Riastradh's Lisp Style Rules

(note: i haven't fully checked over whether my text-munging screws up the examples. also need to fix the syntax highlighting)

This is a guide to Lisp style, written by Taylor R. Campbell, to describe the standard rules of Lisp style as well as a set of more stringent rules for his own style. This guide should be useful for Lisp in general, but there are [or will be in the final draft] parts that are focussed on or specific to Scheme or Common Lisp.

This guide is written primarily as a collection of rules, with rationale for each rule. (If a rule is missing rationale, please inform the author!) Although a casual reader might go through and read the rules without the rationale, perhaps reasoning that reading of the rationale would require more time than is available, such a reader would derive little value from this guide. In order to apply the rules meaningfully, their spirit must be understood; the letter of the rules serves only to hint at the spirit. The rationale is just as important as the rules.

There are many references in this document to `Emacs`, `GNU Emacs`, `Edwin`, and so on. In this document, `Emacs` means any of a class of editors related in design to a common ancestor, the `EMACS` editor macros written for `TECO` on `ITS` on the `PDP-10` in the middle of the nineteen seventies. All such editors – or `all Emacsen`, since `Emacsen` is the plural of `Emacs` – have many traits in common, such as a very consistent set of key bindings, extensibility in Lisp, and so on. `GNU Emacs` means the member of the class of editors collectively known as Emacsen that was written for the GNU Project in the middle of the nineteen eighties, and which is today probably the most popular Emacs. `Edwin` is MIT Scheme's Emacs, which is bundled as part of MIT Scheme, and not available separately. There are other Emacsen as well, such as Hemlock and Climacs, but as the author of this document has little experience with Emacsen other than GNU Emacs and Edwin, there is little mention of other Emacsen.

This guide is a work in progress. To be written:

- Indentation rules for various special operators.
- Philosophical rambling concerning naming.
- Rules for breaking lines.
- Many more examples.
- A more cohesive explanation of the author's principles for composing programs, and their implications.
- Rules for writing portable code.
- Some thoughts concerning extensions to the lexical syntax.
- Rules for writing or avoiding macros.
- Some unfinished rationale.
- More on documentation.
- The `Dependencies` subsection of the `General Layout` section should be put in a different section, the rest of which has yet to be written, on organization of programs, module systems, and portable code.

Feedback is welcome; address any feedback by email to the host mumble.net's user `campbell`, or by `IRC` to Riastradh in the #scheme channel on Freenode (irc.freenode.net). Feedback includes reports of typos, questions, requests for clarification, and responses to the rationale, except in the case of round brackets versus square brackets, the argument surrounding which is supremely uninteresting and now not merely a dead horse but a rotting carcass buzzing with flies and being picked apart by vultures.

As this document has grown, the line between standard Lisp rules and the author's own style has been blurred. The author is considering merging of the partition, but has not yet decided on this with certainty. Opinions on the subject are welcome – is the partition still useful to keep the author's biases and idiosyncrasies out of the standard rules, or has the partition with its arbitrary nature only caused disorganization of the whole document?

Unfortunately, this document is entirely unscientific. It is at best a superstition or philosophy, but one that the author of this document has found to have improved his programs. Furthermore, the author is somewhat skeptical of claims of scientific analyses of these matters: analyzing human behaviour, especially confined to the set of excellent programmers who often have strong opinions about their methods for building programs, is a very tricky task.

Standard Rules

These are the standard rules for formatting Lisp code; they are repeated here for completeness, although they are surely described elsewhere. These are the rules implemented in Emacs Lisp modes, and auxiliary utilities such as Paredit.

The rationale given here is merely the author's own speculation of the origin of these rules, and should be taken as nothing more than it. The reader shall, irrespective of the author's rationale, accept the rules as sent by the reader's favourite deity, or Cthulhu if no such deity strikes adequate fear into the heart of the reader.

Parentheses

Terminology

This guide avoids the term *parenthesis* except in the general use of *parentheses* or *parenthesized*, because the word's generally accepted definition, outside of the programming language, is a statement whose meaning is peripheral to the sentence in which it occurs, and **not** the typographical symbols used to delimit such statements.

The balanced pair of typographical symbols that mark parentheses in English text are *round brackets*, i.e. (and). There are several other balanced pairs of typographical symbols, such as *square brackets* (commonly called simply `brackets` in programming circles), i.e. [and]; *curly braces* (sometimes called simply `braces`), i.e. { and }; *angle brackets* (sometimes `brokets` (for `broken brackets`)), i.e. < and >.

In any balanced pair of typographical symbols, the symbol that begins the region delimited by the symbols is called the *opening bracket* or the *left bracket*, such as (or [or { or <. The symbol that ends that region is called the *right bracket* or the *closing bracket*, such as > or } or] or).

Spacing

If any text precedes an opening bracket or follows a closing bracket, separate that text from that bracket with a space. Conversely, leave no space after an opening bracket and before following text, or after preceding text and before a closing bracket.

Unacceptable:

```
(foo(bar baz)quux)
(foo ( bar baz ) quux)
```

Acceptable:

```
(foo (bar baz) quux)
```

Rationale: This is the same spacing found in standard typography of European text. It is more aesthetically pleasing.

Line Separation

Absolutely do **not** place closing brackets on their own lines.

Unacceptable:

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1
                     )
           )
      )
  )
)
```

Acceptable:

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1)))))
```

Rationale: The parentheses grow lonely if their closing brackets are all kept separated and segregated.

Exceptions to the Above Rule Concerning Line Separation

Do not heed this section unless you know what you are doing. Its title does **not** make the unacceptable example above acceptable.

When commenting out fragments of expressions with line comments, it may be necessary to break a line before a sequence of closing brackets:

```
(define (foo bar)
  (list (frob bar)
        (zork bar)
        ;; (zap bar)
        ))
```

This is acceptable, but there are other alternatives. In Common Lisp, one can use the read-time conditional syntax, `#+` or `#-`, with a feature conditional that is guaranteed to be false or true – `#+(OR)` or `#-(AND)` `--`; for example,

```
(define (foo bar)
  (list (frob bar)
        (zork bar)
        #+(or) (zap bar))).
```

Read-time conditionals are expression-oriented, not line-oriented, so the closing brackets need not be placed on the following line. Some Scheme implementations, and `SRFI 62`, also support expression comments with `#;`, which are operationally equivalent to the above read-time conditionals for Common Lisp:

```
(define (foo bar)
  (list (frob bar)
        (zork bar)
        #;
         (zap bar)))
```

The expression is placed on another line in order to avoid confusing editors that do not recognize S-expression comments; see the section titled `Comments` below for more details. However, the `#;` notation is not standard – it appears in neither the `IEEE 1178` document nor in the `R5RS` –, so line comments are preferable for portable Scheme code, even if they require breaking a line before a sequence of closing brackets.

Finally, it is acceptable to break a line immediately after an opening bracket and immediately before a closing bracket for very long lists, especially in files under version control. This eases the maintenance of the lists and clarifies version diffs. Example:

```
(define colour-names         ;Add more colour names to this list!
  '(
    blue
    cerulean
    green
    magenta
    purple
    red
    scarlet
    turquoise
    ))
```

Parenthetical Philosophy

The actual bracket characters are simply lexical tokens to which little significance should be assigned. Lisp programmers do not examine the brackets individually, or, Azathoth forbid, count brackets; instead they view the higher-level structures expressed in the program, especially as presented by the indentation. Lisp is not about writing a sequence of serial instructions; it is about building complex structures by summing parts. The composition of complex structures from parts is the focus of Lisp programs, and it should be readily apparent from the Lisp code. Placing brackets haphazardly about the presentation is jarring to a Lisp programmer, who otherwise would not even have seen them for the most part.

Indentation and Alignment

The operator of any form, i.e. the first subform following the opening round bracket, determines the rules for indenting or aligning the remaining forms. Many names in this position indicate special alignment or indentation rules; these are special operators, macros, or procedures that have certain parameter structures.

If the first subform is a non-special name, however, then if the second subform is on the same line, align the starting column of all following subforms with that of the second subform. If the second subform is on the following line, align its starting column with that of the first subform, and do the same for all remaining subforms.

In general, Emacs will indent Lisp code correctly. Run `C-M-q (indent-sexp)` on any code to ensure that it is indented correctly, and configure Emacs so that any non-standard forms are indented appropriately.

Unacceptable:

```
(+ (sqrt -1)
  (* x y)
  (+ p q))

(+
   (sqrt -1)
   (* x y)
   (+ p q))
```

Acceptable:

```
(+ (sqrt -1)
   (* x y)
   (+ p q))

(+
 (sqrt -1)
 (* x y)
 (+ p q))
```

Rationale: The columnar alignment allows the reader to follow the operands of any operation straightforwardly, simply by scanning downward or upward to match a common column. Indentation dictates structure; confusing indentation is a burden on the reader who wishes to derive structure without matching parentheses manually.

Non-Symbol Indentation and Alignment

The above rules are not exhaustive; some cases may arise with strange data in operator positions.

Lists

Unfortunately, style varies here from person to person and from editor to editor. Here are some examples of possible ways to indent lists whose operators are lists:

Questionable:

```
((car x)                            ;Requires hand indentation.
   (cdr x)
   foo)

((car x) (cdr x)                    ;GNU Emacs
 foo)
```

Preferable:

```
((car x)                            ;Any Emacs
 (cdr x)
 foo)

((car x) (cdr x)                    ;Edwin
         foo)
```

Rationale: The operands should be aligned, as if it were any other procedure call with a name in the operator position; anything other than this is confusing because it gives some operands greater visual distinction, allowing others to hide from the viewer's sight. For example, the questionable indentation

```
((car x) (cdr x)
 foo)
```

can make it hard to see that `FOO` and `(CDR X)` are both operands here at the same level. However, GNU Emacs will generate that indentation by default. (Edwin will not.)

Strings

If the form in question is meant to be simply a list of literal data, all of the subforms should be aligned to the same column, irrespective of the first subform.

Unacceptable:

```
("foo" "bar" "baz" "quux" "zot"
       "mumble" "frotz" "gargle" "mumph")
```

Questionable, but acceptable:

```
(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
   3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1 6 9 3 9 9 3)
```

Acceptable:

```
("foo" "bar" "baz" "quux" "zot"
 "mumble" "frotz" "gargle" "mumph")

("foo"
 "bar" "baz" "quux" "zot"
 "mumble" "frotz" "gargle" "mumph")
```

Rationale: Seldom is the first subform distinguished for any reason, if it is a literal; usually in this case it indicates pure data, not code. Some editors and pretty-printers, however, will indent unacceptably in the example given unless the second subform is on the next line anyway, which is why the last way to write the fragment is usually best.

Names

Naming is subtle and elusive. Bizarrely, it is simultaneously insignificant, because an object is independent of and unaffected by the many names by which we refer to it, and also of supreme importance, because it is what programming – and, indeed, almost everything that we humans deal with – is all about. A full discussion of the concept of name lies far outside the scope of this document, and could surely fill not even a book but a library.

Symbolic names are written with English words separated by hyphens. Scheme and Common Lisp both fold the case of names in programs; consequently, camel case is frowned upon, and not merely because it is ugly. Underscores are unacceptable separators except for names that were derived directly from a foreign language without translation.

Unacceptable:

```
XMLHttpRequest
foreach
append_map
```

Acceptable:

```
xml-http-request
for-each
append-map
```

Funny Characters

There are several different conventions in different Lisps for the use of non-alphanumeric characters in names.

Scheme

Question Marks: Predicates

Affix a question mark to the end of a name for a procedure whose purpose is to ask a question of an object and to yield a boolean answer. Such procedures are called `predicates`. Do not use a question mark if the procedure may return any object other than a boolean.

Examples: `pair?` `procedure?` `proper-list?`

Non-examples: member assoc any every

Pronounce the question mark as if it were the isolated letter `p`. For example, to read the fragment `(PAIR? OBJECT)` aloud, say: `pair-pee object.`

Exclamation Marks: Destructive Operations

Affix an exclamation mark to the end of a name for a procedure (or macro) whose primary purpose is to modify an object. Such procedures are called `destructive`.

Examples: set-car! append!

Avoid using the exclamation mark willy nilly for just **any** procedure whose operation involves any kind of mutation or side effect; instead, use the exclamation mark to identify procedures that exist **solely** for the purpose of destructive update (e.g., `SET-CAR!`), or to distinguish a destructive, or potentially destructive (in the case of linear-update operations such as `APPEND!`), variant of a procedure of which there also exists a purely functional variant (e.g., `APPEND`).

Pronounce the exclamation mark as `bang`. For example, to read the fragment `(APPEND! LIST TAIL)` aloud, say: `append-bang list tail`.

Asterisks: Variants, Internal Routines, Mutable Globals

Affix an asterisk to the end of a name to make a variation on a theme of the original name.

Example: `let -> let*`

Prefer a meaningful name over an asterisk; the asterisk does not explain what variation on the theme the name means.

Affix an asterisk to the beginning of a name to make an internal routine for that name. Again, prefer a meaningful name over an asterisk.

Affix asterisks to the beginning and end of a globally mutable variable. This allows the reader of the program to recognize very easily that it is badly written!

`WITH-` and `CALL-WITH-`: Dynamic State and Cleanup

Prefix `WITH-` to any procedure that establishes dynamic state and calls a nullary procedure, which should be the last (required) argument. The dynamic state should be established for the extent of the nullary procedure, and should be returned to its original state after that procedure returns.

Examples: `with-input-from-file` `with-output-to-file`

Exception: Some systems provide a procedure `(WITH-CONTINUATION <continuation> <thunk>)`, which calls `<thunk>` in the given continuation, using that continuation's dynamic state. If `<thunk>` returns, it will return to `<continuation>`, not to the continuation of the call to `WITH-CONTINUATION`. This is acceptable.

Prefix `CALL-WITH-` to any procedure that calls a procedure, which should be its last argument, with some arguments, and is either somehow dependent upon the dynamic state or continuation of the program, or will perform some action to clean up data after the procedure argument returns. Generally, `CALL-WITH-` procedures should return the values that the procedure argument returns, after performing the cleaning action.

Examples:

- `CALL-WITH-INPUT-FILE` and `CALL-WITH-OUTPUT-FILE` both accept a pathname and a procedure as an argument, open that pathname (for input or output, respectively), and call the procedure with one argument, a port corresponding with the file named by the given pathname. After the procedure returns, `CALL-WITH-INPUT-FILE` and `CALL-WITH-OUTPUT-FILE` close the file that they opened, and return whatever the procedure returned.
- `CALL-WITH-CURRENT-CONTINUATION` is dependent on the continuation with which it was called, and passes as an argument an escape procedure corresponding with that continuation.
- `CALL-WITH-OUTPUT-STRING`, a common but non-standard procedure definable in terms of `OPEN-OUTPUT-STRING` and `GET-OUTPUT-STRING` from `SRFI 6 (Basic String Ports)`, calls its procedure argument with an output port, and returns a string of all of the output written to that port. Note that it does not return what the procedure argument returns, which is an exception to the above rule.

Generally, the distinction between these two classes of procedures is that `CALL-WITH-...` procedures should not establish fresh dynamic state and instead pass explicit arguments to their procedure arguments, whereas `WITH-...` should do the opposite and establish dynamic state while passing zero arguments to their procedure arguments.

Comments

Write heading comments with at least four semicolons; see also the section below titled `Outline Headings`.

Write top-level comments with three semicolons.

Write comments on a particular fragment of code before that fragment and aligned with it, using two semicolons.

Write margin comments with one semicolon.

The only comments in which omission of a space between the semicolon and the text is acceptable are margin comments.

Examples:

```
;;;; Frob Grovel

;;; This section of code has some important implications:
;;;   1. Foo.
;;;   2. Bar.
;;;   3. Baz.

(define (fnord zarquon)
  ;; If zob, then veeblefitz.
  (quux zot
        mumble             ;Zibblefrotz.
        frotz))
```

Riastradh's Non-Standard Rules

Three principles guide this style, roughly ordered according to descending importance:

1. The purpose of a program is to describe an idea, and not the way that the idea must be realized; the intent of the program's meaning, rather than peripheral details that are irrelevant to its intent, should be the focus of the program, **irrespective** of whether a human or a machine is reading it. [It would be nice to express this principle more concisely.]
2. The sum of the parts is easier to understand than the whole.
3. Aesthetics matters. No one enjoys reading an ugly program.

General Layout

This section contains rules that the author has found generally helpful in keeping his programs clean and presentable, though they are not especially philosophically interesting.

Contained in the rationale for some of the following rules are references to historical limitations of terminals and printers, which are now considered aging cruft of no further relevance to today's computers. Such references are made only to explain specific measures chosen for some of the rules, such as a limit of eighty columns per line, or sixty-six lines per page. There is a real reason for each of the rules, and this real reason is not intrinsically related to the historical measures, which are mentioned only for the sake of providing some arbitrary measure for the limit.

File Length

If a file exceeds five hundred twelve lines, begin to consider splitting it into multiple files. Do not write program files that exceed one thousand twenty-four lines. Write a concise but descriptive title at the top of each file, and include no content in the file that is unrelated to its title.

Rationale: Files that are any larger should generally be factored into smaller parts. (One thousand twenty-four is a nicer number than one thousand.) Identifying the purpose of the file helps to break it into parts if necessary and to ensure that nothing unrelated is included accidentally.

Top-Level Form Length

Do not write top-level forms that exceed twenty-one lines, except for top-level forms that serve only the purpose of listing large sets of data. If a procedure exceeds this length, split it apart and give names to its parts. Avoid names formed simply by appending a number to the original procedure's name; give meaningful names to the parts.

Rationale: Top-level forms, especially procedure definitions, that exceed this length usually combine too many concepts under one name. Readers of the code are likely to more easily understand the code if it is composed of separately named parts. Simply appending a number to the original procedure's name can help only the letter of the rule, not the spirit, however, even if the procedure was taken from a standard algorithm description. Using comments to mark the code with its corresponding place in the algorithm's description is acceptable, but the algorithm should be split up in meaningful fragments anyway.

Rationale for the number twenty-one: Twenty-one lines, at a maximum of eighty columns per line, fits in a GNU Emacs instance running in a 24x80 terminal. Although the terminal may have twenty-four lines, three of the lines are occupied by GNU Emacs: one for the menu bar (which the author of this guide never uses, but which occupies a line nevertheless in a vanilla GNU Emacs installation), one for the mode line, and one for the minibuffer's window. The writer of some code may not be limited to such a terminal, but the author of this style guide often finds it helpful to have at least four such terminals or Emacs windows open simultaneously, spread across a twelve-inch laptop screen, to view multiple code fragments.

Line Length

Do not write lines that exceed eighty columns, or if possible seventy-two.

Rationale: Following multiple lines that span more columns is difficult for humans, who must remember the line of focus and scan right to left from the end of the previous line to the beginning of the next line; the more columns there are, the harder this is to do. Sticking to a fixed limit helps to improve readability.

Rationale for the numbers eighty and seventy-two: It is true that we have very wide screens these days, and we are no longer limited to eighty-column terminals; however, we ought to exploit our wide screens not by writing long lines, but by viewing multiple fragments of code in parallel, something that the author of this guide does very often. Seventy-two columns leave room for several nested layers of quotation in email messages before the code reaches eighty columns. Also, a fixed column limit yields nicer printed output, especially in conjunction with pagination; see the section `Pagination` below.

Blank Lines

Separate each adjacent top-level form with a single blank line (i.e. two line breaks). If two blank lines seem more appropriate, break the page instead. Do not place blank lines in the middle of a procedure body, except to separate internal definitions; if there is a blank line for any other reason, split the top-level form up into multiple ones.

Rationale: More than one blank line is distracting and sloppy. If the two concepts that are separated by multiple blank lines are really so distinct that such a wide separator is warranted, then they are probably better placed on separate pages anyway; see the next section, `Pagination`.

Pagination

Separate each file into pages of no more than sixty-six lines and no fewer than forty lines with form feeds (ASCII `#x0C`, or `^L`, written in Emacs with `C-q C-l`), on either side of which is a single line break (but not a blank line).

Rationale: Keeping distinct concepts laid out on separate pages helps to keep them straight. This is helpful not only for the writer of the code, but also for the reader. It also allows readers of the code to print it onto paper without fiddling with printer settings to permit pages of more than sixty-six lines (which is the default number for many printers), and pagination also makes the code easier to navigate in Emacs, with the `C-x [` and `C-x ]` keys (`backward-page` and `forward-page`, respectively). To avoid excessively small increments of page-by-page navigation, and to avoid wasting paper, each page should generally exceed forty lines.

`C-x l` in Emacs will report the number of lines in the page on which the point lies; this is useful for finding where pagination is necessary.

Outline Headings

Use Emacs's Outline Mode to give titles to the pages, and if appropriate a hierarchical structure. Set `outline-regexp` (or `outline-pattern` in Edwin) to `"\f\n;;;;+ "`, so that each form feed followed by an line break followed by at least four semicolons and a space indicates an outline heading to Emacs. Use four semicolons for the highest level of headings in the hierarchy, and one more for each successively nested level of hierarchy.

Rationale: Not only does this clarify the organization of the code, but readers of the code can then navigate the code's structure with Outline Mode commands such as `C-c C-f`, `C-c C-b`, `C-c C-u`, and `C-c C-d` (forward, backward, up, down, respectively, headings).

Dependencies

When writing a file or module, minimize its dependencies. If there are too many dependencies, consider breaking the module up into several parts, and writing another module that is the sum of the parts and that depends only on the parts, not their dependencies.

Rationale: A fragment of a program with fewer dependencies is less of a burden on the reader's cognition. The reader can more easily understand the fragment in isolation; humans are very good at local analyses, and terrible at global ones.

Naming

This section requires an elaborate philosophical discussion which the author is too ill to have the energy to write at this moment.

Compose concise but meaningful names. Do not cheat by abbreviating words or using contractions.

Rationale: Abbreviating words in names does not make them shorter; it only makes them occupy less screen space. The reader still must understand the whole long name. This does not mean, however, that names should necessarily be long; they should be descriptive. Some long names are more descriptive than some short names, but there are also descriptive names that are not long and long names that are not descriptive. Here is an example of a long name that is not descriptive, from SchMUSE, a multi-user simulation environment written in MIT Scheme:

```
frisk-descriptor-recursive-subexpr-descender-for-frisk-descr-env
```

Not only is it long (sixty-four characters) and completely impenetrable, but halfway through its author decided to abbreviate some words as well!

Do not write single-letter variable names. Give local variables meaningful names composed from complete English words.

Rationale: It is tempting to reason that local variables are invisible to other code, so it is OK to be messy with their names. This is faulty reasoning: although the next person to come along and use a library may not care about anything but the top-level definitions that it exports, this is not the only audience of the code. Someone will also want to read the code later on, and if it is full of impenetrably terse variable names without meaning, that someone will have a hard time reading the code.

Give names to intermediate values where their expressions do not adequately describe them.

Rationale: An `expression` is a term that expresses some value. Although a machine needs no higher meaning for this value, and although it should be written to be sufficiently clear for a human to understand what it means, the expression might mean something more than just what it says where it is used. Consequently, it is helpful for humans to see names given to expressions.

Example: A hash table `HASH-TABLE` maps foos to bars; `(HASH-TABLE/GET HASH-TABLE FOO #F)` expresses the datum that HASH-TABLE maps FOO to, but that expression gives the reader no hint of any information concerning that datum. (LET ((BAR (HASH-TABLE/GET HASH-TABLE FOO #F))) ...) gives a helpful name for the reader to understand the code without having to find the definition of HASH-TABLE.

Index variables such as i and j, or variables such as A and D naming the car and cdr of a pair, are acceptable only if they are completely unambiguous in the scope. For example,

```
(do ((i 0 (+ i 1)))
    ((= i (vector-length vector)))
  (frobnicate (vector-ref vector i)))
```

is acceptable because the scope of i is very clearly limited to a single vector. However, if more vectors are involved, using more index variables such as j and k will obscure the program further.

Avoid functional combinators, or, worse, the point-free (or `point-less`) style of code that is popular in the Haskell world. At most, use function composition only where the composition of functions is the crux of the idea being expressed, rather than simply a procedure that happens to be a composition of two others.

Rationale: Tempting as it may be to recognize patterns that can be structured as combinations of functional combinators – say, `compose this procedure with the projection of the second argument of that other one`, or `(COMPOSE FOO (PROJECT 2 BAR))` –, the reader of the code must subsequently examine the elaborate structure that has been built up to obscure the underlying purpose. The previous fragment could have been written `(LAMBDA (A B) (FOO (BAR B)))`, which is in fact shorter, and which tells the reader directly what argument is being passed on to what, and what argument is being ignored, without forcing the reader to search for the definitions of FOO and BAR or the call site of the final composition. The explicit fragment contains substantially more information when intermediate values are named, which is very helpful for understanding it and especially for modifying it later on.

The screen space that can be potentially saved by using functional combinators is made up for by the cognitive effort on the part of the reader. The reader should not be asked to search globally for usage sites in order to understand a local fragment. Only if the structure of the composition really is central to the point of the narrative should it be written as such. For example, in a symbolic integrator or differentiator, composition is an important concept, but in most code the structure of the composition is completely irrelevant to the real point of the code.

If a parameter is ignored, give it a meaningful name nevertheless and say that it is ignored; do not simply call it `ignored`.

In Common Lisp, variables can be ignored with `(DECLARE (IGNORE ...))`. Some Scheme systems have similar declarations, but the portable way to ignore variables is just to write them in a command context, where their values will be discarded, preferably with a comment indicating this purpose:

```
(define (foo x y z)
  x z                         ;ignore
  (frobnitz y))
```

Rationale: As with using functional combinators to hide names, avoiding meaningful names for ignored parameters only obscures the purpose of the program. It is helpful for a reader to understand what parameters a procedure is independent of, or if someone wishes to change the procedure later on, it is helpful to know what other parameters are available. If the ignored parameters were named meaninglessly, then these people would be forced to search for call sites of the procedure in order to get a rough idea of what parameters might be passed here.

When naming top-level bindings, assume namespace partitions unless in a context where they are certain to be absent. Do not write explicit namespace prefixes, such as `FOO:BAR` for an operation `BAR` in a module FOO, unless the names will be used in a context known not to have any kind of namespace partitions.

Rationale: Explicit namespace prefixes are ugly, and lengthen names without adding much semantic content. Common Lisp has its package system to separate the namespaces of symbols; most Schemes have mechanisms to do so as well, even if the RnRS do not specify any. It is better to write clear names which can be disambiguated if necessary, rather than to write names that assume some kind of disambiguation to be necessary to begin with. Furthermore, explicit namespace prefixes are inadequate to cover name clashes anyway: someone else might choose the same namespace prefix. Relegating this issue to a module system removes it from the content of the program, where it is uninteresting.

Comments

Write comments only where the code is incapable of explaining itself. Prefer self-explanatory code over explanatory comments. Avoid `literate programming` like the plague.

Rationale: If the code is often incapable of explaining itself, then perhaps it should be written in a more expressive language. This may mean using a different programming language altogether, or, since we are talking about Lisp, it may mean simply building a combinator language or a macro language for the purpose. `Literate programming` is the logical conclusion of languages incapable of explaining themselves; it is a direct concession of the inexpressiveness of the computer language implementing the program, to the extent that the only way a human can understand the program is by having it rewritten in a human language.

Do not write interface documentation in the comments for the implementation of the interface. Explain the interface at the top of the file if it is a single-file library, or put that documentation in another file altogether. (See the `Documentation` section below if the interface documentation comments grow too large for a file.)

Rationale: A reader who is interested only in the interface really should not need to read through the implementation to pick out its interface; by putting the interface documentation at the top, not only is such a reader's task of identifying the interface made easier, but the implementation code can be more liberally commented without fear of distracting this reader. To a reader who is interested in the implementation as well, the interface is still useful in order to understand what concepts the implementation is implementing.

Example: http://mumble.net/~campbell/scheme/skip-list.scm

In this example of a single-file library implementing the skip list data structure, the first page explains the purpose and dependencies of the file (which are useful for anyone who intends to use it, even though dependencies are really implementation details), and the next few pages explain the usage of skip lists as implemented in that file. On the first page of implementation, `Skip List Structure`, there are some comments of interest only to a reader who wishes to understand the implementation; the same goes for the rest of the file, none of which must a reader read whose interest is only in the usage of the library.

Avoid block comments (i.e. `#| ... |#`). Use S-expression comments (`#;` in Scheme, with the expression to comment on the next line; `#+(OR)` or `#-(AND)` in Common Lisp) to comment out whole expressions. Use blocks of line comments for text.

Rationale: Editor support for block comments is weak, because it requires keeping a detailed intermediate parse state of the whole buffer, which most Emacsen do not do. At the very least, `#|| ... ||#` is better, because most Emacsen will see vertical bars as symbol delimiters, and lose trying to read a very, very long symbol, if they try to parse #| ... |#, whereas they will just see two empty symbols and otherwise innocuous text between them if they try to parse `#|| ... ||#`. In any case, in Emacs, `M-x comment-region RET`, or `M-; (comment-dwim)`, is trivial to type.

The only standard comments in Scheme are line comments. There are SRFIs for block comments and S-expression comments, but support for them varies from system to system. Expression comments are not hard for editors to deal with because it is safe not to deal with them at all; however, in Scheme S-expression comments, which are written by prefixing an expression with `#;`, the expression to be commented should be placed on the next line. This is because editors that do not deal with them at all may see the semicolon as the start of a line comment, which will throw them off. Expression comments in Common Lisp, however, are always safe.

In Common Lisp, the two read-time conditionals that are guaranteed to ignore any form following them are `#+(OR)` and `#-(AND)`. `#+NIL` is sometimes used in their stead, but, while it may appear to be an obviously false conditional, it actually is not. The feature expressions are read in the `KEYWORD` package, so `NIL` is read not as `CL:NIL`, i.e. the boolean false value, but as `:NIL`, a keyword symbol whose name happens to be `NIL`. Not only is it not read as the boolean false value, but it has historically been used to indicate a feature that might be enabled – in JonL White's New Implementation of Lisp! However, the New Implementation of Lisp is rather old these days, and unlikely to matter much...until Alastair Bridgewater writes Nyef's Implementation of Lisp.

Documentation

On-line references and documentation/manuals are both useful for independent purposes, but there is a very fine distinction between them. Do not generate documentation or manuals automatically from the text of on-line references.

Rationale: *On-line references* are quick blurbs associated with objects in a running Lisp image, such as documentation strings in Common Lisp or Emacs Lisp. These assume that the reader is familiar with the gist of the surrounding context, but unclear on details; on-line references specify the details of individual objects.

Documentation and *manuals* are fuller, organized, and cohesive documents that explain the surrounding context to readers who are unfamiliar with it. A reader should be able to pick a manual up and begin reading it at some definite point, perusing it linearly to acquire an understanding of the subject. Although manuals may be dominated by reference sections, they should still have sections that are linearly readable to acquaint the reader with context.

Round and Square Brackets

Some implementations of Scheme provide a non-standard extension of the lexical syntax whereby balanced pairs of square brackets are semantically indistinguishable from balanced pairs of round brackets. Do not use this extension.

Rationale: Because this is a non-standard extension, it creates inherently non-portable code, of a nature much worse than using a name in the program which is not defined by the R5RS. The reason that we have distinct typographical symbols in the first place is to express different meaning. The only distinction between round brackets and square brackets is in convention, but the precise nature of the convention is not specified by proponents of square brackets, who suggest that they be used for `clauses`, or for forms that are parts of enclosing forms. This would lead to such constructions as

```
(let [(x 5) (y 3)] ...)
```

or

```
(let ([x 5] [y 3]) ...)
```

or

```
(let [[x 5] [y 3]] ...),
```

the first two of which the author of this guide has seen both of, and the last of which does nothing to help to distinguish the parentheses anyway.

The reader of the code should not be forced to stumble over a semantic identity because it is expressed by a syntactic distinction. The reader's focus should not be directed toward the lexical tokens; it should be directed toward the structure, but using square brackets draws the reader's attention unnecessarily to the lexical tokens.

Attribution

The following fragment of HTML/RDF/XML was generated by the Creative Commons license chooser https://creativecommons.org/choose on 2011-05-07. See the question `How do I properly attribute a Creative Commons licensed work?` at http://creativecommons.org/FAQ for more information about attribution.

Created: 2025-12-23 Tue 15:47