Escuela Técnica Superior de Ingeniería Informática

Departamento de Lenguajes y Ciencias de la Computación

SOLUCIONES

? ?
Apellidos Nombre

___ deseo que se publique mi calificación si fuera negativa.

PROGRAMACION DECLARATIVA (y LAB. V)
ORDINARIO DE FEBRERO (20/02/03)

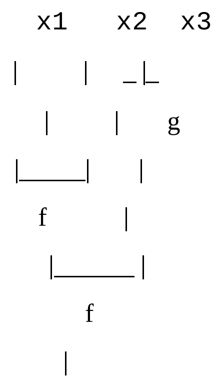
PUNTUACION: A.- 1 **B.-** 1 **C.-** 1 **D.-** 2 **E.-** 1 **F.-** 1 **G.-** 3

? ?

Consideremos la siguiente función de plegado a la derecha de listas no vacías:

```
pliega f  g   [x]    = g x
pliega f  g   (x:xs) = f x (pliega f g xs)
```

A Describa con un dibujo el funcionamiento de dicha función:



B Deducid el tipo de la función `pliega`,

`pliega ::` ____

justificando la respuesta:

SOLUCION. La función tiene 3 argumentos, siendo el último una lista, luego el tipo es t1 -> t2 -> [a] -> b. Por la primera ecuación, g es una función, y x es de tipo a, luego g :: a->b, de donde t2 = a -> b. Por la segunda ecuación, f tiene dos argumentos, siendo el primero de tipo el de x (o sea, a), y el segundo del mismo tipo que el resultado. Luego t1 = a -> b -> b, y finalmente

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

C Utilizando exclusivamente la función pliega define una función para sumar los elementos de una lista no vacía, así como para calcular la longitud de una lista no vacía:

```
suma :: Num a => [a] -> a
suma = pliega (+)  id

longitud :: [a] -> Integer
longitud = pliega (const (+1))  (const 1)
```

D Sea la definición (escriba además su tipo)

```
aplica :: (a -> b) -> [a] -> [b]
aplica h  = pliega ((:) . h) ((:[]) . h)
```

Siendo `map` la función estándar de Prelude probar que se verifica (indicando la técnica utilizada):

? `xs . xs::[a], xs/=[] . aplica h xs = map h xs`

SOLUCIÓN. La técnica es inducción estructural sobre la lista xs. El caso base es para xs ?[x], y la prueba es fácil

```
aplica h [x] = map h [x]
```

? ¡ siendo `f = (:) . h,` y tomando `g = ((:[]) . h),` por la segunda ecuación de `map`

```
g x = h x : map h []
```

? ¡ por definición de `g,` y primera ecuación de `map`

```
(:[])(h x) = h x : []
```

? ¡ cálculo, `[h x] = h x : []`

```
Cierto
```

El paso inductivo es también simple:

```
aplica h (x:xs) = map h (x:xs)
```

? ¡ por las segundas ecuaciones de `aplica` y de `map`

```
f x (aplica h xs) = h x : map h xs
```

? ¡ por definición de `f,` y por Hipótesis de inducción

```
h x : aplica h xs  = h x : aplica h xs
```

? ¡

```
Cierto
```

E Reduzca a forma normalizada la expresión `reem max [3,2,1],` siendo `reem` la siguiente función (escriba su tipo)

```
reem :: (a -> a -> a) -> [a] -> [a]
reem h xs = r where (r,m)       = pliega f g xs
                    g v         = ([m],v)
                    f v (ms,u)  = (m:ms, h v u)
```

SOLUCIÓN. Es necesario reducir la expresión `pliega f g [3,2,1]` en el ámbito de las definiciones locales; aplicando el apartado 1 encontramos:

```
pliega f g [3,2,1] -> ([m,m,m], max 3 (max 2 1)) -> ([m,m,m], 3)
```

Por tanto, `reem max [3,2,1] = r where (r,m) = ([m,m,m], 3)`

de donde `reem max [3,2,1] -> [3,3,3]`

F Utilizando exclusivamente la función `reem` escriba una función para sustituir todos los elementos de una lista por su suma

```
sustituye :: Num a => [a] -> [a]
sustituye = reem   (+)
```

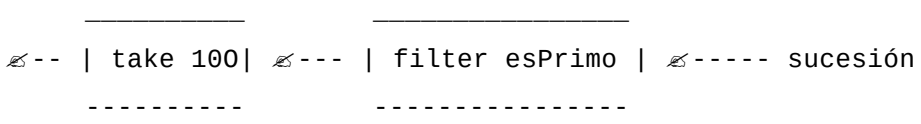
G Describa una red de procesos para calcular los 100 primeros números primos de la sucesión definida con la siguiente recurrencia

$a_0 = 1, \qquad a_{n+1} = n + a_n$

(describa el gráfico así como las ecuaciones correspondientes en Haskell). Las ecuaciones en Haskell son:

```
sucesión = 1: zipWith (+) [0..] sucesión
sol = take 100 (filter esPrimo sucesión)
esPrimo n = [ k | k<-[1..n], n `mod` k == 0 ] == [1,n]
```

El gráfico es:



siendo el gráfico de la red corrensponsdiente al flujo `sucesión` similar al de la página 215 del libro.