# Programming-Idioms.org



## Lisp

**1 Print Hello World**
Print a literal string on standard output

```
(princ "Hello, world!")
```

***Alternative implementation:***

```
(print "Hello World")
```

sbcl common lisp dialect

***Alternative implementation:***

```
(format T "Hello World")
```

**2 Print Hello 10 times**
Loop to execute some code a constant number of times

```
(loop repeat 10 do (write-line "Hello"))
```

**3 Create a procedure**
Like a function which doesn't return any value, thus has only side effects (e.g. Print to standard output)

```
(defun show (message)
  (format t "Hello: ~a" message)
  (values))
```

**4 Create a function**
Create a function which returns the square of an integer

```
(defun square (x)
  (* x x))
```

**5 Create a 2D Point data structure**
Declare a container type for two floating-point numbers ***x*** and ***y***

```
(defstruct point
  (x 0.0d0 :type double-float)
  (y 0.0d0 :type double-float))
```

**6 Iterate over list values**
Do something with each item ***x*** of the list (or array) ***items***, regardless indexes.

```
(map nil #'something items)
```

**7 Iterate over list indexes and values**
Print each index ***i*** with its value ***x*** from an array-like collection ***items***

```
(loop for i from 0 and x across items
      do (format t "~a = ~a~%" i x))
```

**Lisp**

**8 Create a map (associative array)**

Create a new map object ***x***, and provide some (key, value) pairs as initial content.

```
(let ((table (make-hash-table)))
  (setf (gethash 'one table) 1)
  (setf (gethash 'two table) 2))
```

**9 Create a Binary Tree data structure**

The structure must be recursive because ***left*** child and ***right*** child are binary trees too. A node has access to children nodes, but not to its parent.

```
(defclass node ()
  ((value
      :initarg :value :initform nil
      :accessor node-value)
   (left-child
      :initarg :left-child :initform nil
      :accessor node-left-child)
   (right-child
      :initarg :right-child :initform nil
      :accessor node-right-child)))
```

**10 Shuffle a list**

Generate a random permutation of the elements of list ***x***

```
(let ((end (length x)))
 (loop for i from 0 below end
       do (rotatef (aref x i)
                   (aref x (+ i (random (- end i)))))))
```

This sorts a lisp vector.

**11 Pick a random element from a list**

The list ***x*** must be non-empty.

```
(nth (random (length x)) x)
```

**12 Check if list contains a value**

Check if the ***list*** contains the value ***x***.
***list*** is an iterable finite container.

```
(member x list)
```

***Alternative implementation:***

```
(find x list :test #'equal)
```

Returns the value if it is in the list, otherwise returns ***nil***.

**13 Iterate over map keys and values**

Access each key ***k*** with its value ***x*** from an associative array ***mymap***, and print them.

```
(loop for k being each hash-key of mymap
      for x being each hash-value of mymap
      do (format t "~A ~A~%" k x))
```

**Lisp**

**14 Pick uniformly a random floating point number in [a..b)**

Pick a random number greater than or equals to ***a***, strictly inferior to ***b***. Precondition : ***a*** < ***b***.

```
(+ a (random (float (- b a))))
```

**15 Pick uniformly a random integer in [a..b]**

Pick a random integer greater than or equals to ***a***, inferior or equals to ***b***. Precondition : ***a*** < ***b***.

```
(defun r (a  b)
(+ a (random (+ 1 (- b a )))))
```

**19 Reverse a list**

Reverse the order of the elements of the list ***x***.
This may reverse "in-place" and destroy the original ordering.

```
(reverse x)
```

This preserves the value of ***x***.

**20 Return two values**

Implement a function ***search*** which looks for item ***x*** in a 2D matrix ***m***.
Return indices ***i***, ***j*** of the matching cell.
Think of the most idiomatic way in the language to return the two values at the same time.

```
(defun mysearch (x m)
  (loop for i below (array-dimension m 0)
        do (loop for j below (array-dimension m 1)
                 when (eql x (aref m i j))
                 do (return-from my-search
                                 (values i j)))))
```

**21 Swap values**

Swap the values of the variables ***a*** and ***b***

```
(rotatef a b)
```

**22 Convert string to integer**

Extract the integer value ***i*** from its string representation ***s*** (in radix 10)

```
(setf i (parse-integer s))
```

**24 Assign to string the japanese word ネコ**

Declare a new string ***s*** and initialize it with the literal value "ネコ" (which means "cat" in japanese)

```
(defvar *s* "ネコ")
```

**26 Create a 2-dimensional array**

Declare and initialize a matrix ***x*** having ***m*** rows and ***n*** columns, containing real numbers.

```
(make-array (list m n)
   :element-type 'double-float
   :initial-element 1.0d0)
```

**Lisp**

**27 Create a 3-dimensional array**

Declare and initialize a 3D array ***x***, having dimensions boundaries ***m***, ***n***, ***p***, and containing real numbers.

```
(defparameter *x*
  (make-array (list m n p)
              :element-type 'double-float
              :initial-element 0.0d0))
```

**28 Sort by a property**

Sort the elements of the list (or array-like collection) ***items*** in ascending order of ***x***.***p***, where ***p*** is a field of the type ***Item*** of the objects in ***items***.

```
(sort #'< items :key #'p)
```

**31 Recursive factorial (simple)**

Create the recursive function ***f*** which returns the factorial of the non-negative integer ***i***, calculated from ***f***(***i***-1)

```
(defun f (i)
  (if (< i 2)
    1
    (* i (f (- i 1)))))
```

**32 Integer exponentiation by squaring**

Create function ***exp*** which calculates (fast) the value ***x*** power ***n***.

***x*** and ***n*** are non-negative integers.

```
(loop
  for (power rem) = (list p 0) then (multiple-value-list
(floor power 2))
  for multiple = 1 then (if (zerop rem) multiple (* multip
le base))
  for base = x then (* base base)
  when (= power 0) return 1
  when (= power 1) return (* multiple base)))
```

All of the odd values are multiplied into multiple while base is squared with each iteration.

Since specification is "non-negative" handle the case of the power being 0 separately

**36 First-class function : generic composition**

Implement a function ***compose*** which returns composition function ***g***∘***f*** for any functions ***f*** and ***g*** having exactly 1 parameter.

```
(defun compose (f g)
  (lambda (x) (g (f x))))
```

**Lisp**

**37 Currying**

Transform a function that takes multiple arguments into a function for which some of the arguments are preset.

```
(defun curry (fn &rest args)
  (lambda (&rest remaining-args)
    (apply fn (append args remaining-args))))

(defun add (a b)
  (+ a b))

(funcall (curry add 2) 1)
```

**38 Extract a substring**

Find substring ***t*** consisting in characters ***i*** (included) to ***j*** (excluded) of string ***s***.

Character indices start at ***0*** unless specified otherwise.

Make sure that multibyte characters are properly handled.

```
(setf u (subseq s i j))
```

***t*** is not a good choice for a variable name in Lisp, so using ***u*** instead for the result string

**39 Check if string contains a word**

Set the boolean ***ok*** to ***true*** if the string ***word*** is contained in string ***s*** as a substring, or to ***false*** otherwise.

```
(setf ok (search word s))
```

Technically this sets ok to the position of the string. This is more idiomatic than setting it specifically to true, since all numbers are true in lisp. Can use (not (null ...)) if T is desired.

**41 Reverse a string**

Create the string ***t*** containing the same characters as the string ***s***, in reverse order.

The original string ***s*** must remain unaltered. Each character must be handled correctly regardless its number of bytes in memory.

```
(reverse s)
```

**43 Break outer loop**

Look for a negative value ***v*** in 2D integer matrix ***m***. Print it and stop searching.

```
(loop named outer
      for i below (array-dimension m 0)
      do (loop for j below (array-dimension m 1)
               for v = (aref m i j)
               when (minusp v)
               do (progn
                    (print v)
                    (return-from outer))))
```

**Lisp**

**44 Insert element in list**

Insert the element ***x*** at position ***i*** in the list ***s***. Further elements must be shifted to the right.

```
(defun ins (lst x i)
 (if (zerop i) (cons x lst)
     (cons (car lst) (ins (cdr lst) x (- i 1)))))

(setf s (ins s x i))
```

**45 Pause execution for 5 seconds**

Sleep for 5 seconds in current thread, before proceeding with the next instructions.

```
(sleep 5)
```

**46 Extract beginning of string (prefix)**

Create the string ***t*** consisting of the 5 first characters of the string ***s***.
Make sure that multibyte characters are properly handled.

```
(setf *t* (subseq s 0 5))
```

**47 Extract string suffix**

Create string ***t*** consisting in the 5 last characters of string ***s***.
Make sure that multibyte characters are properly handled.

```
(setf *t* (subseq s (- (length s) 5)))
```

**48 Multi-line string literal**

Assign to variable ***s*** a string literal consisting in several lines of text, including newlines.

```
(setf s "a
b
c
d")
```

**49 Split a space-separated string**

Build list ***chunks*** consisting in substrings of the string ***s***, separated by one or more space characters.

```
(defun words (s)
  (if (equalp s "") nil
      (let ((p (position #\Space s )))
    (cond ((null p) (list s))
          ((zerop p ) (words (subseq s 1)))
          (T (cons (subseq s 0 p) (words (subseq s (+ 1 p
))))))))))

(setf chunks (words s))
```

**Lisp**

**50 Make an infinite loop**

Write a loop that has no end clause.

```
(loop)
```

***Alternative implementation:***

```
(defun infinite-loop (x)
  (apply x (infinite-loop x)))
```

Not exactly the same as the original since that was in scheme and this is Common Lisp but it infinitely loops until the control stack is exhausted.

**51 Check if map contains key**

Determine whether the map ***m*** contains an entry for the key ***k***

```
(nth-value 1 (gethash k m))
```

***gethash*** returns a value, and a boolean for whether or not the entry is present.

**53 Join a list of strings**

Concatenate elements of string list ***x*** joined by the separator ", " to create a single string ***y***.

```
(defvar y (format nil "~{~A~^, ~}" x))
```

***Alternative implementation:***

```
(setf y (format nil "~{~a~^,~}" x))
```

***Alternative implementation:***

```
(setf y (reduce (lambda (a b)
                  (concatenate 'string a ", " b))
                x))
```

**54 Compute sum of integers**

Calculate the sum ***s*** of the integer list or array ***x***.

```
(reduce #'+ x)
```

**55 Convert integer to string**

Create the string representation ***s*** (in radix 10) of the integer value ***i***.

```
(setf s (princ-to-string i))
```

**Lisp**

**56 Launch 1000 parallel tasks and wait for completion**

Fork-join : launch the concurrent execution of procedure ***f*** with parameter ***i*** from 1 to 1000.
Tasks are independent and ***f***(***i***) doesn't return any value.
Tasks need not run all at the same time, so you may use a pool.
Wait for the completion of the 1000 tasks and then print "Finished".

```
(lambda (fn)
  (pdotimes (i 1000)
    (funcall fn i)))
```

**57 Filter list**

Create the list ***y*** containing the items from the list ***x*** that satisfy the predicate ***p***. Respect the original ordering. Don't modify ***x*** in-place.

```
(setf y (remove-if-not p x))
```

**58 Extract file content to a string**

Create the string ***lines*** from the content of the file with filename ***f***.

```
(with-open-file (stream f)
  (uiop:slurp-stream-string stream))
```

UIOP is technically not builtin but it is practically builtin since all commonly used implementations ship with it.

***Alternative implementation:***

```
(defvar *lines*
   (with-open-file (stream f)
      (let ((contents (make-string (file-length stream))))
            (read-sequence contents stream)
      :return contents)))
```

native

**59 Write to standard error stream**

Print the message "***x*** is negative" to standard error (stderr), with integer ***x*** value substitution (e.g. "-2 is negative").

```
(format *error-output*
        "~a is negative"
        x)
```

**64 Big integer : value 3 power 247**

Assign to ***x*** the value 3^247

```
(setf x (expt 3 247))
```

**68 Create a bitset**

Create an object ***x*** to store ***n*** bits (***n*** being potentially large).

```
(let ((x (make-array n :element-type 'bit))))
```

**Lisp**

**69 Seed random generator**

Use seed ***s*** to initialize a random generator.

If ***s*** is constant, the generator output will be the same each time the program runs. If ***s*** is based on the current value of the system clock, the generator output will be different each time.

```
call random_seed (put=s)
```

**70 Use clock as random generator seed**

Get the current datetime and provide it as a seed to a random generator. The generator sequence will be different at each run.

```
(setf *random-state* (make-random-state t))
```

This doesn't necessarily use the datetime as there is no portable way to initialize the lisp random-state from an integer.

This is the idiomatic way of getting a random-state that differs with each run.

**71 Echo program implementation**

Basic implementation of the Echo program: Print all arguments except the program name, separated by space, followed by newline.
The idiom demonstrates how to skip the first argument if necessary, concatenate arguments as strings, append newline and print it to stdout.

```
(format t "~{~A~^ ~}~%" *args*)
```

**75 Compute LCM**

Compute the least common multiple ***x*** of big integers ***a*** and ***b***. Use an integer type able to handle huge numbers.

```
(setf x (lcm a b))
```

**77 Complex number**

Declare a complex ***x*** and initialize it with value (***3i*** - ***2***). Then multiply it by ***i***.

```
(let ((x #c(-2 3)))
  (* x #c(0 1)))
```

**78 "do while" loop**

Execute a block once, then execute it again as long as boolean condition ***c*** is true.

```
(loop do (something)
      while c)
```

**79 Convert integer to floating point number**

Declare the floating point number ***y*** and initialize it with the value of the integer ***x*** .

```
(defparameter *y* (float x))
```

**Lisp**

**80 Truncate floating point number to integer**

Declare integer ***y*** and initialize it with the value of floating point number ***x*** . Ignore non-integer digits of ***x*** .
Make sure to truncate towards zero: a negative ***x*** must yield the closest greater integer (not lesser).

```
(truncate x)
```

```
; You might also define it using floor function:
(defun tr (x)
(let ((fl (floor x)))
(if (and (< x 0) (> x fl)) (+ 1 fl) fl ))
)
```

**81 Round floating point number to integer**

Declare the integer ***y*** and initialize it with the rounded value of the floating point number ***x*** .
Ties (when the fractional part of ***x*** is exactly .***5***) must be rounded up (to positive infinity).

```
(defun rnd (y) (floor (+ y 0.5)))
```

**82 Count substring occurrences**

Find how many times string ***s*** contains substring ***t***.
Specify if overlapping occurrences are counted.

```
;;; s=str, t=pattern
(defun cnt_substr (str pattern)
  (loop for i from 0 to (- (length str) (length pattern) )
    sum (if (equal pattern (subseq str i (+ i (length patt
ern )))) 1  0 )))
```

**84 Count bits set in integer binary representation**

Count number ***c*** of 1s in the integer ***i*** in base 2.

E.g. ***i***=6 → ***c***=2

```
(setf c (logcount i))
```

**87 Stop program**

Exit immediately.
If some extra cleanup work is executed by the program runtime (not by the OS itself), describe it.

```
(exit)
```

**92 Save object into JSON file**

Write the contents of the object ***x*** into the file ***data.json***.

```
(with-open-file (out "data.json" :direction :output)
  (yason:encode-plist x out))
```

**93 Pass a runnable procedure as parameter**

Implement the procedure ***control*** which receives one parameter ***f***, and runs ***f***.

```
(defun control (f)
   (funcall f))
```

**Lisp**

**94 Print the type of a variable**

Print the name of the type of ***x***. Explain if it is a static type or dynamic type.

This may not make sense in all languages.

```
(describe x)
```

the result is different, depending on your specific implementation of lisp.

**95 Get file size**

Assign to variable ***x*** the length (number of bytes) of the local file at ***path***.

```
(defvar x (osicat-posix:stat-size (osicat-posix:stat #P"pat
h")))
```

**96 Check string prefix**

Set the boolean ***b*** to ***true*** if string ***s*** starts with prefix ***prefix***, ***false*** otherwise.

```
(defun check-prefix (s prefix)
 (cond ((> (length prefix) (length s)) Nil)
       ( T ( equal prefix (subseq s 0 (length prefix
))))))

(setf b (check-prefix s prefix))
```

**97 Check string suffix**

Set boolean ***b*** to ***true*** if string ***s*** ends with string ***suffix***, ***false*** otherwise.

```
(defun check-suffix (str suf )
  (if (> (length suf) (length str )) nil (equalp (subseq s
tr (- (length str) (length suf)) (length str )) suf )))

(setf b (check-suffix s suffix))
```

**100 Sort by a comparator**

Sort elements of array-like collection ***items***, using a comparator ***c***.

```
(sort items #'c)
```

;;; c is a predicate like (defun c ( u v ) ( > u v ))
;;; items is a list
;;; the sorted list is
;;; note that #'c denotes the function c.

**101 Load from HTTP GET request into a string**

Make an HTTP request with method GET to the URL ***u***, then store the body of the response in the string ***s***.

```
(defvar *s* (string (dex:get u)))
```

**Lisp**

**102 Load from HTTP GET request into a file**

Make an HTTP request with method GET to the URL ***u***, then store the body of the response in the file ***result.txt***. Try to save the data as it arrives if possible, without having all its content in memory at once.

```
(serapeum:write-stream-into-file
   (dex:get u :want-stream t)
              #P"result.txt"
              :direction :output
              :if-exists :supersede
              :if-does-not-exist :create)
```

**103 Load XML file into struct**

Read from the file ***data.xml*** and write its contents into the object ***x***.
Assume the XML data is suitable for the type of ***x***.

```
(let ((x (plump:parse #p"data.xml")))
  ...)
```

**110 Check if string is blank**

Set the boolean ***blank*** to ***true*** if the string ***s*** is empty, or null, or contains only whitespace ; ***false*** otherwise.

```
(setf blank (not (find #\space s :test-not #'eql)))
```

**111 Launch other program**

From current process, run program ***x*** with command-line parameters "a", "b".

```
(sb-ext:run-program '("a" "b") "/path/to/x")
```

SBCL only

**117 Get list size**

Set ***n*** to the number of elements of the list ***x***.

```
(setf n (length x))
```

**118 List to set**

Create the set ***y*** from the list ***x***.
***x*** may contain duplicates. ***y*** is unordered and has no repeated values.

```
(setf y (remove-duplicates x))
```

**119 Deduplicate list**

Remove duplicates from the list ***x***.
Explain if the original order is preserved.

```
(remove-duplicates x)
```

**120 Read integer from stdin**

Read an integer value from the standard input into the variable ***n***

```
(setf n (read-from-string (remove-if-not #'digit-char-p (re
ad-line))))
```

(setf n (read))
if you don't care if it's explicitly an integer

**Lisp**

**123 Assert condition**

Verify that predicate ***isConsistent*** returns true, otherwise report assertion violation.
Explain if the assertion is executed even in production environment or not.

```
(assert (isConsistent))
```

Assertion is always executed.

**125 Measure function call duration**

measure the duration ***t***, in nanoseconds, of a call to the function ***foo***. Print this duration.

```
(time (foo))
```

**126 Multiple return values**

Write a function ***foo*** that returns a ***string*** and a ***boolean*** value.

```
(defun foo () (format t "bar") t)
```

**131 Successive conditions**

Execute ***f1*** if condition ***c1*** is true, or else ***f2*** if condition ***c2*** is true, or else ***f3*** if condition ***c3*** is true.
Don't evaluate a condition when a previous condition was true.

```
(cond (c1 (f1))
      (c2 (f2))
      (c3 (f3)))
```

**132 Measure duration of procedure execution**

Run the procedure ***f***, and return the duration of the execution of ***f***.

```
(time (f))
```

**134 Create a new list**

Declare and initialize a new list ***items***, containing 3 elements ***a***, ***b***, ***c***.

```
(defparameter *items* (list a b c))
```

**136 Remove all occurrences of a value from a list**

Remove all occurrences of the value ***x*** from list ***items***.
This will alter the original list or return a new list, depending on which is more idiomatic.

```
(remove-if (lambda (val) (= val x)) items)
```

Functional implementation that returns a new list. = function assumes that ***items*** is a list of numbers.

**137 Check if string contains only digits**

Set the boolean ***b*** to ***true*** if the string ***s*** contains only characters in the range '0'..'9', ***false*** otherwise.

```
(setf b (every #'digit-char-p s))
```

**141 Iterate in sequence over two lists**

Iterate in sequence over the elements of the list ***items1*** then ***items2***. For each iteration print the element.

```
(map nil #'print (append items1 items))
```

**Lisp**

**147 Remove all non-ASCII characters**

Create string ***t*** from string ***s***, keeping only ASCII characters

```
(remove-if-not (lambda (i) (<= 0 i 127))
               s
               :key #'char-code)
```

**148 Read list of integers from stdin**

Read a list of integer numbers from the standard input, until EOF.

```
(read)
```

**149 Rescue the princess**

As an exception, this content is ***not*** under license CC BY-SA 3.0 like the rest of this website.

**152 Turn a character into a string**

Create string ***s*** containing only the character ***c***.

```
(defparameter *s* (string c))
```

**153 Concatenate string with integer**

Create the string ***t*** as the concatenation of the string ***s*** and the integer ***i***.

```
(format nil "~a~d" s i)
```

**157 Declare constant string**

Initialize a constant ***planet*** with string value "Earth".

```
(defparameter +planet+ "Earth")
```

This is not technically constant, but is the most common way of doing it. The third-party library Alexandria provides define-constant that could define a constant and one could also use define-symbol-macro from the standard library

**160 Detect if 32-bit or 64-bit architecture**

Execute ***f32***() if platform is 32-bit, or ***f64***() if platform is 64-bit.
This can be either a compile-time condition (depending on target) or a runtime detection.

```
(defun detect-architecture ()
  (let ((arch (machine-type)))
    (cond ((equalp arch "X86-64") (f64))
          ((equalp arch "X86") (f32))
          (t "unknown"))))
```

**161 Multiply all the elements of a list**

Multiply all the elements of the list ***elements*** by a constant ***c***

```
(mapcar (lambda (n) (* n c))
        elements)
```

**165 Last element of list**

Assign to the variable ***x*** the last element of the list ***items***.

```
(setf x (car (last items)))
```

**Lisp**

**166 Concatenate two lists**

Create the list ***ab*** containing all the elements of the list ***a***, followed by all the elements of the list ***b***.

```
(setf ab (append a b))
```

**169 String length**

Assign to the integer ***n*** the number of characters of the string ***s***.

Make sure that multibyte characters are properly handled.

***n*** can be different from the number of bytes of ***s***.

```
(setf n (length s))
```

***Alternative implementation:***

```
(length s)
```

**174 Make HTTP POST request**

Make a HTTP request with method POST to the URL ***u***

```
(drakma:http-request u :method :post)
```

***Alternative implementation:***

```
(dex:post u)
```

2x faster than drakma w/ connection-pooling

**175 Bytes to hex string**

From the array ***a*** of ***n*** bytes, build the equivalent hex string ***s*** of ***2n*** digits.

Each byte (256 possible values) is encoded as two hexadecimal characters (16 possible values per digit).

```
(lambda (bytes &aux (size (* 2 (length bytes))))
  (let ((hex (make-array size :element-type 'character :fil
l-pointer 0)))
    (prog1 hex
      (with-output-to-string (o hex)
        (map () (lambda (byte) (format o "~2,'0x" byte)) b
ytes)))))
```

Compute resulting hex string size, write to it as a string with ***format*** function

**178 Check if point is inside rectangle**

Set boolean ***b*** to ***true*** if if the point with coordinates (***x***,***y***) is inside the rectangle with coordinates (***x1***,***y1***,***x2***,***y2***) , or to ***false*** otherwise.

Describe if the edges are considered to be inside the rectangle.

```
(setf b (and (< x1 x x2)
             (< y1 y y2)))
```

**184 Tomorrow**

Assign to variable ***t*** a string representing the day, month and year of the day after the current date.

```
(setf _t (chronicity:parse "tomorrow"))
```

this assumes that "***t***" is lexically bound in the scope of the expression

**Lisp**

**187 Disjoint Set**

Disjoint Sets hold elements that are partitioned into a number of disjoint (non-overlapping) sets.

```
(defstruct disjoint-set
  elements)

(defun make-disjoint-set (elements)
  (make-disjoint-set :elements elements))

(defun partition (disjoint-set)
  (loop for element in (disjoint-set-elements disjoint-set)
        collect
         (let ((found (find element (mapcar #'first (rest d
isjoint-set))
                            :test #'equal)))
           (if found
               (acons element element found :test #'equal)
               (acons element element nil :test #'equa
l)))))
```

**189 Filter and transform list**

Produce a new list ***y*** containing the result of the function ***T*** applied to all elements ***e*** of the list ***x*** that match the predicate ***P***.

```
(defvar y (mapcar #'T (remove-if-not p x)))
```

**Lisp**

**219 Replace multiple spaces with single space**

Create the string ***t*** from the value of string ***s*** with each sequence of spaces replaced by a single space.

Explain if only the space characters will be replaced, or the other whitespaces as well: tabs, newlines.

```
(defun words (str)
  (if (equalp str "") nil
      (let ((p (position #\Space str )))
    (cond ((null p) (list str))
          ((zerop p ) (words (subseq str 1)))
          (T (cons (subseq str 0 p) (words (subseq str (+
1 p ))))))))

(setf s " aa  bbb  cc1 ")

 (let ((ws (words s )))
   (setf t (car ws))
   (dolist (w (cdr ws ))
     (setf t (concatenate 'string t " " w ))))
 (print t)
```

The ***words*** function is https://programming-idioms.org/idiom/49/split-a-space-separated-string/6327/lisp

**221 Remove all non-digits characters**

Create string ***t*** from string ***s***, keeping only digit characters ***0***, ***1***, ***2***, ***3***, ***4***, ***5***, ***6***, ***7***, ***8***, ***9***.

```
(let ((*t* (remove-if-not #'digit-char-p *s*)))
       (format t "~A~%" *t*))
```

**223 for else loop**

Loop through list ***items*** checking a condition. Do something else if no matches are found.

A typical use case is looping through a series of containers looking for one that matches a condition. If found, an item is inserted; otherwise, a new container is created.

These are mostly used as an inner nested loop, and in a location where refactoring inner logic into a separate function reduces clarity.

```
(loop for item in items
      when (check-match item) return (do-something item)
      finally (return (do-something-else))
```

***check-match*** ***do-something*** and ***do-something-else*** are all invented function names.

The finally clause runs only if the return clause does not run.

**Lisp**

**238 Xor byte arrays**

Write in a new byte array ***c*** the ***xor*** result of byte arrays ***a*** and ***b***.

***a*** and ***b*** have the same size.

```
(map '(vector (unsigned-byte 8)) #'logxor a b)
```

Produces ***c***

**243 Print list**

Print the contents of the list or array ***a*** on the standard output.

```
(print a )
```

***Alternative implementation:***

```
(defun plist (lst)
  (if (< (length lst) 20 )
      (loop for elem in lst do (p elem))
      (let ((l (length lst )))
         (p (nth 0 lst )(nth 1 lst )(nth 2 lst )(nth 3 lst
)
         "... (" l " in total)... "
         (nth (- l 4) lst )(nth (- l 3 ) lst )(nth (- l 2
) lst )(nth (- l 1 ) lst ) )
      )))

(defun p (x &rest others)
  (cond ((listp x)( plist x   ))
        ((stringp x)(format t x  ))
        (T (write x)))
  (format t " ")
  (if others (p others))
```

pretty-print a list, abbreviating the middle part if the list is too long.

**256 Count backwards**

Print the numbers 5, 4, ..., 0 (included), one line per number.

```
(dotimes (i 6) (print (- 5 i)))
```

**261 Format time hours-minutes-seconds**

Assign to the string ***x*** the value of fields (hours, minutes, seconds) of the date ***d***, in format HH:MM:SS.

```
(defvar *x* (format NIL "~2,'0D:~2,'0D:~2,'0D" 12 34 56))
```

**Lisp**

**266 Repeated string**

Assign to the string ***s*** the value of the string ***v*** repeated ***n*** times, and write it out.

E.g. ***v***="abc", ***n***=5 ⇒ ***s***="abcabcabcabcabc"

```
(do ((i 0 (+ i 1 )))( (= i n) s)
  (setf s (concatenate 'string s v)))
```

**272 Play FizzBuzz**

Fizz buzz is a children's counting game, and a trivial programming task used to affirm that a programmer knows the basics of a language: loops, conditions and I/O.

The typical fizz buzz game is to count from 1 to 100, saying each number in turn. When the number is divisible by 3, instead say "Fizz". When the number is divisible by 5, instead say "Buzz". When the number is divisible by both 3 and 5, say "FizzBuzz"

```
(loop for i from 1 to 100
   do (print
       (cond ((zerop (mod i 3))
              (if (zerop (mod i 5)) "FizzBuzz" "Fizz"))
             ((zerop (mod i 5)) "Buzz")
             (t (format nil "~d" i)))))
```

**273 Check if folder is empty**

Set the boolean ***b*** to true if the directory at filepath ***p*** is empty (i.e. doesn't contain any other files and directories)

```
(let ((b (directory p)))
  (if (null b) t nil))
```

**277 Remove an element from a set**

Remove the element ***e*** from the set ***x***.

Explains what happens if ***e*** was already absent from ***x***.

```
(defun rmv (elem lst)
 (if (null lst) nil
   (if (equal elem (car lst))
     (cdr lst)
     (cons (car lst) (rmv elem (cdr lst))))) )

(setf x (rmv e x ))
```

**288 Check if set contains a value**

Set the boolean ***b*** to ***true*** if the set ***x*** contains the element ***e***, ***false*** otherwise.

```
(setf b (not (null (member e x))))
```

**299 Comment out a single line**

Write a line of comments.

This line will not be compiled or executed.

```
;; comment
```

Any number of semicolons will make a comment, but there is a conventional amount for different contexts.

**Lisp**

**301 Recursive Fibonacci sequence**

Compute the Fibonacci sequence of ***n*** numbers using recursion.

Note that naive recursion is extremely inefficient for this task.

```
(defun fibonacci (n &optional (a 0) (b 1) (acc ()))
  (if (zerop n)
      (nreverse acc)
      (fibonacci (1- n) b (+ a b) (cons a acc))))
```