# 185.A05 Advanced Functional Programming SS 2020

Wednesday, 25 March 2020

## Assignment 2

### on Chapter 2, Chapter 7, and Chapter 3

**Topics:** Streams, Functional Arrays, Backtracking

**Submission deadline:** Monday, 27 April 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

   `Assignment2.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 2.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment2.hs`. Import declarations for self-defined modules will fail: Only `Assignment2.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.

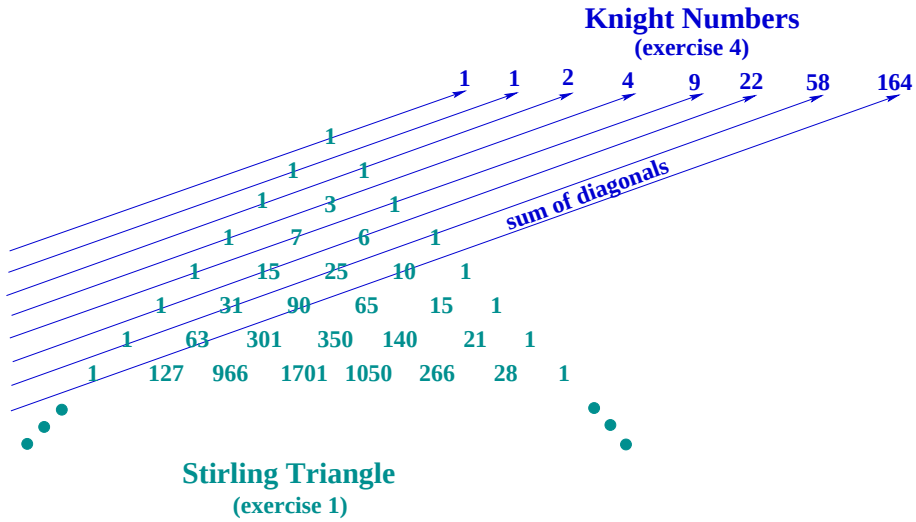
**Programming tasks:**

1. We consider the *Stirling numbers* introduced in assignment 1. This time, however, we want to implement a corecursive 0-ary Haskell function

   ```
   sta :: [Array Integer (Maybe Integer)]
   ```

   yielding the stream of rows of the Stirling triangle as array values, i.e., the stream of array values starting with:

```
[array (0,0) [(0,Just 1)],
 array (-1,1) [(-1,Just 1),(0,Nothing),(1,Just 1)],
 array (-2,2) [(-2,Just 1),(-1,Nothing),(0,Just 3),(1,Nothing),(2,Just 1)],
 array (-3,3) [(-3,Just 1),(-2,Nothing),(-1,Just 7),(0,Nothing),
               (1,Just 6),(2,Nothing),(3,Just 1)],
 array (-4,4) [...],...
```




*Examples:*

```
take 2 sta ->> [array (0,0) [(0,Just 1)],
                array (-1,1) [(-1,Just 1),(0,Nothing),(1,Just 1)]]
head (drop 3 sta) ->> array (-3,3) [(-3,Just 1),(-2,Nothing),(-1,Just 7),
                                    (0,Nothing),(1,Just 6),
                                    (2,Nothing),(3,Just 1)]
```

2. Implement a conversion function from arrays to lists, which strips off the constructor of `Just` values and drops `Nothing` values.

```
conv :: Array Integer (Maybe Integer) -> [Integer]
```

*Examples:*

```
conv (array (0,0) [(0,Just 1)]) ->> [1]
conv (array (-1,1) [(-1,Just 1),(0,Nothing),(1,Just 1)]) ->> [1,1]
conv (array (-3,3) [(-3,Just 1),(-2,Nothing),(-1,Just 7),
                    (0,Nothing),(1,Just 6),
                    (2,Nothing),(3,Just 1)]) ->> [1,7,6,1]
```

3. Using `conv` implement a function `pretty_print` for `[Array Integer (Maybe Integer)]` values:

```
pretty_print :: [Array Integer (Maybe Integer)] -> [[Integer]]
```

*Examples:*

```
pretty_print (take 5 sta) ->> [[1],[1,1],[1,3,1],[1,7,6,1],[1,15,25,10,1]]
pretty_print [head (drop 3 sta)] ->> [1,7,6,1]
pretty_print [rs | rs <- sta, mod (length (conv rs)) 2 /= 0]
 ->> [[1],[1,3,1],[1,15,25,10,1],...
```

4. Adding the numbers in the diagonals of the Stirling triangle, we get a stream of numbers, which we call the stream of *knight numbers* because proceeding to the next number in a diagonal resembles the move of a knight in the chess game (Note: if we were to use the Pascal triangle instead of the Stirling triangle we would get the tail of the stream of Fibonacci numbers this way).

   Implement a corecursive 0-ary Haskell function `kns :: [Integer]` computing the stream of knight numbers as suggested by the above figure, i.e., evaluating `kns` shall yield the stream of numbers starting with:

   ```
   kns ->> [1,1,2,4,9,22,58,164,...
   ```

   *Examples:*

   ```
   take 5 kns ->> [1,1,2,4,9]
   head (drop 3 kns) ->> 4
   [n | n <- kns, mod (length n) 2 /= 0] ->> [1,1,9,...
   ```

5. The *Modified Post's Correspondence Problem* (*MPCP*) is the following: Given two arbitrary lists $A$ and $B$, each of $k$ nonempty strings $s_i$, $t_i$, $0 \leq i \leq k-1$, each string in the regular set of strings $\{0,1\}^+$, say:

   $$A = \langle s_0, s_1, s_2, \ldots, s_{k-1} \rangle \quad B = \langle t_0, t_1, t_2, \ldots, t_{k-1} \rangle$$

   Does there exist a sequence of integers $i_1, i_2, \ldots, i_r$ such that the concatenation of $s_0, s_{i_1}, s_{i_2}, \ldots s_{i_r}$ equals the one of $t_0, t_{i_1}, t_{i_2}, \ldots, t_{i_r}$, i.e.:

   $$s_0 s_{i_1} s_{i_2} \ldots s_{i_r} = t_0 t_{i_1} t_{i_2} \ldots t_{i_r} \, ?$$

   If there is no upper limit on the value of $r$, then *MPCP* is undecidable. If, however, there is such a limit it is decidable.

   For $q \in \mathbb{N}_0$, we introduce the problems $MPCP_{=q}$ and $MPCP_{\leq q}$, which are simplermversions of *MPCP* imposing a limit on the length of the sequence of integers, i.e., on $r$.

   We say that

   (a) $MPCP_{=q}$ has a solution if there is a sequence of integers $i_1, i_2, \ldots, i_q$ of length $q$ such that: $s_0 s_{i_1} s_{i_2} \ldots s_{i_q} = t_0 t_{i_1} t_{i_2} \ldots t_{i_q}$.

   (Note: For $q = 0$, this requirement boils down to $s_0 = t_0$.)

   (b) $MPCP_{\leq q}$ has a solution if there is a sequence of integers $i_1, i_2, \ldots, i_p$, $p \leq q$, such that: $s_0 s_{i_1} s_{i_2} \ldots s_{i_p} = t_0 t_{i_1} t_{i_2} \ldots t_{i_p}$.

Implement two Haskell functions `mcpc_eq` and `mcpc_le` as decision procedures of $MPCP_{=q}$ and $MPCP_{\leq q}$, respectively:

```
type Nat0 = Int
type NullsOnes = String -- Only nonempty strings over {`0`,`1`}

mcpc_eq :: ([NullsOnes],[NullsOnes]) -> Nat0 -> Maybe [Int]
mcpc_le :: ([NullsOnes],[NullsOnes]) -> Nat0 -> Maybe [Int]
```

If an instance of $MPCP_{=q}$ or $MPCP_{\leq q}$ does not have a solution, then `mcpc_eq` and `mcpc_le` shall yield `Nothing`. If there is more than one solution then `mcpc_eq` and `mcpc_le` shall yield a solution of minimum length. If there is more than one solution of minimum length then `mcpc_eq` and `mcpc_le` shall yield among those the one which is lexicographically the smallest one.

Use backtracking to implement `mcpc_eq` and `mcpc_le`.

*Examples:*

```
mcpc_eq (["00","11","010"],["11","00","010"]) 0 ->> Nothing
mcpc_eq (["00","11","010"],["00","11","010"]) 0 ->> Just []
   (because: "00" == "00"
mcpc_eq (["00","11","010"],["0","1011","00"]) 2 ->> Just [2,1]
    (because: "00"++"010"++"11"  ==  "0"++"00"++"1011"

mcpc_le (["00","11","010"],["0","1011","00"]) 1 ->> Nothing
mcpc_le (["00","11","010"],["0","1011","00"]) 2 ->> Just [2,1]
mcpc_le (["00","11","010"],["0","1011","00"]) 3 ->> Just [2,1]

mcpc_le (["00","1","1"],["0","01","01"]) 2 ->> Just [1]
  (because: "00"++"1" ==  "0"++"01"
   note: Just [2] is a solution, too, but [2] is lexicographically
         larger than [1])
mcpc_le (["00","1","1","11"],["0","01","1","011"]) 2 ->> Just [1]
mcpc_le (["00","1","1","11"],["0","0","11","011"]) 2 ->> Just [3]
  (because: "00"++"11" ==  "0"++"011"
   note: Just [1,2] is a solution, too, but [3] is shorter than [1,2]
         notwithstanding of being lexicographically larger; Just [1]
         is not a solution as in the example before).
```

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

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