Written Examination, December 20th, 2017 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 5 problems which are weighted approximately as follows: Problem 1: 15%, Problem 2: 30%, Problem 3: 10%, Problem 4: 15%, Problem 5: 30%

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 book, e.g., List, Set, Map, Seq etc.

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

Problem 1 (15%)

Consider the following F# declarations:

```
let rec f1 a b = if a > b then f1 (a-b) b else a;;

let rec f2 a b = if a > b then 1 + f2 (a-b) b else 0;;
```

In `f1` e_1 e_2 and `f2` e_1 e_2 you can in below questions assume that $e_1 \geq 0$ and $e_2 > 0$.

1. Give evaluations (using $\leadsto$) for `f1 23 3` and `f2 23 3` determining the values of these expressions.
2. Give the types for `f1` and `f2`, and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f1` is tail recursive. Give a brief explanation of why this is the case.
4. The declaration of `f2` is *not* tail recursive. Give a brief explanation of why this is the case and provide a declaration of a tail-recursive variant of `f2` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
5. Give a brief account of the concept of continuation (in the context of tail-recursive functions).

Problem 2 (30%)

We consider now *observations* of birds, where an observation consists of the *species* and the *location* and *time* of the observation. This is captured in the following type declarations:

```
type Species     = string
type Location    = string
type Time        = int
type Observation = Species * Location * Time

let os = [("Owl","L1",3); ("Sparrow","L2",4); ("Eagle","L3",5);
          ("Falcon","L2",7); ("Sparrow","L1",9);  ("Eagle","L1",14)]
```

The value `os` is a list containing 6 observations. Time points are simplified to just being integers, as we just need simple comparisons of time points in this problem.

1. Declare a function `locationsOf: Species -> Observation list -> Location list`, so that `locationsOf` *s os* gives the list of locations from observations of species *s* in *os*. It is allowed to have repeated elements in the resulting list of locations.

We would like to get an *occurrence count* or just *count* of the different species in an observation list. To this end we introduce the type

```
type Count<'a when 'a:equality> = ('a*int) list
```

An element (a_i, c_i) of an occurrence count $[(a_1, c_1); \ldots; (a_n, c_n)]$, denotes that a_i has count c_i. We assume that the a_i's in an occurrence count are all different and that $c_i > 0$.

2. Declare a function `insert` *a occ* that gives the occurrence count obtained from *occ* by incrementing the count of *a* with 1. That is, if there is no count for *a* in *occ* then $(a, 1)$ is included in the resulting occurrence count. Otherwise, there is an element (a, c) in *occ* and that element is replaced by $(a, c + 1)$ in the result. Give the type of `insert`.
3. Declare a function `toCount: Observation list -> Count<Species>` that gives the occurrence count of species in a list of observations. For example, `toCount os` is an occurrence count with 4 elements: (`"Owl"`,1), (`"Sparrow"`,2"), (`"Eagle"`,2) and (`"Falcon"`,1).

A *time interval* (type `Interval`) is a pair (t_1, t_2) of time points, where we assume below that $t_1 \leq t_2$. A time point t is in this interval when $t_1 \leq t \leq t_2$.

```
type Interval = Time * Time
```

4. Declare a function `select` *f intv os* $= [f(o_{i_1}); \ldots; f(o_{i_k})]$, where *f* is a function on observations, *intv* is an interval and $o_{i_1}, \ldots, o_{i_k}$ are all observations from *os* having time points in *intv*.
5. Use `select` to give an expression *e* for a list of pairs of species and locations from observations in `os` that are made in the time interval (4,9). That is, the value of *e* is a list with 4 elements: (`"Sparrow"`,`"L2"`); (`"Eagle"`,`"L3"`); (`"Falcon"`,`"L2"`); (`"Sparrow"`,`"L1"`).

 ... Continued on next page

Problem 3 (10%)

Consider the following F# declarations:

```
let rec f g = function
              | (x::xs,y::ys) -> g x y || f g (x::xs,ys) || f g (xs,y::ys)
              | _             -> false

let h z = f (>) z;;
```

1. Give the (most general) types for `f` and `h`, and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps. Supplement your descriptions with two simple examples, illustrating applications of `f` and `h`.

Problem 4 (15%)

Consider the following F# declarations:

```
type K<'a> = L | M of K<'a> * 'a * K<'a>

let rec c x = match x with
              | L         -> []
              | M(y,v,w) -> c y @ [v] @ c w

let rec d i = function
              | L         -> (L,i)
              | M(x,y,z) -> let (x1,j) = d i x
                            let (z1,k) = d (j+1) z
                            (M(x1,(y,j),z1),k);;
```

1. Give four values of type `K<string * (int list)>`.
2. Give the (most general) types for `c` and `d`.
3. Describe what `c` computes and what `d 0` computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 5 (30%)

In this problem we consider *terms* of the following form:

- A *variable* x is a *term*,
- an *integer constant* c is a *term*, and
- if f is a *function symbol* and $t_1, t_2, \ldots t_n$ are n terms, where $n \geq 0$, then $f(t_1, t_2, \ldots, t_n)$ is a term. The term $f(t_1, t_2, \ldots, t_n)$ is called a *function application*.

We capture terms in F# by the following type declaration:

```
type Term = | V of string | C of int | F of string * Term list
```

where `V` is the constructor for variables, `C` is the constructor for integer constants and `F` is the constructor for function applications. Notice that variables and function symbols are characterized by strings, that is, `V "x"`, `C 3`, `F("f0",[])`, `F("f1",[C 3;F("f0",[])])` and `F("max",[V "x";C 3])` are 5 values of type `Term`.

A term t is called a *ground term* if it does not contain any variables. That is, 3 out of the above 5 terms are ground terms.

1. Declare a function `isGround: Term -> bool` to check whether a term is a ground term.
2. Declare a function `toString: Term -> string` that gives textual representations of terms, as illustrated by the following example:
   ```
   let t6 = F("f3",[F("f2",[C 1; C 2]); F("f1",[V "x"]); F("f0",[])]);;
   toString t6;;
   > val it : string = "f3(f2(1,2),f1(x),f0())"
   ```
   That is, the textual representations of the arguments in ts of a function application `F(`f`,`ts`)` are separated by comma (",").
3. Declare a function `subst` (for substitution) such that `subst` x t' t is the term obtained from t by replacing every occurrence of `V` x in t with t'. State the type of `subst`.

We say that function symbol f is used with *arity* n when f is applied to n terms in an application. For example, `"f0"`, `"f1"`, `"f2"` and `"f3"` are used with arities $0, 1, 2$ and 3, respectively, in `t6`.

A term t is *illegal* if it contains two (or more) function applications involving a function symbol f, that is used with different arities. Otherwise t is called a *legal* term. All example terms above are legal, whereas `F("g", [C 2; F("g",[])])` is an illegal term because function symbol `"g"` is used twice, one time with arity 2 and one time with arity 0.

4. Declare a function `extractArities: Term -> Map<string,int> option`. The value of `extractArities` t is `None` when t is an illegal term. Otherwise `extractArities` t = `Some` m, where m is a map. The keys of m are the function symbols occurring in t and (f, n) is a entry of m, when f is used with arity n in t. For example, `extractArities t6` gives `Some` m, where m contains 4 entries (`"f0"`,0), (`"f1"`,1), (`"f2"`,2) and (`"f3"`,3).