UNIVERSIDAD SANTIAGO DE CHILE

Laboratorio de Paradigmas de Programación
Laboratorio Nº 1: ''Lenguaje Scheme: Sistema de Archivos"

Integrantes:
Claudio Hernández Hernández

Profesor:

Edmund Leiva Lobos

1. Introducción

La programación funcional es un paradigma de programación que, a diferencia de la programación orientada a objetos, se busca que sea mas legible para el programador, y con ello intentar mantener un orden dentro de lo que sería el código o programa.

2. Descripción del Problema

En esta ocasión, como estudiantes nos ha tocado el labor de simular un sistema de archivos empleado en un sistema operativo, en donde se nos pedirá que se pueden realizar ciertas acciones como por ejemplo, el poder crear unidades de memoria con su respectivo nombre, capacidad y la letra que corresponde a su partición, el poder mover archivos entre carpetas almacenadas en estas unidades, etc.

3. Descripción Paradigma

Como ya se nombró en la introducción, este primer laboratorio se realizó con la programación funcional, usando el programación DrRacket, el cual nos permite dar uso del lenguaje de Scheme. Continuando con la programación funcional, al ser un lenguaje enfocado mas al uso de funciones, acompañado tambien de operaciones como la composicion y recursión, hace que sus valores dependan únicamente de los argumentos que recibe, y a su vez, estos argumentos no pueden cambiar sus valores, dejando asi como propiedad para los lenguajes funcionales el ser aplicativos puros.

4. Análisis del Problema

Antes de empezar a hablar de la solución a realizar, primero debemos realizar un breve análisis respecto a los requisitos a cumplir en este trabajo.

1. Tenemos los requisitos no funcionales, donde algunos deben ser obligatoriamente realizados para no optar por la nota mínima, tales como la autoevaluación, el uso de funciones standard,etc.

2. Tambien tenemos los requisitos funcionales, los cuales para recibir una evaluación deben cumplir con sus prerrequisitos de evaluación y de implementación, como por ejemplo la especificación de los TDA de manera correcta, donde tenemos que respetar tanto el trabajo que debe realizar las funciones, como su manera de implementación.

5. Diseño de la Solución

Antes de iniciar la codificación del laboratorio, primero se optó por hacer una descripción mas simple para cada función, intentando siempre mantener la estructura que se requería. También debido a esta descripción simplista, salieron ideas de que cosas se podían agregar al trabajo, y que no estuvieron establecidas en la realización de este, las cuales fueron dejadas aparte para no desviarse de lo pedido. Seguido de ello, se inició la sección de codificación, decidiendo empezar desde lo que primero se presentaba, e ir bajando hasta tener todos TDAs necesarios realizados. Al ver que la cantidad de TDAs mostrados en el informe eran bastantes, y para no mantener un solo archivo con todos los TDAs ahí, se optó por separar y trabajar cada uno de los TDAs en distintos archivos. Por temas de mal entendimiento con algunos TDAs, se decidió el saltar algunos para continuar con los demas, para asi no estar tanto tiempo atascado en el mismo.

6. Aspectos de Implementación

Uno de los elementos que se decidió incorporar de manera externa (llamado a una biblioteca) fue el uso del (require racket/date). Esta biblioteca nos permite de manera mas sencilla el uso de las fechas, utilizando los comandos que pertenecen a dicha implementación.

7. Instrucciones de Uso

En esta sección me gustaría aclarar una modificación realizada a mi código la cual no va correctamente empleada según lo pedido. Al TDA que me refiero es al de añadir una unidad de archivo junto con su letra de partición, nombre y capacidad de su memoria. En esta implementación se nos solicitaba que al implementarlo se debía utilizar una escritura como esta:

Ejemplo de uso

```
;creando la unidad C, con nombre "OS" y capacidad "1000000000 en el sistema
"NewSystem"

((run S0 addDrive) #\C "OS" 10000000000)

;para simplificar, en los siguientes ejemplos se tomará la siguiente definición base

(define S1 ((run S0 addDrive) #\C "OS" 10000000000))
```

Lo que se terminó modificando en mi código se muestra a continuación con un ejemplo:

```
((((run '("NewSystem" (22 5 20023)) addDrive)#\C)"OS)2046
```

Como se puede visualizar, la diferencia con respecto al ejemplo de uso es que la modificación que realicé, separa los 3 valores para añadir la unidad por términos de comodidad y que también almacené cada dato de una unidad en una sublista.

8. Resultado y Autoevaluación

Los Resultados obtenidos respecto a los TDAs terminados fueron positivos, ya que fueron probados varias veces con distintas opciones. Unos ejemplos que se podrían nombrar, es el hecho de que al añadir una unidad en el sistema va a depende primero del largo que contenga el sistema, ya que un archivo sin tener ninguno almacenado antes es distinto a almacenarlo cuando ya hay varios dentro del sistema. Se utilizó la misma idea para el TDA register.
Sobre los TDAs que no se lograron finalizar, es algo distinto, ya que esos no se pudieron terminar de codear, e incluso algunos no se lograron iniciar, por lo que esa parte del laboratorio tendrá varios errores.
La Autoevaluación fue realizada en un archivo .txt externo a este informe, el cual también está integrado en el .zip entregado.

9. Conclusiones del Trabajo

Como cierre de este informe, puedo discernir que el laboratorio no esperaba que fuese tan extenso y complejo de lo que realmente es. El hecho de tener multiples variables funcionando en una sola línea a veces es bastante confuso, por lo que el uso de términos que distinga las funciones es una manera más efectiva para trabajar.
Aún así, considero que si me hubiera organizado mejor, o si se logra tener la posibilidad de optar al comodín, se podría o se podrá dar entrega de un trabajo finalizado que si cumpla con todos los requisitos que se integraron.