

# **Einführung in die Funktionale Programmierung: Einleitung & Motivation**

Prof Dr. Manfred Schmidt-Schauß

WS 2025/26

- 1 Motivation
  - Übersicht Programmierparadigmen
  - Vorteile funktionaler Programmierung
- 2 Funktionale Programmiersprachen
  - Eigenschaften von funktionalen Programmiersprachen
  - Überblick über verschiedene funktionale Programmiersprachen
- 3 Haskell

# Klassifizierung von Programmierparadigmen und -sprachen

Insbesondere: Was unterscheidet Funktionale Programmiersprachen  
von anderen?

# Klassifizierung von Programmiersprachen (1)

## Imperative Programmiersprachen:

- lat. imperare = **befehlen**
- Programm = Folge von Anweisungen (Befehlen)
- Ausführung = **Sequentielle Abarbeitung der Befehle**,  
Befehle **ändern** den **Zustand** des Speichers  
(Wirkung nur über **Seiteneffekte**)

```
X:=10;  
while X < 100 do  
    X:=X+1;
```

- Passend zur von-Neumann Rechnerarchitektur
- Programm beschreibt **WIE** der Zustand modifiziert werden soll
- Prozedurale Programmiersprachen:  
**Strukturierung** des imperativen Codes durch Prozeduren

# Klassifizierung von Programmiersprachen (2)

## Objektorientierte Programmiersprachen:

- Programm: Definition von **Klassen**  
(Klassen = Methoden + Attribute),  
Strukturierung durch **Vererbung**
- Ausführung=veränderliche **Objekte** (Instanzen von Klassen)  
**versenden Nachrichten** untereinander  
sequentielle Abarbeitung innerhalb der Methoden

```
Fahrzeug auto = new Fahrzeug();  
auto.setSpeed(180);
```

- **Zustandsänderung** des Speichers durch Veränderung der Objekte
- werden oft zu den imperativen Sprachen gezählt

# Klassifizierung von Programmiersprachen (3)

## Deklarative Programmiersprachen

- lat. declarare = **erklären**
- Programm = Beschreibt nicht **WIE**, sondern eher **WAS** berechnet werden soll.
- Ausführung= I.a. **keine** Manipulation des Zustands, sondern **Wertberechnung** von Ausdrücken
- Lassen sich grob in **funktionale Programmiersprachen** und **logische Programmiersprachen** aufteilen.

# Klassifizierung von Programmiersprachen (4)

## Logische Programmiersprachen

- Prominentes Beispiel: **Prolog**
- Programm = Menge von **logischen Formeln**  
(in Prolog: Hornklauseln der Prädikatenlogik)  
  
`bruder(X,Y):- istSohn(X,Z), istSohn(Y,Z), X/=Y.`
- Ausführung = **Herleitung neuer Fakten** mittels logischer Schlüsse (z.B. Resolution)
- Ausgabe von (mehreren) Antworten auf Anfragen.

# Klassifizierung von Programmiersprachen (5)

## Funktionale Programmiersprachen

- Programm = Funktionsdefinitionen + Ausdruck (main)

```
quadrat x = x * x  
main = quadrat 5
```

- Ausführung = Auswerten des Ausdrucks (main).
- Resultat der Auswertung: Ein einziger Wert



# Klassifizierung von Programmiersprachen (5)

## Funktionale Programmiersprachen (Forts.)

- In **reinen** funktionalen Programmiersprachen:  
Keine (sichtbare) Manipulation des Zustands
- D.h. **keine Seiteneffekte**, es gilt **referentielle Transparenz**
- Variablen stellen keine Speicherplätze dar, sondern unveränderliche Werte

### Referentielle Transparenz

Gleiche Funktion auf gleiche Argumente liefert stets das gleiche Resultat

# Warum Funktionale Programmierung

Welche Motivation gibt es, sich diese Sprachen anzuschauen?  
Welche Vorteile bieten diese Sprachen?

# Warum Funktionale Programmierung? (1)

- **Andere Sichtweise der Problemlösung:**  
abstraktere Lösungsformulierung, auch hilfreich beim imperativen Programmieren
- **Elegantere Problemlösung:**  
Funktionale Programme sind meistens kürzer und verständlicher, da „mathematischer“
- **Mathematisch einfacher handhabbar:**  
Aussagen wie “mein Programm funktioniert korrekt” oder “Programm A und Programm B verhalten sich gleich” einfacher formulierbar (und in der Regel leichter nachweisbar)
- **Weniger Laufzeitfehler bei starker und statischer Typisierung:**  
Fehlerentdeckung zur Compilezeit

# Warum Funktionale Programmierung? (2)

- **Keine Fehler durch Speicherüberschreiben**
- **Debuggen einfach:** Fehler können unabhängig vom Speicherzustand reproduziert werden
- **Testen einfach:** Unit-Test einfach möglich: Funktionen können unabhängig getestet werden
- **Wiederverwendbarkeit:** Funktionen höherer Ordnung haben hohen Abstraktionsgrad.
- **Compiler Optimierungen:** als korrekt nachweisbar via Semantik.

## Funktion höherer Ordnung

Eingabeparameter (und Ausgabe) dürfen selbst Funktionen sein

# Warum Funktionale Programmierung? (3)

**Beispiel:** Summe und Produkt der Elemente einer Liste

# Warum Funktionale Programmierung? (3)

**Beispiel:** Summe und Produkt der Elemente einer Liste

summe liste =

**if** leer(liste) **then** 0 **else**

„erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

# Warum Funktionale Programmierung? (3)

## Beispiel: Summe und Produkt der Elemente einer Liste

summe liste =

**if** leer(liste) **then** 0 **else**

„erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

produkt liste =

**if** leer(liste) **then** 1 **else**

„erstes Element der Liste“ \* (produkt „Liste ohne erstes Element“)

# Warum Funktionale Programmierung? (3)

## Beispiel: Summe und Produkt der Elemente einer Liste

summe liste =

if leer(liste) then 0 else

„erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

produkt liste =

if leer(liste) then 1 else

„erstes Element der Liste“ \* (produkt „Liste ohne erstes Element“)



# Warum Funktionale Programmierung? (3)

**Beispiel:** Summe und Produkt der Elemente einer Liste

summe liste =

if leer(liste) then 0 else

„erstes Element der Liste“ + (summe „Liste ohne erstes Element“)

produkt liste =

if leer(liste) then 1 else

„erstes Element der Liste“ \* (produkt „Liste ohne erstes Element“)

*Besser abstrahiert:*

reduce e op liste =

if leer(liste) then e else

„erstes Element der Liste“ op (reduce e op „Liste ohne erstes Element“)

summe und produkt sind dann nur Spezialfälle:

summe liste = reduce 0 (+) liste

produkt liste = reduce 1 (\*) liste

# Warum Funktionale Programmierung? (4)

- **Einfache Syntax:** I.a. haben funktionale Programmiersprachen wenige syntaktische Konstrukte / Schlüsselwörter
- **Kompositionalität:** Funktionskomposition erlaubt strukturiertes Zusammensetzen großer Funktionalitäten
- **Parallelisierbarkeit:** Reihenfolge der Auswertung von funkt. Programmen oft irrelevant, bis auf Datenabhängigkeit, daher gut parallelisierbar.
- **Modularität:** Die meisten FP bieten Modulsysteme zur Strukturierung und Kapselung des Codes.

# Warum Funktionale Programmierung? (5)

- **Multi-Core Architekturen:** FP “im Trend”:  
Seiteneffektfreiheit impliziert:  
Race Conditions oder Deadlocks können (im reinen Teil) nicht auftreten, da das nur bei Speicherzugriffen auftreten kann.
- **Große Forschungsgemeinde, neueste Entwicklungen**
- Auch moderne imperative Programmiersprachen übernehmen **FP-Konzepte**, z.B. Java Generics, etc.
- **Concurrent Haskell** verbindet reines funktionales Programmieren mit konkurrentem.
- **Software Transactional Memory:**  
Korrektheitsnachweise werden ermöglicht

# Klassifizierung Funktionaler Programmiersprachen

Anhand welcher Eigenschaften lassen sich FP unterscheiden?

# Wesentliche Eigenschaften von FP (1)

## Pure vs. Impure:

- Pure FP: Keine Seiteneffekte erlaubt
- Impure FP: Seiteneffekte möglich
- Daumenregel:
  - Pure FP = nicht-strikte Auswertung
  - Impure FP = strikte Auswertung
- Haskell: pure FP + (sauber abgetrennte Seiteneffekte)  
nicht-strike Auswertung

# Wesentliche Eigenschaften von FP (2)

## strikte vs. nicht-strikte Auswertung

- Strikte Auswertung (auch call-by-value oder applikative Reihenfolge):  
Definitionseinsetzung **nach** Argumentauswertung
- Nicht-strikte Auswertung (auch call-by-name, call-by-need, lazy, verzögert, Normalordnung):  
Definitionseinsetzung **ohne** Argumentauswertung.

# Wesentliche Eigenschaften von FP (2)

## strikte vs. nicht-strikte Auswertung

- Strikte Auswertung (auch call-by-value oder applikative Reihenfolge):  
Definitionseinsetzung **nach** Argumentauswertung
- Nicht-strikte Auswertung (auch call-by-name, call-by-need, lazy, verzögert, Normalordnung):  
Definitionseinsetzung **ohne** Argumentauswertung.

**Beispiel:** `quadrat  $x = x * x$`

- Strikte Auswertung von `quadrat (1 + 2)`:  
`quadrat (1 + 2) → quadrat 3 → 3 * 3 → 9`
- Nicht-Strikte Auswertung von `quadrat (1 + 2)`  
`quadrat (1 + 2) → (1 + 2) * (1 + 2) → 3 * (1 + 2) → 3 * 3 → 9`

# Wesentliche Eigenschaften von FP (3)

## Eigenschaften des Typsystems

**stark** vs. **schwach**:

- stark: Alle Ausdrücke müssen getypt sein
- schwach: (Manche) Typfehler werden vom Compiler akzeptiert bzw. manche Ausdrücke dürfen ungetypt sein

**dynamisch** vs. **statisch**:

- statisch: Typ des Programms muss zur Compilezeit feststehen.  
Konsequenzen: Keine Typfehler zur Laufzeit, Typinformation zur Laufzeit unnötig
- dynamisch: Typermittlung zur Laufzeit  
Konsequenzen: Typfehler zur Laufzeit möglich, Typinformation zur Laufzeit nötig.

**monomorph** vs. **polymorph**:

- monomorph:: Funktionen / Ausdrücke haben festen Typ
- polymorph: Schematische Typen (mit Typvariablen) erlaubt.  
Funktionen haben viele Typen



# Wesentliche Eigenschaften von FP (4)

## First-Order vs. Higher-Order Funktionen

- first-order: Argumente von Funktionen müssen Datenobjekte sein
- higher-order: Auch Funktionen als Argumente erlaubt

## Speicherverwaltung

- automatische Speicherbereinigung (Garbage Collector): Benutzter Speicher wird automatisch freigegeben.

# Überblick

## Funktionale Programmiersprachen

Eine (unvollständige) Aufzählung von verschiedenen funktionalen Programmiersprachen

# Überblick Funktionale Programmiersprachen (1)

- **Lisp**: Sprachfamilie, List processing, Ende 1950er Jahre, von John McCarthy). Sprache in Anlehnung an den Lambda-Kalkül, ungetypt, strikt.
- **Common Lisp**: Lisp-Dialekt, erste Ausgabe 1984, endgültiger Standard 1994, dynamisch typisiert, auch OO- und prozedurale Anteile, Seiteneffekte erlaubt, strikt,
- **Scheme**: Lisp-Dialekt, streng und dynamisch getypt, call-by-value, Seiteneffekte erlaubt, eingeführt im Zeitraum 1975-1980, letzter Standard aus dem Jahr 2007,
- **Clojure**: relativ neue Sprache, Lisp-Dialekt, der direkt in Java-Bytecode compiliert wird, dynamisch getypt, spezielle Konstrukte für multi-threaded Programmierung (software transactional memory system, reactive agent system)

# Überblick Funktionale Programmiersprachen (2)

- **ML**: Sprachfamilie, Meta Language, statische Typisierung, Polymorphismus, automatische Speicherbereinigung, i.A. call-by-value Auswertung, Seiteneffekte erlaubt, entwickelt von R. Milner 1973
- **SML**: Standard ML, ML-Variante: call-by-value, entwickelt 1990, letzte Standardisierung 1997, Sprache ist vollständig formal definiert, Referenz-Implementierung: Standard ML of New Jersey
- **OCaml**: Objective Caml, Weiterentwicklung von Caml (Categorically Abstract Machine Language), ML-Dialekt, call-by-value, stark und statische Typisierung, polymorphes Typsystem, automatische Speicherbereinigung, nicht Seiteneffekt-frei, unterstützt auch Objektorientierung, entwickelt: 1996
- **Alice ML**: ML Variante, die sich als Erweiterung von SML versteht, mit Unterstützung für Nebenläufigkeit durch sogenannte Futures, call-by-value Auswertung, stark und statisch typisiert, polymorphes Typsystem, entwickelt an der Uni Saarbrücken 2000-2007
- **F#**: Funktionale Programmiersprache entwickelt von Microsoft ab 2002, sehr angelehnt an OCaml, streng typisiert, auch objektorientierte und imperative Konstrukte, call-by-value, Seiteneffekte möglich, im Visual Studio 2010 offiziell integriert

# Überblick Funktionale Programmiersprachen (3)

- **Haskell**: Pure funktionale Programmiersprache, keine Seiteneffekte, strenges und statisches Typsystem, call-by-need Auswertung, Polymorphismus, Komitee-Sprache, erster Standard 1990, Haskell 98: 1999 und nochmal 2003 leichte Revision, neuer Standard Haskell 2010 (veröffentlicht Juli 2010),
- **Clean**: Pure Nicht-strikte funktionale Programmiersprache, stark und statisch getypt, polymorphe Typen, higher-order.

**Dependent Types**: Typen können von Daten abhängen. z.B. Array-Dimension  $n$

- **Agda**: Nicht-strikte funktionale Programmiersprache, higher-order, pure, stark und statisch getypt, polymorphe Typen, **dependent types**. In Agda kann man auch Korrektheitsbeweise von Programmen direkt formulieren.
- **LEAN**: Pure funktionale Programmiersprache, keine Seiteneffekte, higher-order, strikt, streng getypt, mit **Dependent Types**;  
Ziel: Spezifikation und Auswertung zum mathematischen Beweisen.

# Überblick Funktionale Programmiersprachen (4)

- **Erlang**: Strikte funktionale Programmiersprache, dynamisch typisiert, entwickelt von Ericsson für Telefonnetze, sehr gute Unterstützung zur parallelen und verteilten Programmierung, Ressourcen schonende Prozesse, entwickelt ab 1986, open source 1998, hot swapping (Code Austausch zur Laufzeit), Zuverlässigkeit “nine nines” = 99,9999999 %
- **Scala**: Multiparadigmen Sprache: funktional, objektorientiert, imperativ, 2003 entwickelt, läuft auf der Java VM, funktionaler Anteil: Funktionen höherer Ordnung, Anonyme Funktionen, Currying, call-by-value Auswertung, statische Typisierung
- **Curry** Logisch-funktionale Programmiersprache, die auch Nichtdeterminismus erlaubt. <https://www.informatik.uni-kiel.de/~mh/FLP/>

# In der Vorlesung verwenden wir Haskell

## Haskell

- [Webseite](http://www.haskell.org): [www.haskell.org](http://www.haskell.org)
- Interpreter / Compiler:  
[GHCi](http://www.haskell.org/ghc) bzw. [GHC](http://www.haskell.org/ghc): [www.haskell.org/ghc](http://www.haskell.org/ghc)
- Zum Installieren von Haskell:  
[www.haskell.org/downloads](http://www.haskell.org/downloads)
- Dokumentation der Standardbibliotheken:  
[www.haskell.org/ghc](http://www.haskell.org/ghc)

# Ziele der Vorlesung

## Ziele

- Haskell gut kennen lernen
- Konzepte von Haskell verstehen  
Dazu gehören: Syntax, Semantik, Typsystem
- Wir betrachten insbesondere:  
Kernsprachen als Modelle für Haskell
- Grundlagen der Funktionalen Programmierung verstehen



- Eigenschaften und Varianten funktionaler Programmiersprachen
- Der Lambda-Kalkül
- Kernsprachen funktionaler Programmiersprachen
- Syntax, Semantik und Auswertung
- call-by-need Auwertung, Definition und Eigenschaften
- Kernsprache für Haskell
- Tiefergehende Konzepte und Konstrukte von Haskell
- Probabilistische Erweiterung(en)
- Haskell's Typklassensystem
- Typisierungsverfahren bzw. -algorithmen
- Monadisches Programmieren und Monadisches I/O in Haskell
- Exkurs: funktionales Beweissystem Lean