Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura
Departamento de Ciencias de la
Computación

Análisis de Lenguajes de Programación

TP FINAL

Cassinerio Marcos

Enero 2024

1 Descripción del proyecto

1.1 Introducción

El trabajo elegido consiste en el calculo de deudas desde la perspectiva de un usuario.
Este consiste inicialmente en una definición de personas y grupos entre personas. En estos se cargan gastos tanto ajenos como propios.
De aquí se pueden calcular las deudas finales, tanto en grupos como con otra persona, se pueden consultar todos los datos cargados (personas, grupos y gastos), ver los miembros de grupos, y ver las movimientos con otra persona o en un grupo.

1.2 Vista Técnica

Como se dijo anteriormente, en este proyecto inicialmente se definirán personas y grupos, y de ahí se podrán cargar gastos en estos.
La idea es que una vez definidos, se carguen esos datos a un grafo. En este las personas serán los vértices y las deudas las aristas.
Las deudas se calcularan de la siguiente manera, si en un grupo con 4 personas (teniendo en cuenta a uno mismo), uno de ellos hace un gasto compartido de 40 pesos por ejemplo, se generaran aristas desde los 3 restantes hacia el que pago el gasto con un valor de 10.

1.2.1 Calculo de Deudas

Para calcular las deudas finales se utilizo un algoritmo que, una vez creadas las deudas correspondientes elimine todos los ciclos desde los de mayor longitud hasta longitud 2, y luego los caminos cuyas aristas tengan el mismo valor.
Los ciclos se eliminaran de la siguiente manera. Una vez conseguido el ciclo, se tomara el mínimo valor de todas las aristas que lo componen, y se les restara ese valor. Cabe aclarar que las aristas que queden con valor 0 serán eliminadas, de esta manera nos aseguramos que se restara por lo menos una arista.
En el caso de los caminos, se opto por tener en cuenta únicamente los caminos cuyas aristas tengan el mismo valor, ya que sino no se estaría disminuyendo la cantidad máxima de aristas, solo se estarían cambiando. Esto se debe a que si por ejemplo, `A` le debe 20 a `B` y `B` le debe 10 a `C`, una manera de abordar este caso seria hacer que `A` le deba 10 a `B` y a `C`. En caso contrario, si `B` le debiese 20 a `C`, se podrían eliminar ambas aristas creando una única de `A` a `C` con valor 20.

2 Instalación

Para la instalación es necesario tener instalado en nuestro sistema lo siguiente:

- Haskell
- Stack
- Happy

Una vez descargados, se podrá descargar el código fuente del repositorio aquí.
Finalmente, se deberá ejecutar el comando `stack run` en la terminal.

3 Manual de uso

El uso de este proyecto consiste en 2 partes. Inicialmente esta la carga de un archivo de deudas, este deberá tener una extensión `.dbt`. Luego, se podrá hacer uso del interprete para correr, entre otras, las acciones previamente explicadas.
Una vez teniendo todo descargado, se deberá correr el comando `stack run`. Este tomara por defecto el archivo `Default.dbt` situado en `Ejemplos` y generara las deudas adecuadas. También, al correr ese comando, se entrara en el interprete.
Si se desea cargar un archivo diferente, se podrá crear uno nuevo y con el comando `:load archivo` cargarlo, eliminando los datos generados por el archivo anterior.
Antes de hablar sobre cada uno de estos ítem por separado, se presentara la gramática con la que se trabajo.

$$
\begin{array}{rcl}
digit & ::= & \text{'0'}|\text{'1'}|...|\text{'9'} \\
val & ::= & digit \mid digit\ val \\
\\
letter & ::= & \text{'a'}|\text{'b'}|...|\text{'z'} \\
var & ::= & letter \mid letter\ var \\
\\
names & ::= & var\ \text{','}\ names \mid var \\
\\
def & ::= & \text{'}DEFINEP\text{'}\ var \\
 & | & \text{'}DEFINEG\text{'}\ var\ \text{'['}\ names\ \text{']'} \\
 & | & \text{'}DEBTP\text{'}\ var\ val \\
 & | & \text{'}DEBTG\text{'}\ var\ var\ val \\
 & | & \text{'}EXPENSE\text{'}\ var\ val \\
\\
op & ::= & \text{'}CALCULATE\text{'}\ var \\
 & | & \text{'}CALCULATEALL\text{'} \\
 & | & \text{'}REGISTRY\text{'}\ var \\
 & | & \text{'}MEMBERS\text{'}\ var
\end{array}
$$

3.1 Archivos de deuda

El archivo que se carga en un inicio esta pensado para que se carguen únicamente definiciones y gastos. Y estos se escribirán de la manera que lo muestra la gramática `def`. Estos se correrán en el orden en el que fueron escritos, así que si se genera un gasto de una persona, previamente se tendría que haber definida a esta. A continuación se explicara como se unas las operaciones:

- `DEFINEP nombre`: Crea una persona con el nombre "nombre"
- `DEFINEG nombre participantes`: Crea un grupo con el nombre "nombre" y cuyos participantes son "participantes" (además de uno mismo)
- `DEBTP nombre monto`: Crea un gasto entre "nombre" y uno mismo pagado por el otro con un monto "monto"
- `DEBTG nombre grupo monto`: Crea un gasto en el grupo "grupo" pagado por "nombre" con un monto "monto"

- `EXPENSE nombre monto`: En este caso, "nombre" puede ser de una persona o un grupo. Allí se crea un gasto pagado por uno mismo con el monto "monto"

3.2 Interprete

En el interprete se podrán escribir comandos que sirvan para obtener datos a partir de lo cargado en el archivo. Estas son, entre otros, los que se ven en la gramática `op`. A continuación se explicara como se unas las operaciones:

- `CALCULATE nombre`: Calcula las deudas finales sobre la persona o grupo "nombre"
- `CALCULATEALL`: Calcula las deudas finales sobre todas las personas y grupos
- `REGISTRY nombre`: Devuelve las operaciones hechas sobre la persona o grupo "nombre"
- `MEMBERS nombre`: Devuelve los participantes del grupo "nombre" o una lista con ese mismo elemento de ser "nombre" una persona

Adicionalmente, además del comando ya mencionado `:load`, están los siguientes:

- `:browse` Muestra los nombre de los grupos y personas
- `:print` Imprime el entorno
- `:quit` Sale del interprete
- `:help` Muestra la lista de comandos

4 Organización de los archivos

El proyecto consiste de 5 módulos, un parser y el modulo `Main`, donde se implementa la carga del archivo y el interprete. Este ultimo se encuentra en la carpeta `app`, mientras que el resto en `src`. A continuación se hablara brevemente de los módulos:

- `Common`: Incluye las definiciones de los tipos de datos utilizados.
- `Def`: Incluye la función que evalúa las operaciones de creación de deudas. Además se hace uso de las Monadas creadas y se crea el entorno.
- `Graph`: Incluye las funciones que evalúan las operaciones de consulta de deudas.
- `Monads`: Define las clases de monadas.
- `PrettyPrinter`: Modulo correspondientes al Pretty Print que se encarga de mostrar en pantalla los valores solicitados.

También hay una carpeta `Ejemplos` con el archivo que se correrá por defecto al inicial el programa.

5 Decisiones de diseño

5.1 Estructuras utilizadas

Para simplicidad de código se opto por utilizar la misma estructura para parsear la entrada de tanto un archivo como del interprete, esta está encapsulada en el siguiente data:

```
type Name = String

data Exp = DefineP Name
         | DefineG Name [Name]
         | DebtP Name Int
         | DebtG Name Name Int
         | Expense Name Int
         | Calculate Name
         | CalculateAll
         | Registry Name
         | Members Name
  deriving (Eq, Show)
```

Ahora, una vez creados los `Exp` y habiéndose guardado las operaciones en el entorno (definiciones y gastos). Al momento de parsear las operaciones de consulta, se toma el entorno y se crean las deudas correspondientes entre las personas, y estas se ven en la siguiente estructura:

```
type Name = String

data Person = Self | Other Name
  deriving (Eq, Show)

type Operations = M.Map Person (M.Map Person Int)

type Graph = ([Person], Operations)
```

En el type `Graph` quedaran plasmadas las personas como vértices del grafo y las deudas como aristas.
Fuera del hecho de plantear el problema de deudas como un grafo, la decisión de diseño mas importante aquí fue el uso del modulo `Data.Map.Strict` para las aristas. Esta decisión se debió a que al momento de detección de ciclos y caminos, resultaría muy costosa la consulta a la lista de aristas si no usaba la estructura adecuada. Un diccionario pareció ser la estructura que mejor se acoplaría a estas necesidades.

5.2 Guardado en el entorno

Teniendo en cuenta las operaciones de definición, tanto el hecho de definir personas o grupos y cargar gastos sobre estos, se decidió hacer la estructura para el entorno de la siguiente manera:

```
type Name = String

data Person = Self | Other Name
  deriving (Eq, Show)

type Op = (Person, Int)

type Env = [(Name, ([Name], [Op]))]
```

Este type consiste en una lista de tuplas, donde, en el primer elemento de la tupla se encuentra en nombre de la persona o grupo, y en el segundo elemento los participantes y las operaciones hechas.
Aquí, al momento de definir una persona o grupo, se creara un nuevo valor en `Env`. Si es una persona, la lista de nombres en el segundo elemento de la tupla sera únicamente el nombre de la persona y en caso de ser un grupo, ahí se cargaran los nombres de los participantes. En ambos casos, la lista de operaciones quedara vacía.
Cuando se cree una deuda, representada con `DEBTP` o `DEBTG`, se agregara respectivamente en la persona o grupo, el gasto hecho. Y en caso de hacer uno mismo un gasto, representado por `EXPENSE`, se agregara un gasto hecho por la personas `Self`.
Habiendo explicado como funcionan las operaciones `DEBT` y `EXPENSE`, se puede apreciar el por que definir un data `Person`, ya que un gasto podría ser hecho por uno mismo (`Self`) u otra persona (`Other "name"`).

Referencias y Bibliografía

[1] UNR, FCEIA, LCC 2023, Apuntes de clase de la materia ALP

[2] UNR, FCEIA, LCC 2023, TP3 ALP

[3] UNR, FCEIA, LCC 2023, TP4 ALP

[4] DebtG: A Graph Model for Debt Relationship - URL: https://www.mdpi.com/2078-2489/12/9/347