Prolog

es un lenguaje de programación muy sencillo
es un lenguaje de programación declarativo
es un lenguaje de programación lógico

Muy sencillo

Toda la información en Prolog es representada en términos. Hay un solo elemento del lenguaje, que es la cláusula. El mismo tiene la forma:

```
head :- body.
```

Significa que si se cumple el **body** o cuerpo, entonces se cumple el **head** o la cabeza, por lo tanto decimos que se cumple la **regla**
:- es el primer operador que vemos, y que representa el "si" condicional. Se lee de manera que se cumple **head** si **cuerpo**
Si siempre se debe cumplir con la cabeza, entonces podemos omitir el cuerpo. A esto se lo suele denominar **hecho**

```
perro(breton):-true.
perro(breton).
```

Usualmente, en la cátedra vamos a trabajar acompañados por una **base de conocimientos** que no es más que un conjunto de datos con los que trabajamos, expresados en forma de listados de hechos.
Toda base de conocimiento puede ser consultada a través de una consulta. Dará falso cualquier cosa que no pertenezca al conjunto solución

```
?- perro(breton)
```

true

```
?- perro(dalmata)
```

false

Declarativo

Prolog es un lenguaje declarativo. Esto significa que vamos a expresar en que estamos interesados y no en el cómo

```
mujer(flor).
mujer(cami).
mujer(ana).
mujer(maria).

hombre(mati).
hombre(luis).

progenitor(ana, flor).
progenitor(ana, maria).
progenitor(ana, mati).
progenitor(luis, flor).

% Predicado hija/2
hija(Hija, Progenitor) :- mujer(Hija), progenitor(Progenitor, Hija).
```

En este ejemplo hemos creado una regla que dice que Hija es hija de Progenitor si y sólo si Hija es mujer y Progenitor es progenitor de Hija. Tanto mujer/1 como progenitor/2 van a formar parte de nuestra base de conocimiento. Podrían también ser reglas
Para probar que esto funciona podemos ejecutar una **query** o consulta. Las mismas comienzan con el operador ?-

```
?- hija(flor, ana)
```

true

```
?- hija(mati, ana)
```

false

```
?- hija(X, ana)
```

X = flor
X = maria

Lógico

Prolog pertenece al paradigma de programación lógica, por lo cuál su base es puramente lógica. Un lenguaje puro de Prolog puede ser expresado únicamente con un conjunto de cláusulas Horn

`mujer(H) ∧ progenitor(P, H) → hija(H, P)`

Su forma de evaluar es a través del método de prueba de resolución, donde lo que se intenta es probar la contradicción, siendo el más común el SLDNF

Entonces...

Programa: conjunto de predicados.
Predicado: Define relaciones entre sus argumentos (o parámetros). Un predicado está compuesto por un conjunto de cláusulas y se identifica por su nombre y el número de argumentos que toma. El ejemplo anterior que creamos se expresaría como hija/2, y define la relación entre Hija y Progenitor.
Clausula: Es un hecho o una regla. Si cualquier cláusula es verdadera, el predicado lo es también. En el caso anterior, de no contar con progenitor, pero sí con padre y madre, podríamos haber escrito el predicado hija/2 con dos cláusulas.

```
hija(Hija, Progenitor):-  mujer(Hija), padre(Progenitor, Hija).
hija(Hija, Progenitora):- mujer(Hija), madre(Progenitora, Hija).
```

Regla: Está conformada por una cabeza (con 0 a n parámetros) y un cuerpo. El cuerpo también es llamado goal o meta, porque es el objetivo a lo que se quiere alcanzar para verificar la regla.
Hecho: Es una regla que siempre se considera verdadera y son los elementos que forman parte de nuestra base de conocimientos.
Consulta (query): Es la forma de ejecutar los programas en Prolog. Se compone de una meta a alcanzar. Esta meta puede ser exitosa varias veces, dando como resultado un conjunto de datos.
Algunos predicados ya nos vienen dados por prolog. Los iremos conociendo de a poco
Nota: Todas las operaciones, como él y (,) también son predicados (,/2), y puedo escribir tanto

```
mujer(Hija), progenitor(Progenitor, Hija)
```

como

```
,(mujer(Hija), progenitor(Progenitor, Hija))
```

Términos

Al principio mencionamos que toda la información en prolog es representada en términos.
Estos términos pueden ser varias cosas. Nosotros solo llegaremos a utilizar alguno de estos
Variables: Son componentes que vamos a estar utilizando para ser relacionados a otros y obtener resultados. Empiezan con una letra mayúscula o con un _. Existen también variables anónimas donde su nombre solo será "_"
Término simple: Pueden ser:

- atoms: x, mati, 'con espacios'
- integer: 23
- floating: 23.1
- otros números
- otros elementos: lists, strings, pairs, asociations

Término compuesto: Conjunto de términos simples, unidos por operaciones.

Verificación de términos

Swi-prolog incluye varios predicados que permiten verificar estos tipos de términos (doc). Los más importantes son:

- var/1: true si el parámetro es una variable libre (sin valor definido [puede tener otra variable como valor, mientras ambas sean libres])
- number/1: true si el parámetro es un número
- atom/1: true si el parámetro es un atom
- compound/1: true si el parámetro es un término compuesto (eg: 2 + T)

Primer programa

Como primer programa, vamos a comenzar con la base de conocimiento de padre_de y madre_de

```
padre_de(carlos, luis). % luis sería el padre
padre_de(eduardo, luis).
padre_de(franco, carlos).
padre_de(sofia, carlos).
padre_de(julian, carlos).

madre_de(franco, maria).
madre_de(sofia, susana).
madre_de(julian, susana).
```

Intentemos ver qué cláusulas (reglas a partir de ahora, ya que lo vamos a utilizar como sinónimos) podemos crear para aprender con este caso
Primero veamos que podemos consultar

```
?- % Si quisieramos listar la información disponible
padre_de(Hijo, Padre).
```

Hijo	Padre
carlos	luis
eduardo	luis
franco	carlos
sofia	carlos
julian	carlos

```
?- % También podríamos consultar solo aquellos que son padres. Por ahora no nos van a
importar los duplicados
padre_de(_, Padre).
```

Padre
luis
luis
carlos
carlos
carlos

Intentemos crear las regla hijo_de, qué nos de para una persona, su padre y madre
Para ello nos vamos a valer de los **predicados de control**. Estos nos permiten generar relaciones lógicas. Estos son:

- ,/2 para el **and**
- ;/2 para el **or** (|/2 es un alias)
- \+/1 para la negación (not/1 es un alias)

Entonces, intentemos crear la regla hijo_de, donde se tiene que cumplir la regla padre_de **or** madre_de
Nota: Las soluciones estarán presentes debajo. Intentar antes de leerlas :)

```
% Solución
solucion_hijo_de(Progenitor, Hijo):-
    padre_de(Hijo, Progenitor);
    madre_de(Hijo, Progenitor).
```

```
?- % Debería ser carlos y susana
solucion_hijo_de(Progenitor, sofia).
```

Progenitor
carlos
susana

Ahora agreguemos una regla más, abuelo_de:

```
% Solución
solucion_abuelo_de(Hijo, Abuelo):-
    solucion_hijo_de(Padre, Hijo),
    solucion_hijo_de(Abuelo, Padre).
```

```
?- % El abuelo luis tiene 3 nietos
solucion_abuelo_de(Hijo, Abuelo).
```

Hijo	Abuelo
franco	luis
sofia	luis
julian	luis

Para los siguientes ejercicios vamos a agregar un concepto nuevo, el de comparación y unificación
Unificación:

- =\2 Unifica ambos términos, es decir, intenta relacionar ambos términos como equivalentes o da falso
- \=\2 Lo contrario de arriba, implica que ambos términos son estrictamente distintos, o da falso

Comparación:

- ==\2 Da verdadero cuando ambos términos son iguales, sin aplicar unificación
- \==\2 Da verdadero cuando ambos términos son distintos, sin aplicar unificación

Veamos algunas consultas para despejar dudas con estos nuevos operadores:

```
?- a = X
X = a
```

```
?- a = b
false
```

```
?- X = Y
X = Y
```

```
?- a == a
true
```

```
?- a == X
false
```

```
?- X == Y
false
```

¡Ahora sí!
Intentemos crear las reglas:

- progenitor/1
- medio_hermano_de/2
- hermano_de/2

```
solucion_progenitor(X):-
    padre_de(_, X);
    madre_de(_, X).

solucion_medio_hermano_de(H1, H2):-
    padre_de(H1, P1),
    padre_de(H2, P2),
    madre_de(H1, M1),
    madre_de(H2, M2),
    H1 \== H2,
    (
      (P1 == P2, M1 \== M2);
      (P1 \== P2, M1 == M2)
    ).

solucion_hermano_de(H1, H2):-
    padre_de(H1, P),
    padre_de(H2, P),
    madre_de(H1, M),
    madre_de(H2, M),
    H1 \== H2.
```

```
?- solucion_hermano_de(X, Y)
```

X	Y
sofia	julian
julian	sofia

Por último nos falta ver cómo operar con números
Vamos a disponer de los siguientes operadores matemáticos:

- Comparación de mayor / menor: < >
- Comparación de mayor igual / menor igual: =< >= (caritas tristes)
- Comparación con evaluación igual / distinto: =:= =\=
- Unificación con evaluación (legacy): is (evalúa parte derecha y unifica sobre izquierda)
- Operadores unarios: + -
- Operaciones aritméticas: + - * / ** // mod
- Operaciones de bits: << >> /\ \/ xor
- Funciones aritméticas: abs sqrt sin cos tan exp log

Podemos pasar a la consola para probar cómo funcionan.

Incluso podemos intentar hacer una pequeña calculadora si nos animamos!

```
% Predicado base
calculadora(X, Y, Operacion, Resultado):-false. % TODO

% Operacion suma
operacion(X, Y, '+', Resultado):-Resultado is X + Y.
```

Conjuntos

Las consultas de prolog que unifican valores a variables (es decir, que *retorna* información adicional) pueden ser pensadas en forma de conjuntos. Un predicado junto a sus atributos variables pueden ser pensados como un conjunto de información. De esa manera podemos pensar ciertas operaciónes como operaciones de conjuntos (unión \/, intersección /\ y resta --)

Tomemos como ejemplo dos conjuntos de hechos, `parcial1` y `parcial2`

```
parcial1(a, 1).
parcial1(b, 2).
parcial1(c, 3).
parcial1(d, 4).
parcial1(e, 5).
parcial1(f, 6).

parcial2(a, 4).
parcial2(b, 5).
parcial2(c, 6).
parcial2(g, 7).
parcial2(h, 8).
parcial2(i, 9).
```

Resta

```
% Resta
alumnos_solo_parcial1(X):-
    parcial1(X, _),       % Conjunto de alumnos que rindieron parcial1
    \+parcial2(X, _).     % MENOS conjunto de alumnos que rindieron parcial2
% {a,b,c,d,e,f} - {a,b,c,g,h,i} = {d,e,f}

alumnos_solo_parcial2(X):-
    parcial2(X, _),
    \+parcial1(X, _).
% {a,b,c,g,h,i} - {a,b,c,d,e,f} = {g,h,i}
```

Interseccion

```
% Intersección
alumnos_ambos_parciales(X):-
    parcial1(X, _),
    parcial2(X, _).
% {a,b,c,g,h,i} /\ {a,b,c,d,e,f} = {a,b,c}
```

Union

```
% Union
alumnos_unidos(X):-
    parcial1(X, _);
    parcial2(X, _).
% {a,b,c,g,h,i} \/ {a,b,c,d,e,f} = {a,b,c,g,h,i,a,b,c,d,e,f}
```

Union unica

```
% Union unica
alumnos(X):-
    parcial1(X, _), parcial2(X, _)    % interseccion A /\ B
    ;                                 % union
    parcial1(X, _), \+parcial2(X, _)  % resta A-B
    ;                                 % union
    parcial2(X, _), \+parcial1(X, _). % resta B-A
```

Union unica mas simple

```
% Union unica más simple
alumnos2(X):-
    parcial1(X, _);
    (parcial2(X, _), \+parcial1(X, _)).
```

Lo interesante es que ahora tenemos un nuevo conocimiento que es alumno, que podría ser alumnos de una comisión, entonces puedo querer volver a realizar una union para obtener todos los alumnos de una cátedra y así sucesivamente.

Nota: Para poder trabajar con estas 3 operaciones de conjuntos, se debe trabajar siempre con el mismo conjunto de variables en todos los casos.

Por último vamos a ver una operación de conjuntos sobre si mismo, donde se involucran 2 variables distintas, que es el producto cartesiano o producto cruz (X)

```
% Esto nos dará 81 resultados, relacionando los 9 alumnos con los otros 9, uno a uno
producto_cartesiano_alumnos(Alumno1, Alumno2):-
    alumnos(Alumno1),
    alumnos(Alumno2).

% Equipo de alumnos posible.
% Me interesa no repetir los pares en orden inverso, es decir {a, b} y {b, a}
% Para ello puedo usar el operador @</2 para quedarme con el termino menor (incluso p
equipos_posibles(Alumno1, Alumno2):-
    producto_cartesiano_alumnos(Alumno1, Alumno2),
    Alumno1 @< Alumno2.
```

Máximos y mínimos

Prolog permite el calculo de máximos y mínimos a través de operaciones intermedias de conjuntos. Vamos a utilizar las operaciones de conjuntos vistas hasta el momento para poder obtener el mayor o menor de un conjunto.

Tomemos como ejemplo un conjunto de hechos de notas y veamos como obtener la nota más alta. Podemos intentar luego, ver a quien o quienes les pertenece la misma.

```
nota(b, 3).
nota(c, 2).
nota(d, 4).
nota(e, 8).
nota(f, 6).

producto_cartesiano(X, Y):-nota(_, X), nota(_, Y).

seleccion(X, Y):-producto_cartesiano(X, Y), X < Y.

% Para este paso
% - Mi X va a tener todos los valores posibles menos el máximo
% - Mi Y todos menos el mínimo

% Decido con que parte me quedo. Esta parte va a ser lo que se reste del conjunto tot
proyeccion(X):-seleccion(X, _).

% Puedo usar un alias para que lo que contiene quede más claro
todos_menos_max(X):-proyeccion(X).

% Forma más corta. Utilizar solo cuando no se pida el paso a paso en enunciados/exame
% todos_menos_max(X):-nota(_, X), nota(_, Y), X < Y.

% En caso de haber más de un máximo, obtendremos un resultado por cada uno
max_nota(X):-nota(_, X), \+ todos_menos_max(X).

max_nota(X, Alumno):-nota(Alumno, X), \+ todos_menos_max(X).
```

Práctica

Ahora intentemos realizar un ejercicio con todo lo que aprendimos. Vamos a disponer de un conjunto de notas con los alumnos, como en el caso anterior, pero un alumno puede tener más de una nota.

Quiero poder utilizar un predicado llamado mayor_nota/2 que me de la nota más alta pero por cada alumno

```
nota(a, 1).
nota(a, 5).
nota(a, 8).
nota(a, 9).

nota(b, 3).
nota(b, 3).
nota(b, 3).
nota(b, 3).

nota(c, 1).
nota(c, 4).
nota(c, 9).

nota(d, 4).

% Programa
solucion_todos_menos_alta(A, X):-nota(A, X), nota(A, Y), X < Y.

solucion_mayor_nota(A, X):- nota(A, X), \+solucion_todos_menos_alta(A, X).
```

Recursividad

La recursividad en Prolog permite definir un predicado en términos de sí mismo. Esto implica definir casos base, que son condiciones de terminación, junto con casos recursivos que simplifican gradualmente el problema hasta que alcanza el caso base.

Ejemplo

Consideremos un problema básico de recursividad como suele ser el factorial. El factorial de un número entero no negativo N, denotado como N!, es el producto de todos los enteros positivos menores o iguales a N. Es decir, para 7!, la operación debería ser 7 x 6 x 5 x 4 x 3 x 2 x 1.

Esto también se puede expresar como el factorial de 7! es igual a 7 x 6!, que es igual a 7 x 6 x 5! y así sucesivamente.

```
factorial(0, 1).              % Regla: caso base
factorial(N, Resultado):-     % Regla: caso recursivo
    N > 0,                    % para todo N mayor a 0
    N1 is N - 1,              % se define N1 como N - 1
    factorial(N1, Parcial),   % se invoca la recursividad, preguntando por el valor un
    Resultado is Parcial * N. % se obtiene el resultado
```

La estructura de la recursividad es siempre de la misma manera. Depende el caso quizás es necesario definir más de una regla base o regla recursiva, o quizás se deba invocar a la recursividad más de una vez, o la operación no es tan sencilla como en este caso, pero la base se respeta

- Regla base
- Regla recursiva
    - Alcance de la regla
    - Invocación recursividad
    - Operación con resultado parcial

Observen que la consulta solo dio verdadero para los casos donde X >= 0. Eso es porque para casos negativos de N, no está definida la regla

Observen también que estamos utilizando el operador is/2. Este operador no cumple con las reglas del patrón lógico como el resto de los operadores, por lo tanto, la regla necesita tener si o si definidas para el momento de la operación is/2, todo lo que este a su derecha, que es lo que va a ser evaluado. Un mejor operador sería el #=/2 pero no entra de nuestro alcance de la materia por estar incluido en una biblioteca (clpfd)

```
?- % Fallará para la regla recursiva
   factorial(X, Resultado).
```

X	Resultado
0	1

```
Arguments are not sufficiently instantiated
In:
  [2] _1678>0
  [1] ebf321a4-5859-43e7-a543-5be5a4cbabd8:factorial(_1734,_1736) at line 3
```

```
?- % Fallará
   factorial(X, 3628800).
```

```
Arguments are not sufficiently instantiated
In:
  [2] _2000>0
  [1] e3ffb115-fb87-474d-8903-f6c7a837c23b:factorial(_2056,3628800) at line 3
```

```
?- % Muy lento ya que intentará evaluar en cada paso, salvo que me detenga en el primero
   between(0, 100000, X),
   factorial(X, 3628800).
```

X
10

```
Trapped tripwire max_integer_size: big integers and rationals are limited to 0 bytes
```

Listas

En prolog, las listas son colecciones de terminos (variables, terminos simples (atoms, integer, lists, etc), terminos compuestos (2+2, etc)). Estás estructuras ya incluidas en el lenguaje nos van a permitir representar conjuntos sobre los mismos terminos y desarrollar estructuras de datos complejas como pilas, colas, grafos, arboles, etc.

La lista se escribe entre \[\] y sus elementos son separados por coma. A continuación unos ejemplos de listas:

`[1, 2, 3]`
`[a, b, c]`
`[]`
`[a, [b, c], [dalmata, perro(dalmata)]] % Largo: 3`

Prolog tiene un símbolo | (pipe) para listas que se utiliza para armarlas o descomponerlas (según el punto de vista).

Este símbolo define la separación entre la cabeza o **head** y la cola o **tail** de una lista. Ejecutemos unas consultas y veamos su comportamiento

```
?- [Head | Tail] = [1, 2, 3, 4, 5].
```

Head = 1,
Tail = [2, 3, 4, 5]

```
?- [Head | Tail] = [1].
```

Head = 1,
Tail = []

```
?- [Head | Tail] = [].
```

false

```
?- [a, b | Resto] = [a, b, c, d, e].
```

Resto = [c, d, e]

```
?- [a, b | Resto] = [1, 2, 3, 4, 5].
```

false

```
?- [Head, Head2 | Resto] = [1, 2, 3, 4, 5].
```

Head = 1,
Head2 = 2,
Resto = [3, 4, 5]

```
?- [A, B, C, D, E] = [1, 2 + 2, c, \+true, atom(a)]
```

A = 1,
B = 2+2,
C = c,
D = (\+true),
E = atom(a)

La representación de la lista descompuesta se puede ver como una combinación en cadena de cabeza y cola

```
?- [1, 2, 3] == [1 | [2 | [3 | []]]]
```

true

Podemos definir una regla que nos separe la cabeza y la cola para ir comenzando con aplicaciones de listas

```
1 cabeza_cola([H|T], H, T).
```

```
?- cabeza_cola([a, b, c], Cabeza, Cola).
```

Cabeza = a,
Cola = [b, c]

Ahora combinemos el uso de listas en reglas, con recursividad, para lograr realizar operaciones que antes o nos eran imposibles, o su código se volvía muy complicado y rebuscado.

Empecemos por combinar dos listas en una. Para ello voy a utilizar tres argumentos, dos para las listas parciales y uno para la lista combinada. La recursividad hará que eventualmente la segunda lista parcial se unifique como la lista combinada, y al volver sobre la recursion, se le adicionará a esta lista, cada uno de los elementos de la primer lista parcial

```
% Caso base: Si una lista es vacia, y la otra es una lista,
% la lista concatenada es la lista parcial
concatenar([], L, L):-is_list(L).

% Caso recursivo:
% Para toda lista con al menos un elemento
% Invoco a la recursividad hasta obtener L en FL, y T vacia
% Luego antepongo cada elemento del head de la primer lista en orden inverso por la rec
concatenar([H | T], L, [H | FL]) :-
    concatenar(T, L, FL).
```

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

L = [1, 2, 3, 4]

```
?- concatenar([1, 2], [H, H2 | [a]], [1, 2, 3, 4, X]).
```

H	H2	X
3	4	a

```
?- concatenar([], x, L).
```

false

Muchos de los terminos más sencillos para trabajar con listas en prolog ya existen de forma nativa o a través de alguna biblioteca interna. Por ejemplo, nuestro predicado concatenar/3, existe con el nombre append/3.

También tenemos otros dos predicados muy utilizados como length/2 donde unifica en sus argumentos una lista con su largo, y member/2 donde unificara en sus argumentos una lista con sus elementos. Intentemos crearlos nosotros para practicar

```
solucion_largo([], 0).
solucion_largo([_|XS], N):-
    solucion_largo(XS, N1),
    N is N1 + 1.

solucion_miembro(X,[X|_]).
solucion_miembro(X,[_|T]):-solucion_miembro(X,T).
```

Hay casos donde la información la vamos a tener dispuesta en hechos, pero queremos trabajarlo como listas. Para ello debemos tomar el conjunto deseado y pasarlo a una lista. Hay un predicado denominado findall/3 que se encarga de esto mismo. Para utilizarlo, se espera una variable que va a funcionar como un extractor, una meta (goal), y una variable con la lista con los valores recolectados

```
curso1(a).
curso1(b).
curso1(c).

curso2(d).
curso2(e).
curso2(f).

nota(a, 2).
nota(b, 6).
nota(c, 2).
nota(d, 2).
nota(e, 9).
nota(f, 7).

aprobada(Nota):-Nota >= 4.
```

```
?- findall(X, nota(_, X), ListaNotas)
```

X	ListaNotas
X	[2, 6, 2, 2, 9, 7]

```
?- % Listado de alumnos desaprobados
   % El guión bajo delante nos indica que es una variable que no se desea mostrar
   findall(_Y, (nota(_Y, _X), \+aprobada(_X)), ListaNotas)
```

ListaNotas
[a, c, d]

```
?- % Listado de notas aprobadas de alumnos del curso2
   findall(_X, (nota(_A, _X), aprobada(_X), curso2(_A)), ListaNotas)
```

ListaNotas
[9, 7]

Ejercicios

03 - Agencia de viajes

```
Una agencia de viajes propone a sus clientes viajes de una o varias
semanas a Roma, Londres o Túnez. El catálogo de la agencia contiene,
para cada destino, el precio del transporte (con independencia de la
duración) y el precio de una semana de estancia que varía según el
destino y el nivel de comodidad elegidos: hotel, hostal o camping.
Escribir el conjunto de declaraciones que describen este catálogo.
transporte(roma, 20).
transporte(londres, 30).
transporte(tunez, 10).

alojamiento(roma, hotel, 50).
alojamiento(roma, hostal, 30).
alojamiento(roma, camping, 10).
alojamiento(londres, hotel, 60).
alojamiento(londres, hostal, 40).
alojamiento(londres, camping, 20).
alojamiento(tunez, hotel, 40).
alojamiento(tunez, hostal, 20).
alojamiento(tunez, camping, 5).
```

```
1. Crear la regla `viaje/4` tal que se cumpla que: "el viaje a una
Ciudad durante S semanas con un Hospedaje valido con un Precio total"
2. Completar con `viajeeconomico/5`, agregando un parámetro extra que
defina el Precio máximo del viaje.
3. (Opcional) Utilizando el predicado
[var/1](https://www.swi-prolog.org/pldoc/man?predicate=var/1), intentar
definir el precio si Semanas esta definido, sino dejar la expresion
planteada.
4. (Opcional) Utilizando el modulo clpfd `:-
use_module(library(clpfd)).`, reemplazar el is/2 del punto 1 por el
operador #=/2 y comprobar que pasa si no se tiene información
suficiente de las semanas y del precio, y luego comprobar si es capaz
de predecir las semanas. Nota: #=/2 funciona solo para enteros.

viaje(Ciudad, Semanas, Hospedaje, Precio):-
    transporte(Ciudad, PrecioTransporte),
    alojamiento(Ciudad, Hospedaje, PrecioSemanalHospedaje),
    Precio is PrecioTransporte + PrecioSemanalHospedaje * Semanas.

viajeeconomico(Ciudad, Semanas, Hospedaje, Precio, PrecioMinimo):-
    viaje(Ciudad, Semanas, Hospedaje, Precio),
    Precio =< PrecioMinimo.

viaje2(Ciudad, Semanas, Hospedaje, Precio):-
    transporte(Ciudad, PrecioTransporte),
    alojamiento(Ciudad, Hospedaje, PrecioSemanalHospedaje),
    (
        (
            \+var(Semanas),
            Precio is PrecioTransporte + PrecioSemanalHospedaje *
Semanas
        );
        (
            var(Semanas),
            Precio = PrecioTransporte + PrecioSemanalHospedaje *
Semanas
        )
    ).
:- use_module(library(clpfd)).
viaje3(Ciudad, Semanas, Hospedaje, Precio):-
    transporte(Ciudad, PrecioTransporte),
    alojamiento(Ciudad, Hospedaje, PrecioSemanalHospedaje),
    PrecioTransporte + PrecioSemanalHospedaje * Semanas #= Precio.
```

04 - Notas de examen

```
Se dispone de un listado con los resultados de los parciales de los
alumnos del curso de **Paradigmas de Programación**, organizado de la
siguiente manera:
parcial1(luisa, 6).
parcial1(pedro, 5).
parcial1(maria, 8).
parcial1(carlos, 1).
parcial1(laura, 3).
parcial1(agos, 8.5).
parcial1(juan, 10).
parcial1(julio, 2).
parcial2(luisa, 7).
parcial2(pedro, 6).
parcial2(maria, 9).
parcial2(carlos, 8).
parcial2(laura, 4).
parcial2(agos, 8.5).
parcial2(juan, 4).
parcial2(julio, 1).
Se desea obtener:
1. El listado de los alumnos que promocionan la materia
2. El mismo listado pero esta vez incluyendo la nota final (promedio de
los dos parciales) para cada uno
3. El listado de los alumnos que recursan la materia.
4. El listado de los alumnos que obtendrán la cursada.
5. (maximos) A fin de entregar la medalla al mérito, encontrar de
aquellos que promocionan (1b), el o los alumnos con mayor nota final
(nombre y nota)
6. (maximos) Ahora se desea obtener cuales fueron las dos notas más
altas, considerando simplemente a la nota como el promedio de la nota
de parcial1 y parcial2. Solo interesan los números. Un tip es pensar la
resolución en dos etapas, la más alta, y después la más alta de lo
restante
%1
promocion(A, Nota):-
    parcial1(A, N1),
    parcial2(A, N2),
    N1 >= 7,
    N2 >= 7,
    Nota is (N1 + N2) / 2.
```

```
% 2
recursa(A):-
    parcial1(A, N1),
    parcial2(A, N2),
    (N1 < 4; N2 < 4).
% 3
alumno(A):-parcial1(A, _), parcial2(A, _).
cursada(A):-
    alumno(A),
    \+ promocion(A, _),
    \+ recursa(A, _).

% CASI 4 y CASI 5 juntos
nota(Alumno, Nota):-
    parcial1(Alumno, P1),
    parcial2(Alumno, P2),
    Nota is (P1 + P2) / 2.
producto_cartesiano(A, B):-
    nota(_, A), nota(_, B).
seleccion(A, B):-
    producto_cartesiano(A, B),
    A < B.
proyeccion(A):-
    seleccion(A, _).
% alias
todos_menos_max(A):-
    proyeccion(A).
maximo(X):-
    nota(_, X), \+todos_menos_max(X).
```

07 - Conjuntos de actividades

```
Dado el siguiente listado de actividades extracurriculares que realiza
cada estudiante
natacion(a).
natacion(b).
natacion(c).
natacion(d).

futbol(a).
futbol(b).
futbol(e).
futbol(f).
```

```
teatro(a).
teatro(c).
teatro(e).
teatro(g).
Se desea saber
1. Qué estudiantes participan de todas las actividades
2. Qué estudiantes realizan futbol y no teatro
3. Qué estudiantes realizan al menos alguna actividad. Evitar
duplicados
4. Qué estudiantes participan de al menos dos actividades
5. Teniendo un listado de estudiantes total estudiantes/1 comprendido
entre \[a, j\], qué estudiantes no realizan ninguna actividad

% Qué estudiantes participan de todas las actividades
todas_actividades(Estudiante):-
    natacion(Estudiante),
    futbol(Estudiante),
    teatro(Estudiante).

% Qué estudiantes realizan futbol y no teatro
futbol_sin_teatro(Estudiante):-
    futbol(Estudiante),
    \+teatro(Estudiante).

% Qué estudiantes realizan al menos alguna actividad. Evitar duplicados
alguna_actividad(Estudiante):-
    natacion(Estudiante).
alguna_actividad(Estudiante):-
    futbol(Estudiante),
    \+natacion(Estudiante).
alguna_actividad(Estudiante):-
    teatro(Estudiante),
    \+natacion(Estudiante),
    \+futbol(Estudiante).
```

```
% Qué estudiantes participan de al menos dos actividades
% Nota: Si se elige quitar del conjunto a teatro en este caso, se debe
agregar una regla extra que incluya a todas_actividades adicionalmente,
sino quedará fuera "a"
dos_o_mas_actividades(Estudiante):-
    natacion(Estudiante),
    futbol(Estudiante).
dos_o_mas_actividades(Estudiante):-
    natacion(Estudiante),
    teatro(Estudiante),
    \+futbol(Estudiante).
dos_o_mas_actividades(Estudiante):-
    futbol(Estudiante),
    teatro(Estudiante),
    \+natacion(Estudiante).

% Teniendo un listado de estudiantes total estudiantes/1 comprendido
entre [a, j], qué estudiantes no realizan ninguna actividad
estudiantes(a).
estudiantes(b).
estudiantes(c).
estudiantes(d).
estudiantes(e).
estudiantes(f).
estudiantes(g).
estudiantes(h).
estudiantes(i).
estudiantes(j).

sin_actividad(Estudiante):-
    estudiantes(Estudiante),
    \+alguna_actividad(Estudiante).

sin_actividad2(Estudiante):-
    estudiantes(Estudiante),
    \+natacion(Estudiante),
    \+futbol(Estudiante),
    \+teatro(Estudiante).
```

08 - Reuniones empresariales

```
Dada una base de conocimientos de disponibilidad de horarios (bloques
de 1 hora) para reuniones
```

```
disponibilidad_equipo(marketing, 9).
disponibilidad_equipo(marketing, 10).
disponibilidad_equipo(desarrollo, 10).
disponibilidad_equipo(desarrollo, 11).
disponibilidad_equipo(ventas, 9).
disponibilidad_equipo(ventas, 11).

disponibilidad_cliente(google, 9).
disponibilidad_cliente(google, 10).
disponibilidad_cliente(google, 11).
disponibilidad_cliente(apple, 10).
disponibilidad_cliente(apple, 11).
disponibilidad_cliente(microsoft, 9).
disponibilidad_cliente(microsoft, 11).

1. Se desea saber la disponibilidad para formar una reunión entre
nuestros equipos y los clientes. Se debe crear la regla
disponibilidad_reunion/3 que determine las reuniones posibles de cada
uno de nuestros equipos con cada uno de nuestros clientes en cada
horario posible. Para el caso planteado, la salida la siguiente
| Equipo     | Cliente   | Hora |
| ---------- | --------- | ---- |
| marketing  | google    | 9    |
| marketing  | microsoft | 9    |
| marketing  | google    | 10   |
| marketing  | apple     | 10   |
| desarrollo | google    | 10   |
| desarrollo | apple     | 10   |
| ...        | ...       | ...  |

2. Se agrega la posibilidad de reservar un horario de reunión para un
equipo y un cliente en un horario en especifico, con el hecho
reunion_pactada/3. Se pide crear una nueva regla
disponibilidad_reunion_final/3 donde se tenga en cuenta las reuniones
ya pactadas y se excluyan de las reuniones disponibles, pero ojo, si
por ejemplo una de las reuniones pactadas es de ventas con google a las
11, el equipo de ventas no estará disponible a las 11, pero tampoco lo
estará el equipo de google (ya que se considera solo 1 disponibilidad
por equipo y cliente por horario)

Ejemplo: Para las siguientes reuniones pactadas
reunion_pactada(ventas, google, 11).
reunion_pactada(marketing, apple, 10).
```

```
La salida esperada será
| Equipo     | Cliente   | Hora |
| ---------- | --------- | ---- |
| marketing  | google    | 9    |
| marketing  | microsoft | 9    |
| desarrollo | google    | 10   |
| desarrollo | apple     | 11   |
| desarrollo | google    | 11   |
| ventas     | google    | 9    |
| ventas     | microsoft | 9    |
% 1
disponibilidad_reunion(Equipo, Cliente, Hora):-
    disponibilidad_equipo(Equipo, Hora),
    disponibilidad_cliente(Cliente, Hora).

% Base de conocimientos adicional del enunciado
reunion_pactada(ventas, google, 11).
reunion_pactada(marketing, apple, 10).

% 2: Posible solución utilizando la regla ya creada y restas de
conjuntos
disponibilidad_reunion_pactadas(Equipo, Cliente, Hora):-
    disponibilidad_reunion(Equipo, Cliente, Hora),
    \+reunion_pactada(Equipo, _, Hora),
    \+reunion_pactada(_, Cliente, Hora).
```

11 - Recursividad

```
2. Codifique en prolog las reglas necesarias para obtener el término N
en la serie de Fibonacci (sin, y con mejora)
% 2
% sin mejora
fib(0,0).
fib(1,1).
fib(N,R):-
    N > 1,
    NA is N - 1,
    NAA is N - 2,
    fib(NA,RA),
    fib(NAA,RAA),
    R is RA + RAA.
```

```
% con mejora
fibonacci(0,0,0).
fibonacci(1,1,0).
fibonacci(N,R,RA):-
    N > 1,
    NA is N - 1,
    % NAA is N - 2,
    fibonacci(NA, RA, RAA),
    % fibonacci(NAA,RAA,RAAA),
    R is RA + RAA.

fibonacci(N,R):-
    fibonacci(N,R,_).
```

12 - Operaciones de listas

```
Utilizando los conceptos de lista y recursividad y sin utilizar los
predicados existentes para listas en prolog, resolver:

1. Suma de todos los valores de una lista. suma/2 (Lista, Total)
2. Contar cuantos valores repetidos de un elemento hay en una lista.
contar/3 (Lista, Elemento, Cantidad)
3. Buscar todas las posiciones de un elemento en una lista. indice_de/3
(Lista, Elemento, Posicion)

Pudiendo utilizar predicados existentes, resolver

4. Para un listado de 0 a n hechos nota/1, con notas de 1 a 10, hallar
el promedio de las mismas. promedio_nota/1
%1 Suma de todos los valores de una lista
suma([], 0).
suma([Cabeza | Cola], Total):-
    suma(Cola, Acum),
    Total is Acum + Cabeza.
%2 Contar cuantos valores repetidos de un elemento hay en una lista.
% Se podría haber usado ; en vez de definir otra regla
igual_int(A, A, 1).
igual_int(A, B, 0):-A \== B.

contar([], _, 0).
contar([Cabeza | Cola], X, Total):-
    contar(Cola, X, Parcial),
    igual_int(Cabeza, X, A),
    Total is Parcial + A.
```

```
%3 Buscar todas las posiciones de un elemento en una lista

% Si X es el primer elemento de la sublista, Pos es 0
indice_de([X | _], X, 0).

% No importa que relacion tenga cabeza con X, sigo consultando el resto
de los elementos
% Para cada elemento que haya sido matcheada por la regla anterior, voy
a incrementar su posicion hasta la actual
indice_de([_ | Cola], X, Posicion):-
    indice_de(Cola, X, Posicion1),
    Posicion is Posicion1 + 1.

%4 Para un listado de 0 a n hechos nota/1, con notas de 1 a 10, hallar
el promedio de las mismas

nota(2).
nota(6).
nota(2).
nota(2).
nota(9).
nota(7).

promedio_nota(Promedio):-
    findall(X, (nota(X)), ListaNotas),
    ListaNotas \= [], % Previene dividir por 0
    sum_list(ListaNotas, Suma),
    length(ListaNotas, Largo),
    Promedio is Suma / Largo.
```