# Introducción al paradigma lógico

Tiene como característica principal la aplicación de las reglas de la lógica para inferir conclusiones a partir de datos. Conociendo la información y las condiciones del problema, la ejecución de un programa consiste en la búsqueda de un objetivo dentro de las declaraciones realizadas. Esta forma de tratamiento de la información permite pensar la existencia de “programas inteligentes” que puedan responder, no por tener en la base de datos todos los conocimientos, sino por poder inferirlos a través de la deducción.

La importancia del concepto de declaratividad en este paradigma, permite encuadrarlo dentro de los paradigmas declarativos. Al separar el control y la lógica, el programa se transforma en un conjunto de declaraciones formales de

especificaciones que deben ser correctas por definición.

Un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado, sino que está formado por expresiones que describen la solución (o más precisamente, la “declaran”). De esta manera, la clave para hacer un programa lógico es poder explicitar una declaración que describa correctamente la solución del problema.

## Características

El paradigma tiene sus fundamentos en las teorías de la lógica proposicional. De ellas, se toman en particular las **Cláusulas de Horn**, que son una forma de lógica de predicados con una sola conclusión en cada cláusula y un conjunto de premisas de cuyo valor de verdad se deduce el valor de verdad de la conclusión: una conclusión es cierta si lo son simultáneamente todas sus premisas.

Por su esencia **declarativa**, un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado, sino que es el sistema internamente el que proporciona la secuencia de control.

No existe el concepto de asignación de variables, sino el de **unificación**. No hay un “estado” de las variables que se vaya modificando por sucesivas asignaciones, generalmente asociadas a posiciones de memoria, sino que las variables asumen valores al unificarse o “ligarse” con valores particulares temporalmente y se van sustituyendo durante la ejecución del programa

Un programa lógico contiene una base de conocimiento sobre la que se hacen consultas. La base de conocimiento está formada por hechos, que representan la información del sistema expresada como relaciones entre datos, y por reglas lógicas que permiten deducir consecuencias a partir de combinaciones entre los hechos y, en general, otras reglas. Se construye especificando la información del problema real en una base de conocimiento en un lenguaje formal y el problema se resuelve mediante un mecanismo de inferencia que actúa sobre ella. Así pues, una clave de la programación lógica es poder expresar apropiadamente todos los hechos y reglas necesarios que definen el dominio de un problema.

La programación lógica está basada en la noción de que el programa implementa una relación, en vez de una transformación. Los predicados son relaciones, que al no tener predefinido una “dirección” entre sus componentes, permiten que sus argumentos actúen indistintamente como argumentos de entrada y salida. Esta característica se denomina **inversibilidad**. A su vez, a diferencia de las funciones donde está la restricción del a unicidad de la imagen para un elemento determinado del dominio, una relación permite vincular a cada elemento con muchos otros elementos, permitiendo soluciones alternativas. Dado que las relaciones son más generales que las transformaciones, la programación lógica es potencialmente de más alto nivel que la de otros paradigmas.

Internamente, existe un mecanismo, un “motor”, que actúa como **control de secuencia**. Durante la ejecución de un programa va evaluando y combinando las reglas lógicas de la base de conocimiento para lograr los resultados esperados. La implementación del mecanismo de evaluación puede ser diferente en cada lenguaje del paradigma, pero en todos los casos debe garantizar que se agoten todas las combinaciones lógicas posibles para ofrecer el conjunto completo de respuestas alternativas posibles a cada consulta efectuada. El más difundido se denomina **backtracking**, que utiliza una estrategia de búsqueda primero en profundidad.

La recursividad como estrategia lógica para encontrar soluciones, junto con la utilización de listas para representar conjuntos de valores, son dos características típicas de los programas lógicos.

Los lenguajes del paradigma lógico, en general incluyen herramientas para realizar **soluciones** **polimórficas** y manejar el concepto de **orden superior**, entendido como la capacidad de un lenguaje para manejar su propio código como una estructura de datos más. Son un conjunto de funcionalidades que dotan de una enorme expresividad y potencia a los programas.

# Lógica Proposicional

El paradigma tiene sus fundamentos en las teorías de la lógica proposicional. De ella, se toma un tipo especial de lógica conocido como “lógica de predicados de primer orden”. A diferencia de otros paradigmas que se basan en la idea de

“transformación” de datos, la programación lógica se fundamenta en la noción de “relación”.

## Relaciones

Si se consideran dos conjuntos de valores S y T, R es una relación entre S y T si, para cada x Є S e y Є T, R(x, y) es o verdadera o falsa. Si R(x, y) es verdadera, se dice que R se mantiene entre x e y.

Una diferencia importante entre el concepto de relación y el de transformación que utilizan las funciones, es que una relación no sólo es de uno a uno o de muchos a uno, sino que puede ser también de uno a muchos o de muchos a muchos. El concepto matemático de relación es más amplio que el de función.

## Cláusulas de Horn

Son una forma de lógica de predicados con una sola conclusión en cada cláusula y un conjunto de premisas de cuyo valor de verdad se deduce el valor de verdad de la conclusión: una conclusión es cierta si lo son simultáneamente todas sus premisas. Son el fundamento directo a la definición de reglas de inferencia que utilizan los lenguajes de programación lógica.

Un programa lógico es una secuencia de cláusulas de Horn expresadas mediante la sintaxis de un lenguaje en particular. Ante un determinado requerimiento, cuyo resultado depende de probar la validez de una conclusión, se evalúa el valor de verdad de cada una de las premisas. La forma de escribir las cláusulas de Horn es al contrario de lo habitual: se escribe primero el consecuente y luego el antecedente. El antecedente puede ser una conjunción de condiciones que se denomina secuencia de objetivos, cada uno de los cuales puede ser a su vez un consecuente de una nueva cláusula. Si bien es su fundamento matemático, los lenguajes de programación lógica no pueden explotar toda la potencialidad de la lógica matemática, porque hay formalismos que no son implementables. Se la puede usar limitadamente para especificar problemas que sean computables. Limitarse a cláusulas de Horn asegura la implementabilidad, y le permite a la implementación ser aceptablemente eficiente.

## 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.

# Unificación

Los **átomos** son valores primitivos que no tienen otras propiedades que la habilidad de distinguirse unos de otros. Son usados para representar objetos del mundo real que son primitivos en lo concerniente a la aplicación.

Una variable lógica significa la especificación de un dato indeterminado. Las **variables** tienen un significado distinto al del que tienen en los lenguajes de otros paradigmas donde se las asocia con posiciones de memoria donde se asignan valores. Las variables son incógnitas, indeterminaciones de valores, que al ejecutarse un programa asumen un conjunto posible de valores como resultados alternativos. Al no necesitar declaración de tipos de datos, una variable se puede unificar con cualquier tipo de valores. Una variable se declara implícitamente por su ocurrencia en la cláusula, y su alcance es sólo la cláusula en la cual aparece.

## Variables anónimas

Existen variables sin nombre, que se representan mediante el símbolo de subrayado (\_). Pero aunque todas las variables anónimas se escriben igual, son todas distintas. Es decir, mientras que dos apariciones de la secuencia de

caracteres Hola se refieren a la misma variable, dos apariciones de la secuencia \_ se refieren a variables distintas.

Se la utiliza cuando se quiere hacer referencia a una variable que después no se la vuelve a utilizar en ninguna otra proposición, por lo que significa que no interesa su valor para el propósito de la regla.

## Términos

Los términos son el único elemento de un lenguaje lógico, es decir, los datos son términos, el código son términos, incluso el propio programa es un término. No obstante, es habitual, llamar término solamente a los datos que maneja un programa.

Todos los valores y variables mencionados son términos, pero el concepto de término es más amplio. También existen valores compuestos, que contienen varios átomos a la vez. En general, un término se compone de un identificador seguido de cero a N argumentos entre paréntesis y separados por comas. Los argumentos de un término pueden ser otro término o una variable lógica. La cantidad de argumentos se denomina **aridad**.

## Unificación

La unificación es el mecanismo mediante el cual las variables lógicas toman valor. El valor que puede tomar una variable consiste en cualquier término que representa un dato o conjunto de datos. Se le llama también “**Pattern**

**Matching**” (Encaje de patrones), en analogía a otros paradigmas.

No existe el concepto de asignación a celdas de memoria y la noción de estado de una variable, sujeto a modificaciones reiteradas, sino que las variables son “unificadas” con valores particulares. La unificación no debe confundirse con la asignación de los lenguajes imperativos puesto que representa la igualdad lógica.

Cuando una variable no tiene valor se dice que está libre. Pero una vez que asume un valor, éste ya no cambia y se dice que la variable está ligada, unificada o instanciada. Se dice que dos términos unifican cuando existen valores que hacen posible una ligadura (valor) de las variables tal que ambos términos son idénticos sustituyendo las variables por dichos valores.

La unificación se utiliza constantemente en las consultas de hechos y reglas, para ligar las variables, constantes y términos más complejos que se envían y reciben como argumentos. Dentro de la definición de los predicados, para explicitar una unificación se utiliza la igualdad ( = ) entre valores, variables u otras expresiones.

Para saber si dos términos unifican se pueden aplicar las siguientes normas:

* Una variable siempre unifica con un término, quedando ésta ligada a dicho término.
* Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.
* Para que dos términos unifiquen, deben tener el mismo identificador y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.
* Si dos términos no unifican, ninguna variable queda ligada.

## Predicados

Toda relación que se quiera utilizar en un programa lógico se la representa mediante predicados. Un predicado es un término que tiene un identificador y cada uno de los datos que intervienen constituyen sus argumentos. La cantidad

de argumentos puede de dos, tres o más argumentos. También puede darse el caso de hechos con un sólo átomo, donde la relación es unaria, o incluso, sin argumentos.

Los predicados son los elementos ejecutables en el paradigma. Están determinados en la base de conocimiento mediante hechos y a partir de ellos se efectúan las consultas.

## Hechos

Los hechos son una abstracción de la realidad que el sistema pretende modelar. Es la información del sistema que es considerada correcta y se expresan como relaciones entre datos. Los hechos son cláusulas que forman parte de la base de conocimiento de un programa.

## Consultas

Las consultas son la expresión de las preguntas, requerimientos o pedidos que se hacen sobre la base de conocimiento para encontrar todas las soluciones correctas posibles a un problema.

Una consulta es una llamada concreta a un predicado. Todas las consultas tienen un resultado de éxito o fallo tras su ejecución indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso. Cuando tiene

éxito, las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables libres.

### Consultas de validación

La consulta más sencilla consiste en preguntar acerca de si se cumple cierta relación entre datos. Satisfacer una consulta de este tipo significa validar la veracidad de una relación confrontándola con la información de la base de conocimiento, por lo que hay sólo dos respuestas posibles, de naturaleza booleana.

### Consultas de búsqueda con variables

Una consulta que contiene variables se debe comprender como la búsqueda en la base de conocimiento de los valores que puedan asumir la variable para que la consulta sea cierta, es decir, una consecuencia lógica de la información que se dispone. Ante una consulta variable, se buscarán todos los hechos de la base de conocimiento que permitan unificar los valores con las variables y así satisfacer la consulta. Una consulta con una variable puede tener varias soluciones.

Los predicados no son funciones, por lo que las consultas que se hacen sobre ellos no retornan un valor de respuesta en la expresión misma. Son proposiciones que al evaluarlas permiten obtener un valor de verdad. Cuando se invoca un predicado con todos los argumentos con valores constantes excepto uno, generalmente el último, que es variable, lo que podría interpretarse una función con varios parámetros de “entrada” y uno de “salida”, hay que tener presente, no sólo que la solución no es necesariamente única, sino también que es la variable utilizada como argumento la que asume los valores y no la expresión del predicado como tal.

## Hechos universales

Los hechos, además de contener valores constantes, pueden tener variables para permitir generalizar una relación para todos los elementos, y se denominan hechos universales. En estos casos, aunque la unificación se produce en forma similar a las consultas con variables.

# Reglas de inferencia

## Consultas simultáneas

Una consulta formada por varias proposiciones unidas por comas, que son a su vez otras consultas, representa una conjunción, un “y” lógico de todas las proposiciones, es decir, requiere la verificación simultánea de todas las consultas. Se debe tener en cuenta:

* Las consultas se ejecutan secuencialmente de izquierda a derecha.
* Si una consulta falla, las siguientes consultas ya no se ejecutan y la conjunción, en total, falla.
* Si una consulta tiene éxito, algunas o todas sus variables quedan ligadas, y por tanto, dejan de ser variables libres para el resto de las consultas.
* Si todas las consultas tienen éxito, la conjunción tiene éxito y mantiene las ligaduras de los objetivos que la componen.

## Reglas lógicas

Las reglas son cláusulas que permiten definir nuevas relaciones en función de otras ya existentes. Las reglas se basan en las cláusulas de Horn. La lógica consiste en que cuando todos los antecedentes son ciertos, el consecuente también lo es.

Las soluciones de una consulta sobre una regla se deducen si todos los antecedentes de la regla se verifican con la información existente; para lo cual, estos antecedentes se convierten en nuevas consultas que deben equipararse con

hechos o resolverse por otras reglas. El proceso termina cuando todas las consultas han sido probadas. La solución final viene determinada por los valores unificados de las variables de la meta inicial.

La evaluación así descrita es puramente declarativa. Y asume que si se selecciona una regla: o bien sólo existe esa posibilidad o la regla que se necesita para alcanzar la solución es la que de algún modo ha sido seleccionada. Se alcanza la solución si existe un conjunto apropiado de reglas y sustituciones, tales que aplicando las sustituciones a las reglas se puede deducir la meta desde los hechos conocidos.

## Predicados con varias cláusulas

Un predicado se puede implementar mediante una o más cláusulas, que pueden ser tanto hechos como reglas. Para resolver una consulta, se evalúan secuencialmente todas las cláusulas.

Un predicado con varias cláusulas, representa la existencia de alternativas diversas para llegar a los resultados, para verificar una consulta. En otras palabras, representa el “o” lógico, de manera que una consulta por dicho predicado se verifica si se cumple una cláusula o si se cumple otra, o así sucesivamente, si se cumple alguna otra de las cláusulas definidas.

* En una consulta de validación, ante la primera de ellas que permite satisfacer positivamente la consulta, se interrumpe la búsqueda y se devuelve el valor de verdad. Retornará falso cuando habiendo evaluado todas las cláusulas ninguna permitió satisfacer la consulta.
* En una consulta de búsqueda con variables se devuelve el conjunto total de las soluciones alternativas encontradas por cada una de las cláusulas. Independientemente de que si al evaluar la primera cláusula hubo soluciones o no, se evalúa la segunda cláusula en búsqueda de nuevas soluciones y así sucesivamente hasta agotar todas las posibles alternativas.

# Inversibilidad

La reversibilidad o inversibilidad es la propiedad de los predicados de ser definidos mediante cláusulas que permiten realizar consultar con variables en los diferentes argumentos, de manera de poder obtener un conjunto de resultados con soluciones alternativas. A su vez, en los predicados con varios argumentos, cambiando cuál es el argumento en el que se indica una variable, se hace que un mismo predicado tenga varias funcionalidades lógicas. En otras palabras, inversibilidad es el nombre conceptual que recibe la posibilidad de realizar consultas con variables.

## Indeterminación

El mecanismo de evaluación requiere que las variables se unifiquen con valores para poder retornar los resultados posibles de una consulta. Cuando se realizan consultas variables, hay situaciones donde no es posible establecer el

valor de verdad de una proposición. Esto se produce cuando las variables no se logran unificar satisfactoriamente con algún valor, sin que ello signifique que la respuesta sea negativa, sino que hay una indeterminación acerca de

cuales serían esos valores.

En general, son casos donde hay infinitas soluciones posibles o consultas que, siendo válidas en su formulación en términos del lenguaje, representan consultas sin sentido en la realidad.

La indeterminación de las variables marca un límite a la inversibilidad, ya que no todos los predicados son inversibles. Sus restricciones están dadas por las variables que ante determinadas consultas no se pueden unificar. Un predicado puede ser no inversible por su misma definición o solamente ser inversible para ciertos argumentos, dependiendo de la forma en que se efectúan las consultas.

# Mecanismos de evaluación

## Control de secuencia

Internamente, existe un mecanismo, un “motor”, que actúa como control de secuencia. Durante la ejecución de un programa va evaluando y combinando las reglas lógicas de la base de conocimiento para lograr los resultados esperados. La implementación del mecanismo de evaluación puede ser diferente en cada lenguaje del paradigma, pero en todos los casos debe garantizar que se agoten todas las combinaciones lógicas posibles para ofrecer el conjunto completo de respuestas alternativas posibles a cada consulta efectuada. El más difundido se denomina backtracking, que utiliza una estrategia de búsqueda primero en profundidad.

## Backtracking

El backtracking consiste en una búsqueda en la base de conocimiento que se efectúa de arriba hacia abajo, realizando una búsqueda primero en profundidad. Las reglas son resueltas de izquierda a derecha. Cuando se precisa seleccionar una regla, este algoritmo de control selecciona la primera que encuentra, si conduce a un punto muerto, selecciona la segunda, y así hasta que todas las posibilidades han sido probadas.

El mecanismo de backtracking puede expresarse como “volver atrás y probar de nuevo”. Permite evaluar todas las combinaciones posibles y así encontrar todas las soluciones alternativas a un problema dado.

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento recursivo. Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.

El backtracking le da a la programación lógica un estilo distintivo, y es responsable de gran parte de su potencialidad y expresividad. Por otra parte, para resolver problemas complejos requiere uso de muchos recursos, tanto en tiempo como en espacio de almacenamiento.

Esquemáticamente, el backtracking funciona de la siguiente manera: Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuantas soluciones alternativas puede tener. Cada una de las alternativas se denomina punto de elección. Dichos puntos de elección se anotan internamente y de forma ordenada.

* Se escoge el primer punto de elección y se ejecuta el objetivo eliminando el punto de elección en el proceso.
* Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicándole estas mismas normas.
* Si el objetivo falla, da marcha atrás recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y deshaciendo las ligaduras de sus variables. Es decir, comienza el backtracking.
* Cuando uno de esos objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución continúa de nuevo hacia adelante. El punto de elección se elimina en el proceso.
* El proceso se repite mientras haya objetivos y puntos de elección anotados. De hecho, se puede decir que un programa ha terminado su ejecución cuando no le quedan puntos de elección anotados ni objetivos por ejecutar en la secuencia.
* Además, los puntos de elección se mantienen, aunque al final la conjunción tenga éxito. Esto permite posteriormente conocer todas las soluciones posibles.

## 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**.

## 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).

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.

# 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.

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).

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;