

Resumen Paradigma de Programación.

¿Qué es un paradigma de programación? --> Los paradigmas proveen conjuntos de herramientas conceptuales que nos permiten analizar, representar y abordar los problemas, con lo cual estos son las formas, conceptos, reglas, etc de abordar un problema.

Tipos de paradigmas:

- **Programación imperativa** → Leguajes como: Pascal, C, Ada, etc.
En este paradigma se puede tener efectos laterales.
- **Programación orientada a objetos** → tiene como objetivo mejorar la complejidad de los problemas del mundo real, abstrayendo su conocimiento y encapsulándolo en objetos. Describe un sistema en términos de los objetos involucrados, que interactúan entre si enviándose mensajes para llevar a cabo la tarea.

Definiciones:

Objeto: Entidad que tiene dos características: estado y comportamiento.

Los objetos interactúan entre si enviándose mensajes. El **emisor** es quien envía el mensaje junto con los argumentos necesarios, y el **receptor** es el objeto al que se le envía el mensaje y en respuesta ejecutara un **método** que es la implementación de un mensaje.

Estado: Esta representado por los atributos, es decir las propiedades relevantes de un objeto. Y estos se almacenan en variables de instancia.

Comportamiento: Esta representado por una serie de funciones o métodos que modifican o no el estado del objeto.

Clase: Es un modelo para definir objetos que pueden realizar las mismas acciones y poseen características similares.

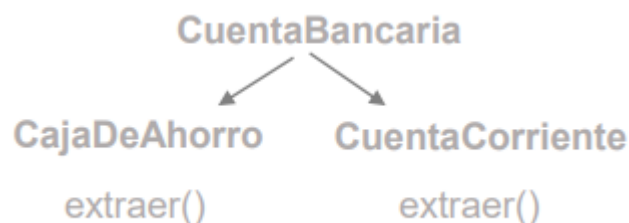
Instancia: Es un objeto en particular de una clase.

Características de la Programación orientada a objetos:

1. **Encapsulamiento:** Una clase encapsula la representación privada y el comportamiento de un objeto, es decir, que un determinado objeto no conoce el funcionamiento de los demás objetos y no lo necesita.
2. **Ocultamiento de la Información:** Solo se puede modificar la estructura interna de un objeto desde fuera de la clase, es decir, a través de los mensajes que entiende el objeto.
3. **Herencia:** Es la propiedad de crear nuevas clases a partir de otras ya existentes, ampliando su estructura y comportamiento. Esta herencia puede ser:
 - **De estado:** Se heredan solo los atributos de la superclase
 - **De comportamiento:** Se heredan solo los métodos de la superclase
 - **Total, o Parcial:** cuando se heredan todos los atributos y métodos es total, y sino parcial.
 - **Simple o Múltiple:** simple cuando se hereda de una solo superclase y múltiple es cuando se hereda de varias superclases.

Cuando se programa utilizando herencia, se **programa por extensión**, es decir, que se extiende el código existente (se rehúsa código). En estos casos cuando se envía un mensaje, el S.O. lo busca en la clase de dicho objeto, y si no lo encuentra porque es heredado, sube en por el árbol jerárquico de clases aplicando una **búsqueda LOOK-UP**.

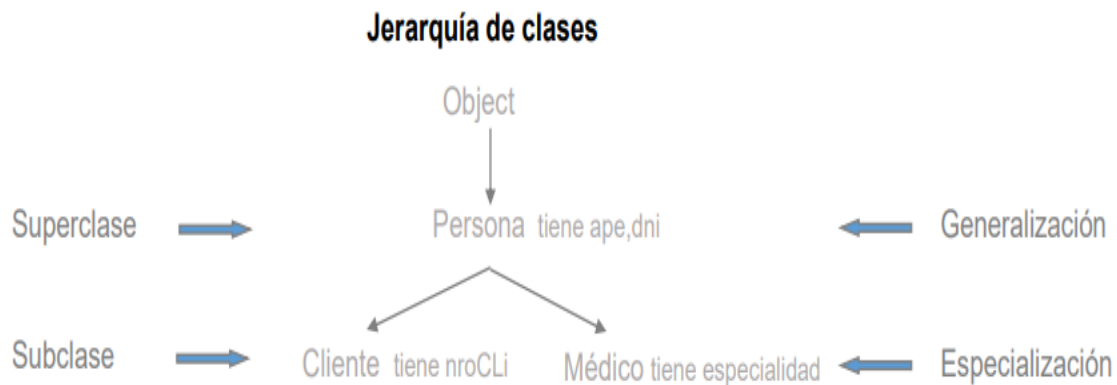
4. **Polimorfismo:** Es la capacidad que tienen objetos diferentes de entender y responder a un mismo mensaje de manera diferente. La interpretación de un mensaje es determinada por el receptor.



5. **Binding dinámico:** significa que la ligadura entre el objeto receptor y el método que se va a ejecutar se realiza en tiempo de ejecución. Y en este paradigma también hay binding dinámico entre la clase del objeto y la variable que apunta a dicho objeto.
6. Por lo general los lenguajes orientados a objetos **no son tipados**.

Jerarquía de clases:

Las clases están organizadas en una **jerarquía de clases**, donde las subclases heredan estado y comportamiento de las superclases. Además, estas subclases pueden agregar nuevos atributos y comportamientos.



Lenguajes de Programación:

- Smalltalk →

Características:

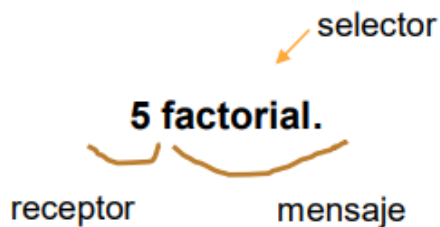
1. Es un lenguaje de programación reflexivo y con tipado dinámico.
2. Considerado e los primeros lenguajes orientados objetos, cuando en realidad, el primero fue **Simula**.
3. Es un **lenguaje orientado a objetos puro**, es decir, que todas las entidades que maneja son objetos. Y es un ambiente completo de desarrollo de programas.
4. En un programa POO, las clases abstractas no pueden tener instancia en cambio las concretas si.

5. La **herencia de estructura** es total y la **herencia de comportamiento** es parcial.

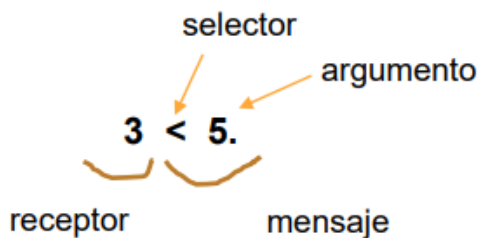
Mensajes:

Tienen tres **componentes principales**: receptor, selector y argumentos. Y existen tres tipos de mensajes:

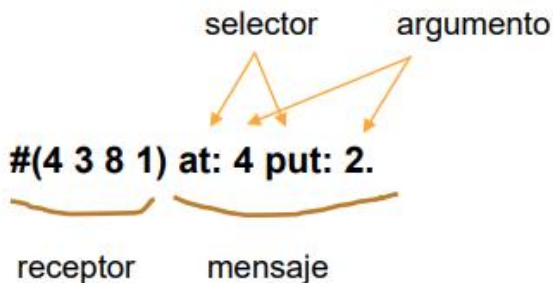
- **Unarios**: no tienen argumentos.



- **Binarios**: tienen solo un argumento. Y se utilizan para operaciones lógico, matemáticas.



- **Palabra clave**: mensajes que contiene una o más palabras clave, y estas tienen un argumento asociado. Se la reconoce por los dos puntos (:)



Orden de ejecución de los mensajes: Los mensajes en Smalltalk se ejecutan en el siguiente orden:

1°) las expresiones que están entre paréntesis ().

2°) las expresiones unarias.

3°) las expresiones binarias.

4°) las expresiones de palabra clave.

Todas de izquierda a derecha.

Mensajes que entienden todos los objetos

<code>unaClase new</code>	(crea una instancia vacía)
<code>unObjeto class</code>	(me devuelve la clase del objeto receptor)
<code>unaClase superClass</code>	(me devuelve la superclase de una clase)
<code>unObjeto isKindOf: aClass</code>	(me devuelve V o F si unObjeto pertenece a la clase aClass o no)
<code>unObjeto isNil</code>	(devuelve true o false si es nil o no)
<code>unObjeto notNil</code>	
<code>unObjeto yourself</code>	(devuelve el objeto receptor del mensaje)
<code>unObjeto inspect</code>	(abre una ventana especial que me permite inspeccionar al objeto receptor para ver su estado y/o modificarlo)

Mensajes para ingresar datos por pantalla.

<code>variable:= Prompter prompt: ' título del mensaje de ingreso de datos'.</code>	//ingresa un string siempre
<code>resp:=MessageBox confirm: 'desea continuar?'.</code>	//ingresa um booleano
<code>x:= (Prompter prompt: 'ingrese valor') asNumber.</code>	//convierte a número entero o real

Mensajes para imprimir en pantalla.

MessageBox **notify**: 'El nombre y apellido del alumno es:', nomyape. //la coma concatena

MessageBox **notify**: 'la distancia entre los puntos es:', (x **displayString**). //displayString convierte a string

suma **inspect**. //abre una ventana de inspección y muestra el contenido del OR

Variables:

Las variables en Smalltalk representan un puntero a un objeto. Y su **asignación** es al siguiente: (**x:=expresión**).

Tipos de variables:

- **De instancia:** Son propias de cada objeto, representan el Estado interno del mismo.
- **De clase:** Son comunes a todos los objetos de la misma clase, es decir, que tienen el mismo valor para todos los objetos de una misma clase.
- **Temporales:** Son locales a un método o aplicación.
- **Argumentos:** Son los parámetros de los métodos.

Estructuras de control:

En Smalltalk **no existen estructuras de control**, sino que **se simulan**. Estas están implementadas en términos de objetos y mensajes. **Casos particulares:**

a) Selección condicional:

(expresión booleana) ifTrue:[TrueBlock] ifFalse:[FalseBlock].

(expresión booleana) ifTrue:[TrueBlock].

(expresión booleana) ifFalse:[FalseBlock].

b) Repetición condicional:

[expresión booleana] whileTrue:[cuerpo del loop].

[expresión booleana] whileFalse:[cuerpo del loop].

c) Repetición de longitud fija:

valorInicial to: valorFinal do: [variable del loop | cuerpo del loop]

Colecciones:

Una colección es un objeto que recopila y organiza otros objetos.

Tipos de colecciones:

- **Bag:** son colecciones no ordenadas que puede contener elementos repetidos.
- **Set:** son colecciones no ordenadas que no permiten elementos repetidos.
- La **clase Interval** representa rangos de números.
- **OrderedCollection**
- **Array**
- **SortedCollection:** Es una colección cuyos elementos por defecto serán ordenados de menor a mayor solo en caso de clases simples como números o strings.
- **Dictionary:** un diccionario Es una estructura de dato para almacenar cualquier tipo de información.

```
d:=Dictionary new at:'lunes' put:'Física';  
                  at:'martes' put:'Computación';  
                  at:'miércoles' put:'Matemática';  
                  at:'jueves' put:'Computación';  
                  at:'viernes' put:'Matemática';  
                  yourself.
```

Operaciones sobre colecciones:

Para insertar	Para recuperar	Para eliminar
add:unObjeto	first	removeFirst
addFirst:	last	removeLast
addLast:	at:	remove:
add: before:		remove: ifAbsent:
add: after:		
at: put:		

Testeos y Conversión de colecciones

Testeos de colecciones:

size

isEmpty

includes:unObjeto

occurrencesOf: un Objeto

Conversión de colecciones:

asSet

asBag

asArray

asOrderedCollection

asSortedCollection

Enumeradores de colecciones

- I. **do:** itera sobre cada elemento de la colección. Ejemplo: col do: [:pieza | pieza reset].
- II. **to: do:** repite un número conocido de veces un proceso. Ejemplo: 1 to:(col size) do:[i | suma:=suma + col at:i].
- III. **select** col1:= alumnos select:[:alu | alu verNota=8]. Retorna en otra colección los alumnos con nota igual a 8
- IV. **reject** col2:= alumnos reject:[:alu | alu verNota=8]. Retorna en otra colección los alumnos con nota distinto a 8
- V. **collect** col3:= alumnos collect:[:alu | alu verNombre]. Retorna en otra colección los nombres de los alumnos
- VI. **detect** a:= alumnos detect:[:alu | alu verNota=8] ifNone:[nil]. Retorna el primer alumno con nota igual a 8 o nil si no hay ninguno
- VII. **a modNota:10.** “modifica la nota de ese alumno encontrado”

Mecanismo de herencia en POO:

Generalización: se agrupan clases creando superclases que tengan características comunes. Es decir, se abstraen características comunes a dos o más clases formando una clase más abstracta.

Especialización: se agrega un nuevo nivel a la jerarquía, es decir se agrega comportamiento a la jerarquía creando nuevas subclases.

Herencia: La herencia es el mecanismo mediante el cual se pueden definir nuevas clases basadas en otras ya existentes, a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación.

Para programar usando herencia se utilizan las **pseudo variables** self y super:

- **self:** hace referencia al objeto receptor del mensaje.
- **super:** hace referencia a la superclase inmediata de la clase que contiene el método en el cual aparece dicha pseudovariable.

- **C++→**

Características:

- Es un lenguaje compilativo híbrido
- Todo programa en C consta de objetos.
- Las clases tienen parte pública y una privada. Y por defecto la clase es privada.
- Las clases contienen visibilidad.
- Presenta herencia simple y múltiple.

Clases y objetos:

Los objetos tienen:

- **Estado:** dado por sus datos miembro.
- **Comportamiento:** dado por sus funciones miembro.

Y todo objeto es una instancia de una clase. Y se comunican entre sí mediante envío de mensajes que desencadena la ejecución de una función miembro asociada.

La clase encapsula estado y comportamiento

Hay 2 funciones miembro asociadas: El constructor y el destructor.

Visibilidad de una clase:

Esta define que se puede acceder y cuál es el tipo de acceso.

- **Privada o protegida:** se accede solo a través de funciones miembro
- **Publica:** se accede directamente

Derivación: define el acceso que tiene la subclase a los datos miembro de la superclase. Y puede ser pública, privada o protegida.

- **Python→**

Características:

- **Lenguaje de programación interpretado**
- **Es multiparadigma.**
- **Soporta programación orientada a objetos e imperativa, y en menor medida funcional.**
- **Es un lenguaje dinámico y multiplataforma.**
- **Presenta herencia simple y múltiple.**

Definición de una clase:

Para definir una clase en Python se coloca la palabra reservada **CLASS** y después el nombre de la clase a crear.

Dentro de la clase se especifican los métodos y los atributos comunes a todos los objetos de dicha clase.

Constructor: Los constructores se ejecutan automáticamente justo después de crear o instanciar un objeto. Los constructores en Python se definen codificando un método especial llamado **__init__**. En el **__init__** se inicializa las variables de instancia del objeto, dejándolas disponibles para comenzar a operar con ellas a través de los métodos.

Otras definiciones:

La representación de un objeto de la vida real en un programa de software se lo denomina **objeto de software**.


Podemos representar **objetos abstractos** que no tienen representación física en el mundo real. Un objeto puede componerse de dos o más objetos, conformando así **objetos compuestos**.

- **Programación funcional** → En este paradigma un programa es un conjunto de definiciones de funciones o ecuaciones matemáticas, cuya evaluación permita obtener el resultado de una consulta. Y un lenguaje es Haskell.

Características:

1. EL S.O. evalúa las funciones y devuelve un resultado, a modo de calculadora.
2. En las funciones por un argumento dado existe un solo estado posible, sin importar el orden en que se evalúe.
3. Las variables se utilizan como variables matemáticas cuyo valor se liga al argumento de la función y no cambia.
4. **Transparencia referencial:** asegura que no habrá efectos laterales que modifiquen el valor final resultante.

Definición de funciones:

<code>cuad :: num -> num</code>	tipo de la función
<code>cuad(x) := x*x</code>	definición o ecuación
	
nombre argumento valor	

Tipo de datos:

- **Standard:** num, bool, char, α
- **Par ordenado:** (num, char), (num, num), etc.
- **Lista:** [] , [X], [X:Xs], [Xs: Xss]
- **Función**

Además, no existen las estructuras de control, solo se puede incluir condiciones en las funciones y para repetir se usa recursión.

- Por ecuaciones simples

`doblex:= 2* x`

- Por análisis de casos o Guardas

`min(x,y):= x, if x < y
:= y, otherwise`

- Con definiciones locales

`potCuarta(x):= cuad(x)*cuad(x)
where cuad(x):=x*x`

- Por patrones

`and(x,true):=x
and(x,false):=false`

- Recursivas

`fact(0):=1
fact(x):= x* fact(x-1)`

- De alto orden(composición de funciones)

usando map, filter, take, take while (se ven con listas)

Las listas: Son conjuntos de elementos del mismo tipo, incluso puede haber listas de listas.

Notación:

- Lista vacía $[]$

Ejemplos:

$[]$

- Lista unitaria $[X]$

$[3]$

- Lista simple Xs \circ $(X: Xs) \rightarrow X \rightarrow \text{cabeza}, Xs \rightarrow \text{cuerpo}$ $[2, 3, 6, 8, 9]$

- Lista de listas Xss \circ $(Xs: Xss) \rightarrow Xs \rightarrow \text{cabeza},$
 $Xss \rightarrow \text{cuerpo}$

$[[2], [2, 4], [8, 11]]$

Operaciones sobre listas:

- Concatenación
 - $\circ \quad [] : Xs = Xs : [] = Xs$
 - $\circ \quad (Xs : Ys) : Zs = Xs : (Ys : Zs)$
- Longitud
 - $\circ \quad \#[] = 0$
 - $\circ \quad \#(Xs : Ys) = \#Xs + \#Ys$
- Head
 - $\circ \quad \text{hd}(X : Xs) = X \quad \text{hd}(Xs : Xss) = Xs$
- Tail
 - $\circ \quad \text{tl}(X : Xs) = Xs \quad \text{tl}(Xs : Xss) = Xss$

Funciones de alto orden

- map
 - $\circ \quad \text{cubo}(X) := X * X * X$
 - $\circ \quad \text{cuboL}(X: Xs) := \text{map}(\text{cubo}(X)) Xs$

$? \text{cuboL}([2, 4, 3])$

$[8, 64, 27]$

- filter

$\text{listaPares}(X: Xs) := \text{filter}(\text{par}(X)) Xs$

where $\text{par}(X) := \text{true}$, if $X \bmod 2 = 0$

$\quad \quad \quad := \text{false}$, otherwise

$? \text{listaPares}[2, 3, 8, 9]$

$[2, 8]$

- take

- `take(0, [21, 3, 19])` → `[]`
- `take(2, [21, 3, 19])` → `[21, 3]`

- takewhile

- `takewhile(par(X), [24, 6, 3, 18, 25])` → `[24, 6]`

Toma los primeros elementos consecutivos que cumplen con la condición especificada.

La reducción consiste en sustituir la función por su definición y simplificar. No importa el orden de reducción, siempre debe dar el mismo resultado:

- Normal: primero se evalúa la función y después se simplifican los argumentos
- Aplicativo: primero se simplifican los argumentos y luego se evalúa la función

Orden normal

```
?cuad(5 + 4)
(5+4)* (5+4)
9 * (5+4)
9 * 9
81
```

Orden aplicativo

```
?cuad(5 + 4)
cuad(9)
9 * 9
81
```

- **Programación lógica** → En este paradigma se trabaja de forma descriptiva, es decir, que el programa no indica cómo resolver el problema sino que establece relaciones entre las entidades del mismo, las cuales describen que hacer.

Características:

- Se lo encuadra en la programación declarativa.
- Está formado por expresiones lógicas que declaran o describen la solución y es el sistema interno quien proporciona la secuencia de control en que se utilizan esas expresiones.
- Tiene como característica principal el uso de reglas lógicas o predicados de 1º orden llamados **cláusulas de horn** para inferir o derivar conclusiones a partir de datos.
- un **algoritmo lógico se construye**: Especificando una base de conocimiento sobre la que se realizan consultas y aplicando un mecanismo de inferencia o deducción sobre dicha base, se infieren conclusiones.

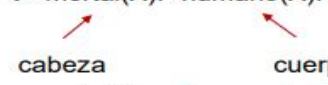
Cada cláusula permite deducir conclusiones a partir del conjunto de datos del problema. Una cláusula de Horn tiene la siguiente forma:

$$c := p_1, p_2, p_3, p_4, \dots, p_k$$

donde c es una conclusión o consecuente y las p_i son las premisas o antecedentes. Se dice que c es verdadera si son verdaderas cada una de las premisas p_i .

Estos están formados por:

- **Hechos**: representan enunciados o predicados siempre verdaderos que muestran relaciones entre los datos del problema. (axiomas)
- **Reglas**: muestran relaciones entre los datos, los hechos y otras reglas, y permiten deducir o demostrar objetivos.

- Hechos → hombre(juan).
hombre(tomás).
mujer(ana).
progenitor(ana, juan).
- Reglas → mortal(X):- humano(X).

que significa: **X es mortal si X es humano**
Si X es humano entonces X es mortal.
madre(X,Y):- mujer(X), progenitor(X,Y).

Consultas: Una consulta es un objetivo a ser demostrado por comparación o pattern matching contra la base de conocimiento. Se trata de aparear la consulta con algún hecho o con la cabeza de una regla. **La consulta puede ser de dos tipos y responden:**

- validación (cuya respuesta es Si o No).
- una búsqueda (que retorna los valores que hacen verdadero al objetivo).

Si **puede demostrar** la consulta u objetivo, responde:

- **Yes** (si es una validación).
- o **el/los valor/es** que satisface/n la consulta (en caso de búsqueda).
- Si no puede demostrar el objetivo, responde **No**, que tiene dos significados: que es falso o que no puede deducirlo con la información contenida en la base de conocimiento.

Si la consulta posee variables, las mismas se **asocian o ligan** con los valores de los hechos o con los argumentos de las cabezas de las reglas. Esto se denomina **unificación**. No hay efectos laterales. Al aparearse las variables de una consulta con la cabeza de una regla, las variables se ligan y luego se aplica la regla: se usa su definición y debe demostrarse cada sub-objetivo de la misma. Este proceso de demostración se denomina **resolución**.

Hay 3 tipos de consultas:

- I. Sin variables → ?gusta(maria, juan) a maria le gusta juan
Validación
- II. Con variables → ?gusta(maria, X) quién le gusta a maria
Búsqueda
- III. Con variable anónima: para saber si existe algún objeto que haga verdadera la consulta → ?gusta(maria, _) hay alguien que le gusta a maria. La respuesta es Si o No

Consultas: reversibilidad el predicado

- Se pueden introducir valores en la consulta esperando que el motor de inferencias encuentre las variables de salida que aparezcan o unifiquen con la consulta.

?- progenitor(ana, X)

X=juan;

No

De quién es progenitor ana?

BC

mujer(ana).

hombre(juan).

progenitor(ana, juan).

- Recíprocamente también puedo introducir un valor de salida y esperar que el motor encuentre qué valores de entrada son adecuados para satisfacer esa salida.

?- progenitor(Y, juan)

Y=ana;

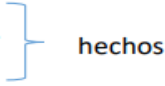
No

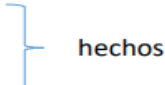
quién es el progenitor de juan?

Motor de inferencias

Es quien controla la ejecución del programa lógico a partir de una consulta. Recorre la base de datos desde el principio y permite derivar o deducir un objetivo en base a dos mecanismos: recursión y backtracking. Gracias al **backtracking**, si falla 1 camino de demostración, retrocede a un punto previo (deshace la última sustitución) y reinicia el proceso de demostración. Cuando agotó todos los caminos posibles, termina la demostración.

Tipos de reglas

- Simples
 - `notrabaja(belen).`
 - `bueno(pedro).`

`cuida(belen, pedro):- notrabaja(belen), bueno(pedro).`
- Con variables
 - `humano(juan).`
 - `humano(ana).`

`mortal(X):- humano(X).`
- Recursivas
 - `antepasado(X,Y):- progenitor(X,Y).`
 - `antepasado(X,Y):- antepasado(X,Z), progenitor(Z,Y).`

Cómo programar en este paradigma (Prolog):

- A) Modelar el problema mediante una Base de Conocimiento formada por hechos que muestren las relaciones de los datos de dicho problema.
- B) Formular luego reglas que se cumplan en el mundo del problema modelado.
- C) Formular consultas al sistema para que demuestre o deduzca la respuesta a partir de la base de conocimiento usando resolución.

- Predicados predefinidos:
 - `true`
 - `fail`
 - `var(X)` devuelve V si X es una variable sin instanciar
 - `nonvar(X)` devuelve T si X tiene valor asignado
 - `integer(X)` devuelve V si X es una variable entera
 - `float(X)` devuelve V si X es una variable real
 - `number(X)` devuelve V si X es una variable numérica
 - `write(X)` imprime el valor de X
 - `read(X)` ingresa un valor a X
 - `==` compara si son iguales
 - `\==` compara si son distintos
- Operadores: `+` `-` `/` `*` `X mod Y`

Listas: Son conjuntos de elementos del mismo tipo. Las listas se denotan con corchetes, separando los elementos por comas. Están divididas en cabeza (X) y cuerpo (Xs) $\rightarrow [X | Xs]$. La lista vacía se denota `[]`. Se recorren recursivamente.

Contienen términos (variables, constantes o estructuras) en general. Es posible anidar listas. Ejemplos: • [1,2,3,4] • [a,b,c,d] • [1,'a', X,[1,2,3]] • [[1,2],[3,4],[5,6]] .

Listas: ejemplos

Longitud de una lista

long([],C):-C is 0.

long([X|Xs], C):- long(Xs,C1), C is C1 + 1.

Suma de los elementos de una lista

sum([],0).

sum([X|Xs], S):-sum(Xs,S1), S is S1 + X.

Cantidad de elementos pares de una lista

par(X):- integer(X), 0 is X mod 2.

impar(X):-integer(X), 1 is X mod 2.

cantP([],0).

cantP([X|Xs], S):- par(X), cantP(Xs,S1), S is S1 + 1.

cantP([X|Xs], S):- impar(X), cantP(Xs,S1), S is S1 .

Verifica si todos los elementos de una lista son hombres

hombre(juan).

hombre(pedro).

hombres([]).

hombres([X|Xs]):-hombre(X),hombres(Xs).

Verifica si todos los elementos de una lista son iguales

todoslg([]).

todoslg([X]).

todoslg([X,Y|Xs]):- X==Y, todoslg(Xs).

todoslg([X,Y|Xs]):- X \==Y, false.

Cuenta los elementos de una lista que son hombres

hombre(juan).

hombre(pedro).

hombres([],0).

hombres([X|Xs], Cont):-hombre(X),hombres(Xs, C), Cont is C +1.

hombres([X|Xs], Cont):- not (hombre(X)) ,hombres(Xs, C), Cont is C.

Cantidad de elementos mayores a 5 de una lista

cantMay5([],0).

cantMay5([X|Xs], S):- X > 5, cantMay5(Xs,S1), S is S1 + 1.

cantMay5([X|Xs], S):- X < 5, cantMay5(Xs,S1), S is S1 .

cantMay5([X|Xs], S):- X == 5, cantMay5(Xs,S1), S is S1 .

Imprimir los elementos de una lista

mostrar([]).

mostrar([X|Xs]):- write(X), nl, mostrar(Xs).

No existe un elemento en una lista

noesta(X, []).

noesta(X, [Y|Ys]):- X \==Y, noesta(X,Ys).

noesta(X, [Y|Ys]):- X ==Y, false.

noesta(X, []).

noesta(X, [Y|Ys]):- X \==Y, noesta(X,Ys).

Acá uso la hipótesis de mundo cerrado: lo que no está en la base de conocimiento, es falso.

Existe un elemento en una lista

`member(X,[]):-fail.`

`member(X,[Y|Ys]):-X==Y, true.`

`member(X,[Y|Ys]) :- X\==Y, member(X,Ys).`

Otra forma:

`member(X,[X|Xs]).`

`member(X,[Y|Ys]) :- member(X,Ys).`

Añadir un elemento a una lista (al pcpio)

`add(X,[],[X]).`

`add(X,L,[X|L]).`

Otra forma

`add(X,[],[X]).`

`add(X,[Y|Ys],[X,Y|Ys]).`

Borrar **un solo** elemento de una lista

`del(X,[],[]).`

`del(X,[Y|Ys],Ys):-X==Y.`

`del(X,[Y|Ys],[Z|Zs]):-X\==Y, del(X,Ys,Zs), Z is Y.`

Otra forma:

`del(X,[],[]).`

`del(X,[X|Xs],Xs).`

`del(X,[Y|Ys],[Y|Zs]) :- X\==Y, del(X,Ys,Zs).`

Verificar si hay algún elemento positivo en la lista

`unPosit([]):- false.`

`unPosit([X|Xs]):- X>0.`

`unPosit([X|Xs]):- (X < 0; X=0), unPosit(Xs).`

