Functional Programming
Exercises Week 13
(for January 10, 2014)

Numbers in parentheses refer to the 6th edition of the course notes.
Exercises marked with ⋆ are optional and can be ignored.

1. Read Chapter 11 of the lecture notes.

2. (Exercise 11.5) Write a function `qsort`, which implements *quicksort*.

Hint: For a non-empty list select the head element as pivot.

3. (Exercise 11.7) Consider the following implementation of *insert* sort.

```
let rec insert x = function
 | [] -> [x]
 | y::ys -> if x < y then x::y::ys else y::(insert x ys)

let rec isort = function
 | [] -> []
 | x::xs -> insert x (isort xs)
```

a) Compute the recurrence equation for `isort`.

b) Solve the recurrence by unfolding it into a tree. Conclude an upper bound for the runtime of `isort`.

c) Does the Master Theorem apply to the recurrence from item 3a?

4. (Exercise 11.11) Consider beans & bowls.

a) Give an upper bound on the number of recursive calls emerging from `beans` m n (in terms of m and n).

b) Draw the recursive calls emerging from `beans 3 3` as a tree. The nodes are labeled `beans` m n for different values of m and n and there is an edge from node `beans` m n to node `beans` m' n' if the former recursively calls the latter.

Hint: Share identical nodes in the tree.

c) Use the tree from item 4b to compute the number of (recursive) calls to `beans` (starting from `beans 3 3`). Check your computation by adding a counter to `beans`.

d) Use the tree from 4b to compute the number of (recursive) calls to `beans` (starting from `beans_dp 3 3`). Check your computation by adding a counter to `beans` (inside `beans_dp`).

5. (Exercise 11.14) Consider Post's Correspondence Problem (PCP). Here PCPs are given as a list of pairs where each pair contains the corresponding words, i.e., the words at the same indices.

The following implementation tries to determine if a PCP has a solution or not by testing all possibilities in a breadth first search. To save memory, common prefixes of two words are removed using the function `trim`.

```
let rec trim = function
 | (x::xs,y::ys) when x = y -> trim (xs,ys)
 | d -> d
```

```
let extend (w1,w2) (d1,d2) = trim (w1@d1,w2@d2)

let solve ds =
 let rec solve = function
  | [] -> false
  | ([],[])::_ -> true
  | (x::_,y::_)::ws when x <> y -> solve ws
  | w::ws -> solve (ws@Lst.map (extend w) ds)
 in solve (Lst.map trim ds)
```

a) Write a function `solve_dp`, which uses dynamic programming to avoid considering the same problems again.

b) Can you give an upper bound on the additional memory needed for the lookup table?

c) Write a function `solve_dps`, which returns (the indices of) a solution.

⋆. (PCP solver competition)

In the lecture on Friday, 2014, Jan 10 there will be a small PCP solver competition.

Setup: The lecturer collects (a) PCP solvers and (b) PCP problems (see below). In the competition each solver runs on each problem for 60 seconds. The tool which solves most problems positively, i.e., outputs a sequence of indices which is a solution wins.[1] **You may work in teams.**

Rules: Your program may not fork subprocesses or call external tools.

To participate in the competition:

a) Integrate your function from Exercise 5 in `PCPSolver.ml` to get a stand-alone executable that accepts dictionaries as command-line arguments (or on `STD_IN` when using the option `-`). Your PCP solver must output the indices of the found solution. Take care that the first line of the output still is `true` or `false`.

b) You may (and should) include up to 5 PCP problems in your submission (see below). Please put each PCP problem in a separate file (i.e., `1.pcp`, ..., `5.pcp`, etc. Test your files such that `./PCPSolver.native - < 1.pcp` works). It is a good idea to craft hard problems which your tool can solve.

c) Submit the archive via email (subject: "PCP competition") to

`Harald (dot) Zankl (at) uibk (dot) ac (dot) at`

before

Wednesday, 2014, Jan 8, 23:59 CET

A dummy submission is available from http://cl-informatik.uibk.ac.at/teaching/ws13/fp/material/PCP_dummy.tar.gz You may submit your archive only once!

[1] For the case that a tool is (provably) buggy it will be disqualified.