

JG|U

JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Programmiersprachen (08.079.030)

7 - Funktionale Programmierung

Tim Süß

Institut für Informatik

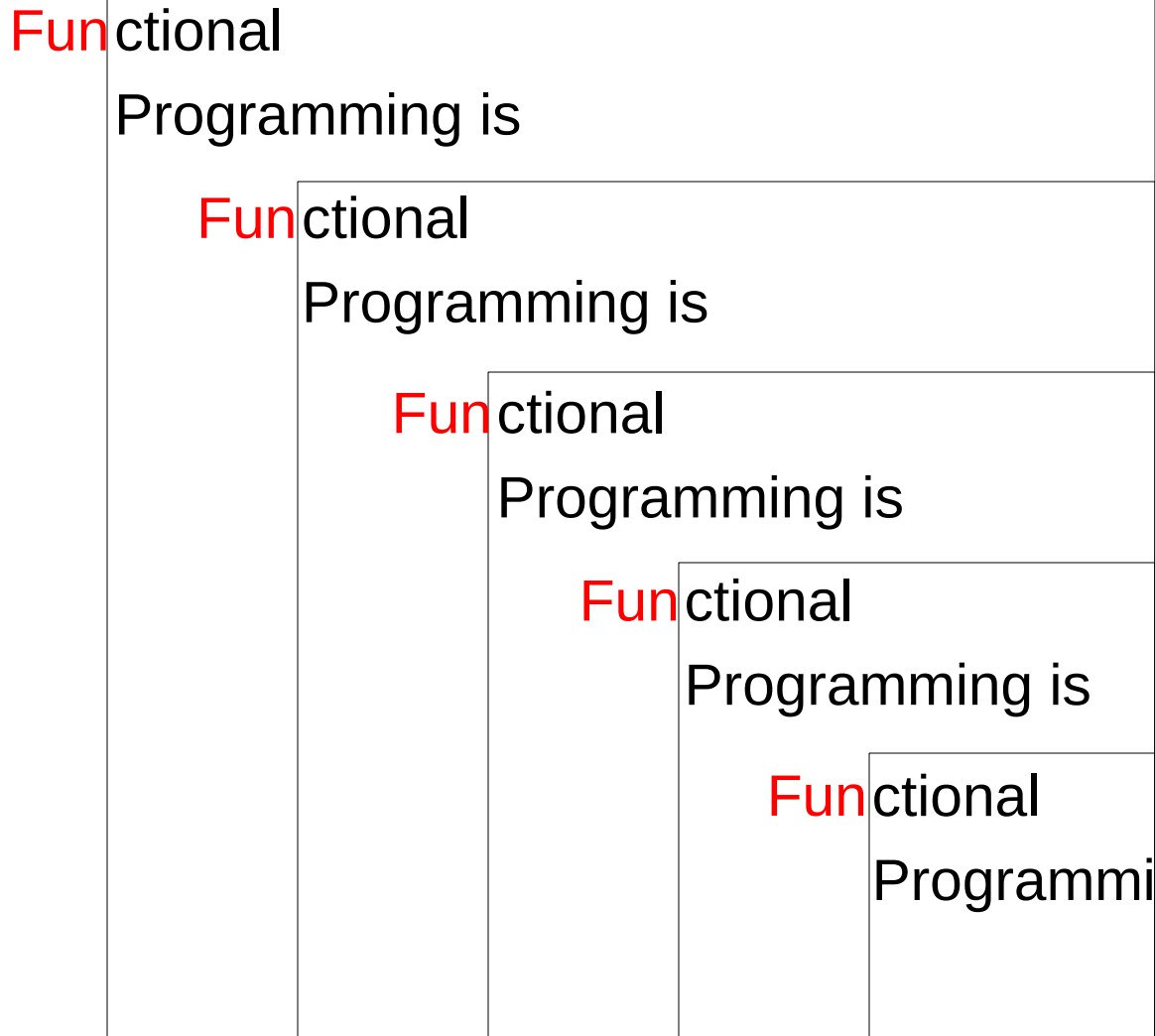
Johannes Gutenberg-Universität Mainz

Funktionale Programmierung

Themen

- Grundbegriffe und Notation von SML
- Rekursionsparadigmen: Induktion, Rekursion über Listen
- End-Rekursion und Programmiertechnik „akkumulierender Parameter“
- Berechnungsschemata mit Funktionen als Parameter
- Funktionen als Ergebnis und Programmiertechnik „Currying“

Functional Programming is Fun



Übersicht zur funktionalen Programmierung

Grundkonzepte: Funktionen und Aufrufe, Ausdrücke
keine Variablen, Zuweisungen, Ablaufstrukturen, Seiteneffekte

Elementare Sprachen (pure LISP) brauchen nur wenige Konzepte:
Funktionskonstruktor, bedingter Ausdruck, Literale, Listenkonstruktor und
-selektoren, Definition von Bezeichnern für Werte

Mächtige Programmierkonzepte durch Verwendung von: rekursiven Funktionen
und Datenstrukturen, Funktionen höherer Ordnung als Berechnungsschemata

Höhere funktionale Sprachen (SML, Haskell):
statische Bindung von Bezeichnern und Typen, völlig orthogonale, höhere
Datentypen, polymorphe Funktionen (Kapitel 6), modulare Kapselung, effiziente
Implementierung

Funktionaler Entwurf: strukturell denken - nicht in Abläufen und veränderlichen
Zuständen, fokussiert auf **funktionale Eigenschaften** der Problemlösung, Nähe zur
Spezifikation, Verifikation, Transformation

Funktionale Sprachen: LISP, Scheme, Hope, SML, Haskell, Miranda, ...
früher: Domäne der KI; heute: Grundwissen der Informatik, praktischer Einsatz

Sprachkonstrukte von SML: Funktionen

Funktionen können direkt notiert werden, ohne Deklaration und ohne Namen:

Funktionskonstruktor (lambda-Ausdruck: Ausdruck, der eine Funktion liefert):

fn **FormalerParameter** => **Ausdruck**

`fn i => 2 * i` Funktion, deren Aufruf das Doppelte ihres Parameters liefert

`fn (a, b) => 2 * a + b`

Beispiel, unbenannte Funktion als Parameter eines Aufrufes:

`map (fn i => 2 * i, [1, 2, 3])`

Funktionen haben **immer einen Parameter**:

statt mehrerer Parameter ein Parameter-Tupel wie (a, b)

(a, b) ist ein **Muster** für ein Paar als Parameter

statt keinem Parameter ein leerer Parameter vom Typ **unit**, entspricht **void**

Typangaben sind optional. Trotzdem prüft der Übersetzer streng auf korrekte Typisierung. Er berechnet die Typen aus den benutzten Operationen (**Typinferenz**)

Typangaben sind nötig zur **Unterscheidung von** `int` **und** `real`

`fn i : int => i * i`

Sprachkonstrukte von SML: Funktionen

allgemeine Form eines Aufrufes: ***Funktionsausdruck Parameterausdruck***

```
Dupl 3  
(fn i => 2 * i) 3
```

Klammern können den Funktionsausdruck mit dem aktuellen Parameter zusammenfassen:

```
(fn i => 2 * i) (Dupl 3)
```

Parametertupel werden geklammert:

```
(fn (a, b) => 2 * a + b) (4, 2)
```

Auswertung von Funktionsaufrufen wie in Vorlesung 6 beschrieben.

Parameterübergabe: **call-by-strict-value**

Sprachkonstrukte von SML: Funktionen

Eine **Definition** bindet den Wert eines Ausdrucks an einen Namen:

```
val four = 4;  
val Dupl = fn i => 2 * i;  
val Foo = fn i => (i, 2*i);  
val x = Dupl four;
```

Eine Definition kann ein **Tupel von Werten** an ein **Tupel von Namen**, sog. **Muster**, binden:

allgemeine Form:

```
val Muster = Ausdruck;  
val (a, b) = Foo 3;
```

Der Aufruf `Foo 3` liefert ein Paar von Werten, sie werden gebunden an die Namen `a` und `b` im Muster für Paare `(a, b)`.

Kurzform für Funktionsdefinitionen:

```
fun Name FormalerParameter = Ausdruck;  
fun Dupl i = 2 * i;  
fun Fac n = if n <= 1 then 1 else n * Fac (n-1);
```

bedingter Ausdruck: Ergebnis ist der Wert des `then`- oder `else`-Ausdruckes

Rekursionsparadigma Induktion

Funktionen für induktive Berechnungen sollen schematisch entworfen werden:

Beispiele:

induktive Definitionen:

$$n! = \begin{cases} 1 & \text{für } n \leq 1 \\ n \cdot (n-1)! & \text{für } n > 1 \end{cases}$$

$$b^n = \begin{cases} 1.0 & \text{für } n \leq 0 \\ b \cdot b^{n-1} & \text{für } n > 0 \end{cases}$$

rekursive Funktionsdefinitionen:

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

```
fun Power (n, b) =  
  if n <= 0  
  then 1.0  
  else b * Power (n-1, b);
```

Schema:

```
fun F a =  
  if Bedingung über a  
  then nicht-rekursiver Ausdruck über a  
  else rekursiver Ausdruck über F("verkleinertes a")
```

Induktion - effizientere Rekursion

Induktive Definition und rekursive Funktionen zur Berechnung von Fibonacci-Zahlen:

induktive Definitionen:

$$\text{Fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{für } n > 1 \end{cases}$$

rekursive Funktionsdefinitionen:

```
fun Fib n =  
    if n = 0  
    then 0  
    else if n = 1  
         then 1  
         else Fib(n-1)+Fib (n-2);
```

Fib effizienter:

Zwischenergebnisse als Parameter, Induktion aufsteigend
(allgemeine Technik siehe „Akkumulierende Parameter“):

```
fun AFib (n, alt, neu) =  
    if n = 1 then neu  
    else AFib (n-1, neu, alt+neu);  
fun Fib n = if n = 0 then 0 else AFib (n, 0, 1);
```

Funktionsdefinition mit Fallunterscheidung

Funktionen können übersichtlicher definiert werden durch

- **Fallunterscheidung** über den Parameter - statt bedingter Ausdruck als Rumpf,
- Formuliert durch **Muster**
- **Bezeichner** darin werden **an Teil-Werte des aktuellen Parameters gebunden**

bedingter Ausdruck als Rumpf:

```
fun Fac n =  
  if n=1 then 1  
    else n * Fac (n-1);  
fun Power (n, b) =  
  if n = 0  
    then 1.0  
    else b * Power (n-1, b);
```

Fallunterscheidung mit Mustern:

```
fun Fac (1) = 1  
  | Fac (n) = n * Fac (n-1);  
fun Power (0, b) = 1.0  
  | Power (n, b) =  
    b * Power (n-1, b);  
fun Fib (0) = 0  
  | Fib (1) = 1  
  | Fib (n) =  
    Fib(n-1) + Fib(n-2);
```

Die Muster werden in der **angegebenen Reihenfolge** gegen den aktuellen Parameter geprüft. Es wird der erste Fall gewählt, dessen Muster trifft. Deshalb muss ein allgemeiner „**catch-all**“-Fall am Ende stehen.

Listen als rekursive Datentypen

Parametrisierter Typ für lineare Listen vordefiniert: (Typparameter 'a; polymorpher Typ)

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

definiert den 0-stelligen Konstruktor nil und den 2-stelligen Konstruktor ::

Schreibweisen für Listen:

```
x :: xs           eine Liste mit erstem Element x und der Restliste xs  
[1, 2, 3]         für 1 :: 2 :: 3 :: nil
```

Nützliche vordefinierte Funktionen auf Listen:

```
hd l             erstes Element von l  
tl l             Liste l ohne erstes Element  
length l         Länge von l  
null l           Prädikat: ist l gleich nil?  
l1 @ l2          Liste aus Verkettung von l1 und l2
```

Funktion, die die Elemente einer Liste addiert:

```
fun Sum l = if null l then 0  
            else (hd l) + Sum (tl l);
```

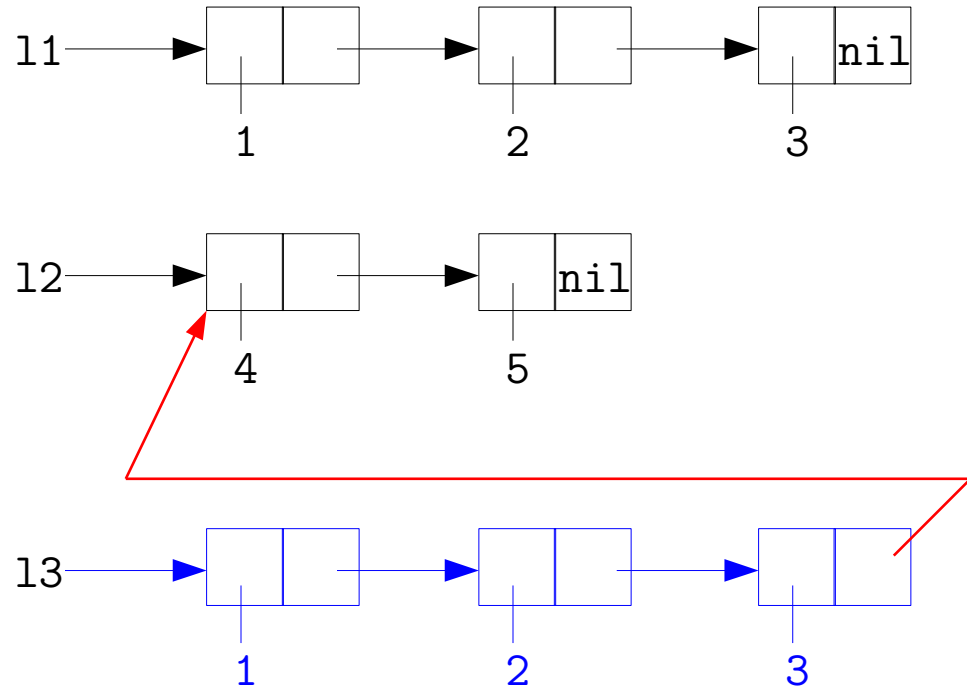
Signatur: Sum: int list -> int

Konkatenation von Listen

In funktionalen Sprachen werden Werte nie geändert.

Bei der Konkatenation zweier Listen wird die Liste des linken Operands **kopiert**.

```
val l1 = [1, 2, 3];  
val l2 = [4, 5];  
val l3 = l1 @ l2;
```



Einige Funktionen über Listen

Liste[n,...,1] erzeugen:

```
fun MkList 0 = nil
|   Mklist n = n :: MkList (n-1);
```

Signatur: `MkList: int -> int list`

Fallunterscheidung mit Listenkonstruktoren `nil` und `::` in Mustern:

Summe der Listenelemente:

```
fun Sum (nil) = 0
|   Sum (h::t) = h + Sum t;
```

Prädikat: Ist das Element in der Liste enthalten?:

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

Polymorphe Signatur: `Member: ('a list * 'a) -> bool`

Liste als Konkatenation zweier Listen berechnen (@-Operator):

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

Die linke Liste wird neu aufgebaut!

Polymorphe Signatur: `Append: ('a list * 'a list) -> 'a list`

Rekursionsschema Listen-Rekursion

lineare Listen sind als **rekursiver Datentyp** definiert:

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

Paradigma: Funktionen haben die gleiche Rekursionsstruktur wie der Datentyp:

```
fun F l = if l=nil      then nicht-rekursiver Ausdruck
           else Ausdruck über hd l und F (tl l);
```

```
fun Sum l = if l=nil    then 0
             else (hd l) + Sum (tl l);
```

Dasselbe in Kurzschreibweise mit Fallunterscheidung:

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

```
fun Sum (nil)        = 0
  | Sum (h::t)       = h + Sum t;
```

Einige Funktionen über Bäumen

Parametrisierter Typ für Bäume:

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil
```

Paradigma: Funktionen haben die gleiche Rekursionsstruktur wie der Datentyp.

Beispiel: einen Baum spiegeln

```
fun Flip (treeNil) = treeNil  
| Flip (node (l, v, r)) = node (Flip r, v, Flip l);
```

polymorphe Signatur: Flip: 'a tree -> 'a tree

Beispiel: einen Baum auf eine Liste der Knotenwerte abbilden (hier in Infix-Form)

```
fun Flatten (treeNil) = nil  
| Flatten (node (l, v, r)) = (Flatten l) @ (v :: (Flatten r));
```

polymorphe Signatur: Flatten: 'a tree -> 'a list

Präfix-Form: ...

Postfix-Form: ...

End-Rekursion

In einer Funktion **f** heißt ein **Aufruf** von **f** **end-rekursiv**, wenn er (als letzte Operation) das Funktionsergebnis bestimmt, sonst heißt er zentral-rekursiv.

Eine **Funktion** heißt **end-rekursiv**, wenn **alle rekursiven Aufrufe end-rekursiv** sind.

Member ist end-rekursiv:

```
fun Member (l, a) =  
  if null l then false  
  else if (hd l) = a  
    then true  
    else Member (tl l, a);
```

Sum ist zentral-rekursiv:

```
fun Sum (nil) = 0  
  | Sum (h::t) = h + (Sum t);
```

Parameter	Ergebnis
[1,2,3]	F
5	
[2,3]	F
5	
[2,3]	F
5	
[]	F
5	

Laufzeitkeller für Member ([1,2,3], 5)

Ergebnis wird durchgereicht -
ohne Operation darauf

End-Rekursion entspricht Schleife

Jede imperative Schleife kann in eine end-rekursive Funktion transformiert werden.

Allgemeines Schema:

```
while ( p(x) ) {x = r(x);} return q(x);  
fun While x = if p x then While (r x) else q x;
```

Jede **end-rekursive** Funktion kann in eine imperative Form transformiert werden:
Jeder **end-rekursive Aufruf** wird durch einen **Sprung** an den Anfang der Funktion (oder durch eine **Schleife**) ersetzt:

```
fun Member (l, a) =  
  if null l then false  
  else if (hd l) = a then true else Member (tl l, a);
```

Imperativ in C:

```
int Member (ElemList l, Elem a) {  
  Begin: if (null (l)) return 0 /*false*/;  
         else if (hd (l) == a) return 1 /*true*/;  
         else { l = tl (l); goto Begin;}  
}
```

Gute Übersetzer leisten diese Optimierung automatisch - auch in imperativen Sprachen.

Technik: Akkumulierender Parameter

Unter bestimmten Voraussetzungen können zentral-rekursive Funktionen in end-rekursive transformiert werden:

Ein **akkumulierender Parameter** führt das bisher berechnete Zwischenergebnis mit durch die Rekursion. Die Berechnungsrichtung wird umgekehrt, z. B.: Summe der Elemente einer Liste **zentral-rekursiv**:

```
fun Sum (nil)= 0
| Sum (h::t)= h + (Sum t);
```

Sum [1, 2, 3, 4] berechnet
 $1 + (2 + (3 + (4 + (0))))$

transformiert in end-rekursiv:

```
fun ASum (nil, a:int) = a
| ASum (h::t, a)      = ASum (t, a + h);
fun Sum l = ASum (l, 0);
```

ASum ([1, 2, 3, 4], 0) berechnet
 $(((((0 + 1) + 2) + 3) + 4))$

Die Verknüpfung (hier +) muß **assoziativ** sein.

Initial wird mit dem **neutralen Element der Verknüpfung** (hier 0) aufgerufen.

Gleiche Technik bei AFib; dort 2 akkumulierende Parameter.

Liste umkehren

Liste umkehren:

```
fun Reverse (nil)= nil
  | Reverse (h::t)= Append (Reverse t, h::nil);
```

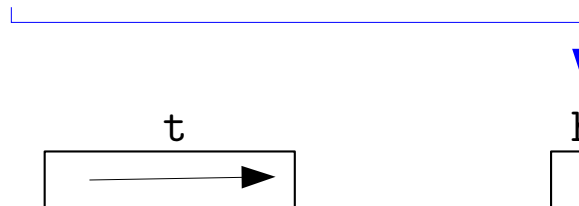
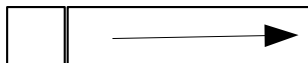
Append dupliziert die linke Liste bei jeder Rekursion von Reverse, benötigt also k mal $::$, wenn k die Länge der linken Liste ist. Insgesamt benötigt Reverse wegen der Rekursion $(n-1) + (n-2) + \dots + 1$ mal $::$, also Aufwand $O(n^2)$.

Transformation von Reverse führt zu linearem Aufwand:

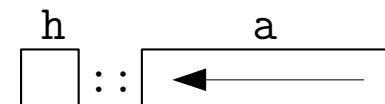
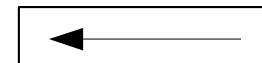
```
fun AReverse (nil, a)= a
  | AReverse (h::t,a)= AReverse (t, h::a);
fun Reverse l = AReverse (l, nil);
```

Idee:

noch umkehren



schon umkehren



Funktionen höherer Ordnung: map

Berechnungsschemata mit Funktionen als Parameter

Beispiel: eine Liste elementweise transformieren

```
fun map(f, nil) = nil  
|   map(f, h::t) = (f h) :: map (f, t);
```

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

Anwendungen von Map, z. B.

```
map (fn i => i*2.5, [1.0,2.0,3.0]);    Ergebnis:[2.5, 5.0, 7.5]  
map (fn x => (x,x), [1,2,3]);        Ergebnis: [(1,1), (2,2), (3,3)]
```

Funktionen höherer Ordnung: foldl

foldl **verknüpft Listenelemente von links nach rechts**

foldl ist mit **akkumulierendem Parameter** definiert:

```
fun foldl (f, a, nil) = a
| foldl (f, a, h::t) = foldl (f, f (a, h), t);
```

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

Für `foldl (f, 0, [1, 2, 3, 4])`
wird berechnet `f(f(f(f(0, 1), 2), 3), 4)`

Anwendungen von foldl

assoziative **Verknüpfungsfunktion** und **neutrales Element** einsetzen:

```
fun Sum l = foldl (fn (a, h:int) => a+h, 0, l);
```

Verknüpfung: **Addition**; Sum addiert Listenelemente

```
fun Reverse l = foldl (fn (a, h) => h::a, nil, l);
```

Verknüpfung: **Liste vorne verlängern**; Reverse kehrt Liste um

Polynomberechnung mit `foldl`

Ein **Polynom** $a_n x^n + \dots + a_1 x + a_0$ sei durch seine **Koeffizientenliste** $[a_n, \dots, a_1, a_0]$ dargestellt

Berechnung eines Polynomwertes an der Stelle x nach dem Horner-Schema:

$$(\dots((0 * x + a_n) * x + a_{n-1}) * x + \dots + a_1) * x + a_0$$

Funktion `Horner` berechnet den Polynomwert für x nach dem Horner-Schema:

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

Verknüpfungsfunktion `fn(a, h)=>a*x+h` hat freie Variable `x`,
sie ist gebunden als Parameter von `Horner`

Aufrufe z. B.

```
Horner [1.0, 2.0, 3.0], 10.0;
```

```
Horner [1.0, 2.0, 3.0], 2.0;
```

Funktionen höherer Ordnung

Einfaches Beispiel für Funktion als **Ergebnis**:

```
fun Choice true  = (fn x => x + 1)
|   Choice false = (fn x => x * 2);
```

Signatur Choice: `bool -> (int -> int)`

Meist sind **freie Variable** der **Ergebnisfunktion** an Parameterwerte der **konstruierenden Funktion** gebunden:

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

 Hintereinanderausführung von g und f

Signatur Comp: `('b->'c * 'a->'b) -> ('a->'c)`

Anwendung: z. B. Bildung einer benannten Funktion Hoch4

```
val Hoch4 = Comp (Sqr, Sqr);
```


Currying

Currying: Eine Funktion mit Parametertupel wird umgeformt in eine Funktion mit einfachem Parameter und einer **Ergebnisfunktion**; z. B. schrittweise Bindung der Parameter:

	Parametertupel	Curry-Form
	<code>fun Add (x, y:int) = x + y;</code>	<code>fun CAdd x = fn y:int => x + y;</code>
Signatur	<code>Add: (int * int) -> int</code>	<code>CAdd: int -> (int -> int)</code>

In Aufrufen müssen alle Parameter(komponenten) schrittweise sofort angegeben werden

`Add (3, 5)`

können die Parameter gebunden werden:

`(CAdd 3) 5`

Auch rekursiv:

```
fun CPower n = fn b =>
  if n = 0 then 1.0 else b * CPower (n-1) b;
```

Signatur CPower: `int -> (real -> real)`

Anwendung:

```
val Hoch3 = CPower 3;
(Hoch3 4)      liefert 64
((CPower 3) 4) liefert 64
```

eine Funktion, die „hoch 3“ berechnet

Kurzschreibweise: Funktionen in Curry-Form

Langform:

```
fun CPower n = fn b =>  
    if n = 0 then 1.0 else b * CPower (n-1) b;
```

Signatur CPower: `int -> (real -> real)`

Kurzschreibweise für Funktion in Curry-Form:

```
fun CPower n b =  
    if n = 0 then 1.0 else b * CPower (n-1) b;
```

Funktion Horner berechnet den Polynomwert für x nach dem Horner-Schema, in Tupelform:

```
fun Horner (koeff, x:real) = foldl (fn(a, h)=>a*x+h, 0.0, koeff);
```

Horner-Funktion in Curry-Form:

CHorner liefert eine Funktion; die Koeffizientenliste ist darin gebunden:

```
fun CHorner koeff x:real = foldl (fn(a, h)=>a*x+h, 0.0, koeff);
```

Signatur CHorner: `(real list) -> (real -> real)`

Aufruf: `val MyPoly = CHorner [1.0, 2.0, 3.0];`

```
...  
MyPoly 10.0
```

Zusammenfassung

Mit den Vorlesungen und Übungen zu Kapitel 7 sollen Sie nun Folgendes können:

- Funktionale Programme unter Verwendung treffender Begriffe präzise erklären
- Funktionen in einfacher Notation von SML lesen und schreiben
- Rekursionsparadigmen Induktion, Rekursion über Listen anwenden
- End-Rekursion erkennen und Programmiertechnik „akkumulierender Parameter“ anwenden
- Berechnungsschemata mit Funktionen als Parameter anwenden
- Programmiertechnik „Currying“ verstehen und anwenden

Danke für Eure Aufmerksamkeit!