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

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 3

(Kapitel 3)

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 Höherstufigkeit, Kaskadierung

Aufgabe 3.1
Schreiben Sie eine höherstufige Prozedur, die nicht kaskadiert ist.

Lösung 3.1:

```
fun f (x:int -> int) :int = 4
```

Aufgabe 3.2
Schreiben Sie eine kaskadierte Prozedur, die nicht höherstufig ist.

Lösung 3.2:

```
fun f x y = x + y
```

Aufgabe 3.3
Geben Sie Beispiele für geschlossene und offene Abstraktionen an.

Wie ergeben sich die Typen Ihrer Abstraktionen? Welche der Argumente werden polymorph getypt?

Lösung 3.3:

- geschlossen:

```
(fn x=> x + 4)
(fn (x,y) => if y then x * x else x)
```

- offen:

```
(fn x => x + a) (fn (x,y,z) => x * y + z + a)
```

2 iter, first

Aufgabe 3.4 (*Zum Aufwärmen*)
Deklarieren Sie eine Prozedur $roundup : int- > int$, die eine gegebene Zahl auf die nächstgrößere durch 10 teilbare Zahl aufrundet. Benutzen Sie $first$, aber keine explizite Rekursion!

Lösung 3.4:

```
fun roundup s = first s (fn x => (x mod 10) = 0)
```

Aufgabe 3.5
Überlegen Sie sich mindestens zwei mathematische Aufgabenstellungen (z.B. Multiplikation durch wiederholte Addition, Quadrieren einer gegebenen reellen oder natürlichen Zahl,...), die sich mit *iter*, $first$, *iterup* oder *iterdn* als Hilfsprozedur elegant lösen lassen. Wenn Sie eine Lösungsidee haben: Können Sie auch eine andere als die von Ihnen gewählte Hilfsprozedur zur Lösung verwenden?
Geben Sie nun die Aufgaben auch Ihrem Nachbarn und lassen Sie sie lösen!
Können Sie auch Probleme finden, die sich sowohl mit *iter* als auch mit $first$ einfach lösen lassen?

Aufgabe 3.6
Welche der folgenden Prozeduren können Sie mit Hilfe der anderen drei ohne explizite Rekursion und ohne weitere Hilfsprozedur darstellen?

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

Probieren Sie alle Möglichkeiten aus! Geben Sie jeweils eine Deklaration an *oder* begründen Sie, warum dies nicht möglich ist.

Aufgabe 3.7 (*Rateabend mit Dieter Schlau*)
Dieter Schlau hat nach langem Überlegen eine Lösung für die sum'-Aufgabe (*Aufgabe 3.9* oder *Aufgabe 3.10* im Buch) gefunden:

```
fun sum' f m k = sum f (m + k) - sum f m
```

Nun fragt Dieter Sie, ob diese Lösung korrekt ist.

(a) Was antworten Sie? (Bedenken Sie, dass Dieter nicht nur „Ja“ oder „Nein“ hören will, er wird solange Ihnen diskutieren, bis Sie ihn überzeugt haben.)

(b) Geben Sie ein konkretes Gegenbeispiel an, bei dem sich sum' nicht wie gefordert verhält.

Lösung 3.7:

(a) Die Prozedur ist so leider nicht korrekt, da sie auch $f\,x$ für $x \leq m$ berechnet. Dies kann dazu führen, dass die Prozedur divergiert.

(b)
```
fun div n = div n

Der folgende Aufruf divergiert bei der gegebenen Definition: sum' (fn x => if n <= 10 then div
x else x) 10 2
```

3 Knobelaufgaben

Aufgabe 3.8 (*Frei nach Cäsar*)
Um Telefonnummern auch über potenziell unzuverlässige Boten weitergeben zu können, kannten schon die alten Römer das Prinzip der Cäsarverschlüsselung: Das Alphabet – für Telefonnummern die Ziffern 0 bis 9 – wird einfach um eine Differenz k (Schlüssel genannt) "verschoben". Beispiel mit Schlüssel 3:

1 wird zu 4
2 wird zu 5
3 wird zu 6
7 wird zu 0
8 wird zu 1
9 wird zu 2

Eine Telefonnummer wird also wie in folgenden Beispielen verschlüsselt (Schlüssel 3):

123 wird zu 456
96894 wird zu 29127

Lassen Sie in den deklarierten Prozeduren die Typen vollständig weg!

(a) Schreiben Sie eine Prozedur $key : int \to int \to int$, die eine Ziffer wie oben gezeigt "verschiebt"!

(b) Schreiben Sie eine Prozedur $digits : int \to int$, die die Anzahl der Ziffern in einer Telefonnumer n ermittelt! Sie dürfen annehmen, dass Telefonnummern keine führenden Nullen enthalten.

(c) Schreiben Sie eine **nicht**-rekursive Prozedur $enc : (int \to int) \to int \to int$, die zu einer Telefonnummernlänge l und einem Schlüssel $k : (int \to int)$ die Verschlüsselung einer Telefonnummer n zurückgibt. Sie dürfen dabei annehmen, dass l immer der Anzahl der Ziffern von n entspricht.

Hinweis: Verwenden Sie *iter* mit einem mehrstelligen Tupel als Akkumulator!

(d) Wie können Sie *enc* verwenden, um die oben gezeigten Telefonnummern zu verschlüsseln?

(e) Für Experten: Wie können Sie *enc* verwenden, um eine Prozedur $enc : (int \to int) \to int \to int$ zu deklarieren, die eine Telefonnummer, die mit einem Schlüssel $k : (int \to int)$ verschlüsselt wurde, wieder zu entschlüsseln?

Lösung 3.8:

(a)
```
fun key k n = (n + k) mod 10
```

(b)
```
fun digits n = if n = 0 then 0 else 1 + digits(n div 10)
```

(c)
```
fun enc k n = # 1 (iter (digits n) (0, n, 1) (fn (e, r, w) => (e + w * k(r mod 10), r div 10,
10 * w)))
```

(d)
```
enc (key 3) 123
enc (key 3) 96894
```

(e)
```
fun reverse k = key (~(k 0))
fun dec k = enc (reverse k)
```

Aufgabe 3.9 (*Vorsicht, scharf!*)
Oh nein! Dieter Schlau ist an Ihren Computer gekommen und hat natürlich die falschen Knöpfe gedrückt. Nachdem Dieter alle bisher deklarierten Prozeduren gelöscht hatte (ja, auch Ihre geliebten Prozeduren *iter*, *first*, *iterup* und *iterdn* sind alle weg), konnten Sie gerade noch

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

in Ihren Interpreter eingeben, bevor Dieters Aktionen das Schlüsselwort *fun* permanent unbenutzbar gemacht hat.

Hinweis: Falls in Ihrem Buch *val rec* erwähnt wird: Das ist leider auch nicht mehr benutzbar.
Wenn Sie von diesem Schlüsselwort noch nie gehört haben, ignorieren Sie es einfach.

Da Dieter leider entkommen konnte, müssen Sie die folgenden Aufgaben leider ohne *fun* und ohne jegliche Hilfsprozedur außer *fortytwo* lösen. Sie dürfen sich auch keine weiteren Hilfsprozeduren deklarieren!

(a) Nachdem Sie realisiert haben, dass *fortytwo* nun Ihre einzige Möglichkeit ist, Rekursion zu benutzen, betrachten Sie diese Prozedur genauer. Was ist ihr Typschema? Was tut sie? Wie kann Sie Ihnen helfen, rekursive Prozeduren zu schreiben? Was tut sie für die Parameter

`(fn x => (x,true)) und 7`

Geben Sie ein verkürztes Ausführungsprotokoll an.

(b) Machen Sie sich mit *fortytwo* vertraut: Geben Sie eine Deklaration an, die den Bezeichner x an den Wert 10! bindet. Lassen Sie den Wert mit *fortytwo* berechnen.

(c) Lösen Sie die Aufgaben 1.13 und 1.17 aus dem Buch mit den gegebenen Einschränkungen.

(d) Richten Sie sich Ihren Interpreter wieder gemütlich ein! Deklarieren Sie sich ihre Lieblingsprozeduren *iter*, *first*, *iterup* und *iterdn* erneut.

(e) Finden Sie eine rekursive Prozedur, die sich Ihrer Meinung nach nur schwer mit *fortytwo* realisieren lässt. Geben Sie diese Prozedur Ihrem Partner, und lassen Sie ihn diese Prozedur umschreiben.

(f) Gibt es rekursive Prozeduren, die sich gar nicht *fortytwo* darstellen lassen? Begründen Sie und geben Sie gegebenenfalls ein Beispiel an.

Lösung 3.9:

(a) *fortytwo* ruft eine Abstraktion - ähnlich wie *iter* - wiederholt auf. Allerdings gibt die Abstraktion hier einen zusätzlichen booleschen Wert aus, der angibt, ob die Berechnung beendet ist, oder ob erneut in Rekursion gegangen werden soll.

`('a -> 'a * bool) -> 'a -> 'a`

(b) `val x = #2 (fortytwo (fn (n,x) => if n > 0 then ((n-1,x*n),false) else ((n,x),true)) (10,1))`

(c) (i) Aufgabe 1.13:

```
val mul = fn (i1,i2) => #3 (fortytwo (fn (x,n,a) => if n > 0 then ((x,n-1,a+x),false) else
((x,n,a),true)) (i1,i2,0))
```

(ii) Aufgabe 1.17:

```
val quer = fn i => #2 (fortytwo (fn (x,a) => if x > 0 then ((x div 10,a + x mod 10),false)
else ((x,a),true)) (i,0))
```

(d) `val iter = fn n => fn s => fn f => #2 (fortytwo (fn (n,s) => if n < 1 then ((n,s),true) else ((n-1,f s),false)) (n,s))`

```
val iterup = fn m => fn n => fn s => fn f => #2 (fortytwo (fn (m,s) => if m > n then ((m,s),true)
else ((m+1,f (m,s)),false)) (m,s))

val iterdn = fn n => fn m => fn s => fn f => #2 (fortytwo (fn (n,s) => if n < m then ((n,s),true)
else ((n-1,f (n,s)),false)) (n,s))

val first = fn s => fn p => fortytwo (fn s => if p s then (s,true) else (s+1,false)) s
```

n/a

Prozeduren, die mehrere Rekursive Aufrufe pro Ausführung benötigen, lassen sich nicht (leicht) mit *fortytwo* darstellen.