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).