Übungsblatt 11— Lösungen

1 Modellierung einer Geldbörse (Altklausuraufgabe)

```
final case class Wallet(
  amountMoney: Int,
  numberOfDocuments: Int,
)
// --- later ---
def transformWallet(w: Wallet): Wallet =
  if w == null then
    changeWallet(w)
  else
    throw new RuntimeException("no wallet");
```

Mögliche Verbesserungen:

- `amountMoney` und `numberOfDocuments` können leicht verwechselt werden, weil sie den gleichen Typ haben. Z.B. eigenen Geld-Typen einführen (ist auch viel Praktischer mit Währungen und so…).
- Wir verwenden niemals `null`. Wenn es möglich sein soll, dass die Eingabe leer ist, dann `Option` verwenden.
- „Boolean blindness“ – In den `if`-Zweigen ist nicht mehr ersichtlich, welches der positive und welches der negative Zweig ist. Explizites pattern matching (z.B. auf Option) oder passende Higher-Order-Functions des verwendeten Typs (z.B. `map`) macht das deutlicher (der Code hat tatsächlich einen Bug: changeWallet wird aufgerufen, *wenn* das übergebene Wallet `null` ist).
- Wir verwenden niemals Exceptions. Wenn es denkbar ist, dass die Operation fehlschlägt benutze `Either` für fail-fast Verhalten oder `Validated` für Fehlerakkumulation.

Eine mögliche, bessere Version:

```
case class Money(amount: Int, currency: String)
final case class Wallet(
  amountMoney: Money,
  numberOfDocuments: Int,
)
// --- later ---
def transformWallet(w: Option[Wallet]): Either[String, Wallet] =
  w.map(changeWallet).toRight("no wallet")
```

2 Modellierung eines Kunden (Altklausuraufgabe)

2.1 Typ „Customer“

```
final case class NonEmptyList[A](head: A, tail: List[A])

enum Customer:
  case Private(name: String, phoneNumber: Option[String])
  case Business(name: String, phoneNumber: NonEmptyList[String])
```

2.2 Mögliches Problem

Name und Telefonnummer haben den gleichen Typ, obwohl sie fundamental unterschiedliche Konzepte beschreiben. Besser wäre, für die Telefonnummer einen eigenen Typ zu verwenden, der auch gleich die Validität prüft.

3 Wiederholung: Parametrizität (Altklausuraufgabe)

```
def p2[A,B,C,D](a: A, b: B)(f: (A,B) => C, g: (A,C) => D): D =
  g(a, f(a,b))
```

Die Erklärung dazu ist auch direkt die Lösung für Teilaufgabe b):

Da die Typen variabel sind, können keine Operationen außer den übergebenen ausgeführt werden. Es gibt keine Möglichkeit, die variablen Typen `C` und `D` zu instantiieren. Da ein `D` zurückgegeben werden muss, muss dieses über die gegebenen Funktionen erzeugt werden.

4 Wiederholung: Lazy Evaluation (Altklausuraufgabe)

4.1 Strikte Parameter vs. Call-by-Name vs. Lazy val

Strikte Parameter werden immer genau einmal ausgewertet, bevor der Methodenkörper ausgeführt wird.

Call-by-Name-Parameter werden nur ausgewertet, wenn die Auswertung im Methodenkörper an eine Stelle gelangt, wo sie benötigt werden. Das Ergebnis wird hierbei nicht gecached, so dass ein Call-by-Name-Parameter u.U. mehrfach ausgewertet wird

lazy val wird wie Call-by-Name nur ausgewertet, wenn der Wert tatsächlich gebraucht wird, das Ergebnis der Auswertung wird jedoch gespeichert, so das dieses maximal einmal berechnet wird.

4.2 Early-Stopping für foldRight

Gegeben ist folgende Funktionssignatur:

```
def foldRight[B](z: B)(f: (A, B) => B): B = this match
  case Cons(x, xs) => f(x, xs.foldRight(z)(f))
  case Nil => z
```

Um diese so anzupassen, dass Early-Stopping möglich ist, müssen wir nur die Signatur anpassen und manche Parameter als call-by-name markieren:

```
def foldRightLazy[B](z: => B)(f: (A, => B) => B):B = this match
  \\ ... same ...
```

Da der rekursive Aufruf im zweiten Parameter zu `f` steht, kann dieser verhindert werden, wenn wir ihn zu einem Call-By-Name-Parameter machen. Der Aufruf findet nun nur dann statt, wenn die übergebene Funktion diesen auch benutzt. Damit kann diese also selbst entscheiden, ob die Rekursion fortgesetzt werden soll. Das `z` muss nicht unbedingt Call-By-Name sein, um dieses

Verhalten zu erlauben, aber so wird dieses auch nur berechnet, falls das Fold die ganze Liste durchläuft.

Eine Lösung ohne Verwendung von Laziness ist auch möglich, jedoch muss hier etwas mehr angepasst werden. Anstatt in der übergebenen Funktion zu entscheiden, ob wir weiter rechnen, übergeben wir eine separate Funktion dafür. Diese gibt abhängig vom aktuellen Element ein Option vom Ergebnistyp zurück. Ist dieses leer, führen wir die Berechnung fort. Enthält es jedoch einen Wert, so geben wir diesen ohne weitere Rekursion zurück.

```
def foldRightLazy[B](z: => B)(p: A => Option[B])(f: (A, B) => B): B = this match
  case Cons(x, xs) => p(x) match
    case None  => f(x, xs.foldRight(z)(p)(f))
    case Some(res) => res
  case Nil => z
```

4.3 foldRight — Aufrufe

Hier zu den beiden oben gezeigten Lösungen passende Aufrufe von foldRight, die Elemente der Liste multiplizieren und bei einer 0 direkt abbrechen und 0 zurückgeben:

```
intList.foldRightLazy(1)((a, b) => if a == 0 then 0 else a * b
```

```
intList.foldRightLazy(1)(if i == 0 then Some(0) else None)(_ * _)
```