

Teil I

Funktionale Programmierung

Einführung: Funktionale Programmierung in Haskell

Haskell Brooks Curry



Quicksort in Haskell:

- Funktionen höherer Ordnung
- anonyme Funktionen
- Pattern Matching

Pivotwahl: erstes Element der Liste

```
qsort []      = []  
qsort (p:ps) =      (qsort (filter (\x -> x<=p) ps))  
                ++ p:(qsort (filter (\x -> x> p) ps))
```

Quicksort in Haskell:

- **Funktionen höherer Ordnung**
- anonyme Funktionen
- Pattern Matching

Pivotwahl: erstes Element der Liste

```
qsort []      = []  
qsort (p:ps) =      (qsort (filter (\x -> x<=p) ps))  
                  ++ p:(qsort (filter (\x -> x> p) ps))
```

Quicksort in Haskell:

- Funktionen höherer Ordnung
- **anonyme Funktionen**
- Pattern Matching

Pivotwahl: erstes Element der Liste

```
qsort []      = []  
qsort (p:ps) = (qsort (filter (\x -> x<=p) ps))  
              ++ p:(qsort (filter (\x -> x> p) ps))
```

Quicksort in Haskell:

- Funktionen höherer Ordnung
- **anonyme Funktionen, Unterversorgung**
- Pattern Matching

Pivotwahl: erstes Element der Liste

```
qsort []      = []  
qsort (p:ps) =      (qsort (filter ( <= p) ps))  
                ++ p:(qsort (filter ( >  p) ps))
```

Quicksort in Haskell:

- Funktionen höherer Ordnung
- anonyme Funktionen
- **Pattern Matching**

Pivotwahl: erstes Element der Liste

```
qsort [] = []  
qsort (p:ps) = (qsort (filter (\x -> x<=p) ps))  
              ++ p:(qsort (filter (\x -> x> p) ps))
```


Quicksort in Haskell:

■ List-Comprehension

Pivotwahl: erstes Element der Liste

```
qsort []      = []  
qsort (p:ps) =      (qsort [x | x <- ps, x <= p])  
                ++ p: (qsort [x | x <- ps, x >  p])
```

Quicksort in Haskell:

- Funktionen höherer Ordnung
- anonyme Funktionen
- Pattern Matching
- **Typinferenz, Polymorphismus**

Pivotwahl: erstes Element der Liste

```
qsort :: (Ord t) => [t] -> [t]
qsort []      = []
qsort (p:ps) =      (qsort (filter (\x -> x<=p) ps))
                  ++ p:(qsort (filter (\x -> x> p) ps))
```

Funktionale Programme sind

- kompakt
- frei von Seiteneffekten
- unabhängig von Anweisungsreihenfolge

⇒ leichter verständlich

⇒ weniger Fehler

Funktionale Programmierung

- befreit den Programmierer vom “Wie?”
- erlaubt die Konzentration auf das “Was?”

Der Begriff Funktion in Sprachen wie Haskell entspricht der mathematischen Sicht:

- Eine Funktion f bildet Elemente aus ihrem Definitionsbereich auf Elemente ihres Wertebereichs ab
- Auswertungen von $f(x)$ haben keinen Effekt auf Daten des Programms, sie liefern lediglich Ergebniswert
- Der Wert $f(x)$ hängt allein von x ab

Hingegen in Java: bei Aufruf $y = f(x)$

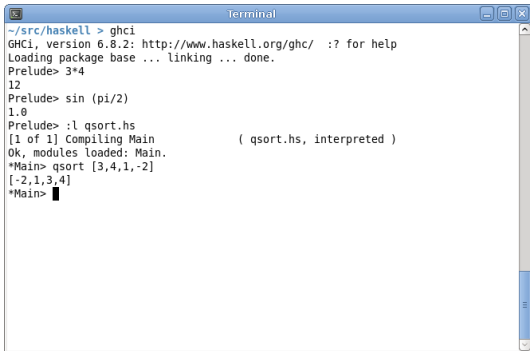
- Seiteneffekte möglich
- y kann abhängen von statischen Variablen, Benutzereingaben, . . .

Haskell:

- rein funktionale Sprache
- Haskell 1.0: 1990, Haskell '98, Haskell 2010

Haskell Platform:

- Interpreter ghci, Compiler, Libraries
- <http://www.haskell.org/platform/>



```
~/src/haskell > ghci
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude> 3*4
12
Prelude> sin (pi/2)
1.0
Prelude> :l qsort.hs
[1 of 1] Compiling Main                ( qsort.hs, interpreted )
Ok, modules loaded: Main.
*Main> qsort [3,4,1,-2]
[-2,1,3,4]
*Main> █
```

Ein Haskell Programm ist eine Folge von Funktionsdefinitionen

■ Einfache Gleichungen:

```
f x = sin x / x
```

```
g x = x * (f (x*x))
```

$$f(x) = \frac{\sin(x)}{x}$$

$$g(x) = x \cdot f(x^2)$$

■ Rekursive Gleichungen:

```
binom n k =  
  if (k==0) || (k==n)  
  then 1  
  else binom (n-1) (k-1)  
        + binom (n-1) k
```

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

Fallunterscheidung: Funktion **if** _ **then** _ **else** _

Ein Haskell Programm ist eine Folge von Funktionsdefinitionen

■ Einfache Gleichungen:

```
f x = sin x / x
```

```
g x = x * (f (x*x))
```

$$f(x) = \frac{\sin(x)}{x}$$

$$g(x) = x \cdot f(x^2)$$

■ Rekursive Gleichungen:

```
binom n k
```

```
| (k==0 || k==n) = 1
```

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

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

Fallunterscheidung: alternative Schreibweise

Programmausführung: Auswertung

- beliebiger Ausdrücke in Umgebung wie `ghci`
- des Ausdrucks `main` in kompilierten Programmen

Programm: `simple.hs`

```
square x = x * x
```

```
cube    x = x * square x
```

```
main = putStr "Hallo_Welt!"
```

Laden und Ausführen in `ghci`:

```
Prelude> :l simple.hs
```

```
[1 of 1] Compiling Main
```

```
( simple.hs, interpreted )
```

```
Ok, modules loaded: Main.
```

```
*Main> cube (1+2)
```

```
27
```

```
*Main>
```


Haskell-Ausdrücke werten schrittweise aus. Wir schreiben

- $e1 \Rightarrow e2$ für einen Auswertungsschritt
- $e1 \Rightarrow^+ e_n$ falls $e1 \Rightarrow e2 \Rightarrow \dots \Rightarrow e_n$

Programm:

```
square x = x * x  
cube    x = x * square x
```

Auswertung:

```
cube (1+2)  ⇒ (1+2) * square (1+2)  
            ⇒    3 * square    3  
            ⇒    3 * (3*3)  
            ⇒    3 *    9  
            ⇒ 27
```

Rekursion

Auswertung rekursiver Funktionen:

- Zwischenausdrücke können mit Eingabegröße wachsen

⇒ Speicherverbrauch wächst

Programm:

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

Auswertung:

```
fak 3 ⇒+ 3 * (fak 2)
      ⇒+ 3 * (2 * (fak 1))
      ⇒+ 3 * (2 * (1 * (fak 0)))
      ⇒+ 3 * (2 * (1 * 1))
      ⇒+ 3 * (2 * 1)
      ⇒+ 3 * 2
      ⇒+ 6
```

rekursive Aufrufe: $\mathcal{O}(n)$

Speicherverbrauch: $\mathcal{O}(n)$

Variante mit Akkumulation

- Übergebe Zwischenergebnisse in Hilfsparameter `acc`

Programm:

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

```
fak n = fakAcc n 1
```

Auswertung:

```
fak   3  ⇒+ fakAkk   3  1  
        ⇒+ fakAkk   2  3  
        ⇒+ fakAkk   1  6  
        ⇒+ fakAkk   0  6  
        ⇒+ 6
```

rekursive Aufrufe: $\mathcal{O}(n)$
Speicherverbrauch: $\mathcal{O}(1)$ ¹

¹Bei aktivierten Compiler-Optimierungen.

Endrekursion (tail recursion)

Linearität: Eine Funktion heißt linear rekursiv, wenn in jedem Definitionszweig nur ein rekursiver Aufruf vorkommt.

Endrekursion: Eine linear rekursive Funktion heißt endrekursiv (tail recursive), wenn in jedem Zweig der rekursive Aufruf nicht in andere Aufrufe eingebettet ist.

- Linear, aber nicht endrekursiv:

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

- Endrekursiv:

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

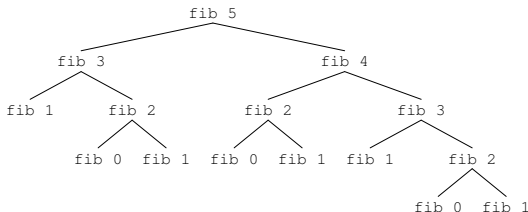
- Endrekursion ermöglicht speicher-effiziente Auswertung

Auswertung rekursiver Funktionen

Programm: Fibonacci-Zahlen 0, 1, 1, 2, 3, 5, 8, ...

```
fib n
| (n == 0) = 0
| (n == 1) = 1
| otherwise = fib (n - 1) + fib (n - 2)
```

Rekursionstyp: nichtlinear, fib erzeugt Aufrufbaum



rekursive Aufrufe: $\mathcal{O}(2^n)$

Programm: Fibonacci-Zahlen 0, 1, 1, 2, 3, 5, 8, ... mit Akkumulatoren

```
fibAkk n n1 n2
| (n == 0)   = n1
| (n == 1)   = n2
| otherwise = fibAkk (n - 1) n2 (n1 + n2)

fib n = fibAkk n 0 1
```

Auswertung:

```
fib 5  $\Rightarrow^+$  fibAkk 5 0 1
       $\Rightarrow^+$  fibAkk 4 1 1
       $\Rightarrow^+$  fibAkk 3 1 2
       $\Rightarrow^+$  fibAkk 2 2 3
       $\Rightarrow^+$  fibAkk 1 3 5
       $\Rightarrow^+$  5
```

rekursive Aufrufe: $\mathcal{O}(n)$

Endrekursion + Akkumulator \simeq while + lokale Variablen

Nichttermination:

```
fibAkk n n1 n2
| (n == 0)  = n1
| (n == 1)  = n2
| otherwise = fibAkk (n-1) n2 (n1+n2)
fib n = fibAkk n 0 1
```

```
fib (-1)
⇒ fibAkk (-1) 0 1
⇒ fibAkk (-2) 1 1
⇒ fibAkk (-3) 1 2
⇒ ...
```

```
f x = f x
```

```
f 5 ⇒ f 5 ⇒ f 5 ⇒ ...
```

Laufzeitfehler:

```
f x = div 1 x
```

```
f 0 ⇒ div 1 0
```

```
*** Exception: divide by zero
```

```
g x = if (x == 1)
      then (error "Eins")
      else x
```

```
g 1 ⇒+ (error "Eins")
```

```
*** Exception: Eins
```


Nichttermination:

<pre>fibAkk n n1 n2 (n == 0) = n1 (n == 1) = n2 otherwise = fibAkk (n-1) n2 (n1+n2) fib n = fibAkk n 0 1</pre>	<pre>fib (-1) ⇒ fibAkk (-1) 0 1 ⇒ fibAkk (-2) 1 1 ⇒ fibAkk (-3) 1 2 ⇒ ⊥</pre>
--	---

<pre>f x = f x</pre>	<pre>f 5 ⇒ f 5 ⇒ f 5 ⇒ ⊥</pre>
----------------------	--------------------------------

Laufzeitfehler:

<pre>f x = div 1 x</pre>	<pre>f 0 ⇒ div 1 0 ⇒ ⊥ *** Exception: divide by zero</pre>
<pre>g x = if (x == 1) then (error "Eins") else x</pre>	<pre>g 1 ⇒⁺ (error "Eins") ⇒ ⊥ *** Exception: Eins</pre>

- In beiden Fällen: Aufruf wertet aus zu \perp ("bottom")

Listen

Listen

Ein Liste ist entweder

- ❶ die leere Liste `[]`, oder
- ❷ eine Liste `(x:xs)`, konstruiert aus Restliste `xs` und Listenkopf `x`

Die Funktion `(:)` ist der Listenkonstruktor (`cons`)

<code>5:(10:(12:(9:(6:[]))))</code>	<code>head [1,2,3]</code>	\Rightarrow^+	<code>1</code>
\equiv <code>5:10:12:9:6:[]</code>	<code>tail [1,2,3]</code>	\Rightarrow^+	<code>[2,3]</code>
\equiv <code>[5,10,12,9,6]</code>	<code>tail [3]</code>	\Rightarrow^+	<code>[]</code>
	<code>null [1,2,3]</code>	\Rightarrow^+	<code>False</code>

Elementare Listenfunktionen:

- `head` und `tail` berechnen Kopf und Restliste nichtleerer Listen
- `null` prüft, ob Liste leer

Definition von Listenfunktionen

- oft Fallunterscheidung per `null l`

Rekursive Listenfunktion: Listenlänge

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

```
length [1,2,3]  $\Rightarrow^+$  1 + (length (tail [1,2,3]))  
                 $\Rightarrow^+$  1 + (1 + (length (tail [2,3])))  
                 $\Rightarrow^+$  1 + (1 + (1 + (length (tail [3]))))  
                 $\Rightarrow^+$  1 + (1 + (1 + 0))  
                 $\Rightarrow^+$  3
```

Definition von Listenfunktionen

- oft Fallunterscheidung per `null l`
- auch kompliziertere Strukturen

Rekursive Listenfunktion: Maximales Element

```
maximum l = if (null l) then error "empty"
            else if (null (tail l)) then head l
            else max (head l) (maximum (tail l))
```

```
maximum [1,3,2] ⇒+ max (head [1,3,2]) (maximum (tail [1,3,2]))
                ⇒+ max 1 (max (head [3,2]) (maximum (tail [3,2])))
                ⇒+ max 1 (max 3 2)
                ⇒+ max 1 3
                ⇒+ 3
```

- Definition unübersichtlich. Abhilfe: Pattern Matching

Pattern Matching: Mehrere Gleichungen zur Definition einer Funktion

- Jede Gleichung gilt für Argumente von speziellem Struktur-Muster
- Überlappende Muster: Erste Gleichung wird angewandt
- Muster sind: Konstanten, Variablen

Zahlen-Muster:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Auswertung: $\text{fib } 2 \Rightarrow \text{fib } (2-1) + \text{fib } (2-2)$

- 2 passt weder auf Muster 0 noch auf Muster 1
- Variablen-Muster n passt immer

Pattern Matching: Mehrere Gleichungen zur Definition einer Funktion

- Jede Gleichung gilt für Argumente von speziellem Struktur-Muster
- Überlappende Muster: Erste Gleichung wird angewandt
- Muster sind: Konstanten, Variablen oder **Konstruktoren**

Listen-Muster:

```
maximum []      = error "empty"  
maximum (x:[]) = x  
maximum (x:xs) = max x (maximum xs)
```

Auswertung: `maximum [1,3,2]` \Rightarrow **max** 1 (maximum [3,2])

- `[1,3,2]` passt weder auf Muster `[]`, noch auf Muster `x:[]`
- `[1,3,2]` passt auf `(x:xs)` mit `x=1` und `xs=[3,2]`

- Zeichenketten sind Listen von Buchstaben
- "Text" Kurzschreibweise für ['T','e','x','t']

⇒ Anwendbar in Mustern
Zuordnung Wochentag → Zahl

```
dayOfWeek "Montag"    = 1  
dayOfWeek "Dienstag"  = 2  
...
```

⇒ Anwendbar als Argument von Listenfunktionen:
Ist y Element einer Liste?

```
isIn []      y = False  
isIn (x:xs) y = if (x == y) then True else (isIn xs y)
```

Ist c eine Zahlzeichen?

```
isDigit c = isIn "123456789" c
```


Imperativ: Aneinanderfügen oder Umdrehen modifiziert Listen

Funktional: Neue Listen werden erstellt

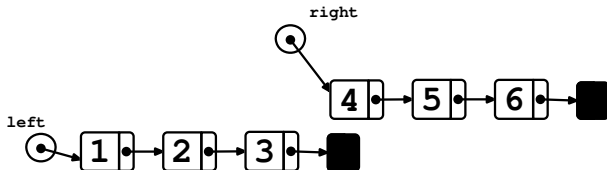
- `app left right` Elemente aus `left` gefolgt von Elementen aus `right`
- `rev list` Elemente aus `list` in umgedrehter Reihenfolge

```
app [] r = r
app (x:xs) r = x:(app xs r)
```

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

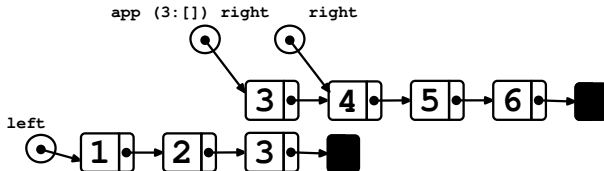
```
app []      r  = r  
app (x:xs) r = x:(app xs r)
```

```
left  = [1,2,3]  
right = [4,5,6]
```



- Struktur von `left` nochmals aufbauen, `right` wiederverwenden!
- Aufwand: $\mathcal{O}(\text{length left})$

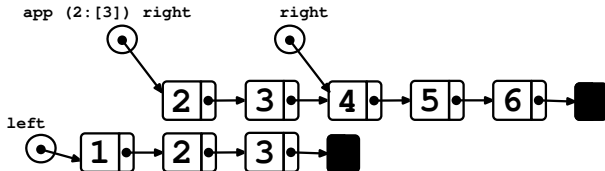
```
app []      r  = r           left  = [1,2,3]
app (x:xs) r = x:(app xs r)  right = [4,5,6]
```



- Struktur von `left` nochmals aufbauen, `right` wiederverwenden!
- Aufwand: $\mathcal{O}(\text{length left})$

```
app []      r  = r  
app (x:xs) r = x:(app xs r)
```

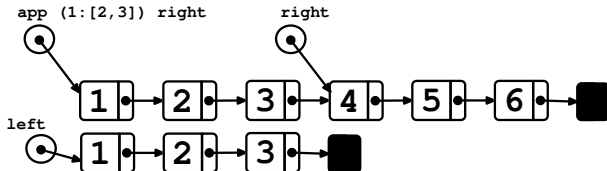
```
left  = [1,2,3]  
right = [4,5,6]
```



- Struktur von `left` nochmals aufbauen, `right` wiederverwenden!
- Aufwand: $\mathcal{O}(\text{length left})$

```
app []      r = r  
app (x:xs) r = x:(app xs r)
```

```
left  = [1,2,3]  
right = [4,5,6]
```



- Struktur von `left` nochmals aufbauen, `right` wiederverwenden!
- Aufwand: $\mathcal{O}(\text{length left})$

Listenumkehrung mittels `app`

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

Auswertung:

$\text{rev } [1,2,3] \Rightarrow^+ \text{app } (\text{app } (\text{app } [] [3]) [2]) [1] \Rightarrow^+ [3,2,1]$

Aufwand ($n = \text{length } l$):

- n Auswertungen von `app`
Listen der Länge $0 \dots n-1$ in linker Position
- Insgesamt $\mathcal{O}\left(\sum_{i=0}^{n-1} i\right) = \mathcal{O}(n^2)$

Effiziente Listenumkehrung mit Listen-Akkumulator

```
rev list = revAcc [] list
  where revAcc acc []      = acc
        revAcc acc (x:xs) = revAcc (x:acc) xs
```

Auswertung:

```
rev [1,2,3] ⇒ revAcc []      [1,2,3]
             ⇒ revAcc [1]    [2,3]
             ⇒ revAcc [2,1]   [3]
             ⇒ revAcc [3,2,1] []
             ⇒ [3,2,1]
```

Aufwand: $\mathcal{O}(\text{length } l)$

<code>a ++ b</code>	Infixnotation für <code>app a b</code>	<code>[1] ++ [2,3] \Rightarrow^+ [1,2,3]</code>
<code>head l</code>	Erstes Element von <code>l</code>	<code>head [1,2,3] \Rightarrow^+ 1</code> <code>head [] $\Rightarrow \perp$</code>
<code>tail l</code>	Restliste von <code>l</code>	<code>tail [1,2,3] \Rightarrow^+ [2,3]</code> <code>tail [] $\Rightarrow \perp$</code>
<code>take n l</code>	Erste <code>n</code> Elemente von <code>l</code>	<code>take 2 [1,2,3] \Rightarrow^+ [1,2]</code> <code>take 5 [1,2,3] \Rightarrow^+ [1,2,3]</code>
<code>drop n l</code>	<code>l</code> ohne erste <code>n</code> Elemente	<code>drop 2 [1,2,3] \Rightarrow^+ [3]</code> <code>drop 5 [1,2,3] \Rightarrow^+ []</code>