WPU 9

Funktionale Programmierung mit Haskell

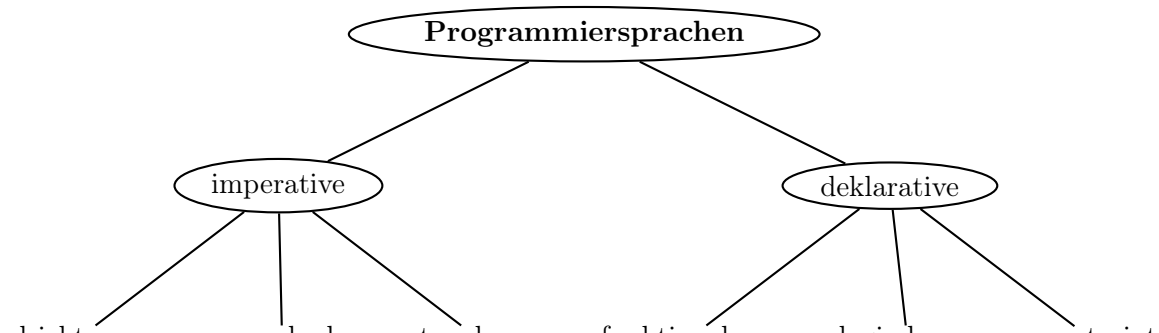
P. Kreißig

Stand 11. November 2018

Inhaltsverzeichnis

1. Funktionale Programmierung

1.1. Einführung



- Imperative Programme beschreiben, wie ein Problem gelöst wird.
- Deklarative Programme beschreiben das Problem, die Lösung übernehmen Compiler und Laufzeitumgebung.
- Bei der funktionalen Programmierung stehen Funktionen (im mathematischen Sinne) im Vordergrund. Diese können als Daten behandelt werden, was das „Rechnen mit Funktionen“ erlaubt.

Wir werden uns mit der funktionalen Programmiersprache HASKELL beschäftigen.

1.2. Werkzeuge

Befehle und Programme müssen für den Computer lesbar übersetzt werden. Dann können sie ausgeführt werden. Das erledigt der *Compilerinterpreter*.

Compiler - maschinen-lesba

Wir nutzen den **G**lasgow**H**askell**C**ompiler (GHC oder GHCi).

Interpreter - fü zur Laufzeit aus

Dieser kann schon einzelne Befehle ausführen.
Programme müssen zuerst in einen *Editor* eingegeben werden. Dazu eignen sich z.B. *Geany* oder *Notepad*.

Sowohl GHCi als auch *Geany* sind frei verfügbar.

1.2.1. Arbeiten mit dem GHCi

Nach dem Starten des GHCi zeigt dieser seine Bereitschaft mit dem Prompt

```
Prelude>
```

Der Name vor dem `>` bezieht sich auf das geladene Modul, wobei `Prelude` eine interne Sammlung gebräuchlicher Funktionen ist.

Mit einem Doppelpunkt : leitet man Compilerkommandos ein, z.B.

:l <Datei>.hs	Laden des Haskell-Programmes Datei.hs (auch :load)
:q	Verlassen der Umgebung (auch :quit)
:r	reload des Programmes (auch :reload)
:?	ghci-Kommandos anzeigen (auch alle dieser Tabelle)
:browse	alle definierten Bezeichner incl. Funktionen des aktuellen Moduls anzeigen `:browse prelude` zeigt alle Kommandos der Standardumgebung
:! <Kommando>	führt ein Shell-Kommando aus
:show paths	zeigt das aktuelle Arbeitsverzeichnis an, hier muss die `.hs`-Datei liegen.
:cd /pfad	wechselt das aktuelle Arbeitsverzeichnis auf z.B. `:cd /home/schueler/haskell/` oder `:cd /media/schueler/meinstickname/haskell/`

Nach dieser Eingabe ergibt

```
Prelude> 2
2
it :: Num a => a
```

2 ist ein Ausdruck, der zu 2 ausgewertet wurde und von der Typklasse `Num` ist.

GHCi lässt sich als Taschenrechner einsetzen

```
Prelude> 42-6*9
-12
it :: Int
Prelude> sin pi
1.2246063538223773e-16
it :: Double
```

Folgendes schafft kein normaler Taschenrechner:

```
Prelude> 42^100
211314374101136073653004404552311399169887833071358006126447793439
156491987549777768821505773215181117202931524793215899487966855318
614582471095039468412671203737 6
it :: Integer
```

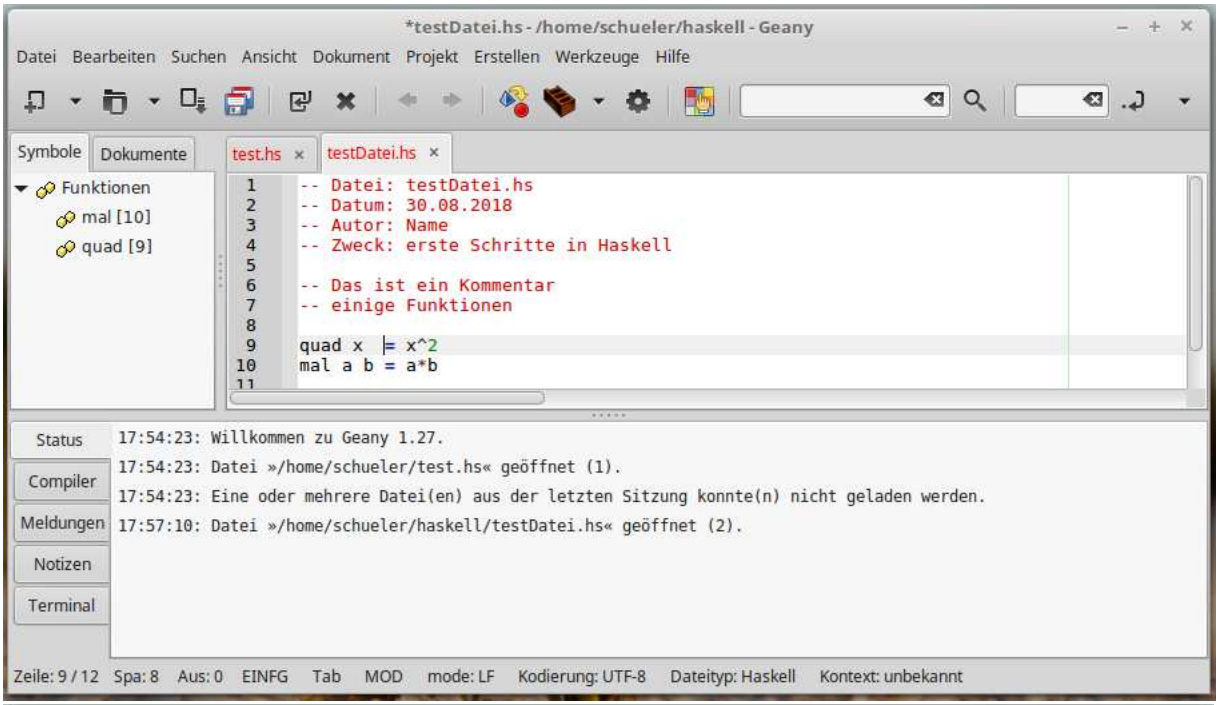
Der `Integer`-Datentyp ist nicht größenbegrenzt.

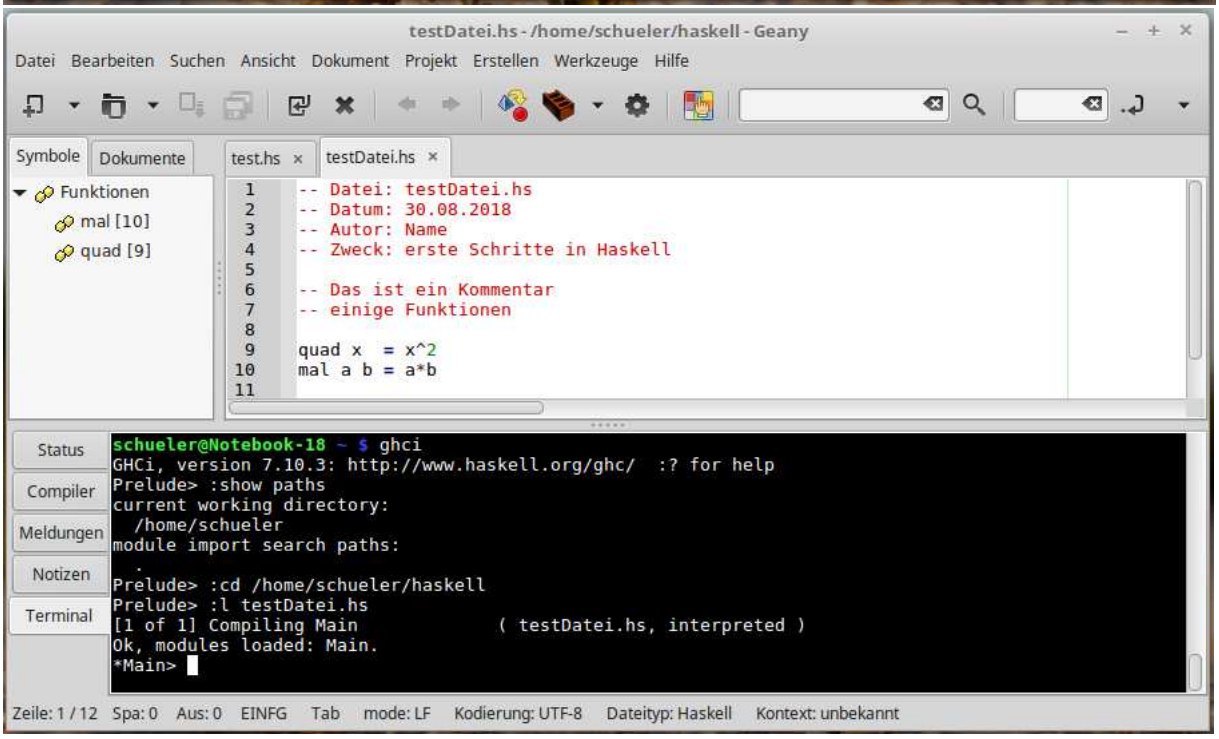
1.2.2. Arbeiten mit Geany

Hier kann man später größere Programme editieren, die dann der GHCi auswertet .

Bei Einstellungen kann man Haskell als Sprache einstellen, damit das Syntaxhighlighting funktioniert.

Damit eine Datei im GHCi geladen werden kann, muss sie im Editor unter `dateiname.hs` abgespeichert werden.


*testDatei.hs - /home/schueler/haskell - Geany
Datei Bearbeiten Suchen Ansicht Dokument Projekt Erstellen Werkzeuge Hilfe
Symbole Dokumente
Funktionen
mal [10]
quad [9]
test.hs
testDatei.hs
1 -- Datei: testDatei.hs
2 -- Datum: 30.08.2018
3 -- Autor: Name
4 -- Zweck: erste Schritte in Haskell
5
6 -- Das ist ein Kommentar
7 -- einige Funktionen
8
9 quad x = x^2
10 mal a b = a*b
11
Status
Compiler
Meldungen
Notizen
Terminal
17:54:23: Willkommen zu Geany 1.27.
17:54:23: Datei »/home/schueler/test.hs« geöffnet (1).
17:54:23: Eine oder mehrere Datei(en) aus der letzten Sitzung konnte(n) nicht geladen werden.
17:57:10: Datei »/home/schueler/haskell/testDatei.hs« geöffnet (2).
Zeile: 9 / 12 Spa: 8 Aus: 0 EINFG Tab MOD mode: LF Kodierung: UTF-8 Dateityp: Haskell Kontext: unbekannt


testDatei.hs - /home/schueler/haskell - Geany
Datei Bearbeiten Suchen Ansicht Dokument Projekt Erstellen Werkzeuge Hilfe
Symbole Dokumente
Funktionen
mal [10]
quad [9]
test.hs
testDatei.hs
1 -- Datei: testDatei.hs
2 -- Datum: 30.08.2018
3 -- Autor: Name
4 -- Zweck: erste Schritte in Haskell
5
6 -- Das ist ein Kommentar
7 -- einige Funktionen
8
9 quad x = x^2
10 mal a b = a*b
11
Status
Compiler
Meldungen
Notizen
Terminal
schueler@Notebook-18 ~ $ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
Prelude> :show paths
current working directory:
/home/schueler
module import search paths:
.
Prelude> :cd /home/schueler/haskell
Prelude> :l testDatei.hs
[1 of 1] Compiling Main (testDatei.hs, interpreted)
Ok, modules loaded: Main.
*Main>
Zeile: 1 / 12 Spa: 0 Aus: 0 EINFG Tab mode: LF Kodierung: UTF-8 Dateityp: Haskell Kontext: unbekannt

2. Grundlegende Programmierung

2.1. Datentypen

Datentypen sind Behälter für Werte eines bestimmten Typs. Es gibt die Grundtypen für Zahlen (Num), Zeichen (Char) und Wahrheitswerte (Bool). Es gibt natürlich auch den Datentyp String, der aber eigentlich eine Liste von Char ist.

Typklasse	Typ	Beschreibung	Beispielwerte
Num	Int	ganze Zahlen (Integer) ca. vorzeichenbehaftete 29-Bit-Zahl	-3,-2,-1,0,1,2,3
	Integer	quasi unbeschränkte Größe möglich	
	Double	Fließkommazahlen	1.0 ; 3.141
	Float		
Char	Char	Zeichen	'a' , 'A' , '3'
	String	Liste von Zeichen	"Hello world"
Bool	Bool	Wahrheitswert	True oder False

Mit der Voreinstellung `:set +t` kann man sich im GHCi den Typ jeweils darstellen lassen. Mit `:unset +t` lässt sich diese Einstellung wieder deaktivieren.

Beispiele:

```
Prelude> :set +t
Prelude> 3+5
8
it :: Integer
Prelude> 3.4 - 6.8
-3.4
it :: Double
Prelude> 'h'
'h'
it :: Char
Prelude> "Hallo"
"Hallo"
it :: [Char]
Prelude> 5 > 7
False
it :: Bool
```

2.2. Funktionen

Funktionen bilden Eingabewerte eindeutig auf Ausgabewerte ab, d.h., jeder Belegung der Eingabewerte wird genau ein Ausgabewert zugeordnet.

Funktionen mit zwei Parametern können sowohl *Infix* als auch *Präfix* notiert werden.

Standard ist Infix	Schreibweise als Präfix
2 + 3	(+) 2 3

Standard ist Präfix	Schreibweise als Infix
ggT 12 18	12 ‘ggT‘ 18

Das Hochkomma ist das rechts neben dem Fragezeichen.

Mit den in HASKELL bereits eingebauten Funktionen kann man direkt am GHCi -Prompt arbeiten und sie wie oben infix oder präfix notieren und den GHCi als Taschenrechner nutzen.
Alle so nutzbaren Funktionen findet man im Anhang A (Eingebaute Funktionen). Man kann sie auch im GHCi mittels `:browse` abrufen. An den Typsignaturen erkennt man, welche und wie viele Parameter die Funktion erwartet. Für eine einzelne Funktion kann man die Typsignatur mit `:t` aufrufen. Beispiel:

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

`(>=)` ist jetzt Präfix notiert. Der Operator `::` liefert die Typsignatur. Für alle Typen a, auf denen eine Ordnungsrelation besteht, wird eine Eingabe von zwei Parametern vom Typ a erwartet und ein Wahrheitswert bool ausgegeben.

```
Prelude> (>=) 3 4
False
it :: Bool
Prelude> 3 >= 4
False
it :: Bool
```

2.2.1. Sektionen

Man kann eine Funktion auch zerlegen, erhält dann eine Sektion. Man erkennt an der Typsignatur, dass `(>=5)` nun einen Wert des Typs `Num`, also eine Zahl erwartet. Für diesen Typ muss eine Ordnung existieren. Die Funktion zeigt dann an, ob die übergebene Zahl größer/gleich 5 ist oder nicht.

```
Prelude> :t (>=5)
(>=5) :: (Num a, Ord a) => a -> Bool
Prelude> (>=5) 7
True
it :: Bool
```

Weitere prominente Beispiele wären

`(*2)`, `(2*)`	verdoppeln-Funktion
`(/2)`	halbieren-Funktion
`(1/)`	Reziprok-Funktion
`(+1)`, `(1+)`	Nachfolger-Funktion

2.2.2. Verkettungen

Verkettete Funktionen werden nacheinander ausgeführt. Es seien z.B. zwei Funktionen $f : D_f \rightarrow$

W_f und $g : D_g \to W_g$ gegeben. Die neue Funktion $h = f \circ g$ beschreibt nun die Verkettung oder Nacheinanderausführung beider Funktionen. Hier gilt nun $h : D_g \to W_f$. Man kann auch schreiben $h(x) = (f \circ g)(x) = f(g(x))$.
Auf das Argument x wird also zuerst die Funktion g angewendet, auf das Ergebnis dann die Funktion f.

```
Prelude> ((+2).(*4)) 5                          --Infix-Notation
22
it :: Integer
Prelude> (.) (+2) (*4) 5          -- Praefixnotation
22
it :: Integer
```

3. Module

Will man thematisch zusammengehörige Funktionen zusammenfassen um sie später einzusetzen, so macht das in Modulen. Dazu öffnet man einen Editor (z.B. `Notepad`).

```
1  -- ersteDatei.hs
2  -- Autor: Franz Kannichts
3  -- Datum: 13.08.2013
4  -- Zweck: erste Haskell-Uebung
5
6  module ErsteDatei
7  where
```

Wichtig:
- `-` leitet einen Kommentar ein, wird also vom Compiler nicht mit übersetzt
- Der Modulname muss mit einem Großbuchstaben beginnen
- die Datei mit der Endung `.hs` also `dateiname.hs` abspeichern

Nach dem `where` folgen die selbst definierten Funktionen, z.B.

```
8  -- Zahl incrementieren
9  inc :: Int -> Int        -- Typdeklaration (nicht zwingend erforderlich)
10 inc x = x + 1            -- Funktionsdefinition
11         -- inc x = x + 1
12         -- ^^^^^  Funktionskopf: Funktionsname und Variablen
13         --          ^^^^^^ Funktionskoerper
14
15 -- Zahl verdoppeln
16 doppelt :: Int -> Int
17 doppelt y = 2 * y
18
19 --Zahl verdoppeln und incrementieren
20 --doppInc :: Int -> Int
21 dopInc x = (*2).(-1)
22
23 -- Test auf Gleichheit dreier Zahlen
24 dreiGleiche :: Int -> Int -> Int -> Bool
25 dreiGleiche x y z = (x == y) && (x == z)
26
27 -- addMul gibt die Summe und das Produkt zweier Zahlen aus
28 -- addMul 2 3   liefert (5,6)
29 addMul :: Int -> Int -> (Int,Int)
30 -- Ausgabetyp ist ein geordnetes Paar
31 addMul x y = (x+y,x*y)
32
33 -- ersteVariable (x,y) gibt nur das x aus
34 ersteVariable :: (Int,Int) -> Int
35 ersteVariable (x,y) = x
36
37 -- Halbiert den Nachfolger
38 nachHalb :: Float -> Float
39 nachHalb = (1/).(+1)
```

Wechselt man nun in den GHCi und lädt dort die Datei `ersteDatei.hs`, kann man die darin enthaltenen Funktionen aufrufen.

```
Prelude> :load "ersteDatei.hs"
[1 of 1] Compiling ErsteDatei       ( ersteDatei.hs, interpreted )
Ok, modules loaded: ErsteDatei.
*ErsteDatei> addMul 2 3
(5,6)
it :: (Int, Int)
*ErsteDatei>
```

Am Prompt erkennt man, dass jetzt nicht mehr das Standardmodul `Prelude` läuft, sondern unser Modul `ersteDatei`.

4. Programmierbesonderheiten

4.1. Fallunterscheidungen (Wächter - Guards)

In den meisten Programmiersprachen gibt es Fallunterscheidungen durch `if...then...`-Anweisungen. Das gibt es in HASKELL auch. Am übersichtlichsten ist es jedoch die Fälle hinter einem `|` anzuordnen.

```
10 -- Fallunterscheidungen
11 -- in Haskell
12
13 module Fallunterscheidung
14 where
15
16 fall1 :: Int -> Int
17 fall1 x = if x==0 then 1 else if x>0 then 2 else 3
18
19 fall2 :: Int -> Int
20 fall2 x | x==0  =1
21        | x>0   =2
22        | x<0   =3
23
24 fall3 :: Int -> Int
25 fall3 x = case x of
26                0       -> 1
27                2       -> 2
28                4       -> 3
29                otherwise -> 4
```

Alle drei Funktionen leisten (fast) dasselbe. Bei `fall3` führt nur die genaue Eingabe von 0, 2, 4 zur Ausgabe von 1, 2 oder 3. In `fall2` steht der `|` für das `if` in der Funktion `fall1`, die nächsten Querstriche dann für das `else if` bzw. das letzte `else`. Das einzelne Gleichheitszeichen `=` entspricht dem `then`. Das `otherwise` deckt alle bis dahin nicht enthaltenen Fälle ab, also „für alle anderen Fälle tue ...“.

4.2. Mustererkennung (PatternMatching)

Mustererkennung ist ein mächtiges Hilfsmittel. Eigentlich „wandert“jeder auszuwertende Ausdruck durch das entsprechende Modul und kommt bei einem passenden Muster zur Anwendung. Das Muster steht links vom Gleichheitszeichen, rechts die auszuführende Aktion. Steht rechts wieder ein Funktionsaufruf, wird wieder links nach dem entsprechenden Muster gesucht.

```
10 istZahl :: Int -> String
11 istZahl 1 = "Eins"
12 istZahl 2 = "Zwei"
13 istZahl 3 = "Drei"
14 istZahl x = "Weder␣1,␣noch␣2,␣noch␣3"
15
16 summe :: Int -> Int
17 summe 0 = 0                                    --(summe1)
18 summe n = n + summe (n-1)                      --(summe2)
```

`istZahl` arbeitet wie `fall3` im letzten Beispiel zur Fallunterscheidung. Wichtig ist, dass der

allgemeine Fall `istZahl x` an letzter Stelle kommt. Der Compiler sucht das Programm von oben nach unten nach einem passenden Muster ab. Stünde `istZahl x` vor `istZahl 3` würde ein Aufruf von `istZahl 3` gar nicht in der entsprechenden zeile ankommen. HASKELL würde sonst erst den allgemeinen vor dem speziellen Fall bearbeiten wollen bzw. es kommt eine Fehlermeldung, dass sich Fälle überlappen.

Bei der Funktion `summe` schauen wir mal, wie der Aufruf `summe 4` im einzelnen abgearbeitet wird.

summe 4	$\rightsquigarrow$	4 + summe (4-1)	(summe2)
	$\rightsquigarrow$	4 + summe 3	(-)
	$\rightsquigarrow$	4 + (3 + summe (3-1))	(summe2)
	$\rightsquigarrow$	4 + (3 + summe 2)	(-)
	$\rightsquigarrow$	4 + (3 + (2 + summe (2-1)))	(summe2)
	$\rightsquigarrow$	4 + (3 + (2 + summe 1))	(-)
	$\rightsquigarrow$	4 + (3 + (2 + (1 + summe (1-1))))	(summe 2)
	$\rightsquigarrow$	4 + (3 + (2 + (1 + summe 0)))	(-)
	$\rightsquigarrow$	4 + (3 + (2 + (1 + 0)))	(summe1)
	$\rightsquigarrow$	4 + (3 + (2 + 1))	(+)
	$\rightsquigarrow$	4 + (3 + 3)	(+)
	$\rightsquigarrow$	4 + 6	(+)
	$\rightsquigarrow$	10	

Ganz rechts steht jeweils die Programmzeile bzw. die eingebaute Funktion, die gerade abgearbeitet wurde. Das „$\rightsquigarrow$“ hat die Bedeutung „... wird ausgewertet zu ...“.
In dieser Form muss man die Auswertung von Programmteilen beherrschen. So kennt man deren Funktionalität oder auch Fehler. Bei dieser Funktion erkennt man, dass die Funktion selber auch wieder im Funktionskörper aufgerufen werden kann. Dieses Vorgehen ist Inhalt des Kapitels über die Rekursion.

5. Rekursion

Während es in anderen Programmiersprachen verschiedene Schleifen gibt, ist bei HASKELL die einzige Wiederholungsmöglichkeit die Rekursion.

1. Rekursionsschritt:
   Dabei wird die Funktion im Funktionskörper noch einmal mit einem Vorgänger aufgerufen. Das würde unendlich lange geben, da es immer einen Vorgänger gibt.
2. Terminierender Fall:
   Also muss es eine Abbruchbedingung geben, den sogenannten terminierenden Fall. Dieser muss im Programm weiter vorne stehen, damit er per Mustererkennung immer wieder überprüft wird.

Das ist bei der Funktion `summe` auf Seite 4 - 2 so der Fall.

5.1. Endrekursion

Würde man die Funktion `summe` auf eine große Zahl anwenden, würden immer mehr Funktionsaufrufe auf dem Stack (Stapel) zwischengespeichert werden, ehe der terminierende Fall erreicht und die Funktion endgültig ausgewertet werden kann. Deshalb wäre es besser, das bereits erreichte Ergebnis zwischenzuspeichern und ständig zu aktualisieren.

Definition: Endrekursive Funktion

Ein rekursive Funktion heißt endrekursiv, wenn für jedes $x \epsilon D_f$ der rekursive Aufruf von $f(x)$ der letzte Schritt zur Berechnung von $f(x)$ ist.

Dazu verwendet man ein interne Hilfsfunktion (`summe'`), die den Zwischenspeicher (`akk`) als zweiten Parameter mitführt.

```
1 summeE :: Int -> Int
2 summeE n = summe' n 0                                    --(summe0)
3                    where
4                     summe' 0 akk = akk                   --(summe1)
5                     summe' n akk = summe' (n-1) (akk + n) --(summe2)
```

summe 4	$\rightsquigarrow$	summe' 4 0	(summe0)
	$\rightsquigarrow$	summe' (4-1) (0 + 4)	((summe2)
	$\rightsquigarrow$	summe' 3 4	(-)(+)
	$\rightsquigarrow$	summe' (3-1) (4+3)	(summe2)
	$\rightsquigarrow$	summe' 2 7	(-)(+)
	$\rightsquigarrow$	summe' (2-1) (7+2)	(summe2)
	$\rightsquigarrow$	summe' 1 9	(-)(+)
	$\rightsquigarrow$	summe' (1-1) (9-1)	(summe2)
	$\rightsquigarrow$	summe' 0 10	(-)(+)
	$\rightsquigarrow$	10	(summe1)

So läuft die Rekursion deutlich speichersparender ab.

6. Listen

Eine Liste ist eine Folge von *gleichartigen* Objekten. Listen treten beim Modellieren realer Situationen durch Programme sehr oft auf, z.B.

- ein Telefonbuch ist eine Liste von Paaren, bestehend aus Name und Telefonnummer
- ein Text ist eine Liste von Wörtern (Strings), die wiederum eine Liste von Buchstaben (char) sind
- ein Bild ist eine Liste von Punkten (pixeln)

6.1. Darstellung von Listen [ ]

Listen realisiert man durch Einfügen der Elemente in eckige Klammern oder bei Strings durch Anführungsstriche oben.

```
Prelude> [1,2,3]
[1,2,3]
it :: [Integer]
Prelude> ['a','b','c']
"abc"
it :: [Char]
Prelude> "abc"
"abc"
it :: [Char]
Prelude> [(1,'a'),(2,'b'),(3,'c')]
[(1,'a'),(2,'b'),(3,'c')]
it :: [(Integer, Char)]
```

Für sortierte Listen gibt es noch eine vereinfachte Darstellungsmöglichkeit.

```
Prelude> [2..10]
[2,3,4,5,6,7,8,9,10]
it :: [Integer]
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
it :: [Integer]
Prelude> ['b','e'..'m']
"behk"
it :: [Char]
```

Elegant ist auch die Möglichkeit der Listenbeschreibung, wie man sie vielleicht aus der Mengenlehre kennt.

```
Prelude> [(x,y)|x<-[1..10],y<-[1..10],x+y==10||x-y==3]
[(1,9),(2,8),(3,7),(4,1),(4,6),(5,2),(5,5),(6,3),(6,4),(7,3),(7,4),
(8,2),(8,5),(9,1),(9,6),(10,7)]
it :: [(Integer, Integer)]
```

Es werden alle Zahlenpaare (x, y) mit $x \epsilon [1, 10]$ und $y \epsilon [1, 10]$ ausgegeben, deren Summe 10 oder deren Differenz 3 ist.

6.2. Elementare Operation auf Listen (:)

Es gibt eine elementare Operationen auf Listen, nämlich das vorne Anfügen eines Elementes an eine Liste mittels des Cons-Operators `:` .

```
Prelude> 1:[2,3]
[1,2,3]
it :: [Integer]
Prelude> 'a':['b','c']
"abc"
it :: [Char]
Prelude> 'a':"bc"
"abc"
it :: [Char]
```

Für das Programmieren wichtig kann man nun Listen „auseinandernehmen“.

Liste	Beschreibung	⇝ Ergebnis
`[]`	leere Liste	⇝ `[]`
`1:[]`	einelementige Liste	⇝ `[1]`
`x:[]`	variable einelementige Liste	⇝ `[x]`
`1:2:[]`	zweielementige Liste	⇝ `[1,2]`
`(x:xs)`	Liste mit erstem Element `x`	⇝

`x` und `xs` snd hierbei variable Bezeichnungen. Man könnte auch schreiben `(x:restliste)` . An dem Cons-Operator `:` erkennt HASKELL , dass das Letzte, also `xs` oder `restliste` eine Liste sein muss.

`(x:y:restliste)`	mindestens zweielementige Liste	⇝

Die Klammern `(x:xs)` und `(x:y:restliste)` sind wichtig, da sich das `x` von dem eventuell davorstehenden Operator gegriffen werden würde.

6.3. Weitere Listenoperationen

Mittels Cons-Operator können nun weitere Listenfunktionen konstruiert werden, die z.T. in HASKELL schon eingebaut sind.

Kopf einer Liste und Schwanz einer Liste (head and tail)

Der Kopf einer Liste ist das erste Element, der Schwanz der Rest der Liste.

```
1  --Autor:
2  --Datum: 15.10.2013
3  --Zweck: Funktionen auf Listen
4
5  module Listen where
6
7  kopf :: [a] -> a
8  kopf [] = error"Die␣Liste␣ist␣leer"
9  kopf (x:xs) = x
10
11 schwanz :: [a] -> [a]
12 schwanz [] = []
13 schwanz (x:xs) = xs
```

z.B. `kopf [3,1,6]` $\rightsquigarrow$ `3`, `schwanz "Ameise"` $\rightsquigarrow$ `"meise"`

Die meisten weiteren Listenfunktionen ergeben sich rekursiv.

Länge einer Liste (length)

Nimmt den Kopf weg, zählt 1, nimmt den Kopf vom Rest, zählt 1 dazu, nimmt den Kopf vom neuen Rest, zählt wieder 1 dazu usw. bis die Liste leer ist.

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

z.B. `laenge "Ameise"` $\rightsquigarrow$ `6`.

Letztes Element (last)

```
19 letztes :: [a] -> a
20 letztes [] = error"kein␣Element␣enthalten"
21 letztes [x] = x
22 letztes (x:xs) = letztes xs
```

Anfang einer Liste (init)

liefert die Liste ohne das letzte Element.

```
34 anfang :: [a] -> [a]
35 anfang [] = error"Liste␣ist␣leer"
36 anfang [x] = []
37 anfang (x:y:xs) = x : anfang (y:xs)
```

GibNtes (!!)

gibt das n-te Element einer Liste aus. Üblicherweise beginnt die Zählung bei Null.

```
39 gibNtes :: Int -> [a] -> a
40 gibNtes 0 (x:restliste) = x
41 gibNtes n (x:restliste) = gibNtes (n-1) restliste
```

Also ergibt dann `gibNtes 3 [1,2,3,4,5]` ⇝ `4` . Das Gleiche liefert auch `[1,2,3,4,5] !! 3`.

Verbinden zweier Listen (++)

Hierbei wird die vordere Liste nach und nach in ihrer Einzelteile zerlegt und dann an die zweite Liste angeconst. Beide Listen sind dann zu einer verbunden (concateniert).

```
43 verbinde :: [a] -> [a] -> [a]
44 verbinde [] ys = ys
45 verbinde (x:xs) ys = x : verbinde xs ys
```

Im GHCi ruft man auf `[1,2,3,4]++[3,4,5,6]` ⇝ `[1,2,3,4,3,4,5,6]` oder das ganze mit `verbinde`.

6.4. Aufgaben

Nachdem wir einige Funktionen implementiert und getestet haben, sollen nun die folgenden Funktionen programmiert und gestestet werden. Schreibe auch zu jeder Funktion die Typdeklaration (notfalls mit schummeln auf der Konsole s. Anfang des Skriptes). Versuche die Funktionen mit verschiedenen Methoden zu implementieren (Wächter, Akkutechnik).

1. `quadlist liste` ⇝ Quadrate der Listenelemente, die natürlich nur Zahlen sein können
2. `istEnthalten x liste` testet, ob x in der Liste ist
3. `delErstes x liste` löscht das erste Auftreten von x in der Liste
4. `delElem x liste` löscht alle x aus der Liste
5. `alleGleich liste` testet, ob alle Elemente der Liste gleich sind
6. `streichGleiche liste` löscht alle Mehrfachvorkommen von Elementen aus der Liste, also `streicheGleiche "Hallohoho"⇝"Halo"`
   Hierdurch wird aus einer Liste eine Menge. In einer Menge kommt jedes Element nur einmal vor!

Die folgenden Funktionen beziehen sich auf solche Mengen.

7. `vereinigung menge1 menge2` ⇝Vereingungsmenge beider Mengen
8. `durchschnitt menge1 menge2` ⇝Durchschnittsmenge beider Mengen

Nun seien die Listen wieder beliebig.

9. `insert x liste` schreibt x zwischen alle Listenelemente
10. `replaceAllElem x y liste` ersetzt jedes Auftreten von x durch y
11. `verdoppleElem liste` verdoppelt alle Elemente, also
    `verdoppleElem [1,2,3]` ⇝ `[1,1,2,2,3,3]`
12. `verdoppleListe liste` verdoppelt die Liste,
    `verdoppleListe [1,2,3]` ⇝ `[1,2,3,1,2,3]`

6.5. Eingebaute Listenfunktionen

Zum Glück haben die HASKELL -Macher viele Listenfunktionen in der Art wie oben schon vorbereitet.

Befehl	Funktion	Aufruf		Ergebnis
`head`	gibt den Kopf der Liste	`head [1,2,3,4]`	⇝	`1`
`tail`	gibt den Schwanz der Liste	`tail [1,2,3,4]`	⇝	`[2,3,4]`
`length`	gibt die Länge einer Liste	`length [5,6,7,8]`	⇝	`4`
`null`	zeigt an, ob die Liste Leer ist	`null [1,2,3]`	⇝	`False`
`last`	gibt das letzte Element	`last [5,6,7,8]`	⇝	`8`
`init`	Liste ohne letztes Element	`init [1,2,3,4]`	⇝	`[1,2,3]`
`!!`	gibt das Element mit dem n-ten Index zurück, die Liste beginnt mit dem nullten Element	`[1,2,3,4,5] !! 3`	⇝	`4`
`take`	liefert die ersten n Elemente	`take 3 "Hallo"`	⇝	`"Hal"`
`drop`	löscht die ersten n Elemente	`drop 3 "Hallo"`	⇝	`"lo"`
`sum`	liefert die Summe der Listenelemente	`sum [2,4,6]`	⇝	`12`
`reverse`	dreht eine Liste um	`reverse "abcd"`	⇝	`"dcba"`
`product`	liefert das Produkt der Listenelemente	`product [2,4,6]`	⇝	`48`
`maximum`	liefert das größte Element einer Liste	`maximum [2,5,3,7,1]`	⇝	`7`
`minimum`	liefert das kleinste Element	`minimum "zahnweh"`	⇝	`'a'`
`++`	verbinden zweier Listen	`"zwerg" ++ "nase"`	⇝	`"zwergnase"`
`map`	wendet eine Funktion f auf jedes Listenelement an	`map (*3) [1,2,3,4]`	⇝	`[3,6,9,12]`
`filter`	liefert die Elemente, für die die übergebene Bedingung war ist	`filter (<5) [1..10]`	⇝	`[1,2,3,4]`

Weitere noch zu kommentierende Funktionen wären
concat, concatMap, foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1, iterate, repeat, replicate, cycle, splitAt, takeWhile, dropWhile, span, break, lines, words, unlines, unwords, and, or, any, all, elem, notElem, lookup,

zip, zip3, zipWith, zipWith3, unzip, unzip3

7. Höhere Listenfunktionen

Bisher hatten wir Funktionen definiert, die Basisdatentypen oder Tupel oder Listen von Basisdatentypen als Eingabeparameter besaßen.

Es gibt auch Funktionen, die andere Funktionen als Eingabeparameter bekommen.

7.1. Verkettung und Currying

Im Grunde war die Verkettung `.` schon solch ein Fall. Es gilt ja $(f.g)x = f\big(g(x)\big)$. Die interne Definition der Verkettung kann so aussehen:

```
1 (.) :: (b->c) -> (a->b) -> a -> c
2 (.) f g x = f(g x)
```

Aufgabe 7.1: Typsignatur der Verkettung

Erläutere diese Typsignatur.

Andererseits liefert jede Funktion mit mehr als einem Eingabeparameter eine Funktion als Ausgabe. Dies liegt an der Linksassoziativität der Auswertung.

Beispiel 7.1 : Currying

Nehmen wir eine Funktion f mit den Eingabeparamern a, b, c, d und e.
Es gilt: $f\,a\,b\,c\,d\,e = \Big(\big(\big((f\,a)\;b\big)\;c\big)\;d\Big)\;e$

Die Funktion f zieht sich mit dem *Curry*-Operator $\_$ das Argument a heran. Es entsteht eine Funktion $(f\,a)$, die nun noch 4 Argumente erwartet.
Diese zieht sich nun das Argument b heran und es entsteht eine Funktion $\big((f\,a)\;b\big)$, die noch 3 Argumente erwartet, ...

Jede Funktion f verarbeitet also zunächst immer nur ein Argument, um dann als neue Funktion ggf. ein weiteres zu konsumieren.

Dieses Verhalten nennt man nach Haskell Brooks Curry auch *Currying*.

7.2. Mapping

Hierbei werden alle Elemente einer Liste an eine Funktion f geschickt, von dieser verarbeitet und die Ergebnisse in die Ergebnisliste gesteckt.

```
1 verdopple :: Num b => [Num] -> [Num]
2 verdopple liste = map (*2) liste
```

`verdopple [1,2,3,4]` $\rightsquigarrow$ `[2,4,6,8]`

Die Funktion `map` ist in der Standardumgebung enthalten, lässt sich aber auch leicht selber bauen.

```
1 mapList :: (t->t1) -> [t] -> [t1]
2 mapList _ []       = []                          _ Platzhalter
3 mapList f (k:rest) = f k : mapList f rest     -- f Funktion
```

Oder elegant so

```
4 mapList1 f liste = [f x | x <- liste]
```

7.3. Filter

Die höhere Listenfunktion `filter` untersucht jedes Listenelement auf eine bestimmte Eigenschaft und schreibt im true-Fall dieses Element in die Ergebnisliste.
`filter` ist in der Standardumgebung enthalten.

```
1 groesser2 (Num a, Ord a) => [a] -> [a]
2 groesser2 liste = filter (>2) liste
```

`groesser [1,2,3,4]` ⇝ `[3,4]`

Auch `filter` kann man nachbauen:

```
1 filterList :: (a->Bool) -> [a] -> [a]
2 filterList p []       = []
3 filterList p (k:rest) | p k = k : filterList p rest
4                               -- wenn p k == True
5                       | otherwise = filterList p rest
6
7 filterList1 p liste = [x | x <- liste, p x]
```

`p` ist das Prädikat, nach dem untersucht wird.

7.4. Faltungen

Durch Faltungen lassen sich Ausdrücke besonders komprimieren. Diese Thema kann bei Interesse im Selbststudium erarbeitet werden. Hierzu existieren die eingebauten Funktionen `foldr` und `foldl`.

7.5. Aufgaben

Löse die Aufgaben jeweils mit `map` oder `filter`.

1. Erstelle eine Liste aller Quadratzahlen bis n. `quadratBis 5` ⇝ `[1,4,9,16,25]`
2. Bestimme die Reste bzgl. der Division durch eine Zahl.
   `resteA 5 [1,2,3,4,5,6,7]` ⇝ `[1,2,3,4,0,1,2]`
   Probiere auch folgende Ausgabe `resteB 5 [1,2,3,4,5,6,7]` ⇝ `[0,1,2,1,0,6,7]`
3. Welche Zahlen sind Null? `istNull [0,1,2,0,1,0]` ⇝ `[0,0,0]`
4. Gib alle Teiler einer Zahl aus. `teiler 12` ⇝ `[1,2,3,4,6,12]`

5. Bestimme die Länge der einzelnen Wörter einer Liste.
   `wortlaenge ["filter","contra","map"]` ⇝ `[6,6,3]`
6. Gib alle Zahlen einer Liste aus, die zwischen a und b liegen.
   `zwischen 3 7 [1..10]` ⇝ `[4,5,6]`
7. Bestimme die Anzahl eines Elementes in einer Liste.
   `wieviele 'l' "Killerwal"` ⇝ `3`
8. Dubliziere eine Liste. `duble "cdefg"` ⇝ `"ccddeeffgg"`
   (Die Funktion `concat` verschmilzt eine Liste von Listen zu einer Liste.)

7.6. Weitere höhere Listenfunktionen

Weitere Funktionen auf Listen, die eine Funktion als zusätzliches Argument enthalten sind:

- `dropWhile :: (a -> Bool) -> [a] -> [a]`
  Von vorne beginnend werden solange Elemente der Liste ausgelassen,wie die Bedingungsfunktion `True` ergibt. Dann wird die Restliste zurück gegeben.
  `dropWhile (<4) [1..8]` ⇝ `[4,5,6,7,8]`
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
  Die Ergebnisliste besteht aus den ersten Elementen der Parameterliste, solange die Bedingungsfunktion `True` ergeben hat.
  `takeWhile (<4) [1..10]` ⇝ `[1,2,3]`
- `until :: (a -> Bool) -> (a->a) -> a -> a`
  Eine Funktion wird solange wiederholt auf den Anfangswert angewendet, bis die Bedingungsfunktion `True` liefert.
  `until (>10) (*2) 2` ⇝ `16`
- `zip :: [a] -> [b] -> [(a,b)]`
  Macht aus zwei Listen eine 2-Tupel-Liste.
  `zip [1,2,3] "abcd"` ⇝ `[(1,'a'),(2,'b'),(3,'c')]`
- `zipWith :: (a->b->c) -> [a] -> [b] -> [c]`
  Eine zweistellige Funktion wird paarweise auf die Parameterlisten angewendet, das jeweilige Ergebnis in der Ergebnisliste ausgegeben.
  `zipWith (+) [1,2,3] [4,5,6]` ⇝ `[5,7,9]`

8. Typklassen

Bisher haben wir die Typsignatur einer Funktion immer nur auf einen konkreten Datentypen (meist `Int`) beschränkt.

Viele Funktionen, insbesondere die Listenfunktionen arbeiten aber auch auf anderen Datentypen. Die Typsignatur müsste allgemeiner verfasst werden.

Die Funktion `haengAn` hängt an eine Liste hinten ein Element an. Die einzige Bedingung ist, das dieses Element vom gleichen Typ wie die übrigen Listenelemente sein muss.

```
Listen> :t haengAn   -- Typsignaturabfrage liefert
haengAn :: [a] -> a -> [a]
```

Die Funktion `gibNtes` soll das n-te Element einer Liste zurückgeben.

```
Listen> :t gibNtes   -- Typsignaturabfrage liefert
gibNtes :: Int -> [a] -> a
```

`a` ist der allgemeine Typ, das n muss aber vom Typ `Int` sein.

Die Funktion `istEnthalten` überprüft, ob ein bestimmtes Element in einer Liste enthalten ist.

```
Listen> :t istEnthalten   -- Typsignaturabfrage liefert
istEnthalten :: Eq a => a -> [a] -> Bool
```

In dieser Typsignatur taucht nun die explizite Angabe einer Typklasse auf, nämlich `Eq a`. Dies besagt, das die Funktion nur auf Datentypen arbeitet, die zur Typklasse `Eq` gehören, deren Elemente vergleichbar sind. Für sie müssen die Operationen `==` und `/=` definiert sein. Um das abzusichern, wird in der Typsignatur darauf verwiesen.

So gibt es mehrere Typklassen, die Datentypen mit bestimmten Eigenschaften zusammenfassen.

Eine besonders wichtige Typklasse ist `Show a`. Sie fasst alle Datentypen zusammen, die sich in Strings konvertieren lassen und somit auf der Konsole ausgeben lassen. Dies trifft natürlich auf alle in Haskell eingebauten Datentypen zu.

Einige wichtige Typklassen sind hier zusammengefasst:

`Typklasse`	bereitgestellte oder bereitzustellende Funktionalitäten	zugehörige Datentypen
`Show`	show	Int, Integer, Char, String, Bool,Float,Double
`Eq`	==, /=	Int, Integer, Char, String, Bool,Float,Double
`Ord`	<, >, ==, /=, <=, >=	Int, Integer, Char, String, Bool,Float,Double
`Num`	+, -, *, / , div, mod, ^ u.a.	Int, Integer, Float,Double
`Enum`	Aufzählungstypen	Int, Integer, Char, String, Bool,Float,Double
`Read`	read (Umkehr zu show)	Int, Integer, Char, String, Bool,Float,Double
`Bounded`	Minimum, Maximum	Char, Bool

A. Eingebaute Funktionen

```
Prelude> :browse
($!) :: (a -> b) -> a -> b
(!!) :: [a] -> Int -> a
($) :: (a -> b) -> a -> b
(&&) :: Bool -> Bool -> Bool
(++) :: [a] -> [a] -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
(=<<) :: Monad m => (a -> m b) -> m a -> m b
data Bool = False | True
class Bounded a where
  minBound :: a
  maxBound :: a
data Char = GHC.Types.C# GHC.Prim.Char#
data Double = GHC.Types.D# GHC.Prim.Double#
data Either a b = Left a | Right b
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
type FilePath = String

data Float = GHC.Types.F# GHC.Prim.Float#
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  sqrt :: a -> a
  log :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  tan :: a -> a
  cos :: a -> a
  asin :: a -> a
  atan :: a -> a
  acos :: a -> a
  sinh :: a -> a
  tanh :: a -> a
  cosh :: a -> a
  asinh :: a -> a
  atanh :: a -> a
  acosh :: a -> a
class Num a => Fractional a where
  (/) :: a -> a -> a
```

```
  recip :: a -> a
  fromRational :: Rational -> a
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (GHC.Base.<$) :: a -> f b -> f a
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
type IOError = GHC.IO.Exception.IOException
data Int = GHC.Types.I# GHC.Prim.Int#
data Integer
  = integer-gmp:GHC.Integer.Type.S# GHC.Prim.Int#
  | integer-gmp:GHC.Integer.Type.J# GHC.Prim.Int# GHC.Prim.ByteArray#
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
data Maybe a = Nothing | Just a
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
data Ordering = LT | EQ | GT
type Rational = GHC.Real.Ratio Integer
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::
    Text.ParserCombinators.ReadPrec.ReadPrec [a]
type ReadS a = String -> [(a, String)]
class (Num a, Ord a) => Real a where
```

```
  toRational :: a -> Rational
class (RealFrac a, Floating a) => RealFloat a where
  floatRadix :: a -> Integer
  floatDigits :: a -> Int
  floatRange :: a -> (Int, Int)
  decodeFloat :: a -> (Integer, Int)
  encodeFloat :: Integer -> Int -> a
  exponent :: a -> Int
  significand :: a -> a
  scaleFloat :: Int -> a -> a
  isNaN :: a -> Bool
  isInfinite :: a -> Bool
  isDenormalized :: a -> Bool
  isNegativeZero :: a -> Bool
  isIEEE :: a -> Bool
  atan2 :: a -> a -> a
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
type ShowS = String -> String
type String = [Char]
(^) :: (Num a, Integral b) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
all :: (a -> Bool) -> [a] -> Bool
and :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
appendFile :: FilePath -> String -> IO ()
asTypeOf :: a -> a -> a
break :: (a -> Bool) -> [a] -> ([a], [a])
concat :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
const :: a -> b -> a
curry :: ((a, b) -> c) -> a -> b -> c
cycle :: [a] -> [a]
drop :: Int -> [a] -> [a]

dropWhile :: (a -> Bool) -> [a] -> [a]
either :: (a -> c) -> (b -> c) -> Either a b -> c
elem :: Eq a => a -> [a] -> Bool
error :: [Char] -> a
even :: Integral a => a -> Bool
filter :: (a -> Bool) -> [a] -> [a]
flip :: (a -> b -> c) -> b -> a -> c
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl1 :: (a -> a -> a) -> [a] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
fromIntegral :: (Integral a, Num b) => a -> b
fst :: (a, b) -> a
gcd :: Integral a => a -> a -> a

getChar :: IO Char
getContents :: IO String
getLine :: IO String
head :: [a] -> a
id :: a -> a
init :: [a] -> [a]
interact :: (String -> String) -> IO ()
ioError :: IOError -> IO a
iterate :: (a -> a) -> a -> [a]
last :: [a] -> a
lcm :: Integral a => a -> a -> a
length :: [a] -> Int
lex :: ReadS String
lines :: String -> [String]
lookup :: Eq a => a -> [(a, b)] -> Maybe b
map :: (a -> b) -> [a] -> [b]
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
maximum :: Ord a => [a] -> a
maybe :: b -> (a -> b) -> Maybe a -> b
minimum :: Ord a => [a] -> a
not :: Bool -> Bool
notElem :: Eq a => a -> [a] -> Bool
null :: [a] -> Bool
odd :: Integral a => a -> Bool
or :: [Bool] -> Bool
otherwise :: Bool
print :: Show a => a -> IO ()
product :: Num a => [a] -> a
putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
read :: Read a => String -> a
readFile :: FilePath -> IO String
readIO :: Read a => String -> IO a
readLn :: Read a => IO a
readParen :: Bool -> ReadS a -> ReadS a
reads :: Read a => ReadS a
realToFrac :: (Real a, Fractional b) => a -> b
repeat :: a -> [a]
replicate :: Int -> a -> [a]
reverse :: [a] -> [a]
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr1 :: (a -> a -> a) -> [a] -> [a]
seq :: a -> b -> b
sequence :: Monad m => [m a] -> m [a]
```

```
sequence_ :: Monad m => [m a] -> m ()
showChar :: Char -> ShowS
showParen :: Bool -> ShowS -> ShowS
showString :: String -> ShowS
shows :: Show a => a -> ShowS
snd :: (a, b) -> b
span :: (a -> Bool) -> [a] -> ([a], [a])
splitAt :: Int -> [a] -> ([a], [a])
subtract :: Num a => a -> a -> a
sum :: Num a => [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
uncurry :: (a -> b -> c) -> (a, b) -> c
undefined :: a
unlines :: [String] -> String
until :: (a -> Bool) -> (a -> a) -> a -> a
unwords :: [String] -> String
unzip :: [(a, b)] -> ([a], [b])
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
userError :: String -> IOError
words :: String -> [String]
writeFile :: FilePath -> String -> IO ()
zip :: [a] -> [b] -> [(a, b)]
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
(||) :: Bool -> Bool -> Bool
```