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

INFORME LABORATORIO 1 – PARADIGMA FUNCIONAL

PARADIGMAS DE PROGRAMACIÓN

Alumno:	**LUIS HENRIQUEZ T.**
Profesores:	Roberto González
Fecha de Entrega:	01-06-2020

Santiago de Chile

1 – 2020

Tabla de contenido

CAPÍTULO 1. INTRODUCCIÓN

1. DESCRIPCIÓN DEL PROBLEMA.

Para este semestre se ha definido hacer una simulación de un software de sistema de control de versiones de código fuente, tomando como ejemplo la implementación de Git. Para la primera entrega se ha determinado realizar la simulación siguiendo las normas del paradigma funcional utilizando el lenguaje basado en listas Racket, en el programa Dr.Racket 6.11.

Este proyecto considera la implementación de 3 elementos básicos de Git que son el **commit**, las **zonas de trabajo** y los **comandos** a ejecutar de Git.

2. DESCRIPCIÓN DEL PARADIGMA.

La programación funcional es un paradigma de programación declarativa cuyos orígenes provienen del cálculo lambda, este está basado en el uso de funciones aritméticas en el cual los datos son inmutables, enfatizando la aplicación de funciones a diferencia de la programación imperativa que enfatiza las variables de estado.

La característica fundamental del paradigma funcional es que no existe la asignación ni el cambio de estado en un programa, las ”variables” son representadas por funciones que no cambian en toda la evaluación del programa, solo son expresiones matemáticas que devuelven nuevo valores a partir de los declarados, es decir construir funciones a partir de las ya existentes. Por lo tanto, es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente.

Otras características del paradigma funcional son las siguientes:

- Recursión
- Funciones como tipos de datos primitivos
- Uso de listas

3. ANÁLISIS DEL PROBLEMA

Dentro del enunciando del laboratorio se presentaron requerimientos Funcionales y no funcionales que se deberán tomar en cuenta para el desarrollo de este.

Entre los requerimientos que afectan directamente a la implementación del programa pueden separarse entre requerimientos obligatorios y requerimientos extra.

Dentro de los obligatorios se encuentran las siguientes funciones:

1.3.1 TDAs: Implementar abstracciones apropiadas para el problema, teniendo en cuenta que se asumirá una estructura de repositorios lineal, en donde cada TDA implementado deberá ser contenido en un archivo independiente el cual contenga sus representaciones, constructores, funciones de pertenecía, selectores, modificadores u otras funciones.

1.3.2 Git: Función que permite aplicar los comandos en el contexto de Git sobre las zonas de trabajo.

1.3.3 Pull: Función que retorna una lista con todos los cambios desde el RemoteRepository al Workspace registrados en la zona de trabajo, siendo el resultado de la función una nueva versión de la zona de trabajo. Ejemplo de uso: ((git pull) zonas)); zonas corresponde a una definición previa de función constante que tiene las cuatro zonas de acuerdo con la representación escogida para el TDA zonas.

ejemplo de uso: ((git pull) zonas))

1.3.4 Add: Función que añade los cambios locales registrados en el Workspace al Index en las zonas de trabajo.

ejemplos de uso: (((git add) (list “file1.rkt” “file2.rkt”)) zonas)

1.3.5 Commit: Función que genera un commit con los cambios almacenados en index especificando un mensaje descriptivo para llevarlos al LocalRepository.

ejemplos de uso: (((git commit) “miCommit”) zonas)

1.3.6 Push: Función que envía los commit desde el LocalRepository local al RemoteRepository registrado en las zonas de trabajo. ejemplo de uso: ((git push) zonas))

1.3.7 Zonas->string: Función que recibe las zonas de trabajo y entrega una representación de estas como un string posible de visualizar de forma comprensible al usuario.
Ejemplo de uso: (zonas->string zonas)

Los requerimientos extras solo serán considerados si la evaluación de los requerimientos obligatorios supera la nota 4.0 y están conformados por la implementación de las siguientes funciones:

- Status: Función que retorna un string con la información del ambiente de trabajo:
  - Archivos agregados al Index
  - Cantidad de commits en el Local Repository
  - La rama actual en la que se encuentra el Local Repository (predeterminado: “master”)
- Log: Función que muestra los últimos 5 commits sobre el repositorio/rama actual, mostrando el mensaje asociado a cada commit.
- Branch: Función que permite crear una nueva “rama” (flujo alternativo de los commits).
- Checkout: Función que permite cambiar de la rama actual a una rama específica por su nombre (string).
- Merge: Función que permite tomar los cambios de otra rama (especificada por su nombre como un string) y sobreescribirlos en la rama actual.
- Add-all: Función que permite agregar todos los archivos del Workspace al Index.

- diff: Función que permite comparar y listar las diferencias entre la rama origen y la rama destino.

CAPÍTULO 2. DISEÑO DE LA SOLUCIÓN

Para el desarrollo de este laboratorio se utilizo el enfoque del paradigma funcional implementado en el lenguaje de programación Racket utilizando el programa Dr.Racket en su versión 6.11.

La estrategia que se decidió utilizar fue la descomposición del el problema en funciones pequeñas con un único objetivo, por lo que la simulación del software de sistema de control de versiones de código fuente Git fue descompuesta en sus componentes básicos: Git, Add, Commit, Push, Pull y el TDA zonas para la simulación del espacio de trabajo que contiene: Workspace, Index, Local Repository, Remote Repository.

La función Git recibe como argumento el nombre del comando a ejecutar: Add, Commit, Pull o Push, esto permite entregar los argumentos de cada función de forma currificada y evitar errores por las diferencias de aridad entre las funciones.

La función add recibe como argumento una lista de archivos y la zona de trabajo, esta revisa que los elementos entregados en la lista pertenezcan a los elementos contenidos en el Workspace y los copia en el Index de la zona de trabajo, retornando una nueva zona de trabajo con el Index actualizado; en caso de que uno o más elementos ingresados no pertenezcan a la lista, se entrega un mensaje informando a usuario.

La función commit recibe como argumento un mensaje descriptivo (string) y la zona de trabajo, esta función recupera los archivos del Index, concatena un mensaje descriptivo al final de la lista de archivo y los mueve al Local Repository de la zona de trabajo, dejando el Index vacío.

La función push recibe como argumento la zona de trabajo, esta toma los archivos almacenados en el Local Repository y los envía al Remote Repository eliminando los duplicados.

La función pull tiene como argumento la zona de trabajo, recupera los archivos del Remote Repository y los trae al Workspace, eliminando duplicados.

CAPÍTULO 3. ASPECTOS DE IMPLEMENTACIÓN

La estructura del proyecto, la carpeta que contiene al proyecto contiene un archivo por cada función implementada, siendo el archivo “git.rkt” el archivo desde el cual se pueden llamar todas las funciones implementadas, aun así, es posible probar las funciones desde sus archivos con los ejemplos escritos en forma de comentarios al final del código de cada función, para esto se hizo uso de una herramienta de Dr.Racket (*provide* - *require*) que permite utilizar funciones desde otro archivo que las requiera, siempre y cuando los archivos se encuentren en la misma carpeta.

Cabe señalar que se hizo uso de la biblioteca string de racket para la implementación de la funcion zonas->string, solo el lenguaje base de Racket.

CAPÍTULO 4. EJEMPLOS Y RESULTADOS OBTENIDOS

Ejemplos de retornos para función git:

```
;Ejemplos funcion Git
;(((git add) (list "archivo1.c"))zonas)
;((git pull)zonas)
;(((git commit)"agrega elementos del index a localRepository")zonas)
```

Los cuales retornan respectivamente:

```
'(("workspace.rkt" "archivo1.c") ("I_archivo2.c" "index.rkt" "archivo1.c") ("localR.rkt" "L_archivo3.c")
("R_archivo4.c" "remote.rkt"))
'(("workspace.rkt" "archivo1.c" "R_archivo4.c" "remote.rkt") ("I_archivo2.c" "index.rkt") ("localR.rkt"
"L_archivo3.c") ("R_archivo4.c" "remote.rkt"))
'(("workspace.rkt" "archivo1.c") () ("localR.rkt" "L_archivo3.c" "I_archivo2.c" "index.rkt" "agrega elementos del
index a localRepository") ("R_archivo4.c" "remote.rkt"))
```

En caso de ingresar una función cuyo nombre no esta definido, el programa falla, así como también si no se respeta la aridad de la función llamada.

```
> ((add (list "archivo1.c")))
...enriquez/add.rkt:8:30: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 0
```

CAPÍTULO 5. EVALUACIÓN FUNCIONES

FUNCIÓN	ESCALA (0 - 1)
GIT	0.75
PULL	0.75
ADD	1
COMMIT	0.75
PUSH	0.75
ZONAS->STRING	1
STATUS	0
LOG	0
CHECKOUT	0
MERGE	0
ADD-ALL	0
DIFF	0
TDA ZONAS	0.75

CAPÍTULO 6. CONCLUSIONES

La programación funcional, responde a la pregunta ¿Qué? mientras que la programación imperativa responde a la pregunta ¿Cómo?
Al responder el "¿Qué?", nos enfocamos en el resultado y no en el procedimiento. Esto implica un nivel mayor de abstracción, lo que a su vez nos lleva a que el código sea más corto y comprensible que su equivalente en programación imperativa, la utilización de funciones que sirven a un solo propósito permite que la optimización sea mas simple pues no causan efectos secundarios.

Pero este paradigma no esta libre de desventajas, como lo son la dificultad para crear un buen código para un programador acostumbrado a la programación imperativa, generación de grandes cantidades de basura, principalmente por la inmutabilidad del paradigma.