9 Funktionale Sprachen: Lisp

nach Peter Thiemann: *Grundlagen der Funktionalen Programmierung*, Teubner, 1994

Lisp

- 1960 von **McCarthy** am MIT entwickelt
- **klassischer Artikel**: J. McCarthy: *Recursive functions of symbolic expressions and their computation by machine, Part I*., Communications of the ACM, 3(4), 184-195, 1960
- sehr **einfacher Interpretierer**: Funktionen **`eval`** (Ausdruck) und **`apply`** (Aufruf)
- sehr **einfache Notation für Daten und Programm**: Zahlen, Symbole, Listen als Paare
  Preis der Einfachheit: Klammerstruktur wird schon bei kleinen Programmen unübersichtlich
- HOF erfordern spezielle Notation
- erste Sprache mit automatischer **Speicherbereinigung (garbage collection)**
- **keine Typisierung (nur Unterscheidung zwischen Atom und Liste)**
- **dynamische Namensbindung**
- **ursprünglich call-by-name**
- auch imperative Variablen
- moderne Dialekte: Common Lisp, Scheme
  call-by-value und statische Namensbindung

Funktionale Sprachen: FP, ML, SML

FP

- Theoretische, einflussreiche Arbeit, Turing Award Lecture:
  J. Backus: *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, 21(8), 613-641, 1978
- In FP gibt es **nur Funktionen** - keine Daten; Berechnungen Kombination von Funktionen

ML, SML

- erster ML-Compiler 1974
  SML 1990: R. Milner, M. Tofte, R. Harper: The Definition of Standard ML, MIT Press, 1990
- erste (bedeutende) funktionale Sprache mit **strenger statischer Typbindung**,
  Hindley/Milner **Typsystem mit parametrischer Polymorphie**
- **Typinferenz**
- **statische Namensbindung**
- **HOF und Currying uneingeschränkt**
- **strikte Aufruf-Semantik (call-by-value)**
- **abstrakte Datentypen, Module, Funktoren**
- **Ausnahmebehandlung**
- **getypte Referenzen** (imperative Aspekte)

Funktionale Sprachen: Miranda, Haskell

Miranda™

- Turner 1985; kommerziell vertrieben
- nicht-strikt (lazy), polymorphe Typen, implementiert mit SKI-Reduktion
- ungewöhnliche Syntax, keine Lambda-Ausdrücke

Haskell

- Entwicklung begann 1987
- **Stand der Technik** in den funktionalen Sprachen
- **statisches Typsystem** mit **parametrischer Polymorphie** und **Überladung durch Typklassen**, **Typinferenz**
- **statische Namensbindung**
- **nicht-strikte Aufruf-Semantik (call-by-need)**
- **HOF und Currying uneingeschränkt**
- voll ausgebautes **Modulsystem**, auch mit **separater Übersetzung**
- **rein-funktionale (Seiten-effektfreie) E/A**: Ströme, Continuations, Monaden
- Syntax für **kompakte Notation**

Scala: objektorientierte und funktionale Sprache

Scala: Objektorientierte Sprache (wie Java, in kompakterer Notation) ergänzt um funktionale Konstrukte (wie in SML); objektorientiertes Ausführungsmodell (Java)

funktionale Konstrukte:

- geschachtelte Funktionen, Funktionen höherer Ordnung, Currying, Fallunterscheidung durch Pattern Matching
- Funktionen über Listen, Ströme, ..., in der umfangreichen Sprachbibliothek
- parametrische Polymorphie, eingeschränkte, lokale Typinferenz

objektorientierte Konstrukte:

- Klassen definieren alle Typen (Typen konsequent oo - auch Grundtypen), Subtyping, beschränkbare Typparameter, Case-Klassen zur Fallunterscheidung
- objektorientierte Mixins (Traits)

Allgemeines:

- statische Typisierung, parametrische Polymorphie und Subtyping-Polymorphie
- sehr kompakte funktionale Notation
- komplexe Sprache und recht komplexe Sprachbeschreibungen
- übersetzbar und ausführbar zusammen mit Java-Klassen
- seit 2003, Martin Odersky, www.scala.org

Übersetzung und Ausführung: Scala und Java

- **Reines Scala-Programm:**
  ein Programm bestehend aus einigen Dateien **`a.scala`**, **`b.scala`**, ... mit Klassen- oder Objekt-Deklarationen in Scala,
  eine davon hat eine **`main`**-Funktion;

  übersetzt mit **`scalac *.scala`**
  ausgeführt mit **`scala MainKlasse`**

```
// Klassendeklarationen

object MainKlasse {
// Funktionsdeklarationen

   def main(args: Array[String]) {

// Ein- und Ausgabe, Aufrufe
   }
}
```

- **Java- und Scala-Programm:**
  ein Programm bestehend aus Scala-Dateien **`a.scala`**, **`b.scala`**, ... und Java-Dateien **`j.java`**, **`k.java`**, ...;
  eine Java-Klasse hat eine **`main`**-Funktion;

  übersetzt mit **`scalac *.scala *.java`**
  dann mit **`javac *.scala *.java`**
  (Pfad zur Bibliothek angeben)
  ausgeführt mit **`java MainKlasse`**

- **Reines Scala-Programm interaktiv:**
  (siehe Übungen)

Benutzung von Listen

Die abstrakte **Bibliotheksklasse `List[+A]`** definiert Konstruktoren und Funktionen über **homogene Listen**

```
val li1 = List(1,2,3,4,5)

val li2 = 2 :: 4 :: -1 :: Nil
```

Verfügbare Funktionen:
`head, tail, isEmpty, map, filter, forall, exist, range, foldLeft, foldRight, range, take, reverse, :::` (append)

zwei Formen für Aufrufe:

```
li1.map (x=>x*x)// qualifizierter Bezeichner map

li1 map (x=>x*x)// infix-Operator map
```

Funktionsdefinitionen mit Fallunterscheidung:

```
def isort(xs: List[Int]): List[Int] = xs match {
   case List() => List()
   case x :: xs1 => insert(x, isort(xs1))
}
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
   case List() => List(x)
   case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

Case-Klassen: Typkonstruktoren mit Pattern Matching

Klassen können **Parameter** haben. Sie sind Instanzvariable der Klasse und Parameter des Konstruktors.
Die **Konstruktoren von Case-Klassen** können zur **Fallunterscheidung** und zum **Binden der Werte** dieser Instanzvariablen verwendet werden. Objekte können ohne **new** gebildet werden; Methoden für strukturellen Vergleich (**==**) und **`toString`** werden erzeugt.

```
abstract class Person
case class King   () extends Person
case class Peer   (degr: String, terr: String, number: Int )
                     extends Person
case class Knight (name: String) extends Person
case class Peasant(name: String) extends Person

val guestList =
   Peer ("Earl", "Carlisle", 7) :: Kling () ::
   Knight ("Gawain") :: Peasant ("Jack Cade") :: Nil

def title (p: Person): String = p match {
   case King () => "His Majesty the King"
   case Peer (d, t, n) => "The " + d + " of " + t
   case Knight (n) => "Sir " + n
   case Peasant(n) => n }

println ( guestList map title )
```

```
List(His Majesty the King, The Earl of Carlisle, Sir Gawain, Jack Cade)
```


Definition polymorpher Typen

Polymorphe Typen werden durch **Klassen mit Typparameter** definiert, z.B. Container-Klassen.
Alternative Konstruktoren werden durch **Case-Klassen** formuliert, z.B Binärbäume.

```
abstract class BinTree[A]
case class Lf[A] ()  extends BinTree[A]
case class Br[A] (v: A, left: BinTree[A], right: BinTree[A])
                     extends BinTree[A]
```

Funktionen über Binärbäume:

```
def preorder[A] (p: BinTree[A]): List[A] = p match {
   case Lf() => Nil
   case Br(v,tl,tr) => v :: preorder (tl) ::: preorder (tr)
}

val tr: BinTree[Int] =
     Br (2, Br (1, Lf(), Lf()), Br (3, Lf(), Lf()))

println ( preorder (tr) )
```

Funktionen höherer Ordnung und Lambda-Ausdrücke

Ausdrucksmöglichkeiten in Scala entsprechen etwa denen in SML, aber
die **Typinferenz polymorpher Signaturen** benötigt an vielen Stellen **explizite Typangaben**

Funktion höherer Ordnung: Faltung für Binärbäume

```
def treeFold[A,B] (f: (A, B, B)=>B, e: B, t: BinTree[A]): B =
   t match {
      case Lf () => e
      case Br (u,tl,tr) =>
         f (u, treeFold (f, e, tl), treeFold (f, e, tr))
}
```

Lambda-Ausdrücke:

`l1.map ( x=>x*x )`	Quadrat-Funktion
`l3.map ( _ => 5 )`	konstante Funktion
`l2.map ( Math.sin _ )`	Sinus-Funktion
`l4.map ( _ % 2 == 0 )`	Modulo-Funktion

```
treefold ( ((_: Int, c1: Int, c2: Int) => 1 + c1 + c2) , 0, t)
```


Currying

Funktionen in **Curry-Form** werden durch mehrere **aufeinanderfolgende Parameterlisten** definiert:

```
def secl[A,B,C] (x: A) (f: (A, B) => C) (y: B) = f (x, y);

def secr[A,B,C] (f: (A, B) => C) (y: B) (x: A) = f (x, y);

def power (x: Int, k: Int): Int =
   if (k == 1) x else
   if (k%2 == 0) power (x*x, k/2) else
                 x * power (x*x, k/2);
```

Im Aufruf einer Curry-Funktion müssen weggelassene Parameter durch _ angegeben werden:

```
def twoPow = secl (2) (power) _ ;
```
Funktion, die 2er-Potenzen berechnet

```
def pow3 = secr (power) (3) _ ;
```
Funktion, die Kubik-Zahlen berechnet

```
println ( twoPow (6) )
println ( pow3 (5) )
println ( secl (2) (power) (3) )
```

Ströme in Scala

In Scala werden **Ströme** in der Klasse **`Stream[A]`** definiert.
Besonderheit: Der **zweite Parameter der cons-Funktion** ist als **`lazy`** definiert, d.h. ein aktueller **Parameterausdruck** dazu wird erst ausgewertet, wenn er benutzt wird, d.h. der Parameterausdruck wird in eine **parameterlose Funktion** umgewandelt und so übergeben.
Diese Technik kann allgemein für Scala-Parameter angewandt werden.

```
def iterates[A] (f: A => A) (x: A): Stream[A] =
   Stream.cons(x, iterates (f) (f (x)))

def smap[A] (sq: Stream[A]) (f: A => A): Stream[A] =
   Stream.cons(f (sq.head), smap[A] (sq.tail) (f) )


val from = iterates[Int] (_ + 1) _

val sq = from (1)

val even = sq filter (_ % 2 == 0)

val ssq = from (7)

val msq = smap (ssq) (x=>x*x)


println( msq.take(10).mkString(",") )
```


Objektorientierte Mixins

Mixin ist ein Konzept in objektorientierten Sprachen: Kleine Einheiten von implementierter Funktionalität können Klassen zugeordnet werden (spezielle Form der Vererbung).
Sie definieren nicht selbst einen Typ und liegen neben der Klassenhierarchie.

```
abstract class Bird { protected val name: String }
```

Verschiedene **Verhaltensweisen** werden hier als **`trait`** definiert:

```
trait Flying extends Bird {
   protected val flyMessage: String
   def fly() = println(flyMessage)
}
trait Swimming extends Bird {
   def swim() = println(name+" is swimming")
}
```

```
class Frigatebird extends Bird with Flying {
   val name = "Frigatebird"
   val flyMessage = name + " is a great flyer"
}
class Hawk extends Bird with Flying with Swimming {
   val name = "Hawk"
   val flyMessage = name + " is flying around"
}

val hawk = (new Hawk).fly(); hawk.swim(); (new Frigatebird).fly();
```

Beispiele für Anwendungen funktionaler Sprachen

aus Peter Thiemann: *Grundlagen der Funktionalen Programmierung*, Teubner, 1994

- Programmierausbildung für Anfänger (z. B. Scheme, Gofer)
- Computeralgebrasysteme wie MACSYMA in Lisp implementiert
- Editor EMACS in Lisp implementiert
- Beweissysteme ISABELLE, LCF, Termesetzung REVE in ML implementiert
- Übersetzer und Interpretierer: SML, Lazy-ML, Glasgow Haskell C. in ML implementiert, Yale Haskell C. in Lisp implementiert
- Firma Ericsson eigene funktionale Sprache Erlang für Software im Echtzeiteinsatz, Telekommunikation, Netzwerkmonitore, grafische Bedienoberflächen

aus J. Launchbury, E. Meijer, Tim Sheard (Eds.): Advanced Functional Programming, Springer, 1996:

- Haggis: System zur Entwicklung grafischer Bedienoberflächen (S. Finne, S. Peyton Jones)
- Haskore Music Tutorial (Paul Hudak)
- Implementing Threads in Standard ML
- Deterministic, Error-Correcting Combinator Parsers (S. D. Swierstra, L. Duponcheel)

Verständnisfragen (1)

1. Einführung

1. Charakterisieren Sie funktionale gegenüber imperativen Sprachen; was bedeutet applikativ?

2. Lisp: FP Grundlagen

2. Charakteristische Eigenschaften von Lisp und seine Grundfunktionen.
3. Programm und Daten in Lisp; Bedeutung der **`quote`**-Funktion.
4. Funktion definieren und aufrufen
5. Dynamische Bindung im Gegensatz zu statischer Bindung.
6. Erklären Sie den Begriff Closure; Zusammenhang zum Laufzeitkeller.

3. Grundlagen von SML

7. Typinferenz: Aufgabe und Verfahren am Beispiel, mit polymorphen Typen.
8. Aufrufsemantik erklärt durch Substitution; call-by-value, call-by-name, call-by-need.
9. Muster zur Fallunterscheidung: Notation, Auswertung; Vergleich mit Prolog.
10. Bindungsregeln in SML (val, and, let, local, abstype, struct).

Verständnisfragen (2)

4. Programmierparadigmen zu Listen

11. Anwendungen für Listen von Paaren, Listen von Listen; Funktionen **`zip`** und **`unzip`**.
12. Matrizen transponieren, verknüpfen; applikativ und funktional.
13. Lösungsraumsuche für Münzwechsel: Signatur erläutern; Listen und Ströme.
14. Polynom-Multiplikation: Darstellungen, Halbierungsverfahren.

5. Module Typen

15. **`datatype`**-Definitionen: vielfältige Ausdruckmöglichkeiten.
16. Gekapselte Typen (**`abstype`**) erläutern.
17. Ausnahmen: 3 Sprachkonstrukte; Einbettung in funktionale Sprache.
18. Modul-Varianten (**`structure`**), Schnittstellen.
19. Generische Module (**`functor`**) erläutern.

Verständnisfragen (3)

6. Funktionen als Daten

20. Wo kommen Funktionen als Daten vor? Beispiele angeben.
21. Currying: Prinzip und Anwendungen
22. Funktionale **`secl`**, **`secr`**: Definition, Signatur und Anwendungen
23. Weitere allgemeine Funktionale (**`o`**, **`iterate`**, **`S K I`**)
24. Funktionale für Listen: **`map`** (1-, 2-stufig), **`filter`**, **`take`**, **`drop`** (-**`while`**)
25. Quantoren: Definition, Anwendung z.B. für disjunkte Listen
26. **`foldl`**, **`foldr`**, **`treefold`** erläutern

7. Unendliche Listen (Ströme)

27. Ströme: Konzept, Implementierung, Anwendungen
28. **`datatype`** für Ströme und Varianten dazu
29. Stromfunktionen, Stromfunktionale
30. Beispiel: Konvergente Folge
31. Ströme rekursiv zusammengesetzt (Sieb des Eratosthenes)
32. Strom aller Lösungen im Lösungsbaum (Signatur der Funktion)
33. Tiefensuche - Breitensuche im Lösungsbaum, 3 Abstraktionen

Verständnisfragen (4)

8. Lazy Evaluation

34. Paradigma lazy: Bedeutung in Sprachkonstrukten, im Vergleich zu eager
35. Lazy Semantik in Haskell, Beispiele für Aufrufe, Listen, Funktionen
36. Listen als Ströme; Vergleich zu Programmierung in SML; Fibonacci als Daten
37. Beispiel Hamming-Folge

9. Funktionale Sprachen

38. Eigenschaften von Lisp zusammenfassen
39. Eigenschaften von SML zusammenfassen
40. Eigenschaften von Haskell zusammenfassen