

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          l 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);
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 401

Ziele:

Erinnerung an die Grundlagen

in der Vorlesung:

Die Funktionen werden kurz erläutert.

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);
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 402

Ziele:

Rekursionsparadigma und Varianten

in der Vorlesung:

Erinnerung an das Rekursionsparadigma und Erläuterung der Varianten.

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;

```

Vorlesung Funktionale Programmierung SS 2013 / Folie 403

Ziele:

Direkt mit akkumulierendem Parameter programmieren

in der Vorlesung:

An den Beispielen

- wird an das Prinzip erinnert und
- die Technik erläutert.

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;
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 404

Ziele:

Listen verknüpfen lernen

in der Vorlesung:

Die Funktionen werden an Beispielen erläutert:

- strukturierte Listen in flache Listen,
- Paar-Listen als Relationen,
- Reihenfolge der Muster,
- zwei akkumulierende Parameter,
- Reihenfolge der Elemente.

Liste aller Lösungen am Beispiel: Münzwechsel (1)

geg.: Liste verfügbarer Münzwerte und auszahlender 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
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 405

Ziele:

Beispielaufgabe verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- die Aufgabe,
- die Repräsentation von Münzsystemen,
- das Greedy-Verfahren

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
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 407

Ziele:

Funktion verstehen

in der Vorlesung:

An Beispielen wird erläutert:

- Funktion spannt Lösungsraum vollständig auf.
- Liste aller Lösungen

Matrix-Operationen mit Listen: Transponieren

```

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

```

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

$$\begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

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

```

Vorlesung Funktionale Programmierung SS 2013 / Folie 408

Ziele:

Matrizen als Listen von Listen darstellen

in der Vorlesung:

An Beispielen wird erläutert:

- Listen-Darstellung,
- unvollständige Muster,
- Rekursionsprinzip

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);
```

$$\left(\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \end{array} \right) \cdot (| | | |)$$

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

$$\left(\text{---} \right) \cdot (| | | |)$$

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

$$\text{---} \cdot |$$

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
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 409

Ziele:

Schrittweise Zerlegung

in der Vorlesung:

Am Beispiel wird erläutert:

- konsequente schrittweise Zerlegung der Aufgabe.

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);
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 410

Ziele:

Eindeutige Repräsentationen

in der Vorlesung:

An Beispielen wird erläutert:

- Polynomrepräsentationen,
- Problem: real-Vergleich mit 0.0 ist nicht immer exakt,
- Fallunterscheidung,
- Aufwand

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
```

Vorlesung Funktionale Programmierung SS 2013 / Folie 411

Ziele:

Halbierungstechnik zur Aufwandsreduktion

in der Vorlesung:

Am Beispiel der Polynommultiplikation wird die Halbierungstechnik erläutert.