# Programming Fundamentals 3
# BAINFOR-17
# Mid-term WS2025–26

Dr Afonso Arriaga, Dr Miroslav Kratochvíl

November 12th 2025

## Rules

- Duration: 90 minutes (1h30).
- Please ensure that you write your **full name** and **student ID** on all provided pieces of paper.
- Ensure clear legibility of your answers; prefer shorter and precise answers. Ambiguous answers will be evaluated as wrong.
- Please write your answers only on the provided pieces of paper.
  - If you run out of writing space, ask for additional paper.
  - If you write your answer to a different place than the reserved space below the question, kindly label all your answers clearly with the corresponding question number, such as *"Q1.1 (a)"*.
- You may have one (1) cheat-sheet on an A4 sheet of paper, with any text you like, on both sides. Please allow the examiners to check your cheat-sheet.
- All communication is strictly forbidden; the only exception is questions that you directly ask the examiners. Use of any electronic devices is strictly forbidden, including for reasons unrelated to PF3, such as for measuring time.
  - When asking questions, do not say any part of your answers aloud; this communicates your solution to others and is thus strictly forbidden. Instead, the examiner will come and look at your answers in your solution sheets.
  - If you think you have a valid reason to be exempt from some of the rules, this reason needs to be agreed upon with the examiners in advance.
  - Attempts to invent exceptions to the rules, or to trigger possible ambiguous interpretations of the rules, will be evaluated as cheating attempts.
  - Ideally, turn off all your electronic devices and either stash them away into your backpack or handbag, or place them on a desk near yourself, so that they can be watched by the examiners. Similarly, place your A4 cheat-sheet and any possible food, drink and backup writing tools (pens, pencils) on the desk too. Please avoid manipulating any objects out of the sight of the examiners (e.g., under the desk or in your backpack).

# Question 1 (Warm-up quiz)

Mark all the correct answers (possibly multiple ones, or none).
(In case you make a mistake and need to change the marking, **avoid double marking or mark corrections!** Instead, explain the correct answer using words in the empty space next to the list, explicitly referring to the letter labels of the desired answers.)

**Q1.1** (1 point) Which of the following are valid lists in Haskell?

(a) `[("five", [5])]`
(b) `[["five", [5]]]`
(c) `[[],"zero"]`
(d) `[7] ++ []`
(e) `[1 ++ 2 ++ 3]`

**Q1.2** (1 point) The expression `[[],[5],[6]]` has type:

(a) `[Int]`
(b) `[[], [Int], [Int]]`
(c) `([], [Int], [Int])`
(d) `[[Int]]`
(e) `[Float]`

**Q1.3** (1 point) A function of type `(Char, String) -> [Bool] -> Int`

(a) ...accepts two arguments, one at a time.
(b) ...takes a function as its first argument.
(c) ...only works if the second argument is a list that contains a single Boolean value.
(d) ...can be partially applied to a tuple of a Char and String.
(e) ...is not a valid Haskell function type.

**Q1.4** (1 point) The expression `Node 123 (Leaf "here") (Leaf "there")` is a valid value of the datatype:

(a) `data Tree Int = Leaf String | Node Int`
(b) `data Tree = Leaf Tree | Node Int Int`
(c) `data Tree = Leaf String | Node Int Tree Tree`
(d) `data Tree a = Leaf a | Node a (Branch a) (Branch a)`
(e) `data Tree n l = Leaf l | Node n (Tree n l) (Tree n l)`

# Question 2 (Prelude)

**Q2.1** (2 points) Define the following functions from `Prelude` using recursion:

(a) `length :: [a] -> Int`

(b) `sum :: [Int] -> Int`
(Note: The type of `sum` in Prelude is actually `(Foldable t, Num a) => t a -> a`, but here we do not require an implementation for such fully generic type.)

**Q2.2** (2 points) Define the following functions from `Prelude` using `foldr`:

(a) `concat :: [[a]] -> [a]`

(b) `filter :: (a -> Bool) -> [a] -> [a]`

**Q2.3** (2 points) Consider the definition of `countNums` below.

```
countNums :: String -> Int
countNums "" = 0
countNums (c:cs)
    | c `elem` ['0'..'9'] = 1 + countNums cs
    | otherwise = countNums cs
```

(a) Show how the expression `countNums "1two3"` is successively evaluated to simpler expressions, explaining how the final result is obtained. (Write answer in form `countNums "1two3" = ... = ... = ... = 2`.)

(b) Reimplement the function without recursion, using `length` and `filter`.

**Q2.4** (3 points) What are the types of the following expressions? (Some definitions admit multiple types, in such case report preferably the most generic one that will still type-check.)

(a) `"one, two, three"`

(b) `minimum`

(c) `("1", '1')`

(d) `([Just 'a', Just 'b', Nothing], ['a', 'b'])`

(e) `zip [1..5] "hello"`

(f) `False:True:[]`

# Question 3 (Algorithms)

**Q3.1** (1.5 points) Selection sort is a simple (yet inefficient) sorting algorithm that repeatedly finds the minimum element in the input list and removes it, consecutively appending the minima to the output to obtain a sorted list. The process repeats until all elements of the input list are removed:

```
selSort [] = []
selSort xs =
  let (m, xs') = popMinimum xs
   in m : selSort xs'

popMinimum [x] = (x, [])
popMinimum (x:xs) =
  let (y, ys) = popMinimum xs
   in if x < y
        then (x, xs)
        else (y, x : ys)
```

Write the type signatures of `selSort` and `popMinimum`:

**Q3.2** (1 point) Show how expression `popMinimum "lol"` is evaluated. The function `popMinimum` is recursive and gets invoked total 3 times. Fill in the exact values of arguments used in the 'calls', and the corresponding logic and 'return' values. (The first one is filled in for you.)

- call: `popMinimum "lol"`
  - call: `popMinimum` .....
    - call: `popMinimum` .....
    - returns: .....
  - condition evaluated in `if`: .....
  - returns: .....
- condition evaluated in `if`: .....
- returns: .....

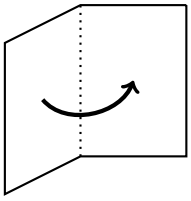
**Q3.3** (0.5 points) A sorting algorithm is considered stable if it preserves the relative order of elements that compare as equal. The provided definition of `selSort` is not a "stable" sorting algorithm. Propose a simple fix to the code that makes `selSort` stable.

**Q3.4** (4 points) Define an infinite list `paperSizes :: [(String, (Float, Float))]` that contains the names and sizes of all A-series papers (A0, A1, A2, A3, A4, ..., all the way to the legendary sub-atomic A∞).

All A-series papers are rectangles with sides in ratio $1 : \sqrt{2}$. Size of A0 is defined so that A0 is exactly 1 square meter in area; thus its longer side measures $\sqrt{\sqrt{2}} \approx 1.189$ meters, and shorter side measures $\frac{\sqrt{\sqrt{2}}}{\sqrt{2}} = \frac{1}{\sqrt{\sqrt{2}}} \approx 0.841$ meters, accordingly with the side ratios.

Each next paper size (A1, A2, ...) is exactly half in area — when computing the size of the next paper in the series, the length of the shorter side of the previous paper becomes the length of the longer side of the next one, and the length of the longer side of the previous paper is divided by 2 to yield the length of the shorter side of the next one. You can imagine this as folding the paper in half along the longer side, like shown in the picture. The sizes should be returned in milimeters:

```
ghci> take 3 paperSizes
[("A0", (1189.207,  840.8964)),
 ("A1", ( 840.8964, 594.6035)),
 ("A2", ( 594.6035, 420.4482))]
```



Hint: There are many different ways to tackle the problem (using either recursion with accumulator variables, or list comprehension, or zip-style list processing). Your solution does not need to be very efficient.

(Use this page as additional space for your answers, if required.)