Neue Muster für Listen

Auf das Muster ...

- `[]` passt die leere Liste **nil**,
- `[`p_1`;` ... `;`p_n`]` passt die Liste $[w_1; \ldots; w_n]$, wenn w_i auf p_i passt.
- p_1`::`p_2 passt $w_1 :: w_2$, wenn w_i auf p_i für $i = 1, 2$ passt.

Grundalgorithmen für Listen

```
let rec length = function
        [] -> 0
    | _::l -> 1 + length l

let rec enthalten = function
      ([],x) -> false
    | (h::t,x) -> x=h || enthalten(t,x)

let rec rev = function
        [] -> []
    | h::t -> rev t @ [h]

let hd (h::t) = h

let tl (h::t) = t

let null l = l = []
```

Effizientes Spiegeln

```
let rec rev_aux = function
     ([], acc) -> acc
   | (h::t,acc) -> rev_aux(t,h::acc)

let rev2 l = rev_aux(l,[])
```

Man erprobe `rev(mklist 10000)` und `rev2(mklist 10000)` wobei `mklist` n eine Liste der Länge n ist.

Man zeige mit dem Satz von der partiellen Korrektheit, dass gilt:

$$\texttt{rev\_aux}(l, acc) = \texttt{rev}(l) \mathbin{@} acc$$

Sortieren durch Einfügen

Eine Liste $[x_1; \ldots; x_n]$ heißt *sortiert*, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$.

```
let rec insertel = function
   (a, []) -> [a]
 | (a, h::t) -> if a <= h then a::h::t else h :: insertel(a,t)

let rec inssort = function
      [] -> []
   | h::t -> insertel(h,sort l)
```

Ist l sortiert, so auch `insertel`(a,l) und es enthält dieselben Elemente wie $a :: l$.

Für beliebiges l ist `inssort`(l) sortiert und enthält dieselben Elemente wie l.

Anwenden einer Funktion auf alle Elemente einer Liste

```
let rec map f l = match l with
        [] -> []
      | h::t -> f h :: map f t
```

Beispiel:

```
# map (fun x -> x * x) [1;2;3;4];;
- : int list = [1; 4; 9; 16]
# map string_of_int [1;2;3;4];;
- : string list = ["1"; "2"; "3"; "4"]
# map (map (fun x -> x*x)) [[1;2]; [3]; [4;5;6]];;
- : int list list = [[1; 4]; [9]; [16; 25; 36]]
```

Weitere Funktionen als Übung

Zerlegen in zwei Teile

```
val split : ('a - > bool) -> 'a list -> 'a list * 'a list
```

Verflachen

```
val flatten : 'a list list -> 'a list
```

Falten

```
val fold_right : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

`fold_right` $f\ x\ [x_1; \ldots; x_n] = f\ x_1 (f\ x_2\ \ldots (f\ x_n\ x)) \ldots)$

Beispiel

Kontovorgänge liegen als Liste von floats vor.

Buchungsbetrag $\leq$ 1000EUR : Gebühr 30 Cent

Buchungsbetrag $>$ 1000EUR : Gebühr 50 Cent

Man schreibe eine Funktion, die die Gesamtgebühr bestimmt.

Zunächst direkt, dann mit `fold_right`.

Weitere Anwendungsbeispiele in [Kröger] und der OCAML Bibliothek.

Stapel

Stapel (Keller, *stack*): Endliche Listen mit Basisfunktionen **nil**, **cons**, **hd**, **tl**, **null** (hier meist bezeichnet als **empty**, **push**, **top**, **pop**, **isempty**.

Idee: Man legt Dinge auf dem Stapel ab und bekommt sie in der Reihenfolge "*last-in-first-out*" (LIFO) zurück.

Der Stapel ist eine häufig verwendete Datenstruktur. Man kann mit ihm z.B. Rekursion simulieren (wird hier nicht behandelt).

Schlange

Schlangen: Endliche Folgen mit den Basisfunktionen **nil**, **hd**, **tl**, sowie **enter** mit Bedeutung **enter**$(l, x) = l@[x]$.

Idee: Dinge werden in die Schlange eingereiht und in der Reihenfolge *first-in-first-out* (FIFO) wieder entnommen.

Schlangen werden sowohl als echte Warteschlangen, als auch als Hilfsdatenstruktur in bestimmten Algorithmen verwendet.

Bemerkung: Die naive Implementierung von Schlangen als Listen ist ineffizient, da die Basisfunktion **enter** Zeit proportional zur Länge der Schlange verbraucht.

Reihungen

Reihungen (Vektoren, *Arrays*): Endliche Folgen mit den Basisfunktionen **init**, **dim**, **get**, **update** mit den Bedeutungen

$$\textbf{init}(n, x) = \underbrace{[x; x; \ldots; x]}_{n \text{ Einträge}}$$

$$\textbf{dim}([x_1; \ldots; x_n]) = n$$

$$\textbf{get}([x_1; \ldots; x_n], i) = x_i$$

$$\textbf{update}([x_1; \ldots; x_n], i, x) = [x_1; \ldots; x_{i-1}; x; x_{i+1}; \ldots; x_n]$$

Im Falle unpassender Indizes sind diese Operationen undefiniert, z.B.: **get**$([x_1; x_2], 3)$ oder **init**$(-2, 0.0)$.

Wir schreiben α **vect** für den Typ der Reihungen mit Einträgen vom Typ α.

Man kann Reihungen als OCAML-Listen implementieren; bei großer Dimension ist das aber ineffizient.

Binäre Suche mit Reihungen

Die folgenden Funktionen setzen voraus, dass die Reihung l aufsteigend sortiert ist.

$$\begin{array}{l}
\text{sucheVonBis} = \textbf{function}(l\text{:}\textbf{string vect}, w\text{:}\textbf{string}, i\text{:}\textbf{int}, j\text{:}\textbf{int})\textbf{bool} \\
\quad \textbf{if } i > j \textbf{ then false else} \\
\quad \textbf{if } i = j \textbf{ then get}(l,i) = w \textbf{ else} \\
\quad\quad \textbf{let } m = \lfloor (i+j)/2 \rfloor \textbf{ in} \\
\quad\quad \textbf{let } w_m = \textbf{get}(l,m) \textbf{ in} \\
\quad\quad \textbf{if } w_m = w \textbf{ then true else} \\
\quad\quad\quad \textbf{if } w < \textbf{get}(l,m) \textbf{ then} \\
\quad\quad\quad\quad \text{sucheVonBis}(l,w,i,m-1) \\
\quad\quad\quad\quad \textbf{else } \text{sucheVonBis}(l,w,m+1,j) \\
\text{suche} = \textbf{function}(l\text{:}\textbf{string vect}, w\text{:}\textbf{string})\text{sucheVonBis}(l,w,1,\textbf{dim}(l))
\end{array}$$

Binärbäume

Definition: Sei A eine Menge. Die Menge $A^\triangle$ der *Binärbäume über A* ist induktiv definiert wie folgt:

1. $A^\triangle$ enthält den *leeren Binärbaum* τ
2. Sind x in A und $l, r \in A^\triangle$, so ist das 3-Tupel $(x, l, r) \in A^\triangle$

Man kann die induktive Definition wie folgt umgehen: Die Menge $A^\triangle_n$ der *Höhe höchstens n* ist rekursiv (über n) definiert durch

1. $A^\triangle_0 = \{\tau\}$
2. $A^\triangle_{n+1} = A^\triangle_n \cup \{(x, l, r) \mid x \in A, l \in A^\triangle_n, r \in A^\triangle_n\}$.

Es ist dann $A^\triangle = \bigcup_{n \geq 0} A^\triangle_n$. Die *Höhe* eines Binärbaumes $t \in A^\triangle$ ist das kleinste n, sodass $t \in A^\triangle_n$.

Terminologie

x heißt *Wurzel* oder *Wurzelbeschriftung* von (x, l, r).

l, r heißen *linker, bzw. rechter Unterbaum* von (x, l, r). Ein Binärbaum der Form (x, τ, τ) heißt *Blatt*. Ein von τ verschiedener Binärbaum heißt *nichtleer*.

Die *Knoten* und *Teilbäume* eines Binärbaums sind rekursiv definiert wie folgt:

1. τ hat keine Knoten und nur τ als Teilbaum.
2. Die Knoten von (x, l, r) sind x und die Knoten von l und die Knoten von r. Die Teilbäume von (x, l, r) sind (x, l, r) und die Teilbäume von l und von r.

Beispiel

$t = (6, (3, (2, \tau, \tau), (8, \tau, (5, \tau, \tau))), (8, (4, \tau, \tau), \tau))$.

Knoten von t: $\{6, 3, 2, 8, 5, 4\}$.

Teilbäume von t:
$\{t, (3, (2, \tau, \tau), (8, \tau, (5, \tau, \tau))), (2, \tau, \tau), (8, \tau, (5, \tau, \tau)), (5, \tau, \tau), (8, (4, \tau, \tau), \tau)), (4, \tau,$