Skript

Vorkurs Informatik

Sommersemester 2015

Conrad Rau
PD Dr. David Sabel
Jens Keppeler
Ronja Düffel
Mario Holldack
Stand: 11. März 2015

Inhaltsverzeichnis

1. Einführung in die Bedienung von Unix-Systemen

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

1.1. Unix und Linux

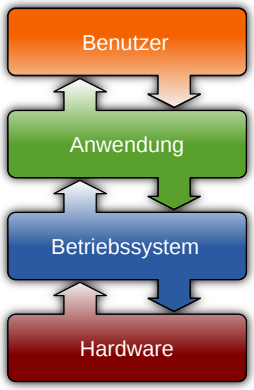


Abbildung 1.1.:

Unix

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. 1.1). 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.

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.

Terminal

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.

textbasiertes Terminal
CLI

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.

graphisches Terminal

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.

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.

1.1.1. Dateien und Verzeichnisse

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

Dateibaum
Verzeichnis

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

übergeordnetes Verzeichnis
Wurzel-verzeichnis

Beispiel 1.1 (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 1.2 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.

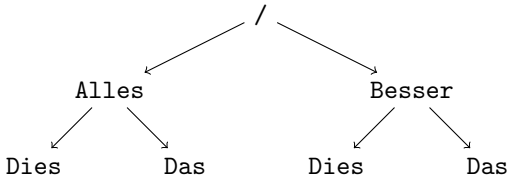


Abbildung 1.2.: Ein Dateibaum

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

Pfad
absolut
relativ
slash

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

Dateizugriffsrechte

Unter UNIX können auch die Zugriffsrechte einzelner Benutzer auf bestimmte Dateien verwaltet werden. Dabei wird unterschieden zwischen Lese-(*r*ead), Schreib-(*w*rite) und Ausführrechten (*x* 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.

1.1.2. Login und Shell

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

Einloggen
Shell
Eingabe-aufforderung

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

1.1.3. 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 aufzufordern den eingetippten Befehl auszuführen, muss die *Return*-Taste (⏎) betätigt werden. Im Folgenden sind Bildschirmein- und -ausgaben in `Schreibmaschinenschrift` gegeben und

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

Später werden wir auch sogenannte Quelltexte darstellen, diese sind

in gestrichelten und eckigen Kästen zu finden.

Passwort ändern

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

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

Für den RBI-Account wird folgender Befehl benötigt.

Netzwerk-passwort

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

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

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

Arbeits-verzeichnis
Home-verzeichnis

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

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

`whoami` : gibt den Benutzernamen aus.

```
> whoami ↩
ronja
```

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

```
> hostname ↩
nash
```

Verzeichnis erstellen
Argument

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

```
> mkdir Zeug ↩
```

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

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

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

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

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

Option

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

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

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

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

Zugriffsrechte

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

```
> chmod go +w protokoll.pdf ↵
```

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

```
> chmod o -rw protokoll.pdf ↵
```

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

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

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

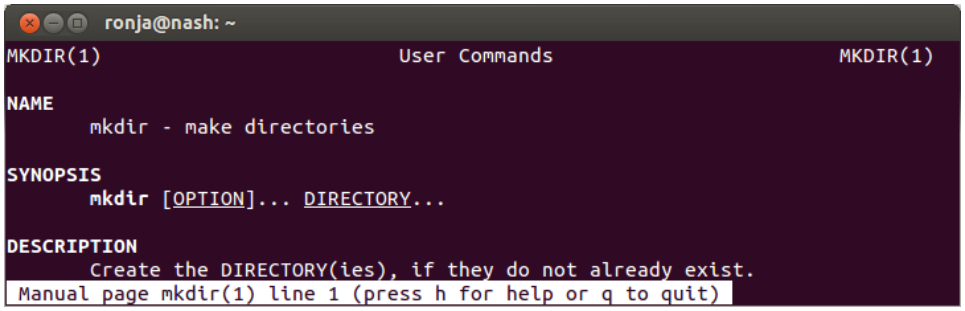


Abbildung 1.3.: man page des Befehls `mkdir`

1.1.4. 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. 1.4).

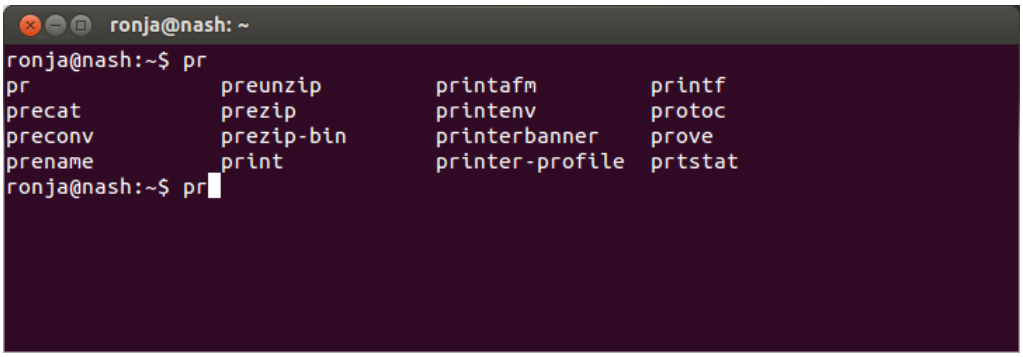


Abbildung 1.4.: Autovervollständigung für die Eingabe `pr`

1.2. Editieren und Textdateien

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

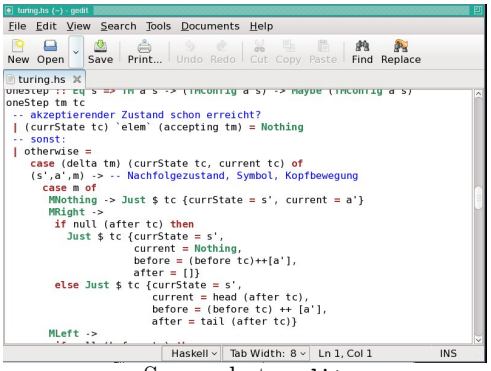
Es gibt unzählige Texteditoren. Unter KDE ist z.B. der Editor `kate` (`http://kate-editor.org/`), unter Gnome der Editor `gedit` (`http://projects.gnome.org/gedit/`) sehr empfehlenswert. Diese können allerdings nur im graphischen Modus ausgeführt werden, dafür ist ihre Bedienung dank vorhandener Mausbedienung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

Abbildung 1.5 zeigt einige Screenshots von Editoren, wobei eine Quellcode-Datei der Programmiersprache Haskell geöffnet ist.

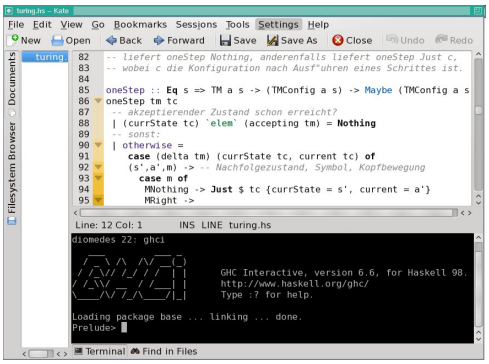
Hat man kein graphisches Interface zur Verfügung, so kann man einen Text-basierten Editor verwenden. Ein solcher weit verbreiteter Texteditor heißt `vi` (sprich: [vi: ai]). Dieser steht nicht nur in der RBI zur Verfügung, er ist auf fast jedem Unix-System vorhanden. Wir werden kurz auf seine Bedienung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

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

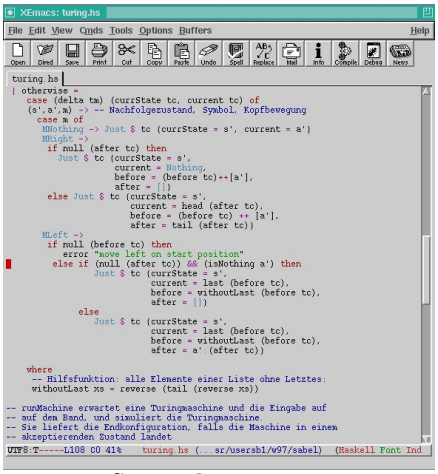
[1] Oftmals handelt es sich schon um eine verbesserte Variante `vim` (für `vi` *improved*). Diese Unterscheidung soll uns hier nicht kümmern.



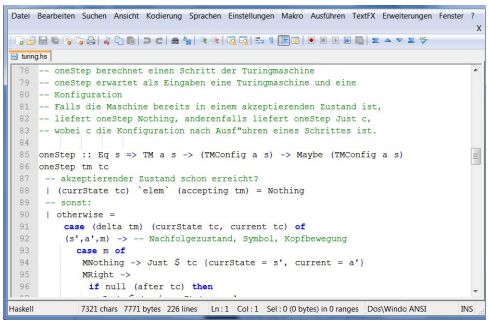
Screenshot `gedit`



Screenshot `kate`



Screenshot `xemacs`



Screenshot Notepad++ (MS Windows)

Abbildung 1.5.: Screenshots verschiedener Editoren

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

2. Im `EINFÜGEN`- und `ERSETZEN`-*Modus* erscheinen die eingegebenen Zeichen direkt auf dem Bildschirm, im Buffer. Im ersteren wird Text neu hinzugefügt, im zweiteren werden bereits vorhandene Zeichen überschrieben. In diese Modi gelangt man vom Befehlsmodus aus mit den Tasten [i] bzw. [R]. Zwischen ihnen kann mit der [Ins]-Taste hin- und hergewechselt werden. Durch Betätigen der [Esc]-Taste gelangt man wieder zurück in den Befehlsmodus.

3. Die `VISUELL`en Modi erlauben das Markieren eines Bereiches des Buffers, um sodann Befehle abzusetzen, die sich auf den markierten Bereich beziehen. Befindet man sich im Befehlsmodus, so gelangt man mit der [v]-Taste in den gewöhnlichen visuellen Modus. Dieser ermöglicht das Markieren eines Textbereiches durch Bewegen des Cursors. Nach Eingabe und Ausführung eines Befehls – etwa [x], „markierten Bereich löschen“ – findet man sich im Befehlsmodus wieder. Auch die [Esc]-Taste führt zurück in den Befehlsmodus.

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

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

2. Programmieren und Programmiersprachen

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

2.1. Programme und Programmiersprachen

Rechner

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

Maschinenprogramm

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

höhere Programmiersprache

Quellcode

Compiler

Interpreter

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

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

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

2.1.1. Imperative Programmiersprachen

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

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

prozedurale Programmiersprache

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

objektorientierte Programmiersprache

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

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

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

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

2.1.2. Deklarative Programmiersprachen

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

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

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

logische Programmiersprache

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

funktionale Programmiersprache

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

referentielle Transparenz

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

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

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

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

2.2. Haskell: Einführung in die Benutzung

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

Das aktuelle Haskell-Logo

Die wichtigste Informationsquelle zu Haskell ist die Homepage von Haskell:

`http://www.haskell.org`

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

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

2.2.1. GHCi auf den Rechnern der RBI

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

2.2.2. GHCi auf dem eigenen Rechner installieren

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

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

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

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

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

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

2.2.3. Bedienung des Interpreters

Nachdem wir den Interpreter gestartet haben (siehe Abschnitt 2.2.1) erhalten wir auf dem Bildschirm

```
/opt/rbi/bin/ghci ↵
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

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

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

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

```
Prelude> 1+2+3+4+ ↵
<interactive>:1:8: parse error (possibly incorrect indentation)
```

Hierbei sollte man die Fehlermeldung durchlesen, bevor man sich auf die Suche nach dem Fehler macht. Sie enthält zumindest die Information, an welcher Stelle des Ausdrucks der Interpreter einen Fehler vermutet: Die Zahlen `1:8` verraten, dass der Fehler in der 1. Zeile und der 8. Spalte ist. Tatsächlich ist dort das `+`-Zeichen, dem keine Zahl folgt. Wir werden später noch weitere Fehlermeldungen betrachten.

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

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

2.2.4. Quelltexte erstellen und im GHCi laden

Normalerweise erstellt man den Quelltext eines Haskell-Programms in einem Editor (siehe Kapitel 1), und speichert das Haskell-Programm in einer Datei. Die Datei-Endung muss hierbei `.hs` lauten. Abbildung 2.1 zeigt den Inhalt eines ganz einfachen Programms. Es definiert für den Namen `wert` (eigentlich ist `wert` eine Funktion, die allerdings keine Argumente erhält) die Zeichenfolge „Hallo Welt!“ (solche Zeichenfolgen bezeichnet man auch als *String*).

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

Abbildung 2.1.: Inhalt der Datei `hallowelt.hs`

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

```
> /opt/rbi/bin/ghci ↩
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load hallowelt.hs ↩
[1 of 1] Compiling Main  ( hallowelt.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

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

```
> /opt/rbi/bin/ghci ↩
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load hallowelt.hs ↩

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

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

```
*Main> wert ↩
"Hallo Welt!"
```

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

Abbildung 2.2 zeigt den Inhalt einer weiteren Quelltextdatei (namens `einfacheAusdruecke.hs`). Dort werden wieder nur Parameter-lose Funktionen definiert, aber rechts vom `=` stehen *Ausdrücke*,

```
zwei_mal_Zwei = 2 * 2

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

beides_zusammenzaehlen = zwei_mal_Zwei + oft_fuenf_addieren
```

Abbildung 2.2.: Inhalt der Datei `einfacheAusdruecke.hs`

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

Funktionsaufruf

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

```
/opt/rbi/bin/ghci ↩
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load einfacheAusdruecke.hs ↩
[1 of 1] Compiling Main  ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> zwei_mal_Zwei ↩
4
*Main> oft_fuenf_addieren ↩
55
*Main> beides_zusammenzaehlen ↩
59
*Main> 3*beides_zusammenzaehlen ↩
177
```

Funktionsnamen

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

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

grossKleinschreibungFalsch.hs:3:0:
    Not in scope: data constructor ‘Zwei_mal_Zwei’
Failed, modules loaded: none.
```

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

2.2.5. Kommentare in Quelltexten

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

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

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

wert2 = "Nochmal Hallo Welt"

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

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

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

wert = "Hallo Welt"

   gleich endet der Kommentar -}

wert2 = "Hallo Welt"
```

2.2.6. Fehler

Syntaxfehler

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

Logischer Fehler

Ein andere Klasse von Fehlern sind sogenannte *logische* oder *semantische* Fehler. Ein solcher Fehler tritt auf, wenn das Programm nicht die gewünschte Funktionalität, aber irgendeine andere Funktionalität, implementiert.

Typfehler

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

Compilezeitfehler

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

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

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

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

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

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

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

Laufzeitfehler

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

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

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

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

3. Grundlagen der Programmierung in Haskell

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

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

3.1. Ausdrücke und Typen

Ausdruck

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

Wert

Typ

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

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

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

`Char`

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

```
> /opt/rbi/bin/ghci ↩
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load einfacheAusdruecke.hs
[1 of 1] Compiling Main ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +t ↩
*Main> zwei_mal_Zwei ↩
4
it :: Integer
*Main> oft_fuenf_addieren ↩
55
it :: Integer
*Main> beides_zusammenzaehlen ↩
59
it :: Integer
```

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

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

<interactive>:1:0:
  Couldn't match expected type `Integer' against inferred type `Char'
  In the expression: 'C' :: Integer
  In the definition of `it': it = 'C' :: Integer
```

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

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

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

3.2. Basistypen

3.2.1. Wahrheitswerte: Der Datentyp `Bool`

Die Boolesche Logik (benannt nach dem englischen Mathematiker George Boole) ist eine sehr einfache Logik. Sie wird in vielen Bereichen insbesondere in wohl jeder Programmiersprache verwendet, sie ist zudem *die* Grundlage der Hardware von Rechnern. Wir führen sie hier nur sehr

oberflächlich ein. Die Boolesche Logik baut auf sogenannten atomaren Aussagen auf. Für eine solche Aussage kann nur gelten: Die Aussage ist entweder *wahr* oder die Aussage ist *falsch*. Beispiele für solche atomaren Aussagen aus dem natürlichen Sprachgebrauch sind:

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

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

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

`Bool`
Datenkonstruktor

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

Junktor

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

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

$\neg$

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

A	$\neg A$
wahr	falsch
falsch	wahr

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

$\wedge$

Die Wahrheitstabelle zum logischen Und ist daher:

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

Die Oder-Verknüpfung (mathematisch durch das Symbol $\vee$ repräsentiert) verknüpft analog zwei Aussagen durch ein „Oder“, d.h. $A_1 \vee A_2$ entspricht der Aussage „Es regnet heute *oder* ich esse heute Pommes“. Die so gebildete Aussage ist wahr, sobald einer der Operanden wahr ist (aber

$\vee$

auch dann, wenn beide wahr sind). D.h.: Wenn es heute regnet, ist die Aussage wahr, wenn ich heute Pommes esse, ist die Aussage wahr, und auch wenn beide Ereignisse eintreten.

Die Wahrheitstabelle zum logischen Oder ist:

A_1	A_2	$A_1 \vee A_2$
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

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

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

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

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

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

`Int`, `Integer`

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

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

Der ausgegebene Typ ist weder `Integer` noch `Int`, sondern `(Num t) => t`. Den Teil vor dem `=>` nennt man *Typklassenbeschränkung*. Wir erklären Typklassen hier nicht genau, aber man kann den Typ wie folgt interpretieren: `1000` hat den Typ `t`, *wenn* `t` *ein Typ der Typklasse* `Num` *ist*[1]. Sowohl `Int` als auch `Integer` sind Typen der Typklasse `Num`, d.h. der Compiler kann für die *Typvariable* `t` sowohl `Integer` als `Int` einsetzen. Lässt man `1000` auswerten, dann muss der Compiler sich für einen Typ des Ergebnisses entscheiden und führt das oben angesprochene Defaulting durch:

```
Prelude> :set +t
Prelude> 1000
1000
it :: Integer
```

Die vordefinierten Operatoren für ganze Zahlen erläutern wir später.

3.2.3. Gleitkommazahlen: `Float` und `Double`

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

`Float`, `Double`

```
Prelude> :type 10.5 ↩
10.5 :: (Fractional t) => t
Prelude> 10.5 ↩
10.5
it :: Double
```

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

```
Prelude> :set +t ↩
Prelude> (1.0000001)::Float ↩
1.0000001
it :: Float
Prelude> (1.00000001)::Float ↩
1.0
it :: Float
Prelude> (1.0000000000000001) ↩
1.0000000000000001
it :: Double
Prelude> (1.00000000000000001) ↩
1.0
it :: Double
```

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

3.2.4. Zeichen und Zeichenketten

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

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

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

`String`

3.3. Funktionen und Funktionstypen

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

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

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

```
Prelude> 2 * -2 ↩

<interactive>:1:0:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix ‘-’ [infixl 6]
        in the same infix expression
Prelude>
```

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

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

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

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

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

Wir testen einige Beispiele:

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

Einige Beispiele für die Vergleiche sind:

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

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

partielle Anwendung

`mod`, `div`

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

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

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

Funktionstyp, `->`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Prelude> not 'C' ↩

<interactive>:1:4:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the first argument of 'not', namely 'C'
  In the expression: not 'C'
  In the definition of 'it': it = not 'C'
```

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

```
Prelude> not 5 ↩

<interactive>:1:4:
    No instance for (Num Bool)
      arising from the literal '5' at <interactive>:1:4
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of 'not', namely '5'
    In the expression: not 5
    In the definition of 'it': it = not 5
```

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

3.4. Einfache Funktionen definieren

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

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

Funktionsdefinition

Die Syntax für Funktionsdefinitionen in Haskell sieht folgendermaßen aus:

$$\textit{funktion\_Name} \quad \textit{par}_1 \quad \ldots \quad \textit{par}_n \quad \texttt{=} \quad \textit{Haskell\_Ausdruck}$$

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

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

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

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

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

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

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

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

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

- `if 4+5 > 7 then 100 else 1000` ist gleich zu `100`, da 4+5 = 9 und 9 größer als 7 ist.

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

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

- `if False then "Hallo" else "Hallihallo"` ist gleich zum String `"Hallihallo"`, da die Bedingung falsch ist.
- `if (if 2 == 1 then False else True) then 0 else 1"` ist gleich zu 0, da der innere `if-then-else`-Ausdruck gleich zu `True` ist.

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

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

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

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

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

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

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

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

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

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

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

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

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

programme/einfacheFunktionen.hs:9:0:
    parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

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

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

Wir testen die Funktion im GHCi:

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

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

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

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

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

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

Funktionen als Parameter

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

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

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

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

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

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

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

$$\texttt{wende\_an\_und\_addiere} :: \underbrace{\texttt{(Integer -> Integer)}}_{\text{Typ von f}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von x}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von y}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ des Ergebnisses}}$$

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

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

nicht weglassen darf, denn der Typ

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

ist implizit rechtsgeklammert, und entspricht daher dem Typ

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

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

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

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

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

```
    Conflicting definitions for ‘x’
    In the definition of ‘sonicht’
Failed, modules loaded: none.
```

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

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

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

Wir betrachten als weiteres Beispiel die Funktion `zweimal_anwenden`:

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

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

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

<interactive>:1:18:
    Couldn't match expected type `Bool' against inferred type `String'
    In the first argument of `zweimal_anwenden', namely
        `(vergleiche True)'
    In the expression: zweimal_anwenden (vergleiche True)
    In the definition of `it': it = zweimal_anwenden (vergleiche True)
```

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

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

Polymorpher Typ

3.5. Rekursion

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

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

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

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

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

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

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

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

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

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

```
    endlos_eins_addieren 1
→   endlos_eins_addieren (1+1)
→   endlos_eins_addieren ((1+1)+1)
→   endlos_eins_addieren (((1+1)+1)+1)
→   ...
```

Rekursionsanfang und Rekursionsschritt

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

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

Der Rekursionsanfang wird gerade durch den `then`-Zweig des `if-then-else`-Ausdrucks abgedeckt: Für alle Zahlen die kleiner oder gleich 0 sind, findet kein rekursiver Aufruf statt, sondern es wird direkt 0 als Ergebnis geliefert. Der Rekursionsschritt besteht darin `erste_rekursive_Funktion` mit `x` erniedrigt um `1` aufzurufen und zu diesem Ergebnis den Wert von `x` hinzuzählen. Man kann leicht überprüfen, dass für jede ganze Zahl x der Aufruf `erste_rekursive_Funktion` x irgendwann beim Rekursionsanfang landen muss, daher *terminiert* jede Anwendung von `erste_rekursive_Funktion` auf eine ganze Zahl.

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

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

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

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

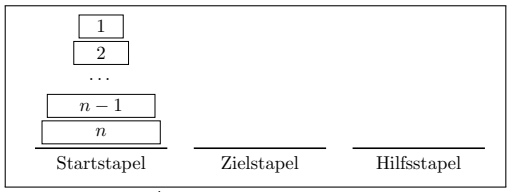
```
  erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15
```

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

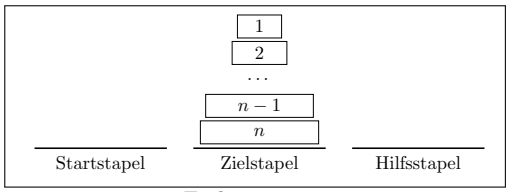
Der Trick beim Entwurf einer rekursiven Funktion besteht darin, dass man eine Problemstellung *zerlegt*: Der Rekursionsanfang ist der einfache Fall, für den man das Problem direkt lösen kann. Für den Rekursionsschritt löst man nur einen (meist ganz kleinen) Teil des Problems (im obigen Beispiel war das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

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

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



Ausgangssituation

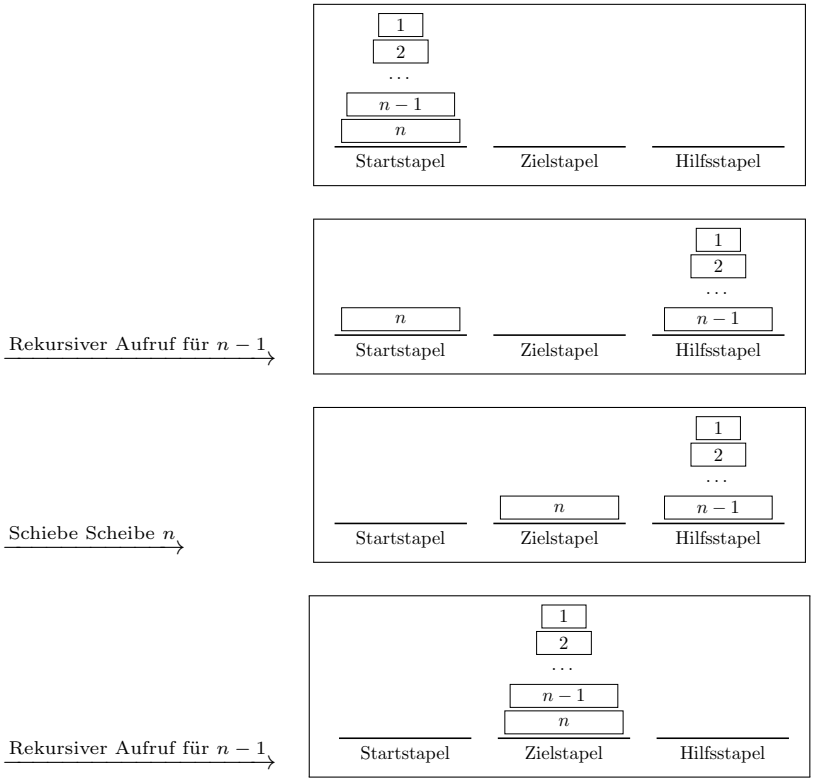


Zielsituation

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

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

Oder mit Bildern ausgedrückt:



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

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

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

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

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

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

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

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

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

Ein Testaufruf im GHCi ist:

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

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

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

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

```
otherwise = True
```

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

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

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

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

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

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

Mit diesen Überlegungen können wir die Funktion implementieren:

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

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

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

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

`let`-Ausdruck

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

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

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

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

3.6. Listen

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

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

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

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

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

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

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

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

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

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

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

3.6.1. Listen konstruieren

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

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

Nil

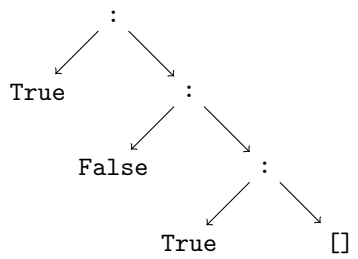
Cons

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

Tatsächlich kann man diese Liste so im GHCi eingeben:

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

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



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

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

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

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

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

3.6.2. Listen zerlegen

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

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

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

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

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

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

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

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

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

Pattern

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

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

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

Eine mögliche Definition für die Funktion `null` ist:

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

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

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

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

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

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

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

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

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

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

3.6.3. Einige vordefinierte Listenfunktionen

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

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

3.6.4. Nochmal Strings

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

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

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

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

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

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

3.7. Paare und Tupel

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

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

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

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

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

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

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

```
eigenesFst (x,y) = x

eigenesSnd (x,y) = y
```

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

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

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

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

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

3.7.1. Die Türme von Hanoi in Haskell

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

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

Die Hanoi-Funktion ist nun wie folgt implementiert:

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

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

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

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

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

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

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

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

4. Einführung in die mathematischen Beweistechniken

Die Mathematik befasst sich mit *Definitionen, Sätzen, Lemmata, ...*

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

  Definition 4.1.
  Seien a und b natürliche Zahlen. $a+b$ beschreibt die Addition der natürlichen Zahlen a und b.

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

  Lemma 4.2 (Binomische Formel)**.**
  $(a+b)^2 = a^2 + 2ab + b^2$

  Ein erster Beweis

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

In der Informatik kämpfen wir ebenfalls mit mathematischen Strukturen, 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. In diesen Bereichen gab es in der Vergangenheit, aufgrund von Programmierfehlern bzw. Fehlern in Algorithmen, mehrere Unfälle. 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 Wir betrachten zunächst, wie wir Aussagen in der Mathematik behandeln können

4.1. Mathematische Aussagen

Definition 4.3.
Eine mathematische Aussage ist eine Aussage, die entweder wahr (**1**) oder falsch (**0**) sein kann. mathematische Aussagen

Beispiel 4.4.
Beispiele für mathematische Aussagen sind:

- Das Auto ist rot
- Eine Zahl ist durch 3 teilbar
- Im Institut für Informatik gibt es ein Lernzentrum
- Wenn es regnet, ist die Strasse nass
- Die Dauer des Vorsemesterkurses beträgt 6 Tage

Notation 4.5.
Wir stellen hier mathematische Aussagen oft durch die Buchstaben $A, B, C, ...$ dar. $A, B, C, ...$

Beispiel 4.6.
Beispiel für die Anwendung der Notation 4.5:

A Das Auto ist rot

B Eine Zahl ist durch 3 teilbar

Definition 4.7 (Negation einer Aussage)**.**
$\neg A$ Sei A eine mathematische Aussage. Die Negation einer Aussage A (symbolisch: $\neg A$, sprich: „nicht A“) ist wahr, falls A falsch ist und ist falsch, falls A wahr ist.

Beispiel 4.8.
Beispiele für die Negaion von Aussagen (Def. 4.7):

A Das Auto ist rot

$\neg A$ Das Auto ist nicht rot

B Eine Zahl ist durch 3 teilbar

$\neg B$ Eine Zahl ist nicht durch 3 teilbar

C Alle Dächer sind rot

$\neg C$ Es gibt ein Dach, welches nicht rot ist

Bemerkung 4.9.
Beachte, dass bei Aussagen über Mengen (vgl. das letzte Beispiel von Beispiel 4.8) gilt:

A Ich bin eine Aussage, die etwas über alle Objekte in einer Menge aussagt

$\neg A$ Ich bin eine Aussage, die sagt, dass die Aussage A für ein Objekt der Menge nicht gilt

Definition 4.10.
$(A \wedge B)$ Seien A und B Aussagen. Die Verundung zweier Aussagen (symbolisch: $(A \wedge B)$, sprich: “A und B”) ist die Aussage, die wahr ist, falls A und B wahr sind.

Beispiel 4.11.
Beispiel für die Verundung zweier Aussagen (Def. 4.10):

A Das Auto ist rot

B Das Auto fährt schnell

$(A \wedge B)$ Das Auto ist rot und fährt schnell

Definition 4.12.
$A \to B$ Seien A und B Aussagen. Aus Aussage A folgt Aussage B (symbolisch: $A \to B$, sprich “Aus A folgt B” oder “Wenn A dann B”), falls gilt: Wenn A wahr ist, dann ist auch B wahr.

Beispiel 4.13.
Beispiele für mathematische Aussagen sind:

A Es regnet

B Die Strasse ist nass

$(A \to B)$ Wenn es regnet, ist die Strasse nass

Definition 4.14.
$A \leftrightarrow B$ Seien A und B Aussagen. Die Aussagen A und B sind äquivalent (symbolisch: $A \leftrightarrow B$, sprich “A genau dann, wenn B”), falls gilt: A ist genau dann wahr, wenn B wahr ist.

Bemerkung 4.15.
Statt $A \leftrightarrow B$ kann man auch $(A \to B) \wedge (B \to A)$ schreiben.

Wahrheitstabellen

In der Aussagenlogik verwendet man häufig Wahrheitstabellen um die Bedeutung von Verknüpfungen deutlich zu machen und darzustellen, wie der Wahrheitswert einer zusammengesetzten Aussage durch die Wahrheitswerte der ursprünglichen Aussagen bestimmt wird. 0 steht hierbei für "falsch", 1 steht für "wahr".

A	**B**	$\neg$**A**	**A** $\wedge$ **B**	**A** $\vee$ **B**	**A** $\rightarrow$ **B**	**A** $\leftrightarrow$ **B**
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Tabelle 4.1.: Wahrheitstabelle für "nicht" ($\neg$), "und"($\wedge$), "oder" ($\vee$), "wenn..., dann ..." ($\rightarrow$) und "genau dann, wenn..."($\leftrightarrow$).

Definition 4.16 (Implikationsfolge)**.**
Seien $A_1, A_2, ..., A_n$ Aussagen und es gelte $A_i \rightarrow A_{i+1}$ für alle $i \in \{1, ..., n-1\}$. Dann schreiben wir kurz

$$A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n$$

und nennen dies Implikationsfolge.

Implikatonsfolge
$A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n$

4.2. Direkter Beweis

Mathematische Aussagen haben häufig die Form "Wenn... dann..." ($A \rightarrow A$). Dabei nennen wir A die **A**nnahme oder Voraussetzung und B die **B**ehauptung oder Folgerung.

Beispiel 4.17.
Betrachte die Aussage: Wenn eine Zahl durch 10 teilbar ist, dann ist sie auch durch 5 teilbar.

Dann teilen wir A und B auf in:

A : Eine Zahl ist durch 10 teilbar

B : Eine Zahl ist durch 5 teilbar

Nun müssen wir, um zu zeigen, dass die Aussage gilt, $A \rightarrow B$ nachweisen. Wir wissen: Wenn A wahr ist, dann muss unbedingt auch B wahr sein, damit $A \rightarrow B$ wahr ist. Wir müssen daher zeigen, wenn A wahr ist, dann ist auch B wahr sein.

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

Definition 4.18 (Teilbarkeit)**.**
Eine Zahl a ist durch b teilbar, wenn es eine ganze Zahl k gibt mit $a = b \cdot k$

teilbar... durch Teilbarkeit

Beispiel 4.19.
70 ist durch 10 teilbar, da $70 = 10 \cdot k$ für $k = 7$.

Zurück zu unserem Beispiel.

Beispiel 4.20.
Sei a durch 10 teilbar. Dann hat a die Form $a = 10 \cdot k$ für eine ganze Zahl k.

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

A 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)

B Eine Zahl ist durch 5 teilbar

Wir beobachten die Implikationsfolge $A \to A_1 \to A_2 \to A_3 \to B$, also $A \to B$. Nun verpacken wir unseren Beweis in einen schönen mathematischen Text:

Lemma 4.21.
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. 4.18) 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. □

4.3. Beweis durch Kontraposition

Nicht immer ist es einfach $A \to B$ 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:

A Die Karte ist vorne blau

B 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 formalisiert: $(A \to B)$. Wir betrachten nun die Negationen von A und B:

$\neg A$ Die Karte ist vorne gelb

$\neg B$ 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: $(\neg B \to \neg A)$.

Intuitiv können wir sagen: Wir können $(\neg B \to \neg A)$ anstatt $(A \to B)$ nachweisen. Diese Beweistechnik nennen wir “Beweis durch Kontraposition”.

Beispiel 4.22.
Wenn a^2 ungerade ist, dann ist a ungerade

Wir betrachten unsere Aussagen:

A a^2 ist eine ungerade Zahl

B a ist eine ungerade Zahl

und deren Negationen:

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

$\neg B$ a ist eine gerade Zahl

Wir betrachten nun wieder unsere Aussagenfolge:

$\neg B$ a ist eine ganze 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

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

Wir haben also $\neg B \to \neg A$ und somit auch $A \to B$ gezeigt. Kompakt geschrieben:

Lemma 4.23.
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 $a^2 = 2^2 \cdot k = 2 \cdot 2 \cdot k^2 = 2 \cdot k'$ 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 wichtig, die richtige Beweistechnik zu wählen, um sich viel Arbeit zu ersparen.

4.4. Beweis durch Widerspruch

Wiederum gibt es Fälle, bei denen der direkte Beweis und der Beweis durch Kontraposition nicht geeignet ist. Wir fordern 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, Jugendliche, 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? Der Supermarkt darf den Alkohol **nicht** verkaufen. Dann hat der Supermarkt seinen Test bestanden. Wir halten fest:

A Der Supermarkt verkauft Alkohol an ihre Kunden

B Die Kunden sind $\geq$ 18 Jahre alt

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

$(A \to B)$ 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 Jugendliche sind jünger als 18 Jahre. Wir erhalten: $(A \wedge \neg B)$. Wenn unter diesen Umständen der Supermarkt kein Alkohol verkauft, (Also wenn wir eine Falschaussage f haben, die unsere Behauptung widerlegt), dann erhalten wir insgesamt: $((A \wedge \neg B) \rightarrow f)$. Wir beweisen also $A \rightarrow B$, indem wir $((A \wedge \neg B) \rightarrow f)$ zeigen, also $(A \wedge \neg B)$ zu einer Falschaussage bzw. Widerspruch führen. Wir nennen diese Beweistechnik *Beweis durch Widerspruch.*

Beispiel 4.24.
Wenn a und b gerade natürliche Zahlen sind, dann ist auch deren Produkt $a \cdot b$ gerade.

Wir teilen unsere Aussage in Teilaussagen auf:

A a und b sind gerade natürliche Zahlen

B $a \cdot b$ ist gerade

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

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

$A \wedge \neg B$: 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.

Lemma 4.25.
Wenn a und b gerade natürliche Zahlen sind, dann ist auch $a \cdot b$ gerade.

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 4.18. 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, gibt $a \cdot b = 2 \cdot k'$ mit $k' = a \cdot k'$. Somit ist $a \cdot b$ gerade. **Widerspruch** □

4.5. Ausblick

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 □), 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

5. Induktion und Rekursion

Strenggenommen sollte dieses Kapitel ein Unterkapitel des vorherigen Kapitels sein, denn die *vollständige Induktion* ist eine mathematische Beweistechnik. Allerdings ist diese Technik für das Verständnis der Lehrveranstaltung Datenstrukturen so wesentlich, dass wir ihr hier ein eigenes Kapitel widmen.

5.1. 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 werden abkürzende Schreibweisen für Summen und Produkte eingeführt.

Definition 5.1.
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 5.2.

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

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

Induktionsanfang: $n = 1$
Behauptung: $A(1)$: Der Satz gilt für $n = 1$.
Beweis:

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

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

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

Beweis:

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

□

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

Beispiel 5.4.

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

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

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

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

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

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

Beweis:

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

□

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

Beispiel 5.6.

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

Beweis. durch vollständige Induktion.

Induktionsanfang: $n = 0$
Behauptung: $0^2 + 0$ ist eine gerade Zahl.
Beweis: $0^2 + 0 = 0 + 0 = 0$ ist eine gerade Zahl. Das stimmt, denn 0 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:

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

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

□

5.1.1. 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 5.8.
Sei $n \in \mathbb{N}$. Dann ist die Folge $a(n) = 1/n$ immer positiv.

Obiger Satz ist zwar wahr, aber wie soll man aus $\frac{1}{n} > 0$ folgern, dass $\frac{1}{n+1} > 0$? Das ist für den Induktionsschritt aber notwendig.

5.1.2. Was kann schief gehen?

Das Prinzip der vollständigen Induktion lässt sich auch mit einem Domino-Effekt vergleichen. Die Bahn läuft durch, d.h. alle Dominosteine fallen um, wenn 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 5.9.
Im folgenden sei Teilbarkeit wie in Definition 4.18 definiert.
Zum Beispiel lässt sich aus der Aussage $A(5\ \textit{ist durch}\ 2\ \textit{teilbar})$ logisch korrekt folgern, dass auch $B(7\ \textit{ist durch}\ 2\ \textit{teilbar})$ gilt. Die Schlussfolgerung ist logisch korrekt, die Aussagen gelten aber nicht, da eben die Voraussetzung nicht gegeben ist. Denn 5 ist nunmal nicht durch 2 teilbar.

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

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

Was ist passiert?
Das Argument „aus Erfahrung weiß man, ein Paar Socken passt immer noch rein“, ist in Bezug auf die Aufgabenstellung nicht konstruktiv. Ein konstruktives Argument hätte sagen müssen, wo genau das extra Paar Socken noch hinpasst.

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

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

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

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

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

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

5.2. Rekursion

In Abschnitt 3.5 wurde bereits ausführlich auf Rekursion eingegangen. Es ist eine Technik bei der man etwas durch sich selbst definiert. Allerdings gibt es immer eine Basisregel, auch *Rekursionsanfang*, die ein erstes Element definiert. In der Programmierung erkennt man Rekursion daran, dass sich eine Funktion direkt oder indirekt selbst aufruft. Das Prinzip der Rekursion lässt sich aber auch für die Definition von mathematischen Funktionen und Mengen verwenden.

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

Fakultätsfunktion

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

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

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

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

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

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

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

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

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

□

Die vollständige Induktion eignet sich also z.B. um die Korrektheit rekursiv definierter Funktionen zu beweisen. So kann man mit vollständiger Induktion ganz ähnlich wie im obigen Beispiel auch die Korrektheit der Aussage in Abschnitt **??**, dass die Haskell-Funktion `erste_rekursive_Funktion x` gerade $\sum_{i=1}^{x} i$ berechnet, beweisen. Dies werden wir in der Übung auch tun.

5.2.1. Türme von Hanoi

Türme von Hanoi

Ein weiterer Algorithmus, der in Abschnitt **??** behandelt wurde, ist der zur Lösung des Problems der *Türme von Hanoi*. Dabei geht es darum, dass ein Stapel von n, nach Größe sortierter Scheiben, von einem *Startstapel* mit Hilfe eines *Hilfsstapels* auf einen *Zielstapel* transportiert werden soll. Dabei darf

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

Die rekursive Lösung des Problems sieht so aus, dass man

1. zunächst rekursiv die $n-1$ Scheiben vom Startstapel auf den Hilfsstapel transportiert
2. dann die n-te Scheibe vom Startstapel auf den Zielstapel transportiert,
3. danach die $n-1$ Scheiben vom Hilfstapel rekursiv auf den Zielstapel transportiert.

Dabei bedeutet *rekursiv* transportieren, dass die Hanoi-Funktion so lange rekursiv aufgerufen wird, bis das Argument tatsächlich 1 ist.

In Pseudo-Code aufgeschrieben, sieht der Algorithmus folgendermaßen aus:

Algorithmus 5.14.

```
1 function hanoi(n, start, ziel, hilf){
2   if (n>1){
3     hanoi(n-1,start,hilf,ziel)
4   }
5   verschiebe Scheibe n von start auf ziel
6   if (n>1){
7     hanoi(n-1,hilf,ziel,start)
8   }
9  }
```

Korrektheit

Satz 5.15.
Algorithmus 5.14 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 Hilfstapel verwendet werden darf.

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

```
1 hanoi(2, A, B, C)                          //Aufruf mit n
2   hanoi(1, A, C, B)                        //Z.3, 1.Aufruf mit (n-1)
3     verschiebe Scheibe 1 von A auf C       //Aufruf mit (n-1), Z.5
4   verschiebe Scheibe 2 von A auf B         //Aufruf mit n, Z.5
5   hanoi(1, C, B, A)                        //Aufruf mit n, Z.7
6     verschiebe Scheibe 1 von C auf B       //2.Aufruf mit (n-1), Z.5
```

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

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

```
1  hanoi(n+1, A, B, C)                          //Aufruf mit n+1
2    hanoi(n, A, C, B)                          //Z.3, Aufruf mit n
3    verschiebe Scheibe n+1 von A auf B         //Z.5
4    hanoi(n, C, B, A)                          //Z.7, Aufruf mit n
```

Zuerst werden also die obersten n Scheiben mit `hanoi(n, A, C, B)` von Stapel A nach Stapel C transportiert. Laut Induktionsvoraussetzung arbeitet `hanoi(n, A, C, B)` korrekt und transportiert den Stapel mit n Scheiben von A nach C. Dann wird Scheibe $n+1$ von A auf B verschoben (3), anschließend werden die n Scheiben auf Stapel C mit `hanoi(n, C, B, A)` auf Stapel B transportiert. 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. Daher 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. `hanoi(n, C, B, A)` transportiert laut Induktionsvoraussetzung den Stapel mit n Scheiben korrekt von C nach B.

Somit 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 darauf auf die Gesetzmäßigkeit:

Satz 5.16.
Um n Scheiben von einem Stapel zu einem anderen zu transportieren, werden mindestens $2^n - 1$ Spielzüge benötigt.

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

Induktionsschritt: $n \to n+1$
Induktionsvoraussetzung: Um n Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens $2^n - 1$ Spielzüge benötigt.
Induktionsbehauptung: Um $n+1$ Scheiben von einem Stapel auf einen anderen zu transportieren, werden mindestens $2^{n+1} - 1$ Spielzüge benötigt.
Beweis: Um $n+1$ Scheiben von einem Stapel A auf einen Stapel B zu transportieren, transportiert man nach Algorithmus 5.14 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:

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

Spielzüge.

□

6. Asymptotik und ihre Anwendung in der Informatik

In der Informatik haben wir es häufig mit Algorithmen zu tun. Diese sind mathematische Rechenvorschriften. Insbesondere sind Programmcodes, die wir mittels einer Programmiersprache in den Computer eingeben (Beispielsweise mit den Programmiersprachen (Python, Haskell, C++, ...)) Algorithmen, da wir dort in den Code Rechenvorschriften hineinschreiben, die von dem Computer ausgeführt werden. Wir stellen uns nun folgende Fragen:

- Ist mein Algorithmus effizient (in dem Sinne, dass der Computer das Problem "schnell" löst)?
- Ich habe zwei Algorithmen. Welcher löst mein Problem schneller (effizienter)?
- Wird mein Algorithmus heute noch fertig?

Um auf diese Fragen Antworten zu finden, analysieren wir Programmcodes und ordnen diesen so genannte Laufzeiten zu. Diese Analyse nennen wir Laufzeitanalyse. Um zu zeigen, wie wichtig effiziente Algorithmen sind und um die Wichtigkeit dieser effizienter Algorithmen zu verdeutlichen betrachten wir folgende Tabelle:

Annahme: Ein einfacher Befehl benötigt 10^{-9}sec

n	n^2	n^3	n^{10}	2^n	$n!$
16	256	4.096	$\geq 10^{12}$	65536	$\geq 10^{13}$
32	1.024	32.768	$\geq 10^{15}$	$\geq 4 \cdot 10^9$	$\geq 10^{31}$
64	4.096	262.144	$\geq 10^{18}$	$\geq 6 \cdot 10^{19}$	
					mehr als
128	16.384	2.097.152	mehr als	mehr als	10^{14} Jahre
256	65.536	16.777.216	10 Jahre	600 Jahre	
512	262.144	134.217.728			
1024	1.048.576	$\geq 10^9$			
10^6	$\geq 10^{12}$	$\geq 10^{18}$			
	mehr als	mehr als			
	15 Minuten	10 Jahre			

Betrachten wir hier etwa einen Algorithmus mit der Laufzeit $n!$ und einen mit der Laufzeit n^2, der das gleiche berechnet, dann würden wir sicherlich den Algorithmus mit der besseren Laufzeit nehmen, da die jahrelange Ausführung keinen Spass macht.

In Abschnitt 6.1 legen wir uns zuerst die mathematischen Grundlagen fest, die wir zu Laufzeitanalyse benötigen. Danach werden wir in Abschnitt 6.2 an einigen Beispielen die Laufzeit verschiedener Algorithmen analysieren.

6.1. Asymptotik

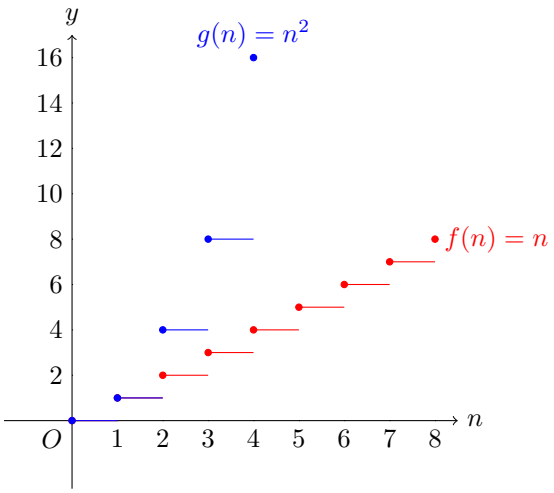
Wir geben zunächst eine Schreibweise an, die uns für zwei Funktionen f, g beschreibt, dass f für $n \to \infty$ höchstens so schnell wächst, wie g:

$f = O(g)$

Definition 6.1 (Die Groß-Oh Notation)**.**
Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ zwei Funktionen, die jeder natürlichen Zahl, eine nicht-negative reelle Zahl zuordnen. Wir schreiben $f = O(g)$ genau dann, wenn es eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt:

$$f(n) \leq c \cdot g(n)$$

Beispiel 6.2.
Wir betrachten die beiden Funktion $f(n) = n$ und Funktionen $g(n) = n^2$ mit $n \in \mathbb{N}$.



Wertetabelle

n	$f(n)$	$g(n)$
0	0	0
1	1	1
2	2	4
3	3	9
4	4	16

Man sieht leicht, dass für $c = 1$ und $n_0 = 0$ gilt: Für alle $n \geq 0$ gilt:

$$f(n) \leq 1 \cdot g(n)$$

Nach Definition gilt also, dass $n = O(n^2)$.

Eine weitere Betrachtung auf die O-Notation liefert uns die folgende Beziehung:

Lemma zu $f = O(g)$

Lemma 6.3.
Sei

$$c := \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Falls der Grenzwert $\frac{f(n)}{g(n)}$ *existiert, und* $0 \leq c < \infty$*, dann ist* $f = O(g)$

Beispiel 6.4.
Wir greifen noch einmal unser Beispiel 6.2 auf und weisen die Beziehung $n = O(n^2)$ mit Lemma 6.3 nach.

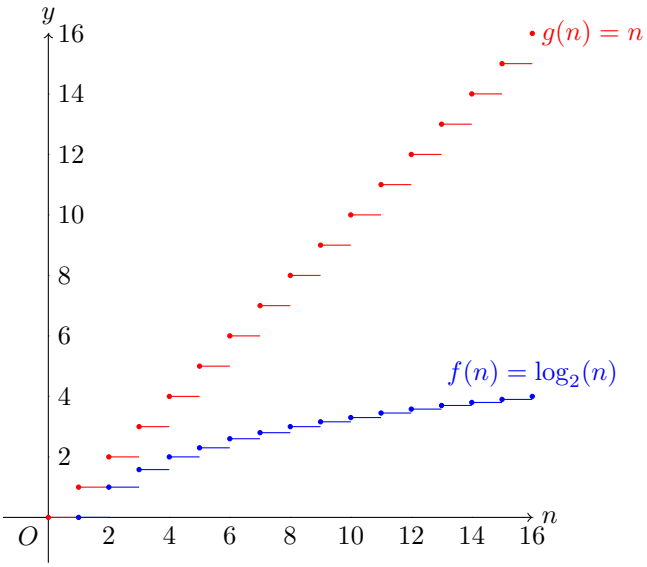
$$c := \lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0$$

Da der Grenzwert $\frac{n}{n^2}$ existiert und da $c = 0$ gilt: $f = O(g)$.

Wir betrachten drei weitere Beispiele:

Beispiel 6.5.
Sei

$$f(n) := \log_2(n)$$
$$g(n) := n$$



Wertetabelle

n	$f(n)$	$g(n)$
1	0	1
2	1	2
4	2	4
8	3	8
16	4	16

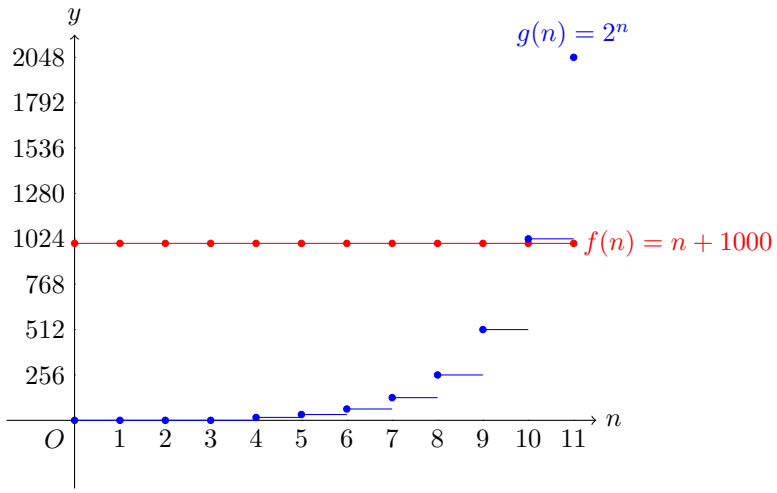
Wir beobachten: für $c = 1$ und $n_0 = 1$ ist unsere Definition erfüllt. D.h. für alle $n \geq 1$ gilt

$$f(n) \leq 1 \cdot g(n)$$

Damit gilt: $\log_2(n) = O(n)$.

Beispiel 6.6.
Sei

$$f(n) := n + 1000$$
$$g(n) := 2^n$$



n	$f(n)$	$g(n)$
0	1000	1
1	1001	2
2	1002	4
3	1003	8
4	1004	16
5	1005	32
6	1006	64
7	1007	128
8	1008	256
9	1009	512
10	1010	1024
11	1011	2048
12	1012	4096

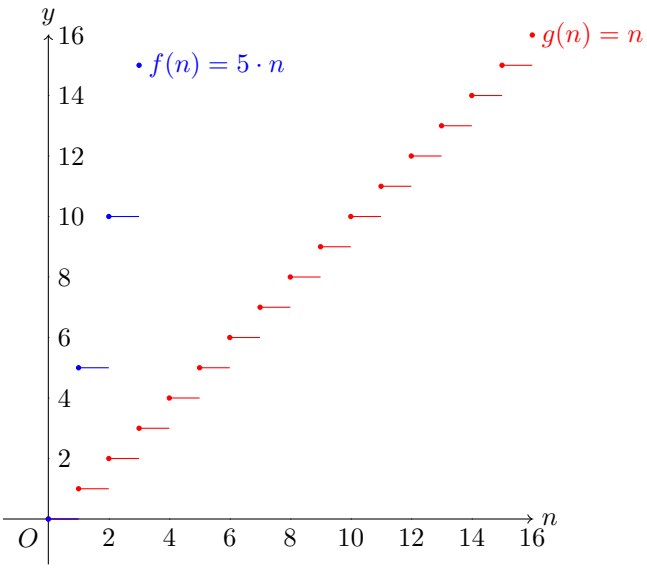
Wir beobachten: für $c = 1$ und $n_0 = 10$ ist unsere Definition erfüllt. D.h. für alle $n \geq 10$ gilt:

$$f(n) \leq 1 \cdot g(n)$$

Damit gilt: $ln(n) = O(n)$.

Beispiel 6.7.

$$f(n) := 5 \cdot n$$
$$g(n) := n$$



n	$f(n)$	$g(n)$
1	5	1
2	10	2
3	15	3
4	20	4
5	25	5

Wir betrachten hier erstmal die Sichtweise nach Lemma 6.3:

$$c := \lim_{n\to\infty} \frac{5 \cdot n}{n} = \lim_{n\to\infty} \frac{5}{1} = \lim_{n\to\infty} 5 = 5$$

Der Grenzwert von $\frac{5 \cdot n}{n}$ existiert mit $c = 5$.

Wir können diese Beziehung auch mit der Definition nachweisen: Für $c = 5$ und $n_0 = 0$ gilt:

$$f(n) \leq c \cdot g(n) \Leftrightarrow 5 \cdot n \leq 5 \cdot n$$

Aus unseren Beispielen können wir folgende Beobachtungen folgern:

Beobachtungen 6.8.
Seien $f, g : \mathbb{N} \to \mathbb{R}_{\leq 0}$.

(a) Falls $f = n^k$ und $g = n^l$ mit $k \geq l$, dann gilt $f = O(g)$.

(b) Sei $c \in \mathbb{R}_{\geq 0}$ eine positive reelle Zahl, dann gilt $c \cdot f(n) = O(f(n))$.

Beweis. Übung □

Eine Wachstums-Hierarchie

Wir betrachten hier eine Wachstums-Hierarchie wichtiger Laufzeitfunktionen.

Wachstums-Hierarchie

Lemma 6.9.
Es sei $a > 1$, $k > 1$ und $b > 1$. Sei:

$$f_1(n) := \log_a n \quad f_2(n) := n \quad f_3(n) := n \cdot \log_a n \quad f_4(n) := n^k \quad f_5(n) := b^n \quad f_6(n) := n!$$

Dann gilt $f_i = O(f_j)$ für $i, j = 1, ..., 5$ und $i < j$.

Beweis. Übung □

Beispiel 6.10.
Für die Funktionen

$$f_1(n) := \log_2 n \quad f_2(n) := n \quad f_3(n) := n \cdot \log_2 n \quad f_4(n) := n^2 \quad f_5(n) := 2^n$$

gilt $f_i = O(f_j)$ für $i, j = 1, ..., 5$ und $i < j$.

6.2. Laufzeitanalyse

In diesem Abschnitt werden wir verschiedene Programme/Algorithmen betrachten und die Laufzeit dieser Algorithmen bestimmen.

Algorithmus zum Zählen

Algorithmus 6.11.

```
1 function zaehle(n){
2   for(int i = 1; i <= n; i++){
3     print i;
4   }
5 }
```

Bedeutung des Algorithmus:
In Zeile 3 haben geben wir auf den Wert der Variable i aus. Diese wird von einer For-schleife umfasst. Diese For-Schleife definiert den Wert i bei $i = 1$ und erhöht sich nach jedem Schritt um einen Wert bis i den Wert n erreicht hat. D.h. wir beginnen bei $i = 1$ und geben den Wert 1 aus, dann erhöhen wir i um 1 und geben den Wert 2 aus u.s.w. bis wir bei $i = n$ angelangt sind.

Wir berechnen nun die Laufzeit des Algorithmus 6.11 Dazu zählen wir, wie oft "print i" ausgeführt wird. Wir definieren nun eine kleine Hilfsfunktion, die uns beim Zählen hilft:

Definition 6.12 (Die Zählfunktion $s_{z,z'}$)**.**
$s_{z,z'}$

$s_{z,z'}$ gebe die Anzahl der ausgeführten Befehle in dem Algorithmus an, die sich zwischen den Zeilen z und z' befinden.

Wir gehen nun wie folgt vor:

- Wir zählen die Anzahl der Befehle
- Wir überführen dies in die O-Notation.

Laufzeitanalyse zum Algorithmus 6.11

Laufzeitanalyse:
Wir betrachten zunächst nur die Zeile 3, in der "print i;" steht. Wir haben hier nur einen Befehl. Somit gilt offensichtlich:

$$s_{3,3} = 1$$

Wir betrachten nun die for-Schleife, die sich um Zeile 3 befindet. Wir haben dort eine Zählvariable i, die beginnend von $i = 1$ bis n hoch zählt. Damit gilt für die Laufzeitbetrachtung von Zeile 2 bis Zeile 4:

$$s_{2,4} = \sum_{i=1}^{n} s_{3,3} = \sum_{i=1}^{n} 1$$

Wir sind nun fertig, da in Zeile 1 und 5 nichts passiert. Wir überführen nun die Zahl in die O-Notation:

$$s_{2,4} = \sum_{i=1}^{n} s_{3,3} = \sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \cdots + 1}_{n-\text{mal}} = n = O(n)$$

Algorithmus zum kleinen 1×1

Algorithmus 6.13.

```
1 function zaehle_2(n){
2   i = 1;
3   while(i <= n){
4     for (int j = 1; j <= n; j++){
5         print i * j;
6     }
7     i = i + 1;
8   }
9 }
```

Bedeutung des Algorithmus:
Wir sehen hier in Zeile 5, dass wir einen Wert $i \cdot j$ ausgeben. Dieser Befehl wird von einer For-Schleife umfasst, die bei $j = 1$ beginnt, j nach einem Schritt um Eins erhöht wird und die for-Schleifen endet, wenn der Wert $j = n$ erreicht wird. In Zeile 2 wird $i = 1$ initialisiert. Dann wird die Befehlsfolge innerhalb der while-Schleife ausgeführt, solange i einen kleineren Wert als n hat. Innerhalb der while-Schleife wird der Wert von i um Eins erhöht (Zeile 7). Innerhalb der while Schleife befindet sich auch die For-schleife, in der $i * j$ ausgegeben wird. Zusammengefasst gibt das Programm folgendes aus:

$$1 \cdot 1 \quad 1 \cdot 2 \quad 1 \cdot 3 \quad \cdots \quad 1 \cdot n \quad 1 \cdot 2 \quad \cdots 2 \cdot n \quad 1 \cdot 3 \quad \cdots n \cdot n$$

Wir listen also, das kleine 1×1 bis zu einem Wert n auf.

Laufzeitanalyse:
Wir betrachten zunächst Zeile 5. Klar: Wir rufen einen Befehl auf und es gilt:

Laufzeitanalyse zu Algorithmus 6.13

$$s_{5,5} = 1$$

Zeile 4 bis 6 ist eine for-Schleife, die bei $j = 1$ beginnt und bei $j = n$ aufhört.

$$s_{4,6} = \sum_{j=i}^{n} s_{5,5} = \sum_{j=1}^{n} 1 = \underbrace{1 + 1 + \cdots + 1}_{n-\text{mal}} = n$$

In Zeile 7 haben wir wieder ein Befehl. D.h. es gilt:

$$s_{4,7} = s_{4,6} + 1 = n + 1$$

Um die while-Schleife zu analysieren, überlegen wir nun, wie oft diese ausgeführt werden muss. Wir sehen, dass die Bedingung der while-Schleife mit i arbeitet. Wir fassen mal zusammen, wie mit i gearbeitet wird:

- i erhält zu Beginn den Wert 2 (Zeile 2)
- i wird **innerhalb der while** Schleife um Eins erhöht
- Die while-Schleife endet, wenn $i > n$

Daraus folgt für $s_{3,8}$:

$$s_{3,8} = \sum_{i=2}^{n} s_{4,7} = \sum_{i=2}^{n} n = \underbrace{n + n + \cdots + n}_{(n-1)-\text{mal}} = (n-1) \cdot n$$

In Zeile 2 wird noch ein Befehl ausgeführt. Also gilt:

$$s_{2,8} = 1 + s_{3,8} = 1 + (n-1) \cdot n$$

In den Zeilen 1 und 9 passiert nichts. Damit sind wir fertig. Zusammenfassend gilt für die Laufzeit:

$$s_{1,9} = 1 + (n-1) \cdot n = 1 + n^2 - n = O(n^2 - n) = O(n^2)$$

Algorithmus binäre Suche

Algorithmus 6.14 (binäre Suche)**.**
Wir nehmen als Eingabe einen Array (eine Liste von Zahlen) A. $A[i]$ bezeichnet die Zahl in dem Array A, die in dem Array an der Position i steht. Die Einträge in dem Array A sind sortiert, d.h. es gilt: $A[1] \leq A[2] \leq \cdots \leq A[n]$.

```
1  function suche(int[] A, int suche){
2    int start = 1;
3    int ende = "Groesse des Array's A";
4    while(start <= ende){
5      int mitte = (start + ende) / 2;
6      if (A[mitte] == suche){
7        fertig = true;
8        return true;
9      } else {
10       if (suche < A[mitte]){
11         ende = mitte - 1;
12       } else {
13         start = mitte + 1;
14       }
15     }
16   }
17   return false;
18 }
```

Bedeutung des Algorithmus:
In den Zeilen 2-3 definieren wir uns zunächst ein paar Variablen. Eine Variable start, die den Wert 1 hat und eine Variable ende, die die Zahl speichert, die der Anzahl der Einträge des Arrays A entspricht. Nun folgt eine Große while-Schleife. Diese läuft solange, bis $start \leq ende$. Innerhalb der while schleife berechnen wir zunächst mitte, die die Zahl darstellt, die genau zwischen dem Wert von start und ende liegt. Wir schauen dann in die Mitte unseres Array und überprüfen, ob dieser Wert, dem Eingabewert "suche" entspricht. Wenn ja, sind wir fertig und das Programm beendet sich und sagt "Jawohl". Wenn A[mitte] nicht dem Eingabewert "suche" entspricht, dann gelangen wir in den Else Teil. Dort schauen wir, ob "suche" ein kleineren Wert als A[mitte] hat:

- Wenn **ja**, dann setzen wir ende auf mitte -1 (Beachte, dass start = 1 und ende = mitte - 1)

- Wenn **nein**, dann setzen wir start auf mitte+1 (Beachte, dass start = mitte+1)

Wir ersetzen die Werte von start und ende und führen dann die while-Schleife wieder aus, welcher die Werte von start und ende wieder verändert. Dies geht so lange, bis start größer als ende ist. Wir wissen, dass $A[1] \leq A[2] \leq \cdots \leq A[n]$. Aus dieser Tatsache kann man sich leicht überlegen, dass der Algorithmus den Wert "suche" in dem Array sucht und diesen ggf. auch findet.
Diese Suchmethode nennt man auch binäre Suche.

Laufzeitanalyse:

Laufzeitanalyse zu Algorithmus 6.14

Wir schauen uns nun zuerst die if-Bedingung an, die den Teil des Codes umschließt, der zwischen Zeile 10 und 14 liegt. Wir berechnen hier die Laufzeit im schlimmsten Fall, d.h. wir gehen davon aus, dass wir den Rumpf (also den if- oder else-Teil) nehmen, der die meisten Schritte benötigt.

$$s_{10,14} = \max\{s_{11,11}, s_{13,13}\} = \max\{1, 1\} = 1$$

Es gilt: $s_{11,11} = s_{13,13} = 1$, da dort jeweils nur ein Befehl ist.
Analog dazu gehen wir mit der weiteren if-Bedingung vor: (d.h. von Zeile 6 bis 15):

$$s_{6,15} = \max\{s_{7,8}, s_{10,15}\} = \max\{2, 1\} = 2$$

In Zeile 5 kommt noch ein Befehl hinzu, also $s_{5,15} = 1 + s_{6,15} = 3$ Für Zeile 4 bis 16 kommt die riesige while-Schleife. Wir untersuchen diese wieder:

- Die while-Schleife läuft solange gilt start≤ende.
- Innerhalb der Schleife verändert sich der Wert von start oder ende.
- Dieser geänderte Wert hängt von der Variablen mitte ab.
- Mitte teilt die Entfernung zwischen start und ende durch 2.
- Entweder gilt start = mitte + 1; oder ende = mitte - 1;, d.h. die Entfernung von start und ende halbiert sich auf jeden Fall
- Diese halbiert sich so lange bis start > ende.

Wir halbieren immer, d.h. wir müssen schon mal besser als $O(n)$ sein. Wir nehmen an, dass wir ein Arrays nehmen, die eine Länge haben, die sich immer gut halbieren lässt. Diese haben zum Beispiel die Länge $2, 4, 8, ...$ also 2^x. Wir wissen:

$$2^x = \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{x-\text{mal}}$$

Wir können hier, offensichtlicher weise x-mal halbieren bei einer Eingabelänge von 2^x. Klar ist auch:

$$\log_2 2^x = x$$

Jetzt wird offensichtlich, dass bei einem Array der Eingabelänge 2^x die while-Schleife $x = \log_2 2^x$ mal ausgeführt werden muss. Dies gilt auch, bei einem Array der Länge n. Zusammenfasst:

$$s_{4,16} = \log_2 n$$

In Zeile 2,3 und 17 haben wir noch einzelne Befehle:

$$s_{2,17} = 3 + \log_2 n$$

Wir sind fertig, da sonst nichts weiter passiert. Jetzt geht es weiter mit der O-Notation:

$$s_{1,18} = s_{2,17} = 3 + \log_2 n = O(\log_2 n)$$

Ein 2^n Algorithmus

Algorithmus 6.15.

```
1 function viel(int n){
2   int zahl = 2;
3   for (int i = 1; i <= n; i++){
4     zahl = zahl * 2;
5   }
6   for (int i = 1; i <= zahl; i++)[
7     print "Hallo Welt";
8   }
9 }
```

Bedeutung des Algorithmus:
Zunächst definieren wir uns eine Variable, die Anfangs den Wert 2 erhält. Dann wird dies innerhalb einer for-Schleife, die von $i = 1$ bis n läuft, verdoppelt. D.h. die For-schleife berechnet:

$$zahl = \underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 2}_{n-\text{mal}} = 2^n$$

Nach dieser For-Schleife hat zahl den Zahlenwert 2^n gespeichert. Dann kommt eine neue For-Schleife, die von $i = 1$ bis "zahl", also 2^n läuft. Es wird dann "Hallo Welt" ausgegeben. Zusammenfassend kann man sagen: Das Programm gibt 2^n-mal "Hallo Welt" aus.

Laufzeitanalyse:

Laufzeitanalyse zu Algorithmus 6.15

Wir arbeiten zunächst die erste for-Schleife von Zeile 3 bis 5 ab. In Zeile 4 wird ein Befehl ausgeführt:

$$s_{4,4} = 1$$

Die For-Schleife läuft von $i = 1$ bis n. D.h. wir haben für die gesamte For-Schleife:

$$s_{3,5} = \sum_{i=1}^{n} s_{4,4} = \sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \cdots + 1}_{n-\text{mal}} = n$$

Wir kümmern uns nun um die zweite For-Schleife. Wir wissen, dass "zahl" bis dahin den Wert 2^n gespeichert hat. Wir laufen von $i = 1$ bis 2^n d.h.:

$$s_{6,8} = \sum_{i=1}^{\text{zahl}} s_{7,7} = \sum_{i=1}^{2^n} s_{7,7} = \sum_{i=1}^{2^n} 1 = \underbrace{1 + 1 + \cdots + 1}_{2^n-\text{mal}} = 2^n$$

Die For-Schleifen werden hintereinander ausgeführt und in Zeile 2 wird außerdem noch ein Befehl ausgeführt. In Zeile 1 und 9 passiert nichts. D.h.

$$s_{1,9} = s_{2,2} + s_{3,5} + s_{6,8} = 1 + n + 2^n = O(n + 2^n)$$

6.3. Rechenregeln der $\mathcal{O}$-Notation

Wie wir in Abschnitt 6.1 und Abschnitt 6.2 schon gesehen haben, können wir einige Tricks verwenden, um die Zahl in der O-Notation zu verändern. Wir listen diese Regeln einmal formal auf:

Lemma 6.16.
- *Sei d eine feste reelle Zahl. Dann gilt $O(d) = O(1)$.*
- *Sei d eine feste reelle Zahl und sei k eine natürliche Zahl. Dann gilt $O(d \cdot n^k) = O(n^k)$.*
- *Sei d eine feste reelle Zahl und sei $f(n)$ eine Funktion, die von einer natürlichen Zahl n abhängt. Dann gilt: $O(d \cdot f(n)) = O(f(n))$.*

Beweis. Übung, siehe Aufgabe . □

Beispiel 6.17.
Beispielsweise ist $O(3) = O(1)$ (Man lässt sich davon leicht überzeugen, dass die Funktionen gleich schnell wachsen. Nämlich gar nicht). Genauso gilt: $O(3n) = O(n)$. Durch geeignete Wahl von c kann man sich davon überzeugen, dass diese Behauptung ebenfalls stimmt. Der Nachweis ist in den Übungen zu erarbeiten.

Außerdem gilt:

Lemma 6.18.
Wenn $f_1 = O(g_1)$ und $f_2 = O(g_2)$, dann gilt: $f_1 + f_2 = O(g_1 + g_2)$

Beispiel 6.19.
Diese Tatsache haben wir sehr oft, in der Laufzeitanalyse verwendet. Beispielsweise, wenn mehrere Befehle hintereinander kamen, wie z.b. in Algorithmus 6.15 :

$$s_{1,9} = s_{2,2} + s_{3,5} + s_{6,8} = 1 + n + 2^n = O(n + 2^n)$$

A. Kochbuch

A.1. Erste Schritte

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

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

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

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

2. **Shell öffnen**:

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

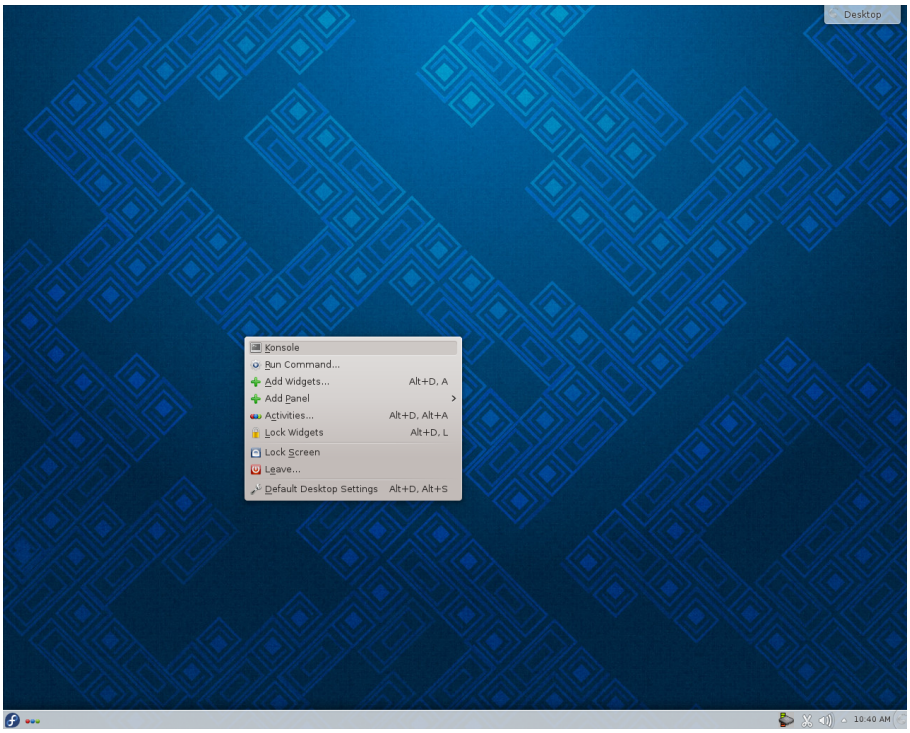


Abbildung A.2.: Terminal öffnen unter KDE

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

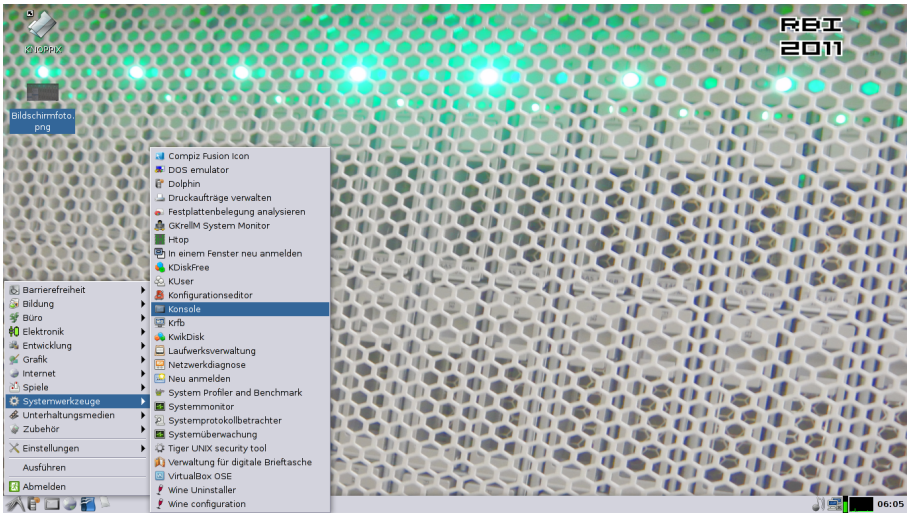


Abbildung A.3.: Terminal öffnen unter Koppix

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

3. **Passwort ändern**:

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

4. **GHCi öffnen**

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

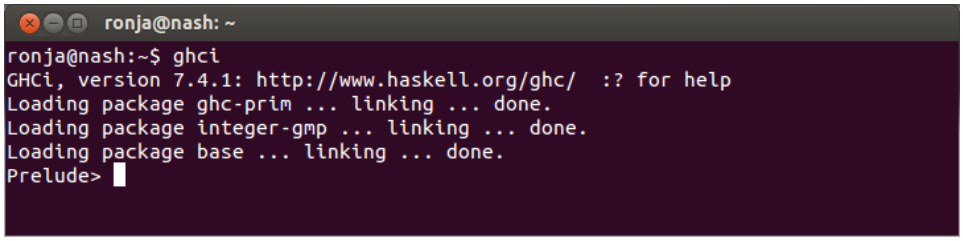


Abbildung A.5.: GHCi

5. **Ein erster Haskell-Befehl**

   - Geben Sie `putStrLn ‘‘Hello world!’’` ein. Dies ist in den meisten Tutorials der meisten Programmiersprachen das Standardbeispiel für ein erstes Programm. Durch Betätigen der Return-Taste ⏎ wird der Befehl direkt ausgewertet.
   - in der Shell erscheint `Hello world!` (Abb. A.6)

Abbildung A.6.: Ein erster Haskell-Befehl

6. **Ein erstes Haskell-Programm**

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

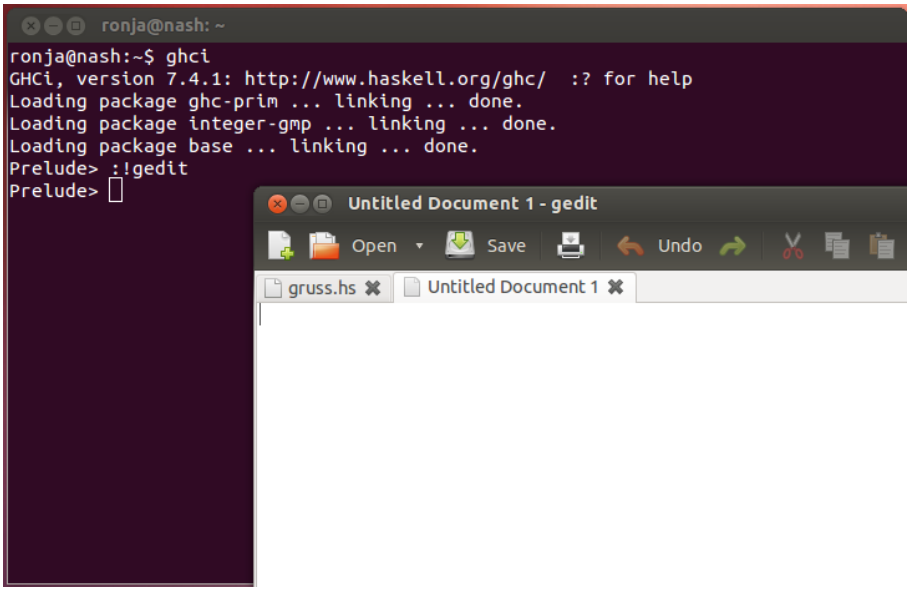


Abbildung A.7.: Öffnen eines Editors

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

Syntax-Highlighting

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

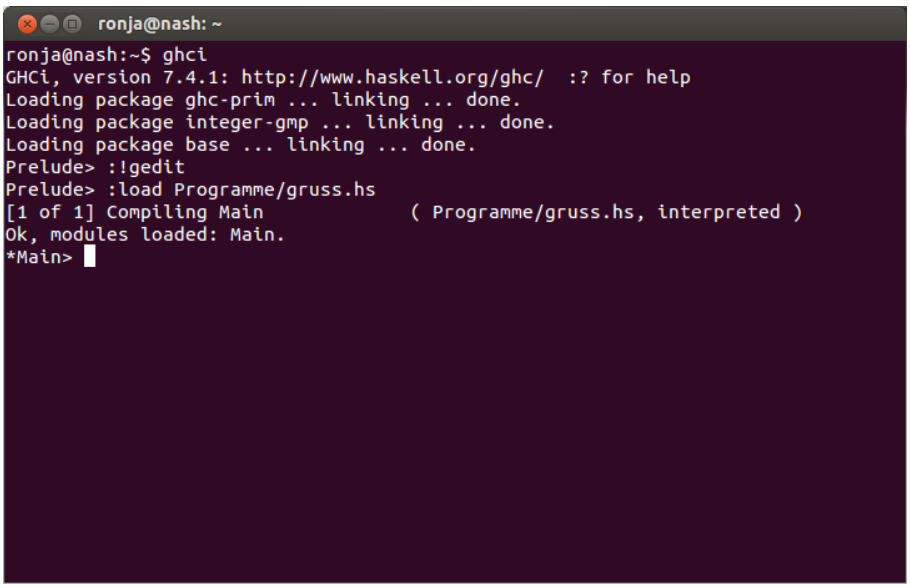


Abbildung A.8.: Laden des Programms

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

Abbildung A.9.: Modulaufruf und Bildschirmausgabe des Programms

7. **Arbeit beenden**

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

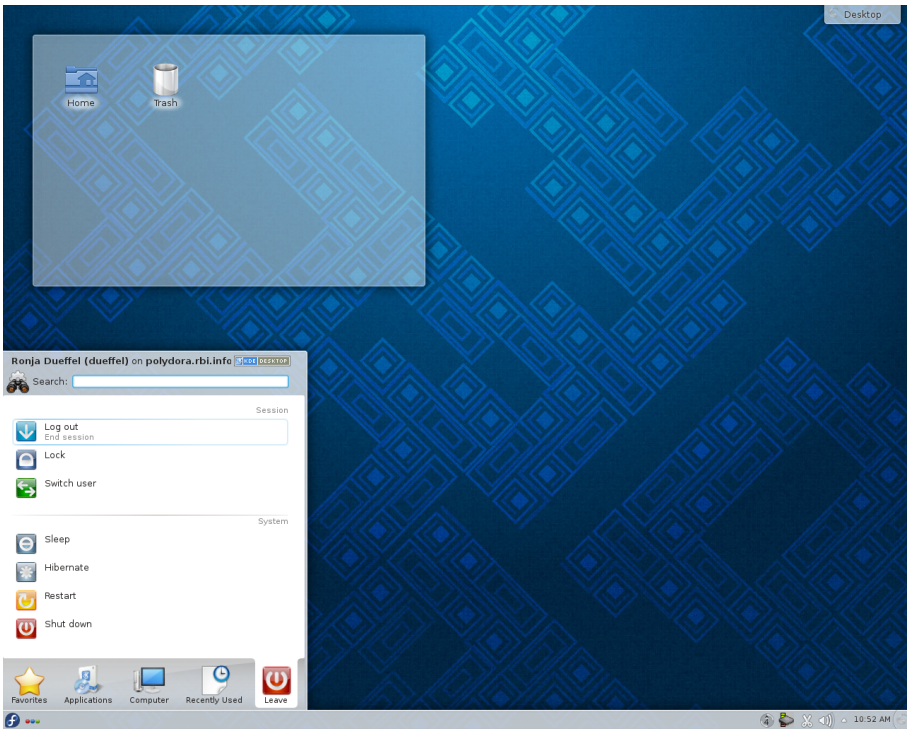


Abbildung A.10.: Ausloggen

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

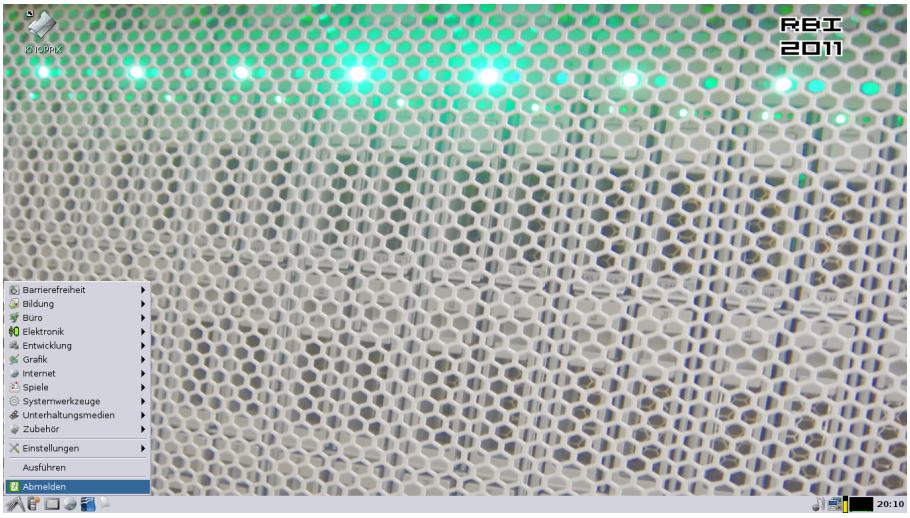


Abbildung A.11.: Herunterfahren

A.2. Remote Login

Auf die Rechner der RBI kann man sich auch über das Internet von einem anderen Rechner (z.B. von zu Hause) aus einloggen. Das ist nützlich, wenn man Programme nicht auf dem eigenen

Rechner installieren möchte, oder um Lösungen zu Programmieraufgaben zu testen, denn meist wird gefordert, dass die Programme auf den RBI-Rechnern laufen. Der Rechner, von dem aus man auf einem RBI-Rechner arbeiten möchte, benötigt hierfür ein *ssh-Client*-Programm. Ferner benötigt man ein *scp-Client*-Programm, um Dateien zwischen den Rechnern auszutauschen. Außerdem muss man wissen, auf welchem RBI-Rechner man sich einloggen möchte. Eine Liste der Rechnernamen findet man auf der RBI-Webseite[1].

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

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

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

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

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

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

   Der `.` steht dabei stellvertretend für das aktuelle Verzeichnis. Es ist auch möglich einen relativen oder absoluten Pfad (siehe Kap. 1) 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.12).

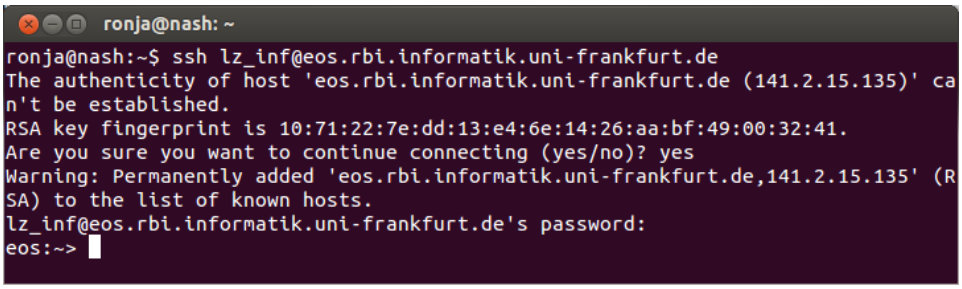
   

   Abbildung A.12.: Auf einem RBI-Rechner einloggen

3. **Programm starten**

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

   ```
   > ghci [meinprogramm.hs] ↵
   ```

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

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

4. **Verbindung beenden**

```
> exit ↵
```

Dieser Befehl schließt die Verbindung zu dem RBI-Rechner.

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

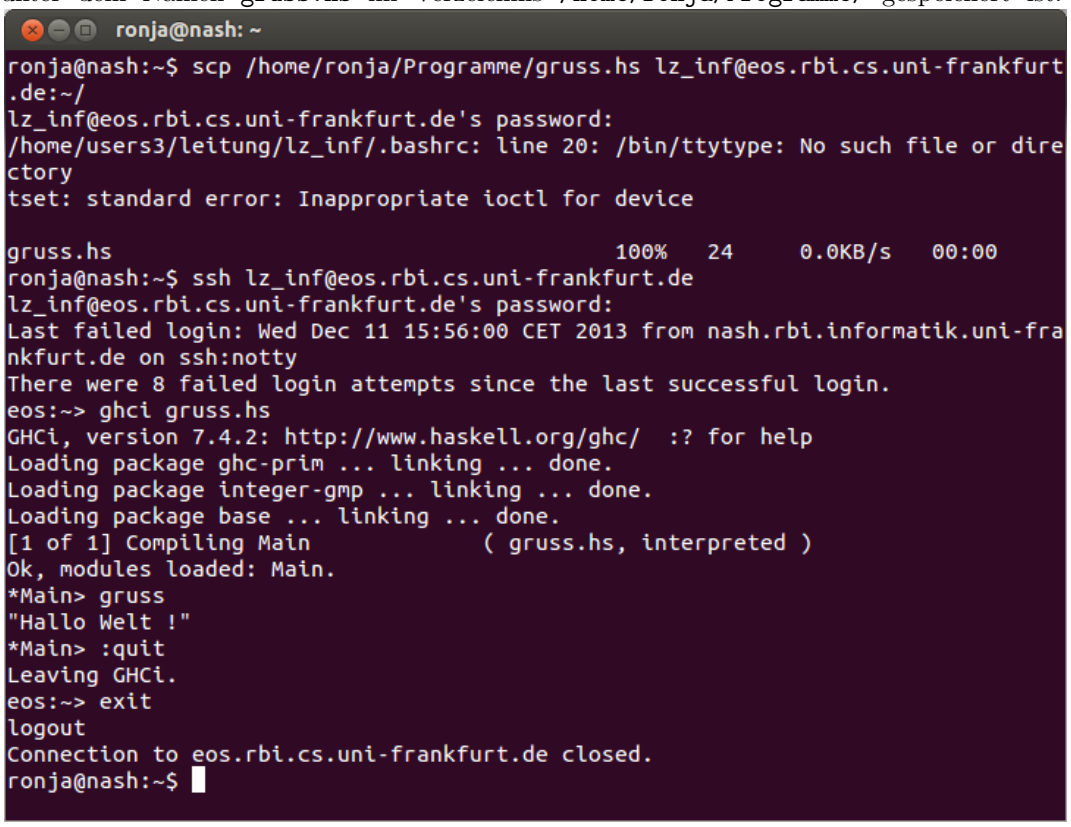


Abbildung A.13.: Ein Haskell-Programm vom eigenen Rechner auf einem RBI-Rechner starten

A.2.2. Windows

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

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

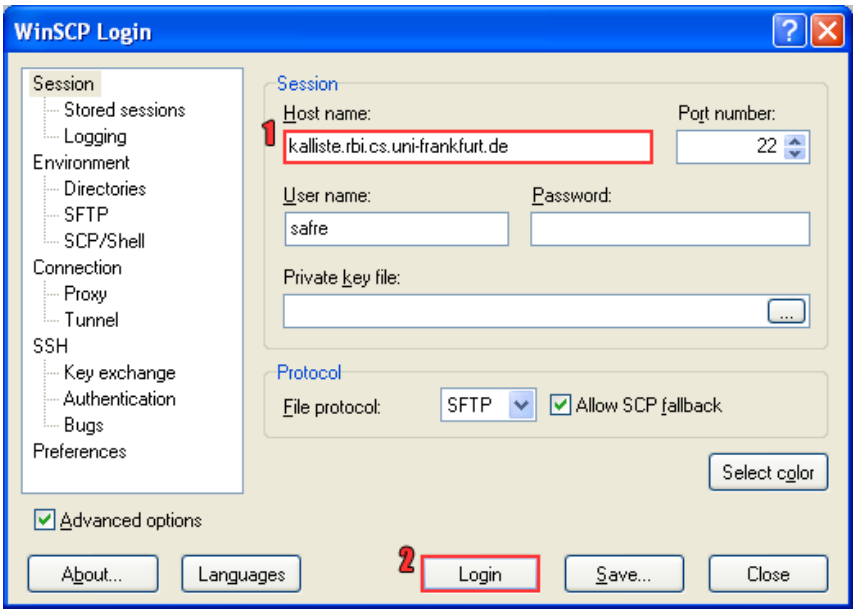


Abbildung A.14.: Das WinSCP-Login-Fenster

Sind die Rechner verbunden, können Daten wie man es vom Dateimanager gewohnt ist, mit drag-and-drop oder über das Menü ausgetauscht werden (Abb. A.15).

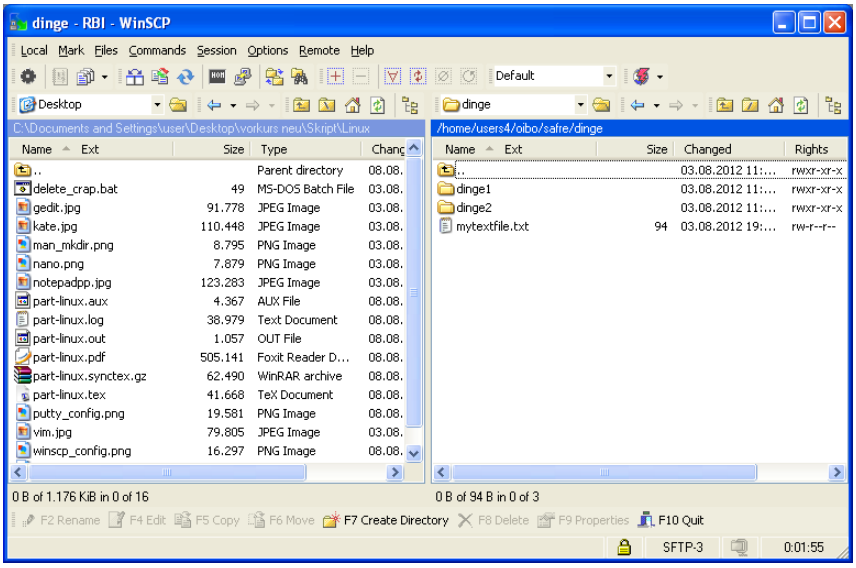


Abbildung A.15.: Das WinSCP-Hauptfenster. Links die Dateien des Windows-Rechners, rechts die Dateien im RBI-Verzeichnis

2. **Einloggen** Um sich von einem Windows-Rechner auf einem RBI-Rechner einzuloggen, benötigt man ein ssh-Client-Programm. *Putty*[3] ist solch ein Programm. Bei der Einstellung des Programms, muss die Adresse des RBI-Rechners angegeben werden, auf dem man sich einloggen möchte (Abb.: A.16).

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

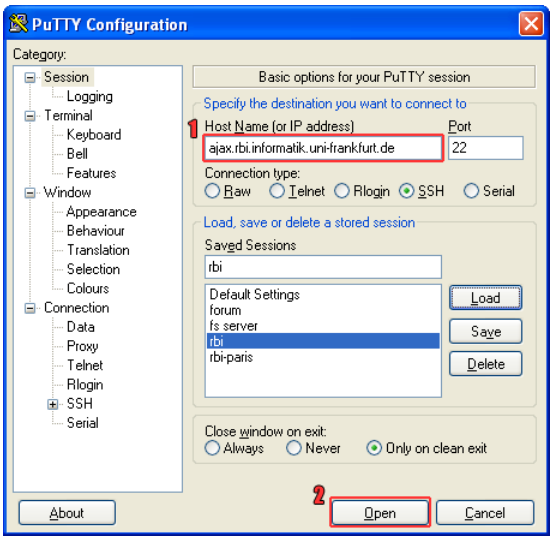


Abbildung A.16.: Einstellung von Putty

Anschließend kann man sich auf dem RBI-Rechner einloggen. Ist der Vorgang erfolgreich, so öffnet sich eine Shell, über die man auf dem RBI-Rechner Kommandos ausführen kann.

3. **Programm starten**

   In der Putty-Shell kann man nun mit folgendem Kommando das Haskell-Programm starten.

   ```
   > ghci [meinprogramm.hs] ↵
   ```

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

B. Imperatives Programmieren und Pseudocode

In diesem Kapitel führen wir allgemein (unabhängig von einer konkreten Programmiersprache) einige wichtige Konzepte der imperativen Programmierung ein. Zur besseren Verständlichkeit kann der Sachverhalt an einigen Stellen etwas vereinfacht dargestellt sein.

In der theoretischen Informatik werden Datenstrukturen und Algorithmen meist in **imperativem** Pseudocode dargestellt und analysiert.

Konzept

Pseudocode

Wie in Kapitel 2 erwähnt, ist das Konzept der imperativen Programmierung folgendes: Ein Programm besteht aus einer Folge von Befehlen (bzw. Anweisungen), die nacheinander ausgeführt werden. Die Ausführung eines jeden Befehls ändert den **Zustand** (Speicher) des Rechners. Pseudocode ist ein informaler Code. Er dient zur Veranschaulichung eines Algorithmus und enthält häufig Elemente der natürlichen Sprache, während die Struktur/Syntax einer Programmiersprache ähnelt. Sinn und Zweck des Pseudocodes ist es, die Schlüsselprinzipen eines Algorithmus oder Programms effizient und unabhängig von der zugrunde liegenden Technologie in einer für Menschen gut verständlichen Art und Weise darzustellen. In den folgenden Beispielen verwenden wir einen C++ nahen Pseudocode und halten uns dabei an die Notation wie sie auch im Skript der Vorlesung "Datenstrukturen" verwendet wird. Um einzelne Befehle einer Sequenz voneinander zu trennen, beenden wir jeden Befehl mit einem Semikolon. Zusammengehörende Codeblöcke fassen wir in geschweiften Klammern {,} zusammen.

B.1. Lesen und Beschreiben von Speicherplätzen: Programmvariablen

Programmvariable

Innerhalb einer höheren, imperativen Programmiersprache wird vom realen Speichermodell abstrahiert. Es werden **Programmvariablen** verwendet, die Namen für Speicherplätze darstellen. Der Wert, der im zugeordneten Speicherplatz steht, kann über den Namen der Variablen gelesen oder auch verändert werden.

Beispiel B.1.
Seien X und Y Programmvariablen. Wenn der zu X zugehörige Speicherplatz den Wert 123 beinhaltet, und der zu Y zugehörige Speicherplatz den Wert 321 beinhaltet, so wird innerhalb eines imperativen Programms der Ausdruck $X + Y$ zu 444 ausgewertet.

Bei der Ausführung werden zunächst die Werte der Speicherplätze X und Y gelesen, anschließend wird addiert.

Zuweisung

In imperativen Programmiersprachen sind Programmvariablen *veränderlich*, d.h. man kann den Wert des entsprechenden Speicherplatz *verändern*. Dies übernimmt ein Befehl, der als **Zuweisung** bezeichnet wird:

$$X := \text{Ausdruck};$$

Der Variablen X wird ein neuer Wert zugewiesen. Hierfür wird zunächst der Wert des Ausdrucks berechnet, und *anschließend* der neue Wert für X in den Speicherplatz geschrieben. Die Reihenfolge ist wichtig!

Beispiel B.2.
Betrachte die Zuweisung

$$X := X + 1;$$

Die rechte Seite der Zuweisung wird *zuerst* zu einem Wert ausgewertet. Wenn X vorher den Wert 123 hat, so ergibt die Berechnung von $X+1$ den Wert 124. *Anschließend* wird der zu X zugehörige Speicherplatz durch den Wert 124 überschrieben.

Beispiel B.3.
Betrachten wir folgende Anweisungsfolge:

$$\begin{aligned} X &:= 3; \\ Y &:= 5; \\ X &:= Y + 1; \\ Y &:= X - 2; \end{aligned}$$

Wir nehmen an X und Y existieren am Anfang nicht. Dann legt der Compiler/Interpreter bei der ersten Zuweisung einen neuen Speicherplatz für die entsprechenden Programmvariablen an. Nach Ausführen der Befehle $X := 3$ und $Y := 5$ sind die Variablen X und Y vorhanden und die zugehörigen Speicherplätze mit den Werten 3 und 5 belegt. Die Zuweisung $X := Y + 1$; modifiziert den Wert von X zu 6 $(5 + 1 = 6)$ und die weitere Zuweisung $Y := X - 2$ führt dazu, dass Y den Wert 4 $(6 - 2 = 4)$ erhält.

undefinierte Programm-variable

Wird eine Programmvariable gelesen, ohne dass ihr zugehöriger Wert vorher per Zuweisung gesetzt wurde, so ist der Wert der Variablen **undefiniert** und es tritt während der Ausführung ein Fehler auf.

Beispiel B.4.
Betrachten wir das imperative Programm:

$$\begin{aligned} X &:= Y + 1; \\ Y &:= X - 2; \end{aligned}$$

Beim Auswerten der ersten Zuweisung $X := Y + 1$ ist der Wert von Y nicht gesetzt, daher kann der Wert von $Y + 1$ nicht ermittelt werden, und es tritt ein Fehler auf.

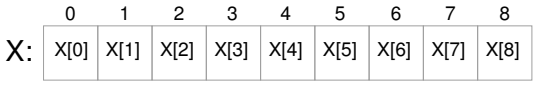
B.2. Felder: Indizierte Speicherbereiche

Array

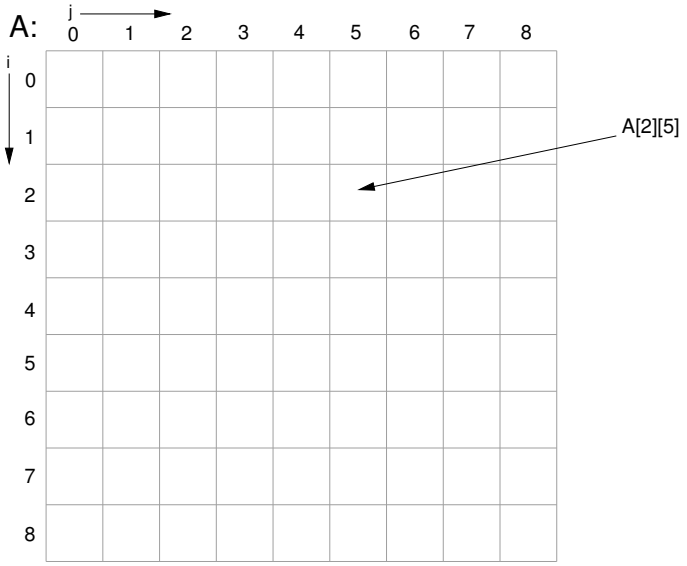
Mit den oben beschriebenen Programmvariablen kann nun auf einzelne Speicherbereiche zugefriffen werden, aber was tut man, wenn man mehrere Werte in den Speicher schreiben möchte, aber noch nicht genau weiß wie viele? Bei diesem Problem hilft die Datenstruktur des **Feldes** (engl. **Array**) weiter. Einen Array kann man sich als nebeneinander liegende Speicherplätze vorstellen. Dem Array wird eine Programmvariable zugewiesen und auf die einzelnen Elemente (Speicherplätze) des Array kann dann über einen **Index** zugegriffen werden. Der Index ist ganzzahlig und beginnt bei 0 [1]. Felder haben eine **Dimension**, und für jede Dimension gibt es einen Index. Wir verwenden die Notation $X[i]$ um den $i + 1$. Speicherplatz eines **eindimensionalen** Feldes X zu bezeichnen.

Index
Dimension von Feldern

[1] Informatiker fangen sehr oft beim Zählen mit der 0 an.



$A[i][j]$ adressiert den $i+1$, $j+1$. Speicherplatz eines **zweidimensionalen** Feldes A. Diese Adressierungen $X[i]$ und $A[i][j]$ kann man genauso wie eine Programmvariable verwenden: Man kann den Wert lesen oder den Wert im Speicherplatz durch eine Zuweisung verändern.

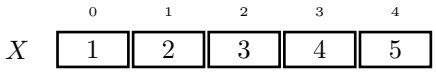


Beispiel B.5.
Sei X ein eindimensionales Array der Größe 5, so schreibt der folgende imperative Algorithmus die Werte 1 bis 5 der Reihe nach in die Speicherplätze des Feldes

$$\begin{aligned} X[0] &:= 1; \\ X[1] &:= 2; \\ X[2] &:= 3; \\ X[3] &:= 4; \\ X[4] &:= 5; \end{aligned}$$

Meist gibt man durch einen Befehl, z. B. der Form `new` $X[5]$, am Anfang an, dass ein Array der Größe 5 erstellt werden soll.

Eine Veranschaulichung des Arrays X nach Ausführung der Zuweisungen ist:



Die Zahlen 0 bis 4 über dem Array, zeigen den entsprechenden Index jedes Array-Elements.

Führt man nun die Befehlssequenz:

$$\begin{aligned} &i := 0; \\ &X[i] := X[i+1]; \\ &i := i+1; \\ &X[i] := X[i+1]; \\ &i := i+1; \\ &X[i] := X[i+1]; \end{aligned}$$

aus, so geschieht das Folgende: Zunächst hat die Variable i den Wert 0. Daher wird im zweiten Befehl das Array-Element $X[0]$ auf den Wert von $X[1]$ (der 2 ist) gesetzt. D.h. nach Ausführung der zweiten Zeile ist X das Array:

	0	1	2	3	4
X	2	2	3	4	5

In der dritten Zeile wird der Wert von i um 1 erhöht (der Wert ist nun 1). In der vierten Zeile wird $X[i]$ (also $X[1]$) auf den Wert von $X[2]$ (also 3) gesetzt. D.h. danach ist X

	0	1	2	3	4
X	2	3	3	4	5

Die nächsten beiden Befehle führen dazu, dass $X[3]$ den Wert 4 erhält, d.h. am Ende kann man X darstellen durch:

	0	1	2	3	4
X	2	3	4	4	5

Ein weiteres Beispiel:

Beispiel B.6.
Wir betrachten den Befehl:

$$X[X[2]] := X[X[0]];$$

Was macht der Befehl? Zunächst müssen wir die rechte Seite der Zuweisung auswerten: $X[X[0]]$ bezeichnet den Speicherplatz im Array X mit dem Index $X[0]$: Daher müssen wir zunächst den Wert von $X[0]$ aus dem Speicher lesen: Er ist 2. Daher ist $X[X[0]]$ der Wert von $X[2]$ und daher 4. Für die linke Seite der Zuweisung müssen wir den entsprechenden Speicherplatz herausfinden, der geändert werden soll. Hierfür müssen wir zunächst den Wert von $X[2]$ lesen. Dieser ist 4. Daher müssen wir den Speicherplatz $X[4]$ mit dem Wert 4 beschreiben. D.h. es ergibt sich das Array:

	0	1	2	3	4
X	2	3	4	4	4

B.3. Kontrollstrukturen: Verzweigungen und Schleifen

B.3.1. Verzweigungen: Wenn-Dann-Befehle

`If-Then-Else`-Befehl

Verzweigungen steuern den Programmablauf: Je nachdem, ob eine Bedingung erfüllt ist oder nicht, wird die eine oder die andere Anweisungsfolge ausgeführt. Wir verwenden hierfür `IF` − `THEN` − `ELSE` als Pseudocode in der Form:

`IF` Bedingung `THEN` {Anweisungfolge$_1$} `ELSE` {Anweisungsfolge$_2$}[2].

[2] Beachte, dass ein solches `IF-THEN-ELSE` im Gegensatz zu Haskell *kein* Ausdruck ist, sondern ein Befehl, d.h. insbesondere das Konstrukt hat keinen Wert, sondern die Ausführung verändert den Speicher.

Beispiel B.7.
Betrachten wir

```
X := 10;
IF X > 5
 THEN {
      X := X + 1;
      Y := X; }
 ELSE{
      X := X − 1; }
Z := X + 1;
```

Für die Ausführung der `IF-THEN-ELSE`-Anweisung wird zuerst der Wert von $X > 5$ berechnet. Da X den Wert 10 hat, wird $X > 5$ zu `True` auswerten. Daher wird in den `THEN`-Zweig gesprungen, und die beiden Befehle $X := X + 1$ und $Y := X$ werden nacheinander ausgeführt. Anschließend wird in Zeile 8 nach dem `IF-THEN-ELSE` gesprungen und die Zuweisung $Z := X + 1$ wird ausgeführt.

Da in imperativen Sprachen `IF-THEN-ELSE`-Konstrukte Befehle sind, kann man auch `IF-THEN`-Konstrukte erlauben. Diese haben keinen `ELSE`-Zweig und sind daher von der Form: `If-Then`-Befehl

`IF` Bedingung `THEN` {Anweisungsfolge$_1$}.

Die Bedeutung ist: Nur wenn die Bedingung erfüllt ist, wird die Anweisungsfolge$_1$ ausgeführt, anderenfalls wird mit dem Befehl nach dem `IF-THEN` fortgefahren.

Beispiel B.8.
Betrachte zum Beispiel:

```
X := 10;
Y := 20;
IF X > Y THEN{
      Y := Y − X; }
R := Y;
```

Der `THEN`-Zweig wird in diesem Programm nicht ausgeführt, da die Bedingung $X > Y$ nicht erfüllt ist. Somit wird R auf den Wert 20 gesetzt.

B.3.2. Schleifen

Schleifen dienen dazu, ein bestimmtes Programmstück wiederholt auszuführen. Wir erläutern hier nur die `FOR`- und die `WHILE`-Schleife.

Die **`FOR`-Schleife** führt ein Programmstück sooft aus, bis die gewünschte Anzahl an Wiederholungen erreicht wurde. Die Syntax in Pseudocode ist: `For`-Schleife

`FOR` (*Variable* = *Startwert*; *Bedingung*; *Anweisung*)
{*Schleifenkörper*}

Hierbei ist *Variable* eine Programmvariable, die zum Zählen verwendet wird (daher wird diese auch oft **Zählvariable** oder **Zähler** genannt). Der **Startwert** wird der Variablen zu Beginn der Ausführung der Schleife zugewiesen. Anschließend wird geprüft, ob die **Bedingung** erfüllt ist. Dabei handelt es sich um einen Ausdruck, der in `true` oder `false` ausgewertet werden kann. Ist die Bedingung erfüllt, werden die Befehle im Schleifenkörper ausgeführt. Anschließend wird die **Anweisung** ausgeführt, (die im allgemeinen den Wert der Variablen verändert). Dann folgt der nächste Durchlauf, der wieder mit dem Prüfen der Bedingung beginnt. Sobald die Bedingung nicht erfüllt ist, wird der Schleifenkörper nicht durchlaufen, und es wird zum Befehl nach der Schleife gesprungen. Zähler, Startwert, Bedingung

Beispiel B.9.

$$
\begin{array}{l}
Y := 0; \\
\texttt{FOR}\ (X := 1; X \leq 3; X{+}{+})\{ \\
\quad Y := Y + X; \} \\
Z := Y;
\end{array}
$$

Zunächst wird Y auf 0 gesetzt, anschließend beginnt die `FOR`-Schleife. Die Zählvariable ist X und hat den Startwert 1. Die Bedingung ist erfüllt, also wird der Schleifenkörper durchlaufen und Y erhält den Wert 1. Nun wird die Anweisung $X{+}{+}$ ausgeführt [3], daher wird X auf 2 gesetzt. Da die Bedingung noch erfüllt ist, wird der Schleifenkörper erneut durchlaufen und Y erhält den Wert 3. Anschließend wird X auf 3 erhöht. Die Bedingung ist weiterhin erfüllt und es folgt ein dritter Durchlauf des Schleifenkörpers, der Y auf 6 setzt. Nun wird durch die Anweisung X auf 4 erhöht. Die Bedingung ist nun nicht erfüllt und die Schleife wird nicht weiter durchlaufen, es wird zur Zuweisung $Z := Y$ gesprungen.

Endlosschleife

Bei der Schleifenprogrammierung muss darauf geachtet werden, dass keine **Endlosschleife** entsteht. Dies ist der Fall, wenn die Bedingung nie falsch wird. Das Programm durchläuft dann endlos den Schleifenkörper.

Beispiel B.10.
Betrachte als schlechtes Beispiel das Programm:

$$
\begin{array}{l}
Y := 0; \\
\texttt{FOR}\ (X := 1; X \leq 3; X := X)\{ \\
\quad Y := Y + X; \} \\
Z := Y;
\end{array}
$$

Da der Wert von X nie verändert wird, ist die Schleifenbedingung immer wahr, und der Schleifenkörper wird endlos oft ausgeführt (man sagt auch „die Schleife wird nie *verlassen*“).

`WHILE`-Schleife

Die **`WHILE`-Schleife** führt ein Programmstück (den Schleifenkörper) aus, *solange* eine Bedingung erfüllt ist. Die Syntax in Pseudocode ist:

`WHILE` *Bedingung*
{*Schleifenkörper*}

Zu Beginn wird die *Bedingung* überprüft. Ist sie erfüllt, so wird der *Schleifenkörper* ausgeführt und es wird wieder zum Anfang gesprungen, d.h. Prüfen der Bedingung. Erst wenn die Bedingung nicht erfüllt ist, wird die Schleife verlassen.

Beispiel B.11.

$$
\begin{array}{l}
X := 1; \\
Y := 0; \\
\texttt{WHILE}\ (X \leq 3)\ \{ \\
\quad\quad Y := Y + X; \\
\quad\quad X := X + 1; \} \\
Z := Y;
\end{array}
$$

Zunächst werden den Variablen X und Y die Werte 1 und 0 zugewiesen, anschließend beginnt die Schleife: Die Bedingung ist erfüllt, daher werden die beiden Zuweisungen durchgeführt (Y erhält den Wert 1, X erhält den Wert 2). Für den nächsten Durchlauf ist die Bedingung erneut erfüllt, daher wird der Schleifenkörper durchlaufen (X und Y erhalten den Wert 3). Beim erneuten Prüfen

[3] “++” stammt hier aus der Programmiersprache C++ und bedeutet, dass X inkrementiert (um 1 erhöht wird).

der Bedingung ist diese wiederum erfüllt und Y erhält den Wert 6, und X den Wert 4. Nun ist die Bedingung nicht mehr erfüllt. Die Schleife wird verlassen und die Zuweisung $Z := Y$ wird ausgeführt.

Man sieht, dass man die `FOR`- und die `WHILE`-Schleife gegeneinander austauschen kann, die Programmierung ist in beiden Fällen sehr ähnlich.

Beispiel B.12.
Als abschließendes Beispiel betrachten wir ein Programm, das in einem Array nach einem bestimmten Wert sucht und den entsprechenden Index in der Variablen R abspeichert, falls der Wert existiert (und -1, falls der Wert nicht existiert): Sei X ein Array der Größe n, und S eine Variable, die den zu suchenden Wert enthält.

$R := -1;$
$Z := 0;$
`WHILE` $R = -1$ und $Z < n$ {
  `IF` $X[Z] = S$ `THEN`
    $\{R := Z;\}$
  $Z := Z++;\}$

Zu Beginn wird der Variablen R der Wert -1 und der Variablen Z der Wert 0 zugewiesen. Mit einer `WHILE`-Schleife wird der Array durchlaufen, solange R den Wert -1 **und** Z einen Wert hat der kleiner als n (die Länge des Arrays) ist. Im Schleifenkörper wird dann überprüft, ob der Wert des Arrays am Speicherplatz mit Index Z ($X[Z]$) gleich dem gesuchten Wert (der in Variable S gespeichert ist) ist. Falls ja, wird der aktuelle Wert von Z dem durch die Variable R adressierten Speicherplatz zugewiesen [4] . Ansonsten wird der Wert von Z um 1 erhöht. Damit wird im nächsten Durchlauf des Schleifenkörpers geschaut, ob das nächste Element im Array den gesuchten Wert enthält. Falls auch dieses Element nicht mit S übereinstimmt, wird Z ein weiteres mal erhöht, usw. bis Z das Ende des Arrays (nämlich n) erreicht hat.

[4] damit hat R nicht länger den Wert -1 und der Schleifenkörper wird nicht noch einmal durchlaufen

C. Zum Lösen von Übungsaufgaben

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

müssen