Written Examination, May 27th, 2019 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: All written material

The problem set consists of 4 problems which are weighted approximately as follows:
Problem 1: 20%, Problem 2: 30%, Problem 3: 25%, Problem 4: 25%

Marking: 7 step scale.

Do not use imperative features, like assignments, arrays and so on, in your solutions.

You are allowed to use the .NET library including the modules described in the textbook, e.g., List, Set, Map, Seq etc.

You are allowed to use functions from the textbook. If you use such a function, then provide a reference to the place where it appears in the textbook.

Problem 1 (20%)

Consider the following F# declaration:

```
let rec f g xs = match xs with
                 | x::y::xs'-> x::(g y)::f g xs'
                 | _        -> xs;;
```

1. Give a step-by-step evaluation (using $\rightsquigarrow$) for `f (fun x -> x+1) [0; 1; 2; 3; 4; 5]` determining the value of the expression. There should at least be one step for every recursive call of `f`.
2. Give the type for `f` and describe what `f` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f` is *not* tail recursive. Provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
4. Provide a declaration of a continuation-based, tail-recursive variant of `f`. Your tail-recursive declaration must be based on an explicit recursion.

 ... Continued on next page

Problem 2 (30%)

A *balance bike* (Danish: "løbecykel") is a bike used by small children, where they can learn how to ride without having to worry about pedals and brakes. In its barest form, a balance bike is constructed from a frame, two wheels, a saddle, handlebars, nuts and bolts, which constitute the *parts* of the bike. The manufacturing of a balance bike may be divided into a sequence of three *tasks*. For example, first mount two wheels on the frame, then mount the saddle, and finally, mount handlebars.

An *assembly line* is a manufacturing process where parts successively are added to an assembly at *workstations* until the final product is obtained. A *part register* (type `PartReg`) associates costs with parts, and a *task register* (type `TaskReg`) associates a pair (d, c) with a task tsk, where d is the time needed to perform tsk and c is the associated cost. We also call d the *duration* of the task:

```
type Part     = string
type Task     = string
type Cost     = int                (* can be assumed to be positive *)
type Duration = int                (* can be assumed to be positive *)
type PartReg  = Map<Part, Cost>
type TaskReg  = Map<Task, Duration*Cost>

(* Part and task registers for balance bikes *)
let preg1 = Map.ofList [("wheel",50); ("saddle",10); ("handlebars",75);
                        ("frame",100); ("screw bolt",5); ("nut",3)];;
let treg1 = Map.ofList [("addWheels",(10,2)); ("addSaddle",(5,2));
                        ("addHandlebars",(6,1))]
```

We observe, from the two example registers, that the cost of a wheel is 50 (say Danish kr.) and mounting a saddle (task `"addSaddle"`) takes 5 time units and costs 2 kr.

A workstation is described by a task (like `"addSaddle"`) and a part list, describing the number of the various parts that are needed to perform the task. Furthermore, an assembly line is a list of workstations:

```
type WorkStation = Task * (Part*int) list
type AssemblyLine = WorkStation list

let ws1 = ("addWheels",[("wheel",2);("frame",1);("screw bolt",2);("nut",2)])
let ws2 = ("addSaddle",[("saddle",1);("screw bolt",1);("nut",1)])
let ws3 = ("addHandlebars",[("handlebars",1);("screw bolt",1);("nut",1)])
let al1 = [ws1; ws2; ws3];;
```

We see that the assembly line for balanced bikes consists of 3 workstations, where, for example, the work station for mounting the saddle requires one piece of each of the parts: saddle, screw bolt and nut.

 ... Continued on next page

A *workstation* $(tsk, [(p_1, k_1); \ldots; (p_n; k_n)])$ *is well-defined* for given part register *preg* and task register *treg*, if (1) there is an entry for *tsk* in *treg*, (2) there is an entry in *preg* for every p_i, where $1 \leq i \leq n$, and (3) the numbers $k_1, \ldots k_n$ are all positive.

Furthermore, an *assembly line is well-defined* for given part register *preg* and task register *treg* if every workstation in the assembly line is well-defined.

1. Declare a function `wellDefWS: PartReg -> TaskReg -> WorkStation -> bool` that checks the well-definedness of a workstation for given part and task registers.

2. Declare a function `wellDefAL: PartReg -> TaskReg -> AssemblyLine -> bool` that checks the well-definedness of an assembly line for given part and task registers. This function should be declared using `List.forall`.

In your answers to the following questions, you can assume that workstations and assembly lines are well-defined.

3. Declare a function `longestDuration`$(al, treg)$, where *al* is an assembly line and *treg* a task register. The value of `longestDuration`$(al, treg)$ is the longest duration of a task in *al*. What is the type of `longestDuration`?

   For example, the longest duration of a task in the assembly line for balanced bikes is 10 (the duration of `"addWheels"`).

4. Declare a function `partCostAL: PartReg -> AssemblyLine -> Cost`, that computes the accumulated cost of all parts needed for one final product of an assembly line for a given part register. For example, the accumulated cost of all parts of a balanced bike is 317 - the cost of one frame, two wheels, one saddle, handlebars, 4 nuts and 4 screw bolts.

   Hint: You may introduce helper functions to deal with workstations and part lists $[(p_1, k_1); \ldots; (p_n; k_n)]$.

5. Declare a function `prodDurCost: TaskReg -> AssemblyLine -> Duration*Cost`, that for a given assembly line and task register, computes a pair $(totalDuration, totalCost)$, where *totalDuration* is the accumulated duration of all durations of tasks in the assembly line and *totalCost* is the accumulated cost of the costs of all tasks in the assembly line (where the cost of parts is ignored). For the balanced bike example, the accumulated duration of the three tasks is 21 and the accumulated cost is 5.

A *stock* is mapping from parts to number of pieces:

```
type Stock = Map<Part, int>
```

6. Declare a function `toStock: AssemblyLine -> Stock`, that for a given assembly line, computes the stock needed to produce a single product.

Problem 3 (25%)

Consider the following F# declarations:

```
let rec h1 g n = if g n = 0
                 then n else h1 g (n+1)

let f1 g = h1 g 0;;
```

1. What is the value of the expression `f1 (fun n -> 16-n*n)`?
2. Give the types for `h1` and `f1` and describe what `f1` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. Provide an expression e so that the evaluation of `f1` e does not terminate.

Consider now the following F# declarations:

```
let s0 = Seq.initInfinite (fun i -> 2*i);;
let s1 = Seq.initInfinite (fun i -> 2*i + 1);;

let rec f2 sx sy = seq { yield (Seq.nth 0 sx, Seq.nth 0 sy)
                         yield! f2 (Seq.skip 1 sx) (Seq.skip 1 sy) }

let h2 sx sy = seq { for (x,y) in f2 sx sy  do
                           yield x
                           yield y};;
```

4. Give descriptions of the values of `s0` and `s1`.
5. What is the value of the expression: `Seq.toList (Seq.take 6 (h2 s0 s1))`?
6. Give the types for `f2` and `h2` and describe what these two functions compute. Your descriptions should focus on *what* they compute, rather than on individual computation steps.

Notice that `Seq.nth 0` s returns the head of a non-empty sequence s and `Seq.skip 1` s returns the tail of a non-empty sequence s.

Problem 4 (25%)

In this problem we shall consider the following extension of the type for search trees from Section 6.4 of the textbook:

```
type T<'a,'b when 'a:comparison> =
             | Leaf
             | Node of  T<'a,'b> * 'a * 'b * T<'a,'b>
```

For a node `Node`$(t_{left}, k, v, t_{right})$, we call k the *key*, v the corresponding *value*, and the pair (k, v) the *entry*.

A value of type `T<'a,'b>` is called a *search tree* if it satisfies the following condition: Every node `Node`$(t_{left}, k, v, t_{right})$ satisfies

- $k' < k$ for every key k' occurring in the left subtree t_{left} and
- $k'' > k$ for every key k'' occurring in the right subtree t_{right}.

This condition is called the *search tree invariant*.

Notice that the search tree invariant implies that keys are unique in a search tree, and hence search trees represent maps.

1. Give an F# value for search tree containing the following five entries: `("a", [1;2;3])`, `("b", [4;5])`, `("c", [6])`, `("d", [])` and `("e", [7;8])`. What is the type of this search tree?
2. Declare a function `entriesOf` t that returns a list containing the entries of the search tree t. The keys should appear in increasing order in the list, that is, the entry with the smallest key appears first and the entry with the largest key appears last in the list. What is the type of the function?
3. Declare a function `find` $k\,t$ that finds the value associated with key k in the search tree t. If k is not a key in t, then a suitable exception should be raised.
4. Declare a function `add` $k\,v\,t$. If k is not a key in the search tree t, then the value of the function is the search tree obtained from t by adding the entry (k, v). If there is an entry (k, v_{old}) in t, then the value of the function is the search tree obtained from t by replacing (k, v_{old}) with (k, v).
5. Declare a function that checks whether a given value t of type `T<'a,'b>` is a search tree, that is, whether t satisfies the search tree invariant.