CSE 6341: LISP Interpreter Project, Part 2

Overview
Project 2 should be built on top of your scanner and parser from Project 1. The goal is to implement an interpreter that handles all language features *except* user-defined functions. As before, *your submission should compile and run in the standard environment on stdlinux.*

The language and its operational semantics were discussed in class. Read the lecture notes very carefully: there are many assumptions, restrictions, and details related to *this particular variation* of LISP. Some aspects of the language semantics differ from the LISP installation on *stdlinux* (you can type *clisp* at the command line to get to that version). *Do not use the behavior of the LISP installation on stdlinux as a correctness criterion.* Write an interpreter for the exact language presented in class: every valid expression should be evaluated correctly, and every invalid expression should lead to an error message.

Evaluation of S-Expressions
Using the binary tree representation for S-expressions (from Project 1), develop an evaluator for these expressions. This is an implementation of function **eval** from the lecture notes, but with one very significant restriction: there are no user-defined functions. Therefore, both the a-list and the d-list are always empty.

As soon as a top-level S-expression is read from *stdin* and parsed, it should be evaluated and the resulting value should be printed using the printing approach from Project 1. ("Top-level" here means that the S-expression is not nested inside another S-expression.) After this, the next expression should be read and evaluated. The following primitives (and only they) are part of the input language for your interpreter for Project 2: T, NIL, CAR, CDR, CONS, ATOM, EQ, NULL, INT, PLUS, MINUS, TIMES, QUOTIENT, REMAINDER, LESS, GREATER, COND, QUOTE. If the input S-expression is not restricted to these primitives, you need to print an error message.

The semantics of these primitives was defined in class. A few additional details are shown below. If you have any doubts about your understanding of a primitive, talk with me.

- Built-in arithmetic functions PLUS, MINUS, TIMES, QUOTIENT, REMAINDER: each function takes two integer arguments and returns an integer. Any other combination of arguments is illegal. Do not worry about overflow or division by zero: your interpreter can assume that it will never happen.
- Built-in relational functions LESS, GREATER: each function takes two integer arguments and returns T or NIL. For example (LESS (PLUS 2 3) 6) evaluates to T and (GREATER 5 6) evaluates to NIL.
- Notation such as ' for QUOTE, + for PLUS, - for MINUS, … is not part of the input language. If your interpreter gets input that uses this notation, it should report an error.
- (REMAINDER 5 3) is 2; (REMAINDER -5 3) is -2; (REMAINDER 5 -3) is 2; (REMAINDER -5 -3) is -2. For any X and Y, (PLUS (TIMES (QUOTIENT X Y) Y) (REMAINDER X Y)) is equal to X.

Output
All output should go to UNIX *stdout*. This includes error messages – do not print to *stderr*. For each input S-expression, print the computed value (i.e., the S-expression produced by the interpreter) on a separate line. Use the printing approach from Project 1.

Invalid Input
Your evaluator should recognize and reject invalid input. Handling of invalid input is part of the language semantics, and it will be taken into account in the grading of the project. Think carefully about the possible errors (based on the discussions in class), and make sure that your implementation handles them.

The errors should be checked when you attempt to evaluate some expression and see a problem. For example, if you are trying to evaluate (CAR …) and the parameter expression evaluates to an atomic value, you will report the error at this time. Similarly, if you are trying to evaluate expressions such as (ATOM . 77) or (CONS 88 99 NIL), you will observe that *car[expr]* is the name of a built-in function, but the rest of the expression does not conform to the expected syntactic structure (e.g., *expr* is not a list, or is a list with the wrong number of elements) and will report an error. The error messages should be as in Project 1.

Testing Your Project

Part of the work for developing an interpreter (or a compiler) is to design good test cases. There is enough information in the lecture notes to come up with a few simple tests, but you should spend some time designing your own test cases and using them to validate the correctness of your interpreter. This process will be quite valuable, as it will ensure that you analyze the detailed semantics of the language and think about (1) coverage of this semantics in your test cases, and (2) possible invalid programs. You need to prepare and submit two files, ValidTests and InvalidTests. Each line in these files should be a separate test case. Each valid test should be an expression that can be successfully evaluated (do not include the expected output in ValidTests, just include the input expression). Each invalid test should result in an error; focus on problems that are not scanner/parser errors, but rather occur during the evaluation.

Project Submission

On or before 11:59 pm, **October 27 (Tuesday),** you should submit the following:

- One or more files for the interpreter (source code); this should include all files for the scanner, the parser, the interpreter, and the pretty-printer. Make any necessary modifications to your code from Project 1. However, *do not use someone else's code for scanning/parsing/printing* – use your own Project 1 code, modified as necessary.
- Makefile and Runfile, as in Project 1
- Text files ValidTests and InvalidTests, with a separate test case on each line (do not include the expected output)
- Text file README.txt, containing
    - Your name on top
    - Additional details the grader may need to build and run your interpreter
    - Design information: describe briefly the overall design. Point to parts of your code that implement the expression evaluation. This text can be short, e.g., 1-2 paragraphs.
    - If you borrowed ideas or anything else from anywhere, describe briefly.

Login to *stdlinux*, in the directory that contains your files. Then use the following command:
submit c6341aa lab2 README.txt Makefile Runfile ValidTests InvalidTests file1 file2 …
Make sure that you submit only the files described above: do not submit files x.o, x.class, x.doc, etc.

If the time stamp on your electronic submission is **12:00 am on the next day or later**, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will **not** be accepted and you will receive zero points for this project. If you resubmit your project, this will override any previous submissions and only the latest submission will be considered – **resubmit at your own risk**. If the grader has problems with compiling or executing your program, she/he will e-mail you; you must respond within 48 hours to resolve the problem. Please check often your email accounts after submitting the project (for about a week or so) in case the grader needs to get in touch with you.

Grading

The grader will build you project using make and will run it as shown in Runfile, using an extensive set of test cases. The grader will then read file README.txt and will check your code, and then will assign a grade. The grade will be primarily based on correctness on valid inputs and handling of invalid inputs.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own: all design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or for documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with **severe** consequences. Details on academic integrity are available from the Committee on Academic Misconduct (*http://oaa.osu.edu/coamresources.html*).
I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.