**KSFUPR01KU** **IT University, F2020**

# Written exam, Functional Programming
# Friday August 21, 2020

Version 1.00 of August 20, 2020

These exam questions comprise 6 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`. Do not use time on formatting your solution in Word or PDF.

You are welcome to use the accompanying file `aug2020Snippets.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%)

In this question we will consider a simple *pizza menu register*, where *menu items* are identified by a unique *Id*. A menu item also has a *Name* and a unit *Price*. The table below shows an example menu.

| Id | Name | Price |
|---|---|---|
| 1 | Vesuvio | 55,00 |
| 2 | Pepperoni | 50,00 |
| 3 | Italiana | 59,00 |
| 4 | Capriccosa | 62,00 |

We define a type `item` representing a menu item and a type `pizzareg` representing the register as a list of items:

```
type item = {Id: int;
             Name: string;
             Price: float}
type pizzareg = item list
```

There are no assumptions on how items are ordered in the pizza register or if there are more items with the same Id.
**Hint**: The .NET libraries `List`, `Map` and `Set` (HR Chapter 5) can ease the implementation of the functions below.

### Question 1.1

- Declare four item values `item1,...,item4` of type `item` each representing one of the menu items in the table above. The value `item1` must represent the menu item with Id 1 etc.
- Declare a value `reg` of type `pizzareg` containing the four menu items in the table above.
- Declare a value `regDup` of type `pizzareg` that has two or more items with the same Id. For instance, you can extend the value `reg` with one or more new items.
- Declare an F# function `emptyReg()` of type `unit -> pizzareg` that returns an empty pizza register.
- Declare an F# function `size` $r$ of type `pizzareg -> int` that returns the size of the pizza register $r$. For instance `size reg` returns 4 and `size (emptyReg())` returns 0.

### Question 1.2

- Declare an F# function `isEmpty` $r$ of type `pizzareg -> bool` that returns true if the pizza register $r$ is empty; otherwise false. For instance `isEmpty reg` returns false and `isEmpty (emptyReg())` returns true.
- Declare an F# function `pId` *i item* of type `int -> item -> bool` that compares the id *i* with the Id component of the *item* record and returns true if they are equal; otherwise returns false. For instance, `pId 2 item1` returns false and `pId 2 item2` returns true.

  Declare an F# function `pName` *n item* of type `string -> item -> bool` that compares the name *n* with the Name component of the *item* record and returns true if they are equal; otherwise returns false. For instance, `pName "Pepperoni" item1` returns false and `pName "Pepperoni" item2` returns true.

- Declare an F# function `tryFind` *p r* of type `(item->bool) -> pizzareg -> item option` that returns the value `Some` *item* if the predicate function *p* applied on some *item* in *r* returns true. For instance, `tryFind (pName "Pepperoni") reg` returns

  ```
  Some {Id = 2;
        Name = "Pepperoni";
        Price = 50.0;}
  ```

  and `tryFind (pId 30) reg` returns `None`.

  In case there are more than one *item* in *r* where *p item* returns true explain which one is returned.

- Declare an F# function `isUniqueById` *r* of type `pizzareg -> bool` that returns true if all Id components of the items in *r* are unique; otherwise returns false. For instance, `isUniqueById reg` returns true and `isUniqueById regDup` returns false.

### Question 1.3

We declare an *order* type as follows:

```
type order = (int * int) list
```

An order is a list of tupples (*id*,*num*) where *id* is the Id of the menu item and *num* the number of orders of this menu item. For instance, an order of three Pepperoni and four Italiana pizzas can be represented as follows:

```
let order1:order = [(2,3);(3,4)]
```

There are no assumptions on how an order is represented. For instance, the order declaration

```
let order2:order = [(2,3);(1,2);(1,3);(2,4)]
```

represents an order with 7 Pepperoni and 5 Vesuvio pizzas.

- Declare an F# function `collectById` *order* of type `order -> order` that collects all order tupples with same Id into one order tupple with the sum of times the Id has been ordered. For instance `collectById order1` may return `[(2,3);(3,4)]` and `collectById order2` may return `[(1,5);(2,7)]`.

  **Hint**: One approach is to use a temporary Map mapping item Id to the number of times this Id has been ordered.

- Declare an F# function `makeOrderList` *reg order* of type

  ```
  pizzareg -> order -> (int*string*float) list
  ```

  that produces a list of tripples (*num*,*name*,*sumPrice*), i.e., the *number* of times pizza with *name* has been ordered and the *summed price* for those pizzas. For instance, `makeOrderList reg order1` returns `[(4,"Italiana",236.0);(3,"Pepperoni",150.0)]` and `makeOrderList reg order2` returns `[(4,"Pepperoni",200.0);(3,"Vesuvio",165.0);(2,"Vesuvio",110.0);(3,"Pepperoni",150.0)]`. Combining with `collectById` you can reduce the result order list. For instance `makeOrderList reg (collectById order2)` returns `[(7,"Pepperoni",350.0);(5,"Vesuvio",275.0)]`.

# Question 2 (25%)

Consider the following F# declaration

```
let rec f i = function
    [] -> []
  | x::xs -> (x+i) :: g (i+1) xs
  | x1::x2::xs -> (x1+i) :: g (i+1) (x2::xs)
and g i = function
    [] -> [i]
  | x::xs -> (x-i) :: f (i+1) xs
```

The types of `f` and `g` are `int -> int list -> int list`.
The expression `f 10 [1..10]` returns the value

```
[11; -9; 15; -9; 19; -9; 23; -9; 27; -9]
```

## Question 2.1

- Compiling the functions `f` and `g` above provides the following compiler warning

  ```
  warning FS0026: This rule will never be matched
  ```

  The compiler reports line 4 (`| x1::x2 ...`) as the source of the warning.

  Explain the reason for the compiler warning.

  Declare new versions of `f` and `g` called `fFix` and `gFix` without the warning and that computes the same values.

- Consider a list $l$ of values $[x_0; \ldots; x_N]$ for any $N \geq 0$. Declare an F# function `sum` $l$ of type `int list -> int list` that returns a new list where each element is the sum of two consequtive elements in the list $l$. If $l$ is empty or has one element only, then the empty list is returned. A few examples in the table below.

  | **Example** | **Result** |
  |---|---|
  | `sum []` | `[]` |
  | `sum [3]` | `[]` |
  | `sum [3;1]` | `[4]` |
  | `sum [3;1;2]` | `[4;3]` |

- The result of `sum (f 10 [1..10])` is

  ```
  [2; 6; 6; 10; 10; 14; 14; 18; 18]
  ```

  Considering the result of `f 10 [1..10]` above explain, why the result is as expected.

## Question 2.2

- Explain why the two functions `g` and `f` are not tail recursive.
- Declare tail–recursive variants `fA` and `gA` of `f` and `g` respectively using accumulating parameter. Show at least one example of `fA` producing the same result as `f`.
- Explain whether there exists argument values for `fA` such that the computation will result in an unbounded number of recursive function calls.

## Question 3 (25%)

The below type declarations are a simplistic approach to model *family trees*.

```
type name = string
type FamilyTree = Family of name * name * Children list
and  Children = Single of name
              | Couple of FamilyTree
```

We use a mutual recursive type (HR Section 6.6) where the type `FamilyTree` represents a family and the type `Children` represents a children in a family. A children can be a single or part of a couple. The types do not represent who have the role of mother or farther etc. The only assumption is that two names are required to have children.
A family with two persons “Hanne” and “Peter” with no children are represented in the F# value `fam1` below. The value `fam2` represents a family with “Kurt” and “Pia” having a children “Henrik” being a single.

```
let fam1 = Family ("Hanne", "Peter", [])
let fam2 = Family ("Kurt", "Pia", [Single "Henrik"])
```

### Question 3.1

- What is the type of `fam1` and `fam2`? Explain how you can deduce this from the declarations.
- Declare an F# value `fam3` of type `FamilyTree` representing two persons “Charlotte” and “Oliver” having two children. The first children happens to be “Hanne” married to “Peter” and they have no children (`fam1`). The second children is “Kurt” who happens to be married to “Pia” having one children “Henrik” being single (`fam2`).

### Question 3.2

- Declare two mutually recursive functions `numPerFam` *ft* of type `FamilyTree -> int` and `numPerChildren` *ch* of type `Children -> int`. The function `numPerFam` computes the number of persons in the family tree *ft*. The function `numPerChildren` computes the number of persons for the children *ch*. For instance `numPerFam fam1` returns 2 and `numPerFam fam2` returns 3.

  Provide the result of executing `numPerFam fam3`.
- Declare two mutually recursive functions `toListFam` *ft* of type `FamilyTree -> name list` and `toListChildren` *ch* of type `Children -> name list`. The function `toListFam` computes a list of all names part of the family tree *ft*. The function `toListChildren` computes all names part of the children, i.e., the children can be single or part of a couple. For instance `toListFam fam1` returns `["Hanne";"Peter"]` and `toListFam fam2` returns `["Kurt";"Pia";"Henrik"]`. A family can have persons with the same name and hence the result may contain the same name several times.

  Provide the result of executing `toListFam fam3`.

### Question 3.3

- Can you create a value of type `FamilyTree` or `Children` where a single named person can have children? If no, please explain. If yes, please make an example value of type `FamilyTree` or `Children`.

## Question 4 (20%)

Consider the below mutually recursive functions, *F* and *M*, defined for any positive integer $n > 0$.

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - M(F(n-1)) & \text{if } n > 0 \end{cases}$$

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - F(M(n-1)) & \text{if } n > 0 \end{cases}$$

### Question 4.1

- Declare the two mutually recursive F# functions *F n* and *M n* as defined above. Both has type `int -> int`. For instance `List.map F [0..10]` returns

  ```
  [1; 1; 2; 2; 3; 3; 4; 5; 5; 6; 6]
  ```

  and `List.map M [0..10]` returns

  ```
  [0; 0; 1; 2; 2; 3; 4; 4; 5; 6; 6]
  ```

- Declare an F# function `combineFM` *n* of type `int -> int*int` that returns the pair (*F n*,*M n*). For instance `combineFM 4` returns `(3,2)`.

### Question 4.2

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

- Declare an infinite F# sequence `Fseq` of type `seq<int>` that is the sequence of `F` *n* for $n \geq 0$ as defined above. For instance `Seq.item 5 Fseq` returns 3.

- Declare a cached version of the F# sequence `Fseq` called `FseqCache`. For instance `Seq.item 5 FseqCache` returns 3.

  Explain why `FseqCache` and `Fseq` have the same type.

- Declare an F# sequence `combineFMSeq` of type `seq<int*int>` that returns the elements `combineFM` *n* for $n \geq 0$. For instance, `Seq.take 4 combineFMSeq` returns

  ```
  seq [(1, 0); (1, 0); (2, 1); (2, 2)]
  ```

  The sequence `combineFMSeq` must be implemented using *sequence expressions*, see HR Section 11.6.