

Praktische Informatik 3

Einführung in die Funktionale Programmierung

Christoph Lüth

<http://www.informatik.uni-bremen.de/~cxl/>

WS 04/05



Vorlesung vom 25.10.2004: Einführung

Personal

- Vorlesung: Christoph Lüth <cx1>, MZH 8050, Tel. 7585
- Tutoren: Shi Hui <shi>
 - Diedrich Wolter <dwolter>
 - Christian Maeder <maeder>
 - Klaus Lüttich <luettich>
 - Kai-Florian Richter <richter>
 - Dennis Walter <dw>
 - Wolfgang Machert <wmachert>
 - Hennes Märtings <maertins>
- Fragestunde: Berthold Hoffmann <hof>
- Website: www.informatik.uni-bremen.de/~cx1/lehre/pi3.ws04.
- Newsgroup: fb3.lv.pi3.

Termine

- **Vorlesung:** Mo 10 – 12, kleiner Hörsaal („Keksdose“)
- **Tutorien:**

Di	8 – 10	MZH 1380	Wolfgang Machert
	17 – 19	MZH 1380	Diedrich Wolter
Mi	8 – 10	MZH 6240	Hennes Märtings
		MZH 7250	Shi Hui
	13 – 15	MZH 7230	Christian Maeder
		MZH 6240	Klaus Lüttich
		MZH 1380	Dennis Walter
Mi	17 – 19	MZH 1380	Kai-Florian Richter
- **Fragestunde (FAQ):** Mi 9 – 11 Berthold Hoffmann (MZH 8130)

Übungsbetrieb

- Ausgabe der Übungsblätter über die Website Montag nachmittag
- Besprechung der Übungsblätter in den Tutorien;
- Bearbeitungszeit zwei Wochen ab Tutorium,
Abgabe im Tutorium;
- Voraussichtlich sechs 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
 - z.B: Nebenläufigkeit; Grafik und Animation
- Lektüre:
Simon Thompson: Haskell — The Craft of Functional Programming
(Addison-Wesley, 1999)

Scheinkriterien — Vorschlag:

- Erfolgreiche Bearbeitung eines Übungsblattes: $\geq 40\%$
- Alle Übungsblätter erfolgreich zu bearbeiten.
- Es gibt ein Bonusübungsblatt, um Ausfälle zu kompensieren.
- Individualität der Leistung wird sichergestellt durch:
 - Vorstellung einer Lösung im Tutorium
 - Beteiligung im Tutorium
 - Ggf. Prüfungsgespräch (auch auf Wunsch)

Spielregeln

- PI1- und PI2- Schein sind **Voraussetzung**.
- **Quellen angeben** bei
 - **Gruppenübergreifender** Zusammenarbeit;
 - Internetrecherche, Literatur, etc.
- Erster **Täuschungsversuch**:
 - **Null** Punkte
 - Fachgespräch.
- Zweiter **Täuschungsversuch**: **Kein Schein**.
- **Deadline verpaßt?**
 - **Vorher** ankündigen, sonst **null** Punkte.

Einführung in die Funktionale Programmierung

Warum funktionale Programmierung (FP) lernen?

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

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

Referentielle Transparenz

- Programme als Funktionen

$$P : \textit{Eingabe} \rightarrow \textit{Ausgabe}$$

- Keine Variablen — keine Zustände
- Alle Abhängigkeiten explizit:
- Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext:

Referentielle Transparenz

Funktionen als Programme

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:

```
inc x = x + 1
```

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

- Auswertung durch Reduktion von Ausdrücken:

```
addDouble 6 4
```

Programmieren mit Funktionen

- **Programme** werden durch **Gleichungen** definiert:

```
inc x = x + 1
```

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

- Auswertung durch **Reduktion** von **Ausdrücken**:

```
addDouble 6 4  $\rightsquigarrow$  2 * (6 + 4)
```

Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

```
inc x = x + 1
```

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

- Auswertung durch Reduktion von Ausdrücken:

```
addDouble 6 4  $\rightsquigarrow$  2 * (6 + 4)  $\rightsquigarrow$  20
```

- Nichtreduzierbare Ausdrücke sind Werte
 - Zahlen, Zeichenketten, Wahrheitswerte, . . .

Auswertungsstrategie

- Von **außen** nach **innen**, **links** nach **rechts**.

```
inc (addDouble (inc 3) 4)
```

⇒

Auswertungsstrategie

- Von **außen** nach **innen**, **links** nach **rechts**.

`inc (addDouble (inc 3) 4)`

$\rightsquigarrow$ `(addDouble (inc 3) 4) + 1`

$\rightsquigarrow$

Auswertungsstrategie

- Von **außen** nach **innen**, **links** nach **rechts**.

`inc (addDouble (inc 3) 4)`

$\rightsquigarrow$ `(addDouble (inc 3) 4) + 1`

$\rightsquigarrow$ `2 * (inc 3 + 4) + 1`

$\rightsquigarrow$

Auswertungsstrategie

- Von **außen** nach **innen**, **links** nach **rechts**.

`inc (addDouble (inc 3) 4)`

$\rightsquigarrow$ `(addDouble (inc 3) 4) + 1`

$\rightsquigarrow$ `2 * (inc 3 + 4) + 1`

$\rightsquigarrow$ `2 * (3 + 1 + 4) + 1`

$\rightsquigarrow$

Auswertungsstrategie

- Von **außen** nach **innen**, **links** nach **rechts**.

`inc (addDouble (inc 3) 4)`

$\rightsquigarrow$ `(addDouble (inc 3) 4) + 1`

$\rightsquigarrow$ `2 * (inc 3 + 4) + 1`

$\rightsquigarrow$ `2 * (3 + 1 + 4) + 1`

$\rightsquigarrow$ `2 * 8 + 1` $\rightsquigarrow$ `17`

- Entspricht **call-by-need** (verzögerte Auswertung)
 - Argumentwerte werden erst ausgewertet, wenn sie benötigt werden.

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

~>

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

```
~>repeater "hallo"++ repeater "hello"
```

```
~>
```

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

```
⇒ repeater "hallo"++ repeater "hello"
```

```
⇒ ("hallo "++ "hallo ")++ repeater "hallo "
```

```
⇒
```

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

```
~>repeater "hallo"++ repeater "hello"
```

```
~>("hallo "++ "hallo ")++ repeater "hallo "
```

```
~>"hallo hallo "++ repeater "hallo "
```

```
~>
```

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

```
⇒ repeater "hallo"++ repeater "hello"
```

```
⇒ ("hallo "++ "hallo ")++ repeater "hallo "
```

```
⇒ "hallo hallo "++ repeater "hallo "
```

```
⇒ "hallo hallo "++ ("hallo "++ "hallo ")
```

```
⇒
```

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
repeater s = s ++ s
```

```
repeater (repeater "hallo ")
```

```
⇒ repeater "hallo"++ repeater "hello"
```

```
⇒ ("hallo "++ "hallo ")++ repeater "hallo "
```

```
⇒ "hallo hallo "++ repeater "hallo "
```

```
⇒ "hallo hallo "++ ("hallo "++ "hallo ")
```

```
⇒ "hallo hallo hallo hallo"
```

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

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 (Skript)
 - Kommandozeilenmodus: Reduktion von Ausdrücken
 - Keine Definitionen in der Kommandozeile
 - Hugs in Aktion.

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 $f :: a \rightarrow b$

Vorlesung vom 01.11.2004: Funktionen und Typen

Organisatorisches

- PI1/PI2 empfohlen aber nicht Voraussetzung für PI3.
- Tutorien: Ungleichverteilung.
- Übungsblätter:
 - Lösungen in $\text{\LaTeX}$ wie gehabt,
 - Formulare auf Webseite.

Inhalt

- Wie **definiere** ich eine **Funktion**?
 - Syntaktische Feinheiten
 - Von der Spezifikation zum Programm
- **Basisdatentypen**:
 - Wahrheitswerte, numerische und alphanumerische Typen
- **Strukturierte Datentypen**:
 - Listen und Tupel

Wie definiere ich eine Funktion?

Generelle Form:

- **Signatur:**

```
max :: Int -> Int -> Int
```

- **Definition**

```
max x y = if x < y then y else x
```

- Kopf, mit Parametern
- Rumpf (evtl. länger, mehrere Zeilen)
- Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf

Die Abseitsregel

Funktionsdefinition:

$f\ x_1\ x_2\ \dots\ x_n = E$

- **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
```

- Kann geschachtelt werden.

Die Alternative: Literate Programming

- **Literate Haskell** (`.lhs`): Quellcode besteht hauptsächlich aus Kommentar, Programmcode ausgezeichnet.
- In Haskell zwei Stile:
 - Alle Programmzeilen mit `>` kennzeichnen.
 - Programmzeilen in `\begin{code} . . . \end{code}` einschließen
- Umgebung `code` in $\text{\LaTeX}$ definieren:

```
\def\code{\verbatim}  
\def\endcode{\endverbatim}
```
- Mit $\text{\LaTeX}$ **setzen**, mit Haskell **ausführen**: (Quelle, ausführen).

```
test x = if x == 0 then 1 else 0
```

Funktionaler Entwurf und Entwicklung

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

Funktionaler Entwurf und Entwicklung

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

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

⇒ Erster Entwurf

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

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

⇒ 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
```

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

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

- Überprüfung, ob Zug legal:

Erste Verfeinerung

- Gewonnen, wenn Zug legal & Gegner kann nicht gewinnen:

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

- Überprüfung, ob Zug legal:

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

- Gegner kann nicht gewinnen, wenn
 - nur noch ein Hölzchen über, oder
 - kann nur Züge machen, bei denen es Antwort gibt, wo wir gewinnen

- Gegner kann nicht gewinnen, wenn
 - nur noch ein Hölzchen über, oder
 - kann nur Züge machen, bei denen es Antwort gibt, wo wir gewinnen

```
mustLose :: Int-> Bool
```

```
mustLose n
```

```
    | n == 1      = True
```

```
    | otherwise = canWin n 1 && canWin n 2 && canWin n 3
```

- Wir gewinnen, wenn es **legalen, gewinnenden** Zug gibt:

```
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. Testen.

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

Wahrheitswerte: Bool

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

<code>not</code>	<code>:: Bool -> Bool</code>	Negation
<code>&&</code>	<code>:: Bool -> Bool -> Bool</code>	Konjunktion
<code> </code>	<code>:: Bool -> Bool -> Bool</code>	Disjunktion
- Beispiel: ausschließende Disjunktion:

```
exOr :: Bool -> Bool -> Bool
```

```
exOr x y = (x || y) && (not (x && y))
```

- Alternative: explizite Fallunterscheidung

```
exOr x y
```

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

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

- Alternative: explizite Fallunterscheidung

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

- Alternative: explizite Fallunterscheidung

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

- Noch besser: ($y == \text{False} = \text{not } y$)

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

Das Rechnen mit Zahlen

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

Das Rechnen mit Zahlen

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

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte (≥ 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, rem :: Int-> Int-> Int`

`mod, quot :: 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)`
- Fallweise Definitionen möglich

Fließkommazahlen: Double

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
 - Logarithmen, Wurzel, Exponentiation, π und e , trigonometrische Funktionen
 - siehe Thompson S. 44
- Konversion in ganze Zahlen:
 - `fromIntegral :: Int, Integer -> Double`
 - `fromInteger :: Integer -> Double`
 - `round, truncate :: Double -> Int, Integer`
 - Überladungen mit Typannotation auflösen:
`round (fromInt 10) :: Int`
- **Rundungsfehler!**

Strukturierte Datentypen: Tupel und Listen

- **Strukturierte Typen**: konstruieren aus bestehenden Typen neue Typen.

- **Tupel** sind das kartesische Produkt:

(t_1, t_2) = alle Kombinationen von Werten aus t_1 und t_2 .

- **Listen** sind Sequenzen:

$[t]$ = endliche Folgen von Werten aus t

Beispiele

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

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

Beispiele

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

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

- 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`
- Entwurf:
 - **Rekursive Formulierung**:
erster Buchstabe = letzter Buchstabe, und Rest auch Palindrom
 - **Termination**:
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.

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.

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 08.11.2004:
Listenkomprehension,
Polymorphie und Rekursion**

Organisatorisches

Neues Tutorium:

Di	8 – 10	MZH 1380	Wolfgang Machert	36
		GW2 B 1410	Christian Maeder	
	17 – 19	MZH 1380	Diedrich Wolter	23
Mi	8 – 10	MZH 6240	Hennes Märtns	26
		MZH 7250	Shi Hui	25
	13 – 15	MZH 6240	Klaus Lüttich	22
		MZH 1380	Dennis Walter	15
		MZH 7230	Christian Maeder	15
Mi	17 – 19	MZH 1380	Kai-Florian Richter	13

(Idealgröße: 22)

Inhalt

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

Listenkomprehension

- Funktionen auf Listen folgen oft **Schema**:
 - Eingabe **generiert** Elemente,
 - die **getestet** und
 - zu einem Ergebnis **transformiert** werden

Listenkomprehension

- Funktionen auf Listen folgen oft **Schema**:
 - Eingabe **generiert** Elemente,
 - die **getestet** und
 - zu einem Ergebnis **transformiert** werden
- Beispiel **Palindrom**:
 - alle Buchstaben im String **str** zu Kleinbuchstaben.

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

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

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

Listenkomprehension

- 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:

```
[E | c1<- L1, c2<- L2, ..., test1, ..., testn ]
```

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

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

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)]
```

Operationen der Bücherei:

- Buch ausleihen und zurückgeben:

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

- Benutzt (++) zur Verkettung von DBase

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

- Suchfunktionen: Wer hat welche Bücher ausgeliehen usw. Test.

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

Jetzt oder nie — Polymorphie

- Definition von (++):

$(++) :: \text{DBase} \rightarrow \text{DBase} \rightarrow \text{DBase}$

$[] ++ ys = ys$

$(x:xs) ++ ys = x:(xs ++ ys)$

- Verkettung von Strings:

Jetzt oder nie — Polymorphie

- Definition von (++):

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

- Verketteten von Strings:

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

- **Gleiche Definition**, aber **unterschiedlicher Typ**!
 $\rightsquigarrow$ Zwei Instanzen einer allgemeineren Definition.

Polymorphie

- Polymorphie erlaubt **Parametrisierung über Typen**:

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$
$$[] ++ ys = ys$$
$$(x:xs) ++ ys = x:(xs ++ ys)$$

a ist hier eine **Typvariable**.

- Definition wird bei Anwendung instantiiert:

$$[3,5,57] ++ [39, 18] \quad \text{"hi"} ++ \text{"ho"}$$

aber **nicht**

$$[\text{True}, \text{False}] ++ [18, 45]$$

- Typvariable: vergleichbar mit Funktionsparameter

Polymorphie: 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

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

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

Siehe Thompson S. 91/92.

Palindrom zum letzten:

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

Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

<pre>f x y g1 = P otherwise = Q where v1 = M v2 x = N x</pre>	<pre>f x y = let v1 = M v2 x = N x in if g1 then P else Q</pre>
---	--

- `v1`, `v2`, ... werden **gleichzeitig** definiert (Rekursion!);
- Namen `v1` und Parameter (`x`) **überlagern** andere;
- Es gilt die **Abseitsregel**
⇒ Auf gleiche Einrückung der lokalen Definition achten;

Muster (*pattern*)

Funktionsparameter sind **Muster**: `head (x:xs) = x`

Ein **Muster** ist:

- **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:p1` (`ph, p1` sind wieder Muster)
- `[p1,p2,...pn]` ist syntaktischer Zucker für `p1:p2:...pn:[]`

Formen der Rekursion auf Listen

- **Primitive** Rekursion ist **Spezialfall** der **allgemeinen** 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
```

- Weitere Beispiele: `length`, `concat`, `(++)`, ...

- Auswertung:

$$\text{sum } [4, 7, 3] \rightsquigarrow 4 + 7 + 3 + 0$$

$$\text{concat } [A, B, C] \rightsquigarrow A ++ B ++ C ++ []$$

- Allgemeines Muster:

Definition:

$$f [] = A$$

$$f (x:xs) = x \otimes f xs$$

Auswertung:

$$f[x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes A$$

- Startwert (für die leere Liste) $A :: b$
- Rekursionsfunktion $\otimes :: a \rightarrow b \rightarrow b$
- Entspricht einfacher Iteration (**while**-Schleife).
- Vergleiche **Iterator** und **Enumeration** in **JAVA**.

Nicht-primitive Rekursion

- **Allgemeine** Rekursion:
 - Rekursion über **mehrere Argumente**
 - Rekursion über **andere Datenstruktur**
 - **Andere Zerlegung** als Kopf und Rest
- 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 Zerlegung — **Quicksort**:

- zerlege Liste in Elemente **kleiner**, **gleich** und **größer** dem ersten,
- sortiere Teilstücke, konkateniere Ergebnisse

- Andere Zerlegung — **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)
```

Beispiel: das n -Königinnen-Problem

- **Problem:** n Königinnen auf $n \times n$ -Schachbrett
- **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, n -te Königin so stellen, dass sie keine andere bedroht.
 - n -te Königin muß in n -ter Spalte platziert werden.

- Hauptfunktion:

```
queens num = putqueens num where
  putqueens :: Int-> [[Pos]]
  putqueens n | n == 0  = [[]]
               | otherwise =
               [ p++ [(n, m)] | p <- putqueens (n-1),
                               m <- [1.. num],
                               safe p (n, m)]
```

- `[n..m]`: Liste der Zahlen von n bis m
- Mehrere Generatoren in Listenkomprehension.
- Rekursion über Anzahl der Königinnen.

- Sichere neue Position: durch keine andere bedroht

```
safe :: [Pos] -> Pos -> Bool
safe others nu =
    and [ not (threatens other nu)
         | other <- others ]
```

- Verallgemeinerte Konjunktion `and :: [Bool] -> Bool`

- Gegenseitige Bedrohung:

- Bedrohung wenn in gleicher Zeile, Spalte, oder Diagonale.

```
threatens :: Pos -> Pos -> Bool
threatens (i, j) (m, n) =
    (j == n) || (i+j == m+n) || (i-j == m-n)
```

- Test auf gleicher Spalte `i==m` unnötig.

Testen.

Zusammenfassung

- Schemata für Funktionen über Listen:
 - Listenkomprension
 - primitive Rekursion
 - allgemeine Rekursion
- Polymorphie :
 - Abstraktion über Typen durch **Typvariablen**
- Lokale Definitionen mit **where** und **let**
- Überblick: vordefinierte Funktionen auf Listen

**Vorlesung vom 15.11.2004:
Formalisierung und Beweis
Funktionen höherer Ordnung**

Inhalt

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

Rekursive Definition, induktiver Beweis

- Definition ist **rekursiv**

- Basisfall (leere Liste)

- Rekursion ($x:xs$)

```
rev :: [a] -> [a]
```

```
rev [] = []
```

```
rev (x:xs) = rev xs ++ [x]
```

- 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: $P(0)$
- Induktionsschritt:

Induktionsvoraussetzung $P(x)$, zu zeigen $P(x + 1)$.

Beweis durch strukturelle Induktion

Zu zeigen:

Für alle Listen xs gilt $P(xs)$

Beweis:

- Induktionsbasis: $P([])$
- Induktionsschritt:

Induktionsvoraussetzung $P(xs)$, zu zeigen $P(x : xs)$

Ein einfaches Beispiel

Zu zeigen:

$$\forall xs\ ys. \text{len } (xs ++ ys) = \text{len } xs + \text{len } ys$$

Beweis: Induktion über xs .

- Induktionsbasis: $xs = []$

$$\begin{aligned} \text{len } [] + \text{len } ys &= 0 + \text{len } ys \\ &= \text{len } ys \\ &= \text{len } ([] ++ ys) \end{aligned}$$

- Induktionsschritt:

Voraussetzung: $\text{len } xs + \text{len } ys = \text{len } (xs ++ ys)$, dann

$$\begin{aligned} \text{len } (x : xs) + \text{len } ys &= 1 + \text{len } xs + \text{len } ys && \text{(Def. von len)} \\ &= 1 + \text{len } (xs ++ ys) && \text{(Ind.vor.)} \\ &= \text{len } (x : xs ++ ys) && \text{(Def. von len)} \end{aligned}$$

Noch ein Beispiel

Zu zeigen:

$$\forall xs\ ys. \text{rev}\ (xs ++ ys) = \text{rev}\ ys ++ \text{rev}\ xs$$

Beweis: Induktion über xs .

- Induktionsbasis:

$$\begin{aligned} \text{rev}\ ([\] ++ ys) &= \text{rev}\ ys \\ &= \text{rev}\ ys ++ \text{rev}\ [\] \end{aligned}$$

- Induktionsschritt:

Voraussetzung ist $\text{rev}\ (xs ++ ys) = \text{rev}\ ys ++ \text{rev}\ xs$, dann

$$\begin{aligned} \text{rev}\ (x : xs ++ ys) &= \text{rev}\ (xs ++ ys) ++ [x] && \text{(Def. von rev)} \\ &= (\text{rev}\ ys ++ \text{rev}\ xs) ++ [x] && \text{(Ind.vor.)} \\ &= \text{rev}\ ys ++ (\text{rev}\ xs ++ [x]) && \text{(++ assoziativ)} \\ &= \text{rev}\ ys ++ \text{rev}\ (x : xs) && \text{(Def. von rev)} \end{aligned}$$

Fixpunktinduktion

Gegeben: allgemein rekursive Definition

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

Zu zeigen: $\forall x. P(f x)$

Beweis: Annahme $P(f t)$, zu zeigen: $P(E)$.

- Zu zeigen: ein Rekursionsschritt erhält P
- Ein Fall für jede rekursive Gleichung.
- Induktionsverankerung: nichtrekursive Gleichungen.

Fallbeispiel: Verifikation von Mergesort

Verschiedene 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 als Argumente.
- Funktionen sind gleichberechtigt: Werte wie alle anderen
- Vorzüge:
 - Modellierung allgemeiner Berechnungsmuster
 - Höhere Wiederverwendbarkeit
 - Größere Abstraktion

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 `filter`: `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

- Signatur

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

- Definition

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

- **Beispiel:** Summieren von Listenelementen.

```
sum :: [Int] -> Int
```

```
sum xs = foldr (+) 0 xs
```

```
sum [3,12] = 3 + sum [12]  
           = 3+ 12+ sum []  
           = 3+ 12+ 0= 15
```

- **Beispiel:** Summieren von Listenelementen.

```
sum :: [Int] -> Int
```

```
sum xs = foldr (+) 0 xs
```

```
sum [3,12] = 3 + sum [12]
           = 3+ 12+ sum []
           = 3+ 12+ 0 = 15
```

- **Beispiel:** Flachklopfen von Listen.

```
concat :: [[a]] -> [a]
```

```
concat xs = foldr (++) [] xs
```

```
concat [11,12,13,14] = 11++ 12++ 13++ 14++ []
```

Listen zerlegen

- `take`, `drop`: n Elemente vom Anfang
- **Verallgemeinerung**: 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 von `takeWhile` und `dropWhile`

```
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
```

- Damit sortieren durch Einfügen:

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

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?

Beliebiges Sortieren

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

```
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)]
```

Zusammenfassung

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

Vorlesung vom 22.11.2004:
Funktionen Höherer Ordnung
Typklassen

Inhalt

- Funktionen höherer Ordnung
 - Letzte VL: verallgemeinerte Berechnungsmuster (`map`, `filter`, ...)
 - 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
```

Funktionen als Werte

- Funktionskomposition:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(f \ . \ g) \ x = f \ (g \ x)$$

- f nach g

Funktionen als Werte

- Funktionskomposition:

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$(f . g) x = f (g x)$

- f **nach** g

- Funktionskomposition **vorwärts**:

$(>.>) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c$

$(f >.> g) x = g (f x)$

- **Nicht** vordefiniert!

Funktionen als Werte

- Funktionskomposition:

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$(f . g) x = f (g x)$

- f **nach** g

- Funktionskomposition **vorwärts**:

$(>.>) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c$

$(f >.> g) x = g (f x)$

- **Nicht** vordefiniert!

- Identität:

$id :: a \rightarrow a$

$id x = x$

- Nützlicher als man denkt

Anonyme Funktionen

- Funktionen als **Werte** — nicht **jede** Funktion muß einen **Namen** haben.

- Beispiel (`insertsort`)

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

- Besser: statt `less` **anonyme Funktion**

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

- $\backslash x \rightarrow E \equiv f \text{ where } f \ x = E$

- Auch **pattern matching** möglich:

```
map (\(a,b) -> a+b) [(1,4), (3,2)]
```

Beispiel: Primzahlen

- Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben
- Dazu: **filtern** mit $\backslash n \rightarrow n \text{ 'mod' } p \neq 0$

```
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]
```

- NB: Mit 2 anfangen!

Zeigen.

η -Kontraktion

- Beispiel: 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

- Beispiel: 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?!

`any p = or . map p     $\equiv$     any p x = (or . map p) x`

- η -Kontraktion:

- Allgemein: `\x -> E x  $\equiv$  E`

- Bei Funktionsdefinition: `f x = E x  $\Leftrightarrow$  f = E`

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

$$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$$

auf k Argumente mit $k \leq n$

$$e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k$$

werden die Typen der Argumente gekürzt:

$$\begin{aligned} f &:: \cancel{t_1} \rightarrow \cancel{t_2} \rightarrow \dots \rightarrow \cancel{t_k} \rightarrow t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t \\ f \ e_1 \ \dots \ e_k &:: t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t \end{aligned}$$

Partielle Anwendung von Operatoren

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

```
elem x = any (== x)
```

- `(== x)` **Sektion** des Operators `==` (entspricht `\e -> e == x`)
- Auch möglich: `(x ==)` (entspricht `\e -> x == e`)
- Vergleiche `(3 <)` und `(< 3)`

Gewürzte Tupel: Curry

- Unterschied zwischen

$f :: a \rightarrow b \rightarrow c$ und $f :: (a, b) \rightarrow c$?

- Links partielle Anwendung möglich.
- Ansonsten **äquivalent**.

- Konversion:

- Rechts nach links:

$\text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$

$\text{curry } f \ a \ b = f \ (a, b)$

- Links nach rechts:

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$

$\text{uncurry } f \ (a, b) = f \ a \ b$

- Es gilt: $\text{curry} . \text{uncurry} = \text{id}$, $\text{uncurry} . \text{curry} = \text{id}$.

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:
(mit `type Line= String`)

[Line]

Zerlegung des Problems in einzelne Schritte

	Ergebnistyp
1. Text in Zeilen aufspalten: (mit <code>type Line= String</code>)	<code>[Line]</code>
2. Jede Zeile mit ihrer Nummer versehen:	<code>[(Int, Line)]</code>

Zerlegung des Problems in einzelne Schritte

- | | Ergebnistyp |
|--|---------------|
| 1. Text in Zeilen aufspalten:
(mit <code>type Line= String</code>) | [Line] |
| 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: (mit <code>type Line= String</code>)	[Line]
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:
(mit <code>type Line= String</code>) | [Line] |
| 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 unterschiedlichen Zeilen zusammenfassen: | [([Int], Word)] |

Zerlegung des Problems in einzelne Schritte

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

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 von Schritt 1–2

- In Zeilen zerlegen: `lines :: String -> [String]` (vordefiniert)
- 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]` (vordefiniert)

- Berücksichtigt nur Leerzeichen.
- Vorher alle Satzzeichen durch Leerzeichen ersetzen.

Implementierung von Schritt 3

- Zusammengenommen:

```
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)
```

Implementation von Schritt 4

- Liste alphabetisch nach Worten sortieren:

- Ordnungsrelation definieren:

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

- Sortieren mit generischer Sortierfunktion `qsortBy`

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

Implementation von Schritt 5

- **Gleiche Worte** in **unterschiedlichen 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)
```

Implementation von Schritt 6 — Test

- Alle **Worte** mit **weniger als vier Buchstaben** entfernen:

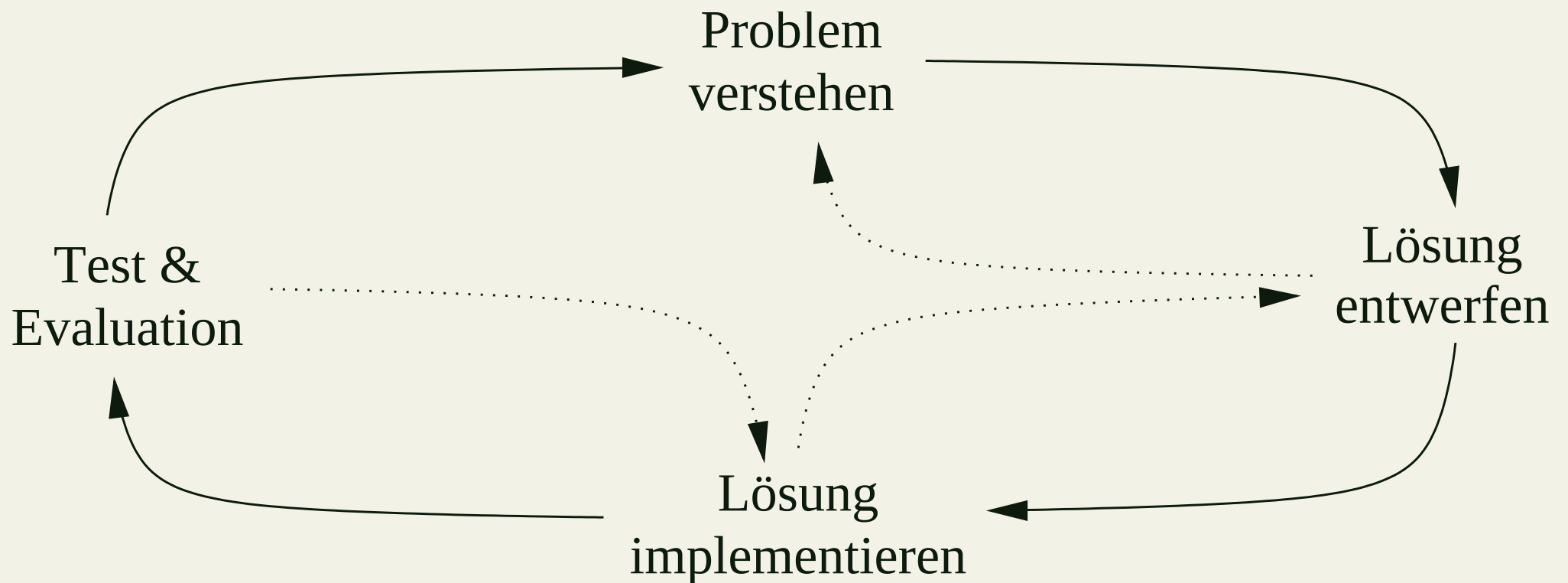
```
shorten :: ([Int],Word) -> ([Int],Word)
shorten = filter (\ (_, wd) -> length wd >= 4)
```

- **Alternative Definition:**

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

- Testen.

Der Programmmentwicklungszyklus



Typklassen

- Allgemeinerer Typ für `elem`:

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

zu `allgemein` wegen `c ==`

- `(==)` kann **nicht** für **alle** Typen definiert werden:
- Gleichheit auf Funktionen (z.B. `(==) :: (Int -> Int) -> (Int -> Int) -> Bool`) **nicht entscheidbar**.

- Lösung: Typklassen

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

`elem c = any (c ==)`

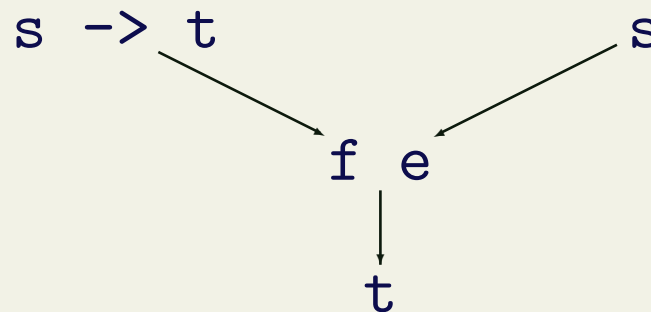
- Für `a` kann jeder Typ eingesetzt werden, für den `(==)` definiert ist.
- `Eq a` ist eine **Klasseneinschränkung** (*class constraint*)

Standard-Typklassen

- `Eq a` für `== :: a -> a -> Bool` (Gleichheit)
- `Ord a` für `<= :: a -> a -> Bool` (Ordnung)
 - Alle Basisdatentypen
 - Listen, Tupel
 - **Nicht** für Funktionen
- `Show a` für `show :: a -> String`
 - Alle Basisdatentypen
 - Listen, Tupel
 - **Nicht** für Funktionen
- `Read a` für `read :: String -> a`
 - Siehe `Show`

Typüberprüfung

- **Ausdrücke** in Haskell: **Anwendung** von Funktionen
- Deshalb Kern der **Typüberprüfung**: **Funktionsanwendung**



- Einfach solange Typen **monomorph**
 - d.h. **keine freien Typvariablen**
 - Was passiert bei **polymorphen** Ausdrücken?

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$$f(x, y) = (x, ['a' .. y])$$

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$$f(x, y) = (x, ['a' .. y])$$
$$a \rightarrow a \rightarrow [a]$$

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$$f(x, y) = (x, ['a' .. y])$$
$$a \rightarrow a \rightarrow [a]$$

Char

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$f(x, y) = (x, ['a' .. y])$

$a \rightarrow a \rightarrow [a]$

Char

Char

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$f(x, y) = (x, ['a' .. y])$

$a \rightarrow a \rightarrow [a]$

Char

Char

[Char]

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$f(x, y) = (x, ['a' .. y])$

$a \rightarrow a \rightarrow [a]$

Char

Char

[Char]

a

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: **Unifikation**.
- Beispiel:

$$\begin{array}{c}
 f(x,y) = (x, ['a' \dots y]) \\
 \qquad \qquad \qquad a \rightarrow a \rightarrow [a] \\
 \qquad \qquad \qquad \text{Char} \\
 \qquad \qquad \qquad \qquad \qquad \text{Char} \\
 \qquad \qquad \qquad \qquad \qquad [Char] \\
 \qquad \qquad \qquad a \\
 \qquad \qquad \qquad (a, [Char]) \\
 f :: (a, Char) \rightarrow (a, [Char])
 \end{array}$$

- Zweites Beispiel:

$$g(m, zs) = m + \text{length } zs$$

- Zweites Beispiel:

$$g(m, zs) = m + \text{length } zs$$
$$[b] \rightarrow \text{Int}$$

- Zweites Beispiel:

$g(m, zs) = m + \text{length } zs$

$[b] \rightarrow \text{Int}$

$[b]$

- Zweites Beispiel:

$g(m, zs) = m + \text{length } zs$

$[b] \rightarrow \text{Int}$

$[b]$

Int

- Zweites Beispiel:

$g(m, zs) = m + \text{length } zs$

$[b] \rightarrow \text{Int}$

$[b]$

Int

Int

- Zweites Beispiel:

$$g(m, zs) = m + \text{length } zs$$

$$[b] \rightarrow \text{Int}$$

$$[b]$$

$$\text{Int}$$

$$\text{Int}$$

$$\text{Int}$$

$$g :: (\text{Int}, [b]) \rightarrow \text{Int}$$

- Drittes Beispiel:

$$h = g . f$$

- $(.) :: (y \rightarrow z) \rightarrow (x \rightarrow y) \rightarrow x \rightarrow z$

- Drittes Beispiel:

$$h = g . f$$

- $(.) :: (y \rightarrow z) \rightarrow (x \rightarrow y) \rightarrow x \rightarrow z$
- $g :: (\text{Int}, [b]) \rightarrow \text{Int}$

- Drittes Beispiel:

$$h = g . f$$

- $(.) :: (y \rightarrow z) \rightarrow (x \rightarrow y) \rightarrow x \rightarrow z$
- $g :: (\text{Int}, [b]) \rightarrow \text{Int}$
- $f :: (a, \text{Char}) \rightarrow (a, [\text{Char}])$

- Drittes Beispiel:

$$h = g . f$$

- $(.) :: (y \rightarrow z) \rightarrow (x \rightarrow y) \rightarrow x \rightarrow z$
- $g :: (\text{Int}, [b]) \rightarrow \text{Int}$
- $f :: (a, \text{Char}) \rightarrow (a, [\text{Char}])$

- Hier **Unifikation** von $(a, [\text{Char}])$ und $(\text{Int}, [b])$ zu
 $(\text{Int}, [\text{Char}])$

- Drittes Beispiel:

$$h = g . f$$

- $(.) :: (y \rightarrow z) \rightarrow (x \rightarrow y) \rightarrow x \rightarrow z$
- $g :: (\text{Int}, [b]) \rightarrow \text{Int}$
- $f :: (a, \text{Char}) \rightarrow (a, [\text{Char}])$

- Hier **Unifikation** von $(a, [\text{Char}])$ und $(\text{Int}, [b])$ zu
 $(\text{Int}, [\text{Char}])$

- Damit

$$h :: (\text{Int}, [\text{Char}]) \rightarrow \text{Int}$$

Typunifikation

- **Allgemeinste Instanz** zweier Typausdrücke s und t
 - Kann undefiniert sein.
- Berechnung **rekursiv**:
 - Wenn beides **Listen**, Berechnung der Instanz der **Listenelemente**;
 - Wenn beides **Tupel**, Berechnung der Instanzen der **Tupel**;
 - Wenn eines **Typvariable**, zu anderem Ausdruck **instanziieren**;
 - ▷ Dabei **Überprüfung** auf **Diskjunktheit**!
 - Wenn beide **unterschiedlich**, **undefiniert**;
 - Dabei **Vereinigung** der **Typeinschränkungen**
- Anschaulich: Schnittmenge der Instanzen.

Zusammenfassung

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

Vorlesung vom 29.11.2004: Algebraische Datentypen

Inhalt

- Letzte VL: Funktionsabstraktion durch Funktionen höherer Ordnung.
- Heute: Datenabstraktion durch algebraische Datentypen.
- Einfache Datentypen: Aufzählungen und Produkte
- Der allgemeine Fall
- Bekannte Datentypen: Maybe, Bäume
- Geordnete Bäume
- Abgeleitete Klasseninstanzen

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**.
- $\implies$ Einfachster algebraischer Datentyp: **Aufzählungstyp**.

Aufzählungen

- Beispiel: Wochentage

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- **Konstruktoren** sind **Konstanten**:

```
Mon :: Weekday, Tue :: Weekday, Wed :: Weekday, ...
```

- Konstruktoren werden **nicht** deklariert.

- **Funktionsdefinition** durch **pattern matching**:

```
isWeekend :: Weekday -> Bool
```

```
isWeekend Sat = True
```

```
isWeekend Sun = True
```

```
isWeekend _   = False
```

Produkte

- Beispiel: **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 29 Nov 2004
```

```
bloomsday = Date 16 Jun 1904
```

```
fstday = Date 1 Jan 1
```

- **Konstruktor:**

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

Funktionsdefinition für Produkte

- 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      = 29
year bloomsday = 1904
```

Alternativen

Beispiel: Eine **geometrische Figur** ist

- ein **Kreis**, gegeben durch **Mittelpunkt** und **Durchmesser**,
- oder ein **Rechteck**, gegeben durch **zwei Eckpunkte**,
- oder ein **Polygon**, gegeben durch **Liste von Eckpunkten**.

Alternativen

Beispiel: Eine **geometrische Figur** ist

- ein **Kreis**, gegeben durch **Mittelpunkt** und **Durchmesser**,
- oder ein **Rechteck**, gegeben durch **zwei Eckpunkte**,
- oder ein **Polygon**, gegeben durch **Liste von Eckpunkten**.

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

- Ein **Konstruktor** pro **Variante**.

```
Circ :: Point -> Int -> Shape
Rect :: Point -> Point -> Shape
Poly :: [Point] -> Shape
```

Funktionen auf Geometrischen Figuren

- Funktionsdefinition durch **pattern matching**.

- Eine Gleichung pro Konstruktor

- Beispiel: Anzahl Eckpunkte

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

- Beispiel: Translation um einen Punkt (Vektor):

```
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)
```

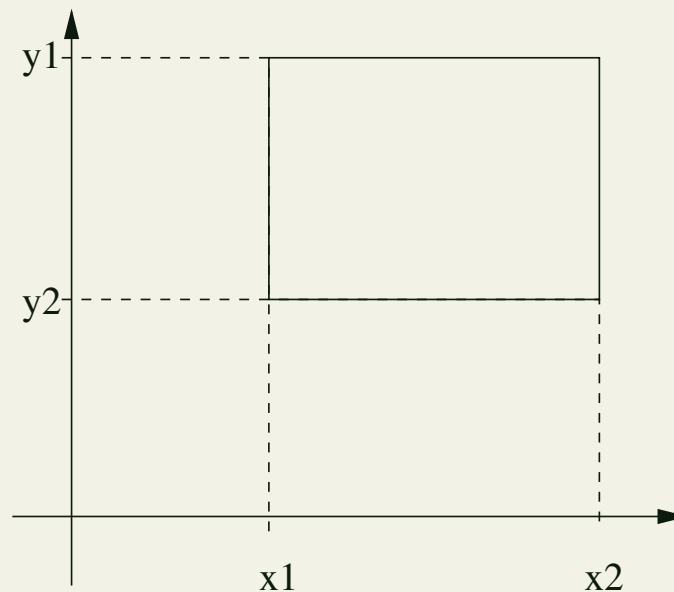
Beispiel: Flächenberechnung

- Einfach für Kreis und Rechteck.

```
area :: Shape -> Double
```

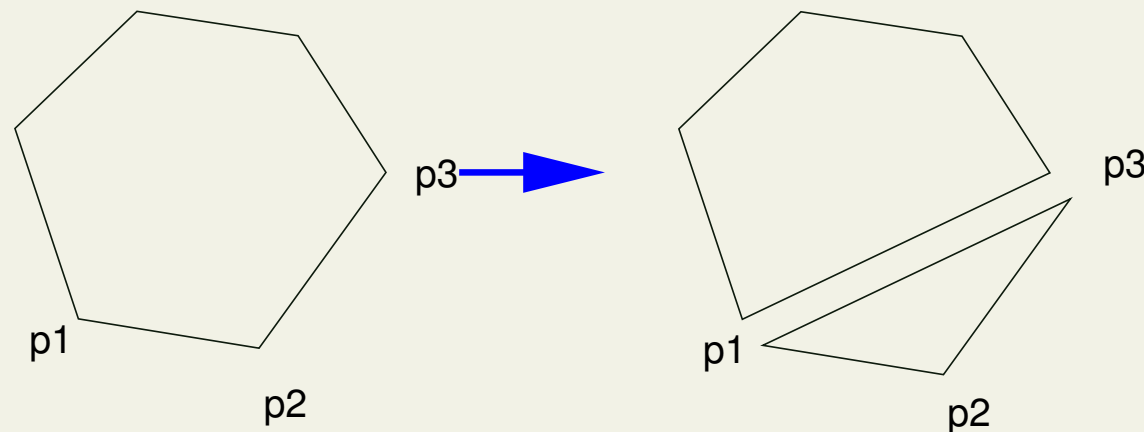
```
area (Circ _ d) = pi * (fromIntegral d)
```

```
area (Rect (x1, y1) (x2, y2)) =  
    abs ((x2 - x1) * (y2 - y1))
```



Beispiel: Flächenberechnung

- Fläche für Polygone:
 - Vereinfachende **Annahme**: Polygone **konvex**
 - **Reduktion** auf einfacheres Problem und **Rekursion**:



```
area (Poly ps) | length ps < 3 = 0
```

```
area (Poly (p1:p2:p3:ps)) =  
    triArea p1 p2 p3 +  
    area (Poly (p1:p3:ps))
```

Beispiel: Flächenberechnung

- Fläche für **Dreieck** mit **Seitenlängen** a, b, c (Heron):

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{mit} \quad s = \frac{1}{2}(a+b+c)$$

```
triArea :: Point-> Point-> Point-> Double
```

```
triArea p1 p2 p3 =
```

```
    let s= 0.5*(a+ b+ c)
```

```
        a= dist p1 p2
```

```
        b= dist p2 p3
```

```
        c= dist p3 p1
```

```
    in  sqrt (s*(s- a)*(s- b)*(s- c))
```

- **Distanz** zwischen **zwei Punkten** (Pythagoras):

```
dist :: Point-> Point-> Double
```

```
dist (x1, y1) (x2, y2) = sqrt((x1-x2)^2+ (y2- y1)^2)
```

Der allgemeine Fall

Definition von T :

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ \quad \dots \\ \quad | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

- Konstruktoren (und Typen) werden groß geschrieben.
- Konstruktoren $C_1, \dots, C_n$ sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \Rightarrow i = j$$

- Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \Rightarrow x_i = y_i$$

- Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Rekursive Datentypen

Der **definierte Typ** **T** kann **rechts** benutzt werden.

- **Beispiel**: einfache arithmetische **Ausdrücke** sind
 - Zahlen (Literele), oder
 - Variablen (Zeichenketten), oder
 - Addition zweier **Ausdrücke**, oder
 - Multiplikation zweier **Ausdrücke**.

```
data Expr = Lit Int
          | Var String
          | Add Expr Expr
          | Mult Expr Expr
```

- Funktion darauf meist auch **rekursiv**.

- Ausdruck auswerten (gegeben Auswertung der Variablen):

```
eval :: (String -> Int) -> Expr -> Int
eval f (Lit n) = n
eval f (Var x) = f x
eval f (Add e1 e2) = eval f e1 + eval f e2
eval f (Mult e1 e2) = eval f e1 * eval f e2
```

- Ausdruck ausgeben:

```
print :: Expr -> String
print (Lit n) = show n
print (Var x) = x
print (Add e1 e2) = "(" ++ print e1 ++ "+" ++ print e2 ++ ")"
print (Mult e1 e2) = "(" ++ print e1 ++ "*" ++ print e2 ++ ")"
```

- Testen.

- Primitive Rekursion auf Ausdrücken
 - Eine Funktion für Rekursionsverankerung
 - Eine Funktion für Variablen
 - Eine binäre Funktion für Addition
 - Eine binäre Funktion für Multiplikation

```
foldE :: (Int-> a)-> (String-> a)  
       -> (a-> a-> a)-> (a-> a-> a)-> Expr-> a
```

```
foldE b v a m (Lit n)      = b n
```

```
foldE b v a m (Var x)      = v x
```

```
foldE b v a m (Add e1 e2) = a (foldE b v a m e1)  
                             (foldE b v a m e2)
```

```
foldE b v a m (Mult e1 e2) = m (foldE b v a m e1)  
                             (foldE b v a m e2)
```

- Damit Auswertung und Ausgabe:

```
eval' f  = foldE id f (+) (*)
print'   =
  foldE show id
    (\s1 s2-> "("++ s1++ "+"++ s2++ ")")
    (\s1 s2-> "("++ s1++ "*"++ s2++ ")")
```

Polymorphe Datentypen

- Algebraische Datentypen parametrisiert über Typen:

```
data Pair a = Pair a a
```

- Paar: zwei beliebige Elemente gleichen Typs.

Zeigen.

Polymorphe Datentypen

- Algebraische Datentypen parametrisiert über Typen:

```
data Pair a = Pair a a
```

- Paar: zwei beliebige Elemente gleichen Typs.

Zeigen.

- Elemente vertauschen:

```
twist :: Pair a -> Pair a  
twist (Pair a b) = Pair b a
```

- Map für Paare:

```
mapP :: (a -> b) -> Pair a -> Pair b  
mapP f (Pair a b) = Pair (f a) (f b)
```

Polymorph und rekursiv: Listen selbstgemacht

Eine **Liste** von **a** ist

- entweder **leer**
- oder ein **Kopf a** und eine **Restliste List a**

```
data List a = Mt | Cons a (List a)
```

- Syntaktischer Zucker der vordefinierten Listen:

```
data [a] = [] | a : [a]
```

- **Nicht** benutzerdefinierbar: Typ **[a]**, Konstruktor **[]**
- **Operatoren** als binäre **Konstruktoren** möglich.

Funktionen auf Listen

- 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
```

Modellierung von Fehlern: Maybe a

- Typ a plus Fehlerelement

- Im Prelude vordefiniert.

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

- Nothing modelliert 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
```

Funktionen auf Maybe a

- Anwendung von Funktion mit Default-Wert für Fehler (**vordefiniert**):

```
maybe :: b -> (a -> b) -> Maybe a -> b  
maybe d f Nothing  = d  
maybe d f (Just x) = f x
```

- **Liften** von Funktionen ohne Fehlerbehandlung:
 - Fehler bleiben **erhalten**.

```
fmap :: (a -> b) -> Maybe a -> Maybe b  
fmap f Nothing = Nothing  
fmap f (Just x) = Just (f x)
```

Binäre Bäume

Ein binärer Baum ist

- Entweder leer,
- oder ein Knoten mit genau **zwei** Unterbäumen.

Knoten tragen eine Markierung.

```
data Tree a = Null
            | Node (Tree a) a (Tree a)
```

- Andere **Möglichkeit**: Markierungen für Knoten und Blätter:

```
data Tree' a b = Null'
               | Leaf' b
               | Node' (Tree' a b) a (Tree' a b)
```

Funktionen auf Bäumen

- Test auf Enthaltensein:

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

Funktionen auf Bäumen

Primitive Rekursion auf Bäumen:

- Rekursionsanfang
- Rekursionsschritt:
 - Label des Knotens,
 - **Zwei** Rückgabewerte für **linken** und **rechten** Unterbaum.

```
foldT :: (a -> b -> b -> b) -> b -> Tree a -> b
```

```
foldT f e Null = e
```

```
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

- Damit Elementtest:

```
member' t x =
```

```
    foldT (\e b1 b2 -> e == x || b1 || b2) False t
```

- Testen.

Funktionen auf Bäumen

- Traversal: `preorder`, `inorder`, `postorder`

```
preorder  :: Tree a -> [a]
```

```
inorder   :: Tree a -> [a]
```

```
postorder :: Tree a -> [a]
```

```
preorder  = foldT (\x t1 t2 -> [x] ++ t1 ++ t2) []
```

```
inorder   = foldT (\x t1 t2 -> t1 ++ [x] ++ t2) []
```

```
postorder = foldT (\x t1 t2 -> t1 ++ t2 ++ [x]) []
```

- Äquivalente Definition ohne `foldT`:

```
preorder' Null = []
```

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

- Testen.

Geordnete Bäume

- Voraussetzung:

- Ordnung auf a ($\text{Ord } a$)
- Es soll für alle Bäume gelten:

$$\forall \text{Tree } a \text{ l r. } \text{member } x \text{ l} \Rightarrow x < a \wedge \text{member } x \text{ r} \Rightarrow a < x$$

- Test auf Enthaltensein vereinfacht:

```
member :: Ord a => Tree a -> a -> Bool
```

```
member Null _ = False
```

```
member (Node l a r) b
```

```
  | b < a  = member l b
```

```
  | a == b = True
```

```
  | b > a  = member r b
```

Geordnete Bäume

- Ordnungserhaltendes Einfügen

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

- **Problem:** Erzeugung ungeordneter Bäume möglich.
 - Lösung erfordert **Verstecken** der Konstrukturen — nächste VL.

Geordnete Bäume

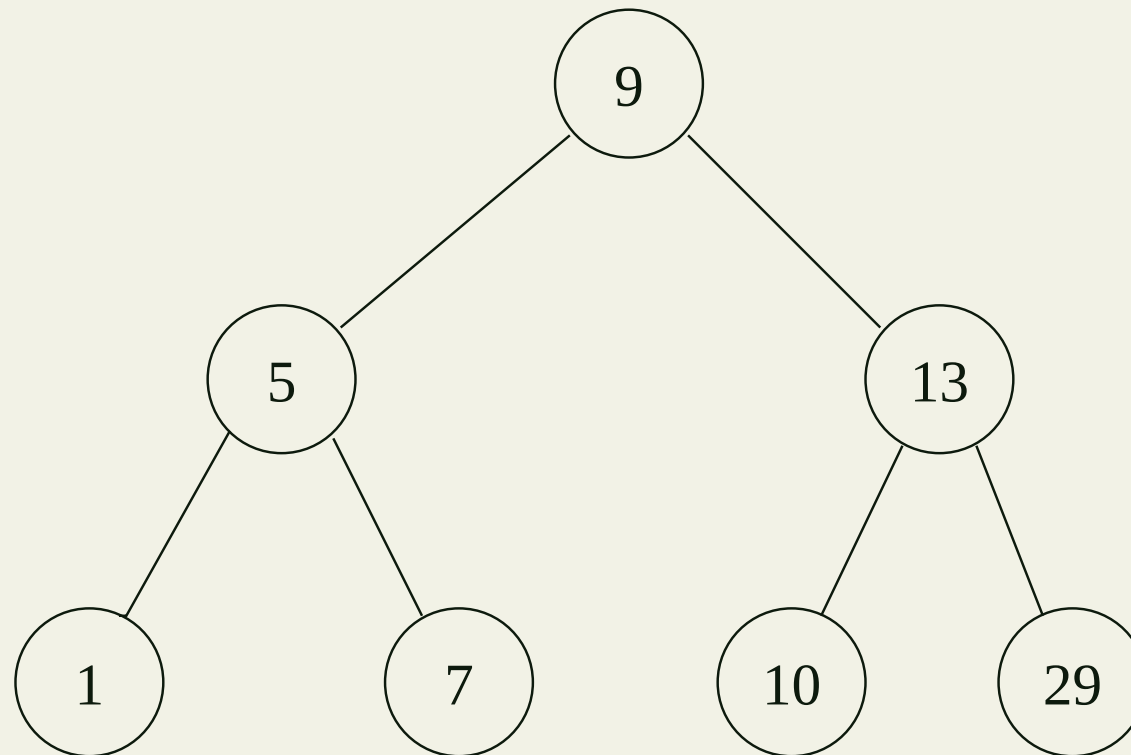
- Löschen:

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

- `join` fügt zwei Bäume ordnungserhaltend zusammen.

Geordnete Bäume

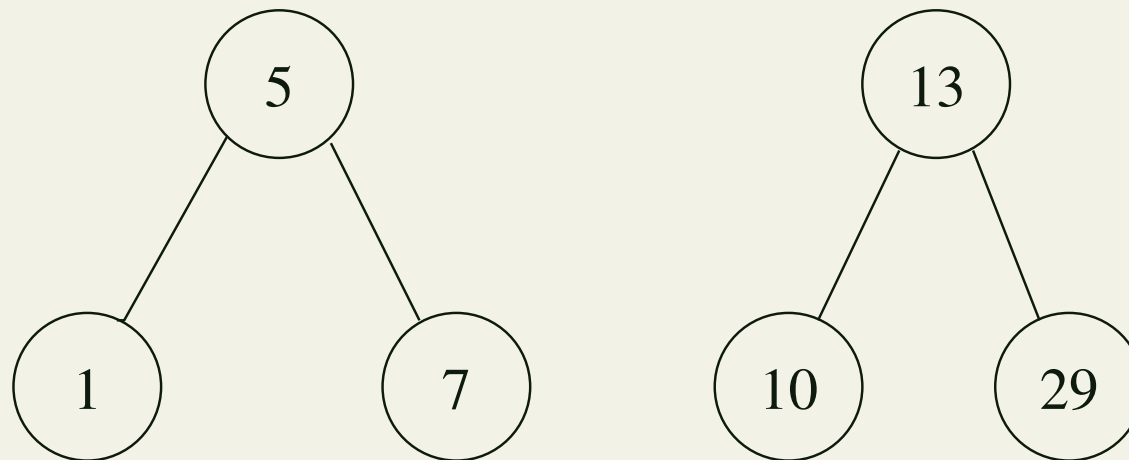
- Beispiel: Gegeben folgender Baum, dann `delete t 9`



- Wurzel wird gelöscht.

Geordnete Bäume

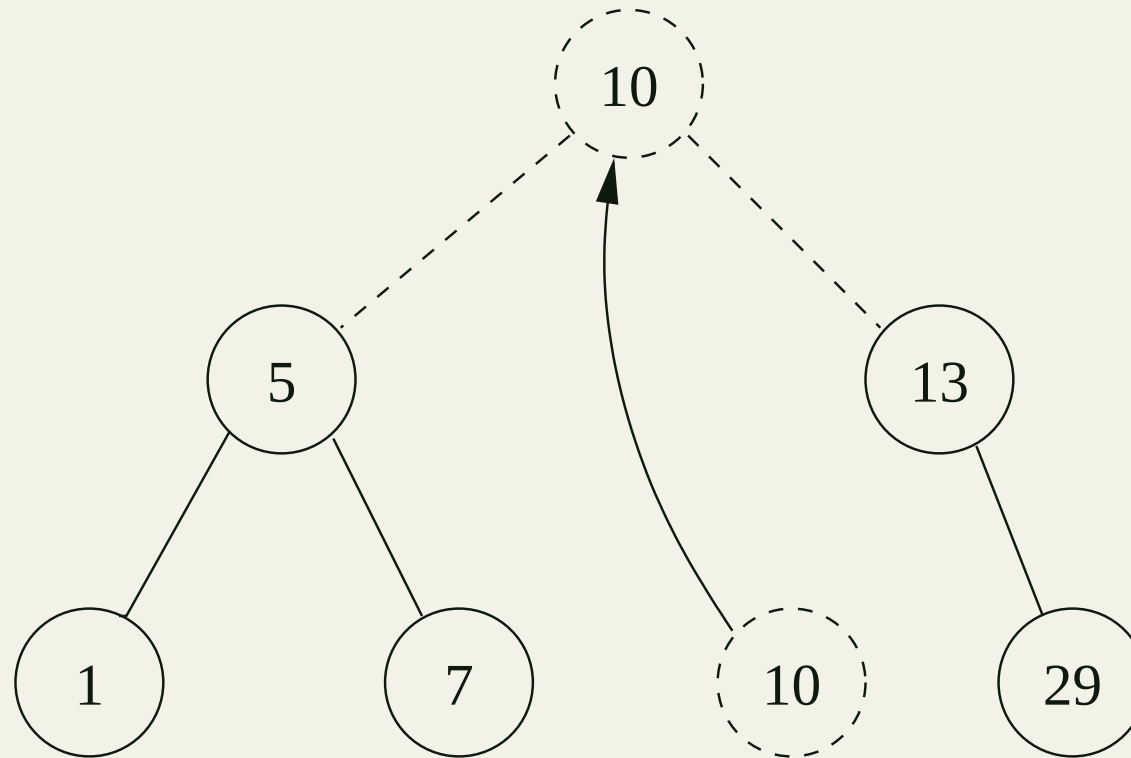
- Wurzel wurde gelöscht:



- `join` fügt Bäume ordnungserhaltend zusammen.

Geordnete Bäume

- `join` fügt zwei Bäume ordnungserhaltend zusammen.



- Wurzel des neuen Baums: Knoten **links unten** im rechten Teilbaum (oder Knoten rechts unten im linken Teilbaum)

Geordnete Bäume

- Implementation:

- `splitTree` spaltet Baum in Knoten links unten und Rest.
- Testen.

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

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)`

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 Bäume:
Verstecken von Konstruktoren nötig — nächste Vorlesung.

Vorlesung vom 06.12.2004

Abstrakte Datentypen

Inhalt

- Letzte VL:
 - Datenabstraktion durch **algebraische Datentypen**
 - Problem mit geordneten Bäumen: **Konstruktoren** sichtbar
- Heute: **abstrakte Datentypen** (ADTs) in Haskell
- Beispiele für bekannte ADTs:
 - **Stapel** und **Schlangen**: **Stack** und **Queue**
 - **Endliche Mengen**: **Set**
 - **Assoziationslisten** (endliche Abbildungen): **Finite Map**
 - **Graphen**

Abstrakte Datentypen

Ein **abstrakter Datentyp** besteht aus einem **Typ** und **Operationen** darauf.

Beispiele:

- **geordnete Bäume**, mit leerer Baum, einfügen, löschen, suchen;
- **Speicher**, mit Operationen lesen und schreiben;
- **Stapel**, mit Operationen **push**, **pop**, **top**;
- **Schlangen**, mit Operationen einreihen, Kopf, nächstes Element;

Repräsentation des Typen **versteckt**.

Module in Haskell

- Einschränkung der Sichtbarkeit durch **Verkapselung**
- **Modul**: Kleinste verkapselbare **Einheit**
- 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
- Gleichzeitig: Übersetzungseinheit (getrennte Übersetzung)

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 :: Ord a => Store a b -> a -> Maybe b`
 - Möglicherweise undefiniert.
- Wert schreiben:
`update :: Ord a => Store a b -> a -> b -> Store a b`

Moduldeklaration

- Moduldeklaration

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

- Signaturen nicht nötig, aber **sehr** hilfreich.

Erste Implementation

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

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

- Leerer Speicher: leere Liste

```
initial = Store []
```

- Lookup (**Neu:** case für Fallunterscheidungen):

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

- Update: neues Paar vorne anhängen

Test.

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

Zweite Implementation

- Speicher als Funktion

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

- Leerer Speicher: konstant undefiniert

```
initial  = Store (const Nothing)
```

- Lookup: Funktion anwenden

```
value (Store f) a  = f a
```

- Update: punktweise Funktionsdefinition

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

Ein Interface, zwei Implementierungen:

```
module Store (Store, initial, value, update) where
```

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

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

```
data Store a b =  
  Store (a-> Maybe b)  
initial =  
  Store (const Nothing)  
value (Store f) a = f a  
  
update (Store f) a b =  
  Store (\x-> if x== a  
             then Just b  
             else f x)
```

Export von Datentypen

- `Store(...)` exportiert Konstruktoren
 - Implementation sichtbar
 - Pattern matching möglich
- `Store` exportiert **nur** den Typ
 - Implementation 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 [qualified] Name [hiding] (Bezeichner)`

- Ohne Bezeichner wird alles importiert

- `qualified`: **qualifizierte** Namen

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

- `hiding`: Liste der Bezeichner wird **nicht** importiert

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

- **Selektiver Import**: nur einiges Bezeichner importieren

```
import List (find)
```

- Miteinander **kombinierbar**.

Schnittstelle vs. Semantik

- Stacks
 - Typ: $St\ a$
 - Initialwert:
 $empty :: St\ a$
 - Wert ein/auslesen
 $push :: a \rightarrow St\ a \rightarrow St\ a$
 $top :: St\ a \rightarrow a$
 $pop :: St\ a \rightarrow St\ a$
 - Test auf Leer
 $isEmpty :: St\ a \rightarrow Bool$
 - Last in, first out.
- Queues
 - Typ: $Qu\ a$
 - Initialwert:
 $empty :: Qu\ a$
 - Wert ein/auslesen
 $enq :: a \rightarrow Qu\ a \rightarrow Qu\ a$
 $first :: Qu\ a \rightarrow a$
 $deq :: Qu\ a \rightarrow Qu\ a$
 - Test auf Leer
 $isEmpty :: Qu\ a \rightarrow Bool$
 - First in, first out.
- Gleiche Signatur, unterschiedliche Semantik.

Implementation von Stack: Liste

- Sehr einfach wg. last in, first out
- `empty` ist `[]`
- `push` ist `(:)`
- `top` ist `head`
- `pop` ist `tail`
- `isEmpty` ist `null`
- Zeigen.

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
 - Invariante: erste Liste leer gdw. Queue leer

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
-----------	----------	-------	----------------

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
deq	4	5 → 7	([7], [5])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
deq	4	5 → 7	([7], [5])
deq	7	5	([5], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
deq	4	5 → 7	([7], [5])
deq	7	5	([5], [])
deq	5		([], [])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
deq	4	5 → 7	([7], [5])
deq	7	5	([5], [])
deq	5		([], [])
deq	error		([], [])

Implementation

- Modulkopf, Exportliste:

```
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 [] _) = error "Qu: first of mt Q"  
first (Qu (x:xs) _) = x
```

- Bei `enq` und `deq` Invariante prüfen

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

`deq (Qu [] _ ) = error "Qu: deq of mt Q"`

`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 Sammlung von Elementen so dass
 - kein Element doppelt ist, und
 - die Elemente nicht angeordnet sind.
- Operationen:
 - leere Menge und Test auf leer,
 - Element einfügen,
 - Element löschen,
 - Test auf Enthaltensein,
 - Elemente aufzählen.

Endliche Mengen — Schnittstelle

- Typ `Set a`
 - Typ `a` mit Gleichheit, besser Ordnung.

```
module Set (Set
  , empty      -- Set a
  , isEmpty    -- Set a -> Bool
  , insert     -- Ord a => Set a -> a -> Set a
  , remove     -- Ord a => Set a -> a -> Set a
  , elem       -- Ord a => a -> Set a -> Bool
  , enum       -- Ord a => Set a -> [a]
) where
```

Endliche Mengen — Implementation

- Implementation durch **balancierte Bäume** (AVL-Bäume)
 - Andere Möglichkeiten: 2-3 Bäume, Rot-Schwarz-Bäume

```
type Set a = AVLTree a
```

- Ein Baum ist **ausgeglichen**, wenn
 - alle Unterbäume ausgeglichen sind, und
 - der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

- Im Knoten Höhe des Baumes an dieser Stelle

```
data AVLTree a = Null
                | Node Int
                    (AVLTree a)
                    a
                    (AVLTree a)
```

- Ausgeglichenheit ist **Invariante**.
- Alle Operationen müssen Ausgeglichenheit bewahren.
- Bei Einfügen und Löschen ggf. **rotieren**.

Simple Things First

- Leere Menge = leerer Baum

```
empty :: AVLTree a  
empty = Null
```

- Test auf leere Menge:

```
isEmpty :: AVLTree a -> Bool  
isEmpty Null = True  
isEmpty _    = False
```

Hilfsfunktionen

- **Höhe** eines Baums
 - Aus Knoten selektieren, **nicht berechnen**.

```
ht :: AVLTree a -> Int
ht Null          = 0
ht (Node h _ _ _) = h
```

- **Neuen Knoten** anlegen, Höhe aus Unterbäumen berechnen.

```
mkNode :: AVLTree a -> a -> AVLTree a -> AVLTree a
mkNode l n r = Node h l n r where
    h = 1 + max (ht l) (ht r)
```

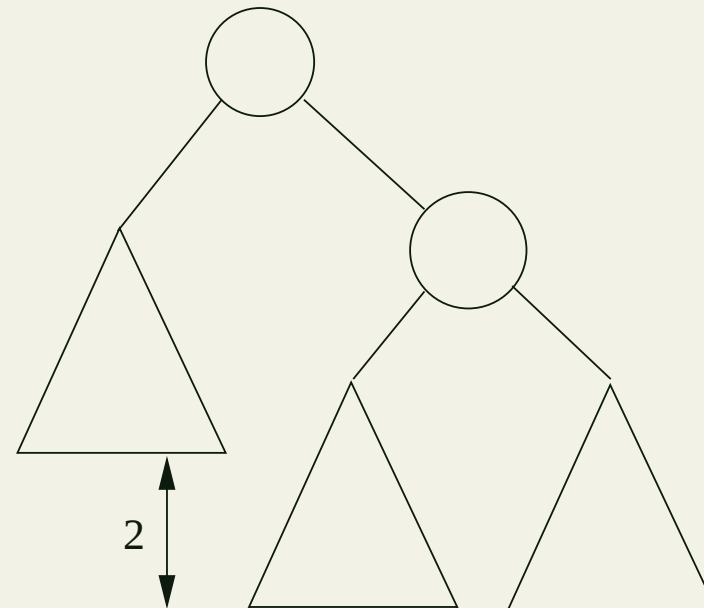
Ausgeglichenheit sicherstellen

- Problem:

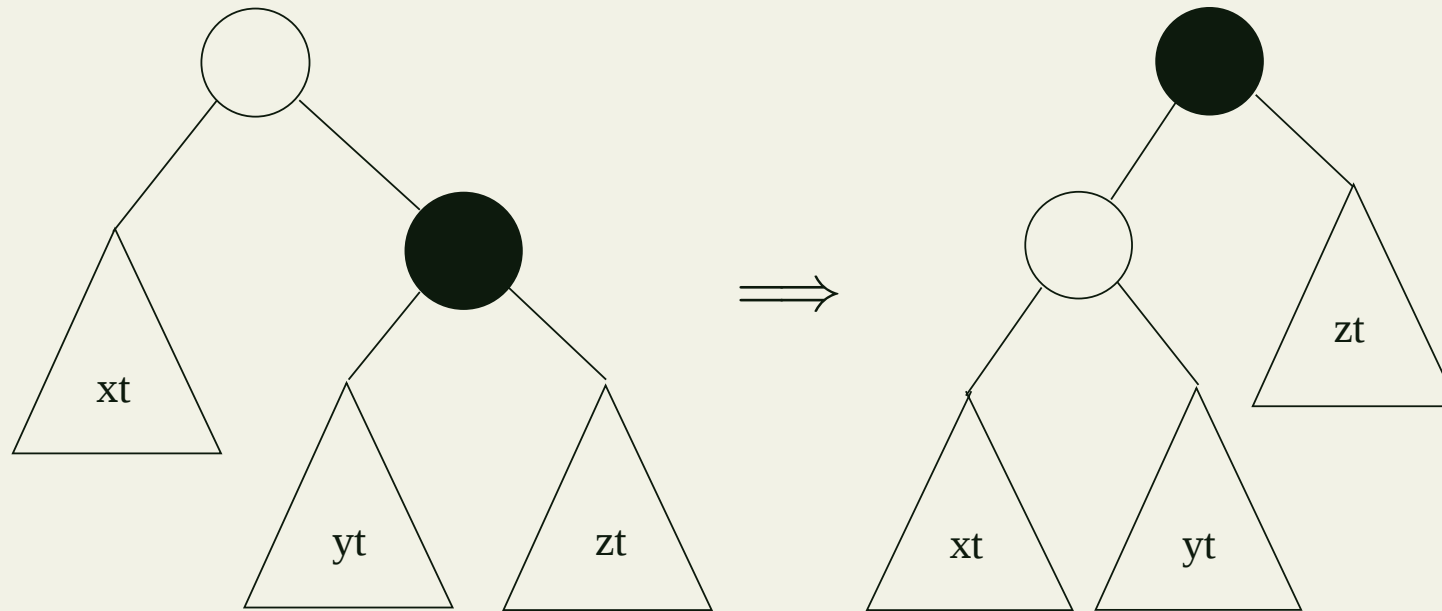
Nach Löschen oder Einfügen zu großer Höhenunterschied

- Lösung:

Rotieren der Unterbäume

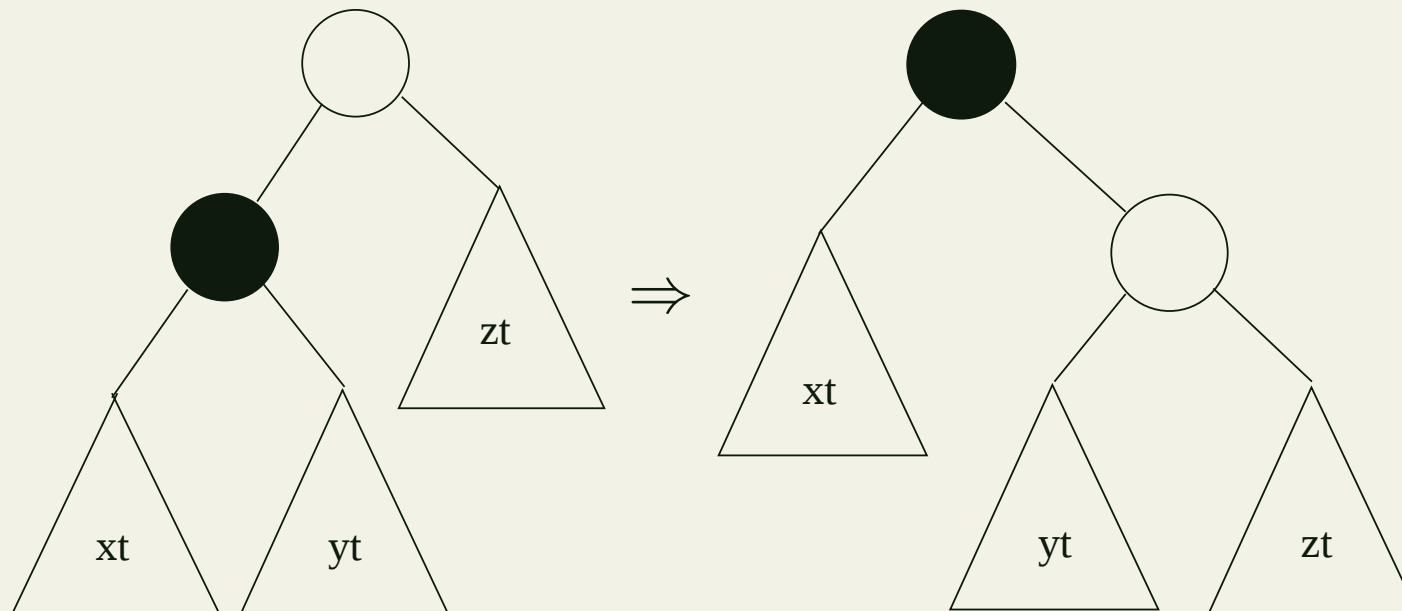


Linksrotation



```
rotl :: AVLTree a -> AVLTree a
rotl (Node _ xt y (Node _ yt x zt)) =
  mkNode (mkNode xt y yt) x zt
```

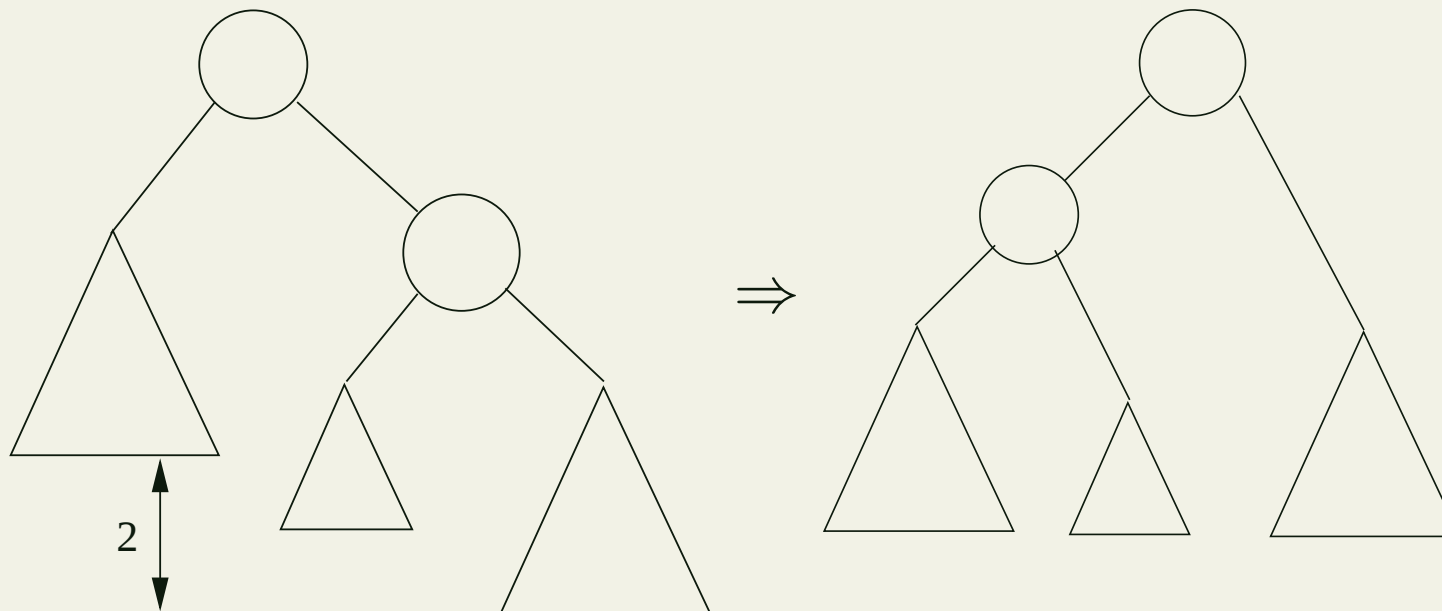
Rechtsrotation



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

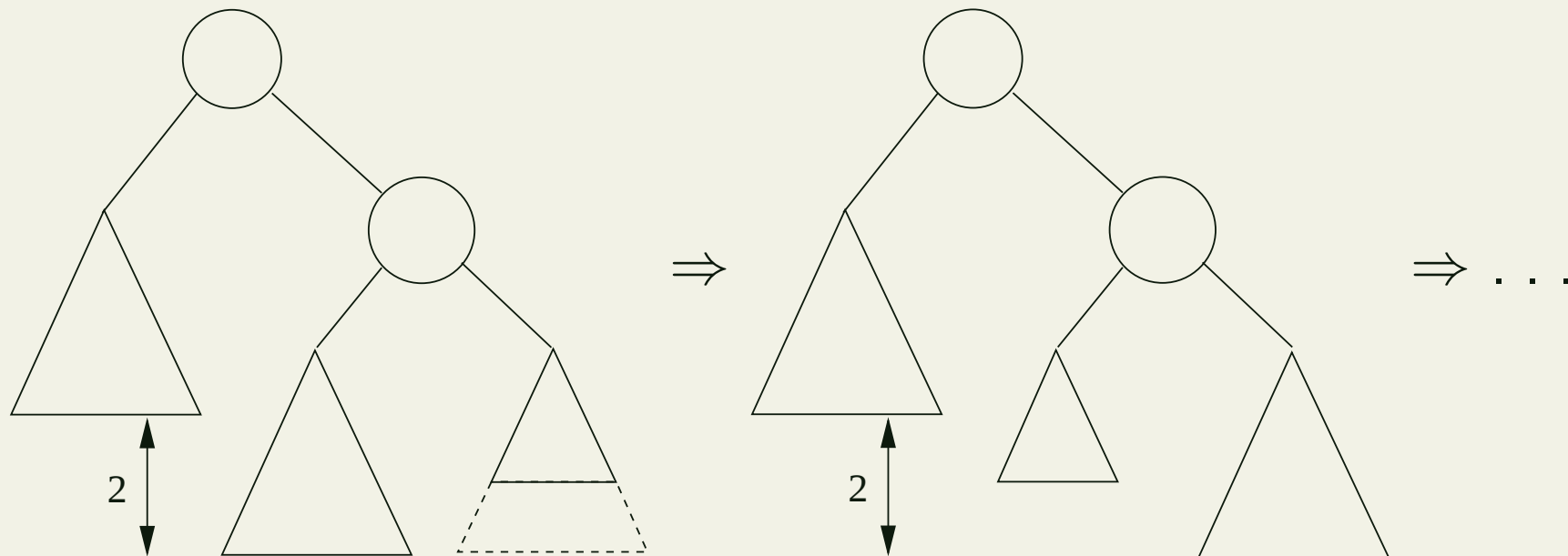
Ausgeglichenheit sicherstellen

- Fall 1: Äußerer Unterbaum zu hoch
- Lösung: Linksrotation



Ausgeglichenheit sicherstellen

- Fall 2: Innerer Unterbaum zu hoch oder gleich hoch
- Reduktion auf vorherigen Fall durch Rechtsrotation des Unterbaumes



Ausgeglichenheit sicherstellen

- Hilfsfunktion: **Balance** eines Baumes

```
bias :: AVLTree a -> Int
```

```
bias (Node _ lt _ rt) = ht lt - ht rt
```

- Zu implementieren: `mkAVL xt y zt`
 - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei;
 - Konstruiert neuen AVL-Baum mit Knoten `y`.
- Fallunterscheidung:
 - `zt` zu hoch, zwei Unterfälle:
 - ▷ Rechter Unterbaum von `zt` höher (Fall 1): `bias zt < 0`
 - ▷ Linker Unterbaum von `zt` höher/gleich hoch (Fall 2):
 $\text{bias } zt \geq 0$
 - `xt` zu hoch, zwei Unterfälle (symmetrisch).

- Konstruktion eines neuen AVL-Baumes mit Knoten y
 - Voraussetzung: Höhenunterschied xt , zt höchstens zwei;

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

Ordnungserhaltendes Einfügen

- Erst Knoten einfügen, dann ggf. rotieren:

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

- mkAVL garantiert Ausgeglichenheit.

Löschen

- Erst Knoten löschen, dann ggf. rotieren:

```
remove :: Ord a => AVLTree a -> a -> AVLTree a
remove Null x = Null
remove (Node h l y r) x
  | x < y  = mkAVL (remove l x) y r
  | x == y = join l r
  | x > y  = mkAVL l y (remove r x)
```

- mkAVL garantiert Ausgeglichenheit.

- `join` fügt zwei Bäume ordnungserhaltend zusammen (s. letzte VL)

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

Menge aufzählen

- Enthaltensein:

```
elem :: Ord a => a -> AVLTree a -> Bool
elem x Null = False
elem x (Node _ lt a rt) | x == a = True
                        | x < a  = x 'elem' lt
                        | x > a  = x 'elem' rt
```

- Mengen aufzählen:

```
foldT :: (a -> b -> b -> b) -> b -> AVLTree a -> b
foldT f e Null = e
foldT f e (Node _ l a r) =
    f a (foldT f e l) (foldT f e r)
```

- Aufzählen der Menge: Inorder-Traversion (via foldT)

Testen.

ADTs vs. Objekte

- ADTs (z.B. Haskell): **Typ** plus **Operationen**
- Objekte (z.B. Java): **Interface**, **Methoden**.
- **Gemeinsamkeiten**: **Verkapselung** (information hiding) der Implementation
- **Unterschiede**:
 - Objekte haben **internen Zustand**, ADTs sind **referentiell transparent**;
 - Objekte haben **Konstruktoren**, ADTs nicht (Konstruktoren nicht unterscheidbar)
 - **Vererbungsstruktur** auf Objekten (**Verfeinerung** für ADTs)
 - Java: **interface** eigenes Sprachkonstrukt, Haskell: Signatur eines Moduls nicht (aber z.B. SML).

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: Implementation durch ausgeglichene Bäume

Vorlesung vom 13.12.2004: Verzögerte Auswertung und unendliche Datenstrukturen

Inhalt

- Verzögerte Auswertung
 - . . . und was wir davon haben.
- Unendliche Datenstrukturen
 - . . . und wozu sie nützlich sind.
- Fallbeispiel: Parserkombinatoren



Verzögerte Auswertung

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

```
silly x y = y; double x = x+ x
```

```
silly (double 3) (double 4)
```

Verzögerte Auswertung

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

```
silly x y = y; double x = x+ x
```

```
silly (double 3) (double 4)  $\rightsquigarrow$  double 4
```

Verzögerte Auswertung

- Auswertung: **Reduktion** von Gleichungen
 - Strategie: Von **außen** nach **innen**; von **links** nach **rechts**.
- Effekt: *call-by-need* Parameterübergabe, **nicht-strikt**
 - Beispiel:
$$\text{silly } x \ y = y; \text{ double } x = x + x$$
$$\text{silly } (\text{double } 3) \ (\text{double } 4) \rightsquigarrow \text{double } 4 \rightsquigarrow 4 + 4 \rightsquigarrow 8$$
 - Erstes Argument von **silly** wird **nicht** ausgewertet.
 - Zweites Argument von **silly** wird erst im Funktionsrumpf ausgewertet.

Strikttheit

- Funktion f ist **strikt** in einem **Argument** x gdw.

$$x \equiv \perp \implies f(x) \equiv \perp$$

- Beispiele:
 - $(+)$ strikt in beiden Argumenten.
 - $(\&\&)$ strikt im ersten, nicht-strikt im zweiten.
 $\text{False} \ \&\& \ \text{error} \rightsquigarrow \text{False}$
 - silly nicht-strikt im ersten. $\text{silly error } 3 \rightsquigarrow 3$
 - Zeigen.
- **Haskell** ist
 - als nicht-strikt **spezifiziert**;
 - (meist) mit verzögerter Auswertung **implementiert**.

Effekte der Nichtstrikttheit

Datenorientierte Programme

- Berechnung der Summe von 1 bis n :

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

Effekte der Nichtstrikttheit

Datenorientierte Programme

- Berechnung der Summe von 1 bis n :

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

$\implies$ Es werden **keine** Listen von Zwischenergebnissen erzeugt.

Effekte der Nichtstrikttheit

Datenorientierte Programme

- Berechnung der Summe von 1 bis n :

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

⇒ Es werden **keine Listen** von **Zwischenergebnissen** erzeugt.

- Minimum einer Liste durch

```
min' xs = head (msort xs)
```

Effekte der Nichtstrikttheit

Datenorientierte Programme

- Berechnung der Summe von 1 bis n :

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

⇒ Es werden **keine** Listen von Zwischenergebnissen erzeugt.

- Minimum einer Liste durch

```
min' xs = head (msort xs)
```

⇒ Liste wird **nicht** vollständig sortiert — binäre Suche.

- **Rucksackproblem:** Elemente auswählen, so dass Wert maximiert:

```
select :: [(Double, Double)] -> Double -> [Int]
select elts w =
    let wgh i = fst (elts !! (i-1))
        val i = snd (elts !! (i-1))
        permissible seq = sum (map wgh seq) <= w
        better seq1 seq2 = compare (sum (map val seq2))
                                   (sum (map val seq1))
    in head (sortBy better
                (filter permissible (seqs (length elts))))

seqs :: Int -> [[Int]]
seqs 0 = [[]]
seqs n = seqs (n-1) ++ map (n :) (seqs (n-1))
```

Unendliche Datenstrukturen: Ströme

- **Ströme** sind **unendliche Listen**.

- Unendliche Liste `[2,2,2,...]`

```
twos = 2 : twos
```

- Liste der 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.

Zeigen.

```
cycle (trace "Foo!" "x")
```

- **Nützlich** für Listen mit **unbekannter Länge**

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis 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))
```

- **Keine** Rekursionsverankerung (vgl. **alte Version**)

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

Testen.

Bsp: Fibonacci-Zahlen

- Aus der Kaninchenzucht.
- Sollte jeder Informatiker kennen.

```
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:

<code>fibs</code>	1	1	2	3	5	8	13	21	34	55
<code>tail fibs</code>	1	2	3	5	8	13	21	34	55	
<code>tail (tail fibs)</code>	2	3	5	8	13	21	34	55		

- Damit ergibt sich:

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

<code>fibs</code>	1	1	2	3	5	8	13	21	34	55
<code>tail fibs</code>	1	2	3	5	8	13	21	34	55	
<code>tail (tail fibs)</code>	2	3	5	8	13	21	34	55		

- Damit ergibt sich:

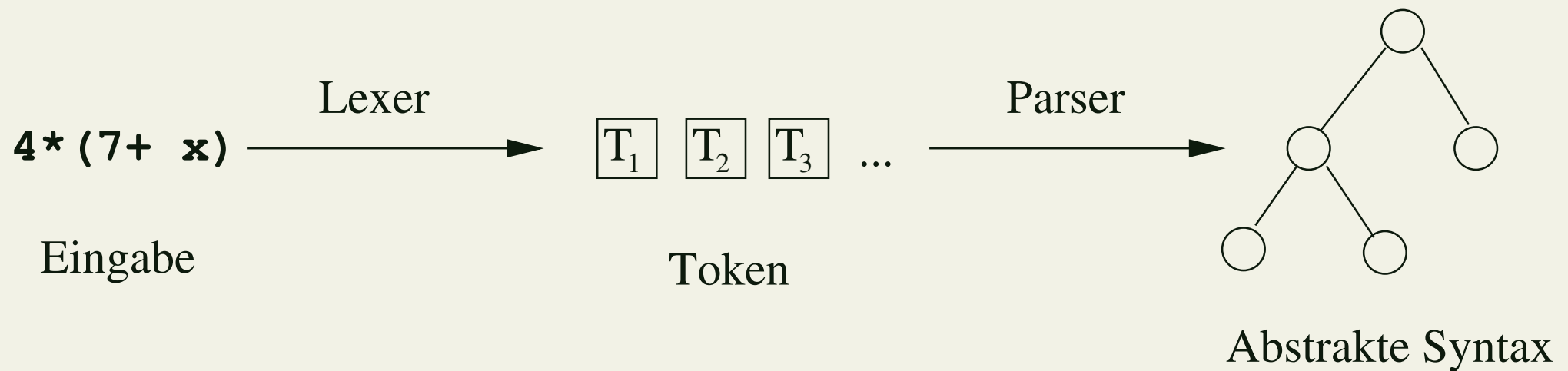
```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- **Aufwand:** **linear**, da `fibs` nur einmal ausgewertet wird.
- `n`-te Fibonaccizahl mit `fibs !! n`

Zeigen.

Fallstudie: Parsierung

- Gegeben: Grammatik
- Gesucht: Funktion, Wörter der Grammatik erkennt



Parser

- **Parser** bilden Eingabe auf Parsierungen ab
 - Mehrere Parsierungen möglich
 - Backtracking möglich
 - Durch verzögerte Auswertung dennoch effizient
- **Basisparser** erkennen Terminalsymbole
- **Parserkombinatoren** erkennen Nichtterminalsymbole
 - Sequenzierung (erst A , dann B)
 - Alternierung (entweder A oder B)
 - Abgeleitete Kombinatoren (z.B. Listen A^* , nicht-leere Listen A^+)

Grammatik für Arithmetische Ausdrücke

$$\textit{Expr} ::= \textit{Term} + \textit{Term} \mid \textit{Term} - \textit{Term} \mid \textit{Term}$$
$$\textit{Term} ::= \textit{Factor} * \textit{Factor} \mid \textit{Factor} / \textit{Factor} \mid \textit{Factor}$$
$$\textit{Factor} ::= \textit{Number} \mid \textit{Variable} \mid (\textit{Expr})$$
$$\textit{Number} ::= \textit{Digit}^+$$
$$\textit{Digit} ::= 0 \mid \dots \mid 9$$
$$\textit{Var} ::= \textit{Char}^+$$
$$\textit{Char} ::= \texttt{a} \mid \dots \mid \texttt{z} \mid \texttt{A} \mid \dots \mid \texttt{Z}$$

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik **abstrakte Syntax** (siehe **altes Beispiel**)

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik **abstrakte Syntax** (siehe altes Beispiel)

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

- Hier Unterscheidung Term, Factor, Number unnötig.

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über Eingabetyp (Token) **a** und Ergebnis **b**

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über Eingabetyp (Token) **a** und Ergebnis **b**
- Müssen mehrdeutige Ergebnisse modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über Eingabetyp (Token) **a** und Ergebnis **b**
- Müssen mehrdeutige Ergebnisse modellieren
- Müssen Rest der Eingabe modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über Eingabetyp (Token) **a** und Ergebnis **b**
- Müssen mehrdeutige Ergebnisse modellieren
- Müssen Rest der Eingabe modellieren

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

Basisparser

- Erkennt nichts:

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

- Erkennt alles:

```
succeed :: b -> Parse a b  
succeed 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 []
```

Basiskombinatoren

- Alternierung:

```
infixl 3 'alt'
```

```
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

```
infixl 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]
```

Basiskombinatoren

- Eingabe weiterverarbeiten:

```
infix 4 'use'
use :: Parse a b-> (b-> c)-> Parse a c
use p f inp = [(f x, r) | (x, r)<- p inp]
```

- Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse a b-> Parse a c-> Parse a c
(>*) :: Parse a b-> Parse a c-> Parse a b
p1 *> p2 = p1 >*> p2 'use' snd
p1 >* p2 = p1 >*> p2 'use' fst
```

Abgeleitete Kombinatoren

- Listen: $A^* ::= AA^* \mid \varepsilon$

```
list :: Parse a b-> Parse a [b]
```

```
list p = p >*> list p 'use' uncurry (:)
        'alt' succeed []
```

- Nicht-leere Listen: $A^+ ::= AA^*$

```
some :: Parse a b-> Parse a [b]
```

```
some p = p >*> list p 'use' uncurry (:)
        'alt' succeed []
```

- NB. Präzedenzen: $>*>$ (5) vor `use` (4) vor `alt` (3)

Einschub: Präzedenzen der Operatoren in Haskell

Höchste Priorität: Funktionsapplikation

```
infixr 9      .
infixr 8      ^, ^^, **
infixl 7      *, /, 'quot', 'rem', 'div', 'mod'
infixl 6      +, -
infixr 5      :
infix  4      ==, /=, <, <=, >=, >
infixr 3      &&
infixr 2      ||
infixl 1      >>, >>=
infixr 1      =<<
infixr 0      $, $!, 'seq'
```

Parsierung Arithmetische Ausdrücke

- Token: Char
- Parsierung von Expr

```
pExpr :: Parse Char Expr
```

```
pExpr = pTerm >* token '+' >*> pTerm 'use' uncurry Plus
      'alt' pTerm >* token '-' >*> pTerm 'use' uncurry Minus
      'alt' pTerm
```

- Parsierung von Term

```
pTerm :: Parse Char Expr
```

```
pTerm = pFactor >* token '*' >*> pFactor 'use' uncurry Times
      'alt' pFactor >* token '/' >*> pFactor 'use' uncurry Div
      'alt' pFactor
```

Der Kern des Parsers

- Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isDigit) 'use' Number. read
        'alt' some (spot isAlpha) 'use' Var
        'alt' token '(' *> pExpr >* token ')'
```

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen:
 - Parsierung konsumiert Eingabe
 - Keine Mehrdeutigkeit

Testen.

```
parse :: String -> Expr
parse i =
  case filter (null . snd)
    (pExpr (filter (not.isSpace) i)) of
    [] -> error "Input does not parse."
    [(e, _)] -> e
    _ -> error "Input is ambiguous."
```

Ein kleiner Fehler

- **Mangel:** $3+4+5$ ist Syntaxfehler — Fehler in der **Grammatik**
- Behebung: leichte Änderung der Grammatik . . .

$$\begin{aligned} Expr &::= Term + Expr \mid Term - Expr \mid Term \\ Term &::= Factor * Term \mid Factor / Term \mid Factor \\ Factor &::= Number \mid Variable \mid (Expr) \\ Number &::= Digit^+ \\ Digit &::= 0 \mid \dots \mid 9 \\ Var &::= Char^+ \\ Char &::= a \mid \dots \mid z \mid A \mid \dots \mid Z \end{aligned}$$

- (vergleiche alt)
- Abstrakte Syntax bleibt . . .

- Entsprechende Änderung des Parsers in `pExpr`

```
pExpr :: Parse Char Expr
```

```
pExpr = pTerm >* token '+' >*> pExpr 'use' uncurry Plus
      'alt' pTerm >* token '-' >*> pExpr 'use' uncurry Minus
      'alt' pTerm
```

- ... und in `pTerm`:

```
pTerm :: Parse Char Expr
```

```
pTerm = pFactor >* token '*' >*> pTerm 'use' uncurry Times
      'alt' pFactor >* token '/' >*> pTerm 'use' uncurry Div
      'alt' pFactor
```

- (vergleiche `alt`)

- `pFactor` und Hauptfunktion bleiben:

Testen.

Zusammenfassung Parserkombinatoren

- **Systematische Konstruktion** des Parsers aus der Grammatik.
- **Abstraktion** durch Funktionen höherer Ordnung.
- Durch verzögerte Auswertung **annehmbare Effizienz**:
 - Grammatik muß **eindeutig** sein (LL(1) o.ä.)
 - Vorsicht bei **Mehrdeutigkeiten**!
 - Effizient implementierte **Büchereien** mit **gleicher Schnittstelle** auch für **große Eingaben** geeignet.

Zusammenfassung

- Verzögerte Auswertung erlaubt unendliche Datenstrukturen
 - Zum Beispiel: Ströme (unendliche Listen)
- Parserkombinatoren:
 - Systematische Konstruktion von Parsern
 - Durch verzögerte Auswertung annehmbare Effizienz

Vorlesung vom 03.01.2005: Ein/Ausgabe in Funktionalen Sprachen

Inhalt

- Wo ist das Problem?
- Aktionen und der Datentyp *IO*.
- Vordefinierte Aktionen
- Beispiel: *Nim*
- Aktionen als Werte

Ein- und Ausgabe in funktionalen Sprachen

Problem: Funktionen mit Seiteneffekten nicht referentiell transparent.

- z. B. `readString :: ... -> String ??`

Ein- und Ausgabe in funktionalen Sprachen

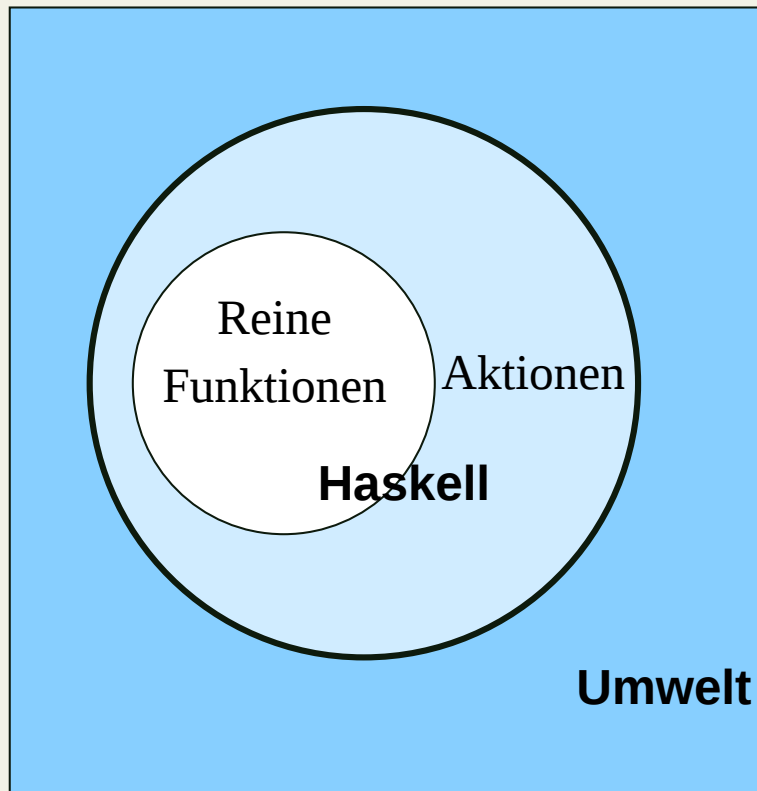
Problem: Funktionen mit Seiteneffekten nicht referentiell transparent.

- z. B. `readString :: ... -> String ??`

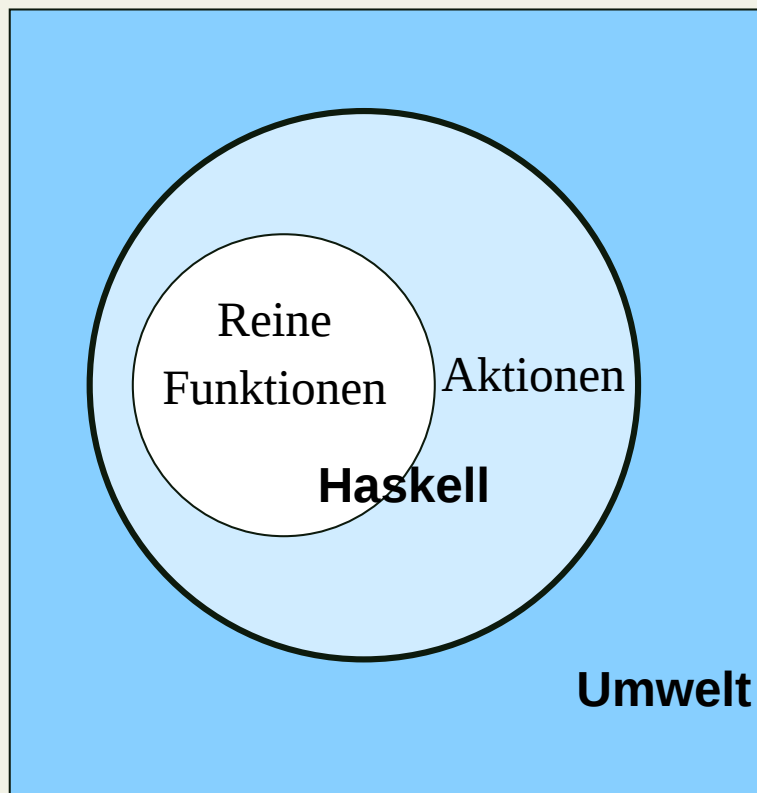
Lösung: Seiteneffekte am Typ `IO` erkennbar — **Aktionen**

- **Aktionen** können **nur** mit **Aktionen** komponiert werden
- „einmal IO, immer IO“

Aktionen als abstrakter Datentyp



Aktionen als abstrakter Datentyp



```
type IO t
```

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

```
return :: a-> IO a
```

Vordefinierte Aktionen (Prelude)

- Zeile von `stdin` lesen:

```
getLine  :: IO String
```

- Zeichenkette auf `stdout` ausgeben:

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

- Zeichenkette mit Zeilenvorschub ausgeben:

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

Einfache Beispiele

- Echo einfach:

```
echo1 :: IO ()  
echo1 = getLine >>= putStrLn
```

Einfache Beispiele

- Echo einfach:

```
echo1 :: IO ()  
echo1 = getLine >>= putStrLn
```

- Echo mehrfach:

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

Einfache Beispiele

- Echo einfach:

```
echo1 :: IO ()  
echo1 = getLine >>= putStrLn
```

- Echo mehrfach:

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

- Umgekehrtes Echo:

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

Die do-Notation

- Vordefinierte Abkürzung:

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

- Syntaktischer Zucker für IO:

<pre>echo = getLine >>= \s-> putStrLn s >> echo</pre>	$\iff$	<pre>echo = do s<- getLine putStrLn s echo</pre>
--	--------	---

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

Module in der Standardbibliothek

- Ein/Ausgabe, Fehlerbehandlung (Modul `IO`)
- Zufallszahlen (Modul `Random`)
- Kommandozeile, Umgebungsvariablen (Modul `System`)
- Zugriff auf das Dateisystem (Modul `Directory`)
- Zeit (Modul `Time`)

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 Standardbibliothek

- Buffered/Unbuffered, Seeking, &c.
- Operationen auf **Handle**

Beispiel: Zeichen, Worte, Zeilen zählen (`wc`)

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

- Testen.

Beispiel: Zeichen, Worte, Zeilen zählen (wc)

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

- Testen.
- Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: wc verbessert.

```
wc' :: String-> IO ()
wc' file =
  do c<- readFile file
     putStrLn (show (cnt (0, 0, 0) False c)++
                " lines, words, characters.")
cnt :: (Int, Int, Int)-> Bool-> String-> (Int, Int, Int)
cnt (l, w, c) _ [] = (l, w, c)
cnt (l, w, c) blank (x:xs)
  | isSpace x && not blank = cnt (l', w+1, c+1) True xs
  | isSpace x && blank      = cnt (l', w, c+1) True xs
  | otherwise              = cnt (l, w, c+1) False xs where
    l' = if x == '\n' then l+1 else l
```

- Datei wird verzögert gelesen und dabei verbraucht.

Ein längeres Beispiel: Nim revisited

- Benutzerschnittstelle von Nim:
- Am Anfang Anzahl der Hölzchen auswürfeln.
- Eingabe des Spielers einlesen.
- Wenn nicht zu gewinnen, aufgeben, ansonsten ziehen.
- Wenn ein Hölzchen übrig ist, hat Spieler verloren.

Alles Zufall?

- Zufallswerte: Modul Random, Klasse Random

```
class Random a where
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a
```

- Warum ist randomIO Aktion?
 - Referentielle Transparenz erlaubt keinen Nichtdeterminismus.
- Instanzen von Random: Basisdatentypen.
- Random enthält ferner
 - Zufallsgeneratoren für Pseudozufallszahlen.
 - Unendliche Listen von Zufallszahlen.

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

- Start des Spiels mit n Hölzchen

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

Benutzereingabe

- Zu implementieren: Benutzereingabe

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

- Nicht sehr befriedigend: Abbruch bei falscher Eingabe.

Fehlerbehandlung

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

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

- Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- Fangbare Benutzerfehler mit

```
userError :: String -> IOError
```

- IOError kann analysiert werden— Auszug aus Modul IO:

```
isDoesNotExistError :: IOError -> Bool
```

```
isIllegalOperation :: IOError -> Bool
```

```
isPermissionError :: IOError -> Bool
```

```
isUserError :: IOError -> Bool
```

```
ioeGetErrorString :: IOError -> String
```

```
ioeGetFileName :: IOError -> Maybe FilePath
```

- read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a => String -> IO a
```

Robuste Eingabe

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

Haupt- und Startfunktion

- Begrüßung,
- Anzahl Hölzchen auswürfeln,
- Spiel 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.
- Besser als jede imperative Sprache.

Beispiel: Kontrollstrukturen

- Endlosschleife:

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

```
forever a = a >> forever a
```

Beispiel: Kontrollstrukturen

- Endlosschleife:

```
forever :: IO a -> IO a
forever a = a >> forever 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 von Aktionen sequenzieren:

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

- Sonderfall: `[()]` als `()`

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

Map und Filter für Aktionen

- Map für Aktionen:

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```
mapM f = sequence . map f
```

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

```
mapM_ f = sequence_ . map f
```

- Filter für Aktionen

- Importieren mit `import Monad (filterM).`

```
filterM :: (a -> IO Bool) -> [a] -> IO [a]
```

Beispiel

- Führt Aktionen zufällig oft aus:

```
atmost :: Int -> IO a -> IO [a]
atmost most a =
  do l <- randomRIO (1, most)
     sequence (replicate l a)
```

Beispiel

- Führt Aktionen zufällig oft aus:

```
atmost :: Int -> IO a -> IO [a]
atmost most a =
  do l <- randomRIO (1, most)
     sequence (replicate l a)
```

- Zufälligen String:

```
randomStr :: IO String
randomStr = atmost 10 (randomRIO ('a', 'z'))
```

Zeigen.

Zusammenfassung

- Ein/Ausgabe in Haskell durch **Aktionen**
 - Aktionen (Typ `IO a`) sind seiteneffektbehaftete Funktionen
 - Komposition von Aktionen durch
$$\begin{aligned}(>>=) &:: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b \\ \text{return} &:: a \rightarrow IO\ a\end{aligned}$$
 - `do`-Notation
- Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- Verschiedene Funktionen der Standardbibliothek:
 - Prelude: `getLine`, `putStr`, `putStrLn`
 - Module: `IO`, `Random`,

**Vorlesung vom 10.01.2005:
Effizienzaspekte in der
funktionalen Programmierung**

Inhalt

- Zeitbedarf: Endrekursion — `while` in Haskell
- Platzbedarf: Speicherlecks
- Verschiedene andere Performancefallen:
 - Überladene Funktionen
 - Listen
- “Usual disclaimers apply.”

Effizienzaspekte

- Erste Lösung: bessere **Algorithmen**.
- Zweite Lösung: **Büchereien** nutzen.
- Effizienzverbesserungen durch
 - **Endrekursion**: Iteration in Haskell
 - **Striktheit**: **Speicherlecks** vermeiden
 - Der ewige Konflikt: **Geschwindigkeit** vs. **Platz**
- Effizienz **muss nicht** im Vordergrund stehen.

Endrekursion

Eine Funktion ist **endrekursiv**, wenn kein rekursiver Aufruf in einem geschachtelten Ausdruck steht.

- D.h. darüber **nur** `if`, `case`, oder Fallunterscheidungen.
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in **Sprung** oder **Schleife** übersetzt.
- Nur **nicht-endrekursive** Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` **nicht** endrekursiv:

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

- `fac` endrekursiv:

```
fac :: Integer -> Integer
fac n = fac0 n 1 where
    fac0 n acc = if n == 0 then acc
                  else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` nicht. (Zeigen)

Beispiele

- Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] -> [a]
```

```
rev' [] = []
```

```
rev' (x:xs) = rev' xs ++ [x]
```

- Hängt auch noch hinten an — $O(n^2)$!

Beispiele

- Liste umdrehen, **nicht** endrekursiv:

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

- Hängt auch noch hinten an — $O(n^2)$!

- Liste umdrehen, endrekursiv und $O(n)$: (Zeigen)

```
rev :: [a] -> [a]
rev xs = rev0 xs [] where
    rev0 []      ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```

Überführung in Endrekursion

- Gegeben eine Funktion $f' f': S \rightarrow T$

$$f' x = \text{if } B x \text{ then } H x$$

$$\text{else } \phi (f' (K x)) (E x)$$
 - Mit $K: S \rightarrow S$, $\phi: T \rightarrow T \rightarrow T$, $E: S \rightarrow T$, $H: S \rightarrow T$.
- Sei ϕ assoziativ ist, $e: T$ neutrales Element
- Dann ist die endrekursive Form:

$$f: S \rightarrow T$$

$$f x = g x e \text{ where}$$

$$g x y = \text{if } B x \text{ then } \phi (H x) y$$

$$\text{else } g (K x) (\phi (E x) y)$$

Beispiel

- Länge einer Liste (nicht-endrekursiv)

```
length' :: [a] -> Int
```

```
length' xs = if (null xs) then 0  
             else 1+ length' (tail xs)
```

- Zuordnung der Variablen:

$$\begin{array}{ll} K(x) \mapsto \text{tail} & B(x) \mapsto \text{null } x \\ E(x) \mapsto 1 & H(x) \mapsto 0 \\ \phi(x, y) \mapsto x + y & e \mapsto 0 \end{array}$$

- Es gilt: $\phi(x, e) = x + 0 = x$ (0 neutrales Element)

- Damit ergibt sich endrekursive Variante:

```
length :: [a] -> Int
length xs = len xs 0 where
    len xs y = if (null xs) then y -- was: y+ 0
               else len (tail xs) (1+ y)
```

- Allgemeines **Muster**:
 - Monoid (ϕ, e) : ϕ assoziativ, e neutrales Element.
 - Zusätzlicher Parameter **akkumuliert** Resultat.

Endrekursive Aktionen

- Nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str<- getLine
              if null str then return ""
              else do rest<- getLines'
                      return (str++ rest)
```

- Endrekursiv:

```
getLines :: IO String
getLines = getit "" where
    getit res = do str<- getLine
                  if null str then return res
                  else getit (res++ str)
```

Fortgeschrittene Endrekursion

- **Akkumulation** von Ergebniswerten durch **closures**
 - **closure**: partiell applizierte Funktion
- Beispiel: die Klasse **Show**
 - Nur Methode **show** wäre zu langsam ($O(n^2)$):

```
class Show a where  
  show :: a -> String
```

Fortgeschrittene Endrekursion

- **Akkumulation** von Ergebniswerten durch **closures**
 - **closure**: partiell applizierte Funktion
- Beispiel: die Klasse **Show**
 - Nur Methode **show** wäre zu langsam ($O(n^2)$):

```
class Show a where  
  show :: a -> String
```
 - Deshalb zusätzlich

```
showsPrec :: Int -> a -> String -> String  
show x      = showsPrec 0 x ""
```
 - String wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

- Beispiel: Mengen als Listen

```
data Set a = Set [a]
```

```
instance Show a => Show (Set a) where
```

```
  showsPrec i (Set elems) =
```

```
    \r-> "{" ++ concat (intersperse ", "  
                          (map show elems)) ++ "}" ++ r
```

- Beispiel: Mengen als Listen

```
data Set a = Set [a]

instance Show a => Show (Set a) where
  showsPrec i (Set elems) =
    \r-> "{" ++ concat (intersperse ", "
                          (map show elems)) ++ "}" ++ r
```

- Nicht perfekt — besser:

```
instance Show a => Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems []          = ("{} " ++)
    showElems (x:xs) = ('{' :) . shows x . show1 xs
      where show1 []       = ('}' :)
            show1 (x:xs) = (', ' :) . shows x . show1 xs
```

Speicherlecks

- **Garbage collection** gibt unbenutzten Speicher wieder frei.
 - Nicht mehr benutzt: Bezeichner nicht mehr im **erreichbar**
- Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
 - “Echte” Speicherlecks wie in C/C++ **nicht möglich**.
- Beispiel: **getLines**, **fac**
 - Zwischenergebnisse werden **nicht ausgewertet**. (Zeigen.)
 - Insbesondere ärgerlich bei **nicht-terminierenden Funktionen**.

Striktheit

- **Strikte Argumente** erlauben Auswertung **vor** Aufruf
 - Dadurch konstanter Platz bei Endrekursion.
- Striktheit durch erzwungene Auswertung:
 - `seq :: a -> b -> b` wertet erstes Argument aus.
 - `($!) :: (a -> b) -> a -> b` strikte Funktionsanwendung
$$f \$! x = x \text{ 'seq' } f x$$
- Fakultät in konstantem Platzaufwand (**zeigen**):

```
fac2 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:

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

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`

- 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.

- Für Monoid (ϕ, e) gilt

$$\text{foldr } \phi \ e \ l = \text{foldl } (\text{flip } \phi) \ e \ l$$

Wann welches fold?

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

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

- **Unendliche** Listen **immer** mit `foldr` falten.

Gemeinsame Teilausdrücke

- Ausdrücke werden intern durch **Termgraphen** dargestellt.
- Argument wird nie mehr als **einmal** ausgewertet: (**Zeigen**)

```
f :: Int-> Int-> Int
```

```
f x y = x+ x
```

```
test1 = f (trace "Eins\n" (3+2)) (trace "Zwei\n" (2+7))
```

```
list1 = [trace "Foo\n" (3+2), trace "Bar\n" (2+7)]
```

- **Sharing** von Teilausdrücken
 - Explizit mit **where** und **let**
 - Implizit (**ghc**) (**Beispiel**)

Memoisation

- Fibonacci-Zahlen als Strom: linearer Aufwand.

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

- **Kleine Änderung:** “unnötiger” Parameter ()

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                     (tail (fibsFn ()))
```

- **Große Wirkung:** Exponentiell. Warum?

Memoisation

- Fibonacci-Zahlen als Strom: linearer Aufwand.

```
fibs :: [Integer]
```

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- **Kleine Änderung:** “unnötiger” Parameter ()

```
fibsFn :: () -> [Integer]
```

```
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())  
                                     (tail (fibsFn ()))
```

- **Große Wirkung:** Exponentiell. Warum?
- Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.

Die Abhilfe: Memoisation

- **Memoisation**: Bereits berechnete Ergebnisse speichern.
- In Hugs: Aus dem Modul `Memo`:

```
memo :: (a -> b) -> a -> b
```

- Damit ist alles wieder gut:

```
fibsFn' :: () -> [Integer]
fibsFn' = memo (\() -> 1 : 1 : zipWith (+)
                                     (fibsFn' ())
                                     (tail (fibsFn' ())))
```

- GHC kann das automatisch.

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber **langsam**.
 - Implementierung von Typklassen: **Verzeichnis** (dictionary) von Klassenfunktionen.
 - Überladung wird zur **Laufzeit** aufgelöst.

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber **langsam**.
 - Implementierung von Typklassen: **Verzeichnis** (dictionary) von Klassenfunktionen.
 - Überladung wird zur **Laufzeit** aufgelöst.
- Bei kritischen Funktionen durch Angabe der Signatur **Spezialisierung erzwingen**.
- NB: **Zahlen** (numerische Literale) sind in Haskell **überladen**!
 - Bsp: `facts` hat den Typ `Num a => a -> a`
`facts n = if n == 0 then 1 else n * facts (n-1)`

Listen als Performance-Falle

- Listen sind **keine** Felder.
- Listen:
 - Beliebiger Länge
 - Zugriff auf n -tes Element in linearer Zeit.
 - Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- Felder:
 - Feste Länge
 - Zugriff auf n -tes Element in konstanter Zeit.
 - Abstrakt: Abbildung Index auf Daten

- Modul `Array` aus der Standardbibliothek

```
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`, `Bool`, Tupel davon, Aufzählungstypen.

Listen von Paaren sind keine endlichen Abbildungen.

- `Data.FiniteMap`: endliche Abbildungen
 - Effizient, weil durch balancierte Bäume implementiert.

```
data Ord key => FiniteMap key elt
```

- Leere Abbildung: `emptyFM` an.
- Hinzufügen mit `addToFM`, `addToFM_C`.
 - Verhalten bei Überschreiben kann spezifiziert werden.
- Auslesen mit `lookupFM`, `lookupWithDefaultFM`.

Zusammenfassung

- **Endrekursion**: `while` für Haskell.
 - Überführung in Endrekursion meist möglich.
 - Noch besser sind strikte Funktionen.
- **Speicherlecks** vermeiden: Striktheit, Endrekursion und Memoisation.
- Überladene Funktionen sind langsam.
- Listen sind keine Felder.
- Listen sind keine endliche Abbildungen.
- Effizienz muss nicht immer im Vordergrund stehen.

Vorlesung vom 17.01.2005:
Grafikprogrammierung I
Die Hugs Graphics Library

Inhalt

- Grafikprogrammierung mit HGL (**Hugs Graphics Library**)
- Einführung in die Schnittstelle, kleine Beispiele.
- Abstraktion über **HGL**:
 - Entwurf und Implementation einer kleinen Geometriebücherei.
 - Kleine Beispiele.
- 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;

- `runGraphics :: IO () -> IO ()`
führt Aktion mit Grafik aus;
- `openWindow :: Title -> Point -> IO Window`
öffnet Fenster;

- `runGraphics :: IO () -> IO ()`
führt Aktion mit Grafik aus;
- `openWindow :: Title -> Point -> IO Window`
öffnet Fenster;
- `drawInWindow :: Window -> Graphic -> IO ()`
zeichnet Grafik in Fenster;

- `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;

- `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

- `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 **X Windows (X11)** und **Microsoft Windows**.
- Bietet:
 - **Fenster**
 - verschiedene **Zeichenfunktionen**
 - Unterstützung für **Animation**
- Bietet **nicht**:
 - Hochleistungsgrafik, 3D-Unterstützung (e.g. OpenGL)
 - GUI-Funktionalität
- **Achtung:** Benötigt **altes Hugs Nov 2002!**

Übersicht HGL

- **Grafik**: wird gezeichnet
 - Atomare Grafiken
 - Modifikation mit Attributen:
 - ▷ Pinsel, Stifte und Textfarben
 - ▷ Farben
 - Kombination von Grafiken
- **Fenster**: worin gezeichnet wird
- Benutzereingaben: **Events**

Basisdatentypen

- Winkel (Grad, nicht Bogenmaß)

```
type Angle      = Double
```

- Dimensionen (Pixel)

```
type Dimension = Int
```

- Punkte (Ursprung: links oben)

```
type Point      = (Dimension, Dimension)
```

Atomare Grafiken

- **Ellipse** (gefüllt) innerhalb des gegebenen Rechtecks

`ellipse :: Point -> Point -> Graphic`

- **Ellipse** (gefüllt) innerhalb des Parallelograms:

`shearEllipse :: Point-> Point-> Point-> Graphic`

- **Bogenabschnitt** einer Ellipse (math. positiven Drehsinn):

`arc :: Point-> Point-> Angle-> Angle-> Graphic`

- Beispiel:

```
drawInWindow w (arc (40, 50) (340, 250) 45 270)
```

```
getKey w
```

```
drawInWindow w (arc (60, 50) (360, 250) (-45) 90)
```

Atomare Grafiken

- **Strecke, Streckenzug:**

```
line      :: Point -> Point -> Graphic  
polyline :: [Point] -> Graphic
```

- **Polygon** (gefüllt)

```
polygon :: [Point] -> Graphic
```

- **Text:**

```
text :: Point -> String -> Graphic
```

- **Leere Grafik:**

```
emptyGraphic :: Graphic
```

Modifikation von Grafiken

- Andere **Fonts**, **Farben**, Hintergrundfarben, . . . :

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

- Modifikatoren sind kumulativ:

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

- Unschön — Klammerwald.
- Abhilfe: (\$) :: (a-> b)-> a-> b (rechtsassoziativ)

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

Attribute

- **Konkrete** Attribute (Implementation sichtbar):

- 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

Attribute: Brush und Pen

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

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

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

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

Schriften

- **Fonts:**

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

- Größe (Breite, Höhe), Winkel (nicht unter X), Fett, Kursiv, Name.
- **Portabilität** beachten — keine exotischen Kombinationen.
- Portable Namen: `courier`, `helvetica`, `times`.
- Wenn Font nicht vorhanden: **Laufzeitfehler**
- **Beispiel.**

Quelle dazu

```
main :: IO ()
main = runGraphics $ do
  w <- openWindow "Fonts" (300, 150)
  f1<- createFont (7,14) 0 True False "Helvetica"
  drawInWindow w $
    withFont f1 (text (10, 10) "Helvetica (bold, 14 pt)")
  f2<- createFont (10, 18) 0 False True "Courier"
  drawInWindow w $
    withFont f2 (text (40, 50) "Courier (italic, 18 pt)")
  f3<- createFont (12,24) 0 False False "Times"
  drawInWindow w $
    withFont f3 (text (80, 100) "Times (24 pt)")
```

Farben

- Nützliche Abkürzung: benannte Farben

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

- Benannte Farben sind einfach `colorTable :: Array Color RGB`
- Dazu Modifikator:

```
withColor :: Color -> Graphic -> Graphic  
withColor c = withRGB (colorTable ! c)
```

Kombination von Grafiken

- Überlagerung (erste über zweiter):

```
overGraphic  ::  Graphic-> Graphic-> Graphic
```

- Verallgemeinerung:

```
overGraphics :: [Graphic]-> Graphic
```

```
overGraphics = foldr overGraphic emptyGraphic
```

Fenster

- Elementare Funktionen:

```
getGraphic :: Window -> IO Graphic
```

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

- Abgeleitete Funktionen:

- In Fenster zeichnen:

```
drawInWindow :: Window -> Graphic -> IO ()
```

```
drawInWindow w g = do
```

```
    old <- getGraphic w
```

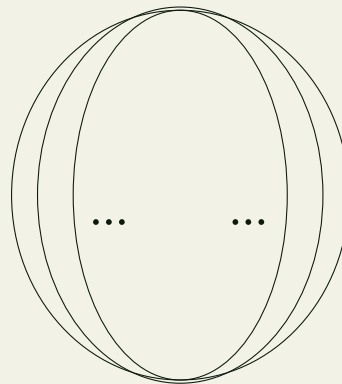
```
    setGraphic w (g 'overGraphic' old)
```

- Grafik löschen

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

Ein einfaches Beispiel

- Ziel: einen gestreiften Ball zeichnen.
- 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
```

- Hauptprogramm:

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

Eine Geometrie-Bücherei

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

Schnittstelle

- Repräsentation der Figuren:

```
data Figure -- abstract
```

- Figuren konstruieren;

```
rect      :: Dimension-> Dimension-> Figure
triangle  :: Dimension-> Angle-> Dimension-> Figure
poly      :: [Point]-> Figure
circle    :: Dimension-> Figure
```

- Figuren manipulieren:

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

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 Funktion**: Drehung um einen Punkt:

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

Implementation

- Repräsentation von **Figure**:
- **Rechtecke** und **Dreiecke** sind spezielle **Polygone**.
 - **Alles** ist ein **Polygon** oder eine **Ellipse**.
- Rotation, Translation und Skalierung **herausrechnen**.
- **Problem**: Nicht kommutativ!
 - d.h. Reihenfolge relevant.

Mathematische Grundlagen

- Lineare Algebra: Vektorraum $\mathbb{R}^2$
- Skalierung mit Faktor F : Skalarmultiplikation in $\mathbb{R}^2$
- Translation um Punkt $\vec{t}$: Addition in $\mathbb{R}^2$
- 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 Figure

- (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

Implementation von Figure

- (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
- Intern: Grundfiguren **Polygon** oder **Circle**
 - Rotation einrechnen
 - Translation und Skalierung akkumulieren

```
data Figure = Polygon Point Double [Point]
             | Circle  Point Double
```

Skalierung und Translation

- Skalierung eines Punktes (x, y) um den Faktor f :

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

```
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) :

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

```
add :: Point -> Point -> Point
```

```
add (x1, y1) (x2, y2) = (x1+ x2, y1+ y2)
```

Rotation

- 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 (x1 * cos w + y1 * sin w),
                 round (-x1 * sin w + y1 * cos w))
    where x1 = fromInt x
          y1 = fromInt y
```

Implementation der Konstruktoren

- Rechtecke:

```
rect a b = let x2= a 'div' 2; y2= b 'div' 2
           in Polygon (0,0) 1
                [(x2, y2), (-x2, y2), (-x2, -y2), (x2, -y2)]
```

- Dreiecke:

```
triangle l1 a l2 =
    Polygon (0,0) 1 [(0, 0), (l1, 0), rot a (l2, 0)]
```

- Polygone:

```
poly ps = Polygon (0, 0) 1 ps
```

- Kreise:

```
circle r = Circle (0, 0) (fromInt r)
```

Implementation der Manipulationsfunktionen:

- Rotation:

```
rotate a (Polygon t r ps) = Polygon (rot a t) r  
                                (map (rot a) ps)  
rotate a (Circle c r)      = Circle (rot a c) r
```

- Skalierung:

```
scale s (Polygon t r ps) = Polygon t (r* s) ps  
scale s (Circle c r)     = Circle c (r* s)
```

- Translation:

```
translate p (Polygon q r ps) = Polygon (add p q) r ps  
translate p (Circle c r)     = Circle (add p c) r
```

Implementation der Zeichenfunktion

- Zusätzliches Argument: Verschiebung zur Fenstermitte

```
draw :: Point-> Figure-> Graphic
```

- Polygone zeichnen:

```
draw m (Polygon q r ps) =  
    polygon (map (add m. add q. smult r) ps)
```

- Kreise zeichnen:

```
draw (x, y) (Circle (cx, cy) r) =  
    ellipse (cx+x- rad, cy+y-rad) (cx+x+ rad, cy+y+rad)  
    where rad= round r
```

Ein kleines Beispielprogramm

- Sequenz von gedrehten, skalierten Figuren.
- Im wesentlichen zwei Funktionen:
 - `drawFigs :: [Figure] -> IO` zeichnet Liste von Figuren in wechselnden Farben
 - `swirl :: Figure -> [Figure]` erzeugt aus Basisfigur unendliche Liste von gedrehten, vergrößerten Figuren

- **Liste** von 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 (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/7))
```

- 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 30 (swirl (rect 15 10)))
  drawFigs (take 50 (swirl (triangle 6 (pi/3) 6)))
```

Zusammenfassung

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

Vorlesung vom 24.01.2005:
Grafikprogrammierung II
Animation und Interaktion

Inhalt

- Erweiterung der Geometriebücherei
- Bewegte Grafiken: Animationen
- Labelled Records
- Haskell on the road

Refaktorisierung der Geometriebücherei

- Basis (des Vektorraums) `Double` statt `Int`

```
type Dimension = Double
```

- Rotation im math. `positiven` Drehsinn.

- Erweiterung Signature von `draw`:

```
draw :: G.Point-> Double-> Figure-> Graphic
```

- ... und `Kollisionserkennung`
- `Zeigen`.

Kollisionserkennung

- Ziel: Interaktion der Figuren erkennen
 - . . . zum Beispiel Überschneidung
 - 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`

Kollisionserkennung: Fälle

- Kreis und Kreis
 - Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien

Kollisionserkennung: Fälle

- Kreis und Kreis
 - Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Polygon und Kreis
 - Kollision, wenn ein Eckpunkt im Kreis

Kollisionserkennung: Fälle

- Kreis und Kreis
 - Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Polygon und Kreis
 - Kollision, wenn ein Eckpunkt im Kreis
- Polygon und Polygon
 - Kollision, wenn ein Eckpunkt des einen im anderen
 - **Voraussetzung:** Polygone konvex

Kollisionserkennung: Fälle

- Kreis und Kreis
 - Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Polygon und Kreis
 - Kollision, wenn ein Eckpunkt im Kreis
- Polygon und Polygon
 - Kollision, wenn ein Eckpunkt des einen im anderen
 - **Voraussetzung:** Polygone konvex
- **Zuerst:** Punkt und Figur, **danach** Figur und Figur

Kollisionserkennung: Punkte und Kreise

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

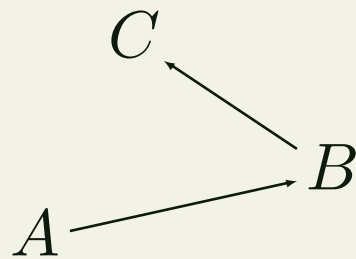
```
inC :: Point-> Double-> Point-> Bool
inC (mx, my) r (px, py) =
    len (px- mx, py- my) <= r
```

- Abstand ist **Länge** (Betrag) des Differenzvektors:

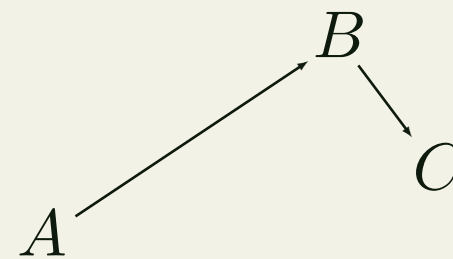
```
len :: Point-> Double
len (x, y)= sqrt (x^2+ y^2)
```

Kollisionserkennung: Punkte und Polygone

- Hilfsmittel: **Orientierung** eines Punktes
- Orientierung von C bez. Strecke $\overline{AB}$:



Positive Orientierung



Negative Orientierung

- Punkt ist innerhalb eines Polygons, wenn gleiche Orientierung bez. aller Seiten.

- Orientierung ist Vorzeichen der **Determinante**:

$$D_{A,B,C} = (B_y - A_y)(C_x - B_x) - (C_y - B_y)(B_x - A_x)$$

- Falls $D_{A,B,C} < 0$, dann Orientierung positiv
- Falls $D_{A,B,C} > 0$, dann Orientierung negativ
- Falls $D_{A,B,C} = 0$, dann Punkte in einer Flucht

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

- `signum` ist Vorzeichen

- Punkt ist innerhalb eines Polygon, wenn gleiche Orientierung bez. aller Seiten.
 - **Voraussetzung:** Polygon ist konvex
 - Punkte **auf** den Seiten werden nicht erkannt
 - Hilfsmittel: **Orientierung** eines Punktes.
- Hilfsfunktion: Liste der Seiten eines Polygons
 - Voraussetzung: Polygon geschlossen

```
sides :: [Point] -> [(Point, Point)]
sides []          = []
sides (x:[])      = []
sides (x:xs)      = (x, head xs):(sides xs)
```

- 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
```

- `nub :: Eq a => [a] -> [a]` entfernt doppelte Elemente
- Ineffizient — `length` berechnet immer Länge der ganzen Liste.

Kollisionserkennung für Punkte

- Einfache Fallunterscheidung

```
contains :: Figure -> Point -> Bool  
contains (Polygon pts) = inP pts  
contains (Circle c r) = inC c r
```

Kollisionserkennung von Formen

- Kreise: Distanz Mittelpunkte kleiner als Summe Radien

```
intersect :: Figure-> Figure-> Bool
intersect (Circle (mx1, my1) r1)
          (Circle (mx2, my2) r2)=
    len (mx2- mx1, my2- my1) <= r1+ r2
```

- Polygone: Eckpunkt des einem im anderen
 - Beachtet Randfall nicht: Polygone **berühren** sich.

```
intersect (Polygon p1) (Polygon p2)=
    any (inP p1) p2 || any (inP p2) p1
```

- Polygon und Kreis: ein Eckpunkt im Kreis
 - Beachtet Randfall nicht: Kreis schneidet **nur** Seite

```
intersect (Polygon p) (Circle c r)=  
    inP p c || any (inC c r) p  
intersect (Circle c r) (Polygon p)=  
    inP p c || any (inC c r) p
```

Polarkoordinaten

- Polarkoordinaten: $P = (r, \phi)$
- Konversion in kartesische Koordinaten: $(r, 0)$ um Winkel ϕ drehen.

```
data Polar = Polar { angle :: Angle, dist  :: Dimension }
cart  :: Polar -> Point
cart p = rot (angle p) (dist p, 0)
```

- Konversion in Polarkoordinaten: r Länge des Vektors, $\phi = \sin^{-1}(\frac{y}{r})$

```
polar :: Point -> Polar
polar (x, y)
  | (x, y) == (0, 0) = Polar {dist= r, angle = 0} -- beliebi
  | otherwise       = Polar {dist= r, angle= asin (y/r)}
                      where r = len (x, y)
```

Benutzereingaben

- Benutzerinteraktion:
 - Tasten
 - Mausbewegung
 - Mausknöpfe
 - Fenster: Größe verändern, schließen
- Grundlegende Funktionen:
 - Letzte Eingabe, auf nächste Eingabe warten:
`getWindowEvent :: Window -> IO Event`
 - Letzte Eingabe, nicht warten:
`maybeGetWindowEvent :: Window -> IO (Maybe Event)`

Benutzereingaben: Events

- 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
```

Benutzereingaben: Events

- 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 umfangreichen Datentypen

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

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

- Kommt Stückzahl oder Preis zuerst?

Probleme mit umfangreichen Datentypen

- Beispiel Warenverwaltung

- Ware mit Bezeichnung, 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 umfangreichen 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
- NB. Feldnamen sind Bezeichner
 - Nicht zwei gleiche Feldnamen im gleichen Modul
 - Felder nicht wie andere Funktionen benennen

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 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= (x, y)}) =  
  withColor Red (ellipse (x- 10, y-10) (x+10, y+10))
```

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
```

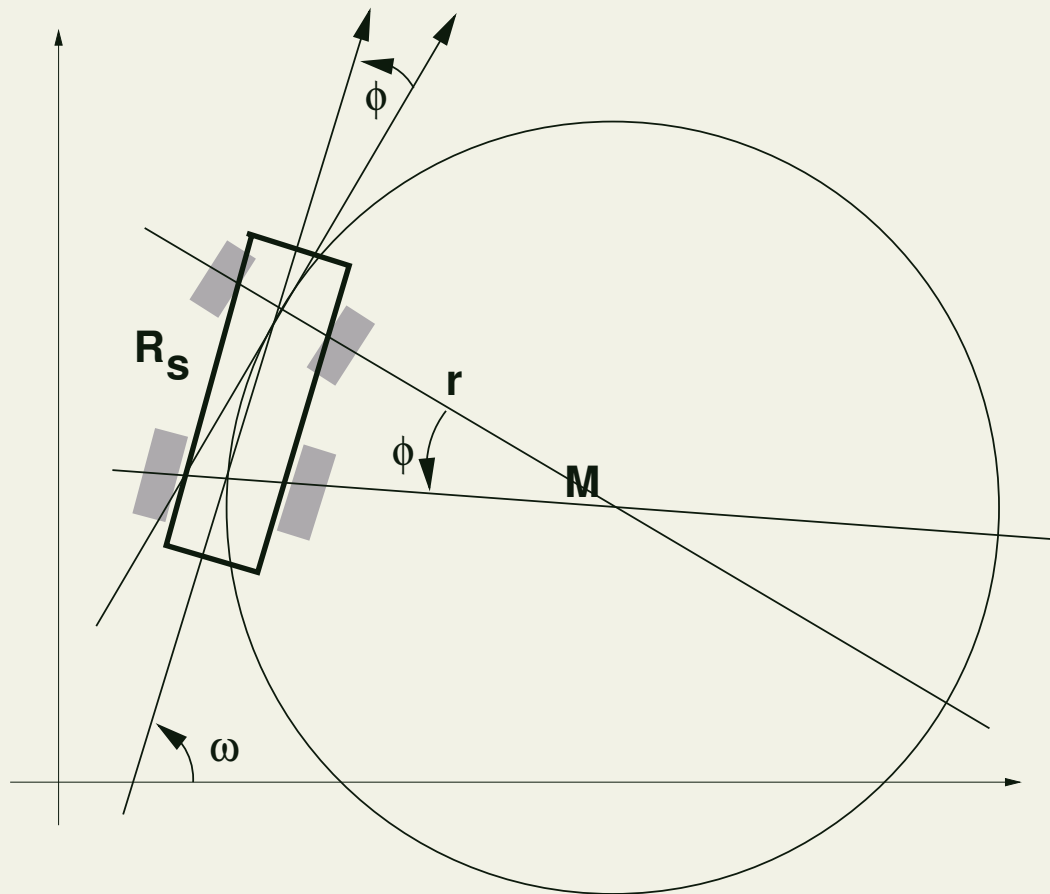
Hauptprogramm

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

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

Ein Autosimulator

Ziel: Simulation des Fahrverhaltens eines Autos



Zustand des Autos: b

- Radstand R_s
- Position $\vec{p} \in \mathbb{R}^2$
- Orientierung $\omega \in \mathbb{R}$
- Geschwindigkeit $v \in \mathbb{R}$
- Lenkwinkel $\phi \in \mathbb{R}$

Daraus

- Lenkradius $r = \frac{R_s}{\tan \phi}$
- Lenkmittelpunkt M
- Drehwinkel um M

Zustandsübergang

- Lenkradius $r = \frac{R_s}{\tan \phi}$
- Lenkwinkel $\omega_A = \omega + \phi + \frac{\pi}{2}$
- Lenkmittelpunkt $\vec{M} = \text{cart}(r, \omega_A) + \vec{p}$
- Drehung im Lenkmittelpunkt: $\zeta = \frac{v \Delta t}{r}$

Bewegung um $l = v \Delta t$ auf Kreisbahn mit $\frac{l}{2\pi r} = \frac{\zeta}{2\pi}$

Zustandsübergang

- Lenkradius $r = \frac{R_s}{\tan \phi}$
- Lenkwinkel $\omega_A = \omega + \phi + \frac{\pi}{2}$
- Lenkmittelpunkt $\vec{M} = \text{cart}(r, \omega_A) + \vec{p}$
- Drehung im Lenkmittelpunkt: $\zeta = \frac{v\Delta t}{r}$
Bewegung um $l = v\Delta t$ auf Kreisbahn mit $\frac{l}{2\pi r} = \frac{\zeta}{2\pi}$
- Neue Position: $\vec{p}' = \text{rot}(\vec{M}, \zeta, \vec{p})$
Rotation um Winkel ζ und Mittelpunkt $\vec{M}$
- Neue Orientierung: $\omega' = \omega + \zeta$
- Lenkwinkel und Beschleunigung: durch Benutzer

Benutzerinteraktion

- Konstanten a_{max} für Beschleunigung, b für Bremsbeschleunigung
- Tasten für *Schneller*, *Langsamer*, *Bremsen*
- *Schneller* drücken: Beschleunigung auf a_{max} setzen
- *Schneller* loslassen: Beschleunigung auf 0
- *Langsamer* drücken: Beschleunigung auf $-a_{max}$ setzen
- *Langsamer* loslassen: Beschleunigung auf 0 setzen
- *Bremsen* drücken: Beschleunigung auf Bremsbeschleunigung
- *Bremsen* loslassen: Beschleunigung auf 0
- Neuer Zustand:
Zustandsübergang plus Benutzerinteraktion.

Zustände

- Zustand des Autos, Gesamtzustand:

```
data Car = Car { a    :: !Double
                , brake :: !Bool
                , phi  :: !Angle
                , w    :: !Angle
                , pos  :: !Point
                , v    :: !Double
                } deriving (Show, Eq)
```

```
data State = State { car    :: !Car
                    }
```

- Zustände sind explizite Parameter

Zustandsübergang

- Sonderfall: $\phi = 0$ — Geradeausfahrt.

```
moveCar :: Car -> Car
moveCar (Car {a= a, brake= brake, phi= phi,
              w= w, pos= pos, v= v})
  | phi == 0  =
    let pos' = winMod (add pos (cart (Polar {angle= w,
                                              dist= v* deltaT})))
    in  Car {a= a, brake= brake, phi= 0, w= w,
              pos= pos', v= v'}
```

Zustandsübergang

- Normalfall: Kurvenfahrt

```
| otherwise =  
  let r      = wheelBase /tan phi -- Lenkradius  
      wa     = w+ phi+ pi/2  -- Lenkwinkel  
      m      = add (cart (Polar{angle= wa, dist= r})) pos  
                                   -- Lenkmittelpunkt  
      zeta   = v*deltaT / r -- Drehwinkel um m  
      pos'   = winMod (rotAround m zeta pos) -- neue Pos'n  
      w'     = w+ zeta -- neue Orientierung  
in Car {a= a, brake= brake, phi= phi,  
        w= w', pos= pos', v= v'}
```

Neue Geschwindigkeit

- Sonderfall: Bremsen

```
where v' = if brake then
            if v > 0 then max 0 (v - brakeAcc*deltaT)
            else 0
        else if abs v <= vMax then v + a*deltaT
        else v
```

- Ansonsten: linearer Anstieg bis v_{max}

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
```

- Alle vorliegenden Events lesen:

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

- Neuer Zustand: Events bearbeiten, Auto bewegen

```
nextState :: [Event]-> State-> IO State
nextState evs s = return
  (setCar moveCar (foldl (flip procEv) s evs))
```

Event bearbeiten:

```
procEv :: Event -> State -> State
procEv (Key {keysym= k, isDown=down})
    | k 'isKey' ' ' && down      = setBrake True
    | k 'isKey' ' ' && not down  = setBrake False
    | isUpKey k && down          = setAcc accMax
    | isUpKey k && not down      = setAcc 0
    | isDownKey k && down        = setAcc (- accMax)
    | isDownKey k && not down    = setAcc 0
procEv (Button{isLeft= True, isDown= True}) = setBrake True
procEv (Button{isLeft= True, isDown= False}) = setBrake False
procEv (MouseMove {pt= (x, y)}) = setAng (mouseToAngle x)
procEv _ = id
```

- Hilfsfunktion: prüft Key

```
isKey :: Key-> Char-> Bool
isKey k c = isCharKey k && keyToChar k == c
```

- Hilfsfunktion: Mausposition in Lenkwinkel

```
mouseToAngle :: Int-> Double
mouseToAngle x =
  let x2= fromInt (winX `div` 2)
  in  asin ((fromInt x - x2) / x2)* phiMax / pi
```

Hauptfunktion

- Window öffnen, Hauptschleife starten

```
main :: IO ()
main = runGraphics $
    do w<- openWindowEx "Haskell Racer!" Nothing
        (winX, winY) DoubleBuffered (Just 30)
        i<- initialState
        loop w i
        closeWindow w
```

- Zeigen!

Zusammenfassung

- Refaktorisierung der Geometriebücherei
 - Basis `Double`, Polarkkoordinaten, 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 einer einfachen Autosimulation

Vorlesung vom 31.01.2005: Roboter!

In eigener Sache

Studentische Hilfskräfte gesucht!

Die AG Krieg-Brückner sucht Haskell-Programmierer!

Tätigkeitsbereich: Werkzeugentwicklung

- Formale Programmentwicklung: <http://www.tzi.de/cofi/hets>
- Benutzerschnittstellen: <http://proofgeneral.inf.ed.ac.uk/kit>

Meldet Euch bei Till Mossakowski (till@tzi.de) oder mir.

Inhalt

- Modellierung von **Zuständen**
- **Zustandsübergänge** als Monaden
- **Eingebettete Sprachen**
- Beispiel: die Roboterkontrollsprache **IRL**
 - Nach **Hudak**, Kapitel 19.

Modellierung von Aktionen

- Aktionen (IO a) sind keine schwarze Magie.
- Grundprinzip:
 - Der Systemzustand wird durch das Programm erreicht.
 - Darf dabei nie dupliziert oder vergessen werden.
 - Auswertungsreihenfolge muß erhalten bleiben.
- $\implies$ Zustandsübergangsmonaden

Zustandsübergangsmonaden

- Typ:

```
data ST s a = ST (s -> (a, s))
```

- Parametrisiert über Zustand `s` und Berechnungswert `a`.

- `IO a` ist Instanz der Typklasse `Monad`:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Komposition von Zustandsübergängen

- Im Prinzip Vorwärtskomposition ($>.>$):

$$\begin{aligned} (>>=) &:: \text{ST } s \text{ } a \rightarrow (a \rightarrow \text{ST } s \text{ } b) \rightarrow \text{ST } s \text{ } b \\ &:: (s \rightarrow (a, s)) \rightarrow (a \rightarrow s \rightarrow (b, s)) \rightarrow (s \rightarrow (b, s)) \\ &:: (s \rightarrow (a, s)) \rightarrow ((a, s) \rightarrow (b, s)) \rightarrow (s \rightarrow (b, s)) \end{aligned}$$

- Damit $f >>= g = \text{uncurry } g . f$.
- Aber: Konstruktor um ST

Komposition von Zustandsübergängen

- Hilfsfunktion zum **auspacken**.

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

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

- **Identität** für Zustandstransformationen:

Aktionen

- Aktionen: Zustandstransformationen auf der 'Welt'
- Typ `RealWorld#` repräsentiert Außenwelt
 - Typ hat genau einen Wert `realworld#`, der nur von initialem Aufruf erzeugt wird.
 - Aktionen: `type IO a = ST RealWorld# a`
- Optimierungen:
 - `ST s a` vordefiniert durch `in-place-update` implementieren;
 - `IO`-Aktionen durch einfachen Aufruf ersetzen.

IRL im Beispiel

- Alle **Roboterkommandos** haben Typ `Robot a`
 - `Robot a` ist Instanz von `Monad`
 - Bewegung `move :: Robot ()`, `turnLeft :: Robot ()`
 - Roboter kann zeichnen: `penUp :: Robot()`, `penDown :: Robot()`
 - Bsp: `Quadrat zeichnen`

```
drawSquare =  
  do penDown; move; turnRight; move;  
    turnRight; move; turnRight; move
```
- Roboter lebt in einer einfachen Welt mit Wänden
 - **Test**, ob Feld vor uns frei: `blocked :: Robot Bool`

Kontrollstrukturen

- Bedingungen und Schleifen:

```
cond  :: Robot Bool-> Robot a-> Robot a-> Robot a
```

```
cond_ :: Robot Bool-> Robot ()-> Robot ()
```

```
while :: Robot Bool-> Robot ()-> Robot ()
```

- Bsp: Ausweichen

```
evade :: Robot ()
```

```
evade = do cond_ blocked turnRight
```

- Bsp: Auf nächste Wand zufahren

```
moveToWall :: Robot ()
```

```
moveToWall = while (isnt blocked)  
                move
```

Roboter auf Schatzsuche

- Welt enthält auch **Münzen**.

- Münzen aufnehmen und ablegen:

```
pickCoin :: Robot () , dropCoin :: Robot ()
```

- Roboter steht auf einer Münze?

```
onCoin :: Robot Bool
```

- Beispiel: Auf dem Weg Münzen sammeln (wie `moveWall`)

```
getCoinsToWall :: Robot ()
```

```
getCoinsToWall = while (isnt blocked) $  
    do move; pickCoin
```

Implementation

- Der Roboterzustand:

```
data RobotState
  = RobotState
    { position    :: Position
    , facing      :: Direction
    , pen         :: Bool
    , color       :: Color
    , treasure    :: [Position]
    , pocket      :: Int
    } deriving Show
```

- Vergleiche `Car` (letzte Woche) — hier: abstrakter

Robot a als Zustandsübergang

- Erster Entwurf:

```
type Robot a = RobotState-> (RobotState, a)
```

- Aber: brauchen die Welt (`Grid`), Roboterzustände **zeichnen**:

```
type Robot a =  
  RobotState-> Grid-> Window-> (RobotState, a, IO())
```

- Aktionen nicht erst aufsammeln, sondern gleich ausführen —
RobotState in IO einbetten.

```
data Robot a =  
  Robot (RobotState -> Grid -> Window -> IO (RobotState, a))
```

- Damit Robot a als Instanz von **Monad**.

Datentypen und Hilfsfunktionen

- **Positionen:**

```
type Position = (Int,Int)
```

- **Richtungen:**

```
data Direction = North | East | South | West  
    deriving (Eq,Show,Enum)
```

- **Drehung:**

```
right,left :: Direction -> Direction
```

- **Die Welt:**

```
type Grid = Array Position [Direction]
```

- Enthält für (x,y) Richtungen erreichbarer Nachbarfelder

Einfache Zustandsmanipulationen

- Zwei Hilfsfunktionen

```
updateState :: (RobotState -> RobotState) -> Robot ()  
queryState  :: (RobotState -> a) -> Robot a
```

- Damit Implementation der einfachen Funktionen — Bsp:

```
turnLeft  :: Robot ()  
turnLeft = updateState (\s -> s {facing = left (facing s)})
```

- Einzige Ausnahme: `blocked`.

Bewegung

- Beim Bewegen:
 - Prüfen, ob Bewegung möglich
 - Neue Position des Roboters zeichnen
 - Neue Position in Zustand eintragen.

```
move :: Robot ()
move = cond_ (isnt blocked) $
    Robot $ \s _ w -> do
        let newPos = movePos (position s) (facing s)
        graphicsMove w s newPos
        return (s {position = newPos}, ())
```

- **Geliftete** Negation: isnt :: Robot Bool-> Robot Bool

Grafik

- Weltkoordinaten (**Grid**): Ursprung (0,0) in Bildmitte
- Eine Position (**Position**) $\sim$ zehn Pixel
- Wände werden zwischen zwei Positionen gezeichnet
- Roboterstift zeichnet von einer Position zum nächsten
- Münzen: gelbe Kreise direkt links über der Position
- Münzen löschen durch übermalen mit schwarzem Kreis

Hauptfunktion

`runRobot :: Robot () -> RobotState -> Grid -> IO ()`

- Fenster öffnen
- Welt zeichnen, initiale Münzen zeichnen
- Auf Spacetaste warten
- Funktion `Robot ()` ausführen
- Aus Endzustand kleine Statistik drucken
- Zeigen.

Beispiel 1

- Roboter läuft in Spirale.
- Nach rechts drehen, n Felder laufen, nach rechts drehen, n Felder laufen;
- Dann n um eins erhöhen;
- Nützliche Hilfsfunktion:

```
for :: Int -> Robot () -> Robot ()  
for n a = sequence_ (replicate n a)
```

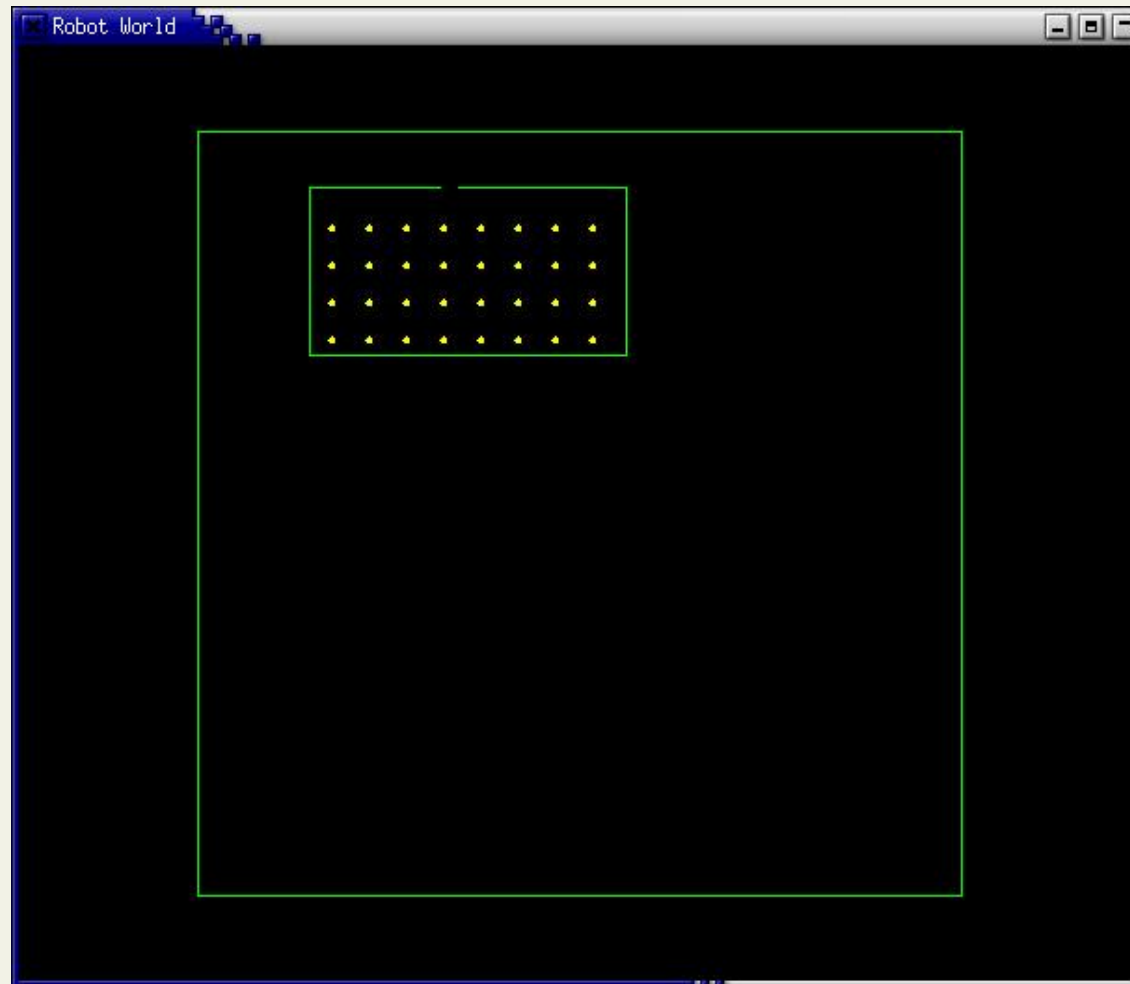
Hauptfunktion

```
spiral :: Robot ()
spiral = penDown >> loop 1 where
  loop n =
    let turnMove = do turnRight; for n move
    in  do for 2 turnMove
        cond_ (isnt blocked) (loop (n+1))

main :: IO ()
main = runRobot spiral s0 g0
```

- Zeigen.

Beispiel 2: eine etwas kompliziertere Welt



Zerlegung des Problems

- **Ziel:** in Welten wie dieser alle Münzen finden.
- Dazu: **Zerlegung** in Teilprobleme
 1. Von Startposition in **Spirale nach außen**
 2. Wenn **Wand** gefunden, **Tür** suchen
 3. Wenn **Tür** gefunden, **Raum betreten**
 4. Danach alle **Münzen** einsammeln

- Schritt 1: Variation der Spirale

```
treasureHunt :: Robot ()
treasureHunt = do
  penDown; loop 1
  where loop n =
    cond blocked findDoor $
      do turnRight; moven n
    cond blocked findDoor $
      do turnRight
        moven n; loop (n+1)
```

- Schritt 2: Tür suchen

```
findDoor :: Robot ()  
findDoor = do  
  turnLeft  
  loop  
  where loop = do  
    wallFollowRight  
    cond doorOnRight  
      (do enterRoom; getGold)  
      (do turnRight; move; loop)
```

- Hilfsfunktion 2.1: Wand folgen

```
wallFollowRight :: Robot ()  
wallFollowRight =  
    cond_ blockedRight $  
        do move; wallFollowRight
```

```
blockedRight :: Robot Bool  
blockedRight = do  
    turnRight  
    b <- blocked  
    turnLeft  
    return b
```

- Hilfsfunktion 2.2: Tür suchen, Umdrehen

```
doorOnRight :: Robot Bool
doorOnRight = do
  penUp; move
  b <- blockedRight
  turnAround; move; turnAround; penDown
  return b
```

```
turnAround :: Robot ()
turnAround = do turnRight; turnRight
```

- Schritt 3: Raum betreten

```
enterRoom :: Robot ()
```

```
enterRoom = do
```

```
    turnRight
```

```
    move
```

```
    turnLeft
```

```
    moveToWall
```

```
    turnAround
```

```
moveToWall :: Robot ()
```

```
moveToWall = while (isnt blocked)
```

```
    move
```

- Schritt 4: Alle Münzen einsammeln

```
getGold :: Robot -> ()
getGold = do
  getCoinsToWall
  turnLeft; move; turnLeft
  getCoinsToWall
  turnRight
  cond_ (isnt blocked) $
    do move; turnRight; getGold
```

- Hilfsfunktion 4.1: Alle Münzen in einer Reihe einsammeln

```
getCoinsToWall :: Robot ()  
getCoinsToWall = while (isnt blocked) $  
                    do move; pickCoin
```

- Hauptfunktion:

```
main = runRobot treasureHunt s1 g3
```

- Zeigen!

Zusammenfassung

- Zustandsübergangsmonaden
 - Aktionen als Zustandsübergänge von `RealWorld#`
- Die Roboterkontrollsprache IRL
 - Einbettung einer imperativen Sprache in Haskell
- Abstrakter als direkte Modellierung (`Robot` vs. `Car`)

Vorlesung vom 07.02.2005

Schlußbemerkungen

Inhalt der Vorlesung

- Organisatorisches
- Noch ein paar Haskell-Döntjes:
 - Concurrent Haskell
 - HaXml — XML in Haskell
 - HTk
- Rückblick über die Vorlesung
- 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 8050) abgeben (oder zum Fachgespräch mitbringen)
- Nicht vergessen: in Liste eintragen!.

Das Fachgespräch

- Dient zur **Überprüfung der Individualität der Leistung**.
 - Insbesondere: Teilnahme an Bearbeitung der Übungsblätter.
 - **Keine Prüfung**.
- Dauer: ca. 5–10 Min; einzeln, auf Wunsch mit Beisitzer
- Inhalt: Übungsblätter
- Bearbeitete Übungsblätter mitbringen — es werden mindestens zwei Aufgaben besprochen, die erste könnt Ihr Euch aussuchen.
- Termine:
 - Mo. 21.02. — Liste vor MZH 8050
 - oder nach Vereinbarung.

HaXML: XML in Haskell

- Fallbeispiel: Vorteile von Typisierung, algebraische Datentypen.
- Was ist eigentlich XML?
 - Eine Notation für polynomiale Datentypen mit viel < und >
 - Eine Ansammlung darauf aufbauender Techniken und Sprachen

HaXML: XML in Haskell

- Fallbeispiel: Vorteile von Typisierung, algebraische Datentypen.
- Was ist eigentlich **XML**?
 - Eine Notation für polynomiale Datentypen mit viel **<** und **>**
 - Eine Ansammlung darauf aufbauender Techniken und Sprachen
- XML hat mehrere Schichten:
 - Basis: Definition von semantisch strukturierten Dokumenten
 - Präsentation von strukturierten Dokumenten
 - Einheitliche Definitionen und Dialekte

XML — Eine Einführung

- Frei definierbare **Elemente** und **Attribute**
 - Vergleiche HTML
- Elemente und Attribute werden in **DTD** definiert
 - Heutzutage: Schemas, Relax NG
 - Entspricht einer Grammatik
- Beispiel: Ein Buch habe
 - Autor(en) — mindestens einen `[String]` (nichtleer)
 - Titel `String`
 - evtl. eine Zusammenfassung `[String]`
 - ISBN-Nummer `String`

Beispiel: 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

Ein 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

- HaXML: **getypte** Einbettung in Haskell

- DTD → algebraischen Datentyp
- Klasse `XmlContent`
- Funktionen

```
readXml :: XmlContent a => String -> Maybe a
```

```
showXml :: XmlContent a => a -> String
```

- Jedes Element einen Typ, Instanz von `XmlContent`

- Übersetzung der DTD: `DtdToHaskell book.dtd Book.hs`

```
module Book where

data Book = Book Book_Attrs (List1 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

- Einfache, getypte Manipulation von XML-Dokumenten
- Beispiel formatierte Ausgabe (Zeigen):

```
prt :: Book-> String
prt (Book (Book_Attrs {bookIsbn=i}) (NonEmpty as) (Title t)
      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 Text.Regex
data Regex -- abstrakt
mkRegex :: String -> Regex -- übersetzt regex
matchRegex :: Regex -> String -> Maybe [String]
```

- Damit neue Version von query (**Zeigen**):

```
query' (Library bs) ex =
    filter (any (isJust.matchRegex (mkRegex ex)).
              getWords) bs
```

Zusammenfassung HaXML

- Transformation in Haskell einfacher, typsicherer &c. als style sheets, XSLT, &c
- Durch XML beschriebene Datentypen sind **algebraisch**
- Nützlich zum **Datenaustausch** — nahtlose Typsicherheit, e.g.

Java $\longleftrightarrow$ Haskell $\longleftrightarrow$ C++

Concurrent Haskell

- **Threads** in Haskell:

```
forkIO :: IO () -> IO ThreadID  
killThread :: ThreadID -> IO ()
```

- Zusätzliche Primitive zur Synchronisation
- Erleichtert Programmierung **reaktiver Systeme**
 - Benutzerschnittstellen, Netzapplikationen, . . .
- hugs: **kooperativ**, ghc: **präemptiv**

- Beispiel: Zeigen

```
module Main where
```

```
import Concurrent
```

```
write :: Char -> IO ()
```

```
write c = putChar c >> write c
```

```
main :: IO ()
```

```
main = forkIO (write 'X') >> write '.'
```

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 req/s bis 700 req
- <http://www.haskell.org/simonmar/papers/web-server.ps.gz>

Grafische Benutzerschnittstellen

- **HTk**
 - Verkapselung von Tcl/Tk in Haskell
 - Nebenläufig mit **Events**
 - Entwickelt an der Uni Bremen (Dissertation E. Karlsen)
 - Mächtig, abstrakte Schnittstelle, rustikale Grafik
- **wxHaskell**
 - Verkapselung von **wxWidgets**
 - wxWidgets: abstrakte GUI-Klassenbibliothek
 - ▷ Implementiert für Linux/GTK, Windows, Mac OS X
 - Neueres Toolkit, ästhetischer, sehr mächtig
 - Verkapselung: Zustandsbasiert mit call-backs

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
```

- Zeigen.

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

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
- Noch viel im Fluß
 - Tools ändern sich
 - Zum Beispiel FFI/HGL

Andere Funktionale Sprachen

- **Standard ML (SML):**
 - Streng typisiert, strikte Auswertung
 - Formal definierte Semantik
 - Fünf aktiv unterstützte Compiler
 - <http://www.standardml.org/>
- **Caml, O'Caml:**
 - Streng typisiert, strikte Auswertung
 - Hocheffizienter Compiler, byte code & native
 - Nur ein Compiler (O'Caml)
 - <http://caml.inria.fr/>

Andere Funktionale Sprachen

- LISP & Scheme
 - Ungetypt/schwach getypt
 - Seiteneffekte
 - Viele effiziente Compiler, aber viele Dialekte
 - Auch industriell verwendet

Funktionale Programmierung in der Industrie

- Erlang
 - schwach typisiert, nebenläufig, strikt
 - Fa. Ericsson — Telekom-Anwendungen
- Nischenanwendungen
- Glasgow Haskell Compiler — Microsoft Research

Funktionale Programmierung in der Industrie

- Erlang
 - schwach typisiert, nebenläufig, strikt
 - Fa. Ericsson — Telekom-Anwendungen
- Nischenanwendungen
- Glasgow Haskell Compiler — Microsoft Research
- Warum nicht erfolgreicher?
 - Programmierung nur kleiner Teil
 - Unterstützung Libraries, Dokumentation etc. nicht ausgereift
 - Nicht verbreitet — Funktionale Programmierer zu teuer
 - Konservatives Management

Was lernt uns funktionale Programmierung?

- Abstraktion
 - Denken in Algorithmen, nicht in Programmiersprachen

Was lernt uns funktionale Programmierung?

- Abstraktion
 - Denken in Algorithmen, nicht in Programmiersprachen
- Konzentration auf **wesentliche** Elemente moderner Programmierung:
 - Typisierung und Spezifikation
 - Datenabstraktion
 - Modularisierung und Dekomposition

Was lernt uns funktionale Programmierung?

- Abstraktion
 - Denken in Algorithmen, nicht in Programmiersprachen
- Konzentration auf **wesentliche** 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?

Hilfe!

- Haskell: primäre Entwicklungssprache an der AG BKB
 - Formale Programmentwicklung: <http://www.tzi.de/cofi/hets>
 - Benutzerschnittstellen: <http://proofgeneral.inf.ed.ac.uk/kit>
- Wir suchen **studentische Hilfskräfte**
 - für diese Projekte
- Wir bieten:
 - Angenehmes Arbeitsumfeld
 - Interessante Tätigkeit
- Wir suchen **Tutoren für PI3**
 - im WS 05/06 — **meldet Euch!**

Tschüß!

