Predicados de orden superior: Cláusula Forall

"Para todo x se cumple p(x)".

Queremos que una condición se cumpla para todas las variables posibles. Tenemos esta base de

conocimientos:

```
materia(pdp, 2).
materia(proba, 2).
materia(sintaxis, 2).
materia(algoritmos, 1).
materia(analisisI, 1).
nota(nicolas, pdp, 10).
nota(nicolas, proba, 5).
nota(nicolas, sintaxis, 8).
nota(malena, pdp, 4).
nota(malena, proba, 2).
nota(raul, pdp, 9).
```

"Un alumno terminó un año si aprobó todas las materias de ese año"

```
terminoAnio(Alumno, Anio):-
	forall(materia(Materia, Anio), (nota(Alumno, Materia, Nota), Nota >= 4)).
```

o mejor

```
terminoAnio(Alumno, Anio):-
	forall(materia(Materia, Anio), aprobo(Alumno, Materia)).

aprobo(Alumno, Materia):-nota(Alumno, Materia, Nota), Nota >= 4.
```

Algunas consultas:
?- terminoAnio(nicolas, 2). Yes

?- terminoAnio(malena, 2). No

?- terminoAnio(raul, 2).
No
¿Es inversible? Nop. Si yo consulto

```
forall(materia(Materia, 2), aprobo(Alumno, Materia))
```

lo que PROLOG hace es afirmar que para todas las materias de 2° año **todos** los alumnos aprobaron esa materia. Si no ligo las variables, forall buscará probar que para todo el juego de incógnitas se cumplen los predicados. ¿Qué puedo hacer para que el forall haga la pregunta por cada alumno?

Y... el alumno no debe ser incógnita al momento de usar el forall:

```
terminoAnio(Alumno, Anio):-
       alumno(Alumno), forall(materia(Materia, Anio), aprobo(Alumno, Materia)).
```

¿Cómo genero el predicado alumno? No hace falta escribir en la base de conocimientos todos los alumnos uno por uno. Me alcanza con basarme en el predicado nota/1:

```
alumno(Alumno):-nota(Alumno, , ).
```

Entonces yo sí puedo preguntar qué alumnos terminaron 2° año:

?- terminoAnio(Alumno, 2).
Alumno = nicolas
(la única macana es que nos repite las soluciones, pero bueh... para esta cursada no nos vamos a preocupar por eso).

¿Puedo preguntar termino(Alumno, Anio)?
¿Qué va a tratar de hacer el forall?

Y. como está armado el predicado va a preguntar si el alumno que encontré aprobó **todas** las materias de **todos** los años. ¿Se entiende cómo la incógnita en el forall me determina preguntas distintas?
¿Qué puedo hacer para que forall pregunte si **un** alumno aprobó **todas** las materias de **un** año?

```
terminoAnio(Alumno, Anio):- alumno(Alumno), anio(Anio),
           forall(materia(Materia, Anio), aprobo(Alumno, Materia)).
```

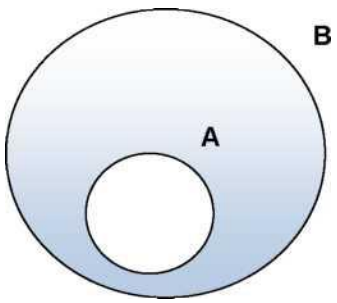
Fíjense que lo que estamos haciendo es fijar un dominio para los argumentos: el primer argumento no es cualquier individuo, tiene que ser un Alumno, el segundo tiene que ser un año de la carrera (no cualquier número).

Dejamos la codificación del predicado anio al lector.

Algunos ejercicios

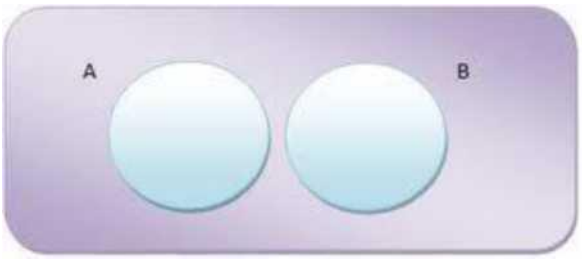
Podemos aplicar el predicado forall para resolver dos ejercicios de lógica de conjuntos:

- incluido: un conjunto A está incluido en otro B si todos los elementos de A están en B.



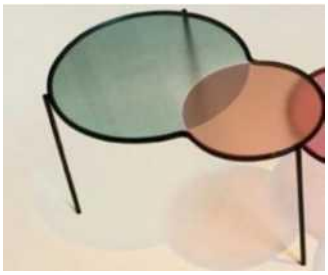

```
incluido(A, B):-forall(member(X, A), member(X, B)).
```

- disjuntos: un conjunto A es disjunto de B si se cumple que todos los elementos de A no están en B.




```
disjuntos(A, B):-forall(member(X, A), not(member(X, B))).
```

¿Dónde uso predicados de orden superior? En forall y en not.



¿Cómo puedo definir la intersección de dos conjuntos?

- intersección: son todos los elementos del conjunto A que también están en B:

```
interseccion(A, B, C):-
                              findall(X, (member(X, A), member(X, B)), C).
```

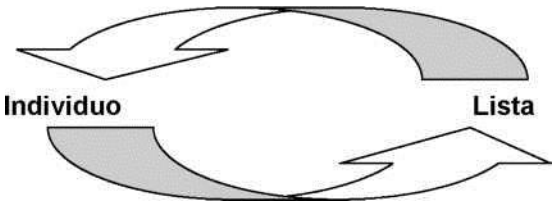
Inversibilidad en member

Una consecuencia de que member sea parcialmente inversible es que me permite preguntarle tanto ?- member(1, [4, 1, 5])

Como
?- member(X, [4, 1, 5])
X = 4 ;
X = 1 ;
X = 5 ;

Lo cual me permite utilizar los elementos que componen una lista en predicados individuales. En este caso member actúa como predicado **generador** de un subdominio (los que pertenecen a esta lista).

member



findall

En general, podemos decir que cuando tenemos que partir de hechos individuales podemos utilizar un predicado de orden superior[1] (findall/3, setof/3, bagof/3, etc.) para generar una lista con los elementos que cumplen un criterio. Y si tenemos la información en formato de lista, podemos utilizar el member/2 para trabajar los individuos en forma particular.

[1] Se recomienda igualmente usar `findall/3` en lugar de `setof/3` o `bagof/3`.

Límites a la inversibilidad - parte II[2]

Hemos visto anteriormente algunos casos en donde los predicados no admitían inversibilidad:

- Negación: cláusula not
- Operaciones aritméticas: +, -, is
- Comparación: >, <

Agregamos entonces dos casos más:

- Cláusula findall
- Cláusula forall

En los cinco casos podemos utilizar predicados generadores...

Generación

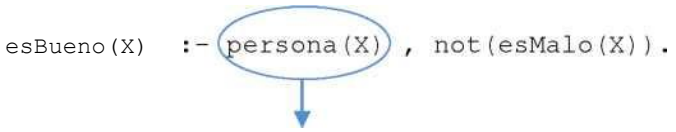
- ¿cómo funcionaba la negación? Por falla ...

```
esMalo(feinmann).
esMalo(hadad).
esMalo(etchecopar).

esBueno(X):-not(esMalo(X)).
```

esBueno: ¿es inversible?
No, porque no puedo preguntarle al motor quiénes son buenos. Sólo puedo preguntar si un individuo es bueno o malo.

En el mismo ejemplo, ¿qué pasa si hago?



Predicado generador

Utilizamos el concepto de generación para dos cosas:

1) le aplicamos un dominio al argumento. Antes era válido preguntar esBueno(3), ahora esBueno es una relación donde sólo intervienen personas. Esto se relaciona con el concepto de *tipo:* en un lenguaje con chequeo estático de tipos podríamos traducir la relación esBueno(X) como:

   ```
   public boolean esBueno(Persona persona) { ... }
   ```

   O sea que esperamos que el argumento sea de *tipo* Persona.

2) Si persona es un hecho o una regla inversible, el predicado esBueno pasa a ser inversible. Ahora sí puedo pedir:

   esBueno(sergioDenis) o esBueno(Quien).

porque persona(X) genera los valores que necesito.

Predicados generadores con cláusulas findall

Modelando un juego de truco tenemos estas cláusulas:

```
carta(juan,   cuatro,   copa).
carta(juan,   siete,   basto).
carta(juan,    tres,    oro).
```

[2] Como lectura complementaria recomendamos ver los ejemplos disponibles en:
http://uqbar.no-ip.org/uqbarWiki/index.php/Paradigma_L%C3%B3gico_-_casos_de_no_inversibilidad

```
carta(mario,  cinco,  espada).
carta(mario,  siete,  espada).
carta(mario, dos, basto).

palo(oro).
palo(basto).
palo(espada).
palo(copa).
```

Queremos saber si un jugador tiene envido: consideramos que tiene envido si tiene al menos dos cartas del mismo palo...

```
tieneEnvido(Persona):-
        findall(Palo, carta(Persona, _, Palo), Palos),
        length(Palos, Cantidad),
        Cantidad >= 2.
```

Cuando consultamos:
?- tieneEnvido(juan).
Yes

Nos dice que Juan ¡tiene envido!
¿Por qué?

Veamos qué devuelve:
?- findall(Palo, carta(juan, _, Palo), Palos).
Palo = _G428
Palos = [copa, basto, oro] ;

Claro, Palo es una variable que está sin ligar antes del findall. revisemos de nuevo la cláusula:

```
tieneEnvido(Persona):-
        findall(Palo, carta(Persona, _, Palo), Palos),
        length(Palos, Cantidad),
        Cantidad >= 2.
```

Esto está diciendo: “una persona tiene envido si tiene al menos 2 cartas *de cualquier palo".*

Si queremos que esto cambie a: “una persona tiene envido si tiene al menos 2 cartas del mismo palo, donde el palo puede ser cualquiera de la baraja española”:

```
tieneEnvido(Persona):- palo(Palo),
        findall(Palo, carta(Persona, _, Palo), Palos),
        length(Palos, Cantidad),
        Cantidad >= 2.
```

Ahora sí Palo está ligado en el momento del findall: puedo tener dos o más cartas de copa, de basto, de oro o de espada...

En general, debemos asegurarnos que en el momento de hacer el findall tengamos ligadas las variables que deban estarlo. Para eso nos ayudamos con predicados generadores.

Predicados generadores con cláusulas forall

Tomemos este caso, donde debemos resolver si un auto le viene perfecto a una persona, donde le viene perfecto = tiene todas las características que la persona quiere.

```
vieneCon(p206, abs). vieneCon(p206,          vieneCon(kadisco, tacometro).
levantavidrios). vieneCon(p206,              quiere(carlos, abs). quiere(carlos,
direccionAsistida).                          mp3). quiere(roque, abs).
vieneCon(kadisco, abs).                      quiere(roque, direccionAsistida)
vieneCon(kadisco, mp3).

leVienePerfecto(Auto, Persona):-
       forall(quiere(Persona, Caracteristica), vieneCon(Auto, Caracteristica)).
```

Esta regla no es inversible en ninguno de los dos argumentos porque el forall necesita que tanto Persona como Auto vengan ligadas.

Quiero que para cada par (Auto, Persona) se fije si **un auto concreto** tiene *todo* lo que **una persona concreta** quiere. En general, cuando hay findall o un forall me tengo que fijar si no necesito que ciertas variables estén ligadas[3].

Por eso tengo que generar tanto los autos como las personas, y eso se hace así:

```
% agrego definición explícita de autos y personas
auto(p206).
auto(kadisco).
persona(carlos).
persona(roque).

% genero
leVienePerfecto(Auto,Persona):-
       auto(Auto), persona(Persona),
       forall(quiere(Persona, Caracteristica), vieneCon(Auto, Caracteristica)).
```

Volviendo al ejemplo de los tipos, el primer argumento no es cualquier argumento, es un auto (cumple el hecho de ser auto), el segundo no puede ser cualquier cosa, tiene que ser una persona.

Una vez más repasamos:

- ciertos predicados escritos en forma natural no son inversibles por distintos motivos: gracias al concepto de **Generación** zafamos de esa limitación mediante la técnica de resolver las variables que están sin ligar
- a qué se debe la limitación: a que hay un motor imperfecto porque no es computacionalmente posible tener todos los hechos en la base de conocimiento. (relacionado con el Principio de Universo Cerrado).

[3] Recordemos que si no ligo las variables el forall es una afirmación general: para todas las personas que quieren algo, existe algún auto que viene con eso.

Explosión combinatoria Ejercicio: Sooooomos los piratas

El capitán Barbazul tiene muchos tripulantes posibles para partir en su barco pirata y nos pidió que lo ayudaramos para ver las posibles tripulaciones que podría llevar. Se conocen los siguientes posibles tripulantes, modelados en base a los siguientes hechos:

```
mujer(betina).
mujer(laura).
mujer(carola).
bonita(laura).
bonita(betina).
cocinera(laura).
pirata(felipe, 27).        /*nombre, edad*/
pirata(marcos, 39).
pirata(facundo, 45).
pirata(tomas, 20).
pirata(gonzalo, 22).
bravo(tomas).
bravo(felipe).
bravo(marcos).
```

La primera premisa que tiene el capitán es que la ferocidad total de sus tripulantes sea de al menos 10, sabiendo que:

- Las mujeres no cocineras tienen una ferocidad de 2.
- Las mujeres cocineras tienen una ferocidad de 4 (porque manejan cuchillos).
- Los piratas bravos tienen una ferocidad de 5.
- Los piratas cobardes, que son aquellos que no son bravos, tienen una ferocidad de 0.

Sólo puede haber mujeres o piratas: los hombres comunes no son del interés del capitán. Pero no es sólo lo anterior lo que se requiere, ya que a Barbazul solo le interesan las mujeres bonitas o cocineras, los piratas bravos y los piratas de más de 40 años.

Podríamos modelar la ferocidad de los posibles tripulantes de la siguiente forma:

```
ferocidad(Persona, 4):-mujer(Persona), cocinera(Persona).
ferocidad(Persona, 2):-mujer(Persona), not(cocinera(Persona)).
ferocidad(Persona, 5):-pirata(Persona, ), bravo(Persona).
ferocidad(Persona, 0):-pirata(Persona, ), cobarde(Persona).
cobarde(Persona):- not(bravo(Persona)).
```

Obviamente, no nos va a servir que le digamos que una tripulación está compuesta por Tomás, Tomás, Tomás, Tomás y Tomás: Tenemos que asegurarnos que cada tripulante aparezca una única vez si no queremos que Barbazul nos tire a los tiburones.
Pero, ¿cómo podemos llegar a eso? Conociendo el conjunto de posibles personas (piratas o mujeres) que pueden ser considerados.

```
personasPosibles(Personas):- findall(Persona, personaPosible(Persona), Personas).
```

¿Y qué es una “persona posible”? Por lo que dijimos antes, puede ser una mujer o un pirata. Lo podemos modelar así, entonces:

```
personaPosible(Persona)mujer(Persona). personaPosible(Persona) :- pirata(Persona, ) .
```

Todo muy bien, todo muy lindo... pero falta armar la tripulación en sí. No me alcanza con conocer el conjunto de personas, porque el conjunto completo es sólo UNA tripulación posible. Necesitamos conocer cada subconjunto de ese conjunto de personas. OK. La parte fácil:

```
armarTripulacion(Tripulacion, Cant):- personasPosibles(Personas), tripulacion(Personas,
      Tripulacion), cumpleFerocidadTotal(Tripulacion).

cumpleFerocidadTotal(Tripulacion):-
      maplist(ferocidad, Tripulacion, Ferocidades),
      sum(Ferocidades, Total),
      Total > 10.
```

Por último, ¿cómo hacemos “cada tripulación posible”? Pensémoslo en partes. Siempre podemos elegir no llevar a nadie (cláusula 1). También podemos elegir llevar a alguien, siempre que ese alguien cumpla alguno de los criterios del capitán (2). Por último, podemos elegir NO llevar a alguien, sin importar si cumple o no cumple los criterios.

```
tripulacion([], []).                                              /* 1 */
tripulacion([Posible|Posibles], [Posible|Tripulantes]):-          /* 2 */
cumple(Posible), tripulacion(Posibles, Tripulantes).
tripulacion([ |Posibles], Tripulantes):- tripulacion(Posibles,    /* 3 */   Tripulantes).

cumple(Persona):-mujer(Persona),    bonita(Persona).
cumple(Persona):-mujer(Persona),    cocinera(Persona). cumple(Persona):-pirata(Persona, ),
bravo(Persona).
cumple(Persona):-pirata(Persona,    Edad), Edad > 40.
```

El efecto que conseguimos es hacer una explosión combinatoria, obtener todas las combinaciones posibles de tripulantes