UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

PARADIGMAS DE PROGRAMACIÓN

Informe N°1: “CAPITALIA”

(Implementado en Scheme)

Alumno:

Angel Benavides Araya

Profesor:

Gonzalo Martinez Ramirez

05 de Mayo 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 Scheme. En particular, se hará el uso del paradigma funcional para llevar a cabo esta implementación.

En primer lugar se describirán 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 programación funcional, 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 funcional para solucionar el problema. Este requerimiento impone desafíos adicionales que deben ser considerados en el planteamiento de la solución.

1.2 Descripción del paradigma

El paradigma funcional nace del cálculo lambda, notación matemática formulada por Alonzo Church en la década de 1930. El cálculo lambda fue redescubierto como una herramienta versátil en las ciencias de la computación en los 60 's (Jung, 2004). En capacidad computacional, el cálculo lambda es equivalente a la máquina de Turing, cimiento del paradigma imperativo. Sin embargo, el funcional enfatiza las funciones puras, que siempre devuelven el mismo resultado con idénticos argumentos y no provocan efectos secundarios, y adopta un estilo declarativo, privilegiando el "qué" sobre el "cómo" y empleando notación prefija, donde el operador precede a sus operandos.

La inmutabilidad de las variables impide el uso de ciclos for o while, de modo que la iteración y el procesamiento de datos se resuelven mediante recursión sobre listas u otras estructuras recursivas. (Bhadwal, s. f.) La transparencia referencial garantiza que cualquier expresión puede sustituirse por su valor sin modificar el comportamiento, lo que facilita optimizaciones como la evaluación perezosa. Además, en este paradigma las funciones se tratan como valores de primera clase: pueden pasarse como argumentos, devolverse o almacenarse, posibilitando funciones de orden superior y técnicas como la currificación, que transforma una función de varios parámetros en una serie de funciones unarias. El resultado es un código más predecible, modular y fácilmente paralelizable, aunque exige una mentalidad distinta a la del estado mutable y puede implicar una curva de aprendizaje y, en ocasiones, sobrecostes de rendimiento.

En definitiva, la programación funcional ofrece un modelo declarativo en el que la pureza de las funciones, la inmutabilidad de los datos y el uso sistemático de la recursión promueven código más predecible y fácil de razonar, mientras que las funciones de orden superior y la currificación aportan gran flexibilidad y expresividad. A cambio, exige adoptar una mentalidad distinta a la de los bucles y el estado mutable, lo cual puede suponer una curva de aprendizaje, pero a la larga se traduce en software más modular, conciso y susceptible de optimizaciones automáticas.

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 presentó la necesidad de implementar un TDA Mazo, para representar los mazos suerte y comunidad. 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 funcional. Los constructores RF2, RF3, RF4, RF6 se representaron como una lista con cada elemento recibido en el dominio, sin embargo, para facilitar la implementación de futuras funciones en el constructor RF3 (constructor propiedad), se utilizaron 3 valores extras en el recorrido: hoteles, maxCasas, maxHoteles. Mientras que en 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.

En RF7 (tablero-agregar-propiedad) se utilizó una gran mezcla de funciones declarativas para poder ordenar el tablero, convirtiéndose en un problema de posicionamiento, dejando con sus respectivas posiciones incluso a casillas que no venían con esta incluida para su fácil acceso a futuro. Se usó map, append, filter, member para poder lograr esto de forma declarativa.

En RF9 se ocupa list-ref y es meramente declarativo para agregar un jugador a la partida y en RF8 se utiliza una función definida en el archivo de funciones auxiliares: “reemplazar-n” definida en el archivo “funcionesAux”, que utiliza recursión natural para reemplazar un valor de una lista.

Para RF10 se utilizó lo dado en el enunciado para hacer la implementación de pseudoaleatoriedad, el requisito como tal era simplemente dejarlo como una lista en base a las seeds introducidas.

Para RF11 (jugador-mover) 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 se utilizó una forma declarativa simple, RF13 (calcular renta de todas las propiedades del jugador) se tuvo que utilizar una mezcla de apply filter y map, para lograr acceder a las posiciones dentro del TDA game. En RF14 (calcular renta de propiedad) se hizo una adaptación del enunciado para manejar de 1 a 4 hoteles, a destacar es la utilización de la función nativa “exact-truncate” para truncar la parte entera de los cálculos y evitar trabajar con floats. A partir de acá los siguiente RF no tienen nada especial que los mencionados anteriormente no posean a diferencia del RF21 (juego-jugar-turno) el cual no se logró implementar debido a una mala administración de tiempo y una gran cantidad de funciones extras a los RF que no se tuvieron en cuenta en la planificación del desarrollo del proyecto.

2.3 Aspectos de implementación

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

Todo el código fue desarrollado en DrRacket versión 8.10, sin utilizar librerías externas, y ejecutado mediante el intérprete de Racket vía CLI. 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 como circulares, debido a la naturaleza del programa. En concreto, se utilizaron las funciones 'lazy-require' y 'require' para gestionar las dependencias.

El archivo main.rkt requiere de los TDA, al mismo tiempo que los TDA requieren de main.rkt. 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 que no se logró implementar correctamente el RF21 (`juego-jugar-turno`). Por esta razón, se incluyeron ejemplos de uso de los RF implementados, con valores diseñados para demostrar su funcionamiento. Además, se dejó comentada la idea de implementación del RF21 para un posible desarrollo futuro, incluyendo el uso del comodín.

Cabe mencionar que dicho script importa todas las funciones del archivo main.rkt (RF) y algunas funciones de los TDA para ejecutar y probar las funcionalidades principales. Los archivos se encuentran listos para ser ejecutados directamente.

2.5 Resultados y autoevaluación

La gran mayoría del laboratorio se pudo desarrollar de manera satisfactoria, con excepción del RF21, el cual no se logró 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. Paralelamente a esta entrega, se continuará trabajando en el código con el objetivo de alcanzar una implementación completa acorde a lo solicitado, para eventualmente y de ser posible hacer uso de la herramienta del comodín para una entrega final más completa.

3. Conclusiones

Los alcances actuales de la implementación se consideran adecuados como primer acercamiento al paradigma funcional. A lo largo del desarrollo se logró implementar la mayoría de los requerimientos funcionales, lo que evidencia una comprensión progresiva del enfoque declarativo y la manipulación de estructuras inmutables. 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 funcional y un conocimiento limitado de funciones propias de Racket/Scheme 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.

4. Bibliografía

1. Jung, A. (2004). A short introduction to the Lambda Calculus. *Unknown*. http://czyborra.com/thti/lambda-achimjung.pdf
2. Bhadwal, A. (s. f.). Functional Programming Languages: Concepts & Advantages. *Hackr.io*. https://hackr.io/blog/functional-programming

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.** Función 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.** Función que crea una partida de CAPITALIA.
7	**TDA Tablero - modificador - Agregar propiedad.** Función para agregar propiedades al tablero. Permite agregar n propiedades.
8	**TDA Juego - modificador - Agregar jugador.** Función para agregar un jugador a la partida.
9	**TDA Juego - selector - obtener jugador actual.** Función para obtener el jugador cuyo turno se encuentra en curso (jugador actual).

10	**TDA juego - otros - lanzar dados.** Función para simular el lanzamiento de 2 dados
11	**TDA Jugador - modificador - Mover Jugador.** Función para mover al jugador en el tablero.
12	**TDA Jugador - modificador - Comprar propiedad.** Función que permite crear comprar una propiedad.
13	**TDA Jugador - otros - Calcular Renta.** Función para calcular la renta de una propiedad.
14	**TDA Propiedad - otros - Calcular Renta.** Función para calcular la renta de una propiedad.
15	**TDA Propiedad- modificador -Construir Casa** Función que permite construir casas en propiedades
16	**TDA Juego - modificador -Construir Hotel** Función que permite construir hoteles en propiedades
17	**TDA Jugador - otros - Pagar Renta.** Función para que un jugador pague renta a otro.
18	**TDA Propiedad - modificador - Hipotecar Propiedad.** Función para hipotecar una propiedad.
19	**TDA Juego - modificador -Extraer carta.** Función para extraer una carta del mazo correspondiente
20	**TDA Jugador - otros - Verificar bancarrota.** Función 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.