LÓGICA

1º AÑO

Clase N.º 7: Lógica de Predicados.

Contenido: **Reglas recursivas, Variable anónima y predicado predefinido.**

Hola, ¿Cómo están? Bienvenidas y bienvenidos a la clase Nº7 de Lógica. La clase anterior realizamos un breve recorrido sobre la programación lógica, y luego nos centramos en Prolog, que es el programa con el cual trabajaremos. Abordamos la composición de una base de conocimiento, vimos que la misma se compone por hechos (propiedades, relaciones) y reglas.
En la clase de hoy, vamos a ver un tipo de reglas, ¡veamos!

Supongamos que en una base de conocimiento tenemos definidos los hechos: padre y abuelo, y queremos definir una regla para "antepasado" ¿Podemos pensar en "progenitor"? Veamos:

```
antepasado(X, Y) :- progenitor(X,Y).
antepasado(X,Y) :- progenitor(X,Z), progenitor (Z,Y).
```

Estas reglas nos llevan hasta los abuelos, pero y ¿los bisabuelos? y ¿los tatarabuelos? Podríamos hacer algo así:

```
antepasado(X, Y) :- progenitor(X,Y).
antepasado(X,Y) :- progenitor(X,Z), progenitor (Z,Y).
antepasado(X,Y) :- progenitor(X, Z1), progenitor (Z1, Z2), progenitor (Z2, Y).
antepasado(X,Y) :- progenitor(X, Z1), progenitor (Z1, Z2), progenitor (Z2, Z3),
progenitor (Z3, Y).
```

Pero ¿si queremos encontrar a nuestros antepasados de la Edad Media? ¿Cuántas reglas tendremos que hacer? ¿Es necesario hacer muchas reglas? ¿Existe una forma más sencilla de no realizar muchas veces la misma regla, que a su vez son más largas aun teniendo los mismos hechos? ¡La buena noticia es que SI! Por suerte, podemos usar la recursividad ¿qué es la recursividad? Se llama recursividad a un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna determinada condición.

En este sentido, Prolog hace uso de la recursividad para tratar todos los problemas en los que tenemos que hacer la misma cosa un número **indeterminado** de veces. Esto se consigue definiendo a una función de manera tal que la función se contiene a sí misma en su propia definición.

Entonces veamos como se resuelve, definitivamente, la situación planteada al inicio de clase: encontrar a nuestros antepasados sin importar cuántas generaciones separen al descendiente Y del antepasado X:

```
antepasado(X, Y) :- progenitor(X,Y).
antepasado(X,Y) :- progenitor(X,Z), antepasado(Z,Y).
```

Veamos un ejemplo utilizando esta regla en concreto: queremos saber si `pepe` es antepasado de `lolo` dada la siguiente base de conocimiento y la regla para antepasados que recién vimos:

```
progenitor(pepe, esteban).
progenitor(esteban, julian).
progenitor(julian, raimundo).
progenitor(raimundo, lolo).
```

Veamos la traza de la pregunta `antepasado(pepe,lolo).`:

```
[trace]  ?- antepasado(pepe,lolo).

   Call: (8) antepasado(pepe, lolo) ? creep

   Call: (9) progenitor(pepe, lolo) ? creep          <== trata de aplicar
la primera regla de "antepasado"...

   Fail: (9) progenitor(pepe, lolo) ? creep          <== ... y falla

   Redo: (8) antepasado(pepe, lolo) ? creep          <== aplica la
segunda regla de "antepasado"

   Call: (9) progenitor(pepe, _L176) ? creep         <== busca satisfacer
la primera parte del cuerpo de la regla, progenitor(X,Z)

   Exit: (9) progenitor(pepe, esteban) ? creep       <== puede satisfacer
la primera parte del cuerpo de la regla, con X=pepe y Z=esteban

   Call: (9) antepasado(esteban, lolo) ? creep       <== busca satisfacer
la segunda parte del cuerpo de la regla, antepasado(Z,Y), para ello
aplicará reglas de "antepasado" con los valores de variable X=esteban y
Y=lolo

   Call: (10) progenitor(esteban, lolo) ? creep      <== trata de aplicar
la primera regla de "antepasado"...

   Fail: (10) progenitor(esteban, lolo) ? creep      <== ... y falla

   Redo: (9) antepasado(esteban, lolo) ? creep       <== aplica la
segunda regla de antepasado

   Call: (10) progenitor(esteban,Z) ? creep     <== busca satisfacer la
primera parte del cuerpo de la regla, progenitor(X,Z), donde ahora
X=esteban

   Exit: (10) progenitor(esteban, julian) ? creep  <== puede satisfacer
la primera parte del cuerpo de la regla, con X=esteban y Z=julián

   Call: (10) antepasado(julian, lolo) ? creep     <== busca satisfacer
la segunda parte del cuerpo de la regla, antepasado(Z,Y), para ello
aplicará reglas de "antepasado" con los valores de variable X=julián y
Y=lolo

   Call: (11) progenitor(julian, lolo) ? creep     <== trata de aplicar
la primera regla de "antepasado"...
```

```
   Fail: (11) progenitor(julian, lolo) ? creep     <== ... y falla

   Redo: (10) antepasado(julian, lolo) ? creep     <== aplica la segunda
regla de antepasado

   Call: (11) progenitor(julian, Z) ? creep    <== busca satisfacer la
primera parte del cuerpo de la regla, progenitor(X,Z), donde ahora
X=julian

   Exit: (11) progenitor(julian, raimundo) ? creep <== puede satisfacer
la primera parte del cuerpo de la regla, con X=julian y Z=raimundo

   Call: (11) antepasado(raimundo, lolo) ? creep     <== busca satisfacer
la segunda parte del cuerpo de la regla, antepasado(Z,Y), para ello
aplicará reglas de "antepasado" con los valores de variable X=raimundo y
Y=lolo

   Call: (12) progenitor(raimundo, lolo) ? creep     <== trata de aplicar
la primera regla de "antepasado"...

   Exit: (12) progenitor(raimundo, lolo) ? creep     <== ... y tiene
éxito!!! se cumple el cuerpo de la regla, por lo tanto

   Exit: (11) antepasado(raimundo, lolo) ? creep     <== podemos
satisfacer la cabeza: que raimundo es antepasado de lolo, con lo cual se
satisfizo el cuerpo de la regla anterior

   Exit: (10) antepasado(julian, lolo) ? creep     <== y podemos
satisfacer la cabeza: que julian es antepasado de lolo, con lo cual se
satisfizo el cuerpo de la regla anterior

   Exit: (9) antepasado(esteban, lolo) ? creep       <== y podemos
satisfacer la cabeza: que esteban es antepasado de lolo, con lo cual se
satisfizo el cuerpo de la regla anterior

   Exit: (8) antepasado(pepe, lolo) ? creep          <== y podemos
satisfacer la cabeza: que pepe es antepasado de lolo, con lo cual se
satisfizo nuestra pregunta inicial!

true ;
```

Aquí les dejamos un video que explica las Reglas recursivas, su utilidad y ejemplos:
https://www.youtube.com/watch?v=zukaF7OzULU&list=PLGfF3KgbxaiwLDxZaSuec2zxNZC7zmQkI&index=6

ACTIVIDAD

Teniendo en cuenta las siguientes relaciones de amistad:

- Juana es amiga de María.
- María es amiga de Rocio.
- Rocio es amiga de Ana.
- Ana es amiga de Louisa.
- Louisa es amiga de Carla.
- Carla es amiga de Sofía.
- Sofía es amiga de Celeste.
- Celeste es amiga de Laura.
- Laura es amiga de Johanna.

1.a Determina las relaciones para cada relación de amistad.
1.b Crea dos reglas, para determinar si alguien es amiga de alguien más, utilizando en una de ellas la recursividad.

Ahora bien, puede pasar -en ocasiones- que algunas reglas contengan un guión bajo "_" dentro de los objetos/personas, es decir, dentro del paréntesis.

Por ejemplo:

```
es_hijo(El,X) :- varon(El), tiene_por_padres(El,X,_).
es_hijo(El,X) :- varon(El), tiene_por_padres(El,_,X).
es_hija(Ella,X) :- mujer(Ella), tiene_por_padres(Ella,X,_).
es_hija(Ella,X) :- mujer(Ella), tiene_por_padres(Ella,_,X).
```

En el cual X representa, depende como esté ubicada, madre o padre. Pero, ¿qué expresan todas esas sentencias?:

Él es hijo de X si se verifica que él es varón y tiene por padres a X, y aquí aparece el guión bajo, ¿por qué? Porque hacemos foco solamente en el padre y no en la madre.
Fíjense en la siguiente: Él es hijo de X si se verifica que él es varón y tiene por padres a X. Entonces en esta sentencia, nos interesa saber si existe en el dominio la madre. Análogamente podemos observar lo mismo con las sentencias de hija.

Entonces una **variable anónima** se representa con el simbolo "_" y denota que en cada instancia de dicha variable se refiere a una variable distinta.

Considerando las siguientes relaciones familiares:

```
tiene_por_padres(eduardo,francisco,victoria).
tiene_por_padres(alicia,francisco,victoria).
tiene_por_padres(luis,eduardo,veronica).
tiene_por_padres(beatriz,mario,alicia).
es_hermana(Ella,X) :- mujer(Ella),tiene_por_padres(Ella,P,M),
tiene_por_padres(X,P,M).
es_hermano(El,X) :- varon(El), tiene_por_padres(El,P,M),
tiene_por_padres(X,P,M).
```

2.a Determina las propiedades que se esconden en las reglas.

2.b Crea las reglas para `es_hijo(El,X)` y `es_hija(Ella,X)` recurriendo a la variable anónima.

2.c Haz preguntas, en Prolog, para verificar si la base de conocimientos construida, es correcta

Por otro lado, en las bases de conocimiento que venimos trabajando, los predicados utilizados son elegidos por la persona que programa y que refieren, más bien, a características. Pero prolog presenta predicados predefinidos, ¿qué son los predicados predefinidos? Son aquellos que ya están definidos en Prolog, es decir, no necesitamos especificarlos. En la clase de hoy, trabajaremos solamente un tipo de predicado predefinido que permite controlar otros predicados, como ser el "not": El objetivo `not(X)` se cumple si fracasa el intento de satisfacer X. El objetivo `not(X)` fracasa si el intento de satisfacer X tiene éxito. Es similar a la negación en la lógica de predicados.

Por ejemplo, la regla `"es_mamifero(X) :- animal(X), not(pone_huevos(X))."` establece que X es un mamífero si es un animal y no pone huevos.

ACTIVIDAD

Teniendo en cuenta las siguientes listas:

- Omnívoros:
  - Oso
  - Mono
  - Ardilla
  - Rata
  - Cerdo
  - Mapache
  - Chimpancé
  - Hurón
  - Erizo
  - Ratón
- Carnívoros:
  - León
  - Tigre
  - Lobo
  - Pantera
  - Guepardo
  - Hiena
  - Zorro
  - Oso polar
  - Águila
  - Tiburón
- Herbívoros:
  - Vaca
  - Cabra
  - Oveja
  - Caballo
  - Ciervo
  - Jirafa
  - Conejo
  - Koala
  - Elefante
  - Canguro

a. Elabora hechos con los animales de la lista.

b. Establece relaciones teniendo en cuenta lo que come cada animal: si come carne y/o hierbas.

c. Por último, crea las reglas para carnívoro, herbívoro y omnívoro utilizando el predicado predefinido “not(X)”.

ACTIVIDAD

Elaborar un programa en Prolog que describa tu propio árbol genealógico que incluya las propiedades según el género de las personas (en este caso hombre/mujer) y las relaciones necesarias que permitan establecer las reglas de: abuelo/a, ancestro/a, tío/a, hijo/a. Por último, realiza, al menos, una consulta por cada regla.

Obs: La base de conocimiento debe contar con variable anónima y reglas recursivas.

En la clase siguiente compartiremos esta actividad.

Llegamos al fin de la clase 7, hemos visto reglas de recursividad, variable anónima y un tipo de predicado predefinido, el not. La clase siguiente avanzaremos un poco más con otros predicados predefinidos ¡las y los esperamos en la siguiente clase!