Lund University
Department of Computer Science

EDAF95 (aka EDAF40)
2nd June 2022, 14–19

Basics of Functional Programming

1. **Simple programming** (1p)

   Write a function

   ```
   permutations :: [a] -> [[a]]
   ```

   that given an arbitrary list with non-repeating elements would produce all the permutations of this list. (A *permutation* of a list is a list containing exactly the same elements, but possibly in different order.)

   Examples:

   ```
   Prelude> permutations []
   []
   Prelude> permutations [1]
   [[1]]
   Prelude> permutations [1,2]
   [[1,2],[2,1]]
   Prelude> permutations [1,2,3]
   [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
   ```

   Note that the order of individual permutations in the above examples is not important, just that they all are to be found somewhere in the answer.

   You may use the assumption that elements in the input list do not repeat. For the repeating case the outcome may be arbitrary.

   Actually, a solution neglecting this fact may be simpler.

2. **Type derivation** (1p)

   (a) (0.4p) Which type has the function `g` defined as

   ```
   g xs = [f x | x <- xs, x > 3]
   ```

   where `f n = replicate n '+'` ?

   (b) (0.3p) Find the type of `curry curry`

   (c) (0.3p) Find the type of `curry . curry`

3. **List comprehension** (1p)

   Write, using list comprehension syntax, a single function definition (try to avoid `if`, `case` and similar constructs) with signature

   `g :: [[Int]] -> [[Int]]`,

   which, from a list of lists of `Int`, returns a list of the tails of those lists using, as filtering condition, that the head of each `[Int]` must be odd. Also, your function must not trigger an error when it meets an empty element, but rather silently skip such an entry. Example:

```
Prelude> g [[1,2],[],[6,2,3],[3],[6,5,4,3],[6],[5,1,1]]
[[2],[],[1,1]]
```

Rewrite now this definition using `map` and `filter` instead of list comprehension.

4. **Binding** (1p)

   What is the type and value of the following expression:

   ```
   "Ukraine" >>= (\u -> flip (:) [] $ id u)
   ```

   Show how you derived the value.

5. **More programming** (2p)

   A relation R is an arbitrary subset of the cartesian product of two arbitrary sets, called *domain* (denoted by A), and *range* (denoted by B), respectively.

   $$R \subset A \times B$$

   For arbitrary A and B there exist, among other, the *total* relation ($R = A \times B$), the *empty* relation ($R = \emptyset$), and all possibilities in-between. So, for every relation R on $A \times B$ the following is true:

   $$(a, b) \in R \rightarrow a \in A \ \ \& \ \ b \in B$$

   Your task is to define the datatype Relation that would take two parameters (corresponding to the types of sets A and B):

   ```
   data Relation a b = ...
   ```

   and then define the following two functions corresponding to the usual operations on sets:

   ```
   union        :: Relation a b -> Relation a b -> Relation a b
   intersection :: Relation a b -> Relation a b -> Relation a b
   ```

   and the following two specific for relations:

   ```
   composition :: Relation b c -> Relation a b -> Relation a c
   closure :: Relation a a -> Relation a a
   ```

   Relation composition works the same way as function composition: if $(a, b) \in R_1$ and $(b, c) \in R_2$ then $(a, c) \in R_2.R_1$. The (transitive) closure finds all pairs that can be created by composing a relation with itself arbitrary many times.

   You may assume that all the relations are finite, if it helps in solving the problem.

Good Luck!