UNIVERSIDAD SANTIAGO DE CHILE

Laboratorio de Paradigmas de Programación
Laboratorio Nº 1: ’’Simulación de un Sistema de Archivos en Scheme”

Integrantes:

Claudio Hernández Hernández

Profesor:

Edmund Leiva Lobos

Asignatura:

Paradigmas de Programación

1. Introducción

El presente reporte corresponde al primer laboratorio del curso de Paradigmas de Programación. En esta ocasión, abordaremos el Paradigma Funcional. utilizando el lenguaje de programación Scheme a través del compilador Dr.Racket.
En este informe, abordaremos un problema específico después de una breve descripción de este. A continuación, nos sumergiremos en la descripción del paradigma funcional y un conjunto de conceptos aplicados en este proyecto.
A medida que avanzamos, analizaremos en detalle el problema planteado y diseñaremos una solución acorde con el enfoque funcional. Posteriormente, compartiremos los aspectos más relevantes de la implementación del proyecto y proporcionaremos instrucciones y ejemplos para su uso.
Finalmente, mostraremos los resultados obtenidos y realizaremos una autoevaluación del trabajo realizado. Cerraremos este reporte con una conclusión que destaque los aprendizajes y las conclusiones obtenidas a través del desarrollo de este laboratorio.

2. Descripción del Problema

Se requiere crear una simulación de una plataforma que funcione como un sistema de archivos, como, por ejemplo, el Explorador de archivos de Windows. En esta plataforma, se permite la creación de unidades de almacenamiento, y seguido de ello el registro de usuarios, los cuales al identificarse e iniciar sesión correctamente tendrá la capacidad de crear tanto archivos como carpetas, editar el contenido realizado y compartirlo con otros usuarios que también están registrados en el sistema.

Para lograr esto, se deben tener en consideración algunos aspectos fundamentales:

1. Unidades lógicas: Corresponden a partes de almacenamiento que permiten acceder, gestionar y organizar de forma independiente. Cada unidad tiene una letra, un nombre y su capacidad. Estos valores permiten identificar a que unidad nos referimos y la cantidad de espacio que tengamos disponible para su uso.

2. Usuarios: Son a los perfiles que han sido ingresados en el sistema. Estos contienen un nombre de usuario que permite identificarlos. Al momento de iniciar sesión con uno de estos usuarios se le permite el uso de las operaciones disponibles en la plataforma.

3. Archivos: Ficheros creados por el usuario. Estos contienen el nombre del usuario que lo haya creado, fecha tanto de creación como de la última modificación realizada, sus atributos de seguridad y el contenido principal de dicho documento. Estos pueden ser manipulados a través de las operaciones disponibles.

4. Operaciones: Consideradas funciones que permiten realizar distintas acciones en el sistema, como por ejemplo el registro de usuarios, la creación de archivos, remover directorios, entre otros.

3. Descripción Paradigma

El paradigma funcional se centra en el uso de funciones como la principal unidad de abstracción y programación. Una función es una transformación que toma elementos de entrada(dominio) y los procesa para obtener un resultado(recorrido).
Este paradigma se diferencia de otros en que se enfoca en "qué"se quiere obtener en lugar de çómo"se llega a ese resultado, lo que implica un mayor nivel de abstracción. A diferencia de otros paradigmas, como el imperativo, no se utilizan variables actualizables.
El lenguaje de programación funcional Scheme se utiliza en este trabajo, aunque no es un lenguaje puramente funcional ya que también permite aspectos de programación imperativa.
El paradigma funcional se basa en el Cálculo Lambda, una notación formal para el estudio de funciones. En lugar de enfocarse en los pasos detallados de un proceso, se concentra las transformaciones que se aplican a los datos para obtener el resultado deseado

El paradigma funcional presenta elementos clave para la construcción de código utilizando este enfoque:

1. Funciones anónimas: Estas funciones, derivadas del Cálculo Lambda se expresan sin nombre y se definen con una entrada y una transformación que se aplica directamente.

2. Composición de funciones: Es una operación donde doos funciones se combinan para generar una tercera función, lo que resulta útil cuando se tienen funciones que se complementen entre sí.

3. Recursividad: Juega un papel fundamental en este paradigma, ya que permite resolver problemas mediante la aplicación de soluciones más pequeñas del mismo problema.

4. Funciones de orden superior: Son funciones que pueden recibir o devolver otras funciones como argumentos. Un ejemplo es la composición de funciones.

5. Currificación: Es la transformación de una función de "n.ªrgumentos en una secuencia de funciones de un solo argumento. Se utilizan funciones anónimas como parte de este concepto de funciones de orden superior.

4. Análisis del Problema

Como primer componente fundamental se tiene el sistema, el cual contiene a todos los demas elementos, como usuarios, documentos, directorios, entre otros. Esta plataforma es responsable de registrar y gestionar todos los cambios realizados de manera exitosa a través de operaciones específicas. Es el pilar fundamental que sostiene y organiza todos los datos y acciones que ocurren dentro del sistema.
Considerando esto, se requiere la implementación del sistema en Scheme, el cual puede ser representado de la siguiente manera:

Sistema: Lista que se compone de un string, una fecha, una lista con sublistas que representan las unidades lógicas, una lista con los usuarios registrados, un string con el usuario logeado, una lista con el directorio que se encuentra al momento de realizar las operaciones, una lista con sublistas que representan a las carpetas, una lista con sublistas que representan a los archivos, y por ultimo una lista que representa a la papelera de reciclaje.(string x list x list x list x string x list x list x list x list).

También se tienen más elementos importantes en el sistema:

- Unidades lógicas: Corresponden a las unidades de almacenamiento creadas antes del registro de usuarios. Contiene una lista para cada unidad lógica. Estas sublistas almacenan una letra que representa su unidad, un nombre y su capacidad de almacenamiento ((char x string x int) x (char x string x int), ...).

- Usuarios registrados: Usuarios que han realizado un registro en el sistema. Contienen el nombre de todos los usuarios que se han registrado (string x string x ...).

- Usuario logeado: Pertenece al usuario que ha iniciado sesión en el sistema. Solo contiene al usuario que se ha logeado (string).

- Directorio actual: Como su nombre dice, corresponde a una lista con la posición actual del usuario en el sistema. Cada acceso está representado por un string dentro de la lista (string x string x ...).

- Carpetas: Directorios que permiten almacenar archivos u otras carpetas dentro de ellas. Cada carpeta es representada con una sublista dentro de la lista. Estas contienen el nombre de la carpeta, dirección en la que se almacena, usuario que la crea, fecha de creación y de última modificación, y su atributo de seguridad ((string x string x string x list x list x string) x (string x string x string x list x list x string) x ...).

- Archivos: Ficheros almacenados dentro de la unidad lógica o de carpetas. Estos al igual que las carpetas también son representadas por sublistas en una lista. Estas sublistas almacenan el nombre del archivo, el tipo de archivo, su contenido, directorio en el que ha sido creado, usuario que la crea, fecha de creación y de última modificación, y los permisos de seguridad ((string x string x string x lista x string x lista x lista x string) (string x string x string x lista x string x lista x lista x string).

- Papelera: Como su nombre dice, representa una papelera de reciclaje. Aquí se almacenan tanto archivos como carpetas que hayan sido eliminadas, para así poder restaurarlas si se desea. Esta contiene sublistas que representan a los elementos eliminados (list x list x list x ...).

Por último, se tienen las funciones que se pueden realizar en la plataforma:

- system: Construye el sistema.

- run: Ejecuta los comandos sobre el sistema.

- add-drive: Añade una unidad con letra única al sistema.

- register: Registra un usuario único en el sistema.

- login: Inicia sesión con un usuario existente en el sistema.

- logout: Cierra sesión del usuario
- switch-drive: Fija la unidad en donde el usuario realizara acciones. Esta solo funciona si hay un usuario logeado.
- make directory (md): Crea una carpeta con nombre especifico dentro de una unidad.
- change directory (cd): Permite cambiar la ruta en la que se realizaran operaciones.
- add-file: Añade un archivo al sistema en la ruta actual.
- delete (del): Elimina un archivo. El elemento queda en la papelera.
- remove-directory (rd): Elimina una carpeta que este vacía. El elemento queda en la papelera.
- copy: Copia un archivo en una ruta determinada.
- move: Mueve un archivo a una ruta determinada.
- rename (ren): Renombra un archivo o carpeta si el nuevo nombre es único en la ruta.
- directory (dir): Lista el contenido de un directorio.
- format: Formatea una unidad, eliminando todo su contenido.
- encrypt: Encripta un archivo o una carpeta con su contenido.
- decrypt: Desencripta un archivo o una carpeta con su contenido.
- plus-one: Transforma un string a código ascii sumando 1 a cada carácter.
- minus-one: Transforma un string a código ascii restando 1 a cada carácter.
- grep: Buscar dentro del contenido de un archivo o de una ruta.
- view-trash: Ver el contenido de la papelera.
- restore: Restaurar contenido especifico de la papelera.

5. Diseño de la Solución

Al momento de diseñar la solución, además de utilizar datos nativos de Scheme, también se deben implementar datos específicos. Estos datos son creados haciendo uso de los llamados TDA, permitiendo la construcción de las operaciones de otros TDAs y para las operaciones mencionadas en el análisis.
El TDA más relevante en este proyecto es:

5.1. TDA SYSTEM

- Representado por una lista que contiene: un string y una fecha.
- Este TDA es construido con un nombre, y con ello se le asigna la fecha de registro.

Para la modificación y realización de los demás elementos que componen a la plataforma, se tienen las funciones disponibles en el sistema:

5.2. OPERACIONES FUNCIONALES

- run: Función de orden superior que recibe otra operación y currifica los elementos necesarios para su funcionamiento. Dependiendo de la operación realizada, entregara los resultados realizados por dicha operación.

- add-drive: Función que realiza 3 distintas iteraciones para comprobar todos los casos posibles. Si se le entrega un system sin argumentos, entrega un resultado falso. En caso de que la función cumpla con los elementos necesarios y sea una lista con menos de tres elementos, se agrega como una sublista la unidad con los elementos que la componen. El tercer caso revisa si los elementos cumplen, pero cambia dos cosas con respecto a la anterior iteración, ya que esta revisa si el largo del sistema es de más de dos elementos, y también realiza una recursión para verificar que la unidad que se desea agregar no esté ya dentro del sistema. Si cumple los requisitos, agrega la nueva unidad en la misma lista que están las demás unidades. Por último, si ninguna de las anteriores iteraciones no es verdadera, arroja un valor falso.

- register: Realiza 3 iteraciones. Primero verifica si el sistema no es una lista nula. Si es nula, entrega un resultado falso. Si no es nula, se verifican los valores a registrar y se revisa si el largo del sistema es menor a 4. Si cumple los requisitos, se agrega en el sistema. En caso de que no cumpla, se realizan las mismas condiciones, pero cambiando que el largo debe ser mayor a 3, y que el usuario no debe estar ya registrado. Al cumplir esto, se añade el nuevo usuario en la lista de usuarios. Si ninguna de las condiciones se cumple, se entrega un resultado falso.

- login: También se realizan 3 iteraciones. Primero se verifica si el sistema es nulo. Si es verdadero, se arroja un resultado falso. Si es falso, se verifican los valores del login, si el usuario a logear existe, y si el largo del sistema es menor a 5. Si es verdadero, se agrega al sistema el usuario logeado. Si es falso, se realizan las mismas verificaciones de antes, solo que el largo del sistema debe ser mayor a 4, y también que no existan un usuario logeado. Si es verdadero, significa que el usuario a logear es el mismo que ya está logeado, por lo que se devuelve el mismo sistema. Si es falso, se arroja un resultado falso.

- logout: Solo tiene una iteración. Se verifican si hay un usuario logeado, y si el largo del sistema es mayor a 4. Si es verdadero, se deslogea del sistema quitando el elemento que representaba que el usuario estaba logeado. Si es falso, arroja un resultado falso, ya que no había ningún usuario logeado.

- switch-drive: Realiza 2 iteraciones. Se verifican los valores del cambio de unidad, si hay un usuario logeado, si la unidad a la que se desea cambiar existe, y que el largo del sistema sea menor o igual a 6. Si es verdadero, se le agrega al sistema la unidad en la que se realizaran las acciones. Si es falso, se vuelve a verificar los elementos, pero ahora el largo del sistema debe ser mayor a 6. Si es verdadero, al sistema se le cambia la dirección en la que estaba anteriormente, por la nueva unidad.

- make directory (md): Se compone de 3 iteraciones. Se verifican los valores del directorio, y el largo del sistema debe ser menor a 7. Si es verdadero, Se agrega el directorio al sistema en una sublista. Si es falso, se vuelven a verificar los valores, pero el largo del sistema debe ser igual a 7, y también revisa si el directorio ya existe en esa dirección. Si es verdadero, se agrega el directorio en la misma lista que se guardan los demás directorios. Si es falso, se verifican los mismos elementos, pero el largo del sistema debe ser mayor a 7. Si es verdadero, se agrega el directorio en la misma lista que los demás directorios.

- change directory (cd): Esta función tiene múltiples iteraciones. Empieza verificando si el sistema es nulo, entregando falso si el resultado es verdadero. En caso contrario, depende del valor que tenga la ruta a la que se desea ir. Si la ruta es "..regresa a la carpeta del nivel anterior en la ruta actual. Si la ruta es /regresa a la raíz de la unidad. Si la ruta es un nombre de una carpeta, se actualiza el directorio, añadiendo el nombre de la carpeta en la lista. Si alguno de estos elementos no se cumple, arroja un resultado falso.

- add-file: Al igual que el anterior, esta función también tiene varias iteraciones. Verifica si el sistema el nulo, entregando falso en caso de que si lo sea. Si no es nulo, se verifican los elementos y se revisa si el sistema no tiene archivos incluidos, entregando el sistema con el nuevo archivo en caso de que sea verdadero. Si es falso, Se compara el archivo a añadir, con los archivos que ya posee el sistema, y el largo del sistema debe ser 8. Si el archivo es nuevo y el sistema cumple con su largo, se añade al sistema junto a los demás archivos. Si es falso, se realizan las mismas comprobaciones, pero con un largo del sistema mayor a 8. Si es verdadero, se añade a la lista de archivos. Si es falso, se arroja un resultado falso.

- delete (del): Verifica si el sistema es nulo, entregando falso en caso de que sea así. Verifica si el largo del sistema es 8, entregando falso si es el caso, ya que no tiene archivos para eliminar. Verifica si el largo del sistema es 8 y si el archivo que se desea eliminar está dentro del sistema, enviando el archivo a la papelera en caso de que se cumpla. Verifica lo

mismo de la iteración anterior, pero con un largo del sistema mayor a 8, entregando lo mismo que el valor anterior, solo que añadiéndole los valores que la papelera ya contiene.

- remove-directory (rd): Verifica si el sistema es nulo, entregando falso en caso de que sea verdadero. Verifica si carpeta a eliminar existe, y si el directorio actual no está dentro de ese fichero, enviando la carpeta a la papelera en caso de que sea verdadero.
- copy: Verifica si el sistema es nulo, entregando falso en caso de que sea así. Verifica si el archivo que se desea copiar existe, y si el directorio al que se desea copiar también existe. Si se cumple esto, se verifica si en el directorio al que se desea copiar no existe un archivo con el mismo nombre, copiando el archivo a ese directorio en caso de que sea verdadero.
- move: Verifica si el sistema es nulo, entregando falso en caso de que sea así. Verifica si el archivo que se desea mover existe, y si el directorio al que se desea mover también existe. Si se cumple esto, se verifica si en el directorio al que se desea mover no existe un archivo con el mismo nombre, moviendo el archivo a ese directorio en caso de que sea verdadero.
- rename (ren): Verifica si el sistema es nulo, entregando falso en caso de que sea así. Verifica si el archivo que se desea renombrar existe, y si el directorio actual no contiene un archivo con ese nombre. Si se cumple esto, se renombra el archivo.
- directory (dir): Lista el contenido de un directorio.
- format: Verifica si el sistema es nulo, entregando falso en caso de que sea así. Verifica si la unidad que se desea formatear existe. Si se cumple esto, se eliminan todos los archivos y carpetas que contiene la unidad.
- encrypt:
- decrypt:
- plus-one:
- minus-one:
- grep:
- view-trash:
- restore:

6. Aspectos de Implementación

Este laboratorio utiliza el compilador Dr.Racket, más específicamente las versiones 6.11 o superiores a esta.
La implementación es principalmente con el uso de listas. Sin embargo, se considera el no uso de funciones tales como CAR o CDR en la elaboración de los TDAs, ya que estas pertenecen al TDA Lista, por lo que se pide encapsular estas funciones dentro de los TDAs creados. La estructura del Código es en archivos independientes. Debido a que la función run se implementó en cada uno de los archivos por separado, no se hizo uso de un archivo main que contiene a una importación de todos los demás ficheros. Debido a ello, para realizar el funcionamiento del Código, se debe realizar cada operación en su respectivo archivo. En total se tienen 24 archivos, los cuales siguen el siguiente nombre:

- "NombreTDA-rut-ApellidoPaterno-ApellidoMaterno.rkt"

6.1. AUXILIAR

Otra alternativa que se tuvo en consideración es no implementar la función run, y así crear el archivo main con todos los ficheros importados en este. Esto conllevaría a tener que llamar a las funciones directamente en este último.

7. Instrucciones de Uso

Primero se debe comprobar que se tienen todos los archivos en su computadora. Junto a esos archivos, hay un archivo independiente que contiene ejemplo que se pueden probar en los archivos. Debido a que algunos solicitan un tamaño para el sistema, es posible que algunos resultados arrojen el resultado "#f".

7.1. MODIFICACION FUNCION ADD-FILE

Debido al no lograr la correcta implementación de la funcion add-file, se optó por realizar un cambio al momento de llamarla, este cambio se puede visualizar en el siguiente ejemplo:

- (define S1((run S0 add-file)(file "foo1.txttxthello world 1")))->(define S1((run S0 add-file)'("foo1.txttxthello world 1")))

Como se puede observar, la función cambia al momento de ingresar los elementos que requiere para su funcionamiento, ya que antes se entregaba entre paréntesis la funcion file y tres strings. En cambio, para que la función haga un correcto funcionamiento se debe solo escribir una lista con los tres strings que requiere la operación.

7.2. Resultados Esperados

Se busca desarrollar una simulación de un sistema de archivos sin problemas ni ambigüedades. La implementación debe estar libre de errores y utilizar de manera adecuada las estructuras de listas y la recursión, que son elementos fundamentales para el desarrollo del proyecto.
Cada funcion debe realizar su tarea correctamente y los Tipos de Datos Abstractos (TDAs) deben tener representaciones adecuadas y bien definidas. En otras palabras, se espera un programa robusto, eficiente y bien organizado que permita un sistema de archivos fluido y sin errores.

8. Resultado y Autoevaluación

Los resultados obtenidos fueron los esperados, ya que se crearon las funciones obligatorias y una gran parte de las operaciones funcionales.
Los errores que se pueden esperar son debido a las funciones que no se lograron realizar completamente, debido al ajustado tiempo restante para la entrega del comodín.
Se realizaron múltiples pruebas con distinto ejemplos para comprobar que las funciones no arrojen algún error en la ejecución del Código. Se lograron completar exitosamente 13 de las 24 funciones.

8.1. Autoevaluación

Es una lista que se realiza de la siguiente manera:

- 0: No realizado
- 0.25: funciona 25 % de las veces
- 0.5: funciona 50 % de las veces
- 0.75: funciona 75 % de las veces
- 1: funciona 100 % de las veces

8.2. Requerimiento No Funcionales

1. Autoevaluacion:1
2. Lenguaje:1
3. Version:1
4. Standard:1
5. No variables:1
6. Documentacion:0.75
7. Dom->Rec:1
8. Organizacion:1
9. Historial:1
10. Script de pruebas:0.5
11. Prerrequisitos:1

8.3. Requerimiento Funcionales

1. TDAs:1
2. system:1
3. run:1
4. add-drive:1
5. register:1
6. login:1
7. logout:1
8. switch-drive:1
9. md:1
10. cd:1
11. add-file:1
12. del:1
13. rd:0.25
14. copy:0.25
15. move:0.25
16. ren:0.25
17. dir:0.25
18. format:0.25
19. encrypt:0.25
20. decrypt:0.25
21. plus-one:0.25
22. minus-one:0.25
23. grep:0.25
24. view-trash:0.25
25. restore:0.25

9. Conclusiones del Trabajo

Después de finalizar y concluir el proyecto, se puede afirmar que se lograron cumplir los objetivos principales de manera satisfactoria. Durante el proceso, se adquirió una mejor compresión del uso de Scheme y Racket, lo que permitió realizar un gran avance del laboratorio en comparación de la primera entrega. Además, se obtuvo un aprendizaje significativo sobre un nuevo paradigma de programación y se pudo aplicar de manera efectiva, aplicando los conceptos fundamentales aprendidos en clase, como el uso de TDAs, recursividad, funciones de orden superior, currificacion, entre otros.
La realización del proyecto presento un desafío significativo debido a la naturaleza de las ideas implementadas. En ocasiones, algunas ideas no funcionaban como se esperaba, lo que requería la revisión y ajuste de enfoques. Lo normal en caso de que ocurriera uno de estos problemas, se optó por realizar cambios cuando era necesario para un correcto funcionamiento. Este enfoque de iterar y adaptarse fue esencial para lograr un proyecto exitoso y funcional.

Nota final esperada: 5.7