# UNIVERSITY OF LONDON
## IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

## EXAMINATIONS 1998

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*

PAPER MC1.7

TURING AND ABSTRACT DATA TYPES
Thursday, April 30th 1998, 10.00 - 11.30

*Answer THREE questions*

For admin. only: paper contains 4 questions

**Section A** *(Use a separate answer book for this Section)*

1 The Turing type `real` cannot hold numbers with great accuracy. This question is about a user-defined type `float` which can be used when more accurate real number addition is required. The declarations below are to be used:

```
const size := 8
const maxi := 10**size
type float : record
                 intpart : int
                 fracpart : int
             end record
```

So the number `3143524.6789573` will be held in `f` of type `float` like this:

```
var f : float
f.intpart  := 3143524
f.fracpart := 6789573
```

a To add two floats together requires normalising the fractional part. If

```
var f,g : float
f.intpart := 0 f.fracpart := 3
g.intpart := 0 g.fracpart := 24
```

then adding `f` and `g` should give `0.54` rather than `0.27`, which is what you would get if integer addition on the fractional parts was used without normalising. Normalising a `float` means taking the fractional part and multiplying it by 10 until the result is between `maxi` and `maxi*10`. So when `f` is normalised `f.fracpart` should contain `300000000` and when `g` is normalised `g.fracpart` should contain `240000000`.

i) Write a **function** `normalise` which takes a `float` `f` and returns a new `float` which is `f` normalised.

ii) Write a **procedure** `normal` which takes a `float` `f` as a **`var`** parameter and normalises it.

b When adding two normalised numbers together, the fractional part may generate an overflow from the fractional part to the integer part of the result. If

```
var f,g : float
f.intpart := 0 f.fracpart := 3
g.intpart := 0 g.fracpart := 84
```

then normalising `f.fracpart` and `g.fracpart` would give `300000000` and `840000000` respectively. Adding these together would give a `fracpart` of `1140000000`. This is not **well-formed**. The correct sum should have `1` as its `intpart` and `140000000` as its `fracpart`.

Write a function `carry` that takes a normalised `float` and returns a normalised float with any overflow problems sorted out. So

```
carry(1.1123456789) = 2.123456789
carry(1.123456789)  = 1.123456789
```

c Write a function add which takes two `floats` and adds them together to produce a `float`. The input `floats` have not been normalised but you can assume that the inputs to the function are well-formed (do not have carry errors).

*The three parts carry, respectively, 40%, 30% and 30% of the marks.*

2 In a version of the game Cows&Bulls, one player thinks of a 4-digit *number with no repetitions*, while the other player repeatedly tries to guess it. After each guess, *also with no repetitions*, player 1 scores the guess by stating the number of bulls and cows. A bull is a correct digit in the correct place. For example, if the secret code is 2143, then:

1234 scores 04
5678 scores 00
2134 scores 22
2143 scores 40.

For this question use the following declarations:

```
const len = 4type size : 1..lentype code : array size of int
```

a Write a function `bulls` which takes as parameters a secret code and a guess and returns the number of positions that the code and guess are identical.

b Write a function `cows` which takes as parameters a secret code and a guess and returns the number of matches of secret to guess which are in different positions.

c The Turing system has a built in procedure `randint` which takes an integer variable, a low value and a high value and returns into the integer variable a random integer between the low value and the high value. So after a call to `randint(tmp,0,9)`, `0<=tmp<=9`. Successive calls to `randint` may produce different values for `tmp`.

i) Write a function `newcode` which takes no input parameters and returns a code as result. Your code may have duplicate numbers in it.

ii) Alter your function `newcode` so that there are no duplicate numbers in it.

d Write a program that allows a player to play a *single* game against the computer. (The player is assumed to know the rules so please don't offer any help or instructions.) The program should first provide a secret code. The user should then type in a guess (you can assume that there are no errors in the input). After each guess the program should output the number of cows and the number of bulls. When the game is over, the program should output the number of guesses. Your program can call functions written to answer earlier parts of this question.

*The four parts carry, respectively, 20%, 25%, 30% and 25% of the marks.*

*Turn over ...*

**Section B** *(Use a separate answer book for this Section)*

3a Define the Abstract Data Type *List* and give the function headers and pre- and post-conditions for the access procedures.

Give the Axioms which any implementation of the ADT List must satisfy.

Give the access procedure headers with pre- and post-conditions for the Abstract Data Type Queue.

Explain the difference between the abstract data types List and Queue.

b Write the Turing code for the following high-level procedures, giving any necessary additional functions in full:

i) **function** RevQtoList (Q:Queue) : List
% pre : takes a Queue, Q
% post : returns a list containing the items of Q in reverse order of access

ii) **function** ListToQ (L:List) : Queue
% pre : takes a List, L
% post : return a Queue containing the items of L with the access order unchanged.

*The two parts carry, respectively, 60% and 40% of the marks.*

4 The Abstract Data Type *FamilyTree* is to be implemented in Turing as follows:

The access procedures:

**function** New FamilyTree (P:Person) : FamilyTree
% post : returns a new FamilyTree containing only Person P.

**procedure** AddSibling (S:Person, N:Name, **var** F:FamilyTree)
% post : adds sibling S of N to FamilyTree F.

**procedure** PrintFamily (F:FamilyTree)
% post : prints name and date of birth of all individuals in F

**procedure** AddSpouse (S:Person, N:Name, **var** F:FamilyTree)
% post : adds spouse S of N to FamilyTree F.

**procedure** AddChild (S:Person, N:Name, **var** F:FamilyTree)
% post : adds child S of N to FamilyTree F.

The following items are stored for each person

first name, surname, date of birth, date of death.

a Give type declarations for all types and structures used in your implementation.

b Write a Turing code implementation of the access procedures given.

c Draw a diagram showing how your implementation would structure the data for the following family:

Joe Smith date of birth 1.1.38, married Emily Jones d. of b. 2.2.40.
Joe and Emily's children are Peter d. of b. 10.5.63, Rachel d. of b. 10.6.64 and Rebecca d. of b. 31.10.67.

Rachel is married to Michael Brown d. of b. 10.7.62.
Rachel and Michael's children are Joanne d. of b.10.7.92 and Jake d. of b. 2.8.95

Rebecca is married to Robert Wong d. of b. 2.3.61

Joe Smith has a brother Steven Smith d. of b. 2.3.41 date of death 18.9.75; and a sister Lynda d. of b. 7.5.47 married to Roger Lloyd d. of b. 27.4.48.

*The three parts carry, respectively, 35%, 50%, and 15% of the marks.*

*End of paper*