— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2015 – Paper 1

1 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

O-notation, lists, binary trees, functional arrays

(*a*) Write brief notes about a tree representation of functional arrays, subscripted by positive integers according to their representation in binary notation. How efficient are the lookup and update operations? [6 marks]

Answer: The underlying data structure is the binary tree. A location in the tree is found by starting at the root, testing whether the subscript is even or odd, and descending into the left or right subtree, respectively; this process terminates when 1 is reached. Here is the code for lookup:

```
exception Subscript
let rec sub k = function
| Lf -> raise Subscript
| Br (v, t1, t2) ->
    if k = 1 then v
    else if k mod 2 = 0 then
      sub (k / 2) t1
    else
      sub (k / 2) t2
```

Lookup and update both take $O(\log n)$ time, where n is the size of the array, because the representation guarantees balancing. The update operation naturally copies only the path from the root to the updated node, rather than the entire tree.

(*b*) Write an OCaml function `arrayoflist` to convert the list $[x_1; \ldots; x_n]$ to the corresponding functional array having x_i at subscript position i for $i = 1, \ldots, n$. Your function should not call the update operation. [6 marks]

Answer: The point is to realise the tree structure directly, rather than repeatedly updating. Here is a straightforward solution:

```
let rec revalts ys zs = function
| [] -> (List.rev ys, List.rev zs)
| [x] -> (List.rev (x::ys), List.rev zs)
| x1::x2::xs -> revalts (x1::ys) (x2::zs) xs

let alts = revalts [] []

let rec arrayoflist = function
| [] -> Lf
| x::xs ->
    let (evens, odds) = alts xs in
     Br (x, arrayoflist evens, arrayoflist odds)
```

There is an elegant solution based on the following "cons" operation for Braun trees:

```
let rec tcons v = function
| Lf -> Br (v, Lf, Lf)
| Br (w, t1, t2) -> Br (v, tcons w t2, t1)
```

(*c*) Consider the task of finding out which elements of an array satisfy the predicate `p`, returning the corresponding subscript positions as a list. For

example, the list $[2; 3; 6]$ indicates that these three designated array elements, and no others, satisfy `p`. Write an OCaml functional to do this for a given array and predicate, returning the subscripts in increasing order. [8 marks]

Answer: The algorithm is a straightforward recursion. Using `merge` delivers a sorted result. A solution that returns an unsorted result, combined with a sorting function, is likely to lose marks due to inelegance and inefficiency.

```
let rec merge xs (ys : int list) =
  match xs, ys with
  | [], ys -> ys
  | xs, [] -> xs
  | x::xs, y::ys ->
      if x<=y then
        x::(merge xs (y::ys))
      else
        y::(merge (x::xs) ys)

let rec mfilter p = function
| Lf -> []
| Br (x, t1, t2) ->
    let ks = merge (List.map (fun k -> 2 * k) (mfilter p t1))
                   (List.map (fun k -> 2 * k + 1) (mfilter p t2))
    in
      if p x then
        1 :: ks
      else
        ks
```

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