Lisp Interpreter

Donovan Griego
New Mexico Institute of Mining and Technology
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
donovan.griego@student.nmt.edu

ABSTRACT

For my lisp interpreter I used Python. The way I accomplish interpreting lisp in a clean and scalable way relies heavily on the list data structure. I first parse the raw string and separate it into elements. I then parse these elements and create lists which are nested based on placement of open and close parentheses. This list is then passed to my evaluation pipeline. There it first passes through an "expand" phase where the function is called recursively on nested lists. This enables me to expand and evaluate function from inside out. This is very scalable and allows me to compute very complex statements.

1. Implementations

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

1.1 Variable Reference

This implementation was a part of the main evaluation pipeline. At this stage, the token is checked against a global variable dictionary. If a match is found then a value is returned in place of the token.

1.2 Constant Literal

Whenever a constant is encountered in the evaluation pipeline it is ignored, thus evaluating its value directly.

1.3 Quotation

When parsing the input string, if a quote is encountered it is moved one element to the right. This will result in the quote being placed inside of its respective list. When it enters the evaluation pipeline it acts as a function call and simply returns the entire list minus the quote before the insides are evaluated. Thus leaving its contents untouched.

1.4 Conditional

This implementation is also placed in the evaluation pipeline. When on the "expand" phase of the pipeline if 'if' is encountered, the next three arguments are left untouched to prevent premature evaluation. After the "expand" phase the token is evaluated and the first argument is checked. If it is found to be true, the second argument is evaluated and returned, otherwise the third is evaluated and returned.

1.5 Variable Definition

In the evaluation pipeline if 'define' is encountered then the first argument is place in the global variable dictionary with the second argument as the value.

1.6 Function Call

At the beginning of the pipeline the first token is checked against the global function dictionary. If a match is found then the formal parameters and definition are fetched. The actual parameters are grabbed and matched up to the formal based on position. This information is stored in a temporary dictionary. The function definition is then checked for matches in the temporary dictionary and its values are replaced with the proper call. This statement then replaces the function call in the original statement before being evaluated. The fact that we learned about scoping in class helped a lot with this implementation.

1.7 Assignment
In the evaluation pipeline if ‘define’ is encountered then the first argument is place in the global variable dictionary with the second argument as the value.

1.8 Function Definition
In the evaluation pipeline if ‘defun’ is encountered then the first argument is place in the global function dictionary with the second argument as the formal parameters and the third as the definition.

1.9 Arithmetic Operators
This is simply done by looking for these operators in the evaluation pipeline. If encountered, they run their operations on the arguments and return their result.

1.10 “car” and “cdr”
This is done in the evaluation pipeline by searching for those keywords. If encountered then they run their respective operations given the arguments and return the proper values. Going over these functions in class and in the homework helped a lot with this implementation.

1.11 “cons”
This implementation is grouped with the others in the pipeline. It puts the first argument into a list then appends the other argument.

1.12 sqrt and exp
This implementation simply takes the arguments and runs the python math functions to return the correct result.

1.13 Comparison Operators
These operators are handled like the rest in that they take arguments and return T if true and Nil for false.

2. Areas of Interest

2.1 Evaluation Pipeline
My implementation of this interpreter relies heavily on recursion to allow extremely complex statements and virtually limitless nesting. It looks at each token and evaluates it separately from the inside-out. This effectively expands the statement to its full form with values included then evaluates it. This was very hard to get set up but worth it because it is extremely scalable and easy to add functions to.

2.2 A Note on Recursion
The definition of functions works perfectly using my implementation, however, when testing defining recursive functions and trying to call them it will throw an exception of hitting the max depth of recursion. It will not evaluated. I have a feeling this has something to do with how I am providing intermediary values between calculations. I am confident in the implementation of the calling but something is wrong when evaluating it. Defining and calling normal functions works fine though.

3. Conclusion
This project was a lot of fun because it prompted me to think outside the box in how I could, from the ground up, build something to understand the complex syntax of lisp. In the beginning I thought through a couple different strategies but they didn’t work. In the end I would say I was successful in final strategy and implementation. I like the problem solving aspect of these kinds of projects because there is more than one solution; Some are smarter than others since since it takes a lot of effort to not do just the bare minimum. In the end I wound up with something I am proud to call my own and will be displaying on my resume.