



Funktionale Programmierung

Woche 3 – Funktionen auf Bäumen

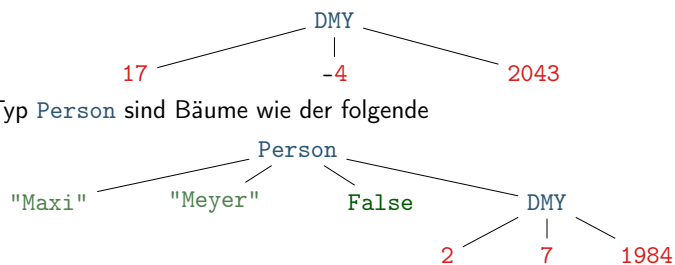
René Thiemann
Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Institut für Informatik

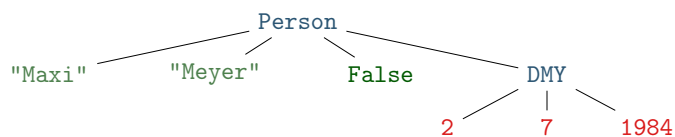
Beispiele nicht-rekursiver Datentypen

```
data Date = DMY Int Int Integer deriving Show
data Person = Person String String Bool Date deriving Show
```

- Werte vom Typ `Date` sind Bäume wie der folgende



- Werte vom Typ `Person` sind Bäume wie der folgende



Letzte Vorlesung

- Daten sind oft in Baum-Form
- jeder Wert und Ausdruck und jede Funktion hat einen **Typ**
- Typen von linker (**lhs**) und rechter (**rhs**) Seite einer Funktions-Definition **lhs = rhs** müssen gleich sein
- vordefinierte Typen**: `Int`, `Integer`, `Float`, `Double`, `String`, `Char`, `Bool`
- benutzerdefinierte **Datentypen**

```
data TName =
    CName1 type1_1 ... type1_N1
    | ...
    | CNameM typeM_1 ... typeM_NM
    deriving Show
```
- ein **Konstruktor** `CNameI :: typeI_1 -> ... -> typeI_NI -> TName` ist eine Funktion, die nicht ausgewertet wird
- `TName` ist **rekursiv**, wenn mindestens ein `typeI_J` genau `TName` ist
- Namen von Typen und Konstruktoren beginnen mit Großbuchstaben

RT et al. (IFI @ UIBK)

Woche 3

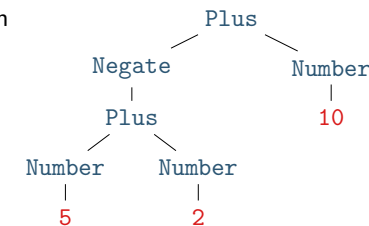
2/24

Beispiel eines rekursiven Datentyps – Expr

```
data Expr =
    Number Integer
    | Plus Expr Expr
    | Negate Expr
    deriving Show
```

- der Typ kann mathematische Ausdrücke $(-(5 + 2)) + 10$ als Haskell-Wert repräsentieren:
`Plus (Negate (Plus (Number 5) (Number 2))) (Number 10)`

- Repräsentation als Baum



- oberer Wert ist nicht gleich `Number 3` und wird auch nicht dazu ausgewertet

Beispiel eines rekursiven Datentyps – Listen

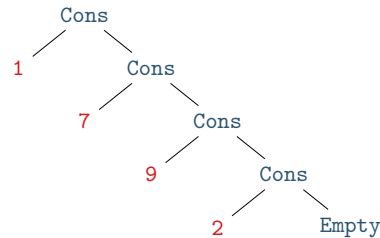
- Listen sind eine spezielle Form von Bäumen, z.B. Listen von `Integer` Zahlen

```
data List =
  Empty
  | Cons Integer List
  deriving Show
```

- Repräsentation der Liste [1, 7, 9, 2]

- in Haskell als Text: `Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))`

- als Baum:



Neue Formen von Funktions-Definitionen

Funktions-Definitionen und Ausdrücke

- bislang hatten alle Funktions-Definitionen die Form

```
funName x1 ... xN = expr
```

wobei

- `x1 ... xN` Variablen sind; eine Funktion kann beliebig viele Argumente haben, inkl. 0
- `expr` ist ein **Ausdruck**, d.h., ein mathematischer Ausdruck bestehend aus
 - Variablen: `x`, `y`, `xs`, `f`, ...
 - Literalen: `5`, `3.4`, `'a'`, `"hello"`, ...
 - Funktions-Anwendungen: `pi`, `square expr`, `average expr1 expr2`, ...
 - Konstruktor-Anwendungen: `True`, `Number expr`, `Cons expr1 expr2`, ...
 - Operator-Anwendungen: `- expr`, `expr1 + expr2`, ...
 - Klammern
- Anmerkung: Funktions- und Konstruktor-Anwendungen binden stärker als Operator-Anwendungen

$(\text{square } 2) + 4 = \text{square } 2 + 4 \neq \text{square } (2 + 4)$

- diese Vorlesung: **Erweiterung der Form von Funktions-Definitionen**, insbesondere um Funktionen auf Datentypen zu definieren

Erstellung neuer Werte – Beispiel für Typ `Expr`

- neue Werte werden einfach durch Konstruktoren erstellt
- Beispiel: Datentyp `Expr`

```
data Expr = Number Int | Plus Expr Expr | Negate Expr
```

(für den Rest der Vorlesung wird `"deriving Show"` nicht explizit hingeschrieben)

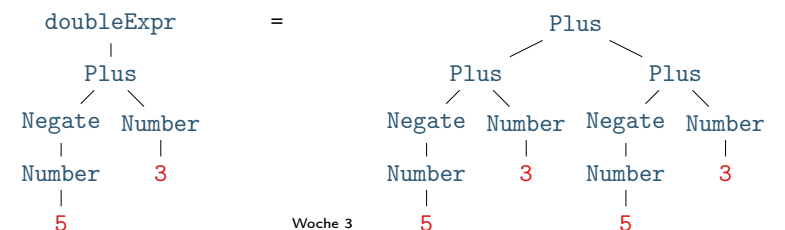
- Aufgabe: definieren Sie eine Funktion zur Verdopplung, d.h., Multiplikation mit 2

- Lösung:

```
doubleNum x = x + x -- Verdopplung einer Zahl
```

```
doubleExpr e = Plus e e -- Verdopplung eines Ausdrucks vom Typ Expr
```

- Auswertung: `doubleExpr`



Erstellung neuer Werte – Beispiel für Typ Person

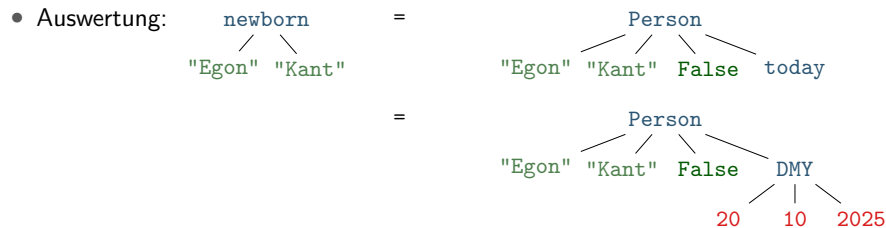
- Erinnerung: Datentyp `Person` ist wie folgt definiert

```
data Date = DMY Int Int Integer
data Person = Person String String Bool Date
```
- Aufgabe: erstellen Sie eine Funktion, die Vor- und Nachname bekommt und daraus ein Wert vom Typ `Person` erstellt, der ein neugeborenes Kind repräsentiert

- Lösung:

```
today = DMY 20 10 2025
```

```
newborn fName lName = Person fName lName False today
```



Funktions-Definitionen mit Pattern

- bislang waren alle Funktions-Definitionen von der Form

```
funName x1 ... xN = expr
```

mit einer Liste von Variablen `x1 ... xN`
- in diesen Definitionen kann man nicht die Struktur der Eingaben erkennen
- Ziel: Fallunterscheidung;
Funktions-Definitionen, die abhängig von Struktur der Eingabe sind
- Beispiel mit dem Datentyp für Fahrzeuge

```
data Brand = Audi | BMW | Fiat | Lamborghini
data Vehicle = Car Brand Double | Bicycle | Truck Int
```

 - Aufgabe: Konvertieren Sie ein Fahrzeug in einen Text
 - Algorithmus:
 - wenn die Eingabe ein **Auto mit x PS** ist, dann erzeuge den Text "a car with x PS"
 - wenn die Eingabe ein **Fahrrad** ist, dann erzeuge den Text "a bicycle"
 - wenn die Eingabe ein **Truck mit x Rädern** ist, dann erzeuge den Text "a x -wheel truck"
- in Haskell wird die Struktur von Bäumen durch **Pattern** (Muster) beschrieben
- die Frage, ob ein Eingabe-Baum zu einem Pattern passt nennt man **Pattern Matching**

Patterns

- ein **Pattern** ist ein Ausdruck folgender Bauart
 - `x` Variablen wie bei einer Funktions-Definition
 - `_` Unterstrich
 - `CName pat1 ... patN` Konstruktor-Anwendung mit Pattern `pat1 ... patN` als Argumenten
 - `x@pat` Variable gefolgt vom @-Zeichen und einem Pattern
- wobei
 - jede Variable nur einmal vorkommen darf
 - Zahlen, Zeichenketten und Zeichen wie Konstruktoren behandelt werden
 - Klammern bei Bedarf genutzt werden können, um verschachtelte Pattern zu bilden

- Beispiele

- `Car brand ps`
- `Car _ ps`
- `Car BMW 100`
- `Car _ (50 + 50)`
- `Person "Egon" lName _ _`
- `p@(Person _ _ _ (DMY 20 10 _))`
- `Person name name _ _`

+ ist kein Konstruktor

doppelte Verwendung einer Variable



Pattern Matching

- Gegeben ein Ausdruck und ein Pattern, **Pattern Matching**
 - entscheidet, ob der Ausdruck zu dem Pattern passt (das Pattern **matcht** den Ausdruck),
 - und im positiven Fall wird eine Substitution als Lösung erzeugt, in der Variablen aus dem Pattern mit (Teil-)Ausdrücken belegt werden
- Beispiele
 - `Car brand ps` matcht `expr`, wenn `expr` ein beliebiges Auto ist; die Substitution speichert die Marke (in Variable `brand`) and die Leistung (in Variable `ps`)
 - `Car _ ps` matcht `expr`, wenn `expr` ein beliebiges Auto ist; die Substitution wird nur die Leistung speichern, die Marke ist nicht verfügbar
 - `Car BMW 100` matcht `expr`, wenn `expr` ein BMW mit genau 100 PS ist; die Substitution ist leer
 - `Person "Mia" lName _ _` matcht `expr`, wenn `expr` eine Person mit Vornamen Mia ist; Substitution speichert Nachnamen in `lName`
 - `p@(Person _ _ _ (DMY 20 10 _))` matcht `expr` wenn `expr` eine Person ist, die heute Geburtstag hat; die Substitution speichert die gesamte Person in Variable `p`

Algorithmus zum Pattern Matching

- im Algorithmus wird eine Substitution in Form $x_1/\text{expr}_1, \dots, x_N/\text{expr}_N$ notiert (hier ist / nicht die Division, sondern der wird-substituiert-durch Operator)
- Algorithmus für Eingabe-Pattern pat und Ausdruck expr
 - Fall pat ist Variable x : Matching gelingt, Substitution ist x/expr
 - Fall pat ist $_$: Matching gelingt, leere Substitution
 - Fall pat ist $x@\text{pat}_1$: Matching gelingt gdw. expr von pat_1 gematcht wird; füge dann x/expr zur Substitution hinzu
 - Fall pat ist $\text{CName pat}_1 \dots \text{pat}_N$:
 - falls expr die Form $\text{OtherCName} \dots$ hat und $\text{CName} \neq \text{OtherCName}$ gilt, dann schlägt das Matching fehl
 - falls expr die Form $\text{CName expr}_1 \dots \text{expr}_N$ hat, dann matche expr_1 mit $\text{pat}_1, \dots$, matche expr_N mit pat_N ; wenn alle Matchings erfolgreich sind, dann gelingt das Matching und die Substitutionen werden vereinigt, ansonsten schlägt das Matching fehl
 - anderenfalls, werte expr erst weiter aus, bis der äußerste Konstruktor sichtbar wird
- Anmerkung: der Algorithmus zur Berechnung von Pattern Matching nutzt selber Pattern Matching

Pattern Matching Algorithmus – Beispiele

- matche Ausdruck `Car BMW (20 + 80)` mit verschiedenen Pattern pat
 - $\text{pat} = x$: Erfolg mit Substitution $x / \text{Car BMW } (20 + 80)$
 - $\text{pat} = \text{Car brand ps}$: Erfolg mit Substitution $\text{brand} / \text{BMW}, \text{ps} / 20 + 80$
 - $\text{pat} = \text{Car brand } _$: Erfolg mit Substitution $\text{brand} / \text{BMW}$
 - $\text{pat} = \text{Car Audi } _$: Fehlschlag
 - $\text{pat} = \text{Car } _ 100$: Erfolg mit leerer Substitution, benötigt Auswertung
- nächstes Beispiel mit Ausdruck `Person "Mia" "Kuhn" True (DMY 20 10 1970)`
 - $\text{pat} = \text{Person "John" lName } _ _$: Fehlschlag
 - $\text{pat} = p@(\text{Person } _ _ _ (\text{DMY } 20 \ 10 \ _))$: Erfolg mit Substitution $p / \text{Person "Mia" "Kuhn" True (DMY 20 10 1970)}$

Funktions-Definitionen mit Pattern Matching

- bislang waren alle Funktions-Definitionen von der Form
`funName x1 ... xN = expr`
- jetzt: Verallgemeinerung der Form auf zwei Weisen
 - eine Funktions-Definition hat die Form
$$\text{funName pat}_1 \dots \text{pat}_N = \text{expr} \quad (\star)$$
wobei alle Variablen der Pattern $\text{pat}_1 \dots \text{pat}_N$ maximal einmal in den Pattern vorkommen
 - es kann mehrere definierende Gleichungen für die gleiche Funktion geben
- Auswertung von `funName expr1 ... exprN` für einzelne Funktions-Gleichung $(\star)$
 - wenn expr_1 von pat_1 gematcht wird und ... und expr_N von pat_N gematcht wird, dann ist die Gleichung anwendbar und `funName expr1 ... exprN` wird durch die rechte Seite expr ersetzt, wobei man die Substitutionen aus dem Matching anwendet
 - ansonsten ist $(\star)$ nicht anwendbar
- Auswertung von `funName expr1 ... exprN`
 - wende die oberste Funktions-Gleichung an, die anwendbar ist
 - falls keine Gleichung anwendbar ist, dann breche mit einer Fehlermeldung ab

Funktions-Definitionen – Beispiele mit dem Typ Person

```
data Date = DMY Int Int Integer
data Person = Person String String Bool Date
data Option = Some Integer | None
```

- Aufgabe: ändere den Nachnamen einer Person
`withLastName lName (Person fName _ m b) = Person fName lName m b`
Anmerkung: in Haskell werden Daten nie verändert, sondern neu erzeugt
- Aufgabe: Berechne das Alter einer Person in Jahren, wenn diese heute Geburtstag hat, sonst gebe nichts zurück
`ageYear (Person _ _ _ (DMY 20 10 y)) = Some (2025 - y)`
`ageYear _ = None`
Anmerkung: die Reihenfolge der Gleichungen ist hier wichtig
- Aufgabe: Erstellen Sie eine Begrüßung für eine Person
`greeting p@(Person name _ _ _) = gHelper name (ageYear p)`
`gHelper n None = "Hello " ++ n`
`gHelper n (Some a) = "Hi " ++ n ++ ", you turned " ++ show a`
Anmerkung: `(++)` verkettet zwei Strings, `show` konvertiert einen Wert in einen String

Vereinigung von Substitutionen und Gleichheit

- betrachten Sie folgendes Programm, um auf Gleichheit von zwei Argumenten zu testen
`equal x x = True`
`equal _ _ = False`
- wie würde die Auswertung von `equal 5 7` funktionieren?
 - erstes Argument: `x` matcht `5`, erhalte Substitution `x / 5`
 - zweites Argument: `x` matcht `7`, erhalte Substitution `x / 7`
 - Vereinigung dieser Substitutionen ist nicht möglich: `x / ???`
- Haskell umgeht dieses Problem durch die Einschränkungen, dass Variablen auf linken Seiten einer definierenden Gleichung nur einmal verwendet werden dürfen; die Funktions-Definition von `equal` ist in Haskell nicht erlaubt
- korrekte Lösung, um auf Gleichheit zu Testen:
 - `(==)` ist ein vordefinierter Operator, um zwei Werte **des gleichen Typs** zu vergleichen, das Ergebnis ist ein Wert vom Typ `Bool`
 - um Gleichheit auf eigenen Datentypen zu verwenden, ersetzen Sie `deriving Show` durch `deriving (Show, Eq)`
 - korrekte Ausdrücke: `5 == 7`, `"Peter" == name`, ..., aber nicht `"five" == 5`

Funktions-Definitionen – Beispiel mit Typ `Bool`

- der Typ `Bool` ist intern wie folgt definiert: `data Bool = True | False`
- folgende Funktion berechnet die Konjunktion zweier `Bool`-Werte
`conj True b = b`
`conj False _ = False`
- Beispiel Berechnung (die Zahlen werden hier nur zur Benennung genutzt)
`conj1 (conj2 True False) (conj3 True True)`
-- check which equation is applicable for conj1
-- first equation triggers evaluation of first argument of conj1 (True)
-- check which equation is applicable for conj2
-- first equation is applicable with substitution b/False
`= conj1 False (conj3 True True)`
-- now see that only second equation is applicable for conj1
`= False`
- Anmerkung: viele Boole'sche Funktionen sind vordefiniert, z.B.,
`(&&)` (Konjunktion), `(||)` (Disjunktion),
`(/=)` (exklusives Oder), `not` (Negation)

Funktions-Definitionen mit Fallunterscheidung

- Design-Prinzip für Funktionen:
definieren Sie Gleichungen, die alle möglichen Arten von Eingaben behandeln
- Beispiel
`data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun`

`weekend Sat = True`
`weekend Sun = True`
`weekend _ = False`
- Beispiel: erstes Element einer Liste
`data List = Empty | Cons Integer List`

`first (Cons x xs) = x`
`first Empty = error "first on empty list"`
- `error` nimmt einen `String` und liefert diesen Text als Fehlermeldung bei der Ausführung
- ohne die zweite Gleichung von `first` würde `first Empty` in einem allgemeinen "non-exhaustive patterns" Fehler resultieren

Rekursive Funktions-Definitionen

- Beispiel: Länge einer Liste
`len Empty = 0`
`len (Cons x xs) = 1 + ???` -- ??? = die Laenge der Liste xs
- potientes Problem: wir würden gerne die Funktion aufrufen, die wir gerade definieren
- das ist bei der Programmierung erlaubt und nennt sich **Rekursion**:
eine Funktion, die sich in der rechten Seite selbst aufruft
`len Empty = 0`
`len (Cons x xs) = 1 + len xs` -- `len xs` ist ein rekursiver Aufruf
- in rekursiven Aufrufen sollte man sicherstellen, dass die Argumente kleiner werden
- die Auswertung funktioniert unverändert wie bisher
`len (Cons 1 (Cons 7 (Cons 9 Empty)))`
`= 1 + (len (Cons 7 (Cons 9 Empty)))`
`= 1 + (1 + (len (Cons 9 Empty)))`
`= 1 + (1 + (1 + (len Empty)))`
`= 1 + (1 + (1 + 0)) = 1 + (1 + 1) = 1 + 2 = 3`

Rekursive Funktions-Definitionen – Verkettung zweier Listen

- Ziel: Verkettung zweier Listen, d.h., die Verkettung von $[1, 5]$ und $[3]$ resultiert in $[1, 5, 3]$
- Lösung: Pattern Matching und Rekursion im ersten Argument

```
append Empty ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

- Beispiel Auswertung

```
append (Cons 1 (Cons 3 Empty)) (Cons 2 (Cons 7 Empty))
= Cons 1 (append (Cons 3 Empty) (Cons 2 (Cons 7 Empty)))
= Cons 1 (Cons 3 (append Empty (Cons 2 (Cons 7 Empty))))
= Cons 1 (Cons 3 (Cons 2 (Cons 7 Empty)))
```

Rekursive Funktions-Definitionen – Mathematische Ausdrücke (1)

- betrachten Sie den Datentyp für mathematische Ausdrücke

```
data Expr =
  Number Integer
  | Plus Expr Expr
  | Negate Expr
```

- Aufgabe: implementieren Sie einen Taschenrechner, d.h. werten Sie einen Ausdruck aus
- Lösung:

```
eval (Number x)    = x
eval (Plus e1 e2)   = eval e1 + eval e2
eval (Negate e)     = - eval e
```

Rekursive Funktions-Definitionen – Mathematische Ausdrücke (2)

- betrachten Sie den Datentyp für mathematische Ausdrücke

```
data Expr =
  Number Integer
  | Plus Expr Expr
  | Negate Expr
```

- Aufgabe: erstellen Sie eine Liste aller Zahlen, die in dem Ausdruck vorkommen
- Lösung:

```
numbers (Number x)    = Cons x Empty
numbers (Plus e1 e2)   = append (numbers e1) (numbers e2)
numbers (Negate e)     = numbers e
```

Zusammenfassung

- Funktions-Definitionen können eine Fallunterscheidung mittels **Pattern Matching** implementieren
 - ein Pattern beschreibt die Struktur eines Baums
 - mehrere Gleichungen sind erlaubt; Haskell versucht diese nach-und-nach anzuwenden, beginnend mit der ersten Gleichung
- Funktions-Definitionen können **rekursiv** sein
 - `funName ... = ... (funName ...) ... (funName ...) ...`
 - die Argumente in rekursiven Aufrufen sollten kleiner sein als die Argumente in linker Gleichungs-Seite