CSE 6341; LISP Interpreter Project, Part 2; Spring 2018
Due: Friday, March 23

This is the second part of the interpreter project and will consist mainly of the Lisp expression *evaluator*, i.e., `eval[]` and associated functions. This part of the project is worth 50 points. Use the same programming language for this part as you did for part 1 since you have to use, in this part, the functions you implemented in part 1. As was the case for part 1 of the project, 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 which can have serious consequences.

In this part of the project, your `main()` function will use the `input()` function you implemented in part 1 of the project to read in the next s-expression from the standard input stream. The s-expression will be expected to be a Lisp expression, unlike in part 1 where it could be any legal s-expression; but it will be the job of eval[] and associated functions to check this, not that of input() or main(). Next, the `main()` function will use your `output()` function from part 1 to output the just read s-expression. The `main()` function will then call the `eval[]` function, passing it the just-read s-expression as the Lisp expression to be evaluated, and `NIL` as a-list. When control returns, `eval[]` will return the resulting value which will, of course, be another s-expression. The `main()` function will then use the `output()` function to output this resulting s-expression. Once this is complete, the `main()` function should loop back to the start and use `input()` to read the next input s-expression; etc. If the next line contains "`$$`", signifying the end of the input stream, you should print an appropriate message and quit. You may want extend your `output()` function so that it outputs also in list notation but this is not required.

If, in any of these steps, you run into an error, you should output an appropriate error message, raise an exception, which the `main()` function should catch, and then continue to read the next s-expression. Note that you may run into an error not only during `input()` but also during evaluation. The main errors of this kind are, *unbound atom*, *all the* `bi`*'s in a* `COND` *evaluating to* `NIL`; use of a function that has not been previously defined; if you come across other kinds of errors, please email or post on Piazza. Note also that you may assume that the `input()` function will not encounter any errors in this part of the project (since those errors really belong in part 1). So the only errors you have to worry about are those that occur during evaluation of the Lisp expressions.

What To Submit And When: On or before 11:59 pm, March 23, you should submit this part of the project. You should submit the same types of files as for part 1: i.e., your source file(s) which includes a standard comment of the form: Author: xxx. The second file should be a Makefile. The third file should be a design-and-documentation file [called 'designP1.txt']; 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 fourth file [called README] should contain clear instructions on how to use (that 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 include `COND`'s with all `bi`'s evaluating to `NIL`, else the machine will crash"] should go in this file. The grader will look at all the files, compile and run your interpreter as per the instructions in your README file, and then assign the grade. Make sure your project works in the standard Linux environment. Do NOT submit tar files; do NOT submit object files; do NOT submit the project by e-mail; if your CSE account has suddenly stopped working, get it to work before the deadline; ....

Additional details: The LISP you implement should include the following primitives:

```
T, NIL,
CAR, CDR, CONS, ATOM, EQ, NULL, INT,
PLUS, MINUS, TIMES, QUOTIENT, REMAINDER, LESS, GREATER
COND, QUOTE, DEFUN.
```

`T` and `NIL` represent true and false. In general atoms may be identifiers or integers. As in part 1, an identifier will start with a letter of the alphabet, and maybe followed by zero or more letters or digits; only uppercase letters will be used. No underscores or other characters are allowed in identifiers. Integers may be signed or unsigned, i.e., 25, $+25$, -25 are all legal; no more than 10 characters in an identifier or integer; you may assume that all integers will fit in a standard 32-bit word.

`CAR, CDR, CONS` are the standard LISP primitive functions. `ATOM` returns `T` if its argument is atomic, and returns `NIL` otherwise. `INT` returns `T` if its argument is an atom that is a number. `EQ` works only on atomic arguments; it returns `T` if its two atomic arguments are the same [or equal, for integers] and `NIL` otherwise. `NULL` returns `T` if its argument is `NIL` and `NIL` otherwise (non-atomic arguments should be acceptable to `NULL`).

`PLUS, MINUS` etc., take two integers as their arguments and return the result, an integer. `LESS` and `GREATER` allow you to compare two integers and return `T` if the first is less or greater than the second respectively.

`COND` and `QUOTE` are the standard LISP forms. `DEFUN` allows the user to define a new function and use it later. A function definition looks like

```
(DEFUN F (X Y) fb)
```

where `F` is the function being defined; its formal parameters are named `X, Y`; and its body is the lisp-expression `fb`. When this is "evaluated", it should add a pair `(F . ((X Y) . fb))` to the d-list [the list of definitions that your interpreter should maintain internally; actually, how you store function definitions on the d-list is upto you; the above is only one possibility]. The features of pure LISP that have not been included are: `LAMBDA, LABEL`. These are not needed because `DEFUN` is sufficient for defining new functions.

When submitting your lab, use the following command:

```
submit c6341aa lab2 .
```

assuming that you are in the directory that contains only the source files, the Makefile, the README file, and the documentation file, that you are supposed to submit.