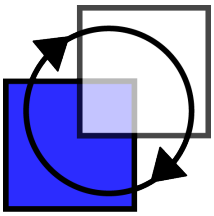
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

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



Programmierung 1 (WS 2020/21)
Aufgaben für die Übungsgruppe E (Lösungsvorschläge)

Hinweis: Diese Aufgaben wurden von den Tutoren für die Übungsgruppe erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

Faltung

Aufgabe TE.1 *(Keine Einser)*
Schreiben Sie eine Prozedur `noOne : int list → int list` mithilfe von Faltung, welche zu einer Liste von Zahlen, diejenigen Auftreten entfernt, welche die Ziffer 1 enthalten. `noOne [1,22,11,353,201]` soll zu `[22,353]` auswerten.

Lösungsvorschlag TE.1

```
fun noOne xs = let
  fun containsOne x = if x mod 10 = 1 then true else
      if x div 10 = 0
      then false
      else containsOne (x div 10)
in
  foldr (fn (i,a) ⇒ if containsOne i then a else i::a) nil xs
end
```

Aufgabe TE.2

(a) Schreiben Sie eine Prozedur `interval : int → int → int → bool` die für eine Zahl x prüft, ob sie im Intervall $[n, m]$ liegt.
(b) Verwenden Sie `interval`, um eine Prozedur `inSum : int list → int → int → int` zu schreiben, die gegeben n und m alle Zahlen aus der Liste zwischen n und m aufsummiert.

Lösungsvorschlag TE.2

(a)
```
fun interval x n m = x>=n andalso x<=m
```

(b)
```
fun inSum xs n m = foldl (fn (x, s) ⇒ if interval x n m then x + s else s) 0 xs
```

Aufgabe TE.3 *(Challenge)*
Schreiben Sie eine Prozedur `at:` α `list → int →` α, die analog zu `nth` das n-te Element der Liste liefert. Verwenden Sie hierfür Faltung mit einer Prozedur als Akku. 🤔

Lösungsvorschlag TE.3

```
fun at xs = foldr (fn (x, f) ⇒ (fn 0 ⇒ x | n ⇒ f (n − 1))) (fn _ ⇒ raise Subscript) xs
```

Vergleichsprozeduren

Aufgabe TE.4 *(Äpfel und Birnen vergleichen)*
Dieter Schlau möchte seinem alten Lehrer endlich beweisen, dass er doch Äpfel und Birnen vergleichen kann. Helfen Sie ihm, eine Prozdur `fruitcompare : string → string → order` zu deklarieren, die zwei Strings miteinander vergleicht. Dabei soll für gleiche Früchte und für zwei nicht-Früchte `EQUAL` ausgegeben werden, eine Birne größer als ein Apfel sein und alle anderen Strings kleiner als die beiden sein.

Lösungsvorschlag TE.4

```
fun fruitcompare "Apfel" "Apfel" = EQUAL
| fruitcompare "Apfel" "Birne" = LESS
| fruitcompare "Birne" "Birne" = EQUAL
| fruitcompare "Birne"  "Apfel" = GREATER
| fruitcompare "Apfel" _ = GREATER
| fruitcompare "Birne" _ = GREATER
| fruitcompare _ "Apfel" = LESS
| fruitcompare _ "Birne" = LESS
| fruitcompare _  _ = EQUAL;
```

Aufgabe TE.5 *(Aus Groß mach Klein)*
Schreiben Sie eine Prozedur `cmpinvert : ( α * α → order) → α * α → order`, welche eine compare-Prozedur invertiert. So soll beispielsweise `cmpinvert Int.compare (3,4)` zu `GREATER` auswerten.

Lösungsvorschlag TE.5

```
fun cmpinvert cmp (a,b) = case cmp(a,b) of
                            GREATER ⇒ LESS
                          | LESS ⇒ GREATER
                          | x ⇒ x
```

Aufgabe TE.6 *(Listige Listen)*
Schreiben Sie eine polymorphe Prozedur `listcmp : (α * α → order) → ( α list * α list → order)`, die zwei Listen gemäß einer Vergleichsprozedur lexikalisch vergleicht. Wenn sich die Listen in ihrer Größe unterscheiden, jedoch die ersten Werte gleich sind, ist eine Liste kleiner/größer wenn sie weniger/mehr Elemente als die andere Liste hat.

Lösungsvorschlag TE.6

```
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
```

Sortieren

Aufgabe TE.7 *(Bubblesort)*
In der folgenden Aufgabe soll *Bubblesort* implementiert werden. Der Algorithmus funktioniert wie folgt:

Die Elemente steigen wie Blasen gemäß ihrer Größe nach oben. Jede Blase steigt dabei so lange auf, bis sie auf eine kleinere stößt. Diese kleinere Blase steigt dann so lange auf, bis sie am Ende ankommt oder ihrerseits eine kleinere anstößt. Dies wird so lange wiederholt, bis die Liste sortiert ist. Dabei sollen alle Mehrfachauftreten aus der Liste entfernt werden, die Liste soll also *strikt absteigend* sortiert werden. Vorhergehende Teilaufgaben dürfen verwendet werden, auch wenn sie nicht gelöst wurden.

> **Beispiel:** `[5, 7, 10, 5, 1]`
> Zunächst werden die 5 und die 7 miteinander verglichen. Da die 5 kleiner als die 7 ist, werden die beiden Elemente vertauscht. Dann werden die 5 und die 10 miteinander verglichen. Da $10 > 5$, vertauscht man die Elemente und vergleicht anschließend die 5 mit der 5. Da man in diesem Fall 2 gleiche Elemente hat, kann man eines davon aus der Liste entfernen, usw. Im Folgenden sehen Sie den Zustand der Liste `[5, 7, 10, 5, 1]` nach einem `bubble`-Aufruf:
>
> `[7, 10, 5, 1]`
>
> Nach dem zweiten Aufruf:
>
> `[10, 7, 5, 1]`

(a) Geben Sie den Zustand der Liste `[4, 2, 7, 1, 9, 4, 8]` nach jedem `bubble`-Aufruf an, bis die Liste sortiert ist.

(b) Schreiben Sie eine Prozedur `bubble : ('α * 'α → order) → 'α list → 'α list`, die beim ersten Element der Liste beginnt und dieses, solange es kleiner als das darauf folgende Element ist, nach oben verschiebt. Sobald man auf ein gleich großes Element trifft, entfernt man es aus der Liste. Trifft man auf ein kleineres Element, wird dieses Element weiter nach oben verschoben.

(c) Schreiben Sie eine Prozedur `isSorted : ('α * 'α → order) → 'α list → bool`, die prüft, ob eine Liste gemäß einer übergebenen Ordnung strikt absteigend sortiert ist.

(d) Schreiben Sie jetzt eine Prozedur `bubbleSort : ('α * 'α → order) → 'α list → 'α list`, die den anfangs beschriebenen Algorithmus umsetzt.

Lösungsvorschlag TE.7

(a) (i) `[4, 7, 2, 9, 4, 8, 1]`

(ii) `[7, 4, 9, 4, 8, 2, 1]`

(iii) `[7, 9, 8, 4, 2, 1]`

(iv) `[9, 8, 7, 4, 2, 1]`

(b)
```
1 fun bubble cmp nil = nil
2   | bubble cmp [x] = [x]
3   | bubble cmp (x::y::yr) = case cmp(x,y) of EQUAL ⇒ bubble cmp (y::yr)
4                                            | LESS ⇒ y::(bubble cmp (x::yr))
5                                            | _ ⇒ x::(bubble cmp (y::yr))
```

(c)
```
1 fun isSorted cmp nil = true
2   | isSorted cmp [x] = true
3   | isSorted cmp (x::y::yr) = case cmp(x,y) of GREATER ⇒ isSorted cmp (y::yr)
4                                              | _ ⇒ false
```

(d)
```
1 fun bubbleSort cmp xs = if (isSorted cmp xs) then xs else bubbleSort cmp (bubble cmp xs)
```

Aufgabe TE.8 *(Quicksort)*
Im Folgenden soll die Sortierprozedur Quick-Sort realisiert werden.

Der dahinter stehende Algorithmus basiert auf folgender Idee: Das erste Element der Liste wird herausgegriffen. Anschließend wird die restliche Liste in zwei Teillisten zerteilt, wobei die eine Liste nur die Elemente kleiner dem ersten Element und die zweite nur diejenigen enthält, die größer oder gleich diesem sind. Diese Teillisten werden wieder rekursiv mittels Quick-Sort sortiert.

(a) Verdeutlichen Sie sich die Funktionsweise von Quick-Sort anhand kleinerer Beispiele und diskutieren Sie die einzelnen Schritte des Algorithmus.

(b) Schreiben Sie eine Prozedur `partition : (`α `* `α `→ order) →`α `→`α `list →`α `list ↳ * `α `list`, die zu einer gegebenen Liste und einem Wert ein Paar von zwei Listen zurückliefert, wobei die erste Liste nur die Elemente kleiner (gemäß der übergebenen Ordnung) und die andere nur die Elemente größer gleich dem Wert enthält.

(c) Schreiben Sie eine Prozedur `qsort : (`α `* `α `→ order) →`α `list →`α `list`, die einer Liste gemäß einer gegebenen Ordnung sortiert. Dazu soll die Liste mithilfe von `partition` geteilt werden, die Teillisten sortiert und wieder sinnvoll zu einer sortierten Liste zusammengesetzt werden. Überlegen Sie sich vorher sorgfältig die Basisfälle Ihrer Prozedur.

Lösungsvorschlag TE.8

(b)
```
fun partition compare x xr =
  foldl (fn (a, (xs, ys)) ⇒
      case compare(a, x) of
        LESS ⇒ (a::xs, ys)
      | _    ⇒ (xs, a::ys))
    (nil, nil) xr
```

(c)
```
fun qsort compare nil = nil
  | qsort compare (x::xr) =
    let
      val (xs, ys) = partition compare x xr
    in
      qsort compare xs @ [x] @ qsort compare ys
    end
```

Aufgabe TE.9 *(Barkeeper im Lockdown)*
Dieter Schlau hat von seinem Freund Chen Kaus, einem Barkeeper, der sich während des Lockdowns damit beschäftigt, alle Flaschen nach ihrem Füllstand zu sortieren, vom Sortieralgorithmus Shakersort (manchmal auch Cocktailsort genannt) erfahren und möchte eine leichte Abwandlung nun gemeinsam mit Ihnen in SML implementieren.
Die Grundidee ist es, eine Liste abwechselnd von oben nach unten und von unten nach oben durchzugehen und dabei benachbarte Elemente vergleichen und ggf. zu vertauschen. Dies wird wiederholt bis keine Vertauschungen mehr stattfinden.

(a) Implementieren Sie zunächst eine Prozedur `upwards : int → int → int list → int list ↳ (int*int list)`, wobei das erste Argument c ein Zähler für die erfolgten Vertauschungen ist, das zweite Argument *last* das zuletzt betrachtete Element, das dritte Argument die noch zu bearbeitende Teilliste und das vierte Argument die bereits bearbeitete Teilliste.
Im dem Fall, dass das dritte Argument die leere Liste ist, sind Sie fertig und geben ein Tupel bestehend aus c und der bearbeiteten Liste, zu der noch *last* hinzugefügt wird, zurück.
Sonst vergleichen Sie *last* mit dem ersten Element der noch zu bearbeitenden Liste und gehen in Rekursion. Dabei übergeben Sie das größere Element als neues *last* und fügen das andere in die bereits bearbeite Liste ein. Ggf. müssen Sie auch den Zähler inkrementieren.

> **Beispiel:** Der Aufruf `upwards 0 3 [2,5,6,1] nil` hat die Rekursionsfolge `upwards 0 3 [2,5,6,1] nil → upwards 1 3 [5,6,1] [2] → upwards 1 5 [6,1] [3,2] → upwards 1 6 [1] [5,3,2] → upwards 2 6 nil [1,5,3,2]` und soll das Tupel `(2, [6, 1, 5, 3, 2])` liefern.

(b) Deklarieren Sie nun die entsprechende Prozedur `downwards : int → int → int list → int ↳ list → (int*int list)`, die wie `upwards` funktioniert, nur dass sie im rekursiven Fall das größere

Element in die bereits bearbeitete Liste einfügt.

(c) Implementieren Sie nun die Prozedur `shakersort : int list → int list` mithilfe von `upwards` und `downwards` sowie der Hilfsprozedur `shakersort '`, die `upwards` und `downwards` abwechselnd aufruft, solange Vertauschungen stattgefunden haben. Denken sie daran, die Liste zu reversieren, bevor Sie sie als Endergebnis ausgeben, falls die zuletzt aufgerufene Prozedur `upwards` war.

Hinweis: Verwenden Sie ein `bool`, um in `shakersort '` zu wissen, in welche Richtung Sie im nächsten Schritt gehen sollen.

Lösungsvorschlag TE.9

(a)
```
fun upwards c last nil ys = (c, last::ys)
  | upwards c last (x::xr) ys = if last > x then
                                   upwards (c+1) last xr (x::ys)
                                else
                                   upwards c x xr (last::ys)
```

(b)
```
fun downwards c last nil ys = (c, last::ys)
  | downwards c last (x::xr) ys = if last < x then
                                     downwards (c+1) last xr (x::ys)
                                  else
                                     downwards c x xr (last::ys)
```

(c)
```
fun shakersort xs =
 let
  fun shakersort' (_, (_, nil)) = nil
    | shakersort' (true, (c, (x::xr))) = if c = 0 then
                                            x::xr
                                         else
                                            shakersort'(false, (upwards 0 x xr nil))
    | shakersort' (false, (c, (x::xr))) = if c = 0 then
                                             rev (x::xr)
                                          else
                                             shakersort'(true, (downwards 0 x xr nil))
 in
  shakersort' (true, (~1, xs))
 end
```

Aufgabe TE.10 *(slowsort)*

Dieter Schlau ist ganz begeistert von Sortieralgorithmen, leider gehen ihm die in der Vorlesung vorgestellten Prozeduren viel zu schnell. Nach einiger Recherche hat er `Slowsort` für sich entdeckt.
`Slowsort` funktioniert folgendermaßen:
Zunächst wird die Liste in zwei Teillisten unterteilt, die beide rekursiv sortiert werden. Danach wird das größte Element der beiden Listen bestimmt. Das größere Element der beiden gefundenen Werte wird an das Ende gehängt. Nach dem gleichen Prinzip wird die verbleibende Liste, ohne das gefundene größte Element, sortiert. gehen Sie hierbei Folgendermaßen vor:

(a) Schreiben Sie eine Prozedur `last: int list → int`, welche das letzte Element einer Liste bestimmt. Bei einer leeren Liste soll sie die Ausnahme `empty` werfen.

(b) Schreiben Sie eine Prozedur `removeLast int list → int list`, welche das letzte Element einer Liste entfernt. Bei einer leeren Liste soll sie die Ausnahme `empty` werfen.

(c) Schreiben Sie abschließend die Prozedur `slowsort int list → int list`. Verwenden sie hierfür neben `last` und `removeLast` noch die Prozedur `split`, welche Sie bereits aus der Vorlesung kennen.

(d) Nun möchte Dieter mit seiner tollen Prozedur auch Listen mit anderen Typen als `int list` sortieren. Welche Änderungen muss er dafür machen?

Lösungsvorschlag TE.10

(a)

```
fun last nil = raise Empty
| last (x::nil) = x
| last (x::xr) = last xr
```

(b)

```
fun removeLast nil = raise Empty
| removeLast (x::nil) = nil
| removeLast (x::xr) = x :: removeLast(xr)
```

(c)

```
fun slowsort nil = nil
| slowsort (x::nil) = [x]
| slowsort xs =
let
        val (ys,zs) = split xs
        val (ys, zs) = (slowsort(ys), slowsort(zs))
        val y = last ys
        val z = last zs
in
        if y > z then (slowsort (removeLast(ys) @ zs)) @ [y] else (slowsort (ys @
        ↳ removeLast(zs))) @ [z]
end
```

```
fun slowsort cmp nil = nil
| slowsort cmp (x::nil) = [x]
| slowsort cmp xs =
let
        val (ys,zs) = split xs
        val (ys, zs) = (slowsort cmp (ys), slowsort cmp (zs))
        val y = last ys
        val z = last zs
in
        case cmp(y,z) of GREATER ⇒ (slowsort cmp (removeLast(ys) @ zs)) @ [y]
        | _ ⇒ (slowsort cmp (ys @ removeLast(zs))) @ [z]
end
```

Konstruktoren

Aufgabe TE.11 *(3D-Drucker)*
Dieter Schlau hat sich einen 3D-Drucker zugelegt. Mit diesem möchte er verschiedene Formen drucken, nämlich Würfel, Kugeln, Zylinder und Prismen. Um immer zu wissen, wie viel Filament (3D-Drucker'tinte') er braucht, will er ein Programm schreiben, das ihm die erforderlichen Materialmengen berechnet. Dabei benötigt er Ihre Hilfe.

(a) Deklarieren Sie einen geeigneten Konstruktortypen, mit dessen Hilfe Sie Würfel, Kugeln, Zylinder und Prismen darstellen können. Ein Würfel wird dabei durch die Seitenlänge, eine Kugel durch ihren Radius, ein Zylinder durch den Radius der kreisförmigen Grundfläche und die Höhe, und ein Prisma durch die Seitenlängen des Dreiecks, das die Grundfläche ist, sowie die Höhe dargestellt. Alle Längen sollen dabei vom Typ `real` sein.

(b) Schreiben Sie eine Prozedur, die für einen gegebenen Körper den Oberflächeninhalt dieses Körpers ausrechnet.

(c) Schreiben Sie eine weitere Prozedur, die das Volumen berechnet.

(d) Schreiben Sie nun zwei Vergleichsprozeduren, die zwei Körper bezüglich Oberflächeninhalt bzw. Volumen vergleicht. Verwenden Sie `Real.compare`.

Hinweis: Auf Seite 114 im Buch finden Sie eine Rechenvorschrift für den Flächeninhalt eines durch 3 Seitenlngen beschriebenen Dreiecks. Die Konstante π steht unter `Math.pi` *zur Verfügung.*

Lösungsvorschlag TE.11

(a)

```
datatype shape3d =
    Cube of real (* Seitenlänge *)
  | Sphere of real (* Radius *)
  | Zylinder of real * real (* Radius, Höhe *)
  | Prism of (real * real * real) * real (* Dreieck (a,b,c), Höhe *)
```

(b)

```
fun surface (Cube a) = 6.0 * a * a
  | surface (Sphere r) = 4.0 * Math.pi * r * r
  | surface (Zylinder (r,h)) = 2.0 * Math.pi * r * (r + h)
  | surface (Prism ((a,b,c),h)) =
      let
          val s = (a + b + c) / 2.0
      in
          (a + b + c) * h + 2.0 * Math.sqrt(s * (s - a) * (s - b) * (s - c))
      end
```

(c)

```
fun volume (Cube a) = a * a * a
  | volume (Sphere r) = 4.0 / 3.0 * Math.pi * r * r * r
  | volume (Zylinder (r,h)) = Math.pi * r * r * h
  | volume (Prism ((a,b,c),h)) = let
                                     val s = (a + b + c) / 2.0
                                 in
                                     h * Math.sqrt(s * (s - a) * (s - b) * (s - c))
                                 end
```

(d)

```
fun compareVolume (a,b) = Real.compare(volume a, volume b)
fun compareSurface (a,b) = Real.compare(surface a, surface b)
```

Aufgabe TE.12 *(Sprachen-Masterrace)*
In dieser Aufgabe möchten wir Ihnen näherbringen, welche Hierarchie (unserer Meinung nach) unter einigen ausgewählten Programmiersprachen herrscht. Deklarieren Sie dazu einen Datentyp `Lang` und schreiben Sie eine Vergleichsprozedur `langCompare : Lang * Lang → order`, welche folgende Beziehung wiedergibt:

`PHP < C < Go < Java < CSCHARF < Brainfuck < Haskell < Oz = Alice < StandardML`

Lösungsvorschlag TE.12

```
datatype lang = PHP | C | Go | Java | CSCHARF | Brainfuck | Haskell | Oz | Alice | StandardML

fun langRank PHP = 0
    | langRank C = 1
    | langRank Go = 2
    | langRank Java = 3
    | langRank CSCHARF = 4
    | langRank Brainfuck = 5
    | langRank Haskell = 6
    | langRank Oz = 7
    | langRank Alice = 7
    | langRank StandardML = 8

fun langCompare (a, b) = Int.compare(langRank a, langRank b)
```

Aufgabe TE.13 *(Ein Dreirad für Dieter)*
Dieter Schlau möchte sich ein neues Gefährt zulegen, also schaut er sich das Magazin des lokalen Autohauses an. Um schneller zu sehen, wie viel eine bestimmte Konfiguration kostet, möchte er ein SML Programm schreiben, das ihm dabei hilft. Leider weiß er nach der Deklaration folgender Datentypen nicht mehr weiter. Helfen Sie Dieter, sein Programm zu vervollständigen!

```
datatype Farbe = Blau | Rot | Schwarz | Rosa
datatype Marke = BMW | Opel | Porsche
datatype Ausstattung = Klimaanlage | Navi | Parkhilfe | Akustikfunktion
datatype Fahrzeug = Auto of (Marke * Farbe * Ausstattung list) | Dreirad of Farbe
```

Der Preis eines Autos berechnet sich nach folgender Formel:

$$\mathit{Grundpreis} + \mathit{Ausstattungsmultiplikator} \cdot \mathit{Farbmultiplikator} \cdot \mathit{Ausstattungspreis}$$

Der Preis eines Dreirads ergibt sich folgendermaßen:

$$100€ + 50€ \cdot \mathit{Farbmultiplikator}$$

Im Prospekt finden sich diese Angaben:

Grundpreis:	BMW: 15000€, Opel: 5000€, Porsche: 60000€
Ausstattungsmultiplikator:	BMW: 2.0, Opel: 1.0, Porsche: 3.5
Farbmultiplikator:	Blau: 0.6, Rot: 1.0, Schwarz: 1.0, Rosa: 2.0
Ausstattung:	Klimaanlage: 500€, Navi: 1500€, Parkhilfe: 2300€, Akustikfunktion: 3200€

(a) Schreiben Sie eine Prozedur `farbmul : Farbe → real`, die den Farbmultiplikator für eine Farbe berechnet.

(b) Schreiben Sie eine Prozedur `ausstattung : Ausstattung list → real`, die die Summe der Ausstattungskosten für eine Ausstattungsliste berechnet. (Ist eine Ausstattung mehrfach in der Liste enthalten, so muss sie auch mehrfach bezahlt werden)

(c) Schreiben Sie eine Prozedur `kosten : Fahrzeug → real`, das für ein Fahrzeug die Kosten berechnet. Zum Beispiel soll `kosten (Dreirad Rosa)` zu 200.0, `kosten (Dreirad Rot)` zu 150.0, `kosten (Auto (BMW, Rot, []))` zu 15000.0 und `kosten (Auto (Opel, Blau, [Navi, Parkhilfe, Parkhilfe]))` zu 8660.0 auswerten.

Lösungsvorschlag TE.13

(a)

```
fun farbmul Blau = 0.6
  | farbmul Rosa = 2.0
  | farbmul _    = 1.0
```

(b)

```
fun ausstattung xs = foldl (fn (Klimaanlage,a) ⇒ a + 500.0
                             | (Navi, a) ⇒ a + 1500.0
                             | (Parkhilfe, a) ⇒ a + 2300.0
                             | (Akustikfunktion, a) ⇒ a + 3200.0) 0.0 xs
```

(c)

```
fun kosten (Dreirad farbe) = 100.0 + 50.0 * farbmul farbe
  | kosten (Auto (BMW, f, a)) = 15000.0 + 2.0 * farbmul f * ausstattung a
  | kosten (Auto (Opel, f, a)) = 5000.0 + 1.0 * farbmul f * ausstattung a
  | kosten (Auto (Porsche, f, a)) = 60000.0 + 3.5 * farbmul f * ausstattung a
```

Aufgabe TE.14 *(Aufgeben ist keine Option)*
Es ist in SML möglich, polymorphe Datentypen zu deklarieren. Ein Beispiel hierfür ist der sogenannte Option-Typ:

```
datatype α option = SOME of α | NONE
```

Mit ihm kann man beispielsweise Prozeduren bauen, die nur manchmal einen Wert zurückgeben - dann verpackt in `SOME`, ansonsten `NONE`.

(a) Schreiben Sie eine Prozedur, die wie `div` zwei Zahlen a und b durcheinander teilt, jedoch mit `option`s arbeitet. Demzufolge soll, solange die Division erfolgreich funktioniert, `SOME (a div b)` zurückgegeben werden, andernfalls `NONE` bei Division durch 0.

(b) Schreiben Sie die Listenprozeduren `hd` und `tl` so, dass sie, anstatt Ausnahmen zu werfen, mit `option`s arbeiten.

(c) Mithilfe von `option`s können wir Partialordnungen modellieren, anstatt wie bisher nur Totalordnungen. Bei einer Totalordnung ist jedes Element mit jedem vergleichbar und entweder größer, kleiner oder gleich. Bei einer Partialordnung können nun zwei Elemente auch nicht vergleichbar sein.

(i) Welchen Typ haben nun partielle Vergleichsprozeduren?

(ii) Schreiben Sie eine solche partielle Vergleichsprozedur, die wie `Int.compare` funktioniert, jedoch nur gerade mit geraden oder ungerade mit ungeraden Zahlen vergleichen kann.

Keine Ihrer Funktionen darf Ausnahmen werfen.

Lösungsvorschlag TE.14

(a)

```
fun divi a 0 = NONE
  | divi a b = SOME (a div b)
```

(b)

```
fun hd nil = NONE
  | hd (x::_) = SOME x
fun tl nil = NONE
  | hd (_::xr) = SOME xr
```

(c) (i) $\forall\alpha : \alpha$ * α → order option

(ii)

```
fun compareEven (a,b) = if a mod 2 <> b mod 2 then NONE else SOME (Int.compare (a,b))
```

Aufgabe TE.15 *(Terminkalender)*
Nachdem Dieter Schlau bisher zweimal vergessen hat, wann die Programmierung 1-Vorlesung stattfindet, will er sich einen Terminkalender in SML schreiben, der ihm beim Erinnern hilft.

(a) Deklarieren Sie geeignete Konstruktortypen für die 12 Monate und 7 Wochentage.

(b) Schreiben Sie eine Prozedur `daysOfMonth : bool → month → int`, die abhängig davon, ob ein Jahr ein Schaltjahr ist, die Anzahl der Tage eines jeden Monats zurückgibt. Das erste Argument gibt an, ob ein Jahr ein Schaltjahr ist.

(c) Schreiben Sie eine Prozedur `isLeapYear : int → bool`, die angibt, ob ein Jahr ein Schaltjahr ist. Zur Erinnerung: Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist. 2020 ist beispielsweise ein Schaltjahr. Ist das Jahr jedoch auch durch 100 teilbar, so ist es kein Schaltjahr (z.B. 1900) - es sei denn, es ist auch durch 400 teilbar, wie beispielsweise 2000, dann ist es ein Schaltjahr.

(d) Dieter Schlau hat sich nun aus dem dCMS die Liste von Tagen, an denen Programmierung 1-Vorlesung ist, abgetippt. Er stellt dabei Tage als Tupel `int * month * int` dar, z.B. den ersten Januar dieses Jahres als `(1, JANUARY, 2019)`. Leider ist die Liste ungeordnet. Geben Sie eine Vergleichsprozedur vom Typ `(int * month * int) * (int * month * int) → order` an, die zwei Daten vergleicht. Verwenden Sie dazu eine Hilfsprozedur, die den Index eines Monats liefert.

(e) Deklarieren Sie eine Ausnahme `InvalidDate`. Ändern Sie dann Ihre Vergleichsprozedur so ab, dass sie auch überprüft, ob die beiden Daten jeweils gültig sind. Wenn sie das nicht sind, soll die Prozedur die Ausnahme `InvalidDate` werfen.

Lösungsvorschlag TE.15

(a)

```
datatype day = MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY | SATURDAY | SUNDAY
datatype month = JANUARY | FEBRUARY | MARCH | APRIL | MAY | JUNE
               | JULY | AUGUST | SEPTEMBER | OCTOBER | NOVEMBER | DECEMBER
```

(b)

```
fun daysOfMonth true FEBRUARY = 29
    | daysOfMonth false FEBRUARY = 28
    | daysOfMonth _ APRIL = 30
    | daysOfMonth _ JUNE = 30
    | daysOfMonth _ SEPTEMBER = 30
    | daysOfMonth _ NOVEMBER = 30
    | daysOfMonth _ _ = 31
```

(c) `fun isLeapYear k = k mod 4 = 0 andalso (k mod 100 <>0 orelse k mod 400 = 0)`

(d)

```
fun monthIndex JANUARY = 0
    | monthIndex FEBRUARY = 1
    | monthIndex MARCH = 2
    .....
    | monthIndex NOVEMBER = 10
    | monthIndex DECEMBER = 11
fun cmpDate ((d1, m1, y1), (d2, m2, y2)) =
    (case Int.compare (y1, y2) of
        EQUAL ⇒ (case Int.compare ((monthIndex m1), (monthIndex m2)) of
                EQUAL ⇒ Int.compare (d1, d2)
                    |   k ⇒ k)
        |   k ⇒ k)
```

(e)

```
exception InvalidDate
fun isValidDate (d, m, y) = d > 0 andalso d <= daysOfMonth (isLeapYear y) m
fun cmpDate ((d1, m1, y1), (d2, m2, y2)) =
    if isValidDate (d1, m1, y1) orelse isValidDate (d2, m2, y2)
    then (case Int.compare (y1, y2) of
                EQUAL ⇒ (case Int.compare ((monthIndex m1), (monthIndex m2)) of
                        EQUAL ⇒ Int.compare (d1, d2)
                    |   k ⇒ k)
            | k ⇒ k)
    else raise InvalidDate
```

Aufgabe TE.16 *(MyList)*
Wir wollen Listen selbst implementieren.

(a) Erstellen Sie einen Konstruktortyp α `mylist`, der zwei Konstruktoren besitzt: `Leer` für die leere Liste und `Comb`, der wie `::` funktioniert.

(b) Schreiben Sie für Ihre Listen die Prozeduren:

```
length : α mylist → int
append : α mylist → α mylist → α mylist
rev :    α mylist → α mylist
hd :     α mylist → α (* Empty *)
tl :     α mylist → α mylist (* Empty *)
null :   α mylist → bool
```

(c) Schreiben Sie für Ihre Listen zusätzlich:

```
map :    (α → β) → α mylist → β mylist
filter : (α → bool) → α mylist → α mylist
foldl :  (α * β → β) → β → α mylist → β
foldr :  (α * β → β) → β → α mylist → β
```

(d) Schreiben Sie Umwandlungsprozeduren:

```
toList :   α mylist → α list
fromList : α list → α mylist
```

Lösungsvorschlag TE.16

(a) `datatype` α `mylist = Leer | Comb of` α `*` α `mylist`

(b)

```
fun length Leer = 0
  | length (Comb(x, xr)) = 1 + length xr

fun append Leer ys = ys
  | append (Comb(x, xr)) ys = Comb(x, append xr ys)

fun rev Leer = Leer
  | rev (Comb(x, xr)) = append (rev xr) (Comb(x, Leer))

fun hd Leer = raise Empty
```

```
  | hd (Comb(x, xr)) = x

fun tl Leer = raise Empty
  | tl (Comb(x, xr)) = xr

fun null Leer = true
  | null (Comb(x, xr)) = false
```

(c)

```
fun map f Leer = Leer
  | map f (Comb(x,xr)) = Comb(f x, map f xr)

fun filter p Leer = Leer
  | filter p (Comb(x,xr)) = if p x then Comb(x, filter p xr)
else filter p xr

fun foldl f s Leer = s
  | foldl f s (Comb(x,xr)) = foldl f (f (x,s)) xr

fun foldr f s Leer = s
  | foldr f s (Comb(x,xr)) = f(x, foldr f s xr)
```

(d)

```
fun toList Leer = nil
  | toList (Comb(x,xr)) = x :: toList xr

fun fromList nil = Leer
  | fromList (x :: xr) = Comb(x, fromList xr)
```

Aufgabe TE.17 *(No Nat)*
In dieser Aufgabe stellen wir die natürlichen Zahlen mit den Werten des Typs

```
datatype nat = O | S of nat
```

wie folgt dar: $0 \mapsto O$, $1 \mapsto S\ O$, $2 \mapsto S\ (S\ O)$, $3 \mapsto S\ (S\ (S\ O))$, und so weiter.

(a) Deklarieren Sie eine Prozedur `pred : nat → nat`, die zu einer Zahl den Vorgänger liefert. Wenn Sie den Vorgänger von O bestimmen wollen, sollen Sie die exception `Empty` werfen.

(b) Deklarieren Sie `iter : nat → α → (α → α) → α` mithilfe von `nat`.

(c) Deklarieren Sie für `nat` eine kaskadierte Prozedur `exp` die der Operation m^n für natürliche Zahlen entspricht. Verwenden Sie dabei keine Operationen für `int`.

Lösungsvorschlag TE.17

(a)

```
fun pred O = raise Empty
  | pred (S n) = n
```

(b)

```
fun iter O s f = s
  | iter (S n) s f = iter n (f s) f
```

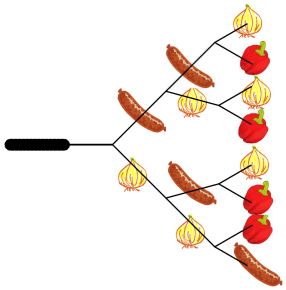
(c)

```
fun add O     y = y
  | add (S x) y = add x (S y)

fun mul O     y = O
  | mul (S x) y = add y (mul x y)

fun exp m O = S O
  | exp m (S n) = mul m (exp m n)
```

Aufgabe TE.18 *(Der Grillspießer)*
Sie haben eine grandiose Erfindung gemacht: Ein Grillspieß, an dem man nicht nur linear Dinge aufspießen kann. Er ist ein binärer Baum und sieht so aus:



Und er kann auf beliebige Größe erweitert werden. Toll!

Auf jeden Abschnitt Ihres Grillspießes kann man genau eine Zutat aufspießen. Die verfügbaren Zutaten sind:

Zutat	Köstlichkeit
Zwiebel	2
Paprika	1
Lyoner	3

Dabei gibt die Köstlichkeit einer Zutat an, wie lecker sie ist.

(a) Definieren Sie einen geeigneten Konstruktortypen `zutat`, der die verschiedenen Zutaten abbildet. Deklarieren Sie dann eine Prozedur `hunger : zutat → int`, die die Köstlichkeit einer Zutat berechnet.

(b) Deklarieren Sie nun einen geeigneten Konstruktortypen `grillspiess`, welcher einen Ihrer Grillspieße abbilden soll.

(c) Man kann eine Zutat auf ihrem Grillspieß erst essen, wenn man alle Zutaten gegessen hat, die nach ihr aufgespießt worden sind. Deklarieren Sie eine Prozedur `mampf : grillspiess → zutat list`, die die Zutaten eines Grillspießes in essbarer Reihenfolge zurück gibt.

(d) Unter einem ungesunden Grillspieß verstehen wir einen Grillspieß, bei dem beim Aufessen gemäß der Prozedur `mampf` aus Aufgabenteil (c) die gleiche Zutat zwei mal hintereinander auftaucht. Schreiben Sie eine Prozedur `igitt : grillspiess → bool`, die Ihnen berechnet, ob ein Grillspieß ungesund ist.

(e) Sie möchten nun aus Ihren zubereiteten Grillspießen den Leckersten auswählen - das ist der Grillspieß mit der in der Summe größten Köstlichkeit. Schreiben Sie also eine Vergleichsprozedur, die Grillspieße nach ihrer Köstlichkeit vergleicht. Beachten Sie, dass ein ungesunder Grillspieß dabei per Definition eine Köstlichkeit von -1 hat.

Ob ein Grillspieß ungesund bzw. lecker ist, oder nicht, hängt dabei natürlich von der in (c) geschriebenen Essreihenfolge ab.

Lösungsvorschlag TE.18

(a)

```
datatype zutat = Zwiebel | Paprika | Lyoner
fun hunger Zwiebel = 2
  | hunger Paprika = 1
  | hunger Lyoner  = 3
```

(b) Die Spitze ist leer, und jede Verzweigung speichert die Kinderspieße sowie die jeweils aufgespießte Zutat für jeden Kinderspieß.

```
datatype grillspiess = Spitze
  | Verzweigung of zutat * grillspiess * zutat * grillspies
```

(c)

```
fun mampf Spitze = nil
  | mampf (Verzweigung (z1, g1, z2, g2)) = mampf g1 @ [z1] @ mampf g2 @ [z2]
```

(d)

```
fun igitt spiess =
let
  fun occuresTwice nil = false
    | occuresTwice [x] = false
    | occuresTwice (x1::x2::xr) = x1 = x2 orelse occuresTwice (x2::xr)
in
  occuresTwice (mampf spiess)
end
```

(e)

```
fun compareSpiess (s1, s2) =
let
  val (s1rf, s2rf) = (mampf s1, mampf s2)
  val sum1 = foldl (fn (x,a) ⇒ hunger x  + a) 0 s1rf
  val sum2 = foldl (fn (x,a) ⇒ hunger x  + a) 0 s2rf
  val sum1 = if igitt s1 then ~1 else sum1
  val sum2 = if igitt s2 then ~1 else sum2
in
  Int.compare (sum1, sum2)
end
```