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

INFORME DE LABORATORIO 2:
SIMULACIÓN DE DOBBLE EN PROLOG

Nombre: Bastián Escribano
Profesor: Gonzalo Martínez
Asignatura: Paradigmas de Programación

09 de mayo 2022

Tabla de Contenidos

1. INTRODUCCIÓN

Un paradigma se entiende como un conjunto de intuiciones, de parecer y de modos de emprender la realidad. Los paradigmas están en todas partes y en el mundo de la programación no es la excepción. Existen quizá miles de lenguajes de programación, los cuales probablemente tengan una forma diferente de llamar a sus métodos (o funciones), donde cambie hasta inclusive su sintaxis, es por esto por lo que, al dominar un lenguaje de programación, al cambiar a otro lenguaje, cuesta y es casi volver a aprender desde cero. Sin embargo, existen unos cuantos paradigmas de programación los cuales, para ser más exactos, declaran la gramática en la cual se está programando, esto significará que sea más fácil adaptarse a un nuevo lenguaje de programación. En este informe se introducirá, se planteará un problema y su respectiva solución a un paradigma en particular, el cual es el paradigma lógico de programación. Específicamente, será solucionado en el lenguaje de programación lógico prolog.

1.1 DESCRIPCIÓN DEL PROBLEMA

Se pide desarrollar una simulación de Dobble, El juego **Dobble** y en particular la generación de las piezas se rige por propiedades matemáticas dentro de los cuales destacan los números primos. Para implementar esto, se deben tener algunos elementos en cuenta:

cardsSet: En base a parámetros que se ingresen, se debe generar un mazo de cartas para que sea un juego válido. El juego Dobble es un juego matemático y matemáticamente tiene que estar bien construido para que tenga sentido.

Game: Se necesitan jugadores, modos de juegos y un mazo para empezar un juego. **Predicados :** Son las predicados que registran usuarios y verifican los turnos, la creación de cartas y los diferentes modos de juego.

Todo debe ser implementado utilizando el Paradigma Lógico, a través del lenguaje de programación Prolog.

DESCRIPCIÓN DEL PARADIGMA

El Paradigma Lógico se basa en la definición de reglas lógicas y es un paradigma declarativo. Contiene una base de conocimientos, la cual es un archivo que contiene los predicados del programa y esto es “todo el mundo” que conoce. Los predicados son cosas que se quieren decir y estos están definidos por clausulas (hechos y reglas).

En el paradigma lógico se tienen algunos elementos bastante importantes a la hora de construir un código utilizando este paradigma, como, por ejemplo:

Backtracking: Al momento de realizar una consulta pueden existir más de un hecho a la base de conocimiento, por lo tanto si al momento de realizar una consulta existe más de una coincidencia en la base de conocimiento, prolog realiza una vuelta atrás (backtracking) y devuelve el siguiente de los conocimientos que presenta la consulta.

Backforward: El backforward resuelve los predicados cuando estos hayan quedados pendientes, es muy similar a una recursión de pila. Cuando existen predicados (o estados) pendientes, realiza la unificación.

Unificación: Es un mecanismo que se emplea como paso de parámetros bidireccional en la ejecución de programas, por lo que resulta imprescindible conocerlo y saber calcular con soltura el unificador de dos expresiones.

Átomo: son aquellas cosas sobre las que basa el conocimiento que queremos expresar (Se escriben en minúsculas).

Predicado: Los predicados son las cosas que queremos decir. Los resultados o variables van en mayúsculas.

Consultas: Son aquellas preguntas que se hacen mediante consola y donde prolog busca en la base de conocimientos para entregar true o false.

Hechos: Cada una de las sentencias se “iguala a” unidades de información de una base de conocimiento. Los hechos y reglas deben terminar con un punto.

Además, se debe evitar el Problema de mundo cerrado que se puede generar al negar un predicado o un hecho y a su vez que el resultado que se espere no se haya definido por lo que no es nada de la base de conocimiento, que en teoría podría ser cualquier cosa. Cabe destacar que cuando prolog da falso a una respuesta, no significa que sea un falso absoluto, si no que cuando pasa esto, significa que se dio el caso en que prolog no fue capaz de encontrar un hecho y/o resultado que satisfaga la pregunta y por lo tanto, retorna falso.

Para la base de conocimientos, no es necesaria que esta contenga todo un conjunto de información, si no que contenga solo la información necesaria para satisfacer una respuesta a un problema.

Los hechos y/o reglas a su vez son definidos por “individuos” o “átomos” que son una representación de lo que se quieres definir como base (ej. name(bastian)). Se debe tener en cuenta que tanto hechos como reglas comienzan con minúsculas.

1.2 OBJETIVOS

Como objetivos del proyecto se tiene aprender sobre el paradigma y la programación lógica, para así obtener la habilidad de programar de otra forma distinta a la que se tiene costumbre actualmente. Otro objetivo es programar correctamente en prolog y aprender a utilizar las herramientas de este lenguaje para completar el proyecto de laboratorio y así poder tener una base para los futuros laboratorios y otros proyectos que en un futuro se desarrollen con la **programación lógica.**

2. DESARROLLO

2.1 ANALISIS DEL PROBLEMA

Matemáticamente, es posible generar un mazo con cartas que contengan una cierta cantidad de símbolos y que se repita por lo menos 1 símbolos en cada carta diferente del mazo, a esto lo llamaremos **el factor de repetición**. Esto es posible a una conexión matemática que se basa en números primos y en potencia de números primos. Este factor es el más importante y su nombre es **orden**, por lo tanto, el orden debe ser necesariamente un número primo o una potencia de un número primo para que se cumpla que exista al menos una repetición por carta.

Por ejemplo, el número 2 es un número primo, por lo tanto el orden puede ser 2. Ahora, a través de un cálculo matemático es posible determinar la cantidad máxima de cartas que pueden ser generadas en base a ese orden y la cantidad de símbolos que debe tener el mazo para que se cumpla el factor de repetición.

$$cardsSet(getOrden) = getOrden^2 + getOrden + 1$$

El cardsSet será la cantidad de cartas asociadas al orden y también la cantidad de símbolos que deberá tener el cardsSet. Por lo tanto, nuestro mazo estará compuesto de

$$cardsSet(2)=2^2+2+1=7$$

La cantidad de símbolos por carta está determinada por

$$cantSímbolos(getOrden) = getOrden + 1$$

Sin embargo, es posible determinar la cantidad de cartas y símbolos totales a través de otra manera matemática

$$cardsSet(getSímbolosPorCarta) = getSímbolosPorCarta^2 - getSímbolosPorCarta + 1$$

De ambas formas será posible encontrar la cantidad total de cartas y símbolos

$$cardsSet(3)=3^2 -3+1=7$$

Como se habla de una cantidad máxima de símbolos necesariamente por carta, no se debe descuidar el factor de repetición para que exista la conexión entre carta y carta.

En la figura 1. Es posible observar tras el ejemplo inicial el patrón de repetición que podría tener el mazo con cartas que contengan 3 símbolos **diferentes**.

	#1	#2	#3	#4	#5	#6	#7
A	1	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	1	0	1	0
E	0	1	0	0	1	0	1
F	0	0	1	1	0	0	1
G	0	0	1	0	1	1	0

Figura 1: cardsSet con 3 símbolos por carta (orden 2).

Además de la generación del mazo, se debe generar el juego, para esto se deberán más funcionalidades dentro de la generación del mazo y del propio juego.

- **CardsSet:** Corresponde a un conjunto de asociaciones, donde un usuario genera un mazo de juego a partir de una serie de parámetros a definir. Posteriormente el cardsSet será útil para un juego.
- **game:** Corresponden a aquellos que interactúan con el juego, registrándose en esta para luego poder realizar las operaciones disponibles. Por ejemplo, iniciar un juego.

  Para poder implementar a los elementos mencionados anteriormente en Prolog, se pueden representar de la siguiente manera:

- **cardsSet:** Una lista que contiene una lista de símbolos, el número de símbolos, la cantidad de cartas (que en caso de que sea negativa, se debe generar la cantidad máxima posible) y un predicado que sea capaz de generar un número aleatorio. (Elements (list) X numE(int) X maxC(int) X Seed (int) X CS (list))
- **game:** Una lista que contiene una lista que contiene el número de jugadores, un cardsSet, el modo de juego y un predicado que sea capaz de generar un número aleatorio: *numPlayers(int) X cardsSet X mode (string) X seed (int) X game (TDA Game)*

Cada **TDA** (Tipo de Dato Abstracto) está adentro de un archivo único, por lo tanto solo bastará con consultar al único y mismo archivo .pl

2.2 DISEÑO DE LA SOLUCIÓN

Para diseñar la solución, no solo es necesario utilizar los tipos de datos nativos de prolog, sino que también se deben crear tipos de datos específicos, a través de lo que se conoce como TDA, utilizando la siguiente estructura: Representación, Constructores, Predicados de Pertenencia, Selectores y otros predicados asociados al TDA.

Los TDAs creados son utilizados para la construcción de las operaciones de otros TDAs y para las operaciones obligatorias y opcionales. Los TDAs más importantes son:

2.2.1.1 TDA cardsSet

- Representación: Cada vez que se haga ingreso de las variables al predicado respectivamente, se hará entrega de un mazo de cartas listo para iniciar un juego. Este será construido a través de los elementos que reciba la función constructora del cardsSet.
- Constructor: Dado una lista de elementos, el número de símbolos por carta, la cantidad máxima de cartas y una función que genera un número aleatorio, se crea una lista que contiene una lista con símbolos correspondientes a un set de cartas válido.
- Función de Pertenencia, Selectores, Modificadores y Otras Funciones: *ver tabla 1 en anexos, página 12.*

2.2.1.2 TDA game

- Representación: Para iniciar un juego se necesita la cantidad de jugadores, un mazo, el modo de juego y un número aleatorio que podría servir en un caso particular.

- Constructor: Dado una lista que contiene el número de jugadores, un mazo, un modo de juego específico y una seed que devuelva un número aleatorio al llamarla. Se devolverá una lista específica como todas las características mencionadas.
- Función de Pertenencia, Selectores y Otras Funciones: *ver tabla 2 en anexos, página 13.*

2.2.2 OPERACIONES OBLIGATORIAS

cardsSetIsDobble: Se implementa un predicado que verifica si el mazo efectivamente es un mazo válido.

cardsSetNthCard: Se implementa un predicado con recursividad de cola, que obtiene la n-ésima (nth) carta desde el conjunto de cartas partiendo desde 0 hasta (totalCartas-1).

cardsSetFindTotalCards: Se implementa un predicado que a partir de una carta de muestra, determina la cantidad total de cartas que se deben producir para construir un conjunto válido. Para esto es necesario completar la función nthCard, ya que se debe indicar la carta número n a evaluar.

cardsSetToMissingCards: Se implementa un predicado que a partir de un conjunto de cartas retorna el conjunto de cartas que hacen falta para que el set.

cardsSetToString: Se implementa un predicado con recursividad natural que convierte un conjunto de cartas a una representación basada en strings que pueda visualizarse a través de la función display. Para esto se realiza un cambio de tipos de datos y transforma todo a un string. Posteriormente con el predicado write se puede observar los símbolos o elementos de forma ordenada.

dobbleGameRegister: Se implementa un predicado para registrar a un jugador en un juego. Los jugadores tienen un nombre único y no puede exceder la cantidad de jugadores registrados. Para esto se agrega el nombre de registro a una lista vacía y se suma un jugador a la lista de jugadores.

dobbleGameWhoseTurnIsIt: Predicado que retorna el usuario a quién le corresponde jugar en el turno. Para esto, se busca el último jugador que ingreso a la lista de jugadores, en caso de que coincidan, se retornará una lista con el nombre del jugador a quien le corresponda el turno y además el modo de juego en el cual se encuentra el juego.

dobbleGamePlay: Se implementa un predicado que genera un acción en una partida y actúa sobre el tipo de dato abstracto game.

dobbleGameStatus: Se implementa un predicado que devuelve el estado actual del juego, se devolverá una lista con todos los elementos actuales del juego mediante funciones selectoras.

dobbleGameScore: Se implementa un predicado que retorna el puntaje de un jugador a partir de su nombre de usuario. Para esto se implementa la función map con la función anónima y además con la lista de jugadores que se obtiene mediante la función selectora y el argumento de game.

dobbleGameToString: Se implementa un predicado que convierte un juego/partida a una representación basada en strings que posteriormente pueda visualizarse a través de la predicado write. Para esto se convierte cada elemento dentro de game en un string y posteriormente se unen todos en un solo string.

2.3 ASPECTOS DE IMPLEMENTACIÓN

2.3.1 COMPILADOR

Para este proyecto es necesario el IDE Swi-Prolog, específicamente de versión 8.x.x o superior. Como alternativas, también es posible utilizar Visual Studio Code con la extensión de Swi-Prolog o bien, se puede hacer uso de SWISH (Prolog online).

2.3.2 ESTRUCTURA DEL CODIGO

Para esta ocasión, no es necesario establecer una estructura, ya que en el lenguaje de programación lógico, la base mínima y máxima es una única base de conocimientos, por lo tanto mantener una única base de conocimiento es el mínimo y máximo documento donde se podrán realizar consultas, por lo tanto, solo bastará con un archivo, el cual al interior se encontrarán todos los TDA y sus respectivos predicados, tanto obligatorios como no obligatorios.

2.4 INSTRUCCIONES DE USO

2.4.1 EJEMPLOS DE USO

Lo primero que se debe hacer es ejecutar una instrucción que permitirá visualizar con claridad todas las listas, ya que a veces Prolog acorta las listas para que no se extiendan tanto. Para evitar eso, se coloca el siguiente comando:
set_prolog_flag(answer_write_options,[max_depth(0)]).

Una vez ejecutado, ahora se puede cargar la base de conocimientos.
Simplemente se debe ir a la opción File->Consult y luego seleccionar el archivo “dobbleGame_202020917_EscribanoGómez.pl”. Si todo sale bien, prolog dará

como respuesta “true.”, lo cual significa que la base de conocimiento fue cargada con éxito.

Una vez cargada la base de conocimiento, los ejemplos se encuentran al final, ahora solo basta con copiar uno de los ejemplos y ejecutar la consulta.
Sin embargo, se añaden ejemplos de uso en anexos.

2.4.2 RESULTADOS ESPERADOS

Se espera crear una simulación de Dobble correctamente, que funcione sin errores ni ambigüedades y que conserve los fundamentos del paradigma funcional.

Junto con lo anterior, se espera trabajar correctamente con listas y recursión, ya que son fundamentales para el desarrollo del proyecto. Por último, se espera que cada función haga su procedimiento correctamente y que los TDAs tengan representaciones correctas.

2.4.3 POSIBLES ERRORES

Posibles errores pueden suceder si se utiliza mal el orden de los predicados de las operaciones o bien, si no se utilizan correctamente las variables a la hora de ejecutar el código. Además, si bien todos los predicados de cada TDA están bien documentadas y existen ejemplos, no todos los predicados entregan resultados esperados como se encuentra en el apartado de ejemplos del enunciado. Sin embargo existe una definición correcta de dominios, recorridos, descripciones y ejemplos, los cuales podrían llevar por buen camino a cada predicado.

2.5 RESULTADOS Y AUTOEVALUACIÓN

2.5.1 RESULTADOS OBTENIDOS

Los resultados obtenidos fueron los esperados, ya que se logró implementar TODOS los TDA con sus respectivos predicados obligatorios. El programa funciona en cas su totalidad, excluyendo algunos predicados que no funcionan en su totalidad, sin embargo, el objetivo queda más que cumplido. Se logra una exitosa solución a un problema en un paradigma lógico.

2.5.2 AUTOEVALUACIÓN

La Autoevaluación se realiza de la siguiente forma: 0: No realizado – 0.25: Funciona 25% de las veces – 0.5: Funciona 50% de las veces 0.75: Funciona 75% de las veces – 1: Funciona 100% de las veces.

Debido a que se probaron los predicados con varios ejemplos y no se encontraron errores, se considera que todos los predicados funcionan el 100% de las veces.

Para ver la tabla de Autoevaluación, ver la Tabla N°3 en anexos, página 13.

3. CONCLUSIÓN

Tras realizar y finalizar el proyecto, se puede concluir que se cumplió el principal objetivo de este proyecto e informe era introducir y comprender el paradigma lógico el cual se ha cumplido correctamente.

El paradigma lógico es complejo para quien no haya salido del paradigma imperativo u orientado a objetos, sobre todo en un principio, entender la idea de que absolutamente todo es un predicado y unificación, es complicado. Sin embargo, durante de este proyecto se ha presentado un entendimiento exponencial del paradigma. Si bien no se logró completar los requerimientos en su totalidad, se completó la gran mayoría y las funciones que no se completaron quedaron con avance lo cual significa que la idea está y que podría implementarse.

4. BIBLIOGRAFÍA Y REFERENCIAS

1. Flores, V. (2022). “Proyecto Semestral de Laboratorio”. Paradigmas de Programación. Enunciado de Proyecto Online. Recuperado de: https://docs.google.com/document/d/1pORdJwp5WJ7V3DIAQik9B58llpdxpurQRVTKPZHOpS8/edit
2. Chacon, S. y Straub, B. (2020). “Pro Git – Todo lo que necesitas saber sobre Git’’. Libro Online. Recuperado de : https://drive.google.com/file/d/1mHJsfvGCYcIhdrmK- lBI6a1WS-U1AAPi/view
3. Autores Desconocidos. (2021). “Swi-Prolog Documentation’’. Swi-Prolog. Documentación Online. Recuperado de: https://www.swi-prolog.org/pldoc/index.html
4. Multiples Autores. (Septiembre 2013). “El lenguaje de programación PROLOG”. Libro online. Recuperado de: https://drive.google.com/file/d/1DgxI64oAB5do7A-ajCXCTphnGWHTCEmk/view?usp=sharing

5. ANEXOS

Tabla N°1: Predicados TDA cardsSet

Tipo de predicado	Nombre	Descripción
Predicado de Pertenencia	**isDobbleGame**	Predicado que verifica si el cardsSet es un cardsSet válido.
Otros Predicados	**recorrerCarta**	Predicado que recorre una carta y verifica si todos los elementos se repiten solo 1 vez.
Otros Predicados	**isPrime**	Predicado que verifica si un número es primo.
Otros Predicados	**maxCards**	Predicado que agrega a una lista una cantidad de elementos específicos desde una lista.
Otros Predicados	**largoLista**	Predicado que obtiene el largo de una lista.
Otros Predicados	**agregarCarta**	Predicado que agrega una carta a una lista de listas.
Otros Predicados	**primeraCarta**	Predicado que genera la primera carta del mazo.
Otros Predicados	**primerCiclo**	Predicado que simula de un ciclo iterativo.
Otros Predicados	**segundoCiclo**	Predicado que simula de un ciclo iterativo.
Otros Predicados	**tercerCiclo**	Predicado que simula de un ciclo iterativo.
Otros Predicados	**ordenN**	Predicado que genera cartas de orden n.
Otros Predicados	**ordenNN**	Predicado que genera cartas de orden $n^{2.}$
Selector	**nthElement**	Predicado que obtiene el enésimo elemento de una lista.
Otros Predicados	**cardsSetNthCard**	Predicado que obtiene la enésima carta de una lista de cartas.
Otros Predicados	**encontrarCarta**	Predicado que encuentra la carta que coincide (o verifica si existe en el mazo).
Otros Predicados	**cardsSetFindTotalCards**	Predicado que calcula la cantidad de cartas máximas que se pueden generar en base a una carta.
Otros Predicados	**calcularOrden**	Predicado que calcula el orden del juego.
Modificador	**cardsSetMissingCards**	Predicado que devuelve un cardsSet válido a partir de un cardsSet de muestra.
Otros Predicados	**cadenaCarta**	Predicado que transforma una carta a una cadena.
Otros Predicados	**cadenaCartas**	Predicado que transforma una lista de cartas a cadena.
Otros Predicados	**cardsSetToString**	Predicado que transforma todo el conjunto de cartas a una cadena.

Tabla N°2: Predicados TDA game

Tipo de predicado	Nombre	Descripción
Selector	**getAreaDeJuego**	Predicado que obtiene el área de juego.
Selector	**getPiezasDisponibles**	Predicado que obtiene las piezas disponibles.
Selector	**getTurns**	Predicado que obtiene el número del turno.
Selector	**getPlayersScore**	Predicado que obtiene la lista de puntajes de los jugadores.
Selector	**getPlayers**	Predicado que obtiene la lista de jugadores.
Selector	**getNumPlayers**	Predicado que obtiene el número total de jugadores.
Selector	**getStatus**	Predicado que obtiene el estado del juego.
Selector	**getMode**	Predicado que obtiene el modo de juego.
Selector	**dobbleGameRegister**	Predicado que genera un nuevo juego.
Selector	**dobbleGameWhoseTurnIsIt**	Predicado que devuelve al jugador que le toca jugar.
Modificador	**dobbleGamePlay**	Predicado que inicia un juego mediante la especificación de una acción.
Otros Predicados	**dobbleGameStatus**	Predicado que devuelve el estado actual del juego.
Otros Predicados	**dobbleGameScore**	Predicado que muestra el puntaje de un jugador a partir de un nombre.
Otros Predicados	**dobbleGameToString**	Predicado que convierte los elementos del juego a una cadena.

Tabla N°3: Autoevaluación de los requerimientos funcionales

Requerimientos Funcionales	Evaluación
TDAs	1
cardsSetIsDobble	1
cardsSetNthCard	1
cardsSetFindTotalCards	1
cardsSetMissingCards	0.5
cardsSetToString	1
dobbleGameRegister	1
dobbleGameWhoseTurnIsIt	0.75
dobbleGamePlay	0.5
dobbleGameStatus	1
dobbleGameScore	0.75
dobbleGameToString	0.5