

Tabla de Contenido

Lenguaje Declarativo Minimalista para el Dilema del Prisionero	1
1. Objetivos	4
1.1 Objetivos Generales	4
1.2 Objetivos específicos	4
2. Descripción	4
3. Introducción	5
3.1 ¿Cuál es el dilema del prisionero?	5
3.2 Funciones del Lenguaje	6
4. Estructura del Lenguaje	6
5. Sistema de Tipos	7
5.1 Reglas de Tipado	7
6. Sección de Definición de Estrategias	8
6.1 Acciones Fundamentales	8
6.2 Elementos Clave	8
6.3 Sintaxis de Estrategia	9
6.4 Operadores	9
6.5 Funciones del Sistema	10
6.6 Estados del Sistema	12
7. Sección de Definición de Partidas	13
7.1 Sintaxis de Partida	13
7.2 Reglas de Puntuación	14
8. Sección de Punto de Entrada	15
8.1 Conceptos Clave	15
8.2 Sintaxis del Punto de Entrada	15
8.3 Validación de Parámetros	15
9. Consideraciones para el Compilador	16
9.1 Análisis Léxico (Lexer)	16
9.2 Manejo de Errores:	16
9.3 Proceso de Ejecución	17
10. Formato de Salida	18
11. Reportes	19
11.2 Tabla de tokens	19
11.3 Reporte de error	19
12. Entorno de Trabajo	20
12.1 Editor	20
12.2 Funcionalidades	20
12.3 Herramientas	21
12.4 Reportes	21

13. Anexo	22
13. 1 Implementación de random	22
13.2 Interfaz de contrato	22
13.3 Implementación concreta	23
13.4 Ejemplo de uso	24
14. Requerimientos Mínimos	25
15. Entregables	25
16. Restricciones	26
17. Fecha de Entrega	26

1. Objetivos

1.1 Objetivos Generales

Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la construcción de una solución de software.

1.2 Objetivos específicos

- Que el estudiante aprenda a generar analizadores léxicos y sintácticos utilizando las herramientas de JFLEX y CUP.
- Que el estudiante aprenda los conceptos de token, lexema, patrones y expresiones regulares.
- Que el estudiante pueda realizar correctamente el manejo de errores léxicos.
- Que el estudiante sea capaz de realizar acciones gramaticales utilizando el lenguaje de programación JAVA.

2. Descripción

El curso de Organización de Lenguajes y Compiladores 1 desafía a los estudiantes a construir un sistema que capture la esencia de la evolución de estrategias computacionales, inspirado directamente en los revolucionarios experimentos de Robert Axelrod.

Comprenderán y aplicarán los torneos de estrategias que transforman nuestra comprensión de la cooperación. Deberán diseñar un lenguaje declarativo que permita:

- Modelar estrategias complejas de interacción
- Simular múltiples rondas de decisiones estratégicas
- Analizar cómo emergen comportamientos cooperativos

El objetivo es que cada estudiante recree el proceso de Axelrod: desarrollar un sistema donde estrategias compiten, evolucionan y revelan principios fundamentales de interacción. No se trata solo de programar, sino de experimentar con modelos computacionales que reflejan comportamientos sociales y estratégicos complejos.

3. Introducción

3.1 ¿Cuál es el dilema del prisionero?

El Dilema del Prisionero es un concepto clásico de la teoría de juegos. Se plantea de la siguiente manera:

Dos personas son arrestadas bajo la sospecha de haber cometido un delito grave. Sin embargo, la policía no tiene pruebas suficientes para condenarlos por ese delito, así que los separan y les hacen la misma oferta

- 1. Si uno confiesa y el otro guarda silencio, el que confiesa será liberado, mientras que el otro recibirá una sentencia máxima.*
- 2. Si ambos confiesan, ambos recibirán una sentencia intermedia.*
- 3. Si ninguno confiesa, ambos recibirán una sentencia menor por un delito menor.*

Se supone que ambos prisioneros son completamente egoístas y su única meta es reducir su propia estancia en la cárcel. Los prisioneros tienen dos opciones: cooperar con su cómplice y permanecer callado, o delatar y traicionar a su cómplice y confesar. El resultado de cada elección depende de la elección del cómplice. Por desgracia, **uno no conoce qué ha elegido hacer el otro hasta después de elegir qué hará y se dicte el resultado.**










A \ B	B		
		B COOPERA	B DELATA
A	COOPERA	 A COOPERA  R  R	 S  T
		 A DELATA T  S	 P  P

Figura 1. Posibles resultados del dilema

Los experimentos de Axelrod expandieron significativamente nuestra comprensión de este dilema. Mediante torneos computacionales donde diferentes estrategias competían entre sí, Axelrod demostró que la cooperación puede emerger incluso en sistemas aparentemente egoístas.

3.2 Funciones del Lenguaje

Este lenguaje está diseñado para permitir a los usuarios **definir** y **ejecutar** estrategias en el contexto del dilema del prisionero. Es:

- **Declarativo:** Se enfoca en *qué* hacer mediante reglas y condiciones, no en *cómo* hacerlo paso a paso.
- **Minimalista:** Incluye únicamente los elementos esenciales para su funcionamiento.

En él, se definen **estrategias**, **partidas** y un **punto de entrada** que orquesta la ejecución.

Nota: la extensión de los archivos de entrada será **“.cmp”**

4. Estructura del Lenguaje

El lenguaje consta de **tres secciones** principales:

1. **Definición de Estrategias**
2. **Definición de Partidas**
3. **Punto de Entrada**

4.1 Comentarios

4.1.1. Comentarios de una línea

Este tipo de comentario comenzará con `//` y deberá terminar con un salto de línea

4.1.2. Comentarios multilínea

Este tipo de comentario comienza con `/*` y deberá de terminar con `*/`.

```
// Esto es un comentario de una sola línea.  
/* Esto es un comentario  
multilínea */
```

5. Sistema de Tipos

El lenguaje implementa un sistema de tipos estricto con los siguientes tipos fundamentales:

1. Acción:

- Valores: **C** (**Cooperar**) o **D** (**Delatar**)
- Usado en decisiones de estrategia

2. Entero:

- Números enteros sin decimales
- Usado en contadores y cálculos

3. Flotante:

- Números con decimales
- Usado en proporciones y cálculos de **random**

4. Booleano:

- Valores: **true** o **false**
- Usado en condiciones

5. Lista:-

- Secuencias de acciones
- Usado en historiales

5.1 Reglas de Tipado

1. Comparaciones numéricas:

- a. **random** solo puede compararse con flotantes entre 0.0 y 1.0
- b. Contadores (**get_moves_count**) solo pueden compararse con enteros
- c. **round_number** solo puede compararse con enteros

2. Comparaciones de secuencia:

- a. **get_last_n_moves** falla si n es mayor al historial disponible

3. Comparaciones permitidas:

- a. Entre enteros: Todos los operadores de comparación son válidos para comparar resultados de **get_moves_count** entre sí
- b. **random**: Puede combinarse con otros operadores booleanos en la misma condición.

6. Sección de Definición de Estrategias

6.1 Acciones Fundamentales

En el contexto del Dilema del Prisionero, existen dos acciones posibles:

- **Cooperar (C)**: Representa la decisión de mantener silencio y no delatar al otro prisionero. En términos del juego, es la acción que beneficia al colectivo pero puede resultar en una mayor penalización individual si el otro jugador no coopera.
- **Delatar (D)**: Representa la decisión de traicionar al otro prisionero. En términos del juego, es la acción que puede beneficiar individualmente al jugador a costa del otro, pero si ambos delatan, ambos reciben una penalización mayor que si hubieran cooperado.

6.2 Elementos Clave

Los elementos fundamentales que componen una estrategia son:

- **Nombre**: Identificador único para la estrategia
- **Acción Inicial**: Primera acción a tomar (C o D). Esta acción se ejecuta automáticamente en la primera ronda sin evaluar reglas
- **Reglas**: Conjunto de condiciones y acciones a tomar donde SOLO se permiten operaciones booleanas y comparaciones.

Los operadores permitidos en las condiciones son:

- Comparación: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Booleanos: `&&` (AND), `||` (OR), `!` (NOT)

Composición de Funciones:

- Se permite el encadenamiento de funciones del sistema
- La evaluación se realiza de dentro hacia afuera

Precedencia de Funciones:

- Las funciones anidadas se evalúan siempre desde la más interna hacia la más externa
- Por ejemplo: `get_moves_count(get_last_n_moves(history, 2), D)`
 1. Primero se evalúa `get_last_n_moves(history, 2)`
 2. Luego se evalúa `get_moves_count` sobre el resultado

6.3 Sintaxis de Estrategia

```
strategy <nombre estrategia> {  
  initial: <acción inicial>  
  rules: [  
    if <condición booleana 1> then <acción 1>,  
    if <condición booleana 2> then <acción 2>,  
    ...,  
    if <condición booleana n> then <acción>,  
    else <acción por defecto>  
  ]  
}
```

Nota: Las reglas se evalúan en cascada desde la primera hasta encontrar una condición verdadera.

Nota: Si ninguna condición es verdadera, se ejecuta la acción por defecto definida en el else. Sólo está permitido al final de la cascada de reglas.

Comportamiento en Tiempo de Ejecución:

- Las estrategias se evalúan en cada ronda usando solo la información de rondas anteriores
- Las evaluaciones de las reglas son síncronas: primero se evalúan todas las reglas de ambas estrategias y luego se ejecutan las acciones resultantes
- Si una regla genera un error (por ejemplo, al intentar acceder a más historia de la disponible), el programa falla inmediatamente

6.4 Operadores

- Comparación: ==, !=, >, <, >=, <=
- Booleanos: && (AND), || (OR), ! (NOT)

La precedencia sigue el orden estándar: primero !, luego comparaciones, luego &&/||.

6.5 Funciones del Sistema

Las funciones están diseñadas específicamente para replicar la información disponible en el experimento original de Axelrod:

1. `get_move(history, n)`

- Tipo de Retorno: **Action**
- Descripción: Obtiene la acción en la posición `n` del historial (0-based)
- Error: Si `n` es mayor o igual al tamaño del historial
- Ejemplo:

```
get_move(opponent_history, 0) == C
```

2. `last_move(history)`

- Tipo de Retorno: **Action**
- Descripción: Obtiene la última acción del historial especificado
- Ejemplo:

```
last_move(opponent_history) == C  
last_move(self_history) == D
```

3. `get_moves_count(history, action)`

- Tipo de Retorno: **Integer**
- Descripción: Cuenta cuántas veces aparece una acción específica en el historial.
- Ejemplo:

```
get_moves_count(opponent_history, C) == 3  
get_moves_count(self_history, D) == 2
```

4. `get_last_n_moves(history, n)`

- Tipo de Retorno: `List<Action>`
- Descripción: Obtiene las últimas n jugadas del historial
- Restricciones:
 - n debe ser mayor que 0
 - Error: Si n es mayor que el tamaño del historial
- Ejemplo

```
get_last_n_moves(opponent_history, 3) == [C, D, C]
```

IMPORTANTE: Todas las funciones y contadores del sistema son 0-based. Es decir:

- **round_number** comienza en 0
- **get_move** usa índices desde 0
- **round_number == 0** corresponde a la ronda inicial donde se usa initial

6.6 Estados del Sistema

Estados diseñados para replicar la información disponible en el experimento original:

1. **round_number**
 - a. Tipo: `Integer`
 - b. Descripción: Número de la ronda actual (comienza en 0)
 - c. Uso: `round_number == 1`
2. **opponent_history**
 - a. Tipo: `List<Action>`
 - b. Descripción: Historial completo de movimientos del oponente
3. **self_history**
 - a. Tipo: `List<Action>`
 - b. Descripción: Historial completo de movimientos propios
4. **total_rounds**
 - a. Tipo: `Integer`
 - b. Descripción: Número total de rondas en la partida
 - c. Uso: `round_number == total_rounds`
5. **random**
 - a. Tipo: `Float [0.0-1.0]`
 - b. Descripción: Genera un número aleatorio entre 0 y 1
 - c. Uso: `random < 0.1`
 - d. Nota: el generador de números aleatorios debe reiniciarse en cada partida. Para efectos del proyecto se adjunta un generador determinista.

7. Sección de Definición de Partidas

7.1 Sintaxis de Partida

En este lenguaje, las partidas son estrictamente uno contra uno, limitándose a dos estrategias por partida. La estructura de una partida define los participantes, la duración y el sistema de puntuación.

```
match <nombre partida> {  
  players strategies: [<estrategia 1>, <estrategia 2>]  
  rounds: <número de rondas>  
  scoring: {  
    mutual cooperation: <puntos>,  
    mutual defection: <puntos>,  
    betrayal reward: <puntos>,  
    betrayal punishment: <puntos>  
  }  
}
```

7.2 Reglas de Puntuación

El sistema de puntuación debe cumplir con ciertas condiciones matemáticas que reflejan la naturaleza del dilema del prisionero (cada prisionero es egoísta y busca su menor condena):

- **T > R > P > S** donde:
 - T (Tentación): **betrayal reward** - La recompensa por traicionar cuando el otro coopera
 - R (Recompensa): **mutual cooperation** - El beneficio por cooperación mutua
 - P (Castigo): **mutual defection** - La penalización por defección mutua
 - S (Tonto): **betrayal punishment** - El castigo por ser traicionado

Además:

- Las puntuaciones deben ser no negativas
- Cada tipo de puntuación debe definirse exactamente una vez

8. Sección de Punto de Entrada

8.1 Conceptos Clave

El punto de entrada es crucial para la ejecución del programa, definiendo:

- **Partidas a ejecutar:** Una lista ordenada de las partidas que se jugarán.
- **Semilla aleatoria:** Un valor numérico que garantiza la reproducibilidad de los resultados (seed).

8.2 Sintaxis del Punto de Entrada

```
main {  
  run [<nombre partida 1>, <nombre _parte_2>, ...] with {  
    seed: <entero>  
  }  
  
  run [<nombre _parte_3>] with {  
    seed: <entero diferente>  
  }  
}
```

8.3 Validación de Parámetros

Los parámetros del punto de entrada requieren validación específica:

- **seed:**
 - Debe ser un entero positivo
 - Garantiza que los resultados sean reproducibles

9. Consideraciones para el Compilador

9.1 Análisis Léxico (Lexer)

El analizador léxico debe identificar y categorizar los siguientes tipos de tokens:

1. Palabras Reservadas:

- Elementos estructurales: `strategy`, `match`, `main`
- Secciones: `initial`, `rules`
- Control de flujo: `if`, `then`, `else`
- Configuración: `scoring`, `run`, `with`, `seed`

2. Identificadores:

- Nombres de estrategias definidas por el usuario
- Referencias a estados del sistema

3. Operadores:

- Operaciones lógicas: `&&`, `||`, `!`
- Comparadores: `==`, `!=`, `>`, `<`, `>=`, `<=`

4. Literales:

- Acciones del juego: `C`, `D`
- Valores numéricos: enteros y números de punto flotante
- Valores booleanos: `true`, `false`

9.2 Manejo de Errores:

1. Cualquier error durante la ejecución (índices inválidos, historiales insuficientes, tipos incorrectos, etc.) debe resultar en la terminación inmediata de todo el programa
2. No se permiten recuperaciones parciales o manejo de errores a nivel de estrategia o partida
3. El programa debe terminar completamente ante el primer error encontrado.

9.3 Proceso de Ejecución

La ejecución del programa sigue estas fases secuenciales:

1. Fase de Inicialización:

- Establecer todos los estados internos a sus valores iniciales
- Inicializar los historiales como listas vacías
- Configurar una instancia independiente del generador de números aleatorios para cada estrategia en cada partida usando la semilla proporcionada, garantizando la reproducibilidad

2. Fase de Ejecución de Rondas:

- Evaluar la sección de cálculos de cada estrategia
- Aplicar las reglas para determinar las acciones
- Actualizar los estados internos y los historiales
- Calcular las puntuaciones de la ronda

3. Fase de Finalización:

- Realizar el cálculo final de puntuaciones
- Generar la salida según el modo seleccionado

10. Formato de Salida

El sistema proporciona un formato de salida estándar, diseñado para visualizar claramente el desarrollo de la partida.

Este modo proporciona una vista concisa del desarrollo de la partida. Es ideal para visualizar rápidamente el comportamiento de las estrategias en juego.

```
=== Partida===
Configuración:
  Rondas: 100
  Estrategias: TitForTat vs AdaptiveStrategy
Scoring:
  - Cooperación mutua: 3
  - Defección mutua: 1
  - Traición: 5/1 (traidor/traicionado)

Desarrollo:
Ronda 1: TitForTat=C, AdaptiveStrategy=C (3-3)
Ronda 2: TitForTat=C, AdaptiveStrategy=D (1-5)
Ronda 3: TitForTat=D, AdaptiveStrategy=C (5-1)
...

Resumen:
  TitForTat:
    - Puntuación final: 250
    - Cooperaciones: 67%
    - Defecciones: 33%

  Adaptive Strategy:
    - Puntuación final: 280
    - Cooperaciones: 58%
    - Defecciones: 42%
```

El resultado de la evaluación de los elementos deberá mostrarse en la consola, indicando si fue exitoso o si falló en algún elemento, especificando cuál fue el elemento que falló.

11. Reportes

Los reportes son parte fundamental del sistema, ya que muestra de forma visual las herramientas utilizadas para realizar la ejecución del código. Cada reporte debe ser generado luego de cada ejecución por lo que no deberá mostrarse reportes de análisis anteriores.

A continuación, se muestran ejemplos de estos reportes. Queda a discreción del estudiante el diseño de estos, solo se pide que sean totalmente legibles.

11.2 Tabla de tokens

El reporte de tokens debe tener la información suficiente para poder identificar los tokens reconocidos en el análisis léxico. Este debe ser en una tabla.

#	Lexema	Tipo	Línea	Columna
1	arr	id	5	3
2	3	Double	6	3
3	“Hola”	String	8	3

11.3 Reporte de error

El reporte de error debe contener la información suficiente para detectar y corregir errores en el código fuente.

#	Tipo	Descripción	Línea	Columna
1	Léxico	El carácter “\$” no pertenece al lenguaje	5	3

12. Entorno de Trabajo

12.1 Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades y herramientas que serán de utilidad para el usuario. La función principal del editor será el ingreso del código fuente que será analizado. Queda a discreción del estudiante el diseño.

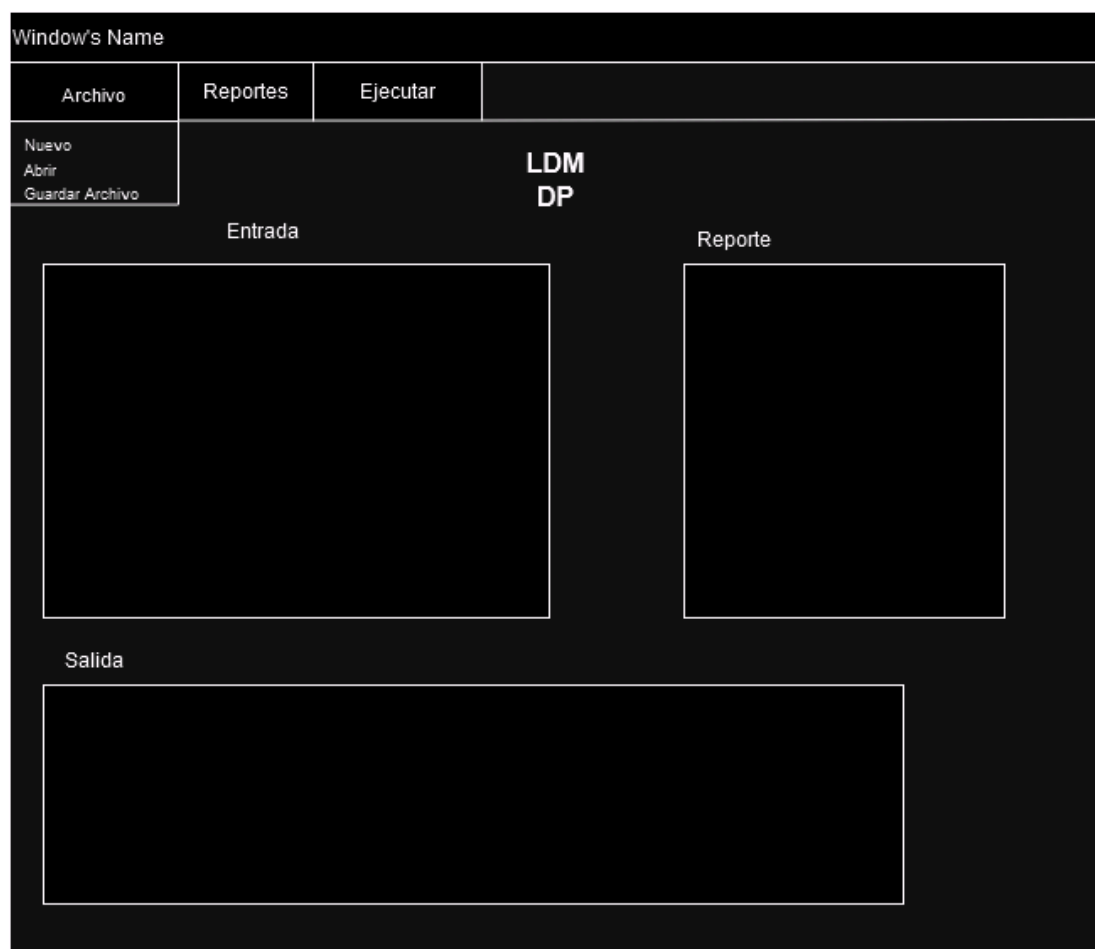


Figura 2. Interfaz Propuesta

12.2 Funcionalidades

- **Nuevo Archivo:** se podrá crear un nuevo archivo con extensión.
- **Abrir:** Abre un explorador de archivos y selecciona uno para seleccionar su información.
- **Guardar:** nos permitirá guardar el archivo actual.

12.3 Herramientas

- **Ejecutar:** se envía la entrada actual al intérprete con la finalidad de realizar el análisis léxico, sintáctico y la ejecución de instrucciones.

12.4 Reportes

- **Reporte de Tokens:** se mostrarán todos los tokens reconocidos en el analizador léxico.
- **Reporte de Error:** se mostrará el error que es y si es tipo léxico.

13. Anexo

13. 1 Implementación de random

Para garantizar que todos tengan la misma implementación del generador de números pseudoaleatorios, se adjunta una implementación diseñada para que puedan implementarla en su proyecto, asegurando el funcionamiento correcto de la idempotencia de las salidas.

13.2 Interfaz de contrato

```
/**
 * Interfaz que define el contrato para generadores de números aleatorios.
 * Permite a los estudiantes implementar diferentes estrategias si lo desean.
 */
interface RandomGenerator {
    /**
     * Genera un número decimal aleatorio entre 0.0 y 1.0.
     * @return Número decimal aleatorio
     */
    double nextDouble();
}
```

13.3 Implementación concreta

```
/**
 * Implementación concreta de un generador de números aleatorios
 * determinista.
 * Diseñado para ser simple y predecible, ideal para entornos educativos.
 */
class DeterministicRandomGenerator implements RandomGenerator {
    // Semilla original
    private final long originalSeed;

    // Generador de números aleatorios
    private Random random;

    /**
     * Constructor privado para control estricto de creación.
     * @param seed Semilla para el generador
     */
    private DeterministicRandomGenerator(long seed) {
        this.originalSeed = seed;
        this.random = new Random(seed);
    }

    /**
     * Método de fábrica para crear generadores.
     * @param seed Semilla para el generador
     * @return Nuevo generador de números aleatorios
     */
    public static RandomGenerator create(long seed) {
        return new DeterministicRandomGenerator(seed);
    }

    @Override
    public double nextDouble() {
        return random.nextDouble();
    }
}
```

13.4 Ejemplo de uso

```
public class RandomTest {

    /**
     * Ejemplo de uso de múltiples generadores independientes.
     * @param args Argumentos de línea de comandos (no utilizados)
     */
    public static void main(String[] args) {
        RandomGenerator gen1 = DeterministicRandomGenerator.create(42);
        RandomGenerator gen2 = DeterministicRandomGenerator.create(69);

        printGenerator("1", gen1);
        printGenerator("1", gen1);
        printGenerator("2", gen2);
        printGenerator("1", gen1);
        printGenerator("2", gen2);
        printGenerator("1", gen1);
        printGenerator("2", gen2);
        printGenerator("2", gen2);

    }

    /**
     * Imprimir números generados.
     * @param id Identificador del generador
     * @param gen Generador de números pseudo-aleatorios
     */
    private static void printGenerator(String id, RandomGenerator gen){
        System.out.println("Generador: " + id + ", a lanzado el número: " +
gen.nextDouble());
    }
}
```