Universidad Peruana Cayetano Heredia

Facultad de Ingeniería

Proyecto Final:
Implementación del Lenguaje Algoritmia

Curso: Implementación de Lenguajes de Programación

Profesor: Mg. Wilder Nina Choquehuayta

Integrantes:

Yosselin Cosme Pérez
Luis Martin Valenzuela Valer
Juan Diego López Vega
Camilo Sebastián Silva Cuzqui

Lima, Perú
2025

Índice general

Capítulo 1

Introducción

El presente documento describe el proceso de construcción del proyecto final del curso **Implementación de Lenguajes de Programación**, cuyo resultado es el lenguaje **Algoritmia**: un lenguaje de dominio específico orientado a la composición musical algorítmica. El proyecto consistió en implementar un doble intérprete capaz de transformar un programa fuente en tres productos finales: una partitura digital, un archivo MIDI y un archivo WAV.

Para ello se emplearon herramientas clave: ANTLR4 para la construcción del lexer y parser, Python para el desarrollo del intérprete, y los programas externos LilyPond y Timidity++ para la generación musical. Este documento explica el funcionamiento del lenguaje, la estructura de su gramática, las decisiones de diseño tomadas y la integración entre las distintas herramientas para producir música a partir de código.

1.1. Objetivos

1.1.1. Objetivo General

Desarrollar e implementar el lenguaje **Algoritmia**, integrando un analizador léxico–sintáctico, un intérprete funcional y un conversor musical que genere artefactos finales en formatos PDF, MIDI y WAV.

1.1.2. Objetivos Específicos

- Definir formalmente la gramática del lenguaje mediante ANTLR4, incorporando su estructura jerárquica, reglas sintácticas, manejo de listas, expresiones, procedimientos y delimitadores de bloques estrictos.
- Implementar un intérprete en Python que ejecute la semántica del lenguaje, incluyendo estructuras de control, llamadas a procedimientos con parámetros, aritmética, listas, lectura, escritura y reproducción musical.
- Construir el componente musical que traduce expresiones numéricas y listas en secuencias de notas compatibles con LilyPond.
- Integrar herramientas externas (LilyPond y Timidity++) para generar los archivos de salida: partitura, archivo MIDI y audio WAV.

1.2. Diseño de la Gramática

La gramática del lenguaje **Algoritmia**, definida en el archivo `Algoritmia (1).g4`, se diseñó buscando precisión sintáctica, claridad y robustez. Una de las características más importantes del diseño actual es el uso de un sistema de **bloques estrictos** delimitados mediante los tokens `|:` y `:|`. Además, la gramática incorpora un token `NL` que representa saltos de línea y permite controlar de manera más exacta la estructura del código fuente.

Este enfoque permite evitar ambigüedades, mejorar la legibilidad del código e imponer reglas claras sobre dónde pueden aparecer instrucciones o bloques.

1.3. Estructura Principal del Programa

Un programa en Algoritmia está compuesto por uno o más procedimientos escritos en mayúscula. Cada procedimiento puede incluir parámetros y debe contener un bloque de instrucciones delimitado por `|:` y `:|`. A diferencia de versiones previas, los saltos de línea ahora forman parte del análisis sintáctico.

```
programa:
    (NL)* procedimiento+ EOF ;

procedimiento:
    ID_MAYUS parametros? bloq_inicio instrucciones
    bloq_fin (NL)* ;
```

Este diseño impone una estructura clara y evita que una instrucción quede fuera de su bloque.

1.4. Instrucciones del Lenguaje

La gramática actualizada centraliza todas las instrucciones en una sola regla:

- asignación con `<-`
- lectura con `<?>`
- escritura con `<w>`
- reproducción musical con `(: expr)` o `<:> expr`
- estructuras de control `if` y `else`
- bucles `while`
- llamadas a procedimientos con parámetros opcionales
- operaciones de listas: inserción `«` y extracción `8<`

```
instruccion
    : asignacion
    | lectura
    | escritura
    | condicional
    | while
    | llamada_proc
    | reproduccion
    | addlista
    | poplista ;
```

Cada instrucción posee una semántica directa, facilitando su implementación en el intérprete.

1.5. Expresiones

El sistema de expresiones incluye:

- operadores aritméticos: `+`, `-`, `*`, `/`, `%`
- operadores relacionales: `=`, `/=`, `<`, `>`, `<=`, `>=`
- operador unario negativo
- acceso a listas por índice: `lista[expr]`
- llamados a procedimientos sin paréntesis

```
factor:
      '-' factor
    | '(' expr ')'
    | INT
    | ID_MINUSCULA
    | ID_MAYUS
    | ID_MINUSCULA '[' expr ']'
    | '#' ID_MINUSCULA
    | lista ;
```

El lenguaje no define booleanos; todas las condiciones se evalúan como enteros (`0` o `1`), lo cual simplifica la ejecución de estructuras de control.

1.6. Reglas Musicales y Operaciones con Listas

El lenguaje permite representar secuencias musicales utilizando listas definidas con llaves:

$$\{\,60 \quad 62 \quad 64\,\}$$

Las operaciones sobre listas permiten:

- insertar un elemento al inicio con `«`
- extraer el primer elemento mediante `8<`

- acceder a cualquier posición usando índices

Estas funciones permiten construir melodías de manera algorítmica y combinarlas con otras estructuras del lenguaje.

1.7. Lexer y Manejo de Comentarios

El lexer del lenguaje hace uso del token `NL` para capturar saltos de línea y asegurar una estructura correctamente delimitada. Los espacios y tabulaciones son ignorados mediante `WS`. Los comentarios se definen mediante el patrón:

```
### ... ###
```

y son completamente ignorados por el analizador.

```
COMMENT: '###' .*? '###' -> skip ;
```

El lexer define además reglas para cadenas, números enteros, identificadores en mayúsculas (para notas o procedimientos) e identificadores en minúsculas (para variables). Este diseño contribuye a una sintaxis clara, limpia y fácil de mantener.

Capítulo 2

Implementación del Intérprete

La implementación del intérprete del lenguaje **Algoritmia** se desarrolló siguiendo una estructura modular que permitió mantener el proyecto claro, organizado y fácil de depurar. Cada componente del sistema —la gramática, el análisis sintáctico, el motor de ejecución, la conversión musical y el entorno de pruebas— fue diseñado como un módulo independiente con responsabilidades bien definidas. Esta separación permitió iterar sobre el diseño del lenguaje sin comprometer la estabilidad del sistema, garantizando que el intérprete pudiera generar correctamente las partituras y los archivos de audio a partir del código escrito en Algoritmia.

2.1. Estructura del Proyecto

El proyecto se encuentra organizado dentro del directorio `src/`, el cual contiene todos los componentes asociados al análisis, evaluación y generación del contenido musical. La estructura final del proyecto se muestra en la Figura 2.1.

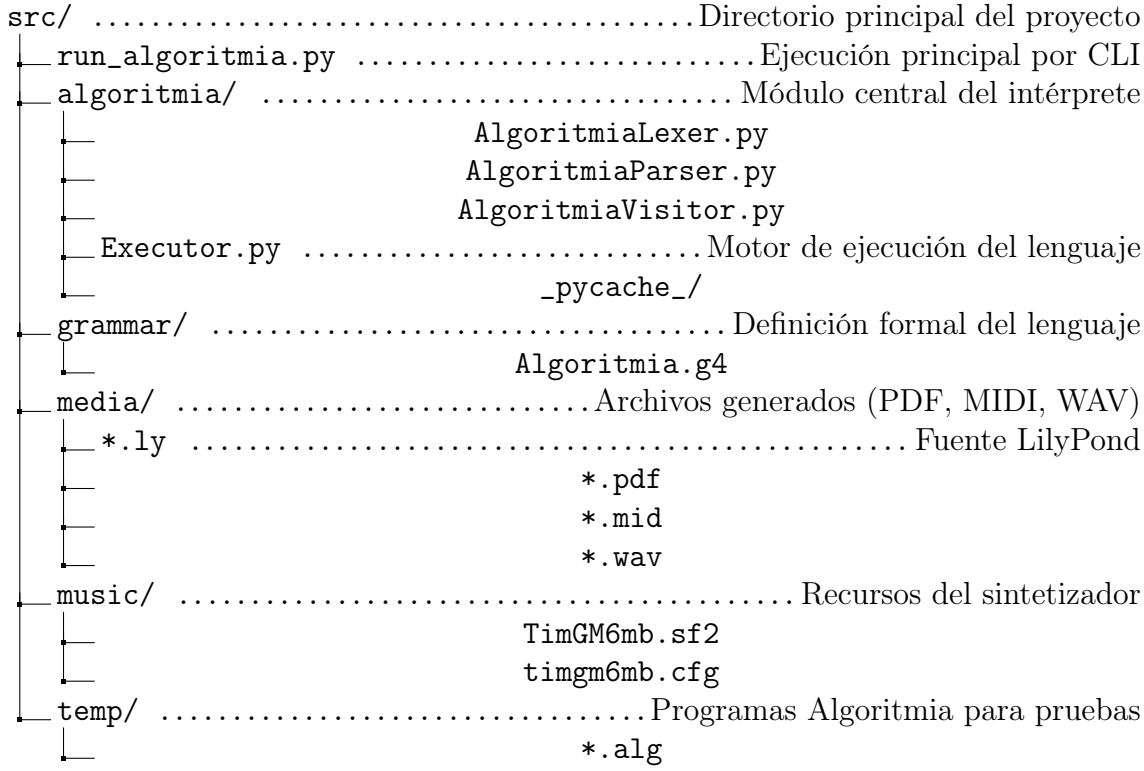

Figura 2.1: Estructura modular actualizada del proyecto Algoritmia.

Cada submódulo cumple una función específica dentro del sistema:

- **grammar**/: contiene la gramática utilizada por ANTLR4 para generar el lexer y el parser.
- **algoritmia**/: núcleo del intérprete, donde se construye el AST y se ejecutan las instrucciones definidas en el lenguaje.
- **media**/: almacena las partituras, archivos MIDI y WAV generados tras la ejecución de un programa.
- **music**/: incluye los bancos de sonido necesarios para la conversión del archivo MIDI a WAV mediante Timidity++.
- **temp**/: contiene los archivos `.alg` utilizados en las pruebas y experimentación del lenguaje.

2.2. Motor del Intérprete: Clase `Executor`

El componente central del sistema es la clase `AlgoritmiaExecutor`, ubicada en `Executor.py`. Esta clase extiende el visitante generado por ANTLR4 e implementa las funciones necesarias para interpretar el árbol de sintaxis abstracta (AST). Entre sus responsabilidades principales se encuentran:

- registrar los procedimientos definidos en el archivo fuente,
- gestionar las variables locales y los parámetros de cada procedimiento,

- evaluar expresiones aritméticas, comparaciones y operaciones con listas,
- controlar estructuras de flujo como `if/else` y `while`,
- ejecutar llamadas a procedimientos,
- recolectar y transformar notas musicales,
- generar los archivos PDF, MIDI y WAV resultantes.

2.2.1. Registro de Procedimientos y Punto de Entrada

Durante el recorrido inicial del árbol sintáctico, el intérprete registra todos los procedimientos definidos en el programa. Cada procedimiento se almacena mediante su identificador en mayúscula y posteriormente puede invocarse desde cualquier parte del código. Este mecanismo permite que el usuario seleccione el procedimiento de inicio desde la línea de comandos, sin restringir la ejecución al clásico `Main`.

```
def visitPrograma(self, ctx):
    for p in ctx.procedimiento():
        name = p.ID_MAYUS().getText()
        self.procedimientos[name] = p

    target = self.entry_point  # Puede ser 'Main' u otro procedimiento
    self.visitProcedimiento(self.procedimientos[target])
```

2.2.2. Gestión de Variables y Ámbitos

Cada vez que se ejecuta un procedimiento, el intérprete crea un nuevo diccionario de variables locales. En lugar de mantener una pila de entornos, la implementación conserva temporalmente el estado previo y lo restaura al finalizar la ejecución. Este diseño mantiene el aislamiento entre procedimientos sin añadir complejidad innecesaria a la memoria del intérprete.

```
prev_vars = self.variables
self.variables = {}
...
self.variables = prev_vars
```

2.2.3. Evaluación de Expresiones

Las expresiones se procesan respetando la precedencia establecida en la gramática. Esto incluye operaciones aritméticas, comparaciones relacionales y concatenaciones de listas. Las comparaciones devuelven 0 o 1, lo que simplifica su uso dentro de las estructuras de control.

```
def visitAsignacion(self, ctx):
    name = ctx.ID_MINUSCULA().getText()
    val = self.visit(ctx.expr())
    self.variables[name] = copy.deepcopy(val) if isinstance(val, list)
    else val
```

2.2.4. Control de Flujo

El intérprete admite estructuras de control mediante visitas recursivas. Para evitar bucles infinitos durante la ejecución de un `while`, la implementación mantiene un contador de iteraciones y detiene el ciclo si se supera un límite predefinido.

```
def visitWhile(self, ctx):
    count = 0
    while self.visit(ctx.expr()) != 0:
        self.visit(ctx.instrucciones())
        count += 1
        if count > self.MAX_WHILE_ITERS:
            raise RuntimeError("Bucle infinito detectado.")
```

2.3. Aritmética Musical

Algoritmia permite operar directamente sobre notas musicales, tratándolas de forma equivalente a enteros. Para ello, las notas se convierten a un sistema interno basado en 7 grados por octava (A–G), donde el salto de octava ocurre entre B y C. La conversión es la siguiente:

- A0 = 0, B0 = 1, C1 = 2, ..., G1 = 6,
- cada octava incrementa el valor en 7 unidades.

```
def _nota_a_int(self, nota):
    # Convierte 'C4' en un entero interno basado en 7 grados por
    octava
```

La función inversa reconstruye la notación musical antes de generar la partitura.

2.4. Generación Musical: PDF, MIDI y WAV

Una vez finalizada la interpretación del programa, el intérprete genera un archivo `.ly` compatible con LilyPond. Este archivo contiene la secuencia de notas recopiladas durante la ejecución. LilyPond se encarga de producir automáticamente la partitura en PDF y el archivo MIDI correspondiente. Finalmente, Timidity++ convierte el archivo MIDI a formato WAV utilizando un archivo de configuración temporal.

```
subprocess.run(["lilypond", "archivo.ly"], cwd=self.OUTPUT_DIR)
subprocess.run(["timidity", midi, "-Ow", "-o", wav, "-c", temp_cfg])
```

Los archivos generados se almacenan en el directorio `media/` para su posterior revisión.

2.5. Ejecución por Línea de Comandos

El script `run_algoritmia.py` permite ejecutar cualquier programa Algoritmia desde la terminal. El usuario puede:

- listar los archivos `.alg` disponibles,

- seleccionar el archivo a ejecutar,
- especificar el procedimiento inicial,
- generar automáticamente los archivos musicales resultantes.

Este módulo funciona como un entorno de pruebas práctico y accesible.

2.6. Programas de Prueba

Los archivos ubicados en la carpeta `temp/` se emplearon para validar diversos aspectos del lenguaje, tales como:

- estructuras de control,
- manipulación de listas,
- evaluación de expresiones aritméticas,
- reproducción de secuencias musicales,
- implementación de algoritmos clásicos como *Torres de Hanoi* y *Piano Sweep*.

La modularidad del intérprete permitió incorporar casos de prueba de diversa complejidad sin alterar la estructura general del sistema.

Capítulo 3

Interfaz Web del Sistema: Algoritmia Studio

La aplicación web **Algoritmia Studio** constituye la interfaz visual del compilador musical desarrollado durante el proyecto. Su función es permitir al usuario escribir código en el lenguaje Algoritmia, compilarlo en tiempo real y visualizar los resultados musicales generados por el intérprete, incluyendo archivos PDF, MIDI, LY y WAV. Toda la interfaz fue diseñada con un enfoque de usabilidad, retroalimentación inmediata y consistencia visual.

3.1. Arquitectura General

La página web está construida en HTML5, CSS y JavaScript, sin frameworks externos. El frontend se comunica con el backend mediante solicitudes `POST` al endpoint `/compile`, enviando:

- El código fuente escrito por el usuario.
- El procedimiento de entrada (Entry Point).
- El estado del modo debug.

La respuesta del servidor incluye la salida de consola generada por el intérprete y los enlaces a los archivos producidos (PDF, MIDI, LY y WAV).

3.2. Editor de Código

El editor es un área `<textarea>` estilizada que simula un entorno de programación. Incluye:

- Resaltado visual mediante tipografías monoespaciadas (JetBrains Mono).
- Un placeholder con un programa de ejemplo escrito en Algoritmia.
- Barras de desplazamiento personalizadas.

El usuario puede escribir, modificar y compilar cualquier procedimiento del lenguaje.

3.2.1. Panel de Controles

Debajo del editor se encuentra un panel con las siguientes funciones:

- **Toggle de Modo Debug**: activa o desactiva la impresión de trazas internas.
- **Cuadro de texto del Entry Point**: permite ingresar el procedimiento inicial.
- **Botón Compilar**: envía el código al backend y deshabilita la interfaz mientras se ejecuta.

3.2.2. Consola Interna

La sección denominada *Terminal Output* muestra:

- Mensajes estándar del intérprete.
- Errores detectados (léxicos, sintácticos o semánticos).
- Trazas del modo debug activado.

Su diseño tipo terminal facilita la lectura de logs.

3.2.3. Zona de Resultados

Tras una compilación exitosa, se habilita un contenedor con los archivos generados:

- Un reproductor de audio para escuchar el archivo WAV generado con TiMidity++.
- Botones de descarga para:
  - Partitura PDF generada por LilyPond.
  - Archivo MIDI.
  - Fuente LilyPond (.ly).
- Vista previa embebida del PDF con la partitura completa.

Esta zona permanece oculta hasta que el intérprete produce resultados válidos.

3.2.4. Estilo y Diseño Visual

La interfaz utiliza un tema oscuro y emplea.

- Paleta basada en tonalidades azul marino y acentos magenta.
- Sombras y bordes suaves para generar profundidad.
- Animaciones sutiles (fade-in y slide-up) para mejorar la experiencia de usuario.
- Iconografía de FontAwesome para reforzar la semántica visual.

El resultado es una interfaz elegante, intuitiva y consistente con el propósito del proyecto.

Capítulo 4

Pruebas y Resultados

Este capítulo presenta el proceso de validación del lenguaje Algoritmia y los programas de prueba incluidos en el directorio `src/temp/`. Las pruebas se realizaron sobre la sintaxis definida en `Algoritmia.g4`, el manejo de listas, procedimientos, estructuras de control y la reproducción musical básica implementada en el intérprete. Las salidas generadas incluyen archivos LilyPond, MIDI y WAV, cuando corresponden.

4.1. Enfoque general

Las pruebas se organizaron en los siguientes ejes:

- Verificación sintáctica mediante la gramática oficial.
- Ejecución de estructuras de control: `if`, `while`.
- Manipulación de listas e índices.
- Llamadas a procedimientos con parámetros.
- Pruebas de reproducción musical mediante las instrucciones `(:)` y `<:>`.

Los resultados se obtuvieron ejecutando los programas desde `run_algoritmia.py`, el cual permite seleccionar un archivo `.alg` y un procedimiento de entrada.

4.2. Pruebas Sintácticas Básicas

Se validaron asignaciones simples, operaciones aritméticas y acceso a listas. Un ejemplo mínimo que pasa correctamente por el analizador sintáctico es:

```
a <- 5
b <- a + 3
<w> b
```

Resultado: El analizador distinguió correctamente asignaciones, expresiones aritméticas e instrucciones de escritura. Errores como paréntesis desbalanceados, operadores mal ubicados o índices ilegales son rechazados durante la fase sintáctica o semántica del intérprete.

4.3. Caso 1: Programa `PROG.ALG`

El archivo `PROG.ALG` incluye un procedimiento principal:

```
Main |:
    <w> "Hello Algoritmia"
    (:) { B C A }
:|
```

Este caso prueba:

- Escritura en consola con `<w>`.
- Reproducción de una lista de valores.
- Manejo de listas literales con la sintaxis real: { elem elem ... }.

Resultado: El intérprete procesa correctamente la lista `{ B C A }` como una secuencia de elementos válidos según la gramática. La instrucción `(:)` dispara la reproducción correspondiente y genera archivos musicales cuando está habilitado el backend.

4.4. Caso 2: Programa `Hanoi`

El archivo `Hanoi.alg` define un procedimiento recursivo:

```
HanoiRec n src dst aux |:
    if n > 0 |:
        HanoiRec (n - 1) src aux dst

        note <- src[#src]
        8< src[#src]
        dst << note
        (:) note

        HanoiRec (n - 1) aux dst src
    :|
:|
```

Y un programa principal:

```
|:
    src <- { C D E F G }
    dst <- {}
    aux <- {}
    HanoiRec #src src dst aux
:|
```

Este archivo prueba:

- Paso de parámetros a procedimientos.

- Acceso a longitud de listas mediante `#lista`.
- Lectura del último elemento con `lista[expr]`.
- Eliminación del último elemento con `8< lista[expr]`.
- Inserción con `lista « expr`.
- Recursión funcional.

Resultado: La recursión y las operaciones sobre listas se evaluaron correctamente. Cada llamada mueve un elemento musical y dispara una reproducción con `(:)`. El número total de reproducciones coincide con el número de movimientos esperados en el algoritmo de Torres de Hanoi para una lista inicial de cinco elementos.

4.5. Caso 3: Programa `PIANO.ALG`

El archivo `PIANO.ALG` implementa un barrido ascendente:

```
nota <- A0
while nota <= C8 |:
    <:> nota
    <w> "Tocando nota numero:" nota
    nota <- nota + 1
:|
```

Este caso prueba:

- Comparaciones entre identificadores de notas.
- Bucle `while` con incremento.
- Reproducción con el operador alternativo `<:>`.
- Escritura de mensajes de progreso.

Resultado: El programa recorre el rango completo definido por las notas `A0` y `C8`. Cada iteración produce una reproducción y un mensaje. El bucle finaliza correctamente cuando la condición deja de cumplirse.

4.6. Observaciones Generales

A partir de los tres programas analizados pueden afirmarse las siguientes conclusiones:

- La gramática soporta listas, procedimientos, recursión, índices y operaciones básicas.
- Las estructuras de control funcionan correctamente bajo bucles y condicionales anidados.

- Las instrucciones `(:)` y `<:>` actúan como puntos de reproducción musical.
- La manipulación de listas empleada en Hanoi es consistente con las reglas definidas.
- Los programas de prueba no presentan ambigüedades sintácticas y se ejecutan de manera estable.

En conjunto, estos resultados demuestran que Algoritmia es capaz de manejar estructuras algorítmicas generales (recursión, bucles, listas) combinadas con reproducción musical, lo que valida su uso como lenguaje de composición algorítmica.

Capítulo 5

Conclusiones

El desarrollo del lenguaje Algoritmia permitió demostrar la viabilidad de construir un lenguaje de dominio específico orientado a la composición algorítmica musical. A partir de una gramática formal definida en ANTLR4 y un intérprete estructurado en Python, fue posible implementar un sistema capaz de ejecutar programas que combinan estructuras clásicas de programación con operaciones musicales simbólicas.

Los resultados obtenidos permiten establecer las siguientes conclusiones principales:

1. **El intérprete cumple con los objetivos fundamentales del lenguaje.** Algoritmia procesa de forma correcta procedimientos, expresiones aritméticas, condicionales, bucles, listas y reproducciones musicales. Los casos evaluados demuestran que la ejecución es consistente y que el flujo de instrucciones se gestiona adecuadamente conforme a la gramática definida.

2. **La gramática y la sintaxis demostraron ser estables y libres de ambigüedad.** La decisión de mantener instrucciones musicales simples, con un único argumento, permitió evitar conflictos en el análisis sintáctico y facilitó la interpretación de listas, expresiones y estructuras de control. Esto se reflejó en un comportamiento predecible incluso en programas extensos o recursivos.

3. **La generación automática de archivos PDF, MIDI y WAV se integró con éxito.** El sistema convierte las instrucciones musicales en archivos `.ly` compatibles con LilyPond, y a partir de ellos genera partituras en PDF, archivos MIDI y audio WAV mediante Timidity++. Este flujo de trabajo permitió validar la representación musical producida por el lenguaje y confirma su utilidad práctica en composición y análisis.

4. **La arquitectura modular facilitó la integración con múltiples interfaces.** La separación entre el núcleo ejecutor en `Executor.py`, los programas de prueba y la interfaz de ejecución por consola permitió adaptar el proyecto a diferentes entornos, incluyendo un frontend web basado en HTML/JS. Esta modularidad abre la posibilidad de seguir ampliando el sistema sin alterar el funcionamiento del intérprete principal.

5. **Los casos integrales confirmaron la robustez del sistema.** Programas como *Piano Sweep*, *Hanoi* y composiciones que combinan listas, recursión y reproducción musical evidenciaron que el intérprete puede manejar secuencias largas, recursividad y manipulación de listas sin pérdida de estabilidad. Esto valida tanto la corrección de la gramática como la consistencia de la ejecución.

En conjunto, los resultados muestran que Algoritmia constituye una implementación funcional, coherente y extensible de un lenguaje para composición algorítmica. Su diseño modular y su integración con herramientas externas de notación musical permiten proyectar futuras extensiones, como soporte para dinámicas, articulaciones, compases, nuevos operadores musicales o estructuras de composición avanzadas. El proyecto representa un aporte académico al vínculo entre lenguajes de programación, teoría musical y generación algorítmica.