1 INTRODUCCIÓN

El tema de este segundo curso de Metodologías de Programación es la Programación Funcional. En este primer capítulo se presenta un panorama general de este paradigma de programación, con el objetivo de que ustedes puedan responder ¿Porqué es relevante estudiar la programación funcional? Para ello revisaremos conceptos como función, transparencia referencial, funciones de orden superior y recurrencia. Cuando se considere necesario, ilustraremos estos conceptos con código en Ocaml o en Lisp, los lenguajes que utilizaremos en este curso.

1.1 ¿PORQUÉ LA PROGRAMACIÓN FUNCIONAL ES RELEVANTE?

Resulta curioso que cuando explicamos las ventajas de la programación funcional sobre otros paradigmas de programación, lo hacemos en términos negativos, resaltando cuales son las prácticas de otros paradigmas de programación que **no** están presentes en la programación funcional:

- En programación funcional no hay instrucciones de asignación;
- La evaluación de un programa funcional no tiene efectos colaterales; y
- Las funciones pueden evaluarse en cualquier orden, por lo que en programación funcional no hay que preocuparse por el flujo de control.

Siguiendo la propuesta de Hughes [10], este capítulo se centra en presentar las ventajas de la programación funcional en términos positivos. La idea central es que las **funciones de orden superior** y la **evaluación postergada**, son herramientas conceptuales de la programación funcional que nos permiten descomponer problemas más allá del **diseño modular** que inducen otros paradigmas de programación, como la estructurada. Omitir la operación de asignación, los efectos colaterales y el flujo de control son simples medios para este fin.

El capítulo se organiza de la siguiente manera. Primero presentaremos una serie de conceptos muy básicos sobre **funciones puras** y la composición de las mismas como parte del diseño modular de programas funcionales. A continuación profundizaremos en esta idea a través de los conceptos de **interfaz manifiesta**, **transparencia referencial**, **función de orden superior**, y **recurrencia**.

1.2 FUNCIONES PURAS

En matemáticas una función provee un mapeo entre objetos tomados de un conjunto de valores llamado **dominio** y objetos en otro conjunto llamado **codominio** o **rango**.

Ejemplo 1 *Un ejemplo simple de función es aquella que mapea el conjunto de los enteros a uno de los valores en* $\{\mathtt{pos}, \mathtt{neg}, \mathtt{cero}\}$*, dependiendo si el entero es positivo, negativo o cero.*

Llamaremos a esta función signo. *El dominio de* signo *es entonces el conjunto de los números enteros y su rango es el conjunto* {pos, neg, cero}.

Podemos caracterizar nuestra función mostrando explícitamente los elementos en el dominio y el rango y el mapeo que establece la función. A este tipo de caracterizaciones se les llama por **extensión**.

Ejemplo 2 *Caracterización de la función signo por extensión:*

$$
\begin{aligned}
&\vdots \\
\text{signo}(-3) &= \text{neg} \\
\text{signo}(-2) &= \text{neg} \\
\text{signo}(-1) &= \text{neg} \\
\text{signo}(0) &= \text{cero} \\
\text{signo}(1) &= \text{pos} \\
\text{signo}(2) &= \text{pos} \\
\text{signo}(3) &= \text{pos} \\
&\vdots
\end{aligned}
$$

También podemos caracterizar una función a través de reglas que describan el mapeo que establece la función. A esta descripción se le llama por intensión o **intensional**.

Ejemplo 3 *Caracterización intensional de la función signo:*

$$
\text{signo}(x) = \begin{cases} \text{neg} & \textit{si } x < 0 \\ \text{cero} & \textit{si } x = 0 \\ \text{pos} & \textit{si } x > 0 \end{cases}
$$

En la definición intensional de signo/1, x se conoce como el **parámetro formal** de la función y representa cualquier elemento dado del dominio. Como sabemos, 1 es la **aridad** de la función signo, es decir, el número de parámetros formales de la función en cuestión. El cuerpo de la regla simplemente especifica a que elemento del rango de la función, mapea cada elemento del dominio. La regla que define signo/1 representa un conjunto infinito de ecuaciones individuales, una para cada valor en el dominio. Debido a que esta función aplica a todos los elementos del dominio, se dice que se trata de una función **total**. Si la regla omitiera a uno o más elementos del dominio, diríamos que es una función **parcial**.

Ejemplo 4 *La función parcial signo2 indefinida cuando* $x = 0$*:*

$$
\text{signo2}(x) = \begin{cases} \text{neg} & \textit{si } x < 0 \\ \text{pos} & \textit{si } x > 0 \end{cases}
$$

En los lenguajes funcionales **fuertemente tipificados**, como Ocaml [3, 4, 14, 21], el dominio y rango de una función debe especificarse, ya sea explícitamente o bien mediante su sistema de **inferencia de tipos**. En los lenguajes tipificados dinámicamente, como Lisp [6, 7, 18, 22], esto no es necesario. Veamos esta diferencia en el siguiente ejemplo, donde definimos suma/2 en Ocaml, una función que suma sus dos argumentos:

```
1 # let suma x y = x + y ;;
2 val suma : int -> int -> int = <fun>
```

la línea 2 nos dice que $suma/2$ es una función que va de los enteros (int), a los enteros y a los enteros. Esto es, que dados dos enteros, computará un tercero[1]. Ocaml tiene predefinidos una serie de **tipos de datos** primitivos, que incluyen a los enteros. Observen que en este caso, Ocaml ha inferido el tipo de dato de los parámetros de la función y la función misma.

En Lisp, la definición de add sería como sigue:

```
1 CL-USER > (defun suma (x y) (+ x y))
2 SUMA
```

la línea 2 nos dice que $suma$ ha sido definida como función, pero ha diferencia de de la versión en Ocaml, los parámetros formales y la función misma no están restringidos a ser necesariamente enteros. Por ejemplo, la misma función puede sumar números reales. Esto no significa que estas expresiones no tengan un tipo asociado, sino que el tipo en cuestión será definido dinámicamente en tiempo de ejecución.

Revisemos otro ejemplo. Si queremos definir $signo/1$ en Ocaml, antes debemos definir un tipo de datos que represente al rango de la función, en este caso el conjunto $\{pos, cero, neg\}$:

```
1  # type signos = Neg | Cero | Pos ;;
2  type signos = Neg | Cero | Pos
3  # let signo x =
4      if x<0 then Neg else
5      if x=0 then Cero else Pos ;;
6  val signo : int -> signos = <fun>
7  # signo 5 ;;
8  - : signos = Pos
9  # signo 0 ;;
10 - : signos = Cero
11 # signo (-2) ;;
12 - : signos = Neg
```

Observen que la función $signo/1$ está definida como un mapeo de enteros a signos (línea 6). Los paréntesis en la línea 11 son necesarios, pues '$-/1$ es una función y queremos aplicar $signo/1$ a -2 y no a $-$.

En Lisp, la definición de signo no requiere de una declaración de tipo:

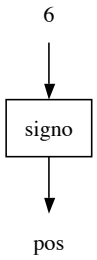
```
1 CL-USER> (defun signo (x)
2             (cond ((< x 0) 'neg)
3                   ((zerop x) 'cero)
4                   (t 'pos)))
5 SIGNO
6 CL-USER> (signo 3)
7 POS
8 CL-USER> (signo -2)
9 NEG
```

1 ¿Saben ustedes qué hace esta función si sólo se le da un entero?

```
CL-USER> (signo 0)
CERO
```

Podemos ver a una función como una caja negra con entradas representadas por sus parámetros formales y una salida representando el resultado computado por la función. La salida obviamente es uno de los valores del rango de la función en cuestión. La elección de qué valor se coloca en la salida está determinada por la regla que define a la función.

Ejemplo 5 *La función signo como caja negra:*



6 aquí se conoce como **parámetro actual** de la función, es decir el valor que se le provee a la función. Al proceso de proveer un parámetro actual a una función se le conoce como **aplicación** de la función. En el ejemplo anterior diríamos que signo se aplica a 6, para expresar que la regla de signo es invocada usando 6 como parámetro actual. En muchas ocasiones nos referiremos al parámetro actual y formal de una función como los **argumentos** de la función. La aplicación de la función signo a 6 puede expresarse en notación matemática:

$$\mathtt{signo}(6)$$

Decimos que está aplicación *evaluó* a pos, lo que escribimos como:

$$\mathtt{signo}(6) \to \mathtt{pos}$$

La expresión anterior también indica que pos es la salida de la caja negra signo cuando se le provee el parámetro actual 6.

La idea de una función como una transformadora de entradas en salidas es uno de los fundamentos de la programación funcional. Las cajas negras proveen bloques de construcción para un programa funcional, y uniendo varias cajas es posible especificar operaciones más sofisticadas. El proceso de "ensamblar cajas" se conoce como **composición** de funciones.

Para ilustrar este proceso de composición, definimos a continuación la función max que computa el máximo de un par de números m y n

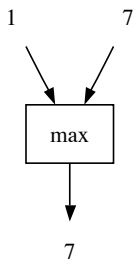
$$\mathtt{max}(m,n) = \begin{cases} m & \text{si } m > n \\ n & \text{en cualquier otro caso} \end{cases}$$

El dominio de max es el conjunto de pares de números enteros y el rango es el conjunto de los enteros. Esto se puede escribir en notación matemática como:

$$max : int \times int \rightarrow int$$

Podemos ver a max como una caja negra para computar el máximo de dos números:

Ejemplo 6 *La función max como caja negra:*



Lo cual escribimos:

$$max(1,7) \rightarrow 7$$

En Ocaml, nuestra función $max/1$ quedaría definida como:

```
1 # let max (m,n) = if m > n then m else n ;;
2 val max : 'a * 'a -> 'a = <fun>
3 # max(1,7) ;;
4 - : int = 7
```

La línea 2 nos dice que $max/1$ es una **función polimórfica**, es decir, que acepta argumentos de varios tipos. Lo mismo puede computar el máximo de un par de enteros, que de un par de reales. Observan también que la aridad de la función es uno, pues acepta como argumento un par de valores numéricos de cualquier tipo. El resultado computado es del mismo tipo que los valores numéricos de entrada. Si queremos estrictamente una función de pares de enteros a enteros podemos usar:

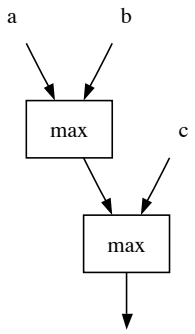
```
1 # let max ((m:int),(n:int)) = if m > n then m else n ;;
2 val max : int * int -> int = <fun>
3 # max(3.4,5.6) ;;
4 Characters 3-12:
5   max(3.4,5.6) ;;
6       ^^^^^^^^^
7 Error: This expression has type float * float but is here used
8        with type int * int
```

Ahora podemos usar max como bloque de construcción para obtener funciones más complejas. Supongamos que requerimos de una función que computa el máximo de tres números, en lugar de sólo dos. Podemos definir está función como $max3$:

$$max3(a,b,c) = \begin{cases} a & \text{si } a \geqslant b \text{ y } a > c \text{ o } a \geqslant c \text{ y } a > b \\ b & \text{si } b \geqslant a \text{ y } b > c \text{ o } b \geqslant c \text{ y } b > a \\ c & \text{si } c \geqslant a \text{ y } c > b \text{ o } c \geqslant b \text{ y } c > a \\ a & \text{en cualquier otro caso} \end{cases}$$

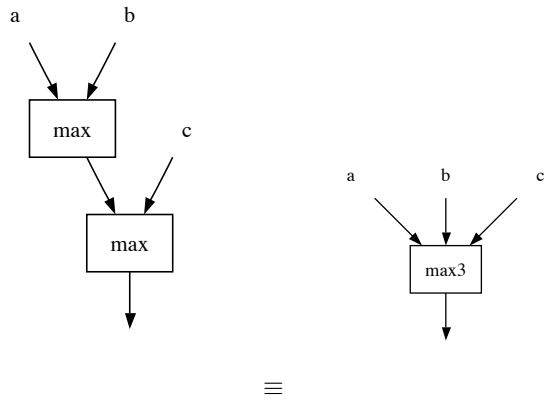
El último caso se requiere cuando $a = b = c$. Esta definición es bastante complicada. Una forma mucho más elegante de definir $max3$ consiste en usar la función max previamente definida.

Ejemplo 7 *La función max3 como composición usando max:*



Lo cual escribimos como $max3(a,b,c) = max(max(a,b),c)$. Podemos tratar a $max3$ como una caja negra con tres entradas.

Ejemplo 8 *La función max3 como caja negra usando la composición de max:*



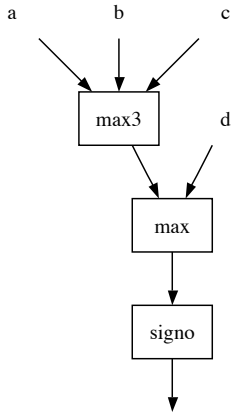
$\equiv$

En Ocaml, la función $max3/1$ se escribiría como:

```
# let max3 (a,b,c) = max(max (a,b), c) ;;
val max3 : int * int * int -> int = <fun>
```

Ahora podemos olvidarnos de los detalles internos de $max3$ y usar esta función como una caha negra para construir nuevas funciones.

Ejemplo 9 *La función signomax4 computa el signo del máximo de 4 números:*



Que escribímos como $signomax4(a, b, c, d) = signo(max(max3(a, b, c), d))$. En Ocaml se definiría como:

```
# let signomax4(a,b,c,d) = signo(max(max3(a,b,c),d)) ;;
val signomax4 : int * int * int * int -> signos = <fun>
```

Así que, dados un conjunto de funciones **predefinidas** o **primitivas**, por ejemplo aritmética básica, podemos construir nuevas funciones en términos de esas primitivas. Luego, esas nuevas funciones pueden usarse para construir nuevas funciones más complejas.

1.3 TRANSPARENCIA REFERENCIAL

La propiedad fundamental de las funciones matemáticas que permite la analogía con los bloques de construcción se llama **transparencia referencial**. Intuitivamente esto quiere decir el valor de una expresión, depende exclusivamente del valor de las sub expresiones que lo componen, evitando así la presencia de **efectos colaterales** propios de lenguajes que presentan **opacidad referencial**.

Una función con referencia transparente tiene como característica que dados los mismos parámetros para su aplicación, obtendremos siempre el mismo resultado. Mientras que en matemáticas todas las funciones ofrecen transparencia referencial, ese no es el caso en los lenguajes de programación. Consideren la función `GetInput()`, su salida depende del lo que el usuario teclee! Multiples llamadas a la función `GetInput` con el mismo parámetro (una cadena vacía), producen diferentes resultados.

Veamos otro ejemplo. Una persona evaluando la expresión $(2ax + b)(2ax + c)$ no se molestaría jamás por evaluar dos veces la sub expresión $2ax$. Una vez que determina que $2ax = 12$, la persona substituirá 12 por ambas ocurrencias de $2ax$. Esto se debe

a que una expresión aritmética dada en un contexto fijo, producirá siempre el mismo valor como resultado. Dados los valores $a = 3$ y $x = 2$, $2ax$ será siempre igual a 12. La trasparencia referencia resulta del hecho de que los operadores aritméticos no tienen memoria, por lo que toda llamada al operador con los mismos parámetros actuales, producirá la misma salida.

¿Porqué es importante una propiedad como la transparencia referencial? Por las matemáticas sabemos lo importante de poder substituir iguales por iguales. Esto nos permite derivar nuevas ecuaciones, a partir de las ecuaciones dadas, transformar expresiones en formas más útiles y probar propiedades acerca de tales expresiones. En el contexto de los lenguajes de programación, la transparencia referencia permite además optimizaciones tales como la eliminación de subexpresiones comunes, como en el ejemplo anterior $2ax$.

Observemos ahora que pasa con lenguajes que no ofrecen transparencia referencial. Consideren la siguiente definición de una pseudofunción en Pascal:

```
1  function F (x:integer) : integer;
2     begin
3        a := a+1;
4        F := x*x;
5     end
```

Debido a que F guarda un registro en a del número de veces que la función ha sido aplicada, no podríamos eliminar la subexpresión común en la expresión $(a + 2 * F(b)) * (c + 2 * F(b))$. Esto debido a que al cambiar el número de veces que F ha sido aplicada, cambia el resultado de la expresión.

1.4 FUNCIONES DE ORDEN SUPERIOR

Otra idea importante en la programación funcional es el concepto de **función de orden superior**, es decir, funciones que toman otras funciones como sus argumentos, o bien, regresan funciones como su resultado. La derivada y la antiderivada en el cálculo, son ejemplos de funciones que mapean a otras funciones.

Las funciones de orden superior permiten utilizar la **técnica de Curry** en la cual una función es aplicada a sus argumentos, uno a la vez. Cada aplicación regresa una función de orden superior que acepta el siguiente argumento. He aquí un ejemplo en Ocaml para la función $suma/2$ en dos versiones. La primera de ellas enfatiza el carácter curry del ejemplo:

```
1  # let suma = function x -> function y -> x+y ;;
2  val suma : int -> int -> int = <fun>
3  # let suma x y = x + y;;
4  val suma : int -> int -> int = <fun>
5  # suma 3 4 ;;
6  - : int = 7
7  # suma 3 ;;
8  - : int -> int = <fun>
```

El tipo de $suma/2$ nos indica que está función toma sus argumentos uno a uno. Puede ser aplicada a dos argumentos, como suele hacerse normalmente (líneas 5 y 6);

pero puede ser llamada con un sólo argumento, !regresando una función en el domino de enteros a enteros! (líneas 7 y 8). Esta aplicación espera un segundo argumento para dar el resultado de la suma.

Las funciones de orden superior pueden verse como un nuevo pegamento conceptual que permite usar funciones simples para definir funciones más complejas. Esto se puede ilustrar con un problema de procesamiento de listas: la suma de los miembros de una lista. Recuerden que una lista es un **tipo de dato recursivo**, la lista es una lista vacía (nil) o algo pegado ($cons$) a una lista:

```
1  listade X ::= nil | cons X (listade X)
```

Por ejemplo: nil es la lista vacía; a veces la lista vacía también se representa como $[]$; la lista $[1]$, es una abreviatura de $cons\ 1\ nil$; y la lista $[1,2,3]$ es una abreviatura de $cons\ 1\ (cons\ 2\ (cons\ 3\ nil))$.

La $sumatoria$ de los elementos de una lista se puede computar con una función recursiva:

$$\begin{aligned} sumatoria\ nil &= 0 \\ sumatoria\ (cons\ num\ lst) &= num + sumatoria\ lst \end{aligned}$$

Si pensamos en cómo programar el $producto$ de los miembros de una lista, observaremos que esa operación y la $sumatoria$ que acabamos de definir, pueden plantearse como un patrón recursivo recurrente general, conocido como $reduce$:

$$sumatoria = reduce\ suma\ 0$$

donde por conveniencia, en lugar de un operador infijo $+$ para la suma, usamos la función $suma/2$ definida previamente:

$$suma\ x\ y = x + y$$

La definición de $reduce/3$ es la siguiente:

$$\begin{aligned} reduce\ f\ x\ nil &= x \\ reduce\ f\ x\ (cons\ a\ l) &= (f\ a)(reduce\ f\ x\ l) \end{aligned}$$

Observen que al definir $sumatoria/1$ estamos usando $reduce/3$ con sólo dos argumentos, por lo que el llamado resulta en una función del argumento restante. En general, una función de aridad n, aplicada a sólo $m < n$ argumentos, genera una función de aridad $n - m$. En el resto del artículo ejemplificare los conceptos con código en Ocaml. Observen la definición de $suma/2$ y las dos llamadas a esta función:

```
1  # let suma x y = x + y;;
2  val suma : int -> int -> int = <fun>
3  # suma 4 5;;
4  - : int = 9
5  # suma 4;;
6  - : int -> int = <fun>
```

la definición de suma/2 (línea 1) nos dice que suma es una función de los enteros, a los enteros, a los enteros (línea 2); esto es, el resultado de computar suma es también un entero, como puede observase en la primera aplicación de la función (línea 3) cuyo resultado es el entero nueve (línea 4). Sin embargo, la segunda llamada a la función (línea 5), da como resultado otra función de los enteros a los enteros (línea 6), esto es, una función del argumento restante. Funciones como la obtenida en esta última llamada reciben el nombre de *curryficadas*. Veamos la definición de reduce/3 y sumatoria/1:

```
1  # let rec reduce f x l = match l with
2    [] -> x |
3    h::t -> f h (reduce f x t) ;;
4  val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
5  # let sumatoria = reduce suma 0 ;;
6  val sumatoria : int list -> int = <fun>
```

la definición de reduce es recurrente, de ahí que se incluya la palabra reservada rec (línea 1) para que el nombre de la función pueda usarse en su misma definición al hacer el llamado recurrente. Esta función recibe tres argumentos una función f, un elemento x y una lista de elementos l. Si la lista de elementos está vacía, la función regresa x; si ese no es el caso, aplica f al primero elemento de la lista (línea 2), y esa función curryficada es aplicada al reduce del resto de la lista (línea 3). La signatura de reduce/3 (línea 4) nos indica que hemos definido una **función de orden superior**, es decir, una función que recibe funciones como argumento. La lectura de los tipos aquí es como sigue: l es una lista de elementos de tipo 'a (cualquiera que sea el caso); x es de tipo 'b; puesto que f se aplica a elementos de la lista y en última instancia a x, se trata de una función de 'a a 'b a 'b; y por tanto el resultado de la función reduce es de tipo 'b. La función suma/2 recibe una lista de enteros y produce como resultado un entero. La inferencia de tipos en este caso es porque hemos introducido 0 en la llamada y Ocaml puede inferir que 'b es este caso es el conjunto de los enteros.

Ahora viene lo interesante, podemos definir el producto de una lista reutilizando reduce:

```
1  # let mult x y = x * y ;;
2  val mult : int -> int -> int = <fun>
3  # let producto = reduce mult 1;;
4  val producto : int list -> int = <fun>
5  # producto [1;2;3;4];;
6  - : int = 24
7  # suma [1;2;3;4];;
8  - : int = 10
```

Una forma intuitiva de entender reduce es considerarla como un transformador de listas que substituye cada cons por el valor de su primer argumento f y cada nil por el valor del segundo x. Así, la llamada a producto[1; 2; 3] es en realidad una abreviatura de

producto cons 1 (cons 2 (cons 3 []))

que reduce convierte a

mult 1 (mult 2 (mult 3 1))

lo cual evalúa a 6.

Veamos otro ejemplo sobre este patrón recurrente aplicado a listas. Podemos hacer explicito el operador *cons* con la siguiente definición:

```
# let cons x y = x :: y;;
val cons : 'a -> 'a list -> 'a list = <fun>
# cons 1 [] ;;
- : int list = [1]
# cons 1 (cons 2 []) ;;
- : int list = [1; 2]
```

cuya signatura nos dice que x debe ser un elemento de un cierto tipo, y que y es una lista de elementos de ese mismo tipo. La salida de la función es la lista construida al agregar x al frente de y, como lo muestran los ejemplos a partir de la línea 3. Ahora que contamos con *cons*/2 podemos definir *append*/2 usando esta definición:

```
# let append lst1 lst2 = reduce cons lst2 lst1;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2] [3;4] ;;
- : int list = [1; 2; 3; 4]
```

O, siguiendo la misma estrategia, podemos definir una función que reciba una lista enteros y regrese la lista formada por el doble de los enteros originales:

```
# let dobleycons num lst = cons (2*num) lst;;
val dobleycons : int -> int list -> int list = <fun>
# let dobles = reduce dobleycons [];;
val dobles : int list -> int list = <fun>
# dobles [1;2;3;4];;
- : int list = [2; 4; 6; 8]
```

Las funciones de orden superior nos permiten definir fácilmente la **composición funcional** o y el **mapeo** sobre listas *map*:

```
# let o f g h = f (g h);;
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let map f = reduce (o cons f) [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

De forma que ahora podemos definir *dobles2*/1 en términos de una **función anónima** y un mapeo:

```
# let dobles2 l = map (fun x -> 2*x) l;;
val dobles2 : int list -> int list = <fun>
# dobles2 [1;2;3;4] ;;
- : int list = [2; 4; 6; 8]
```

o bien, podemos extender nuestro concepto de sumatoria para que trabaje sobre matrices representadas como listas de listas:

```
# let sumatoria_matriz = o sumatoria (map sumatoria);;
val sumatoria_matriz : int list list -> int = <fun>
```

```
# sumatoria_matriz [[1;2];[3;4]];;
- : int = 10
```

1.5 RECURSIVIDAD

Las iteraciones de la programación tradicional, son normalmente implementados de manera recursiva en la programación funcional. Las **funciones recursivas** [15], como hemos visto, se definen en términos de ellas mismas, permitiendo de esta forma que una operación se repita una y otra vez. La recursión a la cola permite optimizar la implementación de estas funciones. La razón por la cual las funciones recursivas son naturales en los lenguajes funcionales, es porque normalmente en ellos operamos con estructuras de datos (tipos de datos) recursivas. Aunque las listas están definidas en estos lenguajes, observen las siguientes definiciones de tipo "lista de" y "arbol de" en Ocaml:

```
# type 'a lista = Nil | Cons of 'a * 'a lista;;
type 'a lista = Nil | Cons of 'a * 'a lista
# type 'a arbolbin = Hoja
                     | Nodo of 'a * 'a arbolbin * 'a arbolbin;;
type 'a arbolbin = Hoja | Nodo of 'a * 'a arbolbin * 'a arbolbin
```

son definiciones de tipos de datos !recursivas! Una lista vacía es una lista y algo pegado a una lista vacía es una lista. Una hoja es un árbol, y un nodo pegado a dos arboles, es un árbol. A continuación definiremos $miembro/2$ para estos dos tipos de datos:

```
# let rec miembro elt lst = match lst with
    | Nil -> false
    | Cons(e,l) -> if e=elt then true else miembro elt l;;
    val miembro : 'a -> 'a lista -> bool = <fun>
# let rec miembro elt arbol = match arbol with
    | Hoja -> false
    | Nodo(e,izq,der) -> if e=elt then true
      else miembro elt izq || miembro elt der;;
      val miembro : 'a -> 'a arbolbin -> bool = <fun>
# miembro 2 (Cons(1,Nil)) ;;
- : bool = false
# miembro 1 (Cons(1,Nil)) ;;
- : bool = true
# miembro 2 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja)));;
- : bool = true
# miembro 4 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja)));;
- : bool = false
```

Los patrones y la recursividad pueden factorizarse utilizando funciones de orden superior, los catamorfismos y los anamorfismos son los ejemplos más obvios. Estas funciones de orden superior juegan un papel análogo a las estructuras de control de la programación imperativa.

1.6 CONSIDERACIONES

Hemos presentado las bondades de la programación funcional para responder a la pregunta de porqué es relevante estudiar este paradigma de programación. Una postura similar para responder a esta pregunta puede encontrarse en el artículo de Hugues [10] *Why Fuctional Programming matters.* Hudak [9] nos ofrece otro artículo introductorio interesante, por la perspectiva histórica que asume, al presentar los lenguajes funcionales de programación. Dos textos donde pueden revisarse los conceptos generales de la programación funcional, son el libro de Field y Harrison [5] y el de MacLennan [13].