**Let's Learn Lisp (Racket Lisp)**

**CORE CONCEPTS:**

Lisp is programming written as a binary tree:



\*For those not familiar, a binary tree is a type of data structure where each node of the tree can have 0 to 2 child nodes, starting with exactly 1 node. The tree is read top-down, left-to-right. Below I’ve numbered the nodes in the order which we visit them:



Everything in Lisp is an atom or a cons cell. An atom is any kind of data: a number, string, symbol, function, macro, etc. A cons cell (it's named that for reasons not worth explaining) is a special data structure which forms the basis for the binary tree.

**THE CONS CELL:**

A cons cell is similar to a c struct with 2 pointers (always enclosed within '( )' parenthesis):

//c style example psuedocode

cons-struct {

pointer\* pointer1;

pointer\* pointer2;

}

;Comments in lisp are preceded by ';'

;Lisp uses the 'cons' function to make a

;new cons-cell

(cons pointer1 pointer2)

;pointer1 is normally called the 'car' (for reasons

;not worth explaining). pointer2 is normally called

;the 'cdr'

(cons car cdr)

Shorthand for 'cons' is a period '.' between paired pointers, with the outermost parenthesis prepended with a single quote ‘. Period separated lists are how Lisp will output a list when printing:

‘(left-pointer . right-pointer)

;The single quote tells Lisp “Don’t evaluate this as a function”

;The (list) and (cons) functions handle that part internally

The Lisp runtime automatically understands if a pointer is an atom or another cons cell

**LISTS:**

Lists are made of cons cells hooked together:

‘(left-pointer1 . right-pointer-to-the-next-cons-cell)

Which ends up looking like this in code:

‘(left-pointer1 .(left-pointer2 . right-pointer2))

A list filled with real numbers could look like this:

‘(1 . (2 . 3))

A list implicitly ends with a 'null' in racket lisp (in most other lisps

'null' is 'nil'). Here’s a code example with a tree representation:



Lists can be extended indefinitely:

‘(1 . (2 . (3 . (4 . (5 . null))))

The shorthand for the above can omit the periods:

‘(1 2 3 4 5)

Lists can by dynamically constructed using the ‘list’ function:

(list 1 2 3 4 5)

Lists can contain any data type

;#t is boolean True, #f is boolean False

‘(1 #t “a string” ‘a-symbol)

While what is considered a ‘list’ in lisp always takes the form of a singly linked list, cons cells are not limited to the above structure, they're a binary tree that can bind together any child lists:



The differing structures that cons cells can have become much more important in functions, which can choose the "path" the tree is evaluated and return early. Functions are shown later on below

**ATOMS:**

The content in the cons cell tree can be anything (lisp is dynamic and untyped by default) \**including functions*\*. When Lisp finds a function it executes it:

(+ 1 2 3) ;returns 6

;Most functions internally accept lists implicitly so you

;don't necessarily have to construct it using (list). + is

;the addition function, and it will create a sum total of

;everything that comes after it in its child list

When a Lisp runtime implementation executes code it is just moving through a tree evaluating functions and atoms. A code example with both the code representation and the tree representation from the perspective of the Lisp runtime:



**FORMATTING:**

To format Lisp code to be readable: every time we enter a child list (a new set of parenthesis) we can indent by 2 spaces on a newline:

;Valid:

(if my-boolean

(do-this-if-true value-A value-B)

(do-this-if-false value-C))

;Honestly just do what makes the most sense to make the lines readable

;Also Valid:

(if

my-boolean

(do-this-if-true

value-A

value-B)

(do-this-if-false

value-C))

The last thing that Lisp evaluates in a child tree is returned to the parent tree:

(if my-boolean

;if this evaluates it will return whatever

;'do-this-if-true' returns to the parent context

(do-this-if-true value-A value-B)

;if this evaluates it will return whatever

;'do-this-if-false' returns to the parent context

(do-this-if-false value-C)) ;

**EXAMPLE FUNCTIONS:**

Add up the total sum of the provided numbers (returns 6 in this case)

(+ 1 2 3)

Subtract entries past the first from the first value (returns -4 in this case)

(- 1 2 3)

Returns #t if var1 and var2 are equal (returns #f in this case)

(equal? 1 2)

Create a list scope and declare any number of variables in the first argument cell. '[ ]' is functionally identically to '( )' and are used in special places to help improve readability. var1 and var2 will disappear when we leave the (let) scope

(let ([var1 val1] [var2 val2]) do-things-with-var1-and-var2-here)

Similar to c, assembles a string and sends it to stdout. '~a' grabs the next thing the list and inserts it into the string

(printf "a string to ~a to the console\n" "print")

Declare and define a variable in the parent scope (not visible to the parent's parent scope)

(define var value)

Assign a new value to a variable

(set! var new-value)

Define a function. Args will go out of scope when leaving this function. The last atom evaluated is returned to the parent context.

(define (function-name arg1 arg2 .. argn)

do-things-here)

**PUTTING IT TOGETHER:**

Using what’s written above, here’s some example lisp code:

(define (is-a-equal-to-b a)

(define b (+ 1 2 3))

(equal? a b))

(let ([c "We are equal!!"]

[d "We are not equal!!!!!!"])

(if (is-a-equal-to-b 5)

(printf "~a\n" c)

(printf "~a\n" d)))

Here we define the function named “is-a-equal-to-b” with the single argument ‘a’

(define (is-a-equal-to-b a)

Next we define the variable ‘b’ to be the sum of 1, 2, and 3 (thus b == 6)

(define b (+ 1 2 3))

Finally we check if a is equal to be and return the true/false result

(equal? a b))

In this section of code we define string variables ‘c’ and ‘d’ within a local ‘let’ scope.

(let ([c "We are equal!!"]

[d "We are not equal!!!!!!"])

We then call the function “is-a-equal-to-b” with 5 as the input

(if (is-a-equal-to-b 5)

If the function returns #t we print “We are equal!!” to the terminal. Otherwise we print “We are not equal!!!!!!”

(printf "~a\n" c)

(printf "~a\n" d)))

Here’s the result if I run that code in the Racket command line interpreter (the REPL):  
$ racket

Welcome to Racket v7.0.

> (define (is-a-equal-to-b a)

(define b (+ 1 2 3))

(equal? a b))

>

(let ([c "We are equal!!"]

[d "We are not equal!!!!!!"])

(if (is-a-equal-to-b 5)

(printf "~a\n" c)

(printf "~a\n" d)))

We are not equal!!!!!!

**YOU NOW KNOW EVERYTHING YOU NEED TO UNDERSTAND BASIC LISP**

Everything else is either learning what functions exist OR the macro system. There are some shorthand non-parenthesis enclosed keywords but they are the exception.

**THINGS LISP DOES WELL:**

**1) Manipulation of data structures**:

Lisps are typically good at manipulating data structures, and the most common data structures (maps, lists, dictionaries, hash tables, vectors) are supported in most Lisps by default and implemented in a consistent and powerful way.

It’s a bit difficult to give a thorough example of this, but I can point to the Racket Lisp documentation for a couple excellent examples:

(map): <https://docs.racket-lang.org/reference/pairs.html#%28def._%28%28lib._racket%2Fprivate%2Fmap..rkt%29._map%29%29>

(hash-for-each): <https://docs.racket-lang.org/reference/hashtables.html#%28def._%28%28quote._~23~25kernel%29._hash-for-each%29%29>

**2) Runtime code construction and compilation:**

Lisp’s simple syntax makes it easy for a Lisp runtime to translate Lisp expressions and convert them directly to memory allocated runnable versions.

This means that a standard Lisp ability is to dynamically read in, ‘compile’ (evaluate), and run new Lisp code \**during runtime*\*. You can have your program generate new lisp code as a (quote)d string and evaluate it with the (eval) function. Programs can share executable code *as text* but get the speed of memory mapped structures. (The initial (eval) is slow, but evaluated code can stay in memory and be subsequently used at maximum speed).

**3) Powerful macros:**

Another form of the power of cons cells and runtime interpretation takes is that macros are more powerful in Lisp than in many other languages because the code can be transformed recursively during initial code evaluation or during runtime (as opposed to before the compilation process, as in C). IE, because the actual runnable code is a tree of mutable pointers, its trivial to do things like append, prune, or transform the tree both before and during runtime.

As far as I’m aware, macros that do not require run-time evaluation are evaluated on the incoming source code and also run at maximum speed during runtime.

For comparison, not having as powerful macro capabilities is why the Qt framework has to generate code using its Meta-Object Compiler (moc), because c/c++ preprocessor directives only support single evaluation text replacement macros.

**4) Dynamic and Static typing:**

Lisp is also a dynamically typed language by default, meaning that a variable's type is determined at runtime (Lisp did it 30 years before Python!). Most lisps support optional typing systems, sometimes more than one, allowing developers to write code that best fits their needs.

**5) Dynamic human interaction via the REPL:**

Lisp runtimes have a dynamic execution environment called the REPL (Read-eval-print-loop) which allows the user to write and evaluate new code at runtime. It functions very similarly to Python's interpreter. To read a story on the amazing power of the REPL in combination with the above where the NASA Jet Propulsion Labs fixed a coding error in a rocket flight control system \*IN FLIGHT\*:

<http://www.flownet.com/gat/jpl-lisp.html>

**5) AGILE development & Rapid prototyping:**

Lisp has a reputation among proponents as being good for rapid prototyping, fitting well into the AGILE development model. The reason for this seems to be that the above points are all helpful for rapid development. My own experience with Lisp has reflected this view. Here’s LispWorks’ take on this topic (LispWorks is a cross-platform development IDE for Common Lisp): <http://www.lispworks.com/products/myths_and_legends.html>

**6) Speed**

Lisp runtimes come in a wide variety of flavors and speeds. Racket Lisp (the specific implementation shown in this guide) is a Scheme descendent, a minimal Lisp implementation, and is not the fastest lisp implementation out there. Racket does have a considerable amount of excellent libraries, hence my current usage of it.

However, Common Lisp implementations, the most popular ANSI standard, are known for being very fast. In the fastest implementations, compilation actually becomes machine code instead of Lisp virtual machine bytecode. Examples include CMUCL, SBCL, ClozureCL (source: <https://stackoverflow.com/questions/913671/are-there-lisp-native-code-compilers>).

In some cases it can be as fast (or faster) than C. See the following document comparing C to CMUCL (Carnegie Melon University Common Lisp): <http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf>.

Such speed isn’t necessarily the average case, but Steel Bank Common Lisp does rank even with Java according to these benchmarks: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/lisp.html>

As a note it seems that Lisp speed is greatest when using the type systems provided by their specific implementation, so the compiler/interpreter can make the most efficient optimizations.