Programming Paradigms
Compendium

Hans Hüttel

Autumn 2025

1. Find a Haskell expression whose type is `(Ord a1, Eq a2) => a2 -> a2 -> (a1, a1) -> a1`

   Comment The type is a function type. The function must be curried and takes three arguments, the third of which is a pair. There are type class constraints for a1 and a2. In this case, comparisons regarding $<$ (due to Ord) and $=$ (due to Eq) can be placed in a condition.

   ```
   plop' x y (u,v) = if (x == y) && (u > v) then u else u
   ```

2. Use list abstraction to define a function flop that, when given a list of pairs, returns a list of pairs whose components are swapped. The list can be empty. For example, `flop [(1,'a'),(3,'r'),(9,'e')]` should return the list `[('a',1),('r',3),('e',9)]`.

   Comment Here we have

   ```
   flop xs = [(y,x) | (x,y) <- xs ]
   ```

3. A list $[a_1, a_2, \ldots, a_n]$ is *decreasing* if $a_1 \geq a_2 \geq \ldots \geq a_n$. Write a function descending that returns True if the list it receives as an argument is decreasing, and False otherwise. For example, `descending [6,5,5,1]` should return True, and `descending ["plip","pli","ppp"]` should return False.

   Comment The solution is

   ```
   descending (x:y:ys) | x >= y = descending (y:ys)
                       | x < y = False
   descending xs = True
   ```

   Note which patterns are needed.

4. Implement the function sumrows. The function takes a list of lists of numbers and returns a one-dimensional list of numbers, where each number is equal to the sum of the corresponding row in the input list. If a list is empty, its sum is 0. For example, `sumrows [[1,2], [3,4]]` should give us `[3, 7]`, and `sumrows [[],[],[1]]` should give us `[0,0,1]`.

   Comment This is a classic use of map together with sum.

   ```
   sumrows xs = map sum xs
   ```

5. Define a Haskell datatype Aexp for arithmetic expressions with addition, multiplication, numbers, and variables. The construction rules in the abstract syntax are

   $$E ::= n \mid x \mid E_1 + E_2 \mid E_1 \cdot E_2$$

   Assume that variables x are strings, and that numbers n are integers.

Comment The construction rules can be translated directly. Each case in the construction rules should have its own term constructor.

```
data Aexp = N Int | V String | Plus Aexp Aexp | Mult Aexp Aexp
```

6. Define a function thesame that takes a list of pairs xs and returns a list of pairs where the first and second components are equal. For example,

   ```
   thesame [(1,2),(4,4),(6,7),(17,17)]
   ```

   should return

   ```
   [(4,4),(17,17)]
   ```

   What should the type of thesame be?

   Comment The type class Eq (book, page 31) is the family of types that have an implementation of equality. And from the description of thesame, we can see that the components of a pair must be able to be checked for equality. So

   ```
   thesame :: Eq a => [(a,a)] -> [(a,a)]
   ```

7. Use recursion to define a Haskell value letters, which is a sequence of actions that does the following:

   - Receives a string
   - Prints the characters in the string one at a time, each followed by a newline

   What we want is, for example:

   ```
   *Main> letters
   dingo
   d
   i
   n
   g
   o
   *Main>
   ```

   Comment Here it is:

   ```
   letters = do
           x <- getLine
           each x
           return ()
         where
           each [] = do
                       return ()
           each (x:xs) = do
                            putChar x
                            putChar '\n'
                            each xs
   ```

8. Here is a type declaration for simple expressions.

   ```
   data Exp a = Var a | Val Integer | Add (Exp a) (Exp a) | Mult (Exp a) (Exp a) deriving
     Show
   ```

   Show how to make this type an instance of Functor. When would it be useful to consider Exp a as a functor? Think of a good example!

Comment Here we need to define fmap, and it is a good idea to first see what type this function should have. It should be

```
fmap :: (a -> b) -> Exp a -> Exp b
```

So, from an expression where variables have type a, we must construct an expression where variables have type b. The solution is therefore

```
instance Functor Exp where
  fmap f (Var x) = Var (f x)
  fmap f (Val n) = Val n
  fmap f (Add e1 e2) = Add (fmap f e1) (fmap f e2)
  fmap f (Mult e1 e2) = Mult (fmap f e1) (fmap f e2)
```

A good example of when it is useful to think of this type as a functor is *renaming variables*. Renaming variables consists of replacing variable occurrences with other variable occurrences. A more general example is *substitution* (of which renaming variables is a special case).

9. Here is a function readNumber that takes a s of type String and returns a value of type Maybe Int. It evaluates 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
```

For example, readNumber "484000" evaluates to Just 484000, and readNumber "plip" evaluates to Nothing. Use do-notation to create a function readAndAdd that takes two arguments of type String . If the two arguments represent numbers, the function should return a value of type Maybe Int. If one or both arguments are Nothing, the result should be Nothing. The function should behave as follows.

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

Your solution must *not* use pattern matching or local declarations, only the usual monad constructs.

Comment Here is a solution.

```
readAndAdd :: String -> String -> Maybe Int
readAndAdd s1 s2 = do
    x <- readNumber s1
    y <- readNumber s2
    return (x+y)
```

10. Give a definition of the list naturals of natural numbers, using *recursion*. The smallest natural number is 1.

Comment We have

```
naturals = from 1
         where
            from n = n : (from (n+1))
```