UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1998

BEng Honours Degree in Computing Part I
MEng Honours Degrees in Computing Part I
BSc Honours Degree in Mathematics and Computer Science Part I
MSci Honours Degree in Mathematics and Computer Science Part I
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the
Associateship of the Royal College of Science
Associateship of the City and Guilds of London Institute

PAPER 1.2 / MC1.2

REASONING ABOUT PROGRAMS
Tuesday, May 5th 1998, 2.00 - 3.30

Answer THREE questions

For admin. only: paper contains 4 questions

1 Recall that list concatenation ++ is defined by:

```
[]++ys=ys,      (x:xs)++ys = x:(xs++ys)
```

We define the Haskell function `inlist`, telling whether x is in the list ys, by:

```
inlist :: a -> [a] -> Bool
inlist x []         = False
inlist x (y:ys)     | x==y = True
                    | otherwise = inlist x ys
```

Now we define a datatype to represent binary trees:

```
data Btree a  =  Emptytree | Node (Btree a) a (Btree a)
```

Define the function squash, to turn a Btree into a list, as follows:

```
squash :: Btree a -> [a]
squash Emptytree      = []
squash (Node t1 y t2) = squash t1 ++ y:squash t2
```

Define the function intree, the tree analogue of inlist, as follows:

```
intree :: a -> Btree a -> Bool
intree x Emptytree        = False
intree x (Node t1 y t2)   | x==y = True
                          | otherwise = intree x t1
                                        || intree x t2
```

a State the principle of list induction, and a principle of structural induction suitable for the type `Btree`.

b Prove by list induction on ys that for all x of type a and all ys, zs of type [a],

```
inlist x (ys++zs) = (inlist x ys) || (inlist x zs).
```

c Using your answer to part b, prove by structural induction that for all x of type a and all t of type `Btree`,

```
intree x t = inlist x (squash t).
```

You may assume without proof that ++ is associative.

The three parts carry, respectively, 20%, 40%, 40% of the marks.

2 This question asks you to develop the following Turing function:

```
ExclOr(A: array 0.. of boolean) : boolean
r = exclusive-or of A(0), A(1), …, A(Upper(A)).
```

The post-condition can be equivalently stated as: **r** is true if an odd number of A(0), A(1), …, A(Upper(A)) are true, and false otherwise.

a The procedure `ExclOr` could be written using a loop. Draw a diagram of `A`, showing suitable pointers, etc., representing the situation at the beginning of an arbitrary cycle in the execution of the loop.

b Write the body of `ExclOr`. Include the loop variant and invariant as comments.

c Show that the loop code re-establishes the loop invariant. Remember to check that all array accesses are legal.

d Show that the loop terminates, and that when it does, the post-condition is set up.

The four parts carry, respectively, 20%, 30%, 30%, 20% of the marks.

3a What is a *tail recursive function?* What is an *accumulating parameter,* and how is it related to tail recursion?

b A recursive procedure that calculates the integer part of the log to base 2 of a positive integer is given below.

```
function Lg(n : int) : int
% pre  n > 0
% post r = integer part of log_2(n)
   if n=1 then result 0
   else result 1 + Lg(n div 2)
   end if
end Lg
```

i) Give the definition of a tail recursive function `TrLg` that, with a suitable choice of value for the accumulating parameter, produces the same result as `Lg`. Your answer may be in either Turing or Haskell.

ii) What arguments must you give the `TrLg` function to calculate `Lg n`?

c Write the Turing code of a procedure `LpLg` that calculates `TrLg` by using a loop, without recursion. Do not forget to include a pre-condition, post-condition, loop variant, and loop invariant as comments.

d i) Show that the loop code in `LpLg` that you wrote in part c re-establishes the invariant.

ii) Use this to show that `LpLg` does produce the same result as `TrLg` when given any arguments that meet its pre-condition.

The four parts carry, respectively, 20%, 15%, 35%, 30% of the marks.

Turn over ...

4 This question asks you to write a Turing procedure `Rank` to rearrange a large array `M` of integers, containing students' degree marks. `M` is indexed by integers between `lower(M)` and `upper(M)`. The pass-fail boundary mark will be provided as a parameter `Passmark` to the procedure. The post-condition for your program is

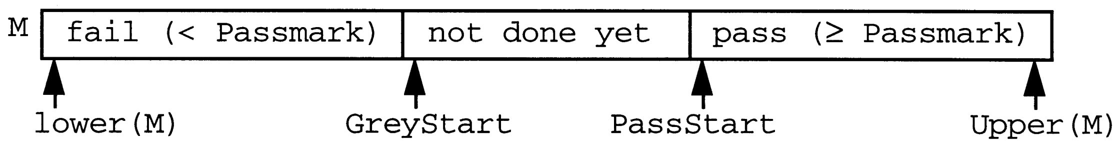
```
M is a rearrangement of M0,  and
(A)i,j : int(lower(M) ≤ i ≤ j ≤ upper(M)
            & M(i) ≥ Passmark → M(j) ≥ Passmark).
```

You are given a procedure `Swap(M : array of int , i,j:int)`, with pre-condition

```
lower(M) ≤ i,j ≤ upper(M),
```

to swap the i^{th} and j^{th} entries of M. You may only rearrange `M` by using `Swap`.

You are to implement `Rank` using a loop, rearranging `M` in one pass. The diagram shows the array `M` at the beginning of an arbitrary iteration of the loop.



a Use the diagram to derive a loop invariant and loop variant.

b Write the Turing code for `Rank`, including the program header, any pre-condition, and the post-condition.

c Verify that your loop code re-establishes the loop invariant. Remember to check that array accesses are legal and that any use of `Swap` meets its pre-condition.

The three parts carry, respectively, 25%, 30%, 45% of the marks.

End of paper