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.

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº1: Primeros contactos con el lenguaje

Autora: A. Pradera

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 con lógica "pura"

Considere el predicado lógico "puro" (implementado sin usar la aritmética del lenguaje) 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 ejecutar 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, uno por uno, en el código dado más arriba y haciendo cada vez 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: ...

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

consulta 3: ...

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

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

consulta 4: ...

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

consulta 5: ...

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

consulta 6: ...

```
?- 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 s(s(0)) (es decir, dos)? ¿Cuál(es)?

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

consulta 8: ¿existe algún natural tal que sumado consigo mismo da s(s(s(0))) (es decir, tres)? ¿Cuál(es)?

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

consulta 9: ¿Hay naturales menores o iguales que s(s(s(0)))? ¿Cuál(es)?
Pista: X es menor o igual que s(s(s(0))) si existe un natural Z tal que X+Z=s(s(s(0))).

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

consulta 10: ¿Hay naturales estrictamente menores que s(s(s(0)))? ¿Cuál(es)?
Pista: X es estrictamente menor que s(s(s(0))) si existe un natural Z *no nulo* tal que X+Z=s(s(s(0))).

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

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 %  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

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

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

```
1 % menorig(?X, ?Y)
2 % cierto si X e Y son naturales y X es menor o igual que Y
3 % (predicado lógico puro, sin usar la aritmética de Prolog)
4 %
```

```
5 %  INTRODUZCA AQUÍ SU CÓDIGO
```

Pruebe su código realizando consultas significativas:

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

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

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

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.

```
1  % PREDICADOS BÁSICOS
2
3  % progenitor(?X, ?Y)
4  % Cierto si X es progenitor (madre o padre) de Y
5  progenitor(pepa, pepe1).
6  progenitor(pepe, pepe1).
7  progenitor(pepe, pepa2).
8  progenitor(pepe1, pepa11).
9  progenitor(pepe1, pepa12).
10 progenitor(pepa12, pepe121).
11
12 % mujer(?X)
13 % Cierto si X es una mujer
14 mujer(pepa).
15 mujer(pepa2).
16 mujer(pepa11).
17 mujer(pepa12).
18
19 % varon(?X)
20 % Cierto si X es un varón
21 varon(pepe).
22 varon(pepe1).
23 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?

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

¿Quiénes son los progenitores de pepe1?

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

¿Quién es la madre de pepe1?

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

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

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

¿Quiénes son mujeres?

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

¿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 *semi-anónimas* para resolver este tipo de situaciones.

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

¿Quiénes son progenitores comunes de pepa11 y pepa12?

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

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

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

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. Por ejemplo, una forma, INADECUADA, de definirlo sería la siguiente:

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

Esta implementación es correcta respecto a la definición actual de los predicados "progenitor" y "mujer", pero podría dejar de serlo si estos cambian. Una forma adecuada de implementar el predicado "madre" es hacerlo mediante una *regla* que exprese su definición en el caso general: para cualesquiera dos personas X e Y, X será la madre de Y si resulta que X es una mujer y es además progenitora de Y:

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

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 %  INTRODUZCA AQUÍ SU CÓDIGO
```

Pruebe ahora su código realizando consultas significativas:

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

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

`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 %  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

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

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

```
1 % hija(?X,?Y), cierto si X es una hija de Y.
2
3 %  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

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

```
% abuelo(?X,?Y), cierto si X es abuelo (varón) de Y.
%
%  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

`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.
%
%  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

`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.
%
%  INTRODUZCA AQUÍ SU CÓDIGO
```

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

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

`ancestro(?X,?Y), cierto si X es un ancestro (mujer o varón) 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.
%
%  INTRODUZCA AQUÍ SU CÓDIGO
%
```

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

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

`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 utilizando 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.

```
1 % pariente(?X,?Y), cierto si X es un pariente (mujer o varón) de Y.
2 %
3 %  INTRODUZCA AQUÍ SU CÓDIGO
```

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, será 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)).
```

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

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

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

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

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

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

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

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

Observe que la respuesta a la consulta anterior incluye `X=Z` en lugar de `X=a` como daría el algoritmo de unificación estudiado en clase, en el que al *componer* la sustitución `{X=Z}` con `{Z=a}` se obtiene `X=a`. Aunque SWI Prolog no siempre hace explícita esta composición de sustituciones, internamente sí que la está haciendo, como demuestra la consulta siguiente.

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

En efecto, la respuesta a la consulta anterior es negativa, porque cuando Prolog llega a `X=d` lo que realmente comprueba es `a=d` (porque previamente ha sustituido la X por a).

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

```
?- 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)).
```

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:

```
1  % progenitor_a(?X, ?Y): cierto si X es progenitor/a de Y
2  progenitor_a(pepa, pepito).
3  progenitor_a(pepito, pepon).
4
5  % ancestro(?X, ?Y): cierto si X es un ancestro de Y
6  % VERSIÓN 1: caso base en primer lugar, caso recursivo con
7  % recursión a la derecha (recursión final o recursión de cola)
8  ancestro1(X, Y) :-
9      progenitor_a(X, Y).
10 ancestro1(X, Y) :-
11     progenitor_a(X, Z),
12     ancestro1(Z, Y).
13
14 % VERSIÓN 2: caso base en último lugar, caso recursivo con
15 % recursión a la derecha (recursión final o recursión de cola)
16 ancestro2(X, Y) :-
17     progenitor_a(X, Z),
18     ancestro2(Z, Y).
19 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

Autora: A. Pradera

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*, unifica con el resultado de evaluar Y. Si Y no es una expresión aritmética o no puede evaluarla, Prolog produce un error. Si X, *tal cual está*, 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.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

```
% factorial(+X, ?Y)
% cierto si Y es el factorial del número natural X.

factorial(0, 1).

factorial(X, Y) :-
    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: ciertos 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.

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.

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(1, F).
```

¿Qué cree que ocurriría ante las mismas consultas si se suprimiese la línea `X > 0` del código anterior? ¿Por qué? Compruébelo comentando dicha línea y ejecutando de nuevo las dos consultas anteriores.

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
 %
```

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

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

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

Consulta 2: ¿Qué paises existen con menos de 50 millones de habitantes, área menor que 1 millón de km cuadrados y densidad menor que 100?

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

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

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

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.

```
% natural(+X)
% cierto si X es un número natural
% (puede usar el predicado predefinido integer(+X))
%
```

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

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

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

```
% fib(+N, ?F)
% cierto si F es el número de Fibonacci asociado con N.
% Recuerde que cada número natural n tiene asociado otro número
% natural f(n), denominado número de Fibonacci, que se calcula
% como sigue: f(0)= f(1)=1 ; f(n) = f(n-1)+f(n-2) si n>1.
%
```

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

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

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

```
% 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).
%
```

```
?- 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}
%
```

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

```
?- 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.
%
```

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

```
?- 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.

```
% eval(+E, ?V)
% cierto si E es una expresión aritmética y V es su valor

eval(N, N) :-
    number(N).

eval(A+B, V) :-
    eval(A, VA), eval(B, VB), V is VA + VB.

%% ESCRIBA A CONTINUACIÓN EL RESTO DEL CÓDIGO
```

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

```
?- 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:

```
% cubo
% predicado sin argumentos que solicita un número y escribe
% el cubo de dicho número, realizando el proceso anterior de
% forma reiterada hasta que el usuario introduce la palabra fin.



% 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.
%
```

```
?- cubo.
```

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

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

PRÁCTICA DE PROLOG Nº3: Listas y predicado de corte

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),
    !,
    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).
```

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.
```

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).
```

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

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

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

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

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

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

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

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

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

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

Para las siguientes consultas tenga en cuenta que el predicado predefinido `sumlist(L,S)` es cierto cuando L es una lista de números y S es su suma (S vale 0 cuando L es la lista vacía).

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

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

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

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

En las dos siguientes consultas `p` y `q` solo son nombres de términos compuestos cuyo único objeto es unificar sus argumentos.

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

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

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.

```
% enesimo(?L, +N, ?E)
% cierto si E es el elemento de la lista L situado
% en la posición N, entendiendo que la cabeza de la
% lista está situada en la posición 1 (debe fallar en
% cualquier otro caso).
%
% Proporcione una implementación RECURSIVA.
%
```

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

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

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

```
% enesimo(?L, +N, ?E)
% cierto si E es el elemento de la lista L situado
% en la posición N, entendiendo que la cabeza de la
% lista está situada en la posición 1.
%
% Proporcione una implementación NO RECURSIVA, basada en el uso
% de algunos de los predicados predefinidos para el manejo de listas.
% PISTA: si usa append para descomponer la lista L en dos trozos
% tales que el primer trozo tenga tamaño N-1, el primer elemento del
% segundo trozo será el N-ésimo de L.
%
```

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

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

```
% 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 situada
% en la posición 1. Si N es mayor que la longitud de
% L, NL será igual a L.
%
% Proporcione una implementación RECURSIVA.
%
```

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

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

```
% 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 situada
% 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.
%
```

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

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

```
1 % elimina_n_bis(?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
5 % en la posición 1. **El predicado debe fallar si N es mayor
6 % que la longitud de L.**
7 %
8 % Proporcione una implementación RECURSIVA.
9 %
```

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

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

```
1 % elimina_n_bbis(?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
5 % en la posición 1. El predicado debe fallar si N es mayor
6 % que la longitud de L.
7 %
8 % Proporcione una implementación NO RECURSIVA, basada en el uso
9 % de algunos de los predicados predefinidos para el manejo de listas.
10 % PISTA: descomponga la lista en dos trozos adecuados y vuelva
11 % a componerlos obviando el elemento n-ésimo.
12 %
```

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

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

```
1 % borrartodos(+L,+E,?NL)
2 % cierto si NL es la lista resultante después de
3 % eliminar de L todos los elementos idénticos a E
4 % (si los hay). Tenga en cuenta que no deben eliminarse elementos
5 % que unificables con E, sólo aquellos que sean idénticos. Por
6 % ejemplo, borrartodos([a,b],X,[a,b]) y borrartodos([a,X],a,[X])
7 % deben ser ciertos.
8 %
```

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

Create a Program | Query | Markdown | HTML **cell here**

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

```
1 % insertardetras(+L,+E1,+E2,?NL)
```

```
% cierto si NL es la lista resultante de insertar
% E2 justo detrás de cada aparición de E1 en L
%
```

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

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

```
% 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


% VERSIÓN 2: Utilizando el predicado de corte
```

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

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

```
% 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.
%
```

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

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

```
% aplana(+L,?NL)
% cierto si L es una lista y NL es la lista formada por todos los
% términos que aparecen en L, quitando las listas internas de L en
% 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.
%
```

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

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

```
% 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].
%
```

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

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

```
% 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.
%
```

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

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

```
% 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
%  **
%  *
%  ***
```

PROGRAMACIÓN LÓGICA

3º GRADO EN INGENIERÍA INFORMÁTICA, URJC

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

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, 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).

```

```
% 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.
```

```
8  ancestro(X,Y) :-
9      progenitor(X,Y).
10 ancestro(X,Y) :-
11     progenitor(X,Z),
12     ancestro(Z,Y).
13
```

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

```
1 % num_descendientes(+X,?N)
2 % cierto si X tiene N descendientes (N puede ser 0).
3 %
```

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

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

```
1 % ancestros(+X, ?L)
2 % cierto si L es una lista conteniendo todos los
3 % ancestros de X. Debe fallar si X no tiene ancestros.
4 %
```

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

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

```
1 % listas_descencientes(?L)
2 % cierto si L es una lista de terminos desc(X,DX) donde
3 % X es una persona con descendientes y DX es una lista
4 % con sus descendientes.
5 %
```

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

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

```
1 % desc_no_directos(+X, ?L)
2 % cierto si L es una lista (puede ser vacía) conteniendo
3 % a todos los descendientes de X salvo sus hijas/os
4 %
```

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

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

```
1 % ascendientes_varones_hasta_2(+X,?L)
2 % cierto si L contiene todos los ascendientes varones
3 % de X de grado máximo 2 (es decir, padre y abuelos).
```

```
4 % Debe fallar si no hay ninguno
5 %
```

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

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

```
1 % con_descendientes(-L, -N)
2 % cierto si L es el conjunto de personas que tiene algún
3 % descendiente y N es el número de esas personas
4 % (si no hubiese ninguna, debe devolver lista vacía y cero).
5 %
```

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

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:

```
1 % progenitores(?L1, ?L2)
2 % cierto si L1 y L2 son listas tales que cada persona de L1 es
3 % progenitor/a de la persona situada en la misma posición de L2.
4 % Una de las dos listas debe ser de entrada.
```

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

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

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

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
2 %% bloques
3 %
```

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.

```
1 % peso_medio(+B, ?M)
2 % cierto si M es el peso medio de los bloques que están
3 % apilados sobre B pero que no están justo encima suyo.
4 %
```

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

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

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

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)]

```
1 % pesos(-L)
2 % cierto si L es la lista de tuplas (B,P,PA) donde B es un
3 % bloque sobre el que hay apilado al menos otro bloque, P es el
4 % peso de B y PA es el peso que soporta B.
```

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]).
```

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

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

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

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

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

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

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

```
?- include((apilado(d),\+ 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) 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).
```

```
?- 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).
```

```
?- 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),
   !.
```

```
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).
```

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.

```
1 % borrartodos(+L,+E,?NL)
2 % cierto si NL es la lista resultante después de
3 % eliminar de L todas las ocurrencias de E
4 % (si las hay)
5 %
```

```
% 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.
%
```

```
% 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.
%
```

```
% 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
%  **
%  *
%  ***
```

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 pertenecient
% a L que cumplen Obj se obtiene la propia lista.
```

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

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

```
% 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
%
```

?- Your query goes here ...

?- Your query goes here ...

?- Your query goes here ...

```
% alguno(+Obj, +L)
% cierto si Obj se ejecuta con éxito sobre al menos uno de los element
% de la lista L. Por ejemplo, la consulta "?- alguno(number, [X,b])."
% daría falso, mientras que "?- alguno(number, [X,2])." daría cierto.
%
```

?- Your query goes here ...

?- Your query goes here ...

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

?- Your query goes here ...

?- Your query goes here ...

```
% map_pos(+Obj1, +Obj2, +L, ?NL)
% cierto si Obj2 se ejecuta con éxito sobre todas las parejas de
% elementos de las listas L y NL situados en la misma posición tales
% que Obj1 se ejecuta con éxito sobre dicha posición, entendiendo que
% las posiciones se cuentan empezando por 1 (la cabeza de una lista
% está en la posición 1). Por ejemplo, si se dispone de los predicados
% "par(+X)", cierto si X es un número par y "cuadrado(+X,?Y)",
% cierto si Y es el cuadrado del número X, la respuesta a la consulta
% "?- map_pos(par,cuadrado,[1,3,5,7],L)." sería "L = [1,9,5,49]".
%
```

?- Your query goes here ...

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

Utilice el predicado `map_pos` anterior para ofrecer una implementación alternativa al predicado `map_pares/3` implementado en los apuntes del Tema PL3: `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 se aplica con éxito sobre las posiciones pares
% de L y NL
%
```

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

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

```
% 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).
%
```

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

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

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 %
```

5.2. Resolución del problema

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

```
1 % solucion(-ListaAsignaciones, -Porcentaje, -Media)
2 % Cierto si 'ListaAsignaciones' es una lista con todas las soluciones
3 % un problema de asignación de libros descrito mediante los predicados
4 % adecuados, y 'Porcentaje' y 'Media' son los valores definidos más ar
5 %
6 %
```

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