Written Examination, December 19th, 2012 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: 25%, Problem 2: 35%, Problem 3: 20%, Problem 4: 20%

Marking: 7 step scale.

Problem 1 (Approx. 25%)

In this problem we will consider simple competitions, where persons, identified by their names, achieve scores. A result is a pair (n, sc) consisting of a name n (given by a string) and a score sc (given by an integer). This leads to the following declarations:

```
type Name   = string;;
type Score  = int;;
type Result = Name * Score;;
```

A score is called legal if it is greater than or equal to 0 and smaller than or equal to 100.

1. Declare a function `legalResults: Result list -> bool` that checks whether all scores in a list of results are legal.

2. Declare a function `maxScore` that extracts the best score (the largest one) in a non-empty list of results. If the list is empty, then we do not care about the result of the function.

3. Declare a function `best: Result list -> Result` that extracts the best result from a non-empty list of results. An arbitrary result with the best score can be chosen if there are more than one. If the list is empty, then we do not care about the result of the function.

4. Declare a function `average: Result list -> float` that finds the average score for a non-empty list of results. If the list is empty, then we do not care about the result of the function.

5. Declare a function `delete: Result -> Result list -> Result list`. The value of `delete` r rs is the result list obtained from rs by deletion of the first occurrence of r, if such an occurrence exists. If r does not occur in rs, then `delete` r $rs = rs$.

6. Declare a function `bestN: Result list -> int -> Result list`, where the value of `bestN` rs n, for $n \geq 0$, is a list consisting of the n best results from rs. The function should raise an exception if rs has fewer than n elements.

... Continued on next page

Problem 2 (Approx. 35%)

In this problem we consider simple *type checking* in connection with a simple imperative language. We consider types given by the following declaration of a type `Typ`.

```
type Typ = | Integer
           | Boolean
           | Ft of Typ list * Typ;;
```

Hence, we have an integer type (constructor `Integer`), a Boolean type (construct `Boolean`) and function types constructed using the constructor `Ft`, where `Ft`$([t_1; t_2; \ldots; t_n], t)$, is the type for a function having n arguments with types $t_1, \ldots, t_n$ and the value of the function has type t. The addition function has the type `Ft([Integer;Integer],Integer)` and the greater than function has the type `Ft([Integer;Integer],Boolean)`, for example.

A *declaration* is a pair (x, t) of type `Decl`, which associates the type t with a *variable* x:

```
type Decl = string * Typ;;
```

For a list of declarations $[(x_0, t_0); \ldots; (x_n, t_n)]$ we shall require that the variables are all different, that is, $x_i \neq x_j$, when $i \neq j$.

1. Declare a function `distinctVars: Decl list -> bool`, where `distinctVars` *decls* returns true if all variables in *decls* are different.

You can from now on assume that the variables in a declaration list are different.

A *symbol table* associates types with the variables and functions in programs. We model symbol tables by values of the following `Map` type, where an entry associate a type with a string:

```
type SymbolTable = Map<string,Typ>;;
```

2. Declare a function `toSymbolTable: Decl list -> SymbolTable` that transforms a list of declarations into a symbol table.

3. Declare a function `extendST: SymbolTable -> Decl list -> SymbolTable`, where the value of `extendST` *sym decls* is the symbol table obtained from *sym* by adding entries (x, t), for every declaration (x, t) in *decls*. An existing entry in *sym* having x as key will be overridden by this operation.

We consider expressions generated from variables (constructor `V`) using function application (constructor `A`), where, e.g., `A(">",[V "x";V "y"])` represents the comparison $x > y$:

```
type Exp = | V of string
           | A of string * Exp list;;
```

Suppose that a symbol table *sym* associates the type `Integer` with `"x"` and `"y"`, and the type `Ft([Integer;Integer],Boolean)` with `">"`. All symbols (variables and functions) in the expression `A(">",[V "x";V "y"])` are therefore defined in *sym*. Furthermore, the expression is well-typed since the types of the arguments to `>` match the argument types in `Ft([Integer;Integer],Boolean)`, and the type of `A(">",[V "x";V "y"])` is `Boolean`.

4. Declare a function `symbolsDefined: SymbolTable -> Exp -> bool`, where the value of the expression `symbolsDefined` *sym* *e* is true if there is an entry in *sym* for every symbol (variable or function) occurring in *e*.

5. Declare a function `typOf: SymbolTable -> Exp -> Typ`, so that `typOf` *sym* *e* gives the type of *e* for the symbol table *sym*. The function should raise an exception if *e* is not well-typed. You may assume that all symbols in *e* are defined in *sym*.

We consider statements generated from assignments using sequential composition, if-then-else statements, while statements and block statements:

```
type Stm = | Ass of string * Exp          // assignment
           | Seq of Stm * Stm             // sequential composition
           | Ite of Exp * Stm * Stm       // if-then-else
           | While of Exp * Stm           // while
           | Block of Decl list * Stm;;   // block
```

The *well-typedness* of a statement for a given symbol table *sym* is given by:

- An assignment `Ass`(x, e) is well-typed if x and the symbols of e are defined in *sym* and x and e have the same type.
- A sequential composition `Seq`(stm_1, stm_2) is well-typed if stm_1 and stm_2 are.
- An if-then-else statement `Ite`(e, stm_1, stm_2) is well-typed if the symbols in e are defined in *sym*, e has type `Boolean`, and stm_1 and stm_2 are well-typed.
- A while statement `While`(e, stm) is well-typed if the symbols in e are defined in *sym*, e has type `Boolean`, and *stm* is well-typed.
- A block statement `Block`$(decls, stm)$ is well-typed if the variables in *decls* are all different, and *stm* is well-typed in the symbol table obtained by extending *sym* with the declarations of *decls*.

6. Declare a function `wellTyped: SymbolTable -> Stm -> Bool` that checks that a statement is well-typed for a given symbol table, and if so returns true.

Problem 3 (20%)

Consider the following F# declarations:

```
let rec h a b =
    match a with
    | []   -> b
    | c::d -> c::(h d b);;

type T<'a,'b> = | A of 'a | B of 'b | C of T<'a,'b> * T<'a,'b>;;

let rec f1 = function
    | C(t1,t2) -> 1 + max (f1 t1) (f1 t2)
    | _        -> 1;;

let rec f2 = function
    | A e | B e  -> [e]
    | C(t1,t2)   -> f2 t1 @ f2 t2;;

let rec f3 e b t =
    match t with
    | C(t1,t2) when b -> C(f3 e b t1, t2)
    | C(t1,t2)        -> C(t1, f3 e b t2)
    | _        when b -> C(A e, t)
    | _               -> C(t, B e);;
```

1. Give the type of `h` and describe what `h` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
2. Write a value of type `T<int,bool>` using all three constructors `A`, `B` and `C`.
3. Write a value of type `T<'a list,'b option>` using all three constructors `A`, `B` and `C`.
4. Give the types of `f1`, `f2` and `f3`, and describe what each of these three functions compute.

... Continued on next page

Problem 4 (Approx. 20%)

Consider the following F# declarations:

```
type 'a tree = | Lf
               | Br of 'a * 'a tree * 'a tree;;

let rec sumTree = function
    | Lf            -> 0                                (* sT1 *)
    | Br(x, t1, t2) -> x + sumTree t1 + sumTree t2;;    (* sT2 *)

let rec toList = function
    | Lf            -> []                               (* tL1 *)
    | Br(x, t1, t2) -> x::(toList t1 @ toList t2);;     (* tL2 *)

let rec sumList = function
    | []    -> 0                                        (* sL1 *)
    | x::xs -> x + sumList xs;;                         (* sL2 *)

let rec sumListA n = function
    | []    -> n                                        (* sLA1 *)
    | x::xs -> sumListA (n+x) xs;;                      (* sLA2 *)
```

1. Prove that

   $$\texttt{sumTree}\ t = \texttt{sumList}(\texttt{toList}\ t)$$

   holds for all trees t of type `int tree`.

   In the proof you can assume that

   $$\begin{aligned}&\texttt{sumList}((\texttt{toList}\ t_1)\ \texttt{@}\ (\texttt{toList}\ t_2))\\ &= \texttt{sumList}(\texttt{toList}\ t_1) + \texttt{sumList}(\texttt{toList}\ t_2)\end{aligned}$$

   holds for all trees t_1 and t_2 of type `int tree`.

2. Prove that

   $$\texttt{sumListA}\ n\ xs = n + \texttt{sumList}(xs)$$

   holds for all integers n and all lists xs of type `int list`.

Written Examination, December 18th, 2013 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 3 problems which are weighted approximately as follows: Problem 1: 30%, Problem 2: 30%, Problem 3: 40%

Marking: 7 step scale.

Problem 1 (Approx. 30%)

A *multiset* (or *bag*) is a generalization of a set, where an element e is associated with a *multiplicity*, i.e. the number of times e occurs in the multiset. We shall represent a finite multiset ms by a list of pairs $[(e_1, n_1); \ldots; (e_k, n_k)]$, where a member (e_i, n_i) represents that e_i is a member of ms with multiplicity n_i, i.e. e_i occurs n_i times in ms.

For a representation $[(e_1, n_1); \ldots; (e_k, n_k)]$ of a multiset we require that every multiplicity n_i is positive, and that the elements are distinct, i.e. $e_i \neq e_j$, for $i \neq j$. This property is called the *multiset invariant*. A consequence of this is that the empty multiset is represented by the empty list.

We shall use the type `Multiset<'a>` declared as follows:

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

For example `[("b",3); ("a",5); ("d",1)]` has type `Multiset<string>` and represents the multiset with 3 occurrences of "b", 5 of "a" and 1 of "d".

1. Declare a function `inv: Multiset<'a> -> bool` such that `inv`(ms) is true when ms satisfies the multiset invariant.

In your solutions to the below questions, you can assume that multisets occurring in arguments satisfy the multiset invariant, and the declared functions must preserve this property, i.e. results must satisfy this invariant as well.

2. Declare a function `insert: 'a -> int -> Multiset<'a> -> Multiset<'a>`, where `insert` e n ms is the multiset obtained by insertion of n occurrences of the element e in ms. For example: `insert "a" 2 [("b",3); ("a",5); ("d",1)]` will result in a multiset having 7 occurrences of "a".

3. Declare a function `numberOf`, where `numberOf` e ms is the multiplicity (i.e. the number of occurrences) of e in the multiset ms. State the type of the declared function.

4. Declare a function `delete`, where `delete` e ms is the multiset obtained from ms by deletion of one occurrence of the element e.

5. Declare a function `union: Multiset<'a> * Multiset<'a> -> Multiset<'a>`, for making the union of two multisets. This function generalizes the union function on sets in a natural way taking multiplicities into account, e.g. the result of

   ```
   union ([("b",3); ("a",5); ("d",1)], [("a",3); ("b",4); ("c",2)])
   ```

   is the multiset containing 8 occurrences of "a", 7 of "b", 2 of "c", and 1 of "d".

 ... Continued on next page

We shall now represent multisets by maps from elements to multiplicities:

```
type MultisetMap<'a when 'a : comparison> = Map<'a,int>;;
```

This representation of a multiset ms has a simpler invariant: the multiplicity n of each entry (e, n) of ms satisfies $n > 0$.

6. Give new declarations for `inv`, `insert` and `union` on the basis of the map representation.

Problem 2 (30%)

Consider the following F# declarations:

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


type 'a Tree = | Lf
               | Br of 'a Tree * 'a * 'a Tree;;

let rec g p = function
              | Lf                  -> None
              | Br(_,a,t) when p a -> Some t
              | Br(t1,a,t2)        -> match g p t1 with
                                      | None -> g p t2
                                      | res  -> res;;
```

Please remember that the declaration of `'a option` is

```
type 'a option = None | Some of 'a;
```

1. Give the types of `f` and `g` 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.
2. 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`.
    3. Give a brief discussion of which tail-recursive version of `f` you prefer?

Consider now the F# declarations:

```
let rec h f (n,e) = match n with
                    | 0 -> e
                    | _ -> h f (n-1, f n e);;

let A = Seq.initInfinite id;;

let B = seq { for i in A do
                  for j in seq {0 .. i} do
                        yield (i,j)        };;

let C = seq { for i in A do
                  for j in seq {0 .. i} do
                      yield (i-j,j)        };;

let X = Seq.toList (Seq.take 4 A);;
let Y = Seq.toList (Seq.take 6 B);;
let Z = Seq.toList (Seq.take 10 C);;
```

3. Consider the function `h` in this question:

    1. What is the value of `h ( * ) (4,1)`?
    2. What is the type of `h`?
    3. Describe briefly what `h` computes.

3. Consider the declarations for `A`, `B`, `C`, `X`, `Y` and `Z`:

    1. Give types for `A`, `B` and `C`.
    2. Give the values of `X`, `Y` and `Z`?
    3. Characterize the values of `A`, `B` and `C`.

... Continued on next page

Problem 3 (40%)

We shall now consider *books* that are described by a list of *chapters*. Each chapter is described by a *title* and a list of *sections*. A section is described by a title and a list of *elements*, which can either be *paragraphs* (characterized by strings) or *sub-sections*. This is captured by the following type declarations:

```
type Title = string;;

type Section = Title * Elem list
and  Elem    = Par of string | Sub of Section;;

type Chapter = Title * Section list;;
type Book    = Chapter list;;
```

The mutual recursion between sections and elements allows for arbitrary nesting of sub-sections. This is illustrated by the following examples:

```
let sec11 = ("Background", [Par "bla"; Sub(("Why programming", [Par "Bla."]))]);;
let sec12 = ("An example", [Par "bla"; Sub(("Special features", [Par "Bla."]))]);;
let sec21 = ("Fundamental concepts",
             [Par "bla"; Sub(("Mathematical background", [Par "Bla."]))]);;
let sec22 = ("Operational semantics",
             [Sub(("Basics", [Par "Bla."])); Sub(("Applications", [Par "Bla."]))]);;
let sec23 = ("Further reading",  [Par "bla"]);;
let sec31 = ("Overview", [Par "bla"]);;
let sec32 = ("A simple example", [Par "bla"]);;
let sec33 = ("An advanced example", [Par "bla"]);;
let sec41 = ("Status", [Par "bla"]);;
let sec42 = ("What's next?", [Par "bla"]);;
let ch1 = ("Introduction", [sec11;sec12]);;
let ch2 = ("Basic Issues", [sec21;sec22;sec23]);;
let ch3 = ("Advanced Issues", [sec31;sec32;sec33;sec34]);;
let ch4 = ("Conclusion", [sec41;sec42]);;
let book1 = [ch1; ch2; ch3; ch4];;
```

1. Declare a function `maxL` to find the largest integer occurring in a list with non-negative integers. The function must satisfy `maxL [] = 0`.

2. Declare a function `overview` to extract the list of titles of chapters from a book. For example, the overview for `book1` is:

```
overview book1 =
   ["Introduction"; "Basic Issues"; "Advanced Issues"; "Conclusion"]
```

 ... Continued on next page

Each chapter occurs at *depth* 1. A top-level section, i.e. a section which is not a sub-section, occurs a depth 2. A sub-section has a depth which is one larger than the depth of the section of which it is an immediate part. For example, the depth of the sub-section with title "Applications" in `book1` is 3 and the section with title "Overview" has depth 2.

3. Declare functions:

```
depthSection: Section -> int
depthElem:    Elem -> int
depthChapter: Chapter -> int
depthBook:    Book -> int
```

to extract the maximal depth of sections, elements, chapters and books. For example the maximal depth of `book1` is 3, as `book1` has sub-sections, but no sub-sub-section.

We shall now make a *table of contents* (type `Toc` below) for a book. In a table of contents we use lists to number entries (see the types `Entry` and `Numbering` below). A *numbering* such as $[i; j; k; l]$ is the number of the l'th sub-sub-section, of the k'th sub-section of the j'th section in the i'th chapter. Notice that such lists have varying lengths. For example, [2] is the number of Chapter 2, i.e the chapter with title "Basic Issues" in the previous example, and [2;2;1] is the number of the sub-section with title "Basics" in Chapter 2.

```
type Numbering = int list;;
type Entry = Numbering * Title;;
type Toc = Entry list;;
```

The table of contents for the previous example is:

```
[([1], "Introduction");
 ([1; 1], "Background");
 ([1; 1; 1], "Why programming");
 ([1; 2], "An example");
 ([1; 2; 1], "Special features");
 ([2], "Basic Issues");
 ([2; 1], "Fundamental concepts");
 ([2; 1; 1], "Mathematical background");
 ([2; 2], "Operational semantics");
 ([2; 2; 1], "Basics");
 ([2; 2; 2], "Applications");
 ([2; 3], "Further reading");
 ([3], "Advanced Issues");
 ([3; 1], "Overview");
 ([3; 2], "A simple example");
 ([3; 3], "An advanced example");
 ([3; 4], "Summary");
 ([4], "Conclusion");
 ([4; 1], "Status");
 ([4; 2], "What's next?")]
```

4. Declare a function, `tocB: Book` $\rightarrow$ `Toc`, to compute the table of contents for a book.

BSWU BFNP **IT University, F2013**

Written exam, Functional Programming
Tuesday 11 June 2013

Version 1.00 of 2013-06-01

These exam questions comprise 7 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators etc. during the examination. You are **not** allowed to use computers, mobile phones, PDA's, iPods, iPads or any other form of device that can execute programs written in F# or C# or Java or Scala or that can communicate with other devices.

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

Question 1 (25 %)

In this question we will consider a simple cash register, where items are identified by a unique *id*. An item also has a *name* and a unit *price*. This leads to the following declarations:

```
type item = {id : int;
             name : string;
             price : float}

type register = item list
```

Question 1.1

Declare a value of type `register` with the following four items:

1. Item with id 1 named "Milk" with price 8.75
2. Item with id 2 named "Juice" with price 16.25
3. Item with id 3 named "Rye Bread" with price 25.00
4. Item with id 4 named "White Bread" with price 18.50

Question 1.2

Declare an exception `Register` and a function

```
getItemById : int -> register -> item
```

so that `getItemById i r` extracts the first occurrence of an item with id `i` from the given register `r`. The function should raise the exception `Register` if the item is not in the register. The exception should contain an additional string explaining the error.

Question 1.3

Declare a function

```
nextId : register -> int
```

so that `nextId r` returns the next id to use in the register. If *maxId* is the maximum id currently used in register `r`, then the next id is defined as *maxId* + 1. The first id to use on an empty register is 1.

Question 1.4

Declare a function

```
addItem : string -> float -> register -> register
```

so that `addItem n p r` adds a new item with name `n` and unit price `p` to the register `r`. The new item must have the next available id as defined by the function `nextId`.

Question 1.5

Declare a function

```
deleteItemById : int -> register -> register
```

so that `deleteItemById i r` deletes an item with id `i` from register `r` and returns the updated register. The register is unchanged if no item with id `i` exists.

Question 1.6

Declare a function

```
uniqueRegister : register -> bool
```

so that `uniqueRegister r` returns true if all items in the register `r` have unique id's, and returns false otherwise.

Question 1.7

Declare a function

```
itemsInPriceRange : float -> float -> register -> register
```

so that `itemsInPriceRange p d r`, given a price `p`, a delta `d` and a register `r`, returns all items in the register whose prices are within the range $[\mathtt{p} - \mathtt{d}, \ldots, \mathtt{p} + \mathtt{d}]$.
The function returns a new register. You may assume that the delta `d` is positive.

Question 2 (25 %)

Consider the following F# declaration:

```
let rec f n m =
  if m=0 then n
  else n * f (n+1) (m-1)
```

Question 2.1

Give the type of `f` and describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.

Question 2.2

The function `f` is not tail recursive. Declare a function `f'` that is a tail recursive version of `f`.
Hint: You can either use a continuation function or an accumulating parameter.

Question 2.3

Consider the following F# declaration:

```
let rec z xs ys =
  match (xs, ys) with
    ([],[]) -> []
  | (x::xs,[]) -> (x,x) :: (z xs ys)
  | ([],y::ys) -> (y,y) :: (z xs ys)
  | (x::xs,y::ys) -> (x,y)::(z xs ys)
```

Give the type of `z` and describe what `z` computes. Your description should focus on what `z` computes, rather than on individual computation steps. Give two examples of input and result values that support your description.

Question 2.4

Consider the following F# declaration:

```
let rec s xs ys =
  match (xs,ys) with
    ([],[]) -> []
  | (xs,[]) -> xs
  | ([],ys) -> ys
  | (x::xs,y::ys) -> x::y::s xs ys
```

Give the type of `s` and describe what `s` computes. Your description should focus on what `s` computes, rather than on individual computation steps. Give two examples of input and result values that support your description.

Question 2.5

The function `s` above is not tail recursive. Declare a function `sC` that is a tail recursive version of the function `s`.
Hint: You can use a continuation function.

Question 3 (30 %)

Consider the following F# declarations

```
type Latex<'a> =
    Section of string * 'a * Latex<'a>
  | Subsection of string * 'a * Latex<'a>
  | Text of string * Latex<'a>
  | End

let text1 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    End))))
```

Question 3.1

What is the type of the declaration `text1` above?

Question 3.2

The type `Latex<'a>` represents simple LaTeX like documents and the value `text1` represents the following text

```
1 Introduction
This is an introduction to ...
1.1 A subsection
As laid out in the introduction we ...
```

Declare a function
`addSecNumbers`
that transforms a value as `text1` above into a new value with section numbers added.
For instance `addSecNumbers text1` must return the following value

```
Section ("Introduction", "1",
  Text ("This is an introduction to ...",
    Subsection ("A subsection", "1.1",
      Text ("As laid out in the introduction we ...",
        End))))
```

For a more complicated example consider the F# declaration

```
let text2 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    Subsection ("Yet a subsection", None,
                      Section ("And yet a section", None,
                        Subsection ("A subsection more...", None,
                          End)))))))
```

The declaration `text2` represents the following text

```
1 Introduction
This is an introduction to ...
1.1 A subsection
As laid out in the introduction we ...
1.2 Yet a subsection
2 And yet a section
2.1 A subsection more...
```

The function application `addSecNumbers text2` must return the value

```
let text2' = Section ("Introduction", "1",
               Text ("This is an introduction to ...",
                 Subsection ("A subsection", "1.1",
                   Text ("As laid out in the introduction we ...",
                     Subsection ("Yet a subsection", "1.2",
                       Section ("And yet a section", "2",
                         Subsection ("A subsection more...", "2.1",
                           End)))))))
```

Question 3.3

What is the type of the function `addSecNumbers`?

Question 3.4

We now extend the type `Latex` to also include labels and references.

```
type Latex<'a> =
    Section of string * 'a * Latex<'a>
  | Subsection of string * 'a * Latex<'a>
  | Label of string * Latex<'a>
  | Text of string * Latex<'a>
  | Ref of string * Latex<'a>
  | End
```

Consider the following F# declaration

```
let text3 = Section ("Introduction", "1",
              Label("intro.sec",
                Text ("In section",
                  Ref ("subsec.sec",
                    Text (" we describe ...",
                      Subsection ("A subsection", "1.1",
                        Label("subsec.sec",
                          Text ("As laid out in the introduction, Section ",
                            Ref ("intro.sec",
                              Text (" we ...",
                                End)))))))))
```

Declare a function

```
buildLabelEnv : LaTeX<'a> -> Map<string,string>
```

that given a declaration such as `text3` above returns an environment that maps label names to sections. You can assume that the function `addSecNumbers` has been implemented on the extended version of the type `Latex<'a>`.

The type of the environment is `Map<string, string>`. For the example `text3`, the returned environment contains the following two entries: `"intro.sec"` → `"1"` and `"subsec.sec"` → `"1.1"`.
Hint: First you call `addSecNumbers` to make sure you have section numbers defined.

Question 3.5

Declare a function

```
toString : Latex<'a> -> string
```

that makes a string representation of the given text as illustrated by the examples above.
You can use the value `nl` defined as

```
let nl : string = System.Environment.NewLine
```

to represent a newline to separate sections and sub-sections from ordinary text. You are not expected to do any form of text formatting, text wrapping etc.
Hint: Your function `toString` can make use of the functions `addSecNumbers` and `buildLabelEnv`.

Question 4 (20 %)

Question 4.1

Consider the F# declaration:

```
let mySeq = Seq.initInfinite (fun i -> if i % 2 = 0 then -i else i)
```

Write the result type and result value of evaluating `Seq.take 10 mySeq`.

Question 4.2

Declare the function

```
finSeq : int -> int -> seq<int>
```

so that `finSeq n M` produces the finite sequence $\mathtt{n}, \mathtt{n}+2, \mathtt{n}+4, \ldots, \mathtt{n}+2\mathtt{M}$.

Question 4.3

Consider the following F# declarations

```
type X = A of int | B of int | C of int * int

let rec zX xs ys =
  match (xs,ys) with
    (A a::aS,B b::bS) -> C(a,b) :: zX aS bS
  | ([],[]) -> []
  | _ -> failwith "Error"

let rec uzX xs =
  match xs with
    C(a,b)::cS -> let (aS,bS) = uzX cS
                  (A a::aS,B b::bS)
  | [] -> ([],[])
  | _ -> failwith "Error"
```

Give the types of the functions `zX` and `uzX`. Describe what the two functions compute. Your description should focus on what is computed, rather than the individual computation steps. Give, for each function, three example input values and result values that support your descriptions.

BSWU BFNP **IT University, F2014**

Written exam, Functional Programming
Wednesday 11 June 2014

Version 1.00 of 2014-06-02

These exam questions comprise 8 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any other form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

If you hand-in electronically, then hand-in one ASCII file only, e.g., `bfnp2014.fsx`. Do not use time on formatting your solution in Word or PDF.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (30 %)

We define an *ordered list* as a generalisation of a list where you can add and remove elements at both ends. An ordered list, *l*, with *n* elements is written $e_1, \ldots, e_n$. We use a comma to separate the elements in the ordered list.

An ordered list is implemented using two F# lists; a *front* list: $e_1::e_2::\ldots$ and a *rear* list: $e_n::e_{n-1}::\ldots$. The idea is to have constant time support for adding elements to the ordered list in both ends. Removing elements from the ordered list may require linear time.

The last element added to the end of the ordered list is the first element in the *rear* list. We require the invariant, that *front* `@ List.rev` *rear* represents the ordered list of elements. You can assume, that

all ordered lists occurring in arguments below satisfy this invariant, and all functions must preserve the invariant.
We will use the type `OrderedList<'a>` declared as follows

```
type OrderedList<'a when 'a : equality> =
  {front: 'a list;
   rear: 'a list}
```

For instance, the value

```
let ex = {front = ['x']; rear = ['z';'y']}
```

has type `OrderedList<char>` and represents the ordered list `'x'`, `'y'`, `'z'`.
You must either declare your own exception or use `failwith` to signal errors in your functions below.

Question 1.1

The declaration below represents an ordered list with the following elements: "Hans", "Brian", "Gudrun".

```
let ol1 = {front = ["Hans"; "Brian"; "Gudrun"]; rear = []}
```

- Declare two values, `ol2` and `ol3`, representing the same ordered list but with the rear list being non-empty.
- How many representations exists with the three elements in the given order that fulfils the invariant?

Question 1.2

We define the *canonical* representation of an ordered list to be the representation where the *rear* list is empty.

- Declare a function `canonical:OrderedList<'a>->OrderedList<'a>`, where `canonical ol` returns the canonical representation of `ol`.
- Declare a function `toList:OrderedList<'a'>->List<'a>`, that returns the list of elements. For instance `toList ex` returns the list `['x';'y';'z']`.

Question 1.3

- Declare a function `newOL:unit->OrderedList<'a>`, that returns a new empty ordered list.
- Declare a function `isEmpty:OrderedList<'a>->bool`, that returns true if the ordered list is empty and otherwise false. For instance, `isEmpty (newOL())` returns true and `isEmpty ex` returns false.

Question 1.4

A stack, LIFO, is a special version of an ordered list where you can only add and remove elements from the front of the list.

- Declare the function `addFront:'a->OrderedList<'a>->OrderedList<'a>`, that adds an element to the front of the list. For instance `addFront 'w' ex` returns the ordered list `'w','x','y','z'`
- Declare the function `removeFront:OrderedList<'a>->'a*OrderedList<'a>`, that removes the first element in the ordered list and returns both the first element and the new list. For instance `removeFront ex` returns a pair with the first element being `'x'` and the second element being the ordered list `'y','z'`.
- Declare the function `peekFront:OrderedList<'a>->'a`, that returns the first element in the ordered list, without removing it from the list. For instance `peekFront ex` returns the element `'x'`.

Question 1.5

Declare the function
`append:OrderedList<'a>->OrderedList<'a>->OrderedList<'a>`,
that concatenates two ordered lists. For instance, `append ex ex` returns the ordered list `'x'`, `'y'`, `'z','x','y','z'`.

Question 1.6

Declare a function
`map:('a->'b)->OrderedList<'a>->OrderedList<'b>`,
that applies a function on all elements in the ordered list. The function can be applied on the elements in any order. For instance, `map (fun e -> e.ToString()) ex` returns a new ordered list `"x"`, `"y"`, `"z"` of type `OrderedList<string>`.

Question 1.7

Declare a function
`fold:('State->'T->'State)->'State->OrderedList<'T>->'State`,
with similar semantics as `List.fold`. For instance, `fold (fun acc e -> acc + e.ToString()) "" ex` returns the string `"xyz"`.
Hint: You can use `toList` from above.

Question 1.8

Declare a function
`multiplicity:OrderedList<'a>->Map<'a,int>`,
that returns a mapping from the elements in the ordered list to the number of times the elements are represented in the ordered list. For instance, `multiplicity (addFront 'x' ex)` returns the map where `'x'` is mapped to 2, `'y'` is mapped to 1 and `'z'` is mapped to 1.
Hint: You can use `fold` from above.

Question 2 (20 %)

Consider the following F# declaration:

```
let rec f i = function
    [] -> [i]
  | x::xs -> i+x :: f (i+1) xs
```

The type of `f` is `int -> int list -> int list`. The expression `f 10 [0;1;2;3]` returns the value `[10;12;14;16;14]`.

Question 2.1

- Describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.
- Explain whether the result of calling `f`, with any input, can ever be the empty list?
- Explain whether the function `f`, with any input, can ever go into an infinite loop?

Question 2.2

The function `f` is not tail recursive. Declare a tail-recursive variant, `fA`, of `f` using an accumulating parameter.

Question 2.3

Declare a continuation-based tail-recursive variant, `fC`, of `f`.

Question 3 (20 %)

Question 3.1

Consider the F# declaration:

```
let myFinSeq n M = Seq.map (fun m -> n+n*m) [0..M]
```

Describe the sequence returned by `myFinSeq` when called with an arbitrary integer `n` and `M`.

Question 3.2

Declare the infinite sequence

```
mySeq:int->seq<int>
```

such that `mySeq n` produces the infinite sequence `n+n*i` for `i` $>= 0$. The identifier `i` represents the index of the element in the sequence.
Hint: You can use `Seq.initInfinite`.

Question 3.3

Declare the finite sequence

```
multTable N M
```

to return the element triples `(n,m,n*m)` where `n` $\in [0,\ldots,N]$ and `m` $\in [0,\ldots,M]$.

Question 3.4

Declare the finite sequence

```
ppMultTable:int->int->seq<string>
```

to return the sequence "<n> * <m> is <n*m>" for `n` $\in$ [0,. . .,N] and `m` $\in$ [0,. . .,M], using the sequence `multTable`. The notation <*n*> denotes the textual representation of the integer *n*.
For instance, `Seq.take 4 (ppMultTable 10 10)` returns the value

```
seq ["0 * 0 is 0"; "0 * 1 is 0"; "0 * 2 is 0"; "0 * 3 is 0"]
```

Question 4 (30 %)

We shall now consider *MousePlot*, a small library to describe drawings. In *MousePlot* you have the following *operations*:

```
type opr = MovePenUp
         | MovePenDown
         | TurnEast
         | TurnWest
         | TurnNorth
         | TurnSouth
         | Step
```

Imagine a *mouse* that can turn and step in four directions. The *mouse* has a pen that can be up or down. If the pen is down, a dot is placed on a media, e.g., a piece of paper.
A plot consists of either one *operation* or a sequence of *operations* described by the type *plot*:

```
type plot = Opr of opr
          | Seq of plot * plot
```

A simple rectangle can be declared as follows:

```
let side = Seq(Opr MovePenDown, Seq(Opr Step, Seq(Opr Step, Opr Step)))
let rect = Seq(Seq(Opr TurnEast, side),
             Seq(Opr TurnNorth, Seq(side,
               Seq(Opr TurnWest, Seq(side,
                 Seq(Opr TurnSouth, side))))))
```

The rectangle consists of four sides. Each side is defined as three steps in one of the four directions.

Question 4.1

Consider the example `rect` above.

- Write a function `ppOpr: opr -> string` that returns a pretty printed version of the argument operation. For instance, `ppOpr MovePenUp` returns the string `"MovePenUp"`.
- Write a function `ppOprPlot: plot -> string` that returns a pretty printed version of the argument plot. The returned string must use the notation opr_1 => opr_2 to show that opr_1 comes before opr_2. For instance `ppOprPlot rect` returns the string `"TurnEast => MovePenDown => Step => Step => Step => TurnNorth => MovePenDown => Step => Step => Step => TurnWest => MovePenDown => Step => Step => Step => TurnSouth => MovePenDown => Step => Step => Step"`

Question 4.2

In order to draw a plot, we need to define the *mouse* movements in more detail.

- The *mouse* can turn in four directions: East, West, North and South.
- The *mouse* steps around in a coordinate system defined as:

Step in direction	Δx	Δy
East	+1	0
West	-1	0
North	0	+1
South.	0	-1

For instance, a Step in direction East will increase the x coordinate with one and leave y unchanged.

- The *mouse* stays at the same coordinate when it turns.
- The *mouse* has a pen that can be up (PenUp) or down (PenDown). When down, a dot is placed on a media, e.g., a piece of paper.
- The operation `MovePenDown` makes the *mouse* place a dot at the coordinate where the mouse is located.
- If the pen is down (PenDown), then the operation `Step` will make the *mouse* place a dot at the coordinate it is targeting as defined above.
- Initially the pen is up (PenUp), placed at coordinate (0,0) and heading in the direction East.

Given a plot, the task is to simulate mouse movements and thereby calculate the coordinates where the pen is down, i.e., where the mouse place dots. The simulation uses a *state* containing three components: the *direction* the mouse is heading, the *current coordinate* and the state of the *pen*, i.e., PenUp or PenDown. The *state* is defined as follows:

```
type dir = North
         | South
         | West
         | East
type pen = PenUp
         | PenDown
type coord = int * int
type state = coord * dir * pen
```

For instance, the initial state is defined as

```
let initialState = ((0,0),East,PenUp)
```

The state `((2,-1),South,PenDown)` represents the mouse at coordinate (2,-1) heading South with the pen in down position.

- Declare a function `goStep:state->state` that given a state returns a new state where the mouse has moved one step in the direction given by the state. For instance `goStep initialState` returns the value `((1,0),East,PenUp)`
- Declare a function `addDot:state->coord list->opr->coord list*state`. Given a state, a list of coordinates and an operation, the function performs the operation and if the mouse places a dot, i.e., the pen is down, then adds the coordinate to the list of coordinates. For instance,

```
let (coords1,s1) = addDot initialState [] MovePenDown
let (coords2,s2) = addDot s1 coords1 Step
```

returns

```
val s1 : (int * int) * dir * pen = ((0, 0), East, PenDown)
val coords1 : (int * int) list = [(0, 0)]
val s2 : (int * int) * dir * pen = ((1, 0), East, PenDown)
val coords2 : (int * int) list = [(1, 0); (0, 0)]
```

- Declare a function `dotCoords:plot->coord list`. Given a plot, the function returns the coordinates where the mouse has placed a dot on the media. For instance

```
let coords = dotCoords rect
```

returns

```
val coords : (int * int) list =
  [(0, 0); (0, 1); (0, 2); (0, 3); (0, 3); (1, 3); (2, 3); (3, 3); (3, 3);
   (3, 2); (3, 1); (3, 0); (3, 0); (2, 0); (1, 0); (0, 0)]
```

- Declare a function `uniqueDotCoords:plot->Set<coord>` similar to `dotCoords` except that the coordinates are now returned as a set, i.e., no duplicate coordinates. For instance

```
let coordSet = uniqueDotCoords rect
```

returns

```
val coordSet : Set<int * int> =
  set
    [(0, 0); (0, 1); (0, 2); (0, 3); (1, 0); (1, 3); (2, 0); (2, 3); (3, 0);
     ...]
```

Question 4.3

To shorten the notation used when specifying plots, we shall now extend the type `plot` with a binary overloaded operator, `(+):plot*plot->plot`, to put two plots in sequence to each other. For instance the example rectangle can be written

```
let side2 = Opr MovePenDown + Opr Step + Opr Step + Opr Step
let rect2 = Opr TurnEast + side2 + Opr TurnNorth + side2 +
            Opr TurnWest + side2 + Opr TurnSouth + side2
```

Extend the type `plot`, using *type augmentation*, with the overloaded operator + such that the above can be declared.

BSWU BFNP **IT University, F2015**

Written exam, Functional Programming
Tuesday 2 June 2015

Version 1.10 of 2015-05-27

These exam questions comprise 9 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one ASCII file only, e.g., `bfnp2015.fsx`. Do not use time on formatting your solution in Word or PDF.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (30%)

We define a *multimap* as a generalisation of a map in which more than one value may be associated with and returned for a given key. A multimap m from a set A to a set B is a finite subset A' of A together with a function m defined on A' whose type is $m : A' \to P(B)$, where $P(B)$ is the set of all subsets of B, also called the *power set* of B.
A multimap m can be described in tabular form as shown below.

a_0	$B_0 = \{b_{a0_1}, \ldots\}$
a_1	$B_1 = \{b_{a1_1}, \ldots\}$
$\vdots$	$\vdots$
a_n	$B_n = \{b_{an_1}, \ldots\}$

The left column contains the keys $a_0, \ldots, a_n$ of set A' while the right column contains the corresponding values $m(a_0) = B_0, m(a_1) = B_1, \ldots, m(a_n) = B_n$ where B_i are subsets of elements from the set B.
The *map*-concept in F# is explained in HR on page 113.
A multimap can be implemented using an F# map, `Map`, mapping values from the set A' to an F# list representing a subset of B.
We will use the type `multimap<'a,'b>` declared as follows

```
type multimap<'a,'b when 'a: comparison> =
  MMap of Map<'a,list<'b>>
```

The `Map` type expects the elements in A to be comparable. We do not assume any ordering on the elements in B, i.e., the elements from B are just inserted into a list.
For instance, the value

```
let ex = MMap (Map.ofList [("record",[50]);("ordering",[36;46;70])])
```

has type `multimap<string,int>` and represents two entries from the Index in the book HR:

`ordering`	`[36;46;70]`
`record`	`[50]`

You must either declare your own exception or use `failwith` to signal errors in your functions below.

Question 1.1

The students Sine, Hans, Grete and Peter are signed up for a number of courses as shown below:

`Grete`	
`Hans`	TOPS, HOPS
`Peter`	IFFY
`Sine`	HOPS, IFFY, BFNP

- Declare a value `studReg`, of type `multimap<string,string>`, with the course registrations for the four students above.
- Can you declare another value `studReg2`, of type `multimap<string,string>` with the same course registrations, but where the comparison `studReg = studReg2` returns `false`? You must motivate your answer with an example.

Question 1.2

We define the *canonical* representation of a multimap to be the representation where the elements in the value-lists are ordered.

- Declare a function

```
canonical: multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : comparison
```

where `canonical m` returns the canonical representation of `m`. For instance, the application `canonical studReg` will return a value corresponding to below mapping:

Grete	[]
Hans	["HOPS"';"TOPS"]
Peter	["IFFY"]
Sine	["BFNP"; "HOPS"; "IFFY"]

Hint: The extra type constraint on `'b` is because we now also require an ordering on the value-elements. The F# `Map` implementation already preserves the ordering of the key-elements.

- Declare a function

```
toOrderedList: multimap<'a,'b> -> ('a * 'b list) list
  when 'a : comparison and 'b : comparison
```

For instance, `let studRegOrdered = toOrderedList studReg` defines

```
val studRegOrdered : (string * string list) list =
  [("Grete", []); ("Hans", ["HOPS"; "TOPS"]); ("Peter", ["IFFY"]);
   ("Sine", ["BFNP"; "HOPS"; "IFFY"])]
```

Question 1.3

- Declare a function

```
newMultimap : unit -> multimap<'a,'b> when 'a : comparison
```

that returns a new empty multimap.

- Declare a function

```
sizeMultimap : multimap<'a,'b> -> int * int when 'a : comparison
```

that returns a pair with the number of keys as first component and the total number of value-elements as second component. For instance `let sizeStudReg = sizeMultimap studReg` gives the binding

```
val sizeStudReg : int * int = (4, 6)
```

Question 1.4

- Declare the function `addMultimap k v m` with type

```
addMultimap : 'a -> 'b -> multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : equality
```

that inserts the value `v` for key `k` in the map `m`. You must make sure that the value `v` does not exists more than once for key `k`. The result does not have to be in canonical order. Verify the following test cases returns true:

```
sizeMultimap (addMultimap "Sine" "BFNP" studReg) = (4,6)
sizeMultimap (addMultimap "Grete" "TIPS" studReg) = (4,7)
sizeMultimap (addMultimap "Pia" "" studReg) = (5,7)
```

- Declare the function `removeMultimap k vOpt m` with type

```
removeMultimap : 'a -> 'b option -> multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : equality
```

If `vOpt` is `None`, then remove the key `k` and all values v that `k` maps to, from the map. If `vOpt` is `Some v` then only remove the value `v` that the key `k` maps to. In case `vOpt` is `Some v`, then the set of keys in the map `m` is unchanged, that is, the key `k` may end up mapping to the empty list, like `"Grete"` above. Verify the following test cases returns true:

```
sizeMultimap (removeMultimap "Sine" None studReg) = (3,3)
sizeMultimap (removeMultimap "Sine" (Some "PLUR") studReg) = (4,6)
sizeMultimap (removeMultimap "Kenneth" (Some "BLOB") studReg) = (4,6)
sizeMultimap (removeMultimap "Peter" (Some "IFFY") studReg) = (4,5)
```

Question 1.5

Declare a function `mapMultimap f m` with type

```
mapMultimap : ('a -> 'b -> 'c) -> multimap<'a,'b> -> multimap<'a,'c>
  when 'a : comparison
```

that applies a function `f` on all elements in the multimap `m`. The function `f` takes both the key and value element as argument. The function `f` can be applied to the value elements in any order. For instance,

```
mapMultimap (fun k v -> v+"-F2015") studReg
```

returns a new multimap

```
MMap
  (map
     [("Grete", []); ("Hans", ["TOPS-F2015"; "HOPS-F2015"]);
      ("Peter", ["IFFY-F2015"]);
      ("Sine", ["HOPS-F2015"; "IFFY-F2015"; "BFNP-F2015"])])
```

Question 1.6

Declare a function `foldMultimap f s m` of type

```
foldMultimap : ('s -> 'k -> 't -> 's) -> 's -> multimap<'k,'t> -> 's
  when 'k : comparison
```

with similar semantics as `Map.fold`. For instance,

```
foldMultimap (fun acc k v -> String.length v + acc) 0 studReg
```

returns 24.

Question 2 (20%)

Consider the following F# declaration:

```
let rec f i j xs =
  if xs = [] then
    [i*j]
  else
    let (x::xs') = xs
    x*i :: f (i*j) (-1*j) xs'
```

The type of `f` is `int -> int -> int list -> int list`. The expression `f 10 1 [1 .. 9]` returns the value `[10; 20; -30; -40; 50; 60; -70; -80; 90; -10]`.

Question 2.1

- Describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.
- Explain why the F# compiler reports the warning below when compiling the function `f`.

  > *warning FS0025: Incomplete pattern matches on this expression. For example, the value '[]' may indicate a case not covered by the pattern(s).*

- Write a version of `f`, called `fMatch`, using pattern matching instead of the if-then-else and inner let expressions. Explain why the warning above disappears.

Question 2.2

The function `f` is not tail recursive. Declare a tail-recursive variant, `fA`, of `f`, or `fMatchA` of `fMatch`, using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the F# declaration:

```
let myFinSeq n m = seq { for i in [n .. m] do
                           yield [n .. i] }
```

of type `int -> int -> seq<int list>`

- Describe the sequence returned by `myFinSeq` when called with an arbitrary integer `n` and `m`.
- How many times does the number 12 occur in the value returned by `myFinSeq 10 14`?

Question 3.2

Declare a function `myFinSeq2 n m` of type
`myFinSeq2:int->int->seq<int>`
such that `myFinSeq2 n m` produces the same sequence numbers as `myFinSeq`. The difference is that the result sequence contains integer values only, i.e., no list elements. This is reflected in the type of `myFinSeq2`. For instance `myFinSeq2 3 6` returns the value

```
seq [3; 3; 4; 3; 4; 5; 3; 4; 5; 6]
```

Question 3.3

Consider the function `sum` to sum all elements in an integer list together with a big sequence, `seq4000`, and corresponding array, `array4000`.

```
let sum xs = List.fold (fun r x -> r+x) 0 xs
let seq4000 = myFinSeq 10 4000
let array4000 = Array.ofSeq seq4000
```

- How many lists does the value `array4000` contain?
- An array containing the sums of all lists can be computed sequentially

  ```
  let sums = Array.map sum array4000
  ```

  Give an alternative declaration of `sums` that computes the sums of the lists in parallel.

  Hint: You may use the `Array.Parallel` library as explained in Section 13.6 in the book HR.

Question 4 (30%)

We shall now consider *JSONlite*, a small subset of JSON (JavaScript Object Notation, `www.json.org`). The syntax of *JSONlite* is illustrated with the example below. The example contains the JSON *records*, `"Person1"`, `"Person2"` and `"Address"`. Each person record has a name, `"Name"` and an address, `"Address"`. The address record is the same for both persons. To avoid duplicating the address record we define a *label*, `Addr1`, pointing, `->`, at the address record. For the person, `Person2`, we can *reference*, `Ref`, the address record, `Addr1`. The below string representation is used in Question 4.2.

```
{
  "Person1" : {
    "Name" : "Hans Pedersen",
    "Address" : Addr1 -> {
      "Street" : "Hansedalen",
      "HouseNo" : "27"
    }
  },
  "Person2" : {
    "Name" : "Pia Pedersen",
    "Address" : ref Addr1
  }
}
```

Consider the F# declarations below. The declaration, `persons`, corresponds to the string representation above.

```
type JSONlite =
    Object of list<string * Value>
and Value =
    String of string
  | Record of JSONlite
  | Label of string * Value
  | Ref of string

let address = Object [("Street", String "Hansedalen");
                      ("HouseNo", String "27")]
let person1 = Object [("Name", String "Hans Pedersen");
                      ("Address", Label ("Addr1", Record address))]
let person2 = Object [("Name", String "Pia Pedersen");
                      ("Address", Ref "Addr1")]
let persons = Object [("Person1", Record person1);
                      ("Person2", Record person2)]
```

Question 4.1

Declare an F# value, `student`, of type `JSONlite` corresponding to the example below:

```
{
  "Name" : "Per Simonsen",
  "Field" : "BSWU",
  "Course" : {
    "BFNP" : "10",
    "BPRD" : "7"
  }
}
```

Question 4.2

Declare a function

`ppJSONlite:JSONlite->string`

that returns a string representation of the JSONlite value. For instance, `ppJSONlite persons`, returns a string corresponding to the first example above.

Hint: You can make use of the following template:

```
let nl = System.Environment.NewLine   // New line
let space n = String.replicate n " "  // Make n spaces
let ppQuote s = "\"" + s + "\""       // Put quotes around string s

let ppJSONlite json =
  let rec ppValue' indent = function
      String s -> ...
    | ...
  and ppJSONlite' indent = function
    Object xs -> ...
  ppJSONlite' 0 json
```

The variable `indent` is used to control indentation. You may find the function `String.concat sep xs` useful to concatenate a list of strings, `xs`, with a separator, `sep`.

Question 4.3

Declare a function

`buildEnv: JSONlite -> Map<string,Value>`

such that `buildEnv json` is an environment mapping labels declared in `json` to the corresponding JSON values. The function can assume that no label is defined multiple times. The application `buildEnv person1` returns the following environment:

```
map [("Addr1", Record (Object [("Street", String "Hansedalen");
                               ("HouseNo", String "27")]))]
```

Hint: You can make use of the following template:

```
let buildEnv json =
  let rec buildEnvValue env = function
      String s -> ...
    | ...
  and buildEnvJSONlite env = function
    Object xs -> ...
  buildEnvJSONlite Map.empty json
```

Question 4.4

Labels and references are not part of the JSON specification. Declare a function

`expandRef JSONlite->JSONlite`

that replaces all references to JSON values with the values themselves. You do this using the environment returned by `buildEnv`. The function can assume all references exists. For instance, if `persons` is bound to the value

```
Object
  [("Person1",...);
```

```
   ("Person2",
     Record (Object [("Name", String "Pia Pedersen");
                     ("Address", Ref "Addr1")]))]
```

then `expandRef persons` returns the following value:

```
Object
  [("Person1", ...);
   ("Person2",
    Record (Object [("Name", String "Pia Pedersen");
                    ("Address", Record (Object [("Street", String "Hansedalen");
                                                ("HouseNo", String "27")]))]))]
```

where `Ref "Addr"` has been expanded to `Record (Object [("Street",....`

BSWU BFNP **IT University, F2016**

Written exam, Functional Programming
Friday 3 June 2016

Version 1.01 of 2016-06-06

These exam questions comprise 9 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one ASCII file only, e.g., `bfnp2016.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `jun2016Snippets.txt` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3005303`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (30%)

We define a *multiset* as an unordered collection of elements that may contain duplicates. A multiset is a generalization of a set (HR page 104). For instance, $\{a, b, b, c\}$ and $\{a, b, c\}$ represents the same set $\{a, b, c\}$ but are different multisets. The *multiplicity* of an element is the number of instances of the element in the multiset. The multiplicity of a is 1, b is 2 and c is 1 for the multiset $\{a, b, b, c\}$. The two multisets $\{a, b, b, c\}$ and $\{a, b, c, b\}$ are equal as the order of the elements is irrelevant.
Formally a multiset may be defined as a tuple (A, m) where A is the underlying set of elements and m a (possibly partial) function $A \to \mathbb{N}_{\geq 1}$ (positive natural numbers), i.e., a partially defined multiplicity function ranging over the elements in A. If $m(a)$ is not defined for some $a \in A$, then the element a is not in the multiset represented by (A, m).
A multiset can be implemented using an F# map, `Map` (HR page 113). The map represents the partially defined multiplicity function over some type $'a$:

```
type Multiset<'a when 'a: comparison> = MSet of Map<'a, int>
```

For instance, the value

```
let ex = MSet (Map.ofList [("a",1);("b",2);("c",1)])
```

represents the multiset $\{a, b, b, c\}$ (also written $\{a, b, c, b\}$ etc.). The map must fulfil the invariant that for all keys in the map, the associated value is ≥ 1. The below value, for the multiset $\{b, b, c\}$, does **not** fulfil this invariant

```
let wrong = MSet (Map.ofList [("a",0);("b",2);("c",1)])
```

Question 1.1

The table below shows the result of rolling a dice 12 times, i.e., we got 1 eye 2 times, 3 eyes 5 times etc.

Num eyes	**1**	**2**	**3**	**4**	**5**	**6**
Result	2	1	5	0	2	2

- Declare a multiset value `diceSet`, representing the result of rolling the dice above, i.e., the multiset
  $\{1, 1, 2, 3, 3, 3, 3, 3, 5, 5, 6, 6\}$.
- What is the type of the value `diceSet`.
- Say you have an F# list of standard mathematical functions, e.g., `[System.Math.Sin; System.Math.Cos; System.Math.Sin]`. Are you able to represent this as a multiset with the representation chosen above, e.g., a value of type `Multiset<float->float>`? Explain your answer.

Question 1.2

- Declare a function

  ```
  newMultiset : unit -> Multiset<'a> when 'a : comparison
  ```

  that returns a new empty multiset.

- Declare a function

```
isEmpty : Multiset<'a> -> bool when 'a : comparison
```

that returns true if a multiset is the empty set. For instance `isEmpty(newMultiset())` returns true.

Question 1.3

- Declare a function

```
add : 'a -> Multiset<'a> -> Multiset<'a> when 'a : comparison
```

where `add k ms` returns the multiset `ms` with the element `k` added. For instance `add "a" ex` returns the value

```
MSet (map [("a", 2); ("b", 2); ("c", 1)])
```

- Declare a function

```
del : 'a -> Multiset<'a> -> Multiset<'a> when 'a : comparison
```

where `del k ms` returns the multiset `ms` with the element `k` deleted, if exists. If the element `k` does not exists, then the multiset `ms` is returned. For instance, `del "c" ex` returns the value

```
MSet (map [("a", 1); ("b", 2)])
```

Remember to fulfil the invariant of the values in the map explained above.

Question 1.4

- Declare a function

```
toList : Multiset<'a> -> 'a list when 'a : comparison
```

where `toList ms` returns the multiset as a list value. For instance `toList ex` may return the list

```
["a"; "b"; "b"; "c"]
```

- Declare a function

```
fromList : 'a list -> Multiset<'a> when 'a : comparison
```

where `fromList xs` returns the multiset corresponding to the elements in the list `xs`. For instance `fromList ["a";"a";"b"]` returns the multiset

```
MSet (map [("a",2);("b",1)])
```

Question 1.5

- Declare a function

```
map : ('a -> 'b) -> Multiset<'a> -> Multiset<'b>
      when 'a : comparison and 'b : comparison
```

  where `map f ms` returns the multiset where the function `f` has been applied on all elements in the multiset `ms`. For instance `map (fun (c:string) -> c.ToUpper()) ex` returns the multiset:

```
MSet (map [("A", 1); ("B", 2); ("C", 1)])
```

- Declare a function

```
fold : ('a -> 'b -> 'a) -> 'a -> Multiset<'b> -> 'a
       when 'b : comparison
```

  where `fold f a ms` returns the accumulated value obtained by applying the accumulator function `f` on all elements in the multiset `ms`, similar to `List.fold`. For instance `fold (fun acc e -> acc+e) "" ex` returns the string `"abbc"`.

Hint: You may use `List.map` and `List.fold`.

Question 1.6

- Declare a function

```
union : Multiset<'a> -> Multiset<'a> -> Multiset<'a>
        when 'a : comparison
```

  where `union ms1 ms2` returns the multiset where all elements in `ms2` have been added to the multiset `ms1`. For instance `union ex ex` returns the multiset:

```
MSet (map [("a", 2); ("b", 4); ("c", 2)])
```

- Declare a function

```
minus : Multiset<'a> -> Multiset<'a> -> Multiset<'a>
        when 'a : comparison
```

  where `minus ms1 ms2` returns the multiset where all elements in `ms2` have been deleted from `ms1`. For instance `minus ex ex` returns the empty multiset `MSet (map[])`.

Hint: You are allowed to use previously declared multiset functions.

Question 2 (20%)

Consider the following F# declarations of `f` and `g`:

```
let rec f n =
  if n < 10 then "f" + g (n+1) else "f"
and g n =
  if n < 10 then "g" + f (n+1) else "g"
```

The types of `f` and `g` are `int -> string`. The expression `f 0` returns the value `"fgfgfgfgfgf"`.

Question 2.1

- The result of evaluating `f 0` is a string that starts and ends with the letter `f`. Describe what arguments `n` to `f`, maybe infinitely many, that will generate a result string that starts with `f` and ends with `f`. The singleton string `"f"` by definition both starts and ends with `f`. Example strings are `"f"`, `"fgf"`, `"fgfgf"`, etc.
- Can you generate the result string `"gfgfgfg"` with the two functions `f` and `g` above?
- Can you for any argument to `f` start an infinite computation?

Question 2.2

The functions `f` and `g` are not tail recursive. Declare tail-recursive variants, `fA` of `f`, and `gA` of `g`, using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the following F# declaration:

```
let myFinSeq (n,m) = seq { for i in [0 .. n] do
                             yield! seq { for j in [0 .. m] do yield j }}
```

The type of `myFinSeq` is `int * int -> seq<int>`. The expression `myFinSeq(1,2)` returns the value `seq [0; 1; 2; 0; 1; 2]`.

- Describe the sequence returned by `myFinSeq` when called with arbitrary non-negative integers `n` and `m`.
- Can you for any arguments to `myFinSeq` generate the following value `seq [0; 1; 2; 0; 1; 2; 0; 1]`. Explain your answer.

Question 3.2

Consider the output below of calling a F# function `myFinSeq2 (n,m)` of type `int * int -> seq<int*seq<int>>` with different values of `n` and `m`:

- `myFinSeq2 (0,0)` returns the value `seq [(0, seq [0])]`
- `myFinSeq2 (1,1)` returns the value `seq [(0, seq [0; 1]); (1, seq [0; 1])]`
- `myFinSeq2 (1,2)` returns the value `seq [(0, seq [0; 1; 2]); (1, seq [0; 1; 2])]`
- `myFinSeq2 (2,1)` returns the value `seq [(0, seq [0; 1]); (1, seq [0; 1]); (2, seq [0; 1])]`

Declare the F# function `myFinSeq2` producing the output with the given arguments above. You can assume `n` and `m` are non-negative.

Question 4 (30%)

We shall now consider *CALClite*, a simple spreadsheet system. The spreadsheet is defined by the following type declarations:

```
type Row = int
type Col = char
type CellAddr = Row * Col
type ArithOp = Add | Sub | Mul | Div
type RangeOp = Sum | Count
type CellDef =
    FCst of float
  | SCst of string
  | Ref of CellAddr
  | RangeOp of CellAddr * CellAddr * RangeOp
  | ArithOp of CellDef * ArithOp * CellDef
type CellValue =
    S of string
  | F of float
type Sheet = Map<CellAddr,CellDef>
```

A *cell address* (`CellAddr`) is defined as a *row*- and *column*-index, written `A1` for column `A` and row `1`. We assume columns are in the interval `'A' ... 'Z'` and rows greater or equal to 1. A *cell definition* (`CellDef`) can be a float constant (`FCst`), a string constant (`SCst`), a reference to another cell (`Ref`), an operation on a range (`RangeOp`) or an binary arithmetic operation (`ArithOp`). A range is defined as all cells within the rectangle defined by the top left cell address and the bottom right cell address. The only operations allowed on a range are summing all cells (`Sum`) and counting the number of cells (`Count`). Evaluating a cell definition (`CellDef`) will either result in a string or a float value represented with the type `CellValue`. A spreadsheet, type `Sheet`, is defined as a mapping from cell adresses to cell definitions.
The result of rolling 12 dice from question 1.1 is repeated below:

```
   |        A |        B |        C |        D |        E |        F |        G |        H
---+----------+----------+----------+----------+----------+----------+----------+---------
 1 |    #EYES |        1 |        2 |        3 |        4 |        5 |        6 |    Total
 2 |   RESULT |     2.00 |     1.00 |     5.00 |     0.00 |     2.00 |     2.00 |    12.00
 3 |      PCT |    16.67 |     8.33 |    41.67 |     0.00 |    16.67 |    16.67 |   100.00
```

The cell `H2` is the sum of the range `B2` to `G2`. The cell `B3` is the percentage number of 1's, i.e., `B2/H2*100.0`. The cells `C3`, ..., `H3` are defined similar to `B3`. The definition of the sheet, called `dice`, is as follows:

```
let header = [((1,'A'),SCst "#EYES");((1,'B'),SCst "1");((1,'C'),SCst "2");
              ((1,'D'),SCst "3");((1,'E'),SCst "4");((1,'F'),SCst "5");
              ((1,'G'),SCst "6");((1,'H'),SCst "Total")]
let result = [((2,'A'),SCst "RESULT");((2,'B'),FCst 2.0);((2,'C'),FCst 1.0);
              ((2,'D'),FCst 5.0);((2,'E'),FCst 0.0);((2,'F'),FCst 2.0);
              ((2,'G'),FCst 2.0);((2,'H'),RangeOp((2,'B'),(2,'G'),Sum))]
let calcPct col = ArithOp(FCst 100.0, Mul, ArithOp(Ref(2,col),Div,Ref(2,'H')))
let pct = [((3,'A'),SCst "PCT");((3,'B'),calcPct 'B');((3,'C'),calcPct 'C');
           ((3,'D'),calcPct 'D');((3,'E'),calcPct 'E');((3,'F'),calcPct 'F');
           ((3,'G'),calcPct 'G');((3,'H'),calcPct 'H')]
let dice = Map.ofList (header @ result @ pct)
```

Question 4.1

Declare an F# value, `heights`, of type `Sheet` corresponding to the sheet below:

```
    |        B |        C
----+----------+---------
  4 |     NAME |   HEIGHT
  5 |     Hans |   167.40
  6 |    Trine |   162.30
  7 |    Peter |   179.70
  8 |          |
  9 |     3.00 |   169.80
```

The cells `B4`, `B5`, `B6`, `B7` and `C4` are constant strings. The cells `C5`, `C6` and `C7` are constant floats. The cell `B9` is the count of cells in the range defined by the cell addresses `B5` and `B7`. The cell `C9` is defined as the sum of the range `C5` and `C7` divided by cell `B9`, i.e., the average height of the three persons.

Question 4.2

In order to evaluate a cell definition we need to evaluate range operations `Sum` and `Count` and to evaluate the arithmetic operations `Sum`, `Div`, `Sub` and `Mul`. The operation `Count` works on both float and string values because we are only counting the number of values. The other operations only work on float values. We use the function `getF` of type `CellValue -> float` to ensure a cell value is a float:

```
let getF = function
    F f -> f
  | S s -> failwith "getF: expecting a float but got a string"
```

Declare the following F# functions using `getF`:

1. `evalRangeOp xs op` of type `CellValue list -> RangeOp -> float`. For instance
   - `evalRangeOp [F 33.0; F 32.0] Sum` returns 65.0
   - `evalRangeOp [] Sum` returns 0.0
   - `evalRangeOp [F 23.0; S "Hans"] Sum` throws `System.Exception`
   - `evalRangeOp [F 23.0; S "Hans"] Count` returns 2.0
2. `evalArithOp v1 v2 op` of type `CellValue -> CellValue -> ArithOp -> float`. For instance
   - `evalArithOp (F 33.0) (F 32.0) Sub` returns 1.0
   - `evalArithOp (S "Hans") (F 1.0) Add` throws `System.Exception`

Question 4.3

In order to print a spreadsheet we need to evaluate all cells. This is done by two mutually recursive functions

- `evalValue v sheet` of type `CellDef -> Sheet -> CellValue`
- `evalCell ca sheet` of type `CellAddr -> Sheet -> CellValue`

Declare the two F# functions above. You can assume the sheet contains no cyclic cell definitions. You may use the following template

```
let rec evalValue v sheet =
  match v with
    FCst f -> F f
  | SCst s -> ..
  | Ref ca -> ..
  | RangeOp ((r1,c1),(r2,c2),op) -> ..
  | ArithOp (v1,op,v2) -> ..
and evalCell ca sheet =
  match Map.tryFind ca sheet with
    None -> S "" // We define an empty cell to be the empty string value.
  | Some v -> evalValue v sheet
```

For instance `evalCell (3,'G') dice` returns the cell value `F 16.67`.

Question 4.4

Declare a F# function `ppBoard sheet` of type `Sheet -> string` that returns a string similar to the layout used for `dice` and `heights` above. For instance, `ppBoard dice` returns the string

```
    |        A |        B |        C |        D |        E |        F |        G |        H
----+----------+----------+----------+----------+----------+----------+----------+---------
  1 |    #EYES |        1 |        2 |        3 |        4 |        5 |        6 |    Total
  2 |   RESULT |     2.00 |     1.00 |     5.00 |     0.00 |     2.00 |     2.00 |    12.00
  3 |      PCT |    16.67 |     8.33 |    41.67 |     0.00 |    16.67 |    16.67 |   100.00
```

You must include the actual output from the `ppBoard` function on the `dice` example in order to obtain full points.

BSWU BFNP–1413003U **IT University, F2017**

Written exam, Functional Programming
Tuesday June 6, 2017

Version 1.03 of 2017-06-25

These exam questions comprise 9 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one ASCII file only, e.g., `bfnp2017.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `jun2017Snippets.txt` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3016512`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (30%)

We define a *priority set* as a prioritised collection of elements that may not contain duplicates. A priority set is a combination of a set (HR page 104) and a priority queue. An element can only be in the set once (set property) but elements are prioritised according to insertion order within the set (priority queue).
We can add and remove elements and ask for membership as with ordinary sets. What is unusual is that we keep track of the order in which elements are inserted, so one can ask for eg. the oldest element in the priority set, the one inserted before all the others.
For instance, inserting the elements a, b and c in an empty priority set in that order results in the priority set $\{a^1, b^2, c^3\}$. We define the *priority number* as an unique number representing the priority given to an element in the set. The priority number is annotated in superscript on each element. The element a is inserted first and hence gets priority number 1, b is inserted second and gets priority number 2 etc.
The smallest possible priority number is defined to be 1. Inserting the element b a second time will not change the priority set and the element b still has the priority number 2. We define the *first element* in the priority set as the element with lowest priority number, i.e., 1. Element a is the first element in the example above.
The unique priority number assigned an element may change as the set changes. For instance, removing element a from the priority set above results in a new priority set where the priority number for b and c has changed: $\{b^1, c^2\}$; b is then the first element in the set .
A priority set can be implemented using a simple F# list, `List` (HR page 93). We do not explicitly store the priority number of an element as it can be calculated from the position of the element in the list. The element with priority number 1 is the first element in the list.

```
type PrioritySet<'a when 'a: equality> = PrioritySet of List<'a>
```

The example above is declared as

```
let psEx = PrioritySet ["a";"b";"c"]
```

Question 1.1

Consider the following elements and assume they are inserted in an empty priority set in this order: `"a"`, `"q"`, `"a"`, `"b"`, `"b"`, `"q"`, `"d"`, `"a"`.

- Declare a value `priSetEx`, being the result of inserting the elements above according to the definition of a priority set.
- What is the type of the value `priSetEx`.
- Declare a value `empty` representing an empty priority set, i.e., priority set with no elements.

Question 1.2

- Declare a function

  ```
  isEmpty : PrioritySet<'a> -> bool when 'a : equality
  ```

  that returns true if a priority set is the empty set. For instance `isEmpty(empty)` returns true. The value `empty` is defined above.
- The *size* of a priority set is the number of elements in the set. Declare a function

```
size : PrioritySet<'a> -> int when 'a : equality
```

that returns the size of a priority set. For instance, `size psEx` returns 3.

- Declare a function `contains` e ps of type

```
contains : 'a -> PrioritySet<'a> -> bool when 'a : equality
```

that returns true if the priority set ps contains an element e. For instance `contains "b" psEx` returns true.

- Declare a function `getPN` e ps of type

```
getPN : 'a -> PrioritySet<'a> -> int when 'a : equality
```

that returns the priority number of element e if exists in priority set ps. Otherwise raises an error exception (`failwith`). For instance `getPN "a" psEx` returns 1.

Question 1.3

- Declare a function `remove` e ps of type

```
remove : 'a -> PrioritySet<'a> -> PrioritySet<'a> when 'a : equality
```

that removes element e from the priority set ps and returns a new priority set. Nothing changes if e does not exists in ps. For instance, `remove "b" psEx` returns the priority set `PrioritySet ["a";"c"]`.

- Declare a function

```
add : 'a -> PrioritySet<'a> -> PrioritySet<'a> when 'a : equality
```

where `add e ps` returns the priority set `ps` with the element `e` added with lowest priority (highest priority number) unless already in the set `ps`. Adding element *h* to priority set $\{a^1, b^2, c^3\}$ gives the priority set $\{a^1, b^2, c^3, h^4\}$. Adding element *b* to $\{a^1, b^2, c^3\}$ gives the unchanged priority set $\{a^1, b^2, c^3\}$.

Question 1.4

- Declare a function `map` *f ps* of type

```
map : ('a -> 'b) -> PrioritySet<'a> -> PrioritySet<'b>
  when 'a : equality and 'b : equality
```

where `map` *f ps* returns the priority set where the function *f* has been applied on all elements in the priority set *ps* in order of priority number. For instance `map (fun (c:string) -> c.ToUpper()) psEx` returns the priority set value `PrioritySet ["A";"B";"C"]`.

- Declare a function `cp` of type

```
cp : PrioritySet<'a> -> PrioritySet<'b> -> PrioritySet<'a * 'b>
       when 'a : equality and 'b : equality
```

where `cp` *ps1 ps2* returns the cartesian product of *ps1* and *ps2*. The result set is generated from *ps1* and *ps2* in order of priority number of *ps1* first and then *ps2*. For instance the cartesian product of $\{A^1, B^2, C^3\}$ and $\{h^1, i^2\}$ is $\{(A,h)^1, (A,i)^2, (B,h)^3, (B,i)^4, (C,h)^5, (C,i)^6\}$. A cartesian product involving an empty set is the empty set, eg. `cp psEx empty` is the empty set.

Question 2 (20%)

Consider the F# declaration:

```
let f curRow =
  let rec f' = function
      []              -> []
    | [_]             -> [1]
    | xs              -> let (x1::x2::xs) = xs
                         x1 + x2 :: f' (x2::xs)
  (1 :: f' curRow)
```

with type `int list -> int list`.

Question 2.1

Describe what `f` computes given the examples below:

- `f [1]` gives `[1; 1]`
- `f [1; 1]` gives `[1; 2; 1]`
- `f [1; 2; 1]` gives `[1; 3; 3; 1]`
- `f [1; 3; 3; 1]` gives `[1; 4; 6; 4; 1]`

Question 2.2

Compiling the function `f` and `f'` above gives the warning:

> *warning FS0025: Incomplete pattern matches on this expression. For example, the value '[_]' may indicate a case not covered by the pattern(s).*

Write a version of `f` and `f'`, called `fMatch` and `fMatch'`, without this warning. Explain why the warning disappears.

Question 2.3

The function `f'` is not tail recursive. Write a tail-recursive version, `fA'` of `f'` (or `fMatch'`) using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the F# declaration:

```
let mySeq s1 s2 =
  seq { for e1 in s1 do
          for e2 in s2 do
            yield! [e1;e2] }
```

of type `seq<'a> -> seq<'a> -> seq<'a>`.

- Describe the sequence returned by `mySeq` when called with arbitrary sequences `s1` and `s2`.
- Can you for any arguments to `mySeq` generate the following value `seq ['A';'D'; 'A';'E'; 'A';'F'; 'B';'D'; 'B';'E'; 'B';'F']`

Question 3.2

Declare a function `mySeq2 s1 s2` of type `seq<'a> -> seq<'b> -> seq<'a * 'b>` such that the cartesian product of `s1` and `s2` is returned. For instance `mySeq2 [1;2] ['A';'B';'C']` gives the result `seq [(1,'A'); (1,'B'); (1,'C'); (2,'A'); (2,'B'); (2,'C')]`.

Question 3.3

Declare a function `mySeq3` of type `int -> seq<int>`, such that `mySeq3 n` produces the infinite sequence $\mathtt{n}^2 - \mathtt{n} * i$ for $i >= 0$. The identifier i is the index of the element in the sequence.
Hint: Consider using `Seq.initInfinite`.

Question 4 (30%)

We shall now consider an internal DSL called *DataSpec* to be used for generating test data. With *DataSpec* one can create a specification of a process to generate an arbitrary number of data records. The DSL for *DataSpec* uses the following F# type declaration:

```
type DataSpec =
    RangeInt of int * int
  | ChoiceString of string list
  | StringSeq of string
  | Pair of DataSpec * DataSpec
  | Repeat of int * DataSpec
```

- `RangeInt` (b, e) specifies the generation of a random number in the integer interval $[b \ldots e]$.
- `ChoiceString`$[s_1, \ldots, s_n]$ specifies the choice of one string among n different strings.
- `StringSeq` s specifies the generation of an unique string by appending an unique number to each string generated. E.g., `StringSeq "A"` may generate the strings `A1`, `A2`, `A3`, ...
- `Pair` (ds_1, ds_2) specifies the generation of a pair of values.
- `Repeat` (n, ds) specifies the generation of a collection $v_1, \ldots, v_n$ of n values where v_i is the i'th value generated by ds, for $1 <= i <= n$. For instance, `Repeat (3,RangeInt(1,5))` may generate the following 3 values `5`, `1` and `4`, i.e., 3 arbitrary values between 1 and 5.

Consider the cash register example (HR page 83), below expressed as an F# value:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5))]
```

A specification, of type `DataSpec`, for a similar register using the DSL above is

```
let reg =
  Repeat(3,Pair(StringSeq "a",
                Pair(ChoiceString["cheese";"herring";"soft drink"],
                     RangeInt(1,100))))
```

The *article codes* are generated as a string sequence, e.g., `a1`, `a2` etc. The *price* is an arbitrary number between 1 and 100. We have only three possible *article names*: cheese, herring and soft drink.

Question 4.1

Declare an F# value `pur`, of type `DataSpec` that is a specification for a purchase like below:

```
let pur = [(3,"a2"); (1,"a1")]
```

The first element in each pair is the *number of pieces* which we choose to be an arbitrary integer between 1 and 10 (`RangeInt`). The second element of each pair is an *article code* specified as a sequence of strings (`StringSeq`). Use the constructors `Pair` and `Repeat` to generate two pairs.

Question 4.2

Declare a function `genValue` *ds* of type

```
genValue : DataSpec -> string
```

such that `genValue` returns a string representation of the values generated given the specification *ds*. Given the randomness built into the data generator, the result of `genValue reg` could be:

```
"[(a1,(cheese,69));(a2,(herring,94));(a3,(cheese,50))]"
```

The randomness does not prohibit the same *article name* to be used several times, e.g., cheese.
Hint: You need a way to generate random numbers to handle `RangeInt` and `ChoiceString`. The function `next` (i_1, i_2) below returns a random integer in the interval $[i_1, \ldots, i_2[$ using the random generator `rand`. You also need a way to generate unique numbers for `StringSeq`. The function `numGen` () below returns a new unique number each time it is called. You may also use the template for `genValue` below:

```
let rand = System.Random()
let next(i1,i2) = rand.Next(i1,i2)
let numGen =
  let n = ref 0
  fun () -> n := !n+1; !n

let rec genValue = function
    RangeInt(i1,i2) -> next(i1,i2).ToString()
  | ChoiceString xs -> ...
  | ...
```

Question 4.3

The declaration of the register `reg` and purchase `pur` above is independent. This means that nothing prevents a purchase from containing *article codes* that do not exists in the register. We fix this by extending the DSL with a way to *label* generated data and the ability to *pick* from this data later. The new type for `DataSpec` is

```
type DataSpec =
    RangeInt of int * int
  | ...
  | Pick of string
  | Label of string * DataSpec
```

We can then define modified versions of the register and purchase specifications:

```
let reg2 = Repeat(3,Pair(Label("articleCode",StringSeq "a"),
                         Pair(ChoiceString["cheese";"herring";"soft drink"],
                              RangeInt(1,100))))
let pur2 = Repeat(2,Pair(RangeInt(1,10), Pick "articleCode"))
```

Every time we generate a new article code using `StringSeq` under the `Label` we add the result value to an environment under the name `"articleCode"`. With the specification for `reg2` above, we will end with an environment mapping the string `"articleCode"` to three possible values because we repeat 3 times. The environment can thus be defined as a map from strings to a list of strings:

```
type Env = Map<string,string list>
```

Declare a function `addToEnv` *s v dEnv* of type

```
addToEnv : string -> string -> Env -> Env
```

that adds the value *v* to the environment *dEnv* under the name *s*. If *s* already exists in *dEnv*, then add *v* to the list of values under *s* already in *dEnv*. For instance, `addToEnv "x" "42" env`, where `env = map [("x", ["43"])]` returns the new environment `map [("x", ["42"; "43"])]`.
Declare a function `pickFromEnv` *s dEnv* of type

```
pickFromEnv : string -> Env -> string
```

that picks an arbitrary value *v* from the environment *dEnv* under the name *s*. Use helper function `next` to pick an arbitrary value. In case *s* does not exists in the environment then raise an exception (`failwith`). For instance, `pickFromEnv "x" env` could return the string `"43"`.

Question 4.4

Declare a function `genValue` *dEnv ds* of type

```
genValue : Env -> DataSpec -> string * Env
```

that returns a string representation of the values generated and a new environment given the input environment *dEnv* and specification *ds*. For instance,

```
let (v,dEnv) = genValue Map.empty reg2
```

may return

```
val v : string = "[(a18,(herring,44));(a19,(herring,7));(a20,(cheese,13))]"
val dEnv : Env = map [("articleCode", ["a20"; "a19"; "a18"])]
```

Applying `genValue dEnv pur2` may then return

```
("[(8,a20);(5,a19)]", map [("articleCode", ["a20"; "a19"; "a18"])])
```

Two article codes `a20` and `a19` have been *picked* from the environment `dEnv` by the `Pick` constructor.
Hint: You can use the template below for `genValue`:

```
let rec genValue dEnv = function
    RangeInt(i1,i2) -> (next(i1,i2).ToString(),dEnv)
  | ChoiceString xs -> let idx = next(0,List.length xs - 1)
                       (...,dEnv)
  | StringSeq s -> ...
  | Pair(ds1,ds2) ->
    let (v1',dEnv1) = genValue dEnv ds1
    let (v2',dEnv2) = genValue dEnv1 ds2
    ...
  | Repeat (n,ds) -> ...
  | Pick s -> (pickFromEnv ..., ...)
  | Label (s,ds) ->
    let (v',dEnv') = genValue dEnv ds
    (v', ...)
```

BSWU BFNP–1413003U **IT University, F2018**

Written exam, Functional Programming
Thursday May 31, 2018

Version 1.00 of 2018-05-28

These exam questions comprise 8 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one ASCII file only, e.g., `bfnp2018.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `bfnp2018Snippets.fsx` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3017414`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

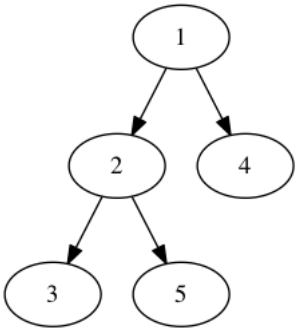
Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

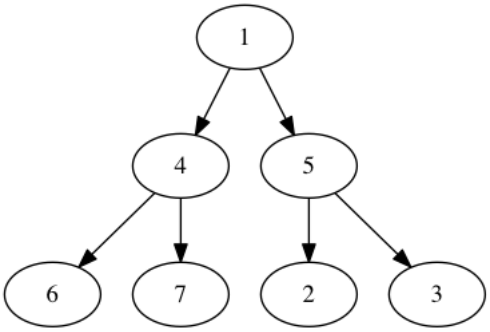
I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (25%)

We define a *heap* as a binary tree whose nodes fulfil the *heap property*. The heap property means that the value stored in a node must be less than or equal to the values stored in the child nodes. The heap property is fulfilled in example 1 below, but **not** in example 2 where 5 is greater than 2 and 3.

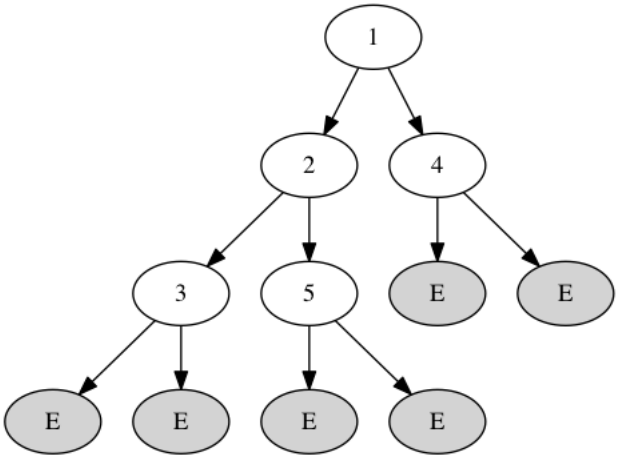


Example 1: The heap property is fulfilled.



Example 2: The heap property is not fulfilled.

In this assignment you will implement a heap based on a binary tree structure. Example 3 below shows the tree from Example 1 with empty nodes E added. The empty nodes are grey.



Example 3: The tree from example 1 with empty nodes added.

We represent the heap with the below polymorphic datatype where empty nodes are represented by `EmptyHP`.

```
type Heap<'a when 'a: equality> =
  | EmptyHP
  | HP of 'a * Heap<'a> * Heap<'a>
```

Question 1.1

- Declare a value `ex3` representing the binary tree shown in example 3 above. You may use the template:

```
let ex3 = HP(1,HP(2,HP(...
              HP(4,...
```

- Write the type of the value `ex3`. Explain why the type is either monomorphic or polymorphic.
- Declare a value `empty` representing an empty heap, i.e. a binary tree with only one empty root node. The type of the empty value is `empty : Heap<'a> when 'a : equality`.

- Declare an F# exception named `HeapError` that can be used to signal an error condition from a function on heaps. The exception should carry a string to be used to describe the error.

Question 1.2

- Declare a function

```
isEmpty : Heap<'a> -> bool when 'a : equality
```

  that returns true if a heap is the empty heap. For instance `isEmpty empty` returns true. The value `empty` is defined above.

- The *size h* of a heap *h* is the number of non–empty nodes in the binary tree. Declare a function

```
size : Heap<'a> -> int when 'a : equality
```

  that returns the size of a heap. For instance, `size ex3` returns 5.

- Declare a function `find` h of type

```
find : Heap<'a> -> 'a when 'a : equality
```

  that returns the minimum value in a non–empty heap, i.e. the root value. For instance `find ex3` returns 1.

- Declare a function `chkHeapProperty` h of type

```
chkHeapProperty : Heap<'a> -> bool when 'a : comparison
```

  that returns true if the heap h fulfils the heap property and otherwise false. The empty heap by definition fulfils the heap property. For instance `chkHeapProperty ex3` returns true.

Question 1.3

- Declare a function `map` *f h* of type

```
map : ('a -> 'b) -> Heap<'a> -> Heap<'b>
  when 'a : equality and 'b : equality
```

  where `map` *f h* returns the heap where the function *f* has been applied on all values in the heap *h*. You decide, but must explain, what order the function *f* is applied to the values in the heap. For instance `map ((+)1) ex3` returns the heap with all values in `ex3` increased by one.

- The heap `ex3` fulfils the heap property. Give an example of a function *f* such that mapping *f* on all values in `ex3` gives a new heap that does not fulfil the heap property. Given your definition of *f*, show that `chkHeapProperty (map f ex3)` returns false.

Question 2 (30%)

We shall now consider a binary *divide–and–conquer* algorithm. We will use the algorithm to implement mergesort. You do not need to know how mergesort works to do this.

Question 2.1

- Declare a function `genRandoms` *n* of type `int -> int[]` that returns an array of *n* random integers. The random integers are larger than or equal to 1 and less than 10000. For instance, `genRandoms 4` may return `[|8803;8686;2936;2521|]`.

  Hint: You can use below to define a generator `random` of type `unit -> int` to generate the random numbers.

```
let random =
  let rnd = System.Random()
  fun () -> rnd.Next(1,10000)
```

- Declare a function `genRandomsP` *n* of type `int -> int[]` that is similar to `genRandom` except that the numbers are generated in parallel to speed up the process.

  Hint: You may use the `Array.Parallel` library as explained in Section 13.6 in the book HR.

Question 2.2

Mergesort consists of three separate steps: splitting the remaining unsorted elements in two halves (`split`), identifying when the list has at most one element, and thus is trivially sorted (`indivisible`) and merging two already sorted lists together (`merge`). We implement each step below.

- Declare a function `split` *xs* of type `'a list -> 'a list * 'a list` which takes a list *xs*, say $[e_1, \ldots, e_n]$ and returns two lists with half elements in each: $([e_1, \ldots, e_{n/2}], [e_{n/2+1}, \ldots, e_n])$. For instance `split [22;746;931;975;200]` returns `([22;746],[931;975;200])`. Define and explain at least three relevant test cases.

- Declare a function `indivisible` *xs* of type `'a list -> bool`. The function returns true if the list is either empty or contains one element only, i.e. the list is trivially sorted; otherwise the function returns false. For instance `indivisible [23;34;45]` returns false.

- Declare a function `merge` *xs ys* of type

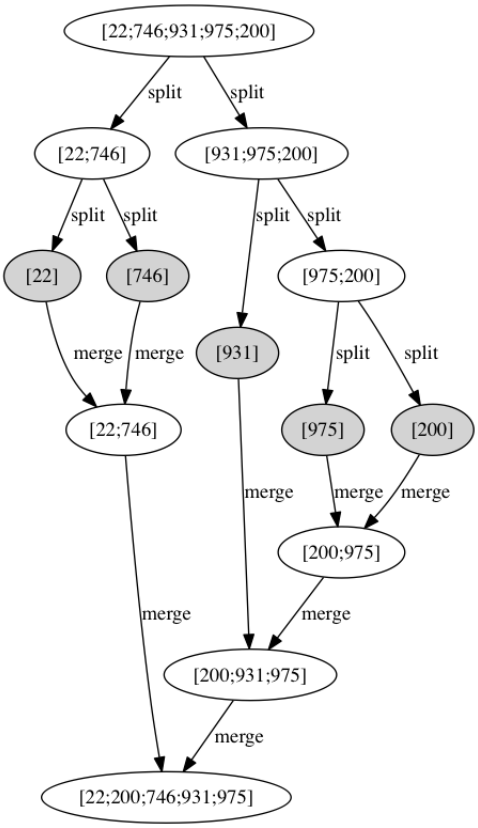
```
merge : 'a list * 'a list -> 'a list when 'a : comparison
```

  that returns the sorted merged list of *xs* and *ys*. The function `merge` can assume the two lists are sorted and does not have to check for that. For instance `merge ([1;3;4;5],[1;2;7;9])` returns `[1;1;2;3;4;5;7;9]`. Define and explain at least three relevant test cases.

Question 2.3

The process of solving a problem *p* using binary divide–and–conquer is to repeatedly divide the problem *p* into problems of half size until the divided problems are indivisible and trivially solved. The divided solutions are then merged together until the entire problem *p* is solved.

In the case of mergesort the problem *p* is the list of elements to sort. Dividing the problem is to use `split` to make two new sublists to sort. Merging two solved problems, i.e. sorted lists, is done by `merge`. Example 4 below shows the process for sorting the list `[22;746;931;975;200]`.



Example 4: Mergesort on [22;746;931;975;200]. Grey nodes are indivisible.

- Declare a function `divideAndConquer` *split merge indivisible p* of type

```
divideAndConquer : ('a -> 'a * 'a) -> ('a * 'a -> 'a)
                     -> ('a -> bool) -> 'a -> 'a
```

that implements the divide and conquer process. You may, but don't have to, use the template below.

```
let divideAndConquer split merge indivisible p =
  let rec dc p =
    if indivisible p
      then ...
      else ...
  dc p
```

Executing `divideAndConquer split merge indivisible [22;746;931;975;200]` should give give the result `[22;200;746;931;975]`.

Question 3 (20%)

In this question we work with sequences as covered in Chapter 11 in HR.

Question 3.1

- Declare the infinite sequence `triNum` of *triangular numbers*. The type of `triNum` is `seq<int>`. The triangular numbers are defined as $x_n = \frac{n(n+1)}{2}$ where x_n is the nth element in the sequence. The first element has index $n = 0$.

  Hint: You may use `Seq.initInfinite`. The sequence is `seq [0;1;3;6;...]`.

- Declare a cached version `triNumC` of `triNum` such that already computed elements are cached. The type of `triNumC` is `seq<int>`.

Question 3.2

The function `filterOddIndex` s filters out all elements e_i of the sequence s where i is odd. The function declaration is based on the assumption that the input sequence s is infinite but unfortunately goes into an infinite loop. For instance `filterOddIndex triNum` never terminates.

```
let rec filterOddIndex s =
  Seq.append (Seq.singleton (Seq.item 0 s))
             (filterOddIndex (Seq.skip 2 s))
```

- Declare your own version `myFilterOddIndex` similar to `filterOddIndex` except that it does not enter an infinite loop but returns the intended sequence.

  Hint: You may be inspired by Section 11.3 in HR. The sequence for `myFilterOddIndex triNum` is `seq [0;3;10;21;...]`.

Question 3.3

The sequence library `Seq` contains a number of functions to manipulate sequences, see Table 11.1 in HR. One such function is `Seq.zip` s_1 s_2 of type

```
(seq<'a> -> seq<'b> -> seq<'a * 'b>)
```

For instance executing `Seq.zip triNum triNum` returns the value

```
seq [(0, 0); (1, 1); (3, 3); (6, 6); ...]
```

- Declare a function `seqZip` of type

  ```
  (seq<'a> -> seq<'b> -> seq<'a * 'b>)
  ```

  that works the same as `Seq.zip`. You are not allowed to use `Seq.zip` but should implement `seqZip` using *sequence expressions* as explained in Section 11.6 in HR. You may use the template below.

  ```
  let rec zipSeq s1 s2 =
    seq {let e1 = Seq.item 0 s1
         let e2 = Seq.item 0 s2
         ... }
  ```

Question 4 (25%)

We now consider an internal domain specific language (DSL) called *Fig* to be used for specifying figures constructed from *basic figures*, that is, circles and lines. The DSL contains constructors for forming a collection of figures (constructor `Combine`), for specifying a move of a figure by a given offset (constructor `Move`) and for naming and referencing figures (constructors `Label` and `Ref`).

```
exception FigError of string
type Point = P of double * double
type Fig =
    Circle of Point * double
  | Line of Point * Point
  | Move of double * double * Fig
  | Combine of Fig list
  | Label of string * Fig
  | Ref of string
```

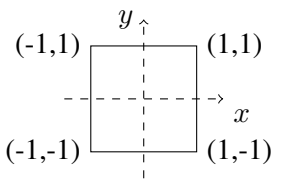
- The exception `FigError` represents an error condition from a function in the library.
- The type `Point` represents a point (x, y) in the two dimensional space.
- The type `Fig` represents the DSL for figures
  - `Circle` (p, r) is the circle with center p and radius r.
  - `Line` (p_1, p_2) is the line between the two points p_1 and p_2
  - `Move` $(d_x, d_y$,*fig*$)$ denotes the figure obtained from *fig* by moving the figures contained in *fig* as specified by d_x and d_y.
  - `Combine` *figs* is the collection of figures in *figs*.
  - `Label` (*lab*,*fig*) gives the *fig* a name *lab*. We assume *fig* does not contain references (`Ref`) such that cyclic structures are avoided.
  - `Ref` *lab* references the figure with name *lab* assuming it exists.

The example `figEx01` represents a figure consisting of a circle with center (1.0,1.0) and radius 2.0 together with a line between the points (0.0,0.0) and (1.0,1.0).

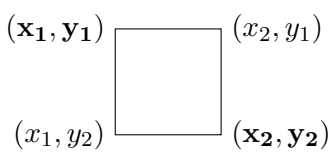
```
let figEx01 = Combine [Circle(P(1.0,1.0),2.0);Line(P(0.0,0.0),P(1.0,1.0))]
```

Question 4.1

- Declare an F# value `rectEx` of type `Fig` that represents a rectangle with the four points (-1,1) , (1,1), (1,-1) and (-1,-1) as shown in the figure to the right.



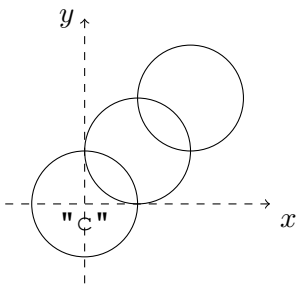
- Declare an F# function `rect` (x_1, y_1) (x_2, y_2) of type `double * double -> double * double -> Fig`, that given two orthogonal coordinates as shown in the figure to the right returns a figure representing the rectangle by its four sides. For instance `rect (-2.0,1.0) (1.0,-1.0)` returns




```
Combine [Line(P(-2.0,1.0),P(1.0,1.0));Line(P(1.0,1.0),P(1.0,-1.0));
         Line(P(1.0,-1.0),P(-2.0,-1.0));Line(P(-2.0,-1.0),P(-2.0,1.0))]
```

Question 4.2

Consider the F# value `figEx02` consisting of a labeled circle `"c"` which is referenced twice. The referenced circles are moved such that we obtain a figure like the one to the right.




```
let figEx02 =
  Combine [Label("c",Circle(P(0.0,0.0),1.0));
           Move(1.0,1.0,Ref "c");
           Move(2.0,2.0,Ref "c")]
```

- Declare an F# function `buildEnv` *fig* of type `Fig -> Map<string,Fig>` that traverses the figure *fig* and builds an environment mapping labels to figures. For instance

  ```
  let envEx02 = buildEnv figEx02
  ```

  binds `envEx02` to the value

  ```
  map [("c",Circle(P(0.0,0.0),1.0))]
  ```

Question 4.3

Given a figure *fig* and an environment *env* mapping labels to figures, we can substitute referenced figures with the actual figures.

- Declare an F# function `substFigRefs` *env fig* of type `Map<string,Fig> -> Fig -> Fig` that substitues all references with actual figures. As we substitute all references there is no need to keep the labels either. The result figure should therefore not contain any references `Ref` or labels `Label`. For instance

  ```
  let substEx02 = substFigRefs envEx02 figEx02
  ```

  binds `substEx02` to the value

  ```
  Combine
    [Circle(P(0.0,0.0),1.0);
     Move(1.0,1.0,Circle(P(0.0,0.0),1.0));
     Move(2.0,2.0,Circle(P(0.0,0.0),1.0))]
  ```

Question 4.4

We now assume that figures do not contain labels and references. For such figures, we can remove the `Move` constructors by updating the positions of the circles and lines. We thus obtain a figure consisting of `Combine`, `Circle` and `Line` constructors only.

- Declare an F# function `reduceMove` *fig* of type `Fig -> Fig` that updates the line and circle positions and removes the `Move` constructors. For instance

  ```
  let reduceEx02 = reduceMove substEx02
  ```

  binds `reduceEx02` to the value

  ```
  Combine [Circle(P(0.0,0.0),1.0);
           Circle(P(1.0,1.0),1.0);
           Circle(P(2.0,2.0),1.0)]
  ```

KSFUPR01KU **IT University, F2019**

Written exam, Functional Programming
Thursday May 16, 2019

Version 1.00 of May 26, 2019

These exam questions comprise 8 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one file only, e.g., `ksfupr2019.<fsx,pdf>`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `ksfupr2019Snippets.fsx` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3018370`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (20%)

In this question we work with sequences as covered in Chapter 11 in HR.

Question 1.1

- Declare an F# sequence `infSeq3` of type `seq<int>` that produces the infinite sequence $i * 3$ for $i \geq 0$. The identifier i is the index of the element in the sequence. The first few elements are `seq [0;3;6;9;...]`.
- Declare a function `finSeq3` n of type `int->seq<int>`, that returns a finite sequence of the n first elements from `infSeq3`.
- Declare a function `sumSeq3` n of type `int->int`, such that `sumSeq3` n produces the sum of all elements x in `finSeq3` n defined above. For instance `sumSeq3 100` returns 14850.

Question 1.2

Consider the F# declaration below:

```
let seqMap2 f s1 s2 =
  seq { for (x,y) in Seq.zip s1 s2 do
          yield f x y }
```

of type `('a->'b->'c)->seq<'a>->seq<'b>->seq<'c>`.

- Describe the sequence returned by *seqMap2* when called with a function *f* and sequences *s1* and *s2* that fulfil the type signature above.
- Given the function

  ```
  let swap (x,y) = (y,x)
  ```

  explain why the following does not work.

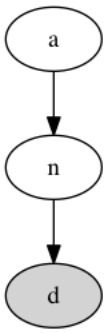
  ```
  seqMap2 swap [1;3;3] [4;5;2]
  ```

  Provide a function `fix` such that below works and show the result `seq [(4,1);(5,3);(2,3)]`.

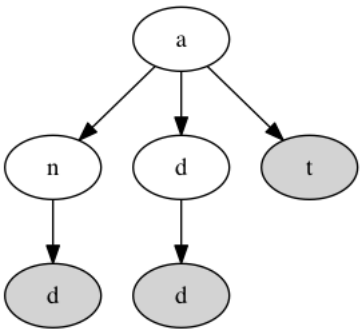
  ```
  seqMap2 (fix swap) [1;3;3] [4;5;2]
  ```

Question 2 (30%)

We define a *trie* as a tree with nodes and edges. A node may have arbitrary many children and edges connect parent nodes to its children.

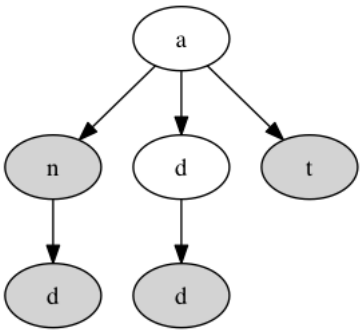


Example 1: A trie with one recognised word: and.

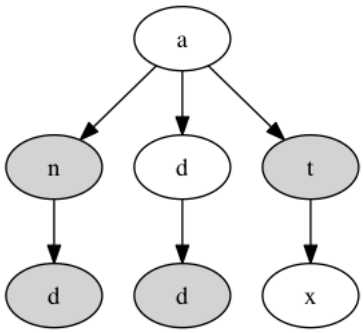


Example 2: A trie with three recognised words: and, add, at.

We define a *letter* to represent one value in a node, i.e., `'a'`, `'n'`, `'d'`, `'t'` in the examples above. We define a *word* to be any concatenated sequence of letters, starting at the root following a path towards a leaf. We have the words `"a"`, `"an"`, `"and"`, `"ad"`, `"add"` and `"at"` defined in the example 1 and 2 above. We define *recognized words* to be the subset of words that are marked as being recognized by the trie. Recognized words are marked grey in the examples. The words `"and"`, `"add"` and `"at"` are recognized in example 2. Example 3 below shows how one path from root to leaf can cover several recognized words, i.e., `"an"` and `"and"`. Example 4 shows that a leaf doesn't have to represent a recognized word, i.e. `"atx"` is not a recognized word.



Example 3: A trie with four recognised words: an, and, add, at.



Example 4: A trie with four recognised words: an, and, add, at.

We represent a trie with the below polymorphic datatype.

```
type TrieNode<'a when 'a : equality> = TN of 'a * bool * TrieNode<'a> list
```

The example 1 is represented by the below value `trie01`:

```
let trie01 = TN('a',false,[TN('n',false,[TN('d',true,[])])])
```

Question 2.1

- Declare values `trie03` and `trie04` representing the two trie examples 3 and 4 above:

```
let trie03 = TN('a',false,[...])
let trie04 = TN('a',false,[...])
```

- Write the type of the value `trie04`. Explain why the type is either monomorphic or polymorphic.
- Declare an F# exception named `TrieError` that carry a string describing an error condition on a function on tries.

Question 2.2

- Declare a function

```
numLetters : TrieNode<'a> -> int when 'a : equality
```

  that returns the number of letters in the trie. For instance `numLetters trie04` returns 7.

- Declare a function

```
numWords : TrieNode<'a> -> int when 'a : equality
```

  that returns the number of recognized words in the trie. For instance `numWords trie04` returns 4.

- Declare a function `exists` *ls t* of type

```
exists : 'a list -> TrieNode<'a> -> bool when 'a : equality
```

  that returns `true` if the list of letters *ls* represents a recognized word in trie *t*; otherwise `false`. For instance `exists ['a';'n'] trie04` returns `true` and `exists ['a';'d'] trie04` returns `false`. Show the result of `exists ['a';'t';'x'] trie04`.

- The difference between `trie03` and `trie04` is the leaf node `'x'` in `trie04`. The path from the root `'a'` to the leaf `'x'` represents the word `"atx"` which is not a recognized word. Declare a function `chkTrie` *t* of type

```
chkTrie : TrieNode<'a> -> bool when 'a : equality
```

  that returns `true` if all paths from root to leaf nodes in trie *t* represents recognized words; otherwise `false`. For instance `chkTrie trie03` returns `true` and `chkTrie trie04` returns `false`.

Question 2.3

- Declare a function `map` *f t* of type

```
map : ('a -> 'b) -> TrieNode<'a> -> TrieNode<'b>
  when 'a : equality and 'b : equality
```

  where `map` *f t* returns the trie where the function *f* has been applied on all letters in the trie *t*. You decide, but must explain, what order the function *f* is applied to the letters in the trie. For instance `map (int) trie01` returns the trie

```
TN(97,false,[TN(110,false,[TN(100,true,[])])])
```

- Write the result executing `map (string) trie03`. What is the type of the result value?

Question 3 (25%)

3.1

A formula F for the compounded interest growth of an amount of money m can be defined as

$$F(m,i,n,k) = \begin{cases} m & \text{if } k \leq 0 \\ F(m,i,n,k-1) * (1.0 + i/n) & \text{if } k > 0 \end{cases}$$

where i is the interest rate per annum compounded n times per year from the kth period to the $(k+1)$th period.

- Declare a function $F\ m\ i\ n\ k$ of type

  ```
  F : float -> float -> float -> int -> float
  ```

  that implements the formula for F above. For instance `F 100.0 0.1 1.0 0` returns 100.0 and `F 100.0 0.1 1.0 10` returns 259.374246.

- Argue whether your implementation of F above is tail–recursive or not. If your implementation of F is not tail recursive, then write a tail–recursive version *FA* of F using an accumulating parameter. The type of *FA* must be the same as for F

3.2

- Declare a function `tabulate` *f start step stop* of type

  ```
  tabulate :(int -> 'a) -> int -> int -> int -> (int * 'a) list
  ```

  that tabulates the function *f* applied on all values between *start* and *stop* increasing with *step* in each iteration, that is, on the values generated by the F# expression: `[start .. step .. stop]`. For instance,

  ```
  tabulate (F 100.0 0.1 1.0) 0 2 4
  ```

  returns `[(0,100.0);(2,121.0);(4, 146.41)]`.

- Declare a function `prettyPrint` *xs* of type

  ```
  prettyPrint : (int * float) list -> unit
  ```

  that takes a list similar to the result of tabulating function `F` above and do a pretty print on the screen. Below shows how the output should be formattet.

  ```
  prettyPrint [(0, 100.0); (2, 121.0); (4, 146.41)];;

    x   | f(x)
  ------+--------
      0 | 100.00
      2 | 121.00
      4 | 146.41
  val it : unit = ()
  ```

Question 4 (25%)

We now consider an internal domain specific language (DSL) called *Positions* to be used to track the buy and sell of stocks. The DSL contains constructors for specifying a position (constructor `Position`) and for specifying actions to change current positions (constructor `Action`).

```
let dt(d,m,y) = System.DateTime(y, m, d)
exception Error of string

type Position =
  | Stock of string
  | Cash of float

type Action =
  | Aquire of System.DateTime * Position
  | Give of System.DateTime * Position
  | Scale of int * Action
  | All of Action list
```

- `dt` is a helper function to create a `DateTime` value.
- The exception `Error` represents an error condition from a function in the library.
- The type `Position` represents two kinds of positions, namely a stock represented by a name, e.g., the stock "Apple" and an amount of cash. We assume all cash and prices are in the same currency.
- The type `Action` represents the possible actions one can do to change the current positions
  - `Aquire`(d,p) is the action of *acquiring* a position (e.g., a stock) at some date d.
  - `Give`(d,p) is the action of *giving* a position (e.g., a stock) away at some date d.
  - `Scale`(n,a) is the action of *scaling* an action.
  - `All` *actions* represent a list of actions.

The example `ex1` represents an action to buy 100 Apple (`APPLE`) stocks at the date 1/2-2018 for the price of 300.3 per stock. The action `Scale` scales the buy (`Aquire`) and payment (`Give`) of one stock to 100 stocks. The price (`Cash`) of 300.3 is for one Apple stock.

```
let ex1 =
  Scale(100,All[Aquire (dt(1,2,2018),Stock "APPLE");
                Give (dt(1,2,2018),Cash 300.3)])
```

Question 4.1

- Declare an F# value `sellApple` of type `Action` that represents the sell of 100 Apple (`APPLE`) stocks at the date 1/3-2018 for the price of 400.4 per stock.

  Hint: You `Give` away the stock and `Aquire` cash.

- Buying and selling stocks require price information, e.g., what is the price for one stock at a certain date. Declare an F# function `price` (*s*,*d*) of type `string*DateTime->float` that returns the price information according to the table below:

Stock	Date	Price
APPLE	1/2-2018	300.3
APPLE	1/3-2018	400.4
ISS	1/2-2018	150.0
ISS	1/3-2018	200.2
TIVOLI	1/2-2018	212.0
TIVOLI	1/3-2018	215.2

  For instance `price ("ISS",dt(1,3,2018))` returns 200.2. The `price` function fails by raising an exception if a price for a date is not known.

Question 4.2

- Based on the example `ex1` of buying 100 Apple stocks, declare an F# function `buyStock` *n s d* of type `int->string->DateTime->Action` that generates an action for buying *n* stocks *s* at the date *d*. You can use the function `price` to get the price for the stock *s* at date *d*. For instance, `buyStock 100 "APPLE" (dt(1,2,2018))` returns the same value as `ex1`. The function should throw an exception if the price is not known at the date *d*.

- Declare an F# function `receiveCash` *c d* of type `float->DateTime->Action`, that given an amount of cash *c* and a date *d* returns an action representing the acquiring of the cash at the date. For instance `receiveCash 100000.0 (dt(1,2,2018))` returns the value `Aquire (2/1/2018 ...,Cash 100000.0)`. (Details of the DateTime component has been deleted).

Question 4.3

Consider the below actions representing the receivable of 100000 and then bying three stocks over two dates `d1` and `d2`.

```
let actions =
  let d1 = dt(1,2,2018)
  let d2 = dt(1,3,2018)
  All [receiveCash 100000.0 d1;
       buyStock 100 "APPLE" d1;
       buyStock 200 "ISS" d1;
       buyStock 50 "TIVOLI" d2]
```

In order to execute actions of bying and selling stocks we need an overview of our current positions, that is, the available cash and stocks. For this, we define below environment *env*:

```
type stockEnv = Map<string,int>
let updStock s n m =
  match Map.tryFind s m with
    None -> Map.add s n m
  | Some n1 -> Map.add s (n+n1) m

type env = float * stockEnv
let emptyEnv = (0.0,Map.empty)
```

A stock environment *stockEnv* is a map from *stock names* to the number of stocks. The function `updStock` *s n m* updates the stock environment *m* with *n* number of stocks *s*. The environment `env` is a pair containing the cash position and the stock environment (*cash*, *stockEnv*)

- Declare an F# function `updEnv` *scaling* (*cash*,*stockEnv*) *pos* of type

  ```
  int -> env -> Position -> env
  ```

  that adds the position *pos* to the environment. Below examples show the result of adding 100 in cash and 100 stocks to an empty environment:

  ```
  > updEnv 100 emptyEnv (Cash 100.0);;
  val it : float * Map<string,int> = (10000.0, map [])

  > updEnv 100 emptyEnv (Stock "APPLE");;
  val it : float * Map<string,int> = (0.0, map [("APPLE", 100)])
  ```

  Hint: You need to cast `scaling` to `float` type for the case with cash.

- Given the `actions` defined above and the ability to maintain an environment with cash and stocks (`updEnv`), we can execute the actions one by one and update the environment with current amount of cash and stocks in our positions. Declare an F# function `execA` *a env* of type

  ```
  Action -> env -> env
  ```

  that given an action *a* and environment *env* executes the action and returns an updated environment representing the amount of cash and stock positions. For instance `execA actions emptyEnv` returns the value (of type `float*Map<string,int>`):

  ```
  (29210.0, map [("APPLE", 100); ("ISS", 200); ("TIVOLI", 50)])
  ```

  Hint: You negate the scaling when implementing `Give`. For instance giving cash of 100 away means reducing current position of cash.

  Hint: You may use below template:

  ```
  let execA action env =
   let rec exec scaling env = function
     Aquire(d,p) -> ...
     | ...
   exec 1 env action
  ```

KSFUPR01KU **IT University, F2020**

Written exam, Functional Programming
Monday May 25, 2020

Version 1.00 of May 24, 2020

These exam questions comprise 7 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand–ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand–in one file only, e.g., `ksfupr2020.<fsx,pdf>`. Do not use time on formatting your solution in Word or PDF.

You are welcome to use the accompanying file `may2020Snippets.fsx`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand–in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand–in must contain the following declaration:

I hereby declare that I myself have created this exam hand–in in its entirety without help from anybody else.

Question 1 (30%)

The concept of maps is described in HR Section 5.3. The examples (**dice1**) and (**dice2**) below shows example maps storing the result of rolling a dice 15 and 20 times respectively, i.e., for table (**dice1**) we got 1 eye 4 times, 2 eyes 2 times etc. The example (**ex1**) shows a map where the characters `'A'`, `'B'` and `'C'` are mapped to their ASCII values 65, 66 and 67.

(**dice1**)

Eyes	Freq.
1	4
2	2
3	3
4	2
5	2
6	2

(**dice2**)

Eyes	Freq.
1	4
2	2
3	3
4	3
5	5
6	3

(**ex1**)

Character	ASCII Code
`'A'`	65
`'B'`	66
`'C'`	67

We define a type `mymap` representing a map implemented as an F# list of pairs (k_i, v_i) for $0 \leq i < n$, where k_i is the keys, v_i the values keys are mapped to and n is the size of the map. The order of the pairs in the list does not matter.

```
type mymap<'a,'b> = MyMap of list<'a*'b>
```

The map **ex1** above can be implemented in different ways because there are no assumptions on the order of the pairs:

```
let ex1 = MyMap [('A',65);('B',66);('C',67)]
let ex1' = MyMap [('C',67);('A',65);('B',66)]
```

Hint: The .NET `List` library (HR Section 5.1) can ease the implementation of the functions below.

Question 1.1

- Declare map values `dice1` and `dice2`, representing the two maps (**dice1**) and (**dice2**) above.
- Explain the type of the two values `ex1` and `dice1`.
- Declare an F# function `emptyMap()` of type `unit -> mymap<'a,'b>` that returns an empty map.
- Declare an F# function `size` m of type `mymap<'a,'b> -> int` that returns the size of the map m. For instance `size ex1` returns 3 and `size (emptyMap())` returns 0.

Question 1.2

- Declare an F# function `isEmpty` m of type `mymap<'a,'b> -> bool` that returns true if the map m is empty; otherwise false. For instance `isEmpty ex1` returns false and `isEmpty (emptyMap())` returns true.
- Declare an F# function `tryFind` k m of type

```
'a -> mymap<'a,'b> -> ('a * 'b) option when 'a : equality
```

that returns the value `Some` (k, v) if k exists in m and v is the value that k is mapped to; otherwise `None` is returned. For instance `tryFind 'B' ex1` returns `Some ('B', 66)` and `tryFind 'D' ex1` returns `None`.

Explain what the constraint `when 'a : equality` in the type for `tryFind` means and why it is necessary.

- Declare an F# function `remove` k m of type `'a -> mymap<'a,'b> -> mymap<'a,'b> when 'a : equality` that removes the entry for k in the map m if exists; otherwise m is unchanged. For instance, `remove 'B' ex1` may return the value `MyMap [('A',65);('C',67)]`.

- Declare an F# function `add` k v m of type `'a -> 'b -> mymap<'a,'b> -> mymap<'a,'b> when 'a : equality` that returns a new map where the pair (k, v) is added to (or replaced in) the map m depending on whether the key k already exists in m. For instance `add 'D' 68 ex1` may return `MyMap [('D', 68);('A', 65);('B', 66);('C',67)]` and `add 'A' 222 ex1` may return `MyMap [('A', 222);('B',66);('C',67)]`.

Question 1.3

- Declare an F# function `upd` f k v m of type `('a -> 'a -> 'a) -> 'b -> 'a -> mymap<'b,'a> -> mymap<'b,'a> when 'b : equality` that checks whether the key k exists in the map m. If k exists and maps to value v', then a map m is returned where k is mapped to the combined value with function f, i.e., $f\ v\ v'$. If k does not exist, a map m with (k, v) added is returned. For instance `upd (+) 'A' 65 ex1` may return `MyMap [('A', 130);('B',66);('C',67)]` and `upd (+) 'D' 68 ex1` may return `MyMap [('D',68);('A',65);('B',66);('C',67)]`.

- Declare an F# function `map` f m of type `('a -> 'b -> 'c) -> mymap<'a,'b> -> mymap<'a,'c>` that returns the map resulting from applying the function f on all entries in the map m. For instance `map (fun k v -> v+2) ex1` may return `MyMap [('A', 67);('B',68);('C',69)]`.

- Declare an F# function `fold` f s m of type `('a -> 'b -> 'c -> 'a) -> 'a -> mymap<'b,'c> -> 'a` that folds the function f over all entries in the map m starting with initial state s. For instance `fold (fun s k v -> s+v) 0 dice1` returns 15, i.e., the number of times the dice was rolled.

Question 2 (25%)

Consider the below function, *collatz*, defined for any positive integer $n > 0$.

$$\mathit{collatz}(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Question 2.1

- Declare an F# function `even` n of type `int -> bool` that returns true if n is even and false otherwise. For instance `even 1` returns false and `even 42` returns true.
- Declare an F# function `collatz` n of type `int -> int` that computes the value *collatz*(n) defined above. For instance, `collatz 45` returns 136. You may assume $n > 0$.
- Declare an F# function `collatz'` n of type `int -> int` that computes the same value *collatz*(n) defined above. However, this function must return `System.Exception` with an error message in case argument $n \leq 0$. For instance `collatz' 45` returns 136 and `collatz' 0` should return an exception similar to `System.Exception: collatz': n is zero or less`.

Question 2.2

We can now, for any $n > 0$, form a sequence by repeatedly applying *collatz* as follows:

$$a_0 = n, \;\; a_1 = \mathit{collatz}(a_0), \;\; \ldots, \;\; a_N = \mathit{collatz}(a_{N-1}), \;\; \ldots$$

This can also, for arbitrary function f, be written as

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f(a_{i-1}) & \text{for } i > 0 \end{cases}$$

The Collatz conjecture states that the sequence $a_0, a_1, \ldots$, computed with function *collatz*, will eventually reach the number 1, regardless of initial integer $n > 0$. The conjecture has not yet been proven.

- Declare an F# function `applyN` f n N of type `('a -> 'a) -> 'a -> int -> 'a list`, that applies function f repeatedly and returns a list with the sequence elements $[a_0;a_1;\ldots;a_N]$ as defined above. For instance `applyN collatz 42 8` returns the list `[42;21;64;32;16;8;4;2;1]`.
- The smallest i for which $a_i = 1$ is called the *total stopping time*. Declare an F# function `applyUntilOne` f n of type `(int -> int) -> int -> int`, that computes the total stopping time for function f with initial argument n. For instance `applyUntilOne collatz 42` returns 8.

Question 2.3

In this subquestion we work with sequences as covered in Chapter 11 in HR. Consider the F# declaration of type `('a -> 'a) -> 'a -> seq<'a>`:

```
let rec mySeq f x =
  seq { yield x
        yield! mySeq f (f x)}
```

- Describe the sequence returned by `mySeq` using `mySeq collatz 42` as an example.
- Declare an F# function `g` x of type `int -> int`, such that `mySeq g 1` returns the sequence $1, 2, 4, 8, 16, 32, \ldots$

Question 3 (20%)

A portfolio of shares traded on a stock exchange can be viewed as a list of transactions as shown below.

```
type name = string
type quantity = float
type date = int * int * int
type price = float
type transType = Buy | Sell
type transData = date * quantity * price * transType
type trans = name * transData

let ts : trans list =
  [("ISS", ((24,02,2014),100.0,218.99,Buy));  ("Lego",((16,03,2015),250.0,206.72,Buy));
   ("ISS", ((23,02,2016),825.0,280.23,Buy));  ("Lego",((08,03,2016),370.0,280.23,Buy));
   ("ISS", ((24,02,2017),906.0,379.46,Buy));  ("Lego",((09,11,2017), 80.0,360.81,Sell));
   ("ISS", ((09,11,2017),146.0,360.81,Sell)); ("Lego",((14,11,2017),140.0,376.55,Sell));
   ("Lego",((20,02,2018),800.0,402.99,Buy));  ("Lego",((02,05,2018),222.0,451.80,Sell));
   ("ISS", ((22,05,2018),400.0,493.60,Buy));  ("ISS", ((19,09,2018),550.0,564.00,Buy));
   ("Lego",((27,03,2019),325.0,625.00,Sell)); ("ISS", ((25,11,2019),200.0,680.50,Sell));
   ("Lego",((18,02,2020),300.0,720.00,Sell))]
```

For instance, on the date of 24 February 2014 a quantity of 100.0 ISS shares are bought for a price of 218.99DKK per share. **You can assume the transactions are ordered by date**.

The goal of this assignment is, for each share, to compute the current quantity in the portfolio and the weighted average share price for buying the shares.

3.1

We first group transactions in the portfolio by sharename represented as a library map (Section 5.3 in HR) from sharename (`name`) to a list of transaction data for that share.

- Declare an F# function `addTransToMap` t m of type

  ```
  trans -> Map<name,transData list> -> Map<name,transData list>
  ```

  that adds a transaction t to map m. For instance

  ```
  let m1 = addTransToMap ("ISS", ((24,02,2014),100.0,218.99,Buy)) Map.empty
  let m2 = addTransToMap ("ISS", ((22,05,2018),400.0,493.60,Buy)) m1
  ```

  returns the map `m2` mapping `"ISS"` to a list of the two transactions:

  ```
  map
    [("ISS",
      [((22, 5, 2018), 400.0, 493.6, Buy); ((24, 2, 2014), 100.0, 218.99, Buy)])]
  ```

  Notice, the transactions are now in reverse order by date. You can use the template below.

  ```
  let addTransToMap (n,td) m =
    match Map.tryFind n m with
    ...
  ```

- You can now use `List.foldBack` to fold the function `addTransToMap` over the list `ts` and build a map mapping sharenames to their transactions. Declare an F# value `shares` of type `Map<name,transData list>` as the result of this computation. You can use the template

```
let shares = List.foldBack ... ts ...
```

that returns the value

```
map
 [("ISS",
   [((24, 2, 2014), 100.0, 218.99, Buy); ((23, 2, 2016), 825.0, 280.23, Buy);
    ((24, 2, 2017), 906.0, 379.46, Buy); ((9, 11, 2017), 146.0, 360.81, Sell);
    ((22, 5, 2018), 400.0, 493.6, Buy); ((19, 9, 2018), 550.0, 564.0, Buy);
    ((25, 11, 2019), 200.0, 680.5, Sell)]);
  ("Lego",
   [((16, 3, 2015), 250.0, 206.72, Buy); ((8, 3, 2016), 370.0, 280.23, Buy);
    ((9, 11, 2017), 80.0, 360.81, Sell); ((14, 11, 2017), 140.0, 376.55, Sell);
    ((20, 2, 2018), 800.0, 402.99, Buy); ((2, 5, 2018), 222.0, 451.8, Sell);
    ((27, 3, 2019), 325.0, 625.0, Sell); ((18, 2, 2020), 300.0, 720.0, Sell)])]
```

3.2

For each share, we now compute the quantity in the portfolio and the weighted average price paid per share unit. Let transactions $t_1, \ldots, t_n$ be given for a share ordered by date. Let tq_i and avg_i be the total quantity and average share price computed for the first i transactions. Let $t_{i+1} = (d, q, p, tType)$ be the next transaction. In case the transaction type *tType* is `Buy` the formulas are $tq_{i+1} = tq_i + q$ and $avg_{i+1} = (avg_i * tq_i + q * p)/(tq_i + q)$. In case *tType* is `Sell` the formulas are $tq_{i+1} = tq_i - q$ and $avg_{i+1} = avg_i$, because a `Sell` transaction does not affect the average price paid in `Buy` transactions.

- Declare an F# function `accTrans` (tq_i,avg_i) (*d*,*q*,*p*,*tType*) that returns the pair (tq_{i+1}, avg_{i+1}) as defined above. You can use the template

```
let accTrans (tq:float,avg:float) ((d,q,p,tType):transData) =
  match tType with
    Buy -> ...
  | Sell -> ...
```

The `accTrans` function must work so that folding it over a list of transactions `ts` as shown below will compute the quantity and average price. For instance

```
let quantityAndAvgPrice ts =
  List.fold accTrans (0.0,0.0) ts

quantityAndAvgPrice [((24,02,2014),100.0,218.99,Buy);
                     ((23,02,2016),825.0,280.23,Buy)]
```

returns the value `(925.0,273.6094595)`.

- Declare an F# value `res` of type `Map<name,(float*float)>` that is the result of mapping the function `quantityAndAvgPrice` over the map of shares `shares` computed in Question 3.1 above. The value for `res` is `map [("ISS",(2435.0,401.1102931)); ("Lego",(353.0,352.1896237))]`.

Question 4 (25%)

Question 4.1

Consider the F# declaration

```
let rec dup = function
    [] -> []
  | x::xs -> x::x::dup xs
```

of type `'a list -> 'a list`.

- Describe the list generated based on the input list. For instance, given input list $[e_0; \ldots; e_n]$ the list returned by `dup` $[e_0; \ldots; e_n]$ is ....
- The function `dup` is not tail recursive. Declare a tail–recursive version `dupA` of `dup` using an accumulating parameter.

Question 4.2

In this question we work with sequences as covered in Chapter 11 in HR.

- Declare an F# function `replicate2` i of type `'a -> seq<'a>` that returns a finite sequence of the value i replicated two times. See example below.

```
> replicate2 4;;
val it : seq<int> = [4; 4]
```

- Declare an infinite sequence `dupSeq` of type `seq<int>`. The sequence consists of all integers $i \geq 0$ in increasing order and replicated twice. The first 10 elements in the sequence are 0; 0; 1; 1; 2; 2; 3; 3; 4; 4; ....

Question 4.3

- Declare an F# function `dupSeq2` s of type `seq<'a> -> seq<'a>` that duplicates all elements in s. For instance `dupSeq2 (seq[1;2])` returns `seq [1;1;2;2]`. The implementation of `dupSeq2` must use sequence expressions (Section 11.6 in HR).