Homework 8
SNU 4190.210 Fall 2014

Chung-Kil Hur

due: 12/10 (Wed) 24:00

Exercise 1 "Queue Module"

A queue is a list-like data structure where the first element added to the queue will be the first one to be removed (first-in-first-out). A queue does not need to be implemented using a single list. Using two lists, one can implement a queue more efficiently as follows.

Suppose the queue Q contains $[a_1, \cdots, a_m, b_1, \cdots, b_n]$, where a_1 is the tail and b_n is the head. This queue can be represented using two lists L and R:

$$L = [a_1, \cdots, a_m], \quad R = [b_n, \cdots, b_1].$$

Here, if one adds a new element x to the queue Q, then you get the following:

$$[x, a_1, \cdots, a_m], [b_n, \cdots, b_1].$$

Instead, if one removes an element from the queue Q, then you get the following:

$$[a_1, \cdots, a_m], [b_{n-1}, \cdots, b_1].$$

When an element is removed, you may need to reverse L and switch it with R. The empty queue is represented as $([], [])$.

Consider the signature (or interface) `Queue`.

```
module type Queue =
  sig
    type element
    type queue
    exception EMPTY_Q
```

```
    val emptyq: queue
    val enq: queue * element -> queue
    val deq: queue -> element * queue
  end
```

Using the idea given above, implement the following two queues:

- Module `StringQ`: Elements of the queue are strings.
- Module `StringQQ`: Elements of the queue are `StringQ.queue`.

Note that the queues can contain duplicated elements.

Give the signature `Queue` to the modules as follows:

```
module StringQ: Queue = struct ... end
module StringQQ: Queue = struct ... end
```

Since the signature `Queue` does not expose the definition of `element`, the following code does not type check.

```
let csQ = StringQ.enq
          (StringQ.enq
           (StringQ.emptyq, "Bob"),
           "Alice")
```

We can solve the problem by defining the module `StringQ` as follows:

```
module StringQ: Queue with type element = string
 = struct ... end
module StringQQ: Queue with type element = StringQ.queue
 = struct ... end
```

□

Exercise 2 "Set Queue"

Define modules `StringSetQ` and `StringSetQQ` by modifying the two modules `StringQ` and `StringQQ` in such a way that they do not allow duplicated elements. More precisely, when an element is added to the queue, first check whether the element is already contained in the queue and if so, do not add it.

```
module StringSetQ: Queue with type element = string
 = struct ... end
module StringSetQQ: Queue with type element = StringSetQ.queue
 = struct ... end
```

□

Exercise 3 "Queue Functor"

Define a functor `QueueMake` that takes a type for elements and generates a queue module with elements of the given type (and with duplication of elements allowed).

First, define an appropriate signature `ArgTy` for taking a type for elements and define `QueueMake` as follows:

```
module QueueMake (Arg: ArgTy): Queue with type element = ...
  = struct ... end
```

Then, make the two modules:

```
module StringQ = QueueMake(...)
module StringQQ = QueueMake(...)
```

The following code should work for `StringQ`:

```
let csQ = StringQ.enq
          (StringQ.enq
           (StringQ.emptyq, "Bob"),
           "Alice")
```

□

Exercise 4 "Tile Design"

Remember the tile design module you wrote in Scheme for Exercise 1 of Homework 5. Re-implement it in OCaml using OCaml's module system as follows. More specifically, fill "`...`" with appropriate code in the following module definitions.

```
type design = TURTLE | WAVE | DRAGON  (* three design patterns *)
type orientation = NW | NE | SE | SW
type box = BOX of orientation * design | GLUED of box * box * box * box
module type FRAME =
  sig
   val box: box
   val rotate: box -> box (* rotate box M to 3 to W to E *)
   val pp: box -> int * int -> unit (* pretty printer *)
   val size: int
  end
module BasicFrame (Design: sig val design: design end): FRAME =
  struct
```

```
    exception NON_BASIC_BOX
    let box = BOX (NW, Design.design)    (* a box is defined *)
    let rotate = ...
    let pp b center = match b with
        BOX(NW,x) -> () (* dummy, fill it if you want *)
      | BOX(NE,x) -> () (* dummy, fill it if you want *)
      | BOX(SE,x) -> () (* dummy, fill it if you want *)
      | BOX(SW,x) -> () (* dummy, fill it if you want *)
      | _ -> raise NON_BASIC_BOX
    let size = 1
  end
module Rotate (Box: FRAME): FRAME =
  struct
   ...
  end
module Glue (Nw: FRAME) (Ne: FRAME) (Se: FRAME) (Sw: FRAME): FRAME =
  struct
   exception DIFFERENT_SIZED_BOXES
   ...
  end
```

Now test the following code using the above modules:

```
module A = BasicFrame(struct let design = TURTLE end)
module B = BasicFrame(struct let design = WAVE end)
module A' = Rotate(A)
module A'' = Rotate(A')
module B' = Rotate(B)
module B'' = Rotate(B')
module A4 = Glue (A) (B) (A') (B')
module B4 = Glue (A) (A') (B) (B')
module A4' = Rotate(A4)
module B4' = Rotate(B4)
module C = Glue (A4) (B4) (A4') (B4')
let bluePrint = C.pp C.box
```

□