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]`

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)`)

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

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

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`

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

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

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`

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`

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`

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.

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.

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.

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.

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)

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)