Facultad de Ciencias - UNAM
Estructuras Discretas 2023-1
Práctica 3: Recursión

Laura Friedberg Gojman
Saúl Adrián Rojas Reyes
José Manuel Madrigal Ramírez

24/febrero/2023
Fecha de entrega: 5/Marzo/2023

Instrucciones

En esta práctica abordaremos el tema de recursión, que consiste en definir funciones aplicando la misma función que estamos definiendo. Del mismo modo que en las prácticas anteriores, junto a este PDF hay un archivo de haskell llamado Practica3.hs con las instrucciones puntuales de los ejercicios. Descárgalo y escribe las soluciones debajo de las instrucciones correspondientes.

Breve introducción a la recursión utilizando factorial

El factorial de un número n es la multiplicación de todos los números enteros positivos iguales o menores a él. Se suele utilizar la notación $n!$ para representar el factorial del número n. Mas adelante formalizaremos esta definición, por lo pronto miremos un par de ejemplos:

$$factorial(5) = 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$factorial(10) = 7! = 7 \times 6! = 5040$$

Observa que en el ejemplo anterior podemos definir el factorial de 7 como:

$$7! = 7 \times 6!$$

Es decir, estamos usando la operación factorial dentro de la definición del factorial de otro número. Este hecho nos ayudará a escribir la definición recursiva de factorial mas adelante.

Por lo pronto veamos cómo el factorial de seis también podría definirse como la multiplicación de seis por el factorial de cinco:

$$7! = 7 \times 6!$$

$$7! = 7 \times 6 \times 5!$$

Y si seguimos por ese camino, eventualmente podríamos escribir:

$$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times \mathbf{1}$$

Después de 1 ya no seguimos expandiendo la operación. Dejamos de multiplicar por números menores y nos detenemos en este punto pues 1 es el menor entero positivo. Lo cual nos lleva a pensar que:

$$1! = 1$$

En efecto, el factorial de 1 es 1 mismo. Ahora analicemos los siguientes ejemplos para destilar una definición general:

$$10! = 10 \times 9!$$

$$9! = 9 \times 8!$$

$$7! = 7 \times 6!$$

$$6! = 6 \times 5!$$

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

Si quisiéramos calcular el factorial de n, tendríamos que:

$$n! = n \times (n-1)!$$

Si unimos esta expresión a la definición de factorial de 1 podemos escribir:

$$n! = \begin{cases} 1 & \text{si } n <= 1 \\ n \times (n-1)! & \text{si } x > 1 \end{cases}$$

Toda definición recursiva se compone de dos conjuntos de casos:

- **Casos base**: definen resultados de la aplicación de la función de forma directa. En el ejemplo anterior $n! = 1$ es el caso base.
- **Casos recursivos**: la definición del resultado se compone de aplicaciones de la función a elementos más *simples* del dominio. En el ejemplo anterior $n! = n \times (n-1)!$ es el caso recursivo.

Finalmente podemos traducir nuestra definición recursiva de factorial a Haskell:

```haskell
factorial :: Int -> Int
factorial 1 = 1                          -- c. base
factorial n = n *  factorial (n - 1)  -- c. recursivo
```

Puedes conocer más sobre factorial en este vídeo.

Ejercicios

1. ¿Cómo se ejecutaría nuestra función `factorial` si la aplicamos a un valor negativo? ¿Qué sucedería si la ejecutamos con 0?

   Transcribe la definición de factorial y modifíca el caso recursivo con una guarda para evitar que reciba argumentos negativos. También añade un caso base para el 0.

2. El n-esimo elemento de la sucesión fibonacci $fib(n)$ puede definirse como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{si } x > 1 \end{cases}$$

   Define una función que devuelva el n-esimo elemento de la sucesión fibonacci. Procura evitar que se cicle con números negativos añadiendo algún mecanismo de seguridad.

   Aquí tenemos unos ejemplos de su ejecución:

```
*Main> fibonacci 31
1346269
*Main> fibonacci 6
8
*Main> fibonacci (-10)
*** Exception: fibonacci no está definido para valores negativos
CallStack (from HasCallStack):
error, called at Practica3.hs:21:20 in main:Main
```

3. Formalizar la idea de que la multiplicación se reduce a una suma nos puede brindar otro ejemplo de recursión. Veamos:

```haskell
multi :: Int -> Int -> Int
multi m 0 = 0                      -- c. base
multi m n = m + (multi m (n - 1))  -- c. recursivo
```

   Observa que al redefinir la operación `multi` como una función se aplica de manera prefija, es decir, el nombre de la función va antes de los operandos, por ejemplo: `multi 3 4`.

   Define recursivamente la operación de "potenciaütilizando multiplicación, análogamente a como utilizamos la suma en en el ejemplo anterior para definir la multiplicación.

```
*Main>potencia 3 7
2187
*Main>potencia 10203 0
1
```

4. En su obra *Elementos*, Euclides describió un método para calcular el máximo común divisor (MCD). Este resultado, expresado originalmente en términos de segmentos, puede condensarse en la siguiente proposición:

   Sean a y b enteros no nulos tales que $a \geq b$. Entonces

   $$MCD(a, b) = MCD(b, r)$$

   donde r es el resto de dividir a entre b.

   Por ejemplo, si queremos encontrar MCD(20, 12), gracias a la proposición anterior sabemos que:

   $$MCD(20, 12) = MCD(12, 8)$$

   ya que el residuo de dividir 20 entre 12 es 8. Si hacemos esto nuevamente obtenemos:

   $$MCD(12, 8) = MCD(8, 4)$$

   pues al dividir 12 entre 8 obtenemos 4. Siguiendo este mismo procedimiento llegamos a que:

   $$MCD(8, 4) = MCD(4, 0)$$

   porque el residuo de dividr 8 entre 4 es 0. Obtener el máximo común divisor de 4 y 0 es muy sencillo. Como todo número divide a 0 y queremos el mayor de los divisores de 4 y 0, claramente MCD(4,0) es 4, con lo cual, siguiendo la cadena de igualdades podemos concluir que:

   $$MCD(20, 12) = 4$$

   Escribe una función que calcule de manera recursiva el MCD de dos enteros positivos.

   Aquí tenemos unos ejemplos de su ejecusión:

   ```
   *Main> euclides 20 12
   4
   *Main> euclides 6 27
   3
   ```

Recursión con listas

Puede que entre los datos de entrada de la función que queramos definir exista una lista. A modo de ejemplo vamos a redefinir la función `producto` de una lista, que multiplicará todos los miembros de una lista de enteros:

a) **Definiendo la firma de tipo:** Es buena práctica definir la firma de tipo de nuestra función antes de definir a la función misma.

```
1 producto :: [Int] -> Int
```

b) **Enumerando los casos:** Poniendo atención a la estructura sobre la cuál haremos recursión enumeramos los casos antes de definirlos. Cuando hacemos recursión sobre listas de antemano debemos contar con un caso para la lista vacía y otro para la lista no vacía. A veces también conviene contar con un caso para la lista que tiene exactamente un elemento.

```
2 producto [] =          -- caso base (lista vacía)
3 producto (x:xs) =      -- caso recursivo (lista no vacía)
```

c) **Caso base:** Para este ejemplo definiremos el producto de la lista vacía como 1, ya que es la identidad de la multiplicación.

```
2 producto [] =  1       -- caso base (lista vacía)
3 producto (x:xs) =      -- caso recursivo (lista no vacía)
```

d) **Caso recursivo:** Podemos comenzar revisando los *ingredientes* con los que contamos como la función en sí misma (producto), los argumentos (`x` y `xs`) o las operaciones para los tipos de dato que estemos usando (+, -. *, etc). En este caso multiplicaremos la cabeza de la lista por el producto del resto de la lista:

```
2 producto [] =  1                  -- caso base
3 producto (x:xs) = x *  producto xs -- caso recursivo
```

Observa que en el caso recursivo utilizamos esta expresión:

(x:xs)

Analisémosla juntos:

- La variable **x** que se encuentra del lado izquierdo del operador *cons* es considerada **la cabeza de la lista** y en este caso es del tipo `Int`.
- La variable **xs** que se encuentra a la derecha, representa el **resto de la lista** que es del tipo `[Int]`, es decir, es otra lista de enteros.
- Los **paréntesis** son muy importantes porque hacen que GHC interprete la expresión interior como un único dato. Intuitivamente podemos pensar que le **brindan cohesión**. Esta expresión en conjunto se llama *patrón*. Veremos otros patrones en prácticas posteriores.

5. *Define una función que decida si todas las variables lógicas de una lista son verdaderas.*

   Aquí hay unos ejemplos de su ejecución en la terminal:

```
*Main>miAnd [True, True, False, True]
False
*Main>miAnd [True, True, True]
True
*Main>miAnd []
True
```

6. *Ahora, suponiendo que tenemos una lista de listas cuyos elementos son únicamente enteros, definamos una función que concatene las listas interiores.*

   Aquí hay unos ejemplos de su ejecución en la terminal:

```
*Main>concatena [[1..9], [], [19..24]]
[1,2,3,4,5,6,7,8,9,19,20,21,22,23,24]
*Main>concatena []
[]
*Main>concatena [[1,2,3], [4..10], [10,20]]
[1,2,3,4,5,6,7,8,9,10,10,20]
```

7. *Define una función que devuelva el último elemento de una lista.*

   Aquí tenemos unos ejemplos de su ejecución en la terminal:

```
*Main>ultimo [42,4,5,43,23]
23
*Main>ultimo []
*** Exception: lista vacía
CallStack (from HasCallStack):
 error, called at Practica3.hs:78:13 in main:Main
```

   Pista: La lista con exactamente un elemento debe figurar en nuestros casos base.

8. *Define una función que tome dos listas ordenadas ascendentemente y las combine en otra lista que, a su vez, esté ordenada ascendentemente.*

   Veamos algunos ejemplos de su ejecución en terminal:

```
*Main>merge [1,5,7,8] [2,3,4,5,6,9]
[1,2,3,4,5,5,6,7,8,9]
*Main>merge [100,200,300] [1,2]
[1,2,100,200,300]
*Main>merge [] [4,5,7]
[4,5,7]
```

   Pista: Considera utilizar el operador `cons ":"`*, al definir el caso recursivo.*

Recursión con cadenas

Recordemos que las cadenas son listas de caracteres por lo que la dinámica de trabajo en los siguientes dos ejercicios será similar a los anteriores:

9. *Define una función que quite los espacios de una cadena dada.*

   Revisemos un par de ejemplos de su ejecución en terminal:

```
*Main>sinEspacios "hola mundo :)"
"holamundo:)"
*Main>sinEspacios "varios espacios seguidos de numeros      4645"
"variosespaciosseguidosdenumeros4645"
```

10. *Define una función que retire todas las ocurrencias – de un carácter dado en una cadena.*

    Aquí tenemos algunos ejemplos de su ejecución en terminal:

```
*Main>sinChar "hola mundo" 'a'
"hol mundo"
*Main>sinChar çadena de b: bbbbbbbbb" 'b'
çadena de : "
```

Entrega

La entrega es **individual** y consistirá en un archivo comprimido que contenga:

- El archivo `Practica1.hs` con el código indicado en los ejercicios.
- Un archivo readMe.txt que incluya tu nombre, los problemas que tuviste al realizar la práctica, cómo los resolviste y un comentario o sugerencia sobre la práctica.

Sube el archivo comprimido a la plataforma Google Classroom utilizando el siguiente formato para el nombre del comprimido (la extensión puede ser .zip, .tar o cualquier otra):

Practica03_ Apellido1-Apellido2-Nombre(s).zip

Por ejemplo, si Ada Lovelace Turing entregase su práctica le pondría como nombre a su archivo comprimido:

Practica03_ Lovelace-Turing-Ada.zip

Cualquier duda que tengas no dudes en enviarme un correo o escribirla en el grupo de telegram. 😄