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

The core concepts are the really important parts and can be explained simply, although they may take a few moments to understand.

Lisp is programming written as a binary tree:

.

/ \

. .

/ \ \

. . .

/ \ \

. . .

Where the code is LITERALLY a bunch of pointers evaluated down a binary tree.

\*For those who may have never studied data trees in school, 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:

1

/ \

2 7

/ \ \

3 4 8

/ \ \

5 6 9

Steps to understand LISP:

1) 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 called that for reasons not worth explaining) is a special data structure which forms the basis for the binary tree.

**THE CONS CELL:**

2) A cons cell is basically 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)

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

‘(car . cdr)

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

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

4) Lisp is smart enough to know if a pointer is an atom or another cons cell

5) 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))

or:

‘(car1 . (car2 . cdr2))

A list filled with real numbers could look like this:

‘(1 . (2 . 3))

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

'null' is 'nil'):

‘(1 . (2 . (3 . null))

;You don't ever have to put in the null, Lisp will automagically add

;it internally, just know the above is valid

7) Lists can be extended indefinitely:

‘(1 . (2 . (3 . (4 . (5 . null))))

8) The shorthand for the above is a list:

(list 1 2 3 4 5)

9) Lists are actually not limited to the above structure, they're really more like a binary tree that can bind together any child lists:

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

‘((1 . ((2 . #t) . "a string")) . ((list 4 5 6) . 3))

;The below isn't code, but hopefully the tree below lets you ;visualize the above, where each newline is going into a new

;child cons cell scope and each line points to a child cons ;cell.

‘( . ( . 3))

/ /

(1 . ) (list 4 5 6)

\

( . "a string")

/

(2 . #t)

10) Lists evaluate left to right, entering child scopes as encountered, and when done evaluating a child it will return to where it left off in the parent scope. This means the below are effectively equal when actually parsing through them:

‘((1 . ((2 . #t) . "a string")) . ((list 4 5 6) . 3))

(list 1 2 #t "a string" 4 5 6 3)

;The differing structures that lists can have becomes much

;more important in functions, which can choose the "path" a

;list is evaluated and return early. Functions are shown

;later on below

**EVERYTHING ELSE:**

11) The stuff in lists can be anything (lisp is dynamic and untyped by default) \**including functions*\*. When Lisp finds a function it executes it:

;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

(+ 1 2 3) ;returns 6

12) Literally, Lisp code is just moving through a list tree evaluating functions and atoms:

(if my-boolean (do-this-if-true value-A value-B) (do-this-if-false value-C))

13) 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

;Valid:

(if

my-boolean

(do-this-if-true

value-A

value-B)

(do-this-if-false

value-C))

14) 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)) ;

15) Some 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 functionly identically to '( )' and are used

;in special places to help improve readability. var1 and var2 will

;disapear 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)

(set! var new-value) ;assign a new value to a variable

;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)

16) If you can vaguely understand the above, you should be able to figure out the below:

(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)))

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

17) Everything else is either learning what functions exist OR the macro system (which is stupendously more powerful than other languages). There are some shorthand non-parenthesis enclosed keywords but they are the exception.

**THE POWER OF LISP**

1) The power of the cons cell:

Understanding the power of Lisp comes from understanding how having an impressively flexible and dynamic struct (the cons cell) as the basis of the language enables expression of other code. For instance, because of the simplicity of language it is \*very\* easy to translate human readable text expressions and convert it directly to memory allocated runnable versions.

This means that a standard Lisp ability is to dynamically read in, ‘compile’, 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).

2) The power of macros:

Another form of the power of cons cells and runtime interpretation takes is that macros are *far* more powerful in Lisp than in other languages because the code can be transformed and run \**during*\* runtime (as opposed to before the compilation process, as in C).

3) Dynamic 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!). This helps make writing code easier, as the programmer is freer to write their code via intent rather than spending a lot of time working out and managing types. Most lisps, however, do support optional typing systems, sometimes more than one!

4) Dynamic human interaction via the REPL:

Lisps all share a common 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 similiarly 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) Freeform naming:

As a side note variable names in Lisp are very unrestricted. In fact, the naming is *so* free that most Lisps support naming conventions just to help readability. The following are all valid names:

foo

foo-bar

\*foo\*

foo\*

&foo

+foo+

%foo

6) Readability

Another Lisp advantage is its simple structure lends itself to readability in my opinion. The reason for this is a combination of the standard indenting process (2 spaces per new scope on a newline OR if the newline is not on a new scope, then match the indentation of the previous argument) and because descriptive variable names in Lisp are encouraged.

Because Lisp code formatting is so easy, we can pack a lot more text on each line. While not \*necessary\* (indeed, a lot of Lisps use a very terse dialect) it does mean you can write descriptive, useful function names without worrying as much about horizontal space concerns. (Warning, the following code includes functions I've written, so there will be some things that won't make sense, just try to get an idea of what's going on if you can).

Here's an example from 3's unit tests for coroutines:

(define-coroutine

(co-test1)

(yield 1)

(yield 2)

#f)

(define ct1 (co-test1))

;Test the coroutine runs correctly

(test-true? "new coroutine alive" (ct1 'alive?) pr wait)

(test-equal? "coroutine yields expected 1" (ct1) 1 pr wait)

(test-true? "coroutine alive" (ct1 'alive?) pr wait)

(test-equal? "coroutine yields expected 2" (ct1) 2 pr wait)

(test-true? "coroutine alive" (ct1 'alive?) pr wait)

(test-true? "coroutine returns #f" (not (ct1)) pr wait)

(test-true? "coroutine dead" (ct1 'dead?) pr wait)

Here's another example where I've put some text on newlines:

(define (go env suspended-coroutine [ret-key #f] [ret-field #f])

(let ([q-idx (get-min-dp-q-idx env)])

(semaphore-wait (get-dp-queue-sem env q-idx))

(enqueue! (get-dp-queue env q-idx)

(list suspended-coroutine ret-key ret-field))

(semaphore-post (get-dp-queue-sem env q-idx))

(thread-resume (get-dp-thread env q-idx))

#t))