2.3.3) sichtbar.

Nun verbessern wir unsere Kontoklasse und schützen sie vor ungewolltem Zugriff von außen (Abb. 2.26). Dabei fügen wir zwei Methoden hinzu, um den Zugriff von außen zu gewährleisten (Abb. **??**).

Programmbeispiel: Definition in der Headerdatei konto.h

```
#ifndef _KONTO_H_
#define _KONTO_H_


#include <string>   // Benutze std::string
using namespace std;

// Basisklasse fuer Konten. Hiervon werden 'Sparkonto' und 'Girokonto' abgeleitet.
class Konto                                  // NEU: Verwende 'class' statt 'struct'
{
protected:                                   // NEU: Attribute (nun gekapselt)
    string inhaber;
    long kontonummer;
    double saldo;

public:
    static int anzKont;                     // Klassenvariable aller Instanzen

    // Methoden (definiert in konto.cpp)
    Konto(string name, long nummer, double betrag);
    virtual ~Konto();                       // NEU: Destruktor ist 'virtuell'

    virtual void einzahlen(double betrag);  // NEU: Funktion ist 'virtuell'
    virtual void auszahlen(double betrag);  // NEU: Funktion ist 'virtuell'

    double kontostand() const;
    const string& get_inhaber() const;

};  // Klassendefinition endet mit Semikolon


#endif /* KONTO_H_ */
```

Abbildung 2.26.: Die verbesserte Klasse Konto

Programmbeispiel: Deklaration in der Programmdatei konto.cpp

```
//! Wem gehoert das Konto?
const string& Konto::get_inhaber() const
{return inhaber;}

//! Abfrage des Kontostands
double Konto::kontostand() const
{return saldo;}
```

Abbildung 2.27.: Erweiterte Implementierung der Klasse Konto

Bemerkung: Mit dieser verbesserten Klassendefinition lässt sich das Programms aus Abb. 2.24 nun nicht mehr übersetzen! Da die Daten gekapselt sind, müssen wir die Zeilen

```
cout << k1->inhaber << "s Kontostand ist: " << k1->saldo << endl;
cout << k2->inhaber << "s Kontostand ist: " << k2->saldo << endl;
```

nun durch

```
cout << k1->get_inhaber() << "s Kontostand ist: " << k1->kontostand() << endl;
cout << k2->get_inhaber() << "s Kontostand ist: " << k2->kontostand() << endl;
```

ersetzen.

2.3.3. Vererbung

Nun hatten wir schon zu Beginn des Beispiels festgestellt, dass eine reale Bank verschiedene Kontoarten unterscheiden können muss. Dementsprechend legen wir nun eine Kindklassen "Sparkonto" an. Der Name der Elternklasse wird bei der Definition der Kindklasse mit einem Doppelpunkt hinter den Klassennamen geschrieben (Abb. 2.28).

Im Konstruktor der Kindklasse rufen wir den Konstruktor der Elternklasse auf. Zusätzlich zu den Attributen der Elternklasse "Konto" benötigt das Sparkonto noch eine Attribut `zinssatz`. Dementsprechend hat der Konstruktor von "Sparkonto" einen weiteren Übergabeparameter `zins`, welcher im Konstruktor dem Attribut `zinssatz` zugewiesen wird. Für den Zugriff auf `zinssatz` programmieren wir die beiden Methoden `get_zinssatz` und `set_zinssatz`. Ferner implementieren wir die Methode `erzeuge_eintrag`, um Ein- und Auszahlungen im Sparbuch zu dokumentieren.

Überschreiben von Methoden

Da diese Funktion in engem Zusammenhang mit Ein- und Auszahlungen steht und verlässlich bei jeder Ein- und Auszahlung ausgeführt werden muss, definieren wir die Methoden `einzahlen(betrag)` und `auszahlen(betrag)` der Elternklasse "Konto" für die Kindklasse "Sparkonto" neu. Diesen Vorgang nennt man auch *überschreiben*von Methoden. Beide Methoden rufen nun am Ende die `erzeuge_eintrag(betrag)`-Methode auf. Die `auszahlen(betrag)`-Methode überprüft zusätzlich, ob der Saldo für die Auszahlung ausreichend ist.

Damit können, wie zuvor bei der Klasse "Konto", Instanzen der Klasse "Sparkonto" erzeugt werden. Beim Aufruf der `einzahlen(betrag)` und `auszahlen(betrag)`-Methoden, werden die Methoden der Klasse "Sparkonto" ausgeführt. Da "Sparkonto" von "Konto" erbt, ist aber auch die Klassenvariable `Konto::anzKont` verwendbar (Abb. 2.30). Aufgrund der Implementierung, dass nämlich jede Instanz der Klasse "Sparkonto" auch eine Instanz der Klasse "Konto" enthält, arbeitet der Zähler auch korrekt.

Programmbeispiel: Definition in der Headerdatei sparkonto.h

```
// Die Klasse 'Sparkonto' erweitert 'Konto' um einen Zinssatz
class Sparkonto : public Konto
{
public:

    // Konstruktor erwartet nun (optional) ein weiteres Arguments
    Sparkonto(string name, long nummer, double betrag, double zins=0.0)
    : Konto (name, nummer, betrag), zinssatz(zins) {}

    // 'Ueberschreibe' (virtuelle) Funktionen zur Ein-/Auszahlung
    void einzahlen(double betrag);
    void auszahlen(double betrag);

    // Methoden zum Setzen und Lesen des Zinssatzes
    inline void set_zinssatz(double r) { zinssatz = r;}
    inline double get_zinssatz() { return zinssatz;}

protected:
    // Zinssatz
    double zinssatz;

    // Ausagbe: Eintrag ins Sparbuch (nicht oeffentlich!)
    void erzeuge_eintrag(double betrag);

};
```

Abbildung 2.28.: Die Klasse Sparkonto, Kindklasse von Konto.

Programmbeispiel: sparkonto.cpp

```
// Ueberschreibe virtuelle Methode Konto::einzahlen
void Sparkonto::einzahlen(double betrag)
{
    Konto::einzahlen(betrag);
    erzeuge_eintrag(betrag);
}

// Ueberschreibe virtuelle Methode Konto::auszahlen
void Sparkonto::auszahlen(double betrag)
{
    if (kontostand() > betrag)
    {
        Konto::auszahlen(betrag);
        erzeuge_eintrag(-betrag);
    }
    else
    {
        cout << "FEHLER: Guthaben reicht nicht aus!" << endl;
    }
}

// Ausgabe: Schreibe Eintrag ins Sparbuch (auf den Bildschirm)
void Sparkonto::erzeuge_eintrag(double betrag)
{
    cout << "Eintrag in Sparbuch #"<< kontonummer <<" (in Euro): "<< betrag;
    cout << " -> Neuer Saldo: " << kontostand() << endl;
}
```

Abbildung 2.29.: Implementierung von Sparkonto.

Programmbeispiel: demo_spar.cpp

```
using namespace std;


int main()
{
    // Setze Anzahl der Konten zurueck
    Konto::anzKont = 0;

    // Erzeuge zwei Konten
    Sparkonto *k1 = new Sparkonto("Joe", 123, 1000.0);
    Sparkonto *k2 = new Sparkonto("Jane", 456, 10000.0, 1.5);

    // Ein-/Auszahlungen
    k1->einzahlen(50.79);
    k2->auszahlen(6000.0);
    k1->auszahlen(2000.0);  // Das hier ist zuviel...

    // Ungekapselte Abfrage der Daten erzeugt nun Compilerfehler
    // cout << k1->inhaber << "s Kontostand ist: " << k1->saldo << endl;

    // Gekapselte Abfrage der Daten ist korrekt
    cout << k1->get_inhaber() << "s Kontostand ist: " << k1->kontostand() << endl;
    cout << k2->get_inhaber() << "s Kontostand ist: " << k2->kontostand() << endl;

    // Abfrage der Zinssaetze
    cout << k1->get_inhaber() << "s Zinssatz ist: " << k1->get_zinssatz() << endl;
    cout << k2->get_inhaber() << "s Zinssatz ist: " << k2->get_zinssatz() << endl;

    // Loesche Konten, ueberpruefe Anzahl
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
    delete k1;
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
    delete k2;
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
}
```

```
$ ./demo_spar
Eintrag in Sparbuch #123 (in Euro): 50.79 -> Neuer Saldo: 1050.79
Eintrag in Sparbuch #456 (in Euro): -6000 -> Neuer Saldo: 4000
FEHLER: Guthaben reicht nicht aus!
Joes Kontostand ist: 1050.79
Janes Kontostand ist: 4000
Joes Zinssatz ist: 0
Janes Zinssatz ist: 1.5
Anzahl der Konten: 2
Anzahl der Konten: 1
Anzahl der Konten: 0
```

Abbildung 2.30.: Implementierung und Ausgabe des Programms zur Verwendung der Klasse Sparkonto.

Nun implementieren wir eine weitere Kindklasse der Klasse “Konto”, die Klasse “Girokonto”. Diese hat zusätzlich zu den Attributen der Elternklasse, das Attribut `kreditrahmen`. Dieses wird im Konstruktor initialisiert. Ferner implementieren wir die Methode `ueberweisen()`, die als Übergabeparameter eine Instanz der Klasse Konto, und einen Betrag erwartet. Zunächst wird überprüft, ob der Kreditrahmen für die Transaktion ausreicht. Falls ja, wird der Betrag vom Konto abgezogen und dem angegebenen Konto gutgeschrieben. Die Gutschreibung geschieht über den Aufruf der `einzahlen()`-Methode des angegebenen Kontos (Abb. 2.31).

Programmbeispiel: Defintion von *Girokonto* in girokonto.h

```
#include "konto.h"

// Ein 'Girokonto' ist ein 'Konto' mit (eingeschraenktem) Ueberziehungsrahmen.
class Girokonto : public Konto
{
public:

    Girokonto(string name, long nummer, double betrag, double kredit)
    : Konto (name, nummer, betrag), kreditrahmen(kredit)
    {}

    void ueberweisen(Konto *k, double betrag);

protected:
    double kreditrahmen;
};



#endif /* GIROKONTO_H_ */
```

Abbildung 2.31.: Die Klasse Girokonto (Kindklasse von Konto).

Programmbeispiel: Implementierung von *Girokonto* in girokonto.cpp

```
using namespace std;

void Girokonto::ueberweisen(Konto *k, double betrag)
{
    // Wurde ein Konto angegeben?
    if (k==NULL)
    {
        cout << "FEHLER: Kein Empaengerkonto angegeben!" << endl;
    }

    // Ueberweise, falls
    if (betrag <= Konto::kontostand() + kreditrahmen)
    {
        this->auszahlen(betrag);
        k->einzahlen(betrag);
    }
    else {
        cout << "FEHLER: Kreditrahmen reicht nicht aus!" << endl;
    }
}
```

Abbildung 2.32.: Die Klasse Girokonto (Kindklasse von Konto).

Nun ist es uns möglich Giro- und Sparkonten anzulegen und auf diese Geld einzuzahlen, bzw. von diesen Geld auszuzahlen. Außerdem können wir Geld von Girokonten auf andere Girokonten, aber auch Sparkonten überweisen (Abb. 2.33):

Programmbeispiel: Verwendung der Klasse Girokonto in demo_giro.cpp

```
#include "girokonto.h"

using namespace std;


int main()
{
    // Setze Anzahl der Konten zurueck
    Konto::anzKont = 0;

    // Erzeuge Giro- und Sparkonto
    Konto *spar = new Sparkonto("Joe", 123, 10000.0);
    Girokonto *giro  = new Girokonto("Jane", 90456, 1057.0, 500.0);

    // Ausfuehren einer Ueberweisung
    giro->ueberweisen(spar, 1000);
    giro->ueberweisen(spar, 1000);              // Erzeugt Laufzeitfehler
    // spar->ueberweisen(giro, 1000);           // Erzeugt Compilerfehler!

    // Gekapselte Abfrage der Daten
    cout << spar->get_inhaber() << "s Kontostand ist: " << spar->kontostand() << endl;
    cout << giro->get_inhaber() << "s Kontostand ist: " << giro->kontostand() << endl;

    // Loesche Konten, ueberpruefe Anzahl
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
    delete giro;
    delete spar;
    cout << "Anzahl der Konten: " << Konto::anzKont << endl;
}
```

```
$ ./demo_giro
Eintrag in Sparbuch #123 (in Euro): 1000 -> Neuer Saldo: 11000
FEHLER: Kreditrahmen reicht nicht aus!
Joes Kontostand ist: 11000
Janes Kontostand ist: 57
Anzahl der Konten: 2
Anzahl der Konten: 0
```

Abbildung 2.33.: Ausgabe des Programms zur Verwendung der Klasse Girokonto

Dabei wird in der `ueberweisen()`-Methode der "Girokonto"-Klasse die `einzahlen()`-Methode der Klasse, dem das Konto angehört, aufgerufen. Je nachdem, ob es sich dabei um ein Girokonto oder ein Sparkonto handelt, wird die `einzahlen()`-Methode der Klasse "Sparkonto" oder die `einzahlen()`-Methode der Klasse "Konto" (denn "Girokonto" hat keine Methode `einzahlen()`) ausgeführt. Die Methode `ueberweisen()` arbeitet also mit Instanzen der Klasse "Konto", aber auch mit Instanzen aller, von "Konto" abgeleiteten Klassen. Die Ausführung der Verhaltensweise `einzahlen()` ist abhängig von der Klasse der jeweiligen Instanz und wird erst zur Laufzeit des Programms aufgelöst. Dies nennt man auch *Polymorphie* (Vielgestaltigkeit)

Polymorphie

Die objektorientiert Programmierung bietet viele Möglichkeiten und kann den Softwareentwurf, vor allem in der frühen Phase, deutlich vereinfachen. Allerdings trifft man immer wieder auf Probleme, die keine saubere objektorientierte Lösung bieten. Ferner bilden sich Kontrollflüsse in objektorientierten Programmen nicht mehr direkt in den Codestrukturen ab und sind daher

für den Entwickler weniger transparent. Mit der wachsenden Bedeutung von paralleler Hardware und nebenläufiger Programmierung ist aber gerade eine bessere Kontrolle und Entwickeler-Transparenz der komplexer werdenden Kontrollflüsse immer wichtiger. Ob sie sich objektorientierte Programmierung für das zu lösende Problem eignet, ist den Vorlieben des/der Softwareentwickler/in überlassen.

3. Aussagenlogik

In der Informatik ist Logik ein zentraler Bestandteil hinsichtlich der korrekten Argumentation und des korrekten Beweisens von Aussagen. Beispielsweise findet sich in Programmiersprachen ihre Unentbehrlichkeit bei der Benutzung von Fallunterscheidungen (`if-then-else`-Konstrukte) wieder. Neben dieser *Kontrollflusssteuerung in Computerprogrammen* gibt es zahlreiche weitere wichtige und nützliche Anwendungen der Aussagenlogik:

- Modellierung von Wissen (z.B. künstliche Intelligenz)
- Auswertung von Datenbankanfragen
- Logikbauteile in der technischen Informatik (Hardware)
- Automatische Verifikation (automatisches Testen eines Systems auf dessen Funktionstüchtigkeit)
- Mathematische Beweise
- Korrektes Argumentieren

Im Folgenden möchten wir die fundamentalen Bestandteile der Aussagenlogik kurz vorstellen und erläutern, wie man Wissen formal mit dem Werkzeug der Logik beschreiben kann.

3.1. Einführung

Bevor wir die Formalismen einführen, müssen wir uns zunächst klar darüber werden, **was** genau eigentlich Aussagen sind, mit denen wir arbeiten können.

Definition 3.1.
Eine *logische Aussage* besteht aus einer oder mehreren Teilaussagen, denen die Wahrheitswerte **0** (für `falsch`) oder **1** (für `wahr`) zugeordnet werden können. Eine Teilaussage nimmt dabei niemals gleichzeitig mehrere Werte an.

Aussage
Wahrheitswert

Mit welchem Wert man dabei eine Teilaussage belegt, hat keinen Einfluss darauf, ob man etwas „Aussage“ nennt oder nicht - lediglich die Eigenschaft, dass man es **kann**, ist der Indikator einer korrekten logischen Aussage.

Beispiel 3.2.
Betrachte folgende Ausdrücke, welche eine *logische Aussage* repräsentieren:

- Die Sonne scheint.
- Es ist hell.
- Der Tisch ist blau.

Offensichtlich können diese Ausdrücke **entweder** wahr **oder** falsch sein und erfüllen damit Definiton 3.1. Vergleiche obige nun mit folgenden Ausdrücken, welche keine logische Aussage darstellen:

- „$5 + 5$“ , „$\frac{5}{7}$“ , „$3 \cdot 2$“ , usw.

- 15 ist eine schöne Zahl („schön“ ist für Zahlen nicht definiert)
- Fragen („Wie spät ist es?“) und Aufforderung („Lach mal!“)

*Ihnen können keine konkreten Wahrheitswerte (*0 *oder* 1*) zugeordnet werden, sodass sie Definition 3.1 nicht entsprechen.*

Im Weiteren möchten wir die fundamentalen Werkzeuge der Aussagenlogik vorstellen, wozu wir die **atomaren Aussagen**, die **Negation** $\neg$, die **Konjunktion** $\wedge$ (Ver*und*ung), die **Disjunktion** $\vee$ (Ver*oder*ung), die **Implikation** $\rightarrow$ und die **Biimplikation** $\leftrightarrow$ zählen.

atomare Aussage

Beispiel 3.3 (Atomare Aussagen)**.**
Die einfachsten *atomaren Aussagen* sind **0** (ein Ausdruck, der immer falsch ist) und **1** (ein Ausdruck, der immer wahr ist). Allgemeiner ist das eine Aussage mit der Eigenschaft, nicht weiter zerlegt werden zu können. Weiter ist es bei der Formalisierung eines sprachlichen Satzes notwendig zu erkennen, welches die Teilaussagen sind.
Der Satz „**Wenn** ich Hunger habe **und** der Supermarkt geöffnet hat, **dann** gehe ich einkaufen.“ , den wir als φ bezeichnen, besitzt drei Teilaussagen:

- $A :=$ „Ich habe Hunger.“
- $B :=$ „Der Supermarkt hat geöffnet.“
- $C :=$ „Ich gehe einkaufen.“

Notation 3.4.
Atomare Aussagen kürzen wir immer mit den Buchstaben A, B, …, Z ab.

Rein formal hat unser obiger Satz die Form

$$\varphi = ((A \wedge B) \rightarrow C)$$

Lies: Wenn A und B wahr sind, dann ist auch C wahr.

Wie lässt sich nun erkennen, wann φ eine wahre Aussage und wann φ eine falsche Aussage darstellt? Dazu muss man sich die möglichen Belegungen von A, B und C ansehen. Das macht man meist mit einer Wahrheitstabelle (Wahrheitstafel). Wir werden später hierauf nochmal genauer eingehen, wenn wir die Bedeutung (Semantik) einer Konjunktion und einer Implikation behandelt haben.

Anmerkung 3.5.

Belegung

Eine *Belegung* für eine aussagenlogische Variable A (bzw. atomare Aussage) ist entweder 0 oder 1, d.h. A kann den Wert 0 (falsch) oder den Wert 1 (wahr) annehmen.

Negation

$\neq$

Beispiel 3.6 (Negation / Verneinung $\neg$)**.**
Jede logische Aussage A lässt sich auch in ihrer negierten Form $\neg A$ darstellen. Dabei gilt, dass $\neg A$ **genau dann** wahr ist, **wenn** A falsch ist. Betrachten wir hierzu folgenden Satz φ: „Alle Kinder spielen gern.“
Dann ist

$$\neg\varphi = \text{„Nicht alle Kinder spielen gern.“}$$

Umgangssprachlich bedeutet $\neg\varphi$, dass es (mindestens) ein Kind gibt, das nicht gerne spielt.
Des Weiteren ist bei der Verneinung von Aussagen zu beachten, an welcher Stelle negiert werden muss. Es wäre falsch zu sagen, dass $\neg\varphi$ der Aussage „Alle Kinder spielen ungern.“ entspricht.

Beispiel 3.7 (Konjunktion $\wedge$)**.**

Konjunktion $\wedge$

Es ist möglich mehrere Teilaussagen miteinander in Form einer Ver*und*ung zu verknüpfen, sodass die dadurch entstehende komplexere Aussage genau dann wahr ist, wenn beide Teilaussagen gleichzeitig wahr sind.
Der Satz φ „Die Sonne scheint **und** der Wind weht stark.“ ist nur dann wahr, wenn sowohl die Aussage A „Die Sonne scheint.“ als auch die Aussage B „Der Wind weht stark.“ wahr ist.

Formal entspricht obiger Satz der Konjunktion von A und B, also

$$\varphi = A \wedge B$$

Beispiel 3.8 (Disjunktion $\vee$)**.**

Disjunktion $\vee$

Analog zur in Beispiel 3.7 eingeführten Konjunktion ist auch eine Ver*oder*ung zweier Teilaussagen A und B möglich. Die dadurch entstehende komplexere Aussage ist genau dann wahr, wenn mindestens eine der Teilaussagen wahr ist.
Betrachten wir den Satz φ „Der Vorhang ist rot oder der Koffer ist blau.“, welcher dann wahr ist, wenn der Vorhang rot ist (A) **oder** der Koffer blau ist (B) **oder** beides der Fall ist.

Formal entspricht obiger Satz der Disjunktion von A und B, also

$$\varphi = A \vee B$$

Hinweis 3.9.
Eine Disjunktion ist **keine** ausschließende Veroderung im Sinne von „entweder ... oder ..., aber nicht beides zusammen“. Hierzu verwendet man in der Aussagenlogik das `exklusive Oder` (formal: $A \oplus B$ bzw. $A \dot{\vee} B$), welches wir im Vorkurs nicht ausführlich behandeln, sondern nur kurz ansprechen.

Beispiel 3.10 (Implikation $\rightarrow$)**.**

Implikation $\rightarrow$

Unter einer Implikation versteht man die Folgerung des Sachverhalts B aus einem Sachverhalt A heraus (wie in Beispiel 3.3, in dem wir eine Implikation bereits behandelt haben). Man hat also eine Voraussetzung, die eine Konsequenz zur Folge hat.

Der Satz φ „**Wenn** es regnet, **dann** ist die Straße nass.“ wird wahr, wenn beide Teilaussagen wahr sind oder die Voraussetzung falsch ist.

Zur Verdeutlichung: Wenn es regnet und die Straße nass ist, dann ist φ logischerweise wahr. Allerdings kann die Straße auch nass sein, wenn es nicht regnet - dann ist unsere Folgerung trotzdem richtig. Umgekehrt wird φ falsch, falls es regnet, die Straße aber nicht nass ist - denn dann ist die Voraussetzung zwar erfüllt, aber die Konsequenz tritt nicht ein.

Als Leitsatz kann man sich Folgendes merken: *Aus Wahrem kann man nur Wahres folgern - aus Falschem kann man alles folgern.*
Entsprechend sind auch die Formalismen einer Implikation φ „Aus A folgt B“ bzw. „Wenn A, dann B“ klar:

$$\varphi = (A \rightarrow B) \quad \text{oder auch} \quad \varphi = (\neg A \vee B)$$

Beispiel 3.11 (Biimplikation $\leftrightarrow$)**.**

Biimplikation $\leftrightarrow$

Eine Biimplikation ist, wie sich aus dem Namen schließen lässt, eine Implikation in beide Richtungen. Sprachlich wird das durch das Konstrukt **„... genau dann, wenn ...“** deutlich.
Der Satz φ „Der Schornstein raucht **genau dann, wenn** die Heizung an ist.“ wird wahr, wenn sowohl die Aussage „Der Schornstein raucht, wenn die Heizung an ist.“ $(A \rightarrow B)$ als auch die Aussage „Die Heizung ist an, wenn der Schornstein raucht.“ $(B \rightarrow A)$ wahr ist.
Formal entspricht φ also einer Biimplikation, das heißt:

$$\varphi = (A \leftrightarrow B) \quad \text{oder auch} \quad \varphi = ((A \rightarrow B) \wedge (B \rightarrow A))$$

Notation 3.12.
Die Zeichen $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ heißen *Junktoren*, mit denen wir Teilaussagen zu komplexen Aussagen verknüpfen können.

Junktoren

Nachdem wir nun die wichtigsten Konstrukte der Aussagenlogik eingeführt haben, können wir einige größere Aussagen formalisieren.

Aufgabe 3.1.
Finde die Teilaussagen des folgenden Satzes und stelle eine äquivalente aussagenlogische Formel auf: „Ich fahre an der Ampel genau dann los, wenn die Ampel grün und die Straße frei ist.“

Lösung:
Die Grundidee bei der Lösungsfindung ist prinzipiell immer gleich: Man sucht nach bestimmten Schlüsselwörtern: [**und**], [**oder**], [**entweder** ... **oder**], [**wenn** ..., **dann** ...], [... **genau dann, wenn** ...] und [**nicht**]. Hat man solche Wörter gefunden, so muss man die Teilaussagen links bzw. rechts davon nur noch aufsplitten und anschließend entsprechend des verknüpfenden Schlüsselworts mit dem richtigen Junktor verbinden.

Wir haben drei Teilaussagen. Setze

$$A := \text{„Ich fahre an der Ampel los.“}$$
$$B := \text{„Die Ampel ist grün.“}$$
$$C := \text{„Die Straße ist frei.“}$$

Die aussagenlogische Formel lautet: $(A \leftrightarrow (B \wedge C))$

Aufgabe 3.2.
Finde die Teilaussagen des folgenden Satzes und stelle eine äquivalente aussagenlogische Formel auf: „Wenn ich keinen Spicker habe und den Multiple-Choice-Test bestehen möchte, dann muss ich die Antworten wissen oder Glück beim Raten haben.“

Lösung:
Wir haben vier Teilaussagen. Setze

$$A := \text{„Ich habe einen Spicker.“}$$
$$B := \text{„Ich möchte den Multiple-Choice-Test bestehen.“}$$
$$C := \text{„Ich muss die Antworten wissen.“}$$
$$D := \text{„Ich muss Glück beim Raten haben.“}$$

Die aussagenlogische Formel lautet: $((\neg A \wedge B) \rightarrow (C \vee D))$

Hinweis: Man hätte auch $A :=$ „Ich habe keinen Spicker.“ setzen können, sodass man als aussagenlogische Formel $((A \wedge B) \rightarrow (C \vee D))$ erhält, was auch eine korrekte Lösung darstellt.

3.2. Aussagenlogik

In diesem Abschnitt möchten wir die in der Einführung erläuterten Begriffe, die wir zunächst anhand von Beispielen behandelt haben, exakt definieren. Dazu legen wir die Syntax (was genau sind eigentlich gültige aussagenlogische Formeln?) und die Semantik (welche Bedeutung haben bestimmte Konstrukte?) fest.

3.2.1. Syntax der Aussagenlogik

Syntax

Die Syntax beschreibt **gültige** (aussagenlogische) Formeln und schließt implizit ungültige aus. Zum besseren Verständnis kann man die Syntax mit dem korrekten Satzbau in der deutschen Sprache vergleichen: „Subjekt, Prädikat, Objekt“ist erlaubt (z.B. „Alfons mäht den Rasen.“), wohingegen „Objekt, Subjekt, Prädikat“nicht erlaubt ist („Den Rasen Alfons mäht.“).

In unserem Kontext möchten wir beispielsweise $(A \rightarrow B)$ zulassen und $(\rightarrow A)B$ verbieten.

Definition 3.13.
Gegeben ist eine unendliche Menge aussagenlogischer Variablen A, ..., Z, A_1, ..., wobei eine aussagenlogische Variable alleine eine gültige Formel darstellt. Komplexere Formeln werden aus den Zeichen (und) sowie den in Notation 3.12 aufgezählten Junktoren $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ nach folgenden Konstruktionsregeln erstellt:

Basisfall:
Sowohl **aussagenlogische Variablen**, als auch **0** und **1**, sind gültige Formeln.

Rekursionsfälle:

1. Ist A eine gültige Formel, so ist auch $\neg A$ eine gültige Formel.
2. Sind A und B gültige Formeln, so ist auch $(A \wedge B)$ eine gültige Formel.
3. Sind A und B gültige Formeln, so ist auch $(A \vee B)$ eine gültige Formel.
4. Sind A und B gültige Formeln, so ist auch $(A \rightarrow B)$ eine gültige Formel.
5. Sind A und B gültige Formeln, so ist auch $(A \leftrightarrow B)$ eine gültige Formel.

Beispiel 3.14.
Gemäß Definition 3.13 ist $(A \leftrightarrow (B \wedge C))$ eine gültige Formel, wobei $(A \vee B \vee C)$ (Klammern fehlen) und $(A \leftarrow B)$ („$\leftarrow$“ nicht definiert) keine gültigen Formeln sind.

3.2.2. Semantik der Aussagenlogik

In der Semantik möchten wir die Bedeutung einzelner Konstrukte festlegen, was prinzipiell nichts Anderes ist, als Regeln zu definieren, die uns sagen, wann eine aussagenlogische Formel in Abhängigkeit der Belegungen ihrer Variablen wahr wird. Wie bereits oben angesprochen werden wir das anhand von **Wahrheitstabellen** (Wahrheitstafeln) machen. In den Spalten links neben dem Doppelstrich stehen alle aussagenlogischen Variablen, während rechts neben dem Doppelstrich die zu untersuchende Formel auftritt. In den Zeilen werden dann alle möglichen Belegungen der Variablen aufgeführt und in Abhängigkeit davon die Formel ausgewertet. Die sprachlichen Begründungen finden sich im Abschnitt 3.1, sodass wir im Folgenden nur noch die exakten Wahrheitswerte auflisten.

Definition 3.15.
Gegeben sind zwei aussagenlogische Variablen A und B. Folgende Wahrheitstabellen zeigen abhängig von den Belegungen von A und B die Auswertung für die rechts neben dem Doppelstrich stehenden Formeln:

A	$\neg A$
0	1
1	0

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

Anmerkung 3.16.
Das exklusive Oder (XOR) $A \oplus B$ lässt sich auch mit $((\neg A \wedge B) \vee (A \wedge \neg B))$ oder auch mit $((A \vee B) \wedge \neg(A \wedge B))$ darstellen.

Wie geht man nun an die Berechnung einer größeren Formel heran? Als nützlich und vor allem übersichtlich erweist es sich immer, die komplexe Formel in ihre Teilformeln zu zerlegen und schrittweise weiterzurechnen. Wie genau das vonstatten geht, wollen wir anhand von Beispiel 3.3 sowie Aufgabe 3.1 und Aufgabe 3.2 zeigen.

Beispiel 3.17.
Wir fragen uns, mit welchen Belegungen die in Beispiel 3.3 angeführte Formel $\varphi = ((A \wedge B) \to C)$ („**Wenn** ich Hunger habe **und** der Supermarkt geöffnet hat, **dann** gehe ich einkaufen.“) wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

A	B	C	$(A \wedge B)$	$((A \wedge B) \to C)$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Offensichtlich ist unsere Formel φ **bis auf die Belegung** $A = 1, B = 1, C = 0$ immer wahr.

Dies lässt sich leicht aus der links stehenden Wahrheitstabelle ablesen.

Die Aufspaltung der Formel hat die Berechnung erleichtert.

Beispiel 3.18.
Wir fragen uns, mit welchen Belegungen die in Aufgabe 3.1 angeführte Formel $\varphi = (A \leftrightarrow (B \wedge C))$ wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

A	B	C	$(B \wedge C)$	$(A \leftrightarrow (B \wedge C))$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Betrachte die Zeilen, bei denen $A = 1$ („Ich fahre an der Ampel los“): Die Formel wird nur dann war, wenn die Ampel grün und die Straße frei ist.

Wenn $A = 0$ („Ich fahre nicht an der Ampel los“), dann stimmt die Formel logischerweise unabhängig von B und C, es sei denn, $B = C = 1$ (also Ampel ist grün und Straße ist frei), denn genau dann, wenn ich dann nicht losfahre ($A = 0$), stimmt unsere Formel nicht mehr.

Beispiel 3.19.
Wir fragen uns, mit welchen Belegungen die in Aufgabe 3.2 angeführte Formel $\varphi = ((\neg A \wedge B) \rightarrow (C \vee D))$ wahr wird. Dazu stellen wir die Wahrheitstabelle auf:

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

Die richtige Interpretation der Werte überlassen wir dem Leser.

Satz 3.20.
Die Wahrheitstabelle für eine aussagenlogische Formel mit insgesamt n Variablen hat genau 2^n Zeilen.

Der Beweis hierzu findet sich in Kapitel 8.1.

3.2.3. Erfüllbarkeit, Allgemeingültigkeit und Äquivalenz

In diesem Abschnitt möchten wir aussagenlogische Formeln genauer klassifizieren und miteinander vergleichen. Dazu führen wir folgende Begriffe ein:

Definition 3.21.
Gegeben ist eine aussagenlogische Formel φ.

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

(*b*) φ heißt *Kontradiktion* (oder *unerfüllbar*), wenn alle Belegungen der Variablen dazu führen, dass die Formel den Wahrheitswert 0 hat. — Kontradiktion

(*c*) φ heißt *Tautologie* (oder *allgemeingültig*), wenn jede Belegung der Variablen dazu führt, dass die Formel den Wahrheitswert 1 hat. — Tautologie

Beispiel 3.22. (*a*) Die Formel $((A \wedge B) \rightarrow C)$ ist nach Beispiel 3.17 erfüllbar (aber nicht allgemeingültig).

(*b*) Die Formel $(A \wedge \neg A)$ ist eine Kontradiktion (unerfüllbar), genauso wie die Formel $((A \rightarrow B) \wedge (A \wedge \neg B))$.

(*c*) Die Formel $(A \vee \neg A)$ ist eine Tautologie (allgemeingültig), genauso wie die Formel $((A \vee B) \vee (\neg A \wedge \neg B))$.

Wie erkennt man, welcher „Klasse“ eine gegebene aussagenlogische Formel angehört? Einer komplexeren Formel sieht man das meistens nicht direkt an! Letztendlich bleibt nichts anderes übrig, als alle möglichen Belegungen durchzurechnen.

Anmerkung 3.23. (*a*) Eine gegebene Formel φ ist genau dann **erfüllbar**, wenn in der Wahrheitstabelle für φ mindestens eine 1 steht.

(*b*) Eine gegebene Formel φ ist genau dann eine **Kontradiktion** (unerfüllbar), wenn in der Wahrheitstabelle für φ ausschließlich 0 stehen.

(*c*) Eine gegebene Formel φ ist genau dann eine **Tautologie** (allgemeingültig), wenn in der Wahrheitstabelle für φ ausschließlich 1 stehen.
Eine gegebene Formel φ ist genau dann eine **Tautologie** (allgemeingültig), wenn $\neg\varphi$ eine Kontradiktion (unerfüllbar) ist.

Mittlerweile ist es uns möglich, aussagenlogische Formeln in gewisse „Klassen“ einzuteilen. Folgend möchten wir nun zwei gegebene Formeln α und β miteinander vergleichen und feststellen, ob diese äquivalent sind.

Definition 3.24.
Äquivalenz

Gegeben sind zwei aussagenlogische Formeln α und β, wobei M die Menge der Variablen in α und N die Menge der Variablen in β ist. Wir sagen, dass α **äquivalent zu** β ist („$\alpha \equiv \beta$“), wenn für alle Belegungen der Variablen in $M \cup N$ der Wahrheitswert von α mit dem Wahrheitswert von β übereinstimmt.

Es stellt sich nach dieser Definition nun die Frage, wie man die Äquivalenz überprüfen kann. Der relativ aufwändige Ansatz besteht wieder darin, die Wahrheitstabellen aufzustellen und die Spalten von α und β auf Gleichheit zu überprüfen.

Beispiel 3.25.
Wir betrachten die beiden Formeln $\alpha = (A \wedge (A \vee B))$ mit $M = \{A, B\}$ und $\beta = A$ mit $N = \{A\}$. Dann ist $M \cup N = \{A, B\}$. Die Wahrheitstabelle sieht wie folgt aus:

A	B	$(A \vee B)$	α	β
0	0	0	0	0
0	1	1	0	0
1	0	1	1	1
1	1	1	1	1

Offensichtlich stimmen die Werte in den Spalten von α und β überein, sodass wir nach Definition 3.24 sagen können, dass $\alpha \equiv \beta$ gilt.

3.2.4. Fundamentale Rechenregeln

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

Satz 3.26.
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***

- $(A \to B) \equiv (\neg A \vee B)$

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

- $(A \leftrightarrow B) \equiv ((A \to B) \wedge (B \to A)) \equiv ((\neg A \vee B) \wedge (\neg B \vee A))$

Anmerkung 3.27.
Die Rechenregeln aus Satz 3.26 lassen sich wie in 3.26(d) durch Aufstellen der Wahrheitstabellen einfach nachprüfen. Entsprechende Aufgaben für 3.26(e) und 3.26(f) finden sich auf dem Übungsblatt.

4. Mengen

In diesem Kapitel stellen wir Mengen und einige ihrer grundlegenden Eigenschaften vor. Zunächst werden wir genau definieren, was wir unter einer *Menge* verstehen wollen. Zum besseren Verständnis geben wir oft Beispiele an. Dann können wir, nur unter Zugrundelegen dieser Definition, einige Eigenschaften von Mengen zeigen. Wir formulieren sie erst als konkrete Behauptung, auch *Satz* genannt. Weniger umfangreiche bzw. leichter zu beweisende Sätze werden hin und wieder auch als *Lemma* oder *Proposition* bezeichnet. Den Satz beweisen wir anschließend mit Hilfe der Methoden, die wir im Logik-Kapitel (3) dieses Skripts bereits kennen gelernt haben, und unter Rückgriff auf die eingeführten Definitionen. Außerdem dürfen wir alle Voraussetzungen in unserer Argumentation benutzen, die im Satz auftauchen. Lautet dieser beispielsweise

„Jede endliche Menge ist in einer anderen endlichen Menge enthalten“, so können wir uns im Beweis auf endliche Mengen beschränken und deren Endlichkeit auch für Schlussfolgerungen benutzen.

Definition 4.1.
Eine *Menge* ist eine Zusammenfassung von wohlunterschiedenen Objekten. Letztere nennen wir *Elemente* der Menge. Hierbei schreiben wir $m \in M$, um auszudrücken, dass m ein Element der Menge M ist. Gleichermaßen drücken wir mit $m \notin M$ aus, dass m kein Element der Menge M ist. Für $a \in M$ und $b \in M$ schreiben wir kurz $a, b \in M$.

Menge
Element
$\in$
$\notin$

Der Begriff *wohlunterschieden* in der obigen Definition besagt, dass die Elemente verschieden sind und von uns auch als voneinander verschieden angesehen werden. Zum Beispiel sind die Farben „Dunkelrot“ und „Hellrot“ an sich verschieden, könnten aber, wenn wir nur gröber unterscheiden wollten, als gleich angesehen werden, da sie ja beide Rottöne sind. Wenn beide in derselben Menge auftreten, wollen wir sie aber *nicht* als gleich bezeichnen. Zum Beispiel könnten wir eine Menge *Rottöne* definieren, darin würden etwa *Hellrot* und *Dunkelrot* als (wohlunterschiedene) Elemente auftreten.

Die Elemente einer Menge müssen aber nicht gleichartig sein. So können Personen, Zahlen, Namen, Farben allesamt gemeinsam in einer Menge auftreten, wenn wir diese so definieren. Mengen können *implizit* oder *explizit* angegeben werden. Letzteres geschieht durch Aufzählung aller Elemente, dabei öffnet man vor dem ersten genannten Element eine geschweifte Klammer { und schließt nach dem letzten eine }. Um Mengen implizit anzugeben, kann man sie mit Worten oder wie in der Definition für $\mathbb{Q}$ im folgenden Beispiel beschreiben.

explizit
implizit

Beispiel 4.2.

Implizit angegebene Mengen:

- die Menge aller Buchstaben des deutschen Alphabets
- die Menge aller Primzahlen
- die Menge aller von Frankfurt am Main aus angeflogenenen Ziele
- die Menge aller Einwohner Deutschlands am 24.08.2013
- die Menge der rationalen Zahlen $\mathbb{Q} := \{\frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0\}$

- die Menge der reellen Zahlen $\mathbb{R}$
- $M := \{z \in \mathbb{Z} \mid \text{es gibt ein } k \in \mathbb{N} \text{ mit } z = 5 \cdot k\}$

Explizit angegebene Mengen:

- die leere Menge $\emptyset$ (Menge, die keine Elemente enthält. Andere Schreibweise: $\{\}$)
- $P := \{\text{ich, du, er, sie, es, wir, ihr, Sie}\}$
- $\mathbb{N} := \{0, 1, 2, 3, \dots\}$ (Menge der natürlichen Zahlen)
- $\mathbb{Z} := \{0, 1, -1, 2, -2, 3, -3, \dots\}$ (Menge der ganzen Zahlen)

Der einzelne vertikale Strich in einer Menge wird als „sodass“ gelesen: Beispielsweise ist die oben angegebene Menge M die „Menge aller ganzen Zahlen z, *sodass* es eine natürliche Zahl k mit $z = 5 \cdot k$ gibt“, d.h. M ist die Menge aller durch 5 teilbaren Zahlen.

Aufgabe 4.1.
Gib die Menge aller Primzahlen unter Verwendung des vertikalen Strichs an. Versuche, eine ähnliche Darstellung für die Menge aller geraden Zahlen zu finden.

Wenn wir Sätze über Mengen formulieren wollen, müssen wir die betreffenden Mengen miteinander vergleichen können. Dafür brauchen wir die folgende Definition.

Definition 4.3.
Seien M und N zwei Mengen.

Gleichheit
(a) Wir bezeichnen M und N als *gleich* (in Zeichen $M = N$), wenn M und N dieselben Elemente enthalten.

Teilmenge
(b) M ist eine *Teilmenge* von N (in Zeichen $M \subseteq N$), wenn jedes Element von M auch ein Element von N ist.

(c) M ist eine *echte Teilmenge* von N (in Zeichen $M \subsetneq N$), wenn $M \subseteq N$, aber nicht $M = N$ gilt.

Obermenge
(d) M ist eine *Obermenge* von N (in Zeichen $M \supseteq N$), wenn $N \subseteq M$ gilt.

(e) M ist eine *echte Obermenge* von N (in Zeichen $M \supsetneq N$), wenn $N \subsetneq M$ gilt.

Gilt für zwei Mengen M und N Gleichheit, so identifizieren wir sie miteinander. Wir sagen dann, es handelt sich um *dieselbe* Menge, und können sie je nach Kontext und Belieben mit M oder N bezeichnen. Genauso identifizieren wir Elemente von Mengen miteinander: Gibt es ein x, was Element einer Menge M und auch Element einer Menge N ist, so sprechen wir nicht von „dem x aus M“ einerseits und „dem x aus N“ andererseits, sondern von „dem x, was Element von M und auch von N ist“ bzw. „dem x, was gemeinsames Element von M und N ist“.

Anstelle von $M \subsetneq N$ bzw. $M \supsetneq N$ findet man auch oft $M \subset N$ bzw. $M \supset N$ in der Literatur. Manchmal ist damit aber auch $M \subseteq N$ bzw. $M \supseteq N$ gemeint. Um Missverständnissen vorzubeugen, wählen wir die eindeutige Schreibweise aus Definition 4.3.

Beispiel 4.4.
Wir können uns schnell überzeugen, dass die folgenden Beziehungen wahr sind:

- $M = M$ und $\emptyset \subseteq M$ für jede Menge M
- $M \subseteq M$ (und $M \supseteq M$) für jede Menge M

- $\{\text{ich, du}\} \subset \{\text{ich, du, er, sie, es, wir, ihr, Sie}\}$
- $\{1, 2\} = \{2, 1\} = \{1, 2, 2\}$
- $\mathbb{N} \subseteq \mathbb{Z}$, sogar $\mathbb{N} \subsetneq \mathbb{Z}$
- $\mathbb{Z} \subseteq \mathbb{Q}$, sogar $\mathbb{Z} \subsetneq \mathbb{Q}$
- $\mathbb{Q} \subseteq \mathbb{R}$, sogar $\mathbb{Q} \subsetneq \mathbb{R}$

Reihenfolge

Das vierte Beispiel oben betont, dass die Gleichheit von Mengen nicht von der Reihenfolge ihrer Elemente abhängt. Das heißt, eine Menge ist durch ihre Elemente eindeutig identifiziert – es gibt nur *eine einzige* Menge, die genau diese Elemente und keine weiteren beinhaltet. Außerdem wird in der expliziten Schreibweise normalerweise jedes Element nur einmal erfasst. Wir schreiben also $\{1, 2\}$ und nicht $\{1, 2, 2\}$. Dies liegt daran, dass die Elemente einer Menge nach Definition 4.1 wohlunterschieden sind. Die beiden Zweier in $\{1, 2, 2\}$ können wir aber nur durch ihre Position in der Menge unterscheiden – weil aber die Reihenfolge keine Rolle spielt, geht auch das nicht. Also ist $\{1, 2, 2\}$ in Einklang mit Definition 4.3 die (eindeutig definierte) Menge, die genau die 1 und die 2 enthält, nämlich $\{1, 2\}$.

Aufgabe 4.2.
Gibt es zwei Mengen M und N, sodass $M \subsetneq N$ und $N \subsetneq M$ gilt? Wenn ja, welche? Wenn nein, warum nicht?

Wie sieht es mit $M \subsetneq N$ und $N \subseteq M$ aus? Kannst du hierfür Beispiele finden?

Nun haben wir alle Werkzeuge, um unseren ersten Satz zu beweisen. Er besagt umgangssprachlich: Zwei Mengen sind identisch, wenn die erste in der zweiten und die zweite in der ersten enthalten ist. Wenn nicht, sind die Mengen auch nicht gleich.

Genau-dann-wenn-Aussagen

Hierbei handelt es sich um eine Genau-dann-wenn-Aussage: *Wenn* die erste in der zweiten und die zweite in der ersten enthalten ist, sind die Mengen gleich. Aber auch *immer wenn* zwei Mengen gleich sind, gelten die beiden Teilmengenbeziehungen. Gelten sie nicht, so können die Mengen nicht gleich sein. D.h. die Voraussetzung der beiden Teilmengenbeziehungen reicht nicht nur aus, um die Gleichheit der Mengen zu folgern, sondern sie ist auch notwendig dafür. Zum Beweis müssen wir zwei Schritte vollziehen: Wir müssen einmal aus den Teilmengenbeziehungen die Gleichheit folgern und einmal aus der Gleichheit die Teilmengenbeziehungen. Die Teilbeweise heben wir voneinander durch „⇒“ (für „*Wenn ...*, dann ...“) und „⇐“ (für „*Nur wenn ...*, dann ...“) ab.

Satz 4.5.
Seien M und N Mengen. M und N sind gleich, genau dann, wenn $M \subseteq N$ und $M \supseteq N$ gilt.

Beweis. Wir wollen eine Pauschalaussage über Mengen erhalten. D.h. wir dürfen uns nicht Mengen aussuchen, für die wir den Satz zeigen, sondern wir müssen ihn *für alle* Mengen M und N beweisen. Seien also M und N zwei *beliebige* Mengen.

Wir müssen beweisen, dass aus $M \subseteq N$ und $M \supseteq N$ die Gleichheit von M und N folgt *und* dass sich aus der Voraussetzung $M = N$ die beiden Teilmengenbeziehungen $M \subseteq N$ und $N \subseteq M$ ergeben.

„⇐“: Wir zeigen jetzt: Wenn $M \subseteq N$ und $M \supseteq N$, dann auch $M = N$.

Nach Definition 4.3 bedeutet $M \subseteq N$, dass jedes Element von M auch eines von N ist. Umgekehrt bedeutet $M \supseteq N$, dass $N \subseteq M$, d.h. jedes Element von N ist auch Element von M. Beides zusammengenommen heißt, dass M und N genau dieselben Elemente enthalten und wiederum nach Definition 4.3 schreiben wir dafür $M = N$ und das wollten wir zeigen.

„⇒“: Wir müssen noch beweisen: Wenn $M = N$, dann gilt auch $M \subseteq N$ und $M \supseteq N$.

Wir können also annehmen, dass M und N gleich sind, nach Definition 4.3 enthalten sie also genau dieselben Elemente. Dann ist natürlich jedes Element von M auch eines von N, also $M \subseteq N$, denn so haben wir es in Definition 4.3 festgelegt. Umgekehrt ist auch $N \subseteq M$, denn jedes Element von N ist auch in M enthalten, die Mengen enthalten ja dieselben Elemente! Aber $N \subseteq M$ ist gleichbedeutend mit $M \supseteq N$ und damit haben wir beide Teilmengenbeziehungen gezeigt. □

direkter Beweis

Die beiden Beweisteile für Satz 4.5 sind Beispiele für *direkte Beweise*: Aus den Voraussetzungen und Definitionen wurde sozusagen auf direktem Wege durch korrektes, exaktes Argumentieren die Behauptung gezeigt. Im nächsten Beweis werden wir noch eine andere Technik kennenlernen. Der zugehörige Satz besagt umgangssprachlich: Wenn eine Menge in einer anderen, „größeren“ und diese in einer dritten enthalten ist, so ist die erste auch in der dritten enthalten – und nicht mit dieser identisch. Zum Beispiel ist die Menge der Frankfurter Einwohner eine echte Teilmenge der Einwohner Hessens und diese wiederum eine Teilmenge der Einwohner der BRD. Auch ohne geographische Kenntnisse kann man daraus direkt folgern, dass alle Frankfurter Einwohner auch in der BRD sesshaft sein müssen und es in der BRD aber noch Einwohner gibt, die nicht in Frankfurt leben – nämlich mindestens die Hessen, die nicht in Frankfurt leben. Und die muss es ja geben, sonst wäre die Menge aller Frankfurter keine *echte* Teilmenge der Hessen. Aber haben wir die Definitionen so gewählt, dass sie mit unserer Intuition übereinstimmen? Es könnte ja sein, dass Definition 4.3 unvollständig ist bzw. so unpassend gewählt wurde, dass die beschriebene Schlussfolgerung damit nicht möglich ist. Wir werden jetzt *beweisen*, dass dem nicht so ist.

Satz 4.6.
Seien L, M und N Mengen. Wenn $L \subsetneq M$ und $M \subseteq N$ gilt, dann gilt auch $L \subsetneq N$.

Beweis. Hier müssen wir wieder zweierlei zeigen: Nach Definition 4.3 bedeuetet $L \subsetneq N$, dass $L \subseteq N$ und $L \neq N$, d.h. L und N sind verschieden.

Wir wollen Satz 4.6 wiederum *für alle* Mengen L, M, N, die den Voraussetzungen des Satzes genügen, beweisen. Also seien L, M und N beliebige Mengen, für die $L \subsetneq M$ und $M \subseteq N$ gilt.

Um daraus $L \subseteq N$ zu folgern, wollen wir uns davon überzeugen, dass jedes Element von L auch eines von N ist. Sei also $x \in L$ beliebig. Wegen $L \subsetneq M$ wissen wir mit Definition 4.3, dass $L \subseteq M$, d.h. jedes Element von L ist auch eines von M. Folglich ist $x \in M$. Genauso können wir aber wegen $M \subseteq N$ folgern, dass $x \in N$. Weil x beliebig war, muss also für *jedes* $x \in L$ gelten, dass $x \in N$. D.h. jedes Element von L ist auch eines von N und das ist nach Definition 4.3 gleichbedeutend mit $L \subseteq N$.

Wie können wir jetzt noch $L \neq N$ beweisen? Wir verwenden einen *Beweis durch Widerspruch*. Angenommen, L und N wären gleich. Dann hätten wir mit den Voraussetzungen die Beziehungen $L \subseteq M$ und $M \subseteq N = L$, d.h. $L \subseteq M$ und $M \subseteq L$ und mit Satz 4.5 wären dann L und M gleich. Das ist aber ein Widerspruch zur Voraussetzung, dass $L \subsetneq M$, insbesondere $L \neq M$ ist. Also muss unsere Annahme falsch gewesen sein, L und N sind nicht gleich! Damit sind wir fertig, denn beide Ergebnisse zusammen beweisen $L \subsetneq N$. □

Beweis durch Widerspruch

Beweise durch Widerspruch wie den obigen benutzt man oft, wenn man keine Idee für einen direkten Beweis hat: Man geht vom Gegenteil des zu Zeigenden aus und leitet daraus dann unter Verwendung der Voraussetzungen, Definitionen und bekannter Resultate einen eindeutigen Widerspruch her. Wenn das gelingt, weiß man, dass die Annahme falsch gewesen sein muss, denn alles andere am Beweis ist ja korrekt. Damit hat man also das Gewünschte bewiesen, denn sein Gegenteil hat sich als falsch erwiesen. Wir haben eben noch eine wichtige Technik für Mengenbeweise kennengelernt: Um Teilmengenbeziehungen zu zeigen, bietet es sich oft an, für ein *beliebiges*

Element der einen Menge seine Zugehörigkeit zur anderen zu zeigen, anstatt nur die Definitionen zu erläutern.

Aufgabe 4.3.
Versuche, auf ähnliche Art für alle Mengen L, M, N die folgenden Behauptungen zu beweisen:

- *Aus $L \subseteq M$ und $M \subsetneq N$ folgt $L \subsetneq N$.*
- *Aus $L \supseteq M$ und $M \supseteq N$ folgt $L \supseteq N$.*

Aus zwei Mengen kann man auf verschiedene Weise eine neue gewinnen. Einige Möglichkeiten, wie dies geschehen kann, beschreibt die folgende Definition.

Definition 4.7.
Seien M und N Teilmengen einer gemeinsamen Obermenge U.

(a) Der *Schnitt* von M und N ist die Menge — Schnitt

$$M \cap N := \{x \mid x \in M \text{ und } x \in N\},$$

also die Menge aller Elemente, die sowohl in M als auch in N auftreten. Wir bezeichnen M und N als *disjunkt*, wenn $M \cap N = \emptyset$, d.h. wenn sie keine Elemente gemeinsam haben.

(b) Die *Vereinigung* von M und N ist die Menge — Vereinigung

$$M \cup N := \{x \mid x \in M \text{ oder } x \in N\},$$

also die Menge aller Elemente, die in mindestens einer der beiden Mengen auftreten.

(c) Die *Differenz* von M und N ist die Menge — Differenz

$$M \backslash N := \{x \mid x \in M \text{ und } x \notin N\},$$

also die Menge aller Elemente von M, die nicht in N auftreten.

(d) Die *symmetrische Differenz* von M und N ist die Menge — symm. Differenz

$$M \Delta N := (M \backslash N) \cup (N \backslash M),$$

also die Menge aller Elemente, die entweder in M oder in N, aber nicht in beiden gleichzeitig sind.

(e) Das *Komplement* von M (bzgl. U) ist die Menge — Komplement

$$\overline{M} := U \backslash M,$$

also die Menge aller Elemente aus U, die nicht in M sind.

(f) Die *Potenzmenge* von M ist die Menge — Potenzmenge

$$\mathcal{P}(M) := \{N \mid N \subseteq M\},$$

also die Menge aller Teilmengen von M.

Beachte, dass das Komplement einer Menge nur definiert ist, wenn die Obermenge, auf die wir uns beziehen, klar ist! Da wir es unmissverständlich auch immer mit Hilfe einer Differenz ausdrücken können, verzichten wir im Folgenden auf seine explizite Verwendung.

Die Elemente einer Potenzmenge sind wiederum Mengen. Weil die leere Menge Teilmenge jeder beliebigen Menge M ist, gilt immer $\emptyset \in \mathcal{P}(M)$. Auch $M \in \mathcal{P}(M)$ ist immer wahr.

Um die soeben definierten Begriffe zu verstehen, geben wir den Schnitt, die Vereinigung, die Differenz, die symmetrische Differenz und die Potenzmenge für Beispielmengen an.

Beispiel 4.8.
Seien $L := \{1, 2\}, M := \{2, 3\}, N := \{3, 4\}$.

- $L \cap M = \{2\}$, denn die 2 ist das einzige Element, was L und M gemein haben.
- $L \cup M = \{1, 2, 3\}$, denn jedes dieser Elemente kommt in mindestens einer der beiden Mengen vor.
- L und N sind disjunkt, denn sie haben kein Element gemein. Daraus folgt auch $L \backslash N = L$ und $N \backslash L = N$.
- $L \backslash M = \{1\}$, denn die 1 ist das einzige Element aus L, das in M nicht auftritt.
- $L \Delta M = \{1, 3\}$, denn die 1 und die 3 sind die einzigen Elemente, die in genau einer der beiden Mengen L und M auftreten (und nicht in beiden).
- $\mathcal{P}(L) = \big\{\emptyset, \{1\}, \{2\}, \{1, 2\}\big\}$, denn dies sind alle Teilmengen von L.

Das Konzept der Potenzmenge werden wir in einem späteren Abschnitt nochmal aufgreifen. Du kannst hier nochmal überprüfen, ob du es verstanden hast.

Aufgabe 4.4.
Du hast Salz, Pfeffer, Paprika und Basilikum im Haushalt. Auf wieviele verschiedene Arten kannst du deinen Salat würzen? Gib alle möglichen Kombinationen an.

Venn-Diagramm

Die Grafiken in Abbildung 4.1 veranschaulichen die in Definition 4.7 beschriebenen Mengen. Man nennt diese Art der Darstellung von Mengen *Venn-Diagramm*. Jeder Kreis symbolisiert die Menge, die mit dem Buchstaben in der Kreismitte gekennzeichnet ist. Die farbig hervorgehobene Fläche steht für die neu definierte Operation bzw. die Menge, die sich daraus ergibt.

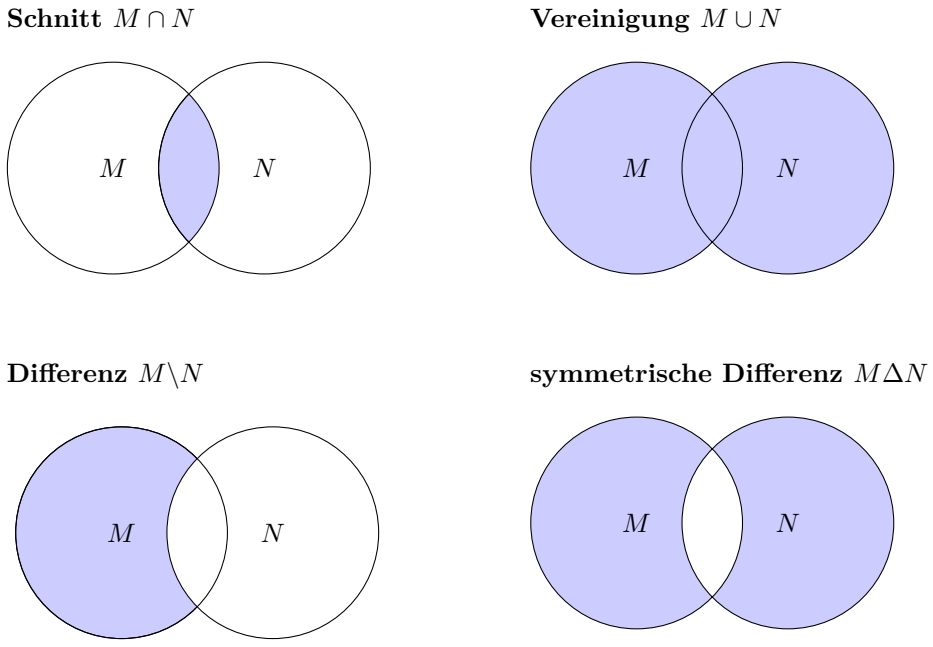


Abbildung 4.1.: Operationen auf zwei Mengen M und N

Manchmal möchte man auch mehr als zwei Mengen vereinigen oder schneiden. Man kann sich beispielsweise den Schnitt von drei Mengen L, M, N als Schnitt des Schnitts von L und M mit N vorstellen. Genauso gut könnte man damit aber auch den Schnitt des Schnitts von M und N mit L meinen! Dass die resultierenden Mengen identisch sind, besagt die erste Behauptung in der nächsten Aufgabe. Deshalb können wir der Einfachheit halber bei Mehrfachvereinigungen und -schnitten die Klammern weglassen und definieren:

Notation 4.9.
Für drei Mengen L, M, N bezeichnen wir mit $L \cap M \cap N$ die Menge $(L \cap M) \cap N = L \cap (M \cap N)$ und mit $L \cup M \cup N$ die Menge $(L \cup M) \cup N = L \cup (M \cup N)$. Entsprechend gehen wir bei k Mengen $M_1, M_2, \ldots, M_k$ vor.

Wir geben noch weitere Resultate über Mengen an, darunter auch das bereits erwähnte, was uns die o.a. Notation erlaubt. Um die Gleichheit zweier Mengen M und N nachzuweisen, beweist man oft $M \subseteq N$ und $M \supseteq N$. Mit Definition 4.3 und Satz 4.5 folgt daraus nämlich direkt $M = N$. Nutze diesen Hinweis in den Beweisen der unten angegebenen Behauptungen.

Aufgabe 4.5.
Überzeuge dich anhand der Venn-Diagramme aus Abb. 4.1 von der Richtigkeit der folgenden Aussagen für alle Mengen L, M, N. Beweise sie dann mit Hilfe der Techniken, die du bisher erlernt hast.

- $M \cap M = M$
- $M \cup M = M$
- $M \cap N = N \cap M$ *(Kommutativgesetz des Schnitts)*
- $M \cup N = N \cup M$ *(Kommutativgesetz der Vereinigung)*
- $L \cap (M \cap N) = (L \cap M) \cap N$ *(Assoziativgesetz des Schnitts)*
- $L \cup (M \cup N) = (L \cup M) \cup N$ *(Assoziativgesetz der Vereinigung)*
- $L \cap (M \cup N) = (L \cap M) \cup (L \cap N)$ *(1. Distributivgesetz)*
- $L \cup (M \cap N) = (L \cup M) \cap (L \cup N)$ *(2. Distributivgesetz)*
- $M = N$ *gilt genau dann, wenn* $M \Delta N = \emptyset$ *ist.*
- $M \backslash N$ *und* $N \backslash M$ *sind disjunkt.*
- $M \cup N = (M \backslash N) \cup (N \backslash M) \cup (M \cap N)$ *und je zwei der drei Mengen, die hier vereinigt werden, sind disjunkt.*

Ein weiteres wichtiges Konzept in der Mengenlehre ist die *Mächtigkeit* einer Menge. Sie besagt, „wie groß“ diese ist.

Definition 4.10.
Sei M eine Menge. Wir bezeichnen M als *endlich* , wenn es ein $k \in \mathbb{N}$ gibt, sodass M genau k Elemente hat. endliche Menge

Die Zahl k ist die *Mächtigkeit* oder *Kardinalität* (in Zeichen $|M|$) von M. Hat M unendlich viele Elemente, so ist seine Mächtigkeit ebenfalls unendlich. Wir schreiben dann $|M| = \infty$ (lies: $\infty =$ „unendlich“). Mächtigkeit

Endliche Mengen werden meistens explizit angegeben, Mengen, deren Mächtigkeit unendlich ist, hingegen mit Pünktchen (vgl. $\mathbb{Z}$ in Beispiel 4.2) oder implizit durch eine Umschreibung.

Beispiel 4.11.

- $|\emptyset| = 0$, da die leere Menge keine Elemente hat.
- $|\{\emptyset\}| = 1$, da diese Menge ein Element, nämlich die leere Menge, hat.
- $|\{\text{er, sie, es}\}| = 3$
- $|\{\text{Haus}, 6000, \text{weiß}\}| = 3$
- $|\{1, 1, 1, 1, 2\}| = 2$, da $\{1, 1, 1, 1, 2\} = \{1, 2\}$.
- $|\{\{a, b\}, c\}| = 2$, da $\{a, b\}$ hier ein *Element* der „äußeren“ Menge ist.
- $|\mathbb{R}| = \infty$, da es unendlich viele reelle Zahlen gibt.

Um dich an den neuen Begriff zu gewöhnen und als Einstimmung auf die folgenden Sätze und Beweise versuche dich doch mal an diesen Übungen:

Aufgabe 4.6.
Bestimme für jede Menge aus Beispiel 4.2 ihre Mächtigkeit.

Wie müssen zwei Mengen aussehen, deren Vereinigung die Mächtigkeit 0 *hat?*

Was kannst du über die Mächtigkeiten zweier Mengen sagen, deren Schnitt die Mächtigkeit ∞ *hat?*

Alle Behauptungen, die wir hier als *Satz* vorstellen, sind wahr und wurden bereits von jemandem bewiesen. Es gibt aber auch Aussagen, für die bisher noch kein Beweis gefunden wurde, es ist offen, ob sie gelten. Außerdem existieren natürlich Behauptungen, die falsch sind. Wie beweist man, dass eine Behauptung nicht korrekt ist? Man kann zum Beispiel versuchen, das Gegenteil mit einem Beweis zu zeigen. Wenn die Behauptung die Form „Für alle ... gilt ...“ hat, ist es noch einfacher: Man muss zeigen, dass die Aussage *nicht* für alle ... gilt. Dafür reicht es, ein Gegenbeispiel zu finden, für das die Aussage nicht wahr ist.

Beispiel 4.12.
Seien M und N zwei Mengen mit $|M| = |N| = \infty$.

Behauptung: $|M \cap N| = \infty$, d.h. der Schnitt zweier Mengen mit Mächtigkeit ∞ hat immer auch Mächtigkeit ∞.

Um diese Behauptung zu widerlegen, müssen wir zwei Mengen M und N finden, die jeweils unendlich viele Elemente enthalten, deren Schnitt aber endlich ist. Setze zum Beispiel

$$M := \{x \mid \text{Es gibt ein } k \in \mathbb{Z} \text{ mit } x = 2 \cdot k\},$$
$$N := \{x \mid \text{Es gibt ein } k \in \mathbb{Z} \text{ mit } x = 2 \cdot k + 1\}.$$

M ist die Menge aller geraden Zahlen, N die Menge aller ungeraden Zahlen. Jede der beiden enthält offensichtlich unendlich viele Elemente, erfüllt also die Voraussetzungen der Behauptung. Trotzdem ist

$$M \cap N = \emptyset$$

und daher

$$|M \cap N| = |\emptyset| = 0 < \infty.$$

Wir haben also zwei Mengen gefunden, über die die Behauptung die Aussage macht, ihr Schnitt habe Mächtigkeit ∞, obwohl die Mächtigkeit 0 ist. Folglich stimmt die Behauptung nicht.

Vorsicht! Im Beispiel oben zeigen wir nicht, dass der Schnitt von zwei Mengen mit Mächtigkeit ∞ *immer* endliche Kardinalität hat. Für manche Mengen, z.B. im Fall $M := \mathbb{N}, N := \mathbb{N}$, ergibt sich ein Schnitt mit Mächtigkeit ∞. Trotzdem haben wir bewiesen, dass nicht die Behauptung, sondern ihr Gegenteil stimmt: *Es gibt* zwei Mengen M und N mit Mächtigkeit ∞, deren Schnitt endlich ist.

Aufgabe 4.7.
Seien M und N zwei beliebige Mengen. Widerlege die folgenden Behauptungen:

- $|\mathcal{P}(M)| \geq 2$
- $|M \cup N| = |M| + |N|$
- $|M \cap N| \geq |M \cup N|$
- $|M \cap N| < |M \cup N|$
- *Aus $M \subseteq N$ folgt $|M| < |N|$.*
- *Aus $M \subsetneq N$ folgt $|M| < |N|$. (Hinweis: Untersuche Mengen mit Mächtigkeit ∞.)*

Die zweite falsche Behauptung aus Aufgabe 4.7 können wir zu einer korrekten Aussage „reparieren“. Es handelt sich wieder um eine Genau-dann-wenn-Aussage, der Beweis besteht daher wie auch der von Satz 4.5 aus zwei Teilen.

Satz 4.13.
Seien M und N endliche Mengen. Dann gilt $|M \cup N| = |M| + |N|$ genau dann, wenn M und N disjunkt sind.

Beweis. Seien M und N Mengen mit endlichen Mächtigkeiten $k_M < \infty$ und $k_N < \infty$.

„$\Leftarrow$“: Wir zeigen zunächst: Wenn M und N disjunkt sind, dann folgt $|M \cup N| = |M| + |N|$.

Angenommen, M und N sind disjunkt.

Jedes x, das in die linke oder rechte Seite der zu beweisenden Gleichung eingeht, liegt in einer der drei Mengen $M \cap N, M\backslash N, N\backslash M$. Nach Definition 4.7 gilt $M \cap N = \emptyset$, denn M und N sind disjunkt. Es gibt daher kein $x \in M \cap N$. Demnach liegt jedes x, das in eine der beiden Seiten der zu zeigenden Gleichungen eingeht, in $M\backslash N$ oder in $N\backslash M$.

Jedes x, das *entweder* Element von M *oder* von N ist, d.h. jedes $x \in (M\backslash N) \cup (N\backslash M)$, trägt zu $|M \cup N|$ genau 1 bei, da es nach Definition 4.3 auch Element von $M \cup N$ ist. Es erhöht aber auch den Wert von $|M| + |N|$ um 1, denn es tritt in genau einer der beiden Mengen M und N auf.

Was haben wir durch diese *Fallunterscheidung* gezeigt? Jedes Element jeder der drei Mengen, die in der zu beweisenden Gleichung auftreten, trägt zur linken Seite gleich viel wie zur rechten Seite bei. D.h. für die Mächtigkeit von $M \cup N$ ergibt sich derselbe Wert wie für die Summe der Mächtigkeiten von M und N, was wir zeigen wollten. Fallunterscheidung

„$\Rightarrow$“: Nun zeigen wir: Wenn $|M \cup N| = |M| + |N|$ gilt, dann müssen M und N disjunkt sein.

Mit dem letzten Resultat aus Aufgabe 4.5 können wir $M \cup N$ als $(M\backslash N) \cup (N\backslash M) \cup (M \cap N)$ schreiben, wobei alle drei vereinigten Mengen paarweise, d.h. je zwei von ihnen, disjunkt sind. Daher gilt auch

$$|M \cup N| = |(M\backslash N) \cup (N\backslash M) \cup (M \cap N)|.$$

Nur die Elemente, die in mindestens einer der beiden Mengen M und N sind, tragen links oder rechts etwas bei, da alle Mengen, die in der Gleichung auftauchen, aus Operationen ausschließlich auf M und N entstehen. D.h. wir müssen nur betrachten, was die Elemente $x \in M \cup N$ beitragen. Links tragen sie jeweils genau 1 bei, weil sie alle Element der Menge $M \cup N$ sind und in einer Menge jedes Element nur einmal zählt. Für den Beitrag rechts benutzen wir wieder eine Fallunterscheidung:

- Gilt $x \in M \backslash N$, so trägt x in der Summe $|M| + |N|$ auch nur 1 bei, weil es in M, aber nicht in N ist und somit nur zum ersten Summanden beiträgt.
- Gilt $x \in N \backslash M$, so trägt x in der Summe $|M| + |N|$ auch nur 1 bei, weil es in N, aber nicht in M ist und somit nur zum zweiten Summanden beiträgt.
- Gilt $x \in M \cap N$, so trägt x in der Summe $|M| + |N|$ jedoch 2 bei, weil es in M und in N ist und somit zu beiden Summanden je 1 beiträgt.

Das heißt, alle Elemente aus $M \cup N$ tragen links und rechts in der Gleichung $|M \cup N| = |M| + |N|$ gleich viel, nämlich 1 bei – außer die $x \in M \cap N$, sie tragen zur Kardinalität links 1, aber rechts 2 bei. Damit die Gleichung trotzdem gilt (und das setzen wir ja in diesem Teil des Beweises voraus), darf es also keine $x \in M \cap N$ geben: M und N müssen disjunkt sein und das war zu zeigen.

Damit haben wir beide Implikationen bewiesen und sind fertig. □

Im obigen Beweis haben wir eine weitere wichtige Technik benutzt: Fallunterscheidungen. Wenn wir nicht überblicken, wie wir mit einem direkten Beweis die Aussage folgern können, bieten sie sich oft an. Wir vereinfachen die aktuelle Situation im Beweis, indem wir mehrere Möglichkeiten separat voneinander behandeln. In jedem davon führen wir den Beweis weiter oder im Idealfall zu Ende, dann müssen wir nur noch erklären, weshalb keine anderen Fälle als die behandelten auftreten können.

Nun kannst du dich selbst an ähnlichen Beweisen versuchen.

Aufgabe 4.8.
Seien M und N beliebige Mengen. Zeige:

- $|M \cup N| \leq |M| + |N|$
- $|M \cap N| \leq \min\{|M|, |N|\}$ *($\min\{a, b\}$ ist die kleinere der beiden Zahlen a und b.)*

  Tipp: Zeige $|M \cap N| \leq |M|$ und $|M \cap N| \leq |N|$ und folgere daraus die Behauptung.

Binomialkoeffizient

Der Binomialkoeffizient $\binom{n}{k}$ *ist die Anzahl der Teilmengen mit Mächtigkeit k einer Menge mit Mächtigkeit n. Beweise die folgende Formel zur Berechnung von $\binom{n}{k}$:*

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Hinweis: Für eine Menge M mit n Elementen markiere eines davon. Unterscheide zwischen den Teilmengen, die das markierte Element enthalten, und denen, die es nicht enthalten. Wie kannst du hier Satz 4.13 anwenden?

Inzwischen kennst du dich mit Mengen schon recht gut aus. In den kommenden beiden Kapiteln benutzen wir die hier vorgestellten Begriffe, um uns mit Relationen und Funktionen zu beschäftigen. Diesen wirst du im Zusammenhang mit Mengen, insbesondere mit $\mathbb{N}$ und $\mathbb{R}$, im Laufe deines Studiums häufig begegnen.

5. Relationen

Wenn uns eine Menge vorliegt, spielt die Reihenfolge ihrer Elemente keine Rolle. Das heißt, zwei Mengen sind gleich, wenn sie dieselben Elemente enthalten, auch wenn diese nicht in derselben Reihenfolge notiert sind. Das haben wir in Definition 4.3 so festgelegt. Manchmal wollen wir aber verschiedene Reihenfolgen von Objekten unterscheiden. Zum Beispiel müssen bei einem Seminar im Laufe des Semesters alle Teilnehmer einen Vortrag halten. Zur Planung der Vorbereitung ist es entscheidend, zu wissen, ob man den ersten oder den letzten Vortrag halten muss! Auch in einer Warteschlange an der Kinokasse bekommen im Normalfall alle ihre Tickets, aber der/die Erste wird kaum seinen Platz mit jemand anderem tauschen wollen, denn er/sie hat noch freie Platzwahl.

Definition 5.1.
Seien $m, n \in \mathbb{N}$.

(*a*) Seien $a_1, a_2, \ldots, a_n$ beliebige Objekte. Wir nennen $(a_1, a_2, \ldots, a_n)$ das *geordnete Tupel* mit den *Komponenten* $a_1, a_2, \ldots, a_n$. Dabei ist a_i die *i-te Komponente* und n die *Länge* des Tupels. — Tupel

(*b*) Ein Tupel der Länge 2 wird auch als *geordnetes Paar* bezeichnet, ein Tupel der Länge 3 als *Tripel*, ein Tupel der Länge 4 als *Quadrupel*, usw. Tupel der Länge n nennen wir kurz n-Tupel. — geordnetes Paar

(*c*) Zwei Tupel $(a_1, a_2, \ldots, a_m)$ und $(b_1, b_2, \ldots, b_n)$ sind *gleich*, wenn $m = n$ und für alle $1 \leq i \leq n$ ihre i-ten Komponenten gleich sind, d.h. $a_i = b_i$ gilt. — Gleichheit

Nach dieser Definition gelten zwei Tupel gleicher Länge nur dann als identisch, wenn sie an jeder Position, d.h. in jeder Komponente, übereinstimmen. Deshalb ist es an dieser Stelle auch nicht sinnvoll, zu definieren, wann wir ein Tupel als „Teiltupel“ eines anderen verstehen wollen, was bei Mengen sehr wohl nützlich war. In einem Tupel können Objekte auch mehrfach auftreten, sie gelten aufgrund ihrer unterschiedlichen Positionen als verschieden.

Beachte: Tupel werden mit runden Klammern gekennzeichnet, Mengen hingegen schreibt man mit geschweiften Klammern!

Beispiel 5.2.

- Das leere Tupel () hat Länge 0.
- Es ist $(1, 2) \neq (1, 2, 1) \neq (1, 2, 2)$, aber $\{1, 2, 1\} = \{1, 2, 2\} = \{1, 2\}$.
- (Anna, Peter) ist ein geordnetes Paar, {Anna, Peter} ist eine zweielementige Menge, manchmal auch *Paar* genannt.
- Es gilt $(a_1, a_2, \ldots, a_n) \neq \{a_1, a_2, \ldots, a_n\}$ für beliebige Objekte $a_1, a_2, \ldots, a_n$.

Auch in diesem Skript ist die Reihenfolge entscheidend: Würden wir erst den Satz und nachträglich die Definition vorstellen, so könnte den Satz niemand verstehen und erst recht nicht beweisen!

Aufgabe 5.1.
Überlege dir weitere Beispiele für Alltagssituationen und Objekte, die man besser als Komponenten eines Tupes als in Form von Mengen erfassen sollte, weil ihre Reihenfolge wichtig ist. Gib auch Beispiele an, in denen nur die Zugehörigkeit zu einer Menge von Interesse ist.

Wir stellen jetzt eine neue Mengenoperation vor. Das Ergebnis ist eine Menge von Tupeln.

Definition 5.3.
kartesisches Produkt
Sei $n \in \mathbb{N}$ und seien $M_1, M_2, \ldots, M_n$ Mengen. Als *kartesisches Produkt* von $M_1, M_2, \ldots, M_n$ bezeichnen wir

$$M_1 \times M_2 \times \ldots \times M_n := \big\{(a_1, a_2, \ldots, a_n) \mid a_i \in M_i \text{ für alle } i \in \{1, 2, \ldots, n\}\big\},$$

also die Menge aller Tupel, bei denen die erste Komponente ein Element von M_1, die zweite eines von M_2 ist usw.

Wenn alle Mengen, von denen wir das kartesische Produkt bilden wollen, identisch sind, können wir dieses kürzer ausdrücken:

Notation 5.4.
Falls $M := M_1 = M_2 = \ldots = M_n$, so schreiben wir M^n anstelle von $M_1 \times M_2 \times \ldots \times M_n$.

Kartesischen Produkten begegnen wir im Alltag immer wieder. Wir zählen hier einige Beispiele auf, vielleicht fallen dir ja auch noch welche ein?

Beispiel 5.5.

- Das kartesische Produkt der Menge W aller weiblichen und der Menge M aller männlichen Besucher eines Tanzkurses ist $W \times M$, die Menge aller Tanzpaare der Form (Frau, Mann).
- Die Menge aller Uhrzeiten im Format SS:MM ist $\{00, 01, \ldots, 23\} \times \{00, 01, \ldots, 59\}$.
- Die Menge aller dreigängigen Menüs, die eine Speisekarte bietet, ist $V \times H \times N$, wobei V die Menge aller Vorspeisen, H die Menge aller Hauptspeisen, N die Menge aller Nachspeisen ist.
- Die reelle Zahlenebene ist $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. Menge aller geographischen Koordinaten ist $L \times B = \{0, 1, \ldots, 59\} \times \{0, 1, \ldots, 59\}$, wobei L die Menge aller Längen- und B die Menge aller Breitengrade ist.

Aufgabe 5.2.
Gib die folgenden kartesischen Produkte in expliziter Schreibweise sowie ihre Mächtigkeit an.

- $\{0, 1\} \times \{0, 1\}$
- M^0 *und* M^1 *für eine beliebige Menge* M
- $M \times \emptyset$ *für eine beliebige Menge* M
- $\{$*rot, gelb, grün*$\} \times \{$*Ball, Kleid, Auto*$\}$

Wie lässt sich die Menge aller Spielkarten eines Skatblatts als kartesisches Produkt zweier Mengen auffassen? Welche Mächtigkeiten haben diese?

Nach dem Lösen der obigen Aufgabe hast du wahrscheinlich schon eine Vermutung, wie sich die Mächtigkeit des kartesischen Produkts $M_1 \times M_2 \times \ldots \times M_n$ aus denen der M_i ergibt. Vielleicht kennst du auch das folgende Resultat noch aus dem Stochastikunterricht in der Schule.

Satz 5.6.
Sei $n \in \mathbb{N}$, seien $M_1, M_2, \ldots, M_n$ Mengen. Dann gilt

$$|M_1 \times M_2 \times \ldots \times M_n| = |M_1| \cdot |M_2| \cdot \ldots \cdot |M_n|.$$

Beweis. Am besten kann man den Satz mittels *vollständiger Induktion* zeigen. Da wir diese noch nicht thematisiert haben und es eine anfangs kompliziert erscheinende Beweistechnik ist, deuten wir sie hier nur an.

Um ein Element des kartesischen Produkts eindeutig zu identifizieren, müssen alle seine Komponenten festgelegt sein. Für die erste Komponente können wir jedes Element aus M_1 nehmen, also haben wir hier $|M_1|$ Optionen. Das so definierte 1-Tupel können wir für die Konstruktion eines 2-Tupels benutzen, wobei wir als zweite Komponente nur Elemente aus $|M_2|$ erlauben. Jedes 1-Tupel aus dem ersten Schritt liefert für ein festes $m \in M_2$ ein anderes 2-Tupel, d.h. jedes $m \in M_2$ erzeugt $|M_1|$ neue 2-Tupel. Also ist die Anzahl der 2-Tupel, deren erste Komponente aus M_1 und deren zweite Komponente aus M_2 stammt, $|M_1| \cdot |M_2|$.

Analog ergibt sich als Anzahl der 3-Tupel, bei denen für alle i die i-te Komponente aus M_i stammt, $|M_1| \cdot |M_2| \cdot |M_3|$ und als Anzahl der n-Tupel mit dieser Eigenschaft

$$|M_1| \cdot |M_2| \cdot \ldots \cdot |M_n|$$

und das war zu zeigen. □

Die im Beweis erwähnte *vollständige Induktion* ist eine beliebte Beweistechnik, wenn man eine Behauptung für alle natürlichen Zahlen n zeigen möchte. Sobald du sie in Kapitel 8 kennen gelernt hast, kannst du den obigen Beweis eleganter formulieren!

vollständige Induktion

Den zugehörigen Satz kannst du dir in der nächsten Aufgabe gleich zunutze machen.

Aufgabe 5.3.
Überprüfe deine Antworten zu Aufgabe 5.2, indem du mit Hilfe von Satz 5.6 *die Kardinalitäten der beschriebenen Mengen ausrechnest.*

Auch wenn es, wie vorhin bemerkt, kompliziert wäre, das Teilmengensymbol $\subseteq$ sinnvoll für Tupel zu definieren, können wir es im Kontext des kartesischen Produkts wieder ins Spiel bringen – denn das kartesische Produkt ist ja eine *Menge* von Tupeln.

Definition 5.7.
Sei $n \in \mathbb{N}$ und seien $M_1, M_2, \ldots, M_n$ Mengen. Eine Teilmenge R des kartesischen Produkts $M_1 \times M_2 \times \ldots \times M_n$ heißt *Relation* von $M_1, M_2, \ldots, M_n$ mit *Stelligkeit* n.

Relation
Stelligkeit

Nach der obigen Definition ist jedes kartesische Produkt $M_1 \times M_2 \times \ldots \times M_n$ selbst eine (n-stellige) Relation von $M_1, M_2, \ldots, M_n$.

Zwei Relationen sind, da sie beide Mengen sind, genau dann gleich, wenn sie dieselben Tupel enthalten. (Beachte hierbei Definition 5.1.)

Beispiel 5.8.

- Sei M das deutsche Alphabet, als Menge aufgefasst. Wenn wir jedes Wort als Tupel seiner Buchstaben interpretieren, ist die Menge aller dreibuchstabigen deutschen Wörter eine Relation von M, M, M.
- Für die in Beispiel 5.5 definierte Menge $W \times M$ ist die Menge aller tatsächlich gebildeten Tanzpaare für den Abschlussball eine Relation von W und M, denn jede(r) TeilnehmerIn des Kurses wird nur mit einem/einer einzigen TanzpartnerIn zum Ball gehen.
- Die Menge aller bereits bestellten dreigängigen Menüs ist eine Relation der Mengen V, H und N, die in Beispiel 5.5 definiert wurden.
- Die Menge $\{(x, y) \mid x = |y|\}$, bei deren Elementen die erste Komponente der Betrag der zweiten ist, stellt eine Relation von $\mathbb{R}$ und $\mathbb{R}$ dar.

Du findest bestimmt noch zahlreiche weitere Beispiele für Relationen, insbesondere in der Mathematik tauchen sie oft auf.

Aufgabe 5.4.
Gib weitere zweistellige Relationen von $\mathbb{R}$ und $\mathbb{R}$ an. Findest du auch welche von $\mathbb{N}$ und $\mathbb{R}$?

Wie kannst du die dir aus der Schule bekannten Funktionsgraphen als Relationen auffassen? Welche Stelligkeit haben sie?

Die Antwort auf die letzten beiden Fragen in Aufgabe 5.4 liefern wir im nächsten Abschnitt, dort thematisieren wir Funktionen als spezielle Relationen.

6. Funktionen

Mit Funktionen hatten wir alle in der Schule schon zu tun. Eine Funktion ist, ausgedrückt mit den Begriffen, die wir neu kennen gelernt haben, eine Relation von zwei Mengen, bei der jedes Element der ersten Menge in genau einem Tupel der Relation als erste Komponente auftritt. D.h. zu jedem Element der ersten Menge gibt es *genau ein* „zugehöriges“ der zweiten Menge.

Definition 6.1.
Seien X und Y Mengen.

(*a*) Eine Relation $f \subseteq X \times Y$, bei der es zu jedem $x \in X$ genau ein $y \in Y$ mit $(x, y) \in f$ gibt, nennen wir *Funktion von X nach Y* (in Zeichen: $f : X \to Y$). Die Menge X heißt *Definitionsbereich* von f und die Menge Y *Bildbereich* von f. Funktion

(*b*) Sei f eine Funktion von X nach Y. Für jedes $x \in X$ bezeichnen wir mit $f(x)$ das eindeutige $y \in Y$, für das $(x, y) \in f$ gilt. Ist Y eine Menge von Zahlen, so sprechen wir statt von $f(x)$ auch vom *Funktionswert* von x.

(*c*) Sei f eine Funktion von X nach Y. Wir nennen die Menge Bild

$$f(X) := \{y \in Y \mid \text{ Es gibt ein } x \in X \text{ mit } f(x) = y\}$$

das *Bild* von f.

Bei einer Funktion von X nach Y gibt es also zu jedem $x \in X$ genau ein zugehöriges $y \in Y$. Dabei sind der Definitionsbereich X und der Bildbereich Y nicht zwangsläufig Mengen von Zahlen, ebensowenig wie bei einer Relation.

Aufgabe 6.1.
Bei welchen der folgenden Relationen handelt es sich um Funktionen? Kannst du deine Behauptung beweisen? Versuche, für jede Funktion auch ihr Bild anzugeben.

- *Die geordneten Paare der Form (Personalausweisnummer, Einwohner Deutschlands über* 16*) als Teilmenge des kartesischen Produkts von* $\mathbb{N}$ *mit den Einwohnern Deutschlands*
- *Die geordneten Paare der Form (Adresse, Einwohner Deutschlands) als Teilmenge des kartesischen Produkts der Menge aller Adressen in Deutschland und der Einwohner Deutschlands*
- *Die Menge aller tatsächlich gebildeten Tanzpaare der Form (Frau, Mann) als Teilmenge des kartesischen Produkts der weiblichen und der männlichen Tanzkursteilnehmer (vorausgesetzt, jede Frau findet einen Tanzpartner)*
- $M \times N \subseteq M \times N$ *für beliebige Mengen* M *und* N *mit* $|N| \leq 1$
- $\mathbb{N} \times \mathbb{N} \subseteq \mathbb{R} \times \mathbb{R}$
- $\{(x, x) \in \mathbb{R}^2\}$
- $\{(x, y) \in \mathbb{R}^2 \mid y = 5x\}$
- $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$
- $\{(x, y) \in \mathbb{Z}^2 \mid x$ *ist Teiler von* $y\}$

- $\{(x, y) \in \mathbb{Z}^2 \mid y \text{ ist Teiler von } x\}$
- $\{(x, y) \in \mathbb{Z}^2 \mid x \text{ ist eine Primzahl und } y \text{ ist Teiler von } x\}$

Wann sind eigentlich zwei Funktionen „gleich“? Wenn wir sie als zweistellige Relationen auffassen, können wir mit den bereits vorgestellten Definitionen die Aussage der nächsten Aufgabe zeigen.

Aufgabe 6.2.
Seien f und g zwei Funktionen mit demselben Definitionsbereich X und demselben Bildbereich Y. Zeige: f und g sind genau dann gleich, wenn für alle $x \in X$ gilt: $f(x) = g(x)$. Fasse hierzu f und g als zweistellige Relationen auf.

Oft werden Funktionen aber auch als eindeutige Zuordnungen eingeführt. Dann wird die Menge, die wir hier mit f identifiziert haben, meist als *Graph von f* bezeichnet.

Notation 6.2.
Funktionen können in Mengenschreibweise mit geschweiften Klammern angegeben werden. Alternativ kann man eine Funktionsvorschrift der Form $f : x \mapsto y$ oder eine Funktionsgleichung der Form $f(x) := y$ (die Definitionsdoppelpunkte werden häufig weggelassen) verwenden, um $(x, y) \in f$ auszudrücken.

Im folgenden Beispiel geben wir zum Verständnis alle Notationsarten an, um die Funktion von $\mathbb{N}$ nach $\mathbb{N}$ zu beschreiben, die jeder natürlichen Zahl ihr Doppeltes zuweist.

Beispiel 6.3.
Die Varianten

- $f_1 = \{(0, 0), (1, 2), (2, 4), (4, 8), \dots\} \subseteq \mathbb{N} \times \mathbb{N}$
- $f_2 = \{(x, y) \in \mathbb{N}^2 \mid y = 2x\}$
- $f_3 : \mathbb{N} \to \mathbb{N}, f(n) := 2n$
- $f_4 : \mathbb{N} \to \mathbb{N}, f : n \mapsto 2n$

beschreiben allesamt dieselbe Funktion. Hierbei sind die Darstellungen von f_2, f_3 und f_4 der von f_1 vorzuziehen, weil sie unmissverständlicher sind. Dies gilt immer, wenn die Mächtigkeit der Funktion unendlich ist (vgl. hierzu den Kommentar nach Definition 4.10).

Wenn zwei Funktionen f und g nicht denselben Definitionsbereich X haben, können sie auch nicht gleich sein:

Satz 6.4.
Seien $f : X_f \to Y_f$ und $g : X_g \to Y_g$ Funktionen.

Falls $X_f \neq X_g$, so sind f und g nicht gleich im Sinne von Definition 4.3.

Beweis. Seien f und g wie in den Voraussetzungen des Satzes beschrieben und gelte $X_f \neq X_g$. Dann gibt es nach der 9. Teilübung aus Aufgabe 4.5 ein Element in $X_f \backslash X_g$ oder eines in $X_g \backslash X_f$, denn $X_f \Delta X_g \neq \emptyset$.

Wenn es ein Element $\tilde{x} \in X_f \backslash X_g$ gibt, so ist

$$\big(\tilde{x}, f(\tilde{x})\big) \in f,$$

da es nach Definiton 6.1 zu *jedem* $x \in X_f$ ein $y \in Y_f$ mit $(x, y) \in f$ gibt. Aber

$$\big(\tilde{x}, f(\tilde{x})\big) \notin g,$$

denn $x \notin X_g$ und $g \subseteq X_g \times Y_g$.

Analog ergibt sich für $\tilde{x} \in X_g \backslash X_f$, dass $\big(\tilde{x}, g(\tilde{x})\big) \in g$, doch $\big(\tilde{x}, g(\tilde{x})\big) \notin f$ ist.

Also ist $f \Delta g \neq \emptyset$ und mit Aufgabe 4.5 (9. Teilübung) können dann f und g nicht gleich sein. □

Wenn wir zwei Funktionen als gleich ansehen, sollen sich diese auch genau gleich verhalten bzw. die gleichen Eigenschaften haben. Mit Hinblick auf die Begriffe „injektiv“, „surjektiv“ und „bijektiv“, die wir für Funktionen einführen werden, ist es sinnvoll, zusätzlich zum selben Definitionsbereich die Gleichheit der Bildbereiche zu fordern. Das führt uns zur folgenden Definition:

Definition 6.5.
Seien X und Y Mengen, seien f und g Funktionen mit dem gleichen Definitionsbereich X und dem gleichen Bildbereich Y.

(*a*) Wir bezeichnen f und g als *gleich* (in Zeichen: $f \equiv g$), wenn für alle $x \in X$ gilt: $f(x) = g(x)$. Gleichheit

(*b*) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *kleiner als oder gleich* g (in Zeichen $f \leq g$), wenn für alle $x \in X$ gilt: $f(x) \leq g(x)$.

(*c*) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *kleiner als* g (in Zeichen $f < g$), wenn für alle $x \in X$ gilt: $f(x) < g(x)$.

(*d*) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *größer als oder gleich* g (in Zeichen $f \geq g$), wenn für alle $x \in X$ gilt: $f(x) \geq g(x)$.

(*e*) Seien $X, Y \subseteq \mathbb{R}$. Die Funktion f ist *größer als* g (in Zeichen $f > g$), wenn für alle $x \in X$ gilt: $f(x) > g(x)$.

Eine Funktion f ist also kleiner als eine Funktion g, wenn der Graph von f „unterhalb“ des Graphen von g verläuft – und sowohl ihre Definitions- als auch ihre Bildbereich übereinstimmen! Denn nur dann lassen sich nach unserer Definition die beiden Funktionen überhaupt vergleichen. Aus $f < g$ folgt, unserer Intuition entsprechend, direkt $f \leq g$, genauso verhält es sich mit $f > g$ und $f \geq g$.

Haben die Graphen der Funktionen Schnittpunkte, die nicht bloß Berührpunkte sind, so gilt weder $f < g$ noch $f > g$ (noch $f \leq g$ noch $f \geq g$ noch $f \equiv g$).

Aufgabe 6.3.
Gib für je zwei der folgenden Funktionen an, ob sie vergleichbar sind, und falls ja, ob $\equiv, \leq, <, \geq, >$ *gilt. Gelingt es dir, ohne die Graphen zu zeichnen?*

- $f_0 : \mathbb{N} \to \mathbb{N},\ f_0(n) := n$
- $f_1 : \mathbb{N} \to \mathbb{Z},\ f_1(n) := n$
- $f_2 : \mathbb{Z} \to \mathbb{Z},\ f_2(n) := -1$
- $f_3 : \mathbb{Z} \to \mathbb{Z},\ f_3(n) := n - 3$
- $f_4 : \mathbb{N} \to \mathbb{N},\ f_4(n) := 4n$
- $f_5 : \mathbb{Z} \to \mathbb{Z},\ f_5(n) := 4n$
- $f_6 : \mathbb{N} \to \mathbb{N},\ f_6(n) := n^3$

- $f_7 : \mathbb{Z} \to \mathbb{Z},\ f_7(n) := n^3$
- $f_8 : \mathbb{N} \to \mathbb{N},\ f_8(n) := 2^n$
- $f_9 : \mathbb{Z} \to \mathbb{R},\ f_9(n) := 2^n$

Definition 6.5(d) kann man mit Hilfe der Definition für $f \leq g$ kürzer ausdrücken. Weißt du, wie? Versuche, auch Definition 6.5(e) kürzer zu fassen.

Zeige: Für zwei Funktionen mit demselben Definitions- und demselben Bildbereich, die beide Teilmengen von $\mathbb{R}$ sind, gilt $f \equiv g$ genau dann, wenn $f \leq g$ und $g \leq f$ gilt.

Du hast bestimmt herausgefunden, dass f_0 und f_1 in der obigen Aufgabe nicht gleich sind. Dennoch nehmen sie auf ihrem gemeinsamen Definitionsbereich die gleichen Werte an, bloß dass der Bildbereich von f_1 „größer als nötig“ gewählt wurde: Er beinhaltet auch Zahlen, die gar nicht als Funktionswerte auftreten. Dies drückt man auch mit der Aussage „f_1 ist nicht surjektiv“aus. Denn Begriff der Surjektivität sowie zwei weitere wollen wir jetzt exakt definieren.

Definition 6.6.
Sei $f : X \to Y$ eine Funktion.

surjektiv (a) Wir bezeichnen f als *surjektiv*, wenn es für jedes $y \in Y$ mindestens ein $x \in X$ mit $f(x) = y$ gibt.

injektiv (b) Wir bezeichnen f als *injektiv*, wenn es für jedes $y \in Y$ höchstens ein $x \in X$ mit $f(x) = y$ gibt.

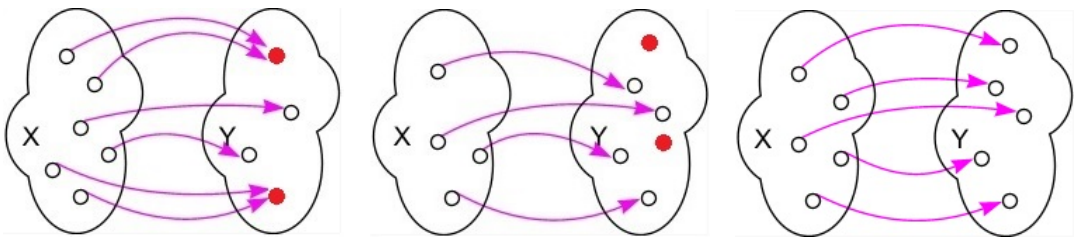
bijektiv (c) Wir bezeichnen f als *bijektiv*, wenn es für jedes $y \in Y$ genau ein $x \in X$ mit $f(x) = y$ gibt.

Es lässt sich schnell zeigen:

Aufgabe 6.4.
Sei $f : X \to Y$ eine Funktion.

f ist genau dann bijektiv, wenn f injektiv und surjektiv ist.

Abbildung 6.1 visualisiert eine surjektive, eine injektive und eine bijektive Funktion von X nach Y. Ein rosa Pfeil zwischen zwei Kringeln bedeutet, dass das Element x am Pfeilanfang durch die Funktion f auf das Element y am Pfeilende abgebildet wird, d.h. $f(x) := y$.



(a) surjektive Funktion (b) injektive Funktion (c) bijektive Funktion

Abbildung 6.1.: Veranschaulichung der Begriffe „surjektiv“, „injektiv“ und „bijektiv“.
Quelle: http://de.wikibooks.org/wiki/Mathematik:_Analysis:_Grundlagen:_Funktionen

Wir untersuchen im folgenden Beispiel noch eine Auswahl an Funktionen auf Surjektivität, Injektivität und Bijektivität, damit du lernst, wie man diese für eine Funktion nachweisen bzw. widerlegen kann.

Beispiel 6.7.

- Die Betragsfunktion $f : \mathbb{R} \to \mathbb{R},\ f(x) := |x|$ ist nicht surjektiv, da es z.B. zu -1 kein $x \in \mathbb{R}$ mit $f(x) = -1$ gibt.

  Sie ist auch nicht injektiv, da z.B. 1 und -1 denselben Funktionswert, nämlich 1 haben: $f(1) = f(-1) = 1$.

  Mit Aufgabe 6.4 kann f dann auch nicht bijektiv sein.

- Die Betragsfunktion $g : \mathbb{R} \to \mathbb{R}_{\geq 0},\ g(x) := |x|$ ist surjektiv, da für jedes nichtnegative y gilt: $f(y) = y$, d.h. zu jedem y im Bildbereich gibt es ein Element im Definitionsbereich, was auf y abgebildet wird.

  Sie ist aus dem gleichen Grund wie f aber weder injektiv noch bijektiv.

- Die Betragsfunktion $h : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0},\ g(x) := |x|$ ist aus dem gleichen Grund wie g surjektiv.

  Sie ist auch injektiv, denn zu jedem y im Bildbereich ist das einzige Element im Definitionsbereich, welches auf y abgebildet wird, y selbst.

  Mit Aufgabe 6.4 ist h auch bijektiv.

Durch Verkleinerung des Definitionsbereichs kann man jede Funktion zu einer injektiven Funktion machen und indem man den Bildbereich einer Funktion „so klein wie möglich“ macht, kann man jede nicht-surjektive Funktion in eine surjektive verwandeln.

Satz 6.8.
Sei $f : X \to Y$ eine Funktion.

f ist genau dann surjektiv, wenn $Y = f(X)$ gilt.

Beweis. Sei $f : X \to Y$ eine Funktion. Wir müssen wieder zwei Richtungen zeigen, da es sich um eine Genau-dann-wenn-Aussage (eine *Äquivalenz*) handelt

„$\Leftarrow$“: Es gelte $Y = f(X)$. Wir wollen zeigen, dass daraus die Surjektivität von f folgt. Also müssen wir nach Definition 6.6 beweisen, dass es für jedes $\tilde{y} \in Y$ mindestens ein $x \in X$ mit $f(x) = \tilde{y}$ gibt.

Sei $\tilde{y} \in Y$ beliebig. Da $Y = f(X)$, ist also $\tilde{y} \in f(X)$. Nach Definition 6.1 gehört demnach $\tilde{y}$ zur Menge $\{y \in Y \mid \text{ Es gibt ein } x \in X \text{ mit } f(x) = y\}$. Das heißt, es gibt ein $x \in X$ mit $f(x) = \tilde{y}$ und das wollten wir zeigen. Also ist f surjektiv.

„$\Rightarrow$“: Wir müssen noch beweisen, dass aus der Surjektivität von f folgt, dass $Y = f(X)$ sein muss.

Das erledigen wir mittels eines Beweises durch Widerspruch. Angenommen, f ist surjektiv, d.h. für jedes $y \in Y$ gibt es ein $x \in X$ mit $f(x) = y$. Wäre $Y \neq f(X)$, so wäre $Y \Delta f(X) \neq \emptyset$ nach Aufgabe 4.5. Also wäre $f(X) \backslash Y \neq \emptyset$ oder $Y \backslash f(X) \neq \emptyset$.

Im Falle von $f(X) \backslash Y \neq \emptyset$ müsste es ein Element in $f(X)$ geben, das nicht in Y liegt. Nach Definition 6.1(c) sind aber alle Elemente aus $f(X)$ erst recht Elemente aus Y, d.h. $f(X) \subseteq Y$, also kann es das gesuchte Element nicht geben.

Falls $Y \backslash f(X) \neq \emptyset$, so müsste es ein Element in Y geben, das nicht in $f(X)$ liegt, d.h. ein $y \in Y$, sodass es *kein* $x \in X$ mit $f(x) = y$ gibt. Das ist aber ein Widerspruch zur Surjektivität von f nach Definition 6.6, denn diese besagt, dass es für jedes $y \in Y$ so ein $x \in X$ gibt.

Jede Möglichkeit in der Fallunterscheidung hat zu einem Widerspruch geführt, also muss die Annahme $Y \neq f(X)$ falsch gewesen sein. Somit gilt $Y = f(X)$. □

Satz 6.8 wird dir bei der Bearbeitung der nächsten Aufgabe hilfreich sein, denn du kannst ihn anwenden, um aus einer beliebigen Funktion eine surjektive zu konstruieren.

Aufgabe 6.5.
Welche der Funktionen aus Aufgabe 6.3 sind injektiv bzw. surjektiv bzw. bijektiv? Gelingt es dir, die nicht-injektiven in injektive bzw. die nicht-surjektiven in surjektive zu verwandeln, indem du Definitions- und Bildbereich anpasst? Verändere diese dafür so wenig wie möglich!

Anwendungen des letzten Satzes, den wir in diesem Kapitel vorstellen wollen, werden dir oft begegnen. Er bildet die Grundlage für die Definition der *Abzählbarkeit* einer nicht-endlichen Menge. Insbesondere die „Durchnummerierungstechnik“, die im Beweis verwendet wird, findet man auch in anderen Kontexten und Argumentationen wieder.

Satz 6.9.
Seien X und Y endliche Mengen.

Es gilt $|X| = |Y|$ genau dann, wenn es eine bijektive Funktion von X nach Y gibt.

Beweis. Seien X und Y endliche Mengen. Wir beweisen erneut beide Teile des Satzes separat.

„⇒“: Angenommen, es gibt eine bijektive Funktion $f : X \to Y$. Wir wollen beweisen, dass dann die Mächtigkeiten von X und Y gleich sind, also beide gleich viele Elemente beinhalten.

Dazu nummerieren wir die Elemente von X durch, sodass $X = \{x_1, x_2, \ldots, x_n\}$ gilt. Das heißt, wir weisen einem der Elemente in X die Bezeichnung x_1 zu, einem anderen die Bezeichnung x_2 usw. Für jedes x_i bezeichnen wir $f(x_i)$ mit y_i. Weil f bijektiv ist, ist es nach Aufgabe 6.4 insbesondere surjektiv. D.h. nach Definition 6.6 existiert kein $y \in Y$, zu dem es kein $x \in X$ gibt. Also ist jedes $y \in Y$ von der Form $f(x_i)$ für irgendein i und jedes erhält daher eine Bezeichnung der Form y_i.

Aus der Bijektivität von f folgt aber ebenso mit Aufgabe 6.4 seine Injektivität. D.h. zu jedem $y \in Y$ gibt es höchstens ein $x_i \in X$ mit $y = f(x_i)$. Also bekommt jedes y nur eine Bezeichnung zugewiesen.

Aus der Bijektivität von f konnten wir so folgern, dass jedes $y \in Y$ genau eine Nummer zugewiesen bekommt. Weil f eine Funktion ist, gibt es zu jedem $xinX$ genau ein y, auf das x abgebildet wird, d.h. keine Nummer wird in Y doppelt vergeben. Also werden genauso viele verschiedene Nummern bei der beschriebenen Durchnummerierung der Elemente von Y wie für die der Elemente von X vergeben, nämlich $|X|$. Also haben X und Y gleich viele Elemente, d.h. $|X| = |Y|$.

„⇐“: Angenommen, X und Y haben die gleiche Mächtigkeit n. Gesucht ist eine bijektive Funktion von X nach Y. Dazu nummerieren wir die Elemente von X erneut zu $X = \{x_1, x_2, \ldots, x_n\}$ durch, ebenso verfahren wir mit Y und erhalten $Y = \{y_1, y_2, \ldots, y_n\}$. Wir definieren nun noch $f(x_i) := y_i$ für alle $i \in \{1, 2, \ldots, n\}$. Dieses f weist jedem Element aus X nach Konstruktion genau eines aus Y zu, also ist f eine Funktion von X nach Y. Zu jedem $y_i \in Y$ gibt es ein Element aus X, was darauf abgebildet wird, nämlich x_i. Demnach ist f surjektiv. Außerdem werden keine zwei Elemente aus X auf dasselbe $y \in Y$ abgebildet, denn für $x_i \neq x_j$ gilt auch $f(x_i) = y_i \neq y_j = f(x_j)$. Das bedeutet, f ist injektiv.

Mit Aufgabe 6.4 folgt die Bijektivität von f und damit die Existenz der gesuchten bijektiven Funktion. □

In der letzten Aufgabe dieses Kapitels kannst du die Anwendung von Satz 6.9 üben sowie prüfen, ob du verstanden hast, wie man die Bijektivität einer Funktion nachweist. Auch die Durchnummerierungstechnik wird hier benutzt.

Aufgabe 6.6.
Sei M eine endliche Menge mit Kardinalität n. Bestimme die Mächtigkeit von $\mathcal{P}(M)$, der Potenzmenge von M.

Hinweis: Definiere dazu X als kartesisches Produkt $\{0,1\}^n$. Nummeriere die Elemente von M durch. Konstruiere dann eine Funktion $f : X \to \mathcal{P}(M)$, indem du jedem $x = (x_1, x_2, \ldots, x_n) \in X$ das Element aus $\mathcal{P}(M)$ zuweist, das genau die $m_i \in M$ mit $x_i = 1$ enthält. Weise nach, dass f tatsächlich eine Funktion und außerdem bijektiv ist. Wende Satz 5.6 an, um die Mächtigkeit von X zu bestimmen und folgere mit Satz 6.9 die Mächtigkeit von $\mathcal{P}(M)$.

Im nächsten Kapitel erklären wir die bereits im Beweis zu Satz 5.6 erwähnte vollständige Induktion als wichtige Beweistechnik ausführlich. Außerdem lernst du Rekursionen kennen, rekursiv definierte Funktionen werden dir insbesondere im Zusammenhang mit Laufzeiten von Algorithmen oft begegnen.

7. Einführung in die Mathematischen Beweistechniken

Die Mathematik befasst sich mit *Definitionen, Sätze, Lemma, ...*

- *Definitionen* legen bestimmte Sachverhalte und/oder Notationen fest. Zum Beispiel:

  Definition 7.1.
  Seien a und b natürliche Zahlen. $a + b$ beschreibt die Addition der natürlichen Zahlen a und b.

- *Sätze, Lemma, ...* sind Aussagen, die aus den verschiedensten Definitionen folgen können. Um Nachzuweisen, das diese Aussagen stimmen, müssen wir einen mathematischen Beweis durchführen. Beispiel:

  Lemma 7.2 (Binomische Formel)**.**
  $(a + b)^2 = a^2 + 2ab + b^2$

  Beweis. $(a + b)^2 = (a + b) \cdot (a + b) = a^2 + a \cdot b + a \cdot b + b^2 = a^2 + 2ab + b^2$ □

In der Informatik haben wir ebenfalls mit mathematischen Strukturen zu tun, über die wir Aussagen treffen möchten. Beispielsweise über einen Algorithmus bzw. über einen Programmcode. Wichtig, ist zum Beispiel der Nachweis über die Korrektheit der Algorithmen in der Flugzeugsteuerung und der Medizintechnik, da es in diesen Bereichen in der Vergangenheit aufgrund von Programmierfehlern bzw. Fehlern in Algorithmen, mehrere Unfälle gab. Wir benötigen daher in der Informatik die mathematischen Beweistechniken um eine eindeutige Antwort auf die Gültigkeit unserer Feststellungen/Aussagen zu finden. (Macht mein Algorithmus [bzw. Programm] das was er soll? Läuft mein Algorithmus so schnell wie er sollte?). Die mathematischen Beweistechniken bilden daher eine wichtige Grundlage. In diesem Kapitel geben wir einen kurzen Einblick über das Führen/Gestalten von Beweisen.

7.1. Direkter Beweis

Mathematische Aussagen haben häufig die Form "Wenn... dann..." (Aussagenlogisch: $p \to q$) Dabei nennen wir p die Voraussetzung und q die Folgerung.
Betrachte die Aussage:

Beispiel 7.3.
Wenn eine Zahl durch 10 teilbar ist, dann ist sie auch durch 5 teilbar.

Wir teilen dann unser p und q auf:

p : Eine Zahl ist durch 10 teilbar

q : Eine Zahl ist durch 5 teilbar

Nun müssen wir, um zu zeigen, dass die Beispielaussage gilt, die Implikation $p \to q$ nachweisen. Wir betrachten dazu die Wertetabelle:

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

In der Tabelle sehen wir: Wenn p falsch ist, dann kann daraus folgen, dass q wahr oder falsch ist, es gilt dann immer $p \to q$. Wenn p wahr ist, dann muss umbedingt auch q wahr sein, damit $p \to q$ wahr ist. Wir müssen daher zeigen, wenn p wahr ist, dann ist auch q wahr.

Wir sehen also:

- *In der dritten Zeile:* Wenn unsere Voraussetzung richtig ist ($p = 1$) und die die Folgerung falsch ist ($q = 0$), dann ist unsere aufgestellte Behauptung falsch.
- *In der vierten Zeile:* Wenn unsere Voraussetzung richtig ist ($p = 1$) und die die Folgerung richtig ist ($q = 1$), dann ist unsere aufgestellte Behauptung richtig.

Für unseren Beweis müssen wir noch klären, was "teilbar" überhaupt bedeutet. Daher definieren wir:

Definition 7.4 (Teilbarkeit)**.**
Eine ganze Zahl a ist durch eine ganze Zahl b *teilbar*, wenn es eine ganze Zahl k gibt mit $a = b \cdot k$ oder kürzer:
$a, b \in \mathbb{Z}$, b *teilt* $a :\Leftrightarrow$ es ex. k mit $a = b \cdot k$ und $k \in \mathbb{Z}$

Beispiel 7.5.
70 ist durch 10 teilbar, da $70 = 10 \cdot k$ für $k = 7$.

Beobachtungen 7.6.
Sei a eine ganze Zahl, die durch 10 teilbar ist. Dann hat a die Form $a = 10 \cdot k$ für eine ganze Zahl k.

Wir nehmen nun unsere Aussage p und formen diese Schritt für Schritt um.

p Eine Zahl ist durch 10 teilbar (Definition der Teilbarkeit)

a_1 $a = 10 \cdot k$ für eine ganze Zahl k ($10 = 2 \cdot 5$)

a_2 $a = 2 \cdot 5 \cdot k$ für eine ganze Zahl k ($k = 2 \cdot k$)

a_3 $a = 5 \cdot k'$ für eine ganze Zahl k' (Definition der Teilbarkeit)

q Eine Zahl ist durch 5 teilbar

Wir beobachten die Implikationsfolge $p \to a_1 \to a_2 \to a_3 \to q$, also $p \to q$. Nun verpacken wir unseren Beweis in einen schönen mathematischen Text:

Lemma 7.7.
Wenn eine Zahl durch 10 teilbar ist, dann ist sie auch durch 5 teilbar

Beweis. Sei a eine Zahl, die durch 10 teilbar ist. Dann hat a die Form $a = 10 \cdot k$ (nach Def. 7.4) für eine ganze Zahl k. Durch Umformen erhalten wir $a = 10 \cdot k = 5 \cdot 2 \cdot k = 5 \cdot k'$ für eine ganze Zahl k'. Nach Definition der Teilbarkeit ist die Zahl a auch durch 5 teilbar. □

7.2. Beweis durch Kontraposition

Nicht immer ist es einfach $p \to q$ direkt nachzuweisen. Daher überlegen wir uns:

Wir betrachten nun ein Spiel. Wir haben eine Menge von Karten. Auf der Vorderseite ist jeweils eine Farbe Blau oder Gelb und auf der Rückseite befindet sich jeweils eine Zahl. Für alle Karten soll nun gelten:

Wenn eine Karte vorne blau ist, dann ist die Zahl auf der Rückseite gerade

Wenn wir nun eine Karte ziehen, und auf der Rückseite dieser Karte befindet sich eine ungerade Zahl, welche Farbe wird sie dann haben?

Natürlich gelb. Wir formalisieren nun unseren Sachverhalt aussagenlogisch:

p Die Karte ist vorne blau

q Auf der Karte befindet sich eine gerade Zahl

Unsere Tatsache: "Wenn eine Karte vorne blau ist, dann ist die Zahl auf der Rückseite gerade", wäre aussagenlogisch formalisiert: $(p \to q)$. Wir betrachten nun die Negationen von p und q:

$\neg p$ Die Karte ist vorne gelb

$\neg q$ Auf der Karte befindet sich eine ungerade Zahl

Wir haben nun gesagt: "Wenn sich auf der Rückseite eine ungerade Zahl befindet, dann ist die Karte vorne gelb", daher gilt dann: $(\neg q \to \neg p)$. Wir betrachten unsere Wertetabelle:

p	q	$(p \to q)$	$\neg q$	$\neg p$	$(\neg q \to \neg p)$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	0	0	1

Fazit: $(p \to q) \equiv (\neg q \to \neg p)$, da die Spalteneinträgt von $(p \to q)$ und $(\neg q \to \neg p)$ übereinstimmen.

Wir können also $(\neg q \to \neg p)$ anstatt $(p \to q)$ nachweisen. Diese Beweistechnik nennen wir "**Beweis durch Kontraposition**".

Achtung: Die Implikationsfolge ändert sich. „Wenn eine Karte vorne blau ist, dann ist die Zahl auf der Rückseite gerade", ist äquivalent zu der Aussage, „Wenn die Zahl auf der Rückseite nicht gerade (also ungerade) ist, dann ist die Karte vorne nicht blau (also gelb)". Was mit der Rückseite ist, wenn die Vorderseite nicht blau (also gelb) ist, können wir anhand der obigen Regel nicht sagen. Gelbe Karten können auf der Rückseite sowohl gerade, als auch ungerade Zahlen haben.

Beispiel 7.8.
Wenn a^2 ungerade ist, dann ist a ungerade

Wir betrachten unsere Aussagen:

p a^2 ist eine ungerade Zahl

q a ist eine ungerade Zahl

und deren Negationen:

$\neg p$ a^2 ist eine gerade Zahl

$\neg q$ a ist eine gerade Zahl

Wir betrachten nun wieder unsere Aussagenfolge:

$\neg q$ a ist eine gerade Zahl (Jede gerade Zahl ist durch 2 teilbar)

a_1 a ist durch 2 teilbar (Definition der Teilbarkeit)

a_2 $a = 2 \cdot k$ für eine ganze Zahl k (quadrieren).

a_3 $a^2 = 2^2 \cdot k^2$ für eine ganze Zahl k. ($k' = 2 \cdot k^2$)

a_4 a^2 ist durch 2 teilbar

$\neg p$ a^2 ist eine ganze Zahl

Wir haben also $(\neg q \to \neg p)$ und somit auch $(p \to q)$ gezeigt.
Kompakt geschrieben:

Lemma 7.9.
Wenn a^2 ungerade ist, dann ist a ungerade

Beweis. Beweis durch Kontraposition
Wir nehmen an, dass a gerade ist. Dann ist a durch 2 teilbar und hat nach Definition der Teilbarkeit die Form $a = 2 \cdot k$ für eine ganze Zahl k. Durch Quadrieren erhalten wir $a^2 = 2^2 \cdot k^2 = 2 \cdot 2 \cdot k^2 = 2 \cdot k'$ mit $k' = 2 \cdot k^2$ für eine ganze Zahl k. Somit ist a^2 durch 2 teilbar und somit gerade. □

Anmerkung: Wenn wir versuchen würden unser Beispiel auf direktem Wege zu lösen, dann wäre dies sehr viel schwerer, da wir von Anfang an ein a^2 hätten und wir uns schwierig auf das a beziehen können. Es ist daher wichitg, die richtige Beweistechnik zu wählen, um sich viel Arbeit zu ersparen.

7.3. Beweis durch Widerspruch

Wiederrum gibt es Fälle, bei denen der direkte Beweis und der Beweis durch Kontraposition nicht geeignet sind. Wir benötigen daher eine dritte Beweistechnik, den Beweis durch Widerspruch:
Einführendes Beispiel: Supermärkte haben im Allgemeinen folgende Regel zu befolgen:

Wenn sie Alkohol an ihre Kunden verkaufen, dann müssen die Kunden $\geq$ 18 Jahre alt sein

Die Polizei führt des öfteren Kontrollen auf Supermärkte durch um die obige Aussage zu überprüfen. Dabei beauftragen diese Jungendliche, die jünger als 18 Jahre alt sind, in den Supermarkt zu gehen um Alkohol zu kaufen. Was muss nun passieren damit die obige Aussage stimmt? Sie dürfen den Alkohol **nicht** verkaufen, dann hat der Supermarkt seinen Test bestanden. Wir halten aussagenlogisch fest:

p Der Supermarkt verkauft Alkohol an seine Kunden

q Die Kunden sind $\geq$ 18 Jahre alt

$\neg q$ Die Kunden sind jünger als 18 Jahre

$(p \to q)$ Wenn sie Alkohol an ihre Kunden verkaufen, dann müssen die Kunden $\geq$ 18 Jahre alt sein

Die Polizei beauftragt Jugendliche Alkohol zu kaufen und diese Jugendlichen sind jünger als 18 Jahre. Wir erhalten: $(p \wedge \neg q)$. Wenn unter diesen Umständen der Supermarkt kein Alkohol verkauft, (Also wenn wir eine Falschaussage f haben, die unsere Behauptung widerlegt), dann erhalten wir insgesamt: $((p \wedge \neg q) \to f)$. Wir beweisen also $p \to q$ indem wir $((p \wedge \neg q) \to f)$ zeigen, also $(p \wedge \neg q)$ zu einer Falschaussage bzw. Widerspruch führen. Den Beweis für die Gültigkeit werden wir in einer Übung führen. Wir nennen diese Beweistechnik *Beweis durch Widerspruch.*

Beispiel 7.10.
Wenn a und b gerade natürliche Zahlen sind, dann ist auch deren Produkt $a \cdot b$ gerade.

Wir teilen unsere Aussage in Teilaussagen auf:

p a und b sind gerade natürliche Zahlen

q $a \cdot b$ ist gerade

$\neg q$ $a \cdot b$ ist ungerade

Wir wenden nun $(p \wedge \neg q) \rightarrow f$ an, wobei wir auf eine falsche Aussage f treffen müssen.

$p \wedge \neg q$: a und b sind gerade natürliche Zahlen und $a \cdot b$ ist ungerade

a_1 : a ist gerade und $b = 2 \cdot k$ für eine natürliche Zahl k und $a \cdot b$ ist ungerade

a_2 : a ist gerade und $a \cdot b = 2 \cdot k \cdot a$ für eine natürliche Zahl k' und $a \cdot b$ ist ungerade

a_3 : a ist gerade und $a \cdot b = 2 \cdot k'$ für eine natürliche Zahl $k' = k \cdot a$ und $a \cdot b$ ist ungerade.

f : a ist gerade und $a \cdot b$ ist gerade und $a \cdot b$ ist ungerade

Die Aussage f ist offensichtlich falsch, da $a \cdot b$ entweder gerade oder ungerade sein muss, aber nich beides sein kann.

Lemma 7.11.
Wenn a und b gerade natürliche Zahlen sind, dann ist auch $a \cdot b$ gerade.

Beweis. Beweis durch Widerspruch
Angenommen zwei natürliche Zahlen a und b sind gerade und $a \cdot b$ ist ungerade. Dann hat b die Form $b = 2 \cdot k$ für eine natürliche Zahl k nach Definition der Teilbarkeit . Multipliziert man diesen Ausdruck mit a, dann ergibt dies $a \cdot b = 2 \cdot k \cdot a$. Da $2 \cdot k$ wieder eine natürliche Zahl ergibt gilt $a \cdot b = 2 \cdot k'$ mit $k' = a \cdot k'$. Somit ist $a \cdot b$ gerade. **Widerspruch** □

7.4. Äquivalenzbeweis

Oftmals haben Aussagen die Form $p \leftrightarrow q$. Diese werden in der Mathematik durch "Genau dann wenn..." ausgedrückt.

Beispiel 7.12.
Genau dann, wenn a gerade ist, ist auch a^2 gerade.

Wir betrachten unsere Wertetabelle:

p	q	$p \leftrightarrow q$	$p \rightarrow q$	$q \rightarrow p$	$(p \rightarrow q) \wedge (q \rightarrow p)$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	1	1	1	1

Wir sehen, dass die dritte Spalte mit der letzten übereinstimmt. Somit können wir $p \leftrightarrow q$ mit $(p \rightarrow q) \wedge (q \rightarrow p)$ ersetzen.
In unserem Beispiel wäre das:

p : a ist gerade

q : a^2 ist gerade

$p \rightarrow q$: "Wenn a gerade ist, dann ist auch a^2 gerade"

$q \to p$: "Wenn a^2 gerade ist, dann ist auch a gerade"

Beweis. $\Rightarrow$ siehe Übung
$\Leftarrow$ siehe Beweis Lemma 7.9 □

Es gibt in der Informatik noch weitere wichtige Beweistechniken:

- Beweis durch vollständige Induktion (für natürliche Zahlen)
- Beweis atomarer Aussagen
- Beweis mit Fallunterscheidung
- ...

Beweise lernt ihr am Besten durch ÜBEN, ÜBEN und ÜBEN (i.d.R. werdet ihr wöchentlich mit mindestens einer Beweisaufgabe konfrontiert).
Tipps zum Beweisen:

- Es ist zu achten, dass keine Gedankensprünge in dem Beweis enthalten sind. Jede Folgerung, die man trifft, sollte klar sein
- Eine Angabe der Beweistechnik am Anfang hilft dem Leser dem Beweis zu folgen
- Beweise sollen möglichst präzise und genau sein, sodass der Leser die Gedankengänge vollständig nachvollziehen kann
- Eine Kennzeichnung am Ende eines Beweises (z.B. durch □ oder qed (quod erat demonstrandum)), auch wenn es dem Schreiber klar erscheint, ist für den Leser hilfreich
- Am Ende von längeren Beweisen ist ein kurzer Satz, was ihr gezeigt habt, hilfreich

8. Induktion und Rekursion

8.1. Vollständige Induktion

Ziel der vollständigen Induktion ist es zu beweisen, dass eine Aussage $A(n)$ für alle $n \in \mathbb{N}$ 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 führen wir abkürzende Schreibweisen für Summen und Produkte ein.

Definition 8.1.
Sei $n \in \mathbb{N}$, und seien $a_1, \ldots, a_n$ 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

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

  Produkt

Beispiel 8.2.

kleiner Gauß

Satz 8.3 (kleiner Gauß)**.**
$A(n)$*:Für alle* $n \in \mathbb{N}$ *gilt:*

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

Zum Beweis des Satzes bietet sich vollständige Induktion an, da wir eine Aussage für alle natürlichen Zahlen beweisen müssen. Für den Induktionsanfang setzen wir $n = 1$, setzen dies in die Formel ein und rechnen aus, ob die Behauptung für $n = 1$ stimmt.
Induktionsanfang: $n = 1$
Behauptung: $A(1)$: Der Satz gilt für $n = 1$.
Beweis:

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

Im Induktionsschritt nehmen wir nun an, dass die Behauptung für ein $n \geq 1$ gilt und zeigen, dass daraus folgt, dass die Behauptung auch für $n+1$, also die nächstgrößere Zahl wahr ist. Hierfür setzten wir für n $n+1$ ein und versuchen zu zeigen, dass die Gleichung stimmt. Es bietet sich ein *direkter Beweis* (vergl. 7.1) an. Wir formen den linken Teil der Gleichung nach allen Künsten der Mathematik schrittweise so um, dass wir am Schluss beim rechten Teil ankommen.
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:

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

□

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 8.4.

Satz 8.5.
Für alle $n \in \mathbb{N}$ und $x \neq 1$ gilt:

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

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

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

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

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

Beweis:

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

□

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

Beispiel 8.6.

Satz 8.7.
Sei $n \in \mathbb{N}$. $n^2 + n$ ist eine gerade (d.h. durch 2 teilbare) Zahl.

Beweis
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 ist als gerade Zahl definiert. Da die 0 aber so speziell ist, kann man zur Sicherheit und zur Übung den Satz auch noch für $n = 1$ beweisen.
Behauptung: $1^2 + 1$ ist eine gerade Zahl.
Beweis: $1^2 + 1 = 1 + 1 = 2$. 2 ist 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:

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

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

Beispiel 8.8.
Wir erinnern uns an Satz 3.20 aus Kapitel 3. Dieser lautet:
Die Wahrheitstabelle für eine aussagenlogische Formel mit insgesamt n Variablen hat genau 2^n Zeilen.

Nun wollen wir den Satz mit Hilfe der vollständigen Induktion beweisen. Da in einer Wahrheitstabelle eine Zeile genau einer möglichen Belegung der Variablen entspricht, müssen wir also zeigen, dass es für n Variablen genau 2^n verschiedene Belegungen gibt.

Beweis
Induktionsanfang: $n = 1$
Behauptung: für eine einzige Variable gibt es genau $2^1 = 2$ verschiedene Belegungen.
Beweis: Eine aussagenlogische Variable kann entweder den Wert 0, oder den Wert 1 annehmen (siehe Anmerkung 3.5). Somit gibt es für eine Variable A genau 2 mögliche Belegungen, nämlich $A = 0$ und $A = 1$.
Induktionsschritt: $n \to n + 1$
Induktionsvoraussetzung: Für $n \geq 1$ aussagenlogische Variablen gilt: es gibt genau 2^n verschiedene Belegungen.
Induktionsbehauptung: Wenn es für n Variablen genau 2^n verschiedene Belegungen gibt, dann gibt es für $n + 1$ Variablen genau 2^{n+1} verschiedene Belegungen.
Beweis: Betrachten wir die $n+1$ Variablen $A_1, A_2, \ldots, A_n, A_{n+1}$. Für die willkürlich herausgegriffene Variable A_x gibt es genau zwei Belegungen, nämlich $A_x = 0$ und $A_x = 1$. Nehmen wir also zunächst an, $A_x = 0$. Dann gibt es für die verbleibenden n Variablen $A_1, \ldots, A_{x-1}, A_{x+1}, \ldots, A_n$ laut Induktionsvoraussetzung genau 2^n Belegungsmöglichkeiten. Das gleiche gilt für den Fall $A_x = 1$. Somit gibt es für alle $n + 1$ Variablen insgesamt $2 \cdot 2^n = 2^{n+1}$ verschiedene Belegungen. □

8.1.1. Wann kann man vollständig 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 8.9.
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 $\frac{1}{n+1} > 0$? Das ist für den Induktionsschritt aber notwendig.

8.1.2. 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 ich den ersten Stein umstoßen kann und gesichert ist, dass jeder Stein n seinen Nachfolger $n + 1$ umstößt.

Dabei ist der Beweis, dass ich den ersten Stein umstoßen kann, $A(n)$ gilt für ein *bestimmtes* n, der *Induktionsanfang*, genau so wichtig, wie der *Induktionsschritt*.

Beispiel 8.10.
Aus der Aussage $A(5\ \textit{ist durch}\ 2\ \textit{teilbar})$ lässt sich ohne Weiteres 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 8.11 (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 Socken 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 8.12 (fehlerhafte Induktion)**.**
Behauptung: Alle Menschen einer Menge M mit $|M| = n$ sind gleich groß.
Induktionsanfang: $n = 1$
Behauptung: In einer Menge von einem Mensch, 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.

8.2. Rekursion

Rekursionsanfang
Rekursionsschritt

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.

Ein bekannter Spruch zur Rekursion lautet:

„Wer Rekursion verstehen will, muss Rekursion verstehen “

Definition 8.13 (Fakultätsfunktion)**.**
Die Funktion $f : \mathbb{N} \to \mathbb{N}$, gegeben durch:

Fakultätsfunktion

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

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 8.14.
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 8.13 gilt $f(0) = 1$. Nach Definition 8.1 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:

$$\begin{aligned} f(n+1) &= (n+1) \cdot f(n) && \text{Definition 8.13} \\ &= (n+1) \cdot \prod_{i=1}^{n} i && \text{Induktionsvoraussetzung} \\ &= (n+1) \cdot n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1 \\ &= \prod_{i=1}^{n+1} i \end{aligned}$$

□

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

Beispiel 8.15.
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: $\mathbb{N}_G = \{x | x = 2n, n \in \mathbb{N}\}$.

8.2.1. 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.

Beispiel 8.16.
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.

1. Monat → 1 Paar

2. Monat → 1 Paar

3. Monat → 2 Paare

4. Monat → 3 Paare

5. Monat → 5 Paare

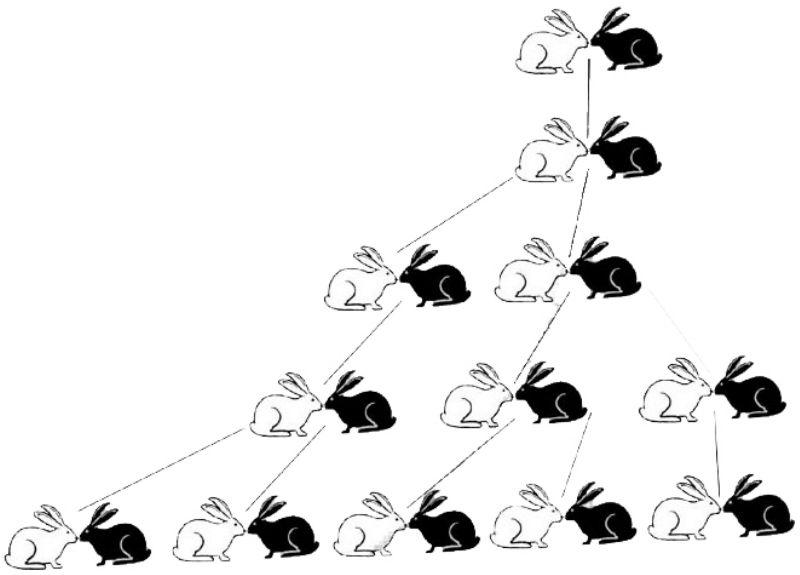


Tabelle 8.1.: Vermehrung der idealen Kaninchenpaare

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 8.17 (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.

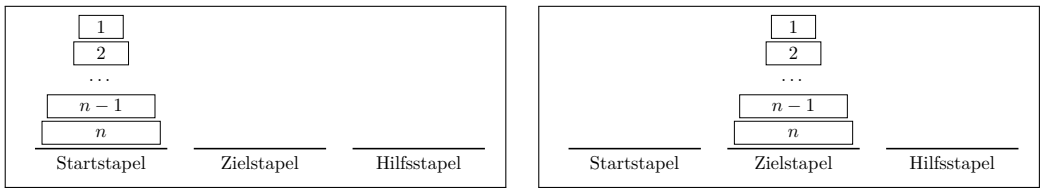
Beispiel 8.18.
Betrachten wir ein weiteres Beispiel:

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

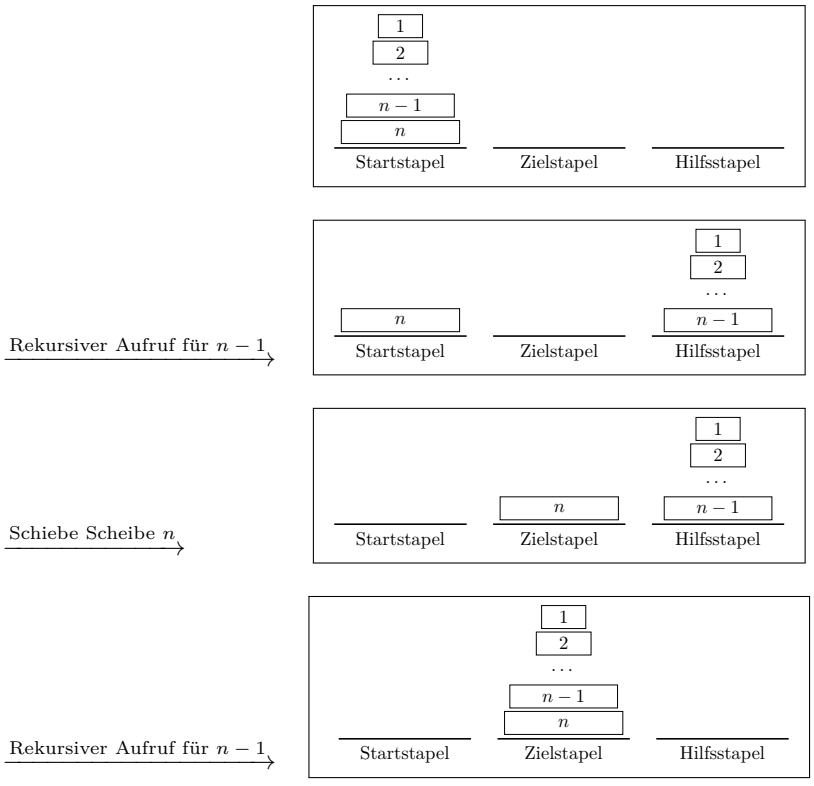
Türme von Hanoi

- 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 8.19.

```
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 vorker lieber die Korrektheit.

Korrektheit

Satz 8.20.
Algorithmus 8.19 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:

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

```
bewege n+1 Scheiben von A nach B, benutze C //Aufruf mit n+1
  bewege n Scheiben von A nach C, benutze B //Z.3, Aufruf mit n
  verschiebe Scheibe n+1 von A auf B        //Z.5
  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 8.21.
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 8.19 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. □

A. Anhang

A.1. Unix und Linux

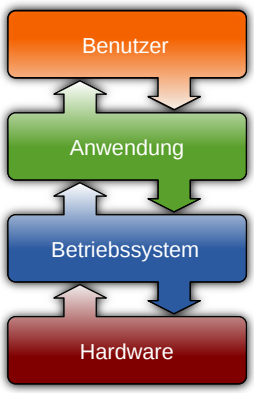


Abbildung A.1.

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. A.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 (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.

graphisches Terminal

GUI

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.

A.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*

übergeordnetes Verzeichnis

Verzeichnis (engl.: *parent directory*). Nur das *Wurzelverzeichnis* (engl.:*root directory*) ist in sich selbst enthalten. Es trägt den Namen „/“.

Wurzel-
verzeichnis

Beispiel A.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 A.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.

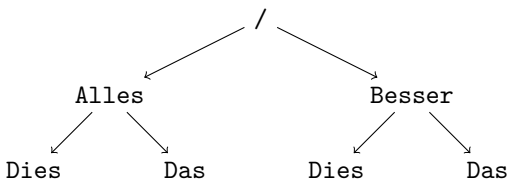


Abbildung A.2.: Ein Dateibaum

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 Verzeichnissen. 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* ecute). 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.

A.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.: 2.2).

A.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.

`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.

Passwort ändern

```
> 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.

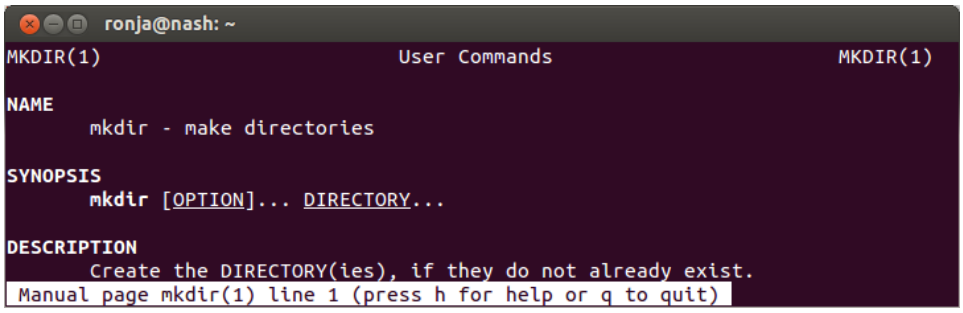


Abbildung A.3.: man page des Befehls `mkdir`

A.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. A.4).

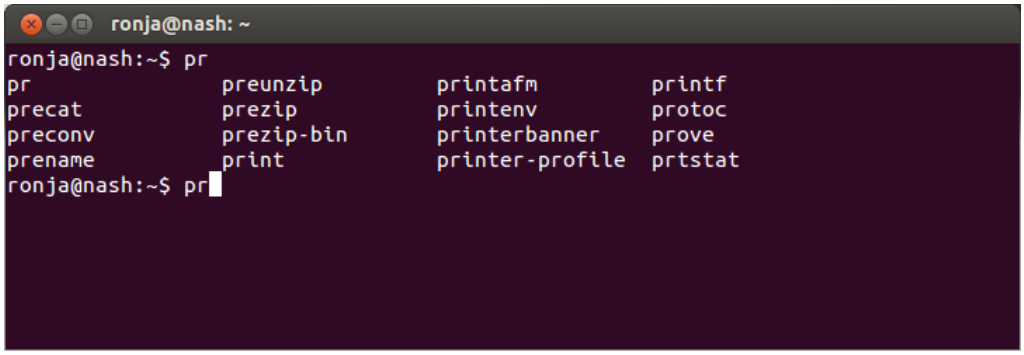


Abbildung A.4.: Autovervollständigung für die Eingabe `pr`

A.2. Entferntes Arbeiten

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 [myprog.py] [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:~/[datei] . ↵
   ```

   Der . steht dabei stellvertretend für das aktuelle Verzeichnis. Es ist auch möglich einen relativen oder absoluten Pfad zu einem anderen Verzeichnis anzugeben.

2. **Einloggen**

   ```
   > 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.5).

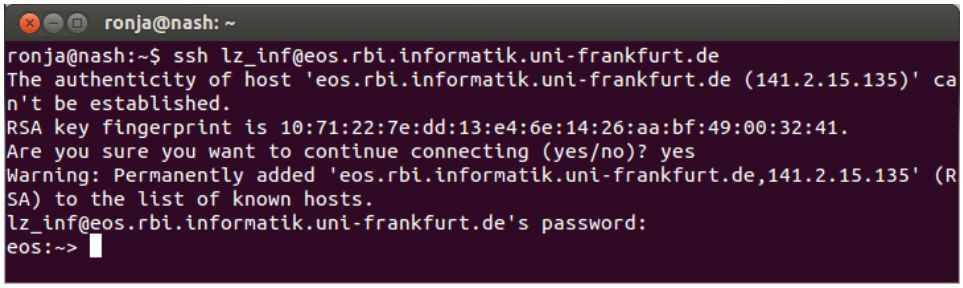
   

   Abbildung A.5.: Auf einem RBI-Rechner einloggen

3. **Programm starten**

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

   ```
   > python3 [meinprogramm.py] ↵
   ```

   gestartet werden.

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

4. **Verbindung beenden**

```
> exit ↵
```

Dieser Befehl schließt die Verbindung zu dem RBI-Rechner.

Abbildung A.6 zeigt das ganze am Beispiel unseres „Hello world!“-Programms welches unter dem Namen `hey.py` im Verzeichnis `/home/ronja/Documents/ME/` gespeichert ist.

Abbildung A.6.: Ein Python-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.7).

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

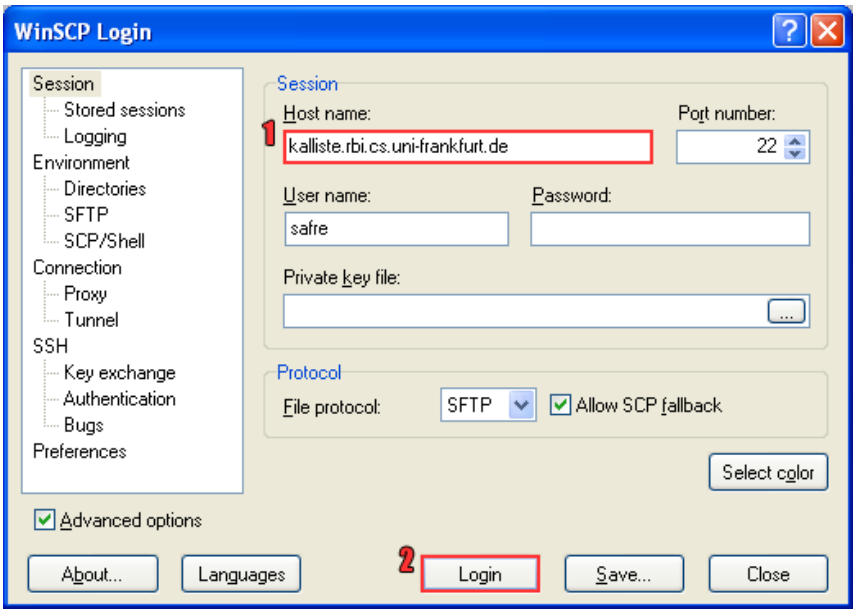


Abbildung A.7.: Das WinSCP-Login-Fenster

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.8).

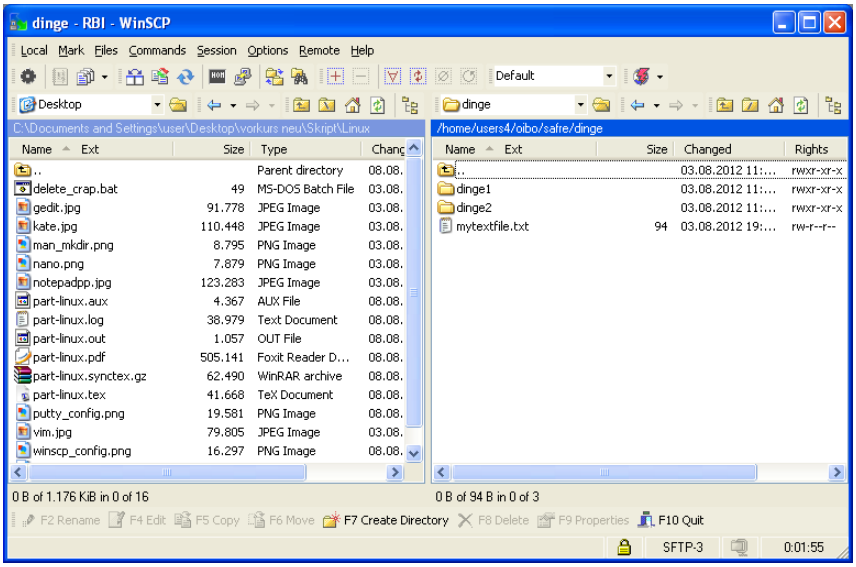


Abbildung A.8.: Das WinSCP-Hauptfenster. Links die Dateien des Windows-Rechners, rechts die Dateien im RBI-Verzeichnis

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.9).

[3]http://www.chiark.greenend.org.uk/∼ sgtatham/putty/download.html

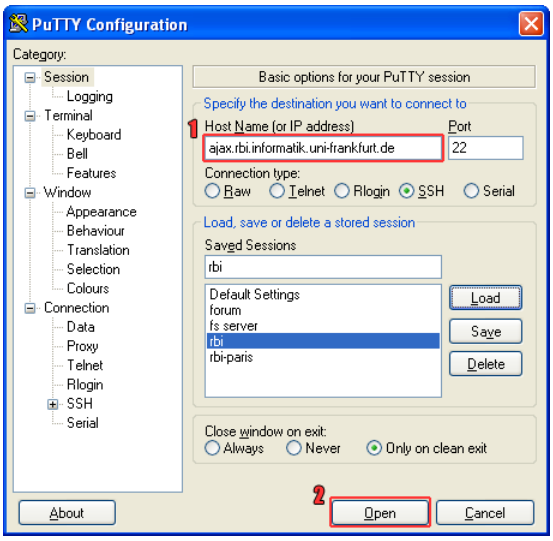


Abbildung A.9.: Einstellung von Putty

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 Python-Programm starten.

   ```
   > python3 [meinprogramm.py] ↵
   ```

4. **Verbindung beenden** Der Befehl `exit` schließt die Verbindung zum RBI-Rechner. Wenn die Trennung erfolgreich war, sollte sich das Putty-Fenster selbstständig schießen.

B. Zum Lösen von Übungsaufgaben

B.1. Neu im Studium

Das Studium geht los. Alles funktioniert irgendwie anders als gewohnt und neben den ganzen organisatorischen Fragen rund ums Studium sollen gleichzeitig noch die Vorlesungsthemen gelernt werden. Im Folgenden sollen einige (hoffentlich nützliche) Hinweise zur Bearbeitung der Übungsaufgaben den Start ins Informatik-Studium erleichtern.

Zu jeder Vorlesung gibt es dazugehörige Übungstermine, sogenannte Tutorien. Dort werden die wöchentlich oder zweiwöchentlich fälligen Übungsblätter zusammen mit einem Tutor (meist ein Student aus einem höheren Semester) besprochen und von den Teilnehmern oder dem Tutor vorgerechnet. Dort können auch Fragen zum Vorlesungsstoff oder zum nächsten Übungsblatt gestellt werden.

Die Lösungen werden also in den Tutorien vorgestellt. Dann kann ich also einfach abwarten, ins Tutorium gehen und dort zugucken, wie die richtige Lösung zur Aufgabe auf dem Übungsblatt vorgerechnet wird. Das ist völlig legitim. In fast allen Vorlesungen sind die Übungen freiwillig und keine Zulassungsvoraussetzung zur Klausur. Auch die von den Professoren gehaltenen Vorlesungen sind nur ein freiwilliges Angebot, um letztlich die Klausur erfolgreich bestehen zu können. Es gibt keine Anwesenheitspflichten.

B.1.1. Wozu soll ich die Übungsaufgaben überhaupt (selbst) machen?

Stellt Euch vor, Ihr müsstet die praktische Führerscheinprüfung machen ohne vorher je am Steuer gesessen zu haben. Stattdessen habt Ihr von der Rückbank aus zugeschaut und konntet sehen, wie das Auto fährt.
Warum würdet Ihr trotzdem aller Wahrscheinlichkeit nach durch die Prüfung fallen?

1. „Der Fahrlehrer hatte keine Ahnung!“ oder
2. „Ihr habt nur gesehen, dass das Auto fährt, aber Ihr habt nicht selber an den Pedalen, dem Lenkrad, den Spiegeln und der Gangschaltung geübt.“ oder
3. „Es waren einfach zu wenige Fahrstunden. Wenn ich nur oft genug zuschaue, dann schaffe ich die Prüfung locker!“.

Zwei Antworten waren Quatsch und Ihr wisst sicherlich, welche beiden das waren. Ihr müsst in Fahrstunden selbst am Steuer sitzen, weil Ihr sonst das Autofahren nicht lernt. Und aus dem gleichen Grund solltet Ihr auch die Übungsaufgaben machen. Nur so könnt Ihr die nötigen Fähigkeiten und Kenntnisse erwerben, die für das Bestehen der Prüfung nötig sind. Wenn Ihr von den Tutoren Eure bearbeiteten Übungsaufgaben korrigiert zurück erhaltet, habt Ihr außerdem ein zeitnahes Feedback über Euren Wissenstand. Könnt Ihr den Stoff anwenden und auch Transferaufgaben lösen? Wo liegen Eure Fehler? Was habt Ihr gut gemacht? Welche Schritte waren nötig bis Ihr auf Eure Lösung gekommen seid, welche Fehlversuche habt Ihr unternommen und was habt Ihr aus den Fehlern gelernt? All dies entgeht Euch, wenn Ihr die Übungsaufgaben nicht oder nur halbherzig bearbeitet. Außerdem werdet Ihr nicht nur dadurch belohnt, dass Ihr viel bessere Chancen habt, den Vorlesungsstoff zu verstehen und eine gute Note in der Klausur zu erzielen. Eure Note wird sogar noch besser ausfallen, weil es in vielen Vorlesungen Bonuspunkte für das Lösen und

Vorrechnen der Aufgaben in den Tutorien gibt. So könnt Ihr teilweise bis zu 20% der Klausurpunkte schon vor dem tatsächlichen Klausurtermin sicher haben und braucht am Prüfungstag nur noch weitere 30% der Klausur lösen, um sie zu bestehen. Die genauen Bonuspunkte-Regelungen hängen von der jeweiligen Vorlesung ab und werden vom Dozenten festgelegt. Manchmal werden die Bonuspunkte erst auf eine bestandene Prüfung angerechnet, z. B. indem sich die Note 4.0 um eine Notenstufe auf 3.7 verbessert oder die 1.3 auf eine 1.0.

B.1.2. Was halten wir fest?

Das Lösen der Übungsblätter lohnt sich und ist ein wichtiger Bestandteil eines erfolgreichen Studiums! Hier investierte Zeit ist sehr gut angelegt.

B.1.3. Konkrete Tipps

Betrachtet das folgende Szenario: Pro Woche gibt es zwei Vorlesungstermine, zu denen ein Dozent die Vorlesungsinhalte vorträgt. In der Regel wird kurz ein allgemeines Konzept erläutert, ein Beispiel dazu gebracht und dann mit dem nächsten Konzept fortgefahren. Ihr seid die meiste Zeit in der Zuhörerrolle. Aktiv werdet Ihr erst, wenn es an das Bearbeiten des Übungsblattes geht, das den Stoff aus der aktuellen Vorlesungswoche behandelt. Dafür habt Ihr meistens eine Woche Zeit. Wie solltet Ihr vorgehen, damit Ihr am Ende der Woche alle Aufgaben gelöst habt und Eure Bonuspunkte sammeln könnt?

- Frühzeitig anfangen! Gute Lösungen brauchen Zeit. Im Idealfall habt Ihr Euch das Übungsblatt schon vor der Vorlesung angeschaut. Dann könnt Ihr nämlich ganz gezielt nach den Informationen Ausschau halten, die Ihr für die Aufgaben des Übungsblattes benötigt.
- In der Vorlesung zuhören und Fragen stellen.
- Die Aufgaben vollständig lesen. Nicht bekannte Begriffe nachschlagen.
- zusätzliche Materialien benutzen (Skript, Folien etc.)
- Die Übungsaufgaben gemeinsam in einer Gruppe besprechen und verschiedene Lösungsalternativen vergleichen; am Ende sollte jeder, unabhängig von den anderen, seine eigene Lösung aufschreiben. Insgesamt kann man den Wert einer guten Arbeitsgruppe gar nicht genug betonen.
- Nicht abschreiben! Es klingt besserwisserisch, aber damit betrügt Ihr Euch selbst und lernt nichts. Außerdem setzt Ihr damit Eure Bonuspunkte aufs Spiel.
- Schreibt Eure Lösungen ordentlich auf. Falls Ihr eine unleserliche Handschrift habt, dann arbeitet daran. Wenn die Korrektur Eurer Lösung für den Tutor zu reiner Dechiffrierung verkommt, dann heißt es im Zweifelsfall, dass Ihr darauf keine Punkte erhaltet. Spätestens in der Klausur wäre das sehr ärgerlich. Und auch wenn Eure Handschrift einem Kunstwerk gleicht und Ihr in der Schule immer ein Sternchen für das Schönschreiben erhalten habt solltet Ihr am Ende trotzdem noch einmal alle Eure Lösungsnotizen auf frischen DIN-A4-Blättern als endgültige Version aufschreiben.
  Wenn Ihr Eure Lösungen am Computer verfassen wollt, dann bietet sich insbesondere in Vorlesungen mit einem großen Anteil an mathematischer Notation das Textsatzsystem LaTeX[1] an. Die meisten Skripte und Übungsblätter werden auf diese Weise verfasst und spätestens für Seminar- oder Abschlussarbeiten sind LaTeX-Kenntnisse von großem Vorteil. Für die gängigen Betriebssysteme gibt es komfortable Entwicklungsumgebungen, die Ihr nutzen könnt, aber nicht müsst. Es gibt auch Menschen, die es lieber einfach halten und nur mit dem Texteditor ihrer Wahl arbeiten.

[1] http://de.wikipedia.org/wiki/LaTeX

- Und zu guter Letzt: Habt Geduld und Durchhaltevermögen! Gerade am Anfang erscheint manches viel schwieriger als es später im Rückblick eigentlich ist. Weitere Tipps zum Thema Übungsaufgaben findet Ihr im folgenden Text von Prof. Dr. Manfred Lehn von der Universität Mainz: `http://www.mathematik.uni-mainz.de/Members/lehn/le/uebungsblatt`. Er spricht dort zwar von „Mathematik“, aber der Text ist genauso auf „Informatik“ anwendbar.

Mit der Zeit werdet Ihr Euch an die universitäre Arbeitsweise gewöhnen und Techniken entwickeln, mit denen Ihr effektiv und effizient arbeiten könnt. Nicht alles funktioniert für jeden. Das hängt von vielen Faktoren ab, beispielsweise von Eurem Lerntyp.[2] Die meisten Menschen sind vorwiegend visuelle Lerntypen, aber es gibt auch Personen, die am besten durch reines Zuhören lernen und wieder andere müssen etwas im wahrsten Sinne des Wortes „begreifen“, um es zu verstehen. Seid Euch über diese Unterschiede bewusst, wählt dementsprechend Eure Lerntechniken, und geht auf Eure Lernpartner ein. Verbindet dabei ruhig auch verschiedene Eingangskanäle, beispielsweise indem Ihr Euch über eine bestimmte Aufgabe unterhaltet, gleichzeitig auf einem Blatt Papier oder an der Tafel das Gesprochene festhaltet und dabei selbst aktiv seid. So bringt Ihr das Gehirn auf Trab und sorgt dafür, dass möglichst viel hängenbleibt.

B.2. Wie geht man an eine Aufgabe in theoretischer Informatik heran?

Wer ein Informatik-Studium an der Goethe-Universität aufnimmt, wird im ersten Semester neben den Programmier- und Mathematik-Vorlesungen auch auf eine Vorlesung aus dem Bereich Theoretische Informatik treffen. Für diejenigen, die im Wintersemester beginnen, ist das die Diskrete Modellierung und im Sommersemester sind es die Datenstrukturen. Auch hier gibt es Übungsaufgaben zu lösen, um insbesondere mathematische Methoden angewandt auf informatische Aufgabenstellungen zu erlernen. Gerade am Anfang stellt die mathematische Genauigkeit für viele eine Hürde da, die es zu meistern gilt. Auch hier ist die Devise: Üben, üben, üben! Doch was ist hier anders als bei den Programmierübungsaufgaben?

Zunächst das Offensichtliche: Es handelt sich um Theorie-Aufgaben. Es muss also nichts programmiert werden in dem Sinne, dass Programmcode über die Tastatur in den Computer eingegeben werden muss und am Ende ein syntaktisch korrektes Programm rauskommt. Stattdessen geht es um theoretische Ideen, die man später für die eigentliche Programmierarbeit verwenden kann. Diese Ideen können mal mehr und mal weniger offensichtlich mit der Programmierung zu tun haben und manchmal geht es auch nur darum, die analytisch-logische Denkweise zu trainieren und für sich selbst Techniken zu entwickeln, wie man am Besten an komplexe Aufgabenstellungen herangeht.

B.2.1. Die Ausgangssituation

Das neue Übungsblatt wurde auf der Vorlesungshomepage veröffentlicht oder in der Vorlesung verteilt.

Welche Ziele wollen wir erreichen?

- Alle Aufgaben richtig lösen.
- Innerhalb der Abgabefrist.
- Dadurch den Vorlesungsstoff so gut verstehen, dass wir ihn auch selbst erklären können.

Was brauchen wir dafür?

[2] `http://www.philognosie.net/index.php/article/articleview/163/`

- Papier, Stift, Mülleimer und Gehirn

Zunächst ist kein Unterschied zu den Praxis-Aufgaben erkennbar. Dort haben wir dieselben Ziele und verwenden ebenfalls Papier und Stift, um Lösungsansätze zu skizzieren, den Mülleimer, da nicht jeder Lösungsansatz zum Erfolg führt und sowieso ist es ein sehr guter Rat, das Gehirn zu benutzen.
Was ist nun der große Unterschied? Wir können nicht mithilfe eines Compilers oder Interpreters am Computer testen, ob unsere Lösung korrekt ist. Davon müssen wir uns selbst überzeugen oder wir müssen warten bis wir vom Tutor unser korrigiertes Übungsblatt zurück erhalten. Wir können daher nicht wie beim Programmieren „einfach mal probieren und gucken, was passiert“, da der direkte Feedback-Kanal fehlt. Aber genau dies ist der Sinn der Theorie-Aufgaben. Bei größeren Software-Projekten können wir nicht einfach auf gut Glück programmieren, um dann beim Test festzustellen, dass unser Ansatz schon grundlegend ungeeignet war. Wir müssen vorher nachdenken, analysieren und Lösungen mittels mathematischer Notationen zu Papier bringen.

B.2.2. Konkret: Wie legen wir los?

Wir können grundsätzlich die gleichen Tipps befolgen, die auch schon in dem allgemeinen Abschnitt über die Bearbeitung der Übungsaufgaben dargestellt werden. Da uns der direkte Computer-Feedback-Kanal fehlt, müssen wir von den Folien und Skripten Gebrauch machen. Das ist keine Option, sondern in den allermeisten Fällen dringend erforderlich. In der Theoretischen Informatik haben wir es mit exakten Definitionen und Schreibweisen zu tun, an die wir uns halten müssen. Trotzdem haben wir ebenso viele Freiheiten beim Aufschreiben wie das z. B. beim Programmieren der Fall ist. Es gibt jedoch eine ganze Reihe von typischen Anfängerfehlern, die wir im Folgenden kennenlernen werden.

Fehler:	**Benutzung undefinierter Symbole**
Erklärung:	Im Text tauchen plötzlich Bezeichnungen auf, ohne dass vorher festgelegt wurde, was die Bezeichnungen bedeuten sollen.
Lösung:	Alle Bezeichnungen vor ihrer Benutzung definieren und ggf. erlaubte Wertebereiche angeben.
Beispiel:	„Berechne nun $100/n$.“ Was ist n? Wahrscheinlich eine Zahl, aber bestimmt nicht 0. Wir müssen sicher stellen, dass die Rechnung gültig ist und ergänzen: „Sei $n \in \mathbb{R} \setminus \{0\}$. Berechne nun $100/n$.“
Fehler:	**Verstoß gegen Definitionen**
Erklärung:	Eine Bezeichnung wurde früher im Text als ein gewisser „Datentyp“ definiert. Später wird diese Definition ignoriert und ein falscher „Datentyp“ verwendet.
Lösung:	Definitionen ggf. im Skript nachschlagen oder die eigenen Definitionen beachten.
Beispiel:	„Seien $A := \{1, 2, 3\}$ und $B := \{2, 4, 6\}$ Mengen und sei $f : A \to B$ eine Funktion mit $f(a) := 2 \cdot a$ für alle $a \in A$. Berechne nun $f(4)$.“ Jetzt könnten wir denken, dass $f(4) = 2 \cdot 4 = 8$ ist, aber es gilt $4 \notin A$ und deshalb verstoßen wir hier gegen die Definition der Funktion f.

Fehler:	**Gedankensprünge**
Erklärung:	Innerhalb eines Beweises kommt es zu Argumentationslücken. Meistens liegt das daran, dass der Start und das Ziel des Beweises schon laut Aufgabenstellung bekannt sind, wir aber den Weg dazwischen nicht finden und dann versuchen, uns durchzumogeln nach dem Motto: „Das merkt der Tutor sowieso nicht.“ ;-) Doch, das merkt der Tutor und dann muss er dafür Punkte abziehen.
Lösung:	So oft wie möglich „warum?“ fragen. Welche Definitionen können wir verwenden? Welche konkreten logischen Schlussfolgerungen können wir aus den Definitionen ziehen. Warum geht das? Wie geht es dann weiter?
Beispiel:	„Es gelte Aussage X. Zu zeigen: Aus X folgt Aussage Y. Beweis: Da X gilt, ist es ziemlich logisch, dass auch Y gilt. q.e.d.“
Fehler:	**Inkonsistente Bezeichnungen**
Erklärung:	Das Problem tritt häufig auf, wenn Studenten sich bei der Lösung einer Aufgabe aus mehreren Quellen „inspirieren“ lassen. Häufig werden Definitionen und Schreibweisen verwendet, die nicht zu den in der Vorlesung vereinbarten passen.
Lösung:	Wenn wir andere Quellen nutzen, sollten wir diese nicht blind übernehmen. Stattdessen sollten wir versuchen, das dort Geschriebene zu verstehen und auf die Aufgabenstellung des Übungsblattes zu übertragen. Dafür muss die ggf. die Notation angepasst werden, damit die eigene Lösung zur Aufgabenstellung passt. Oft sind die Quellen auch unvollständig und/oder fehlerhaft. Wenn wir die Aufgabe wirklich verstanden haben, können wir diese Fehler ohne Probleme beseitigen und haben dabei sogar noch etwas gelernt.
Beispiel:	In der Aufgabenstellung ist von einer Menge A die Rede und in der „eigenen“ Lösung heißt die gleiche Menge auf einmal X. Einfach nur X durch A zu ersetzen und dann den Rest der Quelle abzuschreiben ist natürlich nicht das Ziel der Aufgabe. Außerdem können wir uns sicher sein, dass andere Studenten das genauso tun werden und der Tutor bei der Korrektur der Übungsblätter stutzig wird, weil er immer wieder die gleichen (falschen) Formulierungen liest und sich dann mit dem Betrugsversuch herumärgern muss. Bei einem solchen Verhalten sind die Bonuspunkte in Gefahr, also lieber auf Nummer sicher gehen und nichts riskieren.

B.2.3. Wann können wir zufrieden sein?

Die obige Liste mit Fehlern ist keineswegs vollständig. Es können jede Menge Kleinigkeiten schiefgehen, an die wir im ersten Augenblick nicht unbedingt denken. Um diesen Fehlern vorzubeugen, ist es ratsam, wenn wir versuchen, unsere Lösung aus der Sicht einer dritten Person zu betrachten. Alternativ lassen wir unsere Lösung von einer hilfsbereiten dritten Person selbständig durchlesen. Eine gut aufgeschriebene Lösung sollte diese Person mit wenig Aufwand lesen und verstehen können. Um zu kontrollieren, ob unsere Lösung gut aufgeschrieben ist, können wir uns an den folgenden Leitfragen orientieren.

- Können wir die aufgeschriebene Lösung auch ohne Kenntnis der Aufgabenstellung nachvollziehen?
- Ist uns wirklich jeder Schritt klar?
- Lassen die Formulierungen Interpretationsspielraum zu? Was könnten wir absichtlich falsch verstehen? Wie können wir präziser formulieren?
- Haben wir die Korrekturanmerkungen des Tutors von den vorherigen Aufgabenblättern beachtet? Tauchen die alten Fehler noch auf?

- Benutzen wir die Sprache des Professors? Ist die Lösung so aufgeschrieben, dass sie auch Teil des Skriptes sein könnte?

Falls wir bei der Beantwortung der Fragen Ergänzungen unserer Lösung vornehmen müssen und so viel frischen Text in unsere bisher aufgeschriebene Lösung quetschen, dass es unübersichtlich wird, sollten wir die neue Version der Lösung noch einmal sauber aufschreiben. Jede schöne Lösung kann auch schön aufgeschrieben werden.

B.3. Von der Idee zum fertigen Programm

B.3.1. Was ist die Ausgangssituation?

Informatik = Programmieren? Nein, genauso wenig wie Kunst = Malen ist, aber das Programmieren ist ein wichtiges Werkzeug im Informatik-Studium. Programmierkenntnisse benötigen wir, um die Ideen und Konzepte aus den Vorlesungen praktisch mithilfe von Computern umsetzen zu können. Und diese Ideen und Konzepte nehmen an der Universität einen großen Teil des Informatik-Studiums ein.

Die ersten Vorlesungen gehen los. Ziel: Wir sollen Programmieren lernen. Wie? Hier, das wöchentliche Übungsblatt. Dort steht:

Aufgabe B.1.
Schreiben Sie eine Funktion `fak` *in Python, die als Eingabe eine positive, natürliche Zahl n erhält und als Ergebnis den Wert $n!$ zurückgibt.*

Wird jetzt also vorausgesetzt, dass wir schon programmieren können? Wie sonst sollten wir denn diese Aufgabe überhaupt lösen können? Also stimmt es doch, dass man nur Informatik studieren kann, wenn man schon jahrelang programmiert hat und in der Schule Informatik im Leistungskurs hatte!
Keine Panik! Das gehört auch zum Studium: Informatiker kriegen ein großes Problem vorgesetzt und sollen es lösen. Dafür zerhacken sie es in kleine, einfachere Teilprobleme, die sie leichter lösen können. Die Teillösungen werden dann wieder zu einer großen ganzen Lösung für das ursprüngliche Problem zusammengesetzt. Wie sieht das hier aus? Schauen wir uns Aufgabe B.1 noch einmal an: Diese Aufgabe ist unser großes Problem. Wie zerhacken wir es nun in die kleineren, einfacheren Teilprobleme? Das kommt ganz drauf an, wie viel Erfahrung wir mit dem Programmieren und der logisch-strukturierten Denkweise haben, die im Informatikstudium trainiert wird. An dieser Stelle fangen wir mit ganz kleinen Schritten an.

B.3.2. Schritt für Schritt

- schreiben
- Funktion fak
- Python
- Eingabe
- positive, natürliche Zahl n
- Ergebnis
- $n!$
- zurückgeben

All diese Begriffe müssen wir präzise in unseren eigenen Worten erklären und mit konkreten Arbeitsschritten verbinden können.
„Schreiben Sie ...“ bedeutet „Programmieren Sie ...“, wir müssen also Programmcode schreiben. Es wird zwar nicht gesagt, ob wir das auf Papier oder auf dem Computer oder tun sollen, aber

mit Sicherheit kann das Feedback vom Computer hilfreich sein. Er kann uns auf bestimmte Programmierfehler aufmerksam machen. Weiter geht es mit „eine Funktion `fak`“. Funktionen kennen wir aus der Mathematik. Dort haben wir bestimmt schon so etwas gesehen: $f(x) = x^2$. Diese Funktion mit dem Namen f nimmt einen Wert x entgegen und rechnet das Quadrat x^2 aus. In Programmiersprachen verhält es sich ähnlich. In dieser Aufgabe heißt die Funktion allerdings nicht f, sondern `fak`. Diese Funktion sollen wir in der Programmiersprache Python schreiben. Wie sehen Funktionen in Python aus?

```
def funktionsname(eingabe):
    ...
    return ergebnis
```

Das ist das Grundgerüst unseres Programmes. Wir müssen es nur auf das aktuelle Problem anpassen. In der Aufgabe wird von einer Eingabe namens n gesprochen. Das ist das, was im obigen Code in den Klammern nach dem Funktionsnamen steht. In dem mathematischen Beispiel $f(x) = x^2$ ist die Eingabe das x bei $f(x)$. Unsere Funktion `fak` soll nun ein Ergebnis ausrechnen, nämlich $n!$ und es zurückgeben. Dieses Zurückgeben wird mit dem `return`-Befehl erreicht.

```
def fak(n):
    ...
    hier muss n! ausgerechnet werden
    ...
    return ergebnis
```

So sind wir schon ein ganzes Stück näher an die Lösung gekommen. Immerhin kennen wir jetzt den äußeren Rahmen des Programms. Jetzt geht es weiter ins Detail: n soll irgendeine positive, natürliche Zahl sein, d. h. n ist 1 oder 2 oder 3 oder 4 usw. Das können wir mit Python allerdings nicht ohne Weiteres erzwingen, da Python eine dynamisch typisierte Sprache ist. Diese Information ist nur für uns gedacht. Die Funktion `fak` soll $n!$ ausrechnen. Was bedeutet das Ausrufezeichen?

Definition B.1 (Fakultät)**.**
Für jede positive, natürliche Zahl n ist

$$n! := n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1.$$

Beispiel B.2.
Verdeutlichen wir uns diese Definition an einigen Beispielen:

- $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$
- $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
- $100! = 100 \cdot 99 \cdot 98 \cdot \ldots \cdot 3 \cdot 2 \cdot 1$
- $1! = 1$
- $0!$ ist laut der obigen Definition nicht möglich.

Die mathematische Definition müssen wir in Python-Code ausdrücken. Für die Eingabe n müssen wir alle Zahlen von 1 bis n aufmultiplizieren. Auch hier können uns Schreibweisen aus der Mathematik helfen:

$$n! = \prod_{i=1}^{n} i.$$

Was passiert hier? Wir benutzen eine Variable i, die schrittweise alle Zahlen $1, 2, 3, \ldots, n$ durchläuft. Dies entspricht in der Programmierung gerade einer Schleife mit einem Zähler i. In Python können wir dieses Produkt mit dem folgenden Code realisieren:

```
prod = 1
for i in range(1,n+1):
    prod = prod * i
```

Diesen Code müssen wir nun in den Code der Funktion `fak(n)` einbauen und schon sind wir fast fertig:

```
def fak(n):
    prod = 1
    for i in range(1,n+1):
        prod = prod * i
    return ergebnis
```

Tut der Code was er soll? Für jede positive, natürliche Zahl n wird die Fakultät $n!$ berechnet und in der Variable `prod` gespeichert. Anschließend geben wir die Variable `ergebnis` zurück. Problem: Welcher Wert steht in `ergebnis`? Das wissen wir nicht. Eigentlich sollte dort $n!$ drin stehen. Das müssen wir noch beheben, z. B. indem wir nicht die Variable `ergebnis` mittels `return` zurückgeben, sondern die Variable `prod`.

```
def fak(n):
    prod = 1
    for i in range(1,n+1):
        prod = prod * i
    return prod
```

Nun sind wir fertig und haben eine Lösung für die Aufgabe gefunden. Das ist aber nur eine Lösung, nicht zwingenderweise die einzige Lösung. Statt uns vom Produkt $n! = \prod_{i=1}^{n} i$ inspirieren zu lassen, hätten wir auch eine andere Beobachtung in Python-Code umsetzen können:

$$\begin{aligned} 1! &= 1 \quad \text{und} \\ n! &= n \cdot (n-1)!, \qquad \text{falls } n \geq 2 \end{aligned}$$

Die Fakultät von n können wir rekursiv berechnen, indem wir n mit der Fakultät von $n-1$ multiplizieren. Wie das geht, überlassen wir dem Leser als Hausaufgabe.
Viel Spaß beim Programmieren!

B.4. Wie geht man ein größeres Programmierprojekt an?

Im Laufe eines Informatik-Studiums, im Berufsleben oder in Freizeit-Projekten werden wir damit konfrontiert sein, komplexe Anforderungen in Software umzuwandeln. Wir sind die Compiler, die die echte Welt in Quellcode übersetzen müssen. Dafür gibt es kein automatisiertes Verfahren und es ist eher unwahrscheinlich, dass sich dies in absehbarer Zeit ändert. Allerdings können wir uns in der Vorgehensweise an den im Kapitel „Programmieren und Programmiersprachen“ vorgestellten Schritten orientieren:

1. Zerlegung des Problems in Teilprobleme
2. Präzisierung der Teilprobleme
3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem
4. Entwurf eines groben Programmgerüsts
5. Schrittweise Verfeinerung

6. Umsetzung in eine Programmiersprache

7. Testen und Korrigieren von Fehlern

Diese sieben Schritte stellen lediglich einen Vorschlag dar und sind keinesfalls die einzig richtige Methode, ein größeres Programmierprojekt zu bewältigen. Softwareentwickungsprozesse sind komplex und es wäre viel zu kurz gegriffen zu glauben, dass man sie immer in sieben aufeinander folgende Schritte unterteilen könne. Häufig ändern sich während des Projekts die Anforderungen oder neue Aspekte kommen hinzu, wodurch wir einzelne Phasen der Entwicklung durchaus mehrfach wiederholen müssen. Das ist völlig normal und kein Zeichen für eine schlechte Methodik. Außerdem lassen sich die Phasen nicht immer klar abgrenzen und haben fließende Übergänge.
Wir werden nun die sieben oben erwähnten Phasen anhand eines Beispiel-Projekts näher kennenlernen.

Projekt	**Tic Tac Toe** Das Spiel *Tic Tac Toe* soll unter Benutzung der Programmiersprache Python implementiert werden.
Spielregeln:	Zwei Spieler setzen auf einem 3×3-Spielfeld abwechselnd ein Kreuz (X) bzw. einen Kreis (O) in ein leeres Feld. Es gewinnt derjenige, der zuerst drei seiner Symbole in einer horizontalen, vertikalen oder diagonalen Reihe hat.
Anforderungen:	Das Spiel soll als Konsolenanwendung realisiert werden. Eine grafische Benutzeroberfläche ist nicht notwendig. Beim Start des Spiels soll der Benutzer mit einer kurzen Willkommensnachricht begrüßt werden, in der erklärt wird, mit welchen Eingaben er eine neue Partie starten oder das Spiel wieder beenden kann. Zusätzlich soll die Möglichkeit bestehen, die Spielregeln abzurufen. Bei Fehleingaben innerhalb des Willkommen-Menüs soll der Benutzer darauf hingewiesen werden und weiterhin im Willkommen-Menü verbleiben. Beim Start einer neuen Partie soll zunächst nach den teilnehmenden Spielern und deren Namen gefragt werden. Es kann sich wahlweise um menschliche oder Computer-Spieler handeln. Folgende Kombinationen sind möglich: Mensch-Mensch, Mensch-Computer, Computer-Mensch, Computer-Computer. Letzteres ist die *War Games*[3]-Variante *„A strange game: The only winning move is not to play“*. Während einer Partie müssen die Spielregeln eingehalten werden und nach jedem Zug muss das Spielfeld auf die Siegbedingung untersucht werden. Computer-Spieler sollten sich möglichst intelligent verhalten und den Eindruck erwecken, sie müssten auch „nachdenken“. Sollte ein Spieler gewonnen haben, so ist die Partie beendet und eine Meldung erscheinen, welcher Spieler gesiegt hat. Im Anschluss soll der Benutzer entweder eine weitere Partie mit den gleichen Spielern starten, zurück zum Willkommen-Menü gehen oder das Spiel beenden können. Wenn das Spiel beendet wird, soll zunächst für einige Sekunden ein Abspann auf dem Bildschirm erscheinen.

Der jetzige Stand der Dinge ist, dass uns ein Blocktext mit den Projektanforderungen vorliegt. Er ist nicht vollständig und lässt Interpretationsspielräume offen. Beispielsweise wird wohl jeder seine eigene Vorstellung davon haben, wie der Willkommen- oder der Abspann-Bildschirm aussehen könnte. Es gibt daher nicht *die* richtige oder *die* falsche Implementierung, sondern nur verschiedene Realisierungen der gleichen Anforderung. Widmen wir uns nun dem Sieben-Phasen-Modell.

[3] `http://www.imdb.com/title/tt0086567/`

B.4.1. Zerlegung des Problems in Teilprobleme

Der Projektbeschreibung nach können wir mehrere Teilprobleme ausmachen. Wir haben das eigentliche Spiel *Tic Tac Toe* mit seinen Spielregeln. Dieses Spiel soll im Programm in eine textbasierte Benutzeroberfläche (engl. *command line interface (CLI)*) eingebettet werden. Das CLI reagiert auf die Eingaben des Benutzers und steuert den Programmfluss. Außerdem sollen Computer-Spieler angeboten werden, die über irgendeine Form künstlicher Intelligenz verfügen. Wir können daher die drei Teilprobleme

- die Benutzeroberfläche
- das eigentliche Spiel Tic Tac Toe
- die künstliche Intelligenz der Computer-Spieler

identifizieren. Im nächsten Schritt ermitteln wir genauer, welche Anforderungen an die einzelnen Teilprobleme gestellt wurden. Der Vorteil dieses Ansatzes ist, dass wir die Teilprobleme jetzt sogar unter mehreren Entwicklern aufteilen können, die parallel weiter arbeiten.

B.4.2. Präzisierung der Teilprobleme

Der Einfachheit halber filtern wir an dieser Stelle einfach die relevanten Sätze aus der obigen Problembeschreibung heraus und erstellen daraus eine Beschreibung der einzelnen Teilprobleme. Schlüsselworte sind *kursiv* hervorgehoben.

Die Benutzeroberfläche

- Das Spiel soll als *Konsolenanwendung* realisiert werden.
- Beim Start des Spiels soll der Benutzer mit einer kurzen *Willkommensnachricht* begrüßt werden, in der erklärt wird, mit welchen Eingaben er eine neue Partie starten oder das Spiel wieder beenden kann. Zusätzlich soll die Möglichkeit bestehen, die Spielregeln abzurufen.
- Bei Fehleingaben innerhalb des *Willkommen-Menüs* soll der Benutzer darauf hingewiesen werden und weiterhin im Willkommen-Menü verbleiben.
- Beim Start einer neuen *Partie* soll zunächst nach den teilnehmenden *Spielern* und deren *Namen* gefragt werden. Es kann sich wahlweise um menschliche oder Computer-Spieler handeln. Folgende Kombinationen sind möglich: Mensch-Mensch, Mensch-Computer, Computer-Mensch, Computer-Computer.
- Sollte ein Spieler *gewonnen* haben, so ist die Partie beendet und eine Meldung erscheinen, welcher Spieler gesiegt hat. Im Anschluss soll der Benutzer entweder eine weitere Partie mit den gleichen Spielern starten, zurück zum Willkommen-Menü gehen oder das Spiel beenden können.
- Wenn das Spiel beendet wird, soll zunächst für einige Sekunden ein *Abspann* auf dem Bildschirm erscheinen.

Das eigentliche Spiel Tic Tac Toe

- Während einer Partie müssen die *Spielregeln* eingehalten werden: Zwei Spieler setzen auf einem 3×3-*Spielfeld* abwechselnd ein *Kreuz (X)* bzw. einen *Kreis (O)* in ein leeres Feld.
- Nach jedem Zug muss das Spielfeld auf die *Siegbedingung* untersucht werden. Es gewinnt derjenige, der zuerst drei seiner Symbole in einer horizontalen, vertikalen oder diagonalen Reihe hat. Es gibt drei horizontale, drei vertikale und zwei diagonale Reihen.

Die künstliche Intelligenz der Computer-Spieler

- Computer-Spieler sollten sich *möglichst intelligent* verhalten ...
- und den Eindruck erwecken, sie müssten auch *„nachdenken“*.

B.4.3. Entwickeln einer Lösungsidee für die Teilprobleme und das Gesamtproblem

In diesem Schritt überlegen wir uns langsam, wie wir die Programmabläufe modellieren können und welche Programmierkonstrukte wir dafür verwenden können. Wir überlegen uns, *was* wir später programmieren möchten, aber noch nicht zwingenderweise *wie*.

Die Benutzeroberfläche

Die Benutzeroberfläche hat die Aufgabe, die Eingaben des Benutzers entgegenzunehmen und vom Programm berechnete Ergebnisse auszugeben. Sie führt den Benutzer durch unser Tic-Tac-Toe-Programm.
Das Willkommen-Menü könnte beispielsweise so aussehen:

```
************************************************************
*   _______ _          _______           _______          *
*  |__   __(_)        |__   __|         |__   __|         *
*     | |   _  ___       | | __ _  ___     | | ___   ___  *
*     | |  | |/ __|      | |/ _` |/ __|    | |/ _ \ / _ \ *
*     | |  | | (__       | | (_| | (__     | | (_) |  __/ *
*     |_|  |_|\___|      |_|\__,_|\___|    |_|\___/ \___| *
*                                                          *
* Press N for a new game of Tic Tac Toe or Q to quit.      *
* Press R to show the rules of Tic Tac Toe.                *
*                                                          *
************************************************************
```

Auf eine ähnliche Weise könnten wir den Abspann darstellen:

```
************************************************************
*                                                          *
*                  Thanks for playing                      *
*   _______ _          _______           _______          *
*  |__   __(_)        |__   __|         |__   __|         *
*     | |   _  ___       | | __ _  ___     | | ___   ___  *
*     | |  | |/ __|      | |/ _` |/ __|    | |/ _ \ / _ \ *
*     | |  | | (__       | | (_| | (__     | | (_) |  __/ *
*     |_|  |_|\___|      |_|\__,_|\___|    |_|\___/ \___| *
*                                                          *
*              We hope to see you again soon.              *
*                                                          *
************************************************************
```

Durch die Darstellung des Willkommen-Menüs haben wir uns schon indirekt überlegt, welche Funktionstasten im Programm eine Rolle spielen sollen. `N`, `Q` und `R` haben in diesem Menü eine besondere Bedeutung und lösen bestimmte Aktionen aus. Jeder andere Tastendruck an dieser Stelle stellt eine Fehleingabe dar, die wir mit `Wrong key. Please press N, Q or R.` quittieren. Auf ähnliche Weise können wir uns die anderen Bildschirmausgaben anderen Situationen überlegen.

Das eigentliche Spiel Tic Tac Toe

Wir benötigen ein Spielfeld mit neun Feldern, die entweder leer oder von Spieler 1 bzw. Spieler 2 belegt sind. Das Ende der zweiten Runde könnte auf dem Bildschirm etwa so aussehen

```
_X_
OO_
_X_
```

und in Python durch die folgende Liste realisiert werden, wobei die ersten drei Einträge für die erste Zeile stehen, die nächsten drei für die zweite und die letzten drei für die dritte Zeile des Spielfeldes:
`field = ["_", "X", "_", "O", "O", "_", "_", "X", "_"]`.
Solange das Spiel noch nicht vorbei ist, sind die Spieler abwechselnd am Zug. Hierfür müssen wir uns im Programm merken, wer gerade dran ist. Der aktuelle Spieler kann dann einen Zug machen, d. h. ein Feld auswählen, wo er sein Symbol hinsetzen möchte. Das Programm muss gewährleisten, dass dieses Feld noch nicht besetzt ist. Wählt der Spieler ein besetztes Feld, wird der Zug abgelehnt und er wird erneut um einen gültigen Zug gebeten. Hierbei ist zu unterscheiden, ob ein menschlicher oder ein Computer-Spieler zieht. Der menschliche Spieler gibt seinen Zug über eine Tastatur-Eingabe bekannt, der Computer-Spieler kann sich im Rahmen seiner künstlichen Intelligenz selbst entscheiden. Außerdem benötigen wir im Programm eine Funktion, die nach jedem Zug das Spielfeld, d. h. die Liste `field` darauf überprüft, ob drei X bzw. drei O in einer Reihe stehen.

Die künstliche Intelligenz der Computer-Spieler

Die Anforderungen sind sehr vage formuliert. Wir wissen nur, dass der Computer-Spieler „möglichst intelligent“ sein soll und „nachdenken“ muss. Um uns zu überlegen, wie der Computer-Spieler agiert, betrachten wir das menschliche Spielerverhalten. Wenn wir am Zug sind, schauen wir auf das Spielfeld und sehen die möglichen freien Felder. Für eines dieser Felder müssen wir uns entscheiden. Die einfachste, aber nicht unbedingt klügste Variante ist, wenn wir unser Symbol einfach auf das erste freie Feld setzen, welches wir sehen. In einer ersten Implementierung könnte sich der Computer-Spieler genauso verhalten. Das ist zwar nicht „möglichst intelligent“, sondern „ziemlich doof“, aber es erfüllt seinen Zweck: Der Computer kann spielen. Später können wir das immernoch verbessern. Um das Nachdenken zu simulieren, können wir die Computer-Spieler-Funktion künstlich verlangsamen, etwa indem wir ein Pause-Intervall von ein paar Sekunden einfügen und auf dem Bildschirm eine Meldung wie `<Player (CPU)> is thinking` ausgeben.

B.4.4. Entwurf eines groben Programmgerüsts

Im vorherigen Schritt haben wir versucht, die drei Teilprobleme um detailliertere Beschreibungen zu ergänzen. Nun müssen wir uns langsam der konkreten Implementierung in der Programmiersprache Python nähern. Bevor wir jedoch den Texteditor auspacken und anfangen, den Code zusammenzuhacken, malen wir uns den Programmablauf auf einem Blatt Papier auf. Welche Verarbeitungsschritte gibt es und wie hängen sie zusammen? Dafür können wir uns an *Programmablaufplänen (PAP)*[4] orientieren. Unseren vorherigen Überlegungen zufolge könnten wir den in Abb. B.1 dargestellten Programmablauf erhalten.

Natürlich müssen wir die einzelnen Aktionen noch genauer beschreiben. Beispielsweise können wir die Aktion „Tic Tac Toe spielen“ ebenfalls in ähnlicher Weise darstellen.

[4] `http://de.wikipedia.org/wiki/Programmablaufplan`

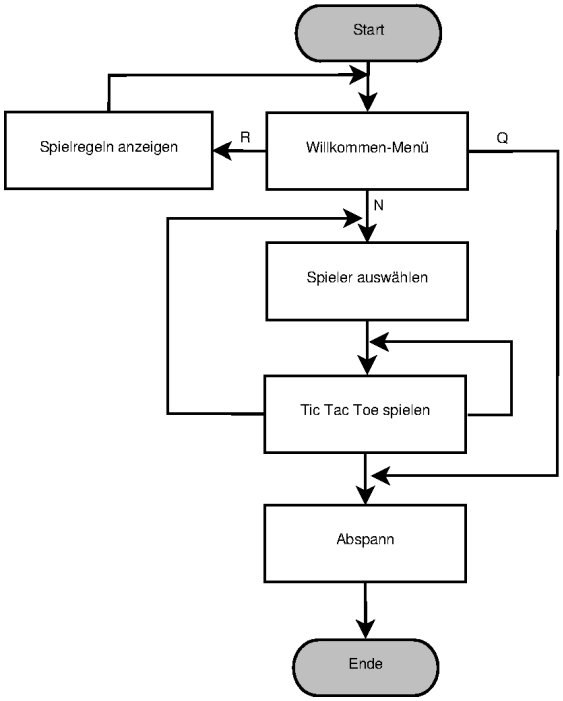


Abbildung B.1.: Programmablauf von Tic Tac Toe

B.4.5. Schrittweise Verfeinerung

Nun kommt es zum Feinschliff unserer Ideen. Wie nennen wir die Funktionen? Welche Parameter gibt es? Welche Funktionen gehören thematisch zusammen? Welche Variablen benötigen wir? Wie prüfen wir, ob das Spiel zu Ende ist, wenn wir das Spielfeld als Liste in Python gegeben haben? Hier bewegen wir uns gedanklich schon auf der Computer-Ebene. Wir überlegen uns, wie der Algorithmus aussieht und was das für den Programmcode bedeutet.

B.4.6. Umsetzung in eine Programmiersprache

Jetzt können wir endlich anfangen zu programmieren. Wir packen unsere Entwicklungsumgebung aus und legen los. Dabei halten wir uns selbstverständlich an die Programmierkonventionen und kommentieren unseren Code, damit wir ihn auch noch eine Weile später noch verstehen können. Wenn wir in den vorherigen Schritten gut gearbeitet haben und schon über ein wenig Programmiererfahrung verfügen, müssen wir in dieser Phase nicht mehr mit dem Problem Tic Tac Toe kämpfen. Unser jetziger Gegner heißt Python. Wahrscheinlich werden uns häufig Fehlermeldungen begegnen, während wir programmieren. Davon sollten wir uns nicht einschüchtern lassen. Es ist sinnvoll, die Fehlermeldungen aufmerksam zu lesen. Was steht da überhaupt? Wenn wir damit nicht weiter kommen, weiß vielleicht ein anderer Programmierer Rat, der die Fehlermeldung schon kennt. Wenn kein potenzieller Antwortgeber in der Nähe sitzt, dann ist eine Internet-Recherche ratsam. Irgendwo auf der Welt wird jemand sehr wahrscheinlich über das gleiche Problem gestolpert sein, der daraufhin in irgendeinem Forum oder Q&A-Portal seine Frage gestellt hat, die dort hoffentlich beantwortet wurde und von uns gefunden werden kann.

B.4.7. Testen und Korrigieren von Fehlern

In der Regel läuft diese Phase teilweise schon parallel zur tatsächlichen Implementierung in Python. Gerade am Anfang testen wir ständig aus, ob unser Code das tut, was wir beabsichtigen. Das ist einerseits gut, da wir Fehler früh erkennen können und ein Gefühl für die Programmiersprache bekommen, kann aber auch ein Hindernis sein, wenn wir uns frühzeitig in kleinen Fehlern verrennen, die den Gesamtfortschritt behindern. Es ist legitim, einen Fehler zunächst nicht zu korrigieren, wenn er kein essentielles Problem für den Programmablauf darstellt. Um solche Fehler niedriger Priorität können wir uns später kümmern, wenn die groben Trümmer zur Seite geräumt sind.
Wichtig in dieser Phase ist, dass wir das implementierte Programm noch einmal mit den anfangs gestellten Anforderungen vergleichen. Tut es genau das, was in der Anforderung spezifiziert wurde? Dies herauszufinden ist die Aufgabe des abschließenden Software-Tests. Leider hat die Sache einen Haken. Wir können durch die Tests nicht beweisen, dass das Programm fehlerfrei ist. Wir können durch geeignete Testfälle nur die Anwesenheit von Fehlern beweisen.

Literaturverzeichnis

[Bre07] Breymann, Ulrich: *C++ : Einführung und professionelle Programmierung*. 9., neu bearb. Aufl. München [u.a.] : Hanser, 2007

[Bre09] Breymann, Ulrich: *Der C++-Programmierer : C++ lernen - professionell anwenden - Lösungen nutzen*. München : Hanser, 2009

[Dav10] Davis, Stephen R.: *Beginning programming with C++ for dummies*. http://proquest.safaribooksonline.com/9780470617977. Version: 2010

[DD09] Deitel, Paul J. ; Deitel, Harvey M.: *C++ for programmers*. http://proquest.safaribooksonline.com/9780137018499. Version: 2009 (Deitel developer series)

[Die00] Dieterich, Ernst-Wolfgang: *C++*. 3., überarb. Aufl. München [u.a.] : Oldenbourg, 2000 http://www.oldenbourg-link.de/isbn/9783486250480

[KR78] Kernighan, Brian W. ; Ritchie, Dennis M.: *The C programming language*. Englewood Cliffs, NJ : Prentice Hall, 1978 (Prentice-Hall software series)

[KR90] Kernighan, Brian W. ; Ritchie, Dennis M. ; Schreiner, Axel T. (Hrsg.): *Programmieren in C: Mit dem C-Reference-Manual in deutscher Sprache*. 2. Ausg., ANSI C. München [u.a.] : Hanser [u.a.], 1990 (PC professionell)

[LLM06] Lippman, Stanley B. ; Lajoie, Josée ; Moo, Barbara E.: *C++ PPimer*. 4. Aufl. Bonn [u.a.] : Addison-Wesley, 2006

[LLM13] Lippman, Stanley B. ; Lajoie, Josée ; Moo, Barbara E.: *C++ primer*. http://proquest.tech.safaribooksonline.de/9780133053043. Version: 5th ed, 2013

[Mey95] Meyers, Scott: *Effective C++ : 50 specific ways to improve your programs and designs*. 9. print. Reading, Mass. [u.a.] : Addison-Wesley, 1995 (Addison-Wesley professional computing series)

[Mey96] Meyers, Scott: *More effective C++: 35 new ways to improve your programs and designs*. 3. printing. Reading, Mass. [u.a.] : Addison-Wesley, 1996 (Addison-Wesley professional computing series)

[Mey05] Meyers, Scott: *Effective C++: 55 specific ways to improve your programs and designs*. 3. ed. Upper Saddle River, NJ [u.a.] : Addison-Wesley, 2005 (Addison-Wesley professional computing series)

[MLL07] Moo, Barbara E. ; Lajoie, Josée ; Lippman, Stanley B.: *C++ primer*. http://proquest.safaribooksonline.com/9783827326126. Version: 4th Aufl, 2007 (Safari Books Online)

[Str] Stroustrup, Bjarne: *The C++ programming language*. Reprinted with corr. Reading, Mass., http://proquest.tech.safaribooksonline.de/9780133522884

[Str10] Stroustrup, Bjarne: *Einführung in die Programmierung mit C++*. München [u.a.] : Pearson Studium, 2010 (IT Informatik)

[Str13] Stroustrup, Bjarne: *The C++ programming language*. http://proquest.tech.safaribooksonline.de/9780133522884. Version: 4th ed, 2013

[Wil08] Willms, Andre: *C++-Programmierung lernen : Anfangen, Anwenden, Verstehen*. 1. Aufl. München [u.a.] : Addison-Wesley, 2008 (Programmer's choice : Klassiker). http://proquest.safaribooksonline.com/9783827326744