Universidad De Santiago De Chile
Facultad de ingeniería
Departamento de ingeniería informática

Informe de Laboratorio 2: Juego Dobble

Nombre: Cristobal Francisco Marchant Osorio
Profesor: Roberto González I.
Asignatura: Paradigmas de programación

9 de mayo del 2022

Tabla de contenidos

Pagina contenido

Introducción

Este informe habla sobre el desarrollo del laboratorio N°2 de paradigmas de programación, este consiste en que gracias al compilador SWI-Prolog, podemos usar el lenguaje de programación Prolog. En este informe se plantean los objetivos de explicar brevemente el problema planteado, darle un análisis a este problema, generar un diseño a la solución de este, mostrar aspectos de la implementación, instrucciones y como hacer usos de estas, mostrar resultados, una autoevaluación y una conclusión acerca del trabajo, todo esto enfocado al paradigma Declarativo-Logico, en donde como ya se mencionó se hará uso del lenguaje Prolog para realizar TDA-Cartas y un TDA-Game, cada uno de estos incluirá predicados para su implementación, dejando invitado al lector a ver cada implementación y que pueda entender el modo de uso de estas.

Descripción del problema

El problema planteado consiste en desarrollar el juego de cartas “Dobble”, en donde cada carta tiene un elemento en común. Sobre este tema tenemos 2 partes, en la primera parte se pide implementar la creación de un mazo y poder manipular este mazo para poder obtener valores como el total de cartas, si un mazo es válido, encontrar una carta en cierta posición, y en la segunda parte se nos pide crear una manera en donde el usuario pueda jugar con estas cartas, el usuario puede registrarse, elegir un modo de juego y elegir un mazo, además implementar funciones que permitirá a los jugadores poder pasar el turno, jugar al juego y finalizar el juego.

Descripción del paradigma

En este trabajo se trabajará bajo los estándares del paradigma lógico, el cual se basa en alto nivel de abstracción y posee una forma de trabajar de manera declarativa, esto quiere decir que establece relaciones entre las entidades.

Las entidades pueden ser entendidas como hechos y/o reglas, como ejemplo se tiene a la entidad “padre” que posee un “hijo”, es decir, existe una relación entre estas entidades que tiene como regla que es padre si posee un hijo, y los hechos pueden ser entendidos con el ejemplo mes(8,agosto), dando a entender que el mes 8 es agosto (atom), cabe destacar de que ambos predicados mencionados anteriormente deben usar minúsculas y para consultas se debe usar una mayúsculas, esto será explicado en el siguiente párrafo.

Los predicados mencionados con anterioridad son elementos a lo que le queremos dar un significado, estos van con mayúsculas. Para terminar un predicado siempre tenemos que termínalo con un punto y si el predicado tiene condiciones estas condiciones debe tener una coma a excepción de la última, la cual posee un punto, esto se le conoce como cláusulas de Horn. Un ejemplo de todo esto es: mes(MM,X):- MM<13, MM>0.
Otro ejemplo: mes(8,X). -> X = agosto, el sentido de esto es consultar para saber que elemento tiene que ser X para que el predicado sea verdadero.

Los atom son los elementos en los cuales basamos lo que queremos expresar, tomando en cuenta que estas sean escritas en minúsculas, por ejemplo, el poner mes(8,agosto). De esta manera Prolog sabrá que agosto es el mes 8.

Las consultas son preguntas que Prolog busca dar respuesta en base a hechos definidos previamente, por ejemplo, mes(8,noviembre). -> false, mes(8,agosto). -> true, esto con el sentido que anteriormente definimos a agosto como el mes 8.

Desarrollo

Análisis del problema

El desarrollo de esta actividad consta en la construcción de 2 TDAs, el TDA para la creación y modificación de cartas, y en el TDA sobre la creación del juego, siendo el primero de este TDA recurrido por el segundo, es por esto que se dispone a explicar el problema en donde podamos analizar los elementos requeridos en los predicados de cada TDA.

TDA cardsSet

Para entender la base del problema es necesario tener en cuenta las definiciones para las variables, por ejemplo, para la creación de las cartas se ingresa un número de elementos por carta, pero este número es un n+1, el n tiene que cumplir condiciones para que este sea válido, como es el caso de que este sea primo, porque si no fuera primo existirían cartas con más de 2 elementos en común o que no tengan ningún elemento en común y esto no es posible, ya que por la definición de las reglas del juego, cada carta tiene que tener un solo elemento en común.

Entonces tenemos los siguientes predicados:

Descripción	Elementos	Expresiones
Constructor	Lista de elementos, cantidad de elementos, máximo de cartas, seed (para aplicar azar), set de cartas	List X Int X Int X int X List
¿Es un mazo valido? (T or F)	Mazo de cartas	List
Una carta en cierta posición	Mazo de cartas, la posición, carta	Int X List
El total de cartas	Una Carta, total de cartas	List
De un mazo recortado encontrar las cartas restantes	Un mazo X mazo con cartas restantes	List X List
Transformar el set de cartas a un string	Un mazo y String	List X Str

TDA Game

Para entender la base de este problema es necesario tener el predicado de la creación de un mazo, este se basa en el supuesto de que el mazo ingresado es un mazo valido y que la cantidad de cartas sean suficiente para la cantidad de jugadores según el modo de juego empleado.

Entonces tenemos los siguientes predicados:

Descripción	Elementos	Expresiones
Constructor de la mesa	Cantidad de jugadores, el mazo de cartas, modo de juego, seed, juego	Int X List X Str X Int X List
Registro de jugador	Usuario, Juego, nuevo juego	Str X List X List
De quien es el turno	Juego y un usuario	List X Str
Permite jugar	Juego, Acción, Juego salida	List X List/Atom X List
Estado actual del juego	Juego y estado del juego	Juego X Str
Puntuación de un jugador	Juego, un usuario y un puntaje	List X Str X Int
Mostrar la mesa como string	Juego y String	List X Str

Diseño de solución

Para diseñar la solución al problema, se planteo la creación de ambos TDA (cardsSet y Game), con forma de lista y siguiendo la estructura propia de un TDA, la cual es una representación una breve descripción, constructores, predicados de pertenencia (True/False), selectores, modificadores y otro tipo de predicados.

TDA cardsSet

A continuación, se explicarán como funcionarán las entradas para el **mazo de cartas**, para el predicado del mazo se tomarán en cuenta 5 aspectos de entrada:

El primer es la lista de elementos, esta se planeo de manera en que mientras sea consultado el mazo, esta vaya reemplazando los números por los elementos del mazo.
El segundo parámetro es el número de elementos por carta, este aplico de manera (((NElem - 1)^2) + NElem) es así como podíamos saber cuántas cartas tendríamos en total.
El tercer parámetro fue el máximo de cartas, esto se pensó en cuanto se tuviera el mazo completo, exista una lista nueva de tamaño MaxC, logrando esto con un append(Lnueva,_,CS).
El cuarto parámetro es el Seed, el cual es un numero de entrada para un predicado que puede obtener un mazo desordenado, este funciona a base de el módulo del Seed sea la posición de una carta, luego esta carta se elimina y así hasta tener 0 cartas en el mazo original (ver anexo, figura 1).
El quinto parámetro es el mazo simplemente.

Para el caso de "**cardsSetIsDobble**" este se pensó en definir un predicado distinto para cada condición, estos son, que cada carta tenga la misma cantidad de elementos, que cada carta solo tenga 1 elemento en común con otra carta y por último que una carta tuviera solo elementos distintos, cuando se tuvieron estos predicados se usaron para crear el predicado, en donde si las 3 eran true, el resultado del "**cardsSetIsDobble**" es true, en cualquier otro caso el resultado será false.

En **cardsSetNthCard** se usó el predicado "Nth0" el cual va de 0 hasta el cardsSet-1.

En el caso de **cardsSetFindTotalCards** se pensó en contar la cantidad de elementos de la carta y aplicar ((n-1)**2)+n.

En el **cardsSetMissingCards** se hizo uso de utilizar el predicado constructor para generar un mazo nuevo, y de este mazo utilizar el predicado "substract" para obtener una lista con las cartas faltantes

En el **cardsSetToString** se pensó de manera en que primeramente una carta se transforme en un string y se agregue a una lista, luego cada carta que se convierto en string se le agregan parámetros estéticos, y para finalizar la lista con las cartas en string y con los parámetros estéticos los juntamos todos en un solo string gracias al atomics_to_string.

TDA Game

Para la creación del constructor del constructor del **dobbleGame** se pensó en definir una lista que funcionará de “mesa de juego” en donde este incluirá el número de jugadores, los datos de cada jugador (definidos por una TDA-player (inicialmente es un número (de Nplayers hasta 1), turno = 0 y puntaje = 0)) y dependiendo del modo de juego, se podrán ver las cartas en juego y las cartas restantes, también incluye la mesa de juego, el mazo de cartas, y el estado de juego.

Para la creación del **dobbleGameRegister** se basó en la definición de player, que encontraba el número de players agregaba gracias al predicado select, restaba 1 al número de players, cuando se intenta registrar a otro volvía a ocurrir esto hasta que el número de players sea 0, caso de que se intente agregar un nombre repetido o más playes de lo indicado al inicio, se arrojara falso.

Para la creación del Selector **dobbleGameWhoseTurnIsIt**, esta se basó en buscar quien estaba de turno según la lógica que la lista siempre vaya de izquierda a derecha y cuando llegaba al final volvía al primero, entonces cuando la comparaba a 2 usuarios del juego se veía cual era mayor, entonces el menor de estos 2 estaba de turno, pero si eran iguales se compararán el segundo con el tercero y si todos son iguales el turno seria del primero de la lista.

Para la creación del **dobbleGamePlay** y la implementación de sus predicados se tiene que en “Action” se pensó en que si el “Action” era null, este daría vuelta 2 cartas del mazo en la mesa. En caso de “Action” ser [pass] simplemente se aumentó en 1 el turno del jugador en turno, luego en el spotIt se pensó en que si el elemento que entregaba era coincidía con ambas cartas en el mazo este era true, entonces se sumaría un turno y se agregaría un punto, en caso de que estuviera malo seria false y solo pasaría el turno. Y por último si “Action” era finish, se hace una comparativa entre los puntajes y determina los ganadores y perdedores del juego, en caso de existir un empate entre todos, se declara empate, luego de estas definiciones se eliminan las cartas de la mesa y las cartas del mazo y se pasa el estado del juego a finish.
El diagrama de cómo funciona la mesa de juego junto a otras funciones establecida hasta este punto se muestra a continuación: (ver anexo, figura 2).

Para **dobbleGameStatus** está al ser un selector entrega el estado de la mesa si es que se lo piden.

En **dobbleGameScore** se hizo uso del predicado select para poder entregar el puntaje de un jugador.

Para **dobbleGameToString** se le dio un orden de modo de que cada parte de la lista “game” pudiera ser entendida simplemente leyéndola, además de que, si el juego estaba finalizado, esta entregaría también a los ganadores y perdedores.

Aspectos de implementación

En el siguiente apartado se darán a mostrar aspectos implementados que son relevantes a la hora de entender el desarrollo del trabajo.
Cabe destacar que se hizo uso de una lista única de aproximada 57 elementos, con los cuales se puede obtener hasta un set de cartas de 8 elementos por cartas, además de que se busca que sea trabajado así.

Estructura del proyecto

El proyecto presenta una estructura que se compone de 3 TDAs (CardsSet, Game y Players) y un archivo “main.pl” en donde se incluye el desarrollo del código junto a los ejemplos.

TDA - CardsSet: Es el encargado de la generación del mazo de cartas que cumple con los parámetros ingresados por el usuario, además de incluir funciones de pertenencias, modificadores, selectores y otras.

TDA - Player: Es un Pequeño TDA definido para la creación del perfil del usuario compuesto por un número, acompañado de 2 ceros (en donde uno representa el turno y el otro el puntaje) y una lista para algún modo de juego extra, implementando 2 predicados de pertenencia.

TDA - Game: Es el encargado de generar una mesa de juego en donde un usuario se pueda registrar y jugar, cabe destacar que este implementa función del TDA – CardsSet y del TDA – Player, también que al igual que el primer TDA, estas implementan predicados de pertenencias, modificadores, selectores y otras.

Bibliotecas empleadas

No se hizo uso de alguna biblioteca adicional.

Compilador

SWI-Prolog, version 8.4.2

Instrucciones de uso

Existen un gran número de ejemplos por cada función principal establecida, estos ejemplos se pueden observar en el final del archivo “main.pl”, de igual manera se mostrarán algunos ejemplos que sean comparables y que cuenten con una breve descripción y resultados esperados.

Para poder hacer uso de estos ejemplos tendremos que ejecutar el código en el terminal de SWI-Prolog (para VSCode usamos swipl en nuestro terminal).
En nuestro terminal escribimos consult(“main.pl”). y a continuación agregamos cualquier ejemplo dado sin “%”
Ej:

```
1 ?- consult("main.pl").
true.

2 ?- cardsSet([a,b,c,d,e,f,g,h,i,j,k,l,m,n],3,4,543523,CS).
CS = [[b, d, f], [a, d, e], [a, f, g], [a, b, c]].
```

Figura 3, Ejemplo cardsSet.

De esta misma manera obtendríamos más ejemplos de este estilo, tales como:

```
% dobbleGame con 3 jugadores, stackMode y un mazo de 3 elementos, con el max de cartas y ordenado
% cardsSet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],3,MAX,0,CS2), dobbleGame(2,CS2,"stackMode",1324,G2).
```

```
% missingCards de un mazo de 4 elementos, max 5 cartas y ordenada
% cardsSet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],4,5,0,CS3), cardsSetMissingCards(CS3, Mcards1).
```

```
% dobbleGame con 3 jugadores, stackMode y un mazo de 3 elementos, con el max de cartas y ordenado
% cardsSet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],3,MAX,0,CS2), dobbleGame(2,CS2,"stackMode",1324,G2).
```

Resultados y autoevaluación

A partir de los resultados obtenidos y poniendo a prueba los ejemplos propuestos en el archivo “main.pl” se cumple con éxito la totalidad de las funciones, en donde se pusieron a prueba los ejemplos para entregar este tipo de resultados.
A continuación, se presenta la autoevaluación.

Elementos por evaluar	Calificación
TDAs	1
TDA cardsSet - constructor	1
TDA cardsSet - IsDobble	1
TDA cardsSet - NthCard	1
TDA cardsSet - FindTotalCards	1
TDA cardsSet - MissingCards	1
TDA cardsSet - ToString	1
TDA game - constructor	1
TDA game - Register	1
TDA game - WhoseTurnIsIt	1
TDA game - Play	1
TDA game - Status	1
TDA game - Score	1
TDA game - ToString	1

Conclusión

Luego de terminar el proyecto se puede concluir que se cumplió con la totalidad de los objetivos planteados. Con respeto a los logros del proyecto también podemos afirmar que se cumplieron con éxito y se pretende trabajar de manera más ordenanza y mejor documentada, explicando un número mayor de ejemplos en próximos trabajos. Una de las complicaciones en relación con este trabajo fue el comprender el uso e implementación de los predicados, en donde gracias a lo visto en clases y tomar como apoyo el laboratorio pasado se pudo llegar a comprender e implementar, también cabe destacar que para algunos predicados planteados requirió de una gran cantidad de tiempo, pero fuera de estos tropiezos, se puede decir que se cumplió la totalidad del proyecto.

Sobre el paradigma declarativo lógico, gracias a la realización de este trabajo fue posible tener un mejor entendimiento sobre este, y pensar de manera más rápida en una solución hacia un problema.

Sobre SWI-prolog no hubo complicaciones, ya que se utilizó el compilador web, para una mejor administración de tiempo, y sobre la plataforma git no hubo ningún inconveniente y se intentó hacer como mínimo un comit diario.

Se espera que para próximos trabajos trabajar con más tiempo, planear la idea de la implementación desde antes y trabajar de manera más ordenada.

Bibliografía

Marcos Merino, 2020. El lenguaje Prolog: un ejemplo del paradigma de programación lógica
https://www.genbeta.com/desarrollo/lenguaje-prolog-ejemplo-paradigma-programacion-logica

Roberto González y Víctor Flores, 2022 Proyecto semestral de laboratorio.
https://docs.google.com/document/d/1tCVZBGScJbXFspQfu6WlDl3PsyCJUcqIgos5__DQz7k/edit#

Roberto González, 2020 Material online Mostrado en clases.

Anexo

```
%----------------------Predicado de Azar----------------------------------------------
% Modificador
% Dominio: Lista X Enetero X Lista
% Descripcion: predicado que permite randomizar una lista
azar(L,Seed,LOut):- azarito(L,Seed,[],LOut).

% Modificador
% Dominio: Lista X Enetero X Lista X Lista
% Descripcion: predicado que permite randomizar una lista
% Caso 1, SeedAux < N
azarito(Lista,Seed,L,LOut):-
    length(Lista, N),
    not(N=0),
    SeedAux is Seed mod N,
    SeedAux < N,
    nth0(SeedAux, Lista, L1),
    delete(Lista,L1,L11),
    append(L, [L1], L2),
    azarito(L11,Seed,L2,LOut),
    !.
% Caso 2, SeedAux >= N, aumenta el SeedAux en 1
azarito(Lista,Seed,L,LOut):-
    length(Lista, N),
    not(N=0),
    SeedAux is Seed mod N,
    not(SeedAux < N),
    Seed1 is Seed+1,
    azarito(Lista,Seed1,L,LOut),
    !.
% Caso 3, Lista vacia
azarito(Lista,_,L,L):-
    length(Lista, N),
    N=0,
    !.
```

Figura 1, Función azar.

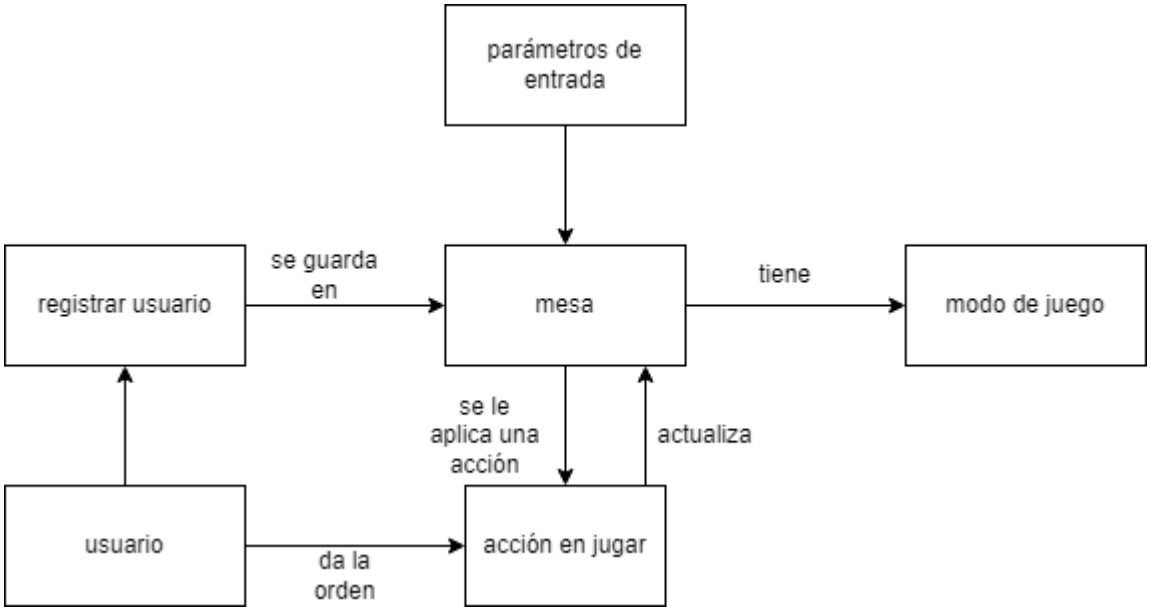

Figura 2, Diagrama del funcionamiento del game.