[**Paradigmas y Lenguajes 3**](#_heading=h.bbm3zchrjq02)

[Tipos de Paradigmas 3](#_heading=h.7hipq18q56cv)

[**Lenguaje de Programación 5**](#_heading=h.6v97vlidz1a0)

[Clasificación de Lenguajes de Programación 7](#_heading=h.ct09tfcnqfgp)

[Generaciones de los lenguajes 9](#_heading=h.v6atx873r8mi)

[Estilos de programación 9](#_heading=h.ymtfzlib927c)

[Categorías de lenguajes funcionales: híbridos y puros 10](#_heading=h.6gni4pjuqcr1)

[**Programación Funcional: Lisp 14**](#_heading=h.ff515a6s07ts)

[Características 14](#_heading=h.2ekt8p498gnp)

[Atom 15](#_heading=h.bx6rl2occ4yl)

[Caracteres y Strings 15](#_heading=h.yggcz8g6vuq)

[Átomos simbólicos 16](#_heading=h.3tcnwiv9oejg)

[Símbolos reservados 16](#_heading=h.wcf1bx9frrca)

[Lista 16](#_heading=h.vyhwfc1pom4t)

[CONS 17](#_heading=h.582jkq9dbz8)

[Listas punteadas 19](#_heading=h.5lp7cu8nig2w)

[Lisp Listener 19](#_heading=h.ivoltyevz3u)

[Funcion Eval 20](#_heading=h.5as7m5dl6zar)

[Funciones varias 21](#_heading=h.9qya5dib7qum)

[Funciones Aritméticas 22](#_heading=h.pm5tsi1idu8y)

[Truncate y Round 23](#_heading=h.68ws6jhgzha4)

[Más funciones aritméticas (float, rational, mod, abs, signum, max, min, MCD, MCM) 24](#_heading=h.92yft7dyr1rs)

[Funciones Matemáticas (sqrt, expt, incf, decf) 25](#_heading=h.h48okvqvhpww)

[Predicados de comparación 26](#_heading=h.kseq092mxqvx)

[Predicados numéricos (numberp, oddp, evenp, integerp, zerop) 27](#_heading=h.wvi3gtc8b7h5)

[Funciones para operar sobre listas (car, cdr, c\*r, last, elt) 28](#_heading=h.k53ih557po46)

[Función de asignación (setq) 29](#_heading=h.a18jvk6nko1d)

[Función de construcción de listas (cons, list, append) 30](#_heading=h.oixmkpscp7j5)

[Otras funciones de listas (butlast, nth, nthcdr, reverse) 33](#_heading=h.h7811bugft7v)

[Funciones Destructivas (rplaca, rplacd, push, pop) 34](#_heading=h.l9s6z8f3mof4)

[Otras Funciones (length, member) 35](#_heading=h.nyp0v3g68v02)

[Funciones de ligadura de datos (fmakunbound, makunbound, boundp) 36](#_heading=h.c2cz98m5w5i3)

[Predicados sobre tipos de datos (atom, listp, null, typep) 36](#_heading=h.8mwesjibliw6)

[Predicados de igualdad (eq, eql, equal, equalp) 37](#_heading=h.m2g5x4efnp9p)

[Operadores Lógicos (and, or, not) 39](#_heading=h.alc7uau5yagj)

[Secuencia de acciones (progn, prog\*, return) 40](#_heading=h.zmidivppeg2)

[Estructuras condicionales (if, cond, when, unless, case, typecase) 41](#_heading=h.yesjt54hjfuq)

[Estructuras iterativas (loop, do, dotimes, dolist) 44](#_heading=h.k9ln9sh7calu)

[Invertir una lista sin reverse 45](#_heading=h.ukl8mig5txe6)

[Funciones 47](#_heading=h.y1hvtz99cddg)

[Función lambda 48](#_heading=h.nuy0e9mghv3b)

[Ligadura de parámetros en la función 48](#_heading=h.6zr8rftt5zsj)

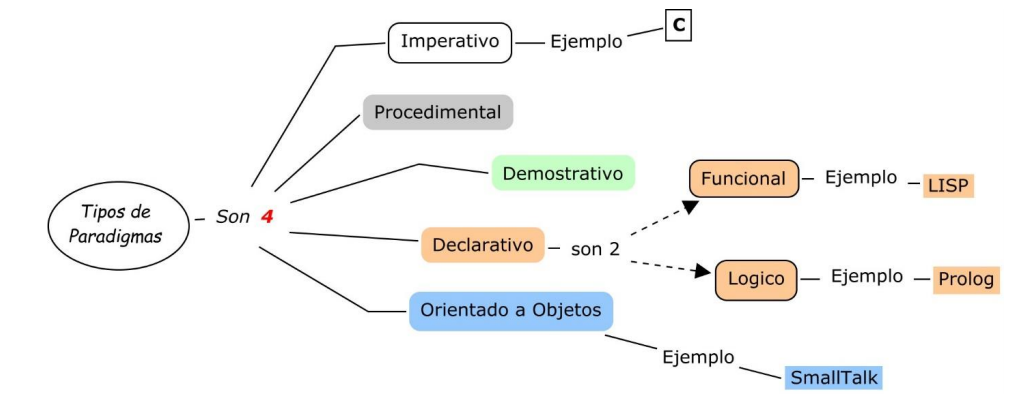
[Let 49](#_heading=h.t6d8albcy5oo)

[Declaraciones de variables 51](#_heading=h.x05cmiespo1k)

# Paradigmas y Lenguajes

Un Paradigma de Programación es una forma de representar y manipular conocimiento. Representando un enfoque para la construcción del software. En donde no es mejor uno que otro, sino que existe uno para cada tipo, con desventajas y ventajas.

## Tipos de Paradigmas



**Paradigma Imperativo**: Modelo abstracto que consiste en un gran almacenamiento de memoria donde la computadora almacena una representación codificada de un cálculo y ejecuta una secuencia de comandos que modifican el contenido de ese almacenamiento. ej: C, Basic, Pascal

**Paradigma Procedimental**: Secuencia computacional realizada etapa a etapa para resolver un problema

Su mayor dificultad reside en determinar si el valor computado es una solución correcta del problema

**Modelos de desarrollo**: Orientado a Eventos y Agentes

**Paradigma Declarativo**: Es un Modelo de desarrollo Funcional y Lógico que se construye mediante reglas y sintaxis de las propiedades derivadas del conjunto de valores que configuran la solución.

**Modelos de Desarrollo**: Se dividen en 2 Funcional y Lógico

* **Paradigma Funcional**: Modelo basado en funciones matemáticas en donde cada función devuelve un solo valor mediante una lista de parámetros (Posee la capacidad de aprender a aprender). Se enfoca en el que hacer y no el cómo hacer. **Ej.:** Lisp, Scheme
* **Paradigma Lógico**: Se basa en un subconjunto del cálculo de predicados, incluyendo instrucciones escritas llamadas cláusulas de Horn. Este paradigma puede deducir nuevos hechos a partir de otros hechos conocidos.

Representa conocimiento mediante relaciones entre objetos, estas relaciones se especifican con reglas y hechos, la ejecución consiste en la demostración de hechos sobre las relaciones por medio de preguntas.

**Ej**: Prolog

**Paradigma Demostrativo**: El programador no especifica procedimentalmente como construir una solución, sino que presentan soluciones de problemas similares.

**Modelos de desarrollo**: Genético.

**Paradigma orientado a objetos**: Disciplina de ingeniería de desarrollo y modelado de software que permite construir más fácilmente sistemas complejos a partir de componentes individuales.

Objetos + Mensajes = Programa.

**Diferencia entre imperativo y declarativo**: (pregunta de parcial)

La diferencia que existe es que el **Imperativo** especifica cada paso del conjunto de instrucciones que deberán ejecutarse. Hablamos de un Algoritmo que describe los pasos necesarios para solucionar un problema.

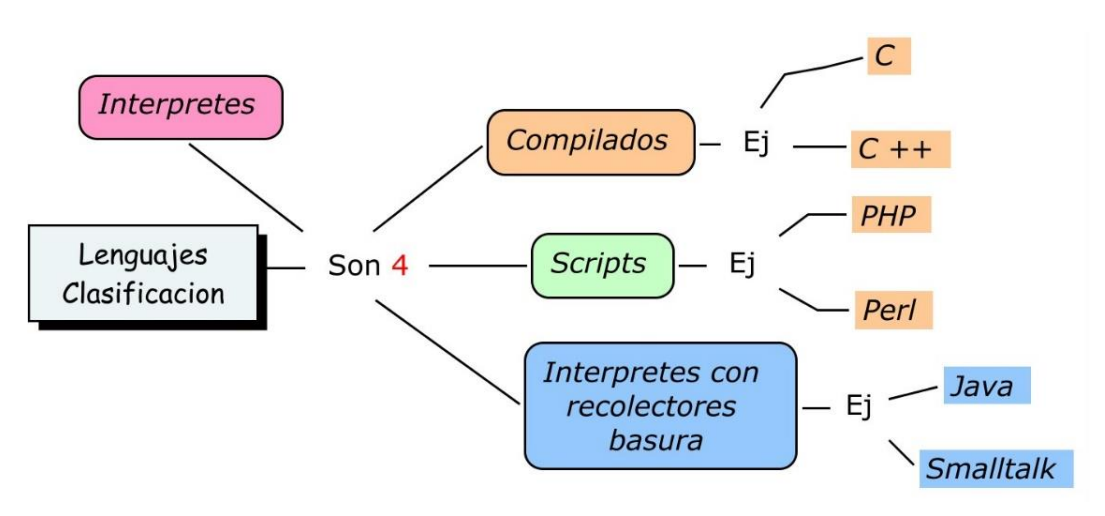
En diferencia con el **Declarativo**, Describe el problema que desea solucionar, pero sin detallar las instrucciones necesarias para gestionarlo ya que de eso se encarga los mecanismos internos.

En conclusión, el **Imperativo** se basa en que hacer y como poder resolverlo mientras que el **declarativo** solo se basa en el que hacer.

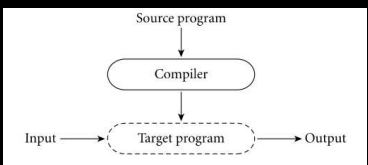
# Lenguaje de Programación

Es un conjunto de Sintaxis y Semánticas que permiten pasar por un canal de comunicación para entregar instrucciones al computador.

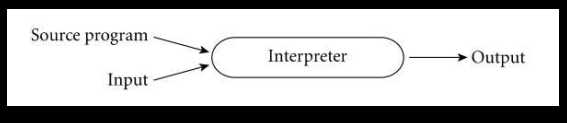
Cualquier software escrito o cualquier lenguaje de programación necesita pasar al menos por un proceso de compilación/interpretación/intermedio de manera que pueda ser traducido al lenguaje máquina.



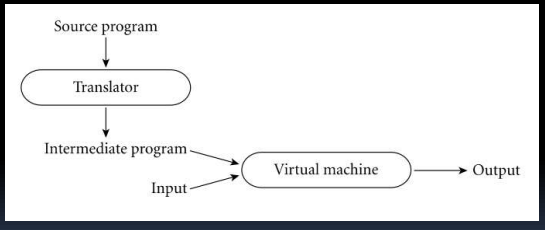
**Compilación**: Hay mayor eficiencia y se diferencia el tiempo de ejecución del tiempo de compilación.



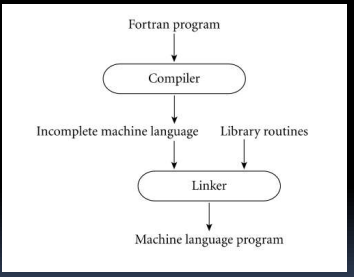
**Interpretación**: No se diferencia entre el tiempo de compilación y el tiempo de ejecución, provee mayor flexibilidad ya que el código se puede construir y ejecutar al mismo tiempo.



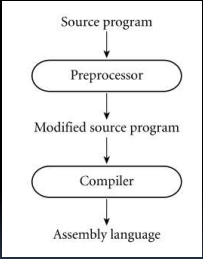
**Ejecución en máquina virtual**: Proceso de traducción y ejecución de un programa en máquina virtual.



**Enlazado de rutinas y librerías**: Proceso de compilación y enlazado de un programa.



**Preprocesamiento**: el preprocesador analiza el código y sustituye macros.



## Clasificación de Lenguajes de Programación

**Programación Imperativa**: Se basa en dar instrucciones al ordenador de cómo hacer las cosas en forma de algoritmos. **ej.:** C, Basic, Pascal.

**Programación orientada a objetos**: Basado en el imperativo, encapsula objetos que incluyen tanto variables como funciones. **ej.:** Java, C#, Python.

**Programación Dinámica**: Proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, no utiliza métodos recursivos. Se puede usar cualquier lenguaje imperativo.

**Programación dirigida por eventos**: Tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos provoquen.

**Programación declarativa**: Basado en describir el problema declarando propiedades y reglas que deben cumplirse. Hay distintos lenguajes para la programación funcional, lógica, o su combinación. **ej.:** LISP, PROLOG

**Programación funcional**: Basado en la definición de predicados, y es de corte más matemático. **ej.:** Scheme, Haskell.

**Programación Lógica**: Basado en la definición de relaciones lógicas. **ej.:**Prolog.

**Programación con restricciones**: Similar a la lógica usando ecuaciones

**Programación multiparadigma**: Es el uso de dos o más paradigmas dentro de un programa. **Ej.:** Python que es orientado a objetos, imperativo, reflexivo y funcional.

**Programación reactiva**: Se basa en la declaración de una serie de objetos emisores de eventos asíncronos y otra serie de objetos receptores de la emisión de estos eventos, los cuales reaccionan a los valores que reciben.

**Lenguaje específico del dominio**: Estos lenguajes son desarrollados para resolver un problema específico, un ejemplo sería SQL para el manejo de las bases de datos.

## Generaciones de los lenguajes

Mientras que el hardware de los ordenadores han pasado por cuatro generaciones, algo parecido sucedió con la programación de los mismos, los lenguajes suelen clasificarse en 5 generaciones, estas generaciones no coincidieron exactamente en el tiempo con las del hardware.

**Primera generación**: Estos ordenadores se programaban directamente en código binario, cada modelo de ordenador suele tener su propio código, por eso se llama código de máquina.

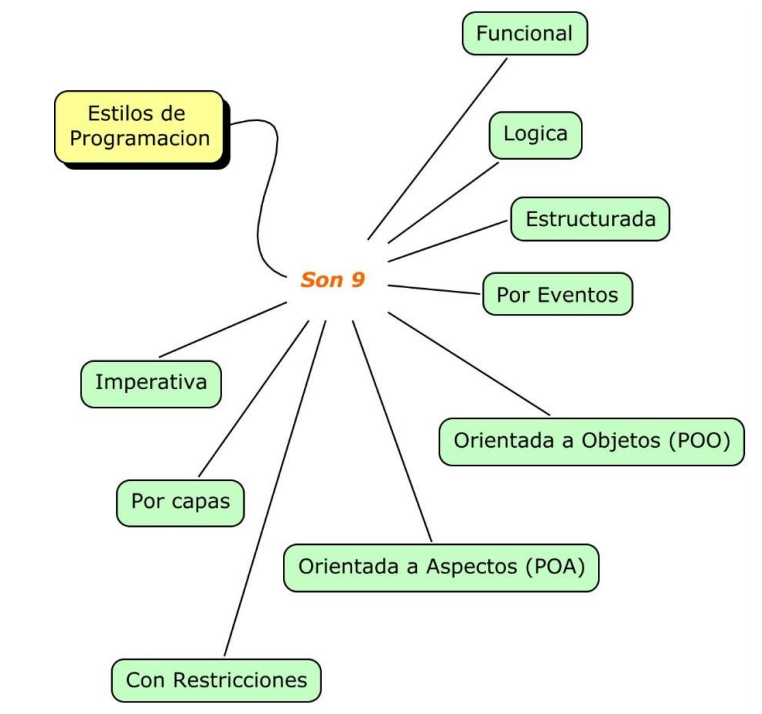
**Segunda generación**: Lenguajes simbólicos, simplifican la escritura de las instrucciones y las hacen más legibles.

**Tercera generación**: Los lenguajes de alto nivel, los cuales sustituyen a las instrucciones simbólicas por códigos independientes de la máquina, son parecidos al lenguaje humano o al de las matemáticas.

**Cuarta generación**: Son aquellas herramientas que permiten construir aplicaciones sencillas combinando piezas prefabricadas, algunas personas proponen reservar el nombre de cuarta generación para la programación orientada a objetos.

**Quinta generación**: Se llama así a los lenguajes de la inteligencia artificial, pero este nombre cayó en desuso con el fracaso del proyecto japonés.

## Estilos de programación



**Imperativa**: Es un paradigma que describe la programación en términos del estado del programa y sentencias que cambian este estado, son un conjunto de instrucciones que le indican al computador cómo realizar una tarea.

La mayoría del hardware está implementado en programación imperativa, ya que el código máquina el cual es ejecutado por estos está escrito de forma imperativa. EJ: Pascal, C.

*Características*: Definición de procedimientos, datos, chequeo de tipos durante la compilación, cambio de estado de variables, pasos de ejecución de un proceso.

**Estructurada**: Escribe una programación de forma clara, utilizando estructuras secuenciales, selectivas e iterativas, siendo innecesario y no permitiéndose el uso de instrucciones de transferencia incondicional.

Según Dijkstra un programa puede escribirse con las 3 instrucciones previamente mencionadas.

**Funcional**: Está basado en la utilización de funciones matemáticas, proviniendo su origen del cálculo Lambda, el cual su teoría fue elaborado por Alonzo Church.

### Categorías de lenguajes funcionales: híbridos y puros

**LA DIFERENCIA ES QUE LOS LENGUAJES HÍBRIDOS SON MENOS DOGMÁTICOS QUE LOS PUROS** (Pregunta de parcial), siendo los híbridos los que admiten conceptos de los lenguajes imperativos (secuencias de instrucciones, asignación de variables).

*Puros:* Haskell, Miranda.

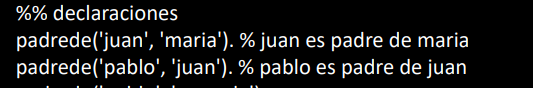
*Híbridos:* Lisp, Scheme.

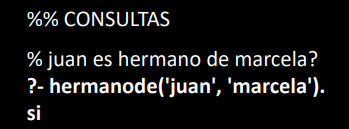
**Lógica**: Consiste en la aplicación del corpus de conocimiento sobre lógica para el diseño de lenguajes de programación, gira en torno al concepto de predicado o relación entre elementos. Su hábitat natural es en aplicaciones de inteligencia artificial o relacionadas:

* **Sistemas expertos**: Un sistema de información que imita las recomendaciones de un experto sobre algún dominio de conocimiento.
* **Demostración automática de teoremas**: Donde un programa genera nuevos teoremas sobre una teoría existente.
* **Reconocimiento de lenguaje natural**: Un programa que es capaz de comprender la información contenida en una expresión lingüística humana.

Los programas que están escritos en lenguaje lógico usan expresiones lógicas, con valor verdadero o falso.

Las expresiones utilizadas en prolog son las cláusulas de Horn





**Orientada a objetos**: Define a los programas en términos de clases de objetos, los cuales son entidades que combinan estado, comportamiento e identidad, esta expresa un programa como un conjunto de objetos que colaboran entre sí para realizar tareas, logrando así módulos más fácil de escribir, mantener y reutilizar.

Un objeto contiene la información que permite definirlo e identificarlo frente a otros objetos.

Dispone de mecanismos de interacción que favorecen la comunicación entre objetos, esto lleva a tratarlos como unidades indivisibles en las que no se separan la información y el procesamiento.

**Diferencias entre estructurada y orientada a objetos**:

* Poo es más moderna.
* En la estructurada se escriben funciones y luego se pasan datos mientras que en Poo se definen objetos con datos y métodos para luego enviar mensajes a los objetos que utilicen dichos métodos en sí mismos.
* Poo se basa en lenguajes que soportan sintáctica y semánticamente la unión entre TAD y sus operaciones.
* Poo incorpora en su entorno de ejecución mecanismos como polimorfismo y envío de mensajes.

**Orientada a aspectos**: Su filosofía se basa en tratar las obligaciones transversales de nuestros programas como módulos separados para lograr una correcta separación de responsabilidades.

*Obligación transversal:* es aquella que se repite en varias partes de un programa independientemente de si las secciones en la que aparece tienen relación directa.

Ej: Un método que actualice lo que se ve en pantalla, se puede llamar desde un método que dibuje algo en pantalla, hasta métodos para recortar.

*Influencias para construir bases sólidas:* reflexión computacional, programación adaptativa, Poo.

Esta toma un enfoque parecido a POO pero con POA nos dice que si alguna funcionalidad de nuestro código se repite en diferentes módulos, lo mejor seria extraerla del programa principal y hacerla un aspecto.

*Ventajas:*

* Provee una fuerte herramienta para modularizar programas.
* Vuelve limpio el código fuente.
* Agiliza el proceso de creación de programas cuando muchas personas están involucradas en el mismo proyecto.
* Se puede mezclar con otros paradigmas
* Permite la comunicación con distintos lenguajes de programación.

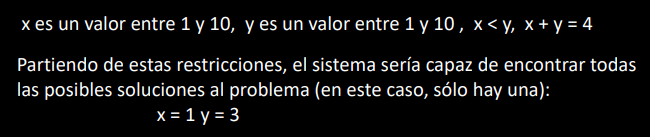
*Desventajas:*

* Sufre de un antipatrón de diseño: acciones a distancia.
* Es difícil de comprender el código ya que el programa hace tareas que no están en los métodos que deberían estar.
* Es un poco complicado identificar cuándo es óptimo utilizar.

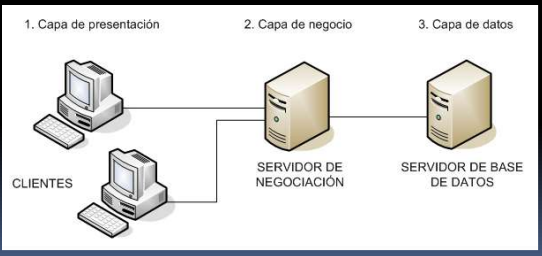
**Programación con restricciones**: Es un paradigma donde las relaciones entre variables son expresadas en términos de restricciones, las cuales deben ser satisfechas por cualquier solución del problema planteado. Es utilizada para la descripción y resolución de problemas combinatorios, especialmente en áreas de planificación y programación de tareas.

Pertenece al paradigma de programación declarativa, nos limitamos a decir qué queremos que pase y el sistema se encargará de hacerlo.

Crea un modelo formado por variables, a las cuales se le debe indicar qué posibles valores pueden tomar y restricciones que expresan relaciones que deben cumplirse entre esas variables.



**Programación por capas**: El objetivo primordial es la separación de la lógica de negocios de la lógica de diseño.



Su principal ventaja es que el desarrollo se puede llevar a cabo en varios niveles, si se debe realizar algún cambio se ataca al nivel requerido sin tener que revisar entre código mezclado.

# Programación Funcional: Lisp

Es un lenguaje que implementa el modelo de funciones recursivas, las cuales proporcionan una definición sintáctica y semánticamente clara.

Las listas son las bases de los programas y de los datos, siendo estas equivalentes. Lisp es un acrónimo de List Processing.

Proporciona un conjunto potente de funciones que manipulan listas, implementadas internamente como punteros.

Los principales fundamentos de control son la **recursión** y las **condicionales simples**.

Todo programa LISP consiste de una función.

No posee asignaciones.

La principal estructura de datos es la Lista.

## Características

**Portabilidad**: Excluye características que no puedan implementarse en la mayoría de las máquinas.

Se diseñó para que fuera fácil construir programas lo menos dependiente posible de las máquinas.

**Consistencia**: muchas implementaciones LISP son internamente inconsistentes en el sentido de que el intérprete y el compilador pueden asignar distintas semánticas al mismo programa.

Esta diferencia radica en el hecho de que el intérprete considera que todas las variables tienen alcance dinámico.

**Expresividad**: Recoge las construcciones más útiles de dialectos Lisp anteriores.

**Eficiencia**: Tiene muchas características diseñadas para facilitar la producción de código compilado de alta calidad.

**Potencia**: Suministradas por paquetes que corren sobre el núcleo.

**Estabilidad**: el núcleo cambia lentamente sólo cuando los expertos encargados del estándar lo decidan luego de examinar las nuevas características.

## Tipos de valores en LISP

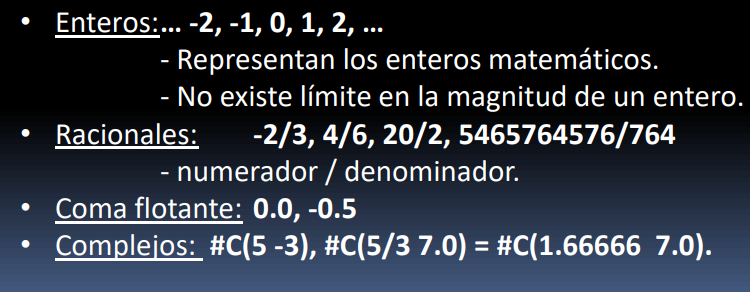
**Atom**: Número, carácter, símbolo, String.

**Cons**: Lista

Pero también están las estructuras de datos definidas por el usuario y los tipos de datos propios del usuario. En LISP no es necesario declarar el tipo de dato.

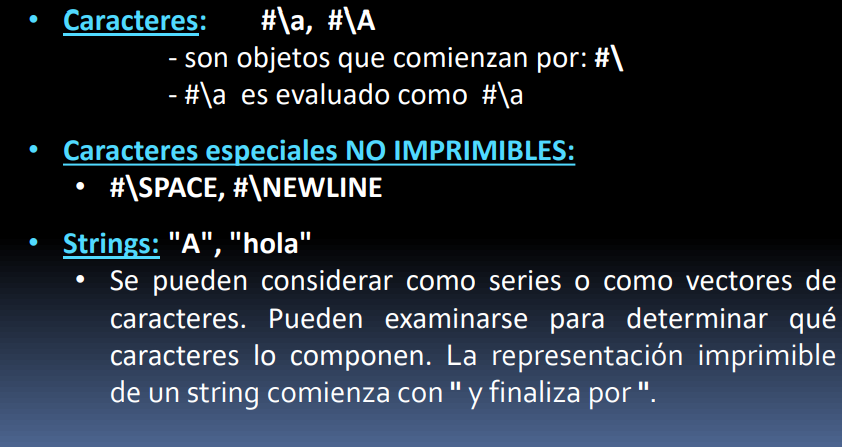
### **Atom**

Estructuras básicas, elementos indivisibles con significado propio. Un átomo se apunta a si mismo



#### Caracteres y Strings

Dos tipos de datos utilizados normalmente para manipular texto



#### **Átomos simbólicos**

Representan algo más, como el nombre de una función o el valor de una variable.

Los nombres de variables: deben ser un símbolo atómico. El nombre imprimible se usa como nombre de la variable y el tipo de valor asociado puede ser un átomo o una lista.

La evaluación de una variable es más complicada que la de los átomos, pues LISP intentará devolver el valor de la variable.

Los nombres de los símbolos son case-insensitive, siendo así que pueden tomar cualquier secuencia de caracteres alfanuméricos a excepción de: () . ‘ “ ” ; .

#### **Símbolos reservados**

Apuntan a ellas mismas: T(true) y NIL (false).

### **Lista**

La forma más fácil y versátil de agrupar objetos.

Es una estructura básica de datos y de un programa

Está delimitada por paréntesis

Está formada por una secuencia de átomos/listas o combinaciones de estas.

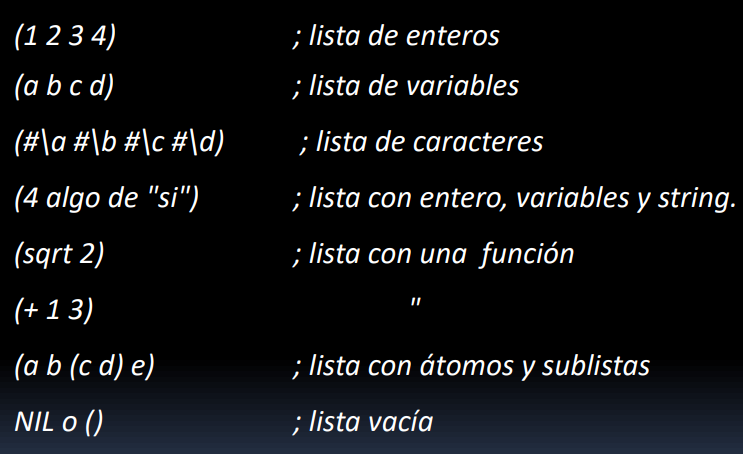
Contiene dos componentes llamados el **CAR** y el **CDR**

Una lista se define recursivamente ya sea como la lista vacía o como un **CONS** cuyo componente **CDR** es una lista.

Los componentes **CAR** de los conses son los elementos de la lista.

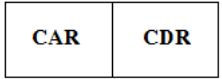
Para cada elemento de la lista hay un **CONS**.





#### CONS

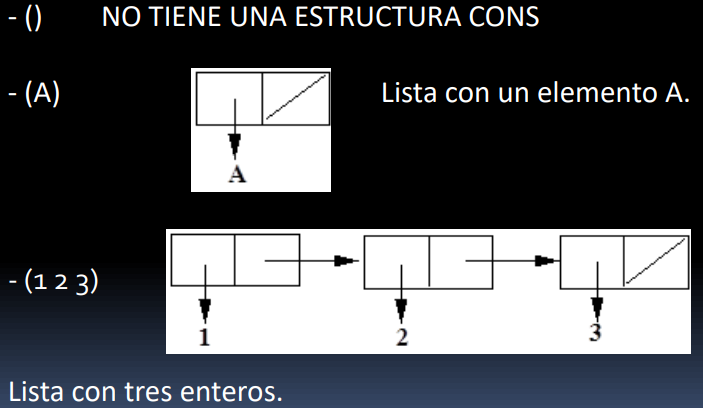
Estructura de información que contiene dos componentes, el CAR y el CDR.

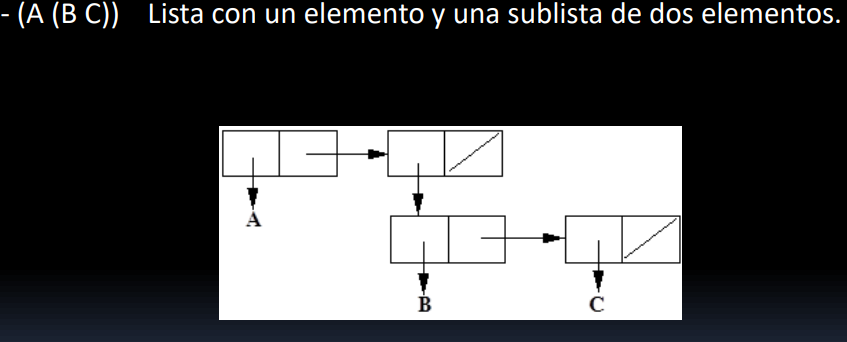


CAR es la primera parte de la estructura y contiene un apuntador al primer elemento de la lista.

El CDR es la segunda parte y contiene un apuntador a la siguiente estructura CONS que contiene los siguientes elementos de la lista

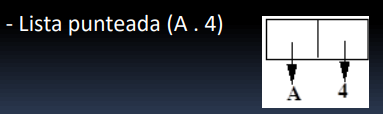
Si no hay más elementos un apuntador a NIL





#### **Listas punteadas**

Ocurre cuando el CDR apunta a un átomo en lugar de a una estructura CONS o NIL



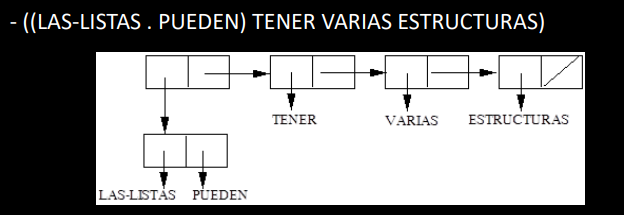
*Ventajas:*

Ahorro de una casilla CONS.

*Desventajas:*

Menos flexibilidad ya que el tratamiento de una lista no podrá depender de la marca de fin de lista.

No se permiten modificaciones de la lista para añadir nuevos CONS (el último elemento no apunta a NIL)



### Lisp Listener

(pregunta de parcial)

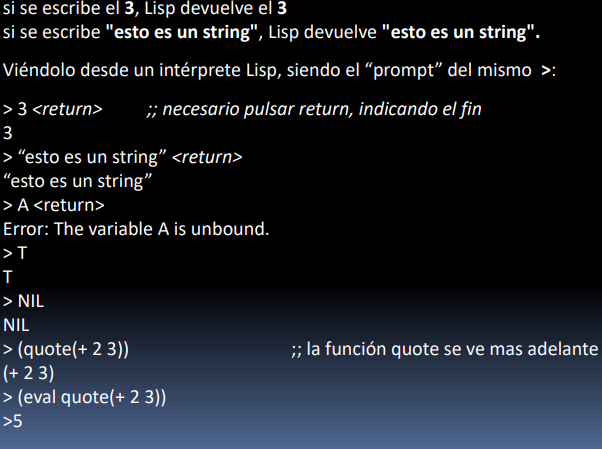
Es el ciclo evaluador que realiza Lisp para ejecutar formas y programas. Cuando recibe una entrada, evalúa la expresión, espera al return, si la forma es atómica o espera a paréntesis de cierre si es una lista.

Finalmente imprime el resultado y el ciclo comienza de nuevo. Este ciclo se llama *READ-EVAL-PRINT*

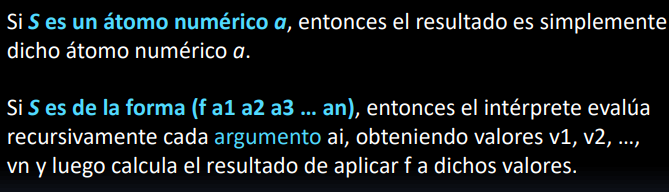
En Lisp: *(PRINT (EVAL (READ))).*

### Funcion Eval

Lisp siempre tratará de evaluar todo a través de la función eval; **(eval form)** donde se evalúa la forma y se devuelve el valor.

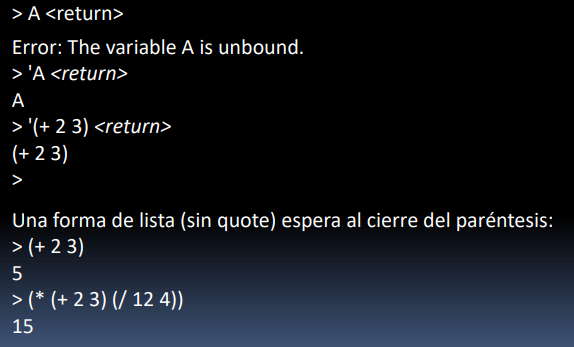


El intérprete de LISP evalúa una expresión S



## Funciones varias

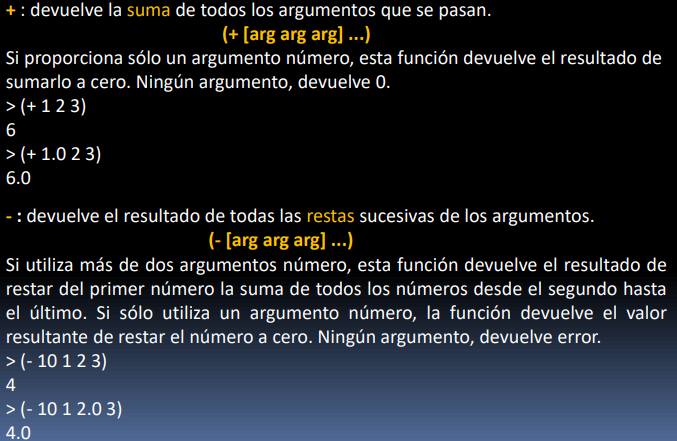
**Función QUOTE**: Provoca que el objeto no sea evaluado, logrando así que una lista pueda ser pasado como datos.

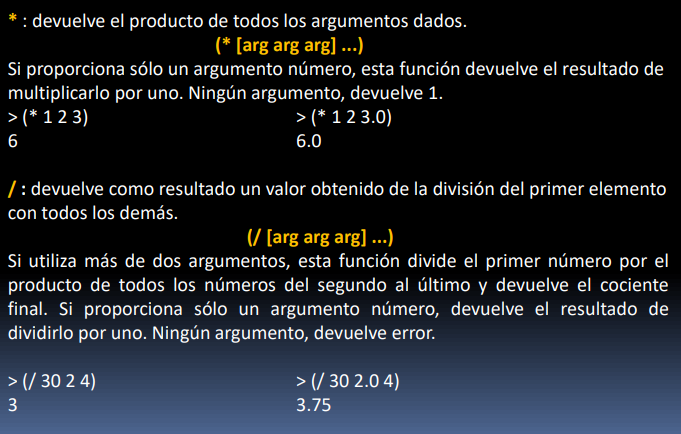


Cuando se desea realizar funciones aritméticas, requiere que sus argumentos sean números, en caso contrario provocaría un error.

**Características**: Si algún argumento no es entero, devolver un resultado decimal, ninguna de las funciones modifica el valor de los argumentos, solo devuelven el resultado.

### Funciones Aritméticas



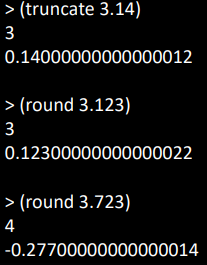


#### Truncate y Round

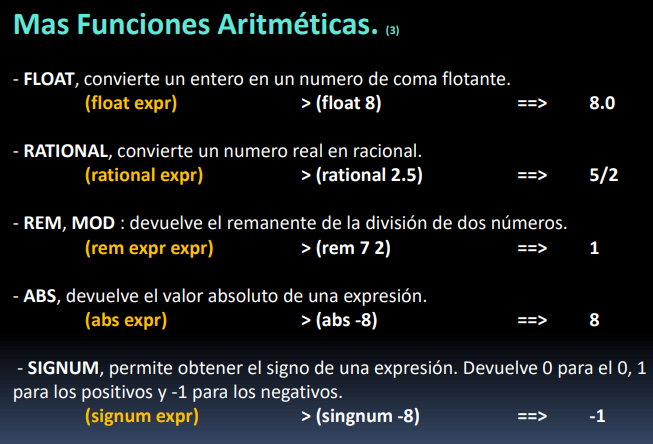
**Truncate**: Trunca una expresión tendiendo a 0. (truncate <expr>)

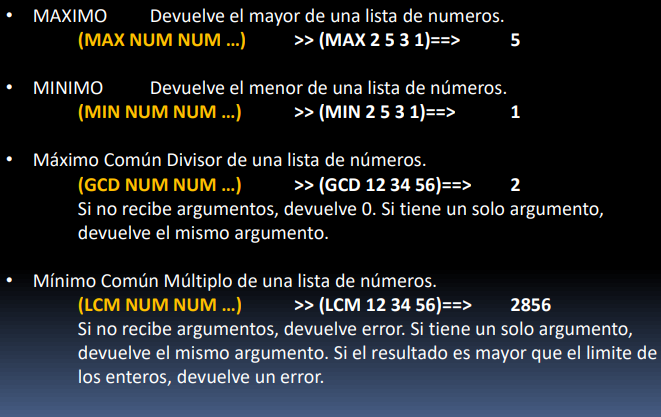
**Round**: Redondea hacia el entero positivo más cercano. (round <expr>)

Ambas devuelven 2 resultados, el resultado de la operación y su remanente

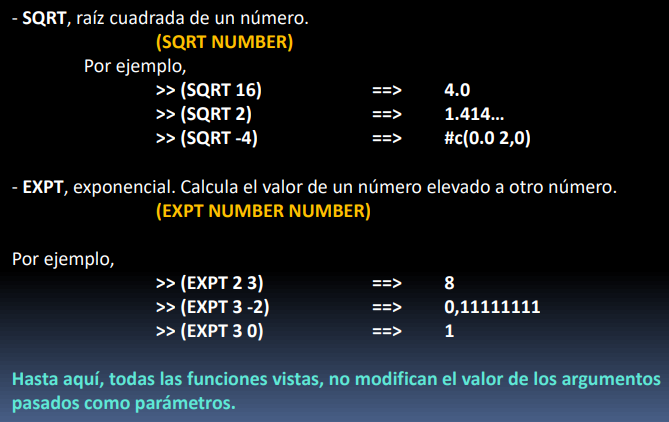
****

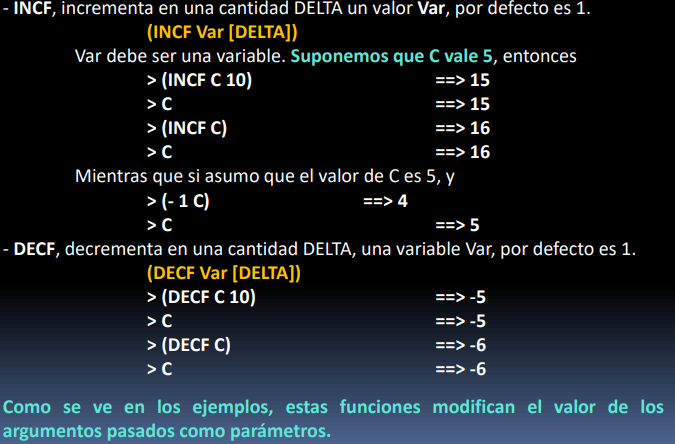
### Más funciones aritméticas (float, rational, mod, abs, signum, max, min, MCD, MCM)

****

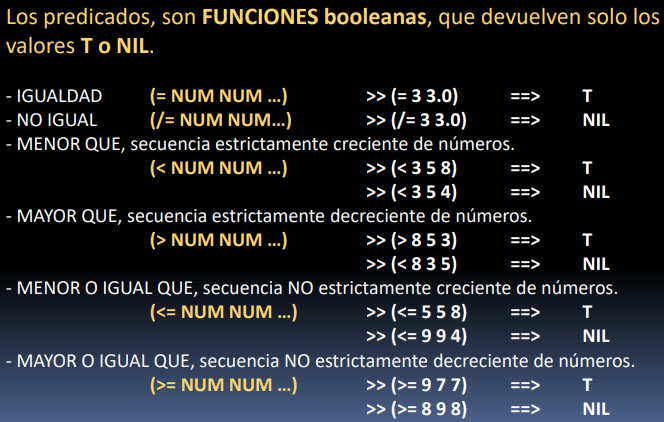
****

### Funciones Matemáticas (sqrt, expt, incf, decf)

****

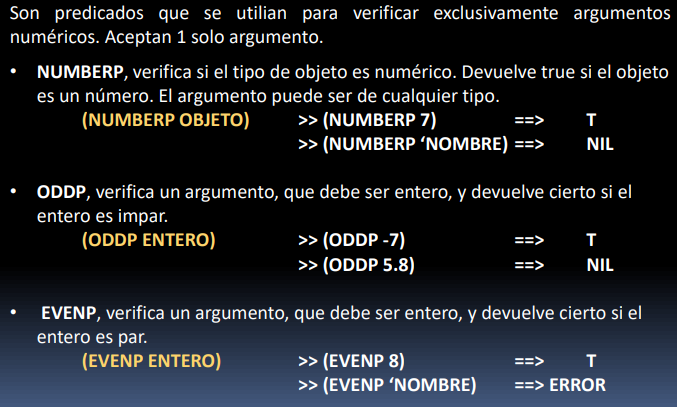
****

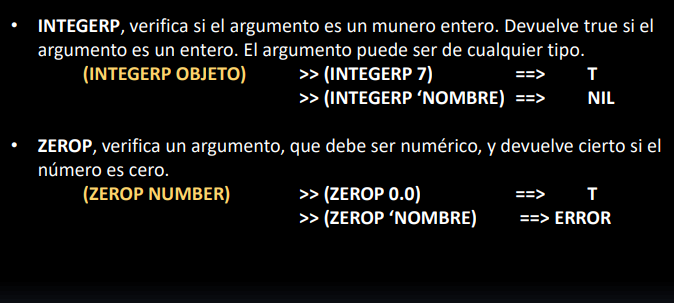
### Predicados de comparación

****

### 

### Predicados numéricos (numberp, oddp, evenp, integerp, zerop)

****

****

### Funciones para operar sobre listas (car, cdr, c\*r, last, elt)

## 

## 

## 

## 

## 

### 

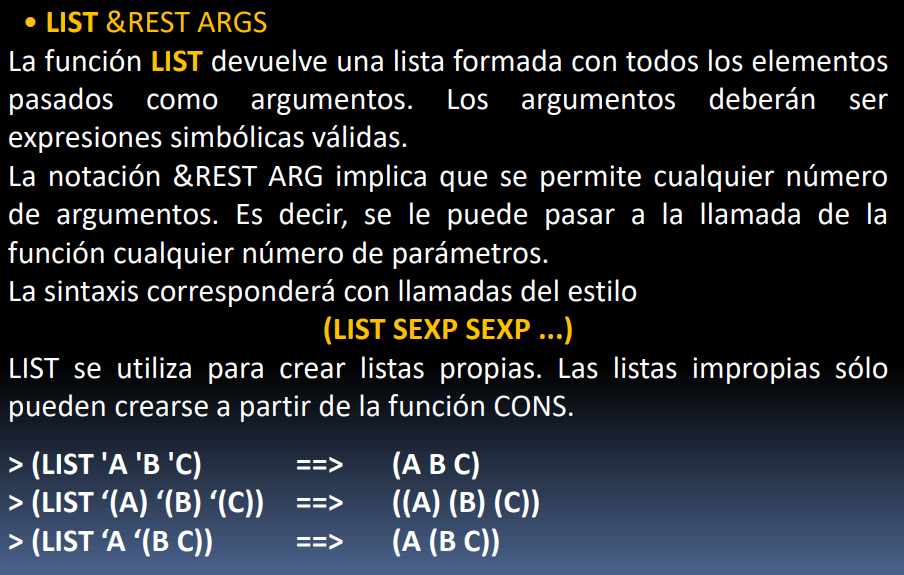
### Función de asignación (setq)

### 

### Función de construcción de listas (cons, list, append)

### 

### 



### 



### 

### Otras funciones de listas (butlast, nth, nthcdr, reverse)

### 

### 

### 

### Funciones Destructivas (rplaca, rplacd, push, pop)

### 

### 

### 

### 

### 

### 

### Otras Funciones (length, member)

### 

### 

### 

### Funciones de ligadura de datos (fmakunbound, makunbound, boundp)

### 

### 

### 

### 

### 

### Predicados sobre tipos de datos (atom, listp, null, typep)

### 

### 

### Predicados de igualdad (eq, eql, equal, equalp)

### 

### 

### 

### 

### 

### Operadores Lógicos (and, or, not)

### 

### 

### 

### 

### 

### Secuencia de acciones (progn, prog\*, return)

### 

### 

### Estructuras condicionales (if, cond, when, unless, case, typecase)

### 

### 

### 

### 

### 

### 

### Estructuras iterativas (loop, do, dotimes, dolist)

### 

### 

### 

#### 

#### Invertir una lista sin reverse

#### 

#### 

#### 

#### 

## Funciones

Objeto lisp que puede invocarse como un procedimiento, puede tomar argumentos y devolver uno o más valores.

En lisp todos los programas se escriben como colecciones de funciones y procedimientos.

La llamada a la función es una lista cuyo primer elemento es el nombre de la función y el resto formarán los parámetros de dicha función.

Lisp comprobará si el primer elemento de la lista se corresponde con alguna función predefinida del usuario, los argumentos de la llamada a la función serán evaluados antes de pasarlos como información a la función.

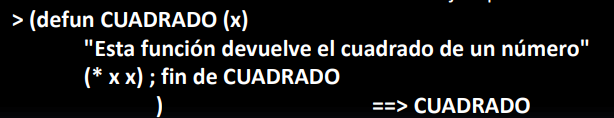
Si el argumento de la función es a su vez otra llamada a función se evalúa primero la función.

Entonces se evalúa primero la función del argumento y el resultado de la misma se pasa a la función principal.

Defun permite al usuario definir sus propias funciones.

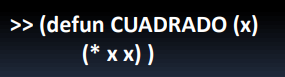
**(DEFUN nombre-función lambda-lista {declaration: string-doc} cuerpo)**

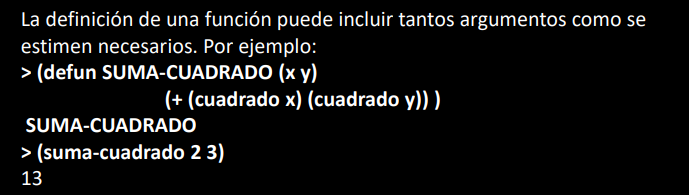
Donde el nombre de la función debe ser un símbolo atómico, lambda-lista es el conjunto de argumentos , el cuerpo es cualquier forma a evaluarse en secuencia cuando se invoca. El parámetro L de la definición se corresponde con una variable local, la cual es accesible solo desde el interior de la función.



Lisp crea un símbolo “CUADRADO”, incorpora en la definición de función el valor correspondiente a dicha función.

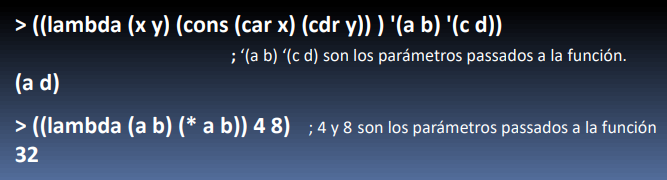
La evaluación de una función cuando esta se define consiste en el nombre de la propia función.





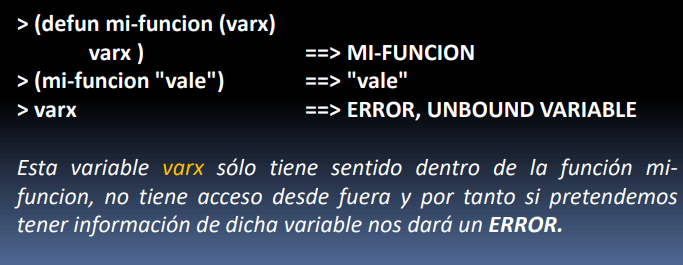
### Función lambda

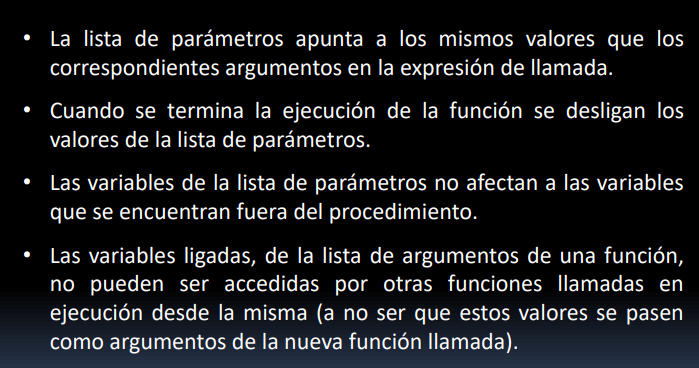
tiene una estructura semejante a las de las funciones, la diferencia a las funciones es que los a1…an ya son los parámetros pasados a la función.



### Ligadura de parámetros en la función

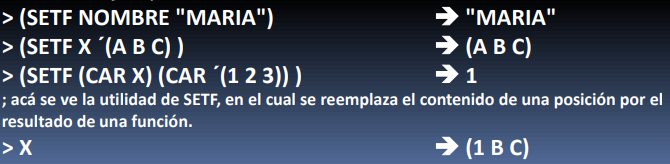
Las variables creadas en la lista de argumentos de un defun son normalmente locales para la función, la lista de parámetros organiza un espacio en memoria para el ámbito de definición de la función.





**Setf**: Al igual que setq es utilizada para asignación de valores a las variables

Setf se utiliza para una localización o dirección en la que hay que realizar la asignación. Realiza las evaluaciones y asignaciones secuencialmente, devolviendo el valor de la última evaluación realizada



#### Let

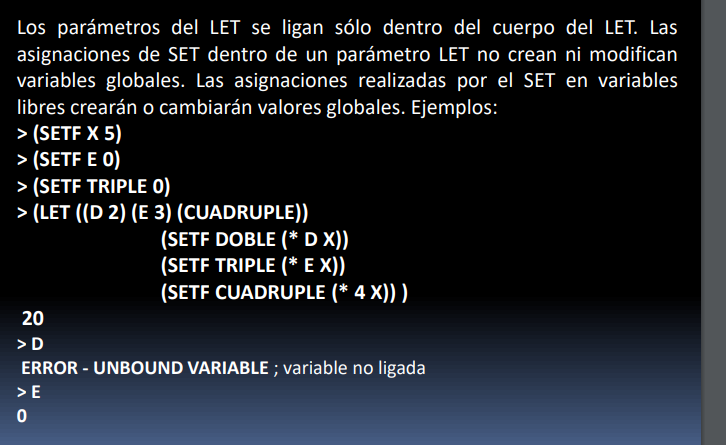
Permite crear variables locales interiores a una función, consta de 2 partes, asignando variables y cuerpo.



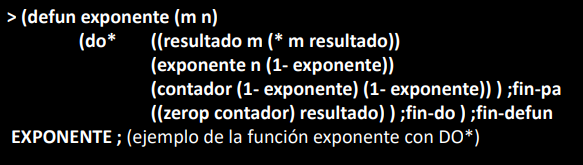
La asignación de variables es una lista de listas donde cada lista consta de un nombre de variable y una forma a evaluar, con let las formas se evalúan asignando paralelamente valores a todas las variables.

Para realizar asignación secuencialmente se utiliza Let\*

El cuerpo de let es como un cuerpo de defun, cualquier valor previo que tuvieran variables con el mismo nombre se guardarán y se restaurarán al finalizar el let.



**Do\***: Asigna variables de forma paralela, Do\* asigna valores secuencialmente



### Declaraciones de variables

Las variables a menos que sean declaradas de otra forma, son de **ámbito léxico (locales)**.

Las variables pueden declararse como especiales y en estos casos se dice que son de **ámbito dinámico (globales)**, para declarar variables especiales para todas las funciones, sin necesidad de declararlas separadamente en cada una de ellas usamos:

Esta es la idea de variable global utilizada en otros lenguajes de programación

