

Funktionale Programmierung mit Haskell

Dr. Hermann Puhlmann
puhlmann@informatik.uni-erlangen.de

März 2003

Entwurf
Nur zur Verwendung im Signal-Kurs 2002/2003

Inhaltsverzeichnis

1	Einordnung funktionaler Programmierung	3
2	Haskell und Hugs	4
3	Alles hat seinen Typ	5
3.1	Elementare Datentypen	6
3.1.1	Wahrheitswerte	6
3.1.2	Zeichen und Zeichenketten	6
3.1.3	Ganze Zahlen	7
3.1.4	Gleitkommazahlen	7
3.2	Operatoren und Funktionen für einfache Typen	8
3.2.1	Vergleichsoperatoren	8
3.2.2	Vordefinierte Funktionen	9
3.3	Übungen	10
4	Funktionen als Programme	12
4.1	Eine Eingabe — eine Ausgabe	12
4.2	Mehrere Eingaben — eine Ausgabe	14
4.3	Je nachdem: Fallunterscheidungen	15
4.4	Immer wieder: Rekursion	17
4.5	Ansammlung von Zwischenergebnissen: Akkumulatoren	18
4.6	Übungen	20
4.6.1	Einfache Funktionen	20
4.6.2	Funktionen mit mehreren Argumenten	21
4.6.3	Mustervergleich und Bedingungen	21
4.6.4	Rekursive Funktionsdefinitionen	22
4.6.5	Akkumulatoren	22
4.6.6	Testaufgabe	22
5	Datenstrukturen	23
5.1	Listen	23
5.1.1	Listendarstellung	23
5.1.2	Listenkonstruktion und -bearbeitung	24
5.2	Tupel	28
5.2.1	Currying und uncurrying	29
5.3	Übungen	30
5.3.1	Listenverarbeitung	30

5.3.2	Tupel und Currying	32
5.3.3	Vermischte Übungen	33
6	Funktionen höherer Ordnung	34
6.1	Überall dasselbe tun: <code>map</code>	34
6.1.1	Fallbeispiel Cäsarchiffre	34
6.1.2	Weitere Beispiele mit <code>map</code>	37
6.2	Elemente auswählen: <code>filter</code>	37
6.3	Alles zusammenfassen: <code>fold</code>	38
6.4	Funktionen manipulieren	41
6.4.1	Eins nach dem anderen: Funktionskomposition	42
6.4.2	Nur nicht zu viel: partielle Auswertung	43
6.4.3	Die Argumentreihenfolge vertauschen: <code>flip</code>	45
6.5	Übungen	46
A	Lösungen zu den Übungen	49
A.1	Lösungen zu Kapitel 3	49
A.2	Lösungen zu Kapitel 4	51
A.3	Lösungen zu Kapitel 5	54
A.4	Lösungen zu Kapitel 6	60

1 Einordnung funktionaler Programmierung

Programmiersprachen klassifiziert man nach sogenannten Programmierstilen oder Programmierparadigmen. Die Hauptunterteilung ist die zwischen imperativen und deklarativen Programmiersprachen.

Dabei orientieren sich imperative Programmiersprachen besonders stark an der gängigen Rechnerarchitektur, indem Programme aus Anweisungen bestehen, die Inhalte des Computerspeichers manipulieren. Grob gesagt enthalten die Programme Namen für Speicherbereiche (sogenannte Variable). Mit diesen Namen kann auf Speicherbereiche zugegriffen werden, mit den Speicherinhalten können Berechnungen durchgeführt werden, und die Ergebnisse der Berechnungen können unter Verwendung des Zuweisungsoperators (meist `:=` oder `=`) wieder im Speicher abgelegt werden.

Im Unterschied dazu sind deklarative Programme Beschreibungen einer Problemlösestrategie innerhalb eines mathematischen Kalküls. Innerhalb der deklarativen Programmierung unterscheidet man zwischen logik-orientierter Programmierung und funktionaler Programmierung.

Hauptvertreter der logik-orientierten Programmierung ist die Sprache PROLOG, in der Programme aus logischen Aussagen und Folgerungsregeln bestehen. Hinzu kommt ein spezieller Auswertungsmechanismus, der zu dieser Problembeschreibung die „wahren Aussagen“ als Lösung liefert.

Funktionale Programme sind eine Ansammlung von Funktionen im mathematischen Sinne. Jede Funktion hat einen Typ, der die Typen der Ein- und Ausgabewerte angibt, und eine Zuordnungs- oder Funktionsvorschrift, die angibt, auf welche Weise aus den Eingaben die Ausgaben generiert werden. Dies mag zunächst nicht sehr vielversprechend aussehen, wenn man an die Funktionen denkt, die im schulischen Mathematikunterricht betrachtet werden und die nur von Zahlen handeln. Bedenkt man jedoch, dass funktionale Programmiersprachen eine reichhaltige Auswahl an Datentypen bereit stellen und eine Funktion als Ein- und Ausgabe beispielsweise Listen von Zeichenketten haben kann (die Funktion könnte die Zeichenketten z. B. alphabetisch sortieren), so kann man erahnen, dass Funktionen ein ausdrucksstarkes Mittel der Programmierung sind.

Noch erfreulicher ist, dass mathematische Funktionen die Eigenschaft haben, zu jeder (erlaubten) Eingabe immer (wieder) dieselbe Ausgabe zu erzeugen (bei sogenannten „Funktionen“ in imperativen Programmiersprachen ist das nicht garantiert), denn diese Eigenschaft wünscht man sich auch von Computerprogrammen. In diesem Sinne ist jedes Computerprogramm (bei dem alle Eingaben zu Beginn vorliegen und das am Ende eine Ausgabe liefert) eine Funktion. Diese Funktion wird in der Regel zu komplex sein, um sie in einer einzigen Funktionsdefinition angeben zu können. Daher teilt man die Programmieraufgabe so lange in kleinere Teilaufgaben auf, bis man jede Teilaufgabe durch eine Funktion ausdrücken kann, die in wenigen Zeilen programmiert wird. Andere Funktionen führen diese Teilbausteine zusammen, und das Zusammenspiel der Teile bewirkt die Ausführung einer komplexen Aufgabe.

Dabei genügt es zum Verständnis eines funktionalen Programms, Funktionsargumente in Funktionsdefinitionen einsetzen zu können, d. h. Funktionen auf Argumente anzuwenden. Auch wenn die Implementierung einer funktionalen Sprache auf einem Computer weitere Techniken erfordert, ist für den funktionalen Programmierer das mathematische Verständnis ausreichend. Es ist nicht notwendig, wie beim imperativen Programmieren die Mechanismen des Speicherzugriffs zu kennen oder wie beim logik-orientierten Programmieren Suchverfahren für wahre Aussagen zu kennen. Das Verständnis des Funktionsbegriffs genügt, und so ist der funktionale Programmierstil für alle mathematisch (wenigstens leicht) vorgebildeten gut zugänglich.

2 Haskell und Hugs

Haskell ist eine moderne funktionale Programmiersprache, benannt nach Haskell B. Curry (1900–1982), einem amerikanischen Logiker und Schüler David Hilberts. Curry war ein Pionier der kombinatorischen Logik und des λ -Kalküls, den mathematischen Grundlagen funktionaler Programmiersprachen.

DIE Internetadresse zur Programmiersprache Haskell ist

`http://www.haskell.org`

Dort finden sich auch Haskell-Compiler und Haskell-Interpreter zum Download. Im Rahmen dieses Materials wird die Verwendung des Haskell-Interpreters HUGS angenommen, der für verschiedene Betriebssysteme (darunter Windows und Linux) von

`http://cvs.haskell.org/Hugs/pages/downloading.htm`

bezogen werden kann (man kommt dorthin auch durch passendes Klicken von `www.haskell.org`). Die (zur Zeit aktuelle) selbstinstallierende Version für Windows-Betriebssysteme auf dieser Seite ist

`hugs98-Nov2002.msi`

Diese Version enthält auch die Hugs-Oberfläche „winhugs“, die das Arbeiten mit Hugs durch verschiedene Knöpfe zum Anklicken dem Arbeitsstil der Mausbenutzer anpasst.

Grundsätzlich benötigt man zum Programmieren mit Hugs zweierlei Programme: Hugs selbst und einen Editor, in dem man die Programme schreibt. Das Vorgehen dazu ist in den Übungsabschnitten beschrieben. Innerhalb der Hugs-Arbeitsfläche kann man Programme ausführen lassen. Die entsprechenden Eingaben schreibt man hinter den Hugs-Prompt, der beispielsweise so aussieht:

```
Prelude>
```

In diesem Text wird nur das Symbol

```
>
```

verwendet, um den Prompt anzudeuten. Zeilen, die mit `>` beginnen, sind also Eingabezeilen, die darauf folgenden Zeilen sind Ausgaben von Hugs. Programmtexte (die man im Editor eingibt) enthalten ebenfalls kein einleitendes `>`-Symbol.

3 Alles hat seinen Typ

Haskell ist eine streng getypte Sprache, d. h. jeder Haskell-Ausdruck hat einen Datentyp, und beim Bilden von zusammengesetzten Ausdrücken muss man darauf achten, dass die Typen der Teilausdrücke zusammen passen. In sehr vielen Fällen kann Haskell die Typen von (Teil-)Ausdrücken selbst erkennen, da sie durch den Verwendungskontext der Ausdrücke erzwungen werden.

Beispiel 1 Die eingebaute Funktion `ord` liefert zu einem Zeichen dessen ASCII-Code. In der Funktionsanwendung `ord x` muss `x` daher eine Variable sein, die den Typ `Char` („Zeichen“) hat¹.

Manchen Ausdrücken können verschiedene gültige Typen zugeordnet werden. Anstelle eines Typs ordnet Haskell solchen Ausdrücken eine *Typklasse* zu, die die verschiedenen zulässigen Typen umfasst.

Beispiel 2 Im Ausdruck `2*x` muss `x` einen Typ haben, der in Multiplikationen mit der Zahl 2 verwendet werden darf. Zu diesen Typen zählen `Int` (ganze Zahlen), `Integer` (ganze Zahlen mit beliebiger Stellenzahl) oder `Float` (Gleitkommazahlen). Diese Typen sind in der Typklasse `Num` (Zahlentypen) enthalten, die daher der Variablen `x` zugeordnet wird.

Obwohl Haskell die Typen der meisten Ausdrücke selbst erkennt und so überprüfen kann, ob zusammengesetzte Ausdrücke korrekt gebildet sind, ist es eine gute Sitte, den beabsichtigten Typ eines Ausdrucks explizit anzugeben. Man schreibt den Typ hinter den Ausdruck und einen doppelten Doppelpunkt. Das erhöht die Lesbarkeit von Programmen und hilft, korrekte Programme zu schreiben. Sehr viele Programmierfehler entpuppen sich nämlich bereits als Typfehler. Haskell vergleicht den angegebenen intendierten Typ mit dem automatisch hergeleiteten und weist bereits an dieser Stelle unstimme Programme zurück.

Beispiel 3 `"Hallo"` hat den Typ `String`. Ist in Hugs die automatische Typangabe eingeschaltet, so wird dies nach Eingabe von `"Hallo"` angezeigt:

```
> "Hallo"
"Hallo" :: String
```

Man kann auch selbst angeben, dass der Ausdruck den Typ `String` hat. Der Ausdruck wird ausgewertet (in diesem Fall einfach wieder ausgegeben), und der Typ wird zur Bestätigung mit angezeigt:

```
> "Hallo" :: String
"Hallo" :: String
```

Gibt man einen falschen Typ an, wird ein Typfehler angezeigt:

```
> "Hallo" :: Int
ERROR - Type error in type annotation
*** Term      : "Hallo"
*** Type     : String
*** Does not match : Int
```

Die Fehlermeldung zeigt beides an: den falsch angegebenen Typ `Int` und den korrekten Typ `String`.

¹Beachte: In `ord x` ist `x` eine Variable, in `ord 'x'` ist `'x'` das Zeichen „kleiner Buchstabe x“.

3.1 Elementare Datentypen

In diesem Abschnitt werden die wichtigen in Haskell vordefinierten Datentypen vorgestellt. Hierzu werden die *Werte* und die *Operationen* genannt, die zu den Datentypen gehören.

3.1.1 Wahrheitswerte

Der Datentyp `Bool` enthält die Wahrheitswerte (Boolesche Werte) `True` (wahr) und `False`.²

Die grundlegenden booleschen Funktionen sind `&&` (und), `||` (oder) und `not` (nicht).

Bei der Auswertung von Ausdrücken, die mit `&&` und `||` zusammengesetzt sind, ist Haskell faul: `a && b` wird zum Wert `False` ausgewertet, sofern `a` bereits `False` liefert. Nur wenn `a` den Wert `True` hat, wird auch `b` ausgewertet, und das Ergebnis ist genau dann `True`, wenn `a` und `b` beide `True` sind.

Bei `a || b` müssen beide Operanden `a` und `b` den Wert `False` haben, damit das Ergebnis `False` ist. Entsprechend wird `b` gar nicht mehr ausgewertet, wenn `a` bereits `True` ist.

Der `not`-Operator bezieht sich nur auf einen Operanden: `not a` ist `True`, wenn `a` den Wert `False` hat und umgekehrt.

3.1.2 Zeichen und Zeichenketten

Der Datentyp `Char` (Character) enthält Zeichen entsprechend dem Unicode Standard. Die ASCII-Nummerierung ist im Unicode eingebettet. Wenn man die Erweiterungen des Unicode nicht benötigt, kann man daher so tun, als handle es sich um ASCII-codierte Zeichen.

Die meisten Zeichen, die auf einer Tastatur sind, kann man als Zeichenkonstanten in einfachen Hochkommas angeben: `'a'` ist der kleine Buchstabe `a`, `'5'` das Zeichen (nicht die Zahl) `5`. Für einige Zeichen braucht man eine besondere Darstellung mit dem Erweiterungssymbol `\`, sei es, weil es ein nicht-druckbares Zeichen ist (neue Zeile) oder weil das Symbol eine besondere Bedeutung in Haskell hat (Anführungszeichen):

neue Zeile	<code>'\n'</code>
Tabulator	<code>'\t'</code>
Backslash	<code>'\\'</code>
Einfaches Anführungszeichen	<code>'\''</code>
Doppelte Anführungszeichen	<code>'\"'</code>

Den ASCII-Code eines Zeichens erhält man mit der Funktion `ord`:

```
> ord 'a'
97 :: Int
```

Umgekehrt liefert die Funktion `chr` das Zeichen zu einer Nummer:

```
> chr 97
'a' :: Char
```

Zeichenketten haben den Typ `String` und werden in doppelte Anführungszeichen eingeschlossen:

```
"Dies ist eine Zeichenkette" :: String
```

²Haskell unterscheidet Groß- und Kleinschreibung. Es kommt hier also auf die großen Anfangsbuchstaben an.

Zu unterscheiden ist auch zwischen einzelnen Zeichen und Zeichenketten, die nur aus einem Zeichen bestehen: `'a' :: Char` und `"a" :: String`. Die leere Zeichenkette enthält gar kein Zeichen und wird `""` geschrieben.

`String` ist kein elementarer Datentyp, sondern ein von `Char` abgeleiteter Typ, der wegen seiner Wichtigkeit und wegen des Zusammenhangs mit Zeichen hier mit behandelt wird. Genau genommen ist ein `String` eine Liste von Zeichen, so dass `String` ein Synonym für `[Char]` ist. Die eckigen Klammern stehen für die Listenelementbildung³.

Für Zeichen und Zeichenketten stehen diese Operatoren zur Verfügung:

Op.	Beispiel	Ergebnis	Bedeutung
<code>:</code>	<code>'H':"allo"</code>	<code>"Hallo"</code>	Anfügen eines Zeichens <i>vor</i> einem String
<code>++</code>	<code>"Ha" ++ "llo"</code>	<code>"Hallo"</code>	Aneinanderhängen zweier Strings
<code>head</code>	<code>head "Hallo"</code>	<code>'H'</code>	Erstes Zeichen (Kopf) eines Strings
<code>tail</code>	<code>tail "Hallo"</code>	<code>"allo"</code>	String ohne das erste Zeichen (Rumpf)

3.1.3 Ganze Zahlen

Haskell kennt zweierlei Datentypen für die ganzen Zahlen. Der Datentyp `Int` enthält die ganzen Zahlen von `-2147483648` bis `2147483647` (was an der internen Darstellung mit 4 Byte liegt). Für viele Zwecke reicht dies aus, doch der Typ `Integer` bietet ganze Zahlen mit einer unbegrenzten Stellenzahl.

Wichtige Operationen für ganze Zahlen sind:

Op.	Beispiel	Ergebnis	Bedeutung
<code>+</code>	<code>5+3</code>	<code>8</code>	Addition zweier Zahlen
<code>*</code>	<code>5*3</code>	<code>15</code>	Multiplikation zweier Zahlen
<code>-</code>	<code>5-3</code>	<code>2</code>	Subtraktion zweier Zahlen
<code>-</code>	<code>-(-3)</code>	<code>3</code>	Umkehrung des Vorzeichens
<code>^</code>	<code>5^3</code>	<code>125</code>	Potenzieren zweier Zahlen
<code>div</code>	<code>div 5 3</code>	<code>1</code>	Ganzzahldivision
<code>mod</code>	<code>mod 5 3</code>	<code>2</code>	Divisionsrest
<code>abs</code>	<code>abs (-5)</code>	<code>5</code>	Betrag einer Zahl

Da der Quotient zweier ganzer Zahlen im Allgemeinen keine ganze Zahl ist, steht der Divisionsoperator `/` hier nicht zur Verfügung.

3.1.4 Gleitkommazahlen

Gleitkommazahlen in Haskell haben die Typen `Float` oder `Double` (für erhöhte Genauigkeit). Sie können als Dezimalzahlen geschrieben werden:

```
3.14159 -23.17 0.007 55 55.0
```

Für ganze Zahlen, die als Gleitkommazahlen verwendet werden sollen (im Beispiel `55`) ist dabei die Angabe der Nachkomma-Null optional. Gibt man sie an, so ist klar, dass es sich nicht um einen `Int`-Wert handeln kann, lässt man sie weg, so könnte es auch ein `Int` sein. Welchen Typ Haskell in solchen Situationen annimmt, hängt von den Operatoren ab, mit denen die Zahl verarbeitet wird.

Neben dieser Notation darf auch die sog. wissenschaftliche Darstellung verwendet werden, bei der — wie in Programmiersprachen üblich — das `e` die Bedeutung „ $\times 10$ -hoch“ hat.

```
3.14159e0 -2.317e1 7e-2 0.55e2
```

Für Gleitkommazahlen sind wie für ganze Zahlen die Operatoren `+`, `*`, `-` und `abs` definiert. Hinzu kommen die Gleitkommadivision `/` (z. B. `(5/3)` mit dem Ergebnis `1.66667`) und die Exponentiation `**` (z. B. `2**0.5` mit dem Ergebnis `1.41421`).

³Allgemeine Listen werden später besprochen.

Es gibt noch zahlreiche weitere Funktionen zur Arbeit mit Gleitkommazahlen wie auch mit anderen Datentypen. Hierzu mehr im nächsten Abschnitt.

3.2 Operatoren und Funktionen für einfache Typen

Alles hat seinen Typ, das gilt nicht nur für Werte, sondern auch für Berechnungsvorschriften, also Operatoren und Funktionen. In diesem Abschnitt erfahren Sie, wie diese Typen gebildet werden.

Doch zunächst zum Verhältnis zwischen Operatoren und Funktionen: Im Grunde genommen sind Operatoren nichts anderes als Funktionen, sie werden jedoch anders aufgeschrieben.

Beispiel 4 *Oben haben Sie `mod` zur Bildung des Restes bei der Division ganzer Zahlen kennen gelernt. In der oben angegebenen Form handelt es sich tatsächlich um eine Funktion, denn der Name `mod` muss vor den Argumenten angegeben werden, die verarbeitet werden:*

```
mod 5 3
```

ergibt den Rest 2 bei der Division von 5 durch 3.

Aus mathematischer Tradition heraus möchte man oft `mod` statt dessen zwischen die Argumente schreiben. Man nennt es dann einen Operator:

```
5 'mod' 3
```

Das Ergebnis ist natürlich wieder 2. Um aus dem Funktionssymbol `mod` ein Operationssymbol zu machen, wird es in Akzente eingeschlossen `'mod'`.

Indem man sie in Akzente einschließt (`'f'`)⁴, kann man auch andere Funktionssymbole `f`, die auf zwei Argumente angewendet werden, in Operationssymbole umwandeln. Umgekehrt wird aus einem Operationssymbol (z. B. `+`) ein Funktionssymbol, indem man es in runde Klammern einschließt. Die Addition von 5 und 3 kann man damit auch so schreiben:

```
(+) 5 3
```

Um der Lesbarkeit Willen empfiehlt es sich jedoch, die Symbole in der aus der Mathematik gewohnten Weise zu verwenden, also die Symbole für die Grundrechenarten und Vergleiche zwischen die Argumente zu schreiben, andere Funktionen aber davor.

3.2.1 Vergleichsoperatoren

In vielen Situationen muss man zwei Werte vergleichen. Hierzu gibt es diese Operatoren (die also *zwischen* die Operanden geschrieben werden):

```
== Test auf Gleichheit
/= Test auf Ungleichheit
< „kleiner als“
<= „kleiner als oder gleich“
> „größer als“
>= „größer als oder gleich“
```

Das Ergebnis dieser Operatoren hat stets den Typ `Bool`. Dies ist jedoch nicht der Typ des Operators selbst, denn `Bool` hat nur die beiden Werte `True` und `False`, und der Typ des Operators muss auch die Typen der benötigten Operanden angeben.

⁴Probieren Sie auf Ihrem Computer aus, welche Taste den benötigten Akzent liefert. Das kann je nach Tastatur verschieden sein.

Hinsichtlich der Operandentypen müssen wir zwischen `==` und `/=` und den anderen Operatoren unterscheiden. Die Operatoren, die „kleiner“ oder „größer“ enthalten, setzen nämlich voraus, dass die Operanden *angeordnet* werden können, so dass hinsichtlich der vorgesehenen Reihenfolge über „kleiner“ oder „größer“ entschieden werden kann. Bei den anderen beiden Operatoren ist eine Anordnung nicht erforderlich, es genügt die Möglichkeit, über die Gleichheit zu entscheiden.

Für beide Situationen stellt Haskell *Typklassen* bereit. Die Klasse `Eq` (Equality) ist die allgemeinere Klasse der Typen, deren Elemente man auf Gleichheit überprüfen kann. Die Klasse `Ord` ist die darin enthaltene Unterklasse der Typen, für deren Elemente man die Anordnung prüfen kann.

Die Standardtypen, die wir oben kennen gelernt haben, gehören `Ord` und damit auch `Eq` an, so dass wir hier keine Unterscheidung treffen müssen.

Als Beispiel betrachten wir daher den Operator `<`. Seinen Typ gibt das Hugs-System so an:

```
> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Dabei ist „>“ in der ersten Zeile der Hugs-Prompt (Eingabeaufforderung), danach kommt die Eingabe `:t` (sag mir den Typ von) `(<)` (der Funktion, die der Operator `<` bewirkt).

Die Ausgabe lesen wir so: „(<)“ Die Funktion, die der Operator `<` bewirkt, „::“ hat folgenden Typ: „`Ord a =>`“ Wenn `a` ein Ordnungstyp ist (d. h. zur Typklasse `Ord` gehört), dann hat `<` den Typ „`a -> a -> Bool`“ gib mir etwas vom Typ `a` und noch etwas vom (selben) Typ `a`, dann erhältst du ein Ergebnis vom Typ `Bool`.

In konkreten Anwendungen der Operatoren wird für den allgemeinen Typ `a` ein konkreter Typ substituiert: In `5 < 3` kann `<` der Typ `Int -> Int -> Bool` zugeordnet werden. Den Typ `String -> String -> Bool` erhält man dagegen für den Vergleich `"alpha"<"alphabetisch"`.

Zusammenfassend erhalten wir diese Typen der Vergleichsoperatoren:

```
==, /=      :: Eq a => a -> a -> Bool
<, <=, >, >= :: Ord a => a -> a -> Bool
```

3.2.2 Vordefinierte Funktionen

Zum Rechnen mit ganzen und Gleitkommazahlen gibt es eine ganze Reihe von Funktionen, die hier zusammen mit Beispielen für ihren Typ angegeben werden (d. h. es steht `Float`, wo auch `Double` stehen könnte etc.). Die genauen Typen kann man sich stets von Hugs anzeigen lassen, indem man „:t“ gefolgt vom Funktionsnamen eingibt.

Funktion	Typ	Beschreibung
<code>abs</code>	<code>Float -> Float</code>	Betrag einer Zahl
<code>ceiling,</code> <code>floor, round</code>	<code>Float -> Float</code>	Aufrunden (Abrunden, kaufmännisch Runden) zur nächsten ganzen Zahl
<code>negate</code>	<code>Float -> Float</code>	Vorzeichen umkehren
<code>signum</code>	<code>Float -> Float</code>	Vorzeichenfunktion (Werte -1 , 0 oder 1 !)
<code>sqrt</code>	<code>Float -> Float</code>	Quadratwurzel
<code>sin, cos, tan</code>	<code>Float -> Float</code>	Trigonometrische Funktionen
<code>asin, acos, atan</code>	<code>Float -> Float</code>	Umkehrfunktionen der trigonometrischen Funktionen
<code>pi</code>	<code>Float</code>	Die Konstante π
<code>exp</code>	<code>Float -> Float</code>	Exponentialfunktion zur Basis e
<code>log</code>	<code>Float -> Float</code>	Natürlicher Logarithmus
<code>logBase</code>	<code>Float -> Float -> Float</code>	Logarithmus zu Basis (erstes Argument)
<code>fromInt</code>	<code>Int -> Float</code>	Typumwandlung von <code>Int</code> nach <code>Float</code>
<code>read</code>	<code>String -> Float</code>	Wandelt eine Zeichenkette (z.B. "3.14") in passende
<code>show</code>	<code>Float -> String</code>	Umwandlung einer Zahl in eine Zeichenkette

Typen von Funktionen erkennt man stets am Pfeil `->`. Hat die Funktion mehr als ein Argument, so kommen im Typ auch mehrere Pfeile vor. So ist der Typ von `logBase` durch

```
logBase :: Float -> Float -> Float
```

gegeben. Informell kann man dies so lesen: Gib mir ein `Float` und noch ein `Float` (die ersten beiden `Floats`), dann ist das Ergebnis ein `Float`. Die Notation lässt sich aber auch genauer erklären. Bei der Angabe von Funktionstypen wird nämlich implizit nach rechts geklammert, so dass der Typ von `logBase` etwas ausführlicher ist:

```
logBase :: Float -> (Float -> Float)
```

Dies liest man so: Gib mir ein `Float`, dann gebe ich eine Funktion zurück, die den Typ `Float -> Float` hat. In der Tat,

```
logBase 2 :: Float -> Float
```

ist der Logarithmus zur Basis 2, und man muss nun noch ein `Float` angeben, ehe das Ergebnis, wieder ein `Float`, berechnet werden kann. Man kann die beiden `Floats` natürlich auch hintereinander angeben, um sofort das Ergebnis zu erhalten:

```
logBase 2 16
```

ergibt 4, da $2^4 = 16$ ist.

3.3 Übungen

Für die nachfolgenden Übungen benötigen Sie den Haskell-Interpreter „hugs“. Sie werden noch keine eigenen Haskell-Programme schreiben, daher sind alle Eingaben direkt in Hugs zu machen. Dort steht beispielsweise

```
Prelude>
```

was anzeigt, dass Hugs die Datei `Prelude.hs` mit den dort vordefinierten Funktionen geladen hat. An dieser Stelle können Sie direkt mit Eingaben an Hugs beginnen, die Sie mit der Eingabe-Taste (Return/Enter) abschließen.

Aufgabe 1: Diese Aufgabe dient dem Vertrautwerden mit Rechenoperationen für Zahlen. Geben Sie dazu in hugs die folgenden Ausdrücke ein und vergleichen Sie das Ergebnis mit Ihren Erwartungen. Mit dem Befehl

```
:s +t
```

schaltet man hugs in einen Modus, in dem außer den Werten auch ihre Typen angezeigt werden (mit `:s -t` schaltet man diesen Modus wieder aus). Verwenden Sie diesen Modus, um auch die Typen der Ausdrücke mit den von Ihnen erwarteten zu vergleichen.

```
4+9          3-5          -7*3          9/3          9 'div' 3
2.3+1.1      9.8-3.1        16*(-2.1)    7.1/2.4     9 'mod' 3
8 'div' 4    9 'div' 4      10 'div' 4   11 'div' 4   12 'div' 4
8 'mod' 4    9 'mod' 4      10 'mod' 4   11 'mod' 4   12 'mod' 4
3^2          3^3           pi           sin (pi/2)   exp 1
```

Aufgabe 2: In dieser Aufgabe geht es um Operationen, die sich nicht nur auf Zahlen beziehen. Gehen Sie für die nachfolgenden Ausdrücke genauso vor wie in Aufgabe 1:

```
'H'          "Hallo"        'H':"allo"    "Hallo"+"du"
head "Hallo"  tail "Hallo"   head (tail "hallo")
ord 'a'      ord 'b'        ord 'c'        ord 'z'
ord 'A'      ord 'Z'        chr 97         chr 98
5 == 7       4 == 4        True           not True
(5==7)&&(4==4) (5==7)|| (4==4) True && False False || True
```

Aufgabe 3: Beschreiben Sie die Wirkung der Haskell-Funktionen

```
div      mod      head      tail      ord      chr
not      &&       ||       ^         sin      exp
```

4 Funktionen als Programme

Dies macht die funktionale Programmierung aus: Der Programmtext besteht ausschließlich aus Funktionen, die Ausführung eines Programms ist ein Funktionsaufruf. Zumindest ist das im Kern so. Tatsächlich kommen zu den Funktionen noch Möglichkeiten hinzu, neue Datentypen zu definieren, Programme in mehrere Module (Dateien) aufzuteilen und Seiteneffekte (z. B. für die Ein- und Ausgabe) systematisch einzubauen. Durch diese Zugaben zur reinen Funktionsansammlung wird aber das Wesentliche des funktionalen Programmierens nicht verwässert: Funktionen sind Funktionen im mathematischen Sinn, d. h. sie stellen eine eindeutige Beziehung zwischen Ein- und Ausgabe her, und Funktionsaufrufe verändern nicht den Zustand des Computers, wie das bei „Funktionen“ in imperativen Programmiersprachen ohne weiteres der Fall sein kann. Es gilt also: Zu einer Eingabe gehört immer die selbe Ausgabe.

Das Versprechen, dass Funktionen in Haskell wirklich Funktionen im mathematischen Sinne sind, mag zunächst Zweifel an der Problemlösekraft dieser Art des Programmierens nähren. Denkt man nur an die üblichen Funktionen des Mathematikunterrichts, die Zahlen andere Zahlen zuordnen, so scheint die Welt der Programmierung hierdurch tatsächlich stark eingeschränkt. Zum Glück beschränkt sich die funktionale Programmierung jedoch nicht hierauf. Definitions- und Wertemengen von Funktionen können alle Datentypen sein, die Haskell bereitstellt und die man selbst zusätzlich definiert. So können Zeichenketten auch Zeichenketten zugeordnet werden, was eine wesentliche Beschäftigung in der Informatik ist. Und eines sollte wirklich beruhigen: Wenn sich Programme wie Funktionen verhalten, sollte es nicht mehr passieren, dass ein Programm sich bei verschiedenen Aufrufen mit denselben Daten unterschiedlich verhält.

4.1 Eine Eingabe — eine Ausgabe

Wir beginnen mit den einfachsten Funktionen, die zu einem Eingabewert einen Ausgabewert bestimmen, und wir beschränken uns zunächst auf die Datentypen, die im vorigen Kapitel vorgestellt wurden. Die Beispiele sind noch recht primitiv, aber sie zeigen schon einige wesentliche Eigenschaften von Haskell-Programmen.

Haskell kann in den meisten Fällen den Typ von Funktionen selbst bestimmen, aber es ist eine gute Sitte, den (beabsichtigten) Typ der Funktion explizit anzugeben. Wir sehen dies an einer Funktion, die eine Zahl verdoppelt:

```
verdoppeln :: Int -> Int
verdoppeln x = 2 * x
```

Die erste Zeile gibt den Namen `verdoppeln` der Funktion an. Den doppelten Doppelpunkt liest man als „hat den Typ“, danach kommt die Typangabe. In diesem Beispiel werden `Int`-Zahlen eingegeben (dafür steht das `Int` links des Pfeils) und `Int`s ausgegeben (`Int` rechts des Pfeils). Die nächste Zeile gibt die Funktionsvorschrift an: Wendet man `verdoppeln` auf einen Wert `x` an, so ist das Ergebnis `2 * x`.

Ist die Funktion in einer Textdatei geschrieben und wurde diese Datei in Hugs geladen, so kann man die Funktion benutzen:

```
> verdoppeln 5
10
```

Im Unterschied zur üblichen mathematischen Notation sei dies angemerkt: Es ist nicht nötig, das Argument der Funktion (d. h. ihre Eingabe) in Klammern einzuschließen (`verdoppeln (x)`). Andererseits darf man aber das Multiplikationszeichen nicht weglassen (wie in $f(x) = 2x$).

Die Funktion `verdoppeln` hätte man übrigens auch mit einem anderen Typ versehen können:

```
verdoppeln :: Float -> Float
verdoppeln x = 2 * x
```

Der Grund dafür ist, dass das Multiplikationszeichen `*` für alle Zahl-Datentypen definiert ist und die `2` auch zu jedem solchen Typ passt.

Es ist an dieser Stelle noch nicht nötig, den Umgang mit Typklassen aktiv zu beherrschen. Da Haskell in Meldungen zu Typfehlern aber auch Typklassen verwendet, sei für dieses Beispiel auch der allgemeinste zulässige Typ angegeben:

```
verdoppeln :: Num a => a -> a
verdoppeln x = 2 * x
```

Man liest die Typangabe so: Sofern `a` ein `Num`-Typ ist (`Num a =>`), kann man die Funktion `verdoppeln` mit dem Typ `a -> a` verwenden. Dabei ist `a` eine Typvariable, für die man konkrete Typen einsetzen kann.

Schließlich sei noch der Typunterschied zwischen `verdoppeln` und `verdoppeln x` hervorgehoben: `verdoppeln` ist eine Funktion und hat damit einen „Pfeiltyp“ wie `Int -> Int`. Dagegen ist in `verdoppeln x` (oder `verdoppeln 5`) schon ein Argument eingesetzt. Dieses kann man auswerten (im Falle von `x`) oder es steht schon direkt als Wert da (`5`). Damit lässt sich der Funktionswert bestimmen (`2 * x` bzw. `10`), so dass die Funktionsanwendung `verdoppeln x` den Ergebnistyp `Int` hat.

Im Folgenden werden weitere Beispiele für einfache Funktionen gegeben:

Die Haskellfunktion `kreisumfang` berechnet den Umfang U eines Kreises mit Radius r nach der Formel $U = 2 \cdot \pi \cdot r$. Dabei kommt die in Haskell eingebaute Konstante `pi` zum Einsatz:

```
kreisumfang :: Float -> Float
kreisumfang r = 2 * pi * r
```

Ein Beispiel für die Anwendung dieser Funktion ist

```
> kreisumfang 3
18.8496
```

Die Funktion `hallo` darf bei der Einführung in eine Programmiersprache nicht fehlen (normalerweise lässt man sie „Hello World“ ausgeben). Hier hat sie eine Zeichenkette zur Eingabe und ergänzt für die Ausgabe das Wort „Hallo“:

```
hallo :: String -> String
hallo text = "Hallo " ++ text
```

Gibt man seinen Namen als Eingabe an, so wird man von Haskell „begrüßt“:

```
> hallo "Hermann"
"Hallo Hermann"
```

Das Aneinanderhängen der beiden Zeichenketten (`"Hallo "` und die Eingabezeichenkette) geschieht durch den Konkatenationsoperator (Aneinanderhängeoperator) `++`. Beachten Sie das extra Leerzeichen innerhalb der Anführungszeichen bei `"Hallo "`, ohne das die Ausgabe `HalloHermann` gewesen wäre.

Fallunterscheidungen kommen in größeren Programmen häufig vor. Wir behandeln die verschiedenen Möglichkeiten ihrer Programmierung weiter unten ausführlich. Trotzdem seien hier schon zwei Möglichkeiten angegeben, um Eingabewerte zu untersuchen:

```
istGleichNull :: Int -> Bool
istGleichNull x = (x==0)
```

Diese Funktion testet von einem `Int`, ob es die Zahl 0 ist, und gibt den entsprechenden Wahrheitswert aus:

```
> istGleichNull 7
False

und

> istGleichNull 0
True
```

Bei der Programmierung muss man den Unterschied zwischen dem einfachen Gleichheitszeichen `=` (zur Definition von Funktionen) und dem doppelten Gleichheitszeichen `==` (zum Test auf Gleichheit zweier Werte) beachten. Da das `==` stärker bindet als das `=`, wären die Klammern in der Funktionsdefinition eigentlich nicht erforderlich. Sie schaden aber auch nichts und machen die Definition vielleicht besser lesbar.

Auf die explizite Nutzung des `==`-Operators verzichtet diese äquivalente Definition:

```
istGleichNull :: Int -> Bool
istGleichNull 0          = True
istGleichNull andereZahl = False
```

Hier wurde die Technik des Mustervergleichs („pattern matching“) verwendet, bei der von oben nach unten die Alternativen durchgegangen werden. Die erste Definition, die zum Funktionsargument passt, wird zur Auswertung herangezogen. Gibt man 0 ein, so passt die erste Zeile mit dem Ergebnis `True`. Für alle anderen Fälle passt diese Zeile nicht, und es wird die zweite Zeile verwendet, die sofort `False` liefert. Dabei ist die Variablenbezeichnung `andereZahl` für Haskell natürlich ohne Bedeutung (man hätte genauso gut `x` nehmen können), sie dient nur der besseren Lesbarkeit.

Als letztes Beispiel betrachten wir noch eine Funktion, die den zweiten Buchstaben einer Zeichenkette bestimmt. Wendet man sie auf kürzere Zeichenketten an, so wird sie einen Fehler verursachen, aber darum kümmern wir uns hier nicht.

```
zweiterBuchstabe :: String -> Char
zweiterBuchstabe text = head(tail text)
```

Hier kommen die beiden Funktionen `head` und `tail` zum Einsatz: `tail` bestimmt den Rest einer Zeichenkette nach Wegnahme des ersten Buchstabens. Im Ergebnis von `tail text` steht daher der zweite Buchstabe von `text` ganz vorne. Mit `head` wird der erste Buchstabe einer Zeichenkette bestimmt, hier also der erste Buchstabe des Restes von `text`. Probieren Sie die Funktion mit verschiedenen Eingaben (auch Zeichenketten der Länge 1) aus!

4.2 Mehrere Eingaben — eine Ausgabe

Um Funktionen mit mehreren Argumenten zu programmieren, gibt man die Argumente in der Funktionsdefinitionen einfach der Reihe nach an:

```
plus :: Int -> Int -> Int
plus x y = x+y
```

Die Funktion `plus` addiert ihre beiden Argumente. Interessant ist dabei der Typ der Funktion, der streng genommen

```
plus :: Int -> (Int -> Int)
```

ist. Es handelt sich also um eine Funktion, die ein (erstes) `Int`-Argument nimmt und damit eine Funktion vom Typ `Int -> Int` erzeugt. Tatsächlich hat die partielle Anwendung von `plus` auf nur ein Argument einen Funktionstyp:

```
> :t plus 3
plus 3 :: Int -> Int
```

Zur Auswertung der Funktion fehlt nämlich noch ein (zweites) `Int`-Argument, so dass immer noch eine Funktion in einer Variablen vorliegt. Die Angabe des zweiten Arguments führt schließlich zur Auswertung der Funktion.

```
> plus 3 5
8
```

Mathematiker kennen diese Interpretation mehrerer Funktionsargumente als Funktionen mit Parametern. Man schreibt hier üblicherweise $f_a(x) = a + x$ und hat das erste Argument als Parameter a deklariert.

In der gleichen Weise lassen sich Funktionen mit drei (und mehr) Argumenten schreiben:

```
dreiersumme :: Int -> Int -> Int -> Int
dreiersumme x y z = x+y+z
```

Auch hier ist der Typ eigentlich von rechts her geklammert

```
dreiersumme :: Int -> (Int -> (Int -> Int))
```

doch besteht in Haskell die Konvention, solche rechtsseitigen Klammern bei Typangaben weglassen zu dürfen. Man liest den Typ von `dreiersumme` dann als „Gib mir ein `Int`, ein `Int` und noch ein `Int`, dann ist das Ergebnis ein `Int`“, d. h. rechts des letzten Pfeils steht der Ergebnistyp, davor die Typen der Argumente.

Bei der Klammerung von Funktionstypen ist zwischen der Klammerung nach rechts (die man weglassen darf) und der nach links zu unterscheiden. Der Typ

```
(Int -> Int) -> Int
```

steht nämlich für eine Funktion, die eine Funktion (und nicht einfache Werte) mit Typ `Int -> Int` zur Eingabe hat und damit ein `Int` als Ausgabe produziert. Funktionen, die Funktionen als Argumente haben, werden in der Mathematik mitunter als Funktionale bezeichnet. Wir behandeln sie später unter der Überschrift „Funktionen höherer Ordnung“.

Eine andere Art, mehrere Argumente an eine Funktion zu übergeben, besteht in der Nutzung strukturierter Datentypen, nämlich Tupel und Listen. Dies behandeln wir weiter unten.

4.3 Je nachdem: Fallunterscheidungen

Funktionen mit Fallunterscheidungen sind aus der Mathematik auch als abschnittsweise definierte Funktionen bekannt. Dabei werden für verschiedene Eingaben unterschiedliche Berechnungsvorschriften verwendet. Der Typ des Ergebnisses muss dennoch in jedem Fall derselbe sein.

Zur Angabe solcher Funktionen sind in der Mathematik verschiedenen Notationen bekannt. Zum einen können einzelne Werte direkt mit Hilfe einer Wertetabelle angegeben werden.

x	$f(x)$
0	0
1	5
2	5
5	6,25

Das kann man in Haskell mit der Technik des Mustervergleichs auch machen:

```
f :: Float -> Float
f 0 = 0
f 1 = 5
f 2 = 5
f 5 = 6.25
```

Allerdings sind auf diese Weise nur wenige Werte definiert, für alle anderen Eingaben wird die Funktion eine Fehlermeldung liefern (undefinierter Wert...). Um ganze Abschnitte eines Zahlenabschnitts zu erfassen, kann man die Angabe von Bedingungen verwenden. Betrachten wir zunächst ein Beispiel in mathematischer Notation:

$$f(x) = \begin{cases} 5, & \text{falls } x < 4 \\ 1.25x, & \text{falls } x \geq 4 \end{cases}$$

Im Beispiel könnte es sich um eine Kostenfunktion mit Sockelbetrag handeln: Ein Artikel kostet das 1,25-fache seiner Menge x , mindestens aber 5 Geldeinheiten.

Da sich die große geschweifte Klammer nicht eintippen läßt, ist die Notation in Haskell etwas anders:

```
f :: Float -> Float
f x
  | x < 4 = 5
  | x >= 4 = 1.25 * x
```

Nach der Angabe von Funktions- und Argumentnamen (`f x`) kommen die Definitionen für die verschiedenen Fälle in separaten Zeilen, die mit einem senkrechten Strich beginnen. Danach folgt jeweils die Bedingung (sogenannter „Guard“) und nach dem Gleichheitszeichen die Funktionsdefinition für den jeweiligen Fall.

Bei Funktionsdefinitionen, die sich wie hier über mehrere Zeilen erstrecken, ist die Beachtung der „Abseitsregel“ wichtig: Alles, was zu einer Definition gehört, muss gegenüber dem Beginn der Definition nach rechts eingerückt sein. Die senkrechten Striche dürfen daher nicht ganz links stehen, sondern werden wenigstens um eine Position nach rechts gerückt. Folgt eine Zeile ohne Einrückung, so handelt es sich um eine neue Definition:

```
f :: Float -> Float
f x          --Definition von f
  | x < 4 = 5
  ...

g          --neue Definition,
          --da nicht eingerueckt
```

Mustervergleich und Definitionen mit Bedingungen lassen sich auch kombinieren. Außerdem kann man das Wort `otherwise` also Synonym für `True` benutzen. Man leitet damit einen abschließenden Fall ein, der verwendet werden soll, wenn keiner der vorherigen Fälle gepasst hat.

```
f :: Float -> Float
f 0 = 0
```

```
f x
| x<4      = 5
| otherwise = 1.25 * x
```

In der Regel kommt man mit diesen beiden Techniken aus, um abschnittsweise definierte Funktionen zu schreiben. Im nächsten Abschnitt werden wir die Ausdruckskraft alleine schon des Mustervergleichs in Zusammenhang mit rekursiver Programmierung sehen. Zuvor aber muss einer Frage vorgebeugt werden: Ja, es gibt auch ein `if-then-else`. Es spielt jedoch eine völlig andere Rolle als in imperativen Programmiersprachen, da es um die Auswahl einer Berechnungsvorschrift geht und nicht um die Ablaufkontrolle in einem imperativen Programm. Die Funktion `f` von oben lässt sich damit auch so programmieren:

```
f :: Float -> Float
f 0 = 0
f x = if (x<4) then 5 else 1.25 * x
```

Nach dem Wort `if` kommt also ein Boolescher Ausdruck, nach dem `then` die Berechnungsvorschrift, die anzuwenden ist, falls der Boolesche Ausdruck den Wert `True` hat. Nun *muss* noch ein `else` kommen, hinter dem die Berechnungsvorschrift steht, die andernfalls zu verwenden ist (und der Ergebnistyp muss derselbe sein wie bei `then`). Es ist nicht möglich im Sinne einer „einseitigen Auswahl“ nur einen `then`-Teil anzugeben. Die Funktionsdefinition muss ja in jedem Fall ein Ergebnis liefern, es ist nur möglich, zwischen verschiedenen Berechnungsvorschriften zu wählen.

Da die Funktionsdefinition mit Bedingungen (guards) übersichtlicher ist, erscheint die Benutzung von `if` in den meisten Fällen unnötig.

4.4 Immer wieder: Rekursion

Bei der rekursiven Definition einer Funktion wird die Funktion selbst in der Definition der Funktionsvorschrift benutzt. Wir beginnen mit einem Beispiel, in dem die Länge einer Zeichenkette bestimmt wird:

```
laenge :: String -> Int
laenge "" = 0
laenge text = 1 + (laenge (tail text))
```

Hier kommt der Funktionsname `laenge` nicht nur links der Gleichheitszeichen, sondern auch rechts davon vor. Dass dadurch eine funktionierende Definition zustande kommt, liegt daran, dass in der letzten Zeile die Funktion `laenge` rechts auf etwas „einfacheres“ als links angewandt wird. Wir betrachten die Auswertung von

```
> laenge "abc"
```

Die Zeichenkette `"abc"` entspricht nicht dem Muster `""`, also ist die zweite Zeile der Funktionsdefinition für die Auswertung zu benutzen:

```
... = 1+ (laenge (tail "abc"))
```

Die Funktion `tail` gibt den Rest einer Zeichenkette nach Entfernung des ersten Symbols zurück, so dass die weiteren Auswertungsschritte diese sind:

```
... = 1+ (laenge "bc")
    = 1+ (1+ (laenge (tail "bc")))
    = 1+ (1+ (laenge "c"))
    = 1+ (1+ (1+ (laenge (tail "c"))))
    = 1+ (1+ (1+ (laenge "")))
    = 1+ (1+ (1+ 0))
    = ... = 3
```

Man sieht, dass zur Berechnung der Länge einer Zeichenkette (auf der rechten Seite) immer wieder auf die Definition der Funktion zurückgegriffen wird. Dabei baut sich die Zeichenkette um jeweils ein Zeichen ab, bis nur noch die leere Kette "" übrig ist. Hierfür gibt die Funktionsdefinition direkt ein Ergebnis (0) an, so dass nun die Addition der Einsen stattfinden kann.

Das Schema bei der Definition von `laenge` ist typisch für rekursive Funktionsdefinitionen: Die Definition besteht aus einem Basisfall (hier für die Zeichenkette "", manchmal gibt es auch mehrere Basisfälle). Hinzu kommt der rekursive Fall (manchmal auch mehrere), bei dem die Berechnung des Funktionswertes für den Ausgangswert (z. B. "abc") erklärt wird unter der Annahme, dass man den Funktionswert für einen kleineren oder weniger komplexen Ausgangswert (z. B. "bc") bestimmen kann. Bei der Auswertung „hangelt“ sich die Funktion bis zum Basisfall hinunter, von wo aus der Funktionswert tatsächlich berechnet wird.

Als zweites Beispiel einer rekursiv definierten Funktion sei noch `summeBis` angegeben, die die natürlichen Zahlen von 1 bis zum angegebenen Argument addiert.⁵

```
summeBis :: Int -> Int
summeBis 0 = 0
summeBis n = n + summeBis (n-1)
```

Schließlich betrachten wir noch ein Beispiel, bei dem zwei Basisfälle benötigt werden. Die Fibonacci-Zahlen treten in Gesetzmäßigkeiten in der Natur an verschiedenen Stellen auf. Für uns ist das Bildungsgesetz dieser Zahlenfolge interessant: Die nullte und erste Fibonacci-Zahl sind jeweils 1. Danach erhält man immer eine weitere Fibonacci-Zahl, indem man die beiden vorangegangenen addiert. So ergibt sich die Zahlenfolge

$$1, 1, 2, 3, 5, 8, 13, 21, \dots$$

In Haskell lässt sich das leicht (wenn auch ineffizient) umsetzen durch

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Die Angabe zweier Basisfälle ist hier nötig, da in der rekursiven Zeile zwei aufeinanderfolgende vorangegangene Werte der Fibonacci-Zahlen benötigt werden (nämlich `fib (n-1)` und `fib (n-2)`). Ist `n` (von größeren Zahlen her kommend) auf 2 „geschrumpft“, so können beide Basisfälle zugleich genutzt werden, so dass die Rekursion abbricht.

4.5 Ansammlung von Zwischenergebnissen: Akkumulatoren

Beim Bestimmen der Länge einer Zeichenkette haben wir gesehen, dass das Ergebnis sich erst am Ende der Rekursion durch Auswertung von $1+(1+(1+\dots+0))$ aufbaut. Hätte man während der rekursiven Aufrufe schon einmal über die Anzahl der Einsen Buch geführt, so hätte das Ergebnis der Längenberechnung bereits beim Auffinden der leeren Zeichenkette vorgelegen.

Da die funktionale Programmierung die Einführung eines Speicherplatzes für die Verwaltung eines solchen Zwischenergebnisses nicht zulässt, schleußt man Zwischenergebnisse als zusätzliche Argumente durch Funktionen. Meist hat man dann eine Funktion mit einem zusätzlichen Argument, bei deren Berechnung die „eigentliche Arbeit“ getan wird, und eine Funktion, die dieses Argument nach außen verbirgt

⁵Mathematiker wissen, dass sich das Ergebnis multiplikativ als $\sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$ berechnen lässt.

und nach innen mit einem sinnvollen Startwert belegt. Wir betrachten das am Beispiel der Längenberechnung:

```
hlaenge :: String -> Int -> Int
hlaenge ""    acc = acc
hlaenge text acc = hlaenge (tail text) (acc + 1)

laenge :: String -> Int
laenge text = hlaenge text 0
```

Die Hilfsfunktion `hlaenge` hat außer einem `String` (dessen Länge bestimmt werden soll) noch ein `Int` als Argument. Dieses Argument steht für die bereits gezählten Symbole. Es ist hier seiner das Zwischenergebnis ansammelnden Rolle entsprechend mit `acc` für Akkumulator bezeichnet.

Man kann die Definition von `hlaenge` so lesen: Ist der Reststring leer (`""`) und wurden schon `acc` viele Zeichen gelesen, so hatte die ursprüngliche Zeichenkette `acc` Elemente (erste Zeile). Ansonsten berechnet sich die Länge der Zeichenkette aus dem, was `text` noch beizutragen hat und den schon gelesenen `acc` Zeichen. Dies ist genauso viel wie die Länge des um eins verkürzten `tail text` zuzüglich `acc + 1`.

Der Aufruf erfolgt mit der Funktion `laenge`, die ihrerseits `hlaenge` mit dem Akkumulatorwert `0` verwendet, denn am Anfang wurden noch keine Zeichen gelesen:

```
laenge "abc" = hlaenge "abc" 0
= hlaenge "bc" (0+1) = hlaenge "bc" 1
= hlaenge "c" (1+1) = hlaenge "c" 2
= hlaenge "" (2+1) = hlaenge "" 3
=3
```

Programmieren Sie zur Übung die Funktion `summeBis` mit Hilfe eines Akkumulators!

Einen Akkumulator können wir auch verwenden, um eine Zeichenkette umzudrehen (spiegeln):

```
spiegle :: String -> String
spiegle text = hspiegle text ""

hspiegle :: String -> String -> String
hspiegle "" acc = acc
hspiegle (x:xe) acc = hspiegle xe (x:acc)
```

In diesem Programm wurde ein Mustervergleich für Zeichenketten verwendet. Dabei passt `""` nur auf die leere Zeichenkette, der Ausdruck `x:xe` passt auf alle anderen Zeichenketten. Da der Doppelpunkt einzelne Zeichen vorne an Zeichenketten anfügt, muss der erste Buchstabe einer Zeichenkette dann mit `x` korrespondieren und der Rest der Kette mit `xe`. Somit hat `x` hier den Typ `Char`, während `xe` ein `String` ist.

In `hspiegle` stellt der Akkumulator den schon gespiegelten Teil der ursprünglichen Zeichenkette dar, wobei die Buchstaben von vorne weggenommen wurden. Um das Spiegeln fortzuführen, muss man den nächsten Buchstaben der verbliebenen Zeichenkette vorne an das Spiegelbild anfügen und mit dem so erhaltenen Rest der ursprünglichen Kette und dem weiter gebauten Spiegelbild entsprechend verfahren. Wir spiegeln `"abc"`:

```
spiegle "abc"
= hspiegle "abc" ""
= hspiegle "bc" ('a':"") = hspiegle "bc" "a"
```

```
= hspiegle "c" ('b':"a") = hspiegle "c" "ba"
= hspiegle "" ('c':"ba") = hspielge "" "cba"
= "cba"
```

Schließlich sei noch eine Funktion für Fibonacci-Zahlen angegeben, die gleich zwei Akkumulatoren benutzt. Probieren Sie die Funktion aus und erklären Sie, warum sie so sehr viel schneller als die Variante von oben arbeitet:

```
fib :: Int -> Int
fib n = hfib n 1 1

hfib :: Int -> Int -> Int -> Int
hfib 0 acc1 acc2 = acc1
hfib 1 acc1 acc2 = acc2
hfib n acc1 acc2 = hfib (n-1) acc2 (acc1 + acc2)
```

4.6 Übungen

In diesen Übungen werden Sie erste eigene Haskell-Programme schreiben. Es ist dazu zweckmäßig, den Haskell-Interpreter Hugs und gleichzeitig einen Editor geöffnet zu haben. In Windows ist das Wordpad gut geeignet, in Linux ist Emacs ein beliebter Editor. Sie können aber auch jeden anderen Editor Ihrer Wahl verwenden. Wichtig ist nur, dass Sie die Programme ohne zusätzliche Formatierung als reinen Text speichern. Es ist üblich, für Haskell-Programme die Dateinamenserweiterung `.hs` zu verwenden.

In den Aufgaben werden Funktionen vorkommen, die nicht für alle möglichen Eingaben vernünftig definiert sind und so zu Fehlern führen (z. B. ist es sinnlos, eine negative Anzahl von Sternchen ausgeben zu wollen). Kümmern Sie sich um diese Fälle nicht, die Behandlung von Ausnahmesituationen geschieht in einem späteren Kapitel.

4.6.1 Einfache Funktionen

Aufgabe 1: Schreiben Sie zum Eingewöhnen im Editor die Funktion

- `verdoppeln` mit dem Typ `Int -> Int`

aus dem Lehrtext.

Speichern Sie die Datei unter einem Namen Ihrer Wahl ab (z. B. `uebungen.hs`). Gehen Sie dann mit der Maus in Ihr Hugs-Fenster und laden Sie mit dem Befehl `:l uebungen.hs` (bzw. mit Angabe Ihres Dateinamens) Ihr Programm. In Windows können Sie bei Verwendung von `winhugs` die Datei auch über eine Menüauswahl laden. Probieren Sie das Programm dann mit verschiedenen Eingaben aus.

Die weiteren Funktionen können Sie in derselben Datei programmieren, in der das obige Beispiel steht. Um die jeweils neueste gespeicherte Version der Datei in Hugs nutzen zu können, genügt es, in Hugs `:r` (für `reload`) einzugeben.

Kleine Warnung: Im Lehrtext wurden teilweise verschiedene Versionen der selben Funktion unter jeweils dem selben Namen programmiert. Das geht beim Programmieren am Computer nicht. Alle Funktionen müssen verschiedene Namen haben, und dies dürfen auch keine Namen vordefinierter Funktionen sein. Sie können aber für verschiedene Varianten ähnliche Namen verwenden (z. B. `laenge1` und `laenge2` für Funktionen zur Bestimmung der Länge einer Zeichenkette, die verschiedene Techniken verwenden).

Programmieren Sie nun ...

- eine Funktion `plus5` mit dem Typ `Float -> Float`

- eine Funktion `zum_quadrat` mit dem Typ `Int -> Int`
- eine Funktion `ist_gleich_7` mit dem Typ `Int -> Bool`
- eine Funktion `groesser_null` mit dem Typ `Int -> Bool`
- eine Funktion `erster_buchstabe` mit dem Typ `String -> Char`
- eine Funktion `dritter_buchstabe` mit dem Typ `String -> Char` (den Fehlerfall, dass die Funktion auf zu kurze Texte angewendet wird, müssen Sie nicht berücksichtigen.)

Aufgabe 2: In dieser Aufgabe sollen Sie Ihr Verständnis der Haskell-Typangaben überprüfen.

Bestimmen Sie dazu (zunächst ohne Computerhilfe) die Typen der folgenden Haskell-Ausdrücke und Haskell-Funktionen. Sofern es mehrere gültige Typen gibt, genügt die Angabe eines dieser Typen.

```
5 'div' 2
tail "Informatik"
(5 == 3)
(5 == 3) || (2 < 4)
dritter text = head (tail (tail (text)))
kleiner_drei zahl = (zahl < 3)
plus_sieben zahl = zahl + 7
```

4.6.2 Funktionen mit mehreren Argumenten

Aufgabe 3: Programmieren Sie eine Funktion `zeitaddieren`, die die Summe zweier Stundenangaben (als `Ints`) bezüglich einer 24-Stunden-Uhr bildet (z. B. $5 + 7 = 12$ und $15 + 11 = 2$).

Aufgabe 4: Programmieren Sie eine Funktion `maximum`, die die größte zweier eingegebener Zahlen bestimmt. Verwenden Sie diese Funktion dann zur Definition einer Funktion `maxvonvier`, die die größte von vier eingegebenen Zahlen bestimmt.

Aufgabe 5: Programmieren Sie eine Funktion `exkl_oder` mit dem Datentyp `Bool -> Bool -> Bool`, die das Ergebnis `True` hat, falls die beiden Eingaben verschiedene Wahrheitswerte haben. Ansonsten soll die Funktion den Wert `False` haben. (Diese Funktion nennt man „Exklusiv-Oder“.)

4.6.3 Mustervergleich und Bedingungen

Aufgabe 6: Programmieren Sie eine Funktion `istgleichdrei`, die feststellt, ob eine eingegebene Zahl die Zahl 3 ist. Benutzen Sie dazu die Technik des Mustervergleichs.

Aufgabe 7: Programmieren Sie die Funktion aus Aufgabe 5 mit der Technik des Mustervergleichs.

Aufgabe 8: Schreiben Sie eine Funktion `istvokal`, die von einem Kleinbuchstaben feststellt, ob er ein Vokal ist. Verwenden Sie dazu die Technik des Mustervergleichs.

Aufgabe 9: Schreiben Sie eine Funktion `my_not`, die dasselbe macht wie die eingebaute Funktion `not`, ohne dass Sie dabei die Funktion `not` verwenden.

Aufgabe 10: Programmieren Sie eine Funktion `istBuchstabe`, die von einem `Char` ermittelt, ob es ein Buchstabe ist (von a bis z und A bis Z, ohne deutsche Sonderzeichen).

4.6.4 Rekursive Funktionsdefinitionen

Aufgabe 11: Schreiben Sie eine Funktion `summevon7bis`, die für natürliche Zahlen n ab 7 die Summe $7 + 8 + \dots + n$ bestimmt.

Aufgabe 12: Schreiben Sie eine Funktion `a_test`, die testet, ob eine eingegebene Zeichenkette den Buchstaben `a` enthält.

Aufgabe 13: Programmieren Sie eine Funktion `sternchen`, die ein `Int` einliest und entsprechend viele Sterne ausgibt. Beispiel:

```
> sternchen 5
"*****" :: String
```

4.6.5 Akkumulatoren

Aufgabe 14: Programmieren Sie die Funktion `summevon7bis` aus Aufgabe 11 mit Hilfe eines Akkumulators.

Aufgabe 15: Programmieren Sie eine Funktion `mal`, die das Produkt von zwei positiven `Ints` bildet und dazu nur Additionen und Subtraktionen verwendet (d. h. das Produkt wird durch wiederholte Addition berechnet, z. B. $3 \cdot 5 = 5 + 5 + 5$).

Programmieren Sie diese Funktion mit und ohne Verwendung eines Akkumulators.

4.6.6 Testaufgabe

Aufgabe 16: Gegeben seien die folgenden Funktionen:

```
groesser_test a b = (a > b)
```

```
produkt a b = a * b
```

```
oder False False = False
oder a      b      = True
```

```
quadrate 0 = 0
```

```
quadrate n = (2*n - 1) + (quadrate (n-1))
```

a. Bestimmen Sie ohne Computerhilfe die Ergebnisse für folgende Aufrufe:

```
groesser_test 8 3
groesser_test 3 5
produkt 9 (-2)
oder True False
quadrate 0
quadrate 1
quadrate 4
```

b. Bestimmen Sie ohne Computerhilfe Datentypen für die oben aufgeführten Funktionen.

5 Datenstrukturen

Die Möglichkeit, aus bestehenden Typen neue konstruieren zu können, trägt zur Ausdrucksstärke einer Programmiersprache bei. Haskell bietet (neben dem Bilden von Funktionstypen) zwei Standardkonstruktionen: Listen- und Tupelbildung. Dabei können die Typkonstruktionen beliebig verschachtelt sein: Listen von Listen, Listen von Tupeln, Listen von Tupeln von Listen, Listen von Funktionen,

5.1 Listen

5.1.1 Listendarstellung

Listen erlauben die Zusammenfassung beliebig vieler Objekte eines gemeinsamen Datentyps zu einem neuen strukturierten Objekt. Dass Listentypen und Funktionen zur Listenverarbeitung zum Haskell-Sprachumfang gehören (während sie in anderen Programmiersprachen erst mühsam definiert werden müssen), erleichtert viele Programmieraufgaben, da „massenhaft“ zu bearbeitende Daten oft in Listenform vorliegen.

In Haskell müssen alle Elemente einer Liste den selben Datentyp haben. So kann man von einer `Int`-Liste sprechen, von `Bool`-Listen oder von Listen, die `Int`-Listen enthalten. Den Datentyp gibt man immer an, indem man den Typ der Elemente in eckige Klammern einschließt:

<code>[Int]</code>	eine <code>Int</code> -Liste
<code>[Float]</code>	eine <code>Float</code> -Liste
<code>[Char]</code>	eine <code>Char</code> -Liste
	synonym: <code>[Char]=String</code>
<code>[[Bool]]</code>	eine Liste von <code>Bool</code> -Listen
<code>[Int->Int]</code>	eine Liste von <code>Int->Int</code> -Funktionen
<code>[a]</code>	eine <code>a</code> -Liste (mit <code>a</code> als Typvariable)

Listen kann man aufschreiben, indem man ihre Elemente innerhalb eckiger Klammern durch Kommas getrennt angibt:

```
[1,4,2,3]      :: [Int]
[True]         :: [Bool]
[[1,2], [5,3,2], []] :: [[Int]]
```

Listen sind keine Mengen im mathematischen Sinne, es kommt also auf die Reihenfolge und Anzahl der Vorkommnisse von Elementen an. Die folgenden Listen sind alle verschieden:

```
[1,4,2,3]      :: [Int]
[4,3,2,1]      :: [Int]
[1,4,4,2,3,3]  :: [Int]
```

Die leere Liste stellt hinsichtlich ihres Typs einen besonderen Fall dar: Sie kann jeden Listentyp annehmen. Welchen Typ sie im Einzelfall hat, hängt vom Zusammenhang ihrer Verwendung ab:

Beispiel	Typ der leeren Liste
<code>[[True, False], []]</code>	<code>[Bool]</code>
<code>[[1,2], [], [3,5]]</code>	<code>[Int]</code>
<code>[[[1,2], [3]], []]</code>	<code>[[Int]]</code>
<code>['h', 'a', '1', '1', 'o'], []]</code>	<code>[Char]</code>

Wegen der besonderen Bedeutung von Zeichenketten gibt es für `[Char]` das Typsynonym `String`. Zeichenketten können in doppelten Anführungszeichen und ohne trennende Kommas geschrieben werden, die folgenden beiden Listen sind daher gleich:

```
['h','a','l','l','o'] :: String
"hallo"             :: String
```

Die leere String-Liste hat die besondere Schreibweise `""`, man darf aber auch die allgemeine Schreibweise leerer Listen verwenden: `[] :: String`.

Für Zahlenlisten gibt es eine spezielle Schreibweise zur Aufzählung der Zahlen eines Intervalls: `[n..m]` ist die Liste der Zahlen von `n` bis `m` mit Einer-Abständen:

```
[1..10] = [1,2,3,4,5,6,7,8,9,10]
[0.5 .. 4] = [0.5, 1.5, 2.5, 3.5]
```

Mit der Notation `[n,p .. m]` erhält man die Zahlen von `n` bis `m` mit dem Abstand `p-n`. Passt die hintere Grenze `m` nicht ganz in dieses Muster, so ist die letzte Zahl der Liste diejenige, die dem Muster entspricht und noch ganz zwischen `n` und `m` liegt. Ist `n` größer als `m`, so bleibt beim Aufwärtszählen die Liste leer. Entsprechendes gilt umgekehrt für das Abwärtszählen.

```
[1,3 .. 9] = [1,3,5,7,9]
[1,3 .. 10] = [1,3,5,7,9]
[0,0.2 .. 1] = [0, 0.2, 0.4, 0.6, 0.8, 1]
[5, 4.3 .. 2] = [5, 4.3, 3.6, 2.9, 2.2]
[5 .. 2] = []
```

5.1.2 Listenkonstruktion und -bearbeitung

Zur Angabe konkreter (nicht zu langer) Listen wird man oft die aufzählende Schreibweise in eckigen Klammern verwenden. Zur Listenbearbeitung muss man fast immer auf den strukturellen Aufbau der Listen zurück greifen.

Wir haben das schon bei der Definition der Funktion `laenge` für Zeichenketten getan:

```
laenge :: String -> Int
laenge "" = 0
laenge text = 1 + (laenge (tail text))
```

Da die Funktion keine speziellen Eigenschaften von Zeichenketten verwendet, können wir sie zur Verwendung mit beliebigen Listentypen anpassen:

```
laenge :: [a] -> Int
laenge [] = 0
laenge liste = 1 + (laenge (tail liste))
```

Die Funktion verwendet das allgemeine Muster der Listenbearbeitung, das dem Aufbau von Listen entspricht: Eine Liste ist entweder leer oder sie besteht aus einem Element, das (vorne) an eine Liste angehängt ist. Den ersten Fall behandelt die Zeile `laenge [] ...`, den zweiten Fall die Zeile `laenge liste ...`. Besteht im zweiten Fall die Liste aus einem Element `x`, das an die Liste `xe` angefügt wurde, so ist `tail liste` diese Liste `xe`. (Hätte man das erste Element `x` benötigt, so hätte man mit `head liste` darauf zugreifen können.)

Mit den Konstruktoren `[]` für die leere Liste und `:` (Doppelpunkt) für das Anfügen eines Elements vor eine Liste wird das noch deutlicher:

```

laenge :: [a] -> Int
laenge []      = 0
laenge (x:xe) = 1 + (laenge xe)

```

Hier greift der Mustervergleich in der Funktionsdefinition die allgemeine Definition von Listen als leere oder zusammengesetzte Listen auf. Ist die Liste zusammengesetzt, so ist das stets durch Anfügen eines Elements x an eine Liste xe geschehen. Der Mustervergleich ordnet daher das erste Element der gesamten Liste der Variablen x zu, die restliche Liste der Variablen xe , so dass über diese Variablen auf die Komponenten der Liste zugegriffen werden kann.

Das Beispiel der Funktion `laenge` enthält alle wichtigen Einzelheiten zur Listenkonstruktion. Diese werden nun in einer allgemeinen Definition herausgestellt:

Definition: Ist a ein Haskell-Typ, so ist $[a]$ der Typ einer *Liste* von Elementen des Typs a . Diese Listen sind folgendermaßen aufgebaut:

1. Die leere Liste $[]$ ist eine Liste vom Typ $[a]$;
2. Ist x ein Element vom Typ a und xe eine Liste vom Typ $[a]$, so ist $x:xe$ eine Liste vom Typ $[a]$.

In einer nach dem zweiten Punkt zusammengesetzten Liste nennt man das erste Element x den *Listenkopf*, die Restliste xe den *Listenrest* oder *Listenrumpf*.

Listen sind in Haskell also rekursiv definiert. Wie die Definition zu verstehen ist, sieht man am besten an einem Beispiel: Dazu betrachten wir, wie man mit Hilfe dieser Definition nachweist, dass $[4,3,2]$ eine Liste (von `Ints`) ist:

Da es sich bei den Zahlen 4, 3 und 2 um `Ints` handelt, müssen wir schauen, warum $[4,3,2]$ eine *Liste* ist. Der Typ ist dann natürlich `[Int]`.

Da $[4,3,2]$ nicht die leere Liste ist, muss es aufgrund des zweiten (also des rekursiven) Falles der Definition eine Liste sein. Dann muss 4 die Rolle des x und $[3,2]$ die Rolle der xe spielen:

```
[4,3,2] = 4 : [3,2]
```

Dazu muss $[3,2]$ eine Liste sein, wozu wieder der rekursive Fall der Definition herangezogen wird. Die Rolle von x spielt 3, die Rolle der xe spielt $[2]$ (und nicht 2):

```
[3,2] = 3 : [2]
```

Nun ist zu begründen, dass $[2]$ eine Liste ist. Wir nehmen 2 für x und die leere Liste $[]$ für xe :

```
[2] = 2 : []
```

Von der leeren Liste $[]$ schließlich wissen wir aus dem ersten Fall der Definition, dass es sich um eine Liste handelt. So können wir die Bausteine jetzt wieder zusammensetzen und sehen, wie die Liste $[4,3,2]$ der Definition entsprechend aufgebaut ist:

```
[4,3,2] = 4 : (3 : (2 : []))
```

Auch wenn es umständlich erscheint, lassen sich alle Listen mit `[]` und `:` konstruieren:

```

7:[] = [7]
'h':'a':'l':'l':'o':[] = "hallo"

```

Die Klammerung der Zusammensetzungen mit dem Doppelpunkt erfolgt dabei von rechts her und darf weggelassen werden.

Mit Listen und ihrer Bearbeitung kommt eine wichtige Eigenschaft von Haskell ins Spiel: Haskell erlaubt die Definition *polymorpher Funktionen*. Das sind Funktionen, die für unterschiedliche Datentypen funktionieren (und dort im Prinzip unabhängig vom Typ dasselbe tun⁶). Im Zusammenhang mit Listen kann man alle Funktionen polymorph definieren, die sich nur auf die Listenstruktur beziehen, die aber nicht mit den Elementen der Liste rechnen müssen. Beispiele sind: Bestimmen der Länge einer Liste, Umdrehen der Elementreihenfolge („Spiegeln“), Listen aneinanderhängen.

Um Typen polymorpher Funktionen anzugeben, verwendet man Typvariable. Hier ist jeder kleine Buchstabe erlaubt, wir verwenden kleine Buchstaben vom Anfang des Alphabets. Die Listenkonstruktoren `[]` und `:` sind selbst polymorphe Operatoren: Die leere Liste kann jeden Typ haben

```
[] :: a
```

und der Doppelpunkt hat den Typ

```
(:) :: a -> [a] -> [a]
```

d. h. er hat ein Element beliebigen Typs `a` und eine im Typ passende Liste als Argumente und erzeugt daraus eine `a`-Liste, nämlich diejenige, in der das Element vorne an die Liste angefügt wurde.

So wie die Funktion `laenge` nutzen Funktionen zur Listenbearbeitung meist die spezielle Konstruktionsweise von Listen aus. Sie werden rekursiv programmiert, wobei der Basisfall die Anwendung der Funktion auf die leere Liste ist. Im rekursiven Fall wird man die Funktion unter Verwendung des Ergebnisses für die um ein Element kürzere Liste definieren. Wir zeigen dies anhand einiger Beispiele, die in unterschiedlicher Weise Polymorphismus ausnutzen.

Für Zahlentypen lässt sich die Summe der Listenelemente so bestimmen:

```
summe :: Num a => [a] -> a
summe []      = 0
summe (x:xe) = x + (summe xe)
```

Die Summation verwendet die Eigenschaft der Listenelemente, addiert werden zu können. Insofern kann man `summe` nicht gänzlich polymorph mit dem Typ `[a]->a` programmieren. Statt dessen muss man für die Typvariable eine Einschränkung machen, dass `a` nämlich ein Nummerentyp ist. Dann ist die Summe der Elemente der leeren Liste gleich 0, während für andere Listen die Summe durch Addition des ersten Elements `x` und der Listensumme der Restliste bestimmt wird.

Zwei Listen mit gleichem Typ kann man aneinanderhängen:

```
aneinanderhaengen :: [a] -> [a] -> [a]
aneinanderhaengen [] liste2      = liste2
aneinanderhaengen (x:xe) liste2 = x:(aneinanderhaengen xe liste2)
```

Der Mustervergleich bezieht sich auf die vordere der beiden Listen. Ist diese leer, so ist das Ergebnis des Aneinanderhängens einfach die zweite Liste. Sonst hat die erste Liste die Gestalt `(x:xe)`, und man hängt beide Listen aneinander, indem man `xe` und `liste2` aneinanderhängt und davor das Element `x` anfügt. Die Funktion ist polymorph ohne Einschränkung an die Typvariable `a`, da es ausschließlich auf die Listenstruktur ankommt.

⁶In manchen Programmiersprachen kann man dasselbe Funktionssymbol je nach Datentyp für völlig verschiedene Aufgaben verwenden. Man spricht dann nicht von Polymorphie, sondern vom *Overloading* des Funktionssymbols.

Da das Aneinanderhängen von Listen öfters benötigt wird, gehört diese Funktion in Form des Operators `++` zum Sprachumfang von Haskell:

```
["erste", "Liste"] ++ ["zweite", "Liste"]
= ["erste", "Liste", "zweite", "Liste"]
```

Da `:` und `++` ähnliche (aber verschiedene) Dinge tun, sei auf den Unterschied zwischen beiden Operatoren hingewiesen. Der Doppelpunkt verbindet ein *Element* mit einer *Liste*, das doppelte Plus dagegen *zwei Listen*. Zur Listenkonstruktion und vor allem zum Mustervergleich mit Listen verwendet man stets den Doppelpunkt. Das doppelte Plus wird zum Aneinanderhängen zweier Listen benutzt, es kann aber nicht bei einem Mustervergleich verwendet werden. Im „Mustervergleich“ von `[1,2]` mit `liste1 ++ liste2` wäre nämlich nicht klar, wie die Liste `[1,2]` auf `liste1` und `liste2` aufzuteilen wäre. Die Aufstellung zeigt die drei verschiedenen Möglichkeiten⁷:

```
[] ++ [1,2] = [1,2]
[1] ++ [2]  = [1,2]
[1,2] ++ [] = [1,2]
```

Zum Abschluß programmieren wir noch eine Sortierfunktion. Dies geschieht in zwei Schritten. Als erstes brauchen wir eine Funktion, die ein Element in eine Liste passenden Typs einfügt. Wir gehen davon aus, dass die Liste schon sortiert vorliegt und das neue Element an die „passende“ Stelle eingefügt werden soll. Die betrachteten Elemente müssen mit `<` verglichen werden können, also muss `a` ein Ordnungstyp sein.

```
einfuegen :: Ord a => a -> [a] -> [a]
einfuegen y []      = [y]
einfuegen y (x:xe)
  | y<x             = y:x:xe
                    --y ist das kleinste Element
  | otherwise       = x:(einfuegen y xe)
                    --x ist das kleinste Element,
                    --y muss in den Rest eingefuegt werden
```

Will man eine komplette Liste sortieren, so gewinnt man mit Hilfe von `einfuegen` das Sortierverfahren „Sortieren durch Einfügen“ (insertion sort), indem man sukzessive die Liste der unsortierten Liste in eine zu Beginn leere Liste einfügt. Dies geschieht in der Hilfsfunktion `hlistiere`, die einen Akkumulator verwendet. Der Aufruf erfolgt durch die Hauptfunktion `sortiere`.

```
sortiere :: Ord a => [a] -> [a]
sortiere liste = hsortiere liste []

hsortiere :: Ord a => [a] -> [a] -> [a]
hsortiere [] zielliste = zielliste
hsortiere (x:xe) zielliste = hsortiere xe (einfuegen x zielliste)
```

Die Funktionsweise des Sortierens durch Einfügen zeigen wir an einem Beispiel:

```
sortiere [5,1,3]
= hsortiere [5,1,3] []
= hsortiere [1,3] (einfuegen 5 [])
= hsortiere [1,3] [5]
```

⁷Die Sprache Prolog erlaubt solche Mustervergleiche, indem in der Berechnung ggf. alle möglichen Zerlegungen in separaten Rechensträngen verfolgt werden.

```

= hsortiere [3] (einfuegen 1 [5])
= hsortiere [3] [1,5]
= hsortiere [] (einfuegen 3 [1,5])
= hsortiere [] [1,3,5]
= [1,3,5]

```

5.2 Tupel

Während Listen eine beliebige Anzahl von Elemente des selben Typs enthalten können, dienen Tupel der Zusammenfassung einer festen Anzahl von Elementen, die jedoch unterschiedliche Typen haben dürfen. Tupel ist der Oberbegriff für Aggregationen verschiedener Größe: Tupel mit zwei Elementen sind als Paare bekannt, Tripel haben drei Elemente, Quadrupel vier. So kann man aus lateinischen Zahlwörtern weitere Namen für Tupel bilden, man spricht aber auch vom n -Tupel, wenn das Tupel n Elemente hat.

Tupel werden in Haskell in runde Klammern geschrieben, die Elemente werden durch Kommas getrennt. Hier sind einige Beispiele zusammen mit ihren Typangaben:

```

(1, 2) :: (Int, Int)
('a', 'b') :: (Char, Char)
(1, 'b') :: (Int, Char)
(5, "Birnen") :: (Int, String)
([1,2,3], 3.14, 'a') :: ([Int], Float, Char)
(True, 7, [False]) :: (Bool, Int, [Bool])
(('a', "b"), 95) :: ((Char, String), Int)

```

Der Typ eines Tupels wird also in runden Klammern angegeben, in denen für jede Position innerhalb des Tupels der Typ des Elements angegeben ist.

Tupel kommen vor allem dann zur Anwendung, wenn die Zusammengehörigkeit einzelner Datenstücke ausgedrückt werden soll. So besteht der Name einer Person aus Vor- und Zuname, was gut mit einem Paar von Strings erfasst werden kann:

```
("Hermann", "Puhlmann") :: (String, String)
```

In der Mathematik werden Punkte der Ebene durch Zahlenpaare beschrieben:

```
(3.9, 2.7) :: (Float, Float)
```

Untersucht man die Häufigkeit, mit der einzelne Buchstaben in einem Text vorkommen, so sind Paare praktisch, die Buchstaben und Zahlen (die Häufigkeit) zusammen führen:

```
('a', 24) :: (Char, Int)
```

Da Paare, also Tupel mit zwei Komponenten, besonders häufig vorkommen, gibt es für sie die Funktionen `fst` (first) und `snd` (second), die als Ergebnis die erste bzw. zweite Komponente eines Paares haben. Beides sind polymorphe Funktionen, deren Typ zwei verschiedene Typvariable enthält:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

Der Grund für die zweierlei Typvariablen ist, dass die Komponenten des Paares verschiedene Typen haben dürfen. Jede der Typvariablen steht für einen der Typen, und bei `fst` hat das Ergebnis denselben Typ wie die erste Komponente, bei `snd` denselben Typ wie die zweite Komponente. In konkreten Anwendungen können dennoch beide Typen übereinstimmen, dann werden `a` und `b` eben mit demselben Typ belegt, etwa in diesem Beispiel, in dem beide den Typ `Int` haben können:

```
> fst (5, 7)
5
```

Mustervergleich bietet eine andere gute Möglichkeit, auf die Komponenten eines Tupels zuzugreifen. Wir programmieren zur Illustration die Funktionen `fst` und `snd` selbst⁸:

```
fst :: (a, b) -> a
fst (x, y) = x
```

```
snd :: (a, b) -> b
snd (x, y) = y
```

Als Beispiel zur Programmierung mit Paaren schreiben wir nun eine Funktion zur Bestimmung der Länge eines Streckenzuges. Wir nehmen an, der Streckenzug ist als Liste von `(Float, Float)`-Paaren gegeben. Die Liste

```
[(1.3, 2.4), (2.3, 3.4), (1.5, 4.8)]
```

stellt also einen Streckenzug aus zwei Teilstücken dar: Die erste Strecke geht vom Punkt (1.3,2.4) zum Punkt (2.3,3.4). Die zweite Strecke geht von (2.3,3.4) nach (1.5,4.8). Um die Gesamtlänge eines solchen Streckenzuges zu bestimmen, müssen die Längen der Teilstücke bestimmt und anschließend aufsummiert werden.

Als erstes schreiben wir eine Funktion, die für eine durch zwei Endpunkte gegebene Strecke die Länge berechnet. Dabei kommt der Satz des Pythagoras zum Einsatz:

```
streckenlaenge :: (Float, Float) -> (Float, Float) -> Float
streckenlaenge (x1,y1) (x2,y2) = sqrt ((x2-x1)^2 + (y2-y1)^2)
```

Diese Funktion ist jetzt auf die Liste der Punkte des Streckenzuges anzuwenden. Dabei ergibt sich aus jeweils zwei aufeinanderfolgenden Punkten eine Streckenlänge. Diese Streckenlängen bilden eine neue Liste (die ein Element weniger als die Liste der Punkte hat).

```
laengenliste :: [(Float, Float)] -> [Float]
laengenliste []
    = []
laengenliste (p:[])
    = [] --nur ein Punkt ist keine Strecke
laengenliste (p1:p2:punkte)
    = (streckenlaenge p1 p2):(laengenliste (p2:punkte))
```

Schließlich sind die Streckenlängen aufzusummieren. Dafür haben wir oben schon die Funktion `summe` geschrieben, die wir hier wieder benutzen. Insgesamt erhalten wir die Funktion

```
streckenzuglaenge :: [(Float, Float)] -> Float
streckenzuglaenge liste = summe (laengenliste liste)
```

5.2.1 Currying und uncurrying

Bei Funktionen mit mehreren Argumenten haben wir die Argumente bisher einfach nach dem Funktionsnamen angegeben — ohne Klammern und ohne Komma. Auf diese Weise ist es möglich, eine Funktion erst mit einem Teil ihrer Argumente zu „füttern“, das Ergebnis ist eine Funktion mit einer entsprechend geringeren Argumentzahl.

⁸Wenn Sie diese Beispiele am Computer ausprobieren, müssen Sie andere Namen als `fst` und `snd` verwenden, da es sonst zum Konflikt mit den vordefinierten Funktionsnamen kommt.

```
plus :: Int -> Int -> Int
plus x y = x+y
```

```
plusDrei :: Int -> Int
plusDrei = plus 3
```

Mit Hilfe von Paaren lässt sich auch eine Additionsfunktion schreiben, der die zu addierenden Zahlen als *ein* strukturiertes Argument übergeben werden:

```
addierePaar :: (Int, Int) -> Int
addierePaar (x,y) = x+y
```

In dieser Form müssen beide Argumente auf einmal angegeben werden, ein unvollständiges Paar kann nicht verarbeitet werden. Bei beiden Formen muss man aufpassen, dass man die Argumente in der passenden Form angibt:

```
plus 3 5           -- aber nicht: plus (3,5)
addierePaar (3,5) -- aber nicht: addierePaar 3 5
```

Die Programmierweise, in der eine Funktion ihre Eingaben nach und nach erhält, nennt man nach Haskell B. Curry, nach dem auch diese Programmiersprache benannt ist, **currying**. Die Programmierweise, die alle Eingaben simultan erfordert, wird entsprechend **uncurrying** genannt. Beide Formen kann man mit Hilfe von Funktionen ineinander überführen. Wir zeigen dies hier für Funktionen mit zwei Veränderlichen.

`curry` ist eine Funktion, die eine Funktion mit einer Paareingabe in die entsprechende Funktion mit sukzessiver Eingabe überführt.

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

Die Funktion wendet man so an:

```
> curry addierePaar 3 5
8
```

Die Funktion `uncurry` bewirkt die umgekehrte Wandlung:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

Die Anwendung erfolgt so:

```
> uncurry plus (3,5)
8
```

5.3 Übungen

5.3.1 Listenverarbeitung

Aufgabe 1: Bauen Sie unter Verwendung von `:` und `[]` die folgenden Listen auf (direkt in Hugs):

1. `[3,9,8]`
2. `["Haskell", "macht", "Spaß"]`
3. `[]`
4. `[(3, "Tomaten"), (9, "Birnen")]`

Notieren Sie sich dabei zu jeder Liste den Datentyp.

Aufgabe 2: Lösen Sie folgenden Aufgaben bitte ohne Computer:

1. Warum ist ["Das ", "ist ", 3, "mal ", "keine ", "Liste"] keine korrekte Haskell-Liste?
2. Geben Sie den Typ von [[1,3], [4,1], [], [9,3,7]] an.
3. Schreiben Sie ein Haskell-Programm `eines`, das entscheidet, ob eine nicht-leere Liste genau ein Listenelement hat. Die Wirkung soll beispielsweise sein:

```
eines ["Dieses ist das Element"] ==> True
eines ["Mehrere ", "Elemente"] ==> False
```

Aufgabe 3: Schreiben sie eine Funktion `kopf` und eine Funktion `rumpf`, die von einer Liste (wir nehmen an, die Liste ist nicht leer) den Kopf bzw. den Rumpf bestimmt, ohne die vordefinierten Funktionen `head` und `tail` zu benutzen. Sie können dazu die Technik des Mustervergleichs verwenden.

Aufgabe 4: Schreiben Sie eine Funktion `ist_leer`, die von einer Liste entscheidet, ob sie leer ist (das Ergebnis ist dann `True`, sonst `False`).

Aufgabe 5:

1. Schreiben Sie eine Funktion `satz`, die die Elemente einer String-Liste (Typ `[String]`) zu einer einzigen Zeichenkette aneinanderhängt wie in diesem Beispiel:

```
> satz ["Haskell ", "macht ", "Spass"]
"Haskell macht Spass"
```

2. Ändern Sie die Funktion `satz` so ab, dass sie den Typ `[[a]] -> [a]` hat. Die Funktionsweise soll entsprechend auf Listen von Listen übertragen werden: die Teillisten werden zu einer einzigen Liste zusammengefügt. (Da diese Aufgabe in funktionalen Programmen häufiger gebraucht wird, ist mit `concat` eine Funktion dieser Wirkung schon vordefiniert. Programmieren Sie die Funktion bitte trotzdem selbst.)
3. Schreiben Sie eine Funktion `einzelne`, die zu einer Liste eine Liste von einelementigen Listen erzeugt. Diese einelementigen Listen enthalten der Reihe nach die Listenelemente der ursprünglichen Liste, wie es das Beispiel zeigt:

```
> einzelne [1,2,3]
[[1], [2], [3]]
```

4. Weisen Sie nach (durch Überlegung auf dem Papier), dass für jede Liste gilt

```
satz (einzelne liste) = liste
```

Warum ist jedoch `einzelne (satz liste)` im Allgemeinen nicht dasselbe wie `liste`?

5. Untersuchen Sie die vordefinierten Funktionen `words`, `unwords`, `lines`, `unlines`. Beschreiben Sie deren Wirkung auch im Vergleich mit `satz` und `einzelne`.

Aufgabe 6: Programmieren Sie eine Funktion

```
istIn :: Eq a => a -> [a] -> Bool
```

die überprüft, ob ein `a`-Wert in einer `a`-Liste enthalten ist. Da man einen Vergleich zwischen `a`-Werten benötigt, muss der Typ `a` in der Typklasse `Eq` sein, die Vergleiche zwischen ihren Elementen mit dem Operator `==` erlaubt.

Aufgabe 7: Gegeben ist diese Haskell-Funktion:

```
listenmaximum :: [Int] -> Int
listenmaximum [] = error "Leere Liste hat kein Maximum"
listenmaximum [x] = x
listenmaximum (x:y:ys)
  | x>y          = listenmaximum (x:ys)
  | otherwise    = listenmaximum (y:ys)
```

Erläutern Sie (z. B. anhand eines Beispiels), wie die Funktion den größten Wert einer `Int`-Liste findet.

Programmieren Sie dann eine Version der Funktion, die eine Hilfsfunktion mit explizitem Akkumulator verwendet.

Aufgabe 8: Die Funktionen `drop` und `take` haben den Typ `Int -> [a] -> [a]`. Von der eingegebenen Liste entfernt `drop n` die ersten `n` Einträge, `take n` liefert die Liste der ersten `n` Einträge als Ergebnis.

1. Verwenden Sie `take` und `drop`, um eine Funktion `vonbis` zu schreiben, die zwei `Ints` und eine Liste als Argument hat. Der Aufruf `vonbis n m liste` soll von `liste` die Teilliste des `n`-ten bis `m`-ten Elements erzeugen.

Beschreiben Sie auch das Verhalten (ihrer Version) der Funktion `vonbis` für diese Fälle: Die `liste` hat weniger als `m` Elemente oder `n` ist größer als `m`.

2. Programmieren Sie die Funktionen `take` und `drop` selbst. Verwenden Sie für die Funktionen andere Namen, um eine Kollision mit den vordefinierten Namen zu vermeiden.

Aufgabe 9: In dieser Aufgabe geht es um zwei Funktionen zum Vergleich zweier Listen. Programmieren Sie eine Funktion...

1. `anfangVon :: Eq a => [a] -> [a] -> Bool`, die entscheidet, ob die erste Liste der Anfang der zweiten Liste ist (dazu müssen die Listenelemente vergleichbar sein, d. h. `a` muss ein `Eq`-Typ sein).
2. `kuerzerAls :: [a] -> [b] -> Bool`, die entscheidet, ob die erste Liste weniger Elemente hat als die zweite.

5.3.2 Tupel und Currying

Bitte programmieren Sie bei den Aufgaben 10-12 jeweils zwei Versionen der Funktion. Eine, bei der die Eingaben „nacheinander“ eingegeben werden (dann haben die Funktionen einen Typ, in dem mehrere Pfeile vorkommen), und eine, bei der alle Eingaben „gleichzeitig“ zu machen sind (dann kommt im Typ der Funktion nur ein Pfeil vor, aber auch ein `Tupeltyp`).

Aufgabe 10: Schreiben Sie eine Funktion `dreiersumme`, die die Summe dreier Zahlen bildet.

Aufgabe 11: Schreiben Sie eine Funktion `grundschuldiv`, die die Division mit Rest folgendermaßen durchführt: Für eine Eingabe (a, b) besteht die Ausgabe aus dem Paar, das aus dem ganzzahligen Divisionsergebnis und dem Divisionsrest besteht.

Aufgabe 12: Schreiben Sie eine Funktion, die die Summe der Quadrate zweier Zahlen bildet (wie man es beim „Pythagoras“ braucht). Gehen Sie dazu in zwei Schritten vor:

1. Schreiben Sie zuerst die Funktion `quadrat`, die eine Zahl quadriert.
2. Benutzen Sie die Funktion `quadrat` bei der Definition der Funktion `quadratsumme`, deren Argument ein Zahlenpaar ist.

Aufgabe 13: Programmieren Sie eine Funktion `kleineGrosseSummen`, die die Elemente einer `Float`-Liste auf folgende Weise summiert: Die Zahlen, die kleiner als Null sind, sollen unberücksichtigt bleiben. Die Zahlen zwischen 0 und 1 (einschließlich) sollen getrennt von denen über 1 addiert werden. Das Ergebnis der Funktion soll ein Paar sein, dessen erste Komponente die Summe der „kleinen“ Zahlen und dessen zweite Komponente die Summe der „großen“ Zahlen ist. Beispiel:

```
> kleineGrosseSummen [-2, 0.5, 1.2, 0.7, 3.6]
(1.2, 4.8)
```

5.3.3 Vermischte Übungen

Aufgabe 14: Aus der Mathematik kennen Sie vielleicht rekursive Definitionen auch von Folgen. Betrachten Sie diese:

$$a_n = \frac{1}{2} \cdot \left(a_{n-1} + \frac{5}{a_{n-1}} \right) \text{ und } a_0 = 1$$

1. Schreiben Sie eine Haskell-Funktion `folge`, so dass `folge n = a_n` ist.
2. Probieren Sie die Funktion `folge` aus. Die Folge konvergiert. Was ist der Grenzwert?

Aufgabe 15:

1. Bestimmen Sie ohne Computerhilfe die Datentypen der folgenden Haskell-Ausdrücke:
 - `(9, "hallo", 'x')`
 - `(3.4, [1, 2, 3])`
 - `[(2, 'j'), (9, 'A')]`
 - `(45, (3, "info"))`
2. Schreiben Sie eine Haskell-Funktion `mittleres`, die aus einem Tripel (3-Tupel) das mittlere Element ausgibt.

6 Funktionen höherer Ordnung

In den ersten Kapiteln haben wir die grundlegenden Datentypen und Typkonstruktoren von Haskell kennen gelernt. Sie wissen jetzt, wie man Funktionen für Listen und Tupel schreibt, wie man Rekursion zur wiederholten Anwendung einer Funktion verwendet und wie man mit Mustervergleich und bedingten Definitionen zwischen Fällen unterscheidet.

Im funktionalen Programmierstil spielen die Funktionen aber eine noch größere Rolle. Sie gelten selbst als Datenobjekte, d.h. man kann Funktionen auch als Argumente und Ergebnisse anderer Funktionen verwenden. Funktionen, die mit Funktionen als Eingabe arbeiten, nennt man *Funktionen höherer Ordnung* (higher order functions, HOFs). Man kann solche Funktionen selbst definieren, einige sind aber Standardfunktion von Haskell. Die Funktionen `map`, `filter`, `foldl` und `foldr` sind HOFs zur Listebearbeitung und gehören zum Sprachumfang von Haskell. Sie machen die Listebearbeitung leichter und übersichtlicher, indem explizite Rekursion vermieden und in die HOFs ausgelagert wird. Wir betrachten diese Funktionen in den ersten Abschnitten dieses Kapitels.

Andere Funktionen höherer Ordnung betreffen den Umgang mit Funktionen selbst. Wir betrachten einige davon im Abschnitt „Funktionen manipulieren“.

6.1 Überall dasselbe tun: map

6.1.1 Fallbeispiel Cäsarchiffre

Um (weiter unten) die Wirkung von `map` zu beschreiben, betrachten wir ein kleines Programmbeispiel zur Chiffrierung von Texten. Wir verwenden dabei eine ganz einfache Geheimschrift, die angeblich auf Cäsar zurückgeht, und die daher Cäsar-Chiffre oder Cäsar-Verschlüsselung heißt. Dabei wird ein Text verschlüsselt, indem man jeden Buchstaben um eine festgelegte Anzahl von Positionen im Alphabet weiter schiebt (wobei man nach dem „z“ wieder mit „a“ beginnt). Die nachfolgende Tabelle gibt das für das Schieben um 3 Positionen an:

Original	a	b	c	d	e	f	g	h	i	j	k	l	m
Verschlüsselt	d	e	f	g	h	i	j	k	l	m	n	o	p
Original	n	o	p	q	r	s	t	u	v	w	x	y	z
Verschlüsselt	q	r	s	t	u	v	w	x	y	z	a	b	c

Buchstabenweise verschlüsselt man damit das Wort „haskell“ zu „kdvnhoo“. Zum Entschlüsseln muss man die Tabelle in der umgekehrten Richtung verwenden. Dennoch ist es recht mühsam, durch Nachschauen in der Tabelle ver- und entschlüsseln zu müssen, außerdem bräuchte man bei einer anderen Verschiebung des Alphabets eine andere Tabelle. Wir schreiben daher ein Programm, das die Aufgabe erledigt.

Wir beschränken uns auf die Verschlüsselung kleiner Buchstaben, und auch Sonderzeichen verwenden wir nicht. Es wäre schön, die Verschiebung um 3 Positionen durch eine Addition von 3 erreichen zu können. Nun kann man aber Buchstaben (Typ `Char`) und Zahlen (Typ `Int`) nicht einfach addieren. Daher berechnen wir zu jedem Buchstaben die Positionsnummer innerhalb des Alphabets, wobei `a` die Nummer 0, `z` die Nummer 25 haben soll:

```
position :: Char -> Int
position 'a' = 0
position 'b' = 1
...
position 'z' = 25
```

Die Nummerierung kann man in Form einer Wertetabelle angeben. Das ist jedoch sehr mühsam. Besser ist es, die eingebaute Nummerierung der Zeichen nach dem ASCII-Code zu verwenden. Dort hat **a** jedoch die Nummer 97 und **z** die Nummer 122, so dass wir von allen Nummern 97 subtrahieren müssen, um die richtigen Positionsnummern zu erhalten:

```
position :: Char -> Int
position zeichen = (ord zeichen) - 97
```

Ist die Position eines Zeichens bestimmt, so kann man die Zahl zur Verschiebung (den sogenannten Schlüssel des Chiffrierverfahrens). Zu beachten ist, dass beim Addieren nach der Nummer 25 nicht die 26, sondern wieder die 0 kommen muss. Das Additionsergebnis ist daher modulo 26 zu berechnen.

```
schiebe3 :: Int -> Int
schiebe3 zahl = mod (zahl+3) 26
```

Das Ergebnis des Schiebens ist die Positionsnummer des Geheimtext-Buchstabens. Hierzu ist wieder der richtige Buchstabe zu bestimmen. Dazu addieren wir auf die Positionsnummer wieder 97, so dass die ASCII-Nummer des Zeichens erreicht wird. Die Funktion `chr` bestimmt dazu den Buchstaben.

```
buchstabe :: Int -> Char
buchstabe zahl = chr (zahl + 97)
```

Jetzt haben wir alle Zutaten, um einen einzelnen Buchstaben verschlüsseln zu können. Wir fassen sie zusammen:

```
caesar3 :: Char -> Char
caesar3 zeichen = buchstabe (schiebe3 (position zeichen))
```

Es ist nicht so schön, dass damit nur die Verschiebung um 3 Zeichen möglich ist. Gute Programme zeichnen sich dadurch aus, dass sie flexibel verwendbar sind. Man erreicht das hier, indem man die Schlüsselzahl (Distanz der Verschiebung) der Funktion `schiebe` als Argument übergibt:

```
schiebe :: Int -> Int -> Int
schiebe schluessel zahl = mod (zahl + schluessel) 26
```

Damit programmiert man auch die flexiblere Funktion `caesar`:

```
caesar :: Int -> Char -> Char
caesar schluessel zeichen
    = buchstabe (schiebe schluessel (position zeichen))
```

Die Funktion `caesar3` ergibt sich jetzt als Spezialfall durch die partielle Auswertung von `caesar`:

```
caesar3 :: Char -> Char
caesar3 = caesar 3
```

Die Funktion `caesar` erlaubt die Verschlüsselung von „haskell“, wenn wir sie für jeden Buchstaben einzeln verwenden:

```
> caesar 3 'h'
'k'
> caesar 3 'a'
'd'
```

Dies ist noch unbefriedigend, die Funktion soll automatisch auf jeden Buchstaben einer Zeichenkette angewandt werden. Wir müssen daher aus der Funktion `caesar` für einzelne Zeichen eine Funktion für Zeichenketten machen, die `caesar` auf jedes Zeichen der Zeichenkette anwendet. Dies ist eine Standardaufgabe der Listenverarbeitung:

```
caesarText :: Int -> String -> String
caesarText schluessel ""
           = ""
caesarText schluessel (x:xe)
           = (caesar schluessel x):(caesarText schluessel xe)
```

Man kann dies also mit den Mitteln der vorangegangenen Kapitel programmieren. Für diese häufig vorkommende Aufgabe der Art „wende eine gegebene Funktion auf jedes Element einer Liste an“ gibt es jedoch eine elegantere Art der Programmierung. Sie verwendet die eingebaute Funktion `map`, die eine Funktion und eine Liste als Argument hat. `map` wendet die Funktion auf jedes Element der Liste an (wobei die Funktion einen Typ haben muss, der die Anwendung auf die Listenelemente erlaubt). Damit lässt sich die Verschlüsselungsfunktion vereinfacht programmieren:

```
caesarText :: Int -> String -> String
caesarText schluessel text = map (caesar schluessel) text
```

Die Funktion `map` ist für die Listenverarbeitung so wichtig, dass es sich lohnt, `map` einmal selbst zu programmieren, um die Wirkungsweise besser zu verstehen. Der Typ von `map` ist

```
map :: (a -> b) -> [a] -> [b]
```

Dabei ist `(a -> b)` der Typ der einzugebenden Funktion, die also zu Elementen vom Typ `a` Elemente vom Typ `b` ausgibt. Außer dieser Funktion wird eine Liste vom Typ `[a]` eingegeben. Da die Elemente der Liste den Typ `a` haben, kann die Funktion auf sie angewandt werden, und bei jeder Funktionsapplikation entsteht ein Ergebniselement vom Typ `b`. So wird zu der Liste der Elemente vom Typ `a` eine Liste von Elementen vom Typ `b` berechnet, d. h. `map` hat den Ausgabetypp `[b]`. Soweit zu den Typen. Jetzt ist noch anzugeben, wie die Berechnung Element für Element durchgeführt wird:

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xe) = (f x):(map f xe)
```

Der leeren Liste wird also die leere Liste zugeordnet. Bei anderen Listen wird `f` auf das erste Element angewendet, und es schließt sich die Liste an, bei der entsprechend verfahren wurde.

Betrachten wir abschließend die Anwendung von `map` im Cäsar-Beispiel. Dort haben wir definiert

```
caesarText schluessel text = map (caesar schluessel) text
```

Die Liste, die verarbeitet wird, wird mit der `String` (bzw. `[Char]`)-Variablen `text` angesprochen. Die Funktion, die mittels `map` elementweise angewendet wird, ist `(caesar schluessel)`. Da `caesar` den Typ `Int -> Char -> Char` hat, hat die partielle Anwendung `(caesar schluessel)`, bei der nur das erste Argument (ein `Int`) angegeben ist, den Typ `Char -> Char`. Somit „passt“ diese Funktion auf die Elemente der Liste `text`, und auch die Elemente der Ergebnisliste haben den Typ `Char`, d. h. das Ergebnis ist wie gewünscht ein `String`.

6.1.2 Weitere Beispiele mit *map*

In vielen Zusammenhängen ist mit allen Elementen einer Liste dasselbe zu tun. So lässt sich *map* auch verwenden, um alle Zahlen einer *Float*-Liste mit einer Konstanten zu multiplizieren. Interpretiert man die Zahlen als Netto-Preise, so lässt sich mittels *map* die Liste der Bruttopreise erzeugen (den Mehrwertsteuersatz setzt man am besten als leicht veränderbare Variable des Programms an).

```
mwstsatz = 16
```

```
bruttopreis :: Float -> Float
bruttopreis nettopreis = nettopreis * (1+mwstsatz/100)
```

```
bruttoliste :: [Float] -> [Float]
bruttoliste liste = map bruttopreis liste
```

Lässt man die Variable für die Nettopreislste weg, so erhält man eine noch elegantere Formulierung, die die partielle Auswertung von *map* benutzt:

```
bruttoliste :: [Float] -> [Float]
bruttoliste = map bruttopreis
```

Eine andere Anwendung von *map* besteht in der Überprüfung einer Bedingung für jedes Element einer Liste. Möchte man für jedes Element einer Zahlenliste überprüfen, ob es größer als Null ist, so mapped man eine entsprechende Funktion über die Liste:

```
groesserNull :: Float -> Bool
groesserNull zahl = (zahl>0)
```

```
groesserNullListe :: [Float] -> [Bool]
groesserNullListe = map groesserNull
```

Die Funktion lässt sich in dieser Art anwenden:

```
> groesserNullListe [1,3,-5,2]
[True,True,False,True]
```

Möchte man wissen, ob alle Elemente der Liste größer als Null sind, so kann man nun die Listenelemente mit *&&* (und) verbinden. Eine elegante Art, dies zu tun, lernen wir im Abschnitt über *fold* kennen.

6.2 Elemente auswählen: *filter*

In manchen Situationen möchte man von einer Liste nur die Elemente behalten, die einer bestimmten Bedingung genügen. Die anderen Elemente sollen aus der Liste entfernt werden. Dies leistet die Funktion *filter*, die wir zunächst selbst programmieren (obwohl sie als Standardfunktion verfügbar ist).

Die Funktion *filter* muss eine Bedingung (das ist eine Funktion mit Ergebnistyp *Bool*) und eine Liste als Argumente haben. Die Bedingung muss auf die Listenelemente anwendbar sein, d. h. der Argumenttyp der Bedingung muss mit dem Typ der Listenelemente übereinstimmen. Die Ergebnisliste hat denselben Typ wie die Argumentliste, da lediglich (ggf.) manche Listenelemente entfernt werden. Damit ergibt sich

```
filter :: (a -> Bool) -> [a] -> [a]
filter bedingung []      = []
filter bedingung (x:xe)
  | (bedingung x)        = x:(filter bedingung xe)
  | otherwise            = filter bedingung xe
```

Elemente `x` werden also ins Ergebnis aufgenommen, falls (`bedingung x`) den Wert `True` liefert, wenn also die Bedingung erfüllt ist. Ansonsten werden sie fallen gelassen, und es wird rekursiv der Rest der Liste gefiltert.

Wir können `filter` verwenden, um die Eingaben zur Cäsar-Verschlüsselung von unerlaubten Zeichen zu befreien. Oben haben wir die Verschlüsselung nur für Kleinbuchstaben definiert. Mit

```
istKlein :: Char -> Bool
istKlein zeichen = ('a'<=zeichen)&&(zeichen<='z')
```

testen wir, ob ein Zeichen im „erlaubten Bereich“ liegt. Dann bleiben mit

```
caesarGefiltert :: Int -> String -> String
caesarGefiltert schluessel text
    = caesarText schluessel (filter istKlein text)
```

nur die Verschlüsselungen der Kleinbuchstaben übrig:

```
> caesarGefiltert 3 "Haskell ist cool"
"dvnhoolvwfrro"
```

Eine andere Frage ist es natürlich, ob man mit dem Verschwinden der Großbuchstaben und Leerzeichen zufrieden ist, oder ob man hier nicht lieber den Verschlüsselungsalgorithmus ändern sollte. Zum Beispiel hätte man mit `map` (und einer geeigneten Funktion) zunächst alle Großbuchstaben in entsprechende Kleinbuchstaben umwandeln können.

Oben haben wir mit `map` eine Funktion programmiert, die zu allen Elementen einer Liste feststellt, ob sie größer als Null sind. Eine leicht andere Aufgabe besteht darin, von einer Liste alle Elemente auszuwählen, die größer als Null sind, und die anderen aus der Liste zu streichen. Hierzu kann man `filter` verwenden:

```
nurPositive :: [Float] -> [Float]
nurPositive = filter groesserNull
```

Die Anwendung ergibt die Teilliste der positiven Zahlen einer Liste:

```
> nurPositive [1,3,-5,2]
[1,3,2]
```

6.3 Alles zusammenfassen: `fold`

Neben dem Anwenden einer Funktion auf jedes Element einer Liste (`map`) und dem Herausfinden von Elementen mit einer bestimmten Eigenschaft (`filter`) gibt es eine dritte Standardaufgabe der Listenverarbeitung: Aus allen Elementen einer Liste *ein* Ergebnis berechnen.

Typische solche Aufgaben sind: Alle Zahlen einer Liste aufsummieren, die logische Konjunktion (`&&`) aller Elemente einer `Bool`-Liste bilden, alle Elemente einer `String`-Liste zu einem einzigen `String` zusammenfassen. Für solche Aufgaben gibt es die Funktion `fold` in verschiedenen Varianten.

Wir beginnen mit dem Aufsummieren von Werten einer `Int`-Liste. Hierzu haben wir im Kapitel über Listen schon die Funktion `summe` kennen gelernt. Wir programmieren nun eine veränderte Summenfunktion, die einen Akkumulator verwendet:

```
summe :: Num a => a -> [a] -> a
summe acc []      = acc
summe acc (x:xe) = summe (acc + x) xe
```

Um die Summe der Elemente einer Liste zu erhalten, ruft man `summe` mit dem Akkumulatorstartwert 0 auf:

```
> summe 0 [1,2,3,4,5]
15
```

Es ist eine ganz ähnliche Aufgabe, das Produkt der Zahlen in einer Liste zu bestimmen. Die Funktion `produkt` zeigt, dass nur das Operationszeichen zu ändern ist:

```
produkt :: Num a => a -> [a] -> a
produkt acc [] = acc
produkt acc (x:xe) = produkt (acc * x) xe
```

Beim Aufruf muss der Akkumulator einen geeigneten Startwert erhalten. Beim Multiplizieren ist dies 1:

```
> produkt 1 [1,2,3,4,5]
120
```

Um zu prüfen, ob alle Elemente einer `Bool`-Liste den Wert `True` haben, kann man sie mit dem Operator `&&` (und) verbinden. Es ergibt sich wieder eine ganz ähnliche Programmstruktur:

```
alleTrue :: Bool -> [Bool] -> Bool
alleTrue acc [] = acc
alleTrue acc (x:xe) = alleTrue (acc && x) xe
```

Beim Aufruf initialisiert man den Akkumulator mit `True`, dem neutralen Element der logischen Konjunktion `&&`:

```
> alleTrue True [True, True, False]
False
```

Schließlich lässt sich ebenso einfach ein Programmstück angeben, das alle `Strings` einer `String`-Liste zu einem einzigen `String` zusammen fasst.

Wir verzichten darauf und betrachten statt dessen die allgemeine Struktur dieser Programme. Stellvertretend berechnen wir noch einmal die `summenfunktion` Schritt für Schritt. Zur Verdeutlichung des Rechenablaufs verzichten wir dabei auf das Berechnen der Werte im Akkumulator und schreiben statt dessen jeweils eine geklammerte Summe:

```
> summe 0 [1,2,3,4]
= summe (0+1) [2,3,4]
= summe ((0+1)+2) [3,4]
= summe (((0+1)+2)+3) [4]
= summe ((((0+1)+2)+3)+4) []
= (((((0+1)+2)+3)+4) = 10
```

Es zeigt sich, dass in der `summenfunktion` die Addition von links nach rechts ausgeführt wird, d. h. eine Klammerung von links her stattfindet. Entsprechend werden auch die Rechnungen bei `produkt` und `alleTrue` von links her geklammert, so dass der Akkumulator das Ergebnis von links her sammelt. Dabei wird stets ein Operator verwendet, der aus zwei Elementen vom Typ der Listenelemente ein neues berechnet. Man könnte sagen, die Liste wird von links her mit Hilfe eines Operators zu einem Wert „zusammengefaltet“. Dies begründet den Namen `foldl` (links falten) für die Funktion, die diese allgemeine Struktur ausdrückt:

```
foldl :: (a -> a -> a) -> a -> [a] -> a
foldl f acc [] = acc
foldl f acc (x:xe) = foldl (f acc x) xe
```

Anstelle eines Operators, der zwischen seine Argumente geschrieben wird, wurde nun ein Funktion mit zwei Argumenten benutzt. Diese ist das erste Argument von `foldl` und hat den Typ `a -> a -> a`. Das zweite Argument ist der Akkumulator vom Typ `a`, das dritte die Liste mit Typ `[a]`.

Mit Hilfe der Funktion `foldl` schreibt man die oben besprochenen Funktionen einfacher. Wir verwenden dabei in runde Klammern gesetzte Operatoren. Auf diese Weise werden aus den Operatoren, die zwischen ihre Argumente geschrieben werden, Funktionen, die vor die Argumente zu schreiben sind (`(+) 3 5` statt `3 + 5`).

```
produkt acc liste = foldl (*) acc liste
alleTrue acc liste = foldl (&&) acc liste
```

Tatsächlich kann man den Typ von `foldl` noch etwas allgemeiner definieren. Es muss nämlich nicht der Fall sein, dass das Ergebnis der Listenfaltung denselben Typ hat wie die Listenelemente.

Als Beispiel betrachten wir eine Funktion, die das Ergebnis einer Buchstabenzählfunktion für die Druckausgabe vorbereitet. Dazu sei `bzaehl` eine Funktion, die die Häufigkeit des Auftretens der einzelnen Buchstaben in einer Zeichenkette zählt. Das Ergebnis sei eine Liste von `(Char, Int)`-Paaren, in denen die Zahl die Häufigkeit des entsprechenden Zeichens nennt.

```
[('a', 7), ('b', 4), ('c', 1)]
```

bedeutet, dass 'a' 7 mal, 'b' 4 mal und 'c' 1 mal auftrat.

Die Ausgabe soll eine Tabelle sein, deren erste Spalte das Zeichen und deren zweite Spalte die Zahl enthält. Um dies ausgeben zu können, muss anstelle der Zahl die Zeichenkette verwendet werden, die die Zahl darstellt (also "7" für 7). Hierzu wird die Standardfunktion `show` verwendet. Zusätzlich sind zwischen Zeichen und Zahl einige Leerzeichen und nach jeder Zeile das Zeichen `'\n'` für „neue Zeile“ einzufügen. Das Paar

```
('a', 7)
```

ist also in die Zeichenkette

```
"a 7\n"
```

zu transformieren. Dies geschieht allgemein durch

```
zeile :: (Char, Int) -> String
zeile (zeichen, zahl) = zeichen:( "  " ++ (show zahl) ++ "\n")
```

Hat man schon einen Teil der Tabelle erstellt, so lässt sich eine neue Zeile (für ein neues Paar der Liste) hinzufügen durch

```
zurTabelle :: String -> (Char, Int) -> String
zurTabelle tabelleBisher paar = tabelleBisher ++ (zeile paar)
```

Um die gesamte Tabelle zu erstellen, kann man `foldl` (mit einem allgemeineren Typ) verwenden:

```
tabelle :: [(Char, Int)] -> String
tabelle paarliste = foldl zurTabelle "" paarliste
```

Der Typ von `foldl` unterscheidet nun zwischen dem Typ `a` des Ergebnisses und dem (hier) anderen Typ `b` der Listenelemente. Das erste Argument von `foldl` ist jetzt eine Funktion, die aus einem `a` und einem `b` ein `a` berechnet. Damit ist der allgemeine Typ von `foldl`

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Wir haben gesehen, dass `foldl` eine Liste von links her zusammen faltet. Genauso gut hätte man von rechts her vorgehen können. Bei der Summation ändert dies nur die Additionsreihenfolge, aber nicht das Ergebnis:

```
((1+2)+3)+4)+5 = 1+(2+(3+(4+5)))
```

In der Tat ist es bei vielen Operationen egal, ob man von links oder von rechts her rechnet. Solche Operationen nennt man „assoziativ“. Bei manchen Operationen macht es jedoch einen Unterschied, auf welcher Seite man mit der Auswertung beginnt. Bei der Subtraktion vieler Zahlen (die in einer Liste enthalten sind) von einem Startwert muss man von links her arbeiten. Dazu kann die `foldl`-Funktion benutzt werden, wobei man den Akkumulator mit dem Startwert initialisiert:

```
> foldl (-) 20 [1,2,3,4,5]
5
```

Beim Potenzieren ist die Konvention dagegen die Klammerung nach rechts: $2^{3^4} = 2^{(3^4)}$. Um eine Zahl (Basis) mit den Zahlen in einer Liste (Exponenten) zu potenzieren, ist `foldl` daher ungeeignet:

```
> foldl (^) 2 [3,4]
4096
```

während $2^{3^4} = 2417851639229258349412352$ ist. Daher gibt es auch eine `fold`-Funktion, die von rechts her rechnet:

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

Hierzu betrachten wir die Anwendung mit dem Potenz-Operator:

```
> foldr (^) 1 [2,3,4]
= (^) 2 (foldr (^) 1 [3,4])
= 2 ^ (foldr (^) 1 [3,4])
= 2 ^ ((^) 3 (foldr (^) 1 [4]))
= 2 ^ (3 ^ (foldr (^) 1 [4]))
= 2 ^ (3 ^ ((^) 4 (foldr (^) 1 [])))
= 2 ^ (3 ^ (4 ^ (foldr (^) 1 [])))
= 2 ^ (3 ^ (4 ^ 1))
= 2 ^ (3 ^ 4)
= 2 ^ 81
= 2417851639229258349412352
```

Bei Verwendung von `foldr` wird der Startwert des Akkumulators also ganz rechts neben die Elemente der Liste geschrieben, und es werden Listenelemente und Startwert von rechts geklammert durch die Operation verbunden.

6.4 Funktionen manipulieren

Funktionale Programmierung erfordert geradezu, dass man eine Programmieraufgabe in kleine Teilaufgaben zerlegt, die jeweils in ganz wenigen Zeilen programmiert werden können. Ein Teil der Kürze funktionaler Programme ist sicher begründet durch die Verwendung von Funktionen höherer Ordnung wie `map` oder `filter`. Damit lassen sich Programmieraufgaben in einer Zeile erledigen, die in mancher imperativen Programmiersprache eine halbe Seite Platz brauchen.

Neben den Listenbearbeitungsfunktionen sind aber auch Funktionen wichtig, die ein flexibles (Wieder)verwenden bereits programmierter Funktionen erlauben. Die nächsten Abschnitte stellen wichtige Funktionen hierzu vor.

6.4.1 Eins nach dem anderen: Funktionskomposition

Beim Zerlegen einer Programmieraufgabe in Teilaufgaben erhält man oft solche Teilaufgaben, die nacheinander zu erledigen sind, um die Gesamtaufgabe zu bearbeiten.

Wir haben ein solches Beispiel schon beim Verschlüsseln eines Textes in Kapitel 6.1.1 gesehen. Um ein einzelnes Zeichen um drei Positionen im Alphabet weiter zu schieben, wurde dort die Funktion `caesar3` definiert:

```
caesar3 :: Char -> Char
caesar3 zeichen = buchstabe (schiebe3 (position zeichen))
```

Die Klammerung zeigt, was der Reihe nach zu tun ist: Erst die `position` eines Zeichens bestimmen, diese dann mit `schiebe3` verändern, und das Ergebnis hiervon in einen `buchstaben` verwandeln. Es geht also um die Hintereinanderanwendung der Funktionen `position`, `schiebe3` und `buchstabe` auf das Argument `zeichen`.

Aus der Mathematik ist mit \circ ein besonderes Symbol für die Hintereinanderanführung oder Komposition von Funktionen bekannt. Sind f und g Funktionen, so schreibt man $f \circ g$ für die Funktion mit

$$(f \circ g)(x) = f(g(x)) ,$$

die Funktion also, die für jedes Argument x zunächst $g(x)$ bestimmt und dieses Ergebnis als Argument von f verwendet, so dass der Funktionswert der Komposition $f(g(x))$ ist. Man muss dabei zweierlei beachten: Zum einen ist $f(g(x))$ etwas anderes als $g(f(x))$, da es auf die Reihenfolge der Ausführung ankommt (bedeutet z. B. f das Verdoppeln einer Zahl und g das Quadrieren, so ist $f(g(3)) = 18$, aber $g(f(3)) = 36$). Zum anderen muss der Ergebnistyp der ersten Funktion (g) mit dem Argumenttyp der zweiten Funktion (f) übereinstimmen, damit f auf $g(x)$ angewendet werden kann.

In Haskell ist der kleine Kreis \circ zu einem Punkt zusammen geschmolzen, und die Komposition von `f` und `g` schreibt man als `f.g`.

Im Beispiel von `caesar3` erhält man daher eine äquivalente Definition durch

```
caesar3 :: Char -> Char
caesar3 zeichen = (buchstabe . schiebe3 . position) zeichen
```

was sehr klar ausdrückt, dass das `zeichen` durch die Komposition der drei angegebenen Funktionen „geschleust“ wird.

Bei einer solchen Komposition mehrerer Funktionen ist es im Übrigen egal, wie man die Komposition klammert:

```
buchstabe . (schiebe3 . position)
```

liefert das selbe Ergebnis wie

```
(buchstabe . schiebe3) . position
```

so dass man die Klammern ganz weglassen darf. Die Klammern in der Definition von `caesar3` dienen nur der Abgrenzung der Komposition vom Argument `zeichen`. Ohne die Klammern würde zuerst `position zeichen` bestimmt, was eine Zahl ist und nicht in eine Komposition mit einer Funktion einfließen kann.

Wir schließen die Betrachtung von `(.)` mit der Bestimmung des Typs des Operators ab. Die Argumente sind zwei Funktionen, die „in der Mitte“ hinsichtlich des Typs zusammen passen müssen. Wegen der Ausführung der Funktionen von rechts nach links steht die „Mitte“ an den „Rändern“ der Argumenttypen. Das Ergebnis ist eine Funktion, die denselben Argumenttyp wie die zweite Funktion und den Ergebnistyp der ersten Funktion hat und den Typ in der Mitte „überspringt“:

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

6.4.2 Nur nicht zu viel: partielle Auswertung

Ebenfalls im Beispiel zur Verschlüsselung haben wir die Funktion `caesar3` als Spezialfall der Funktion `caesar` definiert. Mit

```
caesar :: Int -> Char -> Char
caesar schluessel zeichen
    = buchstabe (schiebe schluessel (position zeichen))
```

ist

```
caesar3 :: Char -> Char
caesar3 = caesar 3
```

Man nennt dies eine *partielle Auswertung* der Funktion `caesar`, denn `caesar` hat zwei Argumente, von denen hier nur eines angegeben ist. Es wird automatisch angenommen, dass dies das erste der beiden erforderlichen Argumente ist. Vom

```
Int -> Char -> Char
```

den wir als „gib mir ein `Int` und ein `Char`, so gebe ich dir ein `Char` zurück“ lesen, bleibt damit übrig

```
Char -> Char
```

was ganz richtig „gib mir ein `Char`, so gebe ich dir ein `Char` zurück“ bedeutet, denn `caesar` wartet nach der Eingabe der `3` gewissermaßen noch auf die Eingabe des zweiten Arguments, das ein `Char` sein muss. Aus der Funktion mit zwei Argumenten ist so durch die partielle Angabe der Argumente eine Funktion mit einem Argument geworden.

Derartige partielle Auswertungen sind für alle Funktionen mit mehreren Argumenten möglich, sofern diese im Curry-Stil nacheinander angegeben werden können. Wir verzichten auf eine allgemeine Formulierung mit beliebiger Argumentanzahl. Das allgemeine Prinzip mag statt dessen am folgenden Beispiel deutlich werden.

Ist `f` eine Funktion mit 5 Argumenten und dem Typ

```
f :: t1 -> t2 -> t3 -> t4 -> t5 -> t
```

und wird sie auf `a1` vom Typ `t1` und `a2` vom Typ `t2` angewandt, so ergibt sich

```
f a1 a2 :: t3 -> t4 -> t5 -> t
```

Durch die partielle Auswertung entsteht also wieder eine Funktion, deren Argumente die noch ausstehenden Argumente der ursprünglichen Funktion sind. Vom Typ der ursprünglichen Funktion werden von links her die Argumenttypen der angegebenen Argumente weg genommen.

Wir betrachten hierzu ein Beispiel, das die Nützlichkeit partieller Auswertungen für die Wiederverwendbarkeit von Funktionen zeigt.

In Kapitel 5.1 haben wir eine Funktion zum *Sortieren durch Einfügen* programmiert, die zum Sortieren von beliebigen Listen geeignet ist, solange deren Elemente einen Ordnungstyp haben:

```
sortiere :: Ord a => [a] -> [a]
sortiere liste = hsortiere liste []
```

Die Einschränkung `Ord a` an den Typ der Listenelemente kommt daher, dass beim Einfügen in eine schon sortierte Liste Vergleiche mit dem Operator `<` zwischen Listenelementen notwendig werden. Solche Vergleiche sind nur für Elemente von Ordnungstypen möglich.

Was aber, wenn eine Liste von Paaren sortiert werden soll? Selbst wenn in jeder Komponente des Paares Ordnungstypen verwendet werden, sind die Paare nicht einfach mit `<` vergleichbar. Zum Vergleich könnte man nämlich sehr verschieden vorgehen. Hat man Paare von Zu- und Vornamen, die alphabetisch sortiert werden sollen, könnte man zuerst nach dem Zunamen sortieren und dann erst bei Namensgleichheit nach dem Vornamen. Man könnte aber auch eine Komponente des Paares bezüglich des Ordners ignorieren und nur auf die andere achten. Es gibt also keine Standardordnung für Paare. Wenn man eine Liste von Paaren ordnen will, muss man explizit angeben, welche Ordnungsfunktion man verwenden will.

Einer Sortierfunktion, die in dem Sinne universell ist, dass jede Liste sortiert werden kann, zu der man eine Ordnung angeben kann, erfordert diese Ordnungsfunktion als weiteres Argument. Ordnungsfunktionen geben zu zwei Argumenten eines Typs ein `Bool`-Ergebnis aus, haben also den Typ

```
kleiner :: a -> a -> Bool
```

Nimmt man die Ordnungsfunktion als erstes Argument der Sortierfunktion und die zu sortierende Liste als zweites, so erhält man für den Typ des Sortierens:

```
sortiere :: (a -> a -> Bool) -> [a] -> [a]
```

Wir können den Typ lesen als: „Sag mir, wie man Elemente vom Typ `a` vergleichen kann und gib mir eine `a`-Liste, so erhältst du eine `a`-Liste zurück“. Die Einschränkung `Ord a` an den Typ der Elemente ist nun nicht mehr nötig, da die einzugebende Ordnungsfunktion an die Stelle des `<` tritt.

Nun geht es darum, die Funktion zu programmieren. Dazu werden auch die verwendeten Hilfsfunktion geändert. Wir beginnen mit dem Einfügen eines Elements in eine schon sortierte Liste.

```
einfuegen :: (a -> a -> Bool) -> a -> [a] -> [a]
einfuegen kleiner y []      = [y]
einfuegen kleiner y (x:xe)
  | kleiner y x             = y:x:xe
  | otherwise               = x:(einfuegen kleiner y xe)
```

Gegenüber der oben definierten Funktion `einfuegen` hat sich der Typ geändert, die Ordnungsfunktion `kleiner` wird nämlich explizit als Argument übergeben. Außerdem wurde überall das `<` (zwischen zwei Elementen) durch ein `kleiner` (vor den beiden Elementen) ersetzt.

Die Sortierfunktion selbst wird damit zu:

```
sortiere :: (a -> a -> Bool) -> [a] -> [a]
sortiere kleiner liste = hsortiere kleiner liste []

hsortiere :: (a -> a -> Bool) -> [a] -> [a] -> [a]
hsortiere kleiner [] zielliste
  = zielliste
hsortiere kleiner (x:xe) zielliste
  = hsortiere kleiner xe (einfuegen kleiner x zielliste)
```

Durch partielle Auswertung dieser „universellen Sortierfunktion“ erhält man Sortierfunktionen für spezielle Zwecke. Die ursprüngliche Sortierfunktion, die den Operator `<` verwendet, erhält man, indem man `(<)` als Ordnungsfunktion angibt.

```
sortiereAufsteigend :: Ord a => [a] -> [a]
sortiereAufsteigend = sortiere (<)
```

Man kann aber auch die umgekehrte Ordnung ($>$) verwenden, um absteigend zu sortieren.

```
sortiereAbsteigend :: Ord a => [a] -> [a]
sortiereAbsteigend = sortiere (>)
```

Schließlich lassen sich, wie versprochen, Datentypen sortieren, für die ($<$) nicht verwendet werden kann. Wir zeigen das für Paare, die nun bezüglich ihrer ersten Komponente aufsteigend sortiert werden sollen. Dann muss die erste Komponente einen Ordnungstyp enthalten, und die Ordnungsfunktion ist

```
kleinerVorne :: Ord a => (a,b) -> (a,b) -> Bool
kleinerVorne (x1,y1) (x2,y2) = x1<x2
```

Die Sortierfunktion zum aufsteigenden Sortieren von Paaren bezüglich ihrer ersten Komponente ist damit

```
sortierePaareVorne :: Ord a => [(a,b)] -> [(a,b)]
sortierePaareVorne = sortiere kleinerVorne
```

Durch Ausnutzung von Polymorphie der allgemeinen Sortierfunktion und partielle Auswertung entstehen so nach Belieben speziellere Sortierfunktionen für viele Zwecke. Den Sortieralgorithmus selbst muss man aber nur einmal programmieren.

6.4.3 Die Argumentreihenfolge vertauschen: flip

Durch partielle Auswertung lassen sich aus allgemeineren Funktionen durch Festlegung z.B. des ersten Argumentes speziellere Funktionen gewinnen. Wie aber geht man vor, wenn man nicht das erste Argument angeben will, sondern das zweite? Hierzu ist die Funktion `flip` nützlich, die in diesem Abschnitt vorgestellt wird. Zuvor soll aber ein Anwendungsbeispiel genannt werden, in dem die partielle Auswertung im zweiten Argument tatsächlich sinnvoll ist.

In Kapitel 6.1.1 wurde die Funktion

```
caesarText :: Int -> String -> String
```

programmiert. Ihr Argument ist eine Zahl (der *Schlüssel* des Chiffrierverfahrens) und ein Text, der verschlüsselt werden soll. Um geheime Nachrichten mit einem Partner auszutauschen, wird man sich auf einen Schlüssel einigen und dann alle Texte hiermit verschlüsseln. Die partielle Auswertung der Funktion im ersten Argument ergibt die dazu nötige Verschlüsselungsfunktion. Hat man als Schlüssel die Zahl 3 vereinbart, so ist dies

```
meineVerschluesselung :: String -> String
meineVerschluesselung = caesarText 3
```

Die Anwendung ergibt beispielsweise

```
> meineVerschluesselung "flip"
"iols"
```

so dass "iols" der Geheimtext zu "flip" ist.

So richtig geheim ist diese Verschlüsselung natürlich nicht, da man sie sehr leicht auch Entschlüsseln kann. So ist zur Verschlüsselung mit dem Schlüssel 3 die erneute Verschlüsselung mit dem Schlüssel $26 - 3 = 23$ die passende Entschlüsselung. Wenn man den Schlüssel jedoch nicht kennt, so probiert man einfach alle Schlüssel von 1 bis 25 aus und überprüft, mit welchem man einen vernünftigen Klartext erhält.

Ist der zu entschlüsselnde Text durch "n1olpt" gegeben, so ist hierauf die Funktion `caesarText` mit wechselnden Schlüsseln anzuwenden. Uns interessieren also die Ergebnisse von

```
caesarText 1 "nlolpt"
caesarText 2 "nlolpt"
caesarText 3 "nlolpt"
...
caesarText 25 "nlolpt"
```

Dazu ist eine partielle Auswertung von `caesarText` im zweiten Argument eine gute Vorbereitung, denn es ist dann nur noch der jeweilige Schlüssel anzugeben. Tatsächlich ist die partielle Auswertung jedoch nur im ersten Argument möglich. Die Vertauschung der beiden Argumente mit `flip` hilft daher weiter:

```
entschluesseleNLOLPT :: Int -> String
entschluesseleNLOLPT = flip caesarText "nlolpt"
```

Die Funktion `flip` bewirkt, dass `caesarText` ihre Argumente in umgekehrter Reihenfolge erwartet. Damit kann nun das zweite Argument zuerst eingegeben werden. Das Ergebnis ist eine Funktion des ersten Arguments von `caesarText`.

Um nicht alle möglichen Schlüssel einzeln eingeben zu müssen, erzeugen wir die Liste der Zahlen von 1 bis 25 und `map`pen die Funktion `entschluesseleNLOLPT` darüber. In der Ergebnisliste suchen wir nach einem sinnvollen Klartextwort.

```
> map entschluesseleNLOLPT [1..25]
["ompmqu","pnqnrsv","qorosw","rpsptx","sqtquy",
"trurvz","usvsua","vtwtxb","wuxuyc","xvyvzd",
"yvwzuae","zxaxbf","aybycg","bzczdhd","cadaei",
"dbebfj","ecfcgk","fdgdhl","geheim","hfifjn",
"igjgko","jkhkhlp","kilimq","ljmjnr","mknkos"]
```

Die Durchsicht dieses Ergebnisses lässt vermuten, dass der Klartext der Botschaft das Wort `geheim` ist.

Wir kommen zurück zur Funktion `flip` im Allgemeinen. Ihr Argument ist eine Funktion mit zwei Argumenten. Das Ergebnis ist eine Funktion, in der die Argumente in ihrer Reihenfolge vertauscht sind. Unter Berücksichtigung der Konvention, dass rechts stehende Klammern in Typangaben weggelassen werden dürfen, ist der Typ von `flip`:

```
flip :: (a -> b -> c) -> b -> a -> c
```

Wie `flip` programmiert wird, ist Gegenstand einer Übungsaufgabe.

6.5 Übungen

Aufgabe 1: Programmieren Sie die folgenden Funktionen aus den Übungen zu Kapitel 5 unter Verwendung von Funktionen höherer Ordnung. Sie sollten auf diese Weise die explizite Verwendung von Rekursion vermeiden:

1. `satz`,
2. `istIn`,
3. `listenmaximum`,
4. `kleineGrosseSummen`

Aufgabe 2: Wir greifen das Thema „Verschlüsselung“ auf und betrachten diesmal das Entschlüsseln eines Textes. Wenn — wie beim Cäsar-Verfahren — die Verschlüsselung nur im Austausch von Buchstaben gegen andere Buchstaben besteht, kann eine Entschlüsselung durch „unbefugte Leser“ durch einen so genannten Kryptoangriff durch Häufigkeitsanalyse erfolgen. Ist der verschlüsselte Text genügend lange, so wird man davon ausgehen, dass die Häufigkeit des Auftretens einzelner Zeichen ähnlich wie in anderen Texten der selben Sprache ist. Tritt im verschlüsselten Text z. B. der Buchstabe „q“ besonders oft auf, so liegt es nahe, dass das „q“ für ein „e“ des Klartextes steht. Zur Entschlüsselung muss man also Häufigkeitstabellen für das Auftreten der Buchstaben im Text erstellen.

In dieser Aufgabe sollen Sie eine Funktion zur Erstellung einer solchen Häufigkeitstabelle erstellen. Gehen Sie dabei von diesen Randbedingungen aus: Beim Verschlüsseln wurden ähnlich wie beim Cäsar-Verfahren lediglich die Buchstaben gegeneinander ausgetauscht. Leerzeichen und Satzzeichen wurden unverändert aus dem Klartext übernommen. Kleine Buchstaben wurden gegen kleine und große gegen die entsprechenden großen vertauscht (also z. B. „q“ für „e“ und dann auch „Q“ für „E“). Es wird nur das Alphabet von A bis Z verwendet, also keine deutschen Sonderzeichen.

Nun soll eine Häufigkeitstabelle erstellt werden, in der nur Buchstaben Berücksichtigung finden, Satzzeichen etc. sollen also nicht gezählt werden. Außerdem wird nicht zwischen Groß- und Kleinbuchstaben unterschieden. Die Tabelle soll die häufigsten Buchstaben zuerst nennen. Für den Text „Holt Otto oder Anna“ soll das Ergebnis so aussehen:

```
o  4
t  3
a  2
n  2
h  1
l  1
d  1
e  1
r  1
```

Dabei ist die Reihenfolge der Buchstaben, die die selbe Häufigkeit haben, nicht relevant.

Gehen Sie zur Programmierung in diesen Schritten vor:

1. Zuerst müssen aus einer Zeichenkette alle Sonderzeichen entfernt und Großbuchstaben in die korrespondierenden Kleinbuchstaben umgewandelt werden.
2. Aus der Zeichenkette, die nur noch kleine Buchstaben enthält, muss eine Liste von Paaren erzeugt werden. Dabei ist jedes Paar vom Typ `(Char, Int)`, enthält also einen Buchstaben zusammen mit einer Häufigkeitsangabe.
3. Die Liste der Häufigkeiten muss absteigend nach der Häufigkeitsangabe (zweite Komponente des Paares) sortiert werden.
4. Das Ergebnis der Sortierung muss für die Ausgabe formatiert werden. Verwenden Sie `'\n'` als Zeichen für „neue Zeile“. Das erzeugt beim Ausprobieren in Hugs zwar nur ein `'\n'` und keine neue Zeile, aber es leistet später beim Schreiben in Dateien das gewünschte.
5. Fügen Sie die bis hier programmierten Teile zusammen zu einer Funktion `textAnalyse :: String -> String`, die die geforderte Aufgabe erfüllt.

Vielleicht wollen Sie die Funktion auf einen längeren Text anwenden, ohne diesen in Hugs eintippen eintippen zu müssen. Der Text soll statt dessen aus einer Datei gelesen und das Ergebnis in eine Datei geschrieben werden. Dazu können Sie folgende Funktion in Ihr Programm einfügen, die hier nicht erläutert wird:

```
anwenden :: (String -> String) -> FilePath -> FilePath -> IO()
anwenden programm einDatei ausDatei
    = readFile einDatei
      >>= \text -> writeFile ausDatei (programm text)
```

Um einen Text aus der Datei `meinText` zu verarbeiten und das Ergebnis in die Datei `meinErgebnis` zu schreiben, geben Sie in Hugs dann ein:

```
anwenden textAnalyse "meinText" "meinErgebnis"
```

A Lösungen zu den Übungen

A.1 Lösungen zu Kapitel 3

Aufgabe 1: Es sind nebeneinander die Haskell-Ausdrücke, ihr Wert und ihr Typ angegeben. Die Typangabe nennt zuerst den allgemeinen Typ des Ausdrucks unter Verwendung von Typklassen, danach kommt noch ein Beispiel eines gültigen konkreten Typs. Für den ersten Ausdruck ist dies

```
4+9      13 :: Num a => a      13 :: Integer
```

Dies bedeutet, `4+9` hat den Wert `13` der zu jedem Zahlentyp (Typklasse `Num`) gehört. Zahlentypen sind zum Beispiel `Int`, `Integer`, `Float` oder `Double`. Anstelle von `13 :: Integer` könnte man daher auch `13 :: Int`, `13 :: Float` oder `13 :: Double` als konkreten Typ angeben. Haskell kennt noch weitere Zahlentypen, etwa zur Darstellung von Brüchen durch die Angabe von Zähler und Nenner. Auch dies sind gültige Typen für `4 + 9` und andere Beispiele, sie werden hier jedoch nicht besprochen.

Konkrete Typen für die Typklasse `Fractional` sind `Float` und `double`, für die Typklasse `Integral` sind es `Int` und `Integer`.

```
3-5          -2 :: Num a => a          -2 :: Int
-7*3         -21 :: Num a => a         -21 :: Int
9/3          3.0 :: Fractional a => a  3.0 :: Double
9 'div' 3    3 :: Integral a => a      3 :: Integer
2.3+1.1     3.4 :: Fractional a => a  3.4 :: Double
9.8-3.1     6.7 :: Fractional a => a  6.7 :: Double
16*(-2.1)   -33.6 :: Fractional a => a -33.6 :: Double
7.1/2.4     2.95833 :: Fractional a => a 2.95833 :: Double
9 'mod' 3    0 :: Integral a => a      0 :: Int
8 'div' 4    2 :: Integral a => a      2 :: Int
9 'div' 4    2 :: Integral a => a      2 :: Int
10 'div' 4   2 :: Integral a => a      2 :: Int
11 'div' 4   2 :: Integral a => a      2 :: Int
12 'div' 4   3 :: Integral a => a      3 :: Int
8 'mod' 4    0 :: Integral a => a      0 :: Int
9 'mod' 4    1 :: Integral a => a      1 :: Int
10 'mod' 4   2 :: Integral a => a      2 :: Int
11 'mod' 4   3 :: Integral a => a      3 :: Int
12 'mod' 4   0 :: Integral a => a      0 :: Int
```

Der Potenzoperator `^` erfordert einen ganzzahligen Exponenten, während die Basis irgendeinen Zahlentyp haben kann. Dies wird im Typ von `3^2` angezeigt (entsprechendes gilt für `3^3`):

```
3^2 :: (Integral a, Num b) => b
```

Der Ausdruck `3^2` kann also den Typ `b` haben, sofern `b` ein `Num`-Typ ist. Zusätzlich wird die Voraussetzung gemacht, dass `a` ein `Integral`-Typ ist. Dies bezieht sich auf den Exponenten und wird angegeben, obwohl der Exponent im Ergebniswert nicht mehr vorkommt. Welche der Typvariablen `a` und `b` sich auf Basis und Exponent beziehen, ist in dieser Darstellung allerdings nicht erkennbar. Hierzu muss man den Typ des Potenzierungsoperators selbst ansehen:

```
(^) :: (Num a, Integral b) => a -> b -> a
```

Hier erkennt man, dass das erste Argument (die Basis) und das Ergebnis Num-Typen haben müssen. Das zweite Argument (der Exponent) hat die stärkere Einschränkung, einen `Integral`-Typen haben zu müssen.

Die Typen der letzten drei Ausdrücken verwenden die Typklasse `Floating`, zu der `Float` und `Double` gehören. Die in `Fractional` enthaltenen Typen für Brüche (in Bruchdarstellung) werden dadurch ausgeschlossen.

```
pi          3.14159 :: Floating a => a    3.14159 :: Double
sin (pi/2)  1.0    :: Floating a => a    1.0    :: Double
exp 1       2.71828 :: Floating a => a    2.71828 :: Double
```

Aufgabe 2: Bei den Typangaben zu diesen Ausdrücken ist zu beachten, dass `String` ein Synonym für `[Char]` (Liste von `Chars`) ist. Beide Angaben sind gleichwertig.

```
'H'          'H' :: Char
"Hallo"      "Hallo" :: String
'H':"allo"    "Hallo" :: [Char]
"Hallo "++"du" "Hallo du" :: [Char]
head "Hallo"  'H' :: Char
tail "Hallo"  "allo" :: [Char]
head (tail "hallo") 'a' :: Char
ord 'a'       97 :: Int
ord 'b'       98 :: Int
ord 'c'       99 :: Int
ord 'z'       122 :: Int
ord 'A'       65 :: Int
ord 'Z'       90 :: Int
chr 97        'a' :: Char
chr 98        'b' :: Char
5 == 7        False :: Bool
4 == 4        True  :: Bool
True          True  :: Bool
not True      False :: Bool
(5==7)&&(4==4) False :: Bool
(5==7)|| (4==4) True  :: Bool
True && False  False :: Bool
False || True  True  :: Bool
```

Aufgabe 3: Die Funktionen `div` und `mod` dienen der Ganzzahldivision. Dabei liefert `div` den ganzzahligen Quotienten und `mod` den Divisionsrest.

Die Funktionen `head` und `tail` können auf Zeichenketten angewendet werden. Dann liefert `head` den ersten Buchstaben (also ein `Char`) und `tail` den Rest ohne den ersten Buchstaben (also ein `String`). Später wird die allgemeinere Bedeutung von `head` und `tail` besprochen werden: Beide Funktionen können nicht nur auf Listen von Zeichen, sondern auf irgendwelche Listen angewendet werden, von denen sie dann das erste Element bzw. den Rest der Liste liefern.

Die Funktionen `ord` und `chr` dienen der gegenseitigen Zuordnung von Nummern und Zeichen des ASCII-Zeichencodes.

Die grundlegenden logischen (Booleschen) Funktionen sind `not` (nicht), `&&` (und) und `||` (oder).

Der Hochpfeil `^` dient der Potenzierung einer Zahl mit einer ganzen Zahl, `sin` und `exp` berechnen für Gleitkommazahlen den Sinus bzw. die Exponentialfunktion.

A.2 Lösungen zu Kapitel 4

Aufgabe 1: Bei den Funktionen, bei denen mit Zahlen gerechnet wird, ist die explizite Typangabe nötig, um den gewünschten (und nicht einen allgemeineren) Typ zu erhalten. Die Wirkung der Funktionen ist in der Aufgabenstellung nicht ausdrücklich beschrieben. Die Intention sollte sich jedoch aus den Namen der Funktionen ergeben.

```
plus5 :: Float -> Float
plus5 x = x+5
```

```
zum_quadrat :: Int -> Int
zum_quadrat x = x*x
```

```
ist_gleich_7 :: Int -> Bool
ist_gleich_7 x = (x==7)
```

```
groesser_null :: Int -> Bool
groesser_null x = (x>0)
```

```
erster_buchstabe :: String -> Char
erster_buchstabe text = head text
```

```
dritter_buchstabe :: String -> Char
dritter_buchstabe text = head (tail (tail text))
```

Aufgabe 2: Für jeden der Ausdrücke ist ein gültiger Typ angegeben:

```
5 `div` 2 :: Int
tail "Informatik" :: String
(5 == 3) :: Bool
(5 == 3) || (2 < 4) :: Bool
```

Die weiteren Ausdrücke sind Funktionsdefinitionen und haben daher Typen mit einem Pfeil.

```
dritter text = head (tail (tail (text))) :: String -> Char
kleiner_drei zahl = (zahl < 3) :: Int -> Bool
plus_sieben zahl = zahl + 7 :: Int -> Int
```

Aufgabe 3:

```
zeitaddieren :: Int -> Int -> Int
zeitaddieren zeit1 zeit2 = mod (zeit1 + zeit2) 24
```

Aufgabe 4:

```
maximum :: Int -> Int -> Int
maximum x y = if (x>y) then x else y
```

oder

```
maximum :: Int -> Int -> Int
maximum x y
  | x>y      = x
  | otherwise = y
```

Einer der zahlreichen Definitionen für das Maximum von vier Werten ist:

```
maxvonvier :: Int -> Int -> Int -> Int -> Int
maxvonvier a b c d = maximum (maximum a b) (maximum c d)
```

Aufgabe 5:

```
exkl_oder :: Bool -> Bool -> Bool
exkl_oder a b = (a /= b)
```

Aufgabe 6:

```
istgleichdrei :: Int -> Bool
istgleichdrei 3      = True
istgleichdrei sonstwas = False
```

Aufgabe 7: Diese Lösung gibt erst die Muster für das Ergebnis `False` an. Trifft keiner dieser beiden Fälle zu, wird automatisch der dritte Fall mit dem Ergebnis `True` verwendet, der die anderen beiden Kombinationen von Eingabewerten erfasst:

```
exkl_oder :: Bool -> Bool -> Bool
exkl_oder True True    = False
exkl_oder False False = False
exkl_oder x y          = True
```

Die folgende Definition funktioniert *nicht*, da Haskell nicht die Intention erkennt, dass zwei gleich benannte Argumente (`x`) für denselben Eingabewert stehen sollen:

```
exkl_oder :: Bool -> Bool -> Bool
exkl_oder x x = False      --geht nicht
exkl_oder x y = True
```

Aufgabe 8:

```
istvokal :: Char -> Bool
istvokal 'a' = True
istvokal 'e' = True
istvokal 'i' = True
istvokal 'o' = True
istvokal 'u' = True
istvokal sonst = False
```

Zu beachten ist, dass in den Mustern die Vokale in einfache Anführungszeichen eingeschlossen sein müssen, um nicht als Variable, sondern als die angegebenen Buchstaben erkannt zu werden.

Aufgabe 9: Die Funktion `not` bewirkt die Boolesche Negation, vertauscht also `True` und `False`.

```
my_not :: Bool -> Bool
my_not True  = False
my_not False = True
```

Aufgabe 10:

```

istBuchstabe :: Char -> Bool
istBuchstabe x
  | ('a'<=x && x<='z') = True
  | ('A'<=x && x<='Z') = True
  | otherwise          = False

```

Aufgabe 11:

```

summevon7bis :: Int -> Int
summevon7bis 7 = 7
summevon7bis n = n + summevon7bis (n-1)

```

Aufgabe 12: Die eingegebene Zeichenkette ist zu untersuchen, bis ein 'a' vorkommt. Dann kann man das Ergebnis `True` ausgeben, ohne den Rest der Kette zu untersuchen. Kommt kein 'a' vor, so wird man die Kette bis hin zur leeren Zeichenkette "" abarbeiten. Ist man dort angekommen, so ist das Ergebnis `False`.

```

a_test :: String -> Bool
a_test ""          = False      --leere Kette
a_test ('a':rest) = True       --vorne ein a
a_test (x:rest)   = a_test rest --vorne kein a

```

Aufgabe 13: Die Funktion muss ihre Eingabe bis zur Null hinunterzählen und in jedem Schritt ein Sternchen ausgeben.

```

sternchen :: Int -> String
sternchen 0 = ""
sternchen n = '*':(sternchen (n-1))

```

Aufgabe 14:

```

summevon7bis :: Int -> Int
summevon7bis n = hsummevon7 n 0

hsummevon7 :: Int -> Int -> Int
hsummevon7 7 acc = acc + 7
hsummevon7 n acc = hsummevon7 (n-1) (acc+n)

```

Aufgabe 15: Man hat in dieser Aufgabe die Wahl, welches der beiden Argumente man zum Zählen und welches man als Summanden nimmt. Hier wird das zweite Argument `y` wiederholt addiert, die Anzahl der Summanden wird durch das erste Argument `x` angegeben.

```

mal :: Int -> Int -> Int
mal 0 y = 0
mal x y = y + (mal (x-1) y)

```

Eine Variante mit Akkumulator ist:

```

malVariante :: Int -> Int -> Int
malVariante x y = hmal x y 0

hmal :: Int -> Int -> Int -> Int
hmal 0 y acc = acc
hmal x y acc = hmal (x-1) y (acc+y)

```

Aufgabe 16: Die angegebenen Funktionen tun das, was man entsprechend ihrer Namensgebung erwartet. Nicht ganz offensichtlich ist das bei der Funktion `quadrate`, die zu einer Zahl n die ersten n ungeraden Zahlen summiert, im Beispiel:

$$\text{quadrate } 4 = 1 + 3 + 5 + 7 = 16 .$$

Es lässt sich beweisen, dass so das Quadrat der eingegeben Zahl n berechnet werden kann.

Aber auch ohne dieses Wissen kann man durch rekursives Einsetzen in die Funktion die Funktionswerte von `quadrate` bestimmen (es ist hier sogar erwünscht, so vorzugehen, damit man die Wirkung der rekursiven Definition erkennt).

Die Ergebnisse sind:

```
groesser_test 8 3 = True
groesser_test 3 5 = False
produkt 9 (-2)   = -18
oder True False = True
quadrate 0       = 0
quadrate 1       = 1
quadrate 4       = 16
```

Die Typen der Funktionen sind zum Beispiel:

```
groesser_test :: Int -> Int -> Bool
produkt :: Int -> Int -> Int
oder :: Bool -> Bool -> Bool
quadrate :: Int -> Int
```

A.3 Lösungen zu Kapitel 5

Aufgabe 1: Dargestellt sind mögliche Eingaben in Hugs zusammen mit der Ausgabe von Hugs, wenn die Angabe von Typen eingeschaltet ist.

```
> 3 : 9 : 8 : []
[3,9,8] :: [Integer]

> "Haskell" : "macht" : "Spass" : []
["Haskell","macht","Spass"] :: [[Char]]
```

Da die einzelnen Strings auch Listen sind, könnte man auch diese einzeln aufbauen. Das wird sehr umständlich und wird hier nur angedeutet:

```
> ('H': 'a': 's': 'k': 'e': 'l': 'l': []) : "macht" : "Spass" : []
["Haskell","macht","Spass"] :: [[Char]]
```

Zur Darstellung der leeren Liste kann man keinen Doppelpunkt verwenden:

```
> []
ERROR: Cannot find "show" function for:
*** Expression : []
*** Of type    : [a]
```

Da die leere Liste hier nicht in einem Zusammenhang steht, der ihren Typ bestimmt, kann der Wert der leeren Liste nicht ausgegeben werden. Dennoch wird (in der Fehlermeldung) ein Typ angegeben, nämlich der polymorphe Typ `[a]`. Das Problem

mit der „show“-Funktion ist dies: Für viele Typen gibt es eine solche Funktion, die verwendet wird, um Werte auf dem Bildschirm (letztlich als Zeichenkette) anzuzeigen. Solange der Typ aber unklar ist ([a] sagt ja nur, dass es sich um irgend einen Listentyp handelt), weiss man nicht, welche „show“-Funktion verwendet werden soll.

Die Klammern im nächsten Ausdruck dienen der Erzeugung der Paare:

```
> (3, "Tomaten") : (9, "Birnen") : []
[(3,"Tomaten"),(9,"Birnen")] :: [(Integer,[Char])]
```

Aufgabe 2:

1. ["Das ", "ist ", 3, "mal ", "keine ", "Liste"] ist keine korrekte Haskell-Liste, weil die Listeneinträge verschiedene Typen haben. Das ist jedoch in Haskell nicht möglich.
2. [[1,3], [4,1], [], [9,3,7]] :: [[Int]]
3. `eines :: [a] -> Bool`
`eines (x:[]) = True`
`eines sonstwas = False`

Das Muster für die einelementige Liste kann man alternativ auch so angeben:

```
eines [x] = True
```

Aufgabe 3: Die Funktionen verwenden den Mustervergleich, um die Bestandteile einer Liste gemäß ihrem Aufbau mit dem Konstruktor `:` (Doppelpunkt) wieder zu isolieren:

```
kopf :: [a] -> a
kopf (x:xe) = x
```

```
rumpf :: [a] -> [a]
rumpf (x:xe) = xe
```

Aufgabe 4: Der Mustervergleich filtert zunächst das Muster `[]` heraus, für das das Ergebnis `True` ist. Für alle anderen Fälle ist das Ergebnis `False`, denn in den anderen Fällen muss es sich um eine zusammengesetzte Liste handeln.

```
ist_leer :: [a] -> Bool
ist_leer [] = True
ist_leer sonst = False
```

Aufgabe 5:

1. Die Funktion verwendet den Operator `++`, um die Teilstrings aneinander zu hängen.

```
satz :: [String] -> String
satz [] = []
satz (x:xe) = x++(satz xe)
```

Eine Variante der Funktion, die einen Akkumulator verwendet, sieht so aus:

```

satz :: [String] -> String
satz wortliste = hsatz wortliste ""

hsatz :: [String] -> String -> String
hsatz [] acc    = acc
hsatz (x:xe) acc = hsatz xe (acc++x)

```

2. Es ist die Typangabe für die Funktion zu ändern, und die leere Liste ist als [] (nicht "") zu schreiben.

```

satz :: [[a]] -> [a]
satz wortliste = hsatz wortliste []

hsatz :: [[a]] -> [a] -> [a]

```

Der Rest des Programms kann unverändert übernommen werden.

3. `einzelne :: [a] -> [[a]]`
`einzelne [] = []`
`einzelne (x:xe) = [x):(einzelne xe)`
4. In `satz (einzelne liste)` wird zuerst `einzelne liste` ausgewertet. Dies ist die Liste der einelementigen Listen, die die Elemente von `liste` enthalten. Mit `satz` werden diese einelementigen Listen wieder zu einer einzigen Liste zusammengefügt, so dass sich wieder die ursprüngliche `liste` ergibt.
- Ein Beweis dieses Sachverhalts verwendet strukturelle Induktion über den Aufbau der Listen (dies war hier nicht verlangt, wird aber zur Illustration vorgeführt):

Behauptung: Für alle Listen `liste` gilt `satz (einzelne liste) = liste`.

Beweis, Induktionsverankerung: Ist `liste` die leere Liste, so ist

```

satz (einzelne liste) = satz (einzelne [])
                    = satz []
                    = []

```

Beweis, Induktionsschluss: Ist `liste` nicht leer, so ist `liste` von der Form `(x:xe)` und hat die Länge n . Wir nehmen an, die Behauptung sei für Listen der Länge $n - 1$ schon bewiesen. Da `xe` die Länge $n - 1$ hat, gilt dann:

```

satz (einzelne liste)
  = satz (einzelne (x:xe))
  = satz ([x):(einzelne xe)) --Def von einzelne
  = [x]++(satz (einzelne xe)) --Def von satz
  = [x]++xe                    --Annahme
  = (x:xe)                      --Def von [x], ++
  = liste

```

Dies vollendet den Beweis.

Umgekehrt kann `einzelne` natürlich nicht die Struktur zurück gewinnen, die durch die Anwendung von `satz` verloren geht. Beispiel:

```

einzelne (satz [[1,2],3])
  = einzelne [1,2,3]
  = [[1], [2], [3]]

```

5. Die Funktionen `words` und `lines` können nur auf Strings angewendet werden. Sie liefern die Liste der Wörter bzw. Zeilen des Strings, die anhand der Worttrenner (Leerzeichen) bzw. Zeilentrenner (das Zeichen `'\n'`) erkannt werden. Diese Trennzeichen gehen dabei verloren.

Umgekehrt fügen `unwords` und `unlines` Listen von Strings wieder zu einem String zusammen, wobei an den Nahtstellen des Zusammenfügens die entsprechenden Trennzeichen hinzu gefügt werden.

Von `satz` unterscheiden sich die `unwords` und `unlines` dadurch, dass sie die Trennzeichen hinzu fügen. Der Unterschied zwischen `einzelne` und `words` bzw. `lines` besteht in der Granularität der erzeugten Listen (und dem Entfernen von Trennzeichen).

Aufgabe 6:

```
istIn :: Eq a => a -> [a] -> Bool
istIn element [] = False
istIn element (x:xe)
  | element==x    = True
  | otherwise     = istIn element xe
```

Aufgabe 7: Eine leere Liste hat keinen größten Wert, daher wird für die leere Liste eine Fehlermeldung ausgegeben. Das wurde in dieser Aufgabe nicht erwartet, zeigt aber am Beispiel, wie ggf. eine Fehlerbehandlung erfolgen kann. Mehr zum Thema Fehlerbehandlung wird in einem späteren Kapitel besprochen.

In der einelementigen Liste `[x]` ist `x` das Maximum. Hat die Liste mehr Elemente, so werden jeweils die zwei ersten verglichen und das kleinere gestrichen. Von der verbleibenden Liste wird mit dem rekursiven Aufruf der Funktion das Maximum gesucht. Dadurch wird an der ersten Position der Liste das Ergebnis „angesammelt“, der Listenkopf fungiert hier als impliziter Akkumulator. Wir zeigen das am Beispiel:

```
listenmaximum [6,3,7]
  = listenmaximum [6,7]
  = listenmaximum [7]
  = 7
```

Eine Variante mit explizitem Accumulator ist:

```
listmax :: [Int] -> Int
listmax [] = error "Leere Liste hat kein Maximum"
listmax liste = hlistmax (head liste) liste

hlistmax :: Int -> [Int] -> Int
hlistmax acc [] = acc
hlistmax acc (x:xe)
  | acc>x      = hlistmax acc xe
  | otherwise  = hlistmax x xe
```

Beim Aufruf der Funktion `hlistmax` muss man entscheiden, welchen Startwert der Akkumulator haben soll. Der Wert 0 ist hier nur geeignet, wenn sicher ist, dass in der Liste positive Zahlen vorkommen. Man könnte auch die kleinste im Rechner darstellbare `Int`-Zahl verwenden. Da ein Listenmaximum aber nur von nicht leeren Listen bestimmt werden kann, ist das erste Element eine natürliche Wahl, die von der Darstellung der Zahlen im Rechner unabhängig ist.

Aufgabe 8:

1. `vonbis :: Int -> Int -> [a] -> [a]`
`vonbis n m liste = take (m-n+1) (drop (n-1) liste)`

Ist $n > m$, so ist das Ergebnis die leere Liste. Hat die Liste zu wenige Elemente, so wird der Teil der Liste ins Ergebnis übernommen, der innerhalb der gewünschten Grenzen liegt, das Ergebnis hat also weniger Elemente als es durch die Grenzen n und m angegeben ist.

2. Bei beiden Funktionen ist zu beachten, dass zwei Basisfälle nötig sind: im ersten sind 0 Elemente zu nehmen oder weg zu lassen, im zweiten ist die betrachtete Liste leer.

```
nimm :: Int -> [a] -> [a]
nimm 0 liste = []
nimm n []    = []
nimm n (x:xe) = x:(nimm (n-1) xe)
```

```
lass :: Int -> [a] -> [a]
lass 0 liste = liste
lass n []    = []
lass n (x:xe) = lass (n-1) xe
```

Aufgabe 9:

1. Die Definition enthält zwei Basisfälle. Einerseits ist die leere Liste Anfang jeder Liste. Andererseits kann eine Liste, die nicht leer (also von der Form $x:xe$) ist, nicht Anfang der leeren Liste sein. Im rekursiven Fall müssen die Listenköpfe übereinstimmen und der Rumpf der ersten Liste ein Anfang des Rumpfes der zweiten Liste sein, damit die erste Liste ein Anfang der zweiten ist.

```
anfangVon :: Eq a => [a] -> [a] -> Bool
anfangVon [] liste = True
anfangVon (x:xe) [] = False
anfangVon (x:xe) (y:ys)
  | x==y          = anfangVon xe ys
  | otherwise     = False
```

2. In der folgenden Definition kommt es auf die Reihenfolge der Basisfälle an. Der erste Basisfall deckt ab, dass keine Liste kürzer als die leere Liste ist. Das Muster im zweiten Fall würde auch zum Fall zweier leerer Listen passen, von denen jedoch die eine nicht kürzer als die andere ist. Der Vergleich zweier leerer Listen wird aber schon vom ersten Basisfall abgedeckt, so dass dieses zweite Muster nur verwendet wird, falls die zweite Liste nicht leer ist. Dann ist die erste (leere) Liste tatsächlich kürzer als die zweite (nicht leere). Im rekursiven Fall wird von beiden Listen je ein Element entfernt, und es wird geprüft, ob der erste Listenrumpf kürzer als der zweite ist.

```
kuerzerAls :: [a] -> [b] -> Bool
kuerzerAls liste [] = False
kuerzerAls [] liste = True
kuerzerAls (x:xe) (y:ys) = kuerzerAls xe ys
```

Aufgabe 10:

```
dreiersumme_curry :: Int -> Int -> Int -> Int
dreiersumme_curry a b c = a+b+c
```

```
dreiersumme_uncurry :: (Int, Int, Int) -> Int
dreiersumme_uncurry (a,b,c) = a+b+c
```

Aufgabe 11: Diese Aufgabe zeigt, dass bei mehreren Eingaben zwischen curry- und uncurry-Version gewählt werden kann. Mehrere Ausgaben sind aber immer zu einem strukturierten Ausgabewert zusammen zu fassen, da Funktionen stets nur eine Ausgabe haben.

```
grundschuldiv_curry :: Int -> Int -> (Int, Int)
grundschuldiv_curry a b = (div a b, mod a b)
```

```
grundschuldiv_uncurry :: (Int, Int) -> (Int, Int)
grundschuldiv_uncurry (a,b) = (div a b, mod a b)
```

Aufgabe 12:

```
quadrat :: Int -> Int
quadrat x = x*x
```

```
quadratsumme_curry :: Int -> Int -> Int
quadratsumme_curry x y = quadrat x + quadrat y
```

```
quadratsumme_uncurry :: (Int, Int) -> Int
quadratsumme_uncurry (x,y) = quadrat x + quadrat y
```

Aufgabe 13: Liegt in einem der Teilbereiche keine Zahl der Liste, so wird die entsprechende Listensumme auf 0 gesetzt. Damit ist für die leere Liste auch das Ergebnis (0,0) sinnvoll.

Die Funktion verwendet eine Hilfsfunktion zum komponentenweisen Addieren zweier Paare.

```
kleineGrosseSummen :: [Float] -> (Float, Float) -> (Float, Float)
kleineGrosseSummen [] = (0, 0)
kleineGrosseSummen (x:xe)
  | x<0           = kleineGrosseSummen xe
  | x<=1         = addPaar (x, 0) (kleineGrosseSummen xe)
  | otherwise     = addPaar (0, x) (kleineGrosseSummen xe)
```

```
addPaar :: (Float, Float) -> (Float, Float)
addPaar (a,b) (c,d) = (a+c, b+d)
```

Aufgabe 14: Die Folge konvergiert gegen $\sqrt{5}$. Das Programm implementiert das Heronsche Verfahren zum näherungsweise Wurzelziehen, wobei der Radikand fest angegeben ist (hier 5, man kann den Radikand aber leicht variabel lassen, indem man ihn als weiteres Funktionsargument angibt).

```
folge :: Int -> Float
folge 0 = 1
folge n = 0.5 * (folge (n-1)+ 5/(folge (n-1)))
```

Diese Definition der Funktion `folge` ist im Übrigen sehr ineffizient, da beim Aufruf von `folge n` der Wert von `folge (n-1)` zweimal berechnet wird. Durch den rekursiven Aufruf ergibt sich hier ein exponentieller Aufwand. Dies lässt sich umgehen, indem man einen `where`-Ausdruck verwendet (was jedoch erst später besprochen wird). Da das Heron-Verfahren äußerst schnell konvergiert, erhält man schon mit kleinem `n` gute Näherungen für die Wurzel, so dass die ungünstige Programmierung nicht ganz so schlimm ist.

Aufgabe 15:

1.
 - `(9, "hallo", 'x') :: (Int, String, Char)`
 - `(3.4, [1, 2, 3]) :: (Float, [Int])`
 - `[(2, 'j'), (9, 'A')] :: [(Int, Char)]`
 - `(45, (3, "info")) :: (Int, (Int, String))`
2. `mittleres :: (a, b, c) -> b`
`mittleres (x, y, z) -> y`

A.4 Lösungen zu Kapitel 6

Aufgabe 1:

```
satz :: [[a]] -> [a]
satz = foldl (++) []
```

```
istIn :: Eq a => a -> [a] -> Bool
istIn a liste = [] /= filter (==a) liste
```

```
nochEinIstIn :: Eq a => a -> [a] -> Bool
nochEinIstIn a = foldl (||) False . map (==a)
```

Für das `listenmaximum` wird die Funktion `foldl1` verwendet, bei der kein „Anfangsmaximum“ benötigt wird.

```
listenmaximum :: Ord a => [a] -> a
listenmaximum = foldl1 max
```

Zum Bilden der „kleinen“ und „großen“ Summen wird eine Hilfsfunktion benötigt, die zu einem Summenpaar einen einzelnen Wert „addiert“.

```
klGrAdd :: (Float, Float) -> Float -> (Float, Float)
klGrAdd (k,g) x
  | x<0      = (k,g)
  | x<=1     = (k+x, g)
  | otherwise = (k, g+x)
```

```
kleineGrosseSummen :: [Float] -> (Float, Float)
kleineGrosseSummen
  = foldl klGrAdd (0,0)
```

Aufgabe 2: Hier kommt ein Programm, das funktioniert. Leider schaffe ich es in dieser Woche nicht mehr, die nötigen Erklärungen dazu zu schreiben.

Natürlich kann man die Aufgabe auch auf viele andere Weisen lösen, mit mehr Funktionen höherer Ordnung oder auch ohne. Wer die Aufgabe nicht komplett lösen konnte, soll bitte nicht traurig sein, vielleicht hat er oder sie doch Teile geschafft

(die sich ja auch separat testen lassen). Die Aufgabe war nicht nur dafür gedacht, dass man sie vielleicht komplett löst, sondern auch dafür, dass man ahnen kann, dass mit Haskell auch „richtige“ Programme geschrieben werden können (die für eine Anwendung gut sind).

```
-----
--- Zu Punkt 1                                     ---
-----
```

```
buchstaben_test :: Char -> Bool
buchstaben_test x
  | 'a'<=x && x<='z'   = True
  | 'A'<=x && x<='Z'   = True
  | otherwise          = False

gross_nach_klein :: Char -> Char --nur fuer Buchstaben
gross_nach_klein x
  | 'a'<=x && x<='z'   = x          --nichts zu aendern
  | 'A'<=x && x<='Z'   = chr (32 + ord x)
  | otherwise          = error "gross_nach_klein undefiniert"

kleine_ohne_sonderzeichen :: String -> String
kleine_ohne_sonderzeichen liste
  = map gross_nach_klein (filter buchstaben_test liste)
```

```
-----
--- Zu Punkt 2                                     ---
-----
```

```
paar_erzeugen :: Char -> (Char, Int)
paar_erzeugen x = (x, 1)

paare_erzeugen :: String -> [(Char, Int)]
paare_erzeugen = map paar_erzeugen

paare_zusammen :: [(Char, Int)] -> [(Char, Int)]
paare_zusammen liste = foldr paar_einfuegen [] liste

paar_einfuegen :: (Char, Int) -> [(Char, Int)] -> [(Char, Int)]
paar_einfuegen (x,n) [] = [(x,n)]
paar_einfuegen (x,n) ((y,m):rest)
  | x==y          = (x, n+m):rest
  | otherwise      = (y,m):(paar_einfuegen (x,n) rest)
```

```
-----
--- Zu Punkt 3                                     ---
-----
```

```
einfuegen :: (a -> a -> Bool) -> a -> [a] -> [a]
einfuegen kleiner y [] = [y]
einfuegen kleiner y (x:xe)
  | kleiner y x      = y:x:xe
```

```

| otherwise          = x:(einfuegen kleiner y xe)

sortiere :: (a -> a -> Bool) -> [a] -> [a]
sortiere kleiner liste = hsortiere kleiner liste []

hsortiere :: (a -> a -> Bool) -> [a] -> [a] -> [a]
hsortiere kleiner [] zielliste
    = zielliste
hsortiere kleiner (x:xe) zielliste
    = hsortiere kleiner xe (einfuegen kleiner x zielliste)

paarvergleich :: Ord b => (a,b) -> (a,b) -> Bool
paarvergleich (x1,y1) (x2,y2) = (y2<y1)

paarlisten_sort :: [(Char, Int)] -> [(Char, Int)]
paarlisten_sort = sortiere paarvergleich

-----
--- Zu Punkt 4                                     ---
-----

formatiere_paar :: (Char, Int) -> String
formatiere_paar (x, n) = x:( "  "++(show n)+"\n")

formatiere_liste :: [(Char, Int)] -> String
formatiere_liste liste
    = foldl1 (++) [] (map formatiere_paar liste)

-----
--- Zu Punkt 5                                     ---
-----

textAnalyse :: String -> String
textAnalyse = formatiere_liste . paarlisten_sort .
              paare_zusammen . paare_erzeugen .
              kleine_ohne_sonderzeichen

-----
--- Anwenden auf Textdatei                         ---
-----

anwenden :: (String -> String) -> FilePath -> FilePath -> IO()
anwenden programm einDatei ausDatei
    = readFile einDatei
      >>= \text -> writeFile ausDatei (programm text)

main :: FilePath -> FilePath -> IO()
main = anwenden textAnalyse

-----
--- Entschuldige mich fuer die fehlenden Erklaerungen ---
--- Vielleicht kommen die gelegentlich noch           ---
-----

```