Seminario 10

F# vs. C# - Funcional vs. Imperativo

Autores:

Manuel Antonio Vilas Valiente - C311

Jessica Nuñez Hernandez - C312

Jaime Antonio Pérez Selman - C312

F#, características fundamentales

¿Qué es?

F# es un lenguaje de programación multiparadigma de código abierto, para la plataforma .NET, que conjunta la programación funcional con las disciplinas imperativa y orientada a objetos. Es una variante del lenguaje de programación ML y es compatible con la implementación Objective Caml. F# fue inicialmente desarrollado por Don Syme de Microsoft Research, pero actualmente está siendo desarrollado por la División de Desarrolladores de Microsoft y es distribuido como un lenguaje totalmente soportado en la plataforma .NET.

F# es un lenguaje fuertemente tipado que utiliza inferencia de tipos. Como resultado, los tipos no necesitan estar declarados explícitamente por el programador; estos serán deducidos por el compilador durante el proceso de compilación. Sin embargo, F# también permite la declaración explícita de tipos de datos. Por ser un lenguaje .NET, F# soporta los objetos y tipos de .NET; F# permite al programador programar de una manera que se asemeja más a como pensamos. Por ejemplo: en la cafetería para pedir un café, nosotros generalmente no le decimos al mesero exactamente los pasos para hacer el café, solo pedimos un café con ciertas características. De tal forma que tenemos menos espacio para cometer errores, porque simplemente escribimos menos código. También facilita enormemente la creación de código asincrónico y paralelo, cosa que en otros lenguajes de .NET nos llevaría mucho más tiempo.

Sintaxis

Algunas diferencias entre F# y la sintaxis estándar de C

- No se usan corchetes para delimitar bloques de código. En cambio, se usa identación. (Como en Python)
- Para separar parámetros se usa un espacio a diferencia de una coma.

Asignación de valores

F# es un lenguaje de expresiones basadas en evaluación impaciente. Las funciones y expresiones que no retornan ningún valor tienen como tipo de retorno `unit`. F# usa la palabra clave `let` para enlazar valores a nombres. El siguiente ejemplo enlaza el valor `7` al nombre `x`:

```
let x = 3 + 4
```

Tipos

Para definir un nuevo tipo se utiliza la palabra clave `type`. Para una programación funcional, F# provee los tipos *tuple*, *record*, *discriminated union*, *list* y *option*.

`tuple`

Una *tupla* representa una colección de *n* valores. El valor *n* es llamado la aridad de la tupla. Una 3-tuple podría ser representado como `(A, B, C)`, donde A, B y C son valores con posiblemente diferente tipos. Una tupla puede ser usada solamente para almacenar valores cuando el número de valores es conocido en tiempo de diseño y permanece constante durante la ejecución.

`record`

Un *record* es un tipo donde los datos son nombrados, por ejemplo: `{ Name:string; Age:int }`. Los récords pueden ser creados como `{ Name="AB"; Age=42 }`. La palabra clave `with` es usada para crear una copia de un récord, por ejemplo: `{ r with Name="CD" }`, el cual crea un nuevo récord copiando `r` y cambiando el valor del campo `Name` (asumiendo que el récord creado en el ejemplo anterior fue nombrado `r`).

`discriminated union`

Un tipo *discriminated union* es un type-safe versión de las uniones de C. Por ejemplo:

```
type A =
   | UnionCaseX of string
   | UnionCaseY of int
```

Los valores de la unión pueden corresponder a cualquiera de los dos casos de unión. Los tipos de los valores de cada caso de unión es incluido en la definición de cada caso.

`list`

El tipo *list* es una lista enlazada] inmutable representada usando la notación `head::tail` (`::` el operador `cons` o de forma de encabezado corto `[item1; item2; item3]`. Una lista vacía se denota como `[]`.

`option`

El tipo *option* es un tipo de unión discriminada con elección `Some(x)` o `None`. Los tipos de F# pueden ser genéricos implementado como tipos genéricos de .NET.

Estrucutras más comunes

`input` y `output`

```
open System
let a = Console.ReadLine() // input
printfn "%s" a // output
```

Funciones

Las funciones se definen mediante la palabra clave `let` o, si la función es recursiva, mediante la combinación de palabras clave `let rec`.

```
let cylinderVolume radius length =
    // Define la variable local pi.
    let pi = 3.14169
    length * pi * radius * radius

let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

`if...then...else expression`

La expresión `if...then...else` ejecuta diferentes bifurcaciones de código y se evalúa como un valor distinto según la expresión booleana especificada.

```
if x = y then "equals"
elif x < y then "is less than"
else "is greater than"
```

`match expression`

La expresión `match` proporciona control de bifurcación basado en la comparación de una expresión con un conjunto de patrones.

```
// Match expression
match test-expression with
    | pattern1 [ when condition ] -> result-expression1
    | pattern2 [ when condition ] -> result-expression2
    |...

let rec fib n =
    match n with
    | 0 | 1 -> n
    | _ -> fib (n - 1) + fib (n - 2)
```

Evaluación `lazy`

Las expresiones con procesamiento diferido no se evalúan inmediatamente, sino cuando realmente se necesita el resultado. Esto puede ayudar a mejorar el rendimiento del código.

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

Expresiones de secuencia

F# provee *expresiones de secuencia* que define una secuencia `seq { ... }`, lista `[ ... ]` o array `[| ... |]` a través de código que genera valores. Por ejemplo:

```
seq { for b in 0 .. 25 do
          if b < 15 then
              yield b*b }
```

forma una secuencia de los cuadrados de los números de 0 a 14 filtrando afuera por los números que están en un rango de 0 a 25. Las secuencias son generadas a medida que va haciendo falta (i.e. are evaluación lazy), mientras que las listas y arrays son evaluadas impacientemente.

Programación Imperativa

F# soporta la programación imperativa e incluye

- `for`

  `for...in`

  ```
  for pattern in enumerable-expression do
      body expresion
  ```

  `for...to`

```
for identifier = start [to | downto ] finish do
    body expression
```

- `while`

  `while...do`

```
while test-expression do
    body expression
```

- arrays, creados con la sintaxis `[| ... |]`
- hash table, creado con la sintaxis `dict [ ... ]` o `System.Collections.Generic.Dictionary<_,_>` type). Valores y campos de records también pueden ser etiquetados como `mutable`. Por ejemplo:

```
// Define 'x' con valor inicial '1'
let mutable x = 1
// Cambia el valor de 'x' a '3'
x <- 3
```

También, F# soporta acceso a todos los tipos y objetos del CLI como los definidos en:

```
System.Collections.Generic
```

Programación orientada a objetos

F# como recursos para programar orientado objetos incluye:

- dot-notation (e.g. `x.Name`)
- expresiones de objetos (e.g. `{ new obj() with member x.ToString() = "hello" }`)
- construcción de objetos (e.g. `new Form()`)
- tipos tests (e.g. `x :? string`)
- type coercions (e.g. `x :?> string`)
- argumentos (e.g. `x.Method(someArgument=1)`)
- setters (e.g. `new Form(Text="Hello")`)
- argumentos opcionales (e.g. `x.Method(OptionalArgument=1)`

En los patterns para programar con objetos se incluye:

- type tests (e.g. `:? string as s`)
- active patterns, el cual puede ser definido sobre tipos de objetos.[4]

Las definiciones de objetos en F# pueden ser clases, estructuras (struct), interfaz, enum o delegados, correspondiendo a las formas de definición encontradas en C#. Por ejemplo, aquí se muestra una clase con un constructor tomando un nombre y una edad, y declarando dos propiedades.

```
/// Una simple definición de tipo de objeto
type Person(name : string, age : int) =
    member x.Name = name
    member x.Age = age
```

QuickSort

El algoritmo QuickSort hace el ordenamiento basándose en ordenar dos subarrays. La idea se basa en poder particionar el array en dos subarrays, dejando en uno de los subarrays todos los valores menores que cierto valor p (pivote) y en el otro subarray todos los valores mayores que p. Se puede tener, entonces, una ordenación del array original si se ordena el primer subarray y se concatena con el resultado de la ordenación del segundo. Con la concatenación basta, porque ningún elemento de la primera parte será mayor que alguno de la segunda, y a su vez, ningún elemento de la segunda parte será menor que alguno de la primera.

Extraído de "Empezar a programar. Un enfoque multiparadigma con C#"

Analizando los lenguajes basados en paradigmas funcionales, esto se puede hacer de forma más simple y *limpia*. Para ambos lenguajes (F# y Haskell), vamos a tomar como pivote al primer elemento de la lista.

F#

```
// Función dedicada a devolver los menores que un valor en una lista
let rec MenorQP (list: 'a list) (pivote : 'a) : 'a list =
    match list with
    | [] -> []
    | (x::lista) -> if x < pivote then x::MenorQP lista pivote else
MenorQP lista pivote

// Función dedicada a devolver los mayores que un valor en una lista
let rec MayorQP (list : 'a list) (pivote : 'a) : 'a list =
    match list with
    | [] -> []
    | (x::lista) -> if x >= pivote then x::MayorQP lista pivote else
MayorQP lista pivote

// Algoritmo de QuickSort
let rec QuickSort (list : 'a list) : 'a list =
    match list with
    | [] -> []
    | (pivote::lista) -> QuickSort(MenorQP lista pivote) @ [pivote] @
QuickSort(MayorQP lista pivote)
```

```
let lista = QuickSort [55;-22;1;2;3;0]
for num in lista do
    printf "%A " num

printfn ""
```

```
output
-22 0 1 2 3 55
```

Es evidente que el código es muy legible y fácil de escribir, la función recursiva `QuickSort` concatena a ambos lados del pivote los valores menores y los mayores que él, auxiliándose de las funciones `MenorQ` y `MayorQ` para logralo.

Haskell

Ya en *Haskell* tenemos varios códigos con los cuales podemos replicar la implementación anterior, pero este nos brinda la posibilidad de utilizar *List Comprehension*. Con esto es posible, como se nos pedía en la orientación, codificar *QuickSort* en tan solo una línea de código, obteniendo la siguiente difinición:

```
quicksort (x:r) = quicksort [y | y <-r, y < x] ++ [x] ++ quicksort [y |
y <-r, y >= x]
```

A este código, como práctica saludable, se le debe agregar:

```
quicksort :: [Int] -> [Int]
quicksort [] = []
```

Problema de las N reinas

Este problema consiste en encontrar una forma de ubicar *N* reinas de ajedrez en un tablero de *N x N* celdas de manera que ningún par de reinas se amenacen mutuamente. Una estrategia para encontrar una posible solución es "enlazar" a cada Reina una fila, y luego se intenta colocar cada una en una columna de su fila correspondiente, de nos ser posible, se intenta con una combinación diferente. De no encontrar una combinación que solucione el problema, entonces se puede decir que es imposible.

C#

La solución en C# sería:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace NReinas {
```

```
    class Program {
        static void Main(string[] args) {
            Stopwatch s = new Stopwatch();
            s.Start();
            Console.WriteLine(NReinas(8));
            s.Stop();
            Console.WriteLine(s.Elapsed);

        }
        public static bool NReinas(int n) {
            return NReinas(new List<(int, int)>(), 1, n);
        }
        public static bool NReinas(List<(int, int)> puestas, int
columna, int n) {
            if (columna == n + 1) return true;

            for (int fila = 1; fila <= n; fila++) {
                if (PosiblePoner(new Tuple<int, int>(fila, columna),
puestas)) {
                    puestas.Add((fila, columna));
                    if (NReinas(puestas, columna + 1, n))
                        return true;
                    puestas.Remove((fila, columna));
                }
            }
            return false;
        }
        public static bool PosiblePoner(Tuple<int, int> nPos, List<(int,
int)> reinas) {
            foreach (var pReina in reinas) {
                int fila = pReina.Item1;
                int columna = pReina.Item2;

                if (SeAmenazan(fila, columna, nPos.Item1, nPos.Item2))
return false;
            }
            return true;
        }
        public static bool SeAmenazan(int x_1, int y_1, int x_2, int
y_2) {
            return FilaColumna(x_1,y_1,x_2,y_2) ||
Diagonal(x_1,y_1,x_2,y_2);
        }
        public static bool Diagonal(int x_1, int y_1, int x_2, int y_2)
{
            return x_1 - y_1 == x_2 - y_2 || x_1 + y_1 == x_2 + y_2;
        }
        public static bool FilaColumna(int x_1, int y_1, int x_2, int
y_2) {
```

```
            return x_1 == x_2 || y_1 == y_2 || x_1 - y_1 == x_2 - y_2;
        }

    }
}
```

```
Time elapsed: 00:00:00.0212292
```

F#

Veamos ahora una solución en programación funcional:

```
open System
let crono = System.Diagnostics.Stopwatch()

// Revisa si dos reinas se amenzan en sus filas o columnas
let FilaColumna (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    x_1 = x_2 || y_1 = y_2

// Revisa si dos reinas se amenazan diagonalmente
let Diagonal (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    (x_1 - y_1) = (x_2 - y_2) || (x_1 + y_1) = (x_2 + y_2)

// Comprueba si las reinas se amenzan
let SeAmenazan (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    FilaColumna (x_1, y_1) (x_2, y_2) || Diagonal (x_1, y_1) (x_2, y_2)

// Comprueba si es posible poner una reina en una posicion
let rec PosiblePoner nPos puestas : bool =
    match puestas with
    | [] -> true
    | x::xs -> if SeAmenazan x nPos then false else PosiblePoner nPos xs

// NReinasB función encargada de resolver el problema
// recibe la cantidad de reinas, el tablero con reinas puestas,
// y la posicion actual (x,y)
let rec NReinasB (n : int) puestas (x: int) (y: int) : bool =
    if y > n then true elif x > n then false else
        (PosiblePoner (x,y) puestas && NReinasB n ((x,y)::puestas) 1
(y+1)) ||
        NReinasB n puestas (x+1) y

// NReinas recibe n (cantidad de reinas y tablero nxn)
let NReinas (n: int) : bool  = NReinasB n [] 1 1

crono.Start()
System.Console.WriteLine(NReinas (8))
crono.Stop()
System.Console.WriteLine("Time elapsed: {0}", crono.Elapsed);;
```

```
crono.Reset()
```

```
Time elapsed: 00:00:00.0118618
```

Como se puede apreciar, la solución en *F#* es el doble de rápida que en *C#*. Las diferencias entre las implementaciones no son muy notables. Ahora, destaca que en *F#*, al las listas ser inmutables, en cada paso en el que se agrega una nueva reina al tablero, la lista anterior debe ser duplicada. Este gasto de memoria no sucede en *C#*, lo cual le da ventaja en este aspecto.

Otro problema al que nos enfrentamos es a la extensibilidad de nuestro código. Ambas soluciones utilizan el mismo algoritmo y las mismas funciones para darle solución al problema, pero a la realizada en el lenguaje funcional sería más difícil agregarle cambios sencillos como cambiar el tipo de retorno, para devolver la lista de tableros que constituyen respuestas correctas, por ejemplo.

Recursión de Cola

La recursividad de cola (traducción libre de *tail recursion*) es un mecanismo que permite tener **funciones recursivas sin temer por posibles desbordamientos de pila**. A diferencia de la recursividad normal, donde cada llamada recursiva implica la creación de un nuevo *frame* en la pila de llamadas (y por lo tanto el peligro de desbordar el tamaño de la pila), en la *tail recursion* es posible realizar dichas llamadas recursivas reaprovechando el *frame* de pila anterior y evitando así el desbordamiento.

Mostremos un ejemplo:

```
let rec factorial =
    function
    | x when x > 1 -> x * factorial (x-1)
    | _ -> 1
```

Esta es una función recursiva típica, donde si, por ejemplo, intentamos hallar el factorial de `Int32.MaxValue` el resultado será `Process is terminated due to StackOverflowException`.

¿Cómo resolvemos este problema con la recursión de cola?

```
let factorial' =
    let rec fact acc =
    function
    | x when x > 1 -> fact (acc*x) (x-1)
    | _ -> 1
    fact 1
```

Cómo podemos ver ahora hay una nueva función dentro de la definición de `factorial'`. Esta función recibirá el valor del acumulador, que será `1` y el parámetro `n`. De esta forma retornamos directamente la función sin hacer ninguna operación `(x*)` sobre la función.

Por el contrario, si queremos aplicar esta tipo de recursión en lenguajes como *C#* veremos que no hay ninguna optimización y el *call stack* se irá llenando con cada llamada a la función.

Aplicando las modificaciones nuestro código quedaría así:

```
open System
let crono = System.Diagnostics.Stopwatch()

// Devuelve el primer elemento de una lista
let Principio (list : 'a list) : 'a =
    match list with
    | [] -> (-1, -1)
    | (p::resto) -> p

// Retorna la lista sin su primer elemento
let Resto (list : 'a list) : 'a list =
    match list with
    | [] -> []
    | (p::resto) -> resto

// Revisa si dos reinas se amenzan en sus filas o columnas
let FilaColumna (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    x_1 = x_2 || y_1 = y_2

// Revisa si dos reinas se amenazan diagonalmente
let Diagonal (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    (x_1 - y_1) = (x_2 - y_2) || (x_1 + y_1) = (x_2 + y_2)

// Comprueba si las reinas se amenzan
let SeAmenazan (x_1 : int, y_1 : int) (x_2 : int, y_2: int) : bool =
    FilaColumna (x_1, y_1) (x_2, y_2) || Diagonal (x_1, y_1) (x_2, y_2)

// Comprueba si es posible poner una reina en una posicion
let rec PosiblePoner nPos puestas : bool =
    match puestas with
    | [] -> true
    | x::xs -> if SeAmenazan x nPos
                    then false
               else PosiblePoner nPos xs

// NReinasB función encargada de resolver el problema
// recibe la cantidad de reinas, el tablero con reinas puestas,
// y la posicion actual (x,y)
```

```
let rec NReinasB (n : int) puestas (x: int) (y: int) : bool =
    if y > n
        then true
    elif x > n && puestas = []
        then false
    elif x > n
        then
            let lResto = Resto puestas
            let (a, b) = Principio puestas
            NReinasB n (lResto) (a + 1) (y-1)
                elif PosiblePoner (x,y) puestas
                    then NReinasB n ((x,y)::puestas) 1 (y+1)
                else NReinasB n puestas (x+1) y

// NReinas recibe n (cantidad de reinas y tablero nxn)
let NReinas (n: int) : bool  = NReinasB n [] 1 1

crono.Start()
System.Console.WriteLine(NReinas (8))
crono.Stop()
System.Console.WriteLine("Time elapsed: {0}", crono.Elapsed);;
crono.Reset()
```

Árbol Binario de Búsqueda

Un árbol binario de búsqueda (ABB) cuenta en cada nodo con un valor que se denomina *llave*, que tiene que ser de tipo `IComparable` (basado en el cual se realizarán las búsquedas); también puede tener un valor opcional, que puede ser de cualquier tipo `T`. Los ABB sirven para representar conjuntos y diccionarios, y sobre ellos se pueden definir operaciones de búsqueda, mínimo, máximo, predecesor, sucesor, inserción y eliminación.

F#

```
[<AllowNullLiteral>]
type Nodo (valor, hijoIzq : Nodo, hijoDer : Nodo) =
    member this.valor = valor
    member this.hijoIzq = hijoIzq
    member this.hijoDer = hijoDer

let rec Agregar (nodo : Nodo) (nValor : IComparable) : Nodo =
    match nodo with
    | null -> Nodo(nValor, null, null)
    | nodo when nodo.valor < nValor -> Nodo(nodo.valor, nodo.hijoIzq,
Agregar nodo.hijoDer nValor)
```

```
    | nodo when nodo.valor > nValor -> Nodo(nodo.valor, Agregar
nodo.hijoIzq nValor,  nodo.hijoDer)
    | nodo when nodo.valor = nValor -> Nodo(nodo.valor, nodo.hijoIzq,
 nodo.hijoDer)

let rec Contiene (nodo : Nodo) (valor : IComparable) : bool =
    match nodo with
    | null -> false
    | nodo when nodo.valor < valor -> Contiene nodo.hijoDer valor
    | nodo when nodo.valor > valor -> Contiene nodo.hijoIzq valor
    | nodo when nodo.valor = valor -> true
```

C#

```
using System;
using System.Collections;

namespace BinaryTreeCS {
    class Program {
        public static void Main(string[] args) {
            return;
        }
        class ABB {
            int value;
            ABB left;
            ABB right;
            public ABB(int value) {
                this.value = value;
                this.left = null;
                this.right = null;
            }
            public ABB(int value, ABB left, ABB right) {
                this.value = value;
                this.left = left;
                this.right = right;
            }
            public void Add(int v) {
                ABB node = SearchNode(v, this);
                if (node == null)
                    this.value = v;
                else
                    if (node.value == v) return;
                if (v < node.value)
                    node.left = new ABB(v);
                else node.right = new ABB(v);

            }
            public bool Contains(int v) {
                ABB node = SearchNode(v, this);
```

```
                if (node != null && node.value == v) return true;
                return false;
            }
            private ABB SearchNode(int x, ABB a) {
                if (a == null) return a;
                if (x < a.value && a.left != null) {
                    a = a.left;
                    return SearchNode(x, a);
                }
                if (x > a.value && a.right != null){
                    a = a.right;
                    return SearchNode(x, a);
                }
                return a;
            }
        }
    }
}
```

Es apreciable que la implementación realizada en *C#* es menos legible y menos extensa que la realizada en *F#*. Pero nuevamente la inmutabilidad nos trae problemas, la función `Agregar` en *F#* crea una nueva copia del árbol cada vez que necesita moverse por uno de los hijos del nodo actual en que nos encontremos. En *C#* no sucede esto, por tanto en uso de memoria vuelve a tener ventaja.

Función Factorial (operador !)

En el lenguaje *F#* es posible definir operadores, para la resolución del problema asignado definiremos el operador `!` para usarlo como función factorial.

F#

```
let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

let (!) n = factorial (n)
printfn $"Factorial of 5 is: %d{! 5}"
```

```
output
Factorial of 5 is: 120
```

La palabra reservada "rec"

Las funciones recursivas (funciones que se llaman a sí mismas) se identifican explícitamente en el lenguaje F# con la palabra clave `rec`. La palabra clave `rec` hace que el nombre ligado al `let` esté disponible en su cuerpo.. La sintaxis que utiliza es la siguiente:

```
// Recursive function:
let rec function-nameparameter-list =
    function-body
```