Tallinn University of Technology

Department of Software Science

Exam paper: Autumn 2021/22

Advanced Programming

ITT8060

Time allowed TWO Hours

Answer ALL FOUR questions

No calculators, mobile phones or other electronic devices capable of storing or retrieving text may be used.

A print text book
(Real World Functional Programming, Functional Programming Using F#, or Expert F#)
is allowed.

DO NOT open the examination paper until instructed to do so

Name:	
Student ID:	
Marks (to be filled by teaching staff):	

Question 1: Multiple choice

Please circle **A**, **B**, **C** or **D** according to which of them best matches the answer. In case there are multiple correct answers you should choose the best one. Only a single circle is considered to be the correct answer. In case you make a mistake, cross out the wrong answer and write clearly next to the question what the answer is.

a. Evaluating the expression `List.filter (fun n -> n % 3 = 0) [3;9;1;8;4]` returns

   a) `[3;9]` b) `[3;9;1;8;4]` c) `[]` d) Type error

b. The value of `List.fold (>>) id [(-) 3; (*) 2] 1` is

   a) 4 b) -6 c) Type error d) None of the above

   (Given the type of the function `(>>)` is `(('a -> 'b) -> ('b -> 'c) -> 'a -> 'c)` and the type of function `id` is `'a -> 'a`)

c. The type of `(lazy (1,2)).Force () |` is

   a) `Lazy<int*int>` b) `int*int` c) `Lazy<int>*int` d) `Lazy<int>*Lazy<int>`

d. Evaluating the expression `Option.bind Some (Some (None :  bool option))` returns, given the type of `Option.bind` is `(('a -> 'b option) -> 'a option -> 'b option)`

   a) `None` b) `false` c) `Some None` d) `Some Some None`

e. Evaluating the expression `let rec s = seq {yield 3; yield! Seq.map (fun n -> n) s}` produces

   a) An infinite sequence b) A sequence of 2 ints c) A list d) An error

f. The type of the following function is

```
let rec c b =  match b with
   | []     -> []
   | d :: e -> e :: (c (List.map id e))
```

   a) `'a list -> unit` b) `'a list -> 'b list` c) `'a list -> 'a list` d) `'a list -> 'a list list`

g. The expression `["a",1;"b",3] |> List.map (id >> fst)` will evaluate to

   a) `3` b) `[1;3]` c) `["a",1;"b",3]` d) `["a";"b"]`

h. Evaluation of the following code will result in

```
let failingWorkflow =
  async { do failwith "fail" }
Async.RunSynchronously failingWorkflow
```

a) A value of type `Async<unit>` being returned; b) “failwith” being printed; c) A delay of 1000 ms; d) None of the above.

i. The function `f` defined below

```
let rec f xs y =
  match xs with
    | [] -> [y]
    | x :: xs' -> f xs' (y x)
```

a) has type `('a -> 'b) list -> 'a -> 'b list`. b) Is tail recursive. c) Both A and B. d) None of the above.

j. Given that the type of `List.collect` is `(('a -> 'b list) -> 'a list -> 'b list)`, the expression `List.collect (fun x -> x ) [2; 3; 5]` will evaluate to

a) `[[2]; [3]; [5]]` b) `[2; 3; 5]` c) type error d) none of the above

k. Given that the type of `Seq.unfold` is `(('a -> ('b * 'a) option) -> 'a -> seq<'b>)`, the expression
`Seq.unfold (fun x -> None: (int * int) option) 1`
will evaluate to

a) `seq None` b) empty sequence c) `seq [1]` d) None of the above.

l. The expression `(printf "X"; (fun n -> printf "Y"; n + n)) (printf "Z"; 2)` evaluates to `4`. What is printed to the console by evaluating this expression?

a) `XYZ` b) `XZY` c) `ZXY` d) `ZYX`

Question 2: Lists

a. Consider the following functions.

```
let foo x = printf "f"; 2 * x
let bar x = printf "b"; [x; x]
```

(i) What is the value that the following expression evaluates to? (2 points)

```
[1..5] |> List.map foo |> List.take 3 |> List.collect bar
```

(ii) What is the string that is printed to the console by evaluating the above expression? (2 points)

(iii) What is the value that the following expression evaluates to? (2 points)

```
[1..5] |> List.filter (fun i -> i % 2 = 0) |> List.map (bar << foo)
```

(iv) What is the string that is printed to the console by evaluating the above expression? (2 points)

b. (i) Define the function `generate : int -> 'a -> 'a -> ('a -> 'a -> 'a) -> 'a list` so that `generate n a b f` generates a list of values $a_0, \ldots, a_{n-1}$ where $a_0 = a$, $a_1 = b$ and $a_{k+2} = f\ a_k\ a_{k+1}$. You may assume that the parameter `n : int` is non-negative.

Example: `generate 8 0 1 (+)` should give the first 8 Fibonacci numbers: `[0;1;1;2;3;5;8;13]`.

Use explicit recursion (i.e., do not use any higher-order functions from the `List` module) and make sure that your solution is *not* tail-recursive. (5 points)

(ii) Explain what it is that makes your solution not tail-recursive. (1 point)

c. Define the function `applyEvens : ('a -> 'a) -> 'a list -> 'a list` that applies the given function to elements at even positions in the list and leaves the other elements the same.

Define it using a fold operation over the input list. Pick an accumulator (type) that allows to keep track of whether you are at an even position or not. The first element in a list is at position 0.

Example: `applyEvens ((*) 2) [1..5]` must evaluate to `[2;2;6;4;10]`. (5 points)

d. Consider the following definitions.

```
let rec t p q w =
match q with
| []      -> w
| r :: q' -> t (r :: p) q' ((p, r, q') :: w)

let h x = t [] x []
```

(i) What is the type of `h`? (3 points)

(ii) What is the result of evaluating: `h [1;2;3;4]`? (3 points)

Here are the types of some functions that may be relevant.

```
List.collect   : ('a -> 'b list) -> 'a list -> 'b list
List.filter    : ('a -> bool) -> 'a list -> 'a list
List.fold      : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
List.foldBack  : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.map       : ('a -> 'b) -> 'a list -> 'b list
List.replicate : int -> 'a -> 'a list
List.take      : int -> 'a list -> 'a list
(<<)           : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

This page has been left blank intentionally. Please use it for your answers.

Question 3: Bags

We consider a *bag* to be a data structure which allows to associate keys with values together with their counts (how many times the key and value are associated). A key cannot be associated with two *different* values but it can be associated with the same value multiple times. Such a bag can be represented by a list of key–value–count triples. If the list contains a triple `(k, v, c)`, then the key `k` and value `v` are associated `c` times. For example, `[(1, 'a' , 1); (2, 'b', 4)]` denotes a bag where the value `'a'` is associated to the key `1` and there is one instance of it in the data structure and the value `'b'` is associated to the key `2` and there are four instances of it in the data structure. In a *valid* representation there should be no two triples with the same key.

a. Consider the following function definition:

```
let rec f x y =
  match y with
   | []           -> None
   | (x', z', s') :: w -> if x' = x then
                             Some (z',s')
                         else
                             f x w
```

   (i) Evaluate the following expression:

   `[(1, 2, 1); (2, 4, 3); (3, 1, 2)] |> f 3`

   (3 points)

   (ii) Give the type of `f`. (3 points)

   (iii) Evaluate `[] |> f "5"`. (3 points)

b. Write a function

```
insert : 'a -> 'b -> ('a * 'b * int) list -> ('a * 'b * int) list
```

that inserts the given key and associated value into the given bag. That is to say that the result (bag) must associate the given value with the given key. If the key is already present in the bag and the values match, then the appropriate count should be incremented. In any other case the result (bag) should associate the given key and value exactly once. If the input bag is valid, then the result bag should also be valid. (5 points)

c. Given a function `repeat` that will apply the function *f* with initial argument *x* on its result *c* times (where c is positive),

```
let rec repeat c f x =
   match c with
   | 0 -> x
   | n -> repeat (n-1) f (f x)
```

write an expression that will apply insert with arguments where key is 2 and value is "b" on an initially empty list 4 times using `repeat`. (2 points)

d. Write a function

```
union : ('a * 'b * int) list -> ('a * 'b * int) list -> ('a * 'b * int) list
```

that forms the union of two bags. The union contains all the key–value–count triples of both bags where coinciding keys and values will have summed counts. If some key is contained by both bags, but the values do not match, only the corresponding value and count in the first bag will be preserved in the union. Use `List.fold` to invoke the combination of `repeat` and `insert` functions repeatedly, starting from the second argument. (4 points)

e. Consider the following code:

```
let rec check m =
  match m with
    | []
        -> false
    | (k, v, c) :: t
        -> match List.filter (fun x,y,z -> x = v) m with
             | [] -> check t
             | _  -> true
```

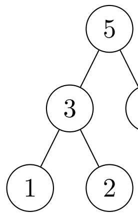
The function `check` is supposed to check if a given list of triples is a valid representation of a bag, that is, it does not contain any key more than once. The code contains bugs. Identify all of them. (5 points)

Question 4: Trees

Here is a definition of node-labelled binary trees that is parametric in the type of labels. A tree can either be *empty* or it can be a *branch* (node) holding a value of type `'a` and two subtrees: left and right.

```
type 'a tree = Empty
| Branch of 'a tree * 'a * 'a tree
```

a. Define a value of type `int tree` that represents the following tree (empty subtrees are omitted from the picture): (2 points)



b. Define the function `truncate : int -> 'a tree -> 'a tree` that truncates the given tree at the given depth (`int`). We consider the root node to be at depth 0. By truncating at depth n we mean that all the subtrees whose root is at depth n in the given tree must be replaced by the empty tree. You may assume that depth is non-negative. (4 points)

c. Define the function `map : ('a -> 'b) -> 'a tree -> 'b tree` that transforms the given tree by applying the given function to every label in the tree. The result tree should have exactly the same shape as the input tree and a label in the result tree should be obtained by applying the given function to the label at the corresponding position in the input tree. (4 points)

d. A *heap* is a tree-based data structure which is essentially an almost complete tree that satisfies the heap property: in a *max* heap, for any given node c, if p is the parent node of c, then the label of p is greater than or equal to the label of c. Define the function `lessOrEq : int tree -> int option -> bool` that checks whether the given tree satisfies the max heap property. If the second argument is `Some n`, then the parent node of the tree (first argument) was labelled by `n`. If the second argument is `None`, then the tree did not have a parent node. (4 points)

e. Define the function `heapLike : int tree -> bool` that evaluates to `true` precisely when the given tree satisfies the max heap property. Use `lessOrEq` in your solution. (1 point)

f. A *binary search tree* (BST) is a binary tree whose (internal) nodes store a label greater than all the labels in the node's left subtree and less than those in its right subtree. We consider trees `('k * 'v) tree` where the labels are pairs of a key and a value to represent dictionaries. Labels of type `'k * 'v` are ordered according to the first component (the keys). Define the function `insert : 'k -> 'v -> ('k * 'v) tree -> ('k * 'v) tree` that inserts the given key and value to the correct position in the tree. Assume that the given tree is a BST and ensure that the result is then also a BST. If there is already a label with the given key in the tree, then the value should be updated. Otherwise a new label must be inserted. (5 points)

g. Define the function `split : 'a tree -> ('a list * 'a * 'a list) option` that decomposes the given tree into three components. The function should evaluate to `Some (ls, a, rs)` if the given tree contains at least one label. Otherwise it should evaluate to `None`. In the first case, `a` should be the label of the root of the tree, `ls` should hold the labels from the nodes that are the left child of their parent, and `rs` should hold the labels from the nodes that are the right child of their parent. The order of elements in the result (in `ls` and `rs`) does not matter but the number of elements (of type `'a`) in the tree and in the result should be the same (i.e., preserve duplicates). (5 points)

This page has been left blank intentionally. Please use it for your answers.