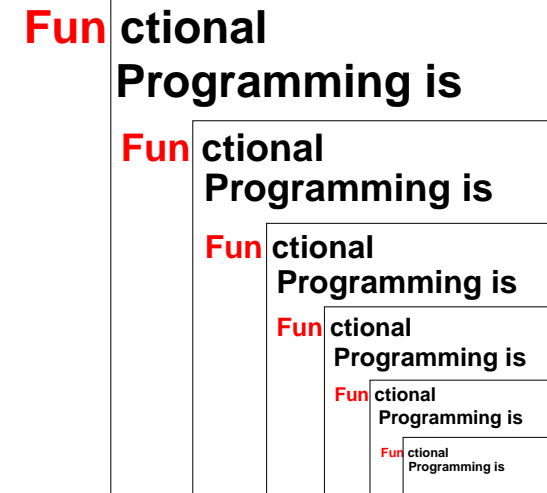


# Funktionale Programmierung

Prof. Dr. Uwe Kastens

SS 2013

## Functional Programming is Fun



## Ziele, Material

Die Teilnehmer sollen

- die **Klarheit und Mächtigkeit** der funktionalen Programmierung erkennen,
- **Paradigmen** der funktionalen Programmierung erlernen,
- **Techniken** der funktionalen Programmierung an praktischen Beispielen erproben und einüben

### Literatur:

- **Vorlesungsmaterial:**  
<http://ag-kastens.upb.de/lehre/material/fp>
- **Textbuch:**  
**L. C. Paulson: ML for the Working Programmer**, 2nd Edition, Cambridge University Press, 1996  
Schwerpunkt stärker auf Paradigmen und Techniken als auf der Sprache SML, enthält viele Beispiele bis hin zu nützlichen Modulen.
- Weiteres Buch zu SML:  
C. Myers, C. Clack, E.Poon: *Programming with Standard ML*, Prentice Hall, 1993
- siehe auch Internet-Links im Vorlesungsmaterial  
<http://ag-kastens.upb.de/lehre/material/fp/wwwrefs.html>

## Inhalt

	Kapitel im Textbuch
1. Einführung	
2. LISP: FP Grundlagen	
3. Grundlagen von SML	2
4. Programmierparadigmen zu Listen	3
5. Typkonstruktoren, Module	2, 4, 7
6. Funktionen als Daten	5
7. Datenströme	5
8. Lazy Evaluation	
9. Funktionale Sprachen: Haskell, Scala	
10. Zusammenfassung	

## Vorkenntnisse in FP aus Modellierung, GPS und PLaC

### Modellierung:

- Modellierung mit Wertebereichen

### GPS:

- Datentypen, parametrisierte Typen (Polytypen)

### • Funktionale Programmierung:

#### SML-Eigenschaften:

- Notation
- Deklarationen, Muster
- Funktionen, Aufrufe
- Listen
- Datentypen, Polymorphie

#### Programmiertechniken:

- Rekursion zur Induktion
- Rekursion über Listen
- akkumulierender Parameter
- Berechnungsschemata (HOF)
- Currying (HOF)

### PLaC:

- Spracheigenschaften und ihre Implementierung
- Typsysteme, Typanalyse für funktionale Sprachen

## Gedankenexperiment

Wähle eine **imperative Sprache**, z. B. Pascal, C.

Erlaube Eingabe nur als Parameter und Ausgabe nur als Ergebnis des Programmaufrufes.

**Eliminiere Variablen mit Zuweisungen.**

**Eliminiere** schrittweise alle **Sprachkonstrukte**, die **nicht mehr sinnvoll anwendbar** sind.

eliminiertes Sprachkonstrukt	Begründung
...	...
...	...

Betrachte die **restliche Programmiersprache**.  
Ist sie **sinnvoll anwendbar**?

**Erweitere sie um nützliche Konstrukte** (nicht Variablen mit Zuweisungen)

ergänzt Sprachkonstrukt	Begründung
...	...
...	...

## 2. LISP: FP Grundlagen

**Älteste** funktionale Programmiersprache (McCarthy, 1960).

Viele weit verbreitete **Weiterentwicklungen** (Common LISP, Scheme).

Ursprünglich wichtigste Sprache für Anwendungen im Bereich **Künstlicher Intelligenz**.

Hier nur **Pure LISP**:

### Notation:

geklammerte Folge von Symbolen  
gleiche Notation für Programm und Daten

### Datenobjekte:

Listen: (1 2 3 4) (ggT 24 36)  
atomare Werte: T, F, NIL, Zahlen, Bezeichner

**Beispiel:** Fakultätsfunktion mit akkumulierendem Parameter (**defun** nicht in Pure Lisp)

```
(defun AFac (n a)
  (cond ((eq n 0) a)
        (T (AFac (- n 1) (* a n)))))
(defun Fac (n) (AFac n 1))
```

**Aufruf:** (Fac 4)

## Grundfunktionen

**Aufruf:** (e0 e1 ... en).

Ausdruck e0 liefert die aufzurufende Funktion, die übrigen liefern Werte der aktuellen Parameter.

### Grundfunktionen:

(cons e l)	Listenkonstruktor
(car l)	Listenkopf
(cdr l)	Listenrest
(null l)	leere Liste?
(eq a b)	Vergleich atomarer Werte
(equal l1 l2)	tiefer Vergleich von Listen
(atom e)	atomar?
(cond (p1 e1) ... (pn en))	<b>Fallunterscheidung; nicht-strikte Auswertung:</b> (p <sub>i</sub> e <sub>i</sub> ) von links nach rechts auswerten bis ein p <sub>j</sub> T liefert; Ergebnis ist dann e <sub>j</sub> . e wird nicht ausgewertet, sondern ist selbst das Ergebnis

(quote e)

(lambda (x1 ... xn) e)

nicht in Pure Lisp:

```
(setq x e)
(defun f (x1 ... xn) e)
```

Funktionskonstruktor mit formalen Parametern x1...xn und Funktionsrumpf e

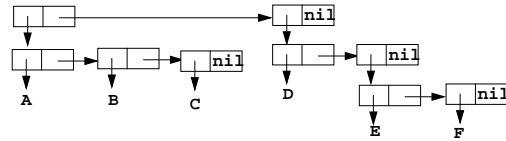
Wert von e wird an x gebunden (top level)

Funktion wird an f gebunden (top level)

## Listen als Programm und Daten

Eine **Liste** ist eine evtl. leere Folge von Ausdrücken;  
 Jeder Ausdruck ist ein Atom oder eine Liste:

```
'()
'((A B C) (D (E F)))
```



```
(cdr (cdr '(1 2 3 4)))
```

Ein **Aufruf** ist eine Liste; beim Auswerten liefert ihr erstes Element eine **Funktion**;  
 die weiteren Elemente liefern die **aktuellen Parameterwerte** des Aufrufes.

Ein **Programm** ist eine Liste von geschachtelten Aufrufen.

Das Programm operiert auf Daten; sie sind Atome oder Listen.

Die Funktion **quote** **unterdrückt die Auswertung** eines Ausdruckes; der Ausdruck selbst ist das Ergebnis der Auswertung:

Die Auswertung von `(quote (1 2 3 4))` liefert `(1 2 3 4)`  
 Kurznotation für `(quote (1 2 3 4))` ist `'(1 2 3 4)`

**Listen, die Daten in Programmen** angeben, müssen durch **quote** gegen Auswerten geschützt werden: `(length (quote 1 2 3))` oder `(length '(1 2 3))`

## Einige Funktionen über Listen

Länge einer Liste:

```
(defun Length (l) (cond ((null l) 0) (T (+ 1 (Length (cdr l))))))
```

Aufruf `(Length '(A B C))` liefert 3

Map-Funktion (`map` ist vordefiniert):

```
(defun Map (f l)
  (cond ((null l) nil)
        (T (cons (funcall f (car l))
                  (Map f (cdr l))))))
```

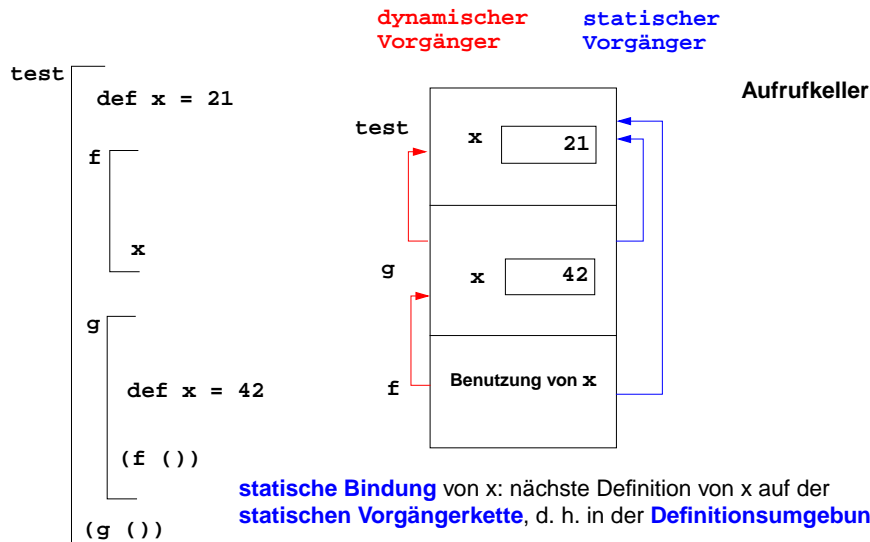
Aufruf `(Map (lambda (n) (cons n (cons n nil))) '(1 2 3))`

liefert `'((1 1) (2 2) (3 3))`

```
(funcall f p)
```

wertet `f` zu einer Funktion aus und ruft diese mit dem Parameter `p` auf

## Statische oder dynamische Bindung von Namen



**statische Bindung** von `x`: nächste Definition von `x` auf der **statischen Vorgängerkette**, d. h. in der **Definitionsumgebung**

**dynamische Bindung** von `x`: nächste Definition von `x` auf der **dynamischen Vorgängerkette**, d. h. in der **Aufrufumgebung**

## Statische oder dynamische Bindung in Common Lisp?

Bindungen in geschachtelten Lambda-Ausdrücken:

```
(print
  ((lambda (x)
    ((lambda (f)
      ((lambda (x)
        (funcall f 1)
      )
      42
    )
    (lambda (n) x)
  )
  21
)
```

Ergebnis bei statischer oder bei dynamischer Bindung?

## Bindungen und Closures

**Definition (freie Variable):** Wenn ein Name  $x$  innerhalb einer Funktion  $f$  nicht durch eine Parameterdefinition oder eine lokale Definition **gebunden** ist, dann bezeichnet  $x$  eine **freie Variable bezüglich der Funktion  $f$** .

Beispiele:

```
fun f (a, b) = a*x+b
fn (a, b) => a*x+b

(defun f (a b) (+ (* a x) b))
(lambda (a b) (+ (* a x) b))
```

Beim **Aufruf** einer Funktion  $f$  werden ihre freien Variablen je nach statischer oder dynamischer Bindung in der Definitions- oder Aufrufumgebung gebunden.

**Funktionen** können **als Daten** verwendet werden: Funktionen als Parameter, Ergebnis, Komponente von zusammengesetzten Werten, Wert von Variablen (imperativ). Für den Aufruf benötigen sie eine Closure:

Die **Closure** einer Funktion  $f$  ist eine **Menge von Bindungen**, in der beim Aufruf von  $f$  die **freien Variablen von  $f$  gebunden** werden.

**Dynamische Bindung:** Closure liegt im Aufrufkeller.

**Statische Bindung:** Closure ist in der Kette der **statischen Vorgänger** enthalten; diese müssen ggf. auf der Halde statt im Laufzeitkeller gespeichert werden, da Schachteln (und deren Variablen) noch benötigt werden, wenn ihr Aufruf beendet ist

## 3. Grundlagen von SML 3.1 Ausdrücke und Aufrufe

**Grundkonzepte funktionaler Sprachen:**

**Funktionen und Aufrufe, Ausdrücke**

**keine Variablen** mit Zuweisungen, keine **Ablaufstrukturen**, keine **Seiteneffekte** (im Prinzip: aber E/A, Interaktion, Abbruch etc.)

Funktionale Sprachen sind **ausdrucksorientiert** (statt anweisungsorientiert): Programme bestehen aus Definitionen und Ausdrücken (statt Anweisungen). Typisch: bedingter Ausdruck statt bedingter Anweisung.

```
if a>b then a-b else b-a
```

Die Auswertung jedes Programmkonstruktes liefert einen Wert (statt einen Effekt zu erzeugen, d.h. den Programmzustand zu ändern).

## Aufruf-Semantik Call-by-value (strikt)

Auswertung von Funktionsaufrufen (`mul (2, 4)`) und von Ausdrücken mit Operatoren (`2 * 4`) sind semantisch gleichwertig.

In SML haben alle Funktionen genau einen Parameter, ggf. ein Tupel.

**Aufruf:** (Funktionsausdruck Parameterausdruck)

**Auswertung nach call-by-value, strikte Auswertung:**

- Funktionsausdruck auswerten und Closure bestimmen;** Ergebnis ist eine Funktion mit einer Closure, in der die freien Variablen der Funktion gebunden werden.
- Parameterausdruck auswerten;** Ergebnis an den formalen Parameter der Funktion binden.
- Funktionsrumpf** mit Bindungen des formalen Parameters und der Closure **auswerten;** Ergebnis ist das Ergebnis der Ausdrucksauswertung.

**Beispiel:**

```
fun sqr x : int = x * x;
fun zero (x : int) = 0;
```

Auswertung modelliert durch **Substitution von innen nach außen:**

```
sqr (sqr (sqr 2)) => sqr (sqr (2 * 2)) => ...
zero (sqr (sqr (sqr 2))) => ...
```

**Bedingte Ausdrücke werden nicht strikt ausgewertet!**

## Aufruf-Semantik Call-by-need - lazy evaluation

**Aufruf:** (Funktionsausdruck Parameterausdruck)

**Auswertung nach call-by-name:**

1. und 3. wie oben

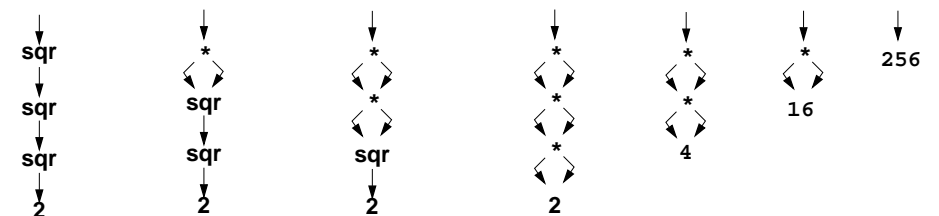
2. **Parameterausdruck** an jedes Auftreten des formalen Parameters im Funktionsrumpf substituieren (nach konsistenter Umbenennung der Namen im Parameterausdruck).

**Beispiel:** Auswertung modelliert durch **Substitution von außen nach innen:**

```
sqr (sqr (sqr 2)) => (sqr (sqr 2)) * (sqr (sqr 2)) => ...
zero (sqr (sqr (sqr 2))) => 0
```

\* wird als Elementaroperator strikt ausgewertet.

**Auswertung nach call-by-need (lazy evaluation):** wie call-by-name, aber der aktuelle Parameter wird **höchstens einmal ausgewertet** und sein Wert ggf. wiederverwendet. modelliert durch **Graph-Substitution von außen nach innen:**



## 3.2 Deklarationen in SML, Muster

Grundform von Deklarationen:

```
val Muster = Ausdruck
```

Der Ausdruck wird ausgewertet und liefert einen Wert *w*.

Das Muster ist hierarchisch aufgebaut aus

- **Bezeichnern**, die gebunden werden sollen; derselbe Bezeichner darf nicht mehrfach in einem Muster vorkommen;
- **Konstruktoren** für Datentypen, z. B. Tupel ( , ), Listen :: oder durch `datatype` eingeführte Konstruktoren, Zahlen;
- `_` anstelle eines Bezeichners (es wird nicht gebunden).

Der Wert *w* wird gemäß der Struktur des Musters zerlegt. Die Teilwerte werden an die entsprechenden Bezeichner gebunden.

```
fun foo x = (x, x);

val x = sqr 3;          val (a, b) = (sqr 2, sqr 3);
val (c, d) = foo 42;    val (x,y)::z = [foo 41, (3,4), (5,6)];
val h::_ = [1, 2, 3];
```

## Funktionsdeklarationen

`val`-Deklaration einer rekursiven Funktion:

```
val rec Fac = fn n => if n <= 1 then 1 else n * Fac (n-1);
```

**Kurzform** für Funktionsdeklarationen:

```
fun Name Parametermuster = Ausdruck;

fun Fac n = if n <= 1 then 1 else n * Fac (n-1);
```

Funktionsdeklaration mit Fallunterscheidung über Muster:

```
fun FName Muster1 = Ausdruck1
  | FName Muster2 = Ausdruck2
  ...;
```

Die Muster werden nacheinander auf den Parameter angewandt, bis das erste trifft.

```
fun app (nil, lr) = lr
  | app (ll, nil) = ll
  | app (h::t, r) = h :: (app (t, r));
```

statt mit bedingten Ausdrücken über den Parameter:

```
fun app (ll, lr) = if ll = nil then lr else
                  if lr = nil then ll else
                  (hd ll) :: (app (tl ll, lr));
```

## Statische Bindung in SML

Auswerten einer `val`-Deklaration erzeugt eine **Menge von Bindungen** *Bezeichner -> Wert*, je eine für jeden Bezeichner im Muster.

In einer **Gruppe von Deklarationen**, die mit `and` verknüpft sind, gelten **alle Bindungen** der Gruppe **in allen Ausdrücken** der Gruppe (Algol-Verdeckungsregel)

```
fun f x = if p x then x else g x and
  g x = if q x then x else f x;
```

In **einzelnen Deklarationen**, die durch `;` getrennt werden, gelten die Definitionen **erst nach dem Ausdruck** der Deklaration.

Ausnahme: `val rec Muster = Ausdruck`; Bindungen gelten schon im Ausdruck.

Jede **einzelne Deklaration** oder Deklarationsgruppe bildet einen einzelnen **Abschnitt** im Sinne der Verdeckungsregeln: **Gleichbenannte Deklarationen verdecken Bindungen** des umfassenden (vorangehenden) Konstruktes:

```
val a = 42;
val b = 2 * a;
val a = 3;
val c = a + 1;
a + b * c;
```

`let`-Konstrukt fasst Deklarationen mit dem Ausdruck zusammen, in dem ihre Bindungen gelten:

```
let D1; D2; ... in Ausdruck end
```

`local`-Konstrukt fasst Deklarationen mit der Deklaration zusammen, in der ihre Bindungen gelten:

```
local D1; D2; ... in Deklaration end
```

## 3.3 Typen, Grundtypen

`int` und `real`:

`real`-Literale: `1.2E3` `7E-5`  
 binäre Operatoren: `+` `-` `*` `/`  
 unäres Minus: `~`  
 sind **überladen** für `int` und `real`.  
 Deshalb sind Typangaben nötig,  
 wenn der Typ der Operanden nicht eindeutig ist:

```
fun sqr (x : real) = x * x;
```

Funktionsbibliotheken `Int`, `Real`, `Math`:

```
Int.min (7, Int.abs i);
Math.sin (r) / r;
```

`bool`:

Literale: `true` `false`  
 Operatoren: `orelse` `andalso` `not`  
**nicht strikt**, d. h. Kurzauswertung (wie in C)  
 Vergleichsoperatoren: `=`, `<>`, `<`, `>`, `>=`, `<=`

`string`:

Literale wie in C: `"Hello World!\n"`  
 Konkatenationsoperator: `^`  
 Funktionsbibliothek `String`

`char`:

Literale: `#"a"` `#"\n"`

## Tupel, Records

FP-3.8

### Tupel:

```
val zerovec = (0.0, 0.0); val today = (5, "Mai", 2010);
```

Funktion mit Tupel als Parameter:

```
fun average (x, y) = (x+y)/2.0; average (3.1, 3.3);
```

### Typdefinitionen:

```
type Vec2 = real * real;  
fun trans ((a,b):Vec2, x):Vec2 = (a+x, b+x);  
trans (zerovec, 3.0);
```

### Records - Tupel mit Selektornamen:

```
type Date = {day:int, month:string, year:int};  
val today = {year=2010, month="Mai", day=5}:Date;  
fun add1year {day=d, month=m, year=y} =  
  {day=d, month=m, year=(y+1)};
```

Benutzung von Selektorfunktionen:

```
#day today;
```

unvollständiges Record-Pattern:

```
fun thisyear ({year,...}:Date) = year = 1997;
```

© 2010 bei Prof. Dr. Uwe Kastens

## Parametrisierte Typen (GdP-5.9)

FP-3.9

### Parametrisierte Typen (Polytypen, polymorphe Typen):

Typangaben mit **formalen Parametern**, die für Typen stehen.

Man erhält aus einem Polytyp einen konkreten Typ durch **konsistentes Einsetzen eines beliebigen Typs** für jeden Typparameter.

Ein Polytyp beschreibt die **Typabstraktion**, die allen daraus erzeugbaren konkreten Typen gemeinsam ist.

**Beispiele** in SML-Notation mit 'a', 'b, ... für Typparameter:

Polytyp	gemeinsame Eigenschaften	konkrete Typen dazu
'a × 'b	Paar mit Komponenten <b>beliebigen</b> Typs	int × real int × int
'a × 'a	Paar mit Komponenten <b>gleichen</b> Typs	int × int (int->real) × (int->real)

rekursiv definierte Polytypen:

'a list = 'a × 'a list   {nil}	int list real list (int × int) list
homogene, lineare Listen	

Verwendung z. B. in **Typabstraktionen** und in **polymorphen Funktionen**

© 2013 bei Prof. Dr. Uwe Kastens

## Polymorphe Funktionen (GdP-5.9a)

FP-3.10

### (Parametrisch) polymorphe Funktion:

eine Funktion, deren **Signatur ein Polytyp** ist, d. h. Typparameter enthält.

Die Funktion ist auf Werte eines jeden konkreten Typs zu der Signatur anwendbar. D. h. sie muss unabhängig von den einzusetzenden Typen sein;

### Beispiele:

Eine Funktion, die die Länge einer beliebigen homogenen Liste bestimmt:

```
fun length l = if null l then 0 else 1 + length (tl l);
```

polymorphe Signatur: 'a list -> int

Aufrufe: length ([1, 2, 3]); length ([1, true], [2, true]);

Funktionen mit Paaren:

```
fun pairself x = (x, x);
```

```
fun car (x, _) = x;
```

```
fun cdar (_, (x, _)) = x;
```

```
fun id x = x;
```

© 2010 bei Prof. Dr. Uwe Kastens

## Typinferenz

FP-3.11

SML ist **statisch typisiert**. **Typangaben** sind meist **optional**.

### Typinferenz:

Der **Typ T** eines Programmobjektes (benannt in Deklaration) oder eines Programmkonstruktes (unbenannter Ausdruck) wird aus dem Programmtext statisch ermittelt und geprüft.

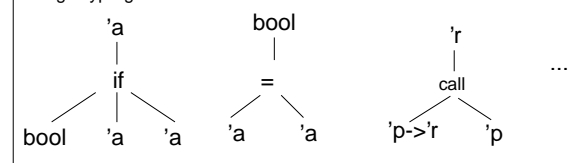
**T** ist der **allgemeinste Typ** (hinsichtlich der Typparameter), der mit den Operationen in der Deklaration bzw. in dem Ausdruck konsistent ist.

### Verfahren:

**Gleichungssystem** mit **Typvariablen** vollständig aufstellen:

- Typ von Literalen ist bekannt.
- Typ von gebundenen Namen ist bekannt.
- Für hier definierte Namen n (in Mustern) Typ(n) einsetzen
- Typregeln für jedes Programmkonstrukt auf Programmbaum systematisch anwenden, liefert **alle** Gleichungen.

einige Typregeln:

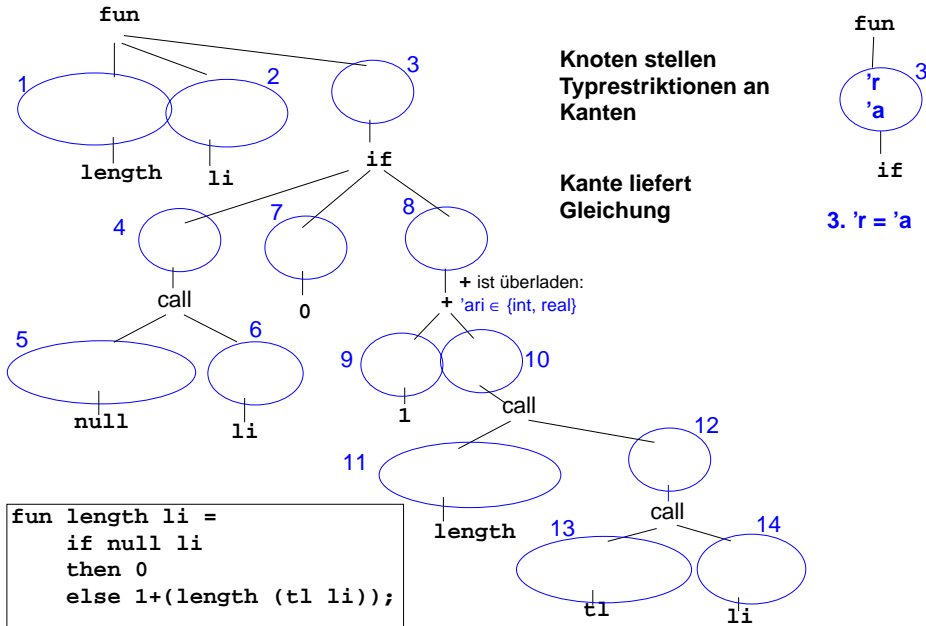


### Gleichungssystem lösen:

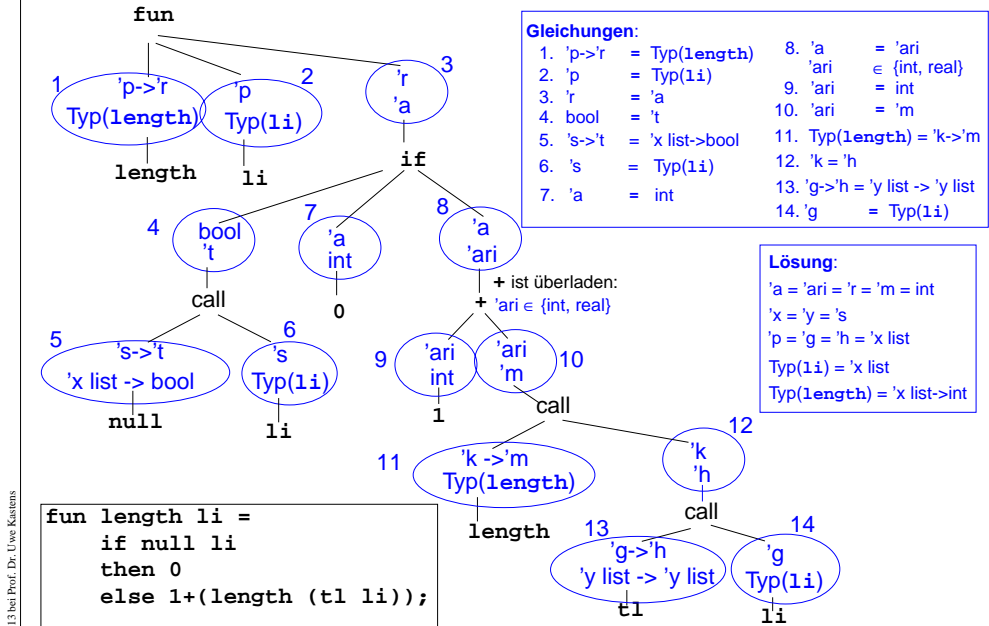
- Widersprüche -> Typfehler
- Alle Typvariablen gebunden -> Typen der definierten Namen gefunden
- Einige Typvariablen bleiben offen -> der Typ ist **polymorph**

© 2010 bei Prof. Dr. Uwe Kastens

### Beispiel zur Typinferenz, Strukturbaum



### Beispiel zur Typinferenz



### 4. Programmierparadigmen zu Listen; Grundlagen

Listen in SML sind **homogen**, die Elemente einer Liste haben denselben Typ. Vordefinierter Typ:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Konstruktoren:

```

nil           = []
x :: xs       :: ist rechtsassoziativ: 3 :: 4 :: 5 :: nil = [3, 4, 5]
    
```

Vordefinierte Funktionen:

```

length l      Anzahl der Elemente in l
hd l          erstes Element von l
tl           1 ohne das erste Element
null l        ist l = nil?
rev l         l in umgekehrter Reihenfolge
l1 @ l2       Konkatenation von l1 und l2
    
```

Beispiel:

```
fun upto (m, n) = if m > n then [] else m :: upto (m+1, n);
```

### Rekursionsmuster Listentyp

Struktur des Datentyps:

```
datatype 'a list = nil | :: of ('a * 'a list)
```

Paradigma: Funktionen haben die **gleiche Rekursionsstruktur wie der Datentyp**:

```

fun F (nil) = nicht-rekursiver Ausdruck
  | F (h::t) = Ausdruck über h und F t
    
```

```

fun prod nil = 1
  | prod (h::t) = h * prod t;
    
```

Varianten:

```

fun member (nil, m) = false
  | member (h::t, m) = if h = m then true else member (t, m);
    
```

```

fun append (nil, r) = r
  | append (l, nil) = l
  | append (h::t, r) = h :: append (t, r);
    
```

Abweichung: Alternative 1- oder mehrelementige Liste; (Patternliste ist nicht vollständig!)

```

fun maxl [m] = m
  | maxl (m::n::ns) = if m > n then maxl (m::ns) else maxl (n::ns);
    
```

## Akkumulierender Parameter für Funktionen auf Listen

### Akkumulierender Parameter

- führt das bisher berechnete **Zwischenergebnis** mit,
- macht die Berechnung **end-rekursiv**,
- wird mit dem **neutralen Element der Berechnung initialisiert**,
- verknüpft die Listenelemente von **vorne nach hinten**.

```
fun zlength nil = 0
| zlength (_::t) = 1 + zlength t;

fun alength (nil, a) = a
| alength (_::t, a) = alength (t, a+1);
```

Beispiel: Nimm die ersten i Elemente einer Liste:

```
fun atake (nil, _, taken) = taken
| atake (h::t, i, taken) = if i>0 then atake (t, i-1, h::taken)
else taken;
```

Die Partner-Funktion drop ist schon end-rekursiv:

```
fun drop (nil, _) = nil
| drop (h::t, i) = if i>0 then drop (t, i-1) else h::t;
```

## Listen aus Listen und Paaren

### Liste von Listen konkatenieren:

```
Signatur: concat: 'a list list -> 'a list
fun concat nil      = nil
| concat (x :: xs) = x @ concat xs;
```

**Aufwand:** Anzahl :: = Gesamtzahl der Elemente; Rekursionstiefe = Anzahl der Teillisten

**Listen von Paaren herstellen:** 2-stellige Relation, Zuordnung  
überzählige Elemente werden weggelassen. Reihenfolge der Muster ist relevant!

```
Signatur: 'a list * 'b list -> ('a * 'b) list
fun zip (x::xs,y::ys) = (x,y) :: zip (xs,ys)
| zip _                = nil;
```

### Paar-Liste auflösen:

```
Signatur: ('a * 'b) list -> 'a list * 'b list
fun unzip nil          = (nil, nil)
| unzip ((x, y) :: pairs) =
  let val (xs, ys) = unzip pairs in (x :: xs, y :: ys) end;
```

end-rekursiv, Ergebnis in umgekehrter Reihenfolge, mit akkumulierenden Parametern **xs, ys**:

```
local fun revUnzip (nil, xs, ys) = (xs, ys)
| revUnzip ((x, y):: pairs, xs, ys) =
  revUnzip (pairs, x::xs, y::ys);
in fun iUnzip z = revUnzip (z, nil, nil) end;
```

## Liste aller Lösungen am Beispiel: Münzwechsel (1)

geg.: Liste verfügbarer Münzwerte und auszuzahlender Betrag  
ges.: Liste von Münzwerten, die den Betrag genau auszahlt

zur Einstimmung:

Greedy-Verfahren mit genau einer Lösung. Es gelte  
(\*) Liste der verfügbaren Münzwerte ist fallend sortiert. Der kleinste Wert ist 1.  
Garantiert Terminierung.

```
fun change (coinvals, 0)          = []
| change (c :: coinvals, amount) =
  if amount < c then change (coinvals, amount)
  else c :: change (c :: coinvals, amount - c);
```

einige Münzsysteme:

```
val euro_coins = [200, 100, 50, 20, 10, 5, 2, 1];
val gb_coins  = [50, 20, 10, 5, 2, 1];
val dm_coins  = [500, 200, 100, 50, 10, 5, 2, 1];
```

Aufrufe mit Ergebnissen:

```
- change (euro_coins, 489);
> val it = [200, 200, 50, 20, 10, 5, 2, 2] : int list
- change (dm_coins, 489);
> val it = [200, 200, 50, 10, 10, 10, 5, 2, 2] : int list
```

## Liste aller Lösungen: Beispiel Münzwechsel (2)

Allgemeines Problem ohne Einschränkung (\*); alle Lösungen gesucht  
Entwurfstechnik: **Invariante über Parameter**

```
Signatur: int list * int list * int -> int list list
gezahlte   verfügbare  Rest-   Liste aller
Stücke     Münzwerte  betrag  Lösungen
```

invariant: Wert gezahlter Stücke + Restbetrag = Wert jeder Lösung.  
invariant: in gezahlten Stücken sind (t-1 verfügbare Münzwerte) nicht benutzt

### Fallunterscheidung für Funktion allChange:

```
Betrag ganz ausgezahlt                eine Lösung
coins _                                0      = [coins]
keine Münzwerte mehr verfügbar         keine Lösung
coins []                                _      = []
```

```
rekursiver Fall:
coins c::coinvals amount =
  if amount < 0
```

```
Betrag so nicht auszahlbar:
  then []
```

```
2 Möglichkeiten verfolgen: c benutzen oder c nicht benutzen
  else allChange (c::coins, c::coinvals, amount-c) @
    allChange (coins, coinvals, amount);
```



## Liste aller Lösungen: Beispiel Münzwechsel (3)

Funktion allChange:

```
fun allChange (coins, _, 0) = [coins]
| allChange (coins, [], _) = []
| allChange (coins, c::coinvals, amount) =
  if amount < 0 then []
  else allChange (c::coins, c::coinvals, amount-c) @
        allChange (coins, coinvals, amount);
```

Aufruf und Liste von Lösungen:

```
- allChange ([], euro_coins, 9);

> val it =
  [ [2, 2, 5], [1, 1, 2, 5], [1, 1, 1, 1, 5],
    [1, 2, 2, 2, 2], [1, 1, 1, 2, 2, 2], [1, 1, 1, 1, 1, 2, 2],
    [1, 1, 1, 1, 1, 1, 2],
    [1, 1, 1, 1, 1, 1, 1, 1]] : int list list

- allChange ([], [5,2], 3);
> val it = [] : int list list
```

## Matrix-Operationen mit Listen: Transponieren

```
fun headcol [] = []
| headcol ((x::_)::rows) = x :: headcol rows;      ( a | b c )
                                                    ( d | e f )

fun tailcols [] = []
| tailcols ((_::xs)::rows) = xs :: tailcols rows;

fun transp ([]::_) = []
| transp rows = headcol rows :: transp (tailcols rows); ( a d )
                                                         ( b e )
                                                         ( c f )
```

Die Fallunterscheidungen sind nicht vollständig (Warnung).  
Es wird angenommen, daß alle Zeilen gleich lang sind.

```
val letterMatr = [ ["a","b","c"], ["d","e","f"] ];
- transp letterMatr;

> val it = [ ["a", "d"], ["b", "e"], ["c", "f"] ] : string list list
```

## Matrix-Operationen mit Listen: Matrix-Multiplikation

**Aufgabe schrittweise zerlegen.** Reihenfolge der Funktionen dann umkehren:

```
fun matprod (rowsA, rowsB) =
  rowListprod (rowsA, transp rowsB);

fun rowlistprod ([], _) = []
| rowlistprod (row::rows, cols) =
  rowprod (row, cols) :: rowlistprod (rows, cols); ( — ):(| | | )

fun rowprod (_, []) = []
| rowprod (row, col::cols) =
  dotprod (row, col) :: rowprod (row, cols); ( — ):(| | | )

fun dotprod ([], []) = 0.0
| dotprod (x::xs, y::ys) = x*y + dotprod(xs, ys); — . |
```

**Aufruf und Ergebnis:**

```
val numMatr = [[1.0,2.0],[3.0,4.0]]; matprod (numMatr, numMatr);
> val it = [[7.0, 10.0], [15.0, 22.0]] : real list list
```

## Listenrepräsentation für Polynom-Arithmetik

Polynome in einer Variablen:  $a_n x^n + \dots + a_1 x^1 + a_0$

Datenrepräsentation: `real list: [an, ..., a1, a0]`

besser für dünn besetzte Koeffizientenlisten:

`(int * real) list: [(n, an), ..., (1, a1), (0, a0)]`

mit:  $a_i \neq 0$ , eindeutig in Potenzen und fallend sortiert

**Beispiel:**  $(x^4 - x + 3) + (x - 5) = (x^4 - 2)$

`sum ([ (4, 1.0), (1, ~1.0), (0, 3.0) ], [ (1, 1.0), (0, ~5.0) ])`  
liefert `[(4, 1.0), (0, ~2.0)]`

**Polynom-Summe:**

```
fun sum ([], us) = us
| sum (ts, []) = ts
| sum ((m, a)::ts, (n, b)::us) =
```

die höchsten Potenzen sind verschieden (2 Fälle):

```
if m > n then (m, a)::sum (ts, (n, b)::us)
else if m < n then (n, b)::sum (us, (m, a)::ts)
```

die höchsten Potenzen sind gleich und werden zusammengefasst:

```
else if a+b=0.0 then sum (ts, us)
else (m, a+b)::sum (ts, us);
```

## Polynom-Arithmetik - Halbierungsverfahren

### Polynom-Produkt:

termprod multipliziert ein Polynom mit  $a \cdot x^m$

```
fun termprod((m,a), []) = []
| termprod((m,a), (n,b)::ts) =
  (m+n, a*b)::termprod((m,a), ts);
```

Multiplikation zweier Polynome mit **Halbierungstechnik**:

```
fun prod([], us) = []
| prod([(m,a)], us) = termprod((m,a), us)
| prod(ts, us) =
  let val k = length ts div 2
  in sum (prod (List.take(ts,k), us),
         prod (List.drop(ts,k), us))
  end;
```

Ebenso mit Halbierungstechnik:

Polynom-Potenz, Polynom-GGT für Polynom-Division

```
- prod (p1, p2);
> val it = [(5, 1.0), (4, ~5.0), (2, ~1.0), (1, 8.0), (0, ~15.0)] :
  (int * real) list
```

## 5. Typen und Module

### Datentypen mit Konstruktoren

**Definition von Datentypen, die verschiedene Wertebereiche zu einem allgemeineren zusammenfassen.**

Beispiel: Modellierung britischer Adelspersonen

King	eindeutig, einmalig	
Peer	Rang, Territorium, Erbfolge	z. B. 7th Earl of Carlisle
Knight	Name	z. B. Galahad
Peasant	Name	z. B. Jack Cade

**Allgemeines Konzept:**

unterscheidbare Vereinigung von Wertebereichen (**discriminated union**).

Verschiedene Techniken in verschiedenen Sprachen:

- Oberklasse u. **spezielle Unterklassen** in objektorientierten Sprachen
- **Record mit Varianten** in Pascal, Modula, Ada
- **Vereinigungstyp** in Algol68
- **struct** mit Unterscheidungskomponente und **union** in C
- **datatype** in SML

## Discriminated Union mit Konstruktor-Funktionen

**Allgemeines Konzept: discriminated union;** In SML realisiert durch:

```
datatype person =
  King
| Peer of string * string * int
| Knight of string
| Peasant of string;
```

Definiert den Typ **person** mit seinen Konstruktoren:

```
King: person
Peer: string * string * int -> person
Knight: string -> person
Peasant: string -> person
```

Notation für Werte:

King, Peer ("Earl", "Carlisle", 7), Peasant ("Jack Cade")

Fallunterscheidung mit Konstruktoren in Mustern:

```
fun title King = "His Majesty the King"
| title (Peer (deg, terr, _)) = "The ^deg^ of ^terr"
| title (Knight name) = "Sir ^name"
| title (Peasant name) = name;
```

Jede **datatype**-Definition führt einen **neuen** Typ ein.

Vorsicht beim Verdecken durch Redefinieren!

## Verallgemeinerte Datentypkonstruktion

**Aufzählungstyp** als Vereinigung 1-elementiger Wertebereiche:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;
Hier sind alle Konstruktoren Konstante.
```

```
datatype order = LESS | EQUAL | GREATER;
verwendet in String.compare: string * string -> order
```

**Konstruktion polymorpher Typen:**

allgemeines Konzept „Fehlerwert“:

```
datatype 'a option = NONE | SOME of 'a;
z. B. in Real.fromString: string -> real option
```

allgemeines Konzept „Wertebereiche paarweise vereinigen“:

```
datatype ('a,'b) union = In1 of 'a | In2 of 'b;
z. B. [(In1 5), (In2 3.1), (In1 2)]
```

**rekursiv definierte polymorphe Typen:**

```
lineare Listen: datatype 'a list = nil | :: of ('a * 'a list);
binäre Bäume: datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;
allgemeine Bäume: datatype 'a mtree =
  Mleaf | MBranch of 'a * ('a mtree) list;
```

## Binäre Bäume

Typdefinition: `datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;`

ein Baum-Wert: `val t2 = Br (2, Br (1, Lf, Lf), Br (3, Lf, Lf));`

Rekursionsmuster: Fallunterscheidung und Rekursion wie in der Typdefinition

```
fun size Lf = 0
  | size (Br (v,t1,t2)) = 1 + size t1 + size t2;
fun preorder Lf = []
  | preorder (Br(v,t1,t2)) = [v] @ preorder t1 @ preorder t2;
```

mit akkumulierendem Parameter:

`preord` stellt der schon berechneten Liste den linearisierten Baum voran:

```
fun preord (Lf, vs) = vs
  | preord (Br(v,t1,t2), vs) = v :: preord (t1, preord (t2, vs));
```

inverse Funktion zu `preorder` baut **balancierten Baum** auf:

```
balpre: 'a list -> 'a tree
fun balpre nil = Lf
  | balpre (x :: xs) =
    let val k = length xs div 2
    in Br(x, balpre (List.take (xs, k)),
          balpre (List.drop (xs, k)))
    end;
```

## Gekapselte Typdefinition

`abstype` definiert neuen ggf. polymorphen Typ mit Operationen darauf.

Außen sind nur die Operationen **sichtbar**,  
nicht aber die **Konstruktorfunktionen** des Datentyps (Implementierung).

Beispiel Wörterbuch mit Binärbaum implementieren:

```
abstype 'a t =
  Leaf | Branch of key * 'a * 'a t * 'a t
with
  exception NotFound of key;
  val empty = Leaf;
  fun lookup (Branch(a,x,t1,t2), b) = ...
  fun insert (Leaf, b, y) = ...
    | insert (Branch(a,x,t1,t2), b, y) = ...
  fun update (Leaf, b, y) = raise NotFound b
    | update (Branch(a,x,t1,t2), b, y) = ...
end;
```

Anwendung:

```
val wb = insert (insert (empty, "Hund", "dog"), "Katze", "cat");
val w = lookup (wb, "Hund");
```

## Zusammenfassung von Typdefinitionen

SML-Konstrukt	Bedeutung	Vergleich mit anderen Sprachen
<code>datatype</code>	neuer Typ mit Konstruktoren	<code>type</code> in Pascal, Klassen in objektorientierten Sprachen
<code>type</code>	anderer Name für existierenden Typ	<code>typedef</code> , <code>#define</code> in C
<code>abstype</code>	neuer Typ mit Operationen aber verborgenen Konstruktoren	ADT mit verborgener Implementierung opaque in Modula, Ada
<b>Modul-Konstrukte:</b>		
<code>structure</code>	Modul mit neuem Namensraum	Record in Pascal, Modula-Modul
<code>signature</code>	beschreibt Schnittstelle bzw. Signaturen	Interface in Java, Modula, Header-Dateien in C,
<code>functor</code>	parametrisierte Struktur	Templates in C++, Generics in Ada, Java Interface als Parametertyp

## Ausnahmebehandlung (Exceptions)

**Motivation:**

**Partielle Funktion** `f total` machen ist umständlich:

- Ergebnistyp `'a` ersetzen durch `datatype 'a option = NONE | SOME of 'a;`
- Fallunterscheidung bei jedem Aufruf von `f`
- dasselbe für jede Funktion, die `f` aufruft;  
Fehlerwert „durchreichen“ bis er „behandelt“ wird.

**SML-Ausnahmen** propagieren Fehler von der Auslösestelle auf dem Berechnungsweg bis zur Behandlung - ohne Änderung von Typen und Funktionen.

**Deklaration:**

```
exception Failure;
exception Fail of string;
ergänzt vordefinierten Typ exn
um eine Variante
```

**auslösen durch Ausdruck**

```
raise Failure
raise (Fail "too small")
```

**Behandlung im Ausdruck**

```
(zu prüfender Ausdruck) handle Failure => E1
  | Fail (s) => E2
```

Das Ergebnis bestimmt der zu prüfende Ausdruck oder `E1` oder `E2`

## Beispiel zur Ausnahmebehandlung

### Beispiel Münzwechsel

Eine Lösung wird durch Backtracking im Lösungsraum gesucht.

Wenn ein Zweig keine Lösung liefern kann, wird eine Ausnahme ausgelöst: **raise Change**

Das Backtracking verweigert am zuletzt durchlaufenen **handle Change**

Gibt es keine Lösung, so bleibt die Ausnahme unbehandelt

**Uncaught exception Change**

```
exception Change;
```

```
fun backChange (coinvals, 0)      = []
  | backChange ([], amount)      = raise Change
  | backChange (c::coinvals, amount) =
    if amount < 0 then raise Change
    else c :: backChange(c::coinvals, amount - c)
      handle Change =>
        backChange(coinvals, amount);
```

## Module in SML

### Module und Sichtbarkeit von Typen, Varianten:

```
structure:
Implementierungsmodul
(vgl. Modula 2);
kapselt Folge von Deklarationen:
```

```
structure Seq =
  struct
    exception Empty;
    fun
      hd (Cons(x,xf)) = x
      | hd Nil = raise Empty;
    ...
  end;
```

Qualifizierte Namen wie `Seq.hd`  
benennen exportierte Größen.

1. Der Modul implementiert eine **Sammlung von Funktionen** zu einem Datentyp; der Datentyp wird außerhalb des Moduls definiert `datatype 'a seq = Nil | Cons of 'a ...`
2. Der Modul implementiert einen Datentyp mit Operationen darauf; die **Implementierung** (Konstruktorfunktionen) soll aussen **qualifiziert sichtbar** sein (`Seq.Cons`); der Datentyp wird innerhalb des Moduls definiert `structure Seq = struct datatype 'a t = Nil | Cons of ... .. exportierte Funktionen end;` Namenskonvention: `t` für den exportierten Typ.
3. wie (2) aber mit **verborgener Implementierung** des Datentyps; Konstruktorfunktionen sind nicht benutzbar): `structure Bar = struct abstype 'a t = Nil | Cons of ... with ... exportierte Funktionen end end;`

## Schnittstellen

**signature:**

### Schnittstelle von Modulen

definiert eine Folge von typisierten Namen (specifications),

die ein Modul mindestens implementieren muss, um die **signature** zu erfüllen;

**signature** ist eigenständiges, benanntes Programmobjekt (vgl. Java Interfaces):

```
signature QUEUE =
sig type 'a t
  exception E
  val empty: 'a t
  val enq: 'a t * 'a -> 'a t
  ...
end;
```

Mehrere Module können eine Schnittstelle unterschiedlich implementieren:

```
structure QueueStd: QUEUE = struct ... end;
structure QueueFast: QUEUE = struct ... end;
```

Man kann auch zu einer existierenden Implementierung eine Definition zufügen, die erklärt, dass sie eine Schnittstelle implementiert (fehlt in Java):

```
structure MyQueue = struct ... end;
...
structure QueueBest: QUEUE = MyQueue;
```

## Generische Module

Ein **generischer Modul** (**functor**) hat Strukturen als generische Parameter. (vgl. Klassen als generische Parameter von generischen Definition in C++, Ada, Java)

Formale generische Parameter sind mit einer Signatur typisiert.

Damit können Anforderungen an den aktuellen generischen Parameter formuliert werden, z. B.

- „muss eine Schlangenimplementierung sein“,
- „muss Typ mit Ordnungsrelation sein“.

Garantiert Typ-sichere Benutzung von Modulfunktionen (nicht in C++ und Ada).

Beispiel: **functor** für Breitensuche mit Schlange:

```
functor BreadthFirst (Q: QUEUE) =
  struct fun enqList q xs =
    foldl (fn (x,r)=> Q.enq(r,x)) q xs;
    fun search next x = ...
  end;
```

Der Funktor wird mit einem zur Signatur passenden Modul **instanziiert**:

```
structure Breadth = BreadthFirst (QueueFast);
```

## Beispiel im Zusammenhang: Wörterbuch-Funktor (1)

Aufzählungstyp (`datatype` mit Konstruktorfunktionen):

```
datatype order = LESS | EQUAL | GREATER;
```

Typen mit Ordnung als Schnittstelle (`signature`):

```
signature ORDER =
sig
  type t
  val compare: t * t -> order
end;
```

Konkreter Modul (`structure`) für String-Vergleiche:

```
structure StringOrder: ORDER =
struct
  type t = string;
  val compare = String.compare
end;
```

## Beispiel im Zusammenhang: Wörterbuch-Funktor (2)

Generischer Modul (`functor`) für Wörterbücher implementiert mit binärem Suchbaum:

```
functor Dictionary (Key: ORDER): DICTIONARY =
struct
  type key = Key.t;
  abstype 'a t = Leaf | Bran of key * 'a * 'a t * 'a t
  with exception E of key;
  val empty = Leaf;
  fun lookup (Leaf, b) = raise E b
  | lookup (Bran(a,x,t1,t2), b) =
    (case Key.compare (a, b) of
     GREATER => lookup (t1, b)
     EQUAL => x
     ...
    )
  ...
end
end;
```

Instanziierung des Funktors für einen Wörterbuch-Modul mit String-Schlüsseln:

```
structure StringDict = Dictionary (StringOrder);
```

Erzeugung und Benutzung eines Wörterbuches:

```
val dict = StringDict.update(..(StringDict.empty, "Kastens", 6686));
val tel = StringDict.lookup (dict, "Kastens");
```

## 6. Funktionen als Daten, Übersicht

**Orthogonales Typsystem:** Funktionen sind beliebig mit anderen Typen kombinierbar

**Notation für Funktionswerte** (Lambda-Ausdruck):

```
fn (z,k) => z*k
```

**Datenstrukturen** mit Funktionen als Komponenten:

z. B. Suchbaum für Funktionen

**Funktionale, Funktionen höherer Ordnung** (higher order functions, HOF):

haben **Funktionen als Parameter oder als Ergebnis**

**Berechnungsschemata:**

Funktion als Parameter abstrahiert Operation im Schema,  
wird bei Aufruf des Schemas konkretisiert

```
foldl (fn (z,k) => z*k, [2,5,1], 1);
```

 (hier noch ohne Currying)

**schrittweise Parametrisierung (Currying):**

Funktion als Ergebnis bindet ersten Parameter,  
nützliche Programmieretechnik, steigert Wiederverwendbarkeit

```
val chorner = fn l => fn x => foldl (fn (z,k) => z*x+k, 1, 0);
```

**nicht-endliche Datenstrukturen (Ströme, lazy evaluation),** (Kapitel 7):

Funktionen als Komponenten von Datenstrukturen,  
z. B. Funktion, die den Rest einer Liste liefert

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

## Notation von Lambda-Ausdrücken

**Auswertung eines Lambda-Ausdruckes**

liefert eine Funktion, als Datum, unbenannt.

**Notation:**

```
fn ParameterMuster => Ausdruck
```

```
fn (z,k) => z*k
```

```
fn (x, y) => Math.sqrt (x*x + y*y)
```

**mit Fallunterscheidung:**

```
fn Muster1 => Ausdruck1
```

```
| Muster2 => Ausdruck2
```

```
| ...
```

```
| Mustern => Ausdruckn
```

```
fn nil => true
```

```
| (_::_) => false
```

Anwendungen von Lambda-Ausdrücken:

```
linsert (1, fn (z,k) => z*x+k, 0)
```

```
(fn (z,k) => z*k) (a, b)
```

```
if b then fn (z,k) => z*k
else fn (z,k) => z+k
```

```
[fn (z,k) => z*k, fn (z,k) => z+k]
```

```
val null = fn nil => true
           | (_::_) => false;
```

```
fun Comp (f, g) = fn x => f (g x);
```

## Currying

**Haskell B. Curry:** US-amerikanischer Logiker 1900-1982, Combinatory Logic (1958); Moses Schönfinkel, ukrainischer Logiker, hat die Idee schon 1924 publiziert:

Funktionen **schrittweise parametrisieren statt vollständig mit einem Parametertupel**.  
abstraktes Prinzip für eine n-stellige Funktion:

### Tupelform:

Signatur: `gn: ('t1 * 't2 * ... * 'tn) -> 'r`  
 Funktion: `fun gn ( p1, p2, ..., pn) = Ausdruck über p1, ..., pn`

Aufrufe: `gn ( a1, a2, ..., an)` liefert Wert vom Typ 'r

### ge-curried:

Signatur: `cgn: 't1->('t2->...->( 'tn->'r)...)...`  
 Funktion: `fun cgn p1 p2 ... pn = Ausdruck über p1, ..., pn`

Aufruf: liefert Wert vom Typ  
`(cgn a1 a2 ... an)` 'r  
`(cgn a1 a2 ... an-1)` 't<sub>n</sub>->'r  
 ...  
`(cgn a1)` 't<sub>2</sub>->(...'t<sub>n</sub>->'r)...

Ergebnisfunktionen tragen die schon gebundenen Parameter in sich.

**Funktion voll-parametrisiert entwerfen - teil-parametrisiert benutzen!**

## Currying: Beispiele

Parametertupel:

```
fun prefix (pre, post) = pre ^ post;
Signatur: string * string -> string
```

ge-curried:

```
lang: fun prefix pre = fn post => pre ^ post;
kurz: fun prefix pre post = pre ^ post;
Signatur: string -> (string -> string)
gleich: string -> string -> string
```

Erster Parameter (**pre**) ist in der Ergebnisfunktion gebunden.

Anwendungen:

```
val knightify = prefix "Sir ";
val dukify = prefix "The Duke of ";
knightify "Ratcliff";
(prefix "Sir ") "Ratcliff";
prefix "Sir " "Ratcliff"; linksassoziativ
```

auch rekursiv: **x** oder **n** ist in der Ergebnisfunktion gebunden

```
fun repxlist x n = if n=0 then [] else x :: repxlist x (n-1);
fun repnlist n x = if n=0 then [] else x :: repnlist (n-1) x;
(repulist 7); (repulist 3);
```

## Funktionen in Datenstrukturen

Liste von Funktionen:

```
val titlefns =
  [prefix "Sir ",
   prefix "The Duke of ",
   prefix "Lord "] : (string -> string) list
```

```
hd (tl titlefns) "Gloucester";
```

Suchbaum mit (string \* (real -> real)) Paaren:

```
val fntree =
  Dict.insert
    (Dict.insert
      (Dict.insert
        (Lf, "sin", Math.sin),
         "cos", Math.cos),
       "atan", Math.atan);
```

```
Dict.lookup (fntree, "cos") 0.0;
```

## Currying als Funktional

**Funktional:** Funktionen über Funktionen; Funktionen höherer Ordnung (HOF)

**secl, secr** (section):

**2-stellige Funktion in Curry-Form** wandeln; dabei den linken, rechten **Operanden binden**:

```
fun secl x f y = f (x, y);
'a -> ('a * 'b -> 'c) -> 'b -> 'c
```

```
fun secr f y x = f (x, y);
('a * 'b -> 'c) -> 'b -> 'a -> 'c
```

Anwendungen:

```
fun power (x, k):real =if k = 1 then x else
                      if k mod 2 = 0then power (x*x, k div 2)
                      else x *power (x*x, k div 2);
```

```
val twoPow = secl 2.0 power; int -> real
```

```
val pow3 = secr power 3; real -> real
```

```
map (1, secr power 3);
```

```
val knightify = (secl "Sir " op^); string -> string
```

`op^` bedeutet infix-Operator `^` als Funktion

## Komposition von Funktionen

Funktional `cmp` verknüpft Funktionen `f` und `g` zu deren Hintereinanderausführung:

```
fun cmp (f, g) x = (f (g x));
```

Ausdrücke mit **Operatoren**, die Funktionen zu neuen **Funktionen verknüpfen**,

2-stelliger **Operator** `o` statt 2-stelliger Funktion `cmp`:

```
infix o;
fun (f o g) x = f (g x);      ('b->'c) * ('a->'b) -> 'a -> 'c
```

Funktionen nicht durch **Verknüpfung von Parametern** in Lambda-Ausdrücken definieren:

```
fn x => 2.0 / (x - 1.0)
```

sondern durch **Verknüpfung von Funktionen** (algebraisch) berechnen:

```
(secl 2.0 op/) o (secl op- 1.0)
```

**Potenzieren von Funktionen**  $f^n(x)$ :

```
fun repeat f n x = if n > 0 then repeat f (n-1) (f x) else x;
repeat: ('a->'a) -> int -> 'a -> 'a
```

Aufrufe:

```
(repeat (secl op/ 2.0) 3 800.0);      (repeat tl 3 [1,2,3,4]);
(repeat (secl op/ 2.0) 3);          (repeat tl 3);
(repeat (secl op/ 2.0));            (repeat tl);
```

[John Backus: Can Programming Be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs; 1977 ACM Turing Award Lecture; CACM, vol. 21, no. 8, 1978]

## Kombinatoren

**Kombinator**: Funktion ohne freie Variable

**Kombinatorischer Term T**:

T ist ein Kombinator oder T hat die Form  $(T1, T2)$  und  $T_i$  sind kombinatorische Terme

Kombinatorische Terme dienen

zur **Verknüpfung** und zu algebraischer **Transformation** von Funktionen,  
zur Analyse und zum **Beweis** von Programmen

**David Turner** (britischer Informatiker) hat 1976 gezeigt, dass **alle Funktionen des Lambda-Kalküls** durch die klassischen Kombinatoren `s` und `k` darstellbar sind.

**Klassische Kombinatoren S K I**:

```
fun I x = x;          Identitätsfunktion          'a -> 'a
fun K x y = y;       bildet Konstante Fkt.        'a -> 'b -> 'a
fun S x y z = x z (y z); wendet x auf y an, nach Einsetzen von z in beide
                        ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

I entspricht `s k k` denn  $((s\ k\ k)\ u) = (s\ k\ k\ u) = (k\ u\ (k\ u)) = u$

**Beispiel**:

Der Lambda-Ausdruck  $(\lambda x (\lambda y (x\ y)))$   
kann in  $(s\ (k\ (s\ I))\ (s\ (k\ k)\ I))$  transformiert werden.

## Reihenberechnung als Schema

Allgemeine Formel für eine **endliche Reihe**:  $\sum_{i=0}^{m-1} f(i)$

```
fun summation f m =
  let fun sum (i, z):real =                akkumulierende Hilfsfunktion
        if i=m then z else sum (i+1, z + (f i))
      in sum (0, 0.0) end;
```

Signatur:  $(int \rightarrow real) \rightarrow int \rightarrow real$

Aufruf `summation (fn k => real(k*k)) 5;` liefert 30

**Doppelsumme**:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} g(i,j) \quad \text{summation als Parameter von summation:}$$

```

      Bindung
      |
summation(fn i => summation (fn j => g(i,j)) n) m
      |
int->real          int->real
```

einfacher `h i j` statt `g(i,j)`: `summation (fn i => summation (h i) n) m;`

Kombination von Funktionen, Konversion nach `real`: `summation (Math.sqrt o real);`

Summe konstanter Werte; Kombinator `k` für konstante Funktion: `summation (K 7.0) 10;`

## Funktionale für Listen: map

Liste elementweise mit einer Funktion abbilden:

```
map f [x1,...,xn] = [f x1,..., f xn]
```

```
fun map f nil = nil
  | map f (x::xs) = (f x) :: map f xs;
Signatur: ('a -> 'b) -> 'a list -> 'b list
```

Anwendungen:

```
map size ["Hello", "World!"];
map (secl 1.0 op/) [0.1, 1.0, 5.0];
```

für 2-stufige Listen (setzt `map` in Curry-Form voraus!):

```
map (map double) [[1], [2, 3]];
```

statt `map f (map g l)` besser `map (f o g) l`

Matrix transponieren:

```
fun transp (nil::_) = nil
  | transp rows =
    map hd rows :: transp (map tl rows);
```

## Funktionale für Listen: Filter

Schema: Prädikatfunktion wählt Listenelemente aus:

```
fun filter pred nil = nil
| filter pred (x::xs) = if pred x then x :: (filter pred xs)
                       else (filter pred xs);
```

Anwendungen:

```
filter (fn a => (size a) > 3) ["Good", "bye", "world"];
fun isDivisorOf n d = (n mod d) = 0;
filter (isDivisorOf 360) [24, 25, 30];
```

Mengendurchschnitt (`mem` ist auf nächster Folie definiert):

```
fun intersect xs ys = filter (secl (op mem) ys) xs;
```

Variationen des Filterschemas:

```
val select = filter;
fun reject f = filter ((op not) o f);
```

```
fun takewhile pred nil = nil
| takewhile pred (x::xs) = if pred x then x::(takewhile pred xs)
                           else nil;

takewhile isPos [3, 2, 1, 0, ~1, 0, 1];
fun dropwhile ... entsprechend
```

## Funktionale für Listen: Quantoren

Existenz und All-Quantor:

```
fun exists pred nil = false
| exists pred (x::xs) = (pred x) orelse (exists pred xs);

fun all pred nil = true
| all pred (x::xs) = (pred x) andalso (all pred xs);
```

Member-Operator:

```
infix mem;
fun x mem xs = exists (secl op= x) xs;
```

Disjunkte Listen?

```
fun disjoint xs ys = all (fn x => all (fn y => y<>x) ys) xs;
oder:
fun disjoint xs ys = all (fn x => (all (secl op<> x) ys)) xs;
```

Quantoren-Funktionale für Listen von Listen:

```
exists (exists pred)      z.B. exists (exists (secl 0 op=))
filter (exists pred)     z.B. filter (exists (secl 0 op=))
takewhile (all pred)     z.B. takewhile (all (secl op> 10))
```

## Funktionale verknüpfen Listenwerte

Listenelemente mit 2-stelliger Funktion `f` verknüpfen:

```
foldl f e [x1, ..., xn] = f(xn, ... f(x2, f(x1, e)))...
foldr f e [x1, ..., xn] = f(x1, ... f(xn-1, f(xn, e)))...
```

`foldl` verknüpft Elemente sukzessive vom ersten zum letzten.

`foldr` verknüpft Elemente sukzessive vom letzten zum ersten.

```
fun foldl f e nil = e          akk. Parameter
| foldl f e (x::xs) = foldl f (f(x, e)) xs;

fun foldr f e nil = e
| foldr f e (x::xs) = f(x, foldr f e xs);
```

Signatur: ('a \* 'b -> 'b) -> 'b -> 'a list -> 'b

Beispiel: `val sum = foldl op+ 0;`

Verknüpfungsreihenfolge bei `foldl` und `foldr`:

```
val difl = foldl op- 0;      difl [1,10]; ergibt 9
val difr = foldr op- 0;     difr [1,10]; ergibt ~9
```

Horner-Schema in Curry-Form:

```
fun horner l x = foldl (fn (h,a) => a*x+h) 0.0 l;
```

Liste umkehren: `fun reverse l = foldl op:: nil l;`

Menge aus Liste erzeugen: `fun setof l = foldr newmem [] l;
setof [1,1,2,4,4];`

## Werte in binären Bäumen

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

Schema:

Für jedes Blatt einen Wert `e` einsetzen und an inneren Knoten Werte mit 3-stelliger Funktion verknüpfen (vergl. `foldr`):

```
fun treefold f e Lf = e
| treefold f e (Br (u,t1,t2)) =
  f(u, treefold f e t1, treefold f e t2);
```

Anwendungen

Anzahl der Knoten:

```
treefold (fn (_, c1, c2) => 1 + c1 + c2) 0 t;
```

Baumtiefe:

```
treefold (fn (_, c1, c2) => 1 + max(c1, c2)) 0 t;
```

Baum spiegeln:

```
treefold (fn (u, t1, t2) => Br (u, t2, t1)) Lf t;
```

Werte als Liste in Preorder (flatten):

```
treefold (fn (u, l1, l2) => [u] @ l1 @ l2) nil t;
```





## Stromfunktionen im Modul Seq

Funktionen für Ströme sind im Modul Seq zusammengefasst:

Seq.hd, Seq.tl, Seq.null, Seq.take, Seq.drop, Seq. @,  
Seq.interleave, Seq.map, Seq.filter, Seq.iterates,  
Seq.from, Seq.fromlist, Seq.tolist

Beispiel: Strom von Zufallszahlen:

```
localval a = 16807.0 and m = 2147483647.0
```

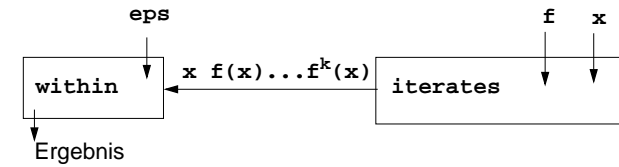
```
fun nextRand seed =  
  let val t = a*seed  
      in t - m * real (Real.floor(t/m))  
      end
```

```
in fun randseq s = Seq.map(secr op/ m)  
    (Seq.iterates nextRand (real s))  
end;
```



## Ströme zusammensetzen

Schema: Konvergenzabbruch für iterierte Funktion



Beispiel: Quadratwurzel iterativ berechnen:

```
fun nextApprox a x = (a/x + x) / 2.0;
```

```
fun within (eps:real) (Cons(x,xf)) =  
  let val Cons (y,yf) = xf()  
      in if Real.abs (x-y) < eps  
         then y  
         else within eps (Cons (y,yf))  
      end;
```

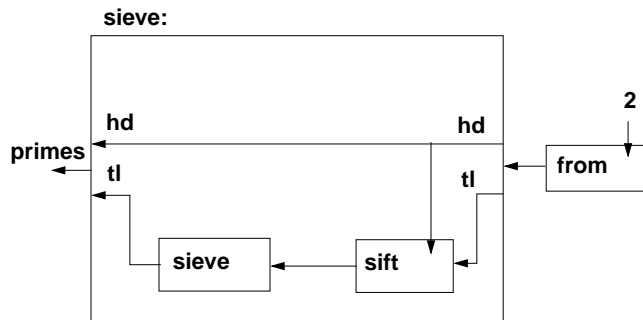
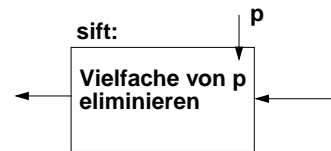
```
fun qroot a =  
  within 1E-12 (Seq.iterates (nextApprox a) 1.0);
```

## Ströme rekursiv zusammensetzen

```
fun sift p =  
  Seq.filter (fn n => n mod p <> 0);
```

```
fun sieve (Cons(p,nf)) =  
  Cons (p, fn() => sieve (sift p (nf())));
```

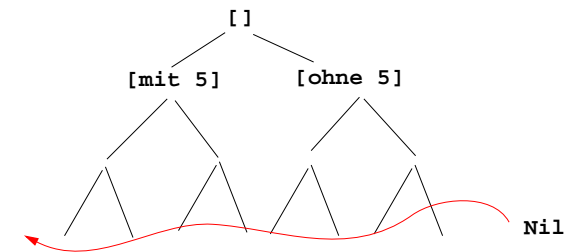
```
val primes = sieve (Seq.from 2);  
Seq.take (primes, 25);
```



Primzahlen mit dem Sieb des Eratosthenes berechnen:

## Strom aller Lösungen im Baum-strukturierten Lösungsraum

Beispiel Münzwechsel: Strom von Lösungen der Form [5, 2, 1, 1] berechnen



- endliche Zahl von Lösungen: abbrechender Strom
- **Listenkonkatenation @** darf nicht in **Stromkonkatenation Seq.@** geändert werden! Strom würde dann **vollständig ausgewertet!**
- Funktion akkumuliert Strom elementweise
- akkumulierender Parameter berechnet Restfunktion des Stromes mit **Cons (x, xf)**

## Beispiel Münzwechsel mit Strömen

### Signatur:

```
int list * int list * int * (unit -> int list seq) -> int list seq
```

### Funktionsdefinition seqChange:

```
fun
    neue Lösung coins in den Strom geben:
    seqChange (coins, coinvals, 0, coinsf) = Seq.Cons (coins, coinsf)
    ist keine Lösung, Strom bleibt unverändert:
    | seqChange (coins, [], amount, coinsf) = coinsf ()
    | seqChange (coins, c::coinvals, amount, coinsf)=
      if amount<0
          ist keine Lösung, Strom bleibt unverändert:
          then coinsf ()
          else seqChange
              erster Zweig „mit Münze c“:
              (c::coins, c::coinvals, amount-c,
                zweiter Zweig „ohne Münze c“, lazy:
                fn() => seqChange (coins, coinvals, amount, coinsf));
```

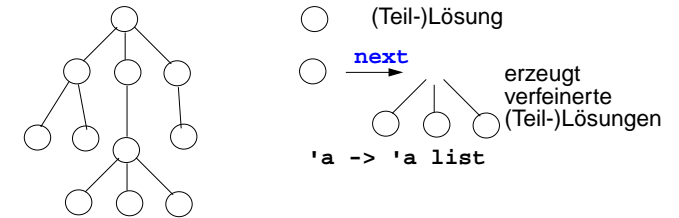
### Aufruf mit abbrechender Rest-Funktion:

```
seqChange ([], gb_coins, 99, fn () => Seq.Nil);
```

liefert die erste Lösung im Paar Seq.Cons ([...], f); die nächste mit Seq.tl it

## Funktional für Tiefensuche in Lösungsbäumen

- Strom entkoppelt Erzeuger und Verwender der Lösungen
- Funktional bestimmt die Suchstrategie des Erzeugers
- Die Aufgabe wird durch **next** und **pred** bestimmt



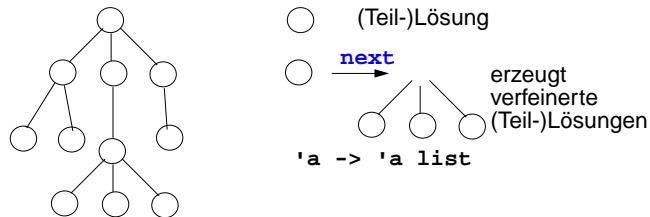
**DFS Tiefensuche: effizient; aber terminiert nicht bei unendlichen Teilbäumen**  
Prädikat **pred** entscheidet, ob eine Lösung vorliegt:

```
fun depthFirst (next, pred) root =
  let fun dfs [] = Nil
      | dfs (x::xs) =
          if pred x
              then Cons (x, fn () => dfs ((next x) @ xs))
              else dfs ((next x) @ xs)
          in dfs [root] end;
```

Keller:

## Funktional für Breitensuche in Lösungsbäumen

- Strom entkoppelt Erzeuger und Verwender der Lösungen
- Funktional bestimmt die Suchstrategie des Erzeugers
- Die Aufgabe wird durch **next** und **pred** bestimmt



**BFS Breitensuche: vollständig; aber speicheraufwendig:**

```
fun breadthFirst (next, pred) root =
  let fun bfs [] = Nil
      | bfs (x::xs) =
          if pred x
              then Cons (x, fn () => bfs(xs @ next x))
              else bfs (xs @ next x)
          in bfs [root] end;
```

Schlange:

## Funktionale anwenden für Münzwechsel

**Knoten des Lösungsbaumes sind Tripel**

**(ausgezählte Münzen, verfügbare Münzwerte, zu zahlender Betrag):**

```
fun predCoins (paid, coinvals, 0) = true
  | predCoins _ _ _ = false;
```

```
fun nextCoins (paid, coinvals, 0) = []
  | nextCoins (paid, nil, amount) = []
  | nextCoins (paid, c::coinvals, amount) =
      if amount < 0
          then []
          else [ (c::paid, c::coinvals, amount-c),
                 (paid, coinvals, amount) ];
```

```
val euro_coins = [200, 100, 50, 20, 10, 5, 2, 1];
val coins52Dep = depthFirst (nextCoins, predCoins) ([],[5,2], 30);
val coins52Bre = breadthFirst (nextCoins, predCoins) ([],[5,2], 30);
val coinsEuroBre = ([], euro_coins, 30);
```

## Funktionale anwenden erzeugung von Palindromen

FP-7.12b

Ein Knoten des Lösungsbaumes ist eine **Liste von Zeichen**:

```
fun nextChar l = ["A"::l, "B"::l, "C"::l];
fun isPalin l = (l = rev l);
```

```
val palinABCBre = breadthFirst (nextChar, isPalin) [];
val palinABCDep = depthFirst (nextChar, isPalin) [];
```

© 2013 bei Prof. Dr. Uwe Kastens

## Weiter verzögerte Auswertung

FP-7.13

Datentyp lazySeq berechnet ein Paar erst, wenn es gebraucht wird:

```
datatype 'a lazySeq = LazyNil | LazyCons of unit -> 'a * 'a lazySeq
fun from k = LazyCons (fn () => (k, from (k + 1)));
```

```
fun take (xq, 0) = nil
  | take (LazyNil, n) = raise Seq.Empty
  | take (LazyCons xf, n) = let val (x, xt) = xf ()
                           in x :: take (xt, n - 1)
                           end;
```

noch weiter verzögert: leerer oder nicht-leerer Strom wird erst entschieden, wenn nötig.

```
datatype 'a seqNode = l1Nil | l1Cons of 'a * 'a l1Seq;
```

```
datatype 'a l1Seq = Seq of unit -> 'a seqNode;
```

© 2010 bei Prof. Dr. Uwe Kastens

## 8. Lazy Evaluation

FP-8.1

eager

vs.

lazy

**allgemeines Prinzip (auch in der SWT):**

**Erst** werden **alle evtl. benötigten Werte** berechnet, dann die Operation darauf ausgeführt.

**Strikte Auswertung:** Wenn ein Operand **bottom** liefert (nicht terminiert), dann liefert auch der Ausdruck **bottom**.

Eine **Berechnung** wird erst dann ausgeführt, **wenn ihr Ergebnis benötigt** wird.

Zusammengesetzte Ergebnisse werden **nur so tief wie nötig ausgewertet**.

Mehrfach benötigte **Ergebnisse** werden nur einmal berechnet und dann **wiederverwendet**.

**Parameterübergabe:** call-by-value

call-by-name, call-by-need

**Datenstrukturen:** Listen

Ströme

**Sprachsemantik:** SML, Lisp

Haskell, Miranda

© 2010 bei Prof. Dr. Uwe Kastens

## Einführung in Notationen von Haskell

FP-8.2

**Definitionen von Funktionen:**

```
add :: Int -> Int -> Int
add x y = x + y
```

vorangestellte Signatur ist guter Stil, aber nicht obligatorisch

```
incl :: Int -> Int
incl = add 1
```

```
inc2 :: Int -> Int
inc2 = (+2)
```

entspricht (`seccr op+ 2`) in SML

```
sub :: Int -> Int -> Int
sub = \x y -> x - y
```

Lambda-Ausdruck in Haskell

**Funktionen über Listen:**

```
lg :: [a] -> Int
lg [] = 0
lg (_:xs) = 1 + lg xs
```

```
xmap f [] = []
xmap f (x:xs) = (f x) : (xmap f xs)
Aufruf z. B.: xmap (+2) [1,2,3]
```

```
quicksort [] = []
quicksort (x:xs) =
  quicksort [y | y <- xs, y < x] ++
  [x] ++
  quicksort [y | y <- xs, y >= x]
```

© 2012 bei Prof. Dr. Uwe Kastens

## Lazy-Semantik in Haskell

FP-8.3

Die Semantik von Haskell ist **konsequent lazy**, nur elementare Rechenoperationen (+, \*, ...) werden strikt ausgewertet.

Beispiele:

```
inf = inf
```

ist wohldefiniert; aber die Auswertung würde nicht terminieren.

```
f x y = if x == 0 then True else y
```

Parameterübergabe call-by-need:

```
f 0 inf                                liefert True
```

```
f inf False                            terminiert nicht, liefert bottom
```

© 2005 bei Prof. Dr. Uwe Kastens

## Lazy Listen in Haskell

FP-8.4

**Listen** in Haskell haben Lazy-Semantik - wie alle Datentypen.

Definition einer **nicht-endlichen Liste** von 1en:

```
ones :: [Int]
ones = 1 : ones
```

```
take 4 ones          liefert [1, 1, 1, 1]
```

Funktionsaufrufe brauchen nicht zu terminieren:

```
numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)
```

```
take 4 (numsFrom 3) liefert [3, 4, 5, 6]
```

© 2004 bei Prof. Dr. Uwe Kastens

## Listen als Ströme verwenden

FP-8.5

Listen können unmittelbar wie Ströme verwendet werden:

```
squares :: [Int]
squares = map (^2) (numsFrom 0)
```

```
take 5 squares          liefert [0, 1, 4, 9, 16]
```

Paradigma Konvergenz (vgl. FP-7.7):

```
within :: Float -> [Float] -> Float
within eps (x1:(x2:xs)) =
  if abs(x1-x2)<eps then x2 else within eps (x2:xs)
```

```
myIterate :: (a->a) -> a -> [a]
myIterate f x = x : myIterate f (f x)
```

```
nextApprox a x = (a / x + x) / 2.0
```

```
groot a = within 1e-8 (myIterate (nextApprox a) 1.0)
```

Strom von Fibonacci-Zahlen:

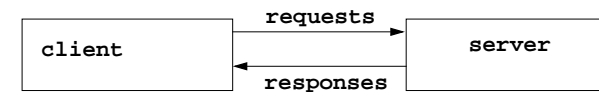
```
fib :: [Int]                zip erzeugt Strom von Paaren
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

```
fibs :: [Int]              zipWith verknüpft die Elemente zweier Ströme
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))
```

© 2014 bei Prof. Dr. Uwe Kastens

## Simulation zyklischer Datenflüsse

FP-8.6



```
reqs = client csInit resps
resps = server reqs
```

```
server :: [Int] -> [Int]
server (req:reqs) = process req : server reqs
```

```
client :: Int -> [Int] -> [Int]
```

```
-- client init (resp:resps) = init : client (next resp) resps
-- Fehler: das zweite Pattern wird zu früh ausgewertet
```

```
-- client init resps = init : client (next (head resps)) (tail resps)
-- funktioniert: Das zweite Pattern wird erst bei Benutzung ausgewertet
```

```
client init ~(resp:resps) = init : client (next resp) resps
-- Das zweite Pattern wird erst bei Benutzung ausgewertet
```

```
csInit      = 0
next resp   = resp
process req = req+1
```

© 2004 bei Prof. Dr. Uwe Kastens

## Beispiel: Hamming-Folge

Erzeuge eine Folge  $X = x_0, x_1, \dots$  mit folgenden Eigenschaften:

1.  $x_{i+1} > x_i$  für alle  $i$
2.  $x_0 = 1$
3. Falls  $x$  in der Folge  $X$  auftritt, dann auch  $2x$ ,  $3x$  und  $5x$ .
4. Nur die durch (1), (2) und (3) spezifizierten Zahlen treten in  $X$  auf.

Funktion zum Verschmelzen zweier aufsteigend sortierten Listen zu einer ohne Duplikate:

```
setMerge :: Ord a => [a] -> [a] -> [a]
setMerge allx@(x:xs) ally@(y:ys) -- allx ist Name für das gesamte Pattern
  | x == y    = x : setMerge xs    ys
  | x < y    = x : setMerge xs    ally
  | otherwise = y : setMerge ally  xs
```

Funktion für die Hamming-Folge, wie definiert:

```
hamming :: [Int]
hamming = 1 : setMerge (map (*2) hamming)
                  (setMerge (map (*3) hamming)
                          (map (*5) hamming))
```

## 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. Toft, 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](http://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`, `exists`, `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) :: King () ::
  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:**

```
11.map ( x=>x*x )           Quadrat-Funktion
13.map ( _ => 5 )           konstante Funktion
12.map ( Math.sin _ )      Sinus-Funktion
14.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 **Parameter Ausdruck** 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 Ausdrucksmöglichkeiten.
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 `sec1`, `secr`: Definition, Signatur und Anwendungen

23. Weitere allgemeine Funktionale (`o`, `iterate`, `s κ 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