Inhalt Kapitel 5: Funktionen höherer Ordnung

Funktionen als Argumente: map

Anwenden einer Funktion auf alle Einträge einer Liste:

```
- fun map(f, []) = []
  | map(f, kopf::rumpf) = f(kopf) :: map(f, rumpf);
val map = fn : ('a -> 'b) * 'a list -> 'b list
- fun double s = s ^ s
map(double, ["eins", "zwei", "drei"]);
val it = ["einseins","zweizwei","dreidrei"] : string list
- map(fn x -> x*x, [4,5,6,7]);
val it = [16,25,36,49] : int list
```

Funktionen als Argumente: maximum

Maximum einer Funktion in einem gegebenen Bereich

```
 fun maximum(f,a,b) = if a=b then f(a) else
   let val m = maximum(f,a,b-1) in
      if m<=f(b) then f(b) else m end;
val maximum = fn : (int -> int) * int * int -> int
- maximum(fn x=>10-(x-5)*(x-5),0,10);
val it = 10 : int
- maximum(fn x=>10-(x-5)*(x-5),0,100);
val it = 10 : int
- maximum(fn x=>10-(x-5)*(x-5),0,1);
val it = ~6 : int
```

Übungen

- Die ersten n Funktionswerte einer gegebenen Funktion addieren: $f \mapsto \sum_{i=1}^{n} f(n)$;
- Finden einer Nullstelle einer Funktion $f : \mathbb{Z} \to \mathbb{Z}$ in einem geg. Bereich
- Finden eines Wertes $x \in [a, b]$, sodass $|f(x)| \leq \epsilon$, wenn $f(a) < 0$ und $f(b) > 0$.
- Berechnen eines Differenzenquotienten $\frac{f(x+h)-f(x)}{h}$

Funktionen als Wert einer Funktion

Funktionen können nicht nur als Argument, sondern auch als Rückgabewert einer Funktion erscheinen.

- $f_k(x) = kx$ "die Mal-k-Funktion"

```
- fun malk (k) = fn x=> k*x;
val malk = fn : int -> int -> int
- malk(5);
val it = fn : int -> int
- malk(5)(4);
val it = 20 : int
- val mal20 = malk(20);
val mal20 = fn : int -> int
- mal20(3);
val it = 60 : int
- mal20(4);
val it = 80 : int
-
```

Tschebyschew–Polynome

- $T_n(x)$—das n-te Tschebyschew-Polynom:

```
fun Tsch n = if n = 0 then fn x => 1.0
             else if n = 1 then fn x=> x
             else fn x => 2.0*x*Tsch (n-1)(x) - Tsch(n-2)(x);
```

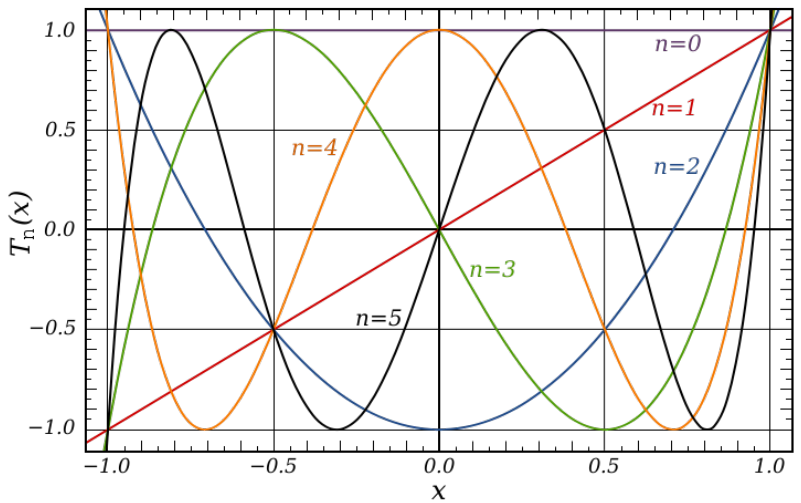

Abbildung: Die ersten 5 Tschebyschew Polynome aus Wikipedia

Polynome als Funktionen

- Repräsentiere $P = \sum_{i=0}^{n} a_i x^i$ als Paar (n, f), wobei
  $f(i) = \begin{cases} a_i, \text{ falls } 0 \leq i \leq n \\ 0, \text{ sonst} \end{cases}$.

  ```
  type poly = {coeff:int->real, deg:int};
  ```
- Man definiert dann Funktionen

  ```
  eval : poly * real -> real
  pluspoly : poly*poly -> polx
  timespoly : poly*poly -> polx
  ...
  ```
- Man beachte, dass diese Funktionen Funktionen als Argumente nehmen und auch Funktionen als Ergebnis zurückliefern.

Currying

- Eine zweistellige Funktion, z.B. Multiplikation, kann auf zwei Arten geschrieben werden:
  - Die übliche Art:

```
- fun times1(x,y) = x * y;
val times1 = fn : int * int -> int
```

  - Als einstellige Funktion, die eine Funktion zurückliefert:

```
- fun times2 x = fn y => x * y;
val times2 = fn : int -> int -> int
- fun times2 x y = x * y;
val times2 = fn : int -> int -> int
```

  Die zweite Definition von `times2` verwendet eine äquivalente Notation.

Currying

- Man bezeichnet `times2` als das *Currying* von `times1`.
- Die Anwendung von `times2` erfolgt so:

```
- times2 7 8;
val it = 56 : int
- times2 7;
val it = fn : int -> int
- it 8;
val it = 56 : int
```

Currying allgemein

- Aus einer zweistelligen Funktion des Typs
  `A * B -> C`
  wird durch Currying eine einstellige Funktion des Typs
  `A -> B -> C`
- Die Pfeile verstehen sich als nach rechts geklammert. Also `A -> B -> C` bedeutet `A -> (B -> C)`.
- Benannt nach Haskell Curry (1900-1982), am. Logiker, Namensgeber der Prog.sprache Haskell
- Currying und Un-currying als SML-Funktion:

```
- fun curry f = fn x => fn  y => f(x,y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
- fun uncurry f = fn (x,y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

- In der Regel werden mehrstellige Funktionen in der "gecurryten Form" angegeben.

Map für Listen

- Oft möchte man eine gegebene Funktion auf alle Einträge einer Liste anwenden:

```
- fun map f [] = []
= | map f (k::r) = f k :: map f r;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

- Es gilt z.B.

$$\texttt{map (fn x=>x*x)}\ [1,2,3,4,5] = [1,4,9,16,25]$$

- Es gilt allgemein:

$$\texttt{map f}\ [a_1,\ldots,a_n] = [\texttt{f}(a_1),\ldots,\texttt{f}(a_n)]$$

Filtern von Einträgen einer Liste

- Die Funktion `filter` sortiert aus einer Liste alle Einträge aus, die ein geg. Prädikat erfüllen:

```
- fun filter p [] = []
  | filter p (k::r) = if p k then
        k :: filter p r else filter p r
val filter = fn : ('a -> bool) -> 'a list -> 'a list
- filter (fn x=>x mod 2 = 0) [8,7,6,5,4,3,2,1];
val it = [8,6,4,2] : int list

val zahlungen =
  [("max",200),("moritz",100),("max",300),
      ("ida",400),("max",700), ("moritz",200)] : (string * i
- filter (fn (name,betrag) => name = "max") zahlungen;
val it = [("max",200),("max",300),("max",700)] :
                                    (string * int) list
```

Alle Listeneinträge zusammenbauen: foldl

```
- fun foldl f start [] = start
  | foldl f start (k::r) = foldl f (f(k,start)) r;
val foldl = fn : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
```

$$\texttt{foldl f start } [a_1, \ldots, a_n] = \texttt{f}(a_n, \ldots \texttt{f}(a_2, \texttt{f}(a_1, \texttt{start}) \ldots)$$

Beispiele

```
- foldl (fn (x,acc) => x+acc) 0 [1,3,4,100];
val it = 108 : int
- foldl (op +) 0 [1,3,4,100];
val it = 108 : int
- foldl (op +) 0 (map (fn (x,y)=> y)
                (filter (fn (x,y)=> x="max") zahlungen));
val it = 1200 : int
```

Foldr

Manchmal möchte man die Einträge einer Liste von rechts nach links verarbeiten.

```
- fun foldr f start [] = start
    | foldr f start (k::r) = f(k,foldr f start r);
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

$$\texttt{foldr f start } [a_1, \ldots, a_n] = \texttt{f}(a_1, \texttt{f}(a_2, \ldots \texttt{f}(a_{n-1}, \texttt{f}(a_n, \texttt{start}) \ldots)$$

Es gilt:

$$\texttt{foldr f start l} = \texttt{foldl f start (rev } l\texttt{)}$$

und diese Darstellung ist wg. Endrekursion manchmal effizienter.

Prüfen, ob es einen bestimmten Eintrag gibt

```
- fun exists p [] = false
    | exists p (k::r) = p k orelse exists p r;
val exists = fn : ('a -> bool) -> 'a list -> bool
 exists (fn x => x mod 2 = 0) [3,1,6,4,8,9,10];
val it = true : bool
- exists (fn x => x mod 2 = 0) [3,71,9,101];
val it = false : bool
```

Komposition

```
fun compose(f,g) = fn x=>f(g x)
```

Komposition ist vordefiniert und kann mit o infixnotiert werden.

```
fun gezahltan name = foldl (op +) 0 o
                     map (fn(x,y)=>y) o
                     filter (fn(x,y)=>x=name)
- gezahltan "max" zahlungen;
val it = 1200 : int
- gezahltan "moritz" zahlungen;
val it = 300 : int
```

Mit den folgenden Gesetzen kann man die Erzeugung von Zwischen-Datenstrukturen reduzieren:

$$\texttt{map}\ f\ \texttt{o}\ \texttt{map}\ g = \texttt{map}\ (f\ \texttt{o}\ g)$$
$$\texttt{foldl}\ f\ s\ \texttt{o}\ \texttt{map}\ g = \texttt{foldl}\ (\texttt{fn(x,y)} \Rightarrow \texttt{f}(g(\texttt{x}), \texttt{y}))\ s$$

Komposition ist assoziativ und hat die Identitätsfunktion als neutrales Element.

Die Kombinatoren S und K

Man definiert:

```
- fun S f g x = (f x) (g x);
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
- fun K x y = x;
val K = fn : 'a -> 'b -> 'a
```

Ein Funktion, die nur aus Applikationen (Anwendungen) und Abstraktionen (`fn`) aufgebaut ist, heißt Kombinator. Komposition ist ein Kombinator, wie auch die obigen `S` und `K`.

Kuriosität: Jeder Kombinator kann aus `S` und `K` durch Applikation definiert werden.

Zusammenfassung

- Funktionen können sowohl als Argument, als auch als Ergebnis auftreten
- Mehrstellige Funktionen können auch durch funktionale Ergebnistypen als einstellige aufgefasst werden (Currying)
- Beispiele höherstufiger Funktionen mit Listen und Polynomen
- Grundlegende Operatoren für Listen.
- Komposition und Kombinatoren.