CSE 6341: LISP Interpreter Project, Part 3

Overview

Project 3 should be built on top of your Project 2 implementation (with any bug fixes for Project 2). The goal is to extend the interpreter to handle 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

In addition to the expressions handled in Project 2, you need to implement handling of DEFUN expressions and expression that apply (i.e., call) the used-defined functions.

A DEFUN expression must be of the form (DEFUN F E_1 E_2) where F is a literal atom different from CAR, CDR, CONS, ATOM, EQ, NULL, INT, PLUS, MINUS, TIMES, QUOTIENT, REMAINDER, LESS, GREATER, COND, QUOTE, DEFUN. Expression E_1 must be a list (could be an empty list) whose elements are literal atoms; these are the names of formal parameters. The elements of this list must be different from each other and must be different from T and NIL. Expression E_2 is arbitrary.

DEFUN expressions are guaranteed to appear only as top-level expressions – they will never be nested inside other expressions. You can also assume that the same function name will not be used again later in another DEFUN expression (although in real functional languages that is certainly possible). Do not check these conditions, just assume that they are always true. After you parse a DEFUN expression, check that it satisfies all restrictions on its name and parameter list; if it does not, report an error and exits to the OS. Otherwise, remember all relevant information in the d-list (or your internal data structure for storing information about user-defined functions). As the result of evaluating the DEFUN expression, print the name of the function to *stdout*.

An expression that applies a user-defined function must be of the form (F E_1 E_2 …). The first element of the list must be a literal atom that matches the name of a function that was defined earlier through DEFUN. The number of parameter expressions must be equal to the number of formal parameters in the function definition. You need to check these conditions before you evaluate any parameter expression E_i. If there is a violation, report an error and exit to the OS. If there is no violation, the processing of the expression is done as described in the lecture notes; see function **apply** and all other relevant helper functions. The value computed by (F E_1 E_2 …) should be printed to *stdout*.

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. The error messages should be as in Projects 1 and 2.

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. In particular, make sure that your interpreter handles correctly *recursive functions*; several examples of such functions were discussed in class. You need to prepare and submit two files, ValidTests and InvalidTests. 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 specific to Project 3 and are not related to the features you implemented in Projects 1 and 2.

Project Submission

On or before 11:59 pm, **November 12 (Thursday),** 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 Projects 1 and 2. However, *do not use someone else's code for evaluation, scanning, parsing, or printing* – use your own Project 1 and 2 code, modified as necessary.
- Makefile and Runfile, as in Projects 1 and 2
- Text files ValidTests and InvalidTests (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 new functionality specific to Project 3. 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 lab3 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.

Important: I will be out of town at a conference on November 12, and we will not have a class on that day. Although I will make every effort to answer emails while I am away, my response time will be slower than usual. Please do **not** wait until the last minute to ask clarification questions. Start work early and give yourself plenty of time before the deadline to resolve any outstanding issues.

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.