

# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 3 vom 01.11.2016: Algebraische 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 Effizienzaspekte
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

- ▶ **Rekursive** Datentypen
  - ▶ Rekursive **Definition**
  - ▶ ... und wozu sie nützlich sind
  - ▶ Rekursive Datentypen in anderen Sprachen
  - ▶ Fallbeispiel: Labyrinth

# Algebraische Datentypen

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

- ▶ Aufzählungen

# Algebraische Datentypen

data T = C<sub>1</sub> t<sub>1,1</sub> ... t<sub>1,k<sub>1</sub></sub>

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

Heute: **Rekursion**

# 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

Heute: **Rekursion**

# Der Allgemeine Fall: Algebraische Datentypen

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

## 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

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

- ▶  $C_i$  sind **Konstruktoren**
  - ▶ Immer vordefiniert
- ▶ **Selektoren** sind Funktionen  $\text{sel}_{i,j}$ :
$$\text{sel}_{i,j} \quad \quad \quad :: T \rightarrow t_{i,k_i}$$
$$\text{sel}_{i,j} (C_i t_{i,1} \dots t_{i,k_i}) = t_{i,j}$$
  - ▶ Linksinvers zu Konstruktor  $C_i$ , partiell
  - ▶ **Können** vordefiniert werden (erweiterte Syntax der **data** Deklaration)
- ▶ **Diskriminatoren** sind Funktionen  $\text{dis}_j$ :
$$\text{dis}_j \quad \quad \quad :: T \rightarrow \text{Bool}$$
$$\text{dis}_j (C_i \dots) = \text{True}$$
$$\text{dis}_j \_ = \text{False}$$
  - ▶ Definitionsbereich des Selektors  $\text{sel}_j$ , **nie** vordefiniert



# Rekursive Datentypen

- ▶ 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 weiteres.

```
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

- ▶ 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)
```

- ▶ 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

- ▶ 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)
```

- ▶ 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**
  - ▶ z.B. einlagern Eier (Liter 3.0) l

## Einlagern (verbessert)

- ▶ 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
```

- ▶ Gesamtsumme berechnen:

```
kasse :: Einkaufswagen → Int  
kasse LeererWagen = 0  
kasse (Einkauf a m e) = cent a m + kasse e
```



# Beispiel: Kassenbon

kassenbon :: Einkaufswagen → String

Ausgabe:

Bob's Aulde Grocery Shoppe

Artikel	Menge	Preis
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:		4.82 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
```

# Rekursive Typen in imperativen Sprachen

# Rekursive Typen in Java

- ▶ Nachbildung durch Klassen, z.B. für Listen:

```
class List {  
    public List(Object el, List tl) {  
        this.elem= el;  
        this.next= tl;  
    }  
    public Object elem;  
    public List next;
```

- ▶ Länge (iterativ):

```
int length() {  
    int i= 0;  
    for (List cur= this; cur != null; cur= cur.next)  
        i++;  
    return i;  
}
```

# Rekursive Typen in C

- ▶ C: Produkte, Aufzählungen, keine rekursiven Typen
- ▶ Rekursion durch Zeiger

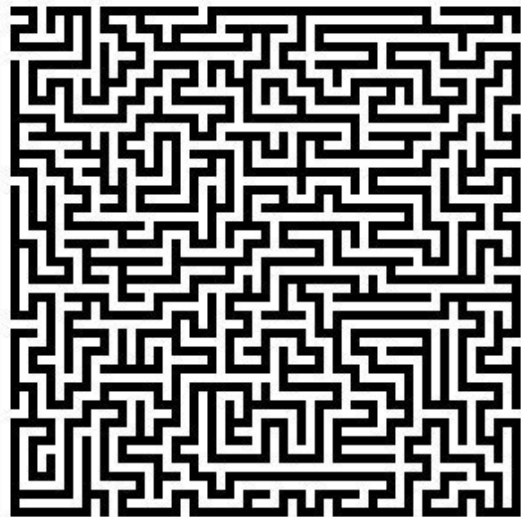
```
typedef struct list_t {  
    void *elem;  
    struct list_t *next;  
} *list;
```

- ▶ Konstruktoren **nutzerimplementiert**

```
list cons(void *hd, list tl)  
{ list l;  
  if ((l = (list)malloc(sizeof(struct list_t))) == NULL) {  
    printf("Out of memory\n"); exit(-1);  
  }  
  l->elem = hd; l->next = tl;  
  return l;  
}
```

# Fallbeispiel

## Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

# Modellierung eines Labyrinths

- ▶ Ein **gerichtetes** Labyrinth ist entweder
  - ▶ eine Sackgasse,
  - ▶ ein Weg, oder
  - ▶ eine Abzweigung in zwei Richtungen.

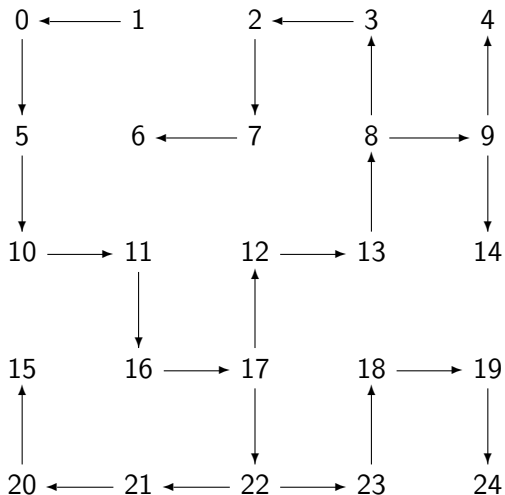
```
data Lab = Dead Id
      | Pass Id Lab
      | TJnc Id Lab Lab
```

- ▶ Ferner benötigt: eindeutige **Bezeichner** der Knoten

```
type Id = Integer
```



# Ein Labyrinth (zyklenfrei)



# Traversion des Labyrinths

- ▶ Ziel: **Pfad** zu einem gegebenen **Ziel** finden
  
- ▶ Benötigt **Pfade** und **Traversion**

```
data Path = Cons Id Path  
          | Mt
```

```
data Trav = Succ Path  
          | Fail
```

# Traversionsstrategie

- ▶ Geht von **zyklenfreien** Labyrinth aus
- ▶ An jedem Knoten prüfen, ob Ziel erreicht, ansonsten
  - ▶ an Sackgasse Fail
  - ▶ an Passagen weiterlaufen
  - ▶ an Kreuzungen Auswahl treffen
- ▶ Erfordert Propagation von Fail:

```
cons :: Id → Trav → Trav
```

```
select :: Trav → Trav → Trav
```

# Zyklenfreie Traversal

```
traverse1 :: Id → Lab → Trav
traverse1 t l
  | nid l == t = Succ (Cons (nid l) Mt)
  | otherwise = case l of
    Dead _ → Fail
    Pass i n → cons i (traverse1 t n)
    TJnc i n m → select (cons i (traverse1 t n))
                        (cons i (traverse1 t m))
```

# Zyklenfreie Traversal

```
traverse1 :: Id → Lab → Trav
traverse1 t l
  | nid l == t = Succ (Cons (nid l) Mt)
  | otherwise = case l of
    Dead _ → Fail
    Pass i n → cons i (traverse1 t n)
    TJnc i n m → select (cons i (traverse1 t n))
                        (cons i (traverse1 t m))
```

- ▶ Wie mit Zyklen umgehen?
- ▶ An jedem Knoten prüfen ob schon im Pfad enthalten

# Traversion mit Zyklen

- ▶ Veränderte **Strategie**: Pfad bis hierher übergeben
  - ▶ Pfad muss **hinten** erweitert werden.
- ▶ Wenn **aktueller** Knoten in bisherigen Pfad **enthalten** ist, Fail
- ▶ Ansonsten wie oben
- ▶ Neue Hilfsfunktionen:

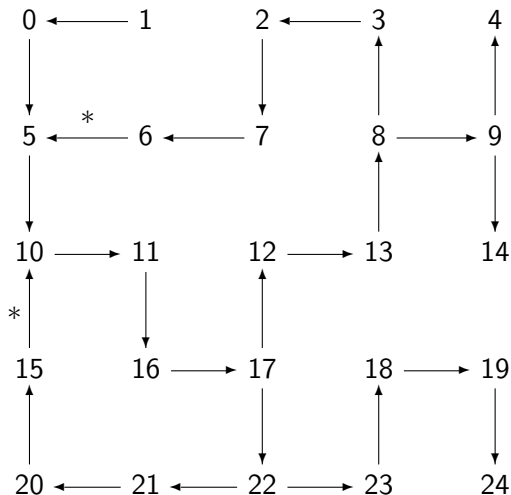
```
contains :: Id → Path → Bool
```

```
snoc :: Path → Id → Path
```

# Traversion mit Zyklen

```
traverse2 :: Id → Lab → Path → Trav
traverse2 t l p
  | nid l == t = Succ (snoc p (nid l))
  | contains (nid l) p = Fail
  | otherwise = case l of
    Dead _ → Fail
    Pass i n → traverse2 t n (snoc p i)
    TJnc i n m → select (traverse2 t n (snoc p i))
                        (traverse2 t m (snoc p i))
```

# Ein Labyrinth (mit Zyklen)





# Ungerichtete Labyrinth

- ▶ In einem **ungerichteten** Labyrinth haben Passagen keine Richtung.
  - ▶ Sackgassen haben einen Nachbarn,
  - ▶ eine Passage hat zwei Nachbarn,
  - ▶ und eine Abzweigung drei Nachbarn.

```
data Lab = Dead Id Lab
      | Pass Id Lab Lab
      | TJnc Id Lab Lab Lab
```

- ▶ Andere Datentypen und Hilfsfunktionen bleiben (*mutatis mutandis*)
- ▶ Jedes nicht-leere ungerichtete Labyrinth hat **Zyklen**.
- ▶ **Invariante** (nicht durch Typ garantiert)

# Traversal in ungerichteten Labyrinth

- ▶ Traversionsfunktion wie vorher

```
traverse3 :: Id → Lab → Path → Trav
traverse3 t l p
  | nid l == t = Succ (snoc p (nid l))
  | contains (nid l) p = Fail
  | otherwise = case l of
    Dead i n → traverse3 t n (snoc p i)
    Pass i n m → select (traverse3 t n (snoc p i))
                       (traverse3 t m (snoc p i))
    TJnc i n m k → select (traverse3 t n (snoc p i))
                          (select (traverse3 t m (snoc p i))
                                   (traverse3 t k (snoc p i)))
```

# Zusammenfassung Labyrinth

- ▶ Labyrinth  $\longrightarrow$  Graph oder Baum
- ▶ In Haskell: gleicher Datentyp
- ▶ Referenzen nicht **explizit** in Haskell
  - ▶ Keine **undefinierten** Referenzen (erhöhte **Programmsicherheit**)
  - ▶ Keine **Gleichheit** auf Referenzen
  - ▶ Gleichheit ist **immer** strukturell (oder **selbstdefiniert**)

# Beispiel: Zeichenketten selbstgemacht

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

```
data MyString = Empty  
              | Cons Char MyString
```

- ▶ **Lineare** Rekursion
  - ▶ Genau ein rekursiver Aufruf

# Rekursive Definition

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

# Funktionen auf Zeichenketten

- ▶ Länge:

```
len :: MyString → Int
len Empty      = 0
len (Cons c str) = 1 + len str
```

# Funktionen auf Zeichenketten

## ► Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str) = 1 + len str
```

## ► Verkettung:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

# Funktionen auf Zeichenketten

## ► Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str)  = 1 + len str
```

## ► Verkettung:

```
cat :: MyString → MyString → MyString
cat Empty t      = t
cat (Cons c s) t = Cons c (cat s t)
```

## ► Umkehrung:

```
rev :: MyString → MyString
rev Empty          = Empty
rev (Cons c t)    = cat (rev t) (Cons c Empty)
```



# Was haben wir gesehen?

- ▶ Strukturell **ähnliche** Typen:
  - ▶ Einkaufswagen, Path, MyString (Listen-ähnlich)
  - ▶ Resultat, Preis, Trav (Punktierte Typen)
- ▶ Ähnliche **Funktionen** darauf
- ▶ Besser: **eine** Typdefinition mit Funktionen, Instantiierung zu verschiedenen Typen

—→ Nächste Vorlesung

# Zusammenfassung

- ▶ Datentypen können **rekursiv** sein
- ▶ Rekursive Datentypen sind **unendlich** (induktiv)
- ▶ Funktionen werden **rekursiv** definiert
- ▶ Fallbeispiele: Einkaufen in Bob's Shoppe, Labyrinthtraversion
- ▶ Viele strukturell ähnliche Typen
- ▶ **Nächste** Woche: Abstraktion über Typen (Polymorphie)