1. Primera parte (en el aula de examen)

Ejercicio 1 (1.5 puntos) *Se considera la función*

```
pares :: [a] -> [(a, a)]
```

tal que `(pares xs)` *es la lista de los pares de elementos consecutivos en la lista* `xs`*. Por ejemplo,*

```
pares [2,1,3,7]  ⇝  [(2,1),(1,3),(3,7)]
```

Se pide:

1. *Escribir una definición **recursiva** de la función* `pares`*.*
2. *Escribir una definición **no recursiva** de la función* `pares`*.*

Solución: Una definición recursiva es

```
paresR [] = []
paresR [_] = []
paresR (x:y:xs) = (x,y):paresR (y:xs)
```

Una definición no recursiva es

```
paresNR xs = zip xs (tail xs)
```

Ejercicio 2 (1.5 puntos) *Se considera la función*

```
posiciones :: Eq a => [a] -> a -> [Int]
```

tal que `(posiciones xs y)` *es la lista de las posiciones de* `y` *en la lista* `xs`*. Por ejemplo,*

```
posiciones "Salamanca" 'a'  ⇝  [1,3,5,8]
```

Se pide

1. *Escribir una definición **no recursiva** de la función* `posiciones`*.*
2. *Escribir una definición **recursiva** de la función* `posiciones`*.*

Solución: Una definición no recursiva es

```
posicionesNR xs y = [i | (i,x) <- zip [0..] xs, x==y]
```

Una definición recursiva es

```
posicionesR xs y = posicionesRaux xs y 0

posicionesRaux :: Eq a => [a] -> a -> Int -> [Int]
posicionesRaux [] y i = []
posicionesRaux (x:xs) y i
    | x == y    = i : posicionesRaux xs y (i+1)
    | otherwise = posicionesRaux xs y (i+1)
```

Ejercicio 3 (1 punto) *Definir la función*

```
permutaciones :: Eq a => [a] -> [[a]]
```

tal que `permutaciones l` *es la lista de las permutaciones de la lista* `l`*. Por ejemplo,*

```
Main> permutaciones [2,3]
[[2,3],[3,2]]
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Solución:

```
permutaciones [] = [[]]
permutaciones xs =
    [a:p | a <- xs, p <- permutaciones(xs \\ [a])]
```

Ejercicio 4 (1.5 puntos)

1. *Se considera el siguiente programa lógico:*

```
p([],[]).                          %R1
p([X|A],[X|B]) :- q(X),p(A,B).     %R2
p([X|A],B) :- no(q(X)),p(A,B).     %R3
q(a).                              %R4
no(P) :- P,!,fail.                 %R5
no(P).                             %R6
```

   Construir el árbol de resolución para el programa anterior y la pregunta:

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

2. *Determinar si al responder Prolog a las siguientes preguntas tiene éxito o falla. Si tiene éxito, escribir los valores que asigna a las variables. Si falla, explicar la razón del fallo.*
   a) `?- not((member(X,[1,a,2]),not(number(X)))).`
   b) `?- name(mundo,[A|B]),append(L,L,[A|B]).`

Solución: El árbol se muestra en la figura 1

```
?- not((member(X,[1,a,2]),not(number(X)))).
No

?- name(mundo,[A|B]),append(L,L,[A|B]).
No
```

Ejercicio 5 (1.5 puntos) *El predicado* `almenos(+N,+P,+L)` *se verifica si al menos* `N` *elementos de la lista* `L` *satisfacen el predicado unario* `P`*. Por ejemplo,*

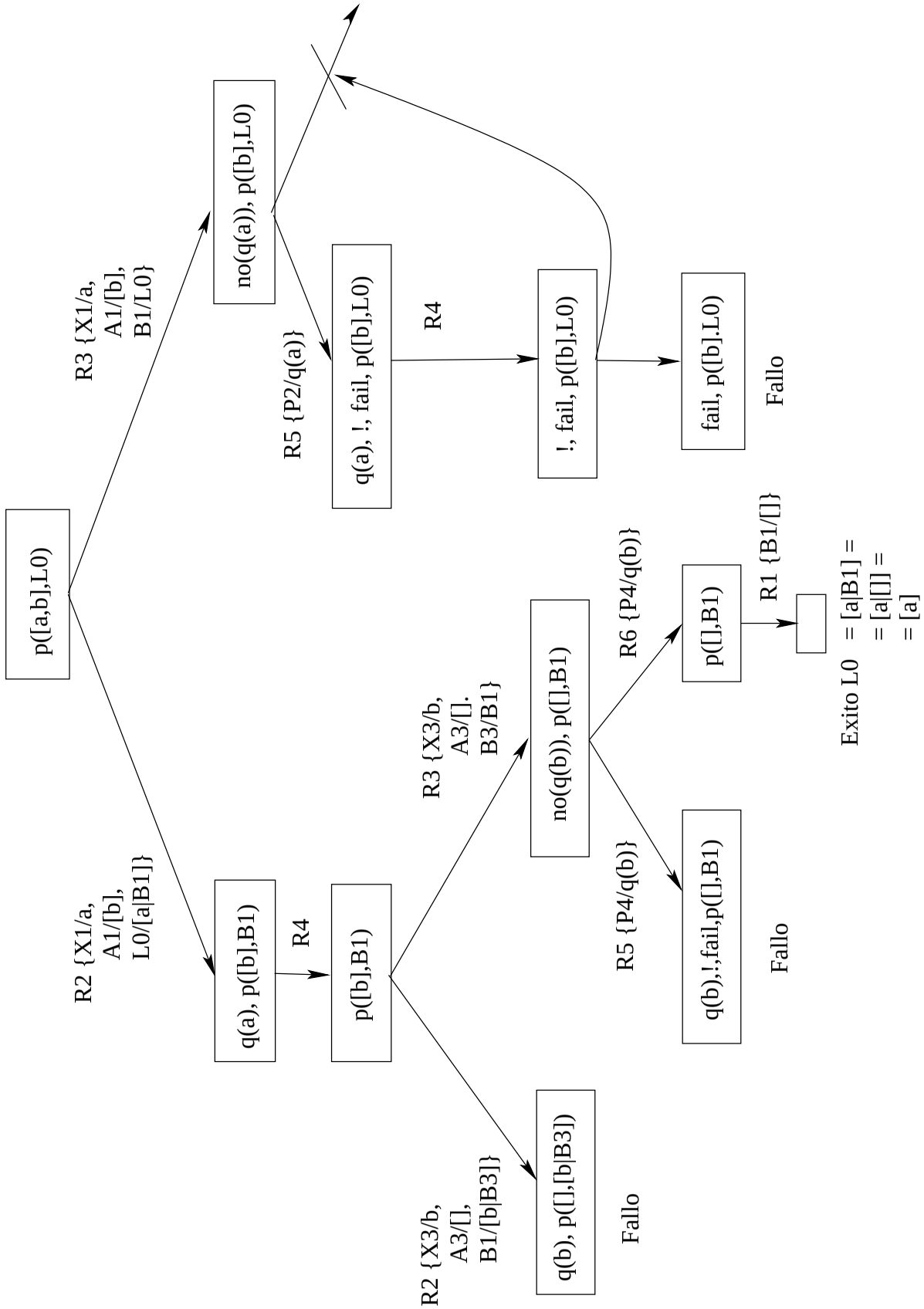

Figura 1: Árbol de resolución

```
?- almenos(3,number,[2,X,a,3,[4]]).
No
?- almenos(3,atomic,[2,X,a,3,[4]]).
Yes
```

Se pide

1. *Escribir una definición **recursiva** en Prolog del predicado* `almenos`.
2. *Escribir una definición **no recursiva** en Prolog del predicado* `almenos`.

Nota: Usar el predefinido `apply(+Term,+List)`.

Solución: Una definición recursiva es

```
almenos(0,_,_) :- !.
almenos(N,P,[X|L]) :-
   % N > 0,
   apply(P,[X]), !,
   M is N-1,
   almenos(M,P,L).
almenos(N,P,[_X|L]) :-
   % N > 0,
   % \+ apply(P,[_X]),
   almenos(N,P,L).
```

Una definición no recursiva es

```
almenos(N,P,L) :-
   findall(X,(member(X,L), apply(P,[X])),L1),
   length(L1,M),
   M >= N.
```

2. Segunda parte (en el aula de prácticas)

Ejercicio 6 (2 puntos) *Se considera la función*

```
busca :: Char -> Int -> Int -> [String] -> [String]
```

tal que `(busca x p n ps)` *es la lista de las palabras de la lista de palabras* `ps` *que tienen longitud* `n` *y posen la letra* `x` *en la posición* `p` *(comenzando en 0). Por ejemplo,*

```
Main> busca 'c' 1 7 ["ocaso", "ocupado", "casa", "ocupada"]
["ocupado","ocupada"]
```

Se pide

1. *Escribir una definición **no recursiva** de la función* `busca`.

2. *Escribir una definición **recursiva** de la función* `busca`.

Solución: Una definición no recursiva es

```
busca x p n ps =
    [y | y <- ps, length y == n, 0 <= p, p < length y, y !! p == x]
```

Una definición recursiva es

```
buscaRec x p n [] = []
buscaRec x p n (y:ps)
    | length y == n && 0 <= p && p < length y && y !! p == x
        = y : buscaRec x p n ps
    | otherwise
        = buscaRec x p n ps
```

Ejercicio 7 (1 punto) *Representaremos expresiones aritméticas en Prolog como sigue:*

Constantes	*1,2,3,...*
Variables	*a,b,c,...*
Suma	*E1 + E2*
Diferencia	*E1 - E2*

Una asignación de valores numéricos a una serie de variables se representará mediante una lista de términos `valor(V,N)`*; por ejemplo,*

```
[valor(a,1),valor(c,2),valor(b,-5)]
```

Definir en Prolog la relación `evaluar(+E,+L,-X)` *que se verifica si* `X` *es el resultado de evaluar la expresión* `E` *según la asignación* `L`*. Por ejemplo,*

```
?- evaluar(a+b-1+c,[valor(a,1),valor(c,2),valor(b,-5)],X).
X = -3 ;
No
?- evaluar(d+1,[valor(a,1),valor(b,2)],X).
No
```

Solución:

```
evaluar(E,_,E) :- number(E),!.
evaluar(E,L,X) :- atom(E),!,busca(E,L,X).
evaluar(E1+E2,L,X) :- evaluar(E1,L,X1),evaluar(E2,L,X2), X is X1 + X2.
evaluar(E1-E2,L,X) :- evaluar(E1,L,X1),evaluar(E2,L,X2), X is X1 - X2.

busca(E,L,X) :- memberchk(valor(E,X),L).
```