Funktionale Programmierung SS 2013 - Lösung 3

Prof. Dr. U. Kastens
Institut für Informatik, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn
13.05.2013

Lösung zu Aufgabe 1

Lösung mit Hilfsfunktionen zur Behandlung der Listenanfänge:

```
(* Version 1 mit Hilfsfunktionen *)
(* entferne am Listenanfang alle Elemente, die gleich sind *)
fun skip (nil) = nil
|   skip (a :: nil) = nil
|   skip (a::b::t) = if a = b then skip(b::t)
                              else b::t;

(* zähle wie viele Elemente am Listenanfang gleich sind *)
fun eqstart(nil) = 0
|   eqstart(a::nil) = 1
|   eqstart(a::b::t) =  if a = b then 1 + eqstart(b::t)
                                 else 1;

fun rlencode (nil) = []
|   rlencode (h::t) = (h, eqstart(h::t)) :: rlencode(skip(h::t));
```

Lösung mit Hilfsparameter zum Zählen:

```
(* Version 2 mit Hilfsparameter c *)
(* Bedeutung von c: das erste Listenelement haben wir links vom aktuellen
   Listenanfang schon c-mal hintereinander gesehen *)
fun rlec(nil,_) = nil
|   rlec(h::nil,c) = [(h,c+1)]
|   rlec(h::i::t,c) = if h = i then rlec(i::t, c+1)
                               else (h,c+1)::rlec(i::t,0)
;
fun rlencode2 l = rlec(l,0);
```

Vergleich der Lösungen:

Vergleichskriterien Laufzeit-Komplexität und Verständlichkeit des Programm-Codes. Version 1 könnte durch die Strukturierung in Teilaufgaben verständlicher sein, Version 2 ist effizienter, da jedes Element der Liste nur einmal berührt wird.

Bitte vergleichen Sie obige Lösungen auch mit denen, die Sie in der Übungsstunde entwickelt haben. Verwendet Ihre Lösung mehr Parameter? Warum?

Die Funktion zum Expandieren:

```
fun rlexpand [] = []
|   rlexpand ((elem,1)::t) = elem :: rlexpand(t)
|   rlexpand ((elem,c)::t) = elem :: rlexpand((elem, c-1)::t);
```

Lösung zu Aufgabe 2

Die Funktion `filecount`:

```
fun filecount (file(_)) = 1
|   filecount (dir(_,nil)) = 0
|   filecount (dir(n,h::t)) = filecount h + filecount (dir(n, t))
;
```

Lösung zu Aufgabe 3

a) Die Datentypdefinition:

```
datatype oper = plus | minus | times | divide;
datatype kantotree = leaf of int| inner of kantotree * oper * kantotree;
```

b) Die Funktion `evaluate`:

```
fun evaluate (leaf(v)) = v
|   evaluate (inner(t1, plus, t2)) = evaluate(t1) + evaluate(t2)
|   evaluate (inner(t1, minus, t2)) = evaluate(t1) - evaluate(t2)
|   evaluate (inner(t1, times, t2)) = evaluate(t1) * evaluate(t2)
|   evaluate (inner(t1, divide, t2)) = evaluate(t1) div evaluate(t2)
;
```

c) Die polymorphe Version:

```
datatype 'a kantotree = leaf of 'a
                      | inner of ('a kantotree) * oper * ('a kantotree);
```

Die Definitionen von `t1`, `t2`, `t3` und `evaluate` sind noch korrekt. `evaluate` hat nun die Signatur

```
fn : int kantotree -> int
```

.