

Práctica N° 7 - Programación lógica

Para resolver esta práctica, recomendamos utilizar el programa “SWI-Prolog”, de distribución gratuita, que puede bajarse de <http://www.swi-prolog.org>. El único *meta predicado* que puede utilizarse para resolver los ejercicios es `not`. No utilizar *cut* (!) ni predicados de *alto orden* (como `setof`). Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

EL MOTOR DE BÚSQUEDA DE PROLOG

Ejercicio 1 ★

Considerar la siguiente base de conocimiento.

```
padre(juan, carlos).           padre(luis, pablo).
padre(juan, luis).            padre(luis, manuel).
padre(carlos, daniel).        padre(luis, ramiro).
padre(carlos, diego).         abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
```

- I. ¿Cuál el resultado de la consulta `abuelo(X, manuel)`?
- II. A partir del predicado binario `padre`, definir en Prolog los predicados binarios: `hijo`, `hermano` y `descendiente`.
- III. Dibujar el árbol de búsqueda de Prolog para la consulta `descendiente(Alguien, juan)`.
- IV. ¿Qué consulta habría que hacer para encontrar a los nietos de `juan`?
- V. ¿Cómo se puede definir una consulta para conocer a todos los hermanos de `pablo`?
- VI. Considerar el agregado del siguiente hecho y regla:

```
ancestro(X, X).
ancestro(X, Y) :- ancestro(Z, Y), padre(X, Z).
```

y la base de conocimiento del ítem anterior.

- VII. Explicar la respuesta a la consulta `ancestro(juan, X)`. ¿Qué sucede si se pide más de un resultado?
- VIII. Sugerir una solución al problema hallado en los puntos anteriores reescribiendo el programa de `ancestro`.

Ejercicio 2

Sea el siguiente programa lógico:

```
vecino(X, Y, [X|[Y|Ls]]).
vecino(X, Y, [W|Ls]) :- vecino(X, Y, Ls).
```

- I. Mostrar el árbol de búsqueda en Prolog para resolver `vecino(5, Y, [5,6,5,3])`, devolviendo todos los valores de `Y` que hacen que la meta se deduzca lógicamente del programa.
- II. Si se invierte el orden de las reglas, ¿los resultados son los mismos? ¿Y el orden de los resultados?

Ejercicio 3 ★

Considerar las siguientes definiciones:

```
natural(0).                               menorOIgual(X, suc(Y)) :- menorOIgual(X, Y).
natural(suc(X)) :- natural(X).            menorOIgual(X,X) :- natural(X).
```

- I. Explicar qué sucede al realizar la consulta `menorOIgual(0,X)`.
- II. Describir las circunstancias en las que puede ocurrir un ciclo infinito en Prolog.

III. Corregir la definición de `menorOIgual` para que funcione adecuadamente.

ESTRUCTURAS, INSTANCIACIÓN Y REVERSIBILIDAD

Ejercicio 4 ★

Definir el predicado `concatenar(?Lista1,?Lista2,?Lista3)`, que tiene éxito si `Lista3` es la concatenación de `Lista1` y `Lista2`. Por ejemplo:

```
?- concatenar([a,b,c], [d,e], [a,b,c,d,e]).    → true.
?- concatenar([a,b,c], [d,e], L).              → L = [a,b,c,d,e].
?- concatenar([a,b,c], L, [a,b,c,d,e]).        → L = [d,e].
?- concatenar(L, [d,e], [a,b,c,d,e]).          → L = [a,b,c].
?- concatenar(L1, L2, [1,2,3]).                → L1 = [], L2 = [1, 2, 3];
                                                L1 = [1], L2 = [2, 3];
                                                L1 = [1,2], L2 = [3];
                                                L1 = [1,2,3], L2 = [].
```

Al igual que la mayoría de los predicados, puede dar **false** después de agotar los resultados.

Nota: este predicado ya está definido en prolog con el nombre `append`.

Ejercicio 5 ★

Definir los siguientes predicados sobre listas:

- I. `last(?L, ?U)`, donde `U` es el último elemento de la lista `L`.
- II. `reverse(+L, -L1)`, donde `L1` contiene los mismos elementos que `L`, pero en orden inverso.
Ejemplo: `reverse([a,b,c], [c,b,a])`.
Mostrar el árbol de búsqueda para el ejemplo dado.
- III. `máxlista(+L, -M)` y `mínlista(+L, -M)`, donde `M` es el máximo/mínimo de la lista `L`.
- IV. `prefijo(?P, +L)`, donde `P` es prefijo de la lista `L`.
- V. `sufijo(?S, +L)`, donde `S` es sufijo de la lista `L`.
- VI. `sublista(?S, +L)`, donde `S` es sublista de `L`.
- VII. `pertenece(?X, +L)`, que es verdadero sii el elemento `X` se encuentra en la lista `L`. (Este predicado ya viene definido en Prolog y se llama `member`).

Ejercicio 6 ★

Definir el predicado `aplanar(+Xs, -Ys)`, que es verdadero sii `Ys` contiene los elementos de todos los niveles de `Xs`, en el mismo orden de aparición. Los elementos de `Xs` son enteros, átomos o nuevamente listas, de modo que `Xs` puede tener una profundidad arbitraria. Por el contrario, `Ys` es una lista de un solo nivel de profundidad.

Ejemplos:

```
?- aplanar([a, [3, b, []], [2]], L). → L=[a, 3, b, 2]
?- aplanar([[1, [2, 3], [a]], [[]]], L). → L=[1, 2, 3, a]
```

Ejercicio 7

Definir los siguientes predicados:

- I. `palíndromo(+L, -L1)`, donde `L1` es un palíndromo construido a partir de `L`.
Ejemplo: `palíndromo([a,b,c], [a,b,c,c,b,a])`.
- II. `doble(?L, ?L1)`, donde cada elemento de `L` aparece dos veces seguidas en `L1`.
Ejemplo: `doble([a,b,c], [a,a,b,b,c,c])`.
- III. `iésimo(?I, +L, -X)`, donde `X` es el `I`-ésimo elemento de la lista `L`.
Ejemplo: `iésimo(2, [10, 20, 30, 40], 20)`.

Ejercicio 8 ★

Considerar el siguiente predicado:

```
desde(X,X).  
desde(X,Y) :- N is X+1, desde(N,Y).
```

- I. ¿Cómo deben instanciarse los parámetros para que el predicado funcione? (Es decir, para que no se cuelgue ni produzca un error). ¿Por qué?
- II. Modificar el predicado para que funcione con la instanciación `desde(+X,?Y)`, de manera que, si `Y` está instanciada, sea verdadero si `Y` es mayor o igual que `X`, y si no lo está genere todos los `Y` de `X` en adelante.

Ejercicio 9 ★

Definir los siguientes predicados:

- I. `intersección(+L1, +L2, -L3)`, tal que `L3` es la intersección sin repeticiones de las listas `L1` y `L2`, respetando en `L3` el orden en que aparecen los elementos en `L1`.
- II. `split(N, L, L1, L2)`, donde `L1` tiene los `N` primeros elementos de `L`, y `L2` el resto. Si `L` tiene menos de `N` elementos el predicado debe fallar. ¿Cuán reversible es este predicado? Es decir, ¿qué parámetros pueden estar indefinidos al momento de la invocación?
- III. `borrar(+ListaOriginal, +X, -ListaSinXs)`, que elimina todas las ocurrencias de `X` de la lista `ListaOriginal`.
- IV. `sacarDuplicados(+L1, -L2)`, que saca todos los elementos duplicados de la lista `L1`.
- V. `reparto(+L, +N, -LListas)` que tenga éxito si `LListas` es una lista de `N` listas ($N \geq 1$) de cualquier longitud - incluso vacías - tales que al concatenarlas se obtiene la lista `L`.
- VI. `repartoSinVacías(+L, -LListas)` similar al anterior, pero ninguna de las listas de `LListas` puede ser vacía, y la longitud de `LListas` puede variar.

Ejercicio 10

Definir el predicado `intercalar(L1, L2, L3)`, donde `L3` es el resultado de intercalar uno a uno los elementos de las listas `L1` y `L2`. Si una lista tiene longitud menor, entonces el resto de la lista más larga es pasado sin cambiar. Indicar la reversibilidad, es decir si es posible obtener `L3` a partir de `L1` y `L2`, y viceversa.
Ejemplo: `intercalar([a,b,c], [d,e], [a,d,b,e,c])`.

Ejercicio 11 ★

Un árbol binario se representará en Prolog con:

- `nil`, si es vacío.
- `bin(izq, v, der)`, donde `v` es el valor del nodo, `izq` es el subárbol izquierdo y `der` es el subárbol derecho.

Definir predicados en Prolog para las siguientes operaciones: `vacío`, `raiz`, `altura` y `cantidadDeNodos`. Asumir siempre que el árbol está instanciado.

Ejercicio 12 ★

Definir los siguientes predicados, utilizando la representación de árbol binario definida en el ejercicio 11:

- I. `inorder(+AB,-Lista)`, que tenga éxito si `AB` es un árbol binario y `Lista` la lista de sus nodos segun el recorrido *inorder*.
- II. `arbolConInorder(+Lista,-AB)`, versión inversa del predicado anterior.
- III. `aBB(+T)`, que será verdadero si `T` es un árbol binario de búsqueda.

IV. `aBBInsertar(+X, +T1, -T2)`, donde T2 resulta de insertar X en orden en el árbol T1.

GENERATE & TEST

Ejercicio 13 ★

Definir el predicado `coprimos(-X, -Y)`, que genere uno a uno *todos* los pares de números naturales coprimos (es decir, cuyo máximo común divisor es 1), sin repetir resultados. Usar la función `gcd` del motor aritmético.

Ejercicio 14 ★

Un cuadrado semi-latino es una matriz cuadrada de naturales (incluido el cero) donde todas las filas de la matriz suman lo mismo. Por ejemplo:

```
1 3 0
2 2 0      todas las filas suman 4
1 1 2
```

Representamos la matriz como una lista de filas, donde cada fila es una lista de naturales. El ejemplo anterior se representaría de la siguiente manera: `[[1,3,0],[2,2,0],[1,1,2]]`.

Se pide definir el predicado `cuadradoSemiLatino(+N, -XS)`. El predicado debe ir devolviendo matrices (utilizando la representación antes mencionada), que sean cuadrados *semi-latino*s de dimensión `N*N`. Dichas matrices deben devolverse de manera ordenada: primero aquellas cuyas filas suman 0, luego 1, luego 2, etc..

Ejemplo: `cuadradoSemiLatino(2,X)`. devuelve:

```
X = [[0, 0], [0, 0]] ;    X = [[0, 1], [1, 0]] ;    X = [[1, 0], [1, 0]] ;    etc.
X = [[0, 1], [0, 1]] ;    X = [[1, 0], [0, 1]] ;    X = [[0, 2], [0, 2]] ;
```

Ejercicio 15

En este ejercicio trabajaremos con triángulos. La expresión `tri(A,B,C)` denotará el triángulo cuyos lados tienen longitudes A, B y C respectivamente. Se asume que las longitudes de los lados son siempre números naturales.

Implementar los siguientes predicados:

- I. `esTriángulo(+T)` que, dada una estructura de la forma `tri(A,B,C)`, indique si es un triángulo válido. En un triángulo válido, cada lado es menor que la suma de los otros dos, y mayor que su diferencia (y obviamente mayor que 0).

Sugerencia: para evitar repetir código, escriba un predicado auxiliar `esCompatible(+A,+B,+C)`, que verifique que el lado A cumpla las condiciones necesarias en relación a B y C. Opcionalmente puede ser `esCompatible(?A,+B,+C)`.

- II. `perímetro(?T,?P)`, que es verdadero cuando T es un triángulo (válido) y P es su perímetro. No se deben generar resultados repetidos (no tendremos en cuenta la congruencia entre triángulos: si dos triángulos tienen las mismas longitudes, pero en diferente orden, se considerarán diferentes entre sí). El predicado debe funcionar para cualquier instanciación de T y P (ambas instanciadas, ambas sin instanciar, una instanciada y una no; no es necesario que funcione para triángulos parcialmente instanciados), debe generar todos los resultados válidos (sean finitos o infinitos), y no debe colgarse (es decir, no debe seguir ejecutando infinitamente sin producir nuevos resultados). Por ejemplo:


```
?- perímetro(tri(3,4,5),12).    → true.
?- perímetro(T,5).              → T = tri(1, 2, 2) ; T = tri(2, 1, 2) ; T = tri(2, 2, 1) ; false.
?- perímetro(tri(2,2,2),P).     → P = 6.
?- perímetro(T,P).              → T = tri(1, 1, 1), P = 3 ; T = tri(1, 2, 2), P = 5 ; ...
```

- III. `triángulo(-T)`, que genera todos los triángulos válidos, sin repetir resultados.

NEGACIÓN POR FALLA

Ejercicio 16

Definir el predicado `diferenciaSimétrica(Lista1, +Lista2, -Lista3)`, que tenga éxito si Lista3 es la lista de todos los elementos que, o bien están en Lista1 pero no en Lista2, o bien están en Lista2 pero no en Lista1. Asumir que las listas no tienen elementos repetidos.

Ejercicio 17 ★

- I. Sean los predicados $P(?X)$ y $Q(?X)$, ¿qué significa la respuesta a la siguiente consulta?
 $?- P(Y), \text{not}(Q(Y)).$
- II. ¿Qué pasaría si se invirtiera el orden de los literales en la consulta anterior?
- III. Sea el predicado $P(?X)$, ¿Cómo se puede usar el **not** para determinar si existe una única Y tal que $P(?Y)$ es verdadero?

Ejercicio 18 ★

Definir el predicado `corteMásParejo(+L,-L1,-L2)` que, dada una lista de números, realiza el corte más parejo posible con respecto a la suma de sus elementos (puede haber más de un resultado). Por ejemplo:

```
?- corteMásParejo([1,2,3,4,2],L1,L2).    → L1 = [1, 2, 3], L2 = [4, 2] ; false.
?- corteMásParejo([1,2,1],L1,L2).        → L1 = [1], L2 = [2, 1] ; L1 = [1, 2], L2 = [1] ; false.
```

Ejercicio 19

Dado un predicado unario P sobre números naturales, definir un predicado que determine el máximo X que satisfaga $P(X)$.

EJERCICIOS INTEGRADORES (OPCIONALES)

Ejercicio 20

En este ejercicio trabajaremos con grafos no orientados. Un grafo no orientado es un conjunto de nodos y un conjunto de ejes sin una dirección específica. Cada eje está representado por un par de nodos y, como se puede viajar en cualquiera de los dos sentidos, el eje (a, b) y el eje (b, a) son el mismo.

No sabemos cuál es la representación interna de un grafo, pero contamos con un predicado `esNodo(+G,?X)` que dado un grafo G dice si X es nodo de G . También tenemos otro predicado `esArista(+G,?X,?Y)` que dice si en G hay un eje de X a Y . Notar que podemos usar `esNodo` para enumerar los nodos del grafo y `esArista` para enumerar los ejes. Instanciando apropiadamente, también podemos usar `esArista` para obtener todos los ejes que tienen a un nodo particular. Cuando `esArista` lista todos los ejes, cada eje lo lista una sola vez en una orientación arbitraria de las dos posibles, pero si se pregunta por cualquiera de las dos, responderá que sí. Suponer que dos nodos son el mismo si y solo si unifican.

Ayuda: para algunos items conviene pensar primero en cómo programar el predicado opuesto al que se pide.

- I. Implementar el predicado `caminoSimple(+G,+D,+H,-L)` que dice si L es un camino simple en el grafo G que empieza en D y termina en H . Un camino simple lo representaremos por una lista de nodos **distintos**, tal que para cada par de nodos consecutivos en L existe un eje en G que los conecta. Notar que el primer elemento de L debe ser D y el último H . Cuando L está sin instanciar, el predicado debe ir devolviendo **todos** los caminos simples desde D a H sin repetidos (es decir, hay que tener cuidado con los ciclos y evitar que el predicado se cuelgue).
- II. Un camino L en un grafo G es Hamiltoniano sii L es un camino simple que contiene a todos los nodos G . Implementar el predicado `caminoHamiltoniano(+G,-L)` que dice si L es un camino Hamiltoniano en G . Recordar que **no** se pueden usar predicados de alto orden, salvo el **not** (en particular no se puede utilizar `setof`).
- III. Implementar el predicado `esConexo(+G)` que dado un grafo dice si éste es conexo. Un grafo G es conexo sii no existe un par de nodos en G tal que no hay un camino simple que los una. Notar que con esta definición un grafo de un nodo (y sin ejes) es conexo.
- IV. Implementar el predicado `esEstrella(+G)` que dado un grafo dice si es un grafo estrella. Un grafo es estrella sii es conexo y hay un nodo común a todos sus ejes.

Ejercicio 21

Trabajaremos con árboles binarios, usando `nil` y `bin(AI, V, AD)` para representarlos en Prolog.

- I. Implementar un predicado `arbol(-A)` que genere estructuras de árbol binario, dejando los valores de los nodos sin instanciar. Deben devolverse todos los árboles posibles (es decir, para toda estructura posible, el predicado debe devolverla luego de un número finito de pedidos). No debe devolverse dos veces el mismo árbol.

```
? arbol(A).
A = nil ;
A = bin(nil, _G104, nil) ;
A = bin(nil, _G107, bin(nil, _G117, nil)) ;
...
```

- II. Implementar un predicado `nodosEn(?A, +L)` que es verdadero cuando `A` es un árbol cuyos nodos pertenecen al conjunto conjunto de átomos `L` (representado mediante una lista no vacía, sin orden relevante y sin repetidos). Puede asumirse que el árbol se recibe instanciado en su estructura, pero no necesariamente en sus nodos.
- III. Implementar un predicado `sinRepEn(-A, +L)` que genere todos los árboles cuyos nodos pertenezcan al alfabeto `L` y usando como máximo una vez cada símbolo del mismo. En este caso, no hay infinitos árboles posibles; es importante que el predicado no devuelva soluciones repetidas y que no se quede buscando indefinidamente una vez terminado el espacio de soluciones.

```
? arbol(A), nodosEn(A, [ka, pow]).
A = nil ;
A = bin(nil, ka, nil) ;
A = bin(nil, pow, nil) ;
A = bin(nil, ka, bin(nil, ka, nil)) ;
A = bin(nil, ka, bin(nil, pow, nil)) ;
...

? arbolSinRepEn(A, [ka, pow]).
A = nil ;
A = bin(nil, ka, nil) ; ...
A = bin(nil, ka, bin(nil, pow, nil)) ;
... ;
No.
```

Ejercicio 22

Se desea desarrollar un programa Prolog que asista en la planificación de la cocina hogareña. Asumir que los productos que se encuentran disponibles en la alacena se modelan con el predicado unario `disponible` y el recetario está definido por el predicado ternario `receta` que indica el tiempo de preparación y los ingredientes requeridos para elaborar cada plato. Por ejemplo,

```
disponible(harina).      disponible(tomate).      disponible(sal).      disponible(cebolla).
disponible(levadura).    disponible(huevo).      disponible(prepizza).  disponible(zanahoria).
disponible(mozzarella).  disponible(aceite).    disponible(azúcar).
```

```
receta(pizzaMozzarellaCasera, 30, [harina, levadura, sal, aceite, tomate, mozzarella]).
receta(pizzaFugazzaCasera, 25, [harina, levadura, sal, aceite, cebolla]).
receta(pizzaMozzarellaRapida, 10, [prepizza, tomate, mozzarella]).
receta(pizzaControversial, 15, [prepizza, tomate, mozzarella, ananá]).
receta(ensaladaMixta, 15, [tomate, huevo, aceite, sal, cebolla, zanahoria]).
receta(tortaDeZanahoria, 120, [harina, huevo, azúcar, levadura, zanahoria]).
receta(pastelDeZanahoria, 120, [zanahoria, harina, levadura, azúcar, huevo]).
```

Definir:

- I. `sonSimilares(?Plato1, ?Plato2)` que es verdadero cuando `Plato1` y `Plato2` tienen recetas que requieren el mismo tiempo de preparación y los mismos ingredientes (aunque potencialmente en otro orden). **No utilizar operadores para unificar variables con variables o listas.**
- II. `quéMeFalta(+Ingredientes, +Plato, -Faltante)` que es verdadero cuando `Faltante` contiene la lista de ingredientes que faltan en `Ingredientes` para preparar `Plato`. **Sugerencia:** usar las listas como conjuntos.
- III. `puedoPreparar(?Plato)` que es verdadero cuando `Plato` tiene una receta cuyos ingredientes se encuentran en la alacena.
- IV. `másRapido(-Plato)` que es verdadero cuando `Plato` es la receta más rápida que puede prepararse con los ingredientes disponibles.