Language for 2016 - Common Lisp

John Cumming

January 6, 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 OSX 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.

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))
```

```
17
```

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
```

3 Core functions

4 Collections

5 Creating Variables

6 Functions

7 Control Operations

8 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:

```
;; First declare some data
;;
(defparameter data '((0.1 . 1.1)
                     (0.9 . 3.2)
                     (2.1 . 5.9)
                     (3.2 . 7.2)
                     (3.9 . 9.0)
                     (5.1 . 11.2)))

;; then some equations
;;
(defparameter equation-list
  (list #'(lambda (x) (+ 1 (* 2 x)))
        #'(lambda (x) (+ 1 (* x x)))
        #'(lambda (x) (+ 1 x))))

;; now create a function that applies a function
;; to a set of input data
;;
(defun apply-function (f d)
  (map 'list #'(lambda (x) (funcall f (car x))) d))

;; A function that returns the error as the difference
;; between two values squared
;;
```

```
53 (defun square-error (test-data calc-data)
54   (expt (- test-data calc-data) 2))
55
56 ;; A function that returns the sum of square errors
57 ;; of a collection of data and the results
58 ;;
59 (defun sum-square-error (f test-data)
60   (reduce #'+
61           (map 'list
62                #'(lambda (test calc)
63                    (square-error (cdr test) calc))
64                test-data (apply-function f test-data))))
65
66 ;; Now we can run the sum of square errors across all equations
67 ;;
68 (map 'list #'(lambda (eq) (sum-square-error eq data))
69      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 68.

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.

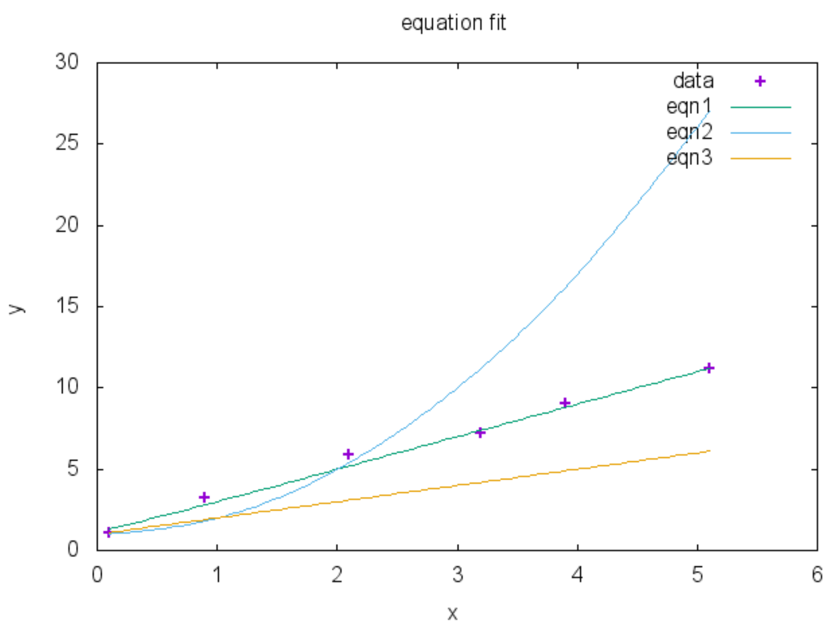
The sum-square-error function takes a function f and applies the `square-error` function to each item in the test data and the corresponding calculated output as calculated by the function f.

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



9 Macro Dispatching Characters

10 Macros

11 Multimethods

12 CLOS

13 Packages

14 Standard Libraries

15 Important Libraries