

Informe de Laboratorio

1: Implementación de juego «Dobble» en Racket



PARADIGMAS DE PROGRAMACIÓN

18/04/2022



Índice general

1	Introducción	3
1.1	Descripción del Problema	3
1.2	Descripción del Paradigma	3
2	Desarrollo	5
2.1	Análisis del Problema	5
2.2	Diseño de la Solución	6
2.2.1	TDA CardsSet	6
2.2.2	TDA GAME	7
2.3	Aspectos de Implementación	8
2.4	Instrucciones de uso:	9
2.4.1	Resultados Esperados	10
2.4.2	Posibles errores	10
2.5	Resultados y autoevaluación	10
3	Conclusiones	11
4	Referencias	12
5	Anexos	13
5.1	Anexo 1: Algoritmo generación de cartas	13
5.2	Anexo 2: Tabla de Pruebas y autoevaluación	14

CAPÍTULO Introducción 1

Este informe corresponde al 1er laboratorio de la asignatura *Paradigmas de Programación*. En este primer laboratorio se empleará el Paradigma Funcional, usando el lenguaje de programación **Racket**, un descendiente del lenguaje **Scheme**. Este se usará a través de el IDE [Integrated Development Environment] DrRacket.

1.1. Descripción del Problema

Se pide crear una implementación de Dobble. Este juego consiste en un mazo de cartas con 55 cartas (57 originalmente) donde dos cartas cualesquiera comparten **solo** un elemento entre si. Para jugar Dobble existen distintos tipos modos. El modo mas conocido consiste en que los jugadores deben tomar dos cartas del mazo y el primero que identifique cual símbolo se encuentre en ambas cartas toma la primera carta y la guarda. Después, se toma otra carta y se repite el proceso hasta que se acabe el mazo. Una vez que se acaben las cartas, el jugador que tenga la mayor cantidad de cartas ganará. Para representar esto en Racket hay que tener en cuenta que la generación de cartas deberá respetar la condición de que solo se repita el símbolo una vez entre cualquier par de cartas. Además hay que tomar en cuenta que tipo de modo de juego seleccionara el usuario.

1.2. Descripción del Paradigma

Para crear la representación de «Dobble» se usará la Programación Funcional, Un paradigma de la programación con un gran enfoque en las funciones, evitando datos mutables. Las funciones se podrán usar como argumento y estas podrán regresar otra función (funciones de alto nivel). Este paradigma esta basado sobre el cálculo lambda, el cual sigue una simple regla de sustituir variables y un esquema simple para definir las funciones. Las principales características de este paradigma son:

- **Funciones Puras:** Siempre producirán la misma salida con los mismos argumentos sin importar otros factores
- **Recursión:** No se hace uso de ciclos **for** o ciclos **while**, y en cambio se hará uso de la recursión, donde se llamara la función una y otra vez hasta que se llegue a un caso base.
- **Funciones son de primera clase:** En el paradigma funcional, las funciones se podrán usar como argumentos y se podrán regresar desde otras funciones. A las

funciones que toman otras funciones como argumentos y/o regresen otras funciones serán llamadas **funciones de alto nivel**.

- **Inmutabilidad:** Las variables serán inmutables, osea que no se podrán modificar una vez sean creadas.

Estas características tendrán sus ventajas y desventajas. Las ventajas de estas son que harán que las funciones puras sean mas fácil de entender ya que no cambiaran de estado y solo dependerán de la entrada que reciban. Pero tendrán la desventaja que podrían requerir de mas rendimiento en caso de mal utilizar la recursión. Además, usualmente se esta acostumbrado a usar loops para programar, lo cual hará pasar a solamente usar recursión algo difícil.



CAPÍTULO Desarrollo 2

2.1. Análisis del Problema

Para este proyecto se tendrá que tener en cuenta los aspectos fundamentales del juego «Dobble». Este juego, como ya ha sido mencionado, tendrá un mazo de cartas con 57 cartas, donde 2 cartas cualesquiera tendrán siempre un elemento en común. Para este hay que tener en cuenta las bases matemáticas del juego, pero en este caso se nos entrega un algoritmo escrito en **Javascript** el cual facilitara la creación del algoritmo en **Racket**. El mazo creado sera representado como una lista de listas, donde cada lista sera una carta diferente. El usuario podrá pedir diferentes acciones respecto al mazo o a una carta [TDA cardsSet]:

- Verificar si un mazo es valido [dobble?]
- Ver el numero de cartas en el mazo [numCards]
- Buscar que carta hay en x posición [nthCard]
- Encontrar cuantas cartas necesita para un mazo valido a partir de una carta [findTotalCards]
- Encontrar el numero de elementos que necesita para armar un mazo valido a partir de una carta [requiredElemets]
- Ver que cartas faltan de un mazo para armar uno valido [missingCards]
- Transformar el mazo a una representación en strings. [cardsSet→string]

Para permitir al usuario poder jugar el juego hay que tener en cuenta que este podrá [TDA game]:

- Ingresar cantidad de jugadores [register y numPlayers]
- Elegir modo de juego [mode, stackMode, myMode y emptyHandsStackMode]
- Elegir que hacer en su turno [play y pass]
- Terminar el juego cuando quiera
- Ver de quien es el turno y el puntaje [whoseTurnIsIt? y status]

2.2. Diseño de la Solución

Para la solución de este problema, se tienen en cuenta todas las acciones que el usuario podrá tomar, ya sea para armar el mazo o para las acciones que podrá tomar al jugar el juego una vez ya tenga el mazo armado.

2.2.1. TDA CardsSet

Creación de cartas (Constructor cardsSet)

El mazo de cartas creado sera representado como una lista de listas, y cada elemento de la carta podrá ser o un numero, o un símbolo introducido por el usuario. El principal problema sera la creación de cartas, para ello se nos entrega el siguiente algoritmo escrito en Javascript. (Ver Anexo 1)

Debido a la diferencias del lenguaje y requerimientos del proyecto, nuestra solución no podrá hacer uso de variables ni de ciclos `while` o `for`, por lo que se necesitara hacer uso de recursión para poder implementarlo. En el algoritmo se puede observar que este posee ciclos `for` anidados, por lo que en la solución crearemos funciones recursivas que llamaran a otras funciones recursivas las cuales generaran el numero requerido para las cartas y los agregaran a una lista. Con las funciones que generaran las cartas creadas, hace falta dejarlas en una sola lista, lo cual se puede lograr con la función de Racket `append`. Ahora se obtiene un mazo de cartas, pero este esta representando por numero y hay que tener en cuenta que el usuario podrá pedir en la generación de cartas que este se genere con una lista de símbolos ingresada por el usuario (Elements), un numero de elementos en cada carta (numE), un máximo de cartas (maxC) y una función que aleatorizara el ordenamiento de las cartas (rndFn). Para generar el mazo de cartas con los símbolos, se tomara generara el mazo y se remplazara el símbolo que este en la posición `i` y se remplazaran todos los `i` que hayan en el mazo con ese símbolo y se repetirá el proceso hasta que no queden mas símbolos o mas cartas. El numero de elementos sera entregado al algoritmo para que los genere y después se separara la lista según la cantidad de elementos en cada carta. Y para el máximo de cartas generado solo bastara con tomar los maxC elementos de la lista. Y finalmente con todo ya listo, se procede a aleatorizar el ordenamiento de las cartas con la función proporcionada por el enunciado.

Pertenencia

Como función de pertenencia tendremos que verificar que el mazo de Dobble entregado por el usuario sea un mazo valido para jugar. Para ello tendremos que verificar que la



cantidad de cartas es la correcta, lo cual se puede verificar con la formula $(n * (n - 1) + 1)$, donde n es el numero de elementos. Además habrá que asegurarnos que los símbolos de las cartas solo se repitan una vez en cada par de cartas.

Selectores

Nuestra función de selector sera la función `nthCard`, la cual nos permitirá encontrar la carta n (elegida por el usuario). Como nuestro mazo esta representado por una lista de lista solo basta con hacer uso de `list-ref`, una función nativa de Racket.

Modificadores

La función modificadora en este TDA sera la función `cardsSet->string`, la cual nos permitirá representar el mazo (lista de listas) como strings.

Otras Funciones

Se tienen otras funciones como `numCards`, `findTotalCards`, `requiredElements` y `missingCards`. En las tres primeras funciones bastara con usar la funcion `length` y para la ultima función, se hará una intersección entre el mazo ingresado y un mazo generado (valido), donde el resultado serán las cartas faltantes, donde el resultado serán las cartas faltantes.

2.2.2. TDA game

El TDA game permitirá al usuario jugar con el set de cartas anteriormente creado. El usuario podrá registrar usuarios y elegir un modo de juego.

Constructor

La función constructora para el TDA game consistirá en una función que podrá recibir como argumentos un Entero, una lista de listas (Mazo), una función (modo de juego) y una función de aleatorizacion (`rndFN`). Esta función devolverá una lista de listas representando todo lo ingresado.



Selectores

Se tendrán varias funciones selectoras, ya que hay que poder obtener de quien es el turno actual (`whoseTurnIsIt?`), obtener el status del juego (`status`) y conseguir el puntaje de cada jugador (`score`).

Modificadores

Se hará uso de este tipo de función para poder registrar a los usuarios que irán a jugar el juego (`register`), para poder mostrar el juego en forma de string (`game->string`) y para que el usuario pueda agregar manualmente una carta (`addCard`)

Otras Funciones

El usuario podrá elegir el modo de juego que quiera usar, ya sea `stackMode`, `emptyHandsStackMode` o una modalidad creada por uno mismo (`myMode`). Además podrá accionar en el juego con la función `play`, y podrá elegir si pasar su turno o seleccionar que simbolo se repite en las dos cartas mostradas.

2.3. Aspectos de Implementación

- Estructura del proyecto: Cada TDA ira en su propio archivo y esto proveerán las funciones necesarias a un archivo principal `Dobble.rkt`, en el cual se podrá ejecutar el código a través de DrRacket y el usuario podrá hacer uso de las funciones de cada TDA.
- Bibliotecas utilizadas: Para este proyecto solo se hizo uso de la librería base de Racket
- Interprete utilizado: Se hizo uso de DrRacket v8.4
- Razones de elección: Se separo en tres archivos para tener un mayor nivel de organización, ya que cada TDA requirió uso de funciones que solo se usarían ahí. Se hizo uso de la biblioteca base debido a que no se requirió tener que hacer uso especifico de otra y el uso de DrRacket fue debido a que el entorno que este tiene esta muy bien integrado con Racket, permitiendo saber fácilmente donde falló el programa, si es que se acabo la memoria (usualmente ocurre por mal uso de la recursión) y debugear fácilmente una parte del código.



2.4. Instrucciones de uso:

- cardsSet:

```
(cardsSet 0 7 -1 randomFn) ;; Genera mazo con numeros, 7 simbolos por carta y todas  
↪ las cartas posibles  
(cardsSet (list "A" "B" "C" "D" "F" "G" "H" "I" "J" "K" "L" "M" "N") 4 10 randomFn) ;;  
↪ Genera mazo con simbolos, 4 elementos por carta y maximo 10 cartas
```

- dobbble?:

```
(dobbble? (cardsSet 0 4 -1 randomFn))
```

- nthCard:

```
(nthCard (cardsSet 0 7 -1 randomFn) 34)
```

- numCards:

```
(numCards (cardsSet 0 7 -1 randomFn))
```

- findTotalCards:

```
(findTotalCards (list "A" "B" "C" "D"))
```

- requiredElements:

```
(requiredElements (list "A" "B" "C" "D"))
```

- missingCards:

```
(missingCards (list(list 1 2 3 4) (list 1 7 6 5)))
```

- cardsSet→string:

```
(cardsSet->string(cardsSet (list "A" "B" "C" "D" "F" "G" "H" "I" "J" "K" "L" "M" "N")  
↪ 3 10 randomFn))
```



2.4.1. Resultados Esperados

Se espera que el mazo se genere sin errores dado que el conjunto de elementos ingresados sea valido.

2.4.2. Posibles errores

Es posible que `missingCards` no genere bien las cartas faltantes si es que se ingresan símbolos.

2.5. Resultados y autoevaluación

Probando el programa creado con los ejemplos demostrados, se puede ver que todo el TDA `cardsSet` funciona correctamente y que el TDA `game` solamente funciona parcialmente debido a que solamente unas pocas funciones de este TDA fueron implementadas. La autoevaluación (Ver Anexo 1) se evaluara de la siguiente forma:

1. Nombre de la función
2. Tipo de Prueba
3. Razón de fallos
4. Completado?
5. Evaluación

La columna de evaluación sera evaluada según lo siguiente:

1. 0: No realizado.
2. 0.25: Implementación con problemas mayores (funciona 25 % de las veces o no funciona)
3. 0.5: Implementación con funcionamiento irregular (funciona 50 % de las veces)
4. 0.75: Implementación con problemas menores (funciona 75 % de las veces)
5. 1: Implementación completa sin problemas (funciona 100 % de las veces)

La función `missingCards` tiene 0.5 de puntaje debido a que funciona solamente cuando se entrega un mazo donde los elementos sean un número entero. El resto de las funciones del TDA `game` tienen 0 puntaje debido a que no fueron implementadas.





CAPÍTULO

Conclusiones 3

El Paradigma Funcional en Racket al principio fue difícil de entender y aplicar, debido a que uno se encuentra acostumbrado a la sintaxis de otros lenguajes como `python` y `javascript`, las cuales son mas parecidas al lenguaje humano, además que estos usualmente hacen uso de ciclos `for` o `while` y el cambio de pasar de estos a solamente tener que usar recursión y no usar variables hizo difícil el comienzo del proyecto. Otra complicación fue que el TDA game no fue completado al 100% debido a complicaciones de tiempo. Aun con estas complicaciones, este proyecto fue de mucha utilidad ya que permitio aprender sobre un diferente paradigma de programación y hacer uso de tecnologías como git.

CAPÍTULO

Referencias 4

1. The Racket Guide. (s. f.). Racket Documentation. Recuperado 2022, de <https://docs.racket-lang.org/guide/index.html>
2. González, R. (2022). Paradigmas de Programación - Proyecto semestral de laboratorio. Recuperado 2022, de 

3. Dore, M. (2021, 29 diciembre). The maths behind Dobble - Micky Dore. Medium. Recuperado 2022, de <https://mickydore.medium.com/dobble-theory-and-implementation-ff21ddb5318>
4. Dore, M. (2021, diciembre 30). The Dobble Algorithm - Micky Dore. Medium. <https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52>

5

5.1. Anexo 1: Algoritmo generación de cartas

```
//Generacion Dobble Javascript
//
let i, j, k
let n = 3 //order of plane, must be a prime number
let numOfSymbols = n + 1 //order of plane + 1let cards = [] //the deck of cards
let card = []; //the current card we are building//to start, we build the first
↪ card
let cards = [];

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)
    }
}
```

5.2. Anexo 2: Tabla de Pruebas y autoevaluación

Función	Tipo de prueba	Razón de fallos	Completado?	Evaluación
cardsSet	Generación de cartas con distintos argumentos.	No hubo fallos	Si	1
dobble?	Se prueba con mazo con símbolos y otro con números	Funciona correctamente	No	1
nthCard	Se busca la carta con distintos mazos	No hubo fallos	Si	1
numCards	Se busca las cartas totales con distintos mazos	No hubo fallos	Si	1
findTotalCards	Se busca la carta con distintas cartas	No hubo fallos	Si	1
requiredElements	Se busca los elementos con distintas cartas	No hubo fallos	Si	1
missingCards	Se busca las cartas faltantes con distintos mazos	Hubo fallos cuando cartas contenían símbolos	No	0.5
cardsSet→string	Se imprimen como strings distintos mazos	No hubo fallos	Si	1
game	Se crea con distintos mazos	No hubo errores	Si	1
stackMode	Se prueba con distintos mazos	No hubo errores	Si	1
register	Se hace prueba con distintos números de usuarios	No hubo errores	Si	1
whoseTurnIsIt?	No hubo pruebas	No se logro implementar	No	0
play	No hubo pruebas	No se logro implementar	No	0
status	No hubo pruebas	No se logro implementar	No	0
score	No hubo pruebas	No se logro implementar	No	0
game→string	No hubo pruebas	No se logro implementar	No	0
addCard	No hubo pruebas	No se logro implementar	No	0
emptyHandsStackMode	No hubo pruebas	No se logro implementar	No	0
myMode	No hubo pruebas	No se logro implementar	No	0

