

CSE 755, LISP Interpreter Project

Winter '12

Note: This is an *individual* project. Discussions with other students in the class is permitted and should, preferably, be carried on in the course newsgroup. But the code you submit must be your own. If you borrow anything from any source, you must document it carefully in your submission.

Grade: You will complete the project in two parts. The first part is worth 40 points; the second is worth 60 points.

Goal: The goal of the project is to implement an interpreter for pure LISP [more or less; see details below]. 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.

What To Submit And When: On or before 11:59 pm, Feb. 22, you should submit the first part of your project; on or before 11:59 pm on March 7, you should submit the second part. In both cases, you should submit four files. One should be your source file; this file should be named `interpreterP1.xxx` (and `interpreterP2.xxx`) where 'xxx' is whatever is the appropriate extension for whatever language you are using. This file should include 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 `READMEP1`] 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 put two periods in a row, 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. (You may use alternate, reasonable file structures –for example, many Java programmers use a fairly large number of files– but the README file must explain this clearly, the Makefile should take care of all the details with regard to compiling etc., and the design.txt file should be the main documentation file.) Your lab must run on the Standard Sun set-up; if you use things like Visual C++ to develop your lab, **you are responsible** for converting them to use Gnu C++ and make sure that it works as intended on our standard Suns. *Make sure of this as you are developing the lab; don't wait until the last day to discover problems.*

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. An identifier will start with a letter of the alphabet, and maybe followed by one 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.

`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.

Approach: The ‘main’ part of the interpreter is already done for you, in the form of the functions `apply`, `eval` etc. All you have to do is to hand-translate them into C++ [or whatever you are using]. What is left are the input function, the output function, and the implementation of the S-expression type. The first part of the interpreter will just consist of the input and output functions; it will also have to include the SExp class (see below). The second part of the interpreter will be the actual interpreter, i.e., the *eval*, *apply* and other functions. The input function has to deal with both the list notation and the dot notation. Essentially, it should read in an s-expression (in list notation or dot notation or mixed) and create the corresponding binary tree. The output function should similarly take a binary tree as input and output the corresponding s-expression to the standard output stream. For the first part of the interpreter, the output should be in dot notation. For the second part, you may extend the output function so that it outputs also in list notation but this is not required.

Perhaps your biggest task is going to be the class corresponding to S-expressions. This class will provide all the primitive functions [including arithmetic functions], and only the primitive functions, i.e., functions like `eval` should not be in this class. Note that you must strictly follow the principle of data abstraction (or information hiding); in other words, functions outside the S-expression class should not know anything about the details of how s-expressions are stored.

When submitting your lab, use the following command:

```
submit c755aa lab1 .
```

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. For the second part, you should change the `lab1` in the line above to `lab2`. Please DO NOT submit object code, .doc files, etc.