— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2004 – Paper 1

1 Foundations of Computer Science (ACN)

This question has been translated from Standard ML to OCaml

(*a*) What does the OCaml function `map` do? Give an example, first coded without `map` and then with it, to illustrate how it can lead to more compact or comprehensible code. [3 marks]

(*b*) Functions `fold_left` and `fold_right` might be defined as

```
let rec fold_left f e = function
  | [] -> e
  | x::xs -> fold_left f (f ex) xs

let rec fold_right f l e = match l with
  | [] -> e
  | x::xs -> f x (fold_right f xs e)
```

Explain what these two functions do and why they may be useful. [4 marks]

(*c*) Here is a typical use of `map`:

```
let mangle n = (n - 2) * (n + 7)
let mangle_list x = map mangle x
```

Show how to express `mangle_list` using one of the "fold" functions rather than `map`. [3 marks]

— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2004 – Paper 1

5 Foundations of Computer Science (ACN)

This question has been translated from Standard ML to OCaml

The following OCaml type can be viewed as defining a lazy or infinite sort of tree where each node in the tree holds an integer:

```
type tr = N of int * unit -> tr * unit -> tr
```

(*a*) Write a function called `ndeep` such that if `n` is an integer and `z` is a tree (i.e. of type `tr`) the call `ndeep n z` will return an ordinary list of all the 2^n integers at depth exactly `n` in the tree. Note that if `n` = 0 it will return a list of length 1, being just the top integer in the tree. Comment on its efficiency. [8 marks]

Answer:

```
let rec ndeep n (N(v, l, r)) =
  if n = 0 then
    [v]
  else
    ndeep (n - 1) (l ()) @ ndeep (n - 1) (r ())
```

The append makes this slower than would be perfect, and experts can do the usual conversion to avoid that.

(*b*) You are given a `tr`, and told that it contains arbitrarily large values at least somewhere in it. You want to find a value from it that is bigger than 100 (but if there are many big values it does not matter which one is returned). Because the tree is infinite you cannot use simple depth-first search: you decide to use iterative deepening. Thus you first check all integers at depth 1, then at depth 2, depth 3, ... and return when you first find a value that is greater than 100.

Use exception handling to return the large value when you find it. Present and explain code that searches the lazy tree. [12 marks]

Answer:

```
exception Found of int

let rec throwifin = function
  | [] -> ()
  | x::xs ->
     if x > 100 then
       raise (Found x)
     else
       throwifin xs

let search z =
  let rec depth n =
    throwifin (ndeep n z);
```

```
    depth (n + 1)
in
  try
    depth 1;
    0 (* Unreachable *)
  with Found x -> x
```

— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2004 – Paper 1

6 Foundations of Computer Science (ACN)

This question has been translated from Standard ML to OCaml

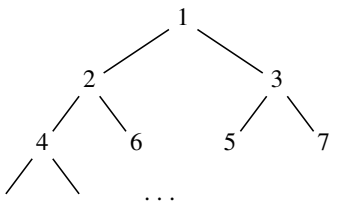
In OCaml it is possible to use functions as values: they can be passed as arguments and returned as results. Explain the notation used to write a function without having to give it a name. [2 marks]

Answer:

```
fun x -> A
```

This question looks at two different ways of implementing functional arrays.

(*a*) One possible functional implementation of an array is based on trees, and the path to a stored value follows the binary code for the subscript:



where in the above diagram the numbers show where in the tree a value with the given subscript will live.

Write code that creates, retrieves values from and updates an array that has this representation, and using big-O notation explain the associated costs. [8 marks]

Answer: I will give here the code to read from the tree-style array, but update is then trivial and follows from it.

```
let rec access n (B(v, l, r)) =
  if n = 1 then
    v
  else if n mod 2 = 1 then
    access ((n - 1) / 2) r
  else
    access (n / 2) l
```

Cost is guaranteed O(log n) where n is subscript being used.

(*b*) A different way of handling functional arrays is to represent the whole array directly by a function that maps from integers to values. To access the item at position k in such an array you just use the array as a function and give it

k as its argument, and to update the array you need to create a new function reflecting the changed value.

If the array is to hold integer values, what OCaml type does it have? [1 mark]

Answer:

```
int -> int
```

Write a function `update a n v` where `a` is a functional array in this style, `n` is an integer index and `v` is a new value. The result of the call to `update` must behave as an array that stores all the values that `a` did except that it is as if an assignment of the style "`a[n] := v`" has been performed. [5 marks]

Answer:

```
let update a n v =
  fun i -> if i = n then v else a n (* easy *)
```

In big-O notation, what is the cost of your `update` function? After a sequence of updates what is the cost of accessing the array? [4 marks]

Answer: Cost of update is O(1), but cost of access is linear in number of updates done.

— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2012 – Paper 1

1 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

lecture 8, general programming skills

Recall that a dictionary of (*key*, *value*) pairs can be represented by a binary search tree. Define the *union* of two binary search trees to be any binary search tree consisting of every node of the given trees.

(*a*) Write an OCaml function `union` to return the union of two given binary search trees. [*Note:* You may assume that they have no keys in common.] [6 marks]

Answer: This particular solution uses the general update function, but there could be many alternative solutions. Taking this function for granted (in other words, omitting it from the solution) isn't acceptable, as that would make the problem almost trivial.

```
let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
    if k < a then
      Br ((a, x), update k v t1, t2)
    else if a < k then
      Br ((a, x), t1, update k v t2)
    else (* a = k *)
      Br ((a, v), t1, t2)

let rec union l r =
  match l, r with
  | (Lf, r) -> r
  | (Br ((k, v), t1, t2), r) ->
      union t1 (union t2 (update k v r))
```

Define a *slice* of a binary search tree to be a binary search tree containing every (*key*, *value*) node from the original tree such that $x \leq key \leq y$, where x and y are the given endpoints.

(*b*) Write an OCaml function `takeSlice` to return a slice – specified by a given pair of endpoints – from a binary search tree. [4 marks]

Answer: The solution is a straightforward recursion.

```
let rec takeSlice x y = function
| Lf -> Lf
| Br ((k, v), t1, t2) ->
    if y < k then
      takeSlice x y t1
    else if k < x then
      takeSlice x y t2
    else
      Br ((k, v), takeSlice x y t1, takeSlice x y t2)
```

This can also be done with `when` (pattern guards):

```
let rec takeSlice (x, y) = function
```

```
| Lf -> Lf
| Br ((k, v), t1, t2) when y < k ->
    takeSlice (x, y) t1
| Br ((k, v), t1, t2) when k < x ->
    takeSlice (x, y) t2
| Br ((k, v), t1, t2) ->
    Br ((k, v), takeSlice (x, y) t1, takeSlice (x, y) t2)
```

(*c*) Write an OCaml function `dropSlice` to *remove* a slice from a binary search tree: given a tree and a pair of endpoints, it should return the binary search tree consisting of precisely the nodes such that $x > key$ or $key > y$. [*Hint:* First consider the simpler task of deleting a node from a binary search tree.]

[8 marks]

Answer: Deletion is not straightforward. The problem is to combine the remaining subtrees while preserving the ordering. A simple approach is to attach the right-hand tree at the far-right end of the left-hand tree, but inevitably, the resulting tree will be unbalanced.

Given deletion, the solution is once again a straightforward recursion.

```
let rec join l r =
  match l r with
  | (Lf, r) -> r
  | (Br (x, t1, t2), r) ->
      Br (x, t1, join t2 r)

let rec dropSlice (x, y) = function
| Lf -> Lf
| Br ((k, v), t1, t2) ->
    if y < k then
      Br ((k, v), dropSlice (x, y) t1, t2)
    else if k < x then
      Br ((k, v), t1, dropSlice (x, y) t2)
    else
      join (dropSlice (x, y) t1) (dropSlice (x, y) t2)
```

(*d*) The tree t need not be identical to that returned by

```
union (takeSlice (x, y) t)
      (dropSlice (x, y) t)
```

Briefly explain how such an outcome is possible. [2 marks]

Answer: They will represent equivalent dictionaries, in that they map the same values to the same keys. However, many distinct binary search trees can represent any particular dictionary. It's highly unlikely that the operation described in the question would preserve the exact structure of a binary search tree.

[*Note:* All OCaml code must be explained clearly and should be free of needless complexity.]

— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2012 – Paper 1

2 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

second half of the course, in particular lectures 10–14

(*a*) Write brief notes on `fun`-notation and curried functions in OCaml. Illustrate your answer by presenting the code for a polymorphic curried function `replicate`, which given a non-negative integer n and a value x, returns the list $[\underbrace{x; \ldots; x}_{n}]$. [6 marks]

Answer: The syntax of `fun`-notation is `fn` x `->` E, denoting a function with argument x that returns when called the value of E. The point is the ability to express such a function without having to name it first. This notation, obviously, always yields expressions that have a function type derived from the types of x and E.

Given `fun`-notation, we can express curried functions, that is, functions that return another function as their result, and this function will typically make use of supplied argument. One advantage of currying is that it allows partial application: regarding such a returned function as useful in its own right.

OCaml provides a special syntax for writing curried functions, which is especially useful in the case of recursion. The function `replicate` can be declared as follows:

```
let rec replicate n x =
  if n = 0 then
    []
  else
    x :: replicate (n - 1) x
```

Its polymorphic type is `int -> 'a -> 'a list`.

(*b*) Write brief notes on references in OCaml. Illustrate your answer by discussing (with the aid of a diagram) the effect of the following two top-level declarations:

```
let rlist = replicate 4 (ref 0) @ List.map ref [1; 2; 3; 4]
let slist = List.map (fun r -> ref !r) rlist
```

[6 marks]

Answer: The key concepts of references include the function `ref`, which creates a reference cell, `!`, which inspects a reference cell, and `:=`, which updates a reference cell with a given value. These cells are mutable, but the references to them are pure values, like everything else in OCaml.

The first declaration yields a list whose first four elements refer to a single shared reference cell containing zero. (The diagram should show this sharing.) The remaining four list elements refer to reference cells containing the integers 1 up to 4, respectively. The second declaration creates a new list referring to freshly-allocated (and therefore distinct) reference cells, the first four containing zero and the remaining four again holding the integers 1 up to 4. Students are expected to understand the use of `List.map` in this question.

(*c*) The following three lines are typed at the OCaml top-level, one after the other. What value is returned in each case? Justify your answer clearly. [*Note:* Recall that an expression of the form v `:=` E has type `unit`.]

```
List.map (fun r -> (r := !r + 1)) rlist
List.map (fun r -> (r := !r - 1; !r)) rlist
List.map (fun r -> (r := !r + 3; !r)) slist
```

[8 marks]

Answer: The values of the three lines are given as follows:

```
- : unit list = [(); (); (); (); (); (); (); ()]
- : int list = [3; 2; 1; 0; 1; 2; 3; 4]
- : int list = [3; 3; 3; 3; 4; 5; 6; 7]
```

The first one is trivial, for students who can remember that the only value of type `unit` is `()`. But students must also recognise that this first line has side effects: the first shared reference in `rlist` is increased by four (in four separate increments) while the remaining four references are increased by one.

For the second one, students need to understand the semi-colon notation, which in this case returns the contents of the reference at the given moment. The first four values of this result illustrate successive decrementing of the shared reference.

For the third one, students need to understand that all the references in `slist` are independent.

— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2013 – Paper 1

1 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

variants, pattern-matching

(*a*) Write brief notes on OCaml variants and pattern-matching in function declarations. [6 marks]

Answer: Solutions should include examples of variant type declarations and mention the concept of a constructor. Examples of pattern-matching should be non-trivial, with nested constructors and (preferably) overlapping patterns.

programming, binary trees

(*b*) A binary tree is either a *leaf* (containing no information) or is a *branch* containing a label and two subtrees (called the *left* and *right* subtrees). Write OCaml code for a function that takes a label and two lists of trees, returning all trees that consist of a branch with the given label, with the left subtree taken from the first list of trees and the right subtree taken from the second list of trees. [6 marks]

Answer: The variant type declaration is not required as part of the answer, but sets the stage. Students are unlikely to know about `List.concat`, but it can be coded in two lines with the help of @ (append).

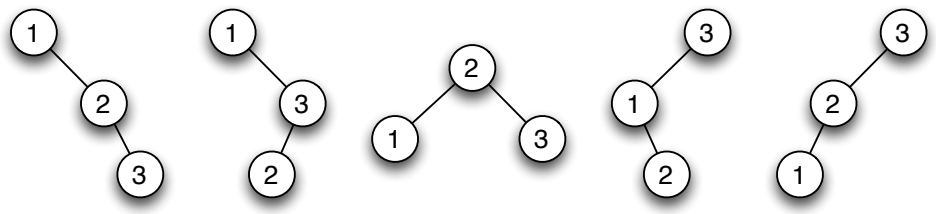
```
type 'a tree = Lf
             | Br of 'a * 'a tree * 'a tree

let make_trees v t1 =
  List.map (fun t2 -> Br (v, t1, t2))

let make_trees2 v t1s t2s =
  List.concat (List.map (fun t1 -> make_trees v t1 t2s) t1s)
```

programming, binary trees

(*c*) Write OCaml code for a function that, given a list of distinct values, returns a list of all possible binary trees whose labels, enumerated in inorder, match that list. For example, given the list `[1; 2; 3]` your function should return (in any order) the following list of trees:



[8 marks]

Answer:

```
let rec anti l1 = function
```

```
  | [] -> []
  | v::l2 ->
      make_trees2 v (anti_inorder (List.rev l1)) (anti_inorder l2) @
      anti (v::l1) l2
and anti_inorder = function
  | [] -> [Lf]
  | xs = anti [] xs
```

Note that the question refers to binary trees, not to binary *search* trees, and it does not impose an ordering constraint on the labels of these trees.

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

— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2013 – Paper 1

2 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

algorithms, lists, curried functions, higher-order functions

The function `perms` returns all $n!$ permutations of a given n-element list.

```
let rec perms = function
| [] -> [[]]
| xs ->
    let rec perms1 xs ys =
      match xs with
      | [] -> []
      | x::xs ->
          List.map (List.cons x) (perms (List.rev ys @ xs)) @
                   perms1 xs (x::ys)
    in
      perms1 xs []
```

(*a*) Explain the ideas behind this code, including the function `perms1` and the expression `List.map (List.cons x)`. What value is returned by `perms [1; 2; 3]`? [7 marks]

Answer: The base case is `[[]]` because the empty list has one permutation, namely `[]`. The idea of the code is that the permutations of a list containing some element x consist of (a) those that begin with x, the tail computed by a recursive call, and (b) those that do not begin with x. The function `perms1` walks down a list, choosing successive list elements to play the role of x above. The expression `List.map (List.cons x)` modifies the list of permutations obtained from the recursive call by inserting x as the first element of each. Here, `List.cons` is a curried function.

```
perms [1; 2; 3] =
[[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
```

---

lazy lists

(*b*) A student modifies `perms` to use an OCaml type of lazy lists, where `appendq` and `mapq` are lazy list analogues of `@` and `List.map`.

```
let rec lperms = function
| [] -> Cons ([], fun () -> Nil)
| xs ->
    let rec fun perms1 xs ys = function
    | [] -> Nil
    | x::xs ->
        appendq (mapq (List.cons x) (lperms (List.rev ys @ xs)))
                (perms1 xs (x::ys))
    in
      perms1 xs []
```

Unfortunately, `lperms` computes all $n!$ permutations as soon as it is called. Describe how lazy lists are implemented in OCaml and explain why laziness is not achieved here. [5 marks]

Answer: OCaml's lazy values do not form part of the syllabus. Lazy lists can be simulated using the following variant type declaration:

```
type 'a seq = Nil
            | Cons of 'a * (unit -> 'a seq)
```

Laziness can be obtained through writing functions of the form `fun () ->` E, for then the expression E is not evaluated until the function is called, with argument `()`.

Thw function above uses lazy list primitives correctly as regards types, but the only occurrence of `fun () ->` protects an instance of `Nil`. All recursive calls to `lperms` take place when the function is called, and therefore all permutations are computed.

lazy lists

(c) Modify the function `lperms`, without changing its type, so that it computes permutations upon demand rather than all at once. [8 marks]

Answer: The trick is to insert an occurrence of `fun () ->` within the recursive calls. One way of doing this is by modifying the function `mapq`. There are other solutions.

```
let rec mapapp f xq yf =
  match xq with
  | Nil ->
      yf ()
  | Cons (x, xf) ->
      Cons(f x, fun () -> mapapp f (xf ()) yf)

let rec lperms = function
| [] -> Cons ([], fun () -> Nil)
| xs ->
    let rec perms1 xs ys =
      match xs with
      | [] -> Nil
      | x::xs ->
          mapapp (List.cons x) (lperms (List.rev ys @ xs))
                 (fun () -> perms1 xs (x::ys))
    in
      perms1 xs []
```

An OCaml version of this Tripos would probably have prohibited the use of the `Lazy` module, but this can also be achieved with:

```
type 'a seq = Nil
            | Cons of 'a * 'a seq lazy_t

let rec mapapp f xq yf =
  match xq with
  | Nil ->
      Lazy.force yf
  | Cons (x, xf) ->
      Cons (f x, lazy (mapapp f (Lazy.force xf) yf))

let rec lperms = function
| [] -> Cons ([], lazy Nil)
| xs ->
    let rec perms1 xs ys =
      match xs with
```

```
  | [] -> Nil
  | x::xs ->
      mapapp (List.cons x) (lperms (List.rev ys @ xs))
             (lazy (perms1 xs (x::ys)))
in
  perms1 xs []
```

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

— *Solution notes* —

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

1 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

types, polymorphism

(*a*) Write brief notes on polymorphism in OCaml, using lists and standard list functions such as `@` (append) and `List.map`. [4 marks]

Answer: Key points are that polymorphism assigns a type to every expression — at compile time — while at the same time allowing natural genericity. For instance, the elements of a list must have the same type, but it can be any type. The type of append, `'a list -> 'a list -> 'a list`, indicates that it combines two lists of the same type, returning another list of that type. The type of `map`, `('a -> 'b) -> 'a list -> 'b list`, indicates that it transforms a list of one type to another, as indicated by the type (`'a -> 'b`) of the function.

---

variants, functions

(*b*) Explain the meaning of the following declaration and describe the corresponding data structure, including the role of polymorphism.

```
type 'a se = Void | Unit of 'a | Join of 'a se * 'a se
```

[4 marks]

Answer: This declares a variant type containing three constructors: `Void`, `Unit` and `Join`. The latter two constructors require arguments, while `Void` is a constant. This is a tree-like data structure with unlabelled binary branching (`Join`), labelled leaves (`Unit`) and unlabelled leaves (`Void`). Type `'a se` is polymorphic, as indicated by the type variable `'a`, which shows that `'a` is the type of the labels. Functions involving the new type can be declared using pattern matching.

---

variants, functions, recursion

(*c*) Show that OCaml lists can be represented using this variant type by writing the functions `encode_list` of type `'a list -> 'a se` and `decode_list` of type `'a se -> 'a list`, such that `decode_list (encode_list xs) = xs` for every list `xs`. [3 marks]

Answer:

```
let rec encode_list = function
| [] -> Void
| x::xs -> Join (Unit x, encode_list xs)

exception Not_a_list
let rec decode_list = function
| Void -> []
| Join (Unit x, v) ->
    x :: decode_list v
| _ -> raise Not_a_list
```

---

functions as values

(*d*) Consider the following function declaration:

```
let rec cute p = function
```

```
| Void -> false
| Unit x -> p x
| Join (u, v) ->
    cute p u || cute p v
```

What does this function do, and what is its type? [4 marks]

Answer: The function `cute` has type `('a -> bool) -> 'a se -> bool`, and `cute p s` returns true if and only if `s` contains an element of the form `Unit x`, where `p x` is true. It is analogous to the function `exists`, for lists.

---

functions as values

(*e*) Consider the following expression:

```
fun p -> cute (cute p)
```

What does it mean, and what is its type? Justify your answer carefully. [5 marks]

Answer: This is a function of type `('a -> bool) -> 'a se se -> bool`. Through the `fun` binder, it takes an argument `p`, which has type `'a -> bool`. Now `cute p` has type `'a se -> bool`, and because `cute` is polymorphic, `cute (cute p)` is well-defined and has type `'a se se -> bool`.

Now if `fun p -> cute (cute p)` is applied to some specific `p` and then to a term `s`, it returns true if and only if `s` contains an element of the form `Unit x`, where `cute p x` is true. Thus the expression is like `cute` but for type `'a se se -> bool`, that is, for the data structure nested in itself.

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

2 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

lists, queues, complexity

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

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

represents the queue $x_1 x_2 \ldots x_m y_n \ldots y_1$.

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:

```
let rec rl_encode = function
| [] -> []
| x::xs ->
    let rec code n = function
    | [] -> [(n, x)]
    | y::ys ->
        if x = y then
          code (n + 1) ys
        else
          (n, x) :: rl_encode (y::ys)
    in
      code 1 xs
```

The type is `'a list -> (int * 'a) list`. The `code` function can also be expressed with guard clauses:

```
    let rec code n = function
    | [] -> [(n, x)]
    | y::ys when x = y -> code (n + 1) ys
    | ys -> (n, x) :: rl_encode ys
```

---

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:

```
let rec genEquals n xs ys =
  match xs, ys with
  | ([], []) -> true
  | ([], y::ys) -> n > 0 && genEquals (n - 1) [] ys
  | (x::xs, []) -> n > 0 && genEquals (n - 1) xs []
  | ((x::xs), (y::ys)) ->
      if x = y then genEquals n xs ys
      else n > 0 && (genEquals (n - 1) xs ys
                     || genEquals (n - 1) (x::xs) ys
                     || genEquals (n - 1) xs (y::ys))
```

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

— *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.