# Paradigma Lógico

## Módulo 2:
## Variables.
## Consultas existenciales.
## Reglas simples y compuestas.
## Inversibilidad.

**por Fernando Dodino**
**Carlos Lombardi**
**Nicolás Passerini**
**Daniel Solmirano**

**revisado por Matías Freyre**

**Versión 2.1**
**Junio 2019**

Distribuido bajo licencia Creative Commons Share-a-like

**Contenido**

# 1 Variables

Volvamos a nuestro ejemplo inicial de las pastas:

```
pastas(ravioles).
pastas(fideos).
```

En las consultas, no solo podemos preguntar si los fideos son pastas, sino también preguntar qué pastas hay en nuestra base de conocimientos:

```
? pastas(Pasta).
Pasta = ravioles _
```

¿Qué diferencia hay entre la consulta `pastas(ravioles)` y `pastas(Pasta)`?

*Pasta* comienza con mayúscula, entonces en lugar de trabajar con un individuo, la consulta se hace con una **variable** o incógnita. Esto significa que no conocemos los individuos que satisfacen la relación pastas/1, y los queremos determinar.

La pregunta pastas(Pasta) se puede leer como

> "¿Cuáles son los individuos que satisfacen el predicado pastas/1?"

## 1.1 Variables anónimas

Si nuestro objetivo es determinar si existe alguna pasta, sin importar cuáles son las pastas concretas que forman parte del universo, la consulta se puede escribir así:

```
? pastas(_).
true
```

Esto implica que debe existir al menos un individuo que satisfaga la relación **pastas/1**. Cuando no nos importe unificar el valor de un individuo que participa de un predicado con una variable, utilizamos el caracter guión bajo (_) y lo llamamos **variable anónima**.

# 2 Múltiples soluciones

En este caso el motor de inferencia Prolog no solo puede inferir si una relación se satisface para ciertos individuos, sino también cuál es el universo de

individuos que cumplen una relación. Solo que lo hace un individuo a la vez, por eso aparece el prompt luego de mostrar el individuo ravioles...

```
? pastas(Pasta).
Pasta = ravioles _
```

Para que siga buscando soluciones, podemos presionar el caracter ; o bien n (next). Si queremos detener la búsqueda, presionamos el caracter . o bien <Enter>.

```
? pastas(Pasta).
Pasta = ravioles ;
Pasta = fideos _
```

De la misma manera, si tenemos esta base de conocimientos:

```
come(juan, ravioles).
come(melina, ravioles).
come(brenda, fideos).
come(juan, fideos).
```

Podemos consultar quiénes comen ravioles:

```
? come(Persona, ravioles).
Persona = juan ;
Persona = melina ;
false
```

Al final se muestra false, lo que se puede leer como “no fue posible encontrar más individuos que satisfagan la relación come/2”.

De forma similar, también podemos consultar qué comidas come Juan:

```
? come(juan, Comida).
Comida = ravioles ;
Comida = fideos ;
false
```

Esta es una característica interesante de Prolog, que nos permite hacer consultas donde en cada argumento pasemos

- **individuos**, en ese caso se dice que están *instanciados*
- o **variables**, en ese caso se dice que están libres

### 2.1 Tipos de consulta

Diferenciamos entonces dos tipos de consulta:

- aquellas en las que queremos determinar si determinada relación se satisface o no, instanciando todos los argumentos:

```
? come(juan, ravioles).
? pastas(bohio).
? pastas(_).
```

  Esto se verifica (true) o no (false).

- las consultas **existenciales**, permiten conocer los individuos que satisfacen una relación, en ese caso alguno de los argumentos debe estar libre (debe ser una variable).

```
? come(juan, Comida).
? come(Persona, Comida).
? pastas(Pasta).
```

Entonces el motor de inferencia Prolog unifica los posibles individuos que satisfagan la relación, si es que existen.

## 3 Inversibilidad

Decimos que un predicado es **inversible** cuando admite consultas con variables libres para sus argumentos: en el caso de los hechos no hay restricciones así que tanto come/2 como pastas/1 son totalmente inversibles.

Más adelante veremos que no siempre es posible hacer consultas existenciales para todos los argumentos, en ese caso tendremos predicados que serán inversibles para el primer, segundo o cuarto argumento.

## 4 Reglas

Recordemos algunos predicados que escribimos hasta el momento:

```
come(juan, ravioles).
come(melina, ravioles).
```

Este tipo de predicados se llaman **hechos**, y se consideran axiomas: cuando hacemos la consulta `come(juan, ravioles)` el motor de inferencia asume que

si existe un hecho escrito en la base de conocimientos, éste se cumple.

Prolog también permite trabajar con otro tipo de predicados más interesante: las **reglas**, que explicaremos a continuación con el siguiente silogismo:

Todo humano es mortal. (cuantificación universal)
Sócrates es humano (cuantificación particular)
Ergo, Sócrates es mortal. (cuantificación particular)

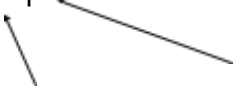
Esto en lógica, se escribe:
$\forall(x)(humano(x) \Rightarrow mortal(x))$
*humano(socrates)*
*entonces mortal (socrates)*

Escribimos en la base de conocimientos:

```
mortal(Persona):-humano(Persona).
humano(socrates).
```

Una regla tiene
- uno o más antecedentes: en este caso es uno solo, humano/1
- un consecuente

Si se cumplen los antecedentes, entonces se satisface el consecuente. Lo que en lógica se escribe p => q, en sintaxis PROLOG se escribe al revés:
q :- p.



consecuente :- antecedente(s). Si se cumplen los antecedentes, entonces se cumple el consecuente. Esta forma de escribir los predicados se llama *cláusula de Horn*.

Dos consecuencias del ejemplo anterior:
- un predicado puede ser
  - un hecho: no tiene antecedentes y se considera cierto porque está escrito en la base de conocimientos
  - o una regla: tiene al menos un antecedente y se satisface para un individuo si éste cumple todos los antecedentes
- no tiene sentido escribir `mortal(socrates)` porque esa información se puede inferir a partir de la base de conocimientos. Aquí empieza a verse alguna de las ventajas de trabajar en el paradigma.

## 4.1 Consultas sobre una regla

Algunas preguntas que puedo hacer:

- Sócrates, ¿es mortal?

```
?- mortal(socrates).
true
```

- Maradona, ¿es mortal?

```
?- mortal(maradona).
false
```

Ya sabemos que, por principio de universo cerrado, todo lo que no forma parte de la base de conocimientos se presume falso. El predicado `mortal(maradona)` implica que `humano(maradona)`. Como no lo podemos probar, `humano(maradona)` *falla* (no se puede satisfacer) y por lo tanto también la consulta `mortal(maradona)`.

- ¿Qué individuos satisfacen **mortal/1**?

```
?- mortal(Persona).
Persona = socrates
true
```

- ¿Existe algún individuo  que satisfaga **mortal/1**?

```
?- mortal(_).
true
```

El predicado mortal/1 ¿es **inversible**?  Sí, ya que admite consultas por individuo o variables en todos sus argumentos.

## 4.2 Declaratividad

Recordemos que cuando tenemos una solución declarativa

- el concepto de secuencia pierde importancia
- y delego el algoritmo a un componente externo, que en el caso de Prolog es el motor de inferencia

Efectivamente no hay estructuras de control en nuestra base de conocimiento, solo reglas y hechos que permiten a Prolog inferir conocimiento. Entender cómo

funciona ese motor de inferencia está fuera del alcance de la materia, el lector interesado puede investigar el *Principio de Resolución de Robinson*[1] para ver cómo se implementa el motor. Nos posicionamos como usuarios de ese motor, y nos basta saber que si incorporamos a la base de conocimientos

```
humano(platon).
```

Cuando hagamos la consulta
`mortal(Persona).`
tendremos ahora dos individuos que satisfagan esa relación

```
Persona = socrates
Persona = platon
```

Recordemos que

- Persona comienza con mayúscula → es una variable (una incógnita)
- platon comienza con minúscula → es un valor (un individuo)

**¿Cómo encuentra las soluciones? No nos interesa.** Sólo vamos a decir que con un mecanismo que se llama *backtracking*[2], así es como encuentra todas las unificaciones posibles para una variable.

## 4.3 Unificación

Cuando hacemos la consulta

```
mortal(Persona).
```

Y nos aparece

```
Persona = socrates
Persona = platon
```

¿qué está pasando? ¿hay una asignación a una variable? No, en Prolog no existe el concepto **asignación**, sino **unificación**: donde en este caso una variable se resuelve como incógnita con uno o más valores.
`Persona = socrates`
Persona es la incógnita, socrates el valor.

[1] escrito por John Alan Robinson: http://dl.acm.org/citation.cfm?id=321253. Más info: http://mathworld.wolfram.com/ResolutionPrinciple.html

[2] Intuitivamente el mecanismo de backtracking puede explicarse en este didáctico video y también en la infalible Wikipedia

Pero es importante recalcar que las consultas **no devuelven nada**: pueden satisfacerse o no, puedo encontrar los individuos que cumplen un predicado, pero de ninguna manera el motor va a "devolver" valores. No puedo utilizar esos valores como si estuviera en una solución imperativa, solamente puedo seguir construyendo reglas para unificar esos valores y hacer más preguntas.

Si entendimos que unificación es diferente de asignación, entenderemos que
`X = X + 1`
es una **condición**, y en particular una que nunca se puede cumplir, ya que es una contradicción en sí misma como regla lógica: si X es 2, no puede ser 3...

# 5 Reglas compuestas

## 5.1 Conjunciones (AND)

Si tenemos estos hechos en la base de conocimientos:

```
viveEn(tefi, lanus).
viveEn(gise, lanus).
viveEn(alf, lanus).
viveEn(dodain, liniers).
docente(alf).
docente(tefi).
docente(gise).
docente(dodain).
```

"Cualquier docente que vive en Lanús es un afortunado"

Aquí tenemos una regla compuesta:
p ∧ q => r

donde p = docente
∧ = conector lógico AND
q = vive en Lanús
r = es afortunado

Pasamos el predicado al formato de cláusula de Horn, donde el conector lógico AND se representa con una coma:

```
afortunado(Persona):-docente(Persona), viveEn(Persona, lanus).
```

Persona es una variable, que se unifica para docente/1 y viveEn/2. Cuando se verifica docente(alf), se busca satisfacer viveEn(alf, lanus), con éxito.

En cambio para el caso docente(dodain) no se cumple viveEn(dodain, lanus), por lo que la regla compuesta afortunado(dodain) no se cumple.

Entonces

- la variable Persona se unifica para todos los individuos que satisfacen docente/1.
- una vez unificada la variable, al tratar de satisfacer viveEn/2 ya no hay incógnitas: Persona es un valor conocido, aquel que encontramos como una solución posible de docente(Persona), y lanus es un individuo.

El predicado afortunado/1 es inversible, como podemos comprobar...


## 5.2 Disyunciones (OR)

"Si una persona es docente o vive en Lanús es afortunada"

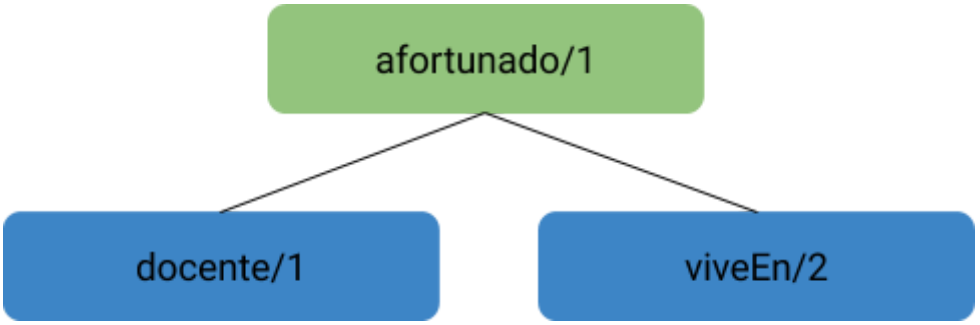
Ahora cambiamos el conector de nuestra regla compuesta.
p ∨ q => r

donde p = docente
∨ = conector lógico OR
q = vive en Lanús
r = es afortunado

Lo pasamos al formato de Prolog, repitiendo las cláusulas en dos reglas diferentes:

```
afortunado(Persona):-docente(Persona).
afortunado(Persona):-viveEn(Persona, lanus).
```

Entonces por cualquiera de las dos ramas que cumpla el antecedente docente/1 o viveEn/2, la regla afortunado/1 se satisfará.




Si pensamos en afortunado/1, docente/1 y viveEn/2 como componentes, entenderemos que afortunado depende de las definiciones de docente y viveEn, es decir que hay un cierto grado de acoplamiento entre ellos: cualquier modificación en docente o viveEn puede impactar en afortunado.

# 6 Definición por comprensión

Así como anteriormente habíamos visto que un conjunto de hechos para el mismo predicado forman la definición por extensión de dicho predicado:

```
animal(tigre).
animal(oso).
animal(elefante).
```

Una regla forma la definición por comprensión del predicado:
a) $A = \{a \in \mathbb{N} / 2 < a < 6\}$
b) $B = \{p \in \mathbb{N} / p < 10$, siendo $p$ par$\}$

Si lo definimos en un lenguaje lógico como PROLOG:

```
A(X):- X > 2, X < 6.
B(X):- X < 10, par(X).
```

# 7 Práctica

## 7.1 Primer ejemplo: Gustos

- Juan gusta de María.
- Pedro gusta de Ana y de Nora.
- Todos los que gustan de Nora gustan de Zulema.
- Julián gusta de las morochas y de las chicas con onda.
- Mario gusta de las morochas con onda y de Luisa.
- Todos los que gustan de Ana y de Luisa, gustan de Laura.
- Después cambiar ese "y" por un "o".

Resolvemos cada uno de los puntos:
*"Juan gusta de María"*

```
gusta(juan, maria).
```

¿Es un predicado monádico o poliádico? Poliádico, porque expresa una relación entre juan y maría. Recordemos que juan y maria van en minúscula, porque son los individuos (átomos) que componen la relación.

*"Pedro gusta de Ana y de Nora"*
El castellano es engañoso, a veces digo "y" cuando quiero decir "o".

```
gusta(pedro, ana).
gusta(pedro, nora).
```

Es decir "Pedro gusta de Ana" es verdadero, lo mismo que "Pedro gusta de Nora".
Entonces en realidad esto es un or; repetir la cláusula es otra forma de decir:
gusta(pedro, ana) or gusta(pedro, nora) se cumple.

*Todos los que gustan de Nora gustan de Zulema* quiere decir: a alguien le gusta Zulema si le gusta Nora.
`p => q` lo transformamos a `q si p`.

```
gusta(Alguien, zulema) :- gusta(Alguien, nora).
```

*"Julián gusta de las morochas y de las chicas con onda"*
Si una chica tiene onda, ¿le gusta a Julián? ¿importa si es morocha o rubia?
Si lo doy vuelta se entiende mejor: Julián gusta de una chica si es morocha o si tiene onda.

```
gusta(julian, Chica):- chica(Chica), morocha(Chica).
gusta(julian, Chica):- chica(Chica), tieneOnda(Chica).
```

*"Mario gusta de las morochas con onda y de Luisa"*

```
gusta(mario, Chica):-chica(Chica), morocha(Chica),
     tieneOnda(Chica).
gusta(mario, luisa).
```

*"Todos los que gustan de Ana y de Luisa, gustan de Laura"*
Si es con "y":

```
gusta(Alguien, laura) :- gusta(Alguien, ana), gusta(Alguien, luisa).
```

Y si es con "o":

```
gusta(Alguien, laura) :- gusta(Alguien, ana).
gusta(Alguien, laura) :- gusta(Alguien, luisa).
```

### 7.2 Segundo Ejemplo: Familia

Dada la siguiente base de conocimientos:

```
progenitor(homero, bart).
progenitor(homero, lisa).
progenitor(homero, maggie).
progenitor(abe, homero).
progenitor(abe, jose).
progenitor(jose, pepe).
progenitor(mona, homero).
progenitor(jacqueline, marge).
progenitor(marge, bart).
progenitor(marge, lisa).
progenitor(marge, maggie).
```

Resolver los predicados hermano, tío, primo y abuelo.

```
hermano(Hermano1, Hermano2):-progenitor(Padre, Hermano1),
      progenitor(Padre, Hermano2), Hermano1 \= Hermano2.
```

El operador \= en PROLOG permite comparar dos individuos:

- si ambos son iguales, el operador no se satisface (falla)
- si son distintos, el operador se cumple.

```
tio(Sobrino, Tio):-progenitor(Padre, Sobrino), hermano(Padre, Tio).
      ==> si el primer argumento es el sobrino y el segundo es el tío.
tio(Tio, Sobrino):-hermano(Tio, Padre), progenitor(Padre, Sobrino).
      ==> si el primer argumento es el tío y el segundo el sobrino.
```

```
primo(Primo1, Primo2):-
      progenitor(Padre, Primo1),
      progenitor(Tio, Primo2),
      hermano(Padre, Tio).

abuelo(Abuelo, Nieto):-
      progenitor(Abuelo, Padre), progenitor(Padre, Nieto).
```

¿Quiénes son los abuelos de bart?

```
? abuelo(Quien, bart).
Quien = abe
```

¿Qué pasó en el medio?

El motor de inferencia de PROLOG **unificó** la variable Quien con el valor "abe". Las variables eran...
...**incógnitas** que el motor trata de resolver dentro del programa lógico.

Cuando hay múltiples resultados las variables se van unificando a distintos valores, como es el caso de:

```
? progenitor(homero, Quien).
```

Acá el motor encontrará 3 resultados posibles para Quien:

```
Quien = bart ;
Quien = lisa ;
Quien = maggie ;
false
```

Pregunta para el lector: ¿son inversibles los predicados que definimos?

## 8 Resumen del capítulo

Además de los valores, el paradigma lógico trabaja con el concepto de variable que permite modelar una incógnita. Los predicados inversibles permiten hacer consultas dejando incógnitas en los argumentos. En ese caso Prolog puede determinar la existencia de individuos que satisfacen una relación, por este motivo se denominan consultas existenciales.

En la base de conocimientos no solo hay hechos o axiomas, sino también reglas, definidas mediante una o más condiciones. Cuando todas se satisfacen permiten verificar que una determinada regla se cumple. En una regla podemos relacionar varios antecedentes mediante conectores lógicos and/or y variables que muestran el poder del motor de inferencia Prolog para encontrar todas las soluciones posibles a un problema.