FUNCTIONAL SPECIFICATION OF ALGORITHMS, LAB EXERCISES WEEK 2, PART 1

DAAN MULDER, 10279245

Mastermind

For the Mastermind implementation, we use the code given for the lab exercises as-is.

```
module Mastermind

where

import Data.List

data Colour   = Red | Yellow | Blue | Green  | Orange
                deriving (Eq,Show,Bounded,Enum)

data Answer   = Black | White deriving (Eq,Show)

type Pattern  = [Colour]
type Feedback = [Answer]

samepos :: Pattern -> Pattern -> Int
samepos _      []                = 0
samepos []     _                 = 0
samepos (x:xs) (y:ys) | x == y    = samepos xs ys + 1
                      | otherwise = samepos xs ys

occurscount ::  Pattern -> Pattern -> Int
occurscount xs []       = 0
occurscount xs (y:ys)
          | y `elem` xs = occurscount
                          (delete y xs) ys + 1
          | otherwise   = occurscount xs ys

reaction :: Pattern -> Pattern -> [Answer]
reaction secret guess = take n (repeat Black)
                     ++ take m (repeat White)
    where n = samepos secret guess
          m = occurscount secret guess - n
```

Then, we define some auxiliary functions that will be used for all exercises. `makeList` generates all possible combinations of n elements of xs. We use `makeList` to define `firstList`, which is a list of all possible patterns before the game has started. `guessing` takes a list *xs* of possible patterns and returns the subset of that list of patterns that are still possible after guessing *guess*.

```
makeList :: [a] -> Int -> [[a]]
makeList xs 1 = [[x] | x <- xs]
makeList xs n = [[x] ++ y | x <- xs, y <- makeList xs $ n-1]

firstList = makeList [Red,Yellow,Blue,Green,Orange] 4

guessing :: Pattern -> Pattern -> [Pattern] -> [Pattern]
guessing secret guess xs = filter (\x -> reaction x guess
                           == reaction secret guess) xs
```

Exercise 1. `exercise1` takes a pattern and returns the number of guesses it took to guess that pattern (and likewise for every exercise). It uses `exercise1play` that always guesses the first element of the current list of possible patterns, for which it uses `guessing` to calculate.

```
exercise1play :: Pattern -> [Pattern] -> Int -> Int
exercise1play secret (x:[]) n = if x == secret then n else -1
exercise1play secret (x:xs) n = exercise1play secret
                                (guessing secret x (x:xs)) n+1

exercise1 :: Pattern -> Int
exercise1 secret = exercise1play secret firstList 0
```

Exercise 2. For exercise 2 we use the following functions to transform the list of possibilities each round: `exercise2list` generates a list of tuples that connect each possible guess with a list of feedbacks for every possible secret, then groups each list to obtain the required partition. Then, `exercise2max` counts the number of elements in each block of the partitions and returns the maximum number for each possible guess. Finally, `exercise2min` returns the first possible guess that has the minimum number over all possible guesses (this function will be used in other exercises as well).

```
exercise2min :: [(Pattern, Int)] -> Pattern
exercise2min xs = fst $ (filter (\ (_,b) -> b
                  == minimum (map snd xs)) xs) !! 0

exercise2max :: [(Pattern, [[Feedback]])] -> [(Pattern, Int)]
exercise2max xs = map (\ (a,b) -> (a,maximum b)) $
                  map (\ (a,b) -> (a,map length b)) xs

exercise2list :: [Pattern] -> [(Pattern, [[Feedback]])]
exercise2list xs = map (\ (a,b) -> (a, group b)) $
                   [(maybeGuess, [reaction maybeSecret maybeGuess |
                   maybeSecret <- xs]) | maybeGuess <- xs]

exercise2play :: Pattern -> [Pattern] -> Int -> Int
exercise2play secret (x:[]) n = if (x == secret) then n else -1
exercise2play secret xs n = exercise2play secret (guessing secret
                            (exercise2min $ exercise2max $ exercise2list xs)
```

```
                                  xs) n+1

exercise2 :: Pattern -> Int
exercise2 secret = exercise2play secret firstList 0
```

Exercise 3. In exercise 3, we use `exercise3prep` to count the number of blocks of each partition generated by `exercise2list`. Afterwards, we use `exercise3max`, which is similar to `exercise2min` except that it returns the possible guess with the maximum number.

```
exercise3max :: [(Pattern, Int)] -> Pattern
exercise3max xs = fst $ (filter (\ (_,b) -> b == maximum (map snd xs)) xs) !! 0

exercise3prep :: [(Pattern, [[Feedback]])] -> [(Pattern, Int)]
exercise3prep xs = map (\ (a,b) -> (a,length b)) xs

exercise3play :: Pattern -> [Pattern] -> Int -> Int
exercise3play secret (x:[]) n = if (x == secret) then n else -1
exercise3play secret xs n = exercise3play secret (guessing secret (exercise3max $ exercise3prep $

exercise3 :: Pattern -> Int
exercise3 secret = exercise3play secret firstList 0
```

Exercise 4. `exercise4sum` counts the number of elements in each block of the partitions and then takes the sum of the squares. For comparison purposes, it is not necessary to divide by the number of total elements, since it will be the same for each possible guess (namely, the total number of currently possible guesses). Finally, we again use `exercise2min` to obtain the guess with the minimum number.

```
exercise4sum :: [(Pattern, [[Feedback]])] -> [(Pattern, Int)]
exercise4sum xs = map (\ (a,b) -> (a,sum $ map (^2) b)) $ map (\ (a,b) -> (a,map length b)) xs

exercise4play :: Pattern -> [Pattern] -> Int -> Int
exercise4play secret (x:[]) n = if (x == secret) then n else -1
exercise4play secret xs n = exercise4play secret (guessing secret (exercise2min $ exercise4sum $

exercise4 :: Pattern -> Int
exercise4 secret = exercise4play secret firstList 0
```

Exercise 5. `exercise5entropy` counts the number of elements in each block V_i of the partitions and then calculates $\sum \#(V_i) \cdot \log(\#(V_i))$. Again, it is not necessary to divide by the number of total elements, since it is the same for each possible guess. No satisfiable way was found to make the log's base depend on the size of the partition. Then, we use $exercise5min$ to find the minimum, which is similar to `exercise2min` except that it works with floats.

```
exercise5min :: [(Pattern, Float)] -> Pattern
exercise5min xs = fst $ (filter (\ (_,b) -> b == minimum (map snd xs)) xs) !! 0

exercise5entropy :: [(Pattern, [[Feedback]])] -> [(Pattern, Float)]
exercise5entropy xs = map (\ (a,b) -> (a,sum $ map (\ x -> fromIntegral x * (log $ fromIntegral :
```

```
exercise5play :: Pattern -> [Pattern] -> Int -> Int
exercise5play secret (x:[]) n = if (x == secret) then n else -1
exercise5play secret xs n = exercise5play secret (guessing secret (exercise5min $ exercise5entrop

exercise5 :: Pattern -> Int
exercise5 secret = exercise5play secret firstList 0
```