CSE 6341: LISP Interpreter Project, Part 1

Overview

The overall goal of this sequence of projects is to build an interpreter for the 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 should 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. If you are using Java, you must subscribe to JDK-CURRENT using "subscribe", and use this version of Java. For C/C++, use gcc on *stdlinux*.

Project 1

For the first project, you will build a lexical analyzer, a parser, and a printer. The rest of the projects will build on your implementation from Project 1. The input language for your parser is defined by the following grammar. Here <S-exp> represents an *S-expression* ("symbolic expression"), a key concept in LISP.

<Start> ::= <S-exp> <Start> | <S-exp> **eof**
<S-exp> ::= **atom** | **(** <S-exp> **.** <S-exp> **)**

Here <Start> and <S-exp> are non-terminals. There are five terminals: **atom (.) eof**

An atom, represented by the **atom** terminal, is either 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. Terminal **eof** is an artificial terminal representing the "end-of-input-file" event.

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 (i.e., parsing) of the token stream. Getting the tokens is typically done on demand: the parser asks the scanner for the next token by calling getNextToken, in order to apply of some production from the grammar.

The scanner should read its input from *stdin* in Unix. The input contains a non-empty sequence of characters. You are guaranteed that the only characters you will ever see in an input file are

- upper-case letters
- digits
- (
- .
-)
- white spaces (e.g., space, tab, end of line, etc.)

To run an interpreter written in Java (similarly for C and C++), you and the grader will use

java Interpreter < f1 > f2

to process a program written in file *f1* and to write the output to file *f2*. The tests distributed with all projects are stored as separate text files and are executed in this manner. The ASCII characters in file *f1* are used to form four kinds of tokens: Atom, OpenParenthesis, ClosingParenthesis, and Dot.

The main function of the scanner is getNextToken. It is repeatedly called by the parser. At a high level, getNextToken does the following

1. if the current character is a white space, reads it and any white spaces that follow it
2. if the current character is '(' returns token OpenParenthesis
3. if the current character is ')' returns token ClosingParenthesis
4. if the current character is '.' returns token Dot
5. if the current character is letter/digit, reads 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 is returned back to the parser, together with all relevant information about the atom.

Syntactic Analysis

The parser processes the stream of tokens produced by the scanner. For each parsed <S-expr>, some result is printed, followed by newline. The result depends on the project; for Project 1, the result is pretty-printing of the S-expression.

Rather than building a parse tree for an S-expression, your parser should build a binary tree that captures the structure of an S-expression. The leaves of this tree are atoms. Let T(S) be the binary tree representation of an S-expression S. Tree T(S) is defined as follows:

- if S is an atom, T(S) contains one node which is the atom itself
- if S is (S_1 . S_2), the root of T(S) is a node whose left child is the root of T(S_1) and right child is the root of T(S_2)

A simple way to build your parser is to have two recursive functions ParseStart and ParseSexp. At a high level, the functions work as follows

- Function ParseStart will call ParseSexp and then will check for end of file. If the end is reached, the parser will terminate. If not, ParseStart will call itself.
- Function ParseSexp will get the next token. If it is not Atom or OpenParenthesis, an error will be reported. If it is Atom, the function returns. If it is OpenParenthesis, the function will call itself, then will get the next token, report an error if it is not Dot, call itself again, get the next token, and report an error if it is not ClosingParenthesis.

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.

Output

All output should go to UNIX *stdout*. This includes error messages – do not print to *stderr*. For each input S-expression, pretty-print the expression followed by newline.

Some S-expressions are considered to be lists: (1) the atom NIL is a list, (2) if S_2 is a list, so is (S_1 . S_2) for any S_1. In your implementation, for each inner node in the binary tree for an S-expression, compute a boolean attribute isList. For a node n, n.isList is true if and only if the subtree rooted at n represents a list. This is needed for printing an input S-expression:

- If n.isList for *every* inner node n, print the entire expression using only list notation
- Otherwise, print the entire expression using only dot notation

Invalid Input

Your scanner and parser 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 about the possible errors, based on the earlier description, and make sure that your implementation handles them.

For any error listed above, 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. Your score on this project will depend on the handling of incorrect input, and on printing error messages as described above.

Testing Your Project

I will post several test cases during the next few days. 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 < test1 > test1.out; diff test1.out test1.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 < invalid1 > invalid1.out; more invalid1.out

should put a message "ERROR: …" in file *inalid1.out* and return back to the OS. Of course, you must do extensive testing with other test cases constructed by you.

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 (for Java):

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

or (for C/C++)

```
Interpreter < inputfile > outputfile
```

Common variations such as

```
Interpreter inputfile > outputfile
Interpreter inputfile outputfile
Interpreter hardcoded-filename-to-read-in-from hardcoded-filename-to-write-out-to
```

will cost you points. If you don't know how to handle *stdin* and *stdout* in C/C++/Java, ask the grader.

Project Submission

On or before 11:59 pm, **October 2 (Friday),** you should submit the following:

- One or more files for the 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.
    - 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 briefly the overall design. Point to parts of your code that implement the binary tree representation. 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 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 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 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.