Programación Declarativa, 2021-1
Nota de clase 7: Introducción a Haskell*

Manuel Soto Romero

28 de octubre de 2020
Facultad de Ciencias UNAM

En esta nota se presentan los conceptos necesarios para tener un primer acercamiento al lenguaje de programación Haskell, conoceremos la sintaxis básica de los programas y aprenderemos a ejecutarlos. Se presentan también algunas primitivas como los tipos de datos, definición de funciones, asignaciones locales, estructuras de datos, apareamiento de patrones y funciones de orden superior.

7.1. Haskell

Haskell es un lenguaje de programación funcional, cuyas principales características son: [3]

- Sin efectos secundarios: No se puede asignar un valor a una variable y luego cambiarlo.
- Transparencia referencial: Si una función es llamada con los mismos parámetros más de una vez, siempre se obtiene el mismo valor.
- Evaluación perezosa: No se evalúan expresiones hasta que sea estrictamente necesario.
- Tipificado estáticamente: El tipo de una expresión se conoce en tiempo de compilación.
- Inferencia de tipos: No es necesario etiquetar explícitamente cada expresión con un tipo, el sistema de tipos lo puede deducir.

Los lenguajes ubicados dentro de este estilo de programación se basan en mecanismos formales como el Cálculo λ que estudiamos en la Nota de clase 9 y en la Teoría de Categorías que revisaremos más adelante y que se estudian a profundidad en los cursos de Lenguajes de Programación y Álgebra Moderna.

7.2. Tipos básicos

Booleanos (`Bool`)

Representan a las constante lógicas.

*Basada en el *Manual de Prácticas para la Asignatura de Programación Declarativa.*

```
Prelude> True
True
Prelude> False
False
```

Números enteros (`Int`, `Integer`)

El tipo `Int` representa números enteros acotados, en máquinas de 32 bits, el valor máximo es 2147483647 y el valor mínimo es -2147483648. Los números negativos siempre deben escribirse entre paréntesis.

```
Prelude> 1729
1729
Prelude> (-12)
-12
```

El tipo `Integer` representa números enteros no acotados, es decir, se pueden representar números muy grandes.

```
Prelude> 12345678912345678912345678912345678901234567890098765432123456789O
12345678912345678912345678912345678901234567890098765432123456789O
```

Números flotantes (`Float`, `Double`)

El tipo `Float` representa números reales de precisión simple.

```
Prelude> 25.132742
25.132742
```

El tipo `Double` representa números reales de precisión doble.

```
Prelude> 25.132741228718345
25.132741228718345
```

Caracteres (`Char`)

El tipo `Char` representa caracteres, estos se delimitan por comillas simples.

```
Prelude> 'a'
'a'
```

Cadenas (`String`)

El tipo `String` representa secuencias de caracteres delimitadas por comillas dobles.

```
Prelude> ‘‘Hola’’
‘‘Hola’’
```

7.3. Funciones predefinidas

Haskell provee algunos operadores predefinidos en su núcleo para trabajar con los tipos básicos. Estos operadores son en realidad funciones expresadas tanto en forma infija como prefija. Para usar una función en notación prefija se debe colocar el nombre de la función y separar por espacios sus parámetros y para usarla infijamente, se debe colocar la función entre los dos parámetros de la función y delimitar el nombre de la misma entre los símbolos ‘ ‘.

Funciones aritméticas

Se listan las principales funciones usadas en aritméticas.

Nombre	**Símbolo**	**Notación**
Suma	`+`	Infija
Resta	`-`	Infija
Producto	`*`	Infija
División (Entera)	`div`	Prefija
División (Real)	`/`	Infija
Potencia (Entera)	`^`	Infija
Potencia (Real)	`**`	Infija

```
Prelude> 1 + 2
3
Prelude> 5 - 2
3
Prelude> 10 * 2
20
Prelude> 2 ^ 3
8
Prelude> div 13 2
6
Prelude> 13 ‘div‘ 2
6
Prelude> 2.0 ** 3.0
8.0
Prelude> 13 / 2
6.5
```

Funciones lógicas

Se listan los principales operadores usados en lógica matemáticas, pudiéndose definir otros en términos de éstos.

Nombre	Símbolo	Notación
Negación	`not`	Prefija
Disyunción	`||`	Infija
Conjunción	`&&`	Infija

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

Funciones relacionales

Nombre	Símbolo	Notación
Menor	`<`	Infijo
Menor o igual	`<=`	Infija
Mayor	`>`	Infija
Mayor o igual	`>=`	Infija
Igual	`==`	Infijo
Distinto	`/=`	Infijo

```
Prelude> 10 < 20
True
Prelude> 10 <= 10
True
Prelude> 10 > 2
True
Prelude> 2 >= 2
True
Prelude> 3 == 3
True
Prelude> 3 /= 4
True
```

7.4. Definición de funciones

Además de las funciones predefinidas en el lenguaje, es posible definir funciones propias. La sintaxis que se usa para definir estas funciones es muy parecida a la usada en matemáticas. Por ejemplo, en matemáticas se puede definir la siguiente función:

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

Para usar la función (evaluarla), se sustituye el valor de la entrada por un concreto:

$$f(4) = (4) + 2 = 6$$

En Haskell la misma función se puede definir como en el Código 1. Las líneas 1 y 4 representan comentarios de varias líneas y una línea respectivamente, usados para documentar el programa. La línea 2 define un módulo y la línea 5 es la definición como tal de la función.

```
{- Módulo con ejemplos de definición de funciones -}
module Ejemplo where

   -- Función que toma un número y le suma dos.
   f x = x + 2
```

Código 1: Un primer módulo en Haskell

El código escrito en Haskell se coloca en módulos (bibliotecas) que incluyen las definiciones de funciones u otros tipos de datos. Haskell provee diversos módulos para manipular números, cadenas, funciones de estadística y probabildiad, álgebra lineal, listas, entre muchas otras[1]. Por convensión el nombre de un módulo debe ser el mismo que el del archivo donde se definió (con extensión `.hs`).

```
Prelude> :l Ejemplo.hs
[1 of 1] Compiling Ejemplo ( Ejemplo.hs, interpreted )
Ok, 1 module loaded.
*Ejemplo> f 4
6
```

Para hacer una función más robusta, se especifica una firma o contrato, es decir, cuántos parámetros recibe, de qué tipo son éstos y qué valor regresa. Es equivalente a especificar el dominio y contradominio de una función en matemáticas.

Para especificar la firma de una función, se provee el nombre de la función separado de dos símbolos de dos puntos (`::`), un espacio y el tipo de sus parámetros, separando los mismos con una flecha (`->`), el último tipo de dato de la firma representa la salida de la función.

En el Código 2, la línea 5 representa la firma de la función `f` indicando que la función recibe un número entero y regresa otro. La ejecución del código muestra como al intentar pasar otro tipo de dato diferente al de la especificación de la función se obtiene un error.

[1] Consultar `https://hackage.haskell.org/packages/`

```
{- Módulo con ejemplos de definición de funciones -}
module Ejemplo where

   -- Función que toma un número y le suma dos.
   f :: Int - > Int
   f x = x + 2
```

Código 2: Segunda versión del módulo Ejemplo

```
*Ejemplo> f 4.5
<interactive>:2:3: error:
· No instance for (Fractional Int) arising from the literal ‘4.5’
· In the first argument of ‘f’, namely ‘4.5’ In the expression: f 4.5 In an
equation for ‘it’: it = f 4.5
*Ejemplo> f 4
6
```

Ejemplo 7.1. En el Código 3 se define una función para calcular el área total de un cilíndro. La línea 3 especifica la firma de la función indicando que recibe dos parámetros de tipo `Float` y regresa un valor del mismo tipo.

```
-- Función que calcula el área total de un cilíndro dada su altura
-- y diámetro.
areaTotal :: Float → Float → Float
areaTotal h d = 2 * pi * (d/2) * (h + (d/2))
```

Código 3: Función que calcula el área total de un cilíndro

7.5. Asignaciones locales

En el Ejemplo 10.1, se aprecia que el cálculo `d/2` se realiza dos veces. Repetir cálculos en un programa, en ocasiones, resulta ser ineficiente. Para este tipo de situaciones muchos lenguajes de programación funcionales, proveen primitivas para introducir variables con un alcance restringido llamadas *asignaciones locales* o *variables locales*. Este tipo de primitivas, genera expresiones más eficientes con respecto a la evaluación, pues el valor ligado a una variable se calcula una única vez. HASKELL provee dos primitivas para esto: `let` y `where`.

7.5.1. Primitiva `let`

Esta primitiva es considerada una expresión por sí misma pues siempre se evalúa a un valor. Tiene la siguiente sintaxis:

```
let x = e1 in e2
```

Se debe evaluar `e1` cuyo resultado se liga a la variable `x` y evaluar `e2` usando el valor actualizado de `x`. Dicha evaluación es el valor de toda la expresión `let`.

Ejemplo 7.2. En el Código 4 se muestra una segunda versión de la función para calcular el área total de un cilíndro, usando la primitiva `let`. La línea 4 especifica la expresión `let`. Se asigna el valor `d/2` a la variable `r` y se usa en el cuerpo `2 * pi * r * (h + r)`.

```
-- Función que calcula el área total de un cilíndro dada su altura
-- y diámetro.
areaTotal :: Float → Float → Float
areaTotal h d = let r = (d/2) in 2 * pi * r * (h + r)
```

Código 4: Función que calcula el área total de un cilíndro usando `let`

7.5.2. Primitiva `where`

La principal diferencia entre `let` y `where` es que la primera define al inicio las variables y la segunda lo hace después de definir una función. Sin embargo, la diferencia más importante es que `where` forma parte de la sintaxis de una función y puede omitirse o dicho de otra forma, la primitiva `where` sólo puede ser usada dentro de una función y no es considerada una expresión por sí misma.

La sintaxis de una función que usa `where` es la siguiente:

```
<nombre_funcion> <parametro>* = resultado where ...
```

Ejemplo 7.3. El Código 5 muestra una tercera versión de la función para calcular el área total de un cilindro, usando `where`. La línea 5 indica que cada aparición de la variable `r` en la línea 5 debe ser instanciada con `(d/2)`.

```
-- Función que calcula el área total de un cilíndro dada su altura
-- y diámetro.
areaTotal :: Float → Float → Float
areaTotal h d = 2 * pi * r * (h + r)
   where r = (d/2)
```

Código 5: Función que calcula el área total de un cilíndro usando `where`

7.6. Condicionales

Al igual que las funciones matemáticas, en HASKELL éstas pueden ser definidas por partes:

$$|x| = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$$

Para implementar este tipo de funciones, se usan *condicionales.*

7.6.1. Condicional `if`

En HASKELL, los condicionales `if` son considerados expresiones, lo cual quiere decir que siempre deben evaluarse a un valor. Esto implica que si se cumple o no la condición, siempre se debe regresar un valor, con lo cual la parte `else` se vuelve obligatoria.

Las expresiones `if` tienen la siguiente sintaxis:

```
if c then t else f
```

Si la condición `c` es verdadera, se evalúa la expresión `t` y en caso contrario se evalúa la expresión `f`.

Ejemplo 7.4. El Código 6 define una función que calcula el valor absoluto de un número usando `if`. La línea 3 muestra el uso de la primitiva `if`. Si la condición `x >= 0` resulta verdadera, se regresa `x`, en caso contrario, se obtiene `x * (-1)` como resultado.

```
-- Función que calcula el valor absoluto de un número.
absoluto :: Int → Int
absoluto x = if x >= 0 then x else x * (-1)
```

Código 6: Función que calcula el valor absoluto de un número usando `if`

Guardias

Cuando se necesita evaluar más de una condición se pueden usar expresiones `if` anidadas, sin embargo, el código puede perder legibilidad conforme el número de condiciones aumente. Para este tipo de situaciones, se recomienda el uso de *guardias.* Una guardia es una expresión booleana. Si ésta se evalúa a verdadero, entonces se ejecuta la parte de la función correspondiente (función por partes), sin evaluar el resto de las guardias. Si se evalúa a falso, se verifica la siguiente guardia y así en lo sucesivo.

Al igual que `where`, las guardias forman parte de la sintaxis de la definición de una función y no son consideradas expresiones, por lo que no pueden usarse fuera de la definición de las mismas. La sintaxis de las guardias es la siguiente:

```
<nombre_función> <parámetro>*
   | guardia1 = ...
   | guardia2 = ...
   | ...
   | otherwise = ...*
    ...
```

Las guardias son denotadas por barras verticales (|) después del nombre de la función y sus parámetros. Por convención se aplica sangría (indentación) para mantenerlas alineadas. Para especificar un valor por defecto, se usa `otherwise` que es en realidad azúcar sintáctica de `True`, es decir, una guardia que se evalúa siempre a verdadero, es equivalente al `else` de las expresiones `if`.

Ejemplo 7.5. El Código 7 muestra la definición de una función que calcula el valor absoluto de un número usando guardias. Las líneas 4 y 5 especifican las guardias que representan cada paso posible de la función por partes.

```
-- Función que calcula el valor absoluto de un número.
absoluto :: Int → Int
absoluto x
   | x >= 0 = x
   | x < 0  = x * (-1)
```

Código 7: Función que calcula el valor absoluto de un número usando guardias

Ejemplo 7.6. El Código 8 muestra una función que transforma un número a la cadena que representa al mes correspondiente usando guardias. Las líneas 4 a 16 muestran las respectivas guaridas yen particular, la línea 16 especifica el caso por defecto de la función, indicando que la función recibió un número inválido o fuera de rango.

```
-- Función que obtiene el nombre de un mes dado su número.
mes :: Int → String
mes n
   | n == 1 = "Enero"
   | n == 2 = "Febrero"
   | n == 3 = "Marzo"
   | n == 4 = "Abril"
   | n == 5 = "Mayo"
   | n == 6 = "Junio"
   | n == 7 = "Julio"
   | n == 8 = "Agosto"
   | n == 9 = "Septiembre"
    | n == 10 = "Octubre"
```

```
    | n == 11 = "Noviembre"
    | n == 12 = "Diciembre"
    | otherwise = error "Número inválido"
```

Código 8: Función que obtiene el nombre de un mes dado el número que lo representa

7.7. Estructuras de datos

Haskell cuenta con varias bibliotecas para usar estructuras de datos, tales como árboles, pilas, colas, tuplas, tablas de dispersión, entre otras. Sin embargo, las dos estructuras de datos más usadas son las *tuplas* y las *listas* que se incluyen dentro del preludio del lenguaje.

7.7.1. Tuplas

Una *tupla* es una estructura que almacena dos o más datos, no necesariamente del mismo tipo. Se delimitan por paréntesis y sus valores se separan por comas. Una vez que una tupla es definida, su número de elementos (tamaño) no puede cambiar.

Ejemplo 7.7. En el código 9 se muestra la función que resuelve una ecuación de segundo grado por medio de la fórmula general, dados los valores de a, b y c:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
-- Función que resuelve una ecuación de segundo grado.
ecCuad :: Float → Float → Float → (Float,Float)
ecCuad a b c =
   (((b*(-1)) + (sqrt (b**2-4*a*c))) / (2*a),
    ((b*(-1)) - (sqrt (b**2-4*a*c))) / (2*a))
```

Código 9: Función que resuelve una ecuación de segundo grado

```
*Ejemplo> ecCuad 1 7 10
(-2.0,-5.0)
```

Los resultados de la ecuación se regresan en una tupla de tamaño dos llamadas pares (líneas 4 y 5). La ejecución del código muestra el resultado de resolver la ecuación $x^2 + 7x + 10 = 0$ obteniendo los resultado $x_1 = -2.0$ y $x_2 = -5.0$ representados mediante la tupla `(-2.0,-5.0)`.

Algunas funciones que operan sobre pares.

`fst`	Extrae el primer elemento de un par.
`snd`	Extrae el segund elemento de un par.
`swap`	Intercambia los elementos de un par.

7.7.2. Listas

En Haskell, una lista se define recursivamente como sigue:

Definición 7.1. *Una lista es alguna de las siguientes:*

- *La lista vacía es una lista y se representa por* `[]`.
- *Si* `x` *es un elemento de un conjunto A y* `xs` *es una lista con elementos en A, entonces* `x:xs` *es también una lista. Se llama a* `x` *la cabeza de la lista y a* `xs` *el resto.*
- *Son todas.*

En su forma más simple, una *lista* es una estructura que almacena varios datos, todos del mismo tipo. Se delimitan por corchetes y sus valores se separan por comas. Para construir listas se usa la función *cons* (`:`). Una lista de caracteres es en realidad una cadena.

```
[1,2,3,4,5]     1:(2:(3:(4:[])))     ‘‘Ciencias’’     [(1,2),(3,4)]     []
```

Otra forma de especificar listas numéricas es mediante lo que se conoce como *rango*, indicando los primeros y último elemento de la lista o sólo los primeros separados por dos puntos (`..`).

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [0,2..10]
[0,2,4,8,10]
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,...
```

Al indicar más de un elemento al inicio de un rango, Haskell infiere si debe seguir algún patrón para generar la lista, sin embargo, sólo funciona con patrones sencillos, como la lista con los números pares.

Otra forma de representar listas es *por comprensión*, al igual que en los conjuntos matemáticos, a partir de otros ya existentes. Por ejemplo, el conjunto

$$\{2x \ : \ x \in \{1,2,3,4,5\}\}$$

produce el conjunto $\{2,4,6,8,10\}$, es decir, el conjunto de números $2x$, tal que, x es un elemento del conjunto $\{1,2,3,4,5\}$.

En HASKELL se tiene una notación por comprensión similar que se puede usar para construir nuevas listas a partir de otras existentes. Por ejemplo:

```
Prelude> [2*x | x <- [1..5]]
[2,4,6,8,10]
Prelude> [even y | y <- [2..6]]
[True,False,True,False,True]
```

Las expresiones `x <- [1..5]` y `y <- [2..6]` son llamadas *generadores* y se puede tener más de uno dentro de las listas por comprensión, separando éstos por comas. La sintaxis de un generador puede ser alguna de las siguientes:

```
<variable> <- <lista>
(<variable>,<variable>) <- <lista de pares>
(<variable>,<variable>,<variable> <- <lista de tuplas de tamaño 3>
...
```

También se pueden emplear guardias para filtrar los valores producidos por los generadores. Si la guarda es verdadera, entonces se mantiene el valor en la lista, en caso contrario, se descarta.

Ejemplo 7.8. En el Código 10 se define una función que obtiene los factores de un número entero. La lista se genera por comprensión en la línea 3, indicando un generador que toma valores de 1 a n (`[1..n]`) con la condición de que el módulo de cada elemento de la lista sea igual a cero (su residuo es cero y por lo tanto son factores). La ejecución de la función muestra los factores del número 10, la lista por comprensión resultante queda como sigue:

```
[x | x <- [1..10], mod 10 x == 0]
```

```
-- Función que obtiene los factores de un número entero.
factores :: Int → [Int]
factores n = [x | x <- [1..n], mod n x == 0]
```

Código 10: Función que calcula los factores de un número entero

```
*Ejemplo> factores 10
[1,2,5,10]
```

Algunas funciones que operan sobre listas:

`head`	Obtiene la cabeza de la lista.
`tail`	Obtiene el resto de la lista.
`last`	Obtiene el último elemento de la lista.
`init`	Quita el último elemento de la lista.
`length`	Calcula la longitud de la lista.
`null`	Indica si la lista es vacía.
`reverse`	Obtiene la reversa de la lista.
`take`	Toma los primeros n elementos de la lista.
`drop`	Elimina los primeros n elementos de la lista.
`elem`	Indica si un dato pertenece a la lista.

7.8. Recursión

Al igual que en PROLOG, en HASKELL y en la mayoría de los lenguajes declarativos, la recursión es un concepto muy importante pues a diferencia de los lenguajes imperativos, no se cuenta con estructuras de repetición como `while` o `for`. En su lugar, se usa el concepto de recursión para repetir acciones.

Para definir funciones recursivas en HASKELL se debe especificar el caso base de la función y el caso recursiva. Una forma de hacer esto es mediante condicionales.

Ejemplo 7.9. En el Código 11 se muestra la función que calcula el n-ésimo número de la sucesión de Fibonacci. Se usan guardias para especificar el caso base (línea 4) y el caso recursivo (línea 5).

```
-- Función que calcula el fibonacci de un número.
fibonacci :: Int → Int
fibonacci n
   | n < 2 = 1
   | otherwise = fibonacci(n-1) + fibonacci(n-2)
```

Código 11: Fibonacci de un número usando condicionales

7.8.1. Apareamiento de patrones

En el Ejemplo 10.9 se hace uso de condicionales para separar el caso base del recursivo, sin embargo, en la práctica, se prefiere el uso de la técnica de *apareamiento de patrones*[2] pues permite definir funciones de una forma más legible. La técnica de apareamiento de patrones consiste en definir el comportamiento de una función en varias partes mediante los posibles constructores (patrones) para el tipo de dato de los parámetros (números, caracteres, listas, tuplas, etcétera) y a diferencia del uso de condicionales que evalúa valores, esta técnica hace una revisión mediante la estructura de las expresiones correspondientes.

Por ejemplo, los patrones para los números enteros pueden ser 1, 2, 3, ... o alguna variable para indicar que se trata de cualquier número, mientras que los patrones para una lista pueden ser `[]` y `x:xs` que representan a la lista vacía y a la lista con cabeza y resto respectivamente.

También se tiene un patrón especial, llamado *variable anónima* representado mediante el símbolo de guión bajo (_), que funciona sobre cualquier tipo de dato para indicar que no importa la forma (estructura) de un parámetro en particular.

Ejemplo 7.10. El Código 12 muestra la función que calcula el Fibonacci de un número usando apareamiento de patrones. Las líneas 3 y 4 muestran los casos base, indicando que la función con entradas 0 y 1 regresan siempre 1. Por otro lado, el caso recursivo se especifica dando el patrón para cualquier número n (línea 5). La función verifica cada patrón en orden.

```
-- Función que calcula el fibonacci de un número.
fibonacci :: Int → Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci(n-1) + fibonacci(n-2)
```

Código 12: Fibonacci de un número usando apareamiento de patrones

Ejemplo 7.11. En el Código 13 se muestra la función que calcula la suma de los primeros n números naturales. La línea 3 muestra el caso base para el cero y la línea 4 muestra el caso recursivo para cualquier número n.

```
-- Suma los primeros n números naturales.
suma :: Int → Int
suma 0 = 0
suma n = n + suma(n-1)
```

Código 13: Suma de los primeros n naturales

[2]Del inglés *Patter Matching*.

Ejemplo 7.12. En el Código 14 se muestra la función que calcula la longitud de una lista. Notar que la lista puede ser de cualquier tipo, con lo cual se dice que es *polimófica*, más adelante se habla de esto. La línea 3 muestra el caso base para la lista vacía y la línea 4 muestra el caso recursivo para cualquier lista que tenga una cabeza `x` y una cola `xs`.

```
-- Calcula la longitud de una lista.
longitud :: [a] → -> Int
longitud [] = 0
longitud (x:xs) = 1 + (longitud xs)
```

Código 14: Longitud de una lista

Ejemplo 7.13. El Código 15 muestra una función que dado un identificador y un ambiente de evaluación, representado mediante una lista de pares, regresa el valor asociado al mismo. La línea 3 muestra el caso base: para cualquier variable y la lista vacía, se reporta un error al no tener información suficiente para buscar. Las líneas 4 y 5 muestra el caso recursivo: para una variable `i` y una lista formada por una cabeza `(sub_id,value)` y cola `xs`. La cabeza de la lista muestra cómo se trata una lista de pares.

```
-- Función que busca valores en un ambiente de evaluación.
busca :: Char → [(Char,Int)] → Int
busca _ [] = error "Variable libre"
busca i ((sub_id,value):xs)
   | i == sub_id = value
   | otherwise busca i xs
```

Código 15: Búsqueda en un ambiente de evaluación

Referencias

[1] Manuel Soto, *Manual de Prácticas para la asignatura de Programación Declarativa*, Proyecto de Apoyo a la Docencia, Facultad de Ciencias UNAM, 2019.

[2] Miran Lipovaca, *Learn You a Haskell for Great Good! A Beginner's Guide*, Primera edición, No Starch Press, 2011.

[3] Favio E Miranda, Elisa Viso, *Matemáticas Discretas,* Segunda edición, Las Prensas de Ciencias, 2016.