062142

B.Sc., B.Eng. and M.Eng. Examinations 1997

Department of Computer Science

Part IIb
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:

```
pow 0 _ = 1
pow n x = if n `mod` 2 == 0 then pp else x * pp
          where
          pp = p * p
           p = pow (n `div` 2) x

rep 0 _ = []
rep n x = if n `mod` 2 == 0 then rr else x : rr
          where
          rr = r ++ r
           r = rep (n `div` 2) x
```

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

Redefine each of `pow` and `rep` as a partial application of `bin`.

(iii) [5 marks] The result of `pow n x` is an `n`-fold multiplication of `1` by `x`. Suppose more generally that `for n f a` is required to give the result of an `n`-fold application of function `f` to argument `a`, that is

```
f (f (...(f a) ...))
```

with `n` applications of `f`. Define the function `for` as a partial application of `bin`. If you need any auxiliary functions other than standard primitives, you should define those too.

(**Hint:** don't try to use the argument `a` in your definition; think how to compute the *function* `for n f` itself, not the result of that function for a given argument.)

Illustrate your solution at work, showing that one consequence of your definition is:

```
for 3 tail  =  tail . tail . tail
```

(iv) [7 marks] The result of `rep` is a *list* of identical items, given a size and an item. Suppose the function `tri` is required to compute *triangles* of identical items, again given a size and an item. A triangle of size n is a list of n lists, with i items in the ith list. For example:

```
tri 3 '*' ==>
  ["*",
   "**",
   "***"]
```

Define `tri` also as a partial application of `bin`. Once again include any auxiliary definitions needed.

Draw a diagram showing how your solution recursively decomposes a triangle of size 7.

2 (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. Name and briefly describe the other four rules.

(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.

3 (20 marks)

The following definition provides a simple program to browse interactively through the items of a list.

```
browse xs = unlines . mapState br xs . lines
```

```
  where
  br (y:ys) "" = (if ys==[] then xs else ys, show y)

mapState f s [] = []
mapState f s (x:xs) = y : mapState f t xs
  where
  (t,y) = f s x
```

(i) [3 marks] Show how `browse` would be run in *gofer*, using the character list `"abc"` as a simple example argument. Your explanation should include how the user steps through successive items of the list, and also what happens when the end of the list is reached.

(ii) [4 marks] The auxiliary function `mapState` plays a central role in the browser program. Give the polymorphic type of `mapState` and also a concise informal specification of it.

(iii) [4 marks] It might be useful for `browse` to display *numbered* list items, with the head as item one. Give details of small modifications to its definition that provide this extension.

(**Hint:** make the state of type `(Int,[a])` instead of just `[a]`).

(iv) [6 marks] Now formulate an additional equation in the definition of `br` so that the user can at any point enter an item number, with the effect that display continues from that item. You may use standard auxiliary functions, but should indicate these clearly. Any non-standard ones should be defined.

(v) [3 marks] The correct interactive behaviour of a program like this list browser depends on *lazy evaluation*. What is lazy evaluation, and how is it linked with interactive behaviour?

Section B: Logic Programming

4 (20 marks)

This question uses the following 'key' for the predicate symbols $n/1$, $t/1$, $b/1$, $p/1$ and $o/2$:

$$\begin{array}{rcl} n(X) & : & X \text{ is a newspaper} \\ t(X) & : & X \text{ is a tabloid} \\ b(X) & : & X \text{ is a broadsheet} \\ p(X) & : & X \text{ is a press baron} \\ o(X,Y) & : & X \text{ owns } Y \end{array}$$

(i) [2 marks] What is *clausal form logic* and why is it useful?

(ii) [8 marks] Convert the following formulae of first-order predicate calculus into *clausal form logic*. Show your working.

(a) [4 marks] $\forall X(t(X) \rightarrow \exists Y(p(Y) \wedge o(Y,X)))$

(b) [4 marks] $\neg\forall X(n(X) \rightarrow \exists Y(p(Y) \wedge o(Y,X)))$

(iii) [10 marks] You are given the following *clauses*:

$\neg n(Z_1) \vee t(Z_1) \vee b(Z_1)$	newspapers are either broadsheets or tabloids
$\left.\begin{array}{l} \neg t(Z_2) \vee p(h(Z_2)) \\ \neg t(Z_3) \vee o(h(Z_3), Z_3) \end{array}\right\}$	every tabloid is owned by a press baron (h is a Skolem function)
$\left.\begin{array}{l} n(c) \\ \neg p(Z_4) \vee \neg o(Z_4, c) \end{array}\right\}$	not every newspaper is owned by a press baron (c is a Skolem constant)

Use *resolution refutation* theorem-proving to show that there are broadsheets that are not owned by any of the press barons, i.e. in first-order predicate calculus:

$$\exists X(b(X) \wedge \forall Y(p(Y) \rightarrow \neg o(Y,X)))$$

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

5 (20 marks)

A robot is to be used to pack items into bags at a supermarket checkout. Each bag will be represented by a Prolog list. A customer will need a *set* of bags and this set will also be modelled by a Prolog list. Hence, the programs below make use of a list of lists.

(i) [4 marks] Assume that the predicate `roomFor/2` has already been defined, where `roomFor(B, I)` is true if there is room in bag `B` for item `I`.

Define `bagItem/3`, where `bagItem(I, BBefores, BAfters)` is true if `BAfters` is the new set of bags that results after bagging item `I`. Item `I` can be placed into any of the existing bags, `BBefores`, provided they have room for it, or it may be placed into a new bag.

(ii) [3 marks] Using `bagItem/3`, define a predicate `bag/2`, where `bag(Is, Bs)` is true if `Bs` is the result of bagging the list of items `Is`. (Hint: use a 3-ary auxiliary predicate.)

(iii) [9 marks] The Prolog database records that some items are heavy, some are medium and some are light, e.g.:

```
weight(persil, heavy).
weight(lard,   medium).
weight(plums,  light).
```

Redefine `bag/2` so that heavy items are bagged prior to medium ones, and medium ones prior to lightweight ones.

(iv) [4 marks] It had originally been thought that a customer's shopping would not be represented by a list of items, but by a number of clauses in the Prolog database, e.g.:

```
item(persil).
item(bakedBeans).
item(pgTips).
```

For this representation, the following (incorrect) definition of `bag/1` was written with the intention that it would give equivalent behaviour to that of `bag/2` from part (ii):

```
bag(Bs) :-
   bag([], Bs).              % [] is the state of the bags so far;
                             % Bs is where the answer will be constructed

bag(BBs, BAs) :-
   item(Item),               % Find an item in the Prolog database;
   bagItem(Item, BBs, BAs),  % Use bagItem/3 to bag it;
   fail.                     % Force a backtrack, to get the next item.

bag(Bs, Bs).                 % When there are no more items, the state
                             % of the bags so far (first argument) is the
                             % answer we want.
```

Carefully explain why this program does not solve the problem as intended.

6 (20 marks)

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

(a) [1 mark] `?- X is 60 - 25.`

(b) [1 mark] `?- X = 3 * 2.`

(c) [1 mark] `?- Y = 2, X is 6 / Y.`

(d) [1 mark] `?- X + 3 is 11.`

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

(f) [1 mark] `?- Z = 6, Y / 3 = Z + 1.`

(g) [1 mark] `?- X = 2, X is X + 1.`

(h) [1 mark] `?- X = 10, 16 is 24 - X + 2.`

(ii) [8 marks] In answering (a) and (b) below you can assume that the following two predicates are built into Prolog:

`var(X)`	tests whether `X` is currently instantiated to a variable
`number(X)`	tests whether `X` is currently instantiated to a number

For example, `?- var(V)` and `?- number(3)` would both return `yes`, whereas `?- var(3)`, `?- number(V)`, `?- var(f(X))` and `?- number(f(X))` would all return `no`.

(a) [3 marks] A Prolog program uses two different representations of arithmetic expressions. (Addition is the only operation allowed in both.)

In *representation one*, arithmetic expressions are represented by terms according to the following rules only:

- for any number `N`, `exp(N)` is an arithmetic expression;
- for any variable `V`, `exp(V)` is an arithmetic expression; and
- if `E1` and `E2` are arithmetic expressions, then `exp(E1, +, E2)` is an arithmetic expression.

In *representation two*, arithmetic expressions are represented by terms according to the following rules only:

- for any number `N`, `N` is an arithmetic expression;
- for any variable `V`, `V` is an arithmetic expression; and
- if `E1` and `E2` are arithmetic expressions, then `+(E1, E2)` is an arithmetic expression.

Define a predicate `transform/2` that converts between the two representations. For example,

```
?- transform(exp(exp(X), +, exp(exp(3), +, exp(4))), E).
E = +(X, +(3, 4))
```

(b) [5 marks] Define a predicate `simplify/2` which takes an arithmetic expression that is represented using *representation two* as its first argument and simplifies it, by evaluating any of its subexpressions that do not contain variables, returning the resulting expression in its second argument. For example,

```
?- simplify(+(X, +(3, 4)), E).
E = +(X, 7)
```

(iii) [4 marks] **Briefly** explain how well you think Prolog's treatment of arithmetic fits the logic programming ideal.

Section C: Functional and Logic Programming

7 (20 marks)

(i) [4 marks] In the instructions for translating the bodies of functions in functional programs to the bodies of clauses in logic programs, we read

> "Each application of a non-constructor becomes a separate goal..."

Using an example, explain why *non-constructors* become goals and describe how *constructor* applications are treated.

(ii) [4 marks] A naïve translation of a pair of boolean-valued functions

```
evenlength :: [a] -> Bool
oddlength  :: [a] -> Bool
```

yields the following overly-complex logic program:

```
evenlength([]) :- true.
evenlength([X|Xs]) :- oddlength(Xs).

oddlength([]) :- fail.
oddlength([X|Xs]) :- evenlength(Xs).
```

Simplify the definitions of the two predicates. Explain your simplifications.

(iii) [6 marks] A logic program includes the following definition of a predicate `sans/3`:

```
sans(E, [], []).

sans(E, [E|Xs], S) :-
   sans(E, Xs, S).

sans(E, [X|Xs], [X|S]) :-
   \+ E = X, sans(E, Xs, S).
```


Continued.

For mode `sans(++, ++, -)`, translate these clauses to equations of a functional program that define a corresponding function. Briefly explain how your translation is done, identifying the correspondences between parts of the original program and parts of its translation.

(iv) [6 marks] The prelude of a functional programming language includes the following definition of a higher-order function

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

zipWith _ _      _      = []
```

A naïve attempt at a logic program translation is

```
zipWith(F, [X|Xs], [Y|Ys], [FXY | Zs]) :-
   F(X, Y, FXY), zipWith(F, Xs, Ys, Zs).

zipWith(_, _, _, []).
```

(a) [3 marks] The translation is an illegal logic program. Why? Replace the illegal part with a legal alternative.

(b) [3 marks] Discuss the consequences if the logic program counterpart of the application `f x y`

- fails, or
- succeeds with multiple solutions.