Written Examination, May 27th, 2024 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: No Aid

The problem set consists of 5 problems which are weighted approximately as follows: Problem 1: 40%, Problem 2: 25%, Problem 3: 5% Problem 4: 20% Problem 5: 10%

Marking: 7 step scale.

In your programs you are allowed to introduce helper functions; but you must also provide a declaration for each of the required functions, so that it has exactly the type and effect asked for.

If a sub-question requires you to define a particular function, then you may use that function in subsequent sub-questions, even if you have not managed to define it yourself.

You are, in general, allowed to use the .NET library including the modules described in the textbook, e.g., List, Set, Map, Seq, etc. But be aware of the special condition stated in Problem 1.

You are not allowed to use imperative features, like assignments, arrays and so on, in your solutions.

... Continued on next page

Problem 1 (40%)

Questions 1 to 5 in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`.

A small Danish ferry, sailing between Denmark and Sweden, has a shop where passengers can buy *items*. The shop maintains a *register* where items are associated with *prices* given in Danish kroner (abbreviated DKK). The shop supports just one other *currency*: Swedish krone (abbreviated SEK). The shop maintains a *rate rt* describing how much 1 DKK is worth in Swedish kroner. A *purchase* is a list of items and a *bill* comprises a corresponding list of item-price pairs and the total price for all items bought. The prices of a bill can either be in Danish kroner or in Swedish kroner. Currencies are not mixed in a bill. The following type declarations model the involved concepts:

```
type Currency  = | DKK             // Danish krone
                 | SEK             // Swedish krone
type Price     = float             // Expected to be positive
type Item      = string
type PricedItem = Item * Price
type Register  = PricedItem list   // Items are unique in a register
                                   // Prices in DKK

type Purchase  = Item list
type Bill      = PricedItem list * Price * Currency
type Rate      = float             // DKK to SEK

let r   = 1.56                     // 1 DKK is 1.56 SEK
let reg = [("Ragusa",22.0); ("Bounty",57.0); ("Licorice",44.0)];;
```

Furthermore, there is a declaration of a rate `r` and a register `reg` containing prices for three items where the prices are given in Danish kroner.

1. Every price occurring in a register must be positive.
   Declare a function `pricesOK: Register -> bool` that checks this property.

You can from now on assume that items are unique in a register and that prices are positive.

2. Declare the following two functions:

   1. The function: `getPrice: Item -> Register -> Price`, that gets the price for an item in a register. The price is in DKK. A suitable exception is raised when the item does not occur in the register.
   2. The function `priceOf: Item -> Register -> Currency -> Rate -> Price`. The value of `priceOf` *itm reg curr rt* is the price of *itm* in *reg* given in currency *curr* using rate *rt*. For example, `priceOf "Bounty" reg SEK r` = 88.92 and `priceOf "Ragusa" reg DKK r` = 22.0.

 ... Continued on next page

3. Declare a function

```
pricedItemsOf: Purchase -> Register -> Currency -> Rate -> PricedItems
```

so that `pricedItemsOf` *pur reg curr rt* is the list of priced items for the items in purchase *pur* using register *reg*. The prices are given in currency *curr* using rate *rt*.
For example, `pricedItemsOf ["Bounty"; "Ragusa"; "Bounty"] reg DKK r` is the list `[("Bounty", 57.0); ("Ragusa", 22.0); ("Bounty", 57.0)]`.

4. Declare a function

```
totalPrice: Purchase -> Register -> Currency -> Rate -> Price
```

so that `totalPrice` *pur reg curr rt* returns the sum of the prices of items in purchase *pur* using register *reg*. The total price is given in currency *curr* using rate *rt*.
For example, `totalPrice ["Bounty"; "Ragusa"; "Bounty"] reg DKK r` is 136.0.

5. Declare a function

```
mkBill: Purchase -> Register -> Currency -> Rate -> Bill
```

so that `mkBill` *pur reg curr rt* is the triple $(pitms, tp, curr)$, where *pitms* is the list of priced items for the items in purchase *pur* using register *reg*, and *tp* is the total price for the purchased items. Prices are given in currency *curr* using rate *rt*.
For example, `mkBill ["Bounty"; "Ragusa"; "Bounty"] reg DKK r` is `([("Bounty", 57.0); ("Ragusa", 22.0); ("Bounty", 57.0)], 136.0, DKK)`.

6. Make non-recursive declarations of

   1. the function `pricesOK` from Question 1,
   2. the function `pricedItemsOf` from Question 3 and
   3. the function `totalPrice` from Question 4.

   You may use the function `priceOf` in your declarations and you may select among the following functions from the `List` library in your declarations.

```
List.forall   : ('a -> bool) -> 'a list -> bool
List.fold     : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.map      : ('a -> 'b) -> 'a list -> 'b list
```

7. We now consider an extension where more currencies are supported, for example: Norwegian kroner, Euro, British pounds, etc. Therefore, currencies are now described by strings:

```
type Currency = string // E.g. "DKK", "SEK", "NOK", "EUR", ....
```

   Give a new declaration for the type `Rate` that captures this extension and revise the declaration `priceOf` accordingly. Justify your choice of type briefly.

Problem 2 (25%)

Consider the following declaration of a function `f`:

```
let rec f xs i = match xs with
                 | x::rest -> (x,i)::f rest (i+1)
                 | []      -> [];;
```

1. State the most general type of `f`. (Notice that any other type of `f` is an instance of the most general type.) Furthermore, present an argument showing why your answer is the most general type.
2. Describe what `f` is computing. Your description should focus on what it is computing rather than on how the computations are performed.
3. Make an evaluation of the expression `f [3;4;5;6] 10`.
4. The declaration of `f` is not tail recursive. Explain briefly why and make a tail-recursive variant of `f` that is based on an accumulating parameter.
5. Make a continuation-based tail-recursive variant of `f`.

Problem 3 (5%)

Consider the declaration of two functions `p1` and `f1`:

```
let p1 x = x>0;;
let rec f1 xys =
   match xys with
   | (x,y)::rest when p1 x      -> x :: f1 rest
   | (x,y)::rest when not(p1 x) -> y :: f1 rest
   | []                         -> [];;

> val p1: x: int -> bool
> val f1: xys: (int * int) list -> int list

>  match xys with
> -------^^^

> warning FS0025: Incomplete pattern matches on this expression.
```

1. The compiler succeeds; but it issues a warning. Explain what caused the warning from the compiler. Should the program be revised?

Problem 4 (20%)

Consider the following tree type `E` for expressions constructed from *constants* (constructor `N`), *variables* (constructor `V`) and *applications* (constructor `App`):

```
type E = N of int | V of string | App of string * EList
and  EList = E list;;

let ex1 = App("f", [N 1; V "x"])
let ex2 = App("g", [ex1; V "y"; V "x"])
```

For an application `App`$(f, [e_1; \ldots; e_n])$, where $n \geq 0$, f is the name (a string) of a *function* that is applied to a (possibly empty) list of expressions $[e_1; \ldots; e_n]$. In the declaration of `ex1` above "f" is applied to a list comprising the constant 1 and the variable "x". In the declaration of `ex2`, "g" is applied to a list with three expressions. The first expression in this list is an application, the second is the variable "y", and the third the variable "x".

Hint: You may consider using mutually recursive function declarations when solving the questions 1., 2. and 3. below.

1. Declare a function `vars: E -> string list` that gives the list af all variables occurring in an expression. The sequence in which variables occur in the list is of no concern. Furthermore, a variable may occur multiple times in the list.

2. Declare a function `fnames: E -> string list` that gives the list af all function names occurring in an expression. For example, "f" and "g" are the two function names occurring in `ex2`.

3. Declare a function `subst: string -> int -> E -> E` so that the value of `subst` x n e is the expression obtained from e by replacing every occurrence of the variable x by the constant n.

4. State the value of `subst "x" 2 ex2`.

Problem 5 (10%)

Consider the following declaration of `f2`. This declaration causes an error message from the F# compiler:

```
let rec f2 xs ys =
   match xs with
   | []      -> ys
   | x::rest -> f2 rest x::ys;;

> | x::rest -> f2 rest x::ys;;
> -------------^^^^^^^^^

> error FS0001: Type mismatch. Expecting a ''a'
> but given a ''a list'
> The types ''a' and ''a list' cannot be unified.
```

1. Explain what caused the error message and make a correction that would remove the error. (A small correction suffices.)

Consider now the following declaration of a type `T` for binary trees and a function `f3`. These declarations are followed by an error message from the F# compiler.

```
type T = A of int | B of T * T;;

let rec f3 t = match t with
               | T i      ->  i
               | T(t1,t2) -> f3 t1 + f3 t2;;
> ---------------^
> The pattern discriminator 'T' is not defined.
```

2. Explain what caused the error message. The intention is that `f3` should return the sum of all integers occurring in a tree of type `T`. Make corrections that would remove the error.