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`.
- Functions are first-class values; the function represented by the name f is obtained by the call `#'f`.

```
;; Recursion with the 'tri'angle numbers: tri n = [0..n].
(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
```

IO: In general `interactive` may take no arguments. It lets us use `M-x` to execute functions; here, this results in `n` being queried to the user.

```
(defun double (n) (interactive "n") (message-box (format "%s" (* n 2))))
```

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`.

Variables

- Global Variables, Create & Update: `(setq name value)`.
  - Generally: `(setq name0 value0 ⋯ namek valuek)`.

  Use `devfar` 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.
- Any sequence of symbols is a valid identifier, including `x, x-y/z, --«==»--` and even ∀∃. Elisp names are case sensitive.
- 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
```

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
- Land of Lisp

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.)

- *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!

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.

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

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

```
(setq name "Jasim")

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

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'.

Lists and List-Like Structures

- Produce a syntactic, un-evaluated list, we use the single quote: `'(1 2 3)`.
- Construction: `(cons 'x0 '(x1 ... xk))` → `(x0 x1 ... xk)`.
- Head, or *contents of the address part of the register*: `(car '(x0 x1 ... xk))` → `x0`.
- Tail, or *contents of the decrement part of the register*: `(cdr '(x0 x1 ... xk))` → `(x1 ... xk)`.

E.g., `(cons 1 (cons "a" (cons 'nice nil))) ≈ (list 1 "a" 'nice) ≈ '(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 'upcase '("a" "b" "cat")) ;; ⇒ ("A" "B" "CAT")
```

Pairs: `(x . y)` ≈ `(cons x y)`.
An association list, or alist, is a list formed of such pairs. They're useful for any changeable collection of key-value pairs. The `assoc` function takes a key and an alist and returns the first pair having that key. In the end, alists are just lists.

(Rose) Trees in lisp are easily formed as lists of lists where each inner list is of length 2: The first symbol is the parent node and the second is the list of children.

Lists are formed by chains of cons cells, so getting and setting are very slow; likewise for alists. If performance is desired, one uses arrays and hash tables, respectively, instead. In particular, the performance of arrays and hash tables always requires a constant amount of time whereas the performance of lists and alists grows in proportion with their lengths. However, the size of an array is fixed —it cannot change and thus grow— and hash tables have a lookup cost as well as issues with "hash collisions". Their use is worth it for large amounts of data, otherwise lists are the way to go.

An array is created like a list but using [only square brackets] with getter `(aref arr index)`.
A hash table is created with `(make-hash-table)` with getter `(gethash key table)`.

What if you look up a key and get `nil`, is there no value for that key or is the value `nil`? `gethash` takes a final, optional, argument which is the value to return when the key is not found; it is `nil` by default.

Generic Setters

Since everything is a list in lisp, if `G` is a way to get a value from variable `x`, then `(setf G e)` updates `x` so that the location `G` now refers to element `e`. Hence, once you have a getter `G` you freely obtain a setter `(setf G ···)`.

```
;; Element update
(setq x '(0 1 2 3))     ;; x ⇒ '(0 1 2 3)
(setf (nth 2 x) 'nice) ;; x ⇒ '(0 1 'nice 3)

;; Circular list
(setq y '(a b c))    ;; y ⇒ '(a b c)
(setf (cdddr y) y)   ;; y ⇒ '(a b c a b . #2)
;; "#2" means repeat from index 2.
(nth 99 y) ;; ⇒ a
```

Records

If we want to keep a list of related properties in a list, then we have to remember which position keeps track of which item and may write helper functions to keep track of this. Instead we could use a structure.

```
(defstruct X "Record with fields/slots 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_i" take an X record and yield its value.

;; Field update: (setf (X-f_i x) val_i)

(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"
```

Advanced OOP constructs can be found within the CLOS, Common Lisp Object System; which is also used as a research tool for studying OOP ideas.

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 ... s_k`)`.
    - ⋆ Maybe monad!
  - Perform multiple statements until one of them succeeds, returns non-`nil`: `(or` s_0 s_1 ... 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. This' akin to C's `goto`'s; declare a label with `catch` and goto it with `throw`.
  - `(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 ... s_k `(throw 'it r)` s_{k+1} ⋯ s_{k+n}`)` ≈ `(progn` s_0 s_1 ⋯ s_k `r)`.
    - ⋆ Provided the s_i are simple function application forms.
- ◇ `and, or` can be thought of as instance of catch/throw, whence they are control flow first and Boolean operations second.

  `(and` s_0 ⋯ s_n `e)` ⇒ when all x_i are true, do e
  `(or` s_0 ⋯ s_n `e)` ⇒ when no x_i is true, do e

Conditionals

- ◇ Booleans: `nil`, the empty list `()`, is considered *false*, all else is *true*.
  - Note: `nil` ≈ `()` ≈ `'()` ≈ `'nil`.
  - (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)` ≈ `(if x y nil)`;
    better: `(when c thenBlock)` ≈ `(if c (progn thenBlock))`.
  - Note the else-clause is a 'block': Everything after the then-clause is considered to be part of it.
  - `(if xs ⋯)` means "if xs is nonempty then ⋯" is akin to C style idioms on linked lists.

Avoid nested if-then-else clauses by using a `cond` statement –a (lazy) generalisation of switch statements. Or make choices by comparing against *only* numbers or symbols —e.g., not strings!– with less clutter by using `case`.

```
(cond
  (test_0
    actionBlock_0)
  (test_1
    actionBlock_1)
  ...
  (t                        ;; optional
    defaultActionBlock))
```

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

`cond` sequentially evaluates the predicates $\texttt{test}_i$ and performs only the action of the first true test; yielding `nil` when no tests are true.

Hint: If you write a predicate, think of what else you can return besides `t`; such as a witness to why you're returning truth –all non-nil values denote true after all. E.g., `(member e xs)` returns the sublist of `xs` that begins with `e`.

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

Let's sum the first 100 numbers in 3 ways.

```
(let ((n 100) (i 0) (sum 0))
  (while (<= i n)
    (incf sum i) (incf i))
    (message (format "sum: %s" sum)))
```

C	Elisp
`x += y`	`(incf x y)`
`x -= y`	`(decf x y)`

`y` is optional, and is 1 by default.

Two instances of a while loop:

```
;; Repeat body n times, where i is current iteration.
(let ((result 0) (n 100))
  (dotimes (i (1+ n) result) (incf result i)))

;; A for-each loop: Iterate through the list [0..100].
(let ((result 0) (mylist (number-sequence 0 100)))
  (dolist (e mylist result) (incf result e)))
```

In both loops, `result` is optional and defaults to nil. It is the return value of the loop expression.

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
```

The special loop construct provides 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 wont look into this much.

Types & Overloading

Since Lisp is dynamically typed, a variable can have any kind of data, possibly different kinds if data at different times in running a program. We can use `type-of` to get the type of a given value; suffixing that with `p` gives the associated predicate; e.g., `function` ↦ `functionp`.

```
;; Difficult to maintain as more types are added.
(defun sad-add (a b)
  (if (and (numberp a) (numberp b))
      (+ a b)
      (format "%s + %s" a b))
)

(sad-add 2 3)       ;; ⇒ 5
(sad-add 'nice "3") ;; ⇒ "nice + 3"

;; Better: Seperation of concerns.
;;
(cl-defmethod add ((a number) (b number)) (+ a b))      ;; number types
(cl-defmethod add ((a t) (b t)) (format "%s + %s" a b)) ;; catchall types

(add 2 3)       ;; ⇒ 5
(add 'nice "3") ;; ⇒ "nice + 3"
```

While list specific functions like list-length and mapcar may be more efficient than generic functions, which require extra type checking, the generic ones are easier to remember. The following generic functions work on lists, arrays, and strings:

- `find-if`, gets first value satisfying a predicate.
- `count`, finds how often an element appears in a sequence
- `position`, finds the index of a given element.
- `some`, check if any element satisfies a given predicate
- `every`, check if every element satisfies the given predicate
- `reduce`, takes a binary operation and a sequence and mimics a for-loop. Use keyword `:initial-value` to specify the starting value, otherwise use head of sequence.
- `sum`, add all numbers; crash for strings.
- `length, subseq, sort`.

Macros

Macros let us add new syntax, like `let1` for single lets:

```
;; Noisy parens!
(let ((x "5")) (message x))

;; Better.
(let1 x "5" (message x))

;; How?
(defmacro let1 (var val &rest body)
 `(let ((,var ,val)) ,@body))

;; What does it look like?
(macroexpand
  '(let1 x "5" (message x)))
;; ⇒ (let ((x 5)) (message x))
```

```
;; No progn; (first x y z) ≈ x
(defmacro first (&rest body)
 (car `,@body))

;; Need to use "progn"!
(defmacro not-first (&rest body)
 `(progn ,@(cdr `,@body)))

(macroexpand '(not-first x y z))
;; `,@body        ⇒ (x y z)
;; (cdr `,@body)  ⇒ (y z)
;; `(progn ,@(cdr `,@body))
;;          ⇒ (progn y z)
```

1. Certain problems are elegantly solved specific language constructs; e.g., list operations are generally best defined by pattern matching.

2. Macros let us *make* the best way to solve a problem when our language does not give it to us.

3. Macro expansion happens before runtime, function execution, and so the arguments passed to a macro will contain raw source code.

   Backquotes let us use the comma to cause the actual variable *names* and *values* to be used –e.g., `x` is a 'meta-variable' and its value, `,x`, refers to a real variable or value.

   The `&rest` marker allows us to have multiple statements at the end of the macro: The macro expander provides all remaining expressions in the macro as a list, the contents of which may be inserted in place, not as a list, using the `,@` splice comma –we need to ensure there's a `progn`.

   `` `(pre ,@(list $s_0 \cdots s_n$) post) ≈ `(pre $s_0 \cdots s_n$ post) ``.

◇ `macroexpand` takes *code* and expands any macros in it. It's useful in debugging macros. The above 'equations' can be checked by running `macroexpand`; e.g., `(when c` $s_0 \cdots s_n$`)` ≈ `(if c (progn` $s_0 \cdots s_n$`) nil)` holds since:

`(macroexpand '(when c` $s_0 \cdots s_n$`))` *;; ⇒ (if c (progn* $s_0 \cdots s_n$*))*

◇ If `var` is an argument to a macro where `,var` occurs multiple times, then since arguments to macros are raw source code, each occurrence of `,var` is an execution of the code referenced by `var`.

Avoid such repeated execution by using a `let` to capture the result, call it `res`, once and use the `res` in each use site.

Now we've made use of the name `res` and our users cannot use that name correctly. Avoid such *unintended* capture by using `gensym` to provide us with a globally unique name which is bound to a variable, say `r`, then we bind the result of executing `var` to the fresh name `,r`.

Whence: `` `(`` $\cdots$ `,var` $\cdots$ `,var` $\cdots$ `)`
⇒ `(let ((r (gensym))) `` `(let ((,r ,var)) ` $\cdots$ `,r` $\cdots$ `,r` $\cdots$ `))`.

Note that the name `r` is outside the backquote; it is part of code that is run at macro expansion time, not runtime. The value of the final `let` is then the backquoted matter, which makes *no* reference to `r`, but instead makes use of the name it refers to, `,r`. Neato!

Ex., remove repeated execution from `(defmacro twice (var) `` `(list ,var ,var))`.

◇ Test that you don't have accidentally variable capture by passing in an insert statement and see how many times insertions are made.

◇ Macros that *intentionally* use variable capture as a feature, rather than a bug, to provide names available in the macro body are called 'anaphoric macros'.

E.g., `(split list no yes)` provides the names `head, tail` in the `yes` body to refer to the head and tail of the given `list`, say via a `let`, but not so in the `no` argument for when `list` is empty. Whence, elegant pattern matching on lists.

Exercise: Define `split`.

`read` and `print`

'Reading' means parsing an expression in textual form and producing a lisp object. E.g., this is a way to load a lisp file. 'Printing' a lisp object mean producing a textual representation. These operations, in lisp, are mostly inverse.

The `read-from-string` command works just like the `read` command, but lets us read a lisp object from a string instead of directly from the console.

```
(defun sum-two ()
  (let (fst snd)
    (setq fst (read))
    (setq snd (read))
    (+ (eval fst) (eval snd))
  )
)

;; Run (sum-two) with inputs (+ 1 2) and (* 3 4)  ;-)
```

Lisp makes writing a REPL astonishingly easy: "Loop as follows: Print the result of evaluating what is read at the prompt."

```
(loop (print (eval (read))))  ;; Beautiful (●●)
```

◇ `loop` merely loops forever.

The `print` and `read` commands work on all kinds of data, such as lists of data structures. Hence, we must use quotes if we want to read a string rather than a symbol, for example.

A major problem with this REPL is that `eval` executes any, potentially malicious, Lisp command entered by the user. Ideally one checks the read lisp object is safe —say, it is one of some allowable commands— and only then evaluates it.