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

Paradigmas de programación.

Laboratorio 1: Paradigma Funcional.

Nombre: Jordan Nicolas Godoy Rojas

Rut: 18.633.313-6

Profesor: Daniel Gacitúa

Sección: C-3

TABLA DE CONTENIDOS

TABLA DE ILUSTRACIONES

INTRODUCCIÓN

En este informe se describirá el problema planteado en el laboratorio 1 de la asignatura Paradigmas de programación, el cual corresponde a la realización de una simulación de un software de sistema de control de versiones de código fuente, en este caso Git, el cual será implementado en el lenguaje de programación Scheme usando el entorno de desarrollo DrRacket. Además, en este informe también se mostrará una solución a dicho problema, mencionando los métodos de resolución utilizados y las limitaciones de esta implementación.

DESCRIPCIÓN DEL PROBLEMA

Como se mencionó anteriormente, la implementación de la solución al problema fue desarrollada en el lenguaje de programación Scheme, el cual corresponde a un lenguaje del paradigma funcional, paradigma que ha sido estudiado en clases. La problemática gira en torno a la implementación de un programa que simula las funcionalidades y comportamientos de un sistema de versionamiento como lo es Git, el cual es un software donde se puede llevar un control y rastreo de los cambios de código fuente en proyectos de software.

Git tiene tres elementos básicos: commit, zonas de trabajo y comandos a ejecutar, siendo el commit la unidad de trabajo básica de Git, ya que constituyen a una copia del estado actual de los archivos de código, y únicamente almacenando los cambios entre el commit actual y anterior. Las zonas de trabajo son las instancias encargadas de almacenar los commits de forma local o remota, existiendo cuatro de estas:

Workspace, Index, Local Repository y Remote Repository. Y finalmente los comandos son acciones que crean, modifican y transfieren los commits entre las zonas de trabajo. Los comandos principales de Git son 6: Pull, Add, Commit, Push, Status, Log.

METODOLOGÍAS Y HERRAMIENTAS UTILIZADAS

Paradigma: Son ejemplos, modelos o patrón que establecen formas o estilos de hacer las cosas.

Paradigma funcional: Tiene sus raíces en el cálculo lambda, debido a que usa una notación formal (en este caso, prefija) para expresar funciones matemáticas. Este paradigma ofrece versatilidad y simpleza a partir de sus principios de abstracción los que se basan en la concepción del mundo a través de funciones.

Recursión lineal/natural: Se caracteriza por su simpleza expresiva y por involucrar a la misma función una única vez como parte de la definición de la misma. Esta recursión deja estados pendientes.

Recursión de cola: Se caracteriza por reusar el espacio, es tan eficiente como la iteración en términos de memoria. No deja estados pendientes.

Currificación: Consiste en transformar una función que utiliza múltiples argumentos en una secuencia de funciones que utilizan un único argumento.

Función de orden superior: Se caracteriza por tomar una o más funciones como entrada y devolver una función como salida.

Tipo de dato abstracto (TDA): Se refiere a una especificación de un tipo de dato en un alto nivel de abstracción. Corresponde a aquello que puede hacerse con el elemento que pertenece a un determinado tipo de dato, pero sin especificar su implementación (el cómo).

Scheme: Lenguaje basado en el paradigma funcional, es impuro debido a que sus estructuras de datos no son inmutables y el mecanismo principal de control de flujo son las recursiones.

ANÁLISIS DEL PROBLEMA

Para realizar la implementación de este programa se trabajará en el lenguaje Scheme y por tanto solo se utilizarán las herramientas de la versión R6RS de este.

Los requerimientos solicitados para la realización de este proyecto son en base a la estructura de tipo de datos abstractos (TDA), la cual consta de 6 niveles de abstracción y solo se utilizarán los niveles que se crean necesarios para el correcto funcionamiento del programa.

La función git es la que nos permite aplicar los comandos (en este caso serán funciones) sobre las zonas de trabajo. Esta función tiene que recibir como parámetro de entrada una función y dejar un registro historio de los comandos expresados (funciones que se utilizaron) en el orden en que fueron aplicados. Esta función tiene que ser realizada con funciones de orden superior.

Por otra parte, la función pull es una función que recibe como parámetro de entrada las zonas de trabajo y retorna una lista con todos los cambios desde el Remote Repository al Workspace, estos cambios que trae la función pull se verán reflejados en la zona de trabajo, debido a que cambiará la lista. Se debe emplear recursión natural o de cola, siendo distinto el estilo de recursión de la función add.

La función add, es la que añade los cambios desde el Workspace al Index, para esto se debe ingresar como entrada una lista de strings o una lista vacía. En caso de que sea una lista de strings, estos deben encontrarse en el Workspace para poder añadir los cambios al índex, en caso contrario no se añadirá ningún archivo, ya que no existen en el Workspace. Por otro lado, si se ingresa como entrada una lista vacía o nula, la función add deberá retornar las zonas de trabajo sin ningún cambio. Se debe emplear recursión natural o de cola, siendo distinto el estilo de recursión de la función pull.

La función commit recibe como entrada un string y la zona de trabajo, el string se utilizará como el mensaje descriptivo que acompañará al commit (archivo en este caso) cuando se le lleva del Index al Local Repository. Los cambios realizados se reflejarán en una nueva lista de zonas de trabajos.

La función Push es la que envía los commit desde el Local Repository al Remote Repository. Los cambios realizados darán como resultado una nueva lista de zonas de trabajo, la cual ya no tendrá commits en el Local Repository.

Y finalmente la función zonas->string, recibe como entrada las zonas de trabajo y retorna una representación de estas mismas, pero como un string de tal manera que puede ser visualizada de forma comprensible para el usuario. Esta función debe utilizar saltos de línea (‘\n’) y no debe utilizarse las funciones write o display dentro de la función.

DISEÑO DE LA SOLUCIÓN

La implementación de la solución para el problema planteado se realizó mediante el uso del lenguaje de programación Scheme y de los conocimientos sobre este en el curso de Paradigmas de programación, además de lo investigado de forma individual.

Para resolver el problema inicial y lograr implementar la simulación del software de control de versiones Git, se dividió el problema en diversos sub-problemas más simples de abordar e implementar, de esta manera fue más sencillo implementar las funcionalidades requeridas para el correcto funcionamiento del programa.

Se parte el programa con la definición del TDA zonas, el cual como representación tiene una lista de listas, siendo cada lista una zona distinta (ej. WorkSpace, Index, LocalRepository y Remote Repository). Para esto se construye una función que crea la lista de nombre zonas.

```
(define (zonas WorkSpace Index LocalRepository RemoteRepository)
                (list WorkSpace Index LocalRepository RemoteRepository))
```

Ilustración 1, función que construye las zonas.

Luego de crear la lista se procede a ver la pertenencia de cada elemento de la lista, para comprobar si son o no listas.

```
(define (zonas? zonas)
  (if (list? zonas)
      #t
      #f))
```

Ilustración 2, función de pertenencia de la lista zonas.

Después se aplican los selectores para obtener cada elemento de la zona de trabajo y para esto se obtienen de la lista principal, en este caso zonas.

Finalmente quedan los modificadores, los cuales serán del manejo de listas, los cuales servirán para agregar elementos, eliminar elementos, comparar elementos, concatenar listas, etc.

Ahora en el programa principal, se realiza el TDA de cada función partiendo por la función git, la cual su constructor es una función currificada la que puede recibir hasta tres parámetros de entrada. Esta estará representada de la siguiente manera:

```
(define git (lambda (comando) (lambda (a) (comando a))(lambda (b) (comando b))))
```

Ilustración 3, función git.

Luego viene la función pull la cual tiene como dominio la lista zonas, en caso de que esta sea nula retorna la lista nula, si no comprueba que el Remote Repository sea nulo, en caso

de serlo se devuelve el resto de zonas incluido el Remote Repository nulo, y si no, a través de recursión de cola se agrega el elemento o los elementos del Remote Repository al WorkSpace. Esto genera una nueva lista de zonas, la cual mantendrá el valor del Remote Repository, pero también lo habrá copiado en el WorkSpace.

```
(define (pull zonas)
```

Ilustración 4, función zonas.

Luego la función add tiene como dominio una lista de strings y la lista zonas, para esto se usan dos funciones auxiliares que sirven para recorrer y comparar completamente dos listas. Esto se realiza con el fin de encontrar los elementos que coincidan con los que se encuentran en el WorkSpace, debido a que la función add recibe como parámetro una lista de elementos de los cuales se agregaran al Index solo los que estén en el WorkSpace. Esta función se realiza mediante recursión natural/lineal, y posee una función de pertenencia que comprueba que el parámetro de entrada archivos sea una lista (de strings o vacía) para que sea una entrada valida.

```
(define add (lambda (archivos)(lambda (zonas)
```

Ilustración 5, función add.

Después tenemos la función commit, la cual recibe como entrada un string y la lista zonas, esta posee una función de pertenencia que comprueba que la entrada sea un string. Dentro de la función se comprueba que el Index no sea nulo, ya que no se puede hacer un commit de algo nulo, en caso de ser nulo termina la función, caso contrario hace una encapsulación de tres funciones de manejo de listas para agregar el elemento del Index con un mensaje descriptivo al Local Repository.

```
(define commit (lambda (mensaje) (lambda (zonas)
```

Ilustración 6, función commit.

La función Push, al igual que la función pull recibe como entrada solo la lista zonas, y pasa todo el contenido del Local Repository al Remote Repository, siempre y cuando exista un archivo para traspasar, en caso contrario solo devuelve la lista como estaba. En caso de que exista un archivo en el Remote Repository quedan ambos en una lista.

```
(define (push zonas)
```

Ilustración 7, función Push.

Y finalmente la función zonas->string que recibe como parámetro de entrado a la lista zonas, esta función cuenta con una función auxiliar, la que recorre cada elemento de cada una de las zonas de trabajos y cuando termina de recorrerlos agrega la etiqueta de la zona y da un salto de línea. Esto es gracias a que como entrada tiene el largo de la lista de zonas y cuando se termina de recorrer una lista el largo disminuye indicándole a la función auxiliar

que cambio de repositorio. Esta función busca entregar una representación de las zonas de trabajo en strings que sea fácil de visualizar para el usuario.

```
(define (zonas->string zonas)
```

Ilustración 8, función zonas->string.

ASPECTOS DE LA IMPLEMENTACIÓN

Como se mencionó anteriormente, la solución fue programada en el lenguaje Scheme, el cual corresponde a un lenguaje de paradigma funcional y el entorno de desarrollo ocupado fue DrRacket versión 7.6. Se utilizó la librería estándar de Scheme, todas las funciones nativas ocupadas en el programa son provistas por ese estándar.

El programa consta de dos archivos el primero llamado main_18633313_GodoyRojas.rkt en el cual se encuentran todas las funciones obligatorias y el segundo zonas_18633313_GodoyRojas.rkt, donde se encuentra el TDA de zonas junto con el manejo de listas, juntando ambos se puede usar correctamente el programa.

INSTRUCCIONES DE USO

El programa funciona en base a funciones especificas y otras que permiten una mejor simulación de git. Las funciones principales son:

- Git: esta función tiene 2 formas de usarse
  1. Si el comando dentro de git solo tiene un argumente de entrada (el comando y la lista zonas) y su respectivo resultado:

```
> ((git pull)(zonas '() '() '() '("lab.rkt")))
'(("lab.rkt") () () ("lab.rkt"))
```

Ilustración 9, uso función git de un argumento.

  2. Si el comando dentro de git tiene dos argumentos de entrada (el comando, argumento y la lista zonas) y su respectivo resultado.

```
> (((git commit)"primera modificacion")(zonas
'("lab.rkt") '("lab.rkt" "pacman.rkt") '()
'("lab.rkt")))
'(("lab.rkt")
  ("lab.rkt" "pacman.rkt")
  ("lab.rkt" "pacman.rkt" . "primera modificacion")
  ("lab.rkt"))
```

Ilustración 10, uso función git de dos argumentos.

- Pull: esta función solo tiene una forma de utilizarse, obteniendo el siguiente resultado:

```
> (pull (zonas '() '() '() '("lab1.rkt")))
'(("lab1.rkt") () () ("lab1.rkt"))
```

Ilustración 11, uso función pull.

- Add: esta función requiere una lista de archivos y la lista zonas, obteniendo lo siguiente:

```
> ((add '("tetris.rkt" "sudoku.c"
"asteroid.rkt"))(zonas '("lab.rkt" "tetris.rkt"
"sudoku.c") '("lab2.rkt") '("lab3.rkt") '("lab4.rkt")))
'(("lab.rkt" "tetris.rkt" "sudoku.c")
  (("tetris.rkt" "sudoku.c") "lab2.rkt")
  ("lab3.rkt")
  ("lab4.rkt"))
```

Ilustración 12, uso función add.

- Commit: se requiere un string que será el commit y la lista zonas, obtenido:

```
> ((commit "modificacion")(zonas  '("lab1.rkt")
'("lab2.rkt") '() '("lab4.rkt")))
'(("lab1.rkt")
  ("lab2.rkt")
  ("lab2.rkt" . "modificacion")
  ("lab4.rkt"))
```

Ilustración 13, uso función commit.

- Push: esta función requiere solo la lista zonas.

```
> (push (zonas '("lab1.rkt") '("lab2.rkt")
'("lab3.rkt") '()))
'(("lab1.rkt") () () (() ("lab3.rkt")))
```

Ilustración 14, uso función push.

- zonas->string: para usar esta función se pasa la función como parámetro a la función display, obteniendo lo siguiente:

```
> (display (zonas->string (zonas '("lab1.rkt")
'("lab2.rkt") '("lab3.rkt") '("lab4.rkt"))))
(lab1.rkt   | WorkSpace
 lab2.rkt   | Index
 lab3.rkt   | Local Repository
 lab4.rkt   | Remote Repository
)
```

Ilustración 15, uso función zonas->string.

RESULTADOS Y AUTOEVALUACIÓN

Este laboratorio tenia los siguientes requerimientos funcionales:

- **TDAs**: Se realizaron los TDA para cada función y para definir las zonas de trabajo, ocupando cada función definida en estos para el funcionamiento del programa.
- **Git**: Se realizo correctamente el uso de git como función de currificación, pero no logro obtenerse el completo requerimiento de este, debido a que no guarda un registro histórico de los comandos utilizados por el usuario. En las pruebas realizadas todas fueron exitosas al usar los comandos de este.
- **Pull**: Se realizo completamente el requerimiento de esta función, siendo capaz de traer el archivo del Remote Repository siempre que exista y se junta con la lista de archivos del WorkSpace. Se probo también usando un archivo nulo, y este no modifica la lista de zonas.
- **Add**: Esta función se probo con un archivo que existe tanto en los archivos y WorkSpace moviéndolo exitosamente. También se probó con un archivo nulo, y este deja la lista sin modificarla, lo cual indica su buen funcionamiento. Y por ultimo se probo con una lista de archivos que no es válida (no es lista), dando como resultado un mensaje que alerta al usuario que el archivo no es válido.
- **Commit**: Se realizo completamente la función commit, probándose con el Index nulo y agregando un commit, el cual no se agregaba porque no tenia archivo al cual realizar el commit, advirtiendo al usuario de esto. También se probó con una entrada distinta de string, el cual advertía al usuario de que la entrada no era válida. Y finalmente se probó con todos los datos válidos, dando como resultado una nueva lista valida con el debido commit realizado.
- **Push**: Esta función se probo de dos maneras, con un elemento existente en el Remote Repository que al mover el otro del Local Repository provoca que la lista se une, lo que esta bien. Y también se probó dándole un archivo del Local Repository nulo, dejando la lista zonas sin cambios.
- **Zonas->string**: Por último, esta función solo se probó dándolo como entrada a la función display generando correctamente la representación de todas las zonas.

Para finalizar se concluye que los resultados fueron buenos en todas las funciones, en el caso del git solo falto agregar el registro histórico, pero funciona correctamente en todo lo demás.

CONCLUSIÓN

Para el desarrollo de git, fue importante incorporar los conceptos aprendidos sobre el paradigma funcional, ya que de esta manera se logro dividir mejor los problemas en otros más pequeños, y por tanto más abordables, especialmente para el paradigma de programación funcional donde cada sentencia debe ser expresada como una función.

El manejo de funciones, la currificación y recursión fueron esenciales para implementar las funcionalidades de git, ya que la recursión es el único control de flujo que se tiene en el lenguaje, y por tanto se volvió una obligación el uso de este. El conocer y dominar mas esta técnica simplifico la programación de la mayoría de las funcionalidades del programa.

El paradigma funcional, con el cual se trabajo en este laboratorio, genero una gran dificultad, ya que uno se ve en la obligación de aprender el manejo de los distintos tipos de recursiones con el fin de lograr un programa funcional completo, aun así, se logro completar gran parte del proyecto satisfactoriamente, y con esto comprender algunos de los conceptos de la programación funcional. Dicho esto, se logro implementar todas las funciones obligatorias del laboratorio.

REFERENCIAS

Pagina web del manual de DrRacket:

- https://docs.racket-lang.org/reference/

Enunciado para el proyecto de paradigmas de programación:

- https://docs.google.com/document/d/1WF8EDXxDPcdJhV0UrQY94HV9_NLKCsGMHHgu_NSApGk/edit