UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

PARADIGMAS DE PROGRAMACIÓN

Informe N°2: “CAPITALIA”

(Implementado en Prolog)

Alumno:

Angel Benavides Araya

Profesor:

Gonzalo Martinez Ramirez

06 de Junio de 2025

Índice de Contenidos

Índice de tablas

1. Introducción

En el presente informe se detalla el desarrollo de la implementación de *Capitalia*, un juego basado en *Monopoly*, utilizando el lenguaje de programación Prolog. En particular, se hará el uso del paradigma lógico para llevar a cabo esta implementación.

En primer lugar se describe brevemente el problema dado y sus características, además de la caracterización del paradigma a utilizar, esto con el objetivo de dar a conocer el panorama general tanto de problema particular como del paradigma y las limitaciones que este conlleva.

Posteriormente se efectuará un análisis detallado del problema en relación a los requerimientos establecidos, permitiendo identificar los principales problemas a resolver. Con dicho análisis, se propondrá un diseño de solución que incluirá la descomposición de los elementos de dicha solución, así como la presentación de los recursos conceptuales y técnicos empleados para llegar a esta idea.

Además, se explicarán las consideraciones tomadas durante la implementación, justificando las decisiones de diseño tomadas. Se incluirá una sección con instrucciones de uso y ejemplos claros para facilitar la comprensión de la ejecución del programa.

Finalmente, se presentarán los resultados obtenidos, una evaluación crítica del funcionamiento del sistema y las conclusiones generales del proyecto, reflexionando sobre las dificultades enfrentadas, los aprendizajes logrados y posibles mejoras futuras.

1.1 Descripción del problema

El problema a abordar consiste en la implementación digital del juego *Capitalia* (clon de *Monopoly*) utilizando la programación lógica, de manera que se logre como mínimo una simulación básica del juego. Esto implica: Representación de jugadores, tablero, propiedades, interacciones (compra/venta), cartas, eventos, casillas especiales y transacciones.

Además, se debe considerar una jugabilidad dinámica, al diferencia del *Monopoly* tradicional, al inicio de la partida se definen diversos parámetros que varían por cada juego. como por ejemplo: cantidad de dados, cantidad de propiedades, número máximo de casas/hoteles, impuesto inicial, entre otros.

Hay ciertas limitaciones con la implementación total de un clon tipo *Monopoly*, en particular, la exigencia de la utilización del paradigma lógico para solucionar el problema. Este requerimiento impone desafíos adicionales que deben ser considerados en el planteamiento de la solución.

1.2 Descripción del paradigma

La programación lógica comenzó a tomar forma en los años 60 como un intento de unir la lógica matemática formal con el desarrollo de software. Esta búsqueda se centró en convertir el razonamiento lógico en una herramienta computacional viable. Fue un esfuerzo interdisciplinario, en particular entre investigadores de la inteligencia artificial y la lógica computacional, que veían en la deducción lógica un método natural para resolver problemas complejos.

En 1972, Alain Colmerauer y Robert Kowalski desarrollaron Prolog (*PROgramming in LOGic*) en la Universidad de Marsella. Este lenguaje no solo implementa, sino que definió el paradigma lógico tal como lo entendemos hoy, al basarse en cláusulas de Horn y un modelo de ejecución centrado en la resolución lógica, la unificación de términos y el backtracking automático.

La programación lógica es un paradigma declarativo donde los programas se definen como un conjunto de hechos y reglas lógicas, en lugar de una secuencia explícita de instrucciones. El control de flujo no lo determina el programador, sino el sistema, que utiliza un proceso de resolución lógica para encontrar soluciones que satisfacen las condiciones establecidas. Como señala el material de Imperial College London, en este paradigma "*el énfasis está en describir el conocimiento sobre el problema más que el procedimiento para resolverlo*" [1].

Este enfoque se apoya en la lógica de predicados de primer orden, y los programas se expresan mayoritariamente como cláusulas de Horn, ideales para realizar deducciones automáticas. En palabras del *Structure and Interpretation of Computer Programs* (SICP), "*la programación lógica proporciona un lenguaje en el cual se describe el mundo en términos de relaciones, y el intérprete usa esas relaciones para responder preguntas sobre el mundo*" [2].

En este proyecto, el paradigma se aplica modelando los elementos del juego como hechos (por ejemplo, propiedades, jugadores, cartas), y comportamientos como reglas (por ejemplo, el cálculo de renta, condiciones de victoria o movimiento). La inferencia de nueva información, como las acciones posibles de un jugador o los efectos de una carta, se realiza automáticamente mediante consultas al sistema lógico, sin necesidad de describir paso a paso cómo llegar a esas conclusiones.

2. Desarrollo

2.1 Análisis del problema

En concreto, se solicitan 21 RF obligatorios (véase tabla 1), equivalentes a 21 funciones que cubren lo más básico para poder hacer funcionar Capitalia. Se solicitan tanto constructores, modificadores y otros. en los TDA: Juego, tablero, propiedad, jugador y carta. Implícitamente será necesario implementar selectores para poder obtener datos al momento de necesitar en la implementación de estas 21 funciones.

Los constructores son bastante básicos y estándar, pero se debe recordar que se puede modificar/agregar cosas al recorrido de estos, facilitando el acceso a posibles datos necesarios más adelante para una implementación más sencilla de cada RF.

2.2 Diseño de la solución

Dados los requisitos funcionales que se deben cubrir, se identificaron los TDA necesarios para el desarrollo de Capitalia y la estructura de estos sigue una dependencia jerarquizada, que se centraliza en el TDA Juego, dentro de este se encuentran: Jugadores, Tablero, Propiedades, Carta. Además, se planeaba la implementación de TDA Mazo y TDA Propiedades_Especiales, para representar los mazos suerte y comunidad, junto a las propiedades especiales. Por falta de tiempo se desechó esta idea. Aunque no se incluyen diagramas visuales, la jerarquía de dependencias entre TDA se respetó estrictamente, siendo el TDA Juego el contenedor principal de la lógica del sistema.

Cada requerimiento funcional (RF) fue abordado de manera individual, empleando las técnicas solicitadas en el enunciado y sin efectos colaterales, en línea con el paradigma lógico. Los constructores RF2, RF3, RF4, RF6 se representaron como una lista con cada elemento recibido en el dominio, respetando el dominio y sin utilizar salidas extras en el recorrido, contrario a lo realizado en el laboratorio anterior. Por otro lado, el RF5 (constructor tablero) no se construyó inmediatamente la estructura, sino que se dejó como una lista de listas, y al momento de llamar a la función para agregar propiedades extras (RF7), se manejó el tema de las posiciones. Además de esto, se asumió que cada posición sería única y que no habrían conflictos entre si. Por lo que no se manejaron casos bordes de este estilo. La misma suposición de realizo para los id's de Cartas, casillas especiales y propiedades.

RF7 (tableroAgregarPropiedades/3) se realizó de forma meramente declarativa haciendo uso de append, no se vio la necesidad de "ordenar" las propiedades por índice como tal. Esto debido al supuesto ya mencionado de que las posiciones no causarían conflictos y cada instancia vendría con la forma: [Pos, [Lista representando la propiedad/carta/casilla especial]].

En RF08 (juegoAgregarJugador/3) se hizo uso del condicional nativo de Prolog (-> ;) además de ser declarativa. Esto fue necesario para manejar casos en los cuales el jugador no haya sido inicializado con el dinero inicial (1500), esto por las indicaciones del profesor en el documento.

RF9 (juegoObtenerJugadorActual/2) se realizó de manera puramente declarativa, utilizando funciones nativas de Prolog, en concreto, la función: nth0 para obtener índices fue de gran utilidad en la implementación.

Para RF10 (juegoLanzarDados/4) se hizo uso de los predicados proporcionados por el profesor en el enunciado para generar la pseudoaleatoriedad y gestionar el control de las semillas. Con esto, fue necesario modular esta función, realizando implementaciones de predicados como: lanzarTodosLosDados/3 y uso del length nativo. La estrategia fue mayoritariamente declarativa.

En RF11 (juegoMoverJugador/4) se evidenció la utilidad de la implementacion de getters y setters para modificar los elementos, especialmente en un lenguaje de programación inmutable, además de esto se utilizó la operación módulo para realizar una correcta actualización de la posición del jugador (evitando salirse del tablero), esto es parecido a un clásico problema simple de Llave Cifrada César, donde se utiliza el módulo para calcular correctamente las vueltas que da el reloj/cifrador.

Para RF12 (jugadorComprarPropiedad) se utilizó una forma declarativa simple con condicionales, RF13 (juegoCalcularRentaPropiedad/3) y RF14 (juegoCalcularRentaJugador/3)) fue necesario utilizar la operación de corte junto a predicados condicional, no se utilizó el if (->), sino que se "segmento" el predicado en casos, dejando 4 predicados con el mismo nombre en el código. Se hizo una adaptación propuesta por el profesor en el enunciado para manejar solo el caso de 1 hotel y n casas (valor determinado al inicio del juego, en su constructor), a destacar es la utilización de la función nativa "floor" para truncar la parte entera de los cálculos y evitar trabajar con floats. RF15 (juegoConstruirCasa) y RF16 (juegoConstruirHotel) son predicados muy similares que fueron resueltos de manera declarativa con condicional para adherirse a las instrucciones. RF17 (jugadorPagarRenta) es declarativa y sencilla. RF18 también es declarativa y solo realiza una verificación y usa un corte. RF19 (juegoExtraerCarta) no fue implementado por requerir la implementación de TDA Mazo mencionada anteriormente, la cual por temas de tiempo no pudo realizarse. RF20 (jugadorEstaEnBancarrota) es simplemente una verificación declarativa, RF21 (juegoJugarTurno) tampoco pudo implementarse a tiempo por la amplia cantidad de reglas del juego y mala planificación de tiempos.

2.3 Aspectos de implementación

Las implementaciones de los Requisitos Funcionales (RF) se realizaron en el archivo "Main_214525321_ANGEL_BENAVIDES_ARAYA.pl", mientras que cada TDA cuenta con un archivo independiente, organizado en funciones constructoras, selectoras, modificadoras y otras.

Todo el código fue desarrollado en SWI-Prolog version 9.2.9 for x86_64-linux, sin utilizar librerías externas, y ejecutado mediante la consola del ordenador (utilizando el comando 'swipl'). El desarrollo se llevó a cabo en un notebook Lenovo ThinkPad X380 Yoga y un PC de escritorio, ambos con el sistema operativo Linux Mint 22.1 x86_64, entorno de escritorio Cinnamon 6.4.8.

2.4 Instrucciones de uso

Para ejecutar el código, todos los archivos deben estar ubicados en el mismo directorio, ya que existen múltiples dependencias entre ellos. Estas incluyen tanto dependencias parciales unidireccionales (se evitaron dependencias circulares), debido a la naturaleza del programa. En concreto, se utilizaron los predicados 'module' y 'use_module' para la comunicación entre archivos.

El archivo main.pl requiere de los TDA. Asimismo, el script de ejecución requiere tanto de main.rkt como de ciertos archivos de los TDA.

El script de ejecución presenta algunas modificaciones respecto a lo solicitado en el documento original, debido a decisiones creativas y problemas con el script base. Por esta razón, el archivo "script_base_214525321_ANGEL_BENAVIDES_ARAYA.pl" se encuentra "fragmentado" en dos.

A continuación se mencionan las instrucciones de ejecución y pruebas de los RF (esto también estará en el README del repositorio). En particular, este instructivo está pensado para ser usado en Linux Mint o cualquier otra distribución con SWI Prolog version 9.2.9 , creo que en Windows se debe usar el programa de SWI Prolog, cargando el archivo y realizando las consultas mencionadas, pero no estoy del todo seguro.

En concreto en Linux Mint se debe navegar al directorio contenedor de los archivos (mediante el uso de cd en consola), una vez ahí, se ejecuta el comando **"swipl"** para abrir la consola de consultas propia de SWI-Prolog, en la consola se realiza la consulta:

?- [script_base_214525321_ANGEL_BENAVIDES_ARAYA]. ←- saltarán múltiples warnings que deben ser ignorados, además de un "true.", esto indica que se pudo leer el archivo correctamente. Una vez ahí, siguen las consultas para verificar el funcionamiento de los RF:

?- rf2to13. ← Este comando ejecutara los ejemplos de RF 2 hasta RF13, cada uno con una breve explicación

Por esto se menciona que el script viene "fragmentado", se planeaba ejecutar todo en la función, pero por inconsistencias y problemas en la actualización de estados para ejemplificar todas las funciones se tuvo que segmentar en subconsultas para ejemplificar el resto de predicados.

Continuando:

?- rf14a. ← Esta consulta ejecuta el primer ejemplo de uso de RF14

?- rf14b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF14

?- rf15a. ← Esta consulta ejecuta el primer ejemplo de uso de RF15

?- rf15b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF15

?- rf16a. ← Esta consulta ejecuta el primer ejemplo de uso de RF16

?- rf16b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF16

?- rf17a. ← Esta consulta ejecuta el primer ejemplo de uso de RF17

?- rf17b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF17

?- rf18a. ← Esta consulta ejecuta el primer ejemplo de uso de RF18

?- rf18b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF18

?- rf20a. ← Esta consulta ejecuta el segundo ejemplo de uso de RF20

?- rf20b. ← Esta consulta ejecuta el segundo ejemplo de uso de RF20

Cabe mencionar que dicho script importa todas las funciones del archivo main.pl y las requeridas de los TDA 's para ejecutar y probar las funcionalidades principales.

2.5 Resultados y autoevaluación

La gran mayoría del laboratorio se pudo desarrollar de manera satisfactoria, con excepción del RF19 y RF21 , los cuales no se lograron implementar. Esto impide que el desarrollo actual represente de forma completa una copia funcional de Monopoly, dado que el RF21 corresponde a la función que ejecuta el flujo principal del juego.

No obstante, se lograron implementar correctamente bloques particulares que forman parte de una implementación efectiva. Se reconoce que una mejor gestión del tiempo habría permitido completar el laboratorio en su totalidad.

3. Conclusiones

Los alcances actuales de la implementación se consideran adecuados como primer acercamiento al paradigma lógico. A lo largo del desarrollo se logró implementar la mayoría de los requerimientos funcionales, lo que evidencia una comprensión progresiva (y frustrante) de los beneficios y desventajas propias del paradigma funcional. Sin embargo, la falla en la implementación del último requerimiento funcional impide una ejecución completa del script proporcionado, lo cual limita la funcionalidad global del sistema.

Una de las principales dificultades fue la necesidad constante de prueba y error, producto de una comprensión incompleta del paradigma lógico y un conocimiento limitado de funciones propias de Prolog que habrían facilitado el desarrollo. A esto se suma la interpretación incorrecta de algunos requisitos iniciales, lo que llevó a retrabajos y replanteamientos que ralentizaron el avance y generaron frustración en algunas etapas. Si se repitiera el desarrollo del laboratorio, se priorizaría

una planificación más detallada de la estructura del programa, así como una lectura más crítica de los requisitos antes de comenzar la implementación. Esto permitiría evitar una parte importante del ciclo de depuración y facilitaría un desarrollo más fluido y eficiente.

El paradigma lógico presentó una barrera de entrada conceptual mucho más alta que el funcional. En Prolog, el flujo de ejecución no es definido por el programador sino por el motor de inferencia, lo que obliga a replantear completamente cómo se estructura una solución. Al principio, esto llevó a una sensación de falta de control sobre la ejecución del programa. La dependencia del backtracking, las reglas implícitas de evaluación y la necesidad de utilizar cortes (!) o condicionales especiales (-> ;) complicaron la depuración y dificultaron la implementación de ciertas reglas que en el paradigma funcional eran mucho más directas.

4. BIBLIOGRAFÍA

1. Imperial College London. (s.f.). *Programming paradigms*. Recuperado de https://www.doc.ic.ac.uk/~klc/PP

2. Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs* (2.ª ed.). MIT Press. Recuperado de https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html

3. Kowalski, R. (1979). *Algorithm = Logic + Control*. *Communications of the ACM, 22*(7), 424–436. Recuperado de https://www.doc.ic.ac.uk/~rak/papers/AlgorithmControl.pdf

5. Anexos

Tabla 1. Descripción de requisitos funcionales

RF	Descripción
1	**TDAs** - Realizar TDAs para cada estructura relevante en un archivo separado cada uno, siguiendo instrucciones.
2	**TDA Jugador - constructor.** Predicado que permite crear un jugador.
3	**TDA Propiedad - constructor.** Crear una propiedad en el juego.
4	**TDA Carta - constructor.** Crear una carta en el juego.
5	**TDA Tablero - constructor.** Crear un tablero de CAPITALIA.
6	**TDA Juego - constructor.** Predicado que crea una partida de CAPITALIA.
7	**TDA Tablero - modificador - Agregar propiedad.** Predicado para agregar propiedades al tablero. Permite agregar n propiedades.
8	**TDA Juego - modificador - Agregar jugador.** Predicado para agregar un jugador a la partida.
9	**TDA Juego - selector - obtener jugador actual.** Predicado para obtener el jugador cuyo turno se encuentra en curso (jugador actual).
10	**TDA juego - otros - lanzar dados.** Predicado para simular el lanzamiento de N dados
11	**TDA Jugador - modificador - Mover Jugador.** Predicado para mover al jugador en el tablero.
12	**TDA Jugador - modificador - Comprar propiedad.** Predicado que permite crear comprar una propiedad.

13	**TDA Jugador - otros - Calcular Renta.** Predicado para calcular la renta de una propiedad sumando todas las casas y hoteles que tenga.
14	**TDA Propiedad - otros - Calcular Renta.** Predicado para calcular la renta de las propiedades de un jugador.
15	**TDA Propiedad- modificador -Construir Casa** Predicado que permite construir una casa en una propiedad
16	**TDA Juego - modificador -Construir Hotel** Predicado que permite construir hoteles en propiedades
17	**TDA Jugador - otros - Pagar Renta.** Predicado para que un jugador pague renta a otro.
18	**TDA Propiedad - modificador - Hipotecar Propiedad.** Predicado para hipotecar una propiedad.
19	**TDA Juego - modificador -Extraer carta.** Predicado para extraer una carta del mazo correspondiente
20	**TDA Jugador - otros - Verificar bancarrota.** Predicado para verificar si un jugador se encuentra en bancarrota (sin dinero).
21	**TDA Juego - modificador - Realizar turno.** Función que ejecuta un turno completo aplicando todas las reglas del juego.