```
;;;; CLISP (Common Lisp) - quick program guide
;;;; Lisp stands for List Processing
;;;; This file contains sample code for Lisp
;;;; Lisp is not case sensitive – Hello, HELLO, and hELLo are the same


;;; ---------- INTRO ----------

;;; Comment
;; Comment that is indented with code
; Comment after a line of code

#||
Multiline Comment
||#

;;;; 1.  Hello World - begin
;;; ~% prints a newline with format
(format t "Hello world~%")
;;;; 1.  Hello World -end


;;; The format statement starts with t to print to the console
;;; The control sequence begins with a ~
;;; ~a : Shows the value
;;; (format t "Hello world ~a" 15)
;;; (format t "Hello world ~b" 15)   ;binary
;;; (format t "Hello world ~x" 15)   ;hex



;;;; 2.  A Simple Function - begin
;;; Print out a string without a newline – (print is another way to print output.)
(print "What's your name ")

;;; Create a variable which receives the value passed by read
;;; A variable name or symbol is made of letters, numbers, and + - _ * = < > ? !
;;; and are lowercase because Lisp isn't case sensitive
;;; You can't use white space in names because list items are separated
;;; with white space
;;; Asterisks surround global variable names
(defvar *name* (read))      ; read from console

;;; Create a function and say hello to value passed
;;; Your supposed to keep closing parentheses on the same line, but that
;;; is up to you if the code is easier to follow
(defun hello-you (*name*)
            (format t "Hello ~a!~%" *name*)
)

;;; Change the case to capitalize just the first letter (:upcase :downcase)
(setq *print-case* :capitalize)   ; setting a quoted value.

(hello-you *name*)  ; calling the function

;;;; 2.  A Simple Function - end

;;;; 2a.  Example - begin
(print "What's your name ")
(defvar *name* (read))      ; read from console

(defun hello-you (*name*)
            (format t "Hello ~a!~%" *name*)
)
(setq *print-case* :capitalize)   ; setting a quoted value.

(hello-you *name*)  ; calling the function
;;;; 2a.  Example - end
```

;;; 3. ***A form is a list with a command function name at the beginning***

```
;;; Everything that follows the command is sent as parameters to the function
(+ 5 4) ; = 9

;;; You can nest a form inside of a form
(+ 5 (- 6 2)) ; = 9

;;; You define a Data Mode command by proceeding with a quote '
'(+ 5 4)

;;; Everything is a list in which each piece is held in a Cons Cell (Consecutive
;;; Cell)
;;; [+] [5] [4] [nil] with nil defining the end of the list

;;; Change the value of a variable with setf
(setf *number* 6)


;;; ---------- FORMAT ---------- similar to the printf format string

(format t "PI to 5 characters ~5f ~%" 3.141593)

(format t "PI to 4 decimals ~,4f ~%" 3.141593)

(format t "10 Percent ~,,2f ~%" .10)
```

```
(format t "10 Dollars ~$ ~%" 10)

;;; ---------- MATH FUNCTIONS ----------

(format t "(+ 5 4) = ~d ~%" (+ 5 4))
(format t "(- 5 4) = ~d ~%" (- 5 4))
(format t "(* 5 4) = ~d ~%" (* 5 4))
(format t "(/ 5 4) = ~d ~%" (/ 5 4)) ; = 5/4
(format t "(/ 5 4.0) = ~d ~%" (/ 5 4.0)) ; = 1.25
(format t "(rem 5 4) = ~d ~%" (rem 5 4)) ; = 1 Returns the remainder
(format t "(mod 5 4) = ~d ~%" (mod 5 4)) ; = 1 Returns the remainder

(format t "(expt 4 2) = ~d ~%" (expt 4 2)) ; = Exponent 4^2
(format t "(sqrt 81) = ~d ~%" (sqrt 81)) ; = 9
(format t "(exp 1) = ~d ~%" (exp 1)) ; = e^1
(format t "(log 1000 10) = ~d ~%" (log 1000 10)) ; = 3 = Because 10^3 = 1000
(format t "(eq 'dog 'dog) = ~d ~%" (eq 'dog 'dog)) ; = T Check Equality
(format t "(floor 5.5) = ~d ~%" (floor 5.5)) ; = 5
(format t "(ceiling 5.5) = ~d ~%" (ceiling 5.5)) ; = 6
(format t "(max 5 10) = ~d ~%" (max 5 10)) ; = 10
(format t "(min 5 10) = ~d ~%" (min 5 10)) ; = 5

;;; ---------- EQUALITY ----------

;;; Symbols are compared with eq

(defparameter *name* 'Derek)
(format t "(eq *name* 'Derek) = ~d ~%" (eq *name* 'Derek))

;;; Everything else is compared with equal for the most part

(format t "(equal 'car 'truck) = ~d ~%" (equal 'car 'truck))
(format t "(equal 10 10) = ~d ~%" (equal 10 10))
(format t "(equal 5.5 5.3) = ~d ~%" (equal 5.5 5.3))
(format t "(equal \"string\" \"String\") = ~d ~%" (equal "string" "String"))
(format t "(equal (list 1 2 3) (list 1 2 3)) = ~d ~%"
          (equal (list 1 2 3) (list 1 2 3)))

;;; equalp can compare strings of any case and integers to floats
(format t "(equalp 1.0 1) = ~d ~%" (equalp 1.0 1))
(format t "(equalp \"Derek\" \"derek\") = ~d ~%" (equalp "Derek" "derek"))



;;; 4  CONDITIONALS - begin
;;; ---------- CONDITIONALS ----------

(defparameter *age* 18) ; Create variable age
```

```
;;; Relational Operators > < >= <= =

;;; Check if age is greater than or equal to 18

(if (= *age* 18)
(format t "You can vote~%")
(format t "You can't vote~%"))

;;; 4  CONDITIONALS - end

;;; How to check for not equal

(if (not (= *age* 18))
(format t "You can vote~%")
(format t "You can't vote~%"))

;;; Logical Operators : and, or, not

(if (and (>= *age* 18) (<= *age* 67) )
(format t "Time for work~%")
(format t "Work if you want~%"))

(if (or (<= *age* 14) (>= *age* 67) )
(format t "You shouldn't work~%")
(format t "You should work~%"))

(defparameter *num* 2)
(defparameter *num-2* 2)
(defparameter *num-3* 2)


;;; 5  Case - begin
;;; Case performs certain actions depending on conditions
(defun get-school (age)
        (case age
                (5 (print "Kindergarten"))
                (6 (print "First Grade"))
                (otherwise (print "middle school"))
        ))

(get-school 7)

(terpri) ; Newline
;;; 5  Case - end

;;; when allows you to execute multiple statements by default
```

```
(defparameter *age* 18) ; Create variable age

(when (= *age* 18)
        (setf *num-3* 18)
        (format t "Go to college you're ~d ~%" *num-3*)
)

;;; With unless code is executed if the expression is false

(unless (not (= *age* 18))
        (setf *num-3* 20)
        (format t "Something Random ~%")
)




;;; ---------- LOOPING ----------

;;; loop executes code a defined number of times
;;; Create a list using numbers 1 through 10
(loop for x from 1 to 10
        do(print x))

;;; Loop until the when condition calls return
(setq x 1)
(loop
        (format t "~d ~%" x)
        (setq x (+ x 1))
        (when (> x 10) (return x))
)

;;; 6  Loop for – through a list – begin

;;; loop for can cycle through a list or iterate commonly
;;; It will execute any number of statements after do
(loop for x in '(Peter Paul Mary) do
        (format t "~s ~%" x)
)
;;; 6  Loop for – through a list – end

(loop for y from 10 to 14 do
        (print y)
)

;;; dotimes iterates a specified number of times
(dotimes (y 12)
        (print y))
```

```lisp
;;; ---------- CONS CELLS / LISTS ----------

;;; Link together 2 objects of data
(cons 'superman 'batman)

;;; Create a list with list
(list 'superman 'batman 'flash)

;;; Add item to the front of another list
(cons 'aquaman '(superman batman))

;;;; CAR and CDR for homework 1
;;;; CAR and CDR stand for "Contents of the Address Register" and "Contents of the Decrement Register".
;;;; the CAR of a list is the first item in the list. Thus the CAR of the list (rose violet daisy buttercup) is rose.

;;;;The CDR of a list is the rest of the list, that is, the cdr function returns the part of the list that follows
;;;;the first item. Thus, while the CAR of the list '(rose violet daisy buttercup) is rose, the rest of the list,
;;;;the value returned by the cdr function, is (violet daisy buttercup).

;;;;

;;; Get the first item out of a list with car
(format t "First = ~a ~%" (car '(superman batman aquaman)))


;;; Get everything but the first item with cdr
(format t "Everything Else = ~a ~%" (cdr '(superman batman aquaman)))

;;; Get the 2nd item d = (batman flash joker) a = (batman)
(format t "2nd Item = ~a ~%" (cadr '(superman batman aquaman flash joker)))

;;; Get the 3rd item = aquaman
(format t "3rd Item = ~a ~%" (caddr '(superman batman aquaman flash joker)))

;;;   cadddr – traverse inside a list
;;; Get the 4th item = flash
(format t "4th Item = ~a ~%" (cadddr '(superman batman aquaman flash joker)))

;;; Get the 4th item = joker
(format t "4th Item = ~a ~%" (cdddddr '(superman batman aquaman flash joker wonder_wowan)))

;;; Get the 2nd item in the second list
;;; d : (aquaman flash joker) (wonderwoman catwoman)
;;; a : (aquaman flash joker)
;;; d : (flash joker)
;;; a : (flash)
```

```
(format t "2nd Item 2nd List = ~a ~%"
(cadadr '((superman batman) (aquaman flash joker) (wonderwoman catwoman))))

;;; Get the 3rd item in the 2nd list = joker
(format t "3rd Item 2nd List = ~a ~%"
(cddadr '((superman batman) (aquaman flash joker) (wonderwoman catwoman))))

;;; = T Is something a list
(format t "Is it a List = ~a ~%" (listp '(batman superman)))

;;; Is 3 a member of the list
(format t "Is 3 in the List = ~a ~%" (if (member 3 '(2 4 6)) 't nil))

;;; Combine lists into 1 list
(append '(just) '(some) '(random words))

;;; defparameter and defvar establish name as a dynamic variable. defparameter unconditionally assigns
;;; the initial-value to the dynamic variable named name. defvar, by contrast, assigns initial-value (if
;;; supplied) to the dynamic variable named name only if name is not already bound.

;;; Push an item on the front of a list
(defparameter *nums* '(2 4 6))
(push 1 *nums*)

;;; Get the nth value from a list
(format t "2nd Item in the List = ~a ~%" (nth 2 *nums*))

;;; Create a plist which uses a symbol to describe the data
(defvar superman (list :name "Superman" :secret-id "Clark Kent"))

;;; This list will hold heroes
(defvar *hero-list* nil)

;;; Adds items to our list
(push superman *hero-list*)

;;; Cycle through all heros in the list and print them out
(dolist (hero *hero-list*)

        ;; Surround with ~{ and ~} to automatically grab data from list
        (format t "~{~a : ~a ~}~%" hero)
)

;;; ---------- ASSOCIATION LIST ----------

;;; The hero name represents the key

(defparameter *heroes*
	'((Superman (Clark Kent))
	(Flash (Barry Allen))
	(Batman (Bruce Wayne))))

;;; Get the key value with assoc
(format t "Superman Data ~a ~%" (assoc 'superman *heroes*))

;;; Get secret identity
(format t "Superman is ~a ~%" (cadr (assoc 'superman *heroes*)))

(defparameter *hero-size*
	'((Superman (6 ft 3 in) (230 lbs))
	(Flash (6 ft 0 in) (190 lbs))
	(Batman (6 ft 2 in) (210 lbs))))

;;; Get height
(format t "Superman is ~a ~%" (cadr (assoc 'Flash *hero-size*)))

;;; Get weight
(format t "Batman is ~a ~%" (caddr (assoc 'Batman *hero-size*)))

;;; ---------- FUNCTIONS ----------

;;; Create a function that says hello
(defun hello ()
	(print "Hello")
	(terpri)) ; Newline

(hello)

;;; Get average
(defun get-avg (num-1 num-2)
	(/ (+ num-1 num-2) 2 ))

(format t "Avg 10 & 50 = ~a ~%" (get-avg 10 50))

;;; You can define some parameters as optional in a function with &optional
(defun print-list (w x &optional y z)
	(format t "List = ~a ~%" (list w x y z))
)

(print-list 1 2 3)

;;; Receive multiple values with &rest
(defvar *total* 0)

(defun sum (&rest nums)
	(dolist (num nums)
                (setf *total* (+ *total* num))
        )
        (format t "Sum: ~a ~%" *total*)
)

(sum 1 2 3 4 5)

;;; Keyword parameters are used to pass values to specific variables
(defun print-list(&optional &key x y z)
        (format t "List: ~a ~%" (list x y z))
)

(print-list :x 1 :y 2)

;;; Functions by default return the value of the last expression
;;; You can also return a specific value with return-from followed by the
;;; function name
(defun difference (num1 num2)
        (return-from difference(- num1 num2))
)

(format t "10 - 2 = ~a ~%" (difference 10 2))

;;; Get Supermans data
;;; When you use ` you are using quasiquoting which allows you to switch from
;;; code to data mode
;;; The function between ,() is code mode

(defun get-hero-data (size)
        (format t "~a ~%"
        `(,(caar size) is ,(cadar size) and ,(cddar size))))

(defparameter *hero-size*
        '((Superman (6 ft 3 in) (230 lbs))
        (Flash (6 ft 0 in) (190 lbs))
        (Batman (6 ft 2 in) (210 lbs))))

(get-hero-data *hero-size*)

;;; Check if every item in a list is a number
(format t "A number ~a ~%" (mapcar #'numberp '(1 2 3 f g)))

;;; You can define functions local only to the flet body
;;; (flet ((func-name (arguments)
;;;         ... Function Body ...))
;;;         ... Body ...)

(flet ((double-it (num)
        (* num 2)))
        (double-it 10))

;;; You can have multiple functions in flet
(flet ((double-it (num)
        (* num 2))
        (triple-it (num)
        (* num 3)))
        (format t "Double & Triple 10 = ~d~%" (triple-it (double-it 10)))
)
;;; 8  Labels – calling a function itself – begin

;;; labels is used when you want to have a function call itself, or if you want
;;; to be able to call another local function inside a function
(labels ((double-it (num)
            (* num 2))
        (triple-it (num)
            (* (double-it num) 3)))
        (format t "Double & Triple 3 = ~d~%" (triple-it 3))
)
;;; 8  Labels – calling a function itself – end

;;; Return multiple values from a function
(defun squares (num)
    (values (expt num 2) (expt num 3)))

;;; Get multiple values from a function
(multiple-value-bind (a b) (squares 2)
    (format t "2^2 = ~d  2^3 = ~d~%" a b)
)
```

```
;;; 9  Higher Order Function – begin

;;; Higher Order Functions
;;; You can use functions as data

(defun times-3 (x) (* 3 x))
(defun times-4 (x) (* 4 x))

;;; Pass in the function without attributes just like a variable
(defun multiples (mult-func max-num)

        ;; Cycle through values up to the max supplied
        ;; dotimes is a macro for integer iteration over a single variable from 0 below some parameter
        ;; value. One of the simples examples would be: CL-USER> (dotimes (i 5) (print i)) 0 1 2 3 4 NIL.
        (dotimes (x max-num)

                ;; funcall is used when you know the number of arguments
                (format t "~d : ~d~%" x (funcall mult-func x))
))

(multiples #'times-3 10)
(multiples #'times-4 10)

;;; 9  Higher Order Function – end
```

;;; ---------- LAMBDA ---------- **(λ)(Head).(body)**
;;; ---------- LAMBDA ---------- **(λ a.a) 3 becomes 3**
;;; ---------- LAMBDA ---------- **(λ a.z) b becomes bz**
;;; ---------- LAMBDA ---------- **(λ a.ab) (λ x.xx) becomes (λ (λ x.xx). (λ x.xx) b)**
;;; ---------- LAMBDA ---------- **(λ (λ x.xx). (λ x.xx) b) becomes (λ x.xx) b**
;;; ---------- LAMBDA ---------- **(λ x.xx) b becomes (λ b.bb) becomes bb**

```
;;; The lambda command allows you to create a function without giving it a name
;;; You can also pass this function just like you pass variables


;lambda expression to get sum of product of four numbers
;mathematical expression is (val1*val2) + (val3*val4)
 (write ((lambda (val1 val2 val3 val4)
  (+  (* val1 val2) (+ (* val3 val4))))
  ;pass the values
  2 4 6 8)
)
```

This will print 56.

```
;another example
(write ((lambda (a b c x)
  (+ (* a (* x x)) (* b x) c))
  4 2 9 3)
)
```

This will print 51.

```
;;; mapcar is a function that calls its first argument with each element of its second argument, in turn.
;;; The second argument must be a sequence.

;;; The ‘map’ part of the name comes from the mathematical phrase, “mapping over a domain”,
;;; meaning ;;; to apply a function to each of the elements in a domain. The mathematical phrase is ;;;
;;; based on the metaphor of a surveyor walking, one step at a time, over an area he is mapping. And
;;; ‘car’, of course, comes from the Lisp notion of the first of a list.

;;; Multiply every item in a list
(mapcar (lambda (x) (* x 2)) ‘(1 2 3 4 5))

;;; Multiply every item in a list
(mapcar (lambda (x) (print (* x 2))) ‘(1 2 3 4 5))

;;; Multiply every item in a list
(mapcar (lambda (x) (print (* x 2))) ‘(5))



;;; ---------- MACROS ----------
;;; A function runs when it is called to execute, while a macro is compiled
;;; first and is available immediately like any other lisp built in function
;;; Macros are functions used to generate code rather then perform actions

(defvar *num* 2)
(defvar *num-2* 0)

;;; It can be irritating to have to use progn with if
(if (= *num* 2)
        (progn
                (setf *num-2* 2)
                (format t "*num-2* = ~d ~%" *num-2*)
        )
        (format t "Not equal to 2 ~%"))
```

```
(defmacro ifit (condition &rest body)

        ;;; The backquote generates the code
        ;;; The , changes the condition to code mode from data mode
        ;;; The &rest body parameter will hold commands in a list
        ;;; The "Can't Drive" Works as the else

        `(if ,condition (progn ,@body) (format t "Can't Drive ~%") ))

(setf *age* 16)

(ifit (>= *age* 16)
        (print "You are over 16")
        (print "Time to Drive")
        (terpri)
)

;;; let can also get confusing with its parentheses

(defun add (num1 num2)
        (let ((sum (+ num1 num2)))
                (format t "~a + ~a = ~a ~%" num1 num2 sum)))

;;; Define a macro to clean up let

(defmacro letx (var val &body body)
        `(let ((,var ,val)) ,@body))

(defun subtract (num1 num2)
        (letx dif (- num1 num2)
                (format t "~a - ~a = ~a ~%" num1 num2 dif)))

(subtract 10 6)

;;; ---------- FILE I O ----------

;;; Write text to a file
;;; A keyword symbol starts with a colon and it only means itself
(with-open-file (my-stream
                                "test.txt"
                                :direction :output ; We are writing to the file
                                :if-exists :supersede) ; If the file exists delete it
        (princ "Some random Text" my-stream))

;;; Read data from a file
(let ((in (open "test.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
    while line do (format t "~a~%" line))
    (close in)
  )
)
```