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

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 7

(Kapitel 7)

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.*

Die Aufgaben entstammen alle aus dem Nachklausurtutorium im Wintersemester 11/12!

Aufgabe 7.1 (*Baumschule*)

(a) Geben Sie die Deklaration des Datentyps *tree* an.

(b) Geben Sie die Deklaration von *fold* an.

(c) Geben Sie die Deklaration des Datentyps *ltr* an.

Lösung 7.1:

- `datatype tree = T of tree list`
- `fun fold f (T ts) = f (map (fold f) ts)`
- `datatype 'a ltr = L of 'a * 'a ltr list`

Aufgabe 7.2 (*Garten*)
Gegeben Sei der folgende Baum: $T[T[], T[T[], T[]], T[T[], T[T[], T[]]]]$

(a) Zeichnen Sie eine Darstellung des Baumes.

(b) Zeichnen Sie alle Unterbäume des Baumes.

(c) Nummerieren Sie die Knoten gemäß Postnummerierung.

(d) Zeichnen Sie alle binären Teilbäume.

(e) Zeichnen Sie alle linearen Teilbäume.

(f) Geben Sie die Adressen aller Blätter an.

(g) Wie viele innere Knoten hat der Baum?

(h) Geben Sie einen Ausdruck an, der den gespiegelten Baum beschreibt.

(i) Welche Tiefe hat der Baum?

(j) Welche Breite hat der Baum?

(k) Welchen Grad hat der Baum?

(l) Sei a der durch die Adresse [3] bezeichnete Knoten.

(i) Geben Sie die Adresse des 1. Nachfolgers von a an.

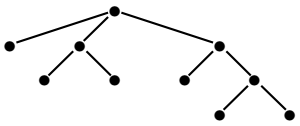
(ii) Geben Sie die Pränummer aller Nachfolger von a an.

(iii) Geben Sie die Pränummer des Vorgängers von a an.

(iv) Geben Sie die Adressen aller Knoten an, die dem Knoten a untergeordnet sind.

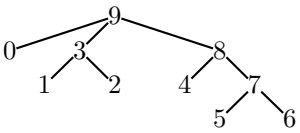
Lösung 7.2:

(a)



(b) Es gibt 3 Unterbäume:

(c)



(d) Es gibt 3 binäre Teilbäume:

(e) Es gibt einen linearen Teilbaum:

(f)
- [1]
- [2, 1]
- [2, 2]
- [3, 1]
- [3, 2, 1]
- [3, 2, 2]

(g) Der Baum hat 4 innere Knoten.

(h) $T[T[T[T[], T[]], T[]], T[T[], T[]], T[]]$

(i) Der Baum hat die Tiefe 3.

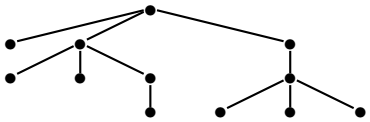
(j) Der Baum hat die Breite 6.

(k) Der Baum hat den Grad 3.

(l) (i) $[3, 1]$
(ii) $6, 7$
(iii) 0
(iv) $[3], [3, 1], [3, 1], [3, 2], [3, 2, 1], [3, 2, 2]$

Aufgabe 7.3 (*Bonsai*)
Zeichnen Sie den kleinsten Baum, für den die Adressen $[2, 3, 1]$ und $[3, 1, 3]$ gültig sind.

Lösung 7.3:



Aufgabe 7.4 (*Winter*)
Zeichnen Sie einen Baum, der keine inneren Knoten hat.

Lösung 7.4:

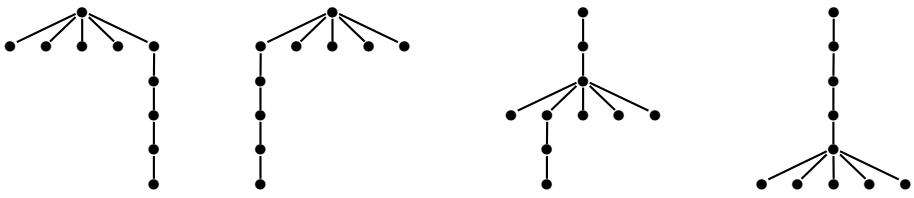
•

Aufgabe 7.5 (*Wildwuchs 1*)
Geben Sie einen Baum mit 5 Knoten an, der genau zwei Teilbäume hat. Wie viele solche Bäume gibt es?

Lösung 7.5:
Der einzige Baum mit 5 Knoten, der genau zwei Teilbäume hat, sieht wie folgt aus:

Aufgabe 7.6 (*Wildwuchs 2*)
Zeichnen Sie einen Baum des Grades 5 und der Tiefe 5. Der Baum soll 10 Knoten besitzen.
Wie viele verschiedene Bäume gibt es, die diese Bedingungen erfüllen?

Lösung 7.6:
Hier sind Bäume, die die Bedingungen erfüllen:



Es gibt $4 \cdot 5 + 1 = 21$ *TODO: PRÜFEN!* verschiedene Bäume, die diese Bedingungen erfüllen.

Aufgabe 7.7 (*Treesharing*)
Zeichnen Sie zwei Bäume mit mindestens 10 Knoten und geben Sie von einem Baum die Prä- und vom anderen die Postlinearisierung an. Lassen Sie Ihren Nachbarn nur anhand der Linearisierungen die beiden Bäume zeichnen. Vergleichen Sie Ihre Bäume!

Aufgabe 7.8 (*Für Dieter*)
Erklären Sie Dieter Schlau (oder wahlweise Ihrem Tutor), was *fold* tut.

Aufgabe 7.9 (*mirror 2.0*)
Deklarieren Sie eine zu *mirror* semantisch äquivalente Prozedur. Benutzen Sie dabei *rev* und *fold* und keine zusätzliche Form der Rekursion.

Lösung 7.9:

```
val mirror = fold (fn ts => T(rev ts))
```

Aufgabe 7.10 (*Herbst*)
Bald wird es wieder Sommer. Zu warm für Dieter Schlau! Helfen Sie ihm, sich gedanklich schon wieder auf den Herbst einzustellen.

Schreiben Sie eine Prozedur *herbst* : *tree* $\rightarrow$ *tree*, die einen Baum nimmt und alle Blätter entfernt.

Beispiel:
Aus dem Baum  wird  .

Wenn nach dem Entfernen der Blätter kein gültiger Baum mehr übrig bliebe, soll die Ausnahme *nixmehrda* geworfen werden. Geben Sie alle Bäume an, für die dies der Fall ist.

Lösung 7.10:

```
exception nixmehrda fun herbst' (T ts) = T (foldl (fn (x, a) => if x = T[] then a else a
@ [herbst' x]) nil ts) fun herbst t = if t = T[] then raise nixmehrda else herbst' t
```

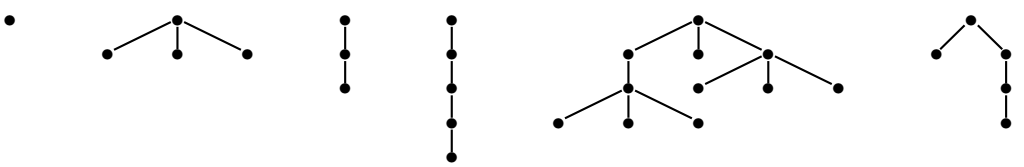
Die Ausnahme *nixmehrda* wird nur für den Baum $T[]$ geworfen, da dann keine Wurzel mehr übrig bliebe.

Aufgabe 7.11 (*Lexikalische Baumordnung*)
Folgende Prozedur implementiert die lexikalische Baumordnung:

```
fun compareTree (T ts, T tr) = List.collate compareTree (ts,tr)
```

(a) Beschreiben Sie (in Worten!) einen Algorithmus, wie man zwei Bäume vergleichen kann.

(b) Ordnen Sie die folgenden Bäume:



(c) Deklarieren Sie eine zu *compareTree* äquivalente Prozedur, ohne dabei die Prozedur *List.collate* zu verwenden.

(d) Gegeben seien die folgenden Prozeduren:

```
fun p1(t::t::tr) = compareTree(t,t) = LESS andalso p1(t::tr)
  | p1 s = true
fun p2 (T ts) = p1 ts andalso List.all p2 ts
```

(i) Welche Eigenschaft erfüllen Bäume, für die *p1 true* liefert?

(ii) Welche Eigenschaft erfüllen Bäume, für die *p2 true* liefert?

(iii) Reine Mengen können durch Bäume dargestellt werden. Schreiben Sie eine Prozedur $direct : tree \to tree$, die zu einem Baum t einen anderen Baum t' liefert, so dass t' das Prädikat *p2* erfüllt, aber t und t' immer noch die gleiche Menge darstellen.

Verwenden Sie eine polymorphe Sortierprozedur.

Lösung 7.11:

(a) Die folgenden Schritte müssen in dieser Reihenfolge ausgeführt werden:

(i) Sind beide Bäume identisch, so sind sie lexikalisch gleich.

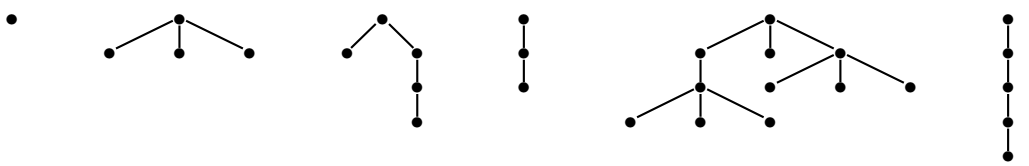
(ii) Ist einer der Bäume atomar, so ist dieser der lexikalisch kleinere Baum.

(iii) Nun müssen die Unterbäume *der Reihe nach* überprüft werden:

i. Ist ein Unterbaum lexikalisch kleiner (ermittelt per Rekursion), so ist der dazugehörige Baum lexikalisch kleiner, sonst werden die nächsten Unterbäume geprüft

ii. Wenn alle Unterbäume überprüft sind, sind die Bäume lexikalisch gleich.

iii. Wenn ein Baum noch ungeprüfte Unterbäume hat und der andere nicht mehr, so ist der Baum mit ungeprüften Unterbäume lexikalisch größer.

(b) 

(c)
```
fun compareTree ((T nil) , (T nil) ) = EQUAL | compareTree ((T nil) , s ) = LESS | compareTree
( s , (T nil) ) = GREATER | compareTree ((T (x::xr)), (T(y::yr))) = (case (compareTree
(x,y)) of EQUAL => compareTree ((T xr), (T yr)) | s => s)
```

(d) (i) Die Unterbäume solche Bäume sind strikt sortiert.

(ii) Solche Bäume sind *gerichtet*.

(e)
```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys)) (nil, nil) xs fun smerge s (nil,ys)
= ys | smerge s (xs,nil) = xs | smerge compare (x::xr,y::yr) = case compare(x,y) of LESS
=> x::smerge compare (xr,y::yr) | EQUAL => x::smerge compare (xr,yr) | GREATER => y::smerge
compare (x::xr,yr) fun ssort compare = let fun ssort nil = nil | ssort [x] = [x] | ssort
xs = let val (ys,zs) = split xs in smerge compare (ssort ys,ssort zs) end in ssort end
fun direct (T ts) = T(ssort compareTree (map direct ts))
```

Lösung 7.11:

(a)
```
fun shape (C s) = T[] | shape (A(e1,e2)) = T[shape e1, shape e2] | shape (GGT xs) = T(map shape xs)
```

(b)
```
fun max nil = raise Domain | max (x::[]) = x | max (x::xr) = Int.max(x, max xr) fun firstdn s p = if p s then s else firstdn (s-1) p fun findGGT xs = firstdn (max xs) (fn n => foldl (fn (x,a) => a andalso x mod n = 0) true xs) fun eval (C x) = x | eval (A(e1,e2)) = eval e1 + eval e2 | eval (GGT xs) = findGGT (map eval xs)
```

Aufgabe 7.12 (*Achtung, Verbrennungsgefahr: Rekonstruktion*)
Schreiben Sie eine Prozedur $rekonst : tree \to \alpha\, list \to \alpha\, ltr$, die aus der Gestalt und der Präprojektion eines markierten Baumes diesen rekonstruiert, d.h. es soll gelten:
$rekonst\ (shape\, t)\ (prep\, t) = t$.

Lösung 7.12:

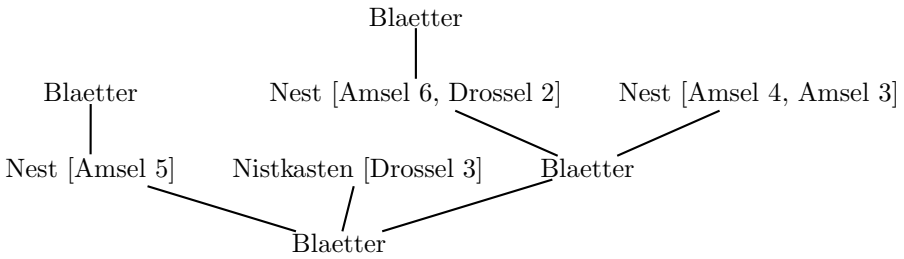
```
exception ThatDoesNotMatch fun rekonst' s nil = raise ThatDoesNotMatch | rekonst' (T ts) (x::xr) = let val (ltrees, xrest) = foldl (fn (x,(lts, xs)) => let val (tree, restlist) = (rekonst' x xs) in (lts @ [tree], restlist) end ) (nil, xr) ts in (L(x, ltrees), xrest) end fun rekonst t xs = let val(ltr, restlist) = rekonst' t xs in if null restlist then ltr else raise ThatDoesNotMatch end
```

Aufgabe 7.13 (*Waldspaziergang mit Dieter Schlau*)
Es wird Frühling!

Die Vöglein zwitschern und Dieter Schlau macht in der Früh einen Spaziergang durch den Wald. Dabei fällt ihm auf, dass man die Bäume um ihn herum auch als markierte Bäume darstellen kann:

Wie schon bekannt besteht ein Baum aus einer Liste an Bäumen, die Marke stellt in diesem Fall entweder ein Nest (mit einer *vogel list*), einen Nistkasten (ebenfalls mit einer *vogel list*) oder einfache Blätter dar. Dabei kennt Dieter Amseln und Drosseln.

Ein Baum könnte Beispielsweise so aussehen:



Wie Ihnen sicher aufgefallen ist, wachsen diese Bäume (realitätsnah) nach oben. Dabei handelt es sich aber um eine willkürliche Festlegung der Darstellung, die Sie verwirren sollte.

Jeder Vogel soll auch einen Nährwert (dargestellt als *int*) haben.

(a) Entwickeln Sie passende Datentypen *vogel*, *astgabel* und *btree*, die Ihnen erlauben, solche Bäume in SML darzustellen!

(b) Während Dieter mit Bleistift und Block auf einer Bank kauert, hat sich ihm der Oberförster des Waldes genähert. Er hat eine ganz besondere Bitte an ihn:

Zur Zeit wird im Dorf stark diskutiert, inwiefern Brutkästen das Ansiedeln von Vögeln und das Wachstum der Bäume selbst beeinflusst.

Die Partei "Baum ist Baum" behauptet, dass das Wachstum der Bäume selbst von Nistkästen behindert wird und fordert deshalb alle Nistkästen abzuschaffen. Um dies zu überprüfen schreiben Sie eine Prozedur $xsize : btree \to int * bool$, die zu einem Tupel auswertet, dessen erste Komponente die Größe des Baumes beschreibt und dessen zweite Komponente angibt, ob der Baum einen Nistkasten enthält.

(c) Bei den Auswertungen der Ergebnisse Ihrer ersten Prozedur ist nicht ganz das gewünschte Ergebnis herausgekommen, deshalb bittet der Parteivorsitzende Sie zusätzlich eine Prozedur $xdepth : btree \to int * bool$ zu schreiben, die zu einem Tupel auswertet, dessen erste Komponente die Höhe *(bzw. die Tiefe)* des Baumes beschreibt und dessen zweite Komponente angibt, ob der Baum einen Nistkasten enthält.

(d) "Alles Unsinn", mein dagegen die Partei "Pro Birds". "Durch die Nistkästen finden viel mehr Vögel ein Zuhause!"

Schreiben Sie eine endrekursive Prozedur $count : btree \to int * int$, die zählt, wie viele Vögel auf einem Baum in einem Nistkasten und wie viele in einem Nest leben.

(e) Die Partei "Baum ist Baum" hat dank Ihrer Arbeit - tatsächlich die nächste Wahl gewonnen und fordert: Weg mit den Nistkästen! Schreiben Sie eine Prozedur $wegdamit : btree \to btree$, die alle Nistkästen durch Blätter ersetzt (um die Vögel müssen Sie sich dabei *selbstverständlich* nicht sorgen).

(f) Puh! Das war vielleicht ein anstrengender Tag!

Dieter Schlau ist komplett k.o., als er endlich zu Hause ankommt. Dort angekommen wartet schon seine Oma auf ihn: Der Grünschnitt in ihrem Garten muss unbedingt noch erledigt werden!

Grünschnitt sieht bei Oma Schlau folgendermaßen aus: Immer wenn von einem Ast aus mehr als zwei Äste weggehen, sollen nur zwei übrig bleiben.

Schreiben Sie eine Prozedur $gruenschnitt : btree \to btree$, die Bäume dementsprechend kürzt!

(g) Für echte Gärtner:

Grünschnitt ist um einiges effektiver, wenn nicht zwei beliebige Äste, sondern die längsten Äste *(die Äste mit der größten Tiefe)*) übrig bleiben.

Verbessern Sie Ihre Prozedur dementsprechend!

(h) Oh nein! Der Tiger ist aus dem Zoo ausgebrochen!

Nicht genug gesättigt von den rein virtuellen Kämpfen gegen Mäuse und den virtuellen Fütter-Experimenten von Dieter streift er durch den Wald und hat es auf die armen kleinen Vöglein abgesehen!

Erfahrungsgemäß klettert ein Tiger einen Baum entlang der Standardtour ab. Dabei verspeist er alle Nester und Nistkästen, denen er begegnet, in einem Bissen (zurück bleiben Blätter). Dadurch wird er weniger hungrig: Sein Hunger sinkt dann um die Numme der Nährwerte der gefressenen Vögel.

Ist der Tiger satt, so verlässt er den Baum auf dem kürzesten Wege.

Schreiben Seine Prozedur $impactCalculator : int \rightarrow btree \rightarrow btree$, die den Hunger eines Tigers und einen Baum nimmt und den Ergebnisbaum ausgibt.

Wenn der Tiger nach dem Verlassen des Baumes immer noch hungrig ist, so frisst er leider *Sie*. Das soll durch die Ausnahme *RIP* angezeigt werden.

Lösung 7.13:

(a)
```
type naehrwert = int datatype vogel = Amsel of naehrwert | Drossel of naehrwert datatype
astgabel = Nest of vogel list | Nistkasten of vogel list | Blaetter datatype btree = BT
of astgabel * btree list
```

(b)
```
fun nistkasten (BT(x, nil)) = (case x of (Nistkasten y) => true | s => false) | nistkasten
(BT(x, ts)) = (case x of (Nistkasten y) => true | s => (foldl (fn (x,s) => x orelse s)
false (map nistkasten ts)) ) fun size (BT (x, nil)) = 1 | size (BT (x, ts)) = 1 + foldl
op+ 0 (map size ts) fun xsize xs = (size xs, nistkasten xs)
```

(c)
```
fun depth (BT (x, nil)) = 0 | depth (BT (x, ts)) = 1 + foldl Int.max 0 (map depth ts) fun
xdepth xs = (depth xs, nistkasten xs)
```

(d)
```
fun count ts = let fun count' (BT (Blaetter, xs)) (a,b) = foldl (fn (x, (a',b')) => count'
x (a',b')) (a,b) xs | count' (BT (Nest x, xs)) (a,b) = foldl (fn (x, (a',b')) => count'
x (a',b')) (a + list.length x,b) xs | count' (BT (Nistkasten x, xs)) (a,b) = foldl (fn
(x, (a',b')) => count' x (a',b')) (a, b+ list.length x) xs in count' ts (0,0) end
```

(e)
```
fun wegdamit (BT (x,nil)) = (case x of (Nistkasten y) => (BT (BLAETTER, nil)) | s => (BT(x,nil)))
| wegdamit (BT (x,ts)) = let val (x', ts) = (case x of (Nistkasten y) => (BLAETTER,nil)
| s => (x, ts)) in (BT (x', map (wegdamit) ts)) end
```

(f)
```
fun gruenschnitt (BT (x, nil)) = BT (x,nil) | gruenschnitt(BT(x, ts)) = case Int.compare
((length ts),2) of GREATER => BT (x, [gruenschnitt (hd ts), gruenschnitt (hd (tl ts))])
| s => BT (x, map gruenschnitt ts)
```

(g)
```
fun depthcmp (xs, ys) = Int.compare (depth xs, depth ys)
fun pisort compare = let fun insert (x,nil) = [x] | insert (x, y::yr) = case compare(x,y)
of GREATER => y::(insert (x,yr)) | s => x::y::yr in foldl insert nil end
val dsort = pisort depthcmp

fun gruenschnitt (BT (x, nil)) = BT (x,nil) | gruenschnitt(BT(x, ts)) = case Int.compare
((length ts),2) of GREATER => BT (x, [gruenschnitt (hd ts), gruenschnitt (hd (tl ts))])
| s => BT (x, map gruenschnitt ts) fun gruenschnitt' (BT (x, nil)) = BT (x,nil) | gruenschnitt'(BT(x,
ts)) = case Int.compare ((length ts),2) of GREATER => let val sortedlist = rev (dsort ts)
in BT (x, [gruenschnitt' (hd sortedlist), gruenschnitt' (hd (tl sortedlist))]) end | s
=> BT (x, map gruenschnitt' ts)
```

(h)
```
exception RIP fun naehrVogel (Amsel x) = x | naehrVogel (Drossel x) = x fun naehrwert Blaetter
= 0 | naehrwert (Nest xs) = foldl (fn (x,a) => a + naehrVogel x) 0 xs | naehrwert (Nistkasten
xs) = foldl (fn (x,a) => a + naehrVogel x) 0 xs fun impactCalculator' n (BT (t, ts)) =
if n <= 0 then (n, (BT (t, ts))) else let val (nh, ats) = foldl (fn (x, (nh, ats)) => let
val (nh',ats') = impactCalculator' nh x in (nh', ats @ [ats']) end) (n - naehrwert t, nil)
ts in (nh, BT(Blaetter, ats)) end fun impactCalculator n bt = let val (nh, ats) = impactCalculator'
n bt in if nh > 0 then raise RIP else ats end
```