School of λ

Haskell für Einsteiger

Julius Angres

29. Mai 2020

Version α

Inhaltsverzeichnis

1 Vorwort

Dieses Buch zu den Grundlagen der Programmierung in HASKELL richtet sich an jeden, der diese funktionale Programmiersprache in den Grundzügen erlernen möchte. Es richtet sich insbesondere, aber nicht nur, an alle Interessierten ohne Vorkenntnisse in der Programmierung bzw. Softwareentwicklung. Dies können Schüler, Studenten, Lehrer oder Hobbyprogrammierer gleichermaßen sein.

Mit vielen motivierenden Beispielen und Übungen zur Selbstkontrolle eignet sich das Buch genau für ein Selbststudium wie auch als Nachschlagewerk oder als Grundlage der Unterrichtsplanung für Lehrkräfte.

Dieses Buch erhebt weder den Anspruch perfekter formaler Korrektheit noch den einen vollständigen Überblick über die Möglichkeiten der Softwareentwicklung mit HASKELL zu bieten. Viele mächtige Konzepte wie beispielsweise Funktionen höherer Ordnung oder Monaden werden zwar erwähnt und beispielhaft verwendet, allerdings ohne einen fundierten theoretischen Unterbau zu diesen weiterführenden Themen zu liefern. Der interessierte und fortgeschrittene Leser möge sich zu diesem Zweck der vielfältigen im Internet frei verfügbaren Skripte von universitären Veranstaltungen bedienen. Einen reichhaltigen Überblick über die praktische Umsetzung auch fortgeschrittener Konzepte bieten auch die beiden Standardwerke *Learn You A Haskell For Great Good*[1] sowie *Real World Haskell*[2]. Hierbei handelt es sich jedoch (mehr oder weniger klassische) Fachbücher, die aus Sicht des Autors für den Einstieg insbesondere für Schüler ohne einschlägige Vorerfahrungen kaum geeignet sind.

[1] PDF online frei verfügbar unter `http://learnyouahaskell.com/`
[2] PDF online frei verfügbar unter `http://book.realworldhaskell.org/`

2 Einleitung

2.1 Vorbereitungen

Zum Start in die Programmierung mit HASKELL sind nur sehr wenige Vorbereitungen erforderlich. Grundsätzlich gibt es zwei verschiedene Möglichkeiten, um zu starten.

- Der schnelle Weg: Aufrufen der Seite `https://repl.it/languages/haskell` mit einem beliebigen Webbrowser auf einem beliebigen Gerät. Da das Ausführen der Programme nicht auf dem eigenen Gerät erfolgt, ist keinerlei Installation notwendig. Der Zugriff kann mit einem PC genauso wie mit einem Tablet oder Smartphone erfolgen.
- Der vollständige Weg: Herunterladen und Installieren der benötigten Programme von der offiziellen Quelle `https://www.haskell.org/platform/`. Hier wählt man sein Betriebssystem aus und folgt einfach den Anleitungen bzw. Schritten der Installationsassistenten.[3]

2.2 Merkmale funktionaler Programmierung

Aufgrund ihrer speziellen Merkmale, die den Einstieg sehr einfach machen, ist die funktionale Programmierung sehr gut für Einsteiger geeignet. Es müssen sehr weniger Befehle, Schlüsselwörter oder syntaktische Besonderheiten erlernt werden, sodass ein erfolgreicher Einstieg mit den ersten kleinen Programmen in der Regel rasch möglich ist. Leser, die bereits Erfahrungen in der imperativen Programmierung z.B. mit `C`, `Python` oder `Java` gemacht haben, werden ebenfalls schnell in die funktionale Welt einsteigen können. Das Verständnis funktionaler Konzepte kann auch hilfreich beim Verbessern des eigenen Programmierstils in der imperativen Welt sein, sodass vermutlich jeder einen Gewinn aus dem Erlernen von HASKELL ziehen kann.

Auch wenn spezielle Vorkenntnisse aus der Programmierung keinesfalls erforderlich sind, ist es doch hilfreich, wenn der Begriff und das Konzept der *Funktion* und der *Variable* bereits aus dem Mathematikunterricht bekannt sind. Dies sollte aber (je nach Schulart, Bundesland, etc.) spätestens am Ende der achten Klasse der Fall sein.[4]

Wenn die einleitenden Beispiele und Hintergründe (noch) zu schwierig erscheinen, können sie auch erst einmal nur überflogen werden. Zu einem späteren Zeitpunkt können dann Details durch nochmaliges Lesen erkannt und verstanden werden. Zur Bearbeitung der ersten praktischen Übungen ist ein detailliertes Verständnis dieses Theorieteils nicht notwendig.

2.3 Funktional vs. imperativ

Funktionale Programmierung unterscheidet sich in einer Reihe von Aspekten von *konventioneller*, imperativer Programmierung. Wir wollen in dieser Einleitung auf einige dieser Unterschiede eingehen und den möglichen Mehrwert des funktionalen Programmierparadigmas aufzeigen. Dazu untersuchen wir zunächst ein ganz einfaches Beispiel für eine Funktion. Ziel ist es eine Funktion zu schreiben, die zwei ganzzahlige Werte als Eingabe hat und diese vertauscht. Hier zunächst eine typisch imperative Lösung des Problems in der Programmiersprache `Java`.

```
void swap(int x, int y)
{
  int tmp = x;
  x = y;
  y = tmp;
}
```

In anderen imperativen Sprachen sieht die Lösung des Problems abgesehen von kleinen syntaktischen Änderungen in der Regel sehr ähnlich aus. Es wird genau beschrieben, *wie* die Eingabewerte x und y wann verändert werden müssen, um das gewünschte Ergebnis zu erzielen. Anders dagegen eine typisch funktionale Lösung in HASKELL.

[3] Für Debian-basierte Linux-Distributionen kann die Installation auch einfach per `apt` mit dem Kommando `sudo apt-get install haskell-platform -yes` aus der Shell heraus erfolgen.

[4] Ein weiterer Vorteil kann Erfahrung im Umgang mit Computer-Algebrasystemen (CAS) sein.

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

Die Funktion beschreibt nur *was* mit den Eingabewerten geschieht, aber nicht wie das zu erfolgen hat.

Es fällt auf, dass die funktionale Lösung nicht nur kürzer, sondern auch verständlicher ist als die imperative Lösung. Es muss nicht jeder Rechenschritt, sondern nur die Definition verstanden werden. Das Programm entspricht letztlich eine symbolischen Darstellung dessen, wie wir Vertauschen mündlich beschreiben würden, etwa: "Nimm zwei Werte und vertausche ihre Reihenfolge."
Ein weiterer Unterschied besteht in der Flexibilität der Lösungen. Die imperative Lösung spricht ausdrücklich von Variablen vom Typ `int` (kurz für Integer, d.h. ganze Zahlen). Mit dieser Funktion können also nur die Werte von Ganzzahlen vertauscht werden, aber z.B. nicht von rationalen Zahlen, Zeichen, Zeichenketten, Tupeln oder anderen, komplexeren Objekten. Für jede Sorte von Objekten müsste die Funktion erneut implementiert werden. Die funktionale Lösung dagegen macht keine Aussage über die Datentypen der Eingabewerte, sondern funktioniert für beliebige (auch zwei unterschiedliche) Typen. Dadurch, dass beschrieben wird, was Vertauschen bedeutet, aber nicht, wie es durchgeführt werden soll, ist die Lösung sehr flexibel. Man spricht hier von Polymorphismus.

Man erkennt, dass die beiden Lösungen zwar dasselbe Problem lösen, jedoch gänzlich unterschiedliche Herangehensweisen offenbaren.

- Imperative Programme beschreiben genau *WIE* ein Problem in einzelnen Schritten nacheinander gelöst werden soll. Die Reihenfolge der Befehle spielt bei der Auswertung eine entscheidende Rolle. Ein imperatives Programm ist eine Sequenz von Befehlen.
- Funktionale Programme beschreiben *WAS* genau das Problem ist. Die Reihenfolge der Auswertung spielt hierbei keine Rolle. Ein funktionales Programm ist eine Reduktion von Funktionsausdrücken, die zu einem Ergebnis reduziert werden.

Wir betrachten ein weiteres, ein wenig komplizierteres Problem. In einer (Zahlen-)Liste soll die Summe aller geraden Einträge gebildet werden. Auch hier zunächst eine imperative Lösung:

```
int evensum(Int[] arr)
{
 int sum = 0;
 for(int i=0; i<arr.length; i++)
 {
  if(arr[i]%2==0) sum += arr[i];
 }
 return sum;
}
```

Man erkennt wieder deutlich, dass beschrieben wird, WIE (und in welcher Reihenfolge) die Datenstruktur manipuliert werden soll, aber nicht, was das zugrundeliegende Problem, d.h. der Zweck der Funktion ist. Im Gegensatz dazu betrachten wir eine funktionale Lösung:

```
evensum :: [Integer] -> Integer
evensum = (foldl (+) 0) . (filter even)
```

Eigentlich ist der Typ, den der Compiler automatisch bestimmt, sogar noch deutlich allgemeiner:

```
evensum :: (Integral a) => [a] -> a
```

Das bedeutet, dass die Funktion `evensum` für alle Typen definiert ist, die sich wie ganze Zahlen verhalten.

3 Datentypen und Ausdrücke

Jeder Ausdruck in der Programmiersprache Haskell hat einen festen Datentyp (kurz: Typ). Das Typsystem ist statisch und stritk, d.h. der Typ eines Ausdrucks kann sich nicht verändern. Wird der Typ eines Ausdrucks bei der Definition nicht explizit angegeben, so wird er vom Compiler inferiert (engl. *inferred*) und zwar in der allgemeinsten Form, die für den Ausdruck möglich ist.
Die wichtigsten bereits vorhandenen Basisdatentypen sind die folgenden:

- `Bool`. Wahrheitswert. Kann nur die Werte `True` oder `False` annehmen.
- `Int`. Ganze Zahlen aus dem Bereich -2^{31} bis $2^{31} - 1$.
- `Integer`. Ganze Zahlen mit beliebiger Genauigkeit.
- `Float`. Gleitkommazahlen zur Darstellung nichtganzzahliger Werte.
- `Char`. Einzelne Zeichen eingeschlossen in einfachen Anführungszeichen (engl. *single quotes*).
- `String`. Zeichenketten, d.h. Listen von `Char`s zur Darstellung von Text.[5]

Der wichtigste komplexe Datentyp sind Listen. Eine Liste ist eine geordnete Ansammlung von Werten *eines* Typs. Listen können aus beliebigen Typen erstellt werden. Sie werden in Haskell in eckige Klammern eingeschlossen und die Elemente werden mit Kommata abgetrennt.
Eine spezielle Liste ist die leere Liste `[]`. Sie enthält keine Elemente und kann für eine Liste beliebigen Typs stehen. Listen mit genau einem Element werden manchmal auch als *Singleton* bezeichnet.

Ein anderer komplexer Datentyp, der häufig vorkommt, sind Tupel. Tupel werden in runde Klammern eingeschlossen und sind eine Ansammlung von Werten (potenziell) unterschiedlicher Datentypen in einer festen Reihenfolge.

Der Typ eines Wertes oder Ausdrucks kann im GHCi-Prompt oder in der Quelltextdatei explizit angegeben werden, indem hinter dem Ausdruck zwei Doppelpunkte und dann der zu verwendende Typ angegeben werden. Der Typ eines Ausdrucks kann im GHCi-Prompt jederzeit mit `:t <Ausdruck>` abgefragt werden[6].

```
> let n = 3 :: Int
> :t n
> n :: Int

> let x = 3 :: Float
> :t x
> x :: Float

> let z = 3
> :t z
> z :: Num a => a
```

Im dritten Beispiel erkennt man, dass der Typ für `z` automatisch ermittelt wurde. Da kein expliziter Typ angegeben wiurde, wird der Wert `Num a` angenommen. Diese sogenannte Typklasse `Num` umfasst alle Datentypen, die sich wie Zahlenwerte (engl. *numerical values*) verhalten.
In Haskell gibt es eine Vielzahl dieser Typklassen, mithilfe derer Definitionen möglichst allgemein und damit gut wiederverwendbar gemacht werden können.
Wichtige vordefinierte Typklassen sind u.a. die folgenden:

- `Num`. Für alle Zahlenwerte (numerische Werte).
- `Fractional`. Für gebrochene, d.h. nicht ganzzahlige Werte.
- `Eq`. Alle Datentypen, die auf Gleichheit getestet werden können.

[5] Genaugenommen handelt es sich bei `String` schon nicht mehr um einen Basisdatentyp, sondern um einen abgeleiteten Typ, da `String` lediglich ein Typsynonym für `[Char]` ist.
[6] Die Ad-hoc-Definition von Ausdrücken in GHCi erfolgt durch das Schlüsselwort `let`

- `Ord`. Alle Datentypen, die sortiert (geordnet nach einer Ordnungsrelation) werden können.
- `Show`. Alle Datentypen, die ausgegeben, d.h. gedruckt dargestellt werden können.

Die letzten drei Typklassen `Eq`, `Ord` und `Show` beschreiben sehr allgemeine Eigenschaften. Alle Basisdatentypen haben diese Eigenschaften bereits, d.h. wir können z.B. sowohl `Bool, Int, Float, Char` als auch `String` vergleichen, sortieren oder am Prompt anzeigen.

Ein Beispiel für die Nützlichkeit von Typklassen, um Programme besser wiederverwendbar zu machen, betrachten wir im Folgenden.

Beispiel Für einen Sortieralgorithmus soll eine Funktion geschrieben werden, die zwei ganze Zahlen miteinander vertauscht[7]. Eine Realisierung in Java könnte etwa wie folgt aussehen:

```
void swap(int x, int y)
{
 int tmp = x;
 x = y;
 y = tmp;
}
```

Die gleiche Funktion in Haskell mit einer zweielementigen Liste als Eingabewert.

```
swap :: [Int] -> [Int]
swap [x,y] = [y,x]
```

Bei einer Weiterentwicklung sollen nun auch Gleitkommazahlen verarbeitet werden. Dies können wir in Haskell einfach umsetzen, indem wir als Eingabewert nicht `[Int]` verwenden, sondern einen variablen Typ `a`, von dem wir nur fordern, dass es sich um einen numerischen Zahlen (Zahlenwert) handelt.

```
swap :: Num a => [a] -> [a]
swap [x,y] = [y,x]
```

Der Rumpf der Funktion hat sich nicht verändert, sondern nur der Typ der Funktion.

[7] Ein detailliertes Verständnis des Codes ist an dieser Stelle nicht erforderlich. Es geht hier vor allem um die Datentypen

3.1 Übungen

Aufgabe 1

Geben Sie folgende Ausdrücke ein und notieren Sie jeweils das Ergebnis sowie den Typ.

(a) `3+7`
(b) `4-8`
(c) `3*4`
(d) `5/8`
(e) `mod 28 3`
(f) `div 28 3`

Aufgabe 2

Laden Sie die Zeichenbibliothek mit `:m Data.Char` und überprüfen Sie die folgenden Ausdrücke.

(a) `'H'`
(b) `toUpper 'a'`
(c) `head "Haskell"`
(d) `digitToInt '8'`
(e) `ord 'a'`
(f) `chr 65`

Aufgabe 3

Notieren Sie das Ergebnis der folgenden String-Verarbeitungen.

(a) `"Has"++"kell"`
(b) `length "Hello"`
(c) `tail "Haskell"`
(d) `reverse "Haskell"`

Aufgabe 4

Testen Sie die folgenden Boole'schen Ausdrücke und notieren Sie das Ergebnis.

(a) `5 == 7`
(b) `"hi" /= "Hi"`
(c) `pi >= 3`
(d) `2 < sqrt 5`
(e) `True`
(f) `not False`
(g) `(2<3) && (3>5)`
(h) `(2<3) || (3>5)`

Aufgabe 5

Gegeben ist die folgende Funktion `addiere`, die die Summe zweier ganzer Zahlen zurückliefert.

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

(a) Erstellen Sie eine Funktion `addieref`, die zwei Werte vom Typ `Float` addiert.
(b) Begründen Sie, warum die folgenden Aufrufe fehlerhaft sind:

```
> addiere 2.0 1
> addiere 1 (2 :: Float)
> addieref 1 (2 :: Int)
```

(c) Erstellen Sie eine polymorphe Version `addierep` unter Verwendung einer geeigneten Typklasse und testen Sie diese mit den Aufrufen aus der vorherigen Teilaufgabe.
(d) Erläutern Sie, warum die Signatur `addiere :: a -> a -> a` nicht sinnvoll ist.

Aufgabe 6

Geben Sie den allgemeinsten Typ für die Funktion `swap` aus dem Beispiel an, wenn im Sortierverfahren beliebige sortierbare Datentypen verarbeitet werden sollen.

Aufgabe 7

Gegeben ist die Funktion `laenge`, die die Anzahl der Elemente in einer Liste von ganzen Zahlen zurückliefert.

```
laenge :: [Int] -> Int
laenge [] = 0
laenge (x:xs) = 1 + laenge xs
```

Verändern Sie den Typ von `laenge` so, dass

- die Länge beliebiger Zahlenlisten bestimmt wird.
- die Länge von Listen beliebigen Typ bestimmt wird.

Aufgabe 4

Gegeben ist die Funktion `kleinerGleich` mit

```
kleinerGleich x y = x <= y
```

(a) Geben Sie einen Typ der Funktion an, der nur aus Basisdatentypen besteht.
(b) Begründen Sie, zu welcher Typklasse x und y (mindestens) gehören müssen und geben Sie einen entsprechend verallgemeinerten Typ von `kleinerGleich` an.

4 Einfache Funktionen

Die Funktion ist das zentrale Element von funktionalen Programmen. In Haskell erfolgt die Definition von Funktionen in enger Anlehnung an die aus der Schule bekannte mathematische Notation von Funktionen.

Beispiel Gegeben ist die Funktion f, die eine natürliche Zahl als Eingabewert erhält und diesen verdoppelt. Mathematische Notation:
$f : \mathbb{N} \to \mathbb{N}, f(x) = 2 \cdot x$
Eine Funktion besteht aus einem Namen, einem Definitionsbereich, einem Wertebereich und einer (Funktions-)Gleichung.
In der Mathematik konzentriert man sich häufig auf den Namen und die Gleichung einer Funktion und sieht Definitions- und Wertebereich als implizit gegeben an.
Die Definition derselben Funktion in Haskell lautet:

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

Die obere Zeile ist die sogenannte Signatur (engl. type annotation) der Funktion. Sie enthält den oder die Typ(en) des/der Eingabewerte und den Typ des Rückgabewertes. Diese werden vom Funktionsnamen durch zwei Doppelpunkte abgetrennt. In der zweiten Zeile befinden sich der Namen der Funktion, die Bezeichnung des Eingabewertes (hier x) und auf der rechten Seite des Gleichheitszeichens der Rumpf der Funktion.
Eine Signatur anzugeben ist in Haskell nicht verpflichtend, aber hilfreich und nützlich und daher gängige Praxis. Der Typ einer Funktion kann im GHCi-Prompt jederzeit durch Eingabe von `:t <Name>` abgefragt werden, wobei `<Name>` durch den tatsächlichen Namen der Funktion ersetzt werden muss. Hat eine Funktion mehr als einen Eingabewert, dann werden alle Eingabewerte ebenfalls durch Pfeile `->` voneinander abgetrennt. Eine Klammerung ist nicht notwendig.
Die Definition einer Funktion in Haskell darf auch aus mehreren Einzelfällen (oft Regeln genannt) bestehen. Wir betrachten die Vorgängerfunktion auf den natürlichen Zahlen. Für jede natürliche Zahl außer 0 kann der Vorgänger durch Subtraktion von 1 ermittelt werden. Da die Null keinen Vorgänger hat, soll die Funktion hier wieder 0 zurückliefern. Die mathematische Definition dieser Funktion lautet:

$$vorg : \mathbb{N} \to \mathbb{N}, vorg(n) = \begin{cases} n-1 & : n > 0 \\ 0 & : n = 0 \end{cases}$$

Umsetzung in Haskell durch Verwendung zweier Definitionen für die beiden Regeln:

```
vorg :: Int -> Int
vorg 0 = 0
vorg n = n - 1
```

Die erste Regel wird nur angewendet, wenn $n = 0$ ist. In allen anderen Fällen wird die zweite Regel angewendet. Beim Funktionsaufruf sucht das Programm von oben nach unten die erste passende Regel und wendet diese an. Es müssen also ganz oben immer zunächst die Spezialfälle (in der Rekursion oft als Abbruchfälle bezeichnet) und darunter die allgemeinen Fälle definiert werden. Die Definition von Funktionen mit mehreren Fällen erfolgt in Haskell häufig auch mithilfe von sogenannten Guards.

4.1 Spezielle einfache Funktionen

Abhängig von der Anzahl ihrer Eingabewerte oder dem Typ ihres Rückgabewertes gibt es spezielle Bezeichnungen für manche einfachen Funktionen, die häufig vorkommen.

- Funktionen mit zwei Eingabewerten werden *binäre* (von lat. bis: zwei) Funktionen genannt. Beispiele für binäre Funktionen sind etwa die arithmetischen Funktionen `(+),(-),(*),(/),mod,div`, aber auch manche logischen Grundfunktionen wie z.B. `(&&)` oder `(||)`.
- Funktionen mit einem Eingabewert werden *unäre* (von lat. unus: eins) Funktionen genannt. Beispiele für unäre Funktionen sind etwa `(+)` und `(-)` als Vorzeichenfunktionen, die Nachfolgerfunktion `succ` oder die logische Negationsfunktion `not`.

- Funktionen, deren Rückgabewert ein Wahrheitswert vom Typ `Bool` ist, werden *Prädikate* genannt. Diese Funktionen werden verwendet, um Objekte auf das Vorhandensein einer bestimmten Eigenschaft zu prüfen. Prädikate können beliebig viele Eingabewerte haben. Insbesondere sind alle logischen Funktionen Prädikate. Weitere Beispiele für Prädikate sind die Vergleichsoperatoren `(<),(==),(<=)` usw. sowie die Funktionen `even` und `odd` oder die Funktion `elem`.

4.2 Anonyme λ-Funktionen

Oft werden bestimmte Hilfsfunktionen in einem Programm nur an einer bestimmten Stelle und für einen einzigen Zweck benötigt. In diesem Fall wird in Haskell häufig auf eine global sichtbare Definition dieser Funktion verzichtet. Stattdessen wird neben `where`-Klauseln zur lokalen Definition von Funktionen oft eine sogenannte anonyme Lambda-Funktion (häufig kurz als λ-Funktion bezeichnet) verwendet. Der Name stammt vom λ-Kalkül aus der Mathematik, der die Grundlage für funktionale Programmiersprachen bildet. Zur Veranschaulichung der Notation betrachten wir zunächst die bekannte Verdoppelungs- und die Inkrementfunktion. Die gewöhnlichen Definitionen dieser Funktionen sind die folgenden:

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

inc :: Int -> Int
inc n = n + 1
```

In Form von anonymen λ-Funktionen können die Funktionen aber auch wie folgt definiert werden:

```
verdoppelnLam :: Int -> Int
verdoppelnLam = \n -> 2 * n

incLam :: Int -> Int
incLam = \n -> n + 1
```

Anonyme λ-Funktionen werden häufig bei der Verwendung von Funktionen höherer Ordnung benutzt.

Beispiel Jedes Element einer Zahlenliste `xs` soll verdreifacht werden. Es kann hierfür eine Funktion `verdreifachen` definiert werden.

```
verdreifachen :: Num a => a -> a
verdreifachen x = 3 * x

listeVerdreifachen :: Num a => [a] -> [a]
listeVerdreifachen xs = map verdreifachen xs
```

Dieser Ansatz hat jedoch einige Schwachstellen:

- Die Definition der ersten Funktion erfordert gemessen an der Funktionalität relativ viel Code.
- Die Funktion `verdreifachen` ist überall sichtbar.
- Der Code ist unübersichtlich, da zum Verständnis von `listeVerdreifachen` erst der Code der Funktion `verdreifachen` betrachtet werden muss.

Unter Verwendung einer anonymen λ-Funktion können alle diese Nachteile vermieden werden. Der Code für die Listenoperation vereinfacht sich zu[8]:

```
listeVerdreifachenLam :: Num a => [a] -> [a]
listeVerdreifachenLam xs = map (\x -> 3 * x) xs
```

[8] Da es sich um eine einfache, vordefinierte arithmetische Funktion handelt, genügt hier auch `map (3*) xs`

4.3 Übungen

Aufgabe 1

Entfernen Sie in der Definition der Verdoppelungsfunktion die Signatur, werten Sie den Typ aus und erläutern Sie das Ergebnis.

Aufgabe 2

Gegeben sind die folgenden beiden Definitionen der Vorgängerfunktion auf den natürlichen Zahlen:

```
vorg2 :: Int -> Int
vorg2 0 = 0
vorg2 1 = 0
vorg2 n = n - 1

vorg3 :: Int -> Int
vorg3 n = n - 1
vorg3 0 = 0
```

(a) Werten Sie die Ausdrücke `vorg2 0, vorg2 3, vorg3 0` und `vorg3 2` aus.
(b) Entscheiden Sie begründet, welche der Definitionen fehlerhaft ist.

Aufgabe 3

Implementieren Sie die folgenden Funktionen in Haskell:

(a) $f(x) = x + 1$ (Inkrementfunktion)
(b) $g(x) = -2x + 4$
(c) $h(x) = x^2 - 4x + 3$

Aufgabe 4

Implementieren Sie die *Heavyside*-Funktion mit

$$heavy : \mathbb{R} \to \mathbb{N}, heavy(x) = \begin{cases} 0 & : x \leq 0 \\ 1 & : \text{sonst} \end{cases}$$

Aufgabe 5

(a) Schreiben Sie für die folgenden Probleme geeignete Funktionen und testen Sie diese. Alle Funktionen erwarten als Eingabe eine ganze Zahl. Versehen Sie Ihre Funktionen mit einer Typannotation. Sollten Sie den Typ nicht vorab bestimmen können, lassen Sie ihn sich mit `:t` im GHCi anzeigen.
   - `add5 - addiert zu einer Zahl den Wert 5`
   - `hoch3 - berechnet die 3. Potenz einer ganzen Zahl`
   - `inc - erhöht eine Zahl um eins`
   - `hello - liefert den String "Hello World!" zurück`

(b) Schreiben Sie die oberen drei Funktionen mithilfe anonymer λ-Funktionen.
(c) Geben Sie curryfizierte Versionen aller obigen Funktionen an.

Aufgabe 6

Binden Sie ganz zu Beginn der Datei (in der ersten Zeile) die Bibliothek Char durch den Befehl `import Data.Char` in Ihr Skript ein und implementieren Sie folgende Funktionen:

(a) `toNum - wandelt einen Großbuchstaben in eine Zahl um.`
    `Dabei wird 'A' auf 0, 'B' auf 1 usw. abgebildet`
(b) `toChar - liefert das zur Zahl passende Zeichen (Umkehrung von (a))`

Aufgabe 7

Schreiben Sie die folgenden Funktionen:

(a) `verschieben x delta n - addiert delta zu x modulo n`
(b) `zylindervolumen r h - bestimmt das Volumen eines Zylinders (Datentyp Float)`

Aufgabe 8

(a) Schreiben Sie die binären Logikfunktionen Und, Oder, Nicht und Entweder-Oder (XOR) ohne die vordefinierten Funktionen &&, || und *not* zu verwenden. Alle Funktionen sollen vom Typ $Bool \to Bool \to Bool$ (oder noch allgemeiner) sein. Da die Funktionen vordefiniert sind, nennen Sie sie bitte `myand` usw. (Es gibt Bonuspunkte für die kürzeste Lösung von XOR.)
(b) Implementieren Sie auch NAND und IFF, indem Sie die Funktionen der vorherigen Teilaufgabe geeignet kombinieren. (Es gibt Bonuspunkte für die kürzesten Lösungen.)

Aufgabe 9

Realisieren Sie die folgenden Funktionen.

(a) `imkreis - liefert True, wenn ein Punkt innerhalb eines Kreises um den Ursprung mit dem Radius r liegt`
(b) `abstand0 - berechnet den Abstand eines Punktes P vom Ursprung`
(c) `abstand - berechnet den Abstand zweier Punkte in der Ebene`
(d) `istteiler - liefert True, wenn die erste Zahl ein Teiler der zweiten ist`
(e) `istvielfaches . analog`

Aufgabe 10

Werten Sie die folgenden Ausdrücke aus:

(a)
```
(
x -> x + x) 3
```
(b)
```
((
x -> 2 * x) . (
y -> y + 1)) 2
```
(c)
```
(
x -> x) 1
```

Aufgabe 11

Schreiben Sie die folgenden Funktionen als anonyme λ-Funktionen.

(a) `add`; Addition zweier beliebiger Zahlen (entspricht `(+)`)
(b) `xor`; Logisches Entweder-Oder
(c) `rest2`; Division modulo 2
(d) `fun`; mit $fun(x) = x^2 - 4x + 3$
(e) `fun` als Komposition der drei Teilfunktionen

5 Mustererkennung

In Haskell ist es anders als in vielen anderen Programmiersprachen möglich, mithilfe von Mustererkennung (engl. *pattern matching*) auf die innere Struktur von Datenstrukturen direkt zuzugreifen. Da es sich hierbei um ein sehr mächtiges, nützliches Werkzeug handelt, wird es entsprechend häufig verwendet. Durch Mustererkennung können Datenstrukturen also zerlegt werden.
Viele klassische, imperative Programmiersprachen enthalten in neueren Versionen auch zumindest ansatzweise Techniken wie Mustererkennung oder auch Funktionen höherer Ordnung.
Ein besonders einfaches Muster, das wir bereits kennen, sind Zahlenwerte. Dieses Muster haben wir bei den Funktionsdefinitionen, die mehr als einen Fall benötigten, schon verwendet. Wir erinnern uns an die Definition der Vorgängerfunktion auf den natürlichen Zahlen.

```
vorg :: Int -> Int
vorg 0 = 0
vorg n = n - 1
```

Die erste Regel besagt, dass, wenn der Eingabewert die Form 0 hat, der Funktionswert 0 ist. Die 0 auf der linken Seite des Gleichheitszeichens ist also ein Muster.
Das am häufigste anzutreffende Muster finden wir bei der Datenstruktur der Liste. Wir betrachten dazu die Funktionen `kopf` und `rest` (im Model Prelude vordefiniert als `head` und `tail`), die das erste Element einer nichtleeren Liste bzw. den Rest einer nichtleeren Liste zurückliefern.

```
kopf :: [a] -> a
kopf [] = error "empty list"
kopf (x:xs) = x

rest :: [a] -> [a]
rest [] = error "empty list"
rest (x:xs) = xs
```

Unabhängig vom Datentyp `a` der Liste ist hier nur die Struktur relevant. Handelt es sich um eine leere Liste, tritt der erste Fall ein. Handelt es sich um eine nichtleere Liste, dann hat diese die Gestalt `(x:xs)`. Das bedeutet, sie besteht aus einem ersten Element `x`, dem Kopf, und einem - möglicherweise leeren - Rest `xs`. Diese beiden Variablennamen können nun auf der rechten Seite des Gleichheitszeichens verwendet werden. Dies ist, da es sich nur um die Struktur der Liste handelt, unabhängig vom Datentyp möglich. Mustererkennung leistet also einen wichtigen Beitrag zu strukturellem Polymorphismus.

6 Fallunterscheidungen

Eines der zentralen Elemente höherer Programmiersprachen ist die Möglichkeit der Durchführung von Fallunterscheidungen. Durch sie wird ermöglicht, die Ausführung eines Programms abhängig von einer logischen Bedingung (einem Prädikat) an unterschiedlichen Stellen fortzusetzen. Ein ganz einfaches Beispiel ist die Prüfung darauf, ob ein gegebener ganzzahliger Wert eine gerade oder eine ungerade Zahl ist.
Viele höhere Programmiersprachen stellen zu diesem Zweck ein sogenanntes *if-then-else*-Konstrukt bereit. Ist das Prädikat wahr, wird das Programm mit dem Code im *then*-Zweig fortgesetzt, ist es falsch mit dem Code im *else*-Zweig. Die Angabe eines *else*-Zweiges ist in den meisten anderen Programmiersprachen optional.
In Haskell gibt es auch eine solche Struktur, die jedoch eher selten verwendet wird. Stattdessen werden Fallunterscheidungen vor allem mit Mustererkennung (in der Regel hauptsächlich bei rekursiven Funktionen) und sogenannten Guards (dargestellt als vertikale Striche) durchgeführt. Wir verdeutlichen Einsatz und Unterschiede dieser Strukturen durch verschiedene Implementierungen des Beispiels mit der Prüfung, ob eine gegebene ganze Zahl gerade ist.

```
gerade :: Int -> Bool
gerade n = if mod n 2 == 0 then True else False
```

```
gerade2 :: Int -> Bool
gerade2 n | mod n 2 == 0 = True
          | otherwise    = False
```

Die Umsetzung mit Guards ermöglicht die übersichtliche Auflistung alle Fälle. Die einzelnen Guards, d.h. die vertikalen Striche müssen dabei alle die gleiche Einrückungstiefe besitzen, damit sie als zusammengehörige Struktur erkannt werden. Das Schlüsselwort `otherwise` fasst alle Fälle zusammen, die nicht vorher ausdrücklich definiert werden.
Noch größer ist der Vorteil von Guards, wenn es mehr als zwei Fälle gibt. Wir betrachten die Vorzeichenfunktion *sgn*, die für alle positiven Zahlen 1, für alle negativen Zahlen -1 und für null 0 zurückliefert. Die Definition in mathematischer Notation ist:

$$sgn : \mathbb{R} \to \mathbb{R}, sgn(x) = \begin{cases} 1 & : x > 0 \\ 0 & : x = 0 \\ -1 & : x < 0 \end{cases}$$

Eine Implementierung mit klassischem *if-then-else* könnte wie folgt aussehen:

```
sgn :: Float -> Int
sgn x = if x > 0 then 1 else (if x == 0 then 0 else -1)
```

Obwohl nur ein weiterer Fall im Vergleich zum vorherigen Beispiel hinzugekommen ist, ist der Code schon recht lang und unübersichtlich durch die Verwendung der vielen Schlüsselwörter[9]. Unter Verwendung von Guards kann `sgn` wie folgt implementiert werden:

```
sgn x :: Float -> Int
sgn x | x > 0      = 1
      | x == 0     = 0
      | otherwise = -1
```

Man erkennt hier deutlich, dass der Unterschied zur mathematischen Notation sehr gering ist. Die geschweifte Klammer wird durch die Guards ersetzt und im rechten Teil werden Prädikate und Funktionsergebnis zwar vertauscht, aber inhaltlich im Wesentlichen direkt abgeschrieben. Bei rekursiven Funktionen werden Guards häufig in Verbindung mit Mustererkennung eingesetzt.

[9] Natürlich kennen andere Programmiersprachen auch platzsparende Konstrukte etwa mit *else if* oder *case*, aber sie benötigen dennoch ein Vielfaches an Syntax

6.1 Übungen

Aufgabe 1

Implementieren Sie die folgenden Funktionen.

(a) `istzweistellig - liefert True, wenn die Zahl zweistellig ist`
(b) `max2 - bestimmt die größere von zwei Zahlen`
(c) `max3 - bestimmt die größte von drei Zahlen. Finden Sie auch eine Lösung, die max2 verwendet`

Aufgabe 2

Definieren Sie Funktionen mit den folgenden Eigenschaften in HASKELL.

(a) Wenn $n \geq 0$ ist, soll n unverändert zurückgegeben werden, ansonsten $-1 \cdot n$. Beschreiben Sie auch die Funktionalität dieser Funktion.
(b) $f(x) = 3x^2 - x^3$, falls $0 \leq x \leq 4$.
(c) $sgn(x) = \begin{cases} 1 & : x > 0 \\ 0 & : x = 0 \\ -1 & : x < 0 \end{cases}$

7 Rekursion

In Haskell gibt es keine Schleifen zur Steuerung des Kontrollflusses wie in den prozeduralen und objektorientierten Programmiersprachen. Wiederholt ablaufende Prozesse werden mit Funktionen höherer Ordnung und durch Rekursion beschrieben. Rekursion bedeutet hier, dass eine Funktion im Verlauf ihrer Berechung sich selbst aufruft bis eine Abbruchbedingung erreicht ist. Rekursive Berechnungen bestehen grundsätzlich aus zwei Fällen:

- Abbruchfall (Stoppfall, Basisfall): er enthält keine Selbstaufrufe, sondern einen Ausdruck, der direkt berechnet werden kann.
- Rekursionsfall: er enthält mindestens einen Selbstaufruf. Sein Ergebnis kann also nicht unmittelbar berechnet werden, sondern hängt von anderen Berechnungen ab.

Beispiel Die Funktion `laenge :: [a] -> Int` erhält eine Liste beliebigen Typs als Eingabe und gibt die Anzahl der Elemente in der Liste zurück. Die einfachste Liste ist die leere Liste `[]`; sie enthält kein Element. In diesem Fall kann das Ergebnis der Funktion sofort berechnet werden. Also:

```
laenge [] = 0
```

Ist die Liste nicht leer, kann sie beliebig viele Elemente enthalten, die nacheinander gezählt werden müssen. Eine nicht leere Liste können wir mit dem Muster `(x:xs)` beschreiben. Wir zählen das Element `x` und rufen dann die Funktion mit dem Rest `xs` auf. Die Länge einer nicht leeren Liste ist also 1 plus die Länge des Rests der Liste. Wir definieren den Rekursionsfall für unserer Funktion wie folgt:

```
laenge (x:xs) = 1 + laenge xs
```

In der Definition der Funktion `laenge` gibt es also einen Abbruchfall und einen Rekursionsfall. Da der Eingabewert der Funktion beim Selbstaufruf immer aus dem Rest des vorherigen Eingabewertes besteht, wird der Eingabewert von `laenge` bei jedem Selbstaufruf kürzer bis schließlich ein Selbstaufruf mit der leeren Liste erfolgt, der direkt berechnet werden kann. Die Berechnung der Funktion `laenge` terminiert also für alle endlichen Listen.
Wir untersuchen die Arbeitsweise von `laenge` anhand der dreielementigen Liste `[2,3,5]`:

$$
\begin{array}{rrl}
 & laenge[2,3,5] & Rekursionsfall.Kopf 2, Rest[3,5] \\
= & 1 + laenge[3,5] & Rekursionsfall.Kopf 3, Rest[5] \\
= & 1 + (1 + laenge[5]) & Rekursionsfall.Kopf 5, Rest[] \\
= & 1 + (1 + (1 + laenge[])) & Abbruchfall \\
= & 1 + (1 + (1 + 0)) & \\
= & 1 + (1 + 1) & \\
= & 1 + 2 & \\
= & 3 &
\end{array}
$$

Im Beispiel erkennt man, dass die Berechnung aus zwei Phasen besteht:
In der ersten Phase entsteht eine Kette von Selbstaufrufen, im Verlauf derer die Liste immer kürzer wird. Keiner der Selbstaufrufe kann direkt zu einem Ergebnis ausgewertet werden, bis am Ende der ersten Phase der Abbruchfall erreicht wird. Dieser kann direkt durch ein Ergebnis (hier: 0) ersetzt werden und die Kette der Selbstaufrufe stoppt.
In der zweiten Phase wird der durch die Kette von Selbstaufrufen entstandene Ausdruck schrittweise ausgewertet und zusammengefasst bis zum Ergebnis.

7.1 Übungen

Aufgabe 1

Nach der Legende zur Erfindung des Schachspiels verlangt der Erfinder von seinem König als Belohnung ein Reiskorn auf das erste Feld des Schachbrettes. Auf jedes weitere Feld soll jeweils das Doppelte an Reiskörnern des vorherigen Feldes liegen. Schreiben Sie eine Funktion `reis :: Int -> Int`, die für eine Feldnummer die korrekte Anzahl Reiskörner berechnet.

Aufgabe 2

Programmieren Sie die folgenden Funktionen.

(a) Die Fakultätsfunktion `fac` mit der Definition $fac(n) = \prod_{k=1}^{n} k$ für alle $n \geq 1$ und $fac(0) := 1$.
(b) Die FIBONACCI-Zahlen `fib` mit $fib(0) = fib(1) = 1$ und $fib(n) = fib(n-1) + fib(n-2)$ für alle $n \geq 2$.

Aufgabe 3

Gegeben sei die Funktion $s : \mathbb{N} \to \mathbb{N}, n \mapsto \begin{cases} 0 & : n = 0 \\ (s(n-1)+1)^2 & : \text{sonst} \end{cases}$.

(a) Realisieren Sie s in HASKELL und berechnen Sie $s(3)$ zur Kontrolle von Hand.
(b) Vergleichen Sie das Wachstum von s und $fac(x)$ sowie $g(x) = x^x$ und beschreiben Sie Ihre Beobachtungen.
(c) Recherchieren Sie die Bedeutung der Funktionswerte von s in der OEIS[10].

Aufgabe 4

Schreiben Sie HASKELL-Funktionen für die folgenden Probleme.

(a) Die Teiler einer Zahl in einer Liste angeben.
(b) Ein Prädikat, das prüft, ob eine gegebene Zahl eine Primzahl ist.
(c) Eine Liste aller Primzahlen kleiner gleich einer gegebenen Zahl angeben.
(d) Die Summe der Quadrate der Elemente einer Liste berechnen.
(e) Eine Liste aller Zahlen, die kleiner als 1000 und durch 3 oder durch 5 teilbar sind, angeben.
(f) Die Anzahl der Ziffern einer natürlichen Zahl angeben.
(g) Die Quersumme einer Zahl berechnen.

Aufgabe 5

Die Funktion `iterate :: (a -> a) -> a -> [a]` nimmt eine Funktion f und einen Startwert a und liefert die (unendliche) Folge $\langle a, f(a), f(f(a)), f(f(f(a))), \cdots \rangle$ zurück.

(a) Implementieren Sie die Funktion als `myiterate`.
(b) Verwenden Sie `iterate`, um einen Pseudozufallszahlengenerator in Form eines LCG[11] zu programmieren. Entwickeln Sie eine Funktion `lcg`, die den Multiplikator, das Inkrement, den Modul und den Startwert (Seed) als Parameter nimmt und eine (potentiell) unendlich lange Folge von Pseudozufallszahlen erstellt. Testen Sie Ihr Programm auch mit den realen Werten der LCG von `C` und `Java`[12].

Aufgabe 6

Programmieren Sie eine rekursive Funktion `snoc :: [a] -> a -> [a]`, die ein Element hinten in eine Liste einfügt. Finden Sie auch eine Variante, die ohne Rekursion auskommt.

[10] `http://oeis.org`, The Online Encyclopedia of Integer Sequences.
[11] siehe `https://de.wikipedia.org/wiki/Kongruenzgenerator`
[12] siehe Abschnitt *Parameters in common use* unter `https://en.wikipedia.org/wiki/Linear_congruential_generator`

Aufgabe 7

Gegeben ist die folgende Funktion `foo`:

```
foo :: Num a => [a] -> a
foo [] = 0
foo (x:xs) = x + foo xs
```

(a) Werten Sie die Aufrufe `foo []`, `foo [7]` und `foo [1,4,7]` ausführlich aus.
(b) Beschreiben Sie die Funktionalität von `foo` und begründen Sie, dass alle Aufrufe der Funktion mit endlichen Eingabewerten terminieren.
(c) Erläutern Sie, was passiert, wenn `foo` mit der Liste `[1..]` aufgerufen wird.

Aufgabe 8

Die sogenannte Fakultät einer natürlichen Zahl n ist das Produkt der Zahlen von 1 bis einschließlich n. Per Definition ist außerdem festgelegt, dass die Fakultät von 0 den Wert 1 hat.

(a) Entwickeln Sie eine Funktion `fac :: Integer -> Integer`, die die Fakultät des Eingabewertes berechnet.
(b) Werten Sie die Aufrufe `fac 0`, `fac 1`, `fac 3` ausführlich aus.

Aufgabe 9

(a) Definieren Sie eine Funktion `enthaelt :: Eq a => a -> [a] -> Bool`, die die Funktionalität der vordefinierten Funktion `elem` besitzt. Die Funktion liefert `True` zurück, wenn ein Element in einer gegebenen Liste enthalten ist und `False`, wenn das Element nicht enthalten ist.
(b) Werten Sie die Aufrufe `enthaelt 0 [0,2]`, `enthaelt 0 [2,0]`, `enthaelt 0 []` und `enthaelt 1 [0,2]` ausführlich aus.

8 Komposition

In größeren Programmen werden häufig komplexe oder verschachtelte Berechnungen durchgeführt. Zur Darstellung solcher Ausdrücke und zur Kennzeichnung der Auswertungsreihenfolge eines Ausdrucks können runde Klammern verwendet werden.

Beispiel Zweites Element einer Liste: `head (tail xs)` Komplizierte Arithmetik: $2 * (x + 1))^2$ Zur optischen Vereinfachung solcher Ausdrücke kann der $-Operator benutzt werden. Grob ausgedrückt trennt er Funktionsanwendungen voneinander ab und hilft so, Klammern zu sparen. Mit dem $-Operator lässt sich der obige Ausdruck wie folgt vereinfachen:

```
head $ last xs
(2 * $ x + 1)^2 --oder sogar
(^2) $ (2*) $ x + 1
```

Vereinfacht gesagt wird jeder Ausdruck rechts des $-Operators zuerst berechnet und dient dann als Eingabewert für die Funktion links des $-Operators.
Aus der Mathematik kennen wir außerdem den ∘-Operator, der die Hintereinanderausführung (Verkettung bzw. Komposition) von Funktionen beschreibt. Er ist für zwei Funktionen f und g wie folgt definiert:

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

Bereits anhand der Definition erkennt man den Zusammenhang zu den verschachtelten Funktionsaufrufen aus dem Beispiel. Mit dem Verkettungsoperator, der in Haskell als Punkt `.` geschrieben wird, lassen sich die Ausdrücke aus dem Beispiel wie folgt darstellen:

```
(head . head) xs
((^2) . (2*) . (+1)) x
```

Werden die Ausdrücke als Funktionen definiert, kann die Schreibweise als Komposition von Teilfunktionen sogar noch weiter vereinfacht werden.

```
zweites :: [a] -> a
zweites = head . last


fun :: Num a => a -> a
fun = (^2) . (2*) . (1+)
```

Das Argument der Funktion, d.h. der Eingabewert muss nicht explizit aufgeführt werden, wenn dieser in der Funktionskomposition ausschließlich ganz rechts auftritt. Die Gültigkeit dieser Vorgehensweise begründet sich aus der η-Konversion des λ-Kalküls, der den theoretischen Unterbau der funktionalen Programmierung bildet.

8.1 Die Funktion `flip`

Manchmal lässt sich Code dadurch vereinfachen, dass die Reihenfolge der Eingabewerte einer binären Funktion vertauscht werden. Genau dies bewirkt die Funktion `flip`. Es gilt:

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

Wir betrachten beispielhaft erneut die Funktion `gerade`, die prüft, ob ihr ganzzahliger Eingabewert eine gerade Zahl ist oder nicht.

```
gerade :: Integral a => a -> Bool
gerade n = 0 == mod n 2
```

Mithilfe von Funktionskomposition `.` und der Funktion `flip` kann die Funktionsdefinition vereinfacht werden. Es gilt `mod n 2 ≡ flip mod 2 n`.

```
gerade2 :: Integral a => a -> Bool
gerade2 n = ((==0) . (flip mod 2)) n
```

Da durch den Einsatz von `flip` das Argument n für die Funktion `mod` nun nur ganz rechts steht, kann sogar η-Konversion angewendet werden.

```
gerade2 = (==0) . (flip mod 2)
```

Bei einer gänzlich durch Funktionskomposition erfolgten Definition spricht man auch von *punktfreier Darstellung* (engl. point free style)[13].

[13]Im Deutschen scheint diese Bezeichnung etwas seltsam, da die Funktionskomposition ja gerade durch einen Punkt ausgedrückt wird. Im Englischen wird das entsprechende Zeichen jedoch als *dot* bezeichnet.

8.2 Übungen

Aufgabe 1

Stellen Sie die folgenden Funktionen einmal mithilfe des `$`-Operators und einmal als Funktionskomposition dar. Testen Sie Ihre Lösungen mit GHCi und geeigneten Eingabewerten.

a)
```
f xs = head (last xs)
```

b)
```
g x = not (True && x)
```

c)
```
h c = (+) 3 (ord c) -- Modul Data.Char laden
```

d)
```
j x = 4 * (x + 11)
```

e)
```
k xs = not (null xs)
```

f)
```
m xs = (||) False (elem 0 xs)
```

g)
```
n x = p (q (r x))
```

9 Listenfunktionen

head, tail, init, last, (++), (:), concat. Strikte Versionen mit Maybe entferne das erste Vorkommen als Beispiel, leichte Sprache mit markieren, zu lang, anzahl

9.1 Übungen

Aufgabe 1

Schreiben Sie eine Funktion `istvokal`, die überprüft, ob ein gegebener Buchstabe ein Vokal ist. Beachten Sie dabei auch Groß- und Kleinschreibung.

Aufgabe 2

Definieren Sie die folgenden Funktionen.

(a) `laenge - bestimmt die Anzahl der Elemente in einer Liste`
(b) `entfernen - entfernt alle Vorkommen eines Elements in einer Liste`
(c) `istsortiert - liefert True, wenn die Liste aufsteigend sortiert ist`
(d) `satzzeichen - berechnet die Anzahl der Satzzeichen (.;:!?) in einem String`

Aufgabe 3

Programmieren Sie folgende Funktionen in Haskell und geben Sie jeweils auch den Typ der definierten Funktionen an:

(a) Definieren Sie eine Funktion `rev`, welche eine Liste umdreht.
(b) Definieren Sie eine Funktion `inconcat`, die eine Liste von Strings zu einem neuen String konkateniert und dabei zwischen je zwei Elemente eine Zeichenkette einfügt. Beispiel: `inconcat "["Theodor","Litt",$$chule"]` ergibt `"Theodor-Litt-Schule"`
(c) Definieren Sie eine zweistellige Funktion `stelle`, die feststellt, an welcher Position (beginnend bei 0) ein Element in einer Liste vorkommt. Da das Element auch nicht in der Liste vorkommen kann, sollten Sie den Typkonstruktor Maybe verwenden. Beispiel: `stelle 1 [2,1,3,1]` ergibt `Just 1`

Aufgabe 4 [schwer]

Implementieren Sie folgende Funktionen in Haskell:

(a) `inits, tails ::[a] -> [[a]]` zur Berechnung aller Anfangs- bzw. Endstücke einer Liste. (Bsp.: `inits [1,2,3]` liefert `[[],[1],[1,2],[1,2,3]]`)
(b) Prädikate `isPrefixOf, isSuffixOf :: Eq a => [a] -> [a] -> Bool` zum Test ob die erste Liste Präfix bzw. Suffix der zweiten Liste ist. Geben Sie dabei (mindestens) zwei verschiedene Versionen der Funktion `isPrefixOf` an.
(c) `insert :: a -> [a] -> [[a]]` zum Einfügen eines Elements an jede beliebige Stelle einer Liste. (Bsp.: `insert 4 [1,2]` liefert `[[4,1,2],[1,4,2],[1,2,4]]`)
(d) `perm :: [a] -> [[a]]` zur Berechnung aller Permutationen einer Liste.
(e) `split :: (a -> Bool) -> [a] -> ([a],[a])` zum Aufteilen einer Liste in zwei Listen unter Berücksichtigung eines Prädikats.

Aufgabe 5

Die sogenannte COLLATZ-Folge ist eine Folge $\langle r_i \rangle$ von natürlichen Zahlen, die wie folgt erzeugt wird.

- Setze $r_0 := n$ mit beliebigem $n \in \mathbb{N}$.
- Falls r_i gerade ist, so gilt $r_{i+1} = \frac{r_i}{2}$. Falls r_i ungerade ist, so gilt $r_{i+1} = 3 \cdot r_i + 1$.
- Führe diese Berechnung rekursiv durch, bis irgendwann $r_i = 1$ für ein i gilt. Dann beende die Berechnung.

Programmieren Sie eine Funktion `collatz :: Int -> [Int]`, die die COLLATZ-Folge zu einem Startwert zurückliefert. (Bonus: Finden Sie heraus, welcher Startwert unter 100 die längste Folge erzeugt.)

Aufgabe 6

(a) Definieren Sie die Liste `[0,2,4,6]` mit einem `let`-Binding direkt am GHCi-Prompt. Nennen Sie die Liste `x`.
(b) Fügen Sie eine 0 vorne an `x` an.
(c) Konkatenieren Sie die Zahlen 8, 10 und 4 auf zwei verschiedene Arten zur Liste `[8,10,4]`.
(d) Erklären Sie, warum der Ausdruck `1++[]` fehlerhaft ist und geben Sie zwei Möglichkeiten an den Fehler zu beheben, sodass die Liste `[1]` als Ergebnis entsteht.
(e) Erklären Sie, warum der Ausdruck `[1]:[3,5]` fehlerhaft ist und geben Sie zwei Möglichkeiten an den Fehler zu beheben, sodass die Liste `[1,3,5]` als Ergebnis entsteht.

Aufgabe 7

Definieren Sie die Liste `g` mit `let g = [1,3,5,7]` am GHCi-Prompt.

(a) Werten Sie `head g` und `tail g` aus.
(b) Werten Sie `head g : (tail g)` aus und erklären Sie das Ergebnis.
(c) Werten Sie `[head g] ++ tail g` aus und erklären Sie das Ergebnis.
(d) Schreiben Sie eine Funktion `second :: [a] -> a`, die das zweite Element einer Liste liefert. Sie dürfen dabei voraussetzen, dass die Liste mindestens zwei Elemente enthält, d.h. ein Abfangen der Fehlerfälle ist nicht notwendig.
(e) Jemand würde die Funktion `head` gerne so erweitern, dass `head [] = []` gilt, anstatt eine Fehlermeldung zu erzeugen. Erläutern Sie, warum dies die sogenannte Listeninvariante `a= head a : (tail a)` verletzen würde.

10 List Comprehension

LC als Darstellungsform, die mengenähnlich ist. Pythagoreische Tripel, PZ < 100 (n), PZZwillinge < 100(n),

10.1 Übungen

Aufgabe 1

Erzeugen Sie eine Wertetabelle mit Wertepaaren $(x, f(x))$ für eine Funktion f im Intervall $[0; 10]$ mithilfe einer List Comprehension.

Aufgabe 2

Programmieren Sie die folgenden Funktionen mithilfe einer List Comprehension

(a) Die Funktion `map`.
(b) Die Funktion `filter`.
(c) Eine Liste aller Primzahlen kleiner als eine gegebene Zahl.
(d) Eine Liste aller Primzahlzwillinge[14] kleiner als eine gegebene Zahl.
(e) Alle Pythagoreischen Tripel, bei denen alle drei Werte kleiner gleich 10 sind.

[14] siehe `https://de.wikipedia.org/wiki/Primzahlzwilling`

11 Funktionen höherer Ordnung

Eine Funktion f wird als Funktion höherer Ordnung (Funktional) bezeichnet, wenn mindestens einer ihrer Eingabewerte selbst bereits eine Funktion ist. Durch die partielle Applikation ist in Haskell genaugenommen jede Funktion mit mehr als einem Eingabewert bereits eine Funktion höherer Ordnung. Die wichtigsten vordefinierten Funktionen höherer Ordnung in Haskell sind `map`, `filter` und `foldr`.

11.1 Elemente einer Liste aussondern

Wir betrachten die folgenden einfachen rekursiven Funktionen:

```
-- gerade Zahlen herausfiltern
gerade [] = []
gerade (x:xs) | x `mod` 2 == 0 = x : gerade xs
              | otherwise      = gerade xs

-- Primzahlen herausfiltern
primzahlen [] = []
primzahlen (x:xs) | prim x    = x : primzahlen xs
                  | otherwise = primzahlen xs
 where
  prim y = [1,y] = teiler y
  teiler z = [ k | k <- [1..z], mod z k == 0 ]

-- Vokale herausfiltern
vokale [] = []
vokale (x:xs) | x `elem` "AEIOU" = x : vokale xs
              | otherwise        = vokale xs

-- zweistellige Zahlen herausfiltern
zweistellige [] = []
zweistellige (x:xs) | x `elem` [10..99] = x : zweistellige xs
                    | otherwise         = zweistellige xs

-- Einswerte aus einer Key-Value-Liste herausfiltern
einsen [] = []
einsen ((k,v):xs) | v == 1    = k : einsen xs
                  | otherwise = einsen xs
```

In diesen Funktionen ist ein Muster erkennbar.
Der erste Fall (Abbruchfall) ist die Behandlung der leeren Liste. Das Ergebnis ist hier stets die leere Liste `[]`, unabhängig von der Signatur der Funktion.
Im zweiten Fall (Rekursionsfall) wird das erste Element `x` der Liste mit einem Prädikat P vom Typ `a -> Bool` geprüft. Ist das Prädikat wahr, wird das Element der Ergebnisliste hinzugefügt, ansonsten nicht. Dann wird die jeweilige Funktion rekursiv mit dem Rest der Liste `xs` aufgerufen.

Wir abstrahieren dieses Muster zu einer flexiblen Funktion höherer Ordnung, die wir zunächst `aussondern` nennen. Die Funktion erhält ein Prädikat, eine Liste von Eingabewerten beliebigen Typs und gibt eine Liste derjenigen Elemente zurück, die das Prädikat erfüllen.
Entsprechend dem identifizierten Muster definieren wir:

```
aussondern :: (a -> Bool) -> [a] -> [a]
aussondern p [] = []
aussondern p (x:xs) | p x       = x : aussondern xs
                    | otherwise = aussondern xs
```

Durch diese Funktion können die einfachen vorherigen Funktionen kompakter geschrieben werden, ohne dass jedes Mal eine passende rekursive Funktion definiert werden muss. Es gilt:

```
-- gerade Zahlen
```

```
gerade2 xs = aussondern ((==0) . flip mod 2) xs
-- oder mit vordefinierter Funktion even
gerade3 = filter even

-- Primzahlen
primzahlen2 xs = aussondern prim xs

-- Vokale
vokale2 xs = aussondern (flip elem "AEIOU") xs

-- zweistellige Zahlen
zweistellige2 xs = aussondern (flip elem [10..99]) xs

-- Einswerte aus Key-Value-Liste
einsen2 xs = aussondern ((==1) . fst) xs
```

Da es sich hierbei um ein sehr häufiges Muster handelt, gibt es auch eine im Modul Prelude vordefinierte Haskell-Funktion für diesen Zweck. Sie heißt `filter`.

11.2 Alle Elemente einer Liste manipulieren

Wir betrachten die folgenden einfachen rekursiven Funktionen:

```
-- verdoppeln der Elemente
verdoppeln [] = []
verdoppeln (x:xs) = (2 * x) : verdoppeln xs

-- inkrementieren der Elemente
inkrement [] = []
inkrement (x:xs) = (x + 1) : inkrement xs

-- logisches Negieren der Elemente
negieren [] = []
negieren (x:xs) = not x : negieren xs

-- Initialen einer Wortliste sammeln
initial [] = []
initial (x:xs) = head x : initial xs
```

In diesen Funktionen ist ein Muster erkennbar.
Der erste (Abbruchfall) behandelt den Umgang mit der leeren Liste als Eingabewert. In diesem Fall ist nichts zu tun und es wird einfach die leere Liste zurückgeliefert.
Im zweiten Fall (Rekursionsfall) wird eine Funktion auf das erste Element `x` der Liste angewendet und das Ergebnis der Ergebnisliste hinzugefügt. Dann wird die Funktion rekursiv mit dem Rest der Liste `xs` aufgerufen.

Wir abstrahieren dieses Muster zu einer flexiblen Funktion höherer Ordnung, die wir zunächst `anwenden` nennen. Die Funktion erhält die anzuwendende Funktion sowie eine Liste von Eingabewerten beliebigen Typs und gibt eine Liste der durch die Funktion veränderten Werte zurück.
Entsprechend dem identifizierten Muster definieren wir:

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

Durch die Funktion `anwenden` können wir vorherigen Funktionen kompakter geschrieben werden.

```
-- verdoppeln
verdoppeln2 xs = anwenden (2*) xs

-- inkrementieren
```

```
inkrement2 xs = anwenden (+1) xs

-- negieren
negieren2 xs = anwenden not xs

-- Initiale sammeln
initial2 xs = anwenden head xs
```

Da es sich hierbei um ein sehr häufiges Muster handelt, gibt es auch eine im Modul Prelude vordefinierte Haskell-Funktion für diesen Zweck. Sie heißt `map`.

11.3 Anwendungsaufgabe

Es soll im Rahmen eines Forschungsprojektes untersucht werden, ob Datenreihen von Klimamodellen Anomalien in Form von zu stark abweichenden Werten (sog. *Ausreisser*) enthalten. Die Rohdaten liegen als numerische Listen vor.
Zwei der Datenlisten `xs, ys` sind nachfolgend angegeben.

```
xs = [1,2,3,2,5,2,1,3,4,5]
ys = [1,3,2,4,0,0,1,1,2,3]
```

a. Geben Sie den allgemeinen Datentyp der beiden Listen in Haskell-Notation an.

Als kritisch gelten hier Werte, die größer als 4 sind. Ein Praktikant hat zum Herausfiltern der Werte die folgende Funktion `groesserVier :: [Int] -> Bool` geschrieben, die aber leider noch fehlerhaft ist.

```
groesserVier :: [Int] -> Bool
groesserVier n | n >= 4     = True
               | otherwise = False
```

b. Begründen Sie, dass die Funktion `groesserVier` ein Prädikat ist.
Korrigieren Sie alle Fehler in der Implementierung des Praktikanten.
[Bonus] Verallgemeinern Sie die Signatur der Funktion so, dass diese für alle numerischen Eingabewerte funktioniert.

Zur Prüfung der Daten msus auch auf Ausreisser nach unten getestet werden. Hier gilt jeder Wert als kritisch, der kleiner als 1 ist.

c. Implementieren Sie eine rekursive Funktion `hatKleineWerte`, die für eine gegebene Liste prüft, ob diese Ausreisser nach unten enthält.

Um mit beliebigen Prädikaten arbeiten zu können, sollen zur Abstraktion Funktionen höherer Ordnung verwendet werden. Man entscheidet sich für die Funktion `filter`. Zusätzlich kann die Funktion `nichtleer` mit

```
nichtleer :: [a] -> Bool
nichtleer [] = False
nichtleer _ = True
```

aus einem früheren Projekt verwendet werden.

d. Entwickeln Sie eine Funktion `enthaeltAnomalie`, die für ein Prädikat p und eine Liste xs genau dann $True$ zurückliefert, wenn xs Anomalien gemäß p enthält und $False$ ansonsten.
Geben Sie den Typ von `enthaeltAnomalie` an.
Zeigen Sie, dass `nichtleer` ≡ `not . null` für alle Listen `xs :: [a]` gilt.

Ein Programmierer in der Forschungsgruppe hält den gewählten Ansatz für zu kompliziert und schlecht wartbar. Er schlägt ein Refactoring mithilfe des Listenprädikats `any` vor und behauptet, dass für alle passenden Listen sogar `not . null . filter p` ≡ `any p` gilt.

e. Prüfen Sie die Behauptung des Programmierers zunächst für die Listen `xs` und `ys` und danach allgemein.

Die Datenreihen `xs` und `ys` sollen zur Analyse graphisch dargestellt werden. Für die passende Skalierung müssen alle Werte dabei verdreifacht werden.

f. Erstellen Sie eine Funktion `skalieren :: [Int] -> [Int]` für diesen Zweck.

Sie werden beauftragt, die Teilarbeiten der Anomalieprüfung und Skalierung zusammenzuführen. Hierfür sollen Sie eine Funktion `pruefenUndSkalieren` entwickeln, die aus den Datenreihen ps und qs die zu hohen bzw. zu niedrigen Werte entfernt und die übrigen Werte wie beschrieben skaliert.

g. Geben Sie die Definition der Funktion `pruefenUndSkalieren` an.
[Bonus] Geben Sie die Definition dieser Funktion im punktfreien Stil (*point free style*) an.

Eine andere Prädikatsfunktion höherer Ordnung ist `all`[15] mit

```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p [] = True
all p (x:xs) = p x && all p xs
```

h. Definieren Sie `all` mithilfe von `filter`.
[Bonus] Definieren Sie `all` mithilfe von `not, any` und `map`.

[15] Tatsächlich ist `all` nicht nur für Listen, sondern für alle Datentypen, die Mitglied der Typklasse `Foldable` sind, definiert

11.4 Übungen

Aufgabe 1

Erzeugen Sie eine Wertetabelle mit Wertepaaren $(x, f(x))$ für eine Funktion f im Intervall $[0; 10]$.

(a) Mithilfe von `zip` und `map`.
(b) Mithilfe von `map` und einer anonymen λ-Funktion.

Aufgabe 2

Implementieren Sie eine Funktion `myfilter` mithilfe einer rekursiven Definition, die das gleiche Verhalten wie die Prelude-Funktion `filter` hat.

Aufgabe 3

Programmieren Sie Funktionen mit den folgenden Eigenschaften unter Verwendung von `map`. Manche Teilaufgaben benötigen anonyme λ-Funktionen.

(a) Alle Elemente einer Zahlenliste sollen mit 3 multipliziert werden.
(b) Alle Elemente einer Zahlenliste sollen quadriert werden.
(c) Alle Elemente einer beliebigen Liste werden auf 1 gesetzt.
(d) Alle Elemente eines Strings sollen in ihren ASCII-Wert umgewandelt werden. Verwenden Sie das Modul `Data.Char` durch `import Data.Char` in der ersten Zeile Ihrer Datei bzw. durch `:m Data.Char` im GHCi.
(e) Alle Elemente einer Liste Boolescher Werte sollen auf $True$ gesetzt werden.
(f) Alle zweiten Einträge einer Liste von Tupeln (a, Int), d.h. (nur) die Integer-Werte, sollen mit 2 multipliziert werden.
(g) In einer Liste von Tupeln vom Typ $(Int, Bool)$ sollen die Integer auf den Rest bei der Division durch 3 gesetzt werden. Alle Booleschen Werte sollen *toggeln*, d.h. ihren Zustand (Wert) wechseln.
(h) Alle Kleinbuchstaben in einem Buchstaben-String sollen in Großbuchstaben verwandelt werden. Alle anderen Zeichen sollen unverändert bleiben. (Entspricht `Data.Char.toUpper`.)

Aufgabe 4 (schwer)

(a) Die vordefinierte Funktion `replicate :: Int -> a -> [a]` erzeugt eine Liste von Kopien eines Eingabewertes.
Beispiel: `replicate 3 5` liefert `[5,5,5]`. Implementieren Sie `replicate` einmal als rekursive Funktion und einmal mithilfe von `map`.
(b) Verwenden Sie die vordefinierte Funktion `any :: (a -> Bool) -> [a] -> Bool)` und `map` um die Funktion `elem` zu implementieren.

Aufgabe 5

Programmieren Sie Filter-Funktionen mit den folgenden Eigenschaften unter Verwendung von `filter`. Manche Teilaufgaben benötigen anonyme λ-Funktionen.

(a) Filtere alle geraden Zahlen einer Zahlenliste unter Verwendung der vordefinierten Funktion `even`.
(b) Filtere alle Zahlen einer Zahlenliste, die kleiner als 100 sind.
(c) Filtere alle Zeichen einer Zeichenkette, deren ASCII-Wert größer als 70 ist.
(d) Filtere alle Zahlen einer Zahlenliste, die bei der Division durch 3 den Rest 1 lassen.
(e) Filtere alle Nullen einer Zahlenliste heraus.
(f) Filtere alle Tupel der Form $(Int, Bool)$, bei denen der zweite Eintrag den Wert $True$ hat.

Aufgabe 6 (schwer)

Verwenden Sie die vordefinierten Funktionen `not` und `null` sowie `filter` um die Funktion `elem` zu implementieren.

Aufgabe 7 (schwer)

Unter der sogenannten *Leichten Sprache* versteht man Texte in deutscher Sprache, die (unter anderem) nur ein einfaches Vokabular aufweisen und aus nicht mehr als acht Wörtern pro Satz bestehen. Programmieren Sie eine Funktion `markiere :: [String] -> [String]`, die für alle Sätze in einem Text (einer Liste von Sätzen) überprüft, ob diese hinreichend kurz sind. Ist dies nicht der Fall, soll der Satz die Markierung `» "` vor dem eigentlichen Text erhalten. Zerlegen Sie das Problem, indem Sie zunächst die Funktion `anzahl :: Char -> String -> Int`, die die Anzahl von Vorkommen eines bestimmten Zeichens in einem Satz liefert, und das Prädikat `zulang :: String -> Bool`, das `True` liefert, wenn ein Satz zu lang ist, definieren.

Aufgabe 8

Programmieren Sie ein Prädikat `evenL :: [Integer] -> Bool`, das genau dann `True` liefert, wenn die übergebene Liste mindestens eine gerade Zahl enthält. Finden Sie sowohl eine rekursive Implementierung als auch mindestens zwei Varianten, die Funktionen höherer Ordnung wie z.B. `map, elem` oder `filter` verwenden. Ebenfalls können Sie eine List Comprehension (siehe vorheriges Kapitel) oder Faltungen (siehe nächstes Kapitel) verwenden. (Bonuspunkt für die kürzeste Lösung.)

12 Faltungen

foldr, foldl. Typen von foldl und foldr geben. Beispiele $foldsumxs = foldl(+)0xs, foldandxs = foldl($
$accx- > acc\&\&x)Truexs$

12.1 Übungen

Aufgabe 1

Implementieren Sie die folgenden Funktionen von HASKELL mithilfe der Faltungsfunktion `foldl`.

(a) Die vordefinierten Funktionen `product, any, all, elem, reverse, length, concat, unzip, replicate, nub, null, map` und `filter`. (Die letzten drei sind etwas schwieriger.)
(b) Die Logikfunktionen $\vee$ (Oder) und $\oplus$ (Entweder-Oder) so, dass sie mit beliebig langen, d.h. auch leeren Listen von Wahrheitswerten als Eingaben funktionieren.
(c) Zwei Funktionen, die den ggT bzw. das Minimum von Werten einer nicht leeren Liste liefern.

Aufgabe 2

Programmieren Sie folgende Funktionen in Haskell und geben Sie jeweils auch den Typ der definierten Funktionen an:

(a) `ggT` zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen. Am einfachsten ist eine Implementierung des Euklidischen Algorithmus. Beispiel: $ggT(66, 24) = 6$ mit der Herleitung:

$$\begin{aligned} 66 &= 2 \cdot 24 + 18 \\ 24 &= 1 \cdot 18 + 6 \\ 18 &= 3 \cdot 6 + 0 \end{aligned}$$

(b) `kgV` zur Berechnung des kleinsten gemeinsamen Vielfachen zweier Zahlen. Nutzen Sie folgenden Zusammenhang zwischen ggT und kgV für Ihre Implementierung:

$$ggT(a, b) \cdot kgV(a, b) = a \cdot b$$

(c) Definieren Sie explizit rekursive Funktionen `ggTL` und `kgVL`, welche die ggT/kgV-Berechnung auf Listen von Zahlen (also beliebig viele Zahlen) erweitern. Geben Sie auch jeweils implizit rekursive Versionen der Funktionen an, indem Sie eine Faltungsfunktion verwenden. Nutzen Sie hierbei aus, dass sich die Berechnung für drei Zahlen wie folgt auf die Berechnung für zwei Zahlen zurückführen lässt.

$$\begin{aligned} ggT(x, y, z) &= ggT(x, ggT(y, z)) \\ kgV(x, y, z) &= kgV(x, kgV(y, z)) \end{aligned}$$

Aufgabe 3

Programmieren Sie eine Funktion `split :: (a -> Bool) -> [a] -> ([a],[a])` zum Aufteilen einer Liste in zwei Listen unter Berücksichtigung eines Prädikats unter Verwendung von `foldr`.

Aufgabe 4

Gegeben seien die folgenden HASKELL-Funktionen. Finden Sie heraus, welche Funktionen sie jeweils darstellen.

(a) `f n = foldr (*) 1 [1..n]`
(b) `g xs = foldl (flip (:)) [] xs`
(c) `h xs = foldr (:) [] xs`

Aufgabe 5

Ein Prozess benötigt eine Verarbeitungszeit von 30 ZE auf einer CPU. Durch Aufteilen in mehrere parallele Threads kann diese Zeit gleichmäßig unter den Threads aufgeteilt werden. Allerdings entsteht beim Zusammenfügen der Teilergebnisse ein Overhead von 2 ZE pro Thread ab den zweiten Thread. (Bei einem Thread gibt es natürlich keinen Overhead). Bei zwei Threads beträgt die Ausführungszeit also 17 ZE, nämlich 15 ZE pro Thread und 2 ZE Overhead.

(a) Stellen Sie die benötigte Ausführungszeit als Funktion in Abhängigkeit von der Anzahl der Threads dar.
(b) Schreiben Sie eine Funktion `minThreadTime` in HASKELL, die eine Funktion und eine ganzzahlige Obergrenze als Parameter nimmt und die minimale Ausführungszeit berechnet.
Hinweis: Um auf einen Startwert zu verzichten, verwenden Sie `foldr1` statt `foldr`.

13 Eigene Datentypen

Typsynonyme mit dem Schlüsselwort type Fenster mit Zustand (und Größe) als Beispiel, Binärbäume als Beispiel mit Verweis auf Suchen und Sortieren Arithm. Ausdrücke

13.1 Übungen

Aufgabe 1

Definieren Sie einen Datentyp für komplexe Zahlen z der Form $z = a + ib$, wobei $Re(z) = a$ der Realteil und $Im(z) = ib$ der Imaginärteil ist. Real- und Imaginärteil sollen durch Werte des Datentyps `Float` realisiert werden. Implementieren Sie danach die komplexe Addition und Multiplikation auf Ihrem Datentyp und schreiben Sie eine Funktion, die zu einer gegebenen komplexen Zahl deren *konjugierte* Zahl zurückliefert.

Aufgabe 2

Definieren Sie einen eigenen Datentyp für Listen als Alternativ zum vordefinierten Datentyp. Eine Liste ist entweder leer oder besteht aus einem Listenkopf (ein Element) und einer Restliste (beliebig viele Elemente, kann auch leer sein). Implementieren Sie danach die Funktionen `car, cdr` und `len`, die die gleiche Funktionalität wie die vordefinierten HASKELL-Funktionen `head, tail` und `length` abbilden sollen.

Aufgabe 3

Betrachten Sie die Modellierung der natürlichen Zahlen durch den Datentyp `Nat` mit der Definition `data Nat = Zero | Succ Nat deriving (Show, Eq)`.

(a) Entwickeln Sie Funktionen `toInt :: Nat -> Int` bzw. `toNat :: Int -> Nat`, die ein Objekt vom Typ `Nat` in einen Integer verwandeln und umgekehrt.
(b) Programmieren Sie eine Funktion für die Addition zweier Objekte vom Typ `Nat`.
(c) Programmieren Sie die Multiplikation für Objekte vom Typ `Nat`. Dafür könnte eine Funktion `dec :: Nat -> Nat` nützlich sein, die den Vorgänger von einem `Nat` liefert. Es genügt, wenn diese mit allen Objekten außer `Zero` funktioniert. Beim Aufruf mit `Zero` soll eine Fehlermeldung erscheinen.
(d) Erweitern Sie die Vorgängerfunktion mithilfe der Maybe-Monade so, dass beim Aufruf mit `Zero` der Wert `Nothing` zurückgegeben wird.
(e) Definieren Sie Ihre eigene Instanz für `Show` zur schöneren bzw. aussagekräftigeren Darstellung von Objekten vom Typ `Nat`. Verwenden Sie hierfür z.B. die Funktion `toInt` zur Darstellung.

Aufgabe 4 [schwer]

Implementieren Sie Mengen als funktionale Datenstruktur. Implementieren Sie folgende Mengenoperationen:

```
emptySet, addToSet, removeFromSet, isElem, union, intersect, without, invert, listToSet
```

Aufgabe 5 [schwer]

Erstellen Sie einen Datentyp `Expr` für arithmetische Ausdrücke mit den vier Grundrechenarten für Zahlenwerte vom Typ `Float`. Implementieren Sie eine Funktion `eval :: Expr -> Float`, die einen beliebig komplexen, wohlgeformten arithmetischen Ausdruck auswertet, d.h. seinen Wert berechnet.
Hinweis: Beachten Sie den Sonderfall der Division durch 0.

14 Suchprobleme

Suchen in Listen und Binärbäumen. Suchen von Elemente (Bool), Wert an einem Index, Index eines Wertes. Im Baum Tiefen- und Breitensuche. Preorder-, Inorder- und Postorder-Traversal

14.1 Übungen

Aufgabe 1

Programmieren Sie die folgenden Funktionen in Haskell.

(a) Das Prädikat `elem`, das überprüft, ob ein Element in einer Liste enthalten ist, in beliebiger Form.
(b) Die Funktion `wertanstelle :: [a] -> Int -> a`, die den Wert an einem gegebenen Index zurückliefert. Für Indizes, die zu keinem Listenelement gehören, soll eine Fehlermeldung ausgegeben werden. (Bonus: Typsichere Version mit der Maybe-Monade.)
(c) Die Funktion `stellevon :: a -> [a] -> Int`, die den Index eines gegebenen Wertes zurückliefert. Für Werte, die nicht in der Liste vorkommen, soll -1 ausgegeben werden. (Bonus: Typsichere Version mit der Maybe-Monade.)
(d) Die Funktion `lookupval :: a -> [(a,b)] -> b`, die aus einer Liste von Schlüssel-Wert-Paaren (*key value pairs*) zu einem gegebenen Schlüssel den zugehörigen Wert zurückliefert.

Aufgabe 2

Gegeben sei die folgende Datenstruktur:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving Show
```

(a) Das Prädikat `telem`, das überprüft, ob ein Element in einem Baum enthalten ist.
(b) Den Pfad zu einem gegebenen Element des Baumes. (Bonus: Typsichere Version mit der Maybe-Monade.)
(c) Implementieren Sie drei Tiefensuchen, die die Elemente der Knoten in Preorder-, Inorder- bzw. Postorder-Traversal ausgeben.
(d) Implementieren Sie eine Breitensuche, die die Elemente der Knoten in einer Liste ausgibt.

Aufgabe 3

Realisieren Sie alle Funktionen aus der vorherigen Aufgabe mit einer Datenstruktur für Binärbäume, die einen dritten Konstruktor der Form `Leaf a` für Blätter aufweist.

15 Sortierprobleme

Insertion Sort, Mediansort ggf. Mergesort. Vergleich zur imperativen Programmierung

15.1 Übungen

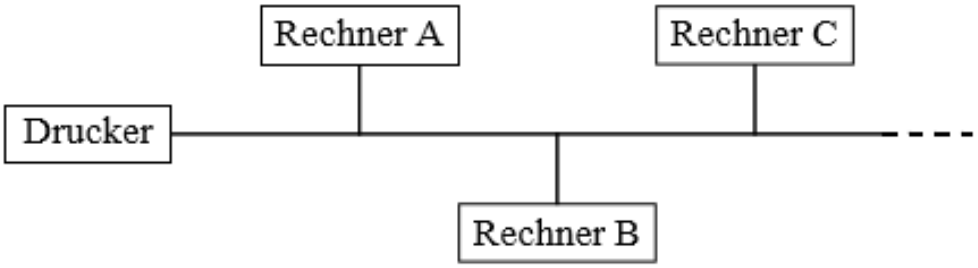
Aufgabe 1

Programmieren Sie die folgenden Sortieralgorithmen.

(a) Bubblesort
(b) Quicksort
(c) Insertion Sort unter Zuhilfenahme einer Faltungsfunktion

16 Druckerwarteschlange

Ein Drucker kann von mehreren Rechnern mit Druckaufträgen versorgt werden[16].



Hierbei kann es vorkommen, dass neue Aufträge erteilt werden, während der Drucker noch einen Auftrag bearbeitet. Man benötigt eine Art Zwischenspeicher, in dem die neu hinzukommenden Aufträge abgelegt werden, bis sie nach dem FIFO-Prinzip vom Drucker bearbeitet werden können. Diesen Zwischenspeicher sollen Sie im Folgenden konzipieren. Wir verwenden daher eine eigene Typklasse `Schlange`, d.h. `type Schlange = [String]`.

Aufgabe 1

Implementieren Sie die folgenden Grundfunktionen für `Schlange` und geben Sie deren Signaturen an.

(a) `leereSchlange - erzeugt eine leere Warteschlange`
(b) `entfernen - das erste Objekt der Warteschlange wird entfernt`
(c) `aufnehmen - ein neues Objekt wird hinten an die Warteschlange angefügt`

Aufgabe 2

Neben den Grundfunktionen soll ermittelt werden wie lang die aktuelle Warteschlange ist. Außerdem wird eine Funktion benötigt, die angibt, an welcher Stelle sich das erste Element von Rechner X befindet. Implementieren Sie die Funktionen `laenge` und `stelle` und geben Sie deren Signaturen an.

Aufgabe 3

Druckaufträge sollen vom Benutzer, d.h. synonym dem Rechner, abgebrochen werden können. Implementieren Sie die Funktionen `loescheErsten` und `loescheAlle`, die den ersten bzw. alle Aufträge eines Benutzers abbrechen. Dazu ist es auch nützlich zu wissen, wer überhaupt einen Druckauftrag angemeldet hat. Entwickeln Sie eine Funktion `benutzer`, die alle Benutzer sammelt, die derzeit einen Auftrag in der Warteschlange haben.

Aufgabe 4

Die Warteschlange soll um Prioritäten erweitert werden zu einem Typ `PSchlange`, der aus Paaren je einem Benutzer und einer Priorität (als `Int`) besteht.

(a) Legen Sie ein passendes Typsynonym für `PSchlange` an.
(b) Implementieren Sie die Grundfunktionen aus Aufgabe 1 auch für `PSchlange`.
(c) Entwickeln Sie eine Funktion, die die Warteschlange nach Priorität aufsteigend sortiert.
(d) Beschreiben Sie, welches Problem bei der Abarbeitung nach Priorität entsteht und machen Sie einen Lösungsvoschlag.

[16] Aufgabenidee entnommen aus *Materialien zum Lehrplan Informatik, Landesmedienzentrum Rheinland-Pfalz, Mai 1999*

17 Wegweiser für Fortgeschrittene

Konzept und Implementierung von Aussagenlogischen Formeln mit Baumstrukturen Erläuterung und Aufbau eines 14-15-Puzzles von rosetta.org Datenbankanbindung mit HDBC.SQLite3 Graphische Oberflächen mit GTK2HS (Benutzeroberflächen) Spieleprogrammierung mit Gloss Deterministische Automaten, einfache Spiele Literatur: Real World Haskell, Learn you a Haskell for great good