Homework 4

You have to submit your solutions as announced in the lecture.
Unless mentioned otherwise, all problems are due 2017-03-09, 11:00.
There will be no deadline extensions unless mentioned otherwise in the lecture.

Problem 4.1 *Correctness*

Points: 2+2+2

Consider the following algorithm for reverting a list:

```
fun revertFun[A](x : List[A]) : List[A] =
   rest := x
   rev := []
   while rest ≠ []
      rev := cons(rest.head, rev)
      rest := rest.tail
   return rev[1]
```

Its specification is

- precondition: nothing
- postcondition: $revertFun(x) == revert(x)$

Prove correctness by doing the following:

1. Give a loop invariant $F(x, rest, rev)$ for the while-loop.
2. Argue informally why it implies partial correctness.
3. Give a termination ordering $T(x, rest, rev)$ for the while-loop.

Solution:

1. $x == revert(rev) + rest$
2. When terminating, $rest == []$. Plugging that into the loop invariant yields $x == reverse(rev)$ and therefore $reverse(x) = rev$. Because rev is the value that is returned, we have $rev == revertFun(x)$. Combining the two yields the postcondition.
3. $length(rest)$

Problem 4.2 *Lists*

Points: 6+6

Implement a data structure for polymorphic lists twice: once using an inductive data type in a functional language, and once using classes/structs or comparable primitives for (mutable or immutable) singly-linked lists in an imperative or object-oriented language.

Specifically, the implementation should have

- the data type definition itself (2 points)
- polymorphic functions for *concat* and *reverse* (2 points each)

Remarks:

- Most programming languages have libraries for lists. Naturally you are not allowed to use those libraries.
- If you use a multi-paradigm programming language that supports both data type and classes, both implementations can use the same programming language. Otherwise, use two different languages.
- Because you have to be polymorphic, you have to use a typed programming language that supports polymorphism. That can be tricky, and the lecture notes contain some examples (Sect. 2.4) to get you started. If you are completely stuck, drop the polymorphism and implement it only for lists of integers. That costs half the points.

[1] This line was accidentally missing in the original formulation. But the intention of the algorithm was clear from the context.

Problem 4.3 *Sorted List* Points: 2+2+(2+2+2)

In any programming language and possibly extending one of your implementations from above, implement the following:

1. An abstract data structure $Ord[A]$ for orders. You may reuse/adapt the examples given in Sect. 2.4.
2. A function $sorted[A](x : List[A], ord : Ord[A]) : bool$ that checks if a list is sorted.[2]
3. 3 concrete instances of $Ord[A]$:
   - $IntSmaller : Ord[int]$ for the order $\leq$
   - $Divisible : Ord[int]$ for the order $|$
   - $Lexicographic : Ord[string]$ for the order in which words are listed in a dictionary

[2]Checking whether a list is sorted works for any order. A *total* order would be needed to ensure any list *can* be sorted.