Background Report: Loopy, A Domain Specific Language for Electronic Dance Music Creation

Edward Cai c4n0b@ugrad.cs.ubc.ca
Kwang Hee Park m8b9@ugrad.cs.ubc.ca
Shobhit Bhatia j9k0b@ugrad.cs.ubc.ca
Ro Lee n3l0b@ugrad.cs.ubc.ca
Benjamin Willox s2g1b@ugrad.cs.ubc.ca

ABSTRACT

Our proposal is a domain specific language, called Loopy, built for EDM (electronic dance music) creators. Loopy will be designed such that it provides solutions to the repetitive nature of EDM, such as loops with various notes. This report will go through an introduction on the background and the rationale of Loopy and explain the technical approaches in building it through going in detail of the concrete, abstract syntaxes and the interpretation.

1. OVERVIEW

We will focus on the value of Loopy as an alternative music creation language primarily for EDM artists. We will illustrate the value by providing a brief overview of the EDM creation market including a look at competing languages. We will examine the features that our language has that will be of particular value to programmers.

Our project will create a domain specific language called Loopy. We will create and annotate a concrete syntax that is easy to understand for the end programmer, an abstract syntax that will be interpreted into a song, and an interpreter that can provide the necessary framework for doing this. We will explore various design choices such as the forms that our concrete syntax can take, various features of the `RSound` library and the lexical scoping of our language. The repetitive and harmonic nature of EDM will influence the development of our language.

This report will examine the technical aspect of our language, provide sample programs that may be turned into music and illustrate why our DSL is worth creating.

2. INTRODUCTION

EDM (electronic dance music) has been around historically since the 1960s, but only ever reached popularity here in North America around the early 2010s after American music industries pushed to rebrand rave culture with EDM[1]. Artists such as Hardwell, Skillrex, and Steve Aoki hit millions of views as EDM rose to popularity and eventually triggered Billboard to introduce a new EDM-focused Dance / Electronics Song Chart in 2013 [2].

At a time where EDM is expanding its territories and artists are experimenting tools to create music more efficiently, our team wanted to come up with a tool that can help artists easily create electronic dance music through programming. We, specifically, wanted to target the repetitive nature of EDM which is comprised of numerous loops and notes that elevate and diminish in both volume and pitch.

3. VALUE PROPOSITION

3.1 The Target Market

Our target market is specific: anyone who produces EDM or applications thereof that finds text-based approach more efficient than a visual patch-based approach.

One can ask does such market exist, and, yes, indeed there is. The history of text-based music creation goes back to 1957 when Max Matthews first created MUSIC[3], the first computer program to generate digital audio waveforms through direct synthesis. This spawned a whole family of computer programs and programming languages dedicated specifically towards creating text-based music creation.

The question still looms however: Are there any text-based music creation approach specifically geared towards fast, efficient EDM creation?

3.2 Alternatives

There are a couple of major DSP (short for digital signal processing) programming languages out there currently: Csound, pyo and SuperCollider.

All of the mentioned programming languages support a vast amount of music creation features. Csound, for example, has abilities to define interactions with various instruments and hardware specifically inside XML tags and has a separate header section with predefined keywords to set sampling rates, number of audio channels, and decibels[4].

The problem is that there's too much; what we want is simplicity. The learning curve should not be high since we are aiming for fast and efficient music creation.

4. TECHNICAL ASPECTS

4.1 Concrete Syntax

The concrete syntax of a language provides the set of rules that define the way programs look to the programmers. When we were creating our concrete syntax our goal was twofold; firstly to provide an easily understandable syntax and secondly to efficiently represent the functionality of our language. We decided upon the following:

```
<L-Expr> ::=
| {<num> <L-Expr>}
| {note <num> <num> <num>}
| {loop <L-Expr>* <num> <num> <num>}
| {segment <L-Expr>* <num>}
```

The first form is the speed modifier. It takes a number and a musical expression and modifies the speed of that musical expression by whatever the number given is. For example the code shown below will play the expression `{loop {note 50 1 1} 1 8 3}` twice as fast.

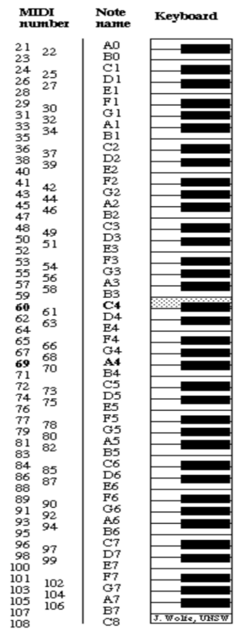
```
{2 {loop {note 50 1 1} 1 8 3}}
```

By twice as fast, we actually mean doubling the BPM. For default, we will go with 128 BPM since that is the average EDM tempo[5], however by providing a speed modifier, artists can freely change the BPM to whichever speed they prefer. The design as to why give it a multiplier compared to actually providing the exact BPM number is that usually in EDM, the tension before the "drop", a moment of instrumental build when the bass and rhythm hit hardest[6] , has a slowly increasing speed. Instead of stating actual BPMs, our team thought that giving multipliers, for example, 1.0, 1.1, 1.2, 1.3, etc. would be a more efficient and intuitive way to state speeds.

The second form is the basis of all songs the user can create. It represents a synthetic musical note.

```
{note <num> <num> <num>}
```

The first parameter is the musical instrument digital interface or MIDI number of the note to be played. We decided to use MIDI note numbers rather than conventional musical notation since it is already widely used in the aforementioned alternative



[Figure 1]

programming languages. *Figure 1* shows a conversion of a typical note and octave into the corresponding MIDI number[7]. The second parameter is the bar in which the note will start playing, and naturally the third number is the duration in bars. The start bar and duration values signify the relative positions that the note takes within a loop.

EDM is repetitive and this is why our third form is important. It is a musical expression that represents a `loop`.

```
{loop <LExpr>* <num> <num> <num>}
```

The parameters that a `loop` can take are first, a list of `LExpr` values to be looped, second, a number that specifies the start bar, third, the total length of the loop in bars, and last, the number that signifies the amount of iterations that the `loop` will take. This will allow artists to repeat sounds an arbitrary amount of times.

The fourth and final form that our musical expression can take is a Segment.

```
{segment <LExpr>* <num>}
```

A segment takes two parameters: a list of `LExpr` values to be played and a number representing the number of total bars of the segment. The main goal of the `segment` is to provide structure. Structure in the sense that users can see how a series of notes and loops creates certain melodies or sections of their musical piece. And by providing the total number of bars, users can ensure that the loops defined within cannot accidentally go over the length that they desired for a `segment`.

So, here's an example:

```
{segment
      {loop
            {note 50 1 4}
            {note 45 1 2}
            {note 30 2 2}
      1 4 4}
16}
```

The example is a segment that runs for 16 bars with a loop. The loop itself starts at the first bar, runs for 4 bars, and repeats itself 4 times. The first note will be playing a #50 MIDI note from the first bar for 4 bars. The second note will be playing a #45 MIDI note from the first bar for 2 bars. Finally, the last note will be playing a #30 MIDI note from the second bar for 2 bars.

This aligns with our goal of creating a language that artists will find useful.

4.2 Lexical Scope

Before we move into abstract syntax, let's go over the scope of our syntax.

In a lexically-scoped language, the value of any identifier can be determined by looking at the program's concrete syntax. Although our DSL does not contain identifiers, it does contain music-speed changing code that can be passed in by the programmer. We felt the need to include a section on lexical scoping since our language is evaluated linearly from left to right but speed changing code can appear anywhere. We wanted to make this as straightforward as possible for the end programmer so we implemented Loopy such that, parameter changing code effects "anything that comes after it".

This can be visualized by picturing these speed changing parameters as being applied to everything they form a closure around. This gives us our lexical scoping; we can tell which parts of the code are changed by simply inspecting a program. Loopy was initially not going to contain lexical scoping but we believe that it is an important functionality for EDM artists to be able to modify specific parts of their music.

```
{2 {note 50 1 2}}
```

The above code demonstrates the lexical scoping in effect. The speed increases by a factor of 2 for all the notes enclosed in its closure.

The code below demonstrates an issue that arises if lexical scope was not supported. The code below plays two notes; one at regular speed and one at double speed. Without lexical scope both these notes could be played at double speed and the end programmer loses some control.

```
{segment
      {note 50 1 1}
      {2 {note 50 1 1}}
}
```

4.3 Abstract Syntax

Below is our abstract syntax:

```
(define-type L-Expr
   [modify-speed (multiplier number?)
                 (expr L-Expr?)]
   [note (mini-num number?)
         (start-bar number?)
         (duration number?)]
   [loop (comps (listof L-Expr?))
         (start-bar number?)
         (duration number?)
         (iteration number?)]
   [segment (comps (listof L-Expr?))
            (total-bars number?)]
)
```

As you can see, there are no huge differences compared to the concrete syntax and thus is not that interesting. The parser will be a trivial case and thus we will not go over it.

4.4 RSound

Before we go into the interpretation stage, let us briefly explain about the RSound, a crucial package for our implementation of Loopy.

RSound[8] is a Racket sound engine that is described by its creators as "an adequate sound engine". It includes procedures for sound control, stream based playback, recording, sound I/O, and network based playback. Our discussion will focus mainly on `rsound` manipulation and single-cycle sounds because they are core to our interpreter and thus our DSL.

Here are some descriptions of the core procedures that we will be using:

```
FRAME-RATE : nonnegative-integer?
```

This is the basic default frame rate. It is set to 44,100 Hz.

```
(play rsound) -> void?
  rsound : rsound?
```

The `play` procedure takes an `rsound` and plays it through the computer's audio output device. Our DSL will use `play` as the final value of our interpreter; the key step in turning all the `rsound` values that we generate into tangible music.

```
(resample factor sound) -> rsound
  factor : positive-real?
  sound : rsound?
```

The `resample` procedure will be used to change the speed. The `factor` parameter is used to change the length of the sound while retaining the frame rate, resulting in a sped up or slowed down sound.

```
(synth-note family
            spec
            midi-note-number
            duration) -> rsound
  family : string?
  spec : number-or-path?
  midi-note-number : natural?
  duration : natural?
```

The `synth-note` procedure is the basis of our note and provides an EDM like synthetic note that can be modified based on parameters. For default we will resort to the "main" family and use the first case of it defined by the spec parameter. The `midi-note-number` refers to the actual note that will be played and, of course, the `duration` in frames.

```
(assemble assembly-list) -> rsound?
  assembly-list :
  (listof (list/c
    rsound?
    nonnegative-integer?))
```

This is a crucial procedure which is used to assemble all of the recursively processed sounds with each sound starting at an offset.

```
(rs-append rsound-1
           rsound-2) -> rsound?
  rsound-1 : rsound?
  rsound-2 : rsound?
```

This procedure will be used in our loop implementation to concatenate sequentially the processed `rsound` objects.

```
(silence frames) -> rsound?
  frames : nonnegative-integer?
```

This procedure returns an empty silent sound for the given frame. It will be used to prepend a buffer to a given `rsound` that does not start at bar 1 (which is always the starting bar).

```
(clip rsound start finish) -> rsound?
  rsound : rsound?
  start : nonnegative-integer?
  finish : nonnegative-integer?
```

The `clip` procedure is used, as its name states, to clip the `rsound` from the start to finish frame. This will be used by our segment case to enforce structure; not let loops or notes go on forever by mistake.

4.5 Interpretation

Knowing that we will be utilizing `RSound`, we will now go over the implementation of our interpretation of the abstract syntax for each aspect. We will provide the implementation of each case, modify-speed, note, loop, and segment.

4.4.1 Speed Modification

For `modify-speed`, there are two parameters: the multiplier and the expression. Since the expression will be interpreted recursively by the interpreter, all we have to do here is to give instructions to resample the `rsound` with new frames. For example, let's say that the multiplier given was 1.5, then we would use the following code:

```
(resample 1.5 stub_sound)
```

This would give back a new sound that is 1.5 times long than the original sound. The sample rate will stay the same due to the semantics of the resample procedure which does not affect our implementation since we are only looking to affect speed.

4.4.2 Note

The note will be created with the aforementioned synth-note and the assemble procedures. First, synth-note will be created:

```
(synth-note
  "main"
  1
  mini-num
  (* FRAME-RATE duration))
```

This means that the `rsound` created gets our default sound (1st case of the main family), plays it at the MIDI note given for the given duration. The duration here has to be multiplied to the default `FRAME-RATE` constant provided in the `rsound` package to make the sound last for the time interval we intended. The default frame rate is set to 44,100 Hz which is equivalent to one second in our context.

This then will be fed to the assemble procedure with a silent sound because we need to fill in the "gaps" in the front of the `rsound` so that later when it is processed by either the loop or segment that the sounds correctly overlap. Here is a clarification:

```
{loop
  {note 40 1 4}
  {note 50 2 3}
1 4 4}
```

Suppose in the example above that the two notes were created simply with `synth-note` and passed back to the loop without the assemble with empty sound stage. Then, the second sound will have no buffer for the first bar so the two notes will play simultaneously instead of the first note starting on the first bar and the second note starting on the second bar. So, basically, the assemble stage is to create a buffer for the notes that have starting bars not at 1.

For the empty sounds we will use a procedure called silence, which just returns frames of silence sounds. An example of an assemble for `{note 40 2 3}` would be as followed:

```
[note (mini-num start-bar dur)
  (define buffer
     (* (- dur start-bar)
         FRAME-RATE))

  (assemble
    (list
      (list (silence buffer) 0)
      (list
        (synth-note
          "main"
          1
          mini-num
          (* FRAME-RATE dur))
       buffer)))]
```

4.4.3 Loop

Loops will have a similar approach to notes, the major differences would be handling of the types of `L-Expr` passed in and the iteration logic.

Loops will recursively assemble all of the components being passed in:

i) Type is `note`:

For notes, we do not have to do more buffer handling since it is already handled in its own interpretation. We just simply need to assemble it from the start frame of 0.

ii) Type is `loop`:

Now, loops have a start-bar and a duration parameter. This might seem non-trivial to handle but it is actually simple. All we need to do is calculate the starting buffer due to the start-bar, assemble a new `rsound` with the silence buffer and the recursively processed loop so that the actual `rsound` that is passed into the "upper" loop is a sound that can be appended to the assemble parameter list with a starting frame of 0.

iii) Type is `segment`:

Segments do not have a starting bar, so just append to the assemble parameter list with a starting frame of 0.

Now when the main assemble procedure has produced a new `rsound` with all the given `L-Expr`, recursively `rs-append` it to itself with a counter that equals the iteration specified for the loop.

```
[loop (exprs sbar dur iter)
  ; Assume processed returns a
  ; recursively processed rsound
  ; of all exprs, and that buffer
  ; handles similarly to the
  ; implementation in the note case.
  (local [(define processed exprs)]
         [(define buffer …)
         [(define loopSound
            (assemble (list
            (list (silence buffer) 0)
            (list processed buffer)))
         )]
         [(define (rec-append s acc
            (if0 acc
              s
              (rec-append
                (rs-append s s)
                (sub1 acc))))])

  (rec-append loopSound iter)]
```

4.4.4 Segment

The segment is simply a structure that defines the length of whichever `L-Expr` values it was given. In a way, we can say that it "clips" the length since loops and notes can technically go on forever without a limit. So, the solution here is to first assemble all the recursively processed `L-`

`Expr` values and then simply clip the result. Here is an example:

```
[segment (exprs total-length)
  ; Assume processed returns a
  ; recursively processed rsound
  ; of all exprs.
  (local [(define processed exprs)])

  (clip processed
        0
        (* FRAME-RATE total-length))]
```

After all the interpretation of the passed in expressions are done, we will simply pass it into the play procedure to play the processed sound.

5. PROJECT POTENTIAL

5.1 Initial Implementation

To achieve this milestone, we must first create a basic parser and interpreter. At this step we plan to lay out the outline for the language's design, breaking it down into specific tasks that we can fulfill.

To complete the proof-of-concept, we plan to build a basic parser and interpreter that implements the necessary components of our language: Notes and Loops. This parser will allow the user to build relatively simple EDM tracks but will more importantly demonstrate that our final language is possible and useful. We chose to only include notes and loops in this milestone we feel that they are sufficient to show the underlying parser and interpreter are working to interpret our DSL.

5.2 Final Project

To complete our final project we will be expanding our parser and interpreter to implement speed changing parameters and segments. This will take our DSL from being able to build simple beats to being able to build complex electronica that has structured components.

The main hurdle that we will have to overcome in finishing this project is the interpretation and scoping of speed changing parameters. We have done some research into possible implementation techniques and have decided that working with the conversion factor between frames and seconds is the best choice. Since a duration of one second is approximately 44,100 frames, we will work with an environment to keep a speed factor that we can use to modify the duration of notes in musical expressions that follow the parameter.

Our final project will also include a tutorial package that demonstrates how EDM artists can write music. It will include some recreation of popular songs that we think will serve as good examples.

6. SUMMARY

Our report has outlined the unique and useful features of Loopy that make it suitable for EDM development. By describing the concrete syntax, abstract syntax, parsing and interpretation, this report has demonstrated the technical aspects that our DSL will be built upon. More importantly this report has shown the thought we have put into each feature, so that our language will have true value to EDM artists.

REFERENCES

[1] Reynolds, Simon. "How Rave Music Conquered America." The Guardian, Guardian News and Media, 2 Aug. 2012, www.theguardian.com/music/2012/aug/02/how-rave-music-conquered-america.

[2] Pietroluongo, Silvio. "New Dance/Electronic Songs Chart Launches With Will.i.am & Britney at No. 1." Billboard, Billboard, 21 Jan. 2013, www.billboard.com/articles/news/1510640/new-danceelectronic-songs-chart-launches-with-william-britney-at-no-1.

[3] DuBois, R. Luke. "The First Computer Musician." The New York Times, The New York Times, 8 June 2011, opinionator.blogs.nytimes.com/2011/06/08/the-first-computer-musician/.

[4] The Canonical CSound Reference Manual, http://www.csounds.com/manual/html/while.html http://newt.phys.unsw.edu.au/jw/notes.html

[5] “Why Is House Music 128 BPM?” *EDM Nerd*, 1 Mar. 2015, www.edmnerd.com/house-music-128-bpm/.

[6] Yenigun, Sam "The 5 Deadliest Drops Of 2010". *www.npr.org*. NPR, 31 December 2010

[7] What Is a Sound Spectrum?, newt.phys.unsw.edu.au/jw/notes.html.

[8] RSound Documentation https://docs.racket-lang.org/rsound/index.html