Autómatas, Teoría de Lenguajes y Compiladores

Proyecto Especial

Diseño e Implementación de un Lenguaje

Stage I

BoardSim - DSL para Diseño de Juegos de Mesa

ITBA

1. Equipo

Nombre	Apellido	Legajo	E-mail
Franco	Ferrari	63094	frferrari@itba.edu.ar
Nicolás Valentín	Arias	62272	narias@itba.edu.ar
Facundo	Lasserre	62165	flasserre@itba.edu.ar
Keoni Lucas	Dubovitsky	62815	kdubovitsky@itba.edu.ar

2. Repositorio

El desarrollo del trabajo estará versionado en el siguiente repositorio: https://github.com/francoferrari02/BoardSim-Compiler

3. Introducción

Antes de comenzar a desarrollar un lenguaje se debe estar en presencia de una necesidad arquitectural, para la cual el lenguaje y su compilador representan una solución adecuada.

La necesidad arquitectural debe involucrar un dominio del mundo real, el cual debe ser abstraído para dar lugar a un modelo computacional. Este modelo computacional es el que finalmente será manipulado programáticamente a través del lenguaje creado. Se espera que dicha manipulación programática del dominio abstraído permita escalar sistemáticamente la transformación concreta y real del mismo, es decir, que fomente un alto grado de automatización, de integración, de mejora continua, permitiendo a su vez la aplicación de tecnologías propias del desarrollo de software (i.e., versionado, testing, modularización, reutilización, IDEs, etc.).

En este proyecto, proponemos **BoardSim**, un DSL (Domain Specific Language) diseñado específicamente para la descripción, prototipado y simulación de juegos de mesa. Este lenguaje permite a diseñadores de juegos, desarrolladores y equipos de producción definir de manera declarativa e imperativa los elementos fundamentales de un juego (tableros, piezas, jugadores, reglas, eventos y mecánicas), y ejecutar simulaciones automatizadas para validar comportamientos, balancear reglas y generar análisis estadísticos. Inspirado en DSLs existentes como Ludax (un DSL acelerado por GPU para juegos de mesa con información perfecta, enfocado en simulación eficiente), Multigame (un DSL implícitamente paralelo para descripción de juegos de tablero) y Rulebook (un DSL que separa estado del juego y reglas para desenredar mecánicas), **BoardSim** abstrae complejidades como la generación de movimientos válidos, manejo de estados dinámicos y validación de interacciones, permitiendo al usuario concentrarse en la creatividad del diseño.

La necesidad surge en la industria de los juegos de mesa y digitales, donde el prototipado tradicional (mediante bocetos físicos, herramientas como Tabletop Simulator o código ad-hoc en lenguajes generales) resulta ineficiente para manejar complejidad creciente, iteraciones rápidas y testing a escala. Con **BoardSim**, un equipo puede especificar mecánicas de juegos clásicos como Ajedrez, Monopoly o TEG (Tácticas y Estrategia de Guerra), o innovar con variantes personalizadas (ej: juegos con elementos asimétricos), simular miles de partidas para detectar imbalances, y generar reportes visuales o estadísticos. Esto escalaría el desarrollo al permitir automatización, integración con IA para jugadores virtuales, y mejoras continuas en el compilador que benefician todos los proyectos sin alterar el código fuente.

4. Dominio

Un diseñador de juegos de mesa, un estudio indie o una empresa como Hasbro necesita prototipar, simular y refinar mecánicas de juegos (ej: Ajedrez con sus reglas de movimiento y jaque, Monopoly con economía y propiedades, o TEG con conquistas territoriales). Actualmente, estos procesos son manuales: se crean prototipos físicos, se documentan reglas en texto, o se implementan scripts personalizados en lenguajes como Python o Unity, lo que implica manejar detalles bajos como validación de bounds, colisiones o estados aleatorios.

Este enfoque manual es viable para conceptos básicos, pero enfrenta desafíos con el aumento de demanda: iteraciones frecuentes para balanceo, testing exhaustivo (ej: simular 10.000 partidas para calcular win rates), adaptación a variantes (ej: Ajedrez con piezas híbridas) y escalabilidad en equipos grandes. Surge una necesidad arquitectural: permitir que los diseñadores especifiquen el juego de forma programática, abstrayendo complejidades, y que esta especificación genere simulaciones automáticas, reportes y validaciones genéricas, minimizando riesgos y costos antes de producción física o digital.

Dado que el dominio implica escalamiento en el desarrollo (automatización de testing, modularización de reglas, integración con herramientas de software), la solución es programática y específica (mecánicas de juegos de mesa: tableros, piezas, interacciones, aleatoriedad), la construcción de un DSL como **BoardSim** resuelve la necesidad, similar a cómo DSLs como GDL (General Game Description Language) permiten describir y razonar sobre juegos generales.

El dominio abarca entidades clave: tableros (de formas variadas como grids, hexágonos o grafos), piezas (con atributos, movimientos y interacciones), jugadores (con recursos, posiciones y estrategias), eventos (dados, cartas, triggers), reglas (condiciones lógicas para victorias, capturas o penalizaciones), y simulación (ejecución de turnos, generación de estados y análisis).

1.1. Especificidad del Dominio

El dominio de los juegos de mesa es específico (**tipo II**): concentrado en un conjunto coherente de entidades (tableros, piezas, reglas, cartas) con interacciones fuertes y bien definidas (movimientos, eventos, victorias), conocidas en detalle desde juegos clásicos hasta modernos.

Esto se mapea a un DSL (**tipo IV** en lenguajes formales), no a un GPL, ya que BoardSim ofrece abstracciones de alto nivel para manipular directamente conceptos del dominio (ej: declarar un "tablero hexagonal" sin implementar matrices), ocultando detalles computacionales como algoritmos de pathfinding o randomización.

Usar un DSL fomenta escalabilidad: diseñadores iteran sin expertise en programación baja, y actualizaciones en el compilador (ej: soporte para IA avanzada) mejoran todos los juegos automáticamente.

5. Artefactos

Los artefactos de entrada son archivos de código fuente en BoardSim (extensión .bsim), que definen el universo del juego (tableros, piezas, reglas) y comandos de simulación.

Artefactos de salida:
- Logs detallados de simulación (texto/JSON) con estados turno-a-turno, movimientos, eventos y resultados finales.
- Reportes estadísticos (ej: win rates, duración promedio de partidas, balances de recursos).
- Visualizaciones opcionales (ej: exportar tableros a SVG o DOT para grafos).

El compilador procesa el .bsim para crear un ejecutable que simula el juego, aceptando parámetros como número de partidas o estrategias de jugadores, fomentando testing automatizado y reutilización.

6. Sistema de Tipos

6.1. Tipado

BoardSim emplea tipado estático y fuerte: tipos se infieren y verifican en compilación, previniendo errores runtime como asignaciones incompatibles. Esto asegura integridad en simulaciones complejas, donde mezclar tipos (ej: posición como string) podría invalidar mecánicas.

Tipos básicos: int (para scores, posiciones), string (nombres, descripciones), bool (condiciones lógicas), array (colecciones de piezas o propiedades).

6.2. Mapeo del Dominio

Tipos dominio-específicos mapean entidades reales:

- **Board:** Estructura para tableros (ej: *Board { string type; int size; array<Cell> cells; }*, donde Cell incluye propiedades como eventos o valores).
- **Piece:** Struct para piezas (ej: *Piece { string type; Position pos; array<Move> moves; map<string, int> attributes; }*, permitiendo atributos custom como "energía" o "ataque").
- **Player:** Struct para jugadores (ej: *Player { int id; int score; Position pos; array<Piece> pieces; map<string, int> resources; }*).
- **Event:** Tipo para eventos (ej: *Event { string trigger; func action; }*, *como cartas o dados*).
- **Rule:** Funciones para lógica (ej: *bool validMove(Piece p, Position to)*).

Esto abstrae el dominio: un tablero se representa como grafo o matriz, piezas como objetos con comportamientos, facilitando manipulación programática.

7. Construcciones

7.1. Estructuras de Control

- **If/ else/ elseif**: Para decisiones condicionales (ej: *if player.score > 0 then buy else bankrupt*).
- **For/while loops:** Para iteraciones (ej: *for turn in 1..maxTurns { rollDice; move; applyEvents; }*).
- **Switch/case:** Para ramificaciones multi-opción (ej: *switch event.type { case "capture": removePiece; }*).

7.2. I/O

- **Input:** Leer configuraciones o inputs runtime (ej: *read int numPlayers; read strategy from "file.json"*).
- **Output:** Imprimir estados (*print boardState*), logs (*log "Turn {turn}: {event}"*), exportar (*export graph to "board.dot"; export stats to "results.json"*).

7.3. Glosario

- **board:** Define tablero (ej: board Monopoly loop 40 cells { cell 0 "GO" event collect 200; }).
- **piece:** Define piezas (ej: piece Pawn moves [forward 1, capture diagonal] attributes { attack:1 }).
- **player:** Define jugadores (ej: player 1 resources { money:1500 } position 0 pieces [Pawn at row 2]).
- **event:** Define eventos (ej: event Card "Advance" action move +3).
- **dice:** Genera aleatoriedad (ej: dice 6 sides).
- **rule:** Define reglas (ej: rule victory if ownsAll or score >=1000).
- **simulate:** Ejecuta simulación (ej: simulate 50 turns strategy random { apply rules; check victory; }).
- **move:** Mueve entidades (ej: move piece to pos if valid).
- **apply:** Aplica eventos/reglas (ej: apply capture if adjacent).
- **if, for, switch, print, log, export, etc.:** Estructuras estándar con semántica dominio-específica.

8. Ejemplos

Ejemplo 1: Simulación de Monopoly con economía básica.

```
board Monopoly loop 40;

cell 0 "GO" event collect 200;
cell 1 "Property1" cost 60 rent 2;
// ... más cells

player 1 money 1500 position 0;
player 2 money 1500 position 0;

dice standard 6;
```

```
rule roll dice + dice;
rule buy if notOwned and money >= cost then own and money -= cost;
rule rent if ownedByOther then money -= rent;

simulate 20 turns {
    pos = (pos + roll) % 40;
    apply buy or rent or event;
    if money < 0 then end "Bankrupt";
}

log "Final scores: Player1 {player1.money}, Player2 {player2.money}";
print board;
```

Descripción: Define tablero circular, jugadores con dinero, reglas de compra/renta, y simula turnos con logs de scores finales.

Ejemplo 2: Simulación de TEG con conquistas.

```
board TEG graph;

node Argentina continent SouthAmerica armies 3 connected [Brazil, Chile];
// ... más nodes

player 1 owns [Argentina] armies 20;
player 2 owns [Brazil] armies 15;

dice attack 6;

rule reinforce add armies (dice * 2) to owned;
rule attack from source to target if connected {
    if source.armies > target.armies then conquer target armies
(source.armies - target.armies);
};

simulate 10 turns strategy aggressive {
    apply reinforce;
    attack random adjacent;
    if ownsContinent "SouthAmerica" then bonus 5;
}
```

```
export graph "teg.dot";
log "Final territories: Player1 {count player1.owns}";
```

Descripción: Modela mapa como grafo, jugadores con territorios/ejércitos, reglas de ataque/refuerzo, y simula conquistas con exportación visual.

9. Casos de Prueba

ID	Debe	Caso de Uso
A1.1	Aceptar	Declarar tablero con cells, formas y eventos básicos.
A1.2	Aceptar	Definir piezas con movimientos y atributos custom.
A1.3	Aceptar	Crear jugadores con resources y posiciones iniciales.
A1.4	Aceptar	Definir eventos como cartas con acciones.
A1.5	Aceptar	Usar dice en rules para aleatoriedad.
A1.6	Aceptar	Simular turns con loops y condicionales.
A1.7	Aceptar	Aplicar interacciones pieza-pieza (ej: capture).
A1.8	Aceptar	Generar outputs como logs y exportaciones.
A1.9	Aceptar	Mezclar tipos en expresiones (ej: position + dice).
A1.10	Aceptar	Ejecutar flows complejos con múltiples reglas.
R1.1	Rechazar	Sintaxis mal formada (ej: missing delimiter).
R1.2	Rechazar	Tipos incompatibles (ej: assign string to position).
R1.3	Rechazar	Variable no declarada en expresión.
R1.4	Rechazar	Duplicado de ID (ej: two boards same name).
R1.5	Rechazar	Movimiento inválido (ej: out of bounds in declaration).

Estos casos validan la robustez del DSL, cubriendo sintaxis, semántica y dominio-específico.