Ejercicio: Salidas posibles (findall/3 + explosión combinatoria)

En la base de conocimientos se registraron distintas salidas o actividades y su respectivo costo:

```
actividad(cine).
actividad(pearlJam).
actividad(holidayOnIce).
actividad(pool).
actividad(bowling).

costo(cine, 50).
costo(pearlJam, 700).
costo(holidayOnIce, 500).
costo(pool, 20).
costo(bowling, 50).
```

Se pide implementar el siguiente predicado:
`actividades(Plata, ActividadesPosibles)`
que nos indique las distintas alternativas que tenemos dada una determinada cantidad de plata.

Tenemos que prestar atención a la forma en que se está pidiendo: “dada una cantidad de plata” se piden “las actividades posibles”… necesitamos que sea **inversible para la lista**.

Ejemplo de consulta:

```
?- actividades(100, X).
X = [cine] ;
X = [cine, pool] ;
X = [cine, bowling] ;
X = [pool] ;
X = [pool, bowling] ;
X = [bowling] ;
X = [] ;
```
← me quedo en casa y no gasto nada

Tenemos las actividades en predicados aislados, con el predicado findall conseguimos la lista de actividades posibles:

```
actividades(Plata, ActividadesPosibles):-
   findall(Actividad, actividad(Actividad), ListaActividades),
   actividadesPosibles(ListaActividades, Plata, ActividadesPosibles).
```

Cuando no tenemos actividades, no hay actividades posibles:

```
actividadesPosibles([] , _ , []).
```

Una actividad posible va a formar parte del conjunto solución si tengo plata suficiente. El resto de las actividades posibles se relacionará con la plata que me quede tras hacer esa actividad:

```
actividadesPosibles([Actividad|Actividades], Plata, [Actividad|Posibles]):-
   costo(Actividad, Costo), Plata > Costo, PlataRestante is Plata - Costo,
   actividadesPosibles(Actividades, PlataRestante, Posibles).
```

Y si decido no hacer esa actividad, tendré la misma plata para las actividades restantes:

```
actividadesPosibles([_|Actividades], Plata, Posibles):-
   actividadesPosibles(Actividades, Plata, Posibles).
```

En la última cláusula puede pasar que no haga la actividad porque:

1) no tengo plata para hacer esa actividad o bien
2) tengo plata para hacer esa actividad pero no me interesa.

El efecto que tiene es que hace una **explosión combinatoria** con todas las soluciones posibles, siempre que no me pase del presupuesto. Eso incluye: ahorrarme la plata y quedarme en casa o ir al cine y no gastar "tanto", ir al cine y a ver a Pearl Jam (y gastar más pero tener más salidas), etc. Todos esos criterios se combinan para dar el árbol de soluciones posible.

Predicados de orden superior: forall/2

"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? Veamos... si yo consulto:

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

Lo que PROLOG hace es afirmar que para todas las materias de 2º 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),  ← para un alumno dado…
      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)).
```

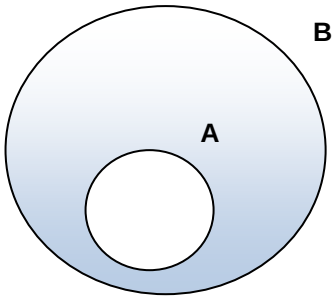
Fíjense que lo que estamos haciendo es fijar un dominio para los argumentos: el primer argumento no es cualquier individuo, tiene que ser un Alumno, el segundo tiene que ser un año de la carrera (no cualquier número).

Dejamos la codificación del predicado anio al lector.

Algunos ejercicios

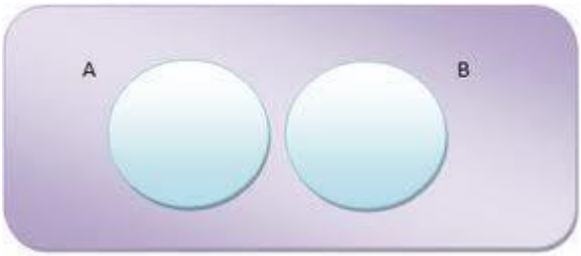
Podemos aplicar el predicado `forall/2` para resolver dos ejercicios de lógica de conjuntos:

- **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:

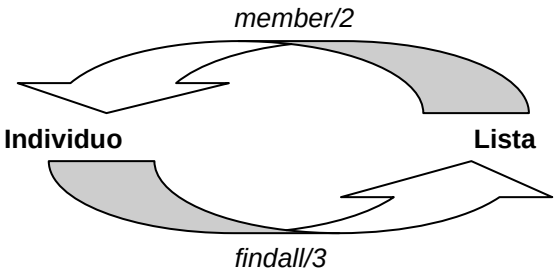
```
interseccion(A, B, C):-
   findall(X, (member(X, A), member(X, B)), C).
```

Inversibilidad en member/2

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).



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.

Límites a la inversibilidad – parte II[2]

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

- **Negación**: predicado not/1
- **Operaciones aritméticas**: is (+, -, *, /)
- **Comparación**: >, <, =<, >=

Agregamos entonces dos casos más:

- Predicado findall/3
- Predicado forall/2

En los cinco casos podemos utilizar predicados generadores.

Generación

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

```
esMalo(feinmann).
esMalo(hadad).
esMalo(echecopar).

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?

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

Predicado generador

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

[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

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 `findall/3`

Modelando un juego de truco tenemos estas cláusulas:

```
carta(juan, cuatro, copa).
carta(juan, siete, basto).
carta(juan, tres, oro).
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).
Palos = [copa, basto, oro] ;
```

Claro, `Palo` es una variable que está sin ligar antes del `findall/3`… dentro del findall varía para cada solución (cada ítem de la lista) y luego del findall queda nuevamente sin unificar, porque no tendría sentido unificarla arbitrariamente con uno cualquiera entre todos. Revisemos de nuevo la cláusula:

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

Esto está diciendo: "una persona tiene envido si tiene al menos 2 cartas (números) *de algún palo*". "Palo" en ese findall es una incógnita, entonces el findall intentará encontrar todas las soluciones que tendría la consulta con ese argumento como incógnita (además del número).

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

```
tieneEnvido(Persona):-
      palo(Palo), ← "para un Palo determinado…"
      findall(Numero, carta(Persona, Numero, Palo), Numeros),
      length(Numeros, Cantidad),
      Cantidad >= 2.
```

Ahora sí `Palo` está ligado en el momento del `findall/3`: 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/3` tengamos ligadas las variables que deban estarlo. Para eso nos ayudamos con predicados generadores.

Predicados generadores con `forall/2`

Tomemos este caso, donde debemos resolver si un auto le viene perfecto a una persona, donde "le viene perfecto" significa que 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) 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:

[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

```
% 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).