Capítulo 2

Programación con Listas

En Prolog la estructura de lista está predefinida como una estructura recursiva lineal cuyas componentes pueden ser heterogéneas porque en Prolog no existe una comprobación tipos. Básicamente una lista es una estructura con dos argumentos: la cabeza de la lista, que es un componente de la lista, y la cola que es otra lista con el resto de los componentes. Normalmente existen dos notaciones para listas, una notación prefija: `.(Cabeza,Cola)` y una notación infija `[Cabeza|Cola]` más cómoda de utilizar, y para denotar la lista vacía se utiliza siempre `[]`. Así, la lista `(a,b,c,d)` se puede escribir en Prolog de cualquiera de las dos formas siguientes:

```
.(a,.(b,.(c,.(d.[]))))        [a|[b|[c|[d|[]]]]],
```

sin embargo, la segunda forma admite simplificaciones que permiten escribir todos los elementos de la lista entre corchetes y separados por comas:

```
[a,b,c,d],
```

o escribir varios elementos a la izquierda del símbolo `|`, también separados por comas, así como cualquier combinación de estas dos formas; con lo que nuestra lista se puede escribir de cualquiera de las formas siguientes:

```
[a|[b,c,d]]    [a,b|[c,d]]    [a,b,c|[d]]    [a,b,c,d|[]] .
```

Es importante observar que en las dos notaciones para listas, el segundo argumento debe ser siempre una lista, pues aunque como hemos dicho Prolog no comprueba tipos, sí es capaz de distinguir las listas de otra clase de términos. Así, notaciones como las siguientes

```
.(a,b)        [a|b]
```

se considerarían erróneas porque `b` es una constante (átomo) y no es una lista. También hay que tener en cuenta que entre los componentes de una

lista pueden aparecer otras listas, así, por ejemplo, `[a,b,[c,d,e]]` sería una lista con tres componentes, el último de los cuales es a su vez una lista, mientras que `[a,b|[c,d,e]]` sería una lista con cinco componentes todos constantes. Para consolidar el uso de las notaciones de lista proponemos el siguiente ejercicio.

Ejercicio 2.0.5 *Aplíquese el algoritmo de unificación a los pares de términos que aparecen en las filas de la siguiente tabla anotando los resultados de cada aplicación (instanciaciones o fallo) en la columna* `resultado`

		`resultado`
`[X|Y]`	`[a,b,c]`	
`[a,[b,c]]`	`[X|Y]`	
`[[a,b],c]`	`[X|Y]`	
`[a|X]`	`[Y|a]`	
`[a|X]`	`[Y|[c]]`	
`[a|X]`	`[Y]`	
`[a,b,Y]`	`[X,T]`	
`[a,b,X]`	`[X|T]`	
`[a,b|Y]`	`[X,T]`	
`[a,b|[Y]]`	`[a,b,c]`	
`[a,b|[Y]]`	`[a,b,c,d]`	

2.1. Dominios para listas

Como hemos dicho en el apartado anterior, Prolog es capaz de distinguir las listas de cualquier otra clase de términos, pero esta distinción la hace a nivel interno para detectar errores de notación durante el proceso de unificación. Si en algún programa queremos saber si un término `T` es una lista (no vacía) podemos intentar unificarlo con una expresión de la forma `[X|Xs]` usando el predicado de unificación explícita '.=.', o podemos definir un predicado recursivo `lista/1` en la forma

```
lista([]).
lista([X|Xs]):-lista(Xs).
```

Este predicado es lo más parecido a una definición del tipo lista que se puede hacer en Prolog y sirve para comprobar si un término es una lista cuando se lo proporciona un argumento conocido, pero también sirve para generar, por reevaluación, listas cuyas componentes son variables libres.

Ejercicio 2.1.1 *Pruébese el predicado anterior con los argumentos siguientes:* `a`, `arbol(r,Iz,De)`, `[a,b]` *y* `[a,b|arbol(r,Iz,De)]`. *Pruébese nuevamente con una variable y pídanse varias reevaluaciones para ver la secuencia de listas que se va generando.*

De forma parecida se pueden construir predicados para la definición de dominios más concretos correspondientes a listas cuyos componentes sean de una determinada clase, siempre que se pueda usar la unificación para distinguir los términos de dicha clase o disponga de un predicado de definición del correspondiente dominio.

Ejercicio 2.1.2 *Construid los predicados* `listanat/1` *y* `listaficha/1` *para la definición de listas de números naturales y de listas de fichas, respectivamente, suponiendo que las fichas son estructuras dadas por una constructora (funtor)* `ficha/3`.

2.2. Elementos de una listas

En este apartado nos ocuparemos de la construcción de predicados que sirven para comprobar si un elemento pertenece o no a una lista o si se encuentra en determinada posición dentro de una lista y veremos que algunos presentan comportamientos adicionales a los inicialmente esperados. Para expresar estos comportamientos utilizaremos los términos siguientes: *test* cuando el comportamiento sea comprobar si se cumple una determinada relación, *generador único* cuando genere un valor único que verifique la relación, *generador acotado* cuando genere un número finito de valores, *generador infinito* cuando genere valores de manera continuada y *anómalo* cuando no produzca resultado.

Ejercicio 2.2.1 *Definid un predicado* `miembro/2` *entre un elemento y una lista de manera que* `miembro(X,Xs)` *sirva para expresar la relación "el elemento X es miembro de la lista Xs". Construid la definición basándola en la unificación con las cabezas de las cláusulas. Probad el comportamiento de este predicado instanciando los dos argumentos, instanciando la lista y dejando el elemento libre, instanciando el elemento y dejando libre la lista y, por último, dejando los dos argumentos libres. Completad la siguiente tabla de comportamiento.*

X	*Xs*	`comportamiento`
+	+	
-	+	
+	-	
-	-	

Ejercicio 2.2.2 *Definid un predicado* `seleccion/3` *tal que la fórmula atómica* `seleccion(X,Xs,Ys)` *exprese la relación "X es un elemento de Xs e Ys es la lista Xs sin X". Definid también un predicado* `eliminacion/3` *tal que* `eliminacion(X,Xs,Ys)` *sirva para expresar la relación "Ys es el resultado de suprimir en la lista Xs todas las apariciones de X". Determinad los usos que se les puede dar a cada uno de los predicados anteriores.*

Ejercicio 2.2.3 *Definid los predicados* `primero/2` ,`ultimo/2` *y* `elemento/3` *tales que* `primero(X,Xs)` *exprese la relación "X es el primer elemento de la lista Xs",* `ultimo(X,Xs)` *exprese la relación "X es el último elemento de la lista Xs" y* `elemento(X,N,Xs)` *corresponda a la relación "X es el elemento situado en la posición N de la lista Xs". Estudiad los usos que se les puede dar a cada uno de estos predicados.*

2.3. Reconocimiento de patrones con listas

En este apartado vamos a ver predicados definidos sobre pares de listas que sirven para establecer relaciones como ser prefijo, sufijo o sublista de una lista. En principio, aprovechando la facilidad de acceso a las cabezas y a las colas de las lista, podemos definir un predicado para la relación prefijo basado en la coincidencia de los elementos de cabeza.

Ejercicio 2.3.1 *Definid un predicado* `prefijo/2` *tal que* `prefijo(Xs,XsYs)` *exprese la relación "Xs es prefijo de la lista XsYs" teniendo en cuenta que la lista vacía es el prefijo más elemental de cualquier lista y que una lista Xs será prefijo de otra XsYs si las cabezas de ambas coinciden y la cola de la primera lista es un prefijo de la cola de la segunda. Completad la siguiente tabla de comportamiento para este predicado*

Xs	*XsYs*	`comportamiento`
+	+	
-	+	
+	-	
-	-	

y explicadla analizando el flujo de datos que se produce en cada cláusula de la definición dada de `prefijo/2`

Aprovechando la misma facilidad debida a la representación de listas, podemos definir un predicado para la relación sufijo basado en la coincidencia de las colas.

Ejercicio 2.3.2 *Definid un predicado* `sufijo/2` *tal que* `sufijo(Ys,XsYs)` *exprese la relación "Ys es sufijo de la lista XsYs" teniendo en cuenta que toda lista es sufijo de sí misma y que una lista es sufijo de otra si es un sufijo de su cola. Completad la siguiente tabla de comportamiento para este predicado*

Ys	*XsYs*	`comportamiento`
+	+	
-	+	
+	-	
-	-	

y explicadla analizando el flujo de datos que se produce en cada cláusula de la definición dada de `sufijo/2`

Sobre la base de los dos predicados anteriores se puede definir un nuevo predicado para la relación sublista.

Ejercicio 2.3.3 *Definid un predicado* `sublista/2`*, de manera que el átomo* `sublista(Ys,XsYsZs)` *exprese la relación "Ys es una sublista de la lista XsYsZs". Completad también la siguiente tabla de comportamiento para este predicado*

Ys	*XsYsZs*	`comportamiento`
+	+	
-	+	
+	-	
-	-	

y explicadla analizando el flujo de datos que se produce en cada cláusula de la definición dada de `sublista/2`

2.4. Encadenamiento y partición de listas

El encadenamiento de dos listas se puede definir en Prolog de una manera bastante simple obteniéndose un predicado `encadena/3` que resulta de gran utilidad en el manejo de listas. Para la definición de este predicado basta tener en cuenta que la cabeza de la lista resultante del encadenamiento será la cabeza de la primera de las dos listas que se encadenan y que la cola resultará de encadenar la cola de la primera lista con la segunda lista completa.

Ejercicio 2.4.1 *Definid un predicado* `encadena/3` *de manera que el átomo* `encadena(Xs,Ys,XsYs)` *exprese la relación "XsYs es es el encadenamiento de la lista Xs seguida de Ys". Completad la tabla de comportamiento para este predicado*

Xs	*Ys*	*XsYs*	`comportamiento`
+	+	+	
+	+	-	
+	-	+	
-	+	-	
-	-	+	
-	+	-	
+	-	-	

y explicadla analizando el flujo de datos que se produce en cada cláusula de la definición dada de `encadena/3`

A partir de los comportamientos observados para el predicado `encadena/3` podemos redefinir algunos predicados vistos en las secciones anteriores; por ejemplo, una lista es prefijo de otra si existe una tercera lista tal que encadenada detrás de la primera produce la segunda. Razonando de forma parecida podemos llegar también a las definiciones de sufijo y sublista.

Ejercicio 2.4.2 *Utilizando el predicado* `encadena/3` *como predicado auxiliar redefinid los predicados* `prefijo/2`, `sufijo/2` *y* `sublista/2` *como* `prefijo'/2`, `sufijo'/2` *y* `sublista'/2`.

Un razonamiento un poco más sutil nos puede llevar a partir del encadenamiento de listas a la redefinición de `miembro/2` y `seleccion/3`.

Ejercicio 2.4.3 *Redefinid los predicados* `miembro/2` *y* `seleccion/3` *como* `miembro'/2` *y* `seleccion'/3` *utilizando el predicado* `encadena/3` *como predicado auxiliar.*

2.5. Predicados aritméticos con listas

En este apartado nos dedicaremos a estudiar predicados que realizan cálculos numéricos sobre listas. El más básico es el que calcula la longitud de una lista, la forma más simple de definir un predicado `longitud/2` tal que `longitud(L,N)` calcule la longitud `N` de una lista `L` es contando los elementos de forma recursiva: primero los de la cola y después los de la lista completa. Aquí tenemos dos definiciones distintas,

```
longitud1([],0).
longitud1([_|Xs],N):-longitud(Xs,M),N is M+1.
```

es una definición basada en la aritmética predefinida de Prolog, y

```
longitud2([],0).
longitud2([_|Xs],s(M)):-longitud(Xs,M).
```

es una definición basada en la aritmética del sucesor definida por el usuario.

Ejercicio 2.5.1 *Compárense las dos definiciones analizando sus comportamientos en los distintos modos de uso y explíquese por qué en el uso (-,+) la primera definición diverge cuando se pide reevaluación y la segunda no.*

Otra forma de calcular la longitud de una lista es utilizar un predicado inmersor con un parámetro acumulador que vaya contando los elementos de la lista a medida que los vamos quitando,

```
longitud3(L,N):-long(L,0,N).
long([],N,N).
longitud([_|Xs],A,N):-NA is 1+A,longitud(Xs,NA,N).
```

en este caso los cálculos se van haciendo antes de la llamada recursiva con lo que tenemos un definición con recursión de cola más eficiente que la primera que vimos.

Además de este predicado para calcular la longitud de una lista, para listas de números se pueden definir otros predicados que realicen cálculos con los elementos que aparecen en la lista.

Ejercicio 2.5.2 *Teniendo en cuenta las definiciones dadas para* `longitud1/2` *y* `longitud3/2` *enunciad definiciones recursivas parecidas para predicados que sirvan para sumar, para multiplicar y para calcular el máximo valor de una lista de números.*

2.6. Ordenación de listas

Teniendo en cuenta la posibilidad que ofrece la representación de listas en Prolog de acceder directamente a varios elementos del comienzo de una lista y compararlos, podemos definir un predicado `ordenada/1` que determine si una lista (de números o de términos) está ordenada o no.

Ejercicio 2.6.1 *Definid un predicado* `ordenada/1` *que sirva para determinar si una lista (de números o de términos) está ordenada o no, teniendo en cuenta que la lista vacía y las listas de un elemento se pueden considerar ordenadas. Utilícese el orden de términos @=<.*

Respecto a la ordenación de listas, la mayoría de los algoritmos se basan en la combinación de una partición de la listas en dos trozos, la ordenación de los trozos y la combinación de los trozos ordenados. Este esquema lo siguen los algoritmos de ordenación por inserción, ordenación por mezcla y ordenación con pivote (quicksort) que lo aplican del siguiente modo:

- El algoritmo de ordenación por inserción parte la lista en cabeza y cola, ordena la cola e inserta la cabeza en el lugar adecuado de la cola ordenada, por lo que únicamente necesita un procedimiento auxiliar para insertar elementos en listas ordenadas.

- El algoritmo de ordenación por mezcla parte la lista en dos trozos aproximadamente de igual longitud, ordena cada trozo y después los mezcla. Este algoritmo necesita dos procedimientos auxiliares: un procedimiento para separar una lista en dos aproximadamente de igual longitud y otro para mezclar listas ordenadas de forma que se obtenga una lista igualmente ordenada.

- El algoritmo de ordenación por pivote usa un elemento como pivote para partir la lista en una lista con los elementos menores o iguales que el pivote y otra con los elementos estrictamente mayores, ordena ambas listas y después las encadena los menores delante de los mayores. Este algoritmo necesita dos procedimientos: uno para separar una lista con ayuda de un pivote en dos listas una con los elementos

menores o iguales que el pivote y otra con los elementos mayores, y otro procedimiento para encadenar dos listas que ya conocemos.

Ejercicio 2.6.2 *Siguiendo las ideas anteriores, definid tres predicados para la ordenación de listas,* `ord_ins/2`, `ord_mez/2` *y* `ord_piv/2` *que apliquen los correspondientes algoritmos. Definid convenientemente los predicados auxiliares que necesita cada uno de ellos.*

Una lista también se puede ordenar corrigiendo todas las inversiones de orden que se presenten entre cada dos elementos consecutivos de la lista.

Ejercicio 2.6.3 *Definid un predicado* `consecutivos/3`*, de manera que la fórmula atómica* `consecutivos(X,Y,Ls)` *exprese la relación "Y es el sucesor inmediato de X en la lista Ls". Utilizando el predicado* `consecutivos/3`*, definid un nuevo predicado* `ord_inv/2` *para la ordenación de listas basado en la corrección de inversiones de elementos consecutivos.*

2.7. Combinatoria con listas

Teniendo en cuenta que una lista es un conjunto de elementos ordenados según el orden de aparición de izquierda a derecha, dentro de los ejercicios con listas podemos considerar la generación de variaciones y permutaciones a partir de los elementos de una lista; pero también podemos considerar la generación de combinaciones y un tratamiento de conjuntos donde ya la ordenación no debe tenerse en cuenta.

A partir del predicado `seleccion/3` que permite seleccionar de forma aleatoria un elemento de una lista devolviendo además una lista con los rstantes elementos, podemos definir predicados para generar variaciones de un determinado tamaño, y en el caso extremo de que el tamaño coincida con la longitud de la lista podemos mejorar la definición para producir permutaciones.

Ejercicio 2.7.1 *Defínase un predicado* `variacion/3`*, tal que la invocación* `variacion(Ls,N,Vs)`*, con* `Ls` *y* `N` *instanciados, sirva para generar una variación* `Vs` *de tamaño* `N` *de los elementos de la lista* `Ls` *y que, por reevaluación, genere todas las variaciones posibles de tamaño* `N`*. Para esto búsquese una definición recursiva teniendo en cuenta que la única variación de tamaño 0 debe ser la lista vacía y que una variación de tamaño* `N` *se puede generar a partir de otra de tamaño* `N-1` *añadiéndole un elemento que no esté contenido.*

Nota: Debe tenerse mucho cuidado con el orden en el que se aparecen la llamada recursiva y la generación del elemento en la cláusula del caso general.

Ejercicio 2.7.2 *Defínase un predicado* `permutacion/2`*, sobre la base de la definición del ejercicio anterior, que sirva para producir cada una de las permutaciones posibles de los elementos de una lista.*

Para producir las combinaciones de un cierto tamaño a partir de los elementos de una lista también podemos recurrir a la definición recursiva añadiendo un elemento a una combinación de tamaño menor, pero en este caso hay que tener en cuenta que el orden en el que aparezcan los elementos no determinará combinaciones distintas como ocurre con la variaciones. En este caso será mejor buscar una forma canónica para representar cada combinación, esta forma puede ser la lista correspondiente respetando el orden de aparición de los elementos en la lista original, para ello la selección del elemento no se debe dejar al predicado `seleccion/3` sino que deberá hacerse en la cabeza de las cláusulas tomando el primer elemento de la lista o no y completando la combinación con los elementos siguientes (y no con los anteriores).

Ejercicio 2.7.3 *Defínase un predicado* `combinacion/3`*, tal que la invocación a* `combinacion(Ls,N,Cs)`*, con* `Ls` *y* `N` *instanciados, sirva para generar una combinación* `Cs` *de tamaño* `N` *de los elementos de la lista* `Ls` *y que, por reevaluación, genere todas las combinaciones posibles de tamaño* `N`*.*

La estructura de lista puede utilizarse como una representación para conjuntos utilizando listas sin elementos repetidos, pero en tal caso debe tenerse en cuenta que un mismo conjunto puede venir representado por varias lista que se diferencien simplemente en un cambio en el orden de aparición de sus elementos.

Ejercicio 2.7.4 *Definid un predicado* `conjunto/1` *que sirva para determinar si una lista puede servir o no para representar un conjunto. Con ayuda del predicado* `encadena/3` *(o del predicado* `seleccion/3`*) utilizado para localizar elementos dentro de una lista defínase un predicado* `cj_iguales/2` *que determine si dos listas, consideradas como representaciones de conjuntos, representan el mismo conjunto.*

Ejercicio 2.7.5 *Definid predicados* `contenido/2` *para expresar la relación de contenido entre conjuntos,* `union/3`*,* `interseccion/3` *y* `diferencia/3` *que sirvan para calcular los resultados de estas operaciones sobre listas que representan conjuntos controlando que no aparezcan elementos repetidos en los resultados.*

Una forma de producir los subconjuntos o partes de un cierto conjunto, de forma recursiva, consiste en producir primero los subconjuntos de un conjunto con un elemento menos, añadirle el elemento que falta a cada uno de estos subconjuntos para producir otra familia de subsonjuntos diferentes de los primeros y agrupar ambas familias.

Ejercicio 2.7.6 *Defínase un predicado* `agregacion/3` *tal que la invocación a* `agregacion(X,LXs,LXXs)` *con* `X` *instanciado y* `LXs` *instanciado a una lista de listas sirva para agregar el elemento* `X` *delante de cada una de las listas que componen* `LXs` *produciendo* `LXXs`*.*

Ejercicio 2.7.7 *Defínase un predicado* `partes/2` *que sirva para producir de forma recursiva una lista con todos los subconjuntos de un conjunto dado, teniendo en cuenta que el único subconjunto del conjunto vacío es él mismo.*

Parte II

Programación Funcional