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.

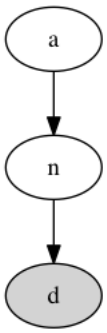
  ```
  seqMap2 swap [1;3;3] [4;5;2]
  ```

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

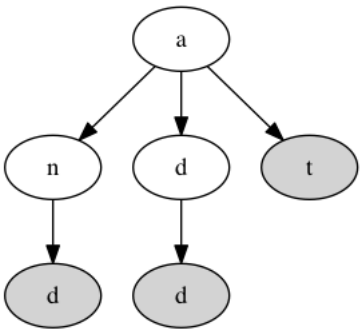
  ```
  seqMap2 (fix swap) [1;3;3] [4;5;2]
  ```

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.

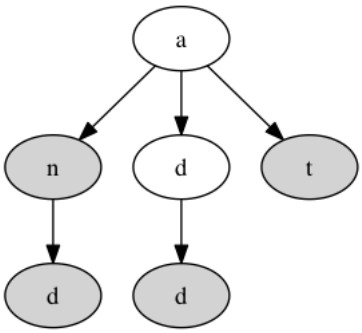


Example 1: A trie with one recognised word: and.

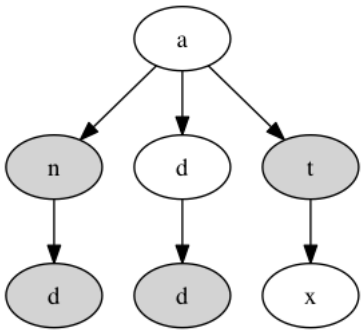


Example 2: A trie with three recognised words: and, add, at.

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
  ```