Informe Laboratorio 1- Paradigmas de programación

“Creando un Dobble”

Nombre: Alan Francisco Jesús Donoso Villegas.

Profesor: Gonzalo Martínez.

Asignatura: Paradigmas de Programación.

Índice

Introducción

Para el diseño de este laboratorio se utiliza el lenguaje de programación “Scheme”, el cual es un lenguaje que utiliza como base el “Paradigma Funcional”, y el lenguaje se aplicará en un intérprete llamado “Dr. Racket”, el cual facilita mucho mejor su uso e interpretación de este lenguaje. Y utilizando este famoso lenguaje es como crearemos el famoso juego de mesa “Dobble”.

Descripción breve del problema

Se nos Solicita recrear el famoso juego de mesa Dobble utilizando Scheme, el cual es un juego de mínimo 2 jugadores, que consta con una variedad de modos, y un total de 55 tarjetas con 8 símbolos en ella y en donde solamente habrá 1 solo símbolo idéntico con otra tarjeta, y así para las 55 tarjetas. Ahora una breve explicación de modalidades Clásicas del Dobble.

Stack: dicho modo consta de colocar las tarjetas boca abajo e ir sacando 2 tarjetas, donde el primer jugador en encontrar cual es el símbolo que tienen idénticos estas tarjetas, se queda con las 2, y así sucesivamente hasta agotar la totalidad del mazo, Una vez agotado el mazo de tarjetas, se cuentan las que tienen cada jugador, y gana el que tenga la mayor cantidad de estas tarjetas.

Empty Hands Stack: Este modo consta de que a cada jugador se le reparte la misma cantidad de cartas (sin vaciar el mazo completo) y se hace una “Pila” con las cartas restantes, una vez empezado el juego, cada jugador voltea una carta que tenga y debe encontrar una coincidencia con la carta volteada de la pila, el jugador que encuentre la coincidencia descarta su carta y la coloca en la Base de la pila junto a la carta volteada. Gana quien se quede primero sin cartas.

Empty Hands All Players: Al igual que el anterior modo, consta de descartarse las cartas, pero este modo tiene como diferencia que se puede buscar una coincidencia con las tarjetas de los demás jugadores, el primero en encontrar alguna coincidencia, descarta su carta junto con la de la pila. Gana el jugador que vacíe su mazo.

Descripción breve del paradigma

El Paradigma funcional utiliza el “Calculo lambda” como base, el cual es un tipo de notación de funciones que utiliza la famosa “Notación Polaca” o también conocida como “Notación prefija”. En base a este calculo lambda es como se construye el Paradigma funcional, el cual como su nombre indica usa programación única y exclusivamente usando funciones.

Las funciones se construyen en base a una entrada (Dominio) y salida (Recorrido) “UNICA”. Si bajo la misma entrada se obtienen resultados distintos, esto quiere decir que nuestra función no cumple con las características del paradigma Funcional.

Bajo esta premisa, podemos distinguir dos cosas, la primera es que el paradigma funcional trabaja a través de la programación “Declarativa” y que se va a centrar en el ¿Qué hacer? Y no en el ¿Cómo se hace?.

Análisis del Problema

Para la realización del programa, primero debemos realizar una “Abstracción” de nuestro problema, y la cual va a ser que nuestro Dobble va a tener 2 grandes partes, las cuales serán definidas (en nuestro contexto de paradigma Funcional) como TDA’s (Tipo de dato abstracto):

TDA Mazo de cartas (cardsSet): dicho TDA se va a centrar en realizar la estructura de nuestro mazo cumpliendo que nuestro mazo sea un mazo de Dobble y que cumpla con las condiciones que un mazo de Dobble posee, es decir, que no se repita ningún elemento en la misma carta, que entre 2 cartas no se repita mas de 1 solo elemento y que el tamaño sea el necesario para poder jugar al Dobble.

TDA Juego de mesa (Game): en este TDA nos encontramos con la estructura de nuestra mesa de juego y de las modalidades a jugar, donde se debe poder representar un juego por turnos, respetar los turnos y terminar cuando haya un ganador de dicha modalidad.

Para realizar el juego, haremos una segunda abstracción, pero esta vez al juego Dobble, y es que para la generación de un mazo hay matemáticas detrás que nos serán de ayuda.

Cada juego de Dobble consta con un “Orden” del mazo el cual es un numero que nos ayuda a la generación de este, y es que el Orden nos dirá todo lo que

necesitamos, este Orden, debe ser siempre un numero primo o resultado de una potencia de algún primo, ya que, si no cumple estas condiciones, se nos imposibilita crear un mazo “Valido”. El orden también nos sirve para calcular la cantidad total de cartas en el mazo (orden ^2 + orden + 1), la cantidad de símbolos totales distintos (orden ^2 + orden + 1) y la cantidad de símbolos que posee cada carta (orden + 1). Conociendo dicha información la generación de cartas, mazo y juego nos resulta mucho más fácil.

Por último, se identificó que se deben tener ciertas funciones importantes para cada TDA:

TDA mazo: constructor del mazo, verificar si es válido, si posee menos cartas encuentra las que faltan, encontrar todos los datos útiles con una sola carta, entre otras funcionalidades.

TDA Juego: constructor del área de juego y del juego mismo, creación de los modos, creación de las acciones a realizar, entre otras funcionalidades.

Diseño de solución

Para el diseño de nuestra Solución crearemos 2 Archivos importantes y el archivo Main, donde cada uno de los Archivos importantes son el TDA cardsSet (mazo) y el TDA game (juego), y donde vincularemos a cada uno de los TDA entre si (si lo requiere) y luego ambos con el Main (si se necesita). Dentro de cada TDA encontraremos las funciones importantes y algunas extras si así lo requiere.

Dentro del TDA cardsSet encontraremos las siguientes funciones:

- **cardsSet(Función requerida):** Función constructora que se encarga de (como su nombre indica) construir el mazo dado ciertos parámetros de entrada, donde, en nuestra implementación se realizará utilizando más funciones auxiliares.
- **CrearCarta1:** Función que se encarga de crear la primera carta del maso y la ingresa dentro de una lista una vez creada se llama a la función CrearCarta2.
- **CrearCarta2:** Esta función se encarga de crear las siguientes n (orden) cartas una vez creadas se llama la función CrearCarta3.
- **CrearCarta3:** Esta función se encarga de crear las siguientes n*n cartas, una vez creado ya el mazo con cartas (pero números en vez de elementos)

se llama a la función para transformar los números por elementos correspondientes.

- **ChangeElements:** Función que se encarga de Cambiar los números de cada carta por elementos, utiliza una función auxiliar llamada Changer.
- **Changer:** Función que se encarga de cambiar un numero por un elemento correspondiente a la posición en el número.
- **MaxCards:** Función que se encarga de descartar cartas según el máximo de cartas ingresado en el constructor cardsSet.
- **dobble? (Función requerida):** Función de pertenencia que se encarga de verificar si el mazo de cartas creado corresponde a un mazo de Dobble, para la verificación utiliza una función auxiliar llamada OneElementListEqualLists?.
- **OneElementListEqualLists?:** Función de pertenencia que se encarga de verificar si solamente hay un Elemento en común de una carta con el resto del mazo, utiliza para su verificación la función auxiliar llamada OneElementEqual?.
- **OneElementEqual?:** Función de pertenencia que se encarga de verificar si una carta tiene solamente un elemento igual con otra carta, para su verificación utiliza la función auxiliar llamada inCard?.
- **inCard?:** Función de pertenencia que se encarga de verificar si un elemento existe o no dentro de una carta, esta puede ser otra carta o la misma carta.
- **validOrden?:** Función de pertenencia que se encarga de verificar si el numero ingresado para el orden, cumple con las condiciones de orden valido, es decir, que sea un primo o en su defecto una potencia de un numero primo.
- **numCards (Función requerida):** Función selector que retorna el número de cartas actuales en el mazo.
- **nthCard (Función requerida):** Función selector que retorna la carta N (número que debe ser mayor a 1 y menor a la cantidad máxima de cartas del mazo) del mazo.
- **findTotalCards (Función requerida):** Función selector que retorna la cantidad total de cartas dada una carta cualquiera.
- **requiredElements (Función requerida):** Función selector que retorna la cantidad total de elementos en el mazo dada una carta cualquiera.
- **missingCards (Función requerida):** Función selector que retorna una lista con las cartas que le faltan al mazo si es que este se encuentra incompleto para su implementación se usa la función auxiliar cicloFiltro.

- **cicloFiltro:** Función que se encarga de eliminar de un mazo nuevo, todas las cartas que estén repetidas con el mazo con menos cartas.
- **cardsSet->string (Función requerida):** Función modificadora que se encarga de transformar el mazo de cartas completo a un string para luego ser llamado por el usuario con un display para su implementación se usa la función auxiliar
  card->string.
- **card->string:** Función modificadora que se encarga de transformar cada carta por separado en un solo string.
- **rndFn (Función requerida):** Función perteneciente a “otras funciones” que se encarga de otorgar un “numero random” (no es del todo cierto, pero simula ello) dado una semilla previamente dada, esta función puede ser utilizada para distintos ámbitos como randomizar el mazo o extraer una carta random, o utilizar números random para la generación del mazo.

Para el TDA game nos encontramos con las siguientes funciones:

- **Game (Función requerida):** Función constructora del TDA game, se encarga de crear el área de juego implementando cada elemento útil para ello dentro de una lista de elementos.
- **register?:** Función de pertenencia que se encarga de verificar si un registrado se encuentra ya dentro de los registros, retorna True o False según corresponda.
- **whoseTurnIsIt? (Función requerida):** Función de pertenencia que se encarga de verificar a quien le corresponde el turno actual.
- **get-Table:** función selectora que se encarga de obtener el tablero de juego del “game”.
- **get-cardsSet:** función selectora que se encarga de obtener el mazo de cartas del “game”.
- **get-Cards:** función selectora que se encarga de obtener el número de cartas restantes del “game”.
- **get-Registers:** función selectora que se encarga de obtener la lista de registrados actuales del “game”.
- **get-CardsRegisters:** función selectora que se encarga de obtener la lista con las cartas que posee actualmente cada registrado del “game”.
- **get-NumPlayers:** función selectora que se encarga de obtener el número de jugadores del “game”.
- **get-Mode:** función selectora que se encarga de obtener el modo de juego del “game”.

- **get-RndFn:** función selectora que se encarga de obtener la función random del “game”.
- **get-Turn:** función selectora que se encarga de obtener el turno actual del “game”.
- **stackMode (Función requerida):** función selectora que retorna las 2 cartas que se encuentran al tope del mazo de cartas (cardsSet).
- **register (Función requerida):** Función modificadora que se encarga de registrar a un nuevo usuario dentro de los registros del game, si el usuario ya está registrado o en su defecto al registrarlo supera el limite de jugadores para el juego, retornará un string explicando el problema.
- **play (Función requerida):** Función que permite realizar la primera jugada dada una acción.
- **status (Función requerida):** Función que retorna el estado actual del juego y si este está finalizado o no.
- **score (Función requerida):** Función que retorna el puntaje actual de los jugadores.
- **game->string (Función requerida):** Función que transforma a string el juego completo para luego el usuario mostrarlo con la función display.
- **addCard (Función requerida):** Función que se encarga de añadir una carta al mazo, pero que tiene que verificarse que esta no se encuentre dentro del mazo y cumpla con las condiciones para ser ingresado dentro de él.
- **empyHandsStackMode (Función requerida):** Función que representa el modo de juego del mismo nombre.
- **myMode (Función requerida):** Función que representa una modalidad de juego nueva sin romper las reglas del juego, el programador define que modo de juego quiere representar en esta función.

Aspectos de implementación

Para la implementación de nuestro código se utilizó el intérprete de Scheme llamado Dr. Racket el cual otorga herramienta bastantes útiles para la aplicación e implementación de nuestra solución.

La estructura del código se divide en dos códigos los cuales son, uno para el TDA cardsSet y el otro para el TDA game, y luego ambos son llamados con require en el archivo Main para realizar el juego.

Los nombres de los archivos son TDA_cardsSet_20711629-7_DonosoVillegas.rkt y TDA_Game_20711629-7_DonosoVillegas.rkt, y el del Main es Main_20711629-7_DonosoVillegas.rkt

Instrucciones de uso

Para poder ejecutar nuestro archivo debe descargar la carpeta completa que se encuentra en el repositorio de github, una vez descargado, se puede abrir cualquiera de los archivos .rkt, los cuales los de gran importancia son el TDA cardsSet y el TDA Game, puede abrir cualquiera de estos y hacer pruebas utilizando los ejemplos ubicados al final de cada código, para ello debe apretar el botón RUN que se encuentra arriba a la derecha del interprete, y en el apartado de la consola ingresar los ejemplos dados.

Es posible que se encuentren errores si es que se ingresa algún parámetro que esté fuera del dominio establecido para ello, procure evitar romper las reglas del dominio para la ejecución y pruebas del programa.

Resultados y autoevaluación

Los resultados que se obtuvieron no fueron completamente los esperados, pero si cumple con la gran mayoría de funciones requeridas y se utilizaron algunas extras para facilitar el trabajo e implementación de las funcionalidades. Hay algunas funciones que no se cumplieron debido al escaso tiempo que tuve para programar (culpa mía por no organizar mejor mi tiempo), pero se buscó realizar lo más posible.

Para la autoevaluación se realizará un puntaje para cada una de las funcionalidades requeridas y requerimientos no funcionales, los puntajes serán realizados bajo la siguiente especificación: 1 funciona al 100%, 0,75 funciona en la gran mayoría de las veces, 0,5 Funciona la mitad de las veces, 0,25 Funciona bajo ciertas condiciones especificas , pero no en otras a pesar de estar dentro del dominio.

• TDA	• 1
• cardsSet	• 1
• Dobble?	• 1
• numCards	• 1
• nthCard	• 1
• findTotalCards	• 1
• requiredElements	• 1
• missingCards	• 0,75
• cardsSet->String	• 1
• game	• 1
• stackMode	• 0,75
• register	• 1
• whoseTurnIsIt?	• 1
• play	• 0
• status	• 0
• score	• 0
• game->string	• 0
• addCard	• 0
• emptyHandsStackMode	• 0
• myMode	• 0

Conclusión

Si bien no se cumplieron con los resultados esperados, lo que se logró, se hizo casi en su totalidad, y se cumplió con el objetivo principal que fue utilizar y aprender correctamente el uso del Paradigma Funcional, Dr. Racket y TDA's que eran el objetivo de este laboratorio (aplicar el paradigma Funcional y Scheme).

Debido a retraso a la hora de empezar con la programación, no se cumplió con el objetivo de versionamiento en git al 100%, ya que todos los commit se concentran casi en la última semana.

Se espera que para el siguiente laboratorio no se empiece tan tarde y se pueda cumplir con la totalidad de los objetivos principales y resultados esperados.