cps721: Assignment 2 (100 points).
Due date: Electronic file - Friday, October 12, 2018, 11:00pm (sharp).
YOU SHOULD NOT USE ";" , "!" AND "–>" IN YOUR PROLOG RULES.
You must work in groups of TWO, or THREE, you can**not** work alone.
Contact the CSSU or post a message on the CSSU Facebook page to find CPS721 partners.

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 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 in a different representation. You lose marks if you give only short answers, but do not explain.

```
[P]  and  [[desk,chair | [lamp|[]]] | Q]
[eng | [Q | [R | S]]]  and  [R | [Q | AI721]]
[[R | S], [R] | S] and [[P | Q], [Q], x, y]
[Var | [[d], d]]  and  [const, X | X]
[P, Q, [a, Q]]  and  [a, b, [R | [R]]]
[Z, d | Z]  and  [[] | [d, []]]
[j, [k, l, m], l, m]  and  [j, [k | LIST], LIST]
[ryerson, P, Q | R]  and [Q | [[cs,Q] | [ryerson | P]]]
```

Handing in solutions: (a) An electronic copy of your file **lists.txt** must be included in your **zip** archive. No printout is required.

2 (20 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). 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. *everyOther(List1,List2)*: $List2$ is a list consisting of every other element of $List1$. You can assume in your program that $List1$ is given, and the 2nd argument is either a given list, or a variable. Examples of queries:

```
The following all succeed:                    The following fail:
everyOther([],[]).                          everyOther([a,b,c],[]).
everyOther([a],[a]).                        everyOther([a,b,c],[a,b]).
everyOther([a,b,c,d,e,f],[a,c,e]).
everyOther([a,b,c],[a,c]).
          The following to test:   ?- everyOther([1,2,3,4,5,6,7,8], X).
```

2. *removeDups(List1,List2)*: *List2* is the result of removing all duplicate elements from the given *List1*, where the duplicate elements removed are those that occur earlier in List1. Examples of queries that succeed:

```
removeDups([a,b,c],[a,b,c]).
removeDups([a,b,a],[b,a]).      /* the first "a" was removed */
removeDups([a,b,c,b,d,b,c,a],[d,b,c,a]). /*the initial  a,b,c,d are removed*/
          The following to test: ?- removeDups([1,2,3,4,5,6,3,2,1], X]).
```

3. *sameFirstLast(List)*: *List* is a non-empty list such that the first and the last elements are the same. You can assume in your program that a given *List* is not empty. Examples of queries:

```
The following all succeed                          The following fail
sameFirstLast([[a],z,b,c,d,[a]]).              sameFirstLast([a,b,[a,b]]).
sameFirstLast([21 | [22,[k | [l,m]],21]]).     sameFirstLast([a,[c,[d]],b]).
          The following to test:  ?- sameFirstLast([1,X1,X2,X3,X4,1]).
```

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**). Include it into your **zip** archive. It is up to you to formulate a range of queries that demonstrates that your programs are working properly. No printouts are required.

3. (34 points) This part will exercise what you have learned about recursion over terms in Prolog. Let term *next(Head,Tail)* represents a Prolog list $[Head \mid Tail]$. For example, *next(7, next(1, next(5, next(0, next(9,nil)))))* represents the list *[7,1,5,0,9]*. We use the constant *nil* to represent the empty list *[ ]*. Write the Prolog programs implementing the following predicates.

1. *appendT(Term1,Term2,Result)*: this predicate is similar to the predicate *append(List1,List2,Result)* that we discussed in class, but it works with terms representing lists. Examples of queries that succeed:

```
?- appendT(next(a, next(b, nil)), next(1, next(2, nil)), Result).
        Result = next(a, next(b, next(1, next(2, nil))))
?- appendT(Init, next(1,next(2,nil)), next(a,next(b,next(1,next(2,nil))))).
        Init = next(a, next(b, nil))
?- appendT(next(a,next(b,nil)), Final, next(a,next(b,next(1,next(2,nil))))).
        Final = next(1, next(2, nil))
```

2. Implement the predicate *list2term(List,Term)*: that takes as its input a usual Prolog List, and produces as its output a Term representing this list. You can use the library predicates $var(X)$ and $atom(X)$; note that both $atom([nil])$ and $atom([\,])$ are true. Examples of queries that succeed:

```
?- list2term([[a], [b, [c]] | [d] ], X).
     X= next(next(a,nil), next(next(b,next(next(c,nil),nil)), next(d,nil)))
?- list2term([a | [b | [c]] ], X).
     X= next(a, next(b, next(c, nil))).
?- list2term([ [a,[b, [c]]], d], X).
     X= next(next(a, next(next(b, next(next(c,nil), nil)), nil)), next(d,nil))
?- list2term([[a, b]], X).
     X= next(next(a, next(b, nil)), nil).
```

3. *flat(Term,FlatTerm)*: *Term* is a given term representing list of lists (they can also contain nested lists inside), and *FlatTerm* is a *Term* 'flattened' so that the elements of *Term*'s sublists (or sub-sublists) are reorganized into a term representing one plain list (that has no sublists), but the sequence of elements remains the same. You can assume in your program that there are no occurrences of empty lists in *Term*, but *Term* can be nil (i.e., the empty list). The order of elements occuring in *Term* remain the same in *FlatTerm*. There should be no terms inside *FlatTerm* that would represent nested lists. Examples of queries that succeed:

```
?- list2term([[a, b]], X), flat(X, Y).
        X = next(next(a, next(b, nil)), nil)    Y = next(a, next(b, nil))
?- list2term([[[a] | [z]],[[b,[c]],[d]]], Term), flat(Term,FlatTerm)
  Term= next(next(next(a,nil), next(z,nil)),
        next(next(next(b, next(next(c,nil),nil)), next(next(d,nil),nil)),nil))
  FlatTerm = next(a, next(z, next(b, next(c, next(d, nil)))))
?- list2term([[k],[[l]],[m | [n]]], Term), flat(Term, FlatTerm).
  Term= next(next(k,nil), next(next(next(l,nil),nil),
                next(next(m, next(n,nil)),nil)))
  FlatTerm = next(k, next(l, next(m, next(n,nil))))
```

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. Request all answers (using either ";" or the button **more**).

4 (22 points). This part of the assignment is asking you to write a recursive program over terms representing binary trees. According to Wikipedia, "In computer science, an AVL tree is a self-balancing binary search tree, and it was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one." Your task is to implement the predicate *avl(Tree)* that takes as an argument a given binary tree *Tree* and verifies for every node in the tree whether the heights of the two children subtrees differ by at most one. Note that you do not need to verify whether *Tree* is actually a binary *search* tree. Your program has to check only whether restriction on heights is true.

Handing in solutions: (a) An electronic copy of your file **avl.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 **avl.txt**). Include it into your **zip** archive. It is up to you to formulate a range of queries that demonstrates that your programs are working properly. No printouts are required.

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked from `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 `terms.pl` or `avl.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*). First, to submit files electronically create a **zip** archive:

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

where `yourLoginName` is the 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 *names* of all people who participated in discussions (see the course management form). You may be penalized for not doing so. Second, upload to D2L into "Assignment 2" folder all individual files and your ZIP file **yourLoginName.zip**

Revisions: If you would like to submit a revised copy of your assignment, then upload your files again. The same contact person must upload the files from a group. 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 upload your files, 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 on time.