UNIVERSITÄT BASEL

Lecturer: Thorsten Möller - **thorsten.moeller@unibas.ch**
Tutors: Miruna Muntean - **miruna.muntean@unibas.ch**
Mark Starzynski - **mark.starzynski@unibas.ch**

Programming Paradigms – Haskell FS 2023

Exercise 3 Due: 14.05.2023 23:55:00

Upload your answers to the questions **and source code** on Adam before the deadline.

Text : For answers to questions, observations and explanations, we suggest writing them in LaTeX. Please hand-in your answers as a **single PDF** file (independent of what tools you use, LaTeX, Markdown etc.).

Source-Code : For coding exercises, the source-code must be provided and has to be **commented in detail** (e.g. how it works, how it is executed, comments on conditions to be satisfied).

Upload : Please archive multiple files into a **single compressed zip-file**. If you upload an updated version of your solutions, the file name should contain a clear and intuitive versioning number. Only the latest version will be graded.

Requisit : In order to take the final exam, you must score at least $2/3$ of all available points throughout the mandatory exercises.

Modalities of work: The exercise can be completed in groups of at the most 2 people. Do not forget to provide the full name of all group members together with the submitted solution.

Question 1: Bite-sized Haskell Tasks (7 points)

Write a Haskell function for each of the following tasks.

We want to keep the solutions to these problems as compact as possible; so an additional restriction is that they must not make use of any additional helper functions other than those that are already pre-defined in Haskell. Put all your functions into a single file and demonstrate them in `main`.

a) Swap the values of the first and last element in a list.

(1 points)

b) Return, as a Boolean, whether 2 consecutive elements in a list are the same.

(1 points)

c) Return a list containg all divisors of a given positive integer.

(1 points)

d) Remove all odd numbers from a list.

(1 points)

e) Calculate the mean of the even numbers in a list.

(1 points)

f) Generate a list of tuples (n, s) where $0 \leq n \leq 30$ and where $s = n^2$; i.e., the output should be the list [(0,0),(1,1),(2,4),(3,9),...,(30,900)].

(1 points)

g) Take two positive integers and return a list containing their shared prime factors.

(1 points)

Question 2: Lazy Evaluation (Call-By-Need/Infinite List) (3 points)

a) Using the concept of lazy evaluation, write a function that returns the infinite list of natural, even numbers. How does Haskell handle infinite lists?

(1 points)

b) Write a function that takes the list from (a) as input and returns a new list containing tuples, such that each tuple has the original value and its result by multiplying by 2. (e.g. `[0, 2, 4, ...]` → `[(0, 0*2), (2, 2*2), (4, 4*2), ...]`)

(1 points)

c) Lastly, write a function that takes the infinite list of tuples from (b) and an Integer as arguments. This function should return the first n entries of the list for which the multiplication by 2 is less or equal to the given Integer.

(1 points)

Hint: There are many ways to solve this task, you can read up on `iterate`, `map`, `take` and `takeWhile` for inspiration.

Question 3: Pattern Matching and Guards (5 points)

The Ackermann function[1] is an example of a total computable function that is not primitive recursive[2]. In this exercise we will use Ackermann's three-argument function:

$$\varphi(m,n,p) = \begin{cases} \varphi(m,n,0) = m+n \\ \varphi(m,0,1) = 0 \\ \varphi(m,0,2) = 1 \\ \varphi(m,0,p) = m, \quad \text{for } p > 2 \\ \varphi(m,n,p) = \varphi(m,\varphi(m,n-1,p),p-1), \quad \text{for } n,p > 0 \end{cases}$$

a) Implement Ackermann's three-argument function using guards.

(2 points)

b) Write the function `ackermannList`. This function should take a list of integers as input and apply the following function to it.

$$F(s) = \begin{cases} 13 & \text{if } s = () \\ (\varphi(x,0,3)) & \text{if } s = (x) \\ (\varphi(x,y,0) & \text{if } s = (x,y) \\ (\varphi(x,y,z) & \text{if } s = (x,y,z) \\ (\varphi(x,y,z))^\frown F(...) & \text{if } s = (x,y,z,...) \end{cases}$$

Where $^\frown$ is the list concatenation and s is the input list.

(3 points)

Question 4: Sorting (5 points)

a) Write a function sorted that takes a list of comparable values and returns whether the list is sorted or not.

(1 points)

b) Implement the function `mergesort`, which takes a list of comparable values and returns the sorted list. As the name already suggests, you should use *merge sort*[3].

Since we want the implementation to be as efficient as possible, the sorting should stop at the latest one iteration after the list was sorted correctly.

[1] **https://en.wikipedia.org/wiki/Ackermann_function**
[2] **https://en.wikipedia.org/wiki/Primitive_recursive_function**
[3] **https://en.wikipedia.org/wiki/Merge_sort**

(2 points)

c) Implement a function `bubblesort`, which takes a list of comparable values and returns the sorted list, sorted using *bubble sort*[4].

   Since we want the implementation to be as efficient as possible, the sorting should stop at the latest one iteration after the list was sorted correctly.

(2 points)

Question 5: Recursion, Map and Fold (7 points)

The function `map` takes an unary function and a list of values as arguments. The given function is then applied to each element in the list.

Similarly, the function `filter` takes a predicate (a function that returns a Boolean) and a list of values. The resulting list only contains the values that satisfy the given predicate. (i.e. for which the return value of the predicate is `True`).

In the following tasks, you will write your own implementations of `map` and `filter` and use them. For obvious reasons, you are not allowed to use the built-in Haskell `map` and `filter` function.

Hints: It might be helpful to look up their function signatures. You can do this with the `:type` command in the Haskell interpreter.

Recursion is a commonly used concept in Haskell and is very useful for solving the following tasks.

a) Write your own implementation of the `map` and `filter` functions.

(2 points)

b) Consider the following list of weights (kg) and heights (cm).

   Weights: [97, 95, 53, 40, 46, 73, 68, 67, 69, 62, 99, 89, 77, 94, 100, 63, 75, 44, 81, 96, 63, 99, 64, 88, 77, 104, 50, 97, 81, 84, 66, 78, 72, 93, 42, 93, 94, 58, 48, 94, 71, 47, 49, 57, 93, 71, 106, 63, 81, 95]

   Heights: [172, 195, 157, 144, 165, 191, 183, 194, 185, 194, 183, 194, 186, 152, 156, 163, 179, 164, 186, 181, 158, 181, 155, 197, 150, 166, 179, 153, 142, 154, 162, 156, 149, 166, 148, 195, 198, 152, 167, 197, 157, 144, 147, 159, 198, 149, 179, 183, 156, 175, 157]

[4] `https://en.wikipedia.org/wiki/Bubble_sort`

Write a main function that uses your implementation of the map and filter functions to calculate the BMI of these weight-height pairs and find the indexes of all individuals that suffer from overweight (BMI > 25) and underweight (BMI < 18.5). Print lists of both calculated BMIs and indexes of overweight/underweight individuals.

Hint: The functions `zip` and `zipWith` might be useful. To retrieve the index of the individuals in the original list it might be a good idea to pair each data point with its index and using list comprehension to extract only the indexes.

(3 points)

c) Write a function that calculates the average value in a list and use it to calculate the average height, weight and BMI in the given data.

Hint: You might want to use the function `foldl` or `foldr`.

(2 points)

Question 6: Currying (5 points)

In the following tasks you are asked to answer questions about *currying*, a central concept of Haskell and other functional programming languages.

a) Explain the concept of currying and state its benefits. Is it possible to generate a curried version of any function? What do the Prelude functions curry and uncurry do?

(2 points)

b) Explain what the following Haskell code does. How is this related to *currying*?

```
foo = \x -> (\y -> x * y)
```

(1 points)

c) Provide the curried version of the following function and demonstrate the use of both the curried and uncurried function.

```
foo :: (Bool, Int, Int) -> Int
foo (x, y, z)
  | x = y + z
  | otherwise = y - z
```

(2 points)

Question 7: Steganography (8 points)

Steganography (**https://en.wikipedia.org/wiki/Steganography**) is the practice of concealing information within another kind of information. For this task you will write a Haskell program to uncover the hidden message from the following three integer lists:

X = [18, 68, 36, 36, 20, 67, 36, 20, 36, 35, 68, 20, 20, 36, 68, 33, 65, 20, 20, 35, 36, 17, 36, 65, 36, 17, 68, 20, 68, 33, 33, 19, 20, 35, 67, 33, 35, 18, 68, 20, 36, 68, 19, 36, 65, 68, 36, 20, 68, 35, 20, 20, 35, 17, 36, 68, 17, 68, 36, 33, 33]

Y = [19, 34, 66, 32, 34, 20, 67, 19, 65, 36, 35, 33, 34, 66, 19, 19, 17, 18, 34, 22, 35, 65, 34, 36, 19, 65, 18, 34, 64, 65, 17, 68, 19, 33, 68, 33, 64, 64, 18, 36, 33, 18, 71, 16, 65, 32, 36, 16, 66, 36, 17, 35, 37, 65, 19, 66, 17, 64, 34, 33, 33]

Z = [34, 65, 32, 67, 20, 66, 33, 66, 35, 18, 65, 16, 17, 32, 64, 36, 66, 33, 16, 35, 70, 18, 32, 35, 17, 66, 65, 16, 33, 34, 18, 67, 18, 36, 64, 34, 66, 66, 16, 33, 20, 65, 20, 33, 34, 65, 64, 65, 20, 20, 32, 16, 65, 18, 33, 33, 66, 35, 17, 19, 19]

The information is hidden as follows: Each list $X = [X_1, ..., X_n]$, $Y = [Y_1, ..., Y_n]$ and $Z = [Z_1, ..., Z_n]$ contains information about the hidden message. The arrays are in order, meaning that X_i, Y_i and Z_i contain information about the same part of the message.

Find the correct list of integers calculated from X, Y, Z and use the ASCII encoding to generate a string. X, Y and Z encode the solution list $S = [S_1, ..., S_m]$ in base 5 where X encodes the highest digit and Z the lowest. A digit that is higher than the given base is still correct in this encoding. A 7 in the second digit is equivalent to $7 * 5^1$. You additionally get a decipher key that holds further instructions:

- Ignore any bits of X_i, Y_i, Z_i that have a higher value than the 4th bit. For example: For any X_i only the value of he lowest four bits $x_{i4}, x_{i3}, x_{i2}, x_{i1}$ matter. This means that 67 corresponds to a 3.
- **Important:** Remove the encoding where the unimportant/throwaway bits of Y_i and Z_i are the same.

Decoding a single character by hand would look like this:

$$X_i = 66, \quad Y_i = 7, \quad Z_i = 17 \quad \rightarrow \quad S_i \ = \ 2 * 5^2 + 7 * 5^1 + 1 * 5^0 \quad \rightarrow \ 'V'$$

a) Write the function `getAsciiChars` that takes a list of integers and returns a list of the corresponding ASCII characters. You may want to use the function `chr` from `Data.Char`.

(2 points)

b) Write the function `uncover` that extracts the hidden information from the three integer lists. Use the bitwise-and `(.&.)` from `Data.bits`. The resulting string should make sense.

Hint: You need a bit mask to extract the important bits.

(6 points)

Question 8: Tic-Tac-Toe (Haskell) (10 points)

In this task - exactly same as you did in the sheet before with Python - you have to implement the game Tic-Tac-Toe [5] on the command line, this time written **entirely** in the Haskell programming language.

Write a Haskell program that allows you to play the game on the command line.

Your program should have the following features:

- Starting player is chosen at random.
- If the board is filled completely without a winner, the game starts anew with the opposite player starting.
- Player can choose whether to play with another human player or against a computer AI.
- How you implement the AI is up to you, at its simplest you can make the computer play at random.
- Game conditions like restart, quit game, change mode (AI or 2nd player) should be implemented.
- Inputting the field of choice can be done via the numpad, e.g. for the lower left field you can press `1+ENTER` and for the middle field you press `5+ENTER` etc (sum of row and column number).

Game session start :

```
Turn of Player X
    1   2   3
  -------------
6 |   |   |   |
  ----+---+----
3 |   |   |   |
  ----+---+----
0 |   |   |   |
  -------------
```

Within a game session:

```
Turn of Player O
    1   2   3
  -------------
6 |   | X | O |
  ----+---+----
3 |   |   | X |
  ----+---+----
0 |   |   |   |
  -------------
```

You may implement only one of the two modes, but it is crucial that the console GUI and controls are the exact same as in your previous Python implementation and according to the provided description. Look at your Python code and compare. Write down your opinion(s) about the comparison, what was easier for you, what more cumbersome, how do the implementations differ?

[5] **https://en.wikipedia.org/wiki/Tic-tac-toe**

Hint: Instead of `system.Random`, where you would have to use Stack as build tool, you can create a random generator with `Data.Time.Clock.POSIX` for example:

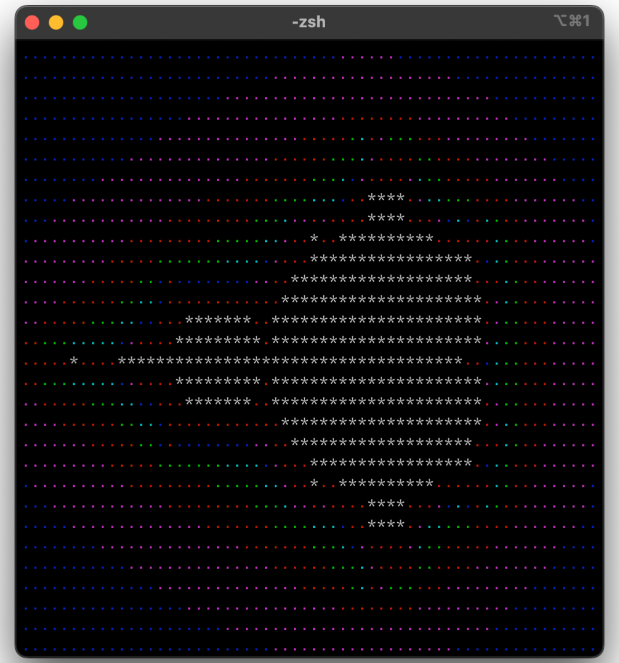
```haskell
import Data.Time.Clock.POSIX

main = do
    putStr "Random number between 0 and 9: "
    time <- getPOSIXTime
    let x = floor time `mod` 10
    putStrLn $ show x
```

You are done, rest of sheet is optional (one fantastic exercise if you have the time ☺).

Question 9: Mandelbrot set (Optional) (0 points)

Implement a Haskell program that computes the *Mandelbrot set*[6] over a range of complex numbers and print a visualization to the console. This will result in an ASCII art pattern as shown in Figure 1. Color the numbers that are not part of the set by counting how many iterations it takes for each of them to 'escape' (more details below).



```
............................................................
............................................................
............................................................
............................................................
............................................................
............................................................
............................................................
....................................****....................
....................................****....................
..............................*..**********.................
..............................*****************.............
............................*******************.............
...........................*********************............
.................*******..**********************............
................*********.**********************............
.....*....************************************..............
................*********.**********************............
.................*******..**********************............
...........................*********************............
............................*******************.............
..............................*****************.............
..............................*..**********.................
....................................****....................
....................................****....................
............................................................
............................................................
............................................................
............................................................
............................................................
............................................................
```

Figure 1: Mandelbrot set

[6] https://en.wikipedia.org/wiki/Mandelbrot_set

Use the following definition to determine whether a given complex number is part of the Mandelbrot set:

$$z_{n+1} = z_n^2 + c$$

A complex number c is part of the Mandelbrot set if the absolute value of z_n remains bounded $\forall n > 0$, when starting with $z_0 = 0$. You can do this by checking if $|z_{100}| < 10$.

Draw the set from $-2 \leq x \leq 1$ with a step size of 0.05 and from $-1.5 \leq y \leq 1.5$ with a step size of 0.1. Transform each point (x, y) to $z_0 = x + yi$ and check whether it belongs to the Mandelbrot set (*) or if it escapes (.). Remember that the origin of the coordinate system should be in the center of the terminal.

a) Write a function that prints the Mandelbrot set (as described above) to the console. **Hint:** The function `putStrLn`[7] might be useful.

(0 points)

b) Write a function that determines how many iterations it takes for a given complex number (that is not part of the Mandelbrot set) to escape. Do this by counting the number of iterations n until $|z_n| > 10^3$. Finally, use this function to add colors to your plot. You can color a string `s` by wrapping it as follows (Python code):

```
s = '\x1b[31m' + s + '\x1b[0m'
```

Where `31` defines the color (see first table below). More available colors are in the first table. We used the mappings from the second table for Figure 1 (you may also use your own).

color	number
black	30
red	31
green	32
yellow	33
blue	34
mangenta	35
cyan	36
white	37

num iterations	color
4	blue
5	mangenta
6	red
7	green
8	cyan
9	blue
10	mangenta
otherwise	red

E.g. `'\x1b[35m'` is mangenta.

(0 points)

[7] **https://wiki.haskell.org/Introduction_to_Haskell_IO/Actions**