Open-book Test. Prolog interpreters allowed.

No collaboration with others allowed in any form.

Part 0. About class presentation:

- **How many assigned presentations have you done in the class this semester?**
- **In a 0-10 scale with 10 meaning the best, how would you rate the effort you have made in the preparation for the presentations on average?**

Part I. Pure Prolog

Hint: Note that all the four problems in this part are conceptually very much related to the two Pure Prolog programs you have written in Quiz #3.

- In Quiz 3, you are able to define a *compact / 2* predicate such that given a list Xs, *compact*(Xs, Ys) will bind Ys to a list that contains every member in Xs exactly once. For example, given the concrete list [4,5,4,6,1,2,1] compact([4,5,4,6,1,2,1], Ys) should bind Ys to a list containing the each of the five distinct members 4,5,6,1,2 exactly once.
- Also in Quiz 3, you are able to define a *numMembers / 2* predicate such that given a list Xs, *numMembers(*Xs, N) will bind N to the distinct number of members in the list. For example, *numMembers(*[4,5,4,6,1,2,1] , N) should bind N to 5 since there are 5 distinct members in the list even if the the length of list is 7.

1. Two lists Xs and Ys such as [4,5,4,6,1,2,1] and [5,4,6,1,2,1] are said to be equivalent in their contents; in other words, every member in Xs appears in Ys and every member in Ys appears in Xs. Define a *equivalentContents/ 2* predicate such that *equivalentContents* (Xs, Ys) is true if and only Xs and Ys are both lists and they have equivalent contents.

2.We can represent a the mathematical concept of sets as lists. For example, the list [4,5,6,1,2] can represent the a set of five distinct members 4,5,6,1 and 2. Define a *setIntersction/ 3* predicate such that given two lists Xs and Ys representing two sets *setIntersction* (Xs, Ys, Zs) will bind Zs to the intersection of Xs and Ys. For example, *setIntersction* ([1,2,3,4], [2,4,6], Zs) should bind Zs to a (compact) list containing the two common members 2 and 4.

3. Define a *setUnion/ 3* predicate such that given two lists Xs and Ys representing two sets *setUnion* (Xs, Ys, Zs) will bind Zs to the union of Xs and Ys. For example, *setUnion* ([1,2,3,4], [2,4,6], Zs) should bind Zs to a (compact) list containing the five members 1,2,3,4 and 6.

4. Define a *setDiff/ 3* predicate such that given two lists Xs and Ys representing two sets *setDiff* (Xs, Ys, Zs) will bind Zs to a (compact) list containing each member that appears in Xs but not in Ys. For example, *setDiff* ([1,2,3,4], [2,4,6], Zs) should bind Zs to a (compact) list containing the two members 1 and 3.

Part II. Eclipse Constraint Logic Programming

5. **(i)** Let us assume that we load the ***ic*** library before executing the query. Show how you can write a Eclipse program to find a pair of **non-negative real numbers** X and Y that minimizes the cost function X+Y under the constraints that X and Y are non-negative, $5X+2Y \leq 10$, and $3X+4Y \leq 12$. **(ii)** Let us assume that we load the ***ic*** library before executing the query. Show how you can write a Eclipse program to find a pair of **non-negative integers** X and Y that minimizes the cost function X+Y under the constraints that X and Y are non-negative, $5X+2Y \leq 10$, and $3X+4Y \leq 12$ **and $3X+2Y \geq 6$** .

6. Consider the coin program in Fig. 12.2 of Chapter 12. **(i)** What is the role of the *check(Pockets)* predicate in the program? Explain it succinctly. **(ii)** What is the purpose of having *once(labeling(ConinsforPrice))* in the *check(Pockets)* predicate? Explain it succinctly. **(iii)** Does the program still work to find solutions correctly if we change *once(labeling(ConinsforPrice))* into simply *labeling(ConinsforPrice)*? If it does not work, explain why it does not? If it still works, what is the difference we'll see between these two versions and explain why that happens?

7. Based on Section 8.5 on non-logical variables, write a predicate *threeTimes* such that *threeTimes(Q)* will succeed if and only if the query *Q* can succeed exactly for three times (i.e. *Q* has exactly 3 different solutions).

8. **(i)** Consider the *drop_until (List, Final, Min)* predicate in the *GEN_SUDOKU* program in Fig. 12.6 of Chapter 12 for generating random Sudoku puzzles. Does the program still always work to correctly generate random Sudoku puzzles with unique solutions if we change *one(Sudoku(Test))* into *once(Sudoku(Test))* in the body of the *drop_until (List, Final, Min)* predicate? If it does not work, explain why it does not? If it still works, what is the difference we'll see between these two versions and explain why that happens? **(ii)** Consider the *add_until (Board, List, Final, Min)* predicate in in the other *GEN_SUDOKU* program in Fig. 12.7 of Chapter 12 for generating random Sudoku puzzles. Does the program still always work to correctly generate random Sudoku puzzles with unique solutions if we remove *var(Val)* in the body of the *add_until (Board, List, Final, Min)* predicate? If it does not work, explain why it does not? If it still works, what is the difference we'll see between these two versions and explain why that happens?