062142R

B.Sc., B.Eng. and M.Eng. Examinations 1996

Department of Computer Science

Part IIb Resit
Declarative Programming

This paper consists of the following Sections:
A: Functional Programming
B: Logic Programming
C: Functional and Logic Programming
Answer each Section in a **separate** answer booklet.

Time allowed: **Three hours**

Silent non-programmable calculators may be used.

N.B.: Candidates should answer **five** questions from this paper, as follows:

(i) **two** questions from Section A; and
(ii) **two** questions from Section B; and
(iii) **one** other question from **any** of the three sections (A, B or C).

Section A: Functional Programming

1 (20 marks)

(i) [3 marks] Briefly define the terms *higher order function*, and *partial application*. Explain how the two are related.

(ii) [5 marks] Compare the following definitions:

```
      ordered (x:y:etc) = (x <= y) && ordered (y:etc)
      ordered     _     = True

neighboursums (x:y:etc) = (x + y) : neighboursums (y:etc)
neighboursums     _     = []
```

Define a function `link` that is a higher-order generalisation of both `ordered` and `neighboursums`, retaining their common characteristics but introducing three auxiliary arguments to allow for the points at which they differ.

Redefine each of `ordered` and `neighboursums` as a partial application of `link`.

(iii) [2 marks] Give an expression-tree diagram showing how `link` uses its three auxiliary arguments to compute its result. (Assume for illustrative purposes that the final argument, a list, has four items.)

(iv) [3 marks] The function `stuttered` takes a list argument. Its truth-valued result indicates whether the argument anywhere contains two identical adjacent items. For example:

```
stuttered ["it's", "a", "a", "ghost"] ⇒ True
```

Define `stuttered` as a partial application of `link`.

(v) [5 marks] The function `stutters` is similar, but its result is a list of offending items. For example:

```
stutters ["it's", "a", "a", "ghost"] ⇒ ["a"]
```

Define `stutters` too as a partial application of link. Also define any non-primitive auxiliary functions you use.

(vi) [2 marks] A list is *smoothed* by replacing each run of two or more identical items by a single item. For example:

```
smooth ["it's", "a", "a", "ghost"] ⇒
        ["it's", "a", "ghost"]
```

Prove that `smooth` *cannot* be defined as a partial application of `link`.
(**Hint:** look for a conflicting pair of *simple* examples — show that if `link` gives the correct result for one it must give an incorrect result for the other.)

2 (20 marks)

Consider the following program.

```
wordCount :: String -> String
wordCount =
  unlines . mapState addWords 0 . lines
  where
  addWords n cs = (n', show n'++" WORDS")
    where
    n' = 1 + length (filter (== ' ') cs) + n

mapState :: (a->b->(a,c)) -> a -> [b] -> [c]
mapState f s [] = []
mapState f s (x:xs) = y : mapState f t xs
  where
  (t,y) = f s x
```

Recall that the auxiliary `lines :: String -> [String]` breaks text into a list of lines, and that `unlines :: [String] -> String` is its inverse.

(i) [4 marks] The expression `(== ' ')` is a *section* of the equality operator. Explain what this means.

The result of `filter t xs` is a list containing those items in the list `xs` that pass the test `t`. Give equations defining `filter`.

(ii) [4 marks] Give the *types* of:

1. the auxiliary function `addWords`?
2. the specialised function `mapState addWords 0`?

(iii) [4 marks] Identify **two** characteristics of the `wordCount` function that make it suitable for *interactive* application when using the *Gofer* interpreter.

Give an illustrative transcript of an interactive evaluation, starting with an appropriate invocation of `wordCount`, and assuming that subsequent user input contains just three lines:

```
Jack be nimble;
Jack be quick;
Jack jump over the candlestick.
```

(iv) [4 marks] A version of `wordCount` for *batch* (*non*-interactive) use still has type `String->String`, but its output is just a single-line report such as `3146 WORDS` giving the number of words in the entire input. Assuming a *non-empty* input, one way to modify the `wordCount` program accordingly is to insert a simple auxiliary function into the compositional pipeline at the point marked `<1>`:

```
unlines . <1> . mapState addWords 0 . lines
```

Define a suitable function to be included at `<1>`. Explain what the function does, and how this achieves the required effect in the program as a whole.

(v) [4 marks] Given an *empty* input, the output of the original `wordCount` is also empty. For the batch version a report `0 WORDS` would be better. Although it would be possible to achieve this by a contrived revision of function `<1>`, a more natural solution introduces a further auxiliary at position `<2>`.

```
unlines . <1> . mapState addWords 0 . <2> . lines
```

Define a suitable function to include at `<2>`. Again explain your solution and how it achieves the desired result.

3 (20 marks)

Consider the following definitions of list-processing functions:

```
start :: [a] -> [a] -> Bool
start []     _      = True
start (x:xs) []     = False
start (x:xs) (y:ys) = x==y && start xs ys

endings :: [a] -> [[a]]
endings []     = [[]]
endings (x:xs) = (x:xs) : endings xs

any :: (a->Bool) -> [a] -> Bool
any p [] = False
any p (x:xs) = p x || any p xs
```

(i) [6 marks] Describe the results of the functions `start`, `endings` and `any` in terms of their arguments. Give both general explanations and specific examples.

(ii) [3 marks] The primitive operator `||` can be defined by the equations:

```
True  || q = True
False || q = q
```

Show by appropriate case analysis that for *any* boolean expression `p`, the following equivalence holds:

```
p || False = p
```

(**Hint**: there are *three* cases.)

(iii) [5 marks] What is the purpose of *fold/unfold transformation*?

Apart from the invention of original equations, there are five rules for deriving equations by transformation. One is *instantiation*: copy an existing equation, but substitute a constructor pattern for a variable. Briefly state the other four.

(iv) [6 marks] A further function `part` is defined by:

```
part :: [a] -> [a] -> Bool
part xs ys = any (start xs) (endings ys)
```

Beginning with instantiation of `ys`, use fold/unfold transformation to derive an alternative definition of `part` that uses only `start` and `||` as auxiliaries. Identify the rule used in each transformation step.

Section B: Logic Programming

4 (20 marks)

This question uses the following 'key' for the predicate symbols $p/1$, $e/1$, and $f/2$, and the constant symbols jr, tc, rp:

$p(X)$	:	X is a pirate ship
$e(X)$	:	X is an enemy ship
$f(X,Y)$	:	ship X flies flag Y
jr	:	the Jolly Roger flag
tc	:	the Tricolore flag
rp	:	the ship called the Ribald Pirate

(i) [2 marks] Convert the following formula of first-order predicate calculus into a natural English equivalent:

$$\forall X (e(X) \leftrightarrow (f(X, jr) \vee f(X, tc)))$$

(ii) [2 marks] Convert the following sentence of English into *first-order predicate calculus*:

All enemy ships fly a flag, but they don't all fly the Jolly Roger.

(iii) [4 marks] Convert the following formula of first-order predicate calculus into *clausal form logic*:

$$\forall X (p(X) \rightarrow \exists Y\, f(X, Y)) \wedge \exists Z\, p(Z)$$

(iv) [6 marks] You are given the following *clauses*:

$\neg p(X) \vee e(X)$	all pirate ships are enemy ships
$\neg p(Y) \vee f(Y, jr)$	all pirate ships fly the Jolly Roger
$\neg f(Z, jr) \vee \neg f(Z, tc)$	all ships fly one or other of the flags
$p(rp)$	the Ribald Pirate is a pirate ship

Continued.

Use *resolution refutation* theorem-proving to show that not all enemy ships fly the Tricolore, i.e. in first-order predicate calculus:

$$\neg \forall V (e(V) \rightarrow f(V, tc))$$

Show your working and present your proof in the form of a refutation tree.

(v) [6 marks]

(a) [1 mark] What is a *Horn clause*?

(b) [1 mark] State an advantage that accrues from having a problem that can be expressed solely as Horn clauses.

(c) [1 mark] Say whether the problem in part (iv) involves only Horn clauses.

(d) [2 marks] Give an example from part (iv) of a *positive Horn clause* and a *negative Horn clause*.

(e) [1 mark] Explain carefully whether a standard Prolog interpreter could solve the problem given in part (iv). Justify your answer.

5 (20 marks)

(i) [8 marks] For the following query

```
?- a(X).
```

and the Prolog program below, what answer is returned first and what is returned on subsequent backtracking (i.e. if a semicolon is typed after each answer)? For full marks, you **must** show your working (e.g. in the form of a refutation tree or something akin to a Prolog trace).

```
a(X) :- a1(X).
a(X) :- a2(X).

b(one).
b(two).
```

```
b(three).

c(two).
c(three).

d(four).
d(five).

a1(X) :- b(X), c(X).
a1(X) :- b(X), !, c(X).
a1(X) :- d(X).

a2(five) :- a1(five).
```

(ii) [4 marks] Show how *negation as finite failure* can be implemented using the in-built predicates cut (!), `call` and `fail`.

(iii) [8 marks] State the outcome of each of the following Prolog queries. Give a sentence of explanation for each outcome.

(a) [1 mark] `?- X = 7 * 4.`

(b) [1 mark] `?- Z = 8, Y / 5 = Z + 2.`

(c) [1 mark] `?- X is 59 - 25.`

(d) [1 mark] `?- Y = 2, X is 8 / Y.`

(e) [1 mark] `?- X + 3 is 10.`

(f) [1 mark] `?- X = 3, X is X + 1.`

(g) [1 mark] `?- X is X + 1.`

(h) [1 mark] `?- X = 11, 15 is 24 - X + 2.`

6 (20 marks)

(i) [6 marks] Enumerate the members of each of the following Prolog lists:

(a) [2 marks] `[[a], [[ ], b], [[c, [d], e]], [ ]]`

(b) [2 marks] `[[f(g), h, h] | [f, g, h(f, g)]]`

(c) [2 marks] `[U, V, W | [X | [[Y, Z] | [ ]]]]`

(ii) [10 marks] Define three predicates `sublist1/2`, `sublist2/2` and `sublist3/2`, as follows:

- `sublist1(Xs, Ys)` is true if all elements of list `Xs` occur at least once in list `Ys`.
- `sublist2/2` is just like `sublist1/2`, except that the elements must appear in `Ys` in the *same order* as they appear in `Xs`.
- `sublist3/2` is just like `sublist2/2`, except that the elements must appear consecutively (next to each other) in `Ys`.

Give definitions for any auxiliary predicates that you use.

(iii) [4 marks] Give the mode of your definition of `sublist1/2`. **Justify** your answer.

(Ignore the possibility of partially instantiated arguments; consider only the cases where the arguments are fully instantiated or uninstantiated. Remember that a mode is only available if it raises no error condition and if the program terminates for that mode.)

Section C: Functional and Logic Programming

7 (20 marks)

(i) [4 marks] Suppose a functional program includes the defining equation:

```
h x = g (f x)
```

For *each* of the following Prolog clauses, say whether the clause would be an appropriate translation of the above equation and, if not, why not.

(a) [1 mark] `h(X) :- g(f(X)).`

(b) [1 mark] `h(X, G) :- f(X, F), g(F, G).`

(c) [1 mark] `h(X, g(f(X))).`

(d) [1 mark] `h(X, F) :- g(X, G), f(G, F).`

(ii) [4 marks] The `dropTo` function is defined by the equation:

```
dropTo i (x:xs) = if i==x then xs else dropTo i xs
```

The following clauses might be intended to define `dropTo` as a predicate, but they contain mistakes to do with the translation of the conditional. Identify and explain **two** mistakes, and give a remedy for each.

```
dropTo(I, [X|Xs], Xs).

dropTo(I, [X|Xs], Ys) :- dropTo(I, Xs, Ys).
```

(iii) [8 marks] Translate the following equations taken from a functional program to Prolog clauses defining a corresponding predicate. Briefly explain how your translation is done, identifying the correspondences between parts of the original program and parts of its translation.

Continued.

```
solve a b c = ( (-b + r)/2 , (-b - r)/2 )
      where
      r = root (b*b - 4*a*c)
```

You may assume a separately defined predicate `root/2`, where `root(N, R)` is true if `R` is the positive square root of the number `N`.

(iv) [4 marks] **Without** resorting to example programs, respond to the following assertions. (Answer briefly: give the main points only.)

(a) [2 marks] "Back-tracking cannot be expressed in functional programs."

(b) [2 marks] "Higher-order routines cannot be defined in logic programs."