1. Preface

This book is a tour through LISP9, a LISP system with a bytecode compiler and an abstract machine for interpreting the code. LISP9 is an interactive system and takes care of a lot of things that may appear trivial from the outside, but require careful design and planning internally. It is a solid, reasonably fast, and portable version of LISP. This text describes every small detail of its implementation in great detail and without cutting corners.

The text shares a considerable intersection with *Scheme 9 from Empty Space* (S9fES), an earlier book of mine, but it concentrates on abstract machine code generation and interpretation, where S9fES employed a simple tree walker. On the other hand, S9fES also covers unbounded integer (“bignum”) arithmetic and algorithms for computations on real numbers, which this book completely omits. The scope is narrower in the present work, but the depth is greater.

The LISP9 language has been inspired a lot by Scheme. R3RS Scheme was one of the first LISP systems I studied and only a short time later its successor, R4RS, appeared. The *Revised*^{4} *Report on the Algorithmic Language Scheme* was a beautiful document and, in my opinion, the pinnacle of the development of Scheme. Later I got involved in the process that lead toward R6RS and, ironically, it is the only R^{n}RS document that lists my name in the acknowledgments. Ironically, because it was the document that made me turn my back on Scheme and, for a long time, on computing in general.

In the end, this turned out to be a good thing, because it made me realize what computer science and programming are to me. They are an expression of *love* and *beauty*. Since I spent some time at university, I have always loved to understand and explain formal systems. Formulating interconnected facts and rules in the most precise and transparent way that I can think of started to become a passion of mine. It is the reason why I write programs and why I write textbooks: to make fundamental characteristics of complex systems as obvious and clear as possible.

Unfortunately this is a lost art these days. Abelson and Sussman's epic statement [SICP1985]

Programs are meant to be read by humans
and only incidentally for computers to execute

which should be an unshakable truth for everybody who ventures to write and understand computer programs, is by many considered to be a liability today. Code must be released quickly, must be fast at any cost, must "create value" and "provide user experience". Indeed Kurlansky may have been right when he suggested that all people who fight for something will in the end adopt the traits they detest most in their enemies [Kurlansky2006]. Hacker culture has become everything that its members disdained in the May of its age: a club of entrepreneurs, marketers, business people.

Even at the risk of marginalizing myself, I refuse to be part of this new "hacker" culture. Silence, introspection, casually browsing and enjoying your own code, pride in your creation, but also, and probably most importantly, humbleness in the face of complexity are values without which the art of programming is deteriorating to craft, then trade, and then obscurity. It is no longer an art at all, but a business, and I will have no part in it!

This background is very well reflected by the LISP9 language and its implementation. The language is minimalistic and terse: a tool for the creation of art and inspiration instead of the conglomeration of massive and hyper-complex structures. Its source code attempts to be a novel written in plain language rather than an encyclopedia, and this book is its secondary literature.

The LISP9 language is so much like Scheme that many programs can probably be translated between them by substituting keywords and special objects. Its semantics should look suspiciously familiar to everybody who knows Scheme. However, its keywords are terse and it revives some traits that have their roots in more traditional LISP.

The terseness of keywords serves a pragmatic purpose: it makes the code brief and, most importantly, allows to keep lines of code limited to a number of characters that is suitable for printing in a 6”×9” book. This is particularly helpful in the age of electronic books, because it makes program code fit comfortably in small displays.

The LISP9 implementation follows an idiomatic coding style that may need some getting used to. Here is a summary:

- function names are verbs, only sometimes nouns
- function names are short, but expressive
- global variable names are short nouns
- local variable names are single-character placeholders

At the lexical level in C code:

- CONSTANTs are all-upper-case
- Global variables are capitalized
- everything else is all-lower-case

In LISP code global variables wear `*earmuffs*`.

In the prose, keywords and global variables are typeset in **boldface** characters and local variables are typeset in *roman italics*.

Part of this style certainly runs counter to the belief that everything must be named with “obvious”, “descriptive”, and “immediately comprehensible” names. I find this explicit style utterly confusing and obtuse. Compare

```
(defun (reverse-and-concatenate
         base-list
         appended-list)
  (if (null base-list)
      appended-list
      (reverse-and-concatenate
        (cdr base-list)
        (cons (car base-list)
              appended-list))))
```

with

```
(defun (reconc a b)
  (if (null a)
      b
      (reconc (cdr a)
              (cons (car a) b))))
```

If you prefer the former style, you will have to adjust quite a bit to the style practiced in this book, but I predict that this will not be as hard as it might seem to be, and the benefit is substantial. White space is as important to code as silence is to music. Without them music is noise and code is an impenetrable wall of text.

In the concrete example, **reconc** ("reverse and concatenate") is a well-known LISP idiom, just like **strcmp** ("string-compare") is a well-known C idiom. Languages thrive on the use of idioms. They make the language brief and clear at the same time. Their only flip side is that they require some accommodation by those who are not used to them. But then you are already using terms like "loading a program" or "doing a function call" without thinking about them too much. You will get used to new idioms as easily as you got used to those.

About variables: given that you understand what **reconc** does, what will a and b be? Which one will be reversed and concatenated to which one? I would say, it is obvious from the context, and the use of "more readable" names would only make the code opaque and hard to comprehend.

This books is intended to be read front-to-back and maybe twice: once focusing on the prose and once on the code. The prose is terse, because I do not want to waste your time, so if your mind drifts for a moment, you probably *will* miss something. All concepts are introduced in a strict bottom-up order, as far as possible, so everything builds on top of previously introduced topics when going through the text in the intended order.

To avoid all issues that one or the other reader may have with the various licenses out there, the LISP9 source code is placed in the public domain or, in legislations that do not have a concept like the public domain, distributed under the Creative Commons Zero (CC0) License. This means that you, the reader, have the ultimate freedom to do whatever you want to do with the code. So does everybody else. Freedom cannot be possessed. A copy of the CC0 can be found here:

`https://creativecommons.org/publicdomain/zero/1.0`

The complete code reproduced in this book can be found in machine-readable form on my homepage,

`http://t3x.org`

Enjoy the tour through the LISP9 system! Do not strain when things get complicated! Take a break and remember the joy of being alive!

Nils M Holm, May 2019