Deskriptive Programmierung

Mehr zur Syntax von Funktionsdefinitionen

Definierende Gleichungen: Grundregeln

- Auf der linken Seite einer definierenden Gleichung in Haskell dürfen
  - keine noch auszuwertenden Ausdrücke, sondern …
  - nur Variablen und Konstanten (sowie Pattern, siehe später …)

  vorkommen:

```
f x (2 * y) = x * y
```

unzulässig!

```
f x 1 = x * 2
```

okay

- Auf der rechten Seite einer definierenden Gleichung dürfen
  - beliebige Ausdrücke, (natürlich) auch auszuwertende, aber . . .
  - nur Variablen von der linken Seite (also keine „frischen“ Variablen)

  vorkommen:

```
f x = x * y
```

unzulässig!

```
f x 1 = x * 2
```

okay

Definierende Gleichungen: Grundregeln

- In der Liste von formalen Parametern einer Funktionsdefinition darf jede Variable nur genau einmal vorkommen:

```
f n 0 n = n^2
```

unzulässig!

stattdessen:

```
f n 0 m | n == m = n^2
```

Funktionsdefinitionen: Fallunterscheidung (1)

Komplexere Funktionsdefinitionen sind aus mehreren Alternativen zusammengesetzt. Jede der Alternativen definiert einen Fall der Funktion:

$$n\,! \;=\; \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)\,! & \text{falls } n > 0 \end{cases}$$

In Haskell, schon gesehen:

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

In Haskell kann auch der obere Stil imitiert werden, allerdings stehen die Bedingungen vor dem Gleichheitszeichen:

```
fac n     | n == 0    =   1
          | n >  0    =   n * fac (n - 1)
```

Funktionsdefinitionen: Fallunterscheidung (2)

$$n\,! \;=\; \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)\,! & \text{falls } n > 0 \end{cases}$$

```
fac  n    |  n == 0    =  1
          |  n >  0    =  n * fac (n - 1)
```

„Wächter“
(engl.: „guards“)

Boolesche Ausdrücke

Wie in der mathematischen Notation werden die „Wächter“ beim Auswerten von oben nach unten durchlaufen, bis zum ersten Mal eine Bedingung erfüllt ist.

Dieser Fall wird dann zum Reduzieren herangezogen.

fac 3

```
fac  n      |  n == 0      =  1
            |  n  >  0     =  n * fac (n - 1)
```

3 * fac (3 - 1)

Die Fakultätsfunktion ist nur partiell definiert: für negativen Inputparameter wird kein „passender“ Fall gefunden, so dass das Resultat undefiniert ist.

fac (–3)

```
fac  n      |  n == 0      =  1
            |  n  >  0     =  n * fac (n - 1)
```

undefiniert

Funktionsdefinitionen: Fallunterscheidung (4)

Überführung in eine total definierte Funktion durch Anfügen eines „catch all“-Falls mit der Pseudo-Bedingung otherwise:

fac (–3)

```
fac n    | n == 0       = 1
         | n > 0        = n * fac (n - 1)
         | otherwise    = 0
```

0

Mitunter hilfreich auch zur Abkürzung:

```
zwischen x y z    | (x <= y) && (y <= z)     = True
                  | (y < x) || (y > z)       = False
```

```
zwischen x y z    | (x <= y) && (y <= z)     = True
                  | otherwise                = False
```

Funktionsdefinitionen: Fallunterscheidung (5)

Variationen:

```
fac n      | n == 0   =  1
           | n > 0    =  n * fac (n - 1)
```

ist „eigentlich“ nur eine Abkürzung für die Notation

```
fac n      | n == 0   =  1
fac n      | n > 0    =  n * fac (n - 1)
```

Noch eine Notationsvariante, in der die erste Bedingung durch einen Konstantenparameter ausgedrückt wird:

```
fac 0                 =  1
fac n      | n > 0    =  n * fac (n - 1)
```

Funktionsdefinitionen: Fallunterscheidung (6)

- offenbar wichtige Grundtechnik:
  Auswahl eines „passenden“ Definitionsfalls für eine auszuwertende Funktionsapplikation

- zwei Auswahlkriterien (in dieser Reihenfolge!):
  - „pattern matching“: dt. ≈ Mustervergleich
  - Auswertung der „Wächterbedingung“

```
(1)   ack 0 n  | n >= 0          = n + 1
(2)   ack m 0  | m > 0           = ack (m - 1) 1
(3)   ack m n  | n > 0 && m > 0  = ack (m - 1) (ack m (n - 1))
```

Ackermann-Funktion

ack 0 0	passt zu (1)
ack 2 0	passt zu (2)
ack 2 1	passt zu (3)

Reihenfolge bei der Abarbeitung von Fällen in Funktionsdefinitionen

- Bei der Auswertung der Applikation ack 0 0 würden alle drei linken Seiten „matchen“!

```
ack  0   n     | n >= 0            =  n + 1
ack  m   0     | m > 0             =  ack  (m - 1)  1
ack  m   n     | n  > 0  &&  m > 0 =  ack  (m - 1)  (ack  m  (n - 1))
```

- Der definierende Fall ist der (von oben nach unten durchlaufen) erste matchende Fall, dessen Wächter erfüllt ist.

- Auf diese Weise ist sichergestellt, dass es immer einen eindeutigen Funktionswert gibt. (… wenn es überhaupt einen gibt!)

- Bei der Ackermann-Funktion liefert jede Reihenfolge der drei Gleichungen dieselbe Funktion. Das ist aber nicht immer so! fac 0 verhält sich hier verschieden:

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

1

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

undefiniert

Deskriptive Programmierung

Pattern Matching

Pattern Matching: Prinzip

konkrete Applikation

`> ack 0 (ack 2 1)`

pattern matching

linke Seite einer Definition

`ack 0 n | ... = ...`

Regeln des Pattern Matching:

- Voraussetzung: identischer Funktionsname
- Konstanten „matchen"
  - sich selbst (z.B.: 1 ↔ 1)
  - jede Variable (z.B.: 1 ↔ n)
- komplexe Ausdrücke matchen
  - jede Variable (z.B.: (fib 3) ↔ x)
  - diejenige Konstante, die ihren Funktionswert bezeichnet (z.B.: (fib 4) ↔ 5)

erzwingt Auswertung!

- Tupel matchen
  - jede Variable, und Tupel gleicher Länge bei komponentenweisem Match (z.B.: (1, False, fib 4) ↔ (1, x, 5))
- …

Auswirkung von Pattern-Matching-„Strategien“

- Beispiele auf Booleschen Werten:

```
not False = True
not True  = False
```

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- etwas kompakter:

```
not False = True
not _     = False
```

```
True && True = True
_    && _    = False
```

anonyme Variablen

- aber effizienter? ja, für manche Eingaben sehr drastisch!

```
False && (ack 4 2 > 0)
```

Auswirkung von Pattern-Matching-„Strategien“

- Beispiele auf Booleschen Werten:

```
not False = True
not True  = False
```

```
True   &&  True   = True
True   &&  False  = False
False  &&  True   = False
False  &&  False  = False
```

- etwas kompakter:

```
not False = True
not _     = False
```

```
True  &&  True  = True
_     &&  _     = False
```

- aber effizienter? ja, für manche Eingaben!

andere Variante:

```
b  &&  True  = b
_  &&  _     = False
```

Matching von links nach rechts!

- nicht möglich:

```
b  &&  b  = b
_  &&  _  = False
```

Alternative Syntax (und Scoping!)

- Sogenannte case-Ausdrücke, zum Beispiel:

```
ifThenElse i t e  =  case i of
                       True  → t
                       False → e
```

- Oder, zum Beispiel:

```
f x y  =  case (x + y, x - y) of
            (z, _)  | z > 0 → y
            (0, x)          → x + y
```

- Was ergibt sich dann wohl aus folgendem Aufruf dieser Funktion?

```
> f  10 (-10)
```

Deskriptive Programmierung

Elementares Arbeiten mit Listen

Damit Pattern Matching so richtig interessant wird: Verarbeitung von Listen

- Haskell-Liste: Folge von Elementen gleichen Typs (homogene Struktur)

- Syntax: Listenelemente in eckige Klammern eingeschlossen.

[1, 2, 3]	Liste von ganzen Zahlen (Typ: Int)
['a', 'b', 'c']	Liste von Buchstaben (Typ: Char)
[ ]	leere Liste (beliebigen Typs)
[[1,2], [ ], [2]]	Liste von Int-Listen

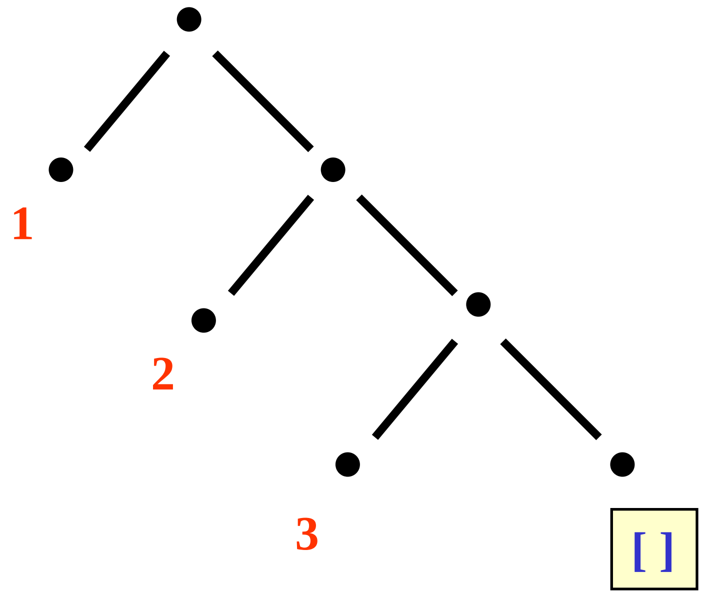
[[1,2], 'a', 3]	<u>keine</u> gültige Liste (verschiedene Elementtypen)

- Im Gegensatz zu dem, was viele Beispiele in der Vorlesung suggerieren mögen, sind Listen in der Praxis oft <u>nicht</u> die Datenstruktur, die man verwenden sollte! (Stattdessen nutzerdefinierte Datentypen, oder Typen aus Bibliotheken wie Data.ByteString, Data.Array, Data.Map, …)

Baumdarstellung von Listen

Listen werden intern als bestimmte Binärbäume dargestellt, deren Blätter mit den einzelnen Listenelementen markiert sind:

[1, 2, 3]



(ausgesprochen: „Nil“)

Jede (endliche) Liste wird durch eine leere Liste abgeschlossen.

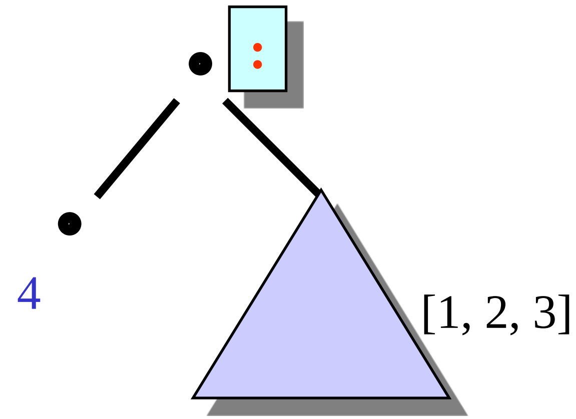
Listenkonstruktor

- Elementarer Konstruktor („Operator“ zum Konstruieren) für Listen:

  : (ausgesprochen: „Cons“)

- Der Konstruktor „**:**“ dient zum Erweitern einer gegebenen Liste um ein Element, das am Listenkopf eingefügt wird:

```
>  4 : [1, 2, 3]
[4, 1, 2, 3]
```



- Alternative Notation für Listen (analog zur Baumdarstellung):

```
4 : 1 : 2 : 3 : [ ]
```

Länge einer Liste

- Funktion zur Bestimmung der Länge einer Liste (vordefiniert):

```
length [ ]        =  0
length (x : xs)   =  length xs  + 1
```

Kopfelement

Restliste

- Beispiel für die Anwendung von length:

```
> length [1, 2]
= length [2]  + 1
= (length [ ]  + 1)  + 1
= (      0         + 1)  + 1
= 1 + 1
= 2
```

Pattern Matching mit Listenkonstruktoren

- Pattern Matching zwischen Listen und Konstruktorausdrücken ist nur zu verstehen, wenn man beide Ausdrücke in Konstruktorform sieht:

length (x : xs) = . . .

length [1, 2] → length (1 : 2 : [ ])

x $\mapsto$ 1
xs $\mapsto$ 2 : [ ]

- Auch beim „rekursiven Abbauen“ von einelementigen Listen ist diese Sichtweise hilfreich:

length (x : xs) = . . .

length [2] → length (2 : [ ])

x $\mapsto$ 2
xs $\mapsto$ [ ]

Alle Klammern auf dieser Folie zwingend!

Konkatenation von Listen

- wichtige Grundoperation für alle Listen: Konkatenieren zweier Listen
  (= Aneinanderhängen)

```
concatenation  [ ]       ys    =    ys
concatenation  (x : xs)  ys    =    x : (concatenation xs ys)
```

- Beispielanwendung:

```
>  concatenation [1, 2]  [3, 4]
[1, 2, 3, 4]
```

- Als Infixoperator vordefiniert:

```
> [1, 2] ++ [3, 4]
[1, 2, 3, 4]
```

Zugriff auf einzelne Listenelemente und Teillisten

- gezielter Zugriff auf einzelne Elemente einer Liste durch weiteren vordefinierten Infixoperator:

  !!

- Zählung der Listenelemente beginnt mit 0 !

```
>  [1, 2, 3] !! 1
2
```

- Zugriff per (x : xs)-Pattern natürlich nur auf nichtleere Listen:

```
tail (x : xs)  =  xs
```

```
>  tail [ ]
ERROR - Pattern match failure: tail []
```

```
head (x : xs)  =  x
```

```
>  head [ ]
ERROR - Pattern match failure: head []
```

(Leider ist der Fehlerursprung in solchen Fällen nicht immer so einfach identifizierbar.)

```
f :: [Int] → [[Int]]
f [ ]               =  [ ]
f [x]               =  [[x]]
f (x : y : zs)      =  if x <= y then (x : s) : ts else [x] : s : ts
    where s : ts = f (y : zs)
```

lokale Definition + Match

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]              =  [ ]
f [x]              =  [[x]]
f (x : y : zs)     =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
    where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = [2] : s' : ts'
                                                         where s' : ts' = f (0 : [ ])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]                = [ ]
f [x]                = [[x]]
f (x : y : zs)       = if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts      where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = [2] : s' : ts'
                                                           where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                               where s' : ts' = f (0 : [ ])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts     where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = [2] : s' : ts'
                                                          where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                              where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                              where s' : ts' = [[0]]
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]              =  [ ]
f [x]              =  [[x]]
f (x : y : zs)     =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts       where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = [2] : s' : ts'
                                                            where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                                where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                                where s' : ts' = [[0]]
= (1 : [2]) : [0] : [ ] = [[1, 2], [0]]
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]              =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]              =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
```

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]            =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]             =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
= ([1, 3], [2, 4])
```

Zwischendurch bemerkt: Layout in Haskell

```
let  y = a * b
     f x = (x + y) / y
in  f c + f d
```

implizites Layout
(„offside rule“)

```
let  { y = a * b; f x = (x + y) / y }
in  f c + f d
```

äquivalent, explizites Layout

```
let  y = a * b
            f x = (x + y) / y
in  f c + f d
```

nicht äquivalent,
inkorrekt

```
let   y = a * b
  f x = (x + y) / y
in  f c + f d
```

(analog für andere Sprachkonstrukte, z.B. where, case)

Pattern Matching über mehreren Argumenten (und „veraltete“ (n + k)-Pattern)

```
drop :: Int → [Int] → [Int]
drop 0       xs       =  xs
drop n       [ ]      =  [ ]
drop (n + 1) (x : xs) =  drop n xs
```

in Haskell 98 erlaubt, in Haskell 2010 nicht mehr!

```
> drop 0 [1, 2, 3]
[1, 2, 3]
```

```
> drop 5 [1, 2, 3]
[ ]
```

```
> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```