General Guidelines

Use Libraries

Look for libraries that solve the problems you are trying to solve before embarking on a project. Making a project with no dependencies is not some sort of virtue. It doesn't aid portability and it doesn't help when it comes to turning a Lisp program into an executable.

Writing a library that solves the same problem as another hurts consolidation. Generally, you should only do this if you are going to make it worth it: Good, complete documentation, examples, and a well-designed website are – taken together – a good reason to write an alternative library to an existing one.

As always, check the licensing information of the libraries you use for incompatibilities.

Write Libraries

Before starting a project, think about its structure: Does *every* component have to be implemented within the project? Or are there parts that might be usable by others? If so, split the project into libraries.

If you set out to write a vast monolith and then lose interest, you will end up with 30% of an ambitious project, completely unusable to others because it's bound to the rest of the unfinished code.

If you think of your project as a collection of independent libraries, bound together by a thin layer of domain-specific functionality, then if you lose interest in a project you will have left a short trail of useful libraries for others to use and build upon.

In short: write many small libraries.

Naming

General style

Names are lowercase, separated by single dashes (`-`), and they are complete words. That is, you should write `user-count` rather than `user-cnt` and `make-text-node` is better `mk-txt-node`.

Variables

Special variables (Mutable globals) should be surrounded by asterisks. These are called earmuffs.

For example:

```
(defparameter *positions* (make-array ...))

(defparameter *db* (make-hash-table))
```

Constants should be surrounded with plus signs. For example:

```
(defconstant +golden-ratio+ 1.6180339)

(defconstant +allowed-operators+ '(+ - * / expt))
```

Predicates

A predicate is a function that, given some input, returns `t` or `nil`.

Predicates should be suffixed with:

`p`
If the rest of the function name is a single word, e.g: `abstractp`, `bluep`, `evenp`.

`-p`
If the rest of the function name is more than one word, e.g `largest-planet-p`, `request-throttled-p`.

Don’t prefix package names

This is what packages are for. The name of a function in a package `myapp.parser` should not start with `parser-`.

If you’re binding a C library where every function name is of the form `library_name_function_name`, use the library’s name (Or, more accurately, a reasonable Common Lisp translation of the name, i.e. `libGUI` should be `lib-gui`) as the package name and the actual name of the function as the function name.

Formatting

Indentation

Indentation is two lines per form, for instance:

```
(defun f ()
  (let ((x 1)
        (y 2))
    (format t "X=~A, Y=~A" x y)
    (terpri)
    t))
```

There are, however, some exceptions. In the `if` special form, both branches must be on the same line:

```
(if (> x 5)
    (format t "Greater than five")
    (format t "Less than or equal to five"))
```

Line length

Lines should not exceed 100 columns. The lower limit of 80 columns, used in other languages, is less fitting for Common Lisp, which encourages descriptive variable names and complete rather than abbreviated function names.

File header

The top of the file should include a four-semicolon comment comment describing the purpose of the file.

You should **not** include copyright or authorship information in file-level comments. The license should not be mentioned outside the system definition and the README.

Documentation

Docstrings everywhere

Common Lisp allows you to add docstrings to functions, packages, classes and individual slots, and you should use this.

Comment Hierarchy

Comments that start with four semicolons, `;;;;`, should appear at the top of a file, explaining its purpose.

Comments starting with three semicolons, `;;;`, should be used to separate regions of the code.

Comments with two semicolons, `;;`, should describe regions of code within a function or some other top-level form, while single-semicolon comments, `;`, should just be short notes on a single line.

Do not use comments to erase code

When testing, it’s fine to use comments, especially multi-line comments, so experimentally remove a large segment of code. However, all projects should use version control, and accordingly, code should be liberally deleted rather than hidden inside a comment.

CLOS

The ideal class definition looks like this:

```
(defclass request ()
  ((url :reader request-url
        :initarg :url
        :type string
        :documentation "Request URL.")
   (method :reader request-method
           :initarg :method
           :initform :get
           :type keyword
           :documentation "Request method, e.g :get, :post.")
   (parameters :reader request-parameters
               :initarg :parameters
               :initform nil
               :type association-list
               :documentation "The request parameters, as an association list."))
  (:documentation "A general HTTP request."))
```

Slot options

The following slot options should be used in this order:

`:accessor`, `:reader` or `:writer`
The name of the accessor method.

`:initarg`
The keyword argument used to initialize the value.

`:initform` (If any)
The initial value of the slot, if it's not explicitly given.

`:type` (If possible)
The type of the slot.

`:documentation`
The slot's documentation string.

MOP-defined slot options should be added after all other slot options and before the `:documentation` option.

Use the :TYPE slot option

Types are documentation, and Common Lisp allows you to declare the type of class slots.

```
(defclass person ()
  ((name :accessor person-name
         :initarg :name
         :type string
         :documentation "The person's name.")
   (age :accessor person-age
        :initarg :age
        :initform 0
        :type integer
        :documentation "The person's age."))
  (:documentation "A person."))
```

Here, the trivial-types library will come in handy.

Checking that the type complies to the initial value is undefined by the standard and left to the implementations. Clozure CL and SBCL 1.5.9 and onwards (or all versions when safety is high) do type checks at compile time.

Flow Control

In short:

- Use `if` when you have a true branch and a false branch.
- Use `when` or `unless` when you're only interested in one condition branch.
- Use `cond` when you have several conditional branches.

Use WHEN, UNLESS

If you have an `if` expression with no else part, you should use `when` instead, and when you have an expression like `(if (not <condition>) ...)` with no else part, you should use `unless`.

For example:

```
(if (engine-running-p car)
    (drive car))

(if (not (seatbelts-fastened-p car))
    (warn-passengers car))
```

Should be:

```
(when (engine-running-p car)
  (drive car))

(unless (seatbelts-fastened-p car)
  (warn-passengers car))
```

Note the difference in indentation.

Keep conditions short

Large conditional expressions are harder to read, so should be factored out into functions.

For example, this:

```
(if (and (fuelledp rocket)
         (every #'strapped-in-p
                (crew rocket))
         (sensors-working-p rocket))
    (launch rocket)
    (error "Aborting launch."))
```

Should be written as:

```
(defun rocket-ready-p (rocket)
  (and (fuelledp rocket)
       (every #'strapped-in-p
              (crew rocket))
       (sensors-working-p rocket)))

(if (rocket-ready-p rocket)
    (launch rocket)
    (error "Aborting launch."))
```

Packages

One Package Per File

Unless it makes sense to have one package cover multiple files.

Avoid :USE

Unless you are really going to need all (or most of) the symbols in a package, it is strongly recommended that you write a manual `:import-from` list as opposed to using `:use`.

For instance, if you're writing a package that uses a couple of symbols from Alexandria, don't do this:

```
(defpackage my-package
  (:use :cl :alexandria))
(in-package :my-package)
```

Instead, do this:

```
(defpackage my-package
  (:use :cl)
  (:import-from :alexandria
                :with-gensyms
                :curry))
```

Hierarchical Package Names

For instance, here is an example package hierarchy for a game:

```
title
  title.graphics
    title.graphics.mesh
    title.graphics.obj
    title.graphics.gl
  title.config
  title.logging
  title.db
```

Project Structure

Directory Structure

The average, small project will look like this:

```
cl-sqlite3/
  src/
    cl-sqlite3.lisp
  t/
    cl-sqlite3.lisp
  .gitignore
  README.md
  cl-sqlite3.asd
  cl-sqlite3-test.asd
```

As an example of a larger project using continuous integration, multiple packages, and optional contrib systems, here's what a hypothetical web scraping framework would look like:

```
spider/
  src/
    http/              # The code here would be under the package `spider.http`
      request.lisp
      response.lisp
    downloader/        # The code here would be under the package
`spider.downloader`
      downloader.lisp
      middleware.lisp
    condition.lisp     # Web scraping-related conditions
    util.lisp          # Utility functions
    settings.lisp      # Settings for the web scraper
  t/
    http.lisp          # Request/response test suite
    downloader.lisp    # Downloader test suite
    final.lisp         # Code to run the test suites, and set up/tear down any
test fixtures
  contrib/
    run-js/            # A contrib module to run JavaScript in the scraper
      README.md        # A description of the module
      run-js.lisp      # The source code
  .gitignore
  .travis.yml          # The .travis.yml file to enable continuous integration
  README.md
  spider.asd
  spider-test.asd
  spider-run-js.asd
```

System Definitions

The following system definition files should be defined:

`project.asd`

The system definition of the library or application itself. Contains most of the metadata.

`project-test.asd`
Unit tests for the project.

Contributed modules should each have their own `.asd` file.

Options

The following system definition options should be specified:

`:author`, `:maintainer`
The original author and current maintainer of the project. If you're writing the system definition for the first time, the two should be equal.

`:license`
The license of the project. Should be a short string with the name of the license.

`:homepage`
A link to the project homepage. It can be an actual homepage, or a link to the GitHub or Bitbucket repo.

`:version`
The project's version string.

The following subsections have example contents of `.asd` files. Note that these are for the simplest case of a small library with a single file. For larger projects, you should split the code across multiple files.

Main System Definition

The main system definition is the first 'entry point' to your project, and as such, should contain all the relevant metadata. It should look like this:

```
(defsystem my-project
  :author "John Q. Lisper <jql@example.com>"
  :maintainer "John Q. Lisper <jql@example.com>"
  :license "MIT"
  :homepage "https://github.com/johnqlisp/my-project"
  :version "0.1"
  :depends-on (:local-time
               :clack)
  :components ((:module "src"
                :serial t
                :components
                ((:file "my-project"))))
  :description "A description of the project."
  :long-description
  #.(uiop:read-file-string
     (uiop:subpathname *load-pathname* "README.md"))
  :in-order-to ((test-op (test-op my-project-test))))
```

Testing System Definition

The system definition file for the test system doesn't need as much metadata as the main system definition file. It should look like this:

```
(defsystem my-project-test
  :author "John Q. Lisper <jql@example.com>"
  :license "MIT"
  :depends-on (:my-project
               :some-test-framework)
  :components ((:module "t"
                :serial t
                :components
                ((:file "my-project")))))
```

Other code

From the ASDF manual:

> However, it is recommended to keep such forms to a minimal, and to instead define defsystem extensions that you use with `:defsystem-depends-on`.

System definition files should contain a system definition and not much more. They should not contain any code, or anything that fiddles with ASDF's setup. If you need to do that, do it in your initialization file.

A read-time call to read the contents of the README file into the system definition's `:long-description` option is fine.

The README

You should use Markdown for the README file, for two reasons:

- Most source code hosting services detect Markdown README files and display them appropriately, which makes them easier to read than a plain text file.
- Quickdocs also extracts Markdown README files, making project descriptions easier to read.

The average `README.md` file should look like this:

```
# [Project title]

[A short, one-line description of the project]

# Overview

[A longer description of the project, optionally with sub-sections like
'Features', 'History', 'Motivation', etc.]

# Usage

[Examples of usage]

# License

Copyright (c) [Year] [Authors]

Licensed under the [Your license here] License.
```

If the project is small enough, you should include a ‘Documentation’ section after ‘Usage’ describing its API. Larger projects should have separate documentation, however.

Use Continuous Integration services

Continuous integration allows you to move testing of your project to an external service, and allows potential users to see that its tests are passing without having to download and test it themselves

CI services are especially useful when the project depends on external dependencies, such as databases, to be installed for testing. An ORM, or database interface library, or a binding to a library written in C, would require specific configuration and setup of external things in order to be tested. CI services allow you to install all the necessary packages and configure the virtual machine/container so testing doesn’t require effort on the part of your users.