UNIVERSITY OF LONDON

GOLDSMITHS COLLEGE

B.Sc. Examination 2006

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 Prolog program syntax, unification, datatypes and syntax-related meta-programming.

   (a) Write down the most specific syntactic category, in Prolog, of the underlined part of each of the following items. For example, the most specific category of the underlined part of `p(X)` is "variable".

      i. `q(2) :- q(a,2).` [1]

      Answer: Clause or Rule (Not Term).

      ii. `q(a(1,X)).` [1]

      Answer: Term or Argument.

      iii. `a = 1.` [1]

      Answer: Functor or Predicate Name.

      iv. `p( X ) :- \+ q(X,Y), r(Y).` [1]

      Answer: Literal or Clause Head.

      v. `p( X ) :- \+ q(X,Y), r(Y).` [1]

      Answer: Clause Body or Conjunction of Literals.

   (b) What are the effects of executing the following queries in SWI Prolog, in terms of success and failure and, where appropriate, the instantiation of variables?

      i. `a(X) = b(Y).` [1]

      Answer: Fail.

      ii. `a(X,Y) = a(1,Z), f(g(Z),X)=f(g(X),Z).` [2]

      Answer: Success: `X=Y=Z=1`.

      iii. `f(X,g(X)) = f(Z,Y), Y = Z.` [4]

      Answer: Success: `X=g(**).` *(1 for success, 2 for right unifier, 1 for exactly right unifier syntax.)*

   (c) Explain the following syntactic types in Prolog in terms of atoms, terms and numbers and truth values:

      i. `atomic` [2]

      Answer: An atomic term is either an atom or a number.

      ii. `literal` [2]

      Answer: A literal is a term whose position in a Prolog clause, at the top level, means that it is associated with a truth value during execution of a program.

(d) Explain in detail the difference between the `=/2` predicate and the `==/2` predicate, giving at least one example of a pair of arguments on which they differ. [5]

Answer: `=/2` *succeeds if its two arguments unify [1];* `==/2` *succeeds if its two arguments are syntactically identical [1]. The two predicates are the same if the terms being compared are fully ground, but not if they contain variables [1]. In the event of a variable* `X` *being unified with any structure, including another variable,* `X` *is set to contain the value with which it is unified; on the other hand a variable can only be syntactically identical with itself [1]. For example,* `f(X,Z) = f(a,Y)` *will succeed, with the unifier* `X=a, Y=Z` *(so* `Y` *and* `Z` *become the same varible); on the other hand,* `f(X,Z) == f(a,Y)` *will fail, because* `a` *(an atom) is not the same as* `X` *(a variable), and* `Z` *and* `Y`*, although both variables, are not the same variable [1].*

(e) Write down Prolog data structures which will unify with the following data and nothing else, making all of the unified values available to further unification.

i. A list containing at least 2 items. [1]

Answer: `[A,B|C]`

ii. Anything. [1]

Answer: `X` *(Any named variable; not* `_`*.)*

iii. A list of exactly 3 items, whose second element is a list containing only the atom `a`.

Answer: `[A,[a],B]` [1]

iv. `B = 24.` [1]

Answer: `C = D`

2. This question is about Prolog list processing and negation.

(a) What variable instantiations, if any, are caused by the following list unifications?

i. `[H|T] = [a,b,c].` [2]

Answer: `H=a, T=[b,c]`.

ii. `[a|b] = [a,b].` [1]

Answer: Fail.

iii. `[[X|Y]] = [[1,2,3]], [X,Y] = Z.` [3]

Answer: `X=1, Y=[2,3], Z=[1,[2,3]]`.

(b) The predicate `member/2` is true if the term in its first argument is an element of the list in its second. It is defined thus:

```
member( H, [H|_] ).
member( X, [_|T] )   :-  member( X, T ).
```

i. The `member/2` predicate, above, can be modified to succeed when its first argument is *not* a member of its second argument, by using Prolog negation, thus:

```
\+ member( Element, List ).
```

Write a predicate, `notmember/2`, which succeeds when its first argument is not a member of its second (which is assumed to be a list). Your predicate should be recursive, and should NOT use Prolog negation, EXCEPT to negate a call of `=/2`. [13]

Answer:

```
notmember( _, [] ).                    [4]
notmember( X, [H|T] ) :-               [3]
           \+ X = H,                   [3]
           notmember( X, T ).          [3]
```

ii. Using `member/2`, write a predicate, `notinboth/3`, which succeeds if its first argument is a member of either its second or its third argument, but not both, assuming they are lists. You may use Prolog negation if you wish. [6]

Answer:
```
notinboth( A, B, C )  :-  member( A, B ),
                          \+ member( A, C ).
notinboth( A, B, C )  :-  member( A, C ),
                          \+ member( A, B ).
```

3. This question is about metaprogramming.

(a) Explain the function of the following metapredicates in Prolog.

i. `nonvar/1` [2]

Answer: `ground/1` *succeeds if and only if its argument is at least partly instantiated.*

ii. `\==/2` [2]

Answer: `\==/2` *succeeds if and only if its two arguments are syntactically non-identical.*

iii. `functor/3` [6]

Answer: `functor/3` *succeeds when its first argument is a term, and second and third arguments are the functor and arity, respectively, of that term.*

(b) Explain in detail the behaviour of the metainterpreter `solve1/1`, below, and compare and contrast the effects of using the alternative, `solve2/1`, on the simple Prolog program, `p/2`, also below, as far as the first solution found. You need not discuss error handling; you will not be penalised for minor syntactic errors in your example; and you need not write out any repeated sequences of events more than once (*i.e.*, just explain that there is a repeat). Use the numbers and letters given in the comments to refer to individual clauses. [15]

```
solve1( true ).                                  % Cl. 1
solve1(( Goal1, Goal2 ))  :-  solve1( Goal1 ),   % Cl. 2
                              solve1( Goal2 ).
solve1( Goal )            :-  clause( Goal, Next ),  % Cl. 3
                              solve1( Next ).


solve2( true ).                                  % Cl. A
solve2(( Goal1, Goal2 ))  :-  solve2( Goal2 ),   % Cl. B
                              solve2( Goal1 ).
solve2( Goal )            :-  clause( Goal, Next ),  % Cl. C
                              solve2( Next ).


p( X, Y ) :- q( X, Y ), r( Y ).

q( a, b ).
q( a, c ).

r( c ).
r( b ).

?- solve( p( a, T ) ).
```

Answer: `solve1/1` *interprets Prolog programs including conjunction using the default Prolog unification mechanism.[3] It operates on the given query and program as follows.*

i. `p( a, T )` *is matched first against* `true` *from Clause 1, which fails; equally, it cannot match against* `( Goal1, Goal2 )`*, so Clause 3 is the only one that can match.* `Goal` *is unified with* `p( a, T )`*.[1]*

ii. `clause( p( a, T ), Next )` *looks for a matching clause in the database and, as a result, unifies* `Next` *with* `q( a, T ), r( T )`*.* `solve1/1` *is then applied to this term.[1]*

iii. *This time, only Clause 2 will match with the term, so* `Goal1` *is unified with* `q( a, T )` *and* `Goal2` *is unified with* `r( T )`*.[1]*

iv. *Now, the body of Clause 2 is executed. First,* `solve1( q( a, T ))`*, is executed. This can only be handled by Clause 3. A match is found,* `T` *is unified with* `b` *and* `Next` *is unified with* `true`*, because this is a fact and not a rule.[1]*

v. `true` *unifies with the argument of Clause 1 of* `solve1/1`*, so this branch of the execution is finished.[1]*

vi. *Prolog now returns to the second literal of the conjoined goal at 3(b)iv, above. The sequence for this branch is identical to that for the first literal generated by Clause 2, above. [1]*

vii. *Clause 2 is now complete, and there is nothing else to execute, so the run has succeeded, with the instantiation* `T=b`*.[1]*

viii. *In contrast,* `solve2` *executes the program in a different order, and so finds a different solution first, as follows.[1]*

ix. *As far as and including 3(b)iii, above, the execution is the same as for* `solve1`*. However, at this point,* `Goal2` *in Clause B is executed by* `solve2`*, rather than* `Goal1`*.[1]*

x. *Therefore,* `Y` *is unified with the value* `c`*.[1]*

xi. *Because* `Y` *has the value* `c`*, when* `solve2` *attempts to execute* `q( X, Y )`*, the second clause of* `q/2` *is unified, and not the first, as above.[1]*

xii. *So the first answer returned is* `T=c`*.[1]*

4. This question is about Prolog program syntax and execution.

(a) Write down the syntactic categories of the underlined parts of the following items. For example, in `p(X)`, `X` is a "variable". Part (iii) has three possible answers: give all of them.

i. `p(C,D) :- q(C,E), w(E,D)).` [1]

Answer: Argument separator (not And).

ii. `X is A + 1` [1]

Answer: Term.

iii. `p([x,y,z]).` [3]

Answer: Term, List or Argument.

(b) The following Prolog program is intended to be queried with a fully instantiated list of numbers as its first argument and an uninstantiated variable as its second; it is not intended to work in any other mode. Study and understand the program, and then answer the questions below.

```
t( List, Answer ) :-
      s( List, null, Answer ).

s( [], Answer, Answer ).
s( [Head|Tail], T, Answer ) :-
      p( Head, T, S ),
      s( Tail, S, Answer ).

p( N, null, t( N, null, null )).
p( N, t( M, T1, T2 ), t( M, T3, T2 )) :-
      N =< M,
      p( N, T1, T3 ).
p( N, t( M, T1, T2 ), t( M, T1, T3 )) :-
      N > M,
      p( N, T2, T3 ).
```

i. What is the name of the special technique applied in the second argument of predicate `s/3`? [1]

Answer: Argument 2 is an accumulator argument.

ii. What is the function of this argument in this program, in particular in relation to argument 3? [4]

Answer: It accumulates the answer as the program progresses, and eventually returns it in argument 3, when all the input data is used up.

iii. Consider the arguments of the `p/3` predicate. Given that arguments 1 and 2 are inputs

in this context and argument 3 is an output, describe what data and data structures are expected for each. (Hint: you may find it helpful to draw the structure that each clause of `p/2` produces.) [6]

Answer: Argument 1 is a number [1]; Arguments 2 and 3 are binary [1] tree structures [1], with a number associated with each node [1]; the constructors of the tree are `null/0`*, to end a branch [1], and* `t/3` *to create a node [1].*

iv. Explain the function of `p/3`. [3]

Answer: `p/3` *inserts [1] a number into a tree of numbers [1] in such a way that the numbers in the tree are ordered increasing left to right [1].*

v. Explain the function of `s/3` with respect to `p/3` and the input data given in the body of `t/2`. (There is no need to explain the operation of `p/3` again.) [6]

Answer: `s/3` *takes each [1] element of the list given as its first argument [1], and adds it, using* `p/3`*, to the tree [1], initially* `null` *[1], given as its second argument [1]. When the end of the list is reached, the resulting tree is returned in argument 3 [1].*

5. This question is about cut and Prolog execution.

   Compare the following two versions of a `partition/4` predicate, both of which contain Prolog cuts. Both versions take a number as their first argument, a list of numbers as their second argument, and return lists of numbers as their third and fourth arguments. Both versions are designed to be used with arguments 1 and 2 as input and arguments 3 and 4 as output. Study the two version of the predicate, deduce its function and answer the questions below.

```
% VERSION 1

partition( _, [], [], [] ).
partition( N, [H|T], [H|T1], T2 ) :-
     H =< N,
     !,                                % Cut 1
     partition( N, T, T1, T2 ).
partition( N, [H|T], T1, [H|T2] ) :-
     H > N,
     !,                                % Cut 2
     partition( N, T, T1, T2 ).

% VERSION 2

partition( _, [], [], [] ).
partition( N, [H|T], [H|T1], T2 ) :-
     H =< N,
     !,                                % Cut 3
     partition( N, T, T1, T2 ).
partition( N, [H|T], T1, [H|T2] ) :-
     !,                                % Cut 4
     partition( N, T, T1, T2 ).
```

   (a) What is the colour of each cut? [4]

   Answer: 1: Green; 2: Green; 3: Red; 4: Green.

   (b) Describe in detail the operation of version 1 when applied to the following query, showing variable unifications at each step, and explicitly including any backtracking behaviour: [13]

```
partition( 2, [1,3], Low, High ).
```

   Answer:

   First, clause 2 unifies, with
   {`N=2, H=1, T=[3], [1|T1]=Low, T2=High`} *[1].*
   `H =< N` *succeeds [1], so commit to this clause (*`!`*) [1], and execute*
   `partition( 2, [3], T1, T2 )` *[1].*
   Clause 2 unifies, giving
   {`N=2, H=1, T=[3], [1|T1]=Low, T2=High, H'=3, T'=[], [3|T1']=T1, T2'=T2`} *[1].*

However, `H' =< N` *fails, so this unification is undone [1]. Clause 3 then unifies, giving*

`{N=2, H=1, T=[3], [1|T1]=Low, T2=High, H'=3, T'=[], T1'=T1, [3|T2']=T2}` *[1].*

`H' > N` *succeeds [1], and we commit (*`!`*) to the clause [1]. We now have to execute*

`partition( 3, [], T1', T2' )` *[1].*

This goal will unify with only the first clause, giving the unifier

`{N=2, H=1, T=[3], [1|T1]=Low, T2=High, H'=3, T'=[], T1'=T1, [3|T2']=T2, []=T1', []=T2'}` *[1],*

which terminates the execution, with the final values

`{ Low=[1], High=[3] }` *[1].*

No other backtracking is possible [1].

(c) Now consider version 2. Explain what advantages and/or disadvantages version 2 has over version 1, in terms of execution. [2]

Answer: Version 2 is marginally faster than Version 1, because it uses mutual exclusivity between clauses 2 and 3 to avoid the need to test `H>N`.

(d) Now consider removing the cuts from both versions, as follows.

```
% VERSION 1 (no cuts)

partition( _, [], [], [] ).
partition( N, [H|T], [H|T1], T2 ) :-
     H =< N,
     partition( N, T, T1, T2 ).
partition( N, [H|T], T1, [H|T2] ) :-
     H > N,
     partition( N, T, T1, T2 ).

% VERSION 2 (no cuts)

partition( _, [], [], [] ).
partition( N, [H|T], [H|T1], T2 ) :-
     H =< N,
     partition( N, T, T1, T2 ).
partition( N, [H|T], T1, [H|T2] ) :-
     partition( N, T, T1, T2 ).
```

What correct and incorrect answer(s) could arise, in either no-cuts version of the predicate, given the following query? [6]

```
partition( 4, [1,7,3,2], Low, High ), Low = [_,_].
```

Answer: In version 1, no answers at all can arise [1], because the correct answer is prevented by the unification of `Low` *with a list of 2 elements [1]. In version 2, the query will not give the correct answer [1], for the same reason, but will give three incorrect answers:*

`Low=[1,3], High=[7,2]` *[1]*

`Low=[1,2], High=[7,3]` *[1]*

```
Low=[2,3], High=[1,7]
```
[1]