Programación Declarativa

3er Curso, Grado en Ingeniería en Informática
Universidad Rey Juan Carlos

Programación Lógica

Prácticas en SWISH Prolog

Durante el curso se realizan cuatro prácticas de Prolog mediante los **notebooks** ofrecidos por la herramienta online de SWI-Prolog, SWISH. Puede consultar las instrucciones para su realización aquí.

Este documento recopila las cuatro prácticas mencionadas. Todos los ejercicios y comentarios incluidos en ellas están basados en fuentes diversas, en particular en las siguientes:

- 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

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº1: Primeros contactos con el lenguaje

Soluciones propuestas, comentadas

Prerrequisitos
Para la realización de esta primera práctica de Programación Lógica es conveniente haber leído y estudiado el tema PL1 (Introducción a la Programación Lógica) y los apartados 1,2 y 3 del tema PL2 (El lenguaje Prolog: aspectos básicos), en los que se describen las características generales del lenguaje Prolog así como su sintaxis y semántica.

1. Suma de números naturales en lógica "pura"

Considere el predicado lógico puro para la suma de números naturales discutido en los apuntes, `"suma/3"`, cuya implementación, precedida de comentarios (líneas que empiezan con el símbolo `%`) se recuerda a continuación:

```
% suma(?X, ?Y, ?Z)
% cierto si Z es la suma de X e Y.
% programa lógico puro (no utiliza la aritmética de Prolog)

% Caso base:
% "Para cualquier X, la suma de X con 0 es X."
% Con la notación aritmética estándar: X+0 = X.

suma(X,0,X).

% Caso recursivo:
% "Para cualesquiera X,Y,Z, si Z es la suma de X e Y, entonces
% s(Z) es la suma de X y s(Y)"
% o, equivalentemente,
% "Para cualesquiera X,Y,Z, la suma de X con s(Y) es s(Z),
% siempre que Z sea la suma de X con Y."
% Con la notación aritmética estándar: X+Y=Z => X+(Y+1) = Z+1.

suma(X,s(Y),s(Z)) :-
    suma(X,Y,Z).
```

1.1. Algunos errores sintácticos y semánticos típicos

A continuación se recuerdan, ilustrándolos con el código de `suma/3`, dos de los errores sintácticos típicos:

- Incluir espacios en blanco entre el nombre del predicado y el paréntesis que viene a continuación. Por ejemplo, escribir `suma (X,0,X).` en lugar de `suma(X,0,X).` produciría un error sintáctico.

- Olvidar que todo hecho y toda regla de un programa terminan necesariamente con un punto. Por ejemplo, escribir `suma(X,0,X)` en lugar de `suma(X,0,X).` produciría un error sintáctico.

Otro error típico, esta vez semántico, se produce al equivocarse al teclear el nombre de una variable. Por ejemplo, si al escribir la primera línea del programa anterior se introduce por error una C en lugar de una X, escribiendo `suma(X,0,C).` , el intérprete producirá el aviso `Singleton variables: [X,C]` . En este caso no se trata de un error sintáctico sino semántico: el código se puede ejectar pero no funcionará correctamente, puesto que la suma de un X dado con 0 no es igual a cualquier valor C, sino que es igual al valor X de partida.

Pruebe las consecuencias que tendrían estos errores introduciéndolos en el código dado más arriba y haciendo alguna consulta, como la que se propone a continuación (que debe tener como única respuesta `Cuanto=0`), para forzar que el código de `suma/3` sea interpretado por Prolog.

```
?- suma(0,0,Cuanto).
```

No olvide dejar el código sin errores sintácticos ni semánticos puesto que se va a utilizar a continuación.

1.2. Realización de algunas consultas

En lo que sigue se practica con el uso del predicado `suma/3` propuesto más arriba. Recuerde que la implementación está basada en programación lógica “pura”, sin utilizar la aritmética de Prolog, y que por lo tanto los números naturales deben representarse mediante la constante 0 (cero) y los términos compuestos s(0) (uno), s(s(0)) (dos), s(s(s(0))) (tres), etc. Si intenta utilizar el predicado suma de forma distinta, haciendo por ejemplo la consulta ?- suma(1, 2, X). verá que el resultado es false. (En efecto, como se estudiará más adelante, la consulta anterior no unifica -"casa"- con la cabeza de ninguna de las dos cláusulas que conforman el programa, puesto que la constante 2 no unifica ni con la constante 0 ni con el término compuesto s(Y)).

Antes de ejecutar las consultas que se proporcionan a continuación, indique qué se pretende averiguar con ellas (puede escribir en los espacios marcados con ...) y qué respuesta(s) cree que deberían producir. Por ejemplo:

consulta 1: ¿Existen naturales X que sean la suma de 1 más 1? ¿Cuáles? Respuesta esperada: X=s(s(0))

```
?- suma(s(0), s(0), X).
```

consulta 2: ¿Existen naturales Y tales que sumados a 1 dan 3? = ¿Existen naturales que sean el resultado de restar 1 a 3? ¿Cuáles? Respuesta: s(s(0))

```
?- suma(s(0), Y, s(s(s(0)))).
```

consulta 3: ¿Existe algún Y (no interesa su valor) tal que sumado a 1 da 3? = Existe algún Y (no interesa su valor) que sea igual a 3-1? Respuesta: true

```
?- suma(s(0), _, s(s(s(0)))).
```

*Observe la diferencia en las respuestas a las dos consultas anteriores, debida al uso de la **variable anónima** "_" en la segunda.*

consulta 4: ¿Existen naturales X,Y cuya suma sea igual a 3? ¿Cuáles? Respuestas: [X=3,Y=0], [X=2,Y=1], [X=1,Y=2], [X=0,Y=3] (con los naturales escritos con la notación de la lógica pura)

```
?- suma(X, Y, s(s(s(0)))).
```

consulta 5: ¿Existen naturales X tales que sumados consigo mismos dan 3? ¿Cuáles? Respuesta: false

```
?- suma(X, X, s(s(s(0)))).
```

consulta 6: ¿Existen naturales X tales que sumándoles algo dan 3? ¿Cuáles? Respuesta: X=3, X=2, X=1, X=0 (con los naturales escritos con la notación de la lógica pura)

```
?- suma(X, _, s(s(s(0)))).
```

Pruebe ahora el ejercicio inverso: escriba y ejecute las consultas en Prolog asociadas con las descripciones dadas.

consulta 7: ¿existe algún natural tal que sumado consigo mismo da 2? ¿Cuál(es)?

```
?- suma(X,X,s(s(0))).
```

consulta 8: ¿existe algún natural tal que sumado consigo mismo da 3? ¿Cuál(es)?

```
?- suma(X,X, s(s(s(0)))).
```

consulta 9: ¿Hay naturales menores o iguales que 3? ¿Cuál(es)?
Pista: un natural X es menor o igual que otro Y si existe un tercer natural Z tal que X+Z=Y.

```
?- suma(X, _, s(s(s(0)))).
```

consulta 10: ¿Hay naturales estrictamente menores que 3? ¿Cuál(es)?
Pista: un natural X es estrictamente menor que otro Y si existe un tercer natural Z *no nulo* tal que X+Z=Y.

```
?- suma(X, s(_), s(s(s(0)))).
```

Observe que todo número natural, salvo el 0, es el sucesor de alguien, por lo que todo número natural no nulo unificará con `s(_)` .

1.3. Uso de `suma/3` para implementar otros predicados

Utilice exclusivamente el predicado `suma/3` para implementar los dos siguientes predicados lógicos puros. Inspírese para ello en las cuatro últimas consultas del apartado anterior. A continuación, pruebe su funcionamiento mediante consultas adecuadas (al menos tres consultas para cada uno de ellos, variando las condiciones de entrada/salida de los argumentos).

```
1 % par(?X)
2 % cierto si X es un natural par
3 % (predicado lógico puro, sin usar la aritmética de Prolog)
4 %
5   par(X) :-
6     suma(Y, Y, X).
7 % un número es par si es igual a 2Y = Y+Y para un cierto Y
```

Pruebe su código realizando consultas significativas, como por ejemplo: ¿es 0 par? ¿es s(s(s(0))) par? ¿Existen números pares? ¿Cuáles son?

```
?- par(0).
```

```
?- par(s(0)).
```

```
?- par(s(s(0))).
```

```
?- par(X).
```

```
% menorig(?X, ?Y)
% cierto si X e Y son naturales y X es menor o igual que Y
% (predicado lógico puro, sin usar la aritmética de Prolog)
%
  menorig(X, Y) :-
    suma(X, _, Y).
  % X es menor o igual que Y si sumándole algo resulta igual a Y
```

Pruebe su código realizando consultas significativas:

```
?- menorig(0, _).
```

```
?- menorig(X, 0).
```

```
?- menorig(s(0), s(0)).
```

```
?- menorig(s(s(0)), s(0)).
```

```
?- menorig(X, s(s(0))).
```

```
?- menorig(X,Y).
```

2. Genealogía

Uno de los ejemplos más habituales para iniciarse en el manejo del lenguaje Prolog es el de las relaciones familiares. A continuación se facilita la implementación de algunos predicados básicos, y se pide la implementación de otros. Eche un vistazo al siguiente programa y continúe leyendo.

```
% PREDICADOS BÁSICOS

% progenitor(?X, ?Y)
% Cierto si X es progenitor (madre o padre) de Y
progenitor(pepa, pepe1).
progenitor(pepe, pepe1).
progenitor(pepe, pepa2).
progenitor(pepe1, pepa11).
progenitor(pepe1, pepa12).
progenitor(pepa12, pepe121).

```

```
% mujer(?X)
% Cierto si X es una mujer
mujer(pepa).
mujer(pepa2).
mujer(pepa11).
mujer(pepa12).

% varon(?X)
% Cierto si X es un varón
varon(pepe).
varon(pepe1).
varon(pepe121).
```

2.1. Realización de algunas consultas

Utilice exclusivamente los predicados `progenitor`, `mujer` y `varon` para realizar las consultas que se indican a continuación:

¿Es pepa la madre de pepa2?

```
?- progenitor(pepa, pepa2), mujer(pepa).
```

¿Quiénes son los progenitores de pepe1?

```
?- progenitor(X, pepe1).
```

¿Quién es la madre de pepe1?

```
?- progenitor(X, pepe1), mujer(X).
```

¿Tiene pepe1 hijos varones? ¿Quién o quiénes son?

```
?- progenitor(pepe1, X), varon(X).
```

¿Quiénes son mujeres?

```
?- mujer(X).
```

¿Tiene pepa alguna nieta? ¿Quién o quiénes son? OJO: Prolog debe mostrar *solo* las nietas de pepa, sus progenitores no deben aparecer. Recuerde el uso de variables *semianónimas* para resolver este tipo de situaciones.

Una primera aproximación para resolver esta consulta es la siguiente:

```
?- progenitor(pepa,Y), progenitor(Y,N), mujer(N).
```

Las respuestas dadas por Prolog a la consulta anterior son correctas (asocia a N las dos nietas de pepa, pepa11 y pepa12) pero facilita además los valores de los progenitores intermedios en la variable "Y", información que no se requiere. Ejecute la siguiente consulta para comprobar que la variable anónima "_" NO sirve para solucionar este problema:

```
?- progenitor(pepa,_), progenitor(_,N), mujer(N).
```

En efecto, la consulta anterior, además de pepa11 y pepa12, facilita la respuesta pepa2, que no es nieta de Pepa. Esto se debe a que la variable anónima, a diferencia del resto de variables, no tiene por qué unificar consigo misma, es decir, la primera y la segunda aparición de "_" pueden unificar con cosas distintas. Por ello, en la consulta anterior Prolog comprueba que Pepa es progenitora de alguien (no importa de quién) y luego comprueba que N es hija de alguien (tampoco importa de quién) y que es mujer, y estas condiciones también las cumple pepa2 (además de pepa11 y pepa12).

Para obtener las nietas de pepa de forma correcta y sin que aparezcan sus progenitores hay que utilizar ***variables semianónimas****, que son variables que empiezan por dos símbolos de subrayado, "__", o bien por un subrayado seguido de una mayúscula. Compruebe con la consulta incluida a continuación cómo estas variables no aparecen en las soluciones pero sin embargo sí que unifican entre sí (ya no aparece pepa2 como solución porque las dos apariciones de* `_Y` *tienen que ser las mismas):*

```
?- progenitor(pepa,_Y), progenitor(_Y,N), mujer(N).
```

¿Quiénes son progenitores comunes de pepa11 y pepa12?

```
?- progenitor(X, pepa11), progenitor(X, pepa12).
```

¿Tienen pepa11 y pepa12 algún progenitor/a común (no importa quién o quiénes)?

```
?- progenitor(_X, pepa11), progenitor(_X, pepa12).
```

Note de nuevo el uso de una variable semianónima.

2.2. Implementación de algunos predicados

Implemente y pruebe con consultas adecuadas los predicados descritos a continuación, teniendo en cuenta las indicaciones facilitadas para algunos de ellos.

`madre(?X,?Y), cierto si X es madre de Y.`

Este predicado, al igual que los siguientes, debe definirse mediante *reglas* que sean válidas independientemente del contenido concreto que tengan los predicados "progenitor", "mujer" y "varón", y no mediante hechos que habría que actualizar si cambiasen los datos actuales. Por ejemplo, una forma de definirlo sería la siguiente:

```
% madre(?X,?Y), cierto si X es madre de Y.
% versión INADECUADA
madre(pepa, pepe1).
madre(pepa12, pepe121).
```

Esta definición es correcta respecto a la definición actual de los predicados "progenitor" y "mujer", pero podría dejar de serlo si estos cambian. La forma adecuada de definir el predicado "madre" es hacerlo teniendo en cuenta su definición para el caso general: para cualesquiera dos personas X e Y, X será la madre de Y si resulta que X es progenitor de Y y además X es mujer:

```
% madre(?X,?Y), cierto si X es madre de Y.
% versión ADECUADA
    madre(X,Y) :-
        progenitor(X,Y),
        mujer(X).
```

A diferencia de la anterior, esta definición siempre dará la información correcta independientemente del contenido concreto de los predicados "progenitor" y "mujer".

```
1  % madre(?X,?Y), cierto si X es madre de Y.
2
3  madre(X,Y) :-
4      progenitor(X,Y),
5      mujer(X).
```

Pruebe ahora su código realizando consultas significativas:

```
?- madre(pepa,X).
```

```
?- madre(pepe,X).
```

```
?- madre(X,Y).
```

`madre(?X), cierto si X es madre.`

Aunque este predicado tiene mismo nombre que el anterior, se trata de un predicado distinto (recuerde que Prolog permite sobrecargar los nombres de los predicados, distinguiendo entre ellos por su número de parámetros). Una persona es madre si es mujer y es progenitora de alguien, o, lo que es lo mismo, si es madre de alguien (esta segunda posible implementación se basaría en el predicado `madre/2` recién definido). Recuerde el uso de la *variable anónima* para casos como este, en el que no importa quién o quiénes son los hijos.

```
1  % madre(?X), cierto si X es madre.
2
3  %% primera versión, basada en progenitor y mujer
4  % madre(X) :-
5  %     progenitor(X,_),
6  %     mujer(X).
7
8  %% segunda versión, más corta, basada en madre/2
9  madre(X) :-
10     madre(X,_).  % X es madre de alguien
```

Haga a continuación consultas significativas para comprobar que su código funciona correctamente.

```
?- madre(pepa).
```

```
?- madre(pepe).
```

```
?- madre(X).
```

`hija(?X,?Y), cierto si X es hija de Y.`

```
% hija(?X,?Y), cierto si X es una hija de Y.

hija(X,Y) :-
    progenitor(Y,X),
    mujer(X).
```

```
?- hija(X, pepa).
```

```
?- hija(X, pepe1).
```

```
?- hija(X, Y).
```

`abuelo(?X,?Y), cierto si X es abuelo (varón) de Y.`

```
% abuelo(?X,?Y), cierto si X es abuelo (varón) de Y.
%
abuelo(X,Y) :-
    progenitor(X,Z),
    progenitor(Z,Y),
    varon(X).
```

```
?- abuelo(pepa,pepa11).
```

```
?- abuelo(pepe,pepa11).
```

```
?- abuelo(X, Y).
```

`hermana(?X,?Y), cierto si X es hermana de Y.`

Al implementar este predicado considere que para que dos personas sean hermanas basta con que tengan un progenitor en común. Tenga en cuenta además que nadie es hermana de sí misma, es decir, una consulta del estilo "?- hermana(pepa,pepa)" debe devolver falso. El comportamiento anterior se puede conseguir usando adecuadamente el predicado predefinido "X \= Y", cierto si "X" no unifica con "Y".

```
% hermana(?X,?Y), cierto si X es hermana de Y.
%
hermana(X,Y) :-
    progenitor(Z,X),
    progenitor(Z,Y),
    mujer(X),
    X \= Y.
```

```
?- hermana(X,pepa).
```

```
?- hermana(X, pepa11).
```

```
?- hermana(Hermana, DeQuien).
```

`tia(?X,?Y), cierto si X es tía de Y.`

Para implementar este predicado puede usar el predicado `hermana` definido más arriba.

```
% tia(?X,?Y), cierto si X es tía de Y.
%
tia(X,Y) :-
    progenitor(Z,Y),
    hermana(X,Z).
```

```
?- tia(X, pepa11).
```

```
?- tia(Tia, DeQuien).
```

`ancestro(?X,?Y), cierto si X es un ancestro de Y.`

X es ancestro de Y si está por encima en su línea genealógica, es decir, si es progenitor/abuelo/bisabuelo/tatarabuelo/... de Y. Este predicado se puede definir de forma recursiva como sigue:

- Caso base. El caso elemental en el que X es un ancestro de Y es cuando X es progenitor de Y (los padres son los ancestros más cercanos), es decir, para cualesquiera X, Y , si X es progenitor/a de Y entonces X es un ancestro de Y . En Prolog, lo anterior da lugar a una regla.
- Caso recursivo. Para que X sea un ancestro de Y distinto a su progenitor, X tiene que ser progenitor de alguien, pongamos de un cierto Z, y ese Z tiene que ser a su vez un ancestro (aquí está la recursión) de Y. En otras palabras: para cualesquiera personas X, Y, Z, si X es progenitor de Z y Z es un ancestro de Y , entonces X es un ancestro de Y. En Prolog lo anterior se traduce mediante una regla.

```
% ancestro(?X,?Y), cierto si X es un ancestro (mujer o varón) de Y.
%
ancestro(X,Y) :-
    progenitor(X,Y).
ancestro(X,Y) :-
    progenitor(X,Z),
    ancestro(Z,Y).
```

```
?- ancestro(pepa,X).
```

```
?- ancestro(X,Y).
```

`pariente(?X,?Y), cierto si X es pariente de Y.`

Para poder implementar este predicado es necesario tener claro qué se entiende por ``X es pariente de Y''. Una posibilidad sería la siguiente: X es pariente de Y si se da cualquiera de las tres siguientes condiciones:

1. X es ancestro de Y (este es el caso en el que X es padre/abuelo/etc, de Y), o bien
2. Y es ancestro de X (este es el caso en el que X es hijo/nieto/etc de Y), o bien
3. X e Y tienen un ancestro común (este último sería el caso en el que X e Y son hermanos, o primos, etc).

Esta definición, expresada en Prolog, da lugar a tres reglas definidas mediante el predicado `ancestro` implementado más arriba. Tenga en cuenta además que, al igual que en la definición del predicado "hermana", nadie debería ser pariente de sí mismo.

```
% pariente(?X,?Y), cierto si X es un pariente (mujer o varón) de Y.
%
pariente(X,Y) :-
    ancestro(X,Y).
pariente(X,Y) :-
    ancestro(Y,X).
pariente(X,Y) :- %tienen un ancestro común
    ancestro(Z, X),
    ancestro(Z, Y),
        X\=Y.
```

3. El predicado de unificación

Las siguientes consultas utilizan el predicado de unificación de Prolog, `"="` , y el de no unificación, `"\="` , para saber si ciertos pares de expresiones son o no unificables. Para cada una de estas consultas, aplique sobre papel el **Algoritmo de Unificación** estudiado en clase para averiguar cuál será la respuesta dada por Prolog, que puede ser *false* (si la consulta no es cierta), *true* (si la consulta se refiere al predicado de no unificación `"\="` y es cierta), o, en otro caso, el (o uno de ellos si hubiese varios) unificador de máxima generalidad que hace cierta la unificación. A continuación compruebe sus respuestas ejecutando las consultas mediante el intérprete de Prolog (flecha azul de la derecha).

```
?- f(X, g(b,c)) = f(Z, g(Y,c)).
```

Recuerde que cuando dos expresiones son unificables, puede haber varios posibles unificadores de máxima generalidad (umg's) y y cualquiera de ellos es válido: en el ejemplo anterior, tanto `X=Z, Y=b` como `Z=X, Y=b` son umg's, aunque Prolog solo facilite uno de ellos.

```
?- f(X, g(b,c)) = f(Z, g(_Y,c)).
```

La única diferencia con la consulta anterior es que la variable `Y` se ha sustituido por la variable semianónima `_Y` , por lo que ahora no se reporta su valor.

```
?- f(X, g(b,c)) \= f(Z, g(Y,c)).
```

La respuesta a la consulta anterior, "¿Es cierto que estas dos expresiones NO son unificables?", es `false` , puesto que SÍ son unificables (ver consulta previa).

```
?- f(X, g(b,X)) = f(c, g(Y,d)).
```

La respuesta es `false` : en efecto, para unificar el primer argumento es necesario hacer la sustitución `X=c` , y esta sustitución se hace en TODAS las apariciones de la variable `X` , por lo que al llegar al segundo argumento de `g` se tiene `c` en la expresión de la izquierda y `d` en la de la derecha, dos constantes distintas que no pueden unificarse.

```
?- f(X, g(b,X)) \= f(c, g(Y,d)).
```

La respuesta ahora es `true` puesto que, como se acaba de ver en la anterior, las expresiones NO son unificables.

```
?- f(X, g(b,c)) = f(c, g(X,c)).
```

La respuesta es `false`: en efecto, para unificar el primer argumento es necesario hacer la sustitución `X=c`, y esta sustitución se hace en TODAS las apariciones de la variable `X` de LAS DOS EXPRESIONES a unificar, por lo que al llegar al primer argumento de `g` se tiene `b` en la expresión de la izquierda y `c` en la de la derecha, dos constantes distintas que no pueden unificarse.

```
?- f(_X, g(b,c)) = f(c, g(_X,c)).
```

La respuesta sigue siendo false, puesto que las variables semianónimas también tienen que unificar.

```
?- f(_, g(b,c)) = f(c, g(_,c)).
```

La respuesta sin embargo ahora es true, puesto que ocurrencias distintas de la variable anónima NO tienen que unificar.

```
?- p(X, f(Y), g(X)) = p(Z, f(g(a)), g(a)).
```

Aunque SWI Prolog no siempre hace explícita la composición de sustituciones (observe que la respuesta a la consulta anterior es `X=Z, Z=a` en lugar de `X=a, Z=a` como daría el algoritmo de unificación estudiado en clase), internamente sí que está haciendo la composición, es decir, se tiene `X=a`, como demuestra la consulta siguiente.

```
?- p(X, f(Y), g(X)) = p(Z, f(g(a)), g(a)), X=d.
```

En efecto, cuando Prolog llega a `X=d` lo que realmente tiene es `a=d`, y de ahí la respuesta negativa obtenida.

```
?- f(s(s(s(0))),Y,Z) = f(s(X),0,s(X)*Y).
```

El ejemplo anterior ilustra de nuevo cómo la sustitución de una variable repercute en TODAS sus apariciones: cuando se llega al último argumento, `s(X)*Y` es en realidad `s(s(s(0)))*0` por las sustituciones hechas para solventar las discordancias previas, y de ahí que la última sustitución sea `Z = s(s(s(0)))*0`.

```
?- p(X,f(Y)) = p(Y,X).
```

La respuesta positiva obtenida en el último ejemplo difiere de la obtenida aplicando el algoritmo de unificación estudiado. Esto es debido a que el algoritmo de unificación utilizado por SWI Prolog, por razones de eficiencia, *no realiza el test de ocurrencia*. Lo anterior no es grave porque rara vez se presentan situaciones en las que este test sea necesario, aunque si lo fuese SWI Prolog ofrece un predicado de unificación alternativo, `unify_with_occurs_check/2`, que sí que incorpora el test de ocurrencia. Puede comprobarlo con la siguiente consulta, igual a la anterior pero realizando, ahora sí, el test de ocurrencia:

```
?- unify_with_occurs_check(p(X,f(Y)), p(Y,X)).
```

El predicado suma/3 y el test de ocurrencia

La primera cláusula del predicado `suma/3` discutido en el primer apartado de esta práctica es un buen ejemplo de los errores que se pueden producir debido a que la implementación del predicado de unificación `=` **no** contempla el test de ocurrencia. En efecto, considere la siguiente consulta:

```
?- suma(Y, 0, s(Y)).
```

La consulta anterior pregunta por la existencia de algún número natural tal que sumado con 0 sea igual a su sucesor, por lo que la respuesta de Prolog debería ser, indudablemente, `false`. Sin embargo, si ejecuta la consulta, verá que Prolog contesta afirmativamente, haciendo la unificación `Y = s(Y)`. Este error se debe a que Prolog está obviando el test de ocurrencia: al unificar la consulta, `suma(Y,0,s(Y))`, con la primera cláusula del programa, `suma(X,0,X)`, sustituye primero `X` por `Y`, de modo que al llegar al tercer argumento tiene que ver si puede arreglar la discordancia entre `Y` (porque X se ha sustituido por Y) y `s(Y)`. El algoritmo estudiado en clase daría, debido al test de ocurrencia, `false`, pero el algoritmo `=`, como se ha comentado previamente, no incluye este test.

Consultas como la anterior son poco frecuentes, por lo que, gracias a la ganancia en eficiencia, es bastante habitual encontrar código como el facilitado más arriba, aún sabiendo que no siempre funcionará correctamente. Una implementación alternativa del predicado `suma/3`, esta vez correcta para todo tipo de consultas, sería la siguiente:

```
% suma(?X, ?Y, ?Z)
% cierto si Z es la suma de X e Y.
% programa lógico puro (no utiliza la aritmética de Prolog)
% IMPLEMENTACIÓN CON TEST DE OCURRENCIA

suma_oc(X,0,Z) :-
    unify_with_occurs_check(X, Z).

suma_oc(X,s(Y),s(Z)) :-
    suma_oc(X,Y,Z).
```

Compruebe cómo, con este nuevo código, el error anterior desaparece:

```
?- suma_oc(Y, 0, s(Y)).
```

4. El orden importa + uso del depurador

En los apuntes se ha explicado cómo tanto el orden de las cláusulas en un programa como el orden de los predicados en el cuerpo de una regla pueden influir en qué soluciones se encuentran, en qué orden aparecen y en la terminación o no de las consultas que se realizan. En este apartado se pretende que experimente con ello y con el uso del depurador de SWI-Prolog a través del ejemplo dado en los apuntes, que propone las siguientes cuatro posibles implementaciones para el predicado `ancestro(?X,?Y)`, cierto si X es ancestro de Y:

```
% progenitor_a(?X, ?Y): cierto si X es progenitor/a de Y
progenitor_a(pepa, pepito).
progenitor_a(pepito, pepon).

% ancestro(?X, ?Y): cierto si X es un ancestro de Y
% VERSIÓN 1: caso base en primer lugar, caso recursivo con
% recursión a la derecha (recursión final o recursión de cola)
ancestro1(X, Y) :-
    progenitor_a(X, Y).
ancestro1(X, Y) :-
    progenitor_a(X, Z),
    ancestro1(Z, Y).

```

```
% VERSIÓN 2: caso base en último lugar, caso recursivo con
% recursión a la derecha (recursión final o recursión de cola)
ancestro2(X, Y) :-
    progenitor_a(X, Z),
    ancestro2(Z, Y).
ancestro2(X, Y) :-
    progenitor_a(X, Y).
```

```
% VERSIÓN 3: caso base en primer lugar, caso recursivo con
% recursión a la izquierda
ancestro3(X, Y) :-
    progenitor_a(X, Y).
ancestro3(X, Y) :-
    ancestro3(Z, Y),
    progenitor_a(X, Z).

% VERSIÓN 4: caso base en último lugar, caso recursivo con
% recursión a la izquierda
ancestro4(X, Y) :-
    ancestro4(Z, Y),
    progenitor_a(X, Z).
ancestro4(X, Y) :-
    progenitor_a(X, Y).
```

Las cuatro versiones anteriores son equivalentes desde un punto de vista lógico, pero no se comportan igual en Prolog, como puede comprobar a continuación ejecutando la consulta "¿De quién(es) es ancestro pepa?" con cada una de las cuatro versiones.

```
?- ancestro1(pepa, D).
```

```
?- ancestro2(pepa, D).
```

```
?- ancestro3(pepa, D).
```

```
?- ancestro4(pepa, D).
```

Para algunas de las consultas anteriores, compruebe el porqué de las respuestas obtenidas construyendo sobre papel los árboles de Resolución correspondientes (puede cotejarlos con los dibujados en los apuntes) y utilice luego el depurador de errores para seguir paso a paso la construcción del árbol de Resolución por parte de Prolog. Para invocar al depurador en SWISH basta con escribir la palabra "trace, " (con una coma detrás) antes de la consulta y seguir la ejecución con los botones ofrecidos. Por ejemplo:

```
?- trace, ancestro1(pepa, D).
```

```
?- trace, ancestro3(pepa, D).
```

Recuerde por último que no existe una regla universal sobre cómo ordenar las cláusulas de un programa ni los predicados dentro del cuerpo de una regla, pero en general son útiles las dos siguientes recomendaciones:

1. Casos base primero.
2. Recursión a la derecha (cuando sea posible).

La única de las cuatro versiones anteriores que cumple ambas recomendaciones es la primera.

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº2: Aritmética

Soluciones propuestas, comentadas

Prerrequisitos
Para la realización de esta segunda práctica de Programación Lógica es necesario haber leído y estudiado los apartados 1-6 del tema PL-2, en los que se describen las características generales del lenguaje Prolog, su sintaxis y semántica, aritmética, entrada/salida y comparación/clasificación de términos.

1. Uso básico de operadores y predicados aritméticos

Las siguientes consultas contienen operadores y predicados aritméticos así como predicados de unificación y comparación de términos. Piense para cada una de ellas, sin usar el intérprete, cuál(es) sería(n) la(s) respuesta(s) ofrecidas por Prolog. A continuación compruebe sus respuestas ejecutando las consultas (flecha azul de la derecha).

Recuerde en particular el funcionamiento de los predicados aritméticos:

- `X is Y` es cierto si Y es una expresión aritmética evaluable y X, *que puede ser cualquier término y que se mantiene tal cual aparece*, unifica con el resultado de evaluar Y. Si Y no es una expresión aritmética o no puede evaluarla, Prolog produce un error. Si X es unificable con el resultado de evaluar Y, se realiza la unificación, y si no lo es, el predicado falla.
- `X =:= Y, X =\= Y, X > Y, X >= Y, X < Y, X =< Y` son ciertos si X e Y son expresiones aritméticas, se pueden evaluar (*ambas*), y los valores evaluados son iguales/no son iguales/cumplen la relación de orden expresada. Si X o Y no son expresiones aritméticas o no puede evaluarlas, Prolog produce un error. OJO a la forma poco habitual de escribir el menor o igual: `X =< Y` en lugar de `X <= Y` (para evitar la confusión con la implicación lógica).

y las diferencias de los predicados anteriores con los de unificación y comparación:

- `X = Y ( X \= Y )` es cierto si X e Y son términos que son (no son) unificables
- `X == Y ( X \== Y )` es cierto si X e Y son términos que son (no son) literalmente idénticos.

```
?- X is 2*3-4, Y is 2^X.
```

Observe cómo el predicado `is` UNIFICA el argumento de la izquierda con el resultado de evaluar el argumento de la derecha, por lo que el primer objetivo de la consulta anterior unifica X con 2 y el segundo objetivo pasa a ser `Y is 2^2` , produciendo la unificación `Y=4` .

```
?- X is X+1.
```

La consulta anterior produce un error de instanciación: Prolog NO puede evaluar X+1 puesto que X es una variable, sin valor asignado.

```
?- X is a*3^2.
```

La consulta anterior produce un error aritmético: `a*3^2` *NO es una expresión aritmética (debido a la constante no numérica* `a` *).*

```
?- X = 2, X is X+1.
```

La respuesta a la consulta anterior es `false` *puesto que la unificación* `X = 2` *hace que el segundo objetivo se convierta en* `2 is 2+1` *, que falla al intentar unificar* `2` *con el resultado de la expresión de la derecha,* `3` *.*

```
?- X is Y+1.
```

La consulta anterior produce un error de instanciación: Prolog NO puede evaluar Y+1 puesto que Y es una variable sin valor asignado.

```
?- 4 + 2.
```

Error: la expresión `4+2` *no es más que un término compuesto, para evaluarlo debe formar parte de un predicado aritmético.*

```
?- 4 mod 2 =:= X.
```

Produce un error de instanciación: Prolog no puede evaluar X.

```
?- X is 4 mod 2, 6 mod 2 =:= X.
```

El predicado "is" produce la unificación `X=0` *, por lo que el siguiente objetivo es 6 mod 2 =:= 0, cierto.*

```
?- Z = 6/2, X is Z + 1, X >= 2+1.
```

Note la diferencia en el resultado: `Z=6/2` *es una unificación (asocia a Z el término compuesto 6/2, pero no evalúa), mientras que* `X=4` *es el resultado de evaluar la expresión aritmética 6/2 + 1.*

```
?- Z=2*3, 3+4 =:= Z+1.
```

Observe de nuevo cómo `Z=2*3` *simplemente unifica la varialbe Z con el término compuesto 2*3, pero no lo evalúa, mientras que el operador aritmético siguiente,* `=:=` *, sí que evalúa sus argumentos.*

```
?- C=1, T=2*3, L is [C,T].
```

*Error: a la derecha de un "is" solo puede haber expresiones aritméticas, y [1,2*3] es una lista.*

```
?- C=1, T=2*3, L is C*T.
```

Unifica C con 1, T con 2*3 (sin evaluar su valor) y L con 6 (el predicado `is` sí evalúa 1*2*3, unificando el resultado con L).

```
?- 3 =\= 3*a.
```

Error: `3*a` *NO es una expresión aritmética.*

```
?- 3 =\= 3*X, X = 5.
```

Error: `3*X` *es una expresión aritmética pero no es evaluable porque la variable X no se ha unificado previamente con ningún valor.*

```
?- 3 \= 3*a.
```

*Cierto: efectivamente la constante 3 NO es unificable con el término compuesto 3*a.*

```
?- 3 \== 3*a.
```

*Cierto: efectivamente la constante 3 NO es idéntica al término compuesto 3*a.*

```
?- 3 = 1+2.
```

La respuesta es `false` *porque efectivamente NO es verdad que la constante 3 sea unificable con el término compuesto 1+2.*

```
?- 3 == 1+2, X = 3.
```

La respuesta es `false` *porque falla el primer objetivo: NO es verdad que la constante 3 sea idéntica al término compuesto 1+2.*

```
?- 3 =:= 1+2.
```

La respuesta ahora es cierta porque el predicado aritmético =:= evalúa las expresiones a ambos lados y compara los resultados.

```
?- X = 3+5, X =:= 8.
```

El primer objetivo unifica X con el término compuesto 3+5 y el segundo es cierto puesto que el predicado aritmético =:= evalúa las expresiones a ambos lados y compara los resultados.

```
?- X = 3+5, X == 8.
```

La respuesta es `false` *porque ni* `=` *ni* `==` *son operadores aritméticos, por lo que no evalúan: el primer objetivo unifica X con el término compuesto 3+5, y el segundo falla al comprobar que dicho término no es idéntico a 8.*

```
?- X = 3+5, X is 8.
```

Falso puesto que el predicado `is` *evalúa la parte derecha pero NO evalúa la izquierda, por lo que intenta unificar 3+5 con 8, expresiones no unificables.*

```
?- X is 8, X is 3+5.
```

Ahora sí: el primer "is" unifica X con 8, y el segundo comprueba que ese 8 es unificable con el resultado de evaluar 3+5, que da 8.

```
?- X is 3+5, X == 8.
```

"is" evalúa 3+5 y unifica X con el resultado, 8, y el predicado == da cierto puesto que 8 y 8 son idénticos.

```
?- X is 3+5, X=:= 8.
```

De nuevo "is" evalúa 3+5 y unifica X con el resultado, 8, y el predicado numérico =:= da cierto puesto que 8 y 8 son iguales.

2. Cálculo de factoriales

En este apartado se utiliza el predicado para el cálculo de factoriales `"factorial/2"` discutido en los apuntes para destacar algunos detalles de su implementación que son aplicables a muchos otros predicados aritméticos en Prolog:

```
1  % factorial(+X, ?Y)
2  % cierto si Y es el factorial del número natural X.
3
4  factorial(0, 1).
5
6  factorial(X, Y) :-
7      integer(X),
8      X > 0,
9      Z is X - 1,
10     factorial(Z, FZ),
11     Y is X*FZ.
```

2.1. El primer parámetro solo puede ser de entrada

Como se comenta en los apuntes, la aritmética de Prolog, comparada con el uso de la lógica "pura", tiene muchas ventajas (comodidad, eficiencia), pero también algún inconveniente: algunos parámetros pierden la versatilidad de poder usarse tanto de entrada como de salida.

Antes de ejecutar las siguientes consultas, piense qué respuesta(s) se obtendrían y por qué:

```
?- factorial(2, 2).
```

```
?- factorial(2, X).
```

```
?- factorial(X, 2).
```

Como habrá observado, `factorial/2` sabe comprobar y calcular factoriales (primera y segunda consulta) pero sin embargo no es capaz de averiguar si existe algún X cuyo factorial es igual a 2 (tercera consulta). La respuesta negativa en este último caso se debe a la línea `integer(X)`, que evita el error de instanciación que produciría la línea siguiente, `X > 0`, al recibir algo que no puede evaluar. El primer argumento del predicado tiene que ser necesariamente de entrada (como se indica con el "+X" del comentario previo al código), y el predicado es por lo tanto válido para calcular factoriales, pero ya no sirve para averiguar si un número es el factorial de otro.

Lo anterior ocurre en general con todos los predicados implementados usando la aritmética de Prolog: son mucho más cómodos y eficientes que los predicados basados en lógica "pura", pero son menos versátiles porque algunos parámetros (aquellos que aparecen como argumentos de predicados de comparación aritméticos o en la parte derecha de un `is`) ya solo se pueden usar como parámetros de entrada.

2.2. Uso de `integer(X)` para evitar errores en tiempo de ejecución

A continuación se demuestra la utilidad de la línea `integer(X)` para evitar errores en tiempo de ejecución debidos a la introducción de tipos de datos no esperados.

Antes de ejecutar la siguiente consulta, piense qué respuesta(s) se obtendrían y por qué:

```
?- factorial(a, F).
```

Efectivamente, la respuesta correcta ante la pregunta "¿Existe algún número que sea el factorial de a?" debe ser negativa, puesto que "a" no es un número natural y por lo tanto no tiene factorial. Esta respuesta negativa la produce el predicado de clasificación `integer(X)` .

¿Qué respuesta cree que se obtendría ante la misma pregunta si se suprimiese la línea `integer(X)` del código anterior? ¿Por qué? Compruebe su respuesta comentando dicha línea y ejecutando de nuevo la consulta dada.

La comprobación `integer(X)` previene el error aritmético que produce la línea `X > 0` si el valor de X que le llega no es un número.

Suponga ahora que la comprobación `integer(X)` se mantiene pero se sitúa detrás de `X > 0` . ¿Cree que este cambio tendría alguna importancia? ¿Por qué? Compruebe su respuesta intercambiando el orden de las dos líneas en el código anterior y ejecutando de nuevo la consulta dada.

No tiene sentido colocar `integer(X)` detrás de `X > 0` porque no serviría para impedir errores aritméticos en caso de que X no sea un número.

2.3. Uso de `X > 0` para evitar computaciones infinitas

En lo que sigue se muestra cómo la comprobación `X > 0` no solo sirve para asegurar que no se intenta computar el factorial de números negativos, sino también para evitar computaciones infinitas si se solicitan más soluciones una vez obtenida la única solución válida.

Antes de ejecutar las siguientes consultas, piense en los árboles de Resolución correspondientes para saber qué respuesta(s) se obtendría(n) (todas las posibles respuestas) y por qué:

```
?- factorial(-3, F).
```

```
?- factorial(2, F).
```

¿Qué respuesta(s) cree que se obtendrían ante las mismas preguntas si se suprimiese la línea `X > 0` del código anterior? ¿Por qué? Compruebe su respuesta comentando dicha línea y ejecutando de nuevo las dos consultas anteriores.

La comprobación `X > 0` sirve para indicar a Prolog que la regla que la contiene SOLO debe usarse para valores de `X` estrictamente mayores que 0. Si se suprime:

- *la consulta `factorial(-3, F)` llamaría recursivamente a `factorial(-4, FZ)` que a su vez intentaría calcular el factorial de -5 y así sucesivamente hasta producir un desbordamiento de la pila de llamadas recursivas.*
- *la consulta `factorial(2, F)` daría una primera respuesta correcta, `F=2` , al terminar el cálculo con la primera cláusula del programa, pero si se pide a Prolog que busque más soluciones el resultado sería*

otra vez un desbordamiento de pila porque intentaría calcular el factorial de 0 con la segunda regla, llamando recursivamente al factorial de -1, de -2, etc.

3. Cálculo de densidades

En este apartado se practica el uso de la aritmética de Prolog con un ejemplo sencillo. Considere el siguiente programa:

```
% num_habitantes(?X, ?Y)
% cierto si Y es el número de habitantes (en millones) de Y.
num_habitantes('India', 1324).
num_habitantes('China', 1403).
num_habitantes('Brasil', 210).
num_habitantes('España', 47).

% area(?X, ?Y)
% cierto si Y es el área (en millones de km. cuadrados) de X.
area('India', 3.288).
area('China', 9.597).
area('Brasil', 8.512).
area('España', 0.505).
```

Dados los predicados anteriores, implemente el siguiente predicado:

```
% densidad(?X, ?D)
% cierto si D es la densidad de población del país X
% (habitantes por kilómetro cuadrado)

 %  ESCRIBA SU CÓDIGO A CONTINUACIÓN
 %
 densidad(X, D) :-
    num_habitantes(X, Num),
    area(X, A),
    D is Num/A.
```

y pruébelo escribiendo en Prolog y ejecutando, al menos, las siguientes consultas:

Consulta 1: ¿Qué países existen con una densidad mayor que 100?

```
?- densidad(P, _D), _D > 100.
```

Consulta 2: ¿Existe algún país con menos de 50 millones de habitantes, área menor que 1 millón de km cuadrados y densidad menor que 100?

```
?- num_habitantes(_P,_N), _N < 50, area(_P, _A), _A < 1,
   densidad(_P,_D), _D < 100.
```

Consulta 3: ¿Qué países existen con más de 5 millones de kilómetros cuadrados de extensión y con una densidad menor o igual que 100?

```
?- area(X, _A), _A > 5, densidad(X, _D), _D =< 100.
```

Observe el uso de variables semianónimas (las que empiezan por subrayado seguido de mayúscula) en las consultas anteriores: recuerde que las distintas apariciones de una variable semianónima tienen que unificar entre sí pero Prolog no reporta valor para ellas.

4. Implementación de algunos predicados aritméticos

Utilizando los operadores y predicados aritméticos predefinidos de Prolog, escriba y pruebe los siguientes programas para números naturales. Las implementaciones se harán de forma que no se produzca ningún error en tiempo de ejecución (por ejemplo errores de instanciación), para lo cual será necesario comprobar que los argumentos recibidos son efectivamente números naturales.

Cuando su implementación presente **recursión lineal** (una única llamada recursiva) pero sea **recursión no final** (la llamada recursiva no es lo último que se ejecuta), *implemente una segunda versión con recursión final* mediante el uso de parámetros de acumulación, de forma similar a como se hace en los apuntes con el predicado factorial.

Añada todas las consultas de prueba que considere oportunas.

```
1  % natural(+X)
2  % cierto si X es un número natural
3  % (puede usar el predicado predefinido integer(+X))
4
5  natural(X) :-
6       integer(X),
7       X >= 0.
```

```
?- natural(a).
```

```
?- natural(X).
```

```
?- natural(-1.2).
```

```
?- natural(0).
```

```
?- natural(30).
```

```
1  % fib(+N, ?F)
2  % cierto si F es el número de Fibonacci asociado con N.
3  % Recuerde que cada número natural tiene asociado otro número
4  % natural, denominado número de Fibonacci, que se calcula como
5  % sigue: f(0)= f(1)=1 ; f(n) = f(n-1)+f(n-2) si n>1.
6  %
7  % VERSIÓN 1 (INEFICIENTE)
8
9  fib(0, 1).
10 fib(1, 1).
11 fib(N, F) :-
12     integer(N),
13     N > 1,
14     N_1 is N-1,
```

```
    N_2 is N-2,
    fib(N_1, FN_1),
    fib(N_2, FN_2),
    F is FN_1 + FN_2.
```

```
% VERSIÓN 2 (EFICIENTE, PROPUESTA POR SWI-PROLOG)
%
% fib_acc(+N, ?F)
% versión con parámetros de acumulación empezando desde abajo
% y guardando los dos últimos valores calculados.
% implementación propuesta en SWI-Prolog

fib_acc(0, 1).
fib_acc(1, 1).
fib_acc(N, F) :-
        integer(N),
        N > 1,
        fib_acc(1,1,1,N,F).

% Los dos primeros parámetros de fib_acc sirven para almacenar
% los dos últimos números de Fibonacci. El tercer parámetro es
% un índice que se va incrementando hasta llegar a N.

fib_acc(_, F1, N, N, F1).
fib_acc(F0, F1, I, N, F) :-
        I < N,
        F2 is F0 + F1,
        I2 is I + 1,
        fib_acc(F1, F2, I2, N, F).
```

Observe a continuación cómo la primera implementación de Fibonacci no es capaz de calcular el número de Fibonacci de 50 (se produce un desbordamiento de pila) mientras que la segunda sí que lo logra.

```
?- fib(50,X).
```

```
?- fib_acc(50,X).
```

```
% mcd(+M, +N, ?MCD)
% cierto si MCD es el máximo común divisor de M y N.
% Recuerde que una forma sencilla de calcular el mcd
% de dos números naturales es utilizar el algoritmo de Euclides:
% - El mcd de M y 0 es M.
% - El mcd de M y N es igual al mcd de N y de (M mod N)
% (resto de la división entera).
%
mcd(M, 0, M) :-
     natural(M).
mcd(M, N, MCD) :-
     natural(M), natural(N), N>0,
     Resto is M mod N,
     mcd(N, Resto, MCD).
```

?- Your query goes here ...

?- Your query goes here ...

```
% exp(+M, +N, ?E)
% cierto si E es igual a M elevado a N
% Recuerde que el valor x^y puede definirse de forma recursiva
% como sigue:
% - Para todo x, x^0 = 1.
% - Para todo x y para todo y>0, x^y = x * x^{y-1}
%
exp(_, 0, 1).
exp(M, N, E) :-
     natural(M), natural(N), N>0,
     N1 is N-1,
     exp(M, N1, E1),
     E is M*E1.

% con recursión final
exp_acc(M, N, E) :-
    natural(M),
    natural(N),
    exp_acc(M, N, 1, E).

 exp_acc(_, 0, A, A).

 exp_acc(M, N, A, E) :-
    N > 0,
    N1 is N - 1,
    A1 is M*A,
    exp_acc(M, N1, A1, E).
```

?- Your query goes here ...

?- Your query goes here ...

```
% num_t(+N,?T)
% cierto si T es el número triangular asociado con N,
% que se define como la suma de todos los números naturales
% menores o iguales a N.
%
% num_t(+N, ?T)
% cierto si T es el número triangular asociado con N

num_t(1,1).
num_t(N, T) :-
     natural(N), N>1,
     N1 is N-1,
     num_t(N1, T1),
     T is T1+N.
```

```
15
16 % con recursión final
17
18 num_t_acc(N, T) :-
19     natural(N),
20     num_t_acc(N, 1, T).
21
22 num_t_acc(1, A, A).
23 num_t_acc(N, A, T) :-
24     N > 1,
25     A1 is A+N,
26     N1 is N-1,
27     num_t_acc(N1, A1, T).
```

```
?- Your query goes here ...
```

```
?- Your query goes here ...
```

5. Evaluación de expresiones aritméticas

Aunque los predicados aritméticos de Prolog permiten evaluar expresiones aritméticas, es interesante entender cómo podría definirse un predicado para evaluar este tipo de expresiones, es decir, cómo definir el predicado `eval(+E, ?V)` , cierto si E es una expresión aritmética y V es el valor resultante después de evaluar E. Por ejemplo, la consulta "?- eval(5, X)." debe devolver X=5, mientras que "?- eval(2+(3-12), X)." debe devolver X= -7.

Algunas de las reglas del predicado `eval` vendrían dadas por lo siguiente:

- El valor asociado con una expresión aritmética elemental (un número) es el propio número.
- Para evaluar una expresión de la forma A+B, donde A y B son expresiones aritméticas cualesquiera, hay que evaluar (recursivamente) tanto A como B y sumar los valores obtenidos.

Las reglas con las que se escribiría en Prolog el conocimiento anterior están incorporadas al programa que se facilita a continuación. Se pide:

- Complete el programa de forma que sea posible evaluar expresiones aritméticas tales que, además de la suma, admitan la resta, el producto y la división.
- Complete el programa de forma que las expresiones puedan además incluir operandos del tipo "fact(N)", siendo N un número natural, cuyo valor sea el factorial de N. Para ello, habrá que definir una regla con cabeza "eval(fact(N), V)", cierta si V es el factorial del número N, en cuyo cuerpo se usará el predicado `factorial/2` definido más arriba.
- Compruebe que todo lo implementado es correcto evaluando algunas expresiones aritméticas significativas. Por ejemplo, haga la consulta ?- eval(3*2+fact(4), V) y construya sobre papel el árbol de Resolución correspondiente.

```
1 % eval(+E, ?V)
2 % cierto si E es una expresión aritmética y V es su valor
3
4 eval(N, N) :-
5     number(N).
6
7 eval(A+B, V) :-
8     eval(A, VA), eval(B, VB), V is VA + VB.
```

```
 9
10 %% ESCRIBA A CONTINUACIÓN
11
12 eval(A-B, V) :-
13     eval(A, VA), eval(B, VB), V is VA - VB.
14
15 eval(A*B, V) :-
16     eval(A, VA), eval(B, VB), V is VA * VB.
17
18 eval(A/B, V) :-
19     eval(A, VA), eval(B, VB), V is VA / VB.
20
21 eval(fact(N), V) :-
22     eval(N, VN),
23     integer(VN),
24     VN >= 0,
25     factorial(VN, V).
```

```
?- eval(3*2+fact(4), V).
```

```
?- Your query goes here ...
```

```
?- Your query goes here ...
```

6. Algo de entrada/salida

Utilizando las facilidades de Prolog para realizar operaciones de entrada/salida, implemente y pruebe los dos siguientes predicados:

```
 1 % cubo
 2 % predicado sin argumentos que solicita un número y escribe
 3 % el cubo de dicho número, realizando el proceso anterior de
 4 % forma reiterada hasta que el usuario introduce la palabra fin.
 5
 6 cubo :-
 7     write('Introduzca un número (fin para terminar): '),
 8     nl,
 9     read(X),
10     procesa_cubo(X).
11
12 procesa_cubo(X) :-
13     X \= fin,
14     number(X),
15     X3 is X^3,
16     write('El cubo de '),
17     write(X),
18     write(' es '),
19     write(X3),
20     nl,
21     cubo.
```

```
?- cubo.
```

```
% prueba(+N, +T)
% Si N es un entero positivo y T es un término cualquiera,
% el predicado prueba escribe N líneas conteniendo cada una
% de ellas el término T.

prueba(N, T) :-
    integer(N),
    N >= 1,
    escribe(N, T).

escribe(1, T) :-
    write(T).
escribe(N, T) :-
    N > 1,
    write(T),
    nl,
    N1 is N-1,
    escribe(N1, T).
```

```
?- prueba(5, hola).
```

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº3: Listas y predicado de corte

Soluciones propuestas, comentadas

Autora: A. Pradera

Prerrequisitos
Para la realización de esta tercera práctica de Programación Lógica es necesario haber leído y estudiado el tema PL2 completo, especialmente los dos últimos apartados, dedicados al manejo de listas y al predicado de corte.

1. Manejo básico de listas en Prolog

Recuerde que en Prolog:

- La lista vacía se representa mediante `[]`.
- Toda lista no vacía se puede escribir de la forma `[a1,...,an]` y es unificable con un patrón del tipo `[C | R]` siempre y cuando `C` unifique con la cabeza de la lista, `a1`, y R unifique con el resto de la lista, `[a2,...,an]`.
- Toda lista que tenga al menos dos elementos es unificable con un patrón del tipo `[C1, C2 | R]` siempre y cuando `C1` unifique con la cabeza de la lista, `C2` unifique con su segundo elemento y R unifique con el resto de la lista.
- De forma similar, `[C1, C2, C3 | R]` podría unificar con listas de al menos tres elementos, etc.

Note en particular que a la derecha de la barra vertical `|` solo puede haber una lista o algo unificable con una lista, por ejemplo la lista `[1,2]` se puede escribir como `[1,2 | []]`, como `[1 | [2]]` o como `[1 | [2| []]]`, pero *no* como `[1 | 2]`. Esto último *no* es una lista. Tampoco una variable es una lista, por lo que una forma de implementar el predicado `es_lista(+X)`, cierto si X es una lista, sería la siguiente (SWI Prolog ofrece este predicado bajo el nombre de `is_list(+X)`):

```
% es_lista(+X)
% cierto si X es una lista

% X no puede ser una variable y tiene que unificar
% con la lista vacía [] o con el patrón [C|R] siendo
% R a su vez una lista:

es_lista(X) :- nonvar(X), X = [].
es_lista(X) :- nonvar(X), X = [_|R], es_lista(R).

% Cuando sepa cómo manejar el predicado de corte, !, observe que
% una implementación equivalente a la anterior sería la siguiente:

% si X es una variable, se impide el uso de las
% siguientes cláusulas (mediante el predicado de corte) y se
% devuelve falso (mediante el predicado fail, que siempre es falso):
es_list(X) :-
    var(X),
    !,
```

```
20      fail.
21
22  % si X no es una variable, tiene que unificar
23  % con la lista vacía [] o con el patrón [C|R] siendo
24  % R a su vez una lista:
25  es_list([]).
26  es_list([_|R]) :-
27      es_list(R).
```

Pruebe el predicado anterior con algunas consultas como las siguientes, pensando en cómo sería el árbol de Resolución correspondiente, y, en consecuencia, cuál(es) será(n) la(s) respuesta(s) obtenidas.

```
?- es_lista(X).
```

```
?- es_lista(a).
```

Las dos consultas anteriores devuelven false: en efecto, ni las variables ni las constantes son listas. En los árboles de Resolución con `es_lista` serían aplicables las dos cláusulas, pero ambas dan lugar a nodos fallo (en el caso de `es_lista(X)` fallan en `nonvar` y en el caso de `es_lista(a)` falla las unificaciones). En los árboles de Resolución con `es_list`, para la primera consulta serían aplicables las tres cláusulas del programa, pero el corte de la primera poda las otras dos ramas. La segunda consulta solo unifica con la primera cláusula del programa (los argumentos de las otras dos son listas, no unificables con la constante "a") y falla nada más empezar con `var(a)`.

```
?- es_lista([a,b]).
```

```
?- es_lista([a|b]).
```

```
?- es_lista([a|R]).
```

Las dos consultas anteriores devuelven false: en efecto, lo que está a la derecha de "|" tiene que ser una lista, y ni la constante "b" ni la variable "R" lo son.

```
?- R = [b], es_lista([a|R]).
```

En este caso la respuesta es afirmativa, porque al unificar previamente la variable `R` con la lista `[b]`, la expresión `[a|R]` se convierte en `[a|[b]]`, que sí es una lista (la lista `[a,b]`).

```
?- R = b, es_lista([a|R]).
```

La respuesta ahora es sin embargo falsa, puesto que lo que se construye es `[a|b]`, que no es una lista puesto que `b` no lo es.

Para cada una de las siguientes consultas, piense cuál(es) sería(n) la(s) respuesta(s) ofrecidas por Prolog y a continuación compruebe sus respuestas ejecutándolas.

Para comprender las unificaciones que se plantean a continuación es fundamental recordar que el patrón `[C1, C2, .. | R]` solo unfica con listas no vacías tales que `C1` unifica con el primer elemento de la lista, `C2` unifica con el segundo elemento de la lista, etc, y por último `R` unifica con el resto de la lista (que siempre será, a su vez, una lista).

```
?- [] = _.
```

La variable anónima, `_` , unifica con cualquier cosa, luego también, en particular, con la lista vacía.

```
?- [] = [_].
```

La lista vacía, `[]` , NO unifica con `[_]` , ya que este último patrón solo unifica con listas conteniendo exactamente un elemento (que puede ser cualquiera).

```
?- [a] = [A | []].
```

Cierto con `A=a` puesto que la variable `A` unifica claramente con `a` , y la lista vacía unifica directamente con el resto de la lista `[a]` .

```
?- [a,b] = [a | b].
```

Falso puesto que el `b` de `[a | b]` NO unifica con el resto de la lista `[a,b]` , que es `[b]` .

```
?- [a,b] = [A | [b | []]].
```

Cierto con `A=a` , puesto que `[b | []]` unifica con el resto de la lista `[a,b]` , que es `[b]` .

```
?- [X,1] = [1,1,A | R].
```

Falso puesto que la lista de la izquierda tiene solo 2 elementos y la de la derecha tiene al menos 3.

```
?- [X,1,Y] = [1,_,b | Z].
```

La lista de la izquierda tiene exactamente 3 elementos, que unifican sin problema con los tres primeros elementos de la lista de la derecha, y Z tiene que ser la lista vacía, puesto que ese es el resto de la lista de la izquierda sin contar sus 3 primeros elementos.

```
?- [X,1,X] = [1,_,b | Z].
```

La respuesta ahora es `false` : ¡X no puede unificar a la vez con 1 y con b!

```
?- [a,b,[c,d]] = [_, X | Y].
```

Ojo, observe que la unificación que produce la consulta anterior es `Y = [[c,d]]` y NO `Y = [c,d]` : lo que hay a la derecha de `|` debe unificar con EL RESTO de la lista de la `[a,b,[c,d]]` una vez eliminados sus dos primeros elementos, y ese resto es LA LISTA que contiene al único elemento que queda (`[c,d]`), es decir, `[[c,d]]` .

```
?- [a|R] = [a,b,[c],lista].
```

Lo anterior es cierto siempre y cuando se unifique la variable R con el resto de la lista de la derecha, que es la lista compuesta por todos sus elementos salvo el primero.

```
?- [a, [b]] = [A | [b]].
```

Falso puesto que lo que hay a la derecha de `|`, `[b]`, NO unifica con el resto de la lista de la izquierda, que es `[[b]]`.

```
?- [a,[b,c]] = [A|[[B,c]]].
```

Cierto con `A=a` y `B=b`, puesto que:

- *lo que hay a la izquierda de `|`, la variable `A`, unifica con el primer elemento de la lista `[a,[b,c]]` con `A=a`*
- *lo que hay a la derecha de `|`, `[[B,c]]`, unificia con el resto de la lista `[a,[b,c]]`, que es `[[b,c]]`, con `B=b`*

```
?- [a,b,c] = [a, B | [C]].
```

Lo anterior es cierto siempre y cuando se unifique la variable B con la constante b y el término [C] unifique con el resto de la lista de la izquierda, que es la lista [c], y esto último se consigue sustituyendo C por c.

```
?- [[a], [b,c], [D]] = [[A] | R].
```

Cierto con `A=a` y `R = [[b, c], [D]]`, puesto que:

- *lo que hay a la izquierda de `|`, la lista `[A]`, unifica con el primer elemento de la lista de la izquierda, `[a]`, con `A=a`*
- *lo que hay a la derecha de `|`, `R`, unificia con el resto de la lista de la izquierda, que es la lista `[[b, c], [D]]`, con `R = [[b, c], [D]]`*

En la siguiente consulta `p` no es más que el nombre de un término compuesto cuyo único objeto es unificar sus argumentos.

```
?- p([C],3,[C|NL]) = p(R,N,[1,2]), [X|Y] = [N,C|R], Z is 2*X.
```

Note en particular que la consulta hace la unificación `R = [1]` y NO `R = [C]`, debido a que la sustitución inicial `R = [C]` DEBE COMPONERSE con la sustitución posterior `C = 1`. Ojo también con `N` y `X`, AMBAS unificadas con `3` (aunque SWI-Prolog escriba `N = X, X = 3`).

2. Uso de predicados básicos sobre listas y del predicado de corte

Antes de ejecutar las siguientes consultas piense qué ocurrirá al ejecutarlas (se producirá un error, la respuesta será false, la respuesta será true, se producirá una computación infinita, la o las respuestas serán ....). Tenga en cuenta en particular que las implementaciones de `member` y `append` son las estudiadas en clase para los predicados `pertenece` y `concatena`:

- `member(E,L)` `(pertenece)`, usado con `E` de salida y `L` de entrada, al hacer backtracking, va unificando `E` con los distintos elementos de la lista `L` (empezando por su cabeza).
- `append(L1,L2,L)` `(concatena)`, usado con `L1` y `L2` de salida y `L` de entrada, al hacer backtracking, va unificando `L1` y `L2` con las posibles listas que resultan de descomponer en dos trozos la lista `L` (empezando con `L1=[], L2=L`).

En los apuntes puede repasar las implementaciones de estos predicados, así como algunos ejemplos de árboles de Resolución que los involucran. Recuerde también que el predicado de corte, `!`, impide la reevaluación por backtracking de los sub-objetivos que le preceden: por ejemplo, en la evaluación de algo del

estilo `Obj1, Obj2, !, Obj3`, el corte, si se llega a él, impide las posibles reevaluaciones tanto de `Obj1` como de `Obj2`, pero NO impide las posibles reevaluaciones de `Obj3` .

```
?- member(A,[[a,b],[c,d]]), member(B,A).
```

Recuerde que Prolog evalúa de izquierda a derecha y en profundidad:

- *La primera solución del sub-objetivo de la izquierda es `A = [a,b]`, por lo que una vez resuelto le queda por resolver `member(B, [a,b])`. Este último tiene dos soluciones: la primera es `B = a`, dando lugar a la solución global `A = [a,b], B = a` y la segunda es `B = b`, dando lugar a `A = [a,b], B = b`.*
- *La segunda solución del sub-objetivo de la izquierda es `A = [c,d]`, teniendo entonces que resolver `member(B, [c,d])`, lo cual, de forma análoga al caso anterior, da lugar a las otras dos soluciones: `A = [c,d], B = c` y `A = [c,d], B = d`.*

```
?- member(A,[[a,b],[c,d]]), !, member(B,A).
```

Observe cómo en esta última consulta:

- *el corte impide la reevaluación del sub-objetivo que tiene a la izquierda, por lo que desparecen las soluciones asociadas con `A = [c,d]` que sí aparecían en la consulta previa.*
- *el corte NO impide la reevaluación del sub-objetivo que está a su derecha, por lo que `member(B, [a,b])` produce las dos posibles soluciones.*

```
?- member(A,[[a,b],[c,d]]), member(B,A), !.
```

Observe cómo ahora el corte impide la reevaluación de los dos sub-objetivos de la consulta, puesto que ambos están situados antes del corte, encontrando en consecuencia solo la primera solución para cada uno de ellos.

```
?- append(X, Y, [a,b]), length(X, N), N =< 1.
```

En la consulta anterior, el predicado `append` intenta, al reevaluarse, todas las posibles formas de descomponer la lista `[a,b]` en dos trozos, `X` e `Y`, pero las condiciones posteriores eliminan la solución `X=[a,b], Y=[]` puesto que en este caso la longitud de `X, N=2,` no es menor o igual que 1.

```
?- append([X|_], Y, [a,b]), length(Y, N), N > 0.
```

En este caso solo hay una solución para la descomposición de `[a,b]`, que es `[X|_]=[a] (es decir, X=a), Y=[b]`, ya que, por un lado, el patrón `[X|_]` solo unifica con listas NO vacías, y por otro lado se exige que la lista `Y` no sea vacía puesto que tiene que tener longitud no nula.

```
?- append([X,_|_], Y, [a,b]), length(Y, N), N > 0.
```

No hay solución puesto que el patrón `[X,_|_]` solo unifica con listas con al menos DOS elementos y la lista `Y`, por las condiciones posteriores, no puede ser vacía, por lo que en total ambas sumarían al menos TRES elementos, cuando `[a,b]` solo tiene dos.

```
?- append(A, [_], [a,b,c]), reverse(A,B).
```

Recuerde que `[_]` solo unifica con listas conteniendo exactamente UN elemento, por lo que solo puede ser `[_]=[c]`, y por tanto `A=[a,b]` y `B=[b,a]`, la inversa de A.

```
?- append(A, [5], [1,2,3,4,5]), member(X,A), 1 =:= X mod 2.
```

El predicado `append` solo da una solución, `A=[1,2,3,4]`. El predicado `member` recorre, mediante backtracking, todos los elementos de A, pero la condición siguiente solo es cierta para aquellos que son impares.

```
?- append(A, [c], [a,b,c]), member(X,A), !.
```

El predicado `append` solo da una solución, `A=[a,b]`, pero a diferencia de la consulta anterior, en esta el corte impide la reevaluación del predicado `member` al hacer backtracking, por lo que X solo toma un valor, el primero de la lista A.

```
?- append(A, B, [a,b,c]), member(X,A).
```

En esta última consulta, note que la opción `A=[], B=[a,b,c]` no aparece debido al fallo que produce la llamada posterior a `member(X,[])`.

```
?- append([a],B,L), member(b,B).
```

El `append` inicial lo único que indica es que `L` es una lista que empieza por `a` y que `B` es una lista cualquiera. Esto último hace que el `member` siguiente vaya dando todas las posibles listas que contienen a `b` (la que lo tiene en primera posición, la que lo tiene en segunda posición, etc).

```
?- append([C|R], S, [4, 2, 3, 1]), sumlist([C|R], N), N =< 6.
```

El predicado `append` descompone la lista `[4, 2, 3, 1]` en dos trozos, pero como el primero de ellos, `[C|R]`, no puede ser vacío, da lugar a las soluciones `[C|R]=[4] (C=4,R=[]), S=[2,3,1]`, `[C|R]=[4,2] (C=4,R=[2]), S=[3,1]`, `[C|R]=[4,2,3] (C=4,R=[2,3]), S=[1]` y `[C|R]=[4,2,3,1] (C=4,R=[2,3,1]), S=[]`. Sin embargo, las dos condiciones siguientes obligan a que la suma de los elementos de la lista `[C|R]` sea menor o igual que 6, lo cual hace que solo se mantegan las dos primeras descomposiciones.

```
?- append([C|R], S, [4, 2, 3, a]), sumlist([C|R], N), N =< 6.
```

El error aritmético en tiempo de ejecución lo produce el predicado `sumlist` al llegar a la última descomposición posible, `[C|R] = [4,2,3,a], S=[]` (la descomposición anterior falla en el último predicado al ser `N = 9`).

```
?- append([C|R], S, [4, 2, 3, a]), !, sumlist([C|R], N), N =< 6.
```

Observe el efecto del corte, que poda cualquier reevaluación del predicado `append`.

```
?- append([C|R], S, [4, 2, 3, 1]), sumlist([C|R], N), N >= 20.
```

Ninguno de los posibles valores para `[C|R]` cumple que sus elementos sumen al menos 20.

```
?- p(E, [_,E|R]) = p(3, [2|[F,4]]), append(A, B, [F|R]),
   length(A, LA), LA =< 1.
```

Prolog evalúa los objetivos anteriores de izquierda a derecha:

- *En primer lugar resuelve las unificaciones* `E=3` *y* `[_,3|R]=[2,F,4]` *(puesto que E se reemplaza por 3), mediante* `E=3, F=3, R = [4].`
- *El siguiente objetivo tiene que dividir la lista* `[F|R]=[3,4]` *en dos trozos, que, dada la implementación de* `append` *, son inicialmente* `A=[], B=[3,4].`
- *por último calcula la longitud de A, que es 0, y comprueba si 0 es menor o igual que 1. Como esto último es cierto, da la primera solución:*
  `E=3, R=[4], F=3, A=[], B=[3,4], LA=0`
- *Si se piden más soluciones, retrocede, de derecha a izquierda, en busca del primer subobjetivo reevaluable, que resulta ser el* `append` *, cuya segunda solución es* `A=[3], B=[4].` *En este caso se tendrá* `LA=1` *, por lo que sigue siendo cierta la última comprobación y se obtiene la segunda solución:*
  `E=3, R=[4], F=3, A=[3], B=[4], LA=1`
- *Si se piden más soluciones, Prolog vuelve a retroceder y reevalúa el* `append` *, cuya tercera solución es* `A=[3,4], B=[]` *. En este caso se tendrá* `LA=2` *, por lo que la última comprobación ya no es cierta y el objetivo falla.*
- *Ante el fallo anterior Prolog vuelve a retroceder pero ya no le quedan objetivos reevaluables, por lo que termina.*

```
?- q([a|[b,c]], F) = q([E,_|S], E), append(U, V, [F|S]),
   length(U, LU), LU >= 1.
```

Razonamiento muy similar al anterior.

3. Implementación de algunos predicados sobre listas

Implemente los predicados para el manejo de listas en Prolog descritos a continuación y:

- Compruebe que sus implementaciones son correctas haciendo consultas para casos significativos.
- Pida a Prolog todas las posibles soluciones y asegúrese de que los predicados implementados **solo devuelven la o las soluciones correctas. En caso de que no sea así, use el predicado de corte adecuadamente.**
- Cuando su implementación presente **recursión lineal** (una única llamada recursiva) pero no sea **recursión final** (la llamada recursiva es lo último que se ejecuta) ni **recursión final módulo cons** (la llamada recursiva es lo penúltimo que se ejecuta, siendo lo último la construcción de la cabeza de una lista), *implemente una segunda versión con* ***recursión final (o recursión final módulo cons)*** mediante el uso de parámetros de acumulación, de forma similar a como se hace en los apuntes con el predicado factorial o el predicado inversa.

```
1  % enesimo(?L, +N, ?E)
2  % cierto si E es el elemento de la lista L situado
3  % en la posición N, entendiendo que la cabeza de la
4  % lista está situada en la posición 1 (debe fallar
5  % en cualquier otro caso).
6  %
7  % Proporcione una implementación RECURSIVA.
8
9      enesimo(L, N, E) :-
10         integer(N),
11         N >= 1,
12         enesimo_aux(L, N, E).
```

```
13      enesimo_aux([C|_], 1, C).
14      enesimo_aux([_|R], N, E) :-
15      N > 1,
16      N1 is N-1,
17      enesimo_aux(R, N1, E).
```

La comprobación N > 1 de la segunda cláusula de enesimo_aux sirve para indicar, en caso de backtracking, que cuando sea N=1 no debe aplicarse la segunda regla. Esta comprobación podría sustituirse por un corte en el cuerpo de la primera cláusula del programa (es decir, sustituyendo "enesimo([C|_], 1, C)." por "enesimo([C|_], 1, C) :- !."). En cualquier caso esta precaución es recomendable por razones de eficiencia pero no es imprescindible, puesto que aunque entrase por la segunda regla con N=1 Prolog acabaría fallando al vaciarse la lista (el predicado falla para listas vacías). Lo mismo ocurriría si "N" fuese mayor que la longitud de L, por lo que no es necesario comprobar que no lo sea.

```
?- enesimo([],1,E).
```

```
?- enesimo([1,2,3], -1, E).
```

```
?- enesimo([1,2,3],2, E).
```

```
?- enesimo([1,2,3], 4, E).
```

```
1  % enesimo_norec(?L, +N, ?E)
2  % cierto si E es el elemento de la lista L situado
3  % en la posición N, entendiendo que la cabeza de la
4  % lista está situada en la posición 1.
5  %
6  % Proporcione una implementación NO RECURSIVA, basada en el uso
7  % de algunos de los predicados predefinidos para el manejo de listas.
8  % PISTA: si usa append para descomponer la lista L en dos trozos
9  % tales que el primer trozo tenga tamaño N-1, el primer elemento del
10 % segundo trozo será el N-ésimo de L.
11 %
12 enesimo_norec(L, N, E) :-
13     integer(N),
14     N > 0,
15     append(Prefijo, [E|_], L),
16     N1 is N - 1,
17     length(Prefijo, N1),
18     !.   % por eficiciencia: para de buscar una vez encontrado el
19          % (único) prefijo de tamaño N-1.
```

Observe en el código anterior que si N es mayor que el tamaño de la lista, Prolog no será capaz de descomponer la lista en dos trozos de forma que el primero tenga tamaño N-1 y el segundo sea no vacío, por lo que fallará. En otro caso, una vez encontrado el único prefijo posible de tamaño N-1, no tiene sentido seguir buscando descomposiciones, de ahí el corte, !, que evita el backtracking.

```
1  % elimina_n(?L, +N, ?NL)
2  % cierto si NL es la lista resultante después de
3  % eliminar el elemento de L situado en la posición N,
4  % entendiendo que la cabeza de la lista está situada
```

```
% en la posición 1. Si N es mayor que la longitud de
% L, NL será igual a L.
%
% Proporcione una implementación RECURSIVA
%
elimina_n(L, N, NL) :-
    integer(N),
    N >= 1,
    elimina(L, N, NL).
elimina([], _ , []).
elimina([_|R], 1, R).
elimina([C|R], N, [C|NR]) :-
     N > 1,
     N1 is N-1,
     elimina(R, N1, NR).
```

En este último caso la comprobación "N > 1" de la última cláusula (que podría sustituirse por un corte en la cláusula anterior: `elimina([_|R], 1, R) :-!. `) es imprescindible para evitar soluciones incorrectas en caso de backtracking.

```
% elimina_n(?L, +N, ?NL)
% cierto si NL es la lista resultante después de eliminar el
% elemento de L situado en la posición N, entendiendo que la
% cabeza de la lista está en la posición 1.
% Si N es mayor que la longitud de L, NL será igual a L.
%
% Proporcione una implementación NO RECURSIVA, basada en el uso
% de algunos de los predicados predefinidos para el manejo de listas.
%
% PISTA: descomponga la lista en dos trozos adecuados y vuelva
% a componerlos obviando el elemento n-ésimo. Tenga en cuenta
% que si N es mayor que la longitud de la lista, el intento
% de descomposición fallará y tendrá que tratar ese caso aparte.

elimina_n_norec(L, N, NL) :-
    integer(N),
    N > 0,
    append(Prefijo, [_|Sufijo], L),
    N1 is N - 1,
    length(Prefijo, N1),
    !,
    append(Prefijo, Sufijo, NL).

elimina_n_norec(L, N, L) :-
    integer(N),
    N > 0.
```

Note que en la primera regla del programa anterior, si Prefijo tiene tamaño N-1, "_" es el elemento n-ésimo, que hay que eliminar. Gracias al corte de la primera regla, Prolog solo llegará a la segunda si N no cumple las condiciones o no es posible descomponer L de la forma requerida, y esto último solo ocurrirá si N es mayor que la longitud de L.

```
% elimina_n_bis(?L, +N, ?NL)
% cierto si NL es la lista resultante después de
% eliminar el elemento de L situado en la posición N,
% entendiendo que la cabeza de la lista está situada
% en la posición 1. **El predicado debe fallar si N es mayor
% que la longitud de L**
%
% Proporcione una implementación RECURSIVA.
%
  elimina_n_bis(L, N, NL) :-
    integer(N),
    N >= 1,
    elimina_bis(L, N, NL).
  elimina_bis([_|R], 1, R).
  elimina_bis([C|R], N, [C|NR]) :-
     N > 1,
     N1 is N-1,
     elimina_bis(R, N1, NR).
```

```
% elimina_n_bbis(?L, +N, ?NL)
% cierto si NL es la lista resultante después de
% eliminar el elemento de L situado en la posición N,
% entendiendo que la cabeza de la lista está situada
% en la posición 1. El predicado debe fallar si N es mayor
% que la longitud de L.
%
% Proporcione una implementación NO RECURSIVA, basada en el uso
% de algunos de los predicados predefinidos para el manejo de listas.
% PISTA: descomponga la lista en dos trozos adecuados y vuelva
% a componerlos obviando el elemento n-ésimo.
%

 elimina_n_bis_norec(L, N, NL) :-
    integer(N),
    N > 0,
    append(Prefijo, [_|Sufijo], L),
    N1 is N - 1,
    length(Prefijo, N1),
    !,
    append(Prefijo, Sufijo, NL).
```

```
% borrartodos(+L,+E,?NL)
% cierto si NL es la lista resultante después de
% eliminar de L todos los elementos idénticos a E
% (si los hay). Tenga en cuenta que no deben eliminarse
% elementos que sean unificables con E, sólo aquellos que
% sean idénticos. Por ejemplo,borrartodos([a,b],X,[a,b]) y
% borrartodos([a,X], a, [X]) deben ser ciertos.

borrartodos([],_,[]).
borrartodos([C|R],E,NR) :-
```

```
      C == E,
      !,
      borrartodos(R,E,NR).
borrartodos([C|R],E,[C|NR]) :-
      borrartodos(R,E,NR).
```

```
?- Your query goes here ...
```

```
% insertardetras(+L,+E1,+E2,?NL)
% cierto si NL es la lista resultante de insertar
% E2 justo detrás de cada aparición de E1 en L
%

insertardetras([],_,_,[]).
insertardetras([E1|R],E1,E2,[E1,E2|NR]) :-
     !,
     insertardetras(R,E1,E2,NR).
insertardetras([C|R],E1,E2,[C|NR]) :-
     insertardetras(R,E1,E2,NR).
```

Los cortes de los dos predicados anteriores son imprescindibles para evitar soluciones incorrectas en caso de backtracking. Un efecto similar podría haberse obtenido añadiendo las comprobaciones "C \= E" (en borrartodos) o "C \= E1" (en insertardetras) al principio del cuerpo de su última cláusula.

```
% divide(+L, ?LNeg, ?LPos)
% cierto si L es una lista de números, LNeg es una
% lista con los números negativos de L y LPos es
% una lista con los números nulos o positivos de L,
% en ambos casos respetando el orden en el que
% aparecen en L.
% Facilite dos versiones distintas de este predicado,
% la segunda de ellas utilizando el corte.

% VERSIÓN 1: Sin utilizar el predicado de corte
divide_v1([],[],[]).
divide_v1([C|R], [C|R1], R2) :-
     number(C),
     C < 0,
     divide_v1(R,R1,R2).
divide_v1([C|R], R1, [C|R2]) :-
     number(C),
     C >= 0,
     divide_v1(R,R1,R2).
```

```
% VERSIÓN 2: Utilizando el predicado de corte
divide_v2([],[],[]).
divide_v2([C|R], [C|R1], R2) :-
     number(C),
     C < 0,
     !,
     divide_v2(R,R1,R2).
```

```
8  divide_v2([C|R], R1, [C|R2]) :-
9       number(C),
10      % ya no es necesario comprobar C >= 0 puesto que solo
11      % se entrará por esta regla cuando eso ocurra.
12      divide_v2(R,R1,R2).
```

```
1  % cuenta_const(+TC, +L, ?N)
2  % cierto si N es el número de apariciones del término
3  % constante C como elemento de la lista L. Recuerde
4  % que Prolog proporciona predicados predefinidos para la
5  % clasificación de términos, entre los que figura atomic(X),
6  % cierto si X es un término constante. Tenga en cuenta que
7  % no deben contarse elementos que sean unificables con TC,
8  % sólo aquellos que sean idénticos. Por ejemplo, la consulta
9  % “?- cuenta_const(a, [a,[a,p(a),b],X],N).” debe ser cierta con N=1,
10 % puesto que sólo una de las a’s que aparecen en L es un elemento de
11 % L (el resto de a’s están contenidas en el segundo elemento de L)
12 % y la variable X no debe contarse ya que, aunque es unificable con a,
13 % no es una a.
14 %
15 cuenta_const(TC, L, N) :-
16     atomic(TC),
17     cuenta_const1(TC, L, N).
18 cuenta_const1(_, [], 0).
19 cuenta_const1(TC, [C|R], N) :-
20       TC == C,
21       !,
22       cuenta_const1(TC, R, NR),
23       N is NR + 1.
24 cuenta_const1(TC, [_|R], N) :-
25      cuenta_const1(TC, R, N).
```

```
1  % implementación con recursión final
2
3  cuenta_const_acc(TC, L, N) :-
4      atomic(TC),
5      cuenta_const_acc(TC, L, 0, N).
6
7  cuenta_const_acc(_, [], A, A).
8  cuenta_const_acc(TC, [C|R], A, N) :-
9      TC == C,
10     !,
11     A1 is A+1,
12     cuenta_const_acc(TC, R, A1, N).
13 cuenta_const_acc(TC, [_|R], A, N) :-
14     cuenta_const_acc(TC, R, A, N).
```

```
1  % aplana(+L,?NL)
2  % cierto si L es una lista y NL es la lista formada por todos los
3  % términos que aparecen en L, quitando las listas internas de L en
4  % caso de que las haya. Por ejemplo, la consulta
```

```
% ?- aplana([a, [X,p(a,Z),[b,[c]]], V], A).
% debe ser cierta con A = [a, X, p(a,Z), b, c, V]
% Pista: utilice los predicados is_list/1 y append/3.
%
aplana([], []).
aplana([C|R], L) :-
    is_list(C),
    !,
    append(C, R, NL),
    aplana(NL, L).
aplana([C|R], [C|NR]) :-
    aplana(R, NR).
```

```
% empaqueta(+L, ?NL)
% cierto si NL contiene los mismos elementos que la lista L pero
% de forma que toda ristra de elementos iguales consecutivos
% aparece empaquetada en una sublista. Por ejemplo, la consulta
% “empaqueta([a,a,a,b,a,a,c,c,d], X).” debe devolver cierto
% con X=[[a,a,a], [b], [a,a], [c,c], [d]]. Una posible solución
% consiste en implementar y utilizar en la implementación de
% empaqueta el predicado auxiliar lista_cabeza(+L, ?LC, ?LR),
% cierto si L es una lista, LC es la lista formada por la ristra
% de elementos iguales consecutivos que encabeza L y LR es lo que
%  queda en la lista. Por ejemplo, la consulta
% “?- lista_cabeza([a,a,a,b,a,a,c,c,d], X, Y).” debe se cierta
% con X=[a,a,a] e Y=[b,a,a,c,c,d].

% Implementando y usando lista_cabeza
lista_cabeza([], [], []).
lista_cabeza([C], [C], []).
lista_cabeza([C1,C2|R], [C1], [C2|R]) :-
       C1 \== C2,
       !.
lista_cabeza([C,C|R], [C|LC], LR) :-
       lista_cabeza([C|R], LC, LR).

empaqueta([], []).
empaqueta([C|R], [LC|NR]) :-
        lista_cabeza([C|R], LC, LR),
        empaqueta(LR, NR).
```

```
% otra posible implementación de empaqueta, sin usar lista_cabeza

empaquete([], []).
empaquete([C],[[C]]).
empaquete([C1,C2|R], [[C1]|ER]) :-
        C1 \== C2,
        !,
        empaquete([C2|R], ER).
empaquete([C1,C2|R], [[C1|E1]|ER]) :-
        empaquete([C2|R], [E1|ER]).
```

```
% codifica(+L, ?NL), cierto si L es una lista y NL es una lista
% cuyos elementos son listas de la forma [E,N], donde cada uno de
% estos pares representa una ristra de N elementos E consecutivos en
% L. Por ejemplo, la consulta “?- codifica([a,a,a,b,a,a,c,c,d], LL).”
% debe ser cierta con LL = [[a,3], [b,1], [a,2], [c,2], [d,1]].
% Para la implementación de este predicado puede utilizar el
% predicado empaqueta/2 anterior así como el predicado predefinido
% length(?L, ?N), cierto si N es la longitud de la lista L.
%
codifica(L, NL) :-
        empaqueta(L, LL),
        transforma(LL, NL).

transforma([], []).
transforma([[C|LC]|R], [[C,N]|LR]) :-
         length([C|LC], N),
         transforma(R, LR).
```

```
% Implementación directa de codifica, alternativa a la anterior

codific([], []).

 codific([C|R], [[C,NC]|NR]) :-
    codific(R, [[C,N]|NR]),
    !,
    NC is N + 1.

 codific([C|R], [[C,1]|NR]) :-
    codific(R, NR).
```

```
% asteriscos(+L)
% recibe una lista de números naturales positivos y escribe para cada
% uno de ellos una línea con tantos asteriscos como indique el número
% correspondiente. Por ejemplo, la consulta “?- asteriscos([2,1,3]).”
% debe devolver cierto y debe producir el efecto
%  **
%  *
%  ***

asteriscos([]).
asteriscos([C|R]) :-
    escribe_asteriscos(C),
    nl,
    asteriscos(R).

escribe_asteriscos(1) :- write(*).
escribe_asteriscos(N) :-
    N>1,
    write(*),
    N1 is N-1,
    escribe_asteriscos(N1).
```

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº4: Negación, recolección de soluciones y predicados de orden superior

Soluciones propuestas, comentadas

Autora: A. Pradera

Prerrequisitos
Para la realización de esta cuarta práctica de Programación Lógica es necesario haber leído y estudiado el tema PL3, dedicado al predicado de negación, los predicados de recolección de soluciones y los predicados de orden superior. Para el último ejercicio es recomendable además haber estudiado el ejemplo del coloreado de mapas.

1. A vueltas con la genealogía

El objetivo de este ejercicio es ampliar el programa de genealogía de la Práctica 1, algunos de cuyos predicados básicos se recuerdan a continuación.

```
% PREDICADOS BÁSICOS

% progenitor(?X, ?Y)
% Cierto si X es progenitora o progenitor de Y
progenitor(pepa, pepe1).
progenitor(pepe, pepe1).
progenitor(pepe, pepa2).
progenitor(pepe1, pepa11).
progenitor(pepe1, pepa12).
progenitor(pepa12, pepe121).

% mujer(?X)
% Cierto si X es una mujer
mujer(pepa).
mujer(pepa2).
mujer(pepa11).
mujer(pepa12).

% varon(?X)
% Cierto si X es un varón
varon(pepe).
varon(pepe1).
varon(pepe121).
```

```
% ALGUNOS PREDICADOS MÁS
%
% abuelo(?X,?Y), cierto si X es abuelo de Y.
abuelo(X,Y) :-
    progenitor(X,Z),
    progenitor(Z,Y),
```

```
    varon(X).

% ancestro(?X,?Y), cierto si X es un ancestro (mujer o varón) de Y.
ancestro(X,Y) :-
    progenitor(X,Y).
ancestro(X,Y) :-
    progenitor(X,Z),
    ancestro(Z,Y).
```

1.1. Sabiendo que dispone del programa anterior, haga uso de los predicados de recolección de soluciones que considere oportunos para ampliar el programa con los sigiuentes predicados:

```
% num_descendientes(+X,?N)
% cierto si X tiene N descendientes (N puede ser 0).
%
num_descendientes(X,N) :-
    findall(D, ancestro(X,D), L),
    length(L,N).
% findall en lugar de bagof para que N pueda ser 0.
```

```
?- num_descendientes(pepa, N).
```

```
?- num_descendientes(pepe121, N).
```

```
% ancestros(+X, ?L)
% cierto si L es una lista conteniendo todos los
% ancestros de X. Debe fallar si X no tiene ancestros.
%
ancestros(X,L):-
    bagof(Y, ancestro(Y,X), L).
% bagof en lugar de findall para que falle si X
% no tiene ancestros
```

```
?- ancestros(pepa, L).
```

```
?- ancestros(pepe121, L).
```

```
% listas_descencientes(?L)
% cierto si L es una lista de terminos desc(X,DX) donde
% X es una persona con descendientes y DX es una lista
% con sus descendientes.
%
listas_descendientes(L) :-
    bagof(des(X,DX), bagof(Y, ancestro(X,Y), DX), L).
% el bagof interno no puede ser un findall porque no hay que
% cuantificar la X
```

```
?- listas_descendientes(L).
```

```
% desc_no_directos(+X, ?L)
```

```
% cierto si L es una lista (puede ser vacía) conteniendo
% a todos los descendientes de X salvo sus hijas/os
%
desc_no_directos(X, L) :-
    findall(Y, (ancestro(X,Y), \+ progenitor(X,Y)), L).
```

```
?- desc_no_directos(pepa, L).
```

```
?- desc_no_directos(pepa12, L).
```

```
% ascendientes_varones_hasta_2(+X,?L)
% cierto si L contiene todos los ascendientes varones
% de X de grado máximo 2 (es decir, padre y abuelos).
% Debe fallar si no hay ninguno

ascendientes_varones_hasta_2(X,L) :-
    bagof(Y,
          ((progenitor(Y,X), varon(Y)) ; abuelo(Y,X)),
          L).
```

```
?- ascendientes_varones_hasta_2(pepe121,L).
```

```
?- ascendientes_varones_hasta_2(pepa12,L).
```

```
?- ascendientes_varones_hasta_2(pepe,L).
```

```
% con_descendientes(-L, -N)
% cierto si L es el conjunto de personas que tiene algún
% descendiente y N es el número de esas personas
% (si no hubiese ninguna, debe devolver lista vacía y cero).
%
con_descendientes(L, N) :-
    setof(X, Y^progenitor(X,Y),L), !,
    % OJO a la cuantificación existencial de Y, Y^:
    % dame en L el conjunto de todos los X para los que
    % *existe* un Y tal que X es progenitor de Y
    length(L,N).
% es necesario usar setof para evitar repeticiones

con_descendientes([], 0).
% si falla setof, hay que devolver [] y cero.
```

```
?- con_descendientes(L,N).
```

1.2. Sabiendo que dispone del programa anterior, haga uso de algún predicado de orden superior clásico para ampliar el programa con el siguiente predicado:

```
% progenitores(?L1, ?L2)
% cierto si L1 y L2 son listas tales que cada persona de L1 es
% progenitor/a de la persona situada en la misma posición de L2.
```

```
4 % Una de las dos listas debe ser de entrada.
5 %
6   progenitores(L1, L2) :-
7     maplist(progenitor, L1, L2).
```

```
?- progenitores([pepa, pepa12], L).
```

```
?- progenitores([pepa, pepa121], L).
```

La consulta anterior falla porque pepa121 no es progenitora de nadie.

```
?- progenitores([pepe], L).
```

Observe cómo la consulta anterior produce dos soluciones, debido a que el predicado "progenitor" es una relación y no una función (en particular, pepe tiene dos hijos).

```
?- progenitores(L, [pepa11, pepa12]).
```

```
?- progenitores(L, [pepa]).
```

2. El mundo de los bloques, ahora con pesos

Considere el ejemplo estudiado en clase relativo al *mundo de los bloques* que se recuerda a continuación (ahora con un bloque más, el bloque "d", apilado sobre el bloque "c"):

```
1  %% EL MUNDO DE LOS BLOQUES
2  %
3  % encima(?X, ?Y)
4  % cierto si el bloque X está justo encima del bloque Y
5         encima(b,a).
6         encima(c,b).
7         encima(d,c).
8
9  % apilado(?X, ?Y)
10 % cierto si el bloque X está apilado sobre el bloque Y
11 % (no necesariamente justo encima)
12 %
13         apilado(X,Y) :-
14             encima(X,Y).
15
16         apilado(X,Y) :-
17             encima(X,Z),
18             apilado(Z,Y).
```

Suponga además que se conoce el peso de cada uno de los bloques manejados. Se pide:

2.1 Decida cómo representar esta información sobre pesos y complete el programa suponiendo que los bloques a,b,c y d pesan, respectivamente, 70, 30, 30 y 10 kilos.

```
1  %% Decida a continuación cómo representar el peso de los
```

```
%% bloques
%
% peso(?B, ?P),
% cierto si B pesa P kilos
peso(a, 70).
peso(b, 30).
peso(c, 30).
peso(d, 10).
```

2.2 Implemente el predicado `peso_medio(+B, ?M)`, cierto si M es el peso medio de los bloques que están apilados sobre B pero que no están justo encima suyo. El predicado debe fallar (devolver "false") si no hay ningún bloque que cumpla lo anterior. Por ejemplo, si, como en los programas de más arriba, se tuviese la pila de bloques "abcd", con "d" en la cima y con los pesos ya mencionados, la consulta "?- peso_medio(a, M)" devolvería "M=20", mientras que la consulta "peso_medio(c, M)" fallaría.

```
% peso_medio(+B, ?M)
% cierto si M es el peso medio de los bloques que están
% apilados sobre B pero que no están justo encima suyo
%

peso_medio(B, M) :-
    bagof(P, X^(apilado(X, B), \+ encima(X,B), peso(X,P)), LPesos),
    % OJO con la cuantificación existencial de X, X^.
    sumlist(LPesos, PesoTotal),
    length(LPesos, NumBloques),
    M is PesoTotal/NumBloques.

/*
 Observe que en el código anterior no se puede usar
 "findall" porque el enunciado dice que el predicado
 "peso_medio" debe fallar si no hay ningún bloque
 cumpliendo las condiciones pedidas.Por otra parte, es
 necesario usar "bagof" en lugar de "setof" porque
 podría haber bloques con mismo peso apilados sobre el
 bloque dado.
 */
```

```
?- peso_medio(a,M).
```

```
?- peso_medio(c,M).
```

```
?- peso_medio(b,M).
```

2.3 Implemente el predicado `pesos(-L)`, cierto si L es una lista conteniendo todos las tuplas (B,P,PA) donde B es un bloque sobre el que hay apilado al menos otro bloque, P es el peso de B y PA es el peso que soporta B. El predicado debe devolver la lista vacía si no hubiese ningún bloque con las características requeridas. Por ejemplo, si, como en los programas de más arriba, se tuviese la pila de bloques "abcd", con "d" en la cima y con los pesos ya mencionados, la consulta "?- pesos(-L)" devolvería "L = [(a,70,70), (b, 30, 40), (c, 30, 10)]

```
% pesos(-L)
% cierto si L es la lista de tuplas (B,P,PA) donde B es un
```

```
% bloque sobre el que hay apilado al menos otro bloque,
% P es el peso de B y PA es el peso que soporta B.

pesos(L) :-
   findall((B,P,PA),
            cumple(B,P,PA),
           L). % lista vacía si no hay ninguno

cumple(B, P, PA) :-
   % B es un bloque con peso P
   peso(B, P),
   % Recolección de los pesos apilados sobre B
   % (debe ser bagof para fallar si B no tiene bloques apilados
   % y para admitir posibles pesos repetidos)
   bagof(PesoBloque,
         Bloque^(apilado(Bloque, B), peso(Bloque, PesoBloque)),
         % OJO con la cuantificación existencial, ^, de Bloque
         ListaPesos),
   % PA es la suma de los pesos apilados sobre B
   sumlist(ListaPesos, PA).
```

```
%% Equivalente a lo anterior, pero todo en el mismo predicado:

pesos_todo(L) :-
   findall((B,P,PA),
           (peso(B, P),
            bagof(PesoBloque,
                  Bloque^(apilado(Bloque, B),
                          peso(Bloque, PesoBloque)),
                  ListaPesos),
            sumlist(ListaPesos, PA)),
           L).
```

```
?- pesos_todo(L).
```

2.4 Considere las consultas descritas a continuación e indique para cada una de ellas, antes de usar el intérprete, si Prolog produciría algún error o daría algún resultado (especificando error(es) o resultado(s)). Razone su respuesta.

```
?- maplist(apilado(b),[a]).
```

Efectivamente la consulta anterior ejecuta apilado(b,a), que resulta ser cierto, por lo que devuelve true.

```
?- maplist(apilado, [a,b]).
```

La consulta anterior da un error puesto que intenta ejecutar apilado(a) y apilado/1 no está definido (sí lo está apilado/2).

```
?- maplist(apilado, [d,c], [a,b]).
```

La consulta anterior ejecuta apilado(d,a), que es cierto, y apilado(c,b), que también es cierto, por lo que devuelve true.

```
?- maplist(\+ apilado, [a,b], [c,d]).
```

*La consulta anterior produce un error, dado que el objetivo que se pasa a un predicado de orden superior **no puede estar negado**. Para conseguir lo anterior es necesario implementar aparte un nuevo predicado, como se hace a continuación:*

```
1 % noapilado(+X,+Y)
2 % cierto si X no está apilado sobre Y
3 % (observe que los dos parámetros han de ser de entrada puesto que
4 % la negación de Prolog no sirve para computar)
5 noapilado(X, Y) :-
6     \+ apilado(X,Y).
```

El predicado anterior sí se puede usar para averiguar si los bloques de una lista dada no están apilados sobre los correspondientes bloques de otra lista:

```
?- maplist(noapilado, [a,b], [c,d]).
```

```
?- include(apilado(d), [a,b,c,d], L).
```

La consulta anterior ejecuta apilado(d,a), apilado(d,b), apilado(d,c) y apilado(d,d), y se queda con aquellos que resultan ser ciertos (todos menos el último).

```
?- exclude(apilado(c), [a,b,c,d], L), maplist(encima, L, LE).
```

La consulta anterior ejecuta en primer lugar apilado(c,a), apilado(c,b), apilado(c,c) y apilado(c,d), excluyendo aquellos que resultan ser ciertos, por lo que L = [c,d]. A continuación ejecuta encima(c,LE1) y encima(d,LE2), dando lugar a LE=[LE1,LE2]=[b,c].

```
?- include(apilado(c), [a,b,c,d], L), maplist(encima, L, LE).
```

La consulta anterior ejecuta en primer lugar apilado(c,a), apilado(c,b), apilado(c,c) y apilado(c,d), filtrando aquellos que resultan ser ciertos, por lo que L = [a,b]. A continuación, siendo LE=[LE1,LE2], ejecuta encima(a,LE1), que es falso, por lo que maplist falla y en consecuencia la consulta devuelve false.

```
?- include(encima, [c,d], X).
```

La consulta anterior da un error puesto que empieza intentando ejecutar encima(c) y encima/1 no está definido (sí lo está encima/2).

```
?- include((apilado(d),\+ encima(d)), [a,b,c,d], X).
```

*La consulta anterior produce un error, dado que el objetivo que se pasa a un predicado de orden superior **no puede ser compuesto**. Para conseguir lo anterior es necesario implementar aparte un nuevo predicado, como se hace a continuación:*

```
1 % apilado_no_encima(?X,?Y)
2 % cierto si X está apilado sobre Y pero no justo encima suyo
3
```

```
4  apilado_no_encima(X,Y) :-
5     apilado(X,Y),
6     \+ encima(X,Y).
```

```
?- apilado_no_encima(d,Y).
```

El predicado anterior sí se puede usar para filtrar, como se pretendía con la consulta anterior, los bloques de una lista sobre los que el bloque d está apilado pero no está justo encima:

```
?- include(apilado_no_encima(d), [a,b,c,d], X).
```

3. Uso básico de predicados de recolección de soluciones y otros predicados de orden superior

3.1. Considere las consultas descritas a continuación e indique para cada una de ellas, antes de usar el intérprete, si Prolog produciría algún error o daría algún resultado (especificando error(es) y/o resultado(s)). Razone su respuesta.

```
?- A=[a,b], B=[1,2|A], setof(P, C^T^append([C|P],T,B), L),
       maplist(length,L,LL), append(NL, [_], LL), sumlist(NL,S).
```

Prolog evalúa los objetivos de izquierda a derecha:

- Los dos primeros sub-objetivos son unificaciones cuyo efecto es unificar las variables "A y B" con las listas "[a,b] y [1,2,a,b]", respectivamente.
- El predicado "setof" construye en "L" el conjunto de todos los "P" para los que existe un elemento "C" y una lista "T" (recuerde que X^ significa *existe X tal que*) tales que la concatenación de "[C|P]" con "T" es igual a "[1,2,a,b]". Para que ocurra lo anterior necesariamente tiene que ser "C=1" y para el resto de valores existen las siguientes combinaciones: "P=[],T=[2,a,b]", "P=[2],T=[a,b]", "P=[2,a],T=[b]" y "P=[2,a,b],T=[]". Como el predicado "setof" solo se interesa por los "P", resultará "L=[[],[2],[2,a],[2,a,b]]".
- El predicado "maplist" unifica "LL" con la lista resultante de aplicar el predicado "length" a cada elemento de la lista "L" anterior, por lo que resultará "LL = [0,1,2,3]".
- El predicado "append" es cierto si existe una lista "NL" tal que concatenada con una lista de un único elemento da "LL = [0,1,2,3]", y lo anterior solo tiene una solución, "NL=[0,1,2]".
- Por último, el predicado "sumlist" es cierto si su segundo argumento es la suma de los elementos que componen la lista que recibe como primer argumento, por lo que hará la unficación "S=0+1+2=3".
- En definitiva, Prolog devuelve como primera solución:

```
A = [a, b],
B = [1, 2, a, b],
L = [[], [2], [2, a], [2, a, b]],
LL = [0, 1, 2, 3],
NL = [0, 1, 2],
S = 3
```

- Al hacer backtracking para buscar nuevas soluciones se comprueba que ninguno de los sub-objetivos tenía soluciones alternativas, por lo que no hay más soluciones a la consulta dada.

```
?- X = [1,2], Y = [3,4|X], bagof(S, C^P^append(P,[C|S],Y),L),

       append(LL,[_],L), maplist(length,LL,LN), sumlist(LN,N).
```

Aplicando un razonamiento similar al anterior se comprueba por qué Prolog ofrece como única solución la siguiente:

```
X = [1, 2],
Y = [3, 4, 1, 2],
L = [[4, 1, 2], [1, 2], [2], []],
LL = [[4, 1, 2], [1, 2], [2]],
LN = [3, 2, 1],
N = 6
```

```
?- findall(NY, (append(T,Y,[1,2,a,4]),
                length(Y,LY),
                LY >= 2,
                include(integer,Y,NY)),
           R),
   maplist(sumlist, R, NR),
   append(D1, [_|D2], NR),
   !.
```

El predicado `findall` construye en R la lista de todos los NY que resultan de filtrar (include) los números enteros de todos los posibles sufijos Y de [1,2,a,4] con tamaño igual o superior a 2, para lo cual existen las siguientes combinaciones:

- *T=[], Y=[1,2,a,4], LY=4, NY=[1,2,4]*
- *T=[1], Y=[2,a,4], LY=3, NY=[2,4] y*
- *T=[1,2], Y=[a,4], LY=2, NY=[4]*

ya que en el resto de posibles descomposiciones de [1,2,a,4] el sufijo Y tiene tamaño menor que 2. Como el predicado `findall` solo se interesa por los NY, resultará R=[[1, 2, 4], [2, 4], [4]]. A continuación el predicado `maplist/3` aplica `sumlist` a cada una de las listas de R, transformando cada lista en su suma, por lo que se obtiene NR = [1+2+4=7, 2+4=6, 4]. Por último el predicado `append` descompone NR en dos trozos, un prefijo cualquiera D1 y un sufijo que tiene que unificar con [_|D2], lo cual hace que en el primer intento de descomposición se tenga D1=[], _=7, D2=[6,4]. Si se retrocede en busca de más soluciones, las siguientes alternativas de `append` (el único con posibilidades de reevaluación) quedan podadas por el corte final. Por todo lo anterior, la consulta dada daría como única respuesta la siguiente: R = [[1 , 2 , 4] , [2 , 4] , [4]] , NR = [7 , 6 , 4] , D1 = [ ] , D2 = [6 , 4]

```
1 % resta(+X, +Y, ?Z)
2 % cierto si Z es la resta de X menos Y
3 resta(X,Y,Z) :- Z is X-Y.
```

```
?- foldl(resta,[1,2,3,4],10,R).
```

En efecto, teniendo en cuenta la implementación de foldl (ver apuntes), los cálculos efectuados son los siguientes:

- `resta(1, 10, V1)` , que unifica V1 =1-10 =-9
- `resta(2, -9, V2)` , que unifica V2 = 2-(-9) = 11
- `resta(3, 11, V3)` , que unifica V3 = 3-11 = -8
- `resta(4, -8, V4)` , que unifica V4 = 4 - (-8) = 12
- el cómputo termina con `foldl(resta, [], 12, R)` , haciendo la unificación R = 12.

3.2 Proponga una implementación alternativa, basada en el uso de predicados de orden superior, de los predicados `borrartodos, divide, cuenta_const, asteriscos` de la práctica nº 3.

```
% borrartodos(+L,+E,?NL)
% cierto si NL es la lista resultante después de
% eliminar de L todos los elementos idénticos a E
% (si los hay)
%
 borrartodos(L, E, NL) :-
    exclude(==(E), L, NL).
```

En efecto, el código `exclude(==(E), L, NL)` ejecuta `==(E,Li)` para cada uno de los elementos `Li` de la lista `L` y excluye aquellos para los que lo anterior es cierto.

```
?- borrartodos([a,b,X,a],a,L).
```

```
?- borrartodos([], a, L).
```

```
?- borrartodos([a,b,Z], X, L).
```

```
% divide(+L, ?LNeg, ?LPos)
% cierto si L es una lista de números, LNeg es una
% lista con los números negativos de L y LPos es
% una lista con los números nulos o positivos de L,
% en ambos casos respetando el orden en el que
% aparecen en L.
%
  divide(L, LNeg, LPos) :-
    maplist(number, L),
    include(>(0), L, LNeg),
    % ejecuta >(0,Li) para cada elemento Li de L
    % y se queda con los que dan cierto
    include(=<(0), L, LPos).
     % ..... =<(0,Li) ....
```

```
?- divide([1,2,-1,0,4,-1,-4], N,P).
```

```
% cuenta_const(+TC, +L, ?N)
% cierto si N es el número de apariciones del término
% constante TC como elemento de la lista L. Recuerde
% que Prolog proporciona predicados predefinidos para la
% clasificación de términos, entre los que figura atomic(X),
% cierto si X es un término constante. Tenga en cuenta que no
% deben contarse elementos que sean unificables con TC, sólo
% aquellos que sean idénticos. Por ejemplo, la consulta
% “?- cuenta_const(a, [a,[a,p(a),b],X],N).” debe ser cierta
% con N=1, puesto que sólo una de las a’s que aparecen en L es
% un elemento de L (el resto de a’s están contenidas en el
% segundo elemento de L) y la variable X no debe contarse ya que,
% aunque es unificable con a, no es una a.
```

```
14
15  cuenta_const(TC, L, N) :-
16     atomic(TC),
17     include(==(TC), L, NL),
18     length(NL, N).
```

```
1  % Observe que "cuenta_const" no sería más que una instancia
2  % particular del predicado de orden superior "cuantos"
3  % implementado al final de los apuntes del tema PL3:
4
5  % cuantos(+Obj, +L, ?N)
6  % cierto si N es el número de elementos de la lista L sobre
7  % los que se aplica con éxito el objetivo Obj
8  cuantos(Obj, L, N) :-
9      include(Obj, L, NL),
10     length(NL, N).
11
12 % Si se dispone del código genérico anterior, la implementación
13 % adecuada de "cuenta_const" sería la siguiente:
14
15  cuenta_const_b(TC, L, N) :-
16     atomic(TC),
17     cuantos(==(TC), L, N).
```

```
?- cuenta_const(a, [a,[a,p(a),b],X],N).
```

```
1  % asteriscos(+L)
2  % recibe una lista de números naturales positivos y escribe para cada
3  % uno de ellos una línea con tantos asteriscos como indique el número
4  % correspondiente. Por ejemplo, la consulta “?- asteriscos([2,1,3]).”
5  % debe devolver cierto y debe producir el efecto
6  %  **
7  %  *
8  %  ***
9
10   asteriscos(L) :-
11     maplist(nat_pos, L),
12     maplist(escribe_asteriscos, L).
13
14   nat_pos(N) :-
15     integer(N),
16     N > 0.
17
18   escribe_asteriscos(1) :- write(*), nl.
19   escribe_asteriscos(N) :-
20     N>1,
21     write(*),
22     N1 is N-1,
23     escribe_asteriscos(N1).
```

```
?- asteriscos([2,1,3]).
```

4. Implementación de algunos predicados de orden superior

Implemente los predicados para el manejo de listas en Prolog descritos a continuación, a ser posible de varias formas distintas (recursiva o utilizando predicados de recolección de soluciones u otros predicados de orden superior). Compruebe que sus implementaciones son correctas haciendo consultas para casos significativos.

```
% map(+Obj, +L)
% cierto si todos los elementos de L cumplen la propiedad Obj.
% Este predicado está implementado en los apuntes de forma recursiva.
% Proponga una implementación alternativa basada en el uso de algún
% predicado de recolección.
% PISTA: compruebe que al recolectar todos los elementos
% pertenecientes a L que cumplen Obj se obtiene la propia lista.

  map(Obj, L) :-
    % todos los elementos de L cumplen Obj es equivalente a que la
    % lista resultante de recolectar aquellos elementos de L que
    % cumplen Obj coincida con la lista original.
    findall(X, (member(X,L), call(Obj,X)), L).
 % findall para que devuelva cierto, en lugar de fallar, si L=[]

% Otra opción sería usar el predicado include:
 map_bis(Obj, L) :-
    include(Obj, L, L).
```

```
?- map_bis(integer, [1,2]).
```

```
?- map_bis(integer, [1,2,a]).
```

```
?- map_bis(integer, []).
```

```
% ninguno(+Obj, +L)
% cierto si Obj no se ejecuta con éxito sobre ninguno de los elementos
% de la lista L. Por ejemplo, la consulta "?- ninguno(number, [X,b])."
% daría cierto, mientras que "?- ninguno(number, [X,2])." daría falso.
%
% implementación usando include
  ninguno(Obj, L) :-
    include(Obj, L, []).
    % equivalente a findall(X, (member(X,L), call(Obj,X)), []).

% implementación recursiva
  ninguno_rec(_, []).
  ninguno_rec(Obj, [C|R]) :-
    \+ call(Obj, C),
    ninguno_rec(Obj, R).

% otra posible implementación recursiva
  ninguno_rec_b(_, []).
  ninguno_rec_b(Obj, [C|_]) :-
```

```
20      call(Obj, C),
21      !,
22      fail.
23    ninguno_rec_b(Obj, [_|R]) :-
24      ninguno_rec_b(Obj, R).
```

```
?- ninguno(number, []).
```

```
?- ninguno(number, [X,b]).
```

```
?- ninguno(number, [X,2]).
```

```
1  % alguno(+Obj, +L)
2  % cierto si Obj se ejecuta con éxito sobre al menos uno de los element
3  % de la lista L. Por ejemplo, la consulta "?- alguno(number, [X,b])."
4  % daría falso, mientras que "?- alguno(number, [X,2])." daría cierto.
5  %
6   % implementación recursiva
7    alguno_rec(Obj, [C|_]) :-
8      call(Obj, C),
9      !.
10   alguno_rec(Obj, [_|R]) :-
11     alguno_rec(Obj, R).
12
13   % usando include
14   alguno(Obj, L) :-
15     include(Obj, L, [_|_]).
16     % el resultado de include es una lista NO vacía
17
18   % usando ninguno
19   alguno_b(Obj, L) :-
20     \+ ninguno(Obj, L).
```

```
?- alguno(number, []).
```

```
?- alguno(number, [X,b]).
```

```
?- alguno(number, [X,2]).
```

```
1  % map_parcial(+Obj1, +Obj2, +L, ?NL)
2  % cierto si Obj2 se ejecuta con éxito sobre todas las parejas de
3  % elementos de las listas L y NL situados en la misma posición tales
4  % que Obj1 se ejecuta con éxito sobre el elemento correspondiente de L
5  % Por ejemplo, si se dispone del predicado binario "cuadrado(+X,?Y)",
6  % cierto si Y es el cuadrado del número X, la respuesta a la consulta
7  % "?- map_parcial(integer, cuadrado, [a,2,3.5,b], X)." sería
8  % "X = [a, 4, 3.5, b]".
9  %
10
11  map_parcial(_, _, [], []).
```

```
12 map_parcial(Obj1, Obj2, [C|R], [NC|NR]) :-
13                     call(Obj1,C),
14                     !,
15                     call(Obj2,C,NC),
16                     map_parcial(Obj1, Obj2, R, NR).
17 map_parcial(Obj1, Obj2, [C|R], [C|NR]) :-
18                     map_parcial(Obj1, Obj2, R, NR).
19
20 cuadrado(X, Y) :-
21    Y is X^2.
```

```
?- map_parcial(integer, cuadrado, [a,2,3.5,b], X).
```

```
?- map_parcial(>(10), cuadrado, [1,2,3,12,4], L).
```

```
1 % map_pos(+Obj1, +Obj2, +L, ?NL)
2 % cierto si Obj2 se ejecuta con éxito sobre todas las parejas de
3 % elemento de las listas L y NL situados en la misma posición tales
4 % que Obj1 se  ejecuta con éxito sobre dicha posición, entendiendo
5 % que las posiciones se cuentan empezando por 1. Por ejemplo, si se
6 % dispone de los predicados "par(+X)", cierto si X es un número
7 % par y "cuadrado(+X,?Y)", cierto si Y es el cuadrado del número X,
8 % la respuesta a la consulta  "?- map_pos(par,cuadrado,[1,3,5,7],L)."
9 % sería "L = [1,9,5,49]".
10
11 map_pos(Obj1, Obj2, L, NL) :-
12    map_pos(Obj1, Obj2, L, NL, 1).
13 map_pos(_, _, [], [], _).
14 map_pos(Obj1, Obj2, [C|R], [NC|NR], Pos) :-
15    call(Obj1, Pos),
16    !,
17    call(Obj2, C, NC),
18    NPos is Pos + 1,
19    map_pos(Obj1, Obj2, R, NR, NPos).
20 map_pos(Obj1, Obj2, [C|R], [C|NR], Pos) :-
21    NPos is Pos + 1,
22    map_pos(Obj1, Obj2, R, NR, NPos).
23
24 par(X) :-
25    integer(X),
26    X >= 0,
27    X mod 2 =:= 0.
```

```
?- map_pos(par,cuadrado,[1,3,5,7],L).
```

```
?- map_pos(>(4), cuadrado, [1,2,3,4,5,6], L).
```

Utilice el predicado `map_pos` anterior para ofrecer una implementación alternativa al predicado `map_pares/3` implementado en los apuntes del Tema PL3 (página 29, Ejemplo 26): `map_pares/3` es similar a `map/3` pero tal que el objetivo que recibe como primer parámetro se aplica solo a las posiciones pares de la lista

(suponiendo que la cabeza de la lista está en la posición 1). Por ejemplo, la consulta `map_pares(cuadrado, [1,2,3,4], L)` daría como resultado L = [1, 4, 3, 16].

```
% map_pares(+Obj, +L, ?NL)
% cierto si Obj/2 se aplica con éxito sobre las posiciones pares de L
%
  map_pares(Obj, L, NL) :-
    map_pos(par, Obj, L, NL).
```

```
?- map_pares(cuadrado, [1,2,3,4], L).
```

```
% maximo(+L, ?M)
% cierto si "L" es una lista no vacía de números y "M" es el máximo de
% ellos. Implemente este predicado de tres formas distintas:
% 1) de forma recursiva, 2) usando recolección de soluciones y
% 3) por medio de una operación de plegado (en la que el valor
% inicial sería el primer elemento de la lista).
%
%
 %% implementación recursiva
  maximo_rec([C|R], M) :-
    maplist(number, [C|R]),
    maximo_aux([C|R], M).
  maximo_aux([C], C) :- !.
  maximo_aux([C|R], M) :-
    maximo_aux(R, M),
    M >= C,
    !.
  maximo_aux([C|_], C).

%% inplementación mediante recolección
  maximo_recol(L, M) :-
    maplist(number, L),
    setof(X, member(X,L), LOrd),
    % LOrd es L ordenada de menor a mayor (y sin repeticiones)
    append(_, [M], LOrd).
    % el máximo es el último de LOrd.
```

```
%% implementación por plegado
  maximo([C|R], M) :-
    maplist(number, [C|R]),
    foldl(max2, R, C, M).

  max2(Acc, Ele, Acc) :-
    Acc >= Ele, !.
  max2(_Acc, Ele, Ele).
```

```
?- maximo([],M).
```

```
?- maximo([1],M).
```

```
?- maximo([1,3,2],M).
```

5. Programa para la gestión de prácticas de lectura

Este ejercicio tiene por objetivo implementar un programa en Prolog que sirva de ayuda en la gestión de las prácticas de lectura de un curso de literatura, en el que los alumnos tienen que hacer un comentario de texto sobre el libro que se les asigne. La profesora asigna los libros teniendo en cuenta su propia lista de sugerencias, las preferencias indicadas por los alumnos, los fondos de la biblioteca, y las siguientes restricciones:

- Todos los alumnos que han comunicado sus preferencias (y sólo ellos) deben tener asignado un libro, y nada más que uno.
- A un alumno no se le puede asignar un libro sobre el que no haya mostrado interés.
- Si un alumno muestra interés por un libro que no pertenece a la lista de sugerencias de la profesora, dicho libro no será tenido en cuenta en el proceso de asignación.
- Los libros asignados a los alumnos no deben estar ya prestados en la biblioteca (para simplificar se supondrá que la biblioteca dispone de un único ejemplar de cada libro).
- Un mismo libro no puede ser asignado a dos alumnos diferentes.

Además de la asignación de libros de acuerdo con las restricciones anteriores, la profesora también está interesada en conocer los siguientes datos:

- El porcentaje de alumnos, sobre matriculados, que están siguiendo el curso (es decir, los que expresan sus preferencias de lectura).
- La cantidad media de libros indicada por aquellos alumnos que facilitan sus preferencias.

Ejemplo. Suponga que se parte de la siguiente información:

- Hay 5 alumnos matriculados en la asignatura: a1, a2, ... a5.
- Las sugerencias de lectura de la profesora son los libros l1,l2 ... l10.
- Los alumnos que han comunicado sus preferencias de lectura son los siguientes:
  - A la alumna "a1" le gustaría realizar el comentario sobre cualquiera de los tres libros l1, l2 o l3.
  - El alumno "a2" sólo quiere comentar los libros l2, l4 o l11.
  - A la alumna "a3" le gustaría leer l2, l3 o l20.

Dados estos datos, y sabiendo que el libro l1 está prestado, existen dos posibles soluciones que satisfacen las restricciones estipuladas. A continuación se representan dichas soluciones mediante una secuencia de pares (A,L), donde A es el alumno y L es el libro que se le asigna:

- Solución 1: (a1, l2), (a2, l4), (a3, l3)
- Solución 2: (a1, l3), (a2, l4), (a3, l2)

Además, el porcentaje de alumnos que sigue la asignatura es el 60\% (3 de 5), y la media de libros indicados por los alumnos que expresan sus preferencias es igual a 3 ((3+3+3)/3).

5.1. Representación de los datos

La información relativa a los alumnos del curso, los libros sugeridos por la profesora, las preferencias manifestadas por los alumnos y los libros de la biblioteca que ya están prestados deberá representarse mediante los predicados que se consideren oportunos:

```
1 %% Datos del programa
2 %
3 % alumnos(?Alumnos)
4 % cierto si Alumnos es la lista de los alumnos matriculados en el curs
5   alumnos([a1,a2,a3,a4,a5]).
6
7 % libros(?Libros)
```

```
% cierto si Libros es la lista de libros sugeridos por la profesora
  libros([l1,l2,l3,l4,l5,l6,l7,l8,l9,l10]).


% preferencias(?A, ?L)
% cierto si L es la lista de preferencias del alumno A
  preferencias(a1, [l1,l2,l3]).
  preferencias(a2, [l2,l4,l11]).
  preferencias(a3, [l2,l3,l20]).

% prestados(?L)
% cierto si L es la lista de libros no disponibles en la biblioteca
  prestados([l1]).
```

5.2. Resolución del problema

El problema de la asignación de libros se debe resolver implementando un programa en Prolog conteniendo el siguiente predicado:

```
% solucion(-ListaAsignaciones, -Porcentaje, -Media)
% Cierto si 'ListaAsignaciones' es una lista con todas las soluciones
% a un problema de asignación de libros descrito mediante los
% predicados adecuados, y 'Porcentaje' y 'Media' son los valores
% definidos más arriba.
%
  solucion(ListaAsignaciones, Porcentaje, Media) :-
     % lista total de alumnos
     alumnos(Alumnos),
     % lista de alumnos interesados
     findall(A, preferencias(A, _), AlumnosInteresados),
     % lista de libros posibles
     libros(Libros),
     % lista de libros prestados
     prestados(Prestados),
     findall(A,
        asigna(AlumnosInteresados, Libros, Prestados, A),
        ListaAsignaciones),
     porcentaje(Alumnos, AlumnosInteresados, Porcentaje),
     media(AlumnosInteresados, Media).
```

```
asigna([], _, _, []).
  asigna([A|RAls], Libros, Prestados, [(A, Libro)|RAsig]) :-
       % asigna libros al resto de alumnos
       asigna(RAls, Libros, Prestados, RAsig),
       % busca las preferencias del alumno A y elige uno de ellos
       preferencias(A, Preferidos),
       member(Libro, Preferidos),
       % comprueba que la asignación de ese libro es posible:
       member(Libro, Libros),         % está entre las sugerencias
       \+ member(Libro, Prestados),   % no está prestado
       \+ member((_, Libro), RAsig).  % no está ya asignado

```

```
13 porcentaje(Alumnos, AlumnosInteresados, Porcentaje) :-
14       length(Alumnos, NumAlumnos),
15       length(AlumnosInteresados, NumInteredados),
16       Porcentaje is  100*NumInteredados/NumAlumnos.
17
18  media(AlumnosInteresados, Media) :-
19       length(AlumnosInteresados, NumA),
20       findall(Lpref, preferencias(_,Lpref), L),
21       maplist(length, L, LLongitudes),
22       sumlist(LLongitudes, Suma),
23       Media is Suma/NumA.
```

```
?- solucion(L, P, M).
```