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

Laboratorio 1: Paradigma Funcional (Scheme)
Paradigmas de Programación

Alumna: Isidora Toledo Tapia
21.748.592-4

Profesor: Gonzalo Martínez Ramírez

Sección: 13310-0 A-1 GM

Índice de Contenidos

1. INTRODUCCIÓN

Este informe tiene como propósito **documentar el proceso de análisis, diseño e implementación de una solución basada en el paradigma funcional** para la creación de un juego de mesa estratégico de compra y venta de propiedades utilizando el lenguaje Scheme.

2. DESCRIPCIÓN DEL PROBLEMA

El objetivo del laboratorio fue **implementar una versión funcional del juego de mesa CAPITALIA**, utilizando exclusivamente el lenguaje Scheme (Racket). El principal reto consistió en **adaptar un juego que normalmente depende de estados cambiantes (como posiciones, dinero, propiedades, turnos, etc.) a un entorno basado en estructuras inmutables.**

En lugar de modificar directamente el estado de los jugadores, propiedades o del tablero, cada operación debía retornar una nueva versión de la estructura afectada. Esto implicó el diseño de un conjunto de TDAs (Tipos Abstractos de Datos) que modelaran con precisión todos los elementos del juego: jugadores, propiedades, tablero, cartas de suerte y el propio juego. Además, se debía implementar la lógica de avance por turnos, aplicar reglas específicas como la construcción de casas, hipotecas, cobro de renta, cárcel, entre otras, y garantizar que todas las acciones del juego se basaran en funciones puras.

3. DESCRIPCIÓN DEL PARADIGMA

Para este laboratorio se utilizó el paradigma funcional, el cual es un estilo de programación que se basa en el uso de funciones puras, es decir, funciones que dependen únicamente de sus argumentos de entrada. En este paradigma, el flujo de datos se realiza a través de funciones que retornan nuevos valores, sin alterar directamente los existentes.

Durante el desarrollo del laboratorio se aplicaron varios principios fundamentales de este paradigma:

- **Inmutabilidad**: Una vez que un TDA (por ejemplo, jugador o propiedad) es creado, no se modifica, sino que se genera una nueva versión con los cambios aplicados.

- **Recursión**: Dado que no se pueden usar bucles tradicionales como while o for, todas las iteraciones se resuelven mediante funciones recursivas.
- **Composición de funciones**: Muchas funciones se construyeron combinando otras más pequeñas y específicas, lo que favoreció la reutilización del código.
- **Funciones de orden superior**: En algunos casos se utilizan funciones que reciben otras funciones como parámetros o que retornan funciones.

4. ANÁLISIS DEL PROBLEMA

El proyecto requirió de implementar múltiples TDAs: jugador, propiedad, carta, tablero y juego. Además, se debió desarrollar funciones que permitieran simular acciones como lanzar dados, mover jugadores, construir casas, hipotecar propiedades, pagar rentas, entre otras.

Además, exige implementar una cantidad significativa de requisitos funcionales (RF), los cuales representan distintas acciones o comportamientos del juego:

- **RF01–RF10**: Representan los TDAs básicos como jugador, propiedad, tablero, carta, y sus respectivas operaciones (creación, modificación, obtención de datos).
- **RF11–RF20**: Se enfocan en funciones de juego como compra de propiedades, avance en el tablero, construcción de casas, pago de renta, etc.
- **RF21**: Implementación del ciclo de juego con turnos, lo cual exige componer muchas de las funciones anteriores.

5. DISEÑO DE LA SOLUCIÓN

Se optó por diseñar un conjunto de TDAs, cada uno con su representación, constructores, selectores y modificadores.

1. **TDA Jugador**: Representado como una lista que almacena información relevante del jugador, incluyendo su ID, nombre, dinero disponible, posición en el tablero, propiedades adquiridas, entre otros atributos.

2. **TDA Propiedad**: Modelado como una estructura que contiene información clave de cada propiedad del tablero, como el nombre, precio, tipo, renta, estado de construcción (casas u hoteles), y su dueño (referencia al TDA Jugador).

3. **TDA Tablero**: Representado como una lista o estructura que agrupa todas las casillas del juego, incluidas propiedades, cartas, impuestos, y otros espacios especiales. Contiene una colección completa de TDAs Propiedad y otras entidades relevantes.

4. **TDA Juego**: Es el controlador principal del sistema. Se encarga de mantener y gestionar el estado general del juego, incluyendo la lista de jugadores, el turno actual, el tablero (TDA Tablero), y otras dinámicas como el lanzamiento de dados, extracción de cartas, y verificación de condiciones de victoria.

El diseño general del turno se centra en la función principal juego-jugar-turno, que recibe como parámetros el estado actual del juego, un lanzamiento de dados y varios parámetros booleanos que simulan las decisiones del jugador. Esta función se apoya en las demás funciones implementadas en el juego para gestionar el flujo de cada turno.

Para determinar qué acciones llevar a cabo, se utilizan condicionales que evalúan cada posible acción y ejecutan la correspondiente, si es necesario. Además, cada turno requiere actualizar el estado de varios elementos, como el jugador, el tablero, las propiedades y el propio juego, garantizando así que el progreso del juego se mantenga coherente en todo momento.

6. ASPECTOS DE LA IMPLEMENTACIÓN

El proyecto se estructuró en múltiples archivos.rkt, separados por TDA.

- player.rkt
- property.rkt
- board.rkt
- game.rkt
- card.rkt

Se utilizó el intérprete DrRacket superior a la versión 6.11 con el lenguaje estándar #lang racket. No se emplearon bibliotecas externas, cumpliendo con las exigencias del laboratorio.

La documentación fue realizada mediante comentarios por función, incluyendo:

- Dominio: tipos de entrada esperados.
- Recorrido: tipo de valor retornado.
- Descripción del propósito y comportamiento

.

7. INSTRUCCIONES DE USO

Para ejecutar la simulación, se deben correr los archivos script_base, script1 y script2. Además del archivo de pruebas, donde se ven algunas de las funciones requeridas.

8. RESULTADOS OBTENIDOS

Se logró implementar la mayoría de los requerimientos funcionales establecidos, con excepción del número 13: TDA Jugador – Otros – Calcular renta, el cual no fue desarrollado. Por otro lado, el requerimiento funcional 21: TDA Juego – Modificador – Realizar turno, fue parcialmente cumplido, ya que no se logró implementar la actualización correcta del turno y solo permite la decisión de compra de propiedad.

A continuación, se presenta una tabla que resume el cumplimiento de los requerimientos funcionales definidos para el sistema. Cada requerimiento representa una funcionalidad vinculada a los TDAs implementados. El valor "1" indica que el requerimiento fue implementado correctamente, mientras que "0" señala que no se logró implementar por completo o se encuentra pendiente de desarrollo.

Código	Descripción	Implementado
RF1	TDAs	1
RF2	TDA Jugador - constructor	1
RF3	TDA Propiedad - constructor	1
RF4	TDA Carta - constructor	1
RF5	TDA Tablero - constructor	1
RF6	TDA Juego - constructor	1
RF7	TDA Tablero - modificador - Agregar propiedad	1
RF8	TDA Juego - modificador - Agregar jugador	1
RF9	TDA Juego - selector - Obtener jugador actual	1
RF10	TDA Juego - otros - Lanzar dados	1
RF11	TDA Jugador - modificador - Mover jugador	1
RF12	TDA Jugador - modificador - Comprar propiedad	0.75
RF13	TDA Jugador - otros - Calcular renta	0
RF14	TDA Propiedad - otros - Calcular renta	1
RF15	TDA Propiedad - modificador - Construir casa	1
RF16	TDA Propiedad - modificador - Construir hotel	1
RF17	TDA Jugador - otros - Pagar renta	1
RF18	TDA Propiedad - Hipotecar propiedad	1
RF19	TDA Juego - modificador - Extraer carta	1
RF20	TDA Jugador - otros - Verificar bancarrota	1
RF21	TDA Juego - modificador - Realizar turno	0.25

9. CONCLUSIONES

Este laboratorio permitió comprender profundamente las características del paradigma funcional. Aunque al principio resulta contraintuitivo programar sin variables mutables, el enfoque funcional promueve una estructura más clara, modular y predecible del código.

Sin embargo, también se detectaron ciertas dificultades, especialmente en la gestión del estado del juego, ya que **cada modificación implica la creación de una nueva versión del estado completo**, lo cual puede volver el código más complejo

En general, se lograron la mayoría de los objetivos, y los errores encontrados sirvieron para comprender mejor los límites y ventajas de este paradigma.

10. REFERENCIAS

Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press.

Racket Documentation. (n.d.). The Racket Reference. https://docs.racket-lang.org/reference/