cps721: Assignment 2 (100 points).
Due date: Electronic file - Thursday, October 10, 2019, 21:00 (sharp).

YOU SHOULD NOT USE ";" , "!" AND "–>" IN YOUR PROLOG RULES.
You cannot use any library predicates not mentioned in class or below.

You MUST work in groups of TWO, or THREE, you cannot work alone. You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructor. 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 provide 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.*

```
[ [] ]  and  [F | F]
[F, [a,G|V]]  and  [V, [F,d,a]]
[F|G]   and   [[q,p,r,s]]
[F, [] | [c | G]]  and  [a | [G, c]]
[P,a | [d,P|R]]   and   [[a | [b,c]]|[F,G | [I]]]
[F, [G], J | W]  and  [mth110, mth210 | [cps305, cps721]]
[F, G | [k, [G,l]]]   and   [k | [l | [F,J]]]
[a,b | F]  and  [a, G, d | [e, [G | [c]]]]
```

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

2 (25 points) Write the following Prolog programs. 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. See the handout "How to use Eclipse Prolog in labs" for details). Whenever possible, try to write a recursive program directly. You can**not** use programs that we did not discuss in class: you may lose all marks if you use external programs that are not allowed. 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. *add(N,InputList,OutputList)*: *OutputList* is an initial sequence of *N* elements from *InputList*. You can consider this as follows: we read *InputList* element by element, count its elements and add elements one by one to *OutputList* until we added exactly *N* elements or exhausted all elements from *InputList*, whatever happens first. Examples of queries:

```
The following all succeed              The following fail
add(4, [1,2,3], X).                        add(0,[q],[q]).
add(0, [a,b,c,d], X).                      add(1,[],[a,b,c]).
add(0,[],X).                               add(3,[a,b,z,c],[a,b,c]).
```

2. *convert(List1,List2)*: $List2$ is a list of elements of $List1$ converted element-by-element from one symbolic representation to another. You can assume in your program that $List1$ is given as an input, that $List1$ does not have lists nested inside (i.e., $List1$ is a "flat" list), and the 2nd argument is either a given list, or a variable.

Use in your implementation the predicate $dictionary(Elem1, Elem2)$. Assume that you are given a finite number of atomic statements with this predicate, e.g., $dictionary(one, 1)$ and $dictionary(two, 2)$ and so on, but for simplicity, there are no rules defining these predicate. You can write your own atomic statements, e.g., if you would like to translate from one language to another word by word. Examples of queries:

```
The following all succeed:                    The following fail:
convert([],[]).                               convert([two,four],[]).
convert([five], [5]).                         convert([two,six],[two,six]).
convert([one,two | [three,four]], X).            convert([four], 4).
convert([one,two | [three,four]], [1,2,3,4]).
```

3. *grep(E,L,Occurrences)*: *Occurrences* is the list of all occurrences of the element E in a given input list L. The list L may have nested lists inside. Also, the element E can be a list, and in this case the program has to find all occurrences of E as a sub-list inside L. The query `grep([],[a,b,c],X)` should fail, i.e., Prolog should say "No". Make sure your program returns only one answer to any query. *Hint:* you might wish to introduce a helping predicate $grep(E, L, Occurrences, Counter)$, where the last argument keeps track of the position of an element in L that the program ic currently processing. Examples of queries that succeed:

```
grep([a], [a,b,[c,a],[a],d,[a],[[a]],e],X).      /* query returns X=[4,6] */
grep(live,[learn,as,if,you,were,to,live,forever],A).  /* returns A=[7] */
grep([1,2],[a,b,[1,2,3],c,[1|[2]],d],[5]).
          The following to test: ?- grep(1,[1,2,3,4,5,6,3,2,1], X]).
```

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. (31 points) This part will exercise what you have learned about recursion over terms and over lists in Prolog. More specifically, this part is about doing transformations between lists and terms in Prolog. In this part of the 2nd assignment, you can use the library predicate $atomic(X)$ that is true if and only if X is either a constant, or a number, or the empty list. In all other cases, it is false.

- Implement the predicate $cons(List, Term)$ that constructs a term representing a given input $List$. Let the constant $null$ represent the empty list [], and the term $next(Head, Tail)$ represent the list $[Head|Tail]$. Your program takes as an input a given list, and constructs a term representation of this list (a kind of symbolic analog of a linked list). First, for simplicity, assume that $List$ has no nested lists inside, i.e., that $List$ is a "flat" list. The second argument in queries can be either a term, or a variable. Examples of queries:

```
The following all succeed:                     The following fail:
cons([],null).                    cons([a,b],next(a,next(b,next(null,null)))).
cons([a], next(a,null)).                          cons([a,b],null).
cons([a,b],next(a,next(b,null))).                 cons([a],[]).
?- cons([a,b,c],X).  /* query returns X=next(a,next(b,next(c,null)))
```

- Once you have completed the previous program, re-write it to make sure it works also for a more complex case, when an input list has other nested lists inside. Examples of queries:

```
The following all succeed:                                The following fail:
cons([[a]],next(next(a,null),null)).       cons([a,[b]],next(a,next(b,null))).
cons([[[a]]], next(next(next(a,null),null),null).   cons([[a]],next(a,null)).
?- cons([a,[b,[c]]],X).
/*query returns X=next(a, next(next(b, next(next(c, null), null)), null)) */
```

- Implement the predicate $linked2List(T, L)$ that takes as an input a term T representing a list and converts it back to the usual Prolog list L corresponding to T. Of course, for every correctly constructed term T there is only one corresponding list. Note that if you run the predicates $cons(L1, T)$ and $linked2list(T, L2)$ consecutively, then you get the list $L2$ identical to the initial list $L1$. In other words, the predicate $linked2List(T, L)$ is reciprocal to $cons(L, T)$, and they do mutually opposite transformations. Examples:

```
?- linked2List(next(a, null),[a]).

?- cons([a, [b, [c], d], e], X), linked2List(X, List).
X= next(a, next(next(b, next(next(c, null), next(d,null)) ), next(e, null)) )
List = [a, [b, [c], d], e]

?- cons([a, [b, [c], d]], X), linked2List(X, List).
X= next(a, next(next(b, next(next(c, null), next(d, null))), null))
List = [a, [b, [c], d]]

?- cons([a, [b, [c]]], Term), linked2List(Term, L), cons(L, T).
Term = next(a, next(next(b, next(next(c, null), null)), null))
L = [a, [b, [c]]]
T = next(a, next(next(b, next(next(c, null), null)), null))
```

Handing in solutions. An electronic copy of: (a) your program (**terms.pl**) that includes all your Prolog rules; (b) your session with Prolog, showing the queries you submitted and the answers returned (the name of the file must be **terms.txt**) . It is up to you to formulate a range of queries that demonstrates that your program is working properly. Your queries should include not only the examples given above, but a few other tests as well (e.g., test all bases cases). Request all answers (using either ";" or the button **more**).

4 (20 points). This part is asking you to write a recursive program over terms representing binary trees. Let the term $tree(X, Left, Right)$ represent a binary tree with the element X in the root, and two branches. The constant $void$ represents the empty tree. Recall that terms must be either arguments of predicates or arguments of equality.

- Implement the predicate $replace(X, Y, T1, T2)$ that computes the tree $T2$ by replacing all occurrences of X (if any) in the tree $T1$ with Y. You can assume that $T1$ is an input binary tree and that the arguments X and Y are given, but the argument $T2$ is either a variable (representing a binary tree that should be computed) or a given binary tree. Examples of queries that succeed:

```
?- replace(a,b, tree(a, tree(c,void,void), tree(a,void,void)), T).
            T = tree(b, tree(c,void,void), tree(b,void,void))
?- replace(7,9, tree(4, tree(2,tree(1,void,void),tree(3,void,void)),
                        tree(6,tree(5,void,void),tree(7,void,void))), X).
X = tree(4, tree(2,tree(1,void,void),tree(3,void,void)),
            tree(6,tree(5,void,void),tree(9,void,void)))
```

- Implement the predicate $expand(X, Tree1, Tree2)$ that holds if $Tree2$ is an ordered tree resulting from inserting number X into the ordered tree $Tree1$. If X already occurs in $Tree1$, then $Tree1$ and $Tree2$ are identical. You can assume that the number X and $Tree1$ are given as an input and that $Tree1$ is indeed a binary sorted tree (there is no need to check this). Recall that in a binary sorted tree $tree(Root, Left, Right)$ the element $Root$ is greater than or equal than all elements in the $Left$ branch, and also $Root$ is less than or equal than all elements in the $Right$ branch. Recursively, same condition applies to all other non-leaf nodes in the sorted binary tree. If you need more details, read *http://en.wikipedia.org/wiki/Binary_search_tree*

```
?- expand(2, tree(4,void,void),T2), expand(1,T2,T1), expand(3,T1,T3),
      expand(6,T3,T6), expand(5,T6,T5), expand(7,T5,T7).
T2= tree(4, tree(2,void,void), void)
T1= tree(4, tree(2, tree(1,void,void), void), void)
T3= tree(4, tree(2, tree(1,void,void), tree(3,void,void)), void)
T6= tree(4, tree(2, tree(1,void,void), tree(3,void,void)), tree(6,void,void))
T5= tree(4, tree(2, tree(1,void,void), tree(3,void,void)),
                                tree(6, tree(5,void,void), void))
T7= tree(4, tree(2, tree(1,void,void), tree(3,void,void)),
                                tree(6, tree(5,void,void), tree(7,void,void)))
```

Handing in solutions: (a) An electronic copy of your file **tree.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 **tree.txt**). It is up to you to formulate a range of queries that demonstrates that your programs are working properly.

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 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 `recursion.pl` or `term.pl` or `tree.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 on `moon`:

```
zip yourLoginName.zip lists.txt recursion.pl recursion.txt
                                     terms.pl terms.txt tree.pl tree.txt
```

where `yourLoginName` is the Ryerson login name of the person who submits this assignment from a group. Remember to mention at the beginning of each file *student, section numbers* and *last 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**
(make sure it includes **all** files) to D2L into "Assignment 2" folder.

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 submit a revised copy of your assignment, then run simply the submit command again. (The same person must run the submit command.) A new copy of your assignment will override the old copy. You can submit new versions as many times as you like and you do not need to inform anyone about this. Don't ask your team members to submit 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 time stamp of the last file you submit will determine whether you have submitted your assignment in time.