Lisp Style Tips for the Beginner

Heinrich Taube

Composition/Theory -- School of Music -- University of Illinois
hkt@cmp-nxt.music.uiuc.edu, hkt@zkm.de, hkt@ccrma.stanford.edu

---

This document is an informal compendium of beginner's tips on how to develop an efficient, legible style of Lisp coding.

Evaluation

- Avoid using `eval`. The evaluation process is built into Lisp so there is almost no reason to call the evaluator yourself. If you use `eval` to implement something then stop and reconsider the problem because you are almost certainly going about it the wrong way.
- Beware of macros. They are a very important feature of Lisp but a poorly implemented macro can cause bugs that are very difficult for a beginner to solve. Avoid writing macros until you understand how the Lisp reader and evaluator work. Never write a macro to make things "more efficient".

Functions

- Don't implement a Lisp function as you would a C or Pascal program. In general, keep each Lisp function as small as possible. A function should accomplish only one task. By implementing many small functions instead of a few large ones you increase the clarity, modularity and reusability of your code.
- Before you implement a function, make sure that Lisp doesn't already provide it! Common Lisp provides hundreds of functions, macros and special forms, acquaint yourself with relevant sections in Common Lisp: the Language (2nd Edition) before you start a programming task.
- If you implement a function that returns more than one result, uses `values` rather than returning the results in a list. If your function returns no results (like a void in C), use `(values)` to return no values.
- Don't add declarations to your code until you are a seasoned Lisp programmer and you are absolutely certain how the variables will be used.
- Avoid writing huge `case` or `cond` statements. Consider other strategies such as data-driven programming or hash-tables.

Symbols and Variables

- Lisp programmers tend to use just lower case letters. Use the hyphen to form multi-word symbols, as in `multi-word-symbol`. Lisp tends to be verbose, so get used to typing long symbol names!
- Delimit global variable names with `*`, as in `*standard-output*`.
- Delimit global constants with `+`, as in `+default-number+`.
- Use `setf` rather than `setq`. `setf` stands for SET Field. `setf` is a more general form of `setq` that sets almost anything you can "point to", or reference, in Lisp. This includes variables, array locations, list elements, hash table entries, structure fields, and object slots.

Lists

- Use `first`, `rest`, `second`, etc. rather than their `car`, `cdr`, `cadr` equivalents.
- Beware of destructive list operations. They are efficient but can be the source of obscure bugs in your code.
- Remember that `length` must follow pointers to find the end of a list. If you refer to the length of something more than once, save the value in a variable rather than calling `length` several times.
- Remember that `append` copies its arguments. Avoid using `append` inside a loop to add elements to the back of a list. Use the `collect` clause in `loop`, or `push` elements onto a list and then `nreverse` the list to return the original ordering.

  Bad:

```
(let ((result ()))
  (dolist (x list)
    (setf result (append result (list x))))
  result)
```

  Better:

```
(let ((result ()))
  (dolist (x list)
    (push x result))
  (nreverse result))
```

  Best:

```
 (loop for x in list collect x)
```

- Remember that `copy` only copies the outer-most level of a list. Use `copy-tree` to copy all levels of a list.

Conditionals

- Use `when` and `unless` if there is no else clause and you do not depend on the value of the conditional expression. Otherwise, use `if` when the conditional expression is simple, else use `cond`. Always supply a final `t` clause for `cond`.
- Lisp programmers often use the functions `and` and `or` to implement simple conditional evaluation. For example,

```
(and x (setf y t))
```

  is equivalent to

```
(when x
  (setf y t))
```

  and

```
(or x (setf y t))
```

  is equivalent to

```
(unless x
  (setf y t))
```

Iteration

Avoid using `do`, which is almost impossible to read. `dotimes` and `dolist` are fine for simple iteration. If you need to save, store or modify results of an iteration, then `loop` is probably the most legible and efficient construct to use. Some Lisp programmers avoid `loop` because it does not "look like lisp". Loop is nevertheless a convenient and very powerful facility. After 10 minutes of working with `do`, most Lisp programmers are glad to have `loop` around!

Comments

Provide them. Document overall functionality and explain sections of code that are a bit tricky (you will forget how you implemented something in about 2 week's time.) Lisp provides two different types of comments. The semi-colon `;` introduces a line-oriented comment. Lisp ignores whatever follows a semi-colon until the end of the line that the semi-colon is on. The semi-colon does not have to be the first character in the line. Here are two examples:

```
; this is a comment
(abs x) ; need absolute value here!
```

By convention, Lisp programmers distinguish between one, two or three semi-colon comments. A comment introduced by a single semi-colon explains code on the same line as the comment. Two semi-colons mean that the comment is a description about the state of the program at that point, or an explanation of the next section of code. A two semi-colon comment should start at the same indentation column as the code it documents. A three semi-colon comment provides a global explanation and always starts at the left margin. Here is an example containing all three types:

```
;;;  the next 20 functions do various sorts of frobbing
(defun frob1 (num)
  ;; return double frob of num
  (let ((tmp (random num)))      ; breaks if 0, fix!
    (double-frob tmp num :with-good-luck t)))
```

Block comments are made by placing text within `#|` and `|#`. Lisp ignores anything between the balancing delimiters no matter how may lines of text are included. `#| |#` pairs are often used to comment out large sections of program code in a file or function. For example:

```
#|
;;; i think this function is obsolete.
(defun frob2 (list)
  (frob-aux (first list)))
|#
```

comments out a function definition that is no longer used.

Formatting and Indentation

Poorly formatted Lisp code is difficult to read. The most important prerequisite for legible Lisp code is a simple, consistent style of indentation. Some text editors, such as Emacs or Fred, understand Lisp syntax and will automatically perform this task for you. Other text editors, such as NeXTStep's Edit.app, have no understanding of Lisp beyond parentheses matching. Even if you use a text editor that cannot perform Lisp indentation, you should take the time to format your code properly. Here are a few simple rules to follow:

- If your editor supports multiple fonts, always display Lisp code in a fixed-width family like Courier.
- Avoid writing lines longer than 70 characters. Don't assume your reader can shape a window as large as you can, and wrap-around text is almost impossible to read on hard-copy.
- Indent forms in the body of a `defun`, `defmacro` or `let` clause two spaces to the right of the column in which the clause starts. In the following example, both forms in the `defun` are indented two columns. The forms in the body of the let are indented two columns to the right of where the `let` starts:

  ```
  (defun foo (a b &aux c)
    (setf c (* a b))
    (let ((d 1)
          (e (if (zerop b) t nil)))
      (check-compatibility d c)
      (foo-aux a b c d e)))
  ```

- If the arguments to a function occupy more than a single line, indent subsequent lines to the same column position as the first argument. In the following example the indentation clearly shows that there are three arguments to `compute-closure`.

  ```
  (setf s (compute-closure this-function
                           (list other-function my-method)
                           56))
  ```

- Don't use tab to indent and don't close parenthesis on a single line. C style formatting looks silly in Lisp. The example code:

  ```
  (defun my_Foo (x)
      (let ((y (* x 5))
           )
          (values y x)
      )
  )
  ```

  looks much better (to a Lisp programmer) when formatted so:

  ```
  (defun my-foo (x)
    (let ((y (* x 5)))
      (values y x)))
  ```
