Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Introduction

Macros

Lisp's core occupies some kind of local optimum in the space of programming languages. —Modest words from John McCarthy, inventor of lisp

This is a book about programming *macros* in lisp. Unlike most programming books that only give them a cursory overview, this book is a series of tutorials and examples designed to get you programming sophisticated macros as efficiently and quickly as possible. Mastering macros is the final step to graduating from an intermediate lisp programmer to a lisp professional.

Macros are the single greatest advantage that lisp has as a programming language and the single greatest advantage of any programming language. With them you can do things that you simply cannot do in other languages. Because macros can be used to transform lisp into other programming languages and back, programmers who gain experience with them discover that all other languages are just skins on top of lisp. This is the *big deal*. Lisp is special because programming with it is actually programing at a higher level. Where most languages invent and enforce syntactic and semantic rules, lisp is general and malleable. With lisp, you make the rules.

Lisp has a richer, deeper history than all other programming languages. Some of the best and brightest computer scientists throughout our field's brief existence have worked very hard to make it the most powerful and general programming language ever. Lisp also enjoys a number of concise standards, multiple excellent open-source implementations, and more macro conveniences than any other programming language. This book uses only COMMON LISP[ANSI-CL][CLTL2] but many of the ideas are trivially portable to other lisps like Scheme[R5RS]. That said, hopefully this book will convince you that if you want to write macros, COMMON LISP is the lisp to use. While different types of lisp are excellent for other purposes, COMMON LISP is deservedly the tool of choice for the macro professional.

The designers of COMMON LISP have done an excellent job in designing a programming language *right*. Especially after considering implementation quality, COMMON LISP is, with surprisingly few reservations, by far the best programming environment available to current-day programmers. As a programmer you can almost always count on COMMON LISP to have done it the way it ought be done. While it's true the designers and implementors have done it the right way, some feel that they forgot to describe to us just why it's right. To many outsiders COMMON LISP just looks like an enormous collection of strange features, and so are turned off, moving to a more immediately gratifying language—doomed to never experience the true power of macros. Although it is not its primary purpose, this book can be used as a tour of many of the best features in the amazing language that is COMMON LISP. Most languages are designed to be easy to implement; COMMON LISP is designed to be powerful to program. I sincerely hope the creators of COMMON LISP appreciate this book as one of the most complete and accessible treatments of the language's advanced macro features, and also as an enjoyable drop in the ocean of topics that is the macro.

Macros have, not by accident, almost as much history as lisp itself, being invented in 1963 by Timothy Hart[MACRO-DEFINITIONS]. However, macros are still not used to the fullest possible extent by most lisp programmers and are not used at all by all other programmers. This has always been a conundrum for advanced lispers. Since macros are so great, why doesn't everybody use them all the time? While it's true that the smartest, most determined programmers always end up at lisp macros, few start their programming careers there. Understanding why macros are so great requires understanding what lisp has that other languages don't. It requires an understanding of other, less powerful languages. Sadly, most programmers lose the will to learn after they have mastered a few other languages and never make it close to understanding what a macro is or how to take advantage of one. But the top percentile of programmers in any language are always forced to learn some sort of way to write programs that write programs: macros. Because it is the best language for writing macros, the smartest and most determined and most curious programmers always end up at lisp.

Although the top-percentile of programmers is necessarily a small number, as the overall programming population grows so does the number of top-percentile programmers. The programming world sees few examples of the power of macros and understands far fewer, but this is changing. Because of the productivity multiplication that can be achieved through macros, the *age of the macro* is coming, whether the world is ready or not. This book aims to be a base-line preparation for the inevitable future: a world of macros. Be prepared.

The conventional wisdom surrounding macros is to use them only when necessary because they can be difficult to understand, contain extremely subtle bugs, and limit you in possibly surprising ways if you think of everything as functions. These aren't defects in the lisp macro system itself but instead are traits of macro programming in general. As with any technology, the more powerful the tool,

the more ways there are to misuse it. And, as far as programming constructs go, lisp macros are the most powerful tool.

An interesting parallel to learning macros in lisp is that of learning pointers in the C programming language. Most beginning C programmers are able to quickly pick up most of the language. Functions, types, variables, arithmetic expressions: all have parallels in previous intellectual experiences beginners might have had, from elementary school maths to experimenting with simpler programming languages. But most novice C programmers hit a brick wall when they encounter pointers.

Pointers are second nature to experienced C programmers, most of whom consider their complete understanding necessary for the proper use of C. Because pointers are so fundamental, most experienced C programmers would not advise limits on their use for stylistic or learning purposes. Despite this, many C novices feel pointers are an unnecessary complication and avoid their use, resulting in the *FORTRAN in any language* symptom where valuable language features are neglected. The disease is ignorance of the language's features, not poor programming style. Once the features are fully understood, the correct styles are obvious. An auxiliary theme of this book, one that applies to any programming language, is that in programming, style is not something to pursue directly. Style is necessary only where understanding is missing[1] .

[1] *A corollary to this is that sometimes the only way to effectively use something you don't understand is to copy styles observed*

Like C pointers, the macro is a feature of lisp that is often poorly understood, the wisdom on its proper use being very distributed and idealised. If when considering macros you find yourself relying on stylistic aphorisms like

Macros change the syntax of lisp code.

Macros work on the parse tree of your program.

Only use macros when a function won't do.

you are probably missing the big picture when it comes to macro programming. That is what this book hopes to fix.

There are very few good references or tutorials on macro construction. Paul Graham's *On Lisp*[ON-LISP] is one of the exceptions. Every word of *On Lisp* is required reading for anyone interested in macros. *On Lisp* and Graham's other writings were the most important inspirations for the creation of the book you are reading now. Thanks to Paul Graham and other lisp writers, the power that macros provide programmers is widely discussed, yet is unfortunately still widely misunderstood. Despite the wisdom regarding macro programming that can be gleaned from a simple perusal of *On Lisp*, few programmers make the connection between the macro and their real-life programming problems. While *On Lisp* will show you the different types of macros, this book will show you how to use them.

Macro writing is a reflective and iterative process. All complex macros come from simpler macros, often through a long series of improvement-test cycles. What's more, recognising where to apply macros is an acquired skill that comes directly from writing them. When you write a program, you, as a conscious human, are following a system and a process whether you are aware of it or not. Every programmer has a conceptual model of how programming tools work and the creation of code comes as a direct, logical result of this. Once an intelligent programmer begins to think of the act of programming as a logical procedure, the logical next step is for this process to benefit from automation itself. After all, programmers are trained to do exactly this: automate processes.

The crucial first step to understanding macros is to recognise that without careful planning and lots of effort, large portions of any programs will have redundant patterns and inflexible abstractions littered throughout. This can be seen in almost any large software project as duplicated code or as code that is needlessly complex because the right abstractions weren't available to its authors. The effective use of macros entails recognising these patterns and abstractions, and then creating *code to help you code*. It is not enough to understand how to write macros; a professional lisp programmer needs to know why to write macros.

C programmers who are new to lisp often make the mistake of assuming that the primary purpose of a macro is to improve the efficiency of code at run-time[2] *C programmers make this mistake because they are used to a "macro system" that is good for little else.* . While macros are often very useful for this task, by far the most common use of a macro is to make the job of programming a desired application easier. Because large portions of the patterns in most programs are redundantly copied and the generality of their abstractions not fully exploited, properly designed macros can enable programming on literally new planes of expression. Where other languages are rigid and specific, lisp is fluid and generic.

This book is not an introduction to lisp. The topics and material are aimed at professional programmers of non-lisp languages who are curious as to what macros have to offer, and at intermediate lisp students who are ready to really learn what makes lisp special. Basic to intermediate knowledge of lisp programming is assumed, but a deep understanding of closures and macros is not.

This book is also not about theory. All examples involve working, usable code that can help improve your programming, today and now. This book is about using advanced programming techniques to help you program better. In contrast to many other programming books that deliberately use a simple programming style in an attempt to improve accessibility, this book takes the view that the best approach to teaching programming is full utilisation of the language. Although many of the provided code samples use esoteric features of COMMON LISP, such potentially unfamiliar features are described as they are used. For calibration, if you have read and understood[3] *Not necessarily agreed with, of course.* everything in chapter 2, Closures and chapter 3, Macro Basics, for the purposes of this book you can consider yourself past the intermediate stage of lisp understanding.

Part of lisp is discovering things yourself and this book will not deprive you of that. Be warned that this book moves more quickly than most, more quickly than you might be used to. To understand some of the code in this book you may need to consult additional COMMON LISP tutorials or references. After we cover the basics we will move directly into explaining some of the most advanced macro research to-date, much of which borders a large, unexplored gray-area of intellectual terrain. As does all advanced macro programming, this book focuses heavily on *combinations* of macros. This topic has a frightening reputation and is well understood by few, if any, programmers. Combinations of macros represent the most vast and fertile area of research in programming languages today. Academia has squeezed out most of the interesting results from types, objects, and prolog-style logic, but macro programming remains a huge, gaping black hole. Nobody really knows what lies beyond. All we know is that, yes, it is complicated and frightening and currently appears boundless in potential. Unlike too many other programming ideas, the macro is neither an academic concept for churning out useless theoretical publications, nor an empty enterprise software buzzword. Macros are a hacker's best friend. Macros let you program smarter, not harder. Most programmers who come to understand macros decide they never again want to program without them.

While most lisp books are written to make lisp more popular, I am completely unconcerned with lisp's day-to-day public appeal. Lisp isn't going away. I would be perfectly happy if I could continue to use lisp as a *secret weapon* for the remainder of my programming career. If this book has only one purpose, it is to inspire the study and research of macros, just as I have been inspired by them in *On Lisp*. I hope readers of this book might also be so inspired that some day I might enjoy even better lisp macro tools and even more interesting lisp macro books.

Still in awe of lisp's power,

your humble author,

Doug Hoyte

U-Language

Since discussing macros involves discussing discussion itself, we need to be very clear about the conventions we are adopting for this book. What I am writing right now, as conveyed to you by what you are reading and interpreting, is itself a system of expression worth formalising and analysing.

Nobody has understood this better than Haskell Curry, the author of *Foundations Of Mathematical Logic*[FOUNDATIONS]. Curry, because he was not only trying to formalise ideas, but also the very expression of ideas, found it necessary to abstract this concept of a communicative language between writer and reader. He called it the U-Language.

Every investigation, including the present one, has to be communicated from one

person to another by means of language. It is expedient to begin our study by calling attention to this obvious fact, by giving a name to the language being used, and by being explicit about a few of its features. We shall call the language being used the U-Language. [...] There would be no point in calling attention to it, if it were not for the fact that language is more intimately related to our job than of most others.

Throughout this book we will introduce key new concepts or points that otherwise deserve emphasis in *this special font*. When referencing special forms, functions, macros, and other identifiers found in a program, either presented or foreign, we will use **this special font** (notice that some words have multiple meanings, for example **lambda** the COMMON LISP macro versus lambda the concept; **let** the special form versus a list that is a let form).

EXAMPLE-PROGRAM-LISTING

```
(defun example-program-listing ()
  '(this is
     (a (program
          (listing)))))
```

In this book new pieces of code are introduced in the form of *program listings*. Code that is designed for re-use, or for an example of proper implementation, is presented as in the definition of our function **example-program-listing**. But sometimes we wish to demonstrate the use of a bit of code or just want to discuss properties of some expressions without departing the flow of the written text[4] *And this is a foot-note, a relevant but concise departure from the main text.* . In those cases, the code, or example uses of the code, will appear like so:

```
(this is
  (demonstration code))
```

Much writing that teaches programming makes heavy use of isolated, contrived examples to illustrate a point but forgets to tie it in with reality. This book's examples try to be as minimal and direct as possible in order to illustrate the big-picture programming ideas currently being explained. Some writing tries to hide being boring by using cute, quirky identifier names or skin-deep analogies in its examples. Our examples serve only to illustrate ideas. That said, above all this book tries not to take itself (or anything) too seriously. There is humour here, the difference is that you need to look for it.

Because of lisp's interactive nature, the results of evaluating a simple expression can often convey more than the equivalent quantity of U-Language. In such cases, this is how we will show the output from a COMMON LISP Read Evaluate Print Loop (called the *REPL*):

```
* (this is
    (the expression
      (to evaluate)))
```

```
THIS-IS-THE-RESULT
```

Notice how the text we enter is in lower-case but the text returned from lisp is in upper-case. This is a feature of COMMON LISP that allows us to easily scan a REPL print-out and know which expressions we entered versus which were printed out by lisp. More precisely, this feature lets us quickly scan any lisp form that contains symbols—in any file or on any screen—and instantly know whether it has yet been processed by the lisp reader. Also notice that the asterisk character (*) represents a prompt. This character is ideal because it can't be confused with a balanced character and because of its high pixel count that makes it stand out clearly when scanning a REPL session.

Writing complicated lisp macros is an *iterative* process. Nobody sits down and hammers out a page-long macro in the cavalier style common to programs in other languages. This is partly because lisp code contains much more information per page than most other languages and also partly because lisp technique encourages programmers to grow their programs: refining them in a series of enhancements dictated by the needs of the application.

This book distinguishes types of lisp, like COMMON LISP and Scheme, from the more abstract notion of lisp the building material. Another important distinction is made between lisp programming languages and non-lisp programming languages. Sometimes we need to talk about non-lisp languages and, to make as few enemies as possible, would like to avoid picking on any language in particular. To do so, we resort to the following unusual definition:

A language without lisp macros is a Blub.

The U-language word Blub comes from an essay by Paul Graham, *Beating the Averages*[BEATING-AVGS], where Blub is a hypothetical language used to highlight the fact that lisp is not like other languages: lisp is different. Blub is characterised by infix syntax, annoying type systems, and crippled object systems but its only unifying trait is its lack of lisp macros. Blub terminology is useful to us because sometimes the easiest way to understand an advanced macro technique is to consider why the technique is impossible in Blub. The purpose of Blub terminology is not to poke fun at non-lisp languages[5] *There will be a little bit of fun.*
.

ITERATIVE-PROCESS-EXAMPLE

```
(defun example-function% () ; first try
  t)

(defun example-function%% () ; second try
  t)

(defun example-function () ; got it!
  t)
```

In order to illustrate the iterative process of macro creation, this book adopts

the convention where the percent (%) character is appended to the names of functions and macros whose definitions are incomplete or are yet to be improved upon in some other way. Multiple revisions can result in multiple % characters on the end of a name before we settle on the final version with no % characters.

Macros are described in Curry's terminology as *meta-programming*. A meta-program is a program with the sole purpose of enabling a programmer to better write programs. Although meta-programming is adopted to various extents in all programming languages, no language adopts it as completely as lisp. In no other language is the programmer required to write code in such a way to convenience meta-programming techniques. This is why lisp programs look *weird* to non-lisp programmers: how lisp code is expressed is a direct consequence of its meta-programming needs. As this book attempts to describe, this design decision of lisp—writing meta-programs in lisp itself—is what gives lisp the stunning productivity advantages that it does. However, because we create meta-programs in lisp, we must keep in mind that meta programming is different from U-Language specification. We can discuss meta-languages from different perspectives, including other meta-languages, but there is only one U-Language. Curry makes this clear for his system as well:

We can continue to form hierarchies of languages with any number of levels. However, no matter how many levels there are, the U-Language will be the highest level: if there are two levels, it will be the meta-language; if there are three levels, it will be the meta-meta-language; and so on. Thus the terms U-Language and meta-language must be kept distinct.

This is a book about lisp, of course, and lisp's logic system is very different than that described by Curry so we will adopt few other conventions from his work. But Curry's contributions to logic and meta-programming continue to inspire us to this day. Not only because of his profound insights regarding symbolic quotation, but also his beautifully phrased and executed U-Language.

The Lisp Utility

On Lisp is one of those books that you either understand or you don't understand. You either adore it or you fear it. Starting with its very title, *On Lisp* is about creating programming abstractions which are layers *on top of lisp*. After we've created these abstractions we are free to create more programming abstractions which are successive layers on earlier abstractions.

In almost any language worth using, large portions of the language's functionality is implemented with the language itself; Blub languages usually have extensive standard libraries written in Blub. When even implementors don't want to program in the target language, you probably won't want to either.

But even after considering the standard libraries of other languages, lisp is different. In the sense that other languages are composed of primitives, lisp is composed of meta-primitives. Once macros are standardised, as in COMMON

LISP, the rest of the language can be *boot-strapped* up from essentially nothing. While most languages just try to give a flexible enough set of these primitives, lisp gives a meta-programming system that allows any and all sorts of primitives. Another way to think about it is that lisp does away with the concept of primitives altogether. In lisp, the meta-programming system doesn't stop at any so-called primitives. It is possible, in fact desired, for these macro programming techniques used to build the language to continue on up into the user application. Even applications written by the highest-level of users are still macro layers on the lisp onion, growing through iterations.

In this light, there being primitives in a language at all is a problem. Any time there is a primitive, there is a barrier, a non-orthogonality, in the design of the system. Sometimes, of course, this is warranted. Most programmers have no problem treating individual machine code instructions as primitives for their C or lisp compilers to handle. But lisp users demand control over nearly everything else. No other languages are, with respect to the control given to the programmer, as complete as lisp.

Heeding the advice of *On Lisp*, the book you are currently reading was itself designed as another layer on the onion. In the same sense that programs are layered on other programs, this book is layered on *On Lisp*. It is the central theme of Graham's book: well-designed *utilities* can, when combined, work together to give a greater than the sum of the parts productivity advantage. This section describes a collection of useful utilities from *On Lisp* and elsewhere.

MKSTR-SYMB

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))
```

Symb, layered upon **mkstr**, is a general way of creating symbols. Since symbols can be referenced by any arbitrary string, and creating symbols programmatically is so useful, **symb** is an essential utility for macro programming and is used heavily throughout this book.

GROUP

```
(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
                   (rec rest (cons
                              (subseq source 0 n)
                              acc))
```

```
                (nreverse
                  (cons source acc))))))
    (if source (rec source nil) nil)))
```

Group is another utility that consistently pops up when writing macros. Part of this is because of the need to mirror operators like COMMON LISP's **setf** and **psetf** that already group arguments, and part of it is because grouping is often the best way to structure related data. Since we use this functionality so often, it makes sense to make the abstraction as general as possible. Graham's **group** will group by any provided grouping amount, specified by the parameter **n**. In cases like **setf**, where the arguments are grouped into pairs, **n** is 2.

FLATTEN

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec
                        (car x)
                        (rec (cdr x) acc))))))
    (rec x nil)))
```

Flatten is one of the most important utilities in *On Lisp*. Given an arbitrarily nested list structure, **flatten** will return a new list containing all the atoms reachable through that list structure. If we think of the list structure as being a tree, **flatten** will return a list of all the leaves in the tree. If that tree represents lisp code, by checking for the presence of certain objects in an expression, **flatten** accomplishes a sort of *code-walking*, a recurring theme throughout this book.

FACT-AND-CHOOSE

```
(defun fact (x)
  (if (= x 0)
    1
    (* x (fact (- x 1)))))

(defun choose (n r)
  (/ (fact n)
     (fact (- n r))
     (fact r)))
```

Fact and **choose** are the obvious implementations of the factorial and binomial coefficient functions.

License

Because I believe the concepts behind the code presented in this book are as fundamental as physical observations or mathematical proofs, even if I wanted to

I don't believe I could claim their ownership. For that reason you are basically free to do whatever you want with the code from this book. Here is the very liberal license distributed with the code:

```
;; This is the source code for the book
;; _Let_Over_Lambda_ by Doug Hoyte.
;; This code is (C) 2002-2008, Doug Hoyte.
;;
;; You are free to use, modify, and re-distribute
;; this code however you want, except that any
;; modifications must be clearly indicated before
;; re-distribution. There is no warranty,
;; expressed nor implied.
;;
;; Attribution of this code to me, Doug Hoyte, is
;; appreciated but not necessary. If you find the
;; code useful, or would like documentation,
;; please consider buying the book!
```


Thanks

Brian Hoyte, Nancy Holmes, Rosalie Holmes, Ian, Alex, all the rest of my family; syke, madness, fyodor, cyb0rg/asm, theclone, blackheart, d00tz, rt, magma, nummish, zhivago, defrost; Mike Conroy, Sylvia Russell, Alan Paeth, Rob McArthur, Sylvie Desjardins, John McCarthy, Paul Graham, Donald Knuth, Leo Brodie, Bruce Schneier, Richard Stallman, Edi Weitz, Peter Norvig, Peter Seibel, Christian Queinnec, Keith Bostic, John Gamble; the designers and creators of COMMON LISP, especially Guy Steele, Richard Gabriel, and Kent Pitman, the developers and maintainers of CMUCL/SBCL, CLISP, OpenBSD, GNU/Linux.

Special thanks to Ian Hoyte for the cover design and Leo Brodie for the back-cover cartoon.

This book is dedicated to everyone who loves programming.