# 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` $x_0$ $x_1$ ... $x_n$`)`. E.g., `(+ 3 4)` or `(message "hello")`.
  - After the closing parens invoke `C-x C-e` to execute them.
  - *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 (`$arg_0$ ... $arg_k$`) bodyHere)`.

```
;; make then way later invoke
(setq my-f (lambda (x y) (+ x y)))
(funcall my-f 1 2) ;; ⇒ 3
;; (my-f 1 2) ;; invalid!
(funcall my-f 1 2) ;; ⇒ 3
```

```
;; make and immediately invoke
(funcall (lambda (x y) (+ x y)) 1 2)

 ;; works, but is deprecated
((lambda (x y) (+ x y)) 1 2)
```

Functions are first-class values *but* variables and functions have **separate namespaces** —"Elisp is a Lisp-2 Language". The function represented by the name $g$ is obtained by the call `(function g)`, which is also denoted `#'g`. The sharp quote behaves like the usual quote but causes its argument to be compiled. `lambda` is a macro that calls `function` and so there is rarely any need to quote lambdas. If `h` is a variable referring to a function, then `(funcall h` $x_0$ ... $x_n$`)` calls that function on arguments $x_i$.

`(apply 'g` $x_0 \ldots x_k$ `'(`$x_k \ldots x_n$`))` ≈ `(funcall #'g` $x_0 \ldots x_n$`)` ≈ `(g` $x_0 \ldots x_n$`)`

```
;; Recursion with the 'tri'angle numbers: tri n = Σ_{i=0}^{n} i.
(defun tri (f n) (if (<= n 0) 0 (+ (funcall f n) (tri f (- n 1)))))
(tri #'identity 100)            ;; ⇒ 5050
(tri (lambda (x) (/ x 2)) 100) ;; ⇒ 2500

;; Run "C-h o tri" to see TWO items! Location determines dispatch.
(setq tri 100) (tri #'identity tri)      ;; ⇒ 5050
(setq tri (lambda (x) x)) (tri tri 100)  ;; ⇒ 5050
```

→ Use `funcall` or `apply` to call functions bound to variables.

→ Refer to functions outside of function calls by using a sharp quote, `#'`.

We may have positional `optional` arguments, or optional but named arguments —for which position does not matter. Un-supplied optional arguments are bound to `nil`.

```
(cl-defun f (a &optional b (c 5))
   (format "%s %s %s" a b c))

(f 'a)       ;; ⇒ "a nil 5"
(f 'a 'b)    ;; ⇒ "a b 5"
(f 'a 'b 'c) ;; ⇒ "a b c"
```

```
(cl-defun g (a &key (b 'nice) c)
   (format "%s %s %s" a b c))

(g 1 :c 3 :b 2) ;; ⇒ "1 2 3"
(g 1 :c 3)      ;; ⇒ "1 nice 3"
```

Keywords begin with a colon, `:k` is a constant whose value is `:k`.

## Quotes, Quasi-Quotes, and Unquotes

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.

*Elisp expressions are either atoms or function application –nothing else!*
'Atoms' are the simplest objects in Elisp: They evaluate to themselves;
e.g., `5, "a", 2.78, 'hello, [1 "two" three]`.

An English sentence is a list of words; if we want to make a sentence where some of the words are parameters, then we use a quasi-quote –it's like a quote, but allows us to evaluate data if we prefix it with a comma. It's usually the case that the quasi-quoted sentence happens to be a function call! In which case, we use `eval` which executes code that is in data form; i.e., is quoted.

Macros are essentially functions that return sentences, lists, which may happen to contain code.

```
;; Quotes / sentences / data
'(I am a sentence)
'(+ 1 (+ 1 1))

;; Executing data as code
(eval '(+ 1 (+ 1 1)))  ;; ⇒ 3
```

```
(setq name "Jasim")

;; Quasi-quotes: Sentences with a
;; computation, code, in them.
`(Hello ,name and welcome)
`(+ 1 ,(+ 1 1))  ;; ⇒ '(+ 1 2)
```

As the final example shows, Lisp treats data and code interchangeably. A language that uses the same structure to store data and code is called 'homoiconic'.

## 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 —also Land of Lisp
- GNU Emacs Lisp Reference Manual