

## 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.
- ◇ To find out more about the current mode you're in, execute `C-h m` or `describe-mode`. Essentially a comprehensive yet terse reference is provided.

### Functions

- ◇ Function invocation: (`f x0 x1 ... xn`). E.g., `(+ 3 4)` or `(message "hello")`.
  - After the closing parens invoke `C-x C-e` to execute them.
  - *Warning!* Arguments are evaluated **before** the function is executed.
  - 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)` → 43 using function *named* `1+` –the ‘successor function’.
- ◇ Function definition:

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

- The return value of the function is the result of the last expression executed.
- The documentation string may indicate the return type, among other things.

- ◇ Anonymous functions: `(lambda (arg0 ... argk) 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`. More honestly, Elisp has distinct namespaces for functions and variables.

Indeed, `(defun f (x0 ... xk) body) ≈ (fset 'f (lambda (x0 ... xk) body))`.

- Using `fset`, with quoted name, does not require using `funcall`.

- ◇ Recursion and IO: `(defun sum (n) (if (<= n 0) 0 (+ n (sum (- n 1)))))`
  - Now `(sum 100)` → 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.

- ◇ Global Variables, Create & Update: `(setq name value)`.
  - Generally: `(setq name0 value0 ... namek valuek)`. Use `defvar` for global variables since it permits a documentation string –but updates must be performed with `setq`. E.g., `~(defvar my-x 14 "my cool thing")`.
- ◇ Local Scope: `(let ((name0 val0) ... (namek valk)) bodyBlock)`.
  - `let*` permits later bindings to refer to earlier ones.
  - The simpler `let` indicates to the reader that there are no dependencies between the variables.
- ◇ Elisp is dynamically scoped: The caller's stack is accessible by default!

```
(defun woah ()
  "If any caller has a local 'work', they're in for a nasty bug
  from me! Moreover, they better have 'a' defined in scope!"
  (setq work (* a 111))) ;; Benefit: Variable-based scoped configuration.
```

```
(defun add-one (x)
  "Just adding one to input, innocently calling library method 'woah'."
  (let ((work (+ 1 x)) (a 6))
    (woah) ;; May change 'work' or access 'a'!
    work
  )
)

;; (add-one 2) ⇒ 666
```

Useful for loops, among other things:

C	Elisp
<code>x += y</code>	<code>(incf x y)</code>
<code>x--</code>	<code>(decf x)</code>
<code>x++</code>	<code>(incf x)</code>

- ◇ 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.
  - Note: `'x ≈ (quote x)`.
- Akin to English, quoting a word refers to the word and not what it denotes. (This lets us treat *code* as *data*! E.g., `'(+ 1 2)` evaluates to `(+ 1 2)`, a function call, not the value 3! Another example, `*` is code but `'*` is data, and so `(funcall '* 2 4)` yields 8.)
- ◇ *Atoms* are the simplest objects in Elisp: They evaluate to themselves.
  - E.g., 5, "a", 2.78, 'hello, lambda's form function literals in that, e.g., `(lambda (x) x)` evaluates to itself.

Elisp expressions are either atoms or function application –nothing else!

## Block of Code

Use the `progn` function to treat multiple expressions as a single expression. E.g.,

```
(progn
  (message "hello")
  (setq x (if (< 2 3) 'two-less-than-3))
  (sleep-for 0 500)
  (message (format "%s" x))
  (sleep-for 0 500)
  23      ;; Explicit return value
)
```

This' like curly-braces in C or Java. The difference is that the last expression is considered the 'return value' of the block.

Herein, a 'block' is a number of sequential expressions which needn't be wrapped with a `progn` form.

- ◊ Lazy conjunction and disjunction:
  - Perform multiple statements but stop when any of them fails, returns `nil`:  
(`and`  $s_0 s_1 \dots s_k$ ).  
★ Maybe monad!
  - Perform multiple statements until one of them succeeds, returns non-`nil`:  
(`or`  $s_0 s_1 \dots s_k$ ).

We can coerce a statement  $s_i$  to returning non-`nil` as so: (`progn`  $s_i$  `t`). Likewise, coerce failure by (`progn`  $s_i$  `nil`).

- ◊ Jumps, Control-flow transfer: Perform multiple statements and decide when and where you would like to stop.
  - (`catch` 'my-jump bodyBlock) where the body may contain (`throw` 'my-jump `returnValue`);  
the value of the catch/throw is then `returnValue`.
  - Useful for when the bodyBlock is, say, a loop. Then we may have multiple `catch`'s with different labels according to the nesting of loops.  
★ Possibly informatively named throw symbol is 'break.
  - Using name 'continue for the throw symbol and having such a catch/throw as *the body of a loop* gives the impression of continue-statements from Java.
  - Using name 'return for the throw symbol and having such a catch/throw as the body of a function definition gives the impression of, possibly multiple, return-statements from Java –as well as 'early exits'.
  - Simple law: (`catch` 'it  $s_0 s_1 \dots s_k$  (`throw` 'it  $r$ )  $s_{k+1} \dots s_{k+n}$ )  
 $\approx$  (`progn`  $s_0 s_1 \dots s_k$   $r$ ).  
★ Provided the  $s_i$  are simple function application forms.

## List Manipulation

- ◊ Produce a syntactic, un-evaluated list, we use the single quote: '(1 2 3).
- ◊ Construction: (`cons` 'x<sub>0</sub> '(x<sub>1</sub> ... x<sub>k</sub>)) → (x<sub>0</sub> x<sub>1</sub> ... x<sub>k</sub>).
- ◊ Head, or *contents of the address part of the register*: (`car` '(x<sub>0</sub> x<sub>1</sub> ... x<sub>k</sub>)) → x<sub>0</sub>.
- ◊ Tail, or *contents of the decrement part of the register*: (`cdr` '(x<sub>0</sub> x<sub>1</sub> ... x<sub>k</sub>)) → (x<sub>1</sub> ... x<sub>k</sub>).
- ◊ Deletion: (`delete`  $e$   $xs$ ) yields  $xs$  with all instance of  $e$  removed.
  - E.g., (`delete` 1 '(2 1 3 4 1)) → '(2 3 4).

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

Since variables refer to literals and functions have lambdas as literals, we can produce forms that take functions as arguments. E.g., the standard `mapcar` may be construed:

```
(defun my-mapcar (f xs)
  (if (null xs) xs
      (cons (funcall f (car xs)) (my-mapcar f (cdr xs)))))

(my-mapcar (lambda (x) (* 2 x)) '(0 1 2 3 4 5)) ;; => (0 2 4 6 8 10)
(my-mapcar 'uppercase '("a" "b" "cat")) ;; => ("A" "B" "CAT")

(describe-symbol 'remove-if-not) ;; "filter" ;-
```

## Conditionals

- ◊ Booleans: `nil`, the empty list `()`, is considered *false*, all else is *true*.
  - Note: `nil`  $\approx$  `()`  $\approx$  '()  
◦ (Deep structural) equality: (`equal`  $x$   $y$ ).
  - Comparisons: As expected; e.g., (`<=`  $x$   $y$ ) denotes  $x \leq y$ .
- ◊ (`if` condition thenExpr optionalElseBlock)
  - Note: (`if`  $x$   $y$ )  $\approx$  (`if`  $x$   $y$  `nil`); better: (`when`  $c$  thenBlock)  $\approx$  (`if`  $c$  (`progn` thenBlock)).
  - Note the else-clause is a 'block': Everything after the then-clause is considered to be part of it.
- ◊ Avoid nested if-then-else clauses by using a `cond` statement –a generalisation of switch statements.

```
(cond
  (test0
   actionBlock0)
  (test1
   actionBlock1)
  ...
  (t
   defaultActionBlock)) ;; optional
```

Sequentially evaluate the predicates `testi` and perform only the action of the first true test; yield `nil` when no tests are true.

- ◊ Make choices by comparing against *only* numbers or symbols –e.g., not strings!– with less clutter by using `case`:

```
(case 'boberto
  ('bob 3)
  ('rob 9)
  ('bobert 9001)
  (otherwise "You're a stranger!"))
```

With case you can use either `t` or `otherwise` for the default case, but it must come last.

## Exception Handling

We can attempt a dangerous clause and catch a possible exceptional case –below we do not do so via `nil`– for which we have an associated handler.

```
(condition-case nil attemptClause (error recoveryBody))

(ignore-errors attemptBody)
≈ (condition-case nil (progn attemptBody) (error nil))

(ignore-errors (+ 1 "nope")) ;; ⇒ nil
```

## 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, initially 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) ;:-)
```

## Example of Above Constructs

```
(defun my/cool-function (N D)
  "Sum the numbers 0..N that are not divisible by D"
  (catch 'return
    (when (< N 0) (throw 'return 0)) ;; early exit
    (let ((counter 0) (sum 0))
      (catch 'break
        (while 'true
          (catch 'continue
            (incf counter)
            (cond
              ((equal counter N) (throw 'break sum))
              ((zerop (% counter D)) (throw 'continue nil))
              (otherwise (incf sum counter))
            ))))))))
```

```
(my/cool-function 100 3) ;; ⇒ 3267
(my/cool-function 100 5) ;; ⇒ 4000
(my/cool-function -100 7) ;; ⇒ 0
```

Note that we could have had a final redundant `throw 'return`: Redundant since the final expression in a block is its return value.

The special loop constructs provide immensely many options to form nearly any kind of imperative loop. E.g., Python-style ‘downfrom’ for-loops and Java do-while loops. I personally prefer functional programming, so won't look into this much.

## Records

```
(defstruct X "Record with fields fi having defaults di"
  (f0 d0) ... (fk dk))

;; Automatic constructor is “make-X” with keyword parameters for
;; initialising any subset of the fields!
;; Hence (expt 2 (1+ k)) kinds of possible constructor combinations!
(make-X :f0 val0 :f1 val1 ... :fk valk) ;; Any, or all, fi may be omitted
```

*;; Automatic runtime predicate for the new type.*

```
(X-p (make-X)) ;; ⇒ true
(X-p 'nope)    ;; ⇒ nil
```

*;; Field accessors “X-f<sub>i</sub>” take an X record and yield its value.*

*;; Field update: (setf (X-f<sub>i</sub> x) val<sub>i</sub>)*

```
(defstruct book
  title (year 0))

(setq ladm (make-book :title "Logical Approach to Discrete Math" :year 1993))
(book-title ladm) ;; ⇒ "Logical Approach to Discrete Math"
(setf (book-title ladm) "LADM")
(book-title ladm) ;; ⇒ "LADM"
```

## Macros

- ◊ Sometimes we don't want eager evaluation; i.e., we do not want to recursively evaluate arguments before running a function.
- ◊ *Macros* let us manipulate a program's abstract syntax tree, usually by delaying some computations –easy since Emacs lets us treat code as data.

$(e_0 e_1 \dots e_k)$  is known as a *quasi-quote*: It produces syntactic data like `quote` but allows computations to occur –any expression prefixed with a comma as in `,e3`– thereby eagerly evaluating some and delaying evaluation of others.

```
(defun my/lazy-or (this that)
  (if this t that))

(my/lazy-or (print "yup") (print "nope")) ;; evaluates & prints both clauses!
(my/lazy-or t (/ 2 0)) ;; second clause runs needlessly, causing an error.

;; Delay argument evaluation using defmacro
(defmacro my/lazy-or-2 (this that)
  `(if ,this t ,that))

(my/lazy-or-2 (print "yup") (print "nope")) ;; just "yup" ;-)
(my/lazy-or-2 t (/ 2 0)) ;; second clause not evaluated

;; What code is generated by our macro?
(macroexpand '(my/lazy-or-2 t (/ 2 0))) ;; => (if t t (/ 2 0))
```

We've added new syntax to Emacs!

The above 'equations' can be checked by running `macroexpand`;  
e.g.,  $(\text{when } c \ s_0 \dots s_n) \approx (\text{if } c \ (\text{progn } s_0 \dots s_n) \ \text{nil})$  holds since

```
(macroexpand '(when c s0 ... sn)) ;; => (if c (progn s0 ... sn))
```

Woah!

## Hooks

- ◊ We can 'hook' methods to run at particular events.
- ◊ Hooks are lists of functions that are, for example, run when a 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-hook 'org-mode-hook '(lambda () (message-box "It worked!")))
≈ (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)
```

- ◊ The `'after-init-hook` event will run functions after the rest of the init-file has finished loading.

## Reads

- ◊ How to Learn Emacs: A Hand-drawn One-pager for Beginners / A visual tutorial
- ◊ Learn Emacs Lisp in 15 minutes — <https://learnxinyminutes.com/>
- ◊ An Introduction to Programming in Emacs Lisp
- ◊ GNU Emacs Lisp Reference Manual