Lisp in the 21st Century: A Study of Paul Graham's Bel Programming Language

Ryan Foo, supervised by Olivier Danvy
Yale-NUS College, Mathematical, Computational and Statistical Sciences

Abstract

Lisp is a programming language that ignited a revolution from numerical computing to symbolic computing.

We are interested in studying its impact on programming languages today.

We present a brief history of Lisp and explain the concepts of an axiomatic phase and an implementation phase in relation to programming language design. We apply these concepts to understand Paul Graham's new dialect of Lisp, Bel.

Using Python, we implement a properly tail-recursive interpreter for a subset of Bel, which we title Babybel. We provide a REPL and a test suite for Babybel, as well as an analysis of sample programs.

What is Lisp?

LISP is one of the oldest programming languages still in use today, next to FORTRAN.

Lisp was formulated by John McCarthy at MIT. [1] McCarthy showed that one could build a Turing-complete language that operated on symbols using a few operators and a notation for denoting functions inspired by Alonzo Church's lambda calculus. [2]

Lisp is characterized by:
(1) computing with symbolic expressions, rather than numbers.
(2) representation of symbolic expressions and other information by lists in memory
(3) A small set of selector and constructor operations, expressed as functions.

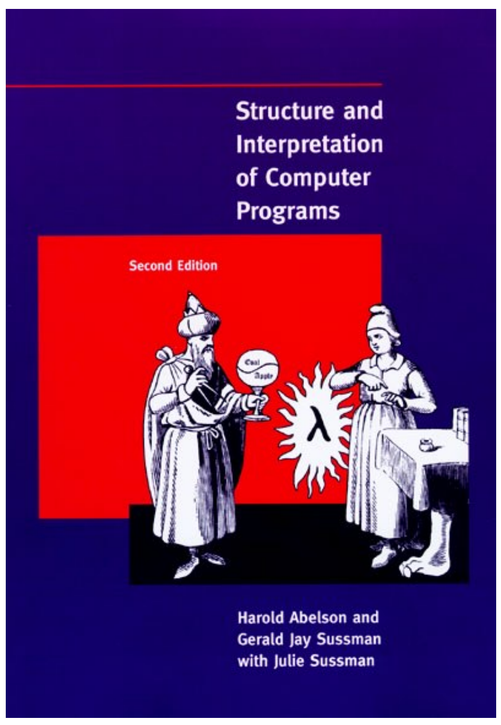
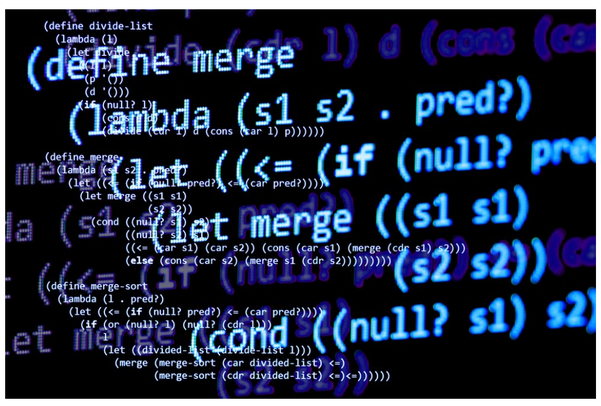


Figure 1: The cover of SICP, used to teach CS at MIT.

Lisp's many forms, and Bel

Lisp comes in many forms, but often looks like this.



The reason it looks like Lisp ("Lots of Silly Parentheses") is because of all its parentheses, which give a uniform syntax, and allow programmers to easily add new features. [3]

That flexibility and malleability allows Lisp programmers to write features extremely fast. [4] Paul Graham speculated that Lisp was successful because of its prolonged development phase, as opposed to implementation. They tried to prolong it even more, which resulted in Bel. We implement a subset of Bel, called Babybel, after the cheese.

Interpreters and Readers

The apparatus to run a programming language using an interpreter consists of a reader and an interpreter. In Lisp, the interpreter has an "evaluator", as it takes Lisp expressions and evaluates them, returning you the result.

The main body of work consists of implementing a reader and interpreter for Babybel.

Eval, or the evaluation function, takes a Bel expression and evaluates it accordingly.

Every program that a programmer writes will either operate on atoms, pairs, or conforms to a special form. The power of Lisp is that because the evaluator is so tightly bounded and axiomatically expressed, one can morph Lisp to have any language features they desire by extending the language to fit their own needs, such that they can then export that language and run it on any computer.

We implemented a properly tail-recursive Babybel evaluator that obeys left-to-right call-by-value, and we can test it using programs of interest.

Programs of Interest

We test our interpreter on Church Numbers, Fibonacci, and Y Combinators.

We build up Church Numbers, which test our environments. A church encoded natural number is either the number 0, or it is the successor of another church encoded natural number. As such, it is a recursively defined data structure.

We implement addition, multiplication and exponentiation on church numbers in Babybel.

We implement the reverse function on lists, as well as the Fibonacci function on Numbers.

We also experimented with recursion by self-application using Y Combinators for all the above programs.

Discussion

This project has enriched me as a computer scientist and programmer in a substantive way.

My claim after having completed this capstone thesis is that Lisp could still prove a powerful educational tool for computer scientists -- if only to demonstrate the relative malleability of code and data, and how these are manipulated at a symbolic level.

Failing anything else, making an earnest effort to learn Lisp will naturally encourage young computer scientists to think about recursive definitions of data structures such as lists as means to manipulate data.

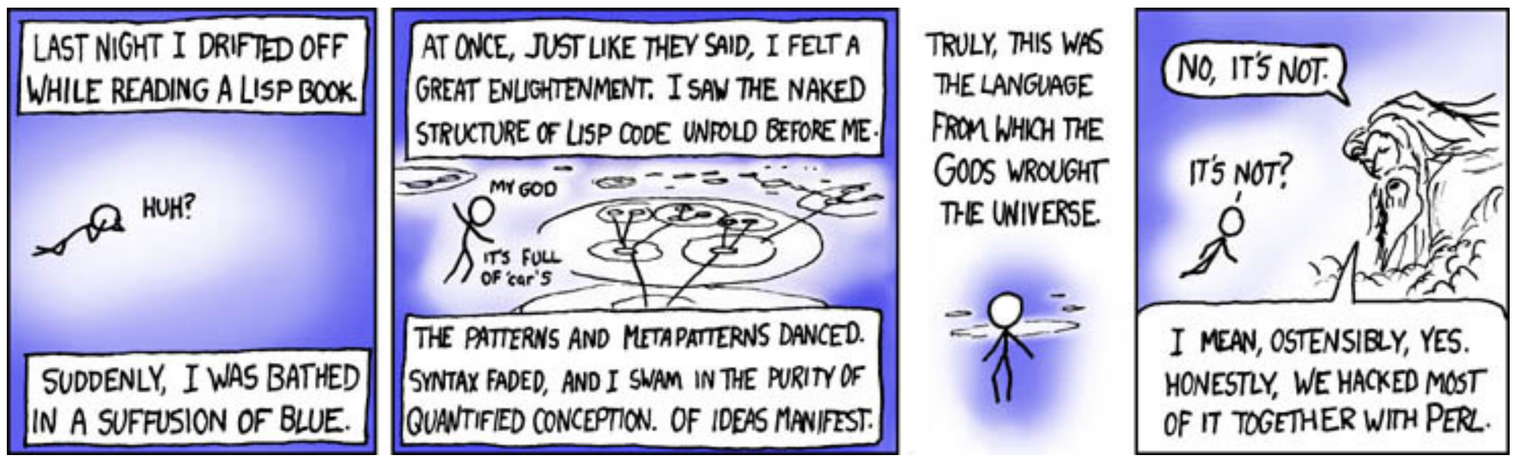


Figure 2: an xkcd comic on Lisp.

Contact

Ryan Foo
Email: ryanfoo@u.yale-nus.edu.sg
Website: https://ryanfoo.com
Phone: (+65) 84990557

References

1. John McCarthy, "Recursive Functions of Symbolic Expressions And Their Computation By Machine, Part 1".
2. Alonzo Church, "The Calculi of Lambda-Conversion".
3. Sean Champ, "Lost Ina Seaof Parentheses".
4. Paul Graham, "Beating the Averages".