Dr. Th. Letschert
Softwarekomponenten

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 6

Aufgabe 1

1. Pythagoreische Tripel können mit einem For-Block leicht berechnet werden:

```
type Triple = (Int, Int, Int)

val p_triples: Seq[Triple] =
  for (
    i <- 1 to 10;
    j <- 1 to 10;
    k <- 1 to 10;
    if i*i + j*j == k*k
  ) yield (i, j, k)
```

   übersetzen Sie diesen Ausdruck in seine "Basisform", die mit *map / flatMap* auskommt.

2. Geben Sie eine Funktion an, die alle Teilsequnzen der Länge 3 einer Sequenz berechnet, d.h. alle Untersequenzen mit 3 Elementen:

```
def subseq3[A](lst: Seq[A]): Seq[Seq[A]] = ...

val w = subseq3(Seq("a", "b", "c", "d"))

println(
   w.map(_.mkString(", ")).mkString("\n")
)
/* a, b, c
   a, b, d
   a, c, d
   b, c, d */
```

   Arbeiten Sie mit einem **for**-Ausdruck.

3. Verallgemeinern Sie die Funktion zur Erzeugung der Untersequenzen die Länge von 3 auf ein beliebiges nicht-negatives n, das als Parameter übergeben wird.

```
def subseq[A](lst: Seq[A], n: Int): Seq[Seq[A]] = ...

val w = subseq(Seq("a","b","c","d"), 3)

 println(
   w3.map(_.mkString(", ")).mkString("\n")
 )
/* a, b, c
   a, b, d
   a, c, d
   b, c, d */
```

Arbeiten Sie soweit möglich auch wieder mit mit einem **for**-Ausdruck.

Hinweis: Mit Rekursion lassen sich Schleifen unbekannter Verschachtlungstiefe realisieren.

Aufgabe 2

Die Matrix-Multiplikation im **imperativen** Stil ist auch in Scala möglich:

```
type Matrix[A] = Array[Array[A]]

// a[M][K] * b[K][N] -> c[M][N]
def matMult[A: Numeric](a: Matrix[A], b: Matrix[A], c: Matrix[A]): Unit = {
  val numericA = summon[Numeric[A]]
  import numericA._

  val M = a.length // Anzahl der Zeilen in Matrix a
  val N = b(0).length // Anzahl der Spalten in Matrix b
  val K = a(0).length // Anzahl der Zeilen in Matrix b

  assert(K == b.length)
  assert(c.length == M)
  assert(c(0).length == N)

  // for-Anweisung
  for (i <- 0 until M;
       j <- 0 until N) {
       c(i)(j) = zero
       for (k <- 0 until K) {
         c(i)(j) = c(i)(j) + a(i)(k) * b(k)(j)
       }
  }
}

val a: Matrix[Int] =
  Array(
    Array(1, 2, 3),
    Array(4, 5, 6)
  )
val b = Array(
  Array(7, 8),
  Array(9, 10),
  Array(11, 12)
)
val c = Array.ofDim[Int](a.length, b(0).length)
matMult(a, b, c)
println(c.map(_.mkString(",")).mkString("\n"))
// 58,64
// 139,154
```

Das Mögliche ist nicht immer das Erwünschte. Konstruieren Sie eine äquivalente funktionale Version:

```
type Matrix[A] = Array[Array[A]]

// a[M][K] * b[K][N] = c[M][N]
def matMult[A: Numeric](a: Matrix[A], b: Matrix[A]) : Matrix[A] = {
  val numericA = summon[Numeric[A]]
  import numericA._

  val M = a.indices // M are indices of rows of Matrix a
```

```
  val N = b(0).indices // N are the indices of columns of Matrix b
  val K = a(0).indices // K is the number of rows of Matrix b and columns of
      matrix a

  assert(K.length == b.length) // a has as many rows as b has columns

  // for-Comprehension
  for ( i <- M .....

}

val a: Matrix[Int] = Seq(
   Seq(1, 2, 3),
   Seq(4, 5, 6)
  )

val b = Seq(
  Seq(7, 8),
  Seq(9, 10),
  Seq(11, 12)
)

val c = matMult(a, b)
println(c.map(_.mkString(",")).mkString("\n"))
// 58,64
// 139,154
```

Auch in dieser Version soll **for** genutzt werden. Hier sind natürlich nur **for-Ausdrücke** erlaubt.

Aufgabe 3

1. Der Typ `ID` mit

   ```
   type ID[A] = A
   ```

   kann als (die identische) Monade definiert werden: (Ergänzen Sie)

   ```
   given idMonad: Monad[ID] with {
     def pure[A](a: A): ID[A] = a
     extension [A](fa: ID[A]) {
       def flatMap[B](f: A => ID[B]): ID[B] = ???
       def map[B](f: A => B): ID[B] = ???
     }
   }
   ```

   Damit können einfache Werte als Monaden übergeben werden. Ein Verwendungsbeispiel ist

   ```
   def sumSquare[F[_]: Monad, N: Numeric](a: F[N], b: F[N]): F[N] = {
     val numeric = summon[Numeric[N]]
     import numeric._
     for {
       x <- a
       y <- b
     } yield x*x + y*y
   }

   // Iteration über Listen
   val sq_list = sumSquare(List(2.0), List(3.0)) // List(13)
   ```

```
// implizite Konversion A => ID[A]
given toID[A]: Conversion[A, ID[A]] with {
  def apply(a: A): ID[A] = a
}

// "Iteration" über einzelne Werte
val sq_id = sumSquare(2.0, 3.0) // 13
```

2. Die identische Monade kann beispielsweise eingesetzt werden, um Code zu testen der unter Produktionsbedingungen asynchon ausgeführt wird. Nach dem Test mit *ID* können Sie jetzt *Future* als Monade verwenden:

```
val sq_f: Future[Double] = sumSquare( Future(2.0), Future(3.0) )

sq_f.onComplete {
  case Success(result) => println(result)
  case Failure(failure) => println("Failed because of " + failure)
}
```

   Natürlich muss *Future* dazu als Instanz von *Monad* definiert werden.

Aufgabe 4

Die Scala-API bietet mit Klassen wie *Option*, *Either*, *Try*, mehrere Möglichkeiten zur Realisation einer Fehlermangement-Monade. Mit ihnen kann eine *Fast-Failure*-Semantik realisiert werden: Wenn ein Fehler auftritt wird sofort alles abgebrochen. Eine Verallgemeinerung dieser speziellen Klassen zu einer allgemeineren Fehlermangement-Monade *Error* könnte eine Monade sein, die eine Methode bereit stellt, mit der das Auftreten eines Fehlers signalisiert werden kann (siehe Foliensatz 6).

Das ist vielleicht noch etwas knapp. Normalerweise wird man auch noch mit einer speziellen Information über die Ursache des Fehlers informiert werden, beispielsweise mit einem Wert vom Typ *E*

Hilfreich könnte es zudem auch sein, wenn die Monade filterbar ist, dann könnte auf einfache Art Fehlersituationen geprüft werden. Z.B. mit einer Methode *ensure* die ein Prädikat und einen Fehlerwert als Argument hat. Der Fehlerwert kommt zum Einsatz, wenn das Prädikat nicht zutrifft.

Damit könnte man eine allgemeine Fehlermangement-Monade mit Fehlernachtichten wie folgt definieren:

```
trait ErrorMonad[F[_], E] {
  def error[A](e: E): F[A]
  def pure[A](x: A): F[A]
  extension [A, B] (x: F[A]) {
    def map(f: A => B): F[B]
    def flatMap(f: A => F[B]): F[B]
    def ensure(p: A => Boolean, orElse: E): F[A]
  }
}
```

1. Mit

```
type StringErrorMonad[G[_]] = ErrorMonad[G, String]

object StringErrorMonad {
  def apply[F[_]: StringErrorMonad] = summon[StringErrorMonad[F]]
}
```

   kann jetzt eine verallgemeinerte Ausdrucksauswertung definiert werden:

```
enum Exp {
  case Const(v: Int)
  case Add(t1: Exp, t2: Exp)
  case Sub(t1: Exp, t2: Exp)
  case Mult(t1: Exp, t2: Exp)
  case Div(t1: Exp, t2: Exp)
}
import Exp._

def eval[F[_]: StringErrorMonad](e: Exp): F[Int] = ...
```

Mit der passenden Typklasseninstanz – etwa ein *Either* – kann dann ein fehlerhafter Ausdruck zu einer Fehlermeldung führen:

```
val exp: Exp =
  Add(Const(18),
    Div(
      Mult(Const(12), Const(4)),
      Const(0)))

val res = eval(exp) // Left(Division by zero)
```

2. Hier ist *eval* auf Strings als Fehlerwerte festgelegt. Kann das auch noch verallgemeinert werden?

   Hinweis: Die Beschränkungs-Syntax zur impliziten Übergabe von Instanzen kommt gelegentlich an Grenze. Man kann dann auf implizite Objekte in klassischer Art zurück greifen. Z.B.:

   ```
   def eval[F[_], E: EvaluationError](
     e: Exp)(
     using fe: ErrorMonad[F, E]): F[Int] = ...
   ```

   statt so etwas wie:

   ```
   // nicht übersetzbar:
   def eval[ G[F[_], E]: ErrorMonad, E: EvaluationError ....] ....
   ```