# Elisp Reference Sheet

*Everything is a list!*

- To find out more about `name` execute `(describe-symbol 'name)`!
  - After the closing parens invoke `C-x C-e` to evaluate.
- To find out more about a key press, execute `C-h k` then the key press.

## Functions

- Function invocation: `(f` $x_0$ $x_1$ `...` $x_n$`)`. E.g., `(+ 3 4)` or `(message "hello")`.
  - After the closing parens invoke `C-x C-e` to execute them.
  - Only prefix invocations means we can use `-,+,*` in *names* since `(f+*- a b)` is parsed as applying function `f+*-` to arguments `a, b`.
    E.g., `(1+ 42)` $\to$ `43` using function *named* `1+` –the 'successor function'.
- Function definition:

```
(defun my-fun (arg0 arg1 ... argk)          ;; header, signature
  "This functions performs task ..."        ;; documentation, optional
  ...sequence of instructions to perform...  ;; body
)
```

- Anonymous functions: `(lambda (`$arg_0$ `...` $arg_k$`) bodyHere)`.

```
;; make and immediately invoke
((lambda (x y) (message (format "x, y ≈ %s, %s" x y))) 1 2)

;; make then way later invoke
(setq my-func (lambda (x y) (message (format "x, y ≈ %s, %s" x y))))
(funcall my-func 1 2)
;; (my-func 1 2) ;; invalid!
```

  The last one is invalid since `(f x0 ... xk)` is only meaningful for functions `f` formed using `defun`.
- Recursion and IO: `(defun sum (n) (if (<= n 0) 0 (+ n (sum (- n 1)))))`
  - Now `(sum 100)` $\to$ `5050`.
- IO: `(defun make-sum (n) (interactive "n") (message-box (format "%s" (sum n))))`
  - The `interactive` option means the value of `n` is queried to the user; e.g., enter 100 after executing `(execute-extended-command "" "make-sum")` or `M-x make-sum`.
  - In general `interactive` may take no arguments. The benefit is that the function can be executed using `M-x`, and is then referred to as an interactive function.

## Conditionals

- Booleans: `nil`, the empty list `()`, is considered *false*, all else is *true*.
  - Note: `nil` $\approx$ `()` $\approx$ `'()` $\approx$ `'nil`.
- `(if` ⟨`condition`⟩ ⟨`thenClause`⟩ ⟨`optionalElseClause`⟩`)`
  - Note: `(if x y)` $\approx$ `(if x y nil)`.

## List Manipulation

- Produce a syntactic, un-evaluated list, we use the single quote: `'(1 2 3)`.
- Construction: `(cons 'x0 '(x1 ... xk))` $\to$ `(x0 x1 ... xk)`.
- Head, or *contents of the address part of the register*: `(car '(x0 x1 ... xk))` $\to$ `x0`.
- Tail, or *contents of the decrement part of the register*: `(cdr '(x0 x1 ... xk))` $\to$ `(x1 ... xk)`.
- Deletion: `(delete e xs)` yields `xs` with all instance of `e` removed.
  - E.g., `(delete 1 '(2 1 3 4 1))` $\to$ `'(2 3 4)`.

E.g., `(cons 1 (cons "a" (cons 'nice nil)))` $\approx$ `(list 1 "a" 'nice)` $\approx$ `'(1 "a" nice)`.

## Variables

- Global Variables: `(setq name value)`; e.g., `(setq my-list '(1 2 3))`.
- Local Scope: `(let ((name0 val0) ... (namek valk)) ...use namei here... )`.
- Quotes: `'x` refers to the *name* rather than the *value* of `x`.
  - This is superficially similar to pointers: Given `int *x = ...`, `x` is the name (address) whereas `*x` is the value.
  - The quote simply forbids evaluation; it means *take it literally as you see it* rather than looking up the definition and evaluating.

```
(setq this 'hello)
(setq that this)

;;  this  → hello
;; 'this  → this
;;  that  → hello
;; 'that  → that
```

Note: `'x` $\approx$ `(quote x)`.

## Block of Code

Use the `progn` function to evaluate multiple statements. E.g.,

```
(progn
  (message "hello")
  (setq x  (if (< 2 3) 'two-less-than-3))
  (sleep-for 1)
  (message (format "%s" x))
)
```

## Reads

- How to Learn Emacs: A Hand-drawn One-pager for Beginners / A visual tutorial
- An Introduction to Programming in Emacs Lisp
- GNU Emacs Lisp Reference Manual

## Loops

Sum the first 10 numbers:

```
(let ((n 100) (i 0) (sum 0))
  (while (<= i n)
    (setq sum (+ sum i))
    (setq i   (+ i   1))
  )
  (message (number-to-string sum))
)
```

Essentially a for-loop:

```
(dotimes (x   ;; refers to current iteration, initally 0
          n   ;; total number of iterations
          ret ;; optional: return value of the loop
         )
  ...body here, maybe mentioning x...
)

;; E.g., sum of first n numbers
(let ((sum 0) (n 100))
  (dotimes (i (1+ n) sum) (setq sum (+ sum i))))
```

A for-each loop: Iterate through a list. Like `dotimes`, the final item is the expression value at the end of the loop.

```
(dolist (elem '("a" 23 'woah-there) nil)
  (message (format "%s" elem))
  (sleep-for 0 500)
)

(describe-symbol 'sleep-for) ;-)
```

## Hooks

Hooks are lists of functions that are called from Emacs Lisp in order to modify the behaviour of something. For example, different modes have their own hooks so that you can add functions that will run when that mode is initialised.
E.g., let's add the `go` function to the list of functions when a buffer is initialised with org-mode.

```
(describe-symbol 'org-mode-hook)

(defun go () (message-box "It worked!"))

  (add-hook 'org-mode-hook 'go)
≈ (add-to-list 'org-mode-hook 'go)

;; Now execute: (revert-buffer) to observe "go" being executed.
;; Later remove this silly function from the list:
(remove-hook 'org-mode-hook 'go)
```