CSE 6341: Lisp Interpreter Project, Part 4

Overview

Project 4 should be built on top of your Project 3 implementation (with any bug fixes for Project 3). 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.*

Conceptually, *eval* is extended to be $eval : S\text{-}expression \times S\text{-}expression \times S\text{-}expression \rightarrow S\text{-}expression$ where the first parameter is the expression *s* to be evaluated, the second parameter is the current association list *a*, and the third parameter is the current function definition list *d*. The language and its operational semantics were discussed in class. Read the lecture notes very carefully, and follow precisely the definition of this particular variation of Lisp.

Evaluation of S-Expressions

As in Project 3, for each top-level expression *s*, you will compute the value of *eval*(s, NIL, *d*) where *s* is the binary tree for that input expression, NIL is an empty association list, and *d* is the current function definition list (d-list). The resulting value is also a binary tree. You will pretty-print this result as was done in Project 3, using list notation.

In addition to the expressions handled in Project 3, you need to implement handling of

- DEFUN expressions
- Expressions that apply (i.e., call) the used-defined functions

A DEFUN expression must be of the form (DEFUN F s_1 s_2) where F is a literal atom different from T, NIL, CAR, CDR, CONS, ATOM, EQ, NULL, INT, PLUS, MINUS, TIMES, LESS, GREATER, COND, QUOTE, DEFUN. Expression s_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, NIL, CAR, CDR, CONS, ATOM, EQ, NULL, INT, PLUS, MINUS, TIMES, LESS, GREATER, COND, QUOTE, DEFUN. Expression s_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 s_1 s_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 s_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 a top-level expression (F s_1 s_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, 2, and 3.

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. I strongly urge you to use function MEMBER as one of your test cases, and to run it on a list with a few thousand elements (you can write a helper function to produce such a long list). If you use Java, you may have to use the Xss option, e.g., `java -Xss20m Main`. This will set the call stack to 20MB; otherwise, the call stack may be too small and you may get a stack overflow exception.

Project Submission

On or before 11:59 pm, **April 1,** you should submit the following:

- One or more files for the scanner, parser, and interpreter (source code)
- A makefile Makefile such that *make* on *stdlinux* will build your project to executable form.
- A text file called Runfile containing a single line of text that shows how to run the interpreter on *stdlinux*.
- If there are any additional details the grader needs to know in order to compile and run your project, please include them in a separate text file README.txt
- ***To simplify grading, put all these files in a single archive file: e.g., x.zip or x.tar.gz. All files should be at the top level in the archive file – that is, do not use any directories in the archive.***

Submit only the files described above: do not submit files x.o, x.class, x.doc, etc.

Login to *carmen.osu.edu* to upload the project. You can submit up to 24 hours after the deadline; if you do so, your project score will be reduced by 10%. If you submit more than 24 hours after the deadline, the submission will not be accepted and you will receive zero points for this project.

If the grader has problems compiling or executing your code, 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.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class 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. Additional details on academic integrity are available from the Committee on Academic Misconduct (see *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.