Prolog: Listas | MRC

Víctor Peinado

14-15 de enero de 2016

Referencias

- (Blackburn, *et al.*, 2006, chap. 4) [1]
- (Clocksin & Mellish, 2003) [2]
- Curso de Programación Lógica de la UPV/EHU [3] [4]

[1] Blackburn, P., Bos, J., Striegnitz, K. *Learn Prolog Now!*. College Publications. Texts in Computer Science, vol 7. 2006. `http://www.learnprolognow.org/lpnpage.php?pageid=online`

[2] Clocksin, W., Mellish, C. S. *Programming in Prolog*. Springer Science & Business Media. 2003. `http://books.google.es/books?id=VjHk2Cjrti8C`

[3] Navarro, M. *Curso de Programación Lógica*. Tema 1. UPV/EHU. `http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema1.pdf`

[4] Navarro, M. *Curso de Programación Lógica*. Tema 2. UPV/EHU. `http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema2.pdf`

Listas

Una **lista** es una secuencia o colección ordenada y finita de términos. Los siguientes ejemplos válidos de listas en Prolog:

- Las listas se especifican entre corchetes y separando los términos con comas:

```
[zeus, afrodita, hera, apolo]
```

- Las listas pueden contener todo tipo de términos: átomos, números, variables y términos complejos:

```
[afrodita, padre(carlos_I, felipe_II), Amigo, 23]
```

- Exise un tipo especial de lista denominado **lista vacía**. Su longitud es cero.

```
[]
```

- Las listas pueden contener listas anidadas.

La lista siguiente tiene tres elementos: el primero es un átomo y los dos últimos son dos litas, cada una con dos elementos.

```
[afrodita, [carlos_I, felipe_II], [perro(bobby), 5.19]]
```

- Podemos encontrar listas anidadas en otras listas anidadas en otras listas... ¿Cuántos elementos tiene esta lista?

Si no te salen seis, vuelve a mirar.

```
[[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]]
```

Estructura de una lista: cabeza y cola

Cualquier lista no vacía puede despomponerse en dos partes: **cabeza** y **cola**. La cabeza es el primer elemento de la lista, y la cola es la todo

lo demás. Dicho de otro modo, la cola es siempre la lista resultante de eliminar la cabeza a la lista original.

En la lista:

```
[zeus, afrodita, hera, apolo]
```

`zeus` es la cabeza y `[afrodita, hera, apolo]` es la cola.

De manera paralela, en la lista:

```
[[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]]
```

`[]` es la cabeza y `[gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]]` es la cola.

Y en la lista:

```
[gato(chaplin)]
```

`gato(chaplin)` es la cabeza y `[]` es la cola.

Fíjate en que la cola siempre es una lista. Aunque sea vacía.

Estructura de las listas vacías `[]`

¿Cómo descomponemos en cabeza y cola una lista vacía? La verdad es que no podemos. Las listas vacías funcionan como un tipo especial de símbolo que tiene unos usos muy importantes, como veremos a continuación, para escribir programas recursivos.

El operador `|`

Prolog tiene un operador predefinido `|` que se utiliza para descomponer una lista en cabeza y cola.

El caso de uso más habitual de `|` consiste en combinarlo con la unificación para extraer elementos de una listas. Fíjate en el siguiente ejemplo:

```
?- [Cabeza|Cola] = [zeus, afrodita, hera, apolo].
Cabeza = zeus,
Cola = [afrodita, hera, apolo].
```

Utilizamos el operador `|` para separar la cabeza de la cola de la lista y mediante el mecanismo de unificación de Prolog asignamos la dos partes de la lista a las variables `Cabeza` y `Cola`. Ten en cuenta que no hay nada especial en las palabras `Cabeza` y `Cola`, basta con que uses variables:

```
?- [X|Y] = [a(b(c)), 84, azul].
X = a(b(c)),
Y = [84, azul].
```

Recuerda que las listas vacías no tienen estructura. Por eso la siguiente unificación falla: `[]` es un tipo muy especial de lista:

```
?- [X|Y] = [].
false.
```

Podemos hacer muchas más cosas con el operador `|`. Es realmente flexible. Podemos lanzar consultas como la siguiente:

```
?- [Primero, Segundo|Cola] = [zeus, afrodita, hera, apolo].
Primero = zeus,
Segundo = afrodita,
Cola = [hera, apolo].

[Primero, Segundo, Tercero|Cola] = [[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]].
Primero = [],
Segundo = gato(chaplin),
Tercero = [2, [b, c]],
Cola = [[], Z, [2, [b, c]]].

?- [Primero, Segundo, Tercero|Cola] = [1, 2, 3].
Primero = 1,
Segundo = 2,
Tercero = 3,
Cola = [].

?- [Primero, Segundo, Tercero, Cuarto|Cola] = [1, 2, 3].
false.
```

La variable anónima _

Ya hemos visto anteriormente que cualquier término cuyo nombre empiece por un guión bajo `_x`, `_cosa`, `_1` se considerará un nombre de variable. Existe además un tipo especial de variable, la variable anónima, representada por el símbolo `_`. Este tipo especial de variable se utiliza en Prolog cuando necesitamos una variable pero no necesitamos instanciarla con ningún valor. Fíjate en los siguientes ejemplos:

```
?- [Primero, Segundo, Tercero, Cuarto|Cola] = [1, 2, 3, 4, 5, 6].
Primero = 1,
Segundo = 2,
Tercero = 3,
Cuarto = 4,
Cola = [5, 6].
```

En este ejemplo capturamos los valores que queremos, los cuatro primeros, y asignamos la lista sobrante a la cola. Imagina que solo necesitamos capturar los elementos segundo y cuarto, además de la cola.

```
?- [_, Segundo, _, Cuarto|Cola] = [1, 2, 3, 4, 5, 6].
Segundo = 2,
Cuarto = 4,
Cola = [5, 6].
```

O solo el tercero.

```
?- [_, _, Tercero|_] = [1, 2, 3, 4, 5, 6].
Tercero = 3.
```

Otro ejemplo más extermo. Tenemos una lista más o menos compleja: `[[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]]` y queeremos extraer la cola del tercer elemento, que es una lista anidada. ¿Qué consulta lanzamos?

```
?- [_, _, [_|Cola]|_] = [[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]].
Cola = [[b, c]].
```

Repitamos el ejemplo usando variables no anónimas, para dejar claro con qué se asigna cada una de ellas:

```
?- [X1, X2, [Cabeza|Cola]|Resto] = [[], gato(chaplin), [2, [b, c]], [], Z, [2, [b, c]]].
X1 = [],
X2 = gato(chaplin),
Cabeza = 2,
Cola = [[b, c]],
Resto = [[], Z, [2, [b, c]]].
```

Miembro de una lista

Un procedimiento habitual en Prolog consiste en definir mecanismos recursivos para inspeccionar y manipular el contenido de listas. Y

lo más básico que vamos a ver es cómo comprobar que determinado elemento *es miembro* o *está incluido* en una lista.

Teniendo en cuenta cómo se se estructuran las listas en Prolog, podríamos definir la relación *miembro de una lista* de la siguiente manera: un elemento es miembro de una lista si:

1. el elemento es la cabeza de la lista, o;
2. el elemento es miembro de la cola de la lista.

El siguiente código de ejemplo define el término complejo `miembro/2` con el que, a partir de cualquier objeto y una lista arbitrarios, podemos comprobar si el objeto en cuestión es miembro de la lista.

```
% miembro de una lista

miembro(X, [X|Cola]).
miembro(X, [Cabeza|Cola]) :- miembro(X, Cola).
```

Fíjate utilizamos el operador `|` para declarar, en el primer hecho, que el objeto en cuestión coindice con la cabeza de la lista, y en la relgla, en una definición recursiva, que el objeto es miembro de la cola de la lista.

Veamos algunos ejemplos de uso:

```
?- miembro(1, [1, 2, 3]).
true.
```

Prolog responde afirmativamente de manera inmediata porque puede unificar el número `1` con las dos ocurrencias de la variable `X` que aparecen en el primero hecho y el objetivo se satisface.

```
?- miembro(2, [1, 2, 3]).
true.
```

En este caso el objetivo se satisface pero no a partir del primer hecho, sino a través de la regla recursiva. En primer lugar, el hecho `miembro(2, [1|[2, 3]])` falla y trata de satisfacer `miembro(2, [2|[3]])`, que sí tiene éxito.

¿Qué ocurre en un ejemplo como el siguiente?

```
?- miembro(4, [1, 2, 3]).
false.
```

Obviamente falla porque el elemento `4` no está incluído en la lista. ¿Pero cómo maneja Prolog este caso? Veamos, la consulta `miembro(4, [1, 2, 3])` no tiene éxito, así que la llamada recursiva trata de satisfacer `miembro(4, [2, 3])` que tambien falla. A continuación, falla

`miembro(4, [3])` y trata de satisfacer `miembro(4, [])`. Aquí el primer hecho no tiene éxito, pero es que la regla recursiva tampoco se puede aplicar, porque las listas vacías no pueden descomponerse en cabeza y cola. Así que Prolog deja de buscar y nos dice que el elemento `4` no está en la lista, tal y como esperábamos que hiciera.

Una vez hemos entendido cómo funciona este predicado `miembro/2`, vamos a ver otros usos más sofisticados. No solo podemos utilizarlo para lanzar preguntas directas y comprobar si un término es miembro de una lista. También podemos utilizar variables para extraer todos y cada uno de los elementos de una lista:

En otros lenguajes de programación, esta consulta equivale a un bucle `for`.

```
?- miembro(Dia, [lunes, martes, miercoles, jueves, viernes, sabado, domingo]).
Dia = lunes ;
Dia = martes ;
Dia = miercoles ;
Dia = jueves ;
Dia = viernes ;
Dia = sabado ;
Dia = domingo ;
false.
```

Por último, una pequeña aclaración sobre la definición que hemos utilizado para `miembro/2`. Si usas un editor con resaltado de sintaxis para Prolog, es probable que te haya marcado de manera especial la variable `Cola` en el primer hecho y la variable `Cabeza` en la regla recursiva. Esto ocurre porque dichas variables aparece una sola vez en cada cláusula y no utilizamos los valores con los que se instancian para nada.

Podemos mejorar el estilo de nuestro programa utilizando variables anónimas en estos casos. La versión definitiva de `miembro/2` quedaría:

```
% miembro de una lista v.2

miembro(X, [X|_]).
miembro(X, [_|Cola]) :- miembro(X, Cola).
```

Recorriendo listas de manera recursiva

Como hemos visto antes, el predicado `miembro/2` funciona recorriendo de manera recursiva una lista: primero comprobamos si determinada condición se daba en la cabeza de la lista y, en caso negativo, repetíamos la operación sobre la lista cola sobrante. Este mecanismo es muy productivo y bastante común en Prolog, así que es preciso coger soltura.

Cuando trabajamos con listas, a menudo necesitamos comparar dos listas diferentes, copiar partes de una en la otra o realizar operaciones similares. Vamos a imaginar un predicado llamado `a2b/2` que tome dos listas como entrada, las compare, y tenga éxito si y solo si el primer argumento sea una secuencia de `a` y el segundo sea una secuencia de `b` de la misma longitud. De modo que la siguiente consulta tendría éxito.

```
?- a2b([a, a, a, a], [b, b, b, b]).
true.
```

Pero las siguientes fallarían:

```
?- a2b([a, a, a, a], [b, b, b]).
false.

?- a2b([1, 2, 3], [b, b, b]).
false.
```

En este caso, como hicimos a la hora de definir `miembro/2`, tenemos que pensar en cómo resolver en primer lugar el ejemplo más sencillo posible. Y cuando trabajamos con listas, el caso más sencillo son las listas vacías. Así que, ¿cómo se debería comportar `a2b/2` cuando se encuentra dos listas vacías? La definición más básica que necesitamos incluir es:

```
a2b([], []).
```

Para listas no vacías, tenemos que pensar de manera recursiva. `a2b/2` tiene que comprobar que la cabeza de la primera lista es una *a* y que la cabeza de la segunda lista es una *b*. Si esto es cierto, debería pasar a analizar de manera recursiva ambas colas.

```
a2b([a|Cola1], [b|Cola2]) :- a2b(Cola1, Cola2).
```

Juega con distintos ejemplos para comprobar que entiendes cómo funciona, tanto cuando tiene éxito como cuando falla: Prolog compara dos listas. Si las cabezas contienen, respectivamente, una *a* y una *b*, se lanza a comparar las colas restantes. A medida que funciona, va comparando listas cada vez más cortas, hasta que termina comparando dos listas vacías.

Por último, fíjate qué otros usos podemos darle a este predicado `a2b/2` cuando especificamos variables como entrada. Podemos usarlo como una especie de traductor entre secuencias de *a* y secuencias de *b*.

Comprueba que funciona. Recuerda el predicado predefinido `trace/0` para visualizar en el intérprete la traza y los caminos de búsqueda de Prolog. Recuerda también que para salir de este modo tienes que ejecutar el predicado `nodebug/0`.

```
?- a2b([a, a, a, a, a], ListaDeBs).
ListaDeBs = [b, b, b, b, b].

?- a2b([a, a], ListaDeBs).
ListaDeBs = [b, b].

?- a2b(ListaDeAs, [b, b, b, b, b, b]).
ListaDeAs = [a, a, a, a, a, a].
```

¿Y qué ocurre si lanzamos la consulta con dos variables?

```
?- a2b(ListaDeAs, ListaDeBs).
ListaDeAs = ListaDeBs, ListaDeBs = [] ;
ListaDeAs = [a],
ListaDeBs = [b] ;
ListaDeAs = [a, a],
ListaDeBs = [b, b] ;
ListaDeAs = [a, a, a],
ListaDeBs = [b, b, b] ;
ListaDeAs = [a, a, a, a],
ListaDeBs = [b, b, b, b] ;
ListaDeAs = [a, a, a, a, a],
ListaDeBs = [b, b, b, b, b] ;
ListaDeAs = [a, a, a, a, a, a],
ListaDeBs = [b, b, b, b, b, b] ;
ListaDeAs = [a, a, a, a, a, a, a],
ListaDeBs = [b, b, b, b, b, b, b] ;
ListaDeAs = [a, a, a, a, a, a, a, a],
ListaDeBs = [b, b, b, b, b, b, b, b] .
```

Ejercicios

1. Otros ejercicios. [5]
2. La sesión práctica. [6]

[5] http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse16

[6] http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse17