Advanced Functional Programming: Assignment 3 (Wed, 04/12/2019)
Topics: Functional Pearls, Functional Arrays, and Streams
Submission deadline: Wed, 05/15/2019 (3pm) (> four weeks!)

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

Store all functions to be written for this assignment in a top-level file `assignment3.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

Functional Pearls: Solving Sudoku Puzzles

1. Implement the simple Sudoku solver `solve` (cf. slide 327) of Chapter 4.5 and its two improved/optimized versions called `solve`, too (cf. slides 345 and 353, resp.). In order to avoid name clashes replace the name `solve` by the unique identifiers `solve`, `solve_opt1`, and `solve_opt2`:

   ```
   solve      = filter valid . expand . choices         (cf. slide 327)
   solve_opt1 = filter valid . expand . prune . choices (cf. slide 345)
   solve_opt2 = search . choices                        (cf. slide 353)
   ```

   Use the same naming convention (i.e., adding the postfix `_opt1` resp. `_opt2`) for renaming auxiliary functions of `solve_opt1` and `solve_opt2` should their implementation differ for `solve`, `solve_opt1`, and `solve_opt2`.

2. The Sudoku solver of Chapter 4.5 models grids representing Sudoku puzzles as lists of rows, i.e., lists of lists:

   ```
   type Matrix a = [Row a]
   type Row a    = [a]
   type Grid     = Matrix Digit
   type Digit    = Char
   ```

   In this exercise, we want to use (static) functional arrays instead of lists of lists to model grids and hence Sudoku puzzles:

   ```
   import Array

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

   instance Ix Index where...

   type Arr_Matrix a = Array Index Arr_Row a
   type Arr_Row a    = Array Index a
   type Arr_Grid     = Arr_Matrix Digit
   type Digit        = Char
   ```

Complete the `instance`-declaration for type `Index` and (re-) implement the three Sudoku solvers of excersise 1 using the array representation of grids instead of the lists of lists representation of exercise 1 (i.e., except of this type change the algorithmic idea shall be the same as in exercise 1, i.e., (re-) implementing does not mean to convert the array representation of the initial grid into the lists of lists representation of exercise 1 which is then solved by the solvers of exercise 1, and afterwards retransformed into the array representation). Add the prefix `arr_` or `Arr_` (depending on the usage context), where necessary in order to resolve name clashes with (auxiliary) function and type names used in exercise 1. If types (e.g., `Digit`) or functions can just be reused from exercise 1, do so and re-use them and do not introduce renamed copies of them:

```
arr_solve      = ...
arr_solve_opt1 = ...
arr_solve_opt2 = ...
```

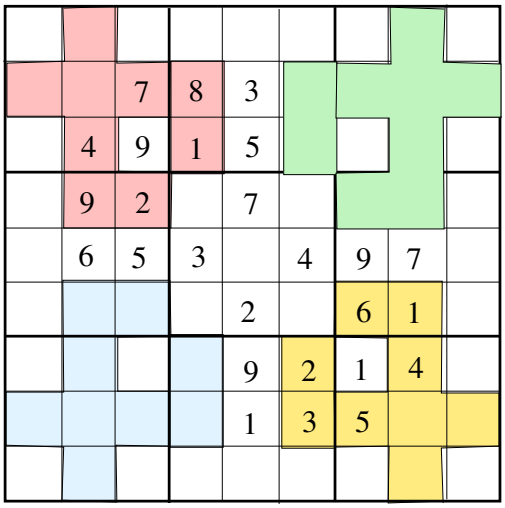
3. In exercise 2, grids are modelled as arrays of arrays. In this exercise, we want to use two-dimensional arrays instead:

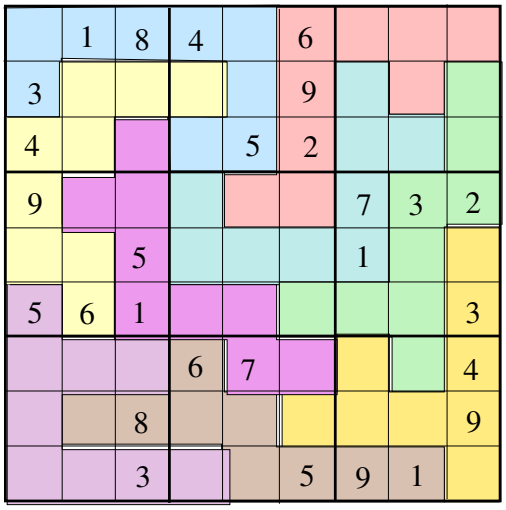
   ```
   type Arr2_Matrix a = Array (Index,Index) a
   type Arr2_Grid     = Arr2_Matrix Digit
   ```

   (Re-) implement the three Sudoku solvers of excersise 1 using the new array representation of grids. In order to resolve possible name clashes, add the prefix `arr2_` or `Arr2_` (depending on the usage context) to (auxiliary) function and type names analogously to exercise 2.

   ```
   arr2_solve      = ...
   arr2_solve_opt1 = ...
   arr2_solve_opt2 = ...
   ```

4. In this exercise we consider a variant of Soduko puzzles called *color Soduku puzzles*:





Color areas consist always of 9 cells; different color areas are disjoint, they do not overlap. The color areas of a color Soduko puzzle may or may not cover the whole grid as illustrated above.

A color Soduko puzzle is correctly solved, if all rows, columns, boxes, and areas of the same color contain the digits from '1' to '9' exactly once.

Using the grid representation of excercise 1 (i.e., lists of lists), adapt `solve_opt1`, and `solve_opt2` of exercise 1 to two new color Soduko solvers `csolve_opt1` and `csolve_opt2`, which take advantage of the color areas of color Soduku puzzles, which enable another pruning step, namely pruning by color areas. Except of taking advantage of this additional pruning opportunity, `csolve_opt1` and `csolve_opt2` shall match their counterparts of exercise 1:

```
type Col_Area  = [(Index,Index)]
type Col_Areas = [Col_Area]

csolve :: Grid -> Col_Areas -> [Grids]
csolve g a = solve g (i.e, csolve coincides with solve of exercise 1)
csolve_opt1 :: Grid -> Col_Areas -> [Grids]
csolve_opt1 = ...
csolve_opt2 :: Grid -> Col_Areas -> [Grids]
csolve_opt2 = ...
```

In order to resolve possible name clashes between function names, add the prefix `c` to (auxiliary) function names, which require a deviating implementation from the one of exercise 1.

5. **Without submission:** Test all Sudoku solvers by means of (valid) initial grids of your own choice. An initial grid is *valid*, if it does not contain duplicates in any row, column, box, or color area. Compare the relative performance of the different solvers. Are there significant differences? If so, what might be the reasons for them? If not, why not?

Note: The solvers will only be tested with valid initial grids. The naive solvers `solve`, `arr_solve`, and `arr2_solve`, and `csolve` can only be expected to terminate in reasonable time when applied to almost completely filled initial grids.

Programming with Streams: Generators, Filters, Selectors, etc.

1. *Every Throw a Hit!*

   We throw darts at a dartboard with k differently numbered segments. There are no double or triple (value) segments. There is also no *bullseye* in the centre. Every segment can multiply be hit when n darts are thrown. Always true: Every throw hits, no throw fails (the dartboard)!

   Is it possible to reach with some number of darts a score of exactly m? Is it possible to reach with exactly n darts a score of exactly m? How many darts are at the minimum required to reach exactly a score of m?

To answer questions like these, generators, filters, transformers, and selectors shall be implemented and appropiately be combined:

```
type Nat1      = Int        -- Natural numbers starting from 1
type Numbers   = Nat1       -- Values of dartboard segments
type Dartboard = [Numbers]  -- Dartboard characterized by a list
                                of purely ascending values
type Turn        = [Numbers] -- Reached scores of a turn (Wurffolge); only
                             -- scores occurring on the dartboard are possible,
                             -- also more than once.
type Turns       = [Turn]    -- Stream of turns
type TargetScore = Nat1      -- Desired overall score > 0
type Throws      = Nat1      -- Number of darts of a turn > 0

gen_turns          :: Dartboard -> Turns
filter_turns_ts    :: Turns -> TargetScore -> Turns
filter_turns_th    :: Turns -> Throws -> Turns
select_turns_minl  :: Turns -> Turns
transf_sort_turns  :: Turns -> Turns
```

where the functions shall have the following meaning:

- `gen_turns`: Generates a stream of turns in accordance with the numbered segments of the dartboard, e.g., all turns with 1 dart, subsequently all turns with 2 darts, etc. Take care that your generator is fair and does not generate duplicates, i.e., generates every turn with a finite number of darts eventually, while not generating duplicates of turns in form of permutations (turns like [7,23,12], [23,7,12], [12,7,23], etc. are considered duplicates; only one of them shall be generated).
- `filter_turns_ts`: Filters the input stream for those turns, whose summed overall score matches the target score.
- `filter_turns_th`: Filters the input stream for those turns with the given number of darts.
- `select_turns_minl`: Picks from the input list the turns with the smallest number of darts.
- `transf_sort_turns`: Sorts the turns of the input stream descendingly.

Using the above generators, filters, transformers, and selectors, implement the following functions:

```
dart_ts   :: Dartboard -> TargetScore -> Turns
dart_tst  :: Dartboard -> TargetScore -> Throws -> Turns
dart_tsml :: Dartboard -> TargetScore -> Turns
```

which shall have the following meaning:

- `dart_ts` yields the (finite number of) turns reaching the target score.

- `dart_tst` yields the (finite number of) turns reaching the target score with the given number of darts.
- `dart_tsml` yields the (finite number of) turns reaching the target score with the smallest number of darts.

Each turn of a result list delivered by the functions `dart_ts`, `dart_tst`, and `dart_tsml` shall be ordered descendingly, the turns themselves shall be ordered lexicographically ascending.

Examples:

```
db = [6,7,16,17,26,27,36,37,46,47]
dart_ts db 23    ->> sort_lex [[7,16],[6,17]] ->> [[6,17],[7,16]]
dart_tst db 55 4 ->> sort_lex [[7,16,16,16],[6,16,16,17],[6,6,7,36],[6,6,6,37],...]
dart_tsml db 100 ->> sort_lex [[6,47,47],[7,46,47],[16,37,47],[17,36,47],[17,37,46],...]
dart_ts db 15    ->> []
```

Important: *Do not use self-defined modules!* If you want to re-use functions (written for earlier assignments), copy these functions to the new submission file. An `import` declaration for self-defined modules will fail, since only the submission file `assignment`i`.hs` , where i, $1 \leq i \leq 8$ (*tentatively*), denotes the running number of the assignment, willl be copied for the (semi-automatic) evaluation. No other file in addition to `assignment`i`.hs` will be copied.