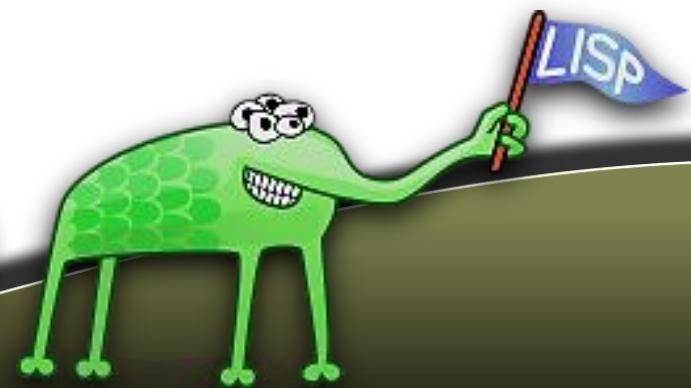


Taller de programación en Common Lisp

01

Fundamentos de Common LISP





SISTEMA MOODLE

NAVIGACIÓN

- Página Principal (home)
- Site announcements
- Cursos

Buscar cursos:

Ir

Cursos disponibles

Fundamentos de Inteligencia Artificial...

Evaluaremos la disciplina que ha contado la

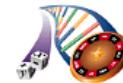
Taller de programación COMMON LISP...



En este taller examinamos el primer lenguaje orientado a programación simbólica (vs programación numérica) y exploramos las capacidades de éste que lo llevaron a convertirse en el lenguaje de programación favorito en el área de Inteligencia Artificial...

convertirse en el lenguaje de programación favorito en el área de Inteligencia Artificial...

Cómputo evolutivo y bioinspirado



El objetivo del curso es que el estudiante aprenda diferentes técnicas de algoritmos evolutivos y bioinspirados para la solución de problemas de optimización .

Teoría de la Computación, A21



Teoría de autómatas y lenguajes formales, el fundamento teórico de la computación.

CALENDARIO

MARCH 2021

Dom	Lun	Mar	Mié	Jue	Vie	Sáb
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
	30	31				

ENLACES EN LÍNEA

(se han visitado 5 minutos: 0)
Ninguno(a)



Libros y material de consulta...



Practical
Pe

Presentaciones de clase...

Tutoriales, ejercicios y m



Lisp Quickstart



Lisp in small



elm
ELM-ART Tutorial



Casting spells in lisp



Tutorials Point



WIKIBOOKS



ILLINOIS
Lisp style tips
for beginners



Lisp Cookbook



Ler con cuidado
el comic al final
de este sitio...



Learn Lisp the hard way

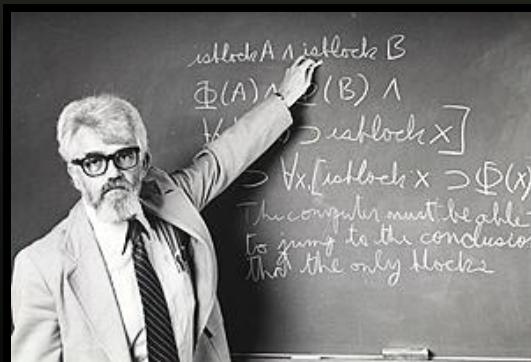
Contexto del lenguaje...



El inicio...

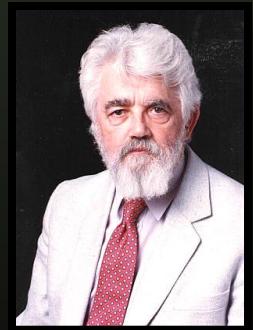


John McCarthy (1927-2011) en el M.I.T. crea *LISP* en 1956, basado en el *Cálculo Lambda* para realizar Procesamiento Simbólico.

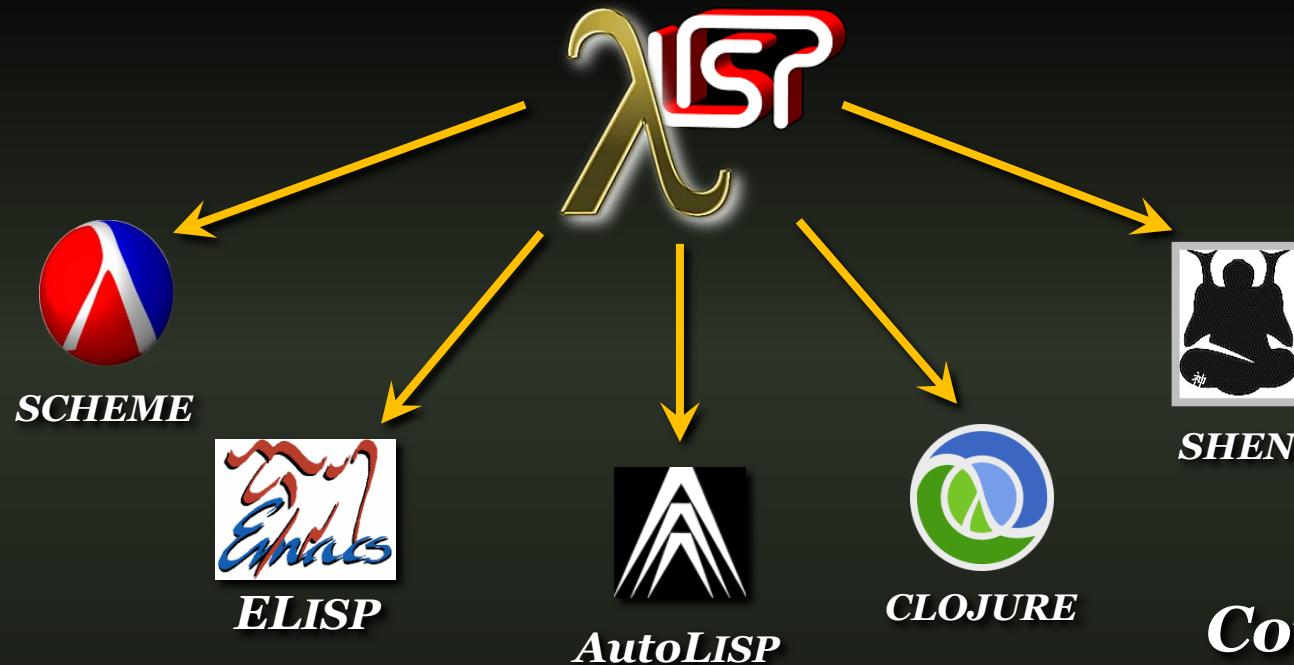


LISt Processor

Especialidad para expresar algoritmos recursivos y manejar tipos de datos dinámicos.



Dialectos y sabores...



Common LISP

ANSI, 1996



Smalltalk



JavaScript



Ruby



Lua



- ◆ Manejo dinámico de la memoria.
- ◆ Soporte completo para recursividad.
- ◆ Aritmética de enteros con precisión arbitraria.
- ◆ Programación multiparadigma.
- ◆ Soporta *Threads*.
- ◆ Modo *Intérprete* y Modo *Compilador*.
- ◆ Un solo estándar (***ANSI INCITS 226-1996 (R2004)***)

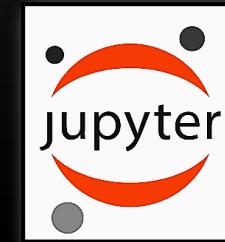


Steel Bank Common Lisp
(SBCL)

- ◆ Intérprete/Compilador sobre línea de comandos...
- ◆ Gratis (licencia GNU)...
- ◆ Multiplataforma (Linux, Mac OS/X, Windows, etc)...

Para programar pueden usar el editor de su preferencia, pero se recomienda usar *EMACS*, *JUPYTER* o *ATOM*...

*[http://functionalrants.wordpress.com/2008/09/06
/how-to-set-up-emacs-slime-sbcl-under-gnulinux/](http://functionalrants.wordpress.com/2008/09/06/how-to-set-up-emacs-slime-sbcl-under-gnulinux/)*



Steel Bank Common Lisp

<http://www.sbcl.org>

Steel Bank Common Lisp

About

Steel Bank Common Lisp (SBCL) is a high performance Common Lisp compiler. It is open source / free software, with a permissive license. In addition to the compiler and runtime system for ANSI Common Lisp, it provides an interactive environment including a debugger, a statistical profiler, a code coverage tool, and many other extensions.

SBCL runs on a number of POSIX platforms, and experimentally on Windows. See the [download](#) page for supported platforms, and [getting started](#) guide for additional help.

The most recent version is [SBCL 2.0.9](#), released September 27, 2020 ([release notes](#)).

Documentation

SBCL's manual is available on the web in [html](#) and [pdf](#) formats. See the doc/manual directory in the source code for the current version in TeXInfo source.

Reporting Bugs

Bugs can either be reported directly to [SBCL's bug database on Launchpad](#), or by sending email to the sbcl-bugs@lists.sourceforge.net mailing list -- no subscription required.

- [About](#)
- [News](#)
- **[Download](#)**
- [Getting Started](#)
- [History and Copyright](#)
- [Porting](#)
- [Maintainer public keys](#)
- [Manual](#)
- [Project Page](#)
- [Bug Database](#)
- [Links](#)



Steel Bank Common Lisp

<http://www.sbcl.org>

Steel Bank Common Lisp

Download

The most recent version of SBCL is 2.0.9, released September 27, 2020. [Release notes](#).

Source: [sbcl-2.0.9-source.tar.bz2](#)

The development version is available from git:

```
git clone git://git.code.sf.net/p/sbcl/sbcl
```

Binaries:

After downloading SBCL, refer to the [getting started](#) page for instructions on how to install the release.

Not all platforms have the latest binaries, but SBCL is still supported and working on these platforms. An older binary (or provided by an OS repository / homebrew / macports) or even a different CL implementation can be used to build the [latest source](#) by following the directions for [compiling it](#).

The Linux binaries might require a recent glibc, but building from source isn't dependent on a particular glibc version

	X86	AMD64	PPC	PPC64	PPC64le	SPARC	Alpha	MIPSbe	MIPSle	ARMel	ARMhf	ARM64	RISC-V 32	RISC-V 64
Linux	1.4.3	2.0.9 newest	1.2.7		1.5.8	1.0.28	1.0.28	1.0.23	1.0.28	1.2.7	1.4.11	1.4.2		
Darwin (Mac OS X)	1.1.6	1.2.11	1.0.47											
Solaris	1.2.7	1.2.7						2.0.4						
FreeBSD	1.2.7	1.2.7	1.0.47											
NetBSD	1.0.22	1.2.7	1.0.23											
OpenBSD	2.0.5	2.0.5	2.0.5						2.0.5		2.0.5			
DragonFly BSD			1.2.7											
Debian GNU/kFreeBSD	1.2.7	1.2.7												
Windows	1.4.14	2.0.0												

	Key
	Available and supported
	Port in progress
	Not available (porters welcome!)
	No such system

	Processors
X86	X86 (32-bit Intel and compatible)
AMD64	64-bit X86 (AMD64, EM64T, Via Nano)
PPC	PowerPC
PPC64	64-bit PowerPC
PPC64le	Little Endian 64-bit PowerPC
SPARC	SPARC and UltraSPARC
Alpha	DEC Alpha
MIPSbe	MIPS (big endian mode)
MIPSle	MIPS (little endian mode)

Laboratorio de Inteligencia Artificial

Todas las funciones de *LISP* se expresan siempre en:

Notación Prefija (de Cambridge)

$$f(x, y, z) \rightarrow (f\ x\ y\ z)$$

etiqueta argumentos

etiqueta argumentos

Aridad
indefinida...

>> (+ 35 46 -12)

69

>> (* 3.2 7)

22.4

>> (- 12 4 2 5)

1

>> (/ 47 3.8)

12.368422

En *Common LISP* no se hace *asignación* de variables como en otros lenguajes...



En su lugar, los símbolos se ASOCIAN a valores de cualquier tipo...

Por ello, las variables (símbolos) no tienen *tipo de datos*, sólo los valores asociados lo tienen...

- ◆ Átomo: 

Numéricos	$7, 3, 45.28, 79.12, \dots$
No numéricos	<i>HOLA, "CASA", VALOR23, NIL, T</i>

 Valores de verdad 

- ◆ Lista: Secuencia de elementos, separados por espacios y delimitada entre paréntesis.

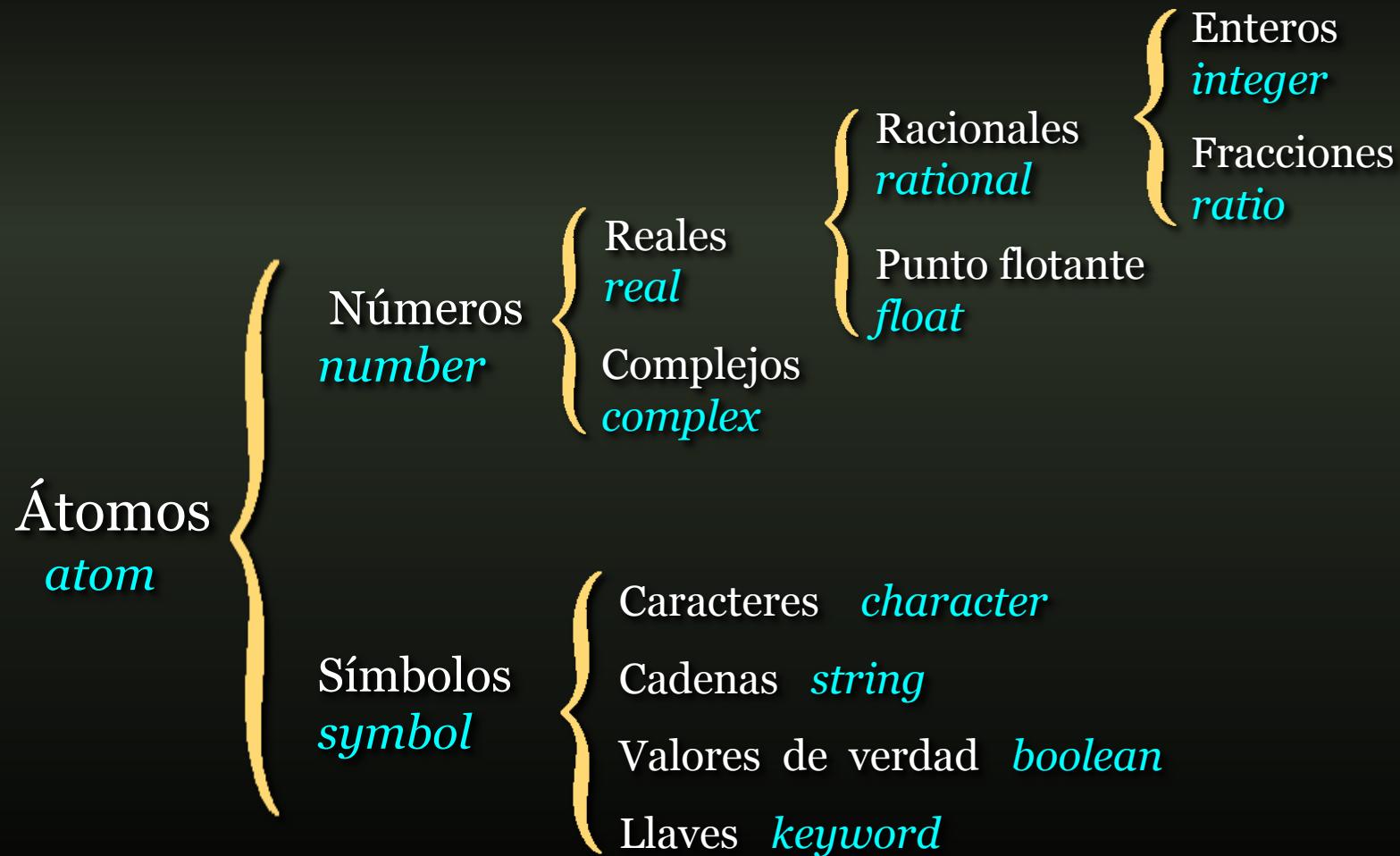
(*Esta es una LISTA*)

(*CASA 3 POLLO 45.2 (5 6 7) (3.1 3.2 3.3)*)

(*Rojo (Verde Azul) () AMARILLO*)

(*(uno dos) (TRES CUATRO CINCO)*)

Los átomos...





Procesamiento de átomos

Precisión arbitraria, no infinita...

```
>> (expt 2 (expt 2 10))  
1797693134862315907729305190789024733617976978942306572734  
3008115773267580550096313270847732240753602112011387987139  
3357658789768814416622492847430639474124377767893424865485  
2763022196012460941194530829520850057688381506823424628814  
7391311054082723716335051068458629823994724593847971630483  
5356329624224137216
```

```
>> (expt 2 (expt 2 (expt 2 100)))
```

Error: Attempt to create an integer which is too large to represent.
[condition type: SIMPLE-ERROR]

En Modo Intérprete...

La labor del intérprete es **EVALUAR** cada expresión y entregar el **VALOR** asociado a cada expresión...

Algunos átomos tienen valor asociado pre-definido:
números, caracteres, cadenas y valores de verdad

Estos átomos pueden ser directamente evaluados por el intérprete:

>> 23.47

23.47

>> "Hola"

"HOLA"

>> T

T

>> nil

NIL

Pero si se intenta evaluar un símbolo que no tiene algún valor asociado, el intérprete genera un error...

```
>> casa
```

<Error>

[condition type: UNBOUND-VARIABLE]

Error: símbolo sin valor
asociado...

```
>> X
```

<Error>

[condition type: UNBOUND-VARIABLE]

Para evitar la evaluación de expresiones se usa la función **QUOTE**:

```
>> ( QUOTE Cañaveral )
```

CAÑAVERAL

```
>> ( QUOTE X)
```

X

Quote evita la evaluación de una expresión *LISP*.

Con listas, **Quote** hace la diferencia entre considerarlas como código o como datos...

Otra forma...

La función **Quote** también se representa mediante un Apóstrofo ' antes del argumento:

```
>> 'Sandía  
SANDÍA
```

```
>> '(Uno Dos Tres)  
(UNO DOS TRES)
```

OJO: **Quote** sólo devuelve el **PRIMER** elemento (átomo o lista) de sus argumentos:

```
>> '(1 2) (3 4) (5 6)  
(1 2)  
<Error>  
[condition type: TYPE-ERROR]
```

El error lo causa la segunda lista pues al intentar evaluarla no encuentra la función “3”...

Expresiones anidadas...

$$2x^2 + 7x + 5 = 0$$

$$ax^2 + bx + c = 0$$

$$x = \frac{-7 \pm \sqrt{7^2 - 4(2)(5)}}{2(2)}$$

```
>> (/ (+ -7.0 (sqrt (- (expt 7 2) (* 4 2 5)))) (* 2 2))
```

-1.0

```
>> (/ (- -7.0 (sqrt (- (expt 7 2) (* 4 2 5)))) (* 2 2))
```

-2.5

División ...

Raíz cuadrada ...

Multiplicación ...

Substracción ...

Exponente ...

Multiplicación ...

Las cadenas...

Tipo especial de átomo alfanumérico, que se expresa entre comillas dobles y que también es un objeto auto-evaluado.

```
>> "Star Trek"
```

```
"Star Trek"
```

```
>> Star Wars
```

```
<Error>
```

```
[condition type: UNBOUND-VARIABLE]
```

```
>> ( length "Inteligencia Artificial" )
```

```
23
```

```
>> (string= "Inteligencia" "Sabiduría" )
```

```
NIL
```

```
>> (string= "Yo soy Investigador" "Yo soy Investigador" )
```

```
T
```

Caracteres...

Las cadenas también son vistas como arreglos de caracteres (con índice inicial = 0)

```
>> (char "Star Trek" 0)
```

#\S

Encontrar el carácter, dentro de una cadena, que ocupa la posición indicada ...

```
>> (char "Star Trek" 7)
```

#\e

El prefijo **#** es la forma estándar de identificar a un carácter

```
>> (char "Star Trek" 4)
```

#\Space

Los caracteres también son objetos auto-evaluados

```
>> #\K
```

```
#\K
```

```
>> (quote #\S)
```

```
#\S
```

```
>> '#\Q
```

```
#\Q
```

```
>> (quote #\ )
```

```
#\Space
```

```
>> '#\
```

```
#\NewLine
```

OJO: cuidado con la diferencia entre escribir y no escribir un espacio después del prefijo **#**

Otros ejemplos...

```
>> (char "Ahuehuéte" 6 )  
#\é
```

```
>> (char= #\u (char "fUnCiOnAl" 1 ) )
```

NIL

Determinar si dos
caracteres son
iguales...

```
>> (char= (char "Dos abanicos" 2 )  
          (char "simpático" 0 )  
          #\s )
```

T

Procesamiento básico de Listas...



En *LISP* existe una estructura de datos fundamental llamada **Celda de Construcción (*Cons Cell*)**...

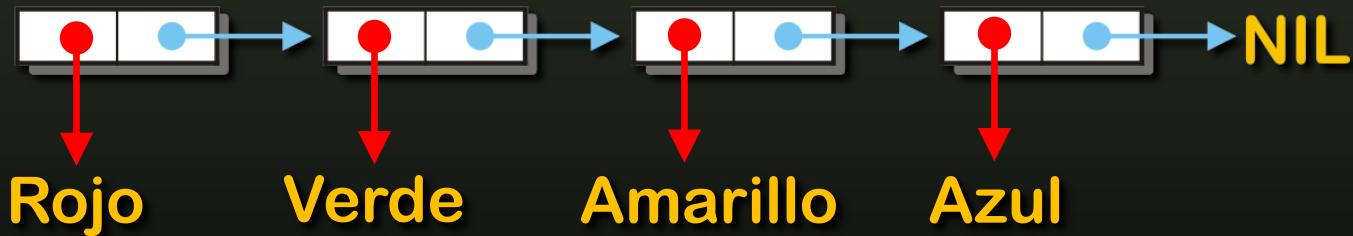
Se trata de una estructura con sólo dos campos y cada uno de ellos contiene un apuntador...



En particular, una celda puede apuntar a otra, con lo que se pueden construir cadenas ligadas de celdas

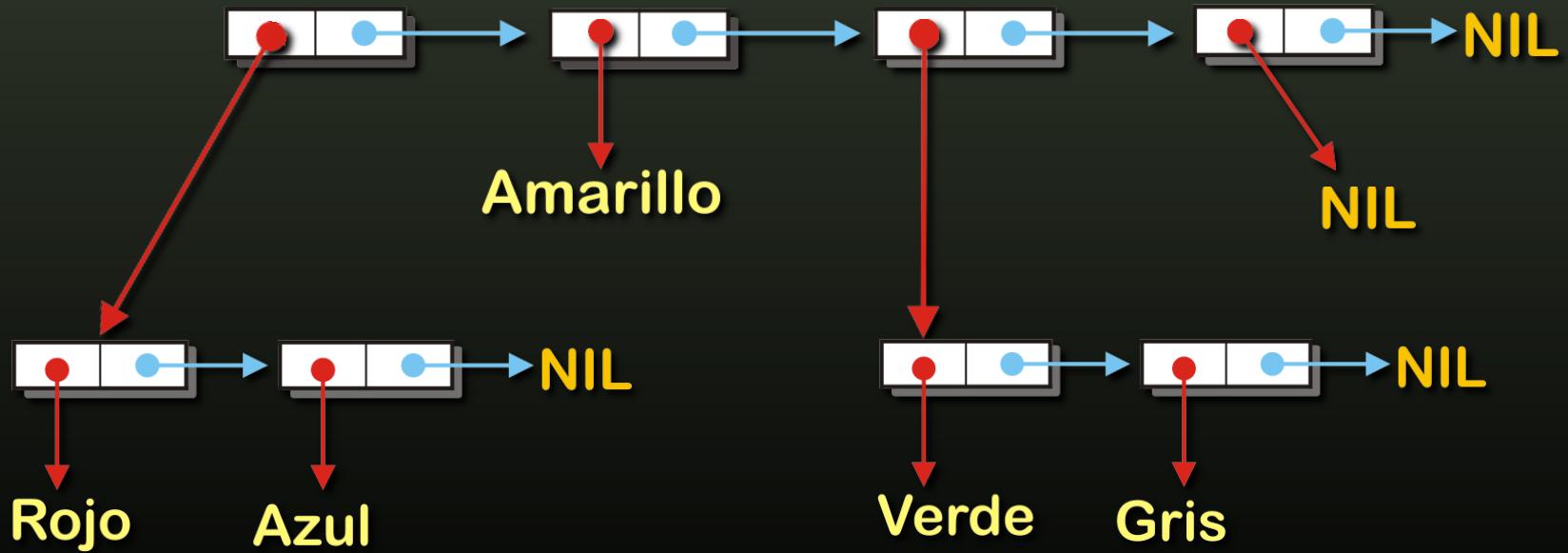
Las listas siempre se representan internamente como listas ligadas de *Celdas de Construcción* (*Cons Cells*).

(Rojo Verde Amarillo Azul)



Una lista puede representar un conjunto de datos, o bien, la invocación a una función previamente definida...

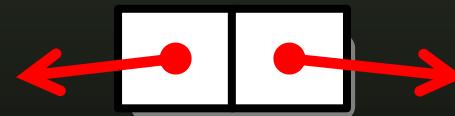
((Rojo Azul) Amarillo (Verde Gris) ())



La función CONS crea una celda de construcción y apunta cada una de sus secciones a los argumentos...

```
>> (cons 'A '(B C) )  
(A B C)
```

```
>> (cons '(1 2) '(A B C) )  
((1 2) A B C)
```



```
>> (cons 2 (cons 4 (cons 6 '(8)))) )
```

```
(2 4 6 8)
```

```
>> (cons 'a (cons 'b (cons 'c '() )) )
```

```
(A B C)
```

Otra forma de construir listas (alternativa a **CONS**) es la función **LIST**:

```
>> ( list 'A 'B  '(C D) 'E )  
( A B (C D) E )
```

```
>> ( list nil )  
(NIL)
```

```
>> ( list '(nil) )  
( (NIL) )
```

A diferencia de **CONS**, la función **LIST** acepta cualquier número de argumentos y construye la lista a partir de ellos como sus elementos integrantes...

CONS vs **LIST**...

- ◆ **CONS** agrega un elemento a una lista (una celda cons)
- ◆ **LIST** construye a partir de puros elementos
- ◆ Las listas construidas por **LIST** siempre terminan en **NIL** (listas propias)
- ◆ Es posible usar **CONS** para construir una lista no-propia (no terminada en **NIL**)

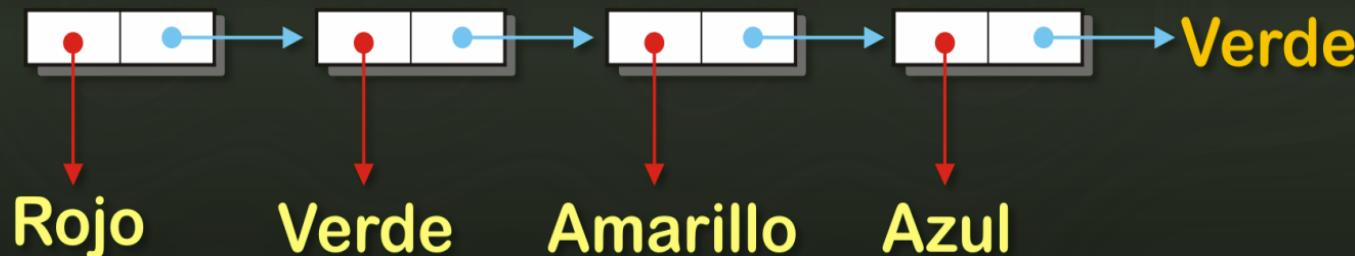
¿?

¿? ?

!!!

Notación de componentes...

¿Cómo se denota o visualiza una lista con la siguiente estructura interna?



Se usa una notación alternativa a las listas llamada *Notación de Parejas Punteadas (separadas por un punto)* (*Dotted-pair*).

(Rojo Verde Amarillo Azul . Verde)

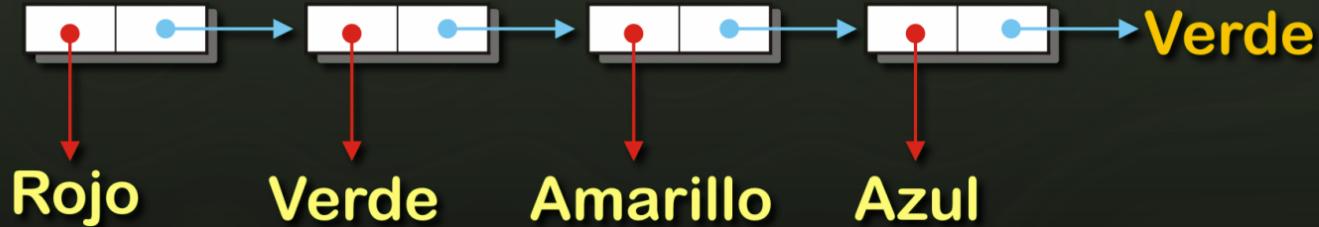
Si los dos parámetros de **CONS** resultan ser átomos (en lugar de un elemento y una lista), entonces construye una *pareja punteada*:

```
>> ( cons 'Buenos 'Días )  
( BUENOS . DÍAS)
```

```
>> ( cons 'A (cons 'B (cons 'C 'D) ) )  
( A B C . D )
```

```
>> ( list (cons 'A 'B) (cons 'C 'D) )  
((A . B) (C . D))
```

La longitud de una lista se define como el **número** de celdas, de **1er nivel**, que la componen



Por lo tanto, la longitud de:

(Rojo Verde Amarillo Azul . Verde)

es 4 y no 5 .

Inicio y resto...

Las funciones FIRST y REST regresan el elemento inicial de una lista y su resto, respectivamente...

```
>> (first '( A B C D ) )  
A
```

```
>> (rest '( A B C D ) )  
(B C D)
```

```
>> (first (cons 'a '(b c)) )  
A
```

```
>> (rest (cons 'a '(b c)) )  
(B C)
```

```
>> (first '() )  
NIL
```

```
>> (rest '() )  
NIL
```

La función **LAST** regresa la última celda (**cons**) que compone una lista. La respuesta, al ser una celda cons, siempre tiene forma de lista...

```
>> (last '(A B C D))  
(D)
```

```
>> (first (last '(a b c d)))  
D
```

```
>> (rest (last '(a b c d)))  
NIL
```

```
>> (last '())  
NIL
```

Además de las funciones **FIRST**, **REST** y **LAST**, están definidas también:

```
>> (second '( (A B) C D (E) F ) )  
C
```

```
>> (third '(A B C D E F))  
C
```

fourth, fifth, sixth, seventh, eighth,
ninth, tenth

Cuando no resultan útiles las funciones **FIRST**, **SECOND**, ...
TENTH ó para realizar búsqueda aleatoria, se puede usar la función **NTH**:

```
>> (nth 12 '(a b c d e f g h i j k l m n o p) )  
M
```

OJO: La función **NTH** considera los índices de posición comenzando en cero.

```
>> ( third '(a b c d) )  
C  
>> ( nth 3 '(a b c d) )  
D
```

La función LENGTH de cadenas también se puede usar sobre listas:

```
>> (length '( a (b c) d) )  
3
```

```
>> (length '( ( ) ( ) ( ) ) )  
3
```

equal compara listas:

```
>> (equal '(a) (first '((a)) ) )  
T
```

```
>> (equal '( a (b c) d ) '( a b c d ) )  
NIL
```

* Nota histórica ...

Antes del estándar ANSI de *Common LISP*, las funciones **FIRST** y **REST** no existían y sus equivalentes se llamaban **CAR** y **CDR** (/cou-der/)

Esas siglas son reliquias del pasado y correspondían al nombre de secciones de instrucción en la *IBM 704* (1958)

Contents of Address portion of Register
Contents of Decrement portion of Register

Aún hoy es fácil encontrar código escrito usando CAR y CDR, sin embargo no es lo recomendado...

En “*LISP* antiguo” las funciones **CAR** y **CDR** se podían combinar para examinar listas anidadas:

Las letras **A** (en **CAR**) y **D** (en **CDR**) se encadenaban en orden inverso para generar nuevas funciones...

```
>> ( CADR '( (A B) C D (E) F ) )
```

C

/ kae-der/

/ cou-dar /

```
>> ( CDAR '( (A B) C D (E) F ) )
```

(B)

```
>> ( CAAR '( (A B) C D (E) F ) )
```

A

```
>> ( CDDR '( (A B) C D (E) F ) )  
(D (E) F)
```

Pero, evidentemente, la combinación dependía de la estructura de la lista de mayor nivel:

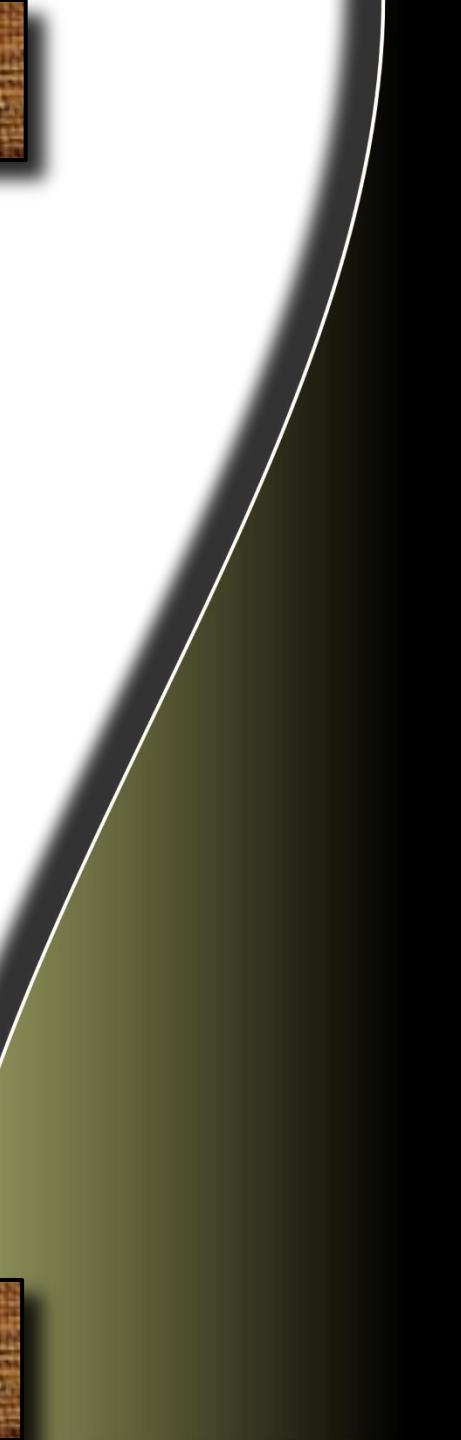
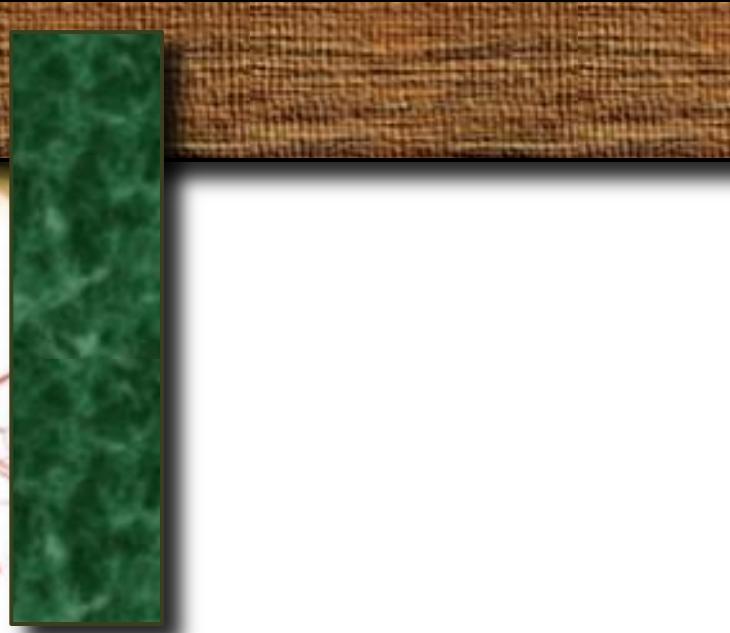
```
>> ( CDADDR '( (A B) C D (E) F ) )
```

Error: Attempt to take the CDR of C which is not listp.
[condition type: TYPE-ERROR]

Equivalencia parcial...

<i>f</i>	pronunciación	equivalencia
CAR	/ kar /	FIRST
CDR	/ cou-der /	REST
CAAR	/ka-ar/	
CADR	/kae-der/	
CDAR	/cou-dar/	SECOND
CDDR	/cou-dih-der/	
CAAAR	/ka-a-ar/	
CAADR	/ka-ae-der/	
CADAR	/ka-dar/	
CADDR	/ka-dih-der /	
CDAAR	/cou-da-ar/	THIRD
CDADR	/cou-dae-der/	
CDDAR	/cou-dih-dar/	
CDDDR	/cou-did-dih-der/	
CADDAR	/ka-dih-dih-der/	FOURTH
...

Modernamente se usan sólo las nuevas funciones, aunque las antiguas aún sirven...



Predicados de Identificación...

Se trata de funciones cuyo resultado es siempre un valor de verdad (**T**, **NIL**) y, sobre los cuales, es posible siempre aplicar los conectores lógicos

(**AND** <arg1> <arg2> ...)

(**OR** <arg1> <arg2> ...)

(**NOT** <arg>)

- ◆ **NOT** acepta estrictamente sólo un argumento
- ◆ **AND** y **OR** aceptan cualquier número de argumentos y evalúan secuencialmente...

El predicado **NULL** verifica si su único argumento, es o no, una lista vacía:

```
>> ( null '(1 2 3) )  
NIL
```

```
>> ( null '() )  
T
```

```
>> ( null NIL )  
T
```

```
>> ( null () )  
T
```

```
>> ( null '(NIL) )  
NIL
```

Otros predicados en *LISP* son los siguientes:

(**numberp** <arg>)

(**oddp** <arg>)

(**evenp** <arg>)

(**>** <arg1> <arg2>)

(**>=** <arg1> <arg2>)

(**<** <arg1> <arg2>)

(**<=** <arg1> <arg2>)

Además de los ya vistos:

(**EQUAL** <arg1> <arg2> ...)

(**STRING=** <arg1> <arg2> ...)

(**CHAR=** <arg1> <arg2> ...)

Los predicados de identificación...

Los predicados de identificación permiten saber si un objeto es de algún tipo de datos específico...

```
>> (atom '(a b c))
```

```
NIL
```

```
>> (atom 312.26)
```

```
T
```

Predicado	Tipo de datos que identifica
atom	átomos
numberp	átomos numéricos
symbolp	átomos simbólicos
listp	listas
realp	números reales
complexp	números complejos
rationalp	números racionales
floatp	números de punto flotante
integerp	números enteros
ratiop	fracciones
characterp	caracteres
alpha-char-p	caracteres alfabéticos
alphanumericp	caracteres alfanuméricos
keywordp	llaves
stringp	cadenas

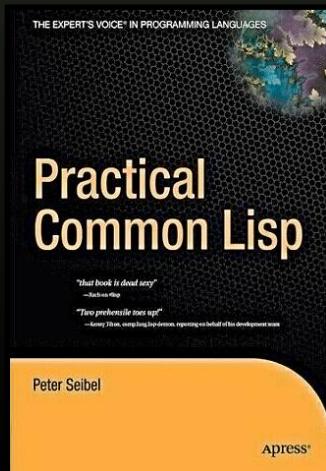
La tradición...

Por tradición, la mayoría de los predicados de identificación terminan con el carácter ‘p’

Yo prefiero usar el carácter ‘?’

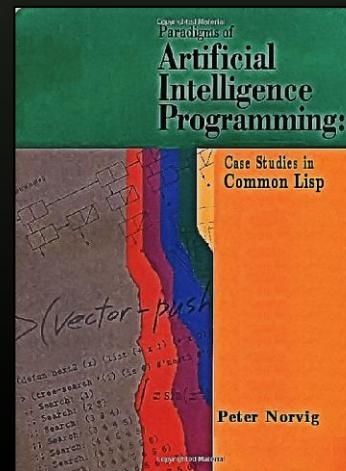
Predicado	Tipo de datos que identifica
atom	átomos
numberp	átomos numéricos
symbolp	átomos simbólicos
listp	listas
realp	números reales
complexp	números complejos
rationalp	números racionales
floatp	números de punto flotante
integerp	números enteros
ratiop	fracciones
characterp	caracteres
alpha-char-p	caracteres alfabéticos
alphanumericp	caracteres alfanuméricos
keywordp	llaves
stringp	cadenas

- 1) Bajar e instalar *SBCL* y configurarlo para su uso con *EMACS*...
- 2) Averiguar las funciones *LISP* para entrada/salida (consola y archivo), así como la forma de compilar un programa y generar código ejecutable...
- 3) Estudiar:



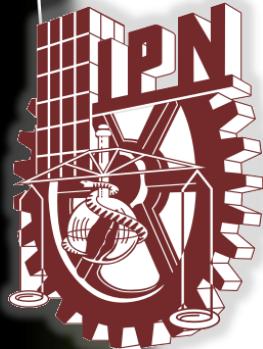
Chapter 1:
Introduction:
Why *LISP*?

Chapter 2:
Lather, Rinse, Repeat:
A Tour of the *REPL*



Preface:
Why *LISP*?
Why Common *LISP*?

Chapter 1:
Introduction to *LISP*
1.1 - 1.4



¡ Gracias !

Dr. Salvador Godoy Calderón

CIC - IPN