Recommended Common Lisp Style  
for the Scone Project

Scott E. Fahlman  
September 24, 2013

This document describes the recommended style for Common Lisp code written by members of the Scone group.

Some of the guidelines below are widely accepted within the Common Lisp community; some reflect principles that I strongly believe in, based on 40+ years of Lisp programming; some are just more or less arbitrary personal preferences. I will try to indicate which of these fall into each category.

But even in the case of arbitrary preferences, it's useful if we all adopt more or less the same style, most of the time. It's important that we all be able to read and build upon one another's code, and that is a lot easier if we all adopt a common coding style for files that will become part of the Scone system. If you inherit someone else's code with the goal of extending or maintaining it, it's nice if you don't feel the need to spend the first week converting everything to your own personal style.

So your code will be more valuable to the group if you follow these guidelines. If you have some deep religious objection to one of these rules, let's discuss that. I've changed my mind about some of these rules over the years, and it could happen again.

The style recommended here is already in use (more or less) in the Scone engine and the other Scone-related Lisp code that I've written, though I won't claim that I've been 100% consistent.

# The Google Common Lisp Style Guide

Google has put out a set of [style guidelines for Common Lisp](https://google.github.io/styleguide/lispguide.xml) that are very close to what I would recommend. I think this is not pure coincidence, but a case of common ancestry. Google itself doesn't use much Common Lisp, but I believe that these guidelines came from ITA Software, whose Common Lisp code powers most of the airline reservation sites on the Internet. Google acquired them in 2011. ITA was founded by people from the MIT-AI Lab, where I learned Lisp. I imported much of the MIT-AI style to CMU.

So instead of writing a style guide from scratch, I will incorporate the Google guidelines by reference. But there are a number of places where I disagree with that document or want to add something, so this document will describe the Scone-project changes and additions to that document. This will follow the major section headings in Google's table of contents.

# Meta-Guide

This is a research project, and we don't need to be as rigid about application of the rules as a big company has to be. If you have some reason to deviate from one of the guidelines in a given case, go ahead. You don't need a note from your supervisor. But if you do this repeatedly, let's discuss it.

Do pay attention to the must/should/may distinctions in the Google document. Even if we don't apply these terms rigidly, they are a good guideline to indicate what rules the Google people think are most important (and in most cases I agree with them).

# General Guidelines

I agree with most of what is said here.

## Priorities

I would take Google's ordering as a good default. Note that our "customers" are the potential users of Scone in the research community and people using Scone in applications.

However, there are some times when runtime efficiency is most important – often just in a small stretch of code implementing the crucial inner loops where the system will spend most of its time. In Scone, for example, upscan, downscan, and a few other "pseudo-parallel" marker-passing operations fit this category. Note that I indicate these parts of the code explicitly, both by comments and by optimize declarations..

When you are going for maximum efficiency, the most important thing is usually to avoid consing, though of course the proper choice of algorithms can have an even bigger effect.

And let me agree explicitly with the admonition to avoid premature optimization. In almost all cases it's best to get something running, see where the bottlenecks are, and then optimize those bits. If there are no bottlenecks that matter, you’re done.

That said, it probably is a good idea to develop a Lisp coding style that is not gratuitously inefficient if there is no benefit in clarity.

## Architecture

A good model to keep in mind is that Lisp is the ideal tool for implementing specialized languages. Actually, in Lisp there's little distinction between a "specialized language" and a library or set of libraries that provide a domain-specific set of abstractions. Usually there is no need for specialized syntax, or only minimal differences, such as Scone's curly-brace notation.

You can often think of your application as implementing a layered series of little languages, each of which provides a new abstraction or new set of capabilities. In Scone, for example, there is a language for creating basic elements and accessing their fields; a language on top of that for allocating and using marker bits; a language for the various marker scans; and then higher level languages for queries and so on. If each of these languages is well designed, reasonably general, and reasonably complete – i.e. not too specialized for the task of the moment – this will allow easy experimentation and extension later. And it's easier to document each little language than a big amorphous tangle of code.

# Formatting

## Spelling and Grammar

Comments should be proper English sentences (or sometimes just phrases), following the conventional rules of capitalization and punctuation. The goal should make things as clear and easy for the reader as possible, not to show off how cool and quirky you are. In particular, do not use jargon, abbreviations, or Twitter-ese that may not be immediately obvious to the reader. Correct spelling and grammar are also strongly encouraged, if you’re capable of producing this – and if not, you may have some serious problems producing acceptable publications.

## Line Length

I disagree with the Google recommendation of 100 as the maximum line length. In Scone we should stick to a maximum of 80 columns, so please break your lines accordingly. There are lots of examples in the Scone engine code showing how to break lines so that the nesting structure of the code is still evident.

This is not just an old-fashioned preference dating from the days when character-based terminals were limited to 80 characters. Occasionally it is useful to print Lisp code files in a fixed-width font, and lines longer than 80 characters can cause problems of readability and/or wrapping. Also, even on hi-res displays, it is often useful to have three Lisp windows open side-by-side (one for writing new code, one for copying old code, and one for a running Lisp used for testing things), and again lines longer than 80 characters can cause problems. I also believe that longer lines make code files harder to read quickly – the eye tends to get lost if the lines are too long.

One exception to the 80-column limit is when you have a long Lisp string and don't want to mess it up by inserting newlines in the middle. But note that if the string is a template in a format statement, you can ~<newline character> to insert an "invisible" newline in the string.

Rarely, there is code so deeply nested that you just can't stay in bounds and still indent it properly. In these cases, give priority to making the nesting clear, and make the code line as long as necessary – but think about reorganizing your code.

## File Header

Code and KB files that are part of the Scone distribution should all have a standard file header with copyright and licensing statements and acknowledgment of the sponsors. This header has changed form time to time, but assume that the other files in the current release have the correct header unless I have told you otherwise.

## Documentation

Please carefully read and follow the information in the Google guidelines about doc-strings.

As you see in the Google examples, there is a tradition in Lisp of capitalizing Lisp names that appear in doc-strings. This came about because in the old days there was no good way to put multiple fonts or other highlighting in code, so capitalization was used instead. This did no harm because Lisp is a case-insensitive language (a slight over-simplification, but never mind that for now…).

I've never seen anyone use the Google-suggested convention of using forward single-quotes to highlight type names within code. That seems confusing to me.

In documents written in a modern text-editor, is preferable to show Lisp names and other code in a distinctive fixed-width font, as is done here: (upscan start-node m).

## Comment Semicolons

As a rule, use triple-semicolons for comments that are between top-level Lisp forms, and that are meant to be flush with the left margin.

If a function has a proper doc-string, there is no need to repeat that information in a top-level comment above the function definition. The doc-string should be used to describe the function's *contract*: what the user needs to know about how to call the function, what it does, and what it returns. Other comments – general descriptions of control flow, how multiple function work together, notes on changes planned for the future, and anything else that might help some future user or maintainer should be in a triple-quote comment preceding the function, or somewhere nearby.

I occasionally use quad-semis, as Google recommends, for file and section headings. But I am not very consistent about this, and usually use more visible dividers for major sections – see the Scone engine code.

Double-semi comments, indented with the code it describes, are quite useful to describe what particular chunks of code in a function are trying to do. Also use a double-semicolon comment to describe each field in a structure or object.

Since-semicolon comments floating to the right of a line of code are messy and confusing. Don't use them. A properly indented double-semi comment is almost always clearer.

I have some customized Emacs functions that are useful in cleaning up the line breaks in a multi-line comment after editing.

## Symbol Naming

The Google guidelines are good here. Lisp allows users to create long, meaningful names for symbols and functions. Use that capability to make your code as clear as possible, even if that requires a bit of extra typing.

An exception: In the Scone engine, I have adopted the convention of using e for element arguments and m for marker arguments. If there are several such arguments to a function, and nothing more interesting to call them, I use e1, e2, e3 and m1, m2, m3, etc. There are so many element and marker args, that this abbreviation seems warranted.

## Predicate Names

The usual Common Lisp convention is to put a "p" at the end of the name of any predicate function, such as elementp. I have always preferred the Scheme convention of using a "?" instead of a "p", as in element?, and I have used this "?" convention in all Scone code. This is a quirk, but please follow this convention in Scone code.

## Mostly Functional Style

This bit of the Google document is complete nonsense. Lisp supports many styles of programming, including a functional, side-effect-free style. That style is occasionally useful, but should certainly not be your default choice for most coding. Feel free to use all the side-effects you want.

## Assignment

I prefer using multiple setq or setf forms, rather than packing a number of assignments into a single form.

I also prefer to use setq whenever possible, saving setf for times when you are doing something more complicated than a simple setq – that is, when you are modifying a "generalized location" represented by an access expression more complicated than a simple variable name.

## Avoid NCONC

This is good advice, since any use of destructive modification on lists is risky – it's very easy to forget that someone outside the current function might also have hold of your list, and any destructive modification can confuse the other program. This is a frequent source of very subtle and hard-to-find bugs, since the manifestation might be far away in space and time from the source of the problem.

Having said that, let me mention that a very frequent idiom is to cons up a list in reverse order, adding new elements to the front, and then to nreverse the result before returning the list or passing it to anyone outside the current function. For example:

(defun list-integers-up-to (n)  
 "N is an integer. Return a list of all the integers less  
 than N, in ascending order starting with 0."  
 ;; RESULT starts as an empty list.  
 (let ((result nil))  
 ;; For all I from 0 to N, push the value of I onto  
 ;; the result list.  
 (dotimes (i n)  
 (push i result))  
 ;; The list is backwards. NREVERSE flips it around  
 ;; efficiently. Then we return the result. Since  
 ;; we created this list from scratch, we know that  
 ;; destructively flipping it can't affect anything  
 ;; outside of this function.  
 (nreverse result))