Fachschaft Informatikstudiengänge

Fachrichtung 6.2 — Informatik

Das Team der Bremser

1. Probeklausur (Lösung) zu Programmierung 1 (WS 07/08)

http://fsinfo.cs.uni-sb.de

Name Matrikelnummer

Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden.
Sie können die Klausur nur auf dem für Sie vorgesehen Platz schreiben. Sie müssen das mit Ihrem Namen und Ihrer Matrikelnummer versehene Heft verwenden.
Hilfsmittel sind nicht zugelassen. Am Arbeitsplatz dürfen nur Schreibgeräte, Getränke, Speisen und Ausweise mitgeführt werden. Taschen und Jacken müssen an den Wänden des Klausursaals zurückgelassen werden.
Das Verlassen des Saals ohne Abgabe des Klausurhefts gilt als Täuschungsversuch.
[Wenn Sie während der Bearbeitung zur Toilette müssen, geben Sie bitte Ihr Klausurheft bei der Aufsicht ab. Es kann immer nur eine Person zur Toilette.]
Alle Lösungen müssen auf den bedruckten rechten Seiten des Klausurhefts notiert werden. Die leeren linken Seiten dienen als Platz für Skizzen und werden **nicht korrigiert.** Notizpapier ist nicht zugelassen. Sie können mit Bleistift schreiben.
Für die Bearbeitung der Klausur stehen 75 [150] Minuten zur Verfügung. Insgesamt können 75 [150] Punkte erreicht werden. Die für jede Aufgabe angebene Punktzahl gibt Ihnen also einen Anhaltspunkt, wieviel Zeit Sie auf die Bearbeitung der Aufgabe verwenden sollten. Zum Bestehen der Klausur genügen 37,5 [75] Punkte.
[Bitte legen Sie zur Identifikation Ihren Personalausweis bzw. Reisepass sowie Ihren Studierendenausweis neben sich.]
Viel Erfolg!

1	2	3	4	5
15	15	14	16	15

Summe
75

Note

Deklarationen

Sie können die folgenden vordeklarierten Prozeduren benutzen:

$$
\begin{aligned}
op@ &: \alpha\ list * \alpha\ list \to \alpha\ list \\
map &: (\alpha \to \beta) \to \alpha\ list \to \beta\ list \\
foldl &: (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta \\
iter &: int \to \alpha \to (\alpha \to \alpha) \to \alpha \\
first &: int \to (int \to bool) \to int \\
List.concat &: \alpha\ list\ list \to \alpha\ list
\end{aligned}
$$

Lösung 1.1 (Sortieren, 15=12+3)

a)
```
fun split xs = foldl (fn (x,(ul,vl)) => (vl,x::ul)) (nil,nil) xs

fun smerge compare xs nil = xs
  | smerge compare nil ys = ys
  | smerge compare (x::xs) (y::ys) = case compare(x,y) of
                                       LESS => x::(smerge compare xs (y::ys))
                                     | EQUAL => x::(smerge compare xs ys)
                                     | GREATER => y::(smerge compare (x::xs) ys)

fun smergesort compare nil = nil
  | smergesort compare [x] = [x]
  | smergesort compare xs =
     let
       val (ul,vl) = split xs
     in
       smerge compare (smergesort compare ul) (smergesort compare vl)
     end
```

b) Mergesort zerteilt mit *split* die Liste in 2 kleinere Listen, die dann sortiert werden und mit *merge* richtig sortiert wieder verschmolzen werden. Es zerteilt also baumrekursiv so lange die Liste, bis sie in elementare Listen zerfallen ist und fügt diese Listen dann Ebene für Ebene gemäß der Sortierung zusammen. Mergesort weist bei allen Listen eine ähnliche und relativ kurze Laufzeit auf.

Lösung 1.2 (Logische Ausdrücke, 15=7+1+7)

a) Simplify vereinfacht logische Ausdrücke direkt, da keine Variablen vorkommen

```
fun simplify (NOT e)        = if simplify e = TRUE then FALSE else TRUE
  | simplify (AND (e1, e2)) = if simplify e1 = TRUE then simplify e2 else FALSE
  | simplify (OR (e1, e2))  = if simplify e1 = TRUE then TRUE else simplify e2
  | simplify other          = other
```

b) Der erweiterte Datentyp:

```
datatype log = TRUE | FALSE | NOT of log | OR of log * log | AND of log * log
             | ANDL of log list
             | ORL of log list
```

c) Änderung an simplify

```
 | simplify (ORL nil)      = FALSE
 | simplify (ORL (c::ol))  = if simplify c = TRUE then TRUE else simplify (ORL ol)
 | simplify (ANDL nil)     = TRUE
 | simplify (ANDL (c::al)) = if simplify c = FALSE then FALSE else simplify (ANDL al)
```

Eine Liste von Ausdrücken, die mittels Konjunktion verknüpft werden, ist falsch, wenn ein Teilausdruck falsch ist. Für die Disjunktion wird diese Liste wahr, sobald ein Ausdruck wahr ist. Auf leeren Listen ist die Konjunktion sinnvollerweise als wahr und die Disjunktion als falsch definiert.

Eine weitere Möglichkeit ist folgende Lösung:

```
 | simplify (ORL es)  = simplify (foldl OR FALSE es)
 | simplify (ANDL es) = simplify (foldl AND TRUE es)
```

Lösung 1.3 (Endrekursion und Akkus, 14=3+6+5)

a)
```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-1) + fib (n-2)
```

b)
```
fun fibi x y 0 = y
  | fibi x y n = fibi y (x+y) (n-1)
```

c)
```
fibi (0+1) (1+1) 3
fibi 1 (1+1) 3
fibi 1 2 3
fibi 2 (1+2) (3-1)
fibi 2 3 (3-1)
fibi 2 3 2
fibi 3 (2+3) (2-1)
fibi 3 5 (2-1)
fibi 3 5 1
fibi 5 (3+5) (1-1)
fibi 5 8 (1-1)
fibi 5 8 0
8
```

Lösung 1.4 (Iteration und Listen, 16=9+7)

a)
```
fun power a n = iter n 1.0 (fn x => a*x)

fun double x = first 0 (fn n => power (x+1.0) n >= 2.0)
```

b) Das kartesische Produkt zweier Listen regelbasiert mit Hilfsprozedur

```
fun kar' (x::xr) (y::yr) zr = (x,y)::kar' (x::xr) yr zr
  | kar' (x::xr)   nil   zr = kar' xr zr zr
  | kar'   nil     yr    zr = nil

fun kar xs ys = kar' xs ys ys
```

mit doppelter Faltung

```
fun kar xs ys = foldl (fn (z,a) => (foldl (fn (z',a') => (z,z')::a') nil ys)@a) nil xs
```

oder mit einfacher Faltung und map

```
fun kar xs = foldl (fn (y, a) => map (fn x => (x, y)) xs @ a) []
```

Lösung 1.5 (Bezeichnerbindung, 15=5+5+5)

a) Der Typ dieser Funktion ist $''a \to ('b \to ''a) \to 'b \to bool \to bool$

b)
```
exception E1 of int
val ( x1 , y1 , z1 ) = ( (), 2 , Empty )
fun f1 x2 = ( x2 ; raise E1 y1 )
val x3 = f1 x1 handle y2 => y2
val q1 = x3
```

c) q wird an den Wert $E\ 2$ gebunden. Sein Typ ist Exception exn.