Exercise Set 1 — 27 August

Fragments

1.1 Typing

Write a type for each of the following expressions:

- `(True,True:[]) ::`
- `\x -> \y -> (x && y, y || x) ::`
- `"a"++"b" ::`
- `map tail ["a","ab","abc"] ::`
- `zip ::`
- `zip ["a","ab","abc"] ["bcd","cd","d"] ::`
- `map zip ::`
- `map (uncurry (++)) (zip ["a","ab","abc"] ["bcd","cd","d"]) ::`

1.2 Lists and functions

Which of the following statements are true (for all finite lists `xs` and all functions `f` and `g` of suitable type)?

- `reverse (map f xs)` = `map f (reverse xs)`
- `map f (map g xs)` = `map g (map f xs)`
- `reverse (reverse xs)` = `reverse xs`
- `map f (map f xs)` = `map f xs`
- `reverse xs` = `xs`

Small programs

1.3 Simple functions on numbers

Write a Haskell function `test :: Int -> Int -> Bool` that takes two integers and returns `True` if and only if the two integers are both odd.

1.4 List manipulation

Write a Haskell function `duplicate :: [Char] -> [Char]` that takes a list of elements and returns a list where every element has been duplicated. For example,
`duplicate "Hello World"` evaluates to `"HHeellllloo  WWoorrlldd"`.

1.5 List manipulation

Write a Haskell function `compress :: [Char] -> [Char]` that eliminates consecutive duplicate elements of a list. For example, `compress "HHeellllloo  WWoorrlldd"` evaluates to `"Helo World"`.

1.6 List manipulation

Write a Haskell function `zipSum :: [Int] -> [Int] -> [Int]` that takes two equally sized lists of ints and returns a single list of ints in which each element at a given index is the sum of the corresponding values at that index from the input lists. For example, `zipSum [1,2,3] [4,5,6]` evaluates to `[5,7,9]`. Then, provide an alternative implementation in terms of the function `zipWith`.

1.7 Using lists for sets: writing recursive functions over lists

Let us use the Haskell type `[Integer]` to represent sets of integers. The representation invariants are that there are no duplicates in the list, and that the order of the list elements is increasing.

Do not use any of the built-in Haskell functions that manipulate lists as sets. Do not use any Haskell libraries for sets.

1. Write a Haskell function `setUnion: [Integer] -> [Integer] -> [Integer]` that takes two sets and returns their union.
2. Write a Haskell function `setIntersection: [Integer] -> [Integer] -> [Integer]` that takes two sets and returns their intersection.
3. Write a Haskell function `setDifference: [Integer] -> [Integer] -> [Integer]` that takes two sets and returns their set difference.
4. Write a Haskell function `setEqual: [Integer] -> [Integer] -> Bool` that takes two sets and returns `True` if and only if the two sets are equal.
5. Write a Haskell function `powerSet: [Integer] -> [[Integer]]` that takes a set S and returns its powerset 2^S. (The powerset 2^S of a set S (sometimes written $P(S)$) is the set of all subsets of S.) Note that the result uses the type `[[Integer]]` to represent sets of sets of integers. Here the representation invariant is that there are no duplicates in the list; the order of the sublists is immaterial.

1.8 Numbers

1. Write a function `mp :: Int -> [Int]` such that `mp n` returns a list consisting of the first `n` Mersenne primes. Test this function for `n` up to 8.

2. What do you need to do in order to compute a few more Mersenne primes, for instance up to 11?