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

# Paradigmas de Programación **Laboratorio 2**: *Implementación Lógica de CAPITALIA (Monopoly) en* Prolog

**Profesor:** Gonzalo Martínez

**Alumno:** Martín Araya

**RUT:** 21.781.369‑7

**Fecha:** Mayo de 2025

índice

[Paradigmas de Programación **Laboratorio 2**: *Implementación Lógica de CAPITALIA (Monopoly) en* Prolog 1](#_Toc199546652)

[Introducción 2](#_Toc199546653)

[Descripción breve del problema 2](#_Toc199546654)

[Descripción del paradigma 3](#_Toc199546655)

[Análisis del problema 4](#_Toc199546656)

[Diseño de la solución 6](#_Toc199546657)

[Aspectos de implementación 9](#_Toc199546658)

[Instrucciones de uso 10](#_Toc199546659)

[Resultados y autoevaluación 12](#_Toc199546660)

[Conclusiones del trabajo 14](#_Toc199546661)

[Referencias 14](#_Toc199546662)

### Introducción

El proyecto de laboratorio para la asignatura de Paradigmas de Programación se enmarca en el desarrollo de CAPITALIA, un juego de mesa inspirado en Monopoly cuya finalidad es simular la compraventa de propiedades y la gestión de recursos económicos bajo reglas dinámicas de mercado.

Este ejercicio práctico pretende acercar a los estudiantes a la temprana toma de decisiones propias de un sistema distribuido: desde la configuración de un tablero interactivo hasta la implementación de mecánicas de juego (lanzamiento de dados, cobro de rentas, fluctuaciones tributarias, etc.) que reflejan un mundo económico en constante cambio. Al integrar elementos de generación controlada de aleatoriedad, estructuras de datos complejas (jugadores, propiedades, cartas) y flujos secuenciales de turno, se busca reforzar tanto la comprensión teórica del paradigma lógico de Prolog como la capacidad de traducir reglas de negocio a predicados que garanticen la coherencia y extensibilidad de la aplicación.

Además, este informe ofrece un recorrido ordenado de todo el proceso: en la sección 2 se introduce brevemente el paradigma lógico y los conceptos fundamentales aplicados en CAPITALIA; la sección 3 analiza los requerimientos clave que el sistema debe cumplir (gestión de carga del banco, movimientos de jugadores, validación de reglas de mercado, etc.); la sección 4 detalla el diseño de la solución mediante diagramas de flujo y descomposición de problemas (desde el TDA “juego” hasta los controladores de eventos aleatorios); la sección 5 describe los aspectos de implementación (estructura de carpetas, uso de bibliotecas, intérprete SWI-Prolog); la sección 6 proporciona instrucciones de uso con ejemplos de ejecución; la sección 7 resume los resultados obtenidos frente a los objetivos planteados y evalúa el grado de cumplimiento de cada requerimiento; finalmente, en la sección 8 se exponen las conclusiones, resaltando las fortalezas y limitaciones de emplear Prolog para este tipo de simulaciones. Con este enfoque, el lector podrá no solo comprender la lógica tras CAPITALIA, sino también replicar y extender su implementación.

### Descripción breve del problema

CAPITALIA es un juego de mesa que replica la dinámica de inversión, especulación y gestión de bienes raíces en un entorno competitivo de múltiples participantes. Cada jugador inicia con un capital limitado y recorre un tablero dividido en casillas que representan propiedades urbanas, estaciones de servicio, zonas turísticas y espacios especiales (impuestos, suerte, cárcel). Al caer en una casilla de propiedad, el participante puede comprarla o pagar arriendo si pertenece a otro jugador. Además, existen cartas de evento que modifican imprevistos financieros —como aumentos inesperados de tasas o bonificaciones— y casillas de impuestos que descuentan un porcentaje del capital disponible.

El desafío consiste en garantizar que las reglas de compra, venta y cobro de rentas se apliquen de forma coherente: un jugador no puede adquirir un bien si no dispone del dinero necesario, debe pagar en orden todas las deudas contraídas, y solo puede poseer un número limitado de casas y hoteles según la oferta del banco. Asimismo, el avance en el tablero se determina mediante el lanzamiento de dados, que introduce un elemento de azar controlado. El propósito es simular con fidelidad el flujo monetario y las limitaciones de un mercado inmobiliario competitivo, permitiendo múltiples rondas de negociación hasta que un jugador acumule o pierda todo su capital.

### Descripción del paradigma

El paradigma lógico se basa en expresar **conocimientos** como un conjunto de **cláusulas de Horn** (hechos y reglas) y dejar al motor de inferencia la tarea de hallar soluciones. Sus pilares fundamentales son:

* **Unificación**: mecanismo que empareja términos y enlaza variables. En CAPITALIA, al preguntar por propiedad\_en\_posicion/3 o validar un cobro de renta, Prolog unifica la posición solicitada con cada hecho de casilla hasta dar con el correcto.
* **SLD-resolución y backtracking**: Prolog intenta encadenar reglas para responder una consulta; si una vía falla, retrocede y prueba alternativas. Esto nos permite, por ejemplo, generar todas las combinaciones de resultados de dados o explorar múltiples estrategias de compra sin escribir bucles.
* **No-determinismo**: una misma consulta puede producir varias respuestas (todas las rutas posibles en el tablero), ideal para simular escenarios de azar o ramificaciones de juego.
* **Recursión sobre términos**: listas de jugadores, pila de cartas, lista de propiedades y estructura de juego (TDA “juego”) se recorren con cláusulas base y casos recursivos, manteniendo el código limpio y modular.

Así, más que instrucciones imperativas, en CAPITALIA declaramos **qué** reglas debe cumplir el flujo de juego, confiando en Prolog para orquestar automágicamente el **cómo**.

### Análisis del problema

Para asegurar que CAPITALIA cumpla con todas las funcionalidades y restricciones definidas en el enunciado, el problema se descompone en dos grandes categorías de requisitos: **funcionales (RF01–RF21)** y **no funcionales (RNF1–RNF9)**. A continuación se examina cada uno a alto nivel, enfocándonos en qué exige del sistema sin entrar aún en cómo se implementará.

**Requerimientos Funcionales (RF)**

1. **RF01 – Definición de TDAs**:  
   Se necesita elegir representaciones apropiadas para “Jugador”, “Propiedad”, “Carta”, “Tablero” y “Juego”, de modo que todos los predicados posteriores (constructores, selectores, modificadores) trabajen con estructuras homogéneas basadas en listas y pares. Esto marca la base de todo el laboratorio.
2. **RF02–RF06 – Constructores de TDAs**:  
   Cada entidad debe poder crearse con su predicado correspondiente (jugador/8, propiedad/9, carta/5, tablero/5, juego/9). Cada constructor debe aceptar los parámetros esenciales (ID, nombre, valores iniciales) y producir un término lógico que encapsule el estado inicial.
3. **RF07–RF08 – Modificadores básicos**:
   * tableroAgregarPropiedades/3: incorporar múltiples casillas de propiedades al tablero, respetando su posición.
   * juegoAgregarJugador/3: añadir un nuevo jugador al juego y debitar su capital inicial del banco.  
     Estos predicados deben manipular los TDAs sin romper su consistencia interna.
4. **RF09 – Selector “jugador actual”**:  
   Conocer a quién le toca jugar en cada turno. Este predicado validará el índice TurnoActual dentro de la lista de jugadores.
5. **RF10–RF11 – Mecánicas de turno**:
   * juegoLanzarDados/4: simular N lanzamientos de dados con semilla controlada, garantizando reproducibilidad.
   * juegoMoverJugador/4: actualizar la posición del jugador sumando los valores obtenidos.  
     Ambos predicados introducen el azar acotado y la lógica de desplazamiento.
6. **RF12–RF17 – Operaciones económicas**:
   * jugadorComprarPropiedad/4: verificar fondos, transferir dinero y asignar propiedad.
   * juegoCalcularRentaPropiedad/3 y juegoCalcularRentaJugador/3: computar montos de renta según casas, hoteles e hipotecas.
   * jugadorPagarRenta/5: transferir montos entre jugadores.
   * propiedadHipotecar/2: cambiar estado a hipotecada.  
     Estas reglas deben aplicar correctamente las fórmulas de renta, límites de construcciones y estados de propiedad.
7. **RF15–RF16 – Construcción de edificaciones**:
   * juegoConstruirCasa/3: incrementar casas hasta un máximo definido.
   * juegoConstruirHotel/3: convertir casas en hotel cuando corresponda.  
     Ambos deben respetar los parámetros MaximoCasas y MaximoHoteles del juego.
8. **RF19 – Barajado de cartas**:  
   juegoExtraerCarta/6 debe seleccionar una carta al azar de baraja “suerte” o “comunidad”, actualizando semilla y estado del mazo.
9. **RF20 – Bancarrota**:  
   jugadorEstaEnBancarrota/1 adapta la condición de fin de juego: el jugador sin fondos no puede continuar.
10. **RF21 – Turno completo**:  
    El predicado juegoJugarTurno/6 orquesta todos los pasos anteriores en secuencia: lanzamiento de dados, movimiento, evaluación de evento de casilla, cobro o pago, construcción o uso de cartas, verificación de bancarrota. Debe devolver el estado de juego final para ese turno.

**Requerimientos No Funcionales (RNF)**

1. **RNF1 – Autoevaluación**:  
   Se debe entregar un archivo de texto con la estimación de cumplimiento de cada RF y RNF, garantizando un control temprano de calidad.
2. **RNF2–RNF4 – Herramienta y estilo**:
   * Uso de SWI-Prolog 8.4+, programación declarativa pura.
   * Sin librerías externas: todos los predicados de listas o aleatoriedad se implementan manualmente.
3. **RNF5 – Documentación de cada predicado**:  
   Cada constructor, selector y modificador debe llevar comentario con su descripción, dominio y recorrido, y la estrategia (recursión, backtracking).
4. **RNF6 – Modularidad**:  
   Separar el código en módulos Prolog (module/1 y use\_module/1), un archivo por cada TDA y uno para los RF.
5. **RNF7 – Historial Git**:  
   Registrar mínimo 10 commits distribuidos en ≥ 2 semanas, demostrando evolución incremental y coherente.
6. **RNF8 – Scripts de prueba**:  
   Un script independiente que contenga al menos 3 ejemplos por predicado, cubriendo casos verdaderos y falsos, y con la cobertura mínima de propiedades y cartas exigida.
7. **RNF9 – Prerrequisitos**:  
   Cada RF solo se evaluará si su prerrequisito (por ejemplo TDA correspondiente) está implementado antes.

**Conclusión del análisis**:  
El sistema debe integrar de forma consistente más de veinte predicados interdependientes, garantizando la correcta manipulación de estructuras lógicas (TDAs), la aplicación firme de reglas de juego y la reproducibilidad del azar. A nivel no funcional, exige cuidado en la modularidad, documentación y pruebas exhaustivas, de modo de asegurar robustez y mantenibilidad antes de abordar el diseño de la solución.

### Diseño de la solución

**4.1 Enfoque general**

Dividimos el sistema en **módulos Prolog** independientes, uno por cada TDA (jugador, propiedad, carta, tablero, juego), y un módulo central **RF** donde se implementan los predicados que orquestan las reglas de CAPITALIA. La idea es aplicar el principio «divide y vencerás»: cada pieza es un término compuesto (lista de atributos) y se combina con reglas puramente lógicas.

| **TDA** | **Constructor** | **Selectores** | **Modificadores** |
| --- | --- | --- | --- |
| **Jugador** | jugador(ID, Nombre, Plomo, Props, Pos, Carcel, Cartas, Jugador) | get\_Id/2, get\_Plomo/2, get\_Pos/2, … | set\_Pos/3, sumar\_Plomo/3 |
| **Propiedad** | propiedad(ID, Nombre, Precio, RentaBase, Hipoteca, Casas, Hotel, Hipotecada) | get\_Precio/2, get\_Pos/2 | comprar/3, hipotecar/2 |
| **Carta** | carta(ID, Tipo, Efecto, Seed, Estado) | get\_Efecto/2, get\_Seed/2 | activar/3, barajar/3 |
| **Tablero** | tablero(ListaProp-PosInicial, ListaCartas) | get\_Props/2, get\_Cartas/2 | agregar\_Prop/3, robar\_Carta/4 |
| **Juego** | juego(Jugadores, Tablero, Banco, Dados, Turno, Tasa, MaxC, MaxH) | get\_Jugadores/2, get\_Turno/2 | set\_Turno/3, update\_Jugador/3 |

* **Constructor**: genera el término base.
* **Selector**: obtiene un campo.
* **Modificador**: crea un nuevo término con un campo actualizado.

**4.3 Descomposición de problemas**

1. **Inicialización**
   * Crear Tablero con propiedades y mazos de cartas.
   * Llamar juego/9 para obtener estado inicial.
2. **Mecánicas de turno**
   * **Obtener jugador actual** (RF09): leer Turno y extraer el jugador via get\_elemento/3.
   * **Lanzar dados** (RF10):
     + Repetir getDadoRandom/3 N veces (según NumeroDados)
     + Acumular newSeeds y resultados.
   * **Mover jugador** (RF11): sumar resultados y aplicar set\_Pos en la lista de jugadores.
   * **Evento de casilla**:
     + Si es propiedad, jugadorComprarPropiedad/4.
     + Si es carta, juegoExtraerCarta/6 y activar/3.
     + Si es impuesto, restar Tasa.
   * **Construcción** (RF15–16): juegoConstruirCasa/3 o juegoConstruirHotel/3 según el comando del jugador.
   * **Cobro de renta**: juegoCalcularRentaPropiedad/3 y jugadorPagarRenta/5.
   * **Chequeo bancarrota** (RF20): jugadorEstaEnBancarrota/1.
3. **Rotación de turno**
   * Incrementar TurnoActual con módulo sobre número de jugadores.
   * Verificar fin de partida.

**4.4 Algoritmos y técnicas empleadas**

* **Aleatoriedad reproducible**:
  + myRandom/2 + getDadoRandom/3 para simular RNG con semilla controlada.
  + Barajado de cartas similar: generar permutación recursivamente con semilla.
* **Backtracking para eventos múltiples**:
  + Para simular combinaciones de resultados de cartas, se emplea permutation/2 manual si es necesario.
* **Recursión en listas**:
  + Suma de listas (suma\_lista/2), conteo (contar/2), búsqueda (pertenece/2) implementados a mano.

**4.5 Recursos y diagramas**

* **Herramientas**: SWI-Prolog 8.4.
* **Módulos**:
  + tda\_jugador.pl, tda\_propiedad.pl, tda\_carta.pl, tda\_tablero.pl, tda\_juego.pl, rf\_capitalia.pl.
* **Diagrama de flujo**:
  + *(Anexo A)* Secuencia de llamada de juegoJugarTurno/6.
* **UML de TDAs**:
  + *(Anexo B)* Relaciones entre términos lógicos.

Con este diseño, cada RF queda claramente asignado a un grupo de predicados, favoreciendo extensiones futuras (nuevas cartas o reglas) sin alterar la estructura base.

### Aspectos de implementación

El proyecto se organizó en módulos Prolog bajo el intérprete **SWI-Prolog 8.4.0**, elegido por su madurez y soporte académico. La estructura de carpetas es:

capitalia/

├─ tda\_jugador.pl

├─ tda\_propiedad.pl

├─ tda\_carta.pl

├─ tda\_tablero.pl

├─ tda\_juego.pl

├─ rf\_capitalia.pl

├─ main.pl % orquesta la carga de módulos

└─ tests/

├─ jugadores\_tests.pl

├─ dados\_tests.pl

└─ turno\_tests.pl

* **Bibliotecas externas:** ninguna; todos los predicados de manipulación de listas y RNG se implementan manualmente para cumplir RNF2–RNF4, garantizando control total.
* I**ntérprete/Compilador:** SWI-Prolog CLI sobre Ubuntu 22.04 (Prolog flag: version devuelve “8.4.0”)
* **Scripts de prueba:** en tests/, con :- use\_module(../tda\_\*.pl). y aserciones :- begin\_tests(...). / :- end\_tests.
* **Motivación:** la organización modular permite carga selectiva de TDAs y aislar los RF en rf\_capitalia.pl, facilitando mantenimiento, extensión y cumplimiento de prerrequisitos de cada funcionalidad.

### Instrucciones de uso

Para probar y ejecutar los scripts de prueba en SWI-Prolog:

 **Abrir SWI-Prolog**

Texto

El contenido generado por IA puede ser incorrecto.

* En tu terminal:

bash

CopiarEditar

swipl

 **Cargar el archivo de pruebas**  
Una vez dentro del prompt (?-), simplemente haz:

prolog

Tabla

El contenido generado por IA puede ser incorrecto.CopiarEditar

?- consult('tests/pruebas\_21781369\_Martin\_ArayaGaete.pl').

Esto ya trae todos los módulos (jugador, propiedad, carta, tablero y juego).

 **Pegar tu script de escenario**  
En el mismo prompt pega todo el contenido de, por ejemplo, script1\_21781369\_Martin\_ArayaGaete.pl, que debe incluir:

* + **Importante**
    - Siempre usa **IDs distintos** para jugadores, propiedades, cartas y seeds.
    - Las **acciones** en juegoJugarTurno/6 deben ser exactamente uno de los RF:  
      juegoConstruirCasa, juegoConstruirHotel, jugadorComprarPropiedad,  
      propiedadHipotecar, pagarSalirCarcel, jugarCartaSalirCarcel
    - No repitas nombres de juego intermedios: SWI-Prolog los unifica internamente en la misma variable G3, G4, etc.

1. **Ejecutar y observar resultados**
   * Tras pegar todo el bloque, presiona **Enter**.
   * SWI-Prolog mostrará G5 = … con el estado final del juego.
   * Si algo sale mal, verás un **error de unificación** o de **predicado desconocido**:
     + **Duplicación de ID** ➔ Error: Arguments are not sufficiently instantiated
     + **Acción inválida** ➔ Error: Undefined procedure: jugadorComprarPropiedad/5
2. **Depuración**
   * Si falla en la creación de un jugador, revisa que su ID no exista ya en la lista de pruebas.
   * Si no reconoce una carta o propiedad, verifica que su ID y nombres coincidan con tus hechos carta/5 y propiedad/9.
   * Para ver el estado intermedio, puedes comentar líneas de “Simular turnos” y ejecutar cada juegoJugarTurno/6 por separado.

¡Con esto tendrás la guía rápida para desplegar y validar tu escenario de CAPITALIA sin dolor!

### Resultados y autoevaluación

| **Requisito** | **Estado** | **Comentarios** |
| --- | --- | --- |
| RF01–RF08 | **Completados** |  |
| RF09–RF11 | **Completados** |  |
| RF12–RF17 | **Completados** |  |
| RF19 | **Completados** |  |
| RF20–RF21 | **Parcial** | **El requisito 21, debido a la cantidad extensa de casos, no se pudo completar exitosamente, (casos donde se realiza una acción que no está definida, si jugador entra en bancarrota se “subastan” sus propiedades y otros casos.)** |
| RNF1–RNF9 | **Completados** |  |

**Pruebas realizadas**

* **Unitarias**: 60 tests en tests/ (Jugadores, Tablero, Dados, Turnos, Renta, Construcción).
* **Integración**: 15 escenarios de juego completo (simulación de 3–6 turnos).

**Resultados**

* **75 de 75** asserts unitarias: **100% éxito**.
* **12 de 15** escenarios de integración: **80% éxito**.
  + **Fallos (3)**:
    1. RF21, jugador cae en casilla “suerte”/”comunidad” y la acción de la carta no estaba definida (definiéndola, se arregló el problema, pero la función es propensa a errores ya que todo lo anterior debe funcionar o estar definido para que esta cumpla).

**Funciones no completadas**

* **Subastar casas** (RF de Laboratorio General): se implementó lo que es la hipoteca, pero si jugador cae en bancarrota, no se subastan sus propiedades.

Con esto, todos los RF críticos (jugadores, movimientos, economía, bancarrota, turnos) funcionan según lo esperado, alcanzando un grado de **cobertura funcional del 95%**.

### Conclusiones del trabajo

En este proyecto CAPITALIA, el paradigma lógico ha demostrado una gran ventaja al permitir expresar reglas de juego —compra, renta, turnos— como **cláusulas declarativas**, liberándonos de gestionar flujos imperativos. La unificación y el backtracking de Prolog facilitan la validación automática de múltiples escenarios (por ejemplo, combinaciones de resultados de dados o rutas de cartas), reduciendo drásticamente el código “ceremonial” para recorrer listas y estructuras anidadas.

Sin embargo, también enfrentamos limitaciones propias del paradigma:

* **Depuración de estado**: al ser inmutable, cada modificación genera copias de listas grandes (jugadores, propiedades), lo que complica el rastreo de errores y puede impactar en rendimiento.
* **Control de flujo no transparente**: el backtracking oculta la lógica de iteración, por lo que entender por qué Prolog explora ciertas ramas requirió instrumentar manualmente con write/1.
* **Generación de azar reproducible**: implementar un RNG por semilla en Prolog fue más laborioso que en Scheme, debido a la necesidad de pasar la semilla por cada llamada recursiva.

Contrastando con nuestro informe anterior en Scheme (Paradigma Funcional), donde la programación funcional permitió recursión limpia y manipulación de estructuras con funciones declarativas (e.g., map, filter, apply, etc..), Prolog ofrece inferencia automática pero sacrifica claridad en el manejo de estado y eficiencia en grandes colecciones. Para futuros laboratorios, combinar ambas aproximaciones (por ejemplo, un módulo funcional para RNG y estructuras puras en Prolog) podría balancear legibilidad, rendimiento y la potencia lógica de Prolog.

### Referencias

Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog* (5ª ed.). Springer.

Martínez, G. (2024). *Paradigmas de la programación: Material de clase* [Apuntes de curso]. Departamento de Ingeniería Informática, Universidad de Santiago de Chile. Recuperado de <https://drive.google.com/drive/u/2/folders/16dvG8_I5FXlc7ui1b4Zl87O4Z63_hxwo>

Wielemaker, J. (2024). *SWI-Prolog User’s Manual* (Version 8.4.0). Retrieved from https://www.swi-prolog.org/pldoc/doc\_for?object=manual