Elisp Cheat Sheet

Links

- GNU Emacs Lisp Reference Manual:
  - GNU Emacs Lisp Coding Conventions
  - Customization Settings
  - Standard Symbol Properties
  - Window Frame Parameters
  - Window Parameters
- Regular Expressions:
  - Free Software Foundation: Emacs Lisp - Syntax of Regular Expressions
  - Free Software Foundation: Emacs Lisp - Table of Syntax Classes
  - EmacsWiki: Regular Expression
  - Jan Goyvaerts: POSIX Bracket Expressions
  - IEEE and The Open Group: The Open Group Base Specifications Issue 7 - Regular Expressions

Assignment

- define a symbol as variable and assign an (possibly calculated) initial value: `(defvar sym &optional init “doc”)`
- do the same as `defvar`, but define *sym* as a constant variable:[1] `(defconst sym init “doc”)`
- declare a symbol as a customizable variable, with a default value *init*: `(defcustom sym init “doc” &rest args)`
- assign a value to a symbol, also evaluating the symbol (not only the value): `(set 'sym val)`
- assign values to variables (dependencies allowed, does not evaluate *var*): `(setq var val …)`
- assign default values to variables (dependencies allowed): `(setq-default var val …)`
- call a function with *args*, one after another: `(apply fun &rest args)`

(Not) Eval

- temp. bind some variables (overwriting outer variables of same name), then evaluate body:[2] `let`
- do the same as `letf`, but dependencies between assigned (init) values in variable bindings are allowed: `let*`
- evaluate body (multiple statements) as a single statement: `(progn …)`
- return a (self-quoting, not evaluated) lambda expression: `(lambda …)`

Control

- *if*..*else* control structure: `(if condition …)`
- and without an *else* path: `(when condition …)`

- an equivalent to `(when (not condition) …)` is: `(unless condition …)`
- try each clause until success: `(cond …)`

Simple Lists

- make a list:
  - `'(a b c d)`
  - `(quote a b c d)`
  - `(list a b c d)`
  - `(append a b '(c d))`
- add element *elt* to list *lst*: `(add-to-list 'lst elt)`
- concatenate any number of lists: `(nconc &rest lists)`
- is element *elt* a member of list *lst*: `(member elt lst)`
- or, to get a true boolean value: `(not (null (member elt lst)))`
- get the first element from a list: `(car lst)`
- get a list starting from the second element of *lst*: `(cdr lst)`
- get the second element (see prev. two lines): `(car (cdr lst))`, or `(cadr lst)`
- get the n-th element: `(nth n lst)`
- set the first element of a list to a new value: `(setcar lst val)`
- number of elements in a list: `(length lst)`
- iterate over all list elements: `(dolist …)`
- stringify and concatenate all list elements: `(mapconcat …)`
- apply a function to all list elements: `(mapc fun seq)`
- apply a function to all elements of a sequence[3] and make a list from the results: `(mapcar fun seq)`
- find a key in *car* of list elements: `(assoc key lst)`

Association Lists (Alist)

Alists can be seen as simple lists with special elements (so called *cons* cells), representing an association between *key* and *val*.

- make a *cons* cell for an association between *key* and *val*: `(cons key val)`
- or, the same as above: `'(key . val)`
- get the association for *key* in *alst*: `(assq key alst)`
- get the (reverse) association for *val* in *alst*: `(rassoc val alst)`
- make a copy of an *alist* (using `cl-lib`): `(copy-alist alst)`

Strings

- test if object is a string: `(stringp obj)`
- compare two strings: `(string= str1 str2)`
- test if string matches a regular expression: `(string-match-p regexp str)`

Symbols

- test if symbols value is not void (symbol exist): `(boundp sym)`
- test if symbols function definition is not void (function exist): `(fboundp sym)`
- test if object is a function: `(functionp obj)`
- quote a function name (for byte-compile):[4] `(function fun)`, or `#'fun`
- call *fun* (this might be a variable, else quote it) with *args*: `(funcall fun args..)`
- get the symbol name as string `(symbol-name sym)`
- make/reference a symbol from a name (string): `(intern name)`
- get function definition: `(symbol-function sym)`
- set function definition: `(fset sym def)`
- get symbol property:[5] `(get sym prop)`
- set symbol property: `(put sym prop val)`
- remove property from symbol (using `cl-lib`): `(remprop sym prop)`

Shell

- execute shell command: `(shell-command …)`
- execute within modified environment

```
(let ((process-environment ...))
  (shell-command ...))
```

- execute with sudo

```
(let ((default-directory "/sudo::"))
  (shell-command "ls"))
```

Interactive

- call interactive command with prefix argument programmatically

```
(let ((current-prefix-arg '(4))) ; with C-u take symbol under point
  (call-interactively #'grep)))  ; autoloads `grep'
```

- pass-through `current-prefix-arg` to interactively called command[6]

```
(defun my-interactive-command ()
   (interactive)
   (call-interactively #'grep))
```

- copy/re-use the interactive specification of another command

```
(eval
   `(defun rho/ispell-change-dictionary (dict &optional arg)
           "Do the same as original (advised) function `ispell-change-
dictionary',
```

```
    but (in addition) update the Flyspell mode lighter with the current
    dictionary."
               ,(interactive-form 'ispell-change-dictionary) ; (interactive
    ...)
               :
               :
```

Major Modes

- a `sample` mode, derived from `c-mode`:

```
;; sample-mode.el -*- coding: utf-8 -*-
;;
;; Sample Mode
;; Copyright (C) 2017 Ralf Hoppe <ralf.hoppe@dfcgen.de>
;;

(defvar sample-mode-hook nil)

(defconst sample-mode-syntax-table
  (let ((table (make-syntax-table)))
    (modify-syntax-entry ?\" "\"" table) ; string delimiter

    (modify-syntax-entry ?/ "<124" table)
    (modify-syntax-entry ?* "<23b" table)
    (modify-syntax-entry ?\n ">b" table) ; \n is comment end for (only) C++
style
    (modify-syntax-entry ?# "<" table)
    table))

(defconst sample-font-lock-keywords
  '(("\\(builtin1\\|builtin2\\)\\>" . font-lock-builtin-face)
    ("\\<\\(keyword1\\|keyword2\\)\\>" . font-lock-keyword-face)
    ("\\<\\(type1\\|type2\\)\\>" . font-lock-type-face)
    ("\\<Sample[0-9a-zA-Z_]*\\>" . font-lock-function-name-face)
    ("\\<\\(SAMPLE\\|Sample_\\)[0-9a-zA-Z_]*\\>" . font-lock-constant-face)
    ))

(define-derived-mode sample-mode c-mode "Sample" :syntax-table sample-mode-
syntax-table
  (set (make-local-variable 'font-lock-defaults) '(sample-font-lock-
keywords))
  (set (make-local-variable 'indent-line-function) 'c-indent-line)
  (setq-local comment-start "// ")
  (setq-local comment-end "")
  (font-lock-fontify-buffer)
  (run-hooks 'sample-mode-hook))

(add-to-list 'auto-mode-alist '("\\.sample\\'" . sample-mode))
```

```
(when (featurep 'speedbar)
  (speedbar-add-supported-extension ".sample"))

(provide 'sample-mode)
```

1)
Citation from Emacs help: *This declares that neither programs nor users should ever change the value. This constancy is not actually enforced by Emacs Lisp, but the symbol is marked as a special variable so that it is never lexically bound.*
2)
For details see the article by Artur Malabarba: Understanding letf and how it replaces flet.
3)
A sequence may be a list, a vector, a bool-vector, or a string.
4)
Strictly avoid using `quote` for that. For details see the article by Artur Malabarba: Get in the habit of using sharp quote.
5)
For standard symbol properties (plist) see Emacs Lisp Manual.
6)
There is nothing special todo in this case, because `current-prefix-arg` is retained during interactive command processing.