1 Programación Funcional

1.1 Ejercicios sobre proyectos

1.1.1 Ejercicio 1

Con los siguientes datos, definidos en el `Ejercicio 4` del `Proyecto 2`

```
-- Sinonimos de tipo
type Altura = Int
type NumCamiseta = Int
-- Tipos algebraicos sin parametros (aka enumerados)
data Zona = Arco | Defensa | Mediocampo | Delantera
data TipoReves = DosManos | UnaMano
data Modalidad = Carretera | Pista | Monte | BMX
data PiernaHabil = Izquierda | Derecha
-- Sinonimo
type ManoHabil = PiernaHabil
-- Deportista es un tipo algebraico con constructores parametricos
data Deportista = Ajedrecista                          -- Constructor sin argumentos
    | Ciclista Modalidad                               -- Constructor con un
      argumento
    | Velocista Altura                                 --  Constructor con un
      argumento
    | Tenista TipoReves ManoHabil Altura               -- Constructor con tres
      argumentos
    | Futbolista Zona NumCamiseta PiernaHabil Altura -- Constructor con ...
```

1. Definir las siguientes funciones:
   - `esDeportista :: Deportista -> Bool` que dado un deportista, nos dice si es un deportista.
   - `esCiclista :: Deportista -> Bool` que dado un deportista, nos dice si es un ciclista.
   - `esVelocista :: Deportista -> Bool` que dado un deportista, nos dice si es un velocista.
   - `esTenista :: Deportista -> Bool` que dado un deportista, nos dice si es un tenista.
   - `esFutbolista :: Deportista -> Bool` que dado un deportista, nos dice si es un futbolista.
2. Definir las siguientes funciones:
   - `contarCiclistas :: [Deportista] -> Int` que dado una lista de deportistas, nos dice cuantos ciclistas hay.
   - `cuantosVelocistasAltos :: [Deportista] -> Int -> Int` que dado una lista de deportistas y una altura, nos dice cuantos velocistas hay de esa altura o mayor.
   - `contarTenistasDiestros :: [Deportista] -> Int` que dado una lista de deportistas, nos dice cuantos tenistas diestros hay.
   - `cuantosZurdosDefensores :: [Deportista] -> Int` que dado una lista de deportistas, nos dice cuantos zurdos defensores hay.

Nota: Para resolver este ejercicio, utilizar pattern matching y no utilizar igualdad de tipos.

1.1.2 Ejercicio 2

En el ejercicio 6 del `Proyecto 2`, definimos a la función `primerElemento` como:

```
primerElemento :: [a] → Maybe a
primerElemento [] = Nothing
primerElemento (x:_) = Just x
```

Definir ahora las siguientes funciones:

- `tercerElemento :: [a] -> Maybe a` que dado una lista, nos devuelve el tercer elemento de la lista si es que existe.
- `ultimoElemento :: [a] -> Maybe a` que dado una lista, nos devuelve el último elemento de la lista si es que existe.
- `elementoEn :: Int -> [a] -> Maybe a` que tome una lista y un índice, y devuelva `Just x` donde `x` es el elemento en la posición indicada, o Nothing si el índice está fuera de rango.
- `maximo :: Ord a => [a] -> Maybe` que devuelva el máximo elemento de una lista no vacía, o `Nothing` si la lista es vacía.
- `filtrarYTomar :: (a -> Bool) -> [a] -> Maybe a` que tome una función de predicado y una lista, y devuelva `Just x` donde `x` es el primer elemento que cumple el predicado, o `Nothing` si ningún elemento lo cumple.

1.1.3 Ejercicio 3

Con el tipo definido

```
data Cola = VaciaC | Encolada Deportista Cola
```

Definir las siguientes funciones:

- `esVacia :: Cola -> Bool` que dado una cola, nos dice si está vacía.
- `longitud :: Cola -> Int` que dado una cola, nos devuelve la cantidad de deportistas encolados.
- `filtrarCola :: (Deportista -> Bool) -> Cola -> Cola` que dado una función de predicado y una cola, nos devuelve una cola con los deportistas que cumplen el predicado.

1.1.4 Ejercicio 4

Con el tipo definido como

```
data Arbol a = Hoja | Rama (Arbol a) a (Arbol a)
```

y teniendo en cuenta el ejercicio 10 del `Proyecto 2++` sobre árboles binarios, definir las siguientes funciones:

- `a_eliminar :: Ord a => a -> Arbol a -> Arbol a` que tome un elemento y un árbol de búsqueda binaria y devuelva un nuevo árbol que no contenga el elemento.
- `a_espejo :: Arbol a -> Arbol a` que tome un árbol y devuelva su espejo, es decir, un árbol con todas las ramas reflejadas.
- `a_hojaMasProfunda :: Arbol a -> Maybe a` que devuelva el valor almacenado en la hoja más profunda del árbol.

1.2 Ejercicios extra

1.2.1 Ejercicio 1

Programar la función

```
estaEnDNI :: Int → Bool
```

que dado un número, devuelve `True` si es una de las cifras de tu DNI.

1.2.2 Ejercicio 2

Programa mediante composición de funciones la función

```
sumaDNI :: [Int] → Int
```

que dada una lista de enteros `xs` suma solo los elementos que son cifras de tu DNI.

Nota: Para los ejercicios 1 y 2, el DNI no se toma como entrada de datos, sino que se utiliza directamente. Para el ejercicio 2, se puede utilizar la función `sumatoria'` definida en el `Proyecto 1`.

1.2.3 Ejercicio 3

Programar la función

```
suma_multiplos :: [Int] → Int → Int
```

que dada una lista de enteros `xs` y un entero `n`, devuelve la suma de los elementos de `xs` que son múltiplos de `n`. La función se debe programar por composición, **sin utilizar recursión**, eligiendo una de estas alternativas:

- Usar `filter` y `sumatoria` definida en el `Proyecto 1`.

```
sumatoria :: [Int] → Int
```

- Usar `sumatoria'` definida en el `Proyecto 1`.

```
sumatoria' :: [a] → (a → Int) → Int
```

En ambos casos se pueden definir las funciones auxiliares que consideren necesarias.

1.2.4 Ejercicio 4

Se va a construir una base de datos de estudiantes de la facultad. Para ello se deben definir los tipos de datos:

- Tipo `Carrera`: es un tipo que tiene cuatro constructores sin parámetros que son `Matematica`, `Astronomia`, `Fisica` y `Computacion`.
- Tipo `Nombre`: es un sinónimo de `String`.
- Tipo `Legajo`: es un sinónimo de `Int`.

Nota: El tipo `Carrera` no debe pertenecer a la clase `Eq`.

Programar utilizando recursión y pattern matching las siguientes funciones:

- `esDeComputacion :: Carrera -> Bool` que dado una carrera, nos dice si es de computación.
- `esDeFisica :: Carrera -> Bool` que dado una carrera, nos dice si es de física.
- `esDeAstronomia :: Carrera -> Bool` que dado una carrera, nos dice si es de astronomía.
- `esDeMatematica :: Carrera -> Bool` que dado una carrera, nos dice si es de matemática.

Y luego la función

```
buscar :: [Estudiante] → Carrera → [Nombre]
```

que dada una lista de estudiantes `xs` y una carrera `c`, devuelve los nombres de los estudiantes en `xs` que estudian la carrera `c`.

2 Programación Imperativa

2.1 Ejercicios sobre proyectos

2.1.1 Ejercicio 1

En el ejercicio 6 del `Proyecto 3`, definimos la función `int pedir_entero (void)` como:

```
int pedir_entero(char name){
    int x;
    printf("Ingrese el valor para la variable %c: ", name);
    scanf("%d", &x);
    return x;
}
```

Defina ahora las siguientes funciones:

- `float pedir_flotante (char name)` que pida un número flotante.
- `char pedir_caracter (char name)` que pida un caracter.
- `void imprimir_mensaje (char* mensaje)` que imprima un mensaje.

2.1.2 Ejercicio 2

En el ejercicio 3 del `Proyecto 4` definimos a la función `int suma_hasta(int n)` como

```
int suma_hasta(int n){
    assert(n > 0);
    int suma = 0;
    while(n > 0){
        suma = n + suma;
        n--;
    }
    printf("La suma es: %d", suma);
    return suma;
}
```

Defina ahora las siguientes funciones:

- `int suma_hasta_par(int n)` que sume los números pares hasta `n`.
- `int suma_hasta_impar(int n)` que sume los números impares hasta `n`.
- `int suma_hasta_multiplo(int n, int m)` que sume los múltiplos de `m` hasta `n`.

2.1.3 Ejercicio 3

En el ejercicio 7 del `Proyecto 4`, definimos la función `int sumatoria(int tam, int a[])` como

```
int sumatoria(int tam, int a[]){
    int suma = 0 ;
    int i = 0;
    while(i < tam){
    suma = a[i] + suma;
    i = i + 1;
  }
  return suma;
}
```

Defina ahora las siguientes funciones:

- `int sumatoria_pares(int tam, int a[])` que sume los números pares de un arreglo.
- `int sumatoria_impares(int tam, int a[])` que sume los números impares de un arreglo.
- `int sumatoria_multiplos(int tam, int a[], int m)` que sume los múltiplos de `m` de un arreglo.

Y luego la función

```
productoria(int tam, int a[])
```

que devuelva el producto de los elementos de un arreglo.

2.2 Ejercicios extra

2.2.1 Ejercicio 1

Definir la función

```
int factorial(int n)
```

que devuelva el factorial de un número `n`.

2.2.2 Ejercicio 2

Definir la función

```
int fibonacci(int n)
```

que devuelva el n-ésimo número de la sucesión de Fibonacci.

2.2.3 Ejercicio 3

Hacer un programa que verifique si el elemento ubicado en el índice `k` de un arreglo `a` es máximo o si es el mínimo. Para ello programar la siguiente función:

```
struct s_minmax_t verificar_minmax(int a[], int tam, int k);
```

donde la estructura `struct s_minmax_t` es

```
struct s_minmax_t {
    bool es_maximo;
    bool es_minimo;
};
```

La función toma un arreglo `a[]`, su tamaño `tam` y un índice `k` y debe devolver una estructura con los dos booleanos que respectivamente indican:

- Todos los elementos de `a[]` son menores o iguales que `a[k]`. (se guarda en `es_maximo`)

- Todos los elementos de `a[]` son mayores o iguales que `a[k]`. (se guarda en `es_minimo`)

La función debe implementarse con un solo ciclo y **no debe mostrar mensajes por pantalla ni pedir valores al usuario**.

En la función main se debe solicitar al usuario ingresar un arreglo de longitud `N`, donde `N` debe definirse como una constante. **El usuario no debe elegir el tamaño del arreglo.** Luego se debe pedir el índice `k` y verificar con `assert` que `k` es un número mayor que `0` y menor que `N`. Finalmente mostrar el resultado de la función `verificar_minmax` por pantalla (los dos valores de la estructura).