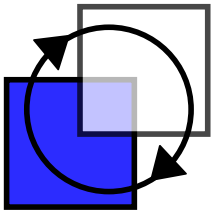
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Übungsblatt XMAS (Lösungsvorschläge)

Lesen Sie im Buch Kapitel 1 - 6.4.1.

Hinweis: Über Aufgaben, die mit 🤔 markiert sind, müssen Sie eventuell etwas länger nachdenken. Falls Ihnen keine Lösung einfällt - kein Grund zur Sorge. Kommen Sie in die Office Hour, unsere Tutor:innen helfen gerne.

1 Einfache Rekursive Prozeduren

Aufgabe XMAS.1
Schreiben Sie eine endrekursive Prozedur `zweiundvierzig : int → int`, die auf eine Zahl solange 1 addiert, bis sie größer als 41 ist.

Lösungsvorschlag XMAS.1

```
1 fun zweiundvierzig (x : int) = if x > 41 then x else zweiundvierzig(x+1)
```

Aufgabe XMAS.2
Schreiben Sie eine Prozedur, die zu $n \in \mathbb{N}$ das kleinste $k \in \mathbb{N}$ mit $k^3 \geq n$ berechnet.

Lösungsvorschlag XMAS.2

```
1 fun f' (k : int, n : int) : int = if k * k * k >= n then k else f' (k + 1, n
  ↳ )
2 fun f (n : int) = f' (1, n)
```

Aufgabe XMAS.3
Der ganzzahlige Rest x mod y lässt sich aus x durch wiederholtes Subtrahieren von y bestimmen. Schreiben Sie eine rekursive Prozedur `mymod : int * int → int`, die für $x \geq 0$ und $y \geq 1$ das Ergebnis x mod y liefert.

Lösungsvorschlag XMAS.3

```
1 fun mymod(x : int, y : int) : int = if x < y then x else mymod(x - y, y)
```

Aufgabe XMAS.4
Schreiben Sie analog zur vorherigen Aufgabe eine Prozedur `mydiv : int * int → int`, die für $x \geq 0$ und $y \geq 1$ das Ergebnis x div y liefert. Schreiben Sie `mydiv` auf zwei Arten: Ohne Hilfsprozedur und mit endrekursiver Hilfsprozedur.

Lösungsvorschlag XMAS.4
Ohne Hilfsprozedur:

```
fun mydiv (x,y) = if x < y then 0 else 1 + mydiv (x-y, y)
```

Mit endrekursiver Hilfsprozedur:

```
fun mydiv' (x,y,a) = if x < y then a else mydiv (x-y, y, a+1)
fun mydiv (x,y) = mydiv'(x,y,0)
```

Aufgabe XMAS.5
Schreiben Sie eine Prozedur `binom : int * int → int`, die für $n, k \geq 0$ den Binomialkoeffizienten $\binom{n}{k}$ berechnet. Vervollständigen Sie zunächst die folgenden Rekursionsgleichungen.

$$\binom{n}{} =$$
$$\binom{}{k} = \qquad \text{für } k > 0$$
$$\binom{n}{k} = \qquad \text{für } n, k > 0$$

Lösungsvorschlag XMAS.5

$$\binom{n}{0} = 1$$
$$\binom{0}{k} = 0 \qquad \text{für } k > 0$$
$$\binom{n}{k} = \frac{n \cdot \binom{n-1}{k-1}}{k} \qquad \text{für } n, k > 0$$

```
fun binom (n: int, k:int) :int = if k=0 then 1
else if n=0
  then 0
  else n * binom (n-1,k-1) div k
```

Aufgabe XMAS.6
Schreiben Sie eine Prozedur `sump : (int → bool) → int → int`, die für ein gegebenes Prädikat `p : int → bool` alle Ziffern einer Zahl addiert, die `p` erfüllen. Z.B. soll `sump (fn z ⇒ true) x` die Quersumme von x berechnen.

Lösungsvorschlag XMAS.6

```
fun sump p 0 = 0
  | sump p x = if p(x mod 10) then x mod 10 + sum p (x div 10)
      else sum p (x div 10)
```

Aufgabe XMAS.7
Schreiben Sie mithilfe von div und mod eine Prozedur `evens : int → int list`, die gegeben eine positive Zahl $x \geq 0$, alle geraden Zahlen zwischen 2 und x in einer Liste aufzählt. Z.B. soll `evens 8 = [2,4,6,8]` gelten. Für ungerade Zahlen und Eingaben < 2 soll `evens` die Ausnahme `Subscript` werfen.

Lösungsvorschlag XMAS.7

```
fun evens 0 = raise Subscript
| evens 1 = raise Subscript
| evens 2 = [2]
| evens x = if x mod 2 = 0 then evens (x div 2) @ [x] else raise Subscript
```

2 Programmiersprachliche Grundlagen

Aufgabe XMAS.8
Geben Sie die Baumdarstellungen der folgenden durch Zeichendarstellungen beschriebenen Phrasen an.

(a) `1 + 2 + 3 + 4`

(b) `1 + 2 * x - y * 3 + 4`

(c) `if x < 3 then 3 else p 3`

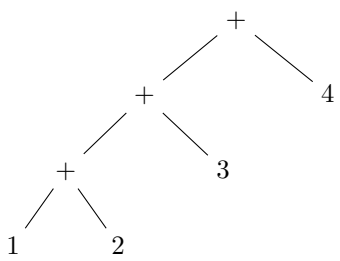
(d) `int * real * (int * real) → bool * unit → real`

(e) `if x > 4 then prozedur 5 else 0`

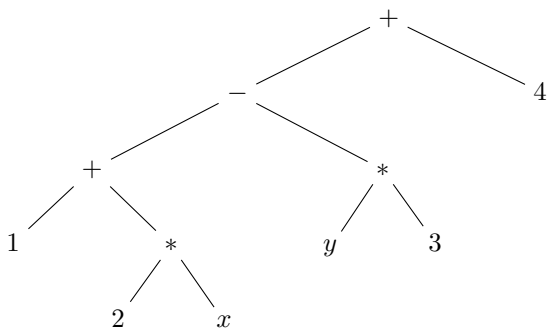
(f)
```
fun fib (n : int) : int =
if n = 0 then 0 else
  if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

Lösungsvorschlag XMAS.8

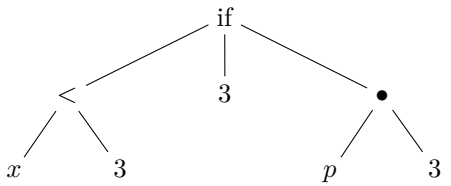
(a)



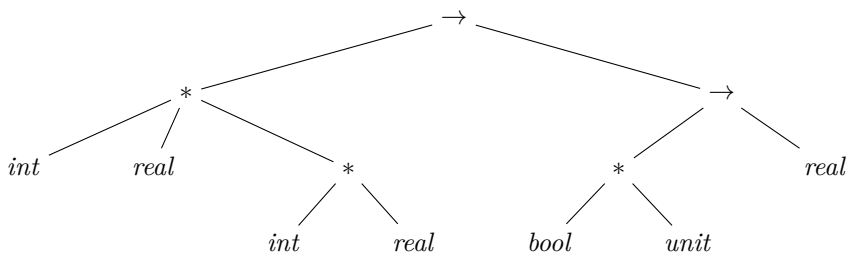
(b)



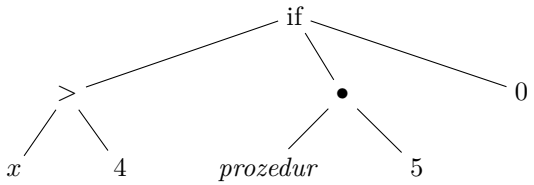
(c)



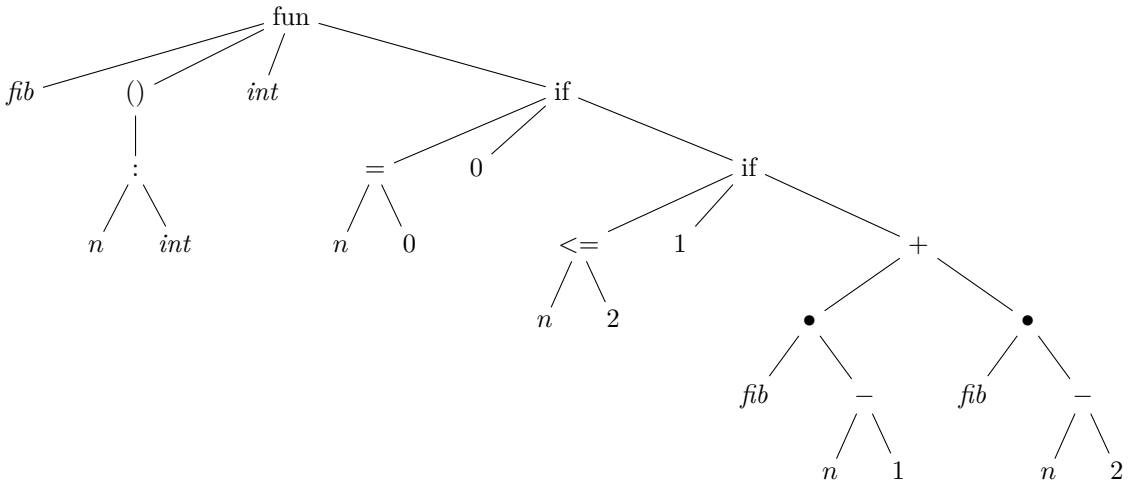
(d)



(e)



(f)

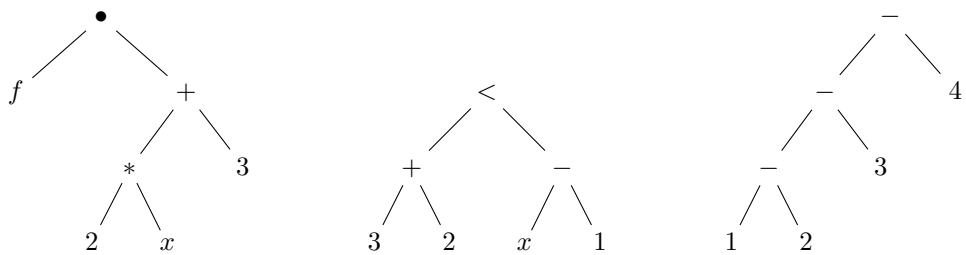


Aufgabe XMAS.9
Welche Umgebung liefert die Ausführung des folgenden Programms in der leeren Umgebung?

```
1 fun f (x:int, y:bool) = if y then x else f(x*2, not y)
2 val x = f(let val f=21 in (f,false) end)
```

Lösungsvorschlag XMAS.9
$[f := (fun f(x, y) = if\ y\ then\ x\ else\ f(x * 2, not\ y), int * int \to int, []), x := 42]$

Aufgabe XMAS.10
Geben Sie für die folgenden Ausdrücke minimal geklammerte Zeichendarstellungen an:



Lösungsvorschlag XMAS.10

(a) `f (2 * x + 3)`

(b) `3 + 2 < x - 1`

(c) `1 - 2 - 3 - 4`

Aufgabe XMAS.11
Geben Sie die freien Bezeichner der folgenden Programme an und geben Sie durch Pfeile die lexikalischen Bindungen an. Welche der folgenden Programme sind geschlossen? Bereinigen Sie die Programme wenn nötig, indem Sie die gebunden Bezeichnerauftreten indizieren.

(a)
```
1 val a = 5
2 val b = 10
3 fun f (x : int) = a + b + c
4 val c = 3
```

(b)
```
1 val a = 0
2 val b = 0
3 val c = 3
4 fun a (x : int) = a + b + c
```

(c)
```
1 val c = 3
2 val c = let
3     val a = 4
4     val a = 5
5     fun f (x : int) = a * x * x + a * x + c
6   in
7   f(4)
8   end
```

(d)
```
1 fun f (x : int) = if x > 4 then let
2     val a = 4
3     val b = 5
4     val c = 3
5   in f(4) end
6 else a * x * x + b * x + c
```

Lösungsvorschlag XMAS.11

(a) Der Bezeichner c in Zeile 3 ist nicht gebunden, damit ist das Programm nicht geschlossen.

(b) Alle Bezeichner sind gebunden und das Programm ist geschlossen.

(c) Alle Bezeichner sind gebunden und das Programm ist geschlossen.

(d) Die Bezeichner a, b und c sind nicht gebunden, daher ist das Programm nicht geschlossen.

Aufgabe XMAS.12
Bestimmen Sie die Typschemata folgender Prozeduren:

(a) `fun f a b = a b`

(b) `fun f a (b, c) d = a(b, c d)`

(c) `val f = fn (x,y) ⇒ fn z ⇒ if x y then y else z`

(d) `val f = fn (x,y) ⇒ (fn z ⇒ if x y then y = y else z) y`

Lösungsvorschlag XMAS.12

(a) $\forall\alpha\beta.$ `f : (` α `→` β `) →` α `→` β

(b) $\forall\alpha\beta\gamma.$ `f : (` α `*` β `→` γ `) →` α `* (` δ `→` β `) →` δ `→` γ

(c) $\forall\alpha.$ `(` α `→ bool) *` α `→` α `→` α

(d) `(bool → bool) * bool → bool`

Aufgabe XMAS.13
Deklarieren Sie polymorphe Prozeduren, die die folgenden Typschemata besitzen, ohne explizite Typangaben wie `(x:int)` zu verwenden.

(a) $\forall\alpha.\ \alpha \to \alpha$

(b) $\forall\alpha,\beta.\ \alpha \to \beta \to \alpha$

(c) $\forall\alpha.\ \alpha \to$ `(real → int)` $\to \alpha$

(d) $\forall\alpha,$‘$\beta.$`(` α `*` α `→ ‘` β `) →` α `→ ‘` β

Lösungsvorschlag XMAS.13

(a) `fun prozedur a = a`

(b) `fun prozedur a b = a`

(c) `fun prozedur a f = if f 3.0 = 2 then a else a`

(d) `fun prozedur f a = f (a,a) = f(a,a)`

3 Bestimmte und Unbestimmte Iteration

Aufgabe XMAS.14
Schreiben Sie mithilfe von `first` eine Prozedur: `divi: int → int → int`, welche für zwei Argumente $x \geq 0$ und $m > 0$ dasselbe Ergebnis liefert wie `x div m`.

Lösungsvorschlag XMAS.14

```
1 fun divi x m = first 1 (fn  a ⇒ (a+1)*m > x)
```

Aufgabe XMAS.15
Bestimmen Sie mithilfe von `first` den kleinsten gemeinsamen Teiler zweier Zahlen größer 1. Für teilerfremde Zahlen – also Zahlen, die keinen gemeinsamen Teiler außer der 1 haben – darf der Aufruf divergieren.

Lösungsvorschlag XMAS.15

```
1 fun kgt (a : int) (b : int) = first 2 (fn s ⇒  a mod s = 0 andalso b mod s
  ↳ = 0)
```

Aufgabe XMAS.16
Schreiben Sie eine Prozedur `itercond` gemäß der folgenden Rekursionsgleichungen:

$$itercond\ n\ s\ d\ t\ f = \begin{cases} s & \text{falls } n \leq 0 \\ itercond\ (n-1)\ (t\ s)\ d\ t\ f & \text{falls } d\ s = true \wedge n > 0 \\ itercond\ (n-1)\ (f\ s)\ d\ t\ f & \text{falls } d\ s = false \wedge n > 0 \end{cases}$$

Der Typ soll `itercond : int` $\rightarrow \alpha \rightarrow$ `(`$\alpha \rightarrow$ `bool)` $\rightarrow$ `(`$\alpha \rightarrow \alpha$`)` $\rightarrow$ `(`$\alpha \rightarrow \alpha$`)` $\rightarrow \alpha$ sein.

Lösungsvorschlag XMAS.16

```
fun itercond n s d t f = if n <= 0 then s
    else (if d s then itercond (n-1) (t s) d t f
  else itercond (n-1) (f s) d t f)
```

Aufgabe XMAS.17
Deklarieren Sie eine Prozedur `iter' : int` $\rightarrow \alpha \rightarrow$ `(`$\alpha \rightarrow \alpha$`)` $\rightarrow \alpha$, sodass sich diese Prozedur für alle Argumente wie das Ihnen bekannte `iter` verhält. Benutzen Sie hierfür nur `iterup` und keine zusätzliche Rekursion.

Lösungsvorschlag XMAS.17

```
fun iter' n s f = iterup 1 n s (fn (a, b) => f b)
```

Aufgabe XMAS.18
Schreiben Sie mit `iter` eine Prozedur `condpart : (`$\alpha \rightarrow$ `bool)` $\rightarrow \alpha$ `list` $\rightarrow \alpha$ `list * `α `list`, die eine Liste gemäß eines Prädikates in zwei Listen aufteilt. In der ersten Liste sollen sich alle Elemente befinden, die das Prädikat erfüllen, in der zweiten all jene, die das Prädikat nicht erfüllen. Beispielsweise soll `condpart (fn x => x mod 2 = 0) [1,3,2,3,5,7,8] = ([8, 2], [7, 5, 3, 3, 1])` gelten.

Lösungsvorschlag XMAS.18

```
fun condpart p xs =
let
  val (x,y,z) = iter (List.length xs) ([],[],xs)
    (fn (ys,zs,(x::xs)) => if p x then (x::ys, zs, xs) else (ys, x::zs, xs))
in
  (x,y)
end
```

Aufgabe XMAS.19
Schreiben Sie eine Prozedur `enumchars : int → int → char list`, die alle durch die Zahlen im Intervall $[m, n]$ beschriebenen Zeichen der Reihenfolge nach auflistet. Überlegen Sie sich, welche Prozedur für Ihre Zwecke am geeignesten ist: `iter`, `iterup` oder `iterdn`.

Lösungsvorschlag XMAS.19
In der Kombination mit dem rechts-klammernden `::`-Operator für Listen eignet sich `iterdn` am besten, da so die aufsteigende Reihenfolge der Ordinalzahlen beibehalten wird:

```
fun chars n m = iterdn n m [] (fn (i,s) => (chr i) :: s)
```

4 Listen und Strings

Aufgabe XMAS.20
Schreiben Sie eine Prozedur `divide : real list → real → real list`, die jede Gleitkommazahl einer Liste durch eine gegebene Zahl teilt.

Lösungsvorschlag XMAS.20

```
1 fun divide xs d = map (fn x ⇒ x/a) xs
```

Aufgabe XMAS.21
Schreiben Sie folgende Prozeduren:

(a) Eine Prozedur `listEvenOdd : int list → (int * bool) list`, so dass

$$listEvenOdd\ [x_1, x_2, \ldots, x_n] = [(x_1, b_1), (x_2, b_2), \ldots, (x_n, b_n)],$$

wobei b_i jeweils *true* sein soll, wenn x_i gerade ist, ansonsten *false* (für alle $i \in \{1, \ldots, n\}$).

(b) Eine Prozedur `countEvenOdd : int list → int * int`, welche zu einer Liste von Zahlen das Tupel (e, o) zurückgibt, sodass e die Anzahl gerader Zahlen in der Liste und o die Anzahl ungerader Zahlen in der Liste angibt.

Lösungsvorschlag XMAS.21

(a)
```
1 fun listEvenOdd    nil     = nil
2 | listEvenOdd (x :: xs) =  if x mod 2 = 0
3   then ((x, true) :: listEvenOdd xs) else ((x, false) :: listEvenOdd xs)
```

(b)
```
1 fun countEvenOdd    nil     = (0, 0)
2 | countEvenOdd (x :: xs) =
3 let
4   val (e,n) = countEvenOdd xs
5 in
6   if x mod 2 = 0 then (e + 1, n) else (e, n + 1)
7 end
```

Aufgabe XMAS.22
Schreiben Sie eine Prozedur `seperate : (α * β) list → α list * β list`, welche aus einer Liste von Tupeln zwei Listen berechnet die jeweils die ersten und zweiten Positionen der Tupel in der selben Reihenfolge enthalten wie die ursprünglichen Liste.

So soll z.B. `seperate [(true, 1),(false,42)]` zu `([true,false],[1,42])` auswerten.

Lösungsvorschlag XMAS.22

```
1 fun separate xs = foldr (fn ((i,s),(is,ss)) ⇒ (i::is,s::ss)) (nil,nil) xs
```

Aufgabe XMAS.23
Schreiben Sie eine Prozedur `substr : int → int → string → string`, die gegeben zwei Zahlen x und y und einen String, einen Teilstring der Länge y, beginnend an Position x ausgibt.

Beispiel: `substr 2 4 " Programmierung "` soll den Teilstring `"ogra"` ausgeben.

(a) Schreiben Sie eine regelbasierte Prozedur `cut : int` $\rightarrow$ α `list` $\rightarrow$ α `list`, die gegeben eine Zahl n und eine Liste die ersten n Elemente der Liste entfernt. Ist die Liste kürzer als n, soll die Ausnahme `Subscript` geworfen werden.

(b) Schreiben Sie eine regelbasierte Prozedur `sub : int` $\rightarrow$ α `list` $\rightarrow$ α `list`, die gegeben eine Zahl n und eine Liste xs eine Liste der ersten n Elemente von xs zurückgibt. Ist xs kleiner als n, dann wird die gesamte Liste zurückgegeben.

(c) Schreiben Sie nun `substr` mithilfe von `cut` und `sub`.

Lösungsvorschlag XMAS.23

(a)
```
fun cut 0 xs = xs
| cut _ nil = raise Subscript
| cut n (x::xr) = cut (n - 1) xr
```

(b)
```
fun sub n nil = nil
| sub 0 xs = nil
| sub n (x::xr) = x::(sub (n-1) xr)
```

(c)
```
fun substr x y s = implode (sub y (cut x (explode s)))
```

Alternative:
```
val substr x y = (implode o sub y o cut x o explode)
```

Aufgabe XMAS.24

Schreiben Sie die folgenden Prozeduren

(a) `init:` α `list` $\rightarrow$ α `list`, die das letzte Element einer Liste entfernt. Bei der leeren Liste soll die leere Liste zurückgegeben werden.

(b) `snoc:` α $\rightarrow$ α `list` $\rightarrow$ α `list`, die ein Element *am Ende* einer Liste einfügt.

auf *drei* verschiedene Weisen:

- rekursiv
- mit `rev`
- mit Faltung

Lösungsvorschlag XMAS.24

(a)
```
fun init nil = nil
| init [x] = nil
| init (x::xr) = x :: init xr

fun init xs = if null xs then xs else rev (tl (rev xs))
fun init xs = if null xs then xs else  foldl op:: nil (tl (foldl op::
↳ nil xs))
```

(b)
```
fun snoc x nil = [x]
| snoc x (y::yr) = y :: snoc x yr

fun snoc x ys = rev (x :: rev ys)
fun snoc x ys = foldr op:: [x] ys
```

Aufgabe XMAS.25
Schreiben Sie eine Prozedur `posx : 'α list → 'α → int option`, die die Position des ersten Auftretens eines Elements x in einer Liste angibt. Enthält die Liste x nicht, soll `NONE` zurückgegeben werden. Schreiben Sie `posx` auf zwei Arten: Regelbasiert und mithilfe von Faltung.

Lösungsvorschlag XMAS.25
Regelbasiert:

```
fun posx' [] a n = NONE
| posx' (x::xr) a n = if x=a then SOME n else posx' xr a (n+1);

fun posx xs a = posx' xs a 0;
```

Mit Faltung:

```
fun posx xs a = #1 (foldr (fn (x,(opt, n)) ⇒ if x=a then (SOME n, n) else (
↳ opt, n+1)) (NONE, 0) xs)
```

Aufgabe XMAS.26
Deklarieren Sie eine Prozedur `undouble : ('α * 'α → order) → 'α list → 'α list`, welche doppelte Elemente aus einer Liste entfernt. Schreiben Sie die Prozedur auf zwei Arten:

(a) mithilfe einer regelbasierten, endrekursiven Hilfsprozedur.

(b) mithilfe von foldl. (Die Reihenfolge in der die Elemente hinterher in der Liste stehen ist egal).

Lösungsvorschlag XMAS.26

```
fun undouble _ a nil = a
| undouble compare a (x::xs) =
  if List.exists (fn y ⇒ compare (x,y) = EQUAL) a
    then undouble compare a xs else undouble compare (x::a) xs

fun undouble' compare xs = foldl
  (fn (x,ys) ⇒ if List.exists (fn y ⇒ compare (x,y) = EQUAL) ys
        then ys else (x::ys))
  nil xs
```

5 Sortieren

Aufgabe XMAS.27
Deklarieren Sie eine Prozedur `intPairCompare : (int * int) * (int * int) → order`, die die lexikalische Ordnung für Paare des Typs `int * int` darstellt. Zum Beispiel soll $[(3,4),(3,5),(4,0)]$ gemäß dieser Ordnung sortiert sein.

Lösungsvorschlag XMAS.27

```
fun intPairCompare ((a, b), (x, y)) =
        case Int.compare (a, x) of EQUAL ⇒ Int.compare (b, y)
        | s     ⇒ s
```

Aufgabe XMAS.28
Schreiben Sie eine Prozedur `listcmp : (α * α → order) → α list * α list → order`, die zwei Listen gemäß einer Vergleichsprozedur lexikalisch vergleicht.

Lösungsvorschlag XMAS.28

```
fun listcmp cmp (nil,nil) = EQUAL
| listcmp cmp (xs, nil) = GREATER
| listcmp cmp (nil, ys) = LESS
| listcmp cmp (x::xr,y::yr) = case cmp (x,y) of
EQUAL ⇒ listcmp cmp (xr,yr)
| s   ⇒ s
```

Aufgabe XMAS.29
Schreiben Sie eine Prozedur `sdsorted : (α * α → order) → α list → α * α option`, die testet, ob eine Liste gemäß eine compare Prozedur *strikt absteigend* sortiert ist. Ist dies der Fall, soll `NONE` zurückgegeben werden. Ist dies nicht der Fall, sollen die ersten beiden Listenelemente zurückgegeben werden, die der Ordnung widersprechen.

Lösungsvorschlag XMAS.29

```
fun sdsorted cmp [] = NONE
| sdsorted cmp [x] = NONE
| sdsorted cmp (x::y::xr) = case cmp(x,y) of
GREATER ⇒ sdsorted cmp (y::xr)
| _ ⇒ SOME(x,y)
```

Aufgabe XMAS.30
Im Folgenden sollen Sie den Sortieralgorithmus Counting-Sort implementieren. Counting-Sort ist ein Sortierverfahren, das Listen von natürlichen Zahlen sortieren kann. Die Idee des Verfahrens ist: Man „zählt" die Häufigkeiten aller Zahlen in der unsortierten Liste, um dann die Liste sortiert auszugeben.

Der Algorithmus funktioniert wie folgt: Ausgehend von einer unsortierten Liste bestehend aus natürlichen Zahlen wird eine „kodierte" Liste erstellt, die genau eins länger ist als der Wert des größten Elements der unsortierten Liste. Dabei repräsentiert das k-te Element dieser kodierten Liste die Häufigkeit des Auftretens der natürlichen Zahl k in der unsortierten Liste. Das Erstellen dieser Liste entspricht also dem „Zählen" der Häufigkeiten.

Um die sortierte Liste zu erhalten, „dekodiert" man die kodierte Liste, indem man jeweils die Zahl k so oft hinzufügt, wie der Wert der kodierten Liste an der Stelle k groß ist.

> **Beispiel:** `[1, 2, 3, 0, 1]` enthält 1 Mal die Zahl 0, 2 Mal die Zahl 1, 1 Mal die Zahl 2 und 1 Mal die Zahl 3, sodass diese kodiert wird zu: `[1, 2, 1, 1]`.
>
> `[1, 2, 1, 1]` wird dann dekodiert zu: `[0, 1, 1, 2, 3]`.

(a) Schreiben Sie eine Prozedur `oneMoreMax : int list → int`, die gegeben eine `int list` die größte Zahl bestimmt und diese um eins erhöht zurückgibt. Wenn die Liste leer ist, soll 0 zurückgegeben werden.

(b) Schreiben Sie eine Prozedur `fill : int → int → int list`, die gegeben zwei natürliche Zahlen k und l eine Liste der Länge l erstellt und diese mit der Zahl k auffüllt.

> **Beispiel:** `fill 3 4` soll `[3, 3, 3, 3]` liefern.

(c) Schreiben Sie eine Prozedur `increment : int * int list → int list`, die, gegeben eine natürliche Zahl n und eine Liste xs, das n-te Element der Liste um eins erhöht. Sie dürfen annehmen, dass n kleiner als die Länge der Liste xs ist.

(d) Schreiben Sie eine Prozedur `encode : int list → int list`, die gegeben eine unsortierte Liste aus natürlichen Zahlen eine „kodierte" Liste erzeugt, also eine Liste, deren k-tes Element angibt, wie oft

der Wert k in der unsortierten Liste auftritt. Sie dürfen die Prozeduren aus den vorigen Aufgabenteilen verwenden, auch wenn Sie diese nicht gelöst haben.

Beispiel: `encode [3, 1, 0, 1, 0, 1]` soll `[2, 3, 0, 1]` liefern.

(e) Schreiben Sie eine Prozedur `decode : int list → int list`, die eine kodierte Liste erhält und diese „dekodiert", also eine sortierte Liste erstellt.

Beispiel: `decode [2, 3, 0, 1]` soll `[0, 0, 1, 1, 1, 3]` liefern.

(f) Schreiben Sie nun `countingSort : int list → int list`. Verwenden Sie dazu die Prozeduren aus den vorherigen Aufgabenteilen.

6 Konstruktortypen

Aufgabe XMAS.31
Betrachten Sie folgenden Datentyp:

```
datatype zaun = Startpfahl of zaun
| Latte of zaun
| Endpfahl
```

(a) Konstruieren Sie zwei unterschiedliche Ausdrücke vom Typ `zaun`, die genau 3 Latten beinhalten und binden Sie diese an `ersterZaun` und `zweiterZaun`.

(b) Schreiben Sie eine Prozedur, welche die Anzahl der Latten in einem Zaun zählt.

(c) Schreiben Sie eine Prozedur, die Ihnen gegeben eine Zahl n, einen Zaun baut, welcher aus n Latten besteht. Jedoch soll nach jeweils 5 Latten ein Pfahl folgen. Außerdem soll eine Ausnahme geworfen werden, wenn die Zahl n nicht durch 5 teilbar ist oder 0 ist.

(d) Erweitern Sie den Datentyp `zaun`, sodass nun jede Latte eine Farbe haben kann. Deklarieren Sie sich vorher einen passenden Datentyp `farbe`.

(e) Schreiben Sie eine Prozedur `colorCheck : farbe → zaun → int`, welche gegeben eine Farbe und einen Zaun überprüft, wie viele Latten dieser Farbe im Zaun vorkommen.

(f) Schreiben Sie eine Prozedur `sameColor : zaun → bool`, welche überprüft, ob alle Latten in einem Zaun dieselbe Farbe haben.

Lösungsvorschlag XMAS.31

(a)
```
val ersterZaun = Startpfahl ( Latte ( Latte ( Latte Endpfahl)))
val zweiterZaun  = Latte (Latte ( Latte Endpfahl))
```

(b)
```
fun lattenCount Endpfahl = 0
|lattenCount (Pfahl x) = 0 + (lattenCount x)
|lattenCount (Latte x) = 1 + (lattenCount x)
```

(c)
```
exception NoZaun

fun zaunHelp n = if n=0 then Endpfahl
else Pfahl (Latte (Latte (Latte (Latte (Latte (zaunHelp (n-5)))))))

fun zaunBauen n = if (n mod 5)<>0 orelse n=0
then raise NoZaun
else zaunHelp n
```

(d)
```
datatype farbe = Rot | Gruen | Blau | Gelb

```

```
3 datatype zaun' = Startpfahl' of zaun'
4 |Latte' of farbe*zaun'
5 |Endpfahl'
```

(e)

```
1 fun colorCheck f Endpfahl' = 0
2 |colorCheck f (Latte' (col,z)) = if f=col then 1+ colorCheck f z
3   else colorCheck f z
4 |colorCheck f (Startpfahl' z) = colorCheck f z
```

(f)

```
1 fun sameColor z =
2 let
3   fun sC a c Endpfahl' = a
4      |sC a c (Latte' (col,z)) = if col=c then sC a c z else false
5      |sC a c (Startpfahl' z) = sC a c z
6 in
7   case z of
8         Endpfahl' ⇒ true
9         |(Latte' (col,z)) ⇒ sC true col z
10        |(Startpfahl' z) ⇒ sameColor z
11 end
```

Aufgabe XMAS.32

Aussagenlogische Formeln können mithilfe von Konstruktortypen modelliert werden. Aussagenlogische Formeln bestehen aus

- Variablen a, b, c, ...
- Konstanten *true* und *false*
- Negationen $\neg\varphi$, wobei φ wieder eine aussagenlogische Formel ist
- Konjunktionen $\varphi_1 \wedge \varphi_2$, wobei φ_1 und φ_2 wieder aussagenlogische Formeln sind
- Disjunktionen $\varphi_1 \vee \varphi_2$, wobei φ_1 und φ_2 wieder aussagenlogische Formeln sind

(a) Deklarieren Sie einen Konstruktortyp `formula`, der aussagenlogische Formeln beschreibt. Geben Sie die Formel $a \wedge \neg(b \vee false)$ in Ihrem Konstruktortypen an.

(b) Deklarieren Sie eine Prozedur `simplify : formula → formula`, die aussagenlogische Formeln gemäß folgender Regeln vereinfacht:

- $\varphi \wedge false \equiv false$ und $false \wedge \varphi \equiv false$
- $\varphi \vee true \equiv true$ und $true \vee \varphi \equiv true$
- $\varphi \vee false \equiv \varphi$ und $false \vee \varphi \equiv \varphi$
- $\varphi \wedge true \equiv \varphi$ und $true \wedge \varphi \equiv \varphi$

Achten Sie darauf, dass $(a \wedge false) \vee (b \vee true)$ korrekterweise zu *true* vereinfacht wird.

(c) Deklarieren Sie ein Typsynonym `env`, das Umgebungen darstellt, die Variablen auf die Konstanten *true* und *false* abbildet.

(d) Deklarieren Sie eine Prozedur `free : env → formula → bool`, die genau dann zu wahr auswertet, wenn eine Umgebung allen Variablen einer Formel einen Wert zuweist.

(e) Deklarieren Sie eine Prozedur `eval : env → formula → bool`, die eine aussagenlogische Formeln gemäß einer Umgebung evaluiert.

```
datatype formula = V of string | C of bool | N of formula | D of formula*
  ↳ formula | K of formula*formula;

fun simplify (V a) = V a
| simplify (C b) = C b
| simplify (N f) = N (simplify f)
| simplify (D(f1, f2)) = (case (simplify f1, simplify f2) of
  (C true, _) ⇒ C true
  | (_, C true) ⇒ C true
  | (C false, f) ⇒ f
  | (f, C false) ⇒ f
  | (f1, f2) ⇒ D(f1, f2))
| simplify (K(f1, f2)) = (case (simplify f1, simplify f2) of
  (C false, _) ⇒ C false
  | (_, C false) ⇒ C false
  | (C true, f) ⇒ f
  | (f, C true) ⇒ f
  | (f1, f2) ⇒ K(f1, f2));

type env = string → bool;

fun free e (V a) = ((e a; true) handle Unbound ⇒ false)
| free e (C _) = true
| free e (N f) = free e f
| free e (D(f1, f2)) = free e f1 andalso free e f2
| free e (K(f1, f2)) = free e f1 andalso free e f2;

fun eval e (V a) = e a
| eval e (C b) = b
| eval e (N f) = eval e f
| eval e (D(f1, f2)) = eval e f1 orelse eval e f2
| eval e (K(f1, f2)) = eval e f1 andalso eval e f2;
```

Aufgabe XMAS.33

Diese Aufgabe setzt den Stoff aus Kapitel 7 voraus. Dieser ist eigentlich nicht relevant für dieses Blatt, die Aufgabe ist trotzdem eine gute Übung zu Konstruktortypen.

Dieter Schlau ist in Weihnachtsstimmung und möchte seinen Weihnachtsbaum schmücken. Da er sich aber keine Gedanken über die Anordnung der einzelnen Dekorationsgegenstände machen will, möchte er eine Prozedur schreiben, die für ihn seinen Baum dekoriert. Dazu will er einen markierten Baum über Listen von Dekorationsgegenständen benutzen. Sein bisheriger Ansatz sieht dabei folgendermaßen aus:

- Zunächst deklariert er sich einen Enumerationstyp mit verschiedenen Dekorationsgegenständen:

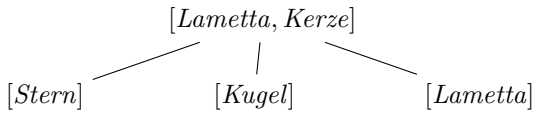
```
datatype deko = Kugel | Kerze | Stern | Lametta
```

- Danach konstruiert er sich seinen Baum mittels einer rekursiven Konstruktionsvorschrift. Dabei steht an erster Komponente die Liste der Dekorationsgegenstände, die zweite Komponente wird wie gehabt für die Baumstruktur verwendet. Das Ganze wird dabei mithilfe des Konstruktors `C` beschrieben:

```
datatype christbaum = C of deko list * christbaum list
```

Helfen Sie Dieter seine Implementierung zu vervollständigen, indem Sie die folgenden Aufgaben lösen.

(a) Überführen Sie folgende Baumdarstellung in einen Ausdruck.



(b) Deklarieren Sie eine Prozedur `convert : tree → christbaum`, die einen nicht makierten Baum vom Typ `tree` in einen Baum des Types `christbaum` überführt. Dabei soll der Ergebnisbaum keine Dekorationsgegenstände enthalten, das heißt die Liste an der ersten Komponente ist immer leer.

(c) Schreiben Sie zum Dekorieren eine Prozedur `decorate : deko → christbaum → christbaum`, die zufällig zu der in jedem Knoten des Baumes enthaltenen `deko list` den übergebenen Dekorationsgegenstand entweder hinzugefügt, oder nicht. Um den Zufall zu realisieren, können Sie die Prozedur `randbool : unit → bool` verwenden, die zufällig zu () entweder `true` oder `false` zurück gibt.

(d) Nun kann Dieter endlich seinen Weihnachtsbaum dekorieren. Schreiben Sie für ihn eine weihnachtliche Prozedur `christmastree : unit → christbaum`, die einen Baum nach dem in der Abbildung gegebenen Schema generiert (das heißt beide Bäume sollen dieselbe Gestalt besizten). Dabei soll sich nur in der Wurzel des Baumes ein Stern befinden, und in den restlichen Knoten eine zufällige Verteilung beliebiger *anderer* Dekorationsgegenstände. Jeder Dekorationsgegenstand darf dabei pro Knoten nicht öfter als einmal vorkommen. Sie dürfen alle bisher deklarierten Prozeduren verwenden.

Lösungsvorschlag XMAS.33

(a)
```
C ([Lametta, Kerze], [C ([Stern], []), C ([Kugel], []), C ([Lametta],
↳ [])])
```

(b)
```
fun convert (T ts) = C ([], map convert ts)
```

(c)
```
fun decorate x (C (ds, ts)) = C (if randbool() then x :: ds else ds, map
↳  (decorate x) ts)
```

(d)
```
fun christmastree () = let
val baum = C ([], [])
val baum = C ([], [baum, baum, baum])
val baum = C ([], [baum, baum, baum])
val dekoriert = decorate Kugel (decorate Kerze (decorate Lametta baum))
in
C ([Stern], [dekoriert])
end
```

Frohe Weihnachten und ein gutes neues Jahr!