Common-Lisp

ANSI Common Lisp is a general purpose, multi-paradigm programming language suited for a wide variety of industry applications. It is frequently referred to as a programmable programming language.

The classic starting point is Practical Common Lisp and freely available.

Another popular and recent book is Land of Lisp.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 0. Syntax
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; General form.

;; Lisp has two fundamental pieces of syntax: the ATOM and the
;; S-expression. Typically, grouped S-expressions are called `forms`.

10  ; an atom; it evaluates to itself

:THING ;Another atom; evaluating to the symbol :thing.

t  ; another atom, denoting true.

(+ 1 2 3 4) ; an s-expression

'(4 :foo  t)  ;another one


;;; Comments

;; Single line comments start with a semicolon; use two for normal
;; comments, three for section comments, and four for file-level
;; comments.

#| Block comments
   can span multiple lines and...
    #|
       they can be nested!
    |#
|#

;;; Environment.

;; A variety of implementations exist; most are
;; standard-conformant. CLISP is a good starting one.

;; Libraries are managed through Quicklisp.org's Quicklisp system.

;; Common Lisp is usually developed with a text editor and a REPL
;; (Read Evaluate Print Loop) running at the same time. The REPL
;; allows for interactive exploration of the program as it is "live"
;; in the system.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Symbols

'foo ; => FOO  Notice that the symbol is upper-cased automatically.

;; Intern manually creates a symbol from a string.

(intern "AAAA") ; => AAAA

(intern "aaa") ; => |aaa|

;;; Numbers
9999999999999999999999 ; integers
#b111                  ; binary => 7
#o111                  ; octal => 73
#x111                  ; hexadecimal => 273
3.14159s0              ; single
3.14159d0              ; double
1/2                    ; ratios
#C(1 2)                ; complex numbers


;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use ' to stop it from
;; being evaluated - literally, "quote" the data.
'(+ 1 2) ; => (+ 1 2)
;; You can also call a function manually:
(funcall #'+ 1 2 3) ; => 6
;; Some arithmetic operations
(+ 1 1)              ; => 2
(- 8 1)              ; => 7
(* 10 2)             ; => 20
(expt 2 3)           ; => 8
(mod 5 2)            ; => 1
(/ 35 5)             ; => 7
(/ 1 3)              ; => 1/3
(+ #C(1 2) #C(6 -4)) ; => #C(7 -2)

                     ;;; Booleans
t                    ; for true (any not-nil value is true)
nil                  ; for false - and the empty list
(not nil)            ; => t
(and 0 t)            ; => t
(or 0 nil)           ; => 0

                     ;;; Characters
#\A                  ; => #\A
#\                   ; => #\GREEK_SMALL_LETTER_LAMDA
#\u03BB              ; => #\GREEK_SMALL_LETTER_LAMDA
```

```
;;; Strings are fixed-length arrays of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character

;; Strings can be concatenated too!
(concatenate 'string "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a sequence of characters
(elt "Apple" 0) ; => #\A

;; format can be used to format strings:
(format nil "~a can be ~a" "strings" "formatted")

;; Printing is pretty easy; ~% is the format specifier for newline.
(format t "Common Lisp is groovy. Dude.~%")


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create a global (dynamically scoped) using defparameter
;; a variable name can use any character except: ()",'`;#|\

;; Dynamically scoped variables should have earmuffs in their name!

(defparameter *some-var* 5)
*some-var* ; => 5

;; You can also use unicode characters.
(defparameter *AΛB* nil)


;; Accessing a previously unbound variable is an
;; undefined behavior (but possible). Don't do it.


;; Local binding: `me` is bound to "dance with you" only within the
;; (let ...). Let always returns the value of the last `form` in the
;; let form.

(let ((me "dance with you"))
  me)
;; => "dance with you"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Structs and Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Structs
(defstruct dog name breed age)
(defparameter *rover*
    (make-dog :name "rover"
              :breed "collie"
              :age 5))
*rover* ; => #S(DOG :NAME "rover" :BREED "collie" :AGE 5)

(dog-p *rover*) ; => true  #| -p signifies "predicate". It's used to
                               check if *rover* is an instance of dog. |#
(dog-name *rover*) ; => "rover"

;; Dog-p, make-dog, and dog-name are all created by defstruct!

;;; Pairs
;; `cons' constructs pairs, `car' and `cdr' extract the first
;; and second elements
(cons 'SUBJECT 'VERB) ; => '(SUBJECT . VERB)
(car (cons 'SUBJECT 'VERB)) ; => SUBJECT
(cdr (cons 'SUBJECT 'VERB)) ; => VERB

;;; Lists

;; Lists are linked-list data structures, made of `cons' pairs and end
;; with a `nil' (or '()) to mark the end of the list
(cons 1 (cons 2 (cons 3 nil))) ; => '(1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => '(1 2 3)
;; and a quote can also be used for a literal list value
'(1 2 3) ; => '(1 2 3)

;; Can still use `cons' to add an item to the beginning of a list
(cons 4 '(1 2 3)) ; => '(4 1 2 3)

;; Use `append' to - surprisingly - append lists together
(append '(1 2) '(3 4)) ; => '(1 2 3 4)

;; Or use concatenate -

(concatenate 'list '(1 2) '(3 4))

;; Lists are a very central type, so there is a wide variety of functionality for
;; them, a few examples:
(mapcar #'1+ '(1 2 3))             ; => '(2 3 4)
(mapcar #'+ '(1 2 3) '(10 20 30))  ; => '(11 22 33)
(remove-if-not #'evenp '(1 2 3 4)) ; => '(2 4)
(every #'evenp '(1 2 3 4))         ; => nil
(some #'oddp '(1 2 3 4))           ; => T
(butlast '(subject verb object))   ; => (SUBJECT VERB)


;;; Vectors

;; Vector's literals are fixed-length arrays
#(1 2 3) ; => #(1 2 3)

;; Use concatenate to add vectors together
(concatenate 'vector #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)
```

```
;;; Arrays

;; Both vectors and strings are special-cases of arrays.

;; 2D arrays

(make-array (list 2 2))

;; (make-array '(2 2)) works as well.

; => #2A((0 0) (0 0))

(make-array (list 2 2 2))

; => #3A(((0 0) (0 0)) ((0 0) (0 0)))

;; Caution- the default initial values are
;; implementation-defined. Here's how to define them:

(make-array '(2) :initial-element 'unset)

; => #(UNSET UNSET)

;; And, to access the element at 1,1,1 -
(aref (make-array (list 2 2 2)) 1 1 1)

; => 0

;;; Adjustable vectors

;; Adjustable vectors have the same printed representation
;; as fixed-length vector's literals.

(defparameter *adjvec* (make-array '(3) :initial-contents '(1 2 3)
      :adjustable t :fill-pointer t))

*adjvec* ; => #(1 2 3)

;; Adding new element:
(vector-push-extend 4 *adjvec*) ; => 3

*adjvec* ; => #(1 2 3 4)



;;; Naively, sets are just lists:

(set-difference '(1 2 3 4) '(4 5 6 7)) ; => (3 2 1)
(intersection '(1 2 3 4) '(4 5 6 7)) ; => 4
(union '(1 2 3 4) '(4 5 6 7))        ; => (3 2 1 4 5 6 7)
(adjoin 4 '(1 2 3 4))     ; => (1 2 3 4)

;; But you'll want to use a better data structure than a linked list
;; for performant work!
```

```lisp
;;; Dictionaries are implemented as hash tables.

;; Create a hash table
(defparameter *m* (make-hash-table))

;; set a value
(setf (gethash 'a *m*) 1)

;; Retrieve a value
(gethash 'a *m*) ; => 1, t

;; Detail - Common Lisp has multiple return values possible. gethash
;; returns t in the second value if anything was found, and nil if
;; not.

;; Retrieving a non-present value returns nil
 (gethash 'd *m*) ;=> nil, nil

;; You can provide a default value for missing keys
(gethash 'd *m* :not-found) ; => :NOT-FOUND

;; Let's handle the multiple return values here in code.

(multiple-value-bind
      (a b)
    (gethash 'd *m*)
  (list a b))
; => (NIL NIL)

(multiple-value-bind
      (a b)
    (gethash 'a *m*)
  (list a b))
; => (1 T)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `lambda' to create anonymous functions.
;; A function always returns the value of its last expression.
;; The exact printable representation of a function will vary...

(lambda () "Hello World") ; => #<FUNCTION (LAMBDA ()) {1004E7818B}>

;; Use funcall to call lambda functions
(funcall (lambda () "Hello World")) ; => "Hello World"

;; Or Apply
(apply (lambda () "Hello World") nil) ; => "Hello World"

;; De-anonymize the function
(defun hello-world ()
   "Hello World")
(hello-world) ; => "Hello World"

;; The () in the above is the list of arguments for the function
(defun hello (name)
   (format nil "Hello, ~a" name))

(hello "Steve") ; => "Hello, Steve"

;; Functions can have optional arguments; they default to nil

(defun hello (name &optional from)
    (if from
        (format t "Hello, ~a, from ~a" name from)
        (format t "Hello, ~a" name)))

 (hello "Jim" "Alpacas") ;; => Hello, Jim, from Alpacas

;; And the defaults can be set...
(defun hello (name &optional (from "The world"))
   (format t "Hello, ~a, from ~a" name from))

(hello "Steve")
; => Hello, Steve, from The world

(hello "Steve" "the alpacas")
; => Hello, Steve, from the alpacas


;; And of course, keywords are allowed as well... usually more
;;   flexible than &optional.

(defun generalized-greeter (name &key (from "the world") (honorific "Mx"))
    (format t "Hello, ~a ~a, from ~a" honorific name from))

(generalized-greeter "Jim")   ; => Hello, Mx Jim, from the world

(generalized-greeter "Jim" :from "the alpacas you met last summer" :honorific "Mr")
; => Hello, Mr Jim, from the alpacas you met last summer

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Common Lisp has a sophisticated equality system. A couple are covered here.

;; for numbers use `='
(= 3 3.0) ; => t
(= 2 1) ; => nil

;; for object identity (approximately) use `eql`
(eql 3 3) ; => t
(eql 3 3.0) ; => nil
(eql (list 3) (list 3)) ; => nil
```

```lisp
;; for lists, strings, and bit-vectors use `equal'
(equal (list 'a 'b) (list 'a 'b)) ; => t
(equal (list 'a 'b) (list 'b 'a)) ; => nil

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if t                 ; test expression
    "this is true"    ; then expression
    "this is false") ; else expression
; => "this is true"

;; In conditionals, all non-nil values are treated as true
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(GROUCHO ZEPPO)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)
; => 'YEP

;; `cond' chains a series of tests to select a result
(cond ((> 2 2) (error "wrong!"))
      ((< 2 2) (error "wrong again!"))
      (t 'ok)) ; => 'OK

;; Typecase switches on the type of the value
(typecase 1
  (string :string)
  (integer :int))

; => :int

;;; Iteration

;; Of course recursion is supported:

(defun walker (n)
  (if (zerop n)
      :walked
      (walker (- n 1))))

(walker 5) ; => :walked

;; Most of the time, we use DOLIST or LOOP


(dolist (i '(1 2 3 4))
  (format t "~a" i))

; => 1234
```

```
(loop for i from 0 below 10
      collect i)

; => (0 1 2 3 4 5 6 7 8 9)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `setf' to assign a new value to an existing variable. This was
;; demonstrated earlier in the hash table example.

(let ((variable 10))
    (setf variable 2))
 ; => 2


;; Good Lisp style is to minimize destructive functions and to avoid
;; mutation when reasonable.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Classes and Objects
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; No more Animal classes, let's have Human-Powered Mechanical
;; Conveyances.

(defclass human-powered-conveyance ()
  ((velocity
    :accessor velocity
    :initarg :velocity)
   (average-efficiency
    :accessor average-efficiency
   :initarg :average-efficiency))
  (:documentation "A human powered conveyance"))

;; defclass, followed by name, followed by the superclass list,
;; followed by slot list, followed by optional qualities such as
;; :documentation.

;; When no superclass list is set, the empty list defaults to the
;; standard-object class. This *can* be changed, but not until you
;; know what you're doing. Look up the Art of the Metaobject Protocol
;; for more information.

(defclass bicycle (human-powered-conveyance)
  ((wheel-size
    :accessor wheel-size
    :initarg :wheel-size
    :documentation "Diameter of the wheel.")
   (height
    :accessor height
    :initarg :height)))
```

```lisp
(defclass recumbent (bicycle)
  ((chain-type
    :accessor chain-type
    :initarg  :chain-type)))

(defclass unicycle (human-powered-conveyance) nil)

(defclass canoe (human-powered-conveyance)
  ((number-of-rowers
    :accessor number-of-rowers
    :initarg :number-of-rowers)))


;; Calling DESCRIBE on the human-powered-conveyance class in the REPL gives:

(describe 'human-powered-conveyance)

; COMMON-LISP-USER::HUMAN-POWERED-CONVEYANCE
;  [symbol]
;
; HUMAN-POWERED-CONVEYANCE names the standard-class #<STANDARD-CLASS
;                                                    HUMAN-POWERED-CONVEYANCE>:
;  Documentation:
;    A human powered conveyance
;  Direct superclasses: STANDARD-OBJECT
;  Direct subclasses: UNICYCLE, BICYCLE, CANOE
;  Not yet finalized.
;  Direct slots:
;    VELOCITY
;      Readers: VELOCITY
;      Writers: (SETF VELOCITY)
;    AVERAGE-EFFICIENCY
;      Readers: AVERAGE-EFFICIENCY
;      Writers: (SETF AVERAGE-EFFICIENCY)

;; Note the reflective behavior available to you! Common Lisp is
;; designed to be an interactive system

;; To define a method, let's find out what our circumference of the
;; bike wheel turns out to be using the equation: C = d * pi

(defmethod circumference ((object bicycle))
  (* pi (wheel-size object)))

;; pi is defined in Lisp already for us!

;; Let's suppose we find out that the efficiency value of the number
;; of rowers in a canoe is roughly logarithmic. This should probably be set
;; in the constructor/initializer.

;; Here's how to initialize your instance after Common Lisp gets done
;; constructing it:
```

```
(defmethod initialize-instance :after ((object canoe) &rest args)
  (setf (average-efficiency object)  (log (1+ (number-of-rowers object)))))

;; Then to construct an instance and check the average efficiency...

(average-efficiency (make-instance 'canoe :number-of-rowers 15))
; => 2.7725887




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 8. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language

;; Common Lisp doesn't come with a WHILE loop- let's add one.
;; If we obey our assembler instincts, we wind up with:

(defmacro while (condition &body body)
    "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
    (let ((block-name (gensym)) (done (gensym)))
        `(tagbody
           ,block-name
           (unless ,condition
               (go ,done))
           (progn
           ,@body)
           (go ,block-name)
           ,done)))

;; Let's look at the high-level version of this:


(defmacro while (condition &body body)
    "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
  `(loop while ,condition
         do
         (progn
            ,@body)))

;; However, with a modern compiler, this is not required; the LOOP
;; form compiles equally well and is easier to read.

;; Note that ``` is used, as well as `,` and `@`. ``` is a quote-type operator
;; known as quasiquote; it allows the use of `,` . `,` allows "unquoting"
;; variables. @ interpolates lists.

;; Gensym creates a unique symbol guaranteed to not exist elsewhere in
```

```
;; the system. This is because macros are expanded at compile time and
;; variables declared in the macro can collide with variables used in
;; regular code.

;; See Practical Common Lisp for more information on macros.
```

Further Reading

- Keep moving on to the Practical Common Lisp book.
- A Gentle Introduction to...

Extra Info

- CLiki
- common-lisp.net
- Awesome Common Lisp

Credits.

Lots of thanks to the Scheme people for rolling up a great starting point which could be easily moved to Common Lisp.

- Paul Khuong for some great reviewing.