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

LABORATORIO I

Paradigma Funcional Aplicado En La Elaboración De Un Sistema Operativo de Archivos

Paradigmas de Programación - 13310
Profesor Roberto González Ibáñez, PhD.

Byron Caices Lima

22 de mayo, 2023

Tabla de Contenidos

1. INTRODUCCIÓN

En el presente informe se abordará el problema propuesto en el enunciado de laboratorio 1 acerca del paradigma funcional aplicado en el lenguaje de programación Racket/Scheme. Se revisará una descripción del problema y del paradigma empleado además del diseño de la solución y aspectos de la implementación.

2. DESCRIPCIÓN DEL PROBLEMA

Se presenta como desafío la creación de un sistema operativo de archivos (con orientación al usuario) en donde se tenga un sistema al cual se le pueden añadir unidades o drives y a estas unidades pueden añadirse carpetas y archivos implementando operaciones tales como añadir/formatear unidad, añadir/borrar/renombrar/encriptar carpeta u archivo. Para esto se debe tener en consideración la existencia de una papelera para permitir operaciones como la de restaurar archivo o carpeta. Finalmente se apunta a simular la consola del sistema introduciendo las operaciones típicas como cd, md, dir, etc.

3. DESCRIPCIÓN DEL PARADIGMA Y CONCEPTOS APLICADOS

El paradigma funcional se basa en la idea de construir un programa a través de la composición y aplicación de funciones. Este paradigma usa el cálculo lambda, el cual permite que las funciones pueden ser pasadas como argumentos a otras funciones y retornadas como valores de otras funciones. La recursividad es una técnica fundamental en la programación funcional. Existen varios tipos, entre ellos la recursividad natural, de cola y arbórea. La natural es la forma más básica de recursividad, donde una función se llama a sí misma para resolver una versión más pequeña del mismo problema dejando estados pendientes. La recursividad de cola es una forma especial de recursividad donde la llamada recursiva va construyendo la solución progresivamente sin dejar estados pendientes. La recursividad arbórea es una forma de recursividad donde una función se llama a sí misma múltiples veces, formando una estructura de árbol de llamadas a funciones. También, se emplea la currificación que es una técnica en la que una función con múltiples argumentos se transforma en una secuencia de funciones, cada una con un solo argumento. Esta técnica permite la aplicación parcial de funciones, lo que puede ser útil para crear nuevas funciones a partir de funciones existentes con algunos argumentos predefinidos.

4. ANÁLISIS DEL PROBLEMA

Dados los requisitos funcionales específicos que se deben cubrir lo primero que se debe realizar para abordar este problema es la identificación de los TDA necesarios. Notamos que se tienen 5 principales elementos para interactuar en un sistema de archivos: Sistema, Unidades, Carpetas, Archivos y Usuarios. (Especificación de TDAs en Anexo 1)

A su vez, este sistema está compuesto por atributos como un nombre, usuarios registrados, rutas accesibles (paths), unidades, contenido de las unidades y ese contenido posee carpetas y archivos, y las carpetas pueden poseer más carpetas y archivos dentro. Al final, se asemeja a la estructura de árbol del Anexo 2. Sin embargo, implementar una estructura arbórea para la construcción del TDA puede ser un poco engorroso por lo que nos queda pensar en otra alternativa usando la estructura de datos más básica que nos ofrece Racket el cual se basa en LISP, es decir, solo utilizar listas. Con esto se puede determinar que para manejar el sistema de archivos basta con trabajar con rutas; añadir una carpeta "Carpeta1" a una unidad "C:" no es más que agregar un path nuevo al sistema "C:/Carpeta1" y además agregar el TDA Carpeta al contenido de la unidad correspondiente para almacenar de alguna formas los TDAs que pueden ser trabajados en el Sistema. Sin embargo, no bastaría solo con agregar la ruta ya que nos faltaría saber, por ejemplo, la metadata de la carpeta o de un archivo como el usuario creador, fecha de creación, fecha de modificación y atributos de seguridad. También quedaría definir qué es borrar un archivo en mi sistema, y se llegó a la conclusión de que borrar una carpeta o archivo no es más que eliminar la ruta (location) de una carpeta o archivo del path del sistema para que así ese ítem eliminado se vuelva inaccesible e inmutable pero que de todas maneras siga estando almacenado en la unidad en que se encontraba antes de ser borrado, tiene lógica ya que un TDA Sistema por sí solo no podría tener la capacidad de almacenar archivos sino que estos son almacenados en una unidad. Luego, considerando que el paradigma funcional no nos permite hacer uso de variables tendremos Sistemas los cuales no pueden ser modificados directamente si no que se tendrá que reconstruir el sistema completo si es que queremos realizar una modificación (no podemos hacer uso de set, por ejemplo). Dados estos fundamentos del análisis del problema queda pasar al diseño de la solución.

5. Diseño De La Solución

Como se mencionó en el punto anterior el TDA Sistema se compone de atributos tales como la hora actual, usuarios registrados, unidades, papelera y paths. La forma en que se decidió representar el sistema corresponde a una lista con listas, exceptuando aquellos valores que por su naturaleza no es necesario que sean una sublista del sistema como, por ejemplo, el nombre de este, la fecha y la ruta actual del sistema, los que para el caso de mi implementación serán un string. Luego, ya las listas que componen al sistema principalmente serán la sublista de paths que contiene todas las rutas accesibles y mutables del sistema (una lista de strings), la sublista que contiene la papelera que contendrá aquellas rutas (strings) que se vuelven inaccesibles ya que su contenido fue eliminado y finalmente la sublista de unidades o drives tendrán cada una otra sublista del contenido del drive en donde se guardarán carpetas y archivos (a los que llamaremos items). Cada TDA Carpeta y Archivo tendrá el atributo location el cual se referirá a la ubicación del item + nombre del item. Con ese atributo location se podrá generar funciones filtros que nos permita crear funciones como copy la cual sea capaz de "buscar" en el contenido de un drive una unidad por tan solo su nombre y current-path por ejemplo: al ejecutar ((run System copy)"file1.txt" "D:/") si actualmente el current-path del sistema es "C:/carpeta1" y carpeta1 efectivamente contiene a file1.txt entonces se aplica una función filtro tal que retorne el contenido del drive filtrando por location, si location == "C:/carpeta1/file1.txt" de esta manera obtenemos o "copiamos" el file1.txt y luego quedaría cambiar su atributo location por el target-path quedando "D:/file1.txt" con eso se debe volver a reconstruir el contenido del drive agregando ese nuevo archivo y además el nuevo path debe ser agregado a la ruta de paths del sistema. Con la función move ocurriría el mismo proceso. Move es ejecutar un copy y posteriormente eliminar el archivo y path inicial y para eso se pueden usar nuevamente los filtros que permite Racket con la función filter y la capacidad de darle una función filtro como argumento gracias al calculo lambda.

Para la gran mayoría de las funciones, se utilizaron funciones de orden superior o composición de funciones, esto con el fin de generar soluciones a subproblemas y con la función principal a partir de dichas soluciones parciales, armar la solución final.

6. Aspectos De La Implementación

Cada función de los requerimientos funcionales se encuentra documentada con su respectivo dominio, recorrido y recursividad (si aplica). La solución fue desarrollada en el lenguaje de programación Racket/Scheme utilizando el interprete DrRacket 8.8 con las funciones propias del lenguaje, no se utilizaron bibliotecas, más que racket-date para acceder a la fecha y hora.

7. Instrucciones De Uso

Ejemplos del punto 1en Anexo 3

1) En cuanto a resultados esperados, para las funciones copy y move se copia o mueve un <<archivo>>, <<carpeta>> o una <<carpeta que contiene archivos>> sin problema alguno, por ejemplo, dado el current-path "C:/" que contiene la "carpeta1" la que a su vez contiene los archivos "file1.txt",…, "fileN.txt" moverá/copiará exitosamente la carpeta y su contenido a la target-path de ejemplo "D:/" agregando a los paths del sistema "D:/carpeta1/fileN.txt" N veces + "D:/carpeta1". Sin embargo, si se intenta copiar la carpeta "carpeta2" (con el mismo origen y mismo target-path que el ejemplo anterior) que contiene la "carpeta22" que a su vez contiene los archivos "doc1.txt",…"docN.txt" habría un error al mover/copiarla: obtendremos como resultado "D:/carpeta2/", "D:/carpeta22/" y los paths "D:/carpeta22/docN.txt" y el resultado esperado debería ser "D:/carpeta2/carpeta22/docN.txt". Esto se debe tener en cuenta al utilizar el programa.

2) A pesar de que no añadí funciones como string-downcase/upcase en el constructor de los tda's en el las funciones requeridas se agregó la función format-path y format-name las cuales se encargan de dar el formato correcto al path y nombres de ítems. También la función del funcionará solo con nombres de archivos.

8. Resultados Y Autoevaluación

El grado de alcance de los requerimientos funcionales fue conseguido casi en su totalidad para las funciones implementadas (16/25 RF). Se cumplieron los resultados esperados a la hora de ejecutar el script de pruebas del enunciado con una pequeña observación en el S43 (relacionado con la función del) y también las pruebas propias. Sin embargo, la función move/copy pueden presentar el problema mencionado anteriormente.

9. CONCLUSIÓN

Se lograron los objetivos respecto a desarrollar un sistema operativo de archivos simplificado a través del paradigma funcional aplicado en una implementación en lenguaje Racket/Scheme. Fue un gran desafío iniciar este proyecto debido a lo acostumbrado que se encuentra alguien al paradigma imperativo, se enfrentó una nueva forma de programar en donde no existían variables, ni ciclos for o while por ende se vio en la obligación de utilizar la recursividad, para conseguir las metas propuestas y cumplir con los requerimientos funcionales y no funcionales.

10. REFERENCIAS

Racket. (s. f.). Recuperado 26 de septiembre de 2022, de https://racket-lang.org

Navarro, S. Universidad San Jorge. (2021, 3 de Junio). *TIPS EN 2 MINUTOS - INFORMÁTICA: Sistema de archivos FAT. Recuperado de https://www.youtube.com/watch?v=bjUhkBBteRM*

Soporte Software. (2014, 1 de Abril). *Sistemas de Archivos.* Recuperado de https://www.youtube.com/watch?v=p2309jhCi9I

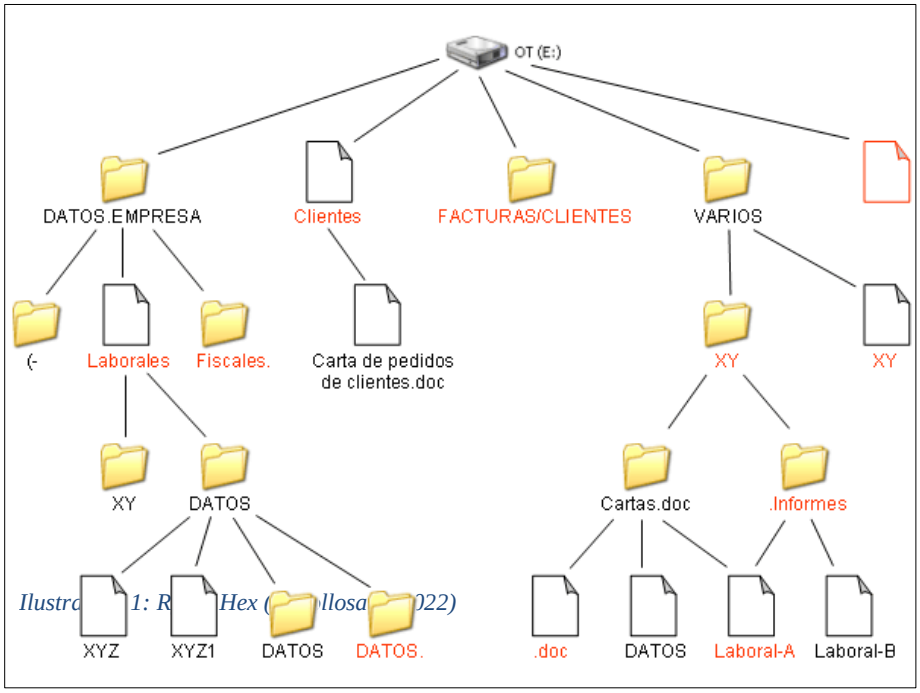
Aulapc (s.f). *Archivos, Carpetas, Directorios, Arboles y Pathnames.* Recuperado de http://www.aulapc.es/basico_archivos_carpetas.html

11. ANEXOS

Anexo 1: Definición de los TDA´s

	TDA System	TDA Drive	TDA Folder	TDA File	TDA User
Constructor	make-system	make-drive	make-folder	make-file	user
Selectores	get-system-name	get-letter	get-folder-name	get-file-name	
	get-loged-user	get-drive-name	get-create-date	get-extension	
	get-current-path	get-drive-cap	get-mod-date	get-text	
	get-users	get-drive-content	get-folder-location	get-file-security	
	get-system-date		get-folder-creator	get-file-password	
	get-drives		get-folder-size	get-create-date-file	
	get-trashcan		get-items	get-mod-date-file	
	get-paths		get-folder-security	get-file-location	
	get-current-drive		get-folder-pass	get-file-creator	
			folder-type	get-file-type	
Modificadores	run	ara la implementacion los modificadores de cada tda se crearon desde el sistema en los R			
	add-drive				
	register				
	login				
	logout				
	switch-drive				
	md				
	cd				
	add-file				
	del				
	rd				
	copy				
	move				
	ren				
	format				
	encrypt				
	decrypt				

Anexo 2: Estructura de árboles en un sistema de archivos



Ilustra 1: R Hex (llosa 022)

Anexo 3: Ejemplos sobre las instrucciones de uso

1) Sea (define S0 (system “newSystem”)) a S0 se le agregan unidades, directorios y archivos tal que resulta el sistema SN con los siguientes atributos:

current-path: “C:/”

system-paths:

‘(“C:/”
“C:/carpeta0”
“C:/carpeta0/carpeta1”
“C:/carpeta0/carpeta1/file1.txt”
…
“C:/carpeta0/carpeta1/fileN.txt”
“D:/”)

Si yo ejecuto ((run SN copy) “carpeta1” “d:/”) entonces se evalúa el siguiente resultado para system-paths:

‘(“C:/”
“C:/carpeta0”
“C:/carpeta0/carpeta1”
“C:/carpeta0/carpeta1/file1.txt”
…
“C:/carpeta0/carpeta1/fileN.txt”
“D:/”
“D:/carpeta0”
“D:/carpeta1”
“D:/carpeta1/file1.txt”
…
“D:/carpeta1/fileN.txt”)

Como vemos efectivamente el contenido se copia a la ruta de destino sin embargo, si carpeta-padre contiene carpetas-hijas entonces las hijas quedan en la ruta de destino. La razón de este error aún no es identificada, por lo que no ha sido arreglada. Lo correcto sería:

(…
“D:/carpeta0/carpeta1/file1.txt”
…
“D:/carpeta0/carpeta1/fileN.txt”)

Anexo 4:

Resumen resultados obtenidos (Listado con todos los requerimientos del proyecto)

#	Requerimientos Funcionales	Alcance(ptos)
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	0.5
13	rd	1
14	copy	0.75
15	move	0.75
16	ren	0
17	dir	0
18	format	1
19	encrypt	0
20	decrypt	0
21	plus-one	0
22	minus-one	0
23	grep	0
24	view-trash	0
25	restore	0
	Requerimientos No Funcionales:	
1	Autoevaluación	1
2	Lenguaje	1
3	Versión	1
4	Standard	1
5	No variables	1
6	Documentación	1