CS510: Special Assignment 1

October 25, 2021

Due: 28 October, 5:00pm

---

1 Assignment Policies

Collaboration Policy. This assignment must be completed **individually**. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment consists of extending `REC` to allow for the queue data structure and its respective operations. The resulting language is dubbed `RECQ`.

A queue is a linear data structure, often implemented as a linked list, that follows the FIFO (First In First Out) principle.

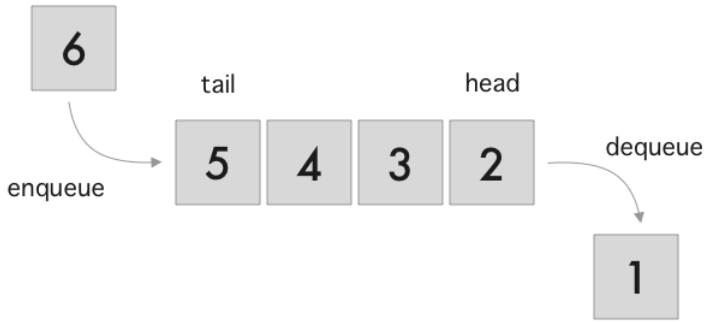

Figure 1: Depicition of a queue data structure

With the FIFO principle, the first inserted object can be accessed in $O(1)$ time. The queue is a simple, versatile data structure with plenty of real world applications. For this assignment, the *addq, remove, emptyqueue, element, size*, and, *empty?* operations will be implemented.

1. Emptyqueue: Creates an empty queue.

```
utop # interp "emptyqueue";;
- : exp_val Recq.Ds.result = Ok (QueueVal [])
```

2. AddQ: Adds an element to the back of the queue.

```
utop # interp "addq(4, addq(3, addq(2, addq(1, emptyqueue))))";;
- : exp_val Recq.Ds.result =
Ok (QueueVal [NumVal 4; NumVal 3; NumVal 2; NumVal 1])
```

   Please note that the order of the QueueVal from right to left represents the queue from front to back. In this example, the first value is 1, followed by 2, followed by 3 and finally ending with 4.

3. Removes: Removes the first element from the queue.

```
utop # interp "remove(addq(4, addq(3, addq(2, addq(1, emptyqueue)))))"
    ↪ ;;
- : exp_val Recq.Ds.result = Ok (QueueVal [NumVal 4; NumVal 3; NumVal
    ↪ 2])
```

   Building upon the example introduced in the addq operation, the remove operation removes the first value of 1, leaving 2 as the next front value followed by 3 and 4.

   If the queue is empty, return an error indicating that the operation failed due to an empty queue. To expedite the grading process, please return "Remove: Queue is empty."

4. Element: Returns the front most element on the queue *without* modifying the queue.

```
utop # interp "element(addq(4, addq(3, addq(2, addq(1, emptyqueue)))))
    ↪ ";;
- : exp_val Recq.Ds.result = Ok (NumVal 1)
```

   If the queue is empty, return an error indicating that the operation failed due to an empty queue. To expedite the grading process, please return "Remove: Queue is empty."

5. Size: Returns the amount of elements in the queue.

```
utop # interp "size(addq(4, addq(3, addq(2, addq(1, emptyqueue)))))";;
- : exp_val Recq.Ds.result = Ok (NumVal 4)
```

6. Empty: Returns a boolean indicating whether the queue is populated or not.

```
utop # interp "empty?(addq(4, addq(3, addq(2, addq(1, emptyqueue)))))"
    ↪ ;;
- : exp_val Recq.Ds.result = Ok (BoolVal false)
```

3 Implementing queues in REC

To facilitate the process of implementing stacks in `REC`, a stub has been provided for you in Canvas. This sub has been obtained by taking the interpreter for `REC` and applying some changes. Here is a summary of the changes:

1. The parser.mly file has been updated so that the parser is capable of parsing expressions such as:

```
utop # parse "element(remove(addq(43, addq(21, addq(28, addq(19, addq
    ↪ (32, remove(addq(1, addq(4, addq(5, emptyqueue)))))))))))";;;
```

   The result of parsing this expression would be:

```
- : expr =
Element
 (Remove
   (AddQ (Int 43,
     AddQ (Int 21,
      AddQ (Int 28,
       AddQ (Int 19,
        AddQ (Int 32,
         Remove (AddQ (Int 1, AddQ (Int 4, AddQ (Int 5, Var "
            ↪ emptyqueue")))))))))))
```

2. Note the new additions to ast.ml:

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
  | Set of string*expr
  | BeginEnd of expr list
  | NewRef of expr
  | DeRef of expr
  | SetRef of expr*expr
  | Pair of expr*expr
  | Fst of expr
  | Snd of expr
  | EmptyQueue
  | AddQ of expr*expr
  | Element of expr
```

```
  | Remove of expr
  | IsEmpty of expr
  | Size of expr
  | Tuple of expr list
  | Debug of expr
```

3. Note the new addition to ds.ml

```
let list_of_queueVal : exp_val -> (exp_val list) ea_result = function
  | QueueVal l -> return l
  | _ -> error "Expected a queue!"
```

4 Trying Out Your Code

We have provided a few test cases in the stub on Canvas. In the parent directory, run:

```
dune runtest
```

This will run the test cases that we have provided for you. Please note that these test cases are by no means exhaustive. We encourage you to thoroughly test your submission on edge cases. You can do so by typing out various commands in the interpreter as demonstrated in section 2.

5 Submission Instructions

Submit a file named `SA1_<SURNAME>.zip` through Canvas. Include all files from the stub.