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

Laboratorio Nº1 : Implementación de “Capitalia” bajo el paradigma funcional.

Nombre: Felipe Aedo Jaramillo
Profesor: Edmundo Leiva
Asignatura: Paradigmas de Programación
5 de mayo de 2025

Índice

Introducción

El presente informe tiene como objetivo presentar y analizar la implementación del juego de mesa multijugador **Capitalia**, una versión inspirada en el clásico *Monopoly*. La estructura del informe se conforma de la siguiente manera: en primer lugar, una breve descripción del problema, seguida por la descripción del paradigma utilizado. A continuación, se procede con el análisis del problema para dar paso al diseño de la solución con sus aspectos varios. Finalmente, se adjunta el manual de uso, resultados obtenidos, para finalmente dar con las respectivas conclusiones.

Descripción del problema

El juego **Capitalia** consiste en una simulación económica por turnos, en la cual los jugadores compiten por adquirir propiedades, construir en ellas y cobrar rentas a otros jugadores que caigan en sus casillas. El objetivo principal es llevar a la bancarrota a los oponentes mediante una gestión estratégica de recursos y decisiones acertadas.

El objetivo principal del proyecto es encontrar una forma eficiente y funcional de representar el estado del juego (tablero, jugadores, propiedades, dinero, etc.) y modelar las reglas de **Capitalia**. Las principales características a implementar incluyen:

- Implementación de **TDAs** para modelar el problema.
- Lógica para ejecución de turnos.
- Correcta alternancia de turnos entre múltiples jugadores.

- Implementación de acciones como compra, venta, construcción y cobro de rentas.

- Manejo de eventos especiales (ej. “Ir a la cárcel”, “Suerte”, “Arca Comunal”).

- Detección de bancarrota y condiciones de finalización del juego.

Descripción del paradigma

El paradigma de **programación funcional** es un modelo que concibe el software como la evaluación de funciones puras y evita el uso de estado mutable. En el lenguaje **Scheme**, que da fuerte soporte a este paradigma, las funciones son ciudadanos de primera clase: pueden pasarse como argumentos, devolverse como resultados y asignarse como valores.

Este enfoque se basa en principios clave como:

- **Funciones puras:** devuelven siempre el mismo resultado para los mismos argumentos y no producen efectos secundarios.

- **Inmutabilidad:** los datos no se modifican, sino que se crean nuevas versiones con los cambios.

- **Recursión:** Natural y de cola para generar bucles.

- **Transparencia referencial:** cualquier expresión puede ser sustituida por su valor sin alterar el comportamiento del programa.

- **Funciones de orden superior:** funciones que manipulan otras funciones como datos.

Este paradigma permite que la implementación de **Capitalia** modele el estado del juego como estructuras inmutables, donde cada acción del jugador genera un nuevo estado sin modificar el anterior. La lógica del juego se estructura mediante composición de funciones y recursión, promoviendo así un diseño claro, modular y predecible.

Análisis del problema

El desarrollo de **Capitalia** en Scheme presenta diversos desafíos técnicos y de diseño, particularmente al trabajar dentro de las restricciones del paradigma funcional:

Diseño de las estructuras de datos:
Representar de manera adecuada los datos mediante **TDAs** (Tipos de Datos Abstractos) resulta esencial, pero también desafiante, dado que el juego requiere múltiples estructuras relacionadas (jugadores, propiedades, cartas, tablero, etc.). La lógica del juego debe ser rigurosa y coherente durante la ejecución de los turnos, por lo que un diseño sólido de los TDAs y sus operadores es clave para garantizar la correcta evolución del estado del juego.

Gestión de turnos y flujo del juego:
El control de los turnos debe mantenerse sincronizado con las acciones del usuario, evitando errores como jugadas fuera de turno o saltos indebidos. Asimismo, el sistema debe manejar diversos escenarios de juego, como encarcelamiento, cobro de impuestos, paso por la salida, cartas de azar, etc., asegurando que cada evento altere el estado del juego según las reglas definidas.

Interacción con el usuario:
Al no contar con una interfaz gráfica, se hace fundamental implementar **scripts de simulación** o pruebas que permitan verificar la correcta ejecución del juego. Estos scripts deben cubrir tanto las funcionalidades básicas como situaciones límite.

Limitaciones del entorno:
Una restricción clave del entorno es la **no utilización de let ni set!**, lo que obliga a trabajar exclusivamente con datos inmutables y funciones puras. Esto implica que toda transformación del estado debe modelarse como una función que devuelve una nueva versión del juego, sin modificar el original.

Diseño de la solución

El diseño de la solución para **Capitalia** se basa en una arquitectura funcional dividida en cinco **TDAs principales**, cada uno con sus operadores correspondientes:

- **TDA jugador:** Representa a cada jugador del juego. Contiene datos como nombre, identificador único, saldo de dinero, posición en el tablero, valor booleano para saber si está en la cárcel, cantidad de cartas para salir de la cárcel, y propiedades en posesión. Los operadores permiten actualizar el saldo, mover al jugador, agregar propiedades y verificar si está en bancarrota.

- **TDA propiedad:** Modela cada propiedad del tablero. Almacena su identificador, nombre, valor, renta, dueño (si lo hay), construcciones, estado de hipoteca. Se implementan funciones para comprarla, construir sobre ella, calcular renta y cambiar de dueño.

- **TDA carta:** Representa una carta de azar o de arca comunal. Contiene un mensaje y una acción asociada que modifica el estado del juego. El TDA permite aplicar la carta a todo el juego, retornando un nuevo estado.

- **TDA tablero:** Contiene las cartas disponibles en el juego, la secuencia de casillas especiales del juego (como "Ir a la cárcel", "Salida", "Suerte", etc.), además de las propiedades y su disposición. Incluye funciones para obtener la casilla en determinada posición.

- **TDA juego:** Actúa como controlador principal. Gestiona la lista de jugadores y la transacción de propiedades, además coordina el flujo de turnos, actualiza el estado general del juego, valida acciones y aplica eventos especiales.

Simulación de Turno y Prueba del Sistema

Para validar el correcto funcionamiento de la lógica central del juego, se desarrollaron tres scripts de simulación que permiten ejecutar un juego completo. Dichos scripts, crean explícitamente todos los elementos necesarios para representar una partida: jugadores, propiedades, cartas y el tablero. La función principal utilizada en este contexto es **“juego-jugar-turno”**, parte del TDA juego, que encapsula la lógica de un turno completo. Esta función toma como entrada el estado actual del juego, una lista de dados generados aleatoriamente de manera determinista, y una lista de banderas que simulan las decisiones del jugador en ese turno (el jugador de turno es obtenido gracias a otra función que busca el player cuyo ID coincida con el turno actual del TDA juego). Las banderas son valores simbólicos o booleanos que indican, por ejemplo, si el jugador desea comprar una propiedad, usar una carta para salir de la cárcel o construir un hotel.

El uso de esta función, llamada **“juego-jugar-turno”**, permite mantener un estilo puramente funcional, evitando el uso de variables y generando un nuevo estado tras cada turno. El lanzamiento de los dados se hace al momento de llamar la función, se hace uso de la función **“(juego-lanzar-dados . seedList)”** en cada llamada para generar los dados. Adicionalmente, la función principal coordina acciones como mover al jugador, aplicar efectos de la casilla donde cae, gestionar compras o pagos de renta, y verificar si el jugador ha quedado en bancarrota. Cada paso produce un nuevo estado del juego, lo que permite seguir el desarrollo del turno de forma transparente.

Aspectos de la implementación

El desarrollo fue realizado en el lenguaje **Scheme (Racket 8.14)**, utilizando únicamente funciones nativas, puras y estructuras inmutables, en conformidad con el paradigma funcional.

- Para la **gestión de entradas**, se diseñó una función que simula la interacción del usuario a través de un script, permitiendo cargar turnos predefinidos para su prueba y validación. Cada acción del juego (como comprar una propiedad o pagar renta) corresponde a una función pura que recibe el estado actual del juego (o un TDA contenido en él) y devuelve uno nuevo.

- La **detección de bancarrotas** se realiza internamente en la función **“juego-jugar-turno”**, evaluando el saldo e imprimiendo un aviso en caso de bancarrota.

- La **representación textual del tablero** se realiza mediante dos listas de pares: una de propiedades asociadas a una posición, y otra de casillas

especiales asociadas a una posición.

- Para simular **cartas de eventos**, se implementaron funciones de orden superior que permiten definir acciones personalizadas asociadas a cada carta, facilitando su uso como estructuras dinámicas.

- La modularidad del proyecto facilita la extensión del juego, como agregar nuevas casillas, tipos de cartas o mecánicas adicionales (ej. hipotecas o subastas).

Instrucciones de uso

1. Descargar el archivo comprimido .zip
2. Localizar la carpeta en el ordenador, descomprimir y acceder.
3. Leer los scripts de prueba y ejecutarlos.
4. Observar detenidamente la salida del script.
5. Es posible crear más juegos de prueba si se sigue la estructura de los scripts entregados.

Resultados

El desarrollo de **Capitalia** es completamente funcional y cumple con los requisitos principales del proyecto. El sistema de turnos se ejecuta de manera correcta, alternando entre jugadores sin errores y gestionando adecuadamente el flujo del juego. Las funciones asociadas a las **cartas** se aplican correctamente al jugador correspondiente, modificando el estado del juego de forma coherente con sus efectos definidos.

Las **rentas**, **compras de propiedades** y **eventos especiales** como "Ir a la cárcel" o "Suerte" funcionan según lo esperado. Estas funcionalidades han sido validadas mediante una serie de **scripts de prueba**, los cuales simulan diversas partidas y situaciones límite. Estos scripts permiten verificar la robustez del programa y aseguran su correcto comportamiento bajo diferentes configuraciones de tablero y jugadas.

La **visualización del estado del juego** se actualiza de forma precisa tras cada acción, facilitando el seguimiento del desarrollo de la partida.

Cabe señalar que algunos aspectos fueron dejados fuera del alcance de esta implementación, tales como el **cobro de dinero al pasar por la casilla de salida**, la **hipoteca de propiedades**, y mecanismos avanzados como **negociación** o **subastas**, por considerarse secundarios frente a los objetivos principales del proyecto. Además, la selección aleatoria de cartas implementada no es determinista, por lo que algunos de los resultados entregados por los scripts pueden variar en cada ejecución.

Conclusiones

El desarrollo de **Capitalia** demostró la viabilidad de implementar un juego de mesa complejo, como Monopoly, utilizando exclusivamente el **paradigma funcional** en el lenguaje **Scheme**. A pesar de las restricciones impuestas (como la inmutabilidad de los datos y la ausencia de construcciones imperativas como let o set!), fue posible modelar correctamente el estado del juego y sus transiciones a través de funciones puras.

El uso de **TDAs** bien definidos permitió estructurar el código de forma modular y clara, facilitando tanto el desarrollo como la extensión futura del sistema. La separación entre jugadores, propiedades, tablero, cartas y lógica del juego asegura una modularidad limpia y bajo acoplamiento entre las estructuras.

Además, el proyecto resalta la capacidad del paradigma funcional para abordar sistemas dinámicos complejos de múltiples entidades, mediante **composición funcional** y **recursión**.

Referencias

- [1] *Racket Documentation* – https://docs.racket-lang.org
- [2] Friedman, D. P., & Felleisen, M. (2001). *The Little Schemer* (4th ed.). MIT Press.
- [3] Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press.
- [4] Hasbro. *Monopoly: Official Rules*.