Praktische Informatik 3

Christoph Lüth

WS 01/02

Vorlesung vom 22.10.2001

Personal

Vorlesung: Christoph Lüth <cxl>
MZH 8120, Tel. 7585

Stud. Tutoren: Christoph Grimmer <crimson>
Thomas Meyer <mclee>
Peter König <pkoenig>
Rafael Trautmann <pirate>

www.informatik.uni-bremen.de/~cxl/pi3
(ab morgen!)

Termine

- Vorlesung:

  Mo 10-12, kleiner Hörsaal („Keksdose“)

- Tutorien:

Di	08-10	MZH 6240	Christoph Grimmer
Di	08-10	MZH 5290	Thomas Meyer
Do	13-15	MZH 7210	Peter König
Do	13-15	MZH 7200	Rafael Trautmann

Übungsbetrieb

- Ausgabe der Übungsblätter über die Website im direkten Anschluß an die Vorlesung.
- Besprechung der Übungsblätter in den Tutorien;
- Bearbeitungszeit eine/zwei Wochen ab Tutorium, Abgabe im Tutorium;
- Voraussichtlich zehn Übungsblätter.

Inhalt der Veranstaltung

- Deklarative und funktionale Programmierung
  - Betonung auf Konzepten und Methodik
- Bis Weihnachten: Grundlagen
  - Funktionen, Typen, Funktionen höherer Ordnung, Polymorphie
- Nach Weihnachten: Ausweitung und Anwendung
  - Prolog und Logik; Nebenläufigkeit/GUIs; Grafik und Animation
- Lektüre:
  Simon Thompson: *Haskell — The Craft of Functional Programming* (Addison-Wesley, 1999)

Scheinrelevanz

„Der in der DPO'93 aufgeführte prüfungsrelevante PI3-Schein kann nicht nur über das SWP sondern alternativ auch über PI3 abgedeckt werden. Die in der DPO zusätzlich aufgeführte Forderung der erfolgreichen Teilnahme am SWP bleibt davon unberührt.“

Scheinrelevanz

„Der in der DPO'93 aufgeführte prüfungsrelevante PI3-Schein kann nicht nur über das SWP sondern alternativ auch über PI3 abgedeckt werden. Die in der DPO zusätzlich aufgeführte Forderung der erfolgreichen Teilnahme am SWP bleibt davon unberührt."

Mit anderen Worten:

- Entweder prüfungsrelevante Studienleistung in PI3 sowie erfolgreiche Teilnahme an SWP
- oder Prüfungsrelevante Studienleistung in SWP

Scheinkriterien — Vorschlag:

- Ein Übungsblatt ist bestanden, wenn mindestens 40% der Punktzahl erreicht wurden.
- Von n ausgegebenen Übungsblättern $n - 1$ bestanden und in diesen 60% der Punkte erreicht.
- Gegebenfalls findet ein Prüfungsgespräch statt:
  - Auf Wunsch des Studenten
  - Tutor kann Individualität der Leistung nicht bezeugen

Warum funktionale Programmierung lernen?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen
- FP konzentriert sich auf wesentlichen Elemente moderner Programmierung:
  - Typisierung und Spezifikation
  - Datenabstraktion
  - Modularisierung und Dekomposition
- Blick über den Tellerrand — Blick in die Zukunft
  - Studium $\neq$ Programmierkurs— was kommt in 10 Jahren?

Programme als Funktionen

$$P : Eingabe \rightarrow Ausgabe$$

Keine Variablen — keine Zustände

Alle Abhängigkeiten explizit:

Ausgabe eines Programms hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext

Referentielle Transparenz

Geschichtliches

- Grundlagen 1920/30

  Kombinatorlogik und λ-Kalkül (Schönfinkel, Curry, Church)

- Erste Sprachen 1960

  LISP (McCarthy), ISWIM (Landin)

- Weitere Sprachen 1970– 80

  FP (Backus); ML (Milner, Gordon), später SML und CAML; Hope (Burstall); Miranda (Turner)

- 1990: Haskell als Standardsprache

Programme als Funktionen

Programmieren durch Rechnen (mit Symbolen):

$$\begin{aligned} 5 * (7 - 3) + 4 * 3 &= 5 * 4 + 12 \\ &= 20 + 12 \\ &= 32 \end{aligned}$$

Benutzt Gleichheiten ($7 - 3 = 4$ etc.), die durch Definition von $+$, $*$, $-$, . . . gelten.

Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

```
addDouble x y = 2 * (x+ y)
```

- Auswertung durch Reduktion von Ausdrücken:

$$\texttt{addDouble 6 4} \rightsquigarrow \texttt{2*(6+ 4)} \rightsquigarrow 20$$

  - Anwendung der Gleichungen:
    von links nach rechts, außen nach innen.
- Nichtreduzierbare Ausdrücke sind Werte
  - Zahlen, Zeichenketten, Wahrheitswerte, . . .

Typisierung

Typen unterscheiden Arten von Ausdrücken

- Basistypen
- strukturierte Typen (Listen, Tupel, etc)

Wozu Typen?

- Typüberprüfung während Übersetzung erspart Laufzeitfehler
- Programmsicherheit

Übersicht: Typen in Haskell

Ganze Zahlen	`Int`	`0 94 -45`
Fließkomma	`Double`	`3.0 3.141592`
Zeichen	`Char`	`'a' 'x' '\034' '\n'`
Zeichenketten	`String`	`"yuck" "hi\nho\"\n"`
Wahrheitswerte	`Bool`	`True False`
Listen	`[a]`	`[6, 9, 20]`
		`["oh", "dear"]`
Tupel	`(a, b)`	`(1, 'a') ('a', 4)`
Funktionen	`a-> b`	

Definition von Funktionen

Zwei wesentliche Konstrukte:

- Fallunterscheidung
- Rekursion

Beispiel:

```
fac :: Int-> Int
fac n = if n == 0 then 1
           else n * (fac (n-1))
```

Auswertung kann divergieren!

Haskell in Aktion: hugs

hugs ist ein Haskell-Interpreter

Klein, schnelle Übersetzung, gemächliche Ausführung

Funktionsweise:

- hugs liest Definitionen (Programme, Typen, . . .) aus Datei
- Kommandozeilenmodus: Reduktion von Ausdrücken
- Keine Definitionen in der Kommandozeile

Zusammenfassung

- Haskell ist eine funktionale Programmiersprache
- Programme sind Funktionen, definiert durch Gleichungen

  Referentielle Transparenz — keine Zustände oder Variablen
- Ausführung durch Reduktion von Ausdrücken
- Typisierung:
  - Basistypen: Zahlen, Zeichen(ketten), Wahrheitswerte
  - Strukturierte Typen: Listen, Tupel
  - Jede Funktion `f` hat eine Signatur `f :: a-> b`

Vorlesung vom 29.10.2001

Inhalt

- Funktionsdefinitionen
- Numerische Basisdatentypen
- Von der Spezifikation zum Programm
- Strukturierte Datentypen: Listen und Tupel

Die Abseitsregel

$$\mathtt{f}\,\mathtt{x}_1\,\mathtt{x}_2\ldots\mathtt{x}_n = \mathtt{E}$$

- Gültigkeitsbereich der Definition von `f`:
  alles, was gegenüber `f` eingerückt ist.
- Beispiel:

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

- Gilt auch verschachtelt.

Bedingte Definitionen

Statt verschachtelter Fallunterscheidungen . . .

```
f x y = if B1 then P else
        if B2 then Q else ...
```

. . . bedingte Gleichungen:

```
f x y
  | B1 = ...
  | B2 = ...
```

Auswertung der Bedingungen von oben nach unten

Wenn keine Bedingung wahr ist: Laufzeitfehler! Deshalb:

```
  | otherwise = ...
```

Kommentare

- Pro Zeile:
  - Ab -- bis Ende der Zeile

```
f x y = irgendwas  -- und hier der Kommentar!
```

- Über mehrere Zeilen:
  - Anfang {-, Ende -}

```
{-
   Hier fängt der Kommentar an
   erstreckt sich über mehrere Zeilen
   bis hier                              -}
f x y = irgendwas
```

Die Wahrheitswerte: `Bool`

- Werte `True` und `False`
- Funktionen:

```
not :: Bool-> Bool              Negation
&&  :: Bool-> Bool-> Bool       Konjunktion
||  :: Bool-> Bool-> Bool       Disjunktion
```

- Beispiel: ausschließende Disjunktion:

```
exOr :: Bool-> Bool-> Bool
exOr x y = (x || y) && (not (x && y))
```

- Alternative:

```
exOr x y
  | x == True  = if y == False then True
                                else False
  | x == False = if y == True  then True
                                else False
```

- Alternative:

```
exOr x y
  | x == True  = if y == False then True
                                else False
  | x == False = if y == True  then True
                                else False
```

- Igitt! Besser: Definition mit pattern matching

```
exOr True  y = not y
exOr False y = y
```

Exkurs: Operatoren in Haskell

- Operatoren: Namen aus Sonderzeichen `!$%&/?+^ ...`
- Werden infix geschrieben: `x && y`
- Ansonsten normale Funktion.
- Andere Funktion infix benutzen:

  ```
  x `exOr` y
  ```

  In Apostrophen einschließen.
- Operatoren in Nicht-Infixschreibweise (präfix):

  ```
  (&&) :: Bool-> Bool-> Bool
  ```

  In Klammern einschließen.

Das Rechnen mit Zahlen

Grundsätzliches Problem: es gibt so viele . . .

Beschränkte Genauigkeit, konstanter Aufwand $\longleftrightarrow$ beliebige Genauigkeit, wachsender Aufwand

Das Rechnen mit Zahlen

Grundsätzliches Problem: es gibt so viele . . .

Beschränkte Genauigkeit, konstanter Aufwand $\longleftrightarrow$ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte ($\geq$ 31 Bit)
- `Integer` - beliebig große ganze Zahlen
- `Rational` - beliebig genaue rationale Zahlen
- `Float` - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: `Int` und `Integer`

- Nützliche Funktionen (überladen, auch für `Integer`):

```
(+), (*), (^), (-) :: Int-> Int-> Int
abs    :: Int-> Int -- Betrag
div    :: Int-> Int-> Int
mod    :: Int-> Int-> Int
```

  Es gilt `(x `div` y)*y + x `mod` y == x`

- Vergleich durch ==, /=, <=, <, . . .
- Achtung: Unäres Minus
  - Unterschied zum Infix-Operator `(-)`
  - Im Zweifelsfall klammern: `abs (-34)`

Fließkommazahlen: `Double`

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen
  - siehe Thompson S. 44
- Konversion in ganze Zahlen:
  - `fromInt :: Int-> Double`
  - `fromInteger :: Integer-> Double`
  - `round, ceiling, floor:: Double-> Int, Integer`
  - Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur

Funktionaler Entwurf und Entwicklung

- Spezifikation:
  - Definitionsbereich (Eingabewerte)
  - Wertebereich (Ausgabewerte)
  - Vor/Nachbedingungen?

  $\rightsquigarrow$ Signatur
- Programmentwurf:
  - Gibt es ein ähnliches (gelöstes) Problem?
  - Wie kann das Problem in Teilprobleme zerlegt werden?
  - Wie können Teillösungen zusammengesetzt werden?

  $\rightsquigarrow$ Erster Entwurf

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Mögliche Verallgemeinerungen?
  - Gibt es hilfreiche Büchereifunktionen?
  - Wie würde man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung

- Implementierung:
  - Termination?
  - Effizienz? Geht es besser? Mögliche Verallgemeinerungen?
  - Gibt es hilfreiche Büchereifunktionen?
  - Wie würde man die Korrektheitheit zeigen?

  $\rightsquigarrow$ Lauffähige Implementierung
- Test:
  - Black-box Test: Testdaten aus der Spezifikation
  - White-box Test: Testdaten aus der Implementierung
  - Testdaten: hohe Abdeckung, Randfälle beachten.

Ein Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.

Ein Beispiel: das Nim-Spiel

- Zwei Spieler nehmen abwechselnd 1–3 Hölzchen.
- Verloren hat derjenige, der das letzte Hölzchen nimmt.
- Ziel: Programm, das entscheidet, ob ein Zug gewinnt.
- Eingabe: Anzahl Hölzchen gesamt, Zug
- Zug = Anzahl genommener Hölzchen
- Ausgabe: Gewonnen, ja oder nein.

```
type Move= Int
winningMove :: Int-> Move-> Bool
```

Programmentwurf

- Verfeinerung: Überprüfung, ob Zug legal

Programmentwurf

- Verfeinerung: Überprüfung, ob Zug legal

```
legalMove :: Int-> Int-> Bool
legalMove total m =
    (m<= total) && (1<= m) && (m<= 3)
```

- Implementation: gewonnen, wenn
  - Gegner verloren (nur noch ein Hölzchen über)
  - Gegner kann nur Züge machen, bei denen es eine Antwort gibt, mit denen Spieler gewinnt

```
winningMove total move =
  legalMove total move &&
  mustLose (total-move)
```

```
mustLose :: Int-> Bool
mustLose n
  | n == 1     = True
  | otherwise = canWin n 1 && canWin n 2 &&
                canWin n 3

canWin :: Int-> Int-> Bool
canWin total move =
  winningMove (total- move) 1 ||
  winningMove (total- move) 2 ||
  winningMove (total- move) 3
```

- Analyse:
  - Effizienz: unnötige Überprüfung bei `canWin`
  - Testfälle: Gewinn, Verlust, Randfälle
- Korrektheit:
  - Vermutung: Mit $4n + 1$ Hölzchen verloren, ansonsten gewonnen.
  - Beweis durch Induktion $\rightsquigarrow$ später.

Strukturierte Datentypen: Tupel und Listen

- Tupel sind das kartesische Produkt:
  `(t1, t2)` = alle möglichen Kombinationen von Werten aus `t1` und `t2`.
- Listen sind Sequenzen (freier Monoid):
  `[t]` = endliche Folgen von Werten aus `t`
  - Sequenz ist wie eine Menge, aber mit einer Ordnung, ohne doppelte Elemente, und endlich.
- Strukturierte Typen: konstruieren aus anderen Typen neue Typen.

- Beispiel: Modellierung eines Einkaufswagens
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

- Beispiel: Modellierung eines Einkaufswagens
  - Inhalt: Menge von Dingen mit Namen und Preis

```
type Item   = (String, Int)
type Basket = [Item]
```

- Beispiel: Punkte, Rechtecke, Polygone

```
type Point   = (Int, Int)
type Line    = (Point, Point)
type Polygon = [Point]
```

Funktionen über Listen und Tupeln

- Funktionsdefinition durch pattern matching:

```
add :: Point-> Point-> Point
add (a, b) (c, d) = (a+ c, b+ d)
```

- Für Listen:
  - entweder leer
  - oder bestehend aus einem Kopf und einem Rest

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

  - Hier hat `x` den Typ `Int`, `xs` den Typ `[Int]`.

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

- Gesamtpreis des Einkaufs:

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

- Translation eines Polygons:

```
move :: Polygon-> Point-> Polygon
move [] p                       = []
move ((x, y):ps) (px, py) = (x+ px, y+ py):
                                (move ps (px, py))
```

Einzelne Zeichen: `Char`

- Notation für einzelne Zeichen: `'a'`,. . .
  - NB. Kein Unicode.
- Nützliche Funktionen:

```
ord :: Char -> Int
chr :: Int -> Char

toLower :: Char-> Char
toUpper :: Char-> Char
isDigit :: Char-> Bool
isAlpha :: Char-> Bool
```

Zeichenketten: `String`

- `String` sind Sequenzen von Zeichenketten:

```
type String = [Char]
```

- Alle vordefinierten Funktionen auf Listen verfügbar.
- Syntaktischer Zucker zur Eingabe:

```
['y','o','h','o'] == "yoho"
```

- Beispiel:

```
count :: Char-> String-> Int
count c []     = 0
count c (x:xs) = if (c== x) then 1+ count c xs
                 else count c xs
```

Beispiel: Palindrome

- Palindrom: vorwärts und rückwärts gelesen gleich (z.B. Otto, Reliefpfeiler)

Beispiel: Palindrome

- Palindrom: vorwärts und rückwärts gelesen gleich (z.B. Otto, Reliefpfeiler)
- Signatur:

```
palindrom: String-> Bool
```

Beispiel: Palindrome

- Palindrom: vorwärts und rückwärts gelesen gleich (z.B. Otto, Reliefpfeiler)
- Signatur:

  `palindrom: String-> Bool`

- Entwurf:
  - Rekursive Formulierung:
    erster Buchstabe = letzer Buchstabe, und Rest auch Palindrom
  - Leeres Wort und monoliterales Wort sind Palindrome
  - Hilfsfunktionen:
    `last: String-> Char`, `init: String-> String`

- Implementierung:

```
palindrom :: String-> Bool
palindrom []     = True
palindrom [x]    = True
palindrom (x:xs) = (x == last xs)
                   && palindrom (init xs)
```

- Implementierung:

```
palindrom :: String-> Bool
palindrom []     = True
palindrom [x]    = True
palindrom (x:xs) = (x == last xs)
                   && palindrom (init xs)
```

- Kritik:
  - Unterschied zwischen Groß- und kleinschreibung

```
palindrom (x:xs) = (toLower x == toLower (last xs))
                   && palindrom (init xs)
```

  - Nichtbuchstaben sollten nicht berücksichtigt werden.

Zusammenfassung

- Funktionsdefinitionen:
  - Abseitsregel, bedingte Definition, *pattern matching*
- Numerische Basisdatentypen:
  - `Int`, `Integer`, `Rational` und `Double`
- Funktionaler Entwurf und Entwicklung
  - Spezifikation der Ein- und Ausgabe $\rightsquigarrow$ Signatur
  - Problem rekursiv formulieren $\rightsquigarrow$ Implementation
  - Test und Korrektheit
- Strukturierte Datentypen: Tupel und Listen
- Alphanumerische Basisdatentypen: `Char` und `String`
  - `type String = [Char]`

Vorlesung vom 05.11.2001

Inhalt

- Letzte Vorlesung
  - Basisdatentypen, strukturierte Typen Tupel und Listen
  - Definition von Funktionen durch rekursive Gleichungen
- Diese Vorlesung: Formen der Rekursion
  - Listenkomprehension
  - primitive Rekursion
  - nicht-primitive Rekursion
- Neue Sprachkonzepte:
  - Polymorphie — Erweiterung des Typkonzeptes
  - Lokale Definitionen
- Vordefinierte Funktionen auf Listen

Listenkomprehension

- Ein Schema für Funktionen auf Listen:
  - Eingabe generiert Elemente,
  - die getestet und
  - zu einem Ergebns transformiert werden
- Beispiel Palindrom:
  - alle Buchstaben im String `str` zu Kleinbuchstaben.
    labelitemiidef:toLower

    ```
    [ toLower c | c <- str ]
    ```
  - Alle Buchstaben aus `str` herausfiltern:

    ```
    [ c | c <- str, isAlpha c ]
    ```
  - Beides zusammen:

```
[ toLower c | c<- str, isAlpha c]
```

- Generelle Form:

```
[ E | c<- L, test1, ... , testn ]
```

- Mit pattern matching:

```
addPair :: [(Int, Int)]-> [Int]
addPair ls = [ x+ y | (x, y) <- ls ]
```

- Auch mehrere Generatoren möglich.

Ein Beispiel: Eine Bücherei

- Problem: Modellierung einer Bücherei
- Datentypen:
  - Ausleihende Personen
  - Bücher
  - Zustand der Bücherei: ausgeliehene Bücher, Ausleiher

```
type Person   = String
type Book     = String
type DBase = [(Person, Book)]
```

- Suchfunktionen: Wer hat welche Bücher ausgeliehen usw.

```
books :: DBase -> Person -> [Book]
books db who = [ book | (pers,book)<- db,
                        pers== who ]
```

- Buch ausleihen und zurückgeben:

```
makeLoan :: DBase -> Person -> Book -> DBase
makeLoan dBase pers bk = [(pers,bk)] ++ dBase
```

  - Benutzt (++) zur Verkettung von Listen

```
returnLoan :: DBase -> Person -> Book -> DBase
returnLoan dBase pers bk
        = [ pair | pair <- dBase ,
                   pair /= (pers,bk) ]
```

Polymorphie — jetzt oder nie.

- Definition von `(++)`:

```
(++) :: DBase->  DBase-> DBase
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Verketten von Strings:

Polymorphie — jetzt oder nie.

- Definition von `(++)`:

```
(++) :: DBase->  DBase-> DBase
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Verketten von Strings:

```
(++) :: String-> String-> String
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

- Gleiche Definition, aber unterschiedlicher Typ!

  $\Longrightarrow$ Zwei Instanzen einer allgemeineren Definition.

- Polymorphie erlaubt Parametrisierung über Typen:

```
(++) :: [a]-> [a]-> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)
```

  a ist hier eine Typvariable.

- Definition wird bei Anwendung instantiiert:

```
[3,5,57] ++ [39, 18]      "hi" ++ "ho"
```

  aber nicht

```
[True, False] ++ [18, 45]
```

- Typvariable: vergleichbar mit Funktionsparameter

- Weitere Beispiele:
  - Länge einer Liste:

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

  - Verschachtelte Listen "flachklopfen":

```
concat :: [[a]]-> [a]
concat []     = []
concat (x:xs) = x ++ (concat xs)
```

  - Kopf und Rest einer nicht-leeren Liste:

```
head :: [a]-> a
head (x:xs) = x
```

```
tail :: [a]-> [a]
tail (x:xs) = xs
```

Undefiniert für leere Liste.

Übersicht: vordefinierte Funktionen auf Listen

`:`	`a-> [a]-> [a]`	Element vorne anfügen
`++`	`[a]-> [a]-> [a]`	Verketten
`!!`	`[a]-> Int-> a`	`n`-tes Element selektieren
`concat`	`[[a]]-> [a]`	"flachklopfen"
`length`	`[a]-> Int`	Länge
`head`, `last`	`[a]-> a`	Erster/letztes Element
`tail`, `init`	`[a]-> [a]`	Rest (hinterer/vorderer)
`replicate`	`Int-> a-> [a]`	Erzeuge `n` Kopien
`take`	`Int-> [a]-> [a]`	Nimmt ersten `n` Elemente
`drop`	`Int-> [a]-> [a]`	Entfernt erste `n` Elemente
`splitAt`	`Int-> [a]-> ([a], [a])`	Spaltet an `n`-ter Position

`reverse`	`[a]-> [a]`	Dreht Liste um
`zip`	`[a]-> [b]-> [(a, b)]`	Macht aus Paar von Listen Liste von Paaren
`unzip`	`[(a, b)]-> ([a], [b])`	Macht aus Liste von Paaren Paar von Listen
`and`, `or`	`[Bool]-> Bool`	Konjunktion/Disjunktion
`sum`	`[Int]-> Int` (überladen)	Summe
`product`	`[Int]-> Int` (überladen)	Produkt

Siehe Thompson S. 91/92.

Palindrom zum letzten:

```
palindrom xs = (reverse l) == l where
             l = [toLower c| c<- xs, isAlpha c]
```

Ein Beispiel: das n-Königinnen-Problem

- Problem: n Königinnen auf $n \times n$-Schachbrett sicher plazieren
- Spezifikation:
  - Position der Königinnen

    ```
    type Pos = (Int, Int)
    ```

  - Eingabe: Anzahl Königinnen, Rückgabe: Positionen

    ```
    queens :: Int-> [[Pos]]
    ```

- Rekursive Formulierung:
  - Keine Königin— kein Problem.
  - Lösung für n Königinnen: Lösung für $n-1$ Königinnen, und n-te Königin so stellen, dass keine andere sie bedroht.
  - n-te Königin muß in n-ter Spalte plaziert werden.

```
queens num = qu num where
  qu :: Int-> [[Pos]]
  qu n | n == 0  = [[]]
       | otherwise =
     [ p++ [(n, m)] | p <- qu (n-1), m <- [1.. num],
                      safe p (n, m)]
safe :: [Pos]-> Pos-> Bool
safe others nu =
   and [ not (threatens other nu) | other <- others ]
threatens :: Pos-> Pos-> Bool
threatens (i, j) (m, n) =
   (j== n) || (i+j == m+n) || (i-j == m-n)
```

- Test `i==m` unnötig.
- `[n..m]`: Liste der Zahlen von `n` bis `m`

Lokale Definitionen

- Lokale Definitionen mit `where` — Syntax:

```
f x y
  | g1 = P
  | g2 = Q where
      v1 = M
      v2 x = N x
```

- `v1`, `v2`, . . . werden gleichzeitig definiert (wichtig bei Rekursion);
- Namen `v1` und Parameter (`x`) überlagern andere Bezeichner;
- Es gilt die Abseitsregel (deshalb auf gleiche Einrückung der lokalen Definition achten);

Muster *(pattern)*

- Funktionsparameter sind Muster: `head (x:xs) = x`
- Muster sind:
  - Wert (`0` oder `True`)
  - Variable (`x`) - dann paßt alles
    - ▷ Jede Variable darf links nur einmal auftreten.
  - namenloses Muster (`_`) - dann paßt alles.
    - ▷ `_` darf links mehrfach, rechts gar nicht auftreten.
  - Tupel `(p1, p2, ... pn)` (`pi` sind wieder Muster)
  - leere Liste `[]`
  - nicht-leere Liste `ph:pl` (`ph, pl` sind wieder Muster)
  - `[p1,p2,...pn]` ist syntaktischer Zucker für `p1:p2:...pn:[]`

Primitive Rekursion auf Listen

- Primitive Rekursion vs. allgemeine Rekursion
- Primitive Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste
- Beispiel:

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

  Auswertung von `sum [4,7,3]`: `4 + 7 + 3 + 0`
- Weitere Beispiele: `length`, `concat`, `(++)`, ...

- Allgemeines Muster:

$$\texttt{f}[\texttt{x1}, ..., \texttt{xn}] = \texttt{x1} \otimes \texttt{x1} \otimes \ldots \otimes \texttt{xn} \otimes A$$

  - Startwert (für die leere Liste) `A :: b`
  - Rekursionsfunktion $\otimes$ `:: a -> b-> b`

Nicht-primitive Rekursion

- Rekursion über mehrere Argumente:

```
zip :: [a]-> [b]-> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

- Rekursion über ganzen Zahlen:

```
take :: Int-> [a]-> [a]
take 0 _  = []
take _ [] = []
take n (x:xs)
  | n > 0     = x: take (n-1) xs
  | otherwise = error "take: negative Argument"
```

- Andere Zerlegungen:
  - Quicksort:

    zerlege Liste in Elemente kleiner gleich und größer dem ersten,

    sortiere Teilstücke, konkateniere Ergebnisse

```
qsort :: [Int]-> [Int]
qsort [] = []
qsort (x:xs) = qsort [ y | y <- xs, y<= x ] ++ [x] ++
               qsort [ y | y <- xs, y > x ]
```

- Mergesort:

  teile Liste in der Hälfte,

  sortiere Teilstücke, füge ordnungserhaltend zusammen.

```
msort :: [Int]-> [Int]
msort xs
  | length xs <= 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
    merge :: [Int]-> [Int]-> [Int]
    merge [] x = x
    merge y [] = y
    merge (x:xs) (y:ys)
      | x<= y     = x:(merge xs (y:ys))
      | otherwise = y:(merge (x:xs) ys)
```

Zusammenfassung

- Schemata für Funktionen über Listen:
  - Listenkomprehension
  - primitive und nicht-rekursive Funktionen
- Polymorphie :
  - Abstraktion über Typen durch Typvariablen
- Lokale Definitionen mit `where`
- Überblick: vordefinierte Funktionen auf Listen

Vorlesung vom 12.11.2001

Inhalt

- Formalisierung und Beweis
  - Vollständige, strukturelle und Fixpunktinduktion
- Verifikation
  - Tut mein Programm, was es soll?
- Fallbeispiel: Verifikation von Quicksort
- Funktionen höherer Ordnung
  - Berechnungsmuster (*patterns of computation*)
  - `map` und `filter`: Verallgemeinerte Listenkomprehension
  - `fold`: Primitive Rekursion

Rekursive Definition, induktiver Beweis

- Definition ist rekursiv
  - Basisfall (leere Liste)
  - Rekursion (`x:xs`)

```
count :: Int-> [Int]-> Int
count x []    = 0
count x (y:ys)
  | x == y     = count x ys + 1
  | otherwise = count x ys
```

  - Reduktion der Eingabe (vom größeren aufs kleinere)
- Beweis durch Induktion
  - Schluß vom kleineren aufs größere

Beweis durch vollständige Induktion

Zu zeigen:

Für alle natürlichen Zahlen x gilt $P(x)$

Beweis:

- Induktionsbasis:

$$P(0)$$

- Induktionssschritt:

$$P(x) \Rightarrow P(x+1)$$

Beweis durch strukturelle Induktion

Zu zeigen:

$$\text{Für alle Listen } xs \text{ gilt } P(xs)$$

Beweis:

- Induktionssbasis (leere Liste):

$$P([])$$

- Induktionssschritt (cons):

$$P(xs) \Rightarrow P(x{:}xs)$$

Beispiel

Lemma: (**count**-++)

```
count x (ys++ zs)= count x ys + count x zs
```

Beweis: Induktion über `ys`

- Induktionssbasis:

```
    count x ([] ++ zs)
=   count x zs                  Definition von ++
=   0 + count x zs
=   count x []+ count x zs      Definition von count
```

- Induktionsschritt: Annahme $P(ys)$, zu zeigen $P(y : ys)$

  1. $y = x$:

```
      count x ((y:ys) ++ zs)
=     count x (y:(ys++zs))
          Nach Def. ++
=     1+ count x (ys++ zs)
          Nach Def. count und Annahme x = y
=     1+ count x ys + count x zs
          Induktionsannahme
=     count x (y:ys) + count x zs
          Nach Def. count und Annahme x = y
```

  2. $y \neq x$: Analog.

Korrektheitsbeweis Quicksort

- Zu zeigen: `qsort` korrekt.

Korrektheitsbeweis Quicksort

- Zu zeigen: `qsort` korrekt.
  - `qsort xs` ist sortiert
  - `qsort xs` enthält dieselben Elemente wie `xs`.
- Sortiertheit:

```
sorted :: [Int]-> Bool
sorted []  = True
sorted [x] = True
sorted (x:y:ys) = x <= y && sorted (y:ys)
```

- Dieselben Elemente: Permutation

$$\texttt{perm xs ys} \Leftrightarrow \forall \texttt{x.count x xs} = \texttt{count x ys}$$

Fixpunktinduktion

Gegeben: rekursive Definition

$$fx = E \quad E \text{ enthält rekursiven Aufruf } ft$$

Zu zeigen: Für alle Listen x gilt $P(fx)$

Beweis:

- Annahme: Es gilt $P(ft)$
- Zu zeigen: $P(E)$
  - Können annehmen, Ergebnis des rekursiven Aufrufs erfüllt P
  - Müssen zeigen, Gesamtergebnis erfüllt P
  - Mit anderen Worten: ein Rekursionsschritt erhält P

Korrektheitsbeweis Quicksort

Zu zeigen:

```
perm xs (qsort xs) && (sorted (qsort xs))
```

Beweis mit Fixpunktinduktion.

- Erste Gleichung:

```
perm [] (qsort []) && (sorted (qsort []))
```

- Zweite Gleichung:

  Abkürzung: seien `l1` $\stackrel{def}{=}$ `[y| y<- xs, y< x]`,
  `l2` $\stackrel{def}{=}$ `[y| y<- xs, x<= y]`

  Induktionsschritt:

```
perm l1 (qsort l1),sorted (qsort l1),
perm l2 (qsort l2),sorted (qsort l2)
⇒ perm xs (qsort l1++ [x]++ qsort l2)&&
   sorted (qsort l1++ [x]++ qsort l2)
```

Berechnungsmuster

- Listenkomprehension I: Funktion auf alle Elemente anwenden
  - `toLower`, `move`, ...
- Listenkomprehension II: Elemente herausfiltern
  - `books`, `returnLoan`, ...
- Primitive Rekursion
  - `++`, `length`, `concat`, ...
- Listen zerlegen
  - `take`, `drop`
- Sonstige
  - `qsort`, `msort`

Funktionen Höherer Ordnung

- Grundprinzip der funktionalen Programmierung
- Funktionen sind gleichberechtigt
  - d.h. Werte wie alle anderen
- Funktionen als Argumente: allgemeinere Berechungsmuster
- Höhere Wiederverwendbarkeit
- Größere Abstraktion

Funktionen als Argumente

- Funktion auf alle Elemente anwenden: `map`
- Signatur:

Funktionen als Argumente

- Funktion auf alle Elemente anwenden: `map`
- Signatur:

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

- Definition

```
map f xs = [ f x | x <- xs ]
```

- oder -

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Funktionen als Argumente

- Elemente filtern: `filter`
- Signatur:

Funktionen als Argumente

- Elemente filtern: `filter`
- Signatur:

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

- Definition

```
filter p xs = [ x | x <- xs, p x ]
```

- oder -

```
filter p []    = []
filter p (x:xs)
  | p x        = x:(filter p xs)
  | otherwise = filter p xs
```

Primitive Rekursion

- Primitive Rekursion:
  - Basisfall (leere Liste)
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Beispiel: `sum :: [Int]-> Int`

  ```
  sum [13, 2] = 13 + (sum [2]) = 13 + 2 + sum [] = 13 + 2 + 0
  ```
- Definition

  ```
  foldr f e []     = e
  foldr f e (x:xs) = f x (foldr f e xs)
  ```
- Signatur

  ```
  foldr :: (a-> b-> b)-> b-> [a]-> b
  ```
- Beispiel: Sortieren durch Einfügen

```
isort :: [Int]-> [Int]
isort xs = foldr ins [] xs
```

Hierbei: `ins` ordnungserhaltendes Einfügen

Listen zerlegen

- `take, drop`: n Elemente vom Anfang
- Längster Präfix für den Prädikat gilt

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

- Restliste des längsten Präfix

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

- Es gilt: `takeWhile p xs ++ dropWhile p xs == xs`
- Kombination der beiden

```
span      :: (a -> Bool) -> [a] -> ([a],[a])
```

```
span p xs = (takeWhile p xs, dropWhile p xs)
```

- Ordnungserhaltendes Einfügen:

```
ins :: Int-> [Int]-> [Int]
ins x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span less xs
  less z = z < x
```

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?
- Ordnung als Argument `ord`
  - Totale Ordnung: transitiv, antisymmetrisch, reflexiv, total
  - Insbesondere: x `ord` $y \wedge y$ `ord` $x \Rightarrow x = y$

```
qsortBy :: (a-> a-> Bool)-> [a]-> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
     qsortBy ord [y| y<-xs, ord y x] ++ [x] ++
     qsortBy ord [y| y<-xs, not (ord y x)]
```

  - NB: Code aus VL hatte einen Fehler – oben korrigierte Version.

Zusammenfassung

- Verifikation und Beweis
  - Beweis durch strukturelle und Fixpunktinduktion
  - Verifikation eines nichttrivialen Algorithmus (Quicksort)
- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Werte
  - Erlaubt Verallgemeinerungen
  - Erhöht Flexibilität und Wiederverwendbarkeit
  - Beispiele: `map`, `filter`, `foldr`
  - Sortieren nach beliebiger Ordnung

Vorlesung vom 19.11.2001

Inhalt

- Funktionen höherer Ordnung
  - Letzte VL: verallgemeinerte Berechnungsmuster (`map`, `filter`, `foldr`, . . .)
  - Heute: Konstruktion neuer Funktionen aus alten
- Nützliche Techniken:
  - Anonyme Funktionen
  - Partielle Applikation
  - η-Kontraktion
- Längeres Beispiel: Erstellung eines Index
- Typklassen: Überladen von Funktionen

Funktionen als Werte

- Zusammensetzen neuer Funktionen aus alten.
- Zweimal hintereinander anwenden:

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

Funktionen als Werte

- Zusammensetzen neuer Funktionen aus alten.
- Zweimal hintereinander anwenden:

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

- n-mal hintereinander anwenden:

```
iter :: Int-> (a-> a)-> a-> a
iter 0 f x = x
iter n f x | n > 0 = f (iter (n-1) f x)
           | otherwise = x
```

- Funktionskomposition:

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

  - `f` nach `g`.

- Funktionskomposition vorwärts:

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

  - Nicht vordefiniert!

- Identität:

```
id :: a-> a
id x = x
```

  - Nützlicher als man denkt.

Anonyme Funktionen und die λ-Notation

- Nicht jede Funktion muß einen Namen haben.
- Beispiel:

```
ins x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span less xs
  less z = z < x
```

- Besser: statt `less` anonyome Funktion

```
ins' x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span (\z-> z < x) xs
```

  - `\x-> E` = `f where f x= E`
  - Auch pattern matching möglich

Beispiel: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl `p` alle Vielfachen heraussieben

Beispiel: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl `p` alle Vielfachen heraussieben
  - Dazu: filtern mit `(\n-> n `mod` p /= 0)`

```
sieve :: [Integer]-> [Integer]
sieve []     = []
sieve (p:ps) =
  p:(sieve (filter (\n-> n `mod` p /= 0) ps))
```

- Primzahlen im Intervall `[1..n]`

```
primes :: Integer-> [Integer]
primes n = sieve [2..n]
```

η-Kontraktion

- Nützliche vordefinierte Funktionen:
  - Disjunktion/Konjunktion von Prädikaten über Listen

```
all, any :: (a-> Bool)-> [a]-> Bool
any p      = or  . map p
all p      = and . map p
```

- Da fehlt doch was?!

η-Kontraktion

- Nützliche vordefinierte Funktionen:
  - Disjunktion/Konjunktion von Prädikaten über Listen

```
all, any :: (a-> Bool)-> [a]-> Bool
any p      = or  . map p
all p      = and . map p
```

- Da fehlt doch was?!
- η-Kontraktion:
  - Allgemein: `\x-> E x` $\Leftrightarrow$ `E`
  - Bei Funktionsdefinition: `f x = E x` $\Leftrightarrow$ `f = E`
  - Hier: Definition äquivalent zu `any p x = or (map p x)`

Partielle Applikation

- Funktionen können partiell angewandt werden:

```
double :: String-> String
double = concat . map (replicate 2)
```

  - Zur Erinnerung: `replicate :: Int-> a-> [a]`

Die Kürzungsregel bei Funktionsapplikation

Bei Anwendung der Funktion

$$\texttt{f :: t}_1 \texttt{ -> t}_2 \texttt{ -> ... -> t}_n\texttt{-> t}$$

auf k Argumente mit $k \leq n$

$$\texttt{e}_1 \texttt{ :: t}_1\texttt{, e}_2 \texttt{ :: t}_2\texttt{, ..., e}_k \texttt{ :: t}_k$$

werden die Typen der Argumente gekürzt:

$$\texttt{f :: }\not{\texttt{t}}_1 \texttt{ -> }\not{\texttt{t}}_2 \texttt{ -> ... -> }\not{\texttt{t}}_k \texttt{ -> t}_{k+1} \texttt{ -> ...-> t}_n \texttt{ -> t}$$
$$\texttt{f e}_1 \texttt{ ...e}_k \texttt{ :: t}_{k+1} \texttt{ -> ...-> t}_n \texttt{ -> t}$$

- Partielle Anwendung von Operatoren:

```
elem :: Int-> [Int]-> Bool
elem x = any (== x)
```

  - `(== x)` Sektion des Operators `==` (entspricht `\e-> e == x`)

Gewürzte Tupel: Curry

- Unterschied zwischen

  `f :: a-> b-> c` und `f :: (a, b)-> c` ?

  - Links partielle Anwendung möglich.
  - Ansonsten äquivalent.
- Konversion:
  - Rechts nach links:

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

  - Links nach rechts:

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

Noch ein Beispiel: Der Index

- Problem:
  - Gegeben ein Text

    ```
    brösel fasel\nbrösel brösel\nfasel brösel blubb
    ```

  - Zu erstellen ein Index: für jedes Wort Liste der Zeilen, in der es auftritt

    ```
    brösel [1, 2, 3]      blubb [3]      fasel [1, 3]
    ```

- Spezifikation der Lösung

```
type Doc = String
type Word= String
makeIndex :: Doc-> [([Int], Word)]
```

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp

  1. Text in Zeilen aufspalten: — `[Line]`
     (mit `type Line= String`)

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp
  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp
  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  3. Zeilen in Worte spalten (Zeilennummer beibehalten):
     `[(Int, Word)]`

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp
  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  3. Zeilen in Worte spalten (Zeilennummer beibehalten):
     `[(Int, Word)]`
  4. Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp

  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  3. Zeilen in Worte spalten (Zeilennummer beibehalten):
     `[(Int, Word)]`
  4. Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`
  5. Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
     `[([Int], Word)]`

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp

  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  3. Zeilen in Worte spalten (Zeilennummer beibehalten):
     `[(Int, Word)]`
  4. Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`
  5. Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
     `[([Int], Word)]`
  6. Alle Worte mit weniger als vier Buchstaben entfernen:
     `[([Int], Word)]`

- Zerlegung des Problems in einzelne Schritte — Ergebnistyp
  1. Text in Zeilen aufspalten: `[Line]`
     (mit `type Line= String`)
  2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  3. Zeilen in Worte spalten (Zeilennummer beibehalten):
     `[(Int, Word)]`
  4. Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`
  5. Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
     `[([Int], Word)]`
  6. Alle Worte mit weniger als vier Buchstaben entfernen:
     `[([Int], Word)]`

- Erste Implementierung:

```
type Line = String
makeIndex =
  lines       >.> -- Doc -> [Line]
  numLines    >.> --     -> [(Int,Line)]
  allNumWords >.> --     -> [(Int,Word)]
  sortLs      >.> --     -> [(Int,Word)]
  makeLists   >.> --     -> [([Int],Word)]
  amalgamate  >.> --     -> [([Int],Word)]
  shorten         --     -> [([Int],Word)]
```

- Implementierung der einzelnen Komponenten:
  - In Zeilen zerlegen:

    `lines :: String-> [String]` aus dem Prelude
  - Jede Zeile mit ihrer Nummer versehen:

```
numLines :: [Line]-> [(Int, Line)]
numLines lines = zip [1.. length lines] lines
```

  - Jede Zeile in Worte zerlegen:

    Pro Zeile: `words:: String-> [String]` aus dem Prelude.
    - ▷ Berücksichtigt nur Leerzeichen.
    - ▷ Vorher alle Satzzeichen durch Leerzeichen ersetzen.

```
splitWords :: Line-> [Word]
splitWords = words . map (\c-> if isPunct c then ' '
                               else c) where
   isPunct :: Char-> Bool
   isPunct c = c `elem` ";:.,\'\"!?(){}-\\[]"
```

Auf alle Zeilen anwenden, Ergebnisliste flachklopfen.

```
allNumWords :: [(Int, Line)]-> [(Int, Word)]
allNumWords = concat . map oneLine where
   oneLine :: (Int, Line)-> [(Int, Word)]
   oneLine (num, line) = map (\w-> (num, w))
                             (splitWords line)
```

- Liste alphabetisch nach Worten sortieren:

  Ordnungsrelation definieren:

```
ordWord :: (Int, Word)-> (Int, Word)-> Bool
ordWord (n1, w1) (n2, w2) =
  w1 < w2 || (w1 == w2 && n1 <= n2)
```

  Generische Sortierfunktion `qsortBy`

```
sortLs :: [(Int, Word)]-> [(Int, Word)]
sortLs = qsortBy ordWord
```

- Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:

  Erster Schritt: Jede Zeile zu (einelementiger) Liste von Zeilen.

```
makeLists :: [(Int, Word)]-> [([Int], Word)]
makeLists = map (\ (l, w)-> ([l], w))
```

Zweiter Schritt: Gleiche Worte zusammenfassen.

▷ Nach Sortierung sind gleiche Worte hintereinander!

```
amalgamate :: [([Int], Word)]-> [([Int],Word)]
amalgamate []  = []
amalgamate [p] = [p]
amalgamate ((l1, w1):(l2, w2):rest)
  | w1 == w2  = amalgamate ((l1++ l2, w1):rest)
  | otherwise = (l1, w1):amalgamate ((l2, w2):rest)
```

- Alle Worte mit weniger als vier Buchstaben entfernen:

```
shorten :: [([Int],Word)] -> [([Int],Word)]
shorten = filter (\ (_, wd)-> length wd >= 4)
```

Alternative Definition:

```
shorten = filter ((>= 4) . length . snd)
```

Typklassen

- Allgemeinerer Typ für `elem`:

  ```
  elem :: a-> [a]-> Bool
  ```

  reicht nicht wegen `c ==`
  - `(==)` kann nicht für alle Typen definiert werden:
  - z.B. `(==) :: (Int-> Int)-> (Int-> Int)-> Bool` ist nicht entscheidbar.
- Lösung: `Typklassen`

  ```
  elem :: Eq a=> a-> [a]-> Bool
  elem c = any (c ==)
  ```

  - Für `a` kann jeder Typ eingesetzt werden, für den `(==)` definiert ist.
  - `Eq a` ist eine Klasseneinschränkung (*class constraint*)

- Standard-Typklassen:
  - `Eq a` für `== :: a-> a-> Bool` (Gleichheit)
  - `Ord a` für `<= :: a-> a-> Bool` (Ordnung)
    - ▷ Alle Basisdatentypen
    - ▷ Listen, Tupel
    - ▷ Nicht für Funktionsräume
  - `Show a` für `show :: a-> String`
    - ▷ Alle Basisdatentypen
    - ▷ Listen, Tupel
    - ▷ Nicht für Funktionsräume
  - `Read a` für `read :: String-> a`
    - ▷ Siehe `Show`
- Typklassen erlauben des Überladen von Funktionsnamen (`(==)` etc.)

Zusammenfassung

- Funktionen als Werte
- Anonyme Funktionen: `\x-> E`
- η-Kontraktion: `f x = E x` $\Rightarrow$ `f` $=$ `E`
- Partielle Applikation und Kürzungsregel
- Indexbeispiel:
  - Dekomposition in Teilfunktionen
  - Gesamtlösung durch pipelining der Teillösungen
- Typklassen erlauben überladene Funktionen.

Vorlesung vom 26.11.2001

Inhalt

- Letzte VL: abstrakte Programme durch Funktionen höherer Ordnung.
- Heute: Datenabstraktion durch algebraische Datentypen.
- Einfache Datentypen: Aufzählungen und Produkte
- Der allgemeine Fall
- Bekannte Datentypen: `Maybe`, Bäume
- Geordnete Bäume und ausgeglichene Bäume

Was ist Datenabstraktion?

- Typsynonyme sind keine Datenabstraktion
  - `type Dayname = Int` wird textuell expandiert.
  - Keinerlei Typsicherheit:
    - `Freitag + 15` ist kein Wochentag;
    - `Freitag * Montag` ist kein Typfehler.
  - Kodierung `0 = Montag` ist willkürlich und nicht eindeutig.
- Deshalb:
  - Wochentage sind nur Montag, Dienstag, . . . , Sonntag.
  - Alle Wochentage sind unterschiedlich.
- $\Longrightarrow$ Einfachster algebraischer Datentyp: Aufzählungstyp.

Aufzählungen

- Bsp: Wochentage

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

- Konstruktoren:

```
Mo :: Weekday, Tu :: Weekday, We :: Weekday, ...
```

  - Konstruktoren werden nicht deklariert.

- Funktionsdefinition durch pattern matching:

```
isWeekend :: Weekday -> Bool
isWeekend Sa = True
isWeekend Su = True
isWeekend _  = False
```

Produkte

- Bsp: Datum besteht aus Tag, Monat, Jahr:

```
data Date  = Date Day Month Year
type Day   = Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
type Year  = Int
```

- Beispielwerte:

```
today     = Date 26 Nov 2001
bloomsday = Date 16 Jun 1904
fstday    = Date 1 Jan 1
```

- Konstruktor:

```
Date :: Day-> Month-> Year-> Date
```

- Funktionsdefinition:
  - Über pattern matching Zugriff auf Argumente der Konstruktoren:

```
day  :: Date-> Day
year :: Date-> Year
day  (Date d m y) = d
year (Date d m y) = y
```

  - `day`, `year` sind Selektoren:

```
day  today     = 26
year bloomsday = 1904
```

Alternativen

- Bsp: Eine geometrische Figur ist
  - eine Kreis, gegeben durch Mittelpunkt und Durchmesser,
  - oder ein Rechteck, gegeben durch zwei Eckpunkte,
  - oder ein Polygon, gegeben durch Liste von Eckpunkten.

```
type Point = (Int, Int)
data Shape = Circ Point Int
           | Rect Point Point
           | Poly [Point]
```

- Ein Konstruktor pro Variante.

```
Circ :: Point-> Int-> Shape
```

- Funktionsdefinition durch pattern matching:

```
corners :: Shape-> Int
corners (Circ _ _) = 0
corners (Rect _ _) = 4
corners (Poly ps)  = length ps

move :: Shape-> Point-> Shape
move (Circ m d) p   = Circ (add p m) d
move (Rect c1 c2) p = Rect (add p c1) (add p c2)
move (Poly ps) p    = Poly (map (add p) ps)
add :: Point-> Point-> Point
add (x, y) (u, v) = (x+ u, y+ v)
```

Das allgemeine Format

Definition von $\mathtt{T}$: $\mathtt{data\ T} =$

$$\begin{array}{rl} & \mathtt{C}_1\ \mathtt{t}_{1,1} \ldots \mathtt{t}_{1,k_1} \\ & \ldots \\ \mid & \mathtt{C}_n\ \mathtt{t}_{n,1} \ldots \mathtt{t}_{n,k_n} \end{array}$$

- Konstruktoren (und Typen) werden groß geschrieben.
- Konstruktoren $\mathtt{C}_1, \ldots, \mathtt{C}_n$ sind disjunkt:

$$\mathtt{C}_i\ \mathtt{x}_1 \ldots \mathtt{x}_n = \mathtt{C}_j\ \mathtt{y}_1 \ldots \mathtt{y}_m \Rightarrow i = j$$

- Konstruktoren sind injektiv:

$$\mathtt{C}\ \mathtt{x}_1 \ldots \mathtt{x}_n = \mathtt{C}\ \mathtt{y}_1 \ldots \mathtt{y}_n \Rightarrow \mathtt{x}_i = \mathtt{y}_i$$

- Konstruktoren erzeugen den Datentyp:

$$\forall x \in T.\, x = \mathtt{C}_i\ \mathtt{y}_1 \ldots \mathtt{y}_m$$

Abgeleitete Klasseninstanzen

- Wie würde man Gleichheit auf `Shape` definieren?

```
Circ p1 i1 == Circ p2 i2 = p1 == p2 && i1 == i2
Rect p1 q1 == Rect p2 q2 = p1 == p2 && q1 == q2
Poly ps    == Poly qs    = ps == qs
_          == _          = False
```

- Schematisierbar:
  - Gleiche Konstruktoren mit gleichen Argumente gleich,
  - alles andere ungleich.
- Automatisch generiert durch `deriving Eq`
- Ähnlich `deriving Ord, Show, Read`

Polymorphe Rekursive Datentypen

- Algebraische Datentypen parametrisiert über Typen:

```
data List a = Mt | Cons a (List a)
            deriving (Eq, Show)
```

- Eine Liste von `a` ist
  - entweder leer
  - oder ein Kopf `a` und eine Restliste `List a`

- Funktionsdefinition:

- Funktionsdefinition:

```
fold :: (a-> b-> b)-> b-> List a-> b
fold f e Mt           = e
fold f e (Cons a as) = f a (fold f e as)
```

- Mit `fold` alle primitiv rekursiven Funktionen, wie:

```
map' f    = fold (Cons . f) Mt
length'   = fold ((+).(const 1)) 0
filter' p = fold (\x-> if p x then Cons x
                      else id) Mt
```

  - Konstante Funktion `const:: a-> b-> a` aus dem Prelude.

Binäre Bäume

```
data Tree a = Null
            | Tree (Tree a) a (Tree a)
              deriving (Eq, Show)
```

- Test auf Enthaltensein:

```
member :: Eq a=> Tree a-> a-> Bool
member Null _ = False
member (Tree l a r) b =
  a == b || (member l b) || (member r b)
```

- Funktionen höherer Ordnung auf Bäumen:

```
foldT :: (a-> b-> b-> b)-> b-> Tree a-> b
foldT f e Null = e
foldT f e (Tree l a r) =
   f a (foldT f e l) (foldT f e r)
```

- Damit: Elementtest, `map`

```
member' t x =
   foldT (\e b1 b2-> e == x || b1 || b2) False t

mapT :: (a-> b)-> Tree a-> Tree b
mapT f = foldT (flip Tree . f) Null
```

- Traversion:

```
preorder :: Tree a-> [a]
inorder  :: Tree a-> [a]
preorder = foldT (\x t1 t2-> [x]++ t1++ t2) []
inorder  = foldT (\x t1 t2-> t1++ [x]++ t2) []
```

  - Äquivalente Definition ohne `foldT`:

```
preorder' Null          = []
preorder' (Tree l a r) = [a] ++preorder' l ++preorder' r
```

Modellierung von Fehlern: `Maybe a`

- Typ `a` plus Fehlerelement
  - Im Prelude vordefiniert.

```
data Maybe a = Just a | Nothing
```

- `Nothing` wird im „Fehlerfall“ zurückgegeben. Bsp:

```
find :: (a-> Bool)-> [a]-> Maybe a
find p []     = Nothing
find p (x:xs) = if p x then Just x
                else find p xs
```

Geordnete Bäume

- Voraussetzung:
  - Ordnung auf `a` (`Ord a`)
  - Es soll für alle `Tree a l r` gelten:

    $$\texttt{member x l} \Rightarrow \texttt{x} < \texttt{a} \wedge \texttt{member x r} \Rightarrow \texttt{a} < \texttt{x}$$

- Test auf Enthaltensein vereinfacht:

```
member :: Ord a=> Tree a-> a-> Bool
member Null _ = False
member (Tree l a r) b
   | b < a  = member l b
   | a == b = True
   | b > a  = member r b
```

- Ordnungserhaltendes Einfügen

```
insert :: Ord a=> Tree a-> a-> Tree a
insert Null a = Tree Null a Null
insert (Tree l a r) b
  | b < a  = Tree (insert l b) a r
  | b == a = Tree l a r
  | b > a  = Tree l a (insert r b)
```

- Problem: man kann trotzdem ungeordnete Bäume erzeugen.

- Löschen:

```
delete :: Ord a=> a-> Tree a-> Tree a
delete x Null = Null
delete x (Tree l y r)
   | x < y  = Tree (delete x l) y r
   | x == y = join l r
   | x > y  = Tree l y (delete x r)
```

  - `join` fügt zwei Bäume ordnungserhaltend zusammen:
  - Wurzel des neuen Teilbaums: Knoten links unten im rechten Teilbaum (oder Knoten rechts unten im linken Teilbaum)

- `join` fügt zwei Bäume ordnungserhaltend zusammen.
- Wurzel des neuen Teilbaums: Knoten links unten im rechten Teilbaum.
- `splitTree` spaltet Baum in Knoten links unten und Rest.

```
join ::  Tree a-> Tree a-> Tree a
join xt Null = xt
join xt yt   = Tree xt u nu where
  (u, nu) = splitTree yt
  splitTree :: Tree a-> (a, Tree a)
  splitTree (Tree Null a t) = (a, t)
  splitTree (Tree lt a rt) =
    (u, Tree nu a rt) where
       (u, nu) = splitTree lt
```

Ausgeglichene Bäume

- Ein Baum ist ausgeglichen, wenn
  - beide Unterbäume ausgeglichen sind
  - der Höhenunterschied zwischen den beiden Unterbäumen höchstens eins beträgt.
- Implementation: bei Einfügen und Löschen ggf. rotieren.
- Im Knoten Höhe des Baumes an dieser Stelle

```
data AVLTree a = AVLNull
               | AVLTree Int
                    (AVLTree a) a (AVLTree a)
```

- Neuen Knoten anlegen:
  - `mkNode` legt neuen Knoten an
  - `mkAVL` kreiiert ausgeglichenen Baum

```
mkNode :: AVLTree a-> a-> AVLTree a-> AVLTree a
mkNode l n r = AVLTree h l n r where
                  h = 1+ max (ht l) (ht r)
ht :: AVLTree a-> Int
ht AVLNull              = 0
ht (AVLTree h _ _ _) = h

mkAVL :: AVLTree a-> a-> AVLTree a-> AVLTree a
```

- Löschen, Einfügen: wie vorher, `mkAVL` statt Konstruktor

- Ordnungserhaltendes Einfügen

```
insert' :: Ord a=> AVLTree a-> a-> AVLTree a
insert' AVLNull a = mkNode AVLNull a AVLNull
insert' (AVLTree n l a r) b
  | b < a  = mkAVL (insert' l b) a r
  | b == a = AVLTree n l a r
  | b > a  = mkAVL l a (insert' r b)
```

  - `mkAVL` muss ggf. rotieren

- Löschen

```
delete' :: Ord a=> a-> AVLTree a-> AVLTree a
delete' x AVLNull = AVLNull
delete' x (AVLTree h l y r)
  | x < y  = mkAVL (delete' x l) y r
  | x == y = join' l r
  | x > y  = mkAVL l y (delete' x r)
```

  - `join'` fügt zwei Bäume ordnungserhaltend zusammen

- Wurzel des neuen Teilbaums: Knoten links unten im rechten Teilbaum
- `splitTree` spaltet Baum in Knoten links unten und Rest

```
join' ::  AVLTree a-> AVLTree a-> AVLTree a
join' xt AVLNull = xt
join' xt yt   = mkAVL xt u nu where
  (u, nu) = splitTree' yt
  splitTree' :: AVLTree a-> (a, AVLTree a)
  splitTree' (AVLTree h AVLNull a t) = (a, t)
  splitTree' (AVLTree h lt a rt) =
    (u, mkAVL nu a rt) where
       (u, nu) = splitTree' lt
```

- Hilfsfunktionen:
  - Balance

```
bias :: AVLTree a-> Int
bias (AVLTree _ lt _ rt) = ht lt - ht rt
```

  - Rechtsrotation

```
rotr :: AVLTree a-> AVLTree a
rotr (AVLTree _ (AVLTree _ ut y vt) x rt) =
  mkNode ut y (mkNode vt x rt)
```

  - Linksrotation

```
rotl :: AVLTree a-> AVLTree a
rotl (AVLTree _ ut y (AVLTree _ vt x rt)) =
  mkNode (mkNode ut y vt) x rt
```

- Hauptfunktion:
  - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei.
  - Gegeben solange nur ein Knoten gelöscht oder eingefügt wird.

```
mkAVL xt y zt
  | hz+1< hx &&
    bias xt< 0 = rotr (mkNode (rotl xt) y zt)
  | hz+1< hx   = rotr (mkNode xt y zt)
  | hx+1< hz &&
    0< bias zt = rotl (mkNode xt y (rotr zt))
  | hx+1< hz   = rotl (mkNode xt y zt)
  | otherwise  = mkNode xt y zt
    where hx= ht xt; hz= ht zt
```

Zusammenfassung

- Algebraische Datentypen erlauben Datenabstraktion durch
  - Trennung zwischen Repräsentation und Semantik und
  - Typsicherheit.
- Algebraischen Datentypen sind frei erzeugt.
- Bekannte algebraische Datentypen:
  - Aufzählungen, Produkte, Varianten;
  - `Maybe a`, Listen, Bäume
- Für geordnete und ausgeglichene Bäume:
  information hiding nötig — nächste Vorlesung.

Vorlesung vom 03.12.2001

Inhalt

- Letzte VL:
  - Datenabstraktion durch algebraische Datentypen
  - Mangelndes information hiding
- Heute: abstrakte Datentypen (ADTs) in Haskell
- Beispiele für bekannte ADTs:
  - `Store`
  - Stapel und Schlangen: `Stack` und `Queue`
  - Endliche Mengen: `Set`
  - Relationen und Graphen

Abstrakte Datentypen

- Ein abstrakter Datentyp besteht aus einem Typ und Operationen darauf.
- Beispiele:
  - Speicher, mit leerer Speicher, lesen, schreiben;
  - Stapel, mit Operationen leerer Stapel, `push`, `pop`, `top`;
  - Schlangen, mit leerer Schlange, einreihen, Kopf, nächstes Element.
- Repräsentation des Typen versteckt.

Module in Haskell

- Ein Modul umfaßt:
  - Definitionen von Typen, Funktionen, Klassen
  - Deklaration der nach außen sichtbaren Definitionen
- Syntax:

  `module` *Name* `(`*sichtbare Bezeichner*`)` `where` *Rumpf*
  - *sichtbare Bezeichner* können leer sein

- Einfaches Beispiel: ein einfacher Speicher (`Store`)
  - Typ `Store a b`, parametrisiert über
    - ▷ Indextyp (muß Gleichheit, oder besser Ordnung, zulassen)
    - ▷ Wertyp
  - Konstruktor: leerer Speicher `initial :: Store a b`
  - Wert lesen `value :: Store a b-> a-> Maybe b`
    - ▷ Möglicherweise undefiniert.
  - Wert schreiben `update :: Store a b-> a -> b-> Store a b`

```
module Store(Store,
             initial, -- Store a b
             value,   -- Store a b-> a-> Maybe b
             update,  -- Store a b-> a-> b-> Store a b
            ) where
```

- Erste Implementation:
  - Speicher als Liste von Paaren (NB. `data`, nicht `type`)

    ```
    data Store a b = St [(a, b)]
    ```

  - Leerer Speicher: leere Liste

    ```
    initial  = St []
    ```

  - Lookup: Listenkomprehension

    ```
    value (St ls) a
      = case filter ((a ==).fst) ls of
          (_,x):_ -> Just x
          []  -> Nothing
    ```

    Neu: `case` für Fallunterscheidungen

  - Update: neues Paar vorne anhängen

    ```
    update (St ls) a b  = St ((a, b): ls)
    ```

- Zweite Implementation:
  - Speicher als Funktion

    ```
    data Store a b = St (a-> Maybe b)
    ```

  - Leerer Speicher: konstant undefiniert

    ```
    initial  = St (const Nothing)
    ```

  - Lookup: Funktion anwenden

    ```
    value (St f) a  = f a
    ```

  - Update: punktweise Funktionsdefinition

    ```
    update (St f) a b
      = St (\x-> if x== a then Just b else f x)
    ```

- Ein Interface, zwei mögliche Implementierungen.

- Export von Datentypen:
  - `Store(..)` exportiert Konstruktoren
    - ▷ Implementation sichtbar
    - ▷ Pattern matching möglich
  - `Store` exportiert nur den Typ
    - ▷ Implemtation nicht sichtbar
    - ▷ Kein pattern matching möglich
  - Typsynonyme immer sichtbar
- Kritik Haskell:
  - Exportsignatur nicht im Kopf (nur als Kommentar)
  - Exportsignatur nicht von Implementation zu trennen (gleiche Datei!)

- Benutzung eines ADTs — Import.

  `import` $Name$ `[qualified] [hiding] (`$Bezeichner$`)`

  - Ohne Bezeichner wird alles importiert
  - `qualified`: qualifizierte Namen

    ```
    import Store2 qualified
    f = Store2.initial
    ```

  - `hiding`: Liste der Bezeichner wird versteckt

    ```
    import Prelude hiding (foldr)
    foldr f e ls = ...
    ```

  - Mit `qualified` und `hiding` Namenskonflikte auflösen.

- Zwei weitere Beispiele:
- Stacks
  - Typ: `St a`
  - Initialwert:

    ```
    empty :: St a
    ```
  - Wert ein/auslesen

    ```
    push :: a-> St a-> St a
    top  :: St a-> a
    pop  :: St a-> St a
    ```
  - Test auf Leer

    ```
    isEmpty :: St a-> Bool
    ```
  - Last in, first out.
- Queues
  - Typ: `Qu a`
  - Initialwert:

    ```
    empty :: Qu a
    ```
  - Wert ein/auslesen

    ```
    enq   :: a-> Qu a-> Qu a
    first :: Qu a-> a
    deq   :: Qu a-> Qu a
    ```
  - Test auf Leer

    ```
    isEmpty :: Qu a-> Bool
    ```
  - First in, first out.

- Implementation von `Stack`: Liste
  - Sehr einfach wg. last in, first out
  - `push` ist `(:)`, `top` ist `tail`, `pop` ist `tail`.
- Implementation von `Queue`:
  - Mit einer Liste — Problem: am Ende anfügen oder abnehmen ist teuer
  - Deshalb zwei Listen:
    - Erste Liste: zu entnehmende Elemente
    - Zweite Liste: hinzugefügte Elemente rückwärts

```
module Queue(Qu, empty, isEmpty, enq, first, deq) where
data Qu a = Qu [a] [a]
empty = Qu [] []
```

- Invariante: erste Liste leer gdw. Queue leer

```
isEmpty (Qu xs _) = null xs
```

- Erstes Element steht vorne in erster Liste

```
first (Qu (x:xs) _) = x
```

- Bei `enq` und `deq` Invariante prüfen

```
enq x (Qu xs ys) = check xs (x:ys)
deq (Qu (_:xs) ys) = check xs ys
```

- `check` prüft die Invariante

```
check [] ys = Qu (reverse ys) []
check xs ys = Qu xs ys
```

Endliche Mengen

- Eine endliche Menge ist eine Sammlung von Elementen so dass
  - kein Element doppelt ist, und
  - die Elemente nicht angeordnet sind.
- Schnittstelle:
  - Typ `Set a`
    - ▷ `a` muß Gleichheit, besser Ordnung zulassen.

```
module Set (Set,
```

  - Menge leer und Test auf leer

```
  empty,   -- Set a
  isEmpty, -- Set a-> Bool
```

- einelementige Menge, Elementtest

```
sing,    -- a-> Set a
member,  -- Ord a=> Set a-> a-> Bool
```

- Vereinigung, Schnitt und Differenz

```
union,   -- Ord a=> Set a-> Set a-> Set a
inter,   -- Ord a=> Set a-> Set a-> Set a
diff,    -- Ord a=> Set a-> Set a-> Set a
```

- Teilmengen (Gleichheit durch Typklasse `Eq`)

```
subSet,  -- Ord a=> Set a-> Set a-> Bool
```

- Funktionen höherer Ordnung

```
mapSet,    -- Ord b=> (a-> b)-> Set a-> Set b
filterSet, -- (a-> Bool)-> Set a-> Set a
foldSet,   -- (a-> b-> b)-> b-> Set a-> b
```

- Hilfsfunktionen: Liste in Menge, Aufzählung der Elemente.

```
  makeSet,    -- Ord a=> [a]-> Set a
  enumSet     -- Ord a=> Set a-> [a]
  ) where
```

- Implementation:
  - Mengen als geordnete oder ausgewogene Bäume — Problem:
    - ▷ Hinzufügen und Suchen einzelner Elemente effizient
    - ▷ Mengenoperationen (Schnitt, Vereinigung) ineffizient
  - Deshalb: Mengen als geordnete Listen ohne Wiederholung

    ```
    data Set a = Set [a] deriving (Eq, Ord)
    ```
  - Implementation von `empty`, `isEmpty`, `sing` sind trivial.
  - `member` nutzt Ordnung (deshalb nicht `elem`)

    ```
    member (Set xs) x = mem xs x where
      mem [] _      = False
      mem (y:ys) x = x== y || x< y && mem ys x
    ```

- Gleichheit von Mengen ist Gleichheit der repräsentierenden Listen
- `subSet` nutzt Ordnung:

```
subSet (Set xs) (Set ys) = sub xs ys
sub :: Ord a=> [a]-> [a]-> Bool
sub [] _  = True
sub _  [] = False
sub (x:xs) (y:ys)
  | x< y  = False
  | x== y = sub xs ys
  | x> y  = sub (x:xs) ys
```

  - ▷ Linearer Aufwand wg. Ordnung
  - ▷ Deshalb keine vordefinierten Funktionen — quadratischer Aufwand:

```
sub xs ys = all (flip elem ys) xs
```

- Vereinigung: doppelte Elemente entfernen und Ordnung beibehalten

```
union (Set xs) (Set ys) = Set (uni xs ys)
uni :: Ord a=> [a]-> [a]-> [a]
uni [] ys = ys
uni xs [] = xs
uni (x:xs) (y:ys)
  | x< y  = x: uni xs (y:ys)
  | x== y = x: uni xs ys
  | x> y  = y: uni (x:xs) ys
```

- Schnitt (`inter`) und Mengendifferenz (`diff`) ähnlich
- `mapSet`, `filterSet`, `foldSet` benutzen `map`, `filter`, `foldr` auf Listen
  - ▷ Bei `map` beachten: Ergebnis muß nicht automatisch Liste repräsentieren – `makeSet` benutzen

- `makeSet` auch exportiert:

```
makeSet :: Ord a=> [a]-> Set a
makeSet = Set . nub . sort
```

  ▷ `nub :: Eq a=> [a]-> [a]` (aus der Standard-Bücherei: `import List`) entfernt Duplikate

- Sets anzeigen:

```
instance Show a=> Show (Set a) where
  show (Set xs) =
    "{"++ concat (intersperse ", " (map show xs))++ "}"
```

  ▷ `intersperse :: a-> [a]-> [a]` (aus der Standard-Bücherei) fügt einzelnes Element zwischen Elemente der Liste ein.

  ▷ Klasseninstanzen werden automatisch exportiert.

Zusammenfassung `Set`

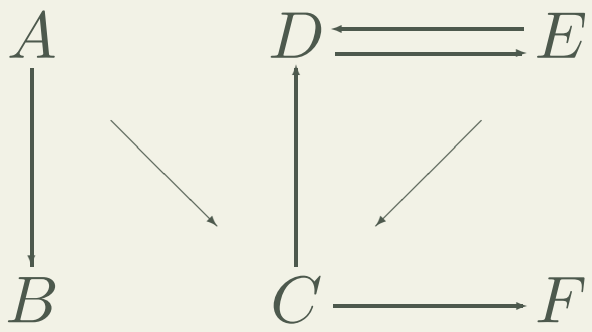
- Linearer Aufwand vieler Operationen durch Ausnutzen der Ordnung.
- Möglich nur durch Verstecken der Datenrepräsentation.

Relationen und Graphen

- Relation R auf X:
  - Menge von Paaren:
    $R \subseteq X \times X$
  - Bsp:

```
{ (A, B), (A, C)
  (C, D), (C, F)
  (D, E), (E, C)
  (E, D)}
```

  - Ungewichteter, gerichterer Graph
  - Bsp:



Algorithmen auf Graphen

- Viele Operationen auf Relationen (Relationenalgebra)
- Mit diesen Operationen Graphalgorithmen formulieren.
  - Pfade: es gibt einen Pfad von p nach q in R gdw. (p, q) im transitiv-reflexiven Abschluß R^* von R:
    $$(p, q) \in R^*$$
  - Transitiv-reflexiver Abschluß R^* von R:
    Kleinste transitive und reflexive Relation, die R enthält.
  - R ist reflexiv gdw. für alle $p \in X$, $(p, p) \in R$.
  - R ist transitiv gdw. für alle $p, q, r \in X$ mit $(p, q) \in R$ und $(q, r) \in R$ gilt, dass $(p, r) \in R$.

- Konstruktion des reflexiven Abschlusses $R^=$:
  Vereinigung mit der Identität auf X

$$1_X = \{(x, x) \mid x \in X\}$$

$$R^= = R \cup 1_R$$

- Konstruktion des transitiven Abschlusses R^+:
  Abschluß unter Relationskomposition

$$R; S = \{(x, z) \mid (x, y) \in R, (y, z) \in S\}$$

  d.h. R^+ ist die kleinste Relation S, so dass $R \subseteq S$ und $S; S = S$

- Implementation:
  - Relation: Mengen von Paaren

    ```
    type Rel a = Set (a, a)
    ```

  - Identität auf R:

    ```
    ident :: Ord a=> Rel a-> Rel a
    ident = mapSet (\x-> (x, x)) . elems
    ```

  - Elemente einer Relation: $\{x \mid (x,y) \in R \vee (y,x) \in R\}$

    ```
    elems :: Ord a=> Rel a-> Set a
    elems r = mapSet fst r `union` mapSet snd r
    ```

  - Reflexiver Abschluß:

    ```
    refl :: Ord a=> Rel a-> Rel a
    refl r = r `union` (ident r)
    ```

- Komposition:
  - ▷ Kreuzprodukt $R \times S$ der Relationen bilden,
  - ▷ Paare $((x, y), (y, z))$ herausfiltern,
  - ▷ diese auf die äußeren Komponenten (x, z) abbilden.

```
compose :: Ord a=> Rel a-> Rel a-> Rel a
compose r s =
  mapSet outer (filterSet eqmid (setProd r s)) where
  eqmid ((a, b), (c, d)) = b== c
  outer ((a, b), (c, d)) = (a, d)
```

- Kreuzprodukt zweier Mengen:

$$A \times B = \{(x, y) \mid x \in X, y \in Y\}$$

Berechnung:

- ▷ alle Elemente aus B einzeln mit A paaren (`adjoin`),
- ▷ Vereinigung („flachklopfen“) der Resultatsmengen (`unionSet`).

```
setProd :: (Ord a, Ord b)=> Set a-> Set b-> Set (a, b)
setProd a = unionSet . mapSet (adjoin a)

adjoin  :: (Ord a, Ord b)=> Set a-> b-> Set (a, b)
adjoin s e = mapSet (\x-> (x, e)) s

unionSet :: Ord a=> Set (Set a)-> Set a
unionSet = foldSet union empty
```

- Berechnung des transitiven Abschlusses:
  kleinster Fixpunkt der Funktion, die S auf $S \cup S; R$ abbildet.

```
trans :: Ord a=> Rel a-> Rel a
trans r = lfp compOne r where
  compOne s = s `union` (s `compose` r)
```

- Kleinster Fixpunkt einer Funktion f ist kleinstes x so dass $fx = x$
  Berechnung: f solange anwenden, bis sich nichts mehr ändert.

```
lfp :: Eq a=> (a-> a)-> a-> a
lfp f x
  | x == f x  = x
  | otherwise = lfp f (f x)
```

  ▷ NB: `lfp` muß nicht terminieren; Termination hier garantiert durch Endlichkeit von R.

Stark verbundene Komponenten

- Zwei Knoten p, q sind stark verbunden
  gdw. es gibt einen Pfad von p nach q und von q nach p:

$$p \sim q \Leftrightarrow (p,q) \in R^* \wedge (q,p) \in R^*$$
$$p \sim q \Leftrightarrow (p,q) \in R^* \cap (R^*)^{-1}$$
$$\sim \; = R^* \cap (R^*)^{-1}$$

  - Inverses R^{-1} von R:

$$R^{-1} = \{(y,x) \mid (x,y) \in R\}$$

- Stark verbundenene Komponenten:
  Äquivalenzklassen von stark verbundenen Knoten:

```
scg :: Ord a=> Rel a-> Set (Set a)
```

- Implementation:
  - Erster Schritt: Berechung der Relation $\sim$

```
connect :: Ord a=> Rel a-> Rel a
connect r = (transrefl r) `inter` (inv (transrefl r))
```

- Inverses von R:

$$R^{-1} = \{(y, x) \mid (x, y) \in R\}$$

```
inv :: Ord a=> Rel a-> Rel a
inv = mapSet (\(x, y)-> (y, x))
```

- Zweiter Schritt: Berechung der Äquivalenzklassen
  - ▷ Gegeben Relation $\{(A, A), (B, B), (C, C), (C, D), (C, E), \ldots\}$
  - ▷ Gesucht Äquivalenzklassen $\{\{A\}, \{B\}, \{C, D, E\}, \{F\}\}$

- Relation R auf X als partielle, mehrdeutige Funktion $r : X \to \mathbb{P}(X)$ gegeben R, dann $r(x) = \{y \mid (x, y) \in R\}$

```
img :: Ord a=> Rel a-> a-> Set a
img r v = mapSet snd (filterSet ((== v) . fst) r)
```

  $\triangleright$ R Äquivalenzrelation: $r(x)$ gibt zu x äquivalente Elemente.

- Berechung der Äquivalenzklassen:
  Zu den Elementen der Relation äquivalente Elemente hinzufügen

```
classes :: Ord a=> Rel a-> Set (Set a)
classes r = mapSet (img r) (elems r)
```

- Damit Hauptfunktion:

```
scg = classes . connect
```

Zusammenfassung

- Abstrakter Datentyp: Datentyp plus Operationen.
- Module in Haskell:
  - Module begrenzen Sichtbarkeit
  - Importieren, ggf. qualifiziert oder nur Teile
- Beispiele für ADTs:
  - Steicher: mehrere Implementationen
  - Stapel und Schlangen: gleiche Signatur, andere Semantik
  - Implementation von Schlangen durch zwei Listen
  - Endliche Mengen: implementiert durch geordnete Listen, Verstecken der Repräsentation nötig für Korrektheit
  - Relationen und Graphen: Graphalgorithmen durch abstrakte Sicht

Vorlesung vom 10.12.2001

Inhalt

- Organisatorisches:
  - Scheinanmeldung (bis Semesterende)
  - Wie geht's weiter?
- Verzögerte Auswertung
  - . . . und was wir davon haben.
- Unendliche Datenstrukturen
  - . . . und wozu sie nützlich sind.
- Beispiel: Parsierung von Texten.
- Effizienzaspekte
  - Wie wir schnellere Programme schreiben.

Wie geht's nach Weihnachten weiter?

- Entweder: Graphik
  - Eine einfache Graphikbücherei für `hugs`.
  - Funktionale Grafik und Animationen.
- oder: Logik
  - Automatische Beweise von einfachen Sätzen.
  - Implementation eines taktischen Theorembeweisers.

Verzögerte Auswertung

- Auswertung: Reduktion von Gleichungen
  - Strategie: Von außen nach innen; von rechts nach links
- Effekt: Parameterübergabe *call by need*, nicht-strikt
  - Beispiel:

    ```
    f x y = y; enum x y = [x..y]
    ```

    ```
    f (enum 1 2) (enum 5 6) ⇝ enum 5 6 ⇝ [5,6]
    ```

  - Erstes Argument von `f` wird nicht ausgewertet.
  - Zweites Argument von `f` wird erst im Funktionsrumpf ausgewertet.
- Effekte der verzögerten Auswertung:
  - Datenorientierte Programme, unendliche Datenstrukturen.

Datenrepräsentationsabstraktion

- Verzögerte Auswertung erlaubt Abstraktion von konkreter Repräsentation der Daten.
- Bsp: Berechung von $\sum_{i=1}^{n} i^2$
  - Bilde Liste `[1..n]`
  - Quadriere jedes Element der Liste (`{map (^2)}`)
  - Bilde Summe (`sum :: [Int]-> Int`)

```
sqrsum n = sum (map (^2) [1..n])
```

- Es werden keine Listen von Zwischenergebnissen erzeugt
- Konzentration auf des Wesentliche.
- Bsp: Minimum einer Liste durch `head (msort xs)`

Unendliche Datenstrukturen: Ströme

- Ströme sind unendliche Listen.
  - ```
    twos = 2 : twos
    ```
  - Die natürlichen Zahlen:

    ```
    nat  = [1..]
    ```
  - Bildung von unendlichen Listen:

    ```
    cycle :: [a]-> [a]
    cycle xs = xs ++ cycle xs
    ```
- Repräsentation durch endliche, zyklische Datenstruktur: Kopf wird nur einmal ausgewertet.

  ```
  ones = (trace "Foo!" 1): ones
  ```
  ```

- Nützlich für Listen mit unbekannter Länge
- Bsp: Berechnung der ersten n Primzahlen
  - Eratosthenes — ab wo sieben?
  - Lösung: Berechnung aller Primzahlen, davon die ersten n.

```
sieve :: [Integer]-> [Integer]
sieve (p:ps) =
  p:(sieve (filter (\n-> n `mod` p /= 0) ps))
```

  - Es fehlt die Rekursionsverankerung: gesiebte Liste wird nie leer.

```
primes :: Int-> [Integer]
primes n = take n (sieve [2..])
```

  - Alternative: mit zusätzlicher Parameter explizit mitzählen . . .

- Zweites Bsp: Fibonacci-Zahlen
  -

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

  - Problem: exponentieller Aufwand.
  - Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
  - Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
              fibs  1  1  2  3  5  8 13 21 34 55
         tail fibs  1  2  3  5  8 13 21 34 55
tail (tail fibs)  2  3  5  8 13 21 34 55
```

  - Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Aufwand: linear, da `fibs` nur einmal ausgewertet wird.

Fallstudie: Parsierung

- Problem:
  - Gegeben: Grammatik
  - Gesucht: Funktion, Wörter der Grammatik erkennt
- Parser bildet Eingabe auf Parsierungen ab.
  - Mehrere Parsierungen möglich.
  - Backtracking möglich.
  - Durch verzögerte Auswertung dennoch effizient.

- Basisparser: erkennen Terminalsymbole
- Kombinatoren: setzen Basisparser zusammen, um Nichtterminalsymbole zu erkennen
  - Sequenzierung (erst A, dann B)
  - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

- Beispielgrammatik für arithmetische Ausdrücke:

```
Expr   ::= Term + Term | Term - Term | Term
Term   ::= Factor * Factor | Factor / Factor
         | Factor
Factor ::= Number | (Expr)
Number ::= Digit | Digit Number
Digit  ::= 0|...| 9
```

- Daraus abstrakte Syntax:

```
data Expr   = Plus Term Term
            | Minus Term Term
            | Term Term
            deriving (Eq, Show)
data Term   = Times Factor Factor
            | Div Factor Factor
            | Factor Factor
            deriving (Eq, Show)
data Factor = Number Int
            | Expr Expr   deriving (Eq, Show)
```

  - Elemente repräsentieren Parsierungen.

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp `a` und Ergebnis `b`

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren
  - Müssen Rest der Eingabe modellieren

- Welcher Typ für Parser?
  - Parser übersetzt Token in abstrakte Syntax
  - Parametrisiert über Eingabetyp `a` und Ergebnis `b`
  - Müssen mehrdeutige Ergebnisse modellieren
  - Müssen Rest der Eingabe modellieren
  - Damit:

    ```
    type Parse a b = [a]-> [(b, [a])]
    ```

- Beispiel: `parse "3+4*5"` $\rightsquigarrow$

```
[ (3, "+4*5"),
  Plus 3 4, "*5"),
  (Plus 3 (Times 4 5), "")]
```

- Basisparser
  - Erkennt nichts:

```
none :: Parse a b
none = const []
```

  - Erkennt alles:

```
suceed :: b-> Parse a b
suceed b inp = [(b, inp)]
```

  - Erkennt einzelne Zeichen:

```
token :: Eq a=> a-> Parse a a
token t = spot (== t)
spot  :: (a-> Bool)-> Parse a a
spot p []     = []
spot p (x:xs) = if p x then [(x, xs)] else []
```

- Kombinatoren
  - Alternierung:

```
alt :: Parse a b-> Parse a b-> Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

  - Sequenzierung:
    - ▷ Rest des ersten Parsers als Eingabe für den zweiten

```
infixr 5 >*>
(>*>) :: Parse a b-> Parse a c-> Parse a (b, c)
(>*>) p1 p2 i = [((y, z), r2) | (y, r1) <- p1 i,
                                (z, r2) <- p2 r1]
```

- Eingabe weiterverarbeiten:

```
build :: Parse a b-> (b-> c)-> Parse a c
build p f inp = [(f x, r) | (x, r)<- p inp]
```

- Listen und nicht-leere Listen:

```
list :: Parse a b-> Parse a [b]
list p = ((p >*> list p) `build` (uncurry (:)))
         `alt` (suceed [])
some :: Parse a b-> Parse a [b]
some p = (p >*> list p) `build` (uncurry (:))
```

- Sequenzierung rechts/links:

```
infixr 5 *>, >*
(*>) :: Parse a b-> Parse a c-> Parse a c
(>*) :: Parse a b-> Parse a c-> Parse a b
p1 *> p2 = (p1 >*> p2) `build` snd
```

- Der Kern des Parsers (Exzerpt):

```
pExpr :: Parse Char Expr
pExpr = (((pTerm >* token '+') >*> pTerm)
                  `build` (uncurry Plus))
          `alt` (pTerm `build` Term)
pTerm :: Parse Char Term
pTerm = (((pFactor >* token '*') >*> pFactor)
                  `build` (uncurry Times))
          `alt` (pFactor `build` Factor)
pFactor :: Parse Char Factor
pFactor = ((some (spot isDigit)) `build` (Number . read))
            `alt` ((token '(' *> pExpr >* token ')')
                                        `build` Expr)
```

- Mangel: 3+4+5 wird nicht erkannt.

- Behebung: leichte Änderung der Grammatik . . .

```
Expr   ::= Term + Expr | Term - Expr| Term
Term   ::= Factor * Term | Factor / Term | Factor
Factor ::= Number | (Expr)
Number ::= Digit | Digit Number     Digit ::= 0 |...| 9
```

- . . . der Grammatik . . .

```
data Expr = Plus Term Expr | Minus Term Expr | Term Term
            deriving (Eq, Show)
data Term = Times Factor Term | Div Factor Term
          | Factor Factor
          deriving (Eq, Show)
data Factor = Number Int | Expr Expr
              deriving (Eq, Show)
```

- . . . und des Parsers:

```
pExpr :: Parse Char Expr
pExpr = (((pTerm >* token '+') >*> pExpr)
                    `build` (uncurry Plus))
        `alt` (pTerm `build` Term)
pTerm :: Parse Char Term
pTerm = (((pFactor >* token '*') >*> pTerm)
                      `build` (uncurry Times))
        `alt` (pFactor `build` Factor)
pFactor :: Parse Char Factor
pFactor = ((some (spot isDigit)) `build` (Number . read))
          `alt` ((token '(' *> pExpr >* token ')')
                                    `build` Expr)
```

Zusammenfassung Parserkombinatoren

- Erlauben schnelle Konstruktion von Parsern aus beliebigen Grammatiken.
- Durch verzögerte Auswertung annehmbare Effizienz.
- Nichts für große Eingaben (dann: Parsergeneratoren).
  - Z.B. geeignet für Kommandozeileneingabe.
  - Ungeeignet als Front-End für einen Übersetzer.

Effizienzaspekte

- Beste Lösung: bessere Algorithmen.
- Zweitbeste Lösung: Büchereien nutzen.
- Effizenzverbesserungen durch
  - Endrekursion und Striktheit: Speicherlecks vermeiden
  - The eternal conflict: speed vs. space

Termgraphendarstellung

- Ausdrücke werden intern durch Termgraphen dargestellt.
- Argument wird nie mehr als einmal ausgewertet:
  -
    ```
    import IOExts (trace)
    ```
  - `trace :: String-> a-> a` druckt `String` bei Auswertung.
    ```
    f :: Int-> Int-> Int
    f x y = x+ x
    test1 = f (trace "Eins\n" (3+2))
              (trace "Zwei\n" (3+2))
    ```
- *Sharing* von Teilausdrücken
  - Explizit mit `where`

- Implizit (`ghc`)

Endrekursion (*tail recursion*)

- Eine Funktion ist endrekursiv, wenn kein rekursiver Aufruf in einem geschachtelten Ausdruck steht.
  - Endrekursion entspricht `goto` oder `while` in imperativen Sprachen.
  - Endrekursive Funktionen werden in Schleifen übersetzt.
  - Nicht-endrekursive Funktionen verbrauchen Platz auf dem Stack.
- Bsp:
  - `fac` nicht endrekursiv:

```
fac n  = if n == 0 then 1 else n * fac (n-1)
```

  - `fac` endrekursiv:

```
fac0 n acc = if n == 0 then acc else fac0 (n-1) (n*acc)
fac' n     = fac0 n 1
```

- Überführung in endrekursive Form: Zwischenergebnisse in zusätzlichen Parameter akkumulieren.
- Aufgesammelte Zwischenergebnisse brauchen immer noch Platz: Auswertung erzwingen.

Striktheit

- Funktion `f` ist strikt in einem Argument `x`, wenn ein undefinierter Wert für das Argument die Funktion undefiniert werden läßt.
  - Bsp: `(+)` strikt in beiden Argumenten
  - `(&&)` strikt im ersten, nicht-strikt im zweiten:
    `False && (1/0 == 1/0)` $\leadsto$ `False`
- Strikte Argumente erlauben Optimierung.
  - Zum Beispiel Auswertung vor Aufruf bei Endrekursion.

- Auswertung erzwingen: `seq :: a-> b-> b` wertet erstes Argument aus
- Fakultät in konstantem Platzaufwand:

```
fac'' n = fac0 n 1 where
  fac0 n acc = seq acc (if n == 0 then acc
                        else fac0 (n-1) (n*acc))
```

`foldr` vs. `foldl`

- `foldr` ist nicht endrekursiv.
- `foldl` ist endrekursiv:

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

- `foldl' :: (a-> b-> a)-> a-> [b]-> a` ist endrekursiv und strikt.

- `foldl` endrekursiv, aber traversiert immer die ganze Liste.
- `foldl'` konstanter Platzaufwand, aber traversiert immer die ganze Liste.
- Wann welches `fold`?
  - Strikte Funktionen mit `foldl'` falten.
  - Wenn nicht die ganze Liste benötigt wird, `foldr`:

```
all :: (a-> Bool)-> [a]-> Bool
all p = foldr ((&&) . p) True
```

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: dictionaries von Klassenfunktionen.
  - Überladung muß zur Laufzeit aufgelöst werden.
- Bei kritischen Funktionen durch Angabe der Signatur Spezialisierung erzwingen.
- NB: Zahlen (numerische Literale) sind in Haskell überladen!
  - Bsp: `facs` hat den Typ `Num a=> a-> a`

```
facs n = if n == 0 then 1 else n* facs (n-1)
```

Listen sind keine Felder

- Wenn Felder benötigt werden, Felder verwenden!
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit.
- Felder:
  - Feste Länge
  - Zugriff auf n-tes Element in konstanter Zeit.
  - Abstrakt: Abbildung Index auf Daten

- Modul `Array` aus der Standardbücherei

```
data Ix a=> Array a b  -- abstract
array     :: (Ix a) => (a,a) -> [(a,b)]
                                -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
(//)      :: (Ix a) => Array a b -> [(a,b)]
                                -> Array a b
```

- Als Indexbereich geeignete Typen (Klasse `Ix`): `Int`, `Integer`, `Char`, `Char`, Tupel davon, Aufzählungstypen.

Zusammenfassung

- Verzögerte Auswertung erlaubt unendliche Datenstrukturen
  - Zum Beispiel: Ströme (unendliche Listen)
- Einfache Parsierung mit Parserkombinatoren
- Effizienz durch
  - Endrekursion
  - Striktheit
  - Spezialisierung

Vorlesung vom 17.12.2001

Inhalt

- Ein/Ausgabe in funktionalen Sprachen
- Wo ist das Problem?
- Der Datentyp IO.
- Vordefinierte Funktionen für E/A.
- Beispiel: `Nim`
- Wie würde man E/A implementieren?

Ein- und Ausgabe in funktionalen Sprachen

- **Problem:**

  „Funktion“ `readLine :: () -> String` würde referentielle Transparenz zerstören.

- **Lösung:**

  Abhängigkeit von der Umwelt am Typ `IO` erkennbar.

  IO als abstrakter Datentyp:

  - `IO t`

    Funktion vom Typ `t`, die Ein/Ausgabe betreibt (Aktion)

  - Aktionen können nur mit Aktionen komponiert werden

    „einmal IO, immer IO“

- IO als abstrakter Datentyp:

```
type IO t
(>>=)  :: IO a-> (a-> IO b)-> IO b
return :: a-> IO a
```

- Vordefinierte Funktionen (Prelude):
  - Zeile von `stdin` lesen:

```
getLine  :: IO String
```

  - String ausgeben:

```
putStr   :: String-> IO ()
```

  - String mit Zeilenvorschub ausgeben:

```
putStrLn :: String-> IO ()
```

- Ein einfaches Beispiel:

```
echo :: IO ()
echo = getLine >>= putStrLn >>= \_ -> echo
```

- Immer noch einfach:

```
ohce :: IO ()
ohce = getLine >>= putStrLn . reverse >> ohce
```

- Vordefinierte Abkürzung:

```
(>>) :: IO t-> IO u-> IO u
f >> g = f >>= \_ -> g
```

Syntaktischer Zucker für `IO`

- Abkürzende Schreibweise:

```
echo =                                  echo =
  getLine                                 do s<- getLine
  >>= \s-> putStrLn s          ⟺            putStrLn s
  >> echo                                 echo
```

  - Rechts sind `>>=`, `>>` `implizit`.
- Es gilt die Abseitsregel.
  - Einrückung der ersten Anweisung nach `do` bestimmt Abseits.
  - Ansonsten sagt `hugs`:

```
Last generator in do {...} must be an expression
```

Ein/Ausgabe in Haskell

- Ein/Ausgabe aus Dateien und `stdin,out` (Modul `IO`)
- Fehlerbehandlung
- Zufallszahlen (Modul `Random`)
- Zugriff auf das Dateisystem (Modul `Directory`)
  - Leider nicht in `hugs`.
- Kommandozeile, Umgebungsvariablen (Modul `System`)

Ein/Ausgabe mit Dateien

- Im Prelude vordefiniert:
  - Dateien schreiben (überschreiben, anhängen):

```
type FilePath =  String
writeFile     ::  FilePath -> String -> IO ()
appendFile    ::  FilePath -> String -> IO ()
```

  - Datei lesen (verzögert):

```
readFile      ::  FilePath             -> IO String
```

- Mehr Operationen im Modul `IO` der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf `Handle`

- Ein Beispiel:

```
wc :: String-> IO ()
wc file =
  do c<- readFile file
     putStrLn (show (length (lines c))
                                ++ " lines ")
     putStrLn (show (length (words c))
                                ++ " words, and ")
     putStrLn (show (length c)++ " characters. ")
```

  - Nicht sehr effizient — Inhalt der Datei wird im Speicher gehalten.

Noch ein Beispiel: `Nim` revisited

- Implementation von `Nim`:
  - Am Anfang Anzahl der Hölzchen auswürfeln.
  - Eingabe des Spielers einlesen.
  - Wenn nicht mehr zu gewinnen, aufgeben, ansonsten ziehen.
  - Wenn ein Hölzchen über ist, hat Spieler verloren.
- Zufallszahlen: Modul `Random`
  - ```
    class Random a where
      randomRIO :: (a, a)-> IO a
      randomIO  :: IO a
    ```
  - Instanzen von `Random`: Basisdatentypen.
  - `Random` enthält ferner Zufallsgeneratoren für Pseudozufallszahlen.
    ```

- `Nim` revisited:
  - Importe und Hilfsfunktionen:

    ```
    import Random (randomRIO)
    ```

  - `wins` liefert `Just`n, wenn Zug n gewinnt; ansonsten `Nothing`

    ```
    wins :: Int-> Maybe Int
    wins n =
      if m == 0 then Nothing else Just m where
         m = (n- 1) `mod` 4
    ```

- Hauptfunktion:

```
play :: Int-> IO ()
play n =
  do putStrLn ("Es sind "++ show n ++
               " Hölzchen im Haufen.")
     if n== 1 then putStrLn "Ich habe gewonnen!"
        else do m<- getInput
                case wins (n-m) of
                  Nothing -> putStrLn "Ich gebe auf."
                  Just l  -> do putStrLn ("Ich nehme "
                                          ++ show l)
                                play (n-(m+l))
```

- Noch zu implementieren: Benutzereingabe

```
getInput' :: IO Int
getInput' =
  do putStr "Wieviele nehmen Sie? "
     n <- do s<- getLine
             return (read s)
     if n<= (0::Int) || n>3 then
       do putStrLn "Ungültige Eingabe!"
          getInput'
       else return n
```

- Nicht sehr befriedigend: Abbruch bei falscher Eingabe.

- Fehlerbehandlung:
  - Fehler werden durch abstrakten Datentyp `IOError` repräsentiert
  - Fehlerbehandlung durch Ausnahmen (ähnlich Java)

```
ioError :: IOError -> IO a     -- "throw"
catch   :: IO a-> (IOError-> IO a) -> IO a
```

  - Nur in `IO t` können Fehler behandelt werden.
  - Fangbare Benutzerfehler mit `userError::String-> IOError`.
  - `IOError` kann analysiert werden— Auszug aus Modul `IO`:

```
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool
ioeGetErrorString     :: IOError -> String
ioeGetFileName        :: IOError -> Maybe FilePath
```

- Robuste Eingabe:
  - `readIO :: Read a=> String-> IO a` wirft im Fehlerfall Ausnahme, die gefangen werden kann.

```
getInput :: IO Int
getInput =
  do putStr "Wieviele nehmen Sie? "
     n <- catch (do s<- getLine
                    readIO s)
                (\_ -> do putStrLn "Eingabefehler."
                          getInput)
     if n<=(0::Int) || n>3 then
       do putStrLn "Ungültige Eingabe!"
          getInput
       else return n
```

- Haupt- und Startfunktion:
  - Begrüßung
  - Anzahl Hölzchen auswürfeln
  - starten.

```
main :: IO ()
main = do putStrLn "\nWillkommen bei Nim!\n"
          n <- randomRIO(5,49)
          play n
```

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich:
  - Endlosschleife:

```
forever :: IO a-> IO a
forever a = a >> forever a
```

  - Iteration (feste Anzahl)

```
forN :: Int-> IO a-> IO ()
forN n a | n == 0    = return ()
         | otherwise = a >> forN (n-1) a
```

- Iteration (variabel, wie `for` in Java)

```
for :: (a, a-> Bool, a-> a)-> (a-> IO ())-> IO ()
for (start, cont, next) cmd =
  iter start where
    iter s = if cont s then cmd s >> iter (next s)
             else return ()
```

- Vordefinierte Kontrollstrukturen (Prelude)
  - Listen bearbeiten:

```
sequence :: [IO a]-> IO [a]
sequence (c:cs) = do x  <- c
                     xs <- sequence cs
                     return (x:xs)

sequence_ :: [IO ()]-> IO ()
```

  - Map für Monaden:

```
mapM :: (a-> IO b)-> [a]-> IO [b]
mapM f = sequence . map f
mapM_ :: (a-> IO ())-> [a]-> IO ()
mapM_ f = sequence_ . map f
```

  - Andere (wie `filterM`) im Modul `Monad`.

Der Blick hinter die Kulissen.

- `IO a` ist keine schwarze Magie.
- Grundprinzip:
  - Der Systemzustand wird durch das Programm gereicht.
  - Darf dabei nie dupliziert werden.
  - Auswertungsreihenfolge muß erhalten bleiben.
- Implementation:
  - Mit Systemzustand `S`:

```
type IO' a = (S-> (a, S))
```

- Komposition:
  - Wir wissen: `a-> s-> (b, s)` $\cong$ `(a, s)-> (b, s)`
  - Damit Definition von `(>>=)`:

```
(>>=) :: IO' a-> (a-> IO' b)-> IO' b
      :: s-> (a, s)-> (a-> (s-> (b, s))) -> (s-> (b, s))
      :: s-> (a, s)-> ((a, s)-> (b, s))  -> (s-> (b, s))
```

  - Entspricht `(>.>) :: (a-> b)-> (b-> c)-> (a-> c)`
  - Im Prinzip:

```
(>>=) :: IO' a-> (a-> IO' b)-> IO' b
f >>= g = f >.> uncurry g
```

    Aber:
    - ▷ Typsynonym nicht abstrakt.
    - ▷ `(>>=)` gehört zu Klasse `Monad`.

- Komposition:
  - `IO'` als abstrakter Datentyp, Systemzustand `s` als Parameter:

    ```
    data IO' s a = IO' (s-> (a, s))
    ```

  - Lokale Funktion: `IO'` auspacken

    ```
    unwrap :: IO' s a-> (s-> (a, s))
    unwrap (IO' f) = f
    ```

  - Instanz der Typklasse `Monad`:

    ```
    instance Monad (IO' s) where
      f >>= g   = IO' (unwrap f >.> uncurry (unwrap. g))
      return  a = IO' (\s-> (a, s))
    ```

- Implementation von Ein/Ausgabe:
  - Systemzustand: Eingabestrom und Ausgabestrom

```
type IO1 = IO' (String, String)
```

  - Eingabe: ersten String in Eingabestrom

```
getLine' :: IO1 String
getLine'  = IO' f where
                f (i, o)= (hd, (if null tl then tl
                                else tail tl, o)) where
                  (hd, tl) = span (/= '\n') i
```

  - Ausgabe: String an den Ausgabestrom hängen

```
putStr' :: String-> IO1 ()
putStr' s = IO' (\(i, o)-> ((), (i, o++ s)))
```

- Programm laufen lassen:

```
run :: IO1 ()-> String-> String
run prog i = o where
  IO' p = prog
  ((), (_, o))= p (i, [])
```

- Beispielprogramm:

```
countXs :: IO1 ()
countXs = cnt 0 where
  cnt x = do putStr' ("Found "++ show x ++ " crosses.\n")
             s<- getLine'
             if null s then return ()
                else cnt (x+ length (filter ('x' ==)
                                    (map toLower s)))
```

Zusammenfassung

- Ein/Ausgabe in Haskell durch `IO a`:
  - Berechungen vom `IO a` hängen von Umwelt ab.
  - Komposition von `IO a` durch

```
(>>=)  :: IO a-> (a-> IO b)-> IO b
return :: a-> IO a
```

  - Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: `getLine, putStr, putStrLn`
  - Module: `IO`, `Random`
- Implementation von `IO`:
  - Explizite Modellierung des Systemzustandes.

Vorlesung vom 07.01.2002

Inhalt

- Grafikprogrammierung mit HGL (Hugs Graphics Library)
- Einführung in die Schnittstelle, kleine Beispiele.
- Abstraktion über `HGL`:
  - Entwurf und Implementation einer kleine Geometriebücherei.
  - Kleines Beispiel.
- Literatur: Paul Hudak, The Haskell School of Expression.

Grafik — erste Schritte.

Das kanonische Beispielprogramm:

```
module Hello where
import GraphicsUtils
hello :: IO ()
hello = runGraphics (do
  w <- openWindow "Hallo Welt?" (300, 300)
  drawInWindow w (text(100, 100) "Hallo Welt!")
  drawInWindow w (ellipse (100,150) (200,250))
  getKey w
  closeWindow w)
```

- `runGraphics :: IO ()-> IO ()`
  führt Aktion mit Grafik aus;
- `openWindow :: Title-> Point-> IO Window`
  öffnet Fenster;
- `drawInWindow :: Window-> Graphic-> IO ()`
  zeichnet Grafik in Fenster;
- ADTs `Window` und `Graphic`:
  Fenster und darin darstellbare Grafiken;
- `getKey :: Window-> IO Char` wartet auf Taste
- `closeWindow :: Window-> IO ()` schließt Fenster

Die Hugs Graphics Library HGL

- Kompakte Grafikbücherei für einfache Grafiken und Animationen.
- Gleiche Schnittstelle zu X11 und Win32 Graphics Device.
- Bietet:
  - Fenster
  - verschiedene Zeichenfunktionen
  - Unterstützung für Animationen
- Bietet nicht:
  - Hochleistungsgrafik, 3D-Unterstützung (e.g. OpenGL)
  - GUI-Funktionalität

Übersicht HGL

- Grafik
  - Atomare Grafiken
  - Modifikatoren
  - Attribute
  - Kombination von Grafiken
  - Pinsel, Stifte und Texfarben
  - Farben
- Fenster
- Benutzereingaben: `Events`

- Atomare Grafiken
  - Größte Ellipse (gefüllt) innerhalb des gegeben Rechtecks

    ```
    ellipse :: Point -> Point -> Graphic
    ```
  - Größte Ellipse (gefüllt) innerhalb des gegeben Parallelograms:

    ```
    shearEllipse :: Point -> Point -> Point -> Graphic
    ```
  - Bogenabschnitt einer Ellipse im math. positiven Drehsinn:

    ```
    arc :: Point -> Point -> Angle -> Angle -> Graphic
    ```
  - Strecke, Streckenzug, Polygon (gefüllt), Text:

    ```
    line         :: Point -> Point -> Graphic
    polyline     :: [Point] -> Graphic
    polygon      :: [Point] -> Graphic
    text         :: Point -> String -> Graphic
    emptyGraphic :: Graphic
    ```

- Modifikation von Grafiken:
  - Darstellung mit anderen Fonts, Farben, Hintergrundfarben,. . . :

```
withFont          :: Font      -> Graphic -> Graphic
withTextColor     :: RGB       -> Graphic -> Graphic
withTextAlignment :: Alignment -> Graphic -> Graphic
withBkColor       :: RGB       -> Graphic -> Graphic
withBkMode        :: BkMode    -> Graphic -> Graphic
withPen           :: Pen       -> Graphic -> Graphic
withBrush         :: Brush     -> Graphic -> Graphic
withRGB           :: RGB       -> Graphic -> Graphic
```

- Modifikatoren sind kumulativ:

```
withFont courier (
  withTextColor red (
    withTextAlignment (Center, Top)
      (text (100, 100) "Hallo?")))
```

- Unschön — Schachtelung.
- Abhilfe: `($) :: (a-> b)-> a-> b` (rechtsassoziativ)

```
withFont courier $
withTextColor red $
withTextAlignment (Center, Top) $
text (100, 100) "Hallo?"
```

- Attribute
  - Konkrete Attribute (Implementation sichtbar)

```
type Angle     = Double
type Dimension = Int
type Point     = (Dimension,Dimension)
```

  - Farben: Rot, Grün, Blau

```
data RGB       = RGB Int Int Int
```

  - Textausrichtung, Hintergrundmodus

```
type Alignment = (HAlign, VAlign)
data HAlign = Left' | Center   | Right'
data VAlign = Top   | Baseline | Bottom
data BkMode = Opaque | Transparent
```

  - Abstrakte Attribute: `Font`, `Brush` und `Pen`

- `Brush` zum Füllen (Polygone, Ellipsen, Regionen)
  - Bestimmt nur durch Farbe

```
mkBrush :: RGB -> (Brush-> Graphic)-> Graphic
```

- `Pen` für Linien (Arcs, Linien, Streckenzüge)
  - Bestimmt durch Farbe, Stil, Breite.

```
data Style = Solid | Dash | Dot | DashDot | DashDotDot | .
mkPen :: Style -> Int-> RGB-> (Pen-> Graphic)-> Graphic
```

- Fonts (oh dear)
  - Punktgröße, Winkel (nicht unter X11), Fett, Kursiv, Name.
  - Portable Namen: `courier`, `helvetica`, `times`.

```
createFont :: Point -> Angle -> Bool -> Bool -> String ->
                  IO Font
```

- Farben
  - Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
           deriving (Eq, Ord, Bounded, Enum, Ix, Show, Rea
```

  - Dazu Modifikator:

```
withColor :: Color-> Graphic-> Graphic
```

  - Benannte Farben sind einfach nur `Array Color RGB`
- Kombination von Grafiken
  - Überlagerung (erste über zweiter):

```
overGraphic  ::  Graphic-> Graphic-> Graphic
overGraphics :: [Graphic]-> Graphic
overGraphics = foldr overGraphic emptyGraphic
```

Fenster

- Elementare Funktionen:

```
getGraphic :: Window -> IO Graphic
setGraphic :: Window -> Graphic-> IO ()
```

- Abgeleitetete Funktionen:
  - In Fenster zeichnen:

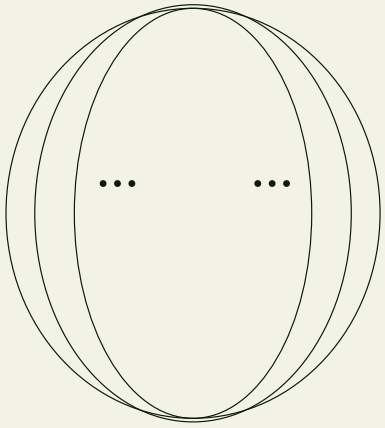
```
drawInWindow :: Window -> Graphic -> IO ()
drawInWindow w g = do
  old <- getGraphic w
  setGraphic w (g `overGraphic` old)
```

  - Grafik löschen

```
clearWindow :: Window -> IO ()
```

Ein einfaches Beispiel

- Ziel: einen gestreiften Ball zeichen.
- Algorithmus: als Folge von konzentrischen Ellipsen:
  - Start mit Eckpunkten (x_1, y_1) und (x_2, y_2).
  - Verringerung von x um Δ_x, y bleibt gleich.
  - Dabei Farbe verändern.



- Bälle zeichnen:

```
drawBalls :: Window-> Color->
                          Point-> Point-> IO ()
drawBalls w c (x1, y1) (x2, y2) =
  if x1 >= x2 then return ()
  else let el = ellipse (x1, y1) (x2, y2)
       in do drawInWindow w (withColor c el)
             drawBalls w (nextColor c)
                         (x1+deltaX, y1)
                         (x2-deltaX, y2)
deltaX :: Int
deltaX = 25
```

- Farbveränderung, zyklisch:

```
nextColor :: Color-> Color
nextColor Red   = Green
nextColor Green = Blue
nextColor Blue  = Red
```

  - Alternative: sich unendlich wiederholender Liste von Farben

```
cycle [Red, Green, Blue]
```

- Hauptprogramm:

```
main :: IO ()
main = runGraphics (do
    w <- openWindow "Balls!" (500,500)
    drawBalls w Blue (25, 25) (485, 485)
    getKey w
    closeWindow w)
```

Eine geometrische Bücherei

- Ziel: Abstrakte Repräsentation von geometrischen Figuren
- Komplementär zu HGL.
- Unterstützung von Translation, Skalierung, Rotation
- Basisfiguren: Rechtecke, Dreiecke, Polygone, Kreise
  - Alle Basisfiguren liegen im Nullpunkt
- Später: Unterstützung von Kollisionserkennung
- Keine Mengen, keine grafischen Attribute (Farben)

- Repräsentation durch algebraischen Datentyp

```
type Dimension = Int
data Figure = Rect Dimension Dimension
            | Triangle Dimension Angle Dimension
            | Polygon [Point]
            | Circle Dimension
            | Translate Point Figure
            | Scale Double Figure
            | Rotate Angle Figure
           deriving (Eq, Ord, Show)
```

- Beispiele:
  - `Rect 10 20` — Rechteck mit Höhe 10, Länge 20
  - `Rotate (pi/2) (Triangle 10 (pi/3) 10)`
    Gleichschenkliges Dreieck, auf dem Kopf stehend.
  - `Rotate pi (Circle 20)` — rotierter Kreis
- Beispiel für abgeleitete Funktionen:
  - Drehung um einen Punkt:

```
rotate :: Point-> Angle-> Figure-> Figure
rotate (px, py) w = Translate (px, py) . Rotate w .
                                    Translate (-px, -py)
```

- Darstellung einer geometrischen Figur:
  - Rotation, Translation und Skalierung herausrechnen.
  - Rechtecke und Dreiecke sind spezielle Formen von Polygonen.
  - Alles ist ein Polygon oder eine Ellipse.
  - Der Nullpunkt $(0, 0)$ des Koordinatensystems ist links unten.

- Mathematische Grundlagen: Vektorraum $\mathbb{R}^2$
  - Skalierung eines Punktes (x, y) um den Faktor f: Multiplikation des Vektorraumes.

$$(x', y') = (fx, fy)$$

```
smult :: Double-> Point-> Point
smult f (x, y)
  | f == 1    = (x, y)
  | otherwise = (round (f* fromInt x),
                 round (f* fromInt y))
```

  - Translation eines Punktes (x, y) um einen Vektor (a, b): Addition des Vektorraumes.

$$(x', y') = (x + a, y + b)$$

```
add :: Point-> Point-> Point
add (x1, y1) (x2, y2) = (x1+ x2, y1+ y2)
```

- Rotation eines Punktes (x, y) um den Winkel ϕ:

$$(x', y') = (x \cos\phi + y \sin\phi, -x \sin\phi + y \cos\phi)$$

```
rot :: Angle-> Point-> Point
rot w (x, y)
  | w == 0    = (x, y)
  | otherwise = (round (x'* cos w+ y'* sin w),
                 round (-x' * sin w + y'* cos w)) where
                x' = fromInt x; y'= fromInt y
```

- Damit Zeichenfunktion:

```
draw :: Figure-> Graphic
draw = draw' ((0, 0), 1, 0) where
```

  - Hauptarbeit findet in `draw'` statt.
  - Argumente: kumulierte Translation, Skalierung, Rotation

```
draw' :: (Point, Double, Angle)-> Figure-> Graphic
```

  - Translation, Skalierung, Rotation aufrechnen:

```
draw' (m, r, phi) (Translate t f) =
  draw' (add m t, r, phi) f
draw' (m, r, phi) (Scale s f) =
  draw' (s*m, s+ r, phi) f
draw' (m, r, phi) (Rotate w f) =
  draw' (rot w m, r, phi+ w) f
```

- Basisfiguren zeichnen:

```
draw' ((mx, my), r, _) (Circle d) =
  ellipse (mx- rad, my-rad) (mx+ rad, my+rad) where
    rad= round (r*fromInt d / 2)
draw' c (Rect a b) =
  poly c [(x2, y2), (-x2, y2),
          (-x2, -y2), (x2, -y2)] where
    x2= a `div` 2; y2= b `div` 2
draw' c (Triangle l1 a l2) =
  poly c [(0, 0), (0, l1), rot a (0, l2)]
draw' c (Polygon pts) = poly c pts
```

- Hilfsfunktion: Polygon zeichnen

```
poly :: (Point, Double, Angle)-> [Point]-> Graphic
poly (m, p, w) = polygon . map (add m. smult p. rot w)
```

Ein kleines Beispielprogramm

- Sequenz von gedrehten, skalierten Figuren.
- Programmstruktur:
  - `drawFigs :: [Figure]-> IO` zeichnet Liste von Figuren in wechselnden Farben
  - `swirl: Figure-> [Figure]` erzeugt aus Basisfigur unendliche Liste von gedrehten, vergrösserten Figuren

- Figuren zeichnen:
  -
```
drawFigs :: [Figure]-> IO ()
drawFigs f =
  runGraphics (do
    w<- openWindow "Here's some figures!" (500, 500)
    drawInWindow w (overGraphics (zipWith withColor
              (cycle [Blue ..])
              (map (draw. Translate (250, 250)) f)))
    getKey w
    closeWindow w)
```
  - `(cycle [Blue ..]` erzeugt unendliche Liste aller Farben.
  - NB: Figuren werden im Fenstermittelpunkt gezeichnet.

- Figuren erzeugen:

```
swirl :: Figure-> [Figure]
swirl = iterate (Scale 1.123. Rotate (pi/17))
```

  - Nutzt `iterate :: (a-> a)-> a-> [a]` aus dem Prelude:

```
iterate f x = [x, f x, f f x, f f f x, ...]
```

- Hauptprogramm:

```
main = do
--  drawFigs (take 300 (swirl
--                        (Triangle 6 (pi/3) 6)))
--  drawFigs (take 300 (swirl (Rect 4 3)))
```

Zusammenfassung

- Die Hugs Graphics Library (HGL) bietet abstrakte Graphikprogrammierung für Hugs.
  - Handbuch und Bezugsquellen auf PI3-Webseite oder `http://www.haskell.org/graphics`
- Darauf aufbauend Entwicklung einer kleinen Geometriebücherei.
- Nächste Woche: Kollisionserkennung und Animation.

Vorlesung vom 14.01.2002

Inhalt

- Ein Fehler!
- Bewegte Grafiken: Animationen
- *Labelled Records*
- Space — the final frontier

Das Problem

- Letzte VL:
  - Definition von `draw' :: (Point, Double, Angle)-> Figure-> Graphic`
  - Skalierung, Rotation und Translation nicht unabhängig.
  - Müssen Interaktion berücksichtigen — aber wie?
- Grundlagen der linearen Algebra:
  - Skalierung mit dem Faktor F: Multiplikation des Vektorraumes
  - Translation um Punkt $\vec{t}$: Addition des Vektorraumes
  - Rotation um den Winkel ω: Multiplikation mit der Rotationsmatrix

$$M_\omega = \begin{pmatrix} \cos\omega & \sin\omega \\ -\sin\omega & \cos\omega \end{pmatrix}$$

- Es gelten folgende Gleichungen:

$$(\vec{p}+\vec{q})M_\omega = \vec{p}M_\omega + \vec{q}M_\omega \quad (1)$$

$$r \cdot (\vec{p}+\vec{q}) = r \cdot \vec{p} + r \cdot \vec{q} \quad (2)$$

$$(r \cdot \vec{p})M_\omega = r \cdot (\vec{p}M_\omega) \quad (3)$$

- Implementation von `draw`:
  - (1) – (3) erlauben Reduktion zu einer Normalform

$$E(\vec{p}) = \vec{t} + s \cdot (\vec{p}M_\omega)$$

    - ▷ Zuerst Rotation um Vektor ω
    - ▷ Dann Skalierung um Faktor s
    - ▷ Dann Translation um Vektor t

- Berichtigter Code:

```
draw' (m, r, phi) (Translate t f) =
  draw' (add m (smult r (rot phi t)), r, phi) f
draw' (m, r, phi) (Scale s f) =
  draw' (m, s+ r, phi) f
draw' (m, r, phi) (Rotate w f) =
  draw' (m, r, phi+ w) f
```

Kollisionserkennung

- Ziel: Interaktion der Figuren erkennen
  - . . . zum Beispiel Kollision
  - Eine Möglichkeit: `Region` (aus HGL)
  - Elementare Regionen:

```
emptyRegion   :: Region
polygonRegion :: [Point]-> Region
```

  - Vereinigung, Schnitt, Subtraktion:

```
unionRegion     :: Region-> Region-> Region
intersectRegion :: Region-> Region-> Region
```

  - aber leider nur Funktion nach `Graphic`, nicht `isEmpty :: Region-> Bool` oder `contains :: Region-> Point-> Bool`

- Deshalb: eigene Implementation
  - Idee: zur Darstellung wird Normalform berechnet
  - Normalform auch zur Kollisionserkennung nutzen
- Normalform: Elementare Figuren (Formen; `Shape`)
  - einfach zu zeichnen
  - einfache Kollisionserkennung
- Konservative Erweiterung des `Geometry`-Moduls
  - Alte Schnittstelle bleibt erhalten.

- Datentyp und Funktionen

```
data Shape = Poly [Point]
           | Circ Point Double
           deriving Show
```

  - Konversion von Figur in Form:

```
shape :: Figure-> Shape
```

  - Form zeichnen:

```
drawShape :: Shape-> Graphic
```

  - Kollisionserkennung:

```
contains  :: Shape-> Point-> Bool
intersect :: Shape-> Shape-> Bool
```

- Konversion Figur nach Form ist altbekannt:
  -

```
shape :: Figure-> Shape
shape = fig' ((0, 0), 1, 0) where
  fig' :: (Point, Double, Angle)-> Figure-> Shape
  fig' (m, r, phi) (Translate t f) =
    fig' (add m (smult r (rot phi t)), r, phi) f
  fig' (m, r, phi) (Scale s f) =
    fig' (m, r* s, phi) f
  fig' (m, r, phi) (Rotate w f) =
    fig' (m, r, phi+ w) f
  fig' c (Rect a b) =
    poly c [(x2, y2), (-x2, y2),
            (-x2, -y2), (x2, -y2)] where
```

```
      x2= a `div` 2; y2= b `div` 2
fig' c (Triangle l1 a l2) =
  poly c [(0, 0), (0, l1), rot a (0, l2)]
fig' c (Polygon pts) = poly c pts
fig' (m, r, _) (Circle d) =
  Circ m (r*fromInt d)
poly :: (Point, Double, Angle)-> [Point]-> Shape
poly (m, p, w) = Poly. chckcls.
                      map (add m. smult p. rot w) where
```

- Prüfung ob Polygon geschlossen (für Kollisionserkennung)

```
chckcls [] = []
chckcls x  = if (head x) == (last x)
                  then x else x++ [head x]
```

- Form zeichnen ist trivial:
  - ```
    drawShape :: Shape-> Graphic
    drawShape (Poly pts) = polygon pts
    drawShape (Circ (mx, my) r) =
        ellipse (mx-r', my- r') (mx+ r', my+ r') where
          r'= round r
    ```
  - Alte Funktion `draw :: Figure-> Graphic` ist `drawShape . shape`.
    ```

- Kolliosionserkennung für Punkte:
  - Auch weiter unten benötigt, deshalb separate Funktionen `inP`, `inC`

```
contains :: Shape-> Point-> Bool
contains (Poly pts)= inP pts
contains (Circ c r)= inC c r
```

  - Punkt ist innerhalb des Kreises gdw. Abstand zum Mittelpunkt kleiner (gleich) Radius

```
inC :: Point-> Double-> Point-> Bool
inC (mx, my) r (px, py) = len (px- mx, py- my) <= r
```

    Abstand ist Länge (Betrag) des Differenzvektors:

```
len :: Point-> Double
len (x, y)= sqrt (fromInt (x^2+ y^2))
```

- Punkt ist innerhalb des Polygons, wenn gleiche Orientierung bezgl. allen Seiten:
  - ▷ Polygon ist konvex
  - ▷ Punkte auf den Seiten werden nicht erkannt
  - ▷ Hilfsfunktion: Determinante berechnen

```
det :: Point-> (Point, Point)-> Int
det (cx,cy) ((ax,ay), (bx,by)) =
  signum ((by-ay)*(cx-bx)-(cy-by)*(bx-ax))
```

  - ▷ Hilfsfunktion: Liste der Seiten eines Polygons (Voraussetzung: geschlossen)

```
sides :: [Point]-> [(Point, Point)]
sides ps | length ps < 2 = []
         | otherwise     = (head ps, head (tail ps)):
                                    sides (tail ps)
```

- Damit Hauptfunktion:
  - ▷ aus Polygon Liste der Seiten bilden,
  - ▷ Determinante aller Seiten bilden,
  - ▷ doppelte Vorkommen löschen;
  - ▷ Ergebnisliste hat Länge 1 gdw. gleiche Orientierung für alle Seiten

```
inP :: [Point]-> Point-> Bool
inP ps c = (length. nub. map (det c). sides) ps == 1
```

- Ineffizient — `length` berechnet immer Länge der ganzen Liste.

- Damit Schnitt von Formen einfach:
  - Schnitt Kreis - Polygon erkennt Randfall nicht: Kreisbogen schneidet nur Seite
  - Zwei Kreis schneiden sich gdw. Distanz der Mittelpunkte kleiner als Summe der Radien

```
intersect :: Shape-> Shape-> Bool
intersect (Poly p) (Circ c r)=
   inP p c || any (inC c r) p
intersect (Circ c r) (Poly p)=
   inP p c || any (inC c r) p
intersect (Poly p1) (Poly p2)=
   any (inP p1) p2 || any (inP p2) p1
intersect (Circ (mx1, my1) r1) (Circ (mx2, my2) r2)=
   len (mx2- mx1, my2- my1) <= r1+ r2
```

- Noch eine Hilfsfunktion:
  - Polarkoordinaten $P = (r, \phi)$
  - Konversion in kartesische Koordinaten:

    Punkt $(r, 0)$ um Winkel ϕ drehen.

```
polar :: Double-> Angle-> Point
polar r phi = rot phi (round r, 0)
```

Benutzereingaben: `Events`

- Benutzereingabe:
  - Tasten
  - Mausbewegung
  - Mausknöpfe
  - Fenster: Größe verändern, schließen
- Grundliegende Funktionen:
  - Letzte Eingabe, auf nächste Eingabe warten:

    ```
    getWindowEvent :: Window -> IO Event
    ```

  - Letzte Eingabe, nicht warten:

    ```
    maybeGetWindowEvent :: Window-> IO (Maybe Event)
    ```

- `Event` ist ein labelled record:

```
data Event
  = Char      { char :: Char }
  | Key       { keysym :: Key, isDown :: Bool }
  | Button    { pt :: Point,
                isLeft, isDown :: Bool }
  | MouseMove { pt :: Point }
  | Resize
  | Closed
 deriving Show
```

- `Event` ist ein labelled record:

```
data Event
  = Char      { char :: Char }
  | Key       { keysym :: Key, isDown :: Bool }
  | Button    { pt :: Point,
                isLeft, isDown :: Bool }
  | MouseMove { pt :: Point }
  | Resize
  | Closed
 deriving Show
```

- Was ist das ?!?

Probleme mit großen Datentypen

- Beispiel Warenverwaltung
  - Ware mit Bezeichung, Stückzahl, Preis (in Cent)

```
data Item = Item String Int Int
```

  - Kommt Stückzahl oder Preis zuerst?
- Beispiel Buch:
  - Titel, Autor, Verlag, Signatur, Fachgebiet, Stichworte

```
data Book' = Book' String String String String String
```

  - Kommt Titel oder Autor zuerst?
    Ist Verlag das dritte oder vierte Argument?

Probleme mit großen Datentypen

- Reihenfolge der Konstruktoren.
  - Typsynonyme (`type Author= String`) helfen nicht
  - neue Typen (`data Author = Author String`) zu umständlich
- Selektion und Update
  - Für jedes Feld einzeln zu definieren.

```
getSign :: Book'-> String
getSign (Book' _ _ _ s _) = s
setSign :: Book'-> String-> Book'
setSign (Book' t a p _ f) s = Book' t a p s f
```

- Inflexibilität
  - Wenn neues Feld hinzugefügt wird, alle Konstruktoren ändern.

Lösung: *labelled records*

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

```
data Book = Book { author :: String,
                   title  :: String,
                   publisher :: String }
```

- Konstruktion:

```
b = Book
  {author = "M. Proust",
   title  = "A la recherche du temps perdu",
   publisher = "S. Fischer Verlag"}
```

- Selektion durch Feldnamen:

```
publisher b --> "S. Fischer Verlag"
author b    --> "M. Proust"
```

- Update:

```
b{publisher = "Rowohlt Verlag"}
```

  - Rein funktional! (`b` bleibt unverändert)

- Patternmatching:

```
print :: Book-> IO ()
print (Book{author= a, publisher= p, title= t}) =
  putStrLn (a++ " schrieb "++ t ++ " und "++
            p++ " veröffentlichte es.")
```

- Partielle Konstruktion und Patternmatching möglich:

```
b2 = Book {author= "C. Lüth"}
shortPrint :: Book-> IO ()
shortPrint (Book{title= t, author= a}) =
  putStrLn (a++ " schrieb "++ t)
```

  - Guter Stil: nur auf benötigte Felder matchen.

- Datentyp erweiterbar:

```
data Book = Book {author :: String,
                  title  :: String,
                  publisher :: String,
                  signature :: String }
```

  Programm muß nicht geändert werden (nur neu übersetzt).

Zusammenfassung labelled records

- Reihenfolge der Konstruktorargumente irrelevant
- Generierte Selektoren und Update-Funktionen
- Erhöht Programmlesbarkeit und Flexibilität

Animation

Alles dreht sich, alles bewegt sich. . .

- Animation: über der Zeit veränderliche Grafik
- Unterstützung von Animationen in HGL:
  - `Timer` ermöglichen getaktete Darstellung
  - Gepufferte Darstellung ermöglicht flickerfreie Darstellung
- Öffnen eines Fensters mit Animationsunterstützung:
  - Initiale Position, Grafikzwischenpuffer, Timer-Takt in Millisekunden

```
openWindowEx :: Title-> Maybe Point-> Size->
                RedrawMode-> Maybe Time-> IO Window
data RedrawMode
  = Unbuffered | DoubleBuffered
```

Eine einfache Animation

- Ein springender Ball:
  - Ball hat Position und Geschwindigkeit:

```
data Ball = Ball { p :: Point,
                   v :: Point }
```

  - Ball zeichnen: Roter Kreis an Position $\vec{p}$

```
drawBall :: Ball-> Graphic
drawBall (Ball {p= p}) =
  withColor Red
    (draw (Translate p (Circle 20)))
```

- Ball bewegen:
  - Geschwindigkeit $\vec{v}$ zu Position $\vec{p}$ addieren
  - In X-Richtung: modulo Fenstergröße 500
  - In Y-Richtung: wenn Fensterrand 500 erreicht, Geschwindigkeit invertieren
  - Geschwindigkeit in Y-Richtung nimmt immer um 1 ab

```
move :: Ball-> Ball
move (Ball {p= (px, py), v= (vx, vy)})=
  Ball {p= (px', py'), v= (vx, vy')} where
    px' = (px+ vx) `mod` 500
    py0 = py+ vy
    py' = if py0> 500 then 500-(py0-500) else py0
    vy' = (if py0> 500 then -vy else vy)+ 1
```

- Hauptprogramm:
  - Fenster öffnen

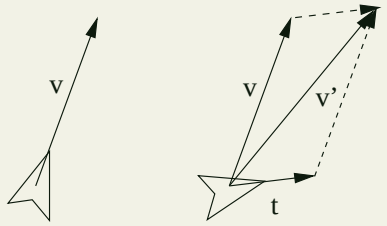
```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Bounce!"
                      Nothing (500, 500) DoubleBuffered
                      (Just 30)
     loop w (Ball{p=(0, 10), v= (5, 0)}) where
```

  - Hauptschleife: Ball zeichnen, auf Tick warten, Folgeposition berechnen

```
        loop :: Window-> Ball-> IO ()
        loop w b =
          do setGraphic w (drawBall b)
             getWindowTick w
             loop w (move b)
```

Ein Raumflugsimulator

- Ziel: Simulation eines Raumschiffs
- Steuerung nur mit Schub und Drehung



- Geschwindigkeit $\vec{v}$, Schub $\vec{t}$

- Zustand des Schiffes zu gegebener Zeit:
  - Position $\vec{p} \in \mathbb{R}^2$
  - Geschwindigkeit $\vec{v} \in \mathbb{R}^2$
  - Orientierung $\phi \in \mathbb{R}$ (als Winkel)
  - Schub $s \in \mathbb{R}$ (als Betrag; $\vec{t}$ ist $polar(s, \phi)$)
  - Winkelbeschleunigung $\omega \in \dot{\mathbb{R}}$
- Zustandsübergang:
  - Neue Position: $\vec{p}' = \vec{p} + \vec{v}'$
  - Neue Geschwindigkeit: $\vec{v}' = v + polar(s, \phi)$
  - Neue Orientierung: $\phi' = \phi + \omega$
  - Winkelbeschleunigung und Schub: durch Benutzerinteraktion

- Benutzerinteraktion:
  - Konstanten W für Winkelbeschleunigung, T für Schub
  - Tasten für $Links$, $Rechts$, $Schub$
  - $Links$ drücken: Winkelbeschleunigung auf $+W$ setzen
  - $Links$ loslassen: Winkelbeschleunigung auf 0 setzen
  - $Rechts$ drücken: Winkelbeschleunigung auf $-W$ setzen
  - $Rechts$ loslassen: Winkelbeschleunigung auf 0 setzen
  - $Schub$ drücken: Schub auf T setzen
  - $Schub$ loslassen: Schub auf 0 setzen
- Neuer Zustand:
  Zustandsübergang plus Benutzerinteraktion.

- Modellierung des Gesamtsystems
  - Für den Anfang nur das Schiff:

```
data State = State { ship  :: Ship }
```

  - Schiffszustand:
    - ▷ `Shape` merken für effiziente Kollisionserkennung!

```
data Ship =
   Ship { pos    :: Point,
          shp    :: Shape,
          vel    :: Point,
          ornt   :: Double,
          thrust :: Double,
          hAcc   :: Double }
```

- Globale Konstanten
  - Das Raumschiff

```
spaceShip :: Figure
spaceShip = Polygon [(15, 0), (-15, 10),
                     (-10, 0), (-15, -10), (15, 0)]
```

  - Fenstergröße

```
winSize :: (Int, Int)
winSize = (1000, 800)
```

  - Schub

```
aDelta :: Double
aDelta = 1
```

  - Maximale Geschwindigkeit

```
vMax :: Double
```

```
vMax = 20
```

- Winkelbeschleunigung

```
hDelta :: Double
hDelta = 0.3
```

- Der Anfangszustand: Schiff in Bildschirmitte, nach Norden gerichtet

```
initialState :: State
initialState =
  State {ship= setShp $
     Ship{pos= (fst winSize `div` 2,
                snd winSize `div` 2),
          vel= (0, 0), ornt= pi/2,
          thrust= 0, hAcc= 0}}
```

- Neuen Schiffszustand berechnen
  - Geschwindigkeit so verändern, dass Betrag Obergrenze v_{max} nie überschreitet.

```
moveShip :: Ship-> Ship
moveShip(Ship {pos= pos0, vel= vel0,
               hAcc= hAcc, thrust= t, ornt= o}) =
  setShp $
    Ship{pos= addWinMod pos0 vel1,
         vel= if l> vMax then smult (vMax/l) vel1
                         else vel1,
         thrust= t, ornt= o+ hAcc, hAcc= hAcc} where
           vel1= add (polar t o) vel0
           l   = len vel1
```

- `Shape` berechnen und setzen:
  - `spaceShip` ist das Aussehen des Schiffes (globale Konstante)
  - Um Orientierung drehen und an Position verschieben.

```
setShp :: Ship-> Ship
setShp s = s{shp= shape (Translate (pos s)
                        (Rotate (ornt s) spaceShip))}
```

  - Vektoraddition modulo Fenstergröße:

```
addWinMod :: (Int,Int) -> (Int,Int) -> (Int,Int)
addWinMod (a, b) (c, d)=
  ((a+ c) `mod` (fst winSize),
   (b+ d) `mod` (snd winSize))
```

- Systemzustand darstellen
  - Gesamter Systemszustand

```
drawState :: State-> Graphic
drawState s = drawShip (ship s)
```

    ▷ Weitere Objekte mit `overGraphics` kombinieren

  - Schiff darstellen (Farbänderung bei Beschleunigung)

```
drawShip :: Ship-> Graphic
drawShip s =
   withColor (if thrust s> 0 then Red else Blue)
             (drawShape (shp s))
```

- Neuen Systemzustand berechnen:
  - Hauptschleife: zeichnen, auf nächsten Tick warten, Benutzereingabe lesen, Folgezustand berechnen

```
loop :: Window-> State-> IO ()
loop w s =
  do setGraphic w (drawState s)
     getWindowTick w
     evs<- getEvs
     s<- nextState evs s
     loop w s where
```

- Folgezustand berechnen: später IO möglich (Zufallszahlen!)

```
nextState :: [Event]-> State-> IO State
nextState evs s =
  do return s1{ship= moveShip (ship s1)} where
     s1= foldl (flip procEv) s evs
```

- Liste aller Eingaben seit dem letzten Tick:

```
getEvs :: IO [Event]
getEvs = do x<- maybeGetWindowEvent w
            case x of
                Nothing -> return []
                Just e  -> do rest <- getEvs
                              return (e : rest)
```

- Eine Eingabe bearbeiten:

```
procEv :: Event-> State-> State
procEv (Key {keysym= k, isDown=down})
 | isLeftKey k && down       = sethAcc hDelta
 | isLeftKey k && not down   = sethAcc 0
 | isRightKey k && down      = sethAcc (- hDelta)
 | isRightKey k && not down = sethAcc 0
 | isUpKey k && down         = setThrust aDelta
 | isUpKey k && not down     = setThrust 0
procEv _ = id
sethAcc :: Double->State-> State
sethAcc a s = s{ship= (ship s){hAcc= a}}
setThrust :: Double-> State-> State
setThrust a s = s{ship= (ship s){thrust= a}}
```

- Das Hauptprogramm
  - Fenster öffnen, Schleife mit Anfangszustand starten

```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Space --- The Final Frontier"
                      Nothing winSize DoubleBuffered
                      (Just 30)
     loop w initialState
     closeWindow w
```

Zusammenfassung

- Konservative Erweiterung von `Geometry` um Kollisionserkennung
- Neues Haskell-Konstrukt: labelled records
  - Reihenfolge der Konstruktorargumente irrelevant
  - Generierte Selektoren und Update-Funktionen
  - Erhöht Programmlesbarkeit und Flexibilität
- Animation:
  - Unterstützung in `HGL` durch Timer und Zeichenpuffer
  - Implementation eines einfachen Raumschiffsimulators

Vorlesung vom 21.01.2002

Inhalt

- Animationen:
  - Schwächen des bisherigen Ansatzes
  - Ein abstrakterer Ansatz
  - Der Datentyp `Behaviour`

Animation Abstrakt

- Letzte Woche:
  - bouncy ball
  - Raumflugsimulator
- Grundprinzip:

```
loop :: Window-> State-> IO ()
loop w s = do setGraphics w (drawState s)
              getWindowTick w
              i<- getEvents
              s<- nextState s i
              loop s
```

- Explizite Eventloop:
  - Zustand anzeigen, Eingabe verarbeiten, neuen Zustand anzeigen

- Problem:
  - Umständlich — Eventloop wird immer von vorne geschrieben
  - Unhandlich — Zusammenhängender Code über Programm verteilt
  - Nicht kompositional — wie kombiniert man verschieden Eventloops?
  - Keine einfache Wiederverwendung möglich
- Lösung: abstrakterer Ansatz
  - Animation : über der Zeit veränderliche Grafik

    ```
    type Time = Double
    type Animation a = Time-> a
    ```

  - Nicht `type Animation = Time -> Graphic`.

Drei einfaches Beispiele

- Ein Pulsierender Ball

```
pulsatingBall :: Animation Figure
pulsatingBall t =  Circle (round (190* sin t))
```

Drei einfaches Beispiele

- Ein Pulsierender Ball

```
pulsatingBall :: Animation Figure
pulsatingBall t =  Circle (round (190* sin t))
```

- Ein Rotierender Ball

```
revolvingBall :: Animation Figure
revolvingBall t = Translate (round (210* cos t),
                             round (210* sin t))
                            (Circle 20)
```

- Kombination: Eine Art Sonnensystem

```
planets :: Animation Graphic
planets t =
  draw (pulsatingBall t) `overGraphic`
    draw (revolvingBall t)
```

Implementation

- Ziel: Funktion

  ```
  animate :: String-> Animation Graphic-> IO ()
  ```

  - Fenster öffnen (`openWindowEx`)
    - ▷ Getaktet und gepuffert
  - Zeit lesen

    ```
    type Time = Integer -- Systemzeit in Millisekunden
    getTime :: IO Time
    ```

  - Graphik generieren und ausgeben
  - Auf Tick warten, wieder von vorne anfangen

```
animate t anim = runGraphics $
  do w <- openWindowEx t Nothing (500, 500)
              DoubleBuffered (Just 30)
     t0 <- getTime
     loop w t0 where
       loop w t0 = do t<- getTime
                      let now = fromInteger(t- t0)/1000
                      setGraphic w (anim now)
                      getWindowTick w
                      loop w t0
```

- NB: `let` — wie `where`, nur vorgestellt

- Animierte Figuren: in der Mitte zeichnen

```
drawCentered :: Figure -> Graphic
drawCentered = draw . Translate (250, 250)

animFig :: String-> Animation Figure -> IO ()
animFig t a = animate t (drawCentered. a)
```

- Das Sonnensystem (reprise)

```
planets' :: Animation Graphic
planets' t =
  drawCentered (pulsatingBall t) `overGraphic`
    drawCentered (revolvingBall t)
```

Funktionen auf Animationen

- Überlagerung zweier Animationen

```
overAnim :: Animation Graphic->
            Animation Graphic->
              Animation Graphic
overAnim a1 a2 t = a1 t `overGraphic` a2 t
```

- Oder auch:

```
emptyAnim :: Animation Graphic
emptyAnim t = emptyGraphic
```

- Generelles Muster:
  - Gegeben: `f :: t1 ... tn -> t`
  - Gesucht: `f' :: Animation t1 ... Animation tn -> Animaton t`
- Mathematisches Prinzip:
  - Geschlossene Struktur (gegenüber dem Funktionenraum)
  - Punktweise Definition von Funktionen
  - Beispiele: für $f, g : X \to Y$

$$f = g \iff \forall x \in X. fx = gx$$
$$f \leq g \iff \forall x \in X. fx \leq gx$$

- Lifting von Operationen auf `t` zu `Animation t`
- Gleicher Name für `f` und `f'` mit Typklassen

Verhalten: `Behaviour`

- Neuer Typname: Behaviour
  - Abstrakte Version von `Animation`
  - Kann allgemeiner sein als `Time -> a`
  - Erste Näherung:

    ```
    data Beh a = Beh (Time-> a)
    ```
- Nutzung existierender Funktionen durch Instanziierung
- Einfachstes Verhalten: die Zeit

  ```
  time :: Beh Time
  time = Beh id
  ```

- Lifting — für jedes `n`:

```
lift0 :: a-> Beh a
lift0 x = (Beh (const x))

lift1 :: (a-> b)-> Beh a-> Beh b
lift1 f (Beh b1) = Beh (f. b1)

lift2 :: (a-> b-> c)-> Beh a-> Beh b-> Beh c
lift2 f (Beh b1) (Beh b2) =
                    Beh (\t-> f (b1 t) (b2 t))
lift3 :: (a-> b-> c-> d)->
                  Beh a-> Beh b-> Beh c-> Beh d
```

Exkurs: Haskell's Standard-Typklassen

- `Num` (Konstanten und Basisoperationen): `0`, `(+)`, `(*)`
- `Real` (Reele Zahlen): `toRational`
- `Fractional` (Brüche): `/`
- `Integral` (Restgruppen): `div`, `mod`
- `RealFrac` (Reele Zahlen als Brüche): `round`, `floor`
- `Floating` (Fließkommaoperationen): `pi`, `sin`
- `RealFloat` (Fließkommazahlen): `isNaN`, `isInfinite`

Typklassen

- „Dummy-Instanzen“ für `Eq`, `Show`
- Numerische Klassen:

```
instance Num a=> Num (Beh a) where
  (+)    = lift2 (+)
  (*)    = lift2 (*)
  negate = lift1 negate
  abs    = lift1 abs
  fromInteger = lift0. fromInteger
```

- Ferner `Fractional a` und `Floating a`
- Damit `time+ 10`, `time* 2`, `sin time` ...

Animierte Figuren

- Basisfiguren:

```
circle :: Beh Int-> Beh Figure
circle = lift1 Circle
poly   :: Beh [Point] -> Beh Figure
rect   :: Beh Int-> Beh Int-> Beh Figure
```

- Transformationen:

```
rotate    :: Beh Double-> Beh Figure-> Beh Figure
scale     :: Beh Double-> Beh Figure-> Beh Figure
translate :: Beh Point-> Beh Figure-> Beh Figure
```

- Wichtig: Argumente alle zeitabhängig (`Beh a`!)
- Behaviours animieren:
  - `anim :: Title-> Beh Graphic-> IO ()`

- `animF :: Title-> Beh Figure-> IO ()`
- Nutzen `animate`, `animFig` von oben

Einfache Beispiele

- Eine Figur im Kreis rotieren lassen:

```
goRound :: Beh Int-> Beh Figure-> Beh Figure
goRound r = rotate time. translate (pairB (r, 0))
```

Einfache Beispiele

- Eine Figur im Kreis rotieren lassen:

```
goRound :: Beh Int-> Beh Figure-> Beh Figure
goRound r = rotate time. translate (pairB (r, 0))
```

- Ein Planet:

```
planet :: Beh Figure
planet = goRound 150 (circle 20)
```

- Ein schwankender Planet:

```
dizzy :: Beh Figure
dizzy = goRound (round' (100*sin (2.7*time)+ 100)) (circle 2
```

- Ein schwankender Planet:

```
dizzy :: Beh Figure
dizzy = goRound (round' (100*sin (2.7*time)+ 100)) (circle 2
```

- Ein Mond:

```
moon :: Beh Figure
moon = goRound 150 (goRound 40 (circle 10))
```

Weitere Hilfsfunktionen

- Figur zeichnen:

```
drawB :: Beh Figure-> Beh Graphic
drawB = lift1 drawCentered
```

- Grafiken überlagern:

```
overB :: Beh [Graphic]-> Beh Graphic
overB = lift1 overGraphics

over :: Beh Graphic-> Beh Graphic-> Beh Graphic
over = lift2 overGraphic
```

- Beispiel:

```
solar :: Beh Graphic
solar = drawB planet `over` drawB moon
```

- Runden:

```
round' :: Beh Double-> Beh Int
round' = lift1 round
```

- Tupelbildung

```
pairB :: (Beh a, Beh b)-> Beh (a, b)
pairB (Beh a, Beh b)= Beh (\t-> (a t, b t))
```

- Runden:

```
round' :: Beh Double-> Beh Int
round' = lift1 round
```

- Tupelbildung

```
pairB :: (Beh a, Beh b)-> Beh (a, b)
pairB (Beh a, Beh b)= Beh (\t-> (a t, b t))
```

- Map und sequencing:

```
mapB :: (a-> b)-> Beh [a]-> Beh [b]
mapB f = lift1 (map f)

seqB :: [Beh a]-> Beh [a]
seqB bs = Beh (\t-> map (\(Beh b) -> b t) bs)
```

Ein etwas längeres Beispiel

- Eine Art Kaleidoskopgenerator: Spiegelung + Verfärbung
- Spiegelung:

```
star :: Beh Figure-> Beh [Figure]
star bf = seqB (zipWith rotate
                  (map lift0 [0,pi/4..2*pi])
                  (repeat bf))
```

- Verfärbung:

```
drawCol :: [RGB]-> Beh [Figure]-> Beh [Graphic]
drawCol cols =
       lift1 (zipWith (\c-> withRGB c.
                              drawCentered)
                      cols)
colour :: Beh [Figure]-> Beh [Graphic]
colour =  drawCol allcols

allcols :: [RGB]
allcols = cycle [r | (c, r) <- colorList,
                     c /= Black]
```

- Beides kombiniert:

```
colourStar :: Beh Figure -> Beh Graphic
colourStar = overB . colour . star
```

Zeittransformationen

- Veränderung der Zeit:

$$\texttt{Beh Time} \cong \texttt{Time -> Time}$$

```
timeTrans :: Beh Time-> Beh a-> Beh a
timeTrans (Beh f) (Beh a) = Beh (\t-> a (f t))
```

- Beispiel: Animation doppelt so schnell laufen lassen

```
hasten :: Beh a-> Beh a
hasten = timeTrans (time*2)
```

- Beispiel: nichtlineare Zeit

```
strange :: Beh a-> Beh a
strange = timeTrans (time* (0.5+ sin time))
```

- Das Kaleidoskop, mit sich selbst invertiert

```
doubleStar :: Beh Figure-> Beh Graphic
doubleStar b = over (colourStar b)
                    (timeTrans (- time)
                               (colourStar b))
```

- Noch ein Beispiel: eine eckige Figur

```
clunky :: Beh Figure
clunky = goRound 150
                 (scale (sin time*6+ 2)
                 (poly (lift0 [(0, 40), (10, -10),
                               (-10, -20)])))
```

Beispiel: Animation verwischen

- Ziel: Verblaßte Kopien hinter der Animation
- Dazu Hilfsfunktion: zeitverzögerte Kopie erstellen

```
iterateB :: (Beh a-> Beh a)-> Beh a-> Beh [a]
iterateB f x = seqB (iterate f x)

copy :: Beh Time-> Beh Figure-> Beh [Figure]
copy delta =
  iterateB (timeTrans (time- delta))
```

- Hauptfunktion:

```
blurr :: Beh Figure-> RGB-> Beh Graphic
blurr f col =
  overB (lift1 (take 10)
         (drawCol cols (copy 0.4 f))) where
  cols :: [RGB]
  cols = iterate (\ (RGB r g b)->
               RGB (dec r) (dec g) (dec b)) col
                                        where
         dec r = min r (r-15)
```

Zusammenfassung

- Abstrakter Ansatz zu Animationen ermöglicht
  - Kompositionalität
  - durch konsequentes Lifting sehr einfache Programme
  - aber: stößt an die Grenzen des Haskell-Typsystems

Vorlesung vom 28.01.2002

Inhalt

- Letze Vorlesung: Animation
- Diese Vorlesung: Modellierung von Benutzereingaben
- Diskrete Ereignisse, kontinuierliches Verhalten.
- Führt zu reaktiver Programmierung
  - s. Hudak, Kapt. 15 ff;
- Beispiel für eine domänenspezifische Sprache (DSL).
  - Hier in Haskell eingebettet.
  - Wegen Haskell's Flexibilität gut möglich.

Modellierung von Benutzereingaben

- Verhalten ist kontinuierlich:

  ```
  data Beh a = Beh (Time-> a)
  ```

- Benutzereingaben sind meist diskret:

  ```
  data Ev a = (Time, a)
  ```

- Benutzereingaben können Verhalten beeinflussen.
- Verhalten `Beh` wird durch Ereignisse `Ev` beeinflußt.
- Mausknöpfe:

  ```
  lbp, rbp  :: Event ()
  ```

Eingaben ändern Verhalten

- Beispiel: auf Knopfdruck blau

```
color1 :: Beh Color
color1 = red `untilB` (lbp ->> blue)
```

- Damit einen Planeten zeichnen:

```
rbplanet :: Beh Graphic
rbplanet = paint color1 planet
```

  - Mit `paint :: Beh Color-> Beh Figure-> Beh Graphic`.

- Rekursive Events

```
color1r :: Beh Color
color1r = red `untilB` lbp ->>
          blue `untilB` lbp ->> color1r
```

  - Erst rot, dann blau, dann wieder rot . . .
  - Beispiel: `paint color1r planet`

- Typen:

```
(->>)  :: Ev a-> b-> Ev b
untilB :: Beh a-> Ev (Beh a)-> Beh a
```

  - Idee: Inhalt des Events ist das zukünftige Verhalten

Auswahl

- Manchmal mehrere Events möglich
- Bei linken Knopf blau, bei rechtem Knopf gelb:

```
color2 :: Beh Color
color2 = red `untilB` (lbp ->> blue .|.
                       rbp ->> yellow)
```

- Typ:

```
(.|.) :: Ev a-> Ev a-> Ev a
```

- Rekursive Auswahl:

```
color2r = red `untilB` colEv where
  colEv = (lbp ->> blue `untilB` colEv) .|.
          (rbp ->> yellow `untilB` colEv)
```

- Beispiel: `paint color2r planet`

Rekursive Events

- Meisten reagieren wir auf Ströme von Events:
  - Siehe `color2r` oben.
- Einfacher mit `switch`

```
color2r' :: Beh Color
color2r' = red `switch` (lbp ->> blue .|.
                         rbp ->> yellow)
```

- Typ und Definition

```
switch :: Beh a -> Ev (Beh a) -> Beh a
a `switch` b = (a `untilB` b) `switch` b
```

Events mit Daten

- Zum Beispiel Tasten: `key :: Ev Char`
- Verarbeitung mit `=>>`:

```
color3 :: Beh Color
color3 = white `switch` (key =>> \c->
           case c of 'r' -> red
                     'g' -> green
                     'y' -> yellow
                     _   -> white)
```

- Beispiel: `paint color3 planet`

- Typ und Definition:

```
(=>>) :: Ev a-> (a-> b)-> Ev b

e ->> v = e =>> \_ -> v
```

Schnappschüsse

- Altes Verhalten bestimmt zukünftiges Verhalten
  - Event muß Schnappschuß des momentanen Verhaltens beinhalten
- Beispiel: `color3`, aber Farbe soll bei beliebiger Taste gleich bleiben.

```
color4 :: Beh Color
color4 = white `switch`
  (key `snapshot` color4 =>> \(c, old)->
       case c of 'r' -> red
                 'g' -> green
                 'y' -> yellow
                 _   -> constB old)
```

- `snapshot` kombiniert Event mit altem Verhalten
- Typ:

  ```
  snapshot :: Ev a -> Beh b -> Ev (a,b)
  ```

  - Variante für z.B. `Event()`:

    ```
    snapshot_ :: Ev a -> Beh b -> Ev b
    ```

Kurzzusammenfassung

```
(->>)      :: Ev a-> b-> Ev b
(=>>)      :: Ev a-> (a-> b)-> Ev b

untilB     :: Beh a-> Ev (Beh a)-> Beh a

  (.|.)    :: Ev a-> Ev a-> Ev a

switch     :: Beh a -> Ev (Beh a) -> Beh a

snapshot   :: Ev a -> Beh b -> Ev (a,b)
snapshot_  :: Ev a -> Beh b -> Ev b
```

Schwingungen

- Beispiel: ein schwingender Ball
- Schwingung der Frequenz f und Amplitude A:

  $$s(t) = A \sin(\omega t)$$

  mit Winkelgeschwindigkeit $\omega = 2\pi f$
- Schwingung in x-Richtung mit $f = \frac{1}{2}$:

```
swinging :: Beh Figure
swinging = translate (pairB (round' x) (round' y))
                     (circ 20) where
     x= 240* sin (pi*time)
```

- Beispiel: paint blue swinging

- Erweiterung: Bei Knopfdruck gedämpfte Schwingung.
- Gedämpfte Schwingung mit Dämpfungsfaktor δ und ungedämpfter Winkelgeschwindigkeit ω_0:

$$s(t) = Ae^{-\delta t}\sin(\omega t)$$

$$\omega = \sqrt{\omega_0^2 - \delta^2}$$

- Benutzung von `snapshot` für den Zeitpunkt:

```
y= 0 `switch` (lbp `snapshot_` time) =>>
   \t0-> let dt = time- constB t0
             om = sqrt((3*2*pi)^2- (0.25)^2)
         in  200* exp (-0.25*dt)* sin (om*dt)
```

- Beispiel: paint blue swinging

Boolesche Events

- Events als Prädikate über Verhalten
- Beispiel: nach drei Sekunden blau paint blink planet

```
blink :: Beh Color
blink = red `untilB` (when (time >* 3) ->> blue)
```

- Typen:

```
when  :: Beh Bool -> Ev ()
while :: Beh Bool -> Ev ()
```

  - `when` tritt nur einmal auf, `while` mehrmals

- Braucht mehr Lifting:
  - Boolesche Operatoren

    ```
    (&&*), (||*) :: Beh Bool -> Beh Bool -> Beh Bool
    ```

  - Prädikate:

    ```
    (<*), (>*) :: Ord a => Beh a -> Beh a -> Beh Bool
    ```

  - Instanz von `Beh` als `Ord` hätte falsche Signatur.

Integration und Ableitung

- Idee: über Verhalten integrieren und ableiten.
- Beispiel: Position s bei Anfangsposition s_0 und Geschwindigkeit v

$$s(t) = s_0 + \int_0^t v(t)dt$$

- Hier durch

```
integral :: Beh Double-> Beh Double
derive   :: Beh Double-> Beh Double
```

  - Integration immer von 0 bis t
  - Nur über `Double`

- Damit der springende Ball:

```
bouncy :: Beh Figure
bouncy = translate (pairB (round' x) (round' y))
                   (circ 20) where
  g = 200
  x = -250 + integral 50
  y = -250 + integral v
  v = integral g `switch`
        (hit `snapshot_` v =>>
           \w-> lift0 (-w) + integral g)
  hit = when (y >* 250)
```

- Beispiel: `paint red bouncy`

Akkumulatives Verhalten

- Schrittweise Veränderung: `step`

```
step :: a -> Ev a -> Beh a
```

- `a ‘step‘ e` fängt als `a` an, und reagiert dann auf `e`.

```
a ‘step‘ e = constB a ‘switch‘ e =>> constB
```

- Neues Verhalten als Funktion des letzten Verhaltens

```
stepAccum :: a-> Event (a-> a)-> Beh a
a ‘stepAccum‘ e = b where
   b = a ‘step‘
       (e ‘snapshot‘ b =>> \(f, a)-> f a))
```

- Beispiel: der Zähler.

```
counter :: Beh Int
counter = 0 `stepAccum` lbp ->> (+1)
```

- Beispiel: Farbe paint colcnt planet

```
colcnt :: Beh Color
colcnt = Black `stepAccum` lbp ->>
         (\c-> if c == maxBound
               then minBound else succ c)
```

Kurzzusammenfassung II

```
when  :: Beh Bool -> Ev ()
while :: Beh Bool -> Ev ()

integral :: Beh Double-> Beh Double
derive   :: Beh Double-> Beh Double

step :: a -> Ev a -> Beh a
stepAccum :: a-> Event (a-> a)-> Beh a
```

Implementationsaspekte

- Implementation des Verhalten: diskrete Zeit
  - Statt `Beh a= Time-> a` jetzt `Beh a= [Time]-> a`
  - Außerdem Reaktion auf beliebige Benutzereingaben:

    ```
    type UserAction = Event
    ```
- Implementation der Typen

  ```
  newtype Beh a
    = Beh (([Maybe UserAction],[Time])-> [a])
  newtype Ev a
    = Ev (([Maybe UserAction],[Time])-> [Maybe a])
  ```

  - i.e. `type Ev a = Beh (Maybe a)`
  - `newtype` ist ähnlich `data`, aber nur zur Typunterscheidung.

- Implementation der gelifteten Funktionen wie vorher.
- Implementation von `reactimate`:
  - Es gibt eine `event queue` (enthält Paare Ereignis) Zeitstempel `(Maybe UserAction, Time)`
  - Neu auftretende Ereingnisse (Benutzereingaben oder „Ticks“) hinten an Schlange anhängen;
  - Von vorne aufgetretene Ereignisse abarbeiten:
    - ▷ Aus Ereignis und Zeitstempel Grafik berechnen
    - ▷ anzeigen
    - ▷ auf Windowtick warten
  - Ereignisschlange wird nie leer, da immer mindestens ein Tick angehängt wird.

Abschließendes Beispiel: Pong

- Spielprinzip: elektronisches Squash
- Spielfeld: Wände, Schläger, Ball:

```
paddleball :: (Double, Double)-> Beh Graphic
paddleball v = walls `over` paddle `over` ball v
```

  - Parameter: Geschwindigkeit

- Die Wände: drei Rechtecke

```
walls :: Beh Graphic
walls = upper `over` left `over` right where
    upper = paint blue (trans (0, -230)
                              (rect 480 20))
    left  = paint blue (trans (-230, 0)
                              (rect 20 440))
    right = paint blue (trans (230, 0)
                              (rect 20 440))
```

  - Hilfsfunktion

```
trans :: (Beh Int, Beh Int)-> Beh Figure-> Beh Figure
trans (x, y) = translate (pairB x y)
```

- Der Schläger:
  - Rechteck $(50, 10)$ auf Y-Position 220
  - X-Position durch Maus bestimmt

```
paddle :: Beh Graphic
paddle = paint red (trans (fst mouse- 250, 220)
                          (rect 50 10))
```

- Der Ball:
  - Geschwindigkeit negieren, wenn Wand getroffen wird
  - Position: Integral über Geschwindigkeit
  - Wand getroffen:
    - In X-Richtung: $x < -200$ oder $x > 200$
    - In Y-Richtung: $y < -200$ oder $y > 200$ und Schläger dort

```
ball :: (Double, Double)-> Beh Graphic
ball (vx, vy) =  paint yellow (trans (round' xpos,
                   round' ypos) (circ 20)) where

  xvel    = vx `stepAccum` xbounce ->> negate
  xpos    = 0+ integral xvel
  xbounce = when (xpos >*  200 ||* xpos <* -200)

  yvel    = vy `stepAccum` ybounce ->> negate
  ypos    = -220+ integral yvel
  ybounce = when (ypos <* -200
            ||* ypos `between` (200, 220) &&*
                (lift1 fromInt (fst mouse)- 250)
                  `between` (xpos- 25, xpos+ 25))
```

- Hilfsfunktion:

```
between :: Beh Double-> (Beh Double, Beh Double)
                      -> Beh Bool
x `between` (a,b) = x >* a &&* x <* b
```

- Hauptfunktion:
  - Parameter: Geschwindigkeit
  - Anfangswertgeschwindigkeit: in X-Richtung zufällig

```
pong :: Double-> IO ()
pong vel = do vx <- randomRIO (vel/2, vel)
              sx <- randomRIO (-1,1)
              reactimate "PONG" (paddleball
                    (signum sx * vx, - abs vel))
```

- Spielen: `pong 100` , `pong 230`

Zusammenfassung

- Diskrete Ereignisse, kontinuierliches Verhalten.
- Ereignisse können Verhalten ändern.
- Typen `Ev a` und `Beh a`
- Konsequentes Lifting:
- Functional Reactive Programming.
- Beispiel für eine domänenspezifische Sprache (DSL).
  - Hier in Haskell eingebettet.
  - Wegen flexibler Syntax, Typklassen und Funktionen höherer Ordnung gut möglich.

Vorlesung vom 04.02.2002

Inhalt der Vorlesung

- Organisatorisches
- Rückblick über die Vorlesung
- Noch ein paar Haskell-Döntjes:
  - HaXML
  - Concurrent Haskell
  - HTk
- Ausblick

Der studienbegleitende Leistungsnachweis

- Bitte Scheinvordruck ausfüllen.
  - Siehe Anleitung.
  - Erhältlich vor FB3-Verwaltung (MZH Ebene 7)
  - Nur wer ausgefüllten Scheinvordruck abgibt, erhält auch einen.
- Bei Sylvie Rauer (MZH 8190) oder mir (MZH 8110) abgeben (oder zum Fachgespräch mitbringen)
- Nicht vergessen: in Liste eintragen!.

Das Fachgespräch

- Dient zur Überprüfung der Individualität der Leistung.
  - Insbesondere: Teilnahme an Beabeitung der Übungsblätter.
  - Keine Prüfung.
- Dauer: ca. 5–10 Min./Nase
- Inhalt: Übungsblätter und dazu relevanter Vorlesungsstoff
- Erste Frage: “Was ist ein funktionales Programm?“
- Termine:
  - Do. 07.02, Do. 14.02 — Liste vor MZH 8110
  - oder nach Vereinbarung.

Grundlagen der funktionalen Programmierung

- Definition von Funktionen durch rekursive Gleichungen
- Auswertung durch Reduktion von Ausdrücken
- Typisierung und Polymorphie
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Beweis durch strukturelle und Fixpunktinduktion

- Fortgeschrittene Features:
  - Modellierung von Zustandsabhängigkeit durch `IO`
  - Überladene Funktionen durch Typklassen
  - Unendliche Datenstrukturen und verzögerte Auswertung
- Beispiele:
  - Parserkombinatoren
  - Grafikprogrammierung
  - Animation
  - Funktionale Reaktive Programmierung

Zusammenfassung Haskell

Stärken:

- Abstraktion durch
  - Polymorphie und Typsystem
  - algebraische Datentypen
  - Funktionen höherer Ordnung
- Flexible Syntax
- Haskell als Meta-Sprache
- Ausgereifter Compiler
- Viele Büchereien

Schwächen:

- Komplexität
- Dokumentation
  - z.B. im Vergleich zu Java's APIs
- Büchereien

Warum funktionale Programmierung lernen?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen
- FP konzentriert sich auf wesentlichen Elemente moderner Programmierung:
  - Typisierung und Spezifikation
  - Datenabstraktion
  - Modularisierung und Dekomposition
- Blick über den Tellerrand — Blick in die Zukunft
  - Studium $\neq$ Programmierkurs— was kommt in 10 Jahren?

Hat es sich gelohnt?

HaXML

- HaXML ist XML für Haskell
  - Fallbeispiel: Vorteile von Typisierung, algebraische Datentypen.
- Was ist eigentlich XML?
  - Eine Notation für polynomiale Datentypen mit viel < und >
  - EBNF für Arme und Webdesigner

HaXML

- HaXML ist XML für Haskell
  - Fallbeispiel: Vorteile von Typisierung, algebraische Datentypen.
- Was ist eigentlich XML?
  - Eine Notation für polynomiale Datentypen mit viel < und >
  - EBNF für Arme und Webdesigner

- XML hat mehrere Schichten:
  - Basis: Definition von semantisch strukturierten Dokumenten
  - Präsentation von strukturierten Dokumenten
  - Einheitliche Definitionen und Dialekte
- Ziel: Trennung von Layout und Semantik

- XML ist erweitertes HTML
  - ohne Layoutinformation
  - dafür frei definierbare Elemente und Attribute
- Elemente und Attribute werden in DTD definiert
  - Entspricht einer Grammatik
- Beispiel: Ein Buch habe
  - Autor(en) — mindestens einen `[String]` (nichtleer)
  - Titel `String`
  - evtl. eine Zusammenfassung `[String]`
  - ISBN-Nummer `String`

- Die DTD für Bücher:

```
<!ELEMENT book (author+, title, abstract?)>
<!ATTLIST book isbn CDATA #REQUIRED>

<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT abstract (paragraph*)>
<!ELEMENT paragraph (#PCDATA)>
```

- `book`, . . . , `paragraph`: Elemente
- `isbn`: Attribut

- Beispiel-Buch:

```
<?xml version='1.0'?>
<!DOCTYPE book SYSTEM "book.dtd">
<book isbn="3540900357">
  <author>Saunders MacLane</author>
  <title>Categories for the Working Mathematician</title>
  <abstract>
    <paragraph>The introduction to category theory
    by one of its principal inventors.
    </paragraph>
 </abstract>
</book>
```

- HaXML:
  - DTD $\longrightarrow$ algebraischen Datentyp
  - Klasse `XmlContent`
  - Funktionen

```
readXml  :: XmlContent a => FilePath -> IO a
writeXml :: XmlContent a => FilePath -> a -> IO ()
```

  - Jedes Element einen Typ, Instanz von `XmlContent`
- Hier:
  - Übersetzung der DTD: `DtdToHaskell book.dtd > book.hs`

```
module DTD_book where
import Xml2Haskell

data Book = Book Book_Attrs [Author] Title (Maybe Abstract
            deriving (Eq,Show)
data Book_Attrs = Book_Attrs { bookIsbn :: String }
                  deriving (Eq,Show)
newtype Author = Author String           deriving (Eq,Show)
newtype Title = Title String             deriving (Eq,Show)
newtype Abstract = Abstract [Paragraph] deriving (Eq,Show)
newtype Paragraph = Paragraph String     deriving (Eq,Show)

instance XmlContent Book where
  ...
```

- Vorteile:
  - Trennung von Repräsentation und Inhalt
  - Einfache, getypte Manipulation von XML-Dokumenten
  - Beispiel formatierte Ausgabe: Zeigen

```
prt :: Book-> String
prt (Book (Book_Attrs  {bookIsbn=i}) as (Title t) abs) =
    authors++ ": "++ t++ "\n"++ abstr++ "ISBN: "++ i where
    authors = if length nms > 3 then  head nms ++ " et al"
              else concat (intersperse ", " nms)
    nms    = map (\(Author a)-> a) as
    abstr = case abs of
              Just (Abstract a)-> unlines (map
                                  (\ (Paragraph p)-> p) a) ++ "\n"
              Nothing -> ""
```

- Noch ein Beispiel: Suche
  - . . . in Bücherei nach Stichwörtern in Titel oder Zusammenfassung:
  - Erweiterte DTD `library.dtd`

    ```
    <!ELEMENT library (book*)>
    ```
  - Damit generierter Typ

    ```
    newtype Library = Library [Book] deriving (Eq,Show)
    ```
  - Hilfsfunktion: Liste aller Wörter aus Titel und Zusammenfassung

    ```
    getWords :: Book-> [String]
    getWords (Book _ _ (Title t) Nothing) = words t
    getWords (Book _ _ (Title t) (Just (Abstract a))) =
      words t ++ (concat (map (\ (Paragraph p)-> words p) a))
    ```

- Anfrage:

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem key. getWords) bs
```

- Nachteil: Groß/Kleinschreibung relevant

- Anfrage:

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem key. getWords) bs
```

- Nachteil: Groß/Kleinschreibung relevant
- Deshalb alles in Kleinbuchstaben wandeln:

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem (map toLower key).
               (map (map toLower).getWords)) bs
```

- Verbesserung: Suche mit regulären Ausdrücken
- Nutzt Regex-Bücherei des GHC:

```
import RegexString
data Regex -- abstrakt
mkRegex :: String -> Regex -- übersetzt regex
matchRegex :: Regex -> String -> Maybe [String]
```

- Damit neue Version von `query`:

```
query (Library bs) ex =
  filter (any (isJust.matchRegex (mkRegex ex)).
          getWords) bs
```

Concurrent Haskell

- `ghc` implementiert Haskell-Threads:
  - Haskell-Threads erzeugen/verwalten:

```
forkIO :: IO () -> IO ThreadID
killThread :: ThreadID -> IO ()
```

  - Zusätzliche Primitive zur Threadsynchronisation
  - Beispiel: Zeigen!

```
write :: Char -> IO ()
write c = putChar c >> write c
main :: IO ()
main = forkIO (write 'X') >> write '.'
```

- Erleichtert Programmierung reaktiver Systeme
  - Benutzerschnittstellen, Netzapplikationen, . . .

Der Haskell Web Server

- Ein RFC-2616 konformanter Webserver
  (Peyton Jones, Marlow 2000)
- Beispiel für ein
  - nebenläufiges,
  - robustes,
  - fehlertolerantes,
  - performantes System.
- Umfang: ca. 1500 LOC, „written with minimal effort“
- Performance: ca. 100 repl/s bis 700 repl

Grafische Benutzerschnittstellen

- HTk
  - Verkapselung von Tcl/Tk in Haskell
  - Nebenläufig mit Events
  - Entwickelt an der AG BKB (Dissertation E. Karlsen)
  - Mächtig, abstrakte Schnittstelle, clunky graphics
- GTk+HS
  - Verkapselung von GTK+ in Haskell
  - Zustandsbasiert mit call-backs
  - Entwickelt an der UNSW (M. Chakravarty)
  - Neueres Tookit, ästethischer, nicht ganz so mächtig

Grafische Benutzerschnittstellen mit HTk

- Statischer Teil: Aufbau des GUI
  - Hauptfenster öffnen

```
main:: IO ()
main =
  do main <- initHTk []
```

  - Knopf erzeugen und in Hauptfenster plazieren

```
b <- newButton main [text "Press me!"]
pack b []
```

- Dynamischer Teil: Verhalten während der Laufzeit
  - Knopfdruck als Event

```
click <- clicked b
```

  - Eventhandler aufsetzen

```
spawnEvent
 (forever
```

  - Eventhandler definieren

```
   (click >>> do nu_label <- mapM randomRIO
                               (replicate 5 ('a','z'))
                 b # text nu_label))
finishHTk
```

- Zeig' mal!

Hilfe!

- Haskell: primäre Entwicklungssprache an der AG BKB
  - Entwicklungsumgebung für formale Methoden (Uniform Workbench)
  - Werkzeuge für die Spezifikationssprache CASL (Tool Set CATS)
- Wir suchen studentische Hilfskräfte
  - für diese Projekte
- Wir bieten:
  - Angenehmes Arbeitsumfeld
  - Interessante Tätigkeit
- Wir suchen Tutoren für PI3
  - im WS 02/03 — meldet Euch!

Tschüß!