

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

7. Manejo de listas

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1 LAS LISTAS EN PROLOG

- Representación
- Patrones

2 ALGUNOS PREDICADOS BÁSICOS SOBRE LISTAS

- Pertenece (member en SWI-Prolog)
- Longitud (length en SWI-Prolog)
- Concatena (append en SWI-Prolog)
- Último (last en SWI-Prolog)
- Inversa (reverse en SWI-Prolog)

3 EJEMPLOS ADICIONALES

- Escritura de una lista
- Suma de los elementos de una lista
- Implementación de autómatas finitos

LAS LISTAS EN PROLOG

Representación de listas en PROLOG

- Las listas de PROLOG son **términos compuestos (estructuras)** y constituyen la estructura de datos más utilizada.
- Sus elementos son **términos** (constantes, variables o términos compuestos), por lo que pueden ser, en particular, listas.
- Las listas no tienen por qué ser homogéneas: una misma lista puede contener términos de clases distintas.
- Toda lista es:
 - O bien **vacía**, y se representa por `[]`.
 - O bien **no vacía**, se representa como `[a1, a2, ..., an]` y tiene:
 - una **cabeza**, que se corresponde con el primer *término* de la lista: `a1`.
 - un **resto**, que es la *lista* formada por todos sus términos salvo el primero: `[a2, ..., an]` **si** `n>1` **o** `[]` **en el caso** `n=1`.

Patrones para listas

- El patrón $[C|R]$ unifica con cualquier lista *no vacía* siempre que C unifique con la cabeza y R unifique con el resto de la lista.
- El patrón $[C1,C2|R]$ unifica con cualquier lista que tenga *al menos dos elementos* siempre que $C1$ unifique con el primer elemento de la lista, $C2$ con el segundo, y R unifique con el resto de la lista.
- El patrón $[C1,C2,C3|R]$ unifica con cualquier lista que tenga *al menos tres elementos* y tal que $C1, C2, C3$ unifiquen con los tres primeros elementos de la lista y R unifique con el resto de la lista.
- El patrón $[C1,C2,C3,C4|R]$...

Observe que lo que está a la derecha de la barra vertical $|$ tiene que ser en todos los casos una *lista*.

Ejemplos (Listas y patrones, 1/2)

- Los términos a , X , $\text{progenitor}(a, X)$, $[a|b]$ o $[a|X]$ *no* son listas (los dos últimos debido a que lo que tienen a la derecha de la barra $|$ no son listas).
- Los términos $[a, X]$, $[a, X|[]]$, $[a|[X]]$ y $[a|[X|[]]]$ son todos ellos listas representando una misma lista: la lista compuesta por los elementos a y X .
- La lista $[X1]$ ($[X1, X2], \dots$) solo unificará con listas que contengan exactamente un (dos, ...) elemento.
- El patrón $[a|X]$ solo unificará con listas con cabeza a , por lo que *no* es unificable por ejemplo con a , $[]$, $[b]$ o $[b, a]$.
- El patrón $[a|X]$ es unificable con cualquiera de las siguientes listas: $[a]$ (mediante la sustitución $X/[]$), $[a, b, c]$ (mediante la sustitución $X/[b, c]$), $[a, [b]]$ (mediante la sustitución $X/[[b]]$), $[a|[b]]$ (mediante la sustitución $X/[b]$), etc.

Ejemplos (Listas y patrones, 2/2)

?- $[X, b] = [a | [B]]$.

$X=a, B=b$.

%[B] tiene que unificar con [b], posible con $B=b$.

?- $[[a, b], c] = [X | Y]$.

$X=[a, b], Y=[c]$.

?- $[[X, f(b)], [c]] = [C | [c]]$.

false.

% la derecha de "|", [c], no unifica con el resto
% de la lista de la izquierda, [[c]].

?- $[[X, a], [c]] = [C | [[D]]]$.

$C=[X, a], D=c$.

%[[D]] tiene que unificar con [[c]], posible con $D=c$.

Ejercicios (Uso básico de listas)

- 1 *Ejercicio nº 1 de la **Práctica de PROLOG nº 3**.*
- 2 *Describa en lenguaje natural (con sus propias palabras) el resultado del siguiente predicado (cierto si ...):*

```
misterio1(E1, E2, [E1,E2|_]).
```

```
misterio1(E1, E2, [_|R]) :-  
    misterio1(E1, E2, R).
```

- 3 *Describa en lenguaje natural (con sus propias palabras) el resultado del siguiente predicado (cierto si ...):*

```
misterio2(L1, L2) :-  
    L1 = [C | _],  
    L2 = [C, C | _].
```

ALGUNOS PREDICADOS BÁSICOS SOBRE LISTAS

- La mayoría de los entornos de programación en PROLOG, entre ellos SWI-Prolog, ofrecen predicados predefinidos implementando las operaciones básicas para el manejo de listas.
- A pesar de ello, en lo que sigue se discute la implementación y el uso de algunos de ellos (**pertenece**, **longitud**, **concatena**, **último** e **inversa**), con el objeto de ilustrar el manejo típico de las listas en PROLOG (**uso de patrones + recursión**) y de destacar su **versatilidad** (predicados con distintos usos dependiendo de si se invocan con ciertos parámetros de entrada o de salida).
- Para comprender mejor el modo en el que PROLOG maneja las listas es muy recomendable *construir los Árboles de Resolución* correspondientes a algunas de las consultas que se proponen a continuación -u otras similares-, comprobando las soluciones obtenidas con las facilitadas por PROLOG.


```
pertenece(?E, ?L) (member en SWI-Prolog)
% cierto si E pertenece a L
```

- Si L es vacía, unificará con el término $[]$, y sea quien sea E , el resultado de “ $?- \text{pertenece}(E, [])$.” debe ser falso. Para ello *basta con no incluir en el programa ninguna cláusula cuya cabeza unifique con* $\text{pertenece}(E, [])$.
- Si L no es vacía, tiene que unificar con el patrón $[C|R]$:
 - Caso base: cierto cuando el elemento buscado coincide con la cabeza de la lista, lo cual se escribe en PROLOG mediante el hecho $\text{pertenece}(C, [C|_])$.
 - Caso recursivo: cierto cuando el elemento buscado *pertenece* al resto de la lista, y esto último se escribe en PROLOG con la regla:
$$\text{pertenece}(E, [_|R]) :- \text{pertenece}(E, R).$$

- En definitiva:

```
pertenece(?E, ?L) (member en SWI-Prolog)
% cierto si E pertenece a L

pertenece(C, [C|_]).

pertenece(C, [_|R]) :-
    pertenece(C, R).
```

Ejemplos (Algunos usos de pertenece (member), 1/2)

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

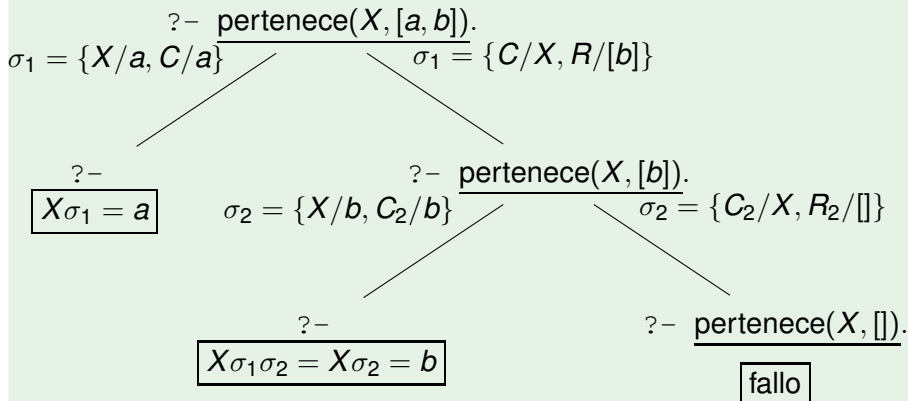
```
?- pertenece(a, L).           % infinitas soluciones
L = [a|_] ; L = [_, a|_] ; L = [_, _, a|_] ; ...
```

Ejemplos (Algunos usos de pertenece (member), 2/2)

% recorrido de una lista mediante backtraking

?- pertenece(X, [a,b]).

X = a ; X = b



Observación (Versatilidad de `pertenece`)

El predicado `pertenece` es un predicado versátil, que sirve para: (1) averiguar si un elemento pertenece a una lista (1er ejemplo anterior) (2) generar listas que contengan un elemento dado (2do ejemplo) (3) recorrer una lista utilizando backtracking (3er ejemplo).

Ejercicios (Predicado `pertenece`)

Dado el código facilitado más arriba para el predicado `pertenece`, construya los Árboles de Resolución correspondientes e indique qué respuestas facilitaría PROLOG, y en qué orden, ante las siguientes consultas:

- ❶ `?- pertenece(a, L) .`
- ❷ `?- pertenece(b, [a,b,c]) .`
- ❸ `?- pertenece(X, [[1,2]]), pertenece(Y, X) .`

```
longitud(?L, ?N) (length en SWI-Prolog)
% cierto si L tiene N elementos
    longitud([], 0).
    longitud([_|R], N) :-
        longitud(R, LR),
        N is LR + 1.
```

Ejemplos (Algunos usos de longitud (length))

```
?- longitud([[a,b,c,d],1], X).
X = 2.
```

```
?- longitud(L, Long).
L = [], Long = 0 ;
L = [_], Long = 1 ;
L = [_, _], Long = 2;
..... (infinitas soluciones)
```

?- longitud([a, b, c, d], 1), X).

C2 $\sigma_1 = \{R/[1], X/N\}$

?- longitud([1], LR), N is LR + 1.

C2 $\sigma_2 = \{R2/[], LR/N2\}$

?- longitud([], LR2), N2 is LR2 + 1, N is N2 + 1.

C1 $\sigma_3 = \{LR2/0\}$

?- N2 is 0 + 1, N is N2 + 1.

$\sigma_4 = \{N2/1\}$

?- N is 1 + 1.

$\sigma_5 = \{N/2\}$

?-

$X\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5 = N\sigma_2\sigma_3\sigma_4\sigma_5 = \dots = N\sigma_5 = 2$

Observación

- Observe que en el caso recursivo de la implementación de `longitud` es necesario sumar 1 a la longitud del resto de la lista, para lo cual es imprescindible:
 - 1 Usar el predicado aritmético `is`, el único capaz de *evaluar* una expresión aritmética y asignar su valor a una variable mediante unificación (no se puede poner simplemente `LR + 1` en la cabeza de la regla).
 - 2 Hacer primero el cálculo recursivo y *después* la evaluación mediante `is` (de lo contrario `is` produciría un error de instanciación), por lo que la recursión es no final.
- El uso de `longitud` con el primer parámetro de salida y el segundo de entrada (por ejemplo `?- longitud(L, 2)`) dará una primera solución y luego entrará en una rama infinita sin soluciones (piense en cómo sería el Árbol de Resolución correspondiente).

La implementación anterior es ineficiente puesto que la recursión no es final. Una implementación alternativa, con recursión final obtenida mediante la introducción de un parámetro de acumulación, sería:

```
% longitud_rc(?L, ?N)
% cierto si L tiene N elementos
% Implementación con recursión de cola

% se añade parámetro de acumulación
longitud_rc(L, N) :-
    longitud(L, 0, N). % 0 = valor caso base

% caso base: se devuelve lo acumulado
longitud([], Ac, Ac).

% caso recursivo: se actualiza el parámetro
longitud(_|R, Ac, N) :-
    NAc is Ac + 1,
    longitud(R, NAc, N).
```


Ejercicios (Predicado `longitud`)

1 Construya el Árbol de Resolución para la consulta

`?- longitud_rc([[a,b,c,d],1],X), Z is X*X.`

(implementación con recursión de cola discutida más arriba).

¿Qué respuestas ofrecería PROLOG, y en qué orden?

2 Construya el Árbol de Resolución para la consulta

`?- longitud(L, X), X >= 1.`

¿Qué respuestas ofrecería PROLOG, y en qué orden?

```
concatena(?L1, ?L2, ?L) (append en SWI-Prolog)  
% cierto si L es la concatenación de L1 y L2
```

- Caso base: si $L1$ es vacía, su concatenación con cualquier otra lista (vacía o no vacía) es esta última, lo cual se escribe en PROLOG mediante el hecho

```
concatena([], L, L).
```

- Caso recursivo: si $L1$ no es vacía tiene que unificar con $[C|R]$, y su concatenación con una lista cualquiera L será $[C|NL]$, siempre y cuando NL sea la concatenación de R con L , lo cual se escribe en PROLOG mediante la siguiente regla recursiva:

```
concatena([C|R], L, [C|NL]) :-  
    concatena(R, L, NL).
```

- En definitiva:

```
concatena(?L1, ?L2, ?L) (append en SWI-Prolog)
% cierto si L es la concatenación de L1 y L2

concatena([], L, L).
concatena([C|R], L, [C|NL]) :-
    concatena(R, L, NL).
```

Observación

Note que la recursión de la implementación anterior no es exactamente de cola, puesto que después de la llamada recursiva queda aún una operación por hacer (añadir el elemento C delante de la lista NL). Esta recursión se denomina **recursión de cola módulo cons** y, a diferencia de lo que ocurre en otros lenguajes, **PROLOG es capaz de optimizar el código** resultante de forma similar al de la recursión de cola normal.

Observación (Versatilidad de `concatena`)

Los ejemplos incluidos a continuación ilustran cómo `concatena`, a pesar de la sencillez de su implementación, es muy flexible y versátil, pudiéndose usar para varios cometidos distintos:

- 1 Para **concatenar** listas.
- 2 Para calcular **prefijos**.
- 3 Para calcular **sufijos**.
- 4 Para **descomponer** una lista en dos sublistas de todas las formas posibles (mediante backtracking).

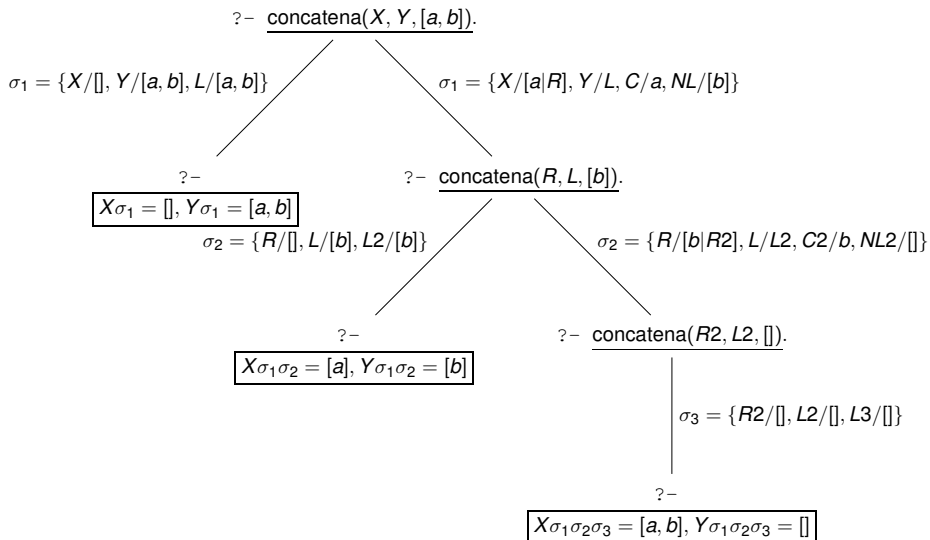
Ejemplos (Algunos usos de concatena (append))

```
?- concatena([a], [b,c], L).      % concatenación  
L = [a, b, c].
```

```
?- concatena([a], L, [a,b,c]).    % sufijo  
L = [b, c].
```

```
?- concatena(L, [b,c], [a,b,c]). % prefijo  
L = [a].
```

```
?- concatena(X, Y, [a,b,c]).      % descomposición  
                                   % en dos sublistas  
                                   % (con backtracking)  
X = [], Y = [a, b, c] ;  
X = [a], Y = [b, c] ;  
X = [a, b], Y = [c] ;  
X = [a, b, c], Y = []
```



Cuidado con el cálculo correcto de los u.m.g.'s: por ejemplo, el primer u.m.g. de la rama de la derecha es $\{X/[a|R], Y/L, C/a, NL/[b]\}$ y no $\{X/[C|R], Y/L, C/a, NL/[b]\}$, puesto que cuando se elige la sustitución $\{C/a\}$, esta se debe **componer** con lo que había hasta el momento, $\{X/[C|R], Y/L\}$.

Observación (Uso de concatena -append- para implementar otros predicados)

La capacidad del predicado de concatenación para dividir una lista en dos sublistas lo convierte en una herramienta muy útil para la implementación de otros predicados que se puedan describir mediante esta división. Por ejemplo:

- El predicado `pertenece(E, L)` se puede también implementar mediante `concatena`: será cierto si `L` se pueda dividir en dos trozos de forma que el segundo de ellos empiece por `E`:

```
pertenece(E, L) :- concatena(_, [E|_], L).
```

- El predicado `prefijo(Pre, L)`, cierto si la lista `P` es un prefijo de `L`, será cierto si existe una lista tal que concatenada con `Pre` da `L`:

```
prefijo(Pre, L) :- concatena(Pre, _, L).
```

Ejercicios (Predicado `concatena`)

- 1 *Razone qué ocurrirá (se producirá un error, la respuesta será false, true, se producirá una computación infinita, la o las respuestas serán) al ejecutar la consulta:*

```
?- concatena(X,_, [a,b]), longitud(X,_N), _N =< 1.
```
- 2 *Implemente de forma recursiva (sin utilizar `concatena`) el predicado `prefijo(Pre, L)` mencionado previamente.*
- 3 *Implemente el predicado `sufijo(Suf, L)`, cierto si la lista `Suf` es un sufijo de `L`, tanto de forma recursiva como mediante el predicado `concatena`.*
- 4 *Describa en lenguaje natural (con sus propias palabras) el resultado (cierto si ...) del predicado*

```
misterio([A|B]) :- concatena(_, [A], [A|B]).
```
- 5 *Construya el Árbol asociado a `?- pertenece(b, [a,b])` suponiendo que `pertenece` está implementado por medio de `concatena` (ver última observación).*


```
% ultimo(?L, ?U) (last en SWI-Prolog)
% cierto si U es el último elemento de la lista L

ultimo([C],C).
ultimo([_|R],U) :-
    ultimo(R,U).
```

Ejemplos (Algunos usos de ultimo (last))

```
?- ultimo([a,b,c], c).
true
?- ultimo([a,b,c], U).
U = c
?- ultimo(L, c).
L = [c]
L = [_, c]
... (infinitas soluciones)
```

```
% inversa(+L, ?LI) (reverse en SWI-Prolog)
% cierto si LI es la inversa de L

inversa([], []).

inversa([C|R], LI) :-
    inversa(R, RI),
    append(RI, [C], LI).
```

Ejemplos (Algunos usos de inversa (reverse))

```
?- inversa([], []).
true
?- inversa([a,b,c], [a,b,c]).
false
?- inversa([a,b,c], I).
I = [c,b,a]
```

La implementación anterior del predicado `inversa` es ineficiente debido a que la recursión no es de cola. Una implementación alternativa, con recursión de cola obtenida introduciendo un parámetro de acumulación:

```
% inversa_rc(+L, ?LI)
% cierto si LI es la inversa de L
% Implementación con recursión de cola.

% se añade parámetro de acumulación
inversa_rc(L, LI) :-
    inversa(L, [], LI). % [] = valor caso base
% caso base: se devuelve lo acumulado
inversa([], Ac, Ac).
% caso recursivo: se actualiza el parámetro
inversa([C|R], Ac, LI) :-
    inversa(R, [C|Ac], LI).
```

Ejercicios (Predicados básicos sobre listas)

- 1 *Ejercicio nº 2 de la **Práctica de PROLOG nº 3** ignorando las preguntas en las que aparece el predicado de corte !.*
- 2 *Dada la consulta
`?- concatena(X,Y,[a]), pertenece(a,X),` describa qué se pretende averiguar con ella, construya el Árbol de Resolución correspondiente e indique qué respuestas ofrece PROLOG.*
- 3 *Implemente el predicado `ultimo` de forma no recursiva mediante el uso del predicado de concatenación.*
- 4 *Implemente de dos formas distintas (una recursiva, la otra usando `concatena`) el predicado `insertarfinal(+L,+E,?NL)`, cierto si NL es la lista resultante después de insertar el elemento E al final de la lista L.*
- 5 *Construya los Árboles de Resolución asociados a algunas de las consultas planteadas en los ejemplos de uso de los predicados básicos sobre listas.*

EJEMPLOS ADICIONALES

Escritura de una lista

- Con el formato $[a_1, a_2, \dots, a_n]$

```
?- read(L), write(L).
```

```
|: [a,b,c]. % lista introducida, acabada en punto
```

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

```
L = [a, b, c].
```

- Un elemento por línea (la primera cláusula es necesaria para que el predicado devuelva 'true').

```
imprime_lista([]).
```

```
imprime_lista([C|R]) :-
```

```
    write(C),
```

```
    nl, % new line
```

```
    imprime_lista(R).
```

Suma de los elementos de una lista

```
% sumalista(+L,?S) (sumlist en SWI-Prolog)
% cierto si S es la suma de los elementos de
% la lista L. S=0 si L=[]
% implementación ineficiente (recursión no de cola)

sumalista([], 0).
sumalista([C|R], S) :-
    sumalista(R, SR),
    S is C + SR.
```

Ejemplos (Algunos usos de sumalista (sumlist))

```
?- sumalista([], 0).           true
?- sumalista([1,3.5,-1], X).   X = 3.5
?- sumalista([1,3.5,c], X).    ERROR ....
```

Implementación eficiente, con recursión de cola obtenida introduciendo un parámetro de acumulación:

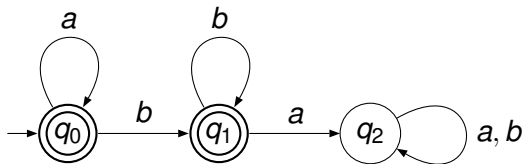
```
% sumalista(+L,?S) (sumlist en SWI-Prolog)
% cierto si S es suma de los elementos de
% la lista L. S=0 si L=[]

% se añade parámetro de acumulación
sumalistaRC(L, S) :-
    sumalista(L, 0, S). % 0 = valor caso base

% caso base: se devuelve lo acumulado
sumalista([], Ac, Ac).

% caso recursivo: se actualiza lo acumulado
sumalista([C|R], Ac, S) :-
    NAc is Ac + C,
    sumalista(R, NAc, S).
```

Implementación de autómatas finitos deterministas (AFDs)



$$L(A) = \{a^n b^m : n, m \geq 0\}$$


```
%% DATOS DEL AFD CONCRETO
```

```
% inicial(?E), cierto si E es el estado inicial  
    inicial(q0).
```

```
% final(?E), cierto si E es un estado final  
    final(q0).  
    final(q1).
```

```
% transicion(?E1, ?S, ?E2), cierto si el autómata  
% transita de 'E1' a 'E2' leyendo el símbolo 'S'  
    transicion(q0, a, q0).  
    transicion(q0, b, q1).  
    transicion(q1, a, q2).  
    transicion(q1, b, q1).  
    transicion(q2, a, q2).  
    transicion(q2, b, q2).
```

%%% FUNCIONAMIENTO DE UN AFD CUALQUIERA

% acepta(?L)

% cierto si el autómata acepta la palabra 'L'

% (representada mediante una lista de símbolos)

 acepta(L) :-

 inicial(I),

 acepta(I, L).

% acepta(?E, ?L)

% cierto si el autómata, partiendo del estado E,

% acepta la palabra 'L'

 acepta(E, []) :-

 final(E).

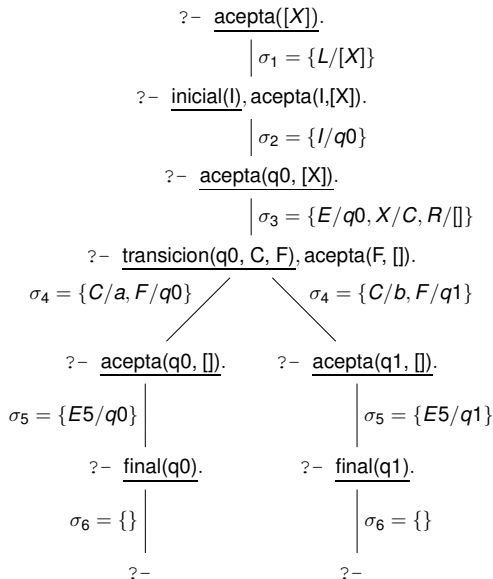
 acepta(E, [C|R]) :-

 transicion(E, C, F),

 acepta(F, R).

Ejemplos (Algunos usos de `acepta`)

```
?- acepta([]).  
true  
?- acepta([a]).  
true  
?- acepta([b]).  
true  
?- acepta([b,a]).  
false  
?- acepta([a,b]).  
true  
?- acepta([X]). % palabras aceptadas de longitud 1  
X = a ;  
X = b
```



$$\underline{X\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6} = \dots = \underline{C\sigma_4\sigma_5\sigma_6} = a$$

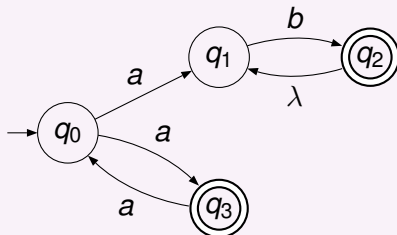
$$\underline{X\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6} = \dots = \underline{C\sigma_4\sigma_5\sigma_6} = b$$

Ejercicios (Manejo de listas, 1/2)

- 1 *Modifique las dos implementaciones propuestas para `sumalista` de modo que el predicado falle, en lugar de ser cierto con $S=0$, en el caso de que la lista sea vacía.*
- 2 *Escriba la consulta adecuada para calcular qué palabras de longitud dos acepta el autómata implementado más arriba y construya el Árbol de Resolución asociado para averiguar qué respuestas ofrece PROLOG, y en qué orden las da.*
- 3 *Haga el ejercicio nº 3 de la **Práctica de PROLOG nº 3** ignorando lo relacionado con el predicado de corte !.*
- 4 *Modifique el programa para AFDs discutido más arriba para que sea capaz de simular autómatas finitos no deterministas (AFNDs). Recuerde que las palabras aceptadas por un AFND son aquellas para las que el autómata puede transitar desde el estado inicial hasta algún estado final, pudiendo usar transiciones vacías (transiciones λ). Ver ejemplo a continuación.*

Ejercicios (Manejo de listas, 2/2)

El siguiente grafo representa un AFND que acepta todas las palabras de la forma $a^n b^m$ tales que $n \bmod 2 \neq 0$, $m \geq 0$, y ninguna otra:



Para adaptar el programa de AFDs a AFNDs, tendrá que decidir cómo representar las transiciones λ y cómo permitir movimientos mediante esas transiciones. Pruebe su programa con ejemplos significativos, y haga el árbol de Resolución necesario para averiguar qué palabras de longitud 3 acepta el AFND anterior.

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. **Logic, Programming and Prolog**. John Wiley & Sons Ltd, 1996.
- **SWI-Prolog**, entorno de programación en Prolog de dominio público.
- **comp.lang.prolog. Faq**

© 2022 Ana Pradera Gómez

Algunos derechos reservados

Este documento se distribuye bajo la licencia

“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,
disponible en

<https://creativecommons.org/licenses/by-sa/4.0/deed.es>