Tallinn University of Technology

Department of Computer Science

Past exam paper: Autumn 2013

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.

Two A4 pages of handwritten notes are permitted.
The print text book (Real World Functional Programming) is allowed.
DO NOT open the examination paper until instructed to do so

Question 1: True or False

Please circle T if the following statement is true and F if the statement is false.

a. (T) (F) The value of List.fold (+) -1 [1;2;3;4] is 9. (2 points)

b. (T) (F) Evaluating the expression ([1;2] :> System.Object) :?> int list will succeed. (2 points)

c. (T) (F) For abitrary n, accessing the last element of [|1 .. n|] will take linear time (as a function of n). (2 points)

d. (T) (F) Evaluating the expression fst (lazy (1,2)) will fail because of type mismatch. (2 points)

e. (T) (F) Taking the head of an empty list will fail at run time. (2 points)

f. (T) (F) Evaluating the expression let x = printfn "hello"; 2 will print hello to the screen (2 points)

g. (T) (F) The type of [(1,2,3)] is (int * int * int) list. (2 points)

h. (T) (F) The type of Some (Some 42) is int option. (2 points)

i. (T) (F) Evaluating the expression Option.bind Some None returns Some None. (2 points)

j. (T) (F) The expression let rec t = seq {yield 1; yield! t} gen-erates an infinite sequence of increasing integers (2 points)

k. (T) (F) The expression List.map (fun n -> n + 2) [1,3,5,7] returns [3,3,5, 7] (2 points)

l. (T)(F) The type definition

```
type Tree =
        | Leaf of int
        | Node of Tree * Tree
```

defines leaf labelled trees (3 points)

a
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
这个函数接受三个参数：
第一个参数是一个二元函数（接受两个参数的函数），用于将累积值和列表中的元素进行组合。
第二个参数是初始的累积值（accumulator），它是在折叠开始时的累积值。
第三个参数是要进行折叠操作的列表。
函数的行为是将列表中的元素从左到右依次传递给二元函数，同时更新累积值。这个过程会持续到列表中的所有元素都被处理完毕。最终，List.fold_left返回的是最终的累积值。

b.试图将整数列表[1;2]先上转换（upcast）为System.Object，然后再尝试将其下转换（downcast）为int list类型。这个操作是不会成功的

([1;2] :> System.Object) 试图将整数列表 [1;2] 上转换为 System.Object 类型。在F#中，这个操作是允许的，因为每个类型在F#中都隐式地继承自 System.Object。
但是，:?> 运算符用于下转换，即将一个基类型转换为派生类型。在这个例子中，你试图将 System.Object 下转换为 int list 类型。这个操作不会成功，因为 System.Object 类型不是 int list 类型的派生类型。因此，下转换会失败，表达式将导致运行时异常。

c 访问数组[|1 .. n|]的最后一个元素的时间是常数时间（constant time）

d 会成功求值，不会因为类型不匹配而导致失败。在这个表达式中，lazy 是一个延迟计算（lazy evaluation）的标记，它表示表达式将被延迟计算，直到它被实际需要的时候。

e 对于一个空列表（empty list），尝试获取其头部元素（head）会在运行时失败

f "hello" 会被打印到屏幕上。在F#中，分号 ; 用于分隔多个表达式，这些表达式会按顺序执行。在这个表达式中，printfn "hello" 和 2 是两个连续的表达式，它们会被依次执行。所以，"hello" 会被打印到控制台上，并且 x 的值将会是 2。

h Some (Some 42) 的类型是 int option option，Some 42 创建了一个整数选项（int option），然后外部的 Some 封装了这个整数选项

i. 对于表达式 Option.bind Some None，它的结果是 Some None。

在F#中，Option.bind 函数用于将一个选项（option）应用到一个函数，该函数的返回值也是一个选项。在这个例子中，Some 是一个包含某个值的选项，而 None 是一个表示空值的选项。当你将 Some 应用到 None 时，它会返回 None，而不会产生运行时错误。所以，Option.bind Some None 的结果是 Some None。

j 它生成一个无限递增整数序列。这是因为在这个递归定义中，t 是一个序列，其中包含了整数1，然后使用 yield! t 表达式来递归地引入了 t 自身，形成了一个无限循环，生成无限递增的整数序列。

k. 在F#中，逗号用于构建元组

l. 定义了一个包含两种类型的树结构：Leaf 构造器用于表示带有整数标签的叶子节点，而 Node 构造器用于表示具有两个子树的内部节点。

Question 2: Trees

一种树形结构的数据表示，其中树的节点可以是字符串（string）或是两个子树的连接（concatenation）

Expressions made of only strings and concatenation can be represented using the following tree data structure:

```
type STree =
  | Val of string
  | Concat of STree * STree
```

a. Define an element of the `STree` type that corresponds to the informal representation `("a" @ "b") @ ("c" @ "d")`. (3 points)

b. Given this function definition:

```
let rec f x =
  match x with
    | Val s -> Val (s.ToUpper())
    | Concat (a,b) -> Concat(a,f b)
```

(i) Evaluate the expression `f (Concat (Concat (Concat (Val "a", Val "b"), Val "c"), Val "d"))`. (3 points)

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

(iii) Explain in words what the function `f` does. (3 points)

c. Write a function that appends an exclamation mark `"!"` to every string in the tree. (5 points)

d. Write a function `flatten : STree -> string` which concatenates all strings in the tree from left to right, e.g. flattening a tree corresponding to `("a" @ "b") @ ("c" @ "d")` would result in a string “abcd”. (5 points)

e. What is the type of the function `g` defined below? (3 points)

```
let rec g f x =
  match x with
   | Val n -> Val (f n)
   | Concat (a,b) -> Concat (g f a,b)
```

```
   Concat
   /   \
Val("ab") Val("cd")
```

a. Concat(Val("ab"), Val("cd"))

b(i) Concat (Concat (Val("ABC"), Val("D")))

b(ii) STree -> STree

(iii) 函数 f 接受一个 STree 类型的输入，然后根据输入的节点类型进行处理：

如果输入节点是 Val s，它将字符串 s 转换为大写，并返回一个新的 Val 节点，表示大写后的字符串。
如果输入节点是 Concat (a, b)，它递归地对子节点 b 调用函数 f，然后用原始的节点 a 和处理过的子节点连接，返回一个新的 Concat 节点。
所以，函数 f 的作用是将输入的树中所有字符串节点转换为大写，并保持树的结构不变。

c.

```
let rec addExclamationMark tree =
  match tree with
  | Val s -> Val (s + "!")
  | Concat (left, right) -> Concat (addExclamationMark left, addExclamationMark right)
```

d

```
let rec flatten tree =
  match tree with
  | Val s -> s
  | Concat (left, right) -> flatten left + flatten right
```

e. ((string -> string) -> STree -> STree)

这个类型表示 g 是一个高阶函数，接受一个从字符串到字符串的函数 f，和一个 STree 类型的输入，并返回一个 STree 类型的输出。它可以将函数 f 应用到 STree 的每个字符串节点上。

Question 3: Lists

a. Define a function `first : 'a list -> 'a` which returns the first element of a list. Define this function using pattern matching. If this list is empty return `failwith "oops"`. (5 points)

b. Given the following function g:

```
let rec g x y =
  match y with
    | a :: b :: cs  -> a :: x b :: g x cs
    | d             -> d
```

(i) Evaluate the expression `g (fun x -> x+1)  [1..4]`. (3 points)

(ii) Give the type of g. (3 points)

(iii) Explain in words what the function g does. (3 points)

c. The function `zip` is supposed to create a list of pairs from the pair of lists given as arguments.

```
let zip (xs,ys) =
  match xs,ys with
     | []      , _         -> []
     | x :: xs', y :: ys' -> (x , y) :: zip (xs,ys)
```

(i) Identify all the bugs. (6 points)

d. Given a function `min : int -> int -> int` which returns the minimum of two arguments, each in the range -100 to 100, write a function `minList : int list -> int` that computes the minimum of a list of integers. (5 points)

a.

```
let first lst =
  match lst with
  | [] -> failwith "oops"
  | hd :: _ -> hd
```

这个函数的作用是将列表 y 中的相邻两个元素进行处理，并将处理结果以交替的顺序连接在一起，其中一个元素经过函数 x 的处理。如果输入的列表元素少于两个，函数将返回原始列表。

(i) [1; 3; 3; 5]

(ii) ('a -> 'b) -> 'a list -> 'b list

(iii) 函数 g 的目的是将输入列表 y 中相邻的两个元素进行处理，其中第二个元素经过函数 x 的处理，然后将它们交替连接起来。如果输入列表元素少于两个，函数将返回原始列表。

模式匹配错误：Pattern Matching Bug:
在模式 (x::xs', y::ys') 中，试图将输入的列表 xs 和 ys 拆分为第一个元素和剩余部分。但是，这与函数的预期行为不符。应该直接匹配输入的列表 xs 和 ys。

修改： 将模式从 (x::xs', y::ys') 改为 (x::xs, y::ys)。

不完整的模式：Non-Exhaustive Patterns:
函数中有两个模式（[], _ 和 (x::xs, y::ys)），但没有处理一个列表为空而另一个不为空的情况。例如，如果 xs 是空的但 ys 不是，模式匹配将无法处理这种情况。

修改： 添加一个模式来处理一个列表为空而另一个不为空的情况。例如，可以添加一个模式类似于 ([], []) -> []，来处理两个列表都为空的情况。

```
let rec zip (xs, ys) =
   match xs, ys with
   | [], [] -> []
   | x::xs, y::ys -> (x, y) :: zip (xs, ys)
   | _, _ -> failwith "Input lists must have the same length"
```

Question 4: Option

There are several 3-valued logics. Kleene 3-valued logic is used in SQL database engines to deal with comparisons involving *null* values.

The behaviour of the 3-valued negation (NOT) can be given as follows:

A	NOT A
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

After noticing that the `option` type adds one value to the set of values of the type it wraps, we decide to use `bool option` type to implement such a logic, with TRUE implemented as `Some true`, FALSE as `Some false` and UNKNOWN as `None`.

a. Write a function that converts from `bool` value to `bool option`. (2 points)

b. Write a function that converts from `bool option` to `bool` (hint: use `failwith "oops"` in the case of `None`). (2 points)

c. Implement the 3-valued negation function `kleeneNeg : bool option -> bool option` in F# by using pattern matching. (4 points)

d. In Kleene logic the behaviour of the disjunction (OR) function can be given by the following table:

A OR B	A = TRUE	A = UNKOWN	A = FALSE
B = TRUE	TRUE	TRUE	TRUE
B = UNKNOWN	TRUE	UNKNOWN	UNKNOWN
B = FALSE	TRUE	UNKNOWN	FALSE

Implement the Kleene 3-valued logic disjuction as `kleeneOr : bool option -> bool option -> bool option`. (5 points)

e. Given that the implication in Kleene logic is defined as $A \rightarrow B = NOT(A)\ OR\ B$, implement Kleene implication as `kleeneImpl : bool option -> bool option -> bool option`. (3 points)

f. In Kleene 3-valued logic it is possible to assign integer values to FALSE = 0, UNKNOWN = 1 and TRUE = 2 and use the built in `min` function to compute the conjunction (AND). For example A AND B = MIN (A,B).

(i) Write a function `kleeneToInt : bool option -> int`. (2 points)

(ii) Write a function `kleeneAnd : bool option -> bool option -> bool option` that computes the conjunction of 2 arguments, and uses the built in `min : int -> int -> int` function. (3 points)

题目涉及到3值逻辑，特别是Kleene 3值逻辑，它在SQL数据库引擎中用于处理涉及空值（NULL values）的比较。在这个逻辑中，有三个可能的取值：TRUE（真），FALSE（假），和Unknown（未知）。为了实现这种逻辑，我们可以使用布尔选项类型（bool option type），其中TRUE被表示为Some true，FALSE被表示为Some false，而Unknown则被表示为None

a

```
let bool_to_option (b: bool) : bool option =
  match b with
  | true -> Some true
  | false -> Some false
```

b

```
let option_to_bool (b_opt : bool option) : bool =
  match b_opt with
  | Some true -> true
  | Some false -> false
  | None -> failwith "oops"
```

c

```
let kleeneNeg (bOpt : bool option) : bool option =
   match bOpt with
   | Some true -> Some false
   | Some false -> Some true
   | None -> None
```

d

```
let kleeneOr (aOpt : bool option) (bOpt : bool option) : bool option =
   match aOpt, bOpt with
   | Some true, _ | _, Some true -> Some true
   | Some false, Some false -> Some false
   | _, _ -> None
```

e

```
let kleeneImpl (aOpt : bool option) (bOpt : bool option) : bool option =
   let notAOpt = kleeneNeg aOpt
   kleeneOr notAOpt bOpt
```

f(i)

```
let kleeneToInt (bOpt: bool option) : int =
   match bOpt with
   | Some true -> 2
   | Some false -> 0
   | None -> 1
```

ITT8060-2013-E2-SAMPLE 5

(2)

```
let kleeneAnd (aOpt: bool option) (bOpt: bool option) : bool option =
   let aInt = kleeneToInt aOpt
   let bInt = kleeneToInt bOpt
   let resultInt = min aInt bInt
   match resultInt with
   | 2 -> Some true
   | 1 -> None
   | 0 -> Some false
   | _ -> failwith "Invalid integer value"
```

g. Given the following function g:

```
let g x = Option.map not x
```

(The type of `Option.map` is `('a -> 'b) -> 'a option -> 'b option` and the type of the Boolean `not` function is `bool -> bool`.)

(i) Give the type of `g`. (2 points)

(ii) Evaluate the expression `g None`. (2 points)

(i) g : bool option -> bool option

(ii) g None = None

(i) 函数 g 的类型可以从它的定义和所使用函数的类型来确定。

Option.map 的类型是 ('a -> 'b) -> 'a option -> 'b option。not 函数的类型是 bool -> bool。在函数定义 g x = Option.map not x 中，x 的类型是 'a option。当 Option.map 被应用于 not 函数时，'a 的类型必须是 bool，因为 not 函数接受一个 bool 类型的参数。

(ii) 对表达式 g None 的求值：

在F#中，Option.map 将一个函数应用于 Option 的内部值（如果是 Some 的话），否则返回 None。在这个情况下，g None 等同于 Option.map not None。

由于 None 中没有值（None 表示值的缺失），将 not 函数或任何函数应用于 None 都会得到 None。因此，g None 的求值结果是：

Tallinn University of Technology

Department of Software Science

Sample exam paper 2019

Advanced Programming

ITT8060

Time allowed TWO hours 30 minutes

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):	

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 answer and write clearly next to the question what the answer is.

a. The expression `List.filter (fun (x,y) -> x>y) [1,2;2,3;3,4;5,0]` returns

A. `[5,0]` B. `[1;2;3;5]` C. `[2;3;4;0]` D. `[1,2;2,3;3,4;5,0]`

a 筛选出列表中所有符合条件 x>y 的元组

b. The value of `0 |> List.fold (>>) id [(+) 1; (*) 3; (-) 1]` is

A. 2 B. -11 C. -2 D. None of the above

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

b 将列表中的函数依次应用到累积器上，使用右复合运算符 (>>)

c. Evaluating the expression `(lazy (1,2))` will return

A. `type error` B. `2` C. `1` D. `None of the above`

c 表示创建一个惰性（lazy）计算的值。在这种情况下，表达式 (1, 2) 的计算会被延迟，直到需要使用这个值的时候才会被计算。

一个懒惰计算的元组 (1, 2) Lazy<int * int>

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

A. `Some false` B. `false` C. `None` D. `Some Some false`

d Option.bind 函数的类型签名为 ('a -> 'b option) -> 'a option -> 'b option。它的作用是将一个 'a option 的值与一个返回 'b option 的函数绑定在一起，如果输入的 'a option 是 Some x，则返回 f x，如果输入是 None，则结果也是 None。

e. The type of the expression
`printfn "nice day!"`
is

A. `string*int` B. `string` C. `unit` D. `int`

e 输出字符串 "nice day!" 到控制台，但它不返回任何有意义的值，因此其类型是unit

f. A function of type `'a list -> 'int list -> 'a` can be applied given the first argument is of type

A. `int list` B. `string list list` C. `all of the above` D. `none of the above`

f 如果有一个函数的类型为 ' a list -> ' b list -> ' a，那么该函数的第一个参数必须是 'a list 类型的，而第二个参数可以是任何类型的 'b list

g. The type of the following function is

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

A. `'a list -> unit` B. `'a list -> 'b list` C. `'a list -> 'a list` D. `int`

g 根据函数的递归结构和模式匹配，该函数的类型应该是 ' a list -> ' a list

h. The type of the expression `let f = fun x -> (fun y -> x+y)` matches the type of the expression

A. `let g x y = x * y` B. `let g x,y = x + y` C. `Both of the above` D. `None of the above`

h
给定的表达式 let f = fun x -> (fun y -> x+y) 创建了一个高阶函数 f，它接受一个参数 x 并返回一个函数，该函数接受参数 y 并返回 x + y 的结果。根据这个描述，f 的类型应该是 'a -> ('a -> 'a)，表示它接受一个 'a 类型的参数并返回一个接受 'a 类型参数的函数

i. Evaluation of the following code will result in

```
let sleepWorkflow = async {
    printfn "starting"
    do! Async.Sleep 2000
    printfn "finished"
}
```

A. A value of type `Async<unit>` being created; B. “starting” and “finishing” being printed; C. A delay of 2000 ms; D. None of the above.

i
创建了一个类型为 Async<unit> 的值。

这段代码定义了一个异步工作流 sleepWorkflow，执行时，它会打印 "starting"，暂停1000毫秒（1秒钟），然后打印 "finished"。但是，仅仅评估所示代码不会执行异步工作流。它仅定义了它并将其赋值给 sleepWorkflow。要执行工作流，你需要使用 Async.RunSynchronously、Async.Start 或其他运行异步工作流的函数。

j. The function `f` defined below

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

A. has type `'a -> 'b list`. B. has type `int -> int list`. C. has type `'a -> seq<'a>`. D. None of the above.

j. 代码定义了一个名为 f 的递归函数，其功能是生成一个无限序列。这个函数接受一个参数 x，然后生成一个序列，这个序列的第一个元素是 x，后面跟着通过递归调用 f 并传入相同的参数 x 所生成的序列的所有元素。

换句话说，这个函数将创建一个包含无限个 x 的序列。

k. Given that the type of `List.reduce` is `(('a -> 'a -> 'a) -> 'a list -> 'a)`, the expression `List.reduce (*) [1]` will evaluate to

A. `1` B. `[1]` C. runtime error; D. none of the above.

k List.reduce 函数在 F# 中用于将列表中的元素通过某个函数进行累积运算，最终得到一个单一的结果

```
 let sum = List.reduce (fun acc x -> acc + x) [1; 2; 3; 4; 5]
printfn "The sum is: %d" sum
```

当列表中只有一个元素时，F# 不会抛出运行时错误，而是直接返回那个单一元素。

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

A. `[[4];[9];[25]]` B. `[30]` C. `[4; 9; 25]` D. None of the above.

l. Apply the function to 2: 2 + 2 = 4, result: [4]
Apply the function to 3: 3 + 3 = 6, result: [6]
Apply the function to 5: 5 + 5 = 10, result: [10]
Concatenate the results: [4; 6; 10]
So, the expression will evaluate to **[4; 6; 10]**.

List.collect 函数在 F# 中的作用是将一个函数应用于列表中的每个元素，并且收集（concatenate）所有结果到一个列表中。其类型签名为 ('a -> 'b list) -> 'a list -> 'b list

m. Given the declaration

```
let rec g x y =
    match y with
    | [] -> 1
    | h :: t -> x h + g x t
```

A. The type of g is `(int -> int) -> int list -> int` B. g is tail recursive C. All of the above D. None of the above

Question 2: Partial functions

A function of type `'a -> 'b option` can be thought of as a function of type `'a -> 'b` that happens to be partial. This means that there may be values of type `'a` where this function is undefined (we cannot produce a value of type `'b`). The type `'a -> 'b option` precisely says that given an `'a` this function may produce a value of type `'b` (failure to produce an output is represented by `None`).

There are library functions `map` and `bind` for `Option` with the given types:

```
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.bind  : ('a -> 'b option) -> 'a option -> 'b option
```

`Option.map` can be used to apply an ordinary function `('a -> 'b)` to an optional value to get an optional result.

`Option.bind` can be used to apply a partial function `('a -> 'b option)` to an optional value to get an optional result.

a. Implement a function `apply` that allows to apply a partial function to an optional value to get an optional result:

```
apply : ('a -> 'b) option -> 'a option -> 'b option
```

For example, the result of `apply (Some id) (Some 3)` must be `Some 3`.

(5 points)

(i) Evaluate `apply (Some ((+) 1)) None` (2 points)

(ii) Evaluate `apply (Some List.head) Some [3;2;1]` (2 points)

b. Implement the function

```
sequence : 'a option list -> 'a list option
```

so that given a list `xs` of optional values it evaluates to

- `Some xs'` when all of the optional values in `xs` were `Some` and `xs'` is a list of the values inside
- `None` when there was at least one `None` in `xs`.

Evaluating `sequence [Some 1; Some 2]` should evaluate to `Some [1; 2]`.

Evaluating `sequence [Some 1; None; Some 2]` should evaluate to `None`.

(7 points)

c. Implement the function

```
sequence' : 'a option list -> 'a list option
```

that behaves according to the specification of `sequence` but is implemented tail recursively.

(7 points)

d. Give the type of the value that results evaluating the expression:

```
[Some [1;2;3]; Some [3;4] ; None] |> sequence
```

(2 points)

部分函数（partial functions）的概念，其中函数的类型为'a -> 'b option，表示这是一个从类型'a到'b的函数，但它可能是不完全定义的，即在某些输入值'a上，该函数可能没有定义（无法产生类型为'b的输出）。这种情况下，函数的返回值是'b option类型，其中None表示没有输出值，而Some x表示有输出值x。

Question 3: Expression trees

Given the following type definitions:

```
type VName = string

type Expr = Var  of VName
          | Neg  of Expr
          | And  of (Expr * Expr)
          | Or   of (Expr * Expr)

type Assignment = (VName * bool) list
```

And the definition of the function `lookup : v:VName -> vs:Assignment -> bool` that looks up a bool value corresponding to the particular variable name VName:

a. Define the function `interpret : e:Expr -> vs:Assignment -> bool` that will determine if an expression e evaluates to true or false given the assingments to the variables in vs. `Neg` represents negation (`not`), `And` represents conjunction (`&&`) and `Or` represents disjunction (`||`).

   For example,
   `interpret (And (Or (Var "X1",Var "X2"), Neg (Var "X2"))) ["X1",true; "X2",false]`
   should evaluate to `true`.

   (5 points)

b. Define a function

   ```
   variables : e:Expr -> VName list
   ```

   that returns all distinct names of the Boolean variables used in the expression. The results should contain distinct values, i.e. each name at most once.

   Hint: consider using `List.sort: ('a list -> 'a list) when 'a : comparison` and/or `List.groupBy: (('a -> 'b) -> 'a list -> ('b * 'a list) list) when 'b : equality` library functions.

   For example, `variables (And (Or (Var "X1", Var "X2"), Neg (Var "X2")))` should evaluate to `["X1";"X2"]`.

   (7 points)

c. Consider the following type of arithmetic expressions:

   ```
   type Expr =
     | Const of int
     | Sum   of Expr * Expr
     | Diff  of Expr * Expr
     | Prod  of Expr * Expr
   ```

   A value `Const n` represents a primitive expression that denotes the number `n`, while values `Sum e1 e2`, `Diff e1 e2`, and `Prod e1 e2`, represent sums, differences, products, respectively.

   Write a function `eval : Expr -> int` that evaluates the given expression.

   (3 points)

d. Write a function `commute : Expr -> Expr` that swaps the arguments of all sums and products in the given expression (which should not change the result of the expression).

   (4 points)

e. Polish notation is a notation for expressions where the operators come before their arguments and no parentheses are used. For example, the expressions `3 + 5`, `3 + (4 * 5)`, and `(3 + 4) * 5` are written `+ 3 5`, `+ 3 * 4 5`, and `* + 3 4 5`, respectively, when using Polish notation.

   Write a function `pn : Expr -> string` that turns the given expression into its representation in Polish notation.

   (6 points)

Question 4: Euler method for numeric approximation

The Euler method can be used to numerically approximate solutions to first-order ordinary differential equations (ODEs) with a given initial value. The ODE has to be provided in the following form:

$$dy(t)/dt = f(t, y(t))$$

with an initial value $y(t_0) = y_0$. This can be numerially approximated with a formula

$$y_{n+1} = y_n + hf(t_n, y_n)$$

An F# implementation of the function is given as `eulerStep` below. We can use the implementation to approximate the Newton's law of cooling where the cooling constant is 0.05 and target temperature is 22.0° C.

The function `newtonNext1` will compute the next pair of `t` and `y` where the returned `y` element in the pair represents the temperature one second later (the time instance is represented by `t`) and target temperature being 22.0° C.

```
let eulerStep f (h : float) t y =
   ((t + h), (y + h * (f t y)))

let newton t y = -0.05 * (y - 22.0)

let newtonNext1 p =
    let (t0,y0) = p
    eulerStep newton 1.0 t0 y0
```

a. Write a function `euler : (float * float) -> (float*float) seq` that for any given pair `(t,y)` computes the infinite sequence consisting of the pairs `(t,y)`, `newtonNext1 (t,y)`, `newtonNext1 (newtonNext1 (t,y))`, and so on. Use sequence expressions in your implementation.

   For initial values, where $y > 22.0$, the sequence will represent the approximation of Newton's law of cooling of an object down to 22.0° C.

   (4 points)

b. Write a function `euler' : (float*float) -> (float*float) seq` that for any given pair `(t,y)` computes the same sequence as `euler (t,y)`. Use the `Seq.unfold` function in your implementation. The type of `Seq.unfold` is `(('a -> ('b * 'a) option) -> 'a -> seq<'b>)`.

   (4 points)

c. The cooling curve is interesting until a small margin ϵ from the target temperature. We want to get the part of the cooling approximation sequence, where the value of y is more than ϵ larger than the target value (22.0 in this case).

   Write a function `coolingApprox : float -> (float * float) seq -> (float * float) list` that turns a given sequence into its prefix that ends as soon as $y \leq 22.0 + \epsilon$. The ϵ is given as the first argument to the `coolingApprox` function. Hint: access the sequence step-by-step e.g. by accessing `Seq.head` and `Seq.tail` and comparing the y value to the margin.

   (7 points)

d. Write a function `coolingApprox' : float -> (float * float) seq -> (float * float) list` that for any given ϵ `epsilon` and sequence `s` computes the same list as `coolingApprox epsilon s`. Make sure that all recursive calls in the definition of `coolingApprox'` are tail calls.

   (10 points)

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"]`

`List.fold : ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State`

1. `List.fold` 是一个函数，它接受一个二元函数、一个初始累积器值和一个列表，然后使用该二元函数将列表中的每个元素与累积器值组合起来。

2. `(>>)` 是一个函数组合运算符，它将两个函数组合成一个新的函数。例如，`(f >> g) x` 等同于 `g(f(x))`。

3. `id` 是一个恒等函数，它返回其参数不变。例如，`id x` 返回 `x`。

4. `[(-)3; (*)2]` 是一个函数列表。列表中的第一个函数 `(-)3` 是一个将其参数减去3的函数，第二个函数 `(*)2` 是一个将其参数乘以2的函数。

5. `1` 是`List.fold`的初始累积器值。

现在，让我们看看这个代码是如何工作的：

1. `List.fold` 从初始累积器值 `id` 开始。

2. 对于列表中的每个函数，它使用 `(>>)` 将累积器与该函数组合起来。这意味着每个新函数都会在前一个函数的结果上应用。

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
```

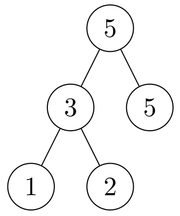
The function check is supposed to check if a given list of triples is a valid representation of a bag, that is, it does not contain any key more than once. The code contains bugs. Identify all of them. (5 points)

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.