



# Programmierung 1 (WS 2020/21)

## Zusatztutorial 1 (Lösungsvorschläge)

### Konstruktortypen

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

#### Nichtrekursive Datentypen

##### Aufgabe Z1.1 (*Nintendogs*)

Legen Sie einen Datentyp `tier` an, der mindestens vier Konstruktoren hat. Tiere sind zum Beispiel Hund, Katze, Maus oder Wellensittich. Stellen Sie außerdem einen Typ `haustier` auf. Ein Haustier ist dabei ein Tier mit Namen. Welche Gleichheit kann man auf Haustieren definieren?

##### Lösungsvorschlag Z1.1

```
1 datatype tier = HUND | KATZE | MAUS | WELLENSITTICH | KROKODIL | TIGER | FUCHS
2 type haustier = tier * string
```

Als Gleichheit zwischen Haustieren würde es sich anbieten zu sagen, dass zwei Haustiere genau dann gleich sind, wenn sie das gleiche Tier sind und den gleichen Namen haben.

##### Aufgabe Z1.2 (*Schokopause*)

Die Kinderwelt freut sich über eine neue Reihe der Überraschungseier. In diesen sind wie immer verschiedene Spielzeuge enthalten. Diesmal enthalten sie bekannte StarWars Miniatur-Figuren! Vertreten sind Luke, Lea, Han Solo, Darth Vader und Chewbacca.

- (a) Überlegen Sie sich einen Konstruktortypen für Überraschungseier und Figuren.
- (b) Leider kam es bei vereinzelt Überraschungseiern zu einem Produktionsfehler, wodurch manche keine Figur enthalten. Wie lässt sich der Konstruktortyp für die Eier nun sinnvoll korrigieren?
- (c) Schreiben Sie eine Prozedur `count : egg list * figure → int`, die gegeben einer Liste von Überraschungseiern und einer Figur zählt, wie oft diese Figur in den Eiern enthalten war.

##### Lösungsvorschlag Z1.2

- (a)

```
1 datatype figure = Luke | Lea | Han | Darth_Vader | Chewbacca
2 datatype egg = StarWars of figure
```
- (b)

```
1 datatype egg = StarWars of figure option
```
- (c)

```
1 fun count nil fig = 0
2   | count ((StarWars NONE) :: xr) fig = count xr fig
3   | count ((StarWars (SOME f)) :: xr) fig = (if fig = f then 1 else 0) + count xr fig
```

### Aufgabe Z1.3 (Computer sagt nein)

Im Folgenden werden wir einen Datentyp *computer* konstruieren, mit dem wir verschiedene Arten von Computern darstellen können. Ein Computer kann entweder ein Netbook, ein Notebook oder ein Tower-PC sein. Alle Computer bestehen bei uns aus Speicher und Prozessor, wobei sowohl Speicher als auch Prozessor eine Ganzzahl speichern, die ihre Performance angibt.

- (a) Implementieren Sie einen geeigneten Konstruktortypen, um Computer in SML gemäß dieser Angaben darstellen zu können.
- (b) Schreiben Sie eine Prozedur `PCcomp : computer * computer → order`, die zwei Computer nach ihrer Performance vergleicht. Bei einem Netbook soll die Performance des Speichers zu zwei Dritteln und die des Prozessors zu einem Drittel in die Bewertung eingehen, bei einem Notebook der Speicher jeweils zur Hälfte und bei einem Tower-PC der Prozessor zu zwei Dritteln und der Speicher zu einem Drittel. Sie dürfen eine Hilfsprozedur verwenden.

#### Lösungsvorschlag Z1.3

- (a) 

---

```
1 datatype computer = Netbook of int * int
2                   | Notebook of int * int
3                   | Tower of int * int
```

---
- (b) 

---

```
1 fun bewertung (Netbook (s, p)) = 4 * s + 2 * p
2   | bewertung (Notebook (s, p)) = 3 * s + 3 * p
3   | bewertung (Tower (s, p)) = 2 * s + 4 * p
4
5 fun PCcomp (x,y) = Int.compare(bewertung x, bewertung y)
```

---

### Aufgabe Z1.4 (Matrjoschkas)

Eine Matrjoschka ist eine (aus Holz gefertigte und bunt bemalte) ineinander schachtelbare, eiförmige russische Puppe<sup>1</sup>. Eine Matrjoschka besteht also aus einer Reihe von Holzpuppen, die bis auf die innerste Puppe alle hohl sind und kleinere Puppen aufnehmen können.

- (a) Deklarieren Sie einen Datentyp in SML, der eine so zusammengesetzte Matrjoschka Puppe *mit Hilfe von Optionen* darstellen kann. Dabei soll `M NONE` die Darstellung der innersten Puppe sein.
- (b) Schreiben Sie eine Prozedur `count : matrjoschka → int`, die für eine gegebene Matrjoschka zählt, aus wie vielen Puppen sie besteht.

#### Lösungsvorschlag Z1.4

- (a) 

---

```
1 datatype matrjoschka = M of matrjoschka option
```

---
- (b) 

---

```
1 fun count (M NONE) = 1
2   | count (M (SOME x)) = 1 + count x
```

---

### Aufgabe Z1.5 (Optionizer)

Schreiben Sie eine Prozedur `optionizer : α option list → α list`, welche eine `option`-Liste in eine Liste übersetzt. Sobald jedoch ein `NONE` in der Liste auftritt, wird die bereits übersetzte Liste verworfen und ab dem `NONE` eine neue Liste aufgebaut.

---

<sup>1</sup><https://de.wikipedia.org/wiki/Matrjoschka>

■ **Beispiel:** `optionizer [SOME a, NONE, SOME b] = [b]`

#### Lösungsvorschlag Z1.5

---

```
1 fun optionizer xs = let
2   fun opt' nil ys = ys
3     | opt' (SOME x :: xr) ys = opt' xr (x :: ys)
4     | opt' (NONE :: xr) ys = opt' xr nil
5 in opt' xs nil end
```

---

#### Aufgabe Z1.6 (Wohnungssuche)

Dieter Schlau sucht für seine WG eine neue Wohnung. Um die optimale Wohnung zu finden, will er sich ein SML-Programm schreiben, das ihm die perfekte Wohnung sucht.

Wohnungen bestehen aus verschiedenen Zimmern, nämlich:

- Schlafzimmern, wobei Dieter an der Anzahl der Kleiderschränke und der Anzahl der Betten interessiert ist;
- Wohnzimmern, wobei Dieter an der Größe (in  $m^2$ ) interessiert ist;
- Badezimmern, wobei Dieter danach schaut, ob sie eine Badewanne haben;
- Fluren, wobei Dieter die Anzahl der Kleiderhaken interessiert.

Weitere Zimmer gibt es nicht.

- (a) Deklarieren Sie einen Konstruktortypen `zimmer`, mit dessen Hilfe Sie Dieters Zimmervorstellungen modellieren können.
- (b) Dieter stellt nun Wohnungen als `zimmer list` da. Schreiben Sie eine Prozedur, die die Anzahl der Betten in einer Wohnung zählt. Verwenden Sie dazu Faltung.
- (c) Schreiben Sie nun eine Prozedur, die aus einem Wohnungskatalog (dargestellt als `zimmer list list`) alle Wohnungen herausucht, die mindestens 5 Betten enthalten.

#### Lösungsvorschlag Z1.6

- (a) `datatype zimmer = Schlaf of int*int | Wohn of real | Bade of bool | Flur of int`
- (b) `fun countbeds ks = foldl (fn (Zimmer (_, x), a) => x + a | (_, a) => a) 0 ks`
- (c) `fun findwohnung ws = filter (fn k => countbeds k >= 5) ws`

## Rekursive Datentypen

#### Aufgabe Z1.7 (MyList - Recap)

Wir wollen Listen selbst implementieren.

Betrachten Sie folgenden Konstruktortypen: `datatype  $\alpha$  mylist = Leer | Comb of  $\alpha$  *  $\alpha$  mylist`

- (a) Schreiben Sie für Ihre Listen die Prozeduren:

- (i) `null :  $\alpha$  mylist  $\rightarrow$  bool`
- (ii) `hd :  $\alpha$  mylist  $\rightarrow$   $\alpha$  (* Empty *)`
- (iii) `length :  $\alpha$  mylist  $\rightarrow$  int`

- (b) Schreiben Sie für Ihre Listen zusätzlich `map : ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\alpha$  mylist  $\rightarrow$   $\beta$  mylist`

### Lösungsvorschlag Z1.7

(a)

---

```
1 fun null Leer = true
2   | null (Comb(x, xr)) = false
3
4 fun hd Leer = raise Empty
5   | hd (Comb(x, xr)) = x
6
7 fun length Leer = 0
8   | length (Comb(x, xr)) = 1 + length xr
```

---

(b)

---

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

---

### Aufgabe Z1.8 (Natürliche Zahlen)

Wir stellen die natürlichen Zahlen mit den Werten folgendes Konstruktortyps dar:

---

```
1 datatype nat = 0 | S of nat
```

---

So ergibt sich z. B.  $0 \mapsto O$ ,  $1 \mapsto SO$ ,  $2 \mapsto S(SO)$ ,  $3 \mapsto S(S(SO))$ .

- (a) Deklarieren Sie eine Prozedur `code : int → nat`, die die Darstellung einer natürlichen Zahl liefert und eine Prozedur `decode : nat → int`, sodass `decode (code n) = n` für alle natürlichen Zahlen gilt.
- (b) Deklarieren Sie eine Prozedur `natless : nat → nat → bool`, die kleiner-als, also  $m < n$  berechnet, ohne dabei Operationen für `int` zu verwenden.

### Lösungsvorschlag Z1.8

(a)

---

```
1 fun code 0 = 0
2   | code n = S (code (n-1))
3 fun decode 0 = 0
4   | decode (S x) = 1 + decode x
```

---

(b)

---

```
1 fun natleq 0 0 = false
2   | natleq 0 n = true
3   | natleq m 0 = false
4   | natleq (S m) (S n) = natleq m n
```

---

## Logische Ausdrücke

Im Folgenden verwenden wir den Konstruktortypen `logic`, mit Hilfe dessen logische Ausdrücke dargestellt werden können. Logische Ausdrücke bestehen dabei aus:

- Konstanten für wahr und falsch
- Variablen (dargestellt als `string`)
- Zwei binären Operatoren: dem logischen Und und dem logischen Oder
- Einem unären Operator: der logischen Negation

---

```
1 datatype logic = True          (* Konstante wahr *)
2                 | False       (* Konstante falsch *)
3                 | Var of string (* Variablen *)
4                 | And of logic * logic (* Und *)
5                 | Or  of logic * logic (* Oder *)
6                 | Not of logic   (* Negation *)
```

---

### Aufgabe Z1.9 (*count*)

Schreiben Sie eine Prozedur `countAnd : logic → int`, die zählt wie oft das logische Und in einem gegebenen Ausdruck auftaucht.

#### Lösungsvorschlag Z1.9

---

```
1 fun countAnd (And(e1,e2)) = 1 + countAnd e1 + countAnd e2
2   | countAnd (Or(e1,e2))  = countAnd e1 + countAnd e2
3   | countAnd (Not(e))     = countAnd e
4   | countAnd x             = 0
```

---

### Aufgabe Z1.10 (*Hardcode-Vergleiche*)

Schreiben sie eine rekursive Prozedur `logicEqual : logic → logic → bool`, die zwei logische Ausdrücke miteinander vergleicht.

#### Lösungsvorschlag Z1.10

---

```
1 fun logicEqual True      True      = true
2   | logicEqual False     False     = true
3   | logicEqual (Var x)   (Var y)   = x=y
4   | logicEqual (Not e1)  (Not e2)   = logicEqual e1 e2
5   | logicEqual (And (e1,e2)) (And (e3,e4)) = logicEqual e1 e3 andalso logicEqual e2 e4
6   | logicEqual (Or (e1,e2)) (Or (e3,e4)) = logicEqual e1 e3 andalso logicEqual e2 e4
7   | logicEqual _        _          = false
```

---

### Aufgabe Z1.11 (*simplify*)

Schreiben Sie eine nicht-rekursive Prozedur `simplifyOnce : logic → logic`, die einen logischen Ausdruck (falls möglich) zu einem semantisch äquivalenten Ausdruck vereinfacht. Dabei sollen folgende Regeln implementiert werden:

- `Or(e, False)` wird zu `e` vereinfacht.
- `Or(False, e)` wird zu `e` vereinfacht.
- `And(e, False)` wird zu `False` vereinfacht.
- `And(False, e)` wird zu `False` vereinfacht.
- `Not(Not e)` wird zu `e` vereinfacht.

#### Lösungsvorschlag Z1.11

---

```
1 fun simplifyOnce (And (False, e)) = False
2   | simplifyOnce (And (e, False)) = False
3   | simplifyOnce (Or (False, e)) = e
4   | simplifyOnce (Or (e, False)) = e
5   | simplifyOnce (Not(Not e)) = e
6   | simplifyOnce e = e
```

---

### Aufgabe Z1.12 (*simplifyMax*)

Schreiben Sie eine Prozedur `simplifyMax : logic → logic`, die einen logischen Ausdruck zu einem semantisch äquivalenten Ausdruck **maximal** vereinfacht.

Hierbei sollen nur die Regeln der vorherigen Aufgabe angewandt werden. Verwenden Sie `simplifyOnce`.

#### Lösungsvorschlag Z1.12

---

```
1
2 fun simplifyMax e = simplifyOnce (case e of
3   Not e1      => Not(simplifyMax e1)
```

---

```
4         | And (e1, e2) ⇒ And(simplifyMax e1, simplifyMax e2)
5         | Or (e1, e2)  ⇒ Or(simplifyMax e1, simplifyMax e2)
6         | e ⇒ e
7     )
```

---