

# Programming Paradigms

## Written exam

Aalborg University

24 February 2023  
13:00 - 16:00 am

### You must read the following before you begin!!!

**What is this?** This problem set consists of 6 problems. The problem set is part of a zip archive together with a file called `solutions.hs`.

**How do I begin?** *Please do the following immediately!!* Unzip the zip archive and save the problem set and `solutions.hs` on your computer. Please add your full name, AAU mail address and study number to the top of your local copy of `solutions.hs` saved to your computer where indicated in the file. If you experience problems with the file extension `.hs`, then rename the file while working and give it the file extension `.hs`, just before you submit it.

**Må jeg skrive på dansk?** You can write your answers in Danish or in English.

**How and where should I write my solution?** Please write your answers by adding them to the local copy of `solutions.hs` on your computer. You must indicate in comments which problem your text concerns and which subproblem it concerns. All other text that is not runnable Haskell code (such as code that contains syntax errors or type errors) must also be written as comments.

Use the format as exemplified in the snippet shown below.

-- Problem 2.2

bingo = 17

-- The solution is to declare a variable called `bingo` with value 17.

**How should I submit my solution?** Submit the local copy of `solutions.hs` that your solutions appear in and nothing else. DO NOT SUBMIT A ZIP ARCHIVE.

**What can I consult during the exam?** During the exam, you are allowed to use the textbook *Programming in Haskell* by Graham Hutton, your own notes and your installation of the Haskell programming environment.

**What can I use for my code?** You are only allowed to use the Haskell Prelude for your Haskell code, unless the text of a specific problem specifically mentions that you should also use another specific module. Do not use any special GHCi directives.

**Is there anything else I must know?** Yes. Please read the text of each problem **very carefully** before trying to solve it. All the information you will need is in the problem text. Please make sure that you understand what is being asked of you; it is a very good idea to read the text more than once.

## Problem 1 – 18 points

Let  $l = [x_1, \dots, x_n]$  be a list of numbers. The *squared norm* of  $l$  is defined by

$$\text{norm } l = \sum_{i=1}^n (x_i)^2$$

As an example, `norm [1, 3, 5, 6] = 12 + 32 + 52 + 62 = 71`.

Your task is to define a Haskell function `norm` that computes the norm of any given list of numbers.

1. What is the type of `norm`? Is `norm` a polymorphic function? If yes, explain if and why it is ad hoc or parametric polymorphic. If no, explain why it is not polymorphic.
2. Give a definition in Haskell of `norm` that uses recursion.
3. Give a definition in Haskell of `norm` (called `norm'`) that uses `foldr` or `foldl`.

Now consider two lists of numbers,  $l_1 = [x_1, \dots, x_n]$ , and  $l_2 = [y_1, \dots, y_n]$ . We assume that  $l_1$  and  $l_2$  have the same length. The *squared distance*  $\text{dist}(l_1, l_2)$  between the lists is defined by

$$\text{dist } l_1 l_2 = \sum_{i=1}^n (x_i - y_i)^2$$

If  $l_1$  and  $l_2$  do not have the same length, the squared distance is not defined.

4. Give a definition of `dist` in Haskell.

## Problem 2 – 18 points

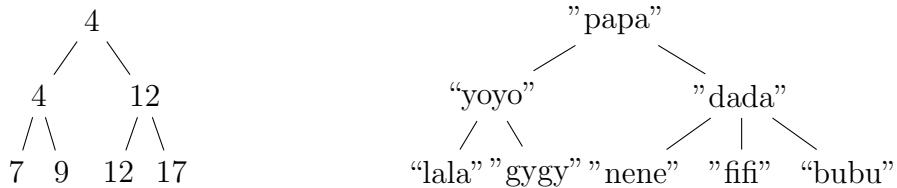
Let  $a$  be an arbitrary type. A  $(2, 3)$ - $a$ -tree is a tree if the following two conditions hold:

- All nodes are labelled with values of type  $a$
- All internal nodes (that is, nodes that are not leaves) have either 2 or 3 children.

A  $(2, 3)$ - $a$ -tree is *uniform* if either

- Every internal node has precisely 2 children (so this is a binary tree), or
- Every internal node has precisely 3 children (so this is then a ternary tree).

Figure 1 shows a  $(2, 3)$ -**Int**-tree that is uniform and a  $(2, 3)$ -**String**-tree that fails to be uniform because the node labelled "dada" has 3 children, where all other internal nodes in the tree have only 2.



(a) A  $(2, 3)$ -**Int**-tree that is uniform      (b) A  $(2, 3)$ -**String**-tree that is not uniform

Figure 1: Two  $(2, 3)$ - $a$ -trees

Your task is now to define a datatype for trees of this kind and a function that can tell us if a given tree is uniform.

1. Define a datatype **TwoThree a** that describes  $(2, 3)$ - $a$ -trees.
2. Represent the trees shown in the Figure as terms of the datatype that you have defined.
3. Define a function **uniform** that will tell us if a  $(2, 3)$ - $a$ -tree is uniform.

## Problem 3 – 16 points

The goal of this problem is to define values, given specifications of them in the form of types.

Below are four types. For each of them, define a Haskell value (which may be a function) that has this particular type as their most general type.

1.  $\text{Ord } a \Rightarrow (a, a) \rightarrow \text{String} \rightarrow \text{Integer}$
2.  $\text{Bool} \rightarrow p \rightarrow p$
3.  $\text{Show } a1 \Rightarrow [a2] \rightarrow a1 \rightarrow \text{IO } ()$
4.  $((a1, a1), b) \rightarrow [a2] \rightarrow ((a1, b) \rightarrow [a3]) \rightarrow [a3]$

Moreover, for each of these four types also indicate if the type involves

- parametric polymorphism only
- overloading (ad hoc-polymorphism) only
- both forms of polymorphism
- no polymorphism

## Problem 4 – 16 points

Here is a function `readNumber` that takes an `s` of type `String` and returns a value of type `Maybe Int`. It will evaluate to `Nothing` if `s` does not represent a valid integer.

```
readNumber :: String -> Maybe Int
readNumber s = case reads s of
    [(n, "")] -> Just n
    _ -> Nothing
```

As an example, we have that `readNumber "484000"` will evaluate to `Just 484000`, and we have that `readNumber "plip"` will evaluate to `Nothing`.

1. Use `do`-notation and the function `readNumber` to define a function `add` that takes two `Maybe Int` values. If the two values represent numbers, the function must return a value of type `Maybe Int`. If one or both arguments is `Nothing`, the result must be `Nothing`

Your solution *must not* use pattern matching or local declarations, only the usual constructs of `do`-notation.

2. Use `do`-notation and the function `add` to create a function `readAndAdd` that takes two arguments of type `String`. If the two arguments represent numbers, the function must return a value of type `Maybe Int`. If one or both arguments is `Nothing`, the result must be `Nothing`

The function should behave as follows.

```
> readAndAdd "5" "3"
Just 8
> readAndAdd "5" "abc"
Nothing
```

Again, your solution *must not* use pattern matching or local declarations, only the usual constructs of `do`-notation.

## Problem 5 – 18 points

The goal of this problem is to define a function `repeatStrings` that takes a list `xs` and a list of integers `ns` and returns a list of elements where each element in `xs` has been repeated the number of times specified by the corresponding integer. If the lengths of `xs` and `ns` are different, the shorter of the two lists defines the outcome.

As examples, we expect that

```
repeatStrings ["plip","bob","bubu"] [3,2,3]
```

will give us

```
["plip","plip","plip","bob","bob","bubu","bubu","bubu"]
```

and that

```
repeatStrings [True,True,False,True] [3,2,3]
```

will give us

```
[True,True,True,True,False,False]
```

and that

```
repeatStrings ["plip","bob"] [3,2,3]
```

will give us

```
["plip","plip","plip","bob","bob"]
```

1. What is the type of `repeatStrings`? Is the function polymorphic? If it is, does it use overloading/ad hoc polymorphism, parametric polymorphism or a mix of them? If it is not, why is this the case?
2. Define a version of `repeatStrings` that uses recursion.
3. Define a version of `repeatStrings` that uses `map`, `foldr` and `zip` but no recursion and no list comprehensions.

## Problem 6 – 16 points

A *triangular number* counts the number of dots arranged in an equilateral triangle. The  $n$ th triangular number is the number of dots in the triangular arrangement with  $n$  dots on each side, and is equal to  $\sum_{k=1}^n k$ , that is, the sum of the  $n$  natural numbers from 1 to  $n$ .

The infinite sequence of triangular numbers, starting with the 0th triangular number, starts as follows.

0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 . . .

1. Give a definition in Haskell of the *infinite* list `triangles` of triangular numbers that uses list comprehension only, but not recursion.
2. Give a definition of the *infinite* list `triangles'` of triangular numbers that uses recursion, but not list comprehension.
3. Give a definition of the *infinite* list `triangles''` of triangular numbers that uses `map` and `foldr` or `foldl` (but no recursion or list comprehension).