Einführung in die Funktionale Programmierung mit Haskell

Datentypen und Polymorphie

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. Mai 2013

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter (z. Bool)

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter (z. Bool)

Funktionsdefinition in Haskell

```
foo :: typ1 -> typ2 -> typ3 -> ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n = expr2
foo pat31 ... pat3n = expr3
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: typ1 -> ... -> typ3 -> ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
    | grd311, ..., grd31k = expr31
    | grd321, ..., grd32l = expr32
```

- Typdeklaration (optional)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Ausgeführt wird erster zutreffender Match von oben-nach unten.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` – erlaubt auch Funktionsdefinitionen
- Funktionsabstraktion: `\x y z -> ...` – anonyme Funktionen
- Funktionsanwendung: `foo arg`

Operationen ==, <, +, *, ... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen `==`, `<`, `+`, `*`,... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *, ... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *, ... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *, ... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,... sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,. . . sind kein Spezialfall in Haskell, sondern

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,. . . sind kein Spezialfall in Haskell, sondern Funktionsanwendung (ggf. in Infix Notation)

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,. . . sind kein Spezialfall in Haskell, sondern Funktionsanwendung (ggf. in Infix Notation)

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Funktionsrumpf

Der Funktionsrumpf ist ein Ausdruck. Werden Patterns+Guards eingesetzt, gibt es einen Ausdruck pro Zweig. Alle diese Ausdrücke müssen den gleichen Typ haben – den Ergebnistyp der Funktion!

Bisher haben wir kennengelernt:

- Konditional: `if-then-else`
- Pattern-Match: `case-of->`
- Lokale Definitionen: `let-in` erlaubt auch Funktionsdefinition
- Funktionsabstraktion: `\x y z -> ...` anonyme Funktionen
- Funktionsanwendung: `foo arg` durch Leerzeichen

Operationen ==, <, +, *,... sind kein Spezialfall in Haskell, sondern Funktionsanwendung (ggf. in Infix Notation)

Zusätzlich haben wir noch lokale Definitionen über top-level Funktionsdefinition und Wächter mit `where` gesehen.

Typabkürzungen

Der Typ `String` ist nur eine Abkürzung für eine `Char`-Liste:

```
type String = [Char]
```

Solche Abkürzungen darf man genau so auch selbst definieren: Hinter dem Schlüsselwort `type` schreibt meinen einen frischen Namen, der mit einem Großbuchstaben beginnt und hinter dem Gleichheitszeichen folgt ein bekannter Typ, z.B.

```
type MyWeirdType = (Double,[(Bool,Integer)])
```

Für die Ausführung von Programmen ist dies unerheblich. Typabkürzungen dienen primär zur Verbesserung der Lesbarkeit. Leider ignoriert GHC/GHCi Typabkürzungen meistens, d.h. GHCi gibt fast immer `[Char]` anstelle von `String` aus.

Es gibt noch weitere zusammengesetzte Typen, z.B. Records, Funktionstypen, etc.

Newtype

Bei Besprechung von Foliensatz 02, “Typabkürzung” S.16, richtig bemerkt, dass Typabkürzungen mit `type` beliebig vertauschbar sind, aber manchmal ungewollt sein kann, z.B. wenn man eine Anzahl Äpfel und Anzahl Birnen als `Int` repräsentiert, aber versehentlich Vertauschungen ausschliessen möchte:

```
type Ebbel = (Int,Int) -- Anzahl Äpfel  und Stückpreis
type Berne = (Int,Int) -- Stückpreis und Anzahl Birnen

abbelUffesse :: Ebbel -> Ebbel
abbelUffesse (zahl,preis) = (zahl-1,preis)

moiBerne :: Berne
moiBerne = (120,3) -- 3 Birnen, 1,20EUR/Stück

> abbelUffesse moiBerne
(129,3)
```

Newtype

Lösung: **Newtype** Deklatation

```
newtype Ebbel = Ebbel (Int,Int) -- Anzahl und Preis
newtype Berne = Berne (Int,Int) -- Preis und Anzahl

abbelUffesse :: Ebbel -> Ebbel
abbelUffesse (Ebbel (zahl,preis)) = Ebbel (zahl-1,preis)

moiBerne = Berne (120,3) -- 3 Birnen, 1,20EUR/Stück

> ebbelUffesse moiBerne
Couldn't match type 'Ebbel' with actual type 'Berne'
```

- **Konstruktor** `Ebbel` zur Konstruktion von Werten
- Pattern-Matching für Typ Ebbel hat Form `(Ebbel _)`

```
ebbel2berne :: Ebbel -> Berne
ebbel2berne (Ebbel (zahl,preis)) = Berne (preis, zahl)
```

Newtype

Lösung: **Newtype** Deklatation

```
newtype Ebbel = Ebbel (Int,Int) -- Anzahl und Preis
newtype Berne = Berne (Int,Int) -- Preis und Anzahl

abbelUffesse :: Ebbel -> Ebbel
abbelUffesse (Ebbel (zahl,preis)) = Ebbel (zahl-1,preis)

moiBerne = Berne (120,3) -- 3 Birnen, 1,20EUR/Stück

> ebbelUffesse moiBerne
Couldn't match type 'Ebbel' with actual type 'Berne'
```

- **Konstruktor** Ebbel zur Konstruktion von Werten
- Pattern-Matching für Typ Ebbel hat Form (Ebbel _)

```
ebbel2berne :: Ebbel -> Berne
ebbel2berne (Ebbel (zahl,preis)) = Berne (preis, zahl)
```

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon Bool mit True?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon Bool mit True?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon Bool mit True?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon `Bool` mit `True`?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon `Bool` mit `True`?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon `Bool` mit `True`?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon `Bool` mit `True`?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Newtype

Syntax für Newtype Deklatation:

```
newtype Typname = Konstruktor <typ>
```

- Schlüsselwort `newtype`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- Gefolgt von genau einem bekannten Typ-parameter

Konstruktoren und Typen können gleich sein.
Verwechslungsgefahr besteht keine:
– das eine ist ein **Typ**,
– das andere ein **Wert**.

Und wer verwechselt schon `Bool` mit `True`?

Newtype nur optimierter Spezialfall einer Datentypdeklaration!

Datentypen

Ein Typ oder **Datentyp** ist eine Menge von Werten.

Beispiele:

- Ein Wert des Typs `Bool` ist entweder `True` oder `False`.
  In Haskell spezifizieren wir dies durch folgende Syntax:

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

  `True` und `False` werden hier auch **Konstruktoren** genannt, da man damit einen Wert des Typs konstruieren kann.

- Der Typ `Int` hat die Werte `0,1,-1,2,-2,3,...`

  ```
  data Int = 0 | 1 | -1 | 2 | -2 | 3 | ...
  ```

  im Gegensatz zu `Bool` nicht tatsächlich so definiert

– Das `|` liest man als “oder”
– Konstuktoren können mit Pattern-Matching behandelt werden
– Datentypdeklaration dieser einfachen Form nennt man auch **Aufzählungen** (engl. “Ennumeration”)

Datentypen

Ein Typ oder **Datentyp** ist eine Menge von Werten.

Beispiele:

- Ein Wert des Typs `Bool` ist entweder `True` oder `False`.
  In Haskell spezifizieren wir dies durch folgende Syntax:

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

  `True` und `False` werden hier auch **Konstruktoren** genannt, da man damit einen Wert des Typs konstruieren kann.

- Der Typ `Int` hat die Werte `0,1,-1,2,-2,3,...`

  ```
  data Int = 0 | 1 | -1 | 2 | -2 | 3 | ...
  ```

  im Gegensatz zu `Bool` nicht tatsächlich so definiert

– Das | liest man als “oder”

– Konstuktoren können mit Pattern-Matching behandelt werden

– Datentypdeklaration dieser einfachen Form nennt man auch **Aufzählungen** (engl. “Ennumeration”)

Datentypen

Ein Typ oder **Datentyp** ist eine Menge von Werten.

Beispiele:

- Ein Wert des Typs `Bool` ist entweder `True` oder `False`.
  In Haskell spezifizieren wir dies durch folgende Syntax:

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

  `True` und `False` werden hier auch **Konstruktoren** genannt, da man damit einen Wert des Typs konstruieren kann.

- Der Typ `Int` hat die Werte `0,1,-1,2,-2,3,...`

  ```
  data Int = 0 | 1 | -1 | 2 | -2 | 3 | ...
  ```

  im Gegensatz zu `Bool` nicht tatsächlich so definiert

– Das | liest man als “oder”

– Konstuktoren können mit Pattern-Matching behandelt werden

– Datentypdeklaration dieser einfachen Form nennt man auch **Aufzählungen** (engl. “Ennumeration”)

Datentypen

Ein Typ oder **Datentyp** ist eine Menge von Werten.

Beispiele:

- Ein Wert des Typs `Bool` ist entweder `True` oder `False`.
  In Haskell spezifizieren wir dies durch folgende Syntax:

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

  `True` und `False` werden hier auch **Konstruktoren** genannt, da man damit einen Wert des Typs konstruieren kann.

- Der Typ `Int` hat die Werte `0`,`1`,`-1`,`2`,`-2`,`3`,...

  ```
  data Int = 0 | 1 | -1 | 2 | -2 | 3 | ...
  ```

  im Gegensatz zu `Bool` nicht tatsächlich so definiert

– Das | liest man als “oder”
– Konstuktoren können mit Pattern-Matching behandelt werden
– Datentypdeklaration dieser einfachen Form nennt man auch **Aufzählungen** (engl. “Ennumeration”)

Beispiel

Behandeln eines Datentypen durch Pattern-Matching:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

next :: Day -> Day
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon
```

- Datentyp `Day` hat 7 Alternativen (Konstuktoren)
- Funktion `next` zählt einfach einen Wochentag weiter

Konstruktoren mit Argumenten

Konstruktoren können sich auch andere Werte merken:

```
data Ebbel = Ebbel (Int,Int) -- 1 Argument
data Berne = Berne Int Int   -- 2 Argumente

createBerne :: Int -> Int -> Berne
createBerne preis zahl = Berne preis zahl


ebbel2berne :: Ebbel -> Berne
ebbel2berne (Ebbel (zahl, preis)) = Berne preis zahl

berne2ebbel :: Berne -> Ebbel -> Berne
berne2ebbel (Berne preis zahl) = Ebbel (zahl, preis)
```

Konstruktoren als Funktionen

Konstruktoren kann man als Funktionen betrachten, die Parameter auf einen Wert des entsprechenden Typs abbilden:

```
data Ebbel = Ebbel (Int,Int) -- 1 Argument
data Berne = Berne Int Int   -- 2 Argumente

createBerne :: Int -> Int -> Berne -- nutzlose Funktion!
createBerne preis zahl = Berne preis zahl

> :type Berne
Berne :: Int -> Int -> Berne
> :type Ebbel
Ebbel :: (Int, Int) -> Ebbel
```

- Im Gegensatz zu `newtype` beliebig viele Argumente möglich, also auch kein Argument oder mehr als eines.
- Auch Infix-Konstruktoren möglich, müssen mit `:` beginnen

Konstruktoren als Funktionen

Konstruktoren kann man als Funktionen betrachten, die Parameter auf einen Wert des entsprechenden Typs abbilden:

```
data Ebbel = Ebbel (Int,Int) -- 1 Argument
data Berne = Berne Int Int   -- 2 Argumente

createBerne :: Int -> Int -> Berne -- nutzlose Funktion!
createBerne preis zahl = Berne preis zahl

> :type Berne
Berne :: Int -> Int -> Berne
> :type Ebbel
Ebbel :: (Int, Int) -> Ebbel
```

- Im Gegensatz zu `newtype` beliebig viele Argumente möglich, also auch kein Argument oder mehr als eines.
- Auch Infix-Konstruktoren möglich, müssen mit `:` beginnen

Kombination: Alternativen & Argumente

```
data Früchte = Apfel (Int,Int)
             | Birne Int Int
             | Banane Int Int Double

meineFrüchte :: [Früchte]
meineFrüchte = [Apfel (2,90), Apfel (3,300),
                Birne 10 1, Banane 7 80 0.3]

hatApfel :: [Früchte] -> Bool
hatApfel           []  = False
hatApfel ((Apfel _):_) = True
hatApfel (  _      :t) = hatApfel t

> hatApfel meineFrüchte
True
> hatApfel (drop 2 meineFrüchte)
False
```

Kombination: Alternativen & Argumente

```
data Früchte = Apfel (Int,Int)
             | Birne Int Int
             | Banane Int Int Double
```

Es empfiehlt sich meist, die Alternativen in einer eigenen Funktion zu behandeln:

```
gesamtPreis :: [Früchte] -> Int
gesamtPreis []     = 0
gesamtPreis (h:t) = preis h + gesamtPreis t
  where
    preis (Apfel (z,p))  = z * p
    preis (Birne  p z )  = z * p
    preis (Banane p z _) = z * p
```

```
> gesamtPreis meineFrüchte
1650
```

Rekursive Datentypen

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

Beispiel: Listen

```
data IntList = LeereListe | ListKnoten Int IntList

myList :: IntList
myList = ListKnoten 1 (ListKnoten 2 (LeereListe))

mySum :: IntList -> Int
mySum LeereListe = 0
mySum (ListKnoten h t) = h + mySum t

> mySum myList
3
```

Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))
```

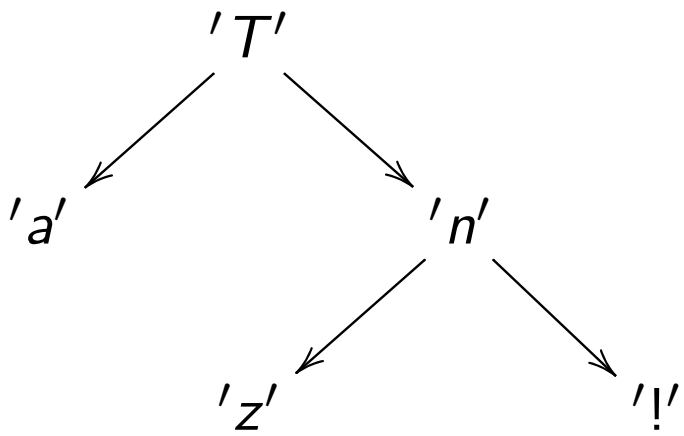
Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))
```



Terminologie

- **Wurzel**-knoten 'T'
- **Blätter** 'a','z','!'
- **linker Teilbaum** von **Knoten** 'n' ist das Blatt 'z'

Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))

dfCollect :: Baum -> String
dfCollect (Blatt c) = [c]
dfCollect (Knoten links c rechts)
          = c : dfCollect links ++ dfCollect rechts
```

Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))

dfCollect :: Baum -> String
dfCollect (Blatt c) = [c]
dfCollect (Knoten links c rechts)
          = c : dfCollect links ++ dfCollect rechts


> dfCollect myBaum
"Tanz!"
```

Wechelseitig Rekursiv

Beispiel:

```
data Datei = Datei String | Verzeichnis Dir
data Dir   = Local String [Datei] | Remote String

data :: Dir
data = Dir "root"
        [Datei "info.txt"
        ,Verzeichnis (Remote "my.url/work")
        ,Verzeichnis (Local "tmp" [])
        ,Datei "help.txt"
        ]
```

- Reihenfolge der Definition ist egal
- Ganz normale Verwendung

Mann könnte in diesem Beispiel auch nur einen Datentyp mit 3 Alternativen definieren, doch dann könnte man kein Funktionen schreiben, welche nur gezielt Verzeichnisse bearbeiten.

Typparameter

Datentypen können **Typvariablen** als Parameter verwenden:

Beispiel: Listen

```
data List a = Leer | Element a (List a)

iList :: List Int
iList = Element 1 (Element 2 (Leer))

iSum :: List Int -> Int
iSum Leer = 0
iSum (Element h t) = h + iSum t

type IntList = List Int -- Typspezialisierung
```

- Typvariablen werden immer klein geschrieben

Typparameter

Datentypen können **Typvariablen** als Parameter verwenden

Beispiel: Listen

```
data List a = Leer | Element a (List a)

iSum :: List Int -> Int

myLength :: (List a) -> Int
myLength Leer = 0
myLength (Element _ t) = 1 + myLength t

> myLength (Element 'a' (Element 'b' Leer))
2
```

Funktion `myLength` kann mit Listen umgehen, die einen beliebigen Typ in sich tragen; im Gegensatz zu `iSum`.
Solche Funktionen nennt man auch **polymorph**.

Polymorphe Funktionen

Beispiele polymorpher Funktionen aus der Standardbibliothek:

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

fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

replicate :: Int -> a -> [a]

drop :: Int -> [a]-> [a]

zip :: [a] -> [b] -> [(a,b)]
```

Records

Wenn ein Datentyp viele Argumente hat, kann man diesen einen Namen geben:

```
data Person' = Mann'  String  Int
             | Frau'  String  Double

data Person  = Mann { name:: String, alter  :: Int    }
             | Frau { name:: String, gewicht:: Double}
```

Der Datenyp `Person` kann ganz genauso verwendet werden, wie `Person'` auch, so gar mit der gleichen Syntax:

```
p1 :: Person
p1 = Mann "Jaimie" 42

showPerson :: Persion -> String
showPerson (Mann n a)  =  n ++ "(" ++ (show a) ++ ")"
```

Records

```
data Person = Mann { name:: String, alter  :: Int    }
            | Frau { name:: String, gewicht:: Double}
p1 = Mann "Jaimie" 42
p2 = Frau { gewicht=58.7, name="Cersei" }
```

Matching darf **Feldnamen** verwendet; die Reihenfolge ist beliebig:

```
showPerson :: Person -> String
showPerson (Mann n a)  =  n ++ "(" ++ (show a) ++ ")"
showPerson  Frau { gewicht=g, name=n }  =  n
```

Pattern matching auf **Record-Felder** darf auch partiell sein:

```
showPerson  Frau { name=n } =  n

isMann :: Persion -> Bool
isMann Mann {} = True
isMann   _     = False
```

Record Projektionen

```
data Person = Mann { name:: String, alter  :: Int    }
            | Frau { name:: String, gewicht:: Double}
p1 = Mann "Jaimie" 42
p2 = Frau { gewicht=58.7, name="Cersei" }
```

Es werden automatisch **partielle Projektionen** definiert:

```
name    :: Person -> String
alter   :: Person -> Int
gewicht :: Person -> Double
```

```
> name p1
"Jaimie"
> gewicht p1
*** Exception: No match in record selector gewicht
```

Record Pseudoupdate

```
data Person = Mann { name:: String, alter  :: Int    }
            | Frau { name:: String, gewicht:: Double}
p1 = Mann "Jaimie" 42
p2 = Frau { gewicht=58.7, name="Cersei" }
```

Es gibt auch funktionale "Field-updates" (Kopien werden erstellt)

```
p3  = p1 { name = "Tyrion" }
p3' = p3 { alter= 1 + alter p3 }
```

Record-Typen kann man ohne große Code-Änderungen nachträglich erweitern, denn

```
p4  = Frau { name="Daenerys" }
```

ist identisch zu

```
p4  = Frau { name="Daenerys", gewicht=undefined }
```

GHC gibt aber entsprechende Warnungen heraus.

Record Pseudoupdate

```
data Person = Mann { name:: String, alter  :: Int    }
            | Frau { name:: String, gewicht:: Double}
p1 = Mann "Jaimie" 42
p2 = Frau { gewicht=58.7, name="Cersei" }
```

Es gibt auch funktionale "Field-updates" (Kopien werden erstellt)

```
p3  = p1 { name = "Tyrion" }
p3' = p3 { alter= 1 + alter p3 }
```

Record-Typen kann man ohne große Code-Änderungen nachträglich erweitern, denn

```
p4  = Frau { name="Daenerys" }
```

ist identisch zu

```
p4  = Frau { name="Daenerys", gewicht=undefined }
```

GHC gibt aber entsprechende Warnungen heraus.

Zusammenfassung: Datentypen

- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- Moderne Programmiersprachen ermöglichen, dass der Benutzer neue Typen definieren kann
- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtetet werden
- **Records** erlauben Benennung dieser Typparameter
- Spezialfall **newtype**: nur 1 Konstruktor mit 1 Argument nur interessant zur Optimierung
- Datentypen können (wechselseitig) rekursiv definiert werden
- **Polymorphe Funktionen** können mit gleichem Code verschiedene Typen verarbeiten

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen – lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_k)
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_k)
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Typklassen

Problem:
Wie können Funktionen wie (==) auf neue benutzerdefinierte Typen angewendet werden?
Antwort: Dank **Typklassen** funktioniert das **Überladen** in Haskell sicher und ohne irgendwelche eingebauten Sonderbehandlungen!