



Funktionale Programmierung

Woche 2 – Bäume und Datentypen

René Thiemann

Philipp Dablander

Joshua Ocker

Michael Schaper

Lilly Schönherr

Adam Pescoller

Letzte Vorlesung

- Algorithmus (informeller Text) $\neq$ Programm (konkrete Programmiersprache)
- Haskell Skript (Code, Programm, Quelltext, ...), z.B., Program.hs
`fahrenheitToCelsius f = (f - 32) * 5 / 9`
besteht aus Funktionsdefinitionen, d.h. Gleichungen wobei links der Name und die Argumente der Funktion stehen, und rechts ein Ausdruck, wie die Funktion auszurechnen ist.
- Funktions- und Variablen-Namen starten mit einem Kleinbuchstaben
- read-eval-print loop (REPL):
lade ein Skript, lese Ausdrücke ein, werte diese aus und zeige das Ergebnis

```
$ ghci Program.hs
... Start-Nachricht ...
ghci> fahrenheitToCelsius (3 + 20) - 7
-12.0
ghci> ... weitere Ausdruecke ...
...
ghci> :q
```

Strukturierte Daten

Unterschiedliche Repräsentationen von Daten

- manche (abstrakte) Dinge können verschieden repräsentiert werden

- Beispiel: Zahlen

- römisch: XI
- dezimal: 11
- binär: 1011
- deutsch: elf
- Strichliste: |||||

- Tatsache: Algorithmen sind abhängig von der Wahl der Repräsentation

- Beispiel: Addition

- dezimal + binär: bearbeite die Ziffern beider Zahlen von rechts nach links

$$\begin{array}{r} 7823 \\ + 909 \\ \hline 8732 \end{array}$$

- Strichliste: schreibe die beiden Zahlen nebeneinander $(||| + || = |||)$
- römisch: Algorithmus? $(IV + IX = XIII)$
- deutsch: nicht gut geeignet $(\text{neunundzwanzig} + \text{zwei} = \text{einunddreißig})$

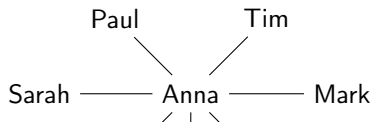
- in Haskell: Zahlen sind vordefiniert, Repräsentation wird vor BenutzerInnen verborgen

Unterschiedliche Repräsentationen von Daten – Fortgeführt

- Repräsentation muss Anwendungsabhängig gewählt werden
- Beispiel: Person
 - Fotograph:

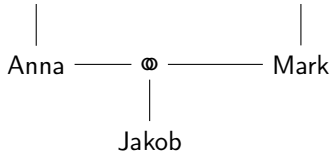


- Soziale Analyse:



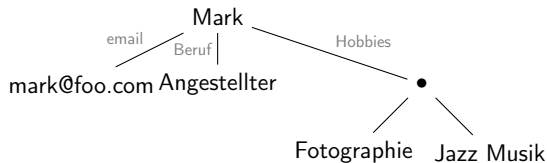
- Marketing: Mark (mark@foo.com, Angestellter, Hobbies: Fotographie, Jazz Musik, ...)

- Ahnenforschung: Conni — ∅ — John Ute — ∅ — Jakob

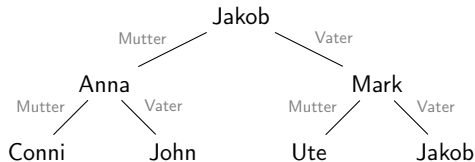


Daten in Baumgestalt

- in der funktionalen Programmierung sind Daten oft in **Baumgestalt**
- ein **Baum**
 - hat genau einen **Wurzel** Knoten
 - kann mehrere Teilbäume haben; Knoten ohne Teilbäume sind **Blätter**
 - Knoten und Kanten können beschriftet sein
- in der Informatik wachsen Bäume üblicherweise von oben nach unten
- Beispiele der vorigen Folie
 - Markteting:

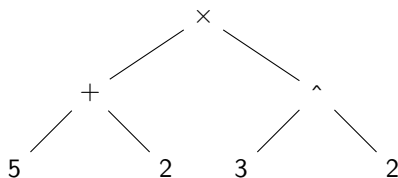


- Ahnenforschung:



Ausdrücke = Bäume

- mathematische Ausdrücke haben eine Baum-Repräsentation
- Beispiel
 - Ausdruck in Text-Form: $(5 + 2) \times 3^2$
 - Ausdruck als Baum

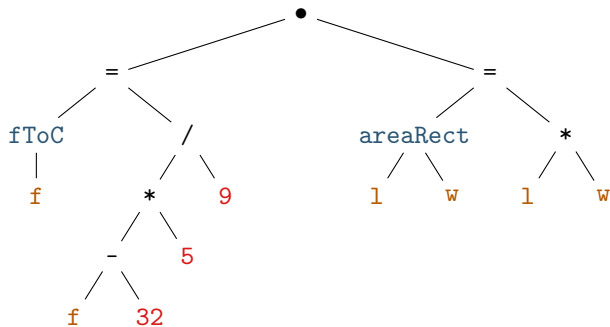


- Anmerkungen
 - **Parsen** ist der Vorgang, einen Text in Baum-Darstellung zu konvertieren
 - Präzedenzen der Operatoren ($^$ bindet stärker als $\times$, und $\times$ stärker als $+$) und Klammern werden nur für das Parsen benötigt
 - $(5 + 2) \times (3^2)$ wird zum obigen Baum geparkt
 - $5 + 2 \times 3^2$ und $((5 + 2) \times 3)^2$ repräsentieren andere Bäume
 - Algorithmus einen Taschenrechners
 - konvertiere Text-Eingabe in einen Baum
 - werte den Baum von unten nach oben aus: starte bei den Blättern und ende bei der Wurzel

Programme = Bäume

- Programme werden ebenfalls als Baum repräsentiert: **Abstrakter Syntax Baum** (AST)
- Beispiel
 - Programm in Text-Form

```
-- some comment
fToC f = (f - 32) * 5 / 9
areaRect l w = l * w
```
 - abstrakter Syntax Baum (skizziert)



- Kommentare und Klammern kommen im Syntax Baum nicht mehr vor

Bäume als Daten

- viele Programme arbeiten mit Bäumen als Daten
- Beispiele
 - Taschenrechner werten Ausdrücke in Baum-Darstellung aus
 - Compiler übersetzt abstrakten Syntax Baum in ausführbaren Binärcode
 - Suchmaschine übersetzt Anfrage in HTML (Baumdarstellung)
 - Textverarbeitungen speichern Dokumente im XML Format (Baumdarstellung)
 - Dateisysteme werden als Baum organisiert
- Bäume können gut als **mentales Modell** oder als Repräsentation von Daten genutzt werden
- Vorteil: Baum-basierte Daten werden in der funktionalen Programmierung gut unterstützt
- nächste Vorlesung: Definition von Funktionen mit Bäumen als Ein- und Ausgabe
- diese Vorlesung: Einschränkung von Bäumen mittels **Typen**

Typen

Typen

- Funktionen werden häufig mit Definitions- und Wertebereich annotiert, z.B.
 - $(!) : \mathbb{N} \rightarrow \mathbb{N}$
 - $(/) : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \rightarrow \mathbb{R}$
 - $\log_2 : \mathbb{R}_{>0} \rightarrow \mathbb{R}$
- Definitions- und Wertebereich bieten nützliche Information
 - Definitionsbereich: mit welchen Werten darf man eine Funktion aufrufen
 - Wertebereich: was sind mögliche Resultate einer Funktion
- Ziel: Spezifikation von Definitions- und Wertebereich von (Haskell-)Funktionen
- Notation
 - **Elemente** oder **Werte**
 - Mathematik: $5, 8, \pi, -\frac{3}{4}, \dots$
 - Haskell: `5, 8, 3.141592653589793, -0.75, ..., "hello", 'c', ...`
 - **Mengen von Elementen** spezifizieren Definitions- und Wertebereich, in Haskell: **Typen**
 - Mathematik: $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{Q} \setminus \{0\}, \dots$
 - Haskell: `Integer` ($\mathbb{Z}$), `Double` ($\mathbb{R}$), `String`, `Char`, ...

Typ-Annotationen

- in der Mathematik drückt $\in$ die Zugehörigkeit aus: $7 \in \mathbb{Z}$, $7 \in \mathbb{R}$, $0.75 \notin \mathbb{Z}$
- in Haskell gibt es **Typ-Annotationen**, die ausdrücken, dass ein Wert oder ein Ausdruck einen Typ hat
 - Format: `expression :: type`
 - Beispiele
 - `7 :: Integer` oder `7 :: Double`
 - `'c' :: Char`
- Haskell Compiler prüft, ob ein Ausdruck in der Tat den Typ einer Annotation hat
 - falls ein Ausdruck nicht den angegebenen Typen hat, gibt es einen Typ-Fehler
 - Beispiele, die zu Fehlern führen
 - `7 :: String` oder `0.75 :: Integer` oder `'c' :: String`
 - `(7 :: Integer) :: Double`
 - Anmerkungen
 - in der Mathematik gilt $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q}$, aber in Haskell sind die Typen `Integer` und `Double` unvergleichbar
 - manche Ausdrücke können mehrere Typen haben (z.B. `7` oder auch `7 + 5`), aber häufig sind explizite Konvertierung zwischen Zahlentypen erforderlich
 - sobald eine Typ-Annotation vorliegt, ist der Typ dieses Ausdrucks fixiert

Typen von Haskell Ausdrücken

- nicht nur Werte, sondern auch Funktionen haben einen Typ, z.B.
 - `(/) :: Double -> Double -> Double`
 - `(+) :: Integer -> Integer -> Integer`
 - `(+) :: Double -> Double -> Double`
 - `head :: String -> Char`

Anmerkungen

- eine Funktion kann mehrere Typen haben, z.B. `(+)`
- Typen $\subsetneq$ Mengen, z.B. ist `(/) :: Double -> Double \ {0} -> Double` nicht erlaubt
- bei der **Typ-Überprüfung** wird sichergestellt, dass Funktions-Aufrufe korrekt sind, d.h. die Typen der Argumente müssen zu den Eingabe-Typen der Funktion passen
- Beispiel: betrachten Sie den Ausdruck `expr1 / expr2`
 - Erinnerung: `(/) :: Double -> Double -> Double`
 - es wird geprüft, dass `expr1` und `expr2` den Typ `Double` haben
 - der Typ des gesamten Ausdrucks `expr1 / expr2` wird dann `Double` sein
- Beispiele
 - `5 + 3 / 2`
 - `5 + '3'` oder `5.2 + 0.8 :: Integer`



Statische Typ-Überprüfung

- Haskell führt eine statische Typ-Überprüfung durch
- **statische** Überprüfung: Typen werden geprüft, bevor die Auswertung startet (Alternative: dynamische Prüfung der Typen während der Auswertung)
- statische Prüfung beim Laden eines Haskell Skripts
 - Prüfung der Typen aller Funktions Definitionen `someFun x ... z = expr`:
prüfe, dass **linke Seite** `someFun x ... z` den **gleichen Typ wie rechte Seite** `expr` hat
 - Konsequenz: Ausdrücke können bei der Auswertung ihren Typ nicht ändern
- statische Prüfung bei Eingabe eines Ausdrucks in REPL
 - prüfe, ob der Ausdruck Typ-korrekt ist, bevor mit der Auswertung begonnen wird
- Vorteile, Milner 1978: **Well typed programs cannot go wrong.**
 - keine Typ-Überprüfung während der Auswertung
 - keine Typ-Fehler während der Auswertung

Eingebaute Typen – Ein Erster Überblick

- Zahlen
 - **Integer** – beliebig genaue ganze Zahlen ($\mathbb{Z}$)
 - **Int** – ganze Zahlen mit fixierter Genauigkeit, mindestens $\{-2^{28}, \dots, 2^{28} - 1\}$ (-100, 0, 999)
 - **Float** – Gleitkomma-Zahlen einfacher Genauigkeit (-12.34, 5.78e36)
 - **Double** – Gleitkomma-Zahlen doppelter Genauigkeit
- Zeichen und Zeichenketten
 - **Char** – ein einzelnes Zeichen ('a', 'Z', ' ', '5')
 - **String** – beliebig lange Zeichenketten ("", "a", "Die Antwort ist 42.")
 - einige Zeichen müssen **escaped** werden mit Hilfe des Backslash-Symbols \:
 - '\t' und '\n' – Tabulator und Zeilenumbruch
 - '\"' und '\'' – doppelte und einfache Anführungszeichen
 - '\\' – das Backslash Zeichen
 - Beispiel: im Programm

```
text = "Sage \"Hallo\"\\n\nwenn Du nach Hause kommst"
```

repräsentiert die Zeichenkette `text` den folgenden Text:
Sage "Hallo"
wenn Du nach Hause kommst
- **Bool** – Ja/Nein-Entscheidungen oder Wahrheits-Werte (True, False)

Datentypen

Aktueller Stand

- jeder Wert und jede Funktion in Haskell hat einen Typ
- Typen werden genutzt, um Definitions- und Wertebereich einer Funktion anzugeben
- Beispiel: `fahrenheitToCelsius :: Double -> Double`
- eingebaute Typen für Zahlen, Zeichenketten und Wahrheitswerte
- bislang nicht vorhanden: Mechanismus, um Baum-basierte Daten zu beschreiben
- Lösung: Definition von (algebraischen) **Datentypen**

Datentyp Definitionen

- Erinnerung: ein Baum besteht aus einer (beschrifteten) Wurzel und mehreren Teilbäumen
- eine **Datentyp** Definition beschreibt eine Menge von Bäumen durch die Angabe aller erlaubten beschrifteten Wurzeln mit zugehörigen Teilbäumen
- Haskell Skripte können mehrere **Datentyp Definitionen** der folgenden Form enthalten

```
data TName =  
    CName1 type1_1 ... type1_N1  
    | ...  
    | CNameM typeM_1 ... typeM_NM  
deriving Show
```

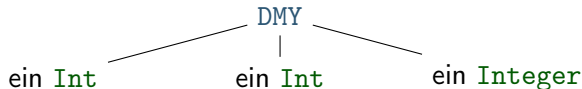
- `data` ist ein Schlüsselwort in Haskell, um einen neuen Datentyp zu erstellen
- `TName` ist der Name des neuen Typs; **Typ-Namen** beginnen immer mit einem Großbuchstaben
- `CName1, ..., CNameM` sind die Namen der erlaubten Wurzeln;
diese nennt man **Konstruktoren** und Sie müssen ebenfalls mit Großbuchstaben beginnen
- `typeI_J` kann ein beliebiger Haskell Typ sein, inklusive `TName`
- `|` wird als Separator zwischen zwei Konstruktoren genutzt
- `deriving Show` wird benötigt, um Werte des Typs `TName` anzuzeigen

Beispiel Datatype Definition – Date

```
data Date = -- Name des Typen
  DMY      -- Name des Konstruktors
    Int    -- Tag
    Int    -- Monat
    Integer -- Jahr
```

deriving Show

- im Beispiel gibt es nur einen Konstruktor: `DMY`
- für Tage und Monate ist die Genauigkeit von `Int` ausreichend
- die Werte des Typs `Date` sind genau Bäume der Gestalt



- in Haskell werden Bäume des Typs `Date` mittels des **Konstruktors** `DMY` erzeugt; `DMY` ist eine Funktion mit Typ `Int -> Int -> Integer -> Date`, die **nicht ausgewertet wird**
- Beispiel-Wert des Typs `Date`: `DMY 14 10 2024`

Beispiel Datentyp Definition – Person

```
data Person = -- Name des Typs
  Person      -- Konstruktor Name kann identisch zum Typ-Namen sein
    String    -- Vorname
    String    -- Nachname
    Bool      -- verheiratet?
    Date      -- Geburtstag
deriving Show
```

- die Verwendung zuvor definierter Typen ist erlaubt, insbesondere `Date`
- somit können Bäume mit mehr als eine Ebene von Teilbäumen entstehen
- das Beispiel-Programm definiert eine Person (und ein Datum)

```
today = DMY 13 10 2025
```

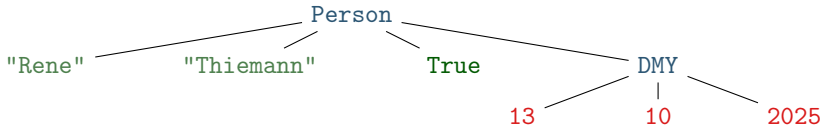
```
myself = Person "Rene" "Thiemann" True today
```

```
-- liefert die gleiche Person wie
```

```
myself = Person "Rene" "Thiemann" True (DMY 13 10 2025)
```

Bäume und Ihre textuelle Repräsentation

- in Haskell müssen Bäume als Text eingegeben werden, und sie werden auch als Text ausgegeben
- um einen Baum mit Konstruktor `C` und Teilbäumen `t1`, ..., `tN` zu definieren
 - schreibt man `C (t1) ... (tN)`;
 - wenn ein Ausdruck `tI` atomar ist, dann kann man die Klammern um `tI` weglassen;
 - das Eingabe Format für Bäume wird auch für Ausdrücke und Funktionsanwendungen genutzt
- Beispiel



- `Person "Rene" "Thiemann" True (DMY 13 10 2025)` ✓
- `Person "Rene" "Thiemann" (5 > 3) (DMY 13 (length "0123456789") 2025)` ✓
- `Person ("Rene", "Thiemann", True, DMY (13, 10, 2025))` ✗
- `Person "Rene" "Thiemann" True DMY 13 10 2025` ✗

Beispiel Datentyp Definition – Vehicle

```
data Brand = Audi | BMW | Fiat | Lamborghini deriving Show
```

```
data Vehicle =
```

```
    Car Brand Double -- Leistung  
    | Bicycle  
    | Truck Int -- Anzahl Raeder  
deriving Show
```

- `Brand` definiert, dass es genau 4 mögliche Marken gibt; alle "Bäume" des Typs `Brand` bestehen nur aus dem Wurzelknoten; solche Datentypen nennt man auch **Aufzählungen**
- es gibt drei Arten von Fahrzeugen, und jedes Fahrzeug speichert unterschiedliche Werte
- Beispiel Ausdrücke des Typs `Vehicle`:

```
Car Fiat (60 + 1)
```

```
Car Audi 149.5
```

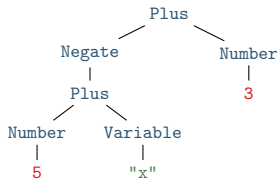
```
Bicycle
```

```
Truck (-7) -- Typen stellen nicht sicher, dass Raeder >= 4 gilt
```

Beispiel Datatype Definition – Expr

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

- Typ `Expr` modelliert arithmetische Ausdrücke mit Addition und Negation
- `Expr` ist ein **rekursiver** Datentyp: `Expr` nutzt `Expr` als Argument eines Konstruktors
- rekursive Datentypen beinhalten Bäume, die beliebig groß werden können
 - Ausdruck $-(5 + x) + 3$ kann als Wert des Typs `Expr` dargestellt werden:
`Plus (Negate (Plus (Number 5) (Variable "x"))) (Number 3)`
 - Ausdruck in Baumdarstellung



Beispiel Datentyp Definition – Listen

- Listen sind nur eine spezielle Art von Bäumen, z.B. Listen von ganzen Zahlen

```
data List =
```

```
  Empty
```

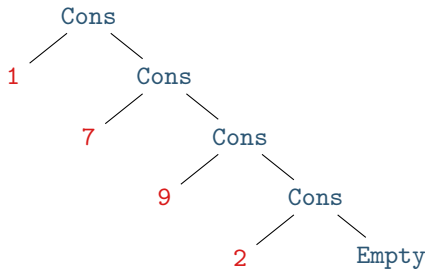
```
  | Cons Integer List
```

```
  deriving Show
```

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

- textuell: `Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))`

- als Baum:



Zusammenfassung

- mentales Modell: Daten = Bäume
- Typ = Menge von Werten; Typen beschreiben, welche Bäume gebildet werden dürfen
- eingebaute Typen für Zahlen und Zeichen(ketten)
- Benutzer-definierbare Datatypen für z.B. Ausdrücke, Listen, Personen,

```
data TName =  
    CName1 type1_1 ... type1_N1  
...   | ...  
      | CNameM typeM_1 ... typeM_NM  
deriving Show
```

- nächste Vorlesung: Funktionen mit Bäumen im Definitions- und Wertebereich