# 185.A05 Advanced Functional Programming SS 2020

Wednesday, 1 April 2020

## Assignment 3

### on Chapter 4, Chapter 7, Chapter 3

**Topics:** Functional Pearls, Functional Arrays, Algorithm Patterns

**Submission deadline:** Monday, 4 May 2020, 12am (at the earliest)
(In case of need, this deadline will be extended (details posted via TISS))

*Regarding the deadline for the second submission:* Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

**Important:**

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 1. If you have any questions regarding these instructions, ask your questions in the TISS forum. Following these instructions is paramount to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment3.hs`

   of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 3.

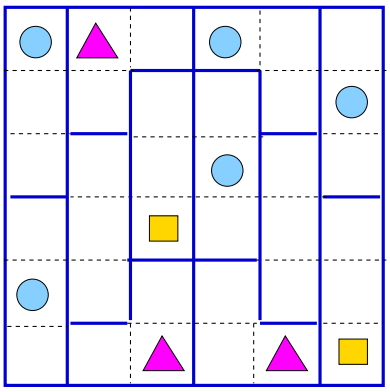
3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment3.hs`. Import declarations for self-defined modules will fail: Only `Assignment3.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

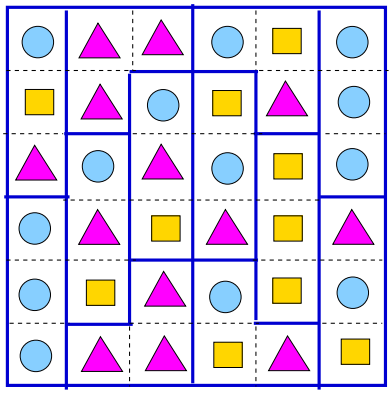
**Programming tasks:**

*Triplet puzzles* resemble Soduko puzzles. In this assignment we consider triplet puzzles and their little brothers, which we call *pairlet puzzles*. The below figures show a triplet and a pairlet puzzle together with their solutions to illustrutate the core idea underlying these puzzles.

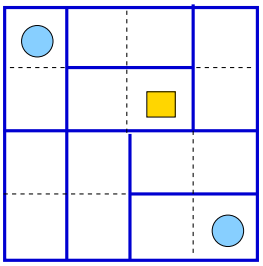
a) Triplet Puzzle



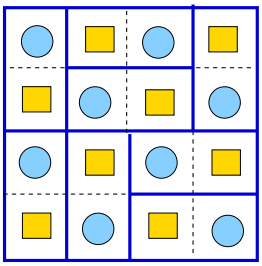
b) Solved Triplet Puzzle



c) Pairlet Puzzle



d) Solved Pairlet Puzzle



The *triplet* (and *pairlet*) *rules* are the following: (1) Fill in the grid so that every field contains one of the symbols *circle*, *square*, or *triangle* (for pairlet puzzles only a circle or a square). (2) The fields in each box contain either the same symbol, or are pairwise disjoint. (3) Vertically or horizontically adjacent fields of different boxes do not contain the same symbol.

We use the data structures below modelling triplet and pairlet puzzles in two ways in order to oppose and compare list and array programming. In detail, puzzles are modelled as (1) lists of rows being lists themselves, too, and as (2) two-dimensional arrays.

```
data Symbol = C | S | T | B deriving (Eq,Ord,Enum,Show)
            -- C: Circle, S: Square, T: Triangle, B: Blank

data Index = One | Two | Three | Four | Five | Six
             deriving (Eq,Ord,Enum,Show)

type OneTo12 = Int -- Only the numbers 1 to 12 (for the 12 boxes of triplets)
type OneTo8  = Int -- Only the numbers 1 to 8 (for the 8 boxes of pairlets)
```

```
-- For pairlets only the Index values One thru Four will be used
type Row_ix = Index
type Col_ix = Index

-- (1) Modelling Puzzles as lists of rows

type Matrix a = [Row a]
type Row a    = [a]

 -- The prefixes T and P stand for triplet and pairlet, resp.
 -- T_Boxes/P_Boxes values define the fields of the
 -- 12/8 boxes of triplet/pairlet puzzles
type T_Boxes = (OneTo12 -> [(Row_ix,Col_ix)])
type P_Boxes = (OneTo8 -> [(Row_ix,Col_ix)])

data T_Puzzle = LT (Matrix Symbol) T_Boxes
data P_Puzzle = LP (Matrix Symbol) P_Boxes

 -- Information on boxes suppressed for solved puzzles
type T_Puzzle_Solved = Matrix Symbol
type P_Puzzle_Solved = Matrix Symbol

-- (2) Modelling Puzzles as two-dimensional arrays

import Data.Array

instance Ix Index where...

 -- Type names as above but with a prefix A standing for array
type AMatrix i a = Array i a
data AT_Puzzle   = AT (AMatrix (Index,Index) Symbol) T_Boxes
data AP_Puzzle   = AP (AMatrix (Index,Index) Symbol) P_Boxes

  -- Information on boxes suppressed for solved puzzles
type AT_Puzzle_Solved = AMatrix (Index,Index) Symbol
type AP_Puzzle_Solved = AMatrix (Index,Index) Symbol
```

In the following, naive initial and advanced sophisticated solvers for triplet and pairlet puzzles shall be implemented.

If puzzles the solvers are applied to do not have a solution, the solvers shall yield the result `Nothing`. If a puzzle has more than one solution, it does not matter which of these solutions a solver returns. You can assume that solvers will only be applied to representations of 'proper' puzzle repesentations.

1. *Algorithms, solvers for the list representation of puzzles*

   (a) Implement two solvers for triplet and pairlet puzzles, respectively, which are straightforward and obviously correct at the expense of possibly being (very) inefficient and low performing. These solvers play the rôle of the initial algorithms solving a functional pearl problem (cf. Chapter 4 on functional pearls, especially, Chapter 4.5 on Soduko puzzles):

   ```
   t_solve_init :: T_Puzzle -> Maybe T_Puzzle_Solved
     -- initial triplet solver, list version

   p_solve_init :: P_Puzzle -> Maybe P_Puzzle_Solved
     -- initial pairlet solver, list version
   ```

   (b) Implement two sophisticated solvers for triplet and pairlet puzzles, respectively, which shall be tuned for performance. Ideally, you develop these solvers by transforming and refining your initial solvers step by step:

   ```
   t_solve :: T_Puzzle -> Maybe T_Puzzle_Solved
     -- sophisticated triplet solver, list version

   p_solve :: P_Puzzle -> Maybe P_Puzzle_Solved
     -- sophisticated pairlet solver, list version
   ```

   The Soduku solver of Richard Bird might serve as a source of inspiration (cf. Chapter 4.5). Maybe some of the algorithm patterns of Chapter 3 are useful, too.

2. *Algorithms, solvers for the array representation of puzzles*

   (a) Complete the instance declaration for index type `Index`:

   ```
   instance Ix Index where...
   ```

   (b) (Re-) implement the initial solvers for triplet, pairlet puzzles on arrays:

   ```
   at_solve_init :: AT_Puzzle -> Maybe AT_Puzzle_Solved
     -- initial triplet solver, array version

   ap_solve_init :: AP_Puzzle -> Maybe AP_Puzzle_Solved
     -- initial pairlet solver, array version
   ```

   (c) (Re-) implement the sophisticated solvers for triplet, pairlet puzzles on arrays:

   ```
   at_solve :: AT_Puzzle -> Maybe AT_Puzzle_Solved
     -- sophisticated triplet solver, array version

   ap_solve :: AP_Puzzle -> Maybe AP_Puzzle_Solved
     -- sophisticated pairlet solver, array version
   ```

3. *Predicates for checking puzzles*

   A triplet/pairlet puzzle is *correct* (i.e., *correctly solved*), if it is sound and complete. It is *sound*, if its entries obey the triplet/pairlet rules (where some of its fields may be empty, i.e., containing symbol `B`). It is *complete*, if none of its fields is empty.

   Implement the following predicate functions allowing to check soundness, completeness, and correctness of puzzles:

```
t_sound    :: T_Puzzle -> Bool
p_sound    :: P_Puzzle -> Bool
t_complete :: T_Puzzle -> Bool
p_complete :: P_Puzzle -> Bool
t_correct  :: T_Puzzle -> Bool
p_correct  :: P_Puzzle -> Bool

at_sound    :: AT_Puzzle -> Bool
ap_sound    :: AP_Puzzle -> Bool
at_complete :: AT_Puzzle -> Bool
ap_complete :: AP_Puzzle -> Bool
at_correct  :: AT_Puzzle -> Bool
ap_correct  :: AP_Puzzle -> Bool
```

4. **Without submission:**

   (a) Test all solvers and predicates with puzzles of your own choice. The initial solvers might terminate in reasonable time only for puzzles which are close to a correctly solved puzzle.

   (b) Compare the performance of the initial solvers and their sophisticated counterparts for both the list and array versions. Is there a performance difference also between solvers working on lists and their counterparts working on arrays?

   (c) How can you prove that your sophisticated solvers are functionally equivalent to your straightforward initial counterparts?

*Iucundi acti labores.*
*Getane Arbeiten sind angenehm.*

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller