CSE-321 Assignment 3 - *Module System* (100 Points)

gla@postech.ac.kr

Due at 11:59pm, March 12

From this programming assignment onward, we will not accept submissions by email under any circumstances. Do not submit your assignment at the last minute before the deadline!

In this assignment, you will explore the module system of ML, in particular, functors and their applications. This assignment consists of two stages. In the first stage, you will write a functor `Vector`, which can be applied to `SCALAR` structures. To be specific, you will first learn the signature `SCALAR`, which declares a set of operations and values necessary to introduce a scalar type. Then you will write a functor `Vector` which builds a `VECTOR` structure from a `SCALAR` structure. Then, you will write yet another functor `Matrix` to facilitate the computation on square matrices. In the second stage, you will solve three graph problems. First, you will write the functor `Closure` which computes the closure of a given matrix by using the functors you wrote in the first stage. Finally, you will write code to solve the three graph problems by exploiting the functor `Closure`.

Remark

- Before you start, make sure that you have read Chapter 2 *The module system* of OCaml manual.
- You may use any functions in the OCaml library.
- You may not use mutable references and arrays of OCaml. For this assignment, you do not need them anyway.

1 Part I (65 points)

1.1 Scalar (10 points)

A `SCALAR` structure is used to manipulate a specific scalar type. For example, `boolean`, `int`, and `float` are all scalar types. The following signature `SCALAR` declares a set of functions on the type `t` and relevant values.

```
module type SCALAR =
sig
  type t                               (* scalar element *)

  (* ScalarIllegal is raised when an operation cannot be performed on given
    scalar elements. *)
  exception ScalarIllegal

  (* identity zero for ++ :
    zero ++ x = x ++ zero = x for any value x of type t. *)
  val zero : t
  (* identity one for ** :
    one ** x = x ** one = x for any value x of type t. *)
  val one : t

  (* infix addition :
    ++ is associative and commutative :
    x ++ (y ++ z) = (x ++ y) ++ z
    x ++ y = y ++ x
    zero is the identity for ++. *)
  val (++) : t -> t -> t

  (* infix multiplication :
    ** is associative :
    x ** (y ** z) = (x ** y) ** z
    one is the identity for **.
    ** does not have to be commutative. *)
  val ( ** ) : t -> t -> t

  (* infix equality test :
    x ==  =  true if x = y *)
  val (==) : t -> t -> bool
end
```

As you see in the above code, there are some invariants associated with each function and value. Thus, whenever you implement a `SCALAR` structure, you need to make sure that all the functions and values satisfy the invariants specified in the signature. Since all the subsequent functors will exploit these invariants, failure to meet the constraints would result in faulty structures being yielded by the functors.

Please write a structure `Boolean` which conforms to the signature `SCALAR`.

```
module Boolean : SCALAR with type t = bool
=
struct
  ...
end
```

1.2 Vector (20 points)

A `SCALAR` structure can represent only a single scalar type. Thus, for each scalar type you wish to use, you need to write a new corresponding `SCALAR` structure. For example, you should write a particular `SCALAR` structure for `int` and another `SCALAR` structure for `float`.

A vector is a collection of values of the same type, which we refer to as elements. Regardless of its element type, the basic operations on vectors can be written in a consistent way as long as we are provided with appropriate operations on elements. For instance, we can exploit the same algorithm for computing the sum of two vectors whatever their element type is. Thus, it would be desirable to generate a structure for a particular vector type by giving the basic operations on its element type to some 'generator', which turns out to be a functor in OCaml. In other words, if you write a functor encapsulating all the operations on vectors, we can generate a structure of any vector type by providing the functor with a set of operations on its element type.

In this problem, you will write a functor `Vector` which takes a `SCALAR` structure and generates a corresponding `VECTOR` structure. The signature `VECTOR` is shown below. The type `elem` is for elements of a particular vector type, and the type `t` is the resultant vector type. Since we need to perform some operations on elements in manipulating vectors, we require that the type `elem` satisfies the following condition: it has an addition operation (associative and commutative), a multiplication operation, an equality operation, an identity for addition, and an identity for multiplication.

```
module type VECTOR =
sig
  type elem                              (* vector elements *)
  type t                                 (* vector *)

  (* VectorIllegal is raised when an operation cannot be performed on given
    vectors. *)
  exception VectorIllegal

  (* vector creation :
    create takes a list of scalar elements and returns a corresponding vector:
    create [x0; ...; xi; ...; x(n-1)] =
      a vector /x0; ...; xi; ...; x(n-1)/ whose i-th element is xi.
    VectorIllegal is raised if the list is empty. *)
  val create : elem list -> t

  (* to_list :
    toList takes a vector and converts it to a list of scalar elements.:
    toList /x0; ...; xi; ...; x(n-1)/ =
      [x0; ...; xi; ...; x(n-1)] whose i-th element is xi. *)
  val to_list : t -> elem list

  (* dimension :
    dim v returns the dimension of v, that is, the number of scalar elements in v. *)
  val dim : t -> int

  (* extraction :
    nth v n returns the n-th scalar element of v, and indices begin at 0.
    If n is out of range, raises VectorIllegal. *)
  val nth : t -> int -> elem

  (* addition :
    ++ is associative and commutative:
    x ++ (y ++ z) = (x ++ y) ++ z
    x ++ y = y ++ x
    VectorIllegal is raised if x and y have different dimensions. *)
  val (++) : t -> t -> t

  (* equality test :
    x == y = true iff x = y.
    VectorIllegal is raised if x and y have different dimensions. *)
  val (==) : t -> t -> bool

  (* inner product :
    inner x y returns the inner product of x and y by using ++ and **
    operations for type elem.
    inner /x0; ...; xn/ /y0; ...; yn/ =
      (x0 ** y0) ++ (x1 ** y1) ++ (xn ** yn)
    VectorIllegal is raised if x and y have different dimensions. *)
  val innerp : t -> t -> elem
end
```

Note that the signature VECTOR does not impose any restriction on the number of elements in a vector, which we call the length (or dimension) of the vector. A typical operation on vectors requires that its arguments have the same length. For instance, the addition of two vectors cannot be defined unless they have the same length.

Please write a functor VectorFn with the following specification:

```
module VectorFn (Scal : SCALAR) : VECTOR with type elem = Scal.t
=
struct
  ...
end
```

Note that you need to choose your own representation scheme for t. We recommend you to think about the representation scheme for vectors before implementing all the other parts.

- **Warning** : If you do not implement the nth function correctly, you may lose not only the credit for nth but also the credit for other functions because grading will be done by using your nth function.

1.3 Matrix (35 points)

The signature MATRIX shown below defines a set of operations and values to manipulate square matrices, which have the *same number* of rows and columns.

```
module type MATRIX =
sig
  type elem
  type t

  exception MatrixIllegal

(* matrix creation :
   create takes a list of elem lists and returns a corresponding matrix.
   The list of elem lists is given in increasing order of matrix rows,
     and each elem list is given in increasing order of matrix columns.
   Therefore, each elem list must have the same size as the list of elem lists.
   MatrixIllegal is raised if each elem list has a different size or the input
     list is empty. *)
  val create : elem list list -> t

  (* identity :
    identity takes a dimension and returns an identity matrix which has a corresponding
      dimension.
    MatrixIllegal is raised if dimension <= 0 *)
  val identity : int -> t

  (* dimension :
    dim m returns the dimension of m, that is, the number of rows or columns in v. *)
  val dim : t -> int
  (* transpose :
    transpose m returns the transpose matrix of m. *)
  val transpose : t -> t

  (* to_list :
    to_list takes a matrix and returns a corresponding list. *)
  val to_list : t -> elem list list

  (* extraction :
   (get m r c) returns the content of m at row r and column c.
   Indices begin at 0.
   MatrixIllegal is raised if either r or c is out of range. *)
  val get : t -> int -> int -> elem

  (* matrix addition :
    MatrixIllegal is raised if the dimension of two matrices is different. *)
  val (++) : t -> t -> t

  (* matrix multiplication :
    MatrixIllegal is raised if the dimension of two matrices is different. *)
  val ( ** ) : t -> t -> t

  (* equality test :
    MatrixIllegal is raised if the dimension of two matrices is different. *)
  val (==) : t -> t -> bool
end
```

`elem` should be distinguished from `t`; it denotes the type of entries of matrices as opposed to the type of matrices, which is `t`. `create` returns a matrix from a list of `elem` lists which is given in row major order.

For instance, with `elem = int`,

```
create [[1; 2; 3]; [-1; -2; -3]; [0; 1; 0]]
```

returns a 3 by 3 matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ -1 & -2 & -3 \\ 0 & 1 & 0 \end{pmatrix}$$

`dim` returns the dimension of the argument. `transpose` returns the transpose matrix of the argument. For instance, If $\mathtt{m} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$, `transpose m` will return $\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$.

Please write a functor `MatrixFn` with the following specification:

```
module MatrixFn (Scal : SCALAR) : MATRIX with type elem = Scal.t
=
struct
  ...
end
```

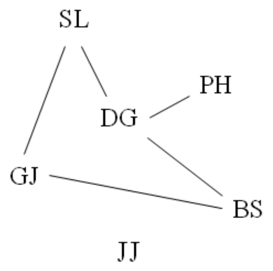
As shown in `with` clause, the structure `Scal` specifies the type of entries of matrices. You need to choose your own representation scheme for matrices.

Hint: You can use the functor `VectorFn` that you wrote in the first part.

- **Warning** : If you do not implement the `get` function correctly, you may lose not only the credit for `get` but also the credit for other functions because grading will be done by using your `get` function.

2 Part II (35 points)

Consider the graph shown below. The vertices represent six cities in Korea and the edges represent roads connecting these cities. For instance, there is no direct route from `GJ` to `DG`, and `JJ` is an isolated city to which there is no route from any other cities.
`SL` = Seoul, `DG` = Daegu, `PH` = Pohang, `GJ` = Gwangju, `BS` = Busan, `JJ` = Jeju



How can we represent this graph with a matrix ? The idea is that we number the six cities from 0 to 5 and build a square matrix of size 6 by assigning true or false to each entry of the matrix according to whether there exists a direct route between the two corresponding cities. With numbering `JJ` = 0, `SL` = 1, `GJ` = 2, `DG` = 3, `BS` = 4, and `PH` = 5, for instance, the resultant matrix D, which we call an adjacency matrix, would be:

$$D = \begin{pmatrix} \texttt{true} & \texttt{false} & \texttt{false} & \texttt{false} & \texttt{false} & \texttt{false} \\ \texttt{false} & \texttt{true} & \texttt{true} & \texttt{true} & \texttt{false} & \texttt{false} \\ \texttt{false} & \texttt{true} & \texttt{true} & \texttt{false} & \texttt{true} & \texttt{false} \\ \texttt{false} & \texttt{true} & \texttt{false} & \texttt{true} & \texttt{true} & \texttt{true} \\ \texttt{false} & \texttt{false} & \texttt{true} & \texttt{true} & \texttt{true} & \texttt{false} \\ \texttt{false} & \texttt{false} & \texttt{false} & \texttt{true} & \texttt{false} & \texttt{true} \end{pmatrix}$$

We can also assign an integer to each edge so that the resultant matrix represents the distance matrix which records the time taken to drive from one city to another city. An example of such a matrix, which we call a distance matrix, is:

$$E = \begin{pmatrix} 0 & -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & 100 & 200 & -1 & -1 \\ -1 & 50 & 0 & -1 & 150 & -1 \\ -1 & 100 & -1 & 0 & 100 & 25 \\ -1 & -1 & 50 & 100 & 0 & -1 \\ -1 & -1 & -1 & -1 & -1 & 0 \end{pmatrix}$$

-1 indicates that there exists no direct road from one city from another. Notice that the distance matrix E is not symmetric. For instance, E tells us that there is a road from `DG` to `PH` (25 units long) but not the other way. Also, it takes 50 units to drive from `GJ` to `SL`, but it takes twice as long to drive the other way.

Another way to assign values to edges is by interpreting each value as the maximum weight limit on cars that use the corresponding road. An example of such a matrix, which we call a weight matrix, is:

$$F = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 100 & 100 & 0 & 0 \\ 0 & 50 & -1 & 0 & 100 & 0 \\ 0 & 100 & 0 & -1 & 200 & 300 \\ 0 & 0 & 50 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

0 indicates that the maximum weight permitted is zero, which means that no cars can take the road or there is no road. -1 is a special constant which means that there is no limit on the weight. Note that in this particular weight matrix, there is no weight limit on the road from `BS` to `DG`.

Now we want to answer such questions as:

1. Can we reach `BS` by driving from `SL`?
2. At least how long does it take to drive to `BS` starting in `SL`, if we can ?
3. I am moving from `SL` to `BS` and want to drive my truck. How much can I load into my truck (without worrying about paying hefty fines)?

All these questions can be answered by simply computing the closure of the matrices with an appropriate definition of matrix operations.

2.1 Closure (10 points)

The closure of a square matrix A is defined as $I + A + A^2 + A^3 + \cdots$ where $I(= A^0)$ is an identity matrix. Depending on the definition of the addition and multiplication operations, A may or may not have its closure. Alternatively, the closure of A can be defined as $I + A + A^2 + \cdots + A^i$ where i is the first positive integer such that $I + A + A^2 + \cdots + A^i$ is equal to $I + A + A^2 + \cdots + A^i + A^{i+1}$.

To see why, note that:
if $I + A + A^2 + \cdots + A^i = I + A + A^2 + \cdots + A^i + A^{i+1}$,
$I + A + A^2 + \cdots + A^i + A^{i+1} + A^{i+2}$
$= I + A(I + A + A^2 + \cdots + A^i + A^{i+1})$
$= I + A(I + A + A^2 + \cdots + A^i)$
$= I + A + A^2 + \cdots + A^i + A^{i+1}$
$= I + A + A^2 + \cdots + A^i$ (by the assumption).

Using a mathematical induction, you can show that $I + A + A^2 + \cdots + A^i$ is actually the closure of A. Thus, you can write a recursive function computing the closure of A as follows:

```
let closure current_closure =
  if current_closure is equal to (I + current_closure {*} A) then
    return current_closure
  else
    closure (I + current_closure {*} A)
(* with the initial invocation 'closure one'. *)
```

Please fill in the following functor `ClosureFn`:

```
module ClosureFn (Mat : MATRIX) :
sig
  (* closure computes the closure of a square matrix.
    If the argument has a closure, the evaluation must terminate.
    If not, the evaluation may not terminate. *)
  val closure : Mat.t -> Mat.t
end
```

```
=
struct
  ...
end
```

2.2 Graph Problems (25 points)

2.2.1 Reachability (5 points)

Consider an adjacency matrix D. Each entry of D specifies whether there exists a direct route between two cities. We assume that there exists a direct route within each city; hence, all the diagonal entries are true. Now, we want to obtain a reachability matrix D^*which specifies whether we can drive from one city to another. It turns out that D^* is simply the closure of D with *boolean-or* as the addition operation and *boolean-and* as the multiplication operation. To see why, suppose that D^i represents the reachability matrix when we are allowed to take i roads (we may take the same road several times). For instance, since we can get to `DG` from `GJ` via `SL`, the corresponding entry in D is false whereas the corresponding entry in D^2 is true. To clarify this, suppose that we are computing D^{i+1} and we know that D^i is the reachability matrix when we are allowed to take i roads. An entry for $city\,X$ and $city\,Y$ in $D^{i+1} = D^i \times D$, is computed as

$$D^{i+1}[city\,X,\, city\,Y] = \sum_{for\,each\,city\,Z} D^i[city\,X,\, city\,Z] \,\times\, D[city\,Z,\, city\,Y].$$

Intuitively, in order to get to $city\,Y$ from $city\,X$ by taking $i+1$ roads, we must pass through a $city\,Z$ in such a way that we first take i roads up to $city\,Z$ and take another road to $city\,Y$. This observation is exactly captured by the above formula: in order for $D^{i+1}[city\,X,\, city\,Y]$ to be true, there must be at least one $city\,Z$ such that both $D^i[city\,X,\, city\,Z]$ and $D[city\,Z,\, city\,Y]$ are true.

Please define a function :

```
reach : bool list list -> bool list list
```

which computes the closure of the argument. The argument to `reach` and its return value have the same format as the argument to function `create` in signature `MATRIX`. `reach` raises an exception `IllegalFormat` if its argument is not valid.

Here are some hints for you.

- Use `BoolMatClosure.closure` (See `hw3.ml` for the definition of `BoolMatClosure.closure`).
- You can test your solution by comparing `solution_al'` and `reach al`.

2.2.2 Shortest distance (10 points)

Now, given a distance matrix E, we want to compute the shortest distance matrix E^* whose entry denotes the shortest distance between each pair of cities. As suggested above, we can obtain E^* by computing the closure of E. However, the meaning of addition and multiplication may have to be modified accordingly. `zero` and `one` should be defined appropriately so as not to invalidate the meaning of the identity matrix I.

First, fill in the following structure `Distance`:

```
module Distance : SCALAR with type t = int
=
struct
  type t = int

  exception ScalarIllegal
  let zero = 999999              (* Dummy value : Rewrite it! *)
  let one = 999999               (* Dummy value : Rewrite it! *)

  let (++) _ _ = raise NotImplemented
  ...
end
```

Second, write a function `distance` which computes the shortest distance matrix from a distance matrix :

```
val distance : int list list -> int list list
```

The argument to `distance` and its return value have the same format as the argument to function `create` in signature `MATRIX`. `distance` raises an exception `IllegalFormat` if its argument is not valid.

Here are some hints for you.

- Think of E^i as the shortest distance matrix when we are allowed to take i roads. Then, what should the meaning of addition and multiplication be in order for $E^{i+1}[city\,X,\, city\,Y]$ to denote the shortest distance from $city\,X$ to $city\,Y$ with $i+1$ roads in between ?

$$E^{i+1}[city\,X,\, city\,Y] = \sum_{for\,each\,city\,Z} E^i[city\,X,\, city\,Z] \times E[city\,Z,\, city\,Y]$$

- You can test your solution by comparing `solution_dl'` and `distance dl`.

2.2.3 Maximum weight problems (10 points)

First fill in the structure `Weight`. Then write a function `weight` which computes the maximum weight matrix from a given weight matrix:

```
val weight : int list list -> int list list
```

An entry of the maximum weight matrix tells the maximum weight that a car can carry whatever route it takes from one city to another. The format of the argument and the return value is the same as in `distance`.

- You can test your solution by comparing `solution_ml'` and `weight ml`.

3 Submission instruction

Download the zip file `hw3.zip` from the course webpage or from `/home/class/cs321/` on `programming2.postech.ac.kr`, and unzip it:

```
gla@ubuntu:~/temp$ unzip hw3.zip
Archive:  hw3.zip
   creating: hw3/
  inflating: hw3/hw3.mli
  inflating: hw3/.depend
  inflating: hw3/hw3.ml
  inflating: hw3/Makefile
  inflating: hw3/common.ml
```

You will write code in `hw3.ml` and never touch other files. The stub file `hw3.ml` looks like:

```
open Common

exception NotImplemented

exception IllegalFormat

module Integer : SCALAR with type t = int
=
struct
  type t = int

...
```

1. Fill the function body with your own code *only if you have a correct implementation.* This is absolutely crucial; if you leave code that does not compile, you will receive *no credit* for this assignment. If you cannot implement a function, just leave it intact! Be sure to doublecheck that your code compiles by running `make`:

   ```
   gla@ubuntu:~/temp/hw3$ ls
   common.ml  hw3.ml  hw3.mli  Makefile
   gla@ubuntu:~/temp/hw3$ make
   ocamlc  -c common.ml -o common.cmo
   ocamlc  -c hw3.mli -o hw3.cmi
   ocamlc  -c hw3.ml -o hw3.cmo
   ocamlc  -o hw3 common.cmo hw3.cmo
   ```

2. When you have the file `hw3.ml` ready for submission, copy it to your hand-in directory on `programming2.postech.ac.kr`. For example, if your Hemos ID is `foo`, copy it to:

   ```
   /home/class/cs321/handin/foo/
   ```