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

Laboratorio 1, Paradigma Funcional

Integrantes: Aldo Castillo

Curso: Paradigmas de Programación

Profesor: Roberto González

30 de Mayo de 2020

Tabla de contenidos

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 todos 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 primera instancia, abordaremos el problema con el Paradigma Funcional con el cual deberemos implementar ciertas funciones que se detallaran más adelante. El Paradigma Funcional es un paradigma declarativo basado en el uso de funciones matemáticas. Además este paradigma depende exclusivamente de los valores de entrada de cada función y no de factores externos, por lo que si los argumentos son los mismos, la salida siempre será igual.

Usando la sintaxis de Scheme con el lenguaje R6RS rigiéndose por el Paradigma Funcional, se deberá solucionar la problemática de representación de las zonas de trabajo, representación de archivos y las funciones para comunicar estos.

1.3. Solución propuesta

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

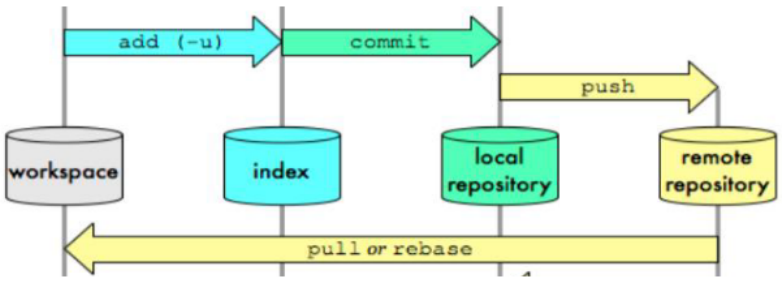

Figura 1: Abstracción de Git.

1. TDA Zonas
2. TDA Archivos
3. 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 Zonas, TDA Archivos 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. Realizar funciones 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 primera 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, además de llevar un registro del puntaje obtenido.

1.4.4. Metodologías y herramientas

Para la realización de este laboratorio se utilizaron los apuntes de clase, sumado a las librerías de Dr.Racket 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 Dr.Racket. 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 Zonas, TDA Archivos y TDA Objeto.

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

```
> ("Nombre.txt" "Contenido")
```

Figura 2: Representación TDA Archivo.

```
> ("Commit" (list archivo1 archivo2...))
```

Figura 3: Representación TDA Objeto.

```
> ((WorkSpace) (Index) (LocalRepository) (RemoteRepository))
```

Figura 4: Representación TDA Zonas.

2.1.2. Pares

Tal como lo dice su nombre, un par agrupa dos elementos de la forma:

[Elemento1, Elemento2]

Los pares tienen como...

1. Dominio: Elementos primitivos de Scheme (solo 2).
2. Recorrido: Pares.

Los pares son TDA's pares, por lo que tienen funciones tales como:

1. Función Constructora (Figura 5).

```
> (cons 1 2)
{1 . 2}
```

Figura 5: Constructor TDA Par.

2. Función Pertenencia (Figura 6).

```
> (pair? 1)
#f
> (pair? (cons 1 2))
#t
```

Figura 6: Pertenencia TDA Par.

3. Funciones Selectoras (Figura 7.)

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

Figura 7: Selectores TDA Par.

2.1.3. 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.

Las listas son TDA's Listas, por lo que tienen funciones tales como:

1. Función Constructora (Figura 8).

```
> (list 1 2 3)
{1 2 3}
> '(1 2 3)
{1 2 3}
```

Figura 8: Constructor TDA Listas.

2. Función Pertenencia (Figura 9).

```
> (list? (list 1 2 3))
#t
> (null? (list))
#t
```

Figura 9: Pertenencia TDA Listas.

3. Funciones Selectoras (Figura 10.)

```
> (car (list 1 2 3))
1
> (cdr (list 1 2 3))
{2 3}
```

Figura 10: Selectores TDA Listas.

4. Otras Funciones (Figura 11.)

```
> (length (list (list 1) (list 2 3)))
2
> (append (list 1) (list 2 3 4))
'(1 2 3 4)
> (reverse (list 1 2 3 4))
'(4 3 2 1)
```

Figura 11: Otras funciones TDA Listas.

2.1.4. 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:

2.1.4.1 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 (Figura 12).

```
> (define (factorialLineal n)
    (if (= n 0)
        1
        (* n (factorialLineal (- n 1)))
        )
    )
```

Figura 12: Recursión Lineal.

2.1.4.2 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 (Figura 13).

```
> (define (factorialDeCola n acum)
  (if (= n 0)
      acum
      (factorialDeCola (- n 1)(* acum n))
      )
  )
```

Figura 13: Recursión de cola.

2.1.4.3 Recursión Arbórea

La recursión Arbórea es aquella que se llama recursivamente 2 o más veces haciendo una operación entre estas llamadas (Figura 14)

```
> (define (fibonacciArborea n)
    (cond ((= n 0)0)
          ((= n 1) 1)
          (else (+ (fibonacciArborea (- n 1))(fibonacciArborea (- n 2))))
          )
  )
```

Figura 14: Recursión Arbórea.

2.2. Desarrollo de la solución

2.2.1. TDA Zonas

La representación del TDA Zonas sera por medio de una matriz que almacenara las cuatro partes de Git (Work Space, Index, Local Repository, Remote Repository) (Figura 15).

Con ayuda de la función CreateZonas se generara una zona vacía, además se presentaron tres ejemplos de zona previamente definidos (zona1, zona2 y zona3).

```
> zona1
'((("commit" ("archivo1.txt" "Hola Mundo")))
  (("Español" ("archivo1.txt" "Hola Mundo")) ("Ingles" ("archivo2.txt" "Hello World")))
  ((("Aleman" ("archivo3.txt" "Hallo Welt"))))
  ())
```

Figura 15: Ejemplo TDA Zonas.

2.2.2. 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.

```
> archivo1
'("archivo1.txt" "Hola Mundo")
```

Figura 16: Ejemplo TDA Archivo.

2.2.3. 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.

```
> ws1
'("commit" ("archivo1.txt" "Hola Mundo"))
```

Figura 17: Ejemplo TDA Objeto WorkSpace.

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

2.3.0.1 Add

Esta función añade los cambios locales del WorkSpace al Index.

```
;#############################################################################################
; Funcion Add

(define (add)
  (lambda (archivos)
    (lambda (zonas)
      (CompararArchivos archivos zonas)
      )
    )
)

;(((add) (list "Prueba1.txt")) emptyZone)
;(((add) (list "archivo2.txt")) zona2)
;(((add) (list "archivo1.txt")) zona1)

;#############################################################################################
```

Figura 18: Función add.

2.3.0.2 Commit

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

```
;#############################################################################################
; Funcion Commit

(define (commit)
  (lambda (comentario)
    (lambda (zonas)
      (setLocalRepo comentario zonas)
    )
  )
)

;(((commit) "Comentario1") emptyZone)
;(((commit) "Comentario2") zona1)
;(((commit) "Comentario3") zona2)

;#############################################################################################
```

Figura 19: Función commit.

2.3.0.3 Push

Función que envía los commit desde Local a Remote repository.

```
;#############################################################################################
; Funcion Push

(define (push)
  (lambda (zonas)
    (setRemoteRepo zonas)
  )
)


;((push) emptyZone)
;((push) zona1)
;((push) zona2)

;#############################################################################################
```

Figura 20: Función push.

2.3.0.4 Pull

Función que retorna una lista con todos los cambios (Commits) desde el Remote repository al Work Space.

```
;#############################################################################################
; Funcion Pull

(define (pull)
  (lambda (zonas)
    (setWorkSpace zonas)
  )
)


;((pull) emptyZone)
;((pull) ((push) zona1))
;((pull) ((push) zona2))

;#############################################################################################
```

Figura 21: Función pull.

2.3.0.5 Git

Función principal, esta se llama cada vez que queremos usar una de las funciones anteriormente descritas del como ((git *función*) zonas)

```
;#############################################################################################
; Funcion Git

(define (git funcion)
  (lambda (zonas)
    (lambda (x)
      (cond
        ((equal? funcion "add") (((add) x) zonas))
        ((equal? funcion "commit") (((commit) x) zonas))
        ((equal? funcion "push") ((push) zonas))
        ((equal? funcion "pull") ((pull) zonas))
        )
      )
    )
)


;#############################################################################################
```

Figura 22: Función git.

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 (Se presentara el estado de "zona1.ªntes de aplicar una función).

```
> zona1
'((("commit" ("archivo1.txt" "Hola Mundo")))
  (("Español" ("archivo1.txt" "Hola Mundo"))
   ("Ingles" ("archivo2.txt" "Hello World")))
  ((("Aleman" ("archivo3.txt" "Hallo Welt"))))
  ())
```

Figura 23: TDA Zonas, zona 1.

```
> (((git "push") zona1) "")
'((("commit" ("archivo1.txt" "Hola Mundo")))
  (("Español" ("archivo1.txt" "Hola Mundo"))
   ("Ingles" ("archivo2.txt" "Hello World")))
  ()
  ((("Aleman" ("archivo3.txt" "Hallo Welt")))))
```

Figura 24: Ejemplo función git.

```
> (((add) (list "archivo1.txt")) zona1)
'(()
  (("Español" ("archivo1.txt" "Hola Mundo"))
   ("Ingles" ("archivo2.txt" "Hello World"))
   ("Español" ("archivo1.txt" "Hola Mundo")))
  ((("Aleman" ("archivo3.txt" "Hallo Welt"))))
  ())
```

Figura 25: Ejemplo función add.

```
> (((commit) "Comentario2") zona1)
'((("commit" ("archivo1.txt" "Hola Mundo")))
  ()
  ((("Aleman" ("archivo3.txt" "Hallo Welt")))
    "Comentario2"
    ("Español" ("archivo1.txt" "Hola Mundo"))
    ("Ingles" ("archivo2.txt" "Hello World"))))
  ())
```

Figura 26: Ejemplo función commit.

```
> ((push) zona1)
'((("commit" ("archivo1.txt" "Hola Mundo")))
  (("Español" ("archivo1.txt" "Hola Mundo"))
   ("Ingles" ("archivo2.txt" "Hello World")))
  ()
  ((("Aleman" ("archivo3.txt" "Hallo Welt")))))
```

Figura 27: Ejemplo función push.

```
> ((pull) ((push) zona1))
'((("commit" ("archivo1.txt" "Hola Mundo"))
   (("Aleman" ("archivo3.txt" "Hallo Welt"))))
  (("Español" ("archivo1.txt" "Hola Mundo"))
   ("Ingles" ("archivo2.txt" "Hello World")))
  ()
  ())
```

Figura 28: Ejemplo función pull.

4. Análisis de resultados

Como se puede ver en los resultados expuestos las funciones push y pull funcionan correctamente, mueven la información necesaria al sector que le corresponde.

Uno de los errores presentes, como se puede ver en el resultado de la función add agrega el .ªrchivo1.txt"desde Work Space a Index pero ese se copia rescatando su çommit", esto se debe a que al hacer la comparación de .^{ex}iste este archivo en el workspace?.^{es}te compara el contenido del archivo y lo cambia según sea necesario, pero al retornar este resultado, se une al commit correspondiente y se agrega nuevamente a la lista de Index.

Otro error visto, es que al aplicar la función Commit, este guarda los archivos de Index en Remote Repository agregando un commit extra que encapsula el TDA Index quedando con un formato erróneo.

Como adicional, la función git no se acomoda directamente a lo pedido en el enunciado, esto debido a que ciertas funciones piden dos entradas (por ejemplo add) y otras una entrada (por ejemplo push) lo que hizo que git necesitara de dos entradas (currificadas) aun si se quiere llamar la función push, esta como segundo parametro puede recibir cualquier dato valido ya que no sera ocupado.

Una posible mejora de estos errores es rehacer los getters y setters correspondientes y aplicarlos en mas casos de las funciones principales para que casos como el de la función commit no ocurran.

5. Conclusión

Según los resultados obtenidos se puede decir que no se logró el cien por ciento de lo que se esperaba en un principio, esto se puede deber a varios factores, el tiempo y formato de clases en donde se dificulto el entendimiento de preguntas y respuestas jugaron en contra.

Se cumplieron la mayoría de los objetivos específicos expuestos, además de esto, en el camino se noto la necesidad de un segundo y tercer TDA, es cuando se creo el TDA Objeto y TDA Archivo haciendo el manejo de los elementos más fácil.

En el caso posible de un comodín para este laboratorio se empleará un la propuesta de mejora antes expuesta además de ajustar ciertos parámetros según el feedback correspondiente.

Para realizar este laboratorio no se uso ninguna pagina o documentación, todo se rescato de antiguos códigos propios.