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

INFORME DE LABORATORIO 1:
SIMULACIÓN DE DOBBLE EN SCHEME

Nombre: Bastián Escribano Gómez
Profesor: Gonzalo Martínez Ramírez
Asignatura: Paradigmas de Programación

18 de abril de 2022

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

Tabla de Contenidos

1. INTRODUCCIÓN

La programación y el escribir código es solo dar instrucciones a una máquina la cual está hecha para realizar lo que se le pide. Para muchos escribir código es un arte, sin embargo escribir código no es más que escribir instrucciones. El arte está en la manera en la cual se plantea una solución y las diferentes formas de abordar un problema buscando los caminos los cuales sean más eficientes y que mejor se adapten a la solución. La forma en la cual puede desmembrar una solución depende de cada persona, pero en el mundo de la programación existen paradigmas de la programación, los cuales ofrecen distintas formas de abordar un problema y plantear una solución. El paradigma funcional, donde se utiliza el lenguaje de programación Scheme a través del compilador Dr. Racket. se basa a través de funciones, esto nace cerca del 1960 cuando nacía el cálculo lambda. Este informe está estructurado de la siguiente forma: Primero, se describe brevemente el problema, luego 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 finalmente una conclusión al proyecto realizado.

1.1 DESCRIPCIÓN DEL PROBLEMA

Se pide desarrollar una simulación de Dobble, El juego **Dobble** y en particular la generación de las piezas se rige por propiedades matemáticas dentro de los cuales destacan los números primos. Para implementar esto, se deben tener algunos elementos en cuenta:

cardsSet: En base a parámetros que se ingresen, se debe generar un mazo de cartas para que sea un juego válido. El juego Dobble es un juego matemático y matemáticamente tiene que estar bien construido para que tenga sentido.

Game: Se necesitan jugadores, modos de juegos y un mazo para empezar un juego.

Operaciones: Son las funciones que registran usuarios y verifican los turnos, la creación de cartas y los diferentes modos de juego.

Todo debe ser implementado utilizando el Paradigma Funcional, a través de la programación en Scheme y el compilador Dr. Racket.

1.2 DESCRIPCIÓN DEL PARADIGMA

El Paradigma Funcional, tal como su nombre lo dice, tiene como unidad de abstracción y programación básica a las funciones.

Una función corresponde a una transformación de elementos, donde se tiene al dominio, que son los elementos de entrada y que serán transformados por el proceso de la función y el recorrido, que es el elemento de salida de la función una vez se ha realizado la transformación.

El paradigma solo trabaja con funciones, por lo que no existe el concepto de variable actualizable en este paradigma. Específicamente en este caso de este trabajo, se hace uso del lenguaje de programación funcional Scheme, aunque cabe destacar de que este no es un lenguaje puro de programación funcional. Un lenguaje puro de programación funcional

En el Paradigma Funcional se tienen algunos elementos bastante importantes a la hora de construir un código utilizando este paradigma, como, por ejemplo:

Funciones anónimas: Provienen directamente del Cálculo Lambda, donde estas funciones se expresan sin nombre, bajo una entrada y una transformación de esta que se aplica a la función. Puede ser utilizadas sin necesidad de definirles un nombre. Un ejemplo es la utilización de funciones anónimas en funciones de filtro o funciones map.

Composición de funciones: Consiste en una operación donde dos funciones generan una tercera función, algo bastante útil cuando se tienen funciones que se complementan entre sí.

Recursividad: Es fundamental en este paradigma, pues permite realizar una gran cantidad de operaciones y procesos donde este se van utilizando soluciones pequeñas del problema. Existen tres tipos de recursión en este Paradigma: recursión natural, recursión de Cola y recursión Arbolea, pero para este proyecto se utilizan los primeros dos tipos.

Funciones de orden superior: Corresponden a aquellas funciones que reciben como dominio a una función o bien dan como resultado a otra función. El mejor ejemplo corresponde a la composición de funciones.

Currificación: Es la evaluación de una función de “n” argumentos, transformada a la evaluación de una secuencia de funciones de un argumento. En la currificación se ocupan funciones anónimas bajo el concepto de funciones de orden superior.

1.3 OBJETIVOS

El objetivo principal del proyecto es aprender sobre el paradigma y la programación funcional, para así obtener la habilidad de programar de otra forma distinta a la que se tiene costumbre actualmente. En el mundo laboral existen miles de ofertas con diferentes lenguajes de programación, sin embargo bajo cada lenguaje existe un limitado número de paradigmas, esto está relacionado directamente con la manera de programar y cómo programar. Un buen desarrollador será capaz de adaptarse a cualquier lenguaje de programación con los previos conocimientos del paradigma en el cual se maneja el lenguaje en cuestión. Además se pretende adquirir habilidad respecto a la programación en Scheme y aprender a utilizar las herramientas de Dr. Racket para completar el proyecto de laboratorio y así poder tener una base para los futuros laboratorios y otros proyectos que en un futuro se desarrollen con la Programación Funcional.

2. DESARROLLO

2.1 ANALISIS DEL PROBLEMA

Matemáticamente, es posible generar un mazo con cartas que contengan una cierta cantidad de símbolos y que se repita por lo menos 1 símbolos en cada carta diferente del mazo, a esto lo llamaremos **el factor de repetición**. Esto es posible a una conexión matemática que se basa en números primos y en potencia de números primos. Este factor es el más importante y su nombre es **orden**, por lo tanto, el orden debe ser necesariamente un número primo o una potencia de un número primo para que se cumpla que exista al menos una repetición por carta.

Por ejemplo, el número 2 es un número primo, por lo tanto el orden puede ser 2. Ahora, a través de un cálculo matemático es posible determinar la cantidad máxima de cartas que pueden ser generadas en base a ese orden y la cantidad de símbolos que debe tener el mazo para que se cumpla el factor de repetición.

$$cardsSet(getOrden) = getOrden^2 + getOrden + 1$$

El cardsSet será la cantidad de cartas asociadas al orden y también la cantidad de símbolos que deberá tener el cardsSet. Por lo tanto, nuestro mazo estará compuesto de

$$cardsSet(2) = 2^2 + 2 + 1 = 7$$

La cantidad de símbolos por carta está determinada por

$$cantSímbolos(getOrden) = \ getOrden + 1$$

Sin embargo, es posible determinar la cantidad de cartas y símbolos totales a través de otra manera matemática

$$cardsSet(getSímbolosPorCarta) = getSímbolosPorCarta^2 - getSímbolosPorCarta + 1$$

De ambas formas será posible encontrar la cantidad total de cartas y símbolos

$$cardsSet(3) = 3^2 - 3 + 1 = 7$$

Como se habla de una cantidad máxima de símbolos necesariamente por carta, no se debe descuidar el factor de repetición para que exista la conexión entre carta y carta.

En la figura 1. Es posible observar tras el ejemplo inicial el patrón de repetición que podría tener el mazo con cartas que contengan 3 símbolos **diferentes**.

	#1	#2	#3	#4	#5	#6	#7
A	1	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	1	0	1	0

Además de la generación del mazo, se debe generar el juego, para esto se deberán más funcionalidades dentro de la generación del mazo y del propio juego.

- **CardsSet:** Corresponde a un conjunto de asociaciones, donde un usuario genera un mazo de juego a partir de una serie de parámetros a definir. Posteriormente el cardsSet será útil para un juego.
- **game:** Corresponden a aquellos que interactúan con el juego, registrándose en esta para luego poder realizar las operaciones disponibles. Por ejemplo, iniciar un juego.

Para poder implementar a los elementos mencionados anteriormente en Scheme, se pueden representar de la siguiente manera:

- **cardsSet:** Una lista que contiene una lista de símbolos, el número de símbolos, la cantidad de cartas (que en caso de que sea negativa, se debe generar la cantidad máxima posible) y una función que sea capaz de generar un número aleatorio. (Elements (list) X numE(int) X maxC(int) X randomFn (fn))
- **game:** Una lista que contiene una lista que contiene el número de jugadores, un cardsSet, el modo de juego y una función que sea capaz de generar un número aleatorio (numPlayers (int) X cardsSet (cardsSet) X mode (fn) X randomFn (fn)

Cada **TDA** (Tipo de Dato Abstracto) tiene un archivo único, donde solo se implemente lo necesario para cada TDA y que el archivo principal contenga las funciones obligatorias y opcionales, donde en este se llamen a las funciones de los TDAs creados. La forma de exportar e implementar funciones desde otros TDA es por medio de la función **require** y la función **provide**.

2.2 DISEÑO DE LA SOLUCIÓN

Para diseñar la solución, no solo es necesario utilizar los tipos de datos nativos de Scheme, sino que también se deben crear tipos de datos específicos, a través de lo que se conoce como TDA, utilizando la siguiente estructura: Representación, Constructores, Funciones de Pertenencia, Selectores, Modificadores y otras funciones asociadas al TDA.

Los TDAs creados son utilizados para la construcción de las operaciones de otros TDAs y para las operaciones obligatorias y opcionales. Los TDAs más importantes son:

2.2.1.1 TDA cardsSet

- Representación: Cada vez que se haga ingreso de los parámetros y la función respectivamente, se hará entrega de un mazo de cartas listo para iniciar un juego. Este será construido a través de los elementos que reciba la función constructora del cardsSet.
- Constructor: Dado una lista de elementos, el número de símbolos por carta, la cantidad máxima de cartas y una función que genera un número aleatorio, se crea una lista que contiene una lista con símbolos correspondientes a un set de cartas válido.
- Función de Pertenencia, Selectores, Modificadores y Otras Funciones: *ver tabla 1 en anexos, P. 11*

2.2.1.2 TDA game

- Representación: Para iniciar un juego se necesita la cantidad de jugadores, un mazo

2.2.2 OPERACIONES OBLIGATORIAS

numCards: Se implementa una función que determina la cantidad de cartas en el cardsSet. Para esto se hace uso de la función length de una lista.

nthCard: Se implementa una función con recursividad de cola, que obtiene la n-ésima (nth) carta desde el conjunto de cartas partiendo desde 0 hasta (totalCartas-1).

findTotalCards: Se implementa una función que a partir de una carta de muestra, determina la cantidad total de cartas que se deben producir para construir un conjunto válido. Para esto es necesario completar la función nthCard, ya que se debe indicar la carta número n a evaluar.

requiredElements: Se implementa una función que a partir de una carta de muestra, determina la cantidad total de elementos necesarios para construir un conjunto válido. Como se sabe que la cantidad de elementos (o símbolos) y la cantidad de cartas posibles se obtiene desde una misma fórmula matemática, se vuelve a utilizar la fórmula para determinar la cantidad de símbolos necesarios para construir un conjunto válido.

missingCards: Se implementa una función que a partir de un conjunto de cartas retorna el conjunto de cartas que hacen falta para que el set. Esta función utiliza la función **map** la cual devuelva una nueva lista como los elementos de la lista inicial modificados.

cardsSet->string: Se implementa una función recursiva natural que convierte un conjunto de cartas a una representación basada en strings que pueda visualizarse a través de la función display. Para esto se realiza un cambio de tipos de datos y transforma todo a un string. Posteriormente con la función display se puede observar los símbolos o elementos de forma ordenada.

stackMode: Se implementa una función que permite retirar y voltear las dos cartas superiores del mazo de cartas en el juego y las dispone en el área de juego. El área de juego es una lista vacía en la cual van ingresando nuevas cartas que se vayan poniendo en juego.

register: Se implementa una función para registrar a un jugador en un juego. Los jugadores tienen un nombre único y no puede exceder la cantidad de jugadores registrados. Para esto se agrega el nombre de registro a una lista vacía y se suma un jugador a la lista de jugadores.

whoseTurnIsIt?: Función que retorna el usuario a quién le corresponde jugar en el turno. Para esto, se busca el último jugador que ingreso a la lista de jugadores, en caso de que coincidan, se retornará una lista con el nombre del jugador a quien le corresponda el turno y además el modo de juego en el cual se encuentra el juego.

status: Se implementa una función que devuelve el estado actual del juego, se devolverá una lista con todos los elementos actuales del juego mediante funciones selectoras.

score: Se implementa una función que retorna el puntaje de un jugador a partir de su nombre de usuario. Para esto se implementa la función map con la función anónima y además con la lista de jugadores que se obtiene mediante la función selectora y el

cartas en el set tienen más de una figura en común. Si una violación ocurre, se devuelve el conjunto de entrada. Para esto se calcula el largo de la carta ingresada, si no coincide la carta no puede entrar, además, se realizará la comprobación si el conjunto es válido o no, en caso de no ser un conjunto válido, no sucederá y se devolverá el conjunto anterior.

emptyHandsStackMode: Se implementa una función que permite retirar y voltear la carta en el tope del stack y una de las cartas del usuario seleccionada de manera aleatoria a partir de la función de aleatorización registrada en el juego. Al terminar un turno, si el jugador acierta se ubican en la base del stack, de lo contrario la carta del jugador vuelva a sus cartas y la del stack vuelve a la base de éste. Para esto se utiliza la función randomFn que recibirá dos parámetros para seleccionar el número aleatorio. Recibirá un número 0 y el máximo de cartas que existen en el mazo, para esto se calculará el número aleatorio entre 0 y el **length del cardsSet.** Posteriormente se agregarán al área de juego setCardsLeft.

reverseStackMode: Se implementa una función que toma las dos primeras cartas del mazo. Modo de juego similar al modo stackMode. stackMode toma las primeras dos cartas del tope y las da vuelta hace lo contrario y toma las dos primeras cartas del mazo y las da vuelta. Posteriormente se agregan ambas cartas al área de juego.

OPERACIONES OPCIONALES

setRandom: Se implementa una función que selecciona aleatoriamente un número entre dos números, esto sirve seleccionar un número aleatorio desde un número 0 hasta un número n que podría ser el length del cardsSet para seleccionar una carta aleatoria en el cardsSet.

cards: Se implementa una función recursiva de cola que contiene otras funciones de orden superior y además currificación. Esta función se encarga de simular un ciclo iterativo anidado para generar un mazo de cartas correctamente. El motivo de simulación de este ciclo for, es porque existe un algoritmo que es capaz de calcular y determinar qué símbolo corresponde a cada carta, sin embargo la forma en la cual se emplea es por medio de ciclos iterativos. Se utiliza una lista vacía para ir agregando cartas según se van completando. La generación de cada carta dependerá del cálculo del orden en el que se encuentre.

getLastPlayer: Se implementa una función que por medio de la función reverse, car y el selector de los jugadores para seleccionar y devolver el último jugador registrado del juego.

2.3 ASPECTOS DE IMPLEMENTACIÓN

2.3.1 COMPILADOR

Para este proyecto es necesario el compilador Dr. Racket, específicamente de versión 6.11 o superior. Dr Racket tiene bastantes herramientas utiles, como por ejemplo debug, que permite revisar pasó a pasó el cambio del Stack del código y a través de esto encontrar errores. Se pueden utilizar todo tipo de funciones de Scheme y Racket.

La implementación se basa en el trabajo con listas, sin embargo, se espera que no se utilicen funciones como CAR o CDR fuera de la elaboración de TDAs, ya que estas son funciones del TDA Lista, por lo que se deben encapsular dentro de los TDAs creados.

2.3.2 ESTRUCTURA DEL CODIGO

2.4 INSTRUCCIONES DE USO

2.4.1 EJEMPLOS DE USO

Primero, se debe verificar que se tengan ambos archivos en una misma carpeta, ya que, de lo contrario, el archivo game.rkt no se podrá ejecutar correctamente si falta un TDA. Luego de eso, se puede compilar, ejecutando el programa a través de la opción “run”.

Una vez ejecutado, en el archivo game.rkt se tienen varios ejemplos para cada una de las operaciones. Para que estos ejemplos puedan funcionar correctamente, se requiere tener creado un cardsSet (o puede ser creado en el momento)

Cada operación respectivamente a cada TDA tendrá al menos 3 ejemplos en la parte superior de la operación.

2.4.2 RESULTADOS ESPERADOS

Se espera crear una simulación de Dobble correctamente, que funcione sin errores ni ambigüedades y que conserve los fundamentos del paradigma funcional.

Junto con lo anterior, se espera trabajar correctamente con listas y recursión, ya que son fundamentales para el desarrollo del proyecto. Por último, se espera que cada función haga su procedimiento correctamente y que los TDAs tengan representaciones correctas.

2.4.3 POSIBLES ERRORES

Si bien todas las funciones de cada TDA están bien documentadas y existen ejemplos, no todas las funciones entregan resultados esperados como se encuentra en el apartado de ejemplos del enunciado. Sin embargo existe una definición correcta de dominios, recorridos, descripciones y ejemplos, los cuales podrían llevar por buen camino a la función.

2.5 RESULTADOS Y AUTOEVALUACIÓN

2.5.1 RESULTADOS OBTENIDOS

Los resultados obtenidos fueron los esperados, ya que se logró crear la gran mayoría de las funciones, sin embargo como se menciona anteriormente, algunas funciones no entregan resultados como se espera que se realice en los ejemplos. Las funciones funcionan correctamente, entregan resultados coherentes, correctos y no tienen errores de sintaxis ni funcionales. Se realizaron múltiples pruebas con distintos ejemplos que están en el encabezado de la definición de cada función.

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.

Para ver la tabla de autoevaluación, *ver la tabla 3 en anexos, P.13.*

3. CONCLUSIÓN

Tras realizar y finalizar el proyecto, se puede concluir que se cumplió el principal objetivo de este proyecto e informe era introducir y comprender el paradigma funcional el cual se ha cumplido correctamente.

El paradigma funcional es complejo para quien no haya salido del paradigma imperativo u orientado a objetos, sobre todo en un principio, entender la idea de que absolutamente todo es una función, es complicado. Sin embargo, durante de este proyecto se ha presentado un entendimiento exponencial del paradigma, utilizando funciones de orden superior, currificadas, recursión y otras características propias del paradigma funcional. Si bien no se logró completar los requerimientos en su totalidad, se completó la gran mayoría y las funciones que no se completaron quedaron con avance lo cual significa que la idea está y que podría implementarse.

4. BIBLIOGRAFÍA Y REFERENCIAS

1. Gonzales, R. (2022). “Proyecto Semestral de Laboratorio”. Paradigmas de Programación. Enunciado de Proyecto Online. Recuperado de: https://docs.google.com/document/d/1tCVZBGScJbXFspQfu6WlDl3PsyCJUcqIgos5__DQz7k/edit
2. Gonzales, R. (2022). “3 – P. Funcional”. Paradigmas de Programación. Material de clases Online. Recuperado de: https://uvirtual.usach.cl/moodle/course/view.php?id=10036&section=11
3. Multiples Autores. (Febrero 1998). “Report on the Algorithmic Language Scheme”. Informe / Estudio Online. Recuperado de: https://uvirtual.usach.cl/moodle/pluginfile.php/369069/mod_resource/content/2/r5rs.pdf
4. Flatt, M. y Bruce, R. (2021). “The Racket Guide’’. The Racket Reference. Documentación Online. Recuperado de: https://docs.racket-lang.org/guide/
5. Marcus Heemstra, (2014). “The Mathematics of Spot It” https://openprairie.sdstate.edu/cgi/viewcontent.cgi?article=1016&context=jur

5. ANEXOS

Tabla N°1: Funciones del TDA cardsSet.

Tipo de función	Nombre	Descripción
Función de Pertenencia	dobble?	Verifica si el cardsSet es un cardsSet válido.
Selector	getElements	Obtiene los elementos de la lista elements del cardsSet.
Selector	getNumE	Obtiene el número de elementos del cardsSet.
Selector	getRandom	Obtiene el número aleatorio del cardsSet.
Selector	getOrderGame	Obtiene el orden del cardsSet.
Modificador	setElements	Obtiene un nuevo cardsSet con los elementos modificados.
Modificador	setNumE	Obtiene un nuevo cardsSet con el número de elementos modificado.
Modificador	setMaxC	Obtiene un nuevo cardsSet con el número de cartas modificado.
Modificador	setRandom	Obtiene un nuevo cardsSet con el número aleatorio modificado.
Otras Funciones	cards	Función que genera un mazo de tipo card.
Otras Funciones	randomFn	Función que genera un número aleatorio.
Otras Funciones	numCards	Función que permite determinar la cantidad de cartas en el set.
Otras Funciones	nthCard	Función que obtiene la n-ésima (nth) carta.
Otras Funciones	findTotalCards	Función que a partir de una carta de muestra, determina la cantidad total de cartas que se deben producir para construir un conjunto válido.
Otras Funciones	requiredElements	Función que a partir de una carta de muestra, determina la cantidad total de elementos necesarios para poder construir un conjunto válido.
Otras Funciones	missingCards	A partir de un conjunto de cartas retorna el conjunto de cartas que hacen falta para que el set sea válido.
Función de Pertenencia	prime?	Función que evalua un número y ve si es primo o no.
Otras Funciones	cardsSet->string	Función que convierte un conjunto de cartas a una representación basada en strings que posteriormente pueda visualizarse a través de la función display.

Tabla N°2: Funciones del TDA game.

Tipo de función	Nombre	Descripción
Función de Pertenencia	isGame?	Verifica si el juego es un juego válido.
Selector	getCardsSet	Obtiene el cardsSet asignado a game.
Selector	getCardsLeft	Obtiene el área de juego.
Selector	getPlayers	Obtiene una lista con los jugadores.
Selector	getModeGame	Obtiene la función del modo de juego.
Selector	getRandomFn	Obtiene un número aleatorio designado al juego.
Selector	getLastPlayer	Obtiene el último jugador registrado en el juego.
Modificador	setCardsSet	Obtiene un nuevo juego con el cardsSet modificado.
Modificador	setCardsLeft	Obtiene un nuevo juego con el área de juego modificado.
Modificador	setPlayers	Obtiene un nuevo juego con la lista de jugadores modificado.
Modificador	setNumPlayers	Obtiene un nuevo juego con el número de jugadores modificado.
Modificador	setMode	Obtiene un nuevo juego con el modo de juego modificado.
Modificador	setRandom	Obtiene un nuevo juego con el número aleatorio modificado.
Otras Funciones	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.
Otras Funciones	register	Función para registrar a un jugador en un juego.
Otras Funciones	play	Función que permite realizar una jugada a partir de la acción especificadas.
Otras Funciones	status	Función que retorna el estado actual del juego.
Otras Funciones	score	Función que retorna el puntaje de un jugador.
Otras Funciones	game->string	Función que convierte un juego/partida para que pueda visualizarse a través de la función display.
Otras Funciones	addCard	Función que permite agregar cartas a un set de manera manual.
Otras Funciones	emptyHandsStackMode	Función que permite retirar y voltear la carta en el tope del stack y una de las cartas del usuario seleccionada de manera aleatoria a partir de la función de aleatorización registrada en el juego.
Otras Funciones	reverseStackMode	Toma las dos primeras cartas del mazo y las dispone en el área de juego.

Tabla N°3: Autoevaluación de los requerimientos funcionales

Requerimientos Funcionales	Evaluación
TDAs	1
numCards	1
nthCard	1
findTotalCards	1
requiredElements	1
missingCards	0.5
cardsSet->string	1
stackMode	1
register	1
whoseTurnIsIt?	0.5
play	0.25
status	1
score	0.5
game->string	0.25
addCard	0.75
emptyHandsStackMode	1
reverseStackMode	1