PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 - Programación Avanzada (I/2016)
Tarea 3

1. Objetivos

- Aplicar conceptos y nociones de programación funcional para el correcto modelamiento de un problema.
- Aprender nociones básicas sobre bases de datos.

2. Introducción

Luego de unas semanas, la plataforma MercadoPreso tuvo una caída horrible debido a su inestabilidad y a la abundancia de hackers que habían en el sistema. Esta masiva fuga de usuarios hizo que la empresa de Bucchi quebrara[1], por lo que usted se encuentra sumido en la miseria nuevamente, y nadie lo quiere contratar.

Paralelamente, el famoso e importante *John Router* intentó comprar la última versión de su Sistema de Manejo de Bases de Datos, a través de la ahora inexistente plataforma MercadoPreso. Mientras lo intentaba obtener, se encontró con un misterioso usuario que siempre apostaba exactamente un peso más que él. Sin esta actualización, *John* no puede manejar sus bases de datos y esto complica mucho su trabajo.

Ante este hecho, *John Router*, en su desesperación, te ha ofrecido una oferta que no puedes rechazar, debido a que podría ser su última posibilidad de tener un trabajo respetable[2]. *John* te ha pedido que escribas un sistema que pueda soportar un lenguaje de consultas que él mismo acaba de inventar, para así reemplazar el sistema antiguo que solía usar (cuya actualización ya ha salido del mercado), y se le haga fácil a él consultar cualquier base de datos.

3. Problema

Para esta tarea, deberá implementar un lenguaje de consultas nunca antes visto. Luego, deberá utilizar este lenguaje para responder consultas que tal vez salven a *John Router* (y a ti) de la miseria.

4. El Lenguaje de Consultas LCJ

LCJ es un lenguaje de acceso a bases de datos creado por *John Router*. Esta permite especificar diversos tipos de consultas en cualquier base de datos, haciendo uso de palabras naturales[3] que facilitan la obtención de información. Las base de datos que utiliza son del tipo relacional, cuya estructura se especifica a continuación.

[1] Desde entonces, el benévolo Bucchi se encuentra desaparecido...
[2] Y porque no tienes nada que comer, claro está. En verdad, es solo por esto último.
[3] Palabras de nuestro querido lenguaje chilensis.

4.1. Bases de Datos Relacionales

Para efectos de este trabajo, podemos ver las bases de datos a trabajar como un montón de tablas llenas de información. Cada tabla tiene un nombre, campos y contenido. Cada campo, a su vez, tiene un nombre y un tipo. El contenido es la información en sí que almacena. Por ejemplo, la siguiente tabla:

```
Persona (RUT str, nombre str, edad int, altura float)
```

Tiene nombre **Persona** y tiene 4 campos: **RUT** del tipo `string`, **nombre** del tipo `string`, **edad** del tipo `integer` y **altura** del tipo `float`. Y este en específico tiene como contenido la información de John y sus colegas:

RUT	nombre	edad	altura
10.456.334-7	John Router	33	1.71
12.345.678-9	Chris Rivers	33	1.60
9.876.654-k	Marcel Sands	40	1.68

Una característica de las bases de datos relacionales, es que muestran información que vincula (relaciona) distintas tablas. Por ejemplo, las tablas Producto y Compra:

Producto:

ID	nombreProducto	precio
0	Taza	5000
1	Plumón	2000

Compra:

Comprador	Producto
10.456.334-7	1
9.876.654-k	1
12.345.678-9	0
12.345.678-9	0
12.345.678-9	0

`Producto` nos detalla información de ciertos productos, y `Compra` nos muestra las compras de John y sus colegas, muestra el RUT del comprador y el ID del producto que compró. Así vemos que John y Marcel compraron un plumón y Chris compró... 3 tazas!!!
Todas las bases de datos que ingresaremos tendrán esta forma, **LCJ** nos permite preguntarle cosas a una base de datos cargada, como: ¿Quiénes compraron plumones? ¿Cuánto gastó cada uno? ¿Ranking ordenado de número de compras? Pero para esto, necesitamos definir el lenguaje de consultas...

4.2. Consultas

Las consultas son comandos escritos que al estar bien formados, especifican una busqueda de ciertas columnas, de ciertas tablas, que cumplan ciertas condiciones y los despliega de cierta forma. La forma más básica de una consulta, es aquella que pide mostrar una, varias o todas las columnas de una tabla:

```
EMPRESTA edad
DE Persona;
```

Resulta ⇒

30
33
40

```
EMPRESTA nombreProducto, precio
DE Producto;
```

Resulta ⇒

Taza	5000
Plumón	2000

```
EMPRESTA TODO
DE Compra;
```

Resulta ⇒

10.456.334-7	1
9.876.654-k	1
12.345.678-9	0
12.345.678-9	0
12.345.678-9	0

En general, una consulta comienza con que debe mostrar (`EMPRESTA`), termina con `';'` y entrega otra tabla como resultado (característica de bases de datos relacionales).

También se pueden seleccionar resultados bajo condiciones utilizando la sentencia `ONDE`:

```
EMPRESTA nombre
  DE Persona
ONDE edad <35;
```

Resulta ⇒

John Router
Chris Rivers

```
EMPRESTA precio
  DE Producto
ONDE nombre = 'Taza' Y precio <10000;
```

Resulta ⇒

5000

Las condiciones pueden ser conjunciones y disyunciones de varias condiciones, que tienen formas detalladas más adelante.
Si queremos utilizar más de una tabla, se nombran las deseadas en la sentencia `DE`. Al hacer esto, se deben calcular todas las combinaciones de cada contenido de una tabla con otra, de forma de poder mezclar contenido de distintas tablas:

```
EMPRESTA TODO
DE Persona, Producto;
```

Resulta ⇒

10.456.334-7	John Router	33	1.71	0	Taza	5000
10.456.334-7	John Router	33	1.71	1	Plumón	2000
12.345.678-9	Chris Rivers	33	1.60	0	Taza	5000
12.345.678-9	Chris Rivers	33	1.60	1	Plumón	2000
9.876.654-k	Marcel Sands	40	1.68	0	Taza	5000
9.876.654-k	Marcel Sands	40	1.68	1	Plumón	2000

Como vemos, se crea una fila combinada por cada persona en la tabla `Persona` y cada producto en la tabla `Producto`. Si además agregáramos `Compra` en la sentecia, por cada una de las filas anteriores, se copia con cada fila de `Compra`, resultando una tabla de tamaño 30 ($= 3 \cdot 2 \cdot 5$). Esto nos permite utilizar información de varias tablas: Obtener el nombre de una Persona con el nombre del producto que compra:

```
EMPRESTA nombre, nombreProducto
  DE Persona, Compra, Producto
ONDE RUT = Comprador Y Producto = ID;
```

Resulta ⇒

John Router	Plumón
Chris Rivers	Taza
Chris Rivers	Taza
Chris Rivers	Taza
Marcel Sands	Plumón

¿Qué sucedió? Al hacer `DE Persona, Compra, Producto`, se forman todas las combinaciones del contenido de las 3 tablas, pero `ONDE RUT = Comprador Y Producto = ID;` filtra aquellas filas donde el RUT de la Persona es igual al Comprador en la Compra y donde el producto de Compra es igual al id de Producto. Aquí esta parte de la tabla totalgenerada:

10.456.334-7	John Router	30	1.71	10.456.334-7	1	0	Taza	5000
10.456.334-7	John Router	30	1.71	10.456.334-7	1	1	Plumón	2000
10.456.334-7	John Router	33	1.60	9.876.654-k	1	0	Taza	5000
10.456.334-7	John Router	33	1.60	9.876.654-k	1	1	Plumón	2000
10.456.334-7	John Router	40	1.68	12.345.678-9	0	0	Taza	5000
10.456.334-7	John Router	40	1.68	12.345.678-9	0	1	Plumón	2000
...	...	...	...	...	...	...	...	...

Finalmente se muestran solamente lo declarado en `EMPRESTA nombre, nombreProducto`.

También, es posible ordenar los datos que entrega. Con la sentencia `ORDENATELOS X` declaras las columnas que definen el orden en que se mostraran los datos entregados:

```
EMPRESTA precio
  DE Producto
ORDENATELOS X precio PA RIBA;
```

Resulta ⇒

2000
5000

```
EMPRESTA nombre, edad, altura
  DE Persona
ORDENATELOS X edad, altura PA BAJO;
```

Resulta ⇒

Marcel Sands	40	1.68
John Router	33	1.71
Chris Rivers	33	1.60

También es posible agrupar filas por valores equivalentes dentro de una columna utilizando `AGRUPATELOS X`. Al señalar dentro del comando las columnas por las cuales agrupar, estas se reducirán a una sola fila por grupo, en base a los valores coincidentes que tengan. Sobre esto se pueden utilizar funciones agregadas, como `CONTEA` (para contar filas por grupo), y `MIN` (para retornar el valor mínimo del grupo). Si no se añade ninguna función agregada a los otros valores, por default se mantendrá la primer fila de cada grupo[4]. A continuación, unos ejemplos:

```
EMPRESTA CONTEA(Comprador), Producto
  DE Compra
AGRUPATELOS X Producto;
```

Resulta ⇒

2	1
3	0

```
EMPRESTA edad, MIN(altura)
  DE Persona
AGRUPATELOS X edad;
```

Resulta ⇒

33	1.60
40	1.68

```
EMPRESTA edad, altura
  DE Persona
AGRUPATELOS X edad;
```

Resulta ⇒

33	1.71
40	1.68

Se pueden aplicar filtros a los grupos utilizando `TENIENDO`:

```
EMPRESTA edad, MIN(altura)
  DE Persona
AGRUPATELOS X edad
TENIENDO MIN(altura) >1.65;
```

Resulta ⇒

40	1.68

Hay sentencias que reciben dos consultas y retornan otra. `UNETELO CN` retorna todas las filas de ambas consultas juntas. Por ejemplo:

```
EMPRESTA nombre
  DE Persona
  UNETELO CN
EMPRESTA nombreProducto
  DE Producto;
```

Resulta ⇒

John Router
Chris Rivers
Marcel Sands
Plumón
Taza

Este tipo de sentencia solo admite que ambas consultas tengan el mismo numero de columnas y sean del mismo tipo, en caso contrario muestra error.
Para reducir el número de filas que retorna una consulta, puede agregarse al final la sentencia `SOLO` que recibe un número que será el número máximo de filas que entregará (en el orden que vengan):

```
EMPRESTA nombre
  DE Persona
  SOLO 1;
```

Resulta ⇒

John Router

[4] Notar en el tercer ejemplo que, al pedir la altura sin el comando `MIN`, nos quedamos con la altura de la primera fila que posee edad 33.

```
EMPRESTA nombre
  DE Persona
ORDENATELO X nombre PA RIBA
  SOLO 1;
```

Resulta ⇒ `Chris Rivers`

4.2.1. Condiciones

El operador ONDE admite conjunción y disyunción de cláusulas que retornan True o False tales como:

```
EMPRESTA nombre, nombreProducto
DE Persona, Compra, Producto
ONDE RUT = Comprador Y Producto = ID;
```

Donde estas se pueden conjugar atraves de los operadores O e Y, estos son el equivalente en python a AND y OR.
Los operadores válidos en ONDE son `<, >, <=, => !=, =` junto con PARECIO A, ENTRE, EN, EXISTE. Los últimos 2 operadores trabajan sobre el resultado de una consulta.
El uso general de una cláusula es: ((Var1) (Operador) (Var2)), donde Var puede ser un conjunto para el caso del operador EXISTE y EN, un valor definido por el usuario en la consulta o puede ser una columna de las tablas por la cual se quiere filtrar. El operador puede ser cualquiera de los definidos anteriormente.

4.2.2. Consultas Anidadas

Uno de los requerimientos más curiosos que debe tener el Sistema de Manejo de Bases de Datos que desarrolles, es el de ser capaz de responder consultas anidadas escritas en **LCJ**. Una consulta anidada es, esencialmente, una consulta de LCJ que retorna una tabla de una sola columna. Luego, esta tabla puede ser usada al interior de otra consulta en la cláusula `ONDE`.
Existen algunas reglas respecto a las consultas anidadas que te pueden simplificar, sólo un poco, la vida:

- Las consultas anidadas deben estar encerradas con paréntesis.
- Las consultas anidadas sólo pueden retornar tablas con una sola columna.
- Si se ocupan consultas anidadas que retornan más de 1 fila, se debe usar el operador `EN` en la consulta padre.

A continuación un ejemplo:

```
EMPRESTA nombre
  DE Persona
 ONDE RUT EN (
  EMPRESTA RUT
   DE Persona
  ONDE edad <35
      );
```

Resulta ⇒

John Router
Chris Rivers

4.2.3. Resumen Sentencias

Una consulta tendrá en general la siguiente estructura:

```
EMPRESTA [columnas-funciones]
  DE [tablas]
ONDE [condiciones]
AGRUPATELOS X [columnas]
TENIENDO [condiciones]
ORDENATELOS X [columnas] [sentido]
SOLO [numero]
```

A continuación, un resumen de lo que realiza cada comando[5]:

- **EMPRESTA:** Define columnas o funciones que serán retornadas por la consulta.
  - **DIVERGENTE:** Indica que se retornen sólo los valores diferentes de la consulta (sin duplicados). Solo puede ir inmediatamente después de `EMPRESTA`. Ej: `EMPRESTA DIVERGENTE nombre DE Persona`: entrega todos los nombres distintos de la tabla persona.
  - **TODO:** Indica que deben retornarse todas las columnas declaradas en `DE`.
- **DE:** Indica de donde (es decir, de que tablas) obtener los datos
- **ONDE:** Especifica las condiciones que las filas de datos deben cumplir.
- **ORDENATELOS X:** Ordena las filas de la consulta según el criterio especificado.
  - **PA RIBA:** Se puede ulitizar luego de la sentencia ORDENATELOS X, indicando que se deben ordenar de forma ascendiente.
  - **PA BAJO:** Se puede ulitizar luego de la sentencia ORDENATELOS X, indicando que se deben ordenar de forma descendiente.
- **AGRUPATELOS X:** Agrupa la consulta en filas según las columnas entregadas.
  - **MATS:** Selecciona el valor máximo del atributo pedido
  - **MIN:** Selecciona el valor mínimo del atributo pedido.
  - **CONTEA:** Cuenta la cantidad de datos (filas).
  - **PROMEDIO:** Retorna el promedio de valores del atributo pedido.
- **TENIENDO:** Se reserva para funciones de agregado (como MATS, MIN, PROMEDIO y CONTEA), con el fin de filtrar la consulta según el resultado de esas funciones.
- Condiciones:
  - **Y:** Se utiliza para declarar que se debe cumplir más de una condicion.
  - **O:** Se utiliza para declarar que se debe cumplir alguna condición de las expuestas.
  - **ENTRE:** Seleciona valores que se ncuentren dentro del rango de la condición declarada.
  - **PARECIO A:** Se utiliza para seleccionar un valor con un patrón específico.
  - **EXISTE:** Se utiliza con consultas anidadas. Si la consulta anidada (o interna) arroja al menos fila, entonces la consulta externa procede.
  - **EN:** Este operador permite especificar múltiples valores en una ONDE cláusula.
- **SOLO:** Se utiliza para entregar una cantidad máxima de filas de la consulta.
- **UNETELO CN:** Se usa para combinar el resultado de dos o más sentencias EMPRESTA. Las sentencias deben tener las mismas columnas y no hay filas repetidas.
- **UNETELO TODO CN:** La única diferencia con ÚNETELO CN es que esta sentencia si permite filas repetidas.
- **COMUN CN:** Se usa para solo retornar aquellas filas que se encuentran en las dos consultas que le dan.
- **SACALE:** Entrega todas las filas de la primera consulta que no estén en le segunda.

[5] Como se dará cuenta, ninguno de los comandos lleva tilde. No es que a los ayudantes les falte **LET**, sino que para asegurar el funcionamiento del programa, dichos comandos **NO DEBEN** llevar tilde.

4.2.4. Ejemplos

```
Cervezas(nombre, tipo, grados, ciudad-origen)
Produccion(cerveceria, nombre-cerveza, fecha, cantidad)
En-Stock(nombre-cerveza, cantidad, precio-unitario)

EMPRESTA TODO
DE Cervezas
ONDE nombre = 'Baltiloca';

EMPRESTA DIVERGENTE nombre, cantidad
DE Cervezas, En-Stock
ONDE  En-Stock.nombre-cerveza = Cervezas.nombre
Y Cervezas.nombre PARECÍO A "Cristal"
ORDÉNATELOS X cantidad PA RIBA;
```

5. Base de Datos

Usted dispondrá de archivos CSV para el uso de bases de datos. Estos podrán ser de cualquier temática, y deberá poder cargarlos en su programa tratando de ser lo más eficiente en memoria, ya que estos podrían ser de gran tamaño (y quedarse sin RAM en el proceso).

5.1. CSV

Los archivos CSV (Comma Separated Values) son simplemente un formato para guardar tablas de la Base de Datos. Todos siguen la misma estructura:

```
Nombre,Numero Alumno,Mail
Fernando,1363299J,faflorenzano@uc.cl
Nicolas,13638785,negebauer@uc.cl
Guillermo,13634526,gafigueroa@uc.cl
```

El nombre del archivo corresponde al nombre de la tabla (**ALUMNOS.csv** tiene información de la tabla **ALUMNOS**). La primera línea corresponde al HEADER, el nombre de las columnas (campos), y las lineas que siguen son las tuplas (datos) de cada tabla.

En nuestro caso, ocuparemos una versión modificada del CSV, donde en el HEADER se incluirá, además, el tipo de dato de las variables:

```
Nombre:string,Numero Alumno:int,Mail:string
Fernando,1363299J,faflorenzano@uc.cl
Nicolas,13638785,negebauer@uc.cl
Guillermo,13634526,gafigueroa@uc.cl
```

Deberá soportar los tipos: int , float , string.

Un CSV se encuentra bien formateado cuando tiene la misma cantidad de columnas en todas sus líneas. Ustedes pueden asumir que probaremos su tarea con CSV bien formateados. Además, pueden asumir que los nombres de los atributos **no se repetirán entre tablas**[6].**NO SE PERMITE EL USO DE LA LIBRERÍA CSV DE PYTHON**, su uso será fuertemente sancionado.

[6] Por ejemplo, en la tabla `PRODUCTOS` no habrá ningún atributo que tenga el mismo nombre que los atributos de `COMPRAS`. No obstante, **podrían** hacer referencia al mismo. Por ejemplo, `PRODUCTOS.ID` y `COMPRAS.PRODUCTO_ID`.

6. Programación Funcional

Debes modelar toda la implementación del lenguaje LCJ con programación funcional. Se permite el uso de loops (for y while) **solo en listas y generadores por comprensión**. A continuación, se mostrarán ejemplos del uso de loops no permitidos y permitidos[7]:

```
 # Este for no se puede utilizar
for empleo in empleos:
        dame_comida(empleo)

#En cambio, este for si se puede utilizar
nombres_empleados = [x.nombre for x in empleados]

# Recuerde que debe utilizar programacion funcional para toda la implementacion de
# LCJ,como por ejemplo funciones lambda y map:
dobles = map(lambda x: x*2,numeros)
```

7. main.py

¡Esta sección es muy importante! No respetar el formato afectará considerablemente la nota de su tarea.

Deben entregar un archivo llamado **main.py**, que al ejecutarlo lea un archivo llamado **consultas.txt** y escriba en el directorio principal un archivo llamado **resultados.txt**.

7.1. consultas.txt

Este archivo tendrá el siguiente formato (donde N es un número que representa el número de tablas y K un número que representa un número de consultas):

```
N
nombretabla1.csv
nombretabla2.csv
...
nombretablaN.csv
K
consulta1;
consulta2;
...
consultaK;
```

7.2. resultados.txt

Este archivo debe tener el siguiente formato:

```
------------ Consulta 1 -----------
RESULTADO CONSULTA 1
------------ Consulta 2 -----------
RESULTADO CONSULTA 2
...
----------- Consulta K -----------.
RESULTADO CONSULTA K
```

[7] Nos interesa que no usen los ciclos `for` y `while` en lo que son las funcionalidades lógicas del programa. Pueden utilizarlos para imprimir valores, por ejemplo.

7.3. Consultas fallidas, ejecución y ejemplo

Su programa debe:

1. Leer consultas.txt
2. Generar resultados.txt y no caerse al fallar en hacer alguna consulta, la idea es que pueda seguir con las otras.
3. Mantener la ejecución a la espera de nuevas consultas cuyos resultados se imprimirán en pantalla. Un ejemplo de manejo de errores a continuación:

```
for i in ["1", 2, 4, "3", 5, 6, "4"]:
    try:
        print(i + 1)
    except:
        print("Error")
# Se imprime lo siguiente
Error
3
5
Error
6
7
Error
[Finished in 0.1s]
```

Un ejemplo[8] de la lógica de main.py a continuación:

```
if __name__ == "__main__":
    # Abre consultas.txt
    for i in consultas:
        try:
            procesar_consulta(i)
            # Escribe los resultados en resultados.txt
        except:
            imprimir_fallo()
            # Escribe que la consulta falló en resultados.txt
    in_loop =  True
    while in_loop:
        consulta = input()
        if consulta:
            procesar_consulta_consola(consulta)
            # Muestra los resultados en la consola
        else:
            in_loop = False
```

De esta forma, el archivo **resultados.txt** debería verse como:

```
-------- CONSULTA 1 ----------
10 | CALAMA
13 | COPIAPO
19 |  LA SERENA
21 | COQUIMBO
25 | OVALLE
32 | SALAMANCA
34 | VALPARAISO
------- CONSULTA 2 ----------
FALLIDA
------- CONSULTA 3 ----------
COMPLETO ITALIANO | 1000 | 2016
```

[8] Es solo un ejemplo, pueden escribirlo de la forma que quieran, siempre que se cumpla el procedimiento pedido.

Junto al enunciado, hay algunos archivos de ejemplo con algunas consultas y sus resultados. **Es importante que su programa funcione con cualquier tipo de archivo (siempre que mantengan la estructura detallada anteriormente). No lo adapten a los archivos de ejemplo.**

8. Restricciones y alcances

- Tu programa debe ser desarrollado en Python 3.4
- Esta tarea es estrictamente individual y está regida por el Código de Honor de la Escuela: Clickear para Leer.
- Su código debe seguir la guía de estilos `PEP8`
- Si no se encuentra especificado en el enunciado asuma que el uso de cualquier librería Python está prohibido. Pregunte por foro si se pueden usar librerías específicas.
- El ayudante puede castigar el puntaje[9] de tu tarea si le parece adecuado. Se recomienda ordenar el código y ser lo más claro y eficiente posible en la creación de algoritmos.
- Debe adjuntar un archivo `README.md` donde se comenten los alcances y el funcionamiento de su sistema (*i.e.* manual de usuario) de forma *concisa* y *clara*.
- Cree un módulo para cada conjunto de clases. Divídalas por las relaciones y los tipos que poseen en común.
- Cualquier aspecto no especificado queda a su criterio, siempre que no pase por encima de ningún otro.

9. Entrega

- **Fecha/hora:** 2 de Mayo - 23:59 hrs
- **Lugar:** GIT - Carpeta: Tareas/T03

No subir la carpeta Ejemplos. Tareas que no cumplan con las restricciones señaladas en este enunciado tendrán la calificación mínima (1.0).

[9] Hasta −5 décimas.