LISP Interpreter Project Report

Pierce Mayadag
New Mexico Institute of Mining and Technology
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
pierce.mayadag@student.nmt.edu

1.

Lisp Construct	Status
Variable Reference	Done
Constant Literal	Done
Quotation	Done
Conditional	Done
Variable Definition	Done
Function Call	Done
Assignment	Done
Function Definition	Done
The arithmetic +, -, *, / operators on integer type.	Done
“car” and “cdr”	Done
The built-in function: “cons”	Done
Sqrt, exp	Done
>, <, ==, <=, >=, !=	Done

Extra Credits Lisp Construct	Status
BIG-Integer on +, -, * arithmetic operators	Not Implemented
Implementing a sequence of 10 or more compressed car&cdr	Not Implemented
“mapcar” built-in function	Not Implemented
“Lambda” built-in function	Not Implemented

2.

In order to implement the above LISP constructs, I created the two LISP types as classes, (Atom and List), a superclass for those types called LispObject, a parser and an evaluator. The parser is made up of two functions, getTokens and readTokens. The function readTokens takes in a string and creates a token from each space-separated sequence of characters. From here, readTokens creates Atoms from any non-parenthesized token and Lists by recursively calling readTokens on each token between parentheses. During this process, any LispObject that is quoted, a number, or a boolean value is marked as “literal”. After this, the evaluator will receive a LispObject, (either an Atom or List) from the parser, for which there are three cases. One, if it is marked as “literal”, then the evaluator will return a string representing that object. Two, if the object is a non-literal Atom, then its name is checked against a map of defined variables for a corresponding value. Three, if the object is a non-literal List, for every object in the list except the first one, the evaluator is recursively called on that object and stored in a List as arguments. Then, if the first object in the list is a name of a known function, then that function is called using the List of arguments. Once the evaluator returns a string, that string is printed to the user.

Every function in the interpreter takes a single List as an argument. The function first checks that the list contains the correct number of elements, then checks that the arguments are the correct type (if it is a built-in function), and then executes the body of its code.

All built in functions that manipulate Lists make use of Java’s built-in List operations.
All built in functions that pertain to numbers/booleans use java’s primitive number and boolean types.

User-defined variables are implemented as a map of Strings and LispObjects. Creating a variable creates a new key-value pair in the map. Accessing a variable checks for the value of a given name within the map.

User-defined functions are implemented using their own class called LispFunction. Instantiation of this class requires a name, list of arguments, and a code body. When the function is called, the List parameter is first checked for the correct number of arguments. Assuming this is correct,

each instance of a formal parameter in the code body is replaced with the actual parameter in the corresponding position. Then the value of calling the evaluator on the code body is returned.

The constructs “if”, “define”, “set!”, and “defun” are handled as separate cases in the evaluator. This is because these constructs require that there parameters are not evaluated first, unlike other functions.

3.

It is clear to me that the most important concept in building this interpreter was recursion. Specifically, there are two types of recursion that make up the meat of the interpreter, parser recursion and evaluator recursion. Parser recursion consists of creating Lists of Lists that recur an arbitrary number of times until an Atom is found as the base case. Evaluator recursion consists of evaluating non-literal objects to other non-literal objects an arbitrary number of times until a literal object is recovered as the base case. These two recursive methods are crucial to the interpreter and rely on the underlying recursive Lisp-type, List.

4.

One challenge I did not forsee initially was handling different edge cases with quotes in the parser. I found that I needed to implement the following rules for interpreting quotes: Objects that would otherwise be non-literal that include a quote in front of them are not printed with the quote, quoted objects within literal lists are printed with the quote, multiple quotes are redundant in the case of otherwise non-literal objects, but objects within literal lists are printed showing all quotes that precede them. My ultimate solution to this was to include two additional parameters to the LispObject class. One parameter keeps track of whether or not the object is literal, the other keeps track of how many quotes should proceed the object when it is printed.

5.

A crucial part of Lisp’s power not present in my implementation is the Atom’s property-list. Given that getprop and setprop were not required to be implemented, I decided to simplify the Atom’s property-list into a single variable for a single value.

6.

So far, I’ve found this project both interesting and enjoyable. I would have to point out that I found the requirements for the project to be vastly unhelpful in terms of how to get started. It would’ve been much more helpful had the project instructions at least mentioned something about implementing Lisp types, a parser, or an evaluator. I was’nt really able to get anywhere until I had discovered the basic workings of a Scheme parser from my first listed source

7. References

[1] Norvig, Peter. (How to Write a (Lisp) Interpreter in (Python)). 2010. Retrieved April 2, 2019 from http://norvig.com/lispy.html.

[2] Soliman, Hamdy. Lecture on the Functional Oriented Paradigm and LISP. March 26, 2019.