NOTES ON EMACS LISP

Frédéric GALLIANO

Université Paris-Saclay, Université Paris Cité, CEA, CNRS, AIM, 91191, Gif-sur-Yvette, France

January 5, 2024

Contents

1 GENERAL

1.1 Basic Functions

1.1.1 Evaluating an expression

C-x C-e → evaluate a fonction in an Emacs window, if the cursor is right after the closing parenthesis ⇔ `M-x eval-last-sexp`.
M-x eval-region → evaluate the code contained in a selected region.
M-x eval-buffer → evaluate the code of the whole buffer.
M-x load-file <file> → evaluate the code contained in the file `<file>`.
M-x eval-expression <expr> → evaluate the expression `<expr>`.
M-x ielm → launch the Emacs Shell, which is the interactive elisp command evaluator.

1.1.2 Help & Documentation

C-h f <funcname> → describe the function ⇔ `M-x describe-function`. The generic character, `*`, can be used.
C-h v <varname> → describe the variable ⇔ `M-x describe-variable`.
M-x emacs-index-search → search the emacs manual.
M-x elisp-index-search → search the elisp manual.
q → get out of the debugger..
C-h e → print the "`*Messages*`" buffer.

1.1.3 Displaying messages

Displaying messages to the user is done with the command `message`:
`(message "Hello")` → every character string.
`(message "the file is %s" (buffer-name))` → `%s` denotes a character string.
`(message "the list is %S" (list 8 2 3))` → `%S` denotes any elisp expression.
`(message "fill-column = %d" fill-column)` → `%d` denotes a number.

1.2 Operators

Operations have the form: `(<operator> <variable1> ... <variableN>)`.

1.2.1 Arithmetic

Elisp considers `2` and `2.` as integers and `2.0` as a real.
+ → addition.
- → subtraction.
***** → multiplication.
/ → division:
- integer (modulo), if the numbers are integers;
- else real division.

% → remainder of the integer division.
expt → exponent.
float → convert an integer to a real.
truncate → truncate a real..
floor → round to the lower integer.
ceiling → round to the upper integer.
round → round to the closest integer.
string-to-number → convert a character to a number.
number-to-string → convert a character to a number.

1.2.2 Booleans

Functions finishing by a `p` (for predicate) often have a boolean result: *e.g* `integerp`, `floatp`.
True → `t`.
False → `nil` ou `()` (liste vide).
Basic operators for numbers → <, >, <=, >=, =, /=, `or`, `and`.
`equal` → compare two variables (values and types).
`string-equal` → compare two strings.

1.3 Variables

- Variables do not need to be declared.
- Variables defined with `set` or `setq` are global.
- Variables defined with `let` are local.

1.3.1 Assigning variables

Variable assignment → `(set 'famous-guitars '(Gibson Fender PRS))` then `(message "Famous guitars are %s" famous-guitars)`.
Alternate assignment → `(setq famous-guitars '(Gibson Fender PRS))`.
Example (incrementation) → `(setq counter 0)` then `(setq counter (+ counter 1))`.
Temporary/local assignment → `(let <varlist> <body>)`.
Case with initializing → `(let (<var1> <val1>) (<var2> <val2>) etc) <body>)` if `<val?>` are omitted, `<var?>` are initialized to `nil`.
Initializing → `(defvar <var> <val>)` assign `<val>` to `<var>` only if `<var>` is `nil` (has not been assigned yet).

1.3.2 Character strings

(length "<string>") → number of characters in `<string>`.
(substring "<string>" <n> <m>) → extract the sub-string in `<string>` between the `<n>` and `<m>` characters.
(replace-regexp-in-string "<regexp>" "<replacement>" "<string>") → replace the characters in `<string>` satisfying the regular expression `<regexp>` by `<replacement>`.
(concat "<str1>" "<str2>") → concatenate the strings `<str1>` and `<str2>`.
(string-math "<regexp>" "<str>") → verify if strings `<str>` corresponds to the regular expression `<regexp>`.
(split-string "<str>" "<separ>") → return the sub-strings in `<str>` obtained by splitting it at the separators `<separ>`.
Comparing string lengths → (string-lessp <a> <b>).

1.3.3 Lists

Lists (ab, cd, *etc.* are "atoms") → (ab cd ef2).
Creating → `(list <el1> ... <eln>)` creates the list of elements `<el1>` to `<eln>`.
List starting with a quote → '(ab cd ef2); the starting quote tells LISP to not interpret the list. If the list does not start with a quote, the first element is a command and the other a re arguments. For instance, the list `(+ 2 2)` returns `4`. It can be evaluated by placing the cursor right after the list and typing C-x C-e.
Initializing a list → `(make-list <length> <initval>)` creates a list of length `<length>` with `<initval>` everywhere.
Ramps of values → `(number-sequence <n> <m> <step>)` creates a list of numbers between `<n>` and `<m>` with a step `<step>`. These numbers can be real or integer. If `<step>` is not provided, it is 1. If, moreover, `<m>` is not provided, the list is simply `(<n>)`.
Extraction → `(elt <list> <ind>)` returns the element in `<list>` at the index `<ind>`.

1.3.4 Vectors

Creating → several methods:
- `(make-vector <length> <value>)` creates a vector of length `<length>` where each element has the value `<value>`;
- `(vector <el1> ... <eln>)` creates a vector containing the elements `<el1>` to `<eln>`;
- `[<el1> ... <eln>]` creates a vector without its elements being evaluated.

Assignment → `(fillarray <vect> <val>)` fills the vector `<vect>` with the values `<val>`.
Length → `(length <vect>)` gives the length of `<vect>`.
Extraction → `(aref <vect> <ind>)` returns the element of `<vect>` at the index `<ind>`.

Modifying → `(aset <vect> <ind> <val>)` assigns the value `<val>` to the element of `<vect>` at index `<ind>`.
Concatenating → `(vconcat <vect1> <vect2>)` returns a vector made with the elements of `<vect1>` followed by those of `<vect2>`.
Converting to a list → `(append <vect> nil)` returns a list containing the elements of `<vect>`.

1.4 Text Editing

The position is expressed in number of characters (starting at 1, at the upper left corner).

1.4.1 Cursor position

(point) → cursor position in number of characters, starting at the beginning of the file.
(region-beginning) → beginning of the selection.
(region-end) → end of the selection.
(line-beginning-position) → position of the beginning of the line from the beginning of the file.
(line-end-position) → position of the end of the line from the beginning of the file.
(what-line) → line number.
(point-min) → beginning of the file.
(point-max) → end of the file.
(push-mark) → set a mark at the current position.

1.4.2 Moving the cursor

(goto-char <n>) → move the cursor to position `<n>`.
(forward-char <n>) → move the cursor forward by `<n>` characters.
(backward-char <n>) → move the cursor backward by `<n>` characters.
(search-forward "<string>") → move the cursor to the end of the first occurrence of `<string>`.
(search-backward "<string>") → move the cursor at the beginning of he previous occurrence of `<string>`.
(skip-chars-forward "<regexp>") → forward search of the first character that is not in `<regexp>`.
(skip-chars-backward "<regexp>") → backward search of the first character that is not in `<regexp>`.
(save-excursion <body>) → execute `<body>`, then move the cursor back to its original position.

1.4.3 Editing

(delete-char <n>) → erase <n> characters, starting at the position of the cursor.
(delete-region <n> <m>) → erase the characters between positions `<n>` and `<m>`.
(insert "<string>") → insert `<string>` at the current position.
(setq x (buffer-substring <n> <m>)) → copy in x the text between positions `<n>` and `<m>`.
(capitalize-region <n> <m>) → convert in upper case letters the text between positions `<n>` and `<m>`.

---

2 WRITING FONCTIONS & SCRIPTS

2.1 Syntaxe of Functions

Simple function → `(defun <name> (<arg1> <arg2> etc) "Doc" (<code>))`.
Fonction interactive → `(defun <name> (<arg1> <arg2> etc) "Doc" (interactive) (<code>))` → once C-x C-e has been done on this function, it can be called by typing:
C-u <args> M-x <name> → if it has arguments;
M-x <name> → otherwise.
Macro → `(lambda <args> <body>)` defines a function with arguments `<args>` and instructions `<body>`.

2.2 Algorithmic

2.2.1 Decision

`(if <test> <body>)` → without alternative.
`(if <test> <bodytrue> <bodyfalse>)` → with alternative.
`(when <test> <expr1> <expr2>)` → other form without alternative.

2.2.2 Iteration

(while <test> <body>) → standard while loop.

2.2.3 Expression blocks

(progn <expr1> <expr2>) → execute <expr1> then <expr2>.

2.3 Scripts

2.3.1 Command line

Launch emacs <options> <file>.el args, with options:
-no-init-file → does not load the initialization file;
-batch → does not launch the editor and does not use the initialization file;
-load=<path> → execute elisp file in the <path>;
-script=<path> → equivalent to --batch --load=<path>;
args → optional arguments, which can be recovered with the variable argv.

---

3 USEFUL PREDEFINED FUNCTIONS

3.1 Buffers

(buffer-name) → buffer name.
(buffer-file-name) → name of the file where the buffer is saved.
(current-buffer) → the buffer itself.
(other-buffer) → the other buffer.
(set-buffer <name>) → set the buffer name to <name>.
(switch-to-buffer <buffer>) → toggle to buffer <name>.
(buffer-size) → returns the number of characters in the buffer.
(save-buffer) → save the buffer.
(kill-buffer "<name>") → closes the buffer <name>.

3.2 Files

(find-file "<file>") → open the file <file> in a buffer.
(write-file "<file>") → write the file <file>. Do not use while executing a script, rather use (write-region (point-min) (point-max) "<file>").
(write-region <n> <m> "<file>") → write a region in a file; <m> can be nil.
(insert-file-contents "<file>") → insert the content of a file <file> at the current position.
(append-to-file <n> <m> "<file>") → insert the text between positions <n> and <m> in the file <file>.
(rename-file "<oldname>" "<newname>") → rename the file.
(file-name-directory "<file>") → extract the directory in the path to file <file>.
(file-name-nondirectory "<file>") → extract the name of the file in the path to <file>.
(file-name-extension "<file>") → extract the extension of file <file>.
(file-name-nonextension "<file>") → extract the name without extension of file <file>.
(directory-files "<path>" <FULL> <MATCH> <NOSORT>) → return a list of files and directories in <path>, with the following optional arguments:
<FULL> → boolean indicating if the path is relative or absolute;
<MATCH> → string that must be in the names;
<NOSORT> → boolean indicating if the list is sorted.
(or load-file-name buffer-file-name) → find the name of the script currently called by emacs.

3.3 Lists

First element of a list → (car '(<a> <b> <c>)) ⇒ <a>.
Nth element in a list → (nth 2 '(<a0> <a1> <a2> <a3>)) ⇒ <a2>.
Removing the first element → (cdr '(<a> <b> <c>)) ⇒ <b> <c>.
Removing the Nth first elements → (nthcdr 3 '(<a1> <a2> <a3> <a4> <a5>)) ⇒ <a4> <a5>.
Adding elements to a list → (cons '<a> (<b> <c>)) ⇒ <a> <b> <c>.
Merging to lists → (append '<list1> '<list2>).
Lenght of a list → (length '<liste>).

Removing the last elements → `(butlast <list> <n>)` removes the last `<n>` elements of `<list>`. If `<n>` is absent, its value is assumed to be 1.
Adding elements at the begining of a list → `(cons <el> <list>)` returns `<list>` with the element `<el>` (can be a list) added at the beginning.
Suppressing elements → `(pop <list>)` removes the first element of the list and returns this element.
Remplacing elements → `(setcar <list> <el>)` replace the first element of `<list>` by `<el>`.

3.4 Sequences

A sequence can be a list, a vector or a string.
mapcar → `(mapcar '<func> <seq>)` apply the function `,func.` to all the elements of the sequence `<seq>`. The function returns the modified sequence.
mapc → `(mapc '<func> '<seq>)` similar to `mapcar`, but returns `nil`.
dolist → `(dolist (<var> <list> <result>) <body>)` returns in `<result>` (optional) the list `<list>` whose elements have been modified according to the instructions in `<body>` on variable `<var>`.
dotimes → `(dotimes (<var> <count> <result>) <body>)` returns in `<result>` (optional) the iteration `<count>` times the instructions of `<body>` on variable `<var>`.

3.5 Text Writing

Printing a string → `(print "<string>")`.
Same without carriage return → `(prin1 "<string>")`.
Formatting → `(print (format "<form>" <s1> ... <sn>))` print strings `<s1>` to `<sn>` following the format `<form>`. Formatting options are:
strings → `%s`;
decimal numbers → `%d`;
scientific notation of real numbers → `%e` or `%g`;
real with a floating decimal → `%f`.

3.6 Shell Commands

`(shell-command "<command>")` → execute the command `<command>`.
`(shell-command-to-string "<command>")` → exwcute the command `<command>` and returns the output in a list.
`(start-process-shell-command "<command>")` → launch the command `<command>` without waiting for the end of its execution.

---

4 USEFUL EXAMPLES

4.1 Inserting Text

```
(defun insert-lisp ()
  "Insert lisp environment at cursor point."
  (interactive)
  (insert "#+BEGIN_SRC lisp\n\n#+END_SRC")
  (backward-char 10))
```

4.2 Inserting Text Around a Region

```
(defun wrap-lisp-region ()
  "Put a region in a lisp environment."
  (interactive)
  (save-excursion
    (goto-char (region-end))
    (insert "\n#+END_SRC\n")
    (goto-char (region-beginning))
    (insert "#+BEGIN_SRC lisp\n")))
```

4.3 Selecting the Current Word

```
(defun select-current-word ()
  "Select the word under cursor.
“word” here is considered any alphanumeric sequence with “_” or “-”."
  (interactive)
  (let (pt)
    (skip-chars-backward "-_A-Za-z0-9")
    (setq pt (point))
    (skip-chars-forward "-_A-Za-z0-9")
    (set-mark pt)))
```

4.4 Selecting the Current Line

```
(defun select-current-line ()
  "Select the current line"
  (interactive)
  (let ((pos (line-beginning-position)))
    (end-of-line)
    (set-mark pos)))
```

4.5 Searching for the Current Word

```
(defun word-definition-lookup ()
"Look up the word under cursor in a browser."
 (interactive)
 (browse-url
   (concat "http://www.answers.com/main/ntquery?s=" (thing-at-point 'symbol))))
```

4.6 Reading a File and Converting It to a String

```
(defun get-string-from-file (filePath)
  "Return filePath's file content."
  (with-temp-buffer
    (insert-file-contents filePath)
    (buffer-string)))
```

4.7 Reading a File and Converting It to a List of Lines

```
(defun read-lines (filePath)
  "Return a list of lines of a file at filePath."
  (with-temp-buffer
    (insert-file-contents filePath)
    (split-string (buffer-string) "\n" t)))
```