Language for 2016 - Common Lisp

John Cumming

February 8, 2016

Contents

1 Introduction

I have chosen common lisp as my language to learn in more depth for 2016 as I have dabbled with it over the past year or so, along with clojure, another very good lisp based language.

I chose common lisp as a language as I prefer the syntax over clojure and because I believe it should be able to produce more efficient code as well as being able to produce stand alone binary files without the dependence on the JVM that clojure has.

I have no specific plan as to how this learning will pan out, other than I intend to dive a bit deeper than I have done in the past.

The tools I use for this learning are SBCL and GNU Emacs with Slime. This is all running under OS-X El Capitan on a Macbook pro.

The complete source code for this article can be found here and the article formatted as PDF here. The org-mode file used to generate this web page and the lisp code can be found here.

Some books I have on lisp can be found here.

I have inter-dispersed larger examples through the code that hopefully build on the sections that have preceded.

2 An Overview of Common Lisp Syntax

Common lisp has a simple syntax for processing lists delimited by '(' and ')'. The lisp processes a list by applying the the first item in the list as an operator and the rest as operands. Lists can be nested:

```
;; Comments can be added using a semi colon
(+
 (* 3 4)
 (+ 2 3))
```

This code applies operand '+' to the result of applying operand '*' to 3 and 4, and the result of applying operand '+' to 2 and 3.

A list can be created as a just list of data, by using the 'quote' operand or by using a shortcut single quote, the following are both equivalent:

```
;; The following lines are equivalent
(equal (quote (1 2 3 4))
       '(1 2 3 4))
;; => T
```

More details on collections can be found in `Collections`.

String are, like most languages, delimited with double quotes.

Backslashes are used as escape characters, much like other languages. However, the use of a vertical bar allows special characters to be used without escaping:

```
;; The following items in the list are equivalent
(equal 'A\(B\) '|A(B)|)
;; => T
```

A hash symbol is a macro symbol, known as the dispatching macro character. There are many of these, for example:

```
;; #' - function abbreviation
;; #\ - character object
;; ,#+ - read-time conditional
;; #c - complex number
;; #( - vector
```

More details can be found in `Macro Dispatching Characters`.

A back quote can be used to allow a template to be used when generating code, with a comma used to evaluate a form and an '@' symbol used to splice a list into the template, for example:

```
(defparameter x '(a b c))
;; x
```

```
`(x)
;; => (x)
`(,x)
;; => ((a b c))
`(,@x)
;; => (a b c)
`(x ,x ,@x foo ,(cadr x) bar ,(cdr x) baz ,@(cdr x))
;; => (x (a b c) a b c foo b bar (b c) baz b c)
```

This is used extensively when writing macros. For more information on macros see Macros.

Colons are used in 2 situations. Firstly it can be used to indicate the package in which a symbol is defined:

```
;; reset is a symbol in the network package
;; (network:reset)
```

Packages are discussed in Packages.

It can also be used to denote a keyword, which is a symbol that always evaluates to itself and is constant. For example:

```
(eql ':foo :foo)
;; => T
```

Keywords are interned in the package KEYWORD and are automatically exported from it:

```
(eql keyword:foo :foo)
;; => T
```

3 Example 1 - Sum of Square Errors

An equation that is used in regression algorithms is the sum of the square of errors for a given dataset and function being fitted to the data.

Given a data set of size m with a single input variable x and a single output value y for each item in the data set and a function that is an attempt to fit a function to the values:

$$y = f(x)$$

Then an error can be calculated based on the sum of the square of the individual errors, giving an estimate of how well fitted the function is to the date:

$$E = \sum_{n=0}^{m} (f(x_n) - y_n)^2$$

Using lisp, we can write some code that takes a data set, computes the error based on several functions:

```
33 ;; First declare some data
34 ;;
35 (defparameter data '((0.1 . 1.1)
36                      (0.9 . 3.2)
37                      (2.1 . 5.9)
38                      (3.2 . 7.2)
39                      (3.9 . 9.0)
40                      (5.1 . 11.2)))
41
42 ;; then some equations
43 ;;
44 (defparameter equation-list
45   (list #'(lambda (x) (+ 1 (* 2 x)))
46         #'(lambda (x) (+ 1 (* x x)))
47         #'(lambda (x) (+ 1 x))))
48
49 ;; now create a function that applies a function
50 ;; to a set of input data
51 ;;
52 (defun apply-function (f d)
53   (map 'list #'(lambda (x) (funcall f (car x))) d))
54
55 ;; A function that returns the error as the difference
56 ;; between two values squared
57 ;;
58 (defun square-error (test-data calc-data)
59   (expt (- test-data calc-data) 2))
60
61 ;; A function that returns the sum of square errors
62 ;; of a collection of data and the results
```

```
63  ;;
64  (defun sum-square-error (f test-data)
65    (reduce #'+
66            (map 'list
67                 #'(lambda (test calc)
68                     (square-error (cdr test) calc))
69                 test-data (apply-function f test-data))))
70
71  ;; Now we can run the sum of square errors across all equations
72  ;;
73  (map 'list #'(lambda (eq) (sum-square-error eq data))
74       equation-list)
```

0.7400005 320.44208 61.350002

The data is defined as a set of cons cells with the car equal to an x value and the cdr equal to a y value. This is the test data that will be used to check the equations. It uses defparameter, but could equally be defined inline at Line 73.

The equations are defined as a list of lambda functions modeling the following equations for fitting to the data:

$y = 2x + 1$
$y = x^2 + 1$
$y = x + 1$

Again, these could have been defined inline at the point of use.

The apply-function function takes a function as an argument and a collection of data as an alist and executes the function taking the car of each item in the alist as the x value to calculate the y value.

The square-error function takes a single test data y value and a single calculated value and calculates the square of the error.

The sum-square-error function takes a function f and applies the `square-error` function to each item in the test data and the corresponding calculated output as calculated by the function f.

The output is generated by applying the sum-square-error function to each equation using the test data.

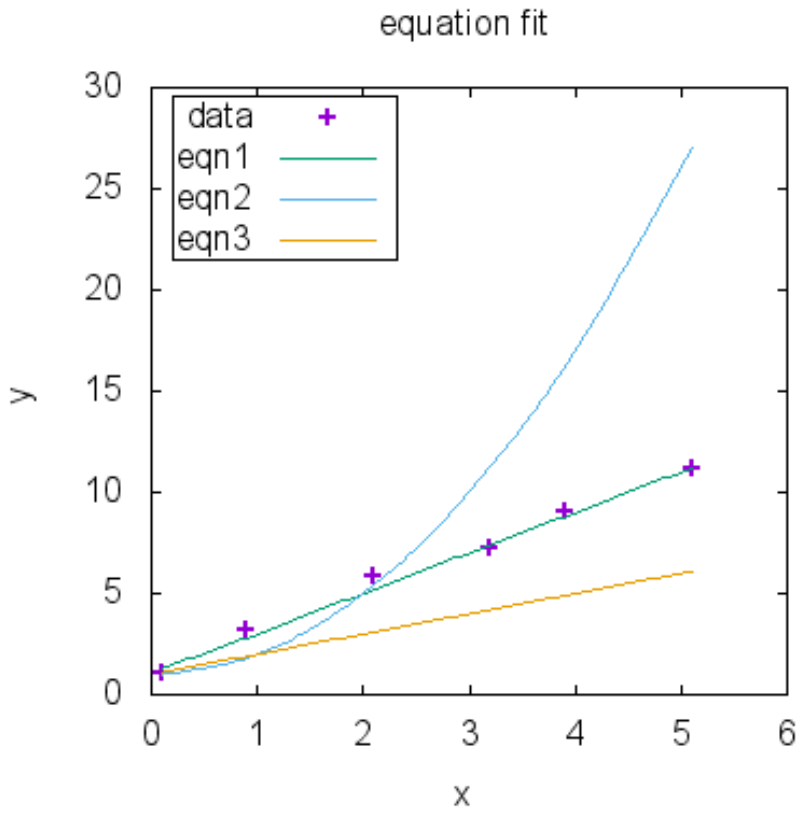
It can clearly be seen from both the results of the sum of square errors and the input data that eqn1 is the best fit.

To confirm this we can plot the data:

x	data	eqn1	eqn2	eqn3
0.1	1.1	1.3	1.01	1.1
0.9	3.2	2.8	1.81	1.9
2.1	5.9	5.2	5.41	3.1
3.2	7.2	7.4	11.24	4.2
3.9	9.0	8.8	16.21	4.9
5.1	11.2	11.2	27.01	6.1

using this gnuplot script:

```
set terminal png nocrop enhanced size 400,400
set key left box linestyle -1
set xlabel 'x'
set ylabel 'y'
set title 'equation fit'
plot data u 1:2 w p lw 2 title 'data', \
     data u 1:3 smooth csplines lw 1 title 'eqn1', \
     data u 1:4 smooth csplines lw 1 title 'eqn2', \
     data u 1:5 smooth csplines lw 1 title 'eqn3'
```



4 NEXT Core functions

- cons

  cons is used to construct lists, it puts a new element at the end of the list, or can be used for creating a pair:

```
(cons 1 3)
;; => (1 . 3)
(cons 3 nil)
;; => (3)
(cons 5 '(1 2 3 4))
;; => (5 1 2 3 4)
```

- car

  Given a list car retrieves the first item in a list:

```
(car '(1 2 3 4))
;; => 1
```

- cdr

  Given a list cdr retrieves the remaining list after the first element

```
(cdr '(1 2 3 4))
;; => (2 3 4)
```

- cadr / cddr etc

  These can be nested to various levels for example:

```
(cadr '(1 2 3 5))
;; => 2
(cddr '(1 2 3 4))
;; => (3 4)
```

- lambda

  lambda is used to create a function special form involving a lambda expression. The expression takes a lambda list and a form and returns a function:

```
(lambda (x) (+ 1 x))
;; => #<FUNCTION (LAMBDA (X)) {10035B665B}>
```

- funcall

  Funcall is used to call a function, which can be created with a lambda. However it becomes more useful when passing lambda expressions or functions as arguments (a little convoluted):

```
(funcall (lambda (x) (+ 1 x)) 3)
;; => 4

(defun do-something (x a)
  (funcall x a))
(do-something (lambda (x) (+ 1 x)) 2)
;; => 3
```

- function

  With the function function we can return a function from a function! This can be used to create a form of partial functions:

```lisp
(defun multiplier (n)
  (function (lambda (x) (* x n))))

(funcall (multiplier 3) 4)
;; => 12

(defun doubler (n)
  (funcall (multiplier 2) n))

(doubler 10)
;; => 20
```

- apply

  The apply function is very similar to funcall, except it takes a list as an argument. This means that it can be used when the number of arguments is unknown at compile time.

```lisp
(apply #'+ 100 '(4 5 6 7))
;; => 122
(defun add-to-ten (&rest args)
  (apply #'+ 10 args))
(add-to-ten 1 2 3 4 5)
;; => 25
```

- read

  The read function reads a single s-expression, skipping whitespace and comments and returns the lisp object denoted by the s-expression.

```lisp
;; given a file code.lisp containing
;;
;; (1 2 3)
;; 456
;; "a string" ; this is a comment
;; ((a b)
;;  (c d))
;;
(defparameter *s* (open "code.lisp"))
;; => *S*
(read *s*)
;; => (1 2 3)
```

```
(read *s*)
;; => 456
(read *s*)
;; => "a string"
(read *s*)
;; ((A B) (C D))
(close *s*)
;; => T
```

- eval

  The eval function just evaluates a lisp expression. It is used in combination with read to execute lisp expressions:

```
(eval (+ 1 2))
;; => 3
;; given a file code.lisp with a line
;; (+ 1 2)
(defparameter *s* (open "code.lisp"))
(eval (read *s*))
;; => 3
```

- print

  The print function prints the representation of a lisp object.

```
(print 2)
;; 2
;; => 2
(print (eval (+ 1 2)))
;; 3
;; => 3
```

- cond

  The primary conditional statement in lisp is the cond function:

```
(defun get-type-name (a)
  (cond ((null a) "null")
        ((atom a) "atom")
        ((listp a) "list")
        (t "unknown")))
(get-type-name nil)
```

```
;; => "null"
(get-type-name 1)
;; => "atom"
(get-type-name '(1))
;; => "list"
```

- quote

  The quote function is described `above`.

- atom

  The atom function is outlined below.

- null

  The null function determines if a symbol is nil:

```
(null nil)
;; => T
(null 1)
;; => nil
(null '())
;; => T
```

- set setf setq
- defun var parameter macro etc
- equality checks
- declare ?

5 Data Structures

5.1 Atoms

Atoms are things that are not cons cells, and can be tested using the atom predicate:

```
(atom 1)
;; => T
(atom :test)
;; => T
```

```lisp
(atom nil)
;; => T
(atom '())
;; => T
```

However, they are not things that cannot be broken down any further:

```lisp
(atom "text")
;; => T
(atom #(1 2 3))
;; => T
```

Symbols are atoms as well:

```lisp
(defun test-atomicity (x) (atom x))
(atom 'test-atomicity)
;; => T
```

And lambda expressions:

```lisp
(atom (lambda (x) (atom x)))
;; => T
```

Examples of some things that are not atoms:

```lisp
(atom '(1 . 3))
;; => NIL
(atom '(1 2 3))
;; => NIL
(atom (cdr '(1 2 3)))
;; => NIL
```

5.2 NEXT Sequences

Sequences are ordered lists of elements and can be manipulated by a variety of standard sequence functions. A sequence is either a vector or a list. Vectors are one dimensional arrays and Lists are linked lists made up of cons cells and are discussed here.

5.2.1 Arrays

- simple array
- bit array

5.2.2 Vectors

- simple vector
- bit vector

5.2.3 Strings

A string is a specialized vector with elements of type character. All the sequence functions `below` can be applied to strings.

- Manipulating the case of a string
  - string-upcase
  - string-downcase
  - string-capitalize
  - nstring-upcase
  - nstring-downcase
  - nstring-capitalize
- Trimming strings
  - string-trim
  - string-left-trim
  - string-right-trim
- Converting to and from strings
  - intern
  - symbol-name
  - string
  - coerce
  - parse-integer
  - read-from-string
  - write-to-string
- Comparing strings
  - string=
  - string/=

- string-equal
- string-not-equal
- string<
- string>
- string<=
- string>=
- string-lessp
- string-greaterp
- string-not-lessp
- string-not-greaterp

5.2.4 Lists

- Cons Cells
- Proper List
- Dotted List
- Circular List

5.2.5 Manipulating Sequences

- concatenate
- copy-seq
- count
- count-if
- count-if-not
- delete
- delete-duplicates
- delete-if
- delete-if-not
- elt

- every
- fill
- find
- find-if
- find-if-not
- length
- map
- mapcar
- map-into
- merge
- mismatch
- notany
- notevery
- nreverse
- nsubstitute
- nsubstitute-if
- nsubstitute-if-not
- position
- position-if
- position-if-not
- reduce
- remove
- remove-duplicates
- remove-if

- remove-if-not
- replace
- reverse
- search
- some
- sort
- stable-sort
- subseq
- substitute
- substitute-if
- substitute-if-not

5.3 Hash Tables

5.4 Trees

5.5 Association Lists

5.6 Property Lists

5.7 Records

5.8 Structures

5.9 Classes

6 Creating Variables

7 Functions

currying / partial no side effects let / flet

8 Control Operations

9 Error Handling

10 Lazyiness

11 Streams

12 Macro Dispatching Characters

set-macro-character symbol macros

13 Macros

14 Multimethods

15 CLOS

16 Packages

17 Standard Libraries

18 Important Libraries

19 Working with GNU Emacs and Slime