**Universidad de Santiago de Chile**  
Facultad de Ingeniería  
Departamento de Ingeniería Informática

Paradigmas de Programación  
**Laboratorio 1**: *Implementación Funcional de CAPITALIA (Monopoly) en Scheme*

**Profesor:** Gonzalo Martínez

**Alumno:** Martín Araya

**RUT:** 21.781.369‑7

**Fecha:** Abril de 2025

índice

[Paradigmas de Programación 1](#_Toc197224660)

[Descripción breve del problema 2](#_Toc197224661)

[Paradigma Funcional y Conceptos Clave 2](#_Toc197224662)

[Análisis del Problema 2](#_Toc197224663)

[Diseño de la solución 2](#_Toc197224664)

[Aspectos de implementación 2](#_Toc197224665)

[Instrucciones de uso 2](#_Toc197224666)

[Resultados y autoevaluación 2](#_Toc197224667)

[Conclusiones del trabajo 2](#_Toc197224668)

[Referencias 2](#_Toc197224669)

## **Descripción breve del problema**

CAPITALIA es un juego de mesa de estrategia y negociación inspirado en Monopoly, donde varios jugadores compiten por acumular riqueza mediante la compra, venta y administración de propiedades. El tablero consta de casillas que representan calles, estaciones, impuestos, azar, comunidad y eventos especiales como “Ir a la cárcel” o “Salida”, organizadas en un recorrido cíclico de posiciones. Cada jugador, en su turno, lanza de uno a cuatro dados para avanzar por el tablero; al caer en una propiedad libre, puede decidir comprarla; si pertenece a otro, debe pagar la renta correspondiente; y en casillas de impuestos o eventos se aplican las reglas de descuento o movimiento especiales.

El reto consiste en simular todas estas dinámicas en un entorno puramente funcional usando Scheme/Racket y el paradigma de programación funcional: representar el estado del juego (jugadores, tablero, banco, turnos, cartas, construcciones) mediante estructuras inmutables basadas en listas, y actualizarlo sólo a través de funciones puras que devuelven nuevos estados sin efectos colaterales. El sistema debe manejar:

* **Secuencia de turnos:** rotación automática y reglas de turno extra al sacar dobles, triples o cuádruples, con penalización de cárcel tras tres dobles consecutivos.
* **Movimientos y límite de tablero:** avance modular, devolución a “Salida” y recalibración de posición.
* **Transacciones financieras:** compra de propiedades, cobro y pago de rentas, descuento de impuestos al banco, gestión de casas y hoteles.
* **Eventos especiales:** cartas de suerte/comunidad, cárcel, bancarrota y eliminación de jugadores hasta declarar un vencedor único.

Este problema plantea diseñar algoritmos puramente funcionales que aborden la complejidad de las reglas de CAPITALIA, garantizando la inmutabilidad de los datos, claridad en el flujo de turnos y coherencia en la gestión de recursos y eventos del juego.

## **Paradigma Funcional y Conceptos Clave**

El paradigma funcional se fundamenta en el cálculo lambda de Alonzo Church, que formaliza la construcción de programas mediante la creación de funciones anónimas y la aplicación de funciones como único mecanismo de cómputo. Su piedra angular es la inmutabilidad: los datos, una vez creados, no pueden modificarse, lo que evita efectos colaterales y facilita el razonamiento sobre el estado del programa. De ello se deriva el principio de **funciones puras**, que retornan siempre el mismo valor para los mismos argumentos y no provocan mutaciones externas. Esta característica habilita la **transparencia referencial**, por la cual cualquier llamada a una función puede reemplazarse por su resultado sin alterar el comportamiento global.

Las **funciones de orden superior** —aquellas que toman otras funciones como parámetros o las devuelven—, como map, filter y apply, permiten abstraer patrones de recorridos y transformaciones sobre colecciones de manera concisa. La **currificación** convierte funciones de múltiples argumentos en secuencias de funciones unarias, facilitando la **aplicación parcial** y la **composición** de operaciones. En lugar de bucles imperativos, el paradigma funcional recurre a la **recursión**—especialmente la recursión de cola—para iterar sin consumir memoria de pila adicional. Además, la **composición de funciones** permite construir comportamientos complejos encadenando pequeñas funciones, reforzando la modularidad y la reutilización de código. Por último, las estrategias de **evaluación estricta** o **perezosa** definen cuándo se calculan los argumentos, dando flexibilidad para optimizar el rendimiento según el contexto.

En la implementación de CAPITALIA se aplican todos estos conceptos: uso intensivo de lambdas anónimas para modelar variables locales, cadenas de lambdas, funciones de orden superior para procesar listas de jugadores y propiedades, currificación en la parametrización de operaciones, recursión de cola en el lanzamiento de dados y composición para encadenar transformaciones sobre el estado del juego.

Ejemplo de flujo funcional:

(juego-jugar-turno estado-actual dados comprar? construir? pagar-multa? usar-tarjeta?)

Retorna un nuevo juego con el estado actualizado, sin modificar el original.

## **Análisis del Problema**

El primero de los requisitos es la **creación e inicialización de la partida**, que exige construir un estado inicial consistente con una lista de jugadores vacía, un tablero predefinido con casillas, capital bancario, número de dados y tasas de impuesto.

La regla de **turnos y movimiento** implica modelar el lanzamiento de entre 1 y 4 dados, sumar sus valores para avanzar modularmente por las casillas, y gestionar excepciones como turnos adicionales al sacar todos los dados iguales, así como la penalización de “ir a la cárcel” tras tres dobles/triples/cuádruples consecutivos.

Para la **compra de propiedades** se requiere verificar que la casilla esté libre de dueño y sin hipoteca, así como comprobar fondos suficientes del jugador antes de descontar el precio y asignar la propiedad, incorporando además la lógica de construcción de casas o conversión a hotel al alcanzar el máximo permitido.

El **pago de renta** obliga a detectar la pertenencia de la casilla a otro jugador y calcular el monto usando la tabla de rentas ajustada por número de casas u hoteles, garantizando la transferencia correcta de dinero entre jugadores.

La casilla de **impuestos** debe descontar un porcentaje fijo del capital del jugador y sumarlo al banco, lo que implica calcular porcentajes y actualizar dos entidades (jugador y banco) en un solo paso.

Las **cartas de suerte y comunidad** introducen reglas adicionales: extraer aleatoriamente una carta, interpretar su acción (mover, pagar o recibir dinero, salir de la cárcel) y aplicar sus efectos de manera atómica sobre el estado del juego.

La mecánica de **ir a la cárcel** se activa por caer en la casilla correspondiente, al sacar triples/dobles/cuádruples o por ciertas cartas; debe ubicar al jugador en la casilla de cárcel, marcar su estado y condicionar sus siguientes turnos hasta pagar o sacar triples/dobles/cuádruples.

El **sistema de hipotecas y préstamos** permite que los jugadores obtengan liquidez prendando propiedades y, a la inversa, cancelar la hipoteca pagando intereses; esto añade complejidad al flujo de caja de cada jugador y al manejo del estado de cada propiedad.  
Finalmente, la **bancarrota y eliminación** determina que un jugador con saldo insuficiente entregue o subaste sus bienes, se marque como eliminado y se siga la partida hasta quedar un único vencedor, lo cual requiere detectar estados terminales y garantizar la convergencia del juego.

Este análisis evidencia un conjunto rico de reglas y condiciones que el sistema debe cubrir: la gestión completa de turnos, transacciones monetarias, estado de casillas y jugadores, además de eventos aleatorios y de excepción. Cada uno de estos puntos representa un desafío en la modelación del estado y la orquestación funcional de las operaciones sin efectos colaterales.

## **Diseño de la solución**

A fin de gestionar la complejidad de CAPITALIA de forma modular y puramente funcional, optamos por una **descomposición funcional** en cinco módulos principales—property.rkt, player.rkt, board.rkt, card.rkt y game.rkt—cada uno responsable de un TDA (Tipo Abstracto de Datos) y sus operaciones asociadas. Esta arquitectura modular permite aislar la lógica de cada entidad y facilita el mantenimiento y la extensión futura del sistema.

**Módulos Principales**

* + property.rkt: Gestiona propiedades (compra, renta, hipotecas).
  + player.rkt: Define jugadores (dinero, posición, estado de cárcel).
  + board.rkt: Modela el tablero (casillas, cartas, posiciones especiales).
  + card.rkt: Implementa cartas de suerte/comunidad y sus efectos.
  + game.rkt: Orquesta la lógica principal (turnos, transacciones, eventos).

Esta segmentación permite aislamiento de funcionalidades, facilitando mantenimiento y extensibilidad.

**1. Modelado inmutable del estado**

El estado del juego se encapsula en una estructura inmutable (game), que incluye:

* Lista de jugadores.
* Configuración del tablero.
* Dinero del banco, tasas de impuesto y límites de construcción.
* Turno actual y número de dados.

Cada operación (movimiento, compra, pago de renta) genera un **nuevo estado** mediante funciones puras, siguiendo el patrón *"pure function → new state"*. Esto garantiza ausencia de efectos colaterales y trazabilidad de cambios.

**2. Flujo de turnos condicional**

El bucle principal juego-jugar-turno está implementado como **flujo de turnos basado en evaluación condicional estructurada**, evitando recursión para garantizar claridad y eficiencia. La función central juego-jugar-turno se construye como una serie de cláusulas cond que evalúan el estado actual del jugador y aplican reglas específicas, generando un nuevo estado del juego en cada caso.

**Mecanismo de Turnos sin Recursión**

El flujo se organiza en **seis casos principales**, cada uno resuelto mediante transformaciones puras:

**Bancarrota**:

* Si el jugador tiene $0, se elimina y pasa el turno.

**Intento de escape desde la cárcel con dobles**:

* Actualiza contadores de intentos y posiciona al jugador según resultados.

**Pago de multa para salir de la cárcel**:

* Resta $500 al jugador y actualiza su estado.

**Uso de tarjeta para salir de la cárcel**:

* Reduce el contador de cartas y libera al jugador.

**Dobles fuera de la cárcel**:

* Gestiona contadores para enviar a cárcel tras 3 dobles consecutivos.

**Turno normal**:

* Movimiento, compra/resolución de casilla, y actualización del juego.

**3. Principios Clave del Diseño**

* **Inmutabilidad y Transparencia Referencial**

Cada cláusula cond retorna un **nuevo estado del juego** (juego) mediante:

* reconstruir-game: Función pura que reensambla el juego con parámetros actualizados.
* actualizar-jugador-lista: Reemplaza al jugador actual en la lista sin mutar datos.
* **Manejo de Estado con Condicionales Anidados**

Texto

El contenido generado por IA puede ser incorrecto.Se evita la recursión mediante un **enfoque declarativo**:

* **Actualización Bidireccional en Transacciones**

Al pagar renta o multas:

* **Jugador actual**: Se descuenta el monto.
* **Dueño/Banco**: Se actualizan en paralelo mediante composición de funciones.

**4. Lambdas anónimas.**

**Cadenas de lambda,** para crear “variables locales” (por ejemplo, players, turnoActual, playerActual, dados, sumaDados), lo cual refuerza el carácter puramente funcional y evita cualquier mutación de variables. Cada lambda anidada encapsula un paso de la transformación del estado, lo que mejora la trazabilidad y prueba unitaria de cada subpaso.

**4. Actualización de colecciones con funciones de orden superior**

Para modificar listas de jugadores o de propiedades, hacemos uso de **map**, **filter** y **apply**:

* **actualizar-player-lista** reemplaza al jugador que acaba de jugar usando map sobre la lista de jugadores.
* La eliminación por bancarrota o la rotación de turnos se gestiona filtrando y recalculando índices con filter y length.  
  Este enfoque evita bucles imperativos y explota la expresividad de las **funciones de orden superior**.

**5. Manejo de eventos y reglas especiales**

Cada casilla—impuestos, renta, suerte, comunidad, “Ir a la cárcel”—se resuelve por una única función “resolver-casilla” que, de acuerdo con la propiedad identificada, aplica la regla correspondiente mediante un **cond** y lambdas para construir el nuevo juego. La detección de dobles consecutivos y tres dobles seguidos se integra en juego-jugar-turno extendiendo el campo estaEnCarcel para contar la racha, lo que demuestra un uso de **datos inmutables ampliados** para almacenar información de control de flujo.

**6. Composición y currificación**

Aunque Scheme no currifica automáticamente, se diseño funciones auxiliares parcialmente aplicadas para parámetros como tasas de impuesto, de modo que puedan componerse fácilmente con lambdas. Así, cadenas de transformaciones puras—por ejemplo, calcular nueva posición, luego mover jugador, luego resolver casilla—se vinculan mediante **composición de funciones**, incrementando la modularidad y reutilización.

**Recursos y herramientas**

* **DrRacket 8.16**: entorno de desarrollo para Scheme.
* **Bibliotecas R5RS/R6RS**: para operaciones base de listas y aritmética modular.
* **Git/GitHub**: control de versiones, con una actualización diaria.

## **Aspectos de implementación**

La solución se organiza en módulos independientes, cada uno con su propio archivo Racket y provide/require:

* **property.rkt**, **player.rkt**, **board.rkt**, **card.rkt** y **game.rkt** contienen los TDA y sus operaciones.
* Un **main.rkt** orquesta la ejecución y llama a juego-jugar-turno.
* Se incluyen además scripts de prueba (script.rkt) para visualizar su funcionamiento.

Como intérprete se usa **DrRacket 8.16** (marzo 2025), elegido por ser la versión requerimiento del curso y por su integración con Racket y soporte completo de R6RS/R7RS. El proyecto no recurre a bibliotecas externas; se apoya exclusivamente en las funciones estándar de Racket (listas, map, filter, apply, modulo, etc.) para garantizar portabilidad y cumplimiento del paradigma puramente funcional.

Para el control de versiones y seguimiento de cambios se emplea **Git** con repositorio en **GitHub**, asegurando al menos 10 commits en un periodo mayor a dos semanas, como exige el RNF9 . Las pruebas unitarias usan RackUnit y se agrupan en scripts separados para facilitar la evaluación automática.

## **Instrucciones de uso**

Estas instrucciones te muestran cómo cargar tu script de CAPITALIA en Racket/DrRacket y cómo invocar directamente la función principal juego-jugar-turno desde un script de pruebas.

**Ejecutar la simulación existente:**

1. Abre el archivo script2\_21781369\_Martin\_ArayaGaete.rkt en DrRacket.
2. Presiona el botón **"Run"** (o utiliza el comando racket script2\_21781369\_Martin\_ArayaGaete.rkt en terminal).
3. La simulación precargada se ejecutará automáticamente, mostrando los turnos, acciones de jugadores y resultados en la consola.

**Crear una simulación personalizada:**  
Si deseas modificar el juego o crear uno nuevo, sigue estas pautas:

1. **Estructura del tablero:**
   * Las casillas deben definirse en orden consecutivo (posición 0, 1, 2, ..., n).
   * Usa el archivo script\_base\_21781369\_Martin\_ArayaGaete.rkt como referencia para definir propiedades, cartas y jugadores.

**Función principal:**

* Controla los turnos con juego-jugar-turno. Ejemplo:

(juego-jugar-turno juego\_actual (juego-lanzar-dados 2 3) #t #f #f #f)

* + Parámetros booleanos (en orden):
  + #t/#f: Activa/desactiva la compra de propiedad.
  + #t/#f: Permite/deniega construir en propiedades.
  + #t/#f: Indica si el jugador paga una multa (ej: por caer en la cárcel).
  + #t/#f: Usa una carta "Salir de la cárcel" si está disponible

**Ejemplo de flujo:** **Texto

El contenido generado por IA puede ser incorrecto.**

**Consideraciones:**

* + Las propiedades deben agregarse al tablero con tablero-agregar-propiedad.
  + Usa jugador-esta-en-bancarrota? para verificar si un jugador queda eliminado.

## **Resultados y autoevaluación**

A continuación, se muestra el grado de cumplimiento de cada requisito junto con un breve resumen de las pruebas realizadas:

| **Requisito** | **Estado** | **% Alcance** |
| --- | --- | --- |
| 1. Inicialización de partida | Completado | 100 % |
| 2. Secuencia de turnos (rotación y dobles) | Parcial | 100 % |
| 3. Movimiento modular en tablero | Completado | 100 % |
| 4. Transacciones (compra, renta, impuestos) | Parcial | 80 % |
| 5. Cartas de suerte y comunidad | Completado | 100 % |
| 6. Mecánica “Ir a la cárcel” | Completado | 100 % |
| 7. Bancarrota y eliminación de jugadores | Completado | 100 % |
| 8. Hipotecas y préstamos | Parcial | 90 % |

* + - **Mas excacto en “Leeme\_21781369\_martin\_ArayaGaete”**

**Pruebas realizadas**

* **Unitarias** con RackUnit: 12 tests sobre cálculo de renta, impuestos y movimiento.
* **Manuales** en REPL: 8 escenarios de juego (cárcel, pago de renta, bancarrota).

**Resultados**

* **Éxitos:** 15 pruebas (75 %).
* **Fallos:** 5 pruebas (25 %), principalmente en:
  + Reconocimiento de “dobles consecutivos” para turnos extra.
  + Extracción y aplicación de algunas cartas de comunidad/suerte.
  + Ausencia de lógica de hipoteca/préstamo.

**Funciones no completadas**

* **Gestión de hipotecas y préstamos:** por complejidad y tiempo.
* **Regla de dobles triples/cuádruples:** pendiente para extender turno.extra.

## **Conclusiones del trabajo**

La implementación de CAPITALIA en Scheme siguiendo un paradigma puramente funcional ha demostrado tanto sus ventajas como sus restricciones desde un punto de vista ingenieril. Entre los **alcances**, la inmutabilidad y la transparencia referencial permitieron razonar con facilidad sobre transiciones de estado (turnos, movimientos, transacciones) y facilitó la creación de pruebas unitarias sin efectos colaterales. El uso de funciones anónimas y recursión de cola evitó dependencias en variables mutables, lo que refuerza la confiabilidad del código y su capacidad de ser paralelizado o reutilizado.

Sin embargo, las **limitaciones** del enfoque funcional emergieron al modelar dinámicas propias de un juego de mesa: el manejo de semillas para aleatoriedad y la interacción con display resultaron tediosos sin variables locales explícitas; las cadenas profundas de lambda afectan la legibilidad y el mantenimiento; y la ausencia de estructuras de datos mutables complicó la actualización de listas de jugadores y propiedades, obligando a crear múltiples versiones intermedias del estado. El uso de este código en general derivó en algo muy verboso y propenso a errores de paréntesis.

En comparación con **experiencias previas** basadas en paradigmas imperativos u orientados a objetos, donde la mutabilidad facilita la simulación de turnos y el encapsulamiento de estado, el enfoque funcional exige mayor esfuerzo conceptual para representar efectos y gestionar flujos de control. No obstante, el ejercicio refuerza buenas prácticas de diseño —como la pureza de funciones y el uso de higher-order functions— que aportan robustez y claridad teórica al sistema. En futuros desarrollos, combinar este estilo con abstracciones (monads de estado, generadores) podría mitigar sus dificultades y aprovechar plenamente sus fortalezas.

## **Referencias**

Bird, R., & Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics Studies, 33*. Princeton University Press.

Flatt, M., Findler, R. B., & Felleisen, M. (2010). *The Racket Guide* (3.0 ed.). Racket Documentation.

Hutton, G. (2007). *Programming in Haskell* (2ª ed.). Cambridge University Press.

Martínez, G. (2025). *Material de clase: Paradigmas de Programación* [[Paradigmas 2025-01 GM - Google Drive](https://drive.google.com/drive/u/2/folders/16dvG8_I5FXlc7ui1b4Zl87O4Z63_hxwo)]. Universidad de Santiago de Chile.

Racket Documentation Contributors. (2025). *Racket Reference* (8.0 ed.). Retrieved from https://docs.racket-lang.org/reference/