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 (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.

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.
- 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
`x += y`	`(incf x y)`
`x--`	`(decf x)`
`x++`	`(incf x)`

- 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 `...` 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.
  - `(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} $\cdots$ s_{k+n}`)` $\approx$ `(progn` s_0 s_1 $\cdots$ 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_0 `'(`x_1 `...` x_k`))` → `(`x_0 x_1 `...` x_k`)`.
- Head, or *contents of the address part of the register*: `(car '(`x_0 x_1 `...` x_k`))` → x_0.
- Tail, or *contents of the decrement part of the register*: `(cdr '(`x_0 x_1 `...` x_k`))` → `(`x_1 `...` x_k`)`.
- 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)))` ≈ `(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")

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

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.
- Avoid nested if-then-else clauses by using a `cond` statement –a generalisation of switch statements.

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

  Sequentially evaluate the predicates $\texttt{test}_i$ 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, 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) ;-)
```

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 wont 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-fᵢ" take an X record and yield its value.

;; Field update: (setf (X-fᵢ x) valᵢ)

(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

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`.

   $\texttt{`(pre ,@(list } s_0 \cdots s_n\texttt{) post)} \approx \texttt{`(pre } s_0 \cdots s_n\texttt{ post)}$.

◇ `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., $\texttt{(when c } s_0 \cdots s_n\texttt{)} \approx \texttt{(if c (progn } s_0 \cdots s_n\texttt{) nil)}$ holds since:

$\texttt{(macroexpand '(when c } s_0\ \boxed{\cdots}\ s_n\texttt{))}$;; ⇒ $\texttt{(if c (progn } s_0 \cdots s_n\texttt{))}$

◇ 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: `` `(⋯,var⋯,var⋯) ``
⇒ `` (let ((r (gensym))) `(let ((,r ,var)) ⋯,r⋯,r⋯)) ``.

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`.

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