Sample Final Examination (Selected Answer Keys), Cmput 325

Consider a definition of `path/2` on a graph, by the logic program P below.

```
path(X,Y) :- connected(X,Y).
path(X,Z) :- connected(X,Y), path(Y,Z).
connected(X,Y) :- arc(X,Y).
connected(X,Y) :- arc(Y,X).
arc(1,2).
arc(2,2).
arc(2,4).
arc(3,2).
arc(5,6).
```

When P is considered a logical formula, it entails some other formulas. Show **all** of the ground instances of `connected(X,Y)` such that for each of these instances α, we have $P \models \alpha$.

Answer:
`connected(1,2), connected(2,1), connected(2,2), connected(2,4), connected(4,2), connected(2,3), connected(3,2), connected(5,6), connected(6,5)`

Consider propositional logic. Let $\Gamma = \{\neg a \vee (b \wedge c),\ a \vee \neg b \vee \neg c,\ b \vee \neg c\}$. Is it the case that $\Gamma \models a$? Verify your answer using a truth table.

Answer: No. Let $a = 0, b = 0, c = 0$. Then, clearly every formula in Γ evaluates to true, but a is false.

For each pair of lists below (in Prolog syntax), indicate whether they are unifiable. In the case that you answer is yes, also show a unifier.

`(i)` `[A,B,C]` `[[a,b]|L]` your answer: Yes, {A/[a,b], L/[B,C]

`(ii)` `[a,b|[]]` `[[A,B]]` your answer: No.

Suppose in constructing a Prolog program, we write a clause

```
p(X) :- q1(X), !, q2(X).
```

Now, someone argues that you should have placed the cut at the end, as

```
p(X) :- q1(X), q2(X), !.
```

Construct an example that shows these two ways of placing the cut in this clause gives you different effects. You should first write a program, and then add each of the above to it to produce two programs, so that the two programs behave differently for the goal `?- p(W)`. *Requirement: excluding the clauses above, your constructed program should have no more than 5 clauses.*

Let the program be:

```
q1(f(Y)).
q2(f(a)).
q2(f(b)).

Along with the first clause above, the goal ?- p(W) generates two answers
W = f(a) and W = f(b).

Along with the second, the same goal generates one answer, W= f(a).
```

Consider the following Prolog program.

```
p([],Q,Q).
p([A|L],S,SL) :-
    A > 5,
    !,
    p(L,[A|S],SL).
p([A|L],S,SL) :- p(L,S,SL).
```

Show all the answers generated by Prolog for the following goal

```
?- p([2,5,7,9,6],[],Res).

Exactly one answer:  Res = [6,9,7]
```

Given the following program, for each goal below, show the result of Prolog execution. In the case that the goal is proved, you should also show the bindings for the variables in the goal. You only need to show the first answer generated by Prolog.

```
r(e, L).
r(c(X,L1),c(X,L2)) :- r(L1,L2).
q(L1,L2) :- r(L1,L2).
q(L1,c(Y,L2)) :- q(L1,L2).


?- q(c(d,e), c(a,c(b,W))).

W = c(d,V) where V is some variable


?- q(c(u,W), c(U,c(b,d))).

U = u, W = e


?- q(c(a,b), c(A,c(B,e))).

No.
```

What are the three ingredients of a constraint satisfaction problem (CSP)? That is, when you formulate a problem as a CSP, what are the three things you must specify?

Ansewr: Variables and their domains, and constraints.

What is a *constraint store*? You may answer this question by saying who is processing the constraint store at runtime.

Answer: A constraint store is a collection of the primitive constrains, decomposed from user specified constraints. It is checked at runtime for consistency whenever the state of these constraints is changed, e.g., a variable is assigned a value.

The following is a constraint logic program without constraints. The purpose is to test your understanding of the basic solving method of CLPFD. Show **all** the answers that will be returned by Sicstus Prolog for the goal: `?- solve(S)`. Show them in the order in which they are generated.

```
:- use_module(library(clpfd)).
solve(Sol) :-
    Sol = [V1,V2,V3],
    domain([V1],2,3),
    domain([V2,V3],5,6),
    labeling([],Sol).

% goal
?- solve(S).

V1 V2 V3

2  5  5
2  5  6
2  6  5
2  6  6
3  5  5
3  5  6
3  6  5
3  6  6
```

Given a list of variables over some domain, we want these variables to be assigned so that the resulting list is in increasing order. Let's write a constraint logic program using CLPFD for this.

```
:- use_module(library(clpfd)).
ordered(N,Sol) :-
    length(Vars,N),  % length/2 is a built-in predicate
    domain(Vars,1,9),
```

```
    constraint(Vars),
    labeling([],Vars),
    Sol=Vars.
```

You should define `constraint(Vars)` so that `Vars` is a list of values in increasing order. For example, if `N=5` then `Sol=[1,2,3,4,5]` is a solution. *Requirement: since in general the value of N can be arbitrary, you should give a recursive definition.*

```
constraint([]).
constraint([A]).
constraint([A,B|L]) :-
       A #< B,
       constraint([B|L]).
```

Consider three variables and their domains as given below

X: its domain is $\{2, 3, 4, 5, 6\}$
Y: its domain is $\{4, 5, 6, 7\}$
Z: its domain is $\{4, 5, 6, 7\}$

and the constraints

$$Y > Z + 1 \quad \text{and} \quad X \leq Y - 1$$

For each variable, indicate which of its domain values cannot participate in any solution. *Do not guess! Wrong answers can reduce your marks gained from correct answers.*

Answer: Let us apply arc-consistency. The new domain for Y is $\{6, 7\}$ and for Z is $\{4, 5\}$. The domain for X is unchanged.

Given a list L of elements, each of which represents a course and its enrollment, e.g. `[cmput325, [john, lily, ken,...]]`, define a predicate

```
courses_taken(+L,+Name,-Courses)
```

such that given such a list L and a student Name, Courses is bound to a list of courses taken by the student.

```
courses_taken([],_,[]).
courses_taken([[C, EnrollList] |L],Name,[C|R]) :-
    member(Name, EnrollList),
    !,
    courses_taken(L,Name,R).
courses_taken([_|L],Name,R) :-
    courses_taken(L,Name,R).
```

The question on the subset/2: The question was not formulated very well. It's fine to use lists to represent sets, as we did in Assignment 3, or as defined below:

```
subset([],[]).
subset([_|L],S1) :- subset(L,S1).
subset([A|L],[A|S1]) :- subset(L,S1).
```

If we query

```
?- subset([a,b,c],W).
```

we will have all subsets bound to W, though elements are ordered in the same way as in the given list. The problem arises if we do subset testing, e.g.,

```
?- subset([a,b,c],[b,a])
```

which fails. To resolve this problem, we can define

```
subset-test(S,S1) :- subset(S,W), permutation(W,S1).
```

where permutatoin/2 can be found in the lecture notes.

Given the predicate subset/2, write a goal that uses findall/3 to find all subsets of a list, say [a,b,c,d].

```
allSubsets(Set, Allsubsets) :-
    findall(S, subset(Set,S), Allsubsets)
```

Given a list L of lists, we want to remove those lists in L that are not minimal in the following sense: L1 in L is minimal if there is no other list L2 in L such that every element of L2 is an element in L1. Define a predicate

```
minimize(+L,-LL)
```

such that LL is L except that non-minimal lists in L are removed. *Hint: you may use the predicate subset/2 defined above.*

```
minimize(L,LL) :- mini(L,L,LL).
mini([],_,[]).
mini([A|R],L,LL) :-
   superset_of_some(A,L),
   !,
   mini(R,L,LL).
mini([A|R],L,[A|LL]) :-
   mini(R,L,LL).
```

```
superset_of_some(A,[A|R]) :-
   !,
   superset_of_some(A,R).
superset_of_some(A,[B|R]) :-
   subset(B,A),
   !.
superset_of_some(A,[B|R]) :-
   superset_of_some(A,R).
```

Consider evaluating the following lambda expression using the context-based interpreter.

```
((lambda (f) (f 2 3)) (lambda (x y) (* x (+ y 1))))
```

(i) Show the context when the subexpression `(f 2 3)` is evaluated.

Answer: Let CT0 denote the initial context.
{f -> [(lambda (x y) (* x (+ y 1))), CT0]} U CT0

(ii) Show the context when the subexpression `(* x (+ y 1))` is evaluated.

Answer: {x -> 2, y -> 3} U CT0

Is the λ-expression $(\lambda xy \mid w)$ equivalent to $(\lambda x \mid w)$? Why?

Answer: No, when they apply to the same expression, say NM, they produce different results.

$$(\lambda xy \mid w)NM \rightarrow w$$
$$(\lambda x \mid w)NM \rightarrow wM$$

Write a Lisp program

```
(defun prefix (L) ......)
```

which returns a list of all prefixes of L. The order in which the prefixes appear in the resulting list is unimportant. E.g.

```
(prefix '(a b c)) => (nil (a) (a b) (a b c))  or  ((a b c) (a b) (a) nil)
```

```
(defun prefix (L)
    (if (null L)
        (cons nil nil)
        (cons L (prefix (rmLast L)))
```

```
    )
)

(defun rmLast (L)
    (if (null (cdr L))
        nil
        (cons (car L) (rmLast (cdr L)))
    )
)
```