A collection of Haskell exercises

Programming Paradigms

Hans

December 2025

1 How to use this document

Section 2 of this document contains 20 programming problems that cover the learning goals of the course *Programming Paradigms*.

- At the exam, you must be able to solve problems of this kind *on your own*. The goal is therefore that you become able to solve problems of this kind on your own and become able to check that your solution makes sense and is correct.
- Solving a problem means that you produce a piece of Haskell code that can actually be executed using GHCi and works as intended.
- Make a plan. Solve at least one problem a day.
- The goal is that you must be able the problem *yourself* under the same conditions that apply at the exam. Solve these problems using only
  - The textbook
  - The compendium
  - GHCi
- You must use GHCi to help develop and check your solution.
- For every problem: Devise a small, suitable collection of test cases and use them. Maybe there are already some test cases in the problem text. Try to find them.

2 The 20 problems

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

   a) `Ord a =>(a, a) -> String -> Integer`
   b) `Bool -> p -> p`
   c) `(Ord a1, Eq a2)=>a2 -> a2 -> (a1, a1)-> a1`
   d) `Show a1 =>[a2] -> a1 -> IO ()`
   e) `((a1, a1), b) -> [a2] -> ((a1, b)-> [a3]) -> [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

   Do not annotate your expressions and values with types. The types must be the ones found by GHCi using type inference.

2. Here is the definition of a Haskell function.

   madras (f,x,y) = f (f x x) y

   Give a curried version of madras that has type (t −> t −> t)−> t −> t −> t,

3. A *palindrome* is a string that is the same written forwards and backwards such as "Otto" or "Madam".

   The goal of this problem is to write a Haskell function ispalindrome that will determine if a string of characters is a palindrome.

   a) First figure out the type of ispalindrome without using the Haskell system. Is the function polymorphic? Why? How?
   b) Now give *two different* definitions of the function, one that uses the reverse function and one that does not.

4. A list $xs = [x_1, \ldots, x_k]$ is a *prefix* of the list ys if we have that $ys = xs$ or $ys = [x_1, \ldots, x_k, y_1, \ldots, y_m]$, that is, if ys consists of at least the elements of xs, possibly followed by more elements. As an example, [3,4,5] is a prefix of [3,4,5,6,484000] . As another example, [] is a prefix of [True,False].

   A list xs is *found within* the list ys if there exist lists z_1 and z_2 (one or both of which may be empty) such that $ys = z_1 ++ xs ++ z_2$. As an example, [3,4,5] is found within [1,2,3,4,5,6,484000] . As another example, [False,False] is found within [True,False,False,True].

   a) Define a function prefix that will tell us if a list is a prefix of another list. Is prefix polymorphic? Why and how?
   b) Use prefix to define a function fwin that will tell us if a list is found within another list. Is fwin polymorphic? Why and how?

5. A list l is increasing wrt. some ordering relation $<$ if whenever x appears earlier in l than y, then $x < y$. The goal is now to define a Haskell function increasing that will take any list as argument and tell us if it is increasing.

   For instance,

   increasing [1,2,7,484000]

   should return True. On the other hand,

   increasing ["ged","abe","hest"]

   should return False.

   a) What should the type of increasing be? Is the function polymorphic? Explain.
   b) Define increasing using recursion.
   c) Define increasing using suitable higher-order functions.

6. Let $l = [x_1, \ldots, x_n]$ 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 $[1, 3, 5, 6] = 1^2 + 3^2 + 5^2 + 6^2 = 71$.

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

   a) 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.
   b) Give a definition in Haskell of norm that uses recursion.
   c) Give a definition in Haskell of norm (called norm') that uses foldr or foldl.

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

   d) Give a definition of dist in Haskell.

7. The function isolate takes a list l and an element x and returns a pair of two new lists (l1 , l2). The first list l1 is a list that contains all elements in l , that are not equal to x. The second list l2 is a list that contains all occurrences of x in l.

   - isolate [4,5,4,6,7,4] 4 evaluates to ([5,6,7],[4,4,4]) .
   - isolate [' g ',' a ',' k ',' a '] 'a' evaluates to ([' g ',' k '], [' a ',' a ']) .

   a) What should the type of isolate be?

   b) Is isolate a polymorphic function? If yes, explain what forms of polymorphism are used. If no, explain why isolate is not polymorphic.

   c) Define isolate in Haskell

      i. using recursion

      ii. using list comprehension

      iii. using foldr

8. A *triangular number* counts the number of dots arranged in an equilateral triangle. The nth 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.

   $$0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 \ldots$$

   a) Give a definition in Haskell of the *infinite* list triangles of triangular numbers that uses list comprehension only, but not recursion.

   b) Give a definition of the *infinite* list triangles ' of triangular numbers that uses recursion, but not list comprehension.

   c) Give a definition of the *infinite* list triangles '' of triangular numbers that uses higher-order functions (but no recursion or list comprehension).

9. The set of perfect cubes is the set of natural numbers that are of the form n^3 for some $n \in \mathbb{N}$, i.e. the infinite set

   $$\{1, 8, 27, 54, \ldots\}$$

   a) Using Haskell, define the infinite list cubes whose elements are the perfect cubes. What is the type of cubes?

   b) Given a natural number n, the *integral cube root* of n is the greatest natural number i such that $i^3 \leq n$. As an example, the integral cube root of 9 is 2, since $2^3 = 8$ but $3^3 = 27$.

      Use list comprehension in Haskell to define the function icr that for any natural number will compute its integral cube root. What is the type of icr?

   c) Use foldr to define the function sumcubes that computers the sum of the first n cubes for any given n. For instance, we should have that sumcubes 3 returns 36.

10. Here is a small piece of Haskell code for defining nested pairs.

    ```
    data Nesting a b = S (a,b) | C (Nesting a b,Nesting a b)
    ```

    a) What is the correct terminology for the Haskell concept of which Nesting is an example?

    b) What is the correct terminology for the Haskell concept of which S is an example?

    c) Here is a Haskell expression.

       ```
       C (C( S(True , True) ,C(S(True , True) ,S(True , False))) ,S(True , False))
       ```

       We say that its *nesting depth* is 3, since every subexpression is found within no more than 3 nested C's.

       Define a depth function that computes the nesting depth of any expression that has type Nesting a b for any a and b and give the type of the depth function.

11. The compression of a list is a list that counts successive elements that are repeated and returns a list of pairs of the form (x, v) where (x, v) indicates that there are v successive elements that are x's. For instance, the compression of

    ```
    [ 1 ,1 ,1 ,2 ,1 ,4 ,4 ,4 ,1 ,1 ,6 ,1 ,6 ,4 ,4 ,4 ,4 ,4]
    ```

    is the list

```
[(1,3),(2,1),(1,1),(4,3),(1,2),(6,1),(1,1),(6,1),(4,5)]
```

and the compression of

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

is the list

```
[(True,3),(False,1),(True,1),(False,2)]
```

The compression of an empty list is the empty list itself.

a) Using Haskell, define a function compress that computes the compression of a list. What is the type of compress? Is the function polymorphic? Justify your answer.

b) If we are given a list of pairs where each pair is of the form (x, v) where v is a natural number, we can decompress the list such that for every element (x, v) we get v successive copies of each x. The decompression of [[(1,3) ,(2,1) ,(1,1) ,(4,3) ,(1,2) ,(6,1) ,(1,1) ,(6,1) ,(4,5)] is therefore [1,1,1,2,1,4,4,4,1,1,6,1,6,4,4,4,4,4] Using Haskell, define a function decompress that computes the decompression of a list. What is the type of decompress? Is the function polymorphic? Justify your answer.

12. Cows say Moo and Roar. Sheep say Baa. Below is a conversation between a cow and a sheep expressed as a list.

```
[Moo,Roar,Baa,Moo,Moo,Baa,Roar]
```

a) Define a datatype Utterances that expresses the sounds that cows and sheep can make.

b) Use your datatype Utterances to define a function apart using *recursion* such that apart xs takes a conversation between a cow and a sheep and splits it into a pair (c, s) where c is a list with the sounds made by the cow in the correct order and s is a list with the sounds made by the sheep in the correct order.

As example, we should have that

apart [Moo,Roar,Baa,Moo,Moo,Baa,Roar]

gives us ([Moo,Roar,Moo,Moo,Roar],[Baa,Baa]).

What is the type of apart? Is the function polymorphic?

c) Now your datatype Utterances to define a function apart using foldr.

13. A function maps different arguments to different values. In Haskell, we can represent any function that is only defined for finitely many cases as an association list.

We can define a Haskell function isfun that will take any association list and tell us if it represents a function.

For instance, isfun [(1,'a') ,(2,'b')] should return True whereas isfun [(1,'a') ,(1,'b')] should return False.

a) What should the type of isfun be?

b) Now define isfun in Haskell.

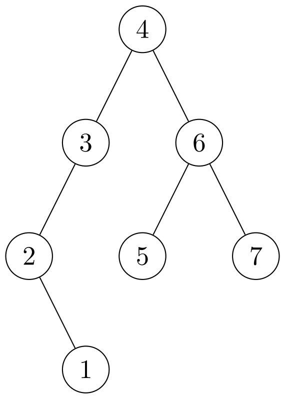
c) We know that a function f is 1-1 if whenever we have x, y with $x \neq y$ we have that $f(x) \neq f(y)$. We can define a Haskell function is11 that will take any association list and tell us if it represents a function which is also 1-1.

For instance, is11 [(1,'a') ,(2,'b')] should return True whereas is11 [(1,'a') ,(2,'a')] should return False.

What should the type of is11 be?

d) Now define is11 in Haskell. It is a good idea to make use of the solutions to the previous subproblems.

14. Here is a binary tree.



a) Define a datatype Btree a for binary trees that have nodes whose labels are taken from the type a.

b) The *leftmost path* in a binary tree is the path that can be obtained by starting at the root and always following the left child of any node, if one exists. If at some point during this traversal there is no left child, the path ends. In the tree drawn above, the leftmost path is the sequence $4, 3, 2$. Define a function leftpath such that leftpath t gives us the list that represents the leftmost path in the tree t.

   Is leftpath polymorphic? If yes, how is it polymorphic? If no, why not?

15. A Unix directory d contains other directories as well as files. Every directory has a name and a finite list of arbitrarily many directories, which are the subdirectories of d (d may have no subdirectories at all). A directory can contain zero or more files. Every file has a name, which is a string, and a size, which is a whole number.

    a) Define an algebraic datatype Directory that describes this.
    b) In your datatype definition, give an example of a value of type Directory
    c) Define a totalsize function that computes the sum of the sizes of files found in a directory and in its subdirectories.

16. We can represent arithmetic expressions S over atoms (that are strings), addition, multiplication and bracketing by the formation rules

$$S ::= a \mid S + S \mid S * S \mid (S)$$

    a) Define a Term that describes arithmetic expressions. Give a term of type Terms that corresponds to the arithmetic expression

$$\texttt{two} * (\texttt{six} + \texttt{four})$$

    b) An expression S is *plain* if all atoms in it are the same atom.
       Define a function plain that can tell us if an arithmetic expression is plain. What should the type of plain be? Is plain a polymorphic function? Justify your answer.
    c) Define a function pretty that can convert any value of the type Term to an arithmetic expression in the form of a string. What should the type of pretty be? Is pretty a polymorphic function? Justify your answer.

17. 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
                                   x  -> Nothing
```

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

    a) 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.

b) 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.

18. The goal of this problem is to define a functor instance for a type and prove a property of the definition that you make.

    Here is a type declaration.

    ```
    newtype Funpair a = Fun (Bool -> a, String -> a)
    ```

    a) Give an example of an expression in Haskell that has type Funpair Integer.

    b) Show how to define Funpair to be a functor.

    c) Show that the functor law

    $$\text{fmap id} = \text{id}$$

    holds for your definition.

19. Here is the declaration of a type WrapString and a declaration that makes it an instance of Functor.

    ```
    newtype WrapString a = WS (a,String) deriving Show

    instance Functor WrapString where
        fmap f (WS (x,s)) = WS (f x,s)
    ```

    a) Extend the above piece of code with an instance declaration such that WrapString becomes an applicative functor also.

    b) Extend the above piece of code with an instance declaration such that WrapString becomes a monad also.

    c) Use a do-block in the WS-monad that you now have to define a function pairup such that we have that pairup (WS (4,"horse"))(WS (5,"plonk")) gives us WS ((4,5),"horse").

20. The goal of this problem is to complete the definition of a monad and define two functions that involve this monad.

    Below is a piece of Haskell code.

    ```
    data W x = Bingo x deriving Show

    instance Functor W where
        fmap f (Bingo x) = Bingo (f x)

    instance Monad W where
        return x = Bingo x
        Bingo x >>= f = f x
    ```

    a) For this to make sense, a definition of W as an applicative functor is missing. Write such a definition.

    b) Use do-notation to define a function wrapadd :: Int -> W Int -> W Int which satisfies that

    $$\text{wrapadd x (Bingo y)} = \text{Bingo (x+y)}$$

    c) Use do-notation to define a function h :: W Int -> W Int -> W Int which satisfies that

    $$\text{h (Bingo x) (Bingo y)} = \text{Bingo (x*y)}$$