cps721: Assignment 2 (100 points).
Due date: Electronic file - Tuesday, October 13, 2015, 21:00 (sharp).

You have to work in groups of TWO, or THREE. You should not work alone.
YOU SHOULD NOT USE "**;**" (disjunction) "**!**" (cut) AND "−>" IN YOUR PROLOG PROGRAMS.

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

```
[mth110 | [mth210 | [cps305 | [cps721 | CPS822]] ] ]  and  [P,R,Q | AI]
[List | [List, a,b,c]]  and  [[], Z | U]
[U,[W|U],a,b]  and  [Z | [ [a|[b] ] | W ] ]
[Var | U]  and  [http, www | [ryerson, ca]]
[k,p | U]  and  [k, W, m, n, [W | [c]]]
[U | W]  and  [W | [[] | U]]
[myself, Q | [P | U]]  and  [W | [[vars| [P]] | [myself | Q]]]
[k, [Q | [l,m]], tree | [book | Z]]  and  [P, U | [R,Q,n]]
```

Handing in solutions: (a) An electronic copy of your file **lists.txt** must be included in your **zip** archive. Make sure you provide detailed explanations for each pair of lists.

2 (46 points). In this part of the assignment you are asked to implement in Prolog a few programs with recursion over lists. Keep all these programs in one file named **recursion.pl** 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. In most cases, try to write recursive programs yourself without using predicates from class. This will help you to learn about writing recursive programs. Use the case-analysis technique explained in class. If you introduce any new "helping" predicates, you have to implement them as well. Test each of your programs on some examples of your choice (including queries with variables when possible): save a copy of all tests in your file **recursion.txt**

1. *nth(N,List,Element)*: the N^{th} element of *List* is *Element*. You can assume that in any query the first argument is a positive integer, the second argument is a possibly empty list (input to your program), and the third argument is either a variable representing the result that has to be computed, or a constant (the N^{th} element of the input list) representing the result to be verified. Examples of queries:

```
The following all succeed
?- nth(1,[a],a).
?- nth(3,[a,b,c],c).
?- nth(2,[a,b,c,d,e,f],b).
```

```
The following all fail
?- nth(0,[],X).
?- nth(1,[a,b,c],b).
?- nth(4,[p,q,r],Z).
```

2. The predicate *remove(X,InputList,OutputList)* holds when *OutputList* is the result of removing each occurrence of an element X from a given (possibly empty) list *InputList*. In queries, *OutputList* can be either a variable representing a list that should be computed, or a list that should be verified.

```
Examples of queries that succeed:
?- remove(a,[c,a,n,a,d,a],[c,n,d]).
?- remove(a,[],[]).
?- remove(a,[[d,e,f],[a,a]],[[d,e,f],[a,a]]).
?- remove([],[[]],X).  returns X=[]
```

```
  Examples of queries that fail:
?- remove(a,[a,b,c,d,a,e],[b,c,d,a,e]).
 ?- remove(a,[],[a]).
 ?- remove(a,[d,e,f,g,h],[d,e,f]).
```

3. *splitEvenOdd(InList,List1,List2)*: *InList* is a given unsorted list of numbers (input to your program), and *List1, List2* are lists of some elements from *InList* such that *List1* is the list of all (if any) even numbers from *InList*, and *List2* is a list containing all odd numbers of *InList*, if any. Both *List1* and List2 should be sequences of elements in same order as in *InList*. Note that *InList* can be empty, and elements there can be repeated. There is no need to check your input for correctness.

```
Examples of queries that succeed:
?- splitEvenOdd([22,9,11,400,0],X,Y).
    returns  X=[22,400,0]  and  Y=[9,11]
?- splitEvenOdd([33],[],[33]).
?- splitEvenOdd([7,8,8,9,7],L1,L2).
    returns  L1=[8,8] and L2=[7,9,7]
```

```
Examples of queries that fail:
?- splitEvenOdd([],[H|T],X).

?- splitEvenOdd([1,2,3,4],[2,4],[3,1]).

?- splitEvenOdd([1,2],[2],[]).
```

4. The predicate *lessThanEq(List1,List2,List3)* is true if and only if the total number of elements in *List1* and *List2* is less than or equal to the number of elements in *List3*. Write a recursive program that implements this predicate. You can assume that in any query, the first, the second and the third arguments will be (possibly empty) given lists (input to your program). Examples of queries:

```
The following all succeed
?- lessThanEq([],[a,b,c],[k,l,m,n]).
?- lessThanEq([a],[[b],[c]],[k,l,m]).
```

```
The following all fail
?- lessThanEq([p,q,r],[a,b,c],[k,l,m,n]).
?- lessThanEq([H|T],[],[]).
```

5. Pascal Triangle is a set of numbers arranged as a triangle: start with "1" at the top, then continue placing numbers below it in a triangular pattern. Each number is sum of the two numbers directly above it. You can find examples and illustrations at the Web page `www.mathsisfun.com/pascals-triangle.html` The predicate *nextRow(Input,Row)* is true if *Input* is a list of numbers representing a row in Pascal Triangle, and *Row* is the next row just below the given input row. Write a recursive program implementing this predicate. The predicate *row(N,List)* is true if *List* is a list of numbers representing N-th row in Pascal Triangle. Write a recursive program implementing directly this predicate. Use predicate *nextRow(Input,Row)* in your implementation. Note you cannot use a formula for calculating entries in the triangle or introduce any other predicates. Your program must be implemented as specified. Examples of output from your program:

```
?- row(7, R).
  R = [1,6,15,20,15,6,1]
?- row(8, R).
  R = [1,7,21,35,35,21,7,1]
```

```
?- row(10, R).
    R = [1,9,36,84,126,126,84,36,9,1]
?- row(9, R).
    R = [1,8,28,56,70,56,28,8,1]
```

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. (30 points).

This part will exercise what you have learned about recursion over terms in Prolog. This part consists of two sub-parts. First, you have to work with lists represented as terms. Second, you have to work with terms representing binary trees.

(a) The predicate *qs(Input,Output)* is true if *Output* is a sorted list of numbers from a given list *Input*. 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 *[ ]*. You have to implement the well-known quick-sort algorithm in Prolog using this term-based notation for lists. Your program should work as follows. Given an input list of numbers, the program should take the head X of this input list *next(X,Others)* and use it as a pivot. This should be implemented using a helping predicate *part(X, Others, Ls, Bs)* which partitions numbers from *Others* into two sub-lists: *Ls* which include only elements less than the pivot X, and *Bs* which include only the numbers bigger than or equal to X. Once this partitioning has been completed, your program must recursively sort the lists *Ls, Bs*, and finally, it must merge the sorted halves together with the pivot into an ordered output list. The latter merge operation should be implemented using a helping predicate *mergeLists(LT1,LT2,Result)* that takes as input two lists *LT1,LT2* represented using terms, as explained above, and combines them into a sorted list *Result* that also has a term-based representation. The following are examples of how your Prolog program should work.

```
?- qs(nil, Sorted).
Sorted = nil
?- qs(next(8, next(5, nil)), Out).
Out = next(5, next(8, nil))
?- qs(next(7, next(1, next(10, next(5, next(0, next(12, next(9, next(2,
              next(11, next(4, next(6, next(8, next(3, nil))))))))))))), X).
X = next(0, next(1, next(2, next(3, next(4, next(5, next(6, next(7, next(8,
              next(9, next(10, next(11, next(12, nil))))))))))))
```

(b) Let the term *tree(X,Left,Mid,Right)* represent a ternary tree with the element X in the root, and three branches *Left, Mid, Right*. The constant $void$ represents the empty tree. In other words, a tree is ternary if nodes have at most 3 children. Recall that terms must be either arguments of predicates or arguments of equality since each term represents a structured object. Implement the predicate $subsFirst(X, Y, T1, T2)$ that computes the output tree $T2$ by replacing the very first occurrence of X (if any) in a tree $T1$ with Y. You can assume that $T1$ is a given input ternary tree and that the arguments X and Y are also given, but the argument $T2$ is either a variable (representing a ternary tree that should be computed) or a given ternary tree. Since a node in a ternary tree has at most 3 children, we have to determine in which order we search them to find the first occurrence of X. Your program has to search first all nodes in *Left* to find the first occurrence of X, but if X is not in *Left*, then your program should try to locate X in the middle branch *Mid*, but X does not occur there, then it must try to locate X in *Right*. In other words, the occurrences of elements are ordered top-down, and within same level left-most occurrences are before occurrences in the middle branch, which are ordered before right-most occurrences. Use in your program a helping predicate *memberTree(X,T)* that is true if element X occurs in one of the nodes of the ternary tree T. *Hint:* this predicate can be implemented similarly to a binary tree example that we have considered in class. A sample ternary tree and examples of queries that succeed can be downloaded together with this assignment. Write your Prolog program in the file **subsFirst.pl**

Handing in solutions. An electronic copy of: (a) your programs (**qs.pl** and **subsFirst.pl**) that include 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 base cases). Request all answers (using either ";" or the button **more**).

4. *Bonus work* (**up to 50 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.* If your assignment is submitted from a group, 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.

This problem tests your knowledge about writing recursive programs that work with Prolog terms (structures).
First order logic (FOL) is considered as one of the candidate languages for knowledge representation. This is because FOL has an expressive, well-known syntax and precise semantics. This part of the assignment is concerned with syntax of FOL only. In this assignment, we consider a language of FOL that consists of the following disjoint sets of distinct primitive symbols:

- FOL Variables: $x, y, z, p, f, m, \ldots$ with or without subscripts; the predicate $fol\_var(X)$ holds if X is one of the symbols declared as a variable.
- FOL Constants: $c, i, b, \ldots$ with or without subscripts; the predicate $fol\_const(X)$ holds if X is one of the symbols declared as a constant.
- FOL Relation symbols: $human$, $owns$, ..., where each relation symbol has a positive number of arguments (called *arity*); the predicate $fol\_rel(P, N)$ holds if P is one of the N-place relation symbols.
- FOL Function symbols: $mother, plus, \ldots$, where each function symbol has a positive number of arguments (called *arity*); the predicate $fol\_func(F, N)$ holds if F is one of the N-argument function symbols.
- FOL Connectives: $\neg$ (neg), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (iff, i.e., if and only if), eqv (equality); here and subsequently, we use the symbol eqv to avoid conflict with $=$ in Prolog.

- FOL Quantifiers: $\forall$ (forall), $\exists$ (exists).

This syntax of FOL allows us to represent complex English assertions. For example, the English sentence "all humans are mortal" can be written in FOL as $\forall x(human(x) \rightarrow mortal(x))$. In a similar vein, the well-known logical argument "All humans are mortal. I am a human. Consequently, I am mortal" can be written in FOL using constant i as follows:

$$\big(\forall x(human(x) \rightarrow mortal(x)) \wedge human(i)\big) \rightarrow mortal(i).$$

The English sentence "everyone has a mother" can be written in FOL as $\forall p \exists m(mother(p)\ eqv\ m)$, and a sentence "everyone has one and the only one mother" can be written in FOL as

$$\forall p \exists m\big((mother(p)\ eqv\ m) \wedge \forall x(\neg(x\ eqv\ m) \rightarrow \neg(x\ eqv\ mother(p)))\big).$$

One of the main strength of FOL comes with its ability to formulate precisely different readings of ambiguous English sentences. To represent the sentence "Everyone who owns a donkey beats it", we can use 2-place relations $own(p, d)$ (a person p owns a donkey d) and 2-place relation $beat(p, d)$ (a person p beats a donkey d). Then, one possible reading of this sentence is $\exists d \forall p(owns(p, d) \rightarrow beats(p, d))$: "there is donkey (a single poor creature) such that every peasant who owns this donkey beats it". A different possible reading is $\forall p \exists d(owns(p, d) \wedge beats(p, d))$: "every peasant has his donkey such that the peasant owns his donkey and beats it". Moreover, FOL was instrumental to clarify the foundations of mathematics, a discipline where the very basic concept of set turned out to be ambiguous at the end of XIX century. This might be hinted with a Barber Puzzle, an overly simplified, but popular version of Bertrand Russell's paradox about sets. Russell states it as follows: "You can define the barber as *one who shaves all those, and those only, who do not shave themselves.* The question is, does the barber shave himself?" Using 1-place related $barber$ and 2-place relation $shaves$, this puzzle can be written as $\exists x(barber(x) \leftrightarrow \forall p(shaves(x, p) \leftrightarrow \neg shaves(p, p)$. It turns out that using the well-defined semantics of FOL, one can prove that negation of this formula holds no matter how we interpret the relations in the puzzle. As far as CPS721 is concerned, FOL might be a language of choice to represent a vast Web of beliefs in a knowledge base. For example, in the beginning of CPS721, we considered as an example a sentence "Someone is a bachelor if and only if there has never been a wedding where that person is the groom". Using 1-place relation $bachelor$, another 1-place relation $wedding$, and assuming that wedding is an event that has an associated participant who is represented using 1-place function $groom$, we can represent this sentence in FOL as $\forall x\big(bachelor(x) \leftrightarrow \neg\exists y\big(\ wedding(y) \wedge (x\ eqv\ groom(y))\ \big)\ \big)$.

In this assignment, we are interested only in verifying simple properties of FOL formulas and FOL terms. Roughly speaking, simple FOL terms are the noun phrases of first-order languages: constants can be thought of as first-order counterparts of proper names (bob, mary), and variables as first-order counterparts of pronouns (his, its, my) or nouns. We can then combine our 'noun phrases' with our various 'relation symbols' to form what we call an atomic FOL formula. Intuitively, an atomic formula is the first-order counterpart of a natural language sentence consisting of a single clause (that is, a simple sentence). Subsequently, we represent both FOL formulas and FOL terms as structures in Prolog (i.e., as Prolog terms). Let's proceed with precise definitions that you can use when you will be writing your program.

A *well-formed FOL term* is:

- either FOL constant, or
- FOL variable, or
- if f is N-place function symbol ($N \geq 1$), and $t_1, \ldots, t_N$ are well-formed terms, then $f(t_1, \ldots, t_N)$ is also a well-formed term. (Note that in the last sentence we need exactly N terms to occupy N argument positions associated with f).

Nothing else can be a *well-formed term.* For example, using 2-place function symbols sum and $prod$ to represent, respectively, $+$ and $\cdot$, and using FOL variable symbols x, y, z, we can compose terms $sum(y, z)$, $prod(x, sum(y, z))$. Using these compound terms, we can write an atomic FOL formula $\forall x \forall y \forall z(prod(x, sum(y, z))\ eqv\ sum(prod(x, y), prod(x, z)))$. (Using a more familiar notation, this can be written as $x \cdot (y + z) = x \cdot y + x \cdot z$.)

A *well-formed FOL formulas* are defined inductively similar to terms:

- Let r be N-place relation symbol ($N \geq 1$), and $t_1, \ldots, t_N$ are terms, then $r(t_1, \ldots, t_N)$ is an (atomic) well-formed formula. (Note that in the last sentence we need exactly N terms to occupy N argument positions associated with r).
- Let t_1, t_2 be well-formed FOL terms, then the equality between these terms $(t_1\ eqv\ t_2)$ is also a well-formed formula.
- If α, β are well-formed formulas, then so are $(\neg\alpha)$, $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$.

- If v is a FOL variable and α is a well-formed formula, then $(\exists v)(\alpha)$ and $(\forall v)(\alpha)$ are also well-formed formulas.

Nothing else can be a *well-formed formula*. In formulas $(\exists v)(\alpha)$ and $(\forall v)(\alpha)$, α is called the *scope* of the quantifier Notice that it might happen that α does not contain the variable v. An occurrence of a FOL variable v is said to be *bound* in a well-formed formula α if either it is the occurrence of v in a quantifier '$(\forall v)$' (in a quantifier '$(\exists v)$', respectively), or it lies within the scope of a quantifier '$(\forall v)$' inside α (of a quantifier '$(\exists v)$' inside α, respectively). Otherwise, a FOL variable that is an argument of one of the relation or one of the function symbols in α is said to be *free* in α. All formulas mentioned as examples above are well-formed FOL formulas, except that sometimes we omitted parentheses to improve readability.

In this part of the assignment, you have to write a few Prolog programs that will check whether a given language contains ambiguous symbols, whether FOL terms and FOL formulas are well formed, and finally, determine whether a given FOL formula has free variables or not.

(a). Design a simple KB using Prolog predicates $fol\_const(C)$, $fol\_var(X)$, $fol\_func(F, N)$, $fol\_rel(R, N)$. Include in your KB, all symbols that you need to represent your own testing formulas. For example, to represent some of the symbols in the examples above, we can write

```
fol_var(h).  fol_var(x).  fol_var(y).  fol_var(z).          fol_const(i).
fol_rel(human,1).     fol_rel(mortal,1).   fol_rel(owns,2).
fol_func(mother,1).   fol_func(prod,2).    fol_func(sum,2).
```

(b). The Prolog predicate $ambiguous(X)$ is true if X is one of the symbols in the FOL that has two different declarations. This predicate is useful to check whether a language in KB is defined properly. For example, if a 2-place symbol is defined both as function and as relation, then we cannot read uniquely formulas with this symbol. However, if same symbol is used for different purposes with different number of arguments, then the number of arguments helps to disambiguate this symbol. Write a complete set of rules that define the predicate $ambiguous(X)$.

(c). The Prolog predicate $wft(T)$ is true if T is a FOL well-formed term. Similarly, the Prolog predicate $wff(F)$ is true if F is a FOL well-formed formula. Write a complete set of Prolog rules to define these two predicates. Use the precise definitions given above. Note that both Prolog predicates take as arguments Prolog structures (terms), see examples in sub-part (d) below. In particular, the Prolog term `exists(d, forall(p, owns(p,d)))` represents FOL formula $(\exists d)((\forall p)(owns(p, d)))$. To represent connectives, include the following declarations of the new operators on the top of your program:

```
:- op(250,yfx, 'eqv').       /* the highest priority operator */
:- op(300,fx,'neg').
:- op(400,yfx,'and').
:- op(500,yfx,'or').
:- op(600,xfx,'implies').
:- op(650,xfx,'iff').       /* the lowest priority operator */
```

The first argument in *op* above is designed to give different priorities to logical connectives according to the well-known conventions: negation *neg* is stronger than conjunction *and*, that is in turn stronger than disjunction *or*, and so on. Thanks to these priorities, the formula $neg\ a\ and\ b\ implies\ c\ or\ d$ has the unique reading $((neg\ a)\ and\ b)\ implies\ (c\ or\ d)$. In your implementation, you can use the system predicate =.. that is being applied to $LHS\ =..\ [Name|Arguments]$ converts a Prolog structure on left hand side into a list. For example,

```
?-  this(is,a,complex,structure(with,several,arguments)) =.. [Name | Arguments].
Name = this
Arguments = [is, a, complex, structure(with,several,arguments)]
```

This system predicate =.. is convenient to parse complex terms and atomic formulas into constituents. (Note that there is no space between = and .. in =..). In your implementation you can also use the system predicate $compound(Expr)$ that is true if $Expr$ is a compound Prolog structure (term). *You are* **not** *allowed to use any other system predicates*. In other words, your rules have to be written using only Prolog predicates from KB, predicates $wft(T)$, $wff(F)$, $compound(Expr)$, =.., or any of the predicates that we discussed in class. Save your Prolog rules in the file **logic.pl**

(d). Test your program on at least 10 different queries of your own choice. Your queries should include formulas different from those mentioned above in this assignment. Remember to include in KB declarations for all symbols that occur in your formulas. For example, for the two formulas mentioned in the beginning, the queries are

```
?- wff( (forall(h, human(h) implies mortal(h)) and human(i)) implies mortal(i)).
?- wff(forall(x, forall(y, forall(z, prod(x, sum(y,z)) eqv sum(prod(x,y),prod(x,z)))))).
```

Save your session with Prolog (queries submitted, and answers returned) in the file **logic.txt**

(e). The Prolog predicate $closed(F)$ is true if F is a well-formed formula and does not exists a FOL variable in KB such that it has at least one free occurrence in F. Write the rule implementing $closed(F)$ with the help of an auxiliary predicate $free\_occurs(X, F)$. The helping predicate $free\_occurs(X, F)$ is true if X is one of the FOL variables that has a free occurrence in F. You have to write a complete set of rules defining the predicate $free\_occurs(X, F)$. *You cannot use any of the system predicates except of those mentioned above.* Keep all these rules in the same file **logic.pl** Once you have completed your program, test it using same formulas or different formulas to show that you get answers "yes", and "no" depending on whether a formula F is closed or not. Keep all results of your test in the file **logic.txt**

Handing in solutions: (a) An electronic copy of the file **logic.pl** with your Prolog program must be included in your **zip** archive; (b) Test your program using your own examples. It is up to you to formulate a range of queries that demonstrates that your program is working properly. Include your session with Prolog in the file **logic.txt** (c) Hand in printouts of **logic.pl** and **logic.txt** in class.

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 to load `recursion.pl` or `qs.pl` or any other Prolog file 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 qs.pl subsFirst.pl terms.txt
[logic.pl]
```

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, run the command

```
submit-cps721 yourLoginName.zip
```

on one of the `moons`. If you need more information about **zip**, read manual pages on *moon*. Make sure you have **ssh** software on your home computer. You might need it to establish a remote connection with departmental servers, if you are going to submit your assignment from home. Otherwise, submit your assignment from labs at any time. 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. Make sure that your Computer Science account at Ryerson is properly configured and you can easily access it from Labs (or from home, if you wish). Talk to one of the system administrators in ENG246 or ENG248 *in advance* if you experience any difficulties.

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. 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 your last ZIP file determines whether you have submitted your assignment on time.