Senior Conference: A Lisp Interpreter

Ju-Han Tarn

First Draft due Mar 7th 2019

1 Introduction

This is a write up for the assigned project in Bryn Mawr College computer science senior conference. This assigned project is about doing an implementation of a basic Lisp interpreter in my choice of programming language. I chose to use Python to implement this Lisp Interpreter is that Python has very powerful and convenient string manipulations and having this is important to do this project as it has a huge portion that deals with processing strings.

2 Motivation

Functional programming is an interesting area to explore. Implementing a Lisp interpreter not only helps me to understand how an interpreter works, but also gives me an opportunity to learn functional programming. Lisp is a syntactically pure and clean functional language, and writing a Lisp interpreter seems to be a feasible semester-long project. And as Steve Yegge said:

> " If you don't know how compilers work, then you don't know how computers work. If you're not 100% sure whether you know how compilers work, then you don't know how they work." [1]

Since interpreters and compilers are both very important, writing this Lisp interpreter is a great chance to learn more about how computers work.

3 Background

There are two important parts in this project, understanding Lisp and understanding how interpreter works. To start with, Lisp is one of the oldest programming language, and there has been some changes on Lisp. Nowadays, the languages that still possess or inherit the features of Lisp are Common Lisp and Scheme. [2]
As far as I observe, Lisp uses prefix expression, meaning that the operators normally appear first, then the operands, which is different to languages like Python or Java. The structure of the grammar is closely align with functional programming languages such as Haskell and Racket, and parentheses are a big thing in Lisp (just like the semi colons and curly brackets in Java).
Next, we will talk about interpreters. As the name suggests, the interpreter "interprets" the input program and "translates" it to an output. I think interpreters act very similar to translator, the difference is that an interpreter yields a value from given programs. An interpreter is consist of two components, a parser and an evaluator. The function of a parser is that it breaks the input program into individual tokens, and then builds data structure that can represent the input program. After having a data structure for the program, the evaluator receives it and calculates the final value for the program. [**?**]

4 High-level Structure

As discussed above, the interpreter has two components: the parser and the evaluator. In order to have these two key parts working, the program includes helper functions. To start with, the read-eval print loop constantly receives new input and check whether the input has matching parentheses. If the input has matching parentheses, then the program will proceed to parsing and evaluating.
The parser tokenizes input into individual tokens, and then builds a data structure that represents the input. The evaluator will then take this data structure and calculate the corresponding value of the input.

5 Implementation

5.1 Type Definition

Given that this is a basic Lisp interpreter, there are only three types need to be handled: Symbol, List and Numbers

- Symbol: Symbols are treated as strings, it can be implemented as strings in Python.
- List: Lists are treated as lists, it can be implemented as lists and have list methods in Python.
- Numbers: The Lisp interpreter will only deal with whole numbers, so numbers are implemented as integer.

5.2 Tokenizer

Since a parser involves with tokenization, a function called "tokenize" is implemented. The tokenizer plays an important role in the interpreter. White spaces indicate what kind of operations will be executed, so it's crucial to break the input into individual tokens by the white spaces. Therefore, before I split the input strings into tokens, I replace right and left parenthesis with itself and an additional white space. Doing this allows the tokenizer to treat the right and left parenthesis as tokens.
There was actually a considerable amount of time spent on figuring out how to tokenize the strings, especially dealing with parentheses. In the Clisp interpreter, it allows users to input parentheses that have no white spaces to separates the other symbols. For example (+ 1 2). If this is not taking into account, then it would be almost impossible to evaluate the expression since it cannot tokenize correctly. To solve this, I make sure that there is always a white space before and after the parenthesis. For example, making (+ 1 2) into (+ 1 2). Having this operation allows the tokenizer to work correctly.

5.3 Parser

The parser in this project is consist of two functions: read from tokens and atom. The purpose of having read from tokens is to build a reasonable data

structure that can represent the input Lisp program. Since the function needs to be able to handle different kind of representations, I believe using recursion is the appropriate way to proceed as the data structure could get very complicated and unpredictable.
Even though the idea of recursion isn't hard, coming up with a recursive algorithm was quite challenging. I've looked through how other people implemented a recursive algorithm for parsing, but their work don't seem intuitive to me.
I eventually realized that using the parenthesis to build a list of lists is the key to the recursive algorithm. In each recursion, the algorithm will pop off the first token and look at the rest of the tokens. When it reads a left parenthesis, the algorithm will start building a list, return it when a right parenthesis appears. During this process, the atom function will check whether the tokens are integers or strings. Once the process is finished, the algorithm will return a list of lists that represents the data structure of the input strings. This list of lists is a tree structure.

5.4 Evaluator

Since evaluator is also about reading the data representation of the input string, recursion seems to be the most straightforward way to handle it. Before reading it, the program builds a dictionary of functions, (arithmetic functions for now) so when the evaluator reads an operator, the evaluator will map the function of that operator in the dictionary.
The recursive algorithm is similar to the parser mentioned above. The evaluator will read the first token, and extract it if it is an operator, then find its corresponding function in the dictionary. Then the evaluator stores and reads the rest of the list. The algorithm iterates the rest of the list and evaluate if and only if there is a sub-list. After evaluating the sub-list, the value of the sub-list will be stored in the list that the evaluator stored beforehand at the the index of the sub-list.
Once all the sub-lists are traversed, then call the previously stored operator on these sub-lists, which are converted into operands by the evaluator.

5.5 Matching Parentheses

The approach of looking for matching parentheses is rather straightforward. There are multiple ways to do this. One can do this with a stack, using the

idea of push and pop. Or using simple arithmetic. The latter one appealed to me more as it seems more convenient. The idea of the arithmetic algorithm is that when there is a left parenthesis, then increment a counter by one. If there is a right parenthesis, then decrement the counter by one. If the counter ever has negative value, then there must be a unmatching parenthesis as negative value shows that there are more right parenthesis then left parenthesis. The counter needs to be 0 at the end otherwise a non-zero value indicates unmatching parenthesis.

5.6 list implementation

to be continued...

6 Related Work

to be continued...

- Racket
- Clojure
- Clisp

References

References

[1] Steve Yegge.
`http://steve-yegge.blogspot.com/2007/06/rich-programmer-food.html`
.

[2] Tutorials Point: Lisp Tutorial
`https://www.tutorialspoint.com/lisp/index.htm`

[interpreter] avid Evans /Introduction to Computing: Explorations in Language, Logic, and Machines, Chapter 11 Interpreter]
`http://computingbook.org/Interpreters.pdf`

[3] Peter Norvig
http://norvig.com/lispy.html

[4] Let's build a simple interpreter
https://ruslanspivak.com/lsbasi-part1/

[5] A simple interpreter from scratch in Python
http://jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python--p