CMSC 399: Senior Conference
Project: A Lisp Interpreter
Spring 2019

Project: In the language of your choice, write a basic Common Lisp interpreter.

Lisp is a family of languages that has its roots in pioneering work in the 1950s. It is still relevant today as the direct inspiration for the language *Clojure*, which considers itself a dialect of Lisp. This assignment will concern *Common Lisp*, a standard variant of Lisp.

Do this first: Learning Lisp

The first step to implementing a language is to understand the language. To do this, download a Lisp interpreter (I recommend CLISP at https://clisp.sourceforge.io/ .)

There are many Lisp tutorials on the internet. You may also find it helpful to read the first two chapters of Paul Graham's *ANSI Common Lisp* (ACL), available at http://www.paulgraham.com/acl.html .

As a good test of your understanding, write a function that takes in a number and checks whether it is prime. If the number is prime, return `nil` (Lisp's empty list). If not, return a list containing two numbers that multiply to the number passed in (neither number should be 1). Write this function both iteratively (using `do`) and recursively.

Project: A List Interpreter

As a general guide, your Lisp interpreter should be capable of running the examples from Chapter 2 of ACL. Here are some specifics:

1. Your interpreter must have a read-eval-print-loop (REPL). This is a command-line prompt where a user can enter a Lisp expression and get a result. Your REPL should allow for multi-line input: if the user's input does not close all its parentheses, continue to accept input until all the parentheses are closed. The CLISP interpreter handles multiline input in this way.

2. Your interpreter will handle only decimal integers. There is no need to implement floating-point numbers or Lisp rationals, for example. The format for numbers is that they optionally start with either a `+` or a `-`, followed by digits.

3. Lisp is different from other languages in its treatment of *symbols*. A symbol in Lisp can contain some punctuation. Specifically, symbols are all sequences containing only letters, digits, and the following punctuation:
   `+ - * / @ $ % ^ & _ = < > ~ .`
   Exception: sequences of characters that can be treated as numbers are numbers, not symbols. So, `+12` is a number, but `12+` and `1-2` are symbols.

4. Your interpreter need not handle strings or characters.

5. Specifications of the functions you will implement are all available at http://www.lispworks.com/documentation/HyperSpec/Front/ . I recommend using the symbol index to find what you're looking for. Your interpreter must support the following functions/special operators.

   ```
   +  -  *  /  <  >  <=  >=  =  /=
   EQUAL  EVAL  APPLY  QUOTE  FUNCTION  IF  AND  OR  NOT  DEFUN
   LIST  CONS  CAR  CDR  LISTP  NULL  NIL  T
   PROGN  LET  SETQ  DO
   PRINC  TERPRI  LOAD  QUIT
   ```

   Your functions must adhere to the behavior described in the specification, with the following exceptions:

   `/`: Division is just integer division, and it needs to work only with 2 arguments.

   `= /= < > >= <=`: These are all defined in the standard to work with an arbitrary number of arguments; your versions can all just take 2 arguments.

   `=`: This does not have to do a type-check. That is, you can have `=` and `EQUAL` be synonyms.

   `EQUAL`: This does structural comparison. Two lists are `EQUAL` if they have the same length and their components are `EQUAL`. Numbers are `EQUAL` if they denote the same number. Symbols are `EQUAL` if they have the same name. Whether or not functions are `EQUAL` is implementation-dependent (that is, you can do whatever you like).

   `CONS`: You may require that the second argument is a list. That is, you never have to worry about "dot-forms" like `(1 . 2)`.

   `LOAD`: This function normally takes a string. You will write yours to take a symbol, and then it will load the file whose name matches the name of the symbol. This means that the name will always be in all-caps, as symbols are in all-caps. For example, `(load 'lisp-file.lisp)` will load a file `LISP-FILE.LISP`. On Macs and Windows, the capitalization does not matter, but it does on Linux (i.e., Powepuff).

   In addition to the forms above, you must support a function `PRINT-SPACE`, which takes no parameters and prints a space character to the output. Using `PRINC`, `TERPRI`, and `PRINT-SPACE`, you can then print some basic debugging strings.

6. Notes on specific sections from Chapter 2 from ACL:

   2.2: Evaluating a list depends on what's at the head (first element) of the list.

- If the head of the list is a special operator, then the definition of that operator determines how to evaluate the list. Your interpreter will not have *macros*. If you come across a description of a macro, just pretend the page says "special operator".
- Otherwise, if the head is a symbol, it must name a function. Evaluate all the arguments first and then call the function.
- Otherwise, the head of the list must be a lambda-expression (these are explained later in the chapter). Evaluate all the arguments and then call the lambda-expression.

Ignore the section "Getting Out of Trouble".

2.4: You do not need to define `THIRD`.

2.9 (Input and Output): Skip this section.

2.10: You do not need to define `DEFPARAMETER`, `DEFCONSTANT`, or `BOUNDP`.

2.11: We will use the much simpler `SETQ` instead of `SETF`. While `SETF` can be used to assign to all sorts of places in Lisp, `SETQ` allows update of only variables. The only example in this section that `SETQ` can't do is the one with `(setf (car x) 'n)`. In your Lisp interpreter, there will be no way to change an element of a list. This is actually a huge simplification and allows the project to be considerably easier.

2.12 (Functional Programming): Skip this section.

2.13: `DO` is quite intricate. One of the challenges of this project is in getting `DO` working. I recommend doing quite a bit of testing in CLISP to understand how `DO` works better. You do not need to implement `DOLIST`.

2.14: Your output (e.g. for `(function +)`) does not need to match CLISP's exactly. You do not need to implement `FUNCALL`, but you do need to write `APPLY`. In Common Lisp, it is possible to use `lambda` without the `#'` prefix, but you do not need to implement this behavior. For example, it would be correct to write `(apply #'(lambda (x y) (+ x (* y 2))) (4 5))`, but leaving out the `#'` should fail.

2.15 (Types): Skip this section.

7. Posted on our course website is *Eval.java*, the JUnit tests I wrote as part of my implementation. While you need not use Java nor try to duplicate any of the structure (like use of exceptions) that you see in that file, seeing my tests may be helpful in figuring out what your interpreter must do.

8. I expect you to have many questions about what you have to implement and how to proceed. This is deliberately somewhat open-ended. At this stage of your computer

science careers, you should be able to know the questions you need to ask to get you further along. Please post these on Piazza!

Writeup & Presentation

Part of your Senior Conference work is writing up your work. This means that your project is not just about code, but also about English writing. Your writeup will have two main sections:

1. You will describe the structure of your interpreter. This writing should be a tour of how you implemented Lisp; for example, a reader should be well equipped to continue your work and implement more of the language. This text should answer questions such as:
    - How do you represent variable scopes? How do you track information that represents the stack of functions being evaluated?
    - How do you represent evaluation?
    - How do you build in pre-defined operations?
    - How do you extend the environment with new variables. Note that this happens in three ways: `LET/DO`, `SETQ`, and `DEFUN`.
    - How are strings parsed? How do you deal with the special marks `'` and `#'`?
    - How do you deal with the fact that functions can be returned from functions?

    Your writeup should *not* simply be answers to these questions, but it should be detailed enough that it answers these questions while describing your code. It is appropriate to include small snippets of your code within your writeup, but most of the page should be English, not code.

    I expect this to take 6-8 pages, single-spaced.

2. The second part of your writeup focuses on your experience in doing this project. What parts went well for you? What new things did you learn? Are there areas of the project you are especially proud of? Are there areas you wish you could re-do? How did the structure of your code affect how easy it was to complete the project? Again, your writeup should not simply be answers to these questions, but these are written here to communicate the general intent of this section.

    I expect this to take 2-3 pages, single-spaced.

Note that you can complete this writeup regardless of how far you get in your actual project.

Beyond your writeup, you will also have to present your work in a short presentation during the senior celebration day, during finals period. Even if your implementation is incomplete, you will be required to present what you have.