Kapitel 1: Einführung

Lernziele dieses Kapitels

1. Unterschiede zwischen typisch imperativer und typisch funktionaler Programmierung
2. Nutzen einer Vielfalt an Sprachen und Konzepten
3. Transparenz des reinen funktionalen Programmierstils
4. Historische Entwicklung der funktionalen Programmierung
5. Wiederverwendbarkeit durch Funktionen als Parameter
6. Einordnung von Haskell: Funktionen höherer Ordnung, statisch getypt, rein funktional, nicht-strikte Auswertung (Laziness)

Imperative und Deklarative Sprachen

1. Imperative Sprachen
   - Beschreibung von Programmabläufen durch Operationen auf Zuständen.
   - Programmierung bedeutet im wesentlichen die Spezifikation des Kontrollflusses. Strukturierung durch Schleifen, Prozeduren, abstrakte Datentypen, ... Objektorientierung: Aufteilung des Zustandes, Delegation der Kontrolle an die Objekte, Abstraktion des Berechnungsfortschrittes durch Methodenaufrufe (Funktionen)
2. Deklarative Sprachen
   - Programmierung ist Spezifikation einer Ein-/Ausgaberelation.
   - Kontrollfluss ist dem Programmierer nicht explizit zugänglich; der Ablauf der Berechnung kann durch gezielte Auswahl aus mehreren Alternativen optimiert werden.

Einteilung der Programmiersprachen

1. Imperative Sprachen (Fortran,C,C++,Java)
2. Deklarative Sprachen
   - Programmierung ist Spezifikation einer Ein-/Ausgaberelation.
   - Kontrollfluss ist dem Programmierer nicht explizit zugänglich; der Ablauf der Berechnung kann durch gezielte Auswahl aus mehreren Alternativen optimiert werden.

   (a) Logiksprachen (Prolog)
   - Automatische Suche des Berechnungsweges durch Backtracking.

   (b) funktionale Sprachen (LISP,ML, Haskell)
   - Ein-/Ausgaberelation ist eine Funktion.
   - Programmierung bedeutet Komposition von Funktionen.

Warum funktional programmieren?

- Beherrschung von komplexen Aufgaben → Modularisierung.
- Problem: Wiederverwendung und Kombination von Teilfunktionen.

Lösung: Funktionale Programmierung

- Funktionen als Datenobjekte und Datenobjekte als Funktionen.
- Funktionale Programmierung erlaubt es, Funktionen
  - während der Ausführung
  - selektiv für eine bestimmte Verwendung
  - typsicher

  zu verändern (an die Aufgabenstellung anzupassen).
- Verwendung in der OOP: Vererbung, flexibler: Funktionsobjekte

Kompositionalität und Wiederverwendbarkeit

(notwendig: Parametrisierung durch *Funktionen*)

- der Sortierfunktion Mergesort
  - Vergleichsfunktion zweier Elemente $<$, $>$, $<_{\text{lex}}$, ...
- eines Divide-and-Conquer-Schemas für Mergesort auf Listen
  - Funktion zum Testen auf Trivialfall: Listenlänge=1?
  - Funktion zur Problemteilung: Teilen einer Liste
  - Funktion zur Lösung trivialer Fälle: Identität
  - Funktion zur Vereinigung von Teillösungen: geordnetes Mischen

Funktionsparameter $\rightarrow$ funktionale Programmierung

Programmierung lebt von einer Vielfalt an Sprachen

1. Irrtum: eine Programmiersprache (z.B. Java) reiche aus
   - im Prinzip reicht jede Turing-mächtige Sprache aus
   - in der Praxis: Unterstützung spezieller Anforderungen nötig
2. Irrtum: Konzepte sind Programmierparadigmen fest zugeordnet
   - Polymorphie, Typklassen und Vererbung gibt es auch in Haskell
   - Funktionale Programmierung bedingt nicht explizite Verwendung von Rekursion (Kombinatoren sind vorzuziehen)
   - Funktionsparameter gab es bereits in Pascal
3. Irrtum: Man muss sich für ein Paradigma entscheiden
   - Es gibt Programmiersprachen-Schnittstellen
   - Es gibt Sprachen mit gemischten Stilen (Objective Caml)

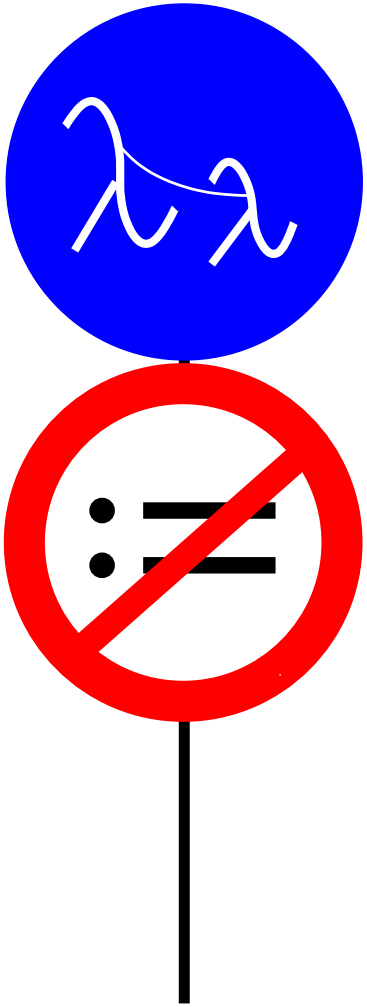
Haskell als Beispielsprache in dieser Vorlesung

- Spracheigenschaften für Sicherheit und Produktivität
  - reine, nicht-strikte funktionale Sprache
  - statisches Typsystem
  - Funktionen höherer Ordnung
- Leistungsfähige Tools, Open-Source, Public-Domain
- Komfortable Syntax, Bsp.: Pythagoräische Tripel

```
pyth :: Int -> [(Int,Int,Int)]
pyth n = [ (a,b,c) | a<-[1..n], b<-[1..n], c<-[1..n], a^2+b^2==c^2 ]

*Main> pyth 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

Reine funktionale Programmierung



\+ **Kompositionalität**

\+ **Wiederverwendbarkeit**

− **Zustandsmanipulation**

− **Seiteneffekte**

Seiteneffekte: Verlust an Wiederverwendbarkeit

Bsp.: Programm in der nicht rein-funktionalen Sprache OCaml

```
open Printf;;
let c = ref 0

let f x = c := !c+1;
          x + !c

let text = let x = 5 in
           c := 0;
           printf "           f(%d) = %d \n"
                              x    (f (x));
           c := 0;
           printf "     f(%d)=f(%d) = %b \n"
                             x     x    (f (x) = f (x));
           c := 0;
           printf "f(%d),f(%d),f(%d) = %d,%d,%d \n"
                       x     x     x    (f (x)) (f (x)) (f (x))
```

Seiteneffekte: Verlust an Wiederverwendbarkeit

Bsp.: Programm in der nicht rein-funktionalen Sprache OCaml

Ausgabe des Programms

```
val c : int ref = contents = 0
val f : int -> int = <fun>
           f(5) = 6
     f(5)=f(5) = false
f(5),f(5),f(5) = 8,7,6
val text : unit = ()
```

Meilensteine der Funktionalen Programmierung

1. λ-Kalkül [Church, Kleene, 1930er]
2. LISP [McCarthy, 1958]
3. APL [IBM, 1962]
4. FP [Backus, 1977]
5. ML [Milner, 1977]
6. Miranda [Turner, 1984]
7. Haskell [Hudak, Wadler u.a., 1990], Standard [1998]
8. MetaOCaml [Sheard, Taha, 2001]
9. FC++ (basierend auf C++ Templates) [McNamara, Smaragdakis, 2001]
10. Generic Haskell [Hinze, Jeuring, Löh, 2004]

λ-Kalkül

- Berechnungen als *Objekte* mathematischer Operationen,
  Bsp.: $(+1) \rightarrow \boxed{(\lambda f \rightarrow f \circ f)} \rightarrow (+2)$
  vollautomatisch in einer funktionalen Programmiersprache

- Äquivalente Begriffe der Berechenbarkeit
  (Äquivalenzbeweise [Church, Kleene, Turing, späte 1930er]):
  - λ-Kalkül [Church, Kleene, frühe 1930er]
  - Rekursive Funktionen [Gödel, 1934]
  - Turingmaschinen [Turing, 1936]
  - Textersetzungssysteme (Markov, Post, u.a.)

LISP

- Verkettete Listen für symb. Differentiation [McCarthy, 1958]
- LISP 1 (*pure* LISP) [McCarthy, 1960] ist funktional, viele spätere LISP-Dialekte sind es nicht (enthalten Zuweisungen)
- Datenobjekte: *S-Expressions* (geklammerte Atome)
- Beispiele (Emacs-LISP):
  - $((\lambda x \rightarrow 2(x+1))\,3)$: `((lambda (x) (* 2 (+ x 1))) 3) = 8`
  - `(car (quote ((A B) (C D) (E F)))) = (A B)`
  - `(cdr (quote ((A B) (C D) (E F)))) = ((C D) (E F))`
  - `(append (quote (A B)) (quote (C D))) = (A B C D)`
- car/cdr: contents of address/data register (IBM 704)
- `(f x)`: Funktionsanwendung, `(quote (f x))`: Datenobjekt

Backus FP (1)

[Backus, 1977]: Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak:

- their primitive word-at-a-time style of programming inherited from their common ancestor — the von Neumann computer,
- their close coupling of semantics to state transitions,
- their division of programming into a world of expressions and a world of statements,
- their inability to effectively use powerful combining forms for building new programs from existing ones, and
- their lack of useful mathematical properties for reasoning about programs.

Backus FP (2)

Bsp.: Skalarprodukt, Implementierung

1. imperativ:

```
c := 0;
for i := 1 step 1 until n do
    c := c + a[i] * b[i];
```

2. funktional: $(/+) \circ (\alpha *) \circ$ `Trans`
   - `Trans`: transpose
   - α: apply-to-all
   - $/$: reduce

Backus FP (3)

Bsp.: Skalarprodukt (`IP`), Ablauf der Berechnung

	$\texttt{IP} :<< 1,2,3 >, < 6,5,4 >>$
Def. IP	$(/+) \circ (\alpha *) \circ \texttt{Trans} :<< 1,2,3 >, < 6,5,4 >>$
Effect $\circ$	$(/+) : ((\alpha *) : (\texttt{Trans} :<< 1,2,3 >, < 6,5,4 >>))$
Apply `Trans`	$(/+) : ((\alpha *) :<< 1,6 >, < 2,5 >, < 3,4 >>)$
Effect α	$(/+) :< * :< 1,6 >, * :< 2,5 >, * :< 3,4 >>$
Apply $*$	$(/+) :< 6,10,12 >$
Effect $/$	$+ :< 6, + :< 10,12 >>$
Apply $+$	$+ :< 6,22 >$
Apply $+$	28

ML

- Meta-Language für den Theorembeweiser *Edinburgh LCF* (Logic for Computable Functions) [Gordon, Milner & Wadsworth 1977]
- Erste Implementierung: übersetzt in LISP, dann interpretiert
- Lange Zeit Lehrsprache für funktionale Programmierung
- Features:
  - Beweisregeln als Funktionen, mit denen gerechnet werden kann → Funktionen höherer Ordnung
  - Polymorphe Typinferenz für abstrakte Typen (Hindley-Milner Typsystem)
  - Auslösung und Behandlung von Exceptions
  - Nicht-funktionale Ein-/Ausgabe (damals keine Alternative)

Haskell

(Die zentrale Programmiersprache in dieser Vorlesung)

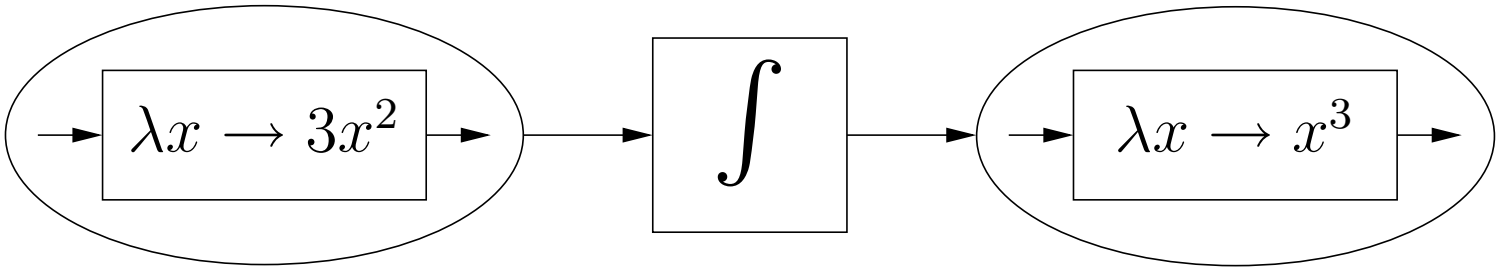
- Public-Domain, Antwort auf kommerzielle Sprache Miranda
- Entwurf durch ein Komitee [Hudak, Wadler, u.a., 1990]
- Grosse Benutzergemeinde `http://www.haskell.org`
- Viele Abstraktionsmechanismen
- Nützliche Tools, z.B. Parsergenerierung, GUIs

Eigenschaften von Haskell

- Funktionen als aktivierbare und modifizierbare Datenobjekte
  + Hoher Abstraktionsgrad → Wiederverwendbarkeit
- Rein funktional (inkl. Ein-/Ausgabe)
  + Einfache Beweise und Transformationen von Programmen
- Statische Typisierung
  + Umfangreiche Fehlervermeidung *möglich*
- Laziness (anforderungsgetriebene Auswertung)
  + Beschreibung zyklischer Strukturen möglich (Hardware)
  − Speicherplatzprobleme bei naiver Programmierung

Funktionen höherer Ordnung

Bsp.: Funktion $\int$ (integriere)



Wiederverwendung durch Verallgemeinerung

(Bsp.: `reduce`)