Common Lisp

¿Cuántos paréntesis necesitas?

Adrián Arroyo Calle

http://adrianistan.eu

Historia

John McCarthy inventó Lisp en 1959 en el MIT. Aparte de diseñar Lisp, es famoso por sus contribuciones a la Inteligencia Artificial, la lógica y al concepto de time-sharing.

List Processing

- Lisp viene de LISt Processing.
- Se le suele considerar el segundo lenguaje de programación de la historia (primero FORTRAN, tercero COBOL).
- El lenguaje se organiza con s-expresiones, que forman un árbol.

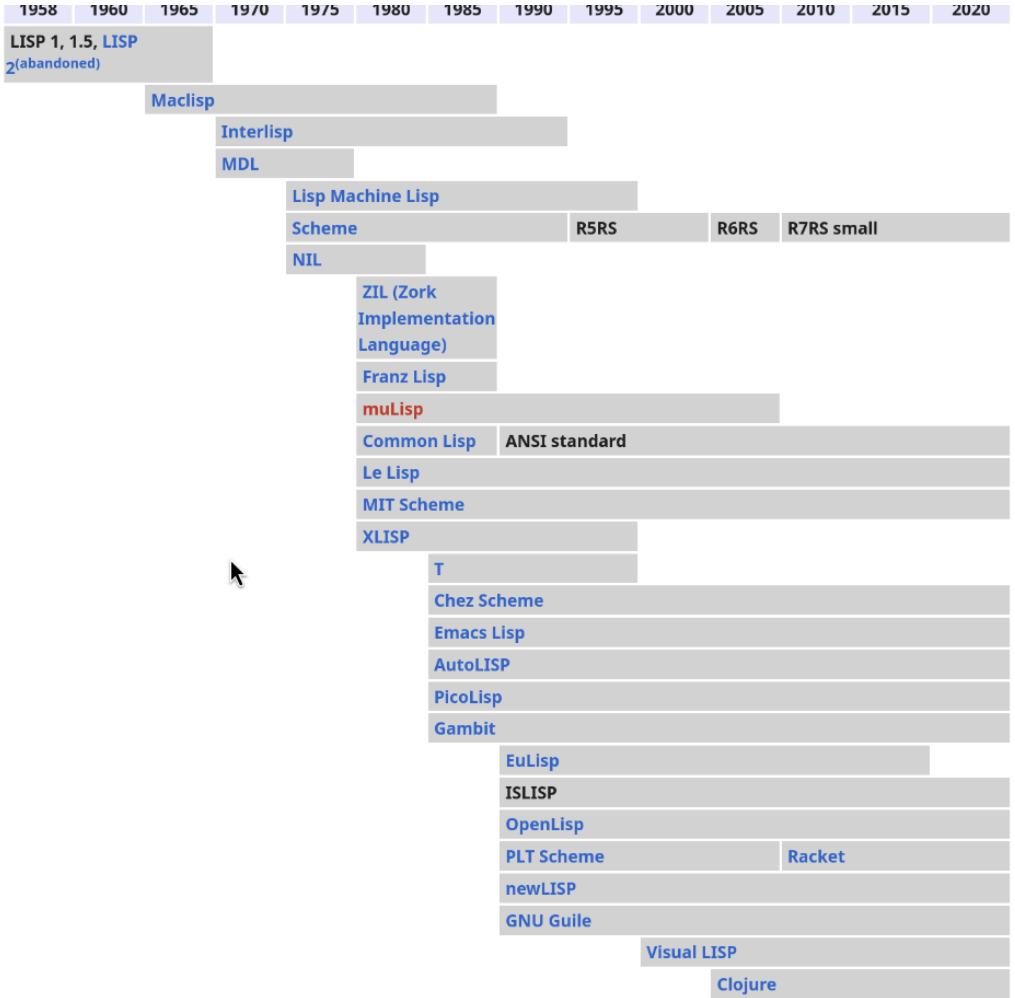
Innovaciones primigenias

- Condicionales (if/else,...)
- Memoria dinámica (malloc, new, ...)
- Programación funcional (map, reduce, ...)
- Recursividad (TCO, ...)
- Recolector de basura
- REPL y compilador autocontenido (eval, ...)
- Macros

La Familia Lisp

- Lisp se volvió muy popular en círculos académicos.
- Se llegaron a diseñar procesadores y sistemas operativos alrededor de Lisp (Genera, Interlisp, Lisp Machines, …)
- Inmensa cantidad de dialectos, todos con s-expresiones, funciones de primer orden y la lista enlazada como estructura de dato fundamental.

Common Lisp


1958
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
LISP 1, 1.5, LISP 2 (abandoned)
Maclisp
Interlisp
MDL
Lisp Machine Lisp
Scheme
R5RS
R6RS
R7RS small
NIL
ZIL (Zork Implementation Language)
Franz Lisp
muLisp
Common Lisp
ANSI standard
Le Lisp
MIT Scheme
XLISP
T
Chez Scheme
Emacs Lisp
AutoLISP
PicoLisp
Gambit
EuLisp
ISLISP
OpenLisp
PLT Scheme
Racket
newLISP
GNU Guile
Visual LISP
Clojure

Scheme

- Una versión de Lisp minimalista, enfocada en la programación funcional
- Guy L. Steele Jr. y Gerald Sussman - 1975
- Impone Tail Call Optimization y Lexical Scope
- Introduce las continuaciones
- Lisp tipo 1

Emacs Lisp

- Variante interna de Emacs basada en Maclisp
- Emacs no es un editor, ~~es un sistema operativo~~, es un entorno de programación Emacs Lisp
- Uno de los más rudimentarios y simples. Mantiene dynamic scoping.

Clojure

- Siglo XXI
- Da más protagonismo a otras estructuras de datos: vectores, diccionarios, sets…
- Estructuras de datos inmutables por defecto
- Soporte a concurrencia nativo
- Implementación principal sobre la JVM, pero también en Dart, .NET, LLVM (Jank), JavaScript (ClojureScript)

Otros sistemas

- Racket, sobre Scheme, lenguaje para lenguajes
- Janet, pequeño intérprete
- Fennel, compila a Lua
- LFE, sobre la plataforma Erlang
- PicoLisp
- ISLISP, estandarizado ISO
- y otros dialectos desaparecidos

Historia Common Lisp

- Objetivo de unificar los Lisp existentes, intentando ser compatible con los Lisp ya existentes en la medida de lo posible.
- Comienza el trabajo en 1981, colaboran entre otros Guy L. Steele Jr.
- Estandarizado en 1994 por ANSI
- Multiparadigma, maneja funcional, imperativo y orientado a objetos
- CLOS
- Macros
- Sistema de condiciones
- Lisp tipo 2

Implementaciones

- Common Lisp es implementado por varios sistemas
  - LispWorks (pago)
  - Allegro Common Lisp (pago)
  - Steel Bank Common Lisp (libre)
  - Armed Bear Common Lisp (libre, sobre la JVM)
  - CLISP (libre)
  - Clozure CL (libre)
  - ECL (libre)
  - Poplog (libre)
- El más popular es SBCL

HyperSpec

https://www.lispworks.com/documentation/HyperSpec/Front/

Welcome to the *Common Lisp HyperSpec*.
I hope it serves your need.

--*Kent Pitman*, X3J13 Project Editor

Starting points Here are some useful starting points:

Highlights Contents..... Chapter 1..... Chapter 2..... Master Index M N Symbol Index S T Glossary, *n.* Index of terms. x3j13 issues

Help A text-only version of this cover sheet is available.

Hola Mundo

Abrimos SBCL:

```
sbcl
# or
rlwrap sbcl
```

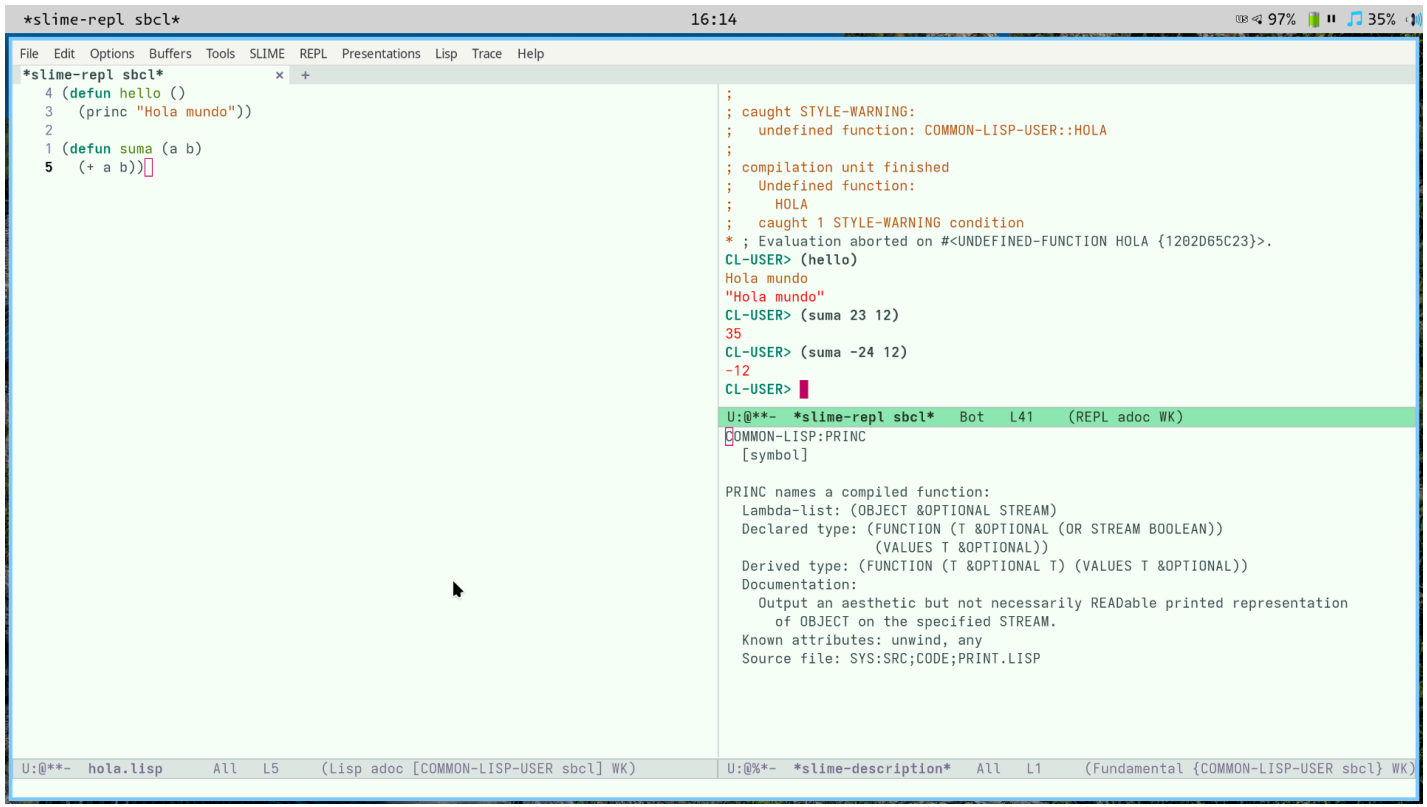
```
* (princ "Hola Mundo")
Hola Mundo
"Hola Mundo"
```

- princ es una función que imprime un objeto valor por salida estándar y además devuelve ese valor

Expresiones

- En Common Lisp todo son expresiones: todo tiene que devolver algo, incluido bucles y condicionales.
- A su vez todo puede ser evaluado como true o false. Todo es true salvo nil, que veremos más adelante.

SLIME


slime-repl sbcl
16:14
97%
35%
File Edit Options Buffers Tools SLIME REPL Presentations Lisp Trace Help
slime-repl sbcl
4 (defun hello ()
3 (princ "Hola mundo"))
2
1 (defun suma (a b)
5 (+ a b))
;
; caught STYLE-WARNING:
; undefined function: COMMON-LISP-USER::HOLA
;
; compilation unit finished
; Undefined function:
; HOLA
; caught 1 STYLE-WARNING condition
* ; Evaluation aborted on #<UNDEFINED-FUNCTION HOLA {1202D65C23}>.
CL-USER> (hello)
Hola mundo
"Hola mundo"
CL-USER> (suma 23 12)
35
CL-USER> (suma -24 12)
-12
CL-USER>
U:@**- *slime-repl sbcl* Bot L41 (REPL adoc WK)
COMMON-LISP:PRINC
[symbol]
PRINC names a compiled function:
Lambda-list: (OBJECT &OPTIONAL STREAM)
Declared type: (FUNCTION (T &OPTIONAL (OR STREAM BOOLEAN))
(VALUES T &OPTIONAL))
Derived type: (FUNCTION (T &OPTIONAL T) (VALUES T &OPTIONAL))
Documentation:
Output an aesthetic but not necessarily READable printed representation
of OBJECT on the specified STREAM.
Known attributes: unwind, any
Source file: SYS:SRC;CODE;PRINT.LISP
U:@**- hola.lisp All L5 (Lisp adoc [COMMON-LISP-USER sbcl] WK)
U:@%*- *slime-description* All L1 (Fundamental {COMMON-LISP-USER sbcl} WK)

SLIME

- SLIME es un entorno de programación para Common Lisp sobre Emacs.
- Algunos comandos
  - `M-x slime` iniciar SLIME en el buffer actual
  - `C-c C-c` compilar buffer actual
  - `C-c C-z` ir al REPL de Common Lisp
  - `C-x C-e` evaluar s-expresión
  - `C-M-x` evaluar función actual

Reader y Evaluator

- En un lenguaje tradicional, el compilador o intérprete es una caja negra
  - Lexer, parser y code generation son etapas que no podemos controlar
- En Lisp está caja se abre.
  - El Reader lee los caracteres para formar s-expresiones (expresiones atómicas o listas con paréntesis)
  - El Evaluator usa las s-expresiones y ejecuta las formas.

- Podemos ejecutar código durante el Reader de modo que generamos más formas que las que originalmente había en el código: mediante macros

S-Expresiones

```
(foo 1 2)
("foo" 1 2)
```

ambas son s-expresiones válidas, pero solo una puede evaluarse como una forma Lisp

Listas y s-expresiones

- La estructura de datos fundamental de Lisp son las listas enlazadas.
- Cada elemento se compone de un item y una referencia al siguiente elemento o nil si no hay más elementos.
- La lista vacía es nil
- Se pueden crear con la función `cons` o con `list`
- Una s-expresión es una lista o un átomo

```
CL-USER> (cons 1 (cons 2 nil))
(1 2)
CL-USER> (list 1 2)
(1 2)
```

Quoting

- Por defecto, todas las s-expresiones se van a intentar evaluar
- Pero mediante el quoting (comilla simple) podemos parar el efecto en una s-expresión

```
CL-USER> '(1 2 3)
(1 2 3)
CL-USER> 'ok
OK
```

Quasiquoting

- Podemos usar backquotes y comas para realizar un quoting selectivo

```
CL-USER> '(1 2 (+ 1 2))
(1 2 (+ 1 2))
CL-USER> `(1 2 ,(+ 1 2))
(1 2 3)
CL-USER> `(1 2 ,(list 3 4))
(1 2 (3 4))
CL-USER> `(1 2 ,@(list 3 4))
(1 2 3 4)
```

Aritmética y comparadores

- Las funciones aritméticas y comparadoras no se diferencian de las demás, pero pueden parecernos inusuales.

```
CL-USER> (+ 1 2 (* 3 4))
15
CL-USER> (> 45 30)
T
CL-USER> (= 30 12)
NIL
```

Funciones

- Usamos `defun` para definir funciones. Se soportan argumentos keyword, opcionales, y variables pero de momento no los vamos a usar.
- La lista de argumentos se ubica después del nombre de la función
- La primera línea bajo la declaración, dentro del body, puede usarse para generar una docstring.

Año bisiesto

Los años bisiestos son aquellos que son divisibles entre 4, salvo en el caso que sean divisibles por 100. En ese caso, solo serán bisiestos si además son divisibles por 400.

Funciones de utilidad: not, and, or, =, mod, zerop

Año bisiesto (solución)

```
(defun leap-year-p (year)
  "Checks if the year is a leap year"
  (and (zerop (mod year 4))
       (or (zerop (mod year 400))
           (not (zerop (mod year 100))))))
```

Condicionales

- Todos los condicionales son expresiones, es decir, tienen que devolver algo.
- Los condicionales básicos son if, case y cond. Existen variantes como when, unless, ...

```
(if (= x 1)
    (do-something)
    (do-something-else))
```

Condicional cond

```
(cond
  ((> x 0) 'positive)
  ((< x 0) 'negative)
  (t 'zero))
```

Pregunta sobre if

- ¿if puede ser una función en Common Lisp?
- ¿y cond?

Factorial

- Calcula el factorial N!, siendo `N! = 1 * 2 * 3...(N-1) * N`
- Usa if

Factorial (solución)

```
(defun factorial (n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Iteración

- Common Lisp no garantiza TCO, por lo que la recursividad puede ser mala idea en ciertos casos.
- Common Lisp es multiparadigma y también nos deja tener bucles imperativos
- dolist, dotimes y el omnipoderoso loop

```
(dolist (i '(0 1 2 3 4))
  (princ i))

(dotimes (n 5)
  (princ n))
```

- Opcionalmente dolist y dotimes admiten una variable de resultado final

Loop

- Loop define un DSL para hablar de la iteración, soporta multitud de formas de uso (como un for-each, como un for de C, como while, como list comprehension, etc)

```
(loop for x from 1 to 100
      when (zerop (mod x 2))
      collect (* x x))
```

Factorial (loop)

```
(defun factorial-loop (n)
  (loop for x from 1 to n
        with y = 1
        do (setf y (* y x))
        finally (return y)))
```

Variables

- En Common Lisp tenemos variables mutables, similares a las de lenguajes imperativos
- Creamos variables globales con defvar y defparameter
- Creamos variables locales con let y `let*`
- Actualizamos el valor de las variables con setf

defvar / defparameter

- Ambos definen variables globales
- Estas variables se suelen nombrar entre asteriscos pero es una convención
- Admiten valor inicial y documentación
- La diferencia es que si la variable ya existe defvar no hace nada, mientras que defparameter machaca el contenido anterior.

```
CL-USER> (defparameter *players* '())
*PLAYERS*
CL-USER> *players*
NIL
```

```
CL-USER> (setf *players* (cons "Adrián" *players*))
("Adrián")
CL-USER> *players*
("Adrián")
```

let

- Con let introducimos variables locales.

```
CL-USER> (let ((x 5) (y 4)) (+ x y))
9
```

- let no permite que las definiciones de variables usen otras variables del mismo let

```
CL-USER> (let ((a 5) (b (+ a 1))) (+ a b))
```

- Prueba a ver qué sucede, quizá te sorprenda que salta un menú de opciones.

```
CL-USER> (let* ((a 5) (b (+ a 1))) (+ a b))
11
```

Lexical scope

- Si las variables que introducimos son locales, estan tienen lexical scope, que es el habitual en los lenguajes de programación modernos
- Es decir, solo puedes acceder a variables si puedes verlas desde tu contexto textual
- Pero en Common Lisp las variables globales tienen dynamic scope

Dynamic scope

- En dynamic scope, la definición de una variable se ve afectada por el contexto donde ha sido llamada, aunque no tenga acceso directo

```
(defvar *x* 5)
(defun foo () (format t "~d~%" *x*))

(foo) => 5
(defun bar ()
    (let ((*x* 10))
      (foo))))
```

```
(bar) => 10
(foo) => 5
```

- La redefinición de x en la función bar ha afectado al valor que ve foo de x, aunque de forma temporal

Listas II

```
CL-USER> (defvar *languages* '("prolog" "emacs lisp"
"clojure" "smalltalk"))
*LANGUAGES*
CL-USER> *languages*
("prolog" "emacs lisp" "clojure" "smalltalk")
CL-USER> (push "kotlin" *languages*)
("kotlin" "prolog" "emacs lisp" "clojure" "smalltalk")
CL-USER> *languages*
("kotlin" "prolog" "emacs lisp" "clojure" "smalltalk")
CL-USER> (pop *languages*)
"kotlin"
```

```
CL-USER> *languages*
("prolog" "emacs lisp" "clojure" "smalltalk")
CL-USER> (append '("java" "scala") *languages*)
("java" "scala" "prolog" "emacs lisp" "clojure" "smalltalk")
CL-USER> (car *languages*)
"prolog"
CL-USER> (cdr *languages*)
("emacs lisp" "clojure" "smalltalk")
CL-USER> (cadr *languages*)
"emacs lisp"
CL-USER> (nth 2 *languages*)
"clojure"
CL-USER> (setf (nth 2 *languages*) "rust")
```

```
"rust"
CL-USER> *languages*
("prolog" "emacs lisp" "rust" "smalltalk")
CL-USER> (length *languages*)
4
```

- Extra punto: ¿Por qué se llaman car y cdr?

Property Lists y Assocs

- Existen algunas listas especiales dentro de Common Lisp. Por ejemplo, las listas que usan keywords symbols y valores o las assoc lists.

```
CL-USER> (getf '(:a 1 :b 2) :a)
1
CL-USER> (assoc 'es '((es . "Hola Mundo") (fr . "Salut
monde")))
(ES . "Hola Mundo")
```

Lambda y funciones de primer orden

- En Common Lisp, las funciones son elementos de primer orden (por ejemplo podemos pasarlo como parámetro a otras funciones).
- Pero Common Lisp es un Lisp tipo 2, quiere decir que hay dos namespaces: uno de variables y otro de funciones. Tenemos que usar `#'` para referirnos a una función.
- También podemos crear funciones anónimas

```
CL-USER> (mapcar #'(lambda (x) (* x x)) '(1 2 3))
(1 4 9)
CL-USER> (remove-if #'evenp '(1 2 3 4 5 6 7 8 9))
(1 3 5 7 9)
```

```
CL-USER> (remove-if-not #'evenp (loop for x from 1 below 10
collect x))
(2 4 6 8)
CL-USER> (reduce #'+ '(1 2 3 4 5 6 7 8 9))
45
CL-USER> (funcall #'mod 10 3)
1
CL-USER> (apply #'mod '(10 3))
1
```

Macros

- Con macros podemos generar formas Lisp basándonos en s-expresiones
- Las macros se ejecutan en antes del código normal, puede que incluso con mucha diferencia de tiempo (compile-file vs load), son parte de lo que en otros lenguajes sería el propio compilador
- Las s-expresiones de las macros no tienen por qué ser formas Lisp válidas

Macro deg2rad

```
(defmacro deg2rad (a)
    (if (numberp a)
        (* a (/ pi 180))
        `(* ,a ,(/ pi 180))))
```

- si la macro se invoca con una constante numérica, directamente se calcula el número en tiempo de compilación.
- si no lo es (se usa una variable), se genera el código para hacer la multiplicación cuando toque

Arrays y Hash Tables

- Podemos crear arrays con `make-array` o con la sintaxis `#(1 2 3 4)`.
- Accedemos a los elementos de un array con `aref`
- Podemos crear hash tables con `make-hash-table`
- Accedemos a los elementos de una hash table con `gethash`

```
CL-USER> (make-array '(4))
#(0 0 0 0)
CL-USER> (aref #(1 2 3) 0)
1
CL-USER> (setf (gethash :es ht) "Hola Mundo")
"Hola Mundo"
```

```
CL-USER> (setf (gethash :zh ht) "你好")
"你好"
CL-USER> ht
#<HASH-TABLE :TEST EQL :COUNT 2 {12028FAEA3}>
CL-USER> (gethash :zh ht)
"你好"
T
```

- En este ejemplo se ve como Common Lisp permite devolver más de un valor. No vamos a entrar en detalle en esta funcionalidad.

progn

- En algunos sitios veremos que no podemos escribir más de una s-expresión. Si queremos hacer más de una cosa deberemos usar progn.

```
(setf numbers '(1 2 3 4 5 6 7 8 9))
(setf even-numbers '())
(setf sum-even-numbers 0)
(dolist (n numbers even-numbers)
  (if (evenp n)
      (progn
        (push n even-numbers)
```

```
      (setf sum-even-numbers (+ sum-even-numbers n)))
    nil))
```

Igualdad

- En Common Lisp existen varias funciones para comparar igualdad
  - = solo para números
  - `eq` compara a nivel de punteros
  - `eql` compara con eq o a nivel numérico
  - `equal` compara si dos valores tienen la misma representación
  - `equalp` también admite como iguales valores de diferentes tipos pero que representen lo mismo o strings sin importar mayúsculas o minúsculas
  - `string=, char=, string-equal, char-equal,` ... existen varias funciones especializadas

Retos

- La conjetura de Collatz propone que desde cualquier número es posible llegar a 1 siguiendo las siguientes normas:
  - Si el número es par, dividir entre 2
  - Si el número es impar, multiplicar por 3 y sumar 1
- Dado un número N, ¿cuántos pasos hay que dar hasta llegar a 1?

Collatz

```
(defun collatz (n)
  (when (> n 0)
    (if (= n 1)
        0
        (+ 1 (if (evenp n)
                 (collatz (/ n 2))
                 (collatz (+ 1 (* n 3))))))))
```

Números perfectos

- Se dice que un número es perfecto si la suma de sus factores es igual a él mismo
  - Por ejemplo, 6, sus factores son 1, 2 y 3. Su suma da 6 también.
- Si la suma de los factores da un número inferior, se dice que es deficiente
- Si la suma de los factores da un número superior, se dice que es abundante
- Dado un número N, devolver si el número es perfecto, deficiente o abundante

Números perfectos

```
(defun classify (number)
  (when (> number 0)
    (let ((factorsum (apply #'+ (factors number))))
         (cond
           ((= factorsum number) "perfect")
           ((< factorsum number) "deficient")
           ((> factorsum number) "abundant")))))

(defun factors (number)
  (loop for i from 1 below number
```

```
        when (zerop (mod number i))
        collect i))
```

Sieve

- La criba de Eratóstenes es un método para calcular números primos hasta N
- Se empieza con una lista de todos los números entre 2 y N
- Iterativamente, se coge el primer elemento de la lista y se declara primo
- Todos los múltiplos de ese número se quitan de la lista y se repite el proceso

Sieve

```
(defun primes-to (n)
  "List primes up to `n' using sieve of Eratosthenes."
  (let ((numbers (loop for i from 2 to n collect i)))
       (primes-sieve numbers)))

(defun primes-sieve (numbers)
  (if (zerop (length numbers))
      '()
      (cons
       (car numbers)
       (primes-sieve
```

```
(remove-if
 #'(lambda (n)
             (zerop
              (mod n (car numbers))))
 (cdr numbers))))))
```

Factores primos

- Obtener los factores primos de N

Factores primos

```
(defun factors (n)
  (let ((prime-factors '())
        (current-test-factor 2)
        (rest n))
       (loop while (not (= rest 1))
             do (if (zerop (mod rest current-test-factor))
                    (progn
                      (push current-test-factor prime-factors)
                      (setf rest (/ rest current-test-factor)))
                    (setf current-test-factor (+ current-test-
```

```
factor 1)))))
                finally (return (nreverse prime-factors)))))
```

Format

- Existe una función cómoda para formatear strings, llamada `format`.
- El primer argumento es el destino que puede ser un stream, t (stdout) o nil
- En caso de nil, format simplemente devuelve el string formateado pero no lo escribe ni en un fichero ni en la terminal.
- Los operadores de format son diferentes a los de printf de C
- `~%` nueva línea
- `~s` imprimir contenido de variable en formato legible por Lisp
- `~d` imprimir número decimal
- `~f` imprimir número con coma flotante

- ~a imprimir contenido de varibale en formato “humano”

```
CL-USER> (format t "Hola, este es el mensaje n ~d para el ~s,
por favor contesta, ~a" 12 "GUI" "GUI")
Hola, este es el mensaje n 12 para el "GUI", por favor
contesta, GUI
```

Abrir archivos

- Podemos usar `with-open-file` para abrir archivos, especificando multitud de opciones

```
(with-open-file (output "output-file.txt"
                        :direction         :output
                        :if-does-not-exist :create
                        :if-exists         :append)
  (format output "1 + 2 = ~d~%" 3))
```