UNIVERSIDAD PERUANA CAYETANO HEREDIA

Facultad de Ciencias e Ingeniería - FACIEN

INFORME ACADÉMICO

CURSO: Implementación de Lenguajes de Programación

TÍTULO: Implementación de un doble intérprete de Algoritmia utilizando Python y ANTLR

Autores:

1. Bernal Belisario Brigitte
2. Condori Mamani Nardy Liz
3. Jauregui Bendezu Frank Kevin
4. Morales Alvarado Jesus Anselmo

Docentes:

1. Choquehuayta Nina Wilder
2. Lovon Ramos Percy Wilianson

Fecha de entrega: 03/12/2025

Lima, 2025

Índice general

1. INTRODUCCIÓN

1.1. Contexto del Proyecto

Este proyecto consiste en realizar e implementar un doble intérprete para el leguaje de Algoritmia mediante Python y ANTLR4 como tecnológias principales. Una característica principal de este intérprete radica en su capacidad dual, es decir, no solo ejecuta código Algoritmia, sino que también lo traduce a representaciones musicales, donde el sistema procesa el código fuente y genera simúltaneamente partituras musicales mediante LilyPond y archivos de audio en formato WAV a través de FluidSynth y Timidity++.

El proyecto surge de la necesidad de implementar todos los conocimiento adquiridos dentro del curso, además, de crear una herramienta educativa innovadora que tenga las opciones de visualizar y escuchar la ejecución de algoritmos, transformando conceptos abstractos de programación en experiencias sensoriales auditivas. Esta implementación nos facilita el aprendizaje de programación con distintos estilos de aprendizaje, particularmente con inclinaciones musicales o auditivas.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo principal del proyecto es desarrollar un intérprete doble que tenga la capacidad de traducir código de lenguaje Algoritmia a representaciones musicales y auditivas, integrando procesamiento de lenguaje natural con generación de contenido multimedia.

1.2.2. Objetivos Específicos

- Diseñar y definir formalmente la grámatica del lenguaje Algoritmia utilizando ANTLR4.
- Implementar el analizador léxico y sintáctico para el procesamiento del código fuente.
- Desarrollar el motor de transformación que convierta estructuras algorítmicas en notación musical.
- Integrar el sistema con LilyPond para la generación automática de partituras.
- Implementar la conversión a audio mediante FluidSynth y Timidity++ para producir archivos en formato WAV.

- Validar el funcionamiento integral del intérprete mediante casos de prueba exhaustivos.

1.3. Alcance y Limitaciones

El proyecto se enfoca en la interpretación de estructuras algorítmicas básicas y su traducción a elementos musicales. Pero aún siendo un proyecto con una estructura básica en algoritmos, cuenta con una serie de alcances y limitaciones que son las siguientes:

1.3.1. Alcances:

- Soporte para estructuras de control fundamentales (secuencias, condicionales, bucles).
- Traducción a variables y operaciones a elementos musicales (notas, ritmos, armonías).
- Generación de partituras en formato PDF mediante LilyPond.
- Producción de archivos de audio en formato WAV.
- Implementación en Python 3 utilizando ANTLR4 para el análisis sintáctico.

1.3.2. Limitaciones:

- El lenguaje de Algoritmia cubre un subconjunto limitado de construcciones de programación.
- La traducción musical se restringe a parámetros melódicos y rítmicos básicos.
- No incluye soporte para programación orientada a objetos avanzada.
- La generación de armónias complejas está fuera del alcance inicial.
- El rendimiento con algoritmos computacionalmente intensivos puede ser limitado.

2. MARCO TEÓRICO

2.1. Lenguajes de Programación y Interpretación

Los lenguajes de programación son sistemas de comunicación formal que están diseñados para expresar computaciones y algoritmos que pueden ser ejecutados por una computadora. Están compuestos por un conjunto de reglas sintácticas y semánticas que definen su estructura y significado, y estos pueden clasificarse en múltiples categorías según su nivel de abstracción, paradigma de programación y métodos de ejecución.

2.1.1. Interpretación vs. Compilación

La interpretación y compilación representan dos enfoques fundamentales para la ejecución de código, donde un intérprete procesa y ejecuta el código fuente directamente, línea por línea, traduciéndolo a instrucciones de máquina en tiempo de ejecucción, en constraste, un compilador traduce todo el código fuente a código máquina de antemano, lo que genera un archivo ejecutable independiente.

El enfoque interpretado ofrece ventajas en térmios de portabilidad y depuración, ya que el mismo código puede ejecutarse en diferentes plataformas sin recompilación, y los errores pueden identificarse con mayor facilidad y precisión durante la ejecución. Sin embargo, generalmente presenta un menor rendimiento en comparación con el código compilado.

2.1.2. Árboles de Sintaxis Abstracta (AST)

Los Árboles de Sintaxis Abstracta son estucturas de datos fundamentales en el procesamiento de lenguajes, ya que representan la estructura jerárquica del código fuente, eliminando detalles sintácticos superficiales y preservando únicamente el significado del programa. El AST sirve como interfaz entre el análisis sintáctico y la generación de código o en este caso la traducción a representaciones musicales.

2.2. ANTLR4

ANother Tool for Language Recognition (ANTLR) es un framework poderoso para la construcción de procesadores de lenguaje. Utiliza gramáticas formales definidas en notación EBNF (Extended Backus-Naur Form) para generar automáticamente analizadores léxicos (lexers) y sintácticos (parsers). La capacidad de ANTLR4 para generar visitors y listeners facilita la implementación de transformaciones y

traducciones sobre el AST, haciendo posible la conversión de código algorítmico a representaciones musicales.

2.3. LilyPond

LilyPond es un sistema de edición de partituras musicales en código abierto que sigue el paradigma de`engraving` automatizado, donde la música se describe mediante un lenguaje de marcado textual y el programa genera automáticamente partituras en formatos como PDF y MIDI. A diferencia de los editores Wysiwyg, LilyPond prioriza la corrección musical y la estética profesional mediante algoritmos que toman decisiones tipográficas basadas en prácticas tradicionales de grabado musical, funcionando en este proyecto como el motor que transforma las estructuras algorítmicas procesadas por ANTLR4 en notación musical precisa para su ejecución sonora.

2.4. Timidity++

Timidity++ es un reproductor de archivos MIDI de código abierto que actúa como sintetizador de software, donde utiliza bancos de sonido (soundfonts) para convertir archivos MIDI en formatos de audio digital como WAV. Su capacidad de procesamiento por lotes y su arquitectura de síntesis por tablas de onda lo convierten en el componente ideal para transformar las partituras generadas por LilyPond en experiencias auditivas tangibles, completando así el pipeline que permite al intérprete de Algoritmia no solo ejecutar código sino también producir resultados sonoros que reflejen de manera visual y auditivamente el comportamiento de los algoritmos implementados.

2.5. FluidSynth

FluidSynth al igual que Timidity++ es un sintetizador de audio en tiempo real de código abierto que implementa el estándar SoundFont 2 para la síntesis de audio basada en samples. En el contexto del proyecto, FluidSynth representa una alternativa tecnológica avanzada a Timidity++ para la conversión de representaciones musicales simbólicas (MIDI) a señales de audio digital (WAV).

2.6. Aplicación en el Proyecto

En este proyecto específico, la interpretación dual implica no solo la ejecución tradicional del código Algoritmia, sino también el mapeo de construcciones algorítmicas a conceptos musicales. Las variables pueden representar notas musicales, las estructuras de control pueden transformarse en patrones rítmicos, y las funciones pueden corresponder a motivos melódicos, creando así una correspondencia sistemática entre el dominio algorítmico y el musical.

3. DISEÑO E IMPLEMENTACIÓN

3.1. Arquitectura del Sistema

Nuestro proyecto esta compuesto por tres módulos principales:

- Intérprete Algoritmia, que está desarrollado en Python con ANTLR4, ya que se encargará del análisis léxico y sintáctico del código fuente.
- Generador de partituras, donde el árbol de sintaxis abstracta (AST) se convierte en código LilyPond para la generación de partituras en formato PDF.
- Sintetizador de audio, que utiliza FluidSynth y Timidity++ para convertir el archivo MIDI generado desde Lilypond en un archivo de audio WAV.

Estos módulos se comunican de forma secuencial, haciendo que el flujo del proyecto sea continuo siguiendo el siguiente diagrama:

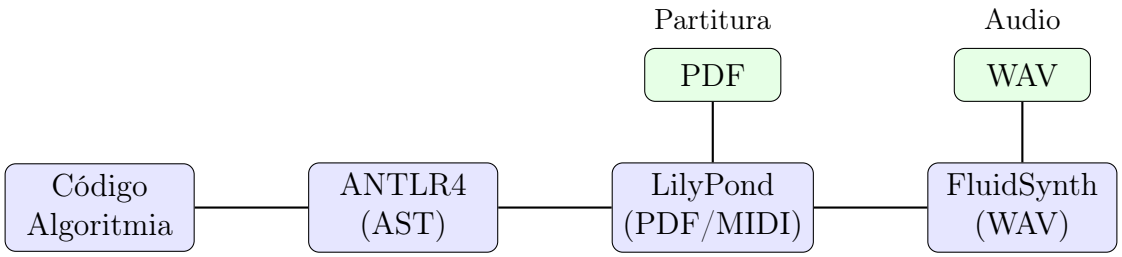

Figura 3.1: Flujo de procesamiento del intérprete doble

3.2. Gramática del Lenguaje Algoritmia

La gramática fue diseñada específicamente para el doble intérprete de Algoritmia utilizando ANTLR4, ya que es aquí donde se define la estructura sintáctica y léxica que nos permitiran traducir estructuras algorítmicas a programas musicales, manteniendo un control de flujo en el sistema. A continuación, se describirá paso a paso las partes del funcionamiento de la gramática en cuestión.

Listing 3.1: Regla Principal (Punto de inicio del Lenguaje)

```
1 grammar Algoritmia;
2
3 root : procedure+ EOF ;
```

- Está es la regla principal de la gramática, ya que establece que todo programa escrito en Algoritmia debe estar compuesto obligatoriamente por uno o más procedimientos (`procedure+`) y debe finalizar con el fin de archivo (`EOF`), lo que significa que no se permite la ejecución de intrucciones sueltas fuera de un procedimiento, garantizando así una estructura de manera organizada y modular del programa.

Listing 3.2: Definición de Procedimientos

```
procedure : ID params? block ;
params : ID+ ;
block : '|:' statement* ':|' ;
```

- La regla `procedure : ID params? block;` define un procedimiento que está compuesto por tres partes, el identificador `ID` que representa el nombre del procedimiento, `params?` los parámetros opcionales y `block` los bloques de instrucciones. Los parámetros `params : ID+;` permiten el ingreso de valores externos al procedimiento que están definidos como uno o más identificadores, asimismo, el bloque (`block : '|:' statement* ':|';`) encapsula el conjunto de instrucciones que se ejecutarán, delimitadas por símbolos de inicio (`|:`) a fin (`:|`), donde el `statement*` contiene cero o más instrucciones para luego encapsularlas dentro de un procedimiento.

Listing 3.3: Instrucciones del Lenguaje (`Statements`)

```
statement
    : assignment
    | read
    | write
    | play
    | conditional
    | iteration
    | procCall
    | listAppend
    | listCut
```

- La regla `statement` agrupa todas las instrucciones permitidas dentro del lenguaje, ya que es aquí donde se definirán todas las acciones que el lenguaje permitirá ejecutar, incluyendo asignaciones, lectura de datos, escritura en pantalla, reproducción musical, estructuras condicionales, estructuras repetitivas, llamadas a procedimientos y manejo de listas (añadir/eliminar), lo que permite que el lenguaje tenga capacidades completas para el control de flujo, entrada, salida y manipulación de datos.

Listing 3.4: Asignación de Variables

```
assignment : ID '<-' expr ;
```

- Está instrucción permite almacenar el resultado de una expresión dentro de una variable utilizando el operador<-, lo cual facilita el manejo de datos dentro del programa, permitiendo guardar valores numéricos, resultados de opraciones, notas musicales o resultados de expresiones complejas.

Listing 3.5: Lectura de Datos

```
read : '<?>' ID ;
```

- Asimismo, está regla permite capturar datos ingresados por el usuario para luego almacenarlos en una variable determinada, representando así el mecanismo de entrada del lenguaje, ya que es fundamental para la interacción entre el programa y el usuario durante la ejecución.

Listing 3.6: Escritura de Datos

```
write : '<w>' writeParam+ ;
writeParam : expr | STRING ;
```

- En este caso, está regla permite mostrar información en pantalla, ya sea en forma de expresiones evaluadas o cadenas de texto, lo que constituye el mecanismo de salida (<w>) del lenguaje que es esencial para la comunicación de resultados al usuario.

Listing 3.7: Reproducción Musical

```
play : '(:)' expr ;
```

- Está instrucción permite la reproducción de sonidos o notas musicales a partir de una expresión, integrando así el componente musical del lenguaje Algoritmia, ya que es una característica propia del lenguaje como tal.

Listing 3.8: Estructuras Condicionales

```
conditional : 'if' expr block ('else' block)? ;
```

- El bloque de código define la estructura condicional (if/else) del lenguaje, permitiendo ejecutar un bloque de instrucciones si una condición es verdadera y, de manera opcional, ejecutar un bloque alternativo si la condición resulta ser falsa, controlando así la toma de decisiones dentro del programa.

Listing 3.9: Estructuras Iterativas (Bucles)

```
iteration : 'while' expr block ;
```

- Asimismo, está regla permite la ejecución repetitiva (while) de un conjunto de instrucciones mientras una condición se mantenga verdadera, facilitando asi la automatización de procesos repetitivos y el control dinámico del flujo en el programa.

Listing 3.10: Llamadas a Procedimientos

```
procCall : ID expr* ;
```

- Por otro lado, está regla permite invocar un procedimiento previamente definido, pasando de cero o más expresiones como parámetros, lo que favorece la reutilización de código y una programación más ordenada y modular.

Listing 3.11: Manejo de Listas

```
# Agregar valores
listAppend : ID '<<' expr ;

# Eliminar valores
listAppend : ID '<<' expr ;
```

- Estas dos reglas permiten respectivamente agregar elementos a una lista y eliminar elementos en una posición determinada, incorporando así estructuras de datos dinámicas que facilitan el almacenamineto y manipulación de múltiples valores.

Listing 3.12: Expresiones del Lenguaje

```
expr
    : expr ('*'|'/'|'%') expr
    | expr ('+'|'-') expr
    | expr ('='|'/='|'<'|'>'|'<='|'>=') expr
    | '(' expr ')'
    | ID '[' expr ']'
    | '#' ID
    | '{' expr* '}'
    | ID
    | NUM
    | NOTE
```

- La regla `expr` define todas las operaciones válidas dentro del lenguaje, incluyendo operaciones aritméticas, comparaciones relacionales, agrupaciones con paréntesis, acceso a listas, obtención del tamaño de una lista, creación de listas, variables, números y notas musicales, constituyéndose como el núcleo del procesamiento lógico y matemático del lenguaje.

Listing 3.13: Tokens del Lenguaje

```
ID : [A-Z][a-zA-Z0-9_]* | [a-z][a-zA-Z0-9_]* ; # letras
NOTE : [A-G][0-8]? ; # A-G con octava
NUM : [0-9]+ ; #numeros
STRING : '"' (~["\r\n])* '"' ; # texto entre comillas
COMMENT : '###' .*? '###' -> skip ; # comentarios
WS : [ \t\r\n]+ -> skip ; # espacios
```

- Los tokens `ID, NOTE, NUM, STRING, COMMENT y WS` definen los elementos léxicos fundamentales del lenguaje, permitiendo identificar correctamente las variables, notas musicales, valores numéricos, cadenas de texto, comentarios y espacios en blanco, donde estos dos últimos son ignorados durante el proceso de análisis léxico.

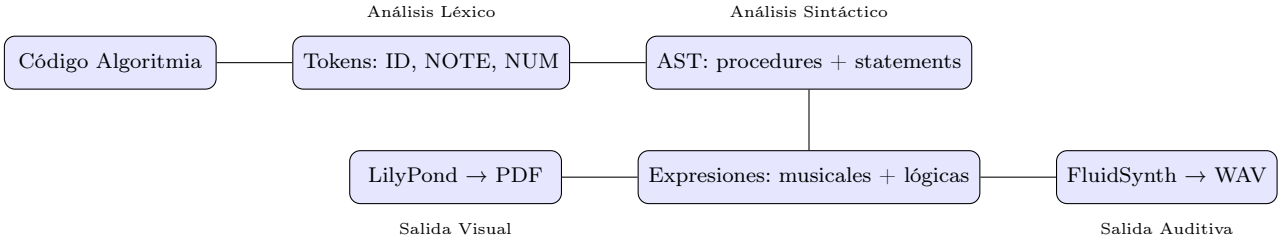

Figura 3.2: Flujo del intérprete basado en la gramática ANTLR4

Esta gramática es el corazón del doble intérprete, permitiendo que un mismo algoritmo genere representaciones musicales tanto visuales como sonoras, además de que cada punto mencionado en la gramática `.g4` se adaptará dentro del archivo `algoritmia.py` para su ejecución.

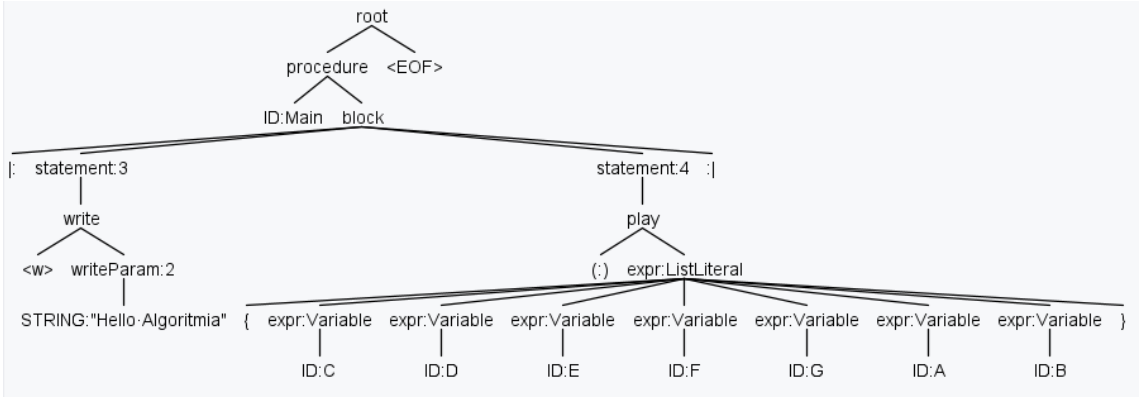

Figura 3.3: Ejemplo de una Melodía Simple (Hello Algoritmia) en un árbol de análisis sintáctico (AST)

3.3. Implementación en Python

El núcleo del intérprete Algoritmia está implementado en Python 3, utilizando ANTLR4 para el análisis léxico y sintáctico lo que asegura compatibilidad con la mayoría de las librerias estándar, además de que permite utilizar caracteríticas de tipado (`typing`) y manejo de excepciones modernas. El intérprete principal se encuentra en `algoritmia.py`, donde se define la clase `AlgoritmiaInterpreter`, que es el responsable de recorrer el árbol sintáctico generado por ANTLR juntamente con los archivos de `AlgoritmiaLexer` y `AlgoritmiaParser`. Cabe destacar que la estructura de ejecución está basada en procedimientos (`procedures`) y manejos de stacks de llamadas (`call_stack`) para que así soporte ámbitos locales de diferentes variables.

Figura 3.4: Archivos generados por ANTLR 4 a partir de la gramática `Algoritmia.g4`

3.3.1. El Analizador Léxico: AlgoritmiaLexer.py

Esté archivo es el analizador léxico del lenguaje Algoritmia y se encarga de reconocer todos los símbolos, palabras reservadas, operadores, números, textos, notas musicales y comentarios del código fuente, transformándolos en tokens que luego son utilizados por el analizador sintáctico para validar la estructura del programa y son generados de manera automática por ANTLR, para ello, se hará la explicación de los puntos más importantes que representan al archivo en cuestión.

Listing 3.14: Definición de la clase `AlgoritmiaLexer` y creación de los automátas DFA

```
class AlgoritmiaLexer(Lexer):

    atn = ATNDeserializer().deserialize(serializedATN())

    decisionsToDFA = [ DFA(ds, i) for i, ds in enumerate(atn.
        decisionToState) ]
```

- Esta es la base principal del analizador léxico y es el encargado de leer el código fuente carácter por carácter para luego convertirlo en tokens y al igual que el analizador sintáctico tiene la función de `serializedATN()` que representa un automáta real de transición para reconocer correctamente los tokens generados, además, de construir los automátas deterministas (DFA) que permiten que el reconocimiento de tokens sea rápido y preciso, los cuales luego son enviados al parser, permitiendo así que el lenguaje Algoritmia sea procesado por etapas correctamente.

Listing 3.15: Definición de tokens internos T_0 a T_28

```
T__0 = 1
T__1 = 2
T__2 = 3
...
T__28 = 29
```

- Estos tokens representan todos los símbolos y palabras reservadas del lenguaje, como `|:`, `:|`, `<-`, `if`, `else`, `while`, operadores matemáticos, comparadores y delimitadores, los cuales son fudamentales para darle estructura al lenguaje.

Listing 3.16: Tokens principales del lenguaje

```
ID = 30
NOTE = 31
NUM = 32
STRING = 33
COMMENT = 34
WS = 35
```

- En este caso, estos tokens representan los elementos básicos del lenguaje, donde el `ID` identifica las diversas variables y nombres, `NOTE` representa las nota musicales, `NUM` los valores numéricos, STRING el texto entre comillas, `COMMENT` los comentarios y `WS` los espacios en blanco que son ignorados durante el análisis y estos mismos aparecen dentro de `AlgortimiaParser.py` para hacer un reconocimiento mas robusto.

Listing 3.17: Nombres de canales y modos

```
channelNames = [ u"DEFAULT_TOKEN_CHANNEL", u"HIDDEN" ]
modeNames = [ "DEFAULT_MODE" ]
```

- Este bloque define cómo se clasificarán los tokens durante el análisis léxico, ya que permite que algunos elementos como los espacios en blanco o comentarios se envíen al canal oculto (`HIDDEN`) para que no interfieran en el análisis sintáctico.

Listing 3.18: Definición de símbolos literales

```
literalNames = [ "<INVALID>",
    "'|:'", "':|'", "'<-'", "'<?>'", "'<w>'", "'(:)'", "'if'", "'else'",
    "'while'", "'<<'", "'8<'", "'['", "']'", "'*'", "'/'", "'%'",
    "'+'", "'-'", "'='", "'/='", "'<'", "'>'", "'<='", "'>='", "'('",
    "')'", "'#'", "'{'", "'}'" ]
```

- Este bloque define todos los símbolos exactos que el lexer reconocerá en el código fuente, incluyendo delimitadores de bloque, operadores, comparadores, estructuras de control y símbolos especiales del lenguaje Algoritmia.

Listing 3.19: Nombres simbólicos de los tokens y reglas léxicas

```
symbolicNames = [ "<INVALID>",
    "ID", "NOTE", "NUM", "STRING", "COMMENT", "WS" ]

ruleNames = [ "T__0", "T__1", "T__2", ..., "ID", "NOTE", "NUM", "STRING", "
    COMMENT", "WS" ]
```

- Es aquí donde se definen los nombres simbólicos que representan los tipos de tokens más importantes del lenguaje y las reglas léxicas que indicarán como deben reconocerse cada tipo de token, permitiendo así que el parser pueda identificar correctamente estas variables, ya sean notas, números, cadenas de texto, comentarios o espacios en blanco.

3.3.2. El Analizador Sintáctico: AlgoritmiaParser.py

El archivo `AlgoritmiaParser.py`, al igual que el archivo Lexer es generado automáticamente por ANTLR, donde constituye el núcleo del analizador sintáctico. Este módulo implementa todas las reglas gramaticales definidas en `Algoritmia.g4` como métodos Python, permitiendo validar la estructura jerárquica de los tokens producidos por el lexer, además, de permitir ejecutar instrucciones con un `Visitor` o `Listener`. A continuación, mencionaremos las partes más importantes de todo el archivo, dado que se implementan todas las estructuras del lenguaje gramatical a Python como asignaciones, condicionales, bucles, expresiones, listas y procedimientos.

Listing 3.20: Función `serializedATN()`

```
def serializedATN():
    return [4,1,35,158,2,0,7,0,...]
```

- Está es una función que contiene la versión codificada de toda la gramática en forma de una secuencia de números que representan un automáta de transición (ATN), el cual es utilizado internamente por ANTLR para reconocer las estructuras del lenguaje como tal, por lo que está sección no es legible para los usuarios, y del mismo modo no debe modificarse manualmente, ya que permite que el parcer funcione de manera eficiente y automática, además, de que también se encuentra en el archivo Lexer.

Listing 3.21: Clase principal `AlgoritmiaParser` y reglas sintácticas

```
class AlgoritmiaParser ( Parser ):

    grammarFileName = "Algoritmia.g4"

    atn = ATNDeserializer().deserialize(serializedATN())

# Tokens del lenguaje (terminales)
literalNames = [ "'|:'", "':|'", "'<-'", "'<?>'", "'<w>'", ... ]

# reglas sintacticas
RULE_root = 0
RULE_procedure = 1
RULE_params = 2
RULE_block = 3
RULE_statement = 4
...
RULE_expr = 15
```

- En esta ocación, la clase `AlgoritmiaParser` es el núcleo del analizador sintáctico, dado que contiene toda la lógica necesaria para reconocer cada construcción de lenguaje gramatical, ya que en ella se definen los tokens, las reglas sintácticas y los métodos que validarán la estructura correcta de los programas, permitiendo de esa manera convertir el texto fuente en un árbol sintáctico que represente su estructura lógica.

Listing 3.22: Reglas de asiganción

```
# ejemplo de una regla gramatical
def assignment(self):
    self.match(AlgoritmiaParser.ID)
    self.match(AlgoritmiaParser.T__2)
    self.expr(0)

# Esto representa esta regla:
assignment: ID '<-' expr ;
```

- La reglas de asignación nos ayudan a definir de cómo se realizará la asignación de valores en el lenguaje, donde indica que primero debe aparecer un identificador, seguido del operador `<-` y finalmente una expresión válida, permitiendo así crear instrucciones como `x <- 5`, lo cual es primordial para alamacenar valores en variables y realizar operaciones dentro del programa.

Listing 3.23: Bloque de instrucciones

```
def block(self):
    self.match(AlgoritmiaParser.T__0) # |:
    while statement():
        ...
    self.match(AlgoritmiaParser.T__1) # :|

# representa:
|:
   instrucciones
:|
```

- Los bloques de instrucciones nos ayudarán a establecer la forma en que se debe agrupar las diversas sentencias dentro del lenguaje mediante los símbolos `|:` para abrir el bloque y `:|` para cerrarlo, permitiendo que las múltiples instrucciones se ejecuten de forma ordenada dentro de estructuras como condicionales o bucles, lo que garantizará una correcta organización del flujo dentro del programa.

Listing 3.24: Estructura condicional `if - else`

```
def conditional(self):
    self.match(AlgoritmiaParser.T__6) # if
    self.expr(0)
    self.block()
```

```
    if self.match(AlgoritmiaParser.T__7): # else
        self.block()

# representa
if condicion
|:
   instrucciones
:|
else
|:
   instrucciones
:|
```

- Las estructuras condicionales nos permitiran ejecutar un bloque de instrucciones dependiendo el resultado de una expresión lógica, donde si la condición se cumple se ejecuta el bloque principal y, en caso contrario, se ejecutará el bloque alternativo `else`, permitiendo de esa manera que los programas tomen decisiones dinámicas durante la ejecución.

Listing 3.25: Estructura repetitiva `while`

```
def iteration(self):
    self.match(AlgoritmiaParser.T__8) # while
    self.expr(0)
    self.block()

# representa
while condicion
|:
   instrucciones
:|
```

- Las estructuras repetitivas nos permitiran ejecutar un bloque de instrucciones de manera iterativa mientras una condición sea verdadera, lo que hacer posible automatizar procesos que requieren repeticiones continuas, como recorrer listas, generar secuencias o controlar procesos que se repetiran dentro del programa Algoritmia.

3.3.3. Módulo Principal: algoritmia.py

El archivo `algoritmia.py`, es el núcleo del intérprete del lenguaje algoritmia, ya que controla toda la ejecución de procedimientos, la gestión de variables, la entrada de datos del usuario y el manejo de la partitura musical, permitiendo que los programas escritos en este lenguaje puedan ejecutarse correctamente a partir del árbol sintáctico generado por ANTLR. Por ende, cabe mencionar que los bloques de códigos críticos incluyen la función de `execute_procedure` que ejecuta los procedimientos definidos en el código fuente, y los métodos `visitAssignment`, `visitRead` y `visitWrite`, ayudan a gestionar las operaciones básicas de variables y entrada/-salida, además de que son esenciales para el flujo de cualquier programa escrito en algoritmia.

Listing 3.26: Clase Principal `AlgoritmiaInterpreter`

```
class AlgoritmiaInterpreter(AlgoritmiaVisitor):
    def __init__(self):
        self.procedures: Dict[str, Procedure] = {}
        self.call_stack: List[Dict[str, Any]] = []
        self.score: List[int] = [] # Partitura como lista de enteros
        self.current_scope: Dict[str, Any] = {}
```

- Este bloque de código define la clase del intérprete de Algoritmia, la cual hereda del `AlgoritmiaVisitor` generado por ANTLR, dado que nos permite recorrer el árbol sintáctico para ejecutar el programa, donde `procedures` almacena los procedimientos definidos, `call_back` maneja la pila de llamadas para controlar los bloques, `score` representa la partitura musical como una lista de valores enteros y `current_scope` mantiene las variables activas del programa durante la ejecución.

Listing 3.27: Manejo de Procedimientos

```
def execute_procedure(self, name: str, args: List[Any]):
    if name not in self.procedures:
        raise RuntimeError(f"Procedimiento '{name}' no encontrado")

    # Crear nuevo ambito local
    local_scope = {}
    for param, arg in zip(proc.params, args):
        local_scope[param] = arg.copy() if isinstance(arg, list) else arg
```

- Este bloque implementa el mecanismo de ejecución de procedimientos dentro del intéprete, donde primero hace la verificación si el procedimiento existe, para luego crear un nuevo ámbito local que evite que las variables interfieran con otros contextos, y finalmente asignar los argumentos a los parámetros del procedimiento en cuestión, copiando las listas para evitar modificaciones accidentales, lo que garantiza una ejecución segura y aislada de cada llamada realizada.

Listing 3.28: Asignación y lectura de variables

```
def visitAssignment(self, ctx):
    self.current_scope[var_name] = value

def visitRead(self, ctx):
    value = int(input())
```

- Aquí es donde se define cómo el intérprete manejará la asignación de variables durante la ejecución del programa, almacenando el valor calculado dentro del ámbito actual (`current_scope`), lo que permitirá que las variables se creen, actualicen y sean utilizadas correctamente en operaciones posteriores. Asimismo, el intérprete capturará el valor ingresado por el usuario, para luego convertirlo en un valor entero, seguidamente utilizarlo dentro del programa para hacer posible que los algoritmos trabajen con datos dinámicos proporcionados en tiempo real.

Sistema de Tipos, Variables y Características implementadas:

El lenguaje de Algoritmia implementa un sistema de tipos de datos simples pero robustos, dado que maneja variables enteras, listas, con un mapeo directo de notas musicales a valores numéricos predefinidos, además, de que nos permite hacer la integración entre la programación y la música, asimismo, gestiona ámbitos locales en procedimientos y evita el aliasing mediante una copia de listas. El lenguaje incluye asignación de variables con `<-`, estructuras de control if/while, procedimientos parametrizados, operaciones de lista (#, `<<`), y entrada/salida estándar (`<w>`, `<?>`), proporcionando así un entorno completo para la programación algorítmica y musical.

3.4. Integración con LilyPond

La generación de partituras de realiza a través de LilyPond mediante la función `generate_lilypond` en `Algoritmia.py`, donde permite convertir la lista de notas musicales almacendas en `self.score` en un archivo `.alg` con notación musical formal. Asimismo, la función `generate_lilypond` transforma cada nota entera en notación LilyPond, considerando octavas y duración (4 para negras). Esta integración nos asegura que cualquier código Algoritmia que reproduzca notas mediante la instrucción `<w>`, o bloques de reproducción (`:`) genere automáticamente partituras en PDF (`base_name.pdf`).

Listing 3.29: Convertir lista de notas a notación LilyPond

```
def generate_lilypond(score: List[int], output_filename: str):
# Generar contenido LilyPond
    lilypond_notes = ' '.join([note_to_lilypond(note) for note in score])
# Escribir archivo .ly
    ly_filename = output_filename.replace('.alg', '.ly')

# Compilar a PDF
subprocess.run(['lilypond', '-o', base_name, ly_file], check=True)
```

- Este bloque de código nos asegura que las secuencias musicales creadas con instrucciones de reproducción `visitPlay` o listas de notas generen automáticamente partituras visuales, integrando la representación musical dentro de la ejecución del intéprete.

Listing 3.30: Generación de Archivo LilyPond

```
lilypond_content = f'''
\\version "2.20.0"
\\score {{
    \\new Staff {{
        \\clef treble
        {{ {lilypond_notes} }}
    }}
    \\layout {{ }}
    \\midi {{ }}
```

```
}}
'''
```

3.5. Generación de Audio con Timidity++ y FluidSynth

La conversión de la partitura a audio se gestiona en `web_app.py` mediante la función `convert_midi_to_wav`, que gestiona la conversión, buscando automaticamente soundfonts (`.sf2`) para FluidSynth y configurando la ejecución de Timidity con parámetros adecuados para WAV. En resumen, este bloque permite transformar archivos MIDI generados desde LilyPond en WAV reproducibles, controlando errores mediante `subprocess.run` con bloques `try_catch` que capturan parámetros de timeout y manejo de excepciones. La lógica incluye realizar una verificación de existencia sobre los archivos que se va a generar, asegurando que la aplicación web pueda entregar un audio reproducible sin intervención manual del usuario.

Listing 3.31: Conversión con Timidity++ y FluidSynth

```
# Conversion con FluidSynth
subprocess.run([FLUIDSYNTH_PATH, '-ni', soundfont, midi_path, '-F',
    wav_path, '-r', '44100'], check=True)

# Conversion con Timidity
subprocess.run([TIMIDITY_PATH, midi_path, '-Ow', '-o', wav_path], check=
    True)
```

Listing 3.32: Configuración de SoundFonts:

```
soundfont_paths = [
    r'C:\soundfonts\FluidR3_GM.sf2',
    '/usr/share/sounds/sf2/FluidR3_GM.sf2',
    '/usr/share/soundfonts/default.sf2',
    'C:\\soundfonts\\default.sf2',
]
```

Cabe especificar que cuando los archivos SoundFonts no están disponibles, el sistema implementa una estrategia de fallback robusta donde FluidSynth intenta localizar los soundfonts en rutas predefinidas y, si no los encuentra, registra el fallo y cede el control a Timidity++, el cual también tratará de generar el audio, dado que ambos dependen de soundfonts, en parte se dice que timidity produce su propio audio, pero no genera sonido desde cero, por lo que necesita archivos que definan el sonido del instrumento, y dentro de Timidity el archivo de configuración principal siempre suele ser **timidity.cfg**.

Entonces si soundfonts no está presente al momento de la coversión presenta fallas, lo que produce que no se genere el audio con FluidSynth y Timidity++, dado que SoundFonts son archivos que contienen muestras de audio de instrumentos reales (piano, cuerdas, metales, etc.) que ambos utilizan para sintetizar sonidos de alta calidad a partir de archivos MIDI. En resumen, nuestro sistema es tolerante a fallos, ya que sin bancos de sonido instalados correctamente, es más probable que no se genere el archivo WAV, sin embargo, el usuario siempre tendrá el archivo MIDI y la partitura generada en PDF.

4. RESULTADOS Y PRUEBAS

En esta sección se presentará los resultados obtenidos tras el desarrollo del intérprete para el lenguaje Algoritmia, construido empleando ANTLR como generador de analizadores, PYTHON como lenguaje anfitrión, juntamente con las herramientas LilyPond, Timidity++ y FluidSynth para la generación de partituras y audio. Como se menciono desde un inicio el objetivo principal fue permitir que un archivo escrito en Algoritmia pueda ser analizado, interpretado, traducido y finalmente ejecutado en forma musical, produciendo tanto la notación tradicional como el sonido correpondiente. Por ende, se mostrarán las evidencias respectivas de los casos empleados con exito en el desarrollo de nuestro intérprete musical.

4.1. Casos de Prueba

Ejemplo 1: Melodía Simple (Hello Algoritmia)

```
Main |:
    <w> "Hola Algoritmia"
    (:) {C D E F G A B C}
:|
```

- El programa muestra el mensaje **‘‘Hola Algoritmia’’** y reproduce la secuencia musical `{C D E F G A B C}` correspondiente a una escala completa de Do Mayor, generando exitosamente la partitura en PDF y el audio en formato WAV con la melodía especificada, demostrando la integración básica entre salida de texto y generación musical en el lenguaje Algoritmia.

Ejemplo 2: Operaciones con Variables y Listas

```
Main |:
    notas <- {C D E F G}
    <w> "Notas iniciales:" notas
    <w> "Longitud:" #notas

    ### Implementar notas ###
    notas << A
    notas << B
    <w> "Notas finales:" notas

    ### Reproducir todas ###
    (:) notas
:|
```

Resultado de la Ejecución:

```
Notas iniciales: [28 29 30 31 32]
Longitud: 5
Notas finales: [28 29 30 31 32 33 34]
Partitura generada: static\outputs\temp_20251121_030841_491064.pdf
Audio generado: static\outputs\temp_20251121_030841_491064.wav
```

- Este ejemplo nos demuestra el manejo de variables y operaciones con listas en Algoritmia, inicializando con una lista musical `[28 29 30 31 32]` correspondientes a las notas `{C D E F G}`, asimismo, muestra su contenido y longitud inicial (`longitud = 5`), luego añade notas A y B mediante el operador `<<`, y finalmente reproduce la escala completa `{C D E F G A B}`, donde genera los archivos PDF, MIDI y WAV con la secuencia musical resultante, evidenciando la integración entre manipulación de datos y síntesis musical.

Ejemplo 3: Escaladas musicales

```
Main |:
    <w> "Escala de Do Mayor (ascendente)"
    i <- 0
    while i < 8 |:
        nota <- C + i
        (:) nota
        i <- i + 1
    :|

    <w> "Escala completada"
:|
```

Resultado de la Ejecución:

```
Escala de Do Mayor (ascendente)
Escala completada
Partitura generada: static\outputs\temp_20251121_032108_104709.pdf
Audio generado: static\outputs\temp_20251121_032108_104709.wav
```

- El programa genera una escala de `Do` mayor ascendente mediante un buble `while` que itera 8 veces, calculando cada nota como (`C + i`) incrementando desde `Do` hasta `Do` una octaba mas arriba, donde reproduce cada nota secuencialmente, y al completar el ciclo genera exitosamente la partitura PDF como el archivo de audio WAV con la escala completa, demostrando la capacidad de generar estructuras musicales mediante lógica iterativa.

Ejemplo 4: Condicionales

```
Main |:
    x <- 15
    if x > 10 |:
        <w> "El numero es mayor que 10"
        (:) {C E G C}
    :| else |:
```

```
        <w> "El numero es menor o igual a 10"
        (:) {C D E F}
    :|
:|
```

Resultado de la Ejecución:

```
El numero es mayor que 10
Partitura generada: static\outputs\temp_20251121_032553_199862.pdf
Audio generado: static\outputs\temp_20251121_032553_199862.wav
```

- En este caso el programa asigna el valor 15 a la variable `x`, luego evalúa la condición `x >10` que resulta verdadera, seguidamente ejecuta la rama `if` mostrando el mensaje `‘‘El número es mayor que 10’’` y reproduce el acorde `{C E G C}`, generando exitosamente la partitura como el archivo de audio con la progresión armónica correspondiente a la condición evaluada, demostrando así el control de flujo condicional.

4.2. Evaluación del Rendimiento

La evaluación de rendimiento del intérprete de Algoritmia desarrallado constituye una etapa fundamental para determinar la eficiencia y fiabilidad del sistema construido, dado que el proyecto integra múltiples componentes ya mencionados (Análisis léxico y sintáctico) basados en ANTLR, Python, etc. Por lo que es necesario analizar de manera individual y conjunta el comportamiento de cada uno de estos módulos.

El objetivo de esta sección es medir el desempeño de nuestro sistema bajo distintas condiciones de uso, identificando los tiempos de procesamiento, posibles cuellos de botella y la eficiencia general del flujo completo: desde la lectura de un programa en Algoritmia hasta la producción del archivo de audio final y el tamaño del documento generado. Esta evaluación nos permite validar que el intérprete no solo cumpla con las especificaciones funcionales, sino que también mantenga un rendimiento adecuado al manejar algoritmos musicales de distinta complejidad.

Cuadro 4.1: Métricas de Tiempo de Ejecución

Caso de Prueba	**Tiempo (s)**	**Tam. PDF (KB)**	**Tam. WAV (KB)**
Melodia Simple	2.42	29	1,389
Operaciones Listas	3.25	29	1,213
Escaladas musicales	3.48	29	1,389
Condicionales	3.40	28	701

- El sistema Algoritmia demuestra un rendimiento consistente y eficiente en la generación de contenido musical, con tiempos de ejecución que oscilan entre los 2.42 y 3.48 segundos para diferentes casos de prueba. La estabilidad en el tamaño de los archivos PDF (2829 KB) indica una generación de partituras optimizada, mientras que la variación de los archivos de audio WAV van desde

los 701 KB hasta los 1389 KB, lo que refleja la complejidad y duración de las composiciones musicales generadas, confirmando la escalabilidad del sistema para diferentes niveles de complejidad algorítmica.

Cuadro 4.2: Tasas de Éxito en Algoritmia

Componente	Tasa de Éxito	Observaciones
Generación PDF	100 %	LilyPond confiable
Generación MIDI	100 %	Integración directa
Generación WAV	90 %	Depende de los bancos de sonido
Ejecución código	95 %	Errores en sintaxis compleja

- La tabla de tasas de éxito revela un desempeño general muy positivo del sistema, destacando una generación perfecta de archivos PDF y MIDI, lo que confirma la robustez de LilyPond y su integración directa. Sin embargo, también se identifica pequeñas áreas de mejora en la generación de archivos WAV, cuya tasa del 90 % es algo limitada por la dependencia de bancos de sonidos externos, y en la ejecución de código, donde un 95 % de éxito indica que la sintaxis compleja aún representa un desafío ocacional que requiere optimización.

4.3. Salidas Generadas

El sistema genera de forma automatizada múltiples salidas musicales a partir del código escrito en el lenguaje Algoritmia y como resultado muestra una visualización de los archivos generados (PDF, MIDI, WAV) con descarga directa y reproducción inmediata desde la interfaz web diseñada con HTML, para demostrar un flujo de trabajo integrado que transita desde la programación musical hasta la obtención de productos musicales listos para utilizar.

Salida de la interfaz Web

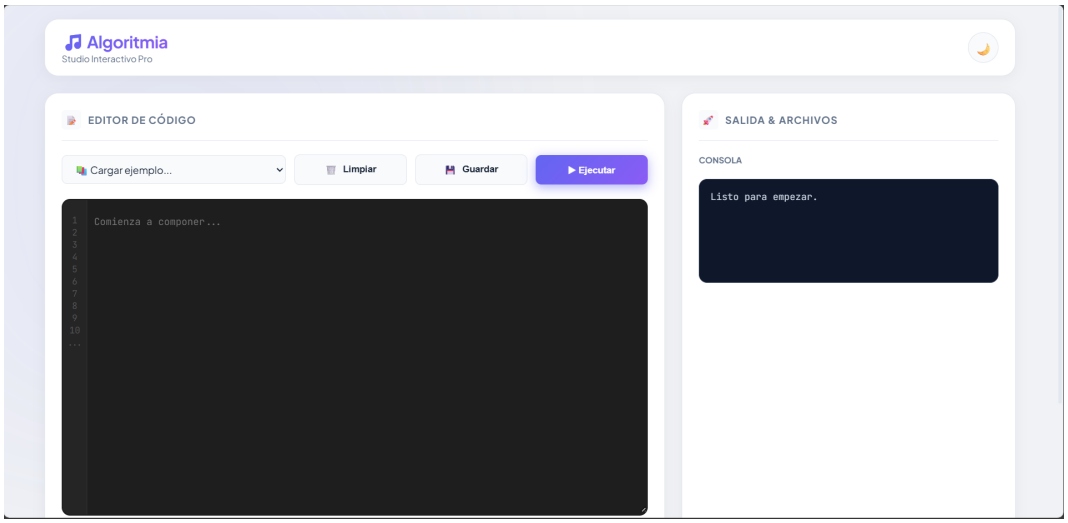

Figura 4.1: Interfaz Web para el usuario

- Para facilitar la interacción con el usuario, se optó por desarrollar una interfaz web que permite a los usuarios experimentar el intérprete de Algoritmia mediante un editor de código integrado con ejemplos predefinidos para que de esa manera pueda visualizar la estructura de archivos resultantes.

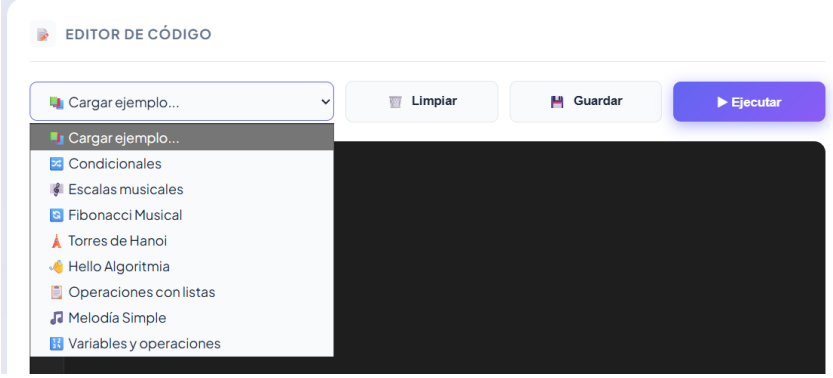

Figura 4.2: Lista de ejemplos para interactuar con el intérprete de Algoritmia

Tipos de archivos de salida

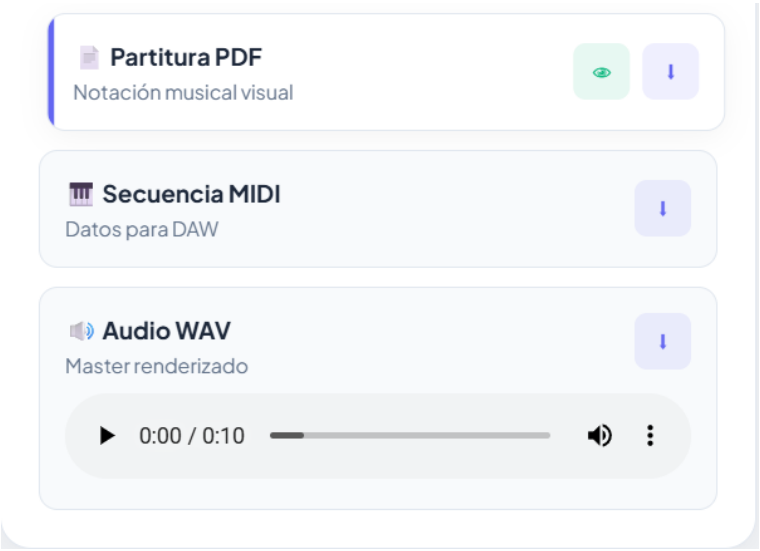

Figura 4.3: Interfaz para la generación de archivos y descarga

Salida de partituras PDF generadas:

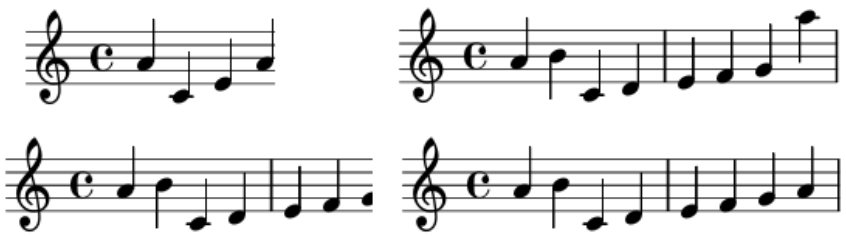

Figura 4.4: Partituras generadas por Algoritmia

5. CONCLUSIONES Y TRABAJO FUTURO

Tras el desarrollo y análisis del proyecto, hemos llegado al final, pero eso no indica que todo fue perfecto, al contrario hemos presentado uno que otro inconveniente, pero no imposible de resolver, entonces a partir de esos hallazgos, vamos a extraer nuestras conclusiones que reflejan el alcance y la efectividad de nuestro trabajo realizado, así como proponer mejoras para futuras etapas del proyecto.

5.1. Logros Alcanzados

El principal logro del proyecto Algoritmia fue el desarrollo exitoso de un intérprete web funcional que permite crear, editar y ejecutar código musical, generando automáticamente partituras PDF, archivos MIDI y audio WAV de calidad. Además, de que se implementó un editor intuitivo con capacidad para cargar ejemplos, ejecutar código en tiempo real y gestionar las salidas generadas, demostrando así la viabilidad de integrar lenguajes de programación especializados en entornos web interactivos accesibles para usuarios sin conocimientos técnicos avanzados.

5.2. Dificultades Encontradas

Durante el desarrollo enfrentamos varias dificultades técnicas, particularmente en la generación de archivos WAV donde la tasa de éxito del 90 % reveló dependecia hacia los bancos de sonido externos, por lo que en un inicio no nos generaba el audio y en la ejecución de código con sintaxis compleja presentó un 5 % de errores, además, se encontraron desafíos en la sincronización entre los diferentes componentes del sistema y en la optimización del rendimiento para manejar correctamente las salidas complejas sin afectar la experiencia dentro del navegador.

5.3. Trabajo Futuro

Como trabajo futuro planteamos mejorar la tasa de éxito en generación del archivo WAV mediante la integración de bancos de sonidos correctamente, además, de implementar un sistema de depuración más robusto para reducir errores en sintaxis compleja, y expandir las funcionalidades del lenguaje con estructuras de control musical avanzadas. Adicionalmente, queremos proyetar la integración de plataformas de distribución musical, y soporte para más formatos de exportacción que amplíen las aplicaciones prácticas del lenguaje en entornos educativos y profesionales.

Bibliografía

[1] Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf. Recuperado de: https://dl.icdst.org/pdfs/files3/a91ace57a8c4c8cdd9f1663e1051bf93.pdf

[2] ANTLR Official Documentation (2024). *ANTLR 4 GitHub Repository*. Recuperado de: https://github.com/antlr/antlr4

[3] Parr, T. (2020). *Language Implementation Patterns*. Pragmatic Bookshelf. Recuperado de: https://nlogn.art/wp-content/uploads/2023/04/Language-Implementation-Patterns.pdf

[4] LilyPond Development Team (2024). *LilyPond Notation Reference*. Recuperado de: https://lilypond.org/doc/v2.24/Documentation/learning

[5] LilyPond Development Team (2024). *Automated Engraving: The LilyPond Approach*. Recuperado de: https://lilypond.org/doc/v2.24/Documentation/essay-big-page.html

[6] Timidity++ Development Team (2024). *Timidity++ Documentation*. Recuperado de: https://timidity.sourceforge.net

[7] SoundFont Technical Specification (2024). *SoundFont® 2.04 File Format Specification*. Recuperado de: https://soundfont3.pages.dev/routes/2.%20SF%202.04%20Spec/README.html

[8] Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). *Modern Compiler Design*. Springer. Recuperado de: https://dpvipracollege.ac.in/wp-content/uploads/2023/01/Modern.Compiler.Design.2nd.pdf