Heutiges Thema

...selbstdefinierte (neue) Datentypen in Haskell!

⇝ Haskells Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
  - *Spezialfälle*
    - Produkttypen
    - Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen (⇝ "unendliche" Datenstrukturen)
- Polymorphe Typen (⇝ Wiederverwendung)

Zum Einstieg... (1)

Aufzählungs-, *Produkt-* und *Summentypen*: Bsp. grundlegender Typmuster...

- *Aufzählungstypen*
  ⇝ Typen mit endlich vielen Werten
  ...typisches Beispiel: Typ `Jahreszeiten` mit Werten `Fruehling`, `Sommer`, `Herbst` und `Winter`
- *Produkttypen*
  ⇝ Typen mit möglicherweise unendlich vielen Tupelwerten
  ...typisches Beispiel: Typ `Person` mit Werten `(Adam, maennlich, 27)`, `Eva, weiblich, 25)`, etc.
- *Summentypen*
  ⇝ Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten
  ...typisches Beispiel: Typ `Verkehrsmittel` als Vereinigung der (Werte der) Typen `Auto`, `Schiff` und `Flugzeug`

Zum Einstieg... (2)

Realisierung von Typdefinitionen in imperativen Sprachen
...hier am Bsp. von Pascal

- *Aufzählungstypen*

```
TYPE jahreszeiten = (fruehling, sommer, herbst, winter);
     spielkartenfarben = (kreuz, pik, herz, karo);
     werktage = (montag, dienstag, mittwoch,
                 donnerstag, freitag);
     wochenende = (sonnabend, sonntag);
     transportmittel = (fahrrad, auto, schiff, flugzeug);
     form = (kreis, rechteck, quadrat, dreieck);
```

- *Produkttypen*

```
TYPE person = RECORD
                name: ARRAY [1..42] OF char;
                geschlecht: (maennlich, weiblich);
                alter: integer
              END;
```


Zum Einstieg... (3)

- *Summentypen*

```
TYPE verkehrsmittel =
       RECORD
         CASE vkm: transportmittel OF
           fahrrad: (tandem: Boolean);
           auto: (hersteller: ARRAY [1..30] OF char;
                  hubraum: real);
           schiff: (name: ARRAY [1..30] OF char;
                    tiefgang: real;
                    heimathafen: ARRAY [1..50] OF char);
           flugzeug: (spannweite: real;
                      sitzplaetze: integer)
       END;
     figur =
       RECORD
         CASE fgr: form OF
           kreis: (radius: real);
           rechteck : (breite, hoehe: real);
           quadrat : (seitenlaenge, diagonale: real);
           dreieck: (s1, s2, s3: real; rechtwkg: boolean);
       END;
```

Zum Einstieg... (4)

Aufzählungstypen, Produkttypen, Summentypen...

- In Pascal ...drei verschiedene Sprachkonstrukte
- In Haskell ...ein *einheitliches* Sprachkonstrukt
  ⇝ die *algebraische Datentypdefinition*

Zum Einstieg... (5)

Obige Einstiegsdatentypbeispiele in Haskell...

- Aufzählungstyp `Jahreszeiten`

```
data Jahreszeiten = Fruehling | Sommer | Herbst | Winter
```

- Produkttyp `Person`

```
data Person = Mensch Name Geschlecht Alter
```

mit

```
type Name  = String
type Alter = Int
data Geschlecht = Maennlich | Weiblich
```


Zum Einstieg... (6)

- Summentyp `Verkehrsmittel`

```
data Verkehrsmittel = Fahrrad Tandem |
                      Auto Hersteller Hubraum |
                      Schiff Name Tiefgang Heimathafen |
                      Flugzeug Spannweite Sitzplaetze
```

mit

```
type Tandem      = Bool
type Hersteller  = String
type Hubraum     = Float
type Name        = String
type Tiefgang    = Float
type Heimathafen = String
type Spannweite  = Float
type Sitzplaetze = Int
```


Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

```
data Typename
  = Con1 t11 ... t1k1 |
    Con2 t21 ... t2k2 |
       ...            |
    Conn tn1 ... tnkn
```

Sprechweisen:

- `Typename` ...*Typname/-identifikator*
- $\texttt{Con}_i$, $i = 1..n$ ...*Konstruktor(en)/-identifikatoren*
- k_i, $i = 1..n$ ...*Stelligkeit* des Konstruktors $\texttt{Con}_i$, $\mathsf{k}_i \geq 0$, $i = 1, \ldots, n$

Beachte: Typ- und Konstruktoridentifikatoren müssen mit einem Großbuchstaben beginnen!

Konstruktoren...

...können als Funktionsdefinitionen gelesen werden:

$\texttt{Con}_i$ :: $\texttt{t}_{i1}$ -> ... -> $\texttt{t}_{ik_i}$ -> Typename

Elemente eines algebraischen Datentyps konstruiert durch...

...Anwendung eines Konstruktors auf Werte "passenden" Typs, d.h....

$\texttt{Con}_i$ $\texttt{v}_{i1}$ ... $\texttt{v}_{ik_i}$ :: typName

wobei $\texttt{v}_{i1}$ :: $\texttt{t}_{i1}$ ... $\texttt{v}_{ik_i}$ :: $\texttt{t}_{ik_i}$, $j = 1, \ldots, k_i$

Beispiele:

- `Mensch "Adam" Maennlich 27 :: Person`
- `Schiff "Donaukönigin" 2.74 "Wien" :: Verkehrsmittel`
- `Fahrrad True :: Verkehrsmittel`

Aufzählungstypen (1)

Nullstellige Konstruktoren führen auf *Aufzählungstypen*...

Beispiele:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
data Wochenende = Sonnabend | Sonntag
data Geschlecht = Maennlich | Weiblich
data Form       = Kreis | Rechteck | Quadrat | Dreieck
```

Insbesondere auch der Typ der Wahrheitswerte...

```
data Bool = True | False
```

...ist Beispiel eines in Haskell vordefinierten Aufzählungstyps.

Aufzählungstypen (2)

Funktionsdefinitionen über Aufzählungstypen...
$\rightsquigarrow$ üblicherweise mit Hilfe von Pattern-matching.

Beispiel:

```
hatEcken :: Form -> Bool
hatEcken Kreis = False
hatEcken _     = True

istLandgebunden :: Verkehrsmittel -> Bool
istLandgebunden Fahrrad = True
istLandgebunden Auto    = True
istLandgebunden _       = False
```


Produkttypen

(Alternativenlose) mehrstellige Konstruktoren führen auf *Produkttypen*...

Beispiel:

```
type Name   = String
type Alter  = Int
data Person = Mensch Name Geschlecht Alter
```

Beispiele: ...für Werte des Typs `Person`.

```
Mensch "Paul Pfiffig" Maennlich 23  :: Person
Mensch "Paula Plietsch" Weiblich 22 :: Person
```

Beachte: Funktionalität der Konstruktorfunktion ist hier...

```
Mensch :: Name -> Geschlecht -> Alter -> Person
```


Produkttypen vs. Tupeltypen (1)

Der Typ `Person` als...

- *Produkttyp*

  `data Person = Mensch Name Geschlecht Alter`

- *Tupeltyp*

  `type Person = (Name, Geschlecht, Alter)`

Vordergründiger Unterschied:
...in der Tupeltypvariante fehlt der Konstruktor (in diesem Bsp.: `Mensch`)

Produkttypen vs. Tupeltypen (2)

...eine Abwägung von Vor- und Nachteilen.

Produkttypen und ihre typischen...

- *Vorteile* gegenüber *Tupeltypen*
  - Objekte des Typs sind mit dem Konstruktor "markiert" (trägt zur Dokumentation bei)
  - Tupel mit zufällig passenden Komponenten nicht irrtümlich als Elemente des Produkttyps manipulierbar (Typsicherheit! Vgl. früheres Beispiel zur Umrechnung EUR in Yen!)
  - Aussagekräftigere (Typ-) Fehlermeldungen (Typsynonyme können wg. Expansion in Fehlermeldungen fehlen).
- *Nachteile* gegenüber *Tupeltypen*
  - Produkttypelemente sind weniger kompakt, erfordern längere Definitionen (mehr Schreibarbeit)
  - Auf Tupeln vordefinierte polymorphe Funktionen (z.B. `fst`, `snd`, `zip`, `unzip`, ...) stehen nicht zur Verfügung.

Andererseits...

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float

myPi    :: Float
daumen  :: Float
maxTemp :: Temperature
myPi     = 3.14
daumen   = 5.55
maxTemp  = Temp 43.2
```

Eine Funktionsdefinition im Stile von...

```
currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

...wird jetzt durch das Typsystem von Haskell verhindert!

Somit als kurzes Fazit...

...unserer Überlegungen:

- *Typsynonyme* wie...

  ```
  type Euro        = Float
  type Yen         = Float
  type Temperature = Float
  ```

  ...erben alle Operationen von `Float` und sind damit beliebig austauschbar – mit allen Annehmlichkeiten und Gefahren, sprich Fehlerquellen.

- *Produkttypen* wie...

  ```
  data Euro        = EUR Float
  data Yen         = YEN Float
  data Temperature = Temp Float
  ```

  ...erben keinerlei Operationen von `Float`, bieten dafür aber um den Preis zusätzlicher Schreibarbeit Typsicherheit!

Summentypen (1)

Mehrere (null- oder mehrstellige) Konstruktoren führen auf Summentypen...

Beispiel:

```
type Radius       = Float
type Breite       = Float
type Hoehe        = Float
type Seite1       = Float
type Seite2       = Float
type Seite3       = Float
type Rechtwinklig = Bool
data XFigur = Kreis Radius |
              Rechteck Breite Hoehe |
              Quadrat Kantenlaenge |
              Dreieck Seite1 Seite2 Seite3 Rechtwinklig |
              Ebene
```

Die Varianten einer Summe werden durch "|" getrennt.

Summentypen (2)

Beispiele: ...für Werte des Typs erweiterte Figur `XFigur`

```
Kreis 3.14                 :: XFigur
Rechteck 17.0 4.0          :: XFigur
Quadrat 47.11              :: XFigur
Dreieck 3.0 4.0 5.0 True :: XFigur
Ebene                      :: XFigur
```


Zwischenfazit

Beobachtung ...auch Aufzählungstypen sind in Haskell als Summentypen realisiert!

Somit ergibt sich folgende Taxonomie algebraischer Datentypen...

- Summentypen
  - *Spezialfälle*
    - Produkttypen
      ⇝ nur ein Konstruktor, mehrstellig
    - Aufzählungstypen
      ⇝ ein oder mehrere Konstruktoren, alle nullstellig

Wichtig in der Praxis: Rekursive Typen (1)

...der Schlüssel zu (potentiell) unendlichen Datenstrukturen.

Technisch:
...zu definierende Typnamen können rechtsseitig in der Definition benutzt werden.

Beispiele: ...Ausdrücke und zwei Binärbaumvarianten

```
data Expr = Opd Int |
            Add Expr Expr |
            Sub Expr Expr |
            Squ Expr

data BinTree1 = Nil | Node Int BinTree1 BinTree1

data BinTree2 = Leaf Int | Node Int BinTree2 BinTree2
```


Rekursive Typen (2)

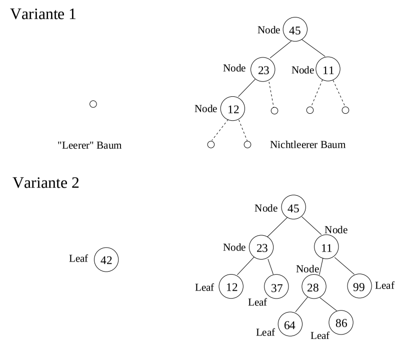
Beispiele ...für Ausdrücke (lies -> als "entspricht").

```
Opd 42 :: Expr                                 -->  42
Add (Opd 17) (Opd 4) :: Expr                   -->  17+4
Add (Squ (Sub (Opd 42) (Squ (2)))) (Opd 12) :: Expr
                                          --> square(42-square(2))+12
```

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

Rekursive Typen (3)

Veranschaulichung der Binärbaumvarianten 1&2 anhand eines Beispiels:



Rekursive Typen (4)

Beispiele ...für Funktionen über Binärbaumvariante 1.

```
valBinTree1 :: BinTree1 -> Int
valBinTree1 Nil              = 0
valBinTree1 (Node n bt1 bt2) = n + valBinTree1 bt1 + valBinTree1 bt2

depthBinTree1  :: BinTree1 -> Int
depthBinTree1 Nil = 0
depthBinTree1 (Node _ bt1 bt2)
              = 1 + max (depthBinTree1 bt1) (depthBinTree1 bt2)
```

Mit diesen Definitionen...

```
valBinTree1 Nil                              ==  0
valBinTree1 (Node 17 Nil (Node 4 Nil Nil))   == 21
depthBinTree1 (Node 17 Nil (Node 4 Nil Nil)) ==  2
depthBinTree1 Nil                            ==  0
```


Rekursive Typen (5)

Beispiele ...für Funktionen über Binärbaumvariante 2.

```
valBinTree2 :: BinTree2 -> Int
valBinTree2 (Leaf n)         = n
valBinTree2 (Node n bt1 bt2) = n + valBinTree2 bt1 + valBinTree2 bt2

depthBinTree2  :: BinTree2 -> Int
depthBinTree2 (Leaf _) = 1
depthBinTree2 (Node _ bt1 bt2)
              = 1 + max (depthBinTree2 bt1) (depthBinTree2 bt2)
```

Mit diesen Definitionen...

```
valBinTree2 (Leaf 3)                                    ==  3
valBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5)))   == 42
depthBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5))) ==  3
depthBinTree2 (Leaf 3)                                  ==  1
```

Wechselweise rekursive Typen

...ein Spezialfall rekursiver Typen.

Beispiel:

```
data Individual = Adult Name Address Biography |
                  Child Name

data Biography  = Parent String [Individual] |
                  NonParent String
```


Polymorphe Typen (1)

Bisher haben wir kennengelernt...

- Polymorphie auf *Funktionen* in Form von...
  - "echte(r)" Polymorphie
  - *Ad hoc* Polymorphie (Überladen, "unechte" Polymorphie))

Jetzt erweitern wir dies um...

- Polymorphie auf *Datentypen*

Polymorphe Typen (2)

Der Schlüssel dazu...

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

Beispiele: ...Paare und Bäume

```
data Pairs a = Pr a a

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


Polymorphe Typen (3)

Mit diesen Definitionen...

```
Pr 17 4    :: Pairs Int
Pr [] [42] :: Pairs [Int]
Pr [] []   :: Pairs [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 Typen unterstützen Wiederververwendung!

Erinnerung: Eine ähnliche Situation hatten wir schon im Zshg. mit *polymorphen Listen* kennengelernt:

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


Polymorphe Typen (4)

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

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

...kann Typunabhängigkeit auch für Funktionen auf Bäumen vorteilhaft ausgenutzt werden, wie hier das Bsp. der Funktion *Tiefe*...

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

...und folgende Aufrufe zeigen:

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


Summentypen erlauben heterogene (1)

...Datenstrukturen, beispielsweise auch *heterogene* Bäume:

```
data CoolTree = LeafS String                   |
                LeafI Int                      |
                NodeS String CoolTree CoolTree |
                NodeI Int CoolTree CoolTree

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ CoolTree
depthCT :: CoolTree -> Integer
depthCT (LeafS _) = 1
depthCT (LeafI _) = 1
depthCT (NodeS _ t1 t2) = 1 + max (depthCT t1) (depthCT t2)
depthCT (NodeI _ t1 t2) = 1 + max (depthCT t1) (depthCT t2)

depthCT1 :: CoolTree -> Integer
depthCT1 (NodeS _ t1 t2) = 1 + max (depthCT t1) (depthCT t2)
depthCT1 (NodeI _ t1 t2) = 1 + max (depthCT t1) (depthCT t2)
depthCT1 _               = 1
```

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

```
depthCT2 :: CoolTree -> Integer
depthCT2 (_ _ t1 t2) = 1 + max (depthCT t1) (depthCT t2)
depthCT2 _           = 1
```

Summentypen erlauben heterogene (2)

...Datenstrukturen. Und *Polymorphie* kann zusätzlich ausgenutzt werden!

```
data PolCoolTree a b = Leafa a                                 |
                       Leafb b                                 |
                       Nodea a (PolCoolTree a b) (PolCoolTree a b) |
                       Nodeb b (PolCoolTree a b) (PolCoolTree a b)

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ PolCoolTree
depthPCT :: PolCoolTree a b -> Integer
depthPCT (Leafa _) = 1
depthPCT (Leafb _) = 1
depthPCT (Nodea _ t1 t2) = 1 + max (depthPCT t1) (depthPCT t2)
depthPCT (Nodeb _ t1 t2) = 1 + max (depthPCT t1) (depthPCT t2)

depthPCT1 :: PolCoolTree a b -> Integer
depthPCT1 (Nodea _ t1 t2) = 1 + max (depthPCT t1) (depthPCT t2)
depthPCT1 (Nodeb _ t1 t2) = 1 + max (depthPCT t1) (depthPCT t2)
depthPCT1 _               = 1
```

Das heißt...

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

Ausblick

Mehr über Polymorphie...

...beim nächsten Mal!