Informe Laboratorio 1: Simulación en Scheme (Racket) de juego “Dobble”

Nombre	RUT
1.- Alejandro Cortés Cortez	20.722.036-1

Profesor: Roberto González
Sección: A-1

Índice:

Introducción: **3**
- Descripción del problema: 3
- Descripción del paradigma 3
- Objetivos 4

Desarrollo: **4**
- Análisis del problema 4
- Diseño de la solución 5
  - TDA card 5
  - TDA cards 6
  - TDA cardsSet 6
  - TDA Area * 6
  - TDA Player 6
  - TDA Players 7
  - TDA Game 7
  - Extras 7
- Aspectos de implementación 7
- Instrucciones de uso 8
  - Posibles errores 8
- Resultado y autoevaluación 8

Conclusión **9**

Bibliografía **9**

Anexos **9**

1. Introducción:

El presente informe trata respecto a la implementación en Racket de un popular juego de mesa llamado “Dobble”, esto basándonos en el Paradigma funcional, del cual se hablara más adelante

1.1. Descripción del problema:

Se pide implementar el juego “Dobble”, el cual es un juego de “cartas” que tiene distintas modalidades de juegos, pero su principal complejidad viene respecto a las cartas que componen el juego, el cual se compone de 55 cartas, donde cada carta tiene en común un solo elemento con cualquier otra carta del juego, para lograr esto, hay que dividir el problema en distintas partes, teniendo así un juego, los jugadores, un área de juego, un set de cartas, y las cartas por si solas.

1.2. Descripción del paradigma

Todo esto se basa en el paradigma funcional, el cual en un breve resumen es un paradigma que basa todo en funciones, por lo que se dice comúnmente que todo dentro de este paradigma son funciones, además, gran parte de su base viene respaldada por él “calculo lambda”, que es una manera distinta a la usual respecto de como ver las funciones, ya que a diferencia de como siempre hemos trabajado, este tipo de cálculo no trabaja con notación infija (el operador al medio de los operandos), sino que trabaja con notación prefija, donde primero tenemos un operador y luego los operandos.

Al estar trabajando todo como funciones, podemos tomar en cuenta axiomas de estas, como saber que para cada entrada, siempre debe existir una sola salida, además, a estas “entradas y salidas” se les conoce como dominio y recorrido, debido a lo antes mencionado. Por ende, lo que se hace principalmente es que para cada elemento del dominio, se le aplicara un procedimiento y respectivamente, entregara un elemento que pertenece al recorrido. Además, para este paradigma principalmente trabajamos de manera recursiva los problemas, esto debido a que es una de las cosas heredadas del calculo lambda.

Este paradigma se engloba dentro de la programación declarativa, la cual no es tan común dentro del mundo de la programación, donde los principales y más reconocidos lenguajes trabajan mediante la programación imperativa, lo que, traducido a lenguaje natural, significa que se programa buscando él “¿Cómo?”, siempre buscando cómo hacer algo, y como se llega a un resultado, pero la idea principal de la programación declarativa es no enfocarnos en lo anterior, sino que enfocarnos en la solución, por lo que ahora la pregunta debe ser “¿Qué?”. A pesar de no ser tan conocida ni popular la programación declarativa, los programadores frecuentemente la utilizan sin saber al respecto, principalmente los que trabajan con bases de datos como SQL o similares.

Al no ser tan usual el uso de este paradigma funcional, hay pocos lenguajes que respeten de manera pura el paradigma, la mayoría hace uso del paradigma en conjunto con algunas cosas de la programación imperativa, como sería el caso de Racket (Scheme), lenguaje que

se usará en este trabajo, y tiene como una de sus bases LISP, por lo que trabaja como un procesador de listas, Racket permite cosas que no se podrían hacer en el paradigma funcional puro, como tener variables, ciclos iterativos, entre otros, pero se tienen algunas ventajas destacables que ayudan a no depender de estas limitaciones, como son las Funciones anónimas, funciones las cuales no necesitan de un nombre, La composición de funciones, donde una función se evalúa dentro de otra función, generando así una nueva función, Las funciones de orden superior, que están relacionadas con la composición de funciones, y son aquellas que dentro de su dominio o recorrido contienen otra función, Currificación, donde tenemos una función con múltiples elementos en su dominio, pero en vez de evaluar todos al instante, se puede ir evaluando a medida que es necesario, Recursividad, que como se mencionó antes, es la principal manera de repetir procesos para simplificar la programación, esta recursividad tiene distintos tipos, entre los cuales destacan la recursión natural, la recursión de cola y la recursión arbórea, pero en este trabajo se priorizaran los dos primeros tipos, aunque principalmente el segundo, recursión de cola, ya que para lo que se requiere, es la más eficiente en su manera de trabajar.

1.3. Objetivos

El principal objetivo de este proyecto es lograr adecuarse a distintas maneras respecto a como solucionar un problema, logrando así adaptar la manera de pensar, tanto como la de programar para ser capaces de resolver el problema entregado, debido a esto surgen distintos objetivos secundarios. Los cuales son en primer punto ser capaces de mejorar la capacidad de analizar soluciones recursivas para un problema, como también implementarlas, lo cual deriva en un segundo objetivo secundario, el cual consiste en aprender a utilizar la herramienta “Dr. Racket” que se basa en el lenguaje Racket, el cual es el tercer objetivo secundario, que consiste en aprender las instrucciones básicas de este lenguaje, pero evitando instrucciones que salgan fuera de lo que es paradigma funcional.

2. Desarrollo:

2.1. Análisis del problema

Para comenzar a analizar esta problemática, debemos tener en cuenta que no se tienen datos almacenados en memoria, como tampoco hay variables, por ende todo lo que sucede en este trabajo va realizándose al momento, con las instrucciones entregadas y después se recibe el resultado mediante la consola, más no queda almacenado en ninguna parte, lo cual implica que además, nada se modifica, sino que se reconstruye. Luego, analizando la principal problemática, tenemos de manera más general, a más específica, distintas componentes, donde primero que nada se requiere tener el Juego, donde a pesar de no ser como funciona el juego originalmente, se solicita que funcione por turnos, los turnos serán asignados a los Jugadores, quienes son aquellos que deberán jugar el juego, ya que más específicamente hablamos de “Dobble”, lo que el jugador deberá hacer consiste en 4 instrucciones primordiales, que son retirar cartas de un stack de cartas

(denominado posteriormente como Área), pasar (no hacer nada en su turno), Comparar cartas, o terminar el juego, ahora, dependiendo de la modalidad de juego, puede variar cada instrucción, en este caso se pide principalmente implementar dos modalidades, la primera de ellas denominada “stackMode”, la cual consiste en ir sacando de a dos cartas desde el stack (que contendrá un set completo de cartas) al área de juego, y el jugador deberá encontrar un elemento en común entre ambas cartas, si la encuentra se las queda el jugador y aumenta así, su puntaje, esta modalidad de juego concluye cuando el stack de cartas queda vacío y gana aquel jugador con mayor puntaje, o sea, más cartas en su poder, la segunda modalidad, que se denomina “EmptyHandsStack” consiste en que cada jugador comenzara con igual cantidad de cartas en su poder y la misma cantidad de cartas queda en el centro, luego se debe voltear una carta del stack para que quede en el área de juego, y el jugador de turno deberá voltear una carta de las que tiene en su poder, para luego buscar coincidencias entre ambas cartas, en caso de encontrarla, ambas cartas se dejan en la base del stack de cartas, en caso de no encontrarla, la carta del jugador vuelve a sus cartas y la que está en el área vuelve a la base del stack, el juego concluye cuando algún jugador queda sin cartas, y por ende, se convierte en el ganador, además del juego y sus modalidades, como se pudo apreciar, se necesitan jugadores, los cuales se deben registrar con un nombre único y así luego estarán habilitados para jugar, cada jugador tendrá su mazo de cartas, que puede estar vacío o con cartas dependiendo de la modalidad, además será necesario tener un área de juego, donde estarán las cartas volteadas, estas cartas provienen de un set de cartas, que será formado a través de un algoritmo *(Dore, 2021) ,* este set de cartas está formado claramente por cartas, las cuales también deberán ser parte del diseño del proyecto, esto será a grandes rasgos la manera en la que se representara el problema según los requerimientos.

2.2. Diseño de la solución

Para la implementación será necesario que cada uno de los componentes principales señalados anteriormente, sean un TDA, que es la representación asignada para poder ser trabajada en el software. Como se trabaja en Racket, que proviene de LISP, la mayoría de representaciones provendrá de Listas. Mostrando desde lo más específico a lo más general, ya que algunos dependen de otros TDA, tendremos los siguientes TDA:
TDA - representación -constructor -> funciones principales

2.2.1. TDA card

- Representación : Es una lista que contendrá los elementos que le sean entregados, pudiendo ser cualquier tipo de elemento que requiera el usuario, en caso de no tener elementos, será un null (lista vacía).
- Constructor: Recibe una cantidad X de elementos y los devuelve en una lista
- Tiene 2 funciones principales, una la cual es un booleano, llamada cartaVacia?, que retorna true si la carta está vacía, si no, retorna false, por lo que es considerada una función de pertenencia, además contiene la función addSymbol, que se encarga de añadir elementos dentro de una carta recibida, por lo que es una función modificadora.

2.2.2. TDA cards

- Representación: Es una lista que contendrá uno o varios Card, en caso de no tenerlos, será un null.
- Constructor: Como si, no tiene un constructor, pero tenemos dos funciones que principalmente irán construyendo el Cards, la primera de ellas es el addCardToCards, la cual se encarga de ir añadiendo al inicio una carta a un Cards entregado, y la función addCardFinalToCards, que hace lo mismo, pero "añadiéndola" al final. (se especifica para el futuro que añadir es una manera simple de explicar, pero realmente la lista se reconstruye con el nuevo valor, además, del resto de valores anteriores)
- Tiene más funciones, las cuales puedes ver detalladas en el anexo (*Tabla de funciones)*.

2.2.3. TDA cardsSet

- Representación: Es una lista, que contendrá cards (un conjunto de card) y además una lista de elementos (que pueden ser de cualquier tipo), estas cartas serán las cartas del juego, generando un set de cartas válido de dobble.
- Constructor: En caso de ser un cardsSet sin nada, su constructor sera emptyCardsSet, que genera un cards vacío y además lo junta con un null (ya que no hay elementos), y en caso contrario, tenemos cardsSet, que genera un conjunto válido de cartas (para ser usado en dobble, lo cual significa que cada carta debe tener un elemento en común con el resto de cartas), ademas de “barajar” estas cartas, según una función de random entregada.
- Tiene más funciones, las cuales puedes ver detalladas en el anexo (*Tabla de funciones)*.

2.2.4. TDA Area *

- Representación: Es una lista, que contiene Cards, las que serán extraídas desde un cardSet, se utiliza para representar las cartas que estarán visibles al momento de jugar (cartas volteadas).
- Constructor: Tenemos dos opciones, gameAreaVacia crea un area vacía y addCardsToArea, que va añadiendo cartas de a una al area.
- Del resto de funciones destacan dos funciones selectoras que son muy importantes, isSymbolInArea? Que consulta si hay un símbolo (que puede ser cualquier elemento contenido en una lista) dentro de un area, esto es muy útil a la hora de hacer un spotIt y también isSymbolInAreaCards? Que consulta si es que el símbolo está en TODAS las cartas sobre el area, igualmente hay más funciones las cuales puedes ver detalladas en el anexo (*Tabla de funciones)*

2.2.5. TDA Player

- Representación: Es una lista, que contiene un nombre, un entero que representa el puntaje del jugador, el cual aumentara cuando sea requerido según en el modo de juego, y un cards que serán las cartas del jugador.

- Constructor: newPlayer, que según un nombre, genera un jugador con ese nombre, con un score inicial de 0 y sin cards.
- Tiene más funciones, las cuales puedes ver detalladas en el anexo (*Tabla de funciones*).

** (sin tilde, para evitar errores al codear)*

2.2.6. TDA Players

- Representación: Es una lista, que contiene uno o varios player, en caso de no tenerlos es un null.
- Constructor: playersVacio, que simplemente es cuando players esta vacio, y addPlayer que va añadiendo player, solamente recibiendo el nombre del player y un players donde guardarlo (el cual considero constructor porque así se va construyendo un players) y addPlayerToFinal que va añadiendo player al final de un players.
- Tiene más funciones, entre las que destacan addCardToPlayer que recibe un nombre, una card, un players (donde debería estar contenido algún player con el nombre recibido) y un null (que será usado después en la recursión, y como es una función interna no influye que pida un null al final), addCardsToPlayer que recibe un cards , y de igual manera que la función anterior le añade las cartas a un player, además tenemos la función isPlayer?, que recibe un nombre y detecta si el player con ese nombre ya esta en un players. El resto de funciones las puedes ver detalladas en el anexo xx.

2.2.7. TDA Game

- Representación: Engloba todo lo anterior para poder ser utilizado como es requerido, Es una lista, que contiene Area, un cardsSet, un entero que representará de quien es el turno de jugar, un entero que representará el máximo de jugadores, un players, un string que dirá cuál es el estado actual del juego, una función que es el modo de juego y una función que aleatoriza números.
- Constructor: game vacío, que crea una lista con los valores por defecto de lo antes descrito, una función game_to_modify que simplemente sirve de intermediario para modificar los game, y un game que recibe una cantidad máxima de jugadores, un cardsSet una función que será el modo y la función que aleatoriza.
- Tiene muchas funciones útiles, así que serán todas descritas en el anexo.

2.2.8. Extras

- Hay funciones extra que servirán en distintas partes, por ende no pertenecen a ninguno de los TDA.

2.3. Aspectos de implementación

Para este trabajo se usó el IDE Dr. Racket, que es el IDE predeterminado de Racket, de hecho, viene por defecto instalado junto a Racket, el cual fue instalado en la versión 6.11, por ende el código fuente es válido para ser ejecutado en esta versión o cualquiera sea superior a ella, Además se usó GIT para el manejo de versiones del proyecto, para así

mantener un registro del constante trabajo realizado, esto debido a que se trabaja en distintos archivos, por ende es muy para mantener un orden como para ver que cosa se modificó en cada momento en caso de tener algún error, también respecto a esto, sé trabajo en distintos archivos de Racket, específicamente uno para cada TDA especificado anteriormente, y dos adicionales, sin uso de bibliotecas ni librerías externas.

2.4. Instrucciones de uso

2.4.1. Posibles errores

Debido al escaso tiempo, la función dobble? del TDA cardsSet puede presentar errores al ingresar cartas con elementos desconocidos, pero son casos muy específicos y no afectan al desarrollo general de la problemática, aunque también deriva a que funciones como addCard del TDA cardsSet igualmente, tengan algunos errores con algunas cartas, se podría solucionar de múltiples maneras, pero el tiempo es escaso.

2.5. Resultado y autoevaluación

3. Conclusión

Afrontar un cambio de paradigma al principio puede resultar bastante complejo, debido a que todos los conocimientos anteriores de programación ya no eran útiles en este lenguaje, esto principalmente enfocado a cosas muy básicas como eran las variables, nunca hubiese sido posible imaginar la idea de programar sin variables, como tampoco era algo esperable tener que estar siempre trabajando con recursión, sin tener la capacidad de realizar ciclos iterativos, pero a medida que se fue avanzando en el proyecto y aprendiendo al respecto, además de la práctica y experiencia, fueron mejorando y ayudando a trabajar de mejor manera, para finalmente lograr comprender de buena manera como desarrollar en este paradigma, y convirtiendo esta experiencia en algo grato, pero lamentablemente no se pudo disfrutar al 100% la experiencia por el acotado tiempo, el cual se encargó de convertir la gran experiencia de aprender algo nuevo, en algo horrible.

4. Bibliografía

Gonzalez, R. (2022, marzo).”Proyecto semestral de laboratorio”. Recuperado 19 de abril de 2022, de https://docs.google.com/document/d/1tCVZBGScJbXFspQfu6WlDl3PsyCJUcqIgos5_DQz7k

Dore, M. (2021, febrero 15). The Dobble Algorithm. Medium. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, *104*, 616-621.

Jung, A. (2004). A short introduction to the Lambda Calculus. *School of Computer Science, The University of Birmingham*.

5. Anexos

5.1. Tabla de Funciones

TDA Cards		
Tipo de función	Función	Descripcion
Constructor	cartasVacias	Crea un Cards vacio
Pertenencia	cartasVacias?	Consulta si es un cards Vacio o no
Selector	getfirstCard	Obtiene la primera card de un cards
Selector	getAnotherCards	Obtiene todos los cards menos el primer card
Selector	getnCard	Obtiene el card en una posicion n dentro de un cards
Selector	largo	Obtiene el la cantidad de card dentro de un cards
TDA Area		
Tipo de función	Función	Descripcion
Selector	firstArea	Obtiene la primera card dentro del area
Selector	restoArea	Obtiene todos los cards menos el primer card del area
TDA Player		
Tipo de función	Función	Descripcion
Selector	getPlayerName	Obtiene el name de un player
Selector	getPlayerScore	Obtiene el score de un player
Selector	getPlayerCards	Obtiene las cards de un player
Modificador	setPlayerScore	Modifica el name de un player
Modificador	setPlayerCards	Modifica el score de un

		player
Modificador	addPlayerCard	Añade una card al cards de un player
TDA Players		
Tipo de función	Función	Descripcion
Selector	firstPlayer	Obtiene el primera player dentro de un players
Selector	anotherPlayers	Obtiene todos los players menos el primer player del area
Selector	cantidadPlayers	Obtiene el la cantidad de player dentro de un players
Selector	getnPlayer	Obtiene el player en la posicion n dentro de un players
TDA Game		
Tipo de función	Función	Descripcion
Selector	getArea	Funcion que obtiene el Area de un Game
Selector	getSetCartas	Funcion que obtiene el cardsSet de un Game
Selector	getTurn	Funcion que obtiene el turno de un Game
Selector	getMaxPlayers	Funcion que obtiene el Maximo de jugadores de un Game
Selector	getPlayers	Funcion que obtiene los jugadores (players) de un Game
Selector	getStatus	Funcion que obtiene el Estado de un Game
Selector	getMode	Funcion que obtiene el de un Game
Selector	getRandom	Funcion que obtiene el de un Game
Modificador	register	Funcion que registra un usuario, añade un player al

		players del game
Modificador	updateArea	Funcion que actualiza el area con un nuevo area
Modificador	updateTurn	Funcion que actualiza el turno con un nuevo turno
Modificador	updateSet	Funcion que actualiza el cardsSet con un nuevo cardsSet
Modificador	updateStatus	Funcion que actualiza el estado con un nuevo estado
Modificador	updatePlayers	Funcion que actualiza los players con un nuevo players
Otros	whoseTurnIsIt?	Funcion que devuelve el player al que le corresponde jugar segun el turno
	repartirCartas	Funcion que reparte cartas desde el cardsSet hacia los jugadores y el area
		en igual cantidad, sera usada en un modo de juego en especifico
Modo de juego	stackMode	Funcion que corresponde al primer modo de juego, siguiendo las reglas descritas
		anteriormente de este juego, para luego enviar a una accion
Modo de juego	empyHandsStackMode	Funcion que corresponde al segundo modo de juego, siguiendo las reglas descritas
		anteriormente de este juego, para luego enviar a una accion
Accion	pass	Funcion que se usa cuando el player desea pasar su turno
		cambia el turno correspondiente y

		devuelve las cartas al cardsSet del juego
Accion	spotIt	Esta es una funcion currificada, que segun el modo de juego, recibe un simbolo y lo
		busca en el area.
Accion	finish	Funcion que termina de manera prematura un juego
	play	Funcion de orden superior que solamente envia al modo de juego correspondiente
		con una accion ingresada, haciendo que estas internamente hagan el trabajo.
Otros	game->String	Funcion que muestra el juego
Extras		
Tipo de función	Función	DescripciónDescripcion
Pertenencia	es_primo?	Comprueba si es que un numero dado, es primo o no
Pertenencia	esta-en	Comprueba si una card esta dentro de un cards
Constructor	simbolos-hasta-n	Crea una lista de numeros que van desde 1 hasta n
Modificador	rand	Funcion que recibe una lista, una funcion que aleatoriza numeros, un numero inicial
		para esa funcion y va aleatorizando una lista, donde va aumentando el numero de uno
		en uno, y si el resultado de la funcion en ese numero es par, añade el numero a la lista,

		si no, da vuelta la lista y lo añade al principio de la lista, pero ahora la lista esta invertida
Modificador	cut	Funcion que recibe una lista y un numero y corta la lista en en ese numero
Otros	randomFn	Función que recibe un número y simula una aleatorización, solo simula debido a que
		parte importante de este paradigma impide que para una misma entrada haya 2 salidas
		diferentes