Advanced Functional Programming: Assignment 4 (Wed, 05/15/2019)
Topic: Algorithm Patterns: Backtracking and Priority-first Search
Submission deadline: Wed, 05/22/2019 (3pm)

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 `assignment4.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

We reconsider the dartboard problem of Assignment 3 but this time, want to solve it using the algorithm patterns for *backtracking* and *priority-first search* instead of generators, selectors, filters, and transformers.

Problem recalled: We throw at a dartboard with k differently numbered segments. There are no double or triple (value) segments, and there is no *bullseye* in the centre. Throwing n darts, every segment can be hit multiple times. Always true: Every throw hits, no throw fails (the dartboard)!

```
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
```

Questions of interest: 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?

1. In order to answer these questions, implement 3 Haskell functions:

   ```
   bt_dart_ts   :: Dartboard -> TargetScore -> Turns
   bt_dart_tst  :: Dartboard -> TargetScore -> Throws -> Turns
   bt_dart_tsml :: Dartboard -> TargetScore -> Turns
   ```

   whose meaning coincides with those of their counterparts `dart_ts`, `dart_tst`, and `dart_tsml` of Assignment 3, whose implementations, howerver, make use of the higher order function for backtracking search:

   ```
   searchDfs :: (Eq node) => (node -> [node]) -> (node -> Bool)
                                                -> node -> [node]
   ```

its argument functions:

```
succ :: node -> [node]
goal :: node -> Bool
```

and possibly two further functions `sort :: Turn -> Turn` and `sort_lex :: Turns -> Turns` for sorting a turn descendingly and a sequence of turns lexicographically ascendingly, respectively.

To this end, define a data type:

```
data Node = ...
```

which carries enough information such that it can also be used for the following exercises, make it an instance of type class `Eq`, and implement three pairs of functions over it:

```
succ_ts :: Node -> [Node]
goal_ts :: Node -> Bool

succ_tst :: Node -> [Node]
goal_tst :: Node -> Bool

succ_tsml :: Node -> [Node]
goal_tsml :: Node -> Bool
```

such that `bt_dart_ts`, `bt_dart_tst`, and `bt_dart_tsml` get there intended meaning, when calling `searchDfs` together with one of these function pairs and the sorting functions for sorting a turn and a sequence of turns, i.e.:

- `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.

As in Assignment 3, each turn of a result list delivered by the functions `dart_ts`, `dart_tst`, and `dart_tsml` shall be ordered descendingly, the turns themselves lexicographically ascending. Depending on the choice of the arguments, the result of each of the functions may be the empty list, if there are no turns matching the requirements.

Examples:

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

2. The higher-order function `searchPfs` for priority-first search of Chapter 3.3 is designed to search for all solutions within a search space.

   Modifying the implementation of `searchPfs`, write a new higher-order function

   ```
   searchPfsFst :: (Ord node) => (node -> [node]) -> (node -> Bool)
                                                     -> node -> [node]
   ```

   which terminates the priority-first search once the first solution has been found. Since there may be no solutions at all in the search space, we keep the result type `[node]` of `searchPfs` for `searchPfsFst`, which allows us to indicate the result of a failed search by yielding the empty list as result.

3. Using `searchPfsFst`, implement two Haskell functions:

   ```
   psf_low  :: Dartboard -> Targetscore -> Turns
   psf_high :: Dartboard -> Targetscore -> Turns
   ```

   with the following meaning. Function `psf_low` yields the turn with the lowest-valued throws yielding the desired overall score, `psf_high` yields vice versa the turn with the highest-valued throws with this property. I.e., starting from the lowest-valued dartboard segment, the lowest-valued turn contains each value so many times such that taking this value again would prevent reaching the desired overall score. Vice versa, starting from the highest-valued dartboard segment, the highest-valued turn contains each value so many times such that taking this value again would prevent reaching the desired overall score. In any case the turns of the result lists of both functions shall be ordered ascendingly.

   To this end, make your data type `Node` an instance of the type class `Ord` and implement two pairs of argument functions:

   ```
   succ_low :: Node -> [Node]
   goal_high :: Node -> Bool

   succ_low :: Node -> [Node]
   goal_high :: Node -> Bool
   ```

   for the call of `searchPfsFst` in `psf_low` and `psf_high`. Is it possible to implement `psf_low` or `psf_high` without referring to `sort` and to possibly succeed with a shared function `goal` for resp. instead of two dedicated functions `goal_low` and `goal_high`? If so, you can implement one of the two functions in terms of the other one.

   Example:

   ```
   db = [6,7,16,17,26,27,36,37,46,47]
   psf_low db 55  ->> [[6,6,6,6,6,6,6,6,7]]
   psf_high db 55 ->> [[6,6,6,37]]
   ```

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.