Praktische Informatik 3: Funktionale Programmierung

Vorlesung 2 vom 25.10.2016: Funktionen und Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II und Effizenzaspekte
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Organisatorisches
- Definition von Funktionen
  - Syntaktische Feinheiten
- Bedeutung von Haskell-Programmen
  - Striktheit
- Definition von Datentypen
  - Aufzählungen
  - Produkte

Organisatorisches

- Verteilung der Tutorien (laut stud.ip):

Mi	08 – 10	GW1 A0160	Berthold Hoffmann	16 (50)	4
	10 – 12	GW1 A0160	Johannes Ganser	43 (50)	9
	12 – 14	MZH 1110	Johannes Ganser	35 (35)	9
	14 – 16	GW1 B2070	Alexander Kurth	25 (25)	7
Do	08 – 10	MZH 1110	Tobias Brandt	33 (35)	10
	10 – 12	GW1 B2130	Tristan Bruns	25 (25)	11

- Insgesamt 50 Gruppen (ca. 8 pro Tutorium)
- Wenn möglich, frühes Mittwochstutorium belegen.

Definition von Funktionen

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind Turing-mächtig.

- Funktionen müssen partiell sein können.

Haskell-Syntax: Charakteristika

- Leichtgewichtig
  - Wichtigstes Zeichen:
- Funktionsapplikation: `f a`
  - Keine Klammern
  - Höchste Priorität (engste Bindung)
- Abseitsregel: Gültigkeitsbereich durch Einrückung
  - Keine Klammern ($\{ \ldots \}$)
- Auch in anderen Sprachen (Python, Ruby)

Haskell-Syntax: Funktionsdefinition

Generelle Form:

- Signatur:

```
max :: Int→ Int→ Int
```

- Definition:

```
max x y = if x < y then y else x
```

  - Kopf, mit Parametern
  - Rumpf (evtl. länger, mehrere Zeilen)
  - Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf
  - Was gehört zum Rumpf (Geltungsberereich)?

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

$$\mathtt{f}\ \mathtt{x}_1\ \mathtt{x}_2 \ldots \mathtt{x}_n = \mathtt{E}$$

- Geltungsbereich der Definition von f:
  alles, was gegenüber f eingerückt ist.
- Beispiel:

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

- Gilt auch verschachtelt.
- Kommentare sind passiv (heben das Abseits nicht auf).

Haskell-Syntax II: Kommentare

- Pro Zeile: Ab -- bis Ende der Zeile

```
f x y = irgendwas  -- und hier der Kommentar!
```

- Über mehrere Zeilen: Anfang {-, Ende -}

```
{-
   Hier faengt der Kommentar an
   erstreckt sich ueber mehrere Zeilen
   bis hier                                  -}
f x y = irgendwas
```

  - Kann geschachtelt werden.

Haskell-Syntax III: Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen ...

```
f x y = if B1 then P else
        if B2 then Q else...
```

...bedingte Gleichungen:

```
f x y
  | B1 =...
  | B2 =...
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: Laufzeitfehler! Deshalb:

```
  | otherwise =...
```

Haskell-Syntax IV: Lokale Definitionen

- Lokale Definitionen mit **where** oder **let**:

```haskell
f x y
  | g = P y
  | otherwise = f x where
    y = M
    f x = N x
```

```haskell
f x y =
 let y = M
     f x = N x
 in  if g then P y
          else f x
```

- f, y, ... werden gleichzeitig definiert (Rekursion!)
- Namen f, y und Parameter (x) überlagern andere
- Es gilt die Abseitsregel
  - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

Bedeutung von Funktionen

Bedeutung (Semantik) von Programmen

- Operationale Semantik:
  - Durch den Ausführungsbegriff
  - Ein Programm ist, was es tut.
- Denotationelle Semantik:
  - Programme werden auf mathematische Objekte abgebildet (Denotat).
  - Für funktionale Programme: rekursiv definierte Funktionen

Äquivalenz von operationaler und denotationaler Semantik

Sei P ein funktionales Programm, $\rightarrow_P$ die dadurch definierte Reduktion, und $[\![P]\!]$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \rightarrow_P v \iff [\![P]\!](t) = v$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1

  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9
- Von innen nach außen (innermost-first):

  inc (double (inc 3)) →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9
- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  → (2*(3+ 1))+ 1
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

```
double :: Int → Int
double x = 2*x
```

- Reduktion von `inc (double (inc 3))`
- Von außen nach innen (outermost-first):

  `inc (double (inc 3))` → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  → (2*(3+ 1))+ 1
  → 2*4+1 → 9

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`

Auswertungsstrategien

```haskell
addx :: String → String
addx s = 'x': s
```

```haskell
double :: String→ String
double s = s++ s
```

- Reduktion von addx (double (addx "y"))
- Von außen nach innen (outermost-first):
  addx (double (addx "y")) →

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          →
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          →
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  →
```

Auswertungsstrategien

```haskell
addx :: String → String
addx s = 'x': s
```

```haskell
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          →
```

Auswertungsstrategien

```haskell
addx :: String → String
addx s = 'x': s
```

```haskell
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          → addx (double ("xy"))
                          →
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          → addx (double ("xy"))
                          → addx ("xy" ++ "xy")
                          →
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von addx (double (addx "y"))
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          → addx (double ("xy"))
                          → addx ("xy" ++ "xy")
                          → addx "xyxy"
                          →
```

Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von `addx (double (addx "y"))`
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          → addx (double ("xy"))
                          → addx ("xy" ++ "xy")
                          → addx "xyxy"
                          → 'x': "xyxy" → "xxyxy"
```

Konfluenz

- Sei $\xrightarrow{*}$ die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

$\xrightarrow{*}$ ist konfluent gdw:
Für alle r, s, t mit $s \xleftarrow{*} r \xrightarrow{*} t$ gibt es u so dass $s \xrightarrow{*} u \xleftarrow{*} t$.

- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Funktionale Programme sind für jede Auswertungsstrategie konfluent.

Termination und Normalform

Definition (Termination)

$\rightarrow$ ist terminierend gdw. es keine unendlichen Ketten gibt:

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots t_n \rightarrow \ldots$$

Theorem (Normalform)

Terminierende funktionale Programme werten unter jeder Auswertungsstragie jeden Ausdruck zum gleichen Wert aus (der Normalform).

- Auswertungsstrategie für nicht-terminierende Programme relevant.
- Nicht-Termination nötig (Turing-Mächtigkeit)

Auswirkung der Auswertungsstrategie

- Outermost-first entspricht call-by-need, verzögerte Auswertung.
- Innermost-first entspricht call-by-value, strikte Auswertung
- Beispiel:

```
repeat :: Int→ String→ String
repeat n s = if n == 0 then ""
             else s ++ repeat (n−1) s
```

```
undef :: String
undef = undef
```

- Auswertung von `repeat 0 undef`

Striktheit

Definition (Striktheit)

Funktion f ist strikt $\iff$ Ergebnis ist undefiniert sobald ein Argument undefiniert ist.

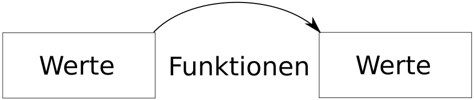
- Denotationelle Eigenschaft (nicht operational)
- Java, C etc. sind call-by-value (nach Sprachdefinition) und damit strikt
- Haskell ist nicht-strikt (nach Sprachdefinition)
  - `repeat0 undef` muss `""` ergeben.
  - Meisten Implementationen nutzen verzögerte Auswertung
- Fallunterscheidung ist immer nicht-strikt.

Datentypen

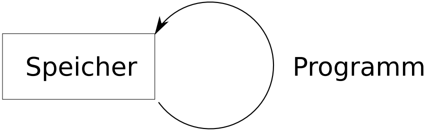
Datentypen als Modellierungskonstrukt

Programme manipulieren ein Modell (der Umwelt)

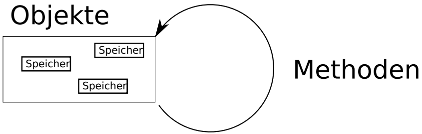
- Funktionale Sicht:



- Imperative Sicht:



- Objektorientierte Sicht:



Typkonstruktoren

- Aufzählungen
- Produkt
- Rekursion
- Funktionsraum

Beispiel: Uncle Bob's Auld-Time Grocery Shoppe

Ein Tante-Emma Laden wie in früheren Zeiten.

Beispiel: Uncle Bob's Auld-Time Grocery Shoppe

Äpfel	Boskoop	55	ct/Stk
	Cox Orange	60	ct/Stk
	Granny Smith	50	ct/Stk
Eier		20	ct/Stk
Käse	Gouda	14,50	€/kg
	Appenzeller	22.70	€/kg
Schinken		1.99	€/100 g
Salami		1.59	€/100 g
Milch		0.69	€/l
	Bio	1.19	€/l

Aufzählungen

- Aufzählungen: Menge von disjunkten Konstanten

$$Apfel = \{Boskoop, Cox, Smith\}$$

$$Boskoop \neq Cox, Cox \neq Smith, Boskoop \neq Smith$$

- Genau drei unterschiedliche Konstanten
- Funktion mit Wertebereich *Apfel* muss drei Fälle unterscheiden
- Beispiel: $preis : Apfel \to \mathbb{N}$ mit

$$preis(a) = \begin{cases} 55 & a = Boskoop \\ 60 & a = Cox \\ 50 & a = Smith \end{cases}$$

Aufzählung und Fallunterscheidung in Haskell

- Definition

```
data Apfel = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der Konstruktoren Boskoop :: Apfel als Konstanten
  - Großschreibung der Konstruktoren

- Fallunterscheidung:

```
apreis :: Apfel → Int
apreis a = case a of
  Boskoop → 55
  CoxOrange → 60
  GrannySmith → 50
```

Aufzählung und Fallunterscheidung in Haskell

- Definition

```
data Apfel = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der Konstruktoren Boskoop :: Apfel als Konstanten
  - Großschreibung der Konstruktoren

- Fallunterscheidung:

```
apreis :: Apfel → Int
apreis a = case a of
  Boskoop → 55
  CoxOrange → 60
  GrannySmith → 50
```

```
data Farbe = Rot | Grn
farbe :: Apfel→ Farbe
farbe d =
 case d of
  GrannySmith → Grn
  _ → Rot
```

Fallunterscheidung in der Funktionsdefinition

- Abkürzende Schreibweisen (syntaktischer Zucker):

$$\begin{array}{l} f\ c_1 == e_1 \\ \ldots \\ f\ c_n == e_n \end{array} \quad \longrightarrow \quad f\ x == \text{case}\ x\ \text{of}\ \begin{array}{l} c_1 \ \rightarrow\ e_1, \\ \ldots \\ c_n \ \rightarrow\ e_n \end{array}$$

- Damit:

```
apreis :: Apfel→ Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50
```

Der einfachste Aufzählungstyp

- Einfachste Aufzählung: Wahrheitswerte

$$Bool = \{False, True\}$$

  - Genau zwei unterschiedliche Werte

- Definition von Funktionen:
  - Wertetabellen sind explizite Fallunterscheidungen

$\wedge$	*true*	*false*
true	*true*	*false*
false	*false*	*false*

$$\begin{array}{lcccl} true & \wedge & true & = & true \\ true & \wedge & false & = & false \\ false & \wedge & true & = & false \\ false & \wedge & false & = & false \end{array}$$

Wahrheitswerte: Bool

- Vordefiniert als

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

- Vordefinierte Funktionen:

```
not  :: Bool → Bool            -- Negation
(&&) :: Bool → Bool → Bool     -- Konjunktion
(||) :: Bool → Bool → Bool     -- Disjunktion
```

- Konjunktion definiert als

```
a && b = case a of False → False
                   True  → b
```

- &&, || sind rechts nicht strikt
  - `1 == 0 && div 1 0 == 0` → `False`
- **if _ then _ else _** als syntaktischer Zucker:

$$\text{if } b \text{ then } p \text{ else } q \longrightarrow \text{case } b \text{ of } \begin{array}{lcl} \text{True} & \rightarrow & p \\ \text{False} & \rightarrow & q \end{array}$$

Produkte

- Konstruktoren können Argumente haben
- Beispiel: Ein Datum besteht aus Tag, Monat, Jahr
- Mathematisch: Produkt (Tupel)

$$
\begin{aligned}
Date &= \{Date\ (n,\ m,\ y) \mid n \in \mathbb{N}, m \in Month, y \in \mathbb{N}\} \\
Month &= \{Jan, Feb, Mar, \ldots\}
\end{aligned}
$$

- Funktionsdefinition:
  - Konstruktorargumente sind gebundene Variablen

$$
\begin{aligned}
year(D(n, m, y)) &= y \\
day(D(n, m, y)) &= n
\end{aligned}
$$

  - Bei der Auswertung wird gebundene Variable durch konkretes Argument ersetzt

Produkte in Haskell

- Konstruktoren mit Argumenten:

```
data Date  = Date Int Month Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
```

- Beispielwerte:

```
today     = Date 25 Oct 2016
bloomsday = Date 16 Jun 1904
```

Produkte in Haskell

- Konstruktoren mit Argumenten:

```
data Date  = Date Int Month Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
```

- Beispielwerte:

```
today     = Date 25 Oct 2016
bloomsday = Date 16 Jun 1904
```

- Über Fallunterscheidung Zugriff auf Argumente der Konstruktoren:

```
day  :: Date→ Int
year :: Date→ Int
day  d = case d of Date t m y→ t
year (Date _ _ y) = y
```

Beispiel: Tag im Jahr

- Tag im Jahr: Tag im laufenden Monat plus Summe der Anzahl der Tage der vorherigen Monate

```
yearDay :: Date→ Int
yearDay (Date d m y) = d + sumPrevMonths m where
  sumPrevMonths :: Month→ Int
  sumPrevMonths Jan = 0
  sumPrevMonths m   = daysInMonth (prev m) y +
                      sumPrevMonths (prev m)
```

- Tage im Monat benötigt Jahr als Argument (Schaltjahr!)

```
daysInMonth :: Month→ Int→ Int
```

```
prev :: Month→ Month
```

- Schaltjahr: Gregorianischer Kalender

```
leapyear :: Int→ Bool
leapyear y = if mod y 100 == 0 then mod y 400 == 0
             else mod y 4 == 0
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

```
data Kaese = Gouda | Appenzeller
```

```
kpreis :: Kaese → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

```haskell
data Kaese = Gouda | Appenzeller
```

```haskell
kpreis :: Kaese → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

- Alle Artikel:

```haskell
data Artikel =
    Apfel Apfel    | Eier
  | Kaese Kaese    | Schinken
  | Salami         | Milch Bool
```

Beispiel: Produkte in Bob's Shoppe

- Mengenangaben:

```
data Menge = Stueck Int   | Gramm  Int
           | Kilo  Double | Liter  Double
```

- Der Preis und seine Berechnung:

```
data Preis = Cent Int | Ungueltig
```

```
preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n)  = Cent (n* apreis a)
preis Eier (Stueck n)       = Cent (n* 20)
preis (Kaese k)(Kilo kg)    = Cent (round(kg*
                                       kpreis k))
preis Schinken (Gramm g)    = Cent (g/100* 199)
preis Salami (Gramm g)      = Cent (g/100* 159)
preis (Milch bio) (Liter l) =
  Cent (round (l* if not bio then 69 else 119))
preis _ _                   = Ungueltig
```

Auswertung der Fallunterscheidung

- Argument der Fallunterscheidung wird nur soweit nötig ausgewertet
- Beispiel:

```
data Foo = Foo Int | Bar

f :: Foo→ Int
f foo = case foo of Foo i → i; Bar → 0

g :: Foo→ Int
g foo = case foo of Foo i → 9; Bar → 0
```

- Auswertungen:

```
            f Bar  →  0
f (Foo undefined)  →  *** Exception: undefined
            g Bar  →  0
g (Foo undefined)  →  9
```

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{array}{lll} \text{data T} & = & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \mid & C_2\ t_{2,1} \ldots t_{2,k_2} \\ & & \ldots \\ & \mid & C_n\ t_{n,1} \ldots t_{n,k_n} \end{array}$$

1. Konstruktoren $C_1, \ldots, C_n$ sind disjunkt:
$$C_i\ x_1 \ldots x_n = C_j\ y_1 \ldots y_m \Longrightarrow i = j$$
2. Konstruktoren sind injektiv:
$$C\ x_1 \ldots x_n = C\ y_1 \ldots y_n \Longrightarrow x_i = y_i$$
3. Konstruktoren erzeugen den Datentyp:
$$\forall x \in T.\ x = C_i\ y_1 \ldots y_m$$

Diese Eigenschaften machen Fallunterscheidung möglich.

Rekursion? $\longrightarrow$ Nächste Vorlesung!

Zusammenfassung

- Striktheit
  - Haskell ist spezifiziert als nicht-strikt
- Datentypen und Funktionsdefinition dual
  - Aufzählungen — Fallunterscheidung
  - Produkte — Projektion
- Algebraische Datentypen
  - Drei wesentliche Eigenschaften der Konstruktoren
- Nächste Vorlesung: Rekursive Datentypen