**CSE 6341; LISP Interpreter Project, Part 1; Autumn 2018**
**Due: 11:59 pm, Oct. 17**

This is the first part of the interpreter project and will consist of the front-end of the interpreter. This will read in an input s-expression in either dot or list notation (or both), convert it into the appropriate internal representation, and ouput the s-expression in the *dot notation*. Then it will read the next s-expression and go through the same process until it does this for all the s-expressions in the input stream.

Each s-expression in the input stream may be over one or more lines. The white-space characters that will be used are the blank character, tabs, and carriage return/newline. Each complete s-expression will be followed by a line containing a single "$" sign. The last s-expression will be followed by a line containing "$$". An identifier will consist of a letter of the alphabet followed by one or more letters or digits; *only* uppercase letters will be used. An integer may be signed or unsigned; thus, `42`, `-42`, `+42`, are all valid. Integers will have no more than 6 digits; and symbolic atom names will have no more than 10 characters. The only characters that will appear on the input stream are uppercase letters, digits, white-space, and $; no operator symbols will be used.

So your program should read in the first s-expression, convert it into the internal representation, and output the s-expression in dot notation. The input s-expression may be ill-formed, e.g., may contain two dots in a row or may contain parentheses where they shouldn't appear, etc.; or you may find a character, such as "#", that should not appear in the input stream at all; or you may find a line with a "$" prematurely, i.e., before the s-expression is complete; etc. In all such cases, you should print an appropriate error message, skip the remaining lines until the next line that contains "$" and then continue to read the next s-expression. Once you have output, in dot notation, one s-expression, you should go past the line containing the `$` sign, read the next s-expression and continue in the same manner until you read a line containing `$$` which indicates the end of the input.

**Important**: This is an *individual* project. Discussions with other students in the class is permitted and should, preferably, take place on the Piazza site. But the code you submit must be your own. If you borrow anything from any source, you must document it carefully in your submission. If you don't, you will be considered guilty of academic misconduct and that can have very serious consequences.

**Grade**: This part of the project is worth 50 points; the second part will also be worth 50 points.

**Goal**: The goal of the overall project is to implement an interpreter for pure LISP [more or less]. You may use any of the following languages: *C++, C, Java*. Do NOT use *Scheme* or *LISP* or *Haskell*. Languages like *Perl* are not recommended. If you want to use some other language, talk to me first to make sure it is acceptable. Do not use *lex* or *yacc* or other similar tools.

**Approach**: The most sensible way to do this project is to define an `SExp` class that corresponds to both atomic and non-atomic s-expressions. Writing in pseudo-C++, you may have something like:

```
class SExp{
  int type; /* 1: integer atom; 2: symbolic atom; 3: non-atom */
  int val;  /* if type is 1 */
  string name; /* if type is 2 */
  SExp* left; SExp* right; /* if type is 3 */
  ...
```

So, on input, if you read in an integer, you would create an instance of `SExp` of type 1 with appropriate value. If you read in a symbolic name, you would have to search through existing symbolic atoms (how?) and either return the appropriate atom if it exists; or, if there is no such atom, create one of type 2 with the particular name, and return that. If, on input, you have a non-atomic s-expression, you call the input function to handle the left and right branches and return an s-expression whose `left` and `right` point to the two branches. What I have described should be fairly straightforward. What is going to be more challenging is dealing with the list notation. Think about how you would handle that and we will discuss it on Piazza.

For the second part of the lab, we will specify the complete list of primitive functions that you have to implement (most likely as part of the `SExp` class since they will require access to the internal representations). For now, atoms such as `CAR` have no special meaning; the only exception is `NIL` which, of course, is essential for understanding the list notation and converting to the dot notation.

**Important Requirement**: Your lab must compile and run on the department's `stdlinux`. Do NOT submit something that runs only on Visual Studio etc.
**Repeat:** Your lab must compile and run on the department's `stdlinux`. Do NOT wait until five minutes before the submission deadline to check whether your lab runs on `stdlinux`. If it doesn't, there will be no special consideration of how you have worked really hard to get it complete, etc.! If it doesn't compile and run `stdlinux`, by definition, you have not completed the lab!

**What To Submit And When**: On or before 11:59 pm, Mar. 1, you should submit the following files. Note that you should *not* submit object files. One should be a file called READMEP1 that contains clear instructions on how to use (this part of) the interpreter, in other words how to compile it and how to run it. Any unusual things about the expected input etc. [such as "don't put two periods in a row, else the machine will crash"]. The second file, called 'designP1.txt', should be a design-and-documentation file; this should be a plain text file describing your interpreter, anything unusual in its design, or in the implementation; if you borrowed ideas or anything else from anywhere, that should be documented in this file. The third file should be a Makefile. The remaining one or more files should be your source file(s). The grader will look at all the files, compile and run your interpreter as per the instructions in your READMEP1 file, and then assign the grade.

**How To Submit**: On or before 11:59 pm, Oct. 17, you should submit your project on Carmen/Canvas. I will post the details later.
**Repeat:** DO NOT submit object code, .doc files, etc.