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

INFORME DE PROLOG

PARADIGMAS DE PROGRAMACIÓN

Nombre: Cristian Espinoza Silva.
Profesor: Miguel Truffa Montenegro.
Ayudante: René Zarate.
Fecha de Entrega: 7 de noviembre.

Santiago de Chile

2 - 2016

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE DIAGRAMAS

CAPÍTULO 1. INTRODUCCIÓN

Se continua, con el tema expuesto en el informe anterior, es sabido que los lenguajes de programación han ido evolucionando. Estos cambios radican en la búsqueda de nuevas modalidades de resoluciones de problemas, es así como surge el paradigma lógico, que basa en operaciones lógicas para determinar un resultado.

En el presente informe se dará a conocer las ventajas y desventajas que se tendrá al momento de abarcar el proyecto, retroalimentando con el desarrollo expuesto en los paradigmas anteriores.

1.1 DESCRIPCIÓN DEL PROBLEMA

El problema consiste en diseñar un juego llamado "BATALLA NAVAL", tiene una temática en esta ocasión de verificar si la jugada realizada es válida.

La matriz está compuesta por diferentes barcos que al transcurso del informe se les mencionará como *ships*, los cuáles ya estarán posicionados al momento de crear dicha matriz.

1.2 OBJETIVOS DEL PROYECTO

El objetivo es realizar un programa que represente el juego "Batalla Naval" desarrollado en el lenguaje de programación Prolog e implementado en el compilador -interprete *Swi-prolog*, ocupando el paradigma de programación lógico.

Los objetivos del proyecto son implementar ciertas funciones, las cuales recaen en las siguientes

- Implementar diferentes tipos de matrices (*Board*), que recaen en las siguientes:
  - 3 matrices de 6 x 6 – Conteniendo 3 *ship* por jugador.
  - 2 matrices de 10 x 10 – Conteniendo 5 *ship* por jugador.
  - 1 matriz de 20 x 20 – Conteniendo 10 *ship* por jugador.
- Probar si la matriz creada, es válida para poder comenzar una partida.
- Generar ataques dentro de la aplicación, además verificar si el ataque es válido para dicha ocasión.
- Transformar la matriz a un tipo de dato *String*, con el fin de dar una mejor representación y apreciación.

1.3 FUNDAMENTOS TEÓRICOS

El paradigma lógico se basa en declaraciones de hechos y reglas. Los hechos son objetos declarados que pasan a estar dentro del conocimiento de la aplicación, en cambio las reglas son procedimientos que basan su respuesta a través hechos que estén declarados dentro de la aplicación. Además de dar posibles soluciones a problemas mediante la lógica de primer orden, también llamada lógica de predicados o cálculo de predicados, es un sistema formal diseñado para estudiar, además se puede cuantificar a lo más, sobre variables para objetos.

El paradigma lógico es un paradigma declarativo que se basa en realizar consultas que son respondidas con verdadero o falso, mediante los hechos que estén declarados en la aplicación.

Para el desarrollo de la aplicación se ocupó el compilador - intérprete *Swi-prolog versión 7.2.3.*

1.4 ASPECTOS DE IMPLEMENTACIÓN

Para abordar cada uno de los problemas que se encuentran al momento de implementar la solución al problema se ocuparon hechos, reglas y recursión de cola.

La solución que más se dio énfasis dentro de la aplicación fueron las reglas que su desarrollo se basa en ir verificando hechos que estén declarados dentro de la aplicación.

El alcance esperado es cumplir con los requisitos expuestos el enunciado entregado, además de desarrollar la funcionalidad extra *boardToString*.

1.5 DESCRIPCIÓN DE LA SOLUCIÓN

La solución que se le dará al problema debe ser basada en el paradigma de programación lógico. Los resultados de cada predicado, serán mostrados por la consola que ofrece el compilador - intérprete *Swi-prolog.*

Se realizaron varias abstracciones para poder satisfacer cada una de las funcionalidades que se solicitan. Con el fin de declarar los hechos necesarios que ayuden a satisfacer las funcionalidades. Así ocupando lo anterior para implementar reglas que darán posibles soluciones a sub-problemas que se presenten.

CAPÍTULO 2. DESARROLLO

2.1 PROYECTO

2.1.1 DEFINICIÓN DE HECHOS

En la aplicación se declaran los siguientes hechos, que se describen en las siguientes imágenes:

```
barco(1).
barco(2).
barco(3).
barco(4).
barco(5).
barco(6).
mar(~).
```

*Figura **Error! Reference source not found.**-1: Declaración de hechos - (Ships).*

Como se observa en la imagen 2.1.1-2, se declaran los hechos que identifican a los *ships* válidos, además del símbolo que va a representar al mar.

Luego se declaran todos los *Board* que se ocupan dentro de la aplicación como hechos, que son los siguientes:

- *Board de 6x6 – 3 ships por jugador.*
- *Board de 10x10 – 5 ships por jugador.*
- *Board de 20x20 – 10 ships por jugador.*

En la siguiente imagen se puede aprecia la declaración del *Board 10x10:*

```
tablero(_,10,10,10,
        [[1,~,~,3,3,3,~,~,~,~],
         [~,~,~,~,~,4,4,4,4,~],
         [~,~,2,2,~,~,~,~,~,~],
         [~,5,5,5,5,5,~,~,~,~],
         [~,~,~,~,~,~,~,~,~,~],
         [~,~,~,5,5,5,5,5,~,~],
         [4,~,~,~,~,~,~,~,~,~],
         [4,~,~,3,~,~,1,~,~,~],
         [4,~,~,3,~,~,~,~,2,2],
         [4,~,~,3,~,~,~,~,~,~]]).
```

*Figura **Error! Reference source not found.**-2: Declaración de hechos - (Board 10x10).*

Cada *ship* es representado por el número de su largo dentro del *Board*, lo que produce que se aprecia de mejor forma la longitud que tiene.

Además, para comprobar el predicado de validar un *Board*, se declaran *Boards* que son inválidos, es decir, que no cumplen con las condiciones elementales del juego. De tal forma, queda mejor visto en la siguiente imagen:

```
tablero(3,10,10,10,
        [[1,~,3,3,3,~,~,~,~,5],
         [~,2,2,~,~,4,~,~,~,5],
         [~,~,~,~,~,4,~,~,~,5],
         [~,~,~,~,~,4,~,~,~,5],
         [~,~,~,~,~,4,~,~,6,5],
         [~,~,~,5,5,5,5,5,~,~],
         [~,~,~,~,~,~,4,~,~,~],
         [3,~,~,~,~,~,4,~,~,~],
         [3,~,~,2,~,~,4,~,~,1],
         [3,~,~,2,~,~,4,~,~,~]]).
```

*Figura **Error! Reference source not found.**-3: Declaración de hecho – (Board 10x10- invalido).*

La diferencia es que contiene un *ship* adicional representado por el número “6”, produce que no todos los tableros sean válidos dentro de la aplicación, ya que no cumple con el requerimiento elemental de la cantidad de *ships* que debe contener un *Board.*

Y por último se declaran los hechos de la cantidad de *ships* que son válidos para cada *Board*, se debe tener en cuenta que contabiliza el largo de cada uno de ellos, se puede apreciar de mejor forma en la figura 2.1.1-5:

```
verificarCantBarcos(6,12).
verificarCantBarcos(10,30).
verificarCantBarcos(20,60).
```

*Figura **Error! Reference source not found.**-4: Declaración de hecho – (Cantidad de ships).*

Se interpreta de la siguiente forma:

- En el primer caso:
  - El numero 6 representa el tamaño del *Board,* por ende, el 12 representa la cantidad de posiciones ocupadas por *ships*, este número se obtiene sumando todas las posiciones que tienen los *ships* dentro del *Board.*

2.1.2 DEFINICIÓN DE REGLAS

Teniendo definidas la totalidad de los hechos explicados anteriormente, se procede a trabajar con ellos para poder formulas las reglas:

2.1.2.1 Predicado createBoard

Este predicado está encargado de crear la matriz, con el cual se llevará inicio a la partida.

Como entrada recibe N, M, NumShips, BOARD, que corresponden a:

- N y M: Dimensiones que tendrá el *Board que se va a utilizar*.
- NumShips: Numero de *ships* que contendrá el *Board.*
- *BOARD:* Variable donde se guarda el *Board* creado.

Cuando se llama a la función con ciertos parámetros se verifica si cumple con alguna de las matrices que se encuentran en el conocimiento de la aplicación, es decir, en los hechos declarados anteriormente.

La aplicación consta de 9 matrices creadas las cuales cada una de ellas son hechos dentro de la aplicación, consta de 3 matrices inválidas, es decir, que no cumplen con las reglas fundamentales, las cuales son las siguientes:

- Que las filas de la matriz no sean un numero par.
- Que contenga un *ship* adicional a, los que principalmente se declaran en los hechos.

Un problema que aparece, es que existe matrices que son idénticas, para solucionar dicho obstáculo, se procede a identificarlas con un número para hacerlas únicas. Al momento de llamar a la función *createBoard* se obtiene un número random que identifica a la matriz que será elegida para la partida.

Se dice que en este caso el predicado *createBoard* no crea una matriz en sí, ya que lo que hace es buscar similitud entre los parámetros entregados y los hechos, si estos últimos cumplen con las condiciones del usuario, se entrega la matriz., en caso contrario no la entrega. Se debe tener en cuenta que no es una matriz inválida necesariamente, solamente que no está dentro de los hechos declarados en la aplicación y eso produce que sea una matriz inválida.

2.1.2.2 Predicado checkBoard

Se encarga de verificar si la matriz cumple con las reglas fundamentales, ya que en la base de conocimiento de la aplicación se crearon 3 tipos de matrices que no cumplen con esas reglas fundamentales, es decir, que existen casos que al llamar al predicado el tablero será inválido.

Recibe como entrada el *Board* que el usuario desea verificar si es válido o no.

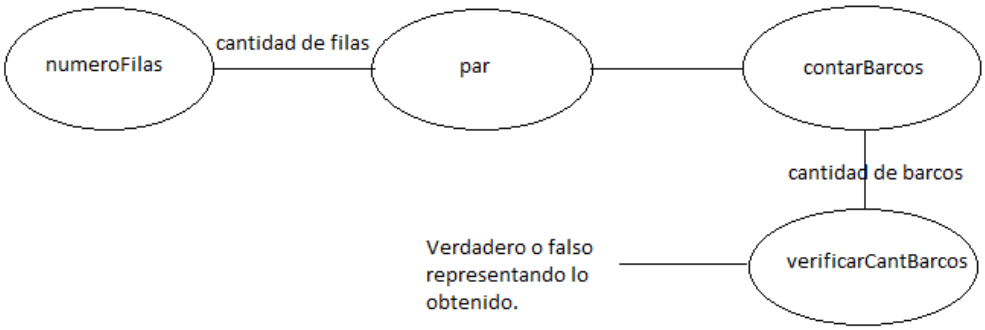

Diagrama 2.1.2.2-1: Diagrama del predicado checkBoard.

Como se observa en el diagrama de abstracción 2.1.2.2-1:

- Predicado *par*: Verifica si la cantidad de filas es par.
- Predicado verificarCantBoard: Verifica si la cantidad de barcos es consistente con los hechos de la aplicación

2.1.2.3 Predicado Play.

Se encarga de verificar si la jugada que va a realizar el usuario es válida.

Como entrada recibe: *BOARD*, Ship, Posiciones que corresponden a:

- *BOARD:* Es la matriz donde se van a comprobar si las posiciones son válidas. Considerar que el tablero para estas comprobaciones se divide por la mitad otorgado:
    - 1 x M hasta N/2xM.
    - N/2 x M hasta N x M.
- *Ship:* Barco con el cual va a realizar el ataque.
- *Posiciones:* Lista de coordenadas donde se realizó el ataque.

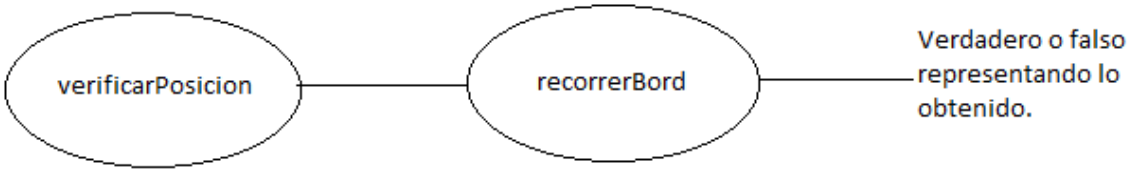

Diagrama 2.1.2.3-2: Diagrama de abstracción del predicado Play.

Como se observa en el diagrama de abstracción 2.1.2.3-2:

- *VerificarPosicion,* tiene el siguiente procedimiento que se aprecia en el diagrama 2.1.2.3-3:

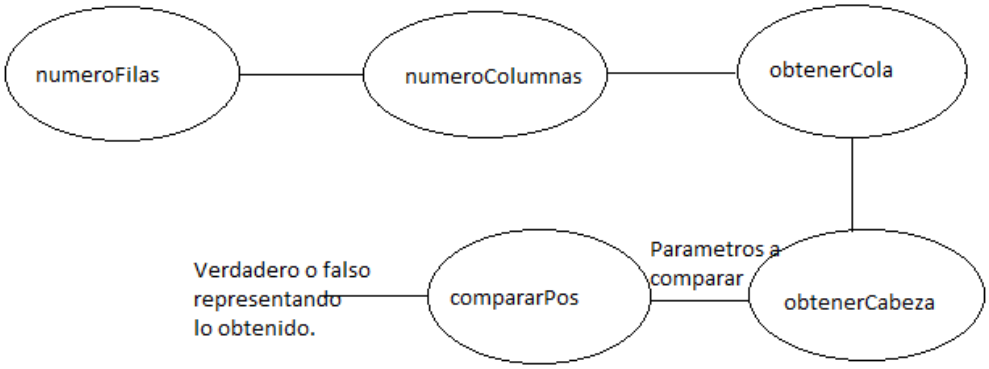

Diagrama 2.1.2.3-3:Diagrama de predicado verificarPosicion

- RecorrerBoard: Se encarga de ir verificando si se encuentra el *ship* con el cual se quiere realizar el ataque solamente por el *Board* del usuario, es decir, por N/2 x M hasta N x M del *Board.*

En caso que las dos comprobaciones anteriores entreguen como resultado verdadero, quiere decir que la jugaba que desea hacer el usuario es una jugada valida en caso contrario se considera una jugada fallida.

2.1.2.4 Predicado BoardToString

Se encarga de entregar la matriz que está confeccionada por una lista de lista, en un tipo de dato string.

Recibe como parámetro: *BOARD, BoardStr* que corresponden a:

- *BOARD:* Contiene almacenado la lista de lista que conforma a la matriz y será lo que se va a transformar a un tipo de dato string.
- *BoardStr:* Será donde se almacenará el string que se creará de la matriz.

2.1.3 RESULTADOS

Los resultados de esta aplicación se pueden ir observando al momento de ir haciendo el llamado a los predicados.

A continuación, se expondrán algunos de los resultados que entregan los predicados implementados:

En la figura 2.1.5.-6: Se puede apreciar cuando se crea un tablero de dimensiones 6 x 6.

```
2 ?- createBoard(6,6,6,BOARD),display(BOARD).
[[3,3,3,~,~,~],[~,~,~,2,2,~],[1,~,~,~,~,~],[~,3,3,3,~,~],[1,~,~,~,~,~],[~
,~,~,2,2,~]]
BOARD = [[3, 3, 3, ~, ~, ~], [~, ~, ~, 2, 2, ~], [1, ~, ~, ~, ~, ~], [~,
3, 3, 3, ~|...], [1, ~, ~, ~|...], [~, ~, ~|...]]
```

*Figura **Error! Reference source not found.**-5 : Llamado al predicado createBoard.*

En la figura 2.1.3-7: Se puede apreciar cuando se comprueba un *Board* si es válido para una partida, es decir, si cumple con las reglas fundamentales.

```
73 ?- createBoard(6,6,6,BOARD),checkBoard(BOARD).
BOARD = [[3, 3, 3, ~, ~, ~], [~, ~, ~, 2, 2, ~], [1, ~, ~, ~, ~, ~], [~,
3, 3, 3, ~|...], [1, ~, ~, ~|...], [~, ~, ~|...]] .
```

*Figura **Error! Reference source not found.**-6: Llamado al predicado checkBoard.*

En la figura 2.1.3-8: Se puede apreciar cuando se realiza una jugada dentro de un tablero, además de verificar si es una jugada válida.

```
72 ?- createBoard(6,6,6,BOARD),play(BOARD,1,[1,2]).
jugada valida
BOARD = [[~, 2, 3, 3, 3, ~], [~, 2, ~, ~, ~, ~], [~, 1, ~, ~, ~, 6], [~,
~, ~, 2, 2|...], [~, 3, 3, 3|...], [1, ~, ~|...]] 
```

*Figura **Error! Reference source not found.**-7: Llamado al predicado play.*

En la figura 2.1.3-9: Se aprecia cuando el *Board* que se está utilizando en la partida se transforma en un tipo de dato string.

```
75 ?- createBoard(6,6,6,BOARD),boardToString(BOARD,BoardString),display(BoardStri
ng).
~,~,~,2,2,~,1,~,~,~,~,~,~,3,3,3,~,~,1,~,~,~,~,~,~,~,~,2,2,~,3,3,3,~,~,~
BOARD = [[3, 3, 3, ~, ~, ~], [~, ~, ~, 2, 2, ~], [1, ~, ~, ~, ~, ~], [~, 3, 3, 3,
 ~|...], [1, ~, ~, ~|...], [~, ~, ~|...]],
BoardString = "~,~,~,2,2,~,1,~,~,~,~,~,~,3,3,3,~,~,1,~,~,~,~,~,~,~,~,2,2,~,3,3,3,
~,~,~" 
```

*Figura **Error! Reference source not found.**-8: Llamado al predicado boardToString.*

CAPÍTULO 3. CONCLUSIÓN

Los alcances y objetivos fueron cumplidos. Se cumplió con los requisitos obligatorios tanto como funcionales como no funcionales, además de realizar boardToString.

Una de los factores que se deben destacar es que se encontraron muchas similitudes con el laboratorio anterior que fue desarrollado en el paradigma funcional, en el cómo se deben recorrer las matrices y encontrar posiciones específicas dentro de una matriz. Pero no se tiene que dejar de lado que entre estos dos paradigmas existe gran diferencia, ya que el paradigma lógico se basa en resoluciones de problemas mediante declaraciones de hechos y reglas.

Por ultimo cabe señalar que los resultados en el laboratorio fueron exitosos, debido que, al momento de realizar pruebas de las funciones creadas se obtuvieron resultados exitosos. Sin dejar de mencionar que al comenzar hubo bastantes pruebas que eran erróneas, pero gracias a las herramientas que entregaba el intérprete *Swi-prolog* tales como *trace* y *debug*, facilitó en gran parte el encontrar y solucionar los errores que se estaban presentando.

CAPÍTULO 4. REFERENCIAS

- http://www.swi-prolog.org/download/stable
- http://seguridadensistemascomputacionalesp.blogspot.cl/2011/09/paradigma-logico.html
- https://es.scribd.com/doc/48831859/Logica-de-Primer-Orden