MiniClef: Sheet Music Generation for Live Coding in Python

Albert Qi
Harvard College
Cambridge, Massachusetts, USA
albertqi@college.harvard.edu

ABSTRACT

Live coding is a dynamic form of artistic expression that allows performers to create original sounds and music in real-time through programming. While it is primarily used for performance purposes, live coding also holds potential as a valuable tool in the realm of education. For example, if I want to explain polyrhythms to students, then I want to be able to generate many different rhythms whenever necessary. Yet, despite its popularity, live coding tools are typically limited to audio generation and do not offer support for creating sheet music. Sheet music is a crucial element in music education, as it helps students learn music theory through visual representation.

To address this gap, I introduce MiniClef, a domain-specific language for live coding embedded within Python that generates accompanying sheet music based on the underlying code. I begin by presenting MiniClef through a series of examples, highlighting its key features and how it can be used to create music. Next, I explain how MiniClef generates sheet music and provide examples that demonstrate its educational potential. Finally, I discuss possible future developments that could further enhance MiniClef's functionality and expand its usefulness in music education.

All source code can be accessed at https://github.com/albertqi/miniclef.

1 INTRODUCTION

Live coding is a performing arts tool where live music is played based on specific code that is written. It is typically improvised and performed in an interactive manner, and the use of programming allows for the production of unique sounds through techniques such as algorithmic composition. Images, light systems, or other accompanying digital media are often displayed alongside the source code as well.

While live coding is mainly used for performance purposes, it can also be a helpful tool for teaching music. For instance, when explaining tempo or polyrhythms, I might want to be able to generate many different rhythms fairly quickly. Live coding would allow me to generate music in real-time and show students how different rhythms can be combined to create complex patterns. Additionally, live coding could be used to explain music theory concepts such as melody or harmony. For example, I might want to show how a melody can be constructed from a chord progression. Through live coding, I could write code that plays a melody based on a given chord progression for students to hear.

However, can students really learn music theory via just audio and source code alone? What if they, for example, wanted to visualize the chord progressions that are being played as well? These are some important questions to ask, and answering them may help improve the students' learning process immensely.

To address these issues, I propose MiniClef, a domain-specific language for live coding embedded in Python that generates sheet music for the corresponding source code. I first introduce MiniClef by example, describing its features and how it can be utilized to generate music. I then outline how MiniClef generates sheet music and showcase some examples that demonstrate its value in an educational setting. Finally, I cover some future work that could amplify MiniClef's usage and practicality even further.

Overall, I make the following contributions within this paper:

(1) I highlight the need to support sheet music generation in live coding.
(2) I introduce MiniClef, a domain-specific language in Python built for sheet music generation in live coding.

(3) I outline various examples of the MiniClef language and its possible use cases.

2 BACKGROUND

I begin by describing some of the existing software for live coding and the unique properties of each.

2.1 SuperCollider

SuperCollider [4] is a programming language that is used specifically for audio synthesis and algorithmic composition. It is based in C++ and is known for its ability to create complex sounds and music. SuperCollider is often used for live coding due to its ability to generate new sounds in real-time, and many artists and musicians utilize it for that reason. However, SuperCollider can be difficult to learn and use as it is fairly complex, and it may not be the best tool for beginners. As a result, many other live coding languages have been developed that are easier to use and simply use SuperCollider for its audio synthesis.

SuperCollider provides a specific server used just for audio synthesis called `scsynth`. It is mostly controlled via the SuperCollider language but can also be utilized by itself. Through Open Sound Control (OSC) messages, MiniClef is able to send messages to the server to generate its sounds.

2.2 FoxDot

FoxDot [2] is a live coding environment designed to make algorithmic music creation more quick and accessible. It is built on top of SuperCollider and uses Python to define `Player` objects that generate music. Note that it requires SuperCollider to be installed since FoxDot relies on SuperCollider for audio synthesis.

Since Python itself has a relatively simple syntax, FoxDot also has a simple syntax that is easy to learn and use. This makes it a great tool for beginners who may not be familiar with programming. For this reason, MiniClef is also built on Python.

2.3 TidalCycles

TidalCycles [3] is a domain-specific language embedded in Haskell that is designed to make rhythmic and harmonic patterns easy to define. It makes use of cycles in order to generate complex rhythms that other live coding environments cannot replicate. Like FoxDot, TidalCycles also uses the SuperCollider server for sound synthesis.

2.4 Sonic Pi

Sonic Pi [1] is a popular live coding environment that is specifically designed for educational purposes. It is a Ruby-based programming environment that follows a computer-science-based approach to music, with a focus on loops, conditionals, and functions. This makes the language a great tool to teach programming concepts to those who may be familiar with music already.

Moreover, Sonic Pi uses its own prebuilt `scsynth` server in order to generate sounds. This abstracts away much of the complexity of SuperCollider's own language and simplifies the installation process. MiniClef follows this approach as well.

3 MINICLEF

I begin by introducing MiniClef by example, showcasing and explaining many of its interesting features.

3.1 Setup

I want MiniClef to be easy-to-install, so the only requirements are Python 3.11 and LilyPond; the latter is necessary for sheet music generation. Unlike FoxDot and TidalCycles, the MiniClef language does not require a specific installation of SuperCollider since a prebuilt version of `scsynth` is utilized instead. For my testing purposes, I perform all of my live coding in a Jupyter notebook with an initial code block containing a simple import.

```
from miniclef import *
```

3.2 Patterns

Patterns are the fundamental building blocks for MiniClef. Each pattern has a name and a list of beats to be played in order. In order to create and play a pattern, I can use the `loop` method. This method takes in a pattern name and a pattern string, whereupon the pattern string is parsed into a `Pattern` object and the corresponding music will play on repeat.

```
loop("p1", "arpy")
```

The above code creates a new pattern with the name `p1` and a list of beats containing just one `arpy` note.

3.3 Synths

The `arpy` note is an example of one of eleven synths currently supported by MiniClef. The full list of synths is as follows:

```
["angel", "arpy", "bass", "bell",
 "pads", "pluck", "ripple", "saw",
 "sinepad", "sitar", "swell"]
```

Each synth is a unique sound that is generated by SuperCollider; each synth can be played by simply referencing its name. Rests can be played via the ~ symbol.

```
loop("p1", "arpy bass pluck")
```

The above code creates a new pattern with the name `p1` and a list of beats containing one `arpy` note, one `bass` note, and one `pluck` note; these three notes will be played in order on repeat. Note that any existing patterns with the name `p1` will be overwritten.

I can also play multiple patterns at the same time, so long as they have different pattern names.

```
loop("p1", "arpy bass pluck")
loop("p2", "sitar")
```

In this case, the `sitar` note will play three times every time the three `arpy`, `bass`, and `pluck` notes play once.

3.4 Sequences

Currently, the music may seem somewhat boring since each beat just plays one note. So, how can I create some more interesting rhythms?

I can use sequences to play multiple notes in one beat, all evenly spaced out. Square brackets denote the start and end of a sequence.

```
loop("p1", "arpy [bass ~ pluck]")
```

The above code first plays one `arpy` note for a normal duration (i.e., one full beat). Then, for the same duration as the `arpy` note, three notes are played afterward in rapid succession. Each of the `bass`, ~, and `pluck` notes takes up 1/3 of the duration to form one full beat.

Note that I can nest sequences inside of each other as well.

```
loop("p1", "arpy [[bass bass bass] pluck]")
```

For this example, let us define `arpy` to have a duration of 1. Then, we know that both `[bass bass bass]` and `pluck` will have a duration of 1/2. Since `[bass bass bass]` has a duration of 1/2, this means that each `bass` note has a duration of 1/6. Thus, we have `arpy` with a duration of 1, `bass` with a duration of 1/6, and `pluck` with a duration of 1/2.

3.5 Chords

I have already showcased that multiple patterns can be played at the same time to create overlapping sounds. Yet, I can also play sounds in parallel (i.e., chords) via a single pattern. Parentheses denote the start and end of a chord.

```
loop("p1", "(arpy bass pluck)")
```

In this example, the `arpy`, `bass`, and `pluck` notes will all play at the same time to form a chord. Each note still has a full duration of 1.

3.6 Cycles

There are two ways to play a pattern such that is not identical each time: cycles and randomness. I will begin by describing cycles. Note that, here, the term "cycles" is different from the notion of "cycles" from TidalCycles.

I can use a cycle to play a different note each time a pattern is played; that is, I can "cycle" through a list of notes. Angle brackets denote the start and end of a cycle.

```
loop("p1", "arpy <bass pluck>")
```

In this example, the notes `arpy` and `bass` will play once in the first iteration. Then, the notes `arpy` and `pluck` will play once in the second iteration. Afterward, the notes `arpy` and `bass` will play once in the third iteration. This cycle will continue indefinitely, where `arpy` plays once and `bass` and `pluck` alternate every iteration.

3.7 Randomness

The second way to play a pattern with some variation is via randomness. I can create a group of notes and then randomly select one note to play each time the pattern is played. Curly braces denote the start and end of a random group.

```
loop("p1", "{arpy bass pluck}")
```

In the code above, either the `arpy`, `bass`, or `pluck` note will play each time the pattern is played. The note that is played is chosen randomly, each with equal probability (in this case, 1/3 probability of choosing any note).

3.8 Pitch

I have shown how to change instruments and rhythm, so now, I also want to look at pitch. I can specify the pitch of a note by adding a semicolon and following it with up to three characters like so:

(1) The note to play (i.e., `C`, `D`, `E`, `F`, `G`, `A`, `B`).
(2) An optional accidental (i.e., `b` or `f` for flats, and `s` or # for sharps).
(3) An optional octave number from `0`-`9`; if no octave is specified, then the octave will default to `4`.

For example, `Cs2` represents a C sharp on the second octave, `Bb` represents a B flat on the fourth octave, and `A5` represents an A natural on the fifth octave.

```
loop("p1", "arpy:Cs2 arpy:A5 arpy")
```

If no pitch is explicitly written (or if the pitch is invalid), then the pitch defaults to middle A (i.e., `A4`). For instance, in the code above, the third `arpy` note has no specified pitch and thus defaults to `A4`.

3.9 Tempo

The tempo for MiniClef can be queried via the `get_bpm` method, which will return the current beats per minute at which the music is being played. By default, MiniClef plays music at 135 beats per minute, which is a common tempo for techno music. I can set the tempo to be faster or slower by updating the beats per minute via the `set_bpm` method.

```
set_bpm(60)
```

3.10 Repeats

Thus far, I have only showcased the `loop` method, which plays music forever. However, what should I do if I only want to play a pattern once? In that case, I can use the `once` method.

```
once("p1", "arpy")
```

I can also play a pattern two or three times via the `twice` and `thrice` methods, respectively.

Finally, if I want to play a pattern a specific number of times, then I can use the `repeat` method, which takes in a number indicating how many times the pattern should be played, the pattern name, and the pattern string to be played.

```
repeat(7, "p1", "arpy")
```

The above code plays the `arpy` note exactly seven times.

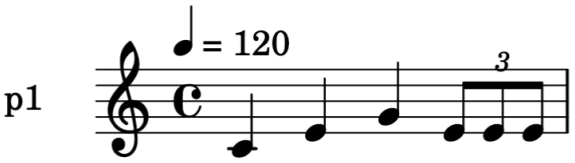

Figure 1: Simple Sheet Music

4 SHEET MUSIC GENERATION

I begin by describing the sheet music generation process in MiniClef and some of the nuances that come with it. Then, I also present many alternate designs that I considered and explain why I ultimately chose the design that I did. Finally, I give some examples of sheet music generation that could be very useful in an educational setting.

4.1 Setup

As mentioned briefly in Section 3.1, MiniClef requires LilyPond to be installed in order to generate sheet music. LilyPond is a music engraving program that produces high-quality sheet music from a text input file. I use the Python package `abjad` in order to interface with LilyPond and generate sheet music.

4.2 Generating Sheet Music

Whenever I play a pattern, MiniClef will automatically generate sheet music for that pattern in a file called `sheet_music.pdf`; the corresponding LilyPond file is also generated in `sheet_music.ly`. The sheet music will show the notes that are being played, the rhythm of the notes, and the pitch of the notes. The sheet music will also show the tempo at which the music is being played.

```
loop("p1", "a:C a:E a:G [a:E a:E a:E]")
```

Figure 1 displays the sheet music generated for the code above; note that I have replaced the synths with `a` for simplicity. The sheet music shows the pattern name, the tempo, and the notes that are being played.

Sheet music is regenerated every beat so that it is always up-to-date with the music that is being played. This allows users to see the sheet music change in real-time as the music plays.

If I am playing multiple patterns at the same time, then the sheet music will show all of the patterns that are being played. Each pattern will be displayed on a

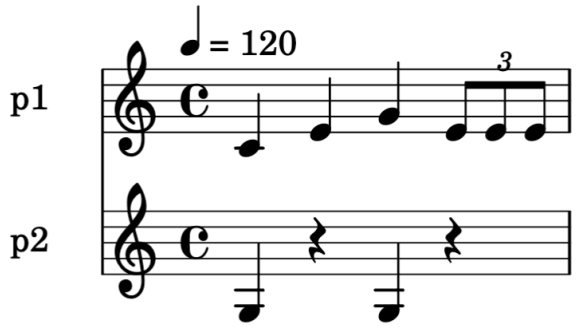

Figure 2: Two Pattern Sheet Music

separate staff, and the staves will be stacked on top of each other in the same score. This allows users to see how the different patterns interact with each other and how they combine to create the resulting music.

```
loop("p1", "a:C a:E a:G [a:E a:E a:E]")
loop("p2", "a:G3 ~ a:G3 ~")
```

Figure 2 displays the sheet music generated for the code above; again, I have replaced the synths with `a` for simplicity. The sheet music shows two patterns, each on its own staff: one for the `p1` pattern and one for the `p2` pattern.

4.2.1 Alternate Design. I considered an alternate design where each staff was not an individual pattern but rather an individual instrument.

```
loop("p1", "arpy bass pluck")
```

For example, in the code above, the alternate design would show three staves: one for the `arpy` pattern, one for the `bass` pattern, and one for the `pluck` pattern. This design would allow users to see the music for each instrument separately.

However, I ultimately decided against this design because MiniClef is primarily pattern-based, so it makes more sense to map not each instrument but rather each pattern to a staff. A single pattern often utilizes multiple instruments, so separating them into different staves would be confusing and simply consist of many short and uninformative staves.

4.3 Cycles

As previously mentioned, MiniClef will generate one staff for each pattern that is being played. But, what happens if a pattern contains a cycle? Should I show all of the notes in the cycle on the sheet music?

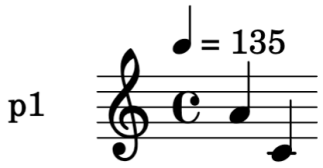

Figure 3: Option 1 Sheet Music

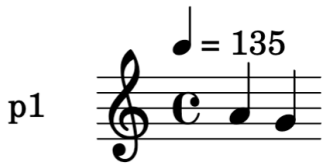

Figure 4: Option 2 Sheet Music

I opt to not display all notes in a cycle but rather display just the note that will immediately be played next. This means that the note that is displayed will change over time as the pattern is played. This way, users can confidently look at the sheet music and deterministically know what music will be played.

```
loop("p1", "arpy <arpy:C arpy:G>")
```

In the code above, the second note in the pattern alternates between C and G. The sheet music that is generated depends on which note is to be played next. Figure 3 displays the sheet music generated when C is to be played next, while Figure 4 displays the sheet music generated when G is to be played next.

4.3.1 Alternate Design. I also considered an alternate design where all notes in a cycle are displayed on the sheet music, each in a different color. This would allow users to see a complete picture of all notes that could be played.

However, I ultimately decided against this design because it would be too cluttered and confusing. The sheet music would be difficult to read and understand, and it would be hard to see what music is actually being played. Additionally, multiple cycles in the same beat would be difficult to display, as the sheet music would show the same color for multiple notes; this would make it challenging to distinguish between different cycles. Instead, by only displaying the next note in the cycle, the sheet music is much cleaner and easier to read.

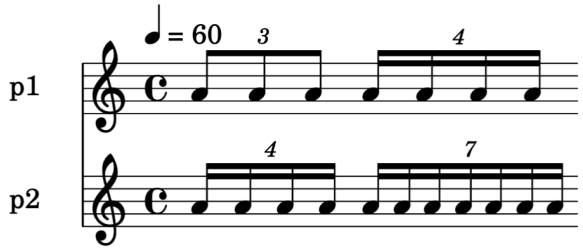

Figure 5: Polyrhythm Sheet Music

4.4 Randomness

Like cycles, randomness also presents a challenge for sheet music generation. Should I display all possible notes that could be played, or should I display just one note that will be played?

I decide to just display the one note that will be played. That is, every time a random note is played, I select a new random note to be played the next iteration; this note is then displayed on the sheet music. This way, users can again know with full confidence exactly what music will be played.

```
loop("p1", "arpy {arpy:C arpy:G}")
```

In the code above, the second note in the pattern is randomly chosen to be either C and G, each with 1/2 probability. The sheet music that is generated depends on which note is randomly chosen to be played next. Figure 3 displays the sheet music generated when C is to be played next, while Figure 4 displays the sheet music generated when G is to be played next.

4.4.1 Alternate Design. I could have also displayed all possible notes that could be played, each in a different color. This would allow users to see all of the possible notes that could be played, but I ultimately decided against this design for the same reasons as with cycles. The sheet music would be too cluttered and difficult to read, and it would be hard to see what music is actually being played. By only displaying the next note that will be played, the sheet music is much more clean.

4.5 Examples

I want to highlight some examples of sheet music generation that could be very useful in an educational setting.

4.5.1 Polyrhythms. Polyrhythms are often a difficult concept for students to understand, but they are an important part of music theory. By generating sheet music for polyrhythms, students can visualize how different rhythms should overlap and how polyrhythms can be constructed.

```
loop("p1", "[a a a] [a a a a]")
loop("p2", "[a a a a] [a a a a a a a]")
```

Figure 5 displays the sheet music generated for the code above; note that I have replaced the synths with a for simplicity. The first beat plays a 3:4 polyrhythm, while the second beat plays a 4:7 polyrhythm.

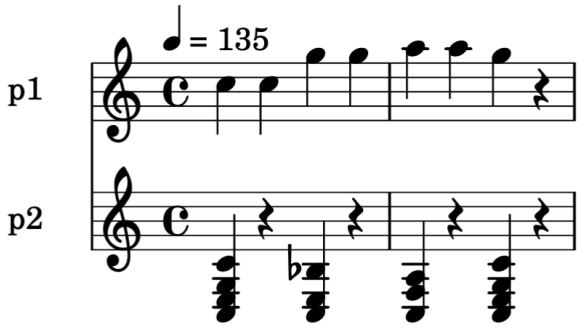

Figure 6: Melody and Harmony Sheet Music

4.5.2 Melody and Harmony. Melody and harmony are also important concepts in music theory. I can generate sheet music where one pattern plays a melody and another pattern plays a harmony. This allows students to see how melody and harmony should work in conjunction with each other.

```
loop("p1", "a:C5 a:C5 a:G5 a:G5
            a:A5 a:A5 a:G5 ~")
loop("p2", "(a:C3 a:E3 a:G3 a:C) ~
            (a:C3 a:E3 a:Bb3) ~
            (a:C3 a:F3 a:A3) ~
            (a:C3 a:E3 a:G3 a:C) ~")
```

Figure 6 displays the sheet music generated for the code above; again, the synths are replaced with a for simplicity. The first pattern plays a melody, while the second pattern plays a harmony.

5 FUTURE WORK

In the future, I would like to expand MiniClef even further by including more synths and adding support for samples. While the current set of synths suffices for demonstration purposes, real-world compositions

might require more instrumental diversity. Additionally, I would want to introduce samples to MiniClef as well. Sampling is a fairly prominent aspect of live coding that is supported by popular languages such as TidalCycles and Sonic Pi, and adding samples to MiniClef would provide users with many more interesting music options.

Moreover, I want to give users the ability to manipulate and alter the sounds that they produce through properties and effects. For example, adding an ADSR envelope as a `Pattern` property would allow users to control the timing of certain sounds, and adding an amplitude property would give users the ability to control the volume of their sounds. This would involve overhauling all of the synth definitions to support these new properties, but this trade-off is well-worth for the added customizability. On a similar note, effects such as striate or vibrato could transform existing music into completely new sounds. They would add another dimension to the music and allow users to produce more unique compositions.

Finally, I would like to explore more alternate designs with sheet music generation. While I have already carefully considered many possible options and believe that I have made a sound decision in implementing sheet music generation as I have, there are always more possibilities to explore. For example, I wonder if there are better ways to showcase cycles or randomness on the sheet music. Additionally, there is currently no text for instruments on the sheet music, and I would like to explore different approaches to displaying this information, too. Ultimately, I want to ensure that the sheet music is as informative and helpful as possible for users.

6 CONCLUSION

Live coding is a popular form of performing arts and is known for its ability to generate unique sounds and music in real-time. Through the use of programming, artists can create complex compositions and experiment with different sounds and rhythms. Live coding can even be useful in an educational setting, as it can help link music theory and computer science together. However, live coding is often limited to audio, and there is a lack of support for sheet music generation. Sheet music is often an important tool for learning music theory and understanding how music is constructed. Thus, adding support for sheet music generation in live coding would greatly enhance its educational value.

In this paper, I propose MiniClef, a domain-specific language embedded in Python that generates sheet music according to the underlying source code. I first introduce MiniClef by example, showcasing its features and how it can be used to generate music. Then, I describe the sheet music generation process in MiniClef and highlight some examples that showcase its educational utility. Finally, I outline some future work that I would like to pursue in order to expand MiniClef even further.

Ultimately, I believe that MiniClef has the potential to be a powerful tool for live coding and music education. By combining live coding with sheet music generation, MiniClef can help bridge the gap between music theory and computer science and provide users with a unique and engaging way to learn music.

REFERENCES

[1] Samuel Aaron and Alan F. Blackwell. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design* (Boston, Massachusetts, USA) *(FARM '13)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/2505341.2505346

[2] Ryan Kirkbride. 2016. FoxDot: Live Coding with Python and SuperCollider. In *Proceedings of the International Conference of Live Interfaces*. 194–198. https://users.sussex.ac.uk/~thm21/ICLI_proceedings/2016/Colloquium/68_FoxDot.pdf

[3] Thor Magnusson and Alex McLean. 2018. Performing with Patterns of Time. In *The Oxford Handbook of Algorithmic Music*. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190226992.013.21

[4] James McCartney. 1996. SuperCollider, a New Real Time Synthesis Language. In *International Conference on Mathematics and Computing*. https://api.semanticscholar.org/CorpusID:5976007