Praktische Informatik II: Funktionale Programmierung am Beispiel der Sprache Haskell

Karsten Schmidt

Einordnung

Programmierparadigmen

imperativ
Problemlösung zerlegen in Folge von Zuweisungen ...

prozedural
... strukturiert durch verschachtelte Prozeduren
Fortran, Pascal, C, Modula, Ada

objektorientiert
... strukturiert als Methoden von Objekten in Klassenhierarchien
Java, C++

deklarativ
Problem zerlegen in ...

logisch
... Menge von Relationen, logisch verknüpft
Prolog

funktional
... Menge von Funktionen, per Komposition verknüpft
Lisp, Haskell

modellbasiert
Problemlösung approximieren mit verschiedenen Modellen
UML

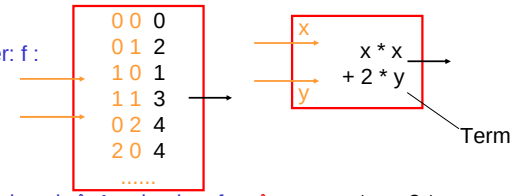
agentenorientiert
Problemlösung delelgieren an autonom handelnde Akteure

und viele mehr

Was ist eigentlich eine Funktion

Beispiel in geläufiger Notation: f(x,y) = $x^2 + 2y$

genauer: f :



Schreibweise als λ-Ausdruck: f = λ x , y . x * x + 2 * y

lies: "f ist diejenige Funktion, die den Argumenten x und y den Wert x * x + 2 * y zuordnet".

... f kann nun selbst in Termen verwendet werden

Funktionsdefinitionen in Haskell

```
f = \x y -> x * x + 2 * y
g x y = x * x + 2 * y
h = \x -> f x x
```

Merke
- -Variablen in imperativen Sprachen sind Symbole für Speicherzustände
- -Variablen in funktionalen Sprachen sind Symbole für Funktionsargumente
- -Eine Konstante kann als 0-stellige Funktion angesehen werden
- -Haskell-Programm = Folge von Funktionsdefinitionen

Haskell-Programme benutzen

1. Haskell-Interpreter besorgen, z.B. Hugs, `www.hugs.org`
2. Textdatei `bla.hs` mit Funktionsdefinitionen erstellen
3. Ruf des Interpreters, z.B. `hugs bla.hs`
4. Funktionen benutzen z.B. `3 + 4 - f(7,8)`
5. Weitereditieren vom Interpreter aus `:edit` bzw. `:e`

Arbeit des Interpreters = Terme ausrechnen

```
f = \x y -> x * x + 2 * y
```

```
> f (3 + 4) 8
```

1. Argumente berechnen
   ```
   f 7 8
   ```
2. Funktionssymbol durch Funktionsdefinition ersetzen
   ```
   (\x y -> x * x + 2 * y) 7 8
   ```
3. Argumentsymbole durch Argumente ersetzen
   ```
   7 * 7 + 2 * 8
   ```
4. Term auswerten
   ```
   65
   ```

... wenn im Term oder den Argumenten weitere Funktionssymbole vorkommen, das Ganze rekursiv...

Ausdrücke/Terme

= Beschreiben von Funktionen

1. Fallunterscheidung

-in funktionaler Sicht: keine Anweisungen, insbesondere keine bedingten Anweisungen, Schleifen

→ Fallunterscheidung ist (vordefinierte) Funktion:

ifthenelse(B,T,E) = { T, falls B ; E, falls nicht B

```
max = \a b -> if a >= b then a else b
```

7

Bewachte Ausdrücke

= Menge von Paaren (Bedingung, Term)
... evaluiert zu demjenigen Term, bei dem Bedingung erfüllt

```
max a b c
  |   a > b && a > c  = a
  |   b > c           = b
  |   otherwise       = c
```

Guard

8

Tupel

Argumente/Rückgaben dürfen auch Tupel sein

```
h a b c = (a + b, a + c, b + c)

f = \(a,b) -> (b,a)

g (a,b) = a + b
```

9

Benannte Teilausdrücke

... für Zwischenergebnisse

```
flaeche x y z = sqrt(s*(s-x)*(s-y)*(s-z))
            where
            s = let a = (x + y + z)
                in a/2

quadratsumme a b = ( sq a ) + ( sq b )
           where sq x = x * x
```

10

Scope von where und let

Where: ganze Gleichung

```
wurzeln p q
       | a*a > q          = [a+b,a-b]
       | a*a==q           = [a]
       | otherwise        = []
    where a = -p/2.0
          b = sqrt(a * a - q)
```

Let: Einzelner Ausdruck

```
wurzeln p q
  | (-p/2.0)^2 > q  = let a = -p/2.0
                          b = sqrt(a*a-q)
                      in [a+b,a-b]
  | (-p/2.0)^2 ==q  = let a = -p/2.0 in [a]
  | otherwise  = []
```

11

Rekursion

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

binom n k
     | k == 0      = 1
     | k == n      = 1
     | otherwise   = binom (n-1) k
                   + binom (n-1) (k-1)
```

12

Musterbasierte Programmierung

... ein bisschen Prolog in Haskell

```
fak 0 = 1
fak n = n * fak (n-1)

fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

... mehr dazu später

Listen

```
[1,2,3,4]
"blablabla"
[]
[1 .. 5]
[1,3 ..17]
[1 ..]
1 : [2,3,4]
[c*d | c<-[1,2,3], d<-[2,3,4], c/=d]
[1,2,3] ++ [4,5]
[[1,2,3],[4],[],[3,4,5]]

> reverse [1,2,3,4]
   [4,3,2,1]
> reverse "blablabla"
   "albalbalb"
```

Listenelemente müssen gleichartig sein!

Arbeit mit Listen

```
qs [] = []
qs (u:us) = qs ls ++ [u] ++ qs rs
            where ls = [x | x <- us, x <= u]
                  rs = [x | x <- us, x >  u]


primes   = sieve [2 ..]
sieve [] = []
sieve (n:ns) = n : (sieve [x | x <- ns,
                           mod x n /= 0])

sublists [] = [[]]
sublists (l:ls) = let ss = sublists ls in
                  ss ++ [l : xs | xs <-ss]
```


Typen

Typ einer Funktion = Beschreibung von Definitions- und Wertbereich

```
h = \x -> x * x
```

```
> :type h
    h :: Integer -> Integer
```

Haskell kann Typ einer Funktion erschließen,
Typ kann aber auch im Programm angegeben werden:

```
h :: Integer -> Integer
h x = x * x
```

→ Lesbarkeit, Fehlerfrüherkennung

Typen

```
f  = \(x,y) → (y,x)
```

```
:t f
   f :: (a,b) -> (b,a)

:t (:)
```

a,b: „beliebige Typen“

erwartet:

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

tatsächlich erscheint aber:

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

?

Currying

Sei `f = \x y -> 2 * x + 3 * y`,
also f : ℤ x ℤ → ℤ
Was bleibt, wenn Argument x fest gesetzt wird, z.B. x = 13?

```
f` = \y -> 26 + 3 * y
```

→ Für alle y: f'(y) = f(13,y).

Für x = 27 entsteht andere Funktion

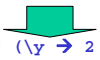
```
f`` = \y → 54 + 3 * y
```

f''(y) = f(27,y)

```
Also:
f = \x → (\y → 2 * x + 3 * y)
```

Currying

```
f = \x y -> 2 * x + 3 * y,
```



```
f = \x → (\y → 2 * x + 3 * y)
```

Argumente als einzeln übergeben gedacht heißt: Jedes Einsetzen EINES Arguments transformiert n-stellige Funktion in (n-1)-stellige Funktion.
→ f : A1 x A2 x ... X An → B ist äquivalent zu
→ f: A1 → (A2 → (.... → (An → B))...) Curried Form
Typen werden in Haskell in curried Form angegeben, Operator -> ist rechtsassoziativ

```
:t f
f :: Integer -> Integer -> Integer
```


Currying

Die „Zwischenfunktionen“ sind explizit verfügbar:

```
dreiminus = (-) 3
```

```
> 3 - 1
2
> (-) 3 1
2
> dreiminus 1
2
```


Sections

Für Operatoren kann man das auch mit dem anderen Argument:

```
aaa = (17-)
bbb = (-17)
```

```
> aaa 3
14
> bbb -3
-20
```


Funktionen als Argument

```
qs lt [] = []
qs lt (h:t) = qs lt l ++ [h] ++ (qs lt r)
          where l = [x | x <- t, lt x h]
                r = [x | x <- t, not (lt x h)]


comparepairs (a,b) (c,d) =
       a < c || (a == c && b < d)

qspairs::[(Integer,Integer)]->[(Integer,Integer)]
qspairs = qs comparepairs
```

```
> :t qs
qs :: (a -> a -> Bool) -> [a] -> [a]
```


Funktionen als Resultat

```
comp :: (b->c)->(a->b)->(a->c)

comp f g = \x -> f (g x)

const3 = \x -> 3

it 0 f = \x -> x
it n f = f . (it (n-1) f)
```

comp auch als Standardoperator . verfügbar

Arrays in Haskell

... sind Funktionen von (Indextyp) in (Komponententyp)

```
-- Beispiel: Array of Strings

read :: (Integer->String)->Integer->String
read f x = f x

write::(Integer->String)->Integer->String->
                        (Integer->String)
write f x y = \z -> if z == x then y else f z
```

Bem.: Es gibt eine Array-Bibliothek für Haskell
→ effizient

Funktionale auf Listen

1. Map: wendet eine Funktion f elementweise auf eine Liste an

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

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

In Haskell in die Listenschreibweise integriert:

```
[f x | x <- list]

> [x * x | x <- [1,2,3,4,5]]
[1,4,9,16,25]
```


Funktionale auf Listen

2. Zip: wendet eine Funktion f elementweise auf mehrere Listen an

```
zip::(a->b->c)->[a]->[b]->[c]

zip f [] [] = []
zip f (x:xs) (y:ys) = (f x y):(zip f xs ys)
```

Vordefiniert: zipWith

```
>>> zipWith (+) [1,2,3]   [4,5,6]
[5,7,9]
```


Funktionale auf Listen

3. Filter: Gibt Liste derjenigen Elemente zurück, die Bedingung erfüllen

```
filter::(a->Bool)->[a]->[a]
filter b [] = []
filter b (x:xs) = (if b x then [x]
                          else [])++(filter b xs)
```

Haskell: In Listenschreibweise integriert

```
> [x | x <- [1,2,3,4,5] , x > 2]
[3,4,5]
```


Funktionale auf Listen

4. Reduce: Rechnet Operationskette auf Liste aus (mit oder ohne Startwert)

```
reduce::(a->b->b)->b->[a]->b
reduce1::(a->a->a)->[a]->a

reduce f i [] = i
reduce f i (x:xs) = f x (reduce f i xs)
reduce1 f [x] = x
reduce1 f (x:xs) = f x (reduce1 f i xs)
```

Haskell: vordefinierte foldr,foldl,foldl1 und foldr1

```
> foldr1 (+) [1,2,3,4,5]
15
> foldr (:) [7,8] [1,2,3,4,5]
[1,2,3,4,5,7,8]
```


Ein Beispiel:

Summe aller ungeraden Zahlen bis n

```
oddsum n = foldr1 (+) [2*x+1 | x<-[0 .. n],
                              2*x+1<=n]
```

Reduce Map Filter

Termauswertungsstrategien

Nach der beschriebenen Auswertungsstrategie dürften

```
elem  7  [1..]
fak 3
if 1 == 1 then 7 else bot
```

```
bot = bot
```

eigentlich nicht terminieren.

Tun sie aber. Haskell nutzt *lazy evaluation*.

Lazy evaluation

= Argumente werden erst ausgewertet, wenn sie gebraucht werden (und nur, soweit sie gebraucht werden)

```
if B then T else E
```

-B wird immer ausgewertet
-Nur einer von T und E wird ausgewertet

```
elem 7 [1 .. ]
```

- `[n .. ]` ist eigentlich Funktion

```
enumFrom::Integer->[Integer]
enumFrom n = n : enumFrom (n+1)
```

-rufende `(elem)` und gerufene `(enumFrom)` koexistieren
-rufende fordert nach Bedarf weitere Teilergebnisse ab
-verwandtes Konzept: Pipes in UNIX

Lazy vs. Strict

Strict = konsequent erst Argumente ausrechnen, dann Funktion abarbeiten

f heißt strikt, wenn: Falls ein Argument undefiniert ist, so ist Funktionswert undefiniert
Bsp: + strikt ifthenelse nicht strikt * nicht strikt
&& nicht strikt gdw strikt

Lazy/Strict Evaluation liefern unterschiedliche Semantik auf nicht strikten Funktionen
Strict: terminiert nie, falls ein Argument undefiniert (error, oder Auswertung terminiert nicht)
Lazy: terminiert manchmal doch.

Lazy vs. Strict

Vorteil strict: konsistent mit Mathematik
(Definitionen mit lazy evaluation sind reihenfolgeabhängig:)

```
take n [] = []
take 0 xs  = []
take n (x:xs) = x:(take (n-1) xs)

take1 0 xs = []
take1 n [] = []
take1 n (x:xs) = x:take1 (n-1) xs)
```

```
> take 0 bot   > take bot []   > take1 0 bot   > take1 bot []
(n.term)       []              []               (n.term)
```


Lazy vs. strict

Vor- oder Nachteil lazy/strict:
lazy vermeidet nicht benötigte Berechnungen, braucht aber overhead wegen der Koexistenz rufender mit gerufener Funktion

Vorteile lazy:
- ermöglicht „unendliche" Daten [1 ..]
- ermöglicht Rekursion
  → ifthenelse wird in allen funktionalen Sprachen lazy evaluiert

Haskell bietet Mittel, die Evaluationsstrategie zu beeinflussen

Datenstrukturen

Haben:
- Standard-Datentypen, u.a.
  - Bool (True, False)
  - Int (wie int in Java)
  - Integer (beliebige Größe)
  - Float/Double (Gleitkommazahlen)
  - Rational (rationale Zahlen beliebiger Genauigkeit)
  - Char
  - String (= [Char])

Datenstrukturen

haben:
- Listen
- Funktionen
- Arrays (= Funktionen)
- Tupel

fehlen:
- Aufzählungstypen (Farbe = rot, gelb, grün,...)
- Bäume
- .... → `data`

Daten und Konstruktoren

```
data Point = Point Float Float

data Tree = Nil
          | Node Int Tree Tree

data Datum = Datum Int String Int

birthday :: Datum
birthday =  Datum 21 "Dezember" 2004

data Switch = On | Off
```

Datentyp

Konstruktor

Typ kann in Typdeklarationen verwendet werden
Konstruktoren dienen zum Anlegen von Daten des Typs

Konstruktoren

Konstruktoren sind Funktionen

```
data Point = Point Float Float
data Tree = Nil
          | Node Int Tree Tree
data Datum = Datum Int String Int
data Switch = On | Off
```

```
> :t Point
Point :: Float -> Float -> Point
> :t Nil
Nil :: Tree
> :t Node
Node :: Tree -> Tree -> Tree
> :t Off
Off :: Switch
```


Zugriff auf Komponenten

durch musterbasierte Programmierung

```
data Datum = Datum Int String Int

month :: Datum -> String
month (Datum t m j) = m

insert :: Tree -> x -> Tree
insert Nil x = Node x Nil Nil
insert (Node y l r) x
       | x == y   = Node y l r
       | x <  y   = Node y (insert l x) r
       | x >  y   = Node y l (insert r x)
```


Beispiel: Geometrie

```
data Point = Point Float Float
dist (Point x1 y1) (Point x2 y2) =
      sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))

data Figure = Line Point Point
            | Triangle Point Point Point
            | Circle Point Float
flaeche :: Figure -> Float
flaeche (Line x y) = 0.0
flaeche (Triangle x y z)=sqrt(s*(s-a)*(s-b)*(s-c))
      where
      a = dist x y
      b = dist x z
      c = dist y z
      s = (a + b + c) / 2
flaeche (Circle x r) = pi * r * r
```


Polymorphe Datentypen

= Datentypen, die einen Typ als Parameter haben
Beispiel: Listen vom Typ a

können selbst definiert werden

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

→

Datenstrukturen: eigene vs. vordefinierte

Vordefinierte Strukturen könnten auch selbst definiert werden (*) = Pseudocode

```
(*)data Char = ... | ´a´ | ´b´ | ... | ´{´ | ...
(*)data Int = -65536 | ... | 0 | 1 | ... | 65535
(*)data Integer = ... | -1 | 0 | 1 | ...
data List a = Nil | Front a (List a)
data Pair a b = Pair a b
```

Also: ´a´ , 0 , [] sind Konstruktoren für einen Datentyp

```
> :t ´a´
´a´ :: Char
>:t []
[] :: [a]
```

Typsynonyme

... definieren keine neuen Typen, aber neue Namen für existierende Typen

```
type Name = String
type String = [Char]   -- vordefiniert

type Telefonliste = [(Name,Integer)]
```


Mehr zu Musterbasierung

Muster = beliebig verschachtelter Term aus Konstruktoren und (nur je einmal vorkommenden) Variablen

Konstruktoren: vordefinierte Typen

```
:    ( , )   0   1121 ´a´
```

oder selbstdefiniert

```
(Node t1 t2)   On   Off   Nil    (Datum x y z)
```


Mehr zu Musterbasierung

Muster können erfolgreich, erfolglos oder divergent sein
erfolgreich → Muster passt
erfolglos → Muster passt nicht
divergent → Musterprüfung terminiert nicht

Muster werden top-down links-rechts geprüft.
Erfolglose Muster werden übergangen.
Divergente Muster führen zu Nichtterminierung des Funktionsrufes
Erfolgreiche Muster führen zu Abarbeitung der zugehörigen Gleichung

Mehr zu Musterbasierung

Komplexe Teilmuster können benannt und in der Gleichung verwendet werden, z.B. statt

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

```
f s@(x:xs) = x:s
```

Wildcards bezeichnen ungenutzte Teilmuster (wie Prolog)

```
month (Datum _ m _) = m
```


Mehr zu Musterbasierung

Muster können *irrefutable* (niemals erfolglos)
z.B. `x  (x , y )`
oder *refutable* (möglicherweise erfolglos) sein
z.B. `[]  0  (x , 0)`

Muster können mit ~ als „lazy“ vereinbart werden
→ immer irrefutable, gematchte Werte werden „on demand“ berechnet, falls Muster nicht passt, Laufzeitfehler.

Fragebögen zur Vorlesung

- bitte auch Rückseite beachten
- bitte Ü-Leiter angeben
- verbale Kommentare erwünscht
- bitte am Ende der Vorlesung zurückgeben

Mehr zu Polymorphie

polymorpher Typ = Menge einfacher Typen mit gemeinsamen Operationen

z.B. [a] = {[Char],[Int],[(Char,Integer)],[[Char]],.....}

gemeinsam: `: [] head tail elem`

Vorteile: etabliert in Mathematik
(+ für Zahlen, Matrizen, ...)
Code-Wiederverwendung

Mehr zu Polymorphie

Haben:

- Polymorphie durch Nichtverwendung einschränkender Operationen

z.B. `> :t f`

```
f = \(x,y) -> (y,x)
f :: (a,b) ->(b,a)
```

- explizite Polymorphie

z.B. `data Tree a`

Typklassen

Manchmal sind gewisse Einschränkungen der Polymorphie nötig
z.B. Sortieren verlangt, dass `<` definiert ist

Lösung: Typklassen

Typklasse = Menge von Typen, für die ein vorgegebener Satz an Funktionen definiert ist

Java-Pendant: Interface

Typklassen

Vordefinierte Typklassen (Auswahl)

Beispiele:
Eq `== /=`
Ord `== /= > < min max <= >=`
Enum `..`
Show
Num
....

```
> :t div
div :: Integral a =>  a -> a -> a
```


Zuordnen eigener Typen zu einer Typklasse

```
instance Eq Point where
  (Point x y) == (Point z t)  = x == z && y == t
```

`/=` generisch (`not ==`) oder selbst definiert

```
instance Eq Point where
  (Point x y) == (Point z t)  = x == z && y == t
  (Point x y) /= (Point z t) = if x /= y
                then True else z /= t
```

```
instance (Eq a) => Eq (Tree a) where
  Nil         == Nil         = True
  (Node a b) == (Node c d) = (a==c) && (b==d)
  _           == _           = False
```

Beispiel

```
mitglied:: Eq t => t -> [t] -> Bool
mitglied x [] = False
mitglied x (y:ys) = (x ==  y) || mitglied x ys
```

Wie sieht eine Typklasse aus?

```
class Eq t where
     (==) :: t -> t -> Bool
     (/=) :: t -> t -> Bool
    a /= b  = not   (a == b)  -- default
```


Wie sieht eine Typklasse aus?

```
class Eq t => Ord t where
   (<), (>) ,(<=),(>=) :: t -> t -> Bool
   min, max:: t -> t -> t
   a <= b  = (a < b) || (a == b)
   a > b = b < a
   a >= b  = b <= a
   min a b = if a <= b then a else b
   max a b = if a <= b then b else a
```


Wenn man zu faul zum implementieren ist...

```
data Tree t = Nil
            | Node t (Tree t) (Tree t)
            deriving (Eq, Ord, Show)
```

→default-Implementationen für die vorgeschriebenen Operationen

Bem.: Typ kann mehreren Klassen angehören

Kann man Typklassen definieren?

Man kann.

```
class Myclass t where
     bla :: t -> Bool
        blub :: t -> t
     blabla :: t -> Bool
        blabla x = bla (blub x)

instance Myclass Point where
     bla (Point x y)  =  x == y
        blub (Point x y) = Point (x+1.0) (y+1.0)
```


Modularisierung

Modul
= Eine Struktureinheit im Programm, kapselt eine Menge von Funktionsdefinitionen

Modulersteller

-nur einige der Funktionen für Nutzer freigegeben
-Implementation änderbar

Modulbenutzer

-braucht oft nur einige Funktionen
-mag evt. andere Namen
-will vielleicht importierte Funktionen weitergeben

Modulkonzept in Haskell 1

```
module mymodule1 where
f = ...
g = ...
```

Default: alles exportiert

```
module mymodule2 where
import mymodule1
h = ...
```

Importiert f und g

```
module mymodule3 (f , ...) where
f = ...
g = ...
```

Exportiert nur f, ...

```
module mymodule4 (f , module bla) where
import bla
f = ...
g = ...
```

exportiert f und re-exportiert alles aus bla

Modulkonzept in Haskell 2

`import mymodule1 (f,...)` Importiert nur f,...

`import mymodule1 hiding (f,...)`
Importiert alles ausser f,...

`import mymodule1 renaming (f to fff, g to ggg)`
Import mit Umbenennung

Abschließendes Beispiel

Implementation eines eigenen Datentyps
naürliche Zahlen beliebiger Genauigkeit

Grundlage: Liste von Ziffern
Arithmetik wie in der Schule („Schriftliches Rechnen“)
d.h.: Dezimalsystem

Start: Ziffern

```
data Digit = Null | Eins | Zwei | Drei | Vier |
             Fuenf | Sechs | Sieben | Acht | Neun

suc::Digit->Digit
suc Null = Eins
suc Eins = Zwei
...
suc Acht = Neun
suc Neun = Null

prd :: Digit -> Digit
prd x = (suc . suc . suc . suc . suc .
              suc . suc . suc . suc) x
```


Ziffern aus Zeichen, in Typklassen einordnen

```
toDigit :: Char -> Digit
toDigit '0' = Null
toDigit '1' = Eins
...
toDigit '8' = Acht
toDigit '9' = Neun

instance Eq Digit where
      Null == Null = True
      Null == x = False
      x == y  = (suc x) == (suc y)

instance Ord Digit where
      Null <= x   = True
      x <= Null   = False
      x <= y    = (prd x) <= (prd y)
```


Datentyp Longnat, Erzeugen aus Integern

```
data Longnat = Longnat [Digit]

integer2Longnat :: Integer -> Longnat
integer2Longnat x
    | x < 0   = error "cannot conv. neg. Int. to Longnat"
    | x == 0  = Longnat []
    | x == 1  = Longnat [Eins]
    | x == 2  = Longnat [Zwei]
    ...
    | x == 8  = Longnat [Acht]
    | x == 9  = Longnat [Neun]
    | otherwise  =
          let (Longnat a) = integer2Longnat (div x 10)
              (Longnat b) = integer2Longnat (mod x 10)
           in Longnat ((if b==[] then [Null] else b) ++ a
      -- Einerstelle zuerst!
```


Erzeugen aus Strings

```
stringtoLongnat :: String -> Longnat
stringtoLongnat s = Longnat ((removeLeadingZeros . reverse)
                             [toDigit x | x <- s])
     -- Einerstelle zuerst!

removeLeadingZeros :: [Digit] -> [Digit]
removeLeadingZeros [] = []
removeLeadingZeros [Null] = []
removeLeadingZeros [x] = [x]
removeLeadingZeros (x:xs) =
    let r = removeLeadingZeros xs
     in
       if r == [] then removeLeadingZeros [x] else (x:r)
```

Vergleichen, Einordnen in Typklassen

```
compar :: [Digit] -> [Digit] -> Int -- -1 kleiner
                                    -- 0 gleich
                                    -- 1 groesser
compar [] [] = 0
compar [] l  = -1
compar l []  = 1
compar (x:xs) (y:ys)
          | res /= 0     = res
          | x < y        = -1
          | x == y       = 0
          | x > y        = 1
     where res = compar xs ys

instance Eq Longnat where
   (Longnat a) == (Longnat b) = compar a b == 0
instance Ord Longnat where
   (Longnat a) <= (Longnat b) = compar a b <= 0
```

Addition

```
addTable :: Digit -> Digit -> Digit
addTable Null x = x
addTable x y = suc (addTable (prd x) y)

carrya::Digit->Digit->Bool
carrya x y = let s = addTable x y in s < x || s < y

add::[Digit]->[Digit]->[Digit]
add [] x = x
add x [] = x
add (x:xs) (y:ys)  = (addTable x y) :
         ((if carrya x y then addCarry else add) xs ys)

addCarry::[Digit]->[Digit]->[Digit]
addCarry [] x = add [Eins] x
addCarry x [] = add x [Eins]
addCarry (Neun:xs) (y:ys) = y : (addCarry xs ys)
addCarry (x:xs) ys = add ((suc x):xs) ys
```

Subtraktion

```
subTable :: Digit -> Digit -> Digit
subTable x Null = x
subTable x y = prd (subTable x (prd y))

carrys::Digit->Digit->Bool
carrys x y = x < y

sub::[Digit]->[Digit]->[Digit]
sub x [] = x
sub [] x = error "result of subtraction out of range"
sub (x:xs) (y:ys) = (subTable x y) :
       ((if carrys x y then subCarry else sub) xs ys)

subCarry [] x = error "result of subtraction out of range"
subCarry x [] = sub x [Eins]
subCarry (x:xs) (Neun:ys) = (x : subCarry xs ys)
subCarry x (y:ys) = sub x ((suc y):ys)
```

Multiplikation

```
multTable :: Digit -> Digit -> [Digit]
multTable Null x = []
multTable x Null = []
multTable x y = add (multTable (prd x) y) [y]

shift::[Digit] -> [Digit]
shift l = Null:l

mult::[Digit]->[Digit]->[Digit]
mult [] x = []
mult x [] = []
mult (x:xs) (y:ys) = add (multTable x y)
                         (add (shift (mult [x] ys))
                           ( add (shift (mult xs [y]))
                              ((shift.shift)(mult xs ys))))
         -- (10a+b)(10c+d)=100ac+10ad+10bc+bd
```

Einordnen in Typklasse Num

```
instance Num Longnat where
 (Longnat a) + (Longnat b) = (Longnat . removeLeadingZeros
 (Longnat a) - (Longnat b) = (Longnat . removeLeadingZeros
 (Longnat a) * (Longnat b) = (Longnat . removeLeadingZeros
 negate x = if x == (Longnat []) then (Longnat []) else
                  error "cannot negate positive Longnat"
```

Einordnen in Typklasse Show

```
toChar :: Digit -> Char
toChar Null = '0'
toChar Eins = '1'
...
toChar Acht = '8'
toChar Neun = '9'

show1:: Longnat -> String
show1 (Longnat []) = "0"
show1 (Longnat l) = foldr (:) [] [toChar x| x<-l]

instance Show Longnat where
      show x = (reverse . show1) x
```

Zusammenfassung

Kern funktionaler Sprachen
- Funktionen, λ-Abstraktion, Currying
- Statt Kontrollstrukturen spezielle Funktionen, Rekursion
- Variablen symbolisieren Funktionsargumente
- keine Seiteneffekte
- Rechnen = Terme auswerten

Zusammenfassung

Gemeinsamkeiten mit anderen Paradigmen (Auswahl)
Typklassen ≈ Java-Interfaces
Polymorphie

Musterbasierung ⊆ Prolog

nicht behandelt: Ein- und Ausgabe

Am Mi keine Vorlesung mehr!