Proposal for an independent study on the design of (mostly functional) languages

(a.k.a., a case for studying Scheme and *SICP*)

Jonathan Lam
to Prof. Sable

January 15, 2021

(I make a lot of references to the various resources in the section Readings to motivate LISP and *SICP*, so feel free to check that out first.)

1 Summary

This is a report intended to motivate an independent study on the design of a language, using Lisp as the object of consideration due to its metaprogramming capabilities.

2 Motivation (and responses to concerns)

My technical experience of the past roughly eight years comprises that of a: polyglot programmer; full-stack developer; (somewhat) dev-ops intern; computer center operator; programming mentor; founder of various high-school programming clubs; freelance web developer; computer hardware hobbyist; collector of old server hardware; independent driver developer; *nix power user; and, most recently student in deep learning and artificial intelligence. Through these roles I've become fairly convinced that the quality of one's code is paramount to how well a program performs; how performant, scalable, and maintainable the source tree is; and how easy it is for others to understand and contribute to a project. My concern is that, having viewed many peers' code via mentoring or group projects (or my coworkers' code when I was an intern), and having reflected on my own code from the past, that good coding quality standards are rarely taught nor enforced in many situations where a deadline must be met (particularly susceptible are academic projects and scientific computing). I wish for a more formal practice; to be able to think regularly in terms of better design principles and meaningfully-structured code.

The second issue is that I would like to pursue some topic in the space of operating systems, compilers, or language design: something in the infrastruc-

ture of the programming stack. I feel that my work has been saturated with an interest in the metaprocess of building programs; I always want to know how the libraries, the languages that comprise those libraries, and the systems that host those languages are built. In other words, I want to understand the series of abstractions, from low-level computing to the design of practical high-level libraries. In the end, I hope to perform research in this field for a graduate degree, but I am not sure what in this somewhat-large space on which to focus.

The intersection of these two topics is *Structure and Interpretation of Computer Programs* (*SICP*). The forward and prefaces to the book nicely declares its intent; here is an excerpt from the prologue by Alan Perlis:

> Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs. Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure – we call them idioms – and learn to combine them into larger structures using organizational techniques of proven value. These techniques are treated at length in this book, and understanding them is essential to participation in the Promethean enterprise called programming. More than anything else, the uncovering and mastery of powerful organizational techniques accelerates our ability to create large, significant programs.

That is, *SICP* is a book about abstraction as a building block of building practical software. Not only that, but the last two chapters specifically focus on designing a new LISP interpreter as an example of a large useful program built with the programming practices emphasized in the earlier parts of the book. The book uses LISP as the object of focus due to its powerful metaprogramming capabilities.

My goal is to turn this into an independent study over the summer so I can have a mentor to ease me through the teachings of this book, to have someone to discuss inquiries with, and to keep me focused. The ultimate goal is that this, along with the related exercises and readings, will help me explore into the field of language design principles so that I can smoothly transition into a research project by the fall semester.

I don't think I need to elaborate on why LISP such a fascinating language, both in its history and its structure, but I want to defend the argument that it (and *SICP*) can still be the language (and book) of choice today in an academic setting.

2.1 Modern irrelevance – Choice of book and language

The largest concern during our meeting last semester was that both LISP and *SICP* are old, and its study may not be the most fruitful. This issue seems

critical, given that many CS programs use languages that are more modern and relevant to practical programming, such as C++, Java, and Python; even the courses that have traditionally used LISP (and SICP) to demonstrate some fundamentals of programming (most notably CS61A at Berkeley and 6.001 at MIT) have switched to Python-based curricula. Similarly, *How to Design Programs* (*HtDP*) was written to address some of the pedagogical concerns about *SICP*. Similarly, it is concerning that many of the reviews of *SICP* come from over a decade ago.

However, I would like to dispel these general concerns. First of all, there are innumerable sources that praise *SICP* for its timeless and language-agnostic ability to change the way a programmer thinks about his code from the building blocks available in the language. With what little experience I have in Scheme, I truly feel this is the case – while barebones LISP cannot compete with high-performance languages (e.g., FORTRAN, C, C++), domain-specific languages (e.g., SQL), or domain-specific libraries (e.g., TensorFlow), there are many appraisals of LISP as teaching you how you can build abstractions in software using the building blocks those languages and libraries have to offer you.

To argue about the relevance of *SICP*, I hope that the fact that they are still being offered at major universities (e.g., 6.037 at MIT), albeit not as the introductory course, speaks for itself. (Similarly, Berkeley's 61A teaches Python in the spirit of *SICP*, and its official course name is still "Structure and Interpretations of Computer Programs.") I also believe that the reasons why these prestigious schools stopped teaching with LISP in their introductory courses (see "Programming by Poking: Why MIT stopped teaching SICP") says something about the changing nature of what it means to work in a CS-related job (i.e., now, it is about "poking" around software libraries rather than really doing the engineering from the most basic building blocks), but my goal is more similar to the original ideal: to be able to understand how to manipulate programming languages well enough from their most basic building blocks by means of abstraction. And that is exactly what *SICP* is about.

Despite its old age, LISP is also still innovating: just looking at Scheme, for example, we have the evolving standard. Currently, seven versions of the Scheme standard (currently at R6RS and R7RS) have been released. The language offers several features that have yet to become a part of any other mainstream language, such as first-class continuations or hygienic macros, and its macro system is still unrivaled among mainstream language (the only other languages I've heard it's comparable to are metaprogramming languages such as MetaLua). There are several Scheme variants that are actively maintained and optimized, such as Chez Scheme by Cisco; other LISP variants, such as Common Lisp and Clojure, have major industry support (and the latter is a JVM language, and thus is as up-to-date as the JVM platform is). Performance-wise, many LISP compilers and interpreters have gotten to the point where performance loss compared to high-performance languages like C is negligible, and most variants of LISP beat out other high-level interpreted languages such as Python (at least plain Python without its BLAS or CUDA integrations). (And for many projects, the computing power of most devices is more than sufficient

for any modern language.) Along with the general fact that you can easily build abstractions upon abstractions (according to "Lisp Is Too Powerful", it "allows people to invent their own little worlds") means that using an efficient declarative language like LISP is a way to prevent accruing technical debt.

I know that some of these claims are weak, but the original concern that *SICP* and *LISP* are out-of-date are also based on pure speculation. With a good deal of Internet searching I was unable to find an example of a modern language feature that LISP was incapable of doing, outside of the typical pitfalls of declarative languages (e.g., explicit memory management, array-based data structures such as hashtables, static typing, etc.), and was unable to find any major modern innovation in functional programming that LISP specifically cannot perform. However, to be complete, I have linked *The Journal of Functional Programming* as a possible resource for this independent study, which could be useful for contemporary research (but this may not be very relevant to LISP or the main task at hand).

As an aside, I mentioned (*HtDP*) was a textbook intended to address some of the academic shortcomings of *SICP*, but I believe that many of those changes are actually in conflict with what I intend to achieve. While I have not read "The Structure and Interpretation of the Computer Science Curriculum" (the paper describing the differences between the books) in detail, from a quick perusal it seems that *HtDP* was intended to be a more student-friendly introduction, with a more gradual learning curve and explicit teachings. From what I understand, it also does not go into all the fuss of developing an interpreter as an ultimate goal as *SICP* does, but rather focuses on the more basic design aspects. However, since *SICP* explicitly goes into the details of building an interpreter; since I have a basic knowledge in LISP; and since I am not averse to the challenging problems (rather, I believe that struggling through these problems slowly are key to learning a topic well), I believe that the classic *SICP* will be the better textbook to achieve my goal.

2.2 Devil's advocate: why *not* a more modern language?

Rather than defend LISP, I now present an opposing view: why *not* use a language like Python, JavaScript, or Scala, which are undoubtedly more relevant nowadays? What does LISP have to offer that the others don't?

This concern was brought up in our meeting last semester, and my response was that LISP is known as the "programmable programmer's language," and I mumbled something about its homoiconicity, and that you could easily extend the syntax of LISP. But your rebuttal is that you can't change the structure of LISP, any more than you can't change the structure of another language like C or Python. Plus, if we're talking about macros, C already has macros – what makes LISP any different or better?

I spent some time trying to clarify this via some Internet searches, many of which are in the Readings section below. My new answer to this is that the "programmable nature" of LISP is due to a combination of its homoiconic nature and its macros. The homoiconic nature means that the parsing of LISP

code is trivial due to its simple, recursive syntax that "can be summarized on a business card" (see "Lisp vs. Haskell") and very clearly corresponds to an AST. This is as opposed to many other languages, which have a plethora of inconsistent syntaxes to indicate different kinds of operations, which gives concision at the cost of terribly-inconvenient parsing. As a result of this, many languages that do have macros (recall that macros are functions which transforms source code; in addition to introducing syntactic sugar, they can also change control flow and thus introduce control (syntactic) primitives and otherwise) only have *lexical* macros, e.g., C's token-replacement preprocessor, which is nice for simple replacements but nothing more than that. LISP's *syntactic* macros, on the other hand, are well-integrated into the language and can perform arbitrary transformations (i.e., can run arbitrary logic on the input code to produce the output code, which is capable of infinitely many transformations and not simply text replacement), as well as provide features such as hygienic macros that are intractable (if not impossible) with lexical macros.

The fact that LISP has these syntactic macros makes it an ideal candidate for a language to build abstractions out of. No, we can't change the barebones syntax of LISP (i.e., parentheses, spaces, and tokens must follow the correct syntax), but we can define new syntactic primitives (e.g., control flow, text replacement, data-as-code, etc.) that allow us to extend the language. (If this is unclear, there are a number of resources in the Readings section which do a better job explaining this than I do here.) This undoubtedly was a key reason that the authors of *SICP* chose LISP, and I can see it as an excellent choice for experimenting with different language constructs and styles.

2.3 Relationship with functional languages?

From the video call earlier, it seemed like your impression of the book (and this independent study) was that it might be a study of functional languages – this is not true. I believe you mentioned that you might consider teaching a course on functional programming in the future at Cooper, which is an endeavor I admire as a good step towards creating better programmers – but that is not my goal here. While Lisp is (primarily) functional, that is neither its defining characteristic nor the reason why it is so powerful.

I gave a talk about Lisp last semester in an IEEExACM-organized educational session (rough transcription) because I love the language and hope to indoctrinate as many people as I can. In the original flyer. In the event poster, I called the event "An introduction to Scheme Lisp and Functional Programming," but when I was generating the presentation I was wondering if this was what I really wanted to convey. Functional programming indeed is something that I don't believe many Cooper students are adequately exposed to (the main culprit being firstly that C and Python being taught in terms in data structures (which are mostly imperatively-defined, see "Disadvantages of purely functional languages"), and secondly that scientific programming doesn't at all promote non-imperative-style coding practices), but it didn't seem to me that it was the selling feature of Lisp. After all, many modern languages are

multi-paradigm now (e.g., modern high-level dynamic programming languages like Python, Scala, JavaScript, Ruby, and even recent editions of traditionally-imperative languages like C++ and Java), so this is not at all impressive anymore, even if LISP was the pioneer in many of the topics in functional programming. Rather, I believe that the fact that Lisp's powerful syntax inherently follow a recursive, expression-based, and therefore declarative nature (since its inception in McCarthy's original paper) is a byproduct of its metaprogramming capabilities, and should be treated as such. As mentioned earlier, I have included *The Journal of Functional Programming* as a possible resource for this class, which should more than suffice innovations in functional programming, but this study is orthogonal to LISP (as it should be).

For a study of functional programming, being forced to work in a purely-functional language such as Haskell or Prolog (the latter of which is a Lisp) is probably a better choice. To address the first concern about modernness, there are some resources in the Readings section that illustrate that Haskell is still relevant and performant, despite being function and high-level.

2.4 Lack of knowledge of LISP for mentorship

You voiced your concern that your last experience with LISP was many years ago, and may not have enough experience with the language to be able to advise me if the independent study were to focus on it. However, my view as a polyglot programmer is that that fact should never really be a problem. Also, I firmly believe that having a second person to discuss ideas with, even if they aren't designated experts in the field, is incredibly useful. Perhaps discussions with someone who is the opposite of an expert – an inquisitive, but otherwise clueless on the matter – is the best way to learn, because then more naïve questions are posed and answered, and you simply double the brainpower. This is a fairly general and idealistic pedagogy, but I find discussions with both informed and uninformed people very useful when self-learning any material.

Of course, you (Prof. Sable) are not generally naïve when it comes to programming languages or the principles of data structures and algorithms (and I like to believe that I am also knowledgeable to some degree), so and this peripheral knowledge should accelerate the rate of learning.

2.5 CL vs. Scheme (vs. Others)

This is related to the concerns about the modernness and usefulness of Scheme, which is such a minimal language that it doesn't have many claims to fame outside of academia (and principally, outside of the realm of *SICP*). I remember you suggested that, instead of Scheme, we could use Common Lisp, which has many more libraries and is thus more "practical." However, I would like to argue this point on three grounds:

Firstly, that there is more than enough mental material to fill a semester (or two, or three) with purely academic exercises in Scheme, using only the exercises in *SICP* or additional exercises (e.g., from *The [Little/Intermediate/Seasoned]*

Schemer). Satisfactorily learning Common Lisp, a much more featured language with larger industry support, very well might be overwhelming for a semester

Secondly, that the ideas in *SICP* are largely language-agnostic (given that other general-purposes languages are less-capable of syntactic abstraction but more featured with their default syntax and libraries). Of course, modern vocational programming or scientific computing is highly language- and library-specific, and doesn't require the kind of analytical, problem-solving skillset that *SICP* offers. Perhaps that makes Scheme less appealing to students who simply want a quick job with their EE/CS degree, but again, that is not the goal for me here.

Thirdly, the fact that Scheme has not gained widespread appeal and is not useful for practical coding is probably largely true, but I have personally found it to be useful and somewhat widespread. In my casual programming experience, I have seen Scheme (mostly Guile Scheme due to its GNU integration) used in various useful projects, such as for xbindkeys (keyboard-to-macro mappings in Linux) and emacs lisp. Similarly, mainstream learning resources and coding platforms for Scheme is not at all limited: an excellent tool I have used in the past to learn new languages, exercism.io, includes a track for (Chez and Guile) Scheme. And for school and hobby projects (except perhaps for high-performance computing, such as deep learning and scientific computing that needs intimate control over memory), I've found Scheme to be plenty useful, e.g., for use in ECE469 Artificial Intelligence and MA352 Discrete Mathematics. I'm sure that I would have used Scheme in many other courses had I been familiar with it earlier.

2.6 Overlap with ECE466 (Compilers)

Knowing that *SICP* builds up towards writing a Lisp interpreter might make it seem to have significant overlap with a compilers course. However, a cursory look at the ECE466 course webpage indicates that the topics covered by this course are very different than the goals of *SICP*. While the compilers class shows how to build a "practical compiler," e.g., ASTs, parsing, control flow optimizations, etc., *SICP* is designed to illustrate how to build more complex applications by abstracting the building blocks of a fully-featured low-level language. In other words, it is not the end goal of Lisp to build a programming language compiler or interpreter, but it is a very illustrative example that shows the power of metaprogramming; this is reinforced by the view in "Short explanation of last two chapters in SICP." Furthermore, the methods with which *SICP* attempts to build a interpreter are fundamentally different from those of compilers, as the building blocks are very different: parsing and memory management are already implicit in Lisp, whereas any practical compiler design is lower-level and would need to take this into consideration. All in all, I hope that both ECE466 and this study will aid in selecting a research topic.

3 (Preliminary) (Tentative) Timeline

I believe that following the book should offer more than enough material for a course semester. As many testimonies of the book say, *SICP* is a tome that takes time to read and digest, and its exercises are not trivial. Even at prestigious universities such as Berkeley and MIT (at MIT it is taught by the original authors of the book), this is offered as a semester-long course (and the authors state in the forward that it is more material than what can be covered in a typical semester), and is used as the textbook for graduate-level courses on symbolic programming as well (see MIT's 6.945 in the Materials). We can follow a schedule similar to that posted on MIT or Berkeley's course webpages, and use the video lectures from MIT. Extra exercises may be pulled from *The [Little/Intermediate/Seasoned] Schemer*.

However, to satisfy the need to feel up-to-date, perhaps additional readings should be completed every week (in the tradition of Cooper's ECE472 Deep Learning course, perhaps a set of 2-5 weekly readings from *The Journal of Functional Programming*), or some of the other scholarly articles or books on Lisp listed below, may serve as additional reading.

I have already covered the first chapter of the book on my own, and have a working knowledge of the basics of LISP through school projects, so I believe we can dive straight into the second chapter. If we plan to cover the latter four chapters of the book for a roughly 13-week agenda for the summer semester, a possible weekly agenda for the independent study might look like:

1. §2.1-2.4: Symbolic data, data abstraction, data hierarchy and closures
2. §2.5: Systems with generic operations
3. Continuation of §2.5: Implement a symbolic algebra system (perhaps with macros and symbolic differentiation?)
4. §3.1-3.2: Local state and the evaluation model
5. §3.3-3.4: Mutable data and concurrency
6. §3.5: Streams (and perhaps a project to implement streams)
7. §4.1: The metacircular evaluator
8. §4.2-4.3: Lazy evaluation, nondeterministic computing
9. §4.4: Logic programming: (perhaps a project to implement a Prolog-like language)
10. §5.1-5.2: Register machines
11. §5.3: Allocation and garbage collection
12. §5.3: The explicit-control evaluator
13. §5.4: Compilation

4 (Preliminary) (Tentative) Deliverables and Assessment

Assignments from *SICP* and *The [Little/Intermediate/Seasoned] Schemer* could be used as weekly assignments, as well as reading reports. The final project may be a Lisp interpreter written in Lisp, following the trajectory of the book.

5 Materials

- *Structure and Interpretation of Computer Programs* (*SICP*)

  a.k.a. the "Wizard book," and the subject of discussion of most of this article

- Structure and Interpretation of Computer Programs (University of California, Berkeley CS61A)

  Python in the tradition of *SICP* (Associated materials: Composing Programs)

- Structure and Interpretation of Computer Programs (MIT 6.001)

  Original course taught by the authors of the book (stopped being taught in early 2000's)

- Structure and Interpretation of Computer Programs (MIT 6.037)

  Newer course taught by the authors of the book (2019)

- Adventures in Advanced Symbolic Programming (MIT 6.945)

  Graduate course on symbolic programming, uses *SICP* as its textbook

- Journal of Functional Programming

  If we want to stay up to date, can choose a few articles out of this every week. Unfortunately may need to pay to subscribe

- *On Lisp*

  A whole book on Lisp by Paul Graham (who is described in the article "How Lisp became God's Own Programming Language" below)

- *The Common Lisp Cookbook*

  The title is self-explanatory

- R6RS

  Latest widely-adopted Scheme standard, including standard libraries

- Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

  Original paper by McCarthy on S-expressions, which formed the basis for Lisp; Lisp was implemented shortly after this paper using these principles

- *The [Little/Intermediate/Seasoned] Schemer*

  A series of problems, separated by difficulty, designed to illustrate various aspects of the Scheme language

- *Let over Lambda: 50 years of Lisp*

  A guidebook to Common Lisp, including more advanced features that you wouldn't normally find elsewhere; first six chapters are available online for free

- *How to Design Programs* (HtDP)

  A book mostly in the tradition of *SICP*, but specifically aimed at its pedagogical shortcomings (e.g., HtDP is intended to require less domain knowledge and be more explicit in its principles); The Structure and Interpretation of the Computer Science Curriculum addresses these pedagogical differences. Second edition came out last year (2018)

- *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* (PAIP)

  Chapters include logic programming and other practical problems in Lisp (e.g., a general purpose solver (GPS), ELIZA, a computer algebra system (CAS) and other symbolic mathematics)

6 Readings to motivate LISP and *SICP*

I realize I skimmed quite a few Q&A posts, articles, books, etc. related to "Why Lisp?" and especially "Why Lisp macros (as opposed to macros in other languages or ordinary first-class functions)?" because macros are what really make Lisp so useful. Here is a brief summary of some articles.

- "Lisp: A language for stratified design"

  1987 research paper by the authors of *SICP* about the usefulness of Lisp as a way to show a complete view on programming techniques through abstraction

- "Scheme vs. Python"

  The title is misleading; this is written by the professor for 61A at Berkeley, which uses *SICP*, and it defends the use of the book and Scheme rather than Python. (Since this post was written, I believe 61A has switched to Python (most likely due to a reason similar to "Programming by Poking: Why MIT stopped teaching SICP"), but it still attempts to follow the tradition of *SICP*)

- "Is there an expiration date for well regarded, but old books on programming?"

No, most of the concepts in these books are fundamental (and therefore timeless), just as the basics of mathematics (e.g., calculus) hasn't changed for centuries; also many concepts are language-agnostic

- "Programming by Poking: Why MIT stopped teaching SICP"

  (The title is misleading: MIT still has courses that use and teach *SICP*, but the iconic 6.001 course that was the introduction to all EECS undergrads at MIT was abolished in favor of an introductory course in Python, 6.0001)

  Unfortunately, much of the real work programmers do nowadays is "programming by poking": messing around with pieces of code (high-level languages and libraries) to just get it to work, people don't have to build systems from the bottom up nowadays, so they switched the introductory course to python

  Personally, I find this reason very sad and reflects the state of most programmers nowadays. But this is contradictory to my motivation (see section above)

- "Notes on Structure and Interpretation of Computer Programs"

  Mentions a few notable topics that *SICP* teaches: homoiconicity, how to write an interpreter, register machines, etc.

- "Why is Haskell (GHC) so darn fast?"

  A discussion on the speed of modern functional languages, and how they achieve their impressive real-world speed (as opposed to the common mode of thinking that functional languages are typically more inefficient)

- "Disadvantages of purely functional languages"

  Written by a pronounced naysayer of Haskell; mostly describes the fact that some common data structures cannot be implemented in a functional way

- "Lisp vs. Haskell"

  Speaks to Lisp's metaprogramming abilities and simplicity ((almost) everything is built out of seven core functions)

- "What is so great about Lisp?"

  General discussion on why so many people admire Lisp. Mostly emphasizes on the simple syntax, which leads naturally to a metacircular interpreter, as well as bringing up a host of interesting readings and Greenspun's tenth rule

  Also mentions a few disadvantages, such as the relatively small number of libraries, difficulty to pick up, and being forced to use dynamic typing (whether this is a boon or poison depends on the application). However, none of these affect learning the fundamentals in an academic context

- "Lisp Is Too Powerful"

  The first sentence is illustrative: "I think one of the problems with Lisp is that it is too powerful. It has so much meta ability that it allows people to invent their own little worlds, and it takes a while to figure out each person's little world"

- "What makes Lisp a programmable programming language? Why does this matter? What are the advantages of being one?"

  Describes what it means when people describe Lisp as a "programmable programming language" – namely, that it can

  Also describes how Haskell doesn't achieve these goals, which shows that (even purely) functional programming does not amount to the metaprogramming capability that Lisp offers. Is probably more true for less-functional languages like Python

- "How Lisp became God's Own Programming Language"

  Perhaps my favorite articles on "Why Lisp" that feels complete in multiple ways. Goes into a history of Lisp (including its motivations and impacts). The last paragraph, about the revival of Lisp after the AI winter and its lingering impact on "modern" languages like Ruby and Python, is especially satisfying.

- "What makes Lisp macros so special?"

  Many good answers here. E.g., can implement list comprehension or infix notation in Lisp, write code replacements (which is more advanced than the C's token-expansion, write something like C#'s LINQ notation; a macro is a function that takes in some source code and outputs source code, but it can perform logic on the expansion as well, including processing the arguments to the macro), changing execution order, the fact that Lisp's homoiconic nature makes macros much more useful than other languages with macros ("in C, you would have to write a custom preprocessor [which would probably qualify as a sufficiently complicated C program]")

- "What are Lisp macros good for, anyway?"

  Illustrative example of a timing macro in Lisp, and comparison to purely functional approaches (i.e., without macros) in Java and Python. Illustrates how macros allow for a different evaluation pattern for its arguments, i.e., rewriting the evaluation path, which cannot be done simply with functions (i.e., we can create new control primitives just like builtin control flow operators)

- "What can you do with Lisp macros that you can't do with first-class functions?"

  Relevant to the previous resource. Useful comment on an answer: "To abstract and hopefully clarify a bit, Lisp macros allow you to control

whether (and when) arguments are evaluated; functions do not. With a function, the arguments are always evaluated when the function is invoked. With a macro, the arguments are passed to the macro as ordinary data. The macro code then decides itself if and when they should be evaluated. This is why you can't write reverse_function without macros: as a function, the argument would be evaluated (and cause an error) before the function body gets a chance to rewrite it."

This also goes over some of the basic ideas, e.g., that macros do expansion at compile-time whereas first-class functions create closures at runtime and have the extra overhead of a function call; this has the added benefit of greater efficiency (similar to macros in any other language)

- Paul Graham's essays (mostly on programming languages)

  Haven't read these yet, but a good number are on Lisp. Paul Graham is a co-founder of ycombinator and helped to revive Lisp after the AI winter with these essays (for more history, see "How Lisp became God's Own Programming Language")

- "Automata via Macros"

  Research paper aiming to bring light to the relevance and need of Lisp-like macros, since they are not widely used in more modern languages (e.g., Java and C). Also focuses on the need for tail-call optimization for macros to be useful and efficient.

  Also has a very succinct list of overview of uses of macros: "providing cosmetics; introducing binding constructs; implementing 'control operators', i.e., ones that alter the order of evaluation; defining data languages"

- "Why do people say Lisp has true macros versus C/C++ macros?"

  A question dedicated to the comparison of Lisp-style macros to those that exist in other languages. The main point is that since the language is homoiconic, you have the power of the entire Lisp language at your hands to transform the input code to the output code, whereas in C/C++ the language is much more complicated and only does token replacements. Similarly, the C preprocessor is very decoupled from the rest of the language and doesn't have any logical utilities (other than token replacement), whereas the Lisp macro system is very intimately tied into its language, as you can perform arbitrarily-complex Lisp code to perform the code transformation

- "Lisp Macro"

  Similar arguments as in the previous articles, with some personal testimonies. Includes a lot of Q&A about use cases (including when not to use macros)

- "Why aren't macros included in most modern programming languages?"

Yet another explanation of macros and how they are different in C and Lisp. While C macros are lexical and carry no syntactic meaning, Lisp macros are syntactic and can be treated as regular code. (E.g., one implication is the use of hygienic macros in Lisp, which safely do not shadow variables in the outside scope)

- "What makes macros possible in lisp dialects but not in other languages?"

  An answer to this question with some technical explanation

- "Short explanation of the last two chapters of SICP"

  Explains that the final two chapters are not a repeat of a compilers course, but show that an interpreter is just a computer program that takes data as input, and thus is a good example of data abstractions

- "SICP - Worth it?"

  Relevant quote: "It's a subtle book. As others say, you do things in it you probably won't ever do again, in a language you'll probably never use again... But it can change (and improve) the way you write and understand programs, in any language."