

Einführung in die Funktionale Programmierung:

Haskell

Prof. Dr. Manfred Schmidt-Schauß

WS 2019/20

Stand der Folien: 2. November 2019

Ziele des Kapitels

- Übersicht über die Konstrukte von Haskell
- Übersetzung der Konstrukte in KFPTSP+seq

Wir erörtern **nicht**:

- Die Übersetzung von `let` und `where`, da kompliziert.
- Aber: Übersetzung in KFPTSP+seq ist möglich durch sog. Fixpunktkombinatoren

Übersicht

- 1 Zahlen
- 2 Algebraische Datentypen
 - Aufzählungstypen
 - Produkttypen
 - Parametrisierte Datentypen
 - Rekursive Datentypen
- 3 Listen
 - Listenfunktionen
 - Ströme
 - Weitere
 - List Comprehensions
- 4 Bäume
 - Datentypen für Bäume
 - Syntaxbäume
- 5 Typdefinitionen

Zahlen in Haskell und KFPTSP+seq

Eingebaute Zahlen in Haskell – Peano-Kodierung für KFPTSP+seq

Haskell: Zahlen

Eingebaut:

- Ganze Zahlen beschränkter Größe: `Int`
- Ganze Zahlen beliebiger Größe: `Integer`
- Gleitkommazahlen: `Float`
- Gleitkommazahlen mit doppelter Genauigkeit: `Double`
- Rationale Zahlen: `Rational`
(verallgemeinert `Ratio α`, wobei
`Rational = Ratio Integer`)

Präfix / Infix

Anmerkung zum Minuszeichen:

- Mehr Klammern als man denkt: `5 + -6` geht nicht,
richtig: `5 + (-6)`
- In Haskell können Präfix-Operatoren (Funktionen) auch infix
benutzt werden
- `mod 5 6` ; infix durch Hochkommata: `5 'mod' 6`
- Umgekehrt können infix-Operatoren auch präfix benutzt
werden
- `5 + 6` ; präfix durch Klammern: `(+) 5 6`

Arithmetische Operationen

Rechenoperationen:

- `+` für die Addition
- `-` für die Subtraktion
- `*` für die Multiplikation
- `/` für die Division
- `mod` , `div`

Die Operatoren sind **überladen**. Dafür gibt es **Typklassen**.

Typ von `(+)` :: `Num a => a -> a -> a`

Genaue Behandlung von Typklassen: **später**

Vergleichsoperatoren

- `==` für den Gleichheitstest
`(==) :: (Eq a) => a -> a -> Bool`
- `/=` für den Ungleichheitstest
- `<`, `<=`, `>`, `>=`, für kleiner, kleiner gleich, größer
und größer gleich
(der Typ ist `(Ord a) => a -> a -> Bool`).

Assoziativitäten und Prioritäten

```

infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, 'quot', 'rem', 'div', 'mod'
infixl 6  +, -

-- The (:) operator is built-in syntax, and cannot
-- legally be given a fixity declaration; but its
-- fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  <<
infixr 0  $, $!, 'seq'

```

Darstellung von Zahlen in KFPTSP+seq

Mögliche Kodierung von Zahlen in KFPTSP+seq: **Peano-Zahlen**:

- **Peano-Zahlen** sind aus **Zero** und (**Succ** **Peano-Zahl**) aufgebaut
- nach dem italienischen Mathematiker Guiseppe Peano benannt

```

data Pint = Zero | Succ Pint
deriving (Eq, Show)

```

Übersetzung:

$$\begin{aligned} \mathcal{P}(0) &:= \text{Zero} \\ \mathcal{P}(n) &:= \text{Succ}(\mathcal{P}(n-1)) \text{ für } n > 0 \end{aligned}$$

Z.B. wird 3 dargestellt als $\text{Succ}(\text{Succ}(\text{Succ}(\text{Zero})))$.

Funktionen auf Peano-Zahlen

```

istZahl :: Pint -> Bool
istZahl x = case x of
    Zero    -> True
    (Succ y) -> istZahl y

```

Keine echte Zahl:

```

unendlich :: Pint
unendlich = Succ unendlich

```

Addition:

```

peanoPlus :: Pint -> Pint -> Pint
peanoPlus x y = if istZahl x && istZahl y then plus x y else bot
where
    plus x y = case x of
        Zero    -> y
        Succ z  -> Succ (plus z y)
    bot = bot

```

Funktionen auf Peano-Zahlen (2)

Multiplikation:

```

peanoMult :: Pint -> Pint -> Pint
peanoMult x y = if istZahl x && istZahl y then mult x y else bot
where
    mult x y = case x of
        Zero    -> Zero
        Succ z  -> peanoPlus y (mult z y)

```

Funktionen auf Peano-Zahlen (2)

Vergleiche:

```

peanoEq :: Pint -> Pint -> Bool
peanoEq x y = if istZahl x && istZahl y then eq x y else bot
where
  eq Zero Zero      = True
  eq (Succ x) (Succ y) = eq x y
  eq _ _            = False

peanoLeq :: Pint -> Pint -> Bool
peanoLeq x y = if istZahl x && istZahl y then leq x y else bot
where
  leq Zero y        = True
  leq x Zero        = False
  leq (Succ x) (Succ y) = leq x y

```

Algebraische Datentypen in Haskell

Aufzählungstypen – Produkttypen – Parametrisierte Datentypen –
Rekursive Datentypen

Aufzählungstypen

Aufzählungstyp = Aufzählung verschiedener Werte

```
data Typname = Konstante1 | Konstante2 | ... | KonstanteN
```

Beispiele:

```

data Bool      = True | False

data Wochentag = Montag | Dienstag | Mittwoch | Donnerstag
               | Freitag | Samstag | Sonntag
  deriving(Show)

```

deriving(Show) erzeugt Instanz der Typklasse Show, damit der Datentyp angezeigt werden kann.

Aufzählungstypen (2)

```

istMontag :: Wochentag -> Bool
istMontag x = case x of
  Montag -> True
  Dienstag -> False
  Mittwoch -> False
  Donnerstag -> False
  Freitag -> False
  Samstag -> False
  Sonntag -> False

```

In Haskell erlaubt (in KFTSP+seq nicht):

```

istMontag' :: Wochentag -> Bool
istMontag' x = case x of
  Montag -> True
  y       -> False

```

Übersetzung: Aus istMontag' wird istMontag

Aufzählungstypen (3)

In Haskell:

Pattern-matching in den linken Seiten der SK-Definition:

```
istMontag'' :: Wochentag -> Bool
istMontag'' Montag = True
istMontag'' _      = False
```

Übersetzung: Erzeuge case-Ausdruck

```
istMontag'' xs = case xs of
    Montag -> True
    ...    -> False
```

Produkttypen

Produkttyp = Zusammenfassung verschiedener Werte

Bekanntes Beispiel: Tupel

```
data Typname = KonstruktorName Typ1 Typ2 ... TypN
```

Beispiel:

```
data Student = Student
    String -- Name
    String -- Vorname
    Int    -- Matrikelnummer
```

Produkttypen (2)

```
setzeName :: Student -> String -> Student
setzeName x name' =
    case x of
        (Student name vorname mnr)
            -> Student name' vorname mnr
```

Alternativ mit Pattern auf der linken Seite der Funktionsdefinition:

```
setzeName :: Student -> String -> Student
setzeName (Student name vorname mnr) name' =
    Student name' vorname mnr
```

Produkttypen und Aufzählungstypen

Man kann beides mischen:

```
data DreiDObjekt =
    Wuerfel Int
  | Quader Int Int Int
  | Kugel Int
```

Wird auch als **Summentyp** bezeichnet, allgemein

```
data Summentyp = Konsdef1 | Konsdef2 | ... | Konsdefn
```

wobei `Konsdef1 ... Konsdefn` Konstruktor-Definition mit Argument-Typen sind (z.B. Produkttypen)

Record-Syntax: Einführung

```
data Student = Student
    String -- Vorname
    String -- Name
    Int    -- Matrikelnummer
```

Nachteil:

Nur die **Kommentare** verraten, was die Komponenten darstellen.

Außerdem **mühsam**: Zugriffsfunktionen erstellen:

```
vorname :: Student -> String
vorname (Student vorname name mnr) = vorname
```

Record-Syntax: Einführung (2)

Änderung am Datentyp:

```
data Student = Student
    String -- Vorname
    String -- Name
    Int    -- Matrikelnummer
    Int    -- Hochschulesemester
```

muss für Zugriffsfunktionen nachgezogen werden

```
vorname :: Student -> String
vorname (Student vorname name mnr hsem) = vorname
```

Abhilfe verschafft die **Record-Syntax**

Record-Syntax in Haskell

Student mit Record-Syntax:

```
data Student = Student {
    vorname      :: String,
    name         :: String,
    matrikelnummer :: Int
}
```

Zur Erinnerung: Ohne Record-Syntax:

```
data Student = Student String String Int
```

⇒ Die Komponenten werden mit **Namen** markiert

Beispiel

Beispiel: Student "Hans" "Mueller" 1234567

kann man schreiben als

```
Student{vorname="Hans", name="Mueller", matrikelnummer=1234567}
```

Reihenfolge der Komponenten egal:

```
Prelude> let x = Student{matrikelnummer=1234567,
    vorname="Hans", name="Mueller"} ↵
```

Record-Syntax

Zugriffsfunktionen sind automatisch verfügbar, z.B.

```
Prelude> matrikelnummer x
1234567
```

- Record-Syntax ist in den Pattern erlaubt
- Nicht alle Felder müssen abgedeckt werden bei Erweiterung der Datenstrukturen, daher kein Problem

```
nachnameMitA Student{nachname = 'A':xs} = True
nachnameMitA _ = False
```

Übersetzung in KFTSP+seq:

Normale Datentypen verwenden
und Zugriffsfunktionen erzeugen

Record-Syntax: Update

```
setzeName :: Student -> String -> Student
setzeName student neuename =
    student {name = neuename}
```

ist äquivalent zu

```
setzeName :: Student -> String -> Student
setzeName student neuename =
    Student {vorname      = vorname student,
             name         = neuename,
             matrikelnummer = matrikelnummer student}
```

Parametrisierte Datentypen

Datentypen in Haskell dürfen **polymorph parametrisiert** sein:

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

Maybe ist polymorph über a

Beispiel für Maybe-Verwendung:

```
safeHead :: [a] -> Maybe a
safeHead xs = case xs of
    [] -> Nothing
    (y:ys) -> Just y
```

Rekursive Datentypen

Rekursive Datentypen:

Der definierte Typ kommt rechts vom = wieder vor

```
data Typ = ... Konstruktor Typ ...
```

Pint war bereits rekursiv:

```
data Pint = Zero | Succ Pint
```

Listen könnte man definieren als:

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

In Haskell, eher Spezialsyntax:

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

Haskell: Geschachtelte Pattern

```
viertesElement (x1:(x2:(x3:(x4:xs))) = Just x4
viertesElement _ = Nothing
```

Übersetzung in KFTSP+seq muss geschachtelte case-Ausdrücke einführen:

```
viertesElement ys = case ys of
  [] -> Nothing
  (x1:ys') ->
    case ys' of
      [] -> Nothing
      (x2:ys'') ->
        case ys'' of
          [] -> Nothing
          (x3:ys''') ->
            case ys''' of
              [] -> Nothing
              (x4:xs) -> Just x4
```

Rekursive Datenstrukturen: Listen

Listenfunktionen – Listen als Ströme – List Comprehensions

Listen von Zahlen

Haskell: spezielle Syntax

- `[startwert..endwert]`

erzeugt: Liste der Zahlen von startwert bis endwert

z.B. ergibt `[10..15]` die Liste `[10,11,12,13,14,15]`.

- `[startwert..]`

erzeugt: unendliche Liste ab dem startwert

z.B. erzeugt `[1..]` die Liste aller natürlichen Zahlen.

Listen von Zahlen (2)

- `[startwert,naechsterWert..endwert]`

erzeugt:

`[startwert,startWert+delta,startWert+2delta,...,endwert]`

wobei `delta=naechsterWert - startWert`

Z.B. ergibt: `[10,12..20]` die Liste `[10,12,14,16,18,20]`.

- `[startWert,naechsterWert..]`

erzeugt: die unendlich lange Liste mit der Schrittweite

`naechsterWert - startWert`.

z.B. `[2,4..]` ergibt Liste aller geraden natürlichen Zahlen

Listen von Zahlen (3)

Nur syntaktischer Zucker, normale Funktionen für den Datentyp Integer:

```
from :: Integer -> [Integer]
from start = start:(from (start+1))

fromTo :: Integer -> Integer -> [Integer]
fromTo start end
  | start > end    = []
  | otherwise = start:(fromTo (start+1) end)

fromThen :: Integer -> Integer -> [Integer]
fromThen start next = start:(fromThen next (2*next - start))

fromThenTo :: Integer -> Integer -> Integer -> [Integer]
fromThenTo start next end
  | start > end = []
  | otherwise  = start:(fromThenTo next (2*next - start) end)
```

Guards

```
f pat1 ... patn
  | guard1 = e1
  | ...
  | guardn = en
```

- Dabei: `guard1` bis `guardn` sind **Boolesche** Ausdrücke, die die Variablen der Pattern `pat1, ..., patn` benutzen dürfen.
- Auswertung von oben nach unten
- erster Guard der zu True auswertet bestimmt Wert.
- `otherwise = True` ist vordefiniert

Übersetzung von Guards in KFPTSP+seq

```
f pat1 ... patn
  | guard1 = e1
  | ...
  | guardn = en
```

ergibt (if-then-else muss noch übersetzt werden):

```
f pat1 ... patn =
  if guard1 then e1 else
  if guard2 then e2 else
  ...
  if guardn then en else s
```

Wobei `s = bot`, wenn keine weitere Funktionsdefinition für `f` kommt, anderenfalls ist `s` die Übersetzung anderer Definitionsgleichungen.

Beispiel

```
f (x:xs)
  | x < 10 = True
  | x > 100 = True
f ys      = False
```

Die korrekte Übersetzung in KFPTSP+seq (mit if-then else), unter der Annahme dass es Peano-Zahlen sind, ist:

```
f = case x of {
  Nil -> False;
  (x:xs) -> if x < 10 then True else
            if x > 100 then True else False
}
```

Zeichen und Zeichenketten

- Eingebauter Typ `Char` für Zeichen
- Darstellung: Einfaches Anführungszeichen, z.B. `'A'`
- Steuersymbole beginnen mit `\`, z.B. `\n`, `\t`
- Spezialsymbole `\\` und `\"`

Strings

- Vom Typ `String = [Char]`
- Sind Listen von Zeichen
- Spezialsyntax `"Hallo"` ist gleich zu
`['H','a','l','l','o']` bzw.
`'H':('a':('l':('l':('o':[]))))`.

Standard-Listenfunktionen

Einige vordefinierte Listenfunktionen, fast alle in `Data.List`

Zeichen und Zeichenketten (2)

- Nützliche Funktionen für `Char`: In der Bibliothek `Data.Char`

Z.B.:

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

isLower :: Char -> Bool
isUpper :: Char -> Bool
isAlpha :: Char -> Bool

toUpper :: Char -> Char
toLower :: Char -> Char
```

Standard-Listenfunktionen (1)

Append: `++`, Listen zusammenhängen

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

Beispiele:

```
*> [[1..10] ++ [100..109]]
[1,2,3,4,5,6,7,8,9,10,100,101,102,103,104,105,106,107,108,109]
*> [[1,2],[2,3]] ++ [[3,4,5]]
[[1,2],[2,3],[3,4,5]]
*> "Infor" ++ "matik"
"Informatik"
```

Laufzeitverhalten: linear in der Länge der ersten Liste

Standard-Listenfunktionen (2)

Zugriff auf Listenelement per Index: !!

```
(!!) :: [a] -> Int -> a
[]      !! _ = error "Index too large"
(x:xs) !! 0 = x
(x:xs) !! i = xs !! (i-1)
```

Beispiele:

```
*> [1,2,3,4,5] !! 3 
4
*> [0,1,2,3,4,5] !! 3 
3
*> [0,1,2,3,4,5] !! 5 
5
*> [1,2,3,4,5] !! 5 
*** Exception: Prelude.(!!): index too large
```

Standard-Listenfunktionen (3)

Index eines Elements berechnen: elemIndex

```
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
elemIndex a xs = findInd 0 a xs
  where
    findInd i a [] = Nothing
    findInd i a (x:xs)
      | a == x    = Just i
      | otherwise = findInd (i+1) a xs
```

Beispiele:

```
*> elemIndex 1 [1,2,3] 
Just 0
*> elemIndex 1 [0,1,2,3] 
Just 1
*> elemIndex 1 [5,4,3,2] 
Nothing
*> elemIndex 1 [1,4,1,2] 
Just 0
```

Standard-Listenfunktionen (4)

Map: Funktion auf Listenelemente anwenden

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

Beispiele:

```
*> map (*3) [1..20] 
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60]
*> map not [True,False,False,True] 
[False,True,True,False]
*> map (^2) [1..10] 
[1,4,9,16,25,36,49,64,81,100]
*> map toUpper "Informatik" 
"INFORMATIK"
```

Standard-Listenfunktionen (5)

Filter: Elemente heraus filtern

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
  | f x      = x:(filter f xs)
  | otherwise = filter f xs
```

Beispiele:

```
*> filter (> 15) [10..20] 
[16,17,18,19,20]
*> filter isAlpha "2017 Informatik 2017" 
"Informatik"
*> filter (\x -> x > 5) [1..10] 
[6,7,8,9,10]
```

Standard-Listenfunktionen (6)

Analog zu filter: **remove**: Listenelemente entfernen

```
remove p xs = filter (not . p) xs
```

Der Kompositionsoperator (.) ist definiert als:

```
(f . g) x = f (g x)
```

alternativ:

```
remove p xs = filter (\x -> not (p x)) xs
```

Beispiele:

```
*> remove (\x -> x > 5) [1..10]
[1,2,3,4,5]
*> remove isAlpha "2017 Informatik 2017"
"2017 2017"
```

Standard-Listenfunktionen (7)

Length: Länge einer Liste

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

Beispiele:

```
*> length "Informatik"
10
*> length [2..20002]
20001
*> length [1..]
^CInterrupted
```

Standard-Listenfunktionen (8)

Length: Bessere Variante (konstanter Platz)

```
length :: [a] -> Int
length xs = length_it xs 0

length_it [] acc = acc
length_it (_:xs) acc = let acc' = 1+acc
                        in seq acc' (length_it xs acc')
```

Standard-Listenfunktionen (9)

Reverse: Umdrehen einer Liste

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]
```

Besser mit Stack: Laufzeit linear

```
reverse :: [a] -> [a]
reverse xs = rev xs []
  where rev [] acc = acc
        rev (x:xs) acc = rev xs (x:acc)
```

```
*> reverse [1..10]
[10,9,8,7,6,5,4,3,2,1]
*> reverse "RELIEFPFEILER"
"RELIEFPFEILER"
*> reverse [1..]
^C Interrupted
```


Standard-Listenfunktionen (14)

Bemerkung zu zip:

Man kann zwar zip3, zip4 etc. definieren um 3, 4, ..., Listen in 3-Tupel, 4-Tupel, etc. einzupacken, aber:

Man kann keine Funktion zipN für n Listen definieren, wobei n ein Argument ist.

Grund: diese Funktion wäre nicht getypt.

Standard-Listenfunktionen (15)

Verallgemeinerung von zip und map:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
zipWith _ _ _ = []
```

Damit kann man zip definieren:

```
zip = zipWith (\x y -> (x,y))
```

Anderes Beispiel:

```
vectorAdd :: (Num a) => [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

Standard-Listenfunktionen (16)

Die Fold-Funktionen:

- `foldl` $\otimes e [a_1, \dots, a_n]$ ergibt $(\dots((e \otimes a_1) \otimes a_2) \dots) \otimes a_n$
- `foldr` $\otimes e [a_1, \dots, a_n]$ ergibt $a_1 \otimes (a_2 \otimes (\dots \otimes (a_n \otimes e) \dots))$

Implementierung:

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (x:xs) = x 'f' (foldr f e xs)
```

`foldl` und `foldr` sind identisch, wenn die Elemente und der Operator $\otimes$ ein Monoid mit neutralem Element e bilden.

Standard-Listenfunktionen (17)

Concat:

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

Beachte: `foldl` bei append wäre ineffizienter!

```
sum = foldl (+) 0
product = foldl (*) 1
```

haben schlechten Platzbedarf, besser strikte Variante von `foldl`:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f e [] = e
foldl' f e (x:xs) = let e' = e 'f' x in e' 'seq' foldl' f e' xs
```

Standard-Listenfunktionen (18)

Beachte die Allgemeinheit der Typen von `foldl` / `foldr`

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

z.B. sind alle Elemente ungerade?

```
foldl (\xa xb -> xa && (odd xb)) True
```

`xa` und `xb` haben verschiedene Typen!

Analog mit `foldr`:

```
foldr (\xa xb -> (odd xa) && xb) True
```

Standard-Listenfunktionen (19)

Varianten von `foldl`, `foldr`:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [] = error "foldr1 on an empty list"
foldr1 _ [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
foldl1 _ [] = error "foldl1 on an empty list"
```

Beispiele

```
maximum :: (Ord a) => [a] -> a
maximum xs = foldl1 max xs

minimum :: (Ord a) => [a] -> a
minimum xs = foldl1 min xs
```

Standard-Listenfunktionen (20)

Scanl, **Scanr**: Zwischenergebnisse von `foldl`, `foldr`

- `scanl` $\otimes e [a_1, a_2, \dots, a_n] = [e, e \otimes a_1, (e \otimes a_1) \otimes a_2, \dots]$
- `scanr` $\otimes e [a_1, a_2, \dots, a_n] = [\dots, a_{n-1} \otimes (a_n \otimes e), a_n \otimes e, e]$

Es gilt:

- `last (scanl f e xs) = foldl f e xs`
- `head (scanr f e xs) = foldr f e xs`

Standard-Listenfunktionen (21)

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f e xs = e : case xs of
  [] -> []
  (y:ys) -> scanl f (e 'f' y) ys

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e [] = [e]
scanr f e (x:xs) = f x q : qs
  where qs@(q:_) = scanr f e xs
```

Anmerkung: "As"-Pattern `Var@Pat`

```
> scanr (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[1,2,3,4,5,6,7,8],[3,4,5,6,7,8],[5,6,7,8],[7,8],[]]
> scanl (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[],[1,2],[1,2,3,4],[1,2,3,4,5,6],[1,2,3,4,5,6,7,8]]
> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
> scanr (+) 0 [1..10]
```

Standard-Listenfunktionen (22)

Beispiele zur Verwendung von `scan`:

Fakultätsfolge:

```
faks = scanl (*) 1 [1..]
```

Z.B.

```
*> take 5 faks
[1,1,2,6,24,120]
```

Funktion, die alle Restlisten einer Liste berechnet:

```
tails xs = scanr (:) [] xs
```

Z.B.

```
*> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

Standard-Listenfunktionen (22b)

Funktionen, die alle Anfangslisten einer Liste berechnen:

```
map reverse (scanl (flip (:)) [] [1..100])
scanl (\x y-> x++[y]) [] [1..100]
map reverse (reverse (scanr (:) [] (reverse [1..100])))
```

Standard-Listenfunktionen (23)

Partitionieren einer Liste

```
partition p xs = (filter p xs, remove p xs)
```

Effizienter:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p [] = ([], [])
partition p (x:xs)
  | p x      = (x:r1,r2)
  | otherwise = (r1,x:r2)
  where (r1,r2) = partition p xs
```

Quicksort mit `partition`

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = let (kleiner,groesser) = partition (<x) xs
                    in quicksort kleiner ++ (x:(quicksort groesser))
```

Listen als Ströme (1)

- Listen in Haskell können **unendlich lang** sein
- Daher kann man Listen auch als Ströme auffassen
- Bei der Stromverarbeitung muss man beachten:
Nie versuchen die **gesamte Liste auszuwerten**
- D.h. Funktionen sollen **strom-produzierend** sein.
- Grobe Regel: Funktion `f :: [Int] -> [Int]` ist **strom-produzierend**, wenn `take n (f list)` für jede unendliche Liste und jedes `n` terminiert
- Ungeeignet daher: `reverse`, `length`, `foldl`,
- Geeignet: `map`, `filter`, `zipWith`, `take`, `drop`

Listen als Ströme (2)

Einige Stromfunktionen für Strings:

- `words :: String -> [String]`
Zerlegen einer Zeichenkette in eine Liste von Wörtern
- `lines :: String -> [String]`
Zerlegen einer Zeichenkette in eine Liste der Zeilen
- `unlines :: [String] -> String`
Einzelne Zeilen in einer Liste zu einem String zusammenfügen
(mit Zeilenumbrüchen)

Beispiele:

```
*> words "Haskell ist eine funktionale Programmiersprache"
["Haskell","ist","eine","funktionale","Programmiersprache"]
*> lines "1234\n5678\n90"
["1234","5678","90"]
*> unlines ["1234","5678","90"] "1234\n5678\n90\n"
```

Listen als Ströme (2)

Mischen zweier sortierter Ströme

```
merge :: (Ord t) => [t] -> [t] -> [t]
merge []      ys = ys
merge xs      [] = xs
merge a@(x:xs) b@(y:ys)
  | x <= y     = x:merge xs b
  | otherwise  = y:merge a ys
```

Beispiel:

```
*> merge [1,3,5,6,7,9] [2,3,4,5,6]
[1,2,3,3,4,5,5,6,6,7,9]
```

Listen als Ströme (3)

Doppelte Elemente entfernen

```
nub xs = nub' xs []
where
  nub' [] _ = []
  nub' (x:xs) seen
    | x `elem` seen = nub' xs seen
    | otherwise    = x : nub' xs (x:seen)
```

Anmerkungen:

- `seen` merkt sich die bereits gesehenen Elemente
- Laufzeit von `nub` ist quadratisch
(kann verbessert werden zu $O(n \log(n))$).

```
elem e [] = False
elem e (x:xs)
  | e == x = True
  | otherwise = elem e xs
```

Listen als Ströme (4)

Doppelte Elemente aus sortierter Liste entfernen:

```
nubSorted (x:y:xs)
  | x == y = nubSorted (y:xs)
  | otherwise = x:(nubSorted (y:xs))
nubSorted y = y
```

ist linear in der Länge der Liste.

Listen als Ströme (5)

Mischen der Vielfachen von 3,5 und 7:

```
*> nubSorted $ merge (map (3*) [1..])
*> (merge (map (5*) [1..]) (map (7*) [1..]))
[3,5,6,7,9,10,12,14,15,18,20,..]
```

Listen als Wörterbuch

Lookup

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key [] = Nothing
lookup key ((x,y):xys)
  | key == x = Just y
  | otherwise = lookup key xys
```

Beispiele:

```
*> lookup 5 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Just 'F'
*> lookup 3 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Nothing
```

Listen als Mengen (1)

Any und All: Wie Quantoren

```
any _ [] = False
any p (x:xs)
  | (p x) = True
  | otherwise = any p xs

all _ [] = True
all p (x:xs)
  | (p x) = all p xs
  | otherwise = False
```

Beispiele:

```
*> all even [1,2,3,4]
False
*> all even [2,4]
True
*> any even [1,2,3,4]
True
```

Listen als Mengen (2)

Delete: Löschen eines Elements

```
delete :: (Eq a) => a -> [a] -> [a]
delete e (x:xs)
  | e == x = xs
  | otherwise = x:(delete e xs)
```

Mengendifferenz: \

```
(\\) :: (Eq a) => [a] -> [a] -> [a]
(\\) = foldl (flip delete)
```

dabei dreht flip die Argumente einer Funktion um:

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

Listen als Mengen (2b)

Beispiele:

```
*> delete 3 [1,2,3,4,5,3,4,3]
[1,2,4,5,3,4,3]
*> [1,2,3,4,4] \\ [9,6,4,4,3,1]
[2]
*> [1,2,3,4] \\ [9,6,4,4,3,1]
[2]
```

Listen als Mengen (3)

Vereinigung und Schnitt

```
union :: (Eq a) => [a] -> [a] -> [a]
union xs ys = xs ++ (ys \\ xs)

intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys = filter (\y -> any (== y) ys) xs
```

```
*> union [1,2,3,4,4] [9,6,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> intersect [1,2,3,4,4] [4,4]
[4,4]
*> intersect [1,2,3,4] [4,4]
[4]
*> intersect [1,2,3,4,4] [4]
[4,4]
```

ConcatMap

Konkatinert die Ergebnislisten: ConcatMap

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

```
*> concatMap (\x-> take x [1..]) [3..7]
[1,2,3,1,2,3,4,1,2,3,4,5,1,2,3,4,5,6,1,2,3,4,5,6,7]
```

List Comprehensions

- Spezielle Syntax zur Erzeugung und Verarbeitung von Listen
- ZF-Ausdrücke (nach der Zermelo-Fränkel Mengenlehre)

Syntax: `[Expr | qual1,...,qualn]`

- Expr: ein Ausdruck
- $FV(Expr)$ sind durch qual1,...,qualn gebunden
- **quali** ist:
 - ein Generator der Form `pat <- Expr`, oder
 - ein Guard, d.h. ein Ausdruck booleschen Typs,
 - oder eine Deklaration lokaler Bindungen der Form `let x1=e1,...,xn=en` (ohne in-Ausdruck!) ist.

List Comprehensions: Beispiele

Liste der natürlichen Zahlen

```
[x | x <- [1..]]
```

Kartesisches Produkt

```
[(x,y) | x <- [1..], y <- [1..]]
```

```
*> take 10 [(x,y) | x <- [1..], y <- [1..]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
```

Liste aller ungeraden Zahlen

```
[x | x <- [1..], odd x]
```

Liste aller Quadratzahlen

```
[x*x | x <- [1..]]
```

List Comprehensions: Beispiele (2)

Liste aller Paare (Zahl, Quadrat der Zahl)

```
[(y,x*x) | x <- [1..], let y = x]
```

```
[a | (a,_,_,_) <- [(x,x,y,y) | x <- [1..3], y <- [1..3]]]
[1,1,1,2,2,2,3,3,3]
```

Map, Filter und Concat

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

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

```
concat xss = [y | xs <- xss, y <- xs]
```

List Comprehensions: Beispiele (3)

Quicksort:

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

Übersetzung in ZF-freies Haskell

```
[ e | True ]      = [e]
[ e | q ]         = [ e | q, True ]
[ e | b, q ]      = if b then [ e | q ] else []
[ e | p <- l, q ]  = let ok p = [ e | q ]
                    ok _ = []
                    in concatMap ok l
[ e | let decls, Q ] = let decls in [ e | Q ]
```

wobei

- ok eine neue Variable,
- b ein Guard,
- q ein Generator, eine lokale Bindung oder ein Guard (nicht True)
- Q eine Folge von Generatoren, Deklarationen und Guards.

Übersetzung in ZF-freies Haskell: Beispiel

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]
```

```
= let ok x = [x*y | y <- ys, x > 2, y < 3]
   ok _ = []
   in concatMap ok xs
```

```
= let ok x = let ok' y = [x*y | x > 2, y < 3]
               ok' _ = []
               in concatMap ok' ys
   ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
= let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
               ok' _ = []
               in concatMap ok' ys
   ok _ = []
   in concatMap ok xs
```

```
= let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
               ok' _ = []
               in concatMap ok' ys
   ok _ = []
   in concatMap ok xs
```

```
= let ok x = let ok' y = if x > 2 then
                           (if y < 3 then [x*y | True] else [])
                           else []
               ok' _ = []
               in concatMap ok' ys
   ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

```
= let ok x = let ok' y = if x > 2 then
                           (if y < 3 then [x*y] else [])
                           else []
               ok' _ = []
               in concatMap ok' ys
   ok _ = []
   in concatMap ok xs
```

Die Übersetzung funktioniert, aber ist nicht optimal,
da Listen generiert und wieder abgebaut werden;
und bei `x <- xs` unnötige Pattern-Fallunterscheidung

Rekursive Datenstrukturen: Bäume in Haskell

Binäre Bäume – N-äre Bäume – Funktionen auf Bäumen –
Syntaxbäume

Funktionen auf Bäumen (4)

Fold auf Bäumen

```
bFold op (Blatt a) = a
bFold op (Knoten a b) = op (bFold op a) (bFold op b)
```

Damit kann man z.B. die Summe und das Produkt berechnen:

```
*> bFold (+) beispielBaum 
66
*> bFold (*) beispielBaum 
39916800
```

Funktionen auf Bäumen (4b)

Allgemeineres Fold auf Bäumen:

```
foldbt :: (a -> b -> b) -> b -> BBaum a -> b
foldbt op a (Blatt x) = op x a
foldbt op a (Knoten x y) = (foldbt op a y) x
```

Der Typ des Ergebnisses kann anders sein als der Typ der Blattmarkierung

Zum Beispiel: Rand eines Baumes:

```
*> foldbt (:) [] beispielBaum 
[1,2,3,4,5,6,7,8,9,10,11]
```

Haskell Bäume Data.Tree

Data.Tree

Hackage-Bibliothek zu gelabelten n-ären Bäumen

```
data Tree a =
  Node {rootLabel :: a
        subForest :: Forest a }
type Forest a = [Tree a]
```

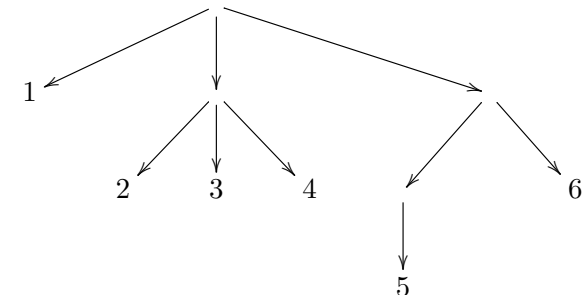
Tests

```
Data.Tree> let t1 = Node {rootLabel= 1, subForest = []} 
Data.Tree> let t2= Node{rootLabel= 2,subForest = [t1]} 
Data.Tree> t2 
Node {rootLabel = 2, subForest = [Node {rootLabel = 1,
  subForest = []}]}
```

N-äre Bäume

```
data NBaum a = NBlatt a | NKnoten [NBaum a]
deriving(Eq,Show)
```

```
beispiel = NKnoten [NBlatt 1,
  NKnoten [NBlatt 2, NBlatt 3, NBlatt 4],
  NKnoten [NKnoten [NBlatt 5], NBlatt 6]]
```

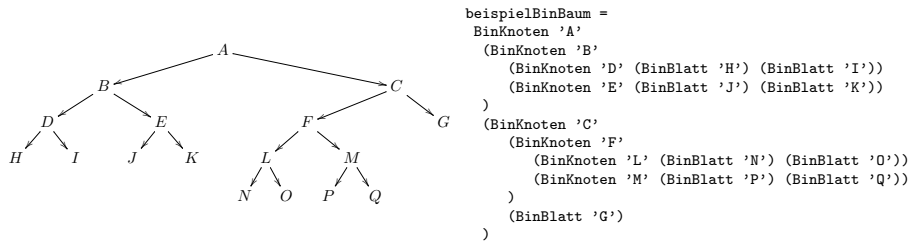


Bäume mit Knotenmarkierungen

Beachte: BBAum und NBAum haben nur Markierungen der Blätter!

Bäume mit Markierung aller Knoten

```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a)(BinBaum a)
  deriving(Eq,Show)
```

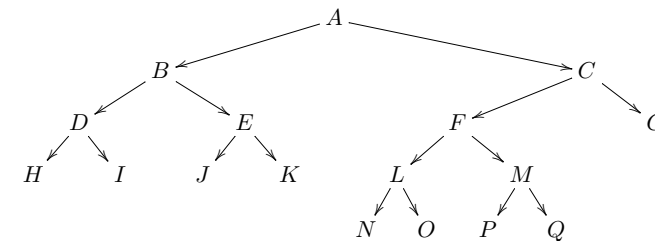


Funktionen auf BinBaum (1)

Knoten in Preorder-Reihenfolge (Wurzel, links, rechts):

```
preorder :: BinBaum t -> [t]
preorder (BinBlatt a)      = [a]
preorder (BinKnoten a l r) = a:(preorder l) ++ (preorder r)
```

preorder beispielBinBaum ----> "ABDHIEJKCFLNOMPQG"

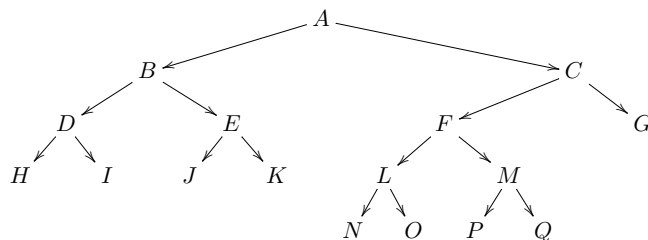


Funktionen auf BinBaum (2)

Knoten in Inorder-Reihenfolge (links, Wurzel, rechts):

```
inorder :: BinBaum t -> [t]
inorder (BinBlatt a) = [a]
inorder (BinKnoten a l r) = (inorder l) ++ a:(inorder r)
```

```
*> inorder beispielBinBaum
"HDIBJEKANLOFPMQCG"
```

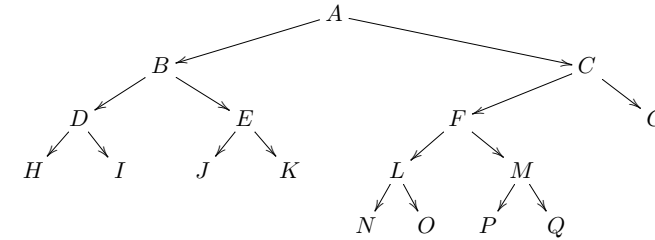


Funktionen auf BinBaum (3)

Knoten in Post-Order Reihenfolge (links, rechts, Wurzel)

```
postorder (BinBlatt a) = [a]
postorder (BinKnoten a l r) =
  (postorder l) ++ (postorder r) ++ [a]
```

```
*> postorder beispielBinBaum
"HIDJKEBNOLPQMFGCA"
```



Funktionen auf BinBaum (2)

Level-Order (Stufenweise, wie Breitensuche)

Schlecht:

```

levelorderSchlecht b =
  concat [nodesAtDepthI i b | i <- [0..depth b]]
where
  nodesAtDepthI 0 (BinBlatt a) = [a]
  nodesAtDepthI i (BinBlatt a) = []
  nodesAtDepthI 0 (BinKnoten a l r) = [a]
  nodesAtDepthI i (BinKnoten a l r) = (nodesAtDepthI (i-1) l)
                                     ++ (nodesAtDepthI (i-1) r)

depth (BinBlatt _) = 0
depth (BinKnoten _ l r) = 1+(max (depth l) (depth r))

```

```

*> levelorderSchlecht beispielBinBaum
"ABCDEFGHJKLMNOPQ"

```

Funktionen auf BinBaum (3)

Level-Order (Stufenweise, wie Breitensuche)

Besser:

```

levelorder b = loForest [b]
where
  loForest xs = map root xs ++ concatMap (loForest . subtrees) xs
  root (BinBlatt a) = a
  root (BinKnoten a _ _) = a
  subtrees (BinBlatt _) = []
  subtrees (BinKnoten _ l r) = [l,r]

```

```

*> levelorder beispielBinBaum
"ABCDEFGHJKLMNOPQ"

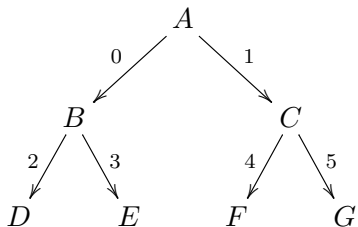
```

Bäume mit Knoten und Kantenmarkierungen

```

data BinBaumMitKM a b =
  BiBlatt a
  | BiKnoten a (b, BinBaumMitKM a b) (b, BinBaumMitKM a b)
  deriving (Eq, Show)

```



```

beispielBiBaum =
  BiKnoten 'A'
    (0, BiKnoten 'B'
      (2, BiBlatt 'D')
      (3, BiBlatt 'E'))
    (1, BiKnoten 'C'
      (4, BiBlatt 'F')
      (5, BiBlatt 'G'))

```

Funktion auf BinBaumMitKM

Map mit 2 Funktionen: auf Blatt- und Knoten-Markierung

```

biMap f g (BiBlatt a) = BiBlatt (f a)
biMap f g (BiKnoten a (kl, links) (kr, rechts)) =
  BiKnoten (f a) (g kl, biMap f g links) (g kr, biMap f g rechts)

```

Beispiel

```

*> biMap toLower even beispielBiBaum
BiKnoten 'a'
  (True, BiKnoten 'b' (True, BiBlatt 'd') (False, BiBlatt 'e'))
  (False, BiKnoten 'c' (True, BiBlatt 'f') (False, BiBlatt 'g'))

```

Anmerkung zum \$-Operator

Definition:

```
f $ x = f x
```

wobei Priorität ganz niedrig, z.B.

```
map (*3) $ filter (>5) $ concat [[1,1],[2,5],[10,11]]
```

wird als

```
map (*3) (filter (>5) (concat [[1,1],[2,5],[10,11]]))
```

geklammert

Syntaxbäume

Auch **Syntaxbäume** sind Bäume

Beispiel: Einfache arithmetische Ausdrücke:

$$\begin{aligned} E &::= (E + E) \mid (E * E) \mid Z \\ Z &::= 0Z' \mid \dots \mid 9Z' \\ Z' &::= \varepsilon \mid Z \end{aligned}$$

Als Haskell-Datentyp (infix-Konstruktoren müssen mit `:` beginnen)

```
data ArEx = ArEx :+: ArEx      data ArEx = Plus ArEx ArEx
          | ArEx **: ArEx      | Mult ArEx ArEx
          | Zahl Int           | Zahl Int
```

Z.B. $(3 + 4) * (5 + (6 + 7))$ als Objekt vom Typ `ArEx`:

```
((Zahl 3) :+: (Zahl 4)) **: ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7)))
```

Syntaxbäume (2)

Interpreter als Funktion in Haskell:

```
interpretArEx :: ArEx -> Int
interpretArEx (Zahl i) = i
interpretArEx (e1 :+: e2) = (interpretArEx e1) + (interpretArEx e2)
interpretArEx (e1 **: e2) = (interpretArEx e1) * (interpretArEx e2)
```

Syntaxbäume: Lambda-Kalkül

Syntax des Lambda-Kalküls als Datentyp:

```
data LExp v =
  Var v          -- x
  | Lambda v (LExp v) -- \v.e
  | App (LExp v) (LExp v) -- (e1 e2)
```

Z.B. $s = (\lambda x.x) (\lambda y.y)$:

```
s :: LExp String
s = App (Lambda "x" (Var "x")) (Lambda "y" (Var "y"))
```

Implementierung der NO-Reduktion

Versuche eine β -Reduktion durchzuführen, dabei frische Variablen mitführen zum Umbenennen

```
tryNOBeta :: (Eq b) => LExp b -> [b] -> Maybe (LExp b, [b])
```

- Einfachster Fall: Beta-Reduktion ist auf Top-Level möglich:

```
tryNOBeta (App (Lambda v e) e2) freshvars =
  let (e',vars) = substitute freshvars e e2 v
  in Just (e',vars)
```

- Andere Anwendungen: gehe links ins Argument (rekursiv):

```
tryNOBeta (App e1 e2) freshvars =
  case tryNOBeta e1 freshvars of
    Nothing -> Nothing
    Just (e1',vars) -> (Just ((App e1' e2), vars))
```

- Andere Fälle: Keine Reduktion möglich:

```
tryNOBeta _ vars = Nothing
```

Implementierung der NO-Reduktion (2)

Implementierung der $\xrightarrow{no,*}$ -Reduktion:

```
reduceNO e = let (e',v') = rename e fresh
              in tryNO e' v'
  where
    fresh = ["x_" ++ show i | i <- [1..]]

tryNO e vars = case tryNOBeta e vars of
  Nothing -> e
  Just (e',vars') -> tryNO e' vars'
```

Implementierung der NO-Reduktion (3)

Hilfsfunktion: Substitution mit Umbenennung:

```
substitute freshvars (Var v) expr2 var
  | v == var = rename (expr2) freshvars
  | otherwise = (Var v,freshvars)

substitute freshvars (App e1 e2) expr2 var =
  let (e1',vars') = substitute freshvars e1 expr2 var
      (e2',vars'') = substitute vars' e2 expr2 var
  in (App e1' e2', vars'')

substitute freshvars (Lambda v e) expr2 var =
  let (e',vars') = substitute freshvars e expr2 var
  in (Lambda v e',vars')
```

Implementierung der NO-Reduktion (4)

Hilfsfunktion: Umbenennung eines Ausdrucks

```
rename expr freshvars = rename_it expr [] freshvars
  where
    rename_it (Var v) renamings freshvars =
      case lookup v renamings of
        Nothing -> (Var v,freshvars)
        Just v' -> (Var v',freshvars)

    rename_it (App e1 e2) renamings freshvars =
      let (e1',vars') = rename_it e1 renamings freshvars
          (e2',vars'') = rename_it e2 renamings vars'
      in (App e1' e2', vars'')

    rename_it (Lambda v e) renamings (f: freshvars) =
      let (e',vars') = rename_it e ((v,f):renamings) freshvars
      in (Lambda f e',vars')
```

Typdefinitionen in Haskell

Drei syntaktische Möglichkeiten in Haskell

- `data`
- `type`
- `newtype`

Verwendung von `data` haben wir bereits ausgiebig gesehen

Typdefinitionen in Haskell (3)

Typdefinition mit `newtype`:

- `newtype` ist sehr ähnlich zu `type`
- Mit `newtype`-definierte Typen dürfen **eigene Klasseninstanz** für Typklassen haben
- Mit `type`-definierte Typen aber nicht.
- Mit `newtype`-definierte Typen haben **einen** neuen Konstruktor
- `case` und `pattern match` für Objekte vom `newtype`-definierten Typ sind immer erfolgreich.

Typdefinitionen in Haskell (2)

`type`; Variante von Typdefinitionen.

Mit `type` definiert man **Typsynonyme**, d.h.:

Neuer Name für bekannten Typ

Beispiele:

```
type IntCharPaar = (Int,Char)
type Studenten = [Student]
type MyList a = [a]
```

Sinn davon: Verständlicher, z.B.

```
alleStudentenMitA :: Studenten -> Studenten
alleStudentenMitA = map nachnameMitA
```

Typdefinitionen in Haskell (4)

Beispiel für `newtype`:

```
newtype Studenten' = St [Student]
```

Diese Definition kann man sich vorstellen als

```
data Studenten' = St [Student]
```

Ist aber nicht semantisch äquivalent dazu, da Terminierungsverhalten anders

Vorteil `newtype` vs. `data`: Der Compiler weiß, dass es nur ein Typsynonym ist und kann optimieren: `case`-Ausdrücke dazu werden eliminiert und durch direkte Zugriffe ersetzt.