

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

2 Foundations of Computer Science (LCP)

lists, queues,  
complexity

- (a) Write brief notes on the queue data structure and how it can be implemented efficiently in ML. In a precise sense, what is the cost of the main queue operations? (It is not required to present ML code.) [6 marks]

---

*Answer:* A queue represents a sequence, allowing elements to be taken from the head and added to the tail. Lists can implement queues, but `append` is a poor means of adding elements to the tail. The solution is to represent a queue by a pair of lists, where

$$([x_1, x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

represents the queue  $x_1x_2 \dots x_my_n \dots y_1$ .

The front part of the queue is stored in order, and the rear part is stored in reverse order. We add elements to the rear part using `cons`, since this list is kept reversed; this takes constant time. To remove an element, we look at the front part, which normally takes constant time, since this list is stored in order. When the last element of the front part is removed, we reverse the rear part, which becomes the new front part.

Queue operations take  $O(1)$  time when *amortized*: averaged over the lifetime of a queue. Even for the worst possible execution, the average cost per operation is constant.

---

lists, exceptions,  
programming

- (b) Run-length encoding is a way of compressing a list in which certain elements are repeated many times in a row. For example, a list of the form  $[a, a, a, b, a, a]$  is encoded as  $[(3, a), (1, b), (2, a)]$ . Write a polymorphic function `rl_encode` to perform this encoding. What is the type of `rl_encode`? [6 marks]

---

*Answer:*

```
fun rl_encode [] = []
  | rl_encode (x::xs) =
    let fun code (n, []) = [(n, x)]
        in code (n, y::ys) =
            if x=y then code (n+1, ys)
            else (n, x) :: rl_encode (y::ys)
        in code (1, xs) end;
```

The type is `'a list -> (int * 'a) list`, due to the equality test.

---

lists,  
programming

- (c) The simple task of testing whether two lists are equal can be generalised to allow a certain number of errors. We consider three forms of error:

- *element mismatch*, as in  $[1, 2, 3]$  versus  $[1, 9, 3]$  or  $[1, 2, 3]$  versus  $[0, 2, 3]$
- *left deletion*, as in  $[1, 3]$  versus  $[1, 2, 3]$  or  $[1, 2]$  versus  $[1, 2, 3]$
- *right deletion*, as in  $[1, 2, 3]$  versus  $[1, 3]$  or  $[1, 2, 3]$  versus  $[1, 2]$

Write a function `genEquals n xs ys` that returns `true` if the two lists `xs` and `ys` are equal with no more than `n` errors, and otherwise `false`. You may assume that `n` is a non-negative integer. [8 marks]

---

*Answer:*

```
fun genEquals n [] [] = true
  | genEquals n [] (y::ys) = n>0 andalso genEquals (n-1) [] ys
  | genEquals n (x::xs) [] = n>0 andalso genEquals (n-1) xs []
  | genEquals n (x::xs) (y::ys) =
    if x=y then genEquals n xs ys
    else n>0 andalso
      (genEquals (n-1) xs ys
orelse genEquals (n-1) (x::xs) ys
orelse genEquals (n-1) xs (y::ys));
```

---

All ML code must be explained clearly and should be free of needless complexity.