Praktische Informatik 3

Christoph Lüth

WS 02/03

Vorlesung vom 21.10.2002: Einführung

Personal

- Vorlesung: Christoph Lüth <cxl>
  MZH 8120, Tel. 7585
- Stud. Tutoren: Felix Beckwermert <foetus>
  Michael Skibbe <mskibbe>
  Dennis Walter <dw>
  Rafael Trautmann <pirate>
  Thomas Meyer <mclee>
- Website: www.informatik.uni-bremen.de/~cxl/pi3.
- Newsgroup: fb3.lv.pi3.

Termine

- Vorlesung:

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

- Tutorien:

Di	10-12	MZH 1380	Dennis Walter
Mi	15-17	MZH 6240	Rafael Trautmann
Mi	15-17	MZH 7250	Thomas Meyer
Mi	17-19	MZH 6240	Felix Beckwermert
Mi	17-19	MZH 1380	Michael Skibbe

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
  - Prolog und Logik; Nebenläufigkeit/GUIs; Grafik und Animation
- Lektüre:
  Simon Thompson: *Haskell — The Craft of Functional Programming* (Addison-Wesley, 1999)

Scheinrelevanz

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

Scheinrelevanz

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

Mit anderen Worten:

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

Scheinkriterien — Vorschlag:

- Ein Übungsblatt ist bearbeitet, wenn mindestens 20% der Punktzahl erreicht wurden.
- Alle Übungsblätter bearbeitet und mindestens 60% der Punktzahl erreicht.
- Individualität der Leistung wird sichergestellt durch:
  - Vorstellung einer Lösung im Tutorium
  - Beteiligung im Tutorium
  - Ggf. Prüfungsgespräch (auch auf Wunsch)

Einführung in FP

Warum funktionale Programmierung 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?

Referentielle Transparenz

- Programme als Funktionen

$$P : Eingabe \rightarrow Ausgabe$$

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

Referentielle Transparenz

Geschichtliches

- Grundlagen 1920/30
  - Kombinatorlogik und λ-Kalkül (Schönfinkel, Curry, Church)
- Erste Sprachen 1960
  - LISP (McCarthy), ISWIM (Landin)
- Weitere Sprachen 1970– 80
  - FP (Backus); ML (Milner, Gordon), später SML und CAML; Hope (Burstall); Miranda (Turner)
- 1990: Haskell als Standardsprache

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 ⇝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 ⇝2*(6+ 4) ⇝20
```

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

Auswertungsstrategie

- Von außen nach innen, links nach rechts.

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

$\rightsquigarrow$

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)
⇝(addDouble (inc 3) 4)+ 1
⇝2*(inc 3+ 4)+ 1
⇝
```

Auswertungsstrategie

- Von außen nach innen, links nach rechts.

```
inc (addDouble (inc 3) 4)
⇝(addDouble (inc 3) 4)+ 1
⇝2*(inc 3+ 4)+ 1
⇝2*(3+ 1+ 4)+ 1
⇝
```

Auswertungsstrategie

- Von außen nach innen, links nach rechts.

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

  ⇝`(addDouble (inc 3) 4)+ 1`

  ⇝`2*(inc 3+ 4)+ 1`

  ⇝`2*(3+ 1+ 4)+ 1`

  ⇝`2*8+1` ⇝`17`

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

Nichtnumerische Werte

- Rechnen mit Zeichenketten:

```
repeat s = s ++ s

repeat (repeat "hallo ")
```

$\rightsquigarrow$

Nichtnumerische Werte

- Rechnen mit Zeichenketten:

```
repeat s = s ++ s

repeat (repeat "hallo ")
⇝repeat "hallo"++ repeat "hello"
⇝
```

Nichtnumerische Werte

- Rechnen mit Zeichenketten:

```
repeat s = s ++ s
```

```
repeat (repeat "hallo ")
⇝repeat "hallo"++ repeat "hello"
⇝("hallo "++ "hallo ")++("hallo "++ "hallo ")
⇝
```

Nichtnumerische Werte

- Rechnen mit Zeichenketten:

```
repeat s = s ++ s
```

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

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion
- Beispiel:

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

- Auswertung kann divergieren!

Haskell in Aktion: hugs

- hugs ist ein Haskell-Interpreter
  - Klein, schnelle Übersetzung, gemächliche Ausführung.
- Funktionsweise:
  - hugs liest Definitionen (Programme, Typen, . . .) aus Datei (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-> b`

Vorlesung vom 28.10.2001: Funktionen und Typen

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

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

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

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

- Gilt auch verschachtelt.

Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen . . .

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

. . . bedingte Gleichungen:

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

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: Laufzeitfehler! Deshalb:

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

Kommentare

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

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

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

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

  - 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 LaTeX definieren:

```
\def\code{\verbatim}
\def\endcode{\endverbatim}
```

- Mit LaTeX setzen, mit Haskell ausführen. (Beispiel)

Funktionaler Entwurf und Entwicklung

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

  $\rightsquigarrow$ Signatur

Funktionaler Entwurf und Entwicklung

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

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

  $\rightsquigarrow$ Erster Entwurf

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

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

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

Ein Beispiel: das Nim-Spiel

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

Ein Beispiel: das Nim-Spiel

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

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

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
- Korrektheit:
  - Vermutung: Mit $4n + 1$ Hölzchen verloren, ansonsten gewonnen.
  - Beweis durch Induktion $\rightsquigarrow$ später.

Wahrheitswerte: `Bool`

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

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

- Beispiel: ausschließende Disjunktion:

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

- Alternative:

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

- Alternative:

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

- Igitt! Besser: Definition mit pattern matching

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

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

Ganze Zahlen: `Int` und `Integer`

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

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

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

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

Fließkommazahlen: `Double`

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

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

- Rundungsfehler!

Strukturierte Datentypen: Tupel und Listen

- Tupel sind das kartesische Produkt:

  `(t1, t2)` = alle möglichen Kombinationen von Werten aus `t1` und `t2`.

- Listen sind Sequenzen:

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

- Strukturierte Typen: konstruieren aus bestehenden Typen neue Typen.

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

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

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

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

- Beispiel: Punkte, Rechtecke, Polygone

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

Funktionen über Listen und Tupeln

- Funktionsdefinition durch pattern matching:

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

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

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

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

- Gesamtpreis des Einkaufs:

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

- Gesamtpreis des Einkaufs:

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

- Translation eines Polygons:

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

Einzelne Zeichen: `Char`

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

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

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

Zeichenketten: `String`

- `String` sind Sequenzen von Zeichenketten:

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

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

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

- Beispiel:

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

Beispiel: Palindrome

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

Beispiel: Palindrome

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

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

Beispiel: Palindrome

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

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

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

- Implementierung:

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

- Implementierung:

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

- Kritik:
  - Unterschied zwischen Groß- und kleinschreibung

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

  - Nichtbuchstaben sollten nicht berücksichtigt werden.

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 04.11.2002: Listenkomprehension, Polymorphie und Rekursion

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

- Ein Schema für Funktionen auf Listen:
  - Eingabe generiert Elemente,
  - die getestet und
  - zu einem Ergebns transformiert werden

Listenkomprehension

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

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

  - Alle Buchstaben aus `str` herausfiltern:

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

  - Beides zusammen:

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

- Generelle Form:

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

- Mit pattern matching:

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

- Auch mehrere Generatoren möglich:

```
[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 [ y | y <- xs, y<= x ]
               ++ [x] ++
               qsort [ 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)]
```

- 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.

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

Polymorphie — jetzt oder nie.

- Definition von `(++)`:

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

- Verketten von Strings:

Polymorphie — jetzt oder nie.

- Definition von `(++)`:

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

- Verketten von Strings:

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

- Gleiche Definition, aber unterschiedlicher Typ!

  $\Longrightarrow$ Zwei Instanzen einer allgemeineren Definition.

- Polymorphie erlaubt Parametrisierung über Typen:

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

  a ist hier eine Typvariable.

- Definition wird bei Anwendung instantiiert:

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

  aber nicht

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

- Typvariable: vergleichbar mit Funktionsparameter

- Weitere Beispiele:
  - Länge einer Liste:

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

  - Verschachtelte Listen "flachklopfen":

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

  - Kopf und Rest einer nicht-leeren Liste:

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

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

Undefiniert für leere Liste.

Übersicht: vordefinierte Funktionen auf Listen

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

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

Siehe Thompson S. 91/92.

Palindrom zum letzten:

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

Lokale Definitionen

- Lokale Definitionen mit `where` — Syntax:

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

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

Muster *(pattern)*

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

Primitive Rekursion auf Listen

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

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

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

- Auswertung:

```
sum [4,7,3]          ⇝  4 + 7 + 3 + 0
concat [A, B, C]     ⇝  A ++ B ++ C++ []
```

- Allgemeines Muster:

$$\texttt{f}[\texttt{x1}, \ldots, \texttt{xn}] = \texttt{x1} \otimes \texttt{x2} \otimes \ldots \otimes \texttt{xn} \otimes \texttt{A}$$

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

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

- 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, und n-te Königin so stellen, dass keine andere sie bedroht.
  - n-te Königin muß in n-ter Spalte plaziert werden.

- Hauptfunktion:

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

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

- Sichere neue Position:
  - Neue Position ist sicher, wenn sie durch keine anderen bedroht wird:

```
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.

slides-3.tex

Zusammenfassung

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

Vorlesung vom 12.11.2001: Formalisierung und Beweis Funktionen höherer Ordnung

Inhalt

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

Rekursive Definition, induktiver Beweis

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

```
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: $P(0)$
- Induktionssschritt: Annahme $P(x)$, zu zeigen $P(x+1)$.

Beweis durch strukturelle Induktion

Zu zeigen:

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

Beweis:

- Induktionssbasis: $P([])$
- Induktionssschritt: Annahme $P(xs)$, zu zeigen $P(x : xs)$

Ein einfaches Beispiel

Lemma: $\texttt{len}\,(xs \texttt{ ++ } ys) = \texttt{len}\,xs + \texttt{len}\,ys$

- Induktionsbasis: $xs = []$

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

- Induktionsschritt:

  Annahme: $\texttt{len}\,xs + \texttt{len}\,ys = \texttt{len}\,(xs \texttt{ ++ } ys)$, dann

$$\begin{aligned}\texttt{len}\,(x : xs) + \texttt{len}\,ys &= 1 + \texttt{len}\,xs + \texttt{len}\,ys \\ &= 1 + \texttt{len}\,(xs \texttt{ ++ } ys) \\ &= \texttt{len}\,(x : xs \texttt{ ++ } ys)\end{aligned}$$

Noch ein Beispiel

Lemma: $\texttt{rev}\,(xs \texttt{ ++ } ys) = \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs$

- Induktionsbasis:

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

- Induktionsschritt:

  Annahme ist $\texttt{rev}\,(xs \texttt{ ++ } ys) = \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs$, dann

$$\begin{aligned} \texttt{rev}\,(x : xs \texttt{ ++ } ys) &= \texttt{rev}\,(xs \texttt{ ++ } ys) \texttt{ ++ } [x] && \text{Def.} \\ &= (\texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs) \texttt{ ++ } [x] && \text{Ind.ann.} \\ &= \texttt{rev}\,ys \texttt{ ++ } (\texttt{rev}\,xs \texttt{ ++ } [x]) && \texttt{++ assoz.} \\ &= \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,(x : xs) \end{aligned}$$

Fixpunktinduktion

- Gegeben: rekursive Definition

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

- Zu zeigen: Für alle Listen x gilt $P(fx)$
- Beweis: Annahme:$P(ft)$, zu zeigen: $P(E)$.
  - d.h. ein Rekursionsschritt erhält P
  - Ein Fall für jede rekursive Gleichung.
  - Induktionsverankerung: nichtrekursive Gleichungen.

Berechnungsmuster

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

Funktionen Höherer Ordnung

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

Funktionen als Argumente

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

Funktionen als Argumente

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

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

- Definition

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

- oder -

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

Funktionen als Argumente

- Elemente filtern: `filter`
- Signatur:

Funktionen als Argumente

- Elemente filtern: `filter`
- Signatur:

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

- Definition

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

- oder -

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

Primitive Rekursion

- Primitive Rekursion:
  - Basisfall (leere Liste)
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- 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 [l1,l2,l3,l4] = l1++ l2++ l3++ l4++ []
```

Listen zerlegen

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

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

- Restliste des längsten Präfix

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

- Es gilt: `takeWhile p xs ++ dropWhile p xs == xs`

- Kombination der beiden

```
span      :: (a -> Bool) -> [a] -> ([a],[a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

- Ordnungserhaltendes Einfügen:

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

- 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 `ord` $y \wedge y$ `ord` $x \Rightarrow x = y$

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

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 18.11.2001: Funktionen Höherer Ordnung Typklassen

Inhalt

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

Funktionen als Werte

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

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

Funktionen als Werte

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

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

- n-mal hintereinander anwenden:

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

- Funktionskomposition:

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

  - `f` nach `g`.

- Funktionskomposition vorwärts:

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

  - Nicht vordefiniert!

- Identität:

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

  - Nützlicher als man denkt.

Anonyme Funktionen

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

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

- Besser: statt `less` anonyome Funktion

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

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

Beispiel: Primzahlen

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

Beispiel: Primzahlen

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

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

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

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

η-Kontraktion

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

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

- Da fehlt doch was?!

η-Kontraktion

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

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

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

Partielle Applikation

- Funktionen können partiell angewandt werden:

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

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

Die Kürzungsregel bei Funktionsapplikation

Bei Anwendung der Funktion

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

auf k Argumente mit $k \leq n$

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

werden die Typen der Argumente gekürzt:

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

$$\texttt{f e}_1\texttt{ ...e}_k\texttt{ :: t}_{k+1}\texttt{ -> ...-> t}_n\texttt{ -> t}$$

- Partielle Anwendung von Operatoren:

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

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

Gewürzte Tupel: Curry

- Unterschied zwischen
  `f :: a-> b-> c` und `f :: (a, b)-> c` ?
  - Links partielle Anwendung möglich.
  - Ansonsten äquivalent.
- Konversion:
  - Rechts nach links:

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

  - Links nach rechts:

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

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
  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

- Zerlegung des Problems in einzelne Schritte Ergebnistyp

  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

  (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

- Zerlegung des Problems in einzelne Schritte Ergebnistyp

  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

  (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

  (c) Zeilen in Worte spalten (Zeilennummer beibehalten):
  `[(Int, Word)]`

- Zerlegung des Problems in einzelne Schritte Ergebnistyp
  - (a) Text in Zeilen aufspalten: `[Line]`
    (mit `type Line= String`)
  - (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
  - (c) Zeilen in Worte spalten (Zeilennummer beibehalten): `[(Int, Word)]`
  - (d) Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`

- Zerlegung des Problems in einzelne Schritte Ergebnistyp

  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

  (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

  (c) Zeilen in Worte spalten (Zeilennummer beibehalten):
  `[(Int, Word)]`

  (d) Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`

  (e) Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
  `[([Int], Word)]`

- Zerlegung des Problems in einzelne Schritte Ergebnistyp

  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

  (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

  (c) Zeilen in Worte spalten (Zeilennummer beibehalten):
  `[(Int, Word)]`

  (d) Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`

  (e) Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
  `[([Int], Word)]`

  (f) Alle Worte mit weniger als vier Buchstaben entfernen:
  `[([Int], Word)]`

- Zerlegung des Problems in einzelne Schritte Ergebnistyp

  (a) Text in Zeilen aufspalten: `[Line]`
  (mit `type Line= String`)

  (b) Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`

  (c) Zeilen in Worte spalten (Zeilennummer beibehalten):
  `[(Int, Word)]`

  (d) Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`

  (e) Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
  `[([Int], Word)]`

  (f) Alle Worte mit weniger als vier Buchstaben entfernen:
  `[([Int], Word)]`

- Erste Implementierung:

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

- Implementierung der einzelnen Komponenten:
  - In Zeilen zerlegen:

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

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

  - Jede Zeile in Worte zerlegen:

    Pro Zeile: `words:: String-> [String]` aus dem Prelude.

    ▷ Berücksichtigt nur Leerzeichen.

    ▷ Vorher alle Satzzeichen durch Leerzeichen ersetzen.

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

Auf alle Zeilen anwenden, Ergebnisliste flachklopfen.

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

- Liste alphabetisch nach Worten sortieren:

  Ordnungsrelation definieren:

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

  Generische Sortierfunktion `qsortBy`

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

- Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:

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

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

Zweiter Schritt: Gleiche Worte zusammenfassen.

▷ Nach Sortierung sind gleiche Worte hintereinander!

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

- Alle Worte mit weniger als vier Buchstaben entfernen:

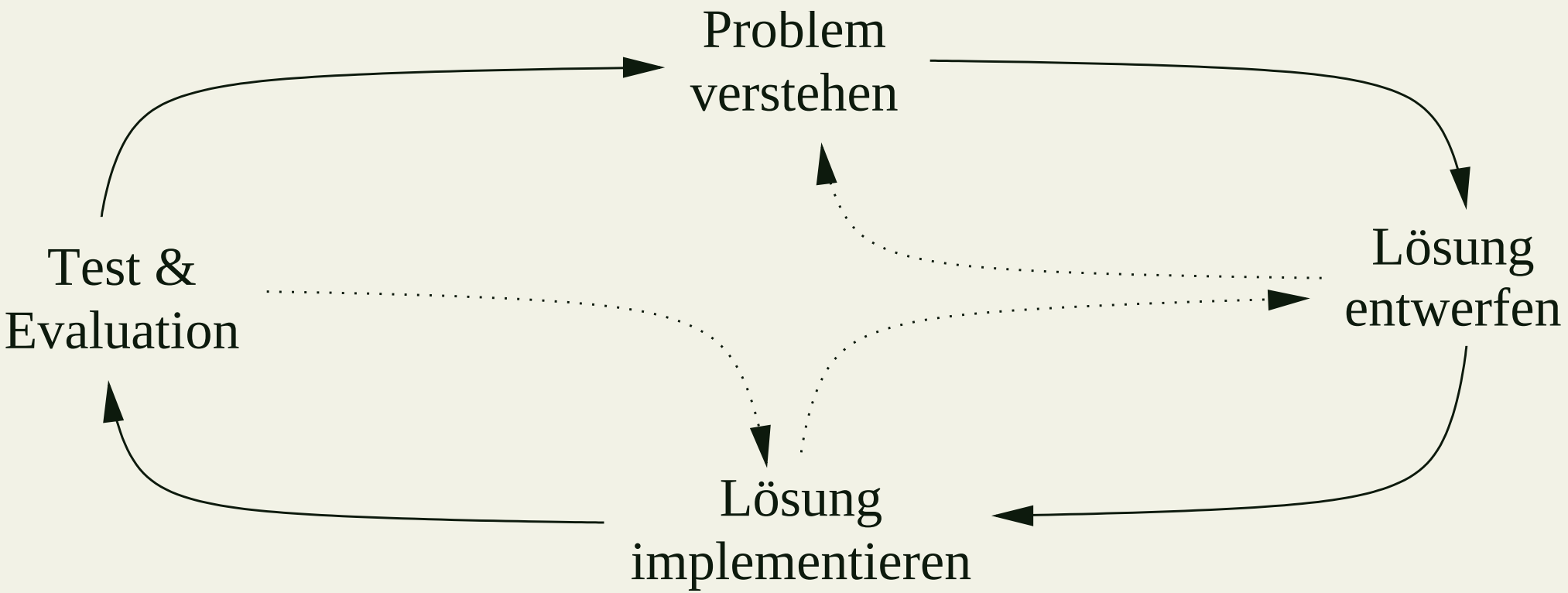
```
shorten :: [([Int],Word)] -> [([Int],Word)]
shorten = filter (\ (_, wd)-> length wd >= 4)
```

Alternative Definition:

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

- Testen.

Der Programmentwicklungszyklus



Typklassen

- Allgemeinerer Typ für `elem`:

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

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

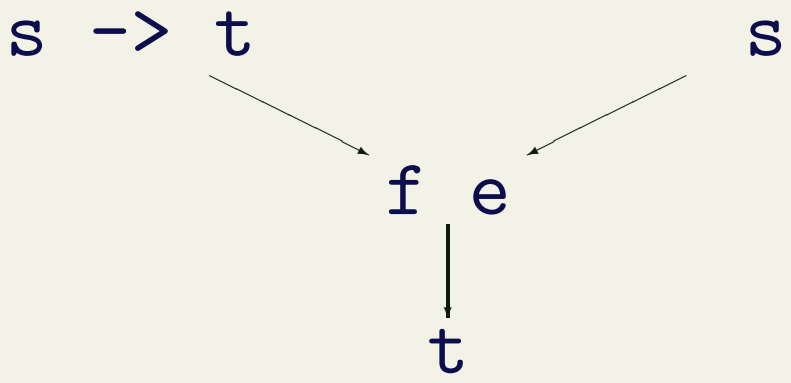
  ```
  elem :: Eq a=> a-> [a]-> Bool
  elem c = any (c ==)
  ```

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

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

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])
                  Char
```

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: Unifikation.
- Beispiel:

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

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: Unifikation.
- Beispiel:

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

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: Unifikation.
- Beispiel:

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

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: Unifikation.
- Beispiel:

```
f(x,y) = (x, ['a'     ..     y])
                Char
                          Char
                     [Char]
              a
                 (a, [Char])
f :: (a, Char)-> (a, [Char])
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs

                          [b]-> Int
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs

                    [b]-> Int
                              [b]
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs

                    [b]-> Int
                             [b]
                          Int
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs

                    [b]-> Int
                                [b]
                           Int
                Int
```

- Zweites Beispiel:

```
g(m, zs) =   m    +    length   zs

                          [b]-> Int
                                    [b]
                               Int
                    Int
                         Int

g :: (Int, [b])-> Int
```

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`
  - `g :: (Int, [b])-> Int`

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`
  - `g :: (Int, [b])-> Int`
  - `f :: (a, Char)-> (a, [Char])`

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`
  - `g :: (Int, [b])-> Int`
  - `f :: (a, Char)-> (a, [Char])`

- Hier Unifikation von `(a, [Char])` und `(Int, [b])` zu `(Int, [Char]`

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`
  - `g :: (Int, [b])-> Int`
  - `f :: (a, Char)-> (a, [Char])`

- Hier Unifikation von `(a, [Char])` und `(Int, [b])` zu

```
(Int, [Char]
```

- Damit

```
h :: (Int, [Char])-> Int
```

Typunifikation

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

Zusammenfassung

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

Vorlesung vom 25.11.2001: Algebraische Datentypen

Inhalt

- Letzte VL: Funktionsabtraktion 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.
- $\Longrightarrow$ Einfachster algebraischer Datentyp: Aufzählungstyp.

Aufzählungen

- Beispiel: Wochentage

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

- Konstruktoren:

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

  - Konstruktoren werden nicht deklariert.

- Funktionsdefinition durch pattern matching:

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

Produkte

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

- Konstruktor:

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

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

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

  - `day`, `year` sind Selektoren:

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

Alternativen

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

Alternativen

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

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

- Ein Konstruktor pro Variante.

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

Funktionen auf Geometrischen Figuren

- Funktionsdefinition durch pattern matching.
- Beispiel: Anzahl Eckpunkte.

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

  - Konstruktor-Argumente werden in ersten Fällen nicht gebraucht.

- Translation um einen Punkt (Vektor):

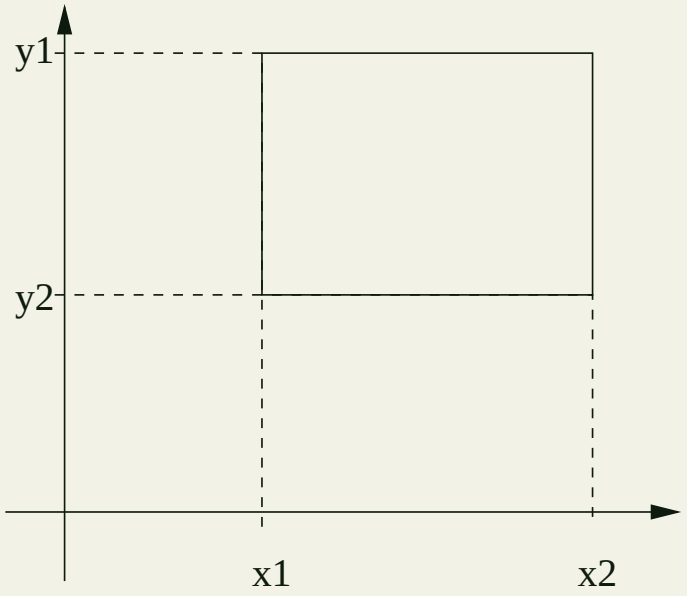
```
move :: Shape-> Point-> Shape
move (Circ m d) p    = Circ (add p m) d
move (Rect c1 c2) p = Rect (add p c1) (add p c2)
move (Poly ps) p     = Poly (map (add p) ps)
```

- Translation eines Punktes:
  - Durch Addition

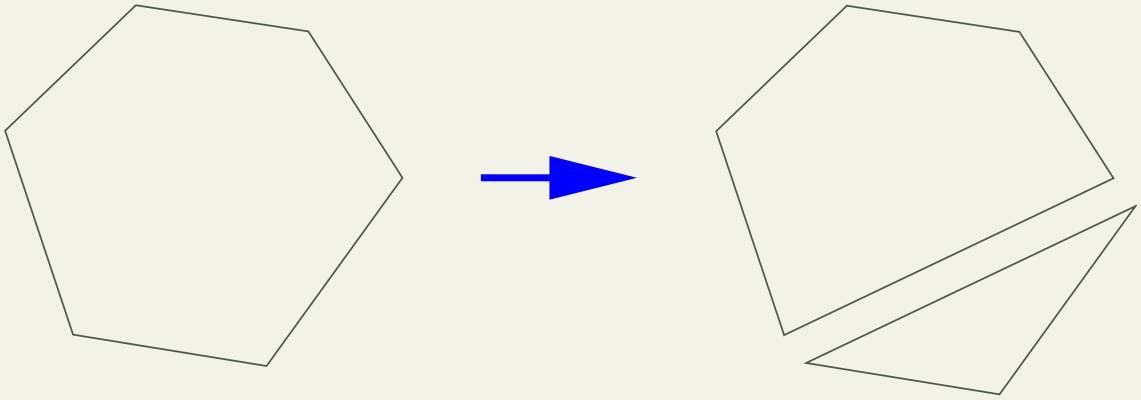
```
add :: Point-> Point-> Point
add (x, y) (u, v) = (x+ u, y+ v)
```

- Berechung der Fläche:
  - Einfach für Kreis und Rechteck.

```
area :: Shape-> Double
area (Circ _ d) = pi* (fromInt d)
area (Rect (x1, y1) (x2, y2)) =
       abs ((x2- x1)* (y2- y1))
```



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



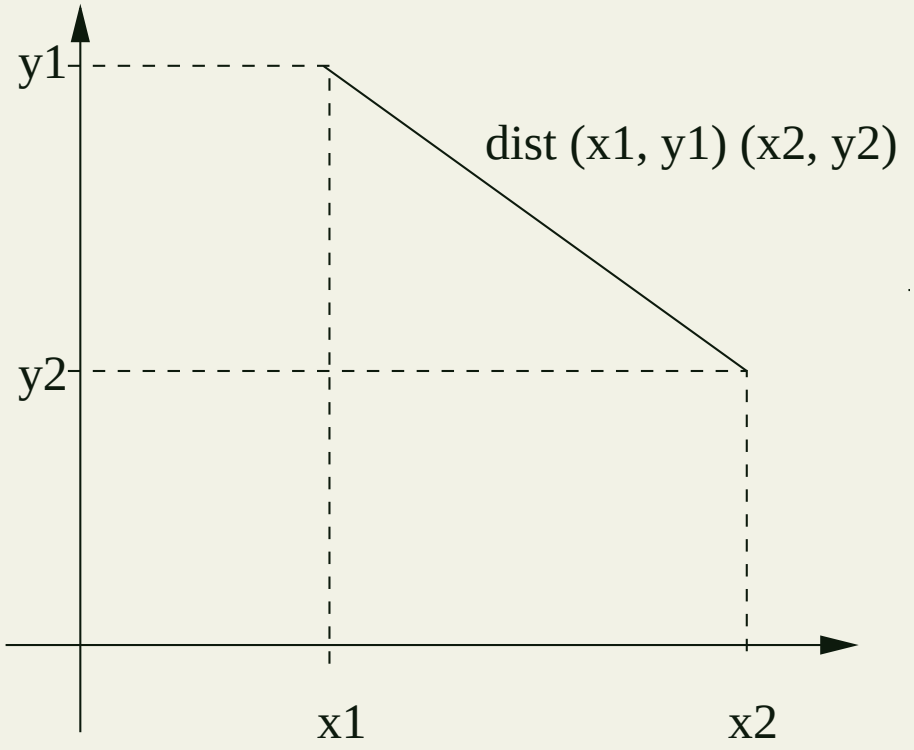
```
area (Poly ps) | length ps < 3  = 0
area (Poly (p1:p2:p3:ps)) =
           triArea p1 p2 p3 +
           area (Poly (p1:p3:ps))
```

- 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)
```

Das allgemeine Format

Definition von $\mathtt{T}$: $\mathtt{data\ T} = \mathtt{C}_1\ \mathtt{t}_{1,1} \ldots \mathtt{t}_{1,k_1}$

$\ldots$

$\mid\ \mathtt{C}_n\ \mathtt{t}_{n,1} \ldots \mathtt{t}_{n,k_n}$

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

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

- Konstruktoren sind injektiv:

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

- Konstruktoren erzeugen den Datentyp:

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

Rekursive Datentypen

Der definierte Typ `T` kann rechts benutzt werden.

- Beispiel: einfache arithmetische Ausdrücke sind
  - Zahlen (Literale), oder
  - Addition zweier Ausdrücke, oder
  - Multiplikation zweier Ausdrücke.

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
```

- Funktion darauf meist auch rekursiv.

- Ausdruck auswerten:

```
eval :: Expr-> Int
eval (Lit n)= n
eval (Add e1 e2) = eval e1+ eval e2
eval (Sub e1 e2) = eval e1- eval e2
```

- Ausdruck ausgeben:

```
print :: Expr-> String
print (Lit n) = show n
print (Add e1 e2) = "("++ print e1++ "+"++ print e
print (Sub e1 e2) = "("++ print e1++ "-"++ print e
```

- Testen.

- Primitive Rekursion auf Ausdrücken
  - Einen Wert für Rekursionsverankerung
  - Eine binäre Funktion für Addition
  - Eine binäre Funktion für Subtraktion

```
foldE :: (Int-> a)-> (a-> a-> a)-> (a-> a-> a)
              -> Expr-> a
foldE b a s (Lit n)     = b n
foldE b a s (Add e1 e2) = a (foldE b a s e1)
                            (foldE b a s e2)
foldE b a s (Sub e1 e2) = s (foldE b a s e1)
                            (foldE b a s e2)
```

- Damit Auswertung und Ausgabe:

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

Polymorphe Rekursive Datentypen

- Algebraische Datentypen parametrisiert über Typen:

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

- Paar: zwei beliebige Elemente gleichen Typs. Zeigen.

Polymorphe Rekursive 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)
```

Listen!

- 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 eingebauten Listen:

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

  - Geht so nicht!
  - Bei benutzerdefinierten Typen Operatoren als binäre Konstruktoren möglich.

- Funktionsdefinition:

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

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

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

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

Modellierung von Fehlern: `Maybe a`

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

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

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

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

Binäre Bäume

- Ein binärer Baum ist
  - Entweder leer,
  - oder ein Knoten mit genau zwei Unterbäumen.
  - Knoten tragen ein Label.

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

- Andere Möglichkeiten:
  - Label für Knoten und Blätter:

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

- 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)
```

- Primitive Rekursion auf Bäumen:
  - Rekursionsschritt:
    - ▷ Label des Knoten
    - ▷ Zwei Rückgabewerte für linken, rechten Unterbaum
  - Rekursionsanfang

```
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, `map`

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

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

- Testen.

- Traversion:

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

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

- Test auf Enthaltensein vereinfacht:

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

- Ordnungserhaltendes Einfügen

```
insert :: Ord a=> Tree a-> a-> Tree a
insert Null a = 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)
```

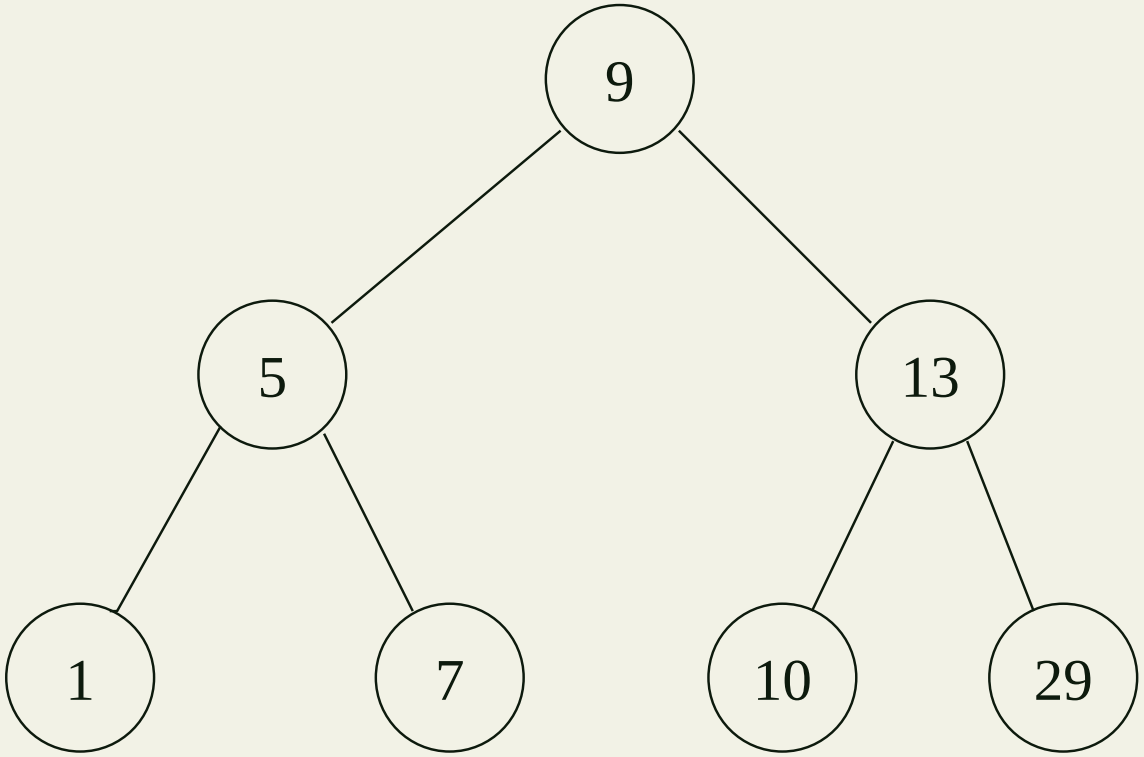
- Problem: Erzeugung ungeordneter Bäume möglich.
  - Lösung erfordert Verstecken der Konstrukturen — nächste VL.

- Löschen:

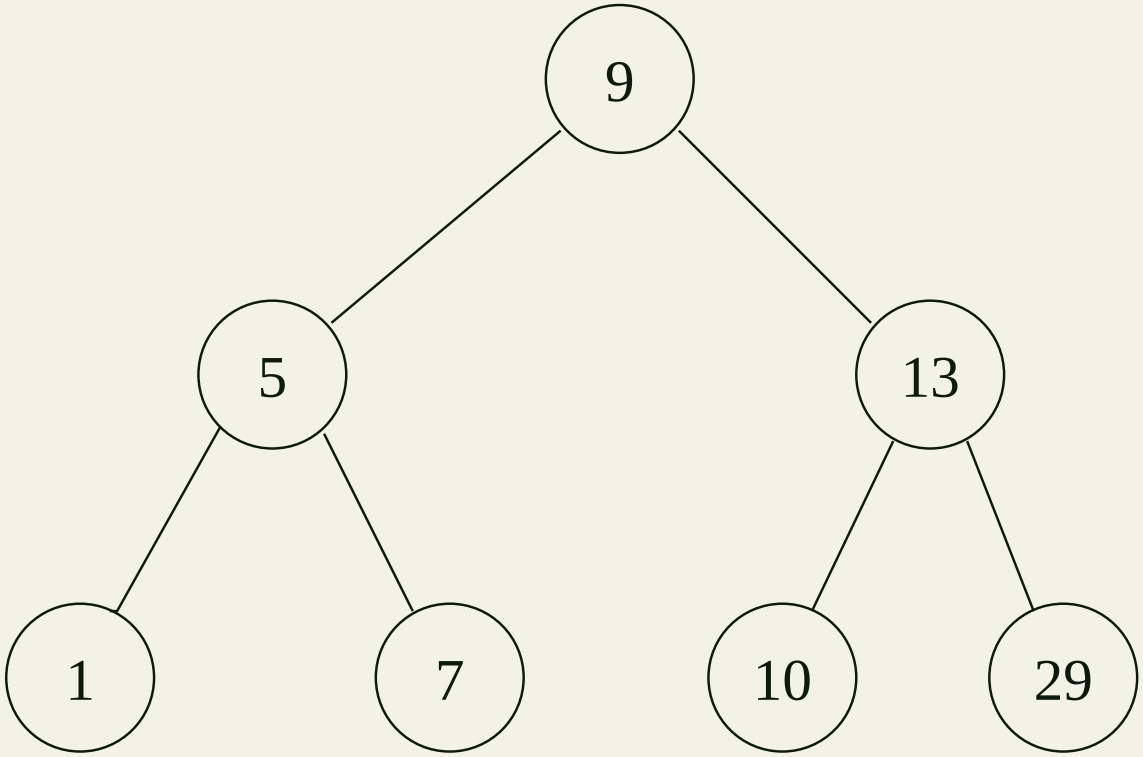
```
delete :: Ord a=> a-> Tree a-> Tree a
delete x Null = Null
delete x (Node l y r)
   | x < y  = Node (delete x l) y r
   | x == y = join l r
   | x > y  = Node l y (delete x r)
```

- `join` fügt zwei Bäume ordnungserhaltend zusammen.

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

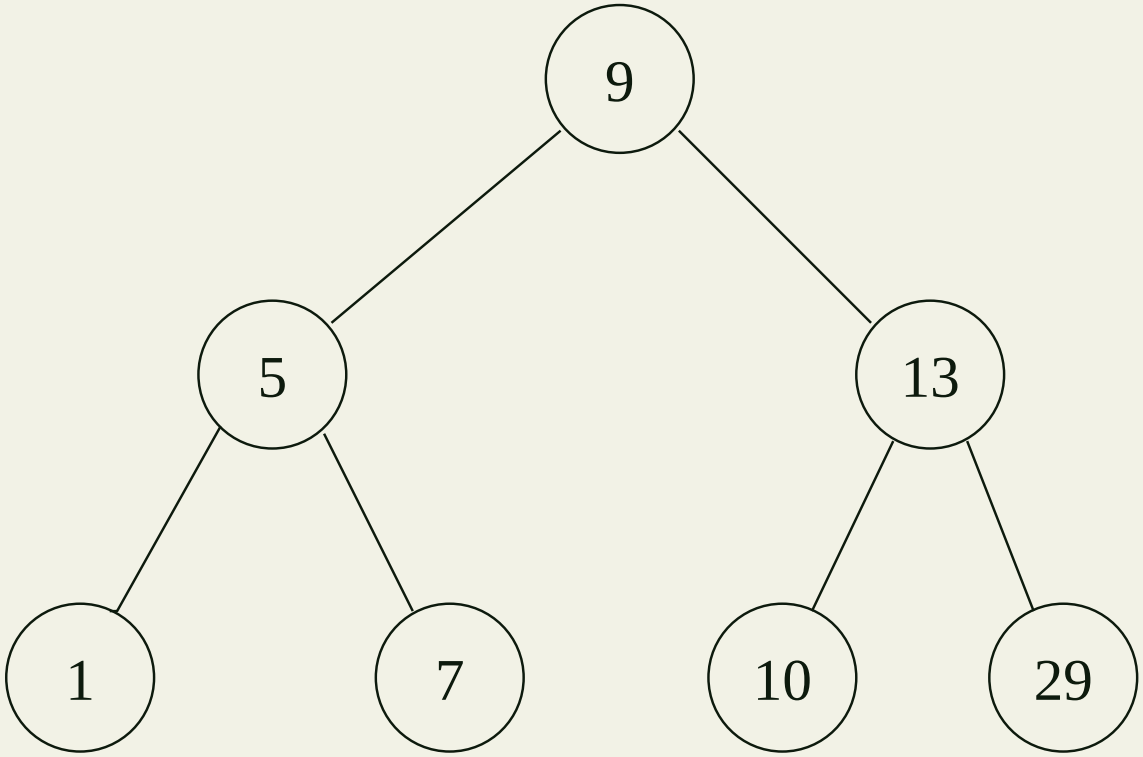


- Beispiel: Gegeben folgender Baum, dann `delete t 9`



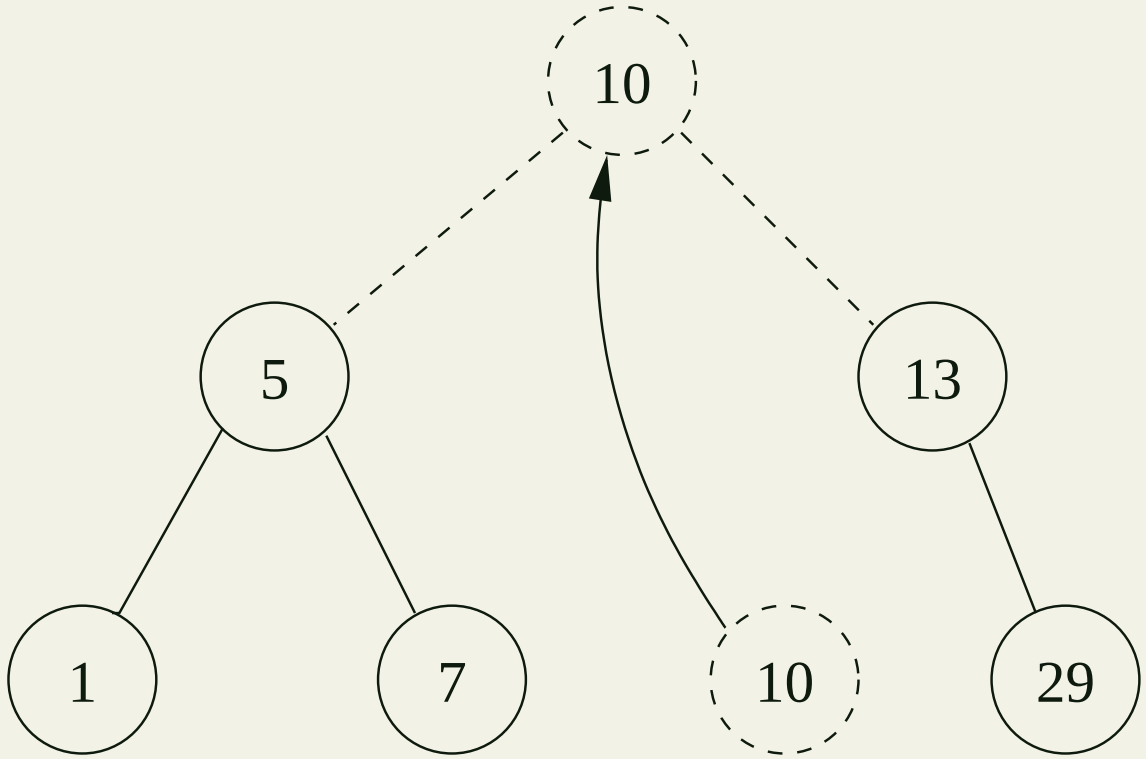
  - Wurzel wird gelöscht

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



  - Wurzel wird gelöscht
  - `join` muß Bäume ordnungserhaltend zusammenfügen

- `join` fügt zwei Bäume ordnungserhaltend zusammen.



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

- 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 02.12.2002: Abstrakte Datentypen

Inhalt

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

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 :: Store a b-> a-> Maybe b`
  - Möglicherweise undefiniert.
- Wert schreiben:
  `update :: 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 = St [(a, b)]
```

- Leerer Speicher: leere Liste

```
initial  = St []
```

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

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

- Update: neues Paar vorne anhängen slides-7-1.tex

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

Zweite Implementation

- Speicher als Funktion

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

- Leerer Speicher: konstant undefiniert

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

- Lookup: Funktion anwenden

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

- Update: punktweise Funktionsdefinition

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

Ein Interface, zwei mögliche Implementierungen.

Export von Datentypen

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

Benutzung eines ADTs — Import.

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

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

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

- `hiding`: Liste der Bezeichner wird versteckt

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

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

Schnittstelle vs. Semantik

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

- Beispiel:

Operation	Queue	Repräsentation

- Beispiel:

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

- Beispiel:

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

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])
`enq 7`	$7 \longrightarrow 4$	([4], [7])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])
`enq 7`	$7 \longrightarrow 4$	([4], [7])
`enq 5`	$5 \longrightarrow 7 \longrightarrow 4$	([4], [5, 7])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])
`enq 7`	$7 \longrightarrow 4$	([4], [7])
`enq 5`	$5 \longrightarrow 7 \longrightarrow 4$	([4], [5, 7])
`deq`	$5 \longrightarrow 7$	([7, 5], [])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])
`enq 7`	$7 \longrightarrow 4$	([4], [7])
`enq 5`	$5 \longrightarrow 7 \longrightarrow 4$	([4], [5, 7])
`deq`	$5 \longrightarrow 7$	([7, 5], [])
`deq`	7	([5], [])

- Beispiel:

Operation	Queue	Repräsentation
`empty`		([], [])
`enq 9`	9	([9], [])
`enq 4`	$4 \longrightarrow 9$	([9], [4])
`deq`	4	([4], [])
`enq 7`	$7 \longrightarrow 4$	([4], [7])
`enq 5`	$5 \longrightarrow 7 \longrightarrow 4$	([4], [5, 7])
`deq`	$5 \longrightarrow 7$	([7, 5], [])
`deq`	7	([5], [])
`deq`		([], [])

- ```
  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 =

```
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
```

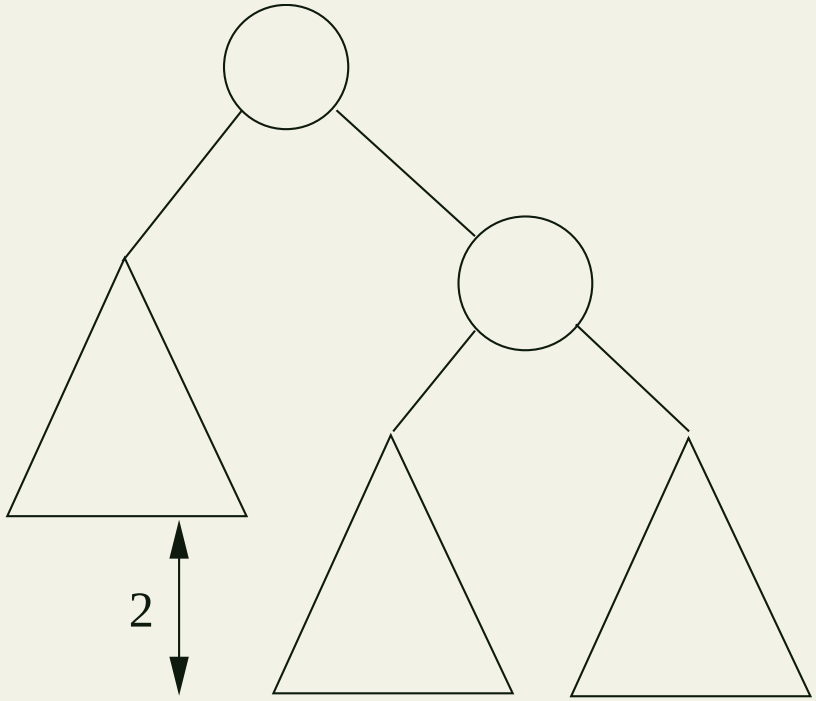
- Neuen Knoten anlegen, Höhe aus Unterbäumen berechnen.

```
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)
```

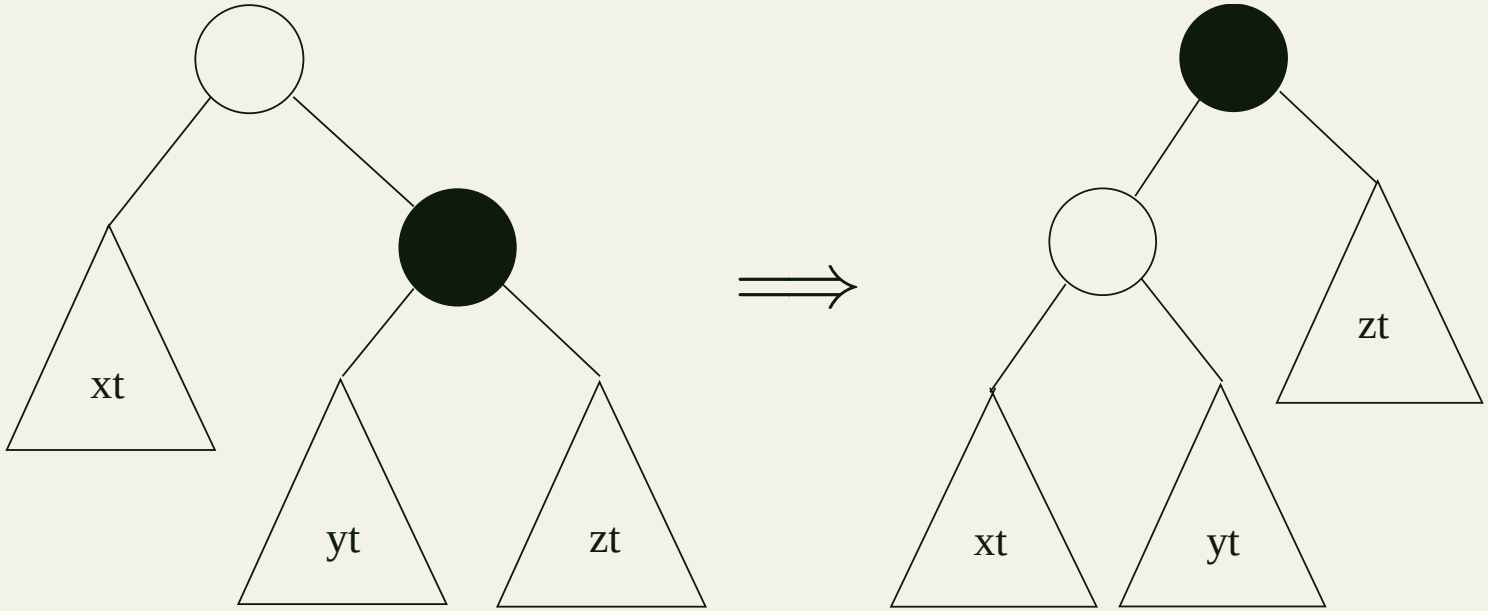
Ausgeglichenheit sicherstellen

- Problem:
  Nach Löschen oder
  Einfügen zu großer
  Höhenunterschied

- Lösung:
  Rotieren der Unterbäume

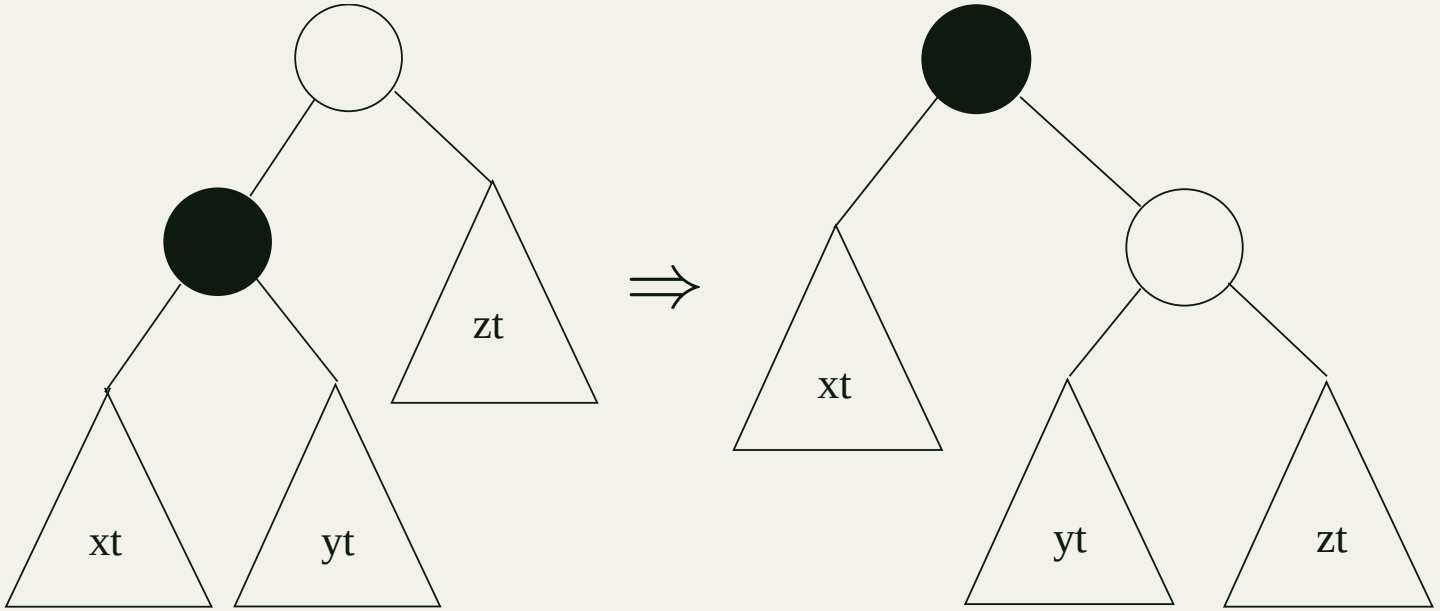


Linksrotation




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

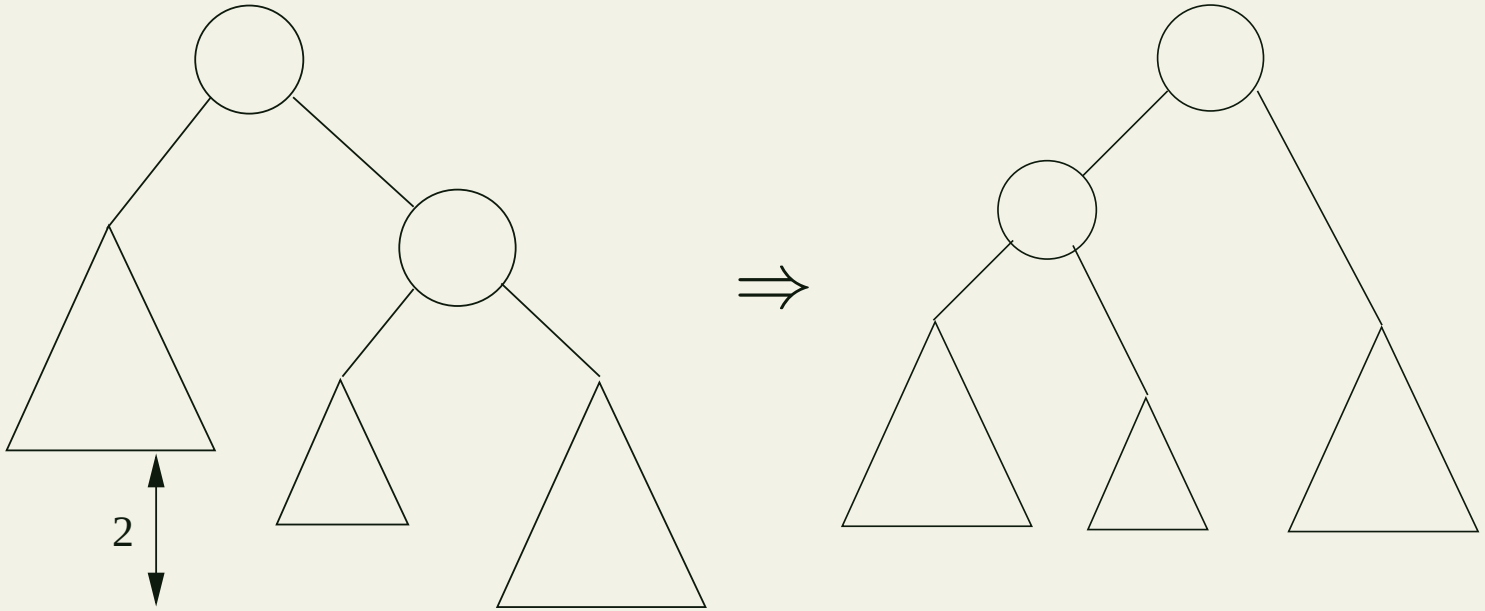
Rechtsrotation




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

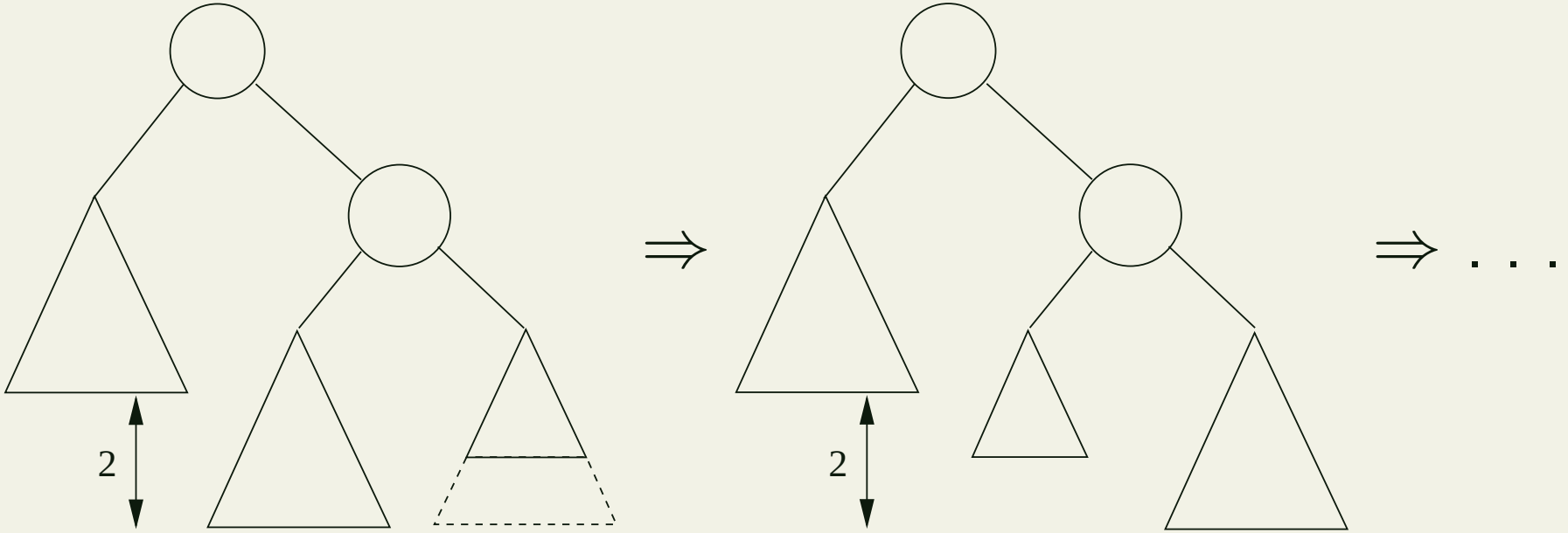
Ausgeglichenheit sicherstellen

- 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
```

- Unterscheidung der Fälle
  - Fall 1: `bias` < 0
  - Fall 2: `bias` ≥ 0
- Symmetrischer Fall (linker Teilbaum zu hoch)
  - Fall 1: `bias` > 0
  - Fall 2: `bias` ≤ 0

- `mkAVL xt y zt`:
  - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei;
  - Konstruiert neuen AVL-Baum mit Knoten `y`.

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

```
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 `fold`)

  ```
  enum :: Ord a=> AVLTree a-> [a]
  enum = foldT (\x t1 t2-> t1++ [x]++ t2) []
  ```

- Enthaltensein: Testen.

  ```
  elem :: Ord a=> a-> AVLTree a-> Bool
  elem e = foldT (\x b1 b2-> e == x || b1 || b2) Fal
  ```

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 09.12.2001: 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 rechts nach links
- Effekt: Parameterübergabe *call by need*, nicht-strikt
  - Beispiel:

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

    `silly (double 3) (double 4)` $\rightsquigarrow$ `double 4` $\rightsquigarrow$ `4+ 4` $\rightsquigarrow 4$
  - Erstes Argument von `silly` wird nicht ausgewertet.
  - Zweites Argument von `silly` wird erst im Funktionsrumpf ausgewertet.

Striktheit

- Funktion `f` ist strikt in einem Argument `x`, wenn ein undefinierter Wert für `x` die Funktion undefiniert werden läßt.
- Beispiele:
  - `(+)` strikt in beiden Argumenten.
  - `(&&)` strikt im ersten, nicht-strikt im zweiten.
    `False && (1/0 == 1/0)` $\rightsquigarrow$ `False`
  - `silly` nicht-strikt im ersten. `silly (1/0) 3` $\rightsquigarrow$ `3`

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

    $\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.
  - Minimum einer Liste durch

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

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

    $\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.
  - Minimum einer Liste durch

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

    $\Longrightarrow$ Liste wird nicht vollständig sortiert — binäre Suche.

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

    $\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.
  - Minimum einer Liste durch

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

    $\Longrightarrow$ Liste wird nicht vollständig sortiert — binäre Suche.

- Unendliche Datenstrukturen.

Unendliche Datenstrukturen: Ströme

- Ströme sind unendliche Listen.
  - Unendliche Liste `[2,2,2,...]`

    ```
    twos = 2 : twos
    ```
  - Die natürlichen Zahlen:

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

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

    ```
    cycle (trace "Foo!" "x")
    ```
- Nützlich für Listen mit unbekannter Länge

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — 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:

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

- Damit ergibt sich:

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

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

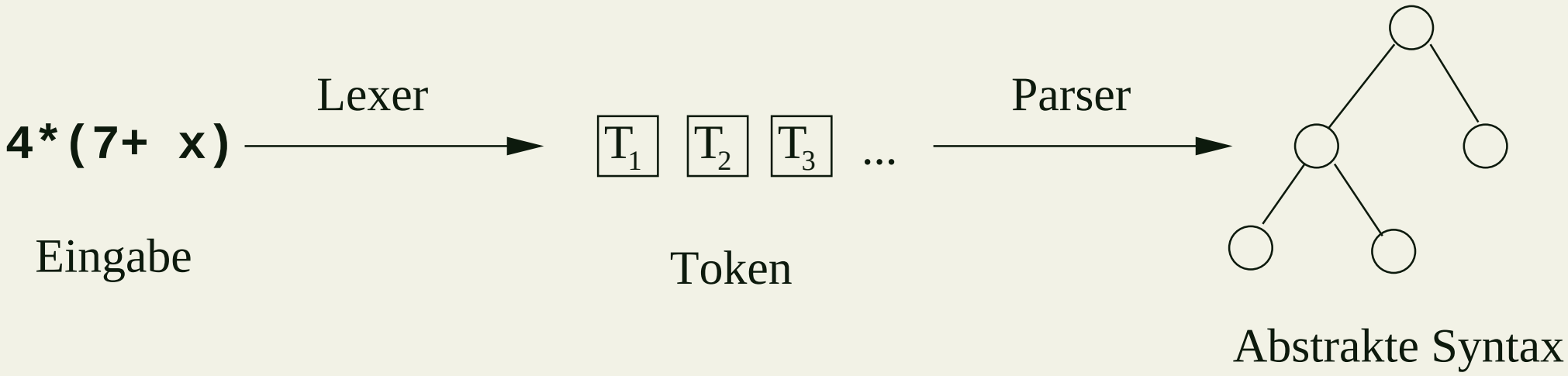
- Damit ergibt sich:

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

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

Fallstudie: Parsierung

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



- Parser bildet Eingabe auf Parsierungen ab.
- Basisparser erkennen Terminalsymbole
- Kombinatoren:
  - Sequenzierung (erst A, dann B)
  - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

  $\Longrightarrow$ Nichtterminalsymbole

Arithmetische Ausdrücke

- Grammatik:

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

- Daraus abstrakte Syntax:

```
data Expr   = Plus  Expr Expr
            | Minus Expr Expr
            | Times Expr Expr
            | Div   Expr Expr
            | Number Int
            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])]
```

- Beispiel:

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

Basisparser

- Erkennt nichts:

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

- Erkennt alles:

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

- Erkennt einzelne Zeichen:

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

  - Warum nicht `none`, `suceed` durch `spot`?

Kombinatoren

- 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]
```

- Eingabe weiterverarbeiten:

```
infix 4 `build`
build :: Parse a b-> (b-> c)-> Parse a c
build 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 `build` snd
p1 >* p2 = p1 >*> p2 `build` fst
```

Abgeleitete Kombinatoren

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

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

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

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

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

Einschub: Präzedenzen

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`
```

Der Kern des Parsers

- Parsierung von `Expr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm
                              `build` uncurry Plus
        `alt` pTerm >* token '-' >*> pTerm
                              `build` uncurry Minus
        `alt` pTerm
```

- Parsierung von `Term`

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pFactor
                                 `build` uncurry Times
        `alt` pFactor >* token '/' >*> pFactor
                                   `build` uncurry Div
        `alt` pFactor
```

- Parsierung von `Factor`

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

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen: Testen.
  - Parsierung braucht Eingabe auf
  - Keine Mehrdeutigkeit

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

Ein kleiner Fehler

- Mangel: 3+4+5 ist Syntaxfehler
- Behebung: leichte Änderung der Grammatik . . .

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

  - (vergleiche alt)

- Abstrakte Syntax bleibt . . .

- Entsprechende Änderung des Parsers in `pExpr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr
                              `build` uncurry Plus
        `alt` pTerm >* token '-' >*> pExpr
                              `build` uncurry Minus
        `alt` pTerm
```

  - (vergleiche alt)

- . . . und in `pTerm`:

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pTerm
                              `build` uncurry Times
        `alt` pFactor >* token '/' >*> pTerm
                                `build` uncurry Div
        `alt` pFactor
```

- `pFactor` und Hauptfunktion bleiben: Testen.

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Abstraktion durch Funktionen höherer Ordnung.
- Durch verzögerte Auswertung annehmbare Effizienz:
  - Einfache Implementierung (wie oben) skaliert nicht
  - Grammatik muß eindeutig sein (LL(1) o.ä.)
  - Gut implementierte Büchereien (wie Parsec) bei eindeutiger Grammatik 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 16.12.2001: 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

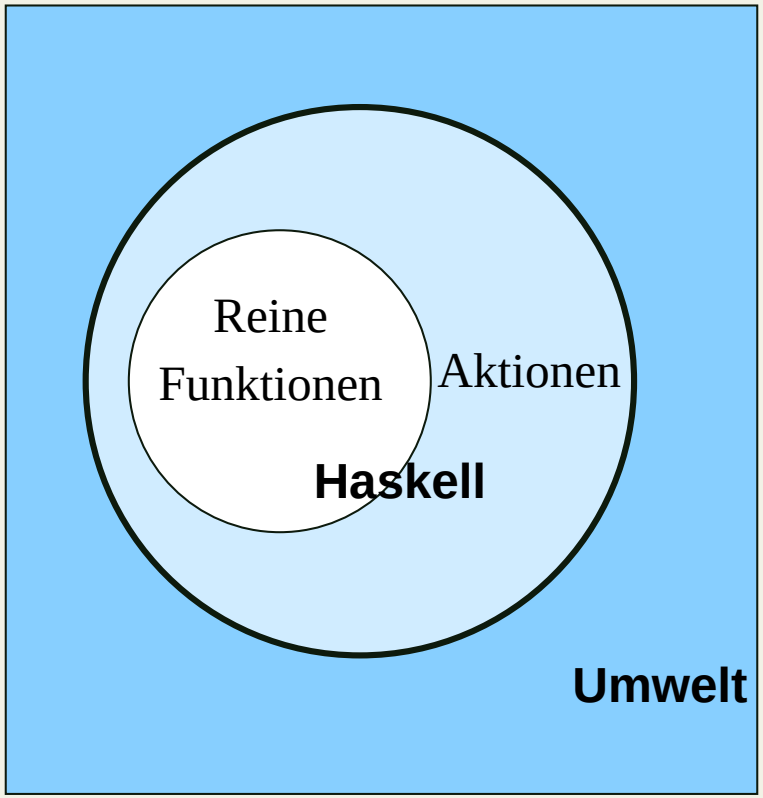
- **Problem:**
  Funktionen mit Seiteneffekten nicht referentiell transparent.
  - z. B. `readString :: () -> String` ??

Ein- und Ausgabe in funktionalen Sprachen

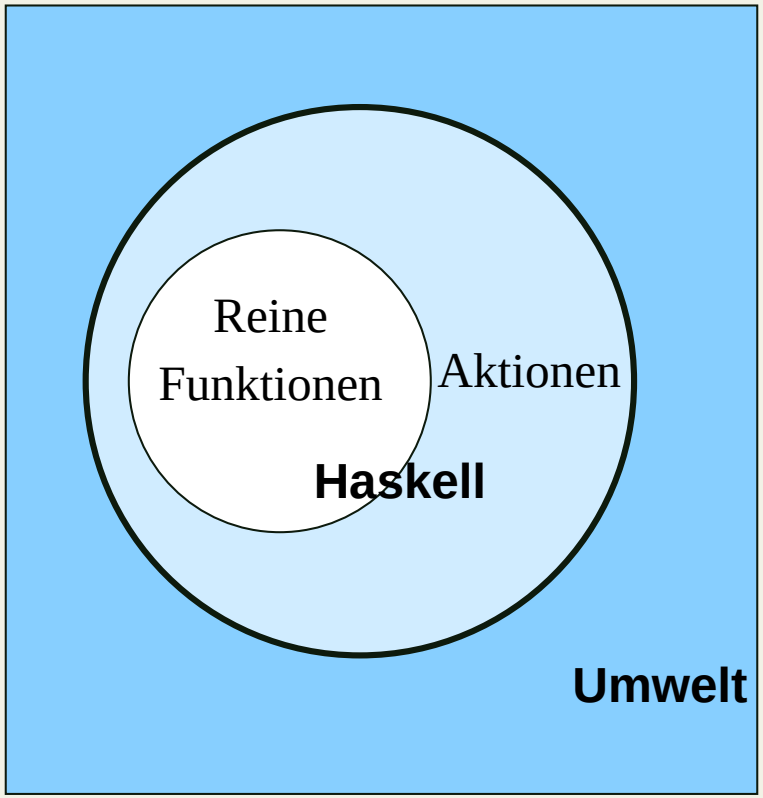
- **Problem:**
  Funktionen mit Seiteneffekten nicht referentiell transparent.
  - z. B. `readString :: () -> String` ??

- **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
```

- String auf `stdout` ausgeben:

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

- String mit Zeilenvorschub ausgeben:

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

Ein einfaches Beispiel

- Echo:

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

- Umgekehrtes Echo:

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

- Vordefinierte Abkürzung:

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

Die do-Notation

- Syntaktischer Zucker für `IO`:

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

  - Rechts sind `>>=`, `>>` implizit.
- Es gilt die Abseitsregel.
  - Einrückung der ersten Anweisung nach `do` bestimmt Abseits.
  - Fallunterscheidungen: `then`-Zweig einrücken.

Module in der Standardbücherei

- 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 Standardbücherei
  - 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))
                              ++ " lines ")
     putStrLn (show (length (words c))
                              ++ " words, and ")
     putStrLn (show (length c)++ " characters. ")
```

Beispiel: Zeichen, Worte, Zeilen zählen (wc)

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

Nicht sehr effizient — Datei wird im Speicher gehalten.

Noch ein 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
```

- 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
```

```
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 l  -> do putStrLn ("Ich nehme "
                                      ++ show l)
                            play (n-(m+l))
```

- Noch zu implementieren: Benutzereingabe

```
getInput' :: IO Int
getInput' =
  do putStr "Wieviele nehmen Sie? "
     n <- do s<- getLine
             return (read s)
     if n<= 0 || 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
```

- Nur in Aktionen können Fehler behandelt werden.

- 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.

- Endlosschleife:

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

- 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

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

  - `do`-Notation
- Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: `getLine, putStr, putStrLn`
  - Module: `IO`, `Random`,

Vorlesung vom 06.01.2003: Grafikprogrammerierung I Die Hugs Graphics Library

Inhalt

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

Grafik — erste Schritte.

- Das kanonische Beispielprogramm:

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

- `runGraphics :: IO ()-> IO ()`
  führt Aktion mit Grafik aus;

- `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 X11 und Win32 Graphics Library.
- Bietet:
  - Fenster
  - verschiedene Zeichenfunktionen
  - Unterstützung für Animationen
- Bietet nicht:
  - Hochleistungsgrafik, 3D-Unterstützung (e.g. OpenGL)
  - GUI-Funktionalität

Übersicht HGL

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

Basisdatentypen

- Winkel (Grad, nicht Bogenmaß)

```
type Angle     = Double
```

- Dimensionen (Pixel)

```
type Dimension = Int
```

- Punkte (Ursprung: links oben)

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

Atomare Grafiken

- Größte Ellipse (gefüllt) innerhalb des gegeben Rechtecks

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

- Größte Ellipse (gefüllt) innerhalb des Parallelograms:

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

- Bogenabschnitt einer Ellipse im math. positiven Drehsinn:

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

- Beispiel:

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

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

- 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`

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

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

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

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

- Fonts
  - Punktgröße, Winkel (nicht unter X11), Fett, Kursiv, Name.
  - Portabilität beachten — keine exotischen Kombinationen.
  - Portable Namen: `courier`, `helvetica`, `times`. Beispiel, Quelle.

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

- Farben
  - Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
```

  - Dazu Modifikator:

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

- Kombination von Grafiken
  - Überlagerung (erste über zweiter):

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

Fenster

- Elementare Funktionen:

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

- Abgeleitetete Funktionen:
  - In Fenster zeichnen:

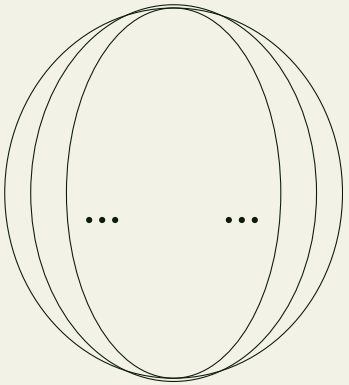
```
drawInWindow :: Window -> Graphic -> IO ()
drawInWindow w g = do
  old <- getGraphic w
  setGraphic w (g `overGraphic` old)
```

  - Grafik löschen

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

Ein einfaches Beispiel

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



- Bälle zeichnen:

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

- Farbveränderung, zyklisch:

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

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

- Repräsentation durch algebraischen Datentyp

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

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

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

Darstellung einer geometrischen Figur

- Rechtecke und Dreiecke sind spezielle Polygone.
- Alles ist ein Polygon oder eine Ellipse.
- Nullpunkt $(0, 0)$ des Koordinatensystems links oben.
- 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 `draw`:
  - (1) – (3) erlauben Reduktion zu einer Normalform

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

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

- 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 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 Zeichenfunktion

- Damit Zeichenfunktion:

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

- Hauptarbeit findet in `draw'` statt:
  - Translation, Skalierung, Rotation in Argumenten kumulieren
  - Basisfiguren entsprechend zeichnen

- Translation, Skalierung, Rotation:
  - Translation, Skalierung, Rotation aufrechnen
  - Dabei Gleichungen (1), (2) beachten

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

- Basisfiguren zeichnen:

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

- Hilfsfunktion: Polygon zeichnen

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

Ein kleines Beispielprogramm

- Sequenz von gedrehten, skalierten Figuren.
- 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. Translate (250, 250)) f)))
  getKey w
  closeWindow w
```

  - `(cycle [Blue ..]` erzeugt unendliche Liste aller Farben.
  - NB: Figuren werden im Fenstermittelpunkt gezeichnet.

- Figuren erzeugen

```
swirl :: Figure-> [Figure]
swirl = iterate (Scale 1.123. Rotate (pi/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 20 (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: Kollisionserkennung und Animation.

Vorlesung vom 13.01.2003: Grafikprogrammierung II Animation und Interaktion

Inhalt

- Erweiterung der Geometriebücherei
- Bewegte Grafiken: Animationen
- *Labelled Records*
- Space — the final frontier

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`

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

Erweitung der Schnittstelle

- Abstrakter Typ `Shape`
- Konversion von Figur in Form:

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

- Form zeichnen:

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

- Kollisionserkennung:

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

- Gesamte Schnittstelle

Implementierung

- Ein `Shape` ist
  - ein geschlossenes Polygon oder ein Kreis
  - mit normalisierten Koordinaten.

```
data Shape = Poly [Point]
           | Circ Point Double
           deriving (Eq, Show)
```

- Konversion `Figure` nach `Shape`:
  - Translation, Rotation, Skalierung herausrechnen;
  - Adaption von draw (Letzte VL)

```
shape :: Figure-> Shape
shape = fig' ((0, 0), 1, 0) where
  fig' :: (Point, Double, Angle)-> Figure-> Shape
  fig' (m, r, phi) (Translate t f) =
    fig' (add m (smult r (rot phi t)), r, phi) f
  fig' (m, r, phi) (Scale s f) =
    fig' (m, r* s, phi) f
  fig' (m, r, phi) (Rotate w f) =
    fig' (m, r, phi+ w) f
  fig' c (Rect a b) =
    poly c [(x2, y2), (-x2, y2),
            (-x2, -y2), (x2, -y2)] where
      x2= a `div` 2; y2= b `div` 2
  fig' c (Triangle l1 a l2) =
    poly c [(0, 0), (0, l1), rot a (0, l2)]
```

```
fig' c (Polygon pts) = poly c pts
fig' (m, r, _) (Circle d) =
  Circ m (r*fromInt d)
poly :: (Point, Double, Angle)-> [Point]-> Shape
poly (m, p, w) = Poly. checkclosed.
                       map (add m. smult p. rot w) where
```

- Prüfung ob Polygon geschlossen

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

Formen zeichnen

- Form zeichnen ist trivial:

```
drawShape :: Shape-> Graphic
drawShape (Poly pts) = polygon pts
drawShape (Circ (mx, my) r) =
    ellipse (mx-r', my- r') (mx+ r', my+ r') where
      r'= round r
```

- Alte Funktion `draw :: Figure-> Graphic`

```
draw :: Figure-> Graphic
draw = drawShape . shape
```

Kollisionserkennung

- Fallunterscheidung:
  - Polygon und Polygon
    - ▷ Kollision, wenn ein Eckpunkt des einen im anderen
    - ▷ Voraussetzung: Polygone konvex
  - Polygon und Kreis
    - ▷ Kollision, wenn ein Eckpunkt im Kreis
  - Kollision Kreis und Kreis
    - ▷ Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Erster Schritt: Kollision von Formen und Punkten.

Kollisionserkennung: Punkte und Kreise

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

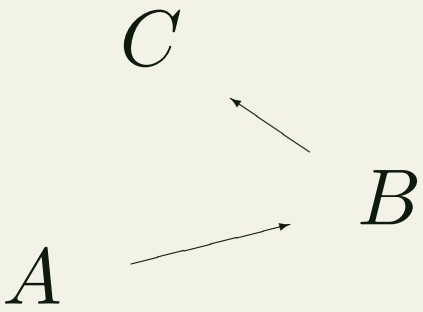
```
inC :: Point-> Double-> Point-> Bool
inC (mx, my) r (px, py) =
    len (px- mx, py- my) <= r
```

- Abstand ist Länge (Betrag) des Differenzvektors:

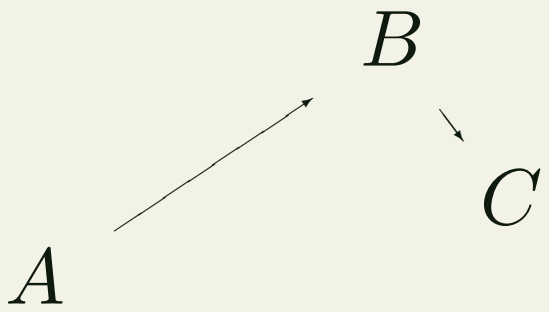
```
len :: Point-> Double
len (x, y)= sqrt (fromInt (x^2+ y^2))
```

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)) =
  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.

Kolliosionserkennung für Punkte

- Einfache Fallunterscheidung

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

Kollisionserkennung von Formen

- Kreise: Distanz Mittelpunkte kleiner als Summe Radien

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

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

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

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

```
intersect (Poly p) (Circ c r)=
   inP p c || any (inC c r) p
intersect (Circ c r) (Poly p)=
   inP p c || any (inC c r) p
```

Noch eine Hilfsfunktion

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

  Punkt $(r, 0)$ um Winkel ϕ drehen.

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

Benutzereingaben: `Events`

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

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

  - Letzte Eingabe, nicht warten:

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

- `Event` ist ein labelled record:

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

- `Event` ist ein labelled record:

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

- Was ist das ?!?

Probleme mit großen Datentypen

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

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

  - Kommt Stückzahl oder Preis zuerst?

Probleme mit großen Datentypen

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

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

  - Kommt Stückzahl oder Preis zuerst?

- Beispiel Buch:
  - Titel, Autor, Verlag, Signatur, Fachgebiet, Stichworte

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

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

Probleme mit großen Datentypen

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

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

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

Lösung: *labelled records*

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

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

- Konstruktion:

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

- Selektion durch Feldnamen:

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

- Update:

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

  - Rein funktional! (`b` bleibt unverändert)

- Patternmatching:

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

- Partielle Konstruktion und Patternmatching möglich:

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

  - Guter Stil: nur auf benötigte Felder matchen.

- Datentyp erweiterbar:

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

  Programm muß nicht geändert werden (nur neu übersetzt).

Zusammenfassung labelled records

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

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

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

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

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

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

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

- Hauptprogramm:
  - Fenster öffnen

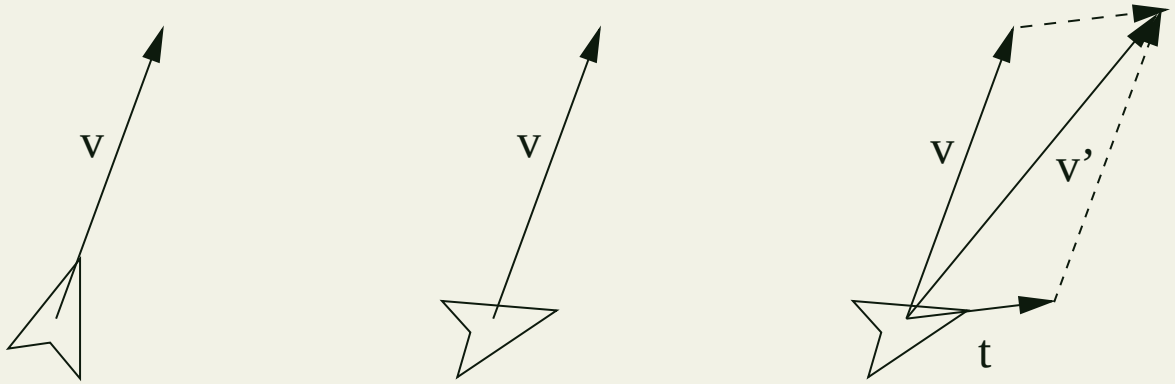
```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Bounce!"
                      Nothing (500, 500) DoubleBuffered
                      (Just 30)
     loop w (Ball{p=(0, 10), v= (5, 0)}) where
```

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

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

Ein Raumflugsimulator

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



- Geschwindigkeit $\vec{v}$, Schub $\vec{t}$
  - Schub operiert immer in Richtung der Orientierung

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

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

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

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

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

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

- Globale Konstanten
  - Das Raumschiff

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

  - Fenstergröße

```
winSize :: (Int, Int)
winSize = (800, 600)
```

  - Schub

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

  - Maximale Geschwindigkeit

```
vMax :: Double
vMax = 20
```

- Winkelbeschleunigung

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

- Der Anfangszustand: Links oben, nach Süden gerichtet

```
initialState :: State
initialState =
  State {ship= setShp $
     Ship{pos= (40, 40),
          vel= (0, 0), ornt= -pi/2,
          thrust= 0, hAcc= 0}}
```

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

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

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

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

  - Vektoraddition modulo Fenstergröße:

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

- Systemzustand darstellen
  - Gesamter Systemszustand

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

    ▷ Weitere Objekte mit `overGraphics` kombinieren

  - Schiff darstellen (Farbänderung bei Beschleunigung)

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

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

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

- Liste aller Eingaben seit dem letzten Tick:

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

- Folgezustand berechnen: später IO möglich (Zufallszahlen!)

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

- Eine Eingabe bearbeiten:

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

- Das Hauptprogramm

  - Fenster öffnen, Schleife mit Anfangszustand starten

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

Zusammenfassung

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

Vorlesung vom 20.01.2003: Effiziente Funktionale Programme

Inhalt

- Zeitbedarf: Endrekursion — `while` in Haskell
- Platzbedarf: Speicherlecks
- Verschiedene andere Performancefallen:
  - Überladene Funktionen
  - Listen
  - Referenzen

Endrekursion

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

- D.h. darüber nur `if`, `case`, guards oder Fallunterscheidungen.
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in Schleifen übersetzt.
- Nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` nicht endrekursiv:

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

- `fac` endrekursiv:

```
fac :: Int-> Int
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 -> T`

```
f' x = if B x then H x
                else φ (f' (K x)) (E x)
```

  - Mit K `:: S -> S`, ϕ `:: T -> T -> T`, E `:: S -> T`, H `:: S -> T`.

- Sei ϕ assoziativ ist, e`:: T` neutrales Element
- Dann ist die endrekursive Form:

```
f :: S -> T
f x = g x e where
 g x y = if B x then φ (H x) y
                else g (K x) (φ (E x) y)
```

Beispiel

- Länge einer Liste

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

- Zuordnung der Variablen:

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

- 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 0 -- was: 0+ 0
             else len (tail xs) (1+ y)
```

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

Endrekursive Aktionen

Eine Aktion ist endrekursiv, wenn nach dem rekursiven Aufruf keine weiteren Aktionen folgen.

- 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 instantiierte Funktion
- Beispiel: die Klasse `Show`
  - Nur `show` wäre zu langsam ($O(n^2)$):

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

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch closures
  - closure: partiell instantiierte Funktion
- Beispiel: die Klasse `Show`
  - Nur `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 ($O(n)$).

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

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

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

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

- Nicht perfekt— besser:

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

Speicherlecks

- Garbage collection gibt unbenutzten Speicher wieder frei.
  - Nicht mehr benutzt: Bezeichner nicht mehr im Scope.
- Eine Funktion hat ein Speicherleck, wenn Speicher belegt wird, der nicht mehr benötigt wird.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden nicht auswertet. (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 `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.
- `foldl` ist endrekursiv:

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

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

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

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

Gemeinsame Teilausdrücke

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

```
f :: Int-> Int
f x = (x+ 1)* (x+ 1)

f (trace "Foo" (3+2))
x + x where x = (trace "Bar" (3+2))
```

- *Sharing* von Teilausdrücken (wie `x`)
  - Explizit mit `where` oder `let`
  - Implizit (`ghc`)

Memoisation

- Kleine Änderung der Fibonacci-Zahlen als Strom:

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

- Große Wirkung: Exponentiell in Space/Time. Warum?

Memoisation

- Kleine Änderung der Fibonacci-Zahlen als Strom:

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

- Große Wirkung: Exponentiell in Space/Time. Warum?
  - Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.
- Memoisation: Bereits berechnete Ergebnisse speichern.
- In Hugs: Aus dem Modul `Memo`:

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

- Damit ist alles wieder gut (oder?)

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

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: dictionaries 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: `facs` hat den Typ `Num a=> a-> a`

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

Listen als Performance-Falle

- Listen sind keine Felder.
- Listen:
  - Beliebig lang
  - 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

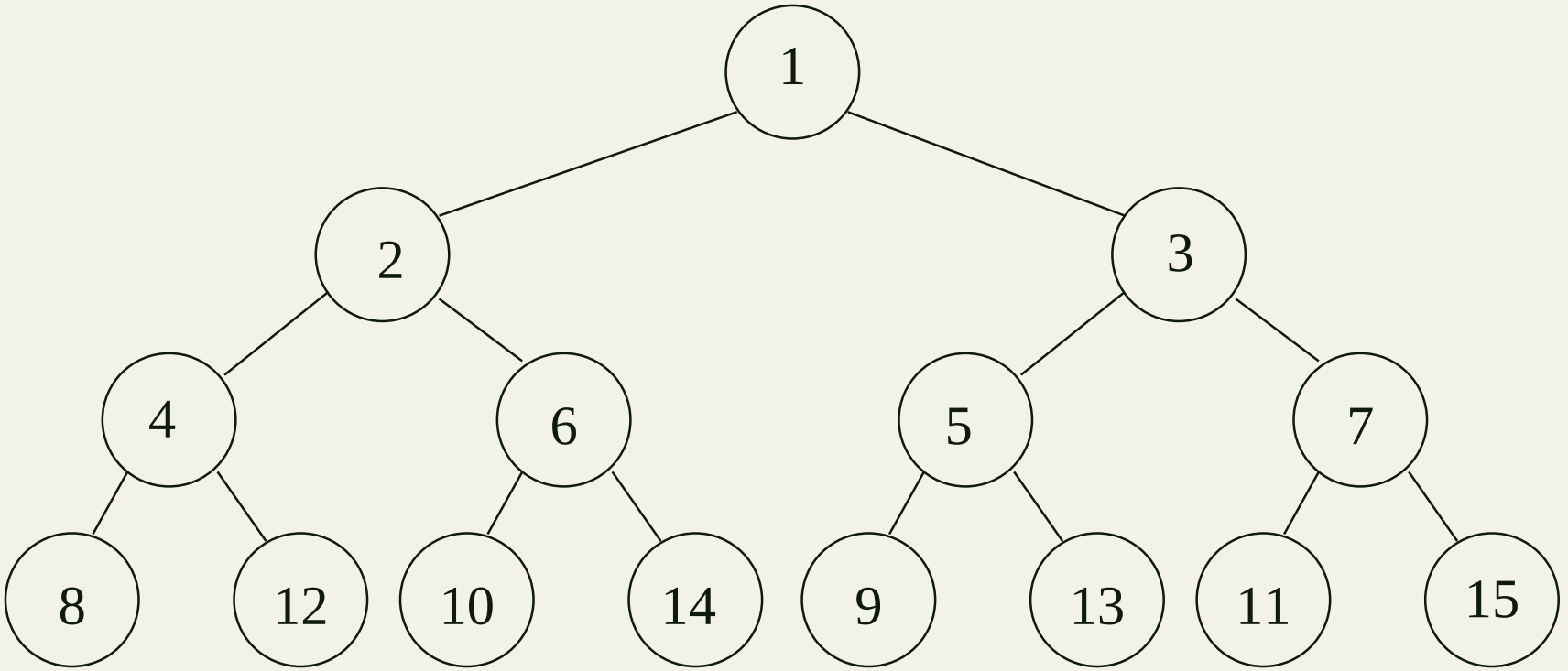
- Modul `Array` aus der Standardbücherei

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

- Als Indexbereich geeignete Typen (Klasse `Ix`): `Int`, `Integer`, `Char`, `Bool`, Tupel davon, Aufzählungstypen.
- In Hugs/GHC vordefiniert (als "primitiver" Datentyp)

Funktionale Arrays

- Idee: Arrays implementiert durch binäre Bäume
- Pfad im Index kodiert: 0 — links, 1 — rechts:



- Schnittstelle:

```
module FArray(
  Array, -- abstract
  (!),  -- :: Array a-> Int-> Maybe a,
  upd,  -- :: Array a-> Int-> a-> Array a,
  remv, -- :: Array a-> Int-> Array
  ) where
```

- Der Basisdatentyp ist ein binärer Baum:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
                   deriving (Show, Read)
type Array a = Tree a
```

  - Invariante: zusammenhängende Indizes.

- Lookup: Baumtraversion

```
(!) :: Tree a -> Int-> Maybe a
Leaf ! _                = Nothing
(Node v t1 t2) ! k
    | k == 1            = Just v
    | k `mod` 2 == 0 = t1 ! (k `div` 2)
    | otherwise         = t2 ! (k `div` 2)
```

- Update: ähnlich.
  - Feld kann erweitert werden, aber immer nur um ein Element!

```
upd :: Tree a -> Int-> a-> Tree a
upd Leaf k v =
  if k== 1 then (Node v Leaf Leaf)
  else error "Tree.upd: illegal index"
upd (Node w t1 t2) k v
  | k == 1     = Node v t1 t2
  | k `mod` 2 == 0
               = Node w (upd t1 (k `div` 2) v) t2
  | otherwise = Node w t1 (upd t2 (k `div` 2) v)
```

- Remove: darf nur für oberstes Element angewandt werden.

```
remv :: Tree a-> Int-> Tree a
remv Leaf _  = Leaf
remv (Node w t1 t2) k
  | k == 1     = Leaf
  | k `mod` 2 == 0
               = Node w (remv t1 (k `div` 2)) t2
  | otherwise = Node w t1 (remv t2 (k `div` 2))
```

- Mangel: Invariante wird bei `remv` nicht geprüft.
- Einfache Abhilfe:

```
type Array a = (Tree a, Int)
```

Zustände und Referenzen

- Zugriff auf Referenzen: Seiteneffekt —Aktion
- In Hugs und GHC: Modul `IORef`
- Abstrakter Datentyp von Referenzen:

```
data IORef a  -- abstract, instance of: Eq
```

- Neue Referenz erzeugen:

```
newIORef    :: a -> IO (IORef a)
```

- Referenz beschreiben:

```
writeIORef  :: IORef a -> a -> IO ()
```

- Referenz auslesen:

```
readIORef   :: IORef a -> IO a
```

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 Arrays.
- Referenzen in Haskell: `IORef`

Vorlesung vom 27.01.2003: Roboter!

Inhalt

- Domain Specific Languages
- Modellierung von Zuständen: Monaden
- Beispiel für eine DSL: Roboterkontrollsprache IRL
  - Nach Hudak, Kapitel 19.

Domain Specifc Languages (DSL)

- DSL: Sprache für speziellen Problembereich
  - Im Gegensatz zu universalen Programmiersprachen
  - Beispiel: LaTeX, Shell-Skripte, Spreadsheets, . . .
- Implementierung von DSLs:
  - Einbettung in Haskell
- Beispiel heute: Imperative Roboterkontrollsprache IRL
- Dazu: imperative Konzepte in Haskell

Zustandsübergangsmonaden

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

Zustandsübergangsmonaden

- Komposition von Zustandsübergängen:
  - Im Prinzip Vorwärtskomposition `(>.>)`:

```
(>>=) :: ST s a-> (a-> ST s b)-> ST s b
(>>=) :: (s-> (a, s))-> (a-> s->  (b, s))-> (s-> (b, s))
(>>=) :: (s-> (a, s))-> ((a, s)-> (b, s))-> (s-> (b, s))
```

  - Damit `f >>= g = uncurry g . f`.
  - Aber: Konstruktor um `ST`, deshalb Hilfsfunktion zum Auspacken.
- Identität für Zustandstransformationen:
  - Identität auf dem Zustand, Identität auf dem Argument.

- Damit:

```
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))
```

- Ausführlicher :

```
instance Monad (ST s) where
  (ST f) >>= g = ST $ \s0-> let (r1, s1)= f s0
                                ST g0   = g r1
                            in  g0 s1
  return a = ST $ \s-> (a, s)
```

Aktionen

- Aktionen: Zustandstransformationen auf der Welt
- Typ `RealWorld#` repräsentiert Außenwelt
  - Typ hat genau einen Wert `realworld#`, der nur für initialen 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`
  - Bewegung `move :: Robot ()`, `turnLeft :: Robot ()`
  - Roboter kann zeichnen: `penUp :: Robot()`, `penDown :: Robot()`
  - Damit: 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
cond1 :: Robot Bool-> Robot ()-> Robot ()
while :: Robot Bool-> Robot ()-> Robot ()
```

- Bsp: Ausweichen

```
evade :: Robot ()
evade = do cond1 blocked turnRight
```

- Bsp: Auf nächste Wand zufahren:

```
moveToWall :: Robot ()
moveToWall = while (isnt blocked)
                   move
```

Roboter auf Schatzsuche

- Welt enhält 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
```

- `Robot a` transformiert Robotstate.
  - 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` als Instanz von `Monad` (vergleiche `ST`):

```
instance Monad Robot where
  Robot sf0 >>= f
    = Robot $ \s0 g w -> do
                (s1,a1) <- sf0 s0 g w
                let Robot sf1 = f a1
                sf1 s1 g w
  return a
    = Robot (\s _ _ -> return (s,a))
```

- Positionen:

```
type Position = (Int,Int)
```

- Richtungen:

```
data Direction = North | East | South | West
     deriving (Eq,Show,Enum)
```

  - Hilfsfunktionen: Rechts-/Linksdrehungen:

```
right,left :: Direction -> Direction
right d = toEnum (succ (fromEnum d) `mod` 4)
left  d = toEnum (pred (fromEnum d) `mod` 4)
```

- Die Welt:

```
type Grid = Array Position [Direction]
```

  - Enthält für Feld `(x,y)` die Richtungen, in denen erreichbare Nachbarfelder sind.

Einfache Zustandsmanipulationen

- Zwei Hilfsfunktionen

```
updateState :: (RobotState -> RobotState)
                                -> Robot ()
queryState  :: (RobotState -> a) -> Robot a
```

- Damit Implementation der einfachen Funktionen (`turnLeft`, `turnRight`, `coins`, &c)
- Einzige Ausnahme: `blocked`.
- Zeigen.

Bewegung

- Beim Bewegen:
  - Prüfen, ob Bewegung möglich
  - Neue Position des Roboters zeichnen
  - Neue Position in Zustand eintragen.

```
move :: Robot ()
move = cond1 (isnt blocked) $
         Robot $ \s _ w -> do
            let newPos = movePos (position s) (facing s)
            graphicsMove w s newPos
            return (s {position = newPos}, ())

movePos :: Position -> Direction -> Position
```

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
- Zeigen.

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:

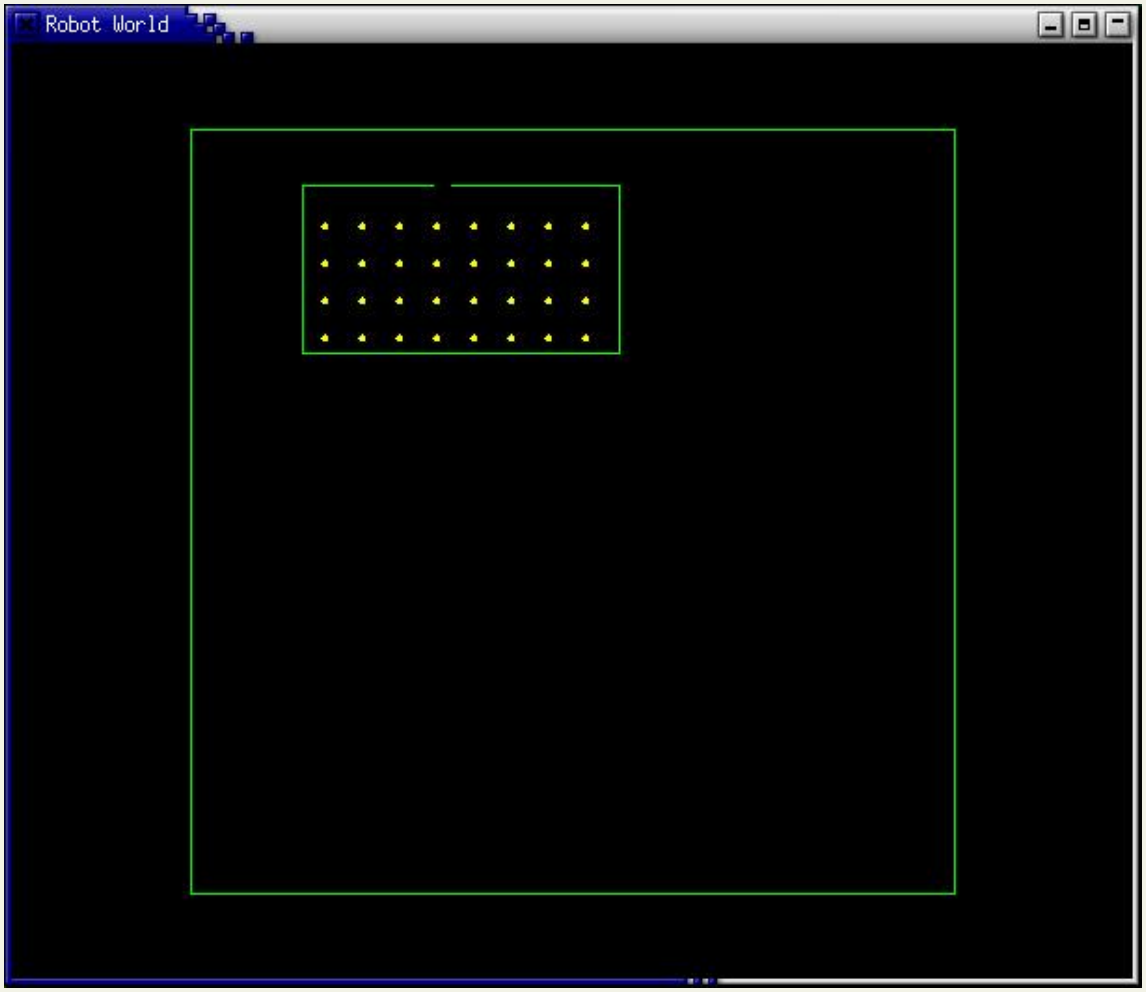
```
spiral :: Robot ()
spiral = penDown >> loop 1 where
 loop n =
   let turnMove = do turnRight; for n move
   in  do for 2 turnMove
          cond1 (isnt blocked) (loop (n+1))

main :: IO ()
main = runRobot spiral s0 g0
```

- Zeigen.

Beispiel 2

Eine etwas komplizierte Welt:



- Ziel: in dieser Welt 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 =
  cond1 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
  cond1 (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

- Zustandstransformationen
  - Aktionen als Transformationen der `RealWorld`
- Die Roboterkontrollsprache IRL
  - Einbettung einer imperativen Sprache in Haskell
  - Der Robotersimulator
- Beispiel für eine domänenspezifische Sprache (DSL).
  - Hier in Haskell eingebettet.
  - Wegen flexibler Syntax, Typklassen und Funktionen höherer Ordnung gut möglich.

Vorlesung vom 03.02.2002
Schlußbemerkungen

Inhalt der Vorlesung

- Organisatorisches
- Noch ein paar Haskell-Döntjes:
  - Web-Scripting mit Haskell
  - Concurrent 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 8110) abgeben (oder zum Fachgespräch mitbringen)
- Nicht vergessen: in Liste eintragen!.

Das Fachgespräch

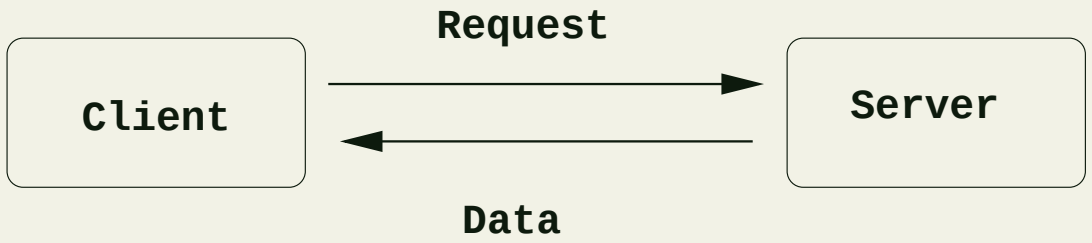
- Dient zur Überprüfung der Individualität der Leistung.
  - Insbesondere: Teilnahme an Beabeitung der Übungsblätter.
  - Keine Prüfung.
- Dauer: ca. 5–10 Min; einzeln, auf Wunsch mit Beisitzer
- Inhalt: Übungsblätter
- Bearbeitete Übungsblätter mitbringen — es werden zwei Aufgaben besprochen, die erste könnt Ihr Euch aussuchen.
- Termine:
  - Do. 06.02, Do. 13.02 — Liste vor MZH 8110
  - oder nach Vereinbarung.

Fallbeispiel: Lehrveranstaltungsevaluation

- Ziel: Webformular zur Evaluation der Lehrveranstaltung
- Erfordert Auswertung auf Serverseite: *server side scripting*

Medieninformatik für Einsteiger

- Das Web:



- `Request` definiert durch Transferprotokoll HTTP
- Datenformat definiert durch MIME
- HTML ist ein spezielles Format

HTTP: Hypertext Transfer Protocol

Client (Webbrowser)	Server (e.g. `apache`)
`connect`	`listen` (Port 80)
`GET index.html`	
	lese Datei `index.html`
	sende Inhalt
Inhalt von `index.html` empfangen	
HTML als Grafik darstellen	

Verschiedene Dokumenttypen: MIME

- Dokumente sind `getypt`: MIME
  - (Multipurpose Internet Mail Extensions, `RFC1521`)
- MIME-Typ: `typ/subtyp`, wie in

  `text/plain`

  `text/html`

  `image/gif`

  `application/postscript`

  `x-application/url-encoded`
- Typ bestimmt Darstellung — HTML ein Typ von vielen

CGI-Skripte *(server side scripting)*

- CGI-Skripte liegen unter
  `/home/www/users/cxl/cgi-bin/<name>`,
  Aufruf als
  `http://www....de/cgi-bin/cgiwrap/cxl/<name>`
- CGI-Skripte werden auf dem Server ausgeführt,
  Ausgabe wird an Client übertragen
- Bei Ausgabe Syntax beachten:

```
Content-Type: text/html

<HTML><BODY><H1>Hello, World</H1></BODY></HTML>
```

Formulare

`Formulare` sind spezielle HTML-Elemente zum Datenaustausch zwischen Client und Server:

Element	Attribut
`FORM`	`ACTION`: `POST`, `GET`
	`METHOD`: URL
`INPUT`	`TYPE`: `TEXT`, `CHECKBOX`, `RADIO`, `SUBMIT`, `RESET` . . .
	`NAME`, `VALUE`, `CHECKED`, `SIZE`, `MAXLENGTH`
`TEXTAREA`	Texteditor
	`NAME`, `ROWS`, `COLS`

Was passiert beim Abschicken *(submit)* eines Formulars?

- Client kodiert Formulardaten als URL
- Server erkennt URL als CGI-Skript und ruft CGI-Skript auf
- Kodierte Daten werden in Kommandozeile/Umgebungsvariablen übergeben
- Ergebnis wird an den Client zurückgeschickt

Ein einfaches Beispiel

```
<form method=get action=http://www...de/cgi-lokal/test>
Name: <input type=text name=name size=20 maxlength=30>
Tick: <input name=ticked value=y type=checkbox>
      <input type=submit>
</form>
```

Kodierung der Daten

- Bei `ACTION=mailto:<adress>` wird Dokument geschickt
- `GET` übergibt Argumente in URL,
  `POST` in Umgebungsvariablen
- `GET` für Anfragen, `POST` für Änderungen
- Beispiel vorherige Seite:

  Methode `GET`: `test?name=Hello+there%3f&ticked=y`

  Methode `POST`: Umgebungsvar. `name`: `Hello+there%3f`,
  `ticked`: `y`

Server Side Web Scripting in Haskell

- Ziel: Trennung von Funktionalität und Repräsentation

```
worker :: Request-> IO (CgiOut a)

wrapper :: (Request-> IO (CgiOut a)) -> IO ()
```

`wrapper` dekodiert URL, erzeugt HTML-Text

- Trennung von Funktionalität und Abstraktion
- Flexibilität: leicht auf anderes Format (e.g. ASP) anzupassen

MIME und HTML in Haskell

- `MIME`-Dokumente repräsentiert durch Typklasse `MIME`
- HTML ist ein abstrakter Datentyp `HTML`
  - `HTML` ist Instanz von `MIME`
  - Vordefinierte Kombinatoren:

```
prose                    :: String -> HTML
h1,h2,h3,h4,h5,h6,h7     :: String -> HTML
href                     :: URL -> [HTML] -> HTML
```

Server Side Scripting Made Very Easy

- Anfragen vom Typ `Request = [(Name, Value)]`
- Skriptfunktion liefert Wert vom Typ `CgiOut a`:

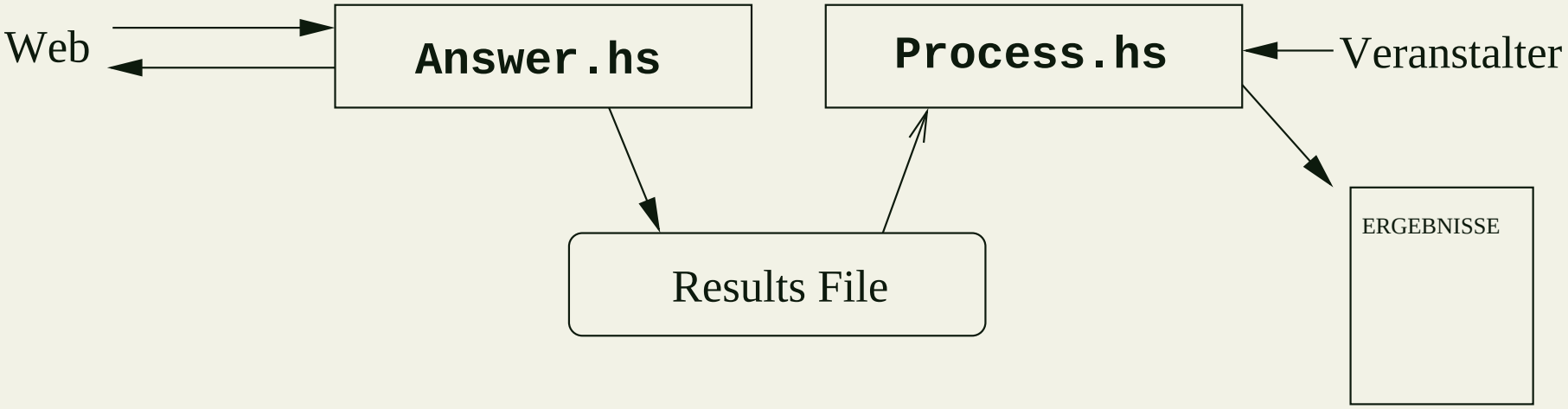
```
module Main (main) where

import CGI

helloWorldHTML :: [(Name,Value)] -> HTML
helloWorldHTML env =
   page "Hello World" [] [h1 "Hello World!"]
main :: IO ()
main = wrapper (\env ->
       do{return (Content{mime= helloWorldHTML env})})
```

Das PI3-Evaluationssystem

- Das Formular
- Gemeinsame Daten: `EvalData.hs`
- CGI-Skript: `Answer.hs`
- Auswertung: `Process.hs` Zeigen



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 '.'
```

Grafische Benutzerschnittstellen

- HTk
  - Verkapselung von Tcl/Tk in Haskell
  - Nebenläufig mit Events
  - Entwickelt an der AG BKB (Dissertation E. Karlsen)
  - Mächtig, abstrakte Schnittstelle, mittelprächtige Grafik
- GTk+HS
  - Verkapselung von GTk+ in Haskell
  - Zustandsbasiert mit call-backs
  - Entwickelt an der UNSW (M. Chakravarty)
  - Neueres Tookit, ästethischer, nicht ganz so mächtig

Grafische Benutzerschnittstellen mit HTk

- Statischer Teil: Aufbau des GUI
  - Hauptfenster öffnen

```
main:: IO ()
main =
  do main <- initHTk []
```

  - Knopf erzeugen und in Hauptfenster plazieren

```
b <- newButton main [text "Press me!"]
pack b []
```

- Dynamischer Teil: Verhalten während der Laufzeit
  - Knopfdruck als Event

```
click <- clicked b
```

  - Eventhandler aufsetzen

```
spawnEvent
 (forever
```

  - Eventhandler definieren

```
    (click >>> do nu_label <- mapM randomRIO
                                  (replicate 5 ('a','z'))
                  b # text nu_label))
finishHTk
```

- Zeigen. (Etwas längeres Beispiel.)

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

Warum funktionale Programmierung lernen?

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

Hat es sich gelohnt?

Hilfe!

- Haskell: primäre Entwicklungssprache an der AG BKB
  - Entwicklungsumgebung für formale Methoden (Uniform Workbench)
  - Werkzeuge für die Spezifikationssprache CASL (Tool Set CATS)
- Wir suchen studentische Hilfskräfte
  - für diese Projekte
- Wir bieten:
  - Angenehmes Arbeitsumfeld
  - Interessante Tätigkeit
- Wir suchen Tutoren für PI3
  - im WS 03/04 — meldet Euch!

Tschüß!