Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Informe laboratorio 1 (Funcional - Scheme)

Alumno: Cristóbal Torres Undurraga
Fecha: 22 de Mayo de 2023
Asignatura: Paradigmas de Programación

1. Introducción

El informe consiste en 11 partes: Introducción, Descripción del Problema, Descripción del Paradigma, Análisis del Problema, Diseño de la Solución, Aspectos de la Implementación, Instrucciones de Uso, Resultados, Conclusiones, Referencias y Anexos.

En el siguiente informe se explicará la implementación de un sistema operativo con el paradigma funcional, en específico, el lenguaje de programación Racket. El paradigma funcional corresponde a un tipo de programación que se basa en funciones.

2. Descripción del Problema

El problema consiste en la simulación de un sistema operativo (Unix-like) simplificado. Debe tener un sistema de archivos (file system), el cual contiene procesos, métodos y reglas. El sistema operativo debe ser capaz de realizar operaciones a través de comandos, los cuales crean, modifican o eliminan elementos del sistema de archivos. Debe tener 5 partes esenciales: sistema, drivers, usuarios, carpetas y archivos. El desafío está en cómo abstraerse del problema y poder organizar el sistema para que se implementen todas las partes y funcionen entre sí por medio de operaciones (register, login, md, add-file, switch-drive, etc)

3. Descripción del Paradigma

El paradigma que se utiliza para este informe es el paradigma Funcional, este consiste en la concepción del mundo a través de funciones. Forma parte del espectro de la programación declarativa, es decir, se enfoca en resolver un problema, el ***que***, no en ***cómo*** se resuelve. Se basa en el cálculo Lambda, que corresponde a un formalismo para la expresión de funciones, y este utiliza la notación *prefija* (operador procede al operando).

El paradigma funcional hace uso de distintas estrategias para la resolución de problemas. Entre estas existen: currificación, es el proceso de traducir la evaluación de una función a una secuencia de funciones de 1 argumento. Funciones de orden superior, debido a la mirada del paradigma funcional (todo son funciones) es permitido que las funciones tengan como dominio y recorrido a otras funciones, permitiendo una composición de estas. Y Evaluación Perezosa, corresponde a posponer la evaluación de una función hasta que sea necesario, evitando evaluaciones repetitivas. (González, 2023).

4. Análisis del Problema

system*:* La función crea un sistema, el cual se abordará mediante listas anidadas, para conseguir una mayor organización de las partes del sistema (datos del sistema, unidades, usuarios, papelera).

run: Funciona como el puente entre el sistema y los comandos, haciendo que sea posible ejecutarlos. Se utiliza currificación para su implementación.

add-drive*:* Añade una unidad (drive) al sistema. Para ello, en conjunto con la creación del TDA drive, se agrega la unidad en la correspondiente posición dentro del sistema. Cabe destacar que cada drive ya posee un carpeta raíz.

register: Agrega usuarios a la sección de usuarios dentro del sistema. Agrega el TDA user a la lista correspondiente

login: Selecciona e inicia sesión de un usuario ya registrado. Para esto, se filtra entre los usuarios dentro del sistema para verificar que exista. También se debe comprobar que no exista otro usuario con la sesión ya iniciada.
logout: Cierra la sesión que esté trabajando en el sistema. La implementación se realiza cambiando el usuario por el nombre predeterminado “N/A”, dentro de los datos del sistema.
switch-drive: Cambia el drive en que se realizan las operaciones, para esto, se verifica con un filtro que el drive exista. Posteriormente se reordenan los drives, dejando en primer lugar el drive objetivo, esto con el fin de agilizar el uso de otras funciones.
md: Función que crea un directorio en la ruta actual. Dentro de cada unidad existe una posición para las carpetas. Éstas no están anidadas entre sí, por lo que para buscar directorios solo es necesario hacer un filtro por los nombres.
cd: Función que cambia el directorio actual. Para esto, se reservan ciertos comandos específicos (“..”, “/”) y en cualquier otro caso se busca la ruta objetivo y se cambia la actual.
add-file: Función que agrega un archivo a la ruta actual. Para su implementación se verifica que el nombre no exista y después se realiza un apéndice en la carpeta correspondiente.
del: Función que elimina un archivo o carpeta con todo lo que contenga dentro de la ruta actual. Todo lo eliminado se envía a la papelera. Primero se reordenan los archivos o carpetas y se eliminan
rd: Función que remueve un directorio, siempre y cuando éste vacío. Para implementarlo, primero se comprueba mediante filtros que el directorio existe. Después se comprueba que no tenga archivos y subdirectorios. En el caso de los subdirectorios, se busca y comprueba la fuente de todas las carpetas para encontrarlos.
copy: Función que copia un archivo o carpeta a una ruta objetivo. Comprobando todos los datos de entrada.
move: Función que mueve un archivo o carpeta desde una ruta a otra. Para la implementación se fusiona copy y del.
ren: Función que modifica el nombre de una carpeta o archivo. Solo se selecciona el elemento y se modifica la ubicación en que se encuentra el nombre.
dir: Función que lista los contenidos de una ruta. Se implementa con recursión para formar un string de salida.
format: Función que formatea una unidad. En la implementación se reordenan las unidades para eliminar el contenido de la primera. Posteriormente se vuelve al orden original.

5. Diseño de la Solución

Para formar y operar en el sistema operativo, se optó por utilizar una representación de listas anidadas. El mayor problema a la hora de organizar el sistema es cómo manejar las carpetas y archivos. Para esto, se utilizó una lista que representa el sistema completo, está lista posee 4 sublistas. En 1era posición están los datos del sistema, es una lista que contiene la información más fundamental y recurrente, estos son el nombre del sistema, la fecha de creación, la unidad (drive) en la que se está trabajando, el usuario que realiza los cambios y la ruta en que se está trabajando. La 2da posición corresponde a las unidades del sistema, cada unidad posee datos propios y las carpetas dentro de esta. Las carpetas son listas que no están anidadas entre sí, en cambio, están desplegadas en la unidad, cada una con una ruta única y sus datos asociados. Dentro de las carpetas se encuentran los archivos como listas con sus datos asociados. En 3ra posición se encuentran los usuarios, estos poseen un nombre único y una fecha de creación. Finalmente, en 4ta posición se encuentra la papelera, dentro se encuentran los archivos y carpetas que fueron eliminados del sistema por medio de operaciones (del). Cada una tiene la ruta en que se encontraba antes de ser eliminado. [Diagrama 1]

Para realizar operaciones dentro del sistema se utilizan comandos, que pueden crear, modificar, seleccionar dentro del sistema. Además, todo comando tiene que cruzar por la función “run” de forma currificada. Para la creación de elementos (como add-drive, register, md, add-file) debe comprobarse que no se incumplan requisitos, como que el elemento ya exista, o ya exista en esa ruta. Para esto se hace uso de recursividad o funciones que filtren elementos. Después se crea el tipo de elemento por medio de constructores, generalmente listas, y se le añaden los datos pertinentes. Posteriormente se incluyen en la cola de su respectiva localización y se rearma el sistema. Para la modificación (login, logout, switch-drive, cd, del, rd, copy, move, ren, format) primero se cambia la posición de los elementos si es requerido (drives, carpetas, archivos) dejando siempre en primera posición el lugar en el que se va a realizar operaciones. Después se comprueba que se cumplan los requisitos de los comandos al igual que en la creación, con la diferencia de que en “copy” y “move” también se debe comprobar que no existan los elementos en la ruta objetivo. En el caso de “del”, se elimina el elemento de su posición original y se lleva a la papelera, añadiendo la ruta en que se encontraba en caso de ser un archivo. Algunas funciones deben aceptar tanto archivos como carpetas, por lo que se optó por diferenciarlo dependiendo del nombre. Las carpetas solo tienen un nombre sin nada en particular. En cambio, los archivos tienen un punto para diferenciar a qué tipo de archivo corresponde. Por la implementación del sistema de archivos, para comprobar si una carpeta tiene subcarpetas, se comparan las fuentes de estas, con un algoritmo recursivo, hasta que solo existan las subcarpetas. Una vez cumplidos los requisitos, se elimina el elemento original y se agrega su versión modificada, en caso de ser necesario. En el caso de que se deba seleccionar (dir), se forma un string de manera recursiva, utilizando el mismo algoritmo recursivo anterior para buscar y seleccionar subdirectorios.

6. Aspectos de la Implementación

Para la implementación del proyecto se estructuró en 6 TDA como archivos: system, drive, folder, file, user y fecha. Se utilizaron estos TDA ya que se pueden realizar todas las operaciones sobre ellas, con un alto grado de organización de las funciones. Cada TDA posee constructor, selectores, modificadores y otras operaciones, con sus respectivas documentaciones y explicaciones de cada constructor.

Solo se utilizó la librería *racket/base*, y en el proyecto se ocupó la versión 8.8 de DrRacket.

7. Instrucciones de uso

Para ejecutar funciones, es necesario tener un sistema sobre el cual trabajar, por medio de la función ***system*** [Figura 1] (en está prueba se utiliza ***display*** solamente para mostrar el sistema creado, no es un requerimiento para ejecutar funciones, a excepción de ***dir***). Ya con un sistema, para modificarlo de cualquier forma, por ejemplo, añadir una unidad (***add-drive***), tiene que pasar por la función ***run***. Para ello, se debe ejecutar aplicando currificación. [Figura 2] Otro ejemplo puede ser usar ***del*** [Figura 3] y [Figura 4]. La excepción a esta regla es la función ***dir***, es la única función que fue implementada que además de ***system***, que no requiere ser currificada con ***run***. Lo que requiere es usar la función ***display*** para mostrar el string resultante. Como ejemplo, se usará el input “/s”, que sirve para mostrar subdirectorios además del contenido del directorio actual. [Figura 5]

8. Resultados

Todos los requerimientos abordados lograron un alcance completo (a excepción de ciertas funcionalidades en del y dir) [Tabla 1]. Se realizaron pruebas en todas las funciones, las cuales se encuentran en el script de pruebas, en las que se comprueba que cumplan los objetivos, no sean case-sensitive, no realicen cambios si archivos, carpetas, usuarios, etc. no existen. Que no realicen cambios con elementos duplicados y que las funciones se ejecuten correctamente cuando se trabaja con más de 1 nivel de carpetas o con distintas unidades.

De las funciones que se implementaron, las que no se completaron fueron: **del**, solo se implementó la versión simple debido a que requiere mucho tiempo hacerla de forma completa. **dir**, no se implementó “/o [-]D” debido a la implementación del TDA Fecha. Todos los requerimientos posteriores a “format” no se implementaron por problemas de tiempo.

9. Conclusiones

A la hora de trabajar en el problema, las principales ventajas que posee el paradigma funcional provienen de su simpleza. Esta simplicidad aumenta su versatilidad. En lugar de enfocarse en definir tipos de datos, como vendría a ser con un paradigma procedural, el paradigma funcional se enfoca en trabajar con cualquier tipo de dato. (Stallman, 2022) La forma en que se representan las funciones permiten modificar de manera sencilla sus contenidos. La existencia de la evaluación perezosa ayuda inmensamente a ejecutar la recursión arbórea, ya que evita tener que ejecutar nuevamente un dato ya calculado.

En contraparte, la falta de variables y estructuras puede representar un desafío al simular un sistema operativo. Pese a ser una implementación más abreviada en algunas ocasiones, generalmente es una solución menos optimizada y más engorrosa. Por ejemplo, representar un árbol utilizando pares podría necesitar más recursos que su contraparte imperativa, además de no ser tan clara, esto debido a que se tiene que entrar en los pares anidados, agregar, modificar o eliminar el dato necesario, y finalmente rearmar los pares.

En conclusión, se logró implementar de manera adecuada el paradigma. El paradigma funcional es una herramienta poderosa gracias a su simpleza. Es sencillo de aprender y aplicar, teniendo resultados satisfactorios. La dificultad recae en adaptarse a un paradigma declarativo y abstraerse de los conocimientos anteriores respecto a la programación en general. Scheme (en este caso Racket) posee una de las documentaciones más completas y amigables hacia el usuario, permitiendo así una transición natural. Finalmente, el problema del laboratorio (sistema operativo / file system) se logró satisfactoriamente. Se consiguió implementar las funciones y abstracciones necesarias para su realización.

10. Referencia

Stallman, R. (2022). *How I do my Computing*. [Web personal] Extraído desde https://stallman.org/stallman-computing.html

González, R. (2023). *Paradigmas de programación. Programación Funcional*. [Diapositivas de PowerPoint]. Extraído desde Moodle Usach Paradigmas de Programación (13204 y 13310) 1-2023

11. Anexos

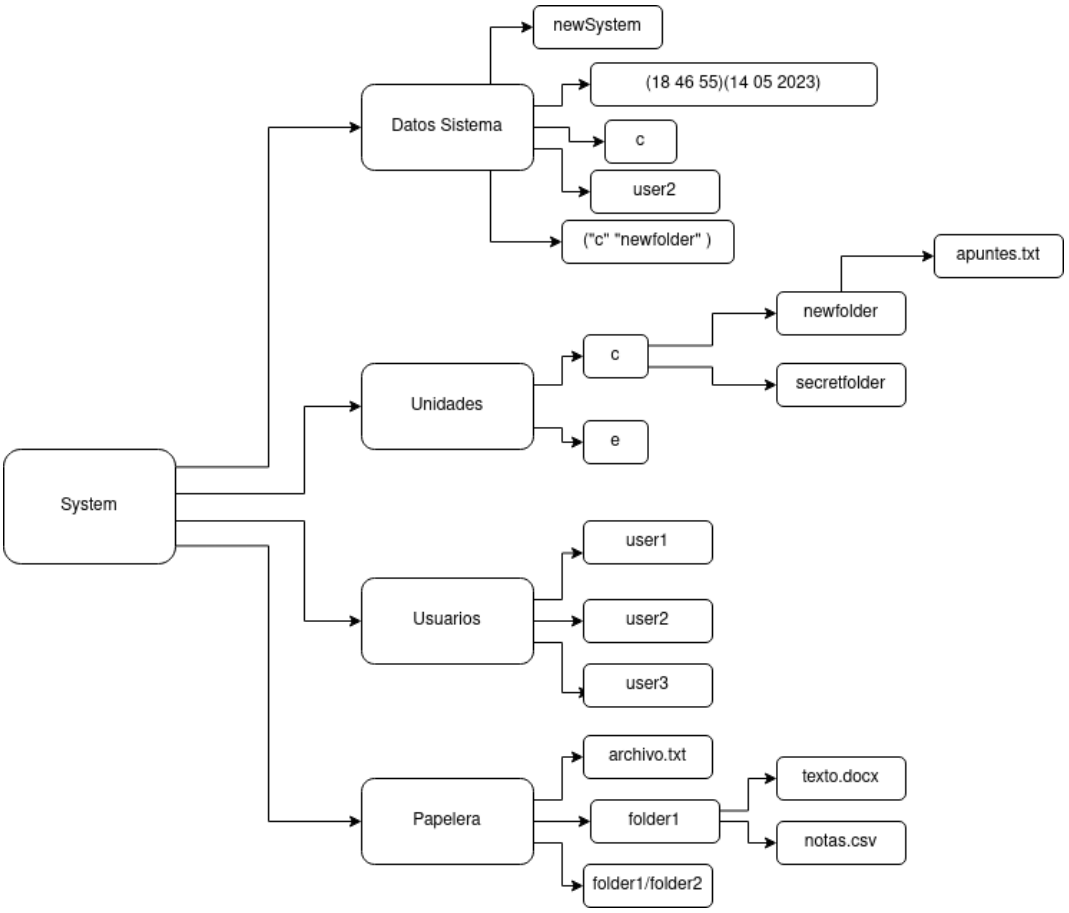

Diagrama 1, Representación del sistema.

```
> (display (system "Unix-like"))
((Unix-like ((23 27 5) (21 5 2023)) N/A N/A N/A) () () ())
```

Figura 1, Creación de un sistema con "system".

```
> S2
'(("newSystem" ((23 30 46) (21 5 2023)) "N/A" "N/A" "N/A")
  ((#\c "SO" 1000 (("c:") "N/A" ((23 30 46) (21 5 2023)) ((23 30 46) (21 5 2023)) ()))))
  ()
  ())
> ((run S2 add-drive) #\D "Util" 2000)
'(("newSystem" ((23 30 46) (21 5 2023)) "N/A" "N/A" "N/A")
  ((#\c "SO" 1000 (("c:") "N/A" ((23 30 46) (21 5 2023)) ((23 30 46) (21 5 2023)) ()))
   (#\d "Util" 2000 (("d:") "N/A" ((23 31 29) (21 5 2023)) ((23 31 29) (21 5 2023)) ())))
  ()
  ())
```

Figura 2, antes y después de agregar una unidad con “add-drive”.

```
> S35
'(("newSystem" ((23 34 14) (21 5 2023)) #\c "user2" ("c:" "folder1"))
  ((#\c
    "SO"
    1000
    (("c:" "folder1") "user2" ((23 34 14) (21 5 2023)) ((23 34 15) (21 5 2023)) ())))
     ("foo1.txt" "txt" "hello world 1" ())
     ("foo2.txt" "txt" "hello world 2" ())
     ("foo3.docx" "docx" "hello world 3" ())
     ("goo4.docx" "docx" "hello world 4" (#\h #\r)))
    ((("c:" "folder2" "folder21" "folder211")
      "user2"
      ((23 34 14) (21 5 2023))
      ((23 34 14) (21 5 2023))
      ()))))
    ((("c:" "folder2" "folder21")
      "user2"
      ((23 34 14) (21 5 2023))
      ((23 34 14) (21 5 2023))
      ()))))
    ((("c:" "folder2") "user2" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ()))))
    ((("c:" "folder3") "user2" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ()))))
    ((("c:") "N/A" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ())))
   (#\d "newD" 2000 (("d:") "user2" ((23 34 15) (21 5 2023)) ((23 34 15) (21 5 2023)) ())))
  (("user1" ((23 34 14) (21 5 2023))) ("user2" ((23 34 14) (21 5 2023))))
  ())
```

Figura 3, sistema original.

```
> ((run S35 del) "goo4.docx")
'(("newSystem" ((23 34 14) (21 5 2023)) #\c "user2" ("c:" "folder1"))
  ((#\c
    "SO"
    1000
    ((("c:" "folder1") "user2" ((23 34 14) (21 5 2023)) ((23 35 21) (21 5 2023)) ()))
     ("foo1.txt" "txt" "hello world 1" ())
     ("foo2.txt" "txt" "hello world 2" ())
     ("foo3.docx" "docx" "hello world 3" ()))
    ((("c:" "folder2" "folder21" "folder211")
      "user2"
      ((23 34 14) (21 5 2023))
      ((23 34 14) (21 5 2023))
      ()))))
    ((("c:" "folder2" "folder21")
      "user2"
      ((23 34 14) (21 5 2023))
      ((23 34 14) (21 5 2023))
      ()))))
    ((("c:" "folder2") "user2" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ()))))
    ((("c:" "folder3") "user2" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ()))))
    ((("c:") "N/A" ((23 34 14) (21 5 2023)) ((23 34 14) (21 5 2023)) ())))
   (#\d "newD" 2000 ((("d:") "user2" ((23 34 15) (21 5 2023)) ((23 34 15) (21 5 2023)) ())))))
  (("user1" ((23 34 14) (21 5 2023))) ("user2" ((23 34 14) (21 5 2023))))
  ((("c:" "folder1") ("goo4.docx" "docx" "hello world 4" (#\h #\r)))))
```

Figura 4, sistema después de eliminar un archivo con “del”.

```
> (display ((run S111 dir) "/s"))

folder1
carpeta_normal
newfoo1.txt
foo2.txt

./folder1:

./carpeta_normal:
```

Figura 5, demostración de la función “dir” con “/s”.

Función	Alcance Conseguido
TDA	Alcance Completo
system	Alcance Completo
run	Alcance Completo
add-drive	Alcance Completo
register	Alcance Completo

login	Alcance Completo
logout	Alcance Completo
switch-drive	Alcance Completo
md	Alcance Completo
cd	Alcance Completo
add-file	Alcance Completo
del	Se implementó una versión simple de la función
rd	Alcance Completo
copy	Alcance Completo
move	Alcance Completo
ren	Alcance Completo
dir	Alcance Completo exceptuando “/o [-]D”
format	Alcance Completo

Tabla 1, alcances de las funciones implementadas.