1. Probeklausur zur Vorlesung Programmierung 1 im Wintersemester 2011/2012

Das Team der Tutoren

03. Dezember 2011

Name

Matrikelnummer

Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden.

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. Mobiltelefone sind ebenfalls dort ausgeschaltet aufzubewahren.

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 gehen.

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 90 Minuten zur Verfügung. Insgesamt können 90 Punkte werden. [Zum Bestehen der Klausur genügen 45 Punkte.]

Bitte legen Sie zur Identifikation Ihren Personalausweis bzw. Reisepass sowie Ihren Studierendenausweis neben sich.

Viel Erfolg!

1	2	3	4	5	6
10	20	15	20	15	10

Summe
90

Note

Bitte lesen

Wir versuchen euch mit dieser Probeklausur eine Vorstellung vom Ablauf und dem Aussehen der Zwischenklausur in Programmierung 1 zu geben. Dennoch ist sie insbesondere was Stoffauswahl, erlaubte Hilfsmittel und nachfolgende Hinweise angeht nicht repräsentativ für die Zwischenklausur.

Falls Sie einen Aufgabenteil *nicht* lösen können, dürfen Sie die angegebenen Prozeduren und ihre Funktionalität dennoch in den darauffolgenden Aufgabenteilen zur Lösung verwenden.

Die folgenden Prozeduren dürfen Sie verwenden, *ohne* sie vorher deklarieren zu müssen. Möchten Sie andere Prozeduren verwenden, so müssen Sie sich diese deklarieren.

```
fun foldl f s nil     = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr

fun foldr f s nil     = s
  | foldr f s (x::xr) = f(x, foldr f s xr)

fun map f nil = nil
  | map f (x::xs) = (f x) :: map f xs

fun iter 0 s f = s
  | iter n s f = iter (n-1) (f s) f

fun iterup m n s f = if m > n
                     then s
                     else iterup (m+1) n (f(m,s)) f

fun iterdn n m s f = if n<m
                     then s
                     else iterdn (n-1) m (f(n,s)) f

fun first s p = if p s
                then s
                else first (s+1) p
```

Sie dürfen auch die Prozedur *Int.compare : int* $*$ *int* $\rightarrow$ *order* verwenden. Außerdem dürfen Sie sich jederzeit zusätzliche Hilfsprozeduren deklarieren.

Aufgabe 1. Zum Aufwärmen: 2.5 + 2.5 + 2.5 + 2.5 = 10 Punkte

a) Implementieren Sie eine Prozedur *mulIntReal : real* $*$ *int* $\rightarrow$ *real*, die eine reelle und eine natürliche Zahl nur mit Hilfe von Addition und Rekursion multipliziert.

```
fun mulIntReal (x,0) = 0.0
  | mulIntReal (x,y) = x + (mulIntReal (x, y-1))
```

b) Schreiben Sie eine Prozedur *delete : (*$\alpha * \alpha \rightarrow$ *order)* $\rightarrow \alpha \rightarrow \alpha$ *list* $\rightarrow \alpha$ *list*, die das letzte Auftreten eines Elements aus einer Liste entfernt, falls es darin enthalten ist, und ansonsten die Liste unverändert zurückgibt.

Tipp: Verwenden Sie ein Tupel, um sich zu merken, ob Sie das Element bereits gelöscht haben.

```
fun delete cmp x zs = #2(foldr(fn (y,(b,ys)) =>
   (case cmp(x,y) of
     EQUAL => if b then (false, ys) else (b, y::ys)
       | _ => (b, y::ys))) (true,nil) zs)
```

c) Schreiben Sie eine Prozedur, die das folgende Typschema besitzt, ohne dabei Typen explizit anzugeben.

$$int * \alpha \rightarrow (\alpha * \beta \rightarrow int) \rightarrow \beta \rightarrow bool$$

```
fun f (i,a1) g b1 = g(a1,b1) < i
```

d) Bestimmen Sie das Typschema der folgenden Prozedurdeklaration:

```
fun jupa a b c = b (a (if c = c then a c else c))
```

$\forall\, '\alpha, \beta$: $('\alpha \rightarrow '\alpha) \rightarrow ('\alpha \rightarrow \beta) \rightarrow '\alpha \rightarrow \beta$

Aufgabe 2. (5 + 3 + 5 + 3 + 4 = 20 Punkte)

Im Restaurant *Hermanns & Eisentraut* sollen Sie die täglich anfallenden Bestellungen auf eine ganz spezielle Art sortieren.

Bestellen kann man in diesem Restaurant nur Kaffee, Tee und Wasser:

```
datatype getraenk = KAFFEE | TEE | WASSER
```

Eine einzelne Bestellung besteht immer aus einem Getränk und wie oft es bestellt wurde. Eine Rechnung besteht aus einer Liste von Bestellungen.

```
type anzahl = int
type bestellung = getraenk * anzahl

type rechnung = bestellung list
```

Das Restaurant *Hermanns & Eisentraut* verlangt von Ihnen nun folgende Sortierung der Rechnung: An erster Stelle der Rechnung sollen alle Kaffeebestellungen stehen, gefolgt von den Teebestellungen. Am Ende sollen die Wasserbestellungen stehen. Innerhalb der einzelnen Blöcke sollen die Bestellungen aufsteigend nach Anzahl sortiert sein.

Folgende Bestellung ist korrekt sortiert:

```
[(KAFFEE, 2), (KAFFEE, 4), (TEE, 1), (WASSER, 5), (WASSER, 7)]
```

Da es in dieser Branche nicht so sehr auf Geschwindigkeit ankommt, entschließen Sie sich, einen besonders ineffizienten, dafür aber interessanten Sortieralgorithmus zu verwenden:

Wir mischen die Liste solange, bis sie irgendwann sortiert ist.

a) Schreiben Sie zunächst eine Prozedur *compare : bestellung* $*$ *bestellung* $\rightarrow$ *order*, die die für Bestellungen gegebene Ordnung umsetzt.

```
fun compare ((KAFFEE, i1), (KAFFEE,i2)) = Int.compare(i1,i2)
  | compare ((KAFFEE,_), (_,_)) = LESS
  | compare ((TEE,i1), (TEE,i2)) = Int.compare(i1,i2)
  | compare ((TEE,_), (KAFFEE,_)) = GREATER
  | compare ((TEE,_), _) = LESS
  | compare ((WASSER,i1), (WASSER, i2)) = Int.compare(i1,i2)
  | compare ((WASSER,_), _) = GREATER
```

b) Implementieren Sie eine Prozedur *unsorted : ($\alpha * \alpha \rightarrow$ order)* $\rightarrow \alpha$ *list* $\rightarrow$ *bool*, die genau dann *true* zurückgibt, wenn die Liste *nicht sortiert* ist.

```
fun unsorted order [] = false
  | unsorted order [h] = false
  | unsorted order (f::s::t) =
        (case(order(f,s)) of
          EQUAL => (unsorted order (s::t))
        | LESS    => (unsorted order (s::t))
        | GREATER => true)
```

c) Als letzten Schritt wollen wir eine Prozedur implementieren, die alle Möglichkeiten bestimmt, wie man eine gegebene Liste mischen kann.

- Schreiben Sie zunächst eine Prozedur *prepend* : $\alpha \rightarrow \alpha$ *list list* $\rightarrow \alpha$ *list list*, die einen Wert an den Anfang jeder Liste einer Liste von Listen anhängt.

```
fun prepend x ys = foldr (fn(a,b) => (x::a)::b) nil ys
```

- Implementieren Sie nun mit Hilfe der Prozeduren *prepend* und *delete* (aus Aufgabe 1) eine Prozedur *mischungen* : $(\alpha * \alpha \rightarrow order) \rightarrow \alpha$ *list* $\rightarrow \alpha$ *list list*, die alle Möglichkeiten, eine gegebene Liste zu mischen (ihre Umordnungen) liefert.

```
fun mischungen cmp nil  = [nil]
  | mischungen cmp [x]  = [[x]]
  | mischungen cmp (xs)   = foldr (fn(a,b) =>
              prepend a (mischungen cmp (delete cmp a xs)) @ b) nil (xs)
```

d) Schreiben Sie nun den angegebenen Sortieralgorithmus mit Hilfe der Prozedur *mischungen* aus Teilaufgabe c).

```
fun sort' order nil     = raise Empty
  | sort' order (x::xr) = if (unsorted order x)
                          then sort' order xr
                          else x

fun bogosort nil = nil
  | bogosort [x] = [x]
  | bogosort xs  = let
                      val ys = mischungen compare xs
                   in sort' compare ys
                    end
```

Aufgabe 3. (6 + 5 + 4 = 15 Punkte)

```
datatype farbe = GRUEN | BLAU | GELB | ROT
datatype haufen = HEU | NADEL of farbe
```

a) Schreiben Sie eine Prozedur *find : haufen list → unit*, die eine von Ihnen definierte Ausnahme *Nadel : farbe → exn* wirft, sobald eine Nadel im Heuhaufen gefunden wurde. Für den Fall, dass der Heuhaufen keine Nadeln enthält, geben Sie *()* zurück.

```
exception Nadel of farbe

fun find nil = ()
  | find (x::xs) = (case x of
       HEU => find xs
    | (NADEL f) => raise (Nadel f))
```

b) Schreiben Sie jetzt eine Prozedur *first : haufen list → farbe*, die die Farbe der ersten Nadel im Heuhaufen bestimmt. Falls keine Nadel im Heuhaufen gefunden wurde, soll die von Ihnen definierte Ausnahme *NurHeu : exn* geworfen werden.

Verwenden Sie die Prozedur *find* aus Aufgabenteil a) und sonst keine weiteren Hilfsprozeduren.

```
exception NurHeu

fun first xs = (find xs; raise NurHeu) handle (Nadel f) => f
```

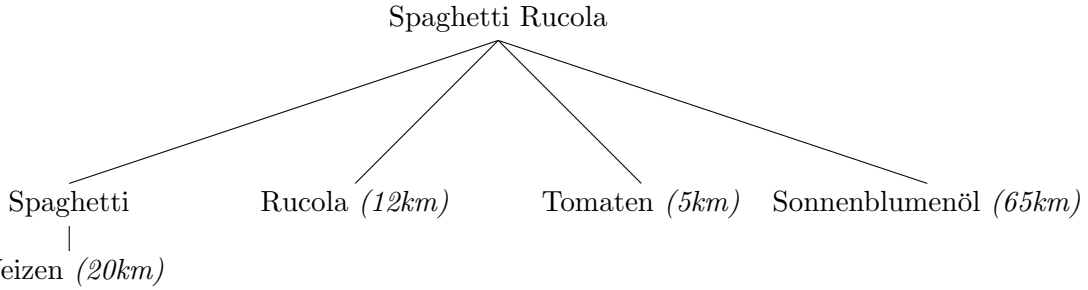
c) Schreiben Sie eine Prozedur *nurNadeln : haufen list → haufen list list*, die aus einem Heuhaufen alle Nadeln aussortiert und eine Liste von vier Listen zurückgibt. Die erste Liste soll alle grünen, die zweite alle blauen, die dritte alle gelben und die vierte letztendlich alle roten Nadeln enthalten.

```
fun nurNadeln xs =
  let
    fun hilf ys = foldl
      (fn (x,(grs, bls, ges, ros)) =>
        case x of
         HEU => (grs, bls, ges, ros)
       | (NADEL y) =>
            (case y of
                 GRUEN => (x::grs, bls, ges, ros)
               | BLAU  => (grs, x::bls, ges, ros)
               | GELB  => (grs, bls, x::ges, ros)
               | ROT   => (grs, bls, ges, x::ros)
             )
        )
    (nil,nil,nil,nil) ys
    val (a,b,c,d) = hilf xs
  in
    a::b::c::d::nil
  end
```

Aufgabe 4. (6 + 4 + 4 + 6 = 20 Punkte)

Da das Restaurant *Hermanns & Eisentraut* sehr umweltbewusst ist, möchte es seinen Kunden gerne zeigen, dass alle Zutaten von der Karte einen besonders kurzen Weg bei der Anlieferung zurückgelegt haben.
Daher werden Sie beauftragt, eine Datenstruktur zu implementieren, die die gewünschten Informationen übersichtlich darstellt.

Hierzu ein Beispiel:



a) Implementieren Sie nun den Datentyp *foodtree*. Dieser soll ein Baum sein, dessen Blätter die Zutat und die Entfernung der Zutat repräsentieren. Die inneren Knoten enthalten nur den Namen eines selbst hergestellten Produkts, sowie die Liste der Produkte bzw. Zutaten, aus denen diese wiederum hergestellt werden. Dies können beliebig viele sein.

```
datatype foodtree = B of string * int | T of string * foodtree list
val example = T("Spaghetti Rucola", [
  T("Spaghetti", [
    B("Weizen", 20)
  ]),
  B("Rucola", 12),
  B("Tomaten", 5),
  B("Sonnenblumenoel", 65)
])
```

b) Ihre Auftraggeber sind mit Ihrer Arbeit zufrieden. Nachdem die ersten Kunden Ihr Werk begutachten durften, sollen Sie nun auch die maximale Kilometeranzahl in einem dieser Bäume ausgeben können. Implementieren Sie also eine Prozedur *max : foodtree → (string ∗ int)*, welche die Zutat mit größter Kilometeranzahl zurückgibt.

```
fun max (T(s,ts)) = foldl (fn ((a,i), (sa,si)) =>
    if i>si
      then (a,i)
      else (sa,si)) ("",0) (map max ts)
| max (B(a,i)) = (a,i)
```

c) Außerdem werden Sie beauftragt eine Prozedur *sum : foodtree → int* zu schreiben, die die Summe aller Einträge bildet und somit die Gesamtkilometerzahl berechnet.

```
fun sum (T(s,ts)) = foldl (fn (a,s) => a+s) 0 (map sum ts)
  | sum (B(s,i)) = i
```

d) Zuletzt sollen Sie noch eine Prozedur *deep : foodtree* $\rightarrow$ *(int* $*$ *string)* bereitstellen, welche die **Tiefe** und den Namen des in Präordnung ersten, am weitesten unten liegenden Knotens berechnet.

```
fun deep (T(s,ts)) =
let
  val ((d,l)::xs) = map deep ts
in
   foldl (fn ((d,l), (s,n)) =>
      if (d+1)>s
       then (d+1,l)
       else (s,n)) (d+1,l) xs
 end
  | deep (B(s,i)) = (0,s)
```

Aufgabe 5. (6 + 4 + 5 = 15 Punkte)

Unter einem Zahlenpalindrom versteht man eine Zahl $m \in \mathbb{N}$, deren Ziffernfolge spiegelsymmetrisch ist, formal gegeben durch: $x_0x_1 \ldots x_nx_n \ldots x_1x_0$ für n gerade und $x_0x_1 \ldots x_{n-1}x_nx_{n-1} \ldots x_1x_0$ für n ungerade, wobei $x_0, x_1, \ldots, x_n \in \{0, 1, 2, \ldots, 9\}$. Beispiele hierfür sind $0, 101, 8998, 8993998, \ldots$ (*Hinweis:* Alle einstelligen Zahlen sind Zahlenpalindrome. Das kleinste Palindrom ist 0).

a) Schreiben Sie eine Prozedur *ispal : int → bool*, welche für ein gegebenes $n \in \mathbb{N}$ überprüft, ob n ein Zahlenpalindrom ist.

```
fun rev' a 0 = a
  | rev' a n = rev' (10*a+n mod 10) (n div 10)

val rev = rev' 0

fun ispal x = x=rev x
```

b) Das größte Zahlenpalindrom, welches sich als Produkt von zwei zweistelligen natürlichen Zahlen darstellen lässt, ist $9009 = 91 \cdot 99$.

Schreiben Sie eine Prozedur *biggest : unit → int*, die das größte Zahlenpalindrom berechnet, welches sich als das Produkt von zwei *dreistelligen* natürlichen Zahlen darstellen lässt.

Tipp: Sie müssen alle möglichen Paare von dreistelligen Zahlen betrachten. Sie können dazu beispielsweise zweimal *iterup* –ineinander verschachtelt– verwenden.

```
fun biggest() = iterup 100 999 0
                (fn(a, b) => iterup 100 999 b
                (fn(c, d) => if ispal (a*c) andalso a*c > d then a*c else d))
```

c) Wir betrachten nun die Funktion $nthpal : \mathbb{N}\backslash\{0\} \to \mathbb{N}$, welche das n-te Zahlenpalindrom berechnet. Zum Beispiel ist $nthpal(1) = 0$, $nthpal(2) = 1$, $nthpal(3) = 2$, $nthpal(11) = 11$,..., $nthpal(54) = 444$,..., $nthpal(99) = 898$, etc....

Schreiben Sie eine Prozedur, die die Funktion *nthpal* berechnet.

```
fun nthpal n = iter n 0 (fn a => first a ispal+1) - 1
```

Aufgabe 6. (5 + 2.5 + 2.5 = 10 Punkte)

a) Überlegen Sie sich, ob der Interpreter die folgenden Deklarationen akzeptiert. Jede richtige Antwort bringt Ihnen 0.5 Punkte, jede falsche -0.5 Punkte. Die minimale Punktzahl in dieser Aufgabe beträgt jedoch 0 Punkte.

Eingabe	wird vom Interpreter akzeptiert	
`fun f x = f x`	Ja ■	Nein □
`fun f x y = f x y`	Ja ■	Nein □
`fun f y z = f + y + z`	Ja □	Nein ■
`val z = (fn i => z i)`	Ja □	Nein ■
`val a = (if 3 < 5 then 5 else 7) + 2`	Ja ■	Nein □
`val b = if 3 < 5 then 5 else 7 + 2`	Ja ■	Nein □
`fun f x = 3 + (#1 x)`	Ja □	Nein ■
`fun f x = 2 * x / 2`	Ja □	Nein ■
`fun f nil = nil` `  |f x::xs = f x`	Ja □	Nein ■
`fun f x = f ( 3 + (#1 x), 5)`	Ja ■	Nein □

b) Warum endet folgendes Programm mit "Uncaught Exception: Q"?

```
exception Q
val ergebnis =
 let
   exception Q
   val v = #1 (4,raise Q,true)
 in
   (true;nil;17) handle Q => ~2
 end
```

Die Ausführung endet mit "Uncaught Exception Q", da die Zuweisung von v nicht vom handle zwischen in und end betroffen ist.

c) Erklären Sie, warum im folgenden Programm der Bezeichner *ergebnis* an 2 gebunden wird.

```
fun out a =
  let
    exception B
    fun f x =
      let
        exception B
        fun g x = if x mod 2 = 0
                  then 0
                  else raise B
      in
        g ( x + 3 )
      end
  in
    (f a) handle B => 1
  end

val ergebnis = (out 4) handle A => 2
```

Der Bezeichner *ergebnis* wird an 2 gebunden, da in

```
 val ergebnis = (out 4) handle A => 2
```

A eine Variable ist, die jede beliebige Exception, die in der Ausführung von *out* auftreten kann, handelt.