Ejemplos programación lógica pura

```
%Ejemplos con relaciones de parentesco
progenitor(luis,maria).
progenitor(ana,maria).
progenitor(luis,alberto).
progenitor(ana,alberto).
progenitor(maria,carlos).
progenitor(juan,carlos).
hombre(luis).
hombre(carlos).
mujer(maria).
mujer(ana).

hijo(X,Y) :- progenitor(Y,X), hombre(X).

abuela(X,Y) :- progenitor(Z,Y), madre(X,Z).
abuelo(X,Y) :- progenitor(Z,Y), padre(X,Z).

madre(X,Y) :- progenitor(X,Y), mujer(X).
padre(X,Y) :- progenitor(X,Y), hombre(X).

hermanos(X,Y):- progenitor(Z,X), progenitor(Z,Y), X\=Y.

tioa(X,Y) :- progenitor(Z,Y), hermanos(Z,X).

descendiente(X,Y) :- progenitor(Y,X).
descendiente(X,Y) :- progenitor(Y,Z), descendiente(X,Z).

%Aritmética no definida
suma(cero,Y,Y).
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).

%Ejemplos con listas
esLista([]).
esLista([_|Ys]) :- esLista(Ys).

pertenece(X,[X|_]).
pertenece(X,[_|Ys]) :- pertenece(X,Ys).


concatenar([],Xs,Xs).
concatenar([X|Xs],Ys,[X|Zs]) :- concatenar(Xs,Ys,Zs).

take(cero,_,[]).
take(_,[],[]).
take(s(N),[X|Xs],[X|Ys]) :- take(N,Xs,Ys).

invertir([],[]).
invertir([X|Xs],Ys) :- invertir(Xs,Zs), concatenar(Zs,[X],Ys).

inv(Xs,Ys) :- inv(Xs,[],Ys).
inv([],Xs,Xs).
inv([X|Xs],Ys,Zs) :- inv(Xs,[X|Ys],Zs).

%Ejemplos con árboles binarios
esArbol(avacio).
esArbol(nodo(_,I,D)) :- esArbol(I), esArbol(D).
```

```
perteneceArbol(X,nodo(X,_,_)).
perteneceArbol(X,nodo(_,I,_)) :- perteneceArbol(X,I).
perteneceArbol(X,nodo(_,_,D)) :- perteneceArbol(X,D).

preOrden(avacio,[]).
preOrden(nodo(X,I,D),[X|L]) :- preOrden(I,LI), preOrden(D,LD),
append(LI,LD,L).

inOrden(avacio,[]).
inOrden(nodo(X,I,D),L) :- inOrden(I,LI), inOrden(D,LD),
append(LI,[X|LD],L).

postOrden(avacio,[]).
postOrden(nodo(X,I,D),L) :- postOrden(I,LI), postOrden(D,LD),
append(LD,[X],LD2), append(LI,LD2,L).
```

Aritmética en Prolog

% La aritmética de Prolog se sale de la programación lógica pura
% El elemento clave es el predicado no reversible is/2 que requiere
% que el segundo parámetro esté instanciado antes de llamar a is

```
factorial(0,1).
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N*F1.
```

% Dado que is requiere que (N-1) esté instanciado, factorial requerirá que N esté instanciado
% por lo que factorial tampoco será reversible. Lo mismo pasará con los siguientes ejemplos.

```
incrementarTodos([],[]).
incrementarTodos([X|Xs],[X1|Ys]) :- X1 is X+1,
incrementarTodos(Xs,Ys).

take(0,_,[]).
take(_,[],[]).
take(N,[X|Xs],[X|Ys]) :- N>0, N1 is N-1, take(N1,Xs,Ys).

sumaVectores([],[],[]).
sumaVectores(([X|Xs],[Y|Ys],[Z|Zs]) :- Z is X+Y,
sumaVectores(Xs,Ys,Zs).
```

Predicados metalógicos simples

% Prolog incluye predicados que no se pueden representar en lógica de primer orden.
% Algunos ejemplos son:
% integer/1, float/1, number/1: Tienen éxito sii su parámetro está instanciado a un entero, float o
% a un número, respectivamente.
% atomic/1: Tiene éxito sii su parámetro está instanciado a una constante
% atom/1: Tiene éxito sii su parámetro está instanciado a a una constante no numérica

% En cualquiera de los casos anteriores, si el parámetro fuera una variable no instanciada, el predicado fallaría.

% Podemos usarlos para hacer un poco reversibles algunos predicados:

```
suma(X,Y,Z) :- number(X), number(Y), Z is X+Y.
suma(X,Y,Z) :- number(X), number(Z), Y is Z-X.
suma(X,Y,Z) :- number(Y), number(Z), X is Z-Y.
```

% También para tratar solo algunos elementos de una lista, por ejemplo sumando solo los que sean números:

```
sumaNumeros([],0).
sumaNumeros([X|Xs],Z) :- number(X), sumaNumeros(Xs,Y), Z is X+Y.
sumaNumeros([_|Xs],Z) :- sumaNumeros(Xs,Z).
```

% aunque la definición anterior tiene un problema que ya resolveremos más adelante usando un "corte"

% Otros ejemplos de predicados metalógicos:
% var/1 tiene éxito sii su parámetro es una variable libre
% nonvar/1 tiene éxito sii su parámetro no es una variable libre, aunque pueda tener variables
% libres, por ejemplo nodo(3,I,D)
% ground/1 tiene éxito sii su parámetro no contiene ninguna variable libre
% ==/2 tiene éxito sii sus dos parámetros son idénticos
% \==/2 tiene éxito sii sus dos parámetros no son idénticos

% En cualquiera de los casos anteriores, si el parámetro fuera una variable no instanciada, el
%predicado fallaría.

Corte: CONTROL MEDIANTE CORTE Y NEGACIÓN

/*El predicado de corte es fundamental en Prolog, pero es un predicado impuro, es decir, está fuera de la programación lógica.

-Es un predicado sin parámetros
-Se escribe !
-Siempre tiene éxito
-Como efecto lateral "poda" alternativas del árbol de búsqueda

Supongamos que tenemos algo así como

predicado :- ...
predicado :- p1, p2, ..., pn, !, q1, q2, ..., qm.
predicado :- ...

Si intentamos evaluar predicado, todo funcionará como habitualmente, haciendo backtracking si hiciera falta tanto con la primera regla como con los pasos pi de la segunda regla.
Ahora bien, si llegamos al corte, entonces descartaríamos todas las reglas relativas a predicado que vengan después (en nuestro ejemplo, la tercera regla) e impediríamos que el backtracking volviera a las pi o a las reglas anteriores correspondientes a predicado.

Podemos interpretar el corte como una compuerta que es trivial de pasar de izquierda a derecha (siempre tiene éxito) pero que es imposible de pasar de derecha a izquierda (una vez que la cruzamos, ya no podemos dar marcha atrás para probar otras alternativas).

Cabe destacar que una vez cruzada la compuerta del corte, sí podremos seguir haciendo backtracking entre las qi, probando tantas alternativas como haga falta entre las qi, pero no podemos volver a antes del corte.

*/

%Recordemos el problema que vimos anteriormente que generaba más soluciones de la cuenta.
%Ahora podemos asegurarnos de que la última regla solo se aplica si X no es un número:

```
sumaNumeros([],0).
sumaNumeros([X|Xs],Z) :- number(X), !, sumaNumeros(Xs,Y), Z is X+Y.
sumaNumeros([_|Xs],Z) :- sumaNumeros(Xs,Z).

%Nos puede valer para no seguir probando opciones si sabemos que no
será necesario:
maximo(X,Y,X) :- X>Y, !.
maximo(X,Y,Y) :- X=<Y.
```

%Pero debemos tener mucho cuidado para no hacer cosas incorrectas como la siguiente:

```
max(X,Y,X) :- X>Y, !.
max(X,Y,Y).
```

%que nos daría un resultado incorrecto en algunas situaciones en las que instanciamos
%los tres parámetros, por ejemplo: ?max(2,1,1). nos devolvería que es cierto.

/* En general, el corte nos permitirá reducir considerablemente los árboles de búsqueda, y nos facilitará determinadas tareas, pero deberemos ser muy cuidadosos porque nos cambiará radicalmente el comportamiento de nuestros programas.

Por ejemplo, supongamos que queremos reescribir el predicado que nos decía si un elemento pertenecía a una lista, de modo que no queramos seguir buscando en cuanto hayamos encontrado que sí pertenece. Entonces podremos hacer lo siguiente:
*/

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

/* Ahora bien, el predicado anterior valdrá para ver si un elemento pertenece a una lista, pero si preguntamos
? pertenece(X,[1,2,3]),pertenece(X,[2,3]).
nos devolverá que no, porque en cuanto el primer pertenece encuentra que con X=1 se satisface, ya no se permite probar con otro valor para la X, así que no llegamos a probar con X=2, que sería el primer valor que cumpliría los dos pertenece. */

Negación por fallo

/*¿Es posible definir un predicado que se comporte como la negación de otro?
NO, pero podemos hacer algo parecido sacando partido tanto del corte como del predicado predefinido fail/0 que siempre falla:

p(X) :- q(X), !, fail.
p(_).

Si q(X) tiene éxito, entonces ponemos un corte para no poder dar marcha atrás, pero justo después de eliminar el resto de opciones gracias a dicho corte, decimos explícitamente que queremos fallar, por lo que p(X) fallará.

Si q(X) TERMINA sin tener éxito, entonces la primera regla falla sin llegar al corte, por lo que sigue probando con la segunda regla, que sí que tiene éxito, por lo que p(X) resulta que sí tiene éxito.

Nótese que si q(X) no termina, entonces p(X) tampoco termina. Por eso, p no es exactamente la negación de q. La negación por fallo es una aproximación a la negación lógica, PERO NO ES LA NEGACIÓN LÓGICA.

Podemos generalizar la idea si usamos el predicado call, que permite invocar un objetivo desde el programa:

not(Objetivo) :- call(Objetivo), !, fail.
not(_).

Nótese que la negación por fallo nunca puede instanciar variables, lo cual tiene implicaciones a la hora de usarlo. Veamos un ejemplo:
*/

```
adulto(anacleto).
hombre(anacleto).
hombre(bonifacio).

hombreYMenor(X) :- hombre(X), not(adulto(X)).

menorYHombre(X) :- not(adulto(X)), hombre(X).
```

/* menorYHombre y hombreYMenor se comportan igual si el parámetro de entrada ya está completamente
instanciado. Ahora bien, menorYHombre se comporta de forma muy diferente a hombreYMenor si el parámetro
es una variable libre.

hombreYMenor(X) se interpreta como: hay un hombre X que además no es adulto, que en nuestro caso se cumple con X=bonifacio

Ahora bien, menorYHombre(X) se interpreta como: no hay ningún adulto, y además hay un hombre,
pero como sí que hay al menos un adulto (anacleto), entonces menorYHombre(X) falla.*/

If then else

/*Uso del punto y coma ";": DISYUNCIÓN e IF-THEN-ELSE

La disyunción lógica en un predicado se suele representar usando una regla para cada alternativa, como en

descendiente(X,Y) :- progenitor(Y,X).
descendiente(X,Y) :- progenitor(Y,Z), descendiente(X,Z).

Ahora bien, también es posible hacerlo usando ; Por ejemplo: */

```
descendiente(X,Y) :- progenitor(Y,X) ; progenitor(Y,Z),
descendiente(X,Z).

progenitor(luis,maria).
progenitor(ana,maria).
progenitor(luis,alberto).
progenitor(ana,alberto).
progenitor(maria,carlos).
progenitor(juan,carlos).
hombre(luis).
hombre(carlos).
mujer(maria).
mujer(ana).
```

/*Podemos definir una estructura IF-THEN-ELSE sacando partido del corte:

siPEntoncesQSiNoR(X) :- P(X), !, Q(X).
siPEntoncesQSiNoR(X) :- R(X).

Cuando se cumple P(X), eliminamos la segunda regla, y nos quedamos como única opción con Q(X).
Por contra, si P(X) termina y falla, entonces no evaluamos ni el corte ni Q(X), nos quedamos simplemente con la opción de R(X).
Prolog incorpora una notación especial para estos casos:
P(X) -> Q(X) ; R(X)

Recordemos el ejemplo

sumaNumeros([],0).
sumaNumeros([X|Xs],Z) :- number(X), !, sumaNumeros(Xs,Y), Z is X+Y.
sumaNumeros([_|Xs],Z) :- sumaNumeros(Xs,Z).

podemos reescribirlo como*/

```
sumaNumeros([],0).
sumaNumeros([X|Xs],Z) :- number(X) -> sumaNumeros(Xs,Y), Z is X+Y ;
sumaNumeros(Xs,Z).
```

Inspección estructuras y orden superior

/*INSPECCIÓN DE ESTRUCTURAS

Existen predicados que nos permiten acceder tanto a los functores como a los argumentos de cualquier término:

functor(T,F,A) tiene éxito si el nombre del functor de T es F y además su aridad es A.
-Puede usarse pasando T como entrada para que nos devuelva tanto F como A como salida
-Puede usarse pasando F, A como entrada para que nos devuelva T como salida
-Puede usarse pasando los tres parámetros de entrada

arg(N,T,A) tiene éxito si el N-ésimo argumento del término T es A.
-Puede usarse pasando N, T como entrada para que nos devuelva A como salida.
-Puede usarse pasando los tres parámetros de entrada
-En SWI también puede usarse pasando T como entrada para que devuelva N, A como salida

T=..L tiene éxito si L es una lista que contiene como primer elemento el functor de T y como resto de elementos todos los argumentos de T.
-Puede usarse pasando T como entrada para que nos devuelva L
-Puede usarse pasando L como entrada para que nos devuelva T
-Puede usarse pasando tanto T como L de entrada

ORDEN SUPERIOR

El predicado call/1 es la base para permitir introducir orden superior en Prolog.
call(X) evalúa el término X como se evaluaría cualquier objetivo, unificando variables si fuera preciso.

¿Cómo sacamos partido de call/1 para el orden superior? Supongamos que dado un cierto predicado, queremos ver si todos los elementos de una lista cumplen dicho predicado o no:*/

```
todosCumplen(_,[]).
todosCumplen(P,[X|Xs]) :- Objetivo=..[P,X], call(Objetivo),
todosCumplen(P,Xs).
%En algunos entornos está permitido hacer directamente Objetivo=P(X)
```

/* La idea general para usar orden superior será usar =.. para construir el objetivo que queramos a partir del functor que nos pasen como parámetro del predicado de orden superior, así como a partir de los parámetros que tengamos que pasar a dicho predicado. Una vez construido el objetivo, lo invocamos.*/

% Veamos otro ejemplo donde haremos algo equivalente al filter de Haskell:

```
filter(_,[],[]).
filter(P,[X|Xs],[X|Ys]) :- Objetivo=..[P,X], call(Objetivo), !,
filter(P,Xs,Ys).
filter(P,[_|Xs],Ys) :- filter(P,Xs,Ys).
```

% Nótese que el corte nos asegura que si llegamos a la tercera regla es porque estamos seguros de que no se cumplía P(X).

% Otro ejemplo:

```
takeWhile(P,[X|Xs],[X|Ys]) :- Objetivo=..[P,X], call(Objetivo), !,
takeWhile(P,Xs,Ys).
takeWhile(_,_,[]).

hombre(anacleto).
hombre(bonifacio).
```

% Se puede probar filter(hombre,[anacleto,bonifacio,maria,bonifacio],L)

Predicados de agregación

/*Los predicados de agregación nos permiten recolectar todas las soluciones que satisfacen un objetivo
  findall(T,Objetivo,L) devuelve en L la lista de todos los términos T que satisfacen Objetivo
Por ejemplo
  findall(X,hermanos(X,maria),L).
devolvería en L la lista de todos los hermanos de maria
  findall(pareja(X,Y),descendiente(X,Y),L).
devolvería en L la lista de todas las parejas (X,Y) tales que X es descendiente de Y.
  findall(X,(hombre(X),adulto(X)),L).
devolvería en L la lista de todos los hombres adultos.

El predicado setof/3 se comporta básicamente igual que findall/3 con la única diferencia de que setof/3 devuelve la lista ordenada y sin repeticiones.*/

```
progenitor(luis,maria).
progenitor(ana,maria).
progenitor(luis,alberto).
progenitor(ana,alberto).
progenitor(maria,carlos).
progenitor(juan,carlos).
hombre(luis).
hombre(carlos).
mujer(maria).
mujer(ana).
hermanos(X,Y):- progenitor(Z,X), progenitor(Z,Y), X\=Y.

hijos(X,L) :- setof(Y,progenitor(X,Y),L).
misHermanos(X,L) :- setof(Y,hermanos(X,Y),L).
misHermanosConRepeticiones(X,L) :- findall(Y,hermanos(X,Y),L).
```

Modificación dinámica de predicados

/* Un programa Prolog puede modificarse a sí mismo, lo cual es especialmente útil si queremos implementar sistemas expertos a los que se van añadiendo nuevas reglas dinámicamente. Puede resultar bastante útil, pero debe hacerse con mucho cuidado, pues puede ser muy complicado entender qué implicaciones tienen dichas modificaciones.

Recordemos que un programa Prolog es un conjunto de reglas. Para modificar dinámicamente un programa se proporcionan predicados que permiten añadir o eliminar reglas:

asserta(C) añade la cláusula C al principio del conjunto de reglas del predicado correspondiente a C
assertz(C) añade la cláusula C al final del conjunto de reglas del predicado correspondiente a C
retract(C) elimina la primera cláusula que unifica con C
retractall(C) elimina todas las cláusulas que unifican con C*/

% Para que un predicado que aparece en un programa pueda ser modificado dinámicamente es
% necesario definirlo como dinámico:

```
:- dynamic progenitor/2.
progenitor(luis,maria).
progenitor(ana,maria).
progenitor(luis,alberto).
progenitor(ana,alberto).
progenitor(maria,carlos).
progenitor(juan,carlos).
```

/*Para ver cómo se va modificando dinámicamente la definición de un predicado, podemos usar el predicado listing. Por ejemplo

?listing(progenitor/2).

nos mostrará la definición actual del predicado progenitor .Si después vamos haciendo modificaciones, podremos ir viendo los cambios si llamamos otra vez a listing(progenitor/2).

?asserta(progenitor(maria,belen)).

?listing(progenitor/2).

?retract(progenitor(luis,X)).

?listing(progenitor/2).

?retractall(progenitor(X,carlos)).

El uso de estos predicados se puede complicar tanto como queramos, haciendo definiciones retorcidas del estilo

?assertz((p(A):-assertz(p(A)),fail)).
?p(a).
?p(X),p(b).*/

/*Puede ser útil para almacenar conocimiento que vayamos calculando. Por ejemplo, podemos calcular fácilmente la tabla de multiplicar utilizando un "bucle de fallo":*/

```
tabla(L):-member(X,L), member(Y,L), Z is X*Y, assertz(mult(X,Y,Z)),
fail.
```

/*Si evaluamos
 ?tabla([1,2,3,4,5,6,7,8,9,10]).
obtnemos que se genera todo el conocimiento de la tabla demultiplicar, que podemos ver con ?listing(mult/3).*/

/* Uno de los ejemplos más típicos de uso es mejorar la definición típica de Fibonacci almacenando resultados intermedios para evitar que se repitan:*/

```
:-dynamic fibAux/2.
fibAux(0,1).
fibAux(1,1).

fib(N,F) :- fibAux(N,F), !.
fib(N,F) :- N1 is N-1, N2 is N-2, fib(N1,F1), fib(N2,F2), F is F1+F2,
assertz(fibAux(N,F)).
```

/* Nótese que la definición de fib es muy similar a la definición trivial de fibonacci, pero delegando todos los casos base en un predicado auxiliar fibAux. Inicialmente solo tenemos dos casos base recogidos en fibAux, pero a medida que se van calculando nuevos
valores de fibonacci, el assertz correspondiente va añadiendo "casos base" a fibAux, de modo que no sea necesario repetir ningún cómputo. Nótese también que usamos un corte al final de la primera regla de fib para no buscar más alternativas una vez que ya tenemos
un caso base. Nótese finalmente que el único predicado dinámico es fibAux, mientras que fib no lo es.*/

/*Otro ejemplo simple de uso de modificación dinámica del programa puede ser el siguiente, donde lo que hacemos es crear un contador que lleve la cuenta de cuántas veces hemos llamado al predicado contar/1*/

```
:-dynamic contador/1.
contador(0).

contar :- contador(X), Y is X+1, retract(contador(X)),
asserta(contador(Y)).
```

Entrada/Salida

/*Los predicados más básicos de Prolog para realizar entrada/salida son los siguientes:

read(T) lee un término de la entrada estándar y lo unifica con T. El término debe escribirse acabando con un punto.
write(T) escribe el término T en la salida estándar
display(T) escribe el término T en la salida estándar
writeq(T) y displayq(T) también escriben en la salida estándar pero añaden comillas si hace falta
nl escribe un salto de línea*/

```
buscaProgenitor :- write('Dime el nombre del hijo: '), read(H), nl,
progenitor(P,H), write('Su progenitor es '),
                    write(P), nl.

progenitor(luis,maria).
progenitor(ana,maria).
progenitor(luis,alberto).
progenitor(ana,alberto).
progenitor(maria,carlos).
progenitor(juan,carlos).
```

/* Para leer y escribir en fichero, una forma sencilla es modificar la entrada estándar y/o la salida estándar
para que sean el fichero que queramos.

see(Archivo) hace que a partir de ahora todas las lecturas de la entrada estándar (ejem read(T)) se hagan desde Archivo
seeing(Archivo) devuelve en Archivo cuál es el archivo que se está usando ahora como entrada estándar (devuelve
user si se está usando realmente la entrada estándar)
seen/0 cierra la entrada estándar actual, dejando como entrada estándar la que tuviéramos antes del último see/1
tell(Archivo) hace que a partir de ahora todas las escrituras de la entrada estándar se hagan a Archivo
telling(Archivo) devuelve en Archivo cuál es el archivo que se está usando ahora como salida estándar (user si estándar real)
told/0 cierra la salida estándar actual, dejando como salida estándar la que tuviéramos antes del último tell/1 */

```
escribeEnFichero :- write('Dime nombre de fichero:'), read(Nombre),
tell(Nombre), write('Estoy escribiendo'), told,
                    writeq('Y ahora escribo en la pantalla').
```

/*También es posible abrir varios ficheros de lectura o varios de escritura sin necesidad de perder acceso a la entrada o a la salida estándar. Como en la mayor parte de lenguajes, existen predicados para abrir fichero, para leer o escribir de fichero y para cerrar fichero:

open(Fichero,Modo,Acceso) abre el fichero Fichero en modo Modo (que puede ser read, write o append). Fichero y Modo deben ser datos de entrada. Devuelve en Acceso acceso al fichero, que podrá usarse con los siguientes predicados.
read(Acceso,Termino) write(Acceso,Termino) nl(Acceso) se comportan como read, write, nl pero leyendo del fichero al que tenemos acceso a través de Acceso
close(Acceso) cierra el acceso al fichero Acceso*/

```
otroEscribeTablaEnFichero:-open('tabla2.txt',write,F),
(tabla(F,[1,2,3,4,5,6,7,8,9,10]);close(F)).

escribeTablaEnFichero:-open('tabla.txt',write,F), escribeCon(F).
escribeCon(F):-tabla(F,[1,2,3,4,5,6,7,8,9,10]).
escribeCon(F):-close(F).
tabla(F,L):-member(X,L), member(Y,L), Z is X*Y, write(F,X*Y=Z), nl(F),
fail.
```

Listas diferencia

/*En Prolog se pueden usar "estructuras incompletas" para mejorar la eficiencia de algunos programas. El ejemplo típico de estructura incompleta es la definición y uso de listas diferencia. La idea básica es aprovechar las variables lógicas para representar punteros a partes aún no definidas. Recordemos la definición de append para concatenar listas: */

```
concatenar([],Ys,Ys).
concatenar([X|Xs],Ys,[X|Zs]) :- concatenar(Xs,Ys,Zs).
```

/*La definición anterior funciona perfectamente. Ahora bien, supongamos que queremos concatenar una primera lista de 1000 elementos con otra de 10. Para ello, la definición que hemos dado tendrá que dar 1000 pasos para ir procesando todos los elementos de la primera lista. ¿Sería posible hacerlo (mucho) más rápido?
En imperativo, la idea pasaría por representar las listas de otra forma, teniendo acceso no solo al primer elemento de la lista, también al último (usando un puntero para acceder a cada extremo de la lista). Si tenemos esa opción, para concatenar dos listas solo tenemos que dar un paso para acceder al último elemento, luego enlazar el último elemento de la primera lista con el primero de la segunda y finalmente indicar que el nuevo final de la lista es el que antes era el final de la segunda lista.
En Prolog podemos hacer algo parecido, usando una variable lógica abierta a modo de puntero (aunque realmente no es un puntero) a través del cual completar el resto de la lista. Dicha variable irá al final de la lista, como si la lista no estuviera completamente definida (porque de hecho no está completamente definida). Una lista diferencia se representará como un par donde el primer elemento es la lista completa y el segundo elemento es la parte final de la lista que todavía está "abierta": ([a,b,c,d|Resto),Resto)
Nótese que si ahora unificamos Resto con cualquier lista, por ejemplo, [e,f], no necesitamos recorrer la lista [a,b,c,d] para llegar hasta Resto, porque ya tenemos acceso directo a Resto. Así pues, directamente obtendríamos ([a,b,c,d,e,f],[e,f]) de forma más rápida que con append.

Normalmente representaremos listas diferencia como L-R o como L\R, siendo L la lista completa y R el "hueco" para completar el resto de la lista.
Veamos cómo hacer la concatenación de listas diferencia:*/

```
appendDif(L1-R1,L2-R2,L3-R3) :- R1=L2, L3=L1, R3=R2.
```

% O más corto todavía:

```
otroAppendDif(L1-R1,R1-R2,L1-R2).
```

% Podéis probarlo con appendDif([1,2,3|X]-X,[4,5|Y]-Y,L-R).

/* Recordemos que no podemos usar appendDif ni otroAppendDif con listas normales, porque solo está definido para listas diferencias, no para listas. Ahora bien, podemos convertir las listas diferencia en listas normales, unificando el resto de la lista con []. Es decir, si unificamos [a,b,c|R]-R con L-R nos queda en L una lista normal [a,b,c]. Y las listas normales también las podemos convertir en listas diferencia: */

```
listaAListaDif([],L-L).
listaAListaDif([X|Xs],[X|Y]-Z) :- listaAListaDif(Xs,Y-Z).
% Probadlo con listaAListaDif([1,2,3,4],L-R).
```

/* Como ejemplo final, supongamos que queremos hacer una rotación de modo que el primer elemento de la lista pase a ser el último. Con listas normales tendríamos que hacer algo así: */

```
rotacion([],[]).
rotacion([X|Xs],Ys) :- append(Xs,[X],Ys).
```

/* Que lógicamente tiene coste O(n) porque requiere pasar por todos los elementos de la lista para añadir uno al final. Sin embargo, con listas diferencia tendría coste O(1): */

```
rotacionDif(L-L,L-L) :- var(L), !.
rotacionDif([X|Xs]-R,Xs-R1) :- R=[X|R1].
% Probadlo con rotacionDif([1,2,3,4|R]-R,L-M).
```

/* O incluso más corto: */

```
otraRotacionDif(L-L,L-L) :- var(L), !.
otraRotacionDif([X|Xs]-[X|R1],Xs-R1).
```