DM580 – EXERCISE SHEET #2

(1) Use library functions to write a function `lastelem :: [a] -> a` that returns the last element of a list.

(2) Consider the mathematical function f defined by cases below.

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 2x - 1 & \text{otherwise} \end{cases}$$

Write a Haskell function `haskf :: Int -> Int` implementing this function.

(3) Is `Int -> Int` the most general type for `haskf`? If not, what is? Justify your answer, whatever it is.

(4) Write a function `empty :: [a] -> Bool` that decides whether the given list is empty or not.

(5) Without using any library functions or operators, show how the following function can be defined using only conditional expressions:

```
conj True True = True
conj _    _    = False
```

Hint: Use two nested conditionals. If you're clever, you can even get away with just one.

(6) Consider the function `safetail :: [a] -> [a]` that behaves in the same way as `tail :: [a] -> a`, except that it maps the empty list to itself. Using your function `empty` from a previous exercise, write `safetail` in three different ways using
   (a) a conditional expression,
   (b) pattern matching, and
   (c) guards.

(7) Consider the function `safediv :: (Integral a) => (a,a) -> [a]` that takes two numbers and, if the second number is nonzero, returns the one-element list consisting of the first number divided by the second. If the second number is zero, `safediv` returns the empty list. Write `safediv` in three different ways using
   (a) a conditional expression,
   (b) pattern matching, and
   (c) guards.

(8) Consider the function `implies :: Bool -> Bool -> Bool` that returns `True` whenever the first argument is `False` *or* the second argument is `True`, and `False` otherwise. Write `implies` in three different ways using
   (a) a conditional expression,
   (b) pattern matching, and
   (c) guards.

(9) Consider the function

```
mul3 :: Int -> Int -> Int -> Int
mul3 x y z = x*y*z
```

Rewrite `mul3` to be defined purely in terms of lambda expressions.

(10) Write a function `largestHead :: (Ord a) => ([a], [a]) -> [a]` that takes a pair of lists and returns the one that has the largest head (i.e., first element). If one of the lists is empty, `largestHead` should return the other one (even if that is also be empty).

(11) *(Advanced)* Write a function `half :: (Fractional a) => [a] -> [a]` that divides each number in a given list by 2. *Hint:* Use a higher-order function.

(12) *(Advanced)* The function `sum :: (Num a) => [a] -> a` that takes a list of integers and returns their sum can be defined as `sum = foldl (+) 0`. However, this is not the only way to define this function! Come up with a definition of `sum` that does not use library functions. *Hint:* Use pattern matching and recursion.