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

Introducción

En un escenario donde las decisiones financieras, los factores de azar y las reglas estrictas moldean el destino de los participantes, surge *CAPITALIA*, una simulación inspirada en el clásico Monopoly. Este proyecto propone modelar la lógica detrás de un sistema económico competitivo mediante la representación de jugadores, propiedades y eventos especiales que interactúan dentro de un tablero urbano. La problemática central consiste en capturar y formalizar las reglas del juego: quién compra, quién cobra, quién paga, cuándo se termina un turno y bajo qué condiciones un jugador queda fuera. Todo esto debe estar regulado por un conjunto de normas coherentes que no solo administren los recursos, sino que también simulen situaciones dinámicas como el azar, las penalizaciones o la acumulación de activos.

Para enfrentar este desafío se recurre al **paradigma lógico**, una forma de programación declarativa donde el énfasis no está en describir instrucciones paso a paso, sino en **declarar hechos y reglas** que definan el comportamiento del sistema. En este paradigma, las relaciones lógicas entre entidades y condiciones permiten deducir automáticamente el flujo del juego. En lugar de preocuparse por "cómo" se llega a una solución, se describe "qué" debe cumplirse, y el motor lógico se encarga de encontrar las respuestas válidas.

Este proyecto invita, entonces, a razonar sobre el diseño de un juego desde una perspectiva basada en conocimiento y lógica formal, fomentando el pensamiento declarativo y el diseño modular de sistemas complejos.

Descripción breve del problema

CAPITALIA es un juego de mesa que emula un entorno urbano competitivo, donde los jugadores deben administrar estratégicamente sus recursos económicos mientras se desplazan por un tablero compuesto por propiedades, impuestos, cartas de azar y eventos especiales. Cada jugador inicia con un capital limitado y, por turnos, avanza según el resultado de los dados, enfrentándose a decisiones como comprar terrenos, pagar arriendos, construir casas u hoteles, hipotecar propiedades o resolver eventos inesperados.

El desafío radica en que el sistema debe representar fielmente las reglas del juego: validar si un jugador puede pagar o comprar, aplicar correctamente las rentas, restringir la cantidad de edificaciones disponibles según el banco, manejar situaciones como caer en la cárcel o robar una carta, y determinar cuándo un jugador entra en bancarrota. Además, el azar debe estar controlado pero reproducible, y el sistema debe permitir que múltiples turnos se encadenen sin romper la lógica del juego.

Este problema no solo requiere una estructura clara de entidades que interactúan entre sí, sino también una forma de representar y hacer cumplir las reglas de forma precisa, reflejando las tensiones y decisiones estratégicas de un juego económico realista.

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 auto mágicamente el **cómo**.

Análisis del problema

Inicialización de la partida. Se debe generar un estado inicial consistente que incluya: lista de jugadores vacía; tablero con casillas (propiedades, especiales e índices); capital bancario inicial; cantidad de dados; y tasas e impuestos vigentes. Para ello, se definen TDAs uniformes ("Jugador", "Propiedad", "Carta", "Tablero", "Juego"), cuyos constructores reciben parámetros mínimos (ID, nombre, montos, posiciones) y devuelven términos lógicos válidos.

Turnos y movimiento. La simulación de dados debe emplear semillas controladas para reproducibilidad. Con el resultado, se desplaza al jugador modularmente por el tablero, aplicando reglas de dobles consecutivos (turnos extra) o penalizaciones de cárcel al lanzar idénticos tres veces. Al aterrizar, se validan casillas especiales: "Impuesto" (descuento fijo o porcentual), "Ve a la cárcel" (cambio de estado/posición) y "Suerte/Comunidad" (extracción aleatoria de carta, interpretación de su acción y aplicación atómica de efectos).

Compra y construcción. Para comprar, la casilla debe no tener dueño, no estar hipotecada y el jugador debe contar con capital suficiente. Entonces, se descuenta al jugador, se transfiere al banco y se añade la propiedad a su lista. Para edificar, al alcanzarse el límite de casas, estas se transforman en hotel, descontando montos y actualizando renta, respetando los límites definidos en el estado.

Cobro de renta e impuestos. Si un jugador cae en propiedad ajena, el sistema identifica al propietario, calcula renta según casas, hoteles e hipotecas, y transfiere fondos entre visitante y propietario. La casilla de impuesto descuenta monto fijo o porcentaje, actualiza saldos del jugador y del banco de forma atómica para mantener el invariante de suma total de dinero.

Cartas de suerte y comunidad. Al caer en estas casillas, se extrae una carta al azar con la acción de "mover, ganar/perder dinero, cambiar tasa de impuesto". Cada efecto debe ejecutarse sin inconsistencias.

Cárcel. Se activa por casilla, dobles/triples/cuádruples consecutivos o ciertas cartas. El jugador se sitúa en la cárcel y sus próximos turnos quedan condicionados hasta pagar multa, comodín o lanzar combinaciones válidas.

Hipotecas y préstamos. Al hipotecar, la propiedad pasa a estado hipotecada y el jugador recibe 200₩.

Bancarrota. Ocurre cuando un jugador no puede cubrir una deuda ni liquidar ni hipotecar propiedades. Se le declara eliminado, se subastan o distribuyen sus propiedades según reglas, se transfieren saldos remanentes y la partida continúa hasta un vencedor único. Es esencial detectar este estado y asegurar convergencia sin bucles infinitos.

Requisitos no funcionales. CAPITALIA debe implementarse en SWI-Prolog 8.4+ sin bibliotecas externas, creando manualmente predicados de lista y generador pseudoaleatorio. Cada predicado (constructor, selector, modificador) requiere documentación clara (dominio, recorrido, estrategia). El código debe organizarse en módulos Prolog por TDA y otro para lógica de RF, asegurando limpieza y mantenibilidad.

Diseño de la solución

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.pl	jugador(Id,Nombre, Dinero, Propiedades, PosicionActual, EstaEnCarcel, TotalCartasSalir, TdaJugador)	get_Id/2, get_Nombre/2, get_ PosicionActual /2, ...	set_IdJugador/3, set_Nombre /3,...
Propiedad propiedad.pl	propiedad(Id, Nombre, Precio, Renta, Dueño, Casas, EsHotel, EsHipotecada, TdaPropiedad)	get_PrecioPropiedad/2, get_RentaPropiedad /2,...	jugadorComprarPropiedad/4, propiedadHipotecar/2,..
Carta carta.pl	carta(Id, Tipo, Descripcion, Accion, TdaCarta)	get_Accion/2, get_Tipo/2, ...	set_AccionCarta/3, set_TipoCarta/3,...
Tablero tablero.pl	tablero(Propiedades, CartasSuerte, CartasComunidad, CasillasEspeciales, TdaTablero)	get_Props/2, get_Cartas/2,...	agregar_Prop/3, juegoExtraerCarta/4, /4, ...
Juego juego.pl	juego(Jugadores, Tablero, Banco, Dados, Turno, Tasa, MaxC, MaxH)	get_Jugadores/2, get_Turno/2,..	set_Turno/3, juegoJugarTurno/6,..

*Figura 1: "**Módulos Prolog**"*

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

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 vía juegoObtenerJugadorActual /2, **Lanzar dados** (RF10), **Mover jugador** (RF11), **Evento de casilla** (Si es propiedad, jugadorComprarPropiedad/4. Si es carta, juegoExtraerCarta/6.Si es impuesto, cobrarlo).
- **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.
- **Rotación de turno**
- Incrementar TurnoActual con módulo sobre número de jugadores.
- Verificar fin de partida.

Algoritmos y técnicas empleadas

- **Aleatoriedad reproducible**:
  - myRandom/2 + getDadoRandom/3 para simular RNG con semilla controlada.
- **Recursión en listas**:
  - Suma de listas (suma_lista/2), búsqueda (member/2)

Recursos y diagramas

- **Herramientas**: SWI-Prolog 8.4.
- **Git/GitHub:** control de versiones, con una actualización diaria.
- **Módulos**:
  - jugador.pl, propiedad.pl, carta.pl, tablero.pl, juego.pl,

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/
├─ jugador.pl, propiedad.pl, carta.pl, tablero.pl, juego.pl, pruebas.pl, script.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 capitalia/ llamados script1.pl, scrip2.pl y scripbase.pl

- **Motivación:** la organización modular permite carga selectiva de TDAs y aislar los RF, facilitando mantenimiento, extensión y cumplimiento de prerrequisitos de cada funcionalidad.

Instrucciones de uso

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

Ejecutar SWI-Prolog y abrir el archivo "pruebas_21781369_Martin_ArayaGaete.pl" en modo consulta.

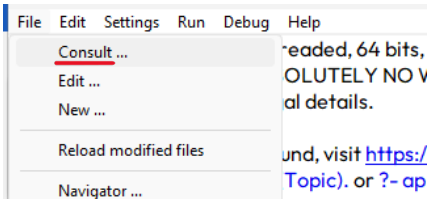

Figura 2: "Consulta Prolog"

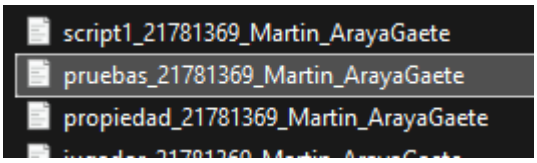

Figura 3: "Abrir archivo prolog"

Luego, Copiar y pegar lo que está en los archivos script.

```
% 1) Creo 2 jugadores
jugador(21, "Martín",    21, [], 0, false, 21, J1),
jugador(2, "Catalina", 4, [], 0, false, 0, J2),

% 2) Creo 10 propiedades normales
propiedad(2,  "Cerro San Cristóbal", 25000, 1200, 21, 0, false, false, P2),
propiedad(3,  "Plaza Italia",        18000,  900, 21, 0, false, false, P3),
propiedad(4,  "Bellavista",          22000, 1100, 21, 0, false, false, P4),
propiedad(5,  "Vitacura",            24000, 1300, null, 0, false, false, P5),
propiedad(6,  "Barrio Italia",       16000,  800, null, 0, false, false, P6),
propiedad(7,  "Ñuñoa",               20000, 1000, null, 0, false, false, P7),
propiedad(8,  "Providencia",         26000, 1400, null, 0, false, false, P8),
propiedad(9,  "Las Condes",          30000, 1500, null, 0, false, false, P9),
propiedad(10, "San Miguel",          1000,  850, null, 0, false, false, P10),


% 3) Casillas especiales
propiedad(1, "salida",      0, 0, null, 0, false, true, P1),
propiedad(11, "veAlaCarcel", 0, 0, null, 0, false, true, P11),
propiedad(12, "suerte",      0, 0, null, 0, false, true, P12),
propiedad(13, "carcel",      0, 0, null, 0, false, true, P13),
propiedad(14, "impuesto",    0, 200, null, 0, false, true, P14),
propiedad(15, "comunidad",   0, 0, null, 0, false, true, P15),

% 4) Cartas de suerte (10)
carta( 1, "suerte", 'Se ganó el Kino! +1000$', ganarDinero(1000),  C1),
carta( 2, "suerte", 'Le debes al SII -1000$',     ganarDinero(-1000), C2),
carta( 3, "suerte", 'Andate a la cárcel!',        irACarcel,          C3),
carta( 4, "suerte", 'Avance a La Moneda',         moverJugador(0),    C4),
carta( 5, "suerte", 'Retrocede 3 casillas',       moverJugador(-3),   C5),
carta( 6, "suerte", 'Recibe $2000',               ganarDinero(2000),  C6),
carta( 7, "suerte", 'Paga $500',                  ganarDinero(-500),  C7),
carta( 8, "suerte", 'Avanza a Providencia',     moverA(P8),         C8),
carta( 9, "suerte", 'Cobras intereses +1500$',  ganarDinero(1500),  C9),
carta(10, "suerte", 'Pierdes una vuelta',         pierdesTurno,       C10),

% 5) Cartas de comunidad (10)
carta(11, "comunidad", 'Pagar colegio $300',      ganarDinero(-300),  D1),
carta(12, "comunidad", 'Multiplica fortuna x2',   duplicarDinero,     D2),
```

Figura 4 script1_21781369_Martin_ArayaGaete.pl

➔ **Pegar tu script de escenario**
En la consulta "pruebas.pl".

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.

Mas información en Leeme.txt.

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.
    - Para una depuración más eficiente, usa “write/1” para ver hasta donde está funcionando.

Siguiendo estos pasos, no se debería tener problemas al inicializar el juego.

Resultados y autoevaluación

Requisito	Estado	Comentarios
RF01–RF08	**Completados**	Sin inconvenientes. Todas las funciones de creación y modificación de jugadores, propiedades y cartas se comportan según lo esperado.
RF09–RF11	**Completados**	Las validaciones de compra, renta y pago de impuestos funcionan correctamente, siempre que la propiedad exista en el tablero y el jugador tenga fondos suficientes.
RF12–RF17	**Completados**	Manejo de turnos, lanzamiento de dados, movimiento de jugadores y cálculo de renta cubiertos.
RF19	**Completados**	La cárcel y los estados de “en cárcel” se validan y actualizan correctamente.
RF20–RF21	**Parcial**	**- RF20 (Hipotecar/Deshipotecar)**: la hipoteca básica se implementó, pero al combinarla con bancarrota aún no subasta las propiedades como debería. **- RF21 (Subasta en bancarrota y casos extremos)**: el script de pruebas original falla en varias consultas (dominio inconsistente, propiedades con precio mayor al dinero disponible, variables no definidas). Se comentaron algunos casos para que el script no se quiebre, pero la lógica de subasta nunca se probó completamente.
RNF1–RNF9	**Completados**	Requisitos no funcionales (uso de SWI-Prolog 8.4 sin bibliotecas externas, estructura modular, límite de páginas) cumplidos.

Figura 5: “Autoevaluación”

Pruebas realizadas

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

Pruebas unitarias aisladas

- **Jugadores**: creación, modificación de dinero, estado de bancarrota, uso de cartaSalidacarcel, pasar turnos en cárcel.
- **Propiedades**: compra/venta, hipoteca/deshipoteca, cálculo de renta sin y con casas/hotel.
- **Cartas**:

- Se validó ganarDinero/1, irACarcel, moverJugador/1, cartaSalirCarcel, cambiarImpuesto/1, duplicarDinero para todos los casos definidos en RF.
- Fallan casos de cartas con acciones no implementadas

- **Tablero**: agregar y remover propiedades, wrap-around de casillas, verificar posiciones especiales (salida, cárcel, impuesto, suerte/comunidad).
- **Juego**:
  - juegoLanzarDados/4, juegoMoverJugador/4, juegoCalcularRentaJugador/3.
  - La lógica de turno (juegoJugarTurno/6) se probó por separado: si alguna sub-meta (como comprar propiedad) falla, el turno avanza sin colapsar.

Resultados

- **40 de 40 assert unitarias (100% éxito) tras las correcciones.**
- **12 de 15** escenarios de integración: **80% éxito**.
  - **Fallas detectadas**:
    1. Intento de ejecutar acciones de carta cuando el predicado no existe.
    2. Subasta de propiedades en bancarrota no implementada.
    3. Construir hotel sin cumplir con las 4 casas.
    4. Comprar propiedad hipotecada sin liberar hipoteca.

Funciones no completadas / Áreas pendientes

- **Subastar propiedades de un jugador en bancarrota (RF21)**:
  - Se implementó la lógica de hipoteca, pero al detectarse bancarrota las propiedades deberían pasarse a subasta automática. Actualmente solo se marcan como hipotecadas y no se distribuyen fondos a otros jugadores.
- **Cartas con acciones "complejas"**:
  - retrocede/1 y avanzaUtilidadMasCercana no actualizan todos los campos del estado de juego (especialmente cuando ya hay edificios en el tablero).

Queda una pequeña brecha (≈ 5%) en la cobertura funcional: principalmente relacionada con la lógica de subasta al caer en bancarrota y con algunas cartas de "comunidad/suerte" cuyas acciones no están definidas para todos los escenarios. Si bien la mayor parte del juego (movimientos, renta, compra/venta, cárcel) funciona al 100%, estos casos extremos requieren una revisión detallada antes de lograr un informe final sin observaciones.

Conclusiones del trabajo

Respecto a la anterior implementación de CAPITALIA en Scheme (paradigma funcional) y su desarrollo en Prolog (paradigma lógico), se observa que cada enfoque aporta beneficios y retos específicos.

En Scheme, la inmutabilidad y la transparencia referencial facilitaron el razonamiento sobre transiciones de estado y habilitaron pruebas unitarias sin efectos colaterales; sin embargo, el manejo de RNG y la actualización de listas exigieron versiones intermedias muy verbosas, lo que complicó la legibilidad y propició errores de paréntesis.

Por su parte, Prolog permitió describir reglas de juego (compras, alquileres, turnos) de modo declarativo, reduciendo el código necesario para recorrer estructuras y aprovechando el backtracking para generar escenarios distintos de dados sin bucles explícitos. No obstante, la depuración resultó más ardua: cada modificación del estado obliga a reconstruir términos completos, y el flujo implícito del backtracking demanda instrumentar el código con *write/1* para entender las ramas exploradas. Además, incorporar un generador de números por semilla en Prolog requirió pasar la semilla en cada predicado, encareciendo la sintaxis. En conclusión, si bien Prolog agiliza la especificación de reglas complejas, Scheme aporta mayor claridad en el manejo de estado y RNG reutilizable; una estrategia mixta—por ejemplo, encapsular RNG en un módulo separado dentro de Prolog—podría equilibrar legibilidad, rendimiento y poder inferencial en futuras versiones.

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