Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computadores
Compiladores e intérpretes (CE-1108)

Implementación de un interprete con ANTLRv4:
Documentación principal

Profesor:
Ing. Marco A. Hernández Vásquez

Estudiantes:
Adriel Sebastián Chaves Salazar (2021031465)
José Manuel Loría Cordero (2022211671)
Emmanuel Esquivel Chavarría (2022312336)
Daniel Duarte Cordero (2022012866)

I Semestre 2025

Tabla de Contenidos

1. Introducción

En el ámbito de la informática, los compiladores e intérpretes son herramientas fundamentales que permiten transformar código fuente escrito por humanos en instrucciones ejecutables por las máquinas [1]. Mientras que los compiladores traducen el código fuente completo a código máquina para su posterior ejecución, los intérpretes ejecutan directamente el código fuente, interpretándolo línea por línea [2]. Esta distinción es crucial para entender los diferentes enfoques en el diseño y la implementación de lenguajes de programación.

El presente trabajo se centra en el desarrollo de un intérprete utilizando la herramienta ANTLR v4 (ANother Tool for Language Recognition) [2], un generador de analizadores léxicos y sintácticos ampliamente utilizado en la industria. A través de este proyecto, se exploran los conceptos fundamentales de la teoría de compiladores, incluyendo el análisis léxico, sintáctico y semántico, así como la implementación de una tabla de símbolos para la gestión de variables [3].

La implementación sigue un enfoque incremental, comenzando con una estructura básica capaz de reconocer tokens y construir árboles de sintaxis, hasta llegar a un intérprete funcional que puede ejecutar expresiones aritméticas, declaraciones de variables y estructuras de control de flujo. Para optimizar el diseño y facilitar la extensibilidad del intérprete, se implementa el patrón de diseño Interpreter [4], que permite representar la gramática del lenguaje como un conjunto jerárquico de clases.

Este documento presenta, en primer lugar, el estado del arte de los conceptos y tecnologías utilizadas en el desarrollo de compiladores e intérpretes. Posteriormente, se profundiza en la investigación sobre el análisis semántico y la tabla de símbolos, componentes esenciales para la funcionalidad del intérprete. Se detalla la implementación básica realizada, seguida de las mejoras implementadas para extender sus capacidades. Finalmente, se presenta una conclusión que sintetiza los aprendizajes obtenidos y se discuten posibles líneas de trabajo futuro, como la implementación de ámbitos de variables y optimizaciones de rendimiento [5].

La relevancia de este trabajo radica no solo en la comprensión de los fundamentos teóricos de los lenguajes de programación, sino también en la aplicación práctica de estos conocimientos mediante el uso de herramientas modernas como ANTLR v4, contribuyendo así a la formación integral en el campo de la informática y las ciencias de la computación.

2. Estado del Arte de Compiladores e Intérpretes

2.1. Fundamentos de Compiladores e Intérpretes

Los compiladores e intérpretes son herramientas fundamentales en el ámbito de la programación que permiten la ejecución de código escrito en lenguajes de alto nivel. Ambos cumplen funciones similares pero con enfoques diferentes que determinan sus características y aplicaciones [6].

Un compilador es un programa que toma como entrada código fuente escrito en un lenguaje de programación de alto nivel y genera como resultado un programa equivalente en código máquina [7]. Este proceso de traducción se realiza una sola vez, y el programa resultante puede ser ejecutado múltiples veces sin necesidad de volver a compilar, lo que ofrece mayor eficiencia en tiempo de ejecución.

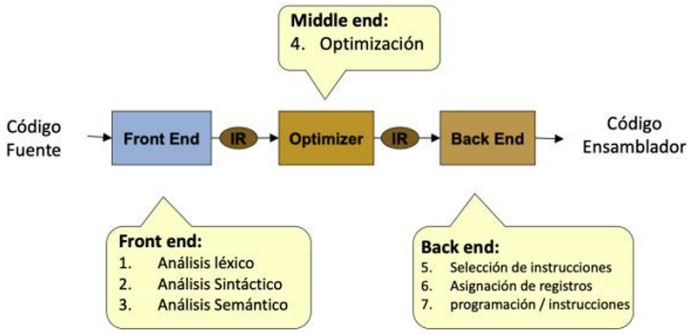

Figura 1: Estructura general de un compilador

Por otro lado, un intérprete ejecuta directamente el código fuente escrito en un lenguaje de alto nivel, procesando cada instrucción en tiempo real y generando los resultados correspondientes [**?**]. Esta ejecución línea por línea ofrece mayor flexibilidad y facilita la depuración, aunque puede resultar menos eficiente en términos de rendimiento.

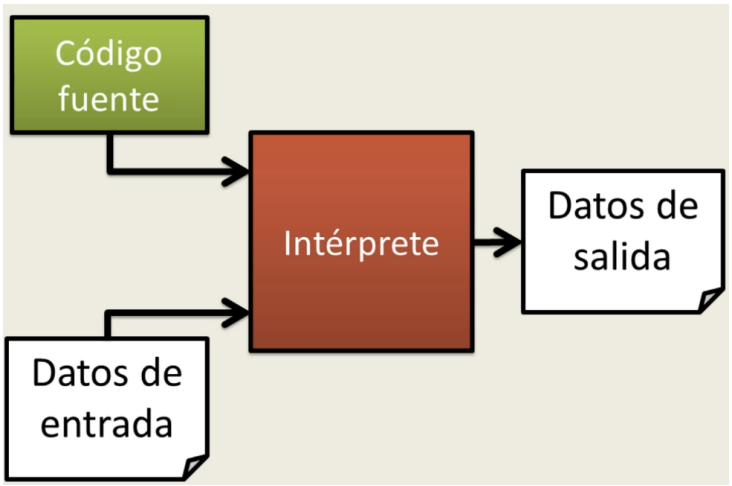

Figura 2: Estructura general de un intérprete

2.2. Etapas del Proceso de Compilación e Interpretación

Tanto los compiladores como los intérpretes siguen una serie de etapas bien definidas para procesar el código fuente, aunque con diferencias en su implementación y momento de ejecución [8].

En el caso de los compiladores, el proceso se divide tradicionalmente en tres grandes fases: Front-end, Middle-end y Back-end [6]. El Front-end se encarga del análisis del código fuente, el Middle-end realiza optimizaciones independientes de la plataforma, y el Back-end genera el código máquina específico para la arquitectura objetivo.

Los intérpretes, por su parte, realizan principalmente las etapas del Front-end, pero en lugar de generar código máquina, ejecutan directamente las instrucciones representadas en el árbol de sintaxis abstracta (AST) [2].

2.3. Análisis Léxico

El análisis léxico, también conocido como escaneo o tokenización, es la primera etapa tanto en compiladores como en intérpretes [7]. Esta fase se encarga de leer el código fuente como una secuencia de caracteres y agruparlos en unidades léxicas significativas llamadas tokens.

El componente encargado de realizar esta tarea se denomina lexer o ana-

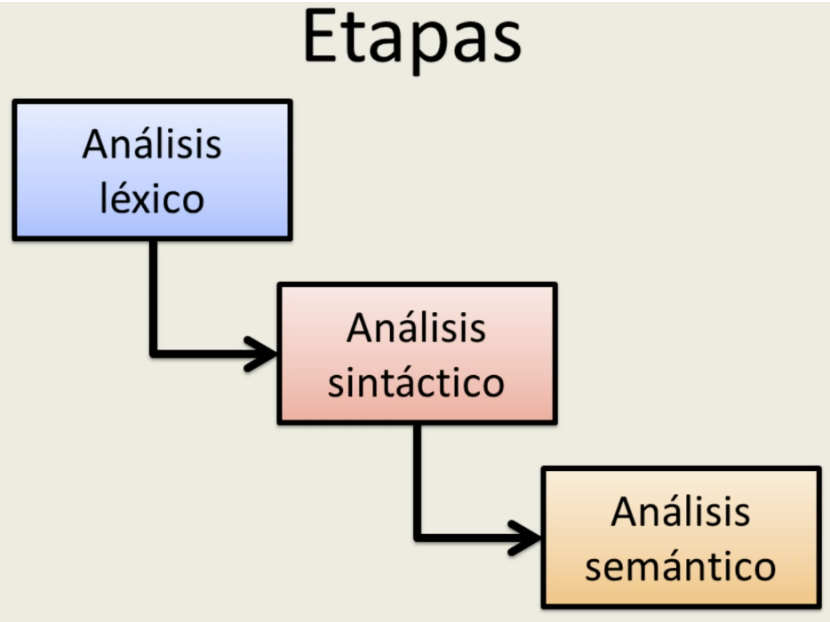

Figura 3: Etapas principales del proceso de interpretación

lizador léxico. Este componente utiliza una gramática regular para definir los patrones que identifican cada tipo de token en el lenguaje [9]. Los tokens pueden representar palabras clave, identificadores, operadores, constantes numéricas, cadenas de texto, entre otros elementos del lenguaje.

El resultado del análisis léxico es un flujo de tokens que preserva el orden en que aparecen en el código fuente original, pero abstrae detalles como espacios en blanco, comentarios y otros elementos no relevantes para la sintaxis del lenguaje [6].

2.4. Análisis Sintáctico

El análisis sintáctico, también conocido como parsing, es la segunda etapa del proceso y se encarga de determinar si la secuencia de tokens generada en la etapa anterior cumple con las reglas gramaticales del lenguaje [8].

El componente encargado de esta tarea es el parser o analizador sintáctico, que utiliza una gramática libre de contexto para definir las reglas de formación de expresiones, declaraciones, bloques y otras estructuras del lenguaje [7].

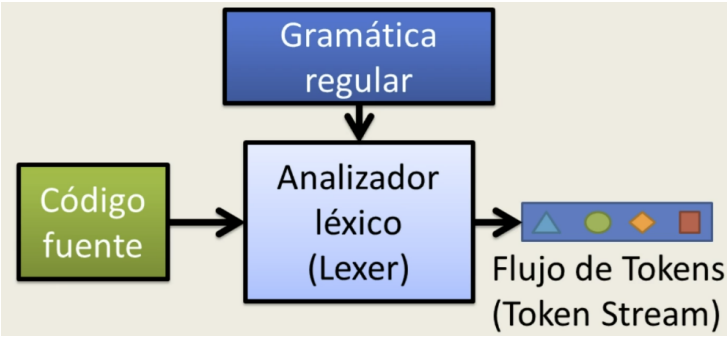

Figura 4: Proceso de análisis léxico y generación de tokens

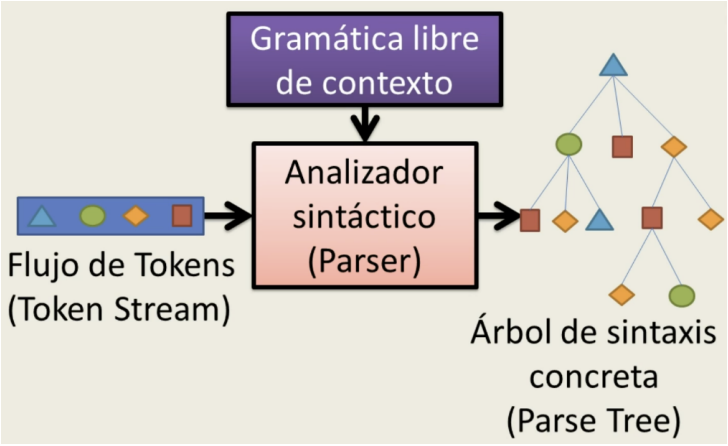

Figura 5: Construcción del árbol de sintaxis concreta durante el análisis sintáctico

El resultado del análisis sintáctico es un árbol de sintaxis concreta (Parse Tree), que representa la estructura jerárquica del programa de acuerdo con las reglas gramaticales del lenguaje [2]. Este árbol muestra cómo cada componente del programa (expresiones, declaraciones, bloques, etc.) se relaciona con los demás siguiendo las reglas de la gramática.

2.5. Análisis Semántico

El análisis semántico constituye la tercera etapa del proceso y se encarga de determinar el significado de las construcciones sintácticamente correctas del programa [6].

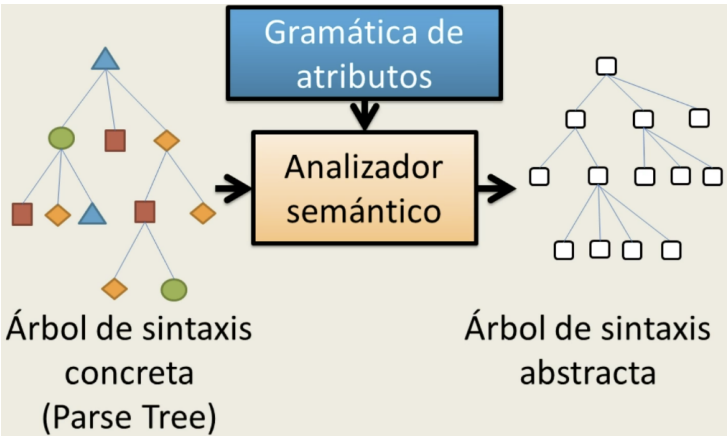

Figura 6: Proceso de análisis semántico y generación del árbol de sintaxis abstracta

Durante esta etapa, se verifica que las operaciones se realicen con tipos compatibles, que las variables estén declaradas antes de su uso, que los identificadores se utilicen de manera consistente con su declaración, entre otras comprobaciones semánticas [9].

Un componente fundamental del análisis semántico es la tabla de símbolos, una estructura de datos que almacena información sobre los identificadores (variables, funciones, tipos, etc.) declarados en el programa, incluyendo su tipo, ámbito, y otros atributos relevantes [8].

El resultado del análisis semántico es un árbol de sintaxis abstracta (AST), que representa la estructura esencial del programa, eliminando detalles sintácticos innecesarios y preparando la información para las siguientes etapas del proceso [7].

2.6. ANTLR como Herramienta para la Construcción de Compiladores e Intérpretes

ANTLR (ANother Tool for Language Recognition) es una herramienta de generación de analizadores léxicos y sintácticos ampliamente utilizada en la construcción de compiladores e intérpretes [2]. Desarrollada por Terence Parr, ANTLR permite definir gramáticas en un formato similar a EBNF (Extended Backus-Naur Form) y genera automáticamente el código necesario para implementar los analizadores léxico y sintáctico correspondientes.

Las principales ventajas de ANTLR incluyen:

- Soporte para múltiples lenguajes de programación como Java, C#, Python, JavaScript, entre otros.
- Generación automática de visitantes y listeners para recorrer el árbol sintáctico.
- Manejo integrado de errores léxicos y sintácticos.
- Soporte para gramáticas LL(*) con predicados semánticos que aumentan su poder expresivo.
- Herramientas visuales para depurar gramáticas y visualizar árboles sintácticos.

ANTLR v4, la versión más reciente, incorpora mejoras significativas en el algoritmo de parsing, permitiendo una mayor expresividad en las gramáticas y simplificando el desarrollo de lenguajes de programación [2].

2.7. Patrones de Diseño en la Implementación de Intérpretes

En el desarrollo de intérpretes, ciertos patrones de diseño resultan particularmente útiles para estructurar el código y facilitar su mantenimiento y extensión [4]. Uno de los más relevantes es el patrón Interpreter, que permite representar la gramática de un lenguaje como una jerarquía de clases correspondientes a las reglas gramaticales.

El patrón Interpreter se basa en la definición de una interfaz o clase abstracta para representar los nodos del árbol de sintaxis abstracta, con métodos para ejecutar o evaluar cada tipo de construcción del lenguaje [4]. Cada regla gramatical se implementa como una clase concreta que implementa esta interfaz, con la lógica necesaria para ejecutar o evaluar la construcción correspondiente.

Este enfoque facilita la extensión del lenguaje con nuevas construcciones, ya que basta con añadir nuevas clases a la jerarquía existente sin modificar las ya implementadas [7]. Además, permite separar claramente la estructura sintáctica del lenguaje de su comportamiento semántico, lo que mejora la modularidad y mantenibilidad del código.

Otros patrones de diseño comúnmente utilizados en la implementación de intérpretes incluyen el Visitor para recorrer el árbol de sintaxis abstracta, el Composite para representar expresiones compuestas, y el Factory Method para crear instancias de los diferentes tipos de nodos del árbol [4].

3. Investigación sobre Análisis Semántico y Tabla de Símbolos

3.1. Análisis Semántico

El análisis semántico es una fase importante del proceso de compilación, el cual sigue al proceso de análisis sintáctico. Su principal objetivo es verificar la validez semántica de las sentencias del programa aceptadas por el analizador sintáctico. En pocas palabras, garantiza que el programa tenga sentido y obedezca las reglas de significado del lenguaje, a pesar de ser sintácticamente correcto [10].

Como afirma Alfonseca, esta fase del compilador verifica la corrección semántica del programa. El proceso implica añadir diversas anotaciones del árbol de análisis sintáctico generado en la fase anterior. Estas anotaciones desempeñan un papel fundamental a la hora de evaluar la corrección semántica del programa y prepararlo para la generación de código posterior [11].

Cabe destacar que existen distintos métodos para implementar la verificación semántica, lo que da lugar a la clasificación de diferentes tipos de compiladores [11]: compiladores de un solo paso y compiladores de dos o más pasos. En la figura 7 se puede observar una representación reducida de ambos compiladores.

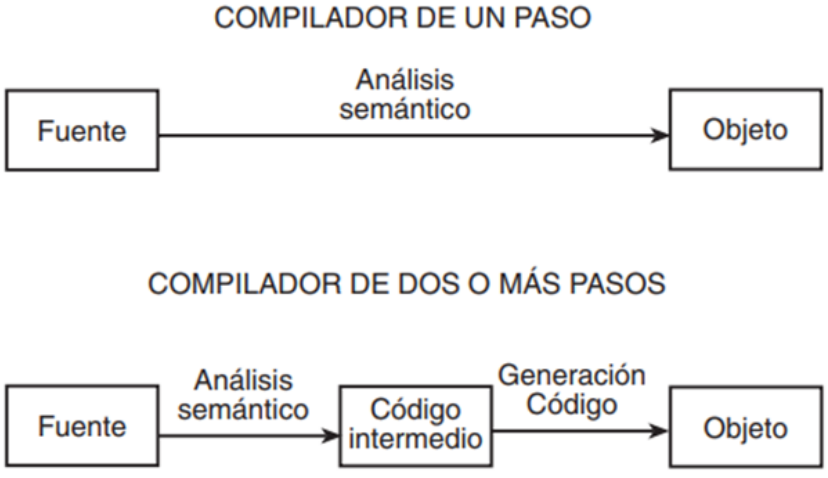

Figura 7: Esquema reducido de la compilación de uno y dos pasos

- **Los compiladores de un solo paso:** Estos compiladores conectan directamente la fase de generación de código con el análisis semántico. Los compiladores de un solo paso son más rápidos que los compiladores de dos o más pasos, pero se sabe que son más difíciles de diseñar y construir [11].

- **Los compiladores de dos o más pasos:** En este enfoque, el analizador semántico genera un código abstracto o código intermedio. La representación intermedia sirve de puente para la generación de código final. Esta representación intermedia es importante, ya que facilita la optimización del código, generando programas eficientes [11].

Las representaciones intermedias se pueden categorizar en dos tipos importantes: la notación de sufijos, que utiliza una pila para la manipulación de datos auxiliares y la evaluación de expresiones, y las tuplas o vectores, que tienen como objetivo abstraer las operaciones de un ensamblador hipotético y especifica qué operaciones deben realizarse, esto implica decidir cuántos componentes contienen las tuplas y qué estructura de información contienen. El uso de representaciones intermedias ofrece una ventaja única: simplifica la generación de código a un nuevo problema de traducción, que se distingue por ser mucho más fácil de traducir que los lenguajes de programación de alto nivel [11].

Si bien los compiladores de un solo paso pueden brindar mayor velocidad en una compilación, la mayoría de los compiladores comerciales han elegido arquitecturas de dos o, a veces, tres pasos por las ventajas obtenidas al separar las fases en la optimización y solo aclarar el proceso en la compilación [11].

3.2. Tabla de Símbolos

La tabla de símbolos es una estructura de datos que contiene la información de un identificador durante el proceso de compilación. El objetivo principal de la tabla de símbolos es recopilar la información completa sobre todos los identificadores que aparecen en el texto del programa fuente. Esta entidad existe durante la compilación y se destruye cuando el código fuente se traduce a código ensamblador [10].

Según lo mencionado por Álvarez, la tabla de símbolos contiene información sobre los diferentes tipos de identificadores del programa, como variables,

tipos de datos, funciones, procedimientos, entre otros. Cada entrada o elemento de la tabla debe contener al menos un identificador con sus atributos asociados [10].

Los atributos estándar que se conservan para cada identificador incluyen [10]:

- **Especificación del identificador:** de qué tipo de elemento se trata (variable, función, procedimiento, etc.).
- **Tipo:** identifica el tipo de dato del identificador (entero, real, booleano, etc.).
- **Dimensión o tamaño:** indica cuánta memoria consume este identificador, especialmente importante para matrices o estructuras de datos.
- **Dirección de inicio para generar código objeto:** especifica la dirección de memoria donde se ubica el identificador.
- **Listas de información adicional:** incluye información muy específica sobre el identificador o el lenguaje de programación.

Cabe mencionar que los atributos de la tabla de símbolos pueden variar considerablemente, dependiendo del lenguaje de programación compilado, sus características específicas y la metodología de desarrollo utilizada para construir el traductor. La creación y el mantenimiento de la tabla de símbolos es un trabajo colaborativo entre el análisis léxico y el análisis sintáctico. El análisis léxico se encarga de identificar la primera lista de identificadores del código fuente, y el análisis sintáctico, y en especial el análisis semántico, se encarga de completar los atributos correspondientes a cada identificador de la tabla [10].

El analizador semántico es el principal responsable de analizar información como el tipo de dato y la validez de uso de los identificadores. Por lo tanto, tras su construcción, la tabla de símbolos se convierte en una fuente de información importante para las etapas posteriores del compilador. Con el mismo acceso, el analizador semántico y el generador de código obtendrán la información necesaria para realizar sus tareas correctamente [10].

Las principales operaciones en las tablas de símbolos son la inserción de nuevos identificadores y la búsqueda de información para los identificadores ya existentes. Además, pueden incluirse operaciones adicionales como “set”, para modificar atributos, y “reset”, para borrar valores. Para compiladores

de alto rendimiento, se suelen utilizar métodos hash para agilizar el acceso a la tabla de símbolos, especialmente en programas grandes con muchos identificadores, ya que permite una búsqueda e inserción rápidas [10].

En resumen, la tabla de símbolos es una estructura de datos vital que sirve como repositorio centralizado de información para los identificadores de un programa. Por lo tanto, su correcta gestión y uso son fundamentales para lograr un proceso de compilación preciso y eficiente.

4. Implementación Básica del Intérprete

4.1. Configuración del Proyecto ANTLR v4

El desarrollo de nuestro intérprete comienza con la configuración adecuada del entorno de trabajo utilizando ANTLR v4 como herramienta principal para la generación de analizadores léxicos y sintácticos [2]. Para este proyecto, se utilizó el entorno de desarrollo IntelliJ IDEA con el arquetipo antlr4-archetype, que proporciona la estructura básica necesaria para implementar un intérprete basado en ANTLR.

La estructura del proyecto se organiza de la siguiente manera:

- **src/main/antlr4/**: Contiene el archivo de gramática `Simple.g4`, donde se definen tanto la gramática regular como la gramática libre de contexto para nuestro lenguaje.
- **target/generated-sources/antlr4/**: Aquí se generan automáticamente las clases Java necesarias para implementar el intérprete:
  - `SimpleLexer.java`: Implementa el analizador léxico.
  - `SimpleParser.java`: Implementa el analizador sintáctico.
  - `SimpleVisitor.java`: Implementa el análisis semántico.
- **src/main/java/**: Contiene la clase `Main.java`, donde se instancian y conectan los componentes del intérprete.
- **test.smp**: Archivo de prueba para nuestro lenguaje.

La configuración inicial requiere la creación del archivo de gramática `Simple.g4`, que servirá como base para la generación automática de las clases necesarias para implementar el intérprete [2].

4.2. Implementación del Analizador Léxico

El analizador léxico, también conocido como lexer, es el primer componente del intérprete y se encarga de transformar el código fuente en un flujo de tokens (Token Stream) [6]. Para implementarlo, es necesario definir en el archivo `Simple.g4` la gramática regular que especifica los patrones que identifican cada tipo de token en nuestro lenguaje.

En nuestra implementación, definimos los siguientes tipos de tokens:

- **Palabras reservadas**: `program`, `var`, `println`
- **Operadores**: `+`, `-`, `*`, `/`, `&&`, `||`, `!`, `>`, `<`, `>=`, `<=`, `==`, `!=`
- **Símbolos de puntuación**: {}, `()`, `;`
- **Identificadores**: Secuencias de letras y números que comienzan con una letra.
- **Constantes numéricas**: Secuencias de dígitos.
- **Espacios en blanco**: Espacios, tabulaciones y saltos de línea (ignorados).

El código correspondiente en el archivo `Simple.g4` se muestra a continuación:

```
grammar Simple;

start
    :   HELLO WORLD
    ;

PROGRAM: 'program';
VAR: 'var';
PRINTLN: 'println';

PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';

AND: '&&';
OR: '||';
NOT: '!';

GT: '>';
LT: '<';
GEQ: '>=';
LEQ: '<=';
```

```
EQ: '==';
NEQ: '!=';

ASSIGN: '=';

BRACKET_OPEN: '{';
BRACKET_CLOSE: '}';

PAR_OPEN: '(';
PAR_CLOSE: ')';

SEMICOLON: ';';

ID: [a-zA-Z][a-zA-Z0-9]*;

NUMBER: [0-9]+;

WS: [ \t\n\r]+ -> skip;
```

Es importante destacar que el orden en el que se definen los tokens es crucial, ya que el análisis se realiza de forma secuencial y los patrones más específicos deben definirse antes que los más generales para evitar ambigüedades [7].

4.3. Implementación del Analizador Sintáctico

Una vez definidos los tokens, el siguiente paso es implementar el analizador sintáctico o parser, que se encarga de construir un árbol de sintaxis concreta (Parse Tree) a partir del flujo de tokens generado por el lexer [6]. Para ello, es necesario definir en el archivo `Simple.g4` la gramática libre de contexto que especifica las reglas sintácticas de nuestro lenguaje.

La gramática libre de contexto se compone de:

- **Símbolos no terminales**: Representan estructuras compuestas por otros elementos (comienzan con minúscula en ANTLR).
- **Símbolos terminales**: Corresponden a los tokens definidos en la gramática regular.

- **Producciones**: Definen cómo se componen los símbolos no terminales a partir de secuencias de otros símbolos.
- **Símbolo inicial**: Representa el punto de partida para el análisis (en nuestro caso, `program`).

El código actualizado en el archivo `Simple.g4` para incluir la gramática libre de contexto se muestra a continuación:

```
grammar Simple;

program: PROGRAM ID BRACKET_OPEN
            sentence*
         BRACKET_CLOSE;

sentence: var_decl | var_assign | println;

var_decl: VAR ID SEMICOLON;
var_assign: ID ASSIGN NUMBER SEMICOLON;
println: PRINTLN NUMBER SEMICOLON;

PROGRAM: 'program';
VAR: 'var';
PRINTLN: 'println';

PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';

AND: '&&';
OR: '||';
NOT: '!';

GT: '>';
LT: '<';
GEQ: '>=';
LEQ: '<=';
EQ: '==';
```

```
NEQ: '!=';

ASSIGN: '=';

BRACKET_OPEN: '{';
BRACKET_CLOSE: '}';

PAR_OPEN: '(';
PAR_CLOSE: ')';

SEMICOLON: ';';

ID: [a-zA-Z][a-zA-Z0-9]*;

NUMBER: [0-9]+;

WS: [ \t\n\r]+ -> skip;
```

Con esta gramática, ANTLR v4 genera automáticamente el código necesario para implementar el analizador sintáctico en la clase `SimpleParser.java`. Para utilizar este analizador, es necesario modificar la clase `Main.java` para instanciar y conectar los componentes del intérprete:

```
public static void main(String[] args) throws IOException {
    String program = args.length > 1 ? args[1] : "test/test." + EXTENSION;

    System.out.println("Interpreting file " + program);

    SimpleLexer lexer = new SimpleLexer(new ANTLRFileStream(program));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    SimpleParser parser = new SimpleParser(tokens);

    SimpleParser.ProgramContext tree = parser.program();

    SimpleCustomVisitor visitor = new SimpleCustomVisitor();
    visitor.visit(tree);

    System.out.println("Interpretation finished");
}
```

4.4. Implementación del Análisis Semántico

El análisis semántico es la tercera fase del intérprete y se encarga de determinar el significado de las construcciones sintácticamente correctas del programa [6]. En nuestra implementación, utilizamos la gramática de atributos proporcionada por ANTLR v4 para añadir información semántica a la gramática libre de contexto.

La gramática de atributos permite:

- Añadir atributos a los símbolos de la gramática.
- Añadir código ejecutable que puede acceder y manipular dichos atributos.

Inicialmente, implementamos un análisis semántico simple que muestra mensajes de depuración para cada tipo de sentencia:

```
var_decl: VAR ID SEMICOLON
    {System.out.println("Declarando variable");};

var_assign: ID ASSIGN expression SEMICOLON
    {System.out.println("Asignando variable");};

println: PRINTLN expression SEMICOLON
    {System.out.println("Imprimiendo por pantalla");};

expression: NUMBER | ID;
```

Posteriormente, mejoramos la implementación para que el intérprete pueda ejecutar realmente las operaciones correspondientes:

```
var_decl: VAR ID SEMICOLON
    {System.out.println("Declarando variable");};

var_assign: ID ASSIGN expression SEMICOLON
    {System.out.println("Asignando variable");};

println: PRINTLN expression SEMICOLON
    {System.out.println($expression.value);};
```

```
expression returns [Object value]:
    NUMBER { $value = Integer.parseInt($NUMBER.text); }
    |
    ID { $value = $ID.text; };
```

Sin embargo, para implementar completamente el manejo de variables, necesitamos una estructura de datos que permita almacenar y recuperar sus valores. Para ello, utilizamos una tabla de símbolos, que es esencialmente un mapa que relaciona nombres de variables con sus valores [8].

Añadimos la tabla de símbolos como un atributo de la clase `SimpleParser` utilizando las directivas `@parser::header` y `@parser::members` de ANTLR v4:

```
@parser::header {
    import java.util.Map;
    import java.util.HashMap;
}

@parser::members {
    Map<String, Object> symbolTable = new HashMap<String, Object>();
}
```

Y actualizamos las reglas semánticas para utilizar esta tabla de símbolos:

```
var_decl: VAR ID SEMICOLON
    {symbolTable.put($ID.text, 0);};

var_assign: ID ASSIGN expression SEMICOLON
    {symbolTable.put($ID.text, $expression.value);};

println: PRINTLN expression SEMICOLON
    {System.out.println($expression.value);};

expression returns [Object value]:
    NUMBER { $value = Integer.parseInt($NUMBER.text); }
    |
    ID { $value = symbolTable.get($ID.text); };
```

4.5. Implementación de Expresiones Aritméticas

Para permitir que nuestro intérprete evalúe expresiones aritméticas, necesitamos ampliar la gramática para incluir operaciones como suma, multiplicación y el uso de paréntesis para controlar la precedencia [7].

La implementación de expresiones aritméticas requiere cuidado para manejar correctamente la precedencia de operadores y evitar la recursividad por la izquierda, que puede causar problemas en el análisis descendente utilizado por ANTLR.

Inicialmente, implementamos una versión simple que solo maneja sumas:

```
expression returns [Object value]:
    t1=term {$value = (int)t1.value;}
    (PLUS t2=term {$value = (int)$value + (int)t2.value;})*;

term returns [Object value]:
    NUMBER {$value = Integer.parseInt($NUMBER.text);}
    | ID {$value = symbolTable.get($ID.text);};
```

Luego, ampliamos la gramática para manejar la precedencia entre suma y multiplicación:

```
expression returns [Object value]:
    t1=factor {$value = (int)t1.value;}
    (PLUS t2=factor {$value = (int)$value + (int)t2.value;})*;

factor returns [Object value]:
    t1=term {$value = (int)t1.value;}
    (MULT t2=term {$value = (int)$value * (int)t2.value;})*;

term returns [Object value]:
    NUMBER {$value = Integer.parseInt($NUMBER.text);}
    | ID {$value = symbolTable.get($ID.text);};
```

Finalmente, añadimos soporte para el uso de paréntesis que permiten alterar la precedencia natural de las operaciones:

```
term returns [Object value]:
    NUMBER {$value = Integer.parseInt($NUMBER.text); }
    | ID {$value = symbolTable.get($ID.text);}
    | PAR_OPEN expression PAR_CLOSE {$value = $expression.value;};
```

4.6. Implementación del Patrón Interpreter

A medida que el lenguaje crece en complejidad, especialmente al añadir estructuras de control como condicionales, el enfoque basado en la evaluación directa dentro de la gramática se vuelve limitado. Para superar estas limitaciones, implementamos el patrón Interpreter, que permite representar la gramática del lenguaje como una jerarquía de clases [4].

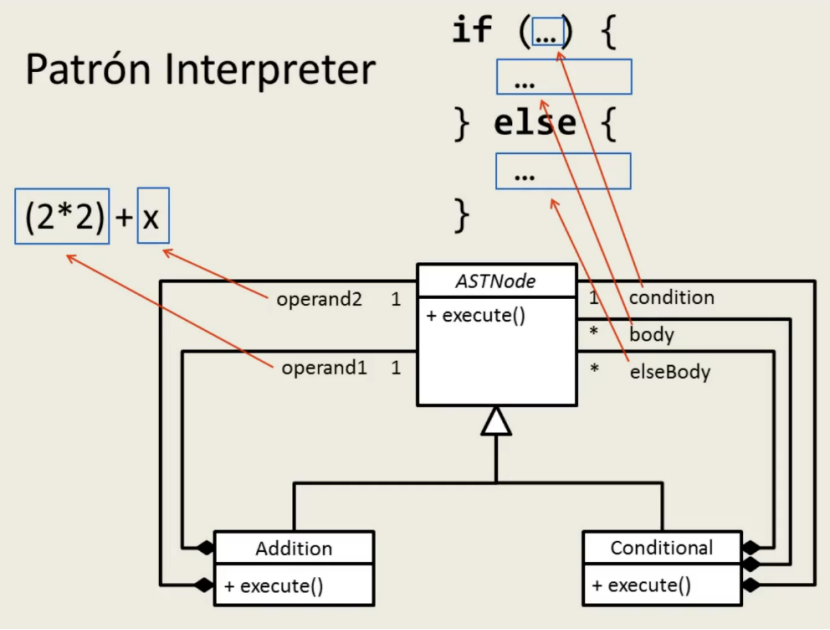

Figura 8: Diagrama de clases del patrón Interpreter implementado

El patrón Interpreter se basa en la creación de una jerarquía de clases cuyas instancias representan los nodos del árbol de sintaxis abstracta (AST). Cada nodo conoce cómo ejecutar el código correspondiente a esa parte del AST.

La estructura básica del patrón incluye:

- Una interfaz o clase abstracta `ASTNode` que define el método `execute()`.
- Clases concretas para cada tipo de expresión o sentencia en el lenguaje.

Implementamos las siguientes clases para nuestro intérprete:

La interfaz `ASTNode`:

```
package com.miorganizacion.simple.interprete.ast;
public interface ASTNode {
    public Object execute();
}
```

La clase `Addition` para sumas:

```
package com.miorganizacion.simple.interprete.ast;
public class Addition implements ASTNode {
    private ASTNode operand1;
    private ASTNode operand2;

    public Addition(ASTNode operand1, ASTNode operand2) {
        super();
        this.operand1 = operand1;
        this.operand2 = operand2;
    }

    @Override
    public Object execute() {
        return (int) operand1.execute() + (int) operand2.execute();
    }
}
```

La clase `Multiplication` para multiplicaciones:

```
package com.miorganizacion.simple.interprete.ast;
public class Multiplication implements ASTNode {
    private ASTNode operand1;
    private ASTNode operand2;

    public Multiplication(ASTNode operand1, ASTNode operand2) {
        super();
        this.operand1 = operand1;
        this.operand2 = operand2;
    }

    @Override
    public Object execute() {
```

```
        return (int) operand1.execute() * (int) operand2.execute();
    }
}
```

La clase `Println` para imprimir en consola:

```
package com.miorganizacion.simple.interprete.ast;
public class Println implements ASTNode {
    private ASTNode data;

    public Println(ASTNode data) {
        super();
        this.data = data;
    }

    @Override
    public Object execute() {
        System.out.println(data.execute());
        return null; //No necesita retornar ningún dato
    }
}
```

La clase `If` para condicionales:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Iterator;
import java.util.List;

public class If implements ASTNode {
    private ASTNode condition;
    private List<ASTNode> body;
    private List<ASTNode> elseBody;

    public If(ASTNode condition, List<ASTNode> body, List<ASTNode> elseBody) {
        super();
        this.condition = condition;
        this.body = body;
        this.elseBody = elseBody;
    }
```

```
    @Override
    public Object execute() {
        if ((boolean) condition.execute()) {
            for (ASTNode n : body) {
                n.execute();
            }
        } else {
            for (ASTNode n : elseBody) {
                n.execute();
            }
        }
        return null;
    }
}
```

La clase `Constant` para valores constantes:

```
package com.miorganizacion.simple.interprete.ast;
public class Constant implements ASTNode {
    private Object value;

    public Constant(Object value) {
        super();
        this.value = value;
    }

    @Override
    public Object execute() {
        return value;
    }
}
```

Una vez implementadas estas clases, modificamos la gramática para utilizar el patrón Interpreter en lugar de la evaluación directa:

```
grammar Simple;

@parser::header {
```

```
    import java.util.Map;
    import java.util.HashMap;
    import java.util.List;
    import java.util.ArrayList;
    import com.miorganizacion.simple.interprete.ast.*;
}

program: PROGRAM ID BRACKET_OPEN
    {
        List<ASTNode> body = new ArrayList<ASTNode>();
    }
    (sentence {body.add($sentence.node);})*
    BRACKET_CLOSE
    {
        for (ASTNode n : body) {
            n.execute();
        }
    };

sentence returns [ASTNode node]:
    println   {$node = $println.node;}
  | conditional {$node = $conditional.node;};

println returns [ASTNode node]:
    PRINTLN expression SEMICOLON
    {$node = new Println($expression.node);};

conditional returns [ASTNode node]:
    IF PAR_OPEN expression PAR_CLOSE
    {
        List<ASTNode> body = new ArrayList<ASTNode>();
    }
    BRACKET_OPEN (s1=sentence {body.add($s1.node);})* BRACKET_CLOSE
    ELSE
    {
        List<ASTNode> elseBody = new ArrayList<ASTNode>();
    }
    BRACKET_OPEN (s2=sentence {elseBody.add($s2.node);})* BRACKET_CLOSE
```

```
    {
        $node = new If($expression.node, body, elseBody);
    };

expression returns [ASTNode node]:
    t1=factor {$node = $t1.node;}
    (PLUS t2=factor {$node = new Addition($node, $t2.node);})*;

factor returns [ASTNode node]:
    t1=term {$node = $t1.node;}
    (MULT t2=term {$node = new Multiplication($node, $t2.node);})*;

term returns [ASTNode node]:
    NUMBER  {$node = new Constant(Integer.parseInt($NUMBER.text));}
  | BOOLEAN {$node = new Constant(Boolean.parseBoolean($BOOLEAN.text));}
  | PAR_OPEN expression PAR_CLOSE {$node = $expression.node;};

PROGRAM: 'program';
VAR: 'var';
PRINTLN: 'println';
IF: 'if';
ELSE: 'else';

PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';

AND: '&&';
OR: '||';
NOT: '!';

GT: '>';
LT: '<';
GEQ: '>=';
LEQ: '<=';
EQ: '==';
NEQ: '!=';
```

```
ASSIGN: '=';

BRACKET_OPEN: '{';
BRACKET_CLOSE: '}';

PAR_OPEN: '(';
PAR_CLOSE: ')';

SEMICOLON: ';';

BOOLEAN: 'true' | 'false';

ID: [a-zA-Z][a-zA-Z0-9]*;

NUMBER: [0-9]+;

WS: [ \t\n\r]+ -> skip;
```

4.7. Gestión de Variables

Finalmente, ampliamos nuestro intérprete para incluir la gestión completa de variables, incluyendo su declaración, asignación y referencia [8]. Para ello, implementamos las siguientes clases adicionales:

La clase `VarDecl` para declaración de variables:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public class VarDecl implements ASTNode {
    private String name;

    public VarDecl(String name) {
        this.name = name;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
```

```
        symbolTable.put(name, 0);
        return null;
    }
}
```

La clase `VarAssign` para asignación de variables:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public class VarAssign implements ASTNode {
    private String name;
    private ASTNode value;

    public VarAssign(String name, ASTNode value) {
        this.name = name;
        this.value = value;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        symbolTable.put(name, value.execute(symbolTable));
        return null;
    }
}
```

La clase `VarRef` para referencia a variables:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public class VarRef implements ASTNode {
    private String name;

    public VarRef(String name) {
        this.name = name;
    }

    @Override
```

```
    public Object execute(Map<String, Object> symbolTable) {
        return symbolTable.get(name);
    }
}
```

También modificamos la interfaz `ASTNode` para incluir la tabla de símbolos como parámetro:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public interface ASTNode {
    public Object execute(Map<String, Object> symbolTable);
}
```

Y actualizamos todas las demás clases para utilizar este nuevo método. Finalmente, modificamos la gramática para incluir estas nuevas funcionalidades:

```
program: PROGRAM ID BRACKET_OPEN
    {
        List<ASTNode> body = new ArrayList<ASTNode>();
        Map<String, Object> symbolTable = new HashMap<String, Object>();
    }
    (sentence {body.add($sentence.node);})*
    BRACKET_CLOSE
    {
        for (ASTNode n : body) {
            n.execute(symbolTable);
        }
    };

sentence returns [ASTNode node]:
      println     {$node = $println.node;}
    | conditional {$node = $conditional.node;}
    | var_decl    {$node = $var_decl.node;}
    | var_assign  {$node = $var_assign.node;};

var_decl returns [ASTNode node]:
```

```
    VAR ID SEMICOLON {$node = new VarDecl($ID.text);};

var_assign returns [ASTNode node]:
    ID ASSIGN expression SEMICOLON {$node = new
    VarAssign($ID.text, $expression.node);};

term returns [ASTNode node]:
      NUMBER  {$node = new Constant(Integer.parseInt($NUMBER.text));}
    | BOOLEAN {$node = new Constant(Boolean.parseBoolean($BOOLEAN.text));}
    | ID      {$node = new VarRef($ID.text);}
    | PAR_OPEN expression PAR_CLOSE {$node = $expression.node;};
```

Con estas modificaciones, nuestro intérprete es capaz de manejar declaraciones de variables, asignaciones, expresiones aritméticas, impresión de resultados y estructuras condicionales, proporcionando una base sólida que puede ser extendida con funcionalidades adicionales como ámbitos de variables, funciones, bucles, entre otros [6].

5. Implementación Avanzada del Intérprete

Una vez establecida la implementación básica del intérprete, se procede a realizar mejoras que amplíen sus capacidades y funcionalidades. En esta sección se describen las extensiones propuestas para enriquecer el lenguaje con operaciones matemáticas y lógicas más avanzadas, aprovechando la estructura modular proporcionada por el patrón Interpreter [4].

5.1. Extensión con Operación Lógica XOR

La primera mejora propuesta es la implementación de la operación lógica XOR (OR exclusivo), que complementa las operaciones lógicas básicas ya implementadas como AND y OR. La operación XOR devuelve verdadero si exactamente uno de sus operandos es verdadero, pero no ambos [6].

5.1.1. Extensión de la Gramática

Para implementar esta operación, primero debemos extender la gramática en el archivo `Simple.g4` para incluir el nuevo operador XOR:

```
grammar Simple;

// Añadir el token para XOR
XOR: '^';

// Modificar la gramática para incluir expresiones lógicas
logical_expr returns [ASTNode node]:
    t1=comp_expr {$node = $t1.node;}
    (
        AND t2=comp_expr {$node = new LogicalAnd($node, $t2.node);}
      | OR t2=comp_expr {$node = new LogicalOr($node, $t2.node);}
      | XOR t2=comp_expr {$node = new LogicalXor($node, $t2.node);}
    )*;

// Resto de la gramática permanece igual
```

5.1.2. Implementación de la Clase LogicalXor

A continuación, implementamos la clase `LogicalXor` que se integra en nuestra jerarquía de clases del patrón Interpreter:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public class LogicalXor implements ASTNode {
    private ASTNode left;
    private ASTNode right;

    public LogicalXor(ASTNode left, ASTNode right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        boolean leftValue = (boolean) left.execute(symbolTable);
        boolean rightValue = (boolean) right.execute(symbolTable);
        return leftValue ^ rightValue;
    }
}
```

Esta implementación utiliza el operador XOR nativo de Java (`^`) para realizar la operación lógica sobre los valores booleanos obtenidos de evaluar los operandos izquierdo y derecho.

5.2. Implementación de Operación Matemática: Serie de Fibonacci

La segunda mejora propuesta es la implementación de una función matemática más compleja: el cálculo de números de la serie de Fibonacci. Esta serie, donde cada número es la suma de los dos anteriores (comenzando con 0 y 1), tiene aplicaciones en diversos campos como la matemática, la informática y la naturaleza [9].

5.2.1. Extensión de la Gramática

Extendemos la gramática para incluir la nueva función `fibonacci`:

```
grammar Simple;

// Añadir la palabra reservada para fibonacci
FIBONACCI: 'fibonacci';

// Añadir la producción para la función fibonacci
term returns [ASTNode node]:
      NUMBER  {$node = new Constant(Integer.parseInt($NUMBER.text));}
    | BOOLEAN {$node = new Constant(Boolean.parseBoolean($BOOLEAN.text));}
    | ID      {$node = new VarRef($ID.text);}
    | FIBONACCI PAR_OPEN expression PAR_CLOSE
        {$node = new Fibonacci($expression.node);}
    | PAR_OPEN expression PAR_CLOSE {$node = $expression.node;};

// Resto de la gramática permanece igual
```

5.2.2. Implementación de la Clase Fibonacci

Implementamos la clase `Fibonacci` para calcular el n-ésimo número de la serie:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;

public class Fibonacci implements ASTNode {
    private ASTNode index;

    public Fibonacci(ASTNode index) {
        this.index = index;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        int n = (int) index.execute(symbolTable);
```

```
        if (n < 0) {
            throw new RuntimeException("Fibonacci no está definido para
            números negativos");
        }

        if (n == 0) return 0;
        if (n == 1) return 1;

        int fib1 = 0;
        int fib2 = 1;
        int result = 0;

        for (int i = 2; i <= n; i++) {
            result = fib1 + fib2;
            fib1 = fib2;
            fib2 = result;
        }

        return result;
    }
}
```

Esta implementación calcula el n-ésimo número de Fibonacci de forma iterativa, evitando la ineficiencia de la implementación recursiva para números grandes [7].

5.3. Implementación de Operación Probabilística: Generador de Números Aleatorios

La tercera mejora propuesta es la implementación de una función probabilística para generar números aleatorios dentro de un rango específico, con la opción de especificar una distribución de probabilidad [9].

5.3.1. Extensión de la Gramática

Extendemos la gramática para incluir la nueva función `random`:

```
grammar Simple;
```

```
// Añadir las palabras reservadas para random
RANDOM: 'random';
UNIFORM: 'uniform';
NORMAL: 'normal';

// Añadir las producciones para la función random
term returns [ASTNode node]:
      NUMBER  {$node = new Constant(Integer.parseInt($NUMBER.text));}
    | BOOLEAN {$node = new Constant(Boolean.parseBoolean($BOOLEAN.text));}
    | ID      {$node = new VarRef($ID.text);}
    | FIBONACCI PAR_OPEN expression PAR_CLOSE
        {$node = new Fibonacci($expression.node);}
    | RANDOM PAR_OPEN expression COMMA expression
        (COMMA UNIFORM {$node = new RandomUniform($expression.node,
        $expression.node);} |
         COMMA NORMAL {$node = new RandomNormal($expression.node,
         $expression.node);})
      PAR_CLOSE
    | PAR_OPEN expression PAR_CLOSE {$node = $expression.node;};

// Añadir token para coma
COMMA: ',';

// Resto de la gramática permanece igual
```

5.3.2. Implementación de las Clases RandomUniform y RandomNormal

Implementamos las clases para generar números aleatorios con diferentes distribuciones:

```
package com.miorganizacion.simple.interprete.ast;
import java.util.Map;
import java.util.Random;

public class RandomUniform implements ASTNode {
    private ASTNode min;
```

```
    private ASTNode max;
    private Random random;

    public RandomUniform(ASTNode min, ASTNode max) {
        this.min = min;
        this.max = max;
        this.random = new Random();
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        int minValue = (int) min.execute(symbolTable);
        int maxValue = (int) max.execute(symbolTable);

        if (minValue >= maxValue) {
            throw new RuntimeException("El valor mínimo debe ser
            menor que el máximo");
        }

        return minValue + random.nextInt(maxValue - minValue + 1);
    }
}

package com.miorganizacion.simple.interprete.ast;
import java.util.Map;
import java.util.Random;

public class RandomNormal implements ASTNode {
    private ASTNode mean;
    private ASTNode stdDev;
    private Random random;

    public RandomNormal(ASTNode mean, ASTNode stdDev) {
        this.mean = mean;
        this.stdDev = stdDev;
        this.random = new Random();
    }
```

```
    @Override
    public Object execute(Map<String, Object> symbolTable) {
        double meanValue = (double) mean.execute(symbolTable);
        double stdDevValue = (double) stdDev.execute(symbolTable);

        if (stdDevValue <= 0) {
            throw new RuntimeException("La desviación estándar
            debe ser positiva");
        }

        // Generamos un número aleatorio con distribución normal
        // usando el método de Box-Muller
        double u1 = random.nextDouble(); // Uniforme (0,1)
        double u2 = random.nextDouble(); // Uniforme (0,1)

        // Transformación Box-Muller
        double z0 = Math.sqrt(-2.0 * Math.log(u1)) *
        Math.cos(2.0 * Math.PI * u2);

        // Ajustamos a la media y desviación estándar deseadas
        return meanValue + stdDevValue * z0;
    }
}
```

5.3.3. Ejemplo de Uso

Con estas mejoras, nuestro lenguaje Simple ahora puede realizar operaciones más avanzadas. Por ejemplo:

```
program AdvancedExample {
    // Ejemplo de uso de XOR
    var a;
    var b;
    a = true;
    b = false;
    println a ^ b; // Imprime: true

    // Ejemplo de uso de Fibonacci
```

```
    var n;
    n = 10;
    println fibonacci(n); // Imprime: 55

    // Ejemplo de uso de Random
    var min;
    var max;
    min = 1;
    max = 100;
    println random(min, max, uniform);
    // Imprime un número aleatorio entre 1 y 100

    // Ejemplo de Random con distribución normal
    var media;
    var desviacion;
    media = 50;
    desviacion = 10;
    println random(media, desviacion, normal);
    // Imprime un número aleatorio con distribución normal
}
```

5.4. Integración de las Mejoras

Estas extensiones demuestran la flexibilidad y extensibilidad proporcionadas por el patrón Interpreter [4], permitiendo añadir nuevas funcionalidades al lenguaje de manera modular sin modificar significativamente la estructura existente [8].

La implementación de estas mejoras enriquece considerablemente las capacidades del intérprete, permitiendo realizar operaciones más complejas y facilitando su uso en diversos contextos, desde fines educativos hasta aplicaciones prácticas [9].

En futuras iteraciones, se podrían implementar características adicionales como:

- Funciones definidas por el usuario
- Estructuras de control más avanzadas (bucles while, for, etc.)
- Manejo de errores más robusto

- Optimizaciones de rendimiento
- Soporte para tipos de datos adicionales (punto flotante, cadenas, etc.)

Estas posibles mejoras seguirían el mismo enfoque modular, extendiendo la gramática y añadiendo nuevas clases a la jerarquía del patrón Interpreter [12].

6. Conclusiones

A lo largo de este trabajo, se ha desarrollado un intérprete funcional utilizando ANTLR v4 como herramienta principal, siguiendo un enfoque incremental que ha permitido comprender a profundidad cada una de las etapas del proceso de interpretación de un lenguaje de programación. Tras este recorrido, podemos extraer varias conclusiones relevantes:

En primer lugar, la implementación de un intérprete siguiendo las etapas clásicas del análisis léxico, sintáctico y semántico ha demostrado ser un enfoque estructurado y efectivo, tal como lo describen Aho et al. [6]. La separación clara de estas fases ha facilitado el desarrollo progresivo del intérprete, permitiendo centrarse en cada problema específico de manera aislada. Sin embargo, también se ha constatado que la interrelación entre estas etapas es compleja y requiere un diseño cuidadoso, especialmente en lo que respecta a la tabla de símbolos, que sirve como puente de comunicación entre las diferentes fases del intérprete [10]. La experiencia adquirida en este proyecto refuerza la importancia de una base teórica sólida antes de abordar la implementación práctica de un intérprete.

En segundo lugar, la aplicación del patrón de diseño Interpreter [4] ha resultado ser especialmente adecuada para la implementación del análisis semántico y la ejecución del código. Este patrón ha proporcionado una estructura modular y extensible que ha facilitado la incorporación de nuevas funcionalidades sin necesidad de modificar significativamente el código existente. La estructura jerárquica de clases correspondientes a las diferentes construcciones del lenguaje ha mejorado la mantenibilidad y legibilidad del código, demostrando así la relevancia de los patrones de diseño en el desarrollo de sistemas complejos como los intérpretes y compiladores [2].

Por último, el uso de herramientas modernas como ANTLR v4 ha simplificado significativamente aspectos técnicamente complejos del desarrollo de intérpretes, como la generación de analizadores léxicos y sintácticos a partir de gramáticas declarativas. Esto ha permitido centrar los esfuerzos en aspectos más conceptuales, como el diseño del lenguaje y la implementación del análisis semántico [9]. No obstante, también se ha observado que estas herramientas no eliminan la necesidad de comprender los fundamentos teóricos subyacentes, sino que más bien requieren un conocimiento profundo de estos para aprovechar todo su potencial. La combinación de herramientas modernas con conocimientos teóricos sólidos representa el enfoque óptimo para el desarrollo de intérpretes y compiladores en la actualidad [7].

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2007, conocido comúnmente como el "Libro del Dragón".

[2] T. Parr, *The Definitive ANTLR 4 Reference*. Dallas, TX: Pragmatic Bookshelf, 2013.

[3] K. C. Louden and K. A. Lambert, *Programming Languages: Principles and Practice*, 3rd ed. Boston, MA: Cengage Learning, 2011.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995, conocido comúnmente como el libro "Gang of Four".

[5] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge, UK: Cambridge University Press, 2004.

[6] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers - Principles, Techniques and Tools*. Edingburgh Gate: Pearson Education Limited, 2014.

[7] K. Louden, *Compiler Construction - Principles and Practice*. U.S.A.: PWS Publishing Company, 1997.

[8] A. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[9] C. Fisher, R. Cytron, and R. LeBlanc Jr., *Crafting a Compiler*. Addison Wesley, Pearson, 2014.

[10] G. A. Álvarez, "Compiladores estructura y procesos de compilación," sin fecha.

[11] M. Alfonseca Moreno, M. de la Cruz Echeandía, A. Ortega de la Puente, and E. Pulido Cañabate, *Compiladores e intérpretes: teoría y práctica*. Madrid: Pearson Educación, 2006.

[12] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. Langendoen, *Modern Compiler Design*, 2nd ed. New York: Springer, 2012.