Informe Laboratorio 2:
Simulación en Prolog de juego “Dobble”

Nombre	RUT
1.- Alejandro Cortés Cortez	20.722.036-1

Profesor: Roberto González
Sección: A-1

Índice:

1. Introducción:

El presente informe trata respecto a la implementación en Prolog de un popular juego de mesa llamado “Dobble”, esto basado en el Paradigma Lógico, El cual será descrito en la sección 1.2, luego se detalla cuáles son los objetivos y que se busca conseguir en este proyecto, para después analizar a profundidad el problema planteado y diseñar una solución, que posteriormente es implementada y testeada, logrando así finalizar el proyecto para sacar las conclusiones pertinentes.

1.1. Descripción del problema:

Se pide implementar el juego “Dobble”, el cual es un juego de “cartas” que tiene distintas modalidades de juego, aunque para este caso, solamente estará disponible una de ellas, la cual será detallada más adelante. La principal complejidad de dobble viene respecto a las cartas que componen el juego, el cual se compone de 55 cartas, donde cada carta tiene en común un solo elemento con cualquier otra carta del juego, esto en el mundo de las matemáticas es conocido como plano afín finito, para lograr generar esto en Prolog, hay que dividir el problema en distintas partes, teniendo así un juego, los jugadores, un área de juego, un set de cartas, y las cartas por si solas.

1.2. Descripción del paradigma:

Todo esto se basa en el paradigma lógico, paradigma el cual de manera resumida propone que todo es una consulta, la cual se genera a través de predicados, y su respuesta solamente puede ser verdadero o falso. Para ahondar más en ello, primero es necesario saber que este paradigma pertenece a un tipo de programación denominada programación declarativa, esto debido a que la principal pregunta que debes hacerte al programar en este tipo de programación, es él “¿Qué?”, buscando así, obtener la solución, mas no enfocarse demasiado en el proceso detrás de ello, como seria en la programación imperativa, la cual es la más reconocida o común dentro del mundo de la programación, y se enfoca principalmente en él “¿Cómo?” hacer algo, aunque a pesar de que la programación declarativa no es la mas conocido ni popular, muchos programadores frecuentemente la utilizan al trabajar con bases de datos, ya que lenguajes como SQL trabajan de esta manera. Sabiendo esto, podemos entrar mas a detalle con el paradigma lógico, este paradigma principalmente requiere tener algo llamado “base de conocimiento”, que por decirlo de otra manera, describe el universo del lenguaje, por lo que la única manera que algo exista dentro de este paradigma, es que sea ingresado dentro de esta base de conocimiento, y para lograr esto, hay que hacerlo a través de predicados, teniendo así, principalmente dos tipos de predicados, aquellos que son “hechos”, los cuales son cosas reales, axiomas dentro del universo lógico, cosas que no se pueden cuestionar y son siempre verdad, como también tendremos “reglas”, que son aquellos predicados que ayudan a modelar la base de conocimiento, a estos, se les dan condiciones que se deben cumplir para que algo sea verdadero, con esto, debemos tener mucho cuidado, ya que hay que definir bien las condiciones a entregar, para así, obtener lo que queremos, puesto que

una mala definición de los límites puede generar reglas erróneas o que entreguen cosas que no deseamos, además, debemos ser cuidadosos con las negaciones, ya que a veces hay cosas que podrían ser verdad, pero no estar definidas en la base de conocimiento, y esto nos llevaría a que según la negación, sean verdaderas, pero realmente no es algo verdadero, sino que simplemente no se tiene el conocimiento, como se mencionaba al principio, este paradigma solo busca que algo sea verdadero o falso, basándonos en eso surgen las variables, estas nos indican que valor debe tener una variable para que un predicado se vuelva verdadero, esto siempre dentro de los límites que tenga definidos la base de conocimiento, además, para cumplir esto, se ocupa la lógica (gracias a ello denominamos al paradigma como paradigma lógico), siendo esta, la principal herramienta para obtener resultados, donde tenemos 2 herramientas principales, que serian la "," que representa un and, y el ";", que representa un or, entre muchos más.

Asociándolo más al lenguaje Prolog en sí, tenemos cosas importantes a destacar, como son las cláusulas de Hornn (manera de expresar las reglas), estas se expresan de la manera predicado(XX) **:-** … , y significa que el predicado es verdadero si, solo si lo que está a la derecha se cumple, estos predicados deben iniciar siempre con una minúscula, ya que cuando algo inicia con mayúscula, en prolog significa que es una variable, además , cabe destacar que este lenguaje funciona principalmente con recursividad, lo que nos ayuda a generar programas grandes, y para lograr esta recursividad trabaja con backtracking, lo cual es usado principalmente en la búsqueda de soluciones, para después generar una unificación de estas.

1.3. Objetivos

Como principal objetivo hay que adaptarse a una manera lógica de generar la solución de problemas, evitando en la mayor medida posible seguir con prácticas heredadas de otros paradigmas, logrando así pensar más en él "¿Qué?", derivado de esto, se busca también seguir adentrándose en maneras recursivas de encontrar soluciones para un problema, e implementándolas de manera correcta en prolog, generando así el último objetivo, que consiste en aprender, practicar y resolver el problema dado, como también, distintos problemas en prolog, buscando implementar todos los predicados utilizados desde una base recursiva, tratando de evitar lo más posible predicados prefabricados, para lograr así comprender como funcionan aquellos.

2. Desarrollo:

2.1. Análisis del problema

Para comenzar a analizar esta problemática, debemos tener en cuenta que no se tienen datos almacenados en memoria, como tampoco hay variables que se puedan modificar, ya que las denominadas variables acá, solamente se pueden unificar una vez, por ende todo lo que sucede en este trabajo va realizándose al momento, con las instrucciones entregadas y normalmente se recibe el resultado en una variable extra, siendo esta la

variable con la que el resto de reglas se cumple, pero no queda almacenado en ninguna parte, lo cual implica que, además, nada se modifica, sino que se reconstruye y unifica. Luego, analizando la principal problemática, tenemos de manera más general, a más específica, distintas componentes, donde primero que nada se requiere tener el Juego (Game), donde serán asignados y registrados los Jugadores, quienes son aquellos que deberán jugar el juego, ya que más específicamente hablamos de “Dobble”, lo que el jugador deberá hacer consiste en 4 instrucciones primordiales, que son retirar cartas de un stack de cartas (denominado posteriormente como Área), pasar (no hacer nada, se podría dar en el caso de que nadie encuentre similitudes), Verificar si un elemento está en las cartas, o terminar el juego, ahora, dependiendo de la modalidad de juego, puede variar cada instrucción, en este caso se implementa una modalidad, la cual llamaremos “stackMode”, este modo consiste en ir sacando de a dos cartas desde el stack (que contendrá un set completo de cartas) al área de juego, y un jugador deberá encontrar un elemento en común entre ambas cartas, si la encuentra se las queda el jugador y aumenta así, su puntaje, esta modalidad de juego concluye cuando el stack de cartas queda vacío o también puede ser terminado en cualquier momento, gana aquel jugador con mayor puntaje, o sea, más cartas en su poder. Además del juego y sus modalidades, como se pudo apreciar, se necesitan jugadores, los cuales se deben registrar con un nombre único y así luego estarán habilitados para jugar, cada jugador tendrá su mazo de cartas, que puede estar vacío o con cartas dependiendo de la modalidad, además será necesario tener un área de juego, donde estarán las cartas volteadas, estas cartas provienen de un set de cartas, que será formado a través de un algoritmo *(Dore, 2021) ,* este set de cartas está formado claramente por cartas, las cuales también deberán ser parte del diseño del proyecto, esto será a grandes rasgos la manera en la que se representara el problema según los requerimientos.

2.2. Diseño de la solución

Para la implementación será necesario que cada uno de los componentes principales señalados anteriormente, sean un TDA, que es la representación asignada para poder ser trabajada en el software. Como principal herramienta tendremos las listas, por lo que la mayoría de representaciones será con Listas, o Listas de Listas. Mostrando desde lo más específico a lo más general, ya que algunos dependen de otros TDA. Gracias a las facilidades de Prolog y su gran manejo de listas, habrá algunos TDA de los que serán descritos, que a pesar de ser usados y estar ahí, no serán implementados, esto debido a que no aportarían nada en la implementación, al contrario, generarían más problemas, por lo que tendremos los siguientes TDA:

2.2.1. TDA card

- Representación : Es una lista que contendrá los elementos que le sean entregados, pudiendo ser cualquier tipo de elemento que requiera el usuario, en caso de no tener elementos, será una lista vacía [].
- No implementado.

2.2.2. TDA cards

- Representación: Es una lista que contendrá uno o varios Card, en caso de no tenerlos, será un [].
- No implementado.

2.2.3. TDA cardsSet

- Representación: Es una lista, que contendrá una lista de elementos (que pueden ser de cualquier tipo), seguido de cards (un conjunto de card), estas cartas serán las cartas del juego, generando así, un set de cartas válido de dobble.
- Predicados importantes: El primero seria cardSet, el cual según un número genera un set de cartas completo y válido, cardsSet, el que se encarga de generar un set de cartas, asignarle elementos, revolver las cartas, y también limitar la cantidad de cartas, cardsSetIsDobble, el cual verifica si un cardsSet recibido, es un set de cartas doble válido, esto lo hace verificando las reglas principales del set de cartas.
- Tiene más Predicados, los cuales puedes ver detallados en el anexo (*Tabla de Predicados)*.

2.2.4. TDA Area

- Representación: Es una lista, que contiene Cards, las que serán extraídas desde un cardSet, se utiliza para representar las cartas que estarán visibles al momento de jugar (cartas volteadas).
- No implementado.

2.2.5. TDA Player

- Representación: Es una lista, que contiene un nombre, un entero que representa el puntaje del jugador, el cual aumentara cuando sea requerido según en el modo de juego, y un cards que serán las cartas del jugador.
- No implementado.

2.2.6. TDA Players

- Representación: Es una lista, que contiene uno o varios player, en caso de no tenerlos es un [].
- No implementado.

2.2.7. TDA Game

- Representación: Engloba todo lo anterior para poder ser utilizado como es requerido, Es una lista, que contiene Area, un cardsSet, un entero que representará el máximo de jugadores, un entero que representará de quien es el turno de jugar, un players, un string que dirá cuál es el estado actual del juego, una string que representa el modo de juego y una semilla que será utilizada para aleatorizar números.
- Predicados importantes: dobbleGame, el cual crea un Game inicial, hace de constructor tanto para el game como para el area y players, dobbleGameRegister, que sirve para añadir jugadores a un game, siempre y cuando no supere el número máximo y el jugador no este previamente inscrito y dobbleGamePlay, que es la

función encargada del juego en sí, llamando así al modo de juego correspondiente, con una acción a realizar.
- Tiene muchos predicados útiles, que serán todos descritos en el anexo.

2.2.8. Extras

- Hay funciones extra que servirán en distintas partes, por ende no pertenecen a ninguno de los TDA.

2.3. Aspectos de implementación

Para este trabajo se usó como IDE Vscode, esto debido a que el IDE nativo de Swi-Prolog era poco eficiente y tenía errores básicos, este IDE se utilizó con la versión 8.4.2.1 de Swi-Prolog, Además se usó GIT para el manejo de versiones del proyecto, logrando así mantener un registro del constante trabajo efectuado, esto debido a que se trabaja en distintos lugares y distintos dispositivos, por ende es muy útil para mantener un orden , sirviendo para ver que cosa se modificó en cada momento, también sirve bastante en caso de tener algún error, logrando así tener un código anterior a este error. Se trabajó en un solo archivo, sin uso de bibliotecas ni librerías externas.

2.4. Instrucciones de uso

2.4.1. Ejemplos de uso

Para comenzar, lo primero que se debe hacer es ingresar en la consola de Swi-Prolog, el siguiente predicado *set_prolog_flag(answer_write_options,[max_depth(0)]).* esto, para conseguir una completa visualización de las listas, luego se carga el archivo correspondiente, para que se carguen sus predicados, a lo que prolog devolverá un true indicando que todo fue cargado de manera correcta.

Estando ya cargado el programa hay que ir al final del archivo, donde se encuentra una sección señalada como EJEMPLOS, donde tenemos mínimo 3 ejemplos para cada uno de los predicados principales (también llamados funcionales), para ejecutarlos simplemente se debe llamar al predicado con una variable como argumento, donde se recibirá lo que corresponda, todos estos siguen un orden de uso básico, que está delimitado por el modo de juego implementado, donde primero es necesario crear el cardsSet, al cual se le deben pasar elementos o un [] (para que se genere con números), luego se le entrega el número de símbolos por carta, este número puede ser cualquiera, pero para que genere un cardsSet válido como dobble es necesario que el sí le restas 1 al número, el resultado sea un número primo, luego se le ingresa el número de cartas que quieres, debe estar dentro de los límites según el número de símbolos, siendo como máximo él $(MaxSimbolos- 1)^2 + (MaxSimbolos - 1) + 1$, por ejemplo, para 3 símbolos por carta, el máximo de cartas es $2^2+2+1 = 7$, luego se le entrega una seed, y una variable donde debe retornar el CardsSet, ya con este cardsSet podremos crear el juego, para ello el primer paso es crear un Game, al cual debes entregarle el máximo de jugadores, el cardsSet, el modo de juego (que en este caso debe ser stackMode), una seed, y una variable donde se recibirá el Game, con

esto ya listo, deben registrarse los jugadores, con nombres únicos cada uno, además, no debe superar el tope impuesto con anterioridad, y a con esto es posible jugar, siempre siguiendo las siguientes reglas, debes llamar al dobbleGamePlay primero ingresando el juego actual, una acción a realizar, y una variable donde tendrás el juego luego de realizada la acción, estas acciones pueden ser [], la cual se encarga de comenzar el juego y también hacer el volteo de cartas, que para este modo de juego, se encarga de sacar 2 cartas del stack y ponerlas en el area de juego, ahí, es cuando alguno de los jugadores debe ingresar la acción spotit seguida de su nombre y el símbolo que según él, está en ambas cartas, si esto es correcto se le entregan las cartas y se le suma el puntaje, si no, se devuelven al stack las cartas del area y debe voltear las siguientes, y así se debe seguir jugando hasta que no haya cartas suficientes en el stack o hasta que alguien decida ingresar la acción finish que se encarga de finalizar el juego, impidiendo así seguir jugando, estando el juego finalizado puedes saber el ganador cuando conviertes el juego en un string.

2.4.2. Posibles errores

A lo largo del proceso se solucionaron todos los errores que fueron encontrados, por ende no se lograron identificar errores posibles si se utilizan bien los predicados, podrían surgir errores al emplear los predicados de distinta manera a como se plantean, pero no fue visto ningún caso particular, algo que no es un error, pero podría no ser perfecto, es que solo selecciona un ganador, por mas que dos jugadores puedan tener un puntaje similar.

2.5. Resultado y autoevaluación

Los resultados de esta experiencia fueron muy positivos, ya que no se encontraron errores, se recibe siempre el resultado esperado, además se logró la implementación de todos los requisitos, las pruebas pueden ser resumidas en los ejemplos, además sé probo con los que venían en el enunciado, sin tener errores en ninguno de ellos, y también fueron probados varias veces al momento de ser creados, y cada error que fue encontrado se logró solucionar, por lo mismo, la autoevaluación está al 100% con el puntaje máximo.

3. Conclusión

Al comenzar a trabajar con este paradigma fue muy difícil comprenderlo, es bastante distinto a lo tradicional, además, también es bastante distinto del paradigma funcional, el cual fue trabajado con anterioridad, a pesar de ambos ser englobados dentro de la categoría de paradigma declarativo, y de trabajar con recursividad, tanto su sintaxis como manera de trabajar son demasiado diferentes, añadiendo, que en el paradigma lógico no tenemos funciones, que era la base del paradigma funcional. Pero luego de trabajar un tiempo y estudiar el lenguaje prolog, se convierte en algo muy simple y cómodo, es un lenguaje y un paradigma muy grato de trabajar, que ayuda mucho al programador, ayudando realmente a enfocarse en la solución más que en el procedimiento, esto debido a que el paradigma funcional también nos genera esa idea, pero en la realidad es distinto, era

mucho más similar a la programación imperativa de siempre, pero en el caso del paradigma lógico, esto ya no es así, realmente puedes enfocarte en la solución, ya que gran parte del problema lo soluciona prolog con backtracking, lo más difícil es idear e implementar de manera correcta las reglas, y entender que el lenguaje no sabe nada más que aquello que le entregas a su base de conocimiento, fuera de esto, también es genial su trabajo con listas, algo que dentro del paradigma funcional era bastante tosco, pero acá se tiene un gran manejo de listas, pudiendo así, ahorrarnos una gran cantidad de trabajo.
Por lo que finalmente se considera un gran lenguaje de programación, como así también un gran paradigma, con herramientas muy cómodas, además de ser muy útil cuando se sabe usar de manera correcta.

4. Bibliografía

Gonzalez, R. (2022, marzo).”Proyecto semestral de laboratorio”. Recuperado 19 de abril de 2022, de https://docs.google.com/document/d/1tCVZBGScJbXFspQfu6WlDl3PsyCJUcqIgos5__DQz7k

Dore, M. (2021, febrero 15). The Dobble Algorithm. Medium. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, *104*, 616-621.

Clocksin, W. F., & Mellish, C. S. (2003). Programming in prolog. Springer Science & Business Media. de https://books.google.es/books?id=VjHk2Cjrti8C.

5. Anexos

5.1. Tabla de Predicados

Extra			
Tipo de Predicados	Tipo de meta	Predicado	Descripción
Auxiliares	Secundarias	addListToList	Añade una lista de listas a una lista de listas.
Auxiliares	Secundarias	estaen	Busca si un elemento está en una lista y cuantas veces esta
Auxiliares	Secundarias	posicionE	Retorna el Elemento que está en una Posición, o también la Posición de un Elemento
Auxiliares	Secundarias	listToString	Convierte una lista en un string, separado por un -
Auxiliares	Secundarias	cardsToString	Convierte una lista de listas en string.
Auxiliares	Secundarias	largo	Devuelve en el segundo argumento el largo de una lista
TDA cardsSet			
Tipo de Predicados	TIpo de meta	Predicado	Descripción
Parte del constructor	Secundarias	firstCard	crea la primera carta del mazo.
Parte del constructor	Secundarias	eachNcard	genera cada una de las cartas "n".
Parte del constructor	Secundarias	nCards	llama al eachNcards con distintos parámetros y lo unifica.
Parte del constructor	Secundarias	eachNNcard	genera cada una de las cartas "nn" o n^2.
Parte del constructor	Secundarias	eachJcycle	llama a eachNNcard por cada cambio de j y lo unifica.
Parte del constructor	Secundarias	nnCards	llama a eachJcycle modificando el i en cada llamada, y lo unifica.
Parte del constructor	Primarias	cardSet	Produce un set de cartas dobble válido, y completo, llamando a las funciones anteriores y unificándolas.
Parte de la pertenencia	Secundarias	serepite	Verifica si un elemento se repite dentro de una lista
Parte de la	Secundarias	repite	Verifica si una lista tiene elementos repetidos

pertenencia			
Parte de la pertenencia	Secundarias	repiteCartas	Verifica si en una lista de listas hay alguna lista con un elemento repetido
Parte de la pertenencia	Secundarias	unaVez	Verifica que un elemento este "Count" veces en una lista
Parte de la pertenencia	Secundarias	elementosComun	Determina cuantos elementos en común tienen dos listas
Parte de la pertenencia	Secundarias	comunCardACards	Aplica el predicado anterior comparando una lista con una lista de listas
Parte de la pertenencia	Secundarias	comunCards	Aplica el predicado anterior para todas las listas de una lista de listas, comparando cada Carta con el resto de las cartas
Pertenencia	Primarias	cardsSetIsDobble	Verifica si una lista de listas es un cardsSet válido
Modificador	Secundarias	addFinal	Añade un Elemento al final de una lista
Auxiliares	Secundarias	myrandom	Genera un número "aleatorio" a partir de otro
Modificador	Secundarias	randomizar	Según un número X, "revuelve" aleatoriamente una lista de listas
Selector	Secundarias	cortar	Deja solamente una X cantidad de Listas, eliminando el resto
Auxiliares	Secundarias	linkearC	Asocia una lista de elementos con una lista de números, basándose en la posición de cada elemento.
Modificador	Secundarias	linkearA	Aplica lo anterior para cada lista de una lista de listas
Constructor	Primarias	cardsSet	Crea un CardsSet válido, con E elementos en cada carta, con una cantidad C de cartas, revolviéndolas según una Seed, y donde sus elementos serán los entregados en L.
Auxiliares	Primarias	cardsSetToString	Crea un string de un CardsSet
Selector	Primarias	cardsSetNthCard	Devuelve la Carta en una posición X de un cardsSet
Selector	Primarias	cardsSetFindTotalCards	A partir de una Carta, calcula cuanto sería el máximo de cartas válidas para dobble
Auxiliares	Secundarias	isCardEqualCard	Compara si una lista tiene exactamente los mismos elementos que otra lista
Selector	Secundarias	isCardInCards	Revisa si una lista está en una lista de listas, usando el predicado anterior Para hacer la comparación
Auxiliares	Secundarias	notInterseccion	Crea una lista de listas, donde sus elementos son todos los que están en Cs, pero no están

			en CAC.
Auxiliares	Primarias	cardsSetMissingCards	Recibe un cardsSet válido, pero incompleto, generando así, un CardsSet alternativo El cual si debe estar completo, y después en Cs2 unifica todas las cartas que faltan para que el Cs sea un CardsSet completo
TDA Game			
Constructor	Primarias	dobbleGame	Crea un Game inicial.
Selector	Secundarias	getPlayers	obtiene una lista de "Players" (Tda no implementado)
Modificador	Secundarias	setPlayers	Cambia la lista de Players de un Game
Selector	Secundarias	getArea	obtiene el "Area" (es una lista de listas) de un Game
Modificador	Secundarias	setArea	Cambia el Area de un Game
Selector	Secundarias	getMode	Obtiene el string que representa el modo de un Game
Selector	Secundarias	getCards	Obtiene las cartas que están en el "stack" de un Game
Modificador	Secundarias	setCards	Modifica las cartas del mazo
Selector	Primarias	dobbleGameStatus	Muestra cuál es el estado del Game actual
Modificador	Secundarias	setStatus	Cambia el estado del juego
Selector	Secundarias	isPlayerInPlayers	Verifica según el Name si un Player está en una lista de Players
Modificador	Primarias	dobbleGameRegister	Registra un nuevo Player en el Game
Selector	Primarias	dobbleGameWhoseTurnIsIt	Dice de quien es el turno (No aplica para el modo de juego implementado)
Selector	Secundarias	scorePlayer	Muestra el Score de un Player segun su Name
Selector	Primarias	dobbleGameScore	Muestra el Score de un Player segun su nombre.
Auxiliares	Primarias	dobbleGameToString	Genera un string para visualizar el Game.
Selector		whoWin	
TDA Game - Modo de juego			
Selector	Secundarias	isElementInAllArea	Verifica si un elemento dado está en todas las listas del Area.
Modificador	Secundarias	modifyPlayerForSM	Al Pls le añade las cartas del Area a un jugador que coincida con su Name, y le suma 1 al Score, esto ya que está pensado para

			este modo de juego, esto queda en el Pls2.
Modificador	Secundarias	stackMode	modo de juego, verifica que hacer según cada Action.
Modificador	Primarias	dobbleGamePlay	Según el modo de un Game, llama a ese modo con el Action correspondiente, es el principal predicado para jugar.