An Introduction to Emacs Lisp

Harry R. Schwartz

April 8, 2014

This article is derived from a talk I gave at the New York Emacs Meetup. A video of that talk is also available.

What we'll be covering

We'll go over the basic features of Emacs Lisp, including an introduction to higher-order functions, but not including macros.

- Atoms
- Functions
- Lists
- Variables
- Defining Functions
- Conditionals
- Recursion
- Anonymous Functions
- Higher-Order Functions

What we *won't* be covering

Most of the Emacs standard library. This tutorial covers the features of the language, but it doesn't address how we manipulate buffers, regions, and so on.

We'll also be making passing references to macros, but we won't be exploring them in any detail.

Introduction

Emacs can be thought of as a giant REPL. Like `bash`, `python`, `irb`, and other REPLs, it **R**eads in an expression, **E**valuates it, **P**rints the results, and **L**oops to read the next expression.

In Emacs' case, the REPL is started when we launch the Emacs application. Every time we interact with the editor, we're just executing some code in the

context of that REPL. Every keystroke, every mouse click, and every menu action correspond to evaluating an expression in this REPL.

Similarly, your `.emacs` file is just the first code that's executed by that REPL after it starts.

Don't worry if this doesn't make too much sense now; you'll get it soon.

To execute this code as we go along, you'll first want to open up a new buffer with C-x b and name it whatever you'd like. Run `M-x emacs-lisp-mode` to put it into the correct mode.

Paste code snippets into this buffer, place your cursor (called "point" in Emacs lingo) right after the expression, and hit C-x C-e. That'll evaluate the expression in the Emacs REPL. The results of the expression will appear in the *echo area* in the bottom-left corner of your window.

Atoms

Atoms are the simplest objects in Emacs Lisp. They evaluate to themselves, so evaluating these objects will just return themselves.

We have integers:

```
42
```

Floats:

```
3.0
```

And strings:

```
"foo!"
```

Functions

Lisp expressions are either atoms or applications of functions (or macro expressions, technically, but that's outside the scope of this article).

To call a function, we wrap it all in parentheses, like so:

```
(+ 1 2 3)
```

That calls the function `+` on the arguments `1`, `2`, and `3`, evaluating to `6`.

When evaluating a function, Lisp first evaluates the arguments to the function, the applies the function to those arguments. This means that the arguments to a function can themselves be function calls:

```
(+ (* 2 3)
   (/ 8 4))
```

In this example, the interpreter first executes `(* 2 3)` and `(/ 8 4)`, which evaluate to `6` and `2` respectively. These values become the arguments to `+`, so the entire expression evaluates to `8`.

Lists

Lisp is really interested in lists; it's the **LIS**t **P**rocessing language, after all.

Let's try to evaluate a list:

```
(1 2 3)
```

When we execute that, we get the error: `Invalid function: 1`. That's reasonable: the interpreter is trying to execute the function `1` on the arguments `2` and `3`, but of course `1` isn't really a function.

If we want to refer to a list without trying to evaluate it, we can wrap it in a call to the `quote` function. In the same way that quoting a word in English refers to the word and not the thing it represents, quoting an expression in Lisp refers to the list itself. To say that another way, quoting an expression treats it as *data*, not as *code*.

```
(quote (1 2 3))
```

And boom! We get `(1 2 3)`, just like we wanted.

It turns out that quoting is something we do pretty often in Lisp, so there's a bit of handy syntax to make it easier.

```
'(1 2 3)
```

`'(1 2 3)` evaluates to `(1 2 3)`, too, and we can generally use `'` and `quote` interchangeably. We'll be using `'` from here on out, because it's terser.

We can quote nested lists, too:

```
'(1 2 (3 4 5) (6 (7)))
; => (1 2 (3 4 5) (6 (7)))
```

We can also quote a list that looks like a function call:

```
'(+ 1 2 3)
; => (+ 1 2 3)
```

This just evaluates to the four-element list `(+ 1 2 3)`. Notice that nothing is evaluated inside the quoted expression.

We can also create a list with the `list` function:

```
(list 1 2 3)
; => (1 2 3)
```

Now that we know how to refer to whole lists, we can start manipulating them. The simplest function is `car`, which just returns the first element of the list.

```
(car '(1 2 3))
; => 1
```

Conversely, `cdr` returns the list minus the first element.

```
(cdr '(1 2 3))
; => (2 3)
```

Why are these functions called `car` and `cdr`? It's just an historical accident, but you can read the whole story here. They're basically vestigial. We'd probably name these functions `first` and `rest` if we were rewriting Emacs from scratch today, but because they're already used so widely, it's just something you'll need to memorize. Sorry. =(

Let's try evaluating the empty list.

```
'()
```

Bizarrely, this evaluates to `nil`! In Lisp, `nil` and the empty list are equivalent. They're literally the same thing. You can append objects to `nil` to create lists, and you can test for the empty list in conditionals (where it's falsy).

The empty list evaluates to itself:

```
()
```

As does `nil`:

```
nil
```

When we take the `cdr` of a one-element list, we get the empty list, `nil`:

```
(cdr '(42))
```

The function `null` determines if a value is `nil` (or `()`). This is especially useful in recursive functions, where we often use the empty list as a base case.

```
(null nil)
```

```
(null '())
```

We've seen how to decompose lists with `car` and `cdr`. The reverse operation, building up a list, is accomplished with `cons`.

`cons` (think "**cons**truct") takes an expression and a list. It returns a new list whose `car` is the result of the expression and whose `cdr` is the list.

```
(cons 1 '(2 3))
```

By `cons`ing `1` onto `'(2 3)` we created the list `'(1 2 3)`.

`cons`ing a value onto an empty list creates a list containing that value:

```
(cons "foo" '())
; => ("foo")
```

We can chain these `cons` calls together to build up a list from scratch:

```
(cons 1 (cons 2 (cons 3 '())))
```

Under the hood, the `list` function is just a series of `cons` calls.

This technique is very common when designing recursive functions. We often recurse through a list using `car` and `cdr` and/or build up a result with `cons`.

There's also a handy `append` function for joining two lists to create a new one:

```
(append '(1 2) '(3 4))
```

This just creates a new list, `(1 2 3 4)`.

A fun exercise (after you've read the next couple sections) might be implementing your own version of `append` using `car`, `cdr`, and `cons`.

Variables

Like virtually every other language, Emacs Lisp has variables. Emacs has many variables that are already defined and in use; for example, `path-separator` contains a string that separates file paths in the system's `$PATH` variable.

```
path-separator
```

In my system, evaluating `path-separator` returns `":"`.

Trying to evaluate an undefined variable raises an error:

```
some-list
```

I got the error `Symbol's value as variable is void: some-list`. That just means that the symbol `some-list` doesn't point to a variable. Let's fix that.

```
(set 'some-list '(1 2 3))
```

We assign variables with `set`. `set` takes the name of a variable (quoted, so it's not evaluated) and a value, and sets the variable to that value. So, if we try to evaluate `some-list` again:

```
some-list
; => '(1 2 3)
```

Unsurprisingly, we set variables fairly often. A long time ago, Lisp programmers decided that having to quote the name of the variable was a bit of a hassle, so they added `setq` as an alias. `setq` just implicitly quotes the first variable, so it's a bit more convenient. In practice, you'll rarely see `set`; most folks favor `setq`, so I'll be using that exclusively from here on.

```
(setq my-list '(foo bar baz))

my-list
; => (foo bar baz)
```

"But Harry," you might justly complain, "I thought you said that the arguments to the function would be evaluated before the function was executed! How does `setq` automatically quote that variable name? It's just a function, right?"

The answer is that `setq` actually *isn't* just a function. `setq` is an example of a *macro*. Macros are a bit like functions, but they also make it possible to manipulate their arguments before they get executed—in this case, the `setq` macro implicitly wraps a `quote` around the first argument. Macros are a big topic that's way outside the scope of this tutorial, but they're just *awesome*, so I'd highly recommend checking them out when you get a chance. They're one of the things that make Lisp really fun to work with.

Now, back to variables.

`setq` defines a variable globally. Defining all of our variables globally is shoddy practice, so we also have locally scoped variables. These are defined with a `let` expression. `let` also does some delayed-evaluation trickery. Its first argument is a list of two-element pairs. Each pair contains a variable name and its initial value.

Every subsequent argument to `let` is evaluated after those variables have been defined.

```
(let ((a 1)
      (b 5))
  (+ a b))
```

Evaluating the expression above should yield `6`. `a` was bound to `1`, `b` was bound to `5`, and we summed them. The return value of a `let` expression is the return value of the last expression in it (in this case, `(+ a b)`).

To prove to ourselves that `a` and `b` were only defined in the scope of the `let` expression, let's try evaluating `a`:

```
a
```

Phew, `Symbol's value as a variable is void: a`. So that's good; `a` isn't defined any more.

Sometimes we want to define a few variables in a `let` expression, and have the value of one depend on the value of another. `let` doesn't allow this (you could think of the values in a `let` expression as being bound in parallel), but a variant, `let*`, does.

```
(let* ((a 3)
       (b (+ a 5)))
  (+ a b))
```

This should return `11`, since `a` was bound to `3` and `b` was bound to `8`.

Why wouldn't we always just use `let*`? Heck, given that we have `let*`, why have `let` at all?

There are some purity arguments—the implementation of `let` is quite simple, but `let*`'s is significantly more complex—but that's not really a compelling reason unless you're a language designer.

When possible, you should use `let` because it's intention-revealing. Other programmers will be able to read through your code and immediately know that there are no dependencies between the variable bindings. That makes reading code a bit easier, and other programmers reading your code (including yourself, in a few months) will appreciate it.

Defining functions

Okay, so. We can evaluate atoms, call functions, build and decompose lists, and bind variables globally or locally. We're making some good progress.

Here's the simplest function I can think of.

```
(defun say-hello ()
  "hello!")
```

In this expression, `defun` defines a function called `say-hello`. The empty list indicates that it takes no arguments. Every subsequent argument to `defun` will be executed sequentially when the function is called, and the return value of the last expression will be the return value of the function. In this case, the *body* of the function is just the string `"hello!"`.

Sound good? Let's call this thing.

```
(say-hello)
```

Sweet, it said `"hello!"` Why hello, function! :D

Next, let's try a function that takes an argument. How about `square`ing a number?

```
(defun square (x)
  (* x x))

(square 2)
```

And behold, `4`. Lookin' fine.

We might wonder, "What happens if we send something non-numeric to our function?" Well,

```
(square "uh-oh")
```

We get the error `Wrong type argument: number-or-marker-p, "uh-oh"`. This is Lisp's way of saying that some function (`*`, in this case) can only operate on numbers.

Lisp is a dynamic, interpreted language. It doesn't have a strong static typing system like Haskell (or even Java!), so there's no way for it to detect problems

like this in advance. This actually gives us a lot of power, and it's often terribly convenient, but it *does* mean that we can shoot ourselves in the foot sometimes.

(Languages like Ruby and Python often overcome these pitfalls with solid suites of unit tests. Lisp is just as testable as those languages, but it doesn't have a strong testing *culture*. I'll save that rant for another day.)

Let's look at a more complex example. In 2D space, to determine the distance between two points we use the Euclidean distance metric (usually formulated as "the square root of the sum of the squares"). Suppose we have two points, *(x1, y1)* and *(x2, y2)*. Let's implement a function to compute the distance between them:

```
(defun distance (x1 y1 x2 y2)
  (sqrt (+ (square (- x2 x1))
           (square (- y2 y1)))))
```

Here we've defined a function that takes four arguments and returns the square root of the sum of the squares. Note that we used our previously-defined `square` function as well as the built-in `sqrt`.

```
(distance 3 0 0 4)
```

Lo and behold, the distance between *(3, 0)* and *(0, 4)* is `5.0`, as we'd expect.

Conditionals

We'll be looking at three types of conditionals, from the simplest to the most general. First, though, we should talk about Boolean expressions.

In Emacs Lisp, every value is *truthy* except `nil` and the empty list `()`. Notably, `0` and `""` and also both truthy. If we want to speak directly about truth, we use `t`, the equivalent of `true` or `True` in other languages. There's no equivalent *false* value in Emacs Lisp; we generally use `nil` instead.

In summary: use `t` (or anything else, really) for truth, and `nil` for falsity.

The functions for *and*, *or*, and *not* are, reasonably enough, `and`, `or`, and `not`. No big shocks here. `and` and `or` can both take any number of arguments.

```
(and t "" 0 7)
; => 7

(or nil "foo" '() "bar")
; => "foo"

(not nil)
; => t
```

Note in the above examples that `and` and `or` return the first argument that satisfies them. `or` will return the first truthy value (or `nil`, if nothing's truthy),

and `and` will return the last argument, if they're all truthy (and `nil` otherwise). You may be tempted to use these return values to write very terse code, but I'd advise you to resist that temptation. Overly clever code will only be harder for you to understand later.

Some functions—`=` or `null`, for example—just return `t` or `nil`. Lisp programmers refer to such functions as *predicates*. Predicates are usually (though not always) distinguished by being suffixed by "`-p`". For example, when we tried to `square` a string in the last section, the interpreter claimed that our string didn't satisfy `number-or-marker-p`, which we can now deduce was a predicate that tested the input of the `*` function.

Now that we can write our own predicates, we can get to conditionals.

If we want to take an action only if a predicate is true, and otherwise do nothing, we can use a `when` expression.

```
(when (= (+ 2 2) 4)
  "passed sanity check!")
```

Pfew, `2 + 2 = 4` and the world is sane.

Personally, I don't use `when` all that much, but it shows up occasionally in my `.emacs` configuration. For example, there are certain things I only want to do if Emacs is running as a stand-alone application (not in a terminal), and I find that `when` is a nice fit for that.

More usefully, `if` evaluates a predicate, then evaluates one expression if it was true, and the other if it was false. Let's wrap an `if` in a function to try it out.

```
(defun evens-or-odds (n)
  (if (= 0 (% n 2))
      "even!"
    "odd!"))

(evens-or-odds 4)
(evens-or-odds 3)
```

We're probably unsurprised to learn that `4` is `"even!"` and `3` is `"odd!"`.

`if` is great, but it'd be nice if we had an extended *if/elseif/. . . /elseif/else* construct like we do in other languages. And, of course, we do! `cond` is a generalization of `if` that can match an arbitrary number of cases. It takes a collection of lists, each of which starts with a Boolean expression. It runs through each such expression searching for one that matches, then evaluates the remaining elements in *that* list and returns the result. This makes more sense as an example:

```
(defun pick-a-word (n)
  (cond
   ((= n 1) "bulbous")
```

```
   ((= n 2) "bouffant")
   ((= n 3) "beluga")
   (t "gazebo!")))

(pick-a-word 2)
; => "bouffant"

(pick-a-word -72)
; => "gazebo!"
```

Notice that our last condition is just `t`. Since `t` is always truthy, we can use it to indicate an *else* or *otherwise* case. If none of the other expressions are truthy, `t` definitely will be.

Recursion

Recursion is the process of calling a function that calls itself. It's pretty trippy, but it's extraordinarily useful and not too difficult to use once you get the hang of it.

A recursive function generally contains a conditional. One branch of the conditional involves somehow simplifying the problem—perhaps taking an element off a list or decrementing a number—then calling the function itself with that simplified quantity and using the results. The other branch is the *base case*, which kicks in when the thing being investigated is at its simplest—perhaps the list is empty, or the number had been reduced to 0.

I'm not going to go into any great depth with recursion, because it's a technique with some complex implications. However, you should see a simple recursive function in Lisp. An implementation of the factorial function is the canonical example; I don't dare buck tradition by demonstrating anything else.

```
(defun factorial (n)
  (if (< n 1)
      1
    (* n (factorial (- n 1)))))

(factorial 5)
; => 120
```

In this case, `factorial` calls itself repeatedly until the base case kicks in when `n` hits 0. The stack of function calls unwinds, eventually returning the product `(* 5 (* 4 (* 3 (* 2 (* 1 1)))))`, which evaluates to `120`.

Anonymous functions

`"foo"` is a string literal. `42` is an integer literal. In the same way, anonymous functions are *function* literals.

Variables point to literal values. When we `(setq foo "bar")`, we're pointing the symbol `foo` to the literal value `"bar"`. Likewise, when we `(defun baz () "quuz")`, we're pointing the symbol `baz` to a function. Without referencing `baz`, how could we refer to the function that `baz` points to?

Lisp uses a macro called `lambda` to create function literals. The name is a reference to the lambda calculus, which you emphatically do *not* have to understand to wield Lisp perfectly effectively.

A note on vocabulary: most programmers use the terms *function literal*, *anonymous function*, and *lambda* or *lambda function* more or less interchangeably. Unless you're a researcher working in programming language theory they mean essentially the same thing.

Okay, let's define us a lambda:

```
(lambda (x) (* x x x))
```

Painless! This is a function that takes an argument and cubes it. Simple enough!

However, we notice that it didn't really do anything. To execute a lambda, just use it like you'd use a function:

```
((lambda (x) (* x x x)) 5)
```

Boom, `125`. We created a lambda which was immediately evaluated and applied to `5`.

We can also call a lambda with `funcall`:

```
(funcall (lambda (x) (* x x x)) 3)
; => 27
```

`funcall` takes a function and (optionally) some arguments, and calls the function on those arguments. The function doesn't even have to be anonymous.

```
(funcall '+ 1 2)
; => 3
```

This is handy when the function's stored in a variable; we'll see that show up a little later.

So here's a cool trick. Since lambdas are just function literals, and since we can bind variable names to literals, we can replicate the functionality of `defun` on our own!

```
(fset 'cube (lambda (x) (* x x x)))

(cube 4)
```

`64`! Pretty awesome.

The first time I tried to do this I did something like `(setq cube ...)`. That doesn't work. Emacs Lisp keeps variables and functions in different namespaces (Emacs Lisp is a Lisp-2), so we have to use `fset` instead.

And no, there's isn't a `fsetq` :)

Higher-order functions

As we've seen, we can pretty much treat functions as just another kind of variable. There's no reason that we can't pass them in as arguments to other functions. Functions that take other functions as arguments are commonly called *higher-order functions*, but otherwise there's nothing special about them. We can easily write our own.

```
(defun transform-unless-zero (fn n)
  (if (= n 0)
      0
    (funcall fn n)))
```

So, if `n` is `0`, we just return `0`, but otherwise we apply the function `fn` to `n` and return the result. Cool?

```
(transform-unless-zero (lambda (n) (+ 1 n)) 0)
; => 0

(transform-unless-zero (lambda (n) (+ 1 n)) 7)
; => 8
```

Higher-order functions aren't terribly scary, and they're really useful.

In fact, they're *so* useful that they show up all over the place in Lisp. One of the most common examples is `mapcar`. It takes a function and a list, applies the function to each element in the list, and returns a new list of results.

```
(mapcar (lambda (n) (+ 1 n)) '(1 2 3 4))
; => (2 3 4 5)
```

Pretty cool! Note that `mapcar` doesn't change the original list. If it'd been stored in a variable, that variable's value wouldn't have changed. So, for example:

```
(setq our-list '(1 2 3 4))
(mapcar (lambda (n) (+ 1 n)) our-list)
; => (2 3 4 5)

our-list
; => (1 2 3 4)
```

If the mapping *had* changed the variable, we'd have called it a *destructive* operation. We like to avoid destructive operations in Lisp (and many other languages), because they introduce side-effects that can make our code harder

to reason about. (Incidentally, a function that operates deterministically with no side-effects is called *referentially transparent*. Impress your friends!)

We can also refer to functions by name, of course.

```
(mapcar 'upcase '("foo" "bar" "baz"))
; => ("FOO" "BAR" "BAZ")
```

It's worth noting that applying `mapcar` to an empty list will always just return an empty list.

```
(mapcar 'upcase '())
; => ()
```

`remove-if-not` is another handy higher-order function. It takes a predicate and a list. It returns a new list that only contains those items that satisfy the predicate.

```
(remove-if-not 'oddp '(0 1 2 3 4 5 6 7 8 9))
; => (1 3 5 7 9)
```

It's also not destructive, and it maintains the order of the items in the list.

Incidentally, if you find the name `remove-if-not` a bit confusing because it contains a double negative, you're not the only one. I still have to consciously think it through sometimes. Other languages often refer to this function as `keep-if`, `filter`, or `select`, all of which seem a bit clearer to me.

An example

Okay, we're in the home stretch. We've covered a lot of ground, so let's pull it all together into a big, fancy example. We're going to implement quicksort, a very clever recursive algorithm for efficiently sorting a list. Here's how the algorithm works:

- Pick some element in the list (we'll just always pick the first element). Call it the *pivot*.
- Create two other lists, containing all the items that are smaller than the pivot and all the items that are larger.
- Apply quicksort to both of the smaller and larger lists, recursing down until we hit the empty list.
- Join the smaller list, the pivot, and the larger list into one new list and return it. Done.

And here's how we'd implement it in Emacs Lisp:

```
(defun qs (items)
  (if (null items)
      '()
    (let* ((pivot (car items))
           (rest (cdr items))
```

```
        (lesser (remove-if-not
                 (lambda (x) (<= x pivot)) rest))
        (greater (remove-if-not
                  (lambda (x) (> x pivot)) rest)))
  (append (qs lesser) (list pivot) (qs greater)))))
```

This might look a little intimidating, but there's actually nothing new here. We're defining a function called `qs` that takes the list `items`. If `items` is empty, it returns an empty list, which is the base case. Otherwise it defines a pivot, uses `remove-if-not` to create the `lesser` and `greater` lists, recursively calls `qs` on them, and joins the results together. Not really that bad, is it?

```
(qs '(3 5 7 8 4 2 5 7 0 8 4 6))
; => (0 2 3 4 4 5 5 6 7 7 8 8)
```

Will ya look at that! It even works!

In real life, of course, we'd probably use the built-in `sort` function. There's no reason to build our own, and the built-in one is almost certainly more efficient.

Next steps

We just went from atoms to quicksort in under five thousand words. Not too shabby!

If you'd like to keep going with Emacs Lisp, there are a few resources I'd like to recommend:

- Bastien's excellent Learn #Emacs Lisp in 15 minutes was a big inspiration for my talk (and thus this article).
- The standard book-length tutorial is *An Introduction to Programming in Emacs Lisp*. This covers a lot more Emacs-specific functionality, like manipulating buffers, regions, and text.
- The official *Emacs Lisp Reference* is thorough and comprehensive. It's a bit heavy for beginners, but still a very useful reference.

I'll also be writing a couple short articles in the next couple weeks about setting keybindings and using the built-in documentation. Keep an eye out!