Writing More ‘Fluent’ Lisp

Samuel Flint
Computer Science and Engineering
University of Nebraska Lincoln
August 25, 2019

In this document, I provide advice on how to improve your programming style in Lisp based on my experience grading your homework so far.

General Style

• Variable names can be long and descriptive. They should never be in CamelCase, instead they should be separated-with-dashes.

• Write code on multiple lines. While

  \( \text{(defun avg } (l) (/(\text{reduce #’}+ l) (\text{length} l))) \)

is fine, it is not a particularly good habit to be in. Instead, try indenting your code:

  \( \text{(defun average } \text{list)} \)
  \( \text{(/} \text{(reduce #’}+ \text{list)} \)
  \( \text{)(length list))) \)

Because, in Lisp, indentation is communicates the ‘structure’ of the code and can dramatically improve readability. If you do not want to manually indent your code, use C-M-q in Emacs.

Lispy Mechanics

• Instead of \(+ \text{var 1}\) or \(- \text{var 1}\) use \((1+ \text{var})\) and \((1- \text{var})\), respectively.

• Be judicious with your control flow constructs.
  – Use \textbf{if} when you have \textit{two} cases, a positive case and a negative case.
  – Use \textbf{when} when you have only a positive case.
  – Use \textbf{unless} when you have only a negative case.
  – Use \textbf{case} if you are checking to see if something matches one of several atomic options (like switch/-
case in C).
  – Use \textbf{cond} in any case where you have more options.

• There are several forms of equal: \texttt{equal, eq, eql, equalp} and \texttt{=}.
  – \texttt{eq}: The two objects are at the same memory location. E.g.:
(eq 'a 'a) ⇒ t
(eq 'a 'b) ⇒ nil
(defvar b 'a) (eq 'a b) ⇒ t

- eql: Either the objects fulfill eq or they are numbers of the same type and value or are the same character.
  (eql 2 2) ⇒ t
  (eql 2 2.0) ⇒ nil
- equal: Numbers and Characters: eql; Symbols: eq; Otherwise: the objects are the same structurally.
  (equal "abc""abc") ⇒ t
  (equal "abc""ABC") ⇒ nil
  (equal '(a (b c)) '(a (b c))) ⇒ t
  (equal '(a b c) '(a (b c))) ⇒ nil
- equalp: equal; if character, then if char-equal (ignores case); if numbers, having the same numerical value (type notwithstanding).
  (equalp #A#a) ⇒ t
  (equalp 2 #(2 0)) ⇒ nil
- =: Only to be used for numbers, follows eql.
- string=: Only to be used for strings. If you need to compare the equality of strings, use this

• let, let* – These are used to introduce bindings and restrict their lexical scoping. Use this form instead of setf at the start of a function. let* performs its bindings serially, so a later binding can rely on the value of an earlier binding.

  (let ((a 1)
     (b 2))
    (+ a b))

  (let* (((a 1)
     (b (1+ a)))
    (+ a b))

  • do, do*, dolist, dotimes – do and do* work similarly, with the starred version binding in parallel. Syntax, generally is of the following form:

    (do ((variable-1 init-form update-form)
    (variable-2 init-form update-form))
    (termination-condition return-value)
    code-here)

  dolist is exactly as it says, it does an action for each element of a list, e.g.,

    (dolist (variable list return-value)
      code-here)

Likewise, for dotimes with an n instead of list.

• loop – Avoid pretty generally, it is hard to debug, and un-lispy.

• collect – If you must use loop, you are likely using it for the sake of collect. Instead of something like:
(let (vals)
    (loop for i from n to m
      do (push i vals))
    (reverse vals))

use:

(loop for i from n to m collect i)

However, there are other uses for collect.

• The Higher-Order Functions:
  
  – map variants – Use these to apply a function to each element of a list (or lists) in turn. E.g.,

  (mapcar #'1+ '(1 2 3 4 5 6 7 8 9 10))

  – reduce – When you have a list of a single type of data reduce allows you to reduce the list into a
   single element using some binary function (i.e., a function that takes two arguments). For instance,
   given a function function-name that produces a list of integers, (reduce #'+ function-name) will
   provide the sum of the list returned.

  – remove-if, remove-if-not, complement – Avoid using remove-if-not, instead, use complement,
   for example, instead of

   (remove-if-not #'evenp list)

   use

   (remove-if (complement #'evenp) list)

   (the existence of oddp notwithstanding).

  – funcall, apply – If you are writing a function that accepts a function as an argument, use either
   funcall or apply to use the passed function.

• lambda – Use this construct, which a has syntax similar to that of defun to define anonymous functions.
You may find this useful in reduce, remove-if or map. E.g.,

(lambda (n)
  (if (oddp n)
    (- n)
    n))