cps721: Assignment 2 (100 points).
Due date: Electronic file - Monday, October 12, 2020, 21:00 (sharp). Read Page 5.

You must work in groups of TWO, or THREE. You cannot work alone.
YOU SHOULD NOT USE ";" "!" AND "–>" IN YOUR PROLOG RULES.

You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructor. You cannot use any external resources (including those which are posted on the Web) to complete this assignment. Failure to do this will have negative effect on your mark. By submitting this assignment you acknowledge that you read and understood the course Policy on Collaboration in homework assignments stated in the CPS721 course management form.

1 (24 points). For each of the following pairs of Prolog lists, state which pairs can be made identical, and which cannot. **Write brief explanations** (name your file **lists.txt**): it is not acceptable to give an answer without explanations. For those pairs that mention variables, and that can be made identical, give the values of their variables that make the two lists the same. As a proof for your answer show transformation from one representation to another (e.g., from ","-based notation to "|"-based notation, or vice versa, when possible). Make sure that you apply only equivalent transformations to a list when you rewrite a list into a different representation. *You lose marks if you give only short answers, but do not explain.*

```
[X | [ Y, [a, Y]]] and [a, b, [X | [Y]]]
[X, [a,Y|V]]  and  [V, [X,d,a]]
[[q,p | [r,s]],[]]   and  [X|Y]
[[w,a,r],[r,a,n,t]]  and [[X|[Y|[Z]]] | [[Z | W]]]
[l,a,m,b,[d,a]]     and    [X,Y | [V,W | [Z,X]]]
[cps109,mth110 | [X | []]] and [P | [Q,[pcs110] | S]]
[P,a,a | [d,P|R]]  and  [[a,b,c]|[X,Y,Z]]
[b, U, V | W]  and  [V | [[c,V] | [b | U]]]
```

Handing in solutions: An electronic copy of your file **lists.txt** must be included in your **zip** archive.

2 (32 points). In this part of the assignment you are asked to implement in Prolog a few programs with recursion over lists. If you wish, you may use (you do not have to) any of the programs we wrote in class, but if you do, be sure to include them in your program file. (If one of the predicates that you would like to use is a part of the ECLiPSe Prolog's standard library of predicates, then rename it and provide rules for the renamed predicate. Read details in the handout "How to use Eclipse Prolog in labs" posted on D2L). You can**not** use programs that we did not discuss in class. Test each of your programs on some examples of your choice (including queries with variables when possible). Keep all these programs in one file named **recursion.pl**

1. The predicate *minim(IntList,Min)* is true if *Min* is the minimum of the integers in a given non-empty list *IntList*. For example, a query
   ?- minim([29,1,8,167], X).
   returns the answer $X = 1$, a query
   ?- minim([12,123,456,12,78,999,123,12],X).
   returns the answer $X = 12$, but a query *?- minim([99,2,17,155],17).*
   must return the answer *no*. Write a recursive program that implements this predicate using any arithmetical operators, but you cannot use any auxiliary predicates. You can assume that in any query the first argument will be a list of integer numbers (input to your program), and the second argument will be either a variable representing the result that has to be computed (the minimal element of the input list) or an integer representing the result to be verified.

2. A polynomial $\mathcal{P}(x_1, x_2, \ldots, x_k)$ is a finite length expression constructed from variables $x_1, \ldots, x_k$ and constants, by using the operations of addition, subtraction, multiplication, and constant non-negative integer exponents. For example, $x^2+4\cdot x-7$ is a polynomial, but $x^2+\frac{4}{x}-7\cdot x^{3/2}$ is not, because its second term involves division by the variable x and also because its third term contains an exponent that is not an integer. Below we consider only polynomials in a single variable x: $a_n\cdot x^n+a_{n-1}\cdot x^{n-1}+\cdots+a_1\cdot x+a_0$. Each polynomial can be represented uniquely by its constants $a_0, \ldots, a_n$ (some or all of them can be equal to 0), that we implement as a list: $[a_0, \ldots, a_n]$. Thus, instead of polynomials, we can work with lists of numbers. Now, suppose that we are given two polynomials: $Pol_1 = [a_0, \ldots, a_n]$ and $Pol_2 = [b_0, \ldots, b_n]$. The predicate $polAdd(Pol_1, Pol_2, PolynSum)$ holds if $PolynSum$ is the sum of two given polynomials represented by the lists Pol_1 and Pol_2. The predicate $constMult(List, C, Result)$ means that multiplying a given polynomial $List$ with

a given constant C yields a new polynomial $Result$. Finally, the predicate $polMult(Pol_1, Pol_2, Product)$ holds if the polynomial $Product$ results from multiplication of two given polynomials Pol_1 and Pol_2. First, you have to implement the predicates $polAdd$ and $constMult$. Second, implement the predicate $polMult$. ***Hint:*** Suppose $Pol_1 = c + x \cdot P$, where P is a polynomial. Then multiplying Pol_1 with Pol_2 yields $(c + x \cdot P) \cdot Pol_2 = c \cdot Pol_2 + x \cdot (P \cdot Pol_2)$. Use this when you write a recursive program for polynomial multiplication. Once you have completed your program, test it on the following and on several other queries of your choice (you must run at least 10 other tests):

```
?- polMult([1, 1], [1, 1], X).        /* (1 + x) · (1 + x) = 1 + 2 · x + x^2 */
X = [1, 2, 1]
?- polMult([1, 1], [1, -1], X).       /* (1 + x) · (1 − x) = 1 − x^2 */
X = [1, 0, -1]
?- polMult([1, 1], [1, 0, -1], X).    /* (1 + x) · (1 − x^2) = 1 + x − x^2 − x^3 */
X = [1, 1, -1, -1]
?- polMult([1,2,-1],[1,-2,1],X).      /* (1 + 2 · x − x^2) · (1 − 2 · x + x^2) = 1 − 4 · x^2 + 4 · x^3 − x^4 */
X = [1, 0, -4, 4, -1]
```

Handing in solutions: (a) An electronic copy of your file **recursion.pl** with all your Prolog rules must be included in your **zip** archive; (b) your session with Prolog, showing the queries you submitted and the answers returned (the name of the file must be **recursion.txt**). It is up to you to formulate a range of queries that demonstrates that your programs are working properly.

3 (44 points). In Part 3 of the previous assignment we considered a decision tree that classifies a given set of 14 records (personal applications for a loan) into 3 categories according to the individual credit risk ("*high*", "*moderate*", "*low*"). In this questions, we assume that atomic statements representing 14 people are given in a different format as follows:

```
value(historyK,p1,bad). value(debt,p1,high). value(collateral,p1,none).
value(hist,p2,unknown). value(debt,p2,high). value(collateral,p2,none).
value(hist,p3,unknown). value(debt,p3,low). value(collateral,p3,none).
value(hist,p4,unknown). value(debt,p4,low). value(collateral,p4,none).
value(hist,p5,unknown). value(debt,p5,low). value(collateral,p5,none).
value(hist,p6,unknown). value(debt,p6,low). value(collateral,p6,adequate).
value(historyK,p7,bad). value(debt,p7,low). value(collateral,p7,none).
value(historyK,p8,bad). value(debt,p8,low). value(collateral,p8,adequate).
value(historyK,p9,good). value(debt,p9,low). value(collateral,p9,none).
value(historyK,p10,good). value(debt,p10,high). value(collateral,p10,adequate).
value(historyK,p11,good). value(debt,p11,high). value(collateral,p11,none).
value(historyK,p12,good). value(debt,p12,high). value(collateral,p12,none).
value(historyK,p13,good). value(debt,p13,high). value(collateral,p13,none).
value(historyK,p14,bad). value(debt,p14,high). value(collateral,p14,none).
```

The predicate $value(Attribut, Person, Boolean)$ holds if in $Person$'s application for a loan, the value of $Attribute$ is $Boolean$ (this is because all attributes that we would like to consider have only 2 possible values). In the 1st assignment, we had the attribute representing *credit history* that could have one of the following values: *bad, unknown, good*. Since we would like to consider here only boolean attributes, we replace it with the two new attributes. The first, *hist*, represents a test whether a person's credit history is known or not: this new attribute can have only two values *known*, and *unknown*. The second, *historyK*, representing known credit history, also can have only two possible values *bad* or *good*. After this minor modification, each attribute can take only one out of two possible values.

In this part of the assignment, you have to write recursive programs that use the predicate $value(A, P, V)$. Each of the programs have to implement the predicates specified below. In addition, you have to test each of your programs on examples provided below and on at least 10 other examples of your choice. The file with atomic statements is available for download from the Assignments folder.

- The predicate $sameValue(Att, Examples, Val)$ is true if all elements in the list $Examples$ agree that Val is the value of attribute Att.

```
?- sameValue(debt, [p3, p4, p5, p6, p7, p8, p9], X).
X = low
```

```
?- sameValue(historyK, [p9, p10, p11, p12, p13], X).
X = good
?- sameValue(historyK, [p8, p9, p10, p11, p12, p13], X).
No
?- sameValue(historyK, [p2, p3, p4, p5], X).
No
?- sameValue(hist, [p2, p3, p4, p5], X).
X = unknown
```

- The predicate $divide(Examples, Att, Boolean, Pos, Neg)$ is true if Pos is the sub-list of the positive examples extracted from the list of $Examples$ with attribute Att having value $Boolean$ and Neg is the sub-list of the remaining examples with attribute Att having the other value.

```
? - divide([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14],debt,X,Pos,Neg).
X = high
Pos = [p1, p2, p10, p11, p12, p13, p14]
Neg = [p3, p4, p5, p6, p7, p8, p9]
?- divide([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14], hist, X, Pos, Neg).
X = unknown
Pos = [p2, p3, p4, p5, p6]
Neg = []
?- divide([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14], historyK, X, Pos,Neg).
X = bad
Pos = [p1, p7, p8, p14]
Neg = [p9, p10, p11, p12, p13]
?- divide([p1,p2,p3,p4,p5,p7,p9,p11,p12,p13,p14], collateral, X, Pos, Neg).
X = none
Pos = [p1, p2, p3, p4, p5, p7, p9, p11, p12, p13, p14]
Neg = []
?- divide([p1,p2,p3,p4,p5,p7,p9,p11,p12,p13,p14], collateral, adequate, Pos, Neg).
Pos = []
Neg = [p1, p2, p3, p4, p5, p7, p9, p11, p12, p13, p14]
```

- The predicate $count(Examples, Att, Bool, T, F)$ is true if there are T instances in $Examples$ with attribute Att having value $Bool$ and F instances with attribute Att having the opposite value.

```
?- count([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14],debt,X,T,F).
X = high
T = 7
F = 7
?- count([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14], hist, X, T, F).
X = unknown
T = 5
F = 0
?- count([p1, p7, p8, p9, p10, p11, p12, p13, p14], hist, good, T, F).
T = 0
F = 0
```

Handing in solutions. An electronic copy of: (a) your program (the name of the file must be **data.pl**) that includes all your rules and atomic statements. (b) An electronic copy of your session with Prolog that shows clearly the queries you submitted to Prolog, and the answers that were returned. The name of this file must be **data.txt**

4. Bonus work (**up to 40 points**):

To make up for a grade on another assignment or test that was not what you had hoped for, or simply because you find this area of Artificial Intelligence interesting, you may choose to do extra work on this assignment. *Do not attempt any bonus work until the regular part of your assignment is complete. This part of the assignment is more challenging than questions in the regular part of your assignment.* Write whether this bonus question was implemented by all people in your team (in this case bonus marks will be divided evenly between all students) or whether it was implemented by one person only (in this case only this student will get all bonus marks). Wrong or incomplete solutions may be graded as 0, i.e., no part marks.

The standard library of Prolog includes the predicates *union(List1,List2,U)* and *intersection(List1,List2,Common)*. Their intention is to provide functionality similar to familiar set operations. For example, *union(List1,List2,U)* succeeds if U is the list which contains the union of elements in $List1$ and those in $List2$. Similarly, *intersection(List1,List2,Common)* succeeds if $Common$ is the list of elements which contains elements common both to $List1$ and $List2$. However, implementation of these predicates is not as general as it should be. Also, there is an implementation of the predicate $intersection$ in lecture notes, but it cannot handle all cases. For example:

```
?- union([c, a, 2, b], [2, b, c, a], [b, 2, c, a]).
No
?- union([c, a, 2, b], [2, b, c, a], X).
X = [2, b, c, a]
?- union([c, a, 2, b], [2, b, c, a], [a, b, c, 2]).
No
?- intersection([c, a, 2, b], [2, b, c, a], [a, b, c, 2]).
No
?- intersection([c, a, 2, b], [b, c, a, 2], [a, b, c, 2]).
No
?- intersection([c, a, 2, b], [b, c, a, 2], [2, a, b, c]).
No
?- intersection([c, a, 2, b], [b, c, a, 2], X).
X = [c, a, 2, b]
```

As these examples demonstrate, for some ordering of elements in the resulting list, the queries fail, but as we know from the set theory, the order of elements in the resulting list should not matter. In other words, in a correct implementation, elements in the union or in the intersection can occupy any positions in the list. Write the recursive programs that implement union, intersection, and a helping predicate *permutation* correctly. Your task is to provide a correct implementation with the following behavior:

```
?- myUnion([a, 2, b], [2, b, a], [b, 2, c, a]).
No
?- myUnion([a, 2, b], [2, b, c, a], [b, 2, c, a]).
Yes
?- myUnion([c, a, 2, b], [2, b, c, a], [b, 2, c, a]).
Yes
?- myUnion([c, a, 2, b], [2, b, c, a], [a, b, c, 2]).
Yes
?- myUnion([a, 2, b], [2], X).
X = [a, b, 2]
Yes (0.00s cpu, solution 1, maybe more)
X = [b, a, 2]
Yes (0.02s cpu, solution 2, maybe more)
X = [b, 2, a]
Yes (0.04s cpu, solution 3, maybe more)
?- myIntersection([c, a, 2, b], [2, b, c, a], [a, b, c, 2]).
Yes
?- myIntersection([c, a, 2, b], [2, b, c, a], [2, a, b, c]).
Yes
```

```
?- myIntersection([c, a, 2, b], [2, b, c, a], [2, a, b, c, 2]).
No
?- myIntersection([c, a, 2, b], [2, b, c, a], [a, 2, a, b, c, 2]).
No
```

1. *permutation(List1,List2)*: *List2* is a list which is rearrangement of elements from *List1* in a different order. Your program should be able to compute successively all possible rearrangements (when you press "more"). Examples of queries:

```
The following all succeed                    The following fail
permutation([a,b,c],[b,a,c]).                  permutation([c,d],[c]).
permutation([a,k,l,b,c,m],[a,b,c,k,l,m]).      permutation([p,q],[a,q,p]).
```

2. *myUnion(List1,List2,List3)* is true if *List1, List2* are given non-empty lists and *List3* contains all elements of *List1, List2*, but it contains each element exactly once. Note that if *List3* is a variable, then your program must compute the union of elements from *List1, List2* in any order. If *List3* is a list of elements given also to your program, then your program must verify whether elements of *List3* are the union of elements from *List1, List2*. (Hint: use your program *permutation(L1,L2)*. Note that elements of *List3* can be arranged differently from their order of occurrence in *List1, List2*.) Examples of queries:

```
The following all succeed                          The following fail
myUnion([k,a],[d,a,t],[t,a,d,k]).                    myUnion([a,b],[k,l,m],[k,l,m]).
myUnion([b,b,c],[99|[99,22]],[b,c,99,22]).           myUnion([c],[c],[c,c]).
```

3. *myIntersection(List1,List2,List3)* is true if *List1, List2* are given non-empty lists and *List3* contains only common elements from *List1, List2*, and also it contains each element exactly once. Note that if *List3* is a variable, then your program must compute the intersection of elements from *List1, List2* in any order. If *List3* is a list of elements given also to your program, then your program must verify whether elements of *List3* are the intersection of elements from *List1, List2*.

Handing in solutions: (a) An electronic copy of your file **bonusA2.pl** with all your Prolog rules must be included in your **zip** archive; (b) your session with Prolog, showing the queries you submitted and the answers returned (the name of the file must be **bonusA2.txt**). It is up to you to formulate a range of extra 10-15 queries that demonstrate your programs work correctly.

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked from the Assignments Web page at

`http://www.scs.ryerson.ca/~mes/courses/cps721/assignments.html`

If your question remains unanaswered, then read previously answered questions again.

If you write your code on a Windows machine, make sure you save your files as plain text that one can easily read on Linux machines. Before you submit your Prolog code electronically make sure that your files do not contain any extra binary symbols: it should be possible to load to load `recursion.pl` or `data.pl` into a recent release 6 of ECLiPSe Prolog, compile your program and ask testing queries. TA will mark your assignment using ECLiPSe Prolog. If you run any other version of Prolog on your home computer, it is your responsibility to make sure that your program will run on ECLiPSe Prolog (release 6 or any more recent release), as required. For example, you can run a command-line version of *eclipse* on moon remotely from your home computer to test your program (read handout about running *ECLiPSe Prolog*). To submit files electronically do the following. First, create a **zip** archive:

```
zip yourLoginName.zip lists.txt recursion.pl recursion.txt data.pl data.txt [bonusA2]
```

where `yourLoginName` is the Ryerson login name of the person who submits this assignment from a group. Oncy one student is allowed to sibmit an assignment from a group. Remember to mention at the beginning of each file *student, section numbers* and *names* of all people who participated in discussions (see the Course Management Form). You may be penalized for not doing so. Second, upload your ZIP file **yourLoginName.zip**
to D2L into "Assignment 2" folder. Make sure it includes **all** your files. Improperly submitted assignments will **not** be marked. In particular, you are **not** allowed to submit your assignment by email to a TA or to the instructor.

Revisions: If you would like to upload a revised copy of your assignment, then simply upload it again. (The same person must upload.) A new copy of your assignment will override the old copy. You can upload new versions as many times as you like and you do not need to inform anyone about this. Do not ask your team members to upload your assignment, because TA will be confused which version to mark: only one person from a group should submit different revisions of the assignment. The groups that submit more than one copy of their solutions will be penalized. The time stamp of the last file you upload will determine whether you have submitted your assignment in time.