Expressing Problems in Their Natural Language

Robert S. Smith

28 March 2020

Preface What is Lisp, where does it come from, and what is it useful for? This is an expedient account of Lisp by way of real examples geared toward programmers and scientists. The examples showcase real Common Lisp code, and it's not expected the unfamiliar reader understand every intricacy. I decided not to pursue much detail about the grand, rich, and dramatic history of Lisp—which could fill an entertaining book unto itself. My writing is loosely based on a presentation of the same title I gave at Rigetti Computing in 2016, in which I justified the growing use of Common Lisp for critical projects. This essay is not a tutorial, it's a pitch.

Whence Lisp came

Lisp is an old language. It was created by John McCarthy in the late 1950s, and later published in a paper *Recursive Functions of Symbolic Expressions and their Computation by Machine* in 1960. Lisp originally was an idea than it was a standardized language, and from that idea came a family of languages. Informally and without further context, any of these languages could be called "Lisp".

Even today, new Lisp languages are invented. For example, Clojure (2007) is a JVM-compatible Lisp focusing on immutable data structures, Hy (2013) is an attempt to cast the semantics of Python into the syntax of Lisp, and Hackett (2017) is a Lisp with Haskell's type system. The idea of Lisp is perhaps so cornerstone to the development of computer science that we will see Lisp languages continue to be invented for as long as computers exist.

Lisp has had a monumental influence on the design and implementation of modern programming languages. Interpretation, object-orientation, garbage collection, arbitrary precision arithmetic, anonymous functions[1], the interactive command loop[2], symbolic programming, and many other ideas were either invented, standardized, or popularized by Lisp. Being the progenitor for important ideas doesn't automatically make Lisp relevant to the modern programmer, but it does make it historically interesting. Fortunately, Lisp has more to offer.

[1] Lisp programmers highly prefer to call them λ-functions!
[2] Also known as a "REPL".

A common Lisp

Lisp represents an idea, and from that idea many languages blossomed. At some point in the 1980s, for a variety of reasons, a com-

mittee called X3J13 banded together under the auspices of ANSI to form a common Lisp, a standardized language that was featureful enough to satisfy everyone: Lisp programmers, Lisp vendors, and users of Lisp applications. After some decade of time and hundreds of thousands of dollars, born was the document *ANSI INCITS 226-1994*, the standard[3] for the language *ANSI Common Lisp*—Common Lisp for short.

Common Lisp is a language that doesn't bear a single programming paradigm. It natively supports imperative, functional, and object-oriented styles of programming right out of the gate, and is capable of allowing additional paradigms atop by way of libraries. I don't find the laundry list of features very helpful in understanding the value of Common Lisp, but I do think it's worth a few notes about implementation.

Common Lisp has a plethora of implementations. I broadly classify them into three sets.

Open-source Steel Bank Common Lisp, Clozure Common Lisp, GNU CLISP[4]

Commercial Allegro Common Lisp by Franz Inc., LispWorks by LispWorks Ltd.

Special Clasp (C++ compatible), Embeddable Common Lisp (C compatible), Armed Bear Common Lisp (Java compatible), JSCL (JavaScript compatible)

There have been *many* more Common Lisp implementations over the past 40 years than these listed above, however, many of them have evolved into one of the above, or have otherwise become abandoned or defunct.

What's special about these implementations is that they all conform to the ANSI standard, which means if you write (ANSI-conforming) code in one implementation, it will generally run without modification in another implementation. This, in turn, means that the Common Lisp code that folks have been writing for 40 years *still* runs! It also means that you are able to reap the benefits of any implementation relatively freely.

[3] A language standard is often seen as a drag. Anybody who has peeked at the C++ standard, for instance, has perhaps vomited. They're useful "legal" documents, not useful programmer documents. However, the ANSI standard for Common Lisp is different. It is written in a friendly yet rigorous manner that makes it useful as an authoritative reference for the language. I myself reference the electronic version, the *Common Lisp HyperSpec* every day.

[4] In the Common Lisp community, CLISP *specifically* refers to the eponymous implementation, and is *not* shorthand for "Common Lisp".

What is Lisp made of?

Just about all of Common Lisp is made up of the same form:

```
(operator operand1 operand2 ...)
```

The `operator` generally denotes some operation or verb, and the `operand`s generally represent values. For instance, `(sqrt (+ 1 (* 2 3)))`

is $\sqrt{1+2\cdot 3}$, and `(if (= x 1) 4 5)` is "if $x = 1$ then return 4 otherwise return 5."

This isn't so different from writing

$$f(x_1, x_2, \ldots) \Longleftrightarrow (f\ x_1\ x_2\ \ldots),$$

though we aren't just referring to math, but a broader programming context (where, say, x_1 might be a string).

Besides function calls, other structures in Common Lisp follow this form. For example, `let` is an operator that introduces variables. A phrase like "let $n = 5$ and $\theta = \pi$ in $\sin n\theta$" might be translated[5] like so:

[5] It doesn't matter how we break the code across lines or space it out.

```
(let ((n     5)
      (theta pi))
  (sin (* n theta)))
```

Functions are defined with the operator `defun`. The operands are the function's name, argument list, and the expression that defines the function. Can you guess what this code does[6]?

[6] Hint: It has to do with the ongoing pandemic.

```
(defun f (days)
  (* 1.565 (expt 1.1194 days)))
```

Common Lisp doesn't just deal with numbers. You can also deal with lists of things. Appropriately so, `list` is an operator[7] which makes lists. Just as `(+ 2 2)` gives us back[8] `4`, the expression

[7] Unlike `defun` or `let`, the operator `list` is an ordinary function.

[8] The Lisp term for "gives us back" is "*evaluates*".

```
(list 1 2 3)
```

gives back the list `(1 2 3)`. Similarly,

```
(list (list 1 (* 1 1))
      (list 2 (* 2 2))
      (list 3 (* 3 3)))
```

gives back the list of lists `((1 1) (2 4) (3 9))`. We could get the first sublist by calling the function appropriately named `first`.

One special data type in Common Lisp not found in many other languages is the *symbol*. It's just a named thing that equals only itself. We can create or refer to a symbol by writing a single quote followed by a name, so the expression `'X` evaluates to the symbol `X`. Symbols are often used where "enums" or strings-as-keys might be used in other languages. Symbols are also used to do tasks in symbolic computation, like symbolic algebra or pattern matching, which is why Lisp was so popular in the study of classical artificial intelligence back in the 1970s and 1980s.

This concludes our introduction to the syntax of Common Lisp, but before we move on, I'd like to chip at the title of this essay. With symbols and lists, we can start writing expressions that send shivers down our spines[9].

[9] When I was around 16 years old, I was reading John Allen's *Anatomy of Lisp*, I certainly got them. It was my moment of "enlightenment" with Lisp when I discovered expressions written in Lisp could produce Lisp itself! Don't feel bad if you don't have the same histrionic reaction.

```
(list 'sqrt 2)
```

will evaluate to `(sqrt 2)`. It does *not* evaluate to `1.41421`, because `list` constructs lists and the quote makes symbols. But did we just make a list, or did we just make code? Well, both. We could evaluate that code with another built-in operator called `eval`. The code

```
(eval (list 'sqrt 2))
```

itself evaluates to `1.4142135`.

From this, we gather that Common Lisp code is actually just a bunch of nested lists. In fact, we could *create* the code for our previous $\sin n\theta$ by writing

```
(list 'let (list (list 'n 5)
                 (list 'theta 'pi))
      (list 'sin (list '* 'n 'theta)))
```

Yes, `'*` will evaluate to a symbol named[10] `*`. And yes, if we evaluate the above expression, we will get[11]

```
(let ((n 5) (theta pi)) (sin (* n theta)))
```

back.

Now, it may seem like a nice party trick, but what if some of the functions we wrote *in* Common Lisp could also *generate* Common Lisp? At minimum, that might allow us to save us some typing of arduous or repetitive programs. What could it mean beyond that though? If it were easy to write programs that, in turn, write programs, we would not only be writing *in* Common Lisp, but we would be programming Common Lisp *itself*. For that reason, Common Lisp has been called the "programmable programming language". Before we see examples of this, we should understand a bit more practically how we interact with Common Lisp.

[10] Nobody said symbols had to be named by letters only!

[11] This time, written on one line instead of being splayed across three.

Interacting with Lisp

The mental model of Common Lisp is quite different from other languages, and it all centers around the concept of an image. The *image* is the state of your code and running program. When you compile a function, it gets installed into the image. When you recompile a function, it gets replaced in the image. You interact with the image via the REPL, a command loop[12] that *Reads* code, *Evaluates* it, *Prints* the result, and *Loops* back to reading. A program which is saved as an executable is merely a binary representation of the Lisp image.

The "installation" of functions into the image suggests an incremental nature to the image's development. Common Lisp code typically isn't executed wholesale[13]; instead, it's loaded into an image,

[12] You can actually implement a primitive REPL easily in Lisp. It's just `(loop (print (eval (read))))`.

[13] Now, of course when the program is "all done" and ready to be run by a user, all of this business about images goes away. The user just sees a statically linked executable like any other.

and the programmer calls functions within that image.

Complementary to "incremental" is "interactive". Usually the programmer is actively making inquiries about the image or modifications to it. By default, in Common Lisp, it's possible to: load files into the image, compile functions, evaluate expressions, inspect objects, look at documentation, debug programs and functions, disassemble code, profile code, and build executables. With a good compiler and IDE[14], it's possible to hop around the source code, jump to errors, look at efficiency hints, and many other things.

[14] In Common Lisp, these days, "a good IDE" is a euphemism for Emacs and a companion package called SLIME: the Superior Lisp Interaction Mode for Emacs.

A short interactive session

Let's suppose a programmer is writing a geometry application, and they're starting from scratch. At the REPL[15], they're writing code to test out their ideas before "committing" to them in their larger program. They start with a few routines for right triangles.

[15] The REPL's prompt is denoted by >.

```
> (defun hypot (x y) (+ (* x x) (* y y)))
> (defun perimeter (x y) (+ x y (hypot x y)))
```

This looks relatively straightforward. The programmer installed into their image a function `hypot` to calculate the hypotenuse length of a right triangle with legs `x` and `y`, as well as a function `perimeter` to compute the perimeter of a right triangle. Everybody knows about the 3-4-5 triangle triangle, so the programmer tests their function out:

```
> (perimeter 3 4)
32
```

Oops, $32 \neq 3+4+5$. What went wrong? Well the programmer didn't compute the hypotenuse correctly. No problem, we can overwrite `hypot` with a corrected version.

```
> (defun hypot (x y) (sqrt (+ (* x x) (* y y))))
WARNING: redefining HYPOT in DEFUN
> (perimeter 3 4)
12.0
```

Notice how we did not need to reinstall the `perimeter` function. Now the programmer can feel confident putting their definitions of `hypot` and `perimeter` into a file called `geo.lisp`. In the future, the programmer can simply[16]

[16] As a matter of fact, using a proper IDE, you could do this with a simple key combination, like `Ctrl+l` in Emacs.

```
> (load "geo.lisp")
```

to install all of their geometry routines in one go.

The programmer is happy with their tested routines, but are now concerned about their efficiency. At the REPL, the programmer writes a little benchmark program that calculates `hypot` in a loop.

```
> (defun bench (n)
    (loop :repeat n
```

```
        :for x := (random 1.0)
        :for y := (random 1.0)
        :do (hypot x y)))
```

Running the benchmark with a hundred million loops takes almost 3 seconds and 6.5 billion processor cycles.

```
> (time (bench 100000000))
Evaluation took:
  2.749 seconds of real time
  2.749359 seconds of total run time (2.746153 user, 0.003206 system)
  100.00% CPU
  6,597,326,094 processor cycles
  0 bytes consed
```

How could we improve? The first thing to do is to look at the disassembly of `hypot`.

```
> (disassemble (function hypot))
; disassembly for HYPOT
; Size: 79 bytes. Origin: #x237CA2A5                          ; HYPOT
; A5:       498B5D10         MOV RBX, [R13+16]
; A9:       48895DF8         MOV [RBP-8], RBX
; AD:       488B55F0         MOV RDX, [RBP-16]
; B1:       488B7DF0         MOV RDI, [RBP-16]

;;; [5] (* X X)

; B5:       FF1425C000B021   CALL QWORD PTR [#x21B000C0]      ; GENERIC-*
; BC:       488BDA           MOV RBX, RDX
; BF:       48895DE0         MOV [RBP-32], RBX
; C3:       488B55E8         MOV RDX, [RBP-24]
; C7:       488B7DE8         MOV RDI, [RBP-24]

;;; [6] (* Y Y)

; CB:       FF1425C000B021   CALL QWORD PTR [#x21B000C0]      ; GENERIC-*
; D2:       488BFA           MOV RDI, RDX
; D5:       488B5DE0         MOV RBX, [RBP-32]
; D9:       488BD3           MOV RDX, RBX

;;; [4] (+ (* X X) (* Y Y))

; DC:       FF1425B000B021   CALL QWORD PTR [#x21B000B0]      ; GENERIC-+
; E3:       B902000000       MOV ECX, 2
; E8:       FF7508           PUSH QWORD PTR [RBP+8]

;;; [3] (SQRT (+ (* X X) (* Y Y)))

; EB:       B8029A4520       MOV EAX, #x20459A02              ; #<FDEFN SQRT>
; F0:       FFE0             JMP RAX
; F2:       CC10             INT3 16
```

Even without being an expert at assembly, the programmer sees how each portion of the `hypot` function is getting compiled, and sees that each arithmetic operation is calling a "generic" form that works for any numerical type. So the programmer knows that the `hypot` ought to be redefined specifying the arguments to be single-precision floats[17], and that the function ought to be optimized.

[17] As it stands, `hypot` is generic, and will allow the user to pass in single- and double-precision floats, arbitrary precision integers, rationals, even complex numbers!

```
L> (defun hypot (x y)
     (declare (optimize speed (safety 0) (debug 0))
              (type (single-float (0.0)) x y))
     (sqrt (+ (* x x) (* y y))))
WARNING: redefining HYPOT in DEFUN
```

With this new definition, the programmer inspects the assembly again.

```
> (disassemble (function hypot))
; disassembly for HYPOT
; Size: 49 bytes. Origin: #x23757638                          ; HYPOT
; 38:       F30F59C9         MULSS XMM1, XMM1
; 3C:       F30F59D2         MULSS XMM2, XMM2
; 40:       F30F58D1         ADDSS XMM2, XMM1
; 44:       F30F5AD2         CVTSS2SD XMM2, XMM2
; 48:       660F57C0         XORPD XMM0, XMM0
; 4C:       F20F51C2         SQRTSD XMM0, XMM2
; 50:       F20F5AC0         CVTSD2SS XMM0, XMM0
; 54:       0FC6C0FC         SHUFPS XMM0, XMM0, #4r3330
; 58:       660F7EC2         MOVD EDX, XMM0
; 5C:       48C1E220         SHL RDX, 32
; 60:       80CA19           OR DL, 25
; 63:       488BE5           MOV RSP, RBP
; 66:       F8               CLC
; 67:       5D               POP RBP
; 68:       C3               RET
```

Not only did the function reduce in footprint, it was also optimized to SSE floating point instructions. With this new function in hand, the programmer re-runs the benchmark.

```
> (time (bench 100000000))
Evaluation took:
  1.547 seconds of real time
  1.548603 seconds of total run time (1.547359 user, 0.001244 system)
  100.13% CPU
  3,714,031,260 processor cycles
  0 bytes consed
```

The programmer is satisfied with the 44% improvement in wall-clock time[18]. Again, the programmer will take the new `hypot` function and save it into the file.

[18] In fact, most of the run time is spent on computing the random numbers. A proper benchmark would disentangle the random number generation from the call to `hypot`.

Object-oriented programming

Common Lisp was the first object-oriented programming language to be standardized. Despite pre-dating every popular object-oriented language today, Common Lisp maintains an object system more powerful than them all, so powerful it has its own name: the *Common Lisp Object System* or *CLOS*[19].

[19] People pronounce this "claws", "kloss" (rhyming with "toss"), "close" (pronounced any way), and my personal favorite "see-loss".

At a high level, CLOS is relatively simple. Everything is an object, and every object is a member of some class, which itself sits in a hierarchy of other classes. Simple so far, and not very different than other object systems.

Where CLOS begins to differ is what a class *is*. In Common Lisp, a class definition *only* specifies slots and ways to access those slots. Slots in Common Lisp are called *attributes*, *members*, or *fields* in other languages. Critically, class definitions do *not* include methods. For instance, if we define a "person" as an entity that has a name and an age, we might define it in Common Lisp like so.

```
(defclass person ()
  ((name :initarg :name
         :accessor person-name)
   (age :initarg :age
        :accessor person-age)))
```

The slots `name` and `age` have some options attached to them, such as how to supply that slot to the constructor (the "initarg") and how to access or change the value in the slot (the "accessor"). Of course, we can define this at the REPL which installs it into the image[20]. Making a `person` object is a matter of calling `make-instance`, a sort of universal constructor.

[20] Just like functions, in Common Lisp, even *classes* can be safely redefined! Common Lisp is truly optimized to the bone for incremental and interactive programming.

```
> (make-instance 'person :name "Billy" :age 37)
#<PERSON {100A3ADC33}>
```

As you might imagine, subclasses can be formed and they inherit all of the slots of the parent class. The subclass also remains the usual membership relationships (e.g., every subclass of `person` is still a `person`). Different than a few languages however is that CLOS allows classes to inherit from multiple other classes.

Classes hold data. What can we *do* with them? Completely disjoint from the class definitions are definitions of special kinds of functions called *generic functions*. Generic functions are functions that can work with different argument classes. If we continue our geometry example from section , we could imagine wanting to compute the area of a shape, or render shapes onto different media. So perhaps we should define the generic functions `area` and `render`.

```
(defgeneric area (shape))
(defgeneric render (shape medium))
```

As we can see, there's not actually any logic yet. We've just established in our image that `area` and `render` are generic functions that will be specialized for different shapes and media.

To actually add logic, we define specializations of these generic functions, called *methods*. It's easier to just show an example using two hypothetical classes `rectangle` (which we presume to have a width and height) and `circle` (which we presume to have a radius). First, we define the `area` methods.

```
(defmethod area ((shape rectangle))
  (* (width shape) (height shape)))
```

```
(defmethod area ((shape circle))
  (* pi (expt (radius shape) 2)))
```

We've defined two specializations of the generic function `area` for our two shapes. When the shape is a `rectangle`, the first method is called. When the shape is a `circle`, the second method is called.

That's pretty straightforward. How about `render`? Well we can't draw a shape if we don't know what we are drawing to. The magic of CLOS is that we can specialize methods on more than just one argument. So using the same *generic function*, we can add *methods* to draw rectangles to GUI canvases and to draw circles to PostScript printers. Here, we have hypothetical media classes `gui-canvas` and `postscript-printer`.

```
(defmethod render ((shape rectangle) (medium gui-canvas))
  (let ((w (width shape))
        (h (height shape)))
    (draw-line medium 0 0  w 0)   ; bottom (0,0) - (w,0)
    (draw-line medium 0 0  0 h)   ; left   (0,0) - (0,h)
    (draw-line medium w 0  w h)   ; right  (w,0) - (w,h)
    (draw-line medium 0 h  w h))) ; top    (0,h) - (w,h)

(defmethod render ((shape circle) (medium postscript-printer))
  ;; The postscript command is
  ;;     0 0 <radius> 0 360 arc closepath stroke
  ;; to draw a circle at the origin.
  (let ((stream (printer-stream medium)))
    (format stream "0 0 ~F 0 360 arc closepath stroke~%" (radius shape))))
```

This decoupling of classes and generic functions leads to very extensible and robust code. The class hierarchy is also usually shallower, because there's less of a need for abstract classes[21]. Instead of abstract classes or interfaces, a Common Lisp programmer will define a *protocol*, which is a collection of generic functions one should implement with their classes. We might say that `area` and `render` constitute the "shape protocol".

[21] It's a common pattern in Python to define an abstract base class to establish all of the required methods a subclass ought to implement

Lifting the language to the problem

> "Lisp is the programmable programming language."
>
> — John Foderaro, Co-Founder of Franz Inc., 1991

We've discussed various sundries of Common Lisp, and we are now prepared to discuss what Lisp does better than every other well established programming language.

Let's take a step back from *programming* languages, and consider *languages* as a more general concept. When we are presented with some sort of problem, the solution is often expressed in some language that naturally fits the framework of the problem. The genius of Einstein wasn't that he calculated Newton's laws better than everybody else, it was that he understood the problem of gravitation in an

entirely different framework than Newton. Had he used the language of forces and ordinary vector calculus, Einstein probably would not have succeeded. Einstein even developed his own notation to ease computations in his new framework of understanding the world.

Once you open your eyes to it, you see it everywhere. Almost everybody across every discipline has either adopted or developed some sort of language. Even programmers do it. When you want to check if the string `s` looks like a binary numeral, do you

```
if len(s) == 0:
    return False
else:
    if s[0] == '0':
        return len(s) == 1
    elif s[0] == '1':
        for c in s[1:]:
            if c not in '01':
                return False
        return True
    else:
        return False
```

or do you `re.match('0|1(0|1)*', s)`? Probably the latter; the notation is not only convenient, it also expresses our idea irreducibly. In this Python example, the regex isn't Python-the-language, but it's instead a syntax common across almost all programming languages specifically for expressing a regular language and testing strings' membership within the language.

We have this correspondence between domains and some convenient language use to express ideas in that domain. There's a term for these languages, *domain-specific languages*, and there are countless examples.

Configuration Files JSON, YAML, TOML, INI, etc.

Old-School Calculators RPN[22]

Finite-State Automata Regex strings

Logic Well-formed formulas

Electrical Circuits 2D circuit diagrams

Parsers BNF grammars, parser combinators

Software Builds Makefiles, CMake, `setup.py`, etc.

Western Music Staff notation, letter notation for the 12-tone chromatic scale

General Relativity Tensors and Einstein notation

Quantum Comp. & Field Theory Circuit diagrams, Feynman diagrams

[22] The HP 50g remains one of my most treasured belongings, and I can always calculate faster than my TI-wielding colleagues because of it.

Fermionic Hamiltonians Algebraic combinations of creation and annihilation operators

I would go as far as saying that one of the greatest lessons of mathematics is that the job of solving a problem is made orders of magnitude easier—sometimes made *trivial*—with the right choice of definitions and the right choice of syntax.

When somebody sees Common Lisp for the first time, the quirky prefix notation with parentheses everywhere usually causes a reaction of modest disgust or disappointment. We all know and love the quadratic formula as

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

or perhaps

```
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

and not as

```
(/ (+ (- b) (sqrt (- (expt b 2)
                     (* 4 a c))))
   (* 2 a))
```

as a typical Lisp programmer would write. But the regularity of Lisp serves to be a usable programming language while making as few decisions as possible as to its syntax. Why? Because it seeks to reserve as much syntactic freedom to the programmer, and not the language designers.

As we shall see, Common Lisp is a language that allows users to add new syntax. Generally, this is achieved by providing facilities to tell Lisp how your syntax translates into otherwise ordinary Common Lisp code. These facilities fall under the umbrella of a single concept: the Lisp *macro*[23]. There are three kinds of Common Lisp macros that allow you different levels of flexibility:

Ordinary Macros These allow you to introduce new syntax as long as it follows the `(operator operand)` parentheses convention.

Reader Macros These allow you to introduce completely new *character* syntax to Common Lisp. You can make Common Lisp look like any other by using character macros.

Compiler Macros These allow you to introduce an alternative compile-time meaning to specific function calls[24].

Ordinary macros may not seem like introducing new syntax since we have to stick to the usual parentheses syntax, but it's not true. Common Lisp's own `loop` macro is an example.

[23] Conrad Barski, the author of *The Land of Lisp*, really didn't want to use the word "macro". It's very overloaded with meaning, especially in software engineering. While I will not adopt his convention of calling them SPELs, you should know that a Lisp macro is only tangentially related to any other macro you've encountered, including those offered by C.

[24] We won't talk about these, but they're neat. If you have a function `f`, you can write code which possibly transforms the call sites of `f`. This is valuable for a variety of reasons, including optimization (e.g., partial evaluation).

```
(loop with names = (lookup-people *database*)
      for name in names
      for length = (length name)
      when (or (= 1 length)
               (char= #\Z (char name 0)))
        do (format t "~S is a weird name!~%" name))
```

This almost looks like some imperative programming language[25], but it's actually just an ordinary macro.

[25] It *is* an imperative programming language for expressing complicated loops! It's just presented as an extension to and within Common Lisp.

Macros give programmers something of a superhuman ability. It makes the programming language itself fair game for modification in order to make problem-solving easier.

> Common Lisp is a blank syntactic canvas, and you paint with a palette of macros.

This is the central principle[26] that allows Common Lisp to be linguistically lifted up to "natively" operate in the domain most naturally suited for a task, as opposed to pummeling the task down to the rigid requirements of the programming language. It's important to remember that almost *every* problem is a domain-specific problem.

[26] The honest truth is that Common Lisp is a *mostly* blank canvas. Common Lisp does come with a heap of syntax so that the average programmer has plenty to work with: characters, arrays, rational numbers, complex numbers, floats, file paths, etc. Each of these things *does* have a non-parentheses syntax. But they pose little to no obstruction in the introduction of new syntax.

It's dangerously easy to wax poetic about Lisp—Common or otherwise. In lieu of going much deeper about the implications of problem solving paradigms, I'll switch gears and present some examples. The examples are chosen such that they can fit on a page. More elaborate code would obviously allow for more elaborate examples, a few of which are referenced in passing.

List comprehensions

For all of Common Lisp's fanfare about lists, it doesn't contain a feature present in many modern languages: list comprehensions. They are often more readable shorthand when manipulating sequences. In Common Lisp, we would like to support list comprehensions with a new operator `lc` with the following grammar:

```
<clause>    ::= `for' <var> `in' <expr>     # iterate across a sequence
              | `for' <var> `below' <expr>  # iterate from 0 to N-1
              | `when' <expr>               # keep only when true

<list-comp> ::= `(lc' <expr> <clause>* `)'.
```

With this grammar, we'd like to be able to write functions like the following:

```
(defun cartesian-product (seq1 seq2)
  (lc (list x y) for x in seq1 for y in seq2))

(defun filter (pred seq)
  (lc x for x in seq when (funcall pred x)))
```

```
(defun multiples (n count)
  (lc (* n i) for i below count))
```

The beauty of this hypothetical construction is that any Python, Haskell, or Erlang programmer can immediately read them. The "native" Common Lisp equivalents are a bit more difficult. For instance, without list comprehensions, the first function could be written[27] like so:

[27] Here, we presume each argument can be a sequence of *any* type like lists, one-dimensional arrays, or strings. If we were to restrict ourselves to *only* one of those data types, we could get a considerably more readable implementation.

```
(defun ugly-cartesian-product (seq1 seq2)
  (let ((result nil))
    (map nil (lambda (x1)
               (map nil (lambda (x2)
                          (push (list x1 x2) result)))))
    (nreverse result)))
```

The hypothetical operator `lc` can be realized as an ordinary macro. It fits the bill since it uses the usual parentheses notation. The main work of the macro is to figure out what the clauses are, and to construct the appropriate code for each. The code for a `for-below` clause is a simple iteration, a `for-in` clause is a sequence traversal[28], and a `when` clause is a simple conditional[29]. An efficient implementation of this macro is just 25 lines of code:

[28] Just as in `ugly-cartesian-product`, we can uses Common Lisp's `map` function for traversal across generic sequences.

[29] Common Lisp actually has a `when` operator. It is an ordinary macro atop `if`!

```
(defun parse-clauses (clauses)
  (optima:ematch clauses
    ((list) nil)
    ((list* 'for var 'below n rest)
     (cons (list ':below var n) (parse-clauses rest)))
    ((list* 'for var 'in seq rest)
     (cons (list ':for var seq) (parse-clauses rest)))
    ((list* 'when pred rest)
     (cons (list ':when pred)   (parse-clauses rest)))))

(defmacro lc (expr &rest clauses)
  (alexandria:with-gensyms (collect x result tail)
    (let ((body  `(,collect ,expr))
          (clauses (reverse (parse-clauses clauses))))
      (dolist (clause clauses body)
        (alexandria:destructuring-case clause
          ((:below var n) (setf body `(dotimes (,var ,n) ,body)))
          ((:for var seq) (setf body `(map nil (lambda (,var) ,body) ,seq)))
          ((:when pred)   (setf body `(when ,pred ,body)))))
      `(let* ((,result (cons nil nil))
              (,tail ,result))
         (flet ((,collect (,x)
                  (rplacd ,tail (cons ,x nil))
                  (setf ,tail (cdr ,tail))))
           ,body
           (cdr ,result))))))
```

This code also has a notable feature. In order to parse out clauses, the `parse-clauses` function uses a package called Optima which implements flexible pattern matching as a library. Pattern matching is a feature that's becoming more and more popular in up-and-coming programming languages, but no Common Lisp programmer has had to wait for it in order to benefit.

Moreover, this code is ripe for extension. We support three kinds of clauses, but it would be trivial to add more clauses. For instance, we could imagine a clause that assigns a temporary variable:

```
(lc ... with <var> = <val>)
```

The beauty of the solution is that the list comprehension operator `lc` presents itself to the programmer as any other operator in the language. We have grown the Common Lisp language with new syntax.

Reverse Polish notation

Earlier, we saw how we would write the quadratic formula in Lisp. If I didn't have the quadratic formula, but I did have the coefficients, the most expedient way to calculate it—for me personally—is to write out the *reverse Polish notation* or *RPN* for the computation. RPN is very easy.

$$2 \cdot (1+3) \iff \texttt{2*(1+3)} \iff \texttt{(/ 2 (+ 1 3))} \iff \texttt{2 1 3 + *}.$$

It takes less than ten minutes to become proficient at it. The rule is simple: If the token is not an operation, it is pushed onto a stack. If it is an operation, it pops the arguments off of the stack, applies the operation, and puts the result on the stack. The quadratic formula written in RPN is:

```
b neg b b * 4 a c * * - sqrt + 2 a * /
```

In Common Lisp, we can write an ordinary macro called `rpn` which will translate RPN notation into an equivalent meaning that Lisp understands. If such a macro existed, we could write

```
(defun quadratic (a b c)
  (rpn b neg b b * 4 a c * * - sqrt + 2 a * /))
```

And in fact, we can, in 10 lines of code.

```
(defun neg (x) (- x))

(defmacro rpn (&rest expr)
  (let ((stack (gensym "STACK")))
    `(let ((,stack nil))
       ,@(loop :for token :in expr :collect
           (case token
             ((sqrt neg) `(push (,token (pop ,stack)) ,stack))
             ((+ - * /)  `(push (,token (pop ,stack) (pop ,stack)) ,stack))
             (otherwise  `(push ,token ,stack))))
       (first ,stack))))
```

Note how in our `quadratic` function, the RPN syntax is automatically able to work with local variables, like the function arguments. There was no need to "feed" them in; it just works.

The point of adding RPN to Common Lisp isn't to be able to write obscure quadratic polynomial solvers. The point is that we've added a syntax to Common Lisp that makes a particular task easier, namely writing out calculations swiftly, without reaching out for yet another incompatible tool. If we can add syntax for RPN, then we can also add syntax for infix[30], regaining what many folks feel is lost by using Common Lisp.

[30] Fortunately, we don't need to. Mark Kantrowitz wrote a library in 1993 adding infix syntax to Common Lisp. The library is available as CMU-INFIX, and allows one to write code like `(n & (n - 1)) == 0` directly into your Lisp programs, which is a trick to check if a positive integer n is a power of two.

Nested sums

Mathematics has compact notation for sums and products because they pop up everywhere:

$$\sum_{k=1}^{5} f(k) = f(1) + f(2) + f(3) + f(4) + f(5)$$
$$\prod_{k=1}^{5} f(k) = f(1) \cdot f(2) \cdot f(3) \cdot f(4) \cdot f(5).$$

These are frequently combined in various ways, especially in counting problems. Were I to ask you to compute

$$\sum_{0 \le i \le 5} \prod_{i \le j \le 2i} \sum_{0 \le k \le j} \sin \sqrt{i+j+k},$$

how would you do it? In most programming languages, the most natural solution is to write a triply nested loop, maintaining intermediate sums and products. In Python, for instance, we would have

```
s = 0
for i in range(6):
    p = 1
    for j in range(i, 2*i+1):
        s2 = 0
        for k in range(j+1):
            s2 += sin(sqrt(i+j+k))
        p *= s2
    s += p
```

It's awkward, error-prone, and makes the computation completely opaque. But it is completely mechanical, and in Common Lisp, it is easy to encode these mechanical code generation rules without fuss. In Common Lisp, we can write the following:

```
> !sum(i to 5) !prod(j = i to (* 2 i)) !sum(k to j) (sin (sqrt (+ i j k)))
27096.02
```

This is *not* pseudocode, but it's also not built into Common Lisp. In this case, we installed a reader macro that gives rules for how to interpret `!op`, and then we taught[31] it what to do when `op` is either `sum` or `prod`.

[31] We could teach it others, like `mean` or `logsum`. We could also teach it recognize the Unicode characters for Σ and Π if we truly wanted to make our code look like mathematics.

The code to implement this functionality is 20 lines. We won't explain it, except to say the crux of the matter is filling out two loop templates for computing a sum or product.

```
(defun parse-limits (limits)
  (optima:ematch limits
    ((list sym 'to uplim)
     (values sym 0 uplim))
    ((list sym '= lowlim 'to uplim)
     (values sym lowlim uplim))))

(set-macro-character #\!
 (lambda (stream char)
   (let ((kind (read stream t nil t)))
     (multiple-value-bind (var low hi)
         (parse-limits (read stream t nil t))
       (let ((expr (read stream t nil t)))
         (ecase kind
           (sum  `(loop :for ,var :from ,low :to ,hi
                        :sum ,expr))
           (prod (let ((p (gensym "P")))
                   `(loop :with ,p := 1
                          :for ,var :from ,low :to ,hi
                          :do (setf ,p (* ,p ,expr))
                          :finally (return ,p)))))))))
```

An adamant functional programmer might detest a syntactic solution here if lazy sequences and combinators could solve it[32]. As a functional programmer myself, I think lazy evaluation and combinators together form a fantastic and powerful *language*. While such a language might have syntax and semantics to support *this* particular problem well, even industrial-strength languages like Haskell must resort to haphazard extensions—like Template Haskell—to achieve even a modicum of syntactic abstraction[33].

[32] And it's important to know that Common Lisp does *not* preclude you from using these tools!

[33] Moreover, Haskell makes use of its own syntactic sugar frequently, like with list comprehensions or `do`-notation, even though the two have a mechanical translation to simple function calls

Is new syntax dangerous?

It's easy to imagine the worst-case scenario of being able to add syntax to a language. You're working on a project with a team, and Bob Smith across the hall added RPN to the code base. Then your boss pulled in a library for doing infix math. Now the code base has an inscrutable mix of prefix notation, infix notation, and postfix notation. Everybody throws their hands up in frustration, and decides to port the code base to Go, which lacks any sort of freedom to express an idea that UNIX old hats didn't think of already.

Common Lisp affords a programmer a great deal of power and flexibility, and without checks and balances, it's true that code can become a mess. But I think this is the case with any language; Java is frequently the butt of any business logic joke about Kafkaesque towers of abstract factories. But good Java code can be written with disciplined collaboration. Lisp is no different, and is not somehow

more prone to being misused. It's just that Lisp has distinct avenues for misuse.

I myself have either been an individual contributor or a manager of teams exceeding a dozen senior Lisp programmers. In my experience, with strict code review and unit testing policies, along with easy and accessible version control, Common Lisp projects run as smoothly as any other—except the work gets done quicker, cheaper, and with happier programmers. ☺

Where Lisp is used

There have been countless companies that have used Common Lisp in a serious capacity over the last several decades, many of which no longer exist. There have even been companies[34] whose products included hardware for executing Lisp efficiently.

[34] Symbolics, Texas Instruments, and Lisp Machines Inc. are but a few examples.

Nonetheless, there are modern companies that use Common Lisp today.

Google Inc. Have you used Bing Travel, CheapTickets, Kayak, or Orbitz? Have you flown American, Delta Air Lines, United Airlines, US Airways, and Virgin Atlantic? Each of these uses flight search and pricing powered by a Common Lisp program called QPX. QPX was designed and developed by a company called ITA Software, which was later acquired by Google in 2010 for USD$700 million. Google continues to develop the software in Common Lisp, and has employees that contribute heavily to the open-source Common Lisp ecosystem.

Rigetti Computing Rigetti manufactures and deploys universal, gate-based quantum computers based on superconducting transmon qubits. The language used to program their quantum computers is called Quil, which is an instruction-based language for expressing hybrid classical/quantum computations. Among their programming SDK is an optimizing Quil compiler called `quilc`, and a high-performance multi-core simulator called `qvm`. The compiler is known for being the highest performing open-source compiler, only rivaled by a single proprietary offering. These software packages are open-source[35].

[35] `https://github.com/rigetti`

Grammarly Grammarly develops a natural language engine for improving writing. Aside from containing the usual offering for spelling and grammar checking, it also has facilities for sentiment analysis (e.g., determining if something sounds happy or angry) and providing suggestions for improving it. Their product is written in Common Lisp and has been written about on their engineering blog.

SISCOG SISCOG was founded in 1986, and makes software for transportation scheduling and management. Their software supports a multitude of European railways, including the London underground. Their principle software is written in hundreds of thousands of lines of Common Lisp.

GrammaTech CodeSonar is software for performing static and binary code analysis developed by GrammaTech. GrammaTech's research team has a "large Common Lisp code base" and have released several open-source software projects.

There are three honorable, historic mentions that are personal favorites.

ViaWeb Paul Graham, Robert Morris, and Trevor Blackwell founded Viaweb in 1995, a company that offered a web-based application for creating online stores. It was sold to Yahoo! in 1998 for around USD$50 million. Graham has become a profound influence in both the Common Lisp community for authoring two books about Common Lisp[36], and the entrepreneurial community for his various essays about start-ups. Graham later founded Y Combinator, a venture capital firm which has funded an exemplary number of hugely finacially successful companies.

Reddit The first version of Reddit was written in Common Lisp. It was eventually rewritten in Python and they describe their experience on their blog[37].

Naughty Dog Both *Crash Bandicoot* and *Jak and Daxter* were landmark and hugely successful games for the Sony Playstation which tested the technical boundaries of the platform. The developers went whole-hog with Common Lisp. They wrote a domain-specific language[38] for representing game entities and state machines, and wrote a compiler for this language. *Crash Bandicoot* used it for only some of its game code, while *Jak and Daxter* used it for almost all of the game code.

[36] *ANSI Common Lisp* and *On Lisp*, the latter of which influenced my own Common Lisp style.

[37] It's important to note that the main issue they ran into was about lack of libraries. Especially with the development of Quicklisp, this has become much less of an issue over the past 15 years.

[38] Andy Gavin, one of the company's co-founders and programmers, describes the very thesis of this essay in an interview by Ars here.

How to get started

It would be remiss to not say how to get started with Common Lisp. There's good new and bad news. The good news is that there are many amazing resources to learn it. There's quite an amazing number of books about Common Lisp, most of which are pretty high quality, but these four books I treasure immensely:

- *Practical Common Lisp* by Peter Seibel. This book is written for programmers, and takes you through Common Lisp by way of a series of interesting projects. It is free online.

- *Paradigms in Artificial Intelligence Programming* by Peter Norvig. This book is a *tour de force* on using Common Lisp to solve problems in classical artificial intelligence. While the title says "artificial intelligence", to me, it's the best book to learn how to *program*. It is also free online.

- *On Lisp* by Paul Graham. Graham refreshingly embraces a very lightweight and bottom-up programming style. The book gets into the nitty-gritty details of macro programming, moreso than Seibel and Norvig. It is my favorite introduction to thinking broadly about building languages. It is also free online.

I also must mention the Common Lisp HyperSpec. This is the *de jure* specification of the language[39].

The bad news is that unless you're already a fan of Emacs or unless you're willing to spend some money, you won't have a Cadillac experience jumping into the current Common Lisp technology stack.

First and foremost, while it's possible, the best Common Lisp experience won't be in the terminal. Most REPLs don't have history, line editing, or tab completion by default[40].

The standard setup for many open source Common Lisp hackers is SBCL, Emacs[41], SLIME[42], and Quicklisp. SBCL is a Common Lisp implementation, Emacs is the editor, SLIME is the Emacs package that turns it into an IDE, and Quicklisp is the package manager. This package combination is bundled in an easy-to-download package called Portacle[43]. While the learning curve of Common Lisp, Emacs, and SLIME all together is quite steep, I think the return is much greater than the cost. It has been a daily driver for me for over a decade.

As of writing, the Atom editor has an extension called `atom-slime` which provides many of the benefits of the Emacs+SLIME setup while being in a more familiar editor to many.

If you want a truly bespoke IDE for writing Common Lisp, then both LispWorks Ltd. and Franz Inc. offer commercial versions of their compiler and IDE[44]. At the time of writing, a full-featured version of either of their products costs more than USD$1000. Both companies also offer something lacking[45] in the open-source world: libraries to create native, cross-platform GUIs.

[39] Don't let the '90s-looking website put you off!

[40] It seems that Common Lisp vendors prefer to leave that for others to decide.

[41] As it happens, Emacs' scripting language is called Emacs Lisp, which has a resemblance to Common Lisp. While mutually intelligible, they're different languages.

[42] SLIME stands for "Superior Lisp Interaction Mode for Emacs", and it is true to its name.

[43] `https://portacle.github.io/`

[44] They are also compatible with the Emacs+SLIME setup.

[45] It is possible to make GUIs using open-source code, like SDL, McCLIM, and Qt. But they're comparatively cumbersome.

Closing

A common question I get is, "if Lisp is so good, why isn't it more popular?" This itself deserves an entire opinion piece, but I offer the following.

Common Lisp by and large doesn't have any shortcomings in

terms of raw capability. You can write exceedingly fast, multithreaded code. You can deploy either as scripts or as an executable. You have most modern programming paradigms on your belt. You have successful companies built around Common Lisp[46]. And you have a great library ecosystem, much better than many up-and-coming languages[47]. So what gives? I think there are three dominant social factors at play that trump most technical considerations: familiarity, employability, and education.

[46] And anecdotally, I've never had any issue hiring Common Lisp programmers.

[47] Though I wouldn't say the available libraries are as voluminous as, say, in Python

A common trait among popular programming languages is their familiarity, especially in syntax[48]. I frequently hear the common pleasantry, "once you've learned one programming language, you've learned them all." It's of course false in general, but it is sort of true within the sphere of similar languages, many of which occupy top spots in popularity rankings.

[48] The irony isn't lost on me.

Usually familiarity and commonness drive an argument about maintainability, as if knowing a language's syntax automatically makes code bases intelligible. And maintainability is used as the final nail of justification in any large organization. It, to me, is an intellectual argument, not one derived empirically. In my experience, empirically, a language like Python allows more hands to touch it, but it also invariably leads to an inferior product.

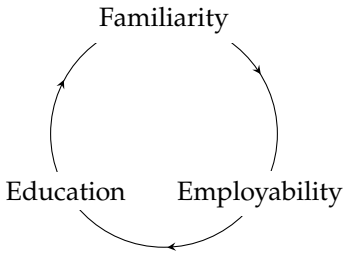
If you were to take a snapshot of the code at Rigetti Computing[49], about half of it is in Python and half of it is in Common Lisp. However, if you measure churn—how frequently old code is displaced with new code—you'd see that Common Lisp has very little while Python has an enormous amount. The code itself is ignoring how we've switched deployment strategies three times, how we did a Python 2→3 port over several months, and how we've proposed scrapping 80% of it because the code has been unable to grow with the needs of experimenters. This is but one data point collected over four years, but it's all seen as "justified" since it's "what we know". Familiarity provides an enormous amount of momentum to even bad solutions.

[49] Compilers, simulators, control software, experiment software, etc.

Beyond familiarity is employability. Of course, an individual's employability is not a independent factor from the employer's familiarity with tools, nonetheless, it's a consideration that individuals must make regardless of their own familiar tools. Powerhouses of software engineers generally advocate for a particular programming stack. Google was known as a Java, C++, and Python shop—and remains so to this day—until they released their own language called Go. Go is familiar to C-derivative programmers, and immediately saw immense popularity. Go, like the aforementioned languages, is now *employable*. Google and many other companies will pay you to write it.

What happens when a skill is highly employable? Many schools[50], especially mid-level schools, will teach it. My brother, currently going to a community college for an associate's degree in computer science, is learning C++ and Java as his very first introduction to programming. Not a single course offers another language to learn.

These social effects are self-reinforcing, and with the gigantic offering of programming languages, folks have little incentive to try anything else. More familiar languages will likely be more employable. More employable languages will often be taught more frequently. And a larger workforce of students knowing a language leads to higher familiarity.



Common Lisp doesn't have a foothold in any of these aspects[51] for most people, except it's modestly employable if you seek it out. For these reasons, Common Lisp is not as popular.

With all of that said, I think Common Lisp is seeing another resurgence[52]. It will rise, it will fall, and it will equip another generation of programmers with one of the most powerful ideas in software engineering history.

[50] Not all schools, though. Schools with programming language researchers often have a curriculum with a variety of languages.

[51] My rationale may seem like a "holier-than thou" argument, but I myself am susceptible to the same biases. Racket is a wonderful and promising programming language that I haven't explored for these reasons.

[52] The penultimate one was probably two decades ago, when SBCL was released. The last one was probably one decade ago with Quicklisp was released. We are due for another about now.