Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación
2do cuatrimestre, 2014

Fecha de entrega: 9 de septiembre

Introducción

Una conocida empresa de software (Ooggle) comenzó a trabajar en el tema de mapear las distintas ciudades del mundo. Para ello crearon Ooggle Amps, una aplicación que muestra distinto tipo de información a lo largo del globo.

Para su funcionamiento esta empresa posee millones de documentos distribuidos en miles de computadores, con información de los distintos elementos que conforman un mapa como calles, edificios, monumentos, entre otros, que luego servirán para dibujar lo que el usuario necesite visualizar.

Lamentablemente, el Framework que están utilizando fue desarrollado en `C++`, por lo tanto nos piden que actualicemos su sistema al flamante `Haskell`, ya que conocen las claras ventajas de utilizar este lenguaje.

Este trabajo se enfocará en la creación de un conjunto de funciones que permitirá trabajar con estos datos utilizando una versión de la técnica **MapReduce**.

Diccionarios

Como primer paso, modelaremos diccionarios (tipo `Dict k v`) que nos servirán para facilitar la tarea de creación del Framework. Para ello definimos el tipo `Dict k v` como un renombre de lista de pares k v, en la cual no habrá claves repetidas:

```
type Dict k v = [(k,v)]

ejemplo :: Dict String [Int]
ejemplo = [("ayudantes",[2,2,2]),("profesores",[2])]
```

En este ejemplo, se trata de un diccionario con claves de tipo `String` y valores de tipo `[Int]`.

MapReduce

MapReduce es una técnica de programación que permite procesar grandes cantidades de datos en paralelo utilizando el poder de cómputo de varias máquinas. Al utilizar la técnica, se deberán repensar los algoritmos para poder aprovechar al máximo el paralelismo.

La técnica implica distribuir en varias máquinas el procesamiento de un conjunto de datos potencialmente muy grande. Para ello, suele utilizarse un framework que se encarga de la distribución de la carga, la ejecución de los algoritmos en cada máquina y, luego, de la combinación de resultados.

A este framework se lo provee de dos funciones: un *Mapper* y un *Reducer*. Estas dos funciones se ejecutarán en distintas etapas del proceso. En una primera instancia se reparte el conjunto de datos de entrada para enviar a distintos procesos que ejecuta el Mapper. Tras un procesamiento intermedio, todos los datos van a parar a un servidor central, donde un Reducer los combina para obtener el resultado final.

En este trabajo nos enfocaremos en simular este framework de programación, y luego lo utilizaremos para resolver problemas sencillos sobre este nuevo modelo. Nuestra versión del framework de MapReduce constará de cuatro procesos fundamentales:

Distribución de Carga

`[a] → [[a]]`

El primer paso consiste en tomar una lista de elementos a procesar (`[a]`) y convertirlo en una lista de listas. De esa manera, las listas resultantes (que contienen parte de los datos a procesar) serán enviadas a distintas máquinas, que se encargarán de hacer el trabajo.

Mapeo

`[a] → [(k,[v])]`

Cada máquina recibe una lista de elementos que debe procesar. Para ello, aplica a cada elemento una función de *mapeo*. El tipo de esta función será `Mapper a k v`, definido como sigue:

```
type Mapper a k v = a → [(k,v)]
```

Es importante notar que esta función debe devolver una lista de tuplas con información, no un diccionario, ya que pueden generarse claves repetidas.

Al aplicar la función a todos los elementos, se consigue una lista de listas de pares k v, es decir `[[(k,v)]]`. Inmediatamente después se unirán estos pares, agrupados por clave. De esta manera, el resultado de esta etapa es una lista de pares k [v] por cada computadora, o sea `[(k,[v])]`.

Combinación

`[[(k,[v])]] → [(k,[v])]`

Una vez que se recibe la información de todas las computadoras, se procede a combinar y ordenar por clave los resultados obtenidos.

Reducción

`[(k, [v])] → [b]`

Finalmente, por cada uno de los pares `(k,[v])` obtenidos del paso anterior, aplicaremos una función de *reducción*. El tipo de esta función será `Reducer k v b`, definido como sigue:

```
type Reducer k v b = (k, [v]) → [b]
```

Luego de aplicar este proceso contaremos con una `[[b]]`, que deberá ser combinada (`concat`), y de esta manera obtendremos el resultado final.

Ejemplo

Supongamos que tenemos un registro gigantesco de las ocasiones en que cada usuario visitó una ciudad (en formato de pares *nombre,ciudad*), y queremos saber la cantidad de visitas que tuvo cada ciudad.

En primer lugar, el framework reparte el conjunto de datos de entrada en los procesos de mapeo que haya disponibles, donde la función *mapper* descarta el nombre de usuario y emite un par donde la clave es la ciudad visitada y un valor cualquiera que represente la visita (en este caso un 1). Hecho esto, cada proceso agrupa sus resultados por clave y los envía a la siguiente etapa.

Acto seguido, el *combiner* recolecta los resultados, agrupando los de misma clave y ordenándolos para que en una última etapa el *reducer* sume la cantidad de visitas individuales de cada ciudad y emita los resultados finales.

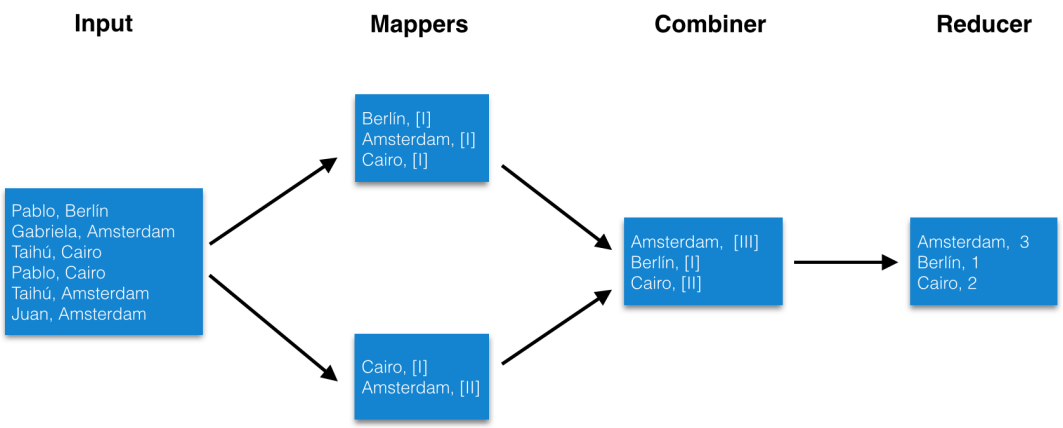

Figura 1: Flujo de datos en un proceso MapReduce

Ejercicios

Ejercicio 1

Implementar la función `belongs :: Eq k => k -> Dict k v -> Bool` que devuelve verdadero en caso de encontrar la clave de tipo k en el diccionario. Definir además `(?)` de manera que funcione como operador infijo para esta operación, tomando primero el diccionario y luego la clave a buscar. Ejemplo:

```
*Main> [("calle",[3]),("ciudad",[2,1])] ? "ciudad"
True
```

□

Ejercicio 2

Definir la función `get :: Eq k => k -> Dict k v -> v` de manera que dada una clave devuelva su valor asociado, asumiendo que se encuentra definido. Definir además el operador infijo (!), que tome primero el diccionario y luego la clave. Ejemplo:

```
*Main> [("calle","San␣Blas"),("ciudad","Hurlingham")] ! "ciudad"
"Hurlingham"
```

□

Ejercicio 3

Definir `insertWith :: Eq k => (v -> v -> v) -> k -> v -> Dict k v -> Dict k v` que se encargue de definir un nuevo valor para una clave dada, utilizando la función que reciba como parámetro. En caso de no existir, deberá agregarse como único valor. Ejemplos:

```
*Main>  insertWith (++) 1 [99] [(1, [1]), (2, [2])]
[(1,[1,99]),(2,[2])]
*Main>  insertWith (++) 3 [99] [(1, [1]), (2, [2])]
[(1,[1]),(2,[2]),(3,[99])]
```

□

Ejercicio 4

Definir `groupByKey :: Eq k => [(k,v)] -> Dict k [v]`. Esta función debe agrupar los datos por clave, generando un diccionario que asocia a cada clave con la lista de todos sus valores.

```
*Main> groupByKey [("calle","Jean␣Jaures"),("ciudad","Brujas"),
                   ("ciudad","Kyoto"),("calle","7")]
[("calle",["Jean␣Jaures","7"]),("ciudad",["Brujas","Kyoto"])]
```

□

Ejercicio 5

Definir `unionWith :: Eq k => (v -> v -> v) -> Dict k v-> Dict k v -> Dict k v`. Esta función debe unir dos diccionarios, utilizando la función que reciba como parámetro en caso de conflicto entre claves. Ejemplo:

```
*Main> unionWith (+) [("rutas",3)] [("rutas", 4), ("ciclos", 1)]
[("rutas",7),("ciclos",1)]
```

□

Ejercicio 6

Implementar la función `distributionProcess :: Int -> [a] -> [[a]]` que, dada una lista de elementos, divide la carga **de manera balanceada** entre una cantidad determinada de máquinas. No es importante el orden en que quedan distribuidos los elementos, pero sí es importante que el balanceo sea óptimo. Por ejemplo:

```
*Main> distributionProcess 5 [1,2,3,4,5,6,7,8,9,10,11,12]
[[1,6,11],[2,7,12],[3,8],[4,9],[5,10]]
```

Recomendamos utilizar la función `replicate :: Int -> a -> [a]` para generar listas vacías, y recorrer 1 vez la lista original para ir repartiendo los elementos en las listas de manera rotativa.□

Ejercicio 7

Definir `mapperProcess :: Eq k => Mapper a k v -> [a] -> [(k,[v])]`. Está función debe aplicar el *mapper* a cada uno de los elementos de tipo `a` y luego agrupar los resultados por clave. □

Ejercicio 8

Implementar `combinerProcess :: (Eq k, Ord k) => [[(k, [v])]] -> [(k,[v])]` que se encarga de recibir la información resultante de cada máquina, y combina estos resultados de manera que queden agrupados por clave, y ordenados de manera creciente según la clave.

□

Ejercicio 9

Definir `reducerProcess :: Reducer k v b -> [(k, [v])] -> [b]` Esta etapa se encarga de aplicar el *reducer* sobre cada par de elementos, y luego aplanar el resultado para unificar las soluciones. □

Ejercicio 10

Finalmente, podemos implementar el framework completo. Definir la función
`mapReduce :: (Eq k, Ord k) => Mapper a k v -> Reducer k v b -> [a] -> [b]`
tomando como guía la explicación de la introducción y los tipos de los argumentos. Tener en cuenta que la carga será distribuída en 100 maquinas. □

Utilización

En los siguientes ejercicios, deberán definirse funciones en términos de mapReduce, es decir

```
nuevaFuncion = mapReduce mapper reducer
```

También es posible que se necesite hacer una composición de aplicaciones de mapReduce, pero no está permitido procesar de otra manera la información, ya que iría en contra del espíritu de la técnica.

Ejercicio 11

Definir en términos de `mapReduce` una función que, dada una lista de nombres de monumentos visitados, calcule cuántas visitas tuvo cada monumento. Por ejemplo:

```
*Main> visitasPorMonumento ["m1","m2","m3","m2"]
[("m1",1),("m2",2),("m3",1)]
```

□

Ejercicio 12

Definir en terminos de `mapReduce` una función que devuelva una lista ordenada según cuántas veces haya sido visitado un monumento.

```
*Main> monumentosTop ["m1","m2","m3","m2","m3","m3","m1","m4"]
["m3","m2","m1","m4"]
```

□

Ejercicio 13

Contamos con el tipo `Structure`, definido de la siguiente manera:

```
data Structure = Street | City | Monument
```

Definir en términos de `mapReduce` la siguiente función, que determine cuántos monumentos existen por cada país. Se puede asumir que siempre estará definida la clave `country`.

```
monumentosPorPais :: [(Structure, Dict String String)] → [(String, Int)]
```

Por ejemplo:

```
items :: [(Structure, Dict String String)]
items = [
    (Monument, [
      ("name","Obelisco"),
      ("latlong","-36.6033,-57.3817"),
      ("country", "Argentina")]),
    (Street, [
      ("name","Int. Guiraldes"),
      ("latlong","-34.5454,-58.4386"),
      ("country", "Argentina")]),
    (Monument, [
      ("name", "Estatua San Martin"),
      ("country", "Argentina"),
      ("latlong", "-34.6033,-58.3817")]),
    (City, [
      ("name", "Paris"),
      ("country", "Francia"),
      ("latlong", "-24.6033,-18.3817")]),
    (Monument, [
      ("name", "Bagdad Bridge"),
      ("country", "Irak"),
      ("new_field", "new"),
      ("latlong", "-11.6033,-12.3817")])
    ]

*Main> monumentosPorPais items
[("Argentina",2),("Irak",1)]
```

□

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante Se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.