

# Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

16. Oktober 2007

**Richard Bird** Introduction to Functional Programming using Haskell,  
Prentice Hall, (1998) (sehr gute Einführung in alle Themen) ISBN:  
0-13-484346-0 (bei amazon.uk, ca. 50 Euro)

**Manuel M. T. Chakravarty und Gabriele C. Keller** : Einführung  
in die Programmierung mit Haskell, Pearson Studium (2004)  
Verständliche Einführung in die Programmierung in Haskell

**Graham Hutton** Programming in Haskell, Cambridge University Press  
(2007)  
Grundlagen; kompakt geschrieben

**Peter Pepper und Petra Hofstedt** Funktionale Programmierung  
(2006)

Grundlagen und Konzepte zu strikten und nicht-strikten funktionalen Programmiersprachen. Haskell, ML und Opal werden parallel besprochen

**Hankin, C.** An Introduction to Lambda Calculi for Computer Scientists, King's College Publications, (2004)

**Hutton, G.** Programming in Haskell, Cambridge University Press, (2007)

---

**Simon Thompson** : Haskell: The Craft of Functional Programming

**Thiemann, Peter** Grundlagen der funktionalen Programmierung,  
Teubner Verlag, (1991) (empfohlene Einführung in alle Themen,  
Programmiersprache Haskell)

**Davie, J.** An introduction into functional programming using Haskell,  
Cambridge University Press, (1992) (Eine Einführung in Haskell  
und andere Themen mit verständlichen Beispielen)

**Hinze, Ralf** , Einführung in die funktionale Programmiersprache  
Miranda, Teubner Verlag, (1992)

**Bird, R., Wadler, Ph.**, Introduction to Functional Programming, Prentice Hall, (1988) (sehr gute Einführung in alle Themen, Programmiersprache Miranda)

**Jeuring, J., Meijer. E., eds.** Advanced functional programming, Lecture Notes in Computer Science 925, Springer-Verlag (1995) (Ein Kurs und verschiedene Anwendungen)

**Abelson, H. Sussmann, G.J.** Structure and Interpretation of Programs, MIT Press, (1985) (Einführung in Informatik anhand der Lisp-Variante Scheme) (gibt es auch in deutsch)

**Thompson, Simon,** Miranda, The craft of functional programming, Addison Wesley, (1995) (Miranda-Einführung)

**Plasmeijer, R., von Eekelen, M.,** Functional Programming and Parallel Graph Rewriting, Addison Wesley, (1993) (Funktionale Programmierung aus der Sicht der Gruppe, die Clean entwickelt)

**Runciman, C. Wakeling, C. ,** Applications of Functional Programming, UCL Press London, (1995) (kurz gehaltene Einführung, ausführliche Diskussion von Anwendungsproblemen in Haskell)

**Barendregt, H.P.** The Lambda Calculus: - Its Syntax and Semantics, North Holland, (1984) (Theorie des Lambda-Kalküls)

**Peyton Jones, S.** The Implementation of Functional Programming Languages, Prentice Hall, (1987) (Standardwerk zur Implementierung von verzögert auswertenden funktionalen Programmiersprachen)

**Peyton Jones, S., Lester, D.,** Implementing Functional Languages, Prentice Hall, 1991 (Begleitbuch zu einem Praktikum)

**Richard Bird, Oege de Moor** Algebra of Programming] Behandelt Programmieren und Programmtransformation auf einer abstrakteren Ebene. Ist mit Haskell zusammen verwendbar.

**Davey, B.A., Priestley, H.A.**, Introduction to Lattices and Order, Cambridge University Press, (1990) (Ordnungen, Verbände, usw: geeignet für Betrachtungen zur Semantik)

**Huet, G.**, Logical foundations of functional programming, Addison Wesley, 1990

**O'Donnel, M.** , Equational Logic as a programming language, MIT Press, 1986

**Sleep, M.R., M.J.Plasmeijer, M.J., Eekelen, M.C.J.D.** , Term graph rewriting, Wiley, 1993

**Kluge, W.**, The organization of reduction, data flow, and control flow systems, MIT Press, (1992)

**Steele, Guy L.** COMMON LISP, THE LANGUAGE, second edition,  
digital press (1990)

**Odersky** Funktionale Programmierung, Kapitel D5 in Rechenberg,  
Pomberger: Informatik Handbuch, Hanser-Verlag

**Klop:** Term Rewriting Systems, Handbook of Logic and Computer  
Science

**Barendregt, H.P.** functional programming and the Lambda-calculus,  
Handbook of Theoretical Computer Science, Vol B, Elsevier,  
(1990)

**Mitchell, J.C.** , Type systems for programming languages, Handbook  
of Theoretical Computer Science, Vol B, Elsevier, (1990)

---

**Backus:** Can programming be liberated from the von Neumann style,  
Comm ACM 21, 218-227, (1978)

**Turner, D.A.** , A new implementation technique for applicative languages, Software Practice & Experience 9, pp. 31-49, (1979)

**Turner, D.A.**, Miranda, a non-strict functional programming language with polymorphic types Proc. Conf. on Programming Languages and Computer Architecture, LNCS 201, Springer Verlag, (1985)

**Milner,R.** Theory of type polymorphism in programming. J. of Computer and System Sciences 17, pp. 348-375, (1978)

- Einführung in Haskell
- operationale und kontextuelle Semantik; Haskell, Kernsprachen KFP, KFPT, KFPTS
- monadisches Programmieren  
Concurrent Haskell und GUI unter wxHaskell
- Typsysteme: parametrischer Polymorphismus, Typklassen
- Programmiertechniken + Datenstrukturen  
Rekursion, Iteration, Modularisierung, Listen: map, filter, fold, ...  
List Comprehensions, Bäume, Graphen, .....
- Typberechnung: Milner und iteratives Verrfahren.

Voraussichtliche Inhalte der Folgevorlesung „Semantik und Analyse“:

- Kurzeinführung Operationale Semantik und kontextuelle Gleichheit
- Analysetechniken: Striktheitsanalyse  
Parallelisierung
- denotationale Semantik und Abstrakte Interpretation
- Konkurrente Funktionale Programmiersprachen (ML-Varianten)  
Operationale Semantik und kontextuelle Gleichheit  
Nichtdeterministische Funktionale Programmiersprachen  
Korrektheit von Implementierungen konkurrenter Sprachen

Programmierspracheneinteilung: imperativ oder deklarativ

## Imperative Programmiersprachen

Folge von Anweisungen

bewirken Gesamteffekte auf die Umgebung (Hauptspeicher, Drucker, Datenbank, Bildschirm, usw.)

Programmieren: **WIE** ist eine Aufgabe zu lösen

## Deklarative Programmiersprachen

Beschreibung des gewünschten Ergebnisses : **WAS** soll erreicht werden

Spezifikationen in verschiedener Form sind deklarativ.

Logische und Funktionale Programmiersprachen

## von-Neumann Programmieridee

**Programm** = Folge von Anweisungen zum Verändern von Werten  
im Hauptspeicher

**Auswertung:** Folge von Zuweisungen, Abfrage von Bedingungen,  
und Verändern von Werten und/oder Programmzähler

## Imperative Programmiersprachen

Folge von Anweisungen manipuliert den Zustand (Speicherinhalt)  
und die Steuerung (Programmzähler)

Programmiersprachen: Assembler, Fortran, Cobol, Algol, Pascal, C,  
C++ , ...

## Objektorientiertes Programmieren

Klassen, Methoden, Vererbung, Objekte,

Wesentliche Neuerung: Strukturierung des Programms und der Daten durch Angabe von Klassen,

von-Neumann Programmieridee ist Teil des Konzeptes:

## Logisches Programmieren (Prolog)

Basiert auf einer Menge von prädikatenlogischen Formeln: Fakten und Regeln (HornklauseIn) als Programm.

Ausführung: Folge von logischen Schlüssen (Resolution)

## Funktionales Programmieren

**Programmieren** = Definition von Funktionen

**Ausführung** = Auswerten von Ausdrücken

**Resultat** Einziges Resultat ist der Rückgabewert

*Resultat = Funktion(Argument<sub>1</sub>, ..., Argument<sub>n</sub>)*

Wesentliche Elemente des funktionalen Programmierens:

- Funktionsaufruf
- Komposition von Funktionen
- rekursives Programmieren

In nicht-strikten FPS gibt es **keinen** expliziten globalen Zustand, **keine** Zuweisungen.

Haskell's monadisches Programmierkonzept erlaubt (kontrollierte) Seiteneffekte und IOs

## Pures Funktionales Programmieren

Prinzip der *referentiellen Transparenz*:

Gleiche Funktion angewendet auf gleiche Argumente ergibt gleichen Wert, d.h. nur die Argumente bestimmen den Wert. Funktionen verändern sich nicht durch die Verarbeitung.

**Variablen** bezeichnen Werte, nicht Speicherplätze; Ausdrücke bezeichnen Werte

**Substitutionsprinzip:** Im Programmtext können Ausdrücke mit gleichem Wert ausgetauscht werden ohne Konsequenz für den Wert des Gesamtausdrucks.

Änderung des Speichers (der Programmierumgebung)  
während eines Funktions / Prozeduraufrufs,  
wobei diese Änderung nach dem Aufruf noch Auswirkungen hat.

- Seiteneffekte werden unterstützt von Hardware, sind effizient; die Übergabe von Werten ist einfach.
- Ein Nachteil ist, dass die Sequentialisierung des Programms durch den Programmierer erforderlich ist, was eine Parallelisierung verhindern bzw. erheblich erschweren kann.
- Die Semantik einer Funktion ist nur erklärbar durch Veränderung des globalen Zustandes (i.a. des Hauptspeichers).

- Eine definierte Funktion / Prozedur hat nicht-lokale Eigenschaften
- Verifikation von Prozeduren ist schwierig und komplex
- Verhalten der Prozedur ist nicht nur von Argumenten, sondern vom aktuellem Speicherinhalt (der aktuellen Umgebung) beeinflusst.
- Die Wiederverwendbarkeit von Funktionen/Prozeduren ist sehr erschwert

- Funktionen höherer Ordnung: Funktionen als Argumente, und als manipulierbare Objekte.
- operationale und denotationale Semantik: zahlreiche korrekte Programmtransformationen und Optimierungen
- Parallelisierbarkeit: einfache Analysen
- parametrisch polymorphes Typsystem und Typklassensystem: Zahlreiche Fehler werden bereits in der Programmierphase vermieden
- aktive Forschergemeinde

- Effizienz zur Laufzeit erfordert hohen Aufwand beim Schreiben eines Compilers.

- Die Schnittstellen zu GUIs, Betriebssystem und anderen IO-Medien erfordern

Kompromiss zwischen zwei Prinzipien:

Seiteneffektfreiheit und deklarative (nichtsequentielle) Spezifikation.

und

Seiteneffekte, Sequentialität

⇒ Haskell's **monadisches Programmieren**

**Modularisierung und Information-Hiding** ist in funktionalen Programmiersprachen als Konzept integriert.

**Referentielle Transparenz** macht die Wiederverwendung von Funktionalität leicht möglich.

**Starke Typisierung** gibt dem Programmierer (und Anwender) die Sicherheit, dass ein großer Teil der möglichen Fehler ausgeschlossen ist.

**Typklassen in Haskell** mit Vererbung sind sehr analog zum **Klassensystem** der OO-Sprachen;  
Ausdrücke haben einen Typ; der Typ gehört zu einer Klasse;  
abhängig von der Klasse werden andere Funktionen (Methoden) benutzt

**von-Neumann-Konzepte** Die folgenden von-Neumann-Anteile fehlen in reinen funktionalen Sprachen:  
Objekt-Identität, (d.h. Objekte, die man auf Pointer-Gleichheit prüfen kann), und  
Veränderbarkeit von Objekten.

- Klassen und Methoden
- Ausdrücke mit Wert
- Zuweisungen
- automatische Speicherverwaltung
- Typsystem, aber dynamische Typfehler sind möglich
- In Java 1.5: polymorphe Typen
- keine Pointer-Arithmetik

$$p(a_1, \dots, a_n)$$

**(call by name)** Funktion  $p$  bekommt  $n$  Argumente (als Pointer bzw. Speicherplätze)  
der Rückgabe-Wert ist (bzw. die Rückgabewerte) an einer bekannten Adresse.

(Seiteneffekte sind notwendig)

**(call by value)** Funktion  $p$  sieht nur die Argumentwerte, Speicherzugriffe sind möglich.

Seiteneffekte sind nicht notwendig, aber i.a. möglich durch Veränderung von globalen Variablen

Pascal, Lisp, Scheme, ML, ...

**(call by value, seiteneffektfrei)** Funktion erhält nur die Argumentwerte, liefert Wert zurück

keine Seiteneffekte

pures Prolog, pures Scheme, pures Lisp, pures ML pur = seiteneffektfrei

**(call by name) bzw. nicht-strikt** Funktion bekommt  $n$  Argumente (i.a. als Ausdrücke), und setzt diese in den Rumpf der Funktionsdefinition ein. Wert wie bei call-by-need (lazy), aber Ausdrücke werden manchmal mehrfach ausgewertet. Unendliche Listen sind möglich.

**(call by value) bzw. strikt** Funktion erhält nur die Argumentwerte, und setzt diese in den Rumpf ein. Keine Mehrfachauswertung, aber auch: keine direkt verfügbaren unendlichen Listen ( Ströme). ML, pures Scheme, pures Lisp, ...

**(call by need) bzw. verzögert (lazy)** Funktion sieht die Argumentausdrücke, benutzt aber Sharing beim Auswerten.

Keine Mehrfachauswertung von Ausdrücken;  
unendliche Listen sind möglich.

Sprachen: Haskell, Clean, Miranda

spezifiziert den Effekt von Einzelbefehlen auf die Variablen-Umgebung  
operationale Semantik = Spezifikation eines Interpreters

Gleichheit auf den Programmen ist durch kontextuelle Gleichheit  
definierbar:

Zwei Programmfragmente  $P_1, P_2$  sind **gleich**,  
wenn in jedem Programm  $P$  das Unterprogramm  $P_1$   
durch  $P_2$  ersetzt werden kann,  
ohne dass sich die Terminierung verändert.

Bildet Anweisungen bzw. Funktionen ab auf Ausdrücke des Lambda-Kalküls bzw eines erweiterten Lambda-Kalküls.

<b>Programm / Ausdruck</b>	<b>Bedeutung</b>
Zeichenketten	mathematische Objekte (denotationale Semantik)
$(s\ 0) + s\ (s\ 0)$	$1 + 2$
$q\ x := x * x$	$q$ wird abgebildet auf die Quadratfunktion über Zahlen
$f\ x := x + f\ x$	$f$ undefiniert (terminiert nicht)

## Typsystem:

**statisch:** zur Compilezeit

**dynamisch:** Test zur Laufzeit

**stark:** Typfehler werden vom Compiler nicht toleriert, alle Ausdrücke sind getypt

**schwach:** Typfehler werden vom Compiler toleriert, manche Ausdrücke sind ungetypt

**monomorph:** Funktionen haben einen festen Typ

**polymorph:** Funktionen haben schematischen Typ (Typvariablen)

**Überladung:** gleicher Funktionsname; typabhängige Implementierung

**Typklassen:** Zusammenfassung von Typen / Funktionen

## Status der Objekte:

**erster Ordnung:** nur Datenobjekte können Argumente/Werte sein.

**höherer Ordnung:** Funktionen sind als Objekte zugelassen können Argument/Wert von Funktionen sein

## Auswertung

**strikt:** Argumente werden bei Übergabe ausgewertet

**nicht-strikt:** (lazy, verzögert): nicht ausgewertete Argumente möglich

**Speicherverwaltung** automatisch / “garbage collector“

**Argumentübergabe mittels Selektoren** / pattern matching (keine Selektoren notwendig)

# Einige Programmiersprachen mit funktionalen Möglichkeiten

Für einige Programmiersprachen sind auch Web-Adressen angegeben.  
Die Auflistung ist eine Auswahl und nicht vollständig.

**Haskell** nicht-strikt, polymorph statisch stark getypt + Typklassen, patterns, Funktionen höherer Ordnung; keine Seiteneffekte,  
[www.haskell.org](http://www.haskell.org)

**CLEAN:** nicht-strikt, polymorph statisch stark getypt; Funktionen höherer Ordnung; patterns, keine Seiteneffekte aktive Entwicklung  
[clean.cs.ru.nl](http://clean.cs.ru.nl)

**Lisp** Common-Lisp Standard: Allegro-Common-Lisp  
CLOS: Common Lisp Object system  
ungetypt; strikt, Funktionen höherer Ordnung; Seiteneffekte, patterns teilweise möglich

**Scheme:** Lisp-Variante: ungetypt; strikt, Fun. höherer Ordnung; Seiteneffekte, Patterns in manchen Varianten, aktive Entwicklungen:  
[www.schemers.org](http://www.schemers.org)

**ML:** Standard ML strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; Seiteneffekte, patterns, aktive Entwicklung  
<http://www.smlnj.org/>  
Variante CAML: [caml.inria.fr/](http://caml.inria.fr/)

**ID** lazy, ungetypt; erster Ordnung; Seiteneffekte

**Erlang** strikt, ungetypt, eingeschränkt höherer Ordnung, funktionale Sprache von Ericsson, Seiteneffekte Anwendung in Telekommunikationsanlagen.

**Sisal:** monomorph getypt, strikt, erster Ordnung, implizite Parallelisierung, gemeinsamer Speicher.

**HOPE** strikt statisch stark getypt; Fun. erster Ordnung; patterns, Seiteneffekte?

**LML** (Lazy-ML) nicht-strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; patterns, keine Seiteneffekte

**OPAL** strikt, pur, funktional, higher-order, polymorphe Typen, (wie ML), aber zusätzlich algebraische Spezifikationen

**CAL** nicht-strikt, funktional, higher-order, polymorphe Typen, eingebettete Java-Typen, eingeschränkte Seiteneffekte ,  
[http://homepage.mac.com/luke\\_e/FileSharing13.html](http://homepage.mac.com/luke_e/FileSharing13.html)

**Miranda** nicht-strikt, polymorph statisch stark getypt; Fun. höherer Ordnung; patterns, keine Seiteneffekte, trademarked by D. Turner.  
<http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/Announce>

**NEL** nicht-strikt, polymorph, statisch stark getypt; Fun. höherer Ordnung; patterns, eingeschränkte Seiteneffekte, industrielle Anwendungen

**C, Pascal** monomorph bzw. schwach getypt, teilweise statisch; teilweise strikt, erster Ordnung; Seiteneffekte keine automatische Speicherverwaltung, keine patterns

**Java** Typsystem: dynamisch getypt, teilweise statisch; strikt, erster Ordnung; Seiteneffekte automatische Speicherverwaltung