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
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 1.7

TURING, ABSTRACT DATA TYPES AND OBJECTS
Thursday, April 30th 1998, 10.00 - 12.00

Answer FOUR questions

For admin. only: paper contains 6 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 : add 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.

Turn over ...

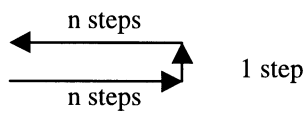
Section C *(Use a separate answer book for this Section)*

5 a Describe the concepts of object, class and inheritance and their relationships. Describe the basic principles of object-oriented programming.

b A **car** class, implemented by the OOT program *car.tu*, has the following methods:

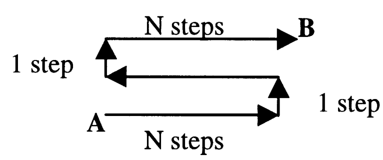
Position (X:Int)	:	set the position of a car
Move (X:Int)	:	move the car X steps
TurnLeft	:	turn the car 90 degrees left from the current direction
TurnRight	:	turn the car 90 degrees right from the current direction

i) write an OOT program to make an n-step U-turn by moving the car in one direction n steps, turning left, moving 1 step, turning left again and moving for another n steps as shown in the following diagram:



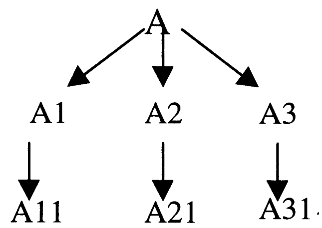
ii) write the implementation of a class **Tcar** by inheriting the class **car** and adding to it a new method Uturn (N:Int) for an N step U-turn

iii) write the implementation of a class **Scar** by inheriting the class **Tcar** and adding to it a new method Sturn(N:Int) for moving a car from position A to position B along the roads having the following intersections :



The two parts carry, respectively, 25% and 75% (25% for each subpart) of the marks.

6a Given a class hierarchy as in the following diagram:



i) Given a pointer p declared as var p : ^A2, write the possible dynamic classes of p.

ii) Given a pointer q, which at run time may locate the objects of class A2 and A21, what is the static class of q ?

iii) Given a pointer p declared as var p : ^A2, what type declarations for q' would make the assignment q': = p legal ?

b Given a class **queue**, implemented by the OOT program *queue.tu*, with the following methods:

Empty:	:	generate an empty queue
IsEmpty	:	test if a queue is empty
Enqueue (X: Int):		add an element into a queue
Dequeue (X: Int):		remove the element at the front of a queue and assign the removed element to X

A tracequeue is just like a queue, except that each time an Enqueue or Dequeue occurs, a tracequeue prints the enqueued or dequeued element. Write the implementation of the class **tracequeue** by inheriting the class **queue** and overriding the corresponding operations.

c Let the OOT program *gqueue.tu* implement generic queue containing pointers to objects of any class. The classes **A** and **B** are implemented by the OOT programs *A.tu* and *B.tu* respectively. Class **A** contains a method showA and class **B** contains a method showB to print their objects. Write a program to print the object located by the pointer at the front of a non-empty queue containing pointers of class **A** and **B**.

The three parts carry, respectively, 25%, 40%, and 35% of the marks.

End of paper