Práctica 6
Prolog

Ejercicio 1

Considerar la siguiente base de conocimiento.

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

1. ¿Cuál el resultado de la consulta `abuelo(X, manuel)`?
2. A partir del predicado binario `padre`, definir en Prolog los predicados binarios: `hijo`, `hermano` y `descendiente`.
3. Dibujar el árbol de búsqueda de Prolog para la consulta `descendiente(Alguien, juan)`.
4. ¿Qué consulta habría que hacer para encontrar a los nietos de `juan`?
5. ¿Cómo se puede definir una consulta para conocer a todos los hermanos de `pablo`?
6. 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.
7. Explicar la respuesta a la consulta `ancestro(juan, X)`. ¿Qué sucede si se pide más de un resultado?
8. Sugerir un 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).
```

1. 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.
2. 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).
natural(suc(X)) :- natural(X)

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

1. Explicar qué sucede al realizar la consulta `menorOIgual(0,X).`
2. Describir las circunstancias en las que puede ocurrir un ciclo infinito en Prolog.
3. Corregir la definición de `menorOIgual` para que funcione adecuadamente.

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:

1. `last(?L, ?U)`, donde `U` es el último elemento de la lista `L`.
2. `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.

3. `maxlista(+L, -M)` y `minlista(+L, -M)`, donde `M` es el máximo/mínimo de la lista `L`.
4. `prefijo(?P, +L)`, donde `P` es prefijo de la lista `L`.
5. `sufijo(?S, +L)`, donde `S` es sufijo de la lista `L`.
6. `sublista(?S, +L)`, donde `S` es sublista de `L`.
7. `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:

1. `palindromo(+L, -L1)`, donde `L1` es un palíndromo construido a partir de `L`.
   Ejemplo: `palindromo([a,b,c], [a,b,c,c,b,a])`.
2. `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])`.
3. `iesimo(?I, +L, -X)`, donde `X` es el `I`-ésimo elemento de la lista `L`.
   Ejemplo: `iesimo(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).
```

1. ¿Cómo deben instanciarse los parámetros para que el predicado funcione? (Es decir, para que no se cuelgue ni produzca un error). ¿Por qué?
2. 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:

1. `interseccion(+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`.

2. `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?

3. `borrar(+ListaOriginal, +X, -ListaSinXs)`, que elimina todas las ocurrencias de `X` de la lista
`ListaOriginal`.

4. `sacarDuplicados(+L1, -L2)`, que saca todos los elementos duplicados de la lista `L1`.

5. `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`.

6. `repartoSinVacias(+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

Escribir el predicado `elementosTomadosEnOrden(+L,+N,-Elementos)` que tenga éxito si `L` es una lista, `N` ≥ 0 y `Elementos` es una lista de `N` elementos de `L`, preservando el orden en que aparecen en la lista original.

Por ejemplo, una de las soluciones de `elementosTomadosEnOrden([1,4,0,2,5],3,X)` es `X=[1,4,5]`.

Ejercicio 12

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: `vacio`, `raiz`, `altura` y `cantidadDeNodos`. Asumir siempre que el árbol está instanciado.

Ejercicio 13

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

1. `inorder(+AB,-Lista)`, que tenga éxito si `AB` es un árbol binario y `Lista` la lista de sus nodos según el recorrido *inorder*.

2. `arbolConInorder(+Lista,-AB)`, versión inversa del predicado anterior.

Ejercicio 14

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 15

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

Ejercicio 16

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:

1. `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)`.

2. `perimetro(?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 ; ...
```

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

Ejercicio 17

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 18

Sean los predicados `P(?X)` y `Q(?X)`, ¿qué significa la respuesta a la siguiente consulta?

```
?- P(Y), not(Q(Y)).
```

¿Qué pasaría si se invirtiera el orden de los literales en la consulta anterior?

Ejercicio 19

Definir el predicado `corteMasParejo(+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:

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

Ejercicio 20

Consideremos árboles binarios en Prolog, representados usando `nil` y `bin(AI, V, AD)`.

1. 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)) ;
...
```

2. 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.

```
? 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)) ;
...
```

3. 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.

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