PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL-3: El lenguaje PROLOG, aspectos avanzados

3. Predicados de orden superior

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

MOTIVACIÓN Y DEFINICIÓN

- Un predicado de orden superior es un predicado tal que al menos uno de sus argumentos, en lugar de ser un término, es un predicado.
- La idea procede de la *Lógica Matemática* y fue introducida en programación por los *lenguajes funcionales*, extendiéndose su uso posteriormente a otros paradigmas.
- Objetivo: producción de código no redundante y reutilizable, por medio de “plantillas” genéricas.
- Beneficios: se consigue un código más conciso, fiable y fácil de mantener.

Ejemplo

```
% todos_nat(+L)
% todos los elementos de L
% son números naturales

todos_nat([]).

todos_nat([C|R]) :-
    natural(C),
    todos_nat(R).

natural(C) :-
    integer(C), C >= 0.
```

```
% sin_variables(+L)
% L no contiene ninguna
% variable.

sin_variables([]).

sin_variables([C|R]) :-
    nonvar(C),
    sin_variables(R).
```

¡Las dos implementaciones anteriores son prácticamente iguales!

Ambas comprueban si todos los elementos de L cumplen algo

- No tiene sentido repetir *n* veces un código que solo se diferencia en qué deben cumplir los elementos de la lista.
- Conviene diseñar un predicado "genérico" capaz de comprobar si todos los elementos de una lista cumplen un cierto objetivo `Obj`:

```
% map(+Obj, +L)
% cierto si todos los elementos de la lista L
% cumplen el objetivo Obj

map(Obj, []).
map(Obj, [C|R]) :-
  <ejecución del objetivo Obj sobre el valor C>,
  map(Obj, R).
```

y definir los anteriores (y otros similares) por medio de él:

```
todos_nat(L) :- map(natural, L).
sin_variables(L) :- map(nonvar, L).
```

- PROLOG hace un uso extenso de predicados de orden superior:
  - El predicado de negación `\+`
  - Los predicados de recolección de soluciones `bagof`, `setof` y `findall`
  - El predicado `map/2` anterior, denominado `maplist/2` en SWI-Prolog (así como otros predicados predefinidos similares, a menudo conocidos como *predicados de orden superior* ***clásicos***, que se verán más adelante)

  son todos ellos predicados de orden superior.

- PROLOG facilita además herramientas (*predicados de orden superior* ***básicos***) que permiten a los programadores diseñar fácilmente sus propios predicados de orden superior.

PREDICADOS DE ORDEN SUPERIOR BÁSICOS

Familia de predicados `call/N`, con `N = 1,2,3,...`

- Los valores de `N` admitidos dependen del intérprete.
- Para `N=1`: `call(+Obj)` intenta ejecutar `Obj`, produciendo un error en caso de que no sea un objetivo ejecutable.
- Para `N>1`: `call(+Obj, +Extra1, +Extra2, ...)` añade los valores `Extra1, Extra2, ...`, en el orden dado, detrás de los argumentos propios de `Obj`, si este los tuviera, e intenta ejecutar el objetivo resultante, produciendo un error en caso de que no sea ejecutable.

Ejemplo (Usos de `call/N`)

Suponga disponible el predicado `gusta/2`:

```
?- call(gusta(X, prolog)). % N=1, Obj=gusta(X,prolog)
?- call(gusta(X), prolog). % N=2, Obj=gusta(X)
?- call(gusta, X, prolog). % N=3, Obj=gusta

% las tres consultas anteriores son equivalentes,
% todas ellas ejecutan "gusta(X, prolog)".

?- call(gusta, X).
% intenta ejecutar "gusta(X)"
ERROR: procedure `gusta(A)' does not exist

?- call(gusta(X), prolog, scala).
% intenta ejecutar "gusta(X,prolog,scala)"
ERROR: procedure `gusta(A,B,C)' does not exist
```

Utilidad de los predicados `call/N`

Los predicados `call/N` se usan fundamentalmente tomando `N>1` para construir y resolver objetivos en tiempo de ejecución y así poder:

- Implementar predicados de orden superior clásicos como los de *aplicación, filtrado y reducción* procedentes de la programación funcional y discutidos a continuación.
- Implementar nuevos predicados de orden superior cuando se detecte la necesidad de utilizar códigos muy similares entre sí que solo se diferencian en algunos detalles parametrizables. Este uso se ilustra al final de esta presentación con varios ejemplos.

Observación

La combinación "`Exe =.. [Obj,C], call(Exe)`" usada anteriormente por motivos técnicos (y que puede aparecer por ello en algunos ejercicios de examen) es equivalente a `call(Obj, C)`.

PREDICADOS DE ORDEN SUPERIOR CLÁSICOS SOBRE LISTAS

Operaciones de aplicación (familia map/N)

- Para `N=2` se obtiene el predicado discutido previamente, `map(+Obj,?L)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre **todos** los elementos de la lista `L`.
- Para `N=3` se obtiene el predicado `map(+Obj,?L1,?L2)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre **todas las parejas** de elementos de las listas `L1` y `L2` situados en la misma posición (los dos primeros, los dos segundos, etc).
- Ambos están disponibles en SWI-Prolog bajo el nombre de `maplist` (`maplist/2` y `maplist/3`), a pesar de lo cual es interesante estudiar su implementación (ver a continuación).

Implementación de map/2 (maplist/2 en SWI-Prolog)

```
% map(+Obj, ?L)
% cierto si Obj se puede aplicar con éxito sobre
% TODOS los elementos de la lista L

map(_, []).

map(Obj, [C|R]) :-
        call(Obj, C),  % uso de call/2
        map(Obj, R).
```

El parámetro `Obj` anterior puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadido, *en último lugar*, un argumento más (el elemento `C` que se va tomando de la lista `L`).

Ejemplo (Usos de map/2)

```
?- map(integer, [1,-4]).
true   % tras ejecutar integer(1) e integer(-4)

?- map(var, [1,X]).
false  % tras ejecutar var(1), que falla

?- map(>, [1,0]).   % ejecuta >(1): error
ERROR: procedure ` >(A)' does not exist

?- map(>(2), [1,0]).
true   % tras ejecutar >(2,1) y >(2,0)

?- map(>(2), [1,3]).
false. % tras ejecutar >(2,1) y >(2,3), que falla

?- map(>(2), [a,2]). % tras ejecutar >(2,a), error
ERROR: Arithmetic: `a/0' is not a function
```

Observación (Árboles de Resolución con `call`)

Los Árboles de Resolución en los que aparece el predicado `call/N` se construyen igual que los de los predicados convencionales, con la única diferencia de que, como se ha explicado antes, `call(+Obj, +Extra1, +Extra2, ...)` simplemente añade los valores `Extra1, Extra2, ...`, en el orden dado, detrás de los argumentos propios del objetivo `Obj`, si este los tuviera, y ejecuta el objetivo resultante (produciendo un error en caso de que no sea ejecutable).

El ejemplo a continuación incluye un Árbol de Resolución de este estilo, el correspondiente a la consulta `?- map(>(2), [1,3]).`

Ejemplo (Árbol de Resolución con `call`)

$$?- \underline{map(>(2),[1,3])}.$$

$$\Big| \{Obj/>(2), C/1, R/[3]\}$$

$$?- \underline{call(>(2),1)}, map(>(2),[3]).$$

$$\Big| \{\}$$

$$?- \underline{>(2,1)}, map(>(2),[3]).$$

$$\Big| \{\}$$

$$?- \underline{map(>(2),[3])}.$$

$$\Big| \{Obj4/>(2), C4/3, R4/[]\}$$

$$?- \underline{call(>(2),3)}, map(>(2),[\,]).$$

$$\Big| \{\}$$

$$?- \underline{>(2,3)}, map(>(2),[\,]).$$

$$\boxed{\text{fallo}}$$

Implementación de map/3 (maplist/3 en SWI-Prolog)

```
% map(+Obj, ?L1, ?L2)
% cierto si Obj se puede aplicar con éxito sobre
% TODAS las parejas de elementos de las listas
% L1 y L2 situados en la misma posición.

map(_,[], []).

map(Obj, [C1|R1], [C2|R2]) :-
        call(Obj, C1, C2),    % uso de call/3
        map(Obj, R1, R2).
```

El parámetro `Obj` anterior puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadidos, *en último lugar*, dos argumentos más (los elementos `C1` y `C2` que se van tomando de las listas `L1` y `L2`, respectivamente).

Ejemplo (Usos de map/3, 1/4)

Suponga disponibles los siguientes predicados:

```
% cuad(+X, ?Y)
% cierto si Y es el cuadrado de X

    cuad(X,Y) :-
        Y is X*X.
    % X debe ser de entrada debido al "is"

% trad(?X, ?Y)
% cierto si Y es el nombre del dígito X

    trad(0, cero).
    trad(1, uno).
        ...
    trad(9, nueve).
```

Ejemplo (Usos de map/3, 2/4)

```
?- map(cuad, [2,3], [4,9]).
true  % ejecuta cuad(2,4) y cuad(3,9)

?- map(cuad, [2,3], [4,8]).
false % ejecuta cuad(2,4) y cuad(3,8), que falla

?- map(cuad, [2,3], [4,9,16]).
false % map falla con listas de tamaño distinto

?- map(cuad, [1,2], L).
L = [1,4]
% ejecuta cuad(1,E1), cuad(2,E2) siendo L=[E1,E2]
```

Ejemplo (Usos de map/3, 3/4)

```
?- map(cuad, L, [4,9]).
% ejecuta cuad(E1,4) siendo L=[E1,E2] pero da error:
% el primer argumento de cuad debe ser de entrada
ERROR: Arguments are not sufficiently instantiated

? map(integer, [1,2], L).
% intenta ejecutar integer(1,E1), siendo L=[E1,E2].
ERROR: procedure `integer(A,B)' does not exist

?- map(trad, [1,3], L).
% ejecuta trad(1,E1), trad(3,E2), siendo L=[E1,E2].
L = [uno,tres]

?- map(trad, L, [1,3]).
 % ejecuta trad(E1,1), que falla, siendo L=[E1,E2]
false.
```

Ejemplo (Usos de map/3, 4/4)

```
?- map(cuad, [1,2,3], L), map(trad, L, NL).
L = [1, 4, 9], NL = [uno, cuatro, nueve]

?- map(>, [3,2], [0,1]).
% ejecuta >(3,0) y >(2,1), ambos ciertos
true
```

Siendo `plus(?I1,?I1,?I3)` el predicado predefinido cierto si `I3=I2+I1` y al menos dos de sus argumentos tienen valor:

```
?- map(plus(1), [1,2], L).
% ejecuta plus(1,1,E1) y plus(1,2,E2), con L=[E1,E2]
L = [2, 3]  % suma 1 a todos los elementos de [1,2]

?- map(plus(1), L, [1,2]).
% ejecuta plus(1,E1,1) y plus(1,E2,2), con L=[E1,E2]
L = [0, 1] % resta 1 a todos los elementos de [1,2]
```

Observación (1 de 2)

Conviene destacar que a diferencia de lo que ocurre con los predicados de recolección, los predicados `map/maplist` (o cualquier otro predicado de orden superior, como los discutidos más adelante, involucrando el uso de `call/N` con `N>1`), NO aceptan objetivos negados ni compuestos, por lo que estos deben implementarse aparte. Por ejemplo, suponga que necesita averiguar si una lista contiene exclusivamente números enteros y constantes:

```
?- map((integer;atom), [1,2,a]).  % o maplist
ERROR procedure `;(A,B,C)' does not exist
```

La solución pasa por implementar aparte el objetivo adecuado:

```
intORatom(X) :- integer(X) ; atom(X).

?- map(intORatom, [1,2,a]).
true
```

Observación (2 de 2)

Suponga ahora que necesita averiguar si dos listas dadas tienen mismo tamaño, ambas contienen solo números naturales positivos y cada elemento de la primera es divisible por el correspondiente elemento de la segunda. Una posible solución:

```
positivo(X) :-                  |    divi(X,Y) :-
    integer(X), X > 0.          |      positivo(X),
                                |      positivo(Y),
                                |      X mod Y =:= 0.
```

```
?- map(divi, [4,2,8], [2,1,4]).
% ejecuta divi(4,2), divi(2,1) y divi(8,4)
true

?- map(divi, [4,4], [3,2]).
% ejecuta divi(4,3), que falla.
false
```

Operaciones de filtrado

- `filter(+Obj,+L,?NL)` (`include` en SWI-Prolog)

  cierto si `NL` es la lista conteniendo aquellos elementos de la lista de entrada `L` sobre los que se puede aplicar con éxito el objetivo `Obj` (`NL=[]` si no hay ninguno).

- `excluye(+Obj,+L,?NL)` (`exclude` en SWI-Prolog)

  cierto si `NL` es la lista resultante tras excluir de la lista de entrada `L` aquellos elementos sobre los que se puede aplicar con éxito el objetivo `Obj` (`NL=[]` si se excluyen todos).

En ambos casos el parámetro `Obj` puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadido, *en último lugar*, un argumento más (el elemento `C` que se va tomando de la lista de entrada `L`).

Implementación de filter (include en SWI-Prolog)

```
% implementación recursiva
    filter(_, [], []).

    filter(Obj, [C|R], NL) :-
          call(Obj, C),              % uso de call/2
          !,
          NL = [C|NR],
          filter(Obj, R, NR).

    filter(Obj, [_|R], NL) :-
          filter(Obj, R, NL).

% implementación basada en "findall"
  filter_v2(Obj, L, NL) :-
     findall(X, (member(X,L), call(Obj,X)), NL).
```

Ejemplo (Usos de filter)

```
?- filter(var, [1, X, Hola, 2, adios], [X, Hola]).
% ejecuta var(1), ..., var(adios)
true % último arg. de entrada, para comprobación

?- filter(positivo, [1, X, Hola, -2, adios], NL).
% ejecuta positivo(1), ..., positivo(adios)
NL = [1]  % positivo: ver última observación

?- filter(>(3), [1,2,3], L).
% ejecuta >(3,1), ..., >(3,3)
L = [1, 2]

?- filter(>(3), [1,b,3], L).
% ejecuta >(3,1), >(3,b): error
ERROR: Arithmetic: `b/0' is not a function
```

Implementación de excluye (exclude en SWI-Prolog)

```
excluye(_, [], []).     % implementación recursiva
excluye(Obj, [C|R], NR) :-
   call(Obj, C),        % uso de call/2
   !,
   excluye(Obj, R, NR).
excluye(Obj, [C|R], [C|NR]) :-
    excluye(Obj, R, NR).

% implementación basada en "filter" y "findall"
excluye_v2(Obj, L, NL) :-
  filter(Obj, L, LF),  % include en SWI-Prolog
  findall(X, (member(X,L), \+ member(X,LF)), NL).

% implementación basada en "findall" y "\+ call"
 excluye_v3(Obj, L, NL) :-
  findall(X, (member(X,L), \+ call(Obj,X)), NL).
```

Ejemplo (Usos de excluye)

```
?- excluye(var, [1, X, Hola, 2, adios], L).
% ejecuta var(1), ..., var(adios)
L = [1, 2, adios]

?- excluye(>(3), [1,2,3], L).
% ejecuta >(3,1), ..., >(3,3)
L = [3]

?- excluye(>(3), [1,b,3], L).
% ejecuta >(3,1), >(3,b): error
ERROR: Arithmetic: `b/0' is not a function

?- excluye(par, [1,b,2], L).
% siendo par(X) :- integer(X), X>=0, X mod 2 =:= 0.
L = [1,b]
```

Observación

Note que las implementaciones recursivas discutidas más arriba para las operaciones de aplicación y filtrado presentan recursión de cola módulo cons, recursión en la cual lo único que queda por hacer después de la llamada recursiva es añadir un elemento en cabeza. Recuerde (ver tema de listas) que PROLOG es capaz de optimizar este tipo de recursión, por lo que se trata de implementaciones eficientes.

Ejercicios (Uso de predicados de orden superior clásicos)

- *Ejercicio nº 1, apartado 1.2, de la Práctica de* PROLOG *nº 4.*
- *Ejercicio nº 2, apartado 2.4, de la Práctica de* PROLOG *nº 4.*
- *Dibuje el Árbol de Resolución correspondiente a la consulta* `?- map(cuad, [2,3], L)` *con las implementaciones de* `map/3` *y* `cuad/2` *mencionadas* ▸ *previamente*.

Soluciones propuestas:

Las soluciones propuestas, comentadas, a los ejercicios de esta práctica están disponibles en Práctica de PROLOG nº 4 con soluciones.

$$? - \underline{map(cuadrado, [2,3], L)}.$$

$$\Big| \quad \{Obj/cuadrado, C1/2, R1/[3], L/[C2|R2]\}$$

$$? - \underline{call(cuadrado, 2, C2)}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{\}$$

$$? - \underline{cuadrado(2, C2)}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{X/2, Y/C2\}$$

$$? - \underline{C2 \text{ is } 2 * 2}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{C2/4\}$$

$$? - \underline{map(cuadrado, [3], R2)}.$$

$$\Big| \; \{Obj_5/cuadrado, C1_5/3, R1_5/[], R2/[C2_5|R2_5]\}$$

$$? - \underline{call(cuadrado, 3, C2_5)}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{\}$$

$$? - \underline{cuadrado(3, C2_5)}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{X_7/3, Y_7/C2_5\}$$

$$? - \underline{C2_5 \text{ is } 3 * 3}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{C2_5/9\}$$

$$? - \underline{map(cuadrado, [\,], R2_5)}.$$

$$\Big| \; \{R2_5/[\,]\}$$

$$?-$$

$$\boxed{\text{éxito}}$$

$$L = [C2|R2] = [4|R2] = [4|[C2_5|R2_5]] = [4|[9|R2_5]] = [4|[9|[\,]]] = [4, 9]$$

Operaciones de reducción o plegado

- Permiten obtener *un único valor* a partir de una o varias listas, combinando sucesivamente los elementos de la(s) lista(s) mediante una cierta operación, en una cierta dirección, y partiendo, opcionalmente, de un valor inicial dado.

- Existen distintas operaciones de este tipo, que se diferencian básicamente en cuántas listas combinan, si usan o no valor inicial y en cómo y en qué orden operan sus elementos para obtener el valor de salida.

- Existen también distintas terminologías para referirse a estas operaciones, no siempre consistentes entre sí.

Operaciones de reducción o plegado

`pliegai(+Obj, +L, +VIni, -VFin)`
`(plegado por la izquierda)`

Obtiene un valor final de salida, `VFin`, combinando de izquierda a derecha los elementos de la lista `L` mediante `Obj` y partiendo del valor inicial `VIni`:

- Si `L=[]`, `VFin = VIni`.
- Si `L=[C|R]`:
  1. Combina `VIni` con `C` mediante `Obj`, dando lugar a `V1`, es decir, ejectuta `Obj(VIni, C, V1)`.
  2. Pliega por la izquierda (recursivamente) el resto de la lista, `R`, con valor inicial el `V1` obtenido en el paso anterior.

Observación

- Desde un punto de vista iterativo, siendo `L=[x1,...,xn]`, el predicado `pliegai(+Obj, +L, +VIni, -VFin)` hace lo siguiente:

```
Obj(VIni,x1,V1)
Obj(V1,x2,V2)
...
Obj(V(n-1),xn,VFin).
```

- El parámetro `Obj` puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadidos, *en último lugar y en el orden dado a continuación*, tres argumentos más: el valor calculado en el paso anterior (`VIni` la primera vez), el elemento de la lista que se combina con él y el valor de salida.

Implementaciones de pliegai y de foldl (SWI-Prolog)

```
% pliegai(+Obj, +L, +VIni, -VFin)

  pliegai(_, [], VIni, VIni).

  pliegai(Obj, [C|R], VIni, VFin) :-
        call(Obj, VIni, C, V1),     %% Aquí difieren
        pliegai(Obj, R, V1, VFin).

% foldl(+Obj, +L, +VIni, -VFin) SWI-Prolog

  foldl(_, [], VIni, VIni).

  foldl(Obj, [C|R], VIni, VFin) :-
        call(Obj, C, VIni, V1),     %% Aquí difieren
        foldl(Obj, R, V1, VFin).
```

Ejemplo (Usos de pliegai y de foldl, 1/2)

```
?- pliegai(suma, [1,2,3,4], 0, S).  % o foldl
S = 10  % suma los elementos de una lista
        % siendo suma(X,Y,Z) :- Z is X+Y.

?- pliegai(prod, [1,2,3,4], 1, S).  % o foldl
S = 24 % multiplica los elementos de una lista
       % siendo prod(X,Y,Z) :- Z is X*Y.

longitud(L, Long) :-              % longitud de lista
  pliegai(suma1, L, 0, Long). % con pliegai
% siendo suma1(Ac, _Elemento, NAc) :- NAc is Ac + 1.

longitud(L, Long) :-              % longitud de lista
  foldl(suma1, L, 0, Long).   % con foldl
% siendo suma1(_Elemento, Ac, NAc) :- NAc is Ac + 1.
```

Ejemplo (Usos de pliegai y de foldl, 2/2)

```
?- longitud([a,b,c], X).
X = 3 % calcula la longitud de una lista

?- pliegai(append, [[1,2],[3]], [], S).
S = [1, 2, 3] % concatena las listas de una lista

?- foldl(append, [[1,2],[3]], [], S).
S = [3,1,2] % concatena las listas en orden inverso
```

Los resultados de los dos últimos ejemplos difieren porque la operación `append`, a diferencia de la suma o el producto de los ejemplos anteriores, no es conmutativa.

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `imprimelista(+L)`, *cuyo efecto es escribir los elementos de la lista* `L` *en el fichero de escritura actual, uno por línea.*
2. `todospares(+L)`, *cierto si* `L` *es una lista compuesta exclusivamente por números naturales pares.*
3. `doblalista(+L1, ?L2)`, *cierto si* `L1` *es una lista de números enteros y* `L2` *contiene los mismos números pero multiplicados por dos.*

Soluciones propuestas:

1. *imprimelista(L) :- maplist(imprimeelemento, L).*
   imprimeelemento(E) :- write(E), nl.

2. *todospares(L) :- maplist(espar, L).*
   espar(N) :- integer(N), N >= 0, N mod 2 =:= 0.

3. *doblalista(L1, L2) :- maplist(integer, L1), maplist(dobla, L1, L2).*
   *dobla(I1, I2) :- I2 is I1*2.*

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `traducerestos(+L1, ?L2)`, *cierto si* `L1` *es una lista de números naturales y* `L2` *contiene los restos obtenidos al hacer la división entera de esos números por diez, escritos en castellano. Por ejemplo, la consulta* `?- traducerestos([1,13,20], X)` *debe devolver* `X=[uno, tres, cero].`
2. `opuestos(+L1, ?L2)`, *cierto si* `L1` *y* `L2` *son listas de números reales tales que, en cada posición,* `L2` *contiene el número opuesto de* `L1`. *Por ejemplo,* `?- opuestos([1,-2,3.5,-1.5],L)` *debe devolver* `L = [-1, 2, -3.5, 1.5].`

Soluciones propuestas:

1.
```
traducerestos(L1, L2) :-
maplist(procesa, L1, L2).

procesa(E, TE) :- integer(E), E >= 0, R is E mod 10, traduce(R, TE).

traduce(0, cero).
traduce(1, uno).
traduce(2, dos).
traduce(3, tres).
traduce(4, cuatro).
traduce(5, cinco).
traduce(6, seis).
traduce(7, siete).
traduce(8, ocho).
traduce(9, nueve).
```

2.
```
opuestos(L1, L2) :-
maplist(number, L1),
maplist(opuesto, L1, L2).

opuesto(X, Y) :- Y is -X.
```

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `multiplosN(+L,+N,?LMN)`, *cierto si* `L` *es una lista de números naturales,* `N` *es un natural positivo y* `LMN` *es la lista de los elementos de* `L` *que son múltiplos de* `N`. *Por ejemplo, la consulta* `?- multiplosN([1,2,3,4,5,6],2,L)` *debe devolver* `L = [2,4,6].`
2. `separaMultiplosN(+L,+N,?LMN, ?R)`, *cierto si* `L` *es una lista de números naturales,* `N` *es un natural positivo,* `LMN` *es la lista de los elementos de* `L` *que son múltiplos de* `N` *y* `R` *es una lista con los que no lo son. Por ejemplo, la consulta* `?- separaMultiplosN([1,2,3,4,5,6],2,LSi, LNo)` *debe devolver* `LSi = [2,4,6]` *y* `LNo = [1,3,5].`

Soluciones propuestas:

1.
```
multiplosN(L, N, LMN) :-
maplist(natural, [N|L]),
N > 0,
include(multiplo(N), L, LMN).

multiplo(N, E):- E mod N =:= 0.
```

2.
```
separaMultiplosN(L, N, LMN, R) :-
multiplosN(L, N, LMN),
exclude(multiplo(N), L, R).
```

Ejercicios (Uso de predicados de orden superior clásicos)

Ejercicio nº 3 de la Práctica de PROLOG nº 4.

Las soluciones comentadas a los ejercicios de esta práctica están disponibles en Práctica de PROLOG nº 4 con soluciones.

IMPLEMENTACIÓN DE NUEVOS PREDICADOS DE ORDEN SUPERIOR

- Los predicados de orden superior básicos `call/N` no solo permiten implementar los predicados de orden superior clásicos (de aplicación, filtrado y plegado) estudiados más arriba, sino que sirven también para implementar fácilmente nuevos predicados de orden superior cuya necesidad pueda surgir en el desarrollo de aplicaciones.
- La técnica para implementar nuevos predicados de orden superior es la misma que la descrita más arriba para los clásicos: se implementan igual que cualquier otro predicado en PROLOG, con la diferencia de que algunos de sus argumentos pueden ser otros predicados, desconocidos en tiempo de compilación, que se invocan mediante `call` y que se concretan y ejecutan en el momento de usar el predicado.

Ejemplo

`cuantos(+Obj,+L,?N)`, cierto si `N` es el número de elementos de la lista `L` sobre los que se aplica con éxito el objetivo `Obj`. Por ejemplo,
`?- cuantos(integer,[1,X,3],C).` daría `C = 2` y
`?- cuantos(var,[1,X,3],C).` daría `C = 1`.

```
% mediante filtrado
cuantos_f(Obj, L, N) :-
  include(Obj, L, NL),
  length(NL, N).
```

```
% implementación recursiva
cuantos(_, [], 0).
cuantos(Obj, [C|R], N) :-
    call(Obj,C),
    !,
    cuantos(Obj, R, NR),
    N is NR + 1.
cuantos(Obj, [_|R], N) :-
    cuantos(Obj, R, N).
```

Ejemplo

`map_pares(+Obj,?L1,?L2)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre todas las parejas de elementos de las listas `L1` y `L2` situados en posiciones pares (los segundos, los cuartos, etc). Por ejemplo, `?- map_pares(cuadrado,[1,2,3,4], L).` daría `L = [1,4,3,16].`

```
map_pares(_, [], []).

map_pares(_, [C], [C]).

map_pares(Obj, [C1,C2|R], [C1,NC2|NR]) :-
                call(Obj, C2, NC2),
                map_pares(Obj, R, NR).
```

Ejercicios (Recolección, orden superior)

1. *Proponga una implementación con recursión de cola para el predicado* `cuantos(+Obj,+L,?N)`*, cierto si* `N` *es el número de elementos de la lista* `L` *sobre los que se aplica con éxito el objetivo* `Obj` *(implementado en el penúltimo ejemplo con recursión no final).*

2. *Estudie la resolución en* PROLOG *del problema del coloreado de mapas (aquí puede encontrar el código).*

3. *Haga los ejercicios 4 y 5 de la Práctica de* PROLOG *nº 4. Las soluciones comentadas a los ejercicios de esta práctica están disponibles en Práctica de* PROLOG *nº 4 con soluciones.*

Soluciones propuestas:

```
% cuantos(+Obj,+L,?N)
% cierto si N es el número de elementos de la lista L
% sobre los que se aplica con éxito el objetivo Obj
% Implementación con recursión de cola
```

```
% añadir parámetro de acumulación con valor caso base
cuantosrc(Obj, L, N) :- cuantosrc(Obj, L, 0, N).


% caso base: se devuelve lo acumulado
cuantosrc(_, [], Ac, Ac).


% caso recursivo cuando C cumple Obj: se suma 1 al acumulador
cuantosrc(Obj, [C|R], Ac, N) :-
call(Obj, C),
!,
NAc is Ac + 1,
cuantosrc(Obj, R, NAc, N).


% caso recursivo cuando C NO cumple Obj: se mantiene el acumulador
cuantosrc(Obj, [_|R], Ac, N) :- cuantosrc(Obj, R, Ac, N).
```

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq