Everything is a list!

- To find out more about `name` execute `(describe-symbol 'name)!`
  - After the closing paren(s) invoke `C-x C-C` 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 paren(s) invoke `C-x C-C` to evaluate them.
  - `Warning!` Arguments are evaluated before the function is executed.
  - Only prefix invocations means we can use `-`, `*`, `in` 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:
  ```elisp
  ;; "define" fun"ctions
  (defun my-fun (arg₀ ... argₙ) ;; 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 ... argₙ)) bodyHere).
  ;; make then way later invoke
  (setq my-f (lambda (x y) (+ x y))) ;; make and immediately invoke
  ;; (funcall my-f 1 2);; ⇒ 3
  (funcall (lambda (x y) (+ x y)) 1 2);; ;; works, but is deprecated
  (funcall my-f 1 2);; ⇒ 3

  ;; Recursion with the ‘tri’angle numbers:
  ;; tri n = (f x₀ x₁ ... xₙ) calls that function on arguments xₙ...
  (apply 'g x₀ ... xₚ 'xₚ...xₙ);; ⇒ (funcall #g x₀ ... xₙ) ≈ (g x₀ ... xₙ)
  ;; Recursion with the ‘tri’angle numbers: tri 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 (lambda (x) x));; ⇒ 5050
  (setq tri (lambda (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.

```elisp
(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"
```

```elisp
(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, `:` is a constant whose value is `:`

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 makes `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.

```elisp
(setq name "Jasim")
(i am a sentence)
(+ 1 (+ 1 1))
('Hello ,name and welcome)
(eval '('+ 1 (+ 1 1))) ;; ⇒ 3
```

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 ‘homeononic’.

Reads

- How to Learn Emacs: A Hand-drawn One-pager for Beginners / A visual tutorial
- Learn Emacs Lisp in 15 minutes — https://learninminutes.com/
- An Introduction to Programming in Emacs Lisp — also Land of Lisp
- GNU Emacs Lisp Reference Manual
Variables

- Global Variables, Create & Update: \((\text{setq name value})\).
  - Generally: \((\text{setq name}_0 \text{ value}_0 \ldots \text{ name}_k \text{ value}_k)\).
  Use \texttt{defvar} for global variables since it permits a documentation string—but updates must be performed with \texttt{setq}. E.g., \((\text{defvar my-x 14 "my cool thing"})\).

\[
\text{(setq x y)} \approx \text{(setq (quote x) y)}
\]

Variables are assigned with \texttt{set}, which takes a quoted identifier, so that it’s not evaluated, and a value to associate to that variable. “set quoted”, \texttt{setq}, avoids the hassle of quoting the name. More generally, \((\text{set sym v})\) assigns the value of \texttt{sym} to have the value \texttt{v}.

- Local Scope: \((\text{let ((name}_0 \text{ val}_0) \ldots \text{ (name}_k \text{ val}_k) \text{ bodyBlock})\).
  - \texttt{let*} permits later bindings to refer to earlier ones.
  - The simpler \texttt{let} indicates to the reader that there are no dependencies between the variables.
  - \texttt{let*} \(\approx\) parallel; \texttt{let*} \(\approx\) sequential.
  - Local functions declared with \texttt{flet} and \texttt{flet*}; e.g., \((\text{flet ((go (x) (+ 2 x)) (go 3)))\).

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

- Elisp is dynamically scoped: The caller’s stack is accessible by default!

\[
\text{(defun woah ()}
\text{
"If any caller has a local ‘work’, they’re in for a nasty bug from me! Moreover, they better have ‘a’ defined in scope!"}
\text{(setq work \((* a 111))) \;; Benefit: Variable-based scoped configuration.}
\text{(woah) \;; May change ‘work’ or access ‘a’!
work}
\text{)}
\text{)}
\text{
\;; (add-one 2) \(\Rightarrow\) \text{666}
\text{)}
\]

Lists and List-Like Structures

- Produce a syntactic, un-evaluated list, we use the single quote: \(’(1 2 \text{3})\).
- Construction: \((\text{cons } x_0 \ ’(x_1 \ldots \ x_k)) \Rightarrow \ (x_0 \ x_1 \ldots \ x_k)\).
- Head, or contents of the address part of the register: \((\text{car } ”(x_0 \ x_1 \ldots \ x_k)) \Rightarrow \ x_0\).
- Tail, or contents of the decrement part of the register: \((\text{cdr } ”(x_0 \ x_1 \ldots \ x_k)) \Rightarrow \ (x_1 \ldots \ x_k)\).

E.g., \((\text{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 \texttt{mapcar} may be construed:

\[
\text{\((\text{defun my-mapcar (f xs)}\)}
\text{\((\text{if (null xs) xs)}\)}
\text{\((\text{cons (funcall f (car xs)) (my-mapcar f (cdr xs)))})\))
\text{\((\text{my-mapcar (lambda (x) \((* 2 x)) \((0 1 2 3 4 5)) \;; \Rightarrow \((0 2 4 6 8 10)\))\)}
\text{\((\text{my-mapcar ’upcase ("a" ”b” ”cat")})\)}
\text{\((\Rightarrow ("A" ”B” ”CAT")\))}
\]

Pairs: \((x\ ,\ y) \(\approx\) (cons x y)\).

An association list, or list, is a list formed of such pairs. They’re useful for any changeable collection of key-value pairs. The \texttt{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 \texttt{(aref arr index)}.

A hash table is created with \texttt{(make-hash-table)} with getter \texttt{(gethash key table)}.

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

Generic Setters

Since everything is a list in lisp, if \texttt{G} is a way to get a value from variable \texttt{x}, then \((\text{setq G e})\) updates \texttt{x} so that the location \texttt{G} now refers to element \texttt{e}. Hence, once you have a getter \texttt{G} you freely obtain a setter \((\text{setq G \(\ldots\)})\).

\[
\text{;; Element update}
\text{(setq x \((0 1 2 3)) \;; x \Rightarrow \((0 1 2 3)\))}
\text{(setq (nth 2 x) ’nice) \;; x \Rightarrow \((0 1 ’nice 3)\))}
\]

\[
\text{;; Circular list}
\text{(setq y \((’a b c)) \;; y \Rightarrow \((’a b c)\))}
\text{(setq (cddr y) y) \;; y \Rightarrow \((’a b c a b . #2)\))}
\text{;; ”#2” means repeat from index 2.}
\text{(nth 99 y) \;; \Rightarrow 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.

(defun make-X (f0 d0) ... (fk dk))

;; Automatic constructor is "make-X" with keyword parameters for
;; initialising any subset of the fields!
;; Hence (ezpt 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) val)

(defstruct X "Record with fields/slots f, having defaults d,"
 (f0 d0) ... (fk dk))

(setf (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 s0 s1 ... sk).
  * Maybe monad!
  - Escape from the given conditions:
  - Perform multiple statements until one of them succeeds, returns non-nil: (or s0 s1 ... sk).

- 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.

- Conditional expressions:
  - 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 s0 s1 ... sk (throw 'it r) sk+1 ... sk+n) ⇒ (progn s0 s1 ... sk x).
* Provided the s 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.

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 ≤ y.
  - (if condition thenExpr optionalElseBlock)
    - Note: (if x y nil): better: (when c thenBlock) ≈ (if (progn thenBlock)).
    - Note the else-clause is a 'block': Everything after the then-clause is consid-
      ered to be part of it.
    - (if xs ...) means "if xs is nonempty then ...": is akin to C style idioms on
      linked lists.

(defun go (x)
  (pcase x
    ('bob 1972)
    ('((a ,.. ,c) (+ a c))
      (otherwise "Shucks!")))
  ;; pattern matching on any type
  (go 'hallo) ;; ⇒ "Shucks!"
  (go 'bob) ;; ⇒ 1972
  (go '(1 2 3)) ;; ⇒ 4
  (go 'bob) ;; ⇒ 1972
  (go 'hallo) ;; ⇒ "Shucks!"
  ;; 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 s0 s1 ... sk).
  ;;    * Maybe monad!
  ;;  - Escape from the given conditions:
  ;;    - Perform multiple statements until one of them succeeds, returns non-nil: (or s0 s1 ... sk).
  ;;  - 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.
  ;;  - Conditional expressions:
  ;;    - 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 s0 s1 ... sk (throw 'it r) sk+1 ... sk+n) ⇒ (progn s0 s1 ... sk x).
  ;;      * Provided the s 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.
  ;;- 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 ≤ y.
  ;;- (if condition thenExpr optionalElseBlock)
  ;;  - Note: (if x y nil): better: (when c thenBlock) ≈ (if (progn thenBlock)).
  ;;  - Note the else-clause is a 'block': Everything after the then-clause is consid-
  ;;    ered to be part of it.
  ;;  - (if xs ...) means "if xs is nonempty then ...": is akin to C style idioms on
  ;;    linked lists.
  ;;(defun go (x)
  ;;  (pcase x
  ;;    ('bob 1972)
  ;;    ('((a ,.. ,c) (+ a c))
  ;;      (otherwise "Shucks!")))
  ;;  ;; pattern matching on any type
  ;;  (go 'hallo) ;; ⇒ "Shucks!"
  ;;  (go 'bob) ;; ⇒ 1972
  ;;  (go '(1 2 3)) ;; ⇒ 4
  ;;  (go 'bob) ;; ⇒ 1972
  ;;  (go 'hallo) ;; ⇒ "Shucks!"
Avoid nested if-then-else clauses by using a \texttt{cond} statement—a (lazy) generalisation of switch statements: It sequentially evaluates the expressions \texttt{test}, and performs only the action of the first true test; yielding \texttt{nil} when no tests are true. Or use pattern matching; which even allows predicates in the case position—\texttt{C-h o ;)}

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

\begin{itemize}
  \item \textbf{Loops}
  \begin{enumerate}
    \item Let’s sum the first 100 numbers in 3 ways.
      \begin{verbatim}
      (let ((n 100) (i 0) (sum 0))
        (while (<= i n)
          (incf sum i) (incf i))
        (message (format "sum: \(\%s\) sum\))
      )
    \end{verbatim}
    \texttt{C} \hspace{1cm} \texttt{Elisp}
    \texttt{x += y (incf x y)} \hspace{1cm} \texttt{x -= y (decf x y)}
    \texttt{y} is optional, and is 1 by default.

    \begin{verbatim}
    ;; 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)))
    \end{verbatim}

    In both loops, \texttt{result} is optional and defaults to \texttt{nil}. It is the return value of the loop expression.

    \textbf{Example of Above Constructs}
    \begin{verbatim}
    (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 ))) )))
          (throw 'return sum)))
    )
    \end{verbatim}

    \begin{verbatim}
    (my/cool-function 100 3) ;; ⇒ 3267
    (my/cool-function 100 5) ;; ⇒ 4000
    (my/cool-function -100 7) ;; ⇒ 0
    \end{verbatim}

    The special loop construct provides immensely many options to form nearly any kind of imperative loop. E.g., Python-style ‘from’-for-loops and Java do-while loops. I personally prefer functional programming, so won’t look into this much.
  \end{enumerate}
\end{itemize}

\textbf{Exception Handling}

We can attempt a dangerous clause and catch a possible exceptional case—below we do not do so via \texttt{nil}—for which we have an associated handler.

\begin{verbatim}
(condition-case nil attemptClause (error recoveryBody))

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

(ignore-errors (+ 1 "nope")) ;; ⇒ nil
\end{verbatim}

\textbf{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 \texttt{type-of} to get the type of a given value; suffixing that with \texttt{p} gives the associated predicate; e.g., \texttt{function \(\rightarrow\) functionp}.

\begin{verbatim}
;; 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: Separation 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"
\end{verbatim}

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:

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

dash is a modern list library for Emacs that uses Haskell-like names for list operations—\texttt{)} Likewise, \texttt{s} is a useful Emacs string manipulation library. In-fact, we can write Emacs extensions using Haskell directly.
Macros let us add new syntax, like `let1` for single lets:

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

;; Better:
(let1 x "5" (message x))
```

## Macros

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

```lisp
;; 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))))
```

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`.

Use list elements in-place:

```lisp
‘(pre ,@((list s₀ ··· sₙ) post)) ≈ ‘(pre s₀ ··· sₙ 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 s₀ ··· sₙ) ≈ (if c (progn s₀ ··· sₙ) nil)` holds since:
  ```lisp
  (macroexpand ‘(when c s₀ ··· sₙ)) ;; ⇒ (if c (progn s₀ ··· sₙ))
  ```
- 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`.

```lisp
(when c s₀ ··· sₙ) ⇒ (if c (progn s₀ ··· sₙ))
```

Whence: `‘(...,var···,var···)`

```lisp
≈ (let ((r (gensym))) ‘(let ((,r ,var)) ··· ,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`.

---

### 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.

```lisp
(defun sum-two ()
  (let (fst snd)
    (setq snd (read))
    (setq fst (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.”

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

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

- `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.