Ejercicios de programación declarativa con Prolog

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 1 de Enero de 2006 (Versión de 7 de noviembre de 2006)

Índice general

Introducción

El objetivo del presente trabajo es presentar una colección de ejercicios para la asignatura "Programación declarativa" de tercer curso de la Ingeniería Informática.

Estos ejercicios complementa los apuntes de introducción a la programación declarativa con Prolog ([1]) y a las transparencias de clase ([2]).

Todos los ejercicios se han comprobado usando la versión 5.6.18 de SWI Prolog.

Capítulo 1

Operaciones con listas

Una lista es la lista vacía o se compone de un primer elemento y un resto, que es una lista. En Prolog, la lista vacía se representa por `[]` y las listas no vacía son de la forma `[X|L]` donde `X` es la cabeza y `L` es el resto.

1.1. Primer elemento

Ejercicio 1.1 *Definir la relación* `primero(?L,?X)` *que se verifique si* `X` *es el primer elemento de la lista* `L`. *Por ejemplo,*

```
?- primero([a,b,c],X).
X = a
```

Obtener las respuestas a las siguientes preguntas:

```
?- primero([X,b,c],a).
?- primero([X,Y],a).
?- primero(X,a).
```

Solución: La definición de `primero` es

```
primero([X|_],X).
```

Las respuestas a las preguntas son

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

1.2. Resto de una lista

Ejercicio 1.2 *Definir la relación* `resto(?L1,?L2)` *que se verifique si* `L2` *es la lista obtenida a partir de la lista* `L1` *suprimiendo el primer elemento. Por ejemplo,*

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

Obtener las respuestas a las siguientes preguntas:

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

Solución: La definición de `resto` es

```
resto([_|L],L).
```

Las respuestas a las preguntas son

```
?- resto([a|L],[b,c]).
L = [b, c]
?- resto(L,[b,c]).
L = [X, b, c]
```

1.3. Construcción de listas

Ejercicio 1.3 *Definir la relación* `cons(?X,?L1,?L2)` *que se verifique si* `L2` *es la lista obtenida añadiéndole* `X` *a* `L1` *como primer elemento. Por ejemplo,*

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

Obtener las respuestas correspondientes a las siguientes preguntas:

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

Solución: La definición de `cons` es

```
cons(X,L,[X|L]).
```

Las respuestas a las preguntas son

```
?- cons(X,[b,c],[a,b,c]).
X = a
?- cons(a,L,[a,b,c]).
L = [b, c]
?- cons(b,L,[a,b,c]).
No
?- cons(X,L,[a,b,c]).
X = a
L = [b, c]
```

1.4. Relación de pertenencia

Ejercicio 1.4 *Definir la relación* `pertenece(?X,?L)` *que se verifique si* `X` *es un elemento de la lista* `L`. *Por ejemplo,*

```
?- pertenece(b,[a,b,c]).
Yes
?- pertenece(d,[a,b,c]).
No
```

Utilizar el programa para responder a las siguientes cuestiones:

1. *¿Es* `c` *un elemento de* `[a,c,b,c]`*?*
2. *¿Cuáles son los elementos de* `[a,b,a]`*?*
3. *¿Cuáles son los elementos comunes de* `[a,b,c]`*, y* `[d,c,b]`*?*

Solución: La definición de `pertenece(X,L)`, por recursión en `L`, es

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

Las respuesta a las preguntas son

1. ¿Es `c` un elemento de `[a,c,b,c]`?

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

2. ¿Cuáles son los elementos de `[a,b,a]`?

```
?- pertenece(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
No
```

3. ¿Cuáles son los elementos comunes de `[a,b,c]`, y `[d,c,b]`?

```
?- pertenece(X,[a,b,c]), pertenece(X,[d,c,b]).
X = b ;
X = c ;
No
```

Nota: La relación `pertenece` se corresponde con la definida `member`.

1.5. Concatenación de listas

Ejercicio 1.5 *Definir la relación* `conc(?L1,?L2,?L3)` *que se verifique si* `L3` *es la lista obtenida escribiendo los elementos de* `L2` *a continuación de los elementos de* `L1`*. Por ejemplo,*

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

Utilizar el programa para responder a las siguientes cuestiones:

1. *¿Qué lista hay que añadirle al lista* `[a,b]` *para obtener* `[a,b,c,d]`*?*
2. *¿Qué listas hay que concatenar para obtener* `[a,b]`*?*
3. *¿Pertenece* `b` *a la lista* `[a,b,c]`*?*
4. *¿Es* `[b,c]` *una sublista de* `[a,b,c,d]`*?*
5. *¿Es* `[b,d]` *una sublista de* `[a,b,c,d]`*?*
6. *¿Cuál es el último elemento de* `[b,a,c,d]`*?*

Solución: La definición de `conc(L1,L2,L3)`, por recursión en `L1`, es

```
conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

Las repuestas a las cuestiones son

1. ¿Qué lista hay que añadirle al lista `[a,b]` para obtener `[a,b,c,d]`?

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

2. ¿Qué listas hay que concatenar para obtener `[a,b]`?

```
?- conc(L1,L2,[a,b]).
L1 = []
L2 = [a, b] ;
L1 = [a]
L2 = [b] ;
L1 = [a, b]
L2 = [] ;
No
```

3. ¿Pertenece `b` a la lista `[a,b,c]`?

```
?- conc(L1,[b|L2],[a,b,c]).
L1 = [a]
L2 = [c]
Yes
?- conc(_,[b|_],[a,b,c]).
Yes
```

4. ¿Es [b,c] una sublista de [a,b,c,d]?

```
?- conc(_,[b,c|_],[a,b,c,d]).
Yes
```

5. ¿Es [b,d] una sublista de [a,b,c,d]?

```
?- conc(_,[b,d|_],[a,b,c,d]).
No
```

6. ¿Cuál es el último elemento de [b,a,c,d]?

```
?- conc(_,[X],[b,a,c,d]).
X = d
```

Nota: La relación conc se corresponde con la definida append.

1.6. Lista inversa

Ejercicio 1.6 *Definir la relación* inversa(+L1,-L2) *que se verifique si* L2 *es la lista obtenida invirtiendo el orden de los elementos de la lista* L1. *Por ejemplo,*

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

Solución: Vamos a presentar dos definiciones de inversa(L1,L2). Ambas son por recursión en L1.

Primera solución: Usando la relación append, se define inversa como

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

Segunda solución: Usando un acumulador, se define inversa como

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

La relación inversa_2_aux(+L1,-L2,+L3) se verifica si L2 es la lista obtenida añadiendo la inversa de L1 a L3 y se define por recursión en L1 como sigue

```
inversa_2_aux([],L2,L2).
inversa_2_aux([X|L1],L2,L3) :-
   inversa_2_aux(L1,L2,[X|L3]).
```

Nota: La relación inversa se corresponde con la relación definida reverse.

1.7. Palíndromo

Ejercicio 1.7 *Un palíndromo es una palabra que se lee igual en los dos sentidos, por ejemplo "oso". Definir la relación* `palíndromo(+L)` *que se verifique si la lista* `L` *es un palíndromo. Por ejemplo,*

```
?- palíndromo([o,s,o]).
Yes
?- palíndromo([o,s,a]).
No
```

Solución: La definición de `palíndromo` es

```
palíndromo(L) :-
   reverse(L,L).
```

1.8. Último elemento

Ejercicio 1.8 *Definir la relación* `último(?X,?L)` *que se verifique si* `X` *es el último elemento de la lista* `L`. *Por ejemplo,*

```
?- último(X,[a,b,c,d]).
X = d
?- último(a,L).
L = [a] ;
L = [X, a] ;
L = [X, Y, a]
Yes
```

Solución: Presentamos tres definiciones de `último`.
Primera solución: Usando `append` se define `último` por

```
último_1(X,L) :-
   append(_,[X],L).
```

Segunda solución: Usando `reverse` se define `último` por

```
último_2(X,L) :-
   reverse(L,[X|_]).
```

Tercera solución: Una definición de `último(X,L)` por recursión en `L` es

```
último_3(X,[X]).
último_3(X,[_|L]) :-
   último_3(X,L).
```

Nota: La relación `último` se corresponde con la relación definida `last`.

1.9. Penúltimo elemento

Ejercicio 1.9 *Definir lar relación* `penúltimo(?X,?L)` *que se verifique si* `X` *es el penúltimo elemento de la lista* `L`. *Por ejemplo,*

```
?- penúltimo(X,[a,b,c,d]).
X = c
?- penúltimo(c,L).
L = [c, X] ;
L = [X, c, Y]
Yes
```

Solución: Se presentan tres definiciones de `penúltimo`.
Primera solución: Usando `append` se define `penúltimo` por

```
penúltimo_1(X,L) :-
   append(_,[X,_],L).
```

Segunda solución: Usando `reverse` se define `penúltimo` por

```
penúltimo_2(X,L) :-
   reverse(L,[_,X|_]).
```

Tercera solución: Una definición de `penúltimo(X,L)` por recursión en `L` es

```
penúltimo_3(X,[X,_]).
penúltimo_3(X,[_,Y|L]) :-
   penúltimo_3(X,[Y|L]).
```

1.10. Selección de un elemento

Ejercicio 1.10 *Definir la relación* `selecciona(?X,?L1,?L2)` *que se verifique si* `L2` *es la lista obtenida eliminando una ocurrencia de* `X` *en* `L1`. *Por ejemplo,*

```
?- selecciona(a,[a,b,a],L).
L = [b, a] ;
L = [a, b] ;
No
?- selecciona(c,[a,b,a],L).
No
?- selecciona(a,L,[1,2]).
L = [a, 1, 2] ;
L = [1, a, 2] ;
L = [1, 2, a] ;
No
?- selecciona(X,[1,2,3],[1,3]).
X = 2 ;
No
```

Solución: La definición de `selecciona(X,L1,L2)`, por recursión en `L1`, es

```
selecciona(X,[X|L],L).
selecciona(X,[Y|L1],[Y|L2]) :-
   selecciona(X,L1,L2).
```

Nota: La relación `selecciona` se corresponde con la definida `select`.

1.11. Inserción de un elemento en una lista

Ejercicio 1.11 *Definir la relación* `inserta(?X,?L1,?L2)` *que se verifique si* `L2` *es una lista obtenida insertando* `X` *en* `L1`*. Por ejemplo,*

```
?- inserta(a,[1,2],L).
L = [a, 1, 2] ;
L = [1, a, 2] ;
L = [1, 2, a] ;
No
```

Solución: La definición de `inserta` es

```
inserta(X,L1,L2) :-
   select(X,L2,L1).
```

1.12. Sublista

Ejercicio 1.12 *Definir la relación* `sublista(?L1,?L2)` *que se verifique si* `L1` *es una sublista de* `L2`*. Por ejemplo,*

```
?- sublista([b,c],[a,b,c,d]).
Yes
?- sublista([a,c],[a,b,c,d]).
No
?- sublista([a,b],L).
L = [a, b|X] ;
L = [X, a, b|Y] ;
L = [X, Y, a, b|Z]
Yes
```

Solución: La definición de `sublista` es

```
sublista(L1,L2) :-
   append(_L3,L4,L2),
   append(L1,_L5,L4).
```

1.13. Permutación

Ejercicio 1.13 *Definir la relación* `permutación(+L1,?L2)` *que se verifique 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
```

Solución: La definición de `permutación(L1,L2)`, por recursión en `L1` es

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

Nota: La relación `permutación(L1,L2)` es equivalente a la definida `permutation(L2,L1)`.

1.14. Lista con todos sus elementos iguales

Ejercicio 1.14 *Definir la relación* `todos_iguales(+L)` *que se verifique si todos los elementos de la lista* `L` *son iguales entre sí. Por ejemplo,*

```
?- todos_iguales([a,a,a]).
Yes
?- todos_iguales([a,b,a]).
No
?- todos_iguales([]).
Yes
```

Solución: La definición de `todos_iguales` es

```
todos_iguales([]).
todos_iguales([_]).
todos_iguales([X,X|L]) :-
   todos_iguales([X|L]).
```

1.15. Paridad de la longitud de una lista

Ejercicio 1.15 *Definir la relación* `longitud_par(+L)` *que se verifique si la longitud de la lista* `L` *es par. Por ejemplo,*

```
?- longitud_par([a,b]).
Yes
?- longitud_par([a,b,c]).
No
```

Solución: La definición de `longitud_par`, por recursión cruzada con la relación `longitud_impar`, es

```
longitud_par([]).
longitud_par([_|L]) :-
   longitud_impar(L).
```

La relación `longitud_impar(+L)` se verifica si la longitud de la lista `L` es impar. Por ejemplo,

```
?- longitud_impar([a,b]).
No
?- longitud_impar([a,b,c]).
Yes
```

La definición de `longitud_impar` es

```
longitud_impar([_]).
longitud_impar([_|L]) :-
   longitud_par(L).
```

1.16. Rotación de un elemento

Ejercicio 1.16 *Definir la relación* `rota(?L1,?L2)` *que se verifique si* `L2` *es la lista obtenida a partir de* `L1` *colocando su primer elemento al final. Por ejemplo,*

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

Solución: La definición de `rota` es

```
rota([X|L1],L) :-
   append(L1,[X],L).
```

1.17. Subconjunto

Ejercicio 1.17 *Definir la relación* `subconjunto(+L1,?L2)` *que se verifique si* `L2` *es un subconjunto de* `L1`*. Por ejemplo,*

```
?- subconjunto([a,b,c,d],[b,d]).
Yes
?- subconjunto([a,b,c,d],[b,f]).
No
?- subconjunto([a,b,c],L).
L = [a, b, c] ;
L = [a, b] ;
L = [a, c] ;
L = [a] ;
L = [b, c] ;
L = [b] ;
L = [c] ;
L = [] ;
No
```

Solución: La definición de `subconjunto(L1,L2)`, por recursión en `L1`, es

```
subconjunto([],[]).
subconjunto([X|L1],[X|L2]) :-
   subconjunto(L1,L2).
subconjunto([_|L1],L2) :-
   subconjunto(L1,L2).
```

Capítulo 2

Aritmética

2.1. Máximo de dos números

Ejercicio 2.1 *Definir la relación* `máximo(+X,+Y,?Z)` *que se verifique si* `Z` *es el máximo de* `X` *e* `Y`*. Por ejemplo,*

```
?- máximo(2,3,X).
X = 3
?- máximo(3,2,X).
X = 3
```

Solución: La definición de `máximo` es

```
máximo(X,Y,X) :-
   X >= Y.
máximo(X,Y,Y) :-
   X < Y.
```

Nota: En Prolog está definida la función `max(X,Y)` que devuelve el máximo de `X` e `Y`. Por ejemplo,

```
?- X is max(5,10).
X = 10
```

2.2. Factorial

Ejercicio 2.2 *Definir la relación* `factorial(+X,?Y)` *que se verifique si* `Y` *es el factorial de* `X`*. Por ejemplo,*

```
?- factorial(3,X).
X = 6
```

Solución: La definición de `factorial(X,Y)`, por recursión sobre `X`, es

```
factorial(1,1).
factorial(X,Y) :-
   X > 1,
   X1 is X-1,
   factorial(X1,Y1),
   Y is X * Y1.
```

2.3. Sucesión de Fibonacci

Ejercicio 2.3 *La sucesión de Fibonacci es 0,1,1,2,3,5,8,13,21,... en la que cada término, salvo los dos primeros, es la suma de los dos anteriores. Definir la relación* `fibonacci(+N,-X)` *que se verifique si* `X` *es el* `N`*–ésimo término de la sucesión de Fibonacci. Por ejemplo,*

```
?- fibonacci(6,X).
X = 8
```

Solución: La definición de `fibonacci(N,X)`, por inducción en `N`, es

```
fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,X) :-
   N > 1,
   N1 is N-1,
   fibonacci(N1,X1),
   N2 is N-2,
   fibonacci(N2,X2),
   X is X1+X2.
```

2.4. Máximo común divisor

Ejercicio 2.4 *Definir la relación* `mcd(+X,+Y,?Z)` *que se verifique si* `Z` *es el máximo común divisor de* `X` *e* `Y`*. Por ejemplo,*

```
?- mcd(10,15,X).
X = 5
```

Solución: La definición de `mcd` es

```
mcd(X,X,X).
mcd(X,Y,Z) :-
   X < Y,
   Y1 is Y-X,
   mcd(X,Y1,Z).
mcd(X,Y,Z) :-
   X > Y,
   mcd(Y,X,Z).
```

2.5. Longitud de una lista

Ejercicio 2.5 *Definir la relación* `longitud(?L,?N)` *que se verifique si* `N` *es la longitud de la lista* `L`*. Por ejemplo*

```
?- longitud([a,b,c],N).
N = 3
?- longitud(L,3).
L = [X, Y, Z]
```

Solución: La definición de `longitud` es

```
longitud([],0).
longitud([_|L],N) :-
   longitud(L,N1),
   N is N1 + 1.
```

Nota: La relación `longitud` se corresponde con la relación definida `length`.

2.6. Lista de números acotada por su longitud

Ejercicio 2.6 *Una lista está acotada si todos sus elementos son menores que su longitud. Definir la relación* `lista_acotada(+L)` *que se verifique si todos los elementos de la lista de números* `L` *son menores que la longitud de* `L`*. Por ejemplo,*

```
?- lista_acotada([1,0,2]).
Yes
?- lista_acotada([1,3,2]).
No
```

Solución: La definición de `lista_acotada` es

```
lista_acotada(L) :-
   length(L,N),
   lista_acotada_aux(L,N).
```

donde `lista_acotada_aux(+L,+N)` se verifica si todos los elementos de la lista de números `L` son menores que `N`. Por ejemplo,

```
?- lista_acotada_aux([1,5,3],7).
Yes
?- lista_acotada_aux([1,5,3],5).
No
```

y está definida por

```
lista_acotada_aux([],_).
lista_acotada_aux([X|L],N) :-
   X < N,
   lista_acotada_aux(L,N).
```

2.7. Máximo de una lista de números

Ejercicio 2.7 *Definir la relación* `max_lista(+L,?X)` *que se verifique si* `X` *es el máximo de la lista de números* `L`*. Por ejemplo,*

```
?- max_lista([1,3,9,5],X).
X = 9
```

Solución: La definición de `max_lista` es

```
max_lista([X],X).
max_lista([X1,X2|L],Y) :-
   X3 is max(X1,X2),
   max_lista([X3|L],Y).
```

2.8. Suma de los elementos de una lista

Ejercicio 2.8 *Definir la relación* `suma_lista(+L,?X)` *que se verifique si* `X` *es la suma de los elementos de la lista de números* `L`*. Por ejemplo,*

```
? - suma_lista([1,3,5],X).
X = 9
```

Solución: La definición de `suma_lista` es

```
suma_lista([],0).
suma_lista([X|L],Y) :-
   suma_lista(L,Y1),
   Y is X+Y1.
```

Nota: La relación `suma_lista` se corresponde con la relación definida `sumlist`.

2.9. Lista de números ordenada

Ejercicio 2.9 *Definir la relación* `ordenada(+L)` *que se verifique si la lista de números* `L` *está ordenada de manera creciente. Por ejemplo,*

```
?- ordenada([1,3,3,5]).
Yes
?- ordenada([1,3,5,2]).
No
```

Solución: La definición de `ordenada` es

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

2.10. Suma parcial de una lista

Ejercicio 2.10 *Definir la relación* `suma_parcial(+L1,+X,?L2)` *que se verifique si* `L2` *es un subconjunto de* `L1` *tal que la suma de sus elementos es* `X`*. Por ejemplo,*

```
?- suma_parcial([1,2,5,3,2],5,L).
L = [1, 2, 2] ;
L = [2, 3] ;
L = [5] ;
L = [3, 2] ;
No
```

Solución: Se presentan dos definiciones de `suma_parcial`.
Primera solución: Una definición, usando `subconjunto` (p. 19) y `suma_lista` (p. 24), es

```
suma_parcial_1(L1,X,L2) :-
   subconjunto(L1,L2),
   suma_lista(L2,X).
```

Segunda solución: Una definición recursiva de `suma_parcial` es

```
suma_parcial_2([],0,[]).
suma_parcial_2([X|L1],Y,[X|L2]) :-
   Y >= X,
   Z is Y-X,
   suma_parcial_2(L1,Z,L2).
suma_parcial_2([_|L1],Y,L2) :-
   suma_parcial_2(L1,Y,L2).
```

2.11. Lista de `N` veces el número `N`

Ejercicio 2.11 *Definir la relación* `lista(+N,-L)` *que se verifique si* `L` *es la lista de longitud* `N` *cuyos elementos son* `N`*. Por ejemplo,*

```
?- lista(3,L).
L = [3, 3, 3]
```

Solución: La definición de `lista` es

```
lista(N,L) :-
   lista_aux(N,N,L).
```

donde `lista_aux(+N,+M,-L)` se verifica si `L` es la lista de longitud `M` cuyos elementos son `N`. Por ejemplo,

```
?- lista_aux(5,4,L).
L = [5, 5, 5, 5]
```

y se define por

```
lista_aux(_,0,[]).
lista_aux(N,M,[N|L]) :-
   M > 0,
   M1 is M-1,
   lista_aux(N,M1,L).
```

2.12. Generación de lista de números

Ejercicio 2.12 *Definir la relación* `lista_de_números(+N,+M,-L)` *que se verifica si L es la lista de los números desde N hasta M, ambos inclusive. Por ejemplo,*

```
?- lista_de_números(3,5,L).
L = [3, 4, 5]
?- lista_de_números(3,2,L).
No
```

Solución: La definición de `lista_de_números` es

```
lista_de_números(N,N,[N]).
lista_de_números(N,M,[N|L]) :-
   N < M,
   N1 is N+1,
   lista_de_números(N1,M,L).
```

Nota: La relación `lista_de_números` se corresponde con la definida `numlist`.

2.13. Intervalo entero

Ejercicio 2.13 *Definir la relación* `entre(+N1,+N2,?X)` *que se verifique si* `X` *es un número entero tal que* $\texttt{N1} \leq \texttt{X} \leq \texttt{N2}$. *Por ejemplo,*

```
?- entre(2,5,X).
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
No
```

Solución: La definición de `entre` es

```
entre(N1,N2,N1) :-
   N1 =< N2.
entre(N1,N2,X) :-
```

```
   N1 < N2,
   N3 is N1+1,
   entre(N3,N2,X).
```

Nota: La relación `entre` se corresponde con la definida `between`.

2.14. K–ésimo elemento

Ejercicio 2.14 *Definir la relación* `elemento_en(+K,?L,?X)` *que se verifique si* `X` *es el* `K`*–ésimo elemento de la lista* `L` *(se empieza a numerar en 1). Por ejemplo,*

```
?- elemento_en(2,[a,b,c,d],X).
X = b
?- elemento_en(2,L,b).
L = [X, b| Y]
```

Solución: La definición de `elemento_en` es

```
elemento_en(1,[X|_],X).
elemento_en(K,[_|L],X) :-
   K > 1,
   K1 is K-1,
   elemento_en(K1,L,X).
```

Nota: La relación `elemento_en` se corresponde con la relación definida `nth1`.

2.15. Multiplicación de las ocurrencias de los elementos de una lista

Ejercicio 2.15 *Definir la relación* `multiplicada(+L1,+N,-L2)` *que se verifica si* `L2` *es la lista obtenida repitiendo* `N` *veces los elementos de la lista* `L1`*. Por ejemplo,*

```
?- multiplicada([a,b,c],3,L).
L = [a, a, a, b, b, b, c, c, c]
```

Solución: La definición de `multiplicada` es

```
multiplicada(L1,N,L2) :-
   multiplicada_aux(L1,N,N,L2).
```

donde `multiplicada_aux(+L1,+K,+N,-L2)` se verifica si `L2` es la lista obtenida repitiendo `K` veces el primer elemento de `L1` y `N` veces los restantes elementos. Por ejemplo,

```
?- multiplicada_aux([a,b,c],2,3,L).
L = [a, a, b, b, b, c, c, c]
```

Su definición es

```
multiplicada_aux([],_,_,[]).
multiplicada_aux([_|L1],0,N,L2) :-
   multiplicada_aux(L1,N,N,L2).
multiplicada_aux([X|L1],K,N,[X|L2]) :-
   K > 0,
   K1 is K-1,
   multiplicada_aux([X|L1],K1,N,L2).
```

Capítulo 3

Estructuras

3.1. Segmentos como objetos estructurados

Ejercicio 3.1 *Supongamos que representamos los puntos del plano mediante términos de la forma* `punto(X,Y)`
donde `X` *e* `Y` *son números, y los segmentos del plano mediante términos de la forma* `segmento(P1,P2)`
donde `P1` *y* `P2` *son los puntos extremos del segmento. Definir las relaciones* `vertical(?S)` *y* `horizontal(?S)`
que se verifiquen si el segmento `S` *es vertical (resp. horizontal). Por ejemplo,*

```
?- vertical(segmento(punto(1,2),punto(1,3))).
Yes
?- vertical(segmento(punto(1,2),punto(4,2))).
No
?- vertical(segmento(punto(1,2),punto(1,3))).
No
?- vertical(segmento(punto(1,2),punto(4,2))).
Yes
```

Usar el programa para responder a las siguientes cuestiones:

1. *¿Es vertical el segmento que une los puntos* `(1,1)` *y* `(1,2)`*?*
2. *¿Es vertical el segmento que une los puntos* `(1,1)` *y* `(2,2)`*?*
3. *¿Hay algún* `Y` *tal que el segmento que une los puntos* `(1,1)` *y* `(2,Y)` *sea vertical?*
4. *¿Hay algún* `X` *tal que el segmento que une los puntos* `(1,2)` *y* `(X,3)` *sea vertical?*
5. *¿Hay algún* `Y` *tal que el segmento que une los puntos* `(1,1)` *y* `(2,Y)` *sea horizontal?*
6. *¿Para qué puntos el segmento que comienza en* `(2,3)` *es vertical?*
7. *¿Hay algún segmento que sea horizontal y vertical?*

Solución: Las definiciones de `vertical` y `horizontal` son

```
vertical(seg(punto(X,_Y),punto(X,_Y1))).
horizontal(seg(punto(_X,Y),punto(_X1,Y))).
```

Las respuestas a las preguntas son

1. ¿Es vertical el segmento que une los puntos `(1,1)` y `(1,2)`?

```
?- vertical(seg(punto(1,1),punto(1,2))).
Yes
```

2. ¿Es vertical el segmento que une los puntos `(1,1)` y `(2,2)`?

```
?- vertical(seg(punto(1,1),punto(2,2))).
No
```

3. ¿Hay algún `Y` tal que el segmento que une los puntos `(1,1)` y `(2,Y)` sea vertical?

```
?- vertical(seg(punto(1,1),punto(2,Y))).
No
```

4. ¿Hay algún `X` tal que el segmento que une los puntos `(1,2)` y `(X,3)` sea vertical?

```
?- vertical(seg(punto(1,2),punto(X,3))).
X = 1 ;
No
```

5. ¿Hay algún `Y` tal que el segmento que une los puntos `(1,1)` y `(2,Y)` sea horizontal?

```
?- horizontal(seg(punto(1,1),punto(2,Y))).
Y = 1 ;
No
```

6. ¿Para qué puntos el segmento que comienza en `(2,3)` es vertical?

```
?- vertical(seg(punto(2,3),P)).
P = punto(2, _G459) ;
No
```

7. ¿Hay algún segmento que sea horizontal y vertical?

```
?- vertical(S),horizontal(S).
S = seg(punto(_G444, _G445), punto(_G444, _G445)) ;
No
?- vertical(_),horizontal(_).
Yes
```

3.2. Base de datos familiar

Ejercicio 3.2 *En este ejercicio vamos a trabajar con una base de datos familiar.*

1. *Representar la información relativa a las siguientes familias:*
   - *En la primera familia,*
     - *el padre es Tomás García Pérez, nacido el 7 de Mayo de 1960, trabaja de profesor y gana 60 euros diarios;*
     - *la madre es Ana López Ruiz, nacida el 10 de marzo de 1962, trabaja de médica y gana 90 euros diarios;*
     - *el hijo es Juan García López, nacido el 5 de Enero de 1980, estudiante;*
     - *la hija es María García López, nacida el 12 de Abril de 1992, estudiante.*
   - *En la segunda familia,*
     - *el padre es José Pérez Ruiz, nacido el 6 de Marzo de 1963, trabaja de pintor y gana 120 euros diarios;*
     - *la madre es Luisa Gálvez Pérez, nacida el 12 de Mayo de 1964, trabaja de médica y gana 90 euros diarios;*
     - *un hijo es Juan Luis Pérez Pérez, nacido el 5 de Febrero de 1990, estudiante;*
     - *una hija es María José Pérez Pérez, nacida el 12 de Junio de 1992, estudiante;*
     - *otro hijo es José María Pérez Pérez, nacido el 12 de Julio de 1994, estudiante.*
2. *Realizar las siguientes consultas:*
   - *¿existe familia sin hijos?*
   - *¿existe familia con un hijo?*
   - *¿existe familia con dos hijos?*
   - *¿existe familia con tres hijos?*
   - *¿existe familia con cuatro hijos.?*
3. *Buscar los nombres de los padres de familia con tres hijos.*
4. *Definir la relación* `casado(X)` *que se verifique si* `X` *es un hombre casado.*
5. *Preguntar por los hombres casados.*
6. *Definir la relación* `casada(X)` *que se verifique si* `X` *es una mujer casada.*
7. *Preguntar por las mujeres casadas.*
8. *Determinar el nombre de todas las mujeres casadas que trabajan.*
9. *Definir la relación* `hijo(X)` *que se verifique si* `X` *figura en alguna lista de hijos.*
10. *Preguntar por los hijos.*
11. *Definir la relación* `persona(X)` *que se verifique si* `X` *es una persona existente en la base de datos.*

12. *Preguntar por los nombres y apellidos de todas las personas existentes en la base de datos.*
13. *Determinar todos los estudiantes nacidos antes de 1993.*
14. *Definir la relación* `fecha_de_nacimiento(X,Y)` *de forma que si* `X` *es una persona, entonces* `Y` *es su fecha de nacimiento.*
15. *Buscar todos los hijos nacidos en 1992.*
16. *Definir la relación* `sueldo(X,Y)` *que se verifique si el sueldo de la persona* `X` *es* `Y`*.*
17. *Buscar todas las personas nacidas antes de 1964 cuyo sueldo sea superior a 72 euros diarios.*
18. *Definir la relación* `total(L,Y)` *de forma que si* `L` *es una lista de personas, entonces* `Y` *es la suma de los sueldos de las personas de la lista* `L`*.*
19. *Calcular los ingresos totales de cada familia.*

Solución: Solución del apartado 1: La representación de la información sobre las dos familias es

```
familia(persona([tomas,garcia,perez],
                fecha(7,mayo,1960),
                trabajo(profesor,60)),
        persona([ana,lopez,ruiz],
                fecha(10,marzo,1962),
                trabajo(medica,90)),
        [ persona([juan,garcia,lopez],
                  fecha(5,enero,1990),
                  estudiante),
          persona([maria,garcia,lopez],
                  fecha(12,abril,1992),
                  estudiante) ]).

familia(persona([jose,perez,ruiz],
                fecha(6,marzo,1963),
                trabajo(pintor,120)),
        persona([luisa,galvez,perez],
                fecha(12,mayo,1964),
                trabajo(medica,90)),
        [ persona([juan_luis,perez,perez],
                  fecha(5,febrero,1990),
                  estudiante),
          persona([maria_jose,perez,perez],
                  fecha(12,junio,1992),
                  estudiante),
          persona([jose_maria,perez,perez],
                  fecha(12,julio,1994),
                  estudiante) ]).
```

Solución del apartado 2: Las consultas, y sus respuestas son,

```
?- familia(_,_,[]).
No
?- familia(_,_,[_]).
No
?- familia(_,_,[_,_]).
Yes
?- familia(_,_,[_,_,_]).
Yes
?- familia(_,_,[_,_,_,_]).
No
```

Solución del apartado 3:

```
?- familia(persona(NP,_,_),_,[_,_,_]).
NP = [jose, perez, ruiz] ;
No
```

Solución del apartado 4:

```
casado(X) :-
   familia(X,_,_).
```

Solución del apartado 5:

```
?- casado(X).
X = persona([tomas, garcia, perez],
            fecha(7, mayo, 1960),
            trabajo(profesor, 60)) ;
X = persona([jose, perez, ruiz],
            fecha(6, marzo, 1963),
            trabajo(pintor, 120)) ;
No
```

Solución del apartado 6:

```
casada(X) :-
   familia(_,X,_).
```

Solución del apartado 7:

```
?- casada(X).
X = persona([ana, lopez, ruiz],
            fecha(10, marzo, 1962),
            trabajo(medica, 90)) ;
X = persona([luisa, galvez, perez],
            fecha(12, mayo, 1964),
            trabajo(medica, 90)) ;
No
```

Solución del apartado 8:

```
?- casada(persona([N,_,_],_,trabajo(_,_))).
N = ana ;
N = luisa ;
No
```

Solución del apartado 9:

```
hijo(X) :-
   familia(_,_,L),
   member(X,L).
```

Solución del apartado 10:

```
?- hijo(X).
X = persona([juan,garcia,lopez],fecha(5,enero,1990),estudiante) ;
X = persona([maria,garcia,lopez],fecha(12,abril,1992),estudiante) ;
X = persona([juan_luis,perez,perez],fecha(5,febrero,1990),estudiante) ;
X = persona([maria_jose,perez,perez],fecha(12,junio,1992),estudiante) ;
X = persona([jose_maria,perez,perez],fecha(12,julio,1994),estudiante) ;
No
```

Solución del apartado 11:

```
persona(X) :-
   casado(X);
   casada(X);
   hijo(X).
```

Solución del apartado 12:

```
?- persona(persona(X,_,_)).
X = [tomas, garcia, perez] ;
X = [jose, perez, ruiz] ;
X = [ana, lopez, ruiz] ;
X = [luisa, galvez, perez] ;
X = [juan, garcia, lopez] ;
X = [maria, garcia, lopez] ;
X = [juan_luis, perez, perez] ;
X = [maria_jose, perez, perez] ;
X = [jose_maria, perez, perez] ;
No
```

Solución del apartado 13:

```
?- persona(persona(X,fecha(_,_,Año),estudiante)), Año < 1993.
X = [juan, garcia, lopez]
Año = 1990 ;
X = [maria, garcia, lopez]
Año = 1992 ;
X = [juan_luis, perez, perez]
Año = 1990 ;
X = [maria_jose, perez, perez]
Año = 1992 ;
No
```

Solución del apartado 14:

```
fecha_de_nacimiento(persona(_,Y,_),Y).
```

Solución del apartado 15:

```
?- hijo(X),fecha_de_nacimiento(X,fecha(_,_,1992)).
X = persona([maria_jose,perez,perez],fecha(12,junio,1992),estudiante) ;
No
```

Solución del apartado 16:

```
sueldo(persona(_,_,trabajo(_,Y)),Y).
sueldo(persona(_,_,estudiante),0).
```

Solución del apartado 17:

```
?- persona(X),
   fecha_de_nacimiento(X,fecha(_,_,Año)),
   Año < 1964,
   sueldo(X,Y),
   Y > 72.
X = persona([jose, perez, ruiz],
            fecha(6, marzo, 1963),
            trabajo(pintor, 120))
Año = 1963
Y = 120 ;
X = persona([ana, lopez, ruiz],
            fecha(10, marzo, 1962),
            trabajo(medica, 90))
Año = 1962
Y = 90 ;
No
```

Solución del apartado 18:

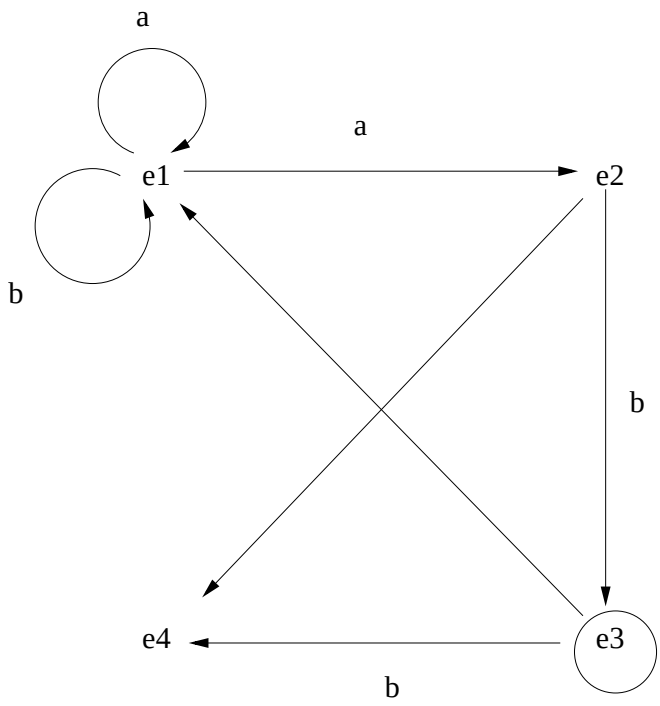
```
total([],0).
total([X|L],Y) :-
   sueldo(X,Y1),
   total(L,Y2),
   Y is Y1 + Y2.
```

Solución del apartado 19:

```
?- familia(X,Y,Z),total([X,Y|Z],Total).
X = persona([tomas,garcia,perez],
            fecha(7,mayo,1960),
            trabajo(profesor,60))
Y = persona([ana,lopez,ruiz],
            fecha(10,marzo,1962),
            trabajo(medica,90))
Z = [persona([juan,garcia,lopez],fecha(5,enero,1990),estudiante),
     persona([maria,garcia,lopez],fecha(12,abril,1992),estudiante)]
Total = 150 ;
X = persona([jose,perez,ruiz],
            fecha(6,marzo,1963),
            trabajo(pintor,120))
Y = persona([luisa,galvez,perez],
            fecha(12,mayo,1964),
            trabajo(medica,90))
Z = [persona([juan_luis,perez,perez],fecha(5,febrero,1990),estudiante),
     persona([maria_jose,perez,perez],fecha(12,junio,1982),estudiante)
     persona([jose_maria,perez,perez],fecha(12,julio,1984),estudiante)]
Total = 210 ;
No
```

3.3. Autómata no–determinista

Ejercicio 3.3 *Consideremos el autómata representado por*



siendo `e3` *el estado final.*

1. *Representar el autómata utilizando las siguientes relaciones*
   - `final(X)` *que se verifica si* `X` *es el estado final.*
   - `trans(E1,X,E2)` *que se verifica si se puede pasar del estado* `E1` *al estado* `E2` *usando la letra* `X`.
   - `nulo(E1,E2)` *que se verifica si se puede pasar del estado* `E1` *al estado* `E2` *mediante un movimiento nulo.*
2. *Definir la relación* `acepta(E,L)` *que se verifique si el autómata, a partir del estado* `E`, *acepta la lista* `L`. *Por ejemplo,*

```
?- acepta(e1,[a,a,a,b]).
Yes
?- acepta(e2,[a,a,a,b]).
No
```

3. *Determinar si el autómata acepta la lista* `[a,a,a,b]`.
4. *Determinar los estados a partir de los cuales el autómata acepta la lista* `[a,b]`.
5. *Determinar las palabras de longitud 3 aceptadas por el autómata a partir del estado* `e1`.
6. *Definir la relación* `acepta_acotada_1(E,L,N)` *que se verifique si el autómata, a partir del estado* `E`, *acepta la lista* `L` *y la longitud de* `L` *es* `N`.
7. *Buscar las cadenas aceptadas a partir de* `e1` *con longitud 3.*
8. *Definir la relación* `acepta_acotada_2(E,L,N)` *que se verifique si el autómata, a partir del estado* `E`, *acepta la lista* `L` *y la longitud de* `L` *es menor o igual que* `N`.
9. *Buscar las cadenas aceptadas a partir de* `e1` *con longitud menor o igual 3.*

Solución: Solución del apartado 1:

```
final(e3).

trans(e1,a,e1).
trans(e1,a,e2).
trans(e1,b,e1).
trans(e2,b,e3).
trans(e3,b,e4).

nulo(e2,e4).
nulo(e3,e1).
```

Solución del apartado 2:

```
acepta(E,[]) :-
   final(E).
acepta(E,[X|L]) :-
   trans(E,X,E1),
   acepta(E1,L).
acepta(E,L) :-
   nulo(E,E1),
   acepta(E1,L).
```

Solución del apartado 3:

```
?- acepta(e1,[a,a,a,b]).
Yes
```

Solución del apartado 4:

```
?- acepta(E,[a,b]).
E=e1 ;
E=e3 ;
No
```

Solución del apartado 5:

```
?- acepta(e1,[X,Y,Z]).
X = a
Y = a
Z = b ;
X = b
Y = a
Z = b ;
No
```

Solución del apartado 6: Presentamos dos definiciones. La primera usando `acepta`

```
acepta_acotada_1a(E,L,N) :-
   length(L,N),
   acepta(E,L).
```

La segunda definición es una variación de la definición de acepta:

```
acepta_acotada_1b(E,[],0) :-
   final(E).
acepta_acotada_1b(E,[X|L],N) :-
   N > 0,
   trans(E,X,E1),
   M is N - 1,
   acepta_acotada_1b(E1,L,M).
acepta_acotada_1b(E,L,N) :-
   nulo(E,E1),
   acepta_acotada_1b(E1,L,N).
```

Nota: La primera definición es más simple y eficiente que la segunda como se observa en el siguiente ejemplo

```
?- time(acepta_acotada_1a(e2,_L,5000)).
% 10,026 inferences, 0.01 CPU in 0.01 seconds (126% CPU, 1002600 Lips)
?- time(acepta_acotada_1b(e2,_L,5000)).
% 20,035 inferences, 0.02 CPU in 0.02 seconds (126% CPU, 1001750 Lips)
```

A partir de ahora, adoptaremos la definición `acepta_acotada_1a`

```
acepta_acotada_1(E,L,M) :-
   acepta_acotada_1a(E,L,M).
```

Solución del apartado 7:

```
?- acepta_acotada_1(e1,L,3).
L = [a, a, b] ;
L = [b, a, b] ;
No
```

Solución del apartado 8: Presentamos dos definiciones. La primera usando `acepta`

```
acepta_acotada_2a(E,L,N) :-
   between(0,N,M),
   length(L,M),
   acepta(E,L).
```

y la segunda modificando `acepta`

```
acepta_acotada_2b(E,[],_N) :-
   final(E).
acepta_acotada_2b(E,[X|L],N) :-
   N > 0,
   trans(E,X,E1),
   M is N-1,
   acepta_acotada_2b(E1,L,M).
acepta_acotada_2b(E,L,N) :-
   N > 0,
   nulo(E,E1),
   acepta_acotada_2b(E1,L,N).
```

Nota: La primera definición es más simple y eficiente que la segunda como se observa en el siguiente ejemplo

```
   ?- time(acepta_acotada_2a(e1,_L,10000)).
   % 47 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
   ?- time(acepta_acotada_2b(e1,_L,10000)).
   % 50,027 inferences, 0.07 CPU in 0.06 seconds (113% CPU, 714671 Lips)
```

A partir de ahora, adoptaremos la definición `acepta_acotada_2a`

```
acepta_acotada_2(E,L,M) :-
   acepta_acotada_2a(E,L,M).
```

Solución del apartado 9:

```
  ?- acepta_acotada_2(e1,L,3).
  L = [a, a, b] ;
  L = [a, b] ;
  L = [b, a, b] ;
  No
```

3.4. El problema del mono y el plátano

Ejercicio 3.4 *Un mono se encuentra en la puerta de una habitación. En el centro de la habitación hay un plátano colgado del techo. El mono está hambriento y desea coger el plátano, pero no lo alcanza desde el suelo. En la ventana de la habitación hay una silla que el mono puede usar. El mono puede realizar las siguientes acciones: pasear de un lugar a otro de la habitación, empujar la silla de un lugar a otro de la habitación (si está en el mismo lugar que la silla), subirse en la silla (si está en el mismo lugar que la silla) y coger el plátano (si está encima de la silla en el centro de la habitación).*

Definir la relación `solución(E,S)` *que se verifique si* `S` *es una sucesión de acciones que aplicadas al estado* `E` *permiten al mono coger el plátano. Por ejemplo,*

```
  ?- solución(estado(puerta,suelo,ventana,sin),L).
  L = [pasear(puerta,ventana),empujar(ventana,centro),subir,coger]
```

donde `estado(PM,EM,PS,X)` *significa que el mono se encuentra en la posición* `PM` *(puerta, centro o ventana) encima de* `EM` *(suelo o silla), la silla se encuentra en la posición* `PS` *(puerta, centro o ventana) y el mono tiene (*`X = con`*) o no (*`X = sin`*) el plátano.*

Solución:

```
solución(estado(_,_,_,con),[]).
solución(E1,[A|L]) :-
   movimiento(E1,A,E2),
   solución(E2,L).
```

La relación `movimiento(estado(PM1,EM1,PS1,X1),A,estado(PM2,EM2,PS2,X2))` se verifica si en el `estado(PM1,EM1,PS1,X1)` se puede aplicar la acción `A` y como resultado de su aplicación se pasa al `estado(PM2,EM2,PS2,X2)`.

```
movimiento(estado(centro,silla,centro,sin),
           coger,
           estado(centro,silla,centro,con)).
movimiento(estado(X,suelo,X,U),
           subir,
           estado(X,silla,X,U)).
movimiento(estado(X1,suelo,X1,U),
           empujar(X1,X2),
           estado(X2,suelo,X2,U)).
movimiento(estado(X,suelo,Z,U),
           pasear(X,Z),
           estado(Z,suelo,Z,U)).
```

3.5. Movimientos del caballo del ajedrez

Ejercicio 3.5 *Supongamos que los cuadros del tablero de ajedrez los representamos por pares de números* `[X,Y]` *con* `X` *e* `Y` *entre* `1` *y* `8`.

1. *Definir la relación* `salta(+C1,?C2)` *que se verifica si el caballo puede pasar en un movimiento del cuadrado* `C1` *al cuadrado* `C2`. *Por ejemplo,*

   ```
   ?- salta([1,1],S).
   S=[3,2];
   S=[2,3];
   No
   ```

2. *Definir la relación* `camino(L)` *que se verifique si* `L` *es una lista de cuadrados representando el camino recorrido por un caballo sobre un tablero vacío. Por ejemplo,*

   ```
   ?- camino([[1,1],C]).
   C=[3,2];
   C=[2,3];
   No
   ```

3. *Usando la relación* `camino`*, escribir una pregunta para determinar los caminos de longitud 4 por los que puede desplazarse un caballo desde cuadro* `[2,1]` *hasta el otro extremo del tablero (*`Y=8`*) de forma que en el segundo movimiento pase por el cuadro* `[5,4]`.

4. *Calcular el menor número de movimientos necesarios para desplazar el caballo del cuadro* `[1,1]` *al* `[2,2]`. *¿Cuántos caminos de dicha longitud hay de [1,1] a [2,2]?*

Solución: Solución del apartado 1:

```
salta([X,Y],[X1,Y1]) :-
   dxy(Dx,Dy),
   X1 is X+Dx,
   correcto(X1),
   Y1 is Y+Dy,
   correcto(Y1).
```

La relación `dxy(?X,?Y)` se verifica si un caballo puede moverse `X` espacios horizontales e `Y` verticales.

```
dxy(2,1).
dxy(2,-1).
dxy(-2,1).
dxy(-2,-1).
dxy(1,2).
dxy(1,-2).
dxy(-1,2).
dxy(-1,-2).
```

La relación `correcto(+X)` se verifica si `X` está entre 1 y 8.

```
correcto(X) :-
   1 =< X,
   X =< 8.
```

Solución del apartado 2:

```
camino([_]).
camino([C1,C2|L]) :-
   salta(C1,C2),
   camino([C2|L]).
```

Solución del apartado 3:

```
  ?- camino([[2,1],C1,[5,4],C2,[X,8]]).
  C1 = [4, 2]   C2 = [6, 6]   X = 7 ;
  C1 = [4, 2]   C2 = [6, 6]   X = 5 ;
  C1 = [4, 2]   C2 = [4, 6]   X = 5 ;
```

```
C1 = [4, 2]   C2 = [4, 6]   X = 3 ;
C1 = [3, 3]   C2 = [6, 6]   X = 7 ;
C1 = [3, 3]   C2 = [6, 6]   X = 5 ;
C1 = [3, 3]   C2 = [4, 6]   X = 5 ;
C1 = [3, 3]   C2 = [4, 6]   X = 3 ;
No
```

Solución del apartado 4:

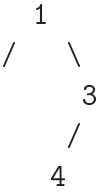
```
?- camino([[1,1],_,[2,2]]).
No
?- camino([[1,1],_,_,[2,2]]).
No
?- camino([[1,1],_,_,_,[2,2]]).
Yes
?- camino([[1,1],C2,C3,C4,[2,2]]).
C2 = [3, 2]   C3 = [5, 3]   C4 = [3, 4] ;
C2 = [3, 2]   C3 = [5, 3]   C4 = [4, 1] ;
C2 = [3, 2]   C3 = [5, 1]   C4 = [4, 3] ;
C2 = [3, 2]   C3 = [1, 3]   C4 = [3, 4] ;
C2 = [3, 2]   C3 = [2, 4]   C4 = [4, 3] ;
C2 = [2, 3]   C3 = [4, 2]   C4 = [3, 4] ;
C2 = [2, 3]   C3 = [3, 5]   C4 = [1, 4] ;
C2 = [2, 3]   C3 = [3, 5]   C4 = [4, 3] ;
C2 = [2, 3]   C3 = [3, 1]   C4 = [4, 3] ;
C2 = [2, 3]   C3 = [1, 5]   C4 = [3, 4] ;
No
```

3.6. Máximo elemento de un árbol binario

Ejercicio 3.6 *Un árbol binario es vacío o consta de tres partes: la raíz (que debe de ser un número positivo), el subárbol izquierdo (que debe ser un árbol binario) y el subárbol derecho (que debe ser un árbol binario). Usaremos la siguiente representación*

- `nil` *representa el árbol vacío*
- `t(I,R,D)` *representa el árbol de la raíz* `R`*, subárbol izquierdo* `I` *y subárbol derecho* `D`*.*

Por ejemplo, `t(t(nil,2,nil),1,t(t(nil,4,nil),3,nil))` *representa el árbol*



Definir la relación `máximo(+T,-X)` *que se verifique si* `X` *es el máximo de los nodos del árbol* `T`*. Por ejemplo,*

```
?- máximo(nil,N).
N = 0
?- máximo(t(nil,2,nil),N).
N = 2
?- máximo(t(t(nil,2,nil),3,nil),N).
N = 3
```

Solución: La definición de `máximo` es

```
máximo(nil,0).
máximo(t(I,R,D),M):-
   máximo(I,MI),
   máximo(D,MD),
   M1 is max(MI,MD),
   M is max(R,M1).
```

Capítulo 4

Retroceso, corte y negación

4.1. Ejemplos de uso del corte

Ejercicio 4.1 *1. Definir la relación* `f(X,Y)` *de forma que:*

- *si* `X` $<$ `3`*, entonces* `Y` $=$ `0`*;*
- *si* `3` $\leq$ `X` $<$ `6`*, entonces* `Y` $=$ `2`*;*
- *si* `6` $\leq$ `X`*, entonces* `Y` $=$ `4`*.*

2. *Construir el árbol de deducción correspondiente a la cuestión*

```
?- f(1,Y), 2 < Y.
```

3. *Definir la relación* `f_1(X,Y)` *a partir de la definición de* `f(X,Y)`*, introduciendo un corte al final de las dos primeras cláusulas.*

4. *Construir el árbol de deducción correspondiente a la cuestión*

```
?- f_1(1,Y), 2 < Y.
```

5. *Construir el árbol de deducción correspondiente a la cuestión*

```
?- f_1(7,Y).
```

6. *En el árbol anterior se observa que se efectúan comparaciones innecesarias (por ejemplo, después de fallar la comparación* `7<3`*, efectúa la comparación* `3=<7`*). Definir la relación* `f_2(X,Y)` *suprimiendo en la definición de* `f_1(X,Y)` *las comparaciones innecesarias.*

7. *Construir el árbol de deducción correspondiente a la cuestión*

```
?- f_2(7,Y).
```

8. *Construir el árbol de deducción correspondiente a la cuestión*

```
?- f_2(1,Y), 2 < Y.
```

9. *Definir la relación* `f_3(X,Y)` *a partir de la definición de* `f_2(X,Y)`*, suprimiendo los cortes.*

10. Obtener las respuestas correspondientes a la cuestión

```
?- f_3(1,Y).
```

Solución: Solución del apartado 1: La definición de `f` es

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

Solución del apartado 2: El árbol de deducción se muestra en la figura 4.1 (página 46).

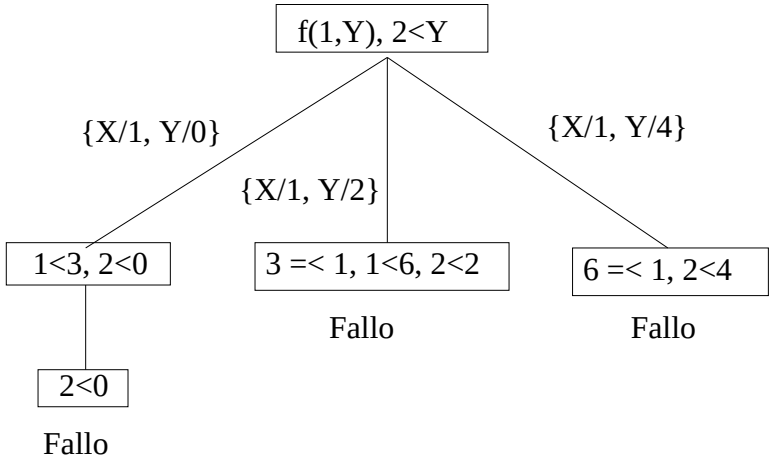


Figura 4.1: Árbol de resolución del apartado 2

Solución del apartado 3: La definición de `f_1` es

```
f_1(X,0) :- X < 3, !.
f_1(X,2) :- 3 =< X, X < 6, !.
f_1(X,4) :- 6 =< X.
```

Solución del apartado 4: El árbol de deducción se muestra en la figura 4.2 (página 47).
Solución del apartado 5: El árbol de deducción se muestra en la figura 4.3 (página 47).
Solución del apartado 6: La definición de `f_2` es

```
f_2(X,0) :- X < 3, !.
f_2(X,2) :- X < 6, !.
f_2(X,4).
```

Solución del apartado 7: El árbol de deducción se muestra en la figura 4.4 (página 47).
Solución del apartado 8: El árbol de deducción se muestra en la figura 4.5 (página 48).
Solución del apartado 9: La definición de `f_3` es

```
f_3(X,0) :- X < 3.
f_3(X,2) :- X < 6.
f_3(X,4).
```

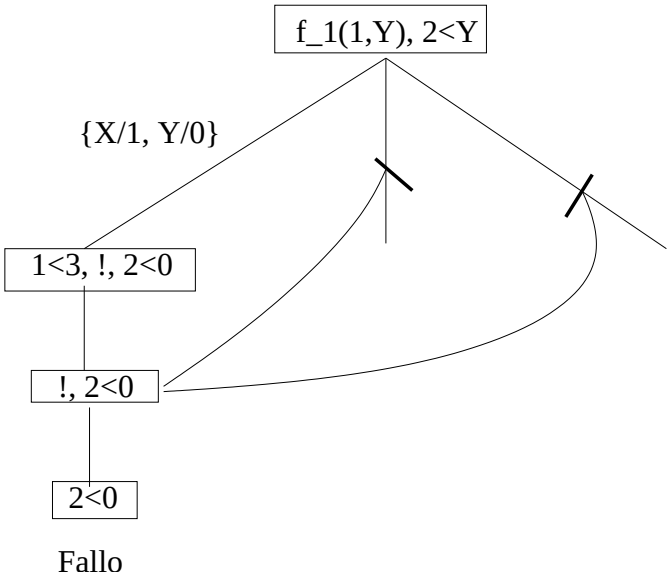


Figura 4.2: Árbol de resolución del apartado 4

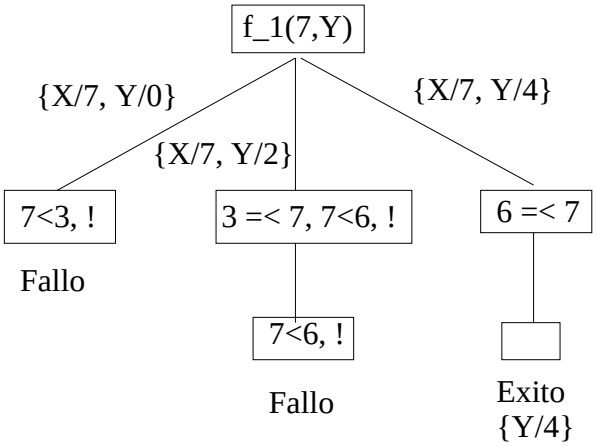


Figura 4.3: Árbol de resolución del apartado 5

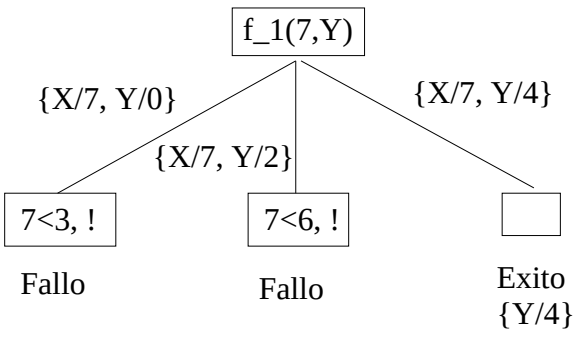


Figura 4.4: Árbol de resolución del apartado 7

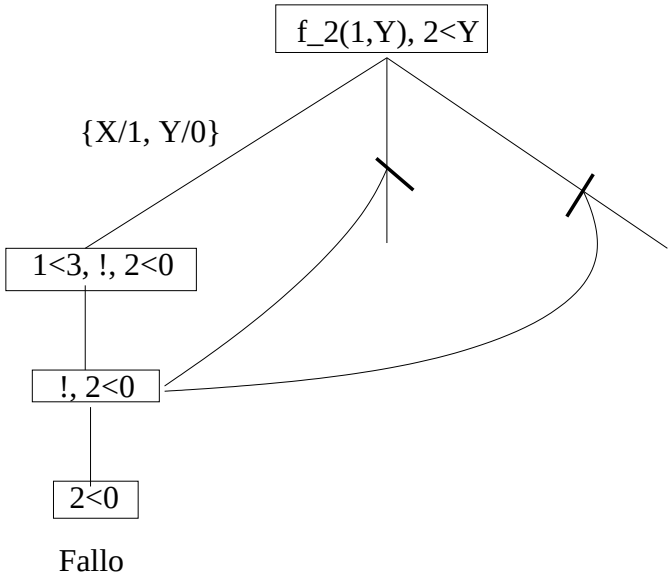


Figura 4.5: Árbol de resolución del apartado 8

Solución del apartado 10: Las respuestas son

```
?- f_3(1,Y).
Y = 0 ;
Y = 2 ;
Y = 4 ;
No
```

4.2. Árboles de deducción de `memberchk`

Ejercicio 4.2 *La relación* `memberchk` *está definida por*

```
memberchk(X,[X|_]) :- !.
memberchk(X,[_|L]) :- memberchk(X,L).
```

Escribir los árboles de SLD resolución correspondientes a las siguientes preguntas

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

Solución:

4.3. Diferencia de conjuntos

Ejercicio 4.3 *Definir la relación* `diferencia(+C1,+C2,-C3)` *que se verifique si* `C3` *es la diferencia de los conjuntos* `C1` *y* `C2`*. Por ejemplo,*

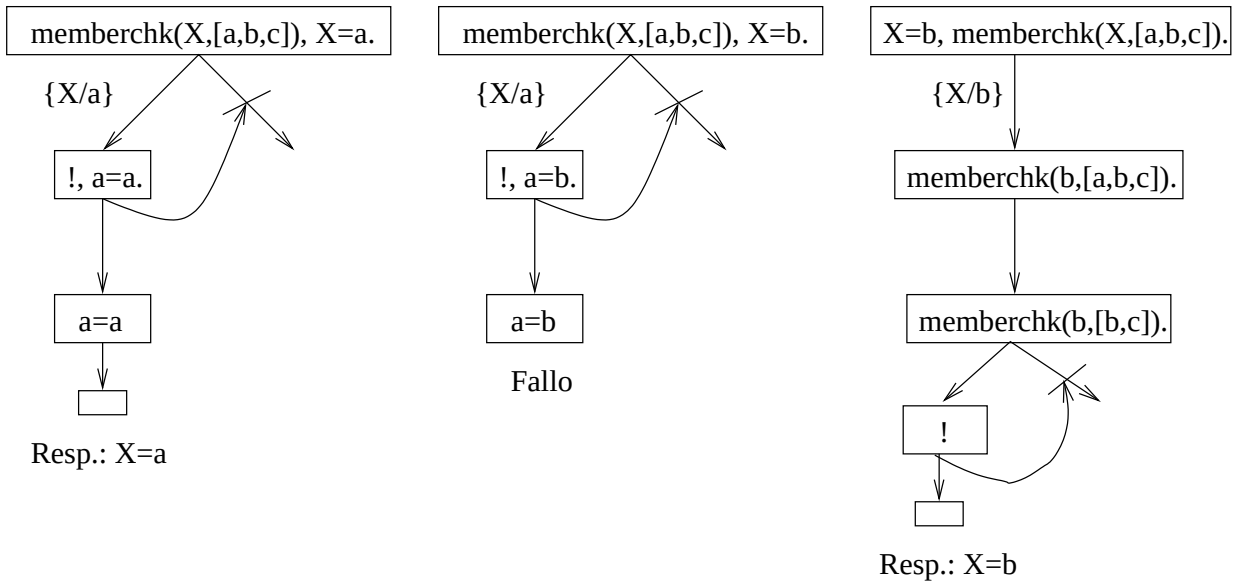


Figura 4.6: Árbol de resolución `memberchk`

```
?- diferencia([a,b],[b,c],X).
X = [a];
No
```

Definir una versión con negación (`diferencia_1`*) y otra con corte (*`diferencia_2`*) y comparar la eficiencia de las distintas definiciones con el ejemplo*

```
?- numlist(1,1000,_L1), time(diferencia_1(_L1,_L1)).
?- numlist(1,1000,_L1), time(diferencia_2(_L1,_L1)).
```

Nota: *La relación* `diferencia` *se corresponde con la relación definida* `subtract`.

Solución: 1ª definición (con negación):

```
diferencia_1([],_,[]).
diferencia_1([X|L],L2,L3):-
   member(X,L2),
   diferencia_1(L,L2,L3).
diferencia_1([X|L],L2,[X|L3]):-
   not(member(X,L2)),
   diferencia_1(L,L2,L3).
```

2ª definición (con corte):

```
diferencia_2([],_,[]).
diferencia_2([X|L],L2,L3):-
   member(X,L2), !,
   diferencia_2(L,L2,L3).
diferencia_2([X|L],L2,[X|L3]):-
   % not(member(X,L2)),
   diferencia_2(L,L2,L3).
```

3ª definición (con corte y memberchk):

```
diferencia_3([],_,[]).
diferencia_3([X|L],L2,L3):-
   memberchk(X,L2), !,
   diferencia_3(L,L2,L3).
diferencia_3([X|L],L2,[X|L3]):-
   % not(memberchk(X,L2)),
   diferencia_3(L,L2,L3).
```

Comparaciones:

```
?- numlist(1,1000,_L1), time(diferencia_1(_L1,_L1,[])).
% 501,501 inferences, 0,24 CPU in 0,25 seconds (97% CPU, 2089587 Lips)
?- numlist(1,1000,_L1), time(diferencia_2(_L1,_L1,[])).
% 501,501 inferences, 0,25 CPU in 0,29 seconds (87% CPU, 2006004 Lips)
?- numlist(1,1000,_L1), time(diferencia_3(_L1,_L1,[])).
% 2,001 inferences, 0,07 CPU in 0,07 seconds (96% CPU, 28586 Lips)
?- numlist(1,1000,_L1), time(subtract(_L1,_L1,[])).
% 2,001 inferences, 0,08 CPU in 0,08 seconds (105% CPU, 25012 Lips)
```

4.4. Agregación de un elemento a un conjunto

Ejercicio 4.4 *Definir la relación* `agregar(+X,+L,-L1)` *que se verifique si* `L1` *es la lista obtenida añadiéndole* `X` *a* `L`*, si* `X` *no pertenece a* `L` *y es* `L` *en caso contrario. Por ejemplo,*

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

Hacer una versión con negación y otra con corte.

Solución: 1ª definición (con negación):

```
agregar_1(X,L,L):-
   member(X,L).
agregar_1(X,L,[X|L]) :-
   not(member(X,L)).
```

2ª definición (con corte):

```
agregar_2(X,L,L):-
   member(X,L), !.
agregar_2(X,L,[X|L]).
```

4.5. Separación de una lista de números en positivos y negativos

Ejercicio 4.5 *Definir la relación* `separa(+L1,+L2,-L3)` *que separa la lista de números* `L1` *en dos listas:* `L2` *formada por los números positivos y* `L3` *formada por los números negativos o cero. Por ejemplo,*

```
?- separa([2,0,-3,5,0,2],L2,L3).
L2 = [2, 5, 2]
L3 = [0, -3, 0]
Yes
```

Proponer dos soluciones, una sin corte y otra con corte.

Solución: Definición con negación:

```
separa_1([],[],[]).
separa_1([N|RL1],[N|RL2],L3) :-
   N > 0,
   separa_1(RL1,RL2,L3).
separa_1([N|RL1],L2,[N|RL3]) :-
   N =< 0,
   separa_1(RL1,L2,RL3).
```

Definición con corte:

```
separa_2([],[],[]).
separa_2([N|RL1],[N|RL2],L3) :-
   N > 0, !,
   separa_2(RL1,RL2,L3).
separa_2([N|RL1],L2,[N|RL3]) :-
   % N =< 0,
   separa_2(RL1,L2,RL3).
```

4.6. Suma de los números pares de una lista de números

Ejercicio 4.6 *Definir la relación* `suma_pares(+L,-N)` *que se verifica si* `N` *es la suma de todos los números pares de la lista de números* `L`*. Por ejemplo,*

```
?- suma_pares([2,3,4],N).
N = 6
?- suma_pares([1,3,5,6,9,11,24],N).
N = 30
```

Hacer una versión con negación y otra con corte.

Solución:
Versión 1 (con negación):

```
suma_pares_1([],0).
suma_pares_1([N|L],X) :-
   par(N),
   suma_pares_1(L,X1),
   X is X1 + N.
suma_pares_1([_N|L],X) :-
   not(par(_N)),
   suma_pares_1(L,X).

par(N):-
   N mod 2 =:= 0.
```

Versión 2 (con corte):

```
suma_pares_2([],0).
suma_pares_2([N|L],X) :-
   par(N), !,
   suma_pares_2(L,X1),
   X is X1 + N.
suma_pares_2([_N|L],X) :-
   % not(par(_N)),
   suma_pares_2(L,X).
```

Versión 3 (con corte y acumulador):

```
suma_pares_3(L,X):-
   suma_pares_3_aux(L,0,X).

suma_pares_3_aux([],Ac,Ac).
suma_pares_3_aux([N|L],Ac,X) :-
   par(N), !,
   Ac1 is Ac + N,
   suma_pares_3_aux(L,Ac1,X).
suma_pares_3_aux([_N|L],Ac,X) :-
   % not(par(_N)),
   suma_pares_3_aux(L,Ac,X).
```

4.7. Exponente de dos en la factorización de un número

Ejercicio 4.7 *Definir la relación* `exponente_de_dos(+N,-E)` *que se verifica si* `E` *es el exponente de 2 en la descomposición de* `N` *como producto de factores primos. Por ejemplo,*

```
?- exponente_de_dos(40,E).
E=3
?- exponente_de_dos(49,E).
E=0
```

Hacer una versión con negación y otra con corte.

Solución: 1ª Versión (con negación):

```
exponente_de_dos_1(N,E):-
   N mod 2 =:= 0,
   N1 is N / 2,
   exponente_de_dos_1(N1,E1),
   E is E1 + 1.
exponente_de_dos_1(N,0) :-
   N mod 2 =\= 0.
```

2ª Versión (con corte):

```
exponente_de_dos_2(N,E):-
   N mod 2 =:= 0, !,
   N1 is N / 2,
   exponente_de_dos_2(N1,E1),
   E is E1 + 1.
exponente_de_dos_2(_,0).
```

4.8. Transformación de lista a conjunto

Ejercicio 4.8 *Definir la relación* `lista_a_conjunto(+L,-C)` *que se verifique si* `C` *es el conjunto correspondiente a la lista* `L` *(es decir,* `C` *contiene los mismos elementos que* `L` *en el mismo orden, pero si* `L` *tiene elementos repetidos sólo se incluye en* `C` *la última aparición de cada elemento). Por ejemplo,*

```
?- lista_a_conjunto([b,a,b,d],C).
C = [a, b, d]
```

Nota: *La relación* `lista_a_conjunto` *se corresponde con la relación definida* `list_to_set`*.*

Solución: La definición de `lista_a_conjunto` es

```
lista_a_conjunto([],[]).
lista_a_conjunto([X|L],C) :-
   member(X,L),
   lista_a_conjunto(L,C).
lista_a_conjunto([X|L],[X|C]) :-
   \+ member(X,L),
   lista_a_conjunto(L,C).
```

La definición anterior puede simplificarse con cortes

```
lista_a_conjunto_1([],[]).
lista_a_conjunto_1([X|L],C) :-
   member(X,L), !,
```

```
   lista_a_conjunto_1(L,C).
lista_a_conjunto_1([X|L],[X|C]) :-
   % \+ member(X,L),
   lista_a_conjunto_1(L,C).
```

3ª definición (con corte y memberchk):

```
lista_a_conjunto_3([],[]).
lista_a_conjunto_3([X|L],L2):-
   memberchk(X,L), !,
   lista_a_conjunto_3(L,L2).
lista_a_conjunto_3([X|L],[X|L2]):-
   % not(member(X,L)),
   lista_a_conjunto_3(L,L2).
```

Comparaciones:

```
  ?- numlist(1,1000,_L1), time(lista_a_conjunto_1(_L1,_l2)).
  % 1,003,001 inferences, 0,40 CPU in 0,41 seconds (97% CPU, 2507503 Lips)
  ?- numlist(1,1000,_L1), time(lista_a_conjunto_2(_L1,_l2)).
  % 501,501 inferences, 0,26 CPU in 0,28 seconds (93% CPU, 1928850 Lips)
  ?- numlist(1,1000,_L1), time(lista_a_conjunto_3(_L1,_l2)).
  % 3,001 inferences, 0,07 CPU in 0,08 seconds (90% CPU, 42871 Lips)
  ?- numlist(1,1000,_L1), time(list_to_set(_L1,_l2)).
  % 3,004 inferences, 0,07 CPU in 0,08 seconds (93% CPU, 42914 Lips)
```

4.9. Signos de crecimientos de sucesiones numéricas

Ejercicio 4.9 *Definir la relación* `crecimientos(+L1,-L2)` *que se verifique si* `L2` *es la lista correspondientes a los crecimientos de la lista numérica* `L1`*; es decir, entre cada par de elementos consecutivos* `X` *e* `Y` *de* `L1` *coloca el signo* `+` *si* `X < Y` *e y signo* `-` *en caso contrario. Por ejemplo,*

```
  ?- crecimientos([1,3,2,2,5,3],L).
  L = [1, +, 3, -, 2, -, 2, +, 5, -]
```

Dar una definición sin corte y otra con corte.

Solución: La definición de `crecimientos` sin usar corte es

```
crecimientos_1([_],[]).
crecimientos_1([X,Y|L1],[X,+|L2]) :-
   X < Y,
   crecimientos_1([Y|L1],L2).
crecimientos_1([X,Y|L1],[X,-|L2]) :-
   X >= Y,
   crecimientos_1([Y|L1],L2).
```

La definición de `crecimientos` usando corte es

```
crecimientos_2([_],[]).
crecimientos_2([X,Y|L1],[X,+|L2]) :-
   X < Y, !,
   crecimientos_2([Y|L1],L2).
crecimientos_2([X,Y|L1],[X,-|L2]) :-
   % X >= Y,
   crecimientos_2([Y|L1],L2).
```

4.10. Descomposición en factores primos

Ejercicio 4.10 *Definir las siguientes relaciones:*

- `menor_divisor_propio(+N,?X)` *que se verifique si* `X` *es el menor divisor de* `N` *mayor o igual que 2. Por ejemplo,*

```
?- menor_divisor_propio(30,X).
X = 2
?- menor_divisor_propio(3,X).
X = 3
```

- `factorización(+N,-L)` *que se verifique si* `L` *es la lista correspondiente a la descomposición del número* `N` *en factores primos (se considera los que elementos de* `L` *están ordenados de manera creciente). Por ejemplo,*

```
?- factorización(12,L).
L = [2, 2, 3] ;
No
?- factorización(1,L).
L = [] ;
No
```

Solución: La definición de `menor_divisor_propio` es

```
menor_divisor_propio(N,X) :-
   N1 is floor(sqrt(N)),
   between(2,N1,X),
   N mod X =:= 0, !.
menor_divisor_propio(N,N).
```

La definición de `factorización` es

```
factorización(1,[]).
factorización(N,[X|L]) :-
   N > 1,
   menor_divisor_propio(N,X),
   N1 is N/X,
   factorización(N1,L).
```

4.11. Menor elemento que cumple una propiedad

Ejercicio 4.11 *Definir la relación* `calcula(+N,+M,?X)` *que se verifique si* `X` *es el menor múltiplo de* `N` *tal que la suma de sus dígitos es mayor que* `M`*. Por ejemplo,*

```
?- calcula(3,10,X).
X = 39
Yes
?- calcula(7,20,X).
X = 399
Yes
```

Solución: La definición de `calcula` es

```
calcula(N,M,X) :-
   múltiplo(N,X),
   suma_digítos(X,N1),
   N1 > M, !.
```

La relación `múltiplo(+N,-X)` se verifica si `X` es un múltiplo de `N`. Por ejemplo,

```
?- múltiplo(5,X).
X = 5 ;
X = 10 ;
X = 15
Yes
```

```
múltiplo(N,N).
múltiplo(N,M) :-
   múltiplo(N,N1),
   M is N+N1.
```

La relación `suma_digítos(+N,-S)` se verifica si `S` es la suma de los dígitos del número `N`. Por ejemplo,

```
?- suma_digítos(237,S).
S = 12
```

```
suma_digítos(N,N) :-
   N < 10, !.
suma_digítos(N,S) :-
   % N >= 10,
   N1 is N // 10,
   R is N - 10*N1,
   suma_digítos(N1,S1),
   S is S1 + R.
```

4.12. Números libres de cuadrados

Ejercicio 4.12 *Un número es libre de cuadrados si no es divisible por el cuadrado de ningún número mayor que 1. Definir la relación* `libre_de_cuadrados(+N)` *que se verifique si el número* `N` *es libre de cuadrados. Por ejemplo,*

```
?- libre_de_cuadrados(30).
Yes
?- libre_de_cuadrados(12).
No
```

Solución: La definición de `libre_de_cuadrados` es

```
libre_de_cuadrados(N) :-
   M is floor(sqrt(N)),
   not((between(2,M,X), N mod (X*X) =:= 0)).
```

4.13. Suma de los números libres de cuadrados

Ejercicio 4.13 *Definir la relación* `suma_libres_de_cuadrados(+L,-S)` *que se verifique si* `S` *es la suma de los números libres de cuadrados la lista numérica* `L`*. Por ejemplo,*

```
?- suma_libres_de_cuadrados([6,12,18,30],S).
S = 36
```

Nota: Dar dos definiciones, una con negación y otra con corte.

Solución: La definición de `suma_libres_de_cuadrados` usando la negación es

```
suma_libres_de_cuadrados_1([],0).
suma_libres_de_cuadrados_1([X|L],S) :-
   libre_de_cuadrados(X),
   suma_libres_de_cuadrados_1(L,S1),
   S is X+S1.
suma_libres_de_cuadrados_1([X|L],S) :-
   not(libre_de_cuadrados(X)),
   suma_libres_de_cuadrados_1(L,S).
```

y la definición usando corte es

```
suma_libres_de_cuadrados_2([],0).
suma_libres_de_cuadrados_2([X|L],S) :-
   libre_de_cuadrados(X), !,
   suma_libres_de_cuadrados_2(L,S1),
   S is X+S1.
suma_libres_de_cuadrados_2([_X|L],S) :-
   % not(libre_de_cuadrados(_X)),
   suma_libres_de_cuadrados_2(L,S).
```

4.14. Máximo número de una lista

Ejercicio 4.14 *Definir la relación* `max_lista(+L,-N)` *que se verifique si* `N` *es el mayor número de la lista* `L`*. Por ejemplo,*

```
?- max_lista([2,a23,5,7+9],N).
N = 5
?- max_lista([-2,a23,-5,7+9],N).
N = -2
?- max_lista([a23,7+9],N).
No
```

Solución: La definición de `max_lista` es

```
max_lista(L,M) :-
   member(M,L),
   number(M),
   not((member(N,L),
        number(N),
        N>M)).
```

4.15. Longitud de las subsucesiones comunes maximales

Ejercicio 4.15 *Definir la relación* `longitud_scm(+L1,+L2,-N)` *que se verifique si* `N` *es la longitud de las subsucesiones comunes maximales de las listas* `L1` *y* `L2`*. Por ejemplo,*

```
?- longitud_scm([2,1,4,5,2,3,5,2,4,3],[1,7,5,3,2],N).
N = 4 ;
No
```

ya que `[1,5,3,2]` *es una subsucesión de las dos listas y no poseen ninguna otra subsucesión común de mayor longitud. Obsérvese que los elementos de la subsucesión no son necesariamente elementos adyacentes en las listas.*

Solución: La definición de `longitud_scm` es

```
longitud_scm([],_,0).
longitud_scm(_,[],0).
longitud_scm([X|L1],[X|L2],N) :-
   !, longitud_scm(L1,L2,M),
   N is M+1.
longitud_scm([X|L1],[Y|L2],N) :-
   % X \= Y,
   longitud_scm(L1,[Y|L2],N1),
   longitud_scm([X|L1],L2,N2),
   N is max(N1,N2).
```

4.16. Elementos repetidos en una lista

Ejercicio 4.16 *Definir la relación* `repetido(-A,+L)` *que se verifique si el elemento* `A` *está repetido (i.e. ocurre más de una vez) en la lista* `L`*. Por ejemplo,*

```
?- repetido(A,[1,2,1,3,4,3]).
A = 1 ;
A = 1 ;
A = 3 ;
A = 3 ;
No
?- repetido(A,[1,2,5]).
No
```

Solución: La definición de `repetido` es

```
repetido(A,L) :-
   select(A,L,R),
   memberchk(A,R).
```

Ejercicio 4.17 *Defimir la relación* `elimina(+X,+L1,-L2)` *que se verifique si* `L2` *es la lista obtenida eliminando todas las ocurrencias de* `X` *en la lista* `L1`*. Por ejemplo,*

```
?- elimina(a,[1,a,b,3,a,a,4,a,c],L).
L = [1, b, 3, 4, c]
```

Solución: La definición de `elimina` es

```
elimina(_,[],[]).
elimina(X,[X|L1],L2) :-
   elimina(X,L1,L2).
elimina(X,[Y|L1],[Y|L2]) :-
   X \= Y,
   elimina(X,L1,L2).
```

Ejercicio 4.18 *Definir la relación* `repetidos(+L1,-L2)` *que se verifique si* `L2` *es la lista de los elementos repetidos de la lista* `L1`*. Por ejemplo,*

```
?- repetidos_1([1,2,4,3,4,1,3,5],L).
L = [1, 4, 3]
```

Solución: La primera definición de `repetidos` es

```
repetidos_1([],[]).
repetidos_1([X|L1],[X|L2]) :-
   memberchk(X,L1),
   elimina(X,L1,L3),
   repetidos_1(L3,L2).
repetidos_1([X|L1],L2) :-
   not(memberchk(X,L1)),
   repetidos_1(L1,L2).
```

con cortes se transforma en

```
repetidos_2([],[]).
repetidos_2([X|L1],[X|L2]) :-
   memberchk(X,L1), !,
   elimina(X,L1,L3),
   repetidos_2(L3,L2).
repetidos_2([_X|L1],L2) :-
   % not(memberchk(_X,L1)),
   repetidos_2(L1,L2).
```

4.17. Subconjunto maximal

Ejercicio 4.19 *Definir la relación* `subconjunto_maximal(+L1,-L2)` *que se verifica si* `L2` *es un subconjunto maximal de* `L1` *(es decir, es un conjunto de elementos de* `L1` *tal que sólo existe un elemento de* `L1` *que no pertenece a* `L2`*). Por ejemplo,*

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

Solución: La definición de `subconjunto_maximal` es

```
subconjunto_maximal(L1,L2) :-
   list_to_set(L1,L3),
   select(_,L3,L2).
```

Nota: La relación `list_to_set` se corresponde con la relación `lista_a_conjunto` definida en la página 53.

4.18. Suma de los elementos con posiciones múltiplos de n

Ejercicio 4.20 *Definir la relación* `suma_posiciones(+N,+L,-S)` *que se verifique si* `S` *es la suma de los elementos de la lista que ocupan las posiciones que son múltiplos de* `N`*. Por ejemplo,*

```
?-suma_posiciones(2,[3,5,7,9,1,2],S).
S = 16
?- suma_posiciones(3,[3,5,7,9,1,2],S).
S = 9
```

Solución: La definición de `suma_posiciones` es

```
suma_posiciones(N,L,S) :-
   elemento_y_resto(N,L,X,L1), !,
   suma_posiciones(N,L1,S1),
   S is X+S1.
suma_posiciones(_,_,0).
```

donde `elemento_y_resto(+N,+L1,-X,-L2)` se verifica si `X` es el elemento `N`–ésimo de `L1` y `L2` es la lista `L1` a partir del elemento `X`. Por ejemplo,

```
?- elemento_y_resto(3,[3,5,7,9,1,2],X,L).
X = 7
L = [9, 1, 2]
```

La definición de `elemento_y_resto` es

```
elemento_y_resto(N,L1,X,L2) :-
   length(L,N),
   append(L,L2,L1),
   last(L,X).
```

4.19. Compresión de listas

Ejercicio 4.21 *Definir la relación* `comprimida(+L1,-L2)` *que se verifique si* `L2` *es la lista obtenida sustituyendo cada sucesión de un elemento de* `L1` *por dicho elemento. Por ejemplo,*

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

Solución: Vamos a presentar dos definiciones. La primera definición de `comprimida` es

```
comprimida([],[]).
comprimida([X],[X]).
comprimida([X,X|L1],L2) :-
   comprimida([X|L1],L2).
comprimida([X,Y|L1],[X|L2]) :-
   X \= Y,
   comprimida([Y|L1],L2).
```

En la segunda definición se usa el corte

```
comprimida_2([],[]).
comprimida_2([X],[X]).
comprimida_2([X,Y|L1],L2) :-
   X = Y, !,
   comprimida_2([X|L1],L2).
comprimida_2([X,Y|L1],[X|L2]) :-
   % X \= Y,
   comprimida_2([Y|L1],L2).
```

4.20. Empaquetamiento de listas

Ejercicio 4.22 *Definir la relación* `empaquetada(+L1,-L2)` *que se verifique si* `L2` *es la lista obtenida sustituyendo cada sucesión de un elemento de* `L1` *por la lista formada por dicha sucesión. Por ejemplo,*

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

Solución: La definición de `empaquetada` es

```
empaquetada([],[]).
empaquetada([X|L1],[L2|L3]) :-
   empaquetada_aux(X,L1,L4,L2),
   empaquetada(L4,L3).
```

La relación `empaquetada_aux(X,L1,L4,L2)` se verifica si `L4` es la lista obtenida eliminando en `L1` todas las ocurrencias iniciales de `X` y `L2` es la lista formada por `X` y las ocurrencias iniciales de `X` en `L1`; por ejemplo,

```
?- empaquetada_aux(a,[a,a,c,c,b,b,b],L4,L2).
L4 = [c, c, b, b, b]
L2 = [a, a, a]
```

La definición de `empaquetada_aux` es

```
empaquetada_aux(X,[],[],[X]).
empaquetada_aux(X,[X|L1],L4,[X|L2]) :-
   empaquetada_aux(X,L1,L4,L2).
empaquetada_aux(X,[Y|L1],[Y|L1],[X]) :-
   X \= Y.
```

La definición anterior puede transformarse introduciendo corte en

```
empaquetada_aux_2(X,[],[],[X]).
empaquetada_aux_2(X,[X|L1],L4,[X|L2]) :-
   !,
   empaquetada_aux_2(X,L1,L4,L2).
empaquetada_aux_2(X,[Y|L1],[Y|L1],[X]).
```

4.21. Codificación por longitud

Ejercicio 4.23 *Definir la relación* `codificada(+L1,-L2)` *que se verifique si* `L2` *es la codificación por longitud de la lista* `L1`*; es decir, las sucesiones de un mismo elemento* `X` *de* `L1` *se codifican por términos de la forma* `N-X` *donde* `N` *es la longitud de la sucesión. Por ejemplo,*

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

(Indicación: Usar la relación `empaquetada` *(4.20)).*

Solución: La definición de `codificada` es

```
codificada(L1,L2) :-
   empaquetada(L1,L),
   codificada_aux(L,L2).
```

La relación `codificada_aux(+L1,-L2)` se verifica si, suponiendo que `L1` es una lista de la forma `[[E1,...,E1],...,[Em,....Em]]`), `L2` es la lista `[N1-E1,...Nm-Em]` donde `Ni` es la longitud de `[Ei,...,Ei]`. Por ejemplo.

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

La definición de `codificada_aux` es

```
codificada_aux([],[]).
codificada_aux([[X|Y]|L1],[N-X|L2]) :-
   length([X|Y],N),
   codificada_aux(L1,L2).
```

4.22. Codificación reducida por longitud

Ejercicio 4.24 *Definir la relación* `codificada_reducida(+L1,-L2)` *que se verifique si* `L2` *es la codificación reducida por longitud de la lista* `L1`*; es decir, las sucesiones de un mismo elemento* `X` *de* `L1` *se codifican por términos de la forma* `N-X` *donde* `N` *es la longitud de la sucesión cuando* `N` *es mayor que 1 y por* `X` *cuando* `N` *es igual a 1. Por ejemplo,*

```
?- codificada_reducida([a,b,b,a,a,a,c,b,b,b],L).
L = [a, 2-b, 3-a, c, 3-b]
```

(Indicación: Usar la relación `codificada` *(4.21)).*

Solución: La definición de `codificada_reducida` es

```
codificada_reducida(L1,L2) :-
   codificada(L1,L),
   codificada_reducida_aux(L,L2).
```

La relación `codificada_reducida_aux(+L1,-L2)` se verifica si `L2` es la lista obtenida transformando los elementos de `L1` de la forma `1-X` por `X` y dejando los restantes elementos de la misma forma (se supone que `L1` es una lista de la forma `[N1-E1,...,Nm-Em]`). Por ejemplo,

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

La definición de `codificada_reducida_aux` es

```
codificada_reducida_aux([],[]).
codificada_reducida_aux([1-X|L1],[X|L2]) :-
   codificada_reducida_aux(L1,L2).
codificada_reducida_aux([N-X|L1],[N-X|L2]) :-
   N > 1,
   codificada_reducida_aux(L1,L2).
```

La definición anterior puede simplificarse introduciendo un corte:

```
codificada_reducida_aux([],[]).
codificada_reducida_aux([1-X|L1],[X|L2]) :-
   !,
   codificada_reducida_aux(L1,L2).
codificada_reducida_aux([N-X|L1],[N-X|L2]) :-
   % N > 1,
   codificada_reducida_aux(L1,L2).
```

4.23. Decodificación de lista

Ejercicio 4.25 *Definir la relación* `decodificada(+L1,-L2)` *que, dada la lista* `L1`*, devuelve la lista* `L2` *cuya codificación reducida por longitud es* `L1`*. Por ejemplo,*

```
?- decodificada([a,2-b,3-a,c,3-b],L).
L = [a, b, b, a, a, a, c, b, b, b]
```

Solución: La definición de `decodificada` es

```
decodificada([],[]).
decodificada([1-X|L1],[X|L2]) :-
   !,
   decodificada(L1,L2).
decodificada([N-X|L1],[X|L2]) :-
   % N > 1,
   !,
   N1 is N - 1,
   decodificada([N1-X|L1],L2).
decodificada([X|L1],[X|L2]) :-
   % X es atómico
   decodificada(L1,L2).
```

4.24. Codificación reducida directa

Ejercicio 4.26 *Definir la relación* `codificada_directa(+L1,-L2)` *que se verifica si* `L2` *es la codificación reducida de* `L1` *(es decir, las sucesiones de un mismo elemento* `X` *de* `L1` *se codifican por términos de la forma*

N-X *donde* N *es la longitud de la sucesión cuando* N *es mayor que 1 y por* X *cuando* N *es igual a 1), pero en su definición no se crean listas de elementos repetidos como en la definición de* codificada_reducida*. Por ejemplo,*

```
?- codificada_directa([a,b,b,a,a,a,c,b,b,b],L).
L = [a, 2-b, 3-a, c, 3-b] ;
```

Solución: La definición de codificada_directa es

```
codificada_directa([],[]).
codificada_directa([X|L1],[T|L2]) :-
   cuenta_y_resto(X,[X|L1],N,L3),
   término(N,X,T),
   codificada_directa(L3,L2).
```

La relación cuenta_y_resto(+X,+L1,-N,-L2) se verifica si N es el número de veces que aparece X en la cabeza de la lista L1 y L2 es el resto de la lista L1 cuando se le quita la sucesión de elementos X de su cabeza. Por ejemplo,

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

La definición de cuenta_y_resto es

```
cuenta_y_resto(X,[X|L1],N,L2) :-
   cuenta_y_resto(X,L1,M,L2),
   N is M+1.
cuenta_y_resto(X,[Y|L],0,[Y|L]) :-
   X \= Y.
cuenta_y_resto(_,[],0,[]).
```

La definición anterior puede simplificarse con cortes:

```
cuenta_y_resto_1(X,[X|L1],N,L2) :-
   !,
   cuenta_y_resto_1(X,L1,M,L2),
   N is M+1.
cuenta_y_resto_1(_,L,0,L).
```

La relación término(+N,+X,-T) se verifica si T es el término correspondiente al número N y al elemento X (es decir, T es X si N es 1 y es N-X en otro caso). Por ejemplo,

```
?- término(1,a,T).
T = a
?- término(2,a,T).
T = 2-a
```

La definición de término es

```
término(1,X,X).
término(N,X,N-X) :-
   N > 1.
```

La definición anterior puede simplificarse con cortes:

```
término_1(1,X,X) :- !.
término_1(N,X,N-X). % :- N > 1.
```

4.25. Cota superior de una lista de números

Ejercicio 4.27 *Definir la relación* `cota_superior(+L,+N)` *que se verifique si* `N` *es una cota superior de* `L` *(es decir, todos los elementos de* `L` *son menores o iguales que* `N`*). Por ejemplo,*

```
?- cota_superior([1,5,3],7).
Yes
?- cota_superior([1,5,3],5).
Yes
?- cota_superior([1,5,3],4).
No
```

Dar dos definiciones, una recursiva y la otra no recursiva.

Solución: La definición recursiva de `cota_superior` es

```
cota_superior_1([],_).
cota_superior_1([X|L],N) :-
   X =< N,
   cota_superior_1(L,N).
```

La definición no recursiva de `cota_superior` es

```
cota_superior_2(L,N) :-
   \+ (member(X,L), X > N).
```

Mediante el siguiente ejemplo se compara la eficiencia de las dos definiciones

```
?- numlist(1,10000,_L), time(cota_superior_1(_L,10000)).
% 20,001 inferences, 0.00 CPU in 0.01 seconds (0% CPU, Infinite Lips)
?- numlist(1,10000,_L), time(cota_superior_2(_L,10000)).
% 30,002 inferences, 0.01 CPU in 0.01 seconds (93% CPU, 3000200 Lips)
?- numlist(1,100000,_L), time(cota_superior_1(_L,100000)).
% 200,001 inferences, 0.07 CPU in 0.07 seconds (99% CPU, 2857157 Lips)
?- numlist(1,100000,_L), time(cota_superior_2(_L,100000)).
% 300,002 inferences, 0.08 CPU in 0.08 seconds (104% CPU, 3750025 Lips)
```

4.26. Dientes de sierra

Ejercicio 4.28 *Definir la relación* `diente(+L,-L1,-X,-L2)` *que se verifique si* `L` *se compone de una lista* `L1` *de números estrictamente creciente hasta un cierto número* `X` *que llamaremos* cima*, de la cima y de una lista* `L2` *de números estrictamente decreciente. Las listas tiene* `L1` *y* `L2` *tienen que ser no vacías y la cima* `X` *es el mayor elemento de* `L`*. Por ejemplo,*

```
?- diente([1,2,5,4,3,1],L1,X,L2).
L1 = [1, 2]
X = 5
L2 = [4, 3, 1] ;
No
?- diente([1,2,5],L1,X,L2).
No
```

Las listas que poseen esta forma de descomposición se llaman dientes*.*

Solución: La definición de `diente` es

```
diente(L,[X1|L1],X,[X2|L2]) :-
   append([X1|L1],[X,X2|L2],L),
   creciente([X1|L1]),
   last([X1|L1],Y),
   Y < X,
   decreciente([X,X2|L2]).
```

La relación `creciente(+L)` se verifica si la lista de números `L` es estrictamente creciente.

```
creciente([_]).
creciente([A,B|L]) :-
   A < B,
   creciente([B|L]).
```

La relación `decreciente(+L)` se verifica si la lista de números `L` es estrictamente decreciente.

```
decreciente([_]).
decreciente([A,B|L]) :-
   A > B,
   decreciente([B|L]).
```

Ejercicio 4.29 *Una* sierra *es una lista numérica compuesta por la yuxtaposición de dientes. Nótese que dos dientes consecutivos deben compartir un elemento. Por ejemplo [1,2,1,3,1] es una sierra compuesta por los dientes [1,2,1] y [1,3,1], pero [1,2,1,1,3,1] no es una sierra.*

Definir la relación `dientes_de_sierra(+L1,?L2)` *que se verifique si* `L1` *es una sierra y* `L2` *es la lista de los dientes de* `L1`*. Por ejemplo,*

```
?- dientes_de_sierra([1,2,1,3,1],L).
L = [[1, 2, 1], [1, 3, 1]] ;
No
?- dientes_de_sierra([1,2,1,1,3,1],L).
No
```

Solución: La definición de `dientes_de_sierra` es

```
dientes_de_sierra(L,[L]) :-
   diente(L,_,_,_), !.
dientes_de_sierra(L,[L1|R]) :-
   append(L1,L2,L),
   diente(L1,_,_,_),
   last(L1,X),
   dientes_de_sierra([X|L2],R).
```

El corte aumenta la eficiencia como se aprecia en el siguiente ejemplo con la definición con corte

```
?- numlist(1,100,_L1), append(_L1,[1|_L1],_L2), time(dientes_de_sierra(_L2,L)).
% 581,868 inferences, 0.36 CPU in 0.37 seconds (96% CPU, 1616300 Lips)
No
```

y el mismo ejemplo con la definición sin el corte

```
?- numlist(1,100,_L1), append(_L1,[1|_L1],_L2), time(dientes_de_sierra(_L2,L)).
% 3,147,270 inferences, 0.92 CPU in 0.94 seconds (98% CPU, 3420946 Lips)
No
```

Capítulo 5

Programación lógica de segundo orden

5.1. Determinación de un número por su factorial

Ejercicio 5.1 *Definir la relación* `factorial_inverso(+X,-N)` *que se verifique si* `X` *es el factorial de* `N`. *Por ejemplo,*

```
?- factorial_inverso(120,N).
N = 5 ;
No
?- factorial_inverso(80,N).
No
```

Solución: Presentamos tres definiciones y comparamos su eficicencia.
La primera definición usa un acumulador para almacenar los candidatos de la solución.

```
factorial_inverso_1(X,N) :-
   factorial_inverso_1_aux(X,1,N).
```

La relación `factorial_inverso_1_aux(+X,+A,-N)` se verifica si `N` es el menor número mayor o igual que `A` cuyo factorial es `X`.

```
factorial_inverso_1_aux(X,A,A) :-
   factorial(A,X).
factorial_inverso_1_aux(X,A,N) :-
   factorial(A,N1),
   N1 < X,
   A1 is A + 1,
   factorial_inverso_1_aux(X,A1,N).
```

La relación `factorial(+N,-X)` se verifica si `X` es el factorial de `N`.

```
factorial(1,1).
factorial(N,X) :-
   N > 1,
   N1 is N-1,
```

```
   factorial(N1,X1),
   X is X1 * N.
```

La segunda definición se diferencia de la anterior en que almacena en memoria los factoriales de los candidatos considerados.

```
factorial_inverso_2(X,N) :-
   factorial_inverso_2_aux(X,1,N).
```

La relación `factorial_inverso_2_aux(+X,+A,-N)` se verifica si `N` es el menor número mayor o igual que `A` cuyo factorial (con memoria) es `X`.

```
factorial_inverso_2_aux(X,A,A) :-
   factorial_con_memoria(A,X).
factorial_inverso_2_aux(X,A,N) :-
   factorial_con_memoria(A,N1),
   N1 < X,
   C1 is A + 1,
   factorial_inverso_2_aux(X,C1,N).
```

La relación `factorial_con_memoria(+N,-X)` se verifica si `X` es el factorial de `N`. Además almacena en la base de datos internas los factoriales calculados.

```
:- dynamic factorial_con_memoria/2.

factorial_con_memoria(1,1).
factorial_con_memoria(N,X) :-
   N > 1,
   N1 is N-1,
   factorial_con_memoria(N1,X1),
   X is X1 * N,
   asserta(factorial_con_memoria(N,X) :- !).
```

En la tercera definición se utiliza dos acumuladores para almacenar el candidato y el factorial del candidato anterior.

```
factorial_inverso_3(X,N) :-
   factorial_inverso_aux_3(X,1,1,N).
```

La relación `factorial_inverso_aux_3(+X,+A,+F,-N)` se verifica si $X = A * (A+1) * \cdots * N$ (de forma que si $A = 1$ entonces $X = N!$).

```
factorial_inverso_aux_3(X,A,F,A) :-
   A*F =:= X.
factorial_inverso_aux_3(X,A,F,N) :-
   F1 is A*F,
   F1 < X, !,
   A1 is A + 1,
   factorial_inverso_aux_3(X,A1,F1,N).
```

En los siguientes ejemplos se compara la eficiencia.

```
?- factorial(250,_X), time(factorial_inverso_1(_X,_N)).
% 249,501 inferences, 0.28 CPU in 0.30 seconds (94% CPU, 891075 Lips)

?- factorial(250,_X), time(factorial_inverso_2(_X,_N)).
% 3,487 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)

?- factorial(250,_X), time(factorial_inverso_3(_X,_N)).
% 1,248 inferences, 0.00 CPU in 0.01 seconds (0% CPU, Infinite Lips)


?- factorial(1000,_X), time(factorial_inverso_1(_X,_N)).
% 321,722 inferences, 0.46 CPU in 0.47 seconds (98% CPU, 699396 Lips)
ERROR: Out of local stack

?- factorial(1000,_X), time(factorial_inverso_2(_X,_N)).
% 13,987 inferences, 0.07 CPU in 0.07 seconds (96% CPU, 199814 Lips)

?- factorial(1000,_X), time(factorial_inverso_3(_X,_N)).
% 4,998 inferences, 0.03 CPU in 0.03 seconds (104% CPU, 166600 Lips)
```

5.2. Árbol de resolución y definiciones equivalentes

Ejercicio 5.2 *Se piden los siguientes apartados:*

1. *Dibujar el árbol de resolución correspondiente al programa*

```
p([],[]).
p([X|A],[X|B]) :-
    X > 4, !,
    p(A,B).
p([X|A],B) :-
    p(A,B).
```

   y al objetivo

```
?- p([5,1,6],B).
```

2. *Explicar la relación que hay entre* `L1` *y* `L2` *cuando se verifica* `p(L1,L2)`.

3. *Dar una definición* no recursiva *del predicado* `p(L1,L2)`.

Solución:

1. El árbol de resolución está en la figura 5.1 (página 72).

2. la relación `p(L1,L2)` se verifica si `L2` es la lista de los elementos de `L1` que son mayores que 4.

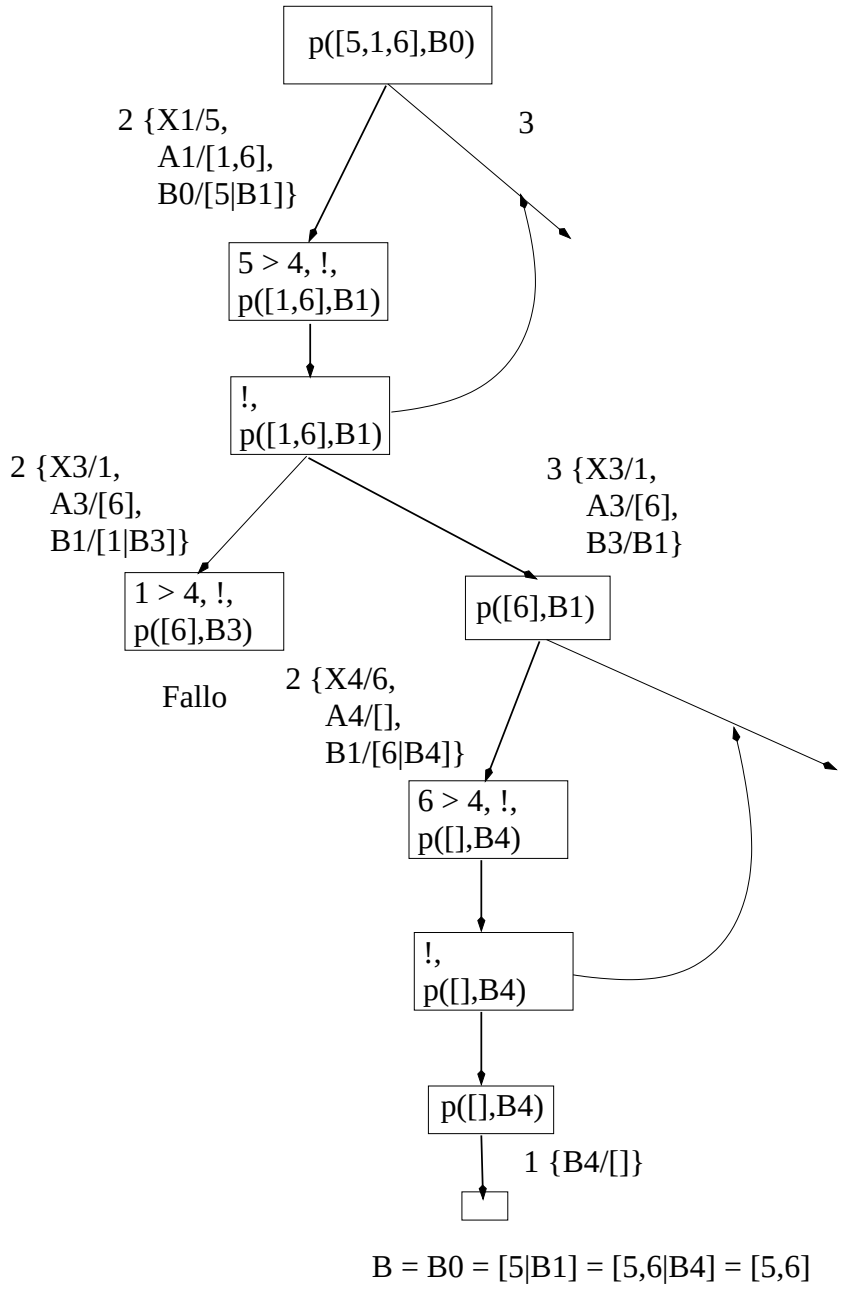


Figura 5.1: Árbol de resolución

3. Una definición no recursiva de p es

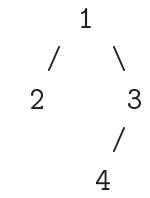
```
p1(L1,L2) :-
   findall(X,(member(X,L1),X>4),L2).
```

5.3. Nodos de una generación en una lista de árboles binarios

Ejercicio 5.3 *Un árbol binario es vacío o consta de tres partes: la raíz (que debe de ser un número positivo), el subárbol izquierdo (que debe ser un árbol binario) y el subárbol derecho (que debe ser un árbol binario). Usaremos la siguiente representación*

- `nil` *representa el árbol vacío*
- `t(I,R,D)` *representa el árbol de la raíz* `R`*, subárbol izquierdo* `I` *y subárbol derecho* `D`*.*

Por ejemplo, `t(t(nil,2,nil),1,t(t(nil,4,nil),3,nil))` *representa el árbol*



Definir la relación `generación(+N,+L1,-L2)` *que se verifique si* `L2` *es la lista de nodos de la generación* `N` *de la lista de árboles* `L1`*. Por ejemplo,*

```
?- generación(0,[t(t(nil,2,nil),3,nil),t(nil,4,t(nil,5,nil))],L).
L = [3, 4]
?- generación(1,[t(t(nil,2,nil),3,nil),t(nil,4,t(nil,5,nil))],L).
L = [2, 5]
?- generación(2,[t(t(nil,2,nil),3,nil),t(nil,4,t(nil,5,nil))],L).
L = []
```

Solución: La definición de `generación` es

```
generación(0,L,G):-
   findall(R,member(t(_,R,_),L),G).
generación(N,L,G):-
   N > 0,
   elimina_raices(L,L1),
   N1 is N-1,
   generación(N1,L1,G).
```

donde `elimina_raices(+L1,-L2)` se verifica si `L2` es la lista de los árboles obtenidos de la lista de árboles `L1` eliminando sus raices. Por ejemplo,

```
?- elimina_raices([t(t(nil,2,nil),3,nil),t(nil,4,t(nil,5,nil))],L).
L = [t(nil, 2, nil), nil, nil, t(nil, 5, nil)]
```

La definición de `elimina_raices` es

```
elimina_raices([],[]).
elimina_raices([nil|L1],L2):-
   elimina_raices(L1,L2).
elimina_raices([t(I,_,D)|L1],[I,D|L2]):-
   elimina_raices(L1,L2).
```

5.4. Lista de elementos únicos

Ejercicio 5.4 *Definir la relación* `únicos(+L1,-L2)` *que se verifique si* `L2` *es la lista de los elementos que ocurren solamente una vez en la lista* `L1`*. Por ejemplo,*

```
?- únicos([2,5,3,2],L).
L = [5, 3]
?- únicos([2,5,5,2],L).
L = []
```

Solución: La definición de `únicos` es

```
únicos(L1,L2) :-
   findall(X,es_único(X,L1),L2).
```

donde la relación `es_único(?X,+L)` se verifica si `X` es un elemento que ocurre solamente una vez en la lista `L`. Por ejemplo,

```
?- es_único(X,[2,5,3,2]).
X = 5 ;
X = 3 ;
No
```

```
es_único(X,L) :-
   select(X,L,R),
   not(memberchk(X,R)).
```

5.5. Elementos más frecuentes de una lista

Ejercicio 5.5 *Definir el predicado* `populares(L1,L2)` *que se verifique si* `L2` *es la lista de los elementos de* `L1` *que aparecen el mayor número de veces. Por ejemplo,*

```
?- populares([rosa,juan,david,manu,rosa,nuria,david],L).
L = [david,rosa]
```

Solución: La definición de `populares` es

```
populares(L1,L2) :-
   setof(X,
         ((member(X,L1),
           cuenta(X,L1,N1),
           not((member(Y,L1),
                cuenta(Y,L1,N2),
                N1 < N2)))),
         L2).
```

La relación `cuenta(+X,+L,-N)` se verifica si `N` es el número de veces que aparece el elemento `X` en la lista `L`. Por ejemplo,

```
?- cuenta(d,[r,j,d,m,r,n,d],N).
N = 2
```

La definición de `cuenta` es

```
cuenta(_,[],0).
cuenta(A,[B|L],N) :-
   A == B, !,
   cuenta(A,L,M),
   N is M+1.
cuenta(A,[_B|L],N) :-
   % A \== _B,
   cuenta(A,L,N).
```

5.6. Problema $3n+1$

Ejercicio 5.6 *Consideremos la función siguiente definida sobre los números naturales:*

$$f(x) = \begin{cases} 3x+1, & \text{si } x \text{ es impar;} \\ x/2, & \text{si } x \text{ es par} \end{cases}$$

se pide:

1. *Definir la relación* `sucesión(+X,?L)` *que se verifique si* `L` *es la lista de los elementos* X, $f(X)$, $f(f(X))$, $\ldots$, $f^n(X)$ *tal que* $f^n(X) = 1$. *Por ejemplo,*

   ```
   ?- sucesión(3,L).
   L = [3, 10, 5, 16, 8, 4, 2, 1]
   ```

   `L` *se llama la* sucesión generada por `N`.

2. *Definir la relación* `longitudes(+X,?L)` *que se verifica si* `L` *la lista de pares* `Y-N` *donde* `Y` *es un número de 1 a* `X` *y* `N` *es la longitud de sucesión generada por* `Y`. *Por ejemplo,*

   ```
   ?- longitudes(5,L).
   L = [1-1, 2-2, 3-8, 4-3, 5-6]
   ```

3. *Definir la relación* `longitud_máx(+X,?P)` *que se verifica si* `P` *es un par de la forma* `Y-N` *donde* `Y` *es un número entre 1 y* `X` *tal que la longitud de la sucesión generada por* `Y` *es la más larga de las sucesiones generadas por 1,2,...,*`X` *y* `N` *es la longitud de dicha sucesión. Por ejemplo,*

```
?- longitud_máx(10,L).
L = 9-20
```

4. *Definir* `menor_que_genera_mayor(+N,-M)` *que se verifique si* `M` *es el menor número natural tal que la longitud de la sucesión generada por* `M` *es mayor que* `N`*. Por ejemplo,*

```
?- menor_que_genera_mayor(100,N).
N = 27
```

Solución: La relación `f(+X,-Y)` se verifica si `Y=f(X)`.

```
f(X,Y) :-
   X mod 2 =:= 0, !,
   Y is X/2.
f(X,Y) :-
   % X mod 2 =/= 0,
   Y is 3*X+1.
```

Solución del apartado 1: La definición de `sucesión` es

```
sucesión(1,[1]) :- !.
sucesión(X,[X|L]) :-
   % X =/= 1,
   f(X,Y),
   sucesión(Y,L).
```

Solución del apartado 2: Presentamos dos definiciones de `longitudes`.
La primera definición de `longitudes` es recursiva.

```
longitudes(X,L) :-
   longitudes_aux(X,L1),
   reverse(L1,L).
longitudes_aux(1,[1-N]) :-
   !,
   sucesión(1,L),
   length(L,N).
longitudes_aux(X,[X-N|L]) :-
   % X > 1,
   sucesión(X,L1),
   length(L1,N),
   Y is X-1,
   longitudes_aux(Y,L).
```

La segunda definición de `longitudes`, usando `findall`, es

```
longitudes_2(X,L) :-
   findall(Y-N,(between(1,X,Y),sucesión(Y,S),length(S,N)),L).
```

Solución del apartado 3: La definición de `longitud_máx` es

```
longitud_máx(X,Y-N) :-
   longitudes(X,L),
   member(Y-N,L),
   \+ (member(_Z-M,L), M > N).
```

Solución del apartado 4: La definición de `menor_que_genera_mayor` es

```
menor_que_genera_mayor(N,M) :-
   menor_que_genera_mayor_aux(N,1,M).
menor_que_genera_mayor_aux(N,M,M) :-
   sucesión(M,L),
   length(L,X),
   X > N, !.
menor_que_genera_mayor_aux(N,X,M) :-
   Y is X+1,
   menor_que_genera_mayor_aux(N,Y,M).
```

5.7. Números perfectos

En los ejercicio de esta sección estudiamos los números perfectos (es decir, iguales a la suma de sus divisores propios) y conceptos relacionados.

Ejercicio 5.7 *Definir la relación* `divisores_propios(+N,-L)` *que se verifique si* `L` *es la lista ordenada de los divisores propios del número* `N`*. Por ejemplo,*

```
?- divisores_propios(42,L).
L = [1, 2, 3, 6, 7, 14, 21]
```

Solución: La definición de `divisores_propios` es

```
divisores_propios(N,L) :-
   N1 is N -1,
   findall(X,(between(1,N1,X), 0 =:= N mod X),L).
```

Ejercicio 5.8 *Definir la relación* `suma_divisores_propios(+N,-S)` *que se verifique si* `S` *es la suma de los divisores propios del número* `N`*. Por ejemplo,*

```
?- suma_divisores_propios(42,S).
S = 54
?- suma_divisores_propios(1,S).
S = 0
```

Solución: La definición de `suma_divisores_propios` es

```
suma_divisores_propios(N,S) :-
   divisores_propios(N,L),
   suma_lista(L,S).
```

La relación `suma_lista(+L,-N)` se verifica si `N` es la suma de los elementos de la lista `L`.

```
suma_lista([],0).
suma_lista([X|L],S) :-
   suma_lista(L,S1),
   S is X + S1.
```

Ejercicio 5.9 *Clasificamos los números naturales en tres tipos:*

- `N` *es de tipo a si* `N` *es mayor que la suma de sus divisores propios*
- `N` *es de tipo b si* `N` *es igual que la suma de sus divisores propios*
- `N` *es de tipo c si* `N` *es menor que la suma de sus divisores propios*

Definir la relación `tipo(+N,-T)` *que se verifique si* `T` *es el tipo del número* `N`*. Por ejemplo,*

```
?- tipo(10,T).
T = a
?- tipo(28,T).
T = b
?- tipo(12,T).
T = c
```

Solución: La definición de `tipo` es

```
tipo(N,T) :-
   suma_divisores_propios(N,S),
   tipo_aux(N,S,T).

tipo_aux(N,S,a) :- N > S, !.
tipo_aux(N,N,b) :- !.
tipo_aux(N,S,c). % :- N < S.
```

Ejercicio 5.10 *Definir la relación* `clasifica(+N,-L)` *que se verifique si* `L` *es la lista de tipos de los números comprendidos entre 1 y* `N`*. Por ejemplo,*

```
?- clasifica(20,L).
L = [a, a, a, a, a, b, a, a, a, a, a, c, a, a, a, a, a, c, a, c]
```

Solución: La definición de `clasifica` es

```
clasifica(N,L) :-
   findall(T,(between(1,N,X),tipo(X,T)),L).
```

Ejercicio 5.11 *Definir la relación* `promedio(+N,-A,-B,-C)` *que se verifique si* `A`*,* `B` *y* `C` *son las cantidades de números naturales menores o iguales que* `N` *de tipo a, b y c, respectivamente. Por ejemplo,*

```
?- promedio(20,A,B,C).
A = 16
B = 1
C = 3
```

Solución: La definición de `promedio` es

```
promedio(N,A,B,C) :-
   clasifica(N,L),
   promedio_aux(L,A,B,C).

promedio_aux([],0,0,0).
promedio_aux([a|L],A1,B,C) :-
   promedio_aux(L,A,B,C),
   A1 is A+1.
promedio_aux([b|L],A,B1,C) :-
   promedio_aux(L,A,B,C),
   B1 is B+1.
promedio_aux([c|L],A,B,C1) :-
   promedio_aux(L,A,B,C),
   C1 is C+1.
```

Ejercicio 5.12 *Definir la relación* `menor(+N,-X)` *que se verifique si* `X` *es el menor número tal que la cantidad de números naturales menores o iguales que* `X` *de tipo a es* `N`*. Por ejemplo,*

```
?- menor(20,X).
X = 25
```

Solución: La definición de `menor` es

```
menor(N,X) :-
   menor_aux(N,N,X).

menor_aux(N,M,M) :-
   promedio(M,N,_,_), !.
menor_aux(N,M,X) :-
   M1 is M+1,
   menor_aux(N,M1,X).
```

5.8. Determinación de triángulos equiláteros

Ejercicio 5.13 *Un polígono se representa por su nombre y las longitudes de sus lados. Definir la relación* `es_equilátero(+P)` *que se verifica si el polígono* `P` *es equilátero (es decir, que todos sus lados son iguales). Por ejemplo,*

```
?- es_equilátero(triángulo(4,4,4)).
Yes
?- es_equilátero(cuadrilátero(3,4,5,3)).
No
```

Solución: La definición de `es_equilátero` es

```
es_equilátero(P) :-
   P =.. [_|L],
   todos_iguales(L).
```

La relación `todos_iguales` está definida en la página 17.

5.9. Operación binaria aplicada a listas

Ejercicio 5.14 *Definir la relación* `operación_listas(+O,+L1,+L2,-L3)` *que se verifica si* `L3` *es la lista obtenida aplicando la operación binaria* `O` *a los elementos de* `L1` *y* `L2` *que ocupan la misma posición. Por ejemplo,*

```
?- operación_lista(+,[1,2,3],[4,5,6],L).
L = [5, 7, 9]
?- operación_lista(*,[1,2,3],[4,5,6],L).
L = [4, 10, 18]
```

Nota: *Se supone que* `L1` *y* `L2` *tienen la misma longitud)*

Solución: La definición de `operación_lista` es

```
operación_lista(_,[],[],[]).
operación_lista(O,[X1|L1],[X2|L2],[X3|L3]) :-
   E =.. [O,X1,X2],
   X3 is E,
   operación_lista(O,L1,L2,L3).
```

5.10. Números en un término

Ejercicio 5.15 *Definir la relación* `números(+T,-L)` *que que se verifique si* `L` *es el conjunto de todos los números que ocurren en el término cerrado* `T`*. Por ejemplo,*

```
?- números(f(a,g(c,5),2),L).
L = [5, 2]
?- números(a+3+b*(sen(2)+3),L).
L = [2, 3]
?- números(a+b,L).
L = []
```

Solución: La definición de `números` es

```
números(T,[T]) :-
   number(T), !.
números(T,L) :-
   % not(number(T)),
   T =.. [_|L1],
   números_de_lista(L1,L).
```

La relación `números_de_lista(+L1,-L2)` se verifica si `L2` es el conjunto de números en la lista de términos `L1`. Por ejemplo,

```
?- números_de_lista([a+3, b*(sen(2)+3)],L).
L = [2, 3]
```

La definición de `números_de_lista` es

```
números_de_lista([],[]).
números_de_lista([T|L1],L2) :-
   números(T,L3),
   números_de_lista(L1,L4),
   union(L3,L4,L2).
```

5.11. Palabra sin vocales

Ejercicio 5.16 *Definir la relación* `elimina_vocales(+P1,-P2)` *que se verifique si* `P2` *es la palabra que se obtiene al eliminar todas las vocales de la palabra* `P1`. *Por ejemplo,*

```
?- elimina_vocales(sevillano,P).
P = svlln
```

Solución: La definición de `elimina_vocales` es

```
elimina_vocales(P1,P2) :-
   name(P1,L1),
   códigos_vocales(L),
   findall(N,(member(N,L1),not(member(N,L))),L2),
   name(P2,L2).
```

La relación `códigos_vocales(?L)` se verifica si `L` es la lista de los códigos ASCII de las vocales.

```
códigos_vocales(L) :-
   name(aeiou,L).
```

5.12. Palabras maximales

Ejercicio 5.17 *Definir la relación* `longitud(+P,-N)` *que se verifique si* `N` *es la longitud de la palabra* `P`. *Por ejemplo,*

```
?- longitud(ana,N).
N = 3
```

Solución: La definición de `longitud` es

```
longitud(P,N) :-
   name(P,L),
   length(L,N).
```

Ejercicio 5.18 *Definir la relación* `palabra_maximal(+L,-P)` *que se verifique si* `P` *es una palabra maximal (es decir, de máxima longitud) de la lista de palabras* `L`. *Por ejemplo,*

```
?- palabra_maximal([eva,y,ana,se,van],P).
P = eva ;
P = ana ;
P = van ;
No
```

Solución: La definición de `palabra_maximal` es

```
palabra_maximal(L,P) :-
   select(P,L,R),
   longitud(P,N),
   not((member(P1,R), longitud(P1,N1), N < N1)).
```

Ejercicio 5.19 *Definir la relación* `palabras_maximales(+L1,-L2)` *que se verifique si* `L2` *es la lista de las palabras maximales de la lista de palabras* `L1`. *Por ejemplo,*

```
?- palabras_maximales([eva,y,ana,se,van],L).
L = [eva, ana, van]
```

Solución: La definición de `palabras_maximales` es

```
palabras_maximales(L1,L2) :-
   findall(P,palabra_maximal(L1,P),L2).
```

5.13. Clausura transitiva de una relación

Ejercicio 5.20 *La clausura transitiva de una relación binaria R es la menor relación transitiva que contiene a R; por ejemplo, la clausura transitiva de* $\{(a,b),(b,c)\}$ *es* $\{(a,b),(b,c),(a,c)\}$. *Definir el predicado* `clausura_transitiva(R,X,Y)` *que se verifique si* `(X,Y)` *está en la clausura transitiva de la relación* `R`. *Por ejemplo, suponiendo que se han definido las relaciones* `p` *y* `q` *por*

```
p(a,b).
p(b,c).
q(c,b).
q(b,a).
```

se tiene

```
?- clausura_transitiva(p,X,Y).
X = a   Y = b ;
X = b   Y = c ;
X = a   Y = c ;
No
?- clausura_transitiva(q,X,Y).
X = c   Y = b ;
X = b   Y = a ;
X = c   Y = a ;
No
```

Solución: La definición de `clausura_transitiva` es

```
clausura_transitiva(R,X,Y) :-
   apply(R,[X,Y]).
clausura_transitiva(R,X,Y) :-
   apply(R,[X,Z]),
   clausura_transitiva(R,Z,Y).
```

5.14. Traducción de cifras a palabras

Ejercicio 5.21 *Definir la relación* `traducción(+L1,-L2)` *que se verifique si* `L2` *es la lista de palabras correspondientes a los dígitos de la lista* `L1`*. Por ejemplo,*

```
?- traducción([1,3],L).
L = [uno, tres]
```

(Indicación: Usar la relación auxiliar `nombre(D,N)` *que se verifica si* `N` *es el nombre del dígito* `D`*).*

Solución: La definición de la relación auxiliar `nombre` es

```
nombre(0,cero).
nombre(1,uno).
nombre(2,dos).
nombre(3,tres).
nombre(4,cuatro).
nombre(5,cinco).
nombre(6,seis).
nombre(7,siete).
nombre(8,ocho).
nombre(9,nueve).
```

Se presentan tres definiciones de `traducción`.

Primera solución: Una definición de `traducción(L1,L2)`, por recursión en `L1`, es

```
traducción_1([],[]).
traducción_1([D|L1],[N|L2]) :-
   nombre(D,N),
   traducción_1(L1,L2).
```

Segunda solución: Una definición de `traducción` usando `findall` es

```
traducción_2(L1,L2) :-
   findall(N,(member(D,L1),nombre(D,N)),L2).
```

Tercera solución: Una definición de `traducción` usando `maplist` es

```
traducción_3(L1,L2) :-
   maplist(nombre,L1,L2).
```

5.15. Transformación de lista dependiente de la posición

Ejercicio 5.22 *Definir la relación* `transforma(+L1,-L2)` *que se verifique si* `L2` *es la lista obtenida sumándole a cada elemento numérico de* `L1` *su posición en la lista. Por ejemplo,*

```
?- transforma([1,1,1,a,b,c,1,1,1],L).
L = [2, 3, 4, a, b, c, 8, 9, 10]
?- transforma([1,2,a,5,2,b,3,1],L).
L = [2, 4, a, 9, 7, b, 10, 9]
```

Dar dos definiciones, una recursiva y otra no–recursiva.

Solución: La definición de recursiva de `transforma` es

```
transforma(L1,L2) :-
   transforma_aux(L1,1,L2).
```

donde `transforma_aux(+L1,+N,-L2)` se verifica si `L2` es la lista obtenida añadiéndole a cada elemento numérico de `L1` la suma de `N` y su posición en la lista. Por ejemplo,

```
transforma_aux([],_,[]).
transforma_aux([X|L1],N,[Y|L2]) :-
   number(X), !,
   Y is X+N,
   N1 is N+1,
   transforma_aux(L1,N1,L2).
transforma_aux([X|L1],N,[X|L2]) :-
   % not(number(X)),
   N1 is N+1,
   transforma_aux(L1,N1,L2).
```

La definición no recursiva de `transforma` es

```
transforma_2(L1,L2) :-
   lista_de_posiciones(L1,L),
   maplist(suma,L1,L,L2).
```

donde `lista_de_posiciones(+L1,-L2)` se verifica si `L2` es la lista de posiciones correspondiente a la lista `L1`. Por ejemplo,

```
?- lista_de_posiciones([1,1,1,a,b,c,1,1,1],L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
lista_de_posiciones(L1,L2) :-
   length(L1,N),
   findall(X,between(1,N,X),L2).
```

y `suma(+X,+Y,-Z)` se verifica si `Z` es la suma de `X` y el número `Y`, cuando `X` es un número y es igual a `X`, en caso contrario. Por ejemplo,

```
?- suma(3,2,Z).
Z = 5
?- suma(b,2,Z).
Z = b
```

```
suma(X,Y,Z) :-
   number(X), !,
   Z is X+Y.
suma(X,_,X).
```

5.16. Aplanamiento de listas

Ejercicio 5.23 *Definir la relación* `aplana(+L1,?L2)` *que se verifique si* `L2` *es la lista obtenida reemplazando, recursivamente, cada lista de* `L1` *por sus elementos. Por ejemplo,*

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

Solución: Para definir `aplana(X,Y)` vamos a generalizar su dominio de forma que si `X` no es una lista, entonces `Y` es la lista cuyo único elemento es `X`.

```
aplana(X,[X]) :-
   \+ is_list(X).
aplana([],[]).
aplana([X|L1],L2) :-
   aplana(X,L3),
   aplana(L1,L4),
   append(L3,L4,L2).
```

Capítulo 6

Estilo y eficiencia en programación lógica

6.1. Número de Hardy

En cierta ocasión, el matemático Ramanujan estaba en un hospital en Inglaterra y su amigo Hardy fue a visitarlo. Hardy comentó que había llegado al hospital en un taxi de matrícula N y esperaba que éste no fuese un mal presagio, ya que N era un número poco interesante. Ramanujan no estuvo de acuerdo ya que inmediatamente dijo que N tiene una propiedad muy especial: N es el menor entero positivo que puede descomponerse de dos maneras distintas como suma de dos cubos.

El objetivo de esta sección es averiguar la matrícula del taxi que llevó a Hardy a visitar a Ramanujan.

Ejercicio 6.1 *Definir la relación* `es_cubo(+N)` *que se verifique si* `N` *es el cubo de un entero. Por ejemplo,*

```
?- es_cubo_1(1000).
Yes
?- es_cubo_1(1001).
No
```

Solución: Presentaremos distintas definiciones y comentaremos su eficiencia.

La primera solución realiza una búsqueda desde 1 hasta N.

```
es_cubo_1(N) :-
   between(1,N,X),
   N is X*X*X.
```

La segunda solución realiza una búsqueda desde 1 hasta $\sqrt[3]{N}$.

```
es_cubo_2(N) :-
   Cota is round(N^(1/3)),
   between(1,Cota,X),
   N is X*X*X.
```

La tercera solución utiliza predicados aritméticos predefinidos.

```
es_cubo_3(N) :-
   N =:= round(N ** (1/3)) ** 3.
```

Para comparar la eficiencia realizamos los siguientes experimentos:

```
?- time(es_cubo_1(1000001)).
% 1,000,003 inferences, 1.23 CPU in 1.25 seconds (99% CPU, 813011 Lips)
No
?- time(es_cubo_2(1000001)).
% 103 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
No
?- time(es_cubo_3(1000001)).
% 3 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
No
```

Se observa que la más eficiente es la tercera definición. La segunda se aproxima a la segunda. La primera es muy ineficiente. En lo que sigue adoptaremos la tercera como definicón de `es_cubo`.

```
es_cubo(N) :-
   es_cubo_3(N).
```

Ejercicio 6.2 *Definir la relación* `descompone(+N,-X,-Y)` *que se verifique si* `X` *e* `Y` *son dos cubos cuya suma es* `N` *y, además,* `X` *es menor o igual que* `Y`*. Por ejemplo,*

```
?- descompone(1008,X,Y).
X = 8
Y = 1000 ;
No
```

Solución: Presentaremos distintas definiciones y comentaremos su eficiencia.
La primera definición realiza una búsqueda en `X` e `Y`.

```
descompone_1(N,X,Y) :-
    between(1,N,X),
    between(1,N,Y),
    es_cubo(X),
    es_cubo(Y),
    X =< Y,
    N is X+Y.
```

La segunda definición realiza la búsqueda en `X` y determina `Y`.

```
descompone_2(N,X,Y) :-
    between(1,N,X),
    es_cubo(X),
    Y is N - X,
    X =< Y,
    es_cubo(Y).
```

La tercera definición realiza una búsqueda acotada.

```
descompone_3(N,X,Y) :-
    Cota is round((N/2)^(1/3)),
    between(1,Cota,M),
    X is M*M*M,
    Y is N-X,
    X =< Y,
    es_cubo(Y).
```

Para comparar la eficiencia realizamos los siguientes experimentos:

```
?- time(descompone_1(1707,X,Y)).
% 11,732,455 inferences, 7.89 CPU in 7.91 seconds (100% CPU, 1487003 Lips)
No
?- time(descompone_2(1707,X,Y)).
% 6,890 inferences, 0.01 CPU in 0.01 seconds (112% CPU, 689000 Lips)
No
?- time(descompone_3(1707,X,Y)).
% 66 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
No

?- time(descompone_1(2331,X,Y)).
% 9,410,498 inferences, 6.31 CPU in 6.33 seconds (100% CPU, 1491363 Lips)
X = 1000
Y = 1331
Yes
?- time(descompone_2(2331,X,Y)).
% 4,062 inferences, 0.01 CPU in 0.01 seconds (189% CPU, 406200 Lips)
X = 1000
Y = 1331
Yes
?- time(descompone_3(2331,X,Y)).
% 73 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
X = 1000
Y = 1331
Yes
```

Se observa que la más eficiente es la tercera definición. En lo que sigue adoptaremos la tercera como definición de `descompone`.

```
descompone(N,X,Y) :-
   descompone_3(N,X,Y).
```

Ejercicio 6.3 *Definir la relación* `ramanujan(+N)` *que se verifique si* `N` *puede descomponerse en suma de dos cubos* exactamente *de dos maneras distintas.*

Solución: La definición de `ramanujan` es

```
ramanujan(N) :-
    setof(par(X,Y),descompone(N,X,Y),[_,_]).
```

Ejercicio 6.4 *Definir la relación* `hardy(-N)` *que se verifique si* `N` *es el menor entero positivo que satisface el predicado* `ramanujan` *anterior. ¿Cuál es la la matrícula del taxi que llevó a Hardy a visitar a Ramanujan?*

Solución: La definición de `hardy` es

```
hardy(N) :-
    hardy_aux(N,1).
hardy_aux(N,N) :-
    ramanujan(N),
    !.
hardy_aux(N,M) :-
    M1 is M+1,
    hardy_aux(N,M1).
```

La matrícula del taxi que llevó a Hardy a visitar a Ramanujan se calcula mediante la siguiente consulta

```
?- hardy(N).
N = 1729
```

Por tanto, la matrícula del taxi es 1729.

6.2. Subconjuntos de suma dada

Ejercicio 6.5 *Definir la relación* `subconjunto_suma(+L1,+N,?L2)` *que se verifique si* `L2` *es un subconjunto de* `L1` *tal que la suma de los elementos de* `L2` *es* `N`*. Por ejemplo,*

```
?- subconjunto_suma([10,7,3,4,2,1],7,L).
L = [7] ;
L = [3, 4] ;
L = [4, 2, 1] ;
No
?- subconjunto_suma([1,2,3],0,L).
L = [] ;
No
```

Solución: Presentaremos cuatro definiciones y comparemos su eficiencia.
La primera definición usa la relación `subconjunto` definida en el ejercicio 1.17 (página 19).

```
subconjunto_suma_1(L1,N,L2) :-
   subconjunto(L1,L2),
   sumlist(L2,N).
```

En la segunda definición se adapta la definición de `subconjunto` teniendo en cuenta la suma de sus elementos.

```
subconjunto_suma_2([],0,[]).
subconjunto_suma_2([X|L1],N,[X|L2]) :-
   N >= X,
   N1 is N-X,
   subconjunto_suma_2(L1,N1,L2).
subconjunto_suma_2([_|L1],N,L2) :-
   subconjunto_suma_2(L1,N,L2).
```

En la tercera definición se define de forma dinámica la relación `subconjuntos(+L1,+N.-S)` que se verifica si `S` es la lista de subconjuntos de `S` tales que la suma de sus elementos es `N`.

```
:- dynamic subconjuntos_suma_calculados_3/3.

subconjunto_suma_3(L1,N,L2) :-
   subconjuntos_suma_calculados_3(L1,N,S), !,
   member(L2,S).
subconjunto_suma_3(L1,N,L2) :-
   findall(L,subconjunto_suma_3_aux(L1,N,L),S),
   asserta(subconjuntos_suma_calculados_3(L1,N,S)),
   member(L2,S).

subconjunto_suma_3_aux([],0,[]).
subconjunto_suma_3_aux([X|L1],N,[X|L2]) :-
   N >= X,
   N1 is N-X,
   subconjunto_suma_3(L1,N1,L2).
subconjunto_suma_3_aux([_|L1],N,L2) :-
   subconjunto_suma_3(L1,N,L2).
```

La cuarta definición es una variación de la anterior de forma que se intercambian los dos primeros argumentos de `subconjuntos_suma_calculados` a fin de que el argumento sobre el que se indexa sea `N`.

```
:- dynamic subconjuntos_suma_calculados_4/3.

subconjunto_suma_4(L1,N,L2) :-
   subconjuntos_suma_calculados_4(N,L1,S), !,
   member(L2,S).
subconjunto_suma_4(L1,N,L2) :-
   findall(L,subconjunto_suma_4_aux(L1,N,L),S),
   asserta(subconjuntos_suma_calculados_4(N,L1,S)),
   member(L2,S).

subconjunto_suma_4_aux([],0,[]).
```

```
subconjunto_suma_4_aux([X|L1],N,[X|L2]) :-
   N >= X,
   N1 is N-X,
   subconjunto_suma_4(L1,N1,L2).
subconjunto_suma_4_aux([_|L1],N,L2) :-
   subconjunto_suma_4(L1,N,L2).
```

Para comparar la eficiencia se realizan los siguientes experimentos.

```
?- numlist(1,20,_L1), time(findall(L,subconjunto_suma_1(_L1,10,L),_S)).
% 25,165,844 inferences, 9.09 CPU in 9.19 seconds

?- numlist(1,20,_L1), time(findall(L,subconjunto_suma_2(_L1,10,L),_S)).
% 1,974 inferences, 0.00 CPU in 0.00 seconds

?- numlist(1,20,_L1), time(findall(L,subconjunto_suma_3(_L1,10,L),_S)).
% 3,242 inferences, 0.01 CPU in 0.01 seconds

?- numlist(1,20,_L1), time(findall(L,subconjunto_suma_4(_L1,10,L),_S)).
% 3,242 inferences, 0.01 CPU in 0.01 seconds


?- numlist(1,140,_L1), time(findall(L,subconjunto_suma_2(_L1,70,L),_S)).
% 104,855,949 inferences, 47.90 CPU in 58.77 seconds

?- numlist(1,140,_L1), time(findall(L,subconjunto_suma_3(_L1,70,L),_S)).
% 593,122 inferences, 18.91 CPU in 21.54 seconds

?- numlist(1,140,_L1), time(findall(L,subconjunto_suma_4(_L1,70,L),_S)).
% 593,158 inferences, 1.96 CPU in 2.18 seconds
```

Se observa que la más eficiente es la cuarta definición.

6.3. Coloreado de mapas

Ejercicio 6.6 *Un mapa puede representarse mediante la relación* `mapa(N,L)` *donde* `N` *es el nombre del mapa y* `L` *es la lista de los pares formados por cada una de las regiones del mapa y la lista de sus regiones vecinas. Por ejemplo, los mapas de la figura 6.1 se pueden representar por*

```
mapa(ejemplo_1,
     [a-[b,c,d], b-[a,d,e], c-[a,d,f], d-[a,b,c,e,f,g],
      e-[b,d,g], f-[c,d,g], g-[d,e,f]]).
mapa(ejemplo_2,
     [a-[b,e,k],   b-[a,c,e,k], c-[b,d,f,k], d-[c,f,k], e-[a,b,g,k],
      f-[c,d,h,k], g-[e,i,k],   h-[f,j,k],   i-[g,j,k], j-[i,h,k],
      k-[a,b,c,d,e,f,g,h,i,j]]).
```

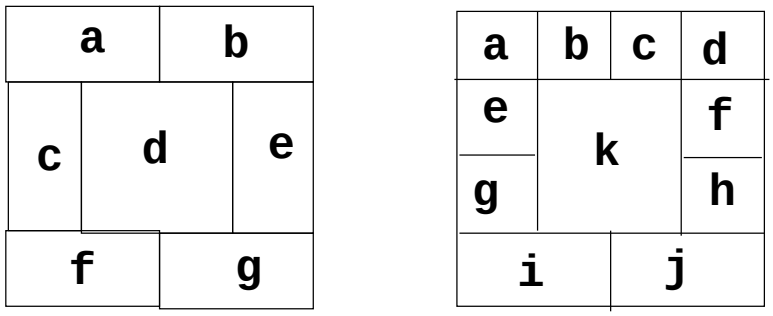


Figura 6.1: Ejemplos de mapas

Definir la relación `coloración(+M,+LC,-S)` *que se verifique si* `S` *es una lista de pares formados por una región del mapa* `M` *y uno de los colores de la lista de colores* `LC` *tal que las regiones vecinas tengan colores distintos. Por ejemplo,*

```
?- coloración(ejemplo_1,[1,2,3],S).
S = [a-1, b-2, c-2, d-3, e-1, f-1, g-2]
```

¿Qué número de colores se necesitan para colorear el segundo mapa? ¿De cuántas formas distintas puede colorearse con dicho número?

Solución: Presentamos dos definiciones y comparamos su eficiencia.
La primera definición de `coloración` es por generación y prueba.

```
coloración_1(M,LC,S) :-
   mapa(M,L),
   coloración_1_aux(L,LC,S).

coloración_1_aux([],_,[]).
coloración_1_aux([R-V|L],LC,[R-C|S]) :-
   member(C,LC),
   coloración_1_aux(L,LC,S),
   not((member(R1,V), member(R1-C,S))).
```

En la segunda definición de `coloración` se usa un acumulador y se adelanta las pruebas.

```
coloración_2(M,LC,S) :-
   mapa(M,L),
   coloración_2_aux(L,LC,[],S).

coloración_2_aux([],_,S,S).
coloración_2_aux([R-V|L],LC,A,S) :-
   member(C,LC),
   not((member(R1,V), member(R1-C,A))),
   coloración_2_aux(L,LC,[R-C|A],S).
```

Para comparar las dos definiciones usaremos el segundo mapa.

```
?- time(coloración_1(ejemplo_2,[1,2,3,4],S)).
% 16,705,318 inferences, 6.18 CPU in 6.20 seconds (100% CPU, 2703126 Lips)
S = [a-1, b-2, c-1, d-2, e-3, f-3, g-1, h-1, i-2, j-3, k-4]

?- time(coloración_2(ejemplo_2,[1,2,3,4],S)).
% 546 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
S = [k-4, j-3, i-2, h-1, g-1, f-3, e-3, d-2, c-1, b-2, a-1]
```

Se observa que la segunda es más eficiente.

En el ejemplo anterior se observa que se puede colorear el segundo mapa con 4 colores. Veamos si puede colorearse con 3 colores.

```
?- coloración_2(ejemplo_2,[1,2,3],S).
No
```

Por tanto, para colorear el segundo mapa se necesitan 4 colores.

El número `N` de formas distintas de colorear el segundo mapa se calcula mediante la siguiente consulta.

```
?- findall(_S,coloración_2(ejemplo_2,[1,2,3,4],_S),_L), length(_L,N).
N = 1032
```

Capítulo 7

Aplicaciones de programación declarativa

7.1. Formación de grupos minimales de asignaturas compatibles

Mediante la relación `lista_de_clase(C,A,L)` se representa la información de los alumnos según los cursos y asignaturas, de forma que `C` es el curso, `A` es la asignatura y `L` es la lista de los alumnos de dicha asignatura. A lo largo del ejercicio vamos a usar como ejemplo la siguiente información.

```
lista_de_clase(c1,asig1,[a1,a5]).
lista_de_clase(c1,asig2,[a1,a2,a3]).
lista_de_clase(c1,asig3,[a1,a3]).
lista_de_clase(c1,asig4,[a2,a4,a6,a8]).
lista_de_clase(c1,asig5,[a2,a4,a5]).
lista_de_clase(c1,asig6,[a6,a7]).
lista_de_clase(c1,asig7,[a3,a7]).
lista_de_clase(c1,asig8,[a6,a7,a8]).
lista_de_clase(c2,asig9,[a9,a11]).
lista_de_clase(c2,asig10,[a10,a12]).
lista_de_clase(c2,asig11,[a10,a11]).
lista_de_clase(c2,asig12,[a9,a12]).
```

Ejercicio 7.1 *Definir la relación* `asignaturas(+C,-L)` *que se verifique si* `L` *es la lista de asignaturas del curso* `C`*. Por ejemplo,*

```
?- asignaturas(c1,L).
L = [asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8]
?- asignaturas(c2,L).
L = [asig9, asig10, asig11, asig12]
```

Solución: La definición de `asignaturas` es

```
asignaturas(C,L):-
   findall(X,lista_de_clase(C,X,_),L).
```

Ejercicio 7.2 *Definir la relación* `grupo_incompatible(+L)` *que se verifique si la lista de asignaturas* `L` *es incompatible (es decir, algún alumno está en las listas de clase de más de una asignatura de la lista* `L`*). Por ejemplo,*

```
?- grupo_incompatible([asig1,asig2]).
Yes
?- grupo_incompatible([asig1,asig4,asig7]).
No
```

Solución: La definición de `grupo_incompatible` es

```
grupo_incompatible(L):-
   select(X,L,R),
   member(Y,R),
   lista_de_clase(_,X,LX),
   lista_de_clase(_,Y,LY),
   member(A,LX),
   member(A,LY).
```

Ejercicio 7.3 *Definir la relación* `lista_incompatible(+L)` *que verifique si la lista de grupos de asignaturas* `L` *es incompatible (es decir, contiene algún grupo incompatible de asignaturas). Por ejemplo,*

```
?- lista_incompatible([[asig9, asig12], [asig11, asig10]]).
Yes
?- lista_incompatible([[asig11, asig12], [asig9, asig10]]).
No
```

Solución: La definición de `lista_incompatible` es

```
lista_incompatible(P) :-
   member(G,P),
   grupo_incompatible(G).
```

Ejercicio 7.4 *Definir la relación* `extensión(+L1,+X,-L2)` *que se verifique si* `L2` *es la lista obtenida añadiendo* `X` *como primer elemento de un elemento de* `L1` *o* `L2` *es la lista obtenida añadiendo* `[X]` *a* `L1`*. Por ejemplo,*

```
?- extensión([[a], [b, c]],d,E).
E = [[d, a], [b, c]] ;
E = [[a], [d, b, c]] ;
E = [[a], [b, c], [d]] ;
No
```

Solución: La definición de `extensión` es

```
extensión([],X,[[X]]).
extensión([L|L1],X,[[X|L]|L1]).
extensión([L|L1],X,[L|L2]) :-
   extensión(L1,X,L2).
```

Ejercicio 7.5 *Definir la relación* `partición(+L,-P)` *que se verifique si* `P` *es una partición de la lista* `L` *(es decir, un conjunto obtenido distribuyendo los elementos de* `L` *en conjuntos no vacíos y sin elementos comunes). Por ejemplo,*

```
?- partición([a,b,c],P).
P = [[a, b, c]] ;
P = [[b, c], [a]] ;
P = [[a, c], [b]] ;
P = [[c], [a, b]] ;
P = [[c], [b], [a]] ;
No
```

Solución: La definición de `partición` es

```
partición([],[]).
partición([X|L1],L2) :-
   partición(L1,L3),
   extensión(L3,X,L2).
```

Ejercicio 7.6 *Definir la relación* `agrupación_compatible_de_asignaturas(+C,-P)` *que se verifique si* `P` *es una partición compatible de las asignaturas del curso* `C`*. Por ejemplo,*

```
?- agrupación_compatible_de_asignaturas(c2,P).
P = [[asig11, asig12], [asig9, asig10]] ;
P = [[asig11, asig12], [asig10], [asig9]] ;
Yes
```

Solución: La definición de `agrupación_compatible_de_asignaturas` es

```
agrupación_compatible_de_asignaturas(C,P) :-
   asignaturas(C,L),
   partición(L,P),
   not(lista_incompatible(P)).
```

Ejercicio 7.7 *Definir la relación* `agrupación_compatible_minimal(+C,-P)` *que se verifique si* `P` *es una partición compatible de las asignaturas del curso* `C` *con el menor número posible de grupos de asignaturas. Por ejemplo,*

```
?- agrupación_compatible_minimal(c2,P).
P = [[asig11, asig12], [asig9, asig10]] ;
No
```

Calcular las agrupaciones compatibles minimales del curso `c1`*.*

Solución: La definición `agrupación_compatible_minimal` es

```
agrupación_compatible_minimal(C,P) :-
   agrupación_compatible_de_asignaturas(C,P),
   length(P,N),
   not((agrupación_compatible_de_asignaturas(C,P1),length(P1,M),M < N)).
```

Las agrupaciones compatibles minimales del curso `c1` se calculan mediante la consulta

```
?- agrupación_compatible_minimal(c1,P).
P = [[asig3, asig5, asig8], [asig1, asig4, asig7], [asig2, asig6]] ;
P = [[asig2, asig8], [asig1, asig4, asig7], [asig3, asig5, asig6]] ;
No
```

7.2. Simulación de una calculadora básica

El objetivo de los siguientes ejercicios es la simulación de una calculadora básica. Para ello consideraremos que en cada momento la calculadora se encuentra en un determinado estado caracterizado por una lista con cuatro elementos `[UCE,UTA,UOA,VIM]` donde

- `UCE` es el último cálculo efectuado,
- `UTA` es la última tecla activada,
- `UOA` es el último operador activado y
- `VIM` es el valor impreso.

El estado inicial es `[0,=,=,0]` y está definido por

```
estado_inicial([0,=,=,0]).
```

Las acciones posibles son pulsar un dígito, una operación aritmética o la de resultado y están definidas por

```
acción(X) :- es_dígito(X).
acción(X) :- es_operación(X).
acción(X) :- es_resultado(X).

es_dígito(0).
es_dígito(1).
es_dígito(2).
es_dígito(3).
es_dígito(4).
es_dígito(5).
es_dígito(6).
es_dígito(7).
es_dígito(8).
es_dígito(9).

es_operación(+).
es_operación(-).
es_operación(*).
es_operación(/).

es_resultado(=).
```

En la siguiente tabla se muestran los estados de la calculadora correspondientes a las acciones indicadas en la última columna

estado	tecla
(0, =, =, 0)	3
(0, 3, =, 3)	+
(3, +, +, 3)	2
(3, 2, +, 2)	1
(3, 1, +, 21)	*
(24, *, *, 24)	2
(24, 2, *, 2)	=
(48, =, =, 48)	

Es decir, si se parte del estado inicial y se realizan las acciones

```
3 + 2 1 * 2 =
```

se obtiene como resultado el número `48`.

Ejercicio 7.8 *Definir la relación* `transición(+E1,+X,?E2)` *que se verifique si* `E2` *es el estado obtenido aplicando la acción* `X` *al estado* `E1`*; es decir, si* `E1` *es* `[UCE,UTA,UOA,VIM]`*, entonces*

- *Si* `X` *es un dígito, entonces*
  - *si* `UTA` *es un dígito,* `E2` *es* `[UCE,X,UOA,10*VIM+X]`*;*
  - *en otro caso,* `E2` *es* `[UCE,X,UOA,X]`*.*
- *Si* `X` *no es un dígito, entonces*
  - *si* `UOA` *es una operación,* `E2` *es* `[UOA(UCE,VIM),X,X,UOA(UCE,VIM)]`
  - *en otro caso,* `E2` *es* `[VIM,X,X,VIM]`*.*

Por ejemplo,

```
?- estado_inicial(E1),
   transición(E1,3,E2),
   transición(E2,+,E3),
   transición(E3,2,E4),
   transición(E4,1,E5),
   transición(E5,*,E6),
   transición(E6,2,E7),
   transición(E7,=,E8).
E1 = [0, =, =, 0]
E2 = [0, 3, =, 3]
E3 = [3, +, +, 3]
E4 = [3, 2, +, 2]
E5 = [3, 1, +, 21]
E6 = [24, *, *, 24]
E7 = [24, 2, *, 2]
E8 = [48, =, =, 48]
```

Solución: Presentamos tres definiciones.
La primera definición de `transición` es

```
transición_1([UCE,UTA,UOA,VIM],X,E) :-
   ( es_dígito(X) ->
     ( es_dígito(UTA) ->
       Y is 10*VIM+X,
       E = [UCE,X,UOA,Y]
     ; % \+ es_dígito(UTA) ->
       E = [UCE,X,UOA,X] )
   ; % \+ es_dígito(X) ->
     ( es_operación(UOA) ->
       T =.. [UOA,UCE,VIM],
       Y is T,
       E = [Y,X,X,Y]
     ; % \+ es_operación(UOA) ->
       E = [VIM,X,X,VIM] )).
```

La segunda definición de `transición` es

```
transición_2([UCE,UTA,UOA,VIM],X,E) :-
   ( es_dígito(X), es_dígito(UTA) ->
     Y is 10*VIM+X,
     E = [UCE,X,UOA,Y]
   ; es_dígito(X), \+ es_dígito(UTA) ->
     E = [UCE,X,UOA,X]
   ; \+ es_dígito(X), es_operación(UOA) ->
     T =.. [UOA,UCE,VIM],
     Y is T,
     E = [Y,X,X,Y]
   ; \+ es_dígito(X), es_resultado(UOA) ->
     E = [VIM,X,X,VIM] ).
```

La tercera definición de `transición` es

```
transición_3([UCE,UTA,UOA,VIM],X,[UCE,X,UOA,Y]) :-
   es_dígito(X),
   es_dígito(UTA),
   Y is 10*VIM+X.
transición_3([UCE,UTA,UOA,_VIM],X,[UCE,X,UOA,X]) :-
   es_dígito(X),
   \+ es_dígito(UTA).
transición_3([UCE,_UTA,UOA,VIM],X,[Y,X,X,Y]) :-
   \+ es_dígito(X),
   es_operación(UOA),
   T =.. [UOA,UCE,VIM],
   Y is T.
```

```
transición_3([_UCE,_UTA,=,VIM],X,[VIM,X,X,VIM]) :-
   \+ es_dígito(X).
```

En lo que sige usaremos la primera.

```
transición(E1,A,E2) :-
   transición_1(E1,A,E2).
```

Ejercicio 7.9 *Definir la relación* `transiciones(+E1,+L,?E2)` *que se verifique si* `E2` *es el estado obtenido aplicando las acciones de la lista* `L` *al estado* `E1`*. Por ejemplo,*

```
?- estado_inicial(_E1), transiciones(_E1,[3,+,2,1,*,2,=],E2).
E2 = [48, =, =, 48]
```

Solución: La definición de `transiciones` es

```
transiciones(E,[],E).
transiciones(E1,[X|L],E3) :-
   transición(E1,X,E2),
   transiciones(E2,L,E3).
```

Ejercicio 7.10 *Definir la relación* `acciones(?L)` *que se verifique si* `L` *es una lista cuyos elementos son acciones. Por ejemplo,*

```
?- acciones([2,+,3,7]).
Yes
?- acciones([2,+,37]).
No
```

Usarlo para calcular el número de posibles listas de acciones de longitud 3.

Solución: La definición de `acciones` es

```
acciones([]).
acciones([X|L]) :-
   acción(X),
   acciones(L).
```

El número `N` de listas de acciones de longitud 3 se calcula mediante la consulta

```
?- findall(_L,(length(_L,3), acciones(_L)),_LAL3), length(_LAL3,N).
N = 3375
```

Ejercicio 7.11 *Para realizar una operación en la calculadora no todas las combinaciones de teclas (acciones) son válidas. Por ejemplo, no podemos teclear dos operaciones consecutivas o dividir por cero. La siguiente relación define las acciones válidas*

```
acciones_válidas(L) :-
   acciones(L),
   empieza_por_dígito(L),
   not(tiene_operaciones_consecutivas(L)),
   not(tiene_resultado_intermedio(L)),
   not(divide_por_cero(L)),
   termina_en_dígito_y_resultado(L).
```

En los apartados de este ejercicio se definen de las relaciones auxiliares.

1. *Definir la relación* `empieza_por_dígito(?L)` *que se verifique si el primer elemento de la lista* `L` *es un dígito.*

2. *Definir la relación* `tiene_operaciones_consecutivas(?L)` *que se verifique si la lista* `L` *contiene dos operaciones consecutivas.*

3. *Definir la relación* `tiene_resultado_intermedio(?L)` *que se verifique si la lista* `L` *contiene el símbolo = en una posición que no es la última.*

4. *Definir la relación* `divide_por_cero(?L)` *que se verifique si en la lista* `L` *aparecen de manera consecutiva el símbolo / y un cero.*

5. *Definir la relación* `termina_en_dígito_y_resultado(?L)` *que se verifique si en la lista* `L` *los últimos elementos son un dígito y el símbolo =.*

Solución:

```
empieza_por_dígito([X|_L]) :-
   es_dígito(X).

tiene_operaciones_consecutivas(L) :-
   append(_A,[X,Y|_B],L),
   es_operación(X),
   es_operación(Y).

tiene_resultado_intermedio(L) :-
   append(_A,[=,_Y|_B],L).

divide_por_cero(L) :-
   append(_A,[/,0|_B],L).

termina_en_dígito_y_resultado(L) :-
   reverse(L,[=,X|_]),
   es_dígito(X).
```

Ejercicio 7.12 *Calcular el número de posibles listas de acciones válidas de longitud 3.*

Solución: El número `N` de listas de acciones válidad de longitud 3 se calcula mediante la consulta

```
?- findall(_L,(length(_L,3), acciones_válidas(_L)),_LAL3), length(_LAL3,N).
N = 100
```

Ejercicio 7.13 *Definir la relación* `cálculo(+N,+M,-L)` *que se verifique si* `L` *es una lista de* `M` *acciones válidas que aplicadas al estado inicial da como resultado el número* `N`*. Por ejemplo,*

```
?- cálculo(5,2,L).
L = [5, =] ;
No
?- cálculo(5,3,L).
L = [0, 5, =] ;
No
?- cálculo(5,4,L).
L = [0, 0, 5, =] ;
L = [0, +, 5, =] ;
L = [1, +, 4, =] ;
L = [1, *, 5, =] ;
L = [2, +, 3, =]
Yes
```

Solución:

```
cálculo(N,M,L) :-
   estado_inicial(E1),
   length(L,M),
   acciones_válidas(L),
   transiciones(E1,L,[N,=,=,N]).
```

7.3. Problema de las subastas

Ejercicio 7.14 *En una subasta se hacen distintas ofertas. Cada oferta incluye un lote de productos y un precio por dicho lote. Las ofertas realizadas se representan mediante la relación* `oferta(O,L,P)` *que se verifica si* `O` *es una oferta por el lote* `L` *con un coste* `P`*. Por ejemplo,* `oferta(a,[1,2,3],30)` *representa la oferta* `a` *en la que se puja por el lote compuesto por los objetos* `1`*,* `2` *y* `3` *por un valor de* `30` *euros.*

Para la aceptación de las ofertas se observan las siguientes reglas:

- *No puede aceptar dos ofertas que contienen un mismo objeto en sus lotes.*
- *Se prefieren las ofertas de mayor ganancia.*

Definir la relación `aceptada(-L)` *que se verifique si* `L` *es una lista de ofertas aceptadas. Por ejemplo, si las ofertas realizadas se definen por*

```
oferta(a,[1,2,3],30).
oferta(b,[1,2,3],20).
oferta(c,[4],20).
oferta(d,[2,4],20).
oferta(e,[1,2],20).
```

entonces,

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

Solución: La definición de `aceptada` es

```
aceptada(L) :-
   aceptable(L),
   ganancia(L,G),
   not((aceptable(L1), ganancia(L1,G1), G1 > G)).
```

La relación `aceptable(?L)` se verifica si `L` es una lista de ofertas aceptable; es decir, una lista de ofertas que no contienen objetos comunes en sus lotes. Por ejemplo, con la definición anterior de `ofertas/3`,

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

```
aceptable(L) :-
   lista_de_ofertas(L1),
   subconjunto(L,L1),
   es_aceptable(L).
```

La relación `lista_de_ofertas(-L)` se verifica si `L` es la lista de todas las ofertas. Por ejemplo, con la definición anterior de `ofertas/3`,

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

```
lista_de_ofertas(L) :-
   findall(O,oferta(O,_,_),L).
```

La relación `subconjunto(?L1,+L2)` se verifica si `L1` es un subconjunto de `L2`. Por ejemplo,

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

```
subconjunto([],[]).
subconjunto([X|L1],[X|L2]) :-
   subconjunto(L1,L2).
subconjunto(L1,[_|L2]) :-
   subconjunto(L1,L2).
```

La relación `es_aceptable(+L)` se verifica si la lista de ofertas `L` es aceptable; es decir, no contiene ofertas con objetos comunes en sus lotes. Por ejemplo, con la definición anterior de `ofertas/3`,

```
?- es_aceptable([c,d]).
No
?- es_aceptable([c,e]).
Yes
```

```
es_aceptable(L) :-
   not(es_inaceptable(L)).
```

La relación `es_inaceptable(+L)` se verifica si `L` es una lista de ofertas inaceptable; es decir, contiene ofertas con objetos comunes en sus lotes. Por ejemplo, con la definición anterior de `ofertas/3`,

```
?- es_inaceptable([c,d]).
Yes
?- es_inaceptable([c,e]).
No
```

```
es_inaceptable(L) :-
   member(O1,L),
   member(O2,L),
   O1 \= O2,
   oferta(O1,L1,_),
   oferta(O2,L2,_),
   se_solapan(L1,L2).
```

La relación `se_solapan(+L1,+L2)` se verifica si `L1` y `L2` se solapan; es decir, tienen elementos comunes. Por ejemplo,

```
?- se_solapan([a,b,c],[d,b,e]).
Yes
?- se_solapan([a,b,c],[d,e]).
No
```

```
se_solapan(L1,L2) :-
   member(X,L1),
   member(X,L2).
```

La relación `ganancia(+L,-G)` se verifica si la ganancia de la lista de ofertas L es `G`. Por ejemplo, con la definición anterior de `ofertas/3`,

```
?- ganancia([a,c],G).
G = 50
```

```
ganancia([],0).
ganancia([O|L],G) :-
   oferta(O,_,G1),
   ganancia(L,G2),
   G is G1+G2.
```

Bibliografía

[1] J. A. Alonso. Introducción a la programación lógica con Prolog, 2006. En `http://www.cs.us.es/~jalonso/publicaciones/2006-int_prolog.pdf`.

[2] J. A. Alonso. Temas de "Programación declarativa" (2005-06), 2006. En `http://www.cs.us.es/~jalonso/publicaciones/2005-06-PD-temas.pdf`.

[3] K. R. Apt. *From logic programming to Prolog*. Prentice Hall, 1996.

[4] P. Blackburn, J. Bos, and K. Striegnitz. Learn Prolog Now!, 2001. En Bib y en Red.

[5] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison–Wesley, 3 edition, 2001.

[6] W. F. Clocksin. *Clause and Effect (Prolog Programming for the Working Programmer)*. Springer–Verlag, 1997.

[7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer–Verlag, 4 edition, 1994.

[8] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, 1997.

[9] Y. Deeville. *Logic Programmng (Systematic Program Development)*. Addison–Wesley, 1990.

[10] J. P. Delahaye. *Cours de Prolog avec Turbo Prolog (Eléments fondamentaux)*. Eyrolles, 1988.

[11] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog*. 2 edition, 2000. En `http://www.ida.liu.se/~ulfni/lpp`.

[12] R. A. O'Keefe. *The Cratf of Prolog*. The MIT Press, 1990.

[13] P. Ross. *Advanced Prolog: Techniques and Examples*. Addison-Wesley, 1989.

[14] P. Schnupp, D. Merritt, and S. S. Muchnick. *Adventure in Prolog*. Springer–Verlag, 1990.

[15] L. Sterling and E. Shapiro. *L'art de Prolog*. Masson, 1990.

[16] T. Van Le. *Techniques of Prolog Programming (with implementation of logical negation and quantified goals)*. John Wiley, 1993.