

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 3 vom 30.10.2018: Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- ▶ Übungsbetrieb diese Woche
 - ▶ Übungsblatt mit 5 Punkten, Bearbeitungszeit bis Mo 12:00

Organisatorisches

- ▶ Übungsbetrieb diese Woche
 - ▶ Übungsblatt mit 5 Punkten, Bearbeitungszeit bis Mo 12:00
- ▶ Termine für E-Klausuren:
 - ▶ Probeklausur: vor Weihnachten
 - ▶ **Hauptklausur**: 08.03.2018 10:00 – 14:15
 - ▶ Wiederholungsklausur: 09.04. oder 11.04. (zweite Semesterwoche)

Fahrplan

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

Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
 - ▶ *Bool* statt `Int`, Namen statt RGB-Codes, ...

Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
 - ▶ *Bool* statt *Int*, Namen statt RGB-Codes, ...
- ▶ **Bessere** Programme (verständlicher und wartbarer)

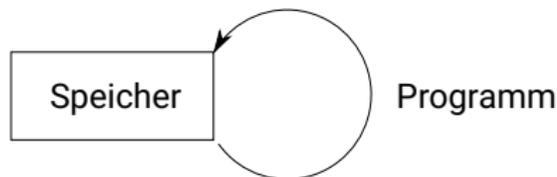
Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
 - ▶ *Bool* statt *Int*, Namen statt RGB-Codes, ...
- ▶ **Bessere** Programme (verständlicher und wartbarer)
- ▶ Datentypen haben **wohlverstandene algebraische Eigenschaften**

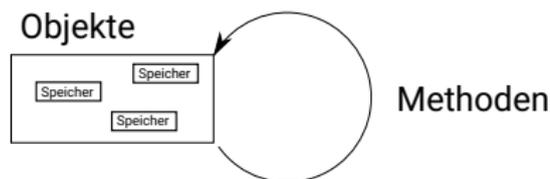
Datentypen als Modellierungskonstrukt

Programme **manipulieren** ein **Modell** der Umwelt:

▶ Imperative Sicht:



▶ Objektorientierte Sicht:



▶ Funktionale Sicht:



Das Modell besteht aus Datentypen.

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

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

$$\text{Boskoop} \neq \text{Cox}, \text{Cox} \neq \text{Smith}, \text{Boskoop} \neq \text{Smith}$$

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

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

Aufzählung und Fallunterscheidung in Haskell

▶ Definition

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

- ▶ Implizite Deklaration der **Konstruktoren** Boskoop :: Apfel als Konstanten
- ▶ **Großschreibung** der Konstruktoren und Typsen

▶ 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 und Typsen

► 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**):

$$\begin{array}{ccc} f\ c_1 == e_1 & & f\ x == \text{case } x \text{ of } c_1 \rightarrow e_1 \\ \dots & \longrightarrow & \dots \\ f\ c_n == e_n & & c_n \rightarrow e_n \end{array}$$

- ▶ Damit:

```
apreis :: Apfelsorte → 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	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

$$\begin{aligned} true \wedge true &= true \\ true \wedge false &= false \\ false \wedge true &= false \\ false \wedge false &= false \end{aligned}$$

Wahrheitswerte: Bool

- ▶ **Vordefiniert** als

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

- ▶ Vordefinierte **Funktionen**:

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

- ▶ **if _ then _ else _** als syntaktischer Zucker:

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

Striktheit Revisited

- ▶ **Konjunktion** definiert als

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

- ▶ Alternative Definition als Wahrheitstabelle:

```
and :: Bool → Bool → Bool
and True  True  = True
and True  False = False
and False True  = False
and False False = False
```

Unterschied?

Striktheit Revisited

- ▶ **Konjunktion** definiert als

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

- ▶ Alternative Definition als Wahrheitstabelle:

```
and :: Bool → Bool → Bool
and True  True  = True
and True  False = False
and False True  = False
and False False = False
```

Unterschied?

- ▶ Erste Definition ist **nicht-strikt** im zweiten Argument.
- ▶ Merke: wir können Striktheit von Funktionen (ungewollt) **erzwingen**

Produkte

- ▶ Konstruktoren können **Argumente** haben
- ▶ Beispiel: Ein **RGB-Wert** besteht aus drei Werten

- ▶ Mathematisch: Produkt (Tripel)

$$\text{Colour} = \{(r, g, b) \mid r \in \mathbb{N}, g \in \mathbb{N}, b \in \mathbb{N}\}$$

- ▶ In Haskell: Konstruktoren mit **Argumenten**

```
data Colour = RGB Int Int Int
```

- ▶ Beispielwerte:

```
yellow :: Colour  
yellow = RGB 255 255 0    — 0xFFFF00
```

```
violet :: Colour  
violet = RGB 238 130 238 — 0xEE82EE
```

Funktionsdefinition auf Produkten

▶ Funktionsdefinition:

- ▶ Konstruktorargumente sind **gebundene** Variablen
- ▶ Wird bei der **Auswertung** durch konkretes Argument ersetzt
- ▶ Kann mit Fallunterscheidung kombiniert werden

▶ Beispiele:

```
red :: Colour → Int
red (RGB r _ _) = r
```

```
green :: Colour → Int
green (RGB _ g _) = g
```

▶ Beispielauswertungen

```
red yellow ~> 255
green violet ~> 130
```

Beispiel: Produkte in Bob's Shoppe

- ▶ Käsesorten und deren Preise:

```
data Kaesesorte = Gouda | Appenzeller
```

```
kpreis :: Kaesesorte → Int
```

```
kpreis Gouda = 1450
```

```
kpreis Appenzeller = 2270
```

Beispiel: Produkte in Bob's Shoppe

- ▶ Käsesorten und deren Preise:

```
data Kaesesorte = Gouda | Appenzeller
```

```
kpreis :: Kaesesorte → Int
```

```
kpreis Gouda = 1450
```

```
kpreis Appenzeller = 2270
```

- ▶ Alle Artikel:

```
data Artikel =
```

```
    Apfel Apfelsorte | Eier  
    | Kaese Kaesesorte | Schinken  
    | Salami           | Milch Bio
```

```
data Bio = Bio | Konv
```

Beispiel: Produkte in Bob's Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

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

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (error ..)

Beispiel: Produkte in Bob's Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

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

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
- ▶ Ausnahmebehandlung **nicht referentiell transparent**

Beispiel: Produkte in Bob's Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

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

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
- ▶ Ausnahmebehandlung **nicht referentiell transparent**
- ▶ Könnten spezielle Werte (0 oder -1) zurückgeben

Beispiel: Produkte in Bob's Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

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

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
 - ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
 - ▶ Ausnahmebehandlung **nicht referentiell transparent**
 - ▶ Könnten spezielle Werte (0 oder -1) zurückgeben
- ▶ Besser: Ergebnis als Datentyp mit explizitem Fehler (**Reifikation**):

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

Beispiel: Produkte in Bob's Shoppe

- ▶ 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)(Gramm g) = Cent (div (g * kpreis k) 1000)
```

```
preis Schinken (Gramm g) = Cent (div (g * 199) 100)
```

```
preis Salami (Gramm g) = Cent (div (g * 159) 100)
```

```
preis (Milch bio) (Liter l) =
```

```
Cent (round (l * case bio of Bio → 119; Konv → 69))
```

```
preis _ _ = Ungueltig
```

Auswertung der Fallunterscheidung

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

```
f :: Preis → Int  
f p = case p of Cent i → i; Ungueltig → 0
```

```
g :: Preis → Int  
g p = case p of Cent i → 1; Ungueltig → 0
```

<

```
add :: Preis → Preis → Preis  
add (Cent i) (Cent j) = Cent (i+j)  
add _ _ = Ungueltig
```

Auswertung der Fallunterscheidung

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

```
f :: Preis → Int
f p = case p of Cent i → i; Ungueltig → 0
```

```
g :: Preis → Int
g p = case p of Cent i → 1; Ungueltig → 0
```

<

```
add :: Preis → Preis → Preis
add (Cent i) (Cent j) = Cent (i + j)
add _ _ = Ungueltig
```

- ▶ Auswertungen:

```
f (Cent undefined) ~> *** Exception: Prelude.undefined
```

```
g (Cent undefined) ~> 1
```

```
f (Cent (g (Cent undefined))) ~> 1
```

```
g (add (Cent 1) (Cent undefined)) ~> 1
```

```
f (add (Cent undefined) Ungueltig) ~> 0
```

Der Allgemeine Fall: Algebraische Datentypen

$$\text{data } T = \begin{array}{l} C_1 \\ | \\ C_2 \\ \vdots \\ | \\ C_n \end{array}$$

► Aufzählungen

Der Allgemeine Fall: Algebraische Datentypen

data T = C₁ t_{1,1} ... t_{1,k₁}

- ▶ **Aufzählungen**
- ▶ Konstrukturen mit **einem** oder **mehreren** Argumenten (Produkte)

Der Allgemeine Fall: Algebraische Datentypen

$$\begin{array}{l} \text{data } T = \\ \quad | \quad C_1 \ t_{1,1} \ \dots \ t_{1,k_1} \\ \quad | \quad C_2 \ t_{2,1} \ \dots \ t_{2,k_2} \\ \quad | \quad \vdots \\ \quad | \quad C_n \ t_{n,1} \ \dots \ t_{n,k_n} \end{array}$$

- ▶ **Aufzählungen**
- ▶ Konstrukturen mit **einem** oder **mehreren** Argumenten (Produkte)
- ▶ Der allgemeine Fall: **mehrere** Konstrukturen

Eigenschaften algebraischer Datentypen

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ | C_2 t_{2,1} \dots t_{2,k_2} \\ \vdots \\ | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

Drei Eigenschaften eines algebraischen Datentypen

① Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$$

Eigenschaften algebraischer Datentypen

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ | C_2 t_{2,1} \dots t_{2,k_2} \\ \vdots \\ | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

Drei Eigenschaften eines algebraischen Datentypen

- ① Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$$

- ② Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \implies x_i = y_i$$

Eigenschaften algebraischer Datentypen

$$\begin{array}{l} \text{data } T = \\ \quad | \quad C_1 \ t_{1,1} \dots t_{1,k_1} \\ \quad | \quad C_2 \ t_{2,1} \dots t_{2,k_2} \\ \quad | \quad \vdots \\ \quad | \quad C_n \ t_{n,1} \dots t_{n,k_n} \end{array}$$

Drei Eigenschaften eines algebraischen Datentypen

- ① Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i \ x_1 \dots x_n = C_j \ y_1 \dots y_m \implies i = j$$

- ② Konstruktoren sind **injektiv**:

$$C \ x_1 \dots x_n = C \ y_1 \dots y_n \implies x_i = y_i$$

- ③ Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i \ y_1 \dots y_m$$

Eigenschaften algebraischer Datentypen

$$\begin{array}{l} \text{data } T = \\ \quad | \quad C_1 \ t_{1,1} \dots t_{1,k_1} \\ \quad | \quad C_2 \ t_{2,1} \dots t_{2,k_2} \\ \quad | \quad \vdots \\ \quad | \quad C_n \ t_{n,1} \dots t_{n,k_n} \end{array}$$

Drei Eigenschaften eines algebraischen Datentypen

- ① Konstruktoren C_1, \dots, C_n sind **disjunkt**:

$$C_i \ x_1 \dots x_n = C_j \ y_1 \dots y_m \implies i = j$$

- ② Konstruktoren sind **injektiv**:

$$C \ x_1 \dots x_n = C \ y_1 \dots y_n \implies x_i = y_i$$

- ③ Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i \ y_1 \dots y_m$$

Diese Eigenschaften machen **Fallunterscheidung** wohldefiniert.

Algebraische Datentypen: Nomenklatur

data $T = C_1 t_{1,1} \dots t_{1,k_1} \mid \dots \mid C_n t_{n,1} \dots t_{n,k_n}$

- ▶ C_j sind **Konstruktoren**

- ▶ **Immer** implizit definiert und deklariert

- ▶ **Selektoren** sind Funktionen $sel_{i,j}$:

$sel_{i,j} \quad \quad \quad :: T \rightarrow t_{i,k_i}$

$sel_{i,j} (C_i t_{i,1} \dots t_{i,k_i}) = t_{i,j}$

- ▶ Partiiell, linksinvers zu Konstruktor C_j

- ▶ **Können** implizit definiert und deklariert werden

- ▶ **Diskriminatoren** sind Funktionen dis_j :

$dis_j \quad \quad \quad :: T \rightarrow \text{Bool}$

$dis_j (C_i \dots) = \text{True}$

$dis_j _ = \text{False}$

- ▶ Definitionsbereichbereich des Selektors sel_j , **nie** implizit

Rekursive Algebraische Datentypen

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ \quad \vdots \\ \quad | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

- ▶ Der definierte Typ T kann **rechts** benutzt werden.
- ▶ Rekursive Datentypen definieren **unendlich große** Wertemengen.
- ▶ Modelliert **Aggregation** (Sammlung von Objekten).
- ▶ Funktionen werden durch **Rekursion** definiert.

Uncle Bob's Auld Time Grocery Shoppe Revisited

- ▶ Das **Lager** für Bob's Shoppe:
 - ▶ ist entweder leer,
 - ▶ oder es enthält einen Artikel und Menge, und noch mehr

```
data Lager = LeeresLager  
           | Lager Artikel Menge Lager
```

Suchen im Lager

- ▶ Rekursive Suche (erste Version):

```
suche :: Artikel → Lager → Menge  
suche art LeeresLager = ???
```

Suchen im Lager

- ▶ Rekursive Suche (erste Version):

```
suche :: Artikel → Lager → Menge  
suche art LeeresLager = ???
```

- ▶ Modellierung des **Resultats**:

```
data Resultat = Gefunden Menge | NichtGefunden
```

- ▶ Damit rekursive **Suche**:

```
suche :: Artikel → Lager → Resultat  
suche art (Lager lart m l)  
  | art == lart = Gefunden m  
  | otherwise   = suche art l  
suche art LeeresLager = NichtGefunden
```

Einlagern

- ▶ Signatur:

```
einlagern :: Artikel → Menge → Lager → Lager
```

- ▶ Erste Version:

```
einlagern a m l = Lager a m l
```

- ▶ Mengen sollen **aggregiert** werden (35l Milch + 20l Milch = 55l Milch)
- ▶ Dazu Hilfsfunktion:

```
addiere (Stueck i) (Stueck j) = Stueck (i+j)  
addiere (Gramm g) (Gramm h) = Gramm (g+h)  
addiere (Liter l) (Liter m) = Liter (l+m)  
addiere m n = error ("addiere:␣" ++ show m ++ "␣und␣" ++ show n)
```

Einlagern

- ▶ Damit einlagern:

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m LeeresLager = Lager a m LeeresLager
einlagern a m (Lager al ml l)
  | a == al    = Lager a (addiere m ml) l
  | otherwise = Lager al ml (einlagern a m l)
```

- ▶ Problem:

Einlagern

- ▶ Damit einlagern:

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m LeeresLager = Lager a m LeeresLager
einlagern a m (Lager al ml l)
  | a == al    = Lager a (addiere m ml) l
  | otherwise = Lager al ml (einlagern a m l)
```

- ▶ Problem: **Falsche Mengenangaben**
 - ▶ Bspw. einlagern Eier (Liter 3.0) l
 - ▶ Erzeugen Laufzeitfehler in addiere
- ▶ Lösung: eigentliche Funktion einlagern wird als **lokale Funktion** versteckt, und nur mit gültiger Mengenangabe aufgerufen.

Einlagern

- ▶ Lösung: eigentliche Funktion einlagern wird als **lokale Funktion** versteckt, und nur mit gültiger Mengenangabe aufgerufen.

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m l =
  let einlagern' a m LeeresLager = Lager a m LeeresLager
      einlagern' a m (Lager al ml l)
      | a == al    = Lager a (addiere m ml) l
      | otherwise = Lager al ml (einlagern' a m l)
  in case preis a m of
    Ungueltig → l
    _        → einlagern' a m l
```

Einkaufen und bezahlen

- ▶ Wir brauchen einen **Einkaufswagen**:

```
data Einkaufswagen = LeererWagen
                    | Einkauf Artikel Menge Einkaufswagen
```

- ▶ Artikel einkaufen:

```
einkauf :: Artikel → Menge → Einkaufswagen → Einkaufswagen
einkauf a m e =
  case preis a m of
    Ungueltig → e
    _ → Einkauf a m e
```

- ▶ Auch hier: ungültige Mengenangaben erkennen
- ▶ Es wird **nicht** aggregiert

Beispiel: Kassenbon

kassenbon :: Einkaufswagen → String

Ausgabe:

* Bob's Aulde Grocery Shoppe *

Artikel	Menge	Preis
Kaese Appenzeller	378 g.	8.58 EU
Schinken	50 g.	0.99 EU
Milch Bio	1.0 l.	1.19 EU
Schinken	50 g.	0.99 EU
Apfel Boskoop	3 St	1.65 EU
=====		
Summe:		13.40 EU

Unveränderlicher
Kopf

Ausgabe von Artikel
und Menge (rekur-
siv)

Ausgabe von kasse

Kassenbon: Implementation

► Kernfunktion:

```
artikel :: Einkaufswagen → String
artikel LeererWagen = ""
artikel (Einkauf a m e) =
  formatL 20 (show a) ++
  formatR 7  (menge m) ++
  formatR 10 (showEuro (cent a m)) ++ "\n" ++ artikel e
```

► Hilfsfunktionen:

```
formatL :: Int → String → String
```

```
formatR :: Int → String → String
```

```
showEuro :: Int → String
```

Beispiel: Zeichenketten selbstgemacht

- ▶ Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** c und eine weitere **Zeichenkette** xs

```
data String = Empty
            | Char :+ String
```

- ▶ **Lineare** Rekursion
 - ▶ Genau ein rekursiver Aufruf
- ▶ Haskell-Merkwürdigkeit #237:
 - ▶ Die Namen von Operator-Konstruktoren müssen mit einem `:` beginnen.

Rekursiver Typ, rekursive Definition

- ▶ Typisches Muster: **Fallunterscheidung**
 - ▶ Ein Fall pro Konstruktor
- ▶ Hier:
 - ▶ Leere Zeichenkette
 - ▶ Nichtleere Zeichenkette

Funktionen auf Zeichenketten

- ▶ Länge:

```
length :: String → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

Funktionen auf Zeichenketten

► Länge:

```
length :: String → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

► Verkettung:

```
(++) :: String → String → String
Empty ++ t = t
(c :+ s) ++ t = c :+ (s ++ t)
```

Funktionen auf Zeichenketten

► Länge:

```
length :: String → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

► Verkettung:

```
(++) :: String → String → String
Empty ++ t = t
(c :+ s) ++ t = c :+ (s ++ t)
```

► Umdrehen:

```
rev :: String → String
rev Empty    = Empty
rev (c :+ t) = rev t ++ (c :+ Empty)
```

Zusammenfassung

- ▶ Algebraische Datentypen: Aufzählungen, Produkte, rekursive Datentypen
- ▶ Rekursive Datentypen sind **unendlich** (induktiv)
- ▶ Funktionen werden durch **Fallunterscheidung** und **Rekursion** definiert
- ▶ Fallbeispiele: Bob's Shoppe, Zeichenketten