Problem 1 (30%)

We consider the use of *appliances* (in Danish 'husholdningsapparater') like washing machines, dishwashers and coffee machines. A *usage* of an appliance a is a pair (a, t), where t is the time span (in hours) the appliance is used. A *usage list* is a list of the individual usages during a full day, that is, 24 hours. This is modelled by:

```
type Appliance = string
type Usage     = Appliance * int

let ad1 = ("washing machine", 2)
let ad2 = ("coffee machine", 1)
let ad3 = ("dishwasher", 2)
let ats = [ad1; ad2; ad3; ad1; ad2]
```

where `ats` is a value of type `Usage list` containing one usage of the dishwasher and two usages of the washing machine and the coffee machine.

1. Declare a function: `inv: Usage list -> bool`, that checks whether all time spans occurring in a usage list are positive.

2. Declare a function `durationOf: Appliance -> Usage list -> int`, where the value of `durationOf` *a ats* is the accumulated time span appliance a is used in the list *ats*. For example, `durationOf "washing machine" ats` should be 4.

3. A usage list *ats* is *well-formed* if it satisfies `inv` and the accumulated time span of any appliance in *ats* does not exceed 24. Declare a function that checks this well-formedness condition.

4. Declare a function `delete`(a, ats), where a is an appliance and *ats* is a usage list. The value of `delete`(a, ats) is the usage list obtained from *ats* by deletion of all usages of a. For example, deleting usage of the coffee machine from `ats` should give `[ad1; ad3; ad1]`. State the type of `delete`.

We now consider the *price* of using appliances. This is based on a *tariff* mapping an appliance to the price for one hour's usage of the appliance:

```
type Price  = int
type Tariff = Map<Appliance, Price>
```

5. Declare a function `isDefined` *ats trf*, where *ats* is a usage list and *trf* is a tariff. The value of `isDefined` *ats trf* is true if and only if there is an entry in *trf* for every appliance in *ats*. State the type of `isDefined`.

6. Declare a function `priceOf: Usage list -> Tariff -> Price`, where the value of `priceOf` *ats trf* is the total price of using the appliances in *ats*. The function should raise a meaningful exception when an appliance is not defined in *trf*.

 ... Continued on next page

Problem 1 (20%)

1. Declare a function: `repeat: string -> int -> string`, so that `repeat` $s\,n$ builds a new string by repeating the string s altogether n times. For example: `repeat "ab" 4 = "abababab"` and `repeat "ab" 0 = ""`.

2. Declare a function `f` $s_1\,s_2\,n$ that builds a string with n lines alternating between s_1 and s_2. For example: `f "ab" "cd" 4 = "ab\ncd\nab\ncd"` and `f "XO" "OX" 3 = "XO\nOX\nXO"`. Note that `\n` is the escape sequence for the newline character.
   Give the type of the function.

3. Consider now certain patterns generated from the strings `"XO"` and `"OX"`. Declare a function `viz` $m\,n$ that gives a string consisting of n lines, where
   - the first line contain m repetitions of the string `"XO"`,
   - the second line contain m repetitions of the string `"OX"`,
   - the third line contain m repetitions of the string `"XO"`,
   - and so on.

   For example, `printfn "%s" (viz 4 5)` should generate the following output
   ```
   XOXOXOXO
   OXOXOXOX
   XOXOXOXO
   OXOXOXOX
   XOXOXOXO
   ```

4. Reconsider the function `repeat` from Question 1.
   1. Make a tail-recursive variant of `repeat` using an accumulating parameter.
   2. Make a continuation-based tail-recursive variant of `repeat`.

Problem 2 (20%)

1. Declare a function `mixMap` so that

$$\texttt{mixMap}\ f\ [x_0; x_1; \ldots; x_m]\ [y_0; y_1; \ldots; y_m] = [f(x_0, y_0); f(x_1, y_1); \ldots; f(x_m, y_m)]$$

2. Declare a function `unmixMap` so that

$$\texttt{unmixMap}\ f\ g\ [(x_0, y_0); (x_1, y_1); \ldots; (x_n, y_n)] = ([f\,x_0; f\,x_1; \ldots; f\,x_n], [g\,y_0; g\,y_1; \ldots; g\,y_n])$$

3. Give the most general types for `mixMap` and `unmixMap`.

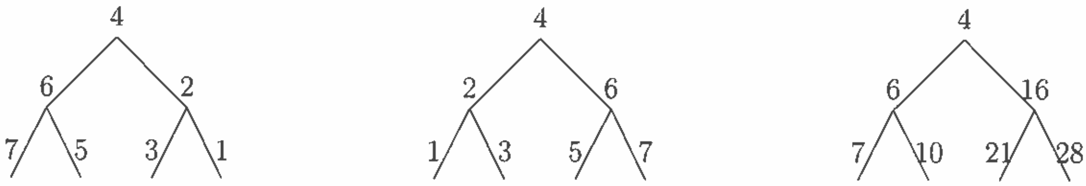
 ... Continued on next page

Problem 3 (30%)

Consider the following F# declarations of a type for binary trees and a binary tree `t`:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;

let t = Br(Br(Br(Lf,1,Lf),2,Br(Lf,3,Lf)),4,Br(Br(Lf,5,Lf),6,Br(Lf,7,Lf)));;
```



`reflect t` The tree `t` `accumulate t`

An illustration of the tree `t` is given in the middle part of the above figure. The left part of the figure shows the reflection of `t`, that is, a mirror image of `t` formed by exchanging the left and right subtrees all the way down.

1. Declare a function `reflect` that can reflect a tree as described above.

The right part of the figure shows a tree obtained from `t` by accumulating the values in the nodes of `t` as they are visited through a pre-order traversal. For example, the values in the nodes of `t` are visited in the sequence: 4, 2, 1, 3, 6, 5, 7. Hence, the node of `accummulate t` corresponding to the node of `t` with value 3, has value $10 = 4+2+1+3$.

2. Declare a function `accumulate` that can accumulate the values in a tree as described above. Hint: You may declare an auxiliary function having an accumulating parameter.

Consider now the following declarations:

```
let rec k i t =
   match t with
   | Lf           -> Lf
   | Br(tl,a,tr) -> Br(k (i*i) tl, i*a, k (i*i) tr);;

let rec h n m =
   function
   | Br(tl,a,tr) when n=m -> h n 1 tl @ [a] @ h n 1 tr
   | Br(tl,_,tr)          -> h n (m+1) tl @ h n (m+1) tr
   | Lf                   -> []

let q n t = h n n t;;
```

3. Give the most general types of `k` and `q` and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

 ... Continued on next page