Rückblick: Eines der Themen vom letzten Mal waren...

...selbstdefinierte (neue) Datentypen in Haskell!

$\leadsto$ Haskells Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
  - Spezialfälle
    - Produkttypen
    - Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen ($\leadsto$ "unendliche" Datenstrukturen)
- Polymorphe Typen ($\leadsto$ Wiederverwendung)

Offen geblieben war die Untersuchung und Diskussion *polymorpher Typen*!

Heutiges Thema...

Polymorphie

- Bedeutung lt. Duden:
  - *Vielgestaltigkeit*, *Verschiedengestaltigkeit*

    ...mit speziellen fachspezifischen Bedeutungsausprägungen
    - In der *Chemie*: das Vorkommen mancher Mineralien in verschiedener Form, mit verschiedenen Eigenschaften, aber gleicher chemischer Zusammensetzung
    - In der *Biologie*: Vielgestaltigkeit der Blätter oder der Blüte einer Pflanze
    - In der *Sprachwissenschaft*: das Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. die verschiedenartigen Pluralbildungen in: die Wiese*n*, die Feld*er*, die Tier*e*)
    - In der *Informatik*, speziell der *Theorie der Programmiersprachen*: $\leadsto$ unser heutiges Thema!

Polymorphie

...im programmiersprachlichen Kontext unterscheiden wir insbesondere zwischen

- Polymorphie
  - auf Funktionen
    - Parametrische Polymorphie ("Echte Polymorphie")
    - Ad hoc Polymorphie (synonym: Überladung)
      $\leadsto$ Haskell-spezifisch: Typklassen
  - auf Datentypen

Polymorphie

Wir beginnen mit...

- (Parametrischer) Polymorphie auf Funktionen

  ...die wir an einigen Beispielen schon kennengelernt haben:
  - Die Funktionale `curry` und `uncurry`
  - Die Funktionen `length`, `head` und `tail`

Rückblick (auf Vorlesungsteil 4)

Die Funktionale `curry` und `uncurry`...

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

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y
```


Rückblick (auf Vorlesungsteil 1)

Die Funktionen `length`, `head` und `tail`...

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

head :: [a] -> a
head (x:_) = x

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


Äußeres Kennzeichen parametrischer Polymorphie

Statt

- (ausschließlich) konkreter Typen (wie `Int`, `Bool`, `Char`,...)

treten in der (Typ-) Signatur der Funktionen

- (auch) *Typparameter*, sog. *Typvariablen*

auf.

Beispiele:

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

length :: [a] -> Int
```


Typvariablen in Haskell

Typvariablen in Haskell sind...

- freigewählte Identifikatoren, die mit einem Kleinbuchstaben beginnen müssen

  z.B.: `a`, `b`, aber auch `fp185161_WS0506`

Im Unterschied dazu sind Typnamen, (Typ-) Konstruktoren in Haskell...

- freigewählte Identifikatoren, die mit einem Großbuchstaben beginnen müssen

  z.B.: `A`, `String`, `Node`, aber auch `Fp185161_WS0506`

Warum Polymorphie auf Funktionen?

...Wiederverwendung (durch Abstraktion)!

$\leadsto$ *ein typisches Vorgehen in der Informatik*!

Einfaches Bsp.: Funktionsabstraktion

Sind viele Ausdrücke der Art

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

zu berechnen, schreibe eine Funktion

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)
```

um die Rechenvorschrift `(a * b + c) * (b + a * c)` wiederverwenden zu können:

```
f 5 37 13
f 15 7 12
f 25 3 10
...
```


Zur Motivation param. Polymorphie (1)

Listen können Elemente sehr unterschiedlicher Typen zusammenfassen, z.B.

- Listen von Basistypelementen
  `[2,4,23,2,53,4] :: [Int]`
- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [[Int]]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3),(1.12,2.22)] :: [Point]`
- Listen von Bäumen
  `[Nil,Node 42 Nil Nil), Node 17 (Node 4 Nil Nil) Nil)]`
- ...
- Listen von Funktionen
  `[fact, fib, fun91] :: [Integer -> Integer]`

Zur Motivation param. Polymorphie (2)

- *Aufgabe*: Bestimme die Länge einer Liste, d.h. die Anzahl ihrer Elemente.
- *Naive Lösung*: Schreibe für jeden Typ eine entsprechende Funktion.

Zur Motivation param. Polymorphie (4)

Umsetzung der naiven Lösung:

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

lengthIntLstLst :: [[Int]] -> Int
lengthIntLstLst []     = 0
lengthIntLstLst (_:xs) = 1 + lengthIntLstLst xs

lengthPointLst :: [Point] -> Int
lengthPointLst []     = 0
lengthPointLst (_:xs) = 1 + lengthPointLst xs

lengthTreeLst :: [BinTree1] -> Int
lengthTreeLst []     = 0
lengthTreeLst (_:xs) = 1 + lengthTreeLst xs

lengthFunLst :: [Integer -> Integer] -> Int
lengthFunLst []     = 0
lengthFunLst (_:xs) = 1 + lengthFunLst xs
```


Zur Motivation param. Polymorphie (5)

Damit möglich:

```
lengthIntLst    [2,4,23,2,53,4] => 6
lengthIntLstLst [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
lengthPointLst  [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
lengthTreeLst   [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil)] => 3
lengthFunLst    [fact, fib, fun91] => 3
```


Zur Motivation param. Polymorphie (6)

Beobachtung:

- Die einzelnen Rechenvorschriften zur Längenberechnung sind i.w. identisch
- Unterschiede beschränken sich auf
  - Funktionsnamen und
  - Typsignaturen

Zur Motivation param. Polymorphie (7)

Sprachen, die parametrische Polymorphie offerieren, erlauben eine elegantere Lösung unserer Aufgabe:

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

length [2,4,23,2,53,4] => 6
length [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
length [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil)] => 3
length [fact, fib, fun91] => 3
```

Funktionale Sprachen, auch Haskell, offerieren parametrische Polymorphie!

Zur Motivation param. Polymorphie (8)

Unmittelbare Vorteile parametrischer Polymorphie:

- Wiederverwendung von
  - Rechenvorschriften und
  - Funktionsnamen (*Gute Namen sind knapp!*)

Monomorphie vs. Polymorphie

Rechenvorschriften der Form

- `length :: [a] -> Int` heißen *polymorph*.

Rechenvorschriften der Form

- `lengthIntLst :: [Int] -> Int`
- `lengthIntLstLst :: [[Int]] -> Int`
- `lengthPointLst :: [Point] -> Int`
- `lengthFunLst :: [Integer -> Integer] -> Int`
- `lengthTreeLst :: [BinTree1] -> Int` heißen *monomorph*.

Sprechweisen im Zshg. mit parametrischer Polymorphie

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

Sprechweisen:

- `a` in der Typsignatur von `length` heißt *Typvariable*. Typvariablen werden gewöhnlich mit Kleinbuchstaben vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...
- Typen der Form...

  ```
  length :: [Point] -> Int
  length :: [[Int]] -> Int
  length :: [Integer -> Integer] -> Int
  ...
  ```

  heißen *Instanzen* des Typs `[a] -> Int`. Letzterer heißt *allgemeinster Typ* der Funktion `length`.

  Bem.: Das Hugs-Kommando `:t expr` liefert stets den (eindeutig bestimmten) *allgemeinsten* Typ eines (wohlgeformten) Haskell-Ausdrucks `expr`.

Weitere Bsp. polymorpher Funktionen

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

id 3 => 3
id ["abc","def"] => ["abc","def"]


zip :: [a] -> [b] -> [(a,b)]      -- "Verschmelzen" von Listen
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = []

zip [3,4,5] ['a','b','c','d'] => [(3,'a'),(4,'b'),(5,'c')]
zip ["abc","def","geh"] [(3,4),(5,4)] => [("abc",(3,4)),("def",(5,4))]


unzip :: [(a,b)] -> ([a],[b])     -- "Entzerren" von Listen
unzip []  = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                   where
                   (xs,ys) = unzip ps

unzip [(3,'a'),(4,'b'),(5,'c')] => ([3,4,5],['a','b','c'])
unzip [("abc",(3,4)),("def",(5,4))] => (["abc","def"],[(3,4),(5,4)])
```


Weitere in Haskell auf Listen vordefinierte (polymorphe) Funktionen

`:`	`:: a->[a]->[a]`	Listenkonstruktor (rechtsassoziativ)
`!!`	`:: [a]->Int->a`	Proj. auf i-te Komp., Infixop.
`length`	`:: [a]->Int`	Länge der Liste
`++`	`:: [a]->[a]->[a]`	Konkatenation zweier Listen
`concat`	`:: [[a]]->[a]`	Konkatenation mehrerer Listen
`head`	`:: [a]->a`	Listenkopf
`last`	`:: [a]->a`	Listen"endelement"
`tail`	`:: [a]->[a]`	Liste ohne Listenkopf
`init`	`:: [a]->[a]`	Liste ohne Listenendelement
`splitAt`	`:: Int->[a]->[[a],[a]]`	Aufspalten einer Liste an Stelle i
`reverse`	`:: [a]->[a]`	Umdrehen einer Liste

...

Polymorphie

Soviel zu parametrischer Polymorphie auf Funktionen...

Wir fahren fort mit

- Polymorphie auf Datentypen
  - Algebraische Datentypen
  - Typsynonymen

Polymorphe algebraische Typen (1)

Der Schlüssel dazu...

$\rightsquigarrow$ Definitionen algebraischer Typen dürfen Typvariablen enthalten und werden dadurch polymorph.

Beispiele: Paare und Bäume...

```
data Pairs a = Pair a a

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


Polymorphe algebraische Typen (2)

Beispiele konkreter Werte von Paaren und Bäumen:

```
data Pairs a = Pair a a

Pair 17 4    :: Pairs Int
Pair [] [42] :: Pairs [Int]
Pair [] []   :: Pairs [a]


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

Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil)       :: Tree Char
Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)     :: Tree Float
Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil) :: Tree [Char]
```

Polymorphe algebraische Typen (3)

Ähnlich wie parametrische Polymorphie unterstützt auch...

- Polymorphie auf algebraischen Datentypen

Wiederververwendung!

Vergleiche dies mit der schon bekannten Situation im Zshg. mit *polymorphen Listen*:

```
length :: [a] -> Int
```


Polymorphe Typen (4)

Ähnlich wie bei der Funktion *Länge* auf Listen...

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

...kann auf algebraischen Typen Typunabhängigkeit generell vorteilhaft ausgenutzt werden, wie hier das Bsp. der Funktion `depth` auf Bäumen zeigt:

```
depth :: Tree a -> Int
depth Nil = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

depth (Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil))       => 2
depth (Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil))     => 2
depth (Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil)) => 2
```


Heterogene algebraische Typen

...sind möglich, z.B. *heterogene* Bäume:

```
data HTree = LeafS String           |
             LeafI Int              |
             NodeF Float HTree HTree |
             NodeB Bool  HTree HTree

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ HTree
depth :: HTree -> Int
depth (LeafS _) = 1
depth (LeafI _) = 1
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: HTree -> Int
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _               = 1
```

Beachte: ...folgendes geht nicht $\leadsto$ *Syntaxfehler*!

```
depth :: HTree -> Int
depth (_ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _           = 1
```

Heterogene polymorphe algeb. Typen

...sind ebenfalls möglich, z.B. *heterogene* polymorphe Bäume:

```
data PHTree a b c d = LeafA a                                      |
                      LeafB b                                      |
                      NodeC c   (PHTree a b c d) (PHTree a b c d) |
                      NodeD d c (PHTree a b c d) (PHTree a b c d)

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ PHTree
depth :: (PHTree a b c d) -> Int
depth (LeafA _)         = 1
depth (LeafB _)         = 1
depth (NodeC _ t1 t2)   = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: (PHTree a b c d) -> Int
depth (NodeC _ t1 t2)   = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _                 = 1
```

Das heißt...

- Auch *"mehrfach" polymorphe* Datenstrukturen sind möglich!

Polymorphe Typsynonyme

...auch *polymorphe Typsynonyme* und *Funktionen* darauf sind möglich:

Beispiel:

```
type List a = [a]

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

Oder kürzer:

```
lengthList :: List a -> Int
lengthList = length
```

...abstützen auf Standardfunktion `length` möglich, da `List a` Typsynonym, kein neuer Typ.

Beachte: `type List = [a]` ist nicht möglich ($\leadsto$ *Typfehler!*)

Zusammenfassung und erste Schlussfolgerungen

Zu...

- Polymorphen Funktionen,
- Polymorphen Datentypen und
- Vorteilen, die aus ihrer Verwendung resultieren.

Polymorphe Typen

...ein (Daten-) Typ `T` heißt *polymorph*, wenn bei der Deklaration von `T` der Grundtyp oder die Grundtypen der Elemente (in Form einer oder mehrerer Typvariablen) als Parameter angegeben werden.

Zwei Beispiele:

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

data List a = Empty |
              (Head a) (List a)
```


Polymorphe Funktionen

...eine Funktion `f` heißt *polymorph*, wenn deren Parameter (in Form einer oder mehrerer Typvariablen) für Argumente unterschiedlicher Typen definiert sind.

Typische Beispiele:

```
depth  :: (Tree a b) -> Int
depth Leaf _         = 1
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

lengthLst :: List a -> Int
lengthLst Empty       = 0
lengthLst (Head _ hs) = 1 + lengthLst hs

-- Zum Vergleich die polymorphe Standardfunktion length
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

$\rightsquigarrow$ *wie schon gesagt, ein typisches Vorgehen in der Informatik*!

...die Essenz eines Datentyps (einer Datenstruktur) ist wie die Essenz darauf arbeitender Funktionen oft unabhängig von bestimmten typspezifischen Details.

(Man vergegenwärtige sich das noch einmal z.B. anhand von Bäumen über ganzen Zahlen, Zeichenreihen, Personen,... oder Listen über Gleitkommazahlen, Wahrheitswerten, Bäumen,... und typischen Funktionen darauf wie etwa zur Bestimmung der Tiefe von Bäumen oder der Länge von Listen.)

Warum polymorphe Typen und Funktionen? (2)

- ...durch Parametrisierung werden gleiche Teile "ausgeklammert" und somit der Wiederverwendung zugänglich!
- ...(i.w.) gleiche Codeteile müssen nicht (länger) mehrfach geschrieben werden.
- ...man spricht deshalb auch von *parametrischer Polymorphie*.

Warum polymorphe Typen und Funktionen? (3)

Polymorphie und die mit ihr verbundene Wiederverwendung unterstützt die...

- Ökonomie der Programmierung (vulgo: "*Schreibfaulheit*")

Insbesondere aber trägt sie bei zu höherer...

- Transparenz und Lesbarkeit
  ...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit (ein Aspekt mit mehreren Dimensionen: Fehlersuche, Weiterentwicklung,...)
  ...als erwünschte Seiteneffekte!
- ...
- Effizienz (der Programmierung)
  $\rightsquigarrow$ *höhere Produktivität, früherer Markteintritt* (*time-to-market*)

Warum polymorphe Typen und Funktionen? (4)

Beachte...

- ...auch in anderen Paradigmen wie etwa imperativer und speziell objektorientierter Programmierung lernt man, den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!
  $\rightsquigarrow$ aktuelles Stichwort: *Generic Java*

Ad hoc Polymorphie

Bisher haben wir besprochen...

- Polymorphie in Form von...
  - (Parametrischer) Polymorphie
  - Polymorphie auf Datentypen

Jetzt ergänzen wir diese Betrachtung um...

- *Ad hoc* Polymorphie (Überladen, "unechte" Polymorphie)

Zur Motivation von Ad hoc Polymorphie

Ausdrücke der Form

```
(+) 2 3          => 5
(+) 27.55 12.8   => 39.63
(+) 12.42 3      => 15.42
```

...sind Beispiele wohlgeformter Haskell-Ausdrücke, wohingegen

```
(+) True False
(+) 'a' 'b'
(+) [1,2,3] [4,5,6]
```

...Beispiele nicht wohlgeformter Haskell-Ausdrücke sind.

Zur Motivation von Ad hoc Polymorphie

Offenbar...

- ist `(+)` nicht monomorph
  ...da `(+)` für mehr als einen Argumenttyp arbeitet
- ist der Typ von `(+)` verschieden von `a -> a -> a`
  ...da `(+)` nicht für jeden Argumenttyp arbeitet

Tatsächlich...

- ist `(+)` typisches Beispiel eines *überladenen* Operators.

Das Kommando `:t (+)` in Hugs liefert

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

Polymorphie vs. Ad hoc Polymorphie

Intuitiv

- Polymorphie
  Der polymorphe Typ `(a -> a)` wie in der Funktion `id :: a -> a` steht abkürzend für:
  $\forall$`(a) a -> a` "...für alle Typen"
- Ad hoc Polymorphie
  Der Typ `(Num a => a -> a -> ->a)` wie in der Funktion `(+) :: Num a => a -> a -> ->a` steht abkürzend für:
  $\forall$`(a`$\in$`Num) a -> a -> a` "...für alle Typen aus `Num`"

Im Haskell-Jargon ist `Num` eine sog.

- *Typklasse*, eine von vielen vordefinierten Typklassen.

Typklassen in Haskell

Informell

- Eine Typklasse ist eine Kollektion von Typen, auf denen eine in der Typklasse festgelegte Menge von Funktionen definiert ist.
- Die Typklasse `Num` ist die Kollektion der numerischen Typen `Int`, `Integer`, `Float`, etc., auf denen u.a. die Funktionen `(+)`, `(*)`, etc. definiert sind.

Hinweis: Vergleiche dieses Klassenkonzept z.B. mit dem Schnittstellenkonzept aus Java. Gemeinsamkeiten, Unterschiede?

Polymorphie vs. Ad hoc Polymorphie

Informell...

- (*Parametrische*) *Polymorphie* ...
  $\leadsto$ gleicher Code trotz unterschiedlicher Typen
- *ad-hoc Polymorphie* (synonym: *Überladen* (engl. *Overloading*))...
  $\leadsto$ unterschiedlicher Code trotz gleichen Namens (mit sinnvollerweise i.a. ähnlicher Funktionalität)

Ein erweitertes Bsp. zu Typklassen (1)

Wir nehmen an, wir seien an der Größe interessiert von

- Listen und
- Bäumen

Der Begriff "Größe" sei dabei typabhängig, z.B.

- Anzahl der Elemente bei Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...

  bei Bäumen

Ein erweitertes Bsp. zu Typklassen (2)

Wir betrachten folgende Baumvarianten...

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

data Tree1 a b = Leaf1 b |
                 Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
             Node2 String Tree2 Tree2
```

...und den Haskellstandardtyp für Listen.

Ein erweitertes Beispiel zu Typklassen

Naive Lösung: Schreibe für jeden Typ eine passende Funktion:

```
sizeT :: Tree a -> Int          -- Zaehlen der Knoten
sizeT Nil          = 0
sizeT (Node n l r) = 1 + sizeT l + sizeT r

sizeT1 :: (Tree1 a b) -> Int    -- Zaehlen der Benennungen
sizeT1 (Leaf1 m)       = 1
sizeT1 (Node1 m n l r) = 2 + sizeT1 l + sizeT1 r

sizeT2 :: Tree2 -> Int          -- Summe der Laengen der Benennungen
sizeT2 (Leaf2 m)     = length m
sizeT2 (Node2 m l r) = length m + sizeT2 l + sizeT2 r

sizeLst :: [a] -> Int           -- Zaehlen der Elemente
sizeLst = length
```


Ein erweitertes Bsp. zu Typklassen (3)

"Smarte" Lösung mithilfe von Typklassen:

```
class Size a where              -- Definition der Typklasse Size
  size :: a -> Int

instance Size (Tree a) where    -- Instanzbildung fuer (Tree a)
  size Nil          = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m)       = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where       -- Instanzbildung fuer Tree2
  size (Leaf2 m)     = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```


Ein erweitertes Bsp. zu Typklassen (4)

Das Kommando `:t size` liefert:

```
size :: Size a => a -> Int
```

...und wir erhalten wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2

size (Leaf1 "adf") => 1
size ((Node1 "asdf" 3
          (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
          (Leaf1 21)) => 7

size (Leaf2 "abc") => 3
size (Node2 "asdf"
       (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac"))
       (Leaf "xy"))      => 17

size [5,3,45,676,7] => 5
size [True,False,True] => 3
```


Definition von Typklassen

Allgemeines Muster einer Typklassendefinition...

```
class Name tv where
    ...signature involving the type variable tv
```

wobei

- `Name` ...Identifikator der Klasse
- `tv` ...Typvariable
- `signature` ...Liste von Namen zusammen mit ihren Typen

Schlussfolgerungen zu Typklassen

Intuitiv...

...Typklassen sind Kollektionen von Typen, für die eine gewisse Menge von Funktionen ("gleicher" Funktionalität) definiert ist.

Beachte...

..."Gleiche" Funktionalität kann nicht syntaktisch erzwungen werden, sondern liegt in der Verantwortung des Programmierers!

Mithin: ...Appell an die *Programmierdisziplin* unabdingbar!

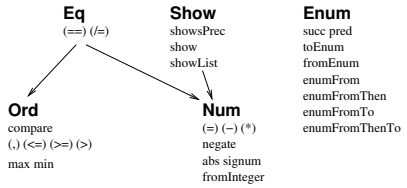
Bsp. einiger in Haskell vordef. Typklassen (1)

Vordefinierte Typklassen in Haskell...

- *Gleichheit* `Eq` ...die Klasse der Typen mit Gleichheitstest
- *Ordnungen* `Ord` ...die Klasse der Typen mit Ordnungsrelationen (wie etwa $<$, $\leq$, $>$, $\geq$, etc.)
- *Aufzählung* `Enum` ...die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29]`)
- *Werte zu Zeichenreichen* `Show` ...die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können
- *Zeichenreihen zu Werten* `Read` ...die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind
- ...

Bsp. einiger in Haskell vordef. Typklassen (2)

Auswahl vordefinierter Typklassen, ihrer Abhängigkeiten, Operatoren und Funktionen in Standard Prelude nebst Bibliotheken:



Quelle: Fethi Rabhi, Guy Lapalme. "Algorithms - A Functional Approach", Addison-Wesley, 1999.

Vorschau auf die kommenden Aufgabenblätter...

Ausgabe des...

- fünften Aufgabenblatts: Mi, den 23.11.2005
  ...Abgabetermine: Mi, den 30.11.2005, und Mi, den 07.12.2005, jeweils 12:00 Uhr
- sechsten Aufgabenblatts: Mi, den 30.11.2005
  ...Abgabetermine: Mi, den 07.12.2005, und Mi, den 14.12.2005, jeweils 12:00 Uhr
- siebten Aufgabenblatts: Mi, den 07.12.2005
  ...Abgabetermine: Mi, den 14.12.2005, und Mi, den 21.12.2005, jeweils 12:00 Uhr

Vorschau auf die nächsten Vorlesungstermine...

- Do, 01.12.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- *Do, 08.12.2005: Keine Vorlesung: Feiertag!*
- Do, 15.12.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Voraussichtlich: Di, 20.12.2005, Vorlesung von 13:00 Uhr bis 14:00 Uhr im Informatik-Hörsaal, Treitlstr.