062142R

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

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)

A function `listPic`, defined as follows, draws a tree diagram of the list of characters given as its argument.

```
listPic = unlines . foldr consPic nilPic
    where
    consPic   = zipWith (++) . itemPic
    nilPic    = ["[]",
                "  ",
                "  "]
    itemPic c = [":-",
                "| ",
                c:" "]
```

(i) [4 marks] What is *partial application*? Give two examples of its use in the `listPic` definition.

(ii) [3 marks] Recall the definition of the `zipWith` function.

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

Why is it correct to say that `zipWith` is *higher order*? Illustrate your answer with reference to the types involved in the application `zipWith (++)`.

(iii) [4 marks] By introducing an explicit argument for `listPic` and two explicit arguments for its auxiliary `consPic`, rewrite the definition of `listPic` eliminating uses of the *composition operator* and of partial application.

(iv) [5 marks] What textual display results from the application `listPic "abc"`? Explain how this display is obtained with reference to `foldr`, `zipWith` and `unlines`.

(v) [4 marks] As defined above, `itemPic` takes an argument of type `Char`. Redefine it to allow any argument of type `t` for which there is a function `show` of type `t -> String`. (Allow for results of `show` of any finite width.)

2 (20 marks)

An integer *association list* has type `[(Int,Int)]`, and each item `(i,v)` comprises a unique index `i` and an associated value `v`. In this question, such lists represent a *sparse vector or array* (in which many of the values are zero); associations are only needed for the non-zero values. Study the following definitions.

```
lookUp j [] = 0
lookUp j ((i,x):ixs) = if i==j then x
                       else lookUp j ixs

update j y [] = [(j,y)]
update j y ((i,x):ixs) = if i==j then (i,y):ixs
                         else (i,x):update j y ixs
```

(i) [3 marks] The `update` function records a list entry whatever the value of `y`. Revise the definition so that zero values are never recorded – an update with a zero value removes any non-zero entry for the same index.

(ii) [4 marks] It might be more efficient to hold the list items in index order. Further revise the definitions of `lookUp` and `update` accordingly.

(iii) [5 marks] The original definitions of `lookUp` can be recast as applications of `foldr`. Complete the following definition by providing the missing expressions at `<1>` and `<2>`.

```
lookUp j vec = foldr f <1> vec
               where f (i,x) lu = <2>
```

However, the original definition of `update` cannot be recast in a similar way. Explain why not.

(iv) [8 marks] Returning once more to the original definitions, prove by list induction on `vec` the equivalence:

```
lookUp k (update k z vec) = z
```

Carefully justify each step in your proof. Follow the structure of the definitions; for example, the proof in the inductive case will split into two sub-cases. You may assume that all the values involved are $\perp$-free.

3 (20 marks)

A simple word-processing program lists each distinct word from its input, also numbering the words in order of first occurrence. The program is interactive, reporting any new words after each line of input.

```
interWordList = concat . mapState wl (0,[]) . lines

wl (n,wsrec) inp =
        ( (n + length wsnew, wsnew ++ wsrec)
        , unlines (zipWith nw [n+1 ..] wsnew)
        )
        where
        wsnew = nub (diff (alphaBlocks inp) wsrec)
        nw n' w = show n' ++ ". " ++ w
```

All but two of the assumed auxiliaries, `alphaBlocks` and `nub`, should be familiar to you. The function `alphaBlocks :: String -> [String]` lists blocks of consecutive letters (assumed to be words) in its argument.

(i) [6 marks] The function `nub` has type `[a] -> [a]`. The result of `nub xs` contains the same items as `xs`, in the same order, but without duplicates: only the first occurrence of each item is retained. Supply the missing expressions at `<1>`, `<2>` and `<3>` to complete the following definition — use the auxiliary argument to record items already seen. **Also**, explain the role that `nub` plays in the program.

```
nub = nub' <1>
nub' acc [] = <2>
nub' acc (x:xs) = <3>
```

(ii) [4 marks] As the name `mapState` suggests, `interWordList` works by computing successive values of a state. What is the type of this state? What information about the input so far does a state value represent?

(iii) [4 marks] Give a transcript showing the `interWordList` being run within the Gofer interpreter. Suppose the user's input contains just the two lines:

```
If all the trees were one great tree,
and all the axes one great axe,
```

(iv) [6 marks] A similar program is required to list after each line of input just those words that have occurred in some previous line. Show how `interWordList` can be modified to provide this program. Explain first what the revised type of the state should be and why.

Section B: Logic Programming

4 (20 marks)

This question uses the following 'key' for the predicate symbols $p/1$, $r/1$, $m/1$, $f/1$ and $a/2$:

$$\begin{array}{rcl rcl} p(X) & : & X \text{ is a pop icon} & r(X) & : & X \text{ can read music} \\ m(X) & : & X \text{ is a musician} & f(X) & : & X \text{ is a music fan} \\ a(X,Y) & : & X \text{ admires } Y & & & \end{array}$$

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

(ii) [3 marks] *Skolemise* the following formulae of first-order predicate calculus:

(a) [1 mark] $\forall X(\neg m(X) \vee \forall Y(\neg m(Y) \vee a(X,Y))) \wedge \exists Z\, m(Z)$

(b) [2 marks] $\exists W(\neg p(W) \vee \forall X \forall Y(p(X) \vee \exists Z\, a(X,Z)))$

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

$$\forall X(m(X) \rightarrow (r(X) \wedge f(X) \wedge \forall Y(m(Y) \rightarrow a(X,Y))))$$

(iv) [11 marks] You are given the following *clauses*:

$$\left.\begin{array}{l} a(g(X_1), X_1) \vee \neg p(X_1) \vee \neg r(X_1) \\ f(g(X_2)) \vee \neg p(X_2) \vee \neg r(X_2) \end{array}\right\}$$ all pop icons who can read music are admired by a music fan (g is a Skolem function)

$r(X_3) \vee \neg m(X_3) \vee \neg p(X_3)$ — all of the pop icons who are musicians can read music

$$\left.\begin{array}{l} m(c) \\ p(c) \\ m(X_4) \vee \neg a(X_4, c) \end{array}\right\}$$ some of the pop icons who are musicians are admired by all musicians (c is a Skolem constant)

Use *resolution refutation* theorem-proving to show that some music fans are musicians i.e. in first-order predicate calculus:

$$\exists X(f(X) \wedge m(X))$$

5 (20 marks)

(i) [4 marks] The following procedure merges two *ordered lists of numbers*:

```
merger([], [], []) :- !.

merger([], [Y|Ys], [Y|Ys]) :- !.

merger([X|Xs], [], [X|Xs]) :- !.

merger([X|Xs], [Y|Ys], [X|Zs]) :- !,
   X =< Y, merger(Xs, [Y|Ys], Zs).

merger([X|Xs], [Y|Ys], [Y|Zs]) :-
   X > Y, merger([X|Xs], Ys, Zs).
```

(a) [2 marks] Explain why the cuts in the first three clauses are unnecessary.

(b) [2 marks] Using an example, explain what is wrong with the cut in the fourth clause.

(ii) [3 marks] Here is a Prolog definition of a merge sort:

```
msort(List, SortedList) :- divide(List, List1, List2),
   msort(List1, SList1), msort(List2, SList2),
   merger(SList1, SList2, SortedList).
```

Give a definition of `divide/3`, as used by this sort program to divide the items in the given list into two smaller lists of roughly equal lengths. (Do not use any cuts.)

(iii) [13 marks]

Background: You are given a *list* of records, each comprising a unique key value and some data, `[rec(Key, Data), rec(Key, Data),...]`. The key values are numbers and the list is already sorted into ascending order of key value.

Another *list* contains transactions that are to be applied to the data records. Transactions comprise a transaction type (either `insert` or `delete`), a unique key value and some data (in the case of `delete` transactions the data field is just a variable): `[trans(Type, Key, Data), trans(Type, Key, Data),...]`. This list too is already sorted into ascending order of key value. (Key values in insertion transactions *do not* already exist in the original list and key values of deletion transactions *do* already exist in the original list.)

Requirement: Define `update(OldRecs, TranRecs, NewRecs)` which applies the transactions to the original list of records to produce a new list of records ordered by the key values. Not all records will have corresponding updates and these records should be copied across to the new list without change. For example:

```
?- update([rec(1, abc), rec(2, def), rec(4, jkl)],
          [trans(insert, 3, ghi), trans(delete, 4, _)],
          NewRecs).
NewRecs = [rec(1, abc), rec(2, def), rec(3, ghi)] ;
no
```

(Again, do not use any cuts.)

6 (20 marks)

`solve/1` is an interpreter for 'pure Prolog', written in Prolog:

```
solve(true).

solve(Goal) :-
   clause(Goal, Body), solve(Body).

solve((Body1, Body2)) :-
   solve(Body1), solve(Body2).
```

(i) [3 marks] Explain briefly, with examples, how the `clause/2` primitive works.

(ii) [6 marks] Define `solve/2` (basing it on `solve/1`), where `solve/2` abandons search paths if their length exceeds a user-supplied *depth-bound*.

(iii) [8 marks] Define a different version of `solve/2` (again basing it on `solve/1`), which generates proof trees for goals that it proves. If a literal `L` is proved by virtue of subproofs $\mathtt{P}_1, \ldots,$ $\mathtt{P}n$, then the proof tree is represented as the Prolog term: `L :- P1, ..., Pn`. For example:

```
?- solve(subset([a, b], [c, b, a]), ProofTree).
ProofTree =
 subset([a,b], [c,b,a]) :-
  (member(a, [c,b,a]) :-
   (member(a, [b,a]) :-
    (member(a, [a]) :-
     true))),
  (subset([b], [c,b,a]) :-
   (member(b, [c,b,a]) :-
    (member(b, [b,a]) :-
     true)),
   (subset([], [c,b,a]) :-
    true)) ;
no
```

(The tree above is formatted for ease of comprehension. There is no requirement for you to format the output of your program.)

(iv) [3 marks] Explain why attempting to execute `solve/1` on itself fails e.g.

```
?- solve(solve(member(b, [a, b, c]))).
no
```

Section C: Functional and Logic Programming

7 (2 marks)

(i) [4 marks] When translating functional programs to logic programs, a constructor on the *right*-hand side of an equation may move to the *left*-hand side of a clause. Explain under what circumstances this is possible. Give an example where it can be done, and another where it cannot.

(ii) [4 marks] Translate the following definition of a function

```
dropTo ::  a -> [a] -> [a]
```

into a corresponding predicate `dropTo/3`.

```
dropTo x []     = []
dropTo x (y:ys) = if x==y then y:ys else dropTo x ys
```

(iii) [3 marks] In the functional language we might generalise `dropTo` to obtain a function

```
dropUntil ::  (a-> Bool) -> [a] -> [a],
```

defined as follows.

```
dropUntil p []     = []
dropUntil p (y:ys) = if p y then y:ys else dropUntil p ys
```

What simple embedding equation redefines `dropTo` in terms of `dropUntil`?

Also, what property enjoyed by functions in a functional language, but not by predicates in a logic language, prevents a direct counterpart to this generalisation in the logic definition?

(iv) [6 marks] A predicate `splitAt/4`, when used in the mode `splitAt(++,++,-,-)`, performs a similar job to `dropTo` but yields *two* segments of the list — the items *before* and the items *starting from* an element of interest.

```
splitAt(X,[],[],[]).
splitAt(X,[X|Ys],[],[X|Ys]).
splitAt(X,[Y|Ys],[X|As],Bs) :-
   \+ X=Y, splitAt(X,Ys,As,Bs).
```

Give a translation into the definition of a corresponding function

```
splitAt ::  a -> [a] -> ([a],[a]).
```

Explain how you obtain each part of the translation.

(v) [3 marks] What would be the effect of deleting the subgoal `\+ X=Y` in the final defining clause for `splitAt/4`?

Also, what control feature present in the evaluation strategy of logic programming, but not in functional programming, prevents a direct counterpart to this change in the functional definition?