
# Introducción a la Programación - Práctica 11

## Ejercicios Integradores

**CONSEJOS:**

- Leer el enunciado en su totalidad y pensar en la forma de resolverlo **ANTES** de empezar a escribir código.
- Si un ejercicio no sale, se puede dejar para después y continuar con los ejercicios que siguen.
- Los ejercicios están pensados para ser hechos después de haber mirado la teórica correspondiente, y esta después de las actividades de indagación.
- Los ejercicios están tomados de las guías prácticas utilizadas en la materia de Introducción a la Programación de la Universidad Nacional de Quilmes por Pablo Ernesto “Fidel” Martínez López y su equipo, Introducción a la Programación de la Universidad Nacional de Hurlingham de Alan Rodas Bonjour y su equipo, de ejercicios y actividades realizadas por Federico Aloi y Miguel Miloro, a su vez basada en las guías Ejercicios de Introducción a la Programación del CIU General Belgrano, elaboradas por Carlos Lombardi y Alfredo Sanzo, y Fundamentos de la Programación del Proyecto Mumuki. Agradecemos a todos los que nos ayudaron con su inspiración.
- **Realizar EN PAPEL los ejercicios que así lo indiquen.**
- **Si un ejercicio indica BIBLIOTECA significa que será útil para la realización de futuros ejercicios tanto en esta guía como en las siguientes, pudiendo ser utilizado sin tener que volver a definirlo. Es útil mantener registro de dichos procedimientos en su carpeta.**

**EJERCICIOS:**

# Parte 1 - Gobsmart

Los directivos del supermercado **`GobsMart`** nos piden modelar el funcionamiento de las cajas de cobro en una sucursal. Para esto usaremos los siguientes tipos:

```
type TipoDePago is variant {
  /* PROP: modelar Tipos de pago aceptados  */
  case Tarjeta  {}
  case MartPago {}
  case Efectivo {}
}

type Producto is record{
  /* PROP: modelar productos
     INV.REP.: precio > 0   */
  field nombre      // String
  field marca       // String
  field precio      // Número
}

type Cliente is record {
  /* PROP: modelar clientes */
  field dni         // String
  field tipoDePago  // TipoDePago
  field productos   // [Producto]
}

type Caja is record {
  /* PROP: modelar cajas
     INV.REP.: número > 0
          facturado >= 0    */
  field número              // Número
  field clientesEsperando   // [Cliente]
  field aceptaPagos         // [TipoDePago]
  field esRápida            // Bool
  field facturado           // Número
}
```

**Nota**: el campo "*facturado*" del tipo **`Caja`** es la cantidad total de plata que recaudó esa caja. Es decir, la suma de los precios de todos los productos de todos los clientes que pasaron ya por ella.

1. Armar ejemplos para visualizar el modelo y sus datos.
2. Implementar las siguientes funciones:
    a) **`cantidadDeClientesEsperandoEn_`**, que dada una caja, describe la cantidad de clientes que están esperando en la misma.
    b) **`cajaMenosOcupadaDe_`**, que dada una lista de cajas describe la caja con menos clientes esperando entre todas las de esa lista.
    c) **`gobsMart_conIngresoDe_aCaja_`**, que dada una lista de cajas, un cliente y un número de caja, describe la misma lista de cajas pero actualizando la caja del número dado para registrar que el cliente dado ingresó en la misma. Tener en cuenta que el cliente se debe agregar al final de la cola de espera.
    d) **`gobsMart_conIngresosDe_`**, que dada una lista de cajas y una lista de clientes, describe una lista de cajas actualizada, donde cada cliente de la lista dada ingresó a una caja que acepta el medio de pago que posee, y que esté menos ocupada en el momento que ingresa.
    e) **`caja_conPrimeroFacturado`**, que dada una caja, describe la caja resultante de facturar al primero de los clientes de la caja dada (o sea, el cliente se retiró y la caja actualizó sus datos). Se puede suponer que el cliente eligió bien la caja de acuerdo a su tipo de pago.
    f) **`gobsMart_conCliente_cambiaACaja_`**, que dada una lista de cajas, el DNI de un cliente y un número de caja, describe la lista de cajas actualizada, donde el cliente con el DNI dado se cambió al número de caja dado. Se puede suponer que existe un cliente con el DNI dado esperando en alguna de las cajas del gobsMart.
    g) **`fila_ConAumentoDePrecioAMarolio`**, que dada una lista de clientes, describe la lista de clientes dada donde se aumente en 10 pesos todos los productos de la marca “Marolio” que tengan en su poder.

# Parte 2 - Pókemon

Se desea modelar una parte de un juego de Pókemon mediante los siguientes tipos:

```
type TipoDePókemon is variant{
  /* PROPÓSITO: Modelar los tipos de Pókemon posibles          */
  case Tierra {}
```

```
  case Agua   {}
  case Fuego  {}
}

type Pókemon is record {
  /* PROPÓSITO: Modelar un Pókemon
     INV.REP.: * La fuerza y el nivel son mayores o iguales a 0
               * Si está debilitado (estáSaludable es falso), su
fuerza es cero                                                    */
  field tipo           // TipoDePókemon
  field fuerza         // Número
  field estáSaludable  // Booleano.
  field nivel          // Número
}
```

1. **Ejercicios con Registros y Variantes**

   Definir las siguientes funciones:

   a) **`es_MásFuerteQue_`**, que dados dos Pókemon indica si el primero tiene más fuerza que el segundo.

   b) **`esDeMayorNivel_Que_`**, que dados dos Pókemon indica si el si el primero tiene un nivel más alto que el segundo.

   c) **`pókemon_PotenciadoEn_`**, que dado un Pókemon y un número, describe el Pókemon resultante de multiplicar la fuerza y el nivel del Pókemon dado por ese número.

   d) **`pókemon_ConValoresDuplicados`**, que dado un Pókemon, describe el Pókemon resultante de duplicar la fuerza y el nivel del Pokémon dado.

   e) **`pókemon_PotenciadoEn_SiEsDeTipo_`**(pokémon, factor, tipo), que dado un Pókemon, un número y un tipo de Pókemon, describe el Pokémon resultante de potenciar al pokemon dado según el factor dado, solamente si es del tipo dado, y el original si no.

   f) **`pókemon_Derrotado`**, que dado un Pókemon, describe el Pokémon resultante de debilitar al Pókemon dado (o sea, su fuerza será 0 y su campo estáSaludable será falso).

2. **Ejercicios con Listas de Registros**

   Definir las siguientes funciones:

   a) **`pókemonDe_Entrenados_`**, que dada una lista de Pókemon y un número, describe la lista resultante de potenciar cada Pókemon de la lista dada en la cantidad dada.

   b) **`pókemonDe_DelTipo_`**, que dada una lista de Pókemon y un tipo de Pókemon, describe la lista de aquellos Pókemon de la lista dada que son del tipo dado.

   c) **`elMásFuerteDe_`**, que dada una lista de Pókemon, describe el Pókemon de nivel más alto de toda la lista; si hay dos o más del mismo nivel más alto, da lo mismo cual se describe. ¿Qué precondición se debe exigir?

d) **pókemonDe_DelTipo_Duplicados**, que dada una lista de Pókemon y un tipo de Pókemon, describe lista de Pókemon resultante de duplicar aquellos Pókemon de la lista original que son del tipo dado, dejando los demás exactamente igual. El orden en la lista resultante debe ser el mismo que en la lista dada.

e) **elPókemonMásDébilDe_**, que dada una lista de Pókemon describe al Pókemon de nivel más bajo de toda la lista; si hay dos o más del mismo nivel más bajo, da lo mismo cual se describe. ¿Cuál es la precondición de esta función?

f) **pókemonDebilitadosDe_**, que dada una lista de Pókemon, describe la lista de aquellos Pókemon de la lista dada que están debilitados.

g) **cantidadDePókemonSaludablesEn_**, que dada una lista de Pókemon, describe la cantidad de Pókemon de la lista que no están debilitados.

h) **existePókemonEn_ConFuerza_Tipo_YNivel_**, que dada una lista de Pókemon, un número para indicar una fuerza, un tipo de Pókemon y un número para indicar un nivel, indica si en la lista dada existe algun Pókemon de ese tipo con esa fuerza y ese nivel.

**3. Ejercicio con Registros y Listas.**

Supongamos además que se definen el siguiente tipo:

```
type Entrenador is record{
  /* PROPÓSITO: Modelar un entrenador de Pókemon.
     INV.REP.: identificador es un número > 0                    */
  field lista            // [Pokemon]
  field identificador    // Número
  field esTáctico        // Booleano
}
```

Definir las siguientes funciones, sin olvidar establecer adecuadamente las precondiciones:

a) **entrenador__**, que dados un identificador y un booleano que indica si el entrenador es táctico, describe un Entrenador con los datos dados. Notar que por defecto el entrenador no tiene Pókemon para entrenar.

b) **entrenador_ConPókemon_Agregado**, que dados un entrenador y un Pókemon, describe al entrenador resultante de agregar al Pókemon dado a la lista de Pókemon del entrenador dado.

c) **cantidadDePókemonDe_**, que dado un entrenador, describe la cantidad de Pókemon del entrenador recibido.

d) **cantidadTotalDePókemonEn_**, que dada una lista de entrenadores, describe la cantidad total de Pókemon entre todos los entrenadores de la lista.

e) **`entrenadorMásAntiguoEntre_Y_`**, que dados dos entrenadores, describe al entrenador más antiguo de ambos (o sea, el que tiene menor número de identificación).

f) **`elMásAntiguoEn_`**, que dada una lista de entrenadores, describe al entrenador más antiguo de la lista dada.

g) **`entrenadorGanadorDeDesafíoEntre_Y_`**, que dados dos entrenadores, describe al Entrenador ganador del desafío.
El desafío consiste en que compitan un Pókemon de cada entrenador hasta que se acaben los Pókemon de uno de ellos, resultando en batallas del primero con el primero, el segundo con el segundo, etc. En cada pelea entre dos Pókemon siempre gana el más fuerte, y si tienen igual fuerza, la pelea no cuenta para ninguno de los dos. Gana el desafío el entrenador que consiga ganar más batallas, o de haber empate en la cantidad de peleas ganadas, el de mayor antigüedad.

h) **`entrenadorGanadorDeDesafioEn_`**, que dada una lista de entrenadores, describe al entrenador ganador del desafío entre todos los entrenadores.
El desafío de cada par de entrenadores es idéntico al que se describió en el punto anterior. En primer lugar compiten el primer entrenador con el segundo, y luego de cada pelea el entrenador que gana compite con el siguiente de la lista, hasta que no haya más entrenadores para competir. El ganador del desafío es el entrenador que gana la última batalla.

i) **`fuerzaTotalDe_`**, que dado un entrenador, describe el número que es la suma de la fuerza de todos los Pókemon de ese entrenador.

j) **`fuerzaTotalEnBatallaEn_`**, que dada una lista de entrenadores, describe el número que es la suma de la fuerza de todos los Pókemon de todos los entrenadores recibidos.

k) **`mejorPókemonDe_ParaJugada`**, que dado un entrenador, si el entrenador es Táctico describe a su Pokémon de mayor nivel con sus valores duplicados, y si no es táctico describe al primero de su lista de Pókemon.

l) **`ganadorDeDesafíoInteligenteDe_CombatesEntre_Y_`**, que dados un número de combates y dos entrenadores, describe al entrenador ganador del desafío inteligente entre ambos.
A diferencia del desafío común, un desafío inteligente consiste solamente en la cantidad de combates dada. En cada combate el entrenador usa su mejor Pókemon para esa jugada, que luego no vuelve a competir. Notar que *no* juegan todos los Pókemon de la lista como en el desafío común, sino solamente la cantidad dada. Nuevamente, gana el que más combates gana, y si hay empate, el de mayor antigüedad.

m) **`ganadorDeDesafíoInteligenteDe_CombatesEn_`**, que dados un número de combates por desafío y una lista de entrenadores, describe al entrenador ganador del desafío inteligente entre todos los entrenadores dados.
El desafío inteligente entre varios entrenadores sigue la misma mecánica que el

común, pero en el desafío entre cada par de entrenadores, se realiza un desafío inteligente.