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

Paradigmas de Programación

Paradigma imperativo Procedural

FELIPE MATURANA GUERRA

Profesor:
Roberto González

Ayudantes:
Kevin Álvarez
Pablo Ulloa
René Zarate

Santiago - Chile
2-2016

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

El paradigma “Funcional” se caracteriza por el uso de funciones matemáticas, en contraste de la programación imperativa la cual enfatiza en los cambios de estados mediante la mutación de variables, este paradigma en cambio no permite la modificación de estados ya que el concepto de variable ni si quiera existe con las mismas propiedades que en el paradigma imperativo, para “cambiar” el tablero por ejemplo, se construye desde 0 con las modificaciones pertinentes. Las funciones reciben parámetros y poseen un único elemento de llegada (codominio), de acuerdo a las buenas prácticas de programación es necesario respetar el conjunto de llegada o el recorrido de la función, es decir, si la función posee como recorrido el conjunto de los enteros, no se debe retornar un booleano, pues, son conjuntos de llegada distinto.

Se espera contrastar la experiencia y resultados obtenidos en esta experiencia en comparación al laboratorio anterior el cual se implementó mediante el paradigma imperativo en lenguaje C. El siguiente laboratorio está implementado en el lenguaje de programación Racket, un lenguaje de programación de alto espectro de la familia Lisp y Scheme, a pesar que es multiparadigma en esta experiencia se utilizará puramente funcional.

CAPÍTULO 2. ANÁLISIS DEL PROBLEMA

2.1 BATTLESHIP

Dentro del juego existen requerimientos específicos los cuales fueron considerados en la etapa del diseño de la solución partiendo desde lo más general como es la representación del tablero hasta los resultados de cada disparo efectuado desde el barco, pasando por funcionalidades como la verificación de la creación de tableros y además la visualización de éstos, sin dejar de pensar en la posterior implementación ya que el paradigma funcional funciona un poco distinto a lo que se está acostumbrado.

El principal problema reside en la implementación ya que la característica principal del paradigma funcional (implementado en Drracket) son las funciones, las cuales son consideradas como ciudadanos de primera clase, esto se traduce en que las funciones sólo poseen un conjunto de llegada (codominio, recorrido, etc) y el elemento de llegada es unitario, en otras palabras sólo posee un retorno. Además, nada es modificado, sólo se construye de 0 cambiando el elemento que se quiera “actualizar”. En el laboratorio anterior la solución se planteó como una matriz la cual era sometida a procedimientos mediante los cuales se actualiza su estado (paradigma imperativo). Razón principal por la cual el diseño cambia para facilitar su implementación en este nuevo paradigma.

CAPÍTULO 3. DISEÑO DE LA SOLUCIÓN

En esta ocasión el tablero se representara como una lista lineal de elementos la cual contendrá solo caracteres. En el diseño se vislumbran dos grandes TDA (Tipo de dato Abstracto), en primer lugar se identifica el TDA Ships y en segundo lugar el TDA Battleship el cual contiene parámetros que permitan un desarrollo adecuado de la partida, un tablero de la forma anteriormente mencionada además, este será el TDA principal el cual contiene dentro de él, el TDA Ships.

3. TDA'S EN DRRACKET

Los TDA's no son propios de este lenguajes, esta herramienta de abstracción de datos es universal y transversal. La solución diseñada del problema utiliza esta herramienta en sus 5 niveles de representación en el código adjunto al informe.

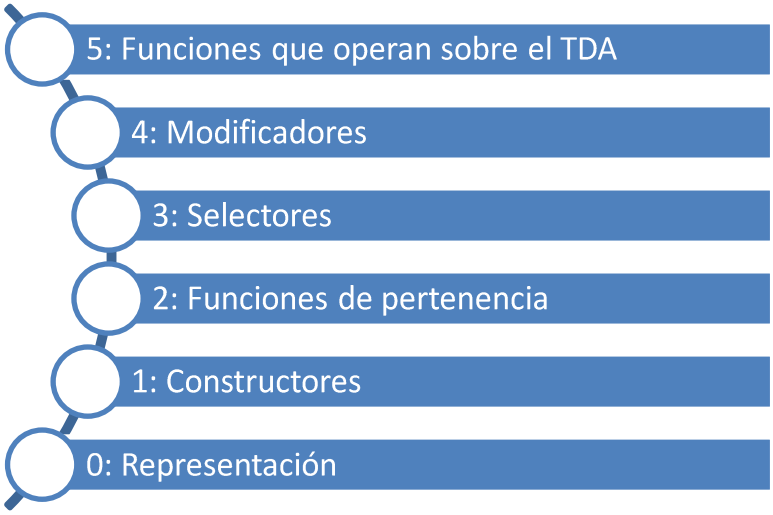

Figura 3-1: Representación estratificada TDA

3.1 TDA SHIPS

Este TDA Ships contiene la representación de los barcos, la implementación mediante listas contiene:

- Largo, Ancho, Vida, Tipo de ataque, Carácter, comandante.

El largo y el ancho es un entero que representa las dimensiones del barco representado, la vida es un entero el cual es el resultado de Largo*Ancho (este parámetro define si un barco fue destruido o no), el tercer elemento representa el tipo de ataque que posee el barco (1 para ataque normal y 2 para ataque en columna), el penúltimo elemento corresponde al carácter que representa a barco en el tablero, es necesario que estos caracteres no se repitan entre los distintos barcos, mientras que el último elemento es un entero el cual representa a quién le pertenece el barco representado (1 para jugador y 2 para enemigo).

En DrRacket la representación es una lista de la siguiente manera: 4 barcos del enemigo son representados de la siguiente forma:

'(4 (3 1 3 0 #\a 2) (3 1 3 0 #\b 2) (3 1 3 1 #\c 2) (3 1 3 1 #\d 2))

Es una lista que que contiene sub-listas con las características de cada barco. El primer entero representa la cantidad de barcos que posee y cada lista representa un barco distinto. Para acceder a ellos se utiliza una notación numérica, el primer barco es el Barco N° 0, mientras que el último es el barco N-1.

El constructor de este TDA recibe dos parámetros, N y Tipos. N corresponde a la cantidad de barcos a posicionar y tipos es una lista que contiene el tipo de barco a posicionar, esta lista tipos posee la siguiente forma:

'(1 #\a 1 #\b 2 #\c 2 #\d)

Donde el entero representa el tipo de barco (define el tipo de ataque que tendrá) y el carácter que representa a cada barco.

3.2 TDA BATTLESHIP

Este es el TDA que se ocupa en todas las funciones requeridas por la coordinación. La implementación mediante listas contiene 11 elementos necesarios para un correcto desarrollo de la partida:

i. Entero que representa la cantidad de filas del tablero creado.
ii. Entero que representa la cantidad de columnas del tablero creado.
iii. Entero que representa quién realizó el último disparo (1 para jugador, 2 para enemigo y 0 si la partida aún no ha comenzado).
iv. Entero que representa si la partida ha comenzado o no (0 si no ha comenzado, 1 si es que ya comenzó).
v. Entero que representa si la partida ha finalizado o no (0 si aún no ha terminado, 1 si ya ha finalizado).
vi. Entero que representa el estado del último disparo (0 si el último disparo cae al agua, 1 si impactó a un barco y no lo destruyó, 2 si lo hizo).
vii. Una lista que representa las N*M celdas del tablero.

viii. Entero que representa la cantidad actual de barcos que posee el enemigo.
ix. Entero que representa la cantidad actual de barcos que posee el jugador.
x. Lista de barcos (TDA Ships) que posee el jugador.
xi. Lista de barcos (TDA Ships) que posee el enemigo.

La lista que representa el tablero posee la siguiente forma:

Ej: Tablero de 3x4 (3 filas y 4 columnas)

‘(#\. #\. #\. #\X #\. #\. #\O #\. #\. #\. #\. #\.)

En el ejemplo anterior la X está en la posición (1,4) y la O en la posición (2,3) ambas en notación matricial sin embargo todas las funciones implementadas en el código utilizan una indexación que parte desde el cero tanto para las filas como para las columnas.

A modo de ejemplo, a continuación se presenta un tablero en por pantalla con show complete 1, al cual el jugado atacó en las posiciones {(0,5), (0,6), (0,7), (0,8), (0,9)} y la CPU respondió con un ataque en columna desde la posición (1,0) hasta (1,4), por lo cual su representación es la siguiente:

```
|.|O|e|e|.|@|O|O|X|@|
|b|O|c|.|.|.|.|d|d|d|
|b|O|c|.|.|.|b|b|b|.|
|b|X|c|.|.|.|.|.|.|.|
|.|O|a|a|.|c|c|c|.|.|
```

Figura 3-2: Tablero Representación TDA

```
'(5
  10
  1
  1
  0
  0
  (#\. #\O #\e #\e #\. #\@ #\O #\O #\X #\@ #\b #\O #\c #\. #\. #\. #\. #\d #\d #\d #\b #\O #\c #\. #\.
#\. #\b #\b #\b #\. #\b #\X #\c #\. #\. #\. #\. #\. #\. #\. #\. #\O #\a #\a #\. #\c #\c #\c #\. #\.)
  3
  4
  (5 (3 1 0 1 #\d 1) (3 1 2 1 #\e 1) (3 1 2 1 #\a 1) (3 1 3 1 #\b 1) (3 1 3 1 #\c 1))
  (4 (3 1 0 0 #\a 2) (3 1 3 0 #\b 2) (3 1 3 1 #\c 2) (3 1 3 1 #\d 2)))
```

Figura 3-3: Representación TDA

3.3 ALGORITMOS UTILIZADOS

La función **play** sólo está disponible para el usuario, es decir, el parámetro ships entrante debe pertenecer al usuario (comandante 1). Al realizar una jugada es necesario realizar una serie de verificaciones para velar por el correcto

funcionamiento del juego, en primer lugar se verifica que las posiciones recibidas como parámetros pertenezcan al tablero del enemigo, esta verificación evitará que el usuario se ataque su propio tablero, luego, de forma inmediata se verifica si es que la parta ha comenzado o no, de no ser así se verifica que cumpla las condiciones como para darla por comenzada (cantidad de barco de ambos jugadores mayor a 0 y la diferencia de éstos no sea superior a Δ1). Si la partida ya ha comenzado no se realiza esta verificación, sin embargo, es necesario verificar si la partida ha finalizado ya que si esto es correcto no tiene sentido que se realicen más jugadas en el tablero. Si cumple con las condiciones para realizar una jugada ocurre lo siguiente:

Disparo (Tablero, posición (x,y), cantidad de posiciones, barco, acumulador)

La posición (x,y) en Tablero está vacía?
Entonces marco con @ esa posición
La posición (x,y) en el Tablero es un barco?
Obtengo el carácter de ese barco y consulto por su vida
Si la vida del barco luego del ataque es 0
Marco la posición con X, disminuye los barcos
Marco la posición con O

Como el ataque es realizado por el jugador, al momento de obtener el carácter del barco obtengo el número del barco al que estoy atacando (N) a través de la función **ObtenerNBarco** (Algoritmo de búsqueda lineal) la cual recibe un carácter y una lista de barcos. Su recorrido son los números enteros. Lo que hace este algoritmo de búsqueda lineal es consultar barco por barco si el carácter obtenido pertenece a ese barco, de ser así se retorna el acumulador de búsqueda.

Una vez el jugador realiza su ataque la misma función realiza el ataque respuesta por parte de la CPU, para este efecto se utiliza la semilla que recibe la función **play**, en primera instancia para generar el número del barco con el cuál se atacará a través de **shipAleatorio** el cual como elemento de llega es un TDA Ships con un barco seleccionado de forma pseudo-aleatoria con la función entregada por la coordinación, posterior a esto se generan posiciones aleatorias con la semilla, si la posición generada ya se ha ataco previamente se realiza un llamado recursivo con una nueva semilla (+ seed 1), si la posición generada está disponible para ser atacada, se verifica el tipo de ataque que posee el barco seleccionado. Si este barco posee tipo de ataque 1, se generan posiciones de acuerdo a ese tipo de ataque, en este caso, ataque por columna, mediante **generadorPosiciones**. En este momento se debe cambiar el tipo de ataque del barco ya que agotó su armamento disponible sin embargo esta característica no se agregó para darle

mayor dinamismo y dificultad al juego. Luego se repite el algoritmo de Disparo expuesto anteriormente pero en esta ocasión para el barco del enemigo y en las posiciones generadas previamente. En ambos ataque se registra quién realizó la última jugada.

CAPÍTULO 4. ASPECTOS DE LA IMPLEMENTACIÓN

El presente proyecto se implementó en el lenguaje de programación Racket, de amplio espectro de la familia Lisp y Scheme, mediante la herramienta DrRacket, el cual es un entorno de desarrollo integrado (IDE). A pesar que éste lenguaje de programación es multiparaidgma el proyecto fue realizado mediante el paradigma puramente funcional, evitando el uso de set! El cual puede emular el trabajo con variables.

El código está estructurado mediante la implementación del TDA Ships y TDA battleship a través de los nivel estratificados. Representación, Función de pertenencia, Selectores, Modificadores y funciones que operan sobre el TDA. Luego de éstas implementaciones se encuentran las funciones requeridas por la coordinación (Requerimientos Funcionales Obligatorios) y luego de éstos los requerimientos extras.

4. HERRAMIENTAS UTILIZADAS

La principal herramienta utilizada en la implementación fue la recursión, nuestra principal aliada. La recursión Lineal fue utilizada en los siguientes casos:

- Ambos constructores: Principalmente porque se utilizó la función nativa **cons** del lenguaje de programación y ésta concatenaba dos elementos y el segundo era el llamado recursivo para seguir concatenando la lista, por lo cual quedaban estados pendientes.

- SetVida, setAtaque, agregarBarco: Principalmente se ocupa esta recursión porque construir listas desde otras ya creadas, pero con nuevos parámetros los cuales recibe en su argumento (“modificar”).

- CreateBoardRL: Al momento de posicionar los barcos en sus 3 celdas correspondientes, la función que cambia el estado del tablero espera por el resultado del tablero que está siendo cambiado el cuál también está siendo “actualizado” (construido desde 0 con parámetros nuevos, en este caso setCelda).

- Board -> string: Tanto para el caso de show complete 0 y 1 se aplica este tipo de recursión, por los motivos anteriormente expuestos acerca del uso de la función **cons** y los llamados recursivos esperan un nuevo estado.

La recursión de cola fue utilizada en los siguientes casos:

- Play: Debido a la cantidad de ataques que se pueden realizar resulta poco eficaz en sentido de recursos utilizados tener tantos estados pendientes utilizando memoria, es por esto que la recursión de cola optimiza la cantidad de recursos utilizados para esta acción, el llamado recursivo se realiza sin estados pendientes, donde la función recibe por parámetro el nuevo tablero actualizado, hasta el caso base.

- ObtenerNBarco: Al ser un algoritmo de búsqueda resulta conveniente no dejar estados pendientes ya que es contraproducente ocupar memoria de forma innecesaria.

4.1 REQUERIMIENTOS EXTRA

La función **getScore** recibe un tablero (elemento de salida) y el conjunto de llegada son los números enteros.

Del tablero obtiene la mitad del enemigo y cuenta la cantidad de disparos exitosos, fallidos, barcos destruidos del enemigo y barco que le han destruidos. La fórmula del cálculo es la siguiente:

$$score = (DE * 100 + BD * 400 + (BJ - BE) * 200) - DF * 50,$$

$$if\ score < 0 \rightarrow score = 0$$

Figura 4-1: Función getScore

DE= Disparos exitosos, BD= Barcos destruidos, BJ= Barcos actual del jugador, BE= Barcos actual del Enemigo.

CAPÍTULO 5. RESULTADOS

De los requerimientos funcionales obligatorios se cumplió a cabalidad cada uno de los 7 mencionados en el informe. El código es capaz de:

1. Ambos TDA se encuentran implementados mediante la estructura de 6 capas estratificadas mencionadas en el capítulo 3 (*Figura 3.1*).

2. Generar un tablero válido de dimensiones NxM el cual es divido por la mitad y se le asigna a cada jugador una mitad. Además esta función posiciona todos los barcos enemigos que reciba por parámetro la función createBoard. Esta función se encuentra implementada con los dos tipos de recursión (Cola y Lineal), verificando que los parámetros de entradas sean congruentes con lo solicitado, por ejemplo que la cantidad de columnas sea par.

3. Verificar las condiciones del tablero, principalmente del TDA Battleship, que los parámetros que contenga el TDA sean congruentes con su implementación, se apoya de la función de pertenencia para verificar los tipos de datos que éste debe contener y además verifica que pertenezcan al recorrido de la función, por ejemplo que el parámetro partida iniciada no sea un número negativo o un número superior a 1 ya que si éste parámetro contiene un 3 por ejemplo cumple con la función de pertenencia porque es un entero, sin embargo no es congruente con su representación ya que éste estado se representa a través de un 0 o un 1.

4. Realizar jugadas en el tablero, disparos de distintas posiciones según la disponibilidad armamentística de los barcos de cada jugador, en este caso, se mantuvo de forma ilimitada los disparos especiales ya que le dan mayor dinamismo al juego y dificultad, de lo contrario basta con cada vez que hagan un disparo en área cambiar el entero TipoATK a 0 con **setTipoATK**.

5. Posicionar barcos en una posición dada con putship, verificando que esté posicionando en la mitad del tablero que le corresponde a él.

6. Imprimir el tablero de forma parcial (sólo la mitad del jugador) o completa.

7. Obtener el puntaje de una partida.

CAPÍTULO 6. CONCLUSIÓN

El lenguaje de programación, o más bien, el paradigma resultó ser menos efectivo en la implementación respecto al paradigma imperativo, sin embargo se esperaba que fuera mucho menos efectivo. El gran manejo de listas que posee se complementa muy bien con la representación que se eligió en esta ocasión, sin embargo es necesario documentarlo de buena forma para que sea entendible y legible para otra persona. A lo largo del proyecto se hizo valer el nombre del paradigma, ya que las funciones son nuestra principal arma y la recursión nuestra mejor aliada. Sin embargo se recomienda el uso de funciones para problemas pequeños ya que de ésta forma el código queda más prolijo y entendible, el método de resolución de problemas división en sub problemas resulta extremadamente útil en este proyecto.

CAPÍTULO 7. REFERENCIAS

Referenciar páginas web:

Hasbro. (2012). *Reglas para 2 Jugadores BATTLESHIP.* (Recuperado 2016). http://www.hasbro.com/common/documents/dad261471c4311ddbd0b0800200c9a66/DC43DE785056900B109B0A81EE470A9E.pdf

Bernal, J (2013). *LENGUAJE IMPERATIVO O PROCEDURAL.* (Recuperado 2016). http://lenguajesdeprogramacionparadigmas.blogspot.cl/2013/03/lenguaje-imperativo-o-procedural.html

AHO, A., HOPCROFT, J.; ULLMAN, J. D. (1987*). Data Structures and Algorithms Addison-Wesley*. Estados Unidos de América.

Kölher, J. (2015). *Apuntes de la Asignatura Análisis de Algoritmos y Estructuras de Datos.* Chile.