Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20

Parte III Programación Lógica

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica 8: Introducción a las listas en Prolog

Índice

1. Objetivo de la Práctica

El objetivo de la práctica es introducir el uso de las listas en Prolog.

2. Listas en Prolog

Intuitivamente, una lista es una secuencia de cero o más elementos. En Prolog, la representación interna de las listas puede verse como un caso especial de la representación de los términos. En este contexto, una lista puede definirse inductivamente como sigue:

1. `[]` es una lista (vacía);
2. `.(E,L)` es una lista (no vacía), si `E` es un elemento y `L` una lista. El elemento `E` se denomina la *cabeza* y `L` es la *cola* de la lista.

En una lista, el símbolo de función "." puede entenderse como un operador binario que *construye* la lista `.(E,L)` *añadiendo* `E` al principio de la lista `L`. Por analogía con otros lenguajes como Haskell, podríamos decir que "`[]`" y "`.(_,_)`" son los constructores del dominio de las listas. Sin embargo, la elección de estos símbolos constructores es completamente arbitraria y bien se podrían haber elegido otros; cualquier símbolo de constante y de función habrían sido válidos, p. ej., "*nil*" o "*cons*". Nuestra preferencia por los dos primeros radica en que algunos sistemas Prolog, en particular SWI-Prolog, emplean esta notación como representación interna para las listas.

La Figura 1 muestra la lista formada por los elementos `1`, `2`, y `3` (es decir, la lista `.(1,.(2,.(3,[]))))` representada como el correspondiente término (un árbol binario).

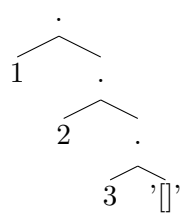

Figura 1: Una lista de enteros.

Construye la representación visual en forma de término binario de las siguientes listas:

```
.(a,.(b,.(c,.(d,[ ]))))
.(1,.(.(juan,.(pedro,[ ])),.(2,[ ])))
.([ ],.(.(a,.(b,[ ])),.(.(1,.(2,.(3,[ ]))),[ ])))
.([ ],[ ])
```

¿Cuál es el unificador más general de `[E|L]` y `[2,2]`?

2.1. Una representación simple para las listas

La representación de las listas mediante los constructores "`[]`" y "`.`" puede llegar a ser verdaderamente confusa. Por este motivo, el lenguaje Prolog proporciona al programador una notación alternativa para representar las listas, que no es otra cosa que azúcar sintáctico para endulzar la representación interna tan poco amigable de las listas. En la notación alternativa, el constructor prefijo binario "`.(_,_)`" se sustituye por el operador infijo "`[_|_]`". Esto permite

representar la lista `.(E,L)` en notación infija, concretamente como una estructura `[E|L]`; es decir, con el primer elemento `E` separado por el símbolo `|` de la lista restante `L`, que contiene todos los elementos de la lista excepto el primero.

Para apreciar el cambio que supone para el programador esta nueva representación de las listas, basta ver que las listas del ejemplo anterior se denotan ahora como:

```
[a|[b|[c|[d|[ ]]]]],
[1|[[juan|[pedro|[ ]]]]|[2|[ ]]]],
[[ ]|[[a|[b|[ ]]]]|[[1|[2|[3|[ ]]]]|[ ]]]],
[[ ]|[ ]]
```

Esta notación todavía no es lo suficientemente legible, por lo que Prolog admite y comprende las siguientes abreviaturas:

1. `[`e_1`|[`e_2`|[`e_3`|...[`e_n`|L]]]]` se abrevia a `[`e_1`, `e_2`, `e_3`, ..., `e_n`|L]`; y
2. `[`e_1`, `e_2`, `e_3`, ..., `e_n`|[ ]]` se abrevia a `[`e_1`, `e_2`, `e_3`, ..., `e_n`]`

Es importante advertir que, con esta nueva simplificación, la lista `[a|[b|[c|[d|[]]]]]` puede representarse de muchas formas equivalentes, siendo la más simple `[a,b,c,d]` aunque serían igualmente equivalentes estas otras representaciones: `[a,b,c,d|[]]`, o `[a,b,c|[d]]`, `[a,b|[c,d]]`, o `[a|[b,c,d]]`.

Observa que una lista no tiene que acabar necesariamente con la lista vacía. De hecho, en muchas ocasiones se utiliza una variable como patrón, que más tarde unificará con el resto de otra lista. Por ejemplo, en la lista `[a, b|L]`, la variable `L` es susceptible de unificar con el resto de cualquier lista cuyos dos primeros elementos sean `a` y `b`. Con las nuevas facilidades, las listas dadas como ejemplo pueden denotarse finalmente como:

```
[a,b,c,d],
[1,[juan,pedro],2],
[[],[a,b],[1,2,3]],
[[],[]]
```

apreciándose una notable mejora en la legibilidad de estas expresiones.

Al igual que Haskell, Prolog permite recuperar información dentro de las listas mediante ajuste de patrones (en este caso, bidireccional), como practicaremos en los siguientes ejercicios.

Ejercicio 1 *Edita un programa Prolog e introduce los siguientes hechos:*

```
countTo(1,[1]).
countTo(2,[1,2]).
countTo(3,[1,2,3]).
countTo(4,[1,2,3,4]).
```

Carga el código y escribe las consultas:

```
?- countTo(X,[_,_,_,Y]).
?- countTo(X,[_,_,_|Y]).
```

¿Es `Y` *un elemento o una lista en cada caso?*

Ejercicio 2 *Prueba las siguientes consultas:*

```
?- countTo(4,[H|T]).
?- countTo(4,[H1,H2|T]).
?- countTo(4,[_,X|_]).
?- countTo(2,[H1,H2|T]).
?- countTo(2,[H1,H2,H3|T]).
```

¿Qué ves? ¿Por qué? ¿Qué efecto tiene usar el símbolo de subrayado ("_")?

Tal y como se ha definido, el constructor de listas "`.(_,_)`" (o el operador "`[_|_]`" si usamos la notación infija) permite añadir un elemento `E` *en cabeza* de una lista `L` para construir una lista más grande. Cualquier otro tipo de operación sobre listas debe de ser definido en Prolog.

Por ejemplo, siguiendo la definición de listas dada al comienzo de este apartado y adoptando la representación infija, es fácil definir inductivamente un predicado `islist` que confirme si un dato es o no una lista, definiendo un caso base para la lista vacía `[]` y un caso recursivo para listas no vacías, formadas usando el constructor `[_|_]`:

```
% islist(L), L es una lista
islist([]).
islist([_|T]) :- islist(T).
```

La mayoría de los sistemas Prolog proporcionan una buena colección de predicados predefinidos para la manipulación de listas. Dos de los más usuales son `member` y `append`:

- El predicado `member(E,L)` permite comprobar si un elemento `E` está contenido en la lista `L`, si bien este predicado es invertible y puede emplearse también para extraer los elementos de una lista.

  Ejercicio 3 *Prueba las siguientes consultas, escribiendo ";" tras cada respuesta del intérprete para explorar todo el espacio de soluciones:*

  ```
  ?- member(2,[5,2,3,2]).
  ?- member(X,[5,2,3]).
  ```

  Ejercicio 4 *Escribe tu propio predicado* `mymember` *a partir del siguiente código (con errores) que debes corregir para obtener la solución.*

  ```
  mymember(E,[E,_]).
  mymember(E,[H|L]) :- mymember(H,L).
  ```

  La definición del predicado `member` en Prolog es:

  ```
  % member(E,L), E pertenece a L
  member(E,[E|_]).
  member(E,[_|L]) :- member(E,L).
  ```

- El predicado `append(L1,L2,L)` permite concatenar las listas `L1` y `L2` para formar la lista `L`. Este predicado es predefinido en Prolog pero puedes definirlo tú mismo. En el siguiente ejercicio estudiamos cómo hacerlo.

Ejercicio 5 *Define un predicado llamado* `myappend` *que haga exactamente lo mismo que* `append` *pero estando éste definido por ti mismo. El código siguiente es un punto de comienzo pero hay un error en él que debes encontrar y corregir.*

```
myappend([],L,L).
myappend([E|L1],L2,[X|L3]) :- X = E, myappend(L1,L2,L1).
```

(Ten en cuenta que a veces Prolog da advertencias sobre "singleton variables"*. Este aviso significa que hay una variable que aparece en un solo lugar y que no se usa en ninguna otra parte de la regla, por lo que usar una variable anónima, representada con un subrayado, sería probablemente más apropiado).*

La definición en Prolog del predicado[1] `append` es como sigue. ¿Por qué funciona esto?

```
% append(L1,L2,L), la concatenación de L1 y L2 es L
append([],L,L).
append([E|L1],L2,[E|L3]) :- append(L1,L2,L3).
```

Al igual que el predicado `member`, este predicado es invertible y admite múltiples usos.

Ejercicio 6 *Prueba las siguientes consultas, escribiendo ";" tras cada respuesta del intérprete para explorar todo el espacio de soluciones:*

```
?- append([a,Y],[Z,d],[X,b,c,W]).
?- append(L1,L2,[a,b,c,d]).
```

Con la ayuda de los predicados `member` y `append` se puede definir una gran variedad de predicados, como veremos en los siguientes subapartados.

2.2. Aplanar una lista.

Vamos a definir la relación `flatten(L,A)`, donde `L` es una *lista de listas*, tan compleja en su anidamiento como queramos imaginar, y `A` es la lista que resulta de reorganizar los elementos contenidos en las listas anidadas en un único nivel. Por ejemplo:

```
?- flatten([[a,b],[c,[d,e]],f],L).
L = [a,b,c,d,e,f]
```

[1] Sin embargo, esta definición no se comporta bien en todos los casos, ya que para un objetivo "`?- append([a], a, L).`" se obtiene como respuesta "`L = [a|a]`", que no es una lista, dado que "a" no lo es. Una forma de solucionar este problema es comprobar que, en efecto, en el segundo argumento del predicado `append` se introduce siempre una lista. Para ello basta con modificar las reglas que definen el predicado `append`:

```
append([],L,L) :- islist(L).
append([E|L1],L2,[E|L3]) :- islist(L2), append(L1,L2,L3).
```

Este hecho vuelve a poner de manifiesto que en un lenguaje sin tipos es el programador el encargado de comprobar que cada objeto que se utiliza es del tipo (dominio) adecuado.

La versión que damos hace uso del predicado predefinido `atomic(X)`, que comprueba si el objeto que se enlaza o vincula a la variable `X` es o no un objeto simple "atómico" (i.e., un símbolo constante —un átomo en la jerga de Prolog—; es decir, un entero o una cadena de caracteres). También se requiere utilizar el predicado predefinido `not`. Aquí y en los próximos apartados, emplearemos `not` con un sentido puramente declarativo, interpretándolo como la conectiva "¬" y sin entrar en el tratamiento especial que el lenguaje Prolog realiza de la negación.

```
% flatten(L, A), A es el resultado de aplanar L
flatten([],[]).
flatten([X|L],[X|P]) :- atomic(X), flatten(L,P).
flatten([X|L],P) :- not(atomic(X)), flatten(X,P_X),
                    flatten(L,P_L), append(P_X,P_L,P).
```

La solución anterior indica que la lista vacía ya está aplanada (primera cláusula). Por otra parte, para aplanar una lista genérica `[X|L]`, se distinguen dos casos (segunda y tercera cláusula, respectivamente):

1. Si el elemento en cabeza `X` es atómico, ya está aplanado; por lo que basta con ponerlo en cabeza de la lista plana y seguir aplanando el resto `L` de la lista que se está aplanando.

2. Si el elemento en cabeza `X` no es atómico, habrá que aplanarlo; después se aplana el resto `L` de la lista original y finalmente se concatenan las listas resultantes del proceso anterior.

2.3. Último elemento de una lista.

El elemento `E` en cabeza de una lista `[E|L]` es directamente accesible por unificación. Sin embargo, para acceder al último de una lista, es preciso definir un predicado específico `last(L,U)`, donde `U` es el último elemento de la lista `L`. Este predicado puede definirse como sigue:

```
% last(L,U), U es el último elemento de la lista L
last([U],U).
last([_|L],U) :- last(L,U).
```

La lectura declarativa de este predicado es clara: `U` es el último elemento de la lista `[_|L]`, si `U` es el último elemento de la lista remanente `L`; en caso de que la lista contenga un solo elemento, es decir, la lista sea de la forma `[U]`, es un hecho que `U` es el último elemento. Por otra parte, el efecto procedural de este predicado consiste en recorrer toda la lista, desechando cada uno de los elementos, hasta alcanzar el último.

El predicado `last(L,U)`, también puede definirse de manera mucho más concisa usando la relación `append`:

```
% last(L,U), U es el último elemento de la lista L
last(L,U) :- append(_,[U],L).
```

La idea que subyace en esta definición es que la lista `L` es la concatenación, a una lista que contiene todos los elementos salvo el último, de la lista `[U]` formada por el último elemento. La definición del predicado `append` y el algoritmo de unificación hacen el resto.

2.4. Operaciones con sublistas.

El predicado `append` también puede utilizarse para definir operaciones con sublistas. Si consideramos que un prefijo de una lista es un segmento inicial de la misma y que un sufijo es un segmento final, podemos definir estas relaciones con gran facilidad haciendo uso del predicado `append`.

```
% prefix(P,L), P es un prefijo de la lista L
prefix(P,L) :- append(P,_,L).

% suffix(P,L), P es un sufijo de la lista L
suffix(P,L) :- append(_,P,L).

% sublist(S,L), S es una sublista de la lista L
sublist(S,L) :- suffix(L1,L),prefix(S,L1).
```

La última definición afirma que `S` es una sublista de `L` si `S` está contenida como prefijo de un sufijo de `L`.

Algunas observaciones:

1. En muchas ocasiones, comprobar los dominios de los argumentos de los predicados definidos puede afectar el rendimiento de un programa por lo que a menudo se omiten dichas comprobaciones.

2. Por otro lado, la llamada indirecta a otros predicados en el cuerpo de una regla también puede afectar al rendimiento. Por este motivo, puede ser aconsejable sustituir la llamada por el cuerpo del predicado al que se llama (renombrando adecuadamente las variables de la cláusula para que su cabeza coincida con la llamada). Por ejemplo, en la definición del predicado `sublist`, podemos reemplazar la llamada `suffix(L1,L)` por `append(_,L1,L)`. De igual forma, se sustituiría `prefix(S,L1)` por `append(S,_,L1)`, lo que conduce a la siguiente definición más eficiente para el predicado `sublist`:

   ```
   % sublist(S,L), S es una sublista de la lista L
   sublist(S,L) :- append(_,L1,L),append(S,_,L1).
   ```

   El proceso de transformación que acabamos de describir es un caso particular de una transformación que se conoce como desplegado *(unfolding)* y es una de las técnicas básicas empleadas en el campo de la transformación automática de los programas lógicos.

2.5. Invertir una lista: parámetros de acumulación y recursión de cola.

La inversión de listas puede definirse de forma directa en términos del predicado `append`:

```
% inverse(L,I), I es la lista que resulta de invertir L
inverse([],[]).
inverse([H|T],L) :- inverse(T,Z), append(Z,[H],L).
```

Esta versión es muy ineficiente debido a que consume y reconstruye la lista original usando el predicado `append`, siendo su coste cuadrático con respecto al número de elementos de la lista que se está invirtiendo. Para eliminar las llamadas a `append` y lograr una mayor eficiencia se hace necesario el uso de un *parámetro de acumulación* que, como indica su nombre, se utiliza para almacenar resultados intermedios. En el caso que nos ocupa, se almacena la lista que acabará por ser la lista invertida, en sus diferentes fases de construcción.

```
% inverse(L,I), I es la lista que resulta de invertir L
% Usando un parámetro de acumulación.
inverse(L,I) :- inv(L,[],I).

% inv(Lista,Acumulador,Invertida)
inv([],I,I).
inv([X|L],A,I) :- inv(L,[X|A],I).
```

Observa que la lista invertida se va construyendo en cada llamada al predicado `inv`, adicionando un elemento `X` a la lista acumulada en el paso anterior mediante el empleo del operador "`[_|_]`", en lugar de emplear el predicado `append`. La Figura 2 ilustra este proceso.

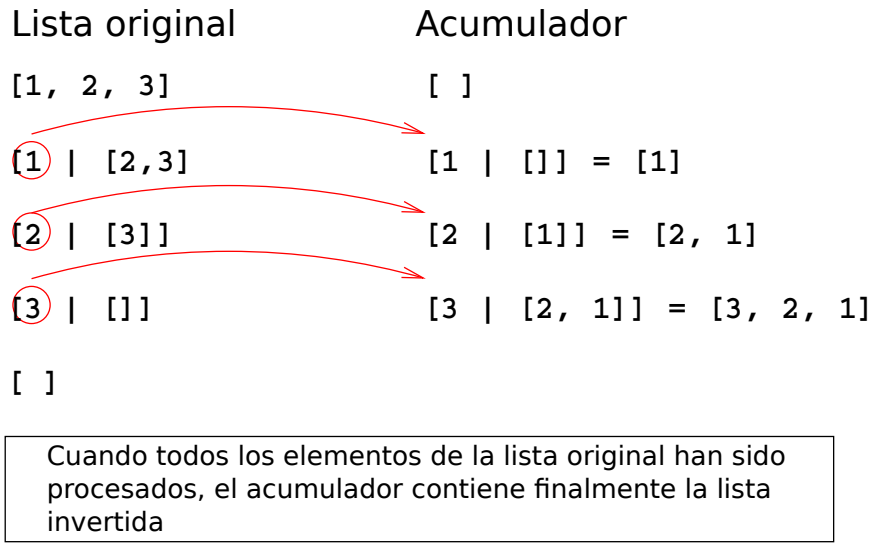

Figura 2: Fases en la inversión de una lista usando un parámetro de acumulación.

La técnica de los parámetros de acumulación es una técnica general que puede aplicarse a numerosos problemas (p. ej., el cómputo eficiente de los números de Fibonacci).

2.6. Ejercicios finales con listas

Ejercicio 7 *Escribe un predicado binario* `swap` *que acepta una lista y genera una lista similar con los dos primeros elementos intercambiados.*

Ejercicio 8 *¿Qué hace el siguiente predicado misterioso y por qué?*

```
mistery([],0).
mistery([_|T],N) :- mistery(T,M), N is M+1.
```

Devuelve True si N es igual al tamaño de la lista.

Ejercicio 9 *Completa el siguiente programa Prolog para que implemente la operación de comprobar si una colección de elementos es subconjunto de otra:*

```
subset([],_).
subset([A|X],Y) :- member(A,Y), [          ].
```

A `subset(X,Y)`

B `append(X,[A],Y)`

C `subset(Y,X)`

D `member(X,Y)`

Ejercicio 10 *Completa el siguiente programa Prolog que comprueba si una lista está ordenada:*

```
sorted([X]).
[                ] :- X =< Y, sorted([Y|Ys]).
```

A `sorted([X|Y,Ys])`

B `sorted([X,[Y|Ys]])`

C `sorted([X,Y|Ys])`

D `sorted(X,Y,Ys)`

Ejercicio 11 *Completa el siguiente programa lógico para que, dado un entero y una lista de enteros, elimine las ocurrencias de dicho entero de la lista. Los predicados predefinidos "==" y "\==" representan la igualdad y la desigualdad, respectivamente. Por ejemplo, la llamada* `remove(3,[1,2,3,1,2,3],L)` *computa la respuesta* `L = [1,2,1,2]`.

```
remove(_,[],[]).
remove(C,[X|R],L) :- X == C, remove(C,R,L).
remove(C,[X|R],W) :- X \== C, [                ], [                ].
```

A `remove(C,R,L), W = [X|L]`

B `remove(C,R,W), L = [X|W]`

C `remove(C,R,L), W = L`

D `remove(C,[X|R],L), W = L`