```scala
/* Final Exam: Advanced Programming, by Andrzej Wąsowski IT University
 * of Copenhagen, Autumn 2024: 06 January 2025
 *
 * The exam consists of 12 questions to be solved within 4 hours.
 * Solve the tasks in the file 'Exam.scala' (this file).
 *
 * You can use all functions provided in the included files,  as well as
 * functions that we implemented in the course. If the source is missing in
 * this folder, you can add it to this file (so that things compile on our
 * side). You can use the standard library functions as well. Staying closer
 * to the course API is likely to yield nicer solutions.
 *
 * You can access any static written materials, printed and online, but you
 * are not allowed to communicate with anybody or with anything (bots).
 * Using GitHub copilot, ChatGPT and similar language models during the exam
 * is not allowed. By submitting you legally declare to have solved the
 * problems alone, without communicating with anybody, and not using
 * language models.
 *
 * Do not modify this file in other ways than answering the questions or
 * adding imports and source of needed functions. Do not reorder the
 * answers, and do not remove question numbers or comments from the file.
 *
 * Submit this file and only this file to LearnIT. Do not convert to
 * any other format than .scala. Do not submit the entire zip archive.
 * The only accepted file format is '.scala'.
 *
 * Keep the solutions within 80 character columns to make grading easier.
 *
 * The answers will be graded manually. We focus on the correctness of
 * ideas, the use of concepts, clarity, and style. We will use undisclosed
 * automatic tests during grading, but not to compute the final grade, but
 * to help us debug your code.
 *
 * We do require that your hand-in compiles.  The directory has a project
 * setup so compilation with scala-cli shall work out-of-the-box. If you
 * cannot make a fragment compile, put your solution in a comment, next to
 * the three question marks. We will grade the solutions in comments as
 * well.
 *
 * We will check whether the file compiles by running
 *
 *    scala-cli compile .
 *
 * Hand-ins that do not compile will automatically fail the exam.
 *
 * There is a skeletong test file in the bundle, that you can use to test
 * your solutions.  It does not contain any useful tests. It is just there
 * to get you started with testing faster.
 *
 * We do not recommend writing and running tests if you are pressed for
 * time. It is a good idea to run and test, if you have time.  The tests
 * should not be handed in.  We only grade the answers to questions below.
 *
 * Good luck!
 **/

package adpro

import org.scalacheck.{Arbitrary, Gen, Prop}
import Arbitrary.*, Prop.*
import org.scalactic.TripleEquals.*

import adpro.laziness.LazyList
```

```scala
import adpro.state.*

object Good:

  /* QUESTION 1 ######################################################
   *
   * Implement a function `goodPairs` that checks whether all pairs of
   * consecutive elements in a list satisfy a predicate. Choose the
   * right higher order function for the task. If you can't solve this
   * with higher order functions, using recursion still makes sense,
   * even if for less points.
   */

  def goodPairs [A] (l: List[A], good: (A,A) => Boolean): Boolean =
    ???



  /* QUESTION 2 #####################################################
   *
   * Recall the functions  curry and uncurry from the course (week 1).
   * In this exercise we use the standard library counterparts,
   * `curried` and `uncurried` see these docs (if you don't recall
   * them):
   *
   * https://scala-lang.org/api/3.4.2/scala/Function$.html#uncurried-d4
   * https://scala-lang.org/api/3.4.2/scala/Function2.html#curried-0
   *
   * Use the right one of these functions to produce a function
   * `goodPairsCurried` by transforming goodPairs programmatically,
   * without writing it from scratch. The expected type is given below.
   *
   * This question can be solved even if you did not answer Q1. Just
   * assume you have the solution for Q1.
   */

  def goodPairsCurried[A]: List[A] => ((A,A) => Boolean) => Boolean =
    ???



  /* QUESTION 3. #####################################################
   *
   * Now Implement function curriedNested that takes a higher order
   * function with the first argument being an uncurried binary
   * function and curries the first argument. See the type
   * specification below.
   *
   * This question can be solved even if you did not answer the
   * previous questions.
   */

  def curriedNested [A, B, C, D] (f: ((A,B) => C) => D)
    : (A => B => C) => D = ???



  /* QUESTION 4 ######################################################
   *
   * Create a function goodPairsHotCurry where both the top-level
   * function and the first argument are curried. Do not implement the
   * function from scratch but use curriedNested and standard library
```

```scala
   * functions to transform `goodPairs`.
   *
   * This question can be solved even if you did not answer the
   * previous questions.
   */

  def goodPairsHotCurry[A]: List[A] => (A => A => Boolean) => Boolean =
    ???

end Good



object MultivariateUniform:

  import pigaro.*
  import adpro.monads.*

  /* QUESTION 5 #####################################################
   *
   * Recall our probabilistic programming library Pigaro.  We want to show
   * that Pigaro's `Dist` type constructor is a monad. Provide evidence (a
   * given, an instance) of Monad for Dist.
   */

  // given ... (add answer here)


 /* QUESTION 6 #####################################################
  *
  * Implement a function `multUni`  that represents a product of
  * n identical uniform distributions, where n is its first argument.
  * A single sample from this distribution is a list of size n.
  *
  * def multUni (n: Int, values: T*): Dist[List[T]]
  *
  * You likely need to use the fact that Dist is a monad. If you do so
  * you should ensure that the function signature enforces this
  * requirement on the caller. Questions 5 and 6 are conceptually
  * related, but this one can be answered without answering Q5.
  */

  def multUni[T] (n: Int, values: T*): Dist[List[T]] = ???

end MultivariateUniform



object Gens:

  /* QUESTION 7 #####################################################
   *
   * Imagine we are writing some tests for a function that takes a value of
   * type Either[A,B] as an input, for some unknown types A and B (type
   * parameters).  We do not have access to any Arbitrary[A] and
   * Arbitrary[B] instances. Instead, we have access to Arbitrary[Option[A]]
   * and Arbitrary[Option[B]] instances.
   *
   * Write a function genEither[A,B] that returns a value of
   * Gen[Either[A,B]] using the Arbitrary[Option[A]] and
   * Arbitrary[Option[B]]. Your implementatation needs to ensure that the
   * arbitraries are available in the scope of the function (the type
   * checker must check for their existance).
   *
```

```
   * We are working with the scalacheck library here, so we use
   * org.scalacheck.Gen and org.scalacheck.Arbitrary, not the book's Gen.
   *
   * A direct recursion is allowed and will award maximum points in this
   * exercise. Non-recusive solutions are also possible.
   */

  def genEither[A,B]: Gen[Either[A,B]] = ???

end Gens



object IntervalParser1:

  import adpro.parsing.*
  import adpro.parsing.Sliceable.*

  /* QUESTION 8 ######################################################
   *
   * Implement a parser that accepts a single integer from a closed
   * interval between low and high.
   *
   *    intBetween(low: Int, high: Int): Parser[Option[Int]]*
   *
   * The parser always succeeds. It returns Some(n) if it parses an integer
   * n. It returns None, if the integer is not in the interval.
   *
   * Use the parser combinator library developed in the course. You may want
   * to use a concrete parser implemetnation. The parser `Sliceable` is
   * included in the exam project.
   */

  def intBetween (low: Int, high: Int): Parser[Int] = ???

end IntervalParser1



object IntervalParser2:

  import adpro.parsing.*

  /* QUESTION 9 #####################################################
   *
   * Notice that `intBetween` is independent of the concrete parser
   * implementation.  We can abstract over the parser type. Implement it
   * again as an extension that works for any implementation of the
   * `Parsing` structure
   *
   * This question depends on the previous one. You need to copy your
   * answer to Q8 and generalize it to an extension of instances of
   * Parsers. Since now our parser implementation is abstract  you may
   * need to build the integer token lexer differently than in Q8 (it
   * depends a bit on which solution you proposed in Q8---you can no
   * longer use methods from Sliceable here).
   *
   * The goal is to have something like this code compile:
   *
   *  import IntervalParser2.*
   *  def f [P[+_]] (p: Parsers[ParseError, P]) =
   *    p.intBetween(0,0) ...
   *
   * HINT: The extension will be for p: Parsers[ParseError, P] for
```

```scala
   * some implementation of `Parsing` represented by type constructor
   * variable P[+_].
   */

  // Write your solution here (below)
  // ...

end IntervalParser2


/* QUESTION 10 ########################################################
 *
 * Implement a type class `Member[F[+_]]` that ensures that its instances
 * provide a method `contains`:
 *
 *   def contains[A] (fa: F[A], a: A): Bolean
 *
 * The intuition is that this method can be used to check whether `fa`
 * contains the element `a` (although this intuition is irrelevant for the
 * task at hand). The type class should be implemented as an abstract trait.
 */

// Add your answer here (bnlow)
// ...



/* QUESTION 11 ########################################################
 *
 * Read the following interface extracted from a railway ticketing system.
 * The question is formulated underneath.
 *
 * The train reservation system accepts payments and creates reservations.
 * Each of the four methods is commented below.  We assume this interface is
 * imperative, so most of the functions have side effects. But this does not
 * matter for the questions below.
 **/

object Trains:
  trait ReservationSystem:

    // Return paymentId if successfully charged the amount; otherwise error
    def pay (CreditCard: String, amount: Int): Either[String, String]

    // Create a reservation, returns a ticket number if successful, or an error
    def reserve (passenger: String, train: String, paymentId: String)
      : Either[String, String]

    // Confirms the validity of the payment with a broker.
    // True if the paymentId is valid
    def validate (paymentId: String): Boolean

    // Returns a set of passengers on the train (a manifest)
    def paxOnTrain (train: String): Set[String]


object FullyAbstractTrains:

  /* Design a fully abstract version of the ReservationSystem interface
   * shown above. In particular abstract away from the details of
   * representation of credit cards, amounts, error messages,passanger
   * names, train numbers, ticket numbers, and payment ids. The idea is not
   * to use String and Int types as representations in the fully abstract
   * version. Either and Boolean are still fine to use, as they do not
```

```
   * represent data here.
   *
   * Because we may be using a distributed data store, we want to abstract
   * away from the representation of sets as query results (So abstract away
   * `Set[_]` as well. Assume though that whatever representation we use for
   * query results, it is a Monad, so that map and flatMap are available,
   * and that we can check whether query results contain an element. The
   * latter requires using the solution of Q10.
   */

  // trait ReservationSystem ... // your solution here




    /* QUESTION 12 ######################################################
     *
     * We want to write some property laws for the fully abstract version of
     * the train reservation system. These tests we cannot run before the
     * implementation is concrete. But they should compile, to support
     * test-first development.
     *
     * Note that this question depends on Q10-11. There are two laws to be
     * written below.
     */

    /* Law 1. A succesful Payment produces a valid PaymentId. Note that both
     * laws have to be members in your abstract version of the train
     * reservation system, so you may need to adjust indentation here to be
     * inside the trait above.
     **/

    def law1: Prop = ???

    /* Law 2. A succesful reservation puts the passenger on the requested
     * train (relates `reserve` with `paxOnTrain`). If `reserve` succeeds
     * then paxOnTrain returns a result containing the passenger.)
     */

    def law2: Prop = ???

end FullyAbstractTrains

// vim:tw=76:cc=70
```