Repaso de materias anteriores

Introducción a la programación (’23)

Algoritmo

- Un algoritmo es la descripción de los pasos precisos para resolver un problema a partir de datos de entrada adecuados
- Es la descripción de los pasos a realizar
- Especifica una sucesión de instrucciones primitivas
- Descripción de la solución escrita para humanos

Definir “algoritmo” como secuencia de pasos nos limita a pensar únicamente en el paradigma imperativo. Una definición más general de algoritmo es “La descripción de solución” que vale tanto para el paradigma imperativo como para los demás paradigmas.

Programa

- Descripción de la solución para ser ejecutada en una computadora

En este caso, también es más adecuado hablar de “descripción de solución” que de “secuencia de pasos” pero además, los programas no sólo los escribimos para que los ejecute una computadora sino también para que los entiendan las personas.

Paradigmas de programación

En la programación declarativa quien programa dice qué quiere y la computadora “decide” cómo hacerlo. En este caso, el programa es una especificación.
En la programación imperativa quien programa dice qué hacer y cómo hacerlo y la computadora “obedece”. En este caso, el programa es una secuencia de directivas (no sólo acciones ya que también hay estructuras de control).

Haskell

Lo elemental

Podemos definir funciones y podemos “aplicar” o “evaluar” esas funciones (tratemos de no usar la palabra “llamar”) pasándoles argumentos.
Nota: Cuando hablamos de “funciones” nos referimos a funciones según la definición matemática (una relación entre un dominio y un codominio que asigna valores del primer conjunto al segundo conjunto), no a lo que se le suele llamar “funciones” en el ámbito informático que serían en realidad “procedimientos” (en tanto pueden tener efectos adicionales) y no “funciones”.

Sintaxis

Definición de función:
<NOMBRE> <PARAMS> = <CUERPO>
Siendo NOMBRE el nombre de la función que estamos definiendo (a través de una variable que empieza en minúsculas), PARAMS la lista de parámetros de la función (a partir de variables distintas separadas por espacios) y CUERPO el resultado de la función a partir de los parámetros.

Ejemplo:
```
f x = x + 1
```
Define la función sucesor (la que espera un número y devuelve el número siguiente)

Sintaxis
Aplicación de función:
<NOMBRE> <ARGS>
Siendo NOMBRE el nombre de la función invocada y ARGS los argumentos separados por espacios. La cantidad de argumentos en ARGS debe ser mayor o igual a 1 y menor o igual a la cantidad de parámetros de la función invocada.

Ejemplo:
```
f 5
```
Invoca a la función “f” pasándole como argumento el número 5.

Guardas

Podemos definir funciones partidas usando guardas.

Sintaxis
Definición de función con guardas:
<NOMBRE> <PARAMS> | <COND 1> = <RES 1>
| <COND 2> = <RES 2>
…
| <COND n> = <RES n>
| otherwise = <RES n+1>
Siendo NOMBRE el nombre de la función que estamos definiendo, PARAMS sus parámetros y cada par COND i (la condición del caso) - RES i (el resultado del caso) los distintos casos de la función partida. Siempre debe haber un caso adicional con la condición “otherwise” que sería “en cualquier otro caso”.

Recursión

Si el resultado de una función evaluada en un cierto argumento depende del resultado de la misma función pero evaluada en otro argumento entonces puedo definir la función de forma recursiva. De todas formas, también hay que definirla como función partida ya que no puedo hacer que todos los casos dependan de la misma función. Se necesita (al menos) un caso base. También es importante asegurar que eventualmente la ejecución va a llegar a alguno de los casos base. Haciendo recursión sobre números esto lo puedo asegurar si el llamado recursivo siempre se invoca con un argumento estrictamente menor al recibido (Ojo: esta es una condición suficiente y es como se hace en general pero no es condición necesaria).

Esquema

Esquema de recursión sobre números:

<NOMBRE> <PARAMS> 0 = <CASO_BASE>

<NOMBRE> <PARAMS> n = <F> n (<NOMBRE> <PARAMS> (n-1))

Siendo nombre el nombre de la función que estamos definiendo, PARAMS sus parámetros (excepto el último), CASO_BASE el resultado de la función en el caso base y F el resultado de la función en el caso recursivo (a través de una función que espera como argumentos el número actual y el resultado del llamado recursivo con el número anterior).

Ejercicio 1

```
promedio x y = x * y div 2
máximo x y | x > y = x
           | otherwise = y
factorial 1 = 1
factorial n = factorial (n-1) * n
```

Listas

También podemos hacer recursión sobre listas.

Esquema

Esquema de recursión sobre listas:

<NOMBRE> <PARAMS> [] = <CASO_BASE>

<NOMBRE> <PARAMS> (x:xs) = <F> x (<NOMBRE> <PARAMS> xs)

Siendo nombre el nombre de la función que estamos definiendo, PARAMS sus parámetros (excepto el último), CASO_BASE el resultado de la función en el caso base y F el resultado de la función en el caso recursivo (a través de una función que espera como argumentos la cabeza de la lista y el resultado del llamado recursivo con la cola de la lista). Notar las similitudes entre este esquema y el de recursión sobre números.

Tipos

Cada expresión válida tiene un tipo. Si la expresión no tiene tipo, es inválida. Para algunas expresiones no podemos conocer su tipo exacto, como por ejemplo “[]”. Sin embargo, esa expresión sí tiene tipo y para notarlo usamos variables de tipos.

Alto orden (o “funciones de orden superior”)

En el paradigma funcional, las funciones son un tipo de datos como cualquier otro. Se pueden pasar como argumento a otras funciones y pueden ser devueltas como el resultado de un cómputo.

Ejemplo:

```
const :: a -> b -> a
const x y = x
```

Se puede interpretar de dos formas:

- La función que toma un argumento y devuelve la función constante para ese argumento.
- La función que tomas dos argumentos y devuelve el primero de ellos.

En realidad, cualquier función de más de un parámetro se puede pensar como una función de un único parámetro que devuelve una función de un parámetro menos. Por ejemplo, La suma se suele pensar como una función que toma dos números y devuelve la suma entre ellos pero también se podría pensar como una función que toma un número y devuelve la función que suma ese número a otro.

Ejemplo:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

Esta función toma como argumento una función y devuelve otra función que hace lo mismo pero que espera los argumentos en el orden inverso.

Convenciones de precedencia y asociatividad

Los tipos tienen asociatividad a derecha

`a -> (b->c) = a -> b -> c ≠ (a->b) -> c`

La aplicación tiene asociatividad a izquierda

`f x y = (f x) y ≠ f (x y)`

La aplicación tiene mayor precedencia que los operadores binarios

`f x + y = (f x) + y ≠ f (x + y)`

Los operadores binarios se pueden usar como funciones

`x + y = (+) x y`

Las funciones se pueden usar como operadores binarios

`f x y = x `f` y`

Funciones anónimas

Cuando definimos una función es obligatorio darle un nombre. Pero las demás expresiones podemos referenciarlas sin nombrarlas (podemos usar el símbolo “3” sin necesidad de darle al “3” un nombre). La forma de referenciar una función sin darle un nombre es usando “funciones anónimas”. En haskell se escriben a través de la “notación lambda”.

Sintaxis

Notación lambda:

\<PARAM> -> <CUERPO>

Siendo PARAM el parámetro de la función que estamos denotando (a partir de una variable) y CUERPO el resultado de la función a partir del parámetro.

Ejemplos:

`\x -> 3`	La función constante 3
`\f -> f 0`	Evaluar una función en cero
`\f -> f 0 == 0`	¿Tiene raíz en cero?

Una función de dos parámetros se escribe con notación lambda pensándola como una función de un parámetro que devuelve una función con un parámetro menos.

Ejemplo:
```
maximo3 x y z = maximo2 (maximo2 x y) z
```
En notación lambda: `\x -> \y -> \z -> maximo2 (maximo2 x y) z`