Algebraische Datentypen in Haskell

```
data Foo = Foo { bar :: Int, baz :: String }
    deriving Show
```

Bezeichnungen (benannte Notation)

- `data Foo` ist Typname
- `Foo { .. }` ist Konstruktor
- `bar, baz` sind Komponenten

```
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
y = Foo 3 "bar"
```

Mathematisch: Produkt `Foo` = `Int` $\times$ `String`

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

```
data T = A { foo :: Int }
       | B { bar :: String, baz :: Bool }
    deriving Show
```

Bespiele (in Prelude vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

Mathematisch: (disjunkte) Vereinigung

`Bool` = { `False` } ∪ { `True` }

Fallunterscheidung, Pattern Matching

```
data T = A { foo :: Int }
       | B { bar :: String }
```

Fallunterscheidung:

```
f :: T -> Int
f x = case x of
    A {} -> foo x
    B {} -> length $ bar x
```

Pattern Matching (Bezeichner `f,b` werden lokal gebunden):

```
f :: T -> Int
f x = case x of
    A { foo = f } -> f
    B { bar = b } -> length b
```

`data` und `case`

typisches Vorgehen beim Programmieren einer Funktion

```
f :: T -> ...
```

Für jeden Konstruktor des Datentyps

```
data T = C1 ...
       | C2 ...
```

schreibe einen Zweig in der Fallunterscheidung

```
f x = case x of
   C1 ... -> ...
   C2 ... -> ...
```

Zusammengesetzte Datentypen

Operationen:

Produkt $A \times B$

Beispiel:

```
data Punkt = Punkt { x :: Float, y :: Float}
data Kreis = Kreis { mp :: Punkt, rad :: Float
```

(disjunkte) Vereinigung $A \cup B$

Beispiel Wahrheitswerte (vordefiniert)

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

```
data Shape = Circle { mp :: Punkt, rad :: Floa
           | Rect { ol, ur :: Punkt}
```

```
umfang :: Shape -> Float
umfang s = case s of
         Circle {} -> 2 * pi * ( rad s )
         Rect ol ur -> ...
```

Potenz $A^B = \{f : B \to A\}$

z.B. `gerade_laenge :: String -> Bool`

Polymorphe Datentypen – Beispiele

```
data Pair a b = Pair a b
```
(Produkt)

Beispiel: 4 Elemente in `Pair Bool Bool` =
{`Pair False False`, `Pair False True`,
`Pair True False`, `Pair True True`}

```
data Either a b = Left a | Right b
```
(Vereinigung)

Beispiel: 4 Elemente in `Either Bool Bool` =
{`Left False`, `Left True`, `Right False`, `Right True`}

```
data Maybe a = Nothing | Just a
```
(Vereinigung)

Listen: `data List a = Nil | Cons a (List a)`

Rekursive Datentypen

Wiederholung: Peano-Zahlen

```
data Nat = Z | S Nat
```

Pattern Matching:

```
f :: Nat a -> ...
f x = case x of
    Z   ->  ...
    S x -> ...
```

Listen (polymorph mit Typvariable `a`)

```
data List a = Nil
    | Cons { head :: a, tail :: List a}
```

Pattern Matching:

```
f :: List a -> ...
f xs = case xs of
    Nil         -> ...
    Cons x xss -> ...
```

Rekursive Funktionen – Beispiele

```
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)
```

(rekursive) Funktionen über Peano-Zahlen, z.B.

```
double :: Nat -> Nat
double x = case x of
   Z    -> Z
   S x' -> S ( S ( double x' ) )
```

(rekursive) Funktionen auf Listen von Peano-Zahlen, z.B.

```
doubles :: List Nat -> List Nat
doubles xs = case xs of
    Nil         -> Nil
    Cons x xss -> Cons ( double x ) ( doubles xss )
```

(rekursive) Funktionen auf polymorphen Listen, z.B.

```
append :: List a ->  List a ->  List a
append xs ys = case xs of
    Nil         -> ys
    Cons x xss -> Cons x (append xss ys)
```

Vordefinierte Datentypen in Haskell

einfach:

- `Bool` (`False`, `True`)
- `Char`
- `Int`, `Integer` (beliebig genau)
- `Float`

zusammengesetzt (Typkonstruktoren):

- selbstdefiniert `constr typ1 ... typn`
  z.B. `HR` , `Nat` , `List a` (polymorph mit Typ-Variable `a`)
- vordefiniert
  - Tupel-Konstruktor (polymorph) `(a, b)`, `(a, b, c)`, `(a1, a2, ...)`
    z.B. `(1, True, 'B') :: (Int, Bool, Char)`
  - Listen-Konstruktor (polymorph) `[a]`,
    z.B. `[3,5,2] :: [Int]`,
    `[['I', 'N'],['B']] :: [[Char]]`
  - `String = [Char]`, z.B. `"INB" = ['I','N','B']`

Typsynonyme

(Um-)Benennung vorhandener Typen (meist als Kurzform)

Beispiel:

```
type String = [ Char ]
type Name = String
type Telefonnummer = Int
type Telefonbuch = [ ( Name ,  Telefonnummer ) ]

nummern :: Name -> Telefonbuch -> [ Telefonnummer ]
nummern name [] = []
nummern name ( ( n , t ) : rest ) ...
```

allgemeiner: Wörterbücher

```
type Woerterbuch a b = [ ( a, b ) ]
```

rekursive Typen sind nicht als Typsynonym definierbar