CSE 6341: Lisp Interpreter Project, Part 1

Overview

This sequence of projects builds an interpreter for a version of Lisp presented in class. You must use C, C++, or Java for your implementation of all projects. Do **not** use scanner generators (e.g., lex or jflex) or parser generators (e.g., yacc or CUP). **Your submission must compile and run in the standard environment on *stdlinux*.** If you work in some other environment, it is your responsibility to port your code to *stdlinux* and to make sure that it works there. **Do not wait until the last minute to compile and test on *stdlinux*** – past experience shows that last-minute problems often occur and lead to low project scores.

For Java: run "subscribe" on *stdlinux* and add **JDK-CURRENT**. For C/C++: use gcc/g++ on *stdlinux* but first run "subscribe" and add **GCC-7.2.0**. After running "subscribe", log out and then log in again.

Project 1

For the first project, you will build a *lexical analyzer* (also known as *scanner*). The rest of the projects will build on your implementation from Project 1. The input to the scanner is a sequence of ASCII characters. You are guaranteed that the only characters you will ever see in the input are:

- upper-case letters: A, B, …, Z
- digits: 0, 1, …, 9
- parentheses: ( )
- ASCII white spaces: space (ASCII value 32), carriage return (ASCII value 13), and line feed (ASCII value 10). In C/C++/Java, use escape sequences \r and \n to refer to carriage return and line feed, respectively.

Your code should not check that the input satisfies this constraint; assume it is always satisfied.

The scanner should read its input from *stdin* in Unix. The input contains a non-empty sequence of characters. To run a scanner written in Java (similarly for C/C++), you and the grader will use

java Interpreter < file1 > file2

from the Unix command line to process a program from *file1* and to write the output to *file2*.

The ASCII characters in *file1* are used to form five kinds of *tokens*: Atom, OpenParenthesis, ClosingParenthesis, ERROR, and EOF. An *atom token* is one of two categories: a literal atom or a numeric atom. A *literal atom* is a non-empty sequence of digits and upper-case letters, starting with a letter. A *numeric atom* is a non-empty sequence of digits. Token ERROR is an artificial token produced when the scanner needs to report a scanning error. Token EOF is an artificial token produced when the "end-of-input-file" event occurs.

The main function of the scanner will contain a loop calling helper function getNextToken. The loop will exit when getNextToken returns the artificial tokens ERROR or EOF. Inside the loop, the main function should keep counters for how many literal atoms, numeric atoms, open parentheses, and closing parentheses were found in the input file. You can assume that each counter will not overflow `int` type in C, C++, or Java. For the literal atoms, the actual ASCII sequences should also be remembered in the order in which they were found (so that they can be printed later). The sum of the values of all numeric atoms should also be computed in the loop. You can assume that this sum will not overflow `int` type in C, C++, or Java. After the loop exits, the following should be printed to Unix *stdout* in **exactly** the following format:

LITERAL ATOMS: number of atoms, atom1, atom2, …
NUMERIC ATOMS: number of atoms, sum of all atoms
OPEN PARENTHESES: number of atoms
CLOSING PARENTHESES: number of atoms

For example, if the input file contains (DEFUN F23 (X) (PLUS X 12 55)) the output should be

LITERAL ATOMS: 5, DEFUN, F23, X, PLUS, X
NUMERIC ATOMS: 2, 67
OPEN PARENTHESES: 3
CLOSING PARENTHESES: 3

If during the execution of the loop getNextToken returns an ERROR token, the only output is the error message (described below); in this case LITERAL ATOMS: …, etc. should not be printed. After the error message is printed, the scanner should immediately exit to the OS.

getNextToken

This helper function forms one token starting from the current position in the input. In the process of forming the token, it "consumes" ASCII characters from the input, until it find enough characters to form a token. At a high level, it applies these steps in the order in which they are listed:

1. If the input is empty, returns token EOF
2. If the current character is a white space, consumes it and any white spaces that follow it; after this is done, checks if the input is empty and returns EOF if it is
3. If the current character is '(' it consumes it and returns token OpenParenthesis
4. If the current character is ')' it consumes it and returns token ClosingParenthesis
5. If the current character is letter/digit, consumes it and all letter/digit characters that follow it. The resulting string is either a literal atom (e.g., "XY3Z"), a numeric atom (e.g., "3415"), or an error (e.g., "34XY"). In the first two cases, an Atom token is returned, together with all relevant information about the atom: its type (literal/numeric) and its value. The value is a string for a literal atom, and an integer (i.e., an `int` in C/C++/Java) for a numeric atom. In the last case, an ERROR token is returned, together with the string value of the bad token, e.g., "34XY". If the main function of the scanner calls getNextToken and receives back an ERROR token, it should print an error message and then immediately exit to the operating system (the rest of the input is ignored). The message should be of the form
   ERROR: Invalid token …
   ERROR is in upper case. The token is also printed: e.g., ERROR: Invalid token 34XY.

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***. The grader should be able to run it as follows:

For Java: java Interpreter < inputfile > outputfile
For C/C++: ./Interpreter < inputfile > outputfile

If you do not know how to handle *stdin* and *stdout* in C/C++/Java, ask the grader.

Project Submission

On or before 11:59 pm, **January 16 (Wednesday),** you should submit the following:

- One or more files for the scanner (just the source code)
- A makefile Makefile such that *make* on *stdlinux* will build your project to executable form. Examples of makefiles are shown on the web page under "Projects"

- A text file called Runfile containing a single line of text that shows how to run the interpreter on *stdlinux*.
    - 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*

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 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.