Introducción a los Algoritmos - 2do. cuatrimestre 2015

Guía 4: Expresiones, funciones y listas.

Objetivos

El objetivo de los siguientes ejercicios es introducirnos en la ***programación funcional****, es decir, al desarrollo de programas como funciones.*

La guía incluye también ejercicios de precedencia y tipado, para que puedas comprobar si tus programas están bien escritos. También aprenderás a usar un interprete de `haskell` *, y a desarrollar tus primeros programas.*

Funciones

En matemática, se dice que una magnitud o cantidad es función de otra si el valor de la primera depende exclusivamente del valor de la segunda. Por ejemplo, supongamos que queremos viajar desde Córdoba a Buenos Aires en auto. La duración del viaje T dependerá de la distancia d entre Córdoba y Buenos Aires y de la velocidad v que lleve el auto. Del mismo modo, el área A de un círculo es función de su radio r. Estas magnitudes a veces pueden vincularse a través de la proporcionalidad directa o ser inversamente proporcionales. Así, el área A de un círculo es proporcional al cuadrado de su radio, lo que se expresa con una fórmula del tipo $A = \pi * r^2$ y la duración de un viaje T es inversamente proporcional a la velocidad del vehículo ($T = d/v$). A la variable que se encuentra a la izquierda de estas fórmulas (el área, la duración) se la denomina variable dependiente*, y a las variables de las que depende (el radio, la velocidad, la distancia) se las llama* variables independiente*.*

De manera más abstracta, el concepto general de función se refiere a una regla que asigna a cada elemento de un primer conjunto un único elemento de un segundo conjunto. Por ejemplo, cada número entero posee un único cuadrado, que resulta ser un número natural (incluyendo el cero):

$$\begin{array}{ccccccccc} \ldots & -2 & -1 & 0 & 1 & 2 & 3 & \ldots \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ & +4 & +1 & 0 & +1 & +4 & +9 & \end{array}$$

Esta asignación constituye una función entre el conjunto de los números enteros y el conjunto de los naturales.

Diremos que una ***función*** *le "asigna" a los elementos de un conjunto, llamado* ***dominio****, elementos de otro conjunto, llamado* ***codominio****. La notación:*

$$\texttt{f} : A \to B$$

indica que `f` *es una función con dominio A y codominio B. La definición de una función tiene la forma*

$$\texttt{f}.x \doteq \langle \textit{expresión que depende de } x \rangle$$

donde `f` *es el nombre de la función y x es/son la/s variable/s independiente/s.*

1. En las siguientes definiciones identificá las variables, las constantes y el nombre de la función

   a) $\texttt{f}.x \doteq 5 * x$

   b) $\texttt{duplica}.a \doteq a + a$

   c) $\texttt{por2}.y \doteq 2 * y$

   d) $\texttt{multiplicar}.zz.tt \doteq zz * tt$

2. Escribí una función que dados dos valores, calcule su promedio.

3. Tomando las definiciones del punto 1 evaluá las siguientes expresiones. Justificá cada paso utilizando la notación aprendida. Luego, controlá los resultados en `Haskell`.

   a) $(\texttt{multiplicar}.(\texttt{f}.5).2) + 1$

   b) $\texttt{por2}.(\texttt{duplica}.(3 + 5))$

4. Tomando las definiciones en el punto 2 demostrá que *duplica* y *por*2, si son aplicadas al mismo valor, dan siempre el mismo resultado. En otras palabras, la expresión $duplica.x = por2.x$ es **válida**.

Tipado de funciones

Tomemos la función `g` *definida así:*

$$g.x.y \doteq 3x - x * y > 0$$

Esta función toma dos argumentos de tipo Num *y devuelve un valor de tipo* $Bool$*. Así, el tipo de* `g` *se declara de la siguiente forma:*

$$g : Num \to Num \to Bool$$

Es decir, los tipos de los argumentos se listan primero, siguiendo el orden en el que serán llamados por la función, y en último lugar se coloca el tipo del resultado de evaluar la función.

5. Dar el tipo de las funciones del ejercicio 1 y el ejercicio 2.

6. Dar el tipo de las siguientes funciones:

   a) $\texttt{g}.y \doteq 8 * y$

   b) $\texttt{h}.z.w \doteq z + w$

   c) $\texttt{j}.x \doteq x \leqslant 0$

Definiciones de Funciones por casos

Hay ocasiones en las que una sola fórmula no alcanza para definir una función. Así, existen funciones que para un conjunto de argumentos requieren una definición y para otro conjunto de argumentos necesitan de otra definición diferente. Es el caso, por ejemplo, de la función `espar` *que dado un natural devuelve* $true$ *si el número es par o* $false$ *si el número es impar.*

Una ***definición por casos*** *de una función tendrá la siguiente forma general:*

$$\begin{array}{lcll} \texttt{f}.x & \doteq & (& B_0 \to f_0 \\ & & \square & B_1 \to f_1 \\ & & & \vdots \\ & & \square & B_n \to f_n \\ & &) & \end{array}$$

donde las B_i *son expresiones de tipo booleano, llamadas* ***guardas*** *y las* f_i *son expresiones del mismo tipo que el resultado de* f*. Para un argumento dado el valor de la función se corresponde con la expresión cuya guarda es verdadera para ese argumento.*

Al trabajar con expresiones booleanas se hace necesario incorporar a nuestro formalismo los operadores $\wedge, \vee, \neg$ *que pueden ser utilizados para combinar dos fórmulas para obtener una nueva fórmula. En* `haskell` *estos operadores se escriben* `&&` *,* `||` *y* `not`*, respectivamente.*

7. Definí las funciones que describimos a continuación, luego implementalas en `haskell` . Por ejemplo:

   Enunciado: *signo* : $Int \to Int$, que dado un entero retorna su signo, de la siguiente forma: retorna 1 si x es positivo, -1 si es negativo y 0 en cualquier otro caso.

   Solución:

   $$\begin{array}{lcll} \texttt{signo}.x & \doteq & (& x > 0 \to 1 \\ & & \square & x < 0 \to -1 \\ & & \square & x = 0 \to 0 \\ & &) & \end{array}$$

   En `haskell` se escribe así:

   ```
   sgn :: Int -> Int
   sgn x | x>0   = 1
         | x<0   = -1
         | x==0  = 0
   ```

   a) *entre0y9* : $Int \to Bool$, que dado un entero devuelve *True* si el entero se encuentra entre 0 y 9.

   b) *rangoPrecio* : $Int \to String$, que dado un número que representa el precio de una computadora, retorne "muy barato" si el precio es menor a 2000, "demasiado caro" si el precio es mayor que 5000, "hay que verlo bien" si el precio está entre 2000 y 5000, y "esto no puede ser!" si el precio es negativo.

c) *absoluto* : $Int \to Int$, que dado un entero retorne su valor absoluto.

d) *esMultiplo2* : $Int \to Bool$, que dado un entero n devuelve *True* si n es múltiplo de 2.
Ayuda: usar `mod`, el operador que devuelve el resto de la división.

Tuplas

Una manera de formar un nuevo tipo combinando los tipos básicos es tomar estos de a pares. Por ejemplo: el tipo $(Num, Char)$ *consiste de todos los pares en los que la primera componente es de tipo Num y la segunda de tipo Char. De la misma manera, se pueden construir ternas, cuaternas, etc. En general, hablaremos de tuplas.*

Ejemplo: podemos definir la función raices, que devuelve las raíces de un polinomio de segundo grado, de manera tal que el resultado sea un par, es decir,

raices : $Num \to Num \to Num \to (Num, Num)$

Por ejemplo, los números racionales pueden ser representados por pares de números enteros. La función que suma dos racionales puede ser definida como:

$$sumRat : (Int, Int) \to (Int, Int) \to (Int, Int)$$
$$sumRat.(a, b).(c, d) = (a * d + b * c, b * d)$$

8. Definí las funciones que describimos a continuación, luego implementalas en `haskell`.

a) *segundo3* : $(Int, Int, Int) \to Int$, que dada una terna de enteros devuelve su segundo elemento.

b) *ordena* : $(Int, Int) \to (Int, Int)$, que dados dos enteros los ordena de menor a mayor.

c) *rangoPrecioParametrizado* : $Int \to (Int, Int) \to String$ que dado un número x, que representa el precio de un producto, y un par $(menor, mayor)$ que represente el rango de precios que uno espera encontrar, retorne "muy barato" si x está por debajo del rango, "demasiado caro" si está por arriba del rango, "hay que verlo bien" si el precio está en el rango, y "esto no puede ser!" si x es negativo.

d) *mayor3* : $(Int, Int, Int) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.
Por ejemplo: *mayor3*.$(1, 4, 3) = (False, True, False)$ y *mayor3*.$(5, 1984, 6) = (True, True, True)$

e) *todosIguales* : $(Int, Int, Int) \to Bool$ que dada una terna de enteros devuelva *True* si todos sus elementos son iguales y *False* en caso contrario.
Por ejemplo: *todosIguales*.$(1, 4, 3) =$ *False* y *todosIguales*.$(1, 1, 1) =$ *True*

Listas

Ahora, comenzaremos a complejizar el lenguaje de nuestras ***expresiones*** *agregando* ***listas****. Una lista (o secuencia) es una colección ordenada de valores, que deben ser todos del mismo tipo; por ejemplo,* $[1, 2, 5]$.

Denotamos a la lista vacía con $[\,]$. *El operador* $\triangleright$ *(llamado "pegar" y notado : en Haskell) es fundamental (se lo denomina constructor) ya que permite construir listas arbitrarias a partir de la lista vacía.* $\triangleright$ *toma un elemento* x *(a izquierda) y una lista* xs *y devuelve una lista con primer elemento* x *seguido de los elementos de* xs. *Por ejemplo* $3 \triangleright [\,] = [3]$, *y* $1 \triangleright [2, 3] = [1, 2, 3]$. *Para denotar listas no vacías utilizamos expresiones de la forma* $[x, y, \ldots, z]$, *que son abreviaciones de* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,])$. *Como el operador* $\triangleright$ *es asociativo a derecha, es lo mismo escribir* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,])$ *que* $x \triangleright y \triangleright \ldots \triangleright z \triangleright [\,]$. *Otros operadores sobre listas son los siguientes:*

- $\#$, *llamado cardinal, toma una lista* xs *y devuelve su cantidad de elementos. Ej:* $\#[1, 2, 0, 5] = 4$. *En Haskell* $\#xs$ *se escribe:* `length xs`.
- $!$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una posición, y devuelve el elemento de la lista que se encuentra en la posición* n *(contando a partir de la posición 0). Ej:* $[1, 3, 3, 6, 2, 3, 4, 5] \,!\, 4 = 2$. *Este operador, llamado índice, asocia a izquierda, por lo tanto* $xs \,!\, n \,!\, m$ *se interpreta como* $(xs \,!\, n) \,!\, m$. *En Haskell* $xs \,!\, n$ *se escribe:* `xs !! n`.

- $\uparrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista con los primeros* n *elementos de* xs*. Ej:* $[1,2,3,4,5,6] \uparrow 2 = [1,2]$*. Este operador, llamado tomar, asocia a izquierda, por lo tanto* $xs \uparrow n \uparrow m$ *se interpreta como* $(xs \uparrow n) \uparrow m$*. En Haskell* $xs \uparrow n$ *se escribe:* `take n xs`*.*
- $\downarrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista sin los primeros* n *elementos de* xs*. Ej:* $[1,2,3,4,5,6] \downarrow 2 = [3,4,5,6]$*. Este operador, llamado tirar, se comporta igual al anterior, interpretando* $xs \downarrow n \downarrow m$ *como* $(xs \downarrow n) \downarrow m$*. En Haskell* $xs \downarrow n$ *se escribe:* `drop n xs`*.*
- $+\!\!+$ *toma una lista* xs *(a izquierda) y otra* ys*, y devuelve la lista con todos los elementos de* xs *seguidos de los elementos de* ys*. Ej:* $[1,2,4] +\!\!+ [1,0,7] = [1,2,4,1,0,7]$*. Este operador, llamado concatenación, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como* $xs +\!\!+ ys +\!\!+ zs$*. En Haskell* $xs +\!\!+ ys$ *se escribe:* `xs ++ ys`*.*
- $\triangleleft$ *toma una lista* xs *(a izquierda) y un elemento y devuelve una lista con todos. los elemento de* xs *seguidos por* x *como último elemento. Ej:* $[1,2] \triangleleft 3 = [1,2,3]$*. Este operador, llamado "pegar a izquierda", es asociativo a izquierda, luego es lo mismo* $([\,] \triangleleft z) \ldots \triangleleft y) \triangleleft x$ *que* $[\,] \triangleleft z \ldots \triangleleft y \triangleleft x$*. En Haskell* $xs \triangleleft x$ *se escribe:* `xs++[x]`*.*

Existen además dos funciones fundamentales sobre listas que listamos a continuación.

- `head`*, llamada cabeza, toma una lista* xs *y devuelve su primer elemento. Ej:* `head [1,2,3]` $= 1$*.*
- `tail`*, llamada cola, toma una lista* xs *y devuelve la sublista que resulta de eliminar el primer elemento. Ej:* `tail [1,2,3]` *= [2,3]*

La aplicación de función asocia a izquierda, por lo tanto en general es necesario utilizar paréntesis para que la expresión quede bien tipada. Si se quiere escribir la expresión `tail (tail xs)` *no se pueden eliminar los paréntesis, puesto que* `tail tail xs` *(que se interpreta como* `(tail tail) xs` *no tiene sentido.*

A continuación, listamos los niveles de precedencia de estos operadores. Los que están más arriba tienen mayor precedencia. Cuando hay más de un operador en un nivel de precedencia, es necesario poner paréntesis para evitar la ambigüedad. Por ejemplo $x \triangleright xs \uparrow n$ *se interpreta como* $x \triangleright (xs \uparrow n)$*.*

!, #, head y tail	*índice, cardinal, head y tail*
$\uparrow, \downarrow$	*tomar y tirar elementos de una lista*
$\triangleright, \triangleleft$	*pegar a derecha y pegar a izquierda*
$+\!\!+$	*concatenar dos listas*

El objetivo de los siguientes ejercicios es familiarizarse con el tipo de listas y extender el método para justificar el tipado de expresiones, considerando expresiones más complejas que las que veníamos trabajando.

9. Utilizá las definiciones intuitivas de los operadores de listas para evaluar las siguientes expresiones. Subrayá la subexpresión resuelta en cada paso justificado. Luego usá un intérprete de `haskell` para verificar los resultados. Por ejemplo:

$$\begin{aligned} & [23,45,6] \,!\, \underline{(\mathit{head}\ [1,2,3,10,34])} \\ =\ & \{\ \text{def. de } \mathit{head}\ \} \\ & \underline{[23,45,6] \,!\, 1} \\ =\ & \{\ \text{def. de } !\ \} \\ & 45 \end{aligned}$$

   a) $\#[5,6,7]$

   b) $[5,3,57] \,!\, 1$

   c) $[0,11,2,5] \triangleright [\,]$

   d) $[5,6,7] \uparrow 2$

   e) $[5,6,7] \downarrow 2$

   f) $\mathit{head}.(0 \triangleright [1,2,3])$

En `haskell` los distintos operadores se pueden escribir así:

$\mathit{head}.xs$	`head xs`
$\mathit{tail}.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs \,!\, n$	`xs !! n`

Teniendo en cuenta la definición intuitiva de los operadores de listas de la introducción a esta sección, escribí las reglas de tipado de cada uno de ellos. Por ejemplo, el operador head toma una lista y devuelve el primer elemento de ella. La lista puede contener elementos de cualquier tipo (todos del mismo), ya sean números, valores booleanos, otras listas, etc. Para denotar esta situación utilizamos variables (en mayúsculas). Entonces podemos decir que el operador head toma una lista de tipo $[A]$, donde la variable A representa cualquier tipo $(Num, Bool, [Num], \ldots)$ y devuelve un elemento de esa lista, por lo tanto debe ser un elemento de tipo A. Esto lo escribimos en notación funcional *(izq.) o en* notación de árbol *(der.):*

$$head : [A] \to A \qquad\qquad \frac{head.[A]}{A}$$

10. Decidí si las siguientes expresiones están bien escritas, agregando paréntesis para hacer explícita la precedencia y la asociatividad, y justificando con el árbol del tipado correspondiente. Usá un intérprete de `haskell` para verificar los resultados.

    a) $-45 \triangleright [1, 2, 3]$
    b) $([1, 2] +\!\!+ [3, 4]) \triangleleft 5$
    c) $0 \triangleleft [1, 2, 3]$
    d) $[\,] \triangleright [\,]$
    e) $([1] +\!\!+ [2]) \triangleleft [3]$
    f) $[1, 5, False]$
    g) $head.[[5]]$
    h) $head.[True, False] +\!\!+ [False]$

Apéndice: Traducción del Formalismo Básico a Haskell y Precedencia

Niveles de Precedencia

$E(x := a)$, .	sustitución y evaluación
$\sqrt{}, (\cdot)^2$	raíces y potencias
$*, /$	producto y división
máx, mín	máximo y mínimo
$+, -$	suma y resta
$=, \leqslant, \geqslant$	operadores de comparación
$\neg$	negación
$\vee \wedge$	disyunción y conjunción
$\Rightarrow \Leftarrow$	implicación y consecuencia
$\equiv \not\equiv$	equivalencia y discrepancia

Tipos de los operadores

$-x$	$- : Num \to Num$
x^y	$(\hat{}) : Num \to Num \to Num$
$\sqrt[r]{x}$	$\sqrt{} : Num \to Num \to Num$
$x * y$	$(*) : Num \to Num \to Num$
x/y	$(/) : Num \to Num \to Num$
x máx y	máx : $Num \to Num \to Num$
x mín y	mín : $Num \to Num \to Num$
$x + y$	$(+) : Num \to Num \to Num$
$x - y$	$- : Num \to Num \to Num$
$x > y$	$(>) : Num \to Num \to Bool$
$x \geqslant y$	$(\geqslant) : Num \to Num \to Bool$
$x < y$	$(<) : Num \to Num \to Bool$
$x \leqslant y$	$(\leqslant) : Num \to Num \to Bool$
$x = y$	$(=) : A \to A \to Bool$
$\neg p$	$\neg : Bool \to Bool$
$p \wedge q$	$(\wedge) : Bool \to Bool \to Bool$
$p \vee q$	$(\vee) : Bool \to Bool \to Bool$
$head.xs$	$head : [A] \to A$
$tail.xs$	$tail : [A] \to [A]$
$x \triangleright xs$	$(\triangleright) : A \to [A] \to [A]$
$xs \triangleleft x$	$(\triangleleft) : [A] \to A \to [A]$
$xs \uparrow n$	$(\uparrow) : [A] \to Int \to [A]$
$xs \downarrow n$	$(\downarrow) : [A] \to Int \to [A]$
$xs +\!\!+ ys$	$(+\!\!+) : [A] \to [A] \to [A]$
$\#xs$	$\# : [A] \to Int$
$xs\,!\,n$	$(!) : [A] \to Int \to A$

Operadores en `haskell`

x^2	`x^2`
x^n	`x^n` con n entero
x^p	`x**p`
$\sqrt{x}$	`sqrt x`
$\sqrt[r]{x}$	`x**(1/r)`
x máx y	`x` ‘`max`‘ `y` o bien `max x y`
x mín y	`x` ‘`min`‘ `y` o bien `min x y`
$x \geqslant y$	`x >= y`
$x \leqslant y$	`x <= y`
$x = y$	`x == y`
$x \neq y$	`x /= y`
$\neg p$	`not p`
$p \wedge q$	`p && q`
$p \vee q$	`p || q`
$head.xs$	`head xs`
$tail.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs\,!\,n$	`xs !! n`