Written Examination, December 18th, 2014 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: 25%, Problem 3: 20%, Problem 4: 35%

Marking: 7 step scale.

Problem 1 (20%)

We consider *relations* that are represented by lists of pairs: $[(x_0, ys_0); (x_1, ys_1); \ldots; (x_n, ys_n)]$. We say that x is related to y when there is a pair (x_i, ys_i) in the list where $x = x_i$ and y is an element of the list ys_i. The following type is used for relations:

```
type Rel<'a,'b> = ('a * 'b list) list

let rel: Rel<int,string> = [(1, ["a"; "b"; "c"]); (4,["b"; "e"])];;
```

The value `rel` describes a relation where, for example, 1 and "b" and 4 and "e" are related, while 1 and "e" and 2 and "a" are not related.

We require that the x_i's in $[(x_0, ys_0); (x_1, ys_1); \ldots; (x_n, ys_n)]$ are all different; but we do not care about repetitions and the order of the elements in ys_i.

1. Declare a function: `apply: 'a -> Rel<'a,'b> -> 'b list`, where `apply` x rel finds the list of elements related to x in rel. For example: `apply 1 rel` = `["a"; "b"; "c"]` and `apply 0 rel = []`.

2. Declare a function `inRelation` x y rel that checks whether x and y are related in rel. For example, `inRelation 4 "e" rel = true` and `inRelation 1 "e" rel = false`.

3. Declare a function `insert` x y rel which returns the relation obtained from rel by adding that x is related to y. For example: `insert 2 "c" [(1,["a"]); (2,["b"])]` could give `[(1, ["a"]); (2, ["c"; "b"])]`.

4. Declare a function `toRel:('a*'b) list -> Rel<'a,'b>` that converts a list of pairs to a relation, e.g. `toRel[(2,"c");(1,"a");(2,"b")]` could give `[(2,["c";"b"]);(1,["a"])]`.

Problem 2 (25%)

1. Declare a function: `multTable: int -> seq<int>` so that `multTable` n gives the sequence of the first 10 numbers in the multiplication table for n. For example, `multTable` 3 is the sequence of numbers $3, 6, 9, 12, \ldots, 30$.

2. Declare a function

   ```
   tableOf: int -> int -> (int -> int -> 'a) -> seq<int*int*'a>
   ```

   so that `tableOf` m n f is the sequence with $n \cdot m$ elements $(i, j, f\, i\, j)$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. The triples $(1, 1, 2), (1, 2, 3), (3, 3, 6), (3, 4, 7)$ are examples of elements in the sequence `tableOf` 3 4 $(+)$. The order in which the elements occur is of no significance.

3. Give a declaration for the infinite sequence of strings `"a"`, `"aa"`, `"aaa"`, `"aaaa"`, ...

Consider the following declaration:

```
let rec f i = function
   | []  -> []
   | x::xs -> (x+i)::f (i*i) xs;;
```

4. Give the (most general) type of `f` and describe what `f` computes. Your description should focus on *what* it computes, rather than on individual computation steps.

5. The function `f` is *not* tail recursive.

   1. Make a tail-recursive variant of `f` using an accumulating parameter.
   2. Make a continuation-based tail-recursive variant of `f`.

Problem 3 (20%)

Consider the following F# declarations:

```
type T<'a> = N of 'a * T<'a> list

let rec f(N(e,es)) = e :: g es
and g = function
        | []    -> []
        | e::es -> f e @ g es;;

let rec h p t =
   match t with
   | N(e,_) when p e -> N(e,[])
   | N(e,es)         -> N(e, List.map (h p) es);;

let rec k (N(_, es)) = 1 + List.fold max 0 (List.map k es);;
```

1. Give three values of type `T<string>`.

2. Give the (most general) types of `f`, `g`, `h` and `k` and describe what each of these four functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 4 (35%)

A *sample space* is the set of all *outcomes* of an experiment. If the experiment is 'toss a coin', the sample space consists of two samples: 'head' (Danish 'krone') or 'tail' (Danish: 'plat'). The probability of each outcome is $\frac{1}{2}$ if a fair coin is used. This is illustrated in the left *probability tree* of Fig. 1, which also contains annotations such as "head?", "head: you win" and "tail: you lose". When an experiment is given by a sequential process, such

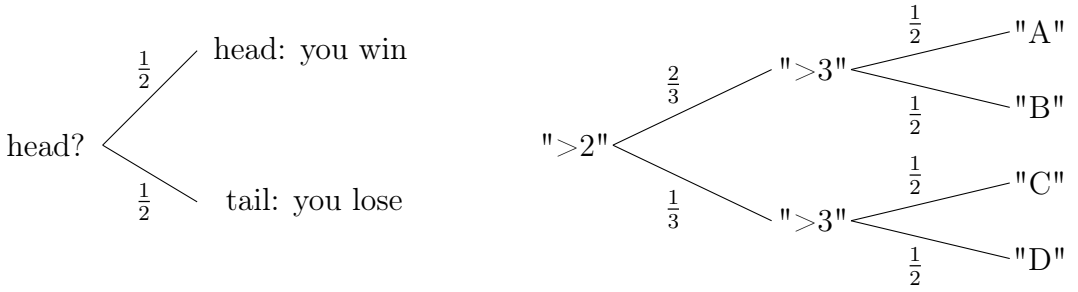

Figure 1: Two probability trees

as tossing a coin three times, a sample is a list where the elements describe the outcome at each stage of the process. If a coin is tossed three times, a list comprising 'tail', 'head' and 'tail' is one sample and the sample space has 8 elements.

We shall consider a simple form of *probability trees* to represent sample spaces of sequential processes, where the outcomes at each stage in the process is either *success* or *failure*. The left tree in Fig. 1 is such a tree when we consider 'head' as success and 'tail' as failure. The right tree in the figure is a probability tree for a process where a dice (Danish: 'terning') is rolled twice. The first roll is successful when more than 2 pips (Danish: 'øjne') are facing up (with probability $\frac{2}{3}$) and the second roll is successful when more than 3 pips are facing up (with probability $\frac{1}{2}$). We shall use the following F# types to model this:

```
type Outcome  = | S | F       // S: for success and F: for failure
type Sample   = Outcome list
type ProbTree = | Branch of string * float * ProbTree * ProbTree
                | Leaf of string
```

The F# representation of the right tree in Fig. 1 (where $\frac{2}{3}$ is approximated by 0.67) is:

```
let exp  = Branch(">2",0.67, Branch(">3",0.5, Leaf "A", Leaf "B")
                           , Branch(">3",0.5, Leaf "C", Leaf "D"))
```

For a branch `Branch`(ds, p, tl, tr), the string ds describes a successful stage of an experiment, the float value p is the probability for a successful outcome leading to the left subtree tl. Therefore, $1.0 - p$ is the probability for a failing outcome leading to the right subtree tr. Notice that the successful branches are the upper branches in Fig. 1.

1. Declare a function `probOK: ProbTree -> bool` that is true iff every probability p occurring in a probability tree satisfies: $0 \leq p \leq 1$.

A list of outcomes *os* is a *correct sample* for a given probability tree t, if traversing t as *os* describes leads to a leaf. For example, if `F` is the head of *os* then the right subtree tr of a branch `Branch`(ds, p, tl, tr) is chosen for further traversal using the tail of *os*. The list of outcomes [`F`; `S`] is a correct sample for the right subtree in Fig. 1 because it leads to the leaf with annotation `"C"`. Any correct sample for this tree has length 2.

2. Declare a function `isSample`(os, t) that is true iff *os* is a correct sample given t. Furthermore, state the type of `isSample`.

The *description* of a correct sample $os = [o_1; \ldots; o_n]$ for a probability tree t is a tuple $([(o_1, ds_1); \ldots; (o_n, ds_n)], p, s)$ where ds_i is the string in a branch node `Branch`(ds_i, p_i, tl_i, tr_i) describing stage i in the experiment according to *os*, p is the probability of the sample (described below), and s is the string in the leaf node reached by *os*. The probability p of the sample *os* is the product $p'_1 \cdot p'_2 \cdot \cdots \cdot p'_n$, where p'_i is the probability of outcome o_i of *os*, that is, p_i if $o_i =$ `S` and $1.0 - p_i$ if $o_i =$ `F`. For example, the description of the sample [`F`; `S`] for the probability tree `exp` is

`([(F,">2");(S,">3")], 0.165, "C")`, because $0.165 = (1.0 - 0.67) \cdot 0.5$.

3. Declare a type `Description` for descriptions and a function `descriptionOf` *os t* that gives the description of the sample *os* for the probability tree t. The function should raise an exception if *os* is not a correct sample.

4. Declare a function `allDescriptions: PropTree -> Set<Description>` that computes the set of all descriptions for a probability tree. The set of all descriptions for `exp`, for example, has the following 4 elements:

```
([(S,">2");(S,">3")], 0.335, "A"),    ([(S,">2");(F,">3")], 0.335, "B"),
([(F,">2");(S,">3")], 0.165, "C") and ([(F,">2");(F,">3")], 0.165, "D")
```

Let *pred* : `string -> bool` be a predicate on strings and t a probability tree. The probability of samples leading to leaves of t whose strings satisfy *pred* is the sum of the probabilities of these samples. For example, the probability of samples for `exp` leading to leaves annotated with either `"B"` or `"C"` is $0.335 + 0.165 = 0.5$.

5. Declare a function `probabilityOf: PropTree -> (string->bool) -> float`, so that `probabilityOf` *t pred* is the probability of reaching leaves `Leaf` *s* where *pred s* is true.

6. Show how `probabilityOf` can be used to calculate the probability of samples for `exp` leading to leaves annotated with either `"B"` or `"C"`.