UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Laboratorio 2, Paradigma Lógico

Integrantes: Aldo Castillo

Curso: Paradigmas de Programación

Profesor: Roberto González

24 de Junio de 2020

Tabla de contenidos

1. Introducción **1**

1.1. Motivación 1

1.2. Descripción del problema 1

1.3. Solución propuesta 2

1.4. Objetivo general y alcances del proyecto 2

1.4.1. Objetivo general 2

1.4.2. Objetivos específicos 2

1.4.3. Alcances 3

1.4.4. Metodologías y herramientas 3

2. Desarrollo de la experiencia **4**

2.1. Fundamentos teóricos 4

2.1.1. TDA's 4

2.1.2. Listas 6

2.1.3. Recursión 6

2.2. Desarrollo de la solución 8

2.2.1. TDA Zona 8

2.2.2. TDA Objeto 8

2.2.3. TDA Archivo 8

2.2.4. Selectores 8

2.2.5. Modificadores 8

2.3. Funciones Principales 9

2.3.1. GitInit 9

2.3.2. GitAdd 9

2.3.3. GitCommit 10

2.3.4. GitPush 11

2.4. Función Extra 12

2.4.1. GitPull 12

3. Exposición de Resultados **13**

4. Análisis de resultados **15**

5. Conclusión **16**

1. Introducción

1.1. Motivación

El principal motivo por el cual se realiza este proyecto es desafiar los conocimientos sobre distintos lenguajes y el poder implementarlos para lograr llegar al final del proyecto de manera satisfactoria. A su vez, otra motivación es el hecho de crear algo que podrá ser usado por un usuario, logrando así un producto final funcional.

1.2. Descripción del problema

En esta ocasión enfrentaremos la problemática que impone la abstracción de Git (Figura 1) abarcado desde el punto de vista del ramo Paradigmas de programación, tal como dice el nombre del ramo, se verán distintos paradigmas (en distintos lenguajes) ejemplificados en esta problemática, algunos de estos son: Paradigma Funcional (scheme – R6RS), Paradigma Lógico (Prolog), Paradigma Orientado a Objetos (Java) entre otros. A final de semestre se espera poder complementar principalmente estos tres paradigmas (y lenguajes) y así tener una app totalmente funcional con interfaz gráfica aplicada con *C#*

Para esta segunda instancia, abordaremos el problema con el Paradigma Lógico con el cual deberemos implementar ciertas funciones que se detallaran más adelante. El Paradigma Lógico como dice su nombre solo entrega valores de verdadero o falso, en este paradigma se trabaja lo que se llama Mundo cerrado, es decir que el programa solo conoce lo que se le da a conocer, por ejemplo, si se le dice que el color rojo se representa como (0,0,255), para el programa eso es verdadero.

Usando la sintaxis de Prolog rigiéndose por el paradigma Lógico, se deberá solucionar la problemática de representación de las zonas de trabajo, representación de archivos y las funciones para comunicar estos.

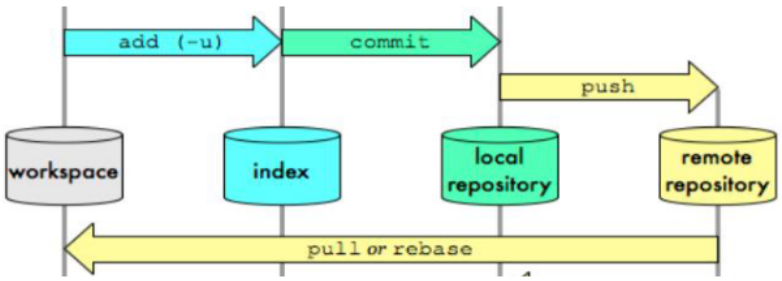

Figura 1: Abstracción de Git.

1.3. Solución propuesta

Para abordar esta problemática, la solución de dividió en varias partes:

1. TDA Zona
2. TDA Archivos
3. TDA Objeto
4. Funciones complementarias

Estas se detallarán más adelante.

1.4. Objetivo general y alcances del proyecto

1.4.1. Objetivo general

El objetivo general de esta primera parte del proyecto es generar el núcleo de la app (TDA Zona, TDA Archivos, TDA Objeto y sus relaciones), además de comprobar todo lo realizado para que este dentro de los estándares pedidos.

1.4.2. Objetivos específicos

1. Crear y representar de manera adecuada el TDA Zonas.
2. Crear y representar de manera adecuada el TDA Archivo.
3. Crear y representar de manera adecuada el TDA Objeto.

4. Realizar funciones Init, Add, Commit, Push y Pull, además de sus funciones auxiliares.

1.4.3. Alcances

En este informe se quiere dar a conocer el funcionamiento de esta segunda parte del proyecto, mostrando sus fundamentos teóricos, métodos y herramientas ocupadas. En cuanto al laboratorio se espera realizar un escenario funcional con los estándares de validación necesarios.

1.4.4. Metodologías y herramientas

Para la realización de este laboratorio se utilizaron los apuntes de clase, sumado a las librerías de SWI-Prolog para el manejo de distintos tipos de datos y funciones.

2. Desarrollo de la experiencia

2.1. Fundamentos teóricos

Para la realización de este laboratorio, se ocuparon diversos tipos de datos y funciones nativas de SWI-Prolog. Estas se detallarán a continuación.

2.1.1. TDA's

Formalmente, un TDA es...

"una clase de objetos cuyo comportamiento lógico está definido por un conjunto de valores y un conjunto de operaciones"
(Dale & Walker 1996, p. 3)

Esto quiere decir que un TDA...

1. Tiene valores.
2. Tiene operaciones.
3. Cumple una función determinada.
4. Puede ser implementado con diferentes tipos de datos u otros TDA's.

El TDA está compuesto por 6 niveles:

1. Representación.
2. Constructor.
3. Pertenencia.
4. Selectores.
5. Modificadores.
6. Otras operaciones.

Para efectos de este laboratorio, el TDA usado es el TDA Zona, TDA Archivos y TDA Objeto.

La representación del TDA Zona sera una matriz (lista de listas) que tendrán el orden: Nombre del repositorio, Autor del repositorio, Fecha de creación, TDA Objeto (WorkSpace), TDA Objeto (Index), TDA Objeto (Local Repository) y TDA Objeto (Remote Repository) (Figura 2).

La representación del TDA Objeto (workspace, index, local repository, remote repository) sera una lista de lista de listas, es decir lista de listas que contendrán: commit (comentario), lista de TDA Archivo (Figura 3).

La representación del TDA Archivos sera una matriz (lista de listas) que tendrán el orden: Nombre de Archivo, contenido del archivo (Figura 4).

```
%TDA Zona
% [Nombre Repo, Autor, Fecha, [Work Space, Index, Local Repository,
% Remote Repository]] [String, String, [TDA Objeto, TDA Objeto, TDA
% Objeto, TDA Objeto]]
```

Figura 2: Representación TDA Zona.

```
%TDA Objeto
% [Commit, Lista de Archivos]
% [String, Lista de TDA Archivo]
```

Figura 3: Representación TDA Objeto.

```
%TDA Archivo
% [Nombre, Contenido]
% [String, String]
```

Figura 4: Representación TDA Archivo.

2.1.2. Listas

Tal como dice su nombre, son una lista de elementos de la forma:

Elemento X Lista | Null

Las listas tienen como ...

1. Dominio: Elemento X Lista | Null.
2. Recorrido: Lista.

[Cabeza | Cola]

Figura 5: Representación Listas en Prolog.

2.1.3. Recursión

Una función recursiva es aquella que se llama a si misma con un cambio en sus parámetros para llegar a un caso base, es por esto que las dos partes principales de la recursión son:

1. Caso base.
2. Llamada Recursiva.

Existen distintos tipos de recursión, pero aquí se detallarán tres:

2.1.3.1 Recursión Lineal o Natural

La recursión Lineal o Natural es aquella que en cada llamada recursiva deja un Stack o estado pendiente los cuales se solucionaran una vez llegue al caso base y retrocederá con el resultado. El uso de stacks hace que la recursión se acote (como máximo, con datos simples) a 45 recursiones.

2.1.3.2 Recursión de cola

La recursión de cola es aquella que en cada llamada recursiva modifica un acumulador, este acumulador hace el trabajo del stack y así no se dejan estados pendientes, esto permite llamados recursivos mucho más grandes en contrario a la recursión Lineal.

2.1.3.3 Recursión Arbórea

La recursión Arbórea es aquella que se llama recursivamente 2 o más veces haciendo una operación entre estas llamadas.

Un ejemplo de recursión usada en el programa seria:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funcion que reemplaza un elemento de una lista
replace([_|Xs], 0, NEW, [NEW|Xs]).

replace([X|XS], COORD, NEW, [X|YS]):-
    COORD > -1,
    NI is COORD - 1,
    replace(XS, NI, NEW, YS), !.

replace(L, _, _, L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 6: Ejemplo de recursión.

2.2. Desarrollo de la solución

Para la mayoría de las funciones aquí expuestas se usaron las funciones getters y setters.

2.2.1. TDA Zona

La representación del TDA Zona sera por medio de una especie de matriz que almacenara el Nombre del repositorio, el autor, su fecha de creación y las cuatro partes de Git (Work Space, Index, Local Repository, Remote Repository) (Figura 2).

2.2.2. TDA Objeto

La representación del TDA Objeto sera por medio de una lista con dos elementos, primero el Commit (variara en cada objeto) y segundo una lista de TDA Archivo (Figura 3).

2.2.3. TDA Archivo

La representación del TDA Archivo sera por medio de una lista con dos elementos, primero el nombre del archivo y segundo el contenido del archivo (Figura 4).

2.2.4. Selectores

Para cada TDA se generaron una serie de selectores (Getters), con los cuales se podrá obtener por ejemplo el commit para el TDA Objeto, el nombre del archivo para el TDA Archivo en otros.

2.2.5. Modificadores

Para cada TDA se generaron una serie de modificadores (Setters), con los cuales se podrá modificar por ejemplo el commit para el TDA Objeto, el nombre del archivo para el TDA Archivo en otros.

2.3. Funciones Principales

2.3.1. GitInit

Función que dado un Nombre de repositorio y Autor (además de la variable de salida), genera un repositorio vació (una instancia de TDA Zona).

```
?- gitInit(NombreRepo, Autor,SALIDA)
```

Figura 7: Función gitInit.

```
%###############################################################
% Funcion que inicializa un repositorio con autor y nombre dados
gitInit(NombreRepo, Autor, RepoOutput):-
    tiempo(T),
    RepoOutput = [NombreRepo, Autor, T, [[], [], [], []] ].


% gitInit("Lab", "Pedro", OUT).
% gitInit("Marco teorico", "Nadie", OUT).
% gitInit("31 Minutos", "Tulio Triviño", OUT).
%###############################################################
```

Figura 8: Código función gitInit.

2.3.2. GitAdd

Esta función añade los cambios locales del WorkSpace al Index.

```
gitAdd(RepoInput, Archivos, SALIDA)
```

Figura 9: Función gitAdd.

```
%###############################################################
% Funcion que pasa el contenido de Work Space a Index
gitAdd(RepoInput, _, RepoOutput):- %Archivos
    get_Work_Space(RepoInput, WS),
    set_Work_Space(RepoInput, [], RepoOutputAux),
    get_Index(RepoInput, I),
    (I = []; WS = []),
    append(I, WS, NEWINDEX),
    set_Index(RepoOutputAux, NEWINDEX, RepoOutput).
```

Figura 10: Código función gitAdd.

2.3.3. GitCommit

Función que genera un commit almacenando los cambios respectivos con un mensaje descriptivo en Local Repository.

```
| gitCommit(RepoInput, Mensaje, SALIDA)
```

Figura 11: Función gitCommit.

```
%#############################################################
% Funcion que agrega el Mensaje (Commit) al contenido de index y lo
% traspasa a Local Repository
gitCommit(RepoInput, Mensaje, RepoOutput):-
    get_Index(RepoInput, I),
    set_Commit(I, Mensaje, NEWINDEX),
    set_Index(RepoInput, [], RepoOutputAux),
    set_Local_Repository(RepoOutputAux, NEWINDEX, RepoOutput).
```

Figura 12: Código función gitCommit.

2.3.4. GitPush

Función que envía los commit desde Local a Remote repository.

```
| gitPush(RepoInput, SALIDA)
```

Figura 13: Función gitPush.

```
% ###############################################################
% Funcion que mueve el Local Repository a Remote Repository
gitPush(RepoInput, RepoOutput):-
    get_Local_Repository(RepoInput, LR),
    set_Local_Repository(RepoInput, [], RepoOutputAux),
    set_Remote_Repository(RepoOutputAux, LR, RepoOutput).
```

Figura 14: Código función gitPush.

2.3.4.1 Git2String

Función que representa el repositorio de entrada como string.

```
| git2String(RepoInput, STRING)
```

Figura 15: Función git2String.

```
% ###############################################################
%  Funcion que representa un repositorio como String
git2String(RepoInput, RepoAsString):-
    convert_To_String(RepoInput, "", RepoAsString).
```

Figura 16: Código función git2String.

2.4. Función Extra

2.4.1. GitPull

Función que retorna una lista con todos los cambios (Commits) desde el Remote repository al Local repository y además pone los archivos correspondientes en el WorkSpace

```
| gitPull(RepoInput, SALIDA)
```

Figura 17: Función gitPull.

```
% ############################################################
% Funcion que mueve el contenido de Remote Repository a Local Repository
% ademas coloca los archivos correspondientes en el Work Space
gitPull(RepoInput, RepoOutput):-
    get_Remote_Repository(RepoInput, RR),
    set_Local_Repository(RepoInput, RR, RepoOutputAux),
    set_Work_Space(RepoOutputAux, RR, RepoOutpurAux2),
    set_Remote_Repository(RepoOutpurAux2, [], RepoOutput).
```

Figura 18: Código función git2String.

3. Exposición de Resultados

Para demostrar funcionamiento de cada función se probara un ejemplo de los puestos en el código y se presentara su resultado.

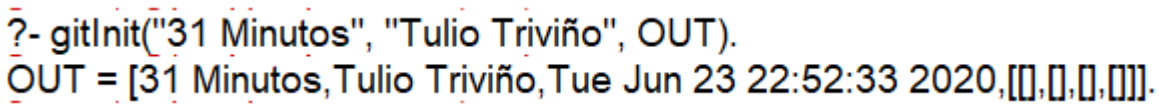

Figura 19: Ejemplo función gitInit.

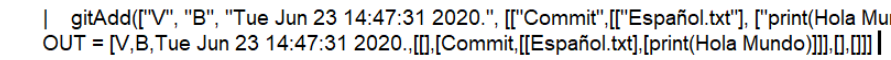

Figura 20: Ejemplo función gitAdd.

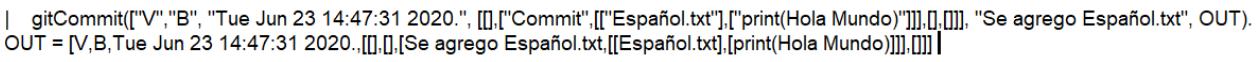

Figura 21: Ejemplo función gitCommit.

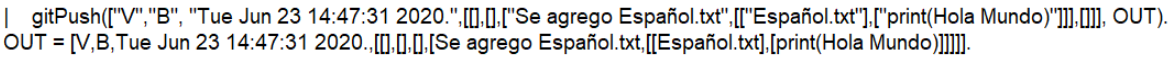

Figura 22: Ejemplo función gitPush.

```
|    git2String(["Lab","Juan", "Mon Jun 23 14:47:31 2020.",[["No Commit",[["WorkSpace.txt","contenido"]]]
No Commit",[["Index.txt","contenido"]]],["Commit Local",[["LocalRepo.txt","contenido"]]],["Commit Remote
"RemoteRepo.txt","contenido"]]]]],OUT).
OUT = ###### Repositorio 'Lab' ######
Fecha de creación: Mon Jun 23 14:47:31 2020.
Autor: Juan
Rama actual: Master
Archivos en WorkSpace:
      WorkSpace.txt

Archivos en Index:
      Index.txt

Commits en Local Repository:
      'Commit Local'
      LocalRepo.txt

Commits en Remote Repository
      'Commit Remote'
      RemoteRepo.txt
##### FIN DE REPRESANTACION DEL REPOSITORIO COMO STRING ##### .
```

Figura 23: Ejemplo función git2String.

```
|    gitPull(["V","B", "Tue Jun 23 14:47:31 2020.",[[],[],[],["Commit Remote",[["d.txt","contenido"]]]], OUT).
OUT = [V,B,Tue Jun 23 14:47:31 2020.,[[Commit Remote,[[d.txt,contenido]]],[],[Commit Remote,[[d.txt,contenido]]],[]]]
```

Figura 24: Ejemplo función gitPull.

4. Análisis de resultados

Como se puede ver en los resultados expuestos se pudieron cumplir los objetivos generales y específicos, logrando crear los tres TDA planteados y sus relaciones, además de las funciones que permiten hace uso óptimo de estos.

Aún así existen mejoras que se pueden aplicar a este modelo, una de estas es el hecho de poder mantener un estado de versiones (Como originalmente lo hace Git) añadiendo una segunda o tercera lista de Objetos al TDA Zona llevando así las versiones correspondientes, además de llevar registro de la fecha en que se hacen cambios en cada zona (algún edit, add, commit, push o pull).

5. Conclusión

Según los resultados obtenidos se puede decir que se logro el cien por ciento de lo que se esperaba en un principio, además de que el plan inicial funcionó debido a la experiencia adquirida en el laboratorio anterior donde hubo que realizar cambios a mitad de camino para satisfacer ciertos requerimientos.

Los objetivos específicos se completaron y fueron una guía para llevar a cabo el laboratorio, se realizaron los TDA en orden de importancia para ir desde lo macro del proyecto a lo micro.

El conocimiento de lo que es y como funciona una TDA fueron parte importante del desarrollo, ya que antes de tratar de hacer cualquier función se hicieron los getters y setters correspondientes (y las funciones necesarias para esto) lo que hizo que las funciones obligatorias (y la función extra) fueran relativamente fácil de programar, ya que haciendo uso de los métodos de cada TDA se simplifica la tarea.