Praktische Informatik 3: Funktionale Programmierung

Vorlesung 1 (18.11.2022): Einführung

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Was ist Funktionale Programmierung?

- Programme als Funktionen — Funktionen als Programme
  - **Keine** veränderlichen Variablen
  - **Rekursion** statt while-Schleifen
- Funktionen als Daten — Daten als Funktionen
  - Erlaubt **Abstraktionsbildung**
- Denken in Algorithmen, nicht in Zustandsveränderung

Lernziele

- **Konzepte** und **typische Merkmale** des funktionalen Programmierens kennen, verstehen und anwenden können:
  - Modellierung mit **algebraischen Datentypen**
  - **Rekursion**
  - Starke **Typisierung**
  - **Funktionen höher Ordnung** (map, filter, fold)
- Datenstrukturen und Algorithmen in einer funktionalen Programmiersprache **umsetzen** und auf einfachere praktische Probleme **anwenden** können.

Modulhandbuch Informatik (Bachelor)

Die Vorlesung *Praktische Informatik 3* vermittelt essenzielles Grundwissen und Basisfähigkeiten, deren Beherrschung für nahezu jede vertiefte Beschäftigung mit Informatik Voraussetzung ist.

I. Organisatorisches

Personal und Termine

- **Vorlesung**:
  Di 14– 16 NW2 C0290 Christoph Lüth <clueth@uni-bremen.de>
  www.informatik.uni-bremen.de/~clueth/
  MZH 4186, Tel. 218-59830
- **Tutoren**:

Di	12– 14	MZH 1110	Tede von Knorre	<tede@uni-bremen.de>
	12– 14	MZH 5600	Raphael Baass	<rbaass@uni-bremen.de>
Mi	14– 16	MZH 1110	Thomas Barkowsky	<barkowsky@uni-bremen.de>
	14– 16	MZH 1450	Alexander Krug	<krug@uni-bremen.de>
	14– 16	Online	Muhammad Tarek Soliman	<soliman@uni-bremen.de>

- **Webseite**: www.informatik.uni-bremen.de/~clueth/lehre/pi3.ws22

Scheinbedingungen

- Übungsblätter:
  - 6 Einzelübungsblätter (fünf beste werden gewertet) und
  - 3 Gruppenübungsblätter (doppelt gewichtet)
- Übungsblätter der letzten Semester werden nur in **Ausnahmefällen** berücksichtigt:
  - Klausur durch **Krankheit** oder **Corona** verpasst.
- Individualität der Leistung: **Elektronische Klausur** am Ende

Elektronische Klausur

- **Termin**: 13.03.2023, 14:00 und 15:45
- **Ort**: Testzentrum am Boulevard neben der Bibliothek
- **Dauer**: 90 Minuten
- **Ablauf**:
  - Einfache Programmierübungen in der Art der Übungsaufgaben
  - Einige Multiple-Choice Fragen als **Bonus**

Scheinbedingungen

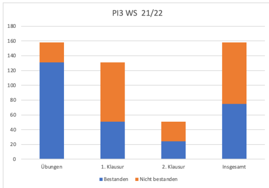
- Mindestens 50% in den Einzelübungsblättern, in allen Übungsblättern und mindestens 50% in der E-Klausur
- Note: 50% Übungsblätter und 50% E-Klausur
- **Notenspiegel** (in Prozent aller Punkte):

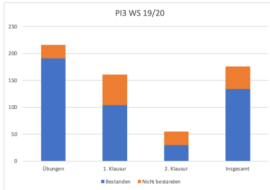
Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	**n/b**

Spielregeln

- **Quellen angeben** bei
  - Gruppenübergreifender Zusammenarbeit
  - Internetrecherche, Literatur, etc.
- **Täuschungsversuch**:
  - Null Punkte, **kein** Schein, **Meldung** an das **Prüfungsamt**
- **Deadline verpaßt?**
  - Triftiger Grund (z.B. Krankheit)
  - **Vorher** ankündigen, sonst **null** Punkte.

Statistik von PI3 im Wintersemester 21/22

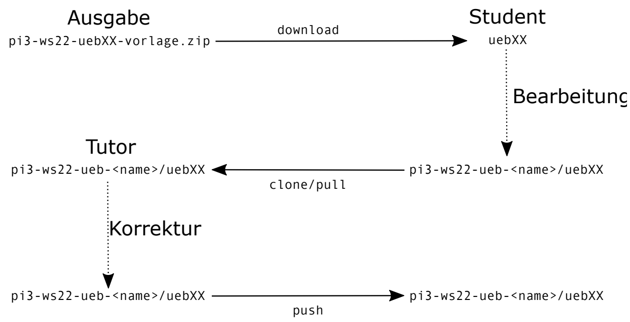




Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite **Montag morgen**
- Besprechung der Übungsblätter in den Tutorien
- 6 Einzelübungsblätter:
  - Bearbeitungszeit bis **Sonntag EOB**
  - Die fünf besten werden gewertet
- 3 Gruppenübungsblätter (doppelt gewichtet):
  - Bearbeitungszeit bis **Sonntag folgender Woche EOB**
  - Übungsgruppen: max. **drei Teilnehmer**
- **Abgabe** elektronisch
- **Bewertung**: Korrektheit, Angemessenheit ("Stil"), Dokumentation

Ablauf des Übungsbetriebs



Warnung

- PI3 ist **nicht** die Fortsetzung von PI1 und PI2.
- Funktionale Programmierung ist **anders** und kann als **schwer** empfunden werden.
- Regelmäßige Bearbeitung der **Übungsblätter** hilft.
- Sucht **rechtzeitig** Unterstützung!

II. Einführung

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

Warum funktionale Programmierung lernen?

- Funktionale Programmierung macht aus Programmierern Informatiker
- Blick über den Tellerrand — was kommt in 10 Jahren?
- **Herausforderungen** der Zukunft:
  - Nebenläufige und **reaktive** Systeme (Mehrkernarchitekturen, serverless computing)
  - Massiv verteilte Systeme („Internet der Dinge")
  - Große Datenmengen („Big Data")

The Future is Bright — The Future is Functional

- Funktionale Programmierung enthält die **wesentlichen** Elemente moderner Programmierung:
  - Datenabstraktion und Funktionale Abstraktion
  - Modularisierung
  - Typisierung und Spezifikation
- Funktionale Ideen jetzt im Mainstream:
  - Reflektion — LISP
  - Generics in Java — Polymorphie
  - Lambda-Funktionen in Java, C++ — Funktionen höherer Ordnung

Geschichtliches: Die Anfänge

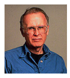
- **Grundlagen** 1920/30
  - Kombinatorlogik und λ-Kalkül (Schönfinkel, Curry, Church)
- Erste funktionale **Programmiersprachen** 1960
  - LISP (McCarthy), ISWIM (Landin)
- **Weitere** Programmiersprachen 1970– 80
  - FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)


Moses Schönfinkel


Haskell B. Curry


Alonzo Church


John McCarthy


John Backus


Robin Milner


Mike Gordon

Geschichtliches: Die Gegenwart

- **Konsolidierung** 1990
  - CAML, Formale Semantik für Standard ML
  - Haskell als Standardsprache
- **Kommerzialisierung** 2010
  - OCaml
  - Scala, Clojure (JVM)
  - F# (.NET)

Warum Haskell?

- **Moderne** Sprache
- Standardisiert, mehrere **Implementationen**
  - Interpreter: `ghci`, `hugs`
  - Compiler: `ghc`, `nhc98`
  - Build: `stack`
- **Rein** funktional
  - Essenz der funktionalen Programmierung

Programme als Funktionen

- Programme als Funktionen:

$$P : Eingabe \to Ausgabe$$

- **Keine veränderlichen Variablen** — kein versteckter **Zustand**
- Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (**referentielle Transparenz**)

Beispiel: Programmieren mit Funktionen

- **Programme** werden durch **Gleichungen** definiert:

```
fact n = if n == 0 then 1 else n* fact(n-1)
```

- Auswertung durch **Reduktion** von **Ausdrücken**:

```
fact 2→ if 2 == 0 then 1 else 2* fact (2-1)
      → if False then 1 else 2* fact 1
      → 2* fact 1
      → 2* if 1 == 0 then 1 else 1* fact (1-1)
      → 2* if False then 1 else  1* fact (1-1)
      → 2* 1* fact 0
      → 2* 1* if 0 == 0 then 1 else 0* fact (0-1)
      → 2* 1* if True then 1 else 0* fact (0-1)
      → 2* 1* 1 → 2
```


Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
rep n s = if n == 0 then "" else s ++ rep (n-1) s
```

- Auswertung:

```
rep 2 "hallo␣"
→ if 2 == 0 then "" else "hallo␣" ++ rep (2-1) "hallo␣"
→ if False then "" else "hallo␣" ++ rep 1 "hallo␣"
→ "hallo␣"++ rep 1 "hallo␣"
→ "hallo␣"++ if 1 == 0 then "" else "hallo␣"++ rep (1-1) "hallo␣"
→ "hallo␣"++ if False then "" else "hallo␣"++ rep 0 "hallo␣"
→ "hallo␣"++ ("hallo␣"++ rep 0 "hallo␣")
→ "hallo␣"++ ("hallo␣"++ if 0 == 0 then "" else "hallo␣"++ rep (0-1) "hallo␣")
→ "hallo␣"++ ("hallo␣"++ if True then "" else "hallo␣"++ rep (-1) "hallo␣")
→ "hallo␣"++ ("hallo␣" ++ "")
→ "hallo␣hallo␣"
```


Auswertung als Ausführungsbegriff

- **Programme** werden durch **Gleichungen** definiert:

$$f(x) = E$$

- **Auswertung** durch **Anwenden** der Gleichungen:
  - Suchen nach Vorkommen von f, e.g. $f(t)$
  - $f(t)$ wird durch $E\left[\frac{t}{x}\right]$ ersetzt
- Auswertung kann **divergieren**!

Ausdrücke und Werte

- Nichtreduzierbare Ausdrücke sind **Werte**
- Vorgebenene **Basiswerte**: Zahlen, Zeichen
  - Durch Implementation gegeben
- Definierte **Datentypen**: Wahrheitswerte, Listen, ...
  - Modellierung von Daten

III. Typen

Typisierung

- **Typen** unterscheiden Arten von Ausdrücken und Werten:

`rep n s = ...`	`n`	Zahl
	`s`	Zeichenkette

- **Wozu** Typen?
  - Frühzeitiges Aufdecken "offensichtlicher" Fehler
  - Erhöhte Programmsicherheit
  - Hilfestellung bei Änderungen

Slogan
"Well-typed programs can't go wrong."
— *Robin Milner*

Signaturen

- Jede Funktion hat eine **Signatur**

```
fact :: Int→ Int
```

```
rep :: Int→ String→ String
```

- **Typüberprüfung**
  - `fact` nur auf `Int` anwendbar, Resultat ist `Int`
  - `rep` nur auf `Int` und `String` anwendbar, Resultat ist `String`

DEMO

Übersicht: Typen in Haskell

Typ	Bezeichner	Beispiel		
Ganze Zahlen	`Int`	`0`	`94`	`-45`
Fließkomma	`Double`	`3.0`	`3.141592`	
Zeichen	`Char`	`'a' 'x'`	`'\034'`	`'\n'`
Zeichenketten	`String`	`"yuck"`	`"hi\nho\"\n"`	
Wahrheitswerte	`Bool`	`True`	`False`	
Funktionen	`a → b`			

- Später **mehr**. **Viel** mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, **konstanter** Aufwand $\longleftrightarrow$ **beliebige** Genauigkeit, **wachsender** Aufwand

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte ($\geq$ 31 Bit)
- `Integer` - beliebig große ganze Zahlen
- `Rational` - beliebig genaue rationale Zahlen
- `Float`, `Double` - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: `Int` und `Integer`

- Nützliche Funktionen (**überladen**, auch für `Integer`):

```
+, *, ^, -  :: Int→ Int→ Int
abs         :: Int→ Int −− Betrag
div, quot   :: Int→ Int→ Int
mod, rem    :: Int→ Int→ Int
```

  Es gilt: `(div x y)*y + mod x y == x`
- Vergleich durch $==, \neq, \leq, <, \ldots$
- **Achtung:** Unäres Minus
  - Unterschied zum Infix-Operator `-`
  - Im Zweifelsfall klammern: `abs (-34)`

Fließkommazahlen: `Double`

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen
- Konversion in ganze Zahlen:
  - `fromIntegral :: Int, Integer→ Double`
  - `fromInteger :: Integer→ Double`
  - `round, truncate :: Double→ Int, Integer`
  - Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- **Rundungsfehler!**

Alphanumerische Basisdatentypen: `Char`

- Notation für einzelne **Zeichen**: 'a',...
- Nützliche **Funktionen**:

```
ord :: Char → Int
chr :: Int → Char

toLower :: Char→ Char
toUpper :: Char→ Char
isDigit :: Char→ Bool
isAlpha :: Char→ Bool
```

- Zeichenketten: String

DEMO

Zusammenfassung

- **Programme** sind **Funktionen**, definiert durch **Gleichungen**
  - Referentielle Transparenz
  - kein impliziter Zustand, keine veränderlichen Variablen
- **Ausführung** durch **Reduktion** von Ausdrücken
- Typisierung:
  - Basistypen: Zahlen, Zeichen(ketten), Wahrheitswerte
  - Jede Funktion `f` hat eine Signatur `f :: a → b`

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 2 (25.10.2022): Funktionen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH — Universität Bremen

Wintersemester 2022/23

Organisatorisches

- **Wichtig:** GitLab-Repos bitte **nicht** öffentlich machen!
  - Settings ⟶ General ⟶ Visibility ⟶ Private (nicht Internal, nicht Public).
- Umverteilung in den Tutorien nötig:

Raphael	50	-14
Tede	48	-12
Thomas	17	+19
Alexander	16	+20
Tarek	50	-14
Insgesamt	181	36

- Eintragung der Tutorien in stud.ip (kommt).

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

- Definition von **Funktionen**
  - Syntaktische Feinheiten
- Bedeutung von Haskell-Programmen
  - Striktheit
- Leben ohne Variablen
  - Funktionen statt Schleifen
  - Zahllose Beispiele

Lernziele
Wir wollen einfache Haskell-Programme schreiben können, eine Idee von ihrer Bedeutung bekommen, und ein Leben ohne veränderliche Variablen führen.

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion

Satz
Fallunterscheidung und Rekursion auf natürlichen Zahlen sind **Turing-mächtig**.

- Funktionen müssen **partiell** sein können — insbesondere nicht-terminierende Rekursion
- Fragen:
  1. Wie schreiben Funktionen in Haskell auf (**Syntax**)?
  2. Was bedeutet das (**Semantik**)?

I. Die Syntax von Haskell

Haskell-Syntax: Charakteristika

- **Leichtgewichtig**
  - Wichtigstes Zeichen:
- Funktionsapplikation: `f a`
  - Klammern sind optional
  - **Höchste** Priorität (engste Bindung)
- Abseitsregel: Gültigkeitsbereich durch Einrückung
  - Keine Klammern (`{ ... }`) (optional)
  - Auch in anderen Sprachen (Python, Ruby)

Funktionsdefinition

Generelle Form:

- **Signatur:**

```
max :: Int→ Int→ Int
```

- **Definition:**

```
max x y = if x < y then y else x
```

  - Kopf, mit Parametern
  - Rumpf (evtl. länger, mehrere Zeilen)
  - Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf
  - Was gehört zum Rumpf (Geltungsberereich)?

Die Abseitsregel

Funktionsdefinition:

```
f x1 x2 x3...xn = e
```

- **Gültigkeitsbereich** der Definition von `f`:
  alles, was gegenüber `f` eingerückt ist.
- Beispiel:

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

- Gilt auch verschachtelt.
- Kommentare sind passiv (heben das Abseits nicht auf).

Kommentare

- Pro Zeile: Ab -- bis Ende der Zeile

```
f x y = irgendwas  -- und hier der Kommentar!
```

- Über mehrere Zeilen: Anfang {-, Ende -}

```
{-
   Hier faengt der Kommentar an
   erstreckt sich ueber mehrere Zeilen
   bis hier                                 -}
f x y = irgendwas
```

  - Kann geschachtelt werden.

Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen . . .

```
f x y = if B1 then P else
        if B2 then Q else R
```

. . . **bedingte Gleichungen**:

```
f x y
  | B1 = P
  | B2 = Q
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: **Laufzeitfehler**! Deshalb:

```
  | otherwise = R
```


Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y                          f x y =
  | g = P y                     let y = M
  | otherwise = f x where           f x = N x
  y = M                         in  if g then P y
  f x = N x                              else f x
```

- `f`, `y`, . . . werden **gleichzeitig** definiert (Rekursion!)
- Namen `f`, `y` und Parameter `x` **überlagern** andere.
  - Parameter überlagern Funktionsnamen (`f f x = f x`)
- Es gilt die **Abseitsregel**
  - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

☞ Siehe Übung 2.??

II. Auswertung von Funktionen

Auswertung von Funktionen

- Auswertung durch **Anwendung** von Gleichungen
- **Auswertungsrelation** $s \to t$:
  - Anwendung einer Funktionsdefinition
  - Anwendung von elementaren Operationen (arithmetisch, Zeichenketten)
- Frage: spielt die **Reihenfolge** eine Rolle?

Auswertung von Ausdrücken

```
inc :: Int → Int          dbl :: Int → Int
inc x = x+ 1              dbl x = 2*x
```

- Reduktion von `inc (dbl (inc 3))`
- Von **außen** nach **innen** (outermost-first):

```
inc (dbl (inc 3))  → dbl (inc 3)+ 1
                   → 2*(inc 3)+ 1
                   → 2*(3+ 1)+ 1 → 2*4+1 → 8+1 → 9
```

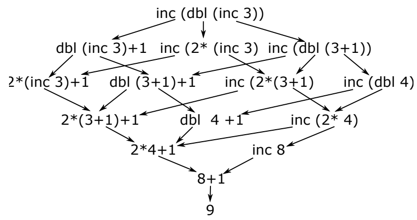
- Von **innen** nach **außen** (innermost-first):

```
inc (dbl (inc 3))  → inc (dbl (3+1)) → inc (dbl 4)
                   → inc (2*4) → inc 8
                   → 8+1 → 9
```


Auswertung von Ausdrücken

```
inc :: Int → Int          dbl :: Int → Int
inc x = x+ 1              dbl x = 2*x
```

- Volle Reduktion von `inc (dbl (inc 3))`:



Konfluenz

- Es kommt immer das gleiche heraus?
- Sei $\xrightarrow{*}$ die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

$\xrightarrow{*}$ ist **konfluent** gdw:
Für alle r, s, t mit $s \xleftarrow{*} r \xrightarrow{*} t$ gibt es u so dass $s \xrightarrow{*} u \xleftarrow{*} t$.

Konfluenz

- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

*Die Auswertungsrelation $\xrightarrow{*}$ für funktionale Programme ist **konfluent**.*

- Beweisskizze:
  Sei `f x = E` und $s \xrightarrow{*} t$:

$$\begin{tikzcd} f\ s \arrow[r, "*"] \arrow[d, "*"] & f\ t \arrow[d, "*"] \\ E\begin{bmatrix} s \\ x \end{bmatrix} \arrow[r, "*"] & E\begin{bmatrix} t \\ x \end{bmatrix} \end{tikzcd}$$

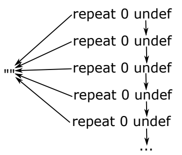
Auswirkung der Auswertungsstrategie

- Auswertungsstrategie ist also egal?
- Beispiel:

```
repeat :: Int→ String→ String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- Auswertung von `repeat 0 undef`:



- outermost-first **terminiert**
- inntermost-first terminiert **nicht**

Termination und Normalform

Definition (Termination)

$\rightarrow$ ist **terminierend** gdw. es **keine unendlichen** Ketten gibt:

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots t_n \rightarrow \ldots$$

Theorem (Normalform)

*Sei $\xrightarrow{*}$ konfluent und terminierend, dann wertet jeder Term zu genau einer **Normalform** aus, die nicht weiter ausgewertet werden kann.*

- Daraus folgt: **terminierende** funktionale Programme werten unter jeder Auswertungsstragie jeden Ausdruck zum gleichen Wert aus (der Normalform).

Auswirkung der Auswertungsstrategie

- Auswertungsstrategie nur für **nicht-terminierende** Programme relevant.
- Leider ist nicht-Termination **nötig** (Turing-Mächtigkeit)
- Gibt es eine **semantische** Charakterisierung?
- Auswertungsstrategie und Parameterübergabe:
  - Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
  - Innermost-first entspricht **call-by-value**, strikte Auswertung

☞ Siehe Übung 2.??

III. Semantik und Striktheit

Bedeutung (Semantik) von Programmen

- **Operationale** Semantik:
  - Durch den **Ausführungsbegriff**
  - Ein Programm **ist**, was es **tut**.
  - In diesem Fall: $\rightarrow$
- **Denotationelle** Semantik:
  - Programme werden auf **mathematische Objekte** abgebildet (Denotat).
  - Für funktionale Programme: **rekursiv** definierte Funktionen

Äquivalenz von operationaler und denotationaler Semantik

Sei P ein funktionales Programm, $\xrightarrow{*}$ die dadurch definierte Reduktion, und $[\![P]\!]$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \xrightarrow{*} v \iff [\![P]\!](t) = v$$

Striktheit

Definition (Striktheit)

Funktion f ist **strikt** $\iff$ Ergebnis ist undefiniert, sobald ein Argument undefiniert ist.

- **Denotationelle** Eigenschaft (nicht operational)
- Haskell ist nach **Sprachdefinition nicht-strikt**
  - `repeat 0 undef` **muss** `""` ergeben.
  - Meisten Implementationen nutzen verzögerte Auswertung
- Andere Programmiersprachen:
  - Java, C, etc. sind call-by-value (nach Sprachdefinition) und damit strikt
  - Fallunterscheidung ist **immer** nicht-strikt, Konjunktion und Disjunktion meist auch.

☞ Siehe Übung 2.??

IV. Leben ohne Variablen

Rekursion statt Schleifen

Fakultät imperativ:

```
r= 1;
while (n > 0) {
  r= n* r;
  n= n- 1;
  }
```

Fakultät rekursiv:

```
fac :: Int→ Int

fac n =
 if n ≤ 0 then 1
 else n* fac (n-1)
```

- Veränderliche Variablen werden zu Funktionsparametern
- Iteration (while-Schleifen) werden zu Rekursion

Rekursive Funktionen auf Zeichenketten

- Test auf die leere Zeichenkette:

```
null :: String→ Bool
null xs = xs == ""
```

- Kopf und Rest einer nicht-leeren Zeichenkette (vordefiniert):

```
head :: String→ Char
tail :: String→ String
```

DEMO

Suche in einer Zeichenkette

- Suche nach einem Zeichen in einer Zeichenkette:

```
count1 :: Char→ String→ Int
```

- In einem leeren String: kein Zeichen kommt vor
- Ansonsten: Kopf vergleichen, zum Vorkommen im Rest addieren

```
count1 c s =
  if null s then 0
  else if head s== c then 1+ count1 c (tail s)
                     else count1 c (tail s)
```

- Übung: wie formuliere ich `count` mit Guards? (Lösung in den Quellen)

DEMO

Strings konstruieren

- : hängt Zeichen vorne an Zeichenkette an (vordefiniert)

```
(:) :: Char→ String→ String
```

- Es gilt: Wenn `not (null s)`, dann `head s : tail s == s`
- Mit `(:)` wird `(++)` definiert:

```
(++) :: String→ String→ String
xs ++ ys
  | null xs   = ys
  | otherwise = head xs : (tail xs ++ ys)
```

- `quadrat` konstruiert ein Quadrat aus Zeichen:

```
quadrat :: Int→ Char→ String
quadrat n c = repeat n (repeat n (c: "") ++ "\n")
```

DEMO

Strings analysieren

- Warum immer nur Kopf/Rest? Warum nicht letztes Zeichen/Anfang?
- Letztes Zeichen (dual zu `head`)

```
last :: String→ Char
last s
  | null s = error "last: empty string"
  | null (tail s) = head s
  | otherwise     = last (tail s)
```

- **Laufzeitfehler** bei leerem String

Strings analysieren

- Anfang der Zeichenkette (dual zu `tail`):

```
init :: String→ String
init s
  | null s = error "init: empty string"     -- nicht s
  | null (tail s) = ""
  | otherwise     = head s : init (tail s)
```

- Damit: Wenn `not (null s)`, dann `init s ++ (last s: "") == s`

Strings analysieren: das Palindrom

- Palindrom: vorwärts und rückwärts gelesen gleich.
- Rekursiv:
  - Alle Wörter der Länge 1 oder kleiner sind Palindrome
  - Für längere Wörter: wenn erstes und letztes Zeichen gleich sind und der Rest ein Palindrom.
- Erste Variante:

```
palin1 :: String→ Bool
palin1 s
  | length s ≤ 1     = True
  | head s == last s = palin1 (init (tail s))
  | otherwise        = False
```

DEMO

Strings analysieren: das Palindrom

- Problem: Groß/Kleinschreibung, Leerzeichen, Satzzeichen irrelevant.
- Daher: nicht-alphanumerische Zeichen entfernen, alles Kleinschrift:

```
clean :: String→ String
clean s
  | null s = ""
  | isAlphaNum (head s) = toLower (head s) : clean (tail s)
  | otherwise = clean (tail s)
```

- Erweiterte Version:

```
palin2 s = palin1 (clean s)
```

DEMO

Fortgeschritten: Vereinfachung von `palin1`

- Das hier ist nicht so schön:

```
palin1 s
  | length s ≤ 1    = True
  | head s == last s = palin1 (init (tail s))
  | otherwise        = False
```

- Was steht da eigentlich:

```
palin1' s = if length s ≤ 1 then True
            else if head s == last s then palin1' (init (tail s))
                                     else False
```

- Damit:

```
palin3 s = length s ≤ 1 || head s == last s && palin3 (init (tail s))
```

  - Terminiert nur wegen Nicht-Striktheit von `||`

Endrekursion

- **Endrekursive** Funktionen verbrauchen keinen Speicherplatz:

Fakultät rekursiv:

```
fac :: Int→ Int
fac n =
  if n ≤ 0 then 1
  else n* fac (n-1)
```

Fakultät **endrekursiv**:

```
fac1 :: Int→ Int→ Int
fac1 n r =
  if n ≤ 0 then r
  else fac1 (n-1) (n*r)
```

```
fac2 n = fac1 n 1
```

- Eine Funktion ist **endrekursiv**, wenn über dem rekursiven Aufruf nur Fallunterscheidungen sind.

DEMO

Suche in einer Zeichenkette

- Endrekursiv:

```
count3 c s = count3' c s 0
count3' c s r =
  if null s then r
  else count3' c (tail s) (if head s == c then 1+r else r)
```

- Endrekursiv mit lokaler Definition

```
count4 c s = count4' s 0 where
  count4' s r =
    if null s then r
    else count4' (tail s) (if head s == c then 1+r else r)
```


Zusammenfassung

- **Bedeutung** von Haskell-Programmen:
  - Auswertungsrelation $\rightarrow$
  - Auswertungsstrategien: innermost-first, outermost-first
  - Auswertungsstrategie für terminierende Programme irrelevant
- **Striktheit**
  - Haskell ist **spezifiziert** als nicht-strikt
  - Meist implementiert durch verzögerte Auswertung
- Leben **ohne Variablen**:
  - Rekursion statt Schleifen
  - Funktionsparameter statt Variablen
- Nächste Vorlesung: Datentypen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 3 (01.11.2022): Algebraische Datentypen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Umverteilung in den Tutorien nötig:

Raphael	37	-2
Tede	39 (43)	0
Thomas	17	+22
Alexander	32	-7
Tarek	71	-32
Insgesamt	196	39

- Eintragung der Tutorien in stud.ip (kommt).

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

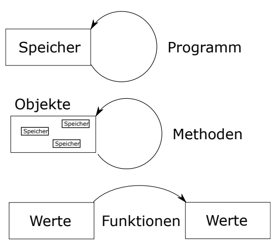
Warum Datentypen?

- Immer nur `Int` ist auch langweilig ...
- **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: $preis : Apfel \to \mathbb{N}$ mit

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

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**):

$$\begin{array}{l} f\ c_1 = e_1 \\ \dots \\ f\ c_n = e_n \end{array} \longrightarrow \begin{array}{l} f\ x = \texttt{case}\ x\ \texttt{of}\ c_1 \to e_1 \\ \qquad\qquad\qquad\quad \dots \\ \qquad\qquad\qquad\quad c_n \to e_n \end{array}$$

- 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

∧	*true*	*false*
true	*true*	*false*
false	*false*	*false*

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

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:

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

☞ Siehe Übung 3.1

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**

```
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:

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

```haskell
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
```

DEMO

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
```

- Alle Artikel:

```haskell
data Artikel =
    Apfel Apfelsorte    | Eier        | Kaese Kaesesorte
  | Schinken            | Salami      | Milch Bio
```

```haskell
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
```

- Preisberechnung

```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:

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

```haskell
preis :: Artikel → Menge→ Preis
```

```haskell
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
```

☞ Siehe Übung 3.3

III. Algebraische Datentypen

Der Allgemeine Fall: Algebraische Datentypen

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

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

Eigenschaften algebraischer Datentypen

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

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
```

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

Rekursive Algebraische Datentypen

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

- 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 = ???
```

- 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: **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
```


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
```

- Verkettung:

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

- Umdrehen:

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


Datentypen und Funktionsdefinition

- Die Definition des **Datentypen** bestimmt wie **Funktionen** auf diesem Datentypen definiert werden können:

Datentyp `T` definiert durch:	Funktionen auf `T` definiert durch:	Beispiel
Aufzählung	Fallunterscheidung	`Apfelsorte`, `Bool`
Produkt	Selektor (pattern matching)	`Artikel`, `Colour`
Rekursion	Rekursion	`Lager`, `Einkaufskorb`, `MyString`

V. Datentypen in Anderen Programmiersprachen

Datentypen in Java

- Speicherverwaltung wie in Haskell (garbage collection)
- Rekursive Typen und Produktypen
- Disjunkte Vereinigung durch Unterklassen

```
abstract class Artikel {
  int preis(Artikel a);
  }

class Eier extends Artikel {
  int preis (Stueck n) {... }
```

```
class Apfel extends Artikel {
  private Apfelsorte s;
  Apfel (Apfelsorte s) { this.s= s; }
  int preis(Menge n) {
    return (s.apreis()* n.stueck());
    }
  }
```

- Sonderfälle:
  - Rekursive Typen mit einem konstanten Konstruktor (bspw. Listen)
  - Reine Aufzählungstypen (nur konstante Konstrukturen, `enum`)

Datentypen in C

- C kennt nur Produkte (`struct`)
- Keine disjunkte Vereinigung
- Rekursion nur über Referenzen (Pointer)
- Leere Liste wird durch `NULL` repräsentiert.
- Konstruktoren müssen selbst implementiert werden
- Keine Speicherverwaltung

```
typedef struct mystring_t {
    char   head;
    struct mystring_t *tail;
    } *mystring_t;
```

```
mystring_t mystring(char head, mystring_t tail)
{
  mystring_t this;
  if ((this= (mystring_t)malloc(sizeof(struct mys
    fprintf(stderr, "Out of memory\n");
    abort();
    }
  this-> head= head; this-> tail= tail;
  return this;
}
```


Zusammenfassung

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

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 4 (08.11.2022): Typvariablen und Polymorphie

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2022/23

Fahrplan

Inhalt

- Letzte Vorlesungen: algebraische Datentypen
- Diese Vorlesung:
  - **Abstraktion** über Typen: Typvariablen und Polymorphie
  - Arten der Polymorphie:
    - Parametrische Polymorphie
    - Ad-hoc Polymorphie
  - Typableitung in Haskell

Lernziele

Wir verstehen, wie in Haskell die Typableitung funktioniert, und was Signaturen wie `head :: [α] → α` und `elem :: Eq α ⇒ α → [α] → Bool` bedeuten.

Ähnliche Datentypen der letzten Vorlesung

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

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

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

- ein **konstanter** Konstruktor
- ein **linear rekursiver** Konstruktor

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb→ Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m+ kasse e
```

```
inventur :: Lager→ Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m+ inventur l
```

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

- ein Fall pro Konstruktor
- **linearer** rekursiver Aufruf

Die Lösung: Polymorphie

Definition (Polymorphie)

Polymorphie ist **Abstraktion über Typen**

Arten der Polymorphie

- **Parametrische** Polymorphie (Typvariablen): Generisch über **alle** Typen
- **Ad-Hoc** Polymorphie (Überladung): Nur für **bestimmte** Typen

Anders als in Java (mehr dazu später).

I. Parametrische Polymorphie

Parametrische Polymorphie: Typvariablen

- **Typvariablen** abstrahieren über Typen

```
data List α = Empty
            | Cons α (List α)
```

- α ist eine **Typvariable**
- `List α` ist ein **polymorpher** Datentyp
- Signatur der Konstruktoren

```
Empty :: List α
Cons  :: α→ List α→ List α
```

- Typvariable α wird bei Anwendung instantiiert

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
`Empty`	`List` α
`Cons 57 Empty`	`List Int`
`Cons 7 (Cons 8 Empty)`	`List Int`
`Cons 'p' (Cons 'i' (Cons '3' Empty))`	`List Char`
`Cons True Empty`	`List Bool`

- Nicht typ-korrekt:

```
Cons 'a' (Cons 0 Empty)
Cons True (Cons 'x' Empty)
```

wegen Signatur des Konstruktors:

```
Cons :: α→ List α→ List α
```


Polymorphe Funktionen

- Parametrische Polymorphie für **Funktionen**:

```
(++) :: List α→ List α→ List α
Empty ++ t        = t
(Cons c s) ++ t = Cons c (s++ t)
```

  - Typvariable vergleichbar mit Funktionsparameter
- Typvariable α wird bei Anwendung instantiiert:

```
Cons 'p' (Cons 'i' Empty) ++ Cons '3' Empty
```

```
Cons 3 Empty ++ Cons 5 (Cons 57 Empty)
```

aber **nicht**

```
Cons True Empty ++ Cons 'a' (Cons 'b' Empty)
```


Beispiel: Der Shop (refaktoriert)

- `Einkaufswagen` und `Lager` als Listen?
- Problem: zwei Typen als Argument

```
type Lager = List (Artikel Menge)
```

  - Geht so **nicht!**
- Lösung: zu einem Typ zusammenfassen

```
data Posten = Posten Artikel Menge
```

- Damit:

```
type Lager = List Posten
type Einkaufskorb = List Posten
```

- **Gleicher** Typ!

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair  :: α→ β→ Pair α β
left  :: Pair α β → α
right :: Pair α β → β
```

Beispielterm	Typ
`Pair 4 'x'`	`Pair Int Char`
`Pair (Cons True Empty) 'a'`	`Pair (List Bool) Char`
`Pair (3+ 4) Empty`	`Pair Int (List α)`
`Cons (Pair 7 'x') Empty`	`List (Pair Int Char)`

☞ Siehe Übung 4.1

II. Vordefinierte Datentypen

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter **syntaktischer Zucker**
- **Listen**

```
data [α] = [] | α : [α]
```

  - Weitere Abkürzungen:
    Listenliterale: `[x]` für `x:[]`, `[x,y]` für `x:y:[]` etc.
    Aufzählungen: `[n .. m]` und `[n, m .. k]` für aufzählbare Typen
- **Tupel** sind das kartesische Produkt

```
data (α, β) = ( fst :: α, snd :: β)
```

  - `(a, b)` = **alle Kombinationen** von Werten aus `a` und `b`
  - Auch n-Tupel: `(a,b,c)` etc. (aber ohne Selektoren)
  - 0-Tupel: `()` (*unit type*, Typ mit genau einem Element)

Vordefinierte Datentypen: Optionen

- Existierende Typen:

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

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

- Instanzen eines **vordefinierten** Typen:

```
data Maybe α = Nothing | Just α
```

- Vordefinierten Funktionen (`import Data.Maybe`):

```
fromJust    :: Maybe α → α        -- partiell
fromMaybe   :: α→ Maybe α→ α
listToMaybe :: [α]→ Maybe α       -- totale Variante von head
maybeToList :: Maybe α→ [α]       -- rechtsinvers zu listToMaybe
```

- Es gilt: `listToMaybe (maybeToList m) = m`
  `length l` $\leq 1 \implies$ `maybeToList (listToMaybe l) = l`

Übersicht: vordefinierte Funktionen auf Listen I

```
(++)          :: [α]→ [α]→ [α]             -- Verkettet zwei Listen
(!!)          :: [α]→ Int→ α               -- n-tes Element selektieren, gezählt ab 0
concat        :: [[α]]→ [α]                -- "flachklopfen"
length        :: [α]→ Int                  -- Länge
head, last    :: [α]→ α                    -- Erstes bzw. letztes Element
tail, init    :: [α]→ [α]                  -- Hinterer bzw. vorderer Rest
replicate     :: Int→ α→ [α]               -- Erzeuge n Kopien
repeat        :: α→ [α]                    -- Erzeugt zyklische Liste
take, drop    :: Int→ [α]→ [α]             -- Erste bzw. letzte n Elemente
splitAt       :: Int→ [α]→ ([α], [α])      -- Spaltet an Index n, gezählt ab 0
reverse       :: [α]→ [α]                  -- Dreht Liste um
zip           :: [α]→ [β]→ [(α, β)]        -- Erzeugt Liste von Paaren
unzip         :: [(α, β)]→ ([α], [β])      -- Spaltet Liste von Paaren
and, or       :: [Bool]→ Bool              -- Konjunktion/Disjunktion
sum, product  :: [Int]→ Int                -- Summe und Produkt (überladen)
```

Vordefinierte Datentypen: Zeichenketten

- String sind Listen von Zeichen:

```
type String = [Char]
```

- Alle vordefinierten Funktionen auf Listen verfügbar.
- **Syntaktischer Zucker** für Stringliterale:

```
"yoho" == ['y','o','h','o'] == 'y':'o':'h':'o':[]
```

- Beispiele:

```
"abc" !! 1  ⇝  'b'
reverse "oof"  ⇝  "foo"
['a','c'..'z']  ⇝  "acegikmoqsuwy"
splitAt 10 "Praktische␣Informatik"  ⇝  ("Praktische","␣Informatik")
```

☞ Siehe Übung 4.2

III. Ad-Hoc Polymorphie

Parametrische Polymorphie: Grenzen

- Eine Funktion f: $\alpha \to \beta$ funktioniert auf **allen** Typen **gleich**.
- Nicht immer der Fall:
  - Gleichheit: `(==) :: α→ α→ Bool`

    Nicht auf allen Typen ist Gleichheit entscheidbar (besonders **Funktionen**)
  - Ordnung: `(≤) :: α→ α→ Bool`

    Nicht auf allen Typen definiert
  - Anzeige: `show :: α→ String`

    Konversion in Zeichenketten höchst divers (Zeichenketten, Listen, Zahlen. . .)

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

Funktion f :: $\alpha \to \beta$ existiert für **mehr als einen**, aber **nicht** für **alle** Typen

- Lösung: **Typklassen**
- Typklassen bestehen aus:
  - **Deklaration** der Typklasse
  - **Instantiierung** für bestimmte Typen
- **Achtung**: hat wenig mit Klassen in Java zu tun

Typklassen: Syntax

- **Deklaration**:

```
class Show α where
  show :: α→ String
```

- **Instantiierung**:

```
instance Show Bool where
  show True  = "Wahr"
  show False = "Falsch"
```


Prominente vordefinierte Typklassen

- Gleichheit: `Eq` für `(==)`
- Ordnung: `Ord` für `(≤)` (und andere Vergleiche)
- Anzeigen: `Show` für `show`
- Lesen: `Read` für `read :: String→ α` (Achtung: Laufzeitefehler!)
- Numerische Typklassen:
  - `Num` für `0, 1, +, -`
  - `Integral` für `quot, rem, div, mod`
  - `Fractional` für `/`
  - `Floating` für `exp, log, sin, cos`

Typklassen in polymorphen Funktionen

- Element einer Liste (vordefiniert):

```
elem :: Eq α ⇒ α→ [α]→ Bool
elem e []     = False
elem e (x:xs) = e == x || elem e xs
```

- Sortierung einer List: `qsort`

```
qsort :: Ord α ⇒ [α]→ [α]
```

- Liste ordnen und anzeigen:

```
showsorted :: (Ord α, Show α)⇒ [α]→ String
showsorted x = show (qsort x)
```


Hierarchien von Typklassen

- Typklassen können andere **voraussetzen**:

```
class Eq α⇒ Ord α where
  (<) :: α→ α→ Bool
  (≤) :: α→ α→ Bool
  a < b = a ≤ b && a ≠ b
```

- **Default**-Definition von `(<)`
- Kann bei Instantiierung überschrieben werden

☞ Siehe Übung 4.3

IV. Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: `Bool`, `Double`
- Funktionstypen: `Double` → `Int` → `Int`, `[Char]` → `Double`
- Typkonstruktoren: `[ ]`, `( ... )`, `Foo`
- Typvariablen
  - `fst` :: $(\alpha, \beta) \to \alpha$
  - `length` :: $[\alpha] \to$ `Int`
  - `(++)` :: $[\alpha] \to [\alpha] \to [\alpha]$
- Typklassen :
  - `elem` :: `Eq` $\alpha \Rightarrow \alpha \to [\alpha] \to$ `Bool`
  - `max` :: `Ord` $\alpha \Rightarrow \alpha \to \alpha \to \alpha$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
  - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

`f m xs = m + length xs`

- `length`: $[\alpha] \to$ `Int`
- `xs`: $[\alpha]$
- `length xs`: `Int`
- `m`: `Int`
- `+`: `Int`

`f` :: `Int` → $[\alpha]$ → `Int`

Typinferenz (nach Hindley-Milner)

- Typinferenz: **Herleitung** des Typen eines Ausdrucks
- Für bekannte Bezeichner wird Typ eingesetzt
- Für Variablen wird allgemeinster Typ angenommen
- Bei der Funktionsanwendung wird **unifiziert**:

`f m xs = m + length xs`

- `m`: α
- `length`: $[\beta] \to$ `Int`
- `xs`: γ; $[\beta]$ $\gamma \mapsto [\beta]$
- `length xs`: `Int`
- `+`: `Int` → `Int` → `Int`
- `m`: `Int` $\alpha \mapsto$ `Int`
- `m +`: `Int` → `Int`
- `m + length xs`: `Int`

`f` :: `Int` → $[\beta]$ → `Int`

Typinferenz

Theorem (Entscheidbarkeit der Typinferenz)

*Die Typinferenz ist **entscheidbar**, und findet immer den **allgemeinsten** Typ, wenn er existiert.*

- Entscheidbarkeit ist nicht alles.
- Grundsätzliche Komplexität ist *DEXPTIME*(n) (deterministisch exponentiell), aber in der Praxis ist das **nie** ein Problem.

DEMO

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

`f x y = (x, 3) : ('f', y) : []`

- `(x, 3)`: α `Int` → $(\alpha,$ `Int`$)$
- `('f', y)`: `Char` β → `(Char,` β`)`
- `[]`: $[\gamma]$
- `('f', y) : []`: `[(Char,` β`)]` $\gamma \mapsto$ `(Char,` β`)`
- `(x, 3) : ('f', y) : []`: `[(Char, Int)]` $\beta \mapsto$ `Int`, $\alpha \mapsto$ `Char`

`f` :: `Char` → `Int` → `[(Char, Int)]`

- Allgemeinster Typ **muss nicht** existieren (Typfehler!)

Und was ist mit Typklassen?

- Typklassen schränken den Typ ein
- Typklassen werden bei der Unifikation **vereinigt**:

`elem` : `Eq` α :: $\alpha \to [\alpha] \to$ `Bool`   `3` : `Num` β :: β

`elem 3` : (`Eq` α, `Num` α) :: $[\alpha] \to$ `Bool`

- Instantiierung muss Typklassen berücksichtigen:

`elem 3` : (`Eq` α, `Num` α) :: $[\alpha] \to$ `Bool`   `"abc"` : `[Char]` $\alpha \mapsto$ `Char`

  - `Char` muss Instanz von `Eq` und `Num` sein.

Typfehler

- Typfehler treten auf, wenn zwei Typen t_1, t_2 nicht **unifiziert** werden können.
- Es gibt drei Arten von Typfehlern:
  1. Typkonstanten nicht unifizierbar: `[True] ++ "a"`
  2. Typ nicht Instanz der geforderten Klasse: `3 + 'a'`
  3. Unifikation gibt **unendlichen** Typ: `x : x`

DEMO

V. Andere Programmiersprachen

Polymorphie in C

- Polymorphie in C: `void *`
- Pointer-to-void ist kompatibel mit allen anderen Pointer-Typen.
- Manueller Typ-Cast nötig
  - Vergl. `Object` in Java
- Extrem Fehleranfälig

Polymorphie in Java

- Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
  - Manuelle Typkonversion nötig, fehleranfällig
- Neu ab Java 1.5: **Generics**
  - Damit **parametrische Polymorphie** möglich
  - **Nachteil:** Benutzung umständlich, weil keine Typherleitung (wegen Kombination mit Subtyping)
  - **Vorteil:** Typkorrektheit sichergestellt
  - Allerdings: Typ-Parameter nur für Klassen, Instanzen nur Objekte.

Ad-Hoc Polymorphie in Java

- `interface` und `abstract class`
- Flexibler in Java: beliebig viele Parameter etc.
- Eingeschränkt durch Vererbungshierarchie
- Ähnliche Standardklassen
  - `toString`
  - `equals` und `==`, keine abgeleitete strukturelle Gleichheit

Polymorphie in Pyton

- In Python werden Typen zur **Laufzeit** geprüft (**dynamic typing**)
- **duck typing**: strukturell gleiche Typen sind gleich
- Polymorphie durch Klassen
- Statt Interfaces kennt Python **Mixins**
  - Abstrakte Klassen ohne Oberklasse

Zusammenfassung

- **Abstraktion** über Typen
  - Uniforme Abstraktion: Typvariable, parametrische Polymorphie
  - Fallbasierte Abstraktion: Überladung, ad-hoc-Polymorphie
- In der Sprache Haskell: **Typvariablen** und **Typklassen**
- Wichtige **vordefinierte** Typen:
  - Listen $[\alpha]$
  - Optionen `Maybe` α
  - Tupel (α, β)

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 (15.11.2022): Funktionen Höherer Ordnung I

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2022/23

Fahrplan

Inhalt

- Funktionen **höherer Ordnung**:
  - Funktionen als gleichberechtigte Objekte
  - Funktionen als Argumente
- Spezielle Funktionen: `map`, `filter`, `fold` und Freunde

Lernziel
Wir verstehen, wie wir mit `map`, `filter` und `fold` wiederkehrende Funktionsmuster kürzer und verständlicher aufschreiben können, und wir verstehen, warum der Funktionstyp in $\alpha \to \beta$ ein Typ wie jeder andere ist.

I. Funktionen als Werte

Funktionen Höherer Ordnung

Slogan
"Functions are first-class citizens."

- Funktionen sind **gleichberechtigt**: Ausdrücke wie **alle anderen**
- **Grundprinzip** der funktionalen Programmierung
- Modellierung **allgemeiner Berechungsmuster**
- Kontrollabstraktion

Ähnliche Datentypen der letzten Vorlesung

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

data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb

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

- ein **konstanter** Konstruktor
- ein **linear rekursiver** Konstruktor

Gelöst durch Polymorphie

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m+ kasse e

inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m+ inventur l

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

Gemeinsamkeiten:
- ein Fall pro Konstruktor
- **linearer** rekursiver Aufruf

Nicht durch Polymorphie gelöst

Gesucht: Einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion ... und **zwei** Instanzen?

```
map f []     = []
map f (c:cs) = f c : map f cs

toL cs = map toLower cs
toU cs = map toUpper cs
```

- **Funktion** `f` als **Argument**
- Was hätte `map` für einen **Typ**?

Funktionen als Werte: Funktionstypen

- Was hätte `map` für einen **Typ**?

```
map f []     = []
map f (c:cs) = f c : map f cs
```

- Was ist der Typ des ersten Arguments? $\alpha \to \beta$
- Was ist der Typ des zweiten Arguments? $[\alpha]$
- Was ist der Ergebnistyp? $[\beta]$
- Alles **zusammengesetzt**:

```
map :: (α → β) → [α] → [β]
```

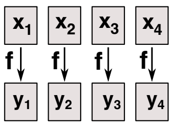
☞ Siehe Übung 5.1

II. Map und Filter

Funktionen als Argumente: map

- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α → β) → [α] → [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



- Auswertung:

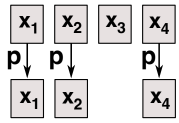
```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
          → 'a':map toLower ('B':[])
          → 'a':toLower 'B':map toLower []
          → 'a':'b':map toLower []
          → 'a':'b':[] ≡ "ab"
```

- **Funktionsausdrücke** werden **symbolisch** reduziert — keine Änderung der Auswertung

Funktionen als Argumente: filter

- Elemente **filtern**: `filter`
- Signatur:

```
filter :: (α → Bool) → [α] → [α]
```



- Definition

```
filter p []    = []
filter p (x:xs)
  | p x        = x: filter p xs
  | otherwise  = filter p xs
```

- Beispiel:

```
digits :: String → String
digits = filter isDigit
```


Beispiel filter: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve' :: [Integer] → [Integer]
sieve' [] = []
sieve' (p:ps) = p: sieve' (filterPs ps) where
   filterPs (q: qs)
      | q `mod` p ≠ 0 = q: filterPs qs
      | otherwise      = filterPs qs
```

- „Sieb" `filterPs`: es werden alle `q` gefiltert mit `mod q p ≠ 0`

Beispiel filter: Sieb des Erathostenes (1. Versuch)

- Es werden alle `q` **gefiltert** mit `mod q p ≠ 0`

```
siev3 :: [Integer] → [Integer]
siev3 [] = []
siev3 (p:ps) = p: siev3 (filter (filterMod p) ps) where
  filterMod p q = q `mod` p ≠ 0
```

- Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
prim3s :: [Integer]
prim3s = siev3 [2..]
```

- **Unschön:** Definition der Hilfsfunktion `filterMod` wird uns „aufgezwungen"

Beispiel filter: Sieb des Erathostenes

- Es werden alle `q` **gefiltert** mit `mod q p ≠ 0`
- Statt Definition eine **namenlose** (anonyme) Funktion `λq → mod q p ≠ 0`

```
sieve :: [Integer] → [Integer]
sieve [] = []
sieve (p:ps) = p: sieve (filter (λq → q `mod` p ≠ 0) ps)
```

- Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
primes :: [Integer]
primes = sieve [2..]
```

- Primzahlzählfunktion $\pi(n)$:

```
pcf :: Integer → Int
pcf n = length (takeWhile (λm → m< n) primes)
```

Primzahltheorem:

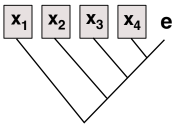
$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log n} = 1$$

☞ Siehe Übung 5.2

III. Strukturelle Rekursion

Strukturelle Rekursion

- **Strukturelle Rekursion**: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)



- Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- Auswertung:

```
sum [4,7,3]       →  4 + 7 + 3 + 0
concat [A, B, C]  →  A ++ B ++ C++ []
length [4, 5, 6]  →  1+ 1+ 1+ 0
```


Strukturelle Rekursion

- **Allgemeines Muster**:

```
f []      = e
f (x:xs) = x ⊗ f xs
```

- Parameter der Definition:
  - Startwert (für die leere Liste) `e :: β`
  - Rekursionsfunktion `⊗ :: α → β→ β`
- Auswertung:

```
f [x1,.., xn] = x1 ⊗ x2 ⊗...⊗ xn ⊗ e
```

- **Terminiert** immer (wenn Liste endlich und ⊗, *e* terminieren)

Strukturelle Rekursion durch `foldr`

- **Stukturelle** Rekursion
  - Basisfall: leere Liste
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Signatur

```
foldr :: (α→ β→ β)→ β→ [α]→ β
```

- Definition

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```


Beispiele: `foldr`

- **Summieren** von Listenelementen.

```
sum :: [Int]→ Int
sum xs = foldr (+) 0 xs
```

- **Flachklopfen** von Listen.

```
concat :: [[a]]→ [a]
concat xs = foldr (++) [] xs
```

- **Länge** einer Liste

```
length :: [a]→ Int
length xs = foldr (λx n→ n+ 1) 0 xs
```


Beispiele: `foldr`

- **Konjunktion** einer Liste

```
and :: [Bool] → Bool
and xs = foldr (&&) True xs
```

- **Konjunktion** von Prädikaten

```
all :: (α→ Bool)→ [α] → Bool
all p xs = and (map p xs)
```


Der Shoppe, revisited.

- Kasse alt:

```
kasse :: Einkaufskorb→ Int
kasse (Ekwg ps) = kasse’ ps where
  kasse’ [] = 0
  kasse’ (p: ps) = cent p+ kasse’ ps
```

- Kasse neu:

```
kasse’ :: Einkaufskorb→ Int
kasse’ (Ek ps) = foldr (λp ps→ cent p+ ps) 0 ps
```

Besser:

```
kasse :: Einkaufskorb→ Int
kasse (Ek ps) = sum (map cent ps)
```


Der Shoppe, revisited.

- Inventur alt:

```
inventur :: Lager→ Int
inventur (Lager ps) = inventur’ ps where
  inventur’ [] = 0
  inventur’ (p: ps) = cent p+ inventur’ ps
```

- Suche nach einem Artikel neu:

```
inventur :: Lager→ Int
inventur (Lager l) = sum (map cent l)
```


Der Shoppe, revisited.

- Suche nach einem Artikel alt:

```
suche :: Artikel→ Lager→ Maybe Menge
suche art (Lager ps) = suche’ art ps where
  suche’ art (Posten lart  m: l)
    | art == lart = Just m
    | otherwise    = suche’ art l
  suche’ art []    = Nothing
```

- Suche nach einem Artikel neu:

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) =
   listToMaybe (map (λ(Posten _ m)→ m)
               (filter (λ(Posten la _) → la == a) ps))
```

Der Shoppe, revisited.

- Kassenbon formatieren neu:

```
kassenbon :: Einkaufskorb→ String
kassenbon ek@(Ek ps) =
  "Bob’s␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "-------------------------------------\n"++
  concatMap artikel ps ++
  "=====================================\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ek))
```

```
artikel :: Posten→ String
```


Iteration mit `foldl`

- `foldr` faltet von rechts:
  `foldr ⊗ [x1,.., xn] e = x1 ⊗ x2 (x2 ⊗ (...(xn ⊗ e)))`
- Warum nicht andersherum?
  `foldl ⊗ [x1,.., xn] e = (((e ⊗ x1) ⊗ x2).) ⊗ xn`
- Definition von `foldl`:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

- `foldl` ist ein **Iterator** mit Anfangszustand `e`, Iterationsfunktion ⊗
- Entspricht einfacher Iteration (`for`-Schleife)

Beispiel: `rev`

- Listen **umdrehen**:

```
rev1 :: [α]→ [α]
rev1 []     = []
rev1 (x:xs) = rev1 xs ++ [x]
```

- Mit `foldr`:

```
rev2 :: [α]→ [α]
rev2 = foldr (λx xs→ xs++ [x]) []
```

- Unbefriedigend: doppelte Rekursion $O(n^2)$!

Beispiel: `rev` revisited

- Listenumkehr **endrekursiv**:

```
rev3 :: [α]→ [α]
rev3 xs = rev0 xs [] where
     rev0 []     ys = ys
     rev0 (x:xs) ys = rev0 xs (x:ys)
```

- Listenumkehr durch falten **von links**:

```
rev4 :: [α]→ [α]
rev4 = foldl (λxs x → x: xs) []
```

```
rev5 :: [α]→ [α]
rev5 = foldl (flip (:)) []
```

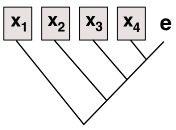
- Nur noch **eine** Rekursion $O(n)$!

`foldr` vs. `foldl`

- `f = foldr ⊗ e` entspricht

```
f []     = e
f (x:xs) = x ⊗ f xs
```

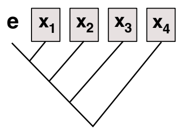
  - **Nicht-strikt** in `xs`, z.B. `and`, `or`
  - Konsumiert nicht immer die ganze Liste
  - Auch für unendliche Listen anwendbar



- `f = foldl ⊗ e` entspricht

```
f xs = g e xs where
  g a []     = a
  g a (x:xs) = g (a ⊗ x) xs
```

  - Effizient (endrekursiv) und **strikt** in `xs`
  - Konsumiert immer die ganze Liste
  - Divergiert immer für unendliche Listen



Wann ist `foldl = foldr`?

Definition (Monoid)

$(\otimes, e)$ ist ein **Monoid** wenn

$$e \otimes x = x \quad \text{(Neutrales Element links)}$$
$$x \otimes e = x \quad \text{(Neutrales Element rechts)}$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \quad \text{(Assoziativät)}$$

Theorem

*Wenn $(\otimes, e)$ **Monoid** und $\otimes$ strikt, dann gilt für alle e, xs*

$$\mathit{foldl} \otimes e\ xs = \mathit{foldr} \otimes e\ xs$$

- Beispiele: `concat`, `sum`, `product`, `length`, `reverse`
- Gegenbeispiel: `all`, `any` (nicht-strikt)

Übersicht: vordefinierte Funktionen auf Listen II

```
map     :: (α→ β)→ [α]→ [β]                  −− Auf alle Elemente anwenden
filter  :: (α→ Bool)→ [α]→ [α]               −− Elemente filtern
foldr   :: (α→ β→ β)→ β→ [α]→ β              −− Falten von rechts
foldl   :: (β→ α→ β)→ β→ [α]→ β              −− Falten von links
mapConcat :: (α→ [β])→ [α]→ [β]              −− map und concat
takeWhile :: (α→ Bool)→ [α]→ [α]             −− längster Prefix mit p
dropWhile :: (α→ Bool)→ [α]→ [α]             −− Rest von takeWhile
span :: (α→ Bool)→ [α]→ ([α], [α])           −− takeWhile und dropWhile
all :: (α → Bool) → [α] → Bool               −− Argument gilt für alle
any :: (α → Bool) → [α] → Bool               −− Argument gilt mind. einmal
elem :: (Eq α) ⇒ α → [α] → Bool              −− Ist Element enthalten?
zipWith :: (α → β → γ) → [α] → [β] → [γ]     −− verallgemeinertes zip
```

- Mehr: siehe `Data.List` ☞ Siehe Übung 5.3

IV. Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

- **Funktionskomposition** (mathematisch)

```
(∘) :: (β→ γ) → (α→ β)→ α→ γ
(f ∘ g) x = f (g x)
```

  - Vordefiniert
  - Lies: `f` nach `g`
- Funktionskomposition **vorwärts**:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(f >.> g) x = g (f x)
```

  - **Nicht** vordefiniert

η-Kontraktion

- "`>.>` ist dasselbe wie ∘ nur mit vertauschten Argumenten"
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- **Da fehlt doch was?!** Nein:

`(>.>) f g a = flip (∘) f g a` ≡ `(>.>) = flip (∘)`

- Warum? η-Kontraktion

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \to \beta \to \gamma \equiv \alpha \to (\beta \to \gamma)$$

  - **Inbesondere:** $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$
- Funktionsanwendung ist linksassoziativ:

`f a b ≡ (f a) b`

  - **Inbesondere:** `f (a b) ≢ (f a) b`
- **Partielle** Anwendung von Funktionen:
  - Für `f` :: $\alpha \to \beta \to \gamma$, `x` :: α ist `f x` :: $\beta \to \gamma$
- Beispiele:
  - `map toLower :: String→ String`
  - `(3 ==) :: Int→ Bool`
  - `concat ∘ map (replicate 2) :: String→ String`

V. Andere Programmiersprachen

Funktionen höherer Ordnung in C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list map(void *f(void *x), list l);

extern list filter(int f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: `qsort` (C-Standard 7.20.5.2)

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```


Funktionen höherer Ordnung in Java

- **Java**: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist **nicht** möglich:

```
interface Collection {
  Object fold(Object f(Object a, Collection c), Object a); }
```

- Aber folgendes:

```
interface Foldable { Object f (Object a); }

interface Collection { Object fold(Foldable f, Object a); }
```

- Vergleiche `Iterator` aus Collections Framework (Java SE 6):

```
public interface Iterator<E> {
  boolean hasNext();
  E next(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

Zusammenfassung

- Funktionen **höherer Ordnung**
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Spezielle Funktionen höherer Ordnung: `map`, `filter`, `fold` und Freunde
- Formen der **Rekursion**:
  - Strukturelle Rekursion entspricht `foldr`
  - Iteration entspricht `foldl`
- Partielle Applikation, η-Äquivalenz, namenlose Funktionen
- Nächste Woche: Rekursive und zyklische Datenstrukturen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 (22.11.2022): Rekursive Datenstrukturen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Die Vorlesung am **06.12.2022** findet im **NW2 A0242** statt.

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

- **Rekursive** Datentypen und **zyklische** Daten
  - ... und wozu sie nützlich sind
  - Fallbeispiel: Labyrinth
- Effizienzerwägungen

Lernziele

1. Wir verstehen, wie in Haskell „unendliche“ Datenstrukturen modelliert werden. Warum sind unendliche Listen nicht wirklich unendlich?
2. Wir wissen, worauf wir achten müssen, wenn uns die Geschwindigkeit unser Haskell-Programme wichtig ist.

Konstruktion zyklischer Datenstrukturen

- **Zyklische** Datenstrukturen haben keine **endliche freie** Repräsentation
  - Nicht durch endlich viele Konstruktoren darstellbar
  - Sondern durch Konstruktoren und **Gleichungen**
- Einfaches Beispiel:

```
ones = 1 : ones
```

- Nicht-Striktheit erlaubt einfache Definition von Funktionen auf zyklische Datenstrukturen
- Aber: Funktionen können **divergieren**

I. Vorteile der Nicht-Strikten Auswertung

Zyklische Listen

- Durch Gleichungen können wir **zyklische** Listen definieren.

```
nats :: [Integer]
nats = natsfrom 0 where
  natsfrom i = i: natsfrom (i+1)
```

- Repräsentation durch endliche, zyklische Datenstruktur
  - Kopf wird nur einmal ausgewertet.

```
fives :: [Integer]
fives = trace "***␣Foo!␣***" 5 : fives
```

DEMO

- Es gibt keine **unendlichen** Listen, es gibt nur Berechnungen von Listen, die nicht terminieren.

Unendliche Weiten?

- Verschiedene Ebenen:
  - Mathematisch — unendliche Strukturen (natürliche Zahlen, Listen)
  - Implementierung — immer endlich (kann unendliche Strukturen **repräsentieren**)
- Berechnungen auf unendlichen Strukturen: Vereinigung der Berechnungen auf allen **endlichen** Teilstrukturen
- Jede Berechnung hat **endlich** viele Parameter.
  - Daher nicht entscheidbar, ob Liste „unendlich“ (zyklisch) ist:

```
isCyclic :: [a]→ Bool
```

Unendliche Listen und Nicht-Striktheit

- Nicht-Striktheit macht den Umgang mit zyklischen Datenstrukturen einfacher
- Beispiel: Sieb des Eratosthenes:

```
sieve :: [Integer]→ [Integer]
sieve [] = []
sieve (p:ps) = p: sieve (filter (λq→ q ‘mod‘ p ≠ 0) ps)
```

  - Bis wo muss ich sieben, um die ersten n-Primzahlen zu berechnen?

```
n_primes :: Int→ [Integer]
n_primes n = sieve [2.. ???]
```

  - Einfacher: Liste **aller** Primzahlen berechnen, davon n-te selektieren.

```
n_primes :: Int→ [Integer]
n_primes n = take n (sieve [2.. ])
```


Fibonacci-Zahlen

- Aus der Kaninchenzucht.
- Sollte jeder Informatiker kennen.

```
fib1 :: Integer→ Integer
fib1 0 = 1
fib1 1 = 1
fib1 n = fib1 (n-1)+ fib1 (n-2)
```

- Problem: **exponentieller Aufwand**.

Fibonacci-Zahlen

- Lösung: zuvor berechnete **Teilergebnisse wiederverwenden**.
- Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
           fibs ⇝ [1, 1, 2, 3,  5,  8, 13, 21, 34, 55 .. ]
      tail fibs ⇝ [1, 2, 3, 5,  8, 13, 21, 34, 55 .. ]
tail (tail fibs) ⇝ [2, 3, 5, 8, 13, 21, 34, 55...]
```

- Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- n-te Fibonaccizahl mit `fibs !! n`:

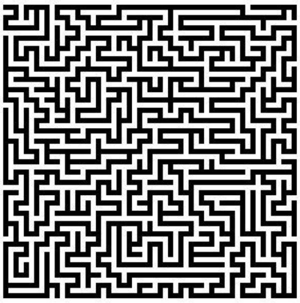
```
fib2 :: Integer→ Integer
fib2 n = genericIndex fibs n
```

- **Aufwand**: **linear**, da `fibs` nur einmal ausgewertet wird.

☞ Siehe Übung 6.1

II. Zyklische Datenstrukturen

Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

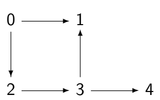
Modellierung eines Labyrinths

- Ein **gerichtetes** Labyrinth ist entweder
  - eine Sackgasse,
  - ein Weg, oder
  - eine Abzweigung in zwei Richtungen.
- Jeder Knoten im Labyrinth hat ein Label α.

```
data Lab α = Dead α
           | Pass α (Lab α)
           | TJnc α (Lab α) (Lab α)
```


Definition von Labyrinthen

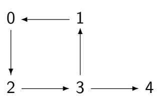
Ein einfaches Labyrinth ohne Zyklen:



Definition in Haskell:

```
s0 = TJnc 0 s1 s2
s1 = Dead 1
s2 = Pass 2 s3
s3 = TJnc 3 s1 s4
s4 = Dead 4
```

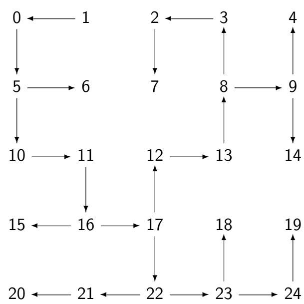
Ein einfaches Labyrinth mit Zyklen:



Definition in Haskell:

```
t0 = Pass 0 t2
t1 = Pass 1 t0
t2 = Pass 2 t3
t3 = TJnc 3 t1 t4
t4 = Dead 4
```


Ein Labyrinth (zyklenfrei)



Traversion des Labyrinths

- Ziel: **Pfad** zu einem gegeben **Ziel** finden
- Benötigt Pfade und Traversion
- Pfade: Liste von Knoten

```
type Path α = [α]
```

- Traversion: erfolgreich (Pfad) oder nicht erfolgreich

```
type Trav α = Maybe [α]
```


Traversionsstrategie

- Geht erstmal von **zyklenfreien** Labyrinth aus
- An jedem Knoten prüfen, ob Ziel erreicht, ansonsten
  - an Sackgasse: Fehlschlag (`Nothing`)
  - an Passagen: Weiterlaufen

```
cons :: α → Trav α → Trav α
cons _ Nothing      = Nothing
cons i (Just is) = Just (i: is)
```

  - an Kreuzungen: Auswahl treffen

```
select :: Trav α → Trav α→ Trav α
select Nothing t = t
select t     _ = t
```

- Erfordert Propagation von Fehlschlägen (in `cons` und `select`)

Zyklenfreie Traversion

- Zusammengesetzt:

```
traverse_1 :: (Show α, Eq α)⇒ α→ Lab α→ Trav α
traverse_1 t l
  | nid l == t = Just [nid l]
  | otherwise = case l of
    Dead _ → Nothing
    Pass i n → cons i (traverse_1 t n)
    TJnc i n m → cons i (select (traverse_1 t n)
                               (traverse_1 t m))
```

DEMO

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

Traversion mit Zyklen

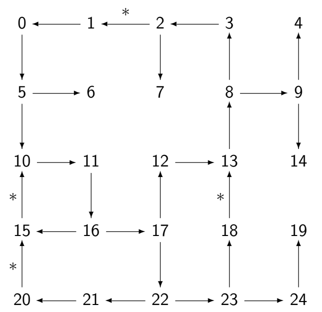
- Veränderte **Strategie**: Pfad bis hierher übergeben
  - Pfad muss hinten erweitert werden ($O(n)$)
  - Besser: Pfad **vorne** erweitern ($O(1)$), am Ende umdrehen
- Wenn aktueller Knoten in bisherigen Pfad enthalten ist, Fehlschlag
- Ansonsten wie oben

Traversion mit Zyklen

```
traverse_2 :: Eq α⇒ α→ Lab α→ Trav α
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
      Dead _ → Nothing
      Pass i n → trav_2 n (i: p)
      TJnc i n m → select (trav_2 n (i: p)) (trav_2 m (i: p))
```

- Kritik:
  - Prüfung `elem` immer noch $O(n)$
  - Abhilfe: **Menge** der besuchten Knoten getrennt von aufgebautem **Pfad**
  - Erfordert effiziente Datenstrukturen für Mengen (`Data.Set`, `Data.IntSet`) ⟶ später

Ein Labyrinth (mit Zyklen)



Der allgemeine Fall: variadische Bäume

- Labyrinth ⟶ **Graph** oder **Baum**
- Labyrinth mit mehr als 2 Nachfolgern: **variadischer Baum**
  - Kürzere Definition erlaubt einfachere Funktionen:

```
traverse :: Eq α⇒ α→ VTree α→ Maybe [α]
traverse t vt = trav [] vt where
  trav p (NT l vs)
    | l == t = Just (reverse (l: p))
    | elem l p = Nothing
    | otherwise = select (map (trav (l: p)) vs)
```

DEMO

Traversion verallgemeinert

- Änderung der Parameter der Traversionsfunktion `trav`:

```
trav :: Eq α⇒ [(VTree α, [α])]→ Maybe [α]
```

  - Liste der nächsten **Kandidaten** mit **Pfad** der dorthin führt.
  - Algorithmus:
    1. Wenn Liste leer, Fehlschlag
    2. Wenn Liste nicht leer, ist der aktuelle Knoten der Kopf der Liste.
    3. Prüfe, ob aktueller Knoten das Ziel ist.
    4. Wenn nicht am Ziel und aktueller Knoten schon besucht, nächsten Kandidaten traversieren
    5. Ansonsten füge Kinder des aktuellem Knotens mit aktuellem Pfad zu Kandidaten hinzu und traversiere weiter
- Tiefensuche: Kinder **vorne** anfügen (Kandidatenliste ist ein **Stack**)
- Breitensuche: Kinder **hinten** anhängen (Kandidatenliste ist eine **Queue**)
- Andere Bewertungen möglich

Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:

$$\begin{tikzcd} 0 \arrow[r] \arrow[d] & 1 \arrow[d] \\ 3 & 2 \arrow[l] \arrow[ul] \end{tikzcd}$$

Definition in Haskell:

```
l00 = NT 0 [l01, l03]
l01 = NT 1 [l02]
l02 = NT 2 [l00, l03]
l03 = NT 3 [l00]
```

- Gesucht: Pfad von 0 zu 3
  - Tiefensuche: `[0, 1, 2, 3]`
  - Breitensuche: `[0, 3]`

Tiefensuche

```
depth_first_search :: Eq α⇒ α→ VTree α→ Maybe [α]
depth_first_search t vt = trav [(vt, [])] where
  trav [] = Nothing
  trav ((NT l ch, p):rest)
    | l == t    = Just (reverse (l:p))
    | elem l p  = trav rest
    | otherwise = trav (more++ rest) where
                    more = map (λc→ (c, l: p)) ch
```


Breitensuche

```
breadth_first_search :: Eq α⇒ α→ VTree α→ Maybe [α]
breadth_first_search t vt = trav [(vt, [])] where
  trav [] = Nothing
  trav ((NT l ch, p):rest)
    | l == t    = Just (reverse (l:p))
    | elem l p  = trav rest
    | otherwise = trav (rest ++ more) where
                    more = map (λc→ (c, l: p)) ch
```

☞ Siehe Übung 6.2

III. Effizienzerwägungen

Beispiel: Listen umdrehen

- Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a]→ [a]
rev' []     = []
rev' (x:xs) = rev' xs ++ [x]
```

  - Hängt auch noch hinten an — $O(n^2)$!
- Liste umdrehen, **endrekursiv** und $O(n)$:

```
rev :: [a]→ [a]
rev xs = rev0 xs [] where
    rev0 []     ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```

- Schneller weil geringere Aufwandsklasse, nicht nur wg. Endrekursion
- Frage: ist Endrekursion immer schneller?

Beispiel: Fakultät

- Fakultät **nicht** endrekursiv:

```
fac1 :: Integer→ Integer
fac1 n  = if n == 0 then 1 else n ∗ fac1 (n−1)
```

- Fakultät endrekursiv:

```
fac2 :: Integer→ Integer
fac2 n     = fac' n 1 where
  fac' :: Integer→ Integer→ Integer
  fac' n acc = if n == 0 then acc
               else fac' (n−1) (n∗acc)
```

- `fac1` verbraucht Stack, `fac2` nicht.
- Ist nicht merklich schneller?!

Verzögerte Auswertung und Speicherlecks

- **Garbage collection** gibt unbenutzten Speicher wieder frei.
  - **Unbenutzt**: Bezeichner nicht mehr Speicher im erreichbar
- Verzögerte Auswertung **effizient**, weil nur bei Bedarf ausgewertet wird
  - Aber Achtung: Speicherleck!
- Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
  - "Echte" Speicherlecks wie in C/C++ nicht möglich.
- Beispiel: `fac2`
  - Zwischenergebnisse werden nicht auswertet.
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- **Strikte Argumente** erlauben Auswertung **vor** Aufruf
  - Dadurch **konstanter** Platz bei **Endrekursion**.
- Erzwungene Striktheit: `seq :: α→ β→ β`

```
 ⊥ 'seq' b  =  ⊥
a 'seq' b  =  b
```

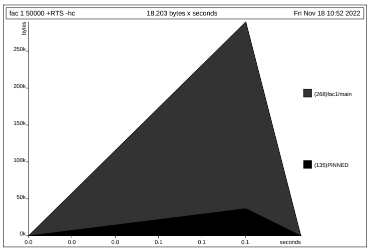
  - `seq` vordefiniert (nicht in Haskell definierbar)
  - `($!) :: (a→ b)→ a→ b` strikte Funktionsanwendung

```
f $! x = x 'seq' f x
```

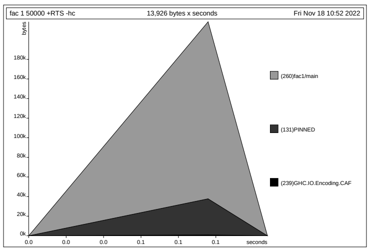
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```
fac3 :: Integer→ Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc (if n == 0 then acc
                         else fac' (n−1) (n∗acc))
```

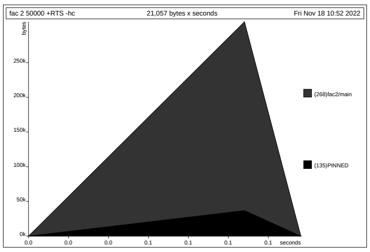
Speicherprofil: `fac1 50000`, nicht optimiert



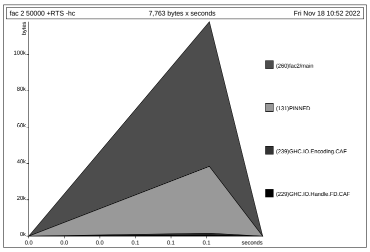
Speicherprofil: `fac1 50000`, optimiert



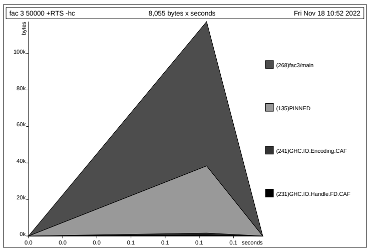
Speicherprofil: `fac2 50000`, nicht optimiert



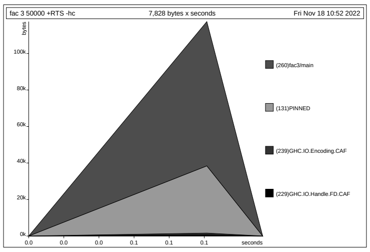
Speicherprofil: `fac2 50000`, optimiert



Speicherprofil: `fac3 50000`, nicht optimiert



Speicherprofil: `fac3 50000`, optimiert



Fakultät als Funktion höherer Ordnung

- Nicht end-rekursiv mit `foldr`:

```
fac_foldr :: Integer→ Integer
fac_foldr i = foldr (∗) 1 [1.. i]
```

- End-rekursiv mit `foldl`:

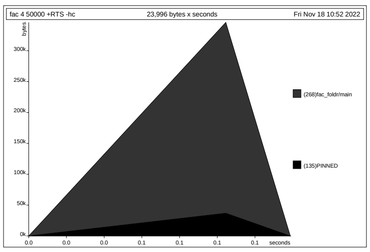
```
fac_foldl :: Integer→ Integer
fac_foldl i = foldl (∗) 1 [1.. i]
```

- End-rekursiv und strikt mit `foldl’`:

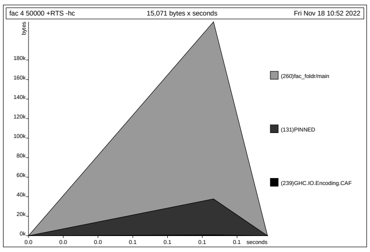
```
fac_foldl’ :: Integer→ Integer
fac_foldl’ i = foldl’ (∗) 1 [1.. i]
```

- **Exakt** die gleichen Ergebnisse!

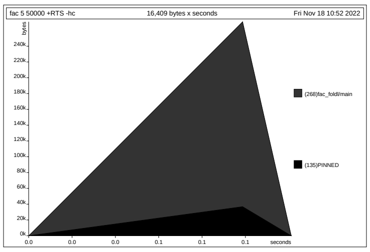
Speicherprofil: `foldr 50000`, nicht optimiert



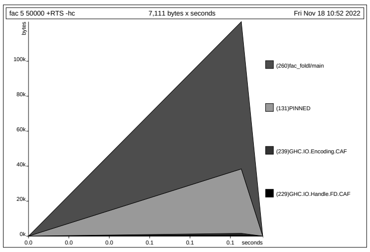
Speicherprofil: `foldr 50000`, optimiert



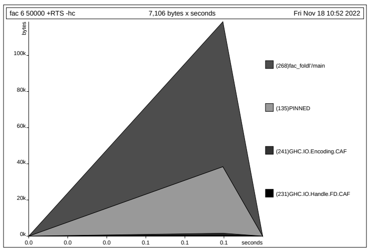
Speicherprofil: `foldl 50000`, nicht optimiert



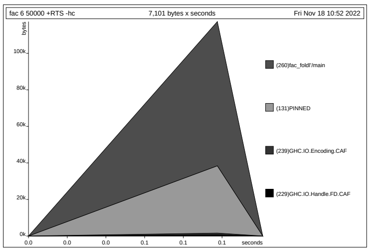
Speicherprofil: `foldl 50000`, optimiert



Speicherprofil: `foldl’ 50000`, nicht optimiert



Speicherprofil: `foldl’ 50000`, optimiert



Fazit Speicherprofile

- Endrekursion **nur** bei **strikten Funktionen** schneller
- Optimierung des *ghc*
  - Meist ausreichend für Striktheitsanalyse
  - Aber **nicht** für Endrekursion
- Deshalb:
  - **Manuelle** Überführung in Endrekursion **sinnvoll**
  - **Compiler-Optimierung** für Striktheit nutzen

Zusammenfassung

- Rekursive Datentypen können **zyklische Datenstrukturen** modellieren
  - Das Labyrinth — Sonderfall eines **variadischen Baums**
  - Unendliche Listen — nützlich wenn Länge der Liste nicht im voraus bekannt
- Effizienzerwägungen:
  - Überführung in Endrekursion sinnvoll, Striktheit durch Compiler

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 (29.11.2020): Funktionen Höherer Ordnung II

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Die Vorlesung **nächste Woche** findet im **NW2 A0242** statt.

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

Heute

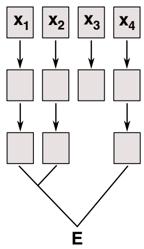
- Mehr über `map` und `fold`
- `map` und `fold` sind nicht nur für Listen

Lernziel
Wir verstehen, warum `map` und `fold` besonders sind, wie sie für andere Datentypen aussehen, und wann wir sie benutzen können.

I. Berechnungsmuster

`map` und `filter` als Berechnungsmuster

- `map`, `filter`, `fold` als Berechnungsmuster:
  1. Anwenden einer Funktion auf **jedes** Element der Liste
  2. möglicherweise **Filtern** bestimmter Elemente
  3. **Kombination** der Ergebnisse zu Endergebnis `E`
- Gut parallelisierbar, skalierbar
- Berechnungsmuster für große Datenmengen
  - Map/Reduce (Google), Hadoop



Listenkomprehension

- Besondere Notation: Listenkomprehension
  `[ f x | x← as, g x ] ≡ map f (filter g as)`
- Beispiel:
  - Remember this?

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) =
  listToMaybe (map (λ(Posten _ m)→ m)
              (filter (λ(Posten la _) → la == a) ps))
```

  - Sieht so besser aus:

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) = listToMaybe [ m | Posten la m ← ps, la == a ]
```


Listenkomprehension mit mehreren Generatoren

- Anderes Beispiel: Primzahlzwillinge

```
twin_primes :: [(Integer, Integer)]
twin_primes = [(x, y) | (x, y) ← zip primes (tail primes), x+2 == y]
```

- Mit mehreren Generatoren werden **alle Kombinationen** generiert:

```
idx :: [String]
idx = [ a: show i| a← ['a'.. 'z'], i← [0.. 9]]
```

Beispiel I: Quicksort

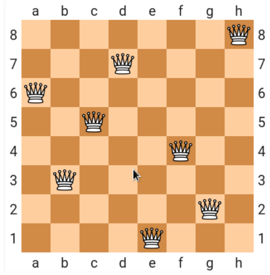
- Quicksort per Listenkomprehension:

```
qsort1 :: Ord α⇒ [α]→ [α]
qsort1 [] = []
qsort1 xs@(x:_) = qsort1 [y | y← xs, y< x ]++
                  [x0| x0← xs, x0 == x ]++
                  qsort1 [z | z← xs,  z> x ]
```

- Erstaunlich effizient
- Einfache Rekursion mit 3-Weg-Split effizienter, aber wesentlich länger
- Grund: Sortierte Liste wird nicht im ganzen aufgebaut

Beispiel II: 8-Damen-Problem

- Problem: Plaziere 8 Damen sicher auf einem Schachbrett



Source: Wikipedia

Beispiel II: n-Damen-Problem

- Position der Königinnen:

```
type Pos = (Int, Int)
type Board = [Pos]
```

- Rekursiv: Lösung für $n-1$ Königinnen, n-te sicher dazu positionieren

```
queens :: Int→ [Board]
queens n = qu n where
  qu :: Int→ [Board]
  qu i | i == 0    = [[ ]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                      safe p (i, j)]
```

- Invariante: n-te Königin in n-ter Spalte

Beispiel II: n-Damen-Problem

- Wann ist eine Königin sicher?

```
safe :: Board→ Pos→ Bool
safe others nu = and [ not (threatens other nu) | other ← others ]
```

- Bedrohung: gleiche Zeile oder Diagonale

```
threatens :: Pos→ Pos→ Bool
threatens (i, j) (m, n) = (j== n) || (i+j == m+n) || (i-j == m-n)
```

  - Diagonalen charakterisiert durch $y = a + x$ bzw. $y = a - x$ für konstantes a
  - Gleiche Spalte (`i == m`) durch Konstruktion ausgeschlossen

☞ Siehe Übung 7.1

II. Map und Fold: Jenseits der Listen

`map` als strukturerhaltende Abbildung

`map` ist die kanonische **strukturerhaltende Abbildung**

- Für `map` gelten folgende Aussagen:

```
map id = id
map f ∘ map g = map (f ∘ g)
length ∘ map f = length
```

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeineren?

⟶ Typklassen?Konstruktorklassen!

Funktoren

- **Konstruktorklassen** sind Typklassen für Typkonstruktoren.
- Die Konstruktorklasse `Functor` für alle Typen mit einer stukturerhaltenden Abbildung:

```
class Functor f where
  fmap :: (α→ β) → f α → f β
```

- Es sollte gelten (kann nicht geprüft werden):

```
fmap id = id
fmap f ∘ fmap g = fmap (f ∘ g)
```

- Infix-Synomym `<$>` für `fmap`

Instanzen von `Functor`

- Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`
- `Maybe` ist eine Instanz von `Functor`:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing  = Nothing
```

  - Propagiert `Nothing` — oft sehr nützlich
- Tupel sind Instanzen von `Functor` im **zweiten** Argument, bspw:

```
instance Functor (a, ) where
  fmap f (a, b) = (a, f b)
```

`foldr` ist kanonisch

`foldr` ist die **kanonische strukturell rekursive** Funktion.

- Alle strukturell rekursiven Funktionen sind als Instanz von `foldr` darstellbar
- Insbesondere auch `map` und `filter` — ☞ Siehe Übung 7.3
- Es gilt: `foldr (:) [] = id`
- Jeder algebraischer Datentyp hat ein `foldr`
- Nicht als Konstrukturklasse darstellbar (wie `Functor` und `fmap`)
  - Anmerkung: Typklasse `Foldable` schränkt Signatur von `foldr` ein

`fold` für andere Datentypen

`fold` ist universell
Jeder algebraische Datentyp `T` hat genau ein `foldr`.

- Kanonische Signatur für `T`:
  - Pro Konstruktor `C` ein Funktionsargument f_C
  - Freie Typvariable β für `T`
- Kanonische Definition:
  - Pro Konstruktor `C` eine Gleichung
  - Gleichung wendet f_C auf Argumente an (und `fold` rekursiv auf Argumente vom Typ `T`)

`fold` für andere Datentypen

- Beispiel:

```
data IL = Cons Int IL | Err String | Mt
```

- Das Fold dazu:

```
foldIL :: (Int→ β→ β)→ (String→ β)→ β→ IL→ β
foldIL f e a (Cons i il) = f i (foldIL f e a il)
foldIL f e a (Err str)   = e str
foldIL f e a Mt          = a
```

- Was ist das?
  - Eine Art Listen von `Int` mit Fehlern („Ausnahmen")
  - Das zweite Argument von `foldIL` fängt aufgetretene Ausnahmen

`fold` für bekannte Datentypen

- `Bool`: Fallunterscheidung:

```
data Bool = False | True

foldBool :: β→ β→ Bool→ β
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

- `Maybe α`: Auswertung

```
data Maybe α = Nothing | Just α

foldMaybe :: β→ (α→ β)→ Maybe α→ β
foldMaybe b f Nothing  = b
foldMaybe b f (Just a) = f a
```

  - Als `maybe` vordefiniert

`fold` für bekannte Datentypen

- Tupel: die `uncurry`-Funktion

```
data (α, β) = (α, β)

foldPair :: (α→ β→ γ)→ (α, β)→ γ
foldPair f (a, b)= f a b
```

- Dazu gehört die Funktion `curry` (beide vordefiniert):

```
curry :: ((α, β)→ γ)→ α→ β→ γ
curry f a b = f (a, b)
```

- Die beiden sind **invers**:

`uncurry ∘ curry = id`   `curry ∘ uncurry = id`

`fold` für bekannte Datentypen

- Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat

foldNat :: β→ (β→ β)→ Nat→ β
foldNat e f Zero  = e
foldNat e f (Succ n) = f (foldNat e f n)
```

- Wendet Funktion `f` `n`-mal auf Startwert `e` an:

`foldNat` $e\ f\ n = f^n(e)$

- Konversion nach `Int`:

```
natToInt :: Nat→ Int
natToInt = foldNat 0 (1+)
```

☞ Siehe Übung 7.2

`fold` für binäre Bäume

- Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

  - Label **nur** in den Knoten
- Instanz von `fold`:

```
foldT :: β→ (α→ β→ β→ β)→ Tree α→ β
foldT e f Mt = e
foldT e f (Node a l r) = f a (foldT e f l) (foldT e f r)
```

- Instanz von `Functor`, kein (offensichtliches) Filter

```
instance Functor Tree where
  fmap f Mt = Mt
  fmap f (Node a l r)= Node (f a) (fmap f l) (fmap f r)
```


Funktionen mit `foldT`

- Höhe des Baumes berechnen:

```
height :: Tree α→ Int
height = foldT 0 (λ_ l r→ 1+ max l r)
```

- Inorder-Traversion der Knoten:

```
inorder :: Tree α→ [α]
inorder = foldT [] (λa l r→ l++ [a]++ r)
```

- Enthält der Baum dieses Element?

```
isElem :: Eq α⇒ α→ Tree α→ Bool
isElem a = foldT False (λb l r→ a == b || l || r)
```

  - Nich-Striktheit von `||` begrenzt Traversion

Kanonische Eigenschaften von `foldT` und `fmap`

- Auch hier gilt:

$$\texttt{foldT Mt Node} = \texttt{id}$$

$$\texttt{fmap id} = \texttt{id}$$

$$\texttt{fmap f} \circ \texttt{fmap g} = \texttt{fmap (f} \circ \texttt{g)}$$

- Gilt für **alle** Datentypen. Insbesondere gilt:

$$\texttt{fold } C_1\ C_2 \ldots C_n = \texttt{id}$$

Falten mit den Konstruktoren ergibt die Identität.

Variadische Bäume

- Das Labyrinth ist ein variadischer Baum:

```
data VTree α = NT α [VTree α]
```

- Auch hierfür `fold` und `map`:

```
foldT :: (α→ [β]→ β)→ VTree α→ β
foldT f (NT a ns) = f a (map (foldT f) ns)
```

```
instance Functor VTree where
  fmap f (NT a ns) = NT (f a) (map (fmap f) ns)
```


Suche im Labyrinth

- Tiefensuche via `foldT`

```
dfs1 :: VTree α→ [Path α]
dfs1 = foldT add where
  add a [] = [[a]]
  add a ps = [ a:p | p ← concat ps]
```

```
dfs2 :: Eq α⇒ VTree α→ [Path α]
dfs2 = foldT add where
  add a [] = [[a]]
  add a ps = [a:p | p ← concat ps, not (a ‘elem‘ p) ]
```

- Problem:
  - `foldT` terminiert **nicht** für **zyklische** Strukturen
  - Auch nicht, wenn `add` prüft ob `a` schon enthalten ist
  - Pfade werden vom **Ende** konstruiert

DEMO

Grenzen von `foldr`

- `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
  - Beispiel: `all = foldr (&&) True`
  - Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- `foldl` ist **nicht** generalisierbar
  - Warum? Nur für **linear rekursive** Typen

Andere Arten der Rekursion

- Andere rekursive Struktur über Listen
  - Quicksort: **baumartige** Rekursion
- Rekursion nicht (nur) über Listenstruktur:
  - `take`: Begrenzung der Rekursion

```
take :: Int→ [α]→ [α]
take n _      | n ≤ 0 = []
take _ []             = []
take n (x:xs)         = x : take (n-1) xs
```

  - Version mit `fold` divergiert für nicht-endliche Listen

☞ Siehe Übung 7.4

III. Funktionen höher Ordnung in anderen Programmiersprachen

Funktionen höherer Ordnung in Python

- Python kennt map, filter, fold:

```
letters = map(chr, range(97, 123))
```

  - Map auf Iteratoren definiert, nicht auf Listen
- Python kennt Listenkomprehension:

```
idx  = [ x+ str(i) for x in letters for i in range(10) ]
```

- Python kennt Lambda-Ausdrücke:

```
num  = map (lambda x: 3*x+1, range (1,10))
```


Zusammenfassung

- Einge Funktionen höherer Ordnung sind speziell:
  - `map` ist die strukturerhaltende Funktion
  - `fold` ist die strukturelle Rekursion über dem Typen
- Jeder Datentyp hat `map` und `fold`
- Konstruktorklassen sind Klassen für Typkonstruktoren
  - Beispiel `Functor`
- Listenkomprehension ist ein nützlicher, leichtgewichtiger syntaktischer Zucker für `map` und `filter`
- Die Vorlesung **nächste Woche** findet im **NW2 A0242** statt.

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 (06.12.2022): Abstrakte Datentypen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Abgabe des 7. Übungsblattes in Gruppen zu **drei** Studenten.
  - Bitte **jetzt** eine Gruppe suchen!
- Morgen ist **Tag der Lehre**.
  - Tutorien sind **freiwillig**.

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- **Teil II: Funktionale Programmierung im Großen**
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- **Abstrakte Datentypen**
  - Allgemeine Einführung
  - Realisierung in Haskell
  - Beispiele

Lernzielen
Wir wollen verstehen, wie und warum wir Datentypen verkapseln.

I. Modularisierung und Abstrakte Datentypen

Warum Modularisierung?

- Übersichtlichkeit der Module — **Lesbarkeit**
- Getrennte Übersetzung — **technische** Handhabbarkeit
- Verkapselung — **konzeptionelle** Handhabbarkeit

Abstrakte Datentypen

Definition (Abstrakter Datentyp)
Ein **abstrakter Datentyp** (ADT) besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:
1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**
3. Einhaltung von **Invarianten** über dem Typ kann garantiert werden

Implementation von ADTs in einer Programmiersprache:
- benötigt Möglichkeit der **Kapselung** (Einschränkung der Sichtbarkeit)
- bspw. durch Module oder Objekte

ADTs vs. algebraische Datentypen

- Algebraische Datentypen
  - **Frei erzeugt** durch **Konstruktoren**
  - Keine Einschränkungen
  - Insbesondere keine Gleichheiten der Konstruktoren (`[]` $\neq$ `x:xs`, `x:ls` $\neq$ `y:ls` etc.)
- ADTs:
  - Keine ausgezeichneten Konstruktoren
  - Einschränkungen und Invarianten möglich
  - Gleichheiten möglich

ADTs vs. Objekte

- ADTs (z.B. Haskell): **Typ** plus **Operationen**
- Objekte (z.B. Java): **Interface**, **Methoden**.
- **Gemeinsamkeiten**:
  - Verkapselung (information hiding) der Implementation
- **Unterschiede**:
  - Objekte haben **internen Zustand**, ADTs sind **referentiell transparent**;
  - Objekte haben Konstruktoren, ADTs nicht
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt
  - Java: `packages` für Sichtbarkeit

ADTs in Haskell: Module

- Einschränkung der Sichtbarkeit durch **Verkapselung**
- **Modul**: Kleinste verkapselbare Einheit
- Ein **Modul** umfaßt:
  - **Definitionen** von Typen, Funktionen, Klassen
  - **Deklaration** der nach außen **sichtbaren** Definitionen
- Gleichzeitig: Modul $\triangleq$ Übersetzungseinheit (getrennte Übersetzung)

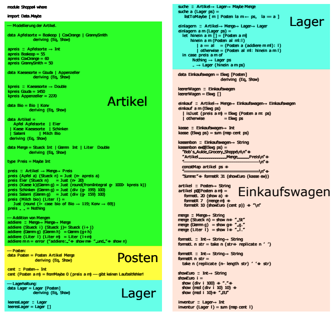
Module: Syntax

- Syntax:

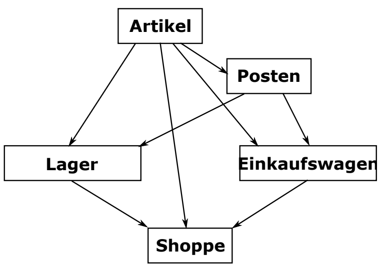
```
module Name(Bezeichner) where Rumpf
```

- Bezeichner können leer sein (dann wird alles exportiert)
- Bezeichner sind:
  - **Typen**: `T`, `T(c1,..., cn)`, `T(..)`
  - **Klassen**: `C`, `C(f1,...,fn)`, `C(..)`
  - Andere Bezeichner: **Werte**, **Felder**, **Klassenmethoden**
  - Importierte **Module**: `module M`
- Typsynonyme und Klasseninstanzen bleiben sichtbar
- Module können **rekursiv** sein *(don't try at home)*

Refakturierung im Einkaufsparadies



Refakturierung im Einkaufsparadies: Modularchitektur



Refakturierung im Einkaufsparadies I: Artikel

- Es wird **alles** exportiert
- Reine Datenmodellierung

```
module Artikel where

data Apfelsorte = Boskoop | CoxOrange | GrannySmith
apreis :: Apfelsorte → Int

data Kaesesorte = Gouda | Appenzeller
kpreis :: Kaesesorte → Double

data Menge = Stueck Int | Gramm Int | Liter Double
addiere :: Menge→ Menge→ Menge
```


Refakturierung im Einkaufsparadies II: Posten

```
module Posten(
  Posten,
  artikel,
  menge,
  posten,
  cent,
  hinzu) where
```

- Implementiert ADT Posten:

```
data Posten = Posten Artikel Menge
              deriving (Eq, Show)
```

```
artikel :: Posten → Artikel
artikel (Posten a _) = a
```

  - Konstruktor wird **nicht** exportiert
- Invariante: Posten hat immer die korrekte Menge zu Artikel

```
posten :: Artikel→ Menge→ Maybe Posten
posten a m =
  case preis a m of
    Just _  → Just (Posten a m)
    Nothing → Nothing
```


Refakturierung im Einkaufsparadies III: Lager

```
module Lager(
  Lager,
  leeresLager,
  einlagern,
  suche,
  liste,
  inventur
  ) where
```

```
import Artikel
import Posten
```

- Implementiert ADT Lager

```
data Lager
```

- Signatur der exportierten Funktionen:

```
leeresLager :: Lager

einlagern :: Artikel→ Menge→ Lager→ Lager

suche :: Artikel→ Lager→ Maybe Menge

liste :: Lager→ [(Artikel, Menge)]

inventur :: Lager→ Int
```

- **Invariante**: Lager enthält keine doppelten Artikel

Refakturierung im Einkaufsparadies IV: Einkaufswagen

```
module Einkaufswagen(
  Einkaufswagen,
  leererWagen,
  einkauf,
  kasse,
  kassenbon
  ) where
```

- ADT durch **Verkapselung**:

```
data Einkaufswagen = Ekwg [Posten]
                        deriving (Eq, Show)
```

  - Ein Typsynmonym würde exportiert
- **Invariante**: Korrekte Menge zu Artikel im Einkaufswagen

```
einkauf :: Artikel → Menge → Einkaufswagen
                          → Einkaufswagen
einkauf a m (Ekwg ps) = case posten a m of
  Just p →  Ekwg (p: ps)
  Nothing → Ekwg ps
```

  - Nutzt dazu ADT `Posten`

Refakturierung im Einkaufsparadies V: Hauptmodul

```
module Shoppe where

import Artikel
import Lager
import Einkaufswagen
```

- Nutzt andere Module

```
w0= leererWagen
w1= einkauf (Apfel Boskoop) (Stueck 3) w0
w2= einkauf Schinken (Gramm 50) w1
w3= einkauf (Milch Bio) (Liter 1) w2
w4= einkauf Schinken (Gramm 50) w3
```


Benutzung von ADTs

- **Operationen** und **Typen** müssen **importiert** werden
- Möglichkeiten des Imports:
  - **Alles** importieren
  - **Nur bestimmte** Operationen und Typen importieren
  - Bestimmte Typen und Operationen **nicht** importieren

Importe in Haskell

- Syntax:

```
import [qualified] M [as N] [hiding][(Bezeichner)]
```

- *Bezeichner* geben an, **was** importiert werden soll:
  - Ohne Bezeichner wird alles importiert
  - Mit `hiding` werden Bezeichner **nicht** importiert
- Für jeden exportierten Bezeichner `f` aus `M` wird importiert
  - `f` und qualifizierter Bezeichner `M.f`
  - `qualified`: **nur qualifizierter** Bezeichner `M.f`
  - Umbenennung bei Import mit `as` (dann `N.f`)
  - Klasseninstanzen und Typsynonyme werden immer importiert
- Alle Importe stehen immer am Anfang des Moduls

Beispiel

```
module M(a,b) where
...
```

Import(e)	Bekannte Bezeichner
`import M`	a, b, M.a, M.b
`import M()`	*(nothing)*
`import M(a)`	a, M.a
`import qualified M`	M.a, M.b
`import qualified M()`	*(nothing)*
`import qualified M(a)`	M.a
`import M hiding ()`	a, b, M.a, M.b
`import M hiding (a)`	b, M.b
`import qualified M hiding ()`	M.a, M.b
`import qualified M hiding (a)`	M.b
`import M as B`	a, b, B.a, B.b
`import M as B(a)`	a, B.a
`import qualified M as B`	B.a, B.b

Quelle: Haskell98-Report, Sect. 5.3.4

Ein typisches Beispiel

- Modul implementiert Funktion, die auch importiert wird
- Umbenennung nicht immer praktisch
- Qualifizierter Import führt zu **langen** Bezeichnern
- `Einkaufswagen` implementiert Funktionen `artikel` und `menge`, die auch aus `Posten` importiert werden:

```
import Posten hiding (artikel, menge)
import qualified Posten as P(artikel, menge)
```

```
artikel :: Posten→ String
artikel p =
    formatL 20 (show (P.artikel p)) ++
    formatR 7  (menge (P.menge p)) ++
    formatR 10 (showEuro (cent p)) ++ "\n"
```

☞ Siehe Übung 8.1

II. Schnittstelle vs. Implementation

Schnittstelle vs. Implementation

- Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben
- Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- Viel gebraucht, oft in Abwandlungen (Hashtables, Sets, Arrays)
- Abstrakter Datentyp für **endliche Abbildungen**:
  - Datentyp

```
data Map α β
```

  - Leere Abbildung:

```
empty :: Map α β
```

  - Abbildung auslesen:

```
lookup :: Ord α⇒ α→ Map α β→ Maybe β
```

  - Abbildung ändern:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
```

  - Abbildung löschen:

```
delete :: Ord α⇒ α→ Map α β→ Map α β
```


Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

```
data Map α β = Map (α→ Maybe β)
```

- Damit einfaches `lookup`, `insert`, `delete`:

```
empty = Map (λx→ Nothing)

lookup a (Map s) = s a

insert a b (Map s) = Map (λx→ if x == a then Just b else s x)

delete a (Map s) = Map (λx→ if x == a then Nothing else s x)
```

- Instanzen von Eq, Show **nicht möglich**
- **Speicherleck**: überschriebene Zellen werden nicht freigegeben

Endliche Abbildungen: Anwendungsbeispiel

- Lager als endliche Abbildung:

```
data Lager = Lager (M.Map Artikel Menge)
```

- Artikel suchen:

```
suche a (Lager l) = M.lookup a l
```

- Ins Lager hinzufügen:

```
einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager l) = case posten a m of
   Just _   → case M.lookup a l of
     Just q  → Lager (M.insert a (addiere m q) l)
     Nothing → Lager (M.insert a m l)
   Nothing → Lager l
```

- Für Inventur fehlt Möglichkeit zur **Iteration**
- Daher: Map als **Assoziativliste** ☞ Siehe Übung 8.2

Map als sortierte Assoziativliste

```
data Map α β = Map { toList :: [(α, β)] }
```

- Invariante: Liste ist in der ersten Komponente aufsteigend sortiert
- `lookup` ist vordefiniert; beim einfügen auch überschreiben;

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert a v (Map s) = Map (insert' s) where
  insert' []         = [(a, v)]
  insert' s0@((b, w):s) | a > b = (b, w): insert' s
                        | a == b = (a, v): s
                        | a < b = (a, v): s0
```

- ... ist aber **ineffizient** (Zugriff/Löschen in $\mathcal{O}(n)$)
- Deshalb: balancierte Bäume

AVL-Bäume und Balancierte Bäume

AVL-Bäume

Ein Baum ist **ausgeglichen**, wenn

- alle Unterbäume ausgeglichen sind, und
- der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

Balancierte Bäume

Ein Baum ist **balanciert**, wenn

- alle Unterbäume balanciert sind, und
- für den linken und rechten Unterbaum l, r gilt:

$$size(l) \leq w \cdot size(r) \quad (1)$$
$$size(r) \leq w \cdot size(l) \quad (2)$$

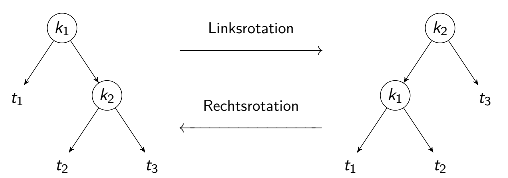
w — **Gewichtung** (Parameter des Algorithmus)

Implementation

- Balanciertheit ist **Invariante**
- Nach Einfügen oder Löschen: Balanciertheit wiederherstellen
- Dabei drei Fälle:
  1. Linker Unterbaum größer $size(l) > w \cdot size(r)$
  2. Rechter Unterbaum größer $size(r) > w \cdots size(l)$
  3. Keiner größer — Baum balanciert

Balanciertheit durch Einfache Rotation

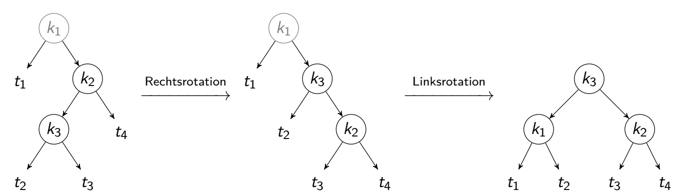
- Sei der rechte Unterbaum größer
- Zwei Unterfälle:
  1. Linkes Enkelkind t_2 größer
  2. Rechtes Enkelkind t_3 größer
- Einfache **Linksrotation** heilt (2)
- Ansonsten: **Doppelrotation** reduziert (1) zu (2)



Balanciertheit durch Doppelrotation

Falls linkes Enkelkind um Faktor α größer als rechtes:

- Nach einer einfachen Rechtsrotation des Unterbaumes ist rechtes Enkelkind größer
- Danach Linksrotation des gesamten Baumes



Implementation in Haskell

- Der Datentyp

```
data Map α β = Empty
             | Node α β Int (Map α β) (Map α β)
             deriving Eq
```

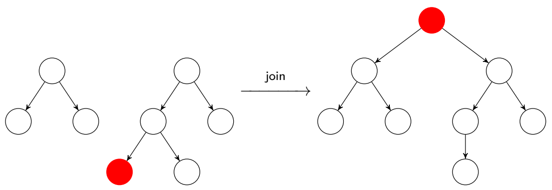
- Parameter:
  - `weight` Gewichtsfaktor w (für Einfachrotation)
  - `ratio` Gewichtsfaktor α (für Doppelrotation)
- Hilfskonstruktor `node`, setzt Größe (`l`, `r` balanciert)
- Selektor `size` für Größe des Baumes (0 für `Empty`)

Hauptfunktion

- `balance k x l r` konstruiert balancierten Baum
  - `l`, `r` sind balanciert und höchstens um einen Knoten unbalanciert
  - Vier Fälle:
    1. Beide Bäume zusammen höchstens einen Knoten $\longrightarrow$ keine Rotation
    2. $w \cdot size(l) < size(r)$: $\longrightarrow$ Linksrotation
    3. $size(l) > w \cdot size(r)$: $\longrightarrow$ Rechtsrotation
    4. Ansonsten: keine Rotation
- `balanceL k x l r` rotiert nach links. Sei r_l und r_r rechter und linker Unterbaum von `r`:
  1. $size(r_l) < \alpha \cdot size(r_r)$, dann einfache Linksrotation
  2. $size(r_l) \geq \alpha \cdot size(r_r)$ dann Doppelrotation (Rechtsrotation `r`, dann Linksrotation)

Hilfsfunktion `join` beim Löschen

- Zwei balancierte Bäume zusammenfügen (nachdem Wurzel gelöscht wurde)
- Linkester Knoten des rechten Unterbaumes wird neue Wurzel
- Mit `balance` wieder ausbalancieren



☞ Siehe Übung 8.3

Zusammenfassung Balancierte Bäume

- Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- Fold: linearer Aufwand ($\mathcal{O}(n)$)
- Guten durchschnittlicher Aufwand
- Auch in der Haskell-Bücherei: `Data.Map` (stark optimiert, mit vielen weiteren Funktionen)

Benchmarking: Setup

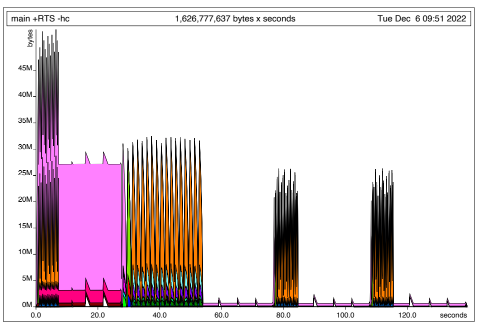
- Wie **schnell** sind die Implementationen **wirklich**?
- Benchmarking: nicht trivial
  - Verzögerte Auswertung und optimierender Compiler
  - Messen wir das **richtige**?
  - Benchmarking-Tool: Criterion
- Setup: `Map Int String` mit 50000 zufälligen Einträgen erzeugen
- Darin:
  - Einmal zufällig lesen (`lookup`), schreiben (`insert`), löschen (`delete`)
  - Sequenz aus fünfmal löschen und schreiben, zweihundertmal lesen (mixed)

Benchmarking: Resultate

	`create`	`lookup`	`insert`	`delete`	mixed
`MapFun`	53,77 ms	255,20 μs	7,74 ns	7,82 ns	131,60 μs
	6,29 ms	1,60 μs	450,40 ps	33,10 ps	3,06 μs
`MapList`	2,60 s	4,35 μs	28,76 μs	31,86 μs	2,21 ms
	17,88 ms	371,80 ns	460,10 ns	656,40 ns	77,04 μs
`MapTree`	77,93 ms	196,70 ns	32,64 μs	32,23 μs	261,50 μs
	4,22 ms	7,53 ns	788,40 ns	681,90 ns	3,39 μs
`Data.Map.Lazy`	63,14 ms	80,27 ns	30,56 μs	31,48 μs	209,10 μs
	2,13 ms	2,77 ns	293,40 ns	405,60 ns	2,02 μs

Einträge: durchschnittliche Ausführungszeit, Standardabweichung

Speicherprofil



Defizite von Haskells Modulsystem

- Signatur ist nur **implizit**
  - Exportliste enthält nur Bezeichner
  - Wünschenswert: Signatur an der Exportliste annotierbar, oder Signaturen in separater Datei
  - In Java: **Interfaces**
- Klasseninstanzen werden **immer** exportiert.
- Kein **Paket-System**

Zusammenfassung

- **Abstrakte Datentypen** (ADTs):
  - Besteht aus **Typen** und **Operationen** darauf
- Realisierung in Haskell durch **Module**
- Beispieldatentypen: endliche Abbildungen
- Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 9 (13.12.2022): Signaturen und Eigenschaften

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- **Teil II: Funktionale Programmierung im Großen**
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
- Teil III: Funktionale Programmierung im richtigen Leben

Abstrakte Datentypen und Signaturen

- Letzte Vorlesung: **Abstrakte Datentypen**
  - Typ plus Operationen
- Heute: **Signaturen** und **Eigenschaften**

Definition (Signatur)
Die **Signatur** eines abstrakten Datentyps besteht aus den Typen, und der Signatur der darüber definierten Funktionen.

- Keine direkte Repräsentation in Haskell
- Signatur: Typ eines Moduls

Lernziele
Wir wollen die Eigenschaften eines Moduls **abstrakt** spezifizieren. Wir können diese Eigenschaften nutzen, um Implementierungen zu testen.

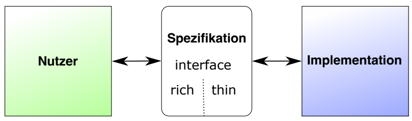
I. Eigenschaften

Signatur und Eigenschaften

- Signatur genug, um ADT **typkorrekt** zu benutzen
  - Insbesondere Anwendbarkeit und Reihenfolge
- Signatur beschreibt nicht die **Bedeutung** (Semantik):
  - Was wird gelesen?
  - Wie verhält sich die Abbildung?
- Signatur ist **Sprache** (Syntax) um **Eigenschaften** zu beschreiben

Axiome als Interface

- Axiome müssen **gelten**
  - für alle Werte der freien Variablen zu `True` auswerten
- Axiome **spezifizieren**:
  - nach außen das **Verhalten** (viele Operationen und Eigenschaften — *rich interface*)
  - nach innen die **Implementation** (wenig Operationen und Eigenschaften — *thin interface*)
- Signatur + Axiome = **Spezifikation**



Formalisierung von Eigenschaften

- Ziel: Eigenschaften **formal** beschreiben, um sie testen oder beweisen zu können.

Definition (Axiome)
Axiome sind Prädikate über den Operationen der Signatur

- Elementare Prädikate P:
  - Gleichheit `s == t`, Ordnung `s < t`
  - Selbstdefinierte Prädikate
- Zusammengesetzte Prädikate
  - Negation `not p`, Konjunktion `p && q`, Disjunktion `p || q`
  - **Implikation** `p ==> q`

Endliche Abbildung: Signatur für `Map`

- Adressen und Werte sind Parameter
- Typ `Map` α β, Operationen:

```
data Map α β

empty  :: Map α β

lookup :: Ord α⇒ α→ Map α β→ Maybe β

insert :: Ord α⇒ α→ β→ Map α β→ Map α β

delete :: Ord α⇒ α→ Map α β→ Map α β
```

Axiome für `Map`

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v s) == Just v
```

```
lookup a (delete a s) == Nothing
```

- **Lesen** an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (insert b v s) == lookup a s
```

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

```
insert a w (insert a v s) == insert a w s
```

- **Schreiben** und **Löschen** über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ insert a v (delete b s) == delete b (insert a v s)
```

- Sehr **viele** Axiome (insgesamt 13)!

☞ Siehe Übung 9.1

Thin vs. Rich Interfaces

- Benutzersicht: **reiches** Interface
  - Viele Operationen und Eigenschaften
- Implementationssicht: **schlankes** Interface
  - Wenig Operation und Eigenschaften
- Konversion dazwischen („Adapter")

Thin vs. Rich Maps

- Rich interface:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
```

```
delete :: Ord α⇒ α→ Map α β→ Map α β
```

- Thin interface:

```
put :: Ord α⇒ α→ Maybe β→ Map α β→ Map α β
```

- Konversion von thin auf rich:

```
insert a v = put a (Just v)
```

```
delete a = put a Nothing
```


Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- **Lesen** an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (put b v s) == lookup a s
```

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w s
```

- **Schreiben** über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ put a v (put b w s) == put b w (put a v s)
```

Thin: 6 Axiome
Rich: 13 Axiome

☞ Siehe Übung 9.2

II. Testen von Eigenschaften

Axiome als Eigenschaften

- Axiome können **getestet** oder **bewiesen** werden
- Tests finden Fehler, Beweis zeigt Korrektheit

> E. W. Dijkstra, 1972
> Program testing can be used to show the presence of bugs, but never to show their absence.

- Arten von Tests:
  - Unit tests (JUnit, HUnit)
  - Black Box vs. White Box
  - Coverage-based (z.B. Pfadabdeckung, MC/DC)
  - Zufallsbasiertes Testen
- Funktionale Programme eignen sich **sehr gut** zum Testen

Zufallsbasiertes Testen in Haskell

- Idee: Eigenschaften sind Konstante vom Typ `Bool`
- Für **freie** Variablen werden zufällige Werte eingesetzt:

```
put a w (put a v s) == put a w s
```

- Erweiterungen zu Bool: **Implikation** ⟹, Allquantor (Typ `Property`)
- Polymorphe Variablen nicht testbar
  - Deshalb Typvariablen **instantiieren**
  - Typ muss genug Element haben (hier `Map Int String`)
  - Durch Signatur Typinstanz erzwingen
- Werkzeug: *QuickCheck*

Axiome mit *QuickCheck* testen

- Eigenschaften als **monomorphe Haskell-Prädikate**
- Für das Lesen:

```
prop1 = QC.testProperty "read_empty" $ λa→
  lookup a (empty :: Map Int String) == Nothing
```

```
prop3 = QC.testProperty "lookup_put␣eq" $ λa v (s :: Map Int String)→
  lookup a (put a v s) == v
```

- *QuickCheck*-Axiome mit `QC.testProperty` in *Tasty* eingebettet
- Es werden N Zufallswerte generiert und getestet (Default $N = 100$)

Axiome mit *QuickCheck* testen

- **Bedingte** Eigenschaften:
  - `A` $\Longrightarrow$ `B` mit `A`, `B` Eigenschaften
  - Es werden solange Zufallswerte generiert, bis *N* die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.
  - Warum?
  - Implikation *false* $\Longrightarrow \phi$ ist immer wahr (und sagt **nichts** über ϕ).

```
prop4 = QC.testProperty "lookup␣other" $ λa b v (s :: Map Int String)→
  a ≠ b ⟹ lookup a (put b v s) == lookup a s
```


Axiome mit *QuickCheck* testen

- **Schreiben**:

```
prop5 = QC.testProperty "put_put␣eq" $ λa v w (s :: Map Int String)→
  put a w (put a v s) == put a w s
```

- **Schreiben** an anderer Stelle:

```
prop6 = QC.testProperty "put_put␣other" $ λa v b w (s :: Map Int String)→
  a ≠ b ⟹ put a v (put b w s) == put b w (put a v s)
```

- Test benötigt **Gleichheit** und **Zufallswerte** für `Map a b`

Beobachtbare und Abstrakte Typen

- **Beobachtbare** Typen: interne Struktur bekannt
  - Vordefinierte Typen (Zahlen, Zeichen), algebraische Datentypen (Listen)
  - Viele Eigenschaften und Prädikate bekannt
- **Abstrakte** Typen: interne Struktur unbekannt
  - Wenige Eigenschaften bekannt, Gleichheit nur wenn definiert
- Beispiel `Map`:
  - beobachtbar: Adressen und Werte
  - abstrakt: Speicher

Beobachtbare Gleichheit

- Auf abstrakten Typen: nur **beobachtbare** Gleichheit
  - Zwei Elemente sind gleich, wenn alle Operationen die gleichen Werte liefern
- Bei **Implementation**: Instanz für `Eq` (`Ord` etc.) entsprechend definieren
  - Die Gleichheit `==` muss die **beobachtbare** Gleichheit sein.
- Abgeleitete Gleichheit (`deriving Eq`) wird **immer** exportiert!

Zufallswerte selbst erzeugen

- Problem: **Zufällige** Werte von **selbstdefinierten** Datentypen
  - Gleichverteiltheit auf die Konstruktoren nicht immer erwünscht (z.B. `[α]`)
  - Konstruktion nicht immer offensichtlich (z.B. `Map`)
- In *QuickCheck*:
  - **Typklasse** `class Arbitrary α` für Zufallswerte
  - Eigene **Instanziierung** kann Verteilung und Konstruktion berücksichtigen

```
instance (Ord a, QC.Arbitrary a, QC.Arbitrary b)⇒
        QC.Arbitrary (Map a b) where
```

  - Zufallswerte in Haskell?

Zufällige Maps erzeugen

- Erster Ansatz: zufällige Länge, dann aus sovielen zufälligen Werten `Map` konstruieren
  - Berücksichtigt `delete` nicht
- Besser: über einen **smart constructor** zufällige Maps erzeugen
  - Muss entweder in `Map` implementiert werden
  - oder benötigt Zugriff auf interne Struktur

☞ Siehe Übung 9.3

III. Syntax und Semantik

Signatur und Semantik

Stacks

Typ: `St α`

Initialwert:

```
empty :: St α
```

Wert ein/auslesen:

```
push :: α→ St α→ St α
top  :: St α→ α
pop  :: St α→ St α
```

Last in, first out (LIFO).

Queues

Typ: `Qu α`

Initialwert:

```
empty :: Qu α
```

Wert ein/auslesen:

```
enq   :: α→ Qu α→ Qu α
first :: Qu α→ α
deq   :: Qu α→ Qu α
```

First in, first out (FIFO).

Gleiche Signatur, unterscheidliche Semantik.

Eigenschaften von `Stack`

- Last in first out (LIFO):

$$\texttt{top}\ (\texttt{push}\ a_1\ (\texttt{push}\ a_2\ \dots\ (\texttt{push}\ a_n\ \texttt{empty}))) = a_1$$

```
top (push a s) == a
```

```
pop (push a s) == s
```

```
push a s ≠ empty
```


Eigenschaften von `Queue`

- First in, first out (FIFO):

$$\texttt{first}\ (\texttt{enq}\ a_1\ (\texttt{enq}\ a_2\ \dots\ (\texttt{enq}\ a_n\ \texttt{empty}))) = a_n$$

```
first (enq a empty) == a
```

```
q ≠ empty ⟹ first (enq a q) == first q
```

```
deq (enq a empty) == empty
```

```
q ≠ empty ⟹ deq (enq a q) == enq a (deq q)
```

```
enq a q ≠ empty
```


Implementation von `Stack`: Liste

Sehr einfach: ein Stack ist eine Liste

```
data St α = St [α]  deriving (Show, Eq)
```

```
empty = St []
```

```
push a (St s) = St (a:s)
```

```
top  (St []) = error "St:␣top␣on␣empty␣stack"
top  (St s)  = head s
```

```
pop  (St []) = error "St:␣pop␣on␣empty␣stack"
pop  (St s)  = St (tail s)
```


Implementation von `Queue`

- Mit einer Liste?
  - Problem: am Ende anfügen oder abnehmen (`last`/`init`) ist teuer ($O(n)$).
- Deshalb **zwei** Listen:
  - Erste Liste: zu entnehmende Elemente
  - Zweite Liste: hinzugefügte Elemente **rückwärts**
  - Invariante: erste Liste leer gdw. Queue leer
- Beispiel für guten **amortisierten** Aufwand.

Repräsentation von `Queue`

Operation	Resultat	Interne Repräsentation	`first`
`empty`	$\langle\,\rangle$	([], [])	`error`
`enq 9`	$\langle 9\rangle$	([9], [])	9
`enq 4`	$\langle 4 \to 9\rangle$	([9], [4])	9
`enq 7`	$\langle 7 \to 4 \to 9\rangle$	([9], [7, 4])	9
`deq`	$\langle 7 \to 4\rangle$	([4, 7], [])	4
`enq 5`	$\langle 5 \to 7 \to 4\rangle$	([4, 7], [5])	4
`enq 3`	$\langle 3 \to 5 \to 7 \to 4\rangle$	([4, 7], [3, 5])	4
`deq`	$\langle 3 \to 5 \to 7\rangle$	([7], [3, 5])	7
`deq`	$\langle 3 \to 5\rangle$	([5, 3], [])	5
`deq`	$\langle 3\rangle$	([3], [])	3
`deq`	$\langle\,\rangle$	([], [])	`error`
`deq`	`error`		

Implementation: Datentyp

- Datentyp:

```
data Qu α = Qu [α] [α]
```

- Invariante:
  1. Anfang der Schlange ist der **Kopf** der ersten Liste
  2. Wenn erste Liste leer, dann ist auch die zweite Liste leer
- Invariante prüfen und ggf. herstellen (**smart constructor**):

```
queue :: [α] → [α] → Qu α
queue [] ys = Qu (reverse ys) []
queue xs ys = Qu xs ys
```


Implementation: Operationen

- Leere Schlange: alles leer

```
empty :: Qu α
empty = Qu [] []
```

- Erstes Element steht vorne in erster Liste

```
first :: Qu α → α
first (Qu [] _)     = error "Queue:␣first␣of␣empty␣Q"
first (Qu (x:xs) _) = x
```

- Bei `enq` und `deq` Invariante prüfen (Funktion `queue`)

```
enq   :: α → Qu α → Qu α
enq x (Qu xs ys) = queue xs (x:ys)
```

```
deq   :: Qu α → Qu α
deq (Qu [] _ )     = error "Queue:␣deq␣of␣empty␣Q"
deq (Qu (_:xs) ys) = queue xs ys
```

☞ Siehe Übung 9.4

Zusammenfassung

- **Signatur**: Typ und Operationen eines ADT
- **Axiome**: über Typen formulierte Eigenschaften
- **Spezifikation** = Signatur + Axiome
  - Interface zwischen Implementierung und Nutzung
  - Testen zur Erhöhung der Konfidenz und zum Fehlerfinden
  - Beweisen der Korrektheit
- `QuickCheck`:
  - Freie Variablen der Eigenschaften werden Parameter der Testfunktion
  - $\Longrightarrow$ für bedingte Eigenschaften

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 10 (20.12.2022): Aktionen und Zustände

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

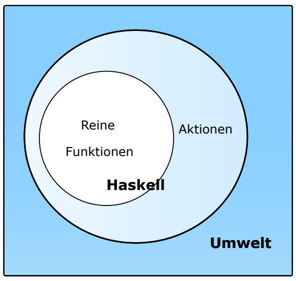
Inhalt

- Ein/Ausgabe in funktionale Sprachen
- Wo ist das **Problem**?
- **Aktionen** und der Datentyp *IO*.
- Vordefinierte Aktionen
- Beispiel: Wortratespiel
- Aktionen als Werte

Lernziele
Wir verstehen, wie wir Ein- und Ausgabe in Haskell funktional modellieren.

I. Funktionale Ein/Ausgabe

Ein- und Ausgabe in funktionalen Sprachen



Problem:
- Funktionen mit Seiteneffekten nicht referentiell transparent.
- `readString :: ... → String` ??

Lösung:
- Seiteneffekte am Typ erkennbar
- **Aktionen**
  - Können **nur** mit **Aktionen** komponiert werden
  - „einmal Aktion, immer Aktion"

Aktionen als abstrakter Datentyp

- ADT mit Operationen **Komposition** und **Lifting**
- Signatur:

```
type IO α

(≫=)    :: IO α → (α→ IO β) → IO β — Komposition

return  :: α→ IO α                 — Lifting
```

- Dazu **elementare** Aktionen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von Standardeingabe (`stdin`) **lesen**:

```
getLine  :: IO String
```

- Zeichenkette auf Standardausgabe (`stdout`) **ausgeben**:

```
putStr   :: String→ IO ()
```

- Zeichenkette mit Zeilenvorschub **ausgeben**:

```
putStrLn :: String→ IO ()
```


Einfache Beispiele

- Echo einfach

```
echo1 :: IO ()
echo1 = getLine ≫= putStrLn
```

- Echo mehrfach

```
echo :: IO ()
echo = getLine ≫= putStrLn ≫= λ_ → echo
```

- Was passiert hier?
  - Verknüpfen von Aktionen mit ≫=
  - Jede Aktion gibt **Wert** zurück

Noch ein Beispiel

- Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine >>= λs→ putStrLn (reverse s) >> ohce
```

- Was passiert hier?
  - **Reine** Funktion `reverse` wird innerhalb von **Aktion** `putStrLn` genutzt
  - Folgeaktion `ohce` benötigt **Wert** der vorherigen Aktion nicht
  - **Abkürzung**: >>

```
p >> q = p >>= λ_ → q
```


Die do-Notation

- Syntaktischer Zucker für `IO`:

```
echo =                          echo =
  getLine                  ⟺      do s← getLine
   >>= λs→ putStrLn s                 putStrLn s
   >> echo                            echo
```

- Rechts sind >>=, >> implizit
- Mit ← gebundene Bezeichner **überlagern** vorherige
- Es gilt die **Abseitsregel**.
  - Einrückung der ersten Anweisung nach `do` bestimmt Abseits.

Drittes Beispiel

- Zählendes, endliches Echo

```
echo3 :: Int→ IO ()
echo3 cnt = do
  putStr (show cnt ++ "?␣")
  s← getLine
  if s ≠ "" then do
    putStrLn $ show cnt ++ ":␣"++ s
    echo3 (cnt+ 1)
  else return ()
```

- Was passiert hier?
  - Kombination aus Kontrollstrukturen und Aktionen
  - **Aktionen** als **Werte**
  - Geschachtelte `do`-Notation

☞ Siehe Übung 10.1

II. Aktionen als Werte

Aktionen als Werte

- **Aktionen** sind **Werte** wie alle anderen.
- Dadurch **Definition** von **Kontrollstrukturen** möglich.
- Endlosschleife:

```
forever :: IO α→ IO α
forever a = a >> forever a
```

- Iteration (feste Anzahl):

```
forN :: Int→ IO α→ IO ()
forN n a | n == 0    = return ()
         | otherwise = a >> forN (n-1) a
```


Kontrollstrukturen

- Vordefinierte Kontrollstrukturen (`Control.Monad`):

```
when :: Bool→ IO ()→ IO ()
```

- Sequenzierung:

```
sequence :: [IO α]→ IO [α]
```

  - Sonderfall: `[()]` als `()`

```
sequence_ :: [IO ()]→ IO ()
```

- Map und Filter für Aktionen:

```
mapM    :: (α→ IO β)→ [α]→ IO [β]
mapM_   :: (α→ IO ())→ [α]→ IO ()
filterM :: (α→ IO Bool) → [α] → IO [α]
```

☞ Siehe Übung 10.2

III. Ein/Ausgabe

Ein/Ausgabe mit Dateien

- Im Prelude **vordefiniert**:
  - Dateien schreiben (überschreiben, anhängen):

```
type FilePath = String
writeFile   :: FilePath → String → IO ()
appendFile  :: FilePath → String → IO ()
```

- Datei lesen (verzögert):

```
readFile    :: FilePath → IO String
```

- "Lazy I/O": Zugriff auf Dateien erfolgt **verzögert**
  - Interaktion von nicht-strikter Auswertung mit zustandsbasiertem Dateisystem kann überraschend sein

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String→ IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file++ ":␣"++
                 show (length (lines cont))++ "␣lines,␣" ++
                 show (length (words cont))++ "␣words,␣"++
                 show (length cont)++ "␣bytes."
```

- Datei wird gelesen
- Anzahl Zeichen, Worte, Zeilen gezählt
- Erstaunlich (hinreichend) effizient

Ein/Ausgabe mit Dateien: Abstraktionsebenen

- **Einfach**: `readFile`, `writeFile`
  - Im `Prelude` **vordefiniert**, **portabel**.
- **Fortgeschritten**: Modul `System.IO` der Standardbücherei
  - Buffered/Unbuffered, Seeking, Operationen auf `Handle`
  - **Portabel** — funktioniert auf allen Platformen
- **Systemnah:** Modul `System.Posix`
  - Filedeskriptoren, Permissions, special devices, etc.
  - **Systemspezifisch** (nicht vollständig portabel).

IV. Ausnahmen und Fehlerbehandlung

Fehlerbehandlung, erster Versuch

- Wie könnten wir **Fehler** modellieren?
  - Fehler werden durch Typ `E` repräsentiert.
  - Berechnung mit Fehler: `Either E` α
  - Fehler **fangen**: `catch :: Either E` $\alpha \to$ `(E` $\to \alpha) \to \alpha$
  - Fehler erzeugen: `Left e`
- Probleme:
  - Ausnahmen sollen **erweiterbar** bleiben.
  - Man muss **entweder alle** Berechnungen mit `Right x` in den Fehlertypen liften,
  - **oder** das Fangen ist nicht referentiell transparent.

Fehlerbehandlung

- **Fehler** werden durch `Exception` repräsentiert (Modul `Control.Exception`)
  - `Exception` ist **Typklasse** — kann durch eigene Instanzen erweitert werden
  - Vordefinierte Instanzen: u.a. `IOError`
- Fehlerbehandlung durch **Ausnahmen** (ähnlich Java)

```
throw :: Exception γ⇒ γ→ α
catch :: Exception γ⇒ IO α → (γ→ IO α) → IO α
try   :: Exception γ⇒ IO α → IO (Either γ α)
```

- Faustregel: `catch` für unerwartete Ausnahmen, `try` für erwartete
- Ausnahmen überall, Fehlerbehandlung **nur in Aktionen**

Fehler fangen und behandeln

> "Ask forgiveness not permission" (Grace Hopper)
>
> Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als vorherige Abfrage von Fehlerbedingungen.

- Warum? Umwelt nicht **sequentiell**.
- Fehlerbehandlung für `wc`:

```
wc2 :: String→ IO ()
wc2 file =
  catch (wc file)
        (λe → putStrLn $ "Fehler:␣"++ show (e :: IOError))
```

- `IOError` kann analysiert werden (siehe `System.IO.Error`)
- `read` mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read α⇒ String→ IO α
```


Ausführbare Programme

- Eigenständiges Programm ist **Aktion**
- **Hauptaktion**: `main :: IO ()` in Modul `Main`
  - ... oder mit der Option `-main-is M.f` setzen
- `wc` als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Control.Exception
```

```
main :: IO ()
main = do
  args ← getArgs
  putStrLn $ "Command␣line␣arguments:␣" ++ show args
  mapM_ wc2 args
```


Beispiel: Traversion eines Verzeichnisbaums

- Verzeichnisbaum traversieren, und für jede Datei eine **Aktion** ausführen:

```
travFS :: (FilePath→ IO ())→ FilePath→ IO ()
travFS action p = catch (do
    cs← getDirectoryContents p
    let cp = map (p </>) (cs \\ [".", ".."])
    dirs  ← filterM doesDirectoryExist cp
    files ← filterM doesFileExist cp
    mapM_ action files
    mapM_ (travFS action) dirs)
  (λe → putStrLn $ "ERROR:␣"++ show (e :: IOError))
```

- Nutzt Funktionalität aus `System.Directory`, `System.FilePath`

☞ Siehe Übung 10.3

V. Anwendungsbeispiel

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α)→ IO α
```

  - Warum ist `randomIO` **Aktion**?
- **Beispiele:**
  - Aktion zufällig oft ausführen:

```
atmost :: Int→ IO α→ IO [α]
atmost most a =
  do l← randomRIO (1, most)
     sequence (replicate l a)
```

  - Zufälligen String erzeugen:

```
randomStr :: IO String
randomStr = atmost 40 (randomRIO ('a','z'))
```

- Hinweis: Funktionen aus `System.Random` zu importieren, muss ggf. installiert werden.

Fallbeispiel: Wörter raten

- Unterhaltungsprogramm: der Benutzer rät Wörter
- Benutzer kann einzelne Buchstaben eingeben
- Wort wird maskiert ausgegeben, nur geratene Buchstaben angezeigt

Wörter raten: Programmstruktur

- Hauptschleife:

```
play :: String→ String→ String→ IO ()
```

  - Argumente: Geheimnis, geratene Buchstaben (enthalten, nicht enthalten)
- Benutzereingabe:

```
getGuess :: String→ String→ IO Char
```

  - Argumente: geratene Zeichen (im Geheimnis enthalten, nicht enthalten)
- Hauptfunktion:

```
main :: IO ()
```

  - Liest ein Lexikon, wählt Geheimnis aus, ruft Hauptschleife auf

☞ Siehe Übung 10.4

Zusammenfassung

- Ein/Ausgabe in Haskell durch **Aktionen**
- **Aktionen** (Typ `IO α`) sind seiteneffektbehaftete Funktionen
- Komposition von Aktionen durch

```
(>>=)  :: IO α→ (α→ IO β)→ IO β
return :: α→ IO α
```

- `do`-Notation
- Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`, `try`).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
  - Module: `System.IO`, `System.Random`
- Nächste Vorlesung: Wie sind Aktionen eigentlich **implementiert**? Schwarze Magie?

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 11 (10.01.2023): Monaden als Berechnungsmuster

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Frohes neues Jahr!

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

Inhalt

- Wie geht das mit `IO`?
- Monaden als allgemeines Berechnungsmuster
- Fallbeispiel: Auswertung von Ausdrücken

Lernziele

Wir verstehen, wie wir Berechnungsmuster wie Seiteneffekte, Partialität oder Mehrdeutigkeit funktional modellieren.

I. Zustandsabhängige Berechnungen

Funktionen mit Zustand

- Idee: Seiteneffekt **explizit** machen
- Funktion $f : \alpha \to \beta$ mit Seiteneffekt in **Zustand** σ:

$$f : \alpha \times \sigma \to \beta \times \sigma$$
$$\cong$$
$$f : \alpha \to \sigma \to \beta \times \sigma$$

- Datentyp für Zustand σ: $\sigma \to \beta \times \sigma$
- Komposition: Funktionskomposition und `uncurry`

```
curry   :: ((α, β) → γ)→ α→ β→ γ
uncurry :: (α→ β→ γ)→ (α, β) → γ
```


In Haskell: Zustände explizit

- **Zustandstransformer:** Berechnung mit Seiteneffekt in Typ σ (polymorph über α)

```
type State σ α = σ→ (α, σ)
```

- Komposition zweier solcher Berechnungen:

```
comp :: State σ α→ (α→ State σ β)→ State σ β
comp f g = uncurry g ∘ f
```

- Trivialer Zustand:

```
lift :: α→ State σ α
lift = curry id
```

- Lifting von Funktionen:

```
map :: (α→ β)→ State σ α→ State σ β
map f g = (λ(a, s)→ (f a, s)) ∘ g
```


Zugriff auf den Zustand

- Zustand lesen:

```
get :: (σ→ α)→ State σ α
get f s = (f s, s)
```

- Zustand setzen:

```
set :: (σ→ σ)→ State σ ()
set g s = ((), g s)
```

Einfaches Beispiel

- Zähler als Zustand:

```
type WithCounter α = State Int α
```

- Beispiel: Funktion, die in Kleinbuchstaben konvertiert und **zählt**

```
cntToL :: String→ WithCounter String
cntToL [] = lift ""
cntToL (x:xs)
  | isUpper x =  cntToL xs ‘comp‘
                 λys→ set (+1) ‘comp‘
                 λ()→ lift (toLower x: ys)
  | otherwise = cntToL xs ‘comp‘ λys→ lift (x: ys)
```

- Hauptfunktion (verkapselt `State`):

```
cntToLower :: String→ (String, Int)
cntToLower s = cntToL s 0
```

☞ Siehe Übung 11.1

II. Monaden

Monaden als Berechnungsmuster

- In `cntToL` werden zustandsabhängige Berechnungen verkettet.
- So ähnlich wie bei Aktionen!

State:	Aktionen:
`type State σ α`	`type IO α`
`comp :: State σ α → (α→ State σ β)→ State σ β`	`(≫=) :: IO α → (α→ IO β) → IO β`
`lift :: α→ State σ α`	`return :: α→ IO α`
`map :: (α→ β)→ State σ α→ State σ β`	`fmap :: (α→ β)→ IO α→ IO β`

Berechnungsmuster — **Monade**

Was ist ein Berechnungsmuster?

- Ein **Berechnungsmuster** hat eine **Einheit** und kann **verknüpft** werden.
- Beispiele:
  - **Seiteneffekte** (Zustand),
  - **Fehler** (Partialität),
  - **Mehrdeutigkeit**,
  - **Aktionen**.
- Eine Monade ist ein **Typkonstruktor**, der zu einem Typ **Berechnungsmuster hinzufügt**.
- **Mathematisch** ist eine Monade eine **verallgemeinerte algebraische Theorie** (durch Operationen und Gleichungen definiert).

Monaden in Haskell

- Monaden sind erstmal Funktoren:

```
class Functor f where
  fmap :: (α→ β) → f α → f β
```

- Es sollte gelten (kann nicht geprüft werden):

```
fmap id = id
fmap f ∘ fmap g = fmap (f ∘ g)
```

  - Standard: "*Instances of Functor should satisfy the following laws.*"

Monaden in Haskell

- Verkettung (≫=) und Lifting (`return`):

```
class (Functor m, Applicative m)⇒ Monad m where
  (≫=)   :: m α → (α → m β) → m β
  return :: α → m α
```

≫= ist assoziativ und `return` das neutrale Element:

```
     return a ≫= k   ==  k a
        m ≫= return  ==  m
m ≫= (x → k x ≫= h)  ==  (m ≫= k) ≫= h
```

- Auch diese Eigenschaften können nicht geprüft werden.
- Den syntaktischen Zucker (`do`-Notation) gibt's umsonst dazu.

Beispiele für Monaden

- Zustandsmonaden: `ST`, `State`, `Reader`, `Writer`
- Fehler und Ausnahmen: `Maybe`, `Either`
- Mehrdeutige Berechnungen: `List`, `Set`

Die Reader-Monade

- Aus dem Zustand wird nur gelesen:

```
data Reader σ α = R {run :: σ→ α}
```

- Instanzen:

```
instance Functor (Reader σ) where
  fmap f (R g) = R (f. g)
```

```
instance Monad (Reader σ) where
  return a = R (const a)
  R f ≫= g = R $ λs→ run (g (f s)) s
```

- Nur eine elementare Operation:

```
get :: (σ→ α)→ Reader σ α
get f = R $ λs→ f s
```

Fehler und Ausnahmen

- `Maybe` und `Either` als Monade:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing  = Nothing
```

```
instance Functor (Either ε) where
  fmap f (Right b) = Right (f b)
  fmap f (Left  a) = Left  a
```

```
instance Monad Maybe where
  Just a >>= g  = g a
  Nothing >>= g = Nothing
  return = Just
```

```
instance Monad (Either ε) where
  Right b >>= g = g b
  Left a  >>= _ = Left a
  return = Right
```

- Berechnungsmodell: **Ausnahmen** (Fehler)
  - `f :: α → Maybe β` ist Berechnung mit möglichem (unspezifiertem) Fehler,
  - `f :: α → Either ε α` ist Berechnung mit möglichem Fehler vom Typ ϵ
  - Fehlerfreie Berechnungen werden verkettet
  - Fehler (`Nothing` oder `Left x`) werden propagiert

Mehrdeutigkeit

- `List` als Monade:
  - Können wir so nicht hinschreiben, Syntax vordefiniert

```
instance Functor [α] where
  fmap = map
```

```
instance Monad [α] where
  a : as >>= g = g a ++ (as >>= g)
  [] >>= g = []
  return a = [a]
```

- Berechnungsmodell: Mehrdeutigkeit
  - `f :: α → [β]` ist Berechnung mit **mehreren** möglichen Ergebnissen
  - Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis

Beispiel

- Berechnung aller Permutationen einer Liste:

1. Ein Element überall in eine Liste einfügen:

```
ins :: α → [α] → [[α]]
ins x [] = return [x]
ins x (y:ys) = [x:y:ys] ++ do
  is ← ins x ys
  return $ y:is
```

2. Damit Permutationen (rekursiv):

```
perms :: [α] → [[α]]
perms [] = return []
perms (x:xs) = do
  ps ← perms xs
  is ← ins x ps
  return is
```

☞ Siehe Übung 11.2

Die Listenmonade in der Listenkomprehension

- Berechnung aller Permutationen einer Liste:

1. Ein Element überall in eine Liste einfügen:

```
ins' :: α → [α] → [[α]]
ins' x [] = [[x]]
ins' x (y:ys) = [x:y:ys] ++ [ y:is | is← ins' x ys ]
```

2. Damit Permutationen (rekursiv):

```
perms' :: [α] → [[α]]
perms' [] = [[]]
perms' (x:xs) = [ is | ps ← perms' xs, is ← ins' x ps ]
```

- Listenkomprehension ≅ Listenmonade

III. IO ist keine Magie

Implizite vs. explizite Zustände

- Wie funktioniert jetzt `IO`?
- Nachteil von `State`: Zustand ist **explizit**
  - Kann dupliziert werden
- Daher: Zustand **implizit** machen
  - Datentyp verkapseln: kein `run`, kein parametrisierter Zustand
  - Zugriff auf `State` nur über elementare Operationen: kein `get` oder `set`

Aktionen als Zustandstransformationen

- **Idee**: Aktionen sind Transformationen auf Systemzustand S
- S beinhaltet
  - Speicher als Abbildung $A \rightharpoonup V$ (Adressen A, Werte V)
  - Zustand des Dateisystems
  - Zustand des Zufallsgenerators
- In Haskell: Typ `RealWorld`
  - "Virtueller" Typ, Zugriff nur über elementare Operationen
  - Entscheidend nur Reihenfolge der Aktionen

☞ Siehe Übung 11.3

IV. Fallbeispiel: Auswertung von Ausdrücken

Monaden im Einsatz

- Auswertung von Ausdrücken:

Algebraische Ausdrücke:

```haskell
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

Auswertung ohne Effekte:

```haskell
eval  :: Expr → Double
eval (Var _)  = 0
eval (Num n)  = n
eval (Plus a b)  = eval a+ eval b
eval (Minus a b) = eval a- eval b
eval (Times a b) = eval a∗ eval b
eval (Div a b)   = eval a/ eval b
```

- Mögliche Arten von Effekten:
  - Partialität (Division durch 0)
  - Zustände (für die Variablen)
  - Mehrdeutigkeit

Auswertung mit Fehlern

- Partialität durch Fehlermonade (`Either`):

```haskell
eval  :: Expr → Either String Double
eval (Var x)  = Left $ "No␣variable␣"++ x
eval (Num n)  = return n
eval (Plus a b)  = do x← eval a; y← eval b; return $ x+ y
eval (Minus a b) = do x← eval a; y← eval b; return $ x- y
eval (Times a b) = do x← eval a; y← eval b; return $ x∗ y
eval (Div a b)   = do
  x← eval a; y← eval b;
  if y == 0 then Left "Division␣by␣zero" else Right $ x/ y
```


Auswertung mit Zustand

- Zustand durch `Reader`-Monade

```haskell
import ReaderMonad
import qualified Data.Map as M

type State = M.Map String Double

eval  :: Expr → Reader State Double
eval (Var i)  = get (M.! i)
eval (Num n)  = return n
eval (Plus a b)  = do x← eval a; y← eval b; return $ x+ y
eval (Minus a b) = do x← eval a; y← eval b; return $ x- y
eval (Times a b) = do x← eval a; y← eval b; return $ x∗ y
eval (Div a b)   = do x← eval a; y← eval b; return $ x/ y
```


Mehrdeutige Auswertung

- Dazu: Erweiterung von `Expr`:

```haskell
data Expr = Var String
          |...
          | Pick Expr Expr
```

```haskell
eval  :: Expr → [Double]
eval (Var i)  = return 0
eval (Num n)  = return n
eval (Plus a b)  = do x← eval a; y← eval b; return $ x+ y
eval (Minus a b) = do x← eval a; y← eval b; return $ x- y
eval (Times a b) = do x← eval a; y← eval b; return $ x∗ y
eval (Div a b)   = do x← eval a; y← eval b; return $ x/ y
eval (Pick a b)  = do x← eval a; y← eval b; [x, y]
```


Kombination der Effekte

- Benötigt **Kombination** der Monaden.
- Monade `Res`:
  - Zustandsabhängig
  - Mehrdeutig
  - Fehlerbehaftet

```haskell
type Exn α = Either String α
data Res σ α = Res { run  :: σ→ [Exn α] }
```

- Berechnungen sind von einem Zustand abhängig, der mehrere Ergebnisse geben kann, von denen einige Fehler sein können.
- Andere Kombinationen möglich.

☞ Siehe Übung 11.4

`Res`: Monadeninstanz

- `Res` α ist `Reader (List (Exn α))`
- `Functor` durch Komposition der `fmap`:

```haskell
instance Functor (Res σ) where
  fmap f (Res g) = Res $ fmap (fmap f). g
```

- `Monad` durch Kombination der jeweiligen Operationen `return` und `≫=`:

```haskell
instance Monad (Res σ) where
  return a = Res (const [Right a])
  Res f ≫= g = Res $ λs→ do ma← f s
                            case ma of
                              Right a→ run (g a) s
                              Left e → return (Left e)
```


`Res`: Operationen

- Zugriff auf den Zustand:

```haskell
get  :: (σ→ Exn α)→ Res σ α
get f = Res $ λs→ [f s]
```

- Fehler:

```haskell
fail  :: String→ Res σ α
fail msg = Res $ const [Left msg]
```

- Mehrdeutige Ergebnisse:

```haskell
join  :: α→ α→ Res σ α
join a b = Res $ λs→ [Right a, Right b]
```


Auswertung mit Allem

- Im Monaden `Res` können alle Effekte benutzt werden:

```haskell
type State = M.Map String Double

eval  :: Expr → Res State Double
eval (Var i)  = get (λs→ case M.lookup i s of
                            Just x→  return x
                            Nothing→ Left $ "No␣such␣variable␣"++ i)
eval (Num n)  = return n
eval (Plus a b)  = do x← eval a; y← eval b; return $ x+ y
eval (Minus a b) = do x← eval a; y← eval b; return $ x- y
eval (Times a b) = do x← eval a; y← eval b; return $ x∗ y
eval (Div a b)   = do x← eval a; y← eval b
                      if y == 0 then fail "Divison␣by␣zero." else return $ x / y
eval (Pick a b)  = do x← eval a; y← eval b; join x y
```

- Systematische Kombination durch **Monadentransformer**

Zusammenfassung

- Monaden sind **Muster** für **Berechnungen** mit **Seiteneffekten**
- Beispiele:
  - Zustandstransformer (`State`)
  - Fehler und Ausnahmen (`Maybe`, `Either`)
  - Nichtdeterminismus (`List`)
- Fallbeispiel Auswertung von Ausdrücken:
  - Kombination aus Zustand, Partialität, Mehrdeutigkeit
- Grenze: Nebenläufigkeit

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 12 (17.01.2023): Funktionale Webanwendungen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

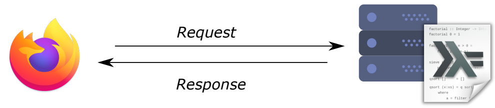
Wintersemester 2022/23

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

I. Eine kurze Einführung in die Webentwicklung

Wie funktioniert das Web?



Kennzeichen einer Webanwendung

- **Zustandsfreiheit**: jeder Request ist ein neuer
- **Nebenläufigkeit**: ein Server, viele Browser (gleichzeitig)
- **Entkoppelung**: Serveranwendung und Browser weit entfernt

Grober Ablauf

$$\underbrace{\texttt{https}}_{\textit{Protokol}}\texttt{://}\underbrace{\texttt{www.informatik.uni-bremen.de}}_{\textit{Server}}\underbrace{\texttt{/home/cxl/}}_{\textit{Pfad}}$$

1. Browser stellt **Anfrage**
   - *Gib mir Seite /home/cxl/*
2. Server nimmt Anfrage entgegen, **löst** Anfrage auf
   - */home/cxl/, das muss die Datei /var/www/cxl/index.html sein.*
3. Server sendet Antwort
   - *Hier ist die Seite:* `<h1>Hallo</h1><p>Foo ba...`

Verfeinerter Ablauf

- Das **Protokoll** ist HTTP (RFC 2068, 7540).
  - HTTP kennt vier Arten von Requests: GET, POST, PUT, DELETE.
- Der Server löst den **Pfad** `/bar/bar/` zu einer **Resource** auf (**Routing**). Das kann eine Datei sein (static routing), oder es wird eine Funktion aufgerufen, die ein Ergebnis erzeugt.
- HTTP kennt verschiedene Arten von **response codes** (100, 404, . . .). Der Inhalt der Antwort ist **beliebig**, und nicht notwendigerweise HTML.

Architekturerwägungen

- Webanwendungen müssen **zustandsfrei** sein und **skalieren**
- Übertragung ist **unzuverlässig**.
- Architekturstil: **REST** (Representational State Transfer)
  - Sammlung von **Architekturprinzipien**
- Dazu: CRUD (create, read, update, delete)

Merkmale von REST-Architekturen

1. Zustandslosigkeit — jede Nachricht in sich vollständig
2. Caching
3. Einheitliche Schnittstelle:
   - Adressierbare Ressourcen — als URL
   - Repräsentation zur Veränderungen von Ressourcen
   - Selbstbeschreibende Nachrichten
   - *Hypermedia as the engine of the application state* (HATEOAS)
4. Architektur: Client-Server, mehrschichtig

Anatomie einer Web-Applikation

- **Routing:** Auflösen der Pfade zu Aktionen
- Eigentliche Aktion
- **Persistentes** Backend
- Erzeugung von HTML (meistens), JSON (manchmal)

☞ Siehe Übung 12.1

II. Web Development in Haskell

Scotty: ein einfaches Web-Framework

From the web-page https://hackage.haskell.org/package/scotty:

Scotty is the cheap and cheerful way to write RESTful, declarative web applications.

- A page is as simple as defining the verb, url pattern, and Text content.
- It is template-language agnostic. Anything that returns a Text value will do.
- Conforms to WAI Application interface.
- Uses very fast Warp webserver by default.

Ein erster Eindruck

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty

import Data.Monoid (mconcat)

main = scotty 3000 $
  get "/:word" $ do
    beam ← param "word"
    html $ mconcat ["<h1>Scotty,␣", beam, "␣me␣up!</h1>"]
```

(Auch von der Webseite.)

Ein erstes Problem

- Repräsentation von Zeichenketten als `type String=[Char]` ist elegant, aber benötigt **Platz** und ist **langsam**.
- Daher gibt es **mehrere** Alternativen:
  - `Data.Text` Unicode-Text, strikt und schnell
  - `Data.Text.Lazy`, Unicode-Text, String kann größer sein als der Speicher
  - `Data.ByteString` Sequenzen von Bytes, kein Unicode, kompakt
- Deshalb `mconcat [...]` oben (`class Monoid`)
- String-Literale können **überladen** werden (`LANGUAGE OverloadedStrings`)
- Mit `pack` und `unpack` Konversion von Strings in oder von `Text`.
- Potenzielle Quelle der Verwirrung: Scotty nutzt `Text.Lazy`, Blaze nutzt `Text`.

HTML

- Scotty gibt nur den Inhalt zurück, aber wir wollen HTML erzeugen.
- Drei Möglichkeiten:
  1. Text selber zusammensetzen: "<h1>Willkommen!</h1>\n<span␣class="..">
  2. Templating: HTML-Dokumente durch Haskell anreichern lassen (Hamlet, Heist)
  3. Zugrundeliegende Struktur (DOM) in Haskell erzeugen, und in Text konvertierten.

Erzeugung von HTML: Blaze

Selbstbeschreibung: https://jaspervdj.be/blaze/

BlazeHtml is a blazingly fast HTML combinator library for the Haskell programming language. It embeds HTML templates in Haskell code for optimal efficiency and composability.

- Kann (X)HTML4 und HTML5 erzeugen.
- Dokument wird als Monade repräsentiert und wird durch Kombinatoren erzeugt:

```haskell
numbers :: Int→ Html
numbers n = docTypeHtml $ do
     H.head $ do
         H.title "Natural␣numbers"
     body $ do
         p "A␣list␣of␣natural␣numbers:"
         ul $ forM_ [1 .. n] (li ∘ toHtml)
```

```haskell
image = img ! src "foo.png" ! alt "A␣foo␣image."
```

- Siehe Tutorial.

Persistenz

- Eine Web-Applikation muss **Zustände** verwalten können
  - Nutzerdaten, Warenbestand, Einkauf, ...
- Üblicher Ansatz: **Datenbank**
  - ACID-Eigenschaften garantiert, insbesondere Nebenläufigkeit
  - Aber: externe Anbindung nötig
- Hier: **Mutable Variables** `MVar a` (nicht durable, aber schnell und einfach)

Nebenläufige Zustände

- Haskell ist **nebenläufig** (hier ein Thread pro Verbindung)
- `MVar` α sind synchronisierte veränderlich Variablen.
- Kann **leer** oder **gefüllt** sein.

```
newMVar  :: α→ IO (MVar α)
readMVar  :: MVar α→ IO α — MVar bleibt gefüllt
takeMVar  :: MVar α→ IO α — MVar danach leer
putMVar   :: MVar α→ α → IO () — Füllt MVar
```

- `readMvar` und `takeMVar` **blockieren**, wenn Variable leer ist
- Erlaubt einfache Synchronisation (vgl. `synchronized` in Java)

Zustand

- Wie können wir den Benutzer **identifizieren**?
- Ein Ansatz: **Cookies**
  - Widerspricht dem REST-Ansatz.
- Hier: über die URL — jeder Benutzer bekommt eine Resource

☞ Siehe Übung 12.**??**

III. Ein Web-Shop für Onkel Bob

Architektur des Web-Shop

Model-View-Controller-Paradigma (Entwurfsmuster):

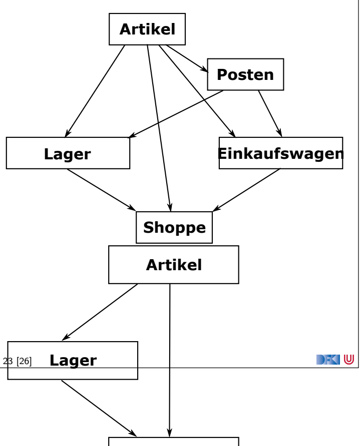
- Das **Model** ist der eigentliche (und persistente) Teil der Anwendung, bestehend aus den Datentypen samt der Funktionen darauf.
- Die **Views** sind Funktionen, die Webseiten aufbauen.
- Der **Controller** übersetzt Anfragen von außen in die Aufrufe der Model-Funktionen, erzeugt aus den Ergebnissen mit den Views Webseiten und schickt diese wieder zurück.

Entwurf der Anwendung

Resource	Methode	Daten
`/`	GET	Home-Page: Angebote anzeigen. Link zu neuem Einkauf
`/einkauf/neu`	GET	Neuen Einkauf beginnt, Einkaufswagen wird zugeteilt. Dann Weiterleitung zu folgender:
`/einkauf/:id`	GET	Einkaufswagen darstellen Link zur Bezahlseite
`/einkauf/:id`	POST	Angegebene Produkte in den Einkaufswagen
`/einkauf/:id/kasse`	GET	Bezahlseite mit Rechnung. Link zur Home-Page
`/einkauf/:id/kaufen`	GET	Bezahlt, Einkaufswagen löschen
`/einkauf/:id/abbruch`	GET	Abgebrochen, Produkte zurück
`/einkauf/lieferung`	POST	Anlieferung von Artikeln
`/einkauf/lager`	GET	Lagerbestand als JSON-Objekt

Model: der Shop

- Einheitliches Interface des Shop.
- Verwaltet Menge von **Einkäufen** (Einkaufswagen),
- Funktionen (Auszug):
  - Neuer Einkaufswagen
  - Produkt in Einkaufswagen
  - Einkauf abschließen/abbrechen
- Rein funktional, ADT `Shop` α
- Änderungen:
  - Einheitliche Mengen
  - Posten nicht mehr als ADT
  - Einkaufswagen nicht mehr als Modul



Controller

- Persistiert den **Zustand** des Shop (nur für Laufzeit des Servers)
- Nutzt `UUID` zur Zuordnung des Einkauf (garantiert eindeutige Bezeichner)
- **Zugriff** auf den Shop:
  - **Ändernd** (muss synchronisieren)
  - **Lesen** (ohne Synchronisation)

View

- Erzeugt Seiten (Templates):

```haskell
homePage :: Text→ [(Posten, Int)]→ Html

shoppingPage :: String→ String→ [Text]→ [(Posten, Int)]
                → Int→ [Posten]→ Html

checkoutPage :: String→ String→ [(Posten, Int)]→ Int→ Html

thankYouPage :: Text→ Html
```

- Weitere Funktionen: Artikelname, Mengeneinheiten, Euros etc.
- Artikel werden über eine eindeutige Kennung (`articleId`) identifiziert.

Zusammenfassung

- Wichtige Prinzipien für Web-Anwendungen:
  - Nebenläufigkeit, Zustandsfreiheit, REST
- Haskell ist für Web-Development gut geeignet:
  - Zustandsfreiheit macht Nebenläufigkeit einfach
  - Bequeme Manipulation von Bäumen
  - Abstraktionsbildung
- Web-Programmierung ist **umständlich**.

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 (24.01.23): Eine praktische Einführung in Scala

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Erinnerung: elektronische Klausur am **13.03.2023 um 14:00/15:45**
- Zwei Slots zu 90 Minuten
- Elektronische Registrierung ab morgen
- Alte Klausuren werden auf der Webseite zur Verfügung gestellt
- Nächste Woche mehr

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

Heute: Scala

- A **sca**lable **la**nguage
- Rein objektorientiert
- Funktional
- Eine "JVM-Sprache"
- Seit 2004 von Martin Odersky, EPFL Lausanne (`http://www.scala-lang.org/`).
- Seit 2011 kommerziell durch Lightbend Inc. (formerly Typesafe)

I. Scala am Beispiel

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {
    var a = x
    var b = y
    while (a != 0) {
      val temp = a
      a = b % a
      b = temp
    }
    return b
  }

def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd (y, x % y)
```

- Variablen, veränderlich (`var`)
  - ***Mit Vorsicht benutzen!***
- Werte, unveränderlich (`val`)
- `while`-Schleifen
  - ***Unnötig!***
- Rekursion
  - Endrekursion wird optimiert
- Typinferenz
  - Mehr als Java, weniger als Haskell
- Interaktive Auswertung

Scala am Beispiel: `02-Rational-1.scala`

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  override def toString = numer +"/"+ denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

- Klassenparameter
- Konstruktoren (`this`)
- Klassenvorbedingungen (`require`)
- private Werte und Methoden
- Methoden, Syntax für Methodenanwendung
- `override` (nicht optional)
- Overloading
- Operatoren
- Singleton objects (`object`)

☞ Siehe Übung 13.1

II. Das Typsystem

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

```scala
abstract class Expr
case class Var(name: String) extends Expr
case class Num (num: Double) extends Expr
case class Plus (left: Expr, right: Expr) extends Expr
case class Minus (left: Expr, right: Expr) extends Expr
case class Times (left: Expr, right: Expr) extends Expr
case class Div (left: Expr, right: Expr) extends Expr
```

```scala
// Evaluating an expression
def eval(expr: Expr): Double = expr match
   case v: Var ⇒ 0     // Variables evaluate to 0
   case Num(x) ⇒ x
   case Plus(e1, e2) ⇒ eval(e1) + eval(e2)
   case Minus(e1, e2) ⇒ eval(e1) - eval(e2)
   case Times(e1, e2) ⇒ eval(e1) * eval(e2)
   case Div(e1, e2) ⇒ eval(e1) / eval(e2)

val e = Times(Num(12), Plus(Num(2.3),Num(3.7)))
```

- `case class` erzeugt
  - Factory-Methode für Konstruktoren
  - Parameter als implizite `val`
  - abgeleitete Implementierung für `toString`, `equals`
  - ...und pattern matching (`match`)
- Pattern sind
  - `case 4` ⇒ Literale
  - `case C(4)` ⇒ Konstruktoren
  - `case C(x)` ⇒ Variablen
  - `case C(_)` ⇒ Wildcards
  - `case x: C` ⇒ getypte pattern
  - `case C(D(x: T, y), 4)` ⇒ geschachtelt

Implementierung algebraischer Datentypen

Haskell:

`data T = C1 | ... | Cn`

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:

`T` → C_1 ... C_n

- Varianten als **Subtypen**
- Problem und Vorteil: **Erweiterbarkeit**
- `sealed` verhindert Erweiterung

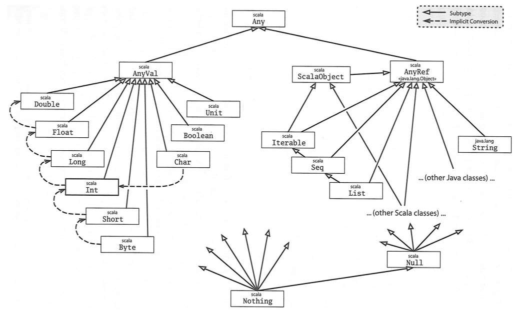
☞ Siehe Übung 13.2

Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

- Werte vs. Objekte
- Scala vs. Java
- `NULL` references

Vererbungshierarchie



Quelle: Odersky, Spoon, Venners: *Programming in Scala*

III. Polymorphie und Vererbung

Parametrische Polymorphie

- Typparameter (wie in Haskell, Generics in Java), Bsp. `List[T]`
- Problem: Vererbung und Polymorphie
- Ziel: wenn `S < T`, dann `List[S] < List[T]`
- **Does not work** — `04-Ref.scala`
- Warum?
  - Funktionsraum nicht monoton im ersten Argument
  - Sei $X \subseteq Y$, dann $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$
  - Sondern $Y \longrightarrow Z \subseteq X \longrightarrow Z$

Typvarianz

`class C[+T]`
- **Kovariant**
- Wenn `S < T`, dann `C[S] < C[T]`
- Parametertyp `T` nur im Wertebereich von Methoden

`class C[T]`
- **Rigide**
- Kein Subtyping
- Parametertyp `T` kann beliebig verwendet werden

`class C[-T]`
- **Kontravariant**
- Wenn `S < T`, dann `C[T] < C[S]`
- Parametertyp `T` nur im Definitionsbereich von Methoden

☞ Siehe Übung 13.3

IV. Strukturierung mit Traits

Traits: `05-Funny.scala`

Was sehen wir hier?

- `Trait` (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- „Abstrakte Klassen ohne Oberklasse“
- Unterschied zu Klassen:
  - Mehrfachvererbung möglich
  - Keine feste Oberklasse (`super` dynamisch gebunden)
  - Nützlich zur Strukturierung (Aspektorientierung)
- Nützlich zur Strukturierung:

  thin interface + *trait* = *rich interface*

  Beispiel: `05-Ordered.scala`, `05-Rational.scala`

Was wir ausgelassen haben. . .

- **Komprehension** (nicht nur für Listen)
- **Gleichheit**: `==` (final), `equals` (nicht final), `eq` (Referenzen)
- *string interpolation*
- **Implizite** Parameter und Typkonversionen
- **Nebenläufigkeit** (Aktoren, Futures)
- Typsichere **Metaprogrammierung**
- Das *simple build tool* `sbt`
- Der JavaScript-Compiler `scala.js`

Schlammschlacht der Programmiersprachen

	Haskell	Scala	Java
Klassen und Objekte	-	+	+
Funktionen höherer Ordnung	+	+	-
Typinferenz	+	(+)	-
Parametrische Polymorphie	+	+	+
Ad-hoc-Polymorphie	+	+	-
Typsichere Metaprogrammierung	+	+	-

Alle: Nebenläufigkeit, Garbage Collection, FFI

Scala — Die Sprache

- Objekt-orientiert:
  - Veränderlicher, gekapselter **Zustand**
  - **Subtypen** und Vererbung
  - **Klassen** und **Objekte**
- Funktional:
  - Unveränderliche **Werte**
  - Parametrische und Ad-hoc **Polymorphie**
  - Funktionen höherer Ordnung
  - Hindley-Milner **Typinferenz**

Beurteilung

- **Vorteile**:
  - Funktional programmieren, in der Java-Welt leben
  - Gelungene Integration funktionaler und OO-Konzepte
  - Sauberer Sprachentwurf, effiziente Implementierung, reiche Büchereien
- **Nachteile**:
  - Manchmal etwas **zu** viel
  - Entwickelt sich ständig weiter
  - One-Compiler-Language, vergleichsweise langsam
- Mehr Scala?
  - Besuchen Sie auch die Veranstaltung **Reaktive Programmierung** (soweit verfügbar)

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 14 (31.01.23): Rückblick und Ausblick

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2022/23

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

Organisatorisches

- Bitte für die Programmierübung (“E-Klausur”) anmelden (stud.ip)
- Bitte an der **Online-Evaluation** teilnehmen (stud.ip)

I. Elektronische Programmierübung

Erinnerung: Scheinkriterien

- Mindestens 50% in den Einzelübungsblättern, in allen Übungsblättern und mindestens 50% in der E-Klausur
- Note: 50% Übungsblätter und 50% E-Klausur
- **Notenspiegel** (in Prozent aller Punkte):

Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	**n/b**

Elektronische Klausur

- **Termin**: 13.03.2023, 14:00– 15:30 und 15:45– 17:15
- **Ort**: Testzentrum am Boulevard neben der Bibliothek
- **Dauer**: 90 Minuten
- **Ablauf**:
  - Einfache Programmierübungen in der Art der Übungsaufgaben
  - Einige Multiple-Choice Fragen als **Bonus**

Aufbau

- Kleine **Programmierübungen**
  - Rahmen vorgegeben, mit kurzen Unit-Tests
  - Tests sind nicht vollständing — Erfüllung **notwendig** aber nicht **hinreichend**.
  - Ziel: Prüfung **elementarer Haskellkenntnisse** (Individualität der Prüfungsleistung)
- **Verständnisfragen**
  - Multiple-Choice-Tests
  - Zusatzaufgaben — Übung auch ohne Verständnisfragen zu bestehen
  - Ziel: Prüfung des **vertieften Verständnisses** des Stoffs
- Wertung: Klausur – 20 Punkte, Verständnisfragen – 5 Punkte

Beispiel Programmierübungen

Definieren Sie eine eine Funktion

```
ostern :: String→ Int
```

die zählt, wie oft in einer Zeichenkette die Zeichenkette `"ei"` enthalten ist.

Beispiel:

```
ostern "ei,␣ei,␣oh,␣eiaiei" ⇝ 4
```

Beispiel Programmierübungen

Definieren Sie eine Funktion

```
concatSnd :: [(a, [b])]→ [(a, b)]
```

welche eine Liste aus Paaren von Elementen und Listen auf eine Liste von Paaren von Elementen abbildet (also die Eingabelisten der zweiten Komponente konkateniert).

Beispiel:

```
concatSnd [(True, "xy"), (False, "foo")] ⇝
   [(True,’x’),(True,’y’),(False,’f’), (False,’o’),(False,’o’)]
concatSnd [(1, [2, 3]), (7, [9, 5])] ⇝
   [(1, 2), (1, 3), (7, 9), (7, 5)]
```


Beispiel Programmierübung

Eine Matrix ist als Liste ihrer Spaltenvektoren dargestellt:

```
data Matrix a = M [[a]]
```

Schreiben Sie eine Funktion

```
row :: Matrix a→ Int→ [a]
```

die die *i*-te Zeile (gezählt ab 1) einer Matrix zurückgibt.

Beispiel:

```
row (M [[3,7,5],[9,2,0],[5,8,1]]) 2 ⇝ [7,2,8]
```


Beispiel Programmierübung

Definieren Sie eine Funktion

```
subseqs :: [a]→ [[a]]
```

welche die nichtleeren Teillisten einer Liste berechnet.

Beispiel:

```
subseqs "pi3" ⇝ ["p", "pi", "pi3", "i", "i3", "3"]
```


Beispiel: Verständnisfrage

Betrachten wir folgende Funktionsdefinition:

```
fun x y z = y z
```

Welche der folgenden Typsignaturen wären für diese Definition typkorrekt?

- ☐ ✓ `fun :: a→ (c→ b)→ c→ b`
- ☐ ✓ `fun :: Int→ ([a]→ Int)→ [a]→ Int`
- ☐ `fun :: a→ b→ c→ b`
- ☐ `fun :: Int→ (b→ c)→ Int→ b`

Beispiel: Verständnisfrage

Betrachten Sie folgende Werte:

```
Otto
Karl Otto "Heinz"
Karl (Karl Otto [1,7]) "17"
```

Für welche der folgenden Typdeklarationen sind diese Werte wohlgetypt:

- ☐ `data T a = Otto | Karl (T a) [a]`
- ☐ ✓ `data T a b = Otto | Karl a b`
- ☐ `data T a = Otto | Karl (T a) String`
- ☐ ✓ `data T a b = Otto a | Karl b [a]`

Beispiel: Verständnisfrage

Betrachten Sie folgenden fehlerhaften Definitionsversuch eines algebraischen Datentypen:

```
data Foo a b = Foo [a] (Foo c Int)
             | bar (Foo Int a)
```

Welche der folgende Aussagen beschreibt tatsächliche Fehler in dieser Definition:

- ☐ ✓ Die Typvariable `c` ist nicht definiert.
- ☐ ✓ Der Konstruktur `bar` ist kleingeschrieben.
- ☐ Der Konstruktor `Foo` heißt genauso wie der Datentyp.
- ☐ Die Typvariable `b` wird auf der rechten Seite der Definition nicht genutzt.

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

count :: Ord a⇒ a→ Tree a→ Int
count _ Leaf = 0
count a (Node l b r) | a < b  = count a l
                     | a == b = 1+ count a r
                     | a > b  = count a r
```

Welche der folgenden Eigenschaften erfüllt `count`?

- ☐ `count` ist **injektiv**
- ☐ ✓ `count` ist **total**
- ☐ `count` ist **partiell**
- ☐ `count` ist **strikt** im **ersten** Argument
- ☐ ✓ `count` ist **strikt** im **zweiten** Argument

Beispiel: Verständnisfrage

Gegeben folgende Funktionsdefinition:

```
f :: a→ [a]→ [a]
f a (b:bs) = b: f a bs
f a []     = [a]
```

Welche Definitionen sind **äquivalent**?

- ☐ `f1 x xs = foldr (:) xs [x]`
- ☐ ✓ `f2 = (flip (++) ∘ (:[]))`
- ☐ `f3 x xs = foldl (flip (:)) xs x`
- ☐ ✓ `f4 a = (++ [a])`

Vorbereitung und Durchführung

- Auf der Webseite sind alte Klausuren verfügbar.
- Für ein **realistisches** Übungsszenario:
  - 90 Minuten Zeit für die Klausuren
  - Windows-10 Rechner, Visual Studio Code
  - Kein Internet (vorher einmal `stack` starten)

II. Rückblick und Ausblick

Warum funktionale Programmierung lernen?

- Funktionale Programmierung macht aus Programmierern Informatiker
- Blick über den Tellerrand — was kommt in 10 Jahren?
- **Herausforderungen** der Zukunft
- Enthält die **wesentlichen** Elemente moderner Programmierung

Zusammenfassung Haskell

Stärken:

- Abstraktion durch
  - Polymorphie und Typsystem
  - algebraische Datentypen
  - Funktionen höherer Ordnung
- Flexible Syntax
- Haskell als Meta-Sprache
- Ausgereifter Compiler
- Viele Büchereien

Schwächen:

- Komplexität
- Büchereien
  - Nicht immer gut gepflegt und integriert
- Nur ein ernsthafter **Compiler**
- Divergierende Ziele:
  - Forschungsplattform **und** nutzbares Werkzeug

Andere Funktionale Sprachen

- **Standard ML** (SML):
  - Streng typisiert, strikte Auswertung
  - Standardisiert, formal definierte Semantik
  - Mehrere aktiv (?) unterstütze Compiler
  - Verwendet in Theorembeweisern (Isabelle, HOL)
  - `http://www.standardml.org/`
- **Caml**, **O'Caml**:
  - Streng typisiert, strikte Auswertung
  - Hocheffizienter Compiler, byte code & nativ
  - Nur ein Compiler (O'Caml)
  - `http://caml.inria.fr/`

Andere Funktionale Sprachen

- **LISP** und **Scheme**
  - Ungetypt/schwach getypt
  - Seiteneffekte
  - Viele effiziente Compiler, aber viele Dialekte
  - Auch industriell verwendet
- **Hybridsprachen**:
  - Scala (Functional-OO, JVM)
  - F# (Functional-OO, .Net)
  - Clojure (Lisp, JVM)
  - Elixir (Erlang VM)

Was spricht gegen funktionale Programmierung?

- Mangelnde Unterstützung:
  - Libraries, Dokumentation, Entwicklungsumgebungen
  - Wird besser (Scala)...
- Programmierung nur kleiner Teil der SW-Entwicklung
- Nicht verbreitet — funktionale Programmierer zu teuer
- Konservatives Management
  - "Nobody ever got fired for buying IBMSAP"

Haskell in der Industrie

- Simon Marlow bei Meta (Facebook), Simon Peyton-Jones bei Microsoft.
- secuCloud in Hamburg (`https://www.secucloud.com/`), now part of Aryaka
- Schaltkreisentwicklung:
  - Bluespec, DSL auf Haskell-Basis; Clash, Haskell mit abhängigen Typen
  - Chisel und SpinalHDL: in Scala eingebettet DSLs
- Galois, Inc: Cryptography (Cryptol DSL)
- Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- Siehe auch: Haskell in Industry (`https://wiki.haskell.org/Haskell_in_industry`)
- Andere Sprachen: Scala, Erlang, Elm, ...

Perspektiven funktionaler Programmierung

- **Forschung**:
  - Ausdrucksstärkere Typsysteme
  - für effiziente Implementierungen
  - und eingebaute Korrektheit (Typ als Spezifikation)
  - Parallelität?
- **Anwendungen**:
  - Eingebettete domänenspezifische Sprachen
  - Zustandsfreie Berechnungen (MapReduce, Hadoop, Spark)
  - **Big Data** and **Cloud Computing**

If you liked this course, you might also like ...

- Die Veranstaltung **Reaktive Programmierung** (findet irregulär stattt)
  - Scala, nebenläufige Programmierung, fortgeschrittene Techniken der funktionalen Programmierung
- Wir suchen **studentische Hilfskräfte** am DFKI, FB CPS
  - Scala und Haskell als Entwicklungssprachen
- Wir suchen **Tutoren für PI3**
  - Im WS 2023/24 — **meldet Euch** bei Thomas Barkowsky (oder bei mir)!

Tschüß!