Paradigma Lógico

Módulo 5:
Recursividad

por Fernando Dodino
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano

Versión 2.1
Julio 2019

Contenido

1 Definición de recursividad

Recursividad es que un elemento de software se defina en función de sí mismo. La recursividad en lógico consistirá en definir un predicado en términos de sí mismo. En otras palabras:

¿Qué elementos tenemos en lógico?	Predicados / cláusulas.
¿qué es un programa lógico?	(OK, un conjunto de cláusulas que) definen relaciones entre individuos
entonces, ¿qué va a ser recursivo?	la definición de un predicado

2 Primer ejemplo: ancestros

Buscamos primero la forma de contarlo declarativamente:

¿Quiénes son mis ancestros?

- Mis padres
- y los ancestros de mis padres.

Observar que estoy definiendo ancestro en términos de ancestro.

Recordemos la diferencia entre el castellano y la lógica - el "y" en castellano funciona como "o" lógico:

```
ancestro(Padre, Persona):-padre(Padre, Persona).
ancestro(Ancestro, Persona):-
    padre(Padre, Persona),
    ancestro(Ancestro, Padre).
```

Esto se lee: “ancestro es mi padre o bien el ancestro de mi papá”.

Para esta base de conocimientos

```
padre(tatara, bisa).
padre(bisa, abu).
padre(abu, padre).
padre(padre, hijo).
```

Podemos hacer la consulta

```
?- ancestro(tatara, Quien).
Quien = bisa ;
Quien = abu ;
Quien = padre ;
Quien = hijo ;
false.
```

Con lo cual nuestra solución recursiva detecta relaciones a *n* niveles de profundidad.

Por otra parte, ¿es posible hacer esta consulta?

```
?- ancestro(Ancestro, Quien).
```

¿Qué conclusiones sacamos acerca de la inversibilidad del predicado ancestro/2?

3 Segundo ejemplo: distancias

En nuestra base de conocimientos sabemos la distancia que hay entre dos lugares:

```
distancia(buenosAires, puertoMadryn, 1300).
distancia(puertoMadryn, puertoDeseado, 732).
distancia(puertoDeseado, rioGallegos, 736).
distancia(puertoDeseado, calafate, 979).
distancia(rioGallegos, calafate, 304).
distancia(calafate, chalten, 213).
```

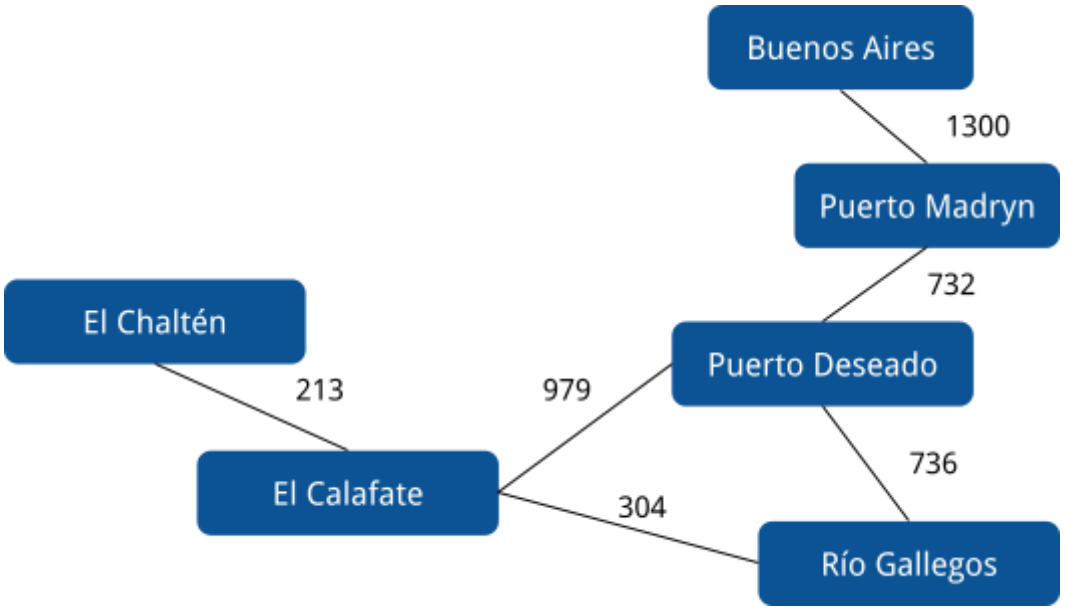
Podemos calcular los kilómetros de viaje entre dos puntos fácilmente:

- porque conocemos la distancia entre esos puntos
- porque conocemos la distancia con otro punto del cual sabemos los kilómetros de viaje con nuestro destino.

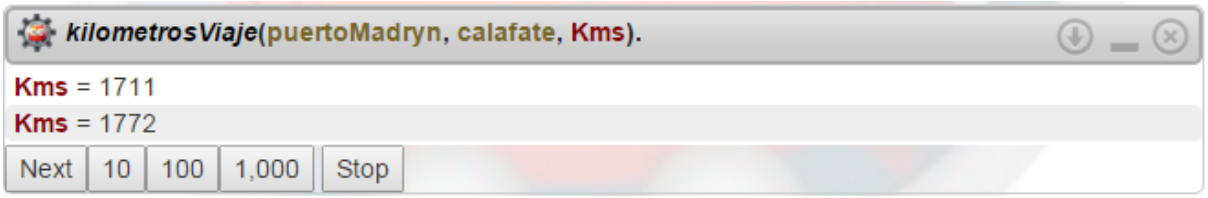
Una vez más tenemos una definición recursiva:

```
kilometrosViaje(Origen, Destino, Kms):-
     distancia(Origen, Destino, Kms).
kilometrosViaje(Origen, Destino, KmsTotales):-
     distancia(Origen, PuntoIntermedio, KmsIntermedios),
     kilometrosViaje(PuntoIntermedio, Destino, KmsFinales),
     KmsTotales is KmsIntermedios + KmsFinales.
```

El mapa que se genera no es lineal:



Y esto permite que Prolog encuentre todas las soluciones posibles para ir de Puerto Madryn a El Calafate:



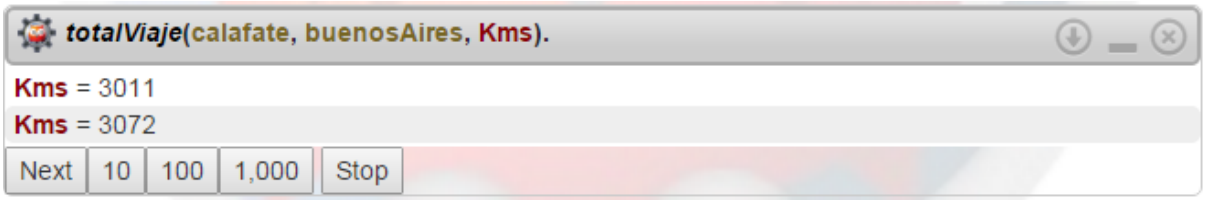
Solo un ajuste más, para poder calcular la distancia de El Calafate a Buenos Aires:

```
?- kilometrosViaje(calafate, buenosAires, Kms).
false
```

Tenemos que incorporar una predicado adicional:

```
totalViaje(Origen, Destino, Kms):-
   kilometrosViaje(Origen, Destino, Kms).
totalViaje(Origen, Destino, Kms):-
   kilometrosViaje(Destino, Origen, Kms).
```

Ahora sí podemos consultar la distancia de Calafate a Buenos Aires:



4 Predicado member/2

Un elemento está en la lista

- si está en la cabeza de la lista
- si está en la cola.

El chiste de esta definición es que como la cola es una lista, la definición del predicado member termina siendo recursiva:

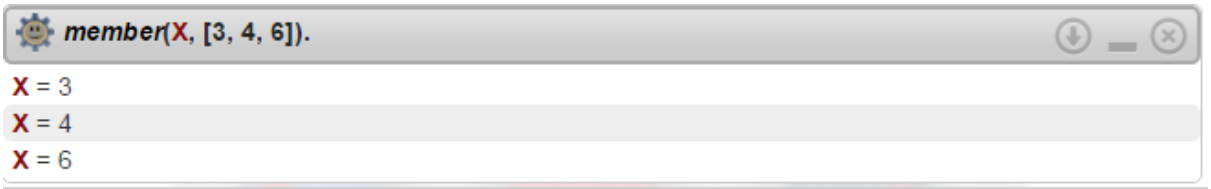
```
member(X, [X|_]).
member(X, [_|Z]):-member(X, Z).
```

¿Qué va a pasar si la lista está vacía?
Evidentemente no puede haber ningún elemento en una lista vacía, porque la lista vacía no es divisible en cabeza y cola, entonces no hago ningún tratamiento para lista vacía[1].

`?- member(5, [5, 8])` unifica con la primera cláusula
`?- member(5, [4, 6])` va a fallar, mientras que
`?- member(5, [4, 5])` se satisface porque 5 está en la cola de la lista (segunda cláusula).
Lo mismo pasará con:
`?- member(5, [4, 3, 5, 6])`. El 5 está en la cola de esta lista.

4.1 Inversibilidad del predicado member

Esta definición permite hacer consultas dejando libre el primer argumento:



[1] ¿Con qué concepto está relacionado esta idea? ¡Exacto! Principio de Universo Cerrado, lo que no se cumple no me interesa que quede en la base de conocimientos.

Por esto el predicado member/2 es inversible para el primer argumento. No tiene tanto sentido hacer una consulta existencial por el segundo argumento: ¿qué individuos satisfacen member(6, Lista)? Cualquier lista que contenga 6 entre sus elementos. Pero no sabemos cuál es el universo posible de esos elementos...

5 Predicado nth1/3

¿Cómo resolver el predicado nth1/3?

Recordemos algunas consultas que podemos hacer:

```
?- nth1(2, [1, 2, 3, 5, 8, 13], Elemento).
Elemento = 2

?- nth1(Posicion, [1, 2, 3, 5, 8, 13], 5).
Posicion = 4
```

nth1/3 o enesimo/3 relaciona la posición que ocupa un elemento en una lista, considerando 1 la primera posición.

- El caso base es cuando el elemento está en la cabeza, y se relaciona con la posición 1
- El caso recursivo es que si el elemento no está en la cabeza y está en la cola, entonces ocupará la posición 1 + la posición que tenga en la cola. Si no está en la cola, no me tengo que preocupar, no ocupa ninguna posición (nuevamente, principio de universo cerrado)

```
enesimo(1, [Elemento|_], Elemento).
enesimo(Posicion, [_|Resto], Elemento):-
      enesimo(PosicionCola, Resto, Elemento),
      Posicion is PosicionCola + 1.
```

6 Resumen

Prolog permite definir predicados recursivos, que deben tener

- un caso base o corte de la recursividad
- al menos un caso recursivo, donde el predicado esté definido en términos de sí mismo.

Un aspecto interesante del motor de inferencia Prolog es que permite encontrar múltiples soluciones también para este tipo de predicados, mientras no haya restricciones de inversibilidad.