

CSE 755: Lisp Interpreter Project

Overview

The goal of this project is to build an interpreter for the version of Lisp presented in class. You must use C++ or Java for the implementation. If there is absolutely, positively no way you can successfully do the project using one of these two languages, talk to me. Do not use scanner generators (e.g., lex or jflex) or parser generators (e.g., yacc or CUP). **Your submission should compile and run in the standard environment on stdsun.** If you work in some other environment, it is your responsibility to port your code to stdsun and to make sure that it works there. If you are using Java, you must subscribe to JDK-CURRENT using the “subscribe” program, and use this version of Java. For C++, use gcc on stdsun.

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. For example, some aspects of the language semantics differ from the Lisp installation on stdsun. Do not use the behavior of the Lisp installation on stdsun as a correctness criterion. You are supposed to write an interpreter for the *exact language* presented in class: every valid input expression in this language should be evaluated correctly, and every invalid input expression in this language should be rejected with an error message.

Lexical Analysis

The lexical analyzer (a.k.a. scanner) should process as input a sequence in ASCII characters and should produce a sequence of tokens that serve as input to the parser. The parser performs syntactic analysis of the token stream, according to some context-free grammar. Getting the tokens is typically done on demand: the parser asks the scanner for the next token, in order to complete the application of some production from the grammar. A simple example of a scanner and a parser is shown in Chapter 2 of the compiler book by Aho, Lam, Sethi, and Ullman (ALSU). The book is “Compilers: Principles, Techniques, and Tools, 2nd edition” (published 2007) and is available on reserve at SEL.

The scanner should read its input from *stdin* in Unix. The input contains a non-empty sequence of lines; to get them, you could use the C++ or Java libraries for file I/O from *stdin*. For example, in Java, at the UNIX prompt you would type `java Interpreter < f1 > f2` to process the lines of text in file *f1* and to write the output to file *f2*. The tests distributed with the project are stored as separate text files and are executed in this manner. The interpreter should exit back to the OS after processing the entire sequence of input lines (i.e., when EndOfFile is reached). If an unexpected EndOfFile is encountered (e.g., in the middle of an S-expression), an error message should be printed and the interpreter should exit to the OS.

The input corresponds to one or more S-expressions. For example, the input could be

```
Input line 1: ( PLus 5
Input line 2:+6
Input line 3:)
Input line 4:(QuotE      (1 . 2
Input line 5:
Input line 6:      ) )
EndOfFile
```

Your scanner will produce the appropriate sequence of tokens. The rest of the interpreter will parse and evaluate each S-expression, and print the resulting S-expression on a separate line:

```
Output line 1:11
Output line 2:(1 . 2)
```

Each input line contains a sequence of characters; however, it is possible that some lines are empty (as in line 5 from above). You are guaranteed that only the following characters will occur on a line: letters ('a', ..., 'z', 'A', ..., 'Z'), digits ('0' ... '9'), opening parenthesis '(', closing parenthesis ')', dot '.', plus '+', minus '-', and standard whitespaces: space and tab. The line will end with a newline. There are four categories of tokens produced by the scanner: Atom, OpenParenthesis, ClosingParenthesis, and Dot. Feel free to add convenience tokens such as EndOfFile or ErrorToken, if you deem necessary. Each token is guaranteed to be entirely contained within a single input line.

Atoms are either literal atoms or numeric atoms. A *literal atom* starts with a letter, followed by a (potentially empty) sequence of letters and/or digits. No other characters are allowed in a literal atom. A *numeric*

atom is an integer (signed or unsigned): e.g. 23, -23, or +23. The only characters that are allowed (on the same input line) to be immediately before or immediately after an Atom token or a Dot token are a whitespace, '(', and ')'. For example, (QUOTE (4 .6)) is invalid and the interpreter should reject it with an error message.

For multi-line expressions, your scanner should automatically read the next input line as necessary, until the parser has processed the entire expression. The scanner should be completely case-*insensitive*. To make the output uniform, the scanner should translate every letter to upper case; the parser/evaluator should work only with upper-case letters.

Syntactic Analysis

The parser processes the stream of tokens produced by the scanner, and builds the tree representation of the corresponding S-expression. The input may use the list notation, the dot notation, or any legal combination. For example, (a b c), (a . (b . (c . NIL))), (a . (b c)), and (a . (b . (c))) are valid inputs. You should make sure that the parser accepts all valid token streams, and rejects all invalid token streams. To avoid the guesswork (i.e., “is my parser really covering all possible cases?”), consider defining a grammar and then systematically constructing a parser that accepts all and only valid strings produced by the grammar. One possible approach is to use the rather strange grammar shown below:

$$\begin{array}{lll}
 \langle S \rangle ::= \langle E \rangle & \langle E \rangle ::= \mathbf{atom} & \langle E \rangle ::= (\langle X \rangle \\
 \langle X \rangle ::= \langle E \rangle \langle Y \rangle & \langle X \rangle ::=) & \langle Y \rangle ::= . \langle E \rangle) \\
 \langle Y \rangle ::= \langle R \rangle) & \langle R \rangle ::= \varepsilon & \langle R \rangle ::= \langle E \rangle \langle R \rangle
 \end{array}$$

This is an example of an *LL(1) grammar* (ALSU, p.222) that accepts both dot notation and list notation. Several techniques can be used to build *predictive parsers* for LL(1) grammars (e.g., ALSU, page 64 and Section 4.4). This particular grammar should work for our problem, even though it probably could be improved. Of course, if you want, feel free to define your own grammar and to build a parser for it.

The parsing process typically constructs a parse tree. However, in this project we want to construct the binary tree representation of the S-expression. So, while the parser is applying its productions, you should be building (incrementally) the corresponding binary tree representation of the S-expression that is being parsed.

Evaluation of S-Expressions

The operational semantics for the evaluator was presented in class. You will have to implement a data structure for S-expressions, and an algorithm for evaluating these expressions. As soon as a top-level expression is read from *stdin* and parsed, it should be evaluated. Only after this, the next expression should be read and parsed. The following primitives (and only they) are part of the input language for your interpreter: T, NIL, CAR, CDR, CONS, ATOM, EQ, NULL, INT, PLUS, MINUS, TIMES, QUOTIENT, REMAINDER, LESS, GREATER, COND, QUOTE, DEFUN. 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.
- Each function parameter should be a literal atom. The parameters of a function should be different from each other, and different from T and NIL.
- You are guaranteed that (DEFUN ..) will appear only as a top-level expression, not as a sub-expression nested in some outer expression. Do not check this property, just assume that it is true.
- Names of built-in functions, or COND/QUOTE/DEFUN cannot be used as function names in DEFUN

- (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

The output should be printed in a form that makes the grader's life easier. 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. If an input expression is (DEFUN F ...), just print as output the upper-case name of the function.

If the output value happens to be a list, use the *list notation*. Otherwise, use the *dot notation*; but if some subexpression is a list, use the list notation for that subexpression. More precisely, you must use (conceptually) the following printing algorithm

- For each non-leaf node in the binary tree for an S-expression, compute a flag `isList`. For a node `n`, `isList(n)` is true if and only if the subtree rooted at `n` is a list.
- Printing an S-expression is a recursive top-down process. If the current tree node is `n`:
 1. If `n` is a leaf, just print the atom at that leaf
 2. Otherwise, if `isList(n)`, print the entire subtree rooted at `n` using the list notation.
 3. Otherwise, print “(“, call **print** recursively on the left child of `n`, print “.” (note the spaces around the dot), call **print** on the right child of `n`, and finally print “)”.
- The recursion starts with the root node of the tree.

For example: the S-expression (4 . (A . B)) should be printed as (4 . (A . B)), the S-expression (4 . (A . NIL)) should be printed as (4 A), the S-expression ((4 . NIL) . (A . B)) should be printed as ((4) . (A . B)). Note that this printing algorithm is different from the one used by the Lisp installation on `stdsun` where, for example, expression (4 . (A . B)) is printed as (4 A . B). Do not use this style: your printing should follow the conceptual algorithm from above.

Invalid Input

Your scanner, parser, and evaluator not only should process correctly all valid input, but also 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.

Most errors will be checked when you attempt to evaluate some expression and see a problem. For example, if you are trying to evaluate (CAR X) and X is currently bound to an atomic value, you will report the error at this time. The only checks that need to be performed immediately after parsing, before the actual evaluation, are for DEFUN expressions. If the input expression is (DEFUN F (X Y) ...), it is not really evaluated – the effect is simply the update of the d-list. At the time when the d-list is updated, you should check the constraints on DEFUN from the lecture notes (e.g., the formals should be distinct, etc.). Do not wait for a call to F to check these constraints.

For any error, you have to catch it and print a message. The message must have the form “ERROR: some description” (ERROR in uppercase). The description should be a few words and should describe the source of the problem. Your interpreter should not crash on invalid input (no segmentation faults, no uncaught exceptions, etc.). If the input expression is invalid, a message “ERROR: ...” should be printed and the interpreter should exit back to the OS. Reporting meaningful error messages in compilers and interpreters is a very hard problem – for example, everyone has seen strange errors reported by real-world compilers. Thus, I do not expect any fancy error messages for this project. *Up to 30% of your score will depend on the handling of incorrect input, and on printing error messages as described above.*

Testing Your Project

On the course web page I have posted two tar files: `projcpp.tar` and `projjava.tar`. Each file contains a simple Makefile and a set of tests. Please examine the Makefile and the tests carefully. You must make sure that your implementation works correctly on the provided tests. For the tests containing valid inputs, you need to do something like (using Unix redirection with `<` and `>` from the Unix shell)

```
myinterpreter < test5 > test5.out; diff test5.out test5.expected
```

You should get no differences from diff, except for trivial differences such as white spaces. For the tests containing invalid inputs, something like

```
myinterpreter < invalid2 > invalid2.out; more invalid2.out
```

should put a message “ERROR: ...” in file *invalid2.out* and return back to the OS. The test cases in the tar files are very, very weak. Of course, you must do extensive testing with many other test cases. Read carefully the lecture notes and be very thorough with all details.

Avoiding Simple Mistakes

Sometimes people lose points for things that are easy to avoid. Specifically, make sure the project reads from *stdin* and writes out to *stdout*. You'll lose points if you don't do it. The grader should be able to run it (as a Java program):

```
java Interpreter < inputfile > outputfile
```

or

```
Interpreter < inputfile > outputfile
```

(for C++). Common variations such as

```
Interpreter inputfile > outputfile
```

```
Interpreter inputfile < outputfile
```

```
Interpreter inputfile outputfile
```

```
Interpreter hardcoded-filename-to-read-in-from hardcoded-filename-to-write-out-to
```

will definitely cost you points. If you don't know how to handle *stdin* and *stdout* in either C++ or Java, feel free to ask the grader – he/she would be more than happy to help you with it.

Follow the specifications for the input and output formats precisely. It's obviously not your biggest concern, but a lot of people needlessly lose points from an otherwise perfect project. This includes the handling of invalid input exactly as specified in the project description.

Project Submission

On or before 11:59 pm, **November 14 (Monday)**, you should submit the following:

- One or more files for the interpreter (source code)
- A makefile Makefile such that *make* on stdsun 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 stdsun.
 - For example, if your makefile produces an executable file called *myinter*, file Runfile should contain the line of text *myinter*
 - Or, for example, if you are using Java and class *MyInterpreter* contains main, Runfile contains *java MyInterpreter*
- Text file README.txt, containing
 - Your name on top
 - Additional details the grader may need in order to build and run your interpreter
 - Design information: describe the overall design, and any unusual aspects of the design or the implementation. *The design description will be worth up to 20 points.* Your description should be at least 3-4 paragraphs long.
 - If you borrowed ideas or anything else from anywhere, describe briefly.

Get on stdsun, in the directory that contains the files you are supposed to submit. Then use the following command: “**submit c755aa lab1 .**” 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 both your xyz.567@osu.edu and xyz@cse.ohio-state.edu 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 your code, and will assign a grade. The project is worth 100 points. 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 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 the 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.