Estructuras Discretas 2023-1

Práctica 1: Introducción a Haskell

Lourdes del Carmen González Huesca Juan Alfonso Garduño Solís
María Fernanda Mendoza Castillo

Fecha de entrega: Sábado 3 de septiembre de 2022

Antes de los ejercicios escribí este resumen de las dos clases del laboratorio con todo lo necesario para que realicen la práctica, cualquier duda al final del documento se encuentra un código QR para contactarme por Telegram. Si no entienden algo, se sienten muy perdidos o quieren ayuda para la instalación de un sistema operativo o programa sientanse con la libertad de contactarme.

Apunte.

En este PDF están todas las funciones que hemos definido en las clases del laboratorio, pero comenzaremos con cómo interpretar un archivo en el modo interactivo de ghci y poder utilizar nuestras propias funciones.

Interpretar un archivo.

Primero tenemos que abrir una terminal en el directorio en el que se encuentra nuestro archivo con extensión *.hs*, para el ejemplo utilizaremos el archivo "apunte.hs" que se encuentra en la parte de laboratorio de la página del curso, una vez hecho esto ejecutamos:

```
$> ghci apunte.hs
```

Con esto ya podemos utilizar las funciones que están definidas en el archivo apunte.hs.

Funciones.

Ahora, ¿cómo hacemos una función?, la sintáxis (manera correcta de escribir y organizar código) en haskell para definir una función se parece mucho al de las funciones algebráicas, por ejemplo, la función

$$f(x, z) = 2x + z$$

En haskell se escribe

```
f x z = 2 * x + z
```

dónde obviamente * es la multiplicación. Así que para definir una función **SIEMPRE** va primero el nombre que le vamos a dar, a continuación los argumentos que recibe, e inmediatamente el simbolo "="para después escribir el comportamiento la función (lo que debe hacer) con los argumentos que le estamos pasando.
En nuestro ejemplo el nombre de la función es `f`, los argumentos son `x`, `z` y lo que va a hacer la función `f` es multiplicar por dos a `x` y a eso sumarle `z`.
Adicionalmente para que una función en haskell este completa debe llevar la firma, esta significa **QUÉ** tiene que recibir nuestra función para trabajar correctamente y **QUÉ** va a devolver. Por ejemplo, para el caso de la función `f` la firma sería:

```
f :: Int -> Int -> Int
```

Esto en español sería: `f` recibe dos enteros y devuelve un entero. Por último, el lugar correcto para colocar la firma de una función es arriba de su primer definición, por eso la sintáxis correcta completa para `f` es:

```
f :: Int -> Int -> Int
f x z = 2 * x + z
```

Pattern matching.

Una vez que ya sabemos como se escribe una función repasemos las primeras funciones que escribimos durante las clases:

```
zuma :: Int -> Int -> Int
zuma a b = a + b

zuma2 :: Int -> Int -> Int
zuma2 a 0 = a       --Primer caso
zuma2 0 b = b       --Segundo caso
zuma2 a b = a + b   --Tercer caso
```

La primer función recibe dos números y devuelve su suma, nada fuera de lo común, pero zuma2 tiene dos casos adicionales, el primero espera que el segundo argumento sea un cero, en cuyo caso devuelve imediatamente el primer arguemtno sin hacer ninguna operación, el segundo caso es similar pero espera que ahora el primer argumento sea cero para devolver el segundo argumento sin hacer ninguna operación y por último el tercer caso entra en acción cuando ninguno de los argumentos es cero y por eso tiene que hacer la operación $a + b$ para devolver el resultado. A esto se le llama *pattern matching* o *caza de patrones* y es una herramienta poderosa que nos brinda Haskell.

Booleanos.

Presentamos ahora los valores de verdad en haskell, que sintácticamente son `True` y `False`, estos nos ayudaran a saber si algo se cumple o no por ejemplo, el enunciado 3 es un número par es falso, o sea que si traducimos el enunciado a código y lo ejecutamos, haskell contestaría con `False`. Por otra parte si preguntamos si 6 es un número par, haskell contestaría `True`.
Así como con los enteros, en haskell podemos operar con los valores Booleanos; Igual que en la lógica clásica, haskell tiene definida la conjunción (and), disyunción (or) y la negacion (not), and y or son operadores binarios, es decir reciben dos argumentos, como la suma o la multiplicación, mientras que la negación es un operador unario que cambia el valor del argumento que recibe y se representan por `&&` para and, `||` para or y `not` para negacion. Por ejemplo en ghci los puedes utilizar de esta manera:

```
Prelude> True && False
Prelude> False || True
Prelude> not True
```

Naturalmente también podemos utilizarlos en el código que escribamos en nuestro archivo .hs

Bisiesto.

El problema que buscamos resolver con esta función es saber si un año representado por un número n es bisiesto o no, así que por lo pronto el esqueleto de nuestra función iría de la siguiente manera:

```
bisiesto :: Int -> Bool
bisiesto n =
```

Primero revisemos los criterios para saber si un año es o no bisiesto:

Un año es bisiesto si es múltiplo de 4, a menos que sea año secular (último año del siglo ej: 100, 200), en cuyo caso solo los años múltiplos de 400 serán bisiestos.

O sea que un año bisiesto debe ser divisible entre cuatro, pero no debe ser divisible entre 100, a menos de que sea múltiplo de 400. Definimos entonces las siguientes proposiciones:

$p =$ el año es divisible entre cuatro

$q =$ el año es divisible entre 100

$r =$ el año es divisible entre 400

Detenidamente, si queremos saber si un año representado por el número n es bisiesto debe cumplir, ser divisible entre 4 (p) y no ser divisible entre 100 ($\neg q$) o ser divisible entre 400 (r), que utilizando la notación de la lógica proposicional y las proposiciones que definimos arriba sería:

$$(p \wedge \neg q) \vee r$$

Ya sabemos como utilizar los operadores booleanos en Haskell, pero ¿Cómo sabemos que un número n es divisible por otro número m?, muy sencillo, debemos de revisar si el residuo de dividir a n por m es cero; Y la operación para obtener el residuo de la división entera de dos números se llama modulo, en haskell el modulo esta representado por la función `mod`, así que por ejemplo si queremos saber el modulo de 10 y 2 con haskell debemos hacer:

```
Prelude> mod 10 2
0
```

Perfecto, ahora que sabemos como obtener el residuo solo queda saber como comprobar si el resultado es cero, para eso vamos a utilizar `==` que nos sirve para saber si dos cosas son equivalentes, por ejemplo:

```
Prelude> 1 == 2
False
```

Obviamente 1 es diferente de 2, por eso haskell responde con `False`, no se debe confundir `=` con `==` , el primero en haskell nos sirve para definir, meintras que `==` sirve para comparar y saber si dos cosas son iguales. Si juntamos todas las piezas ya es fácil ver que para verificar si un número n es divisible por otro número m podemos hacer lo siguiente:

```
(mod n m) == 0
```

Por ejemplo si n es 10 y m es 2, haskell va a contestar `True`. Ahora, traducimos las proposiciones p,q y r para el año n de la siguiente manera:

```
(mod n 4) == 0   -- n es divisible por 4
(mod n 100) == 0 -- n es divisibple por 100
(mod n 400) == 0 -- n es divisibple por 400
```

Y ya solo queda juntarlos con los operadores booleanos:

```
((mod n 4 == 0) && not(mod n 100 == 0)) || (mod n 400 == 0)
```

Esta es justo la parte que le faltaba al esqueleto de función que definimos al inicio, así que la función completa es:

```
bisiesto :: Int -> Bool
bisiesto n = ((mod n 4 == 0) && not(mod n 100 == 0)) || (mod n 400 == 0)
```

Recursividad.

Una definición recursiva es aquella que se define en terminos de sí misma, algo así como cuando no te daban permiso para hacer algo y al preguntar por qué te respondían "porque no". El ejemplo clásico para comenzar con recursión en haskell es la función factorial. El factorial de un número es el mismo número multiplicado por todos su antecesores hasta el 1 y se denota por !, por ejemplo:

$$!5 = 5 * 4 * 3 * 2 * 1$$

en clase se discutió la siguiente solución.

```
fac :: Int -> Int        -- fac recibe un entero y devuelve otro entero
fac 1 = 1                -- Caso base
fac n = n * (fac (n-1)) -- Caso recursivo
```

Esta es una función recursiva porque la estamos utilizando dentro de su propia definición. La evaluación de `fac 4` se vería de la siguiente manera:

```
1)  fac 4 = 4 * (fac 3)    --Entra en el caso del paso recursivo
2)  fac 3 = 3 * (fac 2)    --Entra en el caso del paso recursivo
3)  fac 2 = 2 * (fac 1)    --Entra en el caso del paso recursivo
4)  fac 1 = 1              --ENTRA EN EL CASO BASE
```

así que si sustituimos hacia arriba (utilizamos razonamiento ecuacional) sucede lo siguiente. Por `4)` sabemos que `fac 1 = 1`, sustituimos en `3)`:

```
fac 2 = 2 * 1
fac 2 = 2
```

Acabamos de obtener nuevo conocimiento, `fac 2 = 2`, así que sustituimos en `2)` para terminar su evaluación:

```
fac 3 = 3 * 2
fac 3 = 6
```

Y ahora que sabemos cuanto es `fac 3` podemos sustituir en `1)`:

```
fac 4 = 4 * 6
fac 4 = 24
```

Terminamos, así es como funcióna la recursividad.
Sin embargo, ¿qué pasaría si omitimos el caso base?, nunca terminaría la evaluación, o ¿qué sucedería si antes del caso base pongo el caso revursivo? de nuevo la función nunca terminaría, por eso es importante **NUNCA** olvidar el caso base y siempre poner los casos mas particulares antes de los mas generales.

Listas.

La anterior fue una función recursiva sobre los naturales, pero también podemos hacer recursión sobre estructuras de datos, como las listas, así que primero presentamos las listas de Haskell: Una lista es una estructura de datos que nos permite almacenar cosas, por ejemplo números, booleanos, caracteres, cadenas o cualquier otro tipo que podemos definir en haskell. Una lista que tiene almacenados los números del 1 al 4 se ve asi: `[1,2,3,4]`. Los elementos van separados por comas y encerrados entre corchetes, pero ¿Que pasa si no hay elementos en una lista? bueno, pues solo ponemos los corchetes sin nada en el medio `[]`, y esta es la lista vacía, la única lista que no tiene elementos.
Las listas tienen dos partes fundamentales: la cabeza y la cola, la cabeza es el primer elemento de la lista, por ejemplo la cabeza de `[1,2,3,4]` es 1 y la cola es toda la lista menos la cabeza, por eso la cola de la lista anterior es `[2,3,4]`. En haskell la función que devuelve la cabeza de una lista se llama `head` y la que devuelve la cola se llama `tail`.
Nota que la cabeza es un elemento de la lista, mientras que la cola vuelve a ser una lista.
Para hacer recursión sobre listas debemos seguir los siguientes pasos:

1) Pensar en el caso base con la lista vacía `[]`.
2) Pensar en el caso inductivo con una lista que tenga cabeza y cola `(x:xs)`

A continuación esta la función que obtiene la longitud (número de elementos) de una lista de manera recursiva:

```
lon []     = 0 -- Caso base, la longitud de una lista vacia es cero.
lon (x:xs) = 1 + lon xs -- Caso inductivo, la longitud
                        -- de una lista con cabeza y cola es
                        -- 1 mas la longitud de la cola.
```

¿Como se vería una evaluación de esta función?, tomemos el ejemplo de la lista [1,2,3,4]

```
1) lon [1,2,3,4] = 1 + lon [2,3,4]   -- Caso inductivo
2) lon [2,3,4] = 1 + lon [3,4]       -- Caso inductivo
3) lon [3,4] = 1 + lon [4]           -- Caso inductivo
4) lon [4] = 1 + lon []              -- Caso inductivo
5) lon [] = 0                        -- CASO BASE
```

Si de nuevo sustituimos hacia arriba lo que va a suceder es lo siguiente: Como sabemos que `lon [] = 0`, sustituimos en `4)`

```
lon [4] = 1 + 0
lon [4] = 1
```

Como `lon [4] = 1`, sustituimos en `3)`

```
lon [3,4] = 1 + 1
lon [3,4] = 2
```

Como `lon [3,4] = 2`, sustituimos en `2`)

```
lon [2,3,4] = 1 + 2
lon [2,3,4] = 3
```

Y finalmente sustituimos en `1`) esta nueva información

```
lon [1,2,3,4] = 1 + 3
lon [1,2,3,4] = 4
```

Terminamos. La recursión es una herramienta súper poderosa en los lenguajes de programación funciónales, si bien no es tan intuitivo como un ciclo, sí es más elegante. Las siguientes funciones son variaciones de la función `lon`, en cada caso se cambian cosas mínimas pero el comportamiento pasa a ser completamente diferente.

```
--Función que suma todos los elementos de una lista de números.
f1 [] = 0
f1 (x:xs) = x + f1 xs

--Función que multiplica todos los elementos de una lista de números pero
-- SIEMPRE da 0, ¿por que?
f2 [] = 0
f2 (x:xs) = x * f2 xs
```

Nuestro modulo.

La última función que programamos en clase fue `myMod`, que recibe una n y una m para devolver el equivalente a `mod n m`.

```
myMod n m = if (n < m)
            then n              -- Caso base
            else myMod (n-m) m  -- Caso recursivo
```

Nota que aquí utilizamos un `if` en vez de dividir los casos de la función.

Ejercicios.

Define cada una de las siguientes funciones, no olvides colocar su firma.

1) *(1 punto.)*
   - La función `par n`, que recibe un entero y contesta si es o no un número par.
   - La función `impar n`, que recibe un entero y contesta si es o no un número impar.
2) *(1 punto.)*
   - La función `minimo n m` que recibe dos enteros y contesta con el menor.
   - La función `maximo n m` que recibe dos enteros y contesta con el mayor.
3) *(1 punto.)* La función `absoluto n`, que recibe un entero y regresa su valor absoluto.
4) *(2 puntos.)* La función `divE n m`, que recibe dos enteros y devuelve la división entera de n y m.
5) *(1 punto.)* Tu propia versión de head y tail que se llamen `cabeza` y `cola` respectivamente.
6) *(2 puntos.)* La función `quita n lst`, que recibe un número entero positivo n y una lista lst para devolver lst pero sin sus primeros n elementos.

   ```
   ghci> quita 4 [1,2,3,4,5,6,7,8]
   [5,6,7,8]
   ```
7) *(2 puntos.)* La función `enesimo n lst` que toma un entero n y una lista lst para regresar el enésimo elemento de lst. *Los computologos siempre empezamos a contar desde el cero.*

   ```
   ghci> enesimo 4 [1,2,3,4,5,6,7,8]
   5
   ```

Notas.

- Las instrucciones para entregar las prácticas están en la página del curso, si aún no sabes cuál es haz click aquí.
- La única función que se les debería complicar es `divE`, si no la logras implementar no te preocupes,el viernes en la clase de laboratorio la repasaremos.
- Se responden dudas en el correo pero de preferencia enviame mensaje por Telegram. Puedes acceder a mi contacto haciendo click aqui.

No dudes en contactarme ante cualquier duda, aunque no sea de la materia.