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

INFORME DE LABORATORIO 1:

JUEGO DE CARTAS ESTILO DOBBLE EN SCHEME

Nombre: Nicolas Rojas G.
Profesor: Roberto Gonzales I.
Asignatura: Paradigmas de Programación

Tabla de Contenidos

1. INTRODUCCIÓN

Este informe presenta el laboratorio 1 del curso Paradigmas de Programación, en el cual por este proyecto se utiliza el lenguaje de programación Scheme, que corresponde al paradigma funcional, a través del compilador Dr. Racket. El informe contiene lo siguiente: Descripción del problema, una descripción del paradigma, los objetivos del proyecto, análisis del problema, diseño de la solución del problema, aspectos de implementación, instrucciones y ejemplos de uso, resultados y autoevaluación y una conclusión al proyecto realizado.

1.1 DESCRIPCIÓN DEL PROBLEMA

Se pide crear un programa el cual permita a dos o más personas jugar al juego de mesa Dobble. Este juego consta con un conjunto finito de cartas. En cada tarjeta existen n elementos diferentes. Además, para cualquier par de cartas, existe un y sólo un elemento en común. A partir del conjunto de cartas, se pueden jugar alguno de los siguientes modos de juego:

Stack: Se disponen el conjunto de cartas apiladas por el reverso en el tablero. Luego se retiran las dos primeras cartas de la pila y se voltean para revelar su contenido sobre la mesa. A continuación, el primer jugador que identifique el elemento común entre estas dos cartas se queda con las cartas. El juego termina cuando no quedan más cartas por voltear y gana aquel jugador que tiene más cartas.

EmptyHandsStack: Cada jugador parte con un número igual de cartas en su poder y debe quedar una pila de cartas al centro. Las cartas asignadas al jugador y las del stack deben estar volteadas sin revelar su contenido. A continuación, se voltea una carta desde la pila y todos los jugadores voltean una de las que tienen en su poder. El primero que identifique coincidencias entre su carta y la de la pila, puede descartar su carta junto a la que se volteó desde la pila. La carta descartada y la de la pila se reubican en la base de ésta. Por otro lado, el resto de los jugadores vuelve a recoger su propia carta. El juego continúa hasta que uno de los jugadores quede sin cartas en su poder, quién será el ganador.

EmptyHandsAllPlayers: Una variante de la modalidad 2 es que los jugadores pueden identificar coincidencias entre la carta que voltearon y las de los otros jugadores o la que se volteó desde la pila. El jugador que primero identifique coincidencias puede descartar su carta y la de la pila ubicándolas en la base de ésta.

Las modalidades en este proyecto se juegan por turnos, además, una partida puede terminar prematuramente si así se decide y se determina un ganador a partir del último estado del juego.

1.2 DESCRIPCIÓN DEL PARADIGMA

El paradigma funcional está basado en la definición de predicados y el uso de verdaderas funciones matemáticas. Se ve representado por Scheme (lenguaje que se usa en este proyecto), Haskell, e incluso Python. La programación mediante este paradigma tiene sus raíces en el cálculo lambda, usado para investigar la naturaleza de las funciones, la naturaleza de la computabilidad y su relación con la recursión. Se tienen algunos elementos bastante importantes a la hora de construir un código utilizando este paradigma, como, por ejemplo:

Recursividad: se utiliza normalmente para iterar, llamándose a sí mismas, permitiendo que la operación se realice una y otra vez hasta alcanzar un caso base. Dentro de la recursividad se pueden llevar a cabo funciones recursivas naturales, de cola, arbóreas, entre otras.

Funciones de orden superior: estas funciones permiten la aplicación parcial, una técnica en al que se aplica una función a sus argumentos una a la vez, haciendo que con cada aplicación devuelva una nueva función que acepta el siguiente argumento.

Currificación: es un mecanismo para la invocación parcial de funciones. En la currificación se ocupan funciones anónimas bajo el concepto de funciones de orden superior.

1.3 OBJETIVOS

Este proyecto tiene como objetivo aprender a utilizar las herramientas que nos proporciona el lenguaje funcional, forzando a pensar y utilizar funciones de la manera en que el paradigma funcional nos lo permite, además de reconocer características fundamentales para la programación que ayudarán a entender el resto de los paradigmas que se planean en el curso. También tiene como objetivo aprender a manejar el lenguaje de programación Scheme, utilizarlo mediante el intérprete Dr. Racket y mantener un ritmo de trabajo constante utilizando la herramienta Git para controlar las versiones del proyecto.

2. DESARROLLO

2.1 ANALISIS DEL PROBLEMA

El juego Dobble se rige por propiedades matemáticas dentro de los cuales destacan los números primos, los planos proyectivos finitos y el plano de fano, entre otros, para la generación del mazo de cartas. Mediante el algoritmo en JavaScript (*Ver Figura N°1 en Anexos)* se puede observar una serie de variables que se declaran antes de comenzar a generar las tarjetas. La variable n se puede cambiar para determinar el orden del plano proyectivo para el que se quiera crear una matriz de incidencia, permitiendo a los bucles externos comenzar a agregar los símbolos necesarios para las cartas.

En la programación funcional no existe la iteración como tal, por lo que los ciclos for se definirán a través de funciones con llamadas recursivas haciendo uso de funciones de orden superior.

Para que el juego funcione de manera correcta, se listan las funcionalidades necesarias:

- Generar un conjunto de cartas válido a partir de un conjunto de elementos.
- Determinar si un set de cartas dado es válido.
- A partir de una sola carta, determinar cuántos elementos ser requieren para generar un ser válido.
- A partir de una sola carta, determinar cuántas cartas se pueden generar en el set.
- A partir de un ser incompleto, determinar cuáles son las cartas que faltan para completarlo.
- Permitir la generación manual de un mazo de cartas donde el usuario produce las cartas individualmente y las va insertando en el mazo con las correspondientes verificaciones para garantizar que la carta generada es válida dentro del set.
- Permitir que dos o más jugadores jueguen por turnos en alguna de las modalidades, lo que puede incluir funcionalidades como:
    a) Registrar jugadores en la partida.
    b) Apilar y ordenar cartas.
    c) Repartir cartas.
    d) Controlar turnos.
    e) Determinar ganador.

f) Contabilizar puntaje.

g) Conocer el estado del juego.

h) Terminar un juego.

i) Pasar un turno.

Para crear las cartas, el mazo, y la mesa con los datos necesitamos los siguientes 2 TDA:

- **Cards:** corresponde a una función que usa estructuras basadas en listas, usando explícitamente los tipos de recursividad que nos proporciona el lenguaje, usando funciones de orden superior. Este tiene como argumentos del dominio una lista de elementos que sirven para construir el conjunto de cartas, un número entero que indica la cantidad de elementos que debe llevar una carta, otro número que indica la cantidad máxima de cartas que puede generar un conjunto y una función de aleatorización que debe garantizar transparencia referencial. El recorrido de este TDA es una lista de listas, en la que se almacenan los valores de las cartas del mazo.
- **Game:** corresponde a una función que utiliza funciones recursivas con el uso de funciones de orden superior. Este tiene como argumentos del dominio un entero con la cantidad de jugadores, una lista con un conjunto válido de cartas, una función que determina el modo de juego y una función de aleatorización con transparencia referencial.

Cada TDA (Tipo de Dato Abstracto) debe tener un archivo único, donde solo se implemente lo necesario para cada TDA y un archivo principal que contenga las funciones, donde en este se llamen a las funciones de los TDAs creados.

2.2 DISEÑO DE LA SOLUCIÓN

Debido a que los requerimientos funcionales requieren el uso explícito de los constructores para TDA Cards y TDA Game, los archivos correspondientes a cada TDA tienen funciones que les permitan el uso correcto de ellas con funciones de pertenencia, selectores, modificadores, entre otras.

2.2.1.1 TDA CARDS

- Representación: una lista (mazo) que contiene listas (cartas) con los elementos dados en la construcción del set de cartas.
- Constructor: cardsSet toma como entradas una lista de elementos, el número de elementos que contiene cada carta, el máximo cantidad de cartas a generar y una función de aleatorización.
- Función de Pertenencia, Selectores, Modificadores y Otras Funciones.

2.2.1.2 TDA GAME

- Representación: una lista (mesa) que contiene: una lista con las cartas que están en la mesa, las cartas que tienen los jugadores, la lista de jugadores, la lista con el mazo disponible, un string con el estado de la partida, un string que contiene el nombre del jugador el cual tiene el turno y una lista con las puntuaciones de los jugadores.
- Constructor: game toma como entradas el número de jugadores, la lista del mazo de cartas, una función con el modo de juego y una función de aleatorización.
- Función de Pertenencia, Selectores y Otras Funciones.

2.2.2 FUNCIONES OBLIGATORIAS

cardsSet: función constructora con llamada recursiva de cola, la cual mediante creaConjunto genera las cartas ordenadas usando la lista de elementos y el número de elementos que debe haber por carta, con limitaCartas entrega la cantidad de cartas que se piden en la entrada y con makeShuffle se desordenan aleatoriamente las cartas usando la función de aleatorización.

dobble?: función que se encarga de verificar si las cartas del mazo cumplen las condiciones usando recursión natural. Con validaDiferencias verifica que todos los elementos de una carta sean diferentes y con comparaCartas verifica que tengan solo un elemento en común.

numCards: función que retorna la cantidad de cartas que hay en un mazo.

nthCard: función que retorna una carta del mazo según la posición entregada en la entrada.

findTotalCards: función que, a partir de una carta de muestra, retorna la cantidad de cartas que se deben generar para completar un mazo válido.

requieredElements: función que, a partir de una carta de muestra, retorna la cantidad de elementos que se deben generar para completar un mazo válido.

missingCards: función recursiva de cola que, a partir de un mazo limitado de cartas, genera el resto de las cartas necesarias para formar un mazo válido.

cardsSet->string: función recursiva de cola que, a partir de un mazo de cartas, imprime en pantalla las cartas a través de la función display (esta función tiene display dentro de sí).

game: función recursiva de cola que, usando funciones de orden superior, genera una lista con las cartas que están en la mesa, las cartas que tienen los jugadores, la lista de jugadores, la lista con el mazo disponible, un string con el estado de la partida, un string que contiene el nombre del jugador el cual tiene el turno y una lista con las puntuaciones de los jugadores y todo esto dependiendo del modo de juego y función que se le pase de entrada.

stackMode: función que permite retirar y voltear las dos cartas superiores del stack de cartas en el juego y las dispone en el área de juego.

register: función recursiva de cola que, registra aun usuario dentro de una partida. Si el usuario ya existe, no lo agrega, y si se intenta agregar un usuario más fuera de los argumentos, entrega los ya registrados anteriormente.

whoseTurnIsIt?: función que a partir de la lista game, determina a qué jugador le toca jugar.

2.3 ASPECTOS DE IMPLEMENTACIÓN

2.3.1 COMPILADOR

En este proyecto se utiliza el compilador Dr. Racket versión 6.11 o superior. Este intérprete tiene funciones como Debug que ayudan a encontrar errores a la hora de probar las funciones, y de esa manera poder hacer una autoevaluación de las funciones. Se pueden utilizar todo tipo de funciones de Scheme y Racket, pero no importar librerías.

2.3.2 ESTRUCTURA DEL CODIGO

Para la implementación de este programa, el cual está separado en archivos.rkt según los TDAs creados y el archivo principal, se implementan las funciones al archivo principal con "require".

A los archivos de los TDA se les añade "(provide (all-defined-out))". Para que el archivo main pueda importar las funciones sin necesidad de hacerlo una por una.

En total, se tienen 3 archivos: "tdaCards.rkt", "tdaGame.rkt", y el archivo main "labDobble.rkt".

2.4 INSTRUCCIONES DE USO

2.4.1 EJEMPLOS DE USO

Primero, se debe verificar que estén los 3 archivos dentro de la misma carpeta, ya que, con la falta de uno, el archivo main no podrá funcionar. Una vez verificado lo anterior, se abre el archivo main con el compilador Dr. Racket y se presiona el botón "Run" que está en la parte superior derecha.

Una vez ejecutado, el programa estará listo para recibir órdenes. Al final del código hay algunos ejemplos que puedes utilizar para comprobar la efectividad de cada uno de ellos.

En el caso de querer probar el código utilizando funciones propias, asegura lo siguiente:

1. Crea un elementsSet con al menos la cantidad de letras que tiene el abecedario inglés.
2. Crea un numPlayers mayor o igual a 2.
3. Se recomienda que numElementsPerCard sea un número primo.
4. Se recomienda dejar maxCards en 0 si no sabe cuántas cartas quiere generar.

2.4.2 RESULTADOS ESPERADOS

Se espera que se genere el mazo correctamente utilizando los elementos dados por el usuario para que los jugadores puedan comenzar a jugar con las funciones de modos de juego.

2.4.3 POSIBLES ERRORES

Pueden ocurrir errores en missingCards por la cantidad de elementos que necesita generar. A veces no encuentra un conjunto que sirva para que al llenar el mazo este sea válido.

2.5 RESULTADOS Y AUTOEVALUACIÓN

2.5.1 RESULTADOS OBTENIDOS

Los resultados obtenidos para todas las funciones creadas han sido satisfechos, excepto por missingCards que tiene el problema ya antes mencionado. No consideré una manera óptima y por falta de tiempo es mejor apretar “Play” hasta que genere un conjunto válido. Se lograron hacer 13/20 funciones.

2.5.2 AUTOEVALUACIÓN

La Autoevaluación se realiza de la siguiente forma: 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. (*Ver Figura N°2 en Anexos)*

3. CONCLUSIÓN

Al terminar este proyecto se concluir que los objetivos estuvieron casi del todo cumplidos. No se lograron implementar todas las funciones que se pedían en el proyecto. La mayor complicación que hubo fue el pensar una manera de implementar un algoritmo como el de JavaScript a un paradigma de programación como el funcional. Tener que definir funciones para simular un ciclo for junto con operaciones matemáticas utilizando funciones recursivas fue la parte más desafiante y que más tiempo tomó. Luego de eso el uso de funciones currificadas costó en un principio entender, pero se pudieron implementar en las funciones necesarias. El no verificar constantemente los valores obtenidos por los constructores también jugó una mala pasada, ya que el constructor de cardsSet entregaba valores, pero no los que debería entregar junto con el algoritmo para generar un mazo de cartas válidos.

El laboratorio requiere de mucho tiempo y dedicación, se debe planear con mucha anticipación la evaluación de los argumentos para tener una idea de cómo implementar lo solicitado. A pesar de eso, se pudo arreglar una gran cantidad de errores, permitiendo que el programa a pesar de no estar completo funcione muy bien, con uno que otro detalle como el de missingCards.

A pesar de las complicaciones se aprendió mucho sobre el paradigma funcional, su potencial para resolver problemas, los conceptos clave de este paradigma como la recursión y el manejo de listas y el uso constante de Git para las versiones del programa. Se espera que el conocimiento adquirido sirva para el resto de proyectos del ramo Paradigmas de Programación y para hacer frente en un futuro en el que un lenguaje de programación utilice la programación funcional.

4. BIBLIOGRAFÍA Y REFERENCIAS

1. Gonzales, R. (2022). “Proyecto Semestral de Laboratorio”. Paradigmas de Programación. Enunciado de Proyecto Online. https://docs.google.com/document/d/1tCVZBGScJbXFspQfu6WlDl3PsyCJUcqIgos5 DQz7k/edit
2. Dore, M. (2021, 30 diciembre). The Dobble Algorithm - Micky Dore. Medium. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52
3. E. (2016, 1 marzo). Qué es la «Currificación» en programación funcional. campusMVP.es. https://www.campusmvp.es/recursos/post/Que-es-la-Currificacion-en-programacion-funcional.aspx
4. Flatt, M. y Bruce, R. (2022). “The Racket Guide’’. The Racket Reference. Documentación Online. https://docs.racket-lang.org/guide/

5. ANEXOS

Figura N°1 (Código en JavaScript)

```
let i, j, k

let n = 3                 //order of plane, must be a prime number
let numOfSymbols = n + 1  //order of plane + 1

let cards = [] //the deck of cards
let card = []; //the current card we are building

//to start, we build the first card
for (i = 1; i<= n+1; i++) {
    card.push(i)
}
cards.push(card)

//then we build the next n number of cards
for (j=1; j<=n; j++) {
    card = []
    card.push(1)

    for (k=1; k<=n; k++) {
        card.push(n * j + (k+1))
    }
    cards.push(card)
}

//finally we build the next n² number of cards
for (i= 1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        card = []
        card.push(i+1)

        for (k=1; k<= n; k++) {
            card.push(n+2+n*(k-1)+(((i-1)*(k-1)+j-1)%n))
        }
        cards.push(card)
    }
}
```

Figura N°2 (Autoevaluación)

cardsSet	1
dobble?	1
numCards	1
nthCard	1
findTotalCards	1
requieredElements	1
missingCards	0.75
cardsSet->string	1
game	1
stackMode	1
register	1
whoseTurnIsIt?	1