Paradigma Lógico

Módulo 4:
Predicados de orden superior

por Fernando Dodino
Germán Leiva
Carlos Lombardi
Mariana Matos
Nicolás Passerini
Daniel Solmirano

Versión 2.0
Diciembre 2016

Contenido

1 Predicado not/1

Hasta el momento hemos definido predicados que trabajan con individuos simples o compuestos:

```
? humano(socrates).
? par(2).
? longitud([1, 2, 3], 3).
? nacio(claudio, fecha(26, 11, 1975)).
```

Ahora tenemos la siguiente base de conocimientos de apostadores de ruleta, donde relacionamos los números que juega cada apostador:

```
juega(julia, 3).
juega(beto, 6).
juega(dodain, 5).
juega(juana, 15).
juega(sergio, 3).
```

Y sabemos que un número es yeta si nadie apuesta a él.
`∄x / p(x) se cumple`

Puesto en términos de Prolog:

```
yeta(Numero):-not(juega(_, Numero)).
```

¿Qué aridad tiene el predicado not? → 1
Pero el argumento que recibe... es un predicado.
Entonces es un **predicado de orden superior**.

1.1 Consultas

Podemos intentar algunas consultas

```
?- yeta(3).
false.

?- yeta(25).
true.
```

¿Qué pasa cuando intentamos una consulta existencial?

```
?- yeta(Numero).
false.
```

La consulta no puede satisfacerse. ¿Por qué?

```
yeta(Numero):-not(juega(_, Numero)).
```
[1]

Nosotros definimos en la base de conocimientos qué números juega la gente, pero no le decimos a Prolog cuál es el universo de números que podemos apostar a la ruleta. Y el motor de inferencia de Prolog solo está preparado para encontrar cuáles son todos los individuos que satisfacen una relación, no los que no la satisfacen.

1.2 Inversibilidad

Si queremos que el predicado sea inversible, debemos ligar todas las variables del o de los predicados involucrados en el not/1. En nuestro caso solo tenemos que hacerle saber al motor el universo de números de la ruleta.

```
yeta(Numero):-numeroRuleta(Numero), not(juega(_, Numero)).
```

Los números posibles son del 1 al 36. Para eso nos valemos de un predicado que relaciona dos números con un tercero que se encuentra en el intervalo cerrado entre ellos.

```
numeroRuleta(Numero):- between(1, 36, Numero).
```

numero/1 es un predicado **inversible**, esto significa que admite tanto consultas sí/no como consultas existenciales:

```
? numeroRuleta(1).
true

? numeroRuleta(37).
false

? numeroRuleta(X)
X = 1 ;
X = 2 ;
...
```

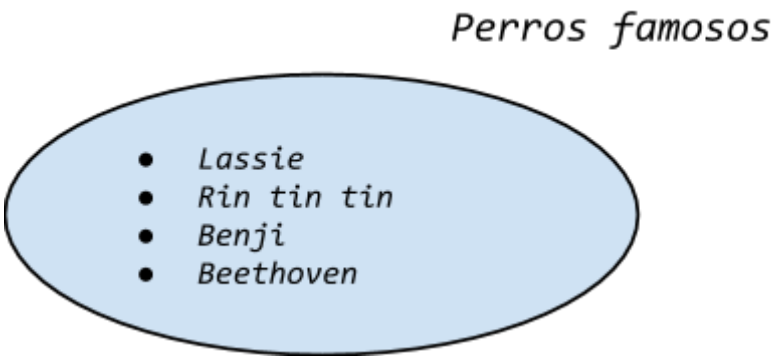
Y de esa manera también podemos admitir consultas existenciales sobre los números yeta, porque `numero/1` funciona como **predicado generador** del universo de números que se pueden apostar a la ruleta:

[1] Recordemos que en rojo marcamos aquellas variables que pueden quedar sin ligar en una consulta con variables

```
?- yeta(Cual).
Cual = 1 ;
Cual = 2 ;
Cual = 4 ;
Cual = 7 ;
...
```

1.3 Repaso de inversibilidad con not/1

Si tengo el Universo de los perros famosos que hicieron películas:



Y recordemos que todo lo que no está en la base de conocimientos no existe.

Entonces, ¿cómo sé cuáles son los perros que no son famosos si no conozco los perros que forman parte de mi universo? No puedo pedirle al motor que resuelva:
`perroComun(Perro):-not(perroFamoso(Perro)).`
si la variable Perro no está unificada.

Puesto en términos lógicos: si yo busco ∄x / `perroFamoso(x) se cumple`
¿de dónde sale el universo de x posibles?

Puedo preguntar perroComun(lassie) o perroComun(sultan), pero no puedo preguntar perroComun(Perro). Por eso el predicado perroComun no es inversible. ¡Sólo sabe los perros famosos, no los comunes!

Solución 1)

```
perroComun(Perro):-perro(Perro), not(perroFamoso(Perro)).
```

O sea, si tengo el universo de perros y el universo de perros famosos, puedo determinar si un perro es común o famoso.

Solución 2) Agregar el complemento de los perros famosos definiendo perroComun como un hecho. De esa manera transformo un predicado negativo en uno asertivo:

```
perroComun(chicho).
perroComun(sultan).
perroComun(copito).
```

En ambos casos la dificultad es que no siempre conozco el conjunto universal.

1.4 Not es de aridad 1

Si nuestra intención es negar la ocurrencia de dos predicados:

¬ (p ∧ q)

debemos hacerlo utilizando paréntesis, para mantener la aridad del not:

```
?- not(10 > 3, juega(dodain, 5)).
ERROR: Undefined procedure: not/2
ERROR:     However, there are definitions for:
ERROR:         not/1
false.

?- not((10 > 3, juega(dodain, 5))).
false.
```

2 Predicado forall/2

Queremos verificar 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, 7).
nota(nicolas, sintaxis, 8).
nota(malena, pdp, 6).
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 >= 6)).
```

o mejor

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

aprobo(Alumno, Materia):-nota(Alumno, Materia, Nota), Nota >= 6.
```

Algunas consultas:

```
?- terminoAnio(nicolas, 2).
true

?- terminoAnio(malena, 2).
false

?- terminoAnio(raul, 2).
false
```

2.1 Forall y la inversibilidad

```
?- terminoAnio(Alumno, 2).
```

no es inversible, porque esto implica que al tratar de unificar los individuos que cumplen la consulta, el predicado forall queda unificado así:

```
... :- forall(materia(Materia, 2), aprobo(Alumno, Materia))
```

```
∀x / p(x) ⇒ q(x)
```

¿Cuáles son las x? Las incógnitas son Alumno y Materia, entonces forall se satisfará si **todos** los alumnos ha terminado segundo año (es decir, **todas** las materias de segundo año). ¿Qué puedo hacer para que el forall haga la pregunta por cada alumno? 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 (la definición es por comprensión):

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

Entonces ahora 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 hacer la consulta `termino(Alumno, Anio)`?
¿Qué va a tratar de hacer el forall?

Y... como está armado el predicado va a preguntar si algún alumno que encontré aprobó **todas** las materias de **todos** los años, o sea si se recibió.

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

> ∀x,y / p(x,y) ⇒ q(x,y)
> x = Materia
> y = Anio
> Todas las incógnitas que participan del forall hacen que se busque que todos los individuos que cuantifiquen esas variables satisfagan los predicados p y q.

¿Qué puedo hacer para que forall pregunte si un alumno aprobó **todas** las materias de **un** año? Nos aseguramos que tanto el alumno como el año estén unificados antes de relacionarlos con el predicado forall.

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

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). Dentro de las reglas del negocio, Materia sí debe ser una incógnita, ya que si lo ligamos antes del forall estaremos preguntando si un alumno de un año aprobó una materia (que no es lo que nos piden).

Dejamos la codificación del predicado anio al lector.

¿Qué ocurre si dejamos nuevamente sin fijar el dominio para el año e intentamos hacer la siguiente consulta?

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

?- terminoAnio(nicolas, 5).
True.
```

¿Cómo puede ser que obtengamos un True? Repasemos la tabla de la verdad de ∀x / p(x) => q(x)

p	q	p => q
True	True	True
True	False	False
False	True	True
False	False	True

Esto ocurre porque cuando p(x) es falso, siempre se satisface el predicado sin importar el valor que tome q(x). Es decir que `materia(Materia, 5)` es **False** y en consecuencia el predicado es **verdadero**. ¿Y qué ocurre si agregamos nuevamente un predicado generador para fijar el dominio de Materia y realizamos la misma consulta?

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

?- terminoAnio(nicolas, 5).
False.
```

Y es debido a que el predicado `anio(5)` es falso y en consecuencia el predicado `terminoAnio` también.

2.2 Segundo ejemplo: de autos y personas

En este caso 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).                    quiere(carlos, abs).
vieneCon(p206, levantavidrios).         quiere(carlos, mp3).
vieneCon(p206, direccionAsistida).      quiere(roque, abs).
vieneCon(kadisco, abs).                 quiere(roque, direccionAsistida).
vieneCon(kadisco, mp3).
vieneCon(kadisco, tacometro).

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) relacione si **un auto concreto** tiene **todo** lo que **una persona concreta** quiere[2].

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).

leVienePerfecto(Auto, Persona):- auto(Auto), persona(Persona),
     forall(quiere(Persona, Chiche), vieneCon(Auto, Chiche)).
```

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

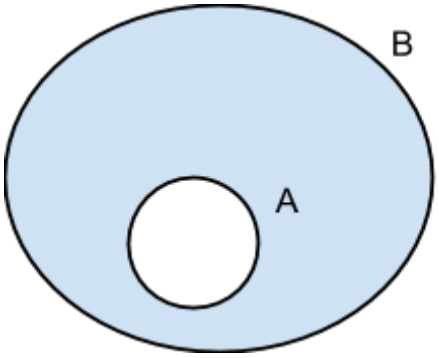
[2] Recordemos que si no ligo las variables el forall es una afirmación general: para todas las personas que quieren algo, existe algún auto que viene con eso

- 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 Ejercicios combinados: lógica de conjuntos

3.1 Incluido

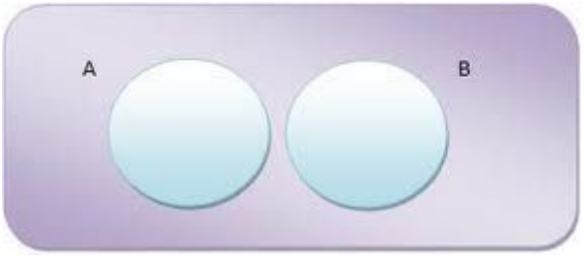
Un conjunto A está incluido en otro B si todos los elementos de A están en B.




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

3.2 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))).
```

4 Del not al forall

Si todos los individuos que cumplen p también cumplen q, esto es similar a decir que no es cierto que exista un individuo que cumpla p que no cumpla q.

En términos lógicos:
Si $\forall\ p(x) \Rightarrow q(x)$, esto equivale a $\nexists\ x\ /\ p(x) \wedge \neg q(x)$

Y si para todos los individuos que cumplen p ninguno cumple q, esto es lo mismo que decir que no es cierto que exista un individuo que cumpla p y q a la vez.

Si $\forall$ p(x) $\Rightarrow$ ¬q(x) , esto equivale a $\nexists$ x / p(x) $\wedge$ q(x)

Veámoslo con un ejemplo concreto.

```
materia(am1).
materia(sysop).
materia(pdp).
alumno(clara).
alumno(matias).
alumno(valeria).
alumno(adelmar).
```

```
estudio(clara, am1).
estudio(clara, sysop).
estudio(clara, pdp).
estudio(matias, pdp).
estudio(matias, am1).
estudio(valeria, pdp).
```

Un alumno estudioso es aquel que estudia para todas las materias...

```
estudioso(Alumno):-alumno(Alumno),
    forall(materia(Materia), estudio(Alumno, Materia)).
```

Repasando la definición lógica de forall: $\forall$ p(x) $\Rightarrow$ q(x)
¿Quiénes hacen de incógnita? Las materias, yo quiero probar si **un** alumno estudia para **todas** las materias. Por eso ligo la incógnita Alumno mediante el predicado generador alumno/1. De esa manera el predicado estudioso es inversible (la comprobación queda a cargo del lector).

Por otra parte, si “un alumno estudioso es aquel que estudia para todas las materias” eso significa que “no es cierto para un alumno estudioso que exista alguna materia en la que no haya estudiado”

$\nexists$ x / p(x) $\wedge$ ¬q(x)

```
estudiosoNot(Alumno):-alumno(Alumno),
    not((materia(Materia), not(estudio(Alumno, Materia)))).
```

Las definiciones son equivalentes, pero ¿cuál es más fácil de leer, más **expresiva**? En la segunda opción se introducen más dificultades para entender qué hace el predicado

- ya que tengo un not anidado
- y los paréntesis dificultan leer el contexto sobre el cual aplica cada not

Si por el contrario quiero buscar qué alumnos son difíciles, descriptos como los alumnos que no estudian para ninguna materia, aquí el lector podría considerar la utilización del not:

```
dificil(Alumno):-alumno(Alumno), not(estudio(Alumno, _)).
```

5 Findall

5.1 Introducción

Si nuestra base de conocimiento es

```
padre(homero,bart).
padre(homero,maggie).
padre(homero,lisa).
```

Podemos hacer varias consultas: ¿es cierto que homero es padre de bart?

```
?- padre(homero,bart).
true
```

¿Es cierto que homero es papá?

```
?- padre(homero, _).
true
```

¿Quiénes son los padres de bart?

```
?- padre(Padre, bart).
Padre = homero _
```

¿Quiénes son los hijos de homero?

```
?- padre(homero, Hijo).
Hijo = bart; ...
```

¿Quienes cumplen la relación padre / hijo?

```
?- padre(Padre, Hijo).
Padre = homero, Hijo = bart ; % etc.
```

Pero a pesar de esta gran gama de consultas hay ciertas preguntas que se vuelven complicadas o imposibles. Vamos a tomar como ejemplo la siguiente pregunta: ¿cuántos hijos tiene homero?

En el estado actual de las cosas tenemos que hacer la consulta

```
?- padre(homero,Hijo).
```

Contamos la cantidad de respuestas, en este caso los posibles valores de Hijo y obtenemos la respuesta … 3. Esto es impracticable cuando el número de respuestas es alto y además no responde a nuestra pregunta de forma directa.

Pensemos en predicados e individuos y definamos un predicado que relacione lo que nos interesa … una persona y su cantidad de hijos. Dicho predicado puede llamarse cantidadDeHijos/2.

```
?- cantidadDeHijos(homero, Cantidad).
Cantidad  = 3
```

Perfecto, ya tenemos definido nuestro objetivo ahora definamos el predicado a través de una regla (lo quiero hacer por comprensión porque quiero que me sirva para cualquier persona, no solo homero):

```
cantidadDeHijos(Padre, Cantidad) :-
      % acá va la magia que le da valor a la variable Hijos,
      length(Hijos, Cantidad).
```

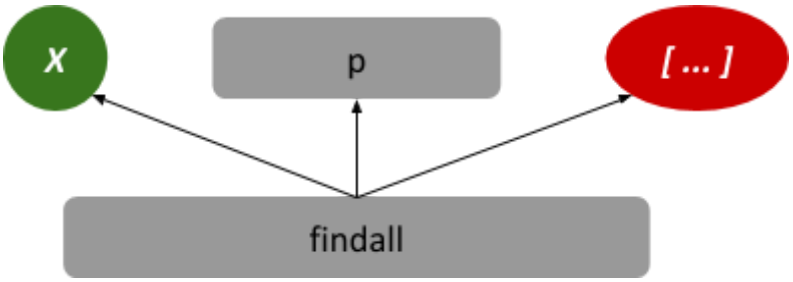
¡Nos falta hacer la magia! ¿Cómo obtener todas las respuestas 'juntas'?

Si tenemos una forma de obtener múltiples respuestas a una consulta y lo que queremos es que todas esas respuestas estén juntas en una lista, hay un predicado en Prolog que hace exactamente eso:

```
findall(UnIndividuoOVariable, Consulta, Conjunto)
```

Entonces: findall es un predicado que relaciona

- un individuo o variable
- con una consulta
- y con el conjunto (lista) de los individuos que satisfacen la consulta.




```
cantidadDeHijos(Padre, Cantidad) :-
      findall(Hijo, padre(Padre, Hijo), Hijos),
      length(Hijos, Cantidad).
```

Con esta definición nuestro objetivo se cumple

```
?- cantidadDeHijos(homero, Cantidad).
Cantidad = 3
```

Observamos que la variable de la cláusula que define cantidadDeHijos llega ligada al findall, por lo tanto la consulta que está adentro (el 2do parámetro) efectivamente tiene 3 respuestas.

5.2 Inversibilidad del predicado findall

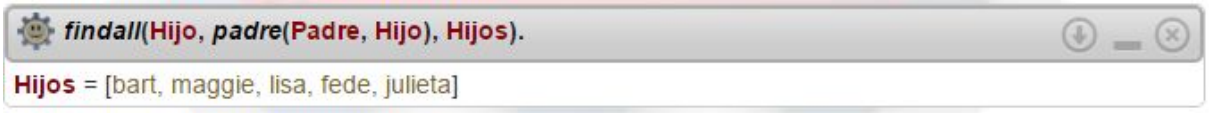
Ahora bien, agreguemos algunos hechos a la base de conocimiento

```
padre(homero,bart).
padre(homero,maggie).
padre(homero,lisa).
padre(juan, fede).
padre(nico, julieta).
```

Y pedimos otra consulta

```
?- cantidadDeHijos(Padre, Cantidad).
Cantidad = 5.
```

No funciona como esperábamos, ¿a qué se debe?
Debemos mirar la consulta que se está realizando en el findall (el 2do parámetro):



Si pensamos cuántas respuestas tiene esa pregunta veremos que ¡efectivamente son 5!

¿Cómo hacemos que el predicado sea inversible? Recordemos las palabras de un hombre sabio...

Ahora sí, sabemos que la variable Padre tiene que estar unificada, porque findall no lo liga:

```
cantidadDeHijos(Padre, Cantidad) :-
      persona(Padre), % Generación, así la variable Padre
                      % llega ligada al findall
      findall(Hijo, padre(Padre, Hijo),Hijos),
      length(Hijos, Cantidad).
```

Otra forma de generar el universo posible es considerar no a las personas, sino a los que son padres, a partir del predicado padre (me interesa que sea padre y no me importa en principio cuáles son sus hijos).

```
cantidadDeHijos(Padre,Cantidad) :-
      padre(Padre, _), % Generacion, asi la variable Padre
                       % llega ligada al findall
      findall(Hijo, padre(Padre, Hijo),Hijos),
      length(Hijos, Cantidad).
```

La diferencia está en la consulta:

```
?- cantidadDeHijos(bart, Cantidad).
```

Si consideramos que el universo de posibles padres son todas las personas, nos dice que bart tiene cero hijos:

```
?- cantidadDeHijos(bart, Cantidad).
Cantidad = 0
```

Con la segunda solución propuesta, bart no es una posible respuesta para (porque no es padre de nadie). Entonces el resultado es diferente:

```
?- cantidadDeHijos(bart, Cantidad).
false
```

En este caso consideramos que es más saludable un 0 que un false. Independientemente de eso, lo que debe quedar de todo esto es que las distintas formas de generar nos pueden dar diferentes resultados como respuesta y hay que elegir qué queremos.

Una pregunta adicional que podría surgir es: ¿qué pasa si no tenemos el predicado persona? Bueno, habrá que agregarlo a la base de conocimientos, y para ello tenemos dos posibilidades:

- **Por extensión:** uno por uno enumerando cada persona (un hecho para cada persona:

```
persona(bart).
persona(lisa).  % ... etc.
```

- **Por comprensión:** con una regla descubrir quiénes podemos considerar persona a partir de la información que ya tenemos. Una forma de hacer eso sería:

```
persona(Papa) :- padre(Papa, _).
persona(Hijo) :- padre(_, Hijo).
```

Es decir, el que es padre de alguien es una persona, y el que es hijo también.

5.3 Consultas más complejas

En el segundo parámetro del findall se puede definir cualquier tipo de consulta, no es necesario involucrar un único predicado.

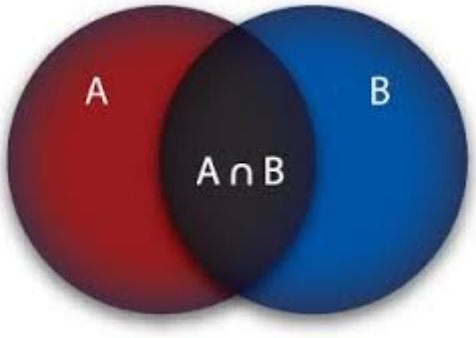
```
findall(X,(p(X),q(X),r(X),...,s(X)), Xs)
```

Solo hay que encerrar entre paréntesis las cláusulas para no cambiar la aridad de findall que es tres (3).

Ejemplo: si queremos hacer un predicado que me diga cuántos hijos varones tiene una persona, los X que me interesan son los que cumplen la consulta `(padre(Padre, Hijo), varon(Hijo))`

```
cuantosPibes(Persona,Cantidad):-
      persona(Persona),
      findall(Hijo, (padre(Persona, Hijo), varon(Hijo)), Varones),
      length(Varones, Cantidad).
```

5.4 Intersección de conjuntos




```
interseccion(Xs,Ys,Zs) :-
      findall(E,(member(E,Xs),member(E,Ys)),Zs).
```

5.5 Niños y juegos

Contamos con una base de conocimientos con la siguiente información en un jardín de infantes:

```
-- jugoCon/3: nene, juego, minutos
jugoCon(tobias, pelota, 15).
jugoCon(tobias, bloques, 20).
jugoCon(tobias, rasti, 15).
jugoCon(tobias, dakis, 5).
jugoCon(tobias, casita, 10).
jugoCon(cata, muniecas, 30).
jugoCon(cata, rasti, 20).
jugoCon(luna, muniecas, 10).
```

Queremos saber

- cuántos minutos jugó un nene según la base de conocimientos
- cuántos juegos distintos jugó (no hay duplicados en la base)

Una vez que tengo la lista de juegos que jugó un nene, puedo contarlos o sumar los minutos que jugó cada uno:

```
juegosQueJugo(Nene, CantidadJuegos):-
    findall(Juego, jugoCon(Nene, Juego, _), Juegos),
    length(Juegos, CantidadJuegos).

?- juegosQueJugo(tobias, N).
N = 5.

minutosQueJugo(Nene, CuantosMinutos):-
```

```
    findall(Minutos, jugoCon(Nene, _, Minutos), ListaMinutos),
    sumlist(ListaMinutos, CuantosMinutos).

?- minutosQueJugo(cata, Minutos).
Minutos = 50.
```

De todas maneras el predicado no es inversible para el primer argumento:

```
?- minutosQueJugo(Nene, Minutos).
Minutos = 125.
```

porque... al no ligar la variable Nene estamos sumando los minutos que jugaron todos los chicos:

```
minutosQueJugo(Nene, CuantosMinutos):-
    findall(Minutos, jugoCon(Nene, _, Minutos), ListaMinutos),
```

Para eso, buscamos que haya un predicado que ligue la variable Nene:

```
minutosQueJugo(Nene, CuantosMinutos):-
    nene(Nene),
    findall(Minutos, jugoCon(Nene, _, Minutos), ListaMinutos),
```

Ahora sí podemos preguntar cuántos minutos jugó cada nene.

5.6 Usando individuos compuestos en el primer parámetro del findall

En ciertas situaciones nos interesa tener una lista de individuos que hasta el momento no existían en nuestro programa o que no están presentes explícitamente en la consulta (o sea, en el 2do parámetro del findall).

Ejemplo: Imagínense que tenemos una solución donde se define el predicado puntaje/2 que relaciona a un equipo con la cantidad de puntos que tiene. Un requerimiento bastante usual en un programa de este estilo, es conocer la tabla de posiciones que se puede ver como un conjunto de individuos o sea una lista en donde cada individuo que la compone es un equipo con su cantidad de puntos.

```
?- findall( ???? , puntaje(Equipo, CantidadPuntos), Tabla ).
```

La pregunta a responder es qué ponemos en ????. Necesitamos definir un individuo que relacione equipo y cantidad de puntos. Para hacer esto nada mejor que un functor:

```
?- findall( puntos(Equipo, CantidadPuntos) ,
            puntaje(Equipo, CantidadPuntos) ,
```

```
            Tabla )
```

Recuerden:

- puntos es un functor, no un predicado
- puntaje es un predicado, no un functor
- Tabla es una lista de functores puntos que verifican la consulta que está como segundo parámetro del findall

5.7 Ejercicio integrador

Consideremos esta base de conocimientos:

```
tiene(juan, foto([juan, hugo, pedro, lorena, laura], 1988)).
tiene(juan, foto([juan], 1977)).
tiene(juan, libro(saramago, "Ensayo sobre la ceguera")).
tiene(juan, bebida(whisky)).
tiene(valeria, libro(borges, "Ficciones")).
tiene(lucas, bebida(cusenier)).
tiene(pedro, foto([juan, hugo, pedro, lorena, laura], 1988)).
tiene(pedro, foto([pedro], 2010)).
tiene(pedro, libro(octavioPaz, "Salamandra")).

premioNobel(octavioPaz).
premioNobel(saramago).
```

Determinamos que alguien es coleccionista si todos los elementos que tiene son valiosos:

- un libro de un premio Nobel es valioso
- una foto con más de 3 integrantes es valiosa
- una foto anterior a 1990 es valiosa
- el whisky es valioso

Definimos entonces el predicado coleccionista:

```
coleccionista(Alguien):-
   forall(tiene(Alguien, Cosa), vale(Cosa)).
```

Vemos cómo funciona el predicado vale/1, para cualquier tipo de Cosa coleccionable:

```
vale(foto(Gente, _)):-length(Gente, Cantidad), Cantidad > 3.
vale(foto(_, Anio)):-Anio < 1990.
vale(libro(Escritor, _)):-premioNobel(Escritor).
vale(bebida(whisky)).
```

Y esto nos permite consultar:

```
? - coleccionista(pedro).
false.

?- coleccionista(juan).
true.
```

Aunque no es válido preguntar:

```
?- coleccionista(Quien).
true.
```

porque Quien no se liga cuando llega al predicado forall/2, por lo que quiere verificar que todas las personas tienen todos los elementos valiosos.

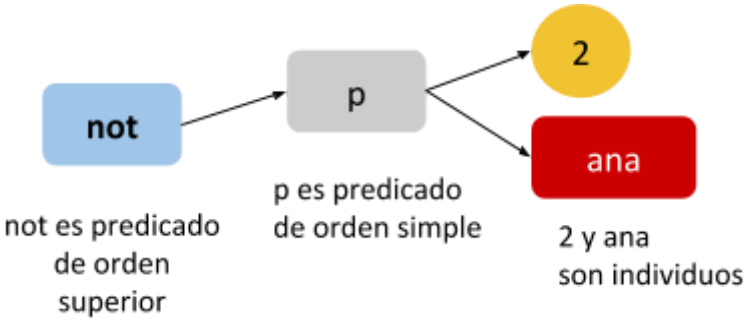
```
coleccionista(Alguien):-
   forall(tiene(Alguien, Cosa), vale(Cosa)).
```

Para resolverlo, debemos utilizamos un predicado generador sobre Alguien, cuya solución queda a cargo del lector.

```
coleccionista(Alguien):-
   tiene(Alguien, _),
   forall(tiene(Alguien, Cosa), vale(Cosa)).
```

6 Resumen

Mientras que los predicados de primer orden definen características o relacionan individuos entre sí, los predicados de orden superior trabajan con predicados como argumentos. En particular, conocimos not/1, forall/2 y findall/3 que son predicados que permiten subir el grado de abstracción:



Mientras que el predicado forall/2 trabaja en base a la cuantificación universal: “para todos se cumple”, not/1 trabaja en base a la cuantificación individual negativa: “no existe x tal que cumpla…”, por lo tanto son predicados que pueden

usarse en forma indistinta aunque trabajar con afirmaciones asertivas suele ser más cómodo que hacerlo negando los predicados.

Por último, para tener predicados inversibles se necesita ligar las variables que participan de las consultas mediante predicados generadores:

- en la negación lógica, porque no es posible en Prolog determinar los individuos que no satisfacen un predicado si no conocemos todo el universo.
- en las cláusulas findall y forall, porque debemos estar atentos a las variables libres y ligadas que participarán dentro de dichas cláusulas.