UNIVERSIDAD DE MÁLAGA
E.T.S.I. INFORMÁTICA

Programación Declarativa

(3^o de Ingeniería Técnica en Informática)
12 de Febrero de 2004

Alumno: _____ Grupo: _____

1. Prolog

Ejercicio 1

$(a)(2\ pts)$ Realiza el árbol de búsqueda para el objetivo `p(X,Y)` y el programa *Prolog*

```
p(X,Y):- q(a,X,Y).
q(X,Y,Y):- s(X),r(b,Y).
q(a,b,a).
q(X,X,Y):- s(Y),r(Y,X).
s(a).
s(b).
r(X,Y):-p(X,a).
```

SOLUCION:

$(b)(0{,}5\ pts)$ Teniendo en cuenta el comportamiento de *Prolog* ¿Qué soluciones proporcionaría para el objetivo anterior? Justifica tu respuesta.

`X/Y; X/b,Y/a; X/a,Y/a; y la última se repite indefinidamente`

Ejercicio 2

Dado el predicado

```
af([X],Ys,[X|Ys]).
af([X,Y|Xs],Ys,[X,Y|Zs]):- af(Xs,Ys,Zs).
```

$(a)(0{,}25\ pts)$ Calcular las respuestas de prolog para las llamadas siguientes

```
?- af([a,b,c],[d,e],Zs).
```

`Zs/[a,b,c,d,e]`

```
?- af([a,b],[b,c,e],Zs).
```

`No hay solución`

$(b)(0{,}75\ pts)$ Da la tabla de comportamiento del predicado `af/3` para los usos $(+,+,-), (-,+,+)$ y $(-,-,+)$.

SOLUCION:

$(+,+,-)$	Generador único	Genera una lista como concatenación de dos. La primera lista debe tener un número impar de elementos
$(-,+,+)$	Generador único	Genera en el primer argumento una lista tal que al concatenar con la segunda produce la tercera. Tiene éxito si la lista generada tiene un número impar de elementos
$(-,-,+)$	Generador acotado	Genera todos los pares de listas que al concatenar produce la tercera. Para tener éxito, la primera debe tener un número impar de elementos

Ejercicio 3

$(a)(1{,}25\ pts)$ Define el predicado `tresR(Xs,Yss)` de manera que dada la lista `Xs`, genere por reevaluación la lista `Yss` de todas las secuencias de tres elementos consecutivos de la lista `Xs`. Por ejemplo:

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

SOLUCION:

```
tresR([X,Y,Z|_Zs],[X,Y,Z]).
tresR([_X|Zs],Ts):-
        tresR(Zs,Ts).
```

$(b)(1{,}25\ pts)$ Define el predicado `tresT(Xs,Ys)` de manera que dada la lista `Xs`, genere la lista `Ys` que contiene a todas las secuencias de tres elementos consecutivos de `Xs`. Por ejemplo:

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

SOLUCION:

```
tresT([X,Y,Z|Zs],[[X,Y,Z]|Ts]):-
        tresT([Y,Z|Zs],Ts).
tresT([_X,_Y],[]).
tresT([_X],[]).
tresT([],[]).
```

Alumno: _____ Grupo: _____

Ejercicio 4

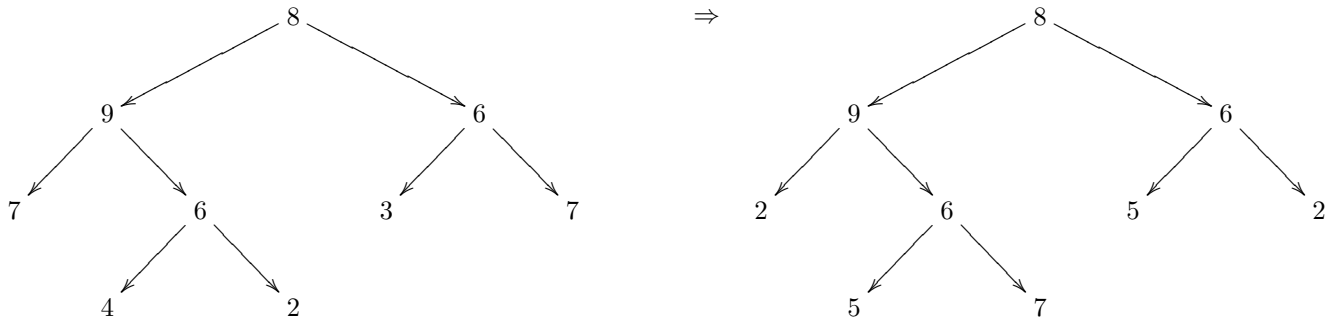
$(a)(1\ pt)$ Da el dominio de definición para árboles binarios con información entre 0 y 10 en cada nodo y hoja (no existe el árbol vacío).

SOLUCION:

```
anh(h(X)):-
    X >= 0,
    X =< 10.
```

```
anh(n(I,X,D)):-
      X >= 0,
      X =< 10,
      anh(I),
      anh(D).
```

$(b)(1{,}5\ pts)$ Define la función `mym/2` que tome un árbol binario con información entre 0 y 10 en nodos y hojas y sustituya cada hoja por la diferencia entre el máximo y mínimo que aparecen en el recorrido desde la raíz hasta dicha hoja (el dato que hay en la hoja interviene para el cálculo de ese máximo y mínimo). Un ejemplo de transformación es el siguiente:



SUGERENCIA.- Define un predicado auxiliar con dos argumentos acumuladores que almacenen el máximo y el mínimo del camino recorrido. Al principio, esos valores serán 0 y 10.

SOLUCION:

```
mym(As,NAs) :- mym(As,10,0,NAs).
mym(n(AI,X,AD),Mi,Ma,n(NAI,X,NAD)):-
        maximo(X,Ma,NMa),
        minimo(X,Mi,NMi),
        mym(AI,NMi,NMa,NAI),
        mym(AD,NMi,NMa,NAD).
mym(h(X),Mi,Ma,h(Z)):-
        maximo(X,Ma,NMa),
        minimo(X,Mi,NMi),
        Z is NMa-NMi.
```

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

$(c)(1{,}5\ pts)$ Definir el predicado `aplicaN(N,Ar,Br)` que aplica `N` veces consecutivas el predicado `mym` primero sobre el árbol `Ar` y después sobre las sucesivos árboles que se van obteniendo. `Br` es el árbol resultante de la última aplicación.

SOLUCION:

```
aplican(0,As,As).
aplican(N,As,NAs):-
        N>0,
        mym(As,IAs),
        N1 is N-1,
        aplican(N1,IAs,NAs).
```

UNIVERSIDAD DE MÁLAGA
E.T.S.I. INFORMÁTICA

Programación Declarativa
(3^o de Ingeniería Técnica en Informática)
12 de Febrero de 2004

Alumno: _____ Grupo: _____

2. Haskell

Ejercicio 1

(a)(1,5 pts) Dada la función:

```
k f xs = concat (map f xs)
```

Expresa la función `k` usando listas por comprensión dando su tipo.

```
k :: (a -> [b]) -> [a] -> [b]

k f xs = [  y  | x <- xs              ,   y <- f x         ]
```

(b)(1 pt) Define, usando el método que quieras (quick sort, inserción, merge, etc.), una función `sort` que ordena una lista. Da el tipo de dicha función y de las funciones auxiliares que utilices.

```
qs :: (Ord a) => [a] -> [a]
qs [] = []
qs (x:xs) = qs [y| y <- xs, y <= x] ++ [x] ++ qs [y| y <- xs, y > x]
```

(c)(1,5 pts) Una rotación de una lista se obtiene quitando cualquier número de elementos del comienzo de la lista y añadiéndolos al final. Define la función `rotaciones`, dando su tipo más general, que devuelve una lista con todas las posibles rotaciones de su argumento. Por ejemplo

`rotaciones [1,2,3]` $\Rightarrow$ `[ [1,2,3], [2,3,1], [3,1,2] ]`

```
rotaciones :: [a] -> [[a]]
rotaciones xs = rot (length xs) xs
 where
  rot 0     _         = []
  rot (n+1) ys@(x:xs) = ys : rot n (xs++[x])
```

(d)(1,5 pts) Define la función `filtra` que tome dos argumentos. El primero es una lista de elementos que llamaremos triviales. El segundo es una lista de listas. La función devolverá todas las listas del segundo argumento cuya cabeza no sea trivial. Si la lista no tiene cabeza, tampoco se debe incluir. Da el tipo de dicha función.

`filtra [2,3] [ [3,4,5],[],[4,5,6],[7,2],[2,7] ]` $\Rightarrow$ `[ [4,5,6], [7,2] ]`

```
filtra :: Eq a => [a] -> [[a]] -> [[a]]
filtra xs = filter buena
 where
  buena []     = False
  buena (w:ws) = w `notElem` xs
```

(e)(1,5 pts) Una *rotación de un título de libro* es una rotación de sus palabras que no empieza por una palabra trivial. Si tenemos

```
triviales :: [String]
triviales  = ["la", "como", "de", "a", "con",  "su"]
```

que nos proporciona una lista de palabras triviales, escribe la función `rotTitulo` que devuelva todas las *rotaciones de un título de libro* que se le pasa como argumento. Por ejemplo

```
rotTitulo "introduccion a la programacion funcional con haskell"
=>
["introduccion a la programacion funcional con haskell",
 "programacion funcional con haskell introduccion a la",
 "funcional con haskell introduccion a la programacion",
 "haskell introduccion a la programacion funcional con"]
```

Para ello, puedes utilizar las funciones predefinidas `words` y `unwords` que respectivamente separan las palabras de una cadena y forman una cadena a partir de una lista de palabras. Por ejemplo:

```
    words  "la casa roja" => [ "la", "casa", "roja" ]
    unwords [ "la", "casa", "roja"  ]} => "la  casa roja"
```

```
rotTitulo :: String -> [String]
rotTitulo  =
  map unwords .
  filtra triviales .
  rotaciones .
  words
```

$(f)(1\ pt)$ Un índice de palabras claves en contexto (KWIC, KeyWords In Context) es una lista ordenada de *rotaciones de títulos*. Utilizando las funciones de los apartados anteriores, define la función `kwic` que tome una lista de títulos y devuelva su índice KWIC.

```
kwic  ["la casa roja", "volver a casa", "roja como su madre"]
=>
["casa roja la", "casa volver a", "madre roja como su",
 "roja como su madre", "roja la casa", "volver a casa"]
```

```
kwic :: [String] -> [String]
kwic  = sort . k  rotTítulo
```

Alumno: _____ Grupo: _____

Ejercicio 2

Sea la definición de árboles no vacíos con información en nodos y hojas siguiente

```
data Arbol a = Hoja a | Nodo (Arbol a) a (Arbol a) deriving Show
```

$(a)(0{,}5\ pts)$ Define de modo recursivo la función `sumNH` que devuelva un par cuya primera componente es la suma de los valores de los nodos y su segunda componente es la suma de los valores de las hojas. Da el tipo mas general de la función `sumNH`.

```
sumNH :: Num a => Arbol a -> (a,a)
sumNH (Hoja x) = (0,x)
sumNH (Nodo i x d ) = (sni+snd+x,shi+shd)
    where
        (sni,shi) = sumNH i
        (snd,shd) = sumNH i
```

$(b)(1{,}5\ pts)$ Redefine la función anterior utilizando el siguiente plegado

```
plegadoArbol :: (b -> a -> b -> b) -> (a -> b) -> Arbol a -> b
plegadoArbol f g (Hoja x)     = g x
plegadoArbol f g (Nodo i x d) = f (plegadoArbol f g i) x (plegadoArbol f g d)
```

que también puede escribirse como

```
plegadoArbol f g = fun
    where
            fun (Hoja x)     = g x
            fun (Nodo i x d) = f (fun i) x (fun d)
```

```
sumNH :: Num a => Arbol a -> (a,a)
sumNH = plegadoArbol f g
    where
    f (sni,shi) x (snd,shd) = (sni+snd+x,shi+shd)
    g x                     = (0,x)
```