062142

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

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) [4 marks] Recall the definition of `foldr`:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

What does it mean to say that `foldr` is a *higher order* function, and why is this a correct claim? Also, explain how the computation defined by `foldr` can be viewed as a generalisation of its list argument.

(ii) [4 marks] Suppose finite sets are represented as lists without repeated items. Set union and set difference can be defined using `foldr`:

```
union xs ys = foldr u xs ys
diff  xs ys = foldr d xs ys
```

Give appropriate definitions for the auxiliaries `u` and `d`. (**Hints:** you may assume a function `contains::[a]->a->Bool` that tests whether its first argument contains its second, and also `filter::(a->Bool)->[a]->[a]`.)

(iii) [4 marks] To define set intersection requires a slightly different technique:

```
inter xs ys = foldr (i xs) [] ys
```

Here `(i xs)` is a *partial application*. What does this mean? Give an appropriate definition of the function `i`, beginning with a type declaration.

(iv) [4 marks] A function `subset::[a]->[a]->Bool` tests whether all elements of its first argument are also elements of its second. This too can be defined using `foldr`:

```
subset xs ys = foldr <1> <2> xs
```

Here `<1>` and `<2>` stand for missing expressions. Complete the definition of `subset` by supplying suitable expressions, along with the definitions of any new auxiliary functions they use.

(v) [4 marks] Not *all* functions of two sets can be computed directly by `foldr` applications of this kind. Testing two sets for equality (ie. whether two lists contain the same elements, perhaps in a different order) is a case in point. Consider the proposed definition:

```
equal xs ys = foldr (e xs) (xs==[]) ys
```

Prove that it is *impossible* to define the auxiliary `e` so that `equal` works correctly. (**Hint:** consider a simple example such as `equal [0,1] [0,1]`. Show that if `e` computes the right results in this example, it must compute the wrong results in others.)

2 (20 marks)

(i) [4 marks] Consider the function `mapState` defined by the equations:

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

What is the *polymorphic type* of `mapState`? Give a concise description of its result in terms of its arguments, using an example to illustrate your answer.

(ii) [4 marks] Consider a very simple line-editor for constructing palindromes – character strings that read the same backwards as forwards.

```
pal = unlines . mapState p [] . lines
  where
  p s [c] = stateWithDisplay (c:s)
  stateWithDisplay s' = (s', display s')
  display [] = ""
  display (c:s) = reverse s ++ [c] ++ s
```

(This could be shortened by unfolding `stateWithDisplay` and `display` applications in-line, but having these auxiliaries will make modification easier in later parts of the question.) This initial version of `pal` accepts just one kind of command. What is its form and what is its effect? Illustrate your answer by giving a transcript of a Gofer evaluation of `run pal` in which the user constructs the palindrome `"-oOo-"`.

(iii) [4 marks] Extend the definition of the auxiliary `p` so that `pal` also accepts another kind of command — a line containing just two characters. The effect should be that all occurrences of the 1st character are replaced by the 2nd.

(iv) [6 marks] Show how the definition of `pal` can be further modified so that an *empty* line of input from the user acts as an undo command. It should cancel the effect of any immediately preceding command – in particular, it should cancel a preceding undo. (**Hint:** modify the state to represent a *pair* of strings.)

(v) [2 marks] Briefly explain why it is important for the interactive working of the `pal` program that `(:)` is a *non-strict* constructor.

3 (20 marks)

(i) [6 marks] Give definitions for two functions `prefixes` and `suffixes` of type `[a] -> [[a]]`. The result of `prefixes xs` should be a list of the non-empty prefixes of `xs`; the result of `suffixes xs` should be a list of the non-empty suffixes of `xs`. For example:

```
prefixes [1,2,3] = [[1], [1,2], [1,2,3]]
suffixes [1,2,3] = [[1,2,3], [2,3], [3]]
```

(**Hints:** for `prefixes`, you may find `map` useful; for `suffixes` no auxiliary function is necessary.)

(ii) [5 marks] Now consider the function `segments` defined as follows (recalling that `concat::[[a]]->[a]` concatenates a list of lists into one long list):

```
segments = concat . map prefixes . suffixes
```

Give a concise explanation of the result of `segments` in terms of its argument. Illustrate using an example. Also, what is the *polymorphic type* of `segments`? Justify your answer with reference to the types of the composed functions in its definition.

(iii) [3 marks] Given a suitably defined `product` function (like `zipWith` but computing results for *all combinations* of items from each of two lists, not just for items at corresponding positions), the following *distribution equation* holds:

```
segments (xs ++ ys) =
  segments xs
  ++ product (++) (suffixes xs) (prefixes ys) ++
  segments ys
```

Complete the following definition of `product` by supplying the missing expressions at `<1>` and `<2>`:

```
product f []     ys = <1>
product f (x:xs) ys = <2> ++ product f xs ys
```

(iv) [6 marks] When `xs = [y]`, the expression `segments (xs ++ ys)` on the left-hand side of the distribution equation reduces to `segments (y:ys)` which is a legal left-hand side for an equation in a program. Using this observation, and appealing to earlier definitions as necessary, derive by transformation a *directly recursive* definition of `segments` that uses *only* `prefixes` and `(++)` as auxiliaries. (**Note:** Give the derivation, not just a final definition.)

Section B: Logic Programming

4 (20 marks)

(i) [10 marks] This part of the question uses the following 'key' for the predicate symbols $n/1$, $nr/1$, $r/1$, $par/2$ and $eq/2$:

$$\begin{array}{rcl} n(X) & : & X \text{ is a node in a tree} \\ nr(X) & : & X \text{ is a non-root node in a tree} \\ par(X,Y) & : & X \text{ is a parent of } Y \\ eq(X,Y) & : & X \text{ and } Y \text{ are equal} \end{array}$$

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

$$\forall X(nr(X) \rightarrow \exists Y(n(Y) \wedge par(Y,X) \wedge \forall Z((n(Z) \wedge par(Z,X)) \rightarrow eq(Y,Z))))$$

(b) [8 marks] Convert the formula of first-order predicate calculus from part (a) of this question into clausal form logic.

(ii) [10 marks] A Prolog program uses two different representations of trees.

In the *clause representation*, trees are represented by `par/2` facts in the Prolog database, where `par(X, Y)` means that `X` is the parent of `Y`. For example, the tree in which `a` is the parent of `b`, `c` and `d`, and `b` is the parent of `e` and `f`, and `f` is the parent of `g` would be represented by the following clauses:

```
par(a, b).
par(a, c).
par(a, d).
par(b, e).
par(b, f).
par(f, g).
```

In the *term representation* trees are represented according to the following rules only:

- the empty tree is represented by the empty list `[]`;
- if `P` is the label on a node and `C1,...,CN` are trees, then `tree(P, [C1,...,CN])` is a tree.

The example tree would be represented as follows (shown indented for clarity):

```
tree(a, [tree(b, [tree(e, []),
                  tree(f, [tree(g, [])]) ]),
         tree(c, []),
         tree(d, []) ]).
```

A programmer needs to convert between these two representations.

(a) [4 marks] Define a predicate `clausify/1`, which, given a tree in the term representation, stores in the Prolog database (using `assert/1`) the corresponding tree in the clause representation. (The order in which clauses are asserted is of no importance.)

(b) [4 marks] Define a predicate `termify/2`, where `termify(R, T)` is true if `T` is the term representation of the tree whose clause representation is stored in the Prolog database and whose root is `R`. Use `findall/3` in your answer.

(c) [2 marks] Explain carefully why part (a) can be solved without using an 'all-solutions' predicate such as `findall/3` but part (b) cannot.

5 (20 marks)

(i) [8 marks] You are given the following *clauses*:

$\neg j(X_1) \vee \neg r(X_1) \vee \neg lp(Y_1) \vee s(X_1, Y_1)$	(All the Japanese robots can solve every logic problem.)
$\neg r(X_2) \vee \neg lp(Y_2) \vee \neg s(X_2, Y_2) \vee i(X_2)$	(Any robot who can solve a logic problem is intelligent.)
$lp(sk)$	(There is at least one logic problem. The constant *sk* is introduced through Skolemisation in the conversion to clausal form.)
$j(a)$	(Arnie is Japanese.)
$r(a)$	(Arnie is a robot.)

Use *resolution refutation* theorem-proving to answer the question: who is intelligent, i.e., in first-order predicate calculus, $\exists X i(X)$. Show your working and present your proof in the form of a refutation tree.

(ii) [12 marks] Facts and rules of a Prolog-like language are to be stored in the Prolog database using a predicate, `pclause/2`, whose first argument represents the head of the fact or rule and whose second argument represents its body, this being a list of subgoals. For example, here is the representation of `ancestor/2`:

```
pclause(parent(liz, chas), []).
pclause(parent(chas, willie), []).

pclause(ancestor(A, D), [parent(A, D)]).
pclause(ancestor(A, D), [parent(A, I), ancestor(I, D)]).
```

A simple logic interpreter for such facts and rules is given on the next page. (The interpreter is similar, but not identical, to the one discussed in the lectures. Like the underlying Prolog interpreter, it searches depth-first, works through subgoals from left-to-right and tries facts and rules in the database from top-to-bottom.)

```
solve([]) :- !.

solve([G|Gs]) :- !,
   pclause(G, B),
   append(B, Gs, BGs),
   solve(BGs).

solve(G) :-
   pclause(G, B),
   solve(B).

append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

(a) [2 marks] Why is this interpreter's implementation of resolution not sound?

(b) [2 marks] This interpreter is also incomplete. Give a short program and a query that would demonstrate this.

(c) [8 marks] Modify the interpreter so that it searches *breadth-first*.

In your answer, you may assume that the facts and rules of the Prolog-like language are restricted to being propositional, e.g.

```
pclause(alf_isa_zebra, [alf_has_hooves, alf_has_stripes]).
```

and, therefore, your interpreter need not be concerned with collecting answer substitutions.

(Hints: (1) your answer, like the depth-first interpreter, should be defined by three `solve/1` clauses; (2) the argument to the predicate `solve/1` will now be an *agenda* of goal lists; (3) use `findall/3` in your answer.)

6 (20 marks)

(i) [6 marks] State which of the following pairs of atoms unify. For those that do unify, give their most general unifiers. For those that do not, **briefly** state why they do not. (p is a binary predicate symbol, f is a binary function symbol, X, Y and Z are variables, and c and d are constants.)

(a) [2 marks] $p(X, c)$ and $p(f(Y, c), Y)$

(b) [2 marks] $p(X, f(X, c))$ and $p(d, f(Y, Y))$

(c) [2 marks] $p(X, f(X, c))$ and $p(Y, Y)$

(ii) [4 marks] An *association list* is a list of pairs. For example, an association list that associates the name `a` with the value `3`, `b` with `dec`, and `c` with `s(s(0))` might take the following form:

```
[(a, 3), (b, dec), (c, s(s(0)))]
```

You may assume that names in association lists are unique.

(a) [2 marks] Define a predicate `lookup/2` where

```
lookup((Name, Val), AL)
```

is true if `Val` is the value associated with `Name` in the association list `AL`.

(b) [2 marks] Define a predicate `update/3` where

```
update((Name, Val), OldAL, NewAL)
```

is true if `NewAL` is the association list just like `OldAL` except that, in `NewAL`, `Name` is associated with the value `Val`. Assume that the name whose value is being overwritten does already appear in the list.

(iii) [10 marks] A sequence of instructions in an imperative language is represented by a Prolog list. The instruction repertoire is as follows:

`incr(L)`	increment the value stored in location `L`
`decr(L)`	decrement the value stored in location `L`
`eq(L1, L2)`	return true if the values stored in locations `L1` and `L2` are equal
`neq(L1, L2)`	return true if the values stored in locations `L1` and `L2` are not equal
`while(T, S)`	while test `T` evaluates to true, repeatedly execute the instruction list `S`

An interpreter uses association lists to pair storage location names with the values stored in those locations, representing numbers as `0`, `s(0)`, `s(s(0))`, etc. Accesses to these association lists are made using `lookup/2` and `update/2` from part (ii).

`execute(S0, P, S)` will be true if execution of program `P` with the store starting off in state `S0` leads to a store in state `S`. For example:

```
?- execute([(a, s(0)), (b, s(s(s(0)))), (c, 0)],
       [while(neq(b, c), [incr(a), decr(b)])], S).

S = [(a, s(s(s(s(0))))), (b, 0), (c, 0)]
```

Here is a definition of `execute/3` and one clause of the definition of `interpret/3`, which is an auxiliary predicate that `execute/3` uses:

```
execute(S,  [],     S).

execute(S0, [I|Is], S) :-
   interpret(S0, I, S1),
   execute(S1, Is, S).

interpret(S0, incr(L), S) :-
   lookup((L, V), S0),
   update((L, s(V)), S0, S).
```

(a) [2 marks] Define an `interpret/3` clause for `decr` instructions (assuming a non-zero value in the named location).

(b) [2 marks] Define an `interpret/3` clause for `eq` instructions.

(c) [2 marks] The following `interpret/3` clause handles `neq` instructions. Define its auxiliary predicate `notEquals/2` *using 'pure' Prolog only.*

```
interpret(S, neq(L1, L2), S) :-
   lookup((L1, V1), S),
   lookup((L2, V2), S),
   notEquals(V1, V2).
```

(d) [4 marks] Extend `interpret/3` for `while` instructions.

Section C: Functional and Logic Programming

7 (20 marks)

(i) [4 marks] Explain why it is usually necessary, when translating functional programs to logic programs, to introduce additional variables bound to the results of intermediate computations. Why are the results of truth-valued computations an exception? Use examples to illustrate your answer.

(ii) [4 marks] The `fib` function is defined by the equations.

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

The following clauses might be intended to define `fib` as a predicate, but they contain mistakes to do with the treatment of arithmetic in logic programs. Identify and explain **two** mistakes, and give a remedy for each.

```
fib(0,1).
fib(1,1).
fib(N, FM+FM1) :- M is N-2, M >= 0, fib(M,FM), fib(M+1,FM1).
```

(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.

```
replace x y [] = []
replace x y (z:zs) =
  if x==z then y:r else z:r
  where
  r = replace x y zs
```

(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.”