Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 4

(Kapitel 4)

Hinweis: Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.

Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant, noch irrelevant.

1 Listen - zum Aufwärmen

Aufgabe 4.1
Stellen Sie die folgenden Listen nur mit *nil* und :: dar.

(a) $[1, 2, 3, 4, 5]$

(b) $[[1, 2, 3], [4, 5], [1]]$

(c) $[[]]$

Lösung 4.1:

(a) $1 :: 2 :: 3 :: 4 :: 5 :: nil$

(b) $(1 :: 2 :: 3 :: nil) :: (4 :: 5 :: nil) :: (1 :: nil) :: nil$

(c) $(nil :: nil) :: nil$

Aufgabe 4.2
Stellen Sie die folgenden Listen mit möglichst wenig *nil*, :: und @ dar:

(a) $[true, true, true, false]$

(b) $[[1], [3], [3], [4]]$

(c) $[[[1, 2], [3, 4]], [[1, 2], [3, 4]]]$

Lösung 4.2:

(a) $[true, true]@[true, false]$

(b) $[[1], [3]]@[[3], [4]]$

(c) $[[[1, 2], [3, 4]]]@[[[1, 2], [3, 4]]]$

Aufgabe 4.3
Schreiben Sie mit Hilfe von *exists* und $forall$ eine Prozedur, die ...

(a) ... testet, ob in einer Liste eine Zahl existiert, die kleiner als 42 ist.

(b) ... testet, ob alle Zahlen durch 5 teilbar sind.

(c) ... testet, ob in einer Liste von Paaren des Typs $int * int$ für jedes Paar gilt: die erste Komponenten ist ein Teiler der zweiten

Lösung 4.3:

(a) `fun smallerThan42 xs = exists (fn x => x < 42) xs`

(b) `fun divisibleBy5 xs = all (fn x => x mod 5 = 0) xs`

(c) `fun divisibleTuple xs = all (fn (a,b) => b mod a = 0) xs`

Aufgabe 4.4
Schreiben Sie eine Prozedur, die eine Liste des Typs *int list list* in eine Liste des Typs *int list* überführt, die die gleichen Zahlen enthält, diese aber nicht in der gleichen Reihenfolge wie zuvor in der Liste stehen. Wie Sie dabei die einelementige Liste behandeln, ist Ihnen überlassen.

Lösung 4.4:

```
fun concat xs = foldl (fn(x,s) => x @ s) nil xs
```

Aufgabe 4.5
Schreiben Sie eine Prozedur, die in einer Liste des Typs *int list list* alle Zahlenwerte verdoppelt.

Lösung 4.5:

```
fun mal2 xs = foldr (fn(x,s) => (map (fn a => 2*a) x)::s) nil xs
```

Aufgabe 4.6
Schreiben Sie mit Hilfe eines case-Ausdrucks eine Prozedur, bei Eingabe der leeren Liste eine beliebige einelementige Liste zurückgibt, bei Eingabe einer einelementigen Liste eine leere Liste und ansonsten die Liste selbst wieder ausgibt.

Schreiben Sie diese Prozedur nun regelbasiert!

Lösung 4.6:

```
fun p [] = [3]
| p [_] = []
| p xs = xs
```

Oder auch:

```
fun p nil = [3]
| p (x::nil) = []
| p xs = xs
```

2 foldl, foldr

Aufgabe 4.7 (*Faltung ohne foldl*)
Betrachten Sie das folgende Beispiel "Kleider anziehen" zur Faltung:

(a) Als Startwert betrachten wir den Menschen, der die Kleidung anzieht.

(b) Die Liste enthält die Kleidung in der Reihenfolge, wie die Kleidung angezogen werden soll.

(c) Unsere Prozedur soll das Überziehen von Kleidung sein.

(d) Verwenden wir nun $foldl$ (im übertragenen Sinne), so erhalten wir einen fertig angezogenen Menschen.

Welche anderen alltäglichen Beispiele für Faltung (mit $foldl$ oder $foldr$) können Sie finden?

Aufgabe 4.8
Schreiben Sie mit Hilfe von Faltung (ob Sie $foldl$ oder $foldr$ verwenden, bleibt Ihnen überlassen) eine Prozedur, die testet, ob alle Wahrheitswerte in einer Liste vom Typ *bool list* den Wert *true* haben.

Lösung 4.8:

```
fun test xs = foldl (fn (x,s) => s andalso x) true xs
```

Aufgabe 4.9
Die Ähnlichkeit zweier Listen l_1 und l_2 ergibt sich als die Anzahl der Listenpositionen, an denen beide Listen die gleiche Komponente enthalten, geteilt durch die Länge der Listen.

(a) Schreiben Sie eine Prozedur *similar*, die die Ähnlichkeit zweier Listen zurückgibt. Falls $|l_1| \neq |l_2|$ soll die Ausnahme *Subscript* geworfen werden. Wenn Sie möchten, können Sie folgende Vorlage verwenden:

```
fun ''a similar xs (ys:''a list) =
let
    val (equals : real, rest : ''a list, total : real) = foldl ...
in
    ...
end
```

(b) Geben Sie den Typ ihrer Prozedur an!

Lösung 4.9:

(a)
```
fun ''a similar xs (ys:''a list) =
let
   val (equals : real, rest : ''a list, total : real) = foldl (fn (x, (equals, (y::yr), total))

                                => (equals + (if x = y then 1.0 else 0.0), yr , total + 1.0)

                                      | (_, (_, nil,_)) => raise Subscript) (0.0,ys, 0.0) xs

in
   if rest <> nil then raise Subscript
   else equals / total
end
```

(b) $similar : {'\alpha}\ list \to {'\alpha}\ list \to real$

Wegen der Verwendung des Gleichheitstests sind die Listen auf Typen mit Gleichheit eingeschränkt!

3 foldl

Aufgabe 4.10
Können Sie $foldl$ mit $foldr$ darstellen und umgekehrt? Wenn ja, geben Sie eine Implementierung an, falls nicht, begründen Sie Ihre Antwort.

Aufgabe 4.11
Betrachten wir die Hilfsprozeduren, die wir in der letzten Woche kennengelernt haben:

- *iter*
- $first$
- *iterup*
- *iterdn*

Mit welchen diesen Prozeduren können Sie $foldl$ implementieren? Geben Sie, wenn ja eine Beispielimplementierung an oder begründen Sie, warum es nicht möglich nicht.

Können Sie umgekehrt diese Prozeduren mit $foldl$ implementieren? Geben Sie auch hier, wenn ja eine Beispielimplementierung an oder begründen Sie, warum es nicht möglich nicht.

Aufgabe 4.12 (*linkshändig*)

(a) Vergleichen Sie die Deklarationen von $foldl$ und $foldr$. Welche der beiden Prozeduren ist endrekursiv? Können Sie die nicht endrekursive Prozedur in eine endrekursive umschreiben?

(b) Welche Beobachtungen machen Sie bei der Verwendung von $foldl$ und $foldr$? Wo liegen die Unterschiede? Wo die Gemeinsamkeiten?

4 Muster

Aufgabe 4.13 (*Muster hoch zwei*)
Betrachten Sie diese Muster:

(a) `x`
(b) `_`
(c) `(_,x)`
(d) `[x,y]`
(e) `(x,y,z)`
(f) `(x,x,y)`

Welche der folgenden Werte treffen diese Muster? Welche Bindungen werden dabei erzeugt?

(a) `42`
(b) `(1,2)`
(c) `[1,2]`
(d) `(1,2,3)`
(e) `(42,42,42)`
(f) `[1,2,3]`
(g) `(1,(2,3),3)`
(h) `[[1,2],[3,4]]`

Lösung 4.13:

(a) x trifft alle Werte, x wird immer an den gesamten Ausdruck gebunden.

(b) _ trifft auch alle Werte

(c) $(\_, x)$ trifft nur (b). Allgemein trifft dieses Muster alle 2-Tupel (Paare).
Die Bindung, die entsteht, ist $x := 2$.

(d) $[x, y]$ Dieses Muster beschreibt alle zweielementigen Listen, also (c) und (h).
Bei (c) sind die Bindungen $x := 1$ und $y := 2$, bei (h) sind es $x := [1, 2]$ und $y := [3, 4]$.

(e) (x, y, z) trifft (d), (e), (g)
Bindungen:
(d): $x := 1, y := 2, z := 3$
(e): $x := 42, y := 42, z := 42$
(g): $x := 1, y := (2, 3), z := 3$

(f) (x, x, y) darf man so in einem Muster leider nicht verwendet werden. Der Interpreter wirft dann die Ausnahme:

```
duplicate variable 'x' in pattern or binding group
```

Man kann so leider nicht ausdrücken, dass zwei gleiche Bezeichner im Muster vorkommen sollen, dazu hat man nur die Möglichkeit die Werte explizit anzugeben (z.b. $(42, 42, x)$).

Aufgabe 4.14
Können Sie jede regelbasierte Prozedur in eine Prozedur umschreiben, die keine Regeln, aber einen case-Ausdruck verwendet?

5 Strings

Aufgabe 4.15
Was halten Sie von folgender Aussage: "Mit Hilfe von *implode* und *explode* kann man alle Prozeduren, die man auf Listen verwenden kann, auch auf Strings anwenden."

Aufgabe 4.16
Überlegen Sie sich eine Aufgabe zum Thema "Strings", von der Sie überzeugt sind, dass Sie beim Verständnis des zentralen Punkts hilft. Lösen Sie die Aufgabe zunächst selbst und lassen Sie dann Ihren Nachbarn die Aufgabe lösen.

6 Knobelaufgaben

Aufgabe 4.17 (*Dynamische Bindungen*)
Dieter Schlau a.k.a. *The Amazing One* hat die Funktion $n!$ mit Hilfe von iter implementiert:

```
fun fac (n:int) : int = iter n 1 (fn (a:int) => a * n)
```

Warum tut diese Lösung nicht das, was sie soll?
Hinweis: Betrachten Sie einen hinreichend großen Abschnitt eines Ausführungsprotokolls!

Lösung 4.17:
Der Bezeichner n wird beim Aufruf der Prozedur genau einmal gebunden und diese Bindung wird nie wieder verändert. Deshalb wird der Akku immer mit dem selben Wert multipliziert, was zur Berechnung der Funktion n^n statt $n!$ führt.

Aufgabe 4.18 (*A wild Dieter Schlau appears*)
Was bisher geschah: Dieter Schlau hat Ihren Interpreter manipuliert und Rekursion unbenutzbar gemacht, nur die rekursive Prozedur *fortytwo* ist Ihnen geblieben:

```
fun fortytwo (f : 'a -> ('a * bool)) (s : 'a) = let val (n,b) = f s in if b then n else fortytwo
f n end
```

Glücklicherweise ist es Ihnen damit aber gelungen, Ihre Lieblingsprozeduren *iter*, *first*, *iterup* und *iterdn* erneut zu deklarieren.

(a) Sie haben in der letzten Woche viele wichtige Prozeduren kennengelernt, allerdings konnten Sie die vorgegebenen Deklarationen nicht übernehmen, da Ihr Interpreter ja (fast) keine Rekursion mehr versteht.

Glücklicherweise sind Sie ja noch in Übung! Deklarieren Sie *foldl* mit *fortytwo*, ohne zusätzliche Rekursion zu benutzen.

(b) Endlich können Sie sich in Ihrem Interpreter wieder wohlfühlen. Sie lehnen sich entspannt zurück, als plötzlich...

Sie sehen einen Blitz durchs Zimmer huschen und bevor Sie verstehen, was los ist, hören Sie schon das Lachen von Dieter Schlau.

Entsetzt blicken Sie in Richtung Ihres Interpreters und stellen fest, dass Dieter alle Prozeduren, deren Typschema *list* enthält, gelöscht hat.

Versuchen Sie nun *hd* und *tl* erneut zu implementieren!

(c) Sie versuchen *hd* und *tl* regelbasiert zu schreiben. Doch entsetzt stellen Sie fest, dass der Interpreter Ihre Definition ablehnt:

```
Lexical error: ill-formed token.
```

Nicht nur das: Sie stellen fest, dass ihr Interpreter scheinbar jede Form von regelbasierten Prozeduren ablehnt. Sie können also als Argument von Prozeduren nur noch Bezeichner angeben, keine Muster.

Gibt es noch Rettung? Können Sie unter diesen Umständen die Prozeduren *hd* und *tl* noch deklarieren? Wenn ja, wie? Wenn nein, warum nicht?

(d) Es kommt noch schlimmer: Dieter Schlau hat bei seinem zweiten Besuch auch ihre geliebte Prozedur *fortytwo* gelöscht.

Retten Sie Ihren Interpreter! Deklarieren Sie eine zu *fortytwo* semantisch äquivalente Prozedur. Nutzen Sie nur zwei der folgenden Hilfsprozeduren: *iter*, *iterup*, *iterdn*, *first*. Natürlich können Sie keine weitere Rekursion benutzen.

Lösung 4.18:

(a) `fun foldl f s xs = #2 (fortytwo (fn (nil,s) => ((nil,s),true) | (x::xr,s) => ((xr,f(x,s)), false)) (xs,s))`

(b) `fun hd (x::xr) = x fun tl (x::xr) = xr`

(c) Nein, jede Hoffnung ist verloren. Es gibt in SML keine Möglichkeit, auf den Inhalt von Listen zuzugreifen, ohne Pattern Matching zu benutzen.

(d) `fun fortytwo (f : 'a -> ('a * bool)) (s : 'a) = iter (first 0 (fn n => #2 (iter n (s, false) (fn (n,b) => (f n)) ) ) ) s (fn s => #1 (f s))`