Common Lisp Coding standards

Robert Strandh

February 2004

Contents

Chapter 1

Purpose

In any language (natural or programming), the number of idiomatic phrases is much lower than the number of grammatical phrases.

While members of many programming language communities do not appreciate the importance of idiomatic code, Common Lisp programmers certainly do. In fact, whether a Common Lisp program can be understood at all by a maintainer crucially depends on elementary rules of indentation and naming conventions being respected.

However, these rules and conventions are often transmitted by "oral tradition" in the Common Lisp community, making it hard for new programmers to access them. This document is an attempt to collect those rules and conventions in a single place, easily accessible to anyone who would like to know.

As with must rules of this type, not everyone agrees in detail with every rule. Natural languages have dialects that introduce some systematic variation in which phrases are idiomatic and which ones are not. The same situation is true in programming languages. When possible, we have made an attempt to describe several such dialects of idiomatic Common Lisp programs. If you find that we have not described some widespread practice, please let us know.

It is our hope that these standards will be adopted by many authors of free software projects in Common Lisp. It is therefore recommended that these standards be followed for contributions of code to one of these projects. It is unlikely that any project will reject code that does not conform in every detail to these standards, but if you hesitate in choosing what construction to use, choosing the one recommended here is certainly preferable.

In fact, most rules that are discussed here are widely agreed upon in the Common Lisp community. Whenever there is disagreement concerning on of the recommendations in this document, we try to mention that fact, why there is disagreement, and what some of the alternatives are.

Chapter 2

Spacing and indentation

The maintainability of a Common Lisp program crucially depends on the way it is indented, and on the way that whitespace is used. In this section, we describe the rules that are widely respected.

In general, the rules for indentation and spacing are designed to be as economical as possible with a very precious resource, namely whitespace. Whitespace does not contain any useful information, so between two equally readable versions of a program with different amount of whitespace, the one with less whitespace is always preferable.

If you are using GNU Emacs or Xemacs, you have to make sure Emacs uses the Common Lisp indentation rules as opposed Emacs Lisp indentation rules which are slightly different. To do so, you have to add the following lines to your `.emacs` file:

```
(require 'cl)
(setq lisp-indent-function 'common-lisp-indent-function)
```

2.1 Whitespace before and after parentheses

Never put parentheses by themselves on a line.

Certainly, do not attempt any parenthesis balancing scheme like C uses.

BAD:

```
(defun fac (n)
  (if (zerop n)
      1
      (* n
         (fac (1- n)
         )
      )
  )
)
```

GOOD:

```
(defun fac (n)
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

As you can see, several lines containing no information were saved.

It is sometimes tempting to put parentheses by themselves on a line at the end of a class definition, with the idea that it makes it easier to add new slots later on like this:

BAD:

```
(defclass myclass (super)
  ((slot1 ...)
   (slot2 ...)
   (slot3 ...)
  ))
```

Here, in order to add a slot, one would put the cursor at the beginning of the line with two parentheses, and type C-o C-i to add a new slot. However it is not much harder to add a slot to this version:

GOOD:

```
(defclass myclass (super)
  ((slot1 ...)
   (slot2 ...)
   (slot3 ...)))
```

One would simply put the cursor at the beginning of the line with the last slot, type C-M-f C-j and add the new slot.

Occasionally, it might be necessary to violate this rule, for instance when the last slot has a comment associated with it like this:

TOLERATED:

```
(defclass myclass (super)
  ((slot1 ...)
   (slot2 ...)
   (slot3 ...) ; a comment
  ))
```

Never put whitespace before a closing parenthesis.
Never put whitespace after an opening parenthesis

This rule is true whether the closing parenthesis is preceded by another closing parenthesis, an opening parenthesis, or some other text like a number or a symbol.

BAD:

```
(defun fac ( n )
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

Here, the parameter list is terminated by a parenthesis preceded by a space which violates the rule. This is better:

GOOD:

```
(defun fac (n)
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

In a sequence of opening parentheses, always put a space before the first one. **In a sequence of closing parentheses, always put a space after the last one**.

BAD:

```
(defun fac(n)
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

Here, the name of the function is followed directly by the parenthesis of the parameter list, which is a direct violation of this rule.

GOOD:

```
(defun fac (n)
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

2.2 The use of newlines

Sometimes, even though a form fits on a line, it is a better idea to split it over several lines. For one thing, it is usually a good idea to keep the function header (name and parameter list) on a line by itself:

BAD:

```
(defun fac (n) (if (zerop n) 1 (* n (fac (1- n)))))
```

GOOD:

```
(defun fac (n)
  (if (zerop n)
      1
      (* n (fac (1- n)))))
```

Though, when the function is very simple, the header and the body can be on the same line:

GOOD:

```
(defun first-arg (a b) a)
```

When an `if`-form is used, it is usually a good idea to split the three subexpressions (condition, then-form, else-form) on different lines:

BAD:

```
(defun fac (n)
  (if (zerop n) 1 (* n (fac (1- n)))))
```

But when the `if`-form is simple it can go on a single line:

GOOD:

```
(defun f (n)
  (if (g n) n (1+ n)))
```

Sometimes, the arguments of a function do not fit the horizontal space, so newlines need to be introduced. The general rule, then, is to split the arguments so that the first one is on the same line as the function name, and the others are each one on a different line:

GOOD:

```
(myfun (if (g n) n (1+ n))
       (compute-highest-value object additional-stuff)
       (compute-some-more-values objects different-stuff))
```

The ordinary rules are applied to each argument, so that it might be split on several lines:

GOOD:

```
(myfun (if (zerop n)
           1
           (* n (fac (1- n))))
       (compute-highest-value object additional-stuff)
       (compute-some-more-values objects different-stuff))
```

Chapter 3

Choice of identifiers

3.1 Form of identifiers

Traditionally, Common Lisp identifiers are written in all lower-case letters with hyphens separating different words, such as in `sheet-medium` or `send-to-process`. This practice allows the use of two sets of Emacs commands: commands using words and commands using expressions. The word commands (M-f, M-b, M-d, etc) can be used to for the word-constituents of an identifier and the expression commands (C-M-f, C-M-b, etc) for the entire identifier.

Please do not use so called *studly caps* (as in `SheetMedium` and `SendToProcess`. Such use looks ugly in Common Lisp, and makes the Emacs word commands useless.

Notice that the tradition is to use hyphen (‘-’) as a word separator. Do not use the underline (‘_’) character to separate words.

3.2 Some conventional use of characters in identifiers

While the Common Lisp standard imposes few restrictions on the form of identifiers, some conventions are widespread, and we highly recommend you use them.

Special variables, and in particular global variables should have asterisks (‘*’) around their names as in `*top-level-frame*`.

Constants (i.e., global variables the value of which never change) are surrounded by plus signs ('+') as in `+conversion-factor+`.

Functions that are mostly used as *predicates* usually have names that end with the letter 'p'. If the name of the function is just one word, the 'p' is just attached to the end of the name, as in `biggerp` or `sequentialp`. If the name of the function has several names (which are then separated by hyphens), '-p' is attached to the end of the name, as in `more-important-p` or `strongly-sequential-p`. Common Lisp itself mostly follows this convention (`endp`, `zerop`, `character-alpha-p`) but not always (mostly for historical reasons and in order not to break backward compatibility) (`null`, etc). We recommend that you follow this convention in all new code.

Macros that operate on *places* usually have names that end with the letter 'f'. Again, Common Lisp itself mostly respects this convention (`rotatef`, `setf`) but not always (`push`, `pop`). We recommend that you follow this convention in all new code.

Sometimes, it is necessary to have two different functions or macros that essentially do the same thing, but whose argument lists (signatures) are not compatible. In this situation, the tradition is end one of the names with an asterisk ('*'). Examples can be found in Common Lisp itself (`let`, `let*`, `do`, `do*`). We recommend you follow this practice in new code.

expand http://www.cliki.net/Naming conventions here.

3.3 Good naming practice

An identifier should be easy to understand and read. Its name should suggest what its purpose is. It should preferably not be too long, but it is better to have a long identifier than one with an incomprehensible name.

In Common Lisp programs, variables often hold containers such as lists, vectors, hash tables, etc. Please do not prefix or suffix such variables with the type of container or words such as 'set', 'collection' (as in `list-of-boats`, `window-list`, `note-set`, etc. The convention is rather to use a simple plural 's' at the end of such identifiers (as in `boats`, `windows`, `notes`). This practice makes the identifiers shorter, and also avoids committing to a particular type of container.

Using heavily abbreviated names reveals the ignorance of the developer with respect to important tools such as the text editor. Instead of having identifiers such as `cptwks`, use `compute-weeks`. If you think that requires too much typing, you have several options: either use Emacs abbrevs so that whenever you type `cptwks`, it expands to `compute-weeks` or use dynamic abbrev expansion (‘M-/’). If you are using the *Slime* package for Emacs (which we recommend), you can also use its completion facility.

Chapter 4

Commenting and documentation strings

4.1 Comments

A Common Lisp *comment* is a sequence of characters starting with a semicolon and ending with a newline.

4.2 Purpose of a comment

Comments are intended for people reading the source code in a text editor as a printed output. The purpose of a comment is to *supplement* the information in the code whenever doing so will help the reader understand the code.

There should be no duplication of information between the code and the comments. Such duplication introduces redundancies that are hard to keep synchronized when the code evolves, in particular since the contents of the comments cannot be verified by the compiler.

In particular, comments should not be used as headers of definitions of functions and classes to duplicate information such as the name and the arguments.

Comments should be viewed as a last resort. Whenever the same information can be stated as compiler-verifiable code, such a solution is always preferable.

4.3 How many semicolons to use

The following convention is used:

- a single semicolon is used for a comment on a line that also has code on it.
- two semicolons are used for a comment on a line that is preceded only by whitespace. The comment refers to the following line(s), and is indented like the line that follows it.
- three semicolons are used for a comment that starts in the first column, and that refers to an entire function, class, or other definition.
- four semicolons are used for a comment that starts in the first column, and that refers to the entire file or to a large part of the entire file.

Example of Comments:

```
;;;; This file contains a library for ...

;;; Check for existing connections and reuse them, since we want to be
;;; able to assume a single unique connection between two connected
;;; nodes.
(defun connect (stuff)
  (let ((var (thing stuff)))
    ;; try to see whether connection exists
    (when (connection var)
      (close-other-stuff var) ; clean up before leaving
      (signal ...))))
```

Notice that editors such as GNU Emacs know about these conventions, and are able to indent lines accordingly.

4.4 Documentation strings

A *documentation string* is a character string that appears in certain predetermined places in the code, such as the first expression in the body of a function, or in a `:documentation` clause in the definition of a class or a generic function.

Documentation strings differ from comments in that they are available at runtime.

Documentation strings should be kept short. They lend themselves to comments that are associated with a particular name (of a function, a class, etc). They do not lend themselves to general explanations about the workings of a library or an application.

The first sentence of a documentation string, which should also be the first line, should be a phrase that by itself gives a short description of the object being described, kind of like the headline of a newspaper.

Subsequent text in the documentation string should expand on the description of the object. This part of the documentation string (in case the object is a function) could contain preconditions, error situations that might be signaled, and possible unexpected behavior.

In a documentation string, when you need to refer to function arguments, names of classes, or other lisp objects, write these names in all uppercase, so that they are easy to find. It is important that this convention be respected, so that automatic documentation tools can be used to introduce various markups.

4.5 Choosing between a comment and a documentation string

Comments are not available at runtime, so the target audience for comments consists of programmers that are reading the source code, for instance in a text editor.

Chapter 5

Packages

Each software component (library or application) should be put in one or more packages distinct from the `common-lisp-user` package.

In this chapter, we discuss how to organize these packages in terms of different files of the software component.

5.1 Declaring packages

It is preferable to use the declarative interface to the package system (`defpackage`, `in-package`) as opposed to the "programmatic" interface in the form of `make-package`, `import`, and `export`.

Arguments in `defpackage` option lists (in particular exported symbols) should preferably be uninterned symbols (written as the name prefixed with #:). Avoid using strings so as to avoid dependencies on the case conversion of symbols. Using keywords is tolerated. In any case, never use ordinary symbols, since they will be interned in the package that is current when the `defpackage` form is seen. The result is multiple, often conflicting symbols with the same name in different packages.

It is recommended that the `defpackage` form be put in a file separate from the code of the component. [remind me what problem this solves].

5.2 Splitting a package into several files

If a package is large, it could be a good idea to divide the code for the package into several files. Each file then contains related definitions.

The first non-comment line in each such file should be a `in-package` form.

Chapter 6

Conditionals

if vs when vs unless vs cond, vs case

special case first (mental load on maintainer)

structure recursive functions as a proofs by induction

6.1 When to use `cond`

There are a few cases when `cond` is preferable to `if`.

The first such case is the one that `cond` was designed for, namely when you have a so called *skip chain*. Rather than writing:

BAD:

```
(if condition-1
    do-something-1
    (if condition-2
        do-something-2
        do-something-3))
```

it is preferable to use `cond` like this:

GOOD:

```
(cond (condition-1 do-something-1)
      (condition-2 do-something-2)
      (t do-something-3))
```

The second case is when the *then* or *else* part of a condition contains several expressions to evaluate, which would require the use of `progn`. Instead of writing like this:

BAD:

```
(if condition
    (progn do-something-1-a
           do-something-1-b)
    do-something-2)
```

or

BAD:

```
(if condition
    do-something-1
    (progn do-something-2-a
           do-something-2-b))
```

you can write:

GOOD:

```
(cond (condition do-something-1-a do-something-1-b)
      (t do-something-2))
```

or

GOOD:

```
(cond (condition do-something-1)
      (t do-something-2-a do-something-2-b))
```

Using `cond` in this situation saves the `progn` since `cond` has an *implicit* `progn` in each clause.

When the actions in a `cond` clause are too wide to fit on a line, split lines like this:

GOOD:

```
(cond (condition
        do-something-1-a
        do-something-1-b
        do-something-1-c
        do-something-1-d
        do-something-1-e
        do-something-1-f
        do-something-1-g)
       (t do-something-2-a do-something-2-b))
```

The third situation when `cond` is useful is when the result of a condition must be returned as the value of the conditional. Instead of writing:

BAD:

```
(let ((val (complicated-calculation a b c)))
  (if val
      val
      (do-something-else x y z)))
```

you can use `cond` like this:

GOOD:

```
(cond ((complicated-calculation a b c))
      (t (do-something-else x y z)))
```

6.2 When to use `case`

The `case` conditional can be seen as a special case of the `cond` conditional that solves the problem of testing the value of an expression against a number of *constant values*.

The reason for the existence of `case` is that it is much faster than `cond` for this special case. In general, the compiler can generate code to search a hash table as opposed to testing each clause sequentially.

For that reason, we recommend you always use `case` in place of `cond` for this special case. In general, `case` is not applicable to any other cases.

6.3 When to use `when` and `unless`

In situations when an `if` with only one branch (either the *then*-branch or the *else*-branch) is called for, please use `when` and `unless` instead.

There are two advantages to using `when` and `unless` instead of `if` in these situation:

- they are much more economical in terms of whitespace,
- they are more precise [insert reference to chapter that talks about preciseness once it is written].

Take for instance the following situation:

```
(if (some-test a b c)
    (progn (some-calculation-1 a b c)
           (some-calculation-2 a b c)
           (some-calculation-3 a b c)
           ...))
```

Respecting the indentation rules of `if` and `progn`, the `(some-calculation-x)` expressions need to be indented 11 positions compared to the `if`-expression itself. Using `when` instead, we get:

```
(when (some-test a b c)
  (some-calculation-1 a b c)
  (some-calculation-2 a b c)
  (some-calculation-3 a b c)
   ...)
```

with an indentation of only 2 positions.

When using `when` and `unless`, make sure the test is not negated with a `not` as in:

BAD:

```
(when (not (some-test a b c))
  (some-calculation-1 a b c)
  (some-calculation-2 a b c)
  (some-calculation-3 a b c)
   ...)
```

Instead use the opposite conditional:

GOOD:

```
(unless (some-test a b c)
  (some-calculation-1 a b c)
  (some-calculation-2 a b c)
  (some-calculation-3 a b c)
   ...)
```

However, do not eliminate negations using `null`. Even though technically `null` has the same definition as `not`, they are not morally equivalent; `not` is used to negate a boolean expression whereas `null` is used to test whether a list is empty:

NOT GREAT:

```
(when (cdr stuff)
  (some-calculation-1 a b c)
  (some-calculation-2 a b c)
  (some-calculation-3 a b c)
   ...)
```

GOOD:

```
(unless (null (cdr stuff))
  (some-calculation-1 a b c)
  (some-calculation-2 a b c)
  (some-calculation-3 a b c)
   ...)
```

Chapter 7

Choice of Common Lisp idioms

In this chapter, we discuss the choice of various Common Lisp idioms and other constructs.

7.1 *let* vs *let**

It is preferable to use *let* whenever there is no sequential dependencies between initializations of local variables. It is preferable to use *let** to using nested *let*, even though every initialization does not necessarily depend on the previous one. For instance :

NOT GREAT:

```
(let ((x (f ...))
      (y (g ...)))
  (let ((z (h x)))
    ...))
```

BETTER:

```
(let* ((x (f ...))
       (y (g ...))
       (z (h x)))
  ...)
```

7.2 *not* vs *null*

While *not* and *null* are operationally equivalent, they are not *morally* equivalent.

Using *not* signals to the reader that the argument is intended to be of type *boolean*.

The function *null*, on the other hand, signals to the reader that the argument is a *list*.

These conventions have some consequences for testing for empty lists in conditionals such as *when* and *unless*. For instance:

BAD:

```
(when l
  ...)

(unless l
  ...)
```

GOOD:

```
(unless (null l)
  ...)

(when (null l)
  ...)
```

7.3 *incf, decf, 1+, 1-*

While the functions *incf*, *decf*, *1+*, *1-* work for all types of numbers, their intended use is for integers only. For rationals, floating point numbers and complex numbers, we recommend the use of combinations of *setf* and the arithmetic operations *+* and *-*.

7.4 *car* and *cdr* vs *first* and *rest*

Traditional wisdom dictates that *fist* and *rest* be used when the argument is morally a *list*, whereas *car* and *cdr* are supposed to be used when the argument is a more general *tree* made up of *cons* cells.

While we recommend following this traditional wisdom, we tolerate the use of *car* and *cdr* for lists, simply because such use is historically common and much existing code does not follow the traditional wisdom.

7.5 Using combinations of *car* and *cdr*

When a program contains extensive use of long combinations of *car* and *cdr* such as *cadddr*, one should suspect a problem of data representation.

Usually, the existence of such functions indicates that a data structure that should have been represented as a class, is instead represented as a list.

At the very least, we recommend that specific accessors be defined, such as:

```
(defun first-name (person) (car person))
(defun last-name (person) (cadr person))
(defun address (person) (caddr person))
```

Such accessors allow for subsequent modifications of the data representation without having to alter client code.

However, these days there are great advantages to using classes instead, in particular because they allow better type checking and generic function dispatch.

If you do have to work with lists to represent data types, we recommend you read up on *destructuring-bind*. In particular as an alternative to writing something like:

```
(let ((first-name (car person))
      (last-name (cadr person))
      (address (caddr person)))
  ...)
```

we recommend you use:

```
(destructuring-bind (first-name last-name address) person
  ...)
```

7.6 Accumulating elements into a list

The traditional way of accumulating elements into a list is to use a combination between *push* and *nreverse* like this:

```
(let ((acc '()))
  (do (...)
      (... (nreverse acc))
    (when ...
      (push elem acc))))
```

Avoid in any case the use of *append* to add new elements to the end of the list, and this for reasons of computational complexity. Using *nreverse* is preferable to *reverse*, so as to avoid useless *consing*.

However, for this particular situation, we rather recommend using *loop*, as in:

```
(loop ...
      when ... collect elem)
```

which is usually much shorter and more clear.

7.7 *reduce* vs *apply*

The Common Lisp standard allows for implementations to impose a limitation on the number of arguments possible to a function call. While these limitations in most implementations are quite high, there might still be a problem when *apply* is used.

In the case of left-associative functions such as *+*, we recommend simply replacing *apply* with *reduce*. For right-associative functions, use the keyword argument *:from-end t* with *reduce*.

The use of *apply* remains acceptable when the length of the list is bounded and known to be smaller than the limitation on the number of arguments in most implementations.

7.8 Using *setf* with more than two arguments

The use of *setf* with more than two arguments can be very useful in certain situations. We recommend it over a sequence of uses of *setf* with two arguments. For instance:

NOT GREAT:

```
(let ((...))
  (setf x y)
  (setf a b))
```

PREFERABLE:

```
(let ((...))
  (setf x y
        a b))
```

However, it is important that exactly two arguments be given on each line. For instance:

BAD:

```
(let ((...))
  (setf x y a b))
```

GOOD:

```
(let ((...))
  (setf x y
        a b))
```

When a pair of arguments to *setf* does not fit on a line, it is preferable *not* to use more than two arguments. In this case, the arguments are aligned vertically. For instance:

BAD:

```
(let ((...))
  (setf (a-very-long "argument to setf requiring lots of room" 1 2 3)
        (another-very-long "argument to setf requiring lots of room")
        (yet-another-very-long "argument to setf requiring lots of roo
        (and-another "argument to setf requiring lots of room" 5 4)))
```

GOOD:

```
(let ((...))
  (setf (a-very-long "argument to setf requiring lots of room" 1 2 3)
        (another-very-long "argument to setf requiring lots of room"))
  (setf (yet-another-very-long "argument to setf requiring lots of roo
        (and-another "argument to setf requiring lots of room" 5 4)))
```

7.9 The use of *tagbody*

The main use of *tagbody* is in code generated by macros. There are few justified usages of *tagbody* in other code. An exception is for coding automata.

Chapter 8

Using conditions

handler-case vs handler-bind

error vs signal vs assert

conditions vs catch/throw. restarts

Chapter 9

Using Common Lisp pathnames

Someone else had better write this chapter. –RS

pathnames are structured objects that represent paths in the filesystem. They are usually pretty good at doing this: your functions that accept filenames should accept pathname designators.

To control where files are found/created, use *default-pathname-defaults* instead of unportable directory-setting functions. Generally you should bind it with LET instead of setting with SETF so that you don't break any file access that may be happening in other threads.

Logical pathnames are not as generally useful as you might think they are. These are the rules which govern when you can use logical pathnames without getting very surprised thirty minutes later:

* When all of the files will be created by the same Lisp implementation and only ever accessed using that Lisp implementation

* and you can name them all using only uppercase letters, digits and the hyphen (-)

* and you don't care too much about how they're represented in the underlying file system

That's about it: pretend that you've got a filesystem image loopback mounted at that point that only Lisp can look inside, and your expectations will be approximately correct.

http://ww.telent.net/diary/2002/8/#26.82823

Chapter 10

Multiprocessing/Threading

The Common Lisp standard does not define any primitives for multiprocessing or threading. Most modern implementations support threads in some form, however.

It is hard to write large end-user applications without using threads. But using threads effectively is not trivial either. In this chapter, we give some guidelines concerning how to write thread-safe code. These guidelines do not only concern authors of applications that do use threading, but also authors of libraries that might be used by such applications.

10.1 The Common Lisp treading model

As we mentioned, the Common Lisp standard does not mention threads. In this section, we give an overview of the Common Lisp threading model as implemented by most systems.

Most Common Lisp implementations that have threads are not themselves thread safe. This means that you cannot count on operations such as modifying a property list or a hash table to be atomic, nor even *safe*. It is the responsibility of the application or library author to make sure that either simultaneous access by different threads cannot happen, or that such accesses are protected by primitives for mutual exclusion.

In most Common Lisp implementations that have threads, the values of *special variables* are shared between the creating and the created thread. Whenever such a variable is *assigned to* without being bound, the other thread sees the new value. Whenever such a variable is *bound*, its storage becomes private to the thread.

10.2 Avoid global state

When writing applications or libraries, you may want to think of the possibility of several simultaneous threads executing your code. For that reason, we recommend that you avoid *global state* (unless the purpose is specifically to share information between threads), in particular:

- do not use property lists of symbols that are shared between threads (which they usually are);
- make sure special variables are *bound* before being assigned to;

10.3 Synchronization

Before SMP (symmetric multi processor) computers were invented, Common Lisp implementations supporting threads counted on the fact that only one thread could run at a given time, and that in order for a different thread to run, the *scheduler* had to be invoked. It was thus safe to count on operations being atomic as long as one could make sure that the scheduler did not run. This situation gave rise to a style of programming using the primitive *without-scheduling* around a block of code that had to be executed atomically.

Some modern Common Lisp implementations run on computers where the operating system (as opposed to the Common Lisp system) take care of the scheduling. For these implementations, there may not be a primitive such as *without-scheduling*, or it may not work as intended.

In addition, if the computer has more than one CPU, it is possible that two threads run *truly simultaneously* so that it is impossible to guarantee atomicity by preventing the scheduler from running.

We recommend that you never use primitives such as *without-scheduling*, and instead use primitives for mutual exclusion and condition variables.

Chapter 11

System construction

11.1 The `asdf` system construction tool

Chapter 12

Object-oriented design

Common Lisp has the most powerful object system around. It is in many ways completely different from that of other object-oriented languages such as Java and C++.

In most other object-oriented languages, the *class* is at the center of the design. This is not the case in Common Lisp. In this chapter, we review some object terminology and give guidelines for object-oriented design.

12.1 Abstract data types

Good object-oriented design revolves around *abstract data types*. An abstract data type is a data type defined by *what you can do with it* as opposed to how it is represented in the computer. Here, "what you can do with it" is going to be one or many collections of *operations*.

An *operation* is ...

A protocol is a collection of operations that share at least one abstract data type.

An abstract data type can participate in several different protocols.

The abstract data type is usually represented as a hierarchy of classes, but not necessarily.

It is usually a good idea to create *internal* protocols as well, in order to facilitate extensions to the software.

structure vs class (we recommend using a class except when structure is necessary for performance reasons)

slot-value and with-slots vs accessors. Reserve slot-value and with-slots for code internal to the module (cf arrow in C)

Chapter 13

Misc stuff to be inserted in other chapters

keyword arguments (avoid them for fast-executing functions)

* Keyword arguments: should probably be recommended when there are a) more than three arguments b) when there is no "natural" way to order arguments c) for multiple optional arguments. Use of optional and keyword arguments togather should be discouraged. Possible performace penalty of keyword arguments should not be emphasized, except in a separate section on optimization.

tagbody (use only in macro bodies)

Index