FIRST IN FIRST OUT (50/50 points)

A queue is a standard FIFO data structure. See wikipedia

In this exercise, we implement a queue with a pair of two lists `(front, back)` such that `front @ List.rev back` represents the sequence of elements in the queue.

1. Write a function `is_empty : queue -> bool` such that `is_empty q` is true if and only if `q` has no element.
2. Write a function `enqueue : int -> queue -> queue` such that `enqueue x q` is the queue as `q` except that `x` is at the end of the queue.
3. Write a function `split : int list -> int list * int list` such that `split l = (front, back)` where `l = back @ List.rev front` and the length of `back` and `front` is `List.length l / 2` or `List.length l / 2 + 1`
4. Write a function `dequeue : queue -> int * queue` such that `dequeue q = (x, q')` where `x` is the front element of the queue `q` and `q'` corresponds to remaining elements. This function assumes that `q` is non empty.

THE GIVEN PRELUDE

```
type queue = int list * int list
```

YOUR OCAML ENVIRONMENT

```
let is_empty ((front, back) :queue) =
  if front @ List.rev back = [] then true else false
;;

let enqueue x ((front, back) : queue) :queue =
  (front, List.rev (List.rev back @ [x]))
;;

let split l =
  let rec split_rec l back = match l with
    | [] -> (l, back)
    | x :: xs ->
        if List.length xs = List.length back || List.length xs + 1 = List.length back
        then (List.rev l, back) else
          split_rec xs (back @ [x])
  in
  split_rec l []
;;

let dequeue ((front, back) : queue) = match (front, List.rev back) with
  | ([], (x :: xs)) -> x, (([], List.rev xs) : queue)
  | ((y :: ys), l) -> y, ((ys, List.rev l) : queue)
;;

```

Evaluate >

Switch >>

Typecheck

Reset Templ

Full-screen

Check & Sa

Exercise complete (click for details) 50 pts

v Exercise 1: `is_empty` Completed, 10 pts

Found `is_empty` with compatible type.
Computing `is_empty ([], [])`
Correct value `true` 1 pt
Computing `is_empty ([], [1])`
Correct value `false` 1 pt
Computing `is_empty ([2], [])`
Correct value `false` 1 pt
Computing `is_empty ([-1; 1; 3; -1; 0], [-5; -1; 3; -3; -1; -5; 0; -4])`
Correct value `false` 1 pt
Computing `is_empty ([1; 0; -3; 3; -2; -2; 3; -4; -5], [])`
Correct value `false` 1 pt
Computing `is_empty ([2; -1; 1; -1; 3; 3; 4], [])`

Computing `is_empty ([4; 4; -1; 2; 3; -4; -1; 2; -1], [-2; 2; 1; 2; -2])`
Correct value `false` 1 pt
Computing `is_empty ([-5; 1; 4; 0; -3; -5; -2; -1; -2], [-4; 1; -3; 4; 0; 2; 0; 1; -3])`
Correct value `false` 1 pt
Computing `is_empty ([1; -3; -4; 1; 1], [])`
Correct value `false` 1 pt

v Exercise 2: `enqueue` Completed, 10 pts

Found `enqueue` with compatible type.
Computing `enqueue 4 ([0; 1; 2; -1; 4; 2; 3; 4], [0; -3; 3; 2; -4; 0; 3; -2; 1])`
Correct value `([0; 1; 2; -1; 4; 2; 3; 4], [4; 0; -3; 3; 2; -4; 0; 3; -2; 1])` 1 pt
Computing `enqueue -3 ([0; -1; 2; 0; -2; -4; -3; -2; 1], [-5; 0; -1; 4; 0; 0; 2; 0])`
Correct value `([0; -1; 2; 0; -2; -4; -3; -2; 1], [-3; -5; 0; -1; 4; 0; 0; 2; 0])` 1 pt
Computing `enqueue 3 ([1; 3; -2; -1; 0; 4; 0; -1], [-2])`
Correct value `([1; 3; -2; -1; 0; 4; 0; -1], [3; -2])` 1 pt
Computing `enqueue -3 ([-3; 4; 1], [3; -4; 1; -2; 2; 2; -3; 0])`
Correct value `([-3; 4; 1], [-3; 3; -4; 1; -2; 2; 2; -3; 0])` 1 pt
Computing `enqueue -2 ([], [-1; -5; -2; 4; -3; -1])`
Correct value `([], [-2; -1; -5; -2; 4; -3; -1])` 1 pt
Computing `enqueue -3 ([-3], [-1; -4; -5; 0; 4; 0; -1])`
Correct value `([-3], [-3; -1; -4; -5; 0; 4; 0; -1])` 1 pt
Computing `enqueue 4 ([-4; 0], [])`
Correct value `([-4; 0], [4])` 1 pt
Computing `enqueue 4 ([-2; -5; 2; -3; -1], [2; 1; -1; -1; 1; 3; 3; -3; 0; -5])`
Correct value `([-2; -5; 2; -3; -1], [4; 2; 1; -1; -1; 1; 3; 3; -3; 0; -5])` 1 pt
Computing `enqueue -3 ([2; 4; -4], [0; 0; -4; 1])`
Correct value `([2; 4; -4], [-3; 0; 0; -4; 1])` 1 pt
Computing `enqueue 1 ([-2; 2; 0; -5; 2; -4; -1], [-1; 0; 1; -5; -1; -4; -2; -4])`
Correct value `([-2; 2; 0; -5; 2; -4; -1], [1; -1; 0; 1; -5; -1; -4; -2; -4])` 1 pt

v Exercise 3: `split` Completed, 20 pts

Found `split` with compatible type.
Computing `split []`
Correct value `([], [])` 1 pt
Front length is 0. Back length is 0. 1 pt
Computing `split [2; -1]`
Correct value `([-1], [2])` 1 pt
Front length is 1. Back length is 1. 1 pt
Computing `split [0; 2; 2; 0; 3; 1; -4; 0; -3; 0]`
Correct value `([0; -3; 0; -4; 1], [0; 2; 2; 0; 3])` 1 pt
Front length is 5. Back length is 5. 1 pt
Computing `split []`
Correct value `([], [])` 1 pt
Front length is 0. Back length is 0. 1 pt
Computing `split [-5]`
Correct value `([-5], [])` 1 pt
Front length is 1. Back length is 0. 1 pt
Computing `split [0; -5]`
Correct value `([-5], [0])` 1 pt
Front length is 1. Back length is 1. 1 pt
Computing `split [-1; -1; 3; 0; -4]`
Correct value `([-4; 0; 3], [-1; -1])` 1 pt
Front length is 3. Back length is 2. 1 pt
Computing `split [-3; 0; -2; -4]`
Correct value `([-4; -2], [-3; 0])` 1 pt
Front length is 2. Back length is 2. 1 pt
Computing `split [-5; 4; 0; 2; 1; 4]`
Correct value `([4; 1; 2], [-5; 4; 0])` 1 pt
Front length is 3. Back length is 3. 1 pt
Computing `split [-4; 1]`
Correct value `([1], [-4])` 1 pt
Front length is 1. Back length is 1. 1 pt

v Exercise 4: `dequeue` Completed, 10 pts

```
Computing dequeue ([], [-4; -1; -5; -2; 1; -5; 3; 1])
Correct value (1, ([], [-4; -1; -5; -2; 1; -5; 3]))                                    1 pt
Computing dequeue ([4; 0; -2; 2], [-5; 0; -5; 3])
Correct value (4, ([0; -2; 2], [-5; 0; -5; 3]))                                        1 pt
Computing dequeue ([4; 0; -5; -2; 1; -1; 0], [2; 3; 0; -3; -3; -2; -3; -3; 1; -4])
Correct value (4, ([0; -5; -2; 1; -1; 0], [2; 3; 0; -3; -3; -2; -3; -3; 1; -4]))      1 pt
Computing dequeue ([], [0; 4; -1; 1; 3])
Correct value (3, ([], [0; 4; -1; 1]))                                                 1 pt
Computing dequeue ([1; -4; 1; -2; 1], [-4; 2; -3; 1; 4; -5; -5; 0; 3])
Correct value (1, ([-4; 1; -2; 1], [-4; 2; -3; 1; 4; -5; -5; 0; 3]))                   1 pt
Computing dequeue ([1; 4; -2; 2; -1], [-2; 0; 4; -1; -4])
Correct value (1, ([4; -2; 2; -1], [-2; 0; 4; -1; -4]))                                1 pt
Computing dequeue ([4; -5; 1; -3; -2], [-4; -5; -2; -5; 1; -5; 3; -3; -1; -5])
Correct value (4, ([-5; 1; -3; -2], [-4; -5; -2; -5; 1; -5; 3; -3; -1; -5]))           1 pt
Computing dequeue ([2; 2; -1; 3; -4; -2; -1], [-5; -5; -3])
Correct value (2, ([2; -1; 3; -4; -2; -1], [-5; -5; -3]))                              1 pt
Computing dequeue ([-2; -2; -4; -1; -4; 0; 0; 4; -1], [-4; 3])
Correct value (-2, ([-2; -4; -1; -4; 0; 0; 4; -1], [-4; 3]))                           1 pt
```