Práctica 10
Listas de Registros

Introducción a la Programación
2do Semestre de 2016

Los ejercicios que corresponden a los **contenidos mínimos** recomendados se encuentran marcados con el simbolo ⊛.

1. Ejercicios introductorios

Ejercicio 1

Suponga un pirata en una isla con un mapa. Su objetivo es encontrar el tesoro en esa isla. La isla esta modelada con el tablero de Gobstones. El mapa es una lista cuyos elementos son registros de tipo `Indicacion` y el tesoro es una celda con 2 bolitas rojas. El punto de partida está indicado con una celda con 1 bolita roja.

```
# Un mapa es una lista de indicaciones
type Indicacion is record
 {
   field direccion    # Dir
   field distancia    # número
 }
```

Implementar la función `busquedaDelTesoro(mapa)`, que tome un mapa y retorna un booleano indicando si encontró o no el tesoro. Tenga en cuenta que el mapa puede tener errores (en el sentido que siguiendo sus indicaciones podría caer al agua – más allá de los límites del tablero hay agua) y el tesoro puede no existir.

Ejercicio 2

Utilizando el tipo `Persona` (práctica de registros), realice las siguientes funciones, suponiendo que no hay dos personas con el mismo número de DNI, donde usamos el tipo Padrón como renombre de listas de personas:

1. `perteneceDNI(padron, dni))`, recibe un padrón y un número de DNI y retorna true si hay una persona con dicho DNI en el padrón.
2. `personaConDNI(padron, dni)`, recibe un padrón y un número de DNI y devuelve la primer persona que tenga el DNI indicado. Puede suponer que existe alguna persona con dicho DNI.
3. `convivientes(padron, domicilio)`, recibe un padrón y un domicilio y retorna todas las personas que viven en dicho domicilio.

4. `sinPersonaConDNI(padron, dni)`, recibe un padrón y un número de DNI y retorna el padrón que se obtiene de sacar a la persona con dicho DNI.

5. `sinPeronasConDNI(padron, dnis)`, recibe un padrón y una lista con números de DNI y retorna el padrón que resulta de eliminar todas las personas correspondientes a los DNIs pasados como parámetro.

6. `nuevosDatosPersona(padron, dniViejo, persona)`, recibe un padrón, un dni y una persona y retorna el padrón que se obtiene de actualizar los datos la persona con el dni viejo para que coincidan con la nueva persona. En otras palabras, debe eliminarse aquella persona del padrón correspondiente al dni viejo, y agregarse la nueva persona.

7. `mudarFamilia(padrón, domicilioAntiguo, domicilioNuevo)` que recibe una lista de personas y dos domicilios, y actualiza los datos de todas las personas con domicilio antiguo para que pasen a tener domicilio nuevo.

8. `dameUnaFamilia(padrón)`, recibe un padrón y retorna una lista de personas convivientes.

9. `familias`, recibe un padrón y retorna la lista de todas las familias (que es una lista de lista de personas).

10. `consistente(padrón)`, recibe un padrón y verifica que no hayan personas con el mismo dni que otras.

Ejercicio 3

Declarar los tipos de registro `Alumno`, `Parcial` y `Curso` con la siguiente información:

- `Parcial` representa un parcial dado por un alumno, y sólo recuerda el tema (un numero) y la nota (otro número).
- Por cada alumno debe guardarse el legajo (un número) y la lista de parciales que rindió. En esa lista puede haber más de un parcial para el mismo tema, en caso que el alumno haya tenido que recuperar.
- Del curso se guarda la lista de alumnos y la lista de temas a evaluar (es una lista de números), por ejemplo "gobstones", "recorridos" y "listas" (cada tema se corresponde con un número).

Implementar las siguientes funciones:

1. `nuevoCurso(temas)`, recibe una lista de temas y crea un curso con una lista vacía de alumnos.

2. `inscripcionAlumno(curso, legajo)`, recibe un curso y un legajo y agrega un alumno con ese legajo al curso, con una lista vacía de parciales.

3. `evaluacionAlumno(curso, legajo, tema, nota)`, recibe un curso, el legajo de un alumno, un tema y una nota, y retorna el curso en el que se registro que el alumno correspondiente al legajo rindió el tema con la nota correspondiente. Como precondición, el alumno correspondiente y el tema del parcial deben formar parte del curso. La

función **debe** denotar BOOM cuando no se cumpla la precondición. **Ayuda:** es particularmente importante la división en subtareas. Realizar todas las operaciones auxiliares que sean necesarias. Como mínimo, se sugiere tener una función que retorne el alumno correspondiente al legajo, otra que elimine al alumno correspondiente al legajo, y una que agregue un alumno cualquiera al curso.

4. `aprobo(curso, legajo, tema)`, recibe un curso, un número de legajo y un tema y devuelve un booleano indicando si el alumno correspondiente al legajo aprobó ese tema. Nótese que un alumno puede contener más de una evaluación para el mismo tema, con haber aprobado una de ellas es suficiente para estar aprobado.

5. `vaBien(curso)`, recibe un curso y devuelve un booleano que indica si el curso tiene más de la mitad de alumnos aprobados. Para considerarse aprobado un alumno tiene que aprobar la totalidad de los temas del curso. Dividir en subtareas

Ejercicio 4

Utilizando el tipo `Celda` (práctica de registros), realice los siguientes ejercicios.

```
/* El tipo Fila es un renombre de Lista de Celda.
   El tipo Tablero es un renombre de Lista de Fila. */
```

1. Escriba una función `leerFila` que denote la lista de `Celda` (`Fila`) que conforman la fila actual.

2. Escriba una función `leerTablero` que denote la lista de `Fila` (`Tablero`) que representan al tablero en su estado actual.

3. Escriba una función `vaciarFila` que dado una `Fila f` retorne la fila con las celdas vacias. Para ello, utilice la función `celdaVacia` que denota una celda vacía.

4. Escriba una función `llenarFila` que dado una `Fila f` y una `Celda c` retorna la fila con todas sus celdas incrementadas en la cantidad de bolitas que establece `Celda c`. Implemente una función `sumarCelda` que dado dos celdas retorne una tercera con la suma de bolitas de ambas celdas para cada color.

5. Escriba una función `llenarFilas` que dado un `Tablero t`, una `Celda c` y una lista de números de fila `fs`, retorne el tablero donde cada fila fue llenada con la `Celda c`.

6. Escriba un procedimiento `DibujarTablero` que dado un `Tablero t`, vacíe el tablero y ponga bolitas en él de acuerdo a la información en `t`. Para ello, programe un procedimiento `DibujarFila` y haga uso de el procedimiento `PonerCelda` (definido en la práctica de registros).

Ejercicio 5

Considerar las siguientes estructuras de registros e implementar las operaciones solicitadas.

```
/* Representa una cantidad de bolitas, contando azules,
   negras, rojas y verdes, respectivamente. */
type Bolitas is record
 {
   field azules    # número
   field negras    # número
   field rojas     # número
   field verdes    # número
 }

/* Representa una cantidad de bolitas de un color */
type CantidadDeColor is record
 {
     field color          # Color
     field cantidad       # número
 }
```

1. `cantidadDeColorEnTablero(c)`: una función que retorne un registro `CantidadDeColor` con la cantidad total de las bolitas de color `c` en el tablero.

2. `regionVacia()`: una función que denote un registro `Bolitas` que representa una región vacía (i.e., con todos los campos en cero).

3. `sumaBolitas(r1, r2)`: una función que, dados dos registros de `Bolitas`, retorne un registro nuevo con la suma de las bolitas de ambas regiones.

4. `bolitasEnColumna()`: una función que retorne un registro `Bolitas` con la suma total por bolitas de la columna de la celda del cabezal. Resolverlo de dos formas: 1. reusando la función `cantidadDeColorEnColumna` 2. haciendo un recorrido del tablero mientras acumula el resultado en una variable de registro `Bolitas`. ¿Qué solución prefiere y por qué?

5. `listaCromaticaDeColumna()`: una función que retorne una **lista** cuyos elementos son registros de `CantidadDeColor`, donde cada registro tiene la cantidad de **un color** en la columna. Notar que en XGobstones, la lista resultante tiene sólo cuatro elementos. Sin embargo, la función debe ser correcta *independientemente* de la cantidad de colores.

6. `bolitasEnColumna` y `listaCromaticaDeColumna` tienen propósitos similares. ¿Qué diferencias puede encontrar? ¿Qué habría que modificar en cada una si la cantidad de colores aumenta?

7. `sublistaCromaticaDeColumna(colores)`: una función que, dada una lista de colores `colores`, retorne la sublista de la `listaCromaticaDeColumna()` con aquellos registros que corresponden a los `colores` pasados como parámetro. ¿Se le ocurre cómo se podría implementar una función análoga usando registros de tipo `Bolitas`? Discuta las deficiencias de los registros a la hora de representar listas, incluso cuando la cantidad de elementos es conocida.

8. `listaBolitasEnCeldas()`: una función que retorne una lista cuyos elementos son registros de `Bolitas`, donde cada registro tiene la cantidad de bolitas de cada color de una celda.

9. `listaCromaticaDeCelda()`: una función que retorna una lista cuyos elementos son registros de `CantidadDeColor`, donde cada registro tiene la cantidad de bolitas de un color. (Esta función tiene que funcionar independientemente de la cantidad de colores que tiene XGobstones)

10. `listaCromaticaDeCeldas()`: una función que retorna una lista cuyos elementos resultan de aplicar listaCromaticaDeCelda() en cada una de las celdas del tablero. Notar que cada elemento de la lista es a su vez una lista de registros `CantidadDeColor`. Discutir la conveniencia de invocar una función para obtener cada elemento interno de la lista.

2. WhileMart

Los directivos del supermercado WhileMart desean optimizar el número de cajas de sus instalaciones, de tal modo que sean mínimas para reducir los costos, pero suficientes para evitar la acumulación y demora de los clientes. Para ello se dispone de la información de los clientes de una sucursal en un día típico, que se modela mediante los siguientes registros:

```
type Cliente is record
 {
   field horaIngresoCola      # número
   field cantProductos        # número
 }

type Caja is record
 {
   field numero               # número
   field clientesEsperando    # Lista de Cliente
 }

type Super is record
 {
   field horaActual           # número
   field lineaDeCajas         # Lista de Caja
   field clientesComprando    # Lista de Cliente
 }
```

Para cada cliente, se conoce la hora a la que empezó o empezará a hacer cola (`horaIngresoCola`) y la cantidad de productos que compró (`cantProductos`). Cada caja del supermercado se identifica por un número de caja (`numero`) e incluye una lista de los clientes que están haciendo cola en esa caja (`clientesEsperando`).

Para simular un supermercado, se cuenta con el registro `Super`, que indica la hora actual en la simulación del supermercado (`horaActual`), la lista de todas las cajas (`lineaDeCajas`) y la lista de todos los clientes que todavía no comenzaron a hacer cola (`clientesComprando`).

Se asumen los siguientes hechos sobre el funcionamiento del supermercado:

- Cuando un cliente termina de comprar y comienza a hacer cola, elige la caja en la que haya menos personas esperando. En caso de empate, elige la caja que tenga el menor número de caja.

- La lista de clientes esperando en una caja representa una cola. El primer elemento de la lista representa el cliente que está siendo atendido.

- Las cajas procesan un producto por unidad de tiempo.
- Cuando una caja termina de procesar todos los productos del cliente que está siendo atendido, dicho cliente se retira y se pasa al siguiente cliente esperando en la cola, si lo hay.
- Se asumirá siempre que las líneas de cajas no contienen cajas con números repetidos.

Se solicita resolver los siguientes ejercicios. Tener en cuenta que *ninguno* de ellos involucra el tablero de XGobstones.

Ejercicio 6

Implementar `cajaMenosOcupada(lineaDeCajas)` una función que, dada una `lineaDeCajas`, denota la caja de que está *menos ocupada*, es decir, aquella en la que hay menos clientes haciendo cola. En caso de empate, se elige la caja que esté menos ocupada y que tenga el número más chico. Por ejemplo, si la caja 1 está atendiendo un cliente, y las cajas 2 y 3 están libres, se elige la caja 2. Puede suponer que `lineaDeCajas` tiene alguna caja.

Ejercicio 7

Implementar `ingresaColaCliente(lineaDeCajas, cliente)` una función que, dada una `lineaDeCajas` y un `cliente`, denota la línea de cajas que se obtiene después de que el cliente comienza a hacer cola. El cliente se ubica en la caja que esté menos ocupada, de acuerdo con el criterio del ejercicio anterior. Puede suponer que `lineaDeCajas` tiene alguna caja.

Ejercicio 8

Implementar `avanzaCaja(lineaDeCajas, nroCaja)` una función que, dada una `lineaDeCajas` y un número `nroCaja`, denota la línea de cajas que se obtiene después de que la caja identificada por `nroCaja` procesa un producto. Si la caja en cuestión no tiene clientes, la línea de cajas debe quedar tal como estaba. Si ya se procesaron todos los productos del cliente que está siendo atendido, dicho cliente se retira de la caja. Como precondición, puede suponer que hay una caja identificada por `nroCaja`.
Ayuda: se recomienda tener funciones que busquen y actualicen una caja de la línea, a partir de su número.

Ejercicio 9

Implementar `avanzaLineaDeCajas(lineaDeCajas)` una función que, dada una `lineaDeCajas`, denota la línea de cajas que se obtiene después de que todas y cada una de las cajas procesan un producto, con las mismas observaciones que en el ejercicio anterior.

Ejercicio 10

Implementar `finalDeLasCompras(whileMart)` una función que, dado un super `whileMart`, denota el super que se obtiene de hacer que todos los clientes cuya `horaIngresoCola` coincide con la hora actual del supermercado ingresen a alguna cola. Notar que cada cliente que ingresa a una cola lo hace según el criterio ya establecido (elige la caja menos ocupada). Puede suponer la existencia de (o implementar) `clientesQueIngresanALas(clientes, hora)` y `clientesQueNoIngresanALas(clientes, hora)` que, dadas una lista de `clientes` y un número `hora`, retornan las listas de clientes que ingresan y no ingresan a una cola a dicha hora, respectivamente.

Ejercicio 11

Implementar `pasoDelTiempo(whileMart)` una función que, dado un super `whileMart`, denota el supermercado después de que pase una unidad de tiempo. Lo que se debe hacer es, en orden:

- Hacer que los clientes que terminaron sus compras en la hora actual pasen a estar esperando en las cajas.
- Procesar un producto en cada una de las cajas de la línea de cajas.
- Avanzar el reloj una unidad de tiempo.

Ejercicio 12

Implementar `supermercadoVacio(whileMart)` una función que, dado un super `whileMart`, denota Verdadero si el supermercado está completamente vacío. Es decir, si no hay clientes comprando y no hay clientes esperando en ninguna caja.

Ejercicio 13

Implementar `horaEnQueQuedaVacio(whileMart)` una función que, dado un super `whileMart`, denota la hora en la que el supermercado queda completamente vacío. *Nota:* Para determinar ese valor, simular el paso del tiempo hasta que el supermercado quede vacío.

3. Tuberías

Una *tubería* consiste de una serie de *piezas*. Cada pieza tiene una longitud y una dirección. Ambos elementos se modelan en XGobstones de la siguiente manera:

```
type Pieza is record
 {
   field tamanho          # número
   field direccion        # Dir
 }
 # Tuberia es un renombre de Lista de Pieza.
```

El campo `tamanho` indica la longitud de la pieza y `direccion` su dirección.
Resolver los siguientes ejercicios (NB: ninguno de ellos requiere el uso del tablero):

Ejercicio 14

Escribir una función `pipesACoordenadas(tuberia, inicio)` que, dada una tubería y una coordenada de inicio (como en la práctica de registros), retorne la lista de coordenadas por las que pasan las piezas de la tubería comenzando en la coordenada de inicio.

Ejercicio 15

Escribir la función `esTuberiaValida(tuberia, inicio)` que, dada una tubería y una coordenada de inicio, indique si la misma es *válida*, es decir si:

1. Todas las coordenadas por las que pasa la tubería son positivas; y
2. La tubería pasa a lo sumo una vez por cada coordenada.

Ejercicio 16

Un *tramo* de una tubería es una secuencia de piezas consecutivas y con la misma dirección. La *longitud de un tramo* es la suma de las longitudes de sus piezas. Escribir `estandarizarTuberia(tuberia,longitud)`, una función que dada una `tubería` y un número `longitud`, retorne la tubería en la que se rearma cada tramo de longitud `t` de la `tuberia` dada como argumento, utilizando piezas de longitud `longitud`. Ello implica rearmar cada tramo colocando:

- `t div longitud` piezas de longitud `longitud` y
- una de `t mod longitud`.

4. Mini Twitter

El presente conjunto de ejercicios tiene por finalidad simular la operatoria de *Twitter*, una de las plataformas sociales más populares de la actualidad. Esta plataforma consiste de una comunidad de *usuarios* que envían mensajes llamados *tweets*. Los usuarios tienen *seguidores*, y son estos seguidores los que pueden ver los tweets de un usuario.

Para modelar la mencionada plataforma, se introducen los registros que a continuación se describen. Cada usuario contiene la identificación del mismo, la lista de tweets que envió hasta el momento y la lista de identificadores de sus seguidores. Cada tweet viene dado por la fecha y hora en que se emitió y por el contenido en sí[1]. Finalmente, la plataforma en sí contiene la lista de usuarios y un número que servirá a la hora de dar de alta nuevos usuarios.

```
type Usuario is record
 {
   field id              # número
   field tweets          # Lista de Tweet
   field seguidores      # Lista de número
 }

type Tweet is record
 {
   field fecha           # número
   field hora            # número
   field mensaje         # número
 }

type Twitter is record
 {
   field usuarios        # Lista de Usuario
   field proximoId       # número
 }

type _ is record
 {
   field        #
   field        #
 }
```

[1]Para simplificar, el contenido de un mensaje es un número en lugar de texto.

Se solicita resolver los siguientes ejercicios. Tener en cuenta que *ninguno* de ellos involucra el tablero de XGobstones.

Ejercicio 17

Implementar `altaDeUsuario(twitter)` una función que, dado un `twitter`, retorna el twitter que se obtiene de dar de alta un nuevo usuario. El usuario agregado no tiene seguidores, ni tweets, y su id está dado por el valor del campo `proximoId(twitter)`. Este campo debe ser incrementado una vez que el alta se efectiviza, de modo tal que el próximo usuario que se agregue no repita ese número de identificación.

Ejercicio 18

Implementar `bajaDeUsuario(twitter, id)` una función que, dado un `twitter` y un número `id`, retorna el twitter que se obtiene de dar de baja el usuario con identificación `id`. Además de eliminar el usuario de la lista de usuarios, tener en cuenta que *también* debe eliminarse `id` de la lista de los usuarios a los que sigue. Como precondición, suponer que hay un usuario con identificador `id`.

Ejercicio 19

Implementar `seguirA(twitter, idSeguidor, idASeguir)` una función que, dado un `twitter`, y dos números `idSeguidor` e `idASeguir`, retorne el twitter que se obtiene de hacer que usuario con identificador `idSeguidor` sigua al usuario con identificador `idASeguir`. Como precondición, puede suponer que `twitter` contiene usuarios con ambos identificadores y que el usuario con identificador `idSeguidor` no sigue al usuario con identificador `idASeguir`.

Ejercicio 20

Implementar `tweetear(twitter, idEmisor, fecha, hora, mensaje)` una función que, dado un `twitter`, un número `idEmisor`, una `fecha`, una `hora` y un `mensaje`, retorne el twitter que se obtiene de agregar un nuevo tweet a la lista de tweets del usuario con identificación `idEmisor`. Puede suponer que el usuario con identificador `idEmisor` existe.

Ejercicio 21

Implementar `tweetsVisiblesPorUsuario(twitter, id)` una función que, dado un `twitter` y un `id`, retorne la lista de todos los tweets que son visibles por el usuario con identificador `id`. Recordar que los tweets visibles son aquellos de todos los usuarios que `id` está siguiendo. Como precondición, puede suponer que el usuario `id` existe.

Ejercicio 22

Implementar `usuariosMasPopulares(twitter)` una función que, dado un `twitter`, retorne la lista con los identificadores de todos los usuarios más populares, es decir, aquellos que tienen más seguidores.

5. Dime DB

El objetivo de esta sección es programar Dime DB, una base de datos películas de que permite que los usuarios califiquen las películas que vieron. Las calificaciones se usan para recomendar otras películas a los usuarios de acuerdo a sus gustos.

La base de películas es simplemente una lista que contiene todas las películas conocidas.

```
# DimeDB es un renombre de Lista de Pelicula
```

A la vez, una película se compone por su título, el nombre de su director y la lista de calificaciones que recibió. Es importante remarcar que pueden haber muchas películas con el mismo título, pero ningún director ha dirigido dos películas con el mismo título. Con respecto a las calificaciones, un usuario puede calificar muchas veces a una película, pero DIME DB guarda únicamente la ultima calificación que se haga. De esta forma, un usuario puede "actualizar" la nota de una película en el sistema.

```
type Pelicula is record
 {
   field titulo          # número
   field director        # número
   field calificaciones  # Lista de Calificacion
 }
```

Por último, una calificación indica qué nota (del 0 al 10) puso un usuario. Cuánto más le haya gustado la película, más alta será la nota.

```
type Calificacion is record
 {
   field usuario         # número
   field nota            # número del 0 al 10
 }
```

Se solicita resolver los siguientes ejercicios. Tener en cuenta que ningún ejercicio involucra el uso del tablero XGOBSTONES.

Ejercicio 23

Implementar `calificar(db, titulo, director, usuario, nota)` una función que, dada una base de peliculas `db`, el `título` y `director` de una película, un identificador de `usuario` y una `nota`, retorna la base de películas en la que `usuario` calificó la película `título` del `director` con la `nota` indicada en la base `db`. Recordar que DIME DB guarda únicamente la última calificación del `usuario` para la película. En caso que `usuario` ya hubiera calificado la película en `db`, la calificación previa debe quitarse. Puede suponer la existencia de (o implementar) la función `reemplazarPorTituloDirector(db, pelicula)` que retorna el `DimeDB` que se obtiene de reemplazar la película `titulo(pelicula)` del `director(pelicula)` por `pelicula`. Puede suponer que la película `título` del `director` existe en `db` y `nota` es un valor del 0 al 10.

Ejercicio 24

Implementar `usuariosDB(db)` una función que, dado una base de películas `db`, retorna la lista de todos los usuarios que hayan calificado al menos una película en `db`, sin repetidos. Recordar la función `sinDuplicados(lista)` que, dada una lista, retorna la lista sin repetidos.

Ejercicio 25

Implementar `gustaronAlUsuario(db, usuario)` una función que, dada una base de peliculas

`db` y un nombre de `usuario`, retorna la lista de **todas** las películas de `db` que le gustaron a `usuario`. Consideramos que a un usuario le gustó una película cuando la calificó con una nota mayor o igual a 8.

Ejercicio 26

Implementar `gustaronDeLaPelicula(db, titulo, director)` una función que, dada una base de películas `db` y el `título` y `director` de una película, retorne la lista con todos los usuarios de `db` a los que le gustó la película `titulo` del `director`. Como precondición, puede suponer que la pelicula `título` del `director` existe en `db`.

Ejercicio 27

Implementar la función `siTeGustoPuedeQueTambienTeGuste(db, titulo, director)` que, dada una base de películas `db` y el `título` y `director` de una película, retorna la lista de películas que les gustó a **todos** los usuarios a los que también le gustó la película `titulo` del `director` (excluyendo del resultado la película `titulo` del `director`). Notar que la lista resultante contiene **todas** las películas de `db` (salvo por la película `titulo` del `director`) en el caso en que a ningún usuario le haya gustado la película `titulo` del `director`. Como precondición, puede suponer que la pelicula `título` del `director` existe en `db`

6. XMail

XMail es un programa simple para gestionar los e-mails de un usuario. Cada usuario almacena de manera conjunta todos los e-mails, tanto enviados como recibidos. Asimismo, utiliza distintas *etiquetas* para poder buscar, filtrar y generar vistas de los e-mails. Las etiquetas no están predeterminadas por el sistema, sino que cada usuario puede agregar las etiquetas que quiera a sus mensajes. Incluso, un mensaje puede tener distintas etiquetas de acuerdo al gusto del usuario. Por ejemplo, un correo puede tener las etiquetas: "yo", "trabajo", "jefe", "compañeros", "asado fin de año". Cada e-mail se modela en XGobstones con el siguiente registro.

```
type Correo is record
 {
   field idRemitente      # número
   field idDestinatarios  # lista de número
   field etiquetas        # lista de número
   field mensaje          # numero
 }
```

El campo `idRemitente` identifica al usuario que envió el e-mail, `idDestinatario` es una lista de usuarios a los que se envió el e-mail, `etiquetas` es la lista de todas las etiquetas de este e-mail, y `mensaje` es el mensaje del correo. Como invariante de representación, se sabe que las listas de etiquetas y destinatarios no tienen repetidos[2].

Tener que etiquetar cada e-mail manualmente es un trabajo tedioso y virtualmente imposible para un usuario que recibe cientos de e-mails por día. Para simplificar esta tarea, XMail le permite al usuario definir *reglas* de etiquetado que aplica automáticamente a cada

[2]Notar que se acepta una lista vacía de destinatarios (esto es útil para mensajes borrador), y que un usuario se envíe un e-mail a sí mismo.

e-mail enviado o recibido. De esta forma, por ejemplo, un usuario puede agregar una regla que etiquete como "recibido" todos los e-mails que él haya recibido, simulando de esta forma la bandeja de entrada. Las reglas son muy simples en esta primera versión de prototipo de la herramienta y se modelan con el siguiente registro de XGobstones.

```
type Regla is record
 {
    field esRemitente     # booleano
    field idUsuario       # número
    field etiqueta        # número
 }
```

El campo `etiqueta` es la etiqueta que será aplicada a todos los mensajes que cumplan la regla. El campo `idUsuario` es un nombre de usuario cualquiera y `esRemitente` es un booleano que denota si la regla se aplica cuando idUsuario es el remitente o cuando es un destinatario. Cómo se aplica una regla a un e-mail depende del valor del campo `esRemitente`. Si `esRemitente` denota true, entonces se agrega la etiqueta de la regla al e-mail cuando el remitente del mismo corresponde al campo `idUsuario` de la regla. Caso contrario, se agrega la etiqueta de la regla al e-mail cuando *alguno* de los destinatarios corresponde con el campo `idUsuario` de la regla. Por ejemplo, la regla con campos `esRemitente <- True`, `idUsuario <- messi@hace.lio`, `etiqueta <- enviados` puede ser usada por el usuario con identificador `messi@hace.lio` para simular la carpeta de e-mails enviados. (**Aclaración**: en XGobstones, `idUsuario` y `etiqueta` son números; por cuestiones de exposición—y para hacer referencial al mundial—es que los escribimos de la manera usual.)

Finalmente, XMail administra la cuenta de un usuario que tiene los siguientes datos.

```
type Usuario is record
 {
   field id              # número
   field correos         # lista de Correo
   field reglas          # lista de Regla
 }
```

El campo `id` es el nombre del usuario, `correos` contiene todos los correos del usuario, tanto enviados como recibidos, y `reglas` contiene todas las reglas que han de aplicarse automáticamente para etiquetar mensajes. Como invariante de representación, la única restricción es que todos los correos sean del usuario (es decir que es su remitente o uno de sus destinatarios). Notar, en particular, que pueden haber reglas duplicadas.

Se solicita escribir las siguientes funciones. Tener en cuenta que ninguna de ellas involucra el uso de tableros.

Ejercicio 28

`function nuevoUsuario(id)` que, dado un número `id`, crea un nuevo usuario con dicho `id`, sin correos y con dos reglas. La primer regla indica que todos los correos recibidos deben etiquetarse con 0 (inbox), mientras que la segunda regla indica que todos los correos enviados han de etiquetarse con 1 (enviado).

Ejercicio 29

`function etiquetasRemitente(reglas, idRemitente)` que, dada una lista de `reglas` y un

número `idRemitente`, retorna la lista con las etiquetas que se obtendrían al aplicar cada una de las `reglas` en e-mails cuyo remitente sea `idRemitente`.
Nota: no importa si la lista de etiquetas retornadas contiene repetidos.
Nota: la lista de reglas podría contener reglas que se aplican por destinatario; obviamente estas reglas serán ignoradas por la función.

Ejercicio 30

`function envioCorreo(usuario, destinatarios, mensaje)` que, dado un `usuario`, una lista de números `destinatarios` y un número `mensaje`, retorna el nuevo usuario que se obtiene luego de enviar un correo a los `destinatarios` con el `mensaje` indicado por el `usuario` del parámetro. Recordar que XMail aplica **todas** las reglas de etiquetado sobre este mensaje, y que ninguna etiqueta debe aparecer duplicada.
Suponer que hay una función `etiquetasDestinatario(reglas,idDestinatario)` que retorna la lista de etiquetas que se obtiene si se aplican las reglas sobre un e-mail que tiene a idDestinatario como uno de los destinatarios.

Ejercicio 31

`function busquedaPorEtiquetas(usuario, etiquetas)` que, dado un `usuario` y una lista de `etiquetas`, retorne la lista de correos del `usuario` que contienen **todas** las `etiquetas` pasadas como parámetro.
Suponer la existencia de una función `incluido(lista, sublista)` que denota verdadero si **todos** los elementos de `sublista` pertenecen a `lista`. Por ejemplo, `incluido([1,2,3], [3,1])` denota verdadero, mientras que `incluido([1,2], [3])` denota falso.

Ejercicio 32

`function eliminacionEtiquetas(usuario, etiquetas)` que, dado un `usuario` y una lista de `etiquetas`, retorna el `usuario` que se obtiene si se eliminan todos los e-mails del `usuario` parámetro que contienen **alguna** de las `etiquetas` pasadas como parámetro.
Recordar la función `interseccion` del ejercicio 26 de la práctica de listas.

Ejercicio 33

`function etiquetas(usuario)` que, dado un `usuario`, retorna la lista de todas las etiquetas de los e-mails del `usuario`. La lista resultante no debe tener elementos repetidos.

Ejercicio 34

`function importacionCorreos(usuario, correos)` que, dado un `usuario` y una lista de `correos`, retorna el usuario que se obtiene de agregar al `usuario` parámetro todos los `correos` luego de aplicar las reglas del `usuario` parámetro.

7. Vuela-vuela

El presente conjunto de ejercicios tiene por finalidad simular el tráfico aéreo, en la visión de una Torre de Control que lleva a cabo el seguimiento de cada nave. Una *Torre de Control* consiste de una lista de *vuelos* y una lista que asocia un *plan de vuelos* (o ruta) con cada par de la forma (origen, destino). La lista de vuelos va a ir variando a medida que los vuelos despegan y aterrizan. Sin embargo, la lista que asocia planes de vuelo para cada par (origen,

destino) siempre es la misma: si estando en origen se quiere ir a destino, entonces habrá un único plan de vuelo que permita conectar esas dos coordenadas.

A su vez, cada *vuelo* tiene un número de vuelo (único), un origen y un destino (que determinan el plan de vuelo a ejecutar), la etapa actual del plan de vuelo en la que se encuentra la nave y el estado. La etapa del plan de vuelo va entre 1 y la longitud del plan de vuelo correspondiente, mientras que el estado es un color que puede ser

- Saliente (Verde): avión listo para despegar.
- En ruta (Azul): avión en vuelo.
- Aterrizado (Negro): avión llegó a destino.

Por último, un *plan de vuelos* consiste de una lista de direcciones N, S, E u O. Los siguientes registros modelan estos datos.

```
type TorreDeControl is record
 {
   field vuelos          # Lista de Vuelo
   field planesDeVuelo   # Lista de PlanDestino
 }

type Vuelo is record
 {
   field numero               # número
   field origen               # Coord
   field destino              # Coord
   field estado               # Color
   field etapaEnPlanDeVuelo   # número
 }

type PlanDestino is record
 {
   field origen          # Coord
   field destino         # Coord
   field planDeVuelo     # Lista de Dir
 }
/* Coord se definió en la práctica
   de registros. */
```

Se solicita resolver los siguientes ejercicios. Tener en cuenta que ningún ejercicio involucra el uso del tablero GOBSTONES con la *excepción* del último.

Ejercicio 35

Implementar `vueloEnEstado(vuelos, estado)` una función que, dada una lista de vuelos y un color que representa un estado, retorna el primer vuelo en la lista que tiene el estado dado como parámetro. Como precondición, hay al menos un vuelo con dicho estado.

Ejercicio 36

Implementar `despegarVuelo(torreDeControl)` una función que, dada una torre de control, retorna la torre de control que se obtiene de despegar al primer vuelo en `vuelos(torreDeControl)` que esté en estado "listo para despegar". Notar que para ello, debe cambiar el

estado del vuelo a "en ruta" e indicar que se encuentra en la etapa 1 del plan de vuelo. Como precondición, hay al menos un vuelo en la torre de control cuyo estado es "listo para despegar".
Ayuda: se sugiere dividir en subproblemas. En particular, es conveniente tener funciones que permitan 1. eliminar un vuelo existente de la torre de control a partir de su número, 2. agregar un vuelo a la lista de control, y 3. actualizar un vuelo de la torre de control, haciendo uso de 1. y 2.

Ejercicio 37

Implementar `aterrizarVuelo(torreDeControl)` una función que, dada una torre de control, retorna la torre de control que se obtiene de aterrizar el primer vuelo en `vuelos(torreDeControl)` que esté en estado "en ruta" y que haya llegado a destino (la etapa coincide con la longitud del plan de vuelo). Notar que para ello debe cambiar el estado del vuelo a "aterrizado". Como precondición, hay al menos un vuelo en condiciones de aterrizar. **Ayuda:** se sugiere una función que retorne el plan de vuelos de un avión, y otra que retorne todos los aviones que llegaron a destino, independientemente de su estado.

Ejercicio 38

Implementar `coordenadaFutura(torreDeControl, nro_vuelo, tiempo)` una función que, dada una torre de control, un número de vuelo y un número que representa un tiempo en etapas, retorne la coordenada sobre la que estará sobrevolando el vuelo correspondiente a `nro_vuelo` dentro de `tiempo` etapas, contando desde la etapa actual. Como precondición, puede suponer que hay un vuelo "en ruta" con el número de vuelo y la etapa actual más `tiempo` no supera la longitud del plan de vuelo.

Ejercicio 39

Implementar `hayColisionEnEtapa(torreDeControl, tiempo)` una función que, dada una torre de control y un número que representa un tiempo en etapas, indique si hay aviones cuyo estado es "en ruta" que colisionaran dentro de `tiempo` etapas, contando desde la etapa actual. Dos aviones colisionan en una etapa si pasan por la misma coordenada en esta etapa.

Ejercicio 40

Implementar `hayColision(torreDeControl)` una función que, dada una torre de control, indique si hay aviones con estado "en ruta" que colisionaran en alguna etapa futura. Tener en cuenta que basta considerar solo n etapas futuras, donde n es la máxima etapa posible (la longitud del plan de vuelos más largo posible).

Nota: El siguiente ejercicio involucra el uso del tablero.

Ejercicio 41

Implementar `t.RenderRadar(torreDeControl)` un procedimiento que, dada una torre de control, dibuja en `t` la ubicación actual de cada vuelo "en ruta". Los vuelos deben codificarse con tantas bolitas verdes como indica su número de vuelo. Cada origen con una bolita azul y cada destino con una bolita negra (el origen y destino debe codificarse a lo sumo una vez en el tablero). Puede asumir la existencia (o implementarla) de una función `enTablero(t, coord)` que indica si una coordenada cae dentro del tablero `t`, como así también el procedimiento `t.PonerEnCoord(coord, color)` que pone una bolita del color indicado en la coordenada indicada (se asume que la coordenada cae dentro del tablero `t`).