# Introducción al paradigma lógico

Comenzaremos a estudiar un nuevo paradigma, el paradigma lógico. Recordemos que los paradigmas nos dan un marco conceptual para resolver problemas; en el caso de la programación, nos dicen cómo modelar las estructuras de datos y los algoritmos.

Hemos visto anteriormente dos visiones distintas para construir software:

* En la visión **imperativa o procedimental** yo defino una secuencia de pasos a ejecutar y guardo estados intermedios en posiciones de memoria que llamo variables. Esto respeta fielmente el modelo matemático de Von Neumann.
* En la visión **declarativa** no hay algoritmo. Sólo definiciones (el qué) y hay alguna magia detrás de todo esto que lo termina resolviendo. Si esa magia es transparente => me puedo concentrar en lo que es esencial al problema (ése es el beneficio de la abstracción).

# Base de conocimiento

Define el alcance de lo que estamos modelando (que forma parte de nuestro universo de cosas conocidas) y lo que no.

pastas(ravioles).

pastas(fideos).

*comidas.pl --> nuestra base de conocimientos*

# Predicados e individuos

La base de conocimiento se conforma de predicados: el predicado *pastas* tiene aridad 1, esto significa que un solo individuo participa de la relación, por eso se escribe **pastas/1**. Los predicados que tienen un solo argumento se los llama monádicos, y expresan características o atributos de los individuos en cuestión.

Los individuos son los elementos que forman parte del universo posible de los predicados, en el ejemplo anterior tanto ravioles como fideos son individuos posibles que satisfacen la relación pastas/1. Es natural asociar la idea de individuo a dato, más adelante profundizaremos en este aspecto.

# Consultas

La **lógica** es una ciencia formal que estudia los principios de la demostración e inferencia válida. La inferencia es el proceso por el cual se derivan conclusiones a partir de premisas. Cuando ejecutamos un programa lógico nos aparece una consola para disparar consultas que toman como entrada las definiciones de la base de conocimiento para extraer conclusiones.

? pastas(ravioles).

true

? pastas(fideos).

true

Es importante el punto para delimitar el final de una consulta, de lo contrario en la consola Prolog nos aparecerá un símbolo pipe ( | ) que indicará que podemos seguir escribiendo la consulta en más de una línea

# Una aplicación en el paradigma lógico

Una aplicación en el Paradigma Lógico es una solución **declarativa** porque

* declaro conocimiento a través de los predicados, en un archivo .pl, utilizando cualquier editor de texto
* en las consultas, un motor de inferencia (en este caso de Prolog) permite sacar conclusiones a partir de ese conocimiento. El motor es el que termina construyendo el algoritmo, para poder resolver las consultas.

# Principio de Universo Cerrado

? pastas(pechitoCerdo).

false

? pastas(gnocchis).

false

Vemos que Prolog nos dice que ni el pechito de cerdo ni los ñoquis son pastas. Esto puede verse de dos maneras:

* los ñoquis no son pastas: la afirmación “los ñoquis son pastas” es **falsa**
* no puedo probar que los ñoquis sean pastas: la afirmación “los ñoquis son pastas” es **desconocida**.

Al comenzar a escribir nuestra base de conocimientos contamos que estábamos definiendo nuestro alcance, el universo de los elementos conocidos, por lo tanto, hay dos opciones:

* considerar tres estados posibles: cierto, falso y desconocido: estos son los lenguajes que trabajan con el **Principio de Universo Abierto**
* considerar a todo lo desconocido como falso: los lenguajes como Prolog entran en esta categoría ya que trabaja con el **Principio de Universo Cerrado**

Vamos a llamar *hecho* a una afirmación codificada en la base de conocimiento, con la sintaxis que vimos antes, y que por lo tanto asumimos como verdadera. Cada hecho es una cláusula.

# Predicados poliádicos

Hemos visto anteriormente que los predicados monádicos son los que tienen aridad 1. Los predicados que tienen más de un argumento se llaman poliádicos, porque expresan relaciones entre individuos.

**come**(juan, ravioles). **come**(brenda, fideos). **gusta**(brenda, fideos).

**Tip:** es conveniente escribir las definiciones de un mismo predicado en forma contigua, y no intercalar definiciones de otros predicados. Si la base de conocimientos se escribe así:

## Consultas sobre predicados poliádicos

Aquí vemos que

* las consultas deben respetar la cantidad de argumentos con la que se definió el predicado. De lo contrario veremos un mensaje de error.
* el orden de los argumentos es importante: gusta(fideos, brenda) se satisface mientras que no es posible probar la relación gusta(brenda, fideos).
  + Las relaciones no son bidireccionales por defecto, esto debemos tenerlo en cuenta a la hora de hacer nuestras definiciones.
  + El sentido de los argumentos se los damos nosotros, también hay que estar atento al lugar que ocupa cada uno de los individuos involucrados en un predicado.

# Definición por extensión y por comprensión

Un conjunto de hechos para el mismo predicado forma la definición por extensión del predicado.

**animal**(tigre). **animal**(oso). **animal**(elefante).

Es como decir Animales = {tigre, oso, elefante}

¿Cuál es la desventaja de este tipo de definición? Requiere la enumeración de todos los elementos que componen el conjunto. Para conjuntos con muchos elementos es, cuanto menos, incómodo.

Otra desventaja que presenta es que no brinda información referente al conjunto más allá de la que el observador le puede dar.

# ¿Dónde aplicamos la lógica en computación?

* Bases de datos: lenguajes de restricciones, lenguajes de consulta (SQL)
* Inteligencia artificial: representación del conocimiento, deducción.
* Ingeniería de software: especificación de sistemas (lenguajes Z)
* Criptografía: verificación de protocolos criptográficos
* Procesamiento del lenguaje natural

# Variables

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?”

## 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.

# 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 **\_**

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*
* **variables**, en ese caso se dice que están libres

## 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. 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).

# 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.

# Reglas

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

Todo hombre es mortal (cuantificación universal)

Sócrates es humano (cuantificación particular)

Ergo, Sócrates es mortal (cuantificación particular)

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.

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
  + 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.

## 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.

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

## Unificación

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.

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.

# Reglas compuestas

## 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).

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).

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.

## Disyunciones (OR)

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.

# Definición por comprensión

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

**animal**(tigre). **animal**(oso). **animal**(elefante).

Si lo definimos en un lenguaje lógico como PROLOG:

**A**(X):- X > 2, X < 6.

**B**(X):- X < 10, par(X).

# Individuos simples

## Números

Los números se utilizan como literales:

**ingrediente**(**1**, pollo).  
**nota**(salvatelli, **8**).

En este caso tenemos 1 y 8 como valores literales. Los números, además de poder compararse por igualdad/desigualdad definen

* operadores aritméticos: la suma (Valor + 2), la resta (Valor - Monto), la multiplicación (3 \* 4), la división (Numero / 2), el valor absoluto (abs(-2)), entre otros
* operaciones de comparación por orden: <, <=, >, >=

En resumen, hay restricciones de inversibilidad cuando trabajamos con estos predicados

* Comparaciones (>, <, >=, <=, =)
* Operaciones aritméticas (+, -, \*)
* Operador is/2

## Strings

También podemos trabajar con cadenas de caracteres, sobre todo cuando necesitamos modelar un individuo que tiene espacios:

**escritor**("Jorge Luis Borges").  
**escritor**("Julio Cortázar").  
**escritor**("Elsa Bornemann").

# Pattern matching

La búsqueda exacta de patrones al enviar argumentos o pattern matching que hemos visto en el Paradigma Funcional también lo encontramos en el Paradigma Lógico, pero debemos estar al tanto de la diferencia entre una función y una relación.

valor(0, 1).  
valor(Incognita, Incognita).

Al hacer la consulta

?- valor(0, Cual).  
Cual = 1 ;  
Cual = 0.

Vemos que 0 unifica:

* con el valor 0 de la primera definición del predicado valor/2
* también con la variable Incognita, que matchea cualquier valor.

¿Por qué se produce esto? Porque una relación es más abarcativa que una función, entonces el motor prueba unificar cada uno de los patrones existentes. Entonces vemos que, si bien tanto el paradigma funcional como el lógico tienen pattern matching, en el primero se devuelve la primera expresión donde el patrón coincida, mientras que en lógico por la naturaleza del paradigma se buscan todos los individuos que satisfacen un predicado.

Prolog no hace chequeo de tipos, cualquier individuo encaja como patrón de una variable sin ligar:

?- valor("cero", Cual).  
Cual = "cero".

Es decir, la variable Incógnita puede ser cualquier individuo, un número, un string, un booleano, etc.

# Individuos compuestos

## Listas

Las listas representan una serie de elementos ordenados, que pueden repetirse.

La lista es una estructura recursiva definido de la siguiente manera:

* el caso base es la lista vacía, que se denota [ ]
* el caso recursivo es una lista con al menos un elemento, que se divide en el primer elemento o cabeza y el resto llamado cola, que es una lista (aquí vemos la definición recursiva). Se denota con el patrón: [**Cabeza**|**Cola**]

## Operaciones básicas sobre listas

A continuación, vamos a aprender algunos predicados que serán útiles a la hora de trabajar con listas. No nos interesa entender cómo se construyen, sino saber utilizarlos convenientemente para resolver los requerimientos que nos pidan de aquí en más.

### Length

length/2 es un predicado que relaciona una lista con su longitud.

?- **length**([picasso, vanGogh, dali], **3**).  
true.  
  
?- **length**([picasso, vanGogh, dali], Cantidad).  
Cantidad = **3**.

### Member

Member/2 verifica si un elemento está en una lista. Algunas consultas posibles:

?- member(**5**, [**5**, **8**]).  
**true**   
  
?- member(Valor, [**5**, **8**]).  
Valor = **5** ;  
Valor = **8**.

?- member(**4**, [**5**, **8**]).  
**false**

Si la lista es una incógnita, el predicado member/2 pierde sentido

Member/2 es un predicado inversible para el primer argumento.

### Append

El predicado append/3 permite relacionar dos listas con la lista concatenada.

?- append([**1**, **3**], [**2**], Resto).  
Resto = [**1**, **3**, **2**].  
  
?- append([**1**, **3**], [**2**], [**1**, **2**, **3**]).  
false.  
  
?- append([**1**, **3**], SegundaLista, [**1**, **3**, borges]).  
SegundaLista = [borges].

?- append(PrimeraLista, SegundaLista, [**1**, **3**, borges]).  
PrimeraLista = [],  
SegundaLista = [**1**, **3**, borges] ;  
PrimeraLista = [**1**],  
SegundaLista = [**3**, borges] ;  
PrimeraLista = [**1**, **3**],  
SegundaLista = [borges] ;  
PrimeraLista = [**1**, **3**, borges],  
SegundaLista = [] ;  
false.

Como vemos Prolog puede satisfacer cuáles son los individuos que cumplen esta relación de diferentes maneras.

El predicado append/3 es inversible y tiene sentido para estos casos:

* el primer argumento como incógnita
* el segundo argumento como incógnita
* el tercer argumento como incógnita
* el primer y segundo argumento como incógnita:

### nth/3

Este predicado relaciona un elemento con la posición que ocupa en una lista. Tenemos dos predicados: nth0/3 que toma el índice partiendo de 0 y nth1/3 que considera el índice partiendo de 1 como primer elemento.

?- nth0(**2**, [**1**, **2**, **3**, **4**, **5**], Elemento).  
Elemento = **3**.  
  
?- nth1(**2**, [**1**, **2**, **3**, **4**, **5**], Elemento).  
Elemento = **2**.

### last/2

Last relaciona una lista con su último elemento

?- last([**1**, **2**, **3**, **4**, **5**], Elemento).  
Elemento = **5**.  
?- last([**1**, **2**, **3**, **4**, **5**], **4**).  
false

### reverse/2

Este predicado se verifica si los elementos de una lista están al reverso en la segunda.

?- reverse(X, [**1**, **2**, **3**]).  
X = [**3**, **2**, **1**] .  
  
?- reverse([**1**, **2**, **3**], [**1**, **2**, **3**]).  
false.

?- reverse([**1**, **2**, **3**], X).  
X = [**3**, **2**, **1**].

### sum\_list/2

Relaciona una lista de números con el total que suman esos números.

?- sum\_list([7,2,6], Total).

Total = 15.

### list\_to\_set/2

Relaciona una lista de elementos repetidos con un conjunto sin repetidos.

?- list\_to\_set([2, 3, 2, 2, 3], Conjunto).

Conjunto = [2, 3].

### max\_member/2

Relaciona el mayor de una lista de elementos. También existe min\_member/2

?- max\_member(Mayor, [borges, auster, tolstoi]).

Mayor = tolstoi.

### subset/2

Relaciona un subconjunto de un conjunto de elementos.

?- subset([1, 2], [1, 2, 3]).

true.

## Pattern matching con listas

Las listas incorporan algunos patrones adicionales

|  |  |
| --- | --- |
| Pattern | Explicación |
| [ ] | una lista vacía.  Para [ ], el patrón coincide.  Para una lista con elementos, el patrón no coincide. |
| [ Elemento ] | una lista con un solo elemento  Para [ 8 ], Elemento vale 8 (el número 8).  Para [ ], el patrón no se satisface.  Para [1, 2], el patrón no se satisface. |
| [ Numero | Números ] | una lista con al menos un elemento que tiene cabeza y cola.  Para [1, 2, 3], la cabeza es el elemento 1,  y la cola = [ 2, 3 ].  Para [1], la cabeza es el elemento 1, y la cola [ ] (la lista vacía)  Para [ ], el patrón no coincide. |

## Functores

Los functores permiten agrupar información relacionada.

nacio(karla, fecha(22, 08, 1979)).

compro(cliente(231024, "Nelson Pedernera"), producto(pirufio, 239, 1)).

Un functor

* no es un predicado, si bien tienen formatos similares el functor no tiene un valor de verdad, no puedo preguntar? fecha(22, 08, 1979)
* tiene un nombre y una aridad que es la cantidad de individuos que lo componen
  + corolario: denota un individuo compuesto
  + dos functores con distinta aridad representan dos abstracciones diferentes
  + es equivalente a la tupla de Funcional, pero el nombre o prefijo permite entender más claramente el origen de los individuos que participan del functor (le da mayor expresividad)
* es especialmente útil cuando necesitamos trabajar elementos heterogéneos, que tienen algo común pero diferente información entre sí
* puede relacionar átomos, otros functores o listas

# Predicado not/1

Ahora tenemos la siguiente base de conocimientos de apostadores de ruleta:

juega(julia, 3).  
juega(beto, 6).  
juega(dodain, 5).

juega(juan, 15).  
juega(sergio, 3).

Y sabemos que un número es yeta si nadie apuesta a él: ∄x / p(x) se cumple

Puesto en términos de Prolog: yeta(Numero):-not(juega(**\_**, Numero)).

¿Qué aridad tiene el predicado not? 1.

Pero el argumento que recibe es un predicado. Entonces es un **predicado de orden superior**.

## Consultas

¿Qué pasa cuando intentamos una consulta existencial?

?- yeta(Numero).  
false.

La consulta no puede satisfacerse. ¿Por qué?

yeta(Numero):-not(juega(**\_**, Numero)).

Nosotros definimos en la base de conocimientos qué números juega la gente, pero no le decimos a Prolog cuál es el universo de números que podemos apostar a la ruleta. Y el motor de inferencia de Prolog solo está preparado para encontrar cuáles son todos los individuos que satisfacen una relación, no los que no la satisfacen.

## Inversibilidad

Si queremos que el predicado sea inversible, debemos ligar todas las variables del o de los predicados involucrados en el not/1. En nuestro caso solo tenemos que dar el universo de números.

yeta(Numero):-numero(**Numero**), not(juega(**\_**, **Numero**)).

Los números posibles son del 1 al 36. Para eso

* nos valemos de un predicado que relaciona dos números con una lista de números desde el primer número hasta el segundo
* y luego member/2 relaciona cualquier elemento posible con esa lista:

numero(N):-

  numlist(1, 36, NumerosRuleta),

  member(N, NumerosRuleta).

numero/1 es un predicado **inversible**, esto significa que admite tanto consultas sí/no como consultas existenciales:

Y de esa manera también podemos admitir consultas existenciales sobre los números yeta, porque numero/1 funciona como **predicado generador** del universo de números que se pueden apostar a la ruleta

# Predicado forall/2

Queremos verificar que una condición se cumpla para todas las variables posibles. Tenemos esta base de conocimientos:

|  |  |
| --- | --- |
| materia(pdp, 2). materia(proba, 2). materia(sintaxis, 2). materia(algoritmos, 1). materia(analisisI, 1). | nota(nicolas, pdp, 10). nota(nicolas, proba, 5). nota(nicolas, sintaxis, 8). nota(malena, pdp, 4). nota(malena, proba, 2). nota(raul, pdp, 9). |

“Un alumno terminó un año si aprobó todas las materias de ese año”

terminoAnio(Alumno, Anio):-   
 forall(materia(Materia, Anio), aprobo(Alumno, Materia)).  
  
aprobo(Alumno, Materia):-nota(Alumno, Materia, Nota), Nota >= 4.

# Lógica de conjuntos

## Incluido

Un conjunto A está incluido en otro B si todos los elementos de A están en B.

incluido(A, B):-forall(member(X, A), member(X, B)).

## Disjuntos

Un conjunto A es disjunto de B si se cumple que todos los elementos de A no están en B.

disjuntos(A, B):-forall(member(X, A), not(member(X, B))).

# Findall

padre(homero,bart).   
padre(homero,maggie).   
padre(homero,lisa).

Pensemos en predicados e individuos y definamos un predicado que relacione lo que nos interesa ... una persona y su cantidad de hijos. Dicho predicado puede llamarse cantidadDeHijos/2.

?- cantidadDeHijos(homero, Cantidad).   
Cantidad  = 3

Perfecto, ya tenemos definido nuestro objetivo ahora definamos el predicado a través de una regla (lo quiero hacer por comprensión porque quiero que me sirva para cualquier persona, no solo homero):

Si tenemos una forma de obtener múltiples respuestas a una consulta y lo que queremos es que todas esas respuestas estén juntas en una lista, hay un predicado en Prolog que hace exactamente eso:

findall(UnIndividuo, Consulta, Conjunto)

Entonces: findall es un predicado que relaciona

* un individuo o variable
* con una consulta
* y con el conjunto (lista) de los individuos que satisfacen la consulta.

cantidadDeHijos(Padre, Cantidad) :-   
 findall(Hijo, padre(Padre, Hijo), Hijos),  
 length(Hijos, Cantidad).

Observamos que la variable de la cláusula que define cantidadDeHijos llega ligada al findall, por lo tanto, la consulta que está adentro (el 2do parámetro) queda que efectivamente tiene 3 respuestas.

## Consultas más complejas

En el segundo parámetro del findall se puede definir cualquier tipo de consulta, no es necesario involucrar un único predicado.

findall(X,(p(X),q(X),r(X),...,s(X)), Xs)

Solo hay que encerrar entre paréntesis las cláusulas para no cambiar la aridad de findall que es tres (3).

## Intersección de conjuntos

interseccion(Xs,Ys,Zs) :- findall(E,(member(E,Xs),member(E,Ys)),Zs).

**Resumen**

Mientras que los predicados de primer orden definen características o relacionan individuos entre sí, los predicados de orden superior trabajan con predicados como argumentos. En particular, conocimos not/1, forall/2 y findall/3 que son predicados que permiten subir el grado de abstracción:

Mientras que el predicado forall/2 trabaja en base a la cuantificación universal: “para todos se cumple”, not/1 trabaja en base a la cuantificación individual negativa: “no existe x tal que cumpla...”, por lo tanto, son predicados que pueden usarse en forma indistinta, aunque trabajar con afirmaciones asertivas suele ser más cómodo que hacerlo negando los predicados.

Por último, para tener predicados inversibles se necesita ligar las variables que participan de las consultas mediante predicados generadores:

* en la negación lógica, porque no es posible en Prolog determinar los individuos que no satisfacen un predicado si no conocemos todo el universo.
* en las cláusulas findall y forall, porque debemos estar atentos a las variables libres y ligadas que participarán dentro de dichas cláusulas.

# Definición de recursividad

Recursividad es que un elemento de software se defina en función de sí mismo. La recursividad en lógico consistirá en definir un predicado en términos de sí mismo. En otras palabras:

|  |  |
| --- | --- |
| ¿Qué elementos tenemos en lógico? | Predicados / cláusulas. |
| ¿qué es un programa lógico? | (OK, un conjunto de cláusulas que) definen relaciones entre individuos |
| entonces, ¿qué va a ser recursivo? | la definición de un predicado |

## Ejemplo: ancestros

Buscamos primero la forma de contarlo declarativamente:

¿Quiénes son mis ancestros?

* Mis padres
* y los ancestros de mis padres.

Observar que estoy definiendo ancestro en términos de ancestro.

Recordemos la diferencia entre el castellano y la lógica - el "y" en castellano funciona como "o" lógico:

**ancestro**(Persona, Ancestro):-**padre**(Persona, Ancestro).  
**ancestro**(Persona, Ancestro):-  
 **padre**(Persona, Padre), **ancestro**(Padre, Ancestro).

# Predicado member/2

Un elemento está en la lista

* si está en la cabeza de la lista
* si está en la cola.

El chiste de esta definición es que como la cola es una lista, la definición del predicado member termina siendo recursiva:

member(X, [X|**\_**]).  
member(X, [**\_**|Z]):-member(X, Z).

¿Qué va a pasar si la lista está vacía?

Evidentemente no puede haber ningún elemento en una lista vacía, porque la lista vacía no es divisible en cabeza y cola, directamente no defino cláusulas y listo.

## Inversibilidad del predicado member

Esta definición permite hacer consultas dejando libre el primer argumento:

Por esto el predicado member/2 es inversible para el primer argumento. No tiene tanto sentido hacer una consulta existencial por el segundo argumento

# Predicado nth1/3

nth1/3 o enesimo/3 relaciona la posición que ocupa un elemento en una lista, considerando 1 la primera posición.

* El caso base es cuando el elemento está en la cabeza
* El caso recursivo es que, si el elemento no está en la cabeza, entonces ocupará la posición 1 + la posición que tenga en la cola.

enesimo(1, [Elemento|**\_**], Elemento).  
enesimo(Posicion, [**\_**|Resto], Elemento):-  
 enesimo(PosicionCola, Resto, Elemento),  
 Posicion is PosicionCola + 1.

**Resumen**

Prolog permite definir predicados recursivos, que deben tener

* un caso base o corte de la recursividad
* al menos un caso recursivo, donde el predicado esté definidos en términos de sí mismo.

Un aspecto interesante del motor de inferencia Prolog es que permite encontrar múltiples soluciones también para este tipo de predicados, mientras no haya restricciones de inversibilidad.

# Elementos de Diseño

¿Qué es diseñar?

* definir componentes
* qué responsabilidades tienen
* y qué relaciones hay entre esos componentes

En particular, dentro del paradigma lógico

* los componentes se implementan mediante predicados e individuos
* ¿qué responsabilidad tiene un componente? para member/2 relacionar un elemento que pertenece a una lista, para forall/2 determinar si todos los individuos que cumplen un predicado cumplen también

# Acoplamiento

El acoplamiento es el grado en que los componentes se conocen. Para explicarlo volvamos al predicado que resuelve cuándo un alumno terminó un año:

terminoAnio(Alumno, Anio):-   
 forall(materia(Materia, Anio), aprobo(Alumno, Materia)).

aprobo(Alumno, Materia):-nota(Alumno, Materia, Nota), Nota >= 6.

Hay acoplamiento entre terminoAnio/2 y los predicados materia/2 y aprobo/2. Este acoplamiento es buscado y necesario para poder resolver nuestro requerimiento.

A su vez, el predicado aprobo/2 se basa en el hecho nota/3. Si la forma de modelar las notas cambia y queremos incluir los aplazos

nota(raul, pdp, [9]).  
nota(herminio, pdp, [2, 2, 4]).

¿Qué predicados debemos cambiar en nuestra solución? aprobo/2, para considerar solamente la última nota:

aprobo(Alumno, Materia):-

   nota(Alumno, Materia, Notas),

   last(Notas, Nota),

   Nota >= 6.

No es necesario cambiar el predicado terminoAnio/2, ya que aprobó no cambió los individuos con los que trabaja. Esto muestra que el acoplamiento entre terminoAnio/2 y aprobo/2 es adecuado.

# Code smells

Los code smells son señales de que nuestra solución -aun funcionando- debería mejorarse. ¿Cómo? Incorporando abstracciones faltantes, utilizando conceptos, haciendo que el código acepte soluciones más generales o sea más expresivo, proceso que se suele llamar **refactorización**.

## Negación vs. Aserción

todosSiguenA(Rey) :- personaje(Rey), not((personaje(Personaje), not(sigueA(Personaje, Rey)))).  
  
sigueA(Alguien, Alguien).  
sigueA(lyanna, jon).  
sigueA(jorah, daenerys).

Bueno, el predicado todosSiguenA/1 es inversible y funciona correctamente. Pero ¿qué tan fácil de entender es? Un tanto complejo de decir que “todos siguen a un personaje si no existe otro personaje que no lo siga a él”. En lugar de trabajar con la doble negación, podemos simplificar lógicamente la solución.

∄x / p(x) ⇒ ￢q(x)

Se transforma a

∀x / p(x) ⇒ q(x)

“todos siguen a un personaje si ... todos los personajes siguen a ese personaje”

todosSiguenA(Rey): - personaje(Rey), forall((personaje(Personaje), sigueA(Personaje, Rey))).

El ejemplo anterior funcionaba y ciertamente no había ningún error conceptual, pero ahora se expone claramente cuándo se satisface el predicado todosSiguenA/1.

## Evitar duplicidades

Una ciudad es interesante si es antigua y tiene más de 10 puntos de interés copados

* un bar es copado si tiene más de 4 variedades de cervezas
* un museo de Ciencias Naturales es copado

baresCopados(Ciudad, Bares) :- findall(Bar, (puntoDeInteres(bar(CantVarCer), Ciudad), CantVarCer > 4), Bares).  
  
museosCopados(Ciudad, Museos) :- findall(Museo, puntoDeInteres(museo(cienciasNaturales), Ciudad), Museos).  
  
ciudadInteresante(Ciudad) :-  
 antigua(Ciudad),  
 baresCopados(Ciudad, Bares),  
 museosCopados(Ciudad, Museos),  
 length(Bares, CantidadBares),  
 length(Museos, CantidadMuseos),  
 CantidadLugaresCopados is CantidadBares + CantidadMuseos,  
 CantidadLugaresCopados > 10.

Claro en CiudadInteresante estamos sumando en forma individual museos y bares. ¿Qué podemos hacer? Trabajarlos en forma polimórfica

Además CantVarCer tiene un nombre poco representativo, cuando no tengamos el enunciado a mano va a ser difícil recordar qué estaba representando.

ciudadInteresante(Ciudad) :-  
 antigua(Ciudad),  
 cosasCopadas(Ciudad, Cosas),  
 length(Cosas, CantidadCosas),  
 CantidadCosas > 10.

cosasCopadas(Ciudad, Cosas) :- findall(Bar, (puntoDeInteres(Cosa, Ciudad), copada(Cosa)), Bares).  
  
copada(museo(cienciasNaturales)).  
copada(bar(VariedadesCerveza)):-VariedadesCerveza > 4.

* Quizá en otro lugar estemos necesitando la abstracción del predicado cosasCopadas/2
* Si además de museos y bares hay avenidas, centros culturales y edificios históricos copados, ¿en qué se ve afectado ciudadInteresante/1? En nada. Pero incluso al predicado cosasCopadas/2 tampoco lo afecta. Esta es una métrica real que se relaciona con mantener bajo el acoplamiento.
* Usemos nombres de variables desambiguados.

## Keep it simple

El error más común para quienes no están acostumbrados a pensar en términos del paradigma es armar listas cuando no son necesarias para la resolución del problema. Esto se pone en evidencia por el uso del findall seguido por un member sobre la lista resultante. El findall arma listas, el member las desarma... son operaciones inversas.

## Lazy predicate

Esto es frecuente de ver independientemente del paradigma en el que se trabaje, y lógico no es la excepción:

estaComplicado(Persona):-personaComplicada(Persona).  
personaComplicada(Persona):- ...

Saber delegar responsabilidades en un predicado es correcto, pero no está bien que un predicado no agregue valor. Los objetivos de los predicados estaComplicado/1 y personaComplicada/1 son los mismos, entonces hay un predicado que está de más, y hay un problema conceptual de ideas duplicadas.

## Inversibilidad

“Un viejo maestro es aquel pensador en el que todos sus pensamientos llegan a nuestros días”. La definición:

viejoMaestro(Pensador) :- forall(pensamiento(Pensador, Pensamiento), llegaANuestrosDias(Pensamiento)).

está ok, solo tiene un detalle: no es inversible. Recordemos que para que un predicado sea inversible tengo que ligar las variables, entonces utilizo un predicado generador:

viejoMaestro(Pensador) :- **pensador(Pensador),** forall(pensamiento(Pensador, Pensamiento), llegaANuestrosDias(Pensamiento)).

En todo caso el problema no es conceptual, sino una restricción que impone el motor de inferencia Prolog. Esta solución no calza en sí como un bad smell, o al menos está en una categoría diferente a las anteriores.

## Incógnitas innecesarias

El número de la suerte de una persona es su día de nacimiento y para Joaquín además es 8.

numeroDeLaSuerte(Persona, Numero) :- diaDelNacimiento(Persona, Numero).  
numeroDeLaSuerte(joaquin, Numero) :- Numero is 8.

Esto se ve bastante también, el uso de incógnitas innecesarias (lo que en otros paradigmas son variables locales de más). La segunda definición puede acortarse simplemente como:

numeroDeLaSuerte(joaquin, 8).

**Resumen**

En el presente apunte no hicimos más que exponer lo que en capítulos anteriores había surgido naturalmente: al resolver un requerimiento hemos tomado decisiones de diseño propias del paradigma lógico como la forma de modelar la información y de relacionar predicados e individuos.

Algunas soluciones se pueden considerar mejores que otras a partir de criterios objetivos, como el grado de acoplamiento que presentan los predicados, la simplicidad de éstos, la evitación del código duplicado, el uso de afirmaciones por sobre las negaciones, y otras características que se estudian a partir de *antipatterns* llamados *code smells*.

# mapList/3

mapList relaciona

* un predicado de aridad n (donde n >  1)
* una lista
* y otra lista que resulta de evaluar el predicado

?-maplist(length, [[1], [2, 6], []],  Transf).  
Transf = [1, 2, 0]  
?- maplist(succ, [5, 6, 7], Transf).  
Transf = [6, 7, 8]

## Inversibilidad

Como el primer argumento es el predicado de transformación, requiere cierto trabajo generar el conjunto de predicados built-in + los desarrollados por nosotros. Vamos a pensar que el predicado es obligatorio instanciarlo.

Pero podemos ver si Prolog es capaz de relacionar la lista original en base a la lista transformada:

?- maplist(succ, Original, [2, 8, 3]).  
Original = [1, 7, 2]  
  
  
?- maplist(length, Original, [2, 1]).  
Original = [[**\_**G400, **\_**G403], [**\_**G409]]

Interesante, ¿eh? Prolog nos pregunta si quiere buscar más soluciones: con lo cual podemos abrir el juego de maplist para pensarlo como una relación, en lugar de una simple función de transformación.

Y qué pasa cuando queremos hacer:

?- maplist(abs, X, [4, 2, 1]).  
ERROR: Arguments are not sufficiently instantiated

Ah… entonces, la inversibilidad ya no depende exclusivamente de los parámetros, sino que también necesitamos que el predicado sea inversible en los argumentos que nosotros queremos.

## Múltiples soluciones

El predicado between permite dejar el último argumento sin instanciar:

?- between(3, 5, X).

X = 3 ;

X = 4 ;

X = 5 ;

Entonces podemos utilizar mapList para generar la explosión combinatoria de los números que están entre 3 y 5 y entre 3 y 6

?- maplist(between(3), [5, 6], X).

|  |  |  |
| --- | --- | --- |
| X = [3, 3] ;  X = [3, 4] ;  X = [3, 5] ;  X = [3, 6] ; | X = [4, 3] ;  X = [4, 4] ;  X = [4, 5] ;  X = [4, 6] ; | X = [5, 3] ;  X = [5, 4] ;  X = [5, 5] ;  X = [5, 6] ; |

# Definiendo predicados de orden superior

En Prolog contamos con el predicado call/1 o call/\_ que permite evaluar un predicado pasado como parámetro:

? call(hijo(Padre, Hijo)).  
Padre = homero  
Hijo = maggie ;   
?- call(hijo(Padre, bart)).  
Padre = homero

Esto pemite subir el grado de abstracción de un predicado:

## Nuestro propio maplist

maplist(**\_**, [], []).  
maplist(PredicadoTransformador, [Orig|Origs], [Transf|Transfs]):-  
    call(PredicadoTransformador, Orig, Transf),  
    maplist(PredicadoTransformador, Origs, Transfs).

## Nuestro propio filter

Podemos pensar en un predicado que relacione los elementos originales de una lista con los que cumplen un determinado criterio. La primera versión la resolvemos con findall:

even(Numero):-0 is Numero rem 2.  
  
filter(Criterio, ListaOriginal, ListaNueva):-

  findall(Elem, (member(Elem, ListaOriginal), call(Criterio, Elem)), ListaNueva).

Existe una versión built-in de este filter que hicimos, y es **include/3**.

El predicado filter no es inversible para los primeros dos argumentos, dado que no podemos satisfacer call(Criterio, Elem) dado member(Elem, Original) sin conocer la lista original.

De hecho, el maplist podríamos haberlo definido en función al findall

maplistF(PredTransf, Original, Nueva):- findall(Result, (member(Elem, Original), call(PredTransf, Elem, Result)), Nueva)

Ahora qué pasa si definimos filter en forma recursiva:

filter2(**\_**, [], []).  
filter2(Criterio, [X|Original], [X|Nueva]):- call(Criterio, X), filter2(Criterio, Original, Nueva).  
filter2(Criterio, [**\_**|Original], Nueva):- filter2(Criterio, Original, Nueva).

Por otra parte, si hubiéramos considerado las últimas dos cláusulas excluyentes:

filter3(**\_**, [], []).  
filter3(Criterio, [X|Original], [X|Nueva]):- call(Criterio, X), filter3(Criterio, Original, Nueva).  
filter3(Criterio, [X|Original], Nueva):- not(call(Criterio, X)),filter3(Criterio, Original, Nueva).

# Efecto colateral en Lógico

## Assert + Retract

Nosotros podemos agregar conocimiento en forma dinámica, con el predicado assert/1:

?- assert(persona(juan)).

Esto trae un efecto colateral, dado que si ahora preguntamos si Juan es una persona la respuesta es sí:

?- persona(juan).

true

Ahora quitamos de la base de conocimientos el hecho que afirma que Juan es una persona:

?- retract(persona(juan)).

Y nuevamente vemos que hay un efecto colateral: al remover un hecho del cual depende, el predicado jugador se ve afectado:

?- jugador(X). false

## Listing + Dynamic

El predicado listing permite ver el conjunto de predicados definidos para la base de conocimientos. Entonces podemos ver que al agregar un predicado esa lista se ve afectada por un cambio de estado:

?- listing.

:- dynamic jugador/1.

El predicado retract/1 puede usarse con variables también, podemos remover así el predicado jugador:

?- retract(jugador(X)).

X = darth\_vader;