Praktische Informatik 3: Funktionale Programmierung

Vorlesung 3 vom 16.11.2020: Algebraische Datentypen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2020/21

Fahrplan

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

Inhalt und Lernziele

- Algebraische Datentypen:
  - Aufzählungen
  - Produkte
  - Rekursive Datentypen

Lernziel

Wir wissen, was algebraische Datentypen sind. Wir können mit ihnen modellieren, wir kennen ihre Eigenschaften, und können auf ihnen Funktionen definieren.

I. Datentypen

Warum Datentypen?

- Immer nur `Int` ist auch langweilig ...

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)

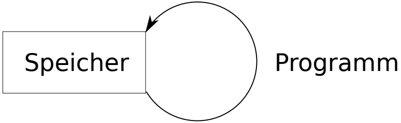
Warum Datentypen?

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

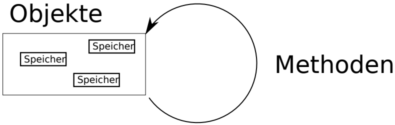
Datentypen als Modellierungskonstrukt

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

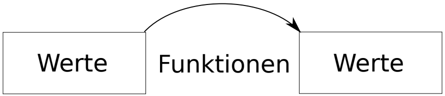
- Imperative Sicht:



- 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

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

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

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

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

Aufzählung und Fallunterscheidung in Haskell

- **Definition**

```haskell
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der **Konstruktoren** `Boskoop :: Apfelsorte` als Konstanten
  - **Großschreibung** der Konstruktoren und Typen

- **Fallunterscheidung**:

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

Aufzählung und Fallunterscheidung in Haskell

- **Definition**

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der **Konstruktoren** `Boskoop :: Apfelsorte` als Konstanten
  - **Großschreibung** der Konstruktoren und Typen

- **Fallunterscheidung**:

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

```
data Farbe = Rot | Gruen
farbe :: Apfelsorte→ Farbe
farbe d =
 case d of
  GrannySmith → Gruen
  _ → Rot
```

Fallunterscheidung in der Funktionsdefinition

- Abkürzende Schreibweisen (**syntaktischer Zucker**):

$$
\begin{array}{l} f\ c_1 = e_1 \\ \ldots \\ f\ c_n = e_n \end{array}
\quad\longrightarrow\quad
f\ x = \mathtt{case}\ x\ \mathtt{of}\ \begin{array}{lcl} c_1 & \to & e_1 \\ \ldots & & \\ c_n & \to & e_n \end{array}
$$

- 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

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

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

Wahrheitswerte: `Bool`

- **Vordefiniert** als

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

- Vordefinierte **Funktionen**:

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

- `if _ then _ else _` als syntaktischer Zucker:

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

Striktheit Revisited

- **Konjunktion** definiert als

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

- Alternative Definition als Wahrheitstabelle:

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

Übung 3.1: Kurze Frage: Gibt es einen Unterschied zwischen den beiden?

DEMO

Striktheit Revisited

- **Konjunktion** definiert als

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

- Alternative Definition als Wahrheitstabelle:

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

Übung 3.1: Kurze Frage: Gibt es einen Unterschied zwischen den beiden?

Lösung:

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

II. Produkte

Produkte

- Konstruktoren können **Argumente** haben
- Beispiel: Ein **RGB-Wert** besteht aus drei Werten
- Mathematisch: Produkt (Tripel)

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

- In Haskell: Konstruktoren mit **Argumenten**

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

  - Beispielwerte:

```haskell
yellow :: Colour
yellow = RGB 255 255 0      -- 0xFFFF00
```

```haskell
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
```

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
```

```
adjust :: Colour→ Float→ Colour
adjust (RGB r g b) f = RGB (conv r) (conv g) (conv b) where
  conv colour = min (round (fromIntegral colour∗ f)) 255
```

Beispiel: Bob's Auld-Time Grocery Shoppe

- Käsesorten und deren Preise:

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

```haskell
kpreis :: Kaesesorte → Int
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

Beispiel: Bob's Auld-Time Grocery 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 | Chemie
```

Beispiel: Bob's Auld-Time Grocery Shoppe

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

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

- Preisberechung

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

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

Beispiel: Bob's Auld-Time Grocery Shoppe

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

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

- Preisberechung

```haskell
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: Bob's Auld-Time Grocery Shoppe

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

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

- Preisberechung

```haskell
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: Bob's Auld-Time Grocery Shoppe

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

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

- Preisberechung

```haskell
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**):

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

Beispiel: Bob's Auld-Time Grocery 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; Chemie → 69))
preis _ _ = Ungueltig
```

DEMO

Jetzt seit ihr dran

Übung 3.1: Refaktorierungen

Was passiert bei folgenden Änderungen an `preis` :

1. Vorletzte Zeile zu `Cent (round (l* case bio of Chemie → 69; Bio→ 119`
2. Vorletzte Zeile zu `Cent (round (l* case bio of Bio→ 119; _ → 69`
3. Vertauschung der zwei vorletzten und letzten Zeile.

Jetzt seit ihr dran

Übung 3.1: Refaktorierungen

Was passiert bei folgenden Änderungen an `preis` :

1. Vorletzte Zeile zu `Cent (round (l* case bio of Chemie → 69; Bio→ 119`
2. Vorletzte Zeile zu `Cent (round (l* case bio of Bio→ 119; _ → 69`
3. Vertauschung der zwei vorletzten und letzten Zeile.

Lösung:

1. Nichts, unterschiedliche Fälle können getauscht werden.
2. Nichts, da _ nur `Chemie` sein kann
3. Der letzte Fall wird nie aufgerufen — der Milchpreis wäre `Ungueltig`

III. Algebraische Datentypen

Der Allgemeine Fall: Algebraische Datentypen

$$
\begin{array}{lll}
\texttt{data T} = & \texttt{C}_1 \\
| & \texttt{C}_2 \\
& \vdots \\
| & \texttt{C}_n
\end{array}
$$

- **Aufzählungen**

Der Allgemeine Fall: Algebraische Datentypen

$\texttt{data T} = \ \ \texttt{C}_1\ \texttt{t}_{1,1}\ \ldots\ \texttt{t}_{1,k_1}$

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

Der Allgemeine Fall: Algebraische Datentypen

$$
\begin{array}{llll}
\texttt{data T} & = & \texttt{C}_1\ \texttt{t}_{1,1}\ \ldots\ \texttt{t}_{1,k_1} \\
& \mid & \texttt{C}_2\ \texttt{t}_{2,1}\ \ldots\ \texttt{t}_{2,k_2} \\
& & \vdots \\
& \mid & \texttt{C}_n\ \texttt{t}_{n,1}\ \ldots\ \texttt{t}_{n,k_n}
\end{array}
$$

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

Eigenschaften algebraischer Datentypen

$$
\begin{array}{llll}
\texttt{data T} & = & \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \\
& \mid & \texttt{C}_2\ \texttt{t}_{2,1} \ldots \texttt{t}_{2,k_2} \\
& & \vdots \\
& \mid & \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_n$ sind **disjunkt**:

$$\texttt{C}_i\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}_j\ \texttt{y}_1 \ldots \texttt{y}_m \Longrightarrow i = j$$

Eigenschaften algebraischer Datentypen

$$
\begin{aligned}
\mathtt{data\ T} = \; & \mathtt{C}_1\ \mathtt{t}_{1,1} \ldots \mathtt{t}_{1,k_1} \\
\mid \; & \mathtt{C}_2\ \mathtt{t}_{2,1} \ldots \mathtt{t}_{2,k_2} \\
& \vdots \\
\mid \; & \mathtt{C}_n\ \mathtt{t}_{n,1} \ldots \mathtt{t}_{n,k_n}
\end{aligned}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\mathtt{C}_1, \ldots, \mathtt{C}_n$ sind **disjunkt**:

$$\mathtt{C}_i\ \mathtt{x}_1 \ldots \mathtt{x}_n = \mathtt{C}_j\ \mathtt{y}_1 \ldots \mathtt{y}_m \Longrightarrow i = j$$

2. Konstruktoren sind **injektiv**:

$$\mathtt{C}\ \mathtt{x}_1 \ldots \mathtt{x}_n = \mathtt{C}\ \mathtt{y}_1 \ldots \mathtt{y}_n \Longrightarrow \mathtt{x}_i = \mathtt{y}_i$$

Eigenschaften algebraischer Datentypen

$$
\begin{array}{lll}
\texttt{data T} = & \texttt{C}_1\ \texttt{t}_{1,1} \dots \texttt{t}_{1,k_1} \\
\mid & \texttt{C}_2\ \texttt{t}_{2,1} \dots \texttt{t}_{2,k_2} \\
 & \vdots \\
\mid & \texttt{C}_n\ \texttt{t}_{n,1} \dots \texttt{t}_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\texttt{C}_1, \dots, \texttt{C}_n$ sind **disjunkt**:
$$\texttt{C}_i\ \texttt{x}_1 \dots \texttt{x}_n = \texttt{C}_j\ \texttt{y}_1 \dots \texttt{y}_m \Longrightarrow i = j$$
2. Konstruktoren sind **injektiv**:
$$\texttt{C}\ \texttt{x}_1 \dots \texttt{x}_n = \texttt{C}\ \texttt{y}_1 \dots \texttt{y}_n \Longrightarrow \texttt{x}_i = \texttt{y}_i$$
3. Konstruktoren **erzeugen** den Datentyp:
$$\forall x \in T.\, x = \texttt{C}_i\ \texttt{y}_1 \dots \texttt{y}_m$$

Eigenschaften algebraischer Datentypen

$$
\begin{array}{lll}
\texttt{data T} = & \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \\
| & \texttt{C}_2\ \texttt{t}_{2,1} \ldots \texttt{t}_{2,k_2} \\
 & \vdots \\
| & \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_n$ sind **disjunkt**:
$$\texttt{C}_i\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}_j\ \texttt{y}_1 \ldots \texttt{y}_m \Longrightarrow i = j$$
2. Konstruktoren sind **injektiv**:
$$\texttt{C}\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}\ \texttt{y}_1 \ldots \texttt{y}_n \Longrightarrow \texttt{x}_i = \texttt{y}_i$$
3. Konstruktoren **erzeugen** den Datentyp:
$$\forall x \in T.\, x = \texttt{C}_i\ \texttt{y}_1 \ldots \texttt{y}_m$$

Diese Eigenschaften machen **Fallunterscheidung** wohldefiniert.

Algebraische Datentypen: Nomenklatur

$$\texttt{data T} = \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \mid \cdots \mid \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n}$$

- $\texttt{C}_i$ sind **Konstruktoren**
  - **Immer** implizit definiert und deklariert
- **Selektoren** sind Funktionen $\texttt{sel}_{i,j}$:

  $$\begin{array}{ll} \texttt{sel}_{i,j} & :: \texttt{T} \to \texttt{t}_{i,k_i} \\ \texttt{sel}_{i,j}\ (\texttt{C}_i\ \texttt{t}_{i,1} \ldots\ \texttt{t}_{i,k_i}) & = \texttt{t}_{i,j} \end{array}$$

  - Partiell, linksinvers zu Konstruktor $\texttt{C}_i$
  - **Können** implizit definiert und deklariert werden
- **Diskriminatoren** sind Funktionen $\texttt{dis}_i$:

  $$\begin{array}{ll} \texttt{dis}_i & :: \texttt{T} \to \texttt{Bool} \\ \texttt{dis}_i\ (\texttt{C}_i \ldots) & = \texttt{True} \\ \texttt{dis}_i\ \_ & = \texttt{False} \end{array}$$

  - Definitionsbereichsbereich des Selektors $\texttt{sel}_i$, **nie** implizit

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 → 99; 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 → 99; Ungueltig → 0
```

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

- Argument von `Cent` wird in `f` ausgewertet, in `g` nicht
- Zweites Argument von `add` wird nicht immer ausgewertet

Rekursive Algebraische Datentypen

$$\begin{array}{lcl} \texttt{data T} & = & \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \\ & \vdots & \\ & | & \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n} \end{array}$$

- 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 **Einkaufskorb:**

```
data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb
```

- Artikel einkaufen:

```
einkauf :: Artikel→ Menge→ Einkaufskorb→ Einkaufskorb
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 :: Einkaufskorb→ String
```

Ausgabe:

```
** Bob’s Aulde-Time 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 (rekursiv)

Ausgabe von `kasse`

Kassenbon: Implementation

- Kernfunktion:

```
artikel :: Einkaufskorb→ String
artikel LeererKorb = ""
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
```

DEMO

Kurz zum Nachdenken

Übung 3.2: Zeichenketten

Wie könnten wohl Zeichenketten (`String`) definiert sein?

IV. Rekursive Datentypen

Beispiel: Zeichenketten selbstgemacht

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

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

- **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 :: MyString→ Int
length Empty      = 0
length (c :+ s) = 1+ length s
```

Funktionen auf Zeichenketten

- Länge:

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

- Verkettung:

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

Funktionen auf Zeichenketten

- Länge:

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

- Verkettung:

```haskell
(++) :: MyString → MyString → MyString
Empty ++ t  = t
(c :+ s) ++ t = c :+ (s++ t)
```

- Umdrehen:

```haskell
rev :: MyString → MyString
rev Empty    = Empty
rev (c :+ t) = rev t ++ (c :+ Empty)
```

DEMO

Zusammenfassung

- Algebraische Datentypen: Aufzählungen, Produkte, rekursive Datentypen
- Drei Schlüsseleigenschaften der Konstruktoren: **disjunkt**, **injektiv**, **erzeugend**
- Rekursive Datentypen sind **unendlich** (induktiv)
- Funktionen werden durch **Fallunterscheidung** und **Rekursion** definiert
- Fallbeispiele: Bob's Shoppe, Zeichenketten