

Programación declarativa (2007–08)

Tema 11: Estilo y eficiencia en programación lógica

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

- Criterios para juzgar la bondad de un programa
- Heurísticas para la programación (Ref. Polya)

2. Métodos de mejora de la eficiencia

- Orden de los literales
- Generación y prueba
- Uso de listas o términos compuestos
- Uso de la unificación implícita
- Acumuladores
- Uso de lemas
- Determinismo
- Añadir al principio
- Listas de diferencias

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

Criterios para juzgar la bondad de un programa

Heurísticas para la programación (Ref. Polya)

2. Métodos de mejora de la eficiencia

Criterios para juzgar la bondad de un programa

- ▶ Corrección.
- ▶ Eficiencia.
- ▶ Facilidad de lectura.
- ▶ Modificabilidad: Modular y transparente.
- ▶ Robustez.
- ▶ Documentación.

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

Criterios para juzgar la bondad de un programa

Heurísticas para la programación (Ref. Polya)

2. Métodos de mejora de la eficiencia

Heurísticas para la programación (Ref. Polya)

1. Entender el problema.
2. Diseñar una solución.
3. Escribir el programa.
4. Revisar la solución

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Orden de los literales y corrección

► Programas:

```
siguiente(a,1).  
siguiente(1,b).
```

```
sucesor_1(X,Y) :-  
    siguiente(X,Y).  
sucesor_1(X,Y) :-  
    siguiente(X,Z),  
    sucesor_1(Z,Y).
```

```
?- sucesor_1(X,Y).  
X = a Y = 1 ; X = 1 Y = b ; X = a Y = b ; No  
?- findall(X-Y,sucesor_1(X,Y),L).  
L = [a-1, 1-b, a-b]
```


Orden de los literales y corrección

► Programas:

```
siguiente(a,1).  
siguiente(1,b).
```

```
sucesor_2(X,Y) :-  
    siguiente(X,Y).  
sucesor_2(X,Y) :-  
    sucesor_2(Z,Y),  
    siguiente(X,Z).
```

```
| ?- sucesor_2(X,Y).
```

```
| X = a   Y = 1 ; X = 1   Y = b ; X = a   Y = b ;
```

```
| ERROR: Out of local stack
```

Orden de los literales y eficiencia

► Orden de los literales y eficiencia

```
?- listing(número).  
número(1). número(2). ... número(1000).
```

```
?- listing(múltiplo_de_100).  
múltiplo_de_100(100). ... múltiplo_de_100(1000).
```

```
?- time((número(_N),múltiplo_de_100(_N))).  
101 inferences in 0.00 seconds  
Yes
```

```
?- time((múltiplo_de_100(_N),número(_N))).  
2 inferences in 0.00 seconds  
Yes
```

Combinaciones

- combinación(+L1,+N,-L2) se verifica si L2 es una combinación N-aria de L1. Por ejemplo,

```
?- combinación([a,b,c],2,L).
```

```
L = [a, b] ;    L = [a, c] ;    L = [b, c] ;    No
```

```
combinación_1(L1,N,L2) :-  
    subconjunto(L2,L1),  
    length(L2,N).
```

```
combinación_2(L1,N,L2) :-  
    length(L2,N),  
    subconjunto(L2,L1).
```

```
combinación(L1,N,L2) :-  
    combinación_2(L1,N,L2).
```

Combinaciones

- `combinaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las combinaciones `N`-arias de `L1`. Por ejemplo,

```
?- combinaciones([a,b,c],2,L).  
L = [[a, b], [a, c], [b, c]]
```

```
combinaciones_1(L1,N,L2) :-  
    findall(L,combinación_1(L1,N,L),L2).
```

```
combinaciones_2(L1,N,L2) :-  
    findall(L,combinación_2(L1,N,L),L2).
```

```
combinaciones(L1,N,L2) :-  
    combinaciones_2(L1,N,L2).
```

Comparación de eficiencia de combinaciones

```
?- findall(_N,between(1,6,_N),_L1),  
    time(combinaciones_1(_L1,2,_L2)),  
    time(combinaciones_2(_L1,2,_L2)).  
429 inferences in 0.00 seconds  
92 inferences in 0.00 seconds  
?- findall(_N,between(1,12,_N),_L1),  
    time(combinaciones_1(_L1,2,_L2)),  
    time(combinaciones_2(_L1,2,_L2)).  
28,551 inferences in 0.01 seconds  
457 inferences in 0.00 seconds  
?- findall(_N,between(1,24,_N),_L1),  
    time(combinaciones_1(_L1,2,_L2)),  
    time(combinaciones_2(_L1,2,_L2)).  
117,439,971 inferences in 57.59 seconds  
2,915 inferences in 0.00 seconds
```

Permutaciones

- `select(?X,?L1,?L2)` se verifica si X es un elemento de la lista L1 y L2 es la lista de los restantes elementos. Por ejemplo,

```
?- select(X,[a,b,c],L).
```

```
X = a    L = [b, c] ;
```

```
X = b    L = [a, c] ;
```

```
X = c    L = [a, b] ;
```

```
No
```

```
?- select(a,L,[b,c]).
```

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

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

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

```
No
```

Permutaciones

- `permutación(+L1,-L2)` se verifica si `L2` es una permutación de `L1`. Por ejemplo,

```
?- permutación([a,b,c],L).
L = [a, b, c] ;   L = [a, c, b] ;
L = [b, a, c] ;   L = [b, c, a] ;
L = [c, a, b] ;   L = [c, b, a] ;
No
```

```
permutación([],[]).
permutación(L1,[X|L2]) :-
    select(X,L1,L3),
    permutación(L3,L2).
```

Predefinida `permutation`.

Variaciones

- `variación(+L1,+N,-L2)` se verifica si `L2` es una variación `N`-aria de `L1`. Por ejemplo,

```
?- variación([a,b,c],2,L).
```

```
L=[a,b];L=[a,c];L=[b,a];L=[b,c];L=[c,a];L=[c,b];No
```

```
variación_1(L1,N,L2) :-  
    combinación(L1,N,L3), permutación(L3,L2).
```

```
variación_2(_,0,[]).  
variación_2(L1,N,[X|L2]) :-  
    N > 0, M is N-1,  
    select(X,L1,L3),  
    variación_2(L3,M,L2).
```

```
variación(L1,N,L2) :- variación_2(L1,N,L2).
```


Variaciones

- `variaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las variaciones `N`-arias de `L1`. Por ejemplo,

```
?- variaciones([a,b,c],2,L).  
L = [[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]
```

```
variaciones_1(L1,N,L2) :-  
    setof(L, variación_1(L1,N,L), L2).
```

```
variaciones_2(L1,N,L2) :-  
    setof(L, variación_2(L1,N,L), L2).
```

```
variaciones(L1,N,L2) :-  
    variaciones_2(L1,N,L2).
```

Comparación de eficiencia de variaciones

```
?- findall(N,between(1,100,N),L1),  
   time(variaciones_1(L1,2,L2)),  
   time(variaciones_2(L1,2,L2)).  
221,320 inferences in 0.27 seconds  
40,119 inferences in 0.11 seconds  
?- findall(N,between(1,200,N),L1),  
   time(variaciones_1(L1,2,L2)),  
   time(variaciones_2(L1,2,L2)).  
1,552,620 inferences in 2.62 seconds  
160,219 inferences in 0.67 seconds  
?- findall(N,between(1,400,N),L1),  
   time(variaciones_1(L1,2,L2)),  
   time(variaciones_2(L1,2,L2)).  
11,545,220 inferences in 19.02 seconds  
640,419 inferences in 2.51 seconds
```

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Ordenación por generación y prueba

- `ordenación(+L1,-L2)` se verifica si `L2` es la lista obtenida ordenando la lista `L1` en orden creciente. Por ejemplo,

```
?- ordenación([2,1,a,2,b,3],L).  
L = [a,b,1,2,2,3]
```

```
ordenación(L,L1) :-  
    permutación(L,L1),  
    ordenada(L1).
```

```
ordenada([]).  
ordenada([_]).  
ordenada([X,Y|L]) :-  
    X @=< Y,  
    ordenada([Y|L]).
```

Ordenación por selección

```
ordenación_por_selección(L1,[X|L2]) :-  
    selecciona_menor(X,L1,L3),  
    ordenación_por_selección(L3,L2).  
ordenación_por_selección([],[]).
```

```
selecciona_menor(X,L1,L2) :-  
    select(X,L1,L2),  
    not((member(Y,L2), Y @< X)).
```

Ordenación por divide y vencerás

```
ordenación_rápida([], []).
```

```
ordenación_rápida([X|R], Ordenada) :-  
    divide(X, R, Menores, Mayores),  
    ordenación_rápida(Menores, Menores_ord),  
    ordenación_rápida(Mayores, Mayores_ord),  
    append(Menores_ord, [X|Mayores_ord], Ordenada).
```

```
divide(_, [], [], []).
```

```
divide(X, [Y|R], [Y|Menores], Mayores) :-
```

```
    Y @< X, !,
```

```
    divide(X, R, Menores, Mayores).
```

```
divide(X, [Y|R], Menores, [Y|Mayores]) :-
```

```
    \% Y @>= X,
```

```
    divide(X, R, Menores, Mayores).
```

Ordenación: comparación de eficiencia

Comparación de la ordenación de la lista $[N, N-1, N-2, \dots, 2, 1]$

N	ordena	selección	rápida
1	5 inf 0.00 s	8 inf 0.00 s	5 inf 0.00 s
2	10 inf 0.00 s	19 inf 0.00 s	12 inf 0.00 s
4	80 inf 0.00 s	67 inf 0.00 s	35 inf 0.00 s
8	507,674 inf 0.33 s	323 inf 0.00 s	117 inf 0.00 s
16		1,923 inf 0.00 s	425 inf 0.00 s
32		13,059 inf 0.01 s	1,617 inf 0.00 s
64		95,747 inf 0.05 s	6,305 inf 0.00 s
128		732,163 inf 0.40 s	24,897 inf 0.01 s
256		5,724,163 inf 2.95 s	98,945 inf 0.05 s
512		45,264,899 inf 22.80 s	394,497 inf 0.49 s

Cuadrado mágico por generación y prueba

- Enunciado: Colocar los números 1,2,3,4,5,6,7,8,9 en un cuadrado 3x3 de forma que todas las líneas (filas, columnas y diagonales) sumen igual.

A	B	C
D	E	F
G	H	I

```
cuadrado_1([A,B,C,D,E,F,G,H,I]) :-  
    permutación([1,2,3,4,5,6,7,8,9],  
                [A,B,C,D,E,F,G,H,I]),  
    A+B+C == 15,    D+E+F == 15,  
    G+H+I == 15,    A+D+G == 15,  
    B+E+H == 15,    C+F+I == 15,  
    A+E+I == 15,    C+E+G == 15.
```


Cuadrado mágico por generación y prueba

► Cálculo de soluciones:

```
?- cuadrado_1(L).
```

```
L = [6, 1, 8, 7, 5, 3, 2, 9, 4] ;
```

```
L = [8, 1, 6, 3, 5, 7, 4, 9, 2]
```

```
Yes
```

► Cálculo del número soluciones:

```
?- findall(_X,cuadrado_1(_X),_L),length(_L,N).
```

```
N = 8
```

```
Yes
```

Cuadrado mágico por comprobaciones parciales

► Programa 2:

```
cuadrado_2([A,B,C,D,E,F,G,H,I]) :-  
    select(A,[1,2,3,4,5,6,7,8,9],L1),  
    select(B,L1,L2),  
    select(C,L2,L3),    A+B+C == 15,  
    select(D,L3,L4),  
    select(G,L4,L5),    A+D+G == 15,  
    select(E,L5,L6),    C+E+G == 15,  
    select(I,L6,L7),    A+E+I == 15,  
    select(F,L7,[H]),   C+F+I == 15, D+E+F == 15.
```

Cuadrado mágico por comprobaciones parciales

► Cálculo de soluciones:

```
?- cuadrado_2(L).  
L = [2, 7, 6, 9, 5, 1, 4, 3, 8] ;  
L = [2, 9, 4, 7, 5, 3, 6, 1, 8]  
Yes
```

► Comprobación que las dos definiciones dan las *mismas* soluciones.

```
?- setof(_X,cuadrado_1(_X),_L),  
   setof(_X,cuadrado_2(_X),_L).  
Yes
```

Comparación de eficiencia del cuadrado mágico

```
?- time(cuadrado_1(_X)).  
161,691 inferences in 0.58 seconds
```

```
?- time(cuadrado_2(_X)).  
1,097 inferences in 0.01 seconds
```

```
?- time(setof(_X,cuadrado_1(_X),_L)).  
812,417 inferences in 2.90 seconds
```

```
?- time(setof(_X,cuadrado_2(_X),_L)).  
7,169 inferences in 0.02 seconds
```

- └ Métodos de mejora de la eficiencia
- └ Uso de listas o términos compuestos

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Uso de listas o términos compuestos

► Ejemplos:

```
?- setof(N,between(1,1000,N),L1),
   asserta(con_lista(L1)),
   Term =.. [f|L1],
   asserta(con_term(Term)).

?- listing(con_lista).
con_lista([1, 2, ..., 999, 1000]).

?- listing(con_term).
con_term(f(1, 2,...,999, 1000)).

?- time((con_lista(_L),member(1000,_L))).
1,001 inferences in 0.00 seconds

?- time((con_term(_T),arg(_,_T,1000))).
2 inferences in 0.00 seconds
```

- En general, sólo usar listas cuando el número de argumentos no es fijo o es desconocido.

- └ Métodos de mejora de la eficiencia
- └ Uso de la unificación implícita

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Uso de la unificación implícita

- `intercambia(+T1,-T2)` se verifica si `T1` es un término con dos argumentos y `T2` es un término con el mismo símbolo de función que `T1` pero sus argumentos intercambiados. Por ejemplo,

```
?- intercambia(opuesto(3,-3),T).  
| T = opuesto(-3, 3)
```

```
intercambia_1(T1, T2) :-  
    functor(T1,F,2),    functor(T2,F,2),  
    arg(1,T1,X1),       arg(2,T1,Y1),  
    arg(1,T2,X2),       arg(2,T2,Y2),  
    X1 = Y2,            X2 = Y1.
```

```
intercambia_2(T1,T2) :-  
    T1 =.. [F,X,Y],  
    T2 =.. [F,Y,X].
```


Uso de la unificación implícita

- ▶ `lista_de_tres(L)` se verifica si `L` es una lista de 3 elementos.
- ▶ Definición 1:

```
lista_de_tres(L) :-  
    length(L, N),  
    N = 3.
```

- ▶ Definición 2 (sin unificación explícita):

```
lista_de_tres(L) :-  
    length(L,3).
```

- ▶ Definición 3 (sin `length`):

```
lista_de_tres([_,_,_]).
```

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Acumuladores

- ▶ `inversa(+L1,-L2)`, `reverse(L1,L2)`, se verifica si `L2` es la lista inversa de `L1`. Por ejemplo,

```
?- inversa([a,b,c],L).  
L = [c, b, a]
```

- ▶ Definición de `inversa` con `append` (no recursiva final):

```
inversa_1([],[]).  
inversa_1([X|L1],L2) :-  
    inversa_1(L1,L3),  
    append(L3,[X],L2).
```

Acumuladores

- Definición de `inversa` con acumuladores (recursiva final):

```
inversa_2(L1,L2) :-  
    inversa_2_aux(L1, [], L2).
```

```
inversa_2_aux([], L, L).  
inversa_2_aux([X|L], Acum, L2) :-  
    inversa_2_aux(L, [X|Acum], L2).
```

Comparación de eficiencia

```
?- findall(_N,between(1,1000,_N),_L1),  
    time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).  
501,501 inferences in 0.40 seconds  
    1,002 inferences in 0.00 seconds  
  
?- findall(_N,between(1,2000,_N),_L1),  
    time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).  
2,003,001 inferences in 1.59 seconds  
    2,002 inferences in 0.00 seconds  
  
?- findall(_N,between(1,4000,_N),_L1),  
    time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).  
8,006,001 inferences in 8.07 seconds  
    4,002 inferences in 0.02 seconds
```

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Uso de lemas

- `fibonacci(N,X)` se verifica si `X` es el `N`-ésimo término de la sucesión de Fibonacci, definida por

$$f(1) = 1$$

$$f(2) = 1$$

$$f(n) = f(n-1)+f(n-2), \text{ si } n > 2$$

```
fibonacci_1(1,1).
```

```
fibonacci_1(2,1).
```

```
fibonacci_1(N,F) :-
```

```
    N > 2,
```

```
    N1 is N-1,
```

```
    fibonacci_1(N1,F1),
```

```
    N2 is N-2,
```

```
    fibonacci_1(N2,F2),
```

```
    F is F1 + F2.
```

Uso de lemas

► Definición de Fibonacci con lemas

```
:- dynamic fibonacci_2/2.
```

```
fibonacci_2(1,1).
```

```
fibonacci_2(2,1).
```

```
fibonacci_2(N,F) :-
```

```
    N > 2,
```

```
    N1 is N-1,
```

```
    fibonacci_2(N1,F1),
```

```
    N2 is N-2,
```

```
    fibonacci_2(N2,F2),
```

```
    F is F1 + F2,
```

```
    asserta(fibonacci_2(N,F)).
```

Comparación de eficiencia en el uso de lemas

```
?- time(fibonacci_1(20,N)).  
40,585 inferences in 1.68 seconds  
N = 6765
```

```
?- time(fibonacci_2(20,N)).  
127 inferences in 0.00 seconds  
N = 6765
```

```
?- listing(fibonacci_2).  
fibonacci_2(20, 6765). fibonacci_2(19, 4181). ...
```

```
?- time(fibonacci_2(20,N)).  
3 inferences in 0.00 seconds  
N = 6765
```

Uso de lemas y acumuladores

► Definición de Fibonacci con un acumulador

```
fibonacci_3(N,F) :-  
    fibonacci_3_aux(N,_,F).
```

`fibonacci_3_aux(+N,-F1,-F2)` se verifica si `F1` es fibonacci de `N-1` y `F2` es fibonacci de `N`.

```
fibonacci_3_aux(0,_,0).  
fibonacci_3_aux(1,0,1).  
fibonacci_3_aux(N,F1,F) :-  
    N > 1,  
    N1 is N-1,  
    fibonacci_3_aux(N1,F2,F1),  
    F is F1 + F2.
```

Comparación de eficiencia en el uso de lemas y acumuladores

► Comparación;

```
?- time(fibonacci_1(20,N)).  
40,585 inferences in 1.68 seconds
```

```
?- time(fibonacci_2(20,N)).  
127 inferences in 0.00 seconds
```

```
?- time(fibonacci_3(20,N)).  
21 inferences in 0.00 seconds
```

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Determinismo

- ▶ `descompone(+E,-N1,-N2)` se verifica si $N1$ y $N2$ son dos enteros no negativos tales que $N1+N2=E$.
- ▶ Definición no determinista:

```
descompone_1(E, N1, N2) :-  
    between(0, E, N1),  
    between(0, E, N2),  
    E ::= N1 + N2.
```

Determinismo

► Definición determinista:

```
descompone_2(E, N1, N2) :-  
    between(0, E, N1),  
    N2 is E - N1.
```

► Comparación de eficiencia:

```
?- time(setof(_N1+_N2,descompone_1(1000,_N1,_N2),_L)).  
1,004,019 inferences in 1.29 seconds
```

```
?- time(setof(_N1+_N2,descompone_2(1000,_N1,_N2),_L)).  
2,018 inferences in 0.01 seconds
```

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Añadir al principio

- ▶ `lista_de_cuadrados(+N,?L)` se verifica si `L` es la lista de los cuadrados de los números de 1 a `N`. Por ejemplo,

```
?- lista_de_cuadrados(5,L).  
L = [1, 4, 9, 16, 25]
```

- ▶ Programa 1 (añadiendo por detrás):

```
lista_de_cuadrados_1(1,[1]).  
lista_de_cuadrados_1(N,L) :-  
    N > 1,  
    N1 is N-1,  
    lista_de_cuadrados_1(N1,L1),  
    M is N*N,  
    append(L1,[M],L).
```

Añadir al principio

- Programa 2 (añadiendo por delante):

```
lista_de_cuadrados_2(N,L) :-  
    lista_de_cuadrados_2_aux(N,L1),  
    reverse(L1,L).  
  
lista_de_cuadrados_2_aux(1,[1]).  
lista_de_cuadrados_2_aux(N,[M|L]) :-  
    N > 1,  
    M is N*N,  
    N1 is N-1,  
    lista_de_cuadrados_2_aux(N1,L).  

```

Añadir al principio

- Programa 3 (con `findall`):

```
lista_de_cuadrados_3(N,L) :-  
    findall(M,(between(1,N,X), M is X*X),L).
```

Añadir al principio

► Comparación:

```
?- time(lista_de_cuadrados_1(10000,_L)).  
50,044,996 inferences in 24.34 seconds  
Yes
```

```
?- time(lista_de_cuadrados_2(10000,_L)).  
50,000 inferences in 0.06 seconds  
Yes
```

```
?- time(lista_de_cuadrados_3(10000,_L)).  
20,012 inferences in 0.04 seconds  
Yes
```

- └ Métodos de mejora de la eficiencia
- └ Listas de diferencias

Tema 11: Estilo y eficiencia en programación lógica

1. Principios generales de buena programación

2. Métodos de mejora de la eficiencia

Orden de los literales

Generación y prueba

Uso de listas o términos compuestos

Uso de la unificación implícita

Acumuladores

Uso de lemas

Determinismo

Añadir al principio

Listas de diferencias

Listas de diferencias

- Representaciones de $[a,b,c]$ como listas de diferencias:

$[a,b,c,d] - [d]$

$[a,b,c,1,2,3] - [1,2,3]$

$[a,b,c|X] - X$

$[a,b,c] - []$

- `conc_ld(LD1,LD2,LD3)` se verifica si LD3 es la lista de diferencia correspondiente a la concatenación de las listas de diferencia LD1 y LD2. Por ejemplo,

`?- conc_ld([a,b|RX]-RX,[c,d|RY]-RY,Z-[]).`

`RX = [c, d] RY = [] Z = [a, b, c, d]`

`?- conc_ld([a,b|_RX]-_RX,[c,d|_RY]-_RY,Z-[]).`

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

`conc_ld(A-B,B-C,A-C).`

Bibliografía

1. I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
 - ▶ Cap. 8: “Programming Style and Technique”
2. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
 - ▶ Cap. 6: “Using Data Structures”
3. M.A. Covington *Efficient Prolog: A Practical Guide*