Paradigma Lógico
(Material complementario)

Ing. Lucas Spigariol - 2007

Principales 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.

En otros paradigmas, las salidas son funcionalmente dependientes de las entradas, por lo que el programa puede verse abstractamente como la implementación de una transformación de entradas en salidas. En cambio, 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**.

Existen también otras herramientas más complejas, como las que buscan incrementar la eficiencia o las que abren a la posibilidad de meta programación, que requieren de una cuidadosa utilización ya que se introducen en el interior del sistema mismo y permiten alterar la naturaleza declarativa del paradigma.

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 la 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.

Ejemplo:

? X = 4 .

Unifica directamente asumiendo la variable **X** el valor **4**. Esto provoca la sensación de que se está asignando el valor a la variable al estilo imperativo, pero es una comparación.

? 4 = X.

También unifica directamente asumiendo la variable **X** el valor **4**.

Un par de términos más complejos como los siguientes.

? a(X,3) = a(4,Z).

Unifican dando valores a las variables: X vale 4, Z vale 3.

Por otra parte, no todas las variables están obligadas a quedar ligadas.

Ejemplo:

? h(X) = h(Y).

Unifican aunque las variables **X** e **Y** no quedan ligadas. No obstante, ambas variables permanecen unificadas entre sí. Si posteriormente se liga **X** al valor 3, entonces automáticamente la variable **Y** tomará ese mismo valor. Lo que esta ocurriendo es que, al unificar los términos dados, se impone la restricción de que **X** e **Y** deben tomar el mismo valor aunque en ese preciso instante no se conozca dicho valor.

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.

Ejemplo:

? k(Z, Z) = k(4, H).

Una misma variable puede aparecer varias veces en los términos a unificar. Por el primer argumento, **Z** se liga al valor **4**. Por el segundo argumento, **Z** y **H** unifican, pero como **Z** se liga a un valor, entonces **H** se liga a ese mismo valor, que es **4**.

? k(Z, Z) = k(4, 3) .

No unifican. Una variable no puede ligarse a dos valores distintos.

? a(b(j, K), c(X)) = a(b(W, c(X)), c(W)).

Unifican, para **j = W**, **K = c(X)**, y **X = W**

? k(_,_) = k(3,4).

Las variables anónimas, al ser todas distintas, unifican perfectamente.

Como la igualdad no es una asignación, sino una comparación, no tiene sentido hablar de variables que sean contadores o acumuladores como en otros paradigmas.

Ejemplo:

? X = X + 1.

no

? X = X + K.

no

Representan comparaciones que son falsa por definición ya que no existe valor alguno de **X** que permita que sean verdadera, por lo tanto no se produce la unificación. No tiene sentido hablar del "estado anterior" de **X** y el "nuevo estado" de **X** como si fuera el resultado de una asignación.

Generación

Una estrategia frecuente para implementar predicados, es que en las reglas definidas por varias consultas, **las primeras generen soluciones posibles** y **las últimas filtren las válidas**. En general, sobre las primeras se hacen consultas de búsqueda con variables y en las últimas consultas de validación.

Ejemplo:

```
regla( X ):- generador( X ),  filtro( X ).
```

Ante una consulta con variables, de todas las respuestas posibles de **generador/1** son respuestas válidas de **regla/1** las que se verifiquen mediante **filtro/1**.

Ejemplo:

```
animal( leon ).
animal( elefante ).
animal( perro ).
animal( gato ).
paraLaCasa( helecho ).
paraLaCasa( perro ).
paraLaCasa( parrilla ).
paraLaCasa( gato ).
mascota( P ):- animal( A ), paraLaCasa( A )
```

De todos los animales posibles, que se obtienen con la consulta a **animal/1**, se considera mascota aquellos que son aptos para llevar a la casa, con el predicado **paraLaCasa/1**. Que haya otros elementos que también se puedan llevar a la casa no afecta en la solución.

```
? mascota( X).
X = perro
X = gato.
```

Polimorfismo

El **polimorfismo permite obtener soluciones más genéricas**, que sean válidas para **diferentes tipos de datos contemplando las particularidades** de cada uno de ellos. Se basa en el carácter débilmente tipado del lenguaje, que permite que no haya que predeterminar el tipo de dato de cada argumento de un predicado, variable, functor o cualquier término.

Si bien todos los predicados pueden recibir cualquier tipo de argumento, muchas veces el uso que se hace de ellos en el interior de las cláusulas que lo definen, delimita un rango de tipos de valores que tiene sentido recibir. No sucede un error si se envía un dato de diferente a los esperados, sino que en general directamente el predicado falla por no poder unificar.

Ejemplo:

Un predicado ultimo/2 que relaciona una lista con el último elemento, puede ser definido así:

```
ultimo([ X ], X )
ultimo([_|Xs], X ):- ultimo( Xs, X ).
```

Si se invoca con una lista que contenga cualquier tipo de elementos, el predicado funciona correctamente

? ultimo([juan, pedro], X).

X = pedro

? ultimo([1,2,3,4], X).

X = 4

Pero si se invoca en el primer argumento con otro tipo de dato que no sea una lista

? ultimo(6, X).

no

La consulta falla unifica debido a que el valor **6** no unifica en ninguna de las dos cláusulas del predicado, que por su misma definición, sólo unifican con listas.

En forma similar sucede con los functores. El predicado **sueldo/2** relaciona un functor que representa a una persona que trabaja de jefe, donde su primer componente es el nombre, la segunda es el sueldo básico y la tercera es un adicional, con el sueldo total que le corresponde, como suma de los dos valores.

sueldo(jefe(_, Basico, Adicional), Sueldo) :- Sueldo is Basico + Adicional.

Las invocaciones que respetan el functor del primer argumento, funcionan correctamente:

? sueldo(jefe(laura, 3000, 800), X).
X = 3800

Las invocaciones que en el primer argumento usen valores o expresiones de cualquier otro tipo, fallan siempre.

? sueldo(laura, X).
No

Una forma de aprovechar al máximo la propiedad polimórfica de que un predicado pueda recibir cualquier tipo de dato y tratarlos indistintamente, cuando luego en las cláusulas se utilizan expresiones que limitan los tipos de datos que dan respuestas positivas, es **agregar más cláusulas** para contemplar los **otros posibles tipos de datos**, y tratarlos de la forma adecuada.

Ejemplo:

Continuando con el ejemplo anterior, se pueden ampliar la base de conocimiento y agregar nuevas cláusulas del predicado sueldo, para que calcule el sueldo para el personal de una empresa representados con diferentes functores, cada uno a su manera.

basico(categoria1, 1000).

basico(categoria2, 1500).

valorHora(15).

Son nuevos hechos

sueldo(jefe(_, Basico, Adicional), Sueldo) :- Sueldo is Basico + Adicional.

Se mantiene igual al ejemplo anterior.

sueldo(empleado(_, Categoria), Sueldo) :- basico(Categoria, Sueldo).

Esta nueva regla indica que los empleados de la empresa cobran un sueldo correspondiente a su categoría. Para ello también se agregan hechos del predicado **basico/2**.

sueldo(contratado(_, Cant), Sueldo) :- valorHora(Valor), Sueldo is Valor * Cant.

En esta otra, se indica que el personal contratado cobra un sueldo según la cantidad de horas que trabajó cada uno y el valor de la hora que está registrado con el predicado valorHora.

Pueden hacerse nuevas consultas:

? sueldo(empleado(juan, categoria1), X).

X = 1000

? sueldo(empleado(raul, categoria2), X).

X = 1500

? sueldo(contratado(esteban, 20)], X).

X = 300

Completando el ejemplo, se incorpora un hecho con el predicado **personal/1** con una lista donde están todos los empleados, jefes y contratados de la empresa.

personal([empleado(juan, categoria1), empleado(ana, categoria1), empleado(raul, categoria2), jefe(jorge, 2000, 500), jefe(laura, 3000, 800), contratado(esteban, 20)]).

También se agrega un nuevo predicado **nombre/2**, con varias cláusulas en las que se relaciona al functor que representa a cada trabajador con su nombre.

nombre(jefe(Nombre, _, _), Nombre).

nombre(empleado(Nombre, _), Nombre).

nombre(contratado(Nombre, _), Nombre).

A partir de lo anterior, se define un predicado **cuantoGana/2**, que relaciona el nombre de una persona que trabaja en la empresa con su correspondiente sueldo.

```
cuantoGana( Nombre, Sueldo ):-
        personal(Lista),
        member(P, Lista),
        nombre(P, Nombre),
        sueldo(P, Sueldo).
```

Se pueden realizar diferentes consultas, siendo la más amplia:

? cuantoGana(Nombre, Sueldo).

Nombre = juan
Sueldo = 1000 ;

Nombre = ana
Sueldo = 1000 ;

Nombre = raul
Sueldo = 1500 ;

Nombre = jorge
Sueldo = 2500 ;

Nombre = laura
Sueldo = 3800 ;

Nombre = esteban
Sueldo = 300

El polimorfismo permite poder construir **predicados genéricos** que contemplen un abanico de casos posibles que se definan y a la vez permitir que ante modificaciones futuras o nuevos problemas a resolver, el impacto del cambio sea mínimo, de manera que muchos predicados puedan quedar expresados de la misma manera y seguir funcionando correctamente.

Ejemplo:

El predicado **cuantoGana/2** es lo suficientemente polimórfico como para soportar cualquier tipo de trabajador con que se lo invoque y seguirá siendo válido, sin modificaciones, para cualquier otro tipo de dato como primer argumento, siempre y cuando los predicados **nombre/2** y **sueldo/2** lo contemplen adecuadamente.

Se agrega otro tipo de trabajador, con un cálculo de sueldo en particular, que para el ejemplo se resuelve con otro predicado auxiliar **calculoCualquiera/4** a partir de los datos del nuevo functor.

nombre(unTrabajador(Nombre, _, _, _), Nombre).

```
sueldo( unTrabajador( _, X, Y, Z ), Sueldo) :- calculoCualquiera( X, Y, Z, Sueldo ).
```

Se pueden implementar soluciones polimorfitas con **cualquier tipo de dato**. Para los que tienen una nomenclatura específica, como los functores o las listas, es más simple y se expresa directamente en la forma de los argumentos para que el mecanismo de unificación ("pattern matching") lo resuelva. Para los otros, se pueden incorporar en cada regla, consultas de **comprobación de tipos de datos**.

Predicados de orden superior

El concepto de **orden superior** es la capacidad de un lenguaje para manejar su propio código como una estructura de datos más.

Las capacidades de orden superior del lenguaje, son un conjunto de funcionalidades generalmente desconocidas en lenguajes de diferentes paradigmas que dotan de una **enorme expresividad y potencia a los programas**. También son una forma de implementar **polimorfismo** en el paradigma lógico.

En particular, un predicado de orden superior es aquél que tiene la capacidad de **recibir un predicado como argumento y poder utilizarlo**. Se escribe un predicado y se lo pasa como argumento a otro y esta último lo ejecuta sin necesidad de saber exactamente qué está evaluando.

Existen algunos **predicados predefinidos** de orden superior que resultan muy útiles y son ampliamente utilizados y también es posible **definir predicados propios** de orden superior.

En cualquier predicado de orden superior, en el argumento que se recibe un predicado, puede enviarse una consulta simple o una expresión que incluye a varias consultas entre paréntesis, ya sea unidas por comas (,) o puntos y comas (;) para representar consultas simultáneas o alternativas respectivamente.

Negación

El predicado de orden superior más simple, y a la vez muy utilizado, es el **not/1**, que representa una negación.

Si **P** es una proposición, entonces **not(P)** es una proposición que niega el valor de verdad asumido para **P**. Así, la negación de algo falso se toma por verdadera.

Otra característica es que las variables del predicado que se niega deben estar ligadas. Cuando se evalúa una consulta donde sus variables no han sido instanciadas, los valores que busca son los que hacen cierta la expresión y luego al ser negados dan un valor falso. No hay forma que se busquen los valores que dan resultado falso para que al negarlos se obtenga un valor verdadero. Sólo si no hay ninguna solución para la consulta variable que es argumento del **not/1** y por lo tanto falla, la consulta con la negación dará verdadero, pero sin ligar a la variable libre con un valor.

Listas con respuestas alternativas

Hay un conjunto de predicados cuya finalidad es almacenar en una lista todas las soluciones de un predicado dado, entendiendo como tales, las ligaduras que se producen en una o varias variables libres que se indican explícitamente. Algunos de ellos son los siguientes:

- **findall/3**

Genera una lista con todas las soluciones del predicado dado según el orden en que se van sucediendo.

Tiene tres argumentos: una primera expresión, la consulta que se quiere evaluar y la lista con un elemento para cada solución alternativa a la consulta.

El primer argumento es una variable o un término que contiene la o las variables libres que interesan de la solución. Es posible que no interesen todas, por lo que se indican sólo las variables necesarias, que han de ser un subconjunto de las que aparecen en el segundo argumento.

El segundo argumento es el predicado del cual se quiere obtener soluciones, para ello, debe contener una o más variables sin ligar. Por otra parte, dicha consulta debe tener un número finito de soluciones, si no, se entraría en un bucle infinito al ejecutar alguno de los predicados.

El tercer argumento es la lista con las soluciones. Cada elemento de la lista va a tener la forma de la expresión del primer argumento. Si no hay soluciones, se obtiene la lista vacía. La lista de soluciones no se ordena y puede contener elementos repetidos. Solamente se obtiene una solución.

Ejemplo:

Ante una invocación de la forma

? findall(T, C, L)

L es la lista de todas las instancias del término **T** tales que la consulta **C** se cumple. Tras la consulta, las variables de **T** y **C** permanecen sin unificar.

Usos correctos de **findall/3**:

findall(X, predicado(X), Lista).

findall(X, predicado(X, Y), Lista).

findall(elemento(X, Y), predicado(X, Y), Lista).

Estas son algunas formas de uso incorrecto del predicado, en las que hay variables del primer argumento que no aparecen en el segundo.

findall(X, predicado(Y), Lista).

findall(X, predicado(Y, Z), Lista).

findall(elemento(X, Y), predicado(Y, Z), Lista).

Ejemplo:

Dada la siguiente base de conocimiento, donde se indica cada materia que cursa cada alumno:

cursa(juan, paradigmas).

cursa(jose, paradigmas).

cursa(juan, operativos).

cursa (ana, sintaxis).

Mediante la siguiente consulta:

? findall(Y, cursa(X, Y), Lista).

Lista = [paradigmas, paradigmas, operativos, sintaxis]

Se obtiene una lista con las soluciones del segundo argumento de **cursa/2**. Las soluciones del primer argumento, con la variable **X**, se ignoran. En la lista aparecen todas las soluciones, incluidas las repetidas. En este caso, son todas las materias que alguien cursa.

? findall(par(X, Y), cursa(X, Y), Lista).

Lista = [par(juan, paradigmas), par(jose, paradigmas), par(juan, operativos), par(ana, sintaxis)]

Se obtiene una lista con las soluciones de los dos argumentos de **cursa/2**. Se las ubica en la lista con la estructura del functor indicado como primer argumento (**par/2**). Son todos los pares de alumno y materia que cursa que hay en la base de conocimiento.

Ejemplo:

Un predicado común de manejo de listas como la concatenación se puede resolver con **findall/3**.

```
concatenacion( XS, YS, ZS ):-
        findall( X, ( member( X, XS ); member( X, YS) ), ZS ).
```

El argumento que se recibe como un predicado está formado por dos consultas, unidas lógicamente por una relación de "o" (;). Se puede leer: la concatenación de dos listas es la lista formada por todos los elementos que cumplen la condición de ser miembros de la primera lista o ser miembros de la segunda.

```
intersección( XS, YS, ZS ):-
        findall(X, ( member( X, XS ), member(X,YS) ), ZS ).
```

Un predicado **intersección/3** relaciona dos listas con una tercera que contiene los elementos comunes a ambas, se expresa de manera muy parecida a la anterior, pero con el segundo argumento con una consulta doble unida por una relación de "y" (,), validando que los elementos sean miembros de las dos listas simultáneamente.

- **bagof/3**

Similar a **findall/3**, ya que genera una lista con todas las soluciones del predicado dado. La diferencia es que tiene una solución por cada variable libre que no haya sido indicada como parte de la solución y además falla cuando no hay soluciones.

Ejemplo:

Para la misma base de conocimiento del ejemplo anterior.

```
?- bagof( Y, cursa (X, Y), Lista).

X = juan
Lista = [paradigmas, operativos] ;

X = jose
Lista = [paradigmas] ;

X = ana
Lista = [sintaxis] ;
```

Para cada solución de las variables libre, en este caso **X**, la lista contiene todas las soluciones de las otras variables, en este caso **Y**. O sea, la lista de materias que cursa cada alumno.

```
?- bagof( X, cursa (X, Y), Lista).

Y = paradigmas ,
Lista = [juan, jose] ;

Y = operativos ,
Lista = [juan] ;

Y = sintaxis ,
Lista = [ana]
```

Para cada solución de las variables libre, en este caso **Y**, la lista contiene todas las soluciones de las otras variables, en este caso **X**. O sea, la lista de alumnos que cursa cada materia.

- **setof/3**

Similar a **bagof/3**. La diferencia es que no incluye las soluciones duplicadas que pudieran existir y que ordena la lista del tercer argumento.

Ejemplo:

Como una alternativa para uno de los predicados auxiliares de los ejemplos anteriores, el siguiente predicado relaciona dos listas con una tercera que es la unión sin repetidos de todos los elementos de ambas listas.

```
unionSinRepetidos( XS, YS, ZS ):- setof(X, (member(X,XS); (member(X,YS)), ZS).
```

Predicados predefinidos

Existe una cantidad enorme de otros predicados de orden superior que realizan las más diversas tareas. A continuación se enumeran sólo algunos.

- **forall/2**

Para todas las soluciones del predicado del primer argumento, evalúa el predicado del segundo argumento. Si alguno de los predicados de los argumentos falla, falla el predicado principal, sin que haya forma de indicar cuál de los predicados falló.

Ejemplo:

Para definir un predicado que indique si un conjunto, representado por una lista, está incluido en otro, sin importar el orden de los elementos, se podría desarrollar una formulación recursiva. Otra forma es invocando al presente predicado.

```
incluido(L1,L2):- forall( member( X, L1 ), member( X, L2 ) ).
```

Se afirma que una lista **L1** está incluida en otra lista **L2**, si para todo elemento **X** que es miembro de la lista **L1** se verifica que también es miembro de la lista **L2**.

- **maplist/3**

Permite aplicar un predicado de "mapeo" entre dos listas de elementos, por la cual cada elemento de una lista se relaciona mediante el mapeo con el elemento de la misma posición de la otra lista.

Ejemplo:

El predicado **capital/2** relaciona cada país con su capital.

```
capital( argentina, buenosaires).
capital( peru, lima).
capital( urugual, montevideo).
```

Para obtener una lista con las capitales, a partir de una lista de países, se puede hacer:

```
? maplist( capital, [ argentina, peru ], Lista).
Lista = [ buenosaires, lima ].
```

Si se agregan a la base de conocimiento nuevos hechos:

```
capital( bolivia, lapaz ).
capital( bolivia, sucre ).
? maplist( capital, [ argentina, peru, bolivia ], Lista).
Lista = [ buenosaires, lima, lapaz ] ;
Lista = [ buenosaires, lima, sucre ]
```

Se obtienen dos respuestas alternativas, dado que el predicado que se evalúa también tiene dos soluciones para uno de los valores. Si hubiera más alternativas para cada consulta individual, se devolverían todas las posibles listas con las combinaciones de valores.

Llamadas de orden superior

Para desarrollar predicados de orden superior es necesario poder evaluar los predicados que se reciben como argumentos. Para ello, y en general, para convertir cualquier término o expresión en una consulta, existe el predicado **call**, con diferentes aridades, de acuerdo a la aridad del predicado que se quiere evaluar.

El primer argumento es el nombre del predicado y los siguientes son los argumentos con los que se desea evaluar al predicado.

Ejemplo:

Un predicado de orden superior llamado **selecciona/3**, que permite obtener una sublista con los elementos de una lista que cumplen cierta condición, recibiendo como primer argumento dicha condición que es un predicado de un solo argumento.

```
selecciona(_, [ ], [ ] ).
selecciona(P, [X|Xs], [X|Ys] ) :-
                    call( P, X ),
                    selecciona( P, Xs, Ys ).
selecciona(P, [X|Xs], Ys) :-
                    not( call( P, X ) ),
                    selecciona( P, Xs, Ys ).
```

La variable **P** se unifica con el predicado que se envíe como argumento. Al evaluar **call**, se envía como primer argumento el predicado **P** y como segundo una variable para que se evalúe como el único argumento de **P**. Si se verifica, el elemento **X** es seleccionado y forma parte de la lista del tercer argumento, y si no, se lo excluye.

Se esperan predicados **P** con un solo argumento, ya que **call** está siendo invocado con dos argumentos: **P** y el argumento de **P**.

Una forma de uso del predicado **selecciona/3** puede ser:

```
varon( juan ).
varon( pedro ).
? selecciona( varon, [olga, juan, estela, pedro], L).
L = [juan, pedro]
```

Ejemplo:

Con los mismos conceptos se puede construir un predicado **mapea/3** que tenga la misma funcionalidad que el predicado predefinido **maplist/3** mencionado anteriormente.

```
mapea( _, [ ], [ ] ).
mapea( P, [X|Xs], [Y|Ys] ) :-
                    call( P, X, Y ),
                    mapea( P, Xs, Ys ).
```

La variable **P** se unifica con el predicado que se envíe como argumento. Como se esperan predicados **P** con dos argumentos, que puedan relacionar los elementos de una lista con los de la otra, se invoca a **call** con tres argumentos: **P** y sus dos argumentos.