UNIVERSITY OF LONDON

GOLDSMITHS COLLEGE

B.Sc. Examination 2008

COMPUTING AND INFORMATION SYSTEMS

IS53020A (CIS335)
Logic Programming

Duration: 2 hours and 15 minutes

Date and Time:

There are five questions in this paper. Your should answer no more than THREE questions. Full marks will be awarded for complete and correct answers to a total of THREE questions. Each question carries 25 marks. The marks for each part of a question are indicated at the end of the part in [.] brackets.

There are 75 marks available on this paper.

No calculators should be used.

THIS PAPER MUST NOT BE REMOVED FROM THE EXAMINATION ROOM

1. This question is about unification in Prolog.

   (a) Write down the meaning of the `=/2` predicate, using the appropriate terminology. [1]

   Answer: `=/2` *succeeds if its arguments unify and fails otherwise.*

   (b) Write down the results of the following unifications in SWI Prolog, in terms of success and failure, and of the final values of the variables where appropriate.

   i. `p( X, 2 ) = p( 1, A )` [1]

   Answer: Success: `X=1, A=2`.

   ii. `q(C, C, 1) = Q( B, A, A )` [2]

   Answer: Success: `A=B=C=1, Q=q`.

   iii. `p( [a], b ) = p( a, B )` [1]

   Answer: Failure.

   iv. `[a,b|C] = [a,B,c]` [2]

   Answer: Success: B=b, C=[c].

   (c) Write down the full Prolog first-order term unification algorithm. You may use English or Prolog, but in either case you should break your answer into clauses, one for each case.[6]

   Answer: To unify two terms:

   i. Compare their functors [1]; if they do not match, then fail [1]; otherwise...
   ii. (optional for efficiency) if the terms have different numbers of arguments, then fail; otherwise...
   iii. if the terms have no arguments then succeed [1]; otherwise...
   iv. for each pair of respective arguments [1]
      A. if one is a variable, let it be identical to the other [1]; otherwise...
      B. unify the two arguments using this procedure[1].

   (d) Explain in detail the process of unification of two terms, `p( r( A, B ), [A|C] )` and `p( X, [1,2,X,4] )`, as represented internally by Prolog. Illustrate your answer with appropriate diagrams. [12]

   Answer: The two terms are represented internally as trees [1], with `p` *as the root [1], and 2 branches [1]. Each variable and constant is the leaf of a branch [1]. The lists are represented in dot notation [1]. The order of unification is as follows:* `p` *with* `p` *[1];* `r( A, B )` *with* `X` *[1]; dot with dot [1];* `A` *with* `1` *[1];* `C` *with* `[2,r(1,B),4]` *[3].*

2. This question is about Prolog list processing and meta-programming.

   You are a software engineer who has been commissioned to write a compiler program, which translates from Prolog to another logic programming language, called Foible. The compiler is to be written in Prolog. To output the new program, the compiler should generate Prolog strings. The following questions ask you to consider how you would implement certain aspects of the compiler.

   Foible has predicates, terms and atoms, in the same way as Prolog, but the first letter of a functor or an atom is a capital letter, and the first letter of a variable is in lower case—the opposite way round from Prolog.

   (a) Given that there is a Prolog library predicate, `cap/2`, which is true when its first argument is a lower-case ASCII character code and its second is the ASCII code of the upper-case version of that character, use the meta-predicate `name/2` to define the predicate `ptof/2`, which succeeds when its first argument is a Prolog atom and its second is the corresponding Foible atom, expressed as a Prolog string. [8]

   Answer:

   ```
   pto2( Prolog, [FirstCap|Rest] ) :- [2]
         atom( Prolog ), [2]
         name( Prolog, [First|Rest] ), [2]
         cap( First, FirstCap ) [2].
   ```

   (b) Without writing code, concisely explain what is the difficulty with taking Prolog variable names, as given by a programmer, as data to a meta-program, and then changing them for use in Foible. What are the consequences of this for the resulting Foible program? [6]

   Answer: Because Prolog uses a non-ground representation [2] for variables in meta-programming, it is not possible for a meta-program to access the actual name of a variable used by the programmer [2]. This means that the variable names in the Foible program will not be the ones that the programmer originally used, but ones generated by the program itself [2].

   (c) Explain how you would keep track of the variable names used in the Prolog program, as the translator goes through it, and the corresponding Foible variable-names, stating precisely which meta-predicates in Prolog you would need to do this and why. You are not expected to write code in this answer, and you may assume that there is a predicate called `newFoibleVarName/1` which generates a new Foible variable name for you. [11]

   Answer: As the translator systematically works through the Prolog program [1], it must keep a record of the variables used in each predicate [1] (there are no global variables in Prolog, so this must be done for each predicate in isolation [1]). The most appropriate data structure for this would be a list [1] of Prolog-variable/Foible-variable pairs [1]. When a Prolog variable is encountered in the program, it must be searched for in the list [1]. If it is found, then the Foible variable name with which it is paired is returned [1]; otherwise a new one is created using `newFoibleVarName/1` *[1]. The key meta-predicates would be* `==/2` *and/or* `\==/2`*, [1] which allow comparison of Prolog variables without unifying them [2].*

3. This question is about recursion over implicit structures.

   In Prolog, it is possible to use a predicate to define an implicit data structure, over which recursion may sometimes be performed.

   (a) The predicate `s/2`, below, defines an implicit data structure. What common explicit data type in computer science corresponds with it? What constructors might you use to represent this data type explicitly in Prolog? [6]

```
s( a, b ).
s( a, c ).
s( b, d ).
s( b, e ).
s( c, w ).
s( c, v ).
```

   Answer: The data structure here is a binary [1] tree [1]. Typical constructors would be `leaf/1` *whose argument is the label of the leaf node [1] and* `branch/3` *where the arguments are the label of the node [1] and the two sub-trees [1], in the same representation [1]. (Any suitable representation is fine, but these points must be made.)*

   (b) The predicate `p/2` defines an implicit data structure. Why will this data structure be unsuitable for controlling a recursive predicate which is meant to apply an operation to each element of the data structure and then terminate? [6]

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

   Answer: The data structure here is a non-connected directed (cyclic) graph [2]. It cannot be used to control terminating recursion: there is an infinite loop in the structure, so there will be an infinite loop in the program [2]. Recursion over it cannot be used to ensure that each data item in the structure is processed because there is no node which is connected to all the others [2].

   (c) Briefly compare and contrast the effect of depth-first search (as implemented in Prolog's proof strategy, for example) and breadth-first search when they are used to search implicit data structures such as those here? In particular, state the advantages and disadvantages of each method, stating any conditions under which either will succeed while the other does not. [6]

   Answer: These methods are best compared in terms of tree search. An implicit data structure of the kind used in the question may be unfolded into a tree by allowing the repetition, in the tree, of any node visited more than once in the search. DFS searches the tree downwards, along one brach at a time, only considering alternative nodes when *the whole of a previous branch has been finished [2]. BFS searches each branch of the tree in parallel, effectively moving down the tree in layers [2]. This means that, if there are any infinite cycles in the data structure, DFS may not find a solution, because it can get stuck in an infinite branch of the search tree before it does so [1]. BFS, however, is guaranteed to find the first solution (in terms of tree depth) if one exists [1]. (Answers concerning space and time complexity are not expected here; if one is given instead of the "practical" answer above, then it can achieve full marks if it contains the right ideas concerning finding solutions.)*

(d) Write down a Prolog query which uses a Prolog built-in predicate to compute the set of values appearing in an arbitrary predicate, `p/2`, and return them as an ordered list in a variable `L`. For example, using the definition of `p/2` above, the answer would be `[a,b,c,d,e,f,g,j]`. You may, if necessary, use the `;` (or) operator. [7]

Answer: `setof( T, A^( p( T, A ) ; p( A, T )), L ).`

Answer: (with marks) `setof` *[1]*`( T, A` *[1]* `^`*[1]*`( p( T, A )` *[1]*`;` *[1]* `p( A, T )` *[1]*`), L` *[1]* `).`

4. This question is about Prolog logical operators and predicate and meta-predicate definition.

   (a) Write down the logical meaning of each of the following Prolog operators.

      i. `,` (comma) [1]

         Answer: Logical "and".

      ii. `;` (semi-colon) [1]

         Answer: Logical "or".

      iii. `\+` [1]

         Answer: Logical "not".

   (b) There is no explicit equivalent of the universal qualifier, $\forall$, (for all) in Prolog. However, it is possible to implement a `foreach/3` meta-predicate. The first argument is a list of values, the second is a variable, and the third is the goal, containing that variable, which we would like to prove true for each of the values in the first argument. Thus, the goal

```
foreach( [a,b,c], X, p( X )).
```

   will succeed if `p/1` is true of `a`, `b` and `c`. Given that a predicate is true for all the elements of a set if it is not false for each element, and using whatever built-in and library predicates you need, write down the definition of `foreach/3`. Your predicate should NOT be recursive. [11]

   Answer:

```
foreach( Values, Variable, Goal ) :-
         \+ ( member( Value, Values ),
              \+ Goal ).
```

   Answer: (with marks)

```
foreach( Values, Variable, Goal ) :- [1]
         \+ [2] ( [2] member( Variable, Values ), [2]
                    \+ [2] Goal [2]).
```

   (c) The `foreach/3` predicate, above, succeeds if the values given in its first argument make the goal in its second argument true. However, it is sometimes useful to know that, beyond this, there are no values other than those given for which the goal is true. Write down a predicate `foreachandonly/3` which fulfils this specification, using existing predicates where possible. [11]

   Answer:

```
foreachandonly( Values, Variable, Goal ) :-
                foreach( Values, Variable, Goal ),
                \+ ( call( Goal ),
                     \+ member( Variable, Values )).
```

Answer: (with marks)

```
foreachandonly( Values, Variable, Goal ) :- [1]
                  foreach( Values, Variable, Goal ), [2]
                  \+ [1] ( call [2]( Goal [2] ),
                           \+ [1] member( Variable, Values ) [2] ).
```

5. This question is about unification and negation in Prolog.

   (a) Briefly, what is the function of unification in Prolog? What important side-effect does it have? [4]

   Answer: Unification is the process of matching [1] literals together during the process of proof [1]. Its important side effect is to instantiate [1] variables [1].

   (b) Logical "not" in Prolog is captured by the procedural definition of *negation as failure* (NAF). Briefly explain what this means. [5]

   Answer: Negation as failure means that, in order to prove that a goal is false [1], we attempt to prove it true [1], and if we fail [1], then the negated goal is deemed to succeed [1], and vice versa *[1].*

   (c) What is the name of the problem sometimes caused by the improper interaction of NAF with unification in Prolog? What goes wrong with the program when it happens, and why? Give an example. Propose a general solution to the problem. [16]

   Answer: Floundering [1] happens when an uninstantiated variable [1] which appears within a negated goal [1] is instantiated during the proof of the negation [1], and is then used again later in the computation [1]. The problem is that any unifications done during the successful proof under negation are undone immediately afterwards [1], and so are not carried through to any conjoined goals [1]. This can lead to the logical result of a goal being incorrect [1]. For example, in this goal:

```
\+ X = 2, X = 5.
```

   the answer should be success, with `X = 5`*. But, in fact, the goal fails, because in trying to prove* `\+ X = 2`*, Prolog attempts to prove* `X = 2`*, where* `X` *is uninstantiated. This succeeds, so the goal fails [2]. However, if the two sub-goals are swapped, the correct logical value is obtained [1]. A general solution to the problem of floundering is to postpone [1] all non-ground [1] negated literals to the last in the computation [1]. If no un-negated goals are left at the end, but there are still negated ones, then an exception can be generated [2].*