UNIVERSITÄT BASEL

Lecturer: Thorsten Möller - **thorsten.moeller@unibas.ch**
Tutors: Nafia Nusrat - **nafia.nusrat@unibas.ch**
Sexhi Picaku - **s.picaku@stud.unibas.ch**
Mark Starzynski - **mark.starzynski@unibas.ch**

# Programming Paradigms – Haskell FS 2025

## Exercise 3 Due: 18.05.2025 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**, preferrably just the files without any folder hierarchy. The naming convention for your zip file is **cs109-e<ExNr>-<Name1>-<Name2>.zip**. 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: Easy Haskell Tasks (8 points)

Write a Haskell function for each of the following tasks. All the tasks should be written within one file, and a single main method should be used to test them.

a) Compute the mean of a list containing fractional numbers.

**(1 points)**

b) Remove all occurrences of a given element from a list.
(For example, removing 5 from [5, 2, 8, 4, 5] should return [2, 8, 4])

**(1 points)**

c) Replace the first and last characters of a string with `"<"` and `">"`.
(For example, `"Functional Programming!"` turns into `"<unctional Programming>"`).

**(1 points)**

d) Generate a list containing the first n prime numbers.

**(1 points)**

e) Determine whether two given numbers are relatively prime, a.k.a. coprime

**(2 points)**

f) Print all numbers below 1000 that are multiples of either 6 or 11.

**(2 points)**

## Question 2: Infinite Lists in Haskell (5 points)

a) Write a function that generates an infinite sequence of prime numbers using Haskell's lazy evaluation.
Why is recursion preferable in this case instead of a loop?
Additionally, print the first 15 prime numbers to the console.

**(1 points)**

b) In the same program, define a function that generates an infinite sequence of consecutive prime pairs (i.e., (2,3), (3,5), (5,7), (7,11), ...)

**(2 points)**

c) Write a function that extracts all prime pairs where the difference between the two elements is exactly 4. (For example, (3,7), (7,11), (13,17), ...)

**(2 points)**

## Question 3: Verhoeff Checksum Validation (4 points)

When people need to enter credit card or identification numbers, the number is usually checked whether it is valid or not. To perform this check, we will use the *Verhoeff* algorithm. A number is valid when it fulfills the following conditions:

- The number should be 16 digits long.
- Using the *Verhoeff* checksum formula, the final check digit should be valid (this algorithm uses Damm's permutation table and multiplication table to validate the number)
- Implement a function that validates a given 16-digit number using the *Verhoeff* algorithm.

For example, "236" is valid, while "1234" is invalid.

## Question 4: Haskell Time System (6 points)

a) Write a function that takes a number of seconds as input and converts it into a formatted 24-hour time representation (HH:MM:SS).

- For example, entering 3661 should return "01:01:01" (1 hour, 1 minute, and 1 second).
- Use pattern matching for better readability.

**(3 points)**

b) Extend the function to display the corresponding day of the week, assuming that the starting day is Monday.The time calculation should take into account days passing. If a user enters 90000 seconds, the output should include "01:00:00 Tuesday", since it exceeds 24 hours.Use list comprehensions to map numbers to weekdays.

**(1 points)**

c) Modify the program so that after entering a number of seconds, the user is asked to enter an amount of seconds to subtract.

- The updated time should be displayed correctly.
- If subtracting seconds moves into the previous day, indicate the change in day (e.g., "23:59:50 Sunday").
- Use guards (|) or where clauses to structure the logic.

**(2 points)**

## Question 5: Temperature Analysis (map, filter) (6 points)

The function `map` takes an unary function and a list of values as argumnets. 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 conatins 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.

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

**Hint:** Recursion is a valuable concept that can often be applied in Haskell and may help you solve the first two subtasks.

a) Write your own implementation of the `map` function.

**(1 points)**

b) Write your own implementation of the `filter` function.

**(1 points)**

c) You are provided with daily emperature data which is given as pairs `(high, low)` in degrees Celsius:

```
temps = [(32,23), (12, 5), (17,10), (29,20), (25,15),
         (10,3), (15, 8), (31,19), (35,27), (22,12)]
```

- Define a function `avgTemp` that takes a day's `(high, low)` and computes the average temperature for that day.
- Use your custom `map` function from a) to apply `avgTemp` to the full list of temperature data. The output should be a list of averages.
- Write another function `classifyTemp` that classifies a day's average temperature as either "cold" (if the average is below 10 degrees Celsius), "hot" (if the average is above 25 degrees Celsius), or "moderate" (otherwise). The output should be a list of day-by-day labels, for example `["hot","cold",...]`.

**(3 points)**

d) Use your custom `filter` and your classification function to extract all hot days from the list, and check whether more than half of the days in the dataset are labeled as "hot". Depending on whether this is true or not, you can output a message like "Heat wave!" or otherwise "No heat wave.".

**(1 points)**

## Question 6: Currying **(6 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*. Is it possible to generate a curried version of any function? What are the benefits of currying a function?

**(3 points)**

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

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

**(2 points)**

c) Provide the curried version of the following function:

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

**(1 points)**

## Question 7: Steganography (8 points)

**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 three integer lists that are provided to you in `encrypted_lists.txt` (or via this link: **https://pastebin.com/raw/TJTAzuUw**)

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 195 corresponds to a 3 → 195 in binary is `11000011` $(1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^1 + 1 \cdot 2^0)$. We consider only the lower nibble (the lower four bits) for the final value, the higher bits are thrown away.
- **Important:** Remove the encoding where the unimportant/throwaway bits of $X_i$ and $Y_i$ are the same.

Decoding a single character by hand would look like this:

$$X_i = 66, \quad Y_i = 7, \quad Z_i = 17 \quad \rightarrow \quad S_i = 2 * 5^2 + 7 * 5^1 + 1 * 5^0 \quad \rightarrow \quad 86 \quad \rightarrow \ 'V'$$

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.

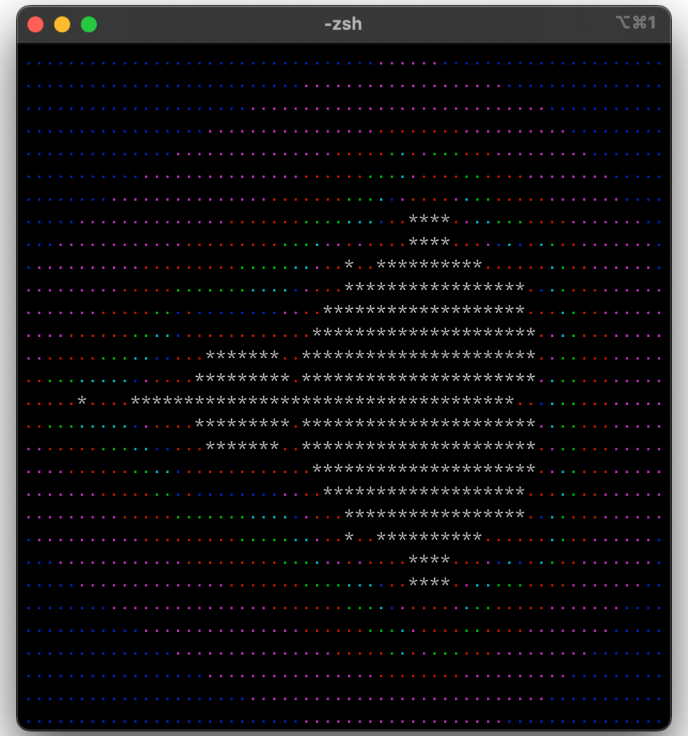
**(5 points)**

c) What specific Haskell language property comes to mind when reading the decyphered text? Explain this property in your own words.

**(1 points)**

## Question 8: Mandelbrot set (7 points)

Implement a Haskell program that computes the *Mandelbrot set*[1] 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

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.

[1] **https://en.wikipedia.org/wiki/Mandelbrot_set**

a) Write an action that prints the Mandelbrot set (as described above) to the console. **Hint:** The action `putStrLn`[2] might be useful.

**(4 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:

```
s = "\ESC[31m" ++ s ++ "\ESC[0m"
```

Where `31` defines the color. More available colors are:

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

E.g. `"\ESC[35m'` is mangenta.

We used the following mappings for Figure 1 (you may also use your own):

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

**(3 points)**

**Hint:** You will need to execute your program on a terminal that supports ANSI escape codes, otherwise you won't render the colors. You can check this with `colorcheck.hs`.

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

## Question 9: Tic-Tac-Toe...again! (Optional) (0 points)

In the last exercise, you have implemented the Tic-Tac-Toe game in Python. Now you may try doing the same in Haskell. This exercise is optional, but completing it will greatly enhance your understanding of the paradigm shift that occurs between these two languages.

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 |   |   |   |
  -------------
```

**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
```