UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1999

BEng Honours Degree in Computing Part I
MEng Honours Degrees in Computing 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 City and Guilds of London Institute

PAPER 1.8

SOFTWARE ENGINEERING –
HASKELL AND PROLOG

Friday, May 7th 1999, 2.00 – 3.30

Answer THREE questions

For admin. only:
paper contains 4 questions

Section A *(Use a separate answer book for this Section. Answer questions in this section using Haskell.)*

1 A part of Pascal's triangle is shown below:

```
          1
         1 1
        1 2 1
       1 3 3 1
      1 4 6 4 1
    1 5 10 10 5 1
  1 6 15 20 15 6 1
```

and so on. The internal elements of a given row of the triangle are formed by summing adjacent elements of the previous row; the first and last elements are set to 1.

a Write a function

```
sumpairs :: [ Int ] -> [ Int ]
```

which given a list of integers of length $n > 1$ will return a new list of length n-1 in which element i contains the sum of elements i and $i + 1$ in the original list, $1 \leq i \leq n-1$. For $n < 2$ `sumpairs` should return the empty list, `[]`.

b Write a function

```
next :: [ Int ] -> [ Int ]
```

using `sumpairs` or otherwise, which given one row of Pascal's triangle as a list of integers, will deliver the next row.

c Define a recursive function

```
allrows :: [ Int ] -> [ [ Int ] ]
```

which given one row of the triangle, r say, will generate the succeeding rows of the triangle (starting from r) in such a way that:

```
pascal :: [ [ Int ] ]
pascal = allrows [1]
```

defines the complete triangle as an infinite data structure.

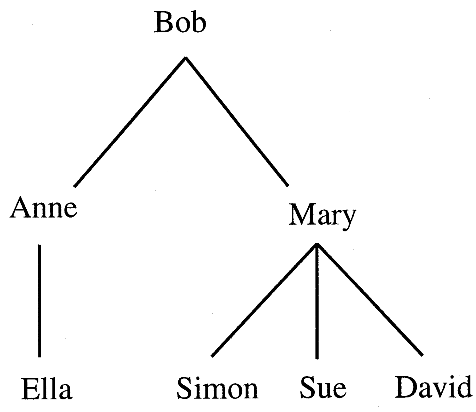
d Rewrite `pascal` in terms of the higher-order function `iterate` defined by:

```
iterate :: ( a -> a ) -> a -> [a]
iterate f x = x : iterate ( f x )
```

Credit will be given for clarity and succinctness, and the use of higher-order functions where appropriate.

The four parts carry equal marks.

2 A descendants hierarchy can be represented as a tree in which each node contains the name of an individual and a subtree for each *immediate* descendant of that individual. Individuals with no children correspond to leaves (terminals) in the tree. Below is an example showing the descendants of a person called `Bob`:



Descendant trees such as this can be represented by the following Haskell data types where parents and non-parents are explicitly distinguished and where the name of an individual is represented as a list of characters:

```
type Name = [ Char ]
data DTree = Childless Name | Parent Name [DTree]
```

a Write down the Haskell expression (of type `DTree`) which corresponds to the tree illustrated in the above figure.

b Define a function `name` which given a `DTree` will return the name of the person at the root of the given tree. Given the above tree, for example, the function should return "`Bob`".

c Using `name`, or otherwise, define a function

```
children :: DTree -> [ Name ]
```

which given the descendant tree for an individual will return a list containing the names of their children (if they have any). Given the above tree, for example, the function should return `[ "Anne", "Mary" ]` in some order.

d Using children, or otherwise, define a function

```
grandchildren :: Name -> DTree -> [ Name ]
```

which given the descendant tree for an individual will return a list containing the names of all their grandchildren (if they have any). For example, given the above tree the function should return the list `[ "Ella", "Simon", "Sue", "David" ]` in some order.

Credit will be given for clarity and succinctness, and the use of higher-order functions where appropriate.

The four parts carry equal marks.

Section B *(Use a separate answer book for this Section. Answer questions in this section using Prolog.)*

3 The relation merge(X, Y, Z) holds when Z is the list obtained by merging the (possibly empty) lists X and Y. More precisely,

> all members of X occur in Z in their same relative order,
> all members of Y occur in Z in their same relative order,
> all members of Z occur in X or in Y, and
> no duplicates in X or in Y are lost when X and Y are merged to give Z.

For example, the following all hold:

```
merge([b], [a, c, d], [a, b, c, d])
merge([a, b], [c, d], [a, b, c, d])
merge([a, b], [c, d], [c, a, d, b])
merge([a, b], [b, b], [b, a, b, b])
```

a Write a Prolog program defining merge which refers to no other relations or Prolog primitives.

b Draw the search tree for the evaluation of the query ?- merge(X, [b], [a, b]) using the program in part a, taking care to show how the value for X is computed in each successful branch.

c Write a single Prolog clause defining common(X, Y) which holds when lists X and Y have some member in common. You must define it *only* in terms of merge.

d Write a single Prolog clause defining hasdup(X) which holds when some member occurs more than once in list X. You must define it *only* in terms of merge.

e Show how your merge program can be amended, by only introducing cut (!), so that it makes merge(X, Y, Z) behave the same as append(X, Y, Z) when X and Y are given as input. Briefly explain how your cut achieves this.

The five parts carry, respectively, 30%, 40%, 10%, 10% and 10% of the marks.

4 In the following Prolog database

```
age(pete, 23).          sibs([pete, chris, mary]).
age(dave, 18).          sibs([ann, pat, john]).
age(john, 21).
age(chris, 21).
age(ann, 24).
age(jane, 20).
age(mary, 17).
age(pat, 15).
```

the predicates have these meanings:

age(X, A)	"person X has an age A"
sibs(S)	"S is a list of persons in which any two distinct members are siblings"

a Write a single Prolog clause defining sibling(X, Y) which holds when, according to the database, X is a sibling of Y. You may use any Prolog primitives.

b Write a Prolog program defining the 0-arity predicate check1 which holds when, for every clause sibs(S) in the database, S contains no duplicates. You may use any Prolog primitives.

c Write a single Prolog clause defining the 0-arity predicate check2 which holds when every sibling in the database has a numerical age. You must define it *only* in terms of sibling and age together with the primitives forall and number.

d Write a single Prolog clause defining the 0-arity predicate check3 which holds when every person having an age in the database has no other age. You must define it *only* in terms of age and the primitives forall and == ("identical to").

e Write a single Prolog clause defining elders(L) which holds when L is a list of all pairs (X, A) such that in the database person X has age A and has at least one sibling and is older than all their siblings. You must define it *only* in terms of age and sibling, together with the primitives findall, forall and > ("greater than"). It does not matter if L contains duplicates.

State which pairs in L you would expect your program to compute for the query ?- elders(L).

The five parts carry, respectively, 5%, 10%, 15%, 20% and 50% of the marks.

End of Paper