SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Programming Language Concepts Final Report

DJcode to WAV compiler

Prepared by Cohort 2 Group 2:

Chu Chi Huen Carina (1006423)

Wong Jia Kang (1006022)

Jalen Andrew Mateo (1006042)

21 April 2025

Table Of Contents

1 Project Overview

1.1 Introduction

This project presents **DJcode**, a custom domain-specific language (DSL) designed for composing music through code. We built a compiler that takes a `.dj` file written in DJcode, analyzes its structure, and generates a valid `.wav` audio file as output. The system performs fundamental compiler stages like lexical analysis and parsing, incorporating Context-Free Grammar (CFG) through Python for the parsing and for the overall linear state transitions for the pipeline. We utilise C code in order to generate sound data based on the user input, and organise it into the sequence that they have indicated. The final output is a fully synthesized waveform that reflects the DJcode instructions.

1.2 Project Context

This DJcode project was developed as a group assignment for the Programming Language Concepts (PLC) course. The assignment required us to demonstrate familiarity with the materials taught in class:

- Implementing lexical, syntactic and semantic analysis.
- Representing a finite state machine (FSM) in our logic.
- Producing a tangible output which in our case, is a binary file WAV file for sound.
- Clean C code

Our team chose to extend compiler concepts into the field of audio programming, thereby linking abstract syntax processing with creative multimedia output. We created a new language called "DJcode" which is a very simple language to define music tracks and playing them, which is discussed further in section 3.

1.3 DJcode Impact

DJcode addresses the challenge of making sound synthesis and musical pattern creation accessible through structured code. Writing raw sound wave data or controlling low-level digital audio processes can be complex and unintuitive for most users. DJcode abstracts these complexities, allowing users to write simple instructions to describe rhythms, instruments, volumes, and measures, which the compiler interprets into actual sound.

This approach:

- Shows how high-level symbolic representations can drive low-level outputs (i.e., waveforms).
- Highlights the value of DSLs in bridging the gap between programming and creative expression.
- Introduces students and beginners to compiler logic in a playful, engaging way

1.4 Project Goal

The primary goal of this project is to demonstrate the application of core compiler principles in a novel and creative context. By designing a language (DJCode) that compiles into sound, we achieved the following:

- Reinforced understanding of compiler stages (lexing, parsing, semantic checks).
- Implemented a real-time use case that is representable by finite state machines (FSM).
- Integrated abstract syntax with waveform generation logic.
- Created a functioning and enjoyable output that rewards correct language use.

2 System Overview

The DJcode Audio Compiler system converts user-written .dj files into playable .wav audio through a structured pipeline involving lexical analysis, parsing, token transformation, sound synthesis, and file generation (refer to Figure 1). First, the Lexer scans the input file to produce tokens.txt, which the Main Parser processes to validate and organize musical structures into transformed_tokens.txt. This transformed data is then interpreted by the Sound Synthesis module. Finally, the WAVGenerator takes the synthesized sound data and encodes it into a standard .wav file.

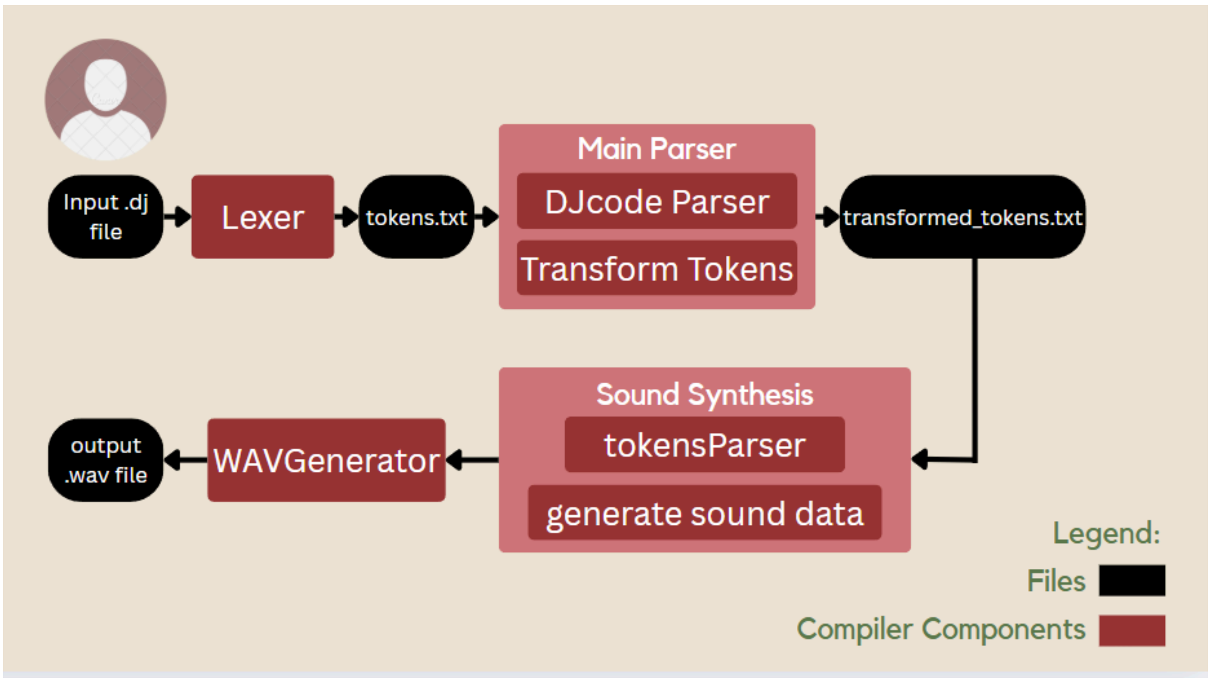


Figure 1: Overall System Pipeline

3 DJcode Design

DJcode is a simple language that is designed for sequencing instrument patterns. Code written in DJcode is saved with a .dj extension. The language is structured into two main sections

1. **Pattern Definitions**: This section allows users to define patterns and specify the instruments and their corresponding sounds (refer to Table 2). Each pattern begins with a label “Pattern1:”, followed by an instrument, and sequence of instrument sounds.

2. **Main block:** This main block determines the execution sequence of previously defined patterns. It always begins with the line “Drop the beat:”. This is followed by a series of Play statements that call the defined patterns and repeat them for a specified number of times.

Example 1:

```
Pattern1:
Drum boom
Triangle dididing

Pattern2:
Drum boom clap tsst
Triangle ding diding

Drop the beat:
Play Pattern1 x2
Play Pattern2 x3
```

Code	Comment
PatternX:	Pattern definitions where X could be [1-4]
Drop the beat:	Represents the main function
Play	Calling the pattern
xN	The number of times N to repeat playing the specified pattern where N could be [1-8]

Table 1: DJcode keywords and syntax

Instrument	Instrument sounds	Comment
Drum	boom, clap,tsst, crash, rest, dun	Basic percussive sounds
Triangle	ding, diding, dididing	Bell-like tones

Table 2: Instrument and corresponding instrument sounds

The overall capabilities of DJcode is that users can define pattern sequences only and then can play these patterns in any order or in any number of loops that they want. However, in the future, the capabilities of DJcode can be extended so that users can play with the BPM in each pattern or they can include individual sounds not defined in patterns to make it more versatile. For the purpose of this project, we have limited the MAX_PATTERNS that a user can define to be 4 and the MAX BEATS PER PATTERN to be 8. This makes it easier to handle but it can be expanded in the future to go beyond these limits.

4 Compiler Components

4.1 Lexical Analysis

The tokenization process in this lexer (lexel.l) is designed to recognize and classify elements of a custom DJ language script (test.dj) into structured tokens. This is done using Lex (Flex), which scans the input file and matches text against a series of regular expressions (regex) defined in the %% section of the file. These regex patterns map specific parts of the input into corresponding tokens that represent syntactic elements of the DJ language (refer to Table 3).

Regex Pattern	Token	Description
“Pattern”	PATTERN	Recognizes pattern block declarations
“:”	COLON	
[0-9]+	NUMBER	Captures numerical value for repetitions, or for patten number
“Drum” | “Triangle”	INSTRUMENT	Matches instrument names
"boom" | "tsst" | "clap" | "dun" | "ding" | "diding" | "dididing" | "crash" | "rest"	INSTRUMENT_SOUND	Recognizes specific instrument sound
“Drop the beat”	MAIN	Identifies the start of the main script block
“Play”	PLAY	Matches the command to play a specific pattern
“x”	LOOP	Captures the repetition operator
[\t\n]+	(ignored)	Whitespace is ignored
.	(ignored)	All other characters not explicitly matched are ignored

Table 3: Lexer’s tokenized output from user input

4.2 Parser & FSM

After the lexer outputs the tokens from the .dj file, these tokens are passed as input to the parser for syntax analysis. The parser, built on a context-free grammar (CFG), verifies that the tokens are arranged in the correct order and conform to the expected structure of the language we expect. Below is an example of how the tokens passed as input are parsed by our parser according to the rules defined in the context-free grammar (CFG).

```
(1) program → pattern_block main_block

(2) pattern_block → named_pattern

(3) pattern_block → pattern_block named_pattern

(4) named_pattern → PATTERN NUMBER COLON instrument_sequence

(5) instrument_sequence → instrument_sound_group

(6) instrument_sequence → instrument_sequence instrument_sound_group

(7) instrument_sound_group → INSTRUMENT sound_sequence

(8) sound_sequence → INSTRUMENT_SOUND

(9) sound_sequence → sound_sequence INSTRUMENT_SOUND

(10) main_block → MAIN COLON main_play_list

(11) main_play_list → main_play_entry

(12) main_play_list → main_play_list main_play_entry

(13) main_play_entry → PLAY PATTERN NUMBER LOOP NUMBER
```

Table 4: CFG rules for our parser

4.2.1 Derivation Example:

```
PATTERN
NUMBER 1
COLON
INSTRUMENT Drum
INSTRUMENT_SOUND boom
INSTRUMENT Triangle
INSTRUMENT_SOUND dididing

PATTERN
```

```
NUMBER 2
COLON
INSTRUMENT Drum
INSTRUMENT_SOUND boom
INSTRUMENT_SOUND clap
INSTRUMENT_SOUND tsst
INSTRUMENT Triangle
INSTRUMENT_SOUND ding
INSTRUMENT_SOUND diding

MAIN
COLON
PLAY
PATTERN
NUMBER 2
LOOP
NUMBER 2
PLAY
PATTERN
NUMBER 2
LOOP
NUMBER 3
```

We will first use the production rule

(1) program → pattern_block main_block

To split them into pattern_block and main_block

pattern_block

Production Rule	Rule used	Result
pattern_block -> pattern_block named_pattern	3	pattern_block named_pattern
pattern_block -> named_pattern	2	named_pattern named_pattern
named_pattern → PATTERN NUMBER COLON instrument_sequence (x2)	4	PATTERN NUMBER COLON instrument_sequence PATTERN NUMBER COLON instrument_sequence

instrument_sequence → instrument_sequence instrument_sound_group (x2)	6	PATTERN NUMBER COLON instrument_sequence instrument_sound_group PATTERN NUMBER COLON instrument_sequence instrument_sound_group
instrument_sequence → instrument_sound_group (x2)	5	PATTERN NUMBER COLON instrument_sound_group instrument_sound_group PATTERN NUMBER COLON instrument_sound_group instrument_sound_group
instrument_sound_group → INSTRUMENT sound_sequence (x4)	7	PATTERN NUMBER COLON INSTRUMENT sound_sequence INSTRUMENT sound_sequence PATTERN NUMBER COLON INSTRUMENT sound_sequence INSTRUMENT sound_sequence
sound_sequence → sound_sequence INSTRUMENT_SOUND (x3)	9	PATTERN NUMBER COLON INSTRUMENT sound_sequence INSTRUMENT sound_sequence PATTERN NUMBER COLON

		INSTRUMENT sound_sequence INSTRUMENT_SOUND INSTRUMENT_SOUND INSTRUMENT sound_sequence INSTRUMENT_SOUND
sound_sequence → INSTRUMENT_SOUND (x4)	8	PATTERN NUMBER COLON INSTRUMENT INSTRUMENT_SOUND INSTRUMENT INSTRUMENT_SOUND PATTERN NUMBER COLON INSTRUMENT INSTRUMENT_SOUND INSTRUMENT_SOUND INSTRUMENT_SOUND INSTRUMENT INSTRUMENT_SOUND INSTRUMENT_SOUND

main_block

Production Rule	Rule used	Result
main_block → MAIN COLON main_play_list	10	MAIN COLON main_play_list
main_play_list → main_play_list main_play_entry	12	MAIN COLON main_play_list main_play_entry
main_play_list → main_play_entry	11	MAIN COLON main_play_entry main_play_entry

main_play_entry → PLAY PATTERN NUMBER LOOP NUMBER	13	MAIN COLON PLAY PATTERN NUMBER LOOP NUMBER PLAY PATTERN NUMBER LOOP NUMBER

The parser can be represented as a FSM as represented in Figure 2.

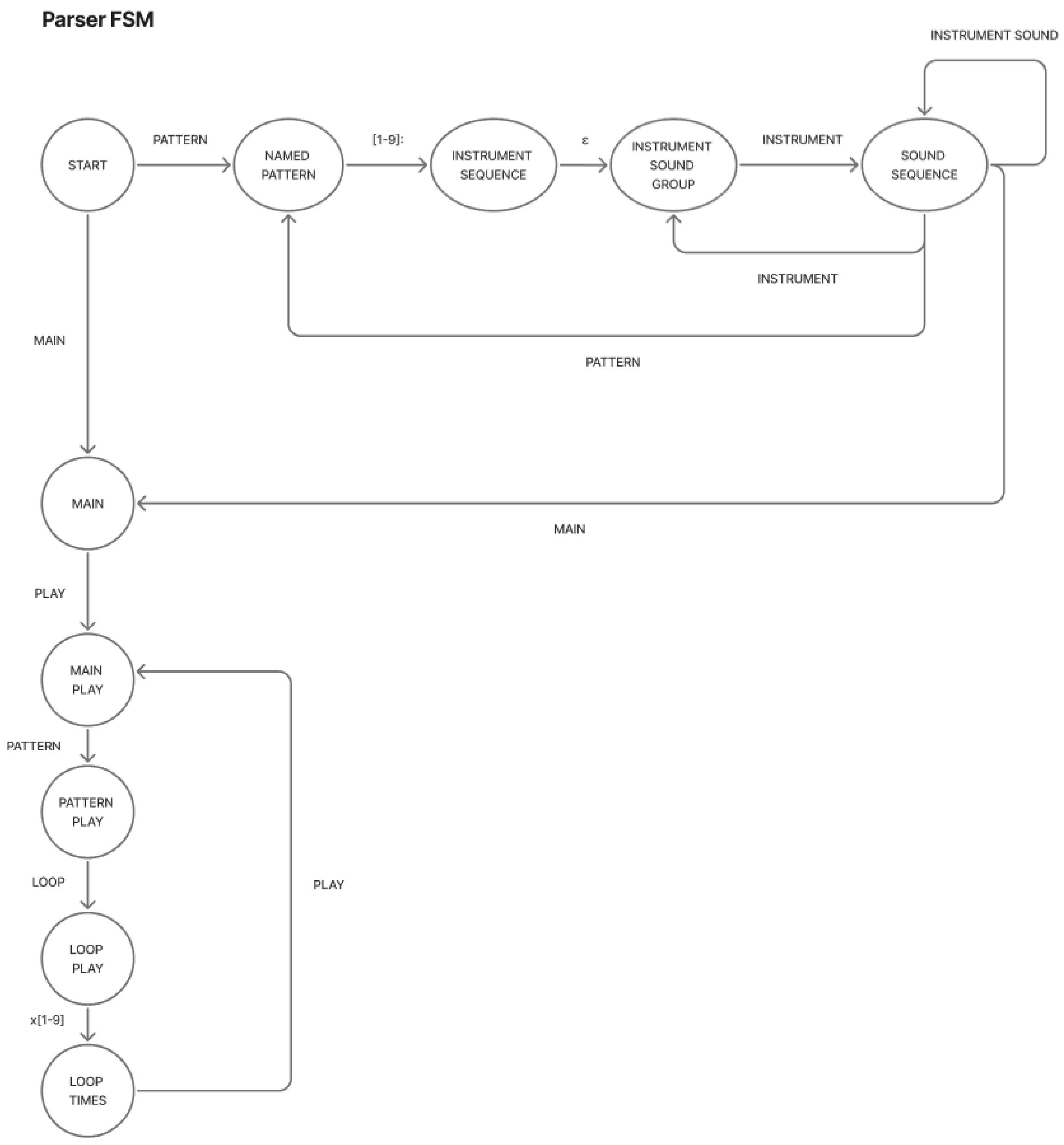


Figure 2: FSM of the parser.py

4.3 Semantic Analysis

Additionally, the parser performs several semantic checks to ensure the DJcode input is logically valid. It verifies that each PATTERN used in the MAIN block has been previously defined, checks that instrument sounds correspond to valid instrument names, and ensures no instrument is declared without an associated sound. These checks prevent runtime errors and maintain the integrity of the musical structure before sound synthesis begins.

5 Sound Synthesis & WAV Output

5.1 Sound Synthesis Basics

To generate meaningful audio from DJcode, we implemented core concepts from digital audio synthesis and basic music theory. At the fundamental level, all sound can be broken down into waveforms, frequency, amplitude, and duration — the building blocks of our system.

5.1.1 Fundamental Waveforms

We support several basic waveform types, each associated with different musical characteristics:

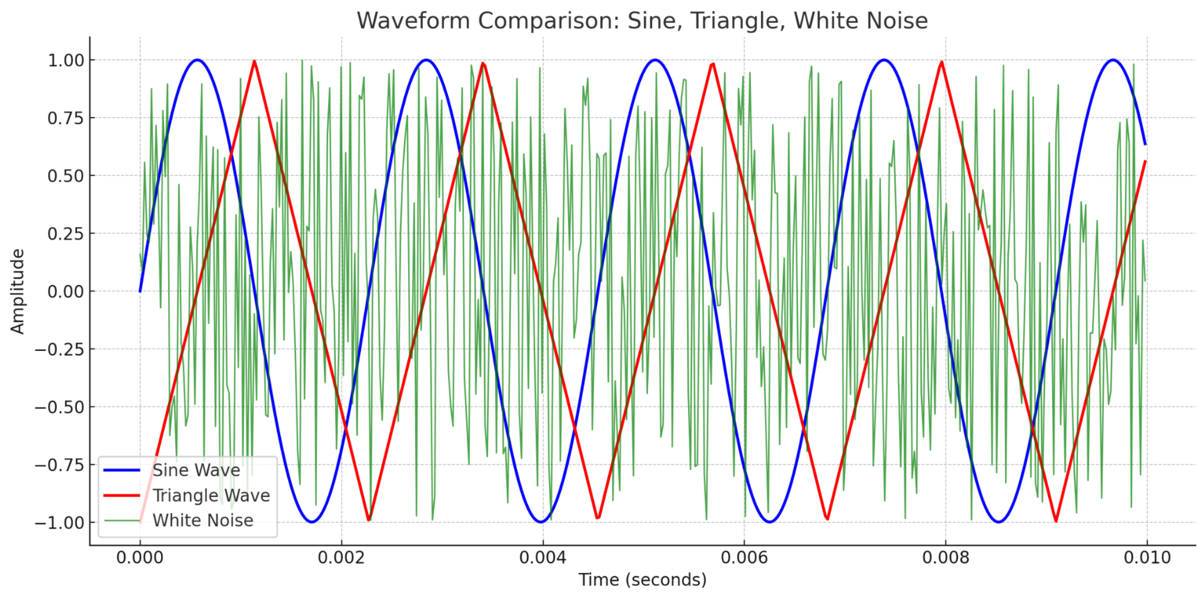


Figure 3: Fundamental waveforms used in the sound data generated

- **Sine wave**: The purest tone, with a single frequency and no harmonics. Used for clean bass (e.g., kick drums).
- **Triangle wave**: A smoother tone with only odd harmonics. Offers a slightly brighter sound than sine, often used for mid-range tones.
- **White noise**: A randomized signal covering all frequencies. Used to mimic unpitched sounds like snares or hi-hats.

We note that square waves and sawtooth waves are also fundamental waveforms but for our initial implementation of this project we have omitted those. In the future, if we wish to expand the capabilities of DJcode we can make use of those to make different sounds.

5.1.2 Frequencies and Volume

Frequency refers to the number of oscillations a wave completes per second, measured in Hertz (Hz). A higher frequency produces a higher pitch, and vice versa.

- For instance, 440 Hz corresponds to the musical note A4.

In DJcode, we associate fixed frequencies to simulate instrument tones without real-time pitch shifting.

For volume, we can essentially scale the values of the soundwave data up to increase volume, or scale them down to reduce volume.

5.1.3 Tempo and Beats

BPM (beats per minute) determines the speed of the song. At 120 BPM, each beat lasts 0.5 seconds.

Each beat is divided into samples based on the audio sample rate (e.g., 44,100 samples per second). Beat duration directly affects how many samples we generate per sound event.

In our case, we do not allow the users to implement BPM changes but in the future, this would be one of the more key features to implement as contemporary songs incorporate a lot of changes in the BPM.

5.1.4 Envelopes

To prevent clicks and shape the sound dynamically, we use a basic ADSR envelope:

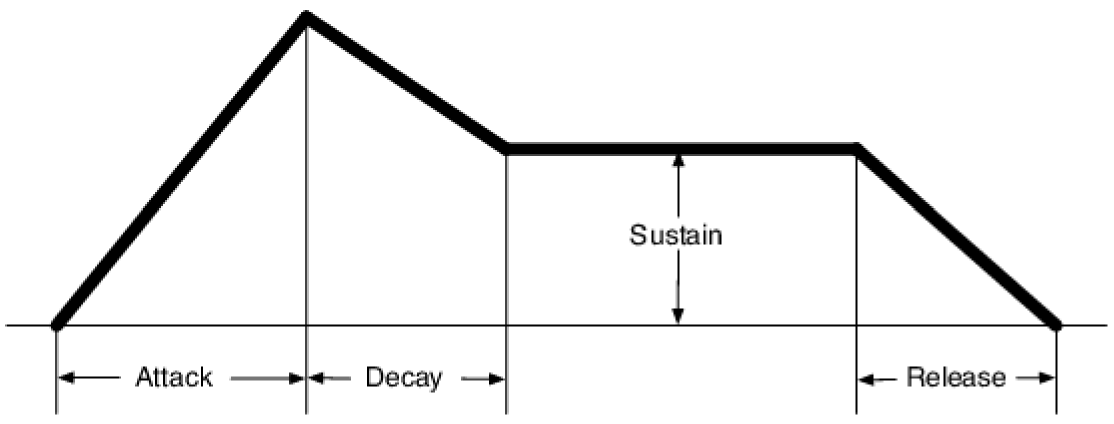


Figure 4: ADSR sound envelope to make generated sounds more natural
Image credit: [Instruments and the Electronic Age: Toward a Terminology for a Unified Description of Playing Technique]

- Attack (A): Gradual increase in amplitude at the start.
- Decay (D): Gradual decrease to sustain value after the sound plays.
- Sustain (S): Constant amplitude in the middle of the sound.

- Release (R): Gradual decrease to silence after sustain.

This ADSR envelope allows for the sounds we produce to sound a lot more natural. We also make use of just Decay for some of our sound profiles i.e. straight away implement a gradual decrease in amplitude to silence. Implementing just Decay is simpler overall and can sometimes achieve the same natural sound that we are looking for.

5.2 Obtaining Sound Profiles For Each Instrument Sound

To simulate realistic instrument sounds in DJcode, we mix multiple basic waveforms, such as sine, triangle, and white noise, each contributing different harmonic content. The result is a custom sound profile for each instrument, shaped further by an amplitude envelope.

For example:

- Our “dun” sound which imitates a floortom in a drum set uses a mixing of white noise, a sine wave and a triangle wave, all with a low frequency and with a decay envelope.
- Our clap sound for the drum actually utilises a sine wave and white noise mixed together.
- Our “dididing” sound for the triangle is obtained by mixing 3 different triangle waves.

These combinations allow us to craft distinct harmonic structures tailored to the perceived character of each instrument.

In code, this mixing is handled by creating a temp variables that accumulates waveform values. This sum is then written into the output master buffer as the final mixed sample.

By layering these elements in code, we build up each instrument sound profile procedurally — avoiding the need for audio samples while maintaining a distinct and musically coherent output.

Each sound profile can fine-tuned by:

- Adjusting the base frequency used.
- Scaling the amplitude to prevent clipping or overpowering in the final mix.
- Timing placement relative to the beat and measure structure for rhythmic clarity.

This iterative mixing process allowed us to ensure each instrument remained distinguishable and musically coherent in the final `.wav` output.

In the future, to expand on this, we can allow users to create their own sound profiles for a specific beat name and instrument through the use of this mixing. This will increase the personalisation abilities of DJcode and could lead to more complex applications.

5.3 WAV Output Generation

Once the mixed and envelope-shaped audio samples are synthesized for each beat, they are stored in a master buffer and prepared for export as a .wav file. The WAV file format is ideal for this purpose due to its uncompressed, raw audio structure and widespread compatibility.

A standard WAV file follows the **RIFF (Resource Interchange File Format)** structure and includes:

1. **RIFF Header** – Specifies the file type and total size.
2. **Format Chunk ("fmt ")** – Contains audio format details like sample rate and bit depth.
3. **Data Chunk ("data")** – Holds the actual audio sample data.

In our code, we define a WAVHeader struct to write the first part of the file in binary format:

```
typedef struct WAVHeader {
    char riff_tag[4];       // "RIFF"
    uint32_t riff_length;   // File size - 8 bytes
    char wave_tag[4];       // "WAVE"
    char fmt_tag[4];        // "fmt "
    uint32_t fmt_length;    // Length of format chunk (16 for PCM)
    uint16_t audio_format;  // PCM = 1
    uint16_t num_channels;  // Mono = 1
    uint32_t sample_rate;   // 44100 Hz
    uint32_t byte_rate;     // SampleRate * NumChannels *
BitsPerSample/8
    uint16_t block_align;   // NumChannels * BitsPerSample/8
    uint16_t bits_per_sample; // 16 bits
    char data_tag[4];       // "data"
    uint32_t data_length;   // NumSamples * NumChannels *
BitsPerSample/8
} WAVHeader;
```

This header is written at the start of the .wav file using fwrite(), followed by the audio sample data.

The audio data in a WAV file consists of a sequence of discrete samples, each representing the amplitude of the sound wave at a specific point in time. These samples are written in order, creating a digital representation of the continuous analog waveform. This is why we can directly use the waveform data for each beat and instrument and append it to the back of the WAV file.

5.4 Link To Lexer and Parser Output

The sound synthesis and WAV generation stages in DJcode rely on a structured intermediate output file, typically named tokens.txt, which is produced by the earlier phases of the compiler, namely the lexer and the parser. This intermediate file acts as the bridge between the frontend (textual DJcode input) and the backend (sound generation pipeline).

The formatted_tokens.txt file includes:

- A list of **pattern declarations**, where each pattern has a name and a sequence of sound tokens (e.g., BOOM, TSST, DIDING).
- A sequence of **PLAY commands**, which specify the **order and repetition** of the patterns to be played.

This file allows us to decouple parsing logic from audio generation logic, enabling modular compilation and easier debugging.

We have thus implemented another C-based parser (tokensParser) which is responsible for reading and interpreting tokens.txt using in a simple, straighforward manner. It performs the following:

- Parses each line and identifies whether it defines a pattern, a sound token, or a play instruction.
- Maps each recognized token to the corresponding sound generation function (e.g., generate_boom_waveform(), generate_clap_waveform()).
- Stores pattern definitions in memory with a fixed limit (4 patterns, 8 beats per pattern) to simplify memory management.
- Keeps track of playback order, including loops for repeated patterns, and stores this as a linear sequence of instructions for beat scheduling.

This parser plays a crucial role in ensuring that only structurally and semantically valid input is passed into the audio backend.

5.5 Related Code Files

- tokensParser.c and tokensParser.h
  - Parses the intermediate tokens.txt file and keeps track of patterns and play commands found.

- soundwaves.c and sounwaves.h
  - Contains the core logic for sound synthesis.
  - Implements functions to generate the actual audio waveform data for different sounds (e.g., specific musical notes, drum sounds, effects).
  - Implements sound envelopes for the audio waveform data

- WAVGenerator.c and WAVGenerator.h
  - Responsible for **WAV file generation.**
  - Makes use of the tokensParser to map sounds described in the tokens.txt file produced in the earlier stage of the compiler, to actual waveform data.
  - Puts the waveform data in loops arranged according to tokens.txt.

6 File Structure

The DJcode compiler project is divided into two primary directories: Lexer_Parser and Sound_Synthesis, each responsible for a different stage of the compilation pipeline.

The Lexer_Parser folder contains all the files related to lexical analysis and parsing. These include lexer.l, which defines the tokenization rules using Flex, and main.c, which serves as the entry point for executing the lexer. Upon execution, the lexer produces a tokens.txt file, which is then processed by transform_tokens.py and parser.py to generate a transformed_tokens.txt file for the sound generator.

The Sound_Synthesis folder contains the logic for interpreting formatted tokens into audible waveforms. It includes tokensParser.c and tokensParser.h, which parse the structured tokens and convert them into a sequence of audio commands. These are then handled by soundwaves.c and WAVGenerator.c, which are responsible for generating waveforms and writing them into .wav files.

The project root also includes a Makefile that automates the entire build process, a .gitignore file, a sample generated WAV file (DJcode_Beats_Example.wav), the user generated WAV file (NEW_DJcode_Beats.wav) and the main documentation file README.md.

```
├── Lexer_Parser
│   ├── djcode
│   ├── lex.yy.c
│   ├── lex.yy.o
│   ├── lexer
│   ├── lexer.l
│   ├── main.c
│   ├── parser.py
│   ├── test.dj
│   ├── tokens.txt
│   ├── transform_tokens.py
│   └── transformed_tokens.txt
├── Makefile
├── NEW_DJcode_Beats.wav
├── README.md
└── Sound_Synthesis
    ├── WAVGenerator.c
    ├── WAVGenerator.h
    ├── dj_generator
    ├── soundwaves.c
    ├── soundwaves.h
    ├── test_dj_generator
    ├── tokens.txt
    ├── tokensParser.c
    └── tokensParser.h
```

7 Setup & Usage Instructions

To run the DJcode compiler, ensure that the following tools and dependencies are installed:

- Flex – for lexical analysis (lexer.l)
- Python 3 – for parsing and transforming token files
- GCC – for compiling C source files
- make – to automate builds
- Math Library – used in waveform functions

Once dependencies are installed, you can build and run the system as follows:

- To compile and execute the full pipeline, use the following command in the root folder: `make all`
  This command will trigger:
  - Lexer compilation and token generation
  - Token transformation via Python
  - Parser validation
  - Audio file generation via the sound synthesis module

- To modify input, replace or edit the .dj file (e.g., test.dj) located in Lexer_Parser/, and update the DJCODE_INPUT path in the Makefile if necessary.
- To clean the project of intermediate files use make clean.
- The final output .wav file will be found in the root directory

The overall states of the compiler as we run through the commands listed in the Make file is as below (refer to Figure 5).

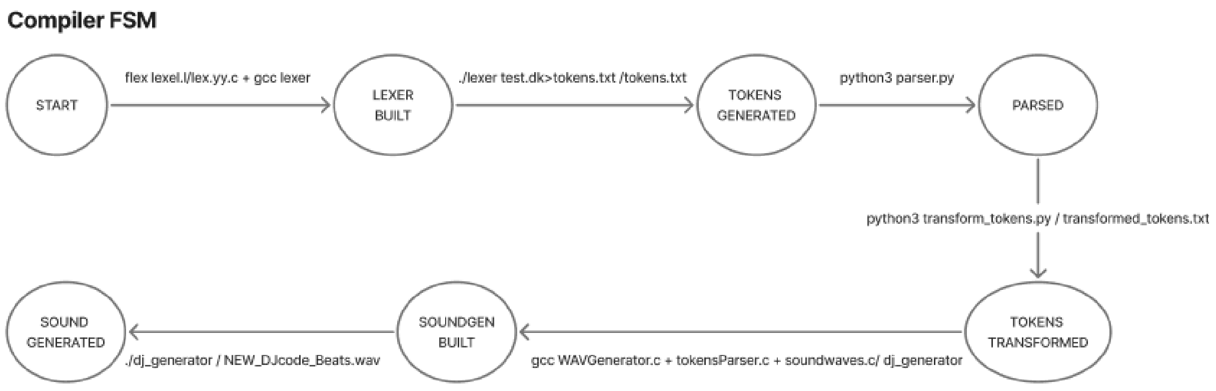

Compiler FSM
START
flex lexel.l/lex.yy.c + gcc lexer
LEXER BUILT
./lexer test.dk>tokens.txt /tokens.txt
TOKENS GENERATED
python3 parser.py
PARSED
python3 transform_tokens.py / transformed_tokens.txt
TOKENS TRANSFORMED
gcc WAVGenerator.c + tokensParser.c + soundwaves.c/ dj_generator
SOUNDGEN BUILT
./dj_generator / NEW_DJcode_Beats.wav
SOUND GENERATED

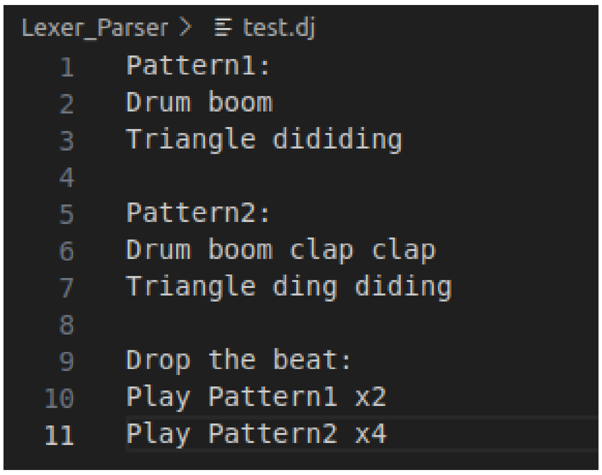
Figure 5: Compiler FSM

8 Results & Demo

This section presents an overview of how the DJcode compiler we have developed transforms a .dj file into a playable .wav file as also described in the usage instructions above, showcasing the full pipeline in action.

Step 1: Input File

Firstly, the compiler takes in a .dj input file written in DJcode syntax, which defines musical patterns using readable keywords for instruments and beats.



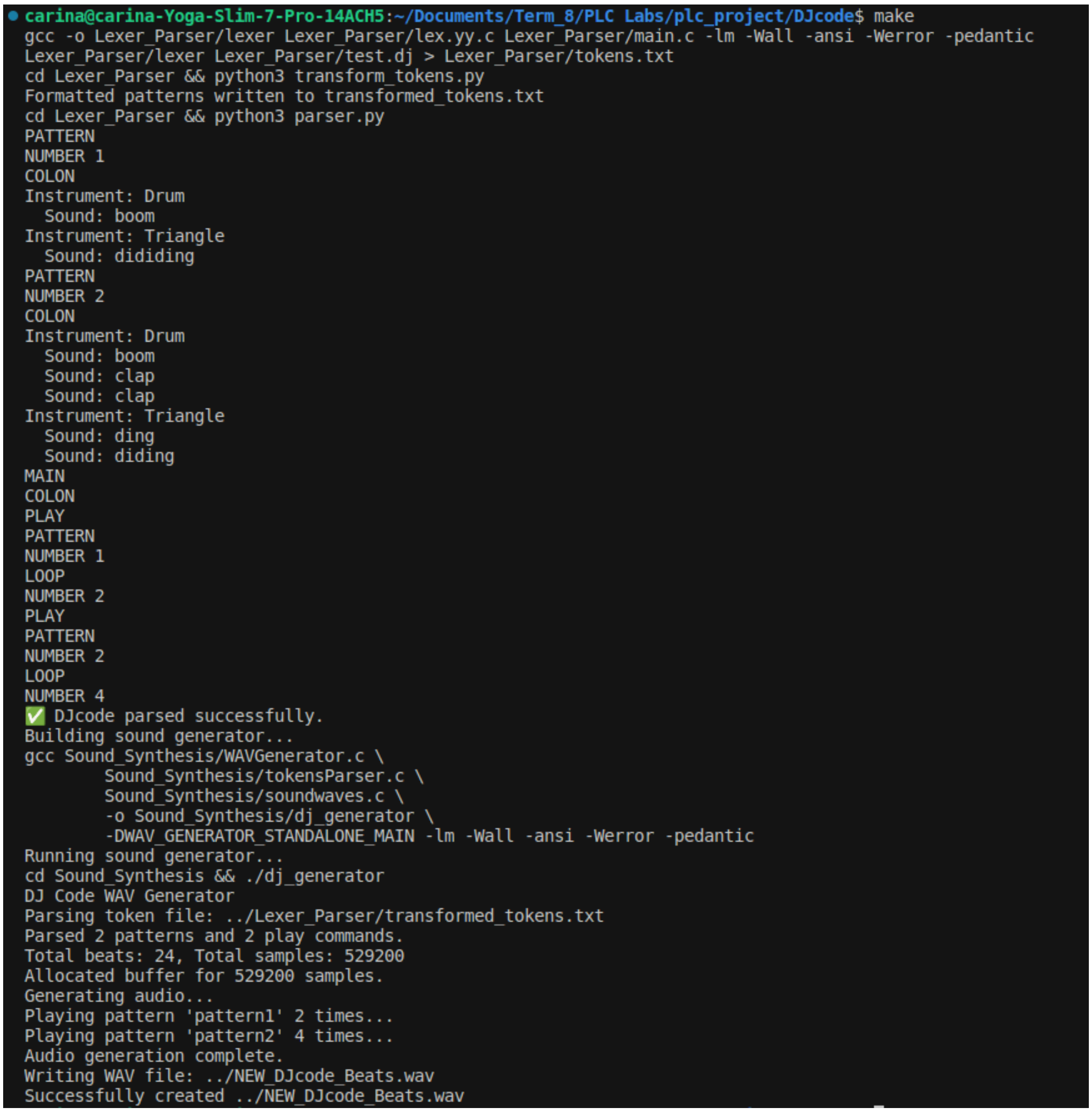
Lexer_Parser > test.dj

```
Pattern1:
Drum boom
Triangle didiing

Pattern2:
Drum boom clap clap
Triangle ding diding

Drop the beat:
Play Pattern1 x2
Play Pattern2 x4
```

Figure 6: Example test.dj input file

Step 2: Compilation Process

After running the command make all, the terminal displays print statements indicating the successful execution of each compilation stage, from lexical analysis and parsing to audio generation.



```
● carina@carina-Yoga-Slim-7-Pro-14ACH5:~/Documents/Term_8/PLC Labs/plc_project/DJcode$ make
  gcc -o Lexer_Parser/lexer Lexer_Parser/lex.yy.c Lexer_Parser/main.c -lm -Wall -ansi -Werror -pedantic
  Lexer_Parser/lexer Lexer_Parser/test.dj > Lexer_Parser/tokens.txt
  cd Lexer_Parser && python3 transform_tokens.py
  Formatted patterns written to transformed_tokens.txt
  cd Lexer_Parser && python3 parser.py
  PATTERN
  NUMBER 1
  COLON
  Instrument: Drum
    Sound: boom
  Instrument: Triangle
    Sound: dididing
  PATTERN
  NUMBER 2
  COLON
  Instrument: Drum
    Sound: boom
    Sound: clap
    Sound: clap
  Instrument: Triangle
    Sound: ding
    Sound: diding
  MAIN
  COLON
  PLAY
  PATTERN
  NUMBER 1
  LOOP
  NUMBER 2
  PLAY
  PATTERN
  NUMBER 2
  LOOP
  NUMBER 4
  ✅ DJcode parsed successfully.
  Building sound generator...
  gcc Sound_Synthesis/WAVGenerator.c \
          Sound_Synthesis/tokensParser.c \
          Sound_Synthesis/soundwaves.c \
          -o Sound_Synthesis/dj_generator \
          -DWAV_GENERATOR_STANDALONE_MAIN -lm -Wall -ansi -Werror -pedantic
  Running sound generator...
  cd Sound_Synthesis && ./dj_generator
  DJ Code WAV Generator
  Parsing token file: ../Lexer_Parser/transformed_tokens.txt
  Parsed 2 patterns and 2 play commands.
  Total beats: 24, Total samples: 529200
  Allocated buffer for 529200 samples.
  Generating audio...
  Playing pattern 'pattern1' 2 times...
  Playing pattern 'pattern2' 4 times...
  Audio generation complete.
  Writing WAV file: ../NEW_DJcode_Beats.wav
  Successfully created ../NEW_DJcode_Beats.wav
```

Figure 7: Terminal Display

Step 3: Final Output

Once compilation is complete, the system produces an output .wav file based on the input code. This file can be played in any standard media player, and accurately reflects the musical structure defined by the user.

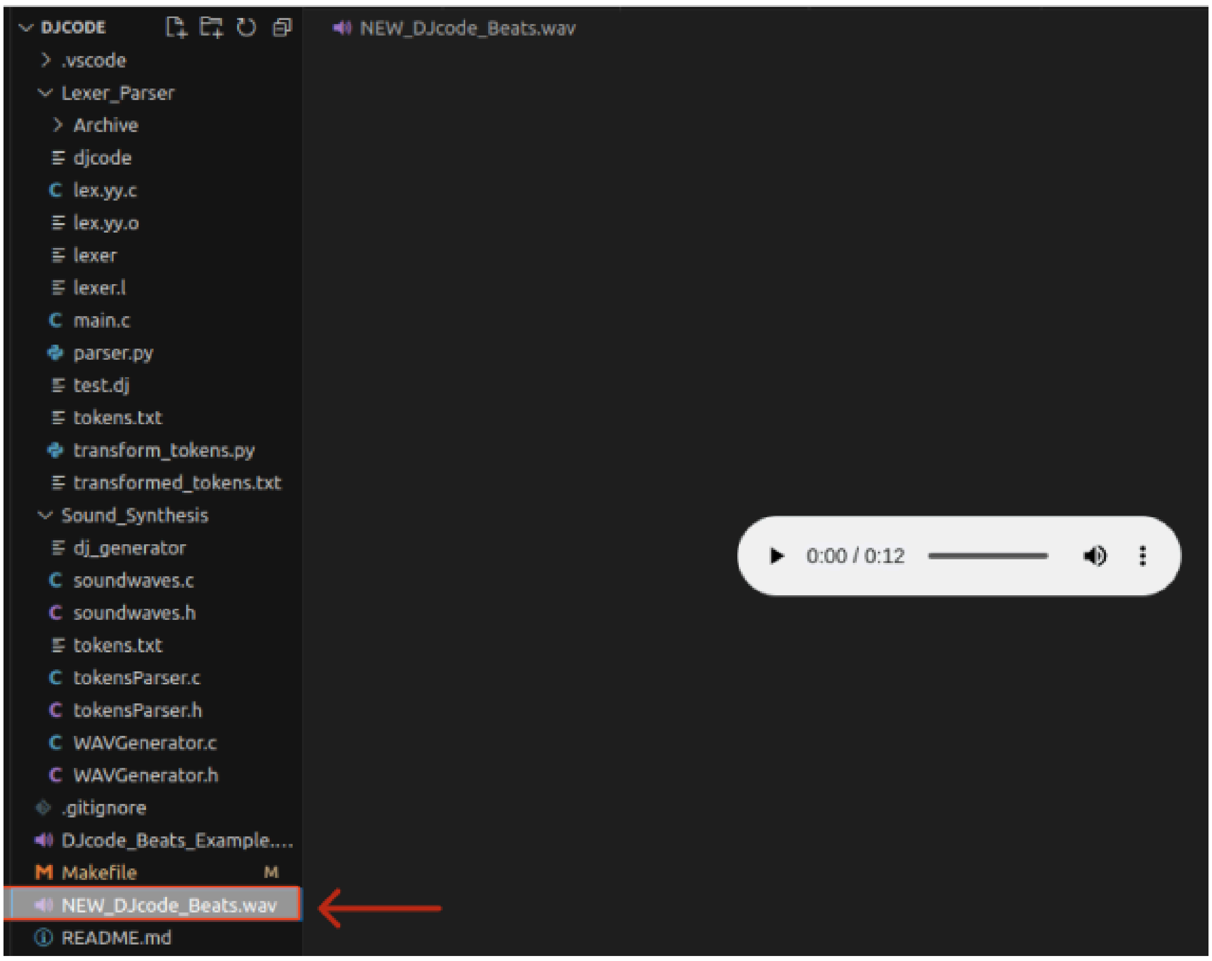


Figure 8: Output WAV file

9 Project Constraints

This project adhered to strict requirements to ensure alignment with the PLC assignment objectives:

- **Parser Implementation**: We developed a parser in Python to process the input .dj file, and tokenize, writing to an output tokens.txt and formatted_tokens.txt file. We also developed a custom straightforward parser in C to process the intermediate formatted_tokends.txt file, manually identifying pattern blocks, instrument tokens, and play commands.

- **FSM Representation**: FSMs were represented at two levels, one within the parser to enforce valid syntax structure (e.g., PATTERN -> sounds -> END), and another in the makefile to manage the overall compilation pipeline.

- **File Input and Output**: All pattern and play data were read from an input .dj file, and audio was written to a .wav file using C standard file I/O. This included manually writing the WAV header and raw PCM data.

- **Clean C Code**: Most code was written in ANSI C and compiles with:
  gcc -Wall -Werror -ansi -pedantic

  We used fixed-size arrays to avoid dynamic memory allocation, resulting in safer and simpler code. However, for the .dj to tokens parsing and lexer, we utilised flex and python to simplify things.

  To compile with the flags without errors, it only works on Windows and Linux systems. For Mac, please change the make file to remove the flags.

- **No External Libraries or Tools**: All functionality, including waveform synthesis, parsing, and WAV generation, was implemented from scratch. We did not use external audio or parsing libraries to maintain full transparency and control. However, we do acknowledge that we used python and flex to help with the parser and lexer.

10 Development Process & Challenges

We began the project by aligning on the core syntax and capabilities of DJcode as a group. This initial design phase helped us define a minimal yet expressive language that could support multiple patterns, instrument sounds, and a structured playback sequence. Once the foundation was set, we divided the work into parallel tracks:

- Sound synthesis and WAV generation, writing clean C code to generate and layer waveforms.
- Develop the parser and lexer with an FSM representation, responsible for processing the input file.
- Code to enable the output of the parser and lexer to be used by the WAV generation.

This separation allowed us to build and test each component independently before integrating them. Once all parts were functional, we moved into integration and full-system testing, ensuring that parsed tokens correctly triggered sound generation.

The challenges we encountered during this development process are as follows:

- Generating sound data and writing it to a file (e.g., harmonics, envelopes, waveforms) to sound like realistic sounds required understanding basic music theory.
- Implementing a manual lexer and parser required careful state management.
- Integration took longer than expected, especially in aligning pattern playback logic with waveform scheduling and handling edge cases like invalid tokens or overflowed buffers.

Despite these challenges, our modular approach enabled us to identify issues early and iteratively refine each part of the system until the final compiler ran smoothly. To ensure that the overall system was working properly, we tested various .dj files with different patterns, unknown tokens and missing END or PATTERN tokens. We verified that the output .wav files correspond to what was in the .dj input file.

Below is the contributions for this project's development. Overall, we felt it was balanced.

Person	Contributions
Carina	- Overall system integration - Implementation of main block for parser and lexer - FSM design - Report and slides
Jia Kang	- Implementation of parser and lexer for .dj file - Decided on key syntax and capabilities of DJcode - Report and slides
Jalen	- Implemented functions to generate sound data - WAV output file writing - Report and slides

Table 4: Contributions for each member

11 Conclusion

In conclusion, we successfully developed a DJcode-to-WAV compiler that transforms simple, symbolic beat-based code into playable .wav audio files. The project involved implementing a parser (represented as an FSM), waveform synthesis logic, and manual file output handling and all of that written in clean, modular ANSI C.

One of the key reflections from this project was realizing how powerful and flexible compiler design can be, even outside the context of general-purpose languages. Building a domain-specific language from scratch allowed us to think critically about syntax design, semantic rules, and the link between symbolic structure and real-world output.

We also deepened our understanding of basic music theory, which was essential for creating realistic instrument sounds. Concepts like harmonics and envelopes became important as well, apart from C syntax and parsing logic. In all, through this project, we learned the value of:

- Modular code structure, which made it easier to split work and integrate components.
- FSMs, which provided clear logic for both parsing and execution stages.
- Testing and validation
- Working within constraints, such as external libraries

Most importantly, this project gave us a glimpse into how language design and low-level programming can enable new forms of expression. DJcode is just a first step, but it showcases how compiler construction can be both technical and creative.

Also, looking ahead, we believe that DJcode is highly extensible. Its architecture makes it easy to expand capabilities such as by supporting user-defined instruments, dynamic BPM changes, new waveforms (e.g. sawtooth, square), or advanced sound effects. With these additions, DJcode could evolve into a more powerful music programming language while still remaining accessible to beginners.