Live Coding Language Design: PHAD

A Major Qualifying Project Report submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for

the Degree of Bachelor of Science

Benjamin Anderson

Norman Delorey

Henry Frishman

Mariana Pachon-Puentes

December 13, 2019

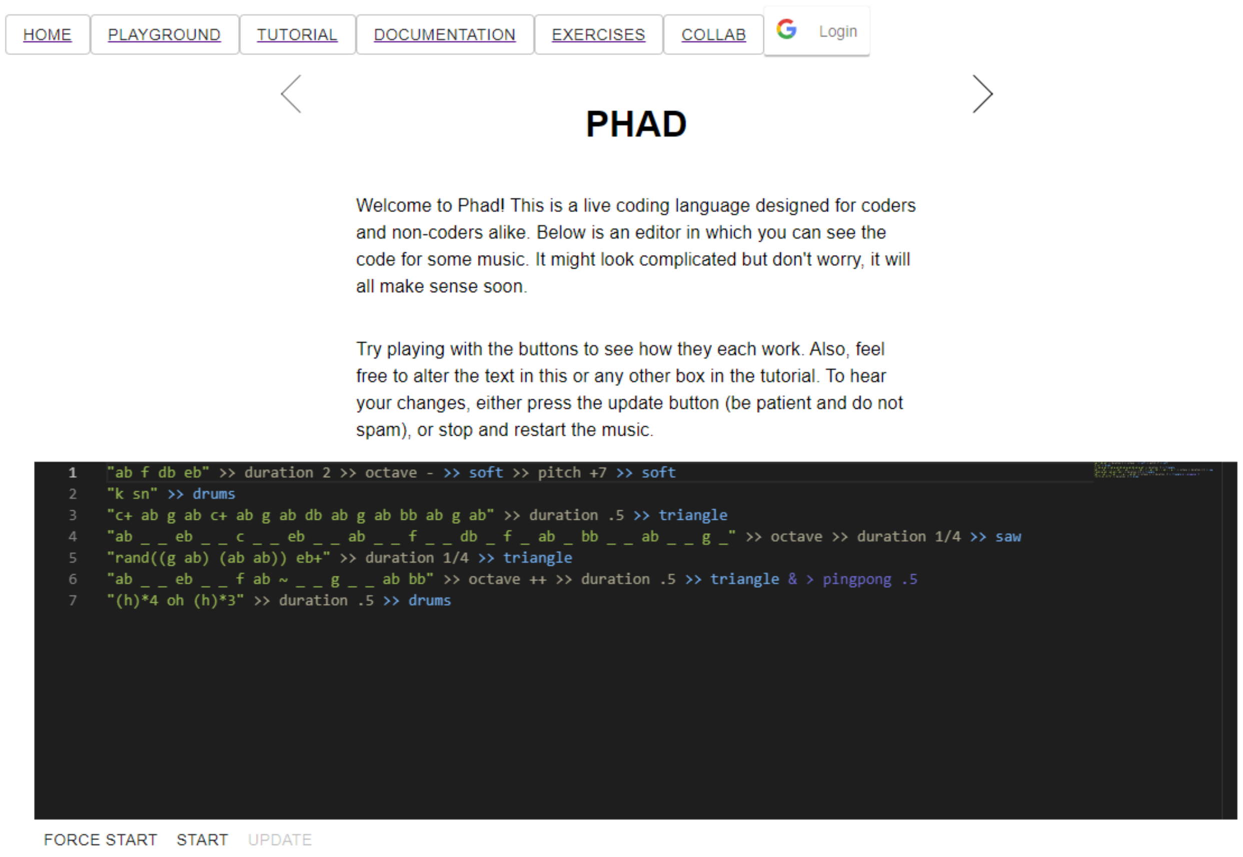
1 Abstract

Phad is a live coding environment built for web browsers. Phad strives for minimal but expressive syntax, with the purpose of achieving both readability and simplicity targeted to users with no programming or music experience. It is built around the continuous modification of notes and instruments, and the combination of both. The Phad environment was built with the user in mind, offering tutorials, exercises, a playground, and collaborative performance, all through the same application accessible here. Users are able to login with a Google Account and save code to our database, making it easy to come back to previous work, or to see what other users have created. The language and system design of Phad was inspired by previous live-coding environments, and was improved iteratively with external feedback as well as our experience at the Algorave. This paper will explore the design decisions and development processes that our group underwent when creating Phad.

2 Introduction

Live coding consists of writing code and modifying it on the spot to generate new outputs. Creative live coding [25] is a technique that can be used for generating sound and images with the use of code. This type of live coding has been popularized through different types of events such as Algoraves [2], during which live coders use all different kinds of software to improvise music or visuals in front of an audience. Although performances can be done by a single person, performing at Algoraves is usually a collaborative activity, where some code the audio portion, and others code the visuals. During these performances, the code used to generate creative outputs is considered a part of the overall performance, and is projected on the screens along with the generated visuals. In the past six years, Algoraves have become highly popular, with live-streams and performances all over the globe. In 2018 alone, Algoraves were hosted at venues in the USA, Argentina, Australia, Mexico, Canada, Japan, Brazil, the UK, and Colombia, and the community keeps growing.

This team was formed with the goal of developing a system that would make live coding music accessible to people with little to no programming or music experience. Phad was developed over the course of a semester and is now completely accessible through the web [3]. A screenshot from the final result of the project can be seen below.



PHAD

Welcome to Phad! This is a live coding language designed for coders and non-coders alike. Below is an editor in which you can see the code for some music. It might look complicated but don't worry, it will all make sense soon.

Try playing with the buttons to see how they each work. Also, feel free to alter the text in this or any other box in the tutorial. To hear your changes, either press the update button (be patient and do not spam), or stop and restart the music.

```
"ab f db eb" >> duration 2 >> octave - >> soft >> pitch +7 >> soft
"k sn" >> drums
"c+ ab g ab c+ ab g ab db ab g ab bb ab g ab" >> duration .5 >> triangle
"ab _ _ eb _ _ c _ _ eb _ _ ab _ _ f _ _ db _ f _ ab _ bb _ _ ab _ _ g _" >> octave >> duration 1/4 >> saw
"rand((g ab) (ab ab)) eb+" >> duration 1/4 >> triangle
"ab _ _ eb _ _ f ab ~ _ _ g _ _ ab bb" >> octave ++ >> duration .5 >> triangle & > pingpong .5
"(h)*4 oh (h)*3" >> duration .5 >> drums
```

Figure 1: A page from Phad's tutorial

3 Background

To research live coding, different live coding music languages and environments were explored. Experimenting with these languages and understanding features in existing languages, helped in the system and language design of Phad, as will be shown later in Table 1. This section describes the eight languages that were explored.

3.1 FoxDot

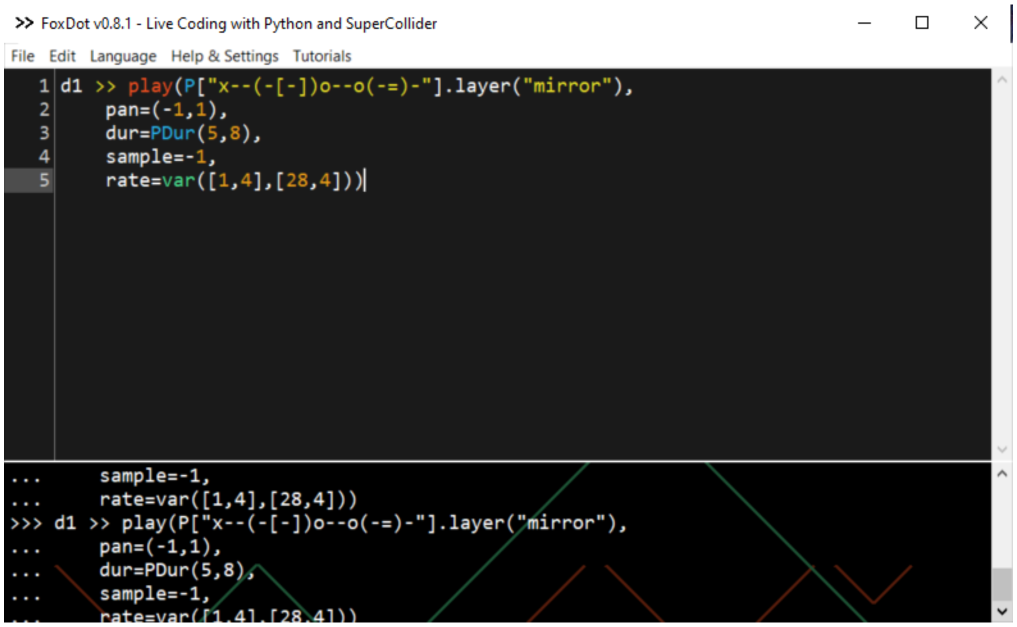
FoxDot [10] is a musical live coding language developed in 2015. It is based in Python, and makes use of the sound synthesis engine SuperCollider to produce music. One of its main appeals is that all code written in FoxDot is parsed by Python, meaning anything possible in that language is possible in FoxDot. Not only does this give a solid starting point for designing the language, but it also makes the language instantly familiar to anyone who has used Python before.

One of the primary focuses in FoxDot is on its method of tracking sequences called patterns. Everything in the language, including notes and how they are played, can be represented with patterns. There are numerous methods for easily creating, manipulating, and combining patterns. Many of these methods are facilitated by the Python's ability to allow for the definition of functions called by using special characters such as indexes and arithmetic. Other methods are more complicated and require calling functions which may take several parameters.

Although FoxDot's patterns are robust and powerful, its strong focus on them makes certain tasks more complicated. For example in order to specify the length of notes, two patterns are given to a function, one corresponding to the pitches of the notes and one corresponding to the durations of the notes. If these patterns do not have the same length, elements of each pattern are paired together until one pattern reaches its end at which point that pattern goes back to its beginning but the other pattern continues. This can be useful for creating abstract sounds but it can make creating long melodies very difficult.

```
d1 >> play(P[0, 2, 4],
    pan=(-1,1),
    dur=[2, 1, 1])
```

In the example above, the notes 0, 2, and 4 are played continuously, and the durations in the dur argument are applied to each of the notes. Since there are the same number of durations as notes each note will always have the same duration. However, there are only two pan values meaning the first note will have a pan of -1, the second will have a pan of 1, the third will cycle back to -1, and the second time the first note is played it will have a pan of 1. This kind of asynchronicity is one of the defining attribues of FoxDot and allows for the creation of unique and interesting music.


>> FoxDot v0.8.1 - Live Coding with Python and SuperCollider
File Edit Language Help & Settings Tutorials

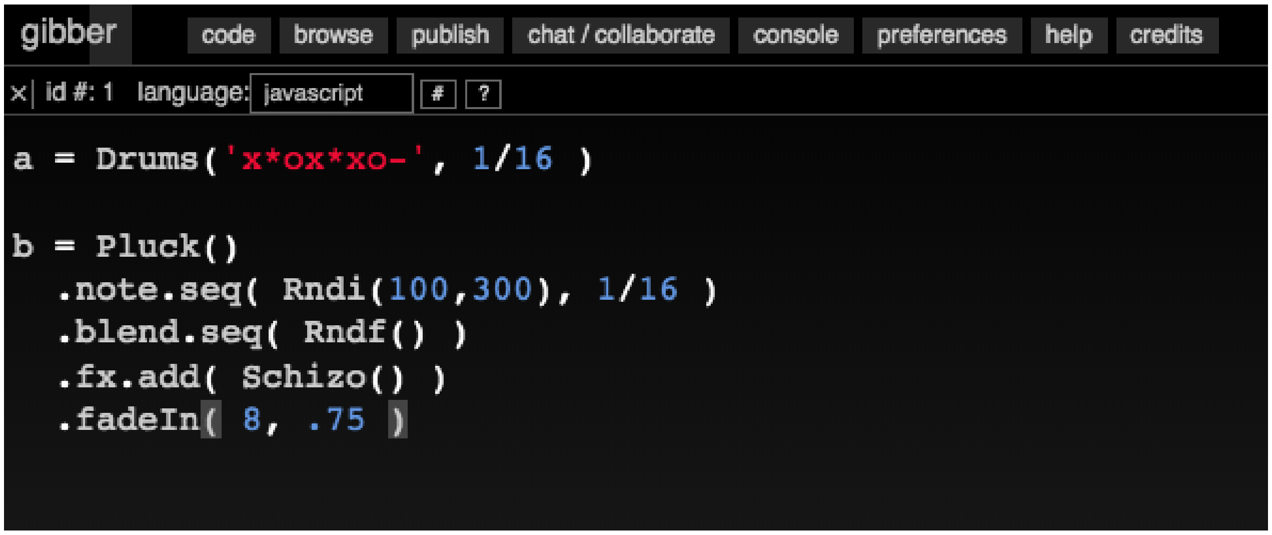
```
d1 >> play(P["x--(-[-])o--o(-=)-"].layer("mirror"),
    pan=(-1,1),
    dur=PDur(5,8),
    sample=-1,
    rate=var([1,4],[28,4]))
```

```
...     sample=-1,
...     rate=var([1,4],[28,4]))
>>> d1 >> play(P["x--(-[-])o--o(-=)-"].layer("mirror"),
...     pan=(-1,1),
...     dur=PDur(5,8),
...     sample=-1,
...     rate=var([1,4],[28,4]))
```

Figure 2: The FoxDot interface

3.2 Gibber

Gibber [21] is is a live coding language created by Charles Roberts that allows for both audio and visual creations. Its uses gibberish, a JavaScript audio API created by Roberts to produce the audio, and WebGL to create its visuals. Because it runs through JavaScript, Gibber is entirely accessible through any web browser[1].



```
a = Drums( 'x*ox*xo-', 1/16 )

b = Pluck()
  .note.seq( Rndi(100,300), 1/16 )
  .blend.seq( Rndf() )
  .fx.add( Schizo() )
  .fadeIn( 8, .75 )
```

Figure 3: The Gibber Online interface

[1]www.gibber.cc

Gibber was designed with the goal of allowing live coding through JavaScript syntax. Because it runs through JavaScript syntax, it is more welcoming to those with JavaScript experience. As seen in figure 2, Gibber stores sound patterns in variables. Sound patterns generally take the form of function calls, which call the sound the user wants to play, along with any modifications [21]. Gibber variables can be played per line, or can be called through an additional function. Gibber has a variety of instruments to play the notes and chords, and also allows for text to speech functionality. Variable in gibber also have the ability to be modified by modulation and effects.

3.3 Hydra

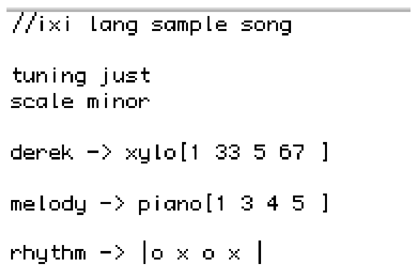
Hydra [8] is a visual live coding language created by Olivia Jack in 2017 and inspired by analog modular synthesizers. This language was built to be used on the browser, using JavaScript and WebGL. Hydra can be written from the online text editor, from the Atom text editor using the `atom` plugin, or by installing the open-source project and running it locally [8]. The design of this language makes live performance very simple, both as an individual performance and as a complement to an audio experience. Hydra is used extensively in the Algorave scene.

Analog modular synthesizers follow the idea that each module completes a specific task, modules are then interconnected, and finally produce complex outputs that might not be possible for a single module to accomplish. Essentially, Hydra follows this modular inspiration in the sense that given an input, it is chained through a series of functions, and eventually produces interesting visual outputs. Hydra functions are therefore not very complex, which allows people with little to no programming experience to have the chance to create very interesting images with it. This means that they do not need to know much about the complexities of JavaScript. A very simple example of Hydra chained functions can be found below:

```
shape(3) // make a 3 sided shape
.scale( // control triangle size following a given function
    () => Math.sin(time) // sine wave based on the current time
)
.rotate() // rotate 10.0 degrees constantly
.out()
```

3.4 ixi lang

Ixi lang is a music live coding language created by Thor Magnusson, a professor at University of Sussex. It was built on top of SuperCollider, a platform for audio synthesis and algorithmic composition, and uses it to create its sound. Because of this, nearly all of the functionality of SuperCollider is accessible directly from ixi lang [11]. Ixi lang can be downloaded as a basic application or as an extension for SuperCollider. It is part of "Ixi Audio", a series of free programs surrounding the design and creation of music. Ixi lang was designed to "free performers from having to think at the level of computer science, enabling them to engage directly with music"[12]. Because of this, it is a good introductory language that is also capable of creating complicated music by those with extensive experience or knowledge of music. "The goal was to be able to create a tune with rhythm and melody within a few seconds from the performance starting."



```
//ixi lang sample song

tuning just
scale minor

derek -> xylo[1 33 5 67 ]

melody -> piano[1 3 4 5 ]

rhythm -> |o x o x |
```

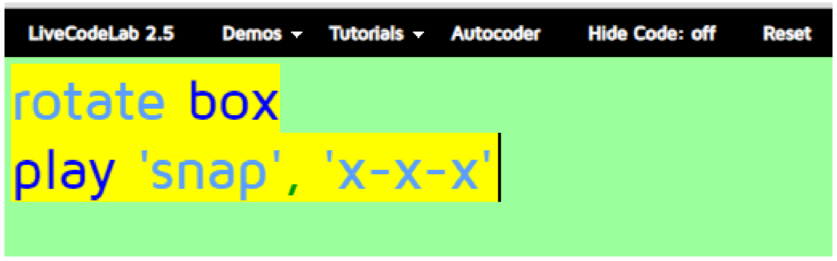
Figure 4: An example of ixi lang code

From a syntax standpoint, ixi lang was made with the aim to "provide a highly simple syntax" in order to keep it intuitive and inviting. [12] Within the language, the user names sequences of notes attached to an instrument which can then be played (as can be seen in figure 3). As These named sequences can attach to effects and filters to provide additional modifications to their sound. Additionally, you can create drum rhythms and patterns with a ranging of complexities.

3.5 LiveCodeLab

LiveCodeLab is a web-based live coding environment for music and visuals [4]. It was created by Davide Della Casa and Guy John, and LiveCodeLab 2.0 was released in 2014[4]. The environment is written in a combination of Javascript and Coffeescript [4]. The visuals are displayed using the Three.js library, and the audio is played with the Buzz.js library [1].

One of the main focuses of LiveCodeLab's design is that it is intuitive to use. Rather than be built off of another language's existing syntax (like JavaScript or Python), it uses a custom grammar built in Peg.js that is designed to be usable without a significant coding background [4]. It focuses on the ability to make audio and visuals with few lines of code. This is especially apparent in the visual component, where short keywords like "box," or "ball" are used to quickly make shapes that can then be edited. These keywords allow for the code written to have an immediate impact. The language uses a similar approach for audio, using the “play” keyword followed by a sample and a pattern. The patterns give a visual indicator of how the pattern will play as you are writing it, further tying into the idea of simple but impactful syntax [4]. In addition to the syntax being intuitive, the editor is also simple to use. It automatically processes your text as you write it, and is simply the text overlaid onto where visuals are shown. This removes the need for start and stop buttons or key commands, which can allow a new user to see the immediate impact of their changes.



```
rotate box
play 'snap', 'x-x-x'
```

Figure 5: Text on the LiveCodeLab Editor

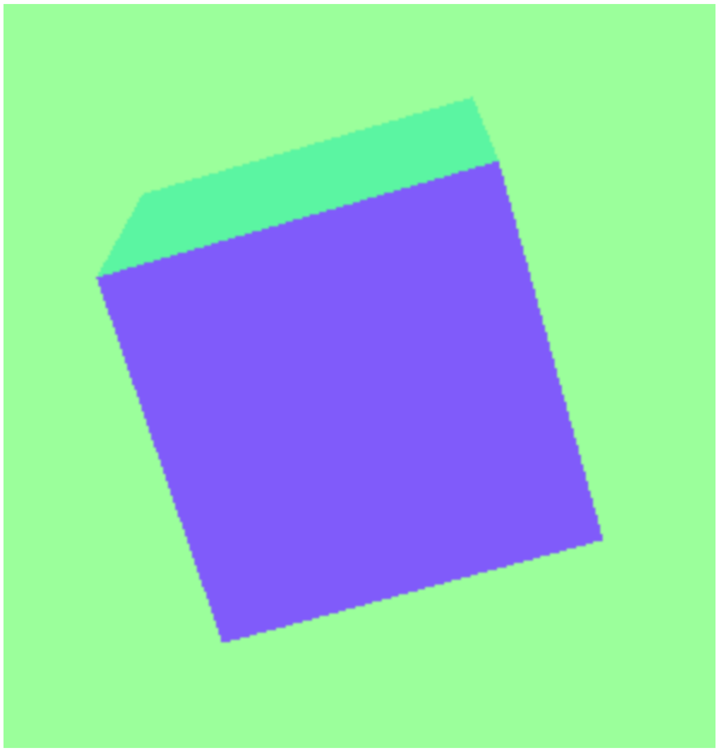


Figure 6: The box generated by the above example

The above section of code is a simple example of something that a user can do in LiveCodeLab. It creates a simple rotating cube (seen above), and plays the snap sound with the provided pattern. The ”snap” sound is a sample, and will continuously loop with the input pattern of ”x-x-x”. The sound plays on each ”x” in the string and does not play on each ”-”. This text is written over the background of the output, which appears in the center of the screen. The editor also offers pre-written demonstrations and tutorials from the drop-down menus in the navigation bar. These will load the content into the editor. The Autocoder does a similar thing, except that it automatically generates the code that will be run. This feature can help the user see what can be done in the language beyond potentially simple examples.

3.6 Orca

Orca [19] is a minimal and experimental live coding language created by Hundred Rabbits in February 2018. The goal of this language is to create procedural sequencers that can control various DAWs[2] such as Ableton Live or Renoise through different types of messages including MIDI[3]. Additionally, Orca has the capability of sending messages to other audio synthesis platforms such as SuperCollider [16], which is widely used in the Algorave, notably to serve TidalCycles.

Hundred Rabbits describe Orca as an *esoteric* programming language [19], and its syntax and features certainly prove this. In Orca, each letter of the alphabet is an operation that takes some type of input and outputs the result of performing that operation on the input. For example, the letter *A* performs an *add* operation and takes 2 inputs, and the letter *M* performs a *multiply* operation. Figure 3.6 shows the function definitions, the inputs and the outputs on the Orca editor.

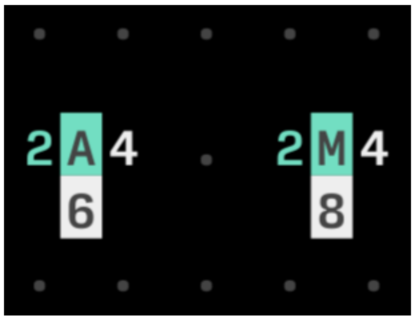
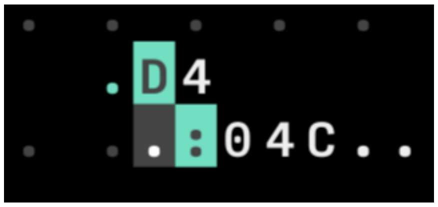


Figure 7: Add and Multiply Operators in Orca

There are multiple operations that create *bang* events, which are necessary to send messages as MIDI, UDP[4], or OSC[5] messages depending on the specification by a character. Figure 3.6 shows an example that uses the *D (delay)* operator to send a bang event every 4 frames, where the rate of the frames is defined by the tempo. At every bang event, a message can be sent. The colon character specifies that a MIDI message will be sent at every bang. The MIDI operator takes three required inputs: the channel, the octave and the

[2]Digital Audio Workstations
[3]Musical Interface Digital Instrument
[4]User Datagram Protocol
[5]Open Sound Control

note. Ultimately, the code on figure 3.6 will send a message to play a C4 note through channel 0 every 4 frames.

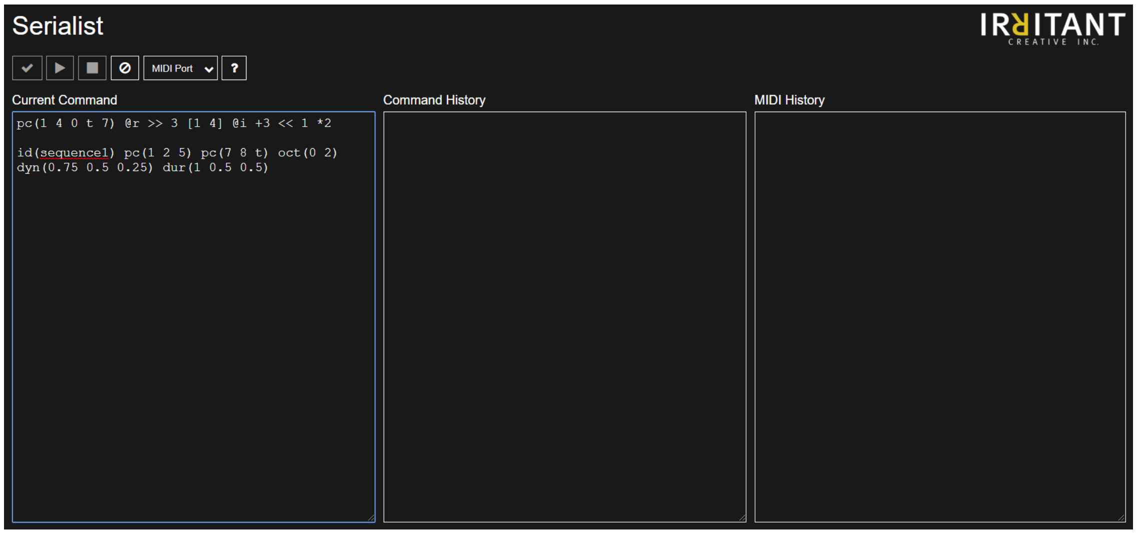


```
.D4
..:04C..
```

Figure 8: An Orca MIDI Message

3.7 Serialist

Serialist [24] is a web-based musical live coding language created by Tony Wallace. It is designed to output MIDI messages to a DAW or software synth. It mainly consists of labeled lines which can contain pitches, octaves, dynamics, and durations. Each of those attributes can be written as a sequence of numbers and modified with one or more of many transformations. Each of the sequences of attributes is traveled through linearly and combined with other attributes to create a single note which is output in the form of a MIDI signal. Serialist makes use of Peg.js to transform input text into a data structure understandable by the program. Each of the different types of attributes has all applicable transformations applied to it, then the result is stored in a labeled list for the program to access. Serialist is a relatively simple language, but it is effective at doing what it does.



```
pc(1 4 0 t 7) @r >> 3 [1 4] @i +3 << 1 *2

id(sequence1) pc(1 2 5) pc(7 8 t) oct(0 2)
dyn(0.75 0.5 0.25) dur(1 0.5 0.5)
```

Figure 9: The Serialist Online Interface

3.8 TidalCycles

While it does not play the sound internally, it generates either MIDI or OSC messages. These messages are then sent to a target application to play sound. This gives the language flexibility to be edited and run on the spot. It is often written in the Atom editor with the TidalCycles plugin to control compilation and output, but there are implementations in other IDEs. TidalCycles pattern syntax is very popular, and a mini-notation created for the browser[20] has contributed to this, allowing for tidal-like patterns to be used more extensively. This browser-based project allows tidal-like patterns to be used within the browser for either sound generation or visuals.

While the language is built on Haskell, it does not rely on knowledge of this base language to use. Instead the language offers extensive installation instructions, user documentation, and a tutorial. The tutorial allows a user to start learning the language without previous experience in live-coding by giving a starting place to get acclimated to both the syntax and the Tidal environment. In addition to the tutorial, the documentation is in-depth, and describes both syntax and core ideas behind the language such as cycles and patterns. While the tutorials are not integrated into the platform due to the nature of the language, they still give new users a starting point to work from. An example of sound generation with TidalCycles is through SuperCollider. This is based on 16 connections to the sound generator, that are named `d1` through `d16` by default. Patterns are then defined and played in loop in a span of predefined time. For example, the following three lines of code would play in the same amount of time, some at a faster tempo than others:

```
d1 $ sound "bd sd"
d1 $ sound "bd sd hh cp mt arpy drum"
d1 $ sound "bd sd hh cp mt arpy drum odx bd arpy bass2 feel future"
```

Patterns can also be created within patterns, and there are many types of patterns including layered patterns, one step per cycle patterns, or repeated patterns.

4 Language Design

A core part of our initial design choices were based around making our language intuitive for non-coders. It was essential to keep the syntax of the language intuitive to newcomers, which meant we had to focus on making the language readable and welcoming. We wanted new users to be able to quickly and effectively make music, regardless of their prior experience with music or programming. Because of this, we chose to shy away from using traditional programming syntax. Our focus was on allowing anybody who has just walked into a performance to understand the correlation between the code and the music. This meant using simple keywords words wherever possible and directing the user to read from left to right, which is also how Phad builds the sounds. The exploration of existing live coding languages extensively guided the design of Phad. Table 1 displays some of the main features of existing languages that aligned with the initial goals of the project, and that we aimed to integrate in the development of Phad.

Language	**Relevant Features**
FoxDot	Sequence creation and manipulation
Gibber	Runs on the web Parts are named and declared through variables Large variety of instruments and usable sounds
Hydra	Runs on the web Runs on Atom Atom plugin can be used concurrently with other plugins
ixi lang	Runs on the web Readable through use of whole words Flexibility with assigning values
Livecodelab	Runs on the web with a simple user interface Accessible syntax
Orca	Minimal Experimental Visual cueues for sound
Serialist	Built with PEG.js Minimal syntax
TidalCycles	Intuitive syntax Based on patterns that are then passed to modifiers and instruments

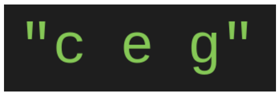
Table 1: Highlighted Features from Existing Languages

Based on these existing feature, we were able to design our own language features and syntax. This section describes the finalized language design of Phad. We will describe each object that was defined in the code, including *Sequences*, which are formed by *Groups*, and to which you can add *Modifiers*. Once a sequence and its modifiers are defined, it can be passed to an *Instrument*, and many *Filters* can be applied to these. This section contains an overview of all type definitions, but further detail can be found in Appendix 10.1.

4.1 Type Definitions

4.1.1 Sequences

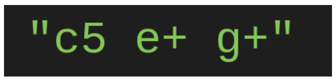
To play music with Phad, the first item that needs to be typed is a Sequence. In Phad, Sequences are always wrapped in quotations. A Sequence consists of a series of Groups, which are a representation of notes. A Group can consist of a Note represented with letters from A to G, a number representing a note position in a scale, a Rest, or a predefined percussion sound. A Group can also consist of any combination of Groups, such as Chords, Random, or any other nesting of these. Figure 4.1.1 show a simple example of a Sequence with the notes `C, E,` and `G` in the default octave (fourth octave).



```
"c e g"
```

Figure 10: Example of a Basic Sequence in the Default Octave

To modify the default octave, Sequences can contain octave modifiers such as `+, -` or a number representing the octave. For example, Figure 4.1.1 shows the pattern in Figure 4.1.1 transposed to the fifth octave.

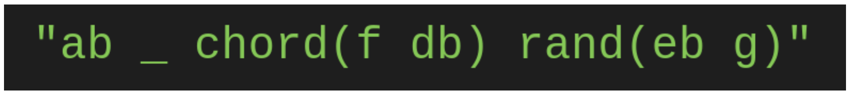


```
"c5 e+ g+"
```

Figure 11: Basic Sequence in the Fifth Octave

As mentioned before, Sequences can also be formed by more complex Groups, or Groups of Groups. For example, Figure 4.1.1 shows an example of a Sequence that contains Notes with accidentals[6], a Rest, a Chord and a Random selection between two notes.

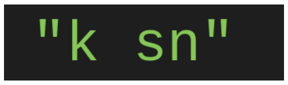
[6]Sharps and flats



```
"ab _ chord(f db) rand(eb g)"
```

Figure 12: Sequence with Different Types of Groups

Another use of Sequences is the drums. Figure 4.1.1 shows a drum beat formed by a kick and a snare.



```
"k sn"
```

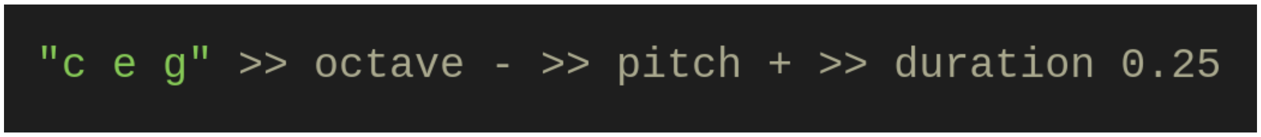
Figure 13: Sequence with a Drum Beat

4.1.2 Modifiers

Once a Sequence is defined, Phad allows users to add Modifiers to it. A Modifier is anything that can affect a previously defined Sequence, and is added to the right of a Sequence preceded by a >>. Below is a list of the Modifiers that have been implemented:

- **Octave:** changes the octave of the Sequence
- **Pitch:** changes the pitch of the Sequence
- **Duration:** changes the duration of each Group of the Sequence by multiplying the default value by the given fraction or decimal
- **Stutter:** repeats each of the highest level groups a given number of time
- **Scale:** valid when the Sequence is formed by numbers, and changes the scale of the Sequence

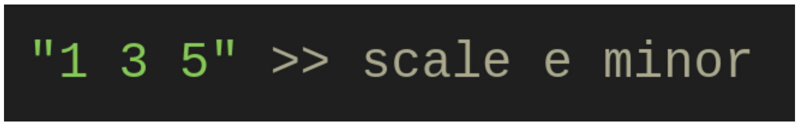
Modifiers can be applied to any Sequence. For example Figure 4.1.2 shows an example of a simple Sequence that is transposed an octave down, a half step up, and where each Group of the Sequence has a duration of 1/4.



```
"c e g" >> octave - >> pitch + >> duration 0.25
```

Figure 14: Simple Sequence with Modifiers

Figure 4.1.2 shows an example of the Scale modifier with a number Sequence. In this example, we have the notes E, G and B in the fourth scale represented as numbers. They correspond to notes 1, 3 and 5 of the E major scale. The number Sequences make it very easy to change keys using the Scale modifier.

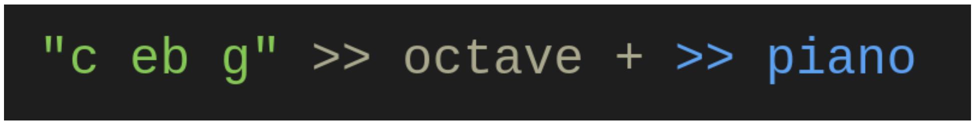


```
"1 3 5" >> scale e minor
```

Figure 15: Number Sequence with Scale Modifier

4.1.3 Instruments

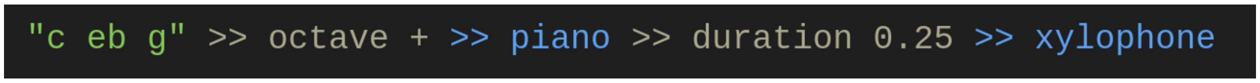
Once a Sequence and its Modifiers have been defined, it is time to make them generate sound. This is where the Instruments come into play. They are added after any Modifier, and once played will cause the sound to loop infinitely. Figure 4.1.3 shows a simple Sequence played in the fifth octave that gets played with a piano sound.



```
"c eb g" >> octave + >> piano
```

Figure 16: Simple Sequence Played with Piano

Phad is parsed left to right, meaning that a single Sequence can have many different sounds depending on the order of its Modifiers and Instruments. If we were to add more Modifiers to the right of the previous example and then another Instrument, we would be able to hear the same Sequence differently. Figure 4.1.3 shows the sequence from Figure 4.1.3 with an additional Duration Modifier and a xylophone Instrument. This means that `"c eb g"` will play with the piano sound at the default speed, and concurrently the same Sequence will play four times faster with a xylophone sound.



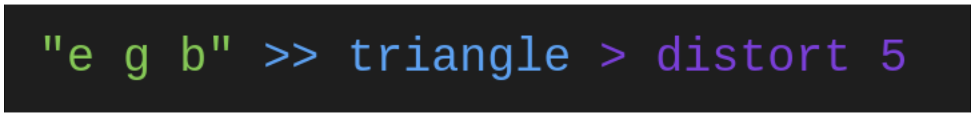
```
"c eb g" >> octave + >> piano >> duration 0.25 >> xylophone
```

Figure 17: Sequence with Multiple Modifiers and Instruments Played Concurrently

A full list of Instruments can be found in Appendix 10.1.14.

4.1.4 Filters

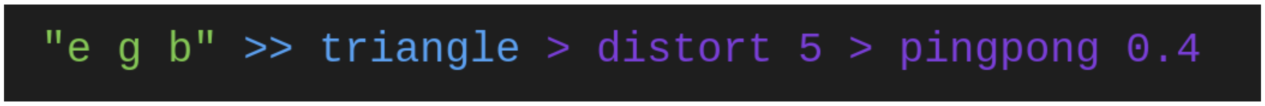
Filters modify the sounds generated by Instruments. As mentioned earlier, Phad is parsed from left to right, so any filters applied to an Instrument should come after the name of the Instrument, as shown in Figure 4.1.4. This example plays the Sequence with the sound of a distorted piano.



```
"e g b" >> triangle > distort 5
```

Figure 18: Simple Sequence Played with Distorted Piano

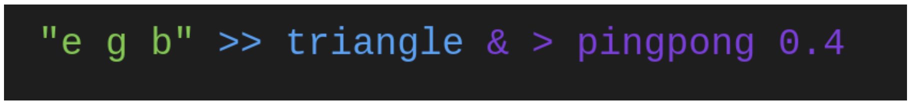
Phad allows users to add multiple filters at once, as seen in Figure 4.1.4. In this case, two filters are applied and the outputted sound consists of just the filtered instrument. The piano sound is first distorted, and then the *pingpong* filter is applied, causing the sound to echo between the left and right speakers with a specified delay.



```
"e g b" >> triangle > distort 5 > pingpong 0.4
```

Figure 19: Two Filters Applied to an Instrument

However, it is also possible to apply filters to instruments and play both the original and filtered instrument in the outputted sound using the `&` character as seen in Figure 4.1.4. In this example, the generated sound will consist of a sound with the *pingpong* sequence played concurrently with the original unfiltered sound.



```
"e g b" >> triangle & > pingpong 0.4
```

Figure 20: Original and Filtered Sound Played Concurently

A full list of filters can be found in Appendix 10.1.16.

4.1.5 Parts

The combination of all the previously defined types form Parts, the main type of our language. Figure 4.1.5 breaks up the sections of a Phad Part.

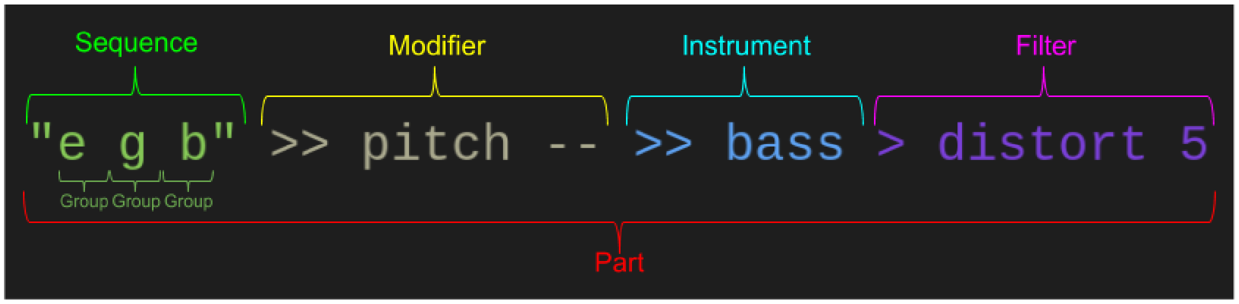
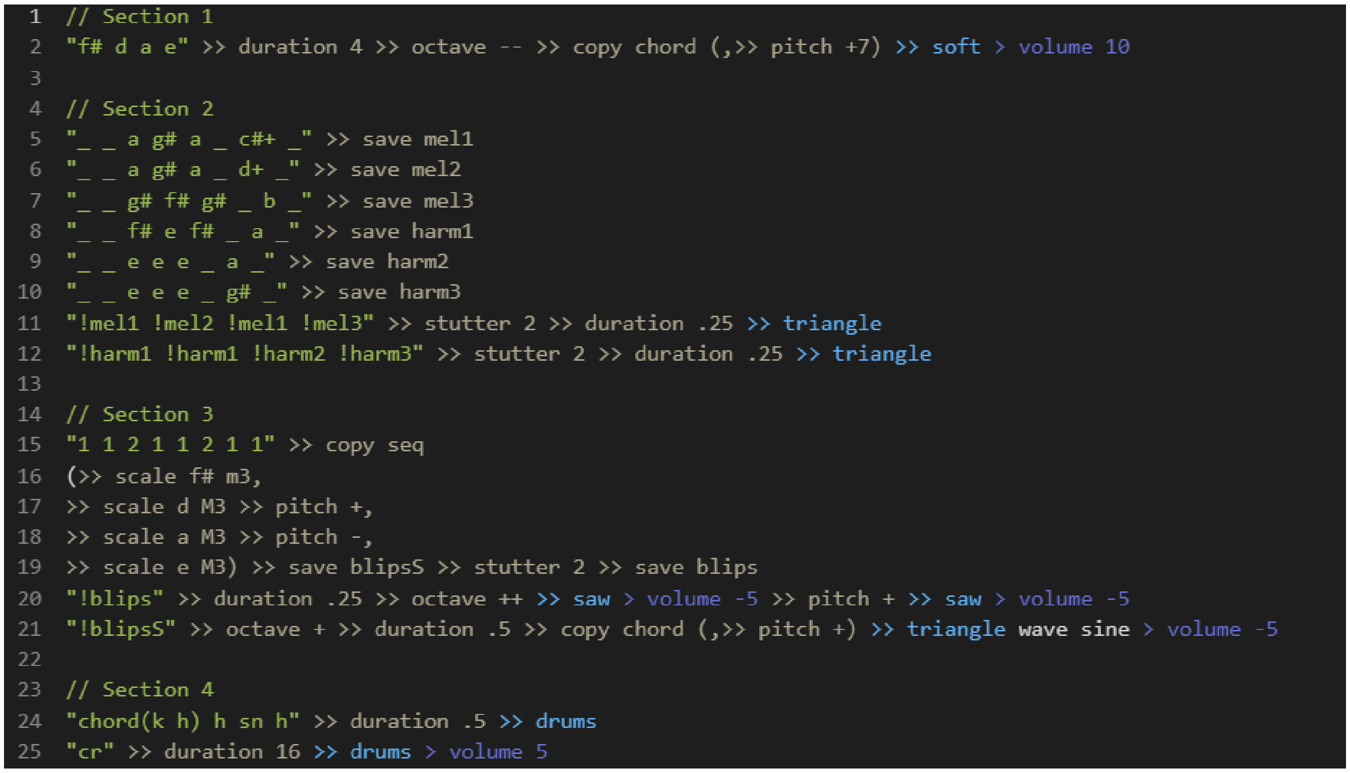


Figure 21: Sections of a Phad Part

4.1.6 Example

The following is an example of a song written in Phad.



```
1  // Section 1
2  "f# d a e" >> duration 4 >> octave -- >> copy chord (,>> pitch +7) >> soft > volume 10
3
4  // Section 2
5  "_ _ a g# a _ c#+ _" >> save mel1
6  "_ _ a g# a _ d+ _" >> save mel2
7  "_ _ g# f# g# _ b _" >> save mel3
8  "_ _ f# e f# _ a _" >> save harm1
9  "_ _ e e e _ a _" >> save harm2
10 "_ _ e e e _ g# _" >> save harm3
11 "!mel1 !mel2 !mel1 !mel3" >> stutter 2 >> duration .25 >> triangle
12 "!harm1 !harm1 !harm2 !harm3" >> stutter 2 >> duration .25 >> triangle
13
14 // Section 3
15 "1 1 2 1 1 2 1 1" >> copy seq
16 (>> scale f# m3,
17 >> scale d M3 >> pitch +,
18 >> scale a M3 >> pitch -,
19 >> scale e M3) >> save blipsS >> stutter 2 >> save blips
20 "!blips" >> duration .25 >> octave ++ >> saw > volume -5 >> pitch + >> saw > volume -5
21 "!blipsS" >> octave + >> duration .5 >> copy chord (,>> pitch +) >> triangle wave sine > volume -5
22
23 // Section 4
24 "chord(k h) h sn h" >> duration .5 >> drums
25 "cr" >> duration 16 >> drums > volume 5
```

Figure 22: A song written in Phad

Section 1 creates a bass line. Line 2 starts with an outline of the structure of the bass line, which is extended and brought down two octaves. Next, each note is made into a chord with the note a perfect fifth above it. To achieve this, the copy modifier makes each note into a chord of two notes. The first note in the chord remains unchanged since there is no space between the open parenthesis and the first comma. The second note is raised by a perfect fifth by the pitch modifier. The sequence is finally played by the soft instrument.

Section 2 defines a melody and harmony. Lines 5-10 save some sequences, which is done both for the sake of readability and because some of the sequences are reused. On lines 11 and 12, the saved sequences are put together, the stutter modifier is used to make each saved sequence play twice, the notes are sped up, and the result is played by the triangle instrument.

Section 3 defines two more melody-like parts. A sequence of notes is created using numbers. It is then sent through the copy modifier which makes the sequence repeat four times, being converted to a different key each time based on the bass line in Section 1. The pitch of the notes is also modified in some sections to keep the range of the notes consistent. The result of this is saved, then stuttered and saved again under a different name. Lines 20 and 21 demonstrate two different ways of doing the same thing. On both lines, once the duration and octave are set, the sequence is played as is and with the pitch raised by one. Since the scales chosen on lines 16-19 are all triads, this creates a nice harmony. On line 20, to achieve this goal, the sequence is played by the saw instrument, then the pitch is raised by one and played by another identical instrument. On line 21 however, the copy modifier is used again to play the notes as they are and with the pitch raised by one.

Section 4 defines two drum lines. The first is a driving groove which makes use of the chord group to play the kick and high hat at the same time. The second is a crash symbol which plays at the start of each time through the bass line.

5 System Design

Phad was designed with the goal of creating an interface that would be easily accessible to everyone, and that would require no setup at all. As mentioned earlier, the best way to achieve this was to create a web-based application. Our goals lead us to pick JavaScript as the main language for the project. Different JavaScript libraries and technologies were used for a variety of parts in this system. Although multiple technologies were considered, the following section will describe the chosen technologies for each endpoint of the Phad system.

5.1 Parsing and Sound Generation

5.1.1 PEG.js

PEG.js [13] is a JavaScript parser generator that makes rule definition relatively easy, allowing for the creation of complex parsers with few lines of code. It is based on the parsing expression grammar (PEG) formalism, introduced by Brian Ford in 2004. PEG consists of certain parsing rules, such as sequences and ordered choices, as well as predicates which are fragments of code which can be run to check whether text should parse. It also outlines a method of labeling parsing rules, which are groups of rules which can themselves contain other labeled rules, allowing for recursion. This tool was chosen for the parsing of Phad due to its extensive documentation and the fact that part of the team had experience with it, all of which would minimize the difficulties with it.

5.1.2 Tone.js

"Tone.js is a Web Audio framework for creating interactive music in the browser." [14] Tone.js contains a large number of tools for creating interactive music such as event scheduling, synthesized instruments, samplers, sound effects, etc. The flexibility of this framework and the amount of documentation lead us to choose it as our sound generation tool.

5.2 Database Storage

Although many storage options were considered, we decided on Database storage for Phad. This would require the creation of a server, a database schema, and a hosting service for both.

5.2.1 Express

Express [23] is a Node.js framework that makes the creation of servers very easy. It can easily be integrated with other Node.js packages, making things such as database integration or communication with the client very smooth and easy to implement. This framework made the most sense, as we wanted to stick to using JavaScript for the entirety of the app, and the member in charge of creating the server already had experience with this. This decision would allow us to focus more on the user interface and the language portion of the application.

5.2.2 PostgreSQL

PostgreSQL is an open source relational database system that has been in development for over 30 years. It was initiated in 1986 as part of the POSTGRES project at the University of California Berkeley [18]. It uses a combination of SQL and other features, resulting in an extensive list of features that simplifies the implementation of a database schema. Additionally, PostgreSQL allows for easy hosting through the use of a free tool like Heroku.

5.2.3 Heroku

"Heroku is a cloud platform that lets companies build, deliver, monitor and scale apps." [22] It can provide free hosting of up to five applications, and all of these can contain database integration. Because the server code would include database credentials that were not to be shared with the world, and GitHub Pages only allowed hosting of public repositories, Heroku was the best option for hosting the server and the database.

5.2.4 Google Authentication

It was decided that Phad would store user information and allow them to save, edit, and delete songs created in the interface. This required the creation and storage of new users. Because we wanted to avoid the security risks that would come with storing passwords in our database, we thought it made more sense to use a third-party authentication system such as the Google OAuth[7] protocol. Some members of the team had experience implementing Google oAuth along with React, which could accelerate the development.

[7] Open Authentication

5.3 User Interface

5.3.1 React.js

React.js [5] is a JavaScript library created and maintained primarily by Facebook since 2013. It is currently among the most popular frameworks used to create user interfaces. React code is formed by *components*, which can be very easily reused throughout an application. React could also give us the ability to use components from other packages such as Material-UI [15], which could minimize our development of the site styling. React also has a built-in function called *fetch* that makes it very easy for the client to communicate with a server. Although only one member of the team had experience with React, we thought it could be useful for the overall creation of the interface.

5.3.2 Monaco Editor

The Monaco Editor is a free and open source code editor created and maintained primarily by Microsoft developers [17]. It comes with many implementable features such as syntax highlighting, error highlighting, cursor tracking, line numbering, auto complete, and other preset and custom commands. These features heavily guided us in our decision, as we thought it would make more sense to search through documentation than to try to implement such features ourselves. Additionally, we thought the Monaco Editor could provide a great interface for programming music within our app.

5.3.3 GitHub

Git is a version-control system designed to coordinate coding among programmers and track changes to source code. It is free to use and open source. This tool allows team members to stay up to date on the code updates, making it easier to collaborate and develop a project. GitHub [6] is among the most popular Git hosting services, and is also a tool all members of the team had used before. This lead us to choose it for the Phad Git repository. Additionally, it allows web applications saved in GitHub public repositories to be easily hosted through GitHub Pages [7].

6 Development

6.1 Parsing and Sound Generation

The overall backend system architecture consists of a parser made in Pegjs which returns a structure of JavaScript classes which use Tone.js to play notes.

The parser starts by dividing the input into lines. Each line's sequences, modifiers, instruments, and filters are parsed and made into Parts, one for each instrument. A Part contains all information necessary for some music to be played, including both the notes and the instrument they are being played on. The Part class has methods for starting and stopping the music it has stored.

Each line can be divided into an initial sequence and one or more modifiers. The initial sequence can contain note names, numbers, rests, drum hits, groups, and variables. All of these classes are guaranteed to contain certain methods allowing them to be played, stopped, and modified as necessary. Because of this they can all be used interchangeably, allowing for a great amount of versatility in how a user may construct their music.

6.1.1 Notes

Note names are represented with the Note class, and store the note which it plays as well as its length. Numbers are represented with the NumNote class which contains the note's number, its scale, its octave, and its length. Rests are stored in the Rest class which contains the length of the rest. Drum hits are stored in the Drum class which has the type of drum hit and a length. All of these classes contain methods for modifying their attributes and causing themselves to play.

6.1.2 Groups

There are several types of groups, each of which have their own class. These types are Chord, Random, Repeat, and Sequential. These classes all store a list of what groups they play as well as other necessary information to allow them to play appropriately. Their behavior is modified to allow them to do what they are supposed to. Chord plays all of its groups at the same time and stops when the longest group has finished, Random chooses one of its groups at random, Repeat plays its group a certain number of times, and Sequential plays all of its groups in order. These groups do not make use of internal memory to store information about how much of them has been played since this would prevent them from being able to be played twice at the same time. Instead, they return an object containing the information they need to know to continue playing from where they stopped, and their parent gives them that object the next time the group needs to be played.

6.1.3 Variables

Variables can be defined using the save modifier, and once saved can be referred to in place of a note. In order to accomplish this, a global object containing the names of user-defined sequences as keys and the sequences as objects is updated by the parser. When a user refers to a variable the parser checks whether a variable has been stored under that name and if it has it returns that sequence, otherwise it throws a parsing error.

6.1.4 Modifiers

After the initial sequence, any number of modifiers can be used. These are ways of modifying the initial sequence in a way that could be written out manually. The parser turns each modifier into a function which is executed on the initial sequence in the same order that the user typed them. There are some special modifiers, such as the save modifier mentioned earlier.

6.1.5 Instruments

Another special modifier is the player modifier which signals that the sequence should be played. It begins with the name of an instrument and is optionally followed by one or more filters. If there are filters, the parser first chains the filters together to connect to each other. Next, a Part class is constructed containing the proper instrument put through the appropriate filters and the part it is playing. When the parser has finished, the Part is returned and may be used to start and stop the music.

To play music, the Part class uses the start method. This method begins by finding the length of what it is going to play and rounding the current time to a multiple of that length. If the method is told to start playing immediately, the current time is rounded to the nearest multiple before the current time, and if it is not the current time is rounded to after the current time. The first note of the sequence is then scheduled to play at the rounded time, and each time a note plays the next note is scheduled to play. This continues until the stop method is called on the Part.

6.2 Database

6.2.1 Schema

The main goal of creating a database is to be able to associate users with their songs and vice-versa. A user entry consists of a unique identifier and a name. A song entry consists of a unique identifier, a title, and the song content. Because this is currently the only storage done in Phad, the database schema is very simple, as can be seen in Figure 6.2.1. To ensure that the songs all have a unique identifier, the database schema has a trigger for every time a new song is stored which assigns the next value in the sequence `songID_seq` to the new song `songID`, overriding the value sent to the database. The title and content of the song are kept from the value sent to the database. Because the user and song relationship is a many-to-many relationship, it was easier to create a third table `UsersSongs` where each row is a pair of userID and songID. This allows for easier implementation of collaborative song editing in the future, as a single song can have multiple owners.

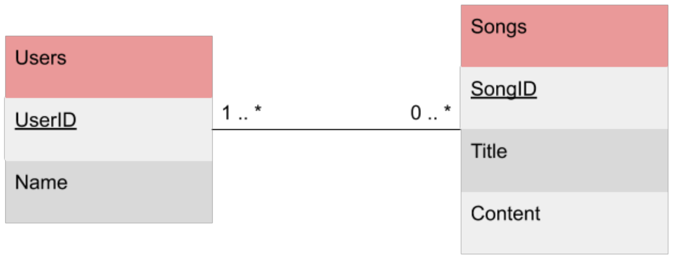


Figure 23: Database Schema

6.2.2 Server and Database Integration

We created an Express server to be able to communicate with the database from the React client. Similar to the database, this server is very simple and only contains a small number of POST requests. The main role of these POST requests is to pass the body of the request as the information to be stored in the database. Figure 2 shows the requests of the server and what each of them does.

Route Name	**Type of Request**	**Body parameters**	**Description**
`"/saveuser"`	POST	`email` `name`	Checks if a user exists in the database based on their email. If they don't, create an entry in the *Users* table. If they do, update the user if necessary.
`"/savesong"`	POST	`songTitle` `songContent` `email`	Adds a new song to the *Songs* table. Adds a new user-song pair in the *User-Songs* table.
`"/updatetitle"`	POST	`songTitle` `songID`	Given the songID and a new title for a song, updates the corresponding entry in the *Songs* table.
`"/updatecontent"`	POST	`songContent` `songID`	Given the songID and a new content for a song, updates the corresponding entry in the *Songs* table.
`"/getallsongs"`	POST	`email`	Given a userID returns all songs associated with that user.
`"/gettotalsongs"`	POST		Returns all songs in the database.
`"/deletesong"`	POST	`songID`	Given a songID deletes the song from the *Songs* table and any associatioin of it from the *UsersSongs* table

Table 2: Highlighted Features from Existing Languages

Because the server would be hosted through Heroku and would communicate with the client hosted on a different server through GitHub Pages, it was necessary to find a solution around CORS issues.[8] The `cors`

[8]Cross Origin Resource Sharing manages cross-origin requests. This means resources that are passed from a singel resource can be rejected by another one for security purposes. CORS is an effective security protection when all resources come from the

npm package made it really easy, as all we did was define some of the origins we knew were going to be used, as can be seen in the following code snippet:

```
const app = express();
app.use(cors({
    origin: [
        'http://localhost:1234',
        'https://mqp-live-coding-language-design.github.io/mqp',
        'https://mqp-live-coding-language-design.github.io'
    ],
    credentials: true
}));
```

In this piece of code we start our app as an Express server, then we make use of the `cors` package to define the origins. The `localhost:1234` origin was used mostly for testing purposes, as port 1234 was the default port used by Parcel to host the app. The other two origins correspond to the link where our app is hosted through GitHub Pages.

6.2.3 Interaction Between Database and Client

The frontend of Phad is built primarily with React and complementary packages. Fortunately, React makes `POST` requests to the server very convenient with the use of the `fetch` function. This function is used for all requests to the server, since they all have very similar structures. The code below shows an example of a `POST` request to obtain the entirety of the songs in the database:

```
fetch('https://mqp-server.herokuapp.com/gettotalsongs', {
  method: 'post',
  headers: { 'Content-Type': 'application/json' },
})
.then((res) => res.json())
.then((data) => {
  setMySongs(data); // Function defined before to set the React state of that component
});
```

The `fetch` function follows the principle of JavaScript Promises. This ensures that each line of code happens syncronously, meaning that once the fetch is complete, the response will be parsed to `json`, and

same host, but it is necessary to find ways around this when this is not the case.

once this is done the json information is used within the application. In this case, the outputted data is used to call the `setMySongs` function, which sets the `mySongs` state in the `MySongs` component, for example. This component, as well as any others that allow any modifications to the database, are only rendered in the case where a user is authenticated. The `react-google-login` package makes it very easy to setup Google authentication on the client side of the application, by simply constructing the component seen in the code below, then responding accordingly based on user success or failure.

```
cookies.get('email')
  ? (
    <GoogleLogout
    clientId="TOKEN"
    buttonText="Logout"
    onLogoutSuccess={logout}
    />
  )
  : (
    <GoogleLogin
    clientId="TOKEN"
    buttonText="Login"
    onSuccess={oAuthSuccess}
    onFailure={logout}
    cookiePolicy="single_host_origin"
    />
  )
```

On success, a cookie with the user's unique identifier is created. This is the main way for the client to know if a user is authenticated or not, and for the server to identify the user. In the case where a user has successfully logged in and the cookie was properly set, we change the Login button to a Logout button. On failure, the cookie isn't set and nothing in the application changes. Ideally, it would be a lot more secure to set a cookie with a more secure identifier, such as a JWT token[9], which would then be decoded by the server to identify the user. Although this was attempted for several hours, security was not a main concern in this iteration of Phad, SO we decided to keep the server and database very minimal and to dedicate more time to other user interface and language features.

[9]JSON Web Token

Once a user is logged in, the React buttons, inputs and dropdown menus are set up to interact with the database, by fetching the paths defined in the Express server. For a song to be stored in the database, it needs to have a title and the PlayBox needs to have some content. One of the most important components of Phad is the PlayBox component, which is how the content of the editor is known at any point, allowing users to save songs without getting an ”Empty song” alert. This component contains the Monaco editor configuration, the logic to pass a string to the parser, and the play and stop Play functions. This component will be explained further in Section 5.3, the User Interface section. As long as the PlayBox component contains a string that the PEG can parse, and a title, the database will have the capability of storing the entire song for that user. Once a song is successfully stored in the database, it can appear in the list of songs for a user when doing the `"/getallsongs" POST` request, which is similar to the request mentioned earlier, with the exception that it only fetches songs associated with a particular user. The goal of `mySongs` is to have a list of song objects that a user can select from. This list allows users to load existing songs from the database onto the current editor, to edit and to delete existing songs. All of the mentioned actions were programmed so the user can interact with a button or any other component visible from the interface, and the database will update with the use of the `fetch` function. It is important for the user to be able to interact with recently saved songs, which is why we make sure the `"/getallsongs" POST` request is done constantly, with the use of the `useEffect` built-in React hook. `useEffect` is a React hook similar to `onComponentDidMount` which runs at every re-render of the application unless specified differently. For example, the following code assures that `useEffect` runs one time only by passing a callback function as the first parameter and `[]` as a second paramer. We only want this to happen every 6 seconds or so, so we define a `setInterval(...)` that makes a fetch to the server only every 6 seconds rather than at every re-render.

```
useEffect(() => {
  const interval = setInterval(() => {
    getTotalSongs();
  }, 6000);
  return () => clearInterval(interval);
}, []);
```

6.2.4 Use of the Database in Collaborative Performance

The Algorave was a very important showcase of our semester of work, so we prioritized building an effective collaborative environment. Initially, the idea was to use the Monaco boxes to create a VS Code plugin, and use a collaborative editing plugin this way. However, Tone.js uses the Web Audio API to generate sound, which is not something we could replicate in VS Code. We then migrated into the idea of using our database to create this collaborative portion of the project. We were able to somewhat succesfully develop the collab tab of Phad: a tab that generates a maximum of 4 view-only editors and plays the sound in each of them. Because there is one single audio context in the entire React App, the tempo and timing of the audio is properly synced. The workflow of the collaborative tab from the perspective of the performer will be discussed further in Section 5.3: User Interface. As for the logic of this tab, we make use of the `"/gettotalsongs"` `POST` request to generate a list of all the songs in the database no matter which user is logged into the application, or if they even are logged in. Once songs are selected from this list, they appear on the screen as seen in Figure 29: 4 editors in collab screen. The basic logic is that every few seconds the current value of the editor is compared to the value of the parts playing. If these two do not match, meaning a user has updated a song in the database that is being visualized in the collab tab, it is necessary to call `stop()` and then `start()` again to update the generated sound. For each PlayBox component that is created upon song selection, we needed to programmatically parse the entire string in the editor, and call `stop()` and `start()` appropriately. This was difficult, as we had relied on user interaction with the *start* and *stop* buttons or key commands to do this, which always made sure that the editor was ready, and that the Tone.js context was activated. Although we found a solution with many conditionals that worked, it would be significantly better to think about how to make use of React state or another more reliable way of keeping track of change to trigger `start()` and `stop()` properly. We knew this was not the most reliable solution, so we tested the interface multiple times before our performance. As usual, it worked during our practice times, but had a very obvious glitch towards the end of the performance that was most likely due to the way the list of parts playing is stopped and started.

6.3 User Interface

6.3.1 Routes

Phad was built with the goal of making live-coding accessible to people with no programming or music experience. For this reason, it was essential to introduce a way to teach users how to use it within the same application. We thought that a tutorial, a full documentation, and some exercises were essential for the user to evaluate themselves when using Phad. Additionally, we wanted to give the user independence when creating music with Phad, and this meant the ability to have the option to have a blank editor, as well as a collaborative environment within the application. The following React code snippet shows the 6 routes that the Phad system is built on:

```
<Route exact path="/" component={LandingPage} />
<Route path="/usertest" component={UTHomepage} />
<Route path="/playground" component={Playground} />
<Route path="/tutorial" component={Tutorial} />
<Route path="/exercises" component={Exercises} />
<Route path="/documentation" component={UserDocumentation} />
<Route path="/collab" component={Collab} />
```

The `react-dom-reroute` npm package made it very easy to define multiple routes for the application, allowing different components to be rendered based on the menu selection. The final result of each one of the tabs can be seen individually in the Results Interface Section.

6.3.2 Coding Interface

The Phad code editor made use of the `'@monaco-editor/react'` npm package. This package made it relatively easy to keep track of the value of the editor for sound generation, editor theme, toggling sound on specific lines of code, and autocomplete.

6.3.3 Editor Theme

The Monaco editor allowed for the theme of the editor to be easily defined. Because a Phad Part is formed by Sequences, Modifiers, Instruments and Filters, we only used 4 different colors to highlight each section, as can be seen in Figure 4.1.5. We were able to hard code all values for Instruments and assign the color *#61AFEF* to them. Anything that was not an Instrument, but was preceded by the >> characters, had to be a Modifier, so this logic allowed us to use the following Regular Expression to recognize a modifier: `/>>[^>"]*/`, and color it *#ABA58E*. Sequences were surrounded by quotations and were colored *#98B755*. Filters were preceded by either & > or > and were colored *#6871D7*. Monaco also made it very easy to change the font size of the editor as part of the theme, which was very useful during the Algorave, when we wanted to make sure the audience could see the code on the big screen.

6.3.4 Start and Stop

As mentioned earlier, the Monaco editor makes it very easy to keep track of its content and use it throughout the application. This allows the `start()` and `stop()` functions to work around this. First, the content of the editor is passed as one single string to the parser, which then parses the content line by line, as mentioned in Section 5.1: Parsing and Sound. The `start()` function goes through each Part of the list generated by the parser, and calls the appropriate Part class function with the same name, which makes sure to play the Part at the appropriate time by using Tone.js. Additionally, it uses React state to set the state of the currently running parts properly. This is done so that when `stop()` is called, it can easily look at the React `runningParts` array and call the Part class function `stop()` which also uses Tone.js to disconnect the instrument of the given part, which stops the sound entirely. This logic happens in the program when a user adds code to the Monaco editor, then clicks the *START* or *STOP* button, or presses *CTRL + Shift* to start or *CTRL + alt* to stop specific lines. This will be explained further in the following section.

6.3.5 Autocomplete

Autocomplete was developed for Instruments, Modifiers, Scales and certain parts of Sequences (chord and rand). Given a list of possible values for each one of these, we defined a set on conditions to match, then used the Monaco *suggestions* option to generate an output. For example, in the case where the user types a >> character, they are either adding a Modifier or an Instrument. Our autocomplete feature would suggest both, by including the following snippet in the Monaco theme definition:

```
% if (text.match(/.*>>[^>]*$/)) {
  return { suggestions: autocomplete.instrument.concat(autocomplete.modifier) };
% }
```

The same principle is followed for all other sections.

6.3.6 Collab

Performance is the most popular way to practice live coding music. Because of this, we wanted to build a collaborative enviroment that would allow the entire team to live code during the Algorave performance. As was mentioned in the previous section, it was decided that the database would play a key role in this tab of the application. This section will explain collaborative coding from the perspective of the performers, and how our team practiced and performed. The first step is for each user to log in and create a new song that they will be editing throughout the performance. Once the titles of these are communicated with the person in charge of sound, the correct songs are selected from the list that opens when clicking the *ALL SONGS* button in the right (view-only) portion of the collab tab. The sound generated will then update automatically based on the changes made by performers to the database. Each user can now freely update their selected song by clicking the *MY SONGS* button from either the playground tab or the left portion of the collab tab, then clicking on the *Update* icon, and accepting the alerts. Both portions of the collab tab are resizable, which allows users to focus on one thing at a time. Altough this interface was enough to perform collaboratively, the team had to get used to working around certain bugs and dealing with certain interactions that might not have been the most user friendly. For example, once a song was selected in the view-only portion a trash icon appeared next to it, which indicated the song would be deleted. The visualization was effectively deleted, but the sound would continue. So we had to be very careful with selecting the correct songs if we wanted to avoid reloading the entire page. Additionally, the updating process could be signigicantly minimized. It currently requires 4 clicks for the update to go through the database, which might be too much effort in a live performance.

6.4 Challenges

As with typical software development, we continually encountered new issues over the course of development. The primary issues we encountered were:

- Monaco Editor integration within React
- Line by line tracking

6.4.1 Monaco Editor Integration

Our very first iteration of Phad was built using a simple HTML text area instead of the Monaco Editor (for more information on the Monaco Editor, see section 5.3.2). This didn't allow for some crucial functionality such as syntax highlighting, autocomplete, and line tracking. Once we decided to use the Monaco Editor, a problem arose with integrating it with React. The Monaco Editor was not built to be used in conjunction with React. This created an array of problems that were incredibly difficult to solve because all of the Monaco Editor's documentation and tutorials aren't written for using it in conjunction with React. Without proper documentation guiding us, properly using all of the Monaco editor's functionality we wanted became a trial and error process. Through this trial and error, we were eventually able to implement all of the features we wanted to.

6.4.2 Line by line tracking

Line by line tracking is crucial for the functionality of Phad. As discussed in section 4, each written line of Phad code is a piece of music (melody, bass line, drum rhythm, etc), and that piece of music can either be turned on or off using Ctrl+Shift and Ctrl+Alt respectively. Because the code is constantly updated and the lines of code are constantly moving around, the program needs to know which line (i.e. piece of music) is running. This requires tracking each line. Direct line tracking isn't directly available through the Monaco Editor. As a result of this, we had to get creative with our approach. The Monaco Editor has the ability to add your own "decorations". These "decorations" are visual components that compliment the written code such as a red underline below a misspelled word. These decorations need to move with the line they're associated with. By creating invisible decorations and constantly retrieving their locations, we were able to track which line was playing and which wasn't. This allows for Phad to have a clear understanding of which line of code is running, even when it changes location.

7 Results

7.1 User Testing

To evaluate our language we went through two rounds of user testing. For both rounds, we collected some preliminary information about each user's experience in both music and programming. The first round of user testing was performed soon after we first deployed the website. We tested then because wanted to receive feedback on our language while we were still adding new features to the language itself. In the first round, we surveyed four people. All four of them had some music background, and two of them had a programming background. This round of testing involved a user working through our tutorial and given exercises with a member of our group present. A full list of these exercises is located in Appendix 10.2.3. These instructions were designed to give our group insight into the effectiveness of our tutorial and how easy our language's syntax is to learn. We received a mix of positive and constructive responses in this process. Users tended to find the language's general syntax easy to work with, with a few exceptions. When asked how intuitive the language was to use on scale of 1 to 5, where 1 is not very intuitive and 5 is very intuitive, two users responded with 3 and two responded with 4. This tells us that the language was not as easy to use for a new user as we would have liked. When looking into the comments given to the group member present we can see which aspects the individual users struggled with. During observations of the exercises, we noticed that two users had some difficulties with how notes are extended. Two users also specifically asked for a wider library of instruments when asked what features they would add. Because of this and some noted confusion when users attempted to try new instruments, increased our library of sampled instruments. This notably added more common instruments such as a piano and a guitar to reduce confusion. We also got feedback on improving our tutorial itself. Three of our four users appeared confused during different points in the tutorial. This was especially seen in our fourth instruction where all of two of the testers commented that it was too long, and in our sixth instruction where two users took longer than we would have liked to understand what was written. In response we took steps to make our tutorial instruction test more concise.

The second round of user testing was not supervised, and happened a month after our first round of testing. This was enough time for us to address the previous round of testing as well as continue to add features to the language and website. Instead we supplied a group of people with the web address of our site and a link to the survey form. We had a total of six responses to our survey. These people were at a meeting for the Creative Coding Club, all of them had programming experience, and five had experience with music. When asked how intuitive the language was to use on a scale of one to five, where one is not very intuitive and five is very intuitive, we received largely positive feedback. Three of the users responded with 5, two with 4, and one with a 3. Additionally, we asked how quickly they were able to learn basic syntax on the same scale. We received four responses of 5, and two responses of 4. This told us that much of the users found our language fairly intuitive to use, but not necessarily as accessible as we would have liked. Because the users tended to feel comfortable with the language's basic syntax fairly quickly, we updated the clarity of our tutorial and documentation pages. This was also due to some of the feedback in the free response section of the survey. While none of the user's explicitly stated that these pages needed to be more clear, we did notice that the users appeared to misunderstand some features that we did include. For instance, one user asked for the ability to play chords, and asked about how our duration increases functioned. This led us to updating both our tutorial and documentation pages to explain each concept in more depth than we previously were.

7.2 Algorave

In order to display Phad, we decided to host our own algorave. We reserved a space on campus and set up an event through the school. During our performance, we used our collaborative editing feature to build a piece together in real time. Our setup consisted of a display laptop which had a database entry open for each performer, a laptop for each performer on which they edited their database entry, and speakers and a projector to display our code. While most of the performance was improvised, we did start with at a pre-written base point that is visible in the first two lines in the bottom right text box in the image below which shows the final outcome of our performance.

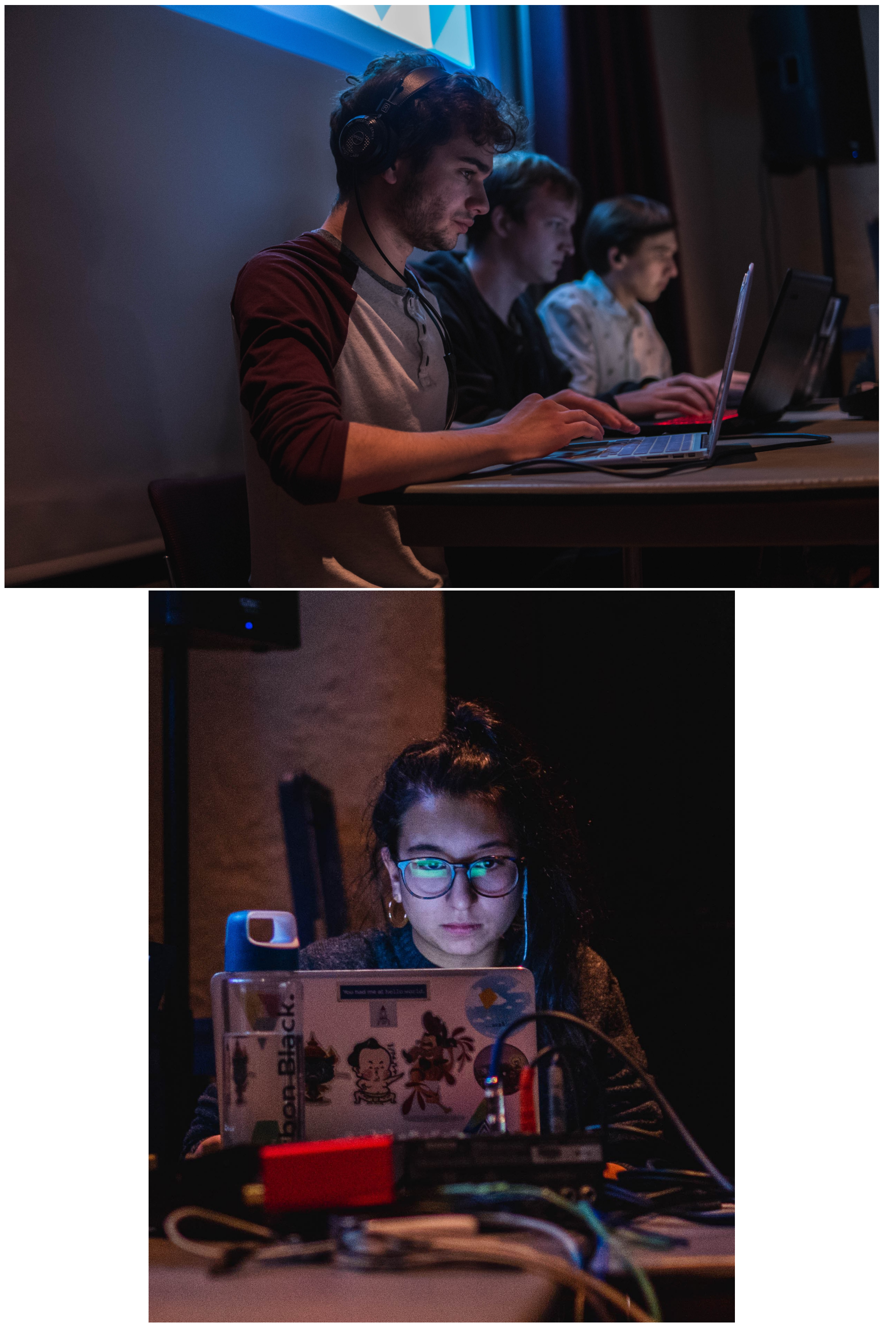
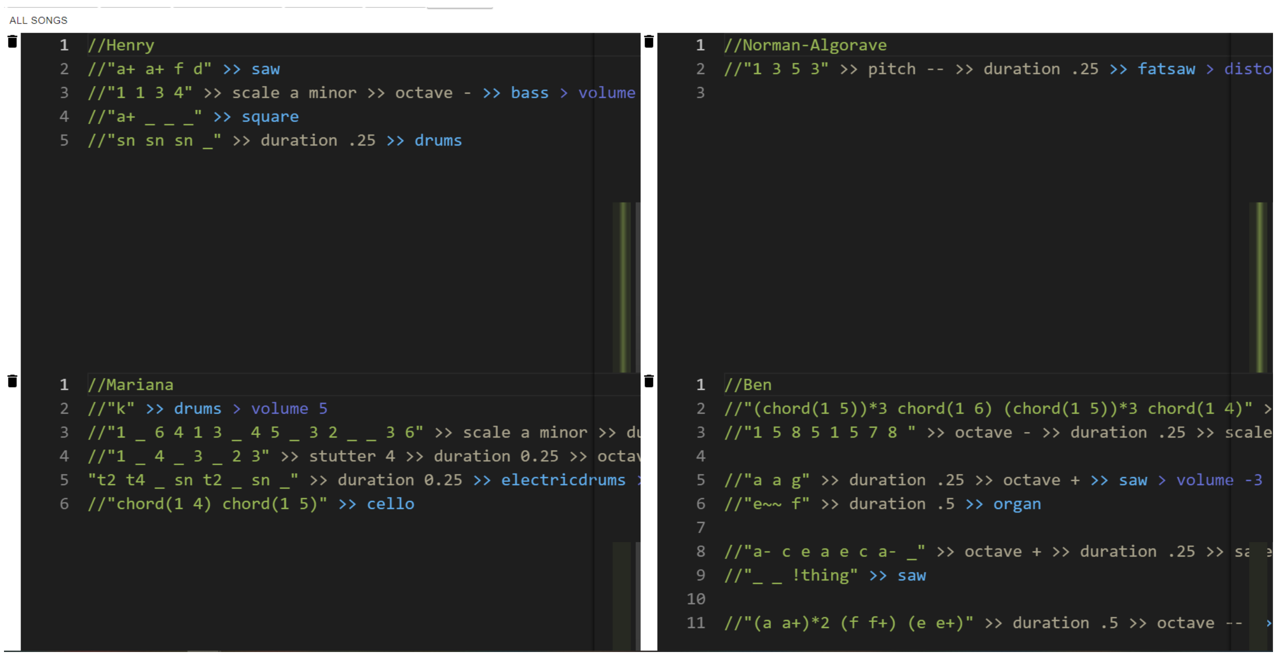


Figure 24: Performing at the Algorave



ALL SONGS

```
//Henry
//"a+ a+ f d" >> saw
//"1 1 3 4" >> scale a minor >> octave - >> bass > volume
//"a+ _ _ _" >> square
//"sn sn sn _" >> duration .25 >> drums
```

```
//Norman-Algorave
//"1 3 5 3" >> pitch -- >> duration .25 >> fatsaw > disto

```

```
//Mariana
//"k" >> drums > volume 5
//"1 _ 6 4 1 3 _ 4 5 _ 3 2 _ _ 3 6" >> scale a minor >> du
//"1 _ 4 _ 3 _ 2 3" >> stutter 4 >> duration 0.25 >> octav
"t2 t4 _ sn t2 _ sn _" >> duration 0.25 >> electricdrums >
//"chord(1 4) chord(1 5)" >> cello
```

```
//Ben
//"(chord(1 5))*3 chord(1 6) (chord(1 5))*3 chord(1 4)" >
//"1 5 8 5 1 5 7 8 " >> octave - >> duration .25 >> scale

//"a a g" >> duration .25 >> octave + >> saw > volume -3
//"e~~ f" >> duration .5 >> organ

//"a- c e a e c a- _" >> octave + >> duration .25 >> sa e
//"_ _ !thing" >> saw

//"(a a+)*2 (f f+) (e e+)" >> duration .5 >> octave --  >
```

Figure 25: Our code at the end of the performance

Overall our performance was a success. Most aspects of the language were used, and different people used the language differently depending on their style. For example, since Phad supports use of both numbers and letters as notes, people were able to use either one depending on their knowledge. Some pre-written lines were written using numbers so that the key of the performance didn't have to be decided until it began. Once the key was set, some performers used numbers to write their melodies while others found it easier to use note names.

The way our language was designed made our performance focused mainly on creating melodic lines. As opposed to other languages which may involve use of more abstract sounds which evolve over time, we focused on repeating melodies which we added to to change the music over time. This made our performance unique to other performers at the algorave.

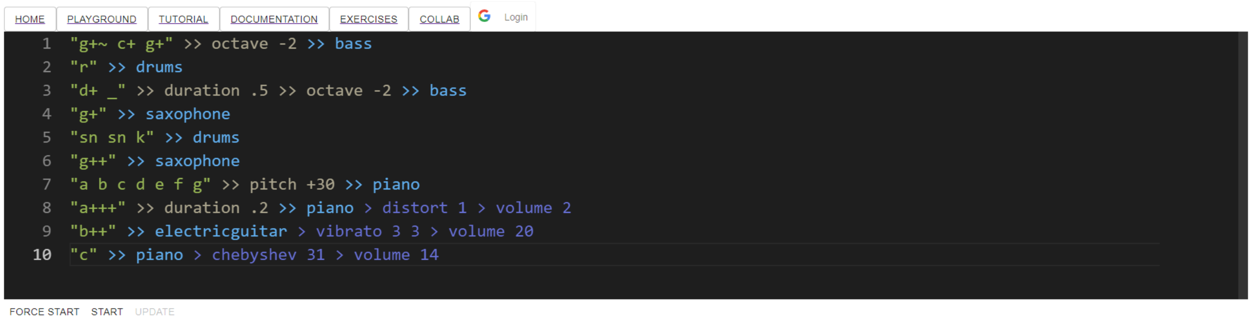
Some features of the language did end up going underused. Although the volume filter was used to adjust levels of different instruments, other filters weren't used much during the performance. This was not due to a problem with the way they were designed; we just didn't think to use them. The save modifier also didn't get used much, mainly because there wasn't much reason to use it. It can be useful when not writing under time constraints, but in the context of the performance there wasn't time to define any complicated sequences that made use of the save modifier.

There was only one problem with the code during the performance. For an unknown reason, one part did not stop playing when it was commented out near the end of the performance. Instead of commenting out parts one at a time until there were none left playing as we had practiced, we slowly turned down the volume on the parts which were still playing when we realized what was happening. Despite this problem, we still believe our performance was successful.

Additionally, our music was accompanied by live-coded visuals. As our language does not natively support visuals, they were done in a language developed by Kit Zellerbach called Barbara. We projected these visuals below the screen where all of our code was displayed.

7.3 Development Results

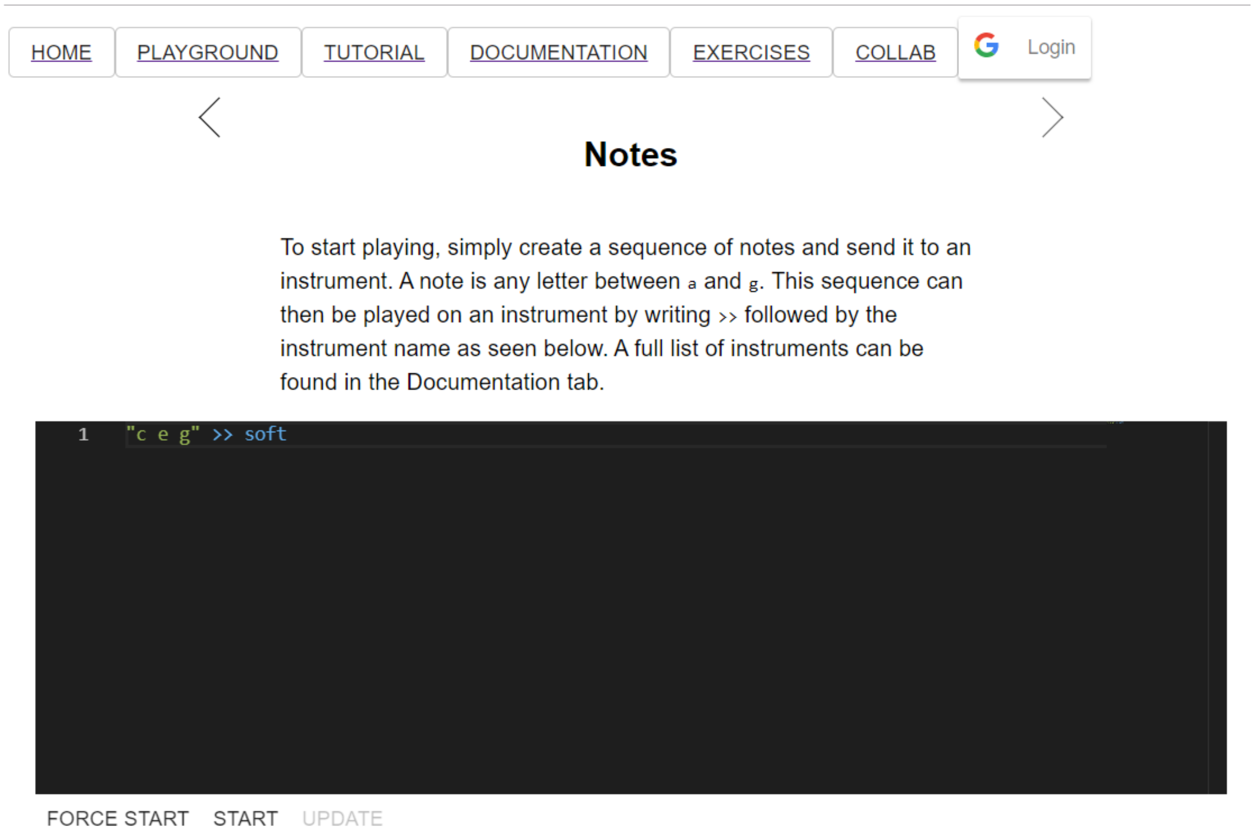
The total development output of our project comprises of a web page [3] that leverages a separately hosted database to store user's code. On the website, we have a landing page, a tutorial, exercises, a playground, and a collaborative playing page. The landing page simply contains a brief introduction to the language, and invites people to try it.



```
"g+~ c+ g+" >> octave -2 >> bass
"r" >> drums
"d+ _" >> duration .5 >> octave -2 >> bass
"g+" >> saxophone
"sn sn k" >> drums
"g++" >> saxophone
"a b c d e f g" >> pitch +30 >> piano
"a+++" >> duration .2 >> piano > distort 1 > volume 2
"b++" >> electricguitar > vibrato 3 3 > volume 20
"c" >> piano > chebyshev 31 > volume 14
```

Figure 26: The Playground page where the user can run their code

From the playground page, the user can write, play, and save their code. This page we decided to keep this page simple, and keep it to an editor and a few select buttons. This was to not overwhelm new users while keeping the essentials to properly use the language.



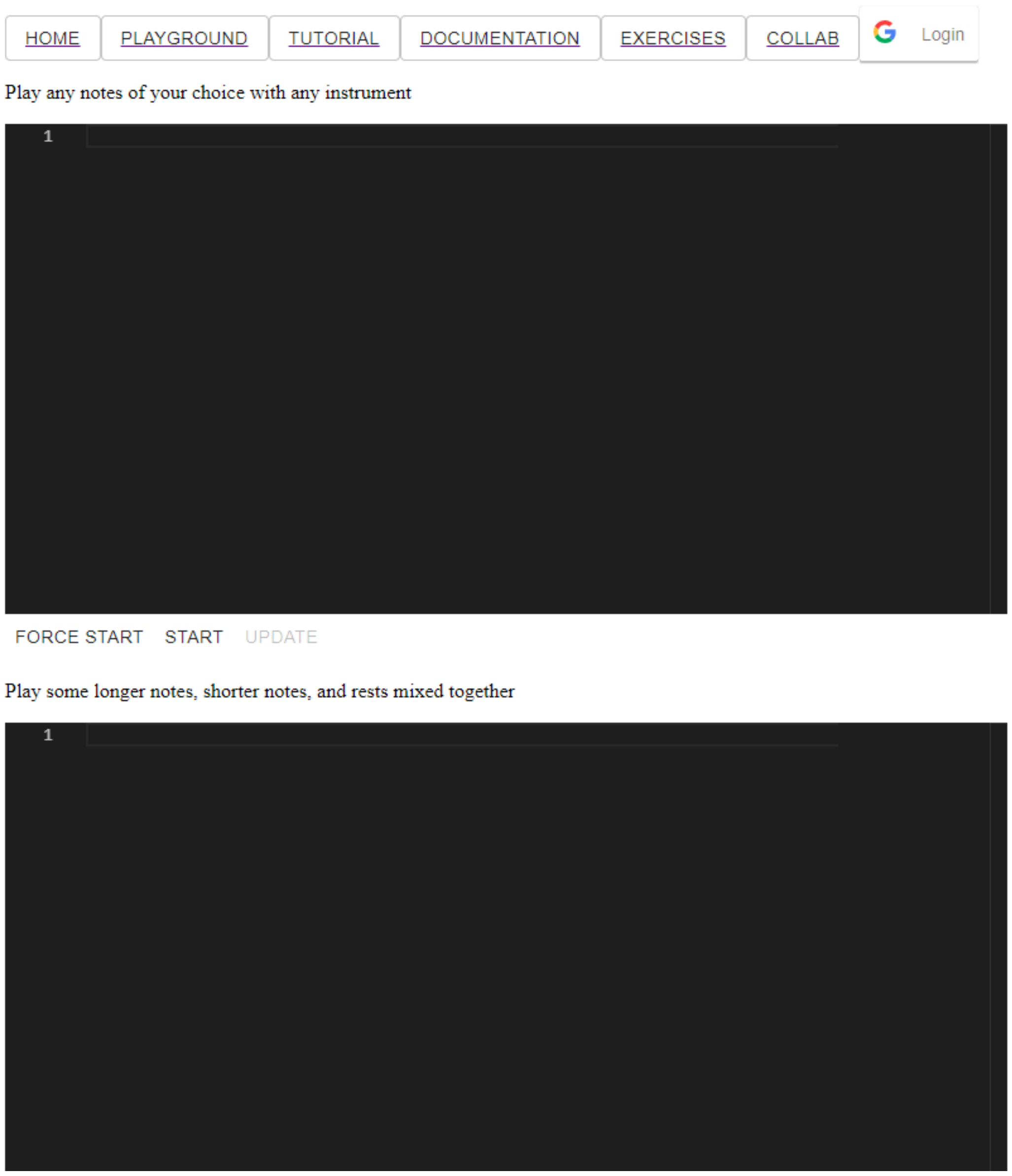
Notes

To start playing, simply create a sequence of notes and send it to an instrument. A note is any letter between `a` and `g`. This sequence can then be played on an instrument by writing `>>` followed by the instrument name as seen below. A full list of instruments can be found in the Documentation tab.

```
"c e g" >> soft
```

Figure 27: The Tutorial where users are introduced to Phad

In our tutorial, users are guided through a set of directions like that above. There is text description of how to use a certain aspect of the language, in this case simple not playing, and the user can play or edit the example. The user can change which instruction they see with the arrows. These instructions go from simple to more advanced as the user progresses through.

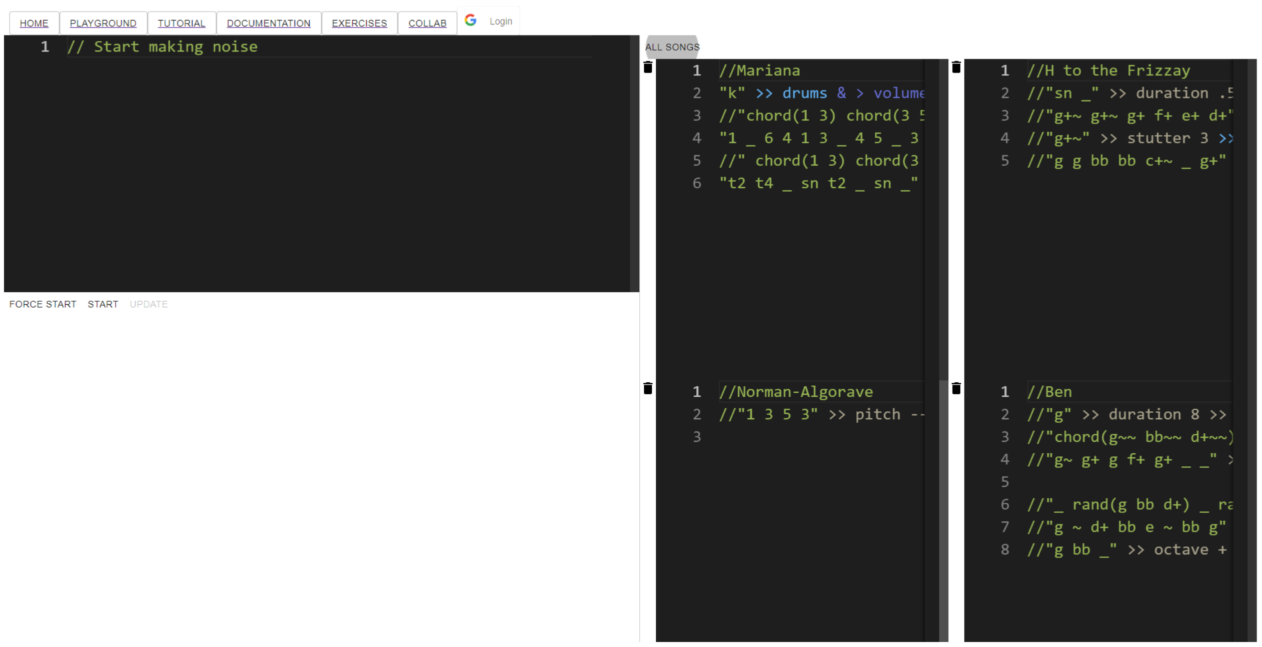


Play any notes of your choice with any instrument

Play some longer notes, shorter notes, and rests mixed together

Figure 28: The Exercise page

On the exercises page, the user is given problems of progressing complexity. Each of these exercises is followed by an editor for the user to write their code with. They are designed to allow newer users the ability to see what they do and do not understand how to do in our language, especially if they do not yet feel comfortable trying to make something new on their own.



```
// Start making noise
```

```
//Mariana
"k" >> drums & > volume
//"chord(1 3) chord(3 5
"1 _ 6 4 1 3 _ 4 5 _ 3
//" chord(1 3) chord(3
"t2 t4 _ sn t2 _ sn _"
```

```
//H to the Frizzay
//"sn _" >> duration .5
//"g+~ g+~ g+ f+ e+ d+"
//"g+~" >> stutter 3 >>
//"g g bb bb c+~ _ g+"
```

```
//Norman-Algorave
//"1 3 5 3" >> pitch --

```

```
//Ben
//"g" >> duration 8 >>
//"chord(g~~ bb~~ d+~~)
//"g~ g+ g f+ g+ _ _" >

//"_ rand(g bb d+) _ ra
//"g ~ d+ bb e ~ bb g"
//"g bb _" >> octave +
```

Figure 29: The Collaborative Editing page that will play from each editor

The collaborative editing page allows the user to pick code saved to the database, and it will continue pulling each file from the database and updating it for the user every 15 seconds. When being loaded into the box the songs are played at the same time, and they will be played alongside any code the user writes in the leftmost editor.

8 Future Work

8.1 Language

Overall, we believe Phad has improved constantly, but requires a bit more extensiveness, primarily in its Instruments and Filters, as it presents some limitations in its features. In the future, we hope more Filters will be added to allow for all kinds of sound generation, reducing the repetitiveness of Phad songs. The problem of repetitiveness can also be addressed by expanding on the ability to progressively change certain behaviors, as most live coding languages do. For instance, this could mean having the ability to change the behavior of Filters over time, and generating different sounds with the same Sequence base, without the need to modify the Filters manually.

8.2 Storage

Although storing information in the database was a good starting point, we believe that as Phad gets larger and is used more extensively, fetching from the database will become very slow and unbearable in the context of live performance. It would be ideal to create an Atom or VS Code plugin, and integrate Phad with a local sound generating software such as SuperCollider or Ableton Live so that users can save their songs locally on their machines. The overall performance of our application seemed to be reduced significantly by the database, causing significant issues in sound generation through Tone.js. This is an issue that would need to be addressed, especially for live coding performances in Phad, for which the database is a key component. Additionally, it became somewhat annoying to reload the page and lose progress in a song if this one was not stored in the database. A potential solution could be to implement local storage for all of the PlayBoxes, allowing for a better workflow in our environment.

8.3 User Interface

The Phad user interface could be improved significantly. The past semester was focused around functionality and user experience concurrently. After the user testing that was done, many design changes were adopted, but user testing was usually not done after the updates. The Phad user interface could be developed further with the help of external perspectives. Most of our project was focused on development and what we thought made sense in the design, but testing was not emphasized. During the last few weeks we were the main users of the interface, and this helped us notice numerous bugs, unexpected behaviors, and annoying requirements. Because the collaborative tab was the last one to be implemented, it contains a lot of bugs that haven't been tested. As mentioned earlier in the paper, during live performance deleting the song does not stop the sound of the deleted song. This could be fixed by calling the stop function before resetting the state of the React component. Another bug was encountered during the Algorave, where even though sounds were commented out and handled properly by the parser, they were still being played. We believe that this had to do with the way changes were handled in the setInterval function, which does not guarantee that all running parts are set properly in the list, causing the stop call to never happen because the program sees there is nothing in the array, and therefore nothing to call stop on.

After the Algorave, some of the audience mentioned that the view of four screens at the same time was maybe not the best way to display the four songs, especially because the lines were cut on right by the editor limit. Some solutions to this could be to change the first step of Part parsing from splitting line by line to split by every other “, as this indicates another Sequence is defined and so by definition a new Part begins. This would allow users to write Parts in multiple lines rather than having long lines that are cut by the editor limit. Another solution to the four screen problem could be to figure out a way to edit the same box at the same time, through a web socket or a similar technology, which would allow us to display a single editor. We could also revert to an initial solution we had for collaborative editing, which was the idea of allowing multiple users to edit and create the same song on different machines. This functionality would be similar to how Google Docs works with multiple users editing the same document simultaneously. This would allow for multiple coders to perform at the same time on the same screen during a live coding performance. To do this, we attempted to use a network-agnostic shared editing platform called Yjs[9]. In theory this would allow us to configure individual Monaco Boxes to support peer to peer editing. Unfortunately we were not able to get this configured and implemented quickly, and we felt that our time would be better spent finishing other aspects of the project. That being said, we did add a collaborative editing page where multiple songs are played at the same time to achieve a similar effect in less development time.

An early idea was to have a static box in the top right corner that displays all current running parts. These would be expressed by displaying which notes are playing and which instrument is playing them. Additionally, clicking on the part would bring your cursor to the location within the Monaco Editor where the part was written.

9 Conclusion

The final iteration of Phad presented a promising new web application for live coding performance. Phad makes use of existing tools to focus on user experience, and create an easily accessible live-coding music environment. The final weeks of our project were focused on bettering the collaborative interface and sharpening our knowledge of the language for the purpose of a live performance. During this time we were able to encounter the more interesting and creative parts of the system we built, as well as many aspects that will require more work. Although a lot can be improved in the Phad system, part of the WPI community showed significant interest in the platform and its future. We hope that Phad can contribute to the increasing popularity of creative coding at WPI, and that students can explore their passions for art and technology through these kinds of projects.

10 Appendices

10.1 Appendix 1: Documentation

10.1.1 Notes

A note is any letter between 'a' and 'g', uppercase or lowercase, corresponding to the musical notes. A note has multiple optional specifiers and modifiers that determine the pitch and octave of the note. The order in which these are applied matters, and is the following:

- **Accidentals**: the '#' or 'b' characters can be added to the right of a note to **sharp** or **flat** it. For example, 'b#' would be a B sharp and 'bb' would be a B flat.
- **Octave number**: any positive integer can be added to the right of a note (or to the right of its accidental if it has one) to specify the octave. For example 'b5' is a 'b' note in the 5th octave. If no octave number is specified, the default value is 4.
- **Octave modifier**: any number of '-' or '+' can be added to the right of a note and its accidental to increase or decrease the octave of the note. For example 'b+++' is a 'b' note in the 7th octave. 'b+++'is equivalent to writing 'b+3' or 'b7'.

10.1.2 Numbers

A number can be any integer, and will correspond to a note in a scale. The scale is specified with the 'scale' sequence modifier, described in more detail in the modifiers section. The default scale is C major.

```
"1 3 5"
    >> scale c major
```

The above example is equivalent to "c e f".

10.1.3 Percussion

Percussion symbols correspond to the following predetermined sounds. Percussion can only be played with the 'drums' instrument.

- 'sn': snare
- 'k': kick
- 'h': hi-hat

- 'oh': open hi-hat
- 'r': ride cymbal
- 'be': ride bell
- 't1': tom 1
- 't2': tom 2
- 't3': tom 3
- 't4': tom 4

10.1.4 Extensions

Notes, numbers, and percussion can be followed by any number of '˜'s. This will cause it to be extended by the number of '˜'s used.

```
"a ~ ~ b ~ c"
```

In the example above, the 'a' will be three times as long as the 'c' and the 'b' will be twice as long as the 'c'.

10.1.5 Group behaviors

- **sequential**: plays groups in order when written as '(a b c)'
- **chord**: plays groups at the same time when written as 'chord(e g)'
- **random**: plays one group per loop, selected at random, when written as 'rand(c e g)'.
- **repeat**: repeat a group a number of times when written as '(c)*3' or '(c e)*3'. The latter is interpreted the same as '(c e c e c e)'.

10.1.6 Sequence modifiers

A sequence modifier is added after the phrase is defined as:

```
"phrase"
  >> sequence modifier
```

Sequence modifiers are executed in order, and each sequence modifier acts without knowledge of future sequence modifiers. For example, consider the following code.

```
"a b c"
  >> octave 3
  >> octave ++
```

When "a b c" >>octave 3 has been executed, the result is equivalent to '"a3 b3 c3"'. This result is then passed into the '>>octave ++' modifier, and the resulting output is equivalent to '"a5 b5 c5"'.

10.1.7 Octave

The 'octave' modifier accepts 3 types of input:

- A sign and a number: '+3' or '-3'
- A number of signs: '+++' or '—'

The number input will set the octave corresponding to that number. The input with signs will increase or decrease the octave relative to a previous 'octave' modifier if a number was specified, or from the default value if nothing was specified before. For example, below the entire sequence will be changed to octave 3, since the default octave is the 4th.

```
"a b c"
  >> octave -
```

10.1.8 Pitch

The 'pitch' modifier changes a sequence by a number of half-steps. It accepts 2 types of input:

- A sign and a number: '-3'
- A number of signs: '—'

When modifying numbers, the pitch modifier is equivalent to changing the numbers by that amount (so "1" >>pitch +' is equivalent to "2").

10.1.9 Duration

The 'duration' modifier changes the duration of each individual group in the phrase. It accepts a fraction or decimal as input, and the duration of each group in a sequence will be multiplied by this value.

10.1.10 Stutter

The 'stutter' modifier repeats each of the highest level groups a given number of times.

```
"a b c d"
  >> stutter 2
  >> triangle
```

For example, the above example would play "a a b b c c d d", but the below example would play "(a b) (a b) (c d) (c d)" because the parentheses are the highest level group.

```
"(a b) (c d)"
  >> stutter 2
  >> triangle
```

10.1.11 Scale

The 'scale' modifier only changes numbers, and changes the scale against which the number will be evaluated. It accepts a key (eg. 'C', 'f#', 'Bb') and a scale type (see below), in that order. Both parameters are optional, and any parameters not used remain the same.

```
"1 3 5"
  >> scale c major >> piano
  >> scale d >> flute
  >> scale minor >> cello
```

In the above example, '"c e g"' would be played on the piano, "d f# a" (D major) would be played on the flute, and "d f a" (D minor) would be played on the cello. The scale types are:

- 'major': Can be abbreviated to 'maj' or 'M'
- 'naturalminor': Can be abbreviated to 'minor', 'min', 'nm', or 'm'
- 'harmonicminor': Can be abbreviated to 'hm'
- 'chromatic': Can be abbreviated to 'ch'
- 'majortriad': Can be abbreviated to 'Mtriad', 'Mt', or 'M3'
- 'minortriad': Can be abbreviated to 'mtriad', 'mt', or 'm3'

10.1.12 Copy

The 'copy' modifier can be used to make use of multiple instances of the same sequence at once. It takes a type of group ('seq', 'chord', or 'rand'), and a series of modifier sequences.

```
"c e g"
  >> copy seq
  (>> pitch +,
  >> octave +)
  >> triangle
```

The above example will first play '"c e g" >>pitch+' then play "c e g" >>octave +". To play both of these at once, the word 'seq' could be replaced with 'chord', and to randomly select one of them it could be replaced with 'rand'.

```
"1 2 3"
  >> copy chord
  ( , >> pitch ++)
  >> triangle
```

The above example plays "1 2 3" unmodified by having only a space between the open paren and the first comma. It is equivalent to "chord((1 2 3) (3 4 5))".

10.1.13 Sequence storage

The 'save' command allows you to save any sequence as it is when the command is reached. The 'save' command will only save **sequences**, which does not include any parts or part modifiers (i.e. instruments and its modifiers).

```
"a b c"
  >> octave 4
  >> save abc_1
  >> pitch ---
  >> save abc_2
```

In the example above, 'abc_1' would contain the sequence and its 'octave' modifier. Then, 'abc_2' would contain all of that, plus the 'pitch' modifier. Saved sequences can be used in place of notes by using '!name':

```
"!abc_1"
  >> <any sequence modifier or part definition>
```

Saved sequences can be combined into groups in the same way as notes.

```
"chord(!abc_1 !abc_2)"
  >> <any sequence modifier or part definition>
```

10.1.14 Parts

A part is defined once the **instrument** of a phrase has been defined in the case of notes and numbers:

```
"a b c"
  >> octave +
  >> saw      // this is the beginning of a part
```

Instruments can be divided into two categories, synths and samplers. The synths are as follows:

- 'triangle': Simple synth with a triangle wave
- 'soft': Sine wave synth with a rich tone quality
- 'fatsaw': Fat synth with a saw wave, distorted sound
- 'saw': Saw wave with very bright tone
- 'square': Fat square wave similar to fatsaw but more forcecful
- 'pls no': Pulse oscillator with harsh tone quality
- 'alien': Smooth but strange sound

The samplers are made from instrument samples and include:

- 'drums' or 'acousticdrums': Used to play drums
- 'electricdrums': Used to play drums
- 'piano'
- 'electricbass' or 'bass'
- 'bassoon'
- 'cello'
- 'clarinet'
- 'contrabass'

- 'flute'
- 'frenchhorn' or 'horn'
- 'acousticguitar'
- 'electricguitar' or 'guitar'
- 'nylonguitar'
- 'harmonium'
- 'harp'
- 'organ'
- 'saxophone'
- 'trumpet'
- 'violin'
- 'xylophone'

10.1.15 Synth Attributes

Synths may be followed by one or more attributes which will change the sound of the instrument:

```
"a b c"
  >> triangle volume 10
```

Attributes can only be applied once each. Specifying the same attribute multiple times will override previous changes to that attribute. Attributes can be any of:

- 'volume': Takes a fraction or decimal, positive or negative, and changes the volume of the instrument by that number of decibals
- 'wave': Takes one of 'sine', 'triangle', 'sawtooth', or 'square', and sets the wave of the instrument to that type.

10.1.16 Filters

Filters change the way an instrument or percussion element sounds. To add a filter:

```
"a b c"
  >> piano
    > <filter>
    > <filter>
  >> <any sequence modifier>
```

Filters act similarly to sequence modifiers in that they are processed sequentially, and every filter acts without knowledge of filters which follow it. The '&' symbol may also be inserted between filters. This will cause the sound to be played at the point at which the '&' symbol is inserted. The sound will always be played at the end of a sequence of filters, regardless of whether a '&' is used.

```
"a b c"
  >> piano
    > pingpong 0.5
```

As an example, the example above will only play the pingpong echo, not the original note. However, the example below will play both the unmodified notes as well as the echo.

```
"a b c"
  >> piano &
    > pingpong 0.5
```

Filters can be any of:

- 'pingpong': Makes the sound echo between speakers. Takes a delay between echos.
- 'volume': Changes volume. Takes a number in decibals where negative values make it quieter and positive values make it louder.
- 'distort': Adds distortion to the sound. Takes a positive number, recommended values between 0 and 1.
- 'pan': Moves sounds from left to right. Takes a number from -1 to 1 as input. -1 corresponds to left, and 1 corresponds to right.
- 'chebyshev': Applies a Chebyshev distortion. Takes a positive integer, recommended values between 1 and 100. Note that odd numbers are very different from even numbers.

- 'tremolo': Quickly pans between left and right ears. Takes two numbers, a frequency in Hz and a maximum pan value (0.0 - 1.0)
- 'vibrato': Quickly shifts the pitch up and down. Takes two numbers, a frequency in Hz and an amount the pitch is modulated (0.0 - 1.0)
- 'lo': Low pass filter attenuates frequencies above the cutoff. This takes an integer frequency.
- 'hi': High pass filter attenuates frequencies below the cutoff. This takes an integer frequency.
- 'limit': Volume limiter. Takes a value in decibals.

10.2 Appendix 2: Round One User Testing

The following are the questions and instructions that we gave to the tested users during the first round of user testing. These questions were hosted in a Google Survey, and the users ran their written code and used the tutorial as located on our website.

10.2.1 Preliminary questions

These are the initial questions that we asked users in order to understand their backgrounds.

- Do you have experience making music? If so, please explain.
- Do you have experience programming? If so, please explain.

10.2.2 Tutorial

The following is the tutorial section prompt in our form. The user was sent to the tutorial on our website, and the group member supervising took notes on how well the user was understanding the material as well as any questions that they had along the way.

- Using this link: `mqp-live-coding-language-design.github.io/mqp/#/tutorial`, please take the time to follow this tutorial. During this time, we encourage you to speak out loud whenever you apply any modifications in the text boxes. You are also encouraged to ask any clarification questions, as this will help us improve our project. We will be observing you and taking notes as you go through this tutorial.

10.2.3 Instructions

This section contains the prompts given to the user after completing the tutorial, where we wanted to evaluate how much of our language they were able to retain. During this we recorded how quickly the user was able to write code to fulfill the expectations, and we recorded any questions or difficulties that they had. The prompt and instructions are as follows:

During this section we want you to do your best to create something from specific instructions we will give you. You are encouraged to (once again) talk out loud and ask any questions you might have. You can also refer to the language tutorial or documentation.

- Play three notes of your choice in the 6th octave with the ‘saw‘ instrument
- Play any two different sets of notes at the same time with two different instruments
- Play any sets of alternating notes and rests
- Make a sick drum beat
- Play the following 4 notes in the 4th octave: A flat, F, D flat, E flat with the ‘soft‘ instrument. Make the notes play faster.
- Using your code from the previous instruction, modify it so that:
- The same notes are played with the same modifiers and instrument
- The same notes but 7 pitches up and a duration are played with the ‘triangle‘ instrument at the same time Play three notes of your choice in the 6th octave with the ‘saw‘ instrument
- Play any two different sets of notes at the same time with two different instruments
- Play any sets of alternating notes and rests
- Make a sick drum beat
- Play the following 4 notes in the 4th octave: A flat, F, D flat, E flat with the ‘soft‘ instrument. Make the notes play faster.
- Using your code from the previous instruction, modify it so that:
  - The same notes are played with the same modifiers and instrument
  - The same notes but 7 pitches up and a duration are played with the ‘triangle‘ instrument at the same time

10.2.4 Get creative

This portion of the testing involved directing the user to the ”playground” portion of our website to write whatever code they wished in the editor. We wanted to see how well users were able to apply what they learned about our language to writing their own music. During this we took notes on any questions, difficulties, or suggestions that the user had. The prompt for this section is as follows:

During this section you can create some sound with anything that you learned today. We provide some examples for inspiration, but feel free to create anything you want.

10.2.5 Evaluation

After the user had some time to use our language, we requested their feedback. This was to receive any evaluations of the language from the user that we may have missed in an earlier section, and field any suggestions for improvement. The prompt can be seen below:

This section hopes to evaluate our language’s features and usability. You are also free to suggest new feature ideas or anything you would like to see this project accomplish.

10.3 Appendix 3: Round Two User Testing

The following are the questions and instructions that we gave to the tested users during the second round of user testing. This round was similar to the first, except that our tutorial and instruction sections are hosted on the website. This allowed this survey to be focused on specific feedback on the language without needing to supervise each user. The tutorial can be found at `mqp-live-coding-language-design.github.io/mqp/#/tutorial` and the exercises for the user can be found at `mqp-live-coding-language-design.github.io/mqp/#/exercises`. These questions were hosted in a Google Survey, and the users ran their written code and used the tutorial as located on our website.

10.3.1 Preliminary questions

These are the initial questions that we asked users in order to understand their backgrounds. Because of the audience for this round of testing and because we wanted more background information to help in evaluating our language the questions are more specific than in the first round.

- Are you a computer science major?
- Please explain your experience with programming

- Please explain your experience with making music

10.3.2 Language Feedback

After the user had some time to use our language, we requested their feedback. We asked more questions in this section than we did in the first round, as we did not hold supervised test sessions. We also received feedback on our website in this session.

We first asked the users to score how easy it was to use our language and website. These questions and their scales are seen below.

- How quickly were you able to understand the basics of the language's syntax? The scale was from 1 to 5, 1 being "It took a while" and 5 being "It was quick"
- How intuitive was this language to use? The scale was from 1 to 5, 1 being "Not intuitive at all" and 5 being "Very intuitive"
- How difficult was it to navigate the website? The scale was from 1 to 5, 1 being "Not difficult" and 5 being "Very difficult"

The next questions were free response and focused on getting specific examples of elements that the user did or did not enjoy using. We also took suggestions for future changes in this section. The questions are as follows:

- Did anything confuse you while you were using the language?
- What did you like about the language?
- What did you dislike about the language?
- Are there any features you would like to see added to the language?
- Are there any UI design choices you would have made differently?
- Were you able to make anything with our language that you would like to share with us?

References

[1] Livecodelab. `https://livecodelab.net/`. Last visited on 12/12/2019.

[2] Algorave. Algorave. `https://algorave.com/`, 2012. Last visited on 12/10/2019.

[3] Benjamin Anderson, Norman Delorey, Henry Frishman, and Mariana Pachon-Puentes. Phad. `https://mqp-live-coding-language-design.github.io/mqp/#/`, 2019. Last visited on 12/12/2019.

[4] Davide Della Casa and Guy John. Livecodelab 2.0 and its language livecodelang. In *FARM@ ICFP*, pages 1–8, 2014.

[5] Facebook. React.js. `https://reactjs.org/`. Last visited on 12/10/2019.

[6] Inc. GitHub. Github. `https://github.com/`. Last visited on 12/10/2019.

[7] Inc. GitHub. Github pages. `https://pages.github.com/`. Last visited on 12/10/2019.

[8] Olivia Jack. Hydra. `https://github.com/ojack/hydra`, 2017. Last visited on 11/04/2019.

[9] Kevin Jahns. yjs/yjs. `https://github.com/yjs/yjs`. Last visited on 12/10/2019.

[10] Ryan Kirkbride. Foxdot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, pages 194–198, 2016.

[11] Thor Magnusson. ixi lang. `http://www.ixi-audio.net/`. Last visited on 12/12/2019.

[12] Thor Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, pages 503–506. Michigan Publishing, 2011.

[13] David Majda. Peg. js: Parser generator for javascript. *URL: http://pegjs. majda. cz*, 2011. Last visited on 12/12/2019.

[14] Yotam Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

[15] Material-UI. Material-ui. `https://material-ui.com/`. Last visited on 12/10/2019.

[16] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[17] Microsoft. The monaco editor. `https://microsoft.github.io/monaco-editor/`. Last visited on 12/10/2019.

[18] University of California Berkeley. Postgresql. `https://www.postgresql.org/about/`, 1986. Last visited on 12/10/2019.

[19] Hundred Rabbits. Orca. `https://github.com/hundredrabbits/Orca`, 2018. Last visited on 11/17/2019.

[20] Charles Roberts and Mariana Pachon-Puentes. Bringing the tidalcycles mini-notation to the browser. In *Proceedings of the Web Audio Conference*, 2019.

[21] Charlie Roberts and JoAnn Kuchera-Morin. Gibber: Live coding audio in the browser. In *Proceedings of the International Computer Music Conference*, 2012.

[22] Salesforce. Heroku. `https://www.heroku.com/what`. Last visited on 12/13/2019.

[23] StrongLoop. Express. `https://expressjs.com/`. Last visited on 12/13/2019.

[24] Tony Wallace. serialist. `https://github.com/irritant/serialist`, 2016. Last visited on 12/12/2019.

[25] Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. Live algorithm programming and a temporary organisation for its promotion. In *Proceedings of the README Software Art Conference*, volume 289, page 290, 2004.