

# Unidad 7

## Programación Lógica y Prolog

7.1 Introducción a Prolog

7.2 Tipos de Datos en Prolog

7.3 Calce de Términos en Prolog

7.4 Listas y Operadores

7.5 Operadores y Aritmética

7.6 *Backtracking*

## 7.1 Introducción a Prolog

# Lenguaje Prolog

- Su nombre viene de PROgramación en LOGica, creado a comienzos de los '70:
  - Robert Kowalski (Edimburgo): lado teórico
  - Maarten van Emden (Edimburgo): demostración práctica
  - Alain Colmerauer (Marsella): Implementación
- Popularidad se debe a David Warren (Edimburgo), que a mediados de los '70 realizó una implementación eficiente.

# Lenguaje Prolog: características

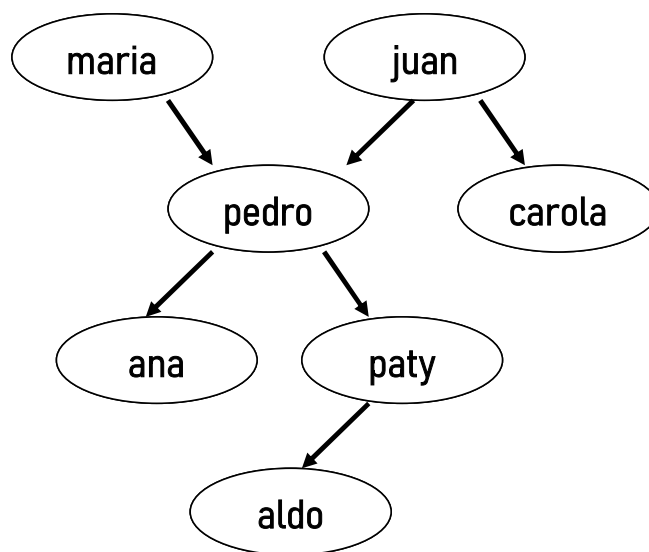
- Basado en lógica simbólica y programación declarativa.
- No se especifica cómo debe hacerse, sino qué debe lograrse (alto nivel).
- El programador se concentra más en el conocimiento que en los algoritmos
  - ¿Qué es conocido? (hechos y relaciones, y reglas )
  - ¿Qué preguntar? (cómo resolverlo)
- Estilo de programación orientado a metas.

# Lenguaje Prolog: aplicaciones

- Pruebas Matemáticas
  - Demostración de teoremas
- Inteligencia Artificial
  - Sistemas Expertos
- Consultas a base de datos
  - Permite inferir relaciones no especificadas a priori.

# Hechos y Consultas

padre(maria, pedro).  
padre(juan, pedro).  
padre(juan, carola).  
padre(pedro, ana).  
padre(pedro, paty).  
padre(paty, aldo).



?- padre(pedro, ana).  
=> yes

?- padre(ana, paty).  
=> no

?- padre(X, carola).  
=> X = juan

?- padre(pedro, X).  
=> X = ana ;  
=> X = paty ;  
=> no

# Hechos y Consultas: ejemplo 1

- Preguntar por el abuelo de aldo:

$\exists X, Y : (X \text{ es padre de } Y) \cap (Y \text{ es padre de aldo})$

que se expresa en Prolog como:

```
?- padre(X, Y), padre(Y, aldo).  
=> X = pedro  
    Y = paty
```

## Hechos y Consultas: ejemplo 2

- Preguntar por los nietos de juan:

$$\exists X, Y : (\text{juan es padre de } X) \cap (X \text{ es padre de } Y)$$

que se expresa en Prolog como:

```
?- padre(juan, X), padre(X, Y).
```

```
=> X = pedro
```

```
    Y = ana ;
```

```
=> X = pedro
```

```
    Y = paty
```



## Hechos y Consultas: ejemplo 3

- Preguntar si ana y paty tienen un padre en común:

$$\exists X : (X \text{ es padre de ana}) \cap (X \text{ es padre de paty})$$

que se expresa en Prolog como:

```
?- padre(X, ana), padre(X, paty).  
=> X = pedro
```

# Reglas

- La relación  $a \subset b$  se expresa en Prolog como  $a :- b$ .
- Una cláusula de este tipo se denomina **regla**, que tiene la siguiente estructura:
  - la **cabeza** (parte izquierda de  $:-$  ) es la **conclusión** ...
  - de la proposición definida en el **cuerpo** (parte derecha de  $:-$ )

# Reglas: Resolución simple

- La relación *hijo de* corresponde a:  
 $\forall X, Y : (Y \text{ es hijo de } X) \subset (X \text{ es padre de } Y)$
- que se expresa en Prolog, usando lo definido anteriormente, como:  
`hijo(X, Y) :- padre(Y, X).`

- **Ejemplo:** la meta siguiente es evaluada como:

La meta: `hijo(paty, pedro)`  
se convierte en submeta: `padre(pedro, paty)`  
Se busca este hecho: `yes`

# Reglas: ejemplo

- Agregar cláusulas sobre el género de las personas: alternativas.

femenino(maria).  
masculino(juan).  
masculino(pedro).  
femenino(carola).  
femenino(ana).  
femenino(paty).  
masculino(aldo).

Relaciones unarias



Usaremos esta forma

genero(maria, femenino).  
genero(juan, masculino).  
genero(pedro, masculino).  
genero(carola, femenino).  
genero(ana, femenino).  
genero(paty, femenino).  
genero(aldo, masculino).

Relaciones binarias

## Reglas: ejemplo

- Se puede definir ahora varias nuevas reglas como:

```
papa(X, Y)      :- padre(X, Y), masculino(X).
mama(X, Y)      :- padre(X, Y), femenino(X).
abuelo(X, Y)     :- padre(X, Z), padre(Z, Y).
hermana(X, Y)    :- padre(Z, X), padre(Z, Y), femenino(X).
```

- Se puede definir ahora varias nuevas reglas como:

```
?- hermana(ana, paty).
=> yes
```

```
?- hermana(X, paty).
```

```
=> X = ana ;
```

```
=> X = paty
```

oops ... paty es hermana de ella misma

¡Falta excluir este caso:

```
hermana(X, Y) :- padre(Z, X), padre(Z, Y),
                diferente(X, Y), femenino(X).
```

# Reglas Recursivas

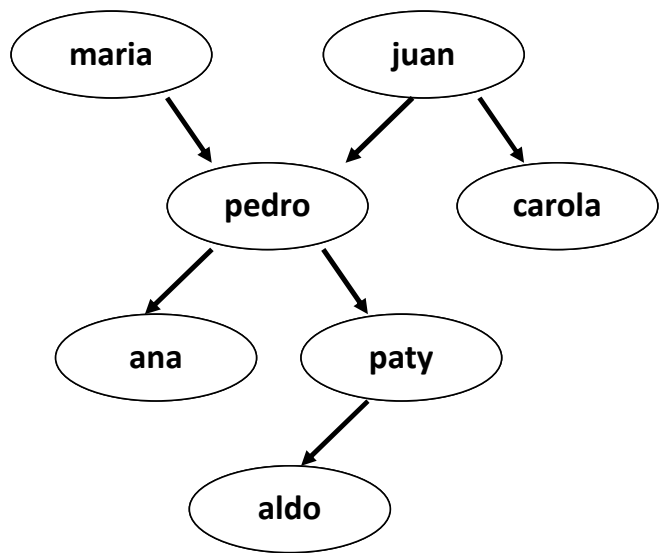
- La relación **antepasado** se define sobre la base de una regla de descendencia directa y otra regla de descendencia indirecta:

$\forall X, Z : (X \text{ es un antepasado de } Z), \text{ si}$   
 $\{X \text{ es padre de } Z\} \vee$   
 $\{\exists Y: (X \text{ es padre de } Y) \wedge (Y \text{ es antepasado de } Z)\}$

lo que en Prolog se expresa como :

`antepasado(X, Z) :- padre(X, Z).`  
`antepasado(X, Z) :- padre(X, Y), antepasado(Y, Z).`

# Reglas Recursivas



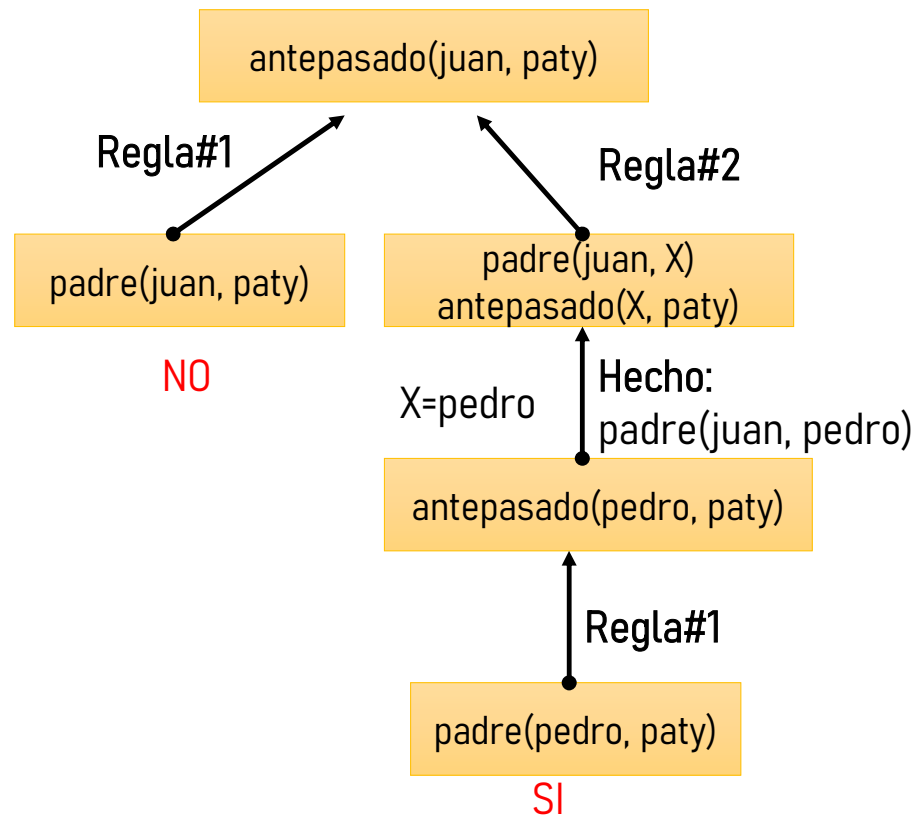
?- antepasado (maria, X).

=> X = pedro ;

=> X = ana ;

=> X = paty ;

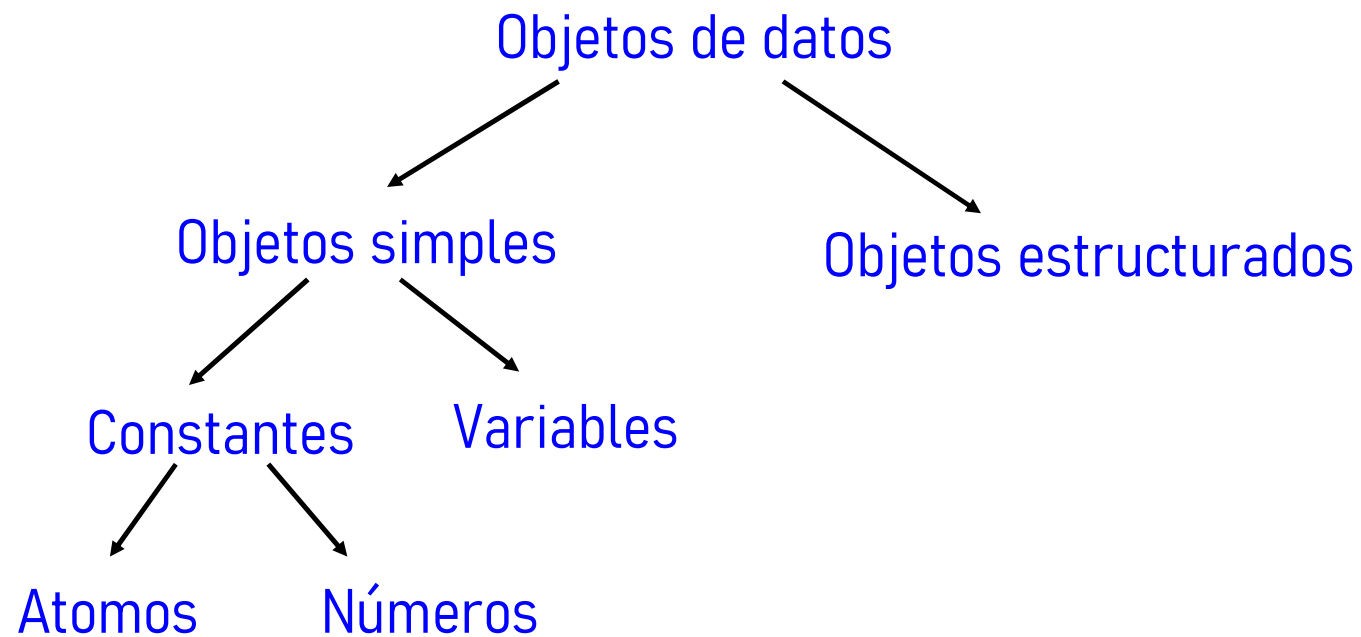
=> X = aldo



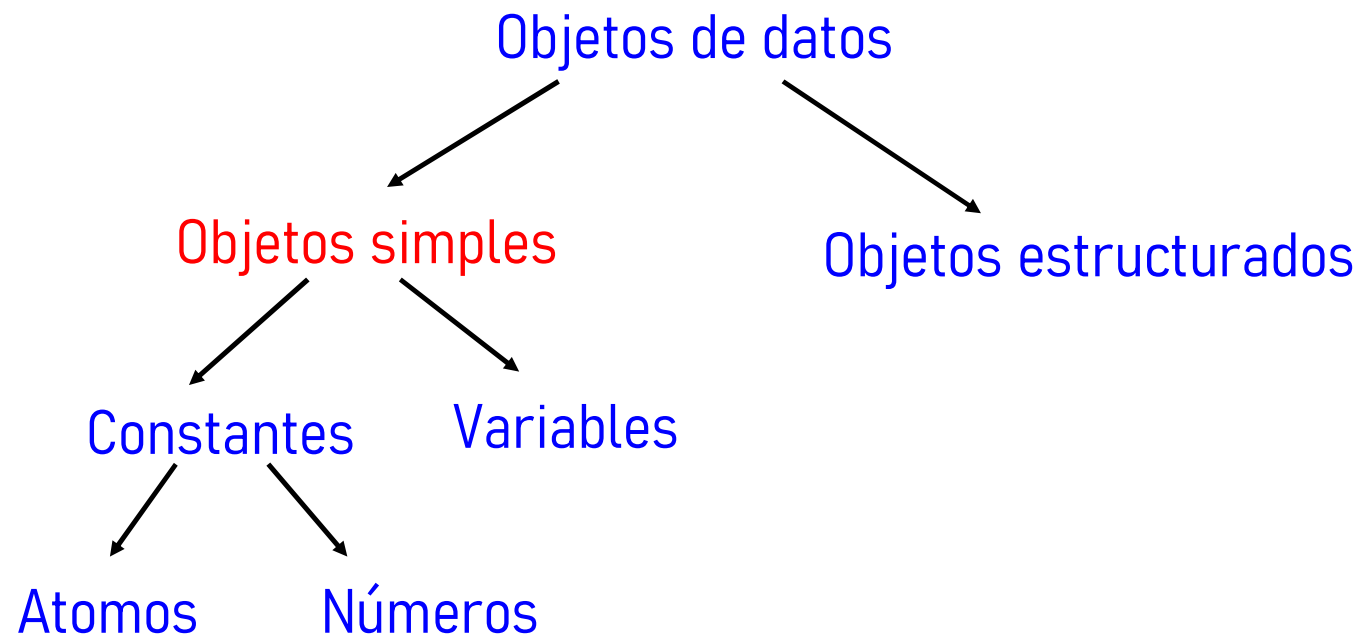
## 7.2 Tipos de Datos en Prolog



# Tipos de Objetos de Datos



# Tipos de Objetos de Datos



# Reconocimiento de Tipos

- Se reconoce el tipo de un dato por su forma sintáctica:
  - No se requiere de declaración de tipos.
- Ejemplo:
  - Variables comienzan con letra en mayúsculas (e.g. **X**).
  - Átomos comienzan con una letra en minúscula (ej.: **pedro**).

# Formación de Variables y Átomos

- Strings formados con una combinación de los siguientes caracteres:
  - Letras mayúsculas `A..Z`
  - Letras minúsculas `a..z`
  - Dígitos `0..9`
  - Caracteres especiales: `+ - * / < > = : . & _ ~`
- Ejemplos:
  - Letras, dígitos y underscore (`_`), comenzando con minúscula: `pedro`, `nil`, `x_25`, `algo_especial`
  - Caracteres especiales: `<----->`, `===>`, ...
  - Citación simple: `'Juan'`, `'San Francisco'`

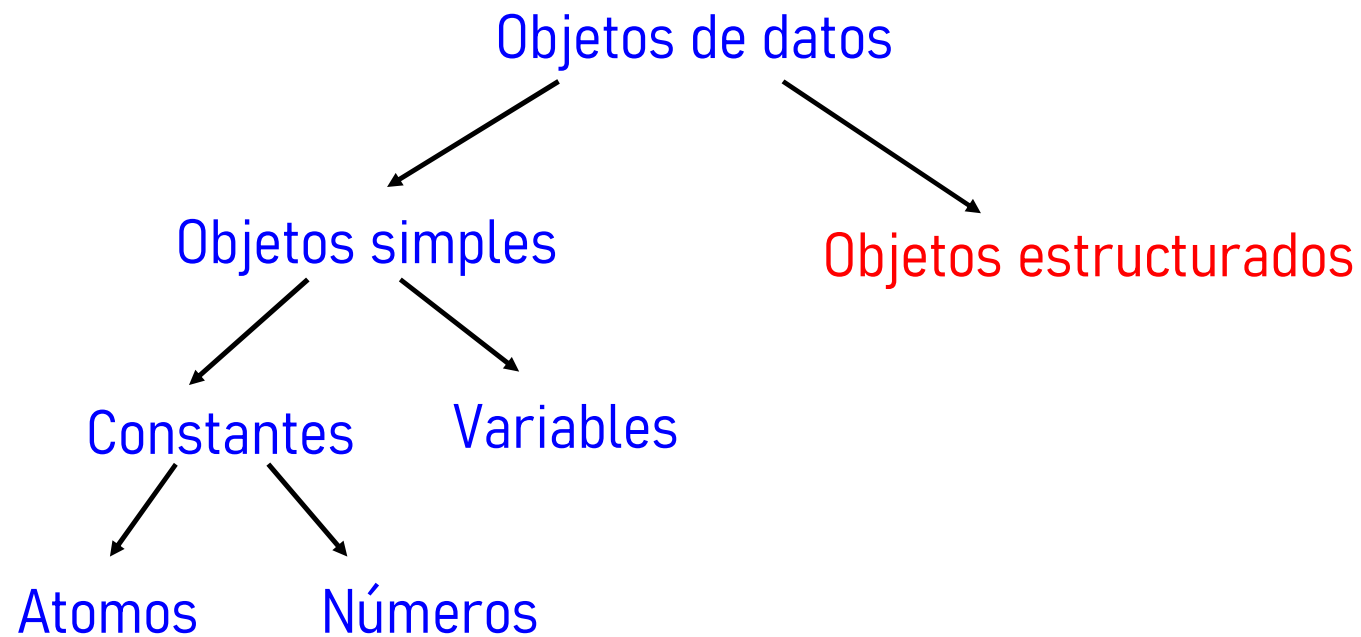
# Números

- Números enteros:        1        3213        0        -323
- Reales:                3.14    -0.0234    100.2
- Observación: dado que Prolog es principalmente un lenguaje de computación simbólica, los números no son su fuerte (el entero es lo que más se usa).

# Variables

- Strings de letras, dígitos y *underscore*, comenzando con mayúscula o *underscore*: `X`      Resultado `_X1`    `_12`
- Si una variable aparece una solo vez en una cláusula, se puede usar variables anónima `_`  
    `?- padre(juan, _).`  
        `yes`                      `% no se imprime variable`  
  
    `tiene_hijo(X) :- padre(X, _).`
- Observación: el ámbito de una variable es la cláusula que lo contiene.

# Tipos de Objetos de Datos

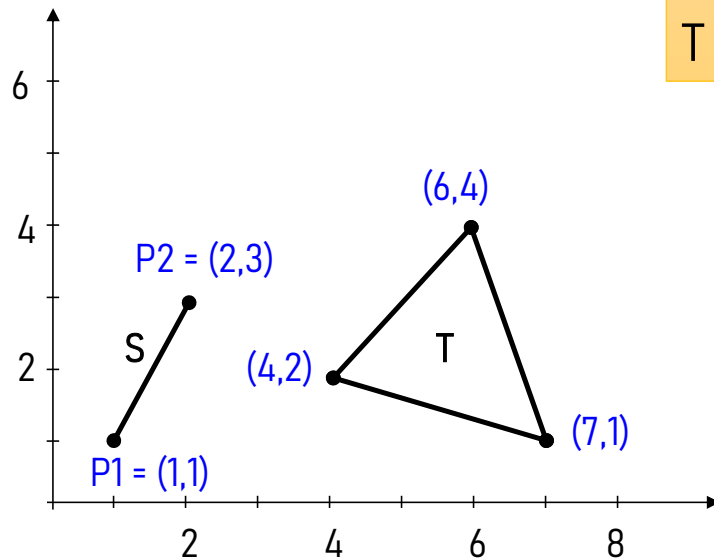


# Objetos Estructurados

- Son objetos que tienen varias componentes.
- Estructuras son tratadas como un único objeto.
- Se construyen usando un *functor*.  
fecha(22, mayo, 2000)
- Componentes pueden ser constantes, variables o estructuras:  
fecha(Dia, mayo, 2000)



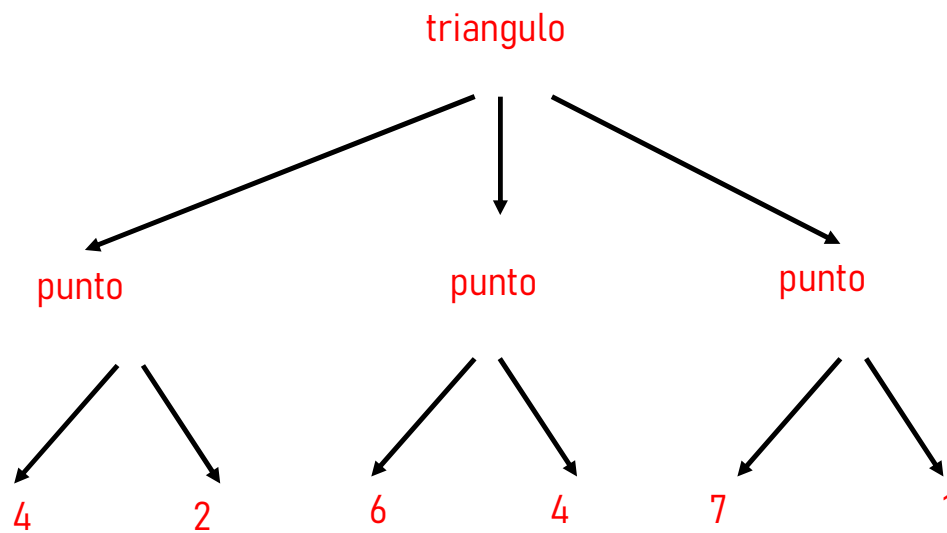
# Objetos Estructurados: Ejemplo 1



P1 = punto(1, 1)  
P2 = punto(2,3)  
S = segmento(P1, P2)  
T = triangulo(punto(4,2), punto(6,4), punto(7,1))

## Objetos Estructurados: Ejemplo 2

`T = triangulo(punto(4,2), punto(6,4), punto(7,1))`



## 7.3 Calce de Términos en Prolog

## Calce de Términos (*matching*)

- La operación más importante sobre términos es el *calce*, que corresponde a la unificación en el cálculo de predicados.
- Dos términos *calzan* si:
  - ① Son idénticos, o
  - ② las variables en ambos términos pueden ser instanciados, sustituyendo variables, tal que los términos se hacen idénticos.

# Calce de Términos: Ejemplos

- Calzar: `fecha(D, M, 2000)` y `fecha(D1, mayo, A1)`,

entonces:

- D se instancia a D1
- M se instancia a mayo
- A1 se instancia a 2000

que como salida de Prolog se escribe:

- D = D1
- M = mayo
- A1 = 2000

?- `fecha(D, M, 2000) = fecha(D1, mayo, A1).`

D = H86,

M = mayo,

D1 = H86,

A1 = 2000.

# Calce de Términos: Ejemplos

- Al intentar calzar:
  - `fecha(D, M, 2000) = fecha(D1, julio, 1956)`

no existe respuesta (salida es `false`).

?- `fecha(D, M, 2000) = fecha(D1, julio, 1956).`  
No.

- En general:
  - No es posible encontrar un calce (se dice que el proceso de calce ha **fracasado**).
  - En caso contrario, se dice que el proceso ha sido **exitoso**.

## Calce de Términos: Grado de Ajuste

- ?- fecha(D, M, 2000) = fecha(D1, mayo, A1).

podría haber sido calzado como:

- D = 1
- D1 = 1
- M = mayo
- A1 = 2000

Pero esta forma es más restrictiva (menos general) que la anterior.

¡Prolog calza el resultado a su forma más general!

# Calce de Términos: Reglas de Calce

Dos términos  $S$  y  $T$  calzan, si:

- ① Si  $S$  y  $T$  son *constantes*, entonces  $S$  y  $T$  calzan si ambos son el mismo objeto.
- ② Si  $S$  es una *variable* y  $T$  cualquier cosa, entonces calzan y  $S$  se instancia como  $T$ . Viceversa, si  $T$  es variable, entonces  $T$  se instancia como  $S$ .
- ③ Si  $S$  y  $T$  son *estructuras*, entonces calzan sólo si:
  - a)  $S$  y  $T$  tienen el mismo functor, y
  - b) Todas sus correspondientes componentes calzan. (instanciaciones resultantes son determinadas por proceso de calce de componentes).



# Calce de Estructuras: Ejemplos

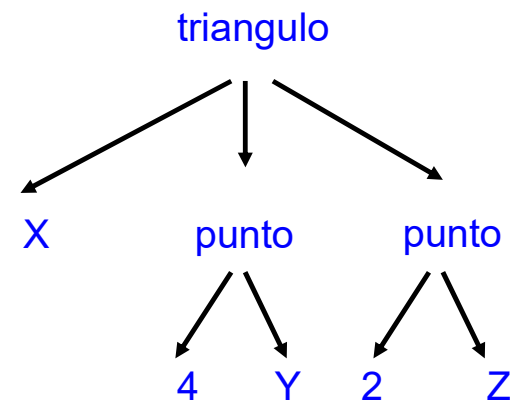
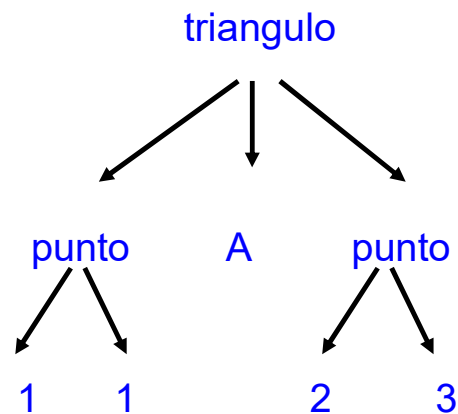
?- `triangulo(punto(1, 1), A, punto(2, 3)) = triangulo(X, punto(4, Y), punto(2, Z)).`

`A = punto(4,H193),`

`X = punto(1,1),`

`Y = H193,`

`Z = 3.`



# Calce de Estructuras: Ejemplos

Hechos:                   vertical(seg(punto(X, Y), punto(X, Y1))).  
                             horizontal(seg(punto(X, Y), punto(X1, Y))).

Consultas:               ?- vertical(seg(punto(1,1), punto(1,2))).  
                             yes  
                             ?- vertical(seg(punto(1,1), punto(2,Y))).  
                             no  
                             ?- horizontal(seg(punto(1,1), punto(2,Y))).  
                             Y = 1  
                             ?- vertical(seg(punto(2,3), Y)).  
                             Y = punto(2,H561)  
                             ?- vertical(S), horizontal(S).  
                             S = seg(punto(H576,H577), punto(H576,H577))

# Significado Declarativo versus Procedural

- La cláusula:  $P :- Q, R$ .  
se *interpreta declarativamente* como:
  - $P$  es verdadero si  $Q$  y  $R$  lo son.
  - o, de  $Q$  y  $R$  se deriva  $P$ .
- En cambio, una *interpretación procedural* sería:
  - Para resolver  $P$ ,  
primero se debe resolver  $Q$  y luego  $R$ .
  - Para satisfacer a  $P$ ,  
primero se debe satisfacer  $Q$  y luego  $R$ .

# Significado Declarativo

- Una meta **G** es verdadera (satisface o se deriva lógicamente de un programa), ssi:
  - ① existe en el programa una cláusula **C**, tal que
  - ② existe una cláusula **I**, instancia de **C**, tal que:
    - la cabeza de **I** es idéntica a **G**, y
    - todas las metas en el cuerpo de **I** son verdaderas.

# Cláusulas Disjuntivas

La cláusula:  $P :- Q; R.$

se interpreta como:  $P :- Q.$   
 $P :- R.$

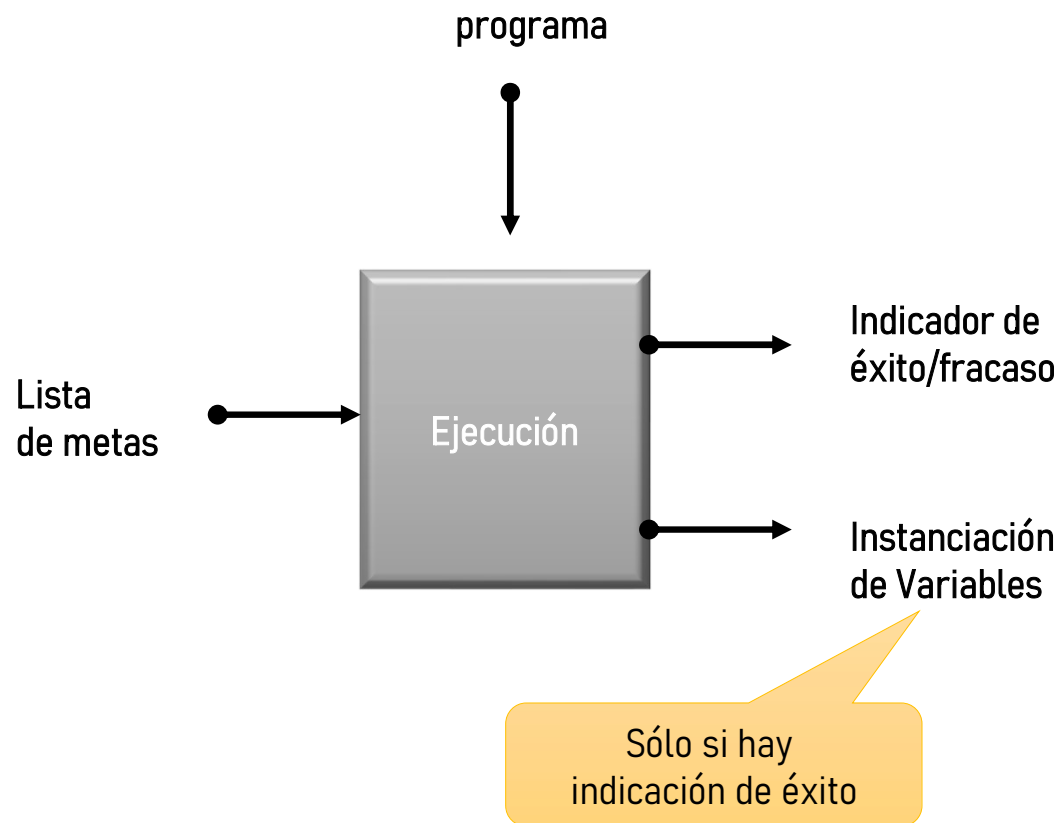
La cláusula:  $P :- Q, R;$   
 $S, T, U.$

se interpreta como:  
 $P :- Q, R.$   
 $P :- S, T, U.$

# Significado Procedural

- Especifica *cómo* responder a una pregunta.
- Para obtener la respuesta es necesario satisfacer una lista de metas.
  - Las metas pueden ser satisfechas si a través de la instanciación de sus variables se permite que del programa se deriven las metas.

# Significado Procedural



## 7.4 Listas y Operadores



# Listas

- Una lista en Prolog se puede escribir como: `[perro, gato, ratón, loro]`
- Sin embargo esto es sólo un *sabor sintáctico*, pues Prolog lo traduce a una forma de estructura.
- Si existe estructura `.(Cabeza, Cola)`, entonces:  
`.(perro, .(gato, .(ratón, .(loro, []))))`  
equivale a la lista anterior (que es más legible).

# Listas: Representación

- Una lista define un árbol binario, similar a las listas propias de *Scheme*.
- Prolog permite una notación similar a los pares:
  - $L = [a \mid Cola]$  , donde  $a$  es la cabeza (cualquier tipo) y  $Cola$  es el resto de la lista (debe ser una lista) .
  - La lista vacía se expresa como  $[]$ .
- Ejemplo:  
 $?- L2 = [a \mid [b \mid []]].$   
 $L2 = [a,b]$

# Listas: algunos Operadores

- Pertenencia del objeto X en la lista L: `member(X, L)`  
?- `member(X, [a, b]).`  
X = a ;  
X = b ;  
no
- Concatenación de listas L1 y L2 en L3: `append(L1, L2, L3)`  
?- `append([a], [b], L).`  
L = [a,b] ;  
no

# Listas: algunos Operadores

- Borrar un elemento X en una lista L: `delete(L, X, L1)`  
?- `delete([a, b, c, b, d], b, L).`  
L = [a,c,d] ;  
no

## Listas: subListas

- Una sublista es una parte de una lista.
- El operador puede ser definido con la siguiente regla:

`sublist(S, L) :- append(L1, L2, L), append(S, L3, L2).`

- Ejemplo:

`?- sublist([b, X], [a, b, c, d]).`

`X = c`

# Listas: Recursión

```
% Permutar la lista L
% =====
permutation([], []).
permutation(L, [X | P]) :- del(X, L, L1),
                           permutation(L1, P).
```

?- permutation([gato, perro, raton], L).

L = [gato,perro,raton] ;

L = [gato,raton,perro] ;

L = [perro,gato,raton] ;

L = [perro,raton,gato] ;

L = [raton,gato,perro] ;

L = [raton,perro,gato] ;

no

## 7.5 Operadores y Aritmética

# Operadores: Notación

- Las operaciones en Prolog se expresan normalmente como *functores*.
- Se permite también especificar operadores especiales con su relación de precedencia mediante directivas al traductor Prolog.
- Este mecanismo permite mejorar la lectura de programas (sabor sintáctico), con mecanismo similar a la sobrecarga de operadores en C++.



# Operadores: Ejemplo

La expresión:

$+(*(2, a), *(b, c))$

podría escribirse como:

$2*a + b*c$

¡¡Que resulta más legible!!

Ejemplo en Prolog:

?-  $X = +(*(2, 3), *(4, 5)).$   
 $X = 2 * 3 + 4 * 5$

?-  $X \text{ is } +(*(2, 3), *(4, 5)).$   
 $X = 26 .$

?-  $X \text{ is } 2*3 + 4*5.$   
 $X = 26$

# Operadores: Definiciones

- Se permite definición de operadores *prefijos*, *infijos* y *postfijos*
- A cada operador se le puede definir el nivel de precedencia mediante un valor (e.g. entre 1-1200 en una implementación).
- Nombre del operador debe ser un átomo.
- Ejemplo: Operador binario infijo *gusta*  
:- op(600, xfx, gusta).

# Operadores: Tipos

- Operador Infijo (tres tipos):     $xfx$      $xfy$      $yfx$
- Operador Prefijo (dos tipos):     $fx$      $fy$
- Operador Postfijo (dos tipos):     $xf$      $yf$

donde la notación se interpreta como.

- $f$  corresponde al nombre del operador
  - $x$  e  $y$  representan los argumentos
  - $x$  representa operando con precedencia estrictamente menor que el operador
  - $y$  representa operando cuya precedencia es menor o igual que el operador
- 
- Esta definición define la asociatividad de los operadores ( $y$  domina sobre  $x$ ).

# Operadores: Predefinidos

`:- op(1200, xfx, ':-').`  
`:- op(1200, fx, [:-, ?-] ).`  
`:- op(1100, xfy, ';' ).`  
`:- op(1000, xfy, ',').`  
`:- op(700, xfx, [=, is, <, >, =<, >=, ==, =\=, \==, :=] ).`  
`:- op(500, yfx, [+ , -] ).`  
`:- op(500, fx, [+ , - , not] ).`  
`:- op(400, yfx, [* , - , div] ).`  
`:- op(300, xfx, mod).`

# Operadores: Ejemplos

?-  $X = 3 + 5$ .

$X = 3 + 5$ .

yes

?-  $X \text{ is } 3 + 5$ .

$X = 8$ .

yes

?-  $X \text{ is } 5/3$ .

$X = 1.6666666666666667$ .

yes

?-  $X \text{ is } 5 - 2 - 1$ .

$X = 2$ .

yes

?-  $1 + 2 = 2 + 1$ .

no

?-  $1 + 2 ::= 2 + 1$ .

yes

## Operadores: Ejemplo del Largo de una Lista

```
largo([], 0).
```

```
largo([_ | Cola], N) :-  
    largo(Cola, N1),  
    N is N1 + 1.
```

```
?- largo([a, b, [c, d], d, e], N).  
    N = 5
```

# Operadores: Ejemplo del Máximo Común Divisor

`mcd(X, X, X).`

`mcd(X, Y, D) :-  
 X < Y, Y1 is Y - X, mcd(X, Y1, D).`

`mcd(X, Y, D) :-  
 Y < X, mcd(Y, X, D).`

`?- mcd(100, 10, X).  
 X = 10`

`?- mcd(27, 36, X).  
 X = 9`

## *7.6 Backtracking*

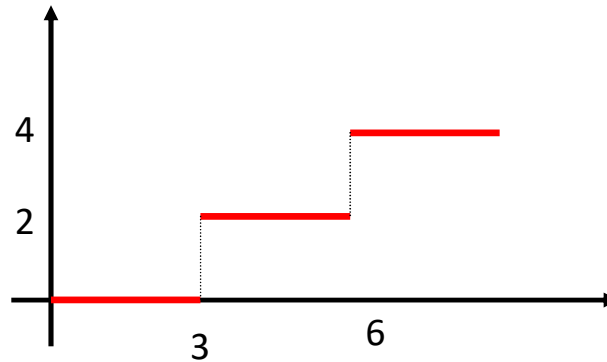


## *Backtracking*

- Prolog realiza *backtracking* automático si falla la satisfacción de una cláusula.
- Sin embargo, en algunos casos el *backtracking* automático es ineficiente.
- El programador puede controlar o prevenir el *backtracking* usando **cut**.

## *Backtracking.* Ejemplo de una Función

- Suponga las siguientes tres reglas para una función de doble escalón:
  - **Regla #1:** Si  $(X < 3)$  entonces  $Y = 0$
  - **Regla #2:** Si  $(X \geq 3)$  y  $(X < 6)$ , entonces  $Y = 2$
  - **Regla #3:** Si  $(X \geq 6)$ , entonces  $Y = 4$



## *Backtracking.* Ejemplo de una Función

% Regla #1

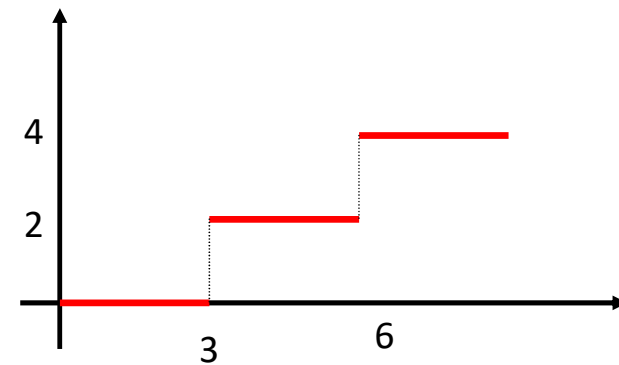
$f(X, 0) :- X < 3.$

% Regla #2

$f(X, 2) :- X \geq 3, X < 6.$

% Regla #3

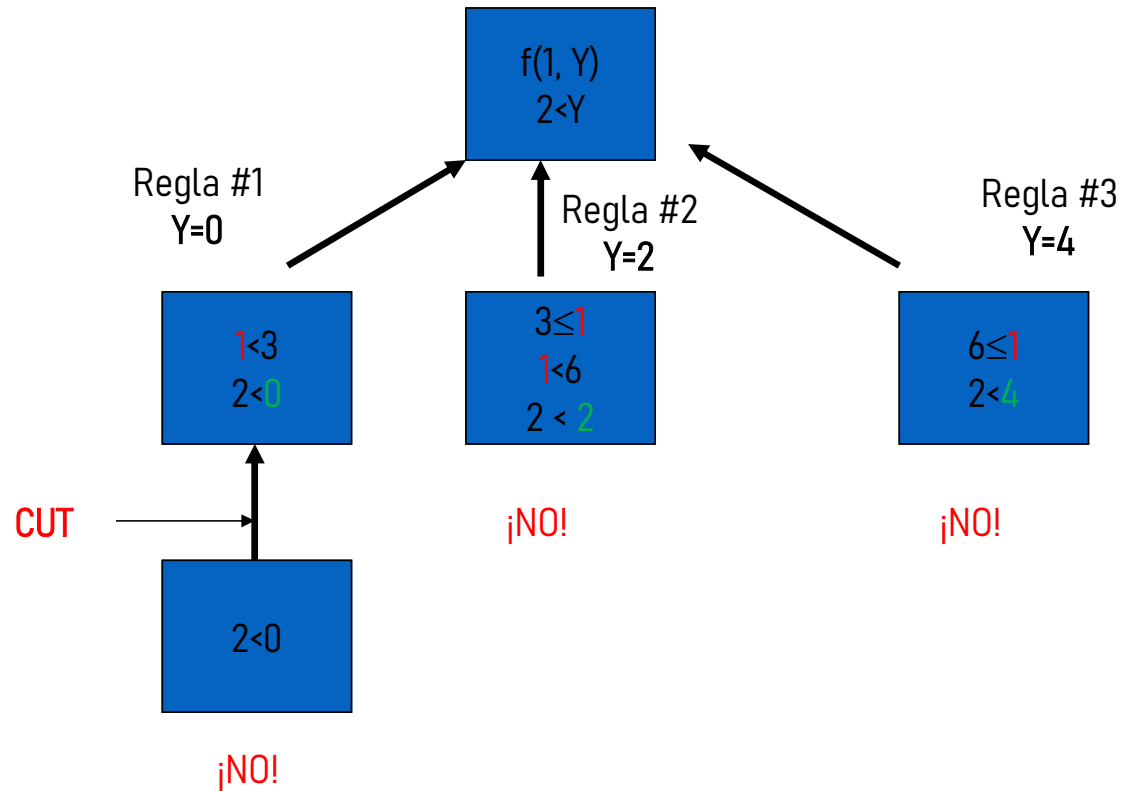
$f(X, 4) :- X \geq 6.$



# Backtracking. Ejemplo de una Función

Evaluación de la Meta

-?  $f(1, Y), 2 < Y.$   
no



## *Backtracking.* Ejemplo de una Función y operador Corte

% Regla #1

$f(X, 0) :- !, X < 3.$

% Regla #2

$f(X, 2) :- X \geq 3, X < 6, !.$

% Regla #3

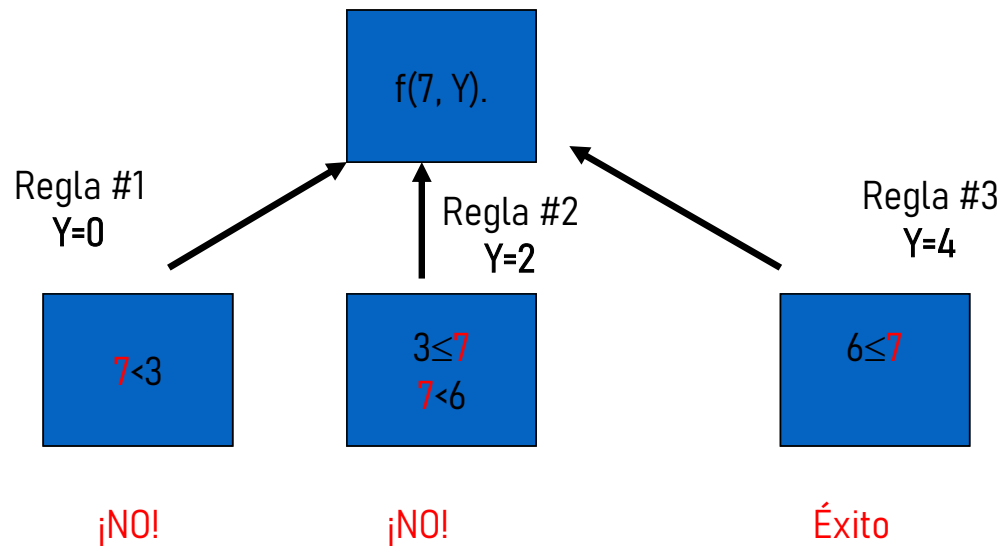
$f(X, 4) :- X \geq 6, !.$

- Ambas definiciones arrojan el mismo resultado.
- Sin embargo, la segunda versión con CUT es más eficiente dado que reconoce antes que la evaluación ha fallado.
- Se puede decir que el CUT ha modificado el *significado procedural* del programa (en este caso, no ha sido modificado el *significado declarativo*).

# *Backtracking.* Ejemplo de una Función y operador Corte

Evaluación de la Meta

?-  $f(7, Y).$   
 $Y=4$



¡¡No se alcanza el CUT en ninguna evaluación!!

## *Backtracking.* Ejemplo de una Función y operador Corte

% Regla #1

$f(X, 0) :- X < 3, !.$

- Ambas definiciones arrojan el mismo resultado.

% Regla #2

$f(X, 2) :- X < 6, !.$

- Sin embargo, la segunda versión con CUT es más eficiente dado que reconoce antes que la evaluación ha fallado.

% Regla #3

$f(X, 4) .$

$-? f(7, Y).$

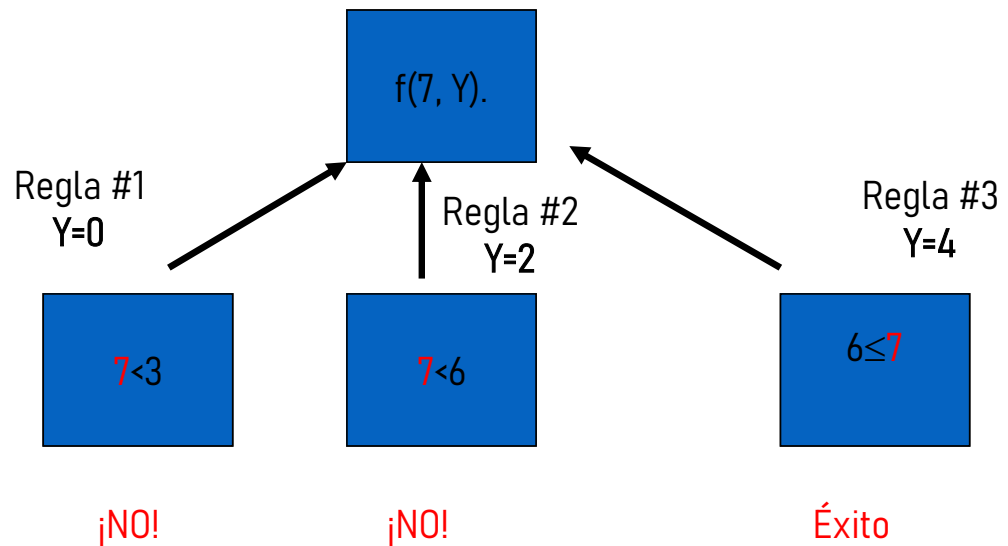
$Y=4$

- Se puede decir que el CUT ha modificado el *significado procedural* del programa (en este caso, no ha sido modificado el *significado declarativo*).

# *Backtracking.* Ejemplo de una Función y operador Corte

Evaluación de la Meta

?-  $f(7, Y).$   
 $Y=4$



¡¡Se reduce el número de evaluaciones!!



## *Backtracking.* Ejemplo de una Función y operador Corte

- Esta optimización sólo funciona con el operador de corte.
- Si se sacaran los cortes y se evalúa  $f(1, Y)$ , se producirían varios resultados.
- En este caso se dice que afecta el significado declarativo.

## *Backtracking.* otros ejemplos con el operador Corte

% La función de membresía que se vio es:

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

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

%se puede transformar en una versión con un corte

% que sólo permite encontrar un solo miembro.

```
member(X, [X | L]):- !.
```

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

## Negación: como Falla

- A María le gustan los animales:

`gusta(maria, X) :- animal(X).`

*¡¡Pero no le gustan las serpientes!!*

- Expresado en Prolog:

`gusta(maria, X) :- serpiente(X), !, fail;  
                  animal(X).`

## Negación: Ejemplos

- La verificación de si dos expresiones difieren:  
`diferente(X, Y) :- X = Y, !, fail;`  
`true.`
- El procedimiento interno not de Prolog se comporta como:  
`not(P) :- P, !, fail;`  
`true.`

## Negación: Aplicación

- La función diferente ahora se puede escribir como:  
`diferente(X, Y) :- not(X = Y).`
- y la regla sobre los gustos de **maria** se escribe como:  
`gusta(maria, X) :- not(X = serpiente), animal(X).`

# Negación: y uso del operador de Corte

- **Ventajas:**

- Se puede aumentar la eficiencia.
- Se pueden expresar reglas que son mutuamente excluyentes.

- **Desventajas:**

- Se pierde correspondencia entre significado declarativo y procedural.
- Cambio del orden de las cláusulas puede afectar significado declarativo.

# Unidad 7

## Programación Lógica y Prolog

FIN

- 7.1 Introducción a Prolog
- 7.2 Tipos de Datos en Prolog
- 7.3 Calce de Términos en Prolog
- 7.4 Listas y Operadores
- 7.5 Operadores y Aritmética
- 7.6 *Backtracking*