

Practica 2



Sergio Garcia Macias

Índice

Índice	2
Tipos de soluciones	3
• Cabecera:	3
• Branches:	3
• Variables locales vs Funciones.	4
nsobrek	5
Soluciones:	5
PITÁGORAS	7
Soluciones:	7
Fibonacci	9
Soluciones	9
Pertenece	11
Soluciones	11
EULER 14	13
Soluciones:	13
EULER 33	15
Soluciones:	15
¿ Conclusión ?	17

Tipos de soluciones

Para la resolución de los ejercicios se nos pide usar todas las formas posibles vistas en clase pero esto conlleva un gran número de variantes en cada uno de los ejercicios. Debido a esto expondré aquí las principales formas en las que podríamos variar la declaración de una función en Haskell.

En la cabecera de la función podemos usar genéricos para que Haskell infiera el código a la hora de su ejecución o podemos usar tipos como `Double` con el que solo podremos llamar a la función con el tipo de dato especificado.

- Cabecera:
 - Genéricos:
ejemplo::(`Fractional a`) => `a -> a`
 - Especializados:
ejemplo:: `-> Double -> Double`

Para crear branches/ramas podemos usar los tradicionales *if then else* de la programación imperativa o también podemos hacer uso del pattern matching con los *guards statements* que nos ofrece Haskell.

- Branches:
 - If then else:

```
if edad >= 18 then
    True
else
    error "Eres menor de edad"
```
 - guards statement:

```
calcular_impuesto ingreso
| ingreso <= 10000 = ingreso * 0.1
| ingreso <= 50000 = ingreso * 0.2
| ingreso <= 100000 = ingreso * 0.3
| otherwise = ingreso * 0.4
```

Las cláusulas *where* se utilizan para definir variables locales en una función. Estas variables locales son visibles sólo en el scope de la función en la que se definen. Se podrían comparar con lo que en otros lenguajes llamamos “funciones anónimas”, “lambda functions” o “closures”.

- Variables locales vs Funciones.

- funciones:

- suma:: Integer -> Integer -> Integer*
suma x y = x + y

- ejemplo:: Integer -> Integer -> Integer*
*ejemplo x y = suma(x y) * y*

- Where

- ejemplo:: Integer -> Integer -> Integer*
*ejemplo x y = suma(x y) * y*
where
suma x y = x+y

nsobrek

Sea la función nsobrek tal que nsobrek n k es el número de combinaciones de n elementos tomados de k en k; es decir:

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

Soluciones:

Este ejercicio consta de dos partes fundamentales. La parte en la que se calcula el factorial de un número y el resto de operaciones presentes en la fórmula.

```
1 fact:: Double -> Double
2 fact 1 = 1
3 fact n = n * fact(n-1)
4
5
6 ifact:: Double -> Double
7 ifact n = (foldr(\x y -> x*y) 1 [1,2..n])
```

Estas son las formas en las que se puede calcular el factorial de un número en Haskell, tanto de forma recursiva (fact) como de forma iterativa (ifact).

El código para calcular el resto de la fórmula es bastante simple. Según las diferentes maneras de declarar funciones en Haskell se podría hacer de muchas más formas pero yo he propuesto las siguientes.

Hago uso de las cláusulas *where* para crear variables locales y poder calcular el factorial además de usar la notación de tipos genéricos en las cabeceras.

```
1 insobrek:: Double -> Double -> Double
2 insobrek n k = ifact(n) / (ifact(k) * ifact (n-k) )
3
4 nsobrek:: Double -> Double -> Double
5 nsobrek n k = fact(n) / (fact(k) * fact (n-k) )
6
7 gnsobrek::(Fractional a) => a -> a -> a
8 gnsobrek n k = wfact(n) / (wfact(k) * wfact (n-k) )
9     where
10         wfact 0 = 1
11         wfact n = n * wfact(n-1)
12
13
14 nsobrek2:: Double -> Double -> Double
15 nsobrek2 n k = wfact(n) / (wfact(k) * wfact (n-k) )
16     where
17         wfact 0 = 1
18         wfact n = n * wfact(n-1)
```

El resultado que obtenemos en todas las versiones es el mismo pero el rendimiento en cada una de ellas difiere:

```
Main> nsobrek2 10 2
45.0
(318 reductions, 641 cells)
Main> nsobrek 10 2
45.0
(279 reductions, 580 cells)
Main> insobrek 10 2
45.0
(535 reductions, 938 cells)
Main> gnsobrek 10 2
45.0
(325 reductions, 646 cells)
```

PITÁGORAS

(No tiene nada que ver con pitágoras)

Definir la función raíces tal que raíces a b c es la lista de las raíces de la ecuación $ax^2 + bx + c = 0$.

Soluciones:

En estos dos ejemplos tenemos soluciones parecidas en las que en la primera función se hace uso de “*if then else*” para controlar el flujo del programa y evitar hacer raíces sobre números negativos mientras que en la segunda función usamos los guards para el mismo propósito.

Las soluciones son muy parecidas y el comportamiento del programa es el mismo.

```
1 pitagoras:: Double -> Double -> Double -> [Double]
2 pitagoras a b c =
3     if b*b - 4*a*c > 0 then
4         [((-b-sqrt(b*b - 4*a*c))/(2*a)), ((-b+sqrt(b*b - 4*a*c))/(2*a))]
5     else
6         error "Error"
7
8 gpitagoras:: Double -> Double -> Double -> [Double]
9 gpitagoras a b c
10    | (b*b - 4*a*c) >= 0 = [
11        ((-b-sqrt(b*b - 4*a*c))/(2*a)),
12        ((-b+sqrt(b*b - 4*a*c))/(2*a))
13    ]
14    | otherwise = error "Error"
```

La siguiente solución carece de control flow para evitar errores pero muestra la forma en la que podemos crear variables locales para organizar un poco mejor el código. En la primera forma de solucionar el problema hacemos uso de dos funciones:

“tagoras” y “pi” mientras que en la segunda forma sólo necesitamos declarar una función: “wpi”, que tendrá dos variables locales para realizar los cálculos correspondientes “tago” y “goras”

```

1  tagoras:: (Double -> Double) -> Double -> Double -> Double -> Double
2  tagoras func a b c = func (sqrt(b*b-4*a*c))
3
4  pi:: Double -> Double -> Double -> [Double]
5  pi a b c = [
6      (-b + tagoras negate a b c) / (2*a),
7      (-b + tagoras id a b c)/(2*a)
8  ]
9
10
11 wpi:: Double -> Double -> Double -> [Double]
12 wpi j i k = [tago j i k, goras j i k]
13     where
14         tago x y z = (-y -sqrt(y*y - 4*x*z)) / (2*x)
15         goras x y z = (-y +sqrt(y*y - 4*x*z)) / (2*x)

```

El resultado que obtenemos es el siguiente:

```

Main> pitagoras 2 2 0
[-1.0,0.0]
(109 reductions, 276 cells)
Main> gpitagoras 2 2 0
[-1.0,0.0]
(111 reductions, 241 cells)
Main> pi 2 2 0
[-1.0,0.0]
(95 reductions, 210 cells)
Main> wpi 2 2 0
[-1.0,0.0]
(130 reductions, 242 cells)

```


Fibonacci

Los números de Fibonacci quedan definidos por las ecuaciones:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Esto produce los siguientes números:

- $f_2 = 1$
- $f_3 = 2$
- $f_4 = 3$
- $f_5 = 5$
- $f_6 = 8$
- $f_7 = 13$
- $f_8 = 21$

y así sucesivamente.

Soluciones

Aunque en este problema plantee menos soluciones que en los anteriores, tiene un punto extra bastante interesante, y es el rendimiento del algoritmo.

```
1 fib :: Integer -> Integer
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib(n-1) + fib(n-2)
5
6
7 ifib :: Integer -> Integer
8 ifib n = fst (foldr(\x y -> (snd y, (fst y + snd y))) (1,1) [1,2..n])
9
10 wfib :: Int -> Integer
11 wfib n = fibHelper n 0 1
12     where
13         fibHelper 0 a _ = a
14         fibHelper n a b = fibHelper (n - 1) b (a + b)
```

Una vez más la evidencia empírica está apunto de demostrar que por muy bonita y segura que la recursividad parezca en la teoría, en la práctica es una de las posibles peores formas de resolver algo. Veamos el resultado de ejecutar las 3 versiones con “:set +sw”

```
Main> fib 10
89
(3177 reductions, 5190 cells)
Main> ifib 10
89
(266 reductions, 406 cells)
Main> wfib 10
89
(179 reductions, 270 cells)
```

Para calcular el fib de 10 la versión recursiva tiene que hacer:

3177 REDUCCIONES
5190 CELLS

A modo de exageración podríamos afirmar que:

“Mi abuela sin saber sumar ni restar es más eficiente que Haskell.

- Sergio García Macías 2023”

La segunda versión es algo más prometedora, hace uso de 266 reducciones y 406 cells, es unas 12 y 13 veces respectivamente más eficiente que la primera versión.

La tercera versión con cláusulas where es la más eficiente, haciendo uso de únicamente 179 reducciones y 270 cells

Indagando un poco más sobre la eficiencia de este algoritmo en Haskell he podido descubrir que la forma tradicional inductiva ($fib(n-1) + fib(n-2)$) es de lo más ineficiente posible. También hemos de tener en cuenta que Haskell es un lenguaje de programación declarativo funcional, por lo que al intentar llegar al caso base de todas las maneras posibles, la complejidad de un problema que a priori parece una simple suma, se vuelve exponencial.¹

¹ Debido a una mala planificación (mia) a la hora de realizar el trabajo no he tenido tiempo para poner gráficas y tablas donde se muestre la diferencia de rendimiento entre las diferentes versiones.

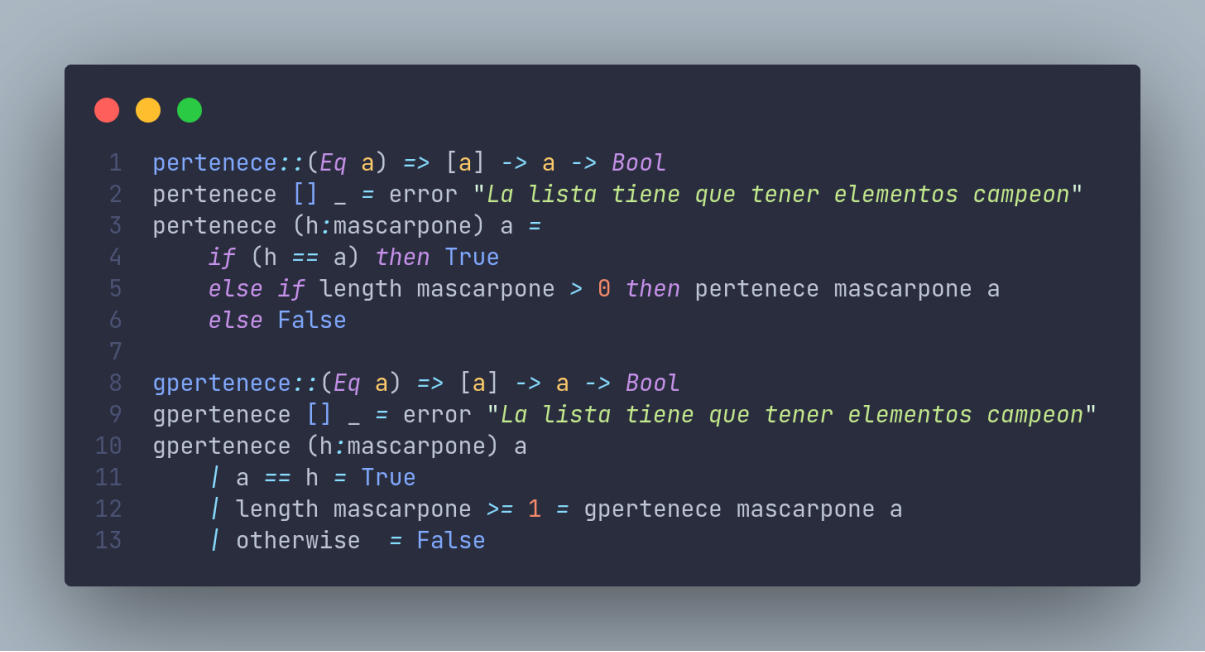
Pertenece

Comprobar la pertenencia a una lista usando una función recursiva.

Soluciones

Para este ejercicio propongo dos soluciones las cuales usan tipos genéricos `Eq` para poder verificar la igualdad entre elementos.

La solución está planteada de dos formas diferentes aunque existen muchas más posibles soluciones si aplicamos los “Tipos de soluciones” vistos en el punto 1.



```
1 pertenece::(Eq a) => [a] -> a -> Bool
2 pertenece [] _ = error "La lista tiene que tener elementos campeón"
3 pertenece (h:mascarpone) a =
4     if (h == a) then True
5     else if length mascarpone > 0 then pertenece mascarpone a
6     else False
7
8 gpertenece::(Eq a) => [a] -> a -> Bool
9 gpertenece [] _ = error "La lista tiene que tener elementos campeón"
10 gpertenece (h:mascarpone) a
11     | a == h = True
12     | length mascarpone >= 1 = gpertenece mascarpone a
13     | otherwise = False
```

El código es bastante auto-explicativo, en las dos formas la cabecera es la misma, recibimos una lista de elementos “*a*” tal que “*a*” es “*Eq*”. En ambas formas también encontramos un caso base donde se considera la posibilidad de encontrar una lista vacía. En tal caso lanzamos un error.

Para la parte recursiva la lógica es la misma, lo que difiere es la sintaxis utilizada. En la primera forma usamos “*if then else*” mientras que en la segunda hacemos uso de “*guards*”. En ambas valoramos los mismos casos y llamamos recursivamente con los mismos argumentos.

Al igual que en los otros ejercicios, el resultado de ambas versiones es el mismo pero difieren en rendimiento:

```
Main> pertenece [1..100] 3
True
(2858 reductions, 3617 cells)
Main> gpertenece [1..100] 3
True
(2862 reductions, 3595 cells)
Main> pertenece [1..100] 1000
False
(37301 reductions, 43083 cells)
Main> gpertenece [1..100] 1000
False
(37601 reductions, 43383 cells)
```

EULER 14

Longest Collatz Sequence

Problem 14

The following iterative sequence is defined for the set of positive integers:

$$\begin{aligned} n &\rightarrow n/2 \text{ (} n \text{ is even)} \\ n &\rightarrow 3n + 1 \text{ (} n \text{ is odd)} \end{aligned}$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1.$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1.

Which starting number, under one million, produces the longest chain?

NOTE: Once the chain starts the terms are allowed to go above one million.

Soluciones:

Para este problema tenemos 3 soluciones bastante parecidas entre sí:

```
1 collatz:: Integer -> Integer -> Integer
2 collatz i 1 = i
3 collatz i n
4     | even n = collatz (i+1) (n `div` 2)
5     | otherwise = collatz (i+1) (n*3+1)
6
7
8 ifcollatz:: Integer -> Integer -> Integer
9 ifcollatz i n =
10    if n == 1 then i
11    else if even n then ifcollatz (i+1) (n `div` 2)
12    else ifcollatz (i+1) (n*3+1)
13
14 wcollatz:: Integer -> Integer -> Integer
15 wcollatz i n = wcollatz' i n
16     where
17         wcollatz' acc 1 = acc
18         wcollatz' acc num
19             | even num = wcollatz' (acc+1) (num `div` 2)
20             | otherwise = wcollatz' (acc+1) (num*3+1)
```

Antes que nada explicar que la cabecera recibe 2 enteros, el offset donde queremos empezar y el número al que queremos aplicar el algoritmo. Para el enunciado original del problema el offset siempre será 0.

Tanto la primera como la segunda y tercera solución hacen el mismo cómputo pero expresado de manera diferente.

```
Main> collatz 0 10
6
(480 reductions, 764 cells)
Main> ifcollatz 0 10
6
(466 reductions, 743 cells)
Main> wcollatz 0 10
6
(513 reductions, 799 cells)
```

Como podemos observar el resultado es el mismo pero la versión en la que se utilizan “*if then else*” es un poco mejor a las otras dos.

En el enunciado se nos pedía calcular cual es la mayor secuencia de collatz por debajo de un millón, esto aunque posible, consume demasiado tiempo por lo que no es recomendable probarlo.

Dejo por aquí la posible solución:

```
1 chunkList :: Int -> [a] -> [[a]]
2 chunkList _ [] = []
3 chunkList n xs = take n xs : chunkList n (drop n xs)
4
5 max_seq :: Int -> [Integer] -> Integer
6 max_seq n range = maximum (map (\chunk -> maximum (map (\e -> collatz 0 e) chunk)) (chunkList n range))
```

EULER 33

Digit Cancelling Fractions

Problem 33

The fraction $49/98$ is a curious fraction, as an inexperienced mathematician in attempting to simplify it may incorrectly believe that $49/98 = 4/8$, which is correct, is obtained by cancelling the 9s.

We shall consider fractions like, $30/50 = 3/5$, to be trivial examples.

There are exactly four non-trivial examples of this type of fraction, less than one in value, and containing two digits in the numerator and denominator.

If the product of these four fractions is given in its lowest common terms, find the value of the denominator.

Soluciones:

Teniendo en cuenta que este es un problema relativamente complicado voy a ofrecer una única solución usando todo lo visto hasta el momento:

```
1  separa_fract:: [Char] -> [[Char]]
2  separa_fract cadena
3      / elem '/' cadena == True = [takeWhile(\x -> x /= '/') cadena, tail (dropWhile(\x -> x /= '/') cadena)]
4      / otherwise = error "No es una entrada valida"
5
6
7  quita_ele_final:: Eq a => a -> [a] -> [a]
8  quita_ele_final target list
9      / length (dropWhile (/=target) list) == 0 = []
10     / otherwise = tail (dropWhile (/=target) list)
11
12 quita_ele:: Eq a => a -> [a] -> [a]
13 quita_ele _ [] = []
14 quita_ele target list = takeWhile (/=target) list ++ quita_ele_final target list
15
16 quitaRepes ::(Eq a) => Int -> [a] -> [a] -> [a]
17 quitaRepes index den nume
18     / index >= length den = nume
19     / length nume == 0 = nume
20     / elem (den!!index) nume = quitaRepes (index+1) den (quita_ele (den!!index) nume)
21     / otherwise = quitaRepes (index+1) den nume
22
23
24 euler33:: [Char] -> [Char]
25 euler33 cadena = primera_parte cadena ++ "/" ++ segunda_parte cadena
26     where
27         primera_parte c = quitaRepes 0 (separa_fract c !! 1) (separa_fract c !! 0)
28         segunda_parte w = quitaRepes 0 (separa_fract w !! 0) (separa_fract w !! 1)
```

El input/output esperado por las funciones es el siguiente:

“separa_fract” recibe una cadena y las separa por el delimitador ‘/’

Input “33/77”

Output [“33”, “77”]

“quita_ele_final” recibe un elemento *target* que tiene que eliminar de la lista *list*. Esta función existe para tener el código más organizado ya que hay que controlar el caso en el que el *tail* del *dropWhile* sea vacío. También podríamos haber puesto *quita_ele_final* en una clausura *where* en *quita_ele*.

“quita_ele” simplemente se encarga de eliminar el primer elemento *target* que encuentre en la lista *list*.

“quitaRepes” es donde ocurre la “magia”. Es la función encargada de eliminar los elementos de *nume* que se encuentran en *den*. Esta función se ejecuta de tal forma que solo eliminará 1 instancia del elemento por lo que sí tenemos:

“3” y “133”

El resultado será:

“13”

Solo hemos quitado un 3.

```
Main> quitaRepes 0 "33" "3377"
"77"
Main> quitaRepes 0 "3" "3377"
"377"
Main> quitaRepes 0 "7" "3377"
"337"
Main> quitaRepes 0 "77" "3377"
"33"
Main> quitaRepes 0 "7773" "3377"
"3"
```

Este de aquí es un ejemplo de la salida producida por la función “quitaRepes”..

```
Main> euler33 "33/77"
"33/77"
(807 reductions, 1041 cells)
Main> euler33 "33/377"
"3/77"
(1031 reductions, 1331 cells)
Main> euler33 "/"
"/"
(263 reductions, 375 cells)
Main> euler33 "1111/1111"
"/"
(2175 reductions, 2775 cells)
Main> euler33 "11121/11121"
"/"
(2983 reductions, 3775 cells)
Main> euler33 "151121/111271"
"5/7"
(3900 reductions, 4872 cells)
Main> 
```

“euler33” es el encargado de unificar todas las funciones anteriores para obtener el resultado obtenido.

¿ Conclusión ?

Aunque Haskell aparente ser un lenguaje “seguro” a priori, sin side effects y libre de errores que no sean culpa del programador, a la hora de realizar los ejercicios me he encontrado con fallos catastróficos.

Hemos de tener en cuenta que el intérprete hugs está hecho por personas, y todo lo que está hecho por personas tiene errores. Hugs no es una excepción:

```
[10:02PM] [ sergio@archlinux:/mnt/Uni/Programacion/Asignaturas/MAC/Practical1 ]
$ hugs

-- -- -- -- --
||  ||  ||  ||  ||  ||  ||__      Hugs 98: Based on the Haskell 98 standard
||__||  ||__||  ||__||  __||      Copyright (c) 1994-2005
||__||  ||__||  ||__||  __||      World Wide Web: http://haskell.org/hugs
||  ||  ||  ||  ||  ||  ||__      Bugs: http://hackage.haskell.org/trac/hugs
||  ||  ||  ||  ||  ||  ||__      Version: September 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> head (reverse [1..2000])
2000
Hugs> head (reverse [1..20000])
20000
Hugs> head (reverse [1..200000])
[1] 71369 segmentation fault (core dumped) hugs
```

Cuando Haskell ejecuta el código sobre el intérprete escrito en C, obtenemos un segfault, un fallo típico de lenguajes donde el manejo de memoria es manual.

En la teoría era muy bonito, pero en la práctica sigue sin funcionar.