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

Paradigmas de programación
Informe

CRISTIAN EDUARDO ESPINOZA SILVA

Profesor:
MIGUEL TRUFFA MONTENEGRO

Santiago - Chile
2016

Tabla de contenido

Tabla de contenido de diagramas

Tabla de contenido de figuras

CAPÍTULO 1. INTRODUCCIÓN

Continuando con el tema presentado en el informe anterior, es sabido que los lenguajes de programación han ido evolucionando de manera exponencialmente, estos cambios radican en la búsqueda de nuevas modalidades de resoluciones de problemas, es así como surge el paradigma funcional que fue diseñado principalmente para poder desarrollar problemas matemáticos, basados en la función lambda.
En el presente informe se dará a conocer las ventajas y desventajas que tendremos al momento de abarcar el proyecto, retroalimentando con el desarrollo expuesto en el paradigma imperativo.

1.2 DESCRIPCIÓN DEL PROBLEMA

El problema consiste en diseñar un juego llamado "BATALLA NAVAL", el cual tiene una temática de derribar *ship* enemigos. Se autogenera una matriz de N x M, donde dichas dimensiones son ingresadas por el usuario. El jugador enemigo debe posiciones sus *ships* en N/2 de la matriz y, el usuario en la otra mitad de la matriz.
El principal objetivo es derribar el barco enemigo, el cual se logra cuando un ataque de un *ship* recae en un *ship* enemigo.
El primer jugador que acabe con la totalidad de los *ships* es el ganador del juego.

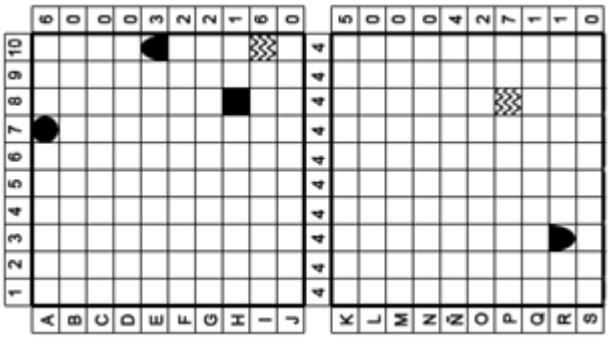

Figura 1.2-1: Temática del juego "Batalla Naval".

1.3 OBJETIVOS DEL PROYECTO

El objetivo es realizar un programa que represente el juego "Batalla Naval" desarrollado en el lenguaje de programación *racket*, ocupando el paradigma de programación funcional.
Los objetivos del proyecto son implementar ciertas funciones las cuales recaen en las siguientes:

- Realizar una matriz de N x M dimensiones.
- Probar si la matriz creada, es válida para poder comenzar una partida.
- Designar las posiciones de los *ships* del oponente en N/2 x M de la matriz y colocar en la otra mitad un *ship* del usuario en alguna posición ingresada.
- Generar ataques dentro de la aplicación, dejando marca en la matriz al realizar el disparo, además de entregar un registro si el disparo causo daños a algún *ship*.
- Transformar la matriz a un tipo de dato *String*, con el fin de dar una mejor representación y apreciación.

1.4 ASPECTOS DE IMPLEMENTACIÓN

Para abordar cada uno de los problemas que se encuentran al momento de implementar la solución al problema se debe ocupar: Similitud débil, División en sub-problemas y Recursión.
Esta última es una de las soluciones que más se ocupa, ya que este paradigma se basa en desarrollar los problemas median recursión lineal y recursión de cola.
Estos conceptos mencionados se llevaron a cabo mediante el lenguaje de programación *Scheme, estándar r6rs*, además para el desarrollo de la aplicación el intérprete *Dr. racket* versión *v6.6.*
El principal alcance que se presenta es el cumplimiento de las funcionalidades pedidas en el enunciado entregado.

1.5 DESCRIPCIÓN DE LA SOLUCIÓN

La solución que se le dará al problema debe ser basada en el paradigma de programación funcional e implementada en el lenguaje de programación *Scheme, estándar r6rs*. Los resultados de cada función, serán mostrados por la consola que ofrece el intérprete *Dr. racket*.
Se realizo una abstracción que fue adaptada a un tipo de dato abstracto (TDA), en el transcurso del informe se revela la implementación que se ocupó para el desarrollo del proyecto.

1.6 FUNDAMENTOS TEÓRICOS

El paradigma funcional se caracteriza por ser un lenguaje el cual no se manejan variables, no permite realizar modificaciones en los objetos creados.

Es un lenguaje declarativo, que se basa en una función matemática que convierte unas entradas en unas salidas, sin estado interno ni efectos laterales.

Agregar que una de sus principales herramientas es el cálculo *lambda*. El cálculo *lambda* es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión.

Scheme es un paradigma funcional y extensión de *Lisp.* Fue desarrollado por *Guy L. Steeele* y *Gerald Jay Sussman* alrededor de la década de los setenta, fue introducido al mundo académico mediante un artículo conocido como "*Lambda Papers de Sussman y Steele"*.

La estructura básica del lenguaje son las listas, que soportar gran cantidad de elementos y no necesariamente tiene que ser del mismo tipo de dato.

Por último, se debe tener presenta que la filosofía de Scheme es minimalista, el objetivo principal no es ir acumulando numerosas funcionalidades.

CAPÍTULO 2. DESARROLLO DE LA APLICACIÓN

2.1 PROYECTO Y PARADIGMA

2.1.1 Tipo de datos abstractos implementados en la aplicación

Un tipo de dato abstracto (TDA), es una representación de un objeto que se le da en un determinado lenguaje de programación, que se implementa mediante las características y comportamientos que tendrá dentro del proyecto.

Tenemos que tener en cuenta que todo TDA, tiene que tener ciertas propiedades que son fundamentales en él, las cuales recaen en:

- Representación.
- Pertenencia: En esta clasificación se encuentran las funciones que se encargan de verificar si él objeto pertenece al TDA. (imagen en anexo)
- Construcción: Esta función dan vida al TDA, son las encargadas de crear el objeto que se declara en la representación. (imagen en anexo)
- Selección: Contiene las funciones que permiten ingresar a objetos específicos que pertenecen al TDA. (imagen en anexo)
- Modificación: Se encuentran las funciones que realizan cambios, principalmente se encargan de generar un nuevo objeto con la modificación solicitada, esto se implementa de dicha forma para no violar el paradigma funcional. (imagen en anexo)
- Otras funciones: Contiene funciones externas que ayudan a manejar y realizar procesos externos que facilitan la solución final. (imagen en anexo)

Se realizó dos TDA dentro de la aplicación, las que recaen en:

- TDA DE SHIP:
    - Representación:
        - Un *ship* se representará con una lista de 3 componentes de una manera similar a: (x, y, z)
            - X: Representará la coordenada “x” de donde está ubicado el *ship* en el matriz creada.
            - Y: Representará la coordenada “y” de donde está ubicado el *ship* en la matriz creada.
            - Z: Representara el estado de la casilla en la cual se está señalando, pudiendo tener 3 estado:
                - (#\~): Representa la casilla con mar.

- (#\~): Representa la casilla atacada.
- (#\" nombre"): Representa que la casilla está ocupada por un *ship*.

- TDA DE BOARD:

  - Representación: La matriz está representada por una lista, que en su interior contiene dos listas.
    Donde la primera lista será donde se guarde la matriz creada y la segunda será donde se guarde la última jugada realizada en la matriz.
    Se podrá apreciar de la siguiente forma: ((x, y, estado) (Historial de última jugada)).

    - Lista (x, y, estado): Está contenida por cada posición que se genere según las dimensiones que tendrá la matriz y el estado que se encuentra.
    - Lista (Historial) -> Está implementada de la siguiente forma: (x , y , ship , resultado) donde:
      - x e y: Son las coordenadas de donde se realizó el disparo.
      - *Ship*: Es el *ship* que realiza el ataque.
      - Resultado: Este está conformado de dos posibles opciones son:
        - 0: Si el disparo fue en el agua o en una posición donde ya se había disparado anteriormente.
        - 1: Si el disparo fue en algún *Ship* enemigo.

2.1.1.1 *Funciones createBoardRL, checkBoardRC*

Son las encargadas de generan la matriz que se utiliza durante el transcurso de la partida, existen dos formas de crear la matriz, a través de recursión lineal y recursión de cola.

Cómo se dijo anteriormente están encargadas de crear la matriz con las dimensiones que el usuario estime conveniente e ingrese a la función.

Como entrada recibe N, M, SHIPS, que corresponden a:

- N e M: Dimensiones que tendrá la matriz que se va a crear.
- Ships: Lista de *ships* enemigos, los cuales se van a posicionar en N/2 x M de la matriz.

Antes de continuar con la creación de la matriz, como se mencionó recibe como parámetro una lista de *ships*, la cual es establecida de la siguiente forma:

Diagrama 2.1.1.1-1: TDA *Ship*.

Como se observa en el diagrama de abstracción:

- agregarBarcos: Se encarga de crear la lista de *ship* que paso el enemigo, asignando aleatoriamente coordenadas a cada uno de ellos.
- verificar: Se encarga de comprobar que las posiciones de los *ships* no sean repetidas.
- makeBarcos: Esta función se encarga de entregar la lista final de *ship*, contiene un proceso interno que verifica que las posiciones de los *ship* no sean repetidas para así evitar que se sobrepongan. Finalizando la salida será la lista que contendrá *ship* con sus respectivas coordenadas.

Con lo anterior se da fin al TDA *ship*, que era el encargado de crear una lista con los *ship* y sus coordenadas, tomando en cuenta que *makeBarcos* es el constructor de dicho TDA.

A continuación, se procede a entregar la lista *ship* a createBoardRC o createBoardRL.

El diagrama que se muestra a continuación descompone el proceso de crear una matriz:

Diagrama 2.1.1.1-2: Proceso de creación de tablero.

Se tomó la función createBoardRC: Esta función no va dejando estados pendientes, ya que la solución de cada llamado recursivo lo entrega como el nuevo parámetro.

Esta función verifica cada posición si es idéntica a alguna coordenada de los *ship* enemigos, si es así abran dos opciones:

- Se coloca un #\~, si las posiciones no coinciden con ninguna de los *ship*, que esto significa mar en la matriz.
- Se coloca un #\" nombre", si las posiciones coinciden con la de algún *ship*.

Al terminar el recorrido de cada una de las posiciones, alcanzará las condiciones que cumplan con su caso base, entonces retorna la matriz creada.

2.1.1.2 *Función checkBoard*

Es la encargada de ver la veracidad de una matriz generada, lo que significa que cumpla con las

condiciones para que se pueda desarrollar el juego sin problemas. Debe verificar que el número de filas sea número par y que cada lista que contiene sea un objeto que pertenece al TDA.

2.1.1.3 *Función putShip*

Está encargada de posicionar un *ship del* usuario en N/2 x M de la matriz.
Está recibe como parámetro un *Board*, positions y *ship*.

- *Board*: Es la matriz actual que se ocupa como base para crear la nueva matriz con sus modificaciones.
  Para no violar el paradigma funcional se crea un nuevo objeto con las modificaciones respectivas, ya que este paradigma no permite modificación en sus variables.
- *Positions*: Es una lista que contiene las coordenadas x e y en donde se colocará el *ship* en la matriz.
- *Ship*: Nombre del *ship.*

Luego, como salida de la función se entrega la nueva matriz generada con las modificaciones realizadas.

Principalmente de lo que se encargara la función "putShip" es hacer el llamado a la función "recorrerBoard", además de verificar si los datos entregados pertenecen al TDA.

El funcionamiento interno de la función "recorrerBoard", se muestra en la siguiente imagen:

```
(else (if(and (equal? x (puntoXB board)) (equal? y (puntoYB board)) )
          (recur (append boardNuevo (list(list h k ship))) (+ 1 h) 0 (cdr board) x y)
          (recur (append boardNuevo (list(list h k (barcoDeListaB board)))) (+ 1 h) 0 (cdr board) x y )
          )
```

Figura 2.1.1.3-2: Recursión de cola (recorrerBoard).

Se encarga de ir creando la nueva matriz, pero modificando la posición donde se quiere agregar el *ship*.

Se utiliza recursión de cola, ya que no va dejando estados pendientes además de ir actualizando los parámetros en el llamado recursivo.

Al momento que la función anterior llega al caso base, que se muestra en la siguiente imagen:

```
(if(null? board) boardNuevo
```

Figura 2.1.1.3-3: Caso base (recorrerBoard) .

Se procede a retornar la nueva matriz creada.

2.1.1.4 *Función boardString*

Esta función está encargada de convertir la matriz que inicialmente era una lista de lista a un tipo de dato *string*, con el fin que el usuario la aprecie de una mejor manera.
Recibe como parámetro:

- *Board*: La matriz que se va a transformar a *string*.
- *ShowComplete*: Es un entero que puede ser:
  - 1: Cuando desea mostrar la totalidad de la matriz.
  - 0: Cuando desea ocultar la parte enemiga en la matriz.

2.1.1.5 *Función play*

Es la función que crea la dinámica en el juego.

Se encarga de poder realizar un ataque dentro del juego, esta función recibe como parámetro: *Board*, *position* y *ship*.

- *Board*: Es la matriz que está confeccionada por lista de lista, que se creó para la partida.
- *Position*: Contiene las coordenadas donde el usuario desea realizar el disparo.
- *Ship*: Es el *ship* que el usuario escoge para que realice el disparo.

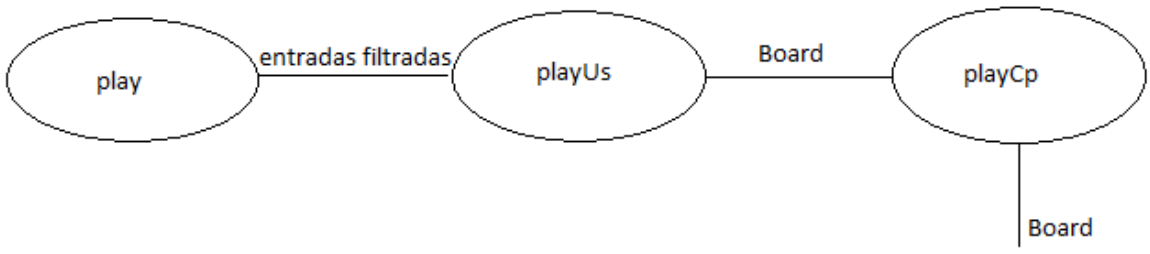

Diagrama 2.1.1.5-3: Diagrama de abstracción de función Play.

La función "play" se encarga de verificarlas entradas y si los objetos pertenecen al TDA, además de realizar el llamado a la función "playUs".

Esta realiza una nueva matriz, con el ataque del usuario realizado en las posiciones indicadas.

Además de llamar a la función "playCp" y entregarle la nueva matriz creada.

Esta función termina de realizar el proceso de ataque y de retornar la matriz final con todas las modificaciones realizadas.

La secuencia de pasos que sigue es la siguiente:

- Primero pide aleatoriamente el *ship* con el cual se va a realizar el ataque, además de generar aleatoriamente las coordenadas donde se va a efectuar el disparo.
- Luego se crea la nueva matriz con las modificaciones realizadas y, de generar el historial de la última jugada realizada, que contiene lo siguiente:
    - Historial: ((autor x y barco resultado) (autor x y barco resultado))
        - Autor: Representa quien realizo el dicho disparo.
        - x e y: Coordenadas donde fue efectuado el disparo.
        - barco: *Ship* que realizo el disparo.
        - resultado: Es posible que muestre algún resultado que contiene la representación.

El último proceso que realiza "playCp" es entregar la nueva matriz con sus respectivas modificaciones en conjunto con el historial de la última jugada.

2.1.2 Resultados

Los resultados de esta aplicación se pueden ir observando al momento de realizar el llamado a las funciones.

A continuación, se expondrán algunos de los resultados al momento de realizar algún tipo de llamado.

```
#| 6. Luego hacemos algunas de las funciones que nos permite la aplicacion |#

;     * Funcion putShip

(define boardListo (putShip (putShip (putShip board '(6 6) #\h) '(9 7) #\t) '(6 9) #\o))
(display boardListo)
```

```
{{0 0 -} {0 1 -} {0 2 -} {0 3 -} {0 4 -} {0 5 -}
{0 6 -} {0 7 -} {0 8 -} {0 9 -} {1 0 -} {1 1 -}
{1 2 -} {1 3 -} {1 4 -} {1 5 -} {1 6 -} {1 7 c}
{1 8 -} {1 9 -} {2 0 -} {2 1 -} {2 2 -} {2 3 -}
{2 4 -} {2 5 -} {2 6 -} {2 7 y} {2 8 -} {2 9 -}
{3 0 -} {3 1 -} {3 2 -} {3 3 -} {3 4 -} {3 5 k}
{3 6 -} {3 7 -} {3 8 -} {3 9 -} {4 0 -} {4 1 -}
{4 2 -} {4 3 -} {4 4 -} {4 5 -} {4 6 -} {4 7 -}
{4 8 -} {4 9 -} {5 0 -} {5 1 -} {5 2 -} {5 3 -}
{5 4 -} {5 5 -} {5 6 -} {5 7 -} {5 8 -} {5 9 -}
{6 0 -} {6 1 -} {6 2 -} {6 3 -} {6 4 -} {6 5 -}
{6 6 h} {6 7 -} {6 8 -} {6 9 o} {7 0 -} {7 1 -}
{7 2 -} {7 3 -} {7 4 -} {7 5 -} {7 6 -} {7 7 -}
{7 8 -} {7 9 -} {8 0 -} {8 1 -} {8 2 -} {8 3 -}
{8 4 -} {8 5 -} {8 6 -} {8 7 -} {8 8 -} {8 9 -}
{9 0 -} {9 1 -} {9 2 -} {9 3 -} {9 4 -} {9 5 -}
{9 6 -} {9 7 t} {9 8 -} {9 9 -}}
> |
```

Figura 2.1.2-4: Matriz nueva con los barcos de los usuarios ubicados.

```
;     * Funcion play

(define final (play boardListo #\h '(0 1) 156432))
(display final)
```

```
{{{0 0 -} {0 1 x} {0 2 -} {0 3 -} {0 4 -} {0 5
-} {0 6 -} {0 7 -} {0 8 -} {0 9 -} {1 0 -} {1 1
-} {1 2 -} {1 3 -} {1 4 -} {1 5 -} {1 6 -} {1 7
c} {1 8 -} {1 9 -} {2 0 -} {2 1 -} {2 2 -} {2 3
-} {2 4 -} {2 5 -} {2 6 -} {2 7 y} {2 8 -} {2 9
-} {3 0 -} {3 1 -} {3 2 -} {3 3 -} {3 4 -} {3 5
k} {3 6 -} {3 7 -} {3 8 -} {3 9 -} {4 0 -} {4 1
-} {4 2 -} {4 3 -} {4 4 -} {4 5 -} {4 6 -} {4 7
-} {4 8 -} {4 9 -} {5 0 -} {5 1 -} {5 2 -} {5 3
-} {5 4 -} {5 5 -} {5 6 -} {5 7 -} {5 8 -} {5 9
-} {6 0 -} {6 1 -} {6 2 -} {6 3 -} {6 4 -} {6 5
-} {6 6 h} {6 7 -} {6 8 -} {6 9 o} {7 0 -} {7 1
-} {7 2 -} {7 3 -} {7 4 -} {7 5 -} {7 6 -} {7 7
x} {7 8 -} {7 9 -} {8 0 -} {8 1 -} {8 2 -} {8 3
-} {8 4 -} {8 5 -} {8 6 -} {8 7 -} {8 8 -} {8 9
-} {9 0 -} {9 1 -} {9 2 -} {9 3 -} {9 4 -} {9 5
-} {9 6 -} {9 7 t} {9 8 -} {9 9 -}} {{computador
ataque en coordenadas  x: 0 e y: 1 Realizado por
el barco h .Resultado del ataque 0} {usuario
ataque en coordenadas  x: 0 e y: 1 Realizado por
el barco y .Resultado del ataque 0}}}
> |
```

Figura 2.1.2-5: Función que realizo los disparos dentro del campo de juego.

```
;     * Funcion board->string

(define boardString (board->string (car final) 0))
(display boardString)
```

```
- x - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - h - - o
- - - - - - - x - -
- - - - - - - - - -
- - - - - - - t - -
```

Figura 2.1.2-6: Función que muestra la matriz final (0 -> Solo mitad del usuario).

```
;     * Funcion que muestran en la matriz en otros formatos

(define Board->json (board->xml board ))
(define Board->xml (board->xml (car final) ))

(display Board->json)
(display "\n")
(display Board->xml)
```

```
<board><fila0>----------</fila0><fila1>-------c--</f
ila1><fila2>-------y--</fila2><fila3>-----k----</fil
a3><fila4>----------</fila4><fila5>----------</fila5
><fila6>----------</fila6><fila7>----------</fila7><
fila8>----------</fila8><fila9>----------</fila9></b
oard>

<board><fila0>-x--------</fila0><fila1>-------c--</f
ila1><fila2>-------y--</fila2><fila3>-----k----</fil
a3><fila4>----------</fila4><fila5>----------</fila5
><fila6>------h--o</fila6><fila7>-------x--</fila7><
fila8>----------</fila8><fila9>-------t--</fila9></b
oard>
```

Figura 2.1.2-7: Función que muestra la matriz en distintos formatos creados.

CAPÍTULO 3. CONCLUSIONES

La resolución del problema utilizando el paradigma funcional fue alcanzada, se cumplió con los requisitos obligatorios tanto como funcionales como no funcionales, además de realizar dos requisitos extras planteados en el enunciado.

Se realizó un trabajo mucho más organizado a comparación del primer laboratorio, en temas de tiempo y eficiencia en el momento de programar. A pesar de ser distintos lenguajes se tenía conciencia del objetivo a cumplir y de conocer cómo manejar algunos obstáculos que se obtuvieron en el proyecto anterior.

Una de las características que conlleva a complejidad mayor al desarrollo del proyecto fue la sintaxis que tiene el lenguaje de programación, es poco usual y es primera experiencia con una sintaxis de dicha manera, además de poder adaptarse a resolver la mayoría de los problemas con llamados recursivos.

Pero sin duda el mayor problema se vio en el momento de no poder utilizar variables, era una herramienta fundamental al momento de programar que llevaba a la solución a varios problemas, sin embargo, en este lenguaje de programación se tuvo que aprender a abarcar los problemas sin dicha herramienta y adaptarse en su totalidad al llamado recursivo.

En comparación con el laboratorio anterior, las maneras de ver las soluciones a los problemas eran mucho más abstractas, además que se tenían que abarcar de distinta manera a lo habitual.

Se debe destacar que el paradigma funcional fue un desafío enfrentarlo, ya que es la primera experiencia con dicha sintaxis, además de buscar los soluciones a los problemas ocupando la recursión en gran medida.

Por ultimo cabe señalar que los resultados en el laboratorio fueron exitosos, debido que al momento de realizar pruebas de las funciones creadas, tuvieron un alto porcentaje de aprobación.

CAPÍTULO 4. REFERENCIAS

- https://racket-lang.org/
- https://drive.google.com/file/d/0B2Y7zxLENeBzREwxcVRTdm1GN3c/view
- http://ceciliaurbina.blogspot.cl/2010/09/lenguajes-funcionales.html
- https://es.wikipedia.org/wiki/Scheme

CAPÍTULO 5. ANEXO

- *TDA SHIP:*

  - (define carácter)
  - (define numero)
  - (define par)
  - (define listaDeBarcos)
  - (define listaDePares)
  - (define lista)

Figura CAPÍTULO 5. 9: Pertenencia - Ship

  - (define puntoX)
  - (define puntoY)
  - (define barcoDeLista)

Figura CAPÍTULO 5. 8: Selectores - Ship

  - (define modificarLista)
  - (define modificarListaBarco)
  - (define modificarListaShip)

Figura CAPÍTULO 5. 10:Modificadores - Ship

  - (define agregarBarcos)
  - (define verIgualesB)
  - (define verificar)
  - (define myRandom)
  - (define getListaRandom)
  - (define verIguales)

Figura CAPÍTULO 5. 11: Otras funciones - Ship

  - (define makeBarcos)

Figura CAPÍTULO 5. 12:Constructor – Ship

- *TDA BOARD:*

- (define busAtack)
- (define checkBoard)
- (define caracterB)
- (define numeroB)
- (define parB)
- (define listaDeBarcosB)

Figura CAPÍTULO 5. 14: Selectores – Board

- (define puntoXB)
- (define puntoYB)
- (define barcoDeListaB)
- (define punto-X)
- (define punto-Y)

Figura CAPÍTULO 5. 13:Pertenencia - Board

- (define colocarAgua)
- (define colocarDisparo)
- (define verPosicionBoard)
- (define similitud)
- (define verDisparo)
- (define DisparoUs)
- (define playCp)
- (define play)
- (define playUs)
- (define colocarBarco)
- (define putShip)
- (define recorrerBoardRespaldo)
- (define recorrerBoard)

Figura CAPÍTULO 5. 16: Modificadores – Board

- (define posXCp)
- (define posYCp)
- (define verIgualesBa)
- (define verIgualesBar)
- (define board-string)
- (define boardStringComplete)
- define boardStringNoComplete)
- (define barcoAtackOpe)
- (define barcosOpenentes)
- (define mitadTablero)

Figura CAPÍTULO 5. 15: Otras funciones - Board

- (define createBoardRC)
- (define createBoardRL)

Figura CAPÍTULO 5. 17: Constructor – Board